TuskFlow: An Efficient Graph Database for Long-Running Transactions [Industry]

Georgios Theodorakis
Neo4j
george.theodorakis@neo4j.com

Hugo Firth
Neo4j
hugo.firth@neo4j.com

James Clarkson
Neo4j
james.clarkson@neo4j.com

Natacha Crooks
UC Berkeley
ncrooks@berkeley.edu

Jim Webber
Neo4j
jim.webber@neo4j.com

ABSTRACT

Mammoth transactions, which involve long-running operations that access many items, are common in graph workloads. Graph analytics tasks, including pattern matching and graph algorithms, can generate large read-write operations that impact significant portions of data, which makes their execution challenging under strict isolation guarantees. Consequently, we face an apparent trade-off between ensuring high isolation and achieving high performance, forcing users to choose between the two.

In this work, we present TuskFlow, an experimental graph database based on Neo4j, designed to efficiently handle mammoth transactions on graphs (the technique is applicable to other models such as relational) while maintaining existing transactional semantics. TuskFlow employs a deterministic protocol that safely reorders regular transactions around mammoths within an epoch. Our protocol supports parallel mammoth execution inspired by graph-parallel algorithms. To minimize conflicts with regular transactions, TuskFlow introduces query- and workload-aware optimizations, including graph entity tagging and partitioning. Our experiments demonstrate that, unlike traditional protocols like two-phase locking or MVCC, TuskFlow avoids blocking write transactions and improves tail latency by up to 45×.

PVLDB Reference Format:
Georgios Theodorakis, Hugo Firth, James Clarkson, Natacha Crooks, and Jim Webber. TuskFlow: An Efficient Graph Database for Long-Running Transactions [Industry]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Neo4jResearch/TuskFlow.

1 INTRODUCTION

Commercial graph database management systems (DBMSs) [2, 47, 49, 69] are rapidly growing in popularity, with the graph database market projected to reach \$5.1 billion by 2026 [26]. These systems

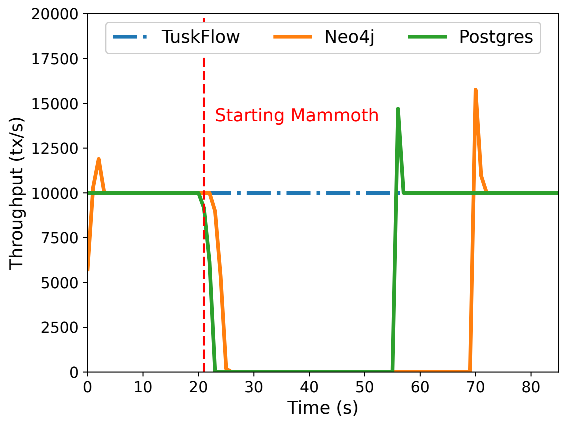

Figure 1: Existing systems cannot handle mammoth txs

allow users to model complex, highly associative data with high-fidelity as nodes and relationships. Common use cases include route planning [77], social networks [9, 75], and retrieval-augmented generation (RAG) for AI applications [43]. While these systems can handle trillion-scale graphs with high transactional performance [50, 70], long-running read-write transactions, known as *mammoths* [18], can severely impact overall execution.

Very large transactions can challenge any DBMS. In traditional workloads, the problem of handling mammoths remains only partially solved [18]. Separating workloads into OLAP (long-running, read-only transactions supported by MVCC) or OLTP (short and write-heavy) [39] allows short-lived transactions to make good progress but at the cost of increased operational complexity and staleness for data analytics. The move to HTAP systems [6, 7, 15, 70] allows some consolidation where read-only analytical and short read-write transactions can co-exist.

Mammoths are particularly prevalent in graph databases, where tasks like graph pattern matching [34], graph algorithms such as community detection and degree centrality [48], cascading deletes, and schema changes [21, 58] are routine but can conflict with other operations. These tasks read from and often write to large parts of the graph, blocking the progress of many short-lived transactions and significantly degrading system performance.

To illustrate the widespread nature of the problem, Fig. 1 compares the throughput of Neo4j (using two-phase locking, or 2PL [11]) and Postgres (using MVCC [10]) under a workload of 100 clients

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

performing consecutive read or write transactions using a random relationship on the WikiTalk dataset (see Table 1). When one mammoth transaction begins to update a property present on all relationships (marked by the dashed vertical line), both systems show zero throughput, as all write transactions are blocked. In addition, the tail latency for short-lived transactions spikes to over 30 s, orders of magnitude higher than the expected (millisecond latency). For larger graphs, the problem worsens as the queuing of conflicting short-lived transactions can lead to time-outs. Moreover, mammoth transactions can cause page cache pollution, further increasing the latency of the blocked transactions.

Vanilla optimistic protocol implementations always abort the mammoth when attempting to commit due to conflicts with short-lived writes [18], meaning that mammoths will always fail. Pessimistic protocols like 2PL block all transactions, and deterministic protocols were not designed for handling such workloads [44].

An alternative to the limitations of existing concurrency protocols would be to forgo strong isolation guarantees and transactional semantics. For example, Neo4j's and MemGraph's graph algorithm libraries [47, 53] allow users to run expensive algorithms without isolation guarantees when writing results back to the database. This forces users to either manually maintain consistency or accept weaker guarantees, which has numerous downsides [76].

Based on our experience at Neo4j, mammoth transactions are frequent enough in graph use cases to motivate research into a solution, and our working hypothesis is that both graphs and their workloads can be part of that solution. Our goal is to design and implement a transactional graph DBMS that addresses the challenges posed by workloads containing mammoth transactions. Specifically, we aim to: (i) ensure mammoths complete successfully; (ii) prevent mammoths from blocking short-lived transactions whenever possible; (iii) maintain high throughput and low latency for short-lived transactions; and (iv) provide strong transactional guarantees (e.g., serializability). Our contributions are:

(i) Deterministic concurrency protocol. We introduce the first deterministic protocol designed to handle mammoth transactions that allows short-lived transactions to be reordered before or after mammoths' effects rather than blocking them for the entire duration. By extending the Aria protocol [44], we manage transactions without prior knowledge of their read/write sets while ensuring conflict serializability (i.e., equivalent to running serially).

(ii) Query- and workload-aware optimizations. To accelerate conflict resolution for short-lived transactions, we introduce graph entity tagging and graph partitioning. By tagging graph entities and labels that a mammoth transaction will access before it executes, we can reduce unnecessary conflicts. Additionally, partitioning the graph into "hot" and "cold" communities based on the most popular nodes and access patterns accelerates mammoth execution and minimizes conflicts with short-lived transactions.

(iii) Pluggable and parallel mammoth execution. We integrate our protocol into the transaction scheduling module of a graph database to allow the parallel execution of mammoths, mirroring how graph-parallel algorithms are implemented. Here, we note that the protocol can be adapted to relational databases by substituting nodes and relationships with tables and graph traversals with joins.

Building on these ideas, we developed TuskFlow, an experimental in-memory transactional graph database based on Neo4j, designed to handle mammoth transactions efficiently without sacrificing correctness or performance. Our experiments demonstrate that TuskFlow improves throughput and tail latency for short-lived transactions by an order of magnitude compared to traditional methods like 2PL or MVCC, as shown in Fig. 1. Furthermore, our micro-benchmarks confirm that graph tagging and partitioning techniques both reduce conflicts and lower tail latency by 3.5 to 11.5× in the presence of mammoth transactions.

The remainder of the paper is structured as follows: we begin with a short survey of state-of-the-art graph DBMSs, transaction protocols, graph partitioning algorithms, and mammoth use cases (Sec. 2). We then introduce our deterministic concurrency protocol (Sec. 3) and conflict resolution techniques (Sec. 4). TuskFlow's design is discussed in Sec. 5, followed by our evaluation (Sec. 6). We finish with related work (Sec. 7), and conclusions (Sec. 8).

2 BACKGROUND

This section covers the foundational concepts of graphs, transactions, and mammoths relevant to the rest of the paper.

2.1 Graph Data Model

In this work, we adopt the labeled property graph (LPG) model [59], in which nodes may have optional labels and are connected by named, directed relationships. Both nodes and relationships can store key-value pairs, known as properties. Similar to Cypher [34], we define an LPG as a tuple $G = \langle N, R, src, tgt, \iota, \lambda, \tau \rangle$, where N is the set of nodes n, and R is the set of relationships r. The function src maps relationships to source nodes, while tgt maps relationships to target nodes. ι is a finite partial function that assigns properties to nodes or relationships, λ maps nodes to a finite (or empty) set of labels, and τ assigns a relationship type to each relationship. Nodes, relationships, and property keys have unique identifiers. Property values can be strings, primitive data types, or arrays, while labels and relationship types are represented as strings.

2.2 Graph Databases and Analytics

MemGraph [47], Neo4j [49], Neptune [2], and TigerGraph [69] are graph databases designed to support OLTP workloads over LPGs. They typically perform well for small transaction sizes and offer Read Committed or Snapshot Isolation. By contrast, analytical graph engines like Pregel [46], Giraph [19], and GraphX [78] are optimized for long-running OLAP queries on (predominantly) static graphs and do not support writes.

To bridge the operational gap between OLTP and OLAP workloads, graph algorithm libraries built on top of graph DBMSs [47, 53] permit writing the results of long-running computations (over graph snapshots) back to the database. However, this approach sacrifices transactional guarantees, potentially compromising the correctness of applications.

Ideally, these computations would run within mammoth transactions, but current systems struggle to maintain good performance at practical isolation levels, making such integration challenging.

```
MATCH (n: Message)-[r]-()
WITH n, COUNT(r) AS degreeCentrality
SET n.degreeCentrality = degreeCentrality
```

Figure 2: Compute degree centrality with Cypher (LDBC Q0)

2.3 Transaction Protocols

Transaction decomposition and specialized protocols for mammoths. One approach to managing long-running transactions [37] is to divide them into smaller units. With nested transactions [22, 45], systems can independently commit or abort these units, known as subtransactions. Decomposition of this nature can improve concurrency and reduce retries by utilizing savepoints [57], but it can also perform poorly under high load [20]. Sagas [35] also decompose long-running transactions into smaller requests but require applications to provide compensating transactions for recovery, adding significant overhead for graph traversals whose updates may be large and complex. Finally, modern distributed databases use specialized protocols [21, 58] to manage mammoths, like schema changes or large-scale deletes. However, these protocols often rely on specific consistency models that do not apply to all transactions and may lack certain guarantees.

Deterministic databases [1, 66] provide serializable guarantees (i.e., the results of concurrent transactions are equivalent to those of a serial execution) and ensure that the re-execution of a set of transactions will always yield the same database state. Calvin [67] achieves this through a two-phase process: first, it locks data and analyzes dependencies for conflicts based on a predetermined order; then, transactions are executed in parallel. Aria [44], another deterministic protocol, also employs two-phase execution but does not require pre-defined read-write sets. In Aria, transactions reserve reads and writes concurrently in the first phase, while conflicts are detected and commit decisions are made in the second phase, all without coordination. While these protocols prevent rollbacks, they suffer from significant performance degradation when handling long-running transactions [44], as other transactions are blocked until the mammoths complete within an epoch. They also incur higher latency due to synchronization between phases and stricter isolation levels than Read Committed or Snapshot Isolation.

2.4 Graph Partitioning Algorithms

Graph partitioning [55] aims to create k disjoint but complete graph partitions $P = P_1, ..., P_k$ that minimize a given cost function (e.g., edge-cut) while satisfying specific balancing constraints [40, 62, 73]. We focus on node partitioning [3], where each node $n \in N$ is assigned to a unique partition P_i. While existing systems use partitioning to reduce communication overhead in graph analytics, we apply this technique to resolve conflicts between mammoth and short-lived transactions. Specifically, we use query- and workload-aware techniques [32, 33], like pattern matching, for OLTP workloads. Our goal is to improve workload-sensitive partition stability [27, 28, 32] by reducing inter-partition traversals for a given workload.

2.5 Mammoth Use Cases

Let us introduce a set of typical use cases for mammoth transactions that set the context for the remainder of the paper. Following a prior

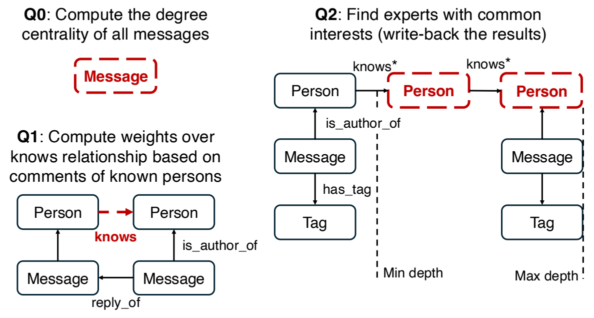
Q0: Compute the degree centrality of all messages
Message
Q1: Compute weights over knows relationship based on comments of known persons
Person
Person
knows
is_author_of
Message
Message
reply_of
Q2: Find experts with common interests (write-back the results)
knows*
knows*
Person
Person
Person
is_author_of
Message
Message
has_tag
Tag
Tag
Min depth
Max depth

Figure 3: Mammoth queries over LDBC social network

classification [18], mammoths fall into two categories: (i) balanced, with nearly equal read and write sets; and (ii) unbalanced, where reads significantly outweigh writes.

High-level graph query languages like Cypher [34] and GQL [29] abstract the complexity of graph operations and the high connectivity of graph structures such that seemingly simple queries might trigger mammoth transactions. For example, in Fig. 2, the Cypher query $Q0$ uses the `MATCH` clause to retrieve all nodes with the label `Message` and their relationships from the LDBC social network [30, 63]. It then counts these relationships for each node and updates a property called `degreeCentrality` using the `SET` clause. Although this query is not computationally intensive (since it uses metadata from the neighborhoods to access relationship counts quickly), it still blocks other transactions from performing writes. This problem is exacerbated by the presence of supernodes (nodes with many neighbors), as they further increase contention.

Fig. 3 illustrates two more examples of mammoth transactions over the social network, where the red dashed lines indicate graph entities being updated. Drawing from the Business Intelligence (BI) workload [63] of LDBC (OLAP queries), we adapted two existing operations to include writes to better reflect our experience with real mammoth transactions. In $Q1$, the weight of interactions between nodes labeled as Person is calculated and written back to the `knows` relationships. An interaction is defined as a direct `Comment` on a `Message` between two people. For the LBDC benchmark, this step is typically part of the pre-computation for BI-19 and BI-20 queries; for our mammoth use cases, we execute it alongside short-lived read-write transactions in Sec. 6.

$Q2$ is a variation of the BI-10 query, which identifies experts within a person's social network sharing common interests (represented by the `Tag` nodes linked to their `Messages`). The social clique is specified by a depth range. Our extension adds operations to update each identified expert at the transaction's end. Consequently, we classify $Q0$ and $Q1$ as balanced mammoths, while $Q2$ is unbalanced. All three mammoth queries are representative of real-world use cases encountered at Neo4j.

3 DETERMINISTIC EXECUTION FOR MAMMOTHS

Mammoth transactions, which perform read-write operations across a large portion of a graph, often conflict with short-lived transactions, causing them to abort or block. This issue occurs due to unpredictable access patterns, as nodes have varying numbers of relationships (e.g., power-law distribution). We believe, however, that we can do better than existing systems by monitoring and analyzing runtime contention. This approach would enable us to identify when concurrent operations can safely proceed in parallel with the mammoth without blocking. Intuitively, this involves checking that an operation does not compromise serializability. Specifically, we aim to ensure each short-lived transaction executes fully either before or after the mammoth in the serialization graph.

Assumptions. Our approach relies on several key assumptions: (i) the system runs only one mammoth at a time, alongside many short-lived transactions; (ii) mammoths may be slow without strict performance requirements, while short-lived transactions expect millisecond-level latency; (iii) mammoths should deterministically commit, meaning they do not abort due to integrity constraints; (iv) once a mammoth leaves a graph region, it cannot re-enter; (v) short-lived transactions see either previously committed changes by the mammoth or data it has not yet or will not ever access; and (vi) users specify if a transaction is a mammoth. The goal is to complete mammoths efficiently while ensuring high throughput, low latency, and transactional guarantees for regular transactions. Assumptions (i)-(iii) align with typical customer workloads, while (iv)-(vi) simplify the transactional protocol. We now provide its high-level overview.

Protocol overview. To facilitate dependency tracking in highly dynamic graphs, we extend deterministic protocols for mammoths. We chose Aria [44], which ensures conflict serializability without requiring the transactions' read-write sets in advance. Aria assigns each transaction a unique transaction identifier (TID) using a sequencing layer. TIDs are used to resolve conflicts deterministically.

All transactions are processed in parallel, in any order, over two phases within an epoch: local execution and commit. In the execution phase (Sec. 3.1), each transaction reads from the current database snapshot, makes reservations, and stores writes locally. In the commit phase (Sec. 3.2), transactions use the reservations to independently commit or abort based on TIDs. The transactions with Write-After-Write (WAW), Read-After-Write (RAW), or mammoth conflicts are aborted and rescheduled in the next epoch. Short-lived transactions cannot abort the mammoth.

To support mammoths without delaying epochs [44] or increasing latency for short-lived transactions, we split them into smaller tasks with a set number of operations per epoch (i.e., budget). These tasks access localized graph areas and the protocol reorders short-lived transactions around them – either before or after the mammoth – preserving dependencies to detect conflicts. As the mammoth traverses different graph regions, these regions transition through mammoth states (discussed in Sec. 3.2) until they are marked `VISITED`, ensuring they are not accessed again. This strategy significantly increases concurrency, allowing all transactions to make progress.

Algorithm 1: Execution and commit protocols

```
1  Function ExecuteTx(tx, db):
2      if tx.type == EXECUTE_LOCALLY then
3          RunLocally(tx, db)
4          tx.type ← RUN_ON_DB
5          Schedule tx for commit phase
6      else
           // Reschedule failed short-lived transactions and the
             mammoth if it has more work
7          if TryToRunOnDatabase(tx, db) == false or (tx.isMammoth and
             tx.hasFinished == false) then
8              tx.type ← EXECUTE_LOCALLY
9              Schedule tx for execution phase
10 Function RunLocally(tx, db):
11     Read from the latest db snapshot
12     Execute locally (mammoths use a budget) and reserve R/Ws in tx.RWSet
   // type is the record type and rid the record id
13 Function ReserveRead(tx, type, rid, db):
14     db.reservationTable[type].mergeRead(rid, tx.epoch, tx.tid)
15 Function ReserveWrite(tx, type, rid, db):
16     db.reservationTable[type].mergeWrite(rid, tx.epoch, tx.tid)
17 Function ReserveMammothRead(tx):
18     db.reservationTable[type].mergeMammothRead(rid, tx.epoch, tx.state)
19 Function ReserveMammothWrite(tx, type, rid, db):
20     db.reservationTable[type].mergeMammothWrite(rid, tx.epoch, tx.state)
21 Function TryToRunOnDatabase(tx, db):
22     if tx.isMammoth == true or HasConflicts(tx, db) == false then
23         Install tx.RWSet writes to db
24         return true
25     return false
26 Function HasConflicts(tx, db):
27     seenVisited ← false
28     seenUnvisited ← false
29     for entry ← tx.RWSet do
30         reservation ← db.reservationTable[entry.type].get(entry.rid)
31         if entry has WAW or budget and reservation.wid < tx.tid then
32             return true
           // Allow reading from visited and not updated regions
33         seenVisited |= reservation.state == VISITED and
             reservation.hasMammothWrite
34         isPending ← reservation.state == PENDING_WRITE or
             (reservation.state == PENDING_READ and entry.isRead == false)
           // Track if we have seen unvisited regions
35         seenUnvisited ← seenUnvisited or reservation.state ==
             UNVISITED
36         if (seenVisited and seenUnvisited) or isPending then  return true
37     return false
```

3.1 Local Execution Phase

During the local execution phase, the mammoth and short-lived transactions read from the same database snapshot at the epoch's start, as shown in line 11 of Alg. 1. This single-snapshot approach eliminates the need for multi-version storage, minimizing changes to the Neo4j codebase. After reading, transactions execute locally, tracking their read-write sets and reserving entries for reads and writes in a global reservation table (line 12). These reservations are crucial for deterministic conflict resolution in the commit phase.

To make a reservation, each transaction specifies the operation type (i.e., `read` or `write`), the record type, the unique record identifier, and the current epoch. For LPGs, record types include nodes, node labels, individual relationships, relationship types, and index entries; additionally, all relationships connected to a node are grouped as a unique record type (i.e., neighborhood). Short-lived transactions and mammoths make reservations slightly differently. Using `ReserveRead` and `ReserveWrite` (lines 13 and 15), short-lived transactions attempt to install reads and writes (tracked separately) in the global reservation table. The transaction with the smallest TID always secures a reservation regardless of the execution order. If a transaction fails to obtain a write reservation, it can skip the commit phase, as it has already encountered a WAW conflict, but must still install its reservations to maintain deterministic results.

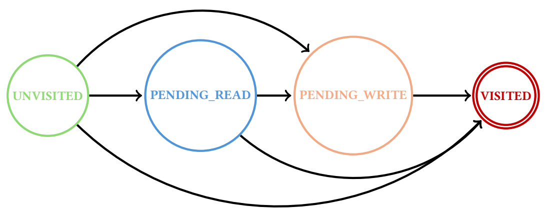

Figure 4: Mammoth protocol state machine

The single mammoth transaction also makes reservations (lines 17 and 19) but does not rely on its TID. Instead, it tracks the state of each record it accesses, indicating whether it still has work to do or has completed that record's read-write operations. The mammoth always installs its reservations independently of short-lived transactions. While we omit the pseudocode for brevity, during the execution phase, mammoths are broken down into smaller tasks based on a predefined epoch budget to prevent monopolizing the system. The budget refers to the number of accesses to various records (e.g., reading a label or updating a property) and allows mammoths to span multiple epochs without affecting other transactions' performance. We explain how it is calculated in Sec. 4.

Once all transactions complete the first phase, the commit phase begins. As shown in lines 4-5 of Alg. 1, workers schedule the transactions for commit after local execution.

For example, consider three transactions with the following read-write sets in epoch 1: (i) $TX1$ writes n_1; (ii) $TX2$ writes n_1 and reads n_2; (iii) mammoth tx reads nodes n_1 through n_3. After the first phase, the reservation table for the node records would have three entries: $\{n_1 : epoch = 1, wid = 1, state = \texttt{VISITED}\}$, $\{n_2 : epoch = 1, rid = 2, state = \texttt{VISITED}\}$, $\{n_3 : epoch = 1, state = \texttt{VISITED}\}$. The variables wid and rid show the earliest write and read TIDs excluding mammoth, while $state$ refers to the mammoth states discussed in the following section. In this case, $TX2$ did not install its write and can skip the commit phase.

3.2 Commit Phase

During the commit phase, each transaction independently makes a deterministic decision based on the reservations from the previous phase for a given epoch. If a short-lived transaction has no conflicts with other short-lived transactions or the mammoth, its changes are applied. If conflicts arise, the transaction is rescheduled to the beginning of the next epoch (lines 7-9) unless it violates an integrity constraint, in which case it is not retried. The order of aborted transactions is preserved, ensuring they will eventually commit because they have lower TIDs than transactions initiated after them.

For conflicts between short-lived transactions, if a transaction has WAW or RAW dependencies on an earlier transaction (one with a smaller TID), it must abort. No rollback is needed, however, as updates are stored locally in `tx.RWSet` and have not yet been applied. For conflicts with a mammoth, the short-lived transaction checks the mammoth's state for that record (e.g., whether the record has already been visited by the mammoth), as shown in lines 33-36.

Since the mammoth is broken down into smaller tasks to distribute its work across epochs based on a set budget, it does not process the entire graph in one pass. While the mammoth is running, records transition through distinct states shown in Fig. 4. Initially, each record is marked as `UNVISITED` and can transition to one of three states: (i) `PENDING_READ`, indicating the mammoth has read but not finished with the record; (ii) `PENDING_WRITE`, meaning the mammoth is still updating the record; or (iii) `VISITED`, indicating all operations on the record are complete.

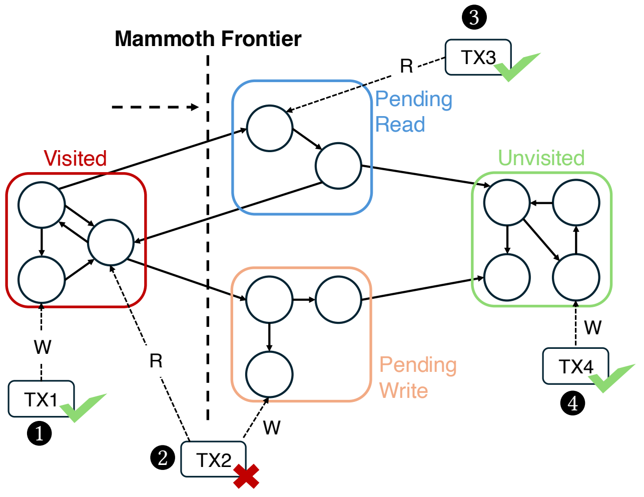

Figure 5: Mammoth conflicts with short-lived transactions

We want short-lived transactions to execute strictly before or after the mammoth's actions, preserving a clear order in the serialization graph. To ensure this property, we use the mammoth's four states to identify potential conflicts. Specifically, a short-lived transaction can either access fully `VISITED` records (line 33) or `UNVISITED` ones (line 35), but not both – except for `VISITED` records without writes. It also cannot interact with records in `PENDING_WRITE` state, though it can read records marked as `PENDING_READ` within an epoch.

Fig. 5 shows an example involving four transactions while a mammoth updates the graph, moving from the left side of the graph to the right (see the dashed line indicating its frontier). $TX1$ (❶) can write to the `VISITED` section of the graph, and $TX4$ (❹) can write to the `UNVISITED` part. $TX3$ (❸) can read from a `PENDING_READ` node but cannot modify it (e.g., properties or relationships), as updating requires reserving the node for a write, which would result in a conflict. $TX2$ (❷) conflicts with the mammoth because it attempts to operate on parts of the graph that the mammoth is currently processing (i.e., `PENDING`) and has already `VISITED`. The same would apply to both `VISITED` and `UNVISITED` sections.

Recall that only one mammoth is allowed at a time to ensure it always makes progress. The mammoth is guaranteed to commit its updates to the database (line 22) deterministically, meaning no validation step (e.g., constraint violations) or other transaction will abort it. At the end of the commit phase (line 7), the worker responsible for the mammoth checks for any remaining work and, if so, schedules it for the next epoch. The mammoth uses local data structures to track its execution progress, allowing it to pause and resume similar to suspendable tasks [36].

3.3 Determinism and Serializability

Compared to the Aria protocol [44], the mammoth can be logically viewed as the transaction with the smallest TID in all epochs. It performs a predefined number of operations using a budget, pauses, and gets scheduled for the next epoch. Since the mammoth cannot be aborted, its execution is deterministic, ensuring that our protocol maintains conflict serializability. Reordering transactions around mammoth actions is safe due to the assumptions (iii) and (iv).

Regardless of the transaction execution order, the reservation table will always store the smallest TID that read or wrote a record during that epoch, along with information about its mammoth state. A transaction can only install its TID if it is smaller, while the mammoth independently installs its state without considering other transactions. This ensures the first phase is deterministic.

The same principle applies to the commit phase, as the reservation table remains unchanged and serves as the single source of truth for detecting conflicts. Each transaction independently uses its read-write set to make decisions in parallel with other transactions.

To ensure clients have a consistent view of the graph within a session, we also introduce bookmarks [51]. Bookmarks prevent clients from accessing `UNVISITED` or `PENDING` sections after reading results from `VISITED` mammoth sections. In other words, transactions within a session cannot time-travel and must wait for the mammoth to complete before proceeding.

3.4 Applicability and Limitations

The logic of a mammoth transitioning through the states shown in Fig. 4 is straightforward. Similar to graph traversal, the process starts from a set of nodes, and as their neighbors are accessed for reads or writes, they are marked as `PENDING`. Once all operations on a node are complete, it transitions to the `VISITED` state and will not be accessed again by the mammoth. This method efficiently handles supernodes, allowing the mammoth to pause and resume processing across multiple epochs rather than completing them in a single epoch.

The mammoth cannot update an already `VISITED` region, complicating the implementation for certain graph algorithms. For example, iterative algorithms like PageRank [46] require many nodes to stay in the `PENDING` state until completion, as nodes need to be revisited across multiple iterations. Nonetheless, our approach can still support such cases. We can extend TuskFlow with finer-grained record management at the property level (e.g., updating only one property in PageRank) or use commutative operations (e.g., incrementing a counter) to enhance concurrency. Our protocol aligns well with our customers' mammoth workloads, with expensive iterative graph algorithms handled by the Neo4j GDS library [53].

While epochs simplify conflict resolution, they also increase latency (equal to the epoch duration) compared to traditional non-deterministic protocols like 2PL or MVCC. Additionally, the mammoth transaction's budget must be chosen carefully to avoid workload imbalances within an epoch. In some cases, adding timeouts for short-lived transactions could enhance client experience when latency exceeds acceptable limits.

The phantom problem [31] can be handled with standard techniques as in [38, 44]. Overall, our protocol allows regular transactions to run concurrently with the mammoth, reducing tail latency.

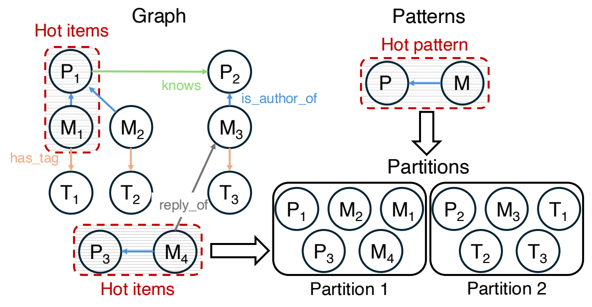

Figure 6: Partitioning based on accesses and patterns: P = Person, M = Message, T = Tag.

In the following section, we will discuss how the graph structure and workloads can further increase concurrency.

Finally, the above mechanisms are not exclusive to graph DBMSs; we observe that they could also be applied to relational systems. In simple terms, graph traversals map to table joins, nodes and relationships become tables, and the protocol proceeds similarly.

4 ACCELERATING CONFLICT RESOLUTION

Many real-world graphs have an irregular structure, often following a power-law distribution [30, 63]. This results in some neighborhoods being more densely connected than others (e.g., social or road networks). Updating high-degree nodes, or supernodes, can lead to unpredictable access bursts, making it hard to evenly distribute the processing load. A small number of these supernodes can become hotspots, causing workload skew and a higher frequency of conflicts. Handling conflicts in read-write graph traversals (i.e., chained data accesses) is more challenging than resolving conflicts in simple point lookups. In some pathological cases, a short-lived transaction may be blocked indefinitely if its path overlaps with both `VISITED` and `UNVISITED` regions. Dense graphs exacerbate this problem, as the higher chance of overlap with mammoths increases the likelihood of contention, significantly degrading performance.

To address this, we need to use the structural properties of graphs to improve conflict resolution. In this section, we introduce techniques aimed at this goal: (i) applying workload- and query-aware partitioning to prioritize the mammoth's access to regions that are less likely to block short-lived transactions due to `VISITED-UNVISITED` conflicts; (ii) parallelizing mammoths to shorten their duration and tuning the deterministic protocol to reduce the epoch span, lowering the latency of short-lived transactions; (iii) annotating parts of the graph accessed by the mammoth before execution to eliminate unnecessary conflicts; and (iv) reordering short-lived transactions to minimize aborts caused by RAW dependencies.

4.1 Query- and Workload-aware Partitioning

In our work, creating disjoint graph partitions serves a dual purpose: (i) they function as work units for parallelizing mammoths across workers (in parallel or distributed execution), and (ii) they prioritize "hot" graph regions, allowing mammoths to visit them first and reduce early contention with short-lived transactions. However, existing partitioning algorithms like Fennel [73] and LGD [62] are workload-agnostic and aim only to reduce communication overhead between partitions (i.e., relationships spanning two partitions).

Algorithm 2: RankingPM partitioner

```
Function partition(db, patterns, numberOfPartitions):
    // Sort node ids by accesses
    nodeIds ← sort(db.getNodeIds(), db.getNodeAccesses())
    partitionSize ← nodeIds.size() / numberOfPartitions
    partitions ← {∅}, curPartition ← ∅, visited ← ∅, idx ← 0
    while visited.size() ≠ nodeIds.size() do
        if curPartition.size() ≥ partitionSize then
            partitions.add(curPartition)
            curPartition ← ∅
        nodeId ← nodeIds[idx++]
        if visited.contains(nodeId) == false then
            visited.add(nodeId)
            curPartition.add(nodeId)
        for pattern ← patterns do
            update curPartition and visited based on pattern
    if curPartition.isEmpty() == false then partitions.add(curPartition)
    return partitions
```

These algorithms assume a uniform traversal likelihood for each relationship, which is realistic for offline graph algorithms like PageRank [46] but not for transactional workloads [32].

To identify graph hotspots, we track frequent access and traversal patterns. Although Neo4j does not track this metadata by default, it can be efficiently collected through sampling (see Sec. 5.1). Ranking "hot" records allows the mammoth to prioritize conflict resolution for frequent accesses. Additionally, using recurring graph patterns improves the workload-sensitive partition stability [32] by forming communities that reduce `VISITED-UNVISITED` conflicts. The patterns capture labels, properties, and relationship directions in LPGs.

Fig. 6 illustrates a subgraph of LDBC where `Person` labeled nodes P_1 to P_3 are linked to `Messages` M they authored. These `Messages` may have `Tags` T or be `Comments` on other `Messages`. Consider the scenario that the most frequent pattern submitted to the database involves reading or updating the messages of a single person (`(p:P)<-[is_author_of]-(m:M)`), and P_1 and P_3 are accessed more often than P_2. To create (for simplicity) two partitions that prioritize frequent accesses, we would include the "hot" nodes P_1, P_3, M_1, and M_4 in the first partition, as well as M_2, since it is likely to be accessed when visiting P_1. The second partition would contain the remaining nodes, which are less frequently accessed due to workload skew and query patterns (e.g., no queries access the `Tags`).

Building on the above idea, we introduce a novel partitioning heuristic called RankingPM[1], which accounts for both record access frequency and recurring graph patterns. Alg. 2 outlines the algorithm's logic: (i) it first sorts all node IDs by access frequency, from most to least frequent (line2); (ii) then, it iterates over these nodes, checking if they have not been visited[2] (line 10) and adds them to the current partition (line 12); (iii) for each node, even if already visited, the algorithm uses the most frequent graph patterns starting from that node to add more nodes to the partition, marking them as visited. This ensures that each partition is disjoint by excluding previously visited nodes.

In lines 6-8, if the current partition exceeds a predefined size of $\frac{|N|}{\#partitions}$, it is considered complete and added to the list of partitions. This can result in varying-sized partitions, potentially leading to imbalanced execution. We handle this at the transaction protocol level by pausing and resuming work, as discussed in Sec. 3.4.

RankingPM is an offline algorithm that requires graph repartitioning when the graph structure changes significantly, which is generally rare in typical transactional workloads that support a specific user-level application. In Sec. 6.3, we show that repartitioning can be completed within tens of seconds, enabling frequent execution. However, we can avoid the overhead of full repartitioning by applying a streaming repartitioning strategy [32, 40]. New nodes are placed into "cold" partitions by default, while deletions remove nodes from their partitions, allowing mammoths to bypass them.

4.2 Parallel Mammoths and Parameter Tuning

The end-to-end latency of short-lived transactions is closely tied to both the duration of the mammoth and the length of each epoch. While the mammoth is running, short-lived transactions are more likely to be aborted because of contention with the mammoth. Moreover, a longer epoch span means that short-lived transactions take longer to commit, even without conflicts. To address this challenge, we must speed up mammoths by executing them in parallel and responsively tuning the epoch duration.

Parallel execution. Since a single mammoth is structured into smaller tasks over disjoint graph partitions, a natural solution is to execute these tasks in parallel. The epoch-based structure of the deterministic protocol aligns well with parallel graph algorithms, such as Breadth-First Search (BFS) [64] or Single-Source Shortest Path (SSSP). For example, LDBC $Q1$ from Fig. 3 can compute relationship weights in parallel, while $Q2$ can perform a parallel BFS to find experts connected to a person and update their properties. In both cases, parallel tasks may perform redundant operations since they synchronize at the epoch's end, with only one task ultimately installing its writes for a given record. The tasks use compare-and-swap (CAS) operations to reserve writes within an epoch for different records and consolidate to skip already visited nodes.

Our evaluation shows that parallelizing mammoth tasks not only reduces the tail latency of short-lived transactions but also allows the system to return to regular execution faster. This prevents the database from becoming overloaded with blocked transactions.

The parameters influencing epoch duration are: (i) the epoch size, the batch size of transactions per epoch; (ii) the mammoth budget, or the operations it performs within an epoch; and (iii) the number of parallel mammoth tasks. A small epoch size may cause short-lived transactions to queue waiting for the mammoth, while an overly large size can prolong the duration unnecessarily. Our experiments in Sec. 6.5 suggest that an optimal epoch size is about five times the average transaction throughput ($tx/$ s), balancing efficiency with recovery from aborts. The mammoth budget and parallel task count depend on workload and hardware configuration. A general guideline is to set the budget as a multiple (e.g., 10×) of the average transaction throughput and assign all workers for parallelism.

4.3 Graph Tagging and Transaction Reordering

When running the transactional protocol for mammoths, an initial assumption would be that they will access any graph record (e.g., nodes, relationships, neighborhoods), meaning every entity should

[1]Named from the steps involved: **Ranking** and **P**attern **M**atching.
[2]It also checks if the nodes satisfy a filter predicate based on labels and property values (e.g., include only `Person` labeled nodes), which we omit for simplicity.

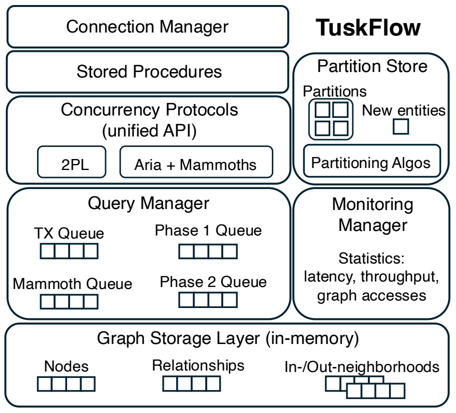

Figure 7: TuskFlow architecture

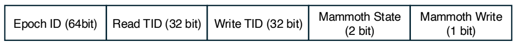

Figure 8: Reservation entry

be marked as UNVISITED. However, this does not hold true for many workloads and can often lead to unnecessary conflicts.

For example, in LDBC query $Q1$, it is known that only nodes labeled as Messages are accessed, or in another case, $Q1$ updates only the knows relationship. Thus, we can optimize execution by extracting the labels, relationship types, and record accesses specified in mammoth queries. By focusing solely on these entities, we can more accurately detect mammoth conflicts. This approach significantly prunes the graph regions considered irrelevant to the mammoth, minimizing conflicts.

Transaction reordering [10, 11, 56] can enhance performance by reducing aborts for short-lived transactions. To keep our protocol deterministic and serializable without coordination, we apply the reordering algorithm from [44], which converts RAW to WAR (Write-After-Read) dependencies. The only modification required is to add a check to the protocol in Alg. 1. Specifically, after line 31, we ensure that a transaction does not have WAR or RAW dependencies with earlier transactions (in addition to WAW) by using the read and write TIDs already recorded in each reservation entry.

5 TUSKFLOW ARCHITECTURE

While integrating our protocol into a deterministic database is straightforward, commercial graph databases [2, 47, 49, 69] are not designed to easily support deterministic transactions. To establish an easy future path to production and minimize the changes for Neo4j, we chose to adapt our protocol to existing databases by integrating it into the transaction scheduling subsystem.

In this section, we introduce TuskFlow, an experimental graph database based on Neo4j that uses two execution modes: (i) 2PL for short-lived transactions; and (ii) our deterministic protocol from Sec. 3 for mixed mammoth and short-lived transactions. We begin with an overview of the system (Sec. 5.1) and then explain how TuskFlow transitions between its two execution modes (Sec. 5.2).

5.1 Overview

Fig. 7 illustrates the components of TuskFlow, implemented as a standalone 14 K-line Java codebase, separate from Neo4j. To reduce the memory footprint of transactional queries, it uses statically allocated object pools for tasks, primitive collections [25], and roaring bitmaps [14] as in [64].

Before clients can submit queries, they connect to the *Connection Manager*, which determines when a transaction is ready for submission or when results can be returned based on bookmarks. As explained in Sec. 3.3, users cannot time-travel between VISITED and UNVISITED regions, which ensures session consistency.

For *Concurrency Protocols*, TuskFlow supports both (strict) 2PL and deterministic execution. To implement 2PL, it uses Forseti, Neo4j's open-source lock manager [52]. For the reservation table needed from Alg. 1, TuskFlow maintains a concurrent hashmap for each record type, with workers installing updates using CAS operations based on TIDs. TIDs are assigned through a monotonically increasing counter at transaction submission. Fig. 8 shows a reservation entry that uses: (i) 64 bits for the epoch; (ii) 64 bits for the earliest read-write TIDs; and (iii) 3 bits for the mammoth state.

Currently, TuskFlow supports pre-compiled stored procedures to simplify integration with Neo4j [54], similar to other deterministic databases [67, 74]. These stored procedures, written in Java, can express arbitrary logic for read-write operations over the graph database and can be used by TuskFlow to extract "hot patterns" for RankingPM. For short-lived transactions, both 2PL and the deterministic protocol share a unified API with simple read and write calls, so users are not forced to deal with the complexity of locking versus reservations – TuskFlow handles this based on the execution mode. However, for mammoths, users must manage the execution and state transitions explicitly by using the ReserveMammothRead and ReserveMammothWrite calls. We plan to develop a compiler that parses Cypher [34] and automatically generates the mammoth logic.

The *Query Manager* schedules transactions and manages transitions between the two execution modes (see Sec. 5.2). The *Graph Storage Layer* contains: (i) a node vector; (ii) a relationship vector; (iii) two vectors of incoming and outgoing relationship IDs; and (iv) auxiliary data structures on labels and relationship types.

The *Monitoring Manager* tracks system metrics like end-to-end latency, average throughput (affecting epoch size), and the count of committed and aborted transactions. During execution, workers profile a subset of read-write accesses and report to the *Monitoring Manager*. Profiling occurs per transaction (all operations recorded or none), improving the collocation of neighboring graph entities for the RankingPM partitioner.

The *Partition Store* manages graph partitions for mammoth transactions and supports hash, Fennel, and RankingPM partitioners. Parallel mammoth tasks retrieve disjoint partitions from the store, and any new nodes post-partitioning go to "cold" partitions. TuskFlow triggers repartitioning periodically, based on a configurable interval and metrics from the *Monitoring Manager*.

5.2 Transaction Scheduling

Alg. 3 outlines the scheduling logic of the *Query Manager*. During the regular execution mode, when no mammoth transactions are submitted, workers retrieve and execute short-lived transactions

Algorithm 3: Query manager transaction scheduling

```
// txQueue stores short-lived txs
// mQueue stores mammoth txs
// p1Queue/p2Queue stores txs for protocol's 1st/2nd phase
Function run(txQueue, mQueue, p1Queue, p2Queue):
    while true do
        if mQueue.isEmpty() == false then
            // Schedule tasks until there is no mammoth
            ScheduleInEpochs(txQueue, mQueue, p1Queue, p2Queue)
            // Aborted queries use regular execution from here
            ResetExecutionMode(txQueue, p1Queue)
Function ScheduleInEpochs(txQueue, mQueue, p1Queue, p2Queue):
    while mQueue.isEmpty() == false do
        mammoth ← mQueue.poll()
        p1Queue.add(mammoth)
        while mammoth.hasFinished == false do
            size ← min(epochSize - p1Queue.size(), txQueue.size())
            for i ← 0 to size do
                tx ← txQueue.poll()
                move tx to p1Queue and set its protocol to deterministic
            p1Counter ← p1Queue.size()
            // start phase 1 and wait until p1Counter == 0
            p2Counter ← p2Queue.size()
            // start phase 2 and wait until p2Counter == 0
Function ResetExecutionMode(txQueue, p1Queue):
    for tx ← p1Queue do  move tx to txQueue and set its protocol to 2PL
```

from the txQueue using 2PL. However, when one or more mammoth transactions arrive, the *Query Manager* switches to deterministic execution mode (line 4) and processes them one at a time. In this mode, workers handle transactions from two queues: p1Queue and p2Queue, corresponding to the two protocol phases.

Based on the epoch size, a fixed number of transactions from the txQueue are moved to the p1Queue (lines 12-14), along with the mammoth transaction (line 9), and a barrier is set. Once all tasks in the first phase are completed, they are moved to the p2Queue, and the second phase begins with a new barrier (see Sec. 3.2). Aborted transactions are moved back to p1Queue before new transactions from txQueue can be added based on available slots within the epoch size (line 11). After all mammoth transactions are processed, the *Query Manager* resets any aborted transactions to 2PL, places them back in txQueue, and resumes regular execution mode.

Table 1: Evaluation datasets

Dataset	Domain	|V|	|E|	|E|/|V|
WikiTalk [42]	communication net.	1 M	7.8 M	7.8
DBPedia [8]	hyperlink	18 M	172 M	9.5
USRoad [60]	rail network	24 M	58 M	4.8
LDBC SF10 [30, 63]	social network	34 M	165 M	9.8

6 EVALUATION

In this section, we evaluate the performance of TuskFlow with mammoths. We show that it does not block short-lived transactions and delivers lower tail latency compared to 2PL and MVCC (Sec. 6.2). Next, we analyze the impact of graph partitioning (Sec. 6.3) and present an optimization breakdown of the conflict resolution techniques (Sec. 6.4) from Sec. 4. Lastly, we offer guidelines for tuning the deterministic protocol (Sec. 6.5) and demonstrate how TuskFlow performs as the transaction throughput increases (Sec. 6.6).

6.1 Experimental Setup

All experiments were performed on an m5.8xlarge AWS EC2 instance with 32 physical cores, 35.8 MiB LLC, and 128 GiB memory, using Amazon Linux 2023 (kernel v. 6.1) and Corretto OpenJDK17.

Datasets and workloads. For our evaluation, we use two real-world graph datasets and a synthetic one to provide a diverse set of scenarios: (i) DBPedia[8], a hyperlink network of Wikipedia where pages are nodes and hyperlinks are relationships; (ii) USRoad [60], a low-degree road network graph with a grid-like structure; and (iii) LDBC [30, 63], which simulates real-world interactions over a social network. Since USRoad is undirected, we create two relationships for each original one, and for LDBC, we store only the properties required for our queries. Table 1 summarizes the graphs.

The LDBC SNB benchmark represents an OLTP workload, yet it only covers short-lived transactions. To simulate mammoths on the LDBC network, we also use the three queries described in Sec. 2.5. These queries represent typical production workloads for both balanced and unbalanced mammoths. Specifically, $Q1$ and $Q2$ are (offline) analytical LDBC-BI queries [63], which we run as (online) mammoth transactions alongside regular ones. Since DBPedia and USRoad do not have labeled nodes, we only use a variation of $Q0$.

Database systems. We compare our deterministic protocol to 2PL using TuskFlow instead of Neo4j[3] for fairness [65]. For MVCC, we use Postgres v.15.0, configured according to best practices. Aria performs similarly to 2PL but with higher latency due to blocking all transactions until the mammoth completes. Unless stated otherwise, we set the epoch budget to 2M ops, use 32 parallel tasks, and set the epoch size to 5× the average transaction throughput.

Metrics. The main performance metrics in the benchmarks are throughput and end-to-end tail latency, measured at the 99th percentile (p99). Candlesticks in the plots represent the 5th, 25th, 50th, 75th, and 99th latency percentiles.

Table 2: p99 latency

Workload	p99 latency (Mammoth duration) in s		
	TuskFlow	2PL	Postgres
DBpedia	0.9 (4.1)	10.5 (11.7)	40.9 (71.6)
USRoad	0.8 (4.8)	13.7 (14.9)	19.2 (91.7)

6.2 Comparison with 2PL and Postgres

To study the efficiency of TuskFlow, we use the LDBC social network with queries $Q0$-$Q2$ and measure the throughput of short-lived transactions. In this experiment, we fix the input rate of short-lived transactions at 10K per second, which is a ballpark representative of a modest real system. All clients submit either read-only or read-write 1-hop queries, with 80% being read-only. These queries start from a random Person (uniform distribution) and either read or update ten random Messages based on the pattern (p:P)<-[is_author_of]-(m:M). We compare our deterministic protocol to 2PL by starting a single mammoth at the 60 s mark.

For the simplest query, $Q0$, which updates all nodes labeled as Messages, Fig. 9 shows that TuskFlow completes the mammoth execution in under 8 s, thanks to the optimizations discussed in

[3]2PL provides conflict serializability, whereas Neo4j offers Read Committed isolation.

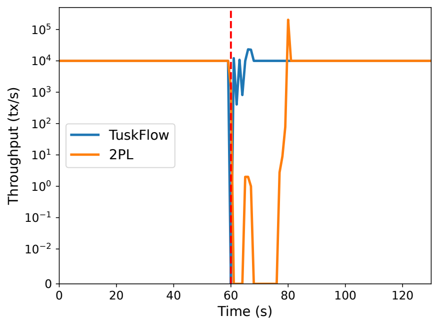

Figure 9: LDBC Q0

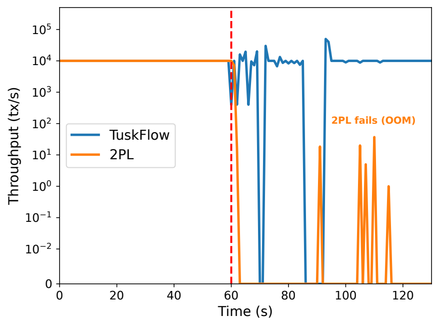

Figure 10: LDBC Q1

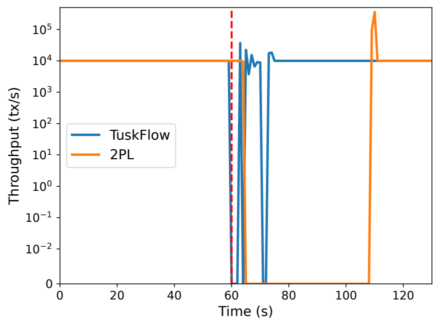

Figure 11: LDBC Q2

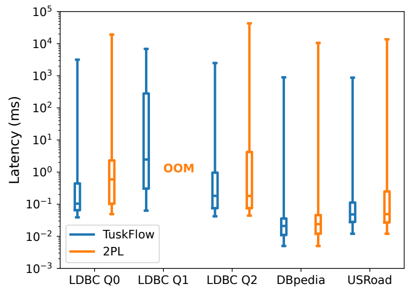

Figure 12: Latency comparison

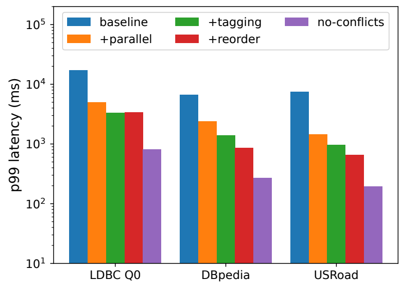

Figure 13: Optimizations breakdown

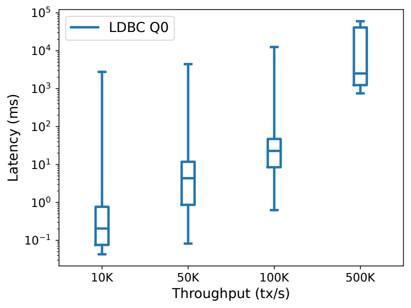

Figure 14: Latency with increasing tx/s

Sec. 4, in particular the parallel execution. In contrast, 2PL blocks all transactions for 21 s, whereas TuskFlow allows both read-only and read-write transactions to make progress and commit.

For *Q*1, the most compute-intensive query, Fig. 10 shows that TuskFlow completes the mammoth in 60 s while maintaining a throughput close to the input rate. However, there are two points where throughput temporarily drops to zero. This happens because many transactions are aborted due to the mammoth, causing the protocol to spend considerable time scheduling them. During these periods, the epoch duration exceeds 1 s, leading to a temporary stall in throughput. In contrast, 2PL fails to complete due to transaction overload, eventually causing an out-of-memory exception, further demonstrating the importance of parallelizing mammoths.

*Q*2 in Fig. 11 exhibits a similar pattern to *Q*0. TuskFlow completes the mammoth in under 15 s, compared to 68 s with 2PL. Although some epochs temporarily block short-lived transactions, the average throughput remains close to 10K tx/s.

Next, we evaluate the impact of mammoths on the end-to-end latency of short-lived transactions, using the LDBC social network with *Q*0-*Q*2, and the DBpedia and USRoad graphs. For DBpedia and USRoad, the mammoth behaves as LDBC *Q*0 and updates all nodes, while the 1-hop queries start from any random node and either read or update ten random outgoing neighbors. Fig. 12 shows that 2PL results in up to 6× higher median and p75 latency. Additionally, p95 (not shown here) and tail latency can be up to 180× and 17.2× higher. This translates into waiting minutes instead of hundreds of milliseconds, which is critical for responsive systems.

To simplify workloads for Postgres, we use the unlabeled DBpedia and USRoad graphs, create hash indexes for in- and out-relationships, and reduce the input rate to 2.5K tx/s, as Postgres cannot otherwise handle the 1-hop traversals. As shown in Table 2, Postgres experiences 24-45× higher tail latency than TuskFlow and performs worse than 2PL, which completes the mammoth faster.

In summary, choosing the right protocol significantly impacts both the throughput and latency of short-lived transactions and TuskFlow greatly improves concurrency. Finally, our profiling of 2PL revealed that the lock manager is a major bottleneck for mammoth transactions.

Table 3: Partitioning with point and 1-hop queries

Workload	Read ratio	p99 latency in s				
		Point queries		1-hop queries		
		Serial	Ranking	Serial	Ranking	R1hop
DBpedia	80%	1.53	1.86	0.86	0.87	0.50
	90%	1.75	2.01	0.54	0.62	0.34
	100%	1.72	1.92	0.23	0.48	0.28
LDBC	80%	3.27	3.91	2.30	2.41	1.04
	90%	3.62	3.61	1.58	1.99	0.67
	100%	4.14	3.62	2.41	1.47	0.93
USRoad	80%	3.74	3.80	1.16	1.02	0.71
	90%	3.68	3.86	1.31	0.93	0.79
	100%	3.45	3.23	1.19	1.02	0.73

6.3 Graph Partitioning with Skewed Workloads

Given most graph workloads will exhibit skewed access patterns, we now evaluate the benefits of the RankingPM partitioner in

Table 4: Partitioning with multi-hop queries

Workload	Read ratio	p99 latency in s					
		Serial	Rank-ing	R1hop	R2hop	R3hop	RPatterns
DBpedia - 2-hops	80%	0.67	0.73	0.40	0.47	-	-
	90%	0.53	0.78	0.32	0.36	-	-
	100%	2.05	3.65	0.57	1.29	-	-
LDBC - 2-hops	80%	2.05	3.65	0.57	1.29	-	-
	90%	1.71	3.45	0.82	1.08	-	-
	100%	2.81	2.79	0.76	1.67	-	-
USRoad - 2-hops	80%	0.99	1.14	0.81	0.88	-	-
	90%	1.10	1.02	0.72	0.65	-	-
	100%	0.98	0.87	1.04	0.76	-	-
DBpedia - 3-hops	80%	0.53	0.80	0.31	0.49	0.32	-
	90%	0.53	0.65	0.27	0.34	0.25	-
	100%	0.49	0.65	0.25	0.24	0.42	-
LDBC - 3-hops	80%	1.75	3.41	0.92	0.98	5.50	-
	90%	1.53	3.75	0.75	1.01	5.20	-
	100%	2.87	3.79	1.15	1.93	7.63	-
USRoad - 3-hops	80%	0.99	1.49	1.11	0.76	0.78	-
	90%	0.90	1.18	1.04	0.85	0.83	-
	100%	0.82	1.06	0.90	0.73	0.65	-
DBpedia - mixed	80%	0.70	0.71	0.63	0.62	0.72	0.59
	90%	0.74	0.90	0.50	0.89	0.80	0.79
	100%	0.29	0.35	0.27	0.22	0.21	0.29
LDBC - mixed	80%	3.24	2.98	1.41	2.48	5.66	0.89
	90%	2.39	3.33	0.83	1.79	4.95	1.05
	100%	3.13	3.26	1.35	2.00	5.17	1.36
USRoad - mixed	80%	1.06	1.36	0.93	0.95	0.66	0.41
	90%	0.68	1.13	0.89	0.65	0.65	0.70
	100%	0.95	1.55	0.83	0.95	1.03	1.01

these scenarios. Specifically, we aim to understand which types of patterns can enhance performance for various query workloads. As a baseline, we use serial partitioning, where nodes are assigned to partitions based on their IDs. This approach can offer good average performance due to its memory-friendly node access, particularly for synthetic graphs like LDBC SNB, where a person's messages are often collocated within the same partition. We omit comparisons with hash and Fennel partitioning algorithms, as they are workload-agnostic and significantly degrade performance in our evaluation.

For this experiment, we use LDBC SNB, DBpedia, and USRoad graphs, with $Q0$ as the mammoth. Short-lived transactions start at a node selected using a Zipfian distribution ($\theta = 1$) applied to randomly shuffled node IDs, and results are averaged across 5 runs. The short-lived transactions have 5 different configurations: (i) point queries; (ii) 1-hop queries; (iii) 2-hop queries; (iv) 3-hop queries; and (v) mixed transactions, which consist of 25% point queries, 55% 1-hop queries, 15% 2-hop queries, and 5% 3-hop queries – similar to common production workloads. In LDBC SNB, multihop queries use the pattern `(p1:P)-[:know]{hops-1}->(p2:P)` to access a `Person` within their social circle and read/write 10 of its `Messages`. In DBpedia and USRoad, the queries traverse `hops-1` outgoing relationships and then access 10 direct relationships from the final node.

For each configuration, we evaluate different variations of RankingPM: (i) Ranking without using patterns for partitioning; (ii) R1hop, which leverages the most frequent 1-hop patterns; (iii) R2hop, using 2-hop patterns; (iv) R3hop, using 3-hop patterns; and (v) RPatterns, which uses all available patterns. Lastly, we vary the percentage of read-only queries between 80% and 100% while keeping the input rate at the maximum capacity that the lock manager can handle. Tables 3 and 4 summarize the results regarding tail latency, with the best approach for each workload highlighted in green.

Starting with point queries and only Ranking (no patterns) as shown in Table 3, serial partitioning results in tail latencies that are 2-21% lower. The exception is for 100% read-only workloads with LDBC SNB and USRoad, where Ranking improves latency by up to 13%. This indicates that relying solely on node accesses for partitioning does not always enhance performance if the reduction in conflicts does not outweigh the benefits of sequential memory accesses. However, with 1-hop queries, utilizing the most frequent 1-hop pattern significantly reduces tail latency by 1.6-2.6×.

For 2-hop queries (Table 4), R1hop generally delivers the best performance, outperforming Serial by 1.5-3.5× for most workloads. The exception is USRoad, where R2hop surpasses R1hop due to the graph's low-degree structure, which favours partitioning that collates long paths. Using the most common 2-hop patterns requires at least 30% more time for partitioning. A similar trend is found with 3-hop queries: R1hop achieves nearly 2× lower tail latency than Serial, except for USRoad, where R3hop performs best, albeit with longer partitioning times. We also observe that using 2- or 3-hop patterns can have negative effects on performance (see LDBC-3-hops), as it can split neighboring nodes that should have been collocated (e.g., a `Person` not connected to her `Messages`).

For mixed workloads, RPartitions, which uses all available patterns regardless of length, generally achieves lower or similar tail latency compared to R1hop. However, RPartitions requires at least 3× more time for partitioning, which can be expensive for large graphs and multiple patterns. Despite this, even R1hop still provides up to 3.6× lower tail latency compared to Serial partitioning.

Overall, using ranking with the most frequent 1-hop patterns (R1hop) offers the best balance between partitioning cost and performance. The effectiveness of the "hot" patterns improves when they include label information – note that only LDBC SNB has labels. Finally, Serial partitioning is expected to perform worse on real-world graphs, where frequent updates can disrupt the collocation of neighboring nodes with numerically close IDs.

6.4 Optimization Breakdown

We study TuskFlow's conflict optimization techniques from Sec. 4 using the LDBC SNB, DBpedia, and USRoad datasets with $Q0$ and 1-hop read-write queries (80% read-only, at 10K tx/s). We measure the tail latency of short-lived transactions across 5 configurations: (i) no optimizations (baseline); (ii) parallel execution; (iii) graph tagging on top of parallel execution; (iv) transaction reordering with the previous optimizations; and (v) emulating an ideal protocol that resolves all conflicts (no-conflicts). Fig. 14 shows that parallel execution reduces tail latency by 2.7-5.2× by shortening the mammoth duration. Graph tagging cuts tail latency by roughly 50%, while transaction reordering improves performance for DBpedia and USRoad by 62% and 49%, respectively. Transaction reordering has no effect on the LDBC SNB graph, as it has fewer write conflicts between regular transactions, though it does help with skewed workloads. Finally, the no-conflicts configuration further reduces latency by 3.5×, indicating the potential of more advanced techniques, such as better transaction reordering algorithms [16].

Table 5: Parameter tuning

Budget (#ops)	#Tasks	p99 latency (Mammoth duration) in s		
		10K tx/s	20K tx/s	40K tx/s
125K	1	18.8 (30.8)	30.5 (45.5)	62.5 (83.4)
	8	8.3 (15.8)	20.7 (30.9)	53.3 (72.3)
	16	8.9 (15.9)	22 (32.1)	53 (72)
	32	7.9 (14.9)	21.8 (31.8)	48.7 (65.9)
	64	8.6 (15.6)	21.2 (31.2)	50.9 (69)
500K	1	12.2 (22.3)	15.6 (26.4)	22.5 (35.4)
	8	5 (10.9)	6.6 (12.8)	12.7 (20.8)
	16	4.2 (9.5)	4.4 (10.7)	10.6 (18.0)
	32	3.2 (8.7)	5.6 (11.2)	10.6 (17.8)
	64	3.9 (9.1)	4.4 (10.3)	10.7 (17.7)
2M	1	12.4 (20.5)	13.8 (22)	16.1 (25.9)
	8	4 (8.5)	4.1 (9.1)	5.7 (11.4)
	16	3.1 (7)	2.9 (7.6)	4.5 (9.4)
	32	2.1 (5.9)	3.2 (7.6)	4.5 (9.1)
	64	2.4 (6.4)	3.3 (7.8)	4.5 (8.9)
4M	1	14.3 (20.3)	14.7 (21.3)	15.3 (22)
	8	4.6 (8.7)	5.1 (9.2)	5.3 (9.5)
	16	3.9 (7.8)	4.3 (8.1)	3.5 (7.7)
	32	3 (6.5)	3.5 (7.4)	3.9 (8.1)
	64	3.4 (7.2)	3.4 (7.3)	4.6 (8.4)

6.5 Parameter Tuning

Since the duration of mammoth transactions and epoch length impact the performance of short-lived transactions, this experiment explores how different protocol parameters affect tail latency. Table 5 summarizes the results, varying the epoch budget from 125K to 4M ops per epoch and the number of parallel tasks from 1 to 64. We set the epoch size to 5× the average transaction throughput, using input rates of 10K, 20K, and 40K tx/s, with the best configuration for each highlighted in green. All experiments use the LDBC SNB graph with 1-hop traversals (80% read-only) and $Q0$ as the mammoth query. We observe that increasing the epoch budget from 125K to 2M reduces tail latency and overall transaction duration, but further increase show diminishing returns. Similarly, increasing the number of parallel mammoth tasks within an epoch improves performance, but only up to 16 or 32 tasks. Based on these results, we conclude that the number of tasks should not exceed the number of available workers, and the mammoth budget should be around 10× the average transaction throughput.

6.6 TuskFlow's Scalability

Finally, we examine the scalability of TuskFlow using the LDBC SNB graph with $Q0$ and 1-hop read-write queries (80% read-only). We measure the end-to-end latency of short-lived transactions while varying the input rate from 10K to 500K tx/s. At 50K tx/s, the median and p75 latency increase by nearly 18×, but tail latency remains largely unaffected compared to 10K tx/s. Doubling the rate to 100K tx/s raises all latency percentiles by around 5.9×, again with only a small impact on tail latency. Even at this rate, latencies remain at the granularity of a second, which is acceptable for many applications. However, when the input rate reaches 500K tx/s, latency increases by up to two orders of magnitude, revealing the limitations of our epoch-based approach due to the high scheduling overhead. A distributed extension could help mitigate this issue by partitioning the transaction load across multiple nodes.

7 RELATED WORK

Transactional benchmarks for mammoths. Most existing relational database benchmarks overlook mammoth transactions, even though they play a crucial role in many applications [18]. For example, HTAP benchmarks [6, 23, 24] combine OLTP queries [71] with read-only queries from OLAP workloads [72]. Only OLxPBench [41] introduces mammoth transactions that consist of read-write operations with analytical reads and demonstrates how they significantly impact the performance of HTAP systems.

Graph benchmarks have primarily focused on analytics [5, 13], with only LDBC [30, 63] attempting to capture short update queries alongside complex read-only ones. TAOBench [17] from Meta is also limited to a small set of large write transactions. The queries introduced in Sec. 2.5 aim to simulate typical production mammoth workloads. In Sec. 6, we highlight that tail latency, in addition to throughput, are both critical metrics to consider, and we also emphasize the significance of skewed access patterns for short-lived transactions when running a mammoth.

Locking techniques. Fine-grained locking [38] is a common strategy to increase concurrency while preserving transactional semantics. While this approach cannot address issues with mammoth transactions, our protocol could adopt similar fine-grained record management at the graph property level [4]. This would help prevent unnecessary conflicts caused by common graph algorithms, like PageRank, which update only a few properties.

Another recent approach for mammoths is lock escalation [18], which locks based on communities and motifs rather than individual graph entities or properties. While this can improve the ability of existing lock managers to handle mammoths, it still blocks short-lived transactions from making progress. Our query- and workload-aware partitioning approach was inspired by this idea, but it constructs communities based on access patterns, enabling concurrency without blocking other transactions.

Transaction scheduling is orthogonal to our epoch-based approach. LDSF [68], a hotspot-aware scheduling algorithm, might help to prioritize transactions that block others within an epoch. Other techniques schedule transactions by identifying "hot" keys and postponing requests [12] or by learning abort patterns between transactions [61]. As discussed in Sec. 6.4, we could achieve at least a 3.5× improvement in tail latency by exploring more advanced reordering algorithms [16]. However, it is crucial to consider how these methods might impact determinism and serializability.

8 CONCLUSION

In this work, we introduce the first deterministic concurrency protocol for mammoths, very large online transactions, that ensures conflict serializability by strategically reordering regular transactions around a mammoth. To further reduce conflicts with short-lived transactions, we propose techniques that exploit graph properties, including query- and workload-aware partitioning, graph entity tagging, mammoth decomposition and parallel execution. These techniques, implemented in TuskFlow, dramatically improve concurrency and reduce latency by up to 45× compared to 2PL or MVCC. Finally, while designed for graph databases, our protocol also applies to relational systems, which face similar problems with very large transactions.

REFERENCES

[1] Daniel J Abadi and Jose M Faleiro. 2018. An overview of deterministic database systems. *Commun. ACM.* (2018).

[2] Amazon Neptune. 2024. https://aws.amazon.com/neptune/. Last access: June 5, 2025.

[3] Konstantin Andreev and Harald Räcke. 2004. Balanced graph partitioning. In *SPAA.*

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *CSUR* (2017).

[5] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: a database benchmark based on the facebook social graph. In *SIGMOD.*

[6] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *SIGMOD.*

[7] Manoussos Athanassoulis, Kenneth Bøgh, and Stratos Idreos. 2019. Optimal column layout for hybrid workloads. *Proc. VLDB Endow.* (2019).

[8] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *Proc. Int. Semant. Web Conf.*

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

[10] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *TODS* (1983).

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

[12] Yang Cao, Wenfei Fan, Weijie Ou, Rui Xie, and Wenyue Zhao. 2023. Transaction Scheduling: From Conflicts to Runtime Conflicts. *PACMMOD* (2023).

[13] Mihai Capotă, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter Boncz. 2015. Graphalytics: A big data benchmark for graph-processing platforms. In *GRADES.*

[14] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. 2016. Better bitmap performance with roaring bitmaps. *Software: practice and experience* (2016).

[15] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, et al. 2022. ByteHTAP: bytedance's HTAP system with high data freshness and strong data consistency. *Proc. VLDB Endow.* (2022).

[16] Audrey Cheng, Aaron Kabcenell, Jason Chan, Xiao Shi, Peter Bailis, Natacha Crooks, and Ion Stoica. 2024. Towards Optimal Transaction Scheduling. *Proc. VLDB Endow.* (2024).

[17] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, et al. 2022. Taobench: An end-to-end benchmark for social network workloads. *Proc. VLDB Endow.* (2022).

[18] Audrey Cheng, Jack Waudby, Hugo Firth, Natacha Crooks, and Ion Stoica. 2024. Mammoths Are Slow: The Overlooked Transactions of Graph Data. *Proc. VLDB Endow.* (2024).

[19] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.* (2015).

[20] CNBC. 2018. Amazon's move off Oracle caused Prime Day outage in one of its biggest warehouses, internal report says. https://www.cnbc.com/2018/10/23/amazonmove-off-oracle-caused-prime-day-outage-in-warehouse.html. Last access: June 5, 2025.

[21] CockroachDB. 2016. How online schema changes are possible in CockroachDB. https://www.cockroachlabs.com/blog/how-online-schema-changes-arepossible-in-cockroachdb/. Last access: June 5, 2025.

[22] CockroachDB. 2020. Nested transactions in CockroachDB 20.1. https://www.cockroachlabs.com/blog/nested-transactions-in-cockroachdb-20-1/. Last access: June 5, 2025.

[23] Fábio Coelho, João Paulo, Ricardo Vilaça, José Pereira, and Rui Oliveira. 2017. Htapbench: Hybrid transactional and analytical processing benchmark. In *ICPE.*

[24] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. 2011. The mixed workload CH-benCHmark. In *DBTest.*

[25] Eclipse Collections. 2024. https://github.com/eclipse/eclipse-collections. Last access: June 5, 2025.

[26] Datanami. 2021. Graph Database Market Worth $5.1 Billion by 2026. https://www.datanami.com/this-just-in/graph-database-market-worth-5-1-billion-by-2026/. Last access: June 5, 2025.

[27] Jean-Charles Delvenne, Michael T Schaub, Sophia N Yaliraki, and Mauricio Barahona. 2013. The stability of a graph partition: A dynamics-based framework for community detection. *Dynamics On and Of Complex Networks, Volume 2: Applications to Time-Varying Dynamical Systems* (2013).

[28] J-C Delvenne, Sophia N Yaliraki, and Mauricio Barahona. 2010. Stability of graph communities across time scales. *PNAS* (2010).

[29] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, et al. 2022. Graph pattern matching in GQL and SQL/PGQ. In *SIGMOD.*

[30] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *SIGMOD.*

[31] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM.* (1976).

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

[33] Hugo Firth, Paolo Missier, and Jack Aiston. 2018. Loom: Query-aware Partitioning of Online Graphs. In *EDBT.*

[34] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *SIGMOD.*

[35] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. *Sigmod Record* (1987).

[36] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. 2019. Neptune: Scheduling suspendable tasks for unified stream/batch applications. In *SOCC.*

[37] Jim Gray et al. 1981. The transaction concept: Virtues and limitations. In *VLDB.*

[38] Jim N Gray, Raymond A Lorie, and Gianfranco R Putzolu. 1975. Granularity of locks in a shared data base. *Proc. VLDB Endow.* (1975).

[39] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *SIGMOD.*

[40] Jiewen Huang and Daniel J Abadi. 2016. Leopard: Lightweight edge-oriented partitioning and replication for dynamic graphs. *Proc. VLDB Endow.* (2016).

[41] Guoxin Kang, Lei Wang, Wanling Gao, Fei Tang, and Jianfeng Zhan. 2022. Olxpbench: Real-time, semantically consistent, and domain-specific are essential in benchmarking, designing, and implementing htap systems. In *ICDE.*

[42] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Governance in social media: A case study of the Wikipedia promotion process. In *ICWSM.*

[43] 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).

[44] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proc. VLDB Endow.* (2020).

[45] Nancy Lynch and Michael Merritt. 1986. Introduction to the theory of nested transactions. *Theoretical Computer Science* (1986).

[46] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD.*

[47] MemGraph. 2024. https://memgraph.com/. Last access: June 5, 2025.

[48] Mark Needham and Amy E Hodler. 2019. *Graph algorithms: practical examples in Apache Spark and Neo4j.* O'Reilly Media.

[49] Neo4j. 2010. Neo4j Graph Data Platform. https://neo4j.com/. Last access: June 5, 2025.

[50] Neo4j. 2021. Neo4j Breaks Scale Barrier with Trillion+ Relationship Graph. https://neo4j.com/press-releases/neo4j-scales-trillion-plus-relationship-graph/. Last access: June 5, 2025.

[51] Neo4j. 2024. Coordinate parallel transactions. https://neo4j.com/docs/java-manual/current/bookmarks/. Last access: June 5, 2025.

[52] 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: June 5, 2025.

[53] Neo4j. 2024. Neo4j Graph Data Science. https://github.com/neo4j/graph-data-science. Last access: June 5, 2025.

[54] Neo4j. 2024. User-defined procedures. https://neo4j.com/docs/java-reference/current/extending-neo4j/procedures/. Last access: June 5, 2025.

[55] Anil Pacaci and M Tamer Özsu. 2019. Experimental analysis of streaming algorithms for graph partitioning. In *sigmod.*

[56] Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *JACM* (1979).

[57] PostgreSQL. 2023. PostgreSQL 15 Documentation SAVEPOINT. https://www.postgresql.org/docs/current/sql-savepoint.html. Last access: June 5, 2025.

[58] Ian Rae, Eric Rollins, Jeff Shute, Sukhdeep Sodhi, and Radek Vingralek. 2013. Online, asynchronous schema change in F1. *Proc. VLDB Endow.* (2013).

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

[60] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence.* http://networkrepository.com

[61] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. 2019. Scheduling OLTP transactions via learned abort prediction. In *aiDM.*

[62] Isabelle Stanton and Gabriel Kliot. 2012. Streaming graph partitioning for large distributed graphs. In *SIGKDD*.

[63] Gábor Szárnyas, Jack Waudby, Benjamin A Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2022. The LDBC social network benchmark: Business intelligence workload. *Proc. VLDB Endow.* (2022).

[64] Georgios Theodorakis, James Clarkson, and Jim Webber. 2024. Aion: Efficient Temporal Graph Data Management.. In *EDBT*.

[65] Georgios Theodorakis, James Clarkson, and Jim Webber. 2024. An Empirical Evaluation of Variable-length Record B+ Trees on a Modern Graph Database System. In *ICDEW*.

[66] Alexander Thomson and Daniel J Abadi. 2010. The case for determinism in database systems. *Proc. VLDB Endow.* (2010).

[67] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*.

[68] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. 2018. Contention-aware lock scheduling for transactional databases. *Proc. VLDB Endow.* (2018).

[69] TigerGraph. 2024. https://www.tigergraph.com/. Last access: June 5, 2025.

[70] 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).

[71] TPC. 2010. TPC BENCHMARK C Standard Specification Revision 5.11 . https://www.tpc.org/TPC_Documents_Current_Versions/pdf/tpc-c_v5.11.0.pdf. Last access: June 5, 2025.

[72] TPC. 2022. TPC BENCHMARK H (Decision Support) Standard Specification Revision 3.0.1. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf. Last access: June 5, 2025.

[73] Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. Fennel: Streaming graph partitioning for massive scale graphs. In *WSDM*.

[74] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*.

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

[76] Jack Waudby, Paul Ezhilchelvan, Jim Webber, and Isi Mitrani. 2020. Preserving reciprocal consistency in distributed graph databases. In *PaPoC*.

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

[78] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. 2013. Graphx: A resilient distributed graph system on spark. In *GRADES*.