Mammoths Are Slow: The Overlooked Transactions of Graph Data

Audrey Cheng
UC Berkeley
accheng@berkeley.edu

Jack Waudby
Neo4j
jack.waudby@neo4j.com

Hugo Firth
Neo4j
hugo.firth@neo4j.com

Natacha Crooks
UC Berkeley
ncrooks@berkeley.edu

Ion Stoica
UC Berkeley
istoica@berkeley.edu

ABSTRACT

This paper argues for better concurrency control to support *mammoth transactions*, which read and write to many items. While these requests are prevalent on graph data, few systems support them efficiently. Currently, developers must make the uncomfortable choice between accepting dismal performance or abandoning transactional semantics. Applications deserve better: we believe that inherent graph properties provide a path forward to designing efficient concurrency control that preserves strong isolation.

PVLDB Reference Format:
Audrey Cheng, Jack Waudby, Natacha Crooks, Hugo Firth, Ion Stoica. Mammoths Are Slow: The Overlooked Transactions of Graph Data. PVLDB, 17(4): 904 - 911, 2023.
doi:10.14778/3636218.3636241

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/jackwaudby/mammoths-are-slow.

1 INTRODUCTION

This paper observes that long running read-write, or *mammoth* transactions are underserved by existing concurrency control protocols and benchmarks. Yet, these requests are increasingly prevalent in graph workloads.

In large part, mammoths have been overlooked by modern systems. Traditionally, databases categorize workloads as either OLAP (read-only, long running) or OLTP (write-intensive, short-lived). HTAP systems [23, 24, 28, 49, 52, 54, 57, 59], which serve both workloads, make similar assumptions: analytical transactions are long running and read-only whereas read-write transactions are short. These systems presume that *long running read-write* transactions are rare and choose to not support them due to the significant challenges in doing so efficiently (Section 3).

However, the increasingly popular graph data model challenges these assumptions. This model enables applications developers to intuitively express complex business logic as graph queries [21, 30, 35, 42]. As a result, developers can easily design transactions that span large parts of the underlying graph. For instance, mammoths are frequently generated by graph analytics workloads, such as pattern matching, in applications for fraud detection, network analytics, and access control [36, 39, 60]. These transactions also result from cascading deletes, schema changes [4, 61], and graph-processing algorithms like community detection [56]. We provide several real-world case studies from graph databases and large social networks in Section 2.

Supporting mammoth transactions with high performance and strong isolation guarantees is difficult because these requests affect large portions of the graph. Consequently, they can block the progress of many other transactions and have outsized impact on overall system performance. We find that naively running mammoth transactions under existing concurrency control methods yields unsatisfactory results. To illustrate, we evaluate Neo4j, a leading graph database, on a variety of mammoth transactions (Section 3). We observe that throughput drops by up to 4.7× during the execution of a *single* mammoth. To the best of our knowledge, *no system* that stores graph data supports general mammoth transactions with high performance.

Despite the prominence of these transactions on graph data, state-of-the-art approaches that explicitly address these requests are limited. At best, systems implement ad hoc mechanisms that are restricted to specific use cases. For instance, TAO, Meta's graph data store, applies a special case algorithm that does not use two-phase commit for large write-only transactions [29]. In the worst (and most common) case, users must give up on transactional semantics. Neo4j's Graph Data Science library [17] and MemGraph's MAGE graph processing library [14] both allow users to run computationally expensive graph algorithms but provide no isolation guarantees for writes to the database.

Consequently, application developers are currently left to deal with mammoths themselves: they must either carefully craft application logic around these transactions to manually preserve consistency or accept the downsides of weaker guarantees. As conversations with engineers from Neo4j reveal, application workarounds (e.g., use of locking primitives or validation procedures outside of the data store) are unwieldy, error-prone, and significantly increase development burden [25, 66]. The lack of isolation can result in data corruption and other undesirable side effects.

We believe that graph data stores *can* and *should* do better. Contrary to the relational model, graphs have inherent structure and maintain invariants that enable more efficient concurrency control. For instance, graph edges can be stored in a sorted adjacency list

Proceedings of the VLDB Endowment, Vol. 17, No. 4 ISSN 2150-8097.
doi:10.14778/3636218.3636241

to prevent random traversals and thus minimize the risk of deadlocks [43]. Moreover, transactional semantics can build on existing consistency guarantees in the graph (e.g., no duplicate edges). Together, these properties can help us design efficient concurrency control protocols for mammoth transactions with strong isolation (e.g., Read Committed, Snapshot Isolation, and Serializability).

In the rest of this paper, we study mammoths in detail and propose paths forward for supporting them. We first characterize common use cases (Section 2) before demonstrating that running even a single mammoth under Read Committed isolation significantly degrades performance (Section 3). Next, we summarize existing approaches for mammoths (Section 4) and outline opportunities to speed up these transactions (Section 5).

2 A NEW ICE AGE: THE EMERGENCE OF MAMMOTHS

Mammoth transactions, defined as long running requests that contain at least one write, are common in the graph model (Section 2.1) and can be divided into two types: 1) *balanced* mammoths with many reads and writes and 2) *unbalanced* mammoths that read many objects but write to only a few. While there are many real-world use cases (Section 2.2), we find that few benchmarks capture these requests (Section 2.3).

2.1 The Graph Data Model

Under the graph data model, developers typically access data through a higher level query language that simplifies graph traversal and intentionally hides what operations are sent to the data store. Furthermore, graph data is often highly connected and regularly contains supernodes, which have a high number of incident edges. Together, these factors drastically increase the frequency of mammoths.

First, we explain how high level query languages induce mammoth transactions. These frameworks make it easy to express many common graph-processing algorithms by deliberately hiding the complex series of traversals and updates sent to an underlying store. For instance, the Cypher query language, which is used Amazon Neptune [8], Memgraph [15], and Neo4j [62], provides native support for traversing the graph with methods such as `MATCH` (searches for patterns) and `MERGE` (matches existing nodes and adds edges between them, or creates new nodes and connects them) [47].

Consequently, even seemingly simple queries can result in large read and write sets on highly interconnected data. As a concrete example, consider a request to find the shortest path between two nodes in a network. The breadth-first search (BFS) needed to answer this query could lead to a mammoth transaction if the network is large (e.g., nodes connected to the Internet). Moreover, application actions that appear as read-only can trigger writes. For example, a `MATCH` call is often used paired with a predicate to update a select group of nodes. In short, the flexibility and abstraction provided by the graph model causes engineers to unerringly create mammoths.

Furthermore, supernodes, with their many neighbors and high centrality, increase contention and complicate supporting mammoth transactions [60]. If a supernode is naively locked by a mammoth, concurrency over the entire graph is significantly reduced.

2.2 Mammoth Case Studies

In this section, we describe a range of case studies based on our experience with Neo4j deployments and graph usage at a large social network. We split mammoth transactions into two categories based on their read to write ratio (by writes, we refer to inserts, updates, and deletes). While some of the use cases we describe *could* be implemented without transactional semantics, application workarounds often have undesirable effects (Section 4.3).

2.2.1 Balanced Mammoths. Balanced mammoths have read and write sets of roughly equal size. Note that we include bulk inserts here since these requests typically check for existence before insertion. We summarize two application use cases below.

Large-scale deletions. Timely deletion has become increasingly important with the rise of data privacy laws, such as the EU's GDPR [12]. In the graph setting, deletion is even more challenging, given how interconnected user data is. If a social media user asks for their account and all relevant data to be deleted, the corresponding request must remove not only all data linked to their user node but also data with several degrees of connection (e.g., all their comments on their friends' posts). For isolation, all data should appear to be removed atomically (a friend should not be able to see their photos after they have deleted their account). Essentially, this deletion is a mammoth transaction touching a sizable piece of the social graph.

Systems today offer limited support for these operations. They only guarantee that data is deleted within a number of days (e.g., 90 days for Google [10] and Meta [19]). Deletions are usually broken up into multiple asynchronous jobs, increasing the risk of user-visible consistency anomalies.

Managing visibility. Next, we describe several examples based on real use cases of Neo4j. Social network users often want to constrain the visibility of certain posts. If a user wants to make all their posts from 2020 visible to only themselves, the application would need to find all these posts by traversing the social graph and changing their visibility status. Similarly, role-based access control (RBAC) systems are often modeled as graphs and encounter an analogous request pattern (though at a smaller scale): mass updates of granted/denied privileges for a given role and set of entities. In both cases, the atomicity and consistency provided by transactions are essential for these requests (e.g. once a visibility update has been acknowledged, another user should not be able to view their posts from 2020). The lack of such guarantees can allow malicious users to leverage race conditions to trigger security vulnerabilities [72].

2.2.2 Unbalanced Mammoths. These transactions contain many reads but few writes.

Identifying fake accounts. To detect anomalous social network behavior, applications need to run complex queries that touch many nodes [70, 76]. Consider a social networking application that seeks to identify bots by checking user interactions. While most of the application's requests are read-only transactions, if a social network user is found to have too much activity (likely a bot rather than a real person), this user's internal profile needs to be flagged for review, resulting in writes at the end of the long running transaction. Transactional semantics are necessary to avoid missing a fake account or mistakenly flagging it multiple times.

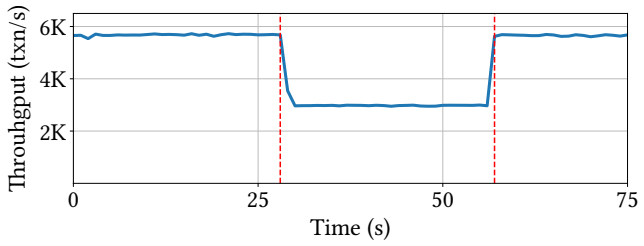

Figure 1: Red lines indicate the start/end of a balanced mammoth, which greatly hampers system throughput.

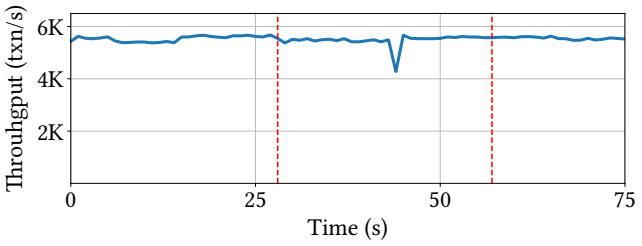

Figure 2: Red lines indicate the start/end of an unbalanced mammoth, which harms system throughput.

Pattern matching. Pattern matching is a graph-processing algorithm that also generates unbalanced mammoth transactions [56]: the application performs many reads in the process of finding relevant nodes/edges and then writes the relevant results. This request pattern is found in many real-world applications, such as fraud detection, network analytics, and access control [36, 39, 60]. Given its widespread usage, Cypher [42] and GQL [35], popular graph query languages, both provide a specific API (`MATCH`) to simplify application queries. As a concrete example, an access control manager could generate an unbalanced mammoth to verify which files in a (potentially large) project are viewable by a given user. This request must traverse over all files (while concurrent requests that add or delete files execute) and atomically authorize the user for all relevant files. Without transactional guarantees, the user may obtain incorrect file permissions.

2.3 The Lack of Benchmarks

Despite the prevalence of mammoth transactions in industry, we are aware of only a single benchmark that includes these requests. The vast majority of HTAP benchmarks [23, 24, 31, 33] combine some OLTP queries (e.g., TPC-C [68]) with the read-only portion of an OLAP workload (e.g., TPC-H [69]). Only OLxPBench [50], a recent benchmark, recognizes the need for "real-time" queries: read-write transactions with analytical reads (these are equivalent to the unbalanced mammoths that we identify in Section 2.2.2). The authors find that throughput decreases by up to 5.9× on a state-of-the-art HTAP system [49] when such requests are executed.

Among graph benchmarks, many focus on analytics or web serving without any transactions [2, 3, 22, 27, 71]. LDBC [37], a standard graph database benchmark, contains complex read queries interspersed with short read requests and insert operations. Though the benchmark recently added a cascading delete, it lacks complex read-write transactions with arbitrarily large read/write sets. TAOBench [29], a benchmark based on Meta's social graph workloads, contains a limited subset of large write transactions.

In evaluating mammoth transactions, users are sensitive to more than just peak performance. From conversations with graph database users, we find that users desire predictability, reliability, and bounded retries, which are rarely evaluated by existing benchmarks. For production systems, robustness is a crucial metric; customers expect stable performance regardless of other requests. While failures within a limited window are acceptable, large disparities in behavior are not. Towards this end, users are open to sometimes severe tradeoffs (see Section 4.3) to achieve these properties.

3 EVALUATING A MODERN GRAPH DATABASE

In this section, we evaluate the performance of Neo4j [62], a popular graph database, on balanced and unbalanced mammoth transactions. Neo4j's default isolation level is Read Committed, though applications can manually acquire explicit write locks to improve isolation. We run Neo4j Enterprise 5.4.0 and our client drivers on Azure Standard D48ds_v5 (48 vCPUSs, 192 GiB RAM) virtual machines. We use the LDBC social network dataset [74] with 10K `Person` nodes and 346K `Knows` edges for our baseline graph. Notably, the number of friendship edges (friendship degree) generated per `Person` is skewed following a power law distribution based on the Meta graph (thus, certain `Person` nodes are supernodes). Our baseline workload consists of an equal ratio of OLTP-style read-only and read-write transactions that access or update data on `Person` nodes. We add both balanced and unbalanced mammoth transactions and measure the drop in throughput.

3.1 Balanced Mammoths

Our balanced mammoth is inspired by community detection algorithms [56], which traverse part of the graph before writing back computation results. This transaction calculates a `Person`'s *importance score*, which captures their influence on the social graph. To compute this score for a given node, the transaction traverses the graph via depth-first search up to a configurable depth (three in our experiment) and records the scores of connected nodes. Then, the transaction updates the scores of all nodes in the query. This mammoth conflicts with the short read-only and read-write transactions that access and update these scores. Transactional guarantees are necessary to avoid Lost Update anomalies.

As Figure 1 shows, throughput drops by 1.9x when a *single* balanced mammoth transaction starts executing at the 28s mark. Since Neo4j locks the node of each write in a transaction, other requests struggle to make progress once the mammoth begins. After the long running request completes at 57s, the throughput of the system recovers. This slowdown would be even more severe if read locks were also acquired for Serializability. In the current setup, the relative slowdown corresponds to the ratio of read-write transactions in the workload since the mammoth blocks writes. For the experiment in Figure 1, we run a 50/50 ratio of read-only to read-write transactions. With a 10/90 ratio, we get up to a 4.7x slowdown (we omit the corresponding chart due to lack of space). While we evaluate the impact of a single mammoth transaction, performance is worse if multiple mammoths are run in parallel.

3.2 Unbalanced Mammoths

To quantify the impact of unbalanced mammoths, we augment the Read 4 query of the LDBC Business Intelligence workload [65]. This transaction focuses on finding the top commenters in online forums per country. We extend this request by adding operations at the end of the transaction to update the status of each `Person` that has been identified as a top poster.

As Figure 2 shows, Neo4j performs better on unbalanced mammoths than balanced ones as it does not hold read locks (all the reads in these transactions can proceed without blocking other requests). However, when writes execute at the end of the mammoth, there is a noticeable slowdown (1.3x) in throughput for other transactions, as many must wait until the mammoth commits.

While we quantify only throughput loss in these experiments, mammoth transactions can impact *physical resources* as well. The size of these requests can cause heavy page cache pollution, increasing latency for all subsequent reads.

3.3 Discussion

Our findings demonstrate that there is limited support for efficiently executing mammoth transactions. While modern graph databases provide some transactional semantics, no system explicitly supports mammoths. Instead, most recommend smaller transaction sizes for better performance and offer only limited isolation levels. Neo4j [62], Neptune [8], and TigerGraph [20] support Read Committed while ArangoDB [9], DGraph [11] and Memgraph [15] provide Snapshot Isolation (SI). Running mammoths at higher isolation levels is even more challenging. For instance, executing the mammoths from our experiments on Memgraph proved impossible as they *always* aborted when attempting to commit due to conflicts with concurrent short transactions.

4 EXISTING APPROACHES FOR MAMMOTH TRANSACTIONS

In this section, we describe existing approaches to handling mammoth transactions. Most research solutions require user intervention: application developers must explicitly designate a transaction as long running and often perform extra work for the underlying system to efficiently process these requests. While there are some specialized protocols that enable fast mammoths of certain types, these methods do not apply to general workloads. In practice, we find that applications developers take ad hoc approaches to supporting mammoth transactions.

4.1 User Intervention

The problem of long-lived transactions (LTTs), first introduced in 1981 [45], is well studied in relational systems. However, many of the existing solutions are impractical for graph data.

Transaction decomposition Many approaches seek to break up large transactions into smaller pieces. Nested transactions enable applications to split operations into sub-transactions that can commit or abort independently [53]. Often implemented using "savepoints" [6, 13, 16, 18], these subtransactions provide additional concurrency and reduce retries at the cost of poor behavior under high load. For instance, an excessive number of savepoints caused an outage at Amazon on Prime Day 2018 [5]. Similarly, sagas [44] allow long running transactions to run as a series of smaller transactions interleaved with other requests but require applications to provide commutative compensating transactions for transparent recovery. For large graph traversals, this constraint would incur prohibitively large overheads.

Semantic concurrency control. Other work focuses on leveraging semantic information to increase concurrency. Altruistic locking introduces a `Donate` method for applications to release locks held by a long running transaction before commit to free up resources [63]. Escrow transactions, a form of semantic concurrency control designed specifically for LLTs, enable commutative operations to execute without blocking other requests [58].

All of these approaches require the users to carefully understand how their transactions will interact with the storage system and the data, which is difficult even in the relational model. Given the use of high level graph frameworks, the lack of transparency into a graph data store makes it even harder for users to manually specify low level operation semantics.

Lazy execution. Deferred, or "lazy", execution recognizes that database state can differ from what has been promised to a client, as long as the state is reconciled when the client explicitly requests it [38, 75]. Consequently, a commit decision can be returned before a transaction is fully executed. To ensure aborts do not occur after a promise to commit is made, past work assumes the full read/write set of each transaction is available (i.e., operating on a deterministic database [38]) or requires the application to specify which operations can be deferred through a custom programming interface [75]. Both these assumptions are challenging for large graph traversals.

To the best of our knowledge, no work explicitly addresses mammoth transactions in the graph setting. Most research efforts for the relational model focus on read-only transactions [51, 75]. Mühe et al. [55] propose tentatively executing large transactions on a consistent snapshot of the system before validating writes to provide both Snapshot Isolation and Serializability. While validation works well for low contention (which the paper focuses on), high conflicts are pervasive in the graph setting due to supernodes.

4.2 Specialized Protocols

Schema changes and large-scale deletes, both of which generate mammoth transactions, are often handled using specialized solutions. Schema changes represent a restricted form of balanced mammoths with a limited number of state transitions. While modern distributed databases, such as Google's F1 [61] and CockroachDB [4], implement efficient protocols for these large-scale changes, they leverage specific definitions of consistency that do not apply to general transactions. On the other hand, large-scale deletions are primarily handled through long running asynchronous jobs that do not block ongoing requests but also offer no transactional guarantees. To trigger these deletions, developers are required to annotate their queries to specify the targeted retention periods of data [32, 64]. With the right concurrency control, we do not need to surface this complexity to applications and can efficiently remove data at the system level.

4.3 Ad Hoc in Practice

Based on interviews with Neo4j and social network engineers, we describe how mammoths are executed on production systems. Currently, application developers are confronted with two undesirable choices: 1) build ad hoc concurrency control primitives at the application level to ensure transactional isolation (e.g., use locking primitives or validation procedures outside of the data store) or 2) manually handle the data anomalies that can arise in the absence of strong isolation guarantees. These workarounds are often inefficient, hard to debug, and in general, complicate the developer process [25, 66]. For example, one common approach is to "stop-the-world" by taking a snapshot of the *entire* graph database before making it read-only. Mammoth writes are then applied to the snapshot before this updated version replaces the existing database state. Glaringly, *no writes* can occur while the mammoth executes, even on keys that the large transaction does not touch. Alternatively, developers choose to manage versioning themselves by storing metadata and filtering queries accordingly. While this approach enables more concurrency, it requires that extra predicates be applied to *all requests*. Overall, these workarounds are inefficient because they require application programmers to reason about concurrency without full visibility into the underlying system.

On the other hand, not ensuring any guarantees causes serious issues: the lack of transactional semantics can lead to data corruption. For instance, requests could easily violate reciprocal consistency [73], which requires that the data on the pair of edges between two nodes be updated atomically.

Clearly, supporting mammoth transactions is a challenging problem. In the next section, we propose approaches to achieve strong isolation for mammoths without hampering performance.

5 TOWARDS SPEEDY MAMMOTHS

Applications on graph data increasingly desire efficient mammoth transaction execution. We believe that it is possible to leverage inherent graph properties and domain-specific workloads to achieve strong isolation without sacrificing performance.

5.1 Borrowing From Relational Systems

Techniques for concurrency control on relational systems can inspire solutions for graph data.

Locking. Given how finer-grained locking has enabled greater concurrency in the relational model [46], we can naturally extend these techniques to the graph by applying locks at the structural (node and edge) and data (attribute) levels. We consider two techniques: finer-grained graph locking and graph lock escalation. Property graphs associate each node/edge with a set of property-value pairs called *attributes* [21]. For example, a `Person` node on the social graph could have three attributes: `Birthday`, `Contact_Info`, and `Location`. A mammoth transaction may only update an attribute on a node (changing a user's `Location`) or modify the structure without altering data (inserting an `Friend` edge does not affect any of the attributes on a `Person` node). Thus, the transaction needs to only lock the attributes or structures it is modifying. Based on our experience at Neo4j, changing only the structure or an attribute is a frequent request pattern. Implementing finer-grained locks will enable greater efficiency without losing transactional semantics.

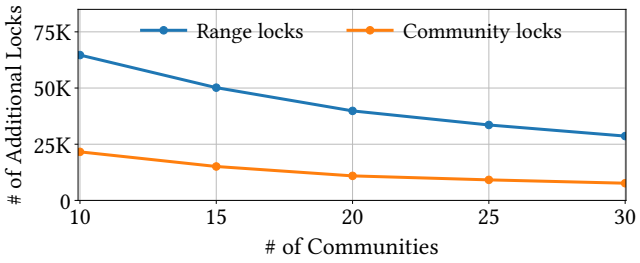

Figure 3: Community locks result in fewer total locks compared to range locks.

Lock escalation is another promising technique. In a relational system, locks are typically applied at varying granularity (e.g., row, range, table, etc.) depending on the number that needs to be acquired. For graph data, *communities* (large subgraphs of highly interconnected nodes) and *motifs* (small subgraphs often accessed together within the same query) provide straightforward boundaries around which we can implement locks. These coarser-grained locks can prevent unnecessary waits and aborts for larger requests. For instance, if a mammoth transaction happens to access several nodes in a community, it is likely that other nodes in the same community will also be accessed, so the system can escalate the individual node locks to a community level lock.

We evaluate the potential benefits of lock escalation with a simple experiment. We generate a graph with 100K nodes and assign each node to a range and a community based on its unique id. Ranges are contiguous groups of ids (e.g., the first range includes node 1 to node 99). For communities, nodes are randomly assigned to one (i.e., each community will contain a set of node ids that may not align with the sequential order of ids). With lock escalation, each operation of a transaction automatically grabs higher level locks after a threshold has been reached (e.g., after 5K locks are held in MySQL [1]). We measure how many *additional* node level locks this translates to for 10 operations after this threshold has been met (we assume 5K locks have already been acquired). We pessimistically select a probability of 85% that each operation will access a node in the same community as the previous one (for many classes of graph data, this is likely an underestimate [34]).

To measure the impact of different lock escalation strategies, we compare range and community level locks. There are an equal number of each though range locks span contiguous sets of node ids while community locks only pertain to the nodes of the community. Figure 3 shows the number of additional locks averaged over 1K transactions. Range locks always cause more locks to be held in total since they do not leverage information about access patterns (which community locks take advantage of). When there are more communities, the number of locks held decreases under both schemes since these coarse-grained locks cover fewer nodes. By leveraging graph information, we can reduce the impact of concurrency control while maintaining strong isolation.

Hotspots. Relational systems have extensively optimized for hotspots. Their analogues in the graph context are supernodes, which link to many other nodes and can affect their query performance [60]. We believe that research on mitigating hotspots for transactions [48, 67] can be extended to the graph setting. For example, LDSF, a hotspot-aware transaction scheduling algorithm,

prioritizes transactions that block many others [67]. One could design a similar algorithm that prioritizes transactions touching nodes with higher outdegrees. Addressing supernodes separately is already common practice in industry: Neo4j implements a relaxed locking algorithm for nodes with more than 50 edges [7].

5.2 Taking Advantage of Graph Features

Despite the challenges that the graph setting presents, there are also additional opportunities resulting from the graph's inherent structures and properties. As a running example, we consider the social network graph from Section 3, which has `Person` nodes and `Knows` edges to represent friendships.

Graph-specific data structures. A range of work has shown that optimizing domain-specific data structures is promising for improving concurrency control [41, 43, 77]. Graph data can be stored in a variety of formats: as an adjacency list, compressed sparse row (CSR), or matrix [26], and we can leverage this for supporting mammoths. For instance, in the restricted setting where the full read/write set of each transaction is known, Sortledton [43] presents an adjacency list-based data structure that stores sorted neighborhoods of nodes, allowing transactions to acquire locks in a consistent order and minimize deadlocks. Prior work has also shown that specialized data structures can improve performance for supernodes: GraphS, a system used at Alibaba to detect cycles in real-time on a large-scale dynamic graph, constructs a supernode index to avoid explicitly exploring all outgoing edges of these nodes when traversing the graph [60]. Such an index can also be helpful in dealing with supernodes for transactions, which may execute an optimized concurrency protocol for these nodes.

Inherent graph properties. Furthermore, the graph data model guarantees unique properties that we can leverage for faster concurrency control. By default, most graph databases provide the following consistency guarantees: no dangling edges, no duplicate edges, and the reverse edge always exists (for undirected graphs). In our social network graph, this would ensure that each `Person` has no friendships with non-existent people, no duplicate friendships, and every friend can be found from either direction. These rules are inherent to the graph data model and are supported by optimized data structures and protocols. Conveniently, they provide a strong foundation for ensuring strong isolation guarantees for mammoth transactions. For example, knowing that both edges exist can make traversal simpler and relaxes what data needs to be kept up-to-date synchronously. A query checking the friendship status on the `Knows` edges between two friends can use the more recent status if the two values from the edges differ. In the distributed setting, storing only one edge of a pair can be sufficient for traversing the partitioned graph: Meta's TAO graph data store asynchronously updates paired edges without any concurrency control since they have the same data [30].

5.3 Workload-Specific Protocols

In large part, graph-processing algorithms display distinct access patterns that can be used to optimize concurrency control. RisGraph [40] leverages the fact that the results of monotonic algorithms (e.g., breadth-first search) are only affected by certain updates to reduce the overhead of concurrency control. The authors categorize update operations as either safe or unsafe; most updates are safe and allowed to run in parallel. Extending this notion to mammoths can reduce the impact of concurrency control without affecting correctness. Moreover, incremental graph computation algorithms are also promising since not all updates need to be applied atomically for valid results. We note that this proposal differs from having application developers manually input semantic information (e.g., creating compensating transactions for sagas [44]). Instead, we imagine that the system will leverage contextual information from particular graph-processing algorithms (e.g., to automatically create compensating transactions).

6 CONCLUSION

Mammoth transactions are an important but underserved class of transactions on graph data stores. We discuss the emergence of mammoths and empirically confirm that these requests cannot execute efficiently on existing systems. We propose several paths forward to supporting mammoth transactions with high performance and strong isolation guarantees. Leveraging inherent graph data structures and properties will provide these requests with the performance and consistency they deserve.

ACKNOWLEDGMENTS

We thank Jim Webber, George Theodorakis, Xiao Shi, and our anonymous reviewers for their insightful feedback. This work is supported by NSF CISE Expeditions Award CCF-1730628, NSF GRFP Award DGE-1752814, a Meta Next-Generation Infrastructure award, and gifts from Amazon, Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, and VMWare.

REFERENCES

[1] 2012. SQL Server Lock Escalation Thresholds. https://learn.microsoft.com/en-us/previous-versions/sql/sql-server-2008-r2/ms184286(v=sql.105)?redirectedfrom=MSDN#lock-escalation-thresholds
[2] 2013. Epinions.com Benchmark in OLTP-Bench. https://github.com/oltpbenchmark/oltpbench/tree/master/src/com/oltpbenchmark/benchmarks/epinions/
[3] 2013. Twitter Benchmark in OLTP-Bench. https://github.com/oltpbenchmark/oltpbench/tree/master/src/com/oltpbenchmark/benchmarks/twitter/
[4] 2016. How online schema changes are possible in CockroachDB. https://www.cockroachlabs.com/blog/how-online-schema-changes-are-possible-in-cockroachdb/
[5] 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/amazon-move-off-oracle-caused-prime-day-outage-in-warehouse.html
[6] 2020. Nested transactions in CockroachDB 20.1. https://www.cockroachlabs.com/blog/nested-transactions-in-cockroachdb-20-1/
[7] 2021. Relationship Chain Locks: Don't Block the Rock! https://neo4j.com/developer-blog/relationship-chain-locks-dont-block-the-rock/
[8] 2023. Amazon Neptune. https://aws.amazon.com/neptune/
[9] 2023. ArangoDB. https://www.arangodb.com/
[10] 2023. Data Deletion on Google Cloud Platform. https://cloud.google.com/security/deletion/
[11] 2023. DGraph. https://dgraph.io/
[12] 2023. General Data Protection Regulation. https://gdpr-info.eu/
[13] 2023. ISO/IEC 9075-2:2016 (SQL standard on savepoints). https://www.iso.org/standard/63556.html
[14] 2023. MAGE – Memgraph Advanced Graph Extensions. https://memgraph.com/docs/mage/
[15] 2023. Memgraph. https://memgraph.com/
[16] 2023. MySQL Reference Manual 13.3.4 SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Statements. https://dev.mysql.com/doc/refman/8.0/en/savepoint.html
[17] 2023. Neo4j Graph Data Science. https://neo4j.com/product/graph-data-science/
[18] 2023. PostgreSQL 15 Documentation SAVEPOINT. https://www.postgresql.org/docs/current/sql-savepoint.html

[19] 2023. Privacy Policy. https://www.facebook.com/privacy/policy/

[20] 2023. TigerGraph. https://www.tigergraph.com/

[21] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (sep 2017), 40 pages.

[22] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.

[23] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 583–598.

[24] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. 2019. Optimal Column Layout for Hybrid Workloads. *Proc. VLDB Endow.* 12, 13 (sep 2019), 2393–2407.

[25] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1327–1342.

[26] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *arXiv preprint arXiv:1910.09017* (2019).

[27] 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 *Proceedings of the GRADES'15* (Melbourne, VIC, Australia) *(GRADES'15)*. Association for Computing Machinery, New York, NY, USA, Article 7.

[28] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, Lei Zhang, Rui Shi, Wei Ding, Kai Wu, Shangyu Luo, Jason Sun, and Yuming Liang. 2022. ByteHTAP: Bytedance's HTAP System with High Data Freshness and Strong Data Consistency. *Proc. VLDB Endow.* 15, 12 (sep 2022), 3411–3424.

[29] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. 2022. TAOBench: An End-to-End Benchmark for Social Network Workloads. *Proceedings of the VLDB Endowment* 15, 12 (2022), 1965–1977.

[30] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. 2021. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3014–3027.

[31] Fábio Coelho, João Paulo, Ricardo Vilaça, José Pereira, and Rui Oliveira. 2017. HTAPBench: Hybrid Transactional and Analytical Processing Benchmark. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (L'Aquila, Italy) *(ICPE '17)*. Association for Computing Machinery, New York, NY, USA, 293–304.

[32] Katriel Cohn-Gordon, Georgios Damaskinos, Divino Neto, Joshi Cordova, Benoît Reitz, Benjamin Strahs, Daniel Obenshain, Paul Pearce, and Ioannis Papagiannis. 2020. DELF: Safeguarding Deletion Correctness in Online Social Networks. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 60, 18 pages.

[33] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The Mixed Workload CH-BenCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems* (Athens, Greece) *(DBTest '11)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages.

[34] J.-C. Delvenne, S. N. Yaliraki, and M. Barahona. 2010. Stability of graph communities across time scales. *Proceedings of the National Academy of Sciences* 107, 29 (June 2010), 12755–12760. https://doi.org/10.1073/pnas.0903215107

[35] 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 *Proceedings of the 2022 International Conference on Management of Data*. 2246–2258.

[36] Bogdan Dumitrescu, Andra Băltoiu, and Ştefania Budulan. 2022. Anomaly Detection in Graphs of Bank Transactions for Anti Money Laundering Applications. *IEEE Access* 10 (2022), 47699–47714. https://doi.org/10.1109/ACCESS.2022.3170467

[37] 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 Qorkload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[38] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. 2014. Lazy Evaluation of Transactions in Database Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 15–26.

[39] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Jingren Zhou, Diwen Zhu, and Rong Zhu. 2021. GraphScope: A Unified Engine for Big Graph Processing. *Proc. VLDB Endow.* 14, 12 (oct 2021), 2879–2892.

[40] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-Millisecond Per-Update Analysis at Millions Ops/s. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 513–527.

[41] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. 2013. KÙZU Graph Database Management System. In *Eleventh Biennial Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2013, Online Proceedings*. www.cidrdb.org.

[42] 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 *Proceedings of the 2018 international conference on management of data*. 1433–1445.

[43] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: a universal, transactional graph data structure. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1173–1186.

[44] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. *SIGMOD Rec.* 16, 3 (dec 1987), 249–259.

[45] Jim Gray. 1988. *The Transaction Concept: Virtues and Limitations*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 140–150.

[46] Jim N Gray, Raymond A Lorie, and Gianfranco R Putzolu. 1975. Granularity of locks in a shared data base. In *Proceedings of the 1st International Conference on very large data bases*. 428–451.

[47] Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. 2019. Updating Graph Databases with Cypher. *Proc. VLDB Endow.* 12, 12 (2019), 2242–2253.

[48] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 658–670.

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

[50] 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 *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1822–1834. https://doi.org/10.1109/ICDE53745.2022.00182

[51] Jongbin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-Lived Transactions Made Less Harmful. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 495–510.

[52] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. 2022. Diva: Making MVCC Systems HTAP-Friendly. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 49–64.

[53] N. A. Lynch and M. Merritt. 1986. *Introduction to the Theory of Nested Transactions*. Technical Report. USA.

[54] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient Isolated Execution of Hybrid OLTP+OLAP Workloads for Interactive Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 37–50.

[55] Henrik Mühe, Alfons Kemper, and Thomas Neumann. 2013. Executing Long-Running Transactions in Synchronization-Free Main Memory Database Systems. In *Sixth Biennial Conference on Innovative Data Systems Research, CIDR 2013, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*. www.cidrdb.org.

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

[57] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 677–689.

[58] Patrick E. O'Neil. 1986. The Escrow Transactional Method. *ACM Trans. Database Syst.* 11, 4 (dec 1986), 405–430.

[59] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2340–2352.

[60] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-Time Constrained Cycle Detection in Large Dynamic Graphs. 11, 12 (aug 2018), 1876–1888.

[61] Ian Rae, Eric Rollins, Jeff Shute, Sukhdeep Sodhi, and Radek Vingralek. 2013. Online, Asynchronous Schema Change in F1. *Proc. VLDB Endow.* 6, 11 (aug 2013), 1045–1056.

[62] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data.* " O'Reilly Media, Inc.".

[63] Kenneth Salem, Héctor García-Molina, and Jeannie Shands. 1994. Altruistic Locking. *ACM Trans. Database Syst.* 19, 1 (mar 1994), 117–165.

[64] Subhadeep Sarkar and Manos Athanassoulis. 2022. Query Language Support for Timely Data Deletion. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang (Eds.). OpenProceedings.org, 2:429–2:434.

[65] Gábor Szárnyas, Jack Waudby, Benjamin A Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter Boncz. 2023. The LDBC Social Network Benchmark: Business Intelligence Workload. (2023).

[66] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 4–18.

[67] Boyu Tian, Jiamin Huang, Barzan Mozafari, and Grant Schoenebeck. 2018. Contention-Aware Lock Scheduling for Transactional Databases. *Proc. VLDB Endow.* 11, 5 (oct 2018), 648–662.

[68] TPC. 2010. *TPC Benchmark H, revision 5.11.* Technical Report. TPC. 1–132 pages. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[69] TPC. 2021. *TPC Benchmark C, revision 3.0.0.* Technical Report. TPC. 1–138 pages. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf.

[70] Bimal Viswanath, M. Ahmad Bashir, Mark Crovella, Saikat Guha, Krishna P. Gummadi, Balachander Krishnamurthy, and Alan Mislove. 2014. Towards Detecting Anomalous User Behavior in Online Social Networks. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 223–238.

[71] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. 2014. BigDataBench: a Big Data Benchmark Suite from Internet Services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 488–499.

[72] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 5–20.

[73] Jack Waudby, Paul Ezhilchelvan, Jim Webber, and Isi Mitrani. 2020. Preserving Reciprocal Consistency in Distributed Graph Databases *(PaPoC '20)*. Association for Computing Machinery, New York, NY, USA, Article 2, 7 pages.

[74] Jack Waudby, Benjamin A. Steer, Arnau Prat-Pérez, and Gábor Szárnyas. 2020. Supporting Dynamic Graphs and Temporal Entity Deletions in the LDBC Social Network Benchmark's Data Generator. In *GRADES-NDA'20: Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Portland, OR, USA, June 14, 2020*, Akhil Arora, Semih Salihoglu, and Nikolay Yakovets (Eds.). ACM, 8:1–8:8. https://doi.org/10.1145/3398682.3399165

[75] Lesley Wevers, Marieke Huisman, and Maurice van Keulen. 2016. Lazy Evaluation for Concurrent OLTP and Bulk Transactions. In *Proceedings of the 20th International Database Engineering &; Applications Symposium* (Montreal, QC, Canada) *(IDEAS '16)*. Association for Computing Machinery, New York, NY, USA, 115–124.

[76] Cao Xiao, David Mandell Freeman, and Theodore Hwa. 2015. Detecting Clusters of Fake Accounts in Online Social Networks. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security* (Denver, Colorado, USA) *(AISec '15)*. Association for Computing Machinery, New York, NY, USA, 91–101.

[77] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (mar 2020), 1020–1034.