Preserving Reciprocal Consistency in Distributed Graph Databases

Jack Waudby
j.waudby2@ncl.ac.uk
Newcastle University
Newcastle, UK

Paul Ezhilchelvan
paul.ezhilchelvan@ncl.ac.uk
Newcastle University
Newcastle, UK

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

Isi Mitrani
isi.mitrani@ncl.ac.uk
Newcastle University
Newcastle, UK

Abstract

In our earlier work, we identified *reciprocal consistency* as an important constraint specific to graph databases. If it can be lost even with a negligible small probability, subsequent inconsistent reads followed by writes can corrupt a distributed graph database within a time period extremely short relative to database lifetime. Reciprocal consistency can of course be maintained as a part of enforcing any known isolation guarantee incurring well established performance costs. However, in practice distributed graph databases are often built atop BASE databases with no isolation guarantees, profiting from increased performance but leaving them susceptible to rapid corruption. A lightweight concurrency control protocol ensuring reciprocal consistency is presented, catering for application programmers that are interested in maintaining performance and the structural integrity of their distributed graph database. Protocol performance is evaluated through simulations.

CCS Concepts. • **Data Management → Graph Databases**; *Reciprocal Consistency*; *Concurrency Control.*

Keywords. Graph Databases, Reciprocal Consistency, BASE

ACM Reference Format:
Jack Waudby, Paul Ezhilchelvan, Jim Webber, and Isi Mitrani. 2020. Preserving Reciprocal Consistency in Distributed Graph Databases. In *PaPoC '20: Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, April 27, 2020, Heraklion, Crete, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/1122445.1122456

PaPoC '20, April 27, 2020, Heraklion, Crete, Greece

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

1 Introduction

Recent years have seen a proliferation in the use of graph processing technologies [9]. Application areas are wide reaching from healthcare, to social networks and fraud detection [10]. Graph databases model data as a *property graph* [15], vertices represent entities and edges represent the relationships between entities. In addition, properties can be stored on both vertices and edges. In the storage layer, edges are represented by two reciprocal pointers, one stored with each vertex the edge connects. This allows for bi-directional traversal and improved query performance [15]. An edge is said to be *reciprocally consistent*, if its two end pointers are mutually reciprocal of each other (details in Section 2).

In practice, graphs can be extremely large, sometimes in the magnitude of 100 billion edges [16], exceeding the storage capacity of a single-node graph database and motivating the need for distributed graph databases. A common distributed graph database design pattern is to first partition graph data over several machines in a cluster; resulting in a number of *distributed edges*, an edge's reciprocal pointers reside in different partitions. Recent work [11] and [17] highlighted that violations of reciprocal consistency in distributed edges introduce corruption into the database. Moreover, due to the *Scale-Free* [3] property exhibited by many real world graphs, this corruption can propagate through the database at alarmingly rates.

When, for example, a BASE database [14] is adapted with a graph processing layer, then violations of reciprocal consistency will occur if that adaptation provides no concurrency control for operations that span partitions in order to offer higher performance. This paper proposes a simple concurrency control protocol, called Delta protocol, that does not impede performance adversely. That is because the protocol is exclusively designed for one purpose only: reciprocal consistency in distributed edges. Its design leverages the fact that the updating of end pointers of a given distributed edge immediately follow each other and the small interval between them is the window for possible conflicts between

concurrent updates. The protocol ensures that any two updates on any given end pointer of any given edge are at least Δ time apart by aborting any write attempt that is within . It ensures reciprocal consistency when Δ is chosen to be larger than the length of the conflict window.

The paper is organized as follows: Section 2 presents reciprocal consistency, Sections 3-4 discuss the architecture of distributed graph databases and describes how corruption occurs. Section 5 outlines the Delta protocol. Sections 6-7 presents the model used to evaluate protocol performance.

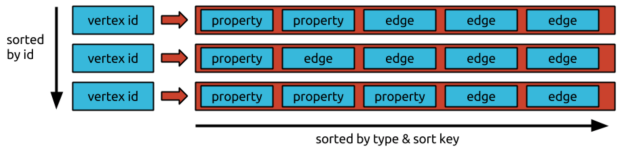

Figure 1. Database records representing vertices, containing properties and a sequence of edges that comprise the adjacency list of a vertex [1].

2 Reciprocal Consistency

In the property graph data model edges have direction, each edge having a pair of *source* and *destination* vertices. In the storage layer, edge information is stored with **both** the source and destination vertices. This, as we explain next, allows an edge to be traversed in both directions and thus facilitates improved query performance.

A common approach to storing graphs (arising from JanusGraph [1] and TitanDB [4]) is given in Figure 1. A vertex is represented by a record that contains one or more vertex properties and a sequence of edge pointers (each shown simply as edge in Figure 1) pointing to other vertices. This sequence of edge pointers is referred to as an adjacency list.

Consider, for example, the statement that Tolkien *wrote* The Hobbit. It is expressed using vertices *a* and *b*, for Tolkien and The Hobbit respectively, and an edge *wrote* running from *a* (source) to *b* (destination).

Using openCypher [2] this can be represented by:

```
MATCH (a:Person), (b:Book)
WHERE a.name = 'Tolkien' AND b.title = 'The Hobbit'
CREATE (a)-[w:WROTE]->(b)
```

Adjacency lists of both *a* and *b* record information about the edge and this information is reciprocal (or inverse) of each other: *a*'s list will indicate '*a wrote b*' while *b*'s will have '*b written* by *a*', Figure 2. Thus, a query 'list all titles by the author who wrote The Hobbit' can be answered starting at (destination vertex) *b* and then traversing to (source vertex) *a*, even though the edge is "directed" from *a* to *b* at model level abstraction. When the adjacency list entries for a given edge are mutually compatible like this, that edge is said to be *reciprocally consistent*, a form of referential integrity. A query can read either the source or destination vertex and is able to reify the edge correctly, returning consistent results.

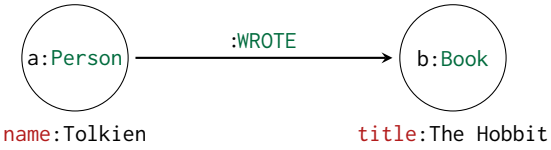

Figure 2. A reciprocally consistent edge.

3 Distributed Graph Databases

A distributed graph database employs a shared-nothing architecture, partitioning a graph between a number of loosely cooperating servers. Graph partitioning is non-trivial and a common approach is to use a *k*-balanced edge cut [13]. The objective of such an approach is to minimize the proportion of edges that span partitions in a manner that balances the distribution of vertices to partitions. Figure 3 depicts a graph database partitioned across 3 servers, $S1$, $S2$ and $S3$. Intra-partition edges are referred to as *local edges* and inter-partition edges are referred to as *distributed edges* (shown with dashed lines in Figure 3). The proportion of distributed edges is always non-negligible ranging from 25-75% [13].

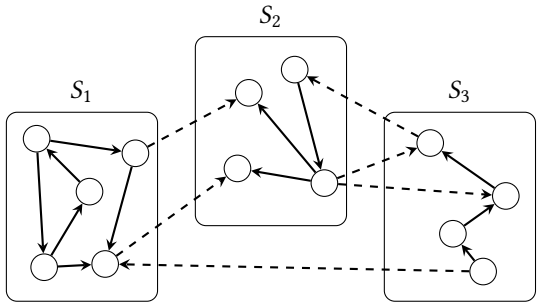

Figure 3. Local and distributed edges

Adjacency lists can now contain edge pointers to vertices on remote servers. Maintaining reciprocal consistency for distributed edges is challenging - especially given an architecture employed by contemporary distributed graph databases ([1], [4]). Often an existing BASE database is used for storage, which is then adapted with a query language expressed in terms of edges and vertices along with some gluecode to bind that interface to the underlying database - we refer to such systems as BASE distributed graph databases[1]. Superficially, opting for this design appears to be a good choice: the application programmer has the modeling convenience of graphs with the operational characteristics from the underlying BASE database.

[1]Typically, each partition is replicated for fault tolerance and availability, these issues are beyond the scope of this paper.

However, the problem with this design is the (lack of) transactional semantics is inherited from the underlying database. BASE databases seldom provide guarantees for multi-operation, multi-object transactions that span partitions. This lack of concurrency control across partitions makes it possible for concurrent updates to interleave in a manner that violates reciprocal consistency of distributed edges[2]. Earlier work investigated how the lack of concurrency control across partitions can undermine reciprocal consistency of distributed edges, causing irreversible corruption that spreads at alarmingly rates ([11], [17]). This process is explained in Section 4.

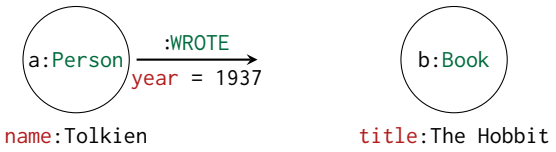

Figure 5. A half-corrupted edge.

4 Corruption in the Absence of Concurrency Control

When a given transaction writes a distributed edge it must infact perform two writes, writing reciprocal entries in the adjacency lists at the source and the destination vertices, which say reside on servers S_i and S_j respectively. Without concurrency control, transaction's writes to a distributed edge can interleave producing a distributed edge in a *half-corrupted* state - reciprocal consistency has been violated. Possible interleavings of concurrent writes to a distributed edge are given in Figure 4. A half-corrupted edge is an example of a *dirty write* (ANSI *P0* [8], Adya *G0* [5]), which is proscribed by the weakest ANSI isolation level **Read Uncommitted**. Under Read Uncommitted the database ensures a total order on transactions, consistently ordering writes from concurrent transactions, which would prevent all interleavings in Figure 4.

For such half-corrupted edges there exists a correct and a incorrect entry. Note, the order in which the two writes take place is not constrained. For example, when updating an edge between a and b it is equally likely to update a then b as it is to update b then a. A graph with half-corrupted edges has suffered *structural corruption*.

Now, if subsequent transactions read the incorrect entry of a half-corrupted edge and write further edges, *semantic corruption* has been introduced into the database. Further semantic corruption spreads by the same mechanism. A database is said to be *operationally corrupt* when a significant proportion of its data records are in a semantically corrupted state, rendering the database of little practical use. To illustrate the process of corruption, consider two transactions T_x and T_y. T_x deletes the *wrote* edge and T_y appends a property *year*:

```
// Tx
MATCH (a:Person)-[w:WROTE]->(b:Book)
WHERE a.name = 'Tolkien' AND b.title = 'The Hobbit'
DELETE w

// Ty
MATCH (a:Person)-[w:WROTE]->(b:Book)
WHERE a.name = 'Tolkien' AND b.title = 'The Hobbit'
SET w.year = 1937
```

Any interleaving in Figure 4 will result in the half-corrupted edge displayed in Figure 5. Assume, the correct ordering of transactions is T_y then T_x and the edge between a and b should still exist. The following transaction, T_z, introduces semantic corruption if it starts at b and adds a new *wrote* edge to an "unknown author" vertex. Further corruption spreads by the same mechanism.

```
// Tz
MATCH (b:Book),(u:Person)
WHERE NOT (:Person)-[:WROTE]->(b:Book)
AND u.name = 'unknown'
CREATE (u)-[:WROTE]->(b)
```

5 Delta Protocol

A straightforward solution to preventing dirty writes is for transactions to take long duration write locks [8], releasing them only once the acquiring transaction has committed or aborted. To prevent deadlock a policy such as `NO_WAIT` deadlock detection is used, which was shown to be the optimal policy in a distributed, partitioned database [12].

Our Delta protocol employs principles behind all these well-tested strategies but has two crucial differences:

- No locks are used.
- A write operation need not await until the preceding write commits but can proceed if at least Δ duration (measured in local clock) has elapsed.

These differences lead to several advantages in the context of graph databases. Firstly, a subset of edges are traversed and modified with a high frequency e.g. critical sections of motorway in a road network, leading to high contention. Secondly, graph transactions tend to be longer-lived than transactions in relational databases. Waiting for earlier writes on edges by long running transactions to commit, these frequently accessed edges significantly limit concurrency and reduce throughput. With these concerns in mind we developed the Delta protocol, which aimed at preventing edges becoming

[2]When operating without concurrency control it is equally possible that local edges become reciprocal consistent, however primitives provided by BASE databases are typically sufficient to ensure reciprocal consistency for local edges.

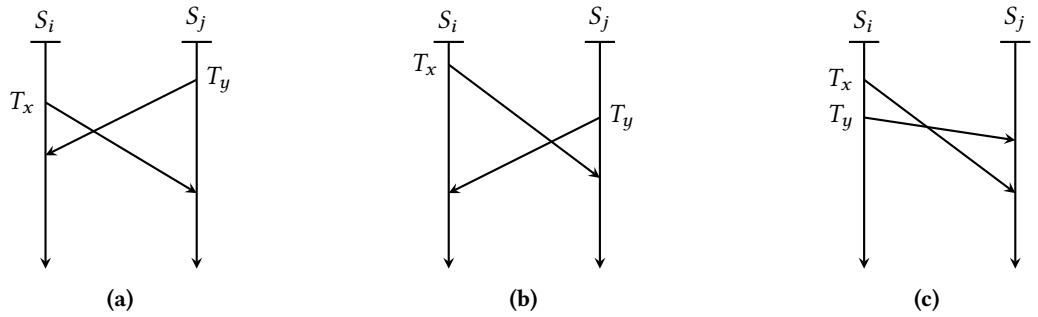

Figure 4. Possible interleavings of concurrent transaction's writes to a distributed edge spanning servers S_i and S_j by transactions T_x and T_y. In (a) T_y begins writing to the distributed edge before T_x, in (b) the converse is true, else they are equivalent. In (c) both transactions begin writing at the same server but overlap in the network and arrive out-of-order.

half-corrupted and hence quashing the seed of corruption whilst keeping performance at an acceptable level.[3]

5.1 Protocol Description

The Delta Protocol has five rules:

1. A transaction's update on an end pointer of an edge is initially tentative which would become permanent only if that transaction is permitted to commit.
2. A tentative write is possible if the end pointer is either in a permanent state or the immediately preceding tentative update was done at least Δ time before (where time is measured as per local clock).
3. If a transaction successfully performs all its tentative updates, then it is permitted to commit; otherwise, it must abort.
4. A transaction commits, when all its tentative updates are made permanent, e.g., using an atomic commitment protocol.
5. Tentative updates of an aborting transactions are ignored. An ignored tentative update can make a new transaction abort for up to Δ time after it was created; it is harmless thereafter and can be garbage collected at any time.

5.2 Correctness Reasoning

Let us define δ as the bound estimate on the interval that may elapse between a transaction updating one end of a distributed edge at one server and updating the other end of the same edge at another server.

Let Δ be chosen such that $\Delta > \delta$. Now consider the interleaving in Figure 4(a) and let t_x be the (global) time when T_x starts at server S_i; similarly t_y be the time T_y starts at server S_j. Note $t_y < t_x$, i.e. $(t_x - t_y) > 0$.

Say, T_y reaches S_i at time $t_y + d$, where d is the actual time elapsed between completing a tentative write at one end and starting at another end, let us assume that $d \leq \delta$. When T_y arrives at S_i it will find a tentative write done at time t_x. In this case, $t_y + d - t_x = d - (t_x - t_y) < d \leq \delta < \Delta$; so, T_y will abort, preventing writes from interleaving and half-corrupting the edge. A similar arguments can be made for the scenario in with Figure 4(b). For the interleaving in Figure 4(c), if $t_y - t_x > \Delta$, then T_y cannot overtake T_x at S_j.

Assume that $d > \Delta$, i.e., the estimate δ does not hold at this moment. In Figure 4(a) interfering updates are avoided only if $t_y + d - t_x = d - (t_x - t_y) < d \leq \delta < \Delta$ otherwise, an edge becomes half-corrupted. This holds for the interleaving in 4(b). Thus, in the extreme case $t_x = t_y$, reciprocal consistency is not guaranteed if $d > \Delta$ when the former exceeds its upper bound estimate δ. For Figure 4(c), if d for T_x is larger than Δ, reciprocal consistency can be violated.

This protocol is strictly weaker than Read Uncommitted isolation, ensuring only writes to distributed edge are totally ordered, preserving reciprocal consistency. The benefit of this approach is reduced contention on frequently accessed distributed edges, as the time a transaction has exclusive access to a given distributed edge is decreased.

In summary, the Delta protocol attempts to prevent the occurrence of half-corrupted distributed edges by aborting transactions but if Δ is exceeded edge can become half-corrupted and process of corruption described in Section 4 can still occur. Two questions naturally arise regards the performance of the protocol. For different values of Δ:

- Given distributed edges can still become half-corrupted if Δ is exceeded, by how much time does the protocol increase the time until operational corruption?
- How many transactions are aborted as a result of the protocol?

6 Modeling

To assess the protocols impact on time to operational corruption the model developed in [11] was extended[4]. A summary

[3] The Delta protocol is solely a concurrency control mechanism for distributed edges, guarantees about vertices, local edges are beyond the scope of this paper.

[4] An implementation and empirical evaluation in an existing systems was not performed as such an evaluation would have been impractical (need to

of the model is now provided, before discussing the extensions.

The system processes transactions that arrive in a Poisson stream with rate λ per second. To simplify the model, each transaction contains a random number of read operations, K, followed by a single write. To model a scale-free graph, edges in the database are divided into T types, popular edges types have higher access probabilities but are a smaller proportion of the total number of edges, N. For each type, a fraction f are distributed edges and the remainder are local edges. The network delay between servers was assumed to be exponentially distributed with rate ρ. At any moment in time an edge can be in one of four states:

1. Local and clean.
2. Distributed and clean.
3. Half-corrupted distributed edge arising from a dirty write.
4. Semantically corrupted.

Probabilities are then derived for a given read operation returning a correct answer (states 1, 2 or the correct record in state 3) and all the reads by a given transaction returning correct answers. Then the probability of edge becoming half-corrupted q_i, by a given transaction arriving at time t and operating on edge of type i is derived. These probabilities are used to construct transition rates $a_{i,j}$ between states, which are used to simulate the process of corrupting the database and obtain estimates for the time to operational corruption, U_γ. At time 0, all edges are clean (free from corruption), when a certain fraction, γ, of all edges become semantically corrupted, the database itself is said to be operationally corrupt. The reader is directed to [11] and [17] for a granular discussion of the initial model.

The Delta protocol impacts the rate of corruption by reducing the probability a transaction corrupts a distributed edge, q_i^{new}. Of interest, therefore, is: how large or small is the value of U_γ for a given value of γ under the Delta protocol? Moreover, to quantify the number of aborts, a second simulation which focused specifically on the subset of frequently accessed distributed edges was performed. The answers to these questions depends on several parameters characterizing the following systemic aspects:

- *Database Size.* Size is expressed by the total number of edges N, and the fraction f of distributed edges.
- *Workload.* Measured as transactions per second (TPS). Significant for measuring U_γ are: the fraction of this load that writes after reads and the number of reads that precede a write.
- *Distributed Write Delays and Choosing* Δ. The smaller the delays the less likely the bound Δ is violated. Conversely, smaller Δ is the more likely the bound Δ is violated.

7 Evaluation

The graph analyzed consisted of seven edge types, $N_1 = 10^4, N_2 = 10^5, N_3 = 10^6, N_4 = 10^7, N_5 = 10^8, N_6 = 10^9, N_7 = 10^{10}$, totaling 11 billion edges, with access probabilities $p_1 = 0.5, p_2 = 0.25, p_3 = 0.13, p_4 = 0.06, p_5 = 0.03, p_6 = 0.02$ and $p_7 = 0.01$; a graph of this size would have approximately 1 billion vertices. The number of read operations per query is geometrically distributed starting at 2, with an average of 15, before a write. In all edge types, a fraction 0.3 are distributed, the remainder are local; in proportion with a good graph partitioning algorithm. The network delay between servers is exponential distributed with a rate of $200ms$. The database is initial clean and considered to be corrupted when 10% ($\gamma = 0.1$) of all edges are corrupted. The time taken until operational corruption U, is measured in days. U considered for a range of transaction arrival rates, $\lambda = (1000, ..., 10000)$; a typical graph workload comprises of 90% read-only transactions and 10% read-write transactions [6], hence the chosen range reflects a total workload of $\lambda = (10000, ..., 100000)$. The following Δ values were considered $\Delta = 50, 75, 100ms$.

The results for measuring the impact of Δ on the time until operational corruption are given in Figure 6. Under no isolation, U ranges between 500-50 days. For $50ms$ this increases to 74-1 years. For $\Delta = 75, 100ms$ the time to corruption vastly exceeds lifetime of any system making data corruption resulting from half-corrupted edges of little practical concern.

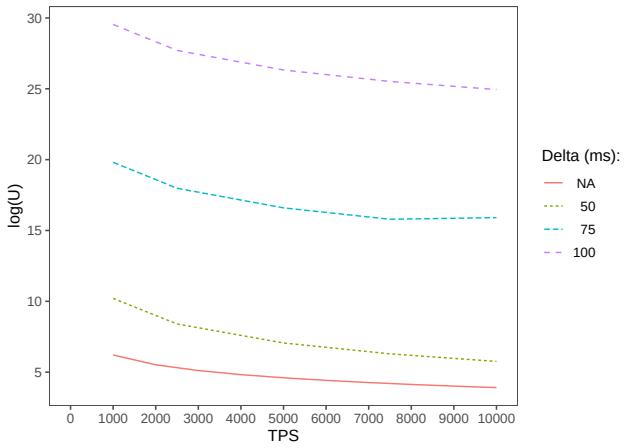

Figure 6. Time, log(U), until operational corruption ($\gamma = 0.1$) for $\Delta = 50, 75, 100ms$.

The abort rates for $\Delta = 50, 75, 100ms$ are given in Figure 7 for the most popular edge type, $N = 10^4$. The simulation

compare database state at end of experiment with the linearizable truth), slow (real time) and expensive (requiring many hours of storage and compute time)

was ran for 10 seconds for a range of transaction arrival rates, $\lambda = (1000, ..., 10000)$. For $\Delta = 50$ the abort rate varies between $1 - 5\%$, this increases to between $1 - 7\%$ and $1 - 9\%$ for $\Delta = 75$ and $\Delta = 100$ respectively.

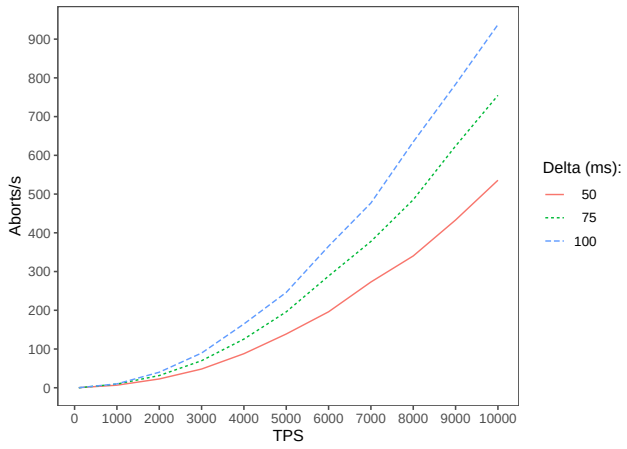

Figure 7. Aborts per seconds for $\Delta = 50, 75, 100ms$.

8 Conclusions

Database concurrency control has been a long researched area, however to the best of our knowledge this is first attempt at a developing protocol specific for distributed graph databases. We presented a lightweight protocol for providing reciprocal consistency and mitigating the problem of high contention in a distributed graph database. The Delta protocol leverages the fact writes to distributed edges always consists of two sequential writes to entries in the adjacency lists of vertices the edge connects. The protocol provides guarantees weaker than Read Uncommitted isolation (the weakest ANSI isolation level). However, such a mechanism is believed to be valuable in practice, given the popularity of BASE distributed graph databases and the rate at which corruption can spread if left unchecked. Simulations indicate the protocol rules out corruption resulting from half-corrupted distributed edges in realistic database lifetime, with the abort rate being in a reasonable range. For future work, we intend on implementing the protocol to assess the validity of the simulations and measure performance against a BASE distributed graph database operating without any concurrency control. Moreover, we plan on investigating the suitability of higher isolation levels in a distributed graph database, **Read Atomic** isolation [7] seems particularly well suited.

References

[1] 2020. JanusGraph Documentation. http://janusgraph.org/.
[2] 2020. openCypher Documentation. https://www.opencypher.org.
[3] 2020. Scale-Free Networks. https://en.wikipedia.org/wiki/Scale-free_network.
[4] 2020. TitanDB Documentation. https://titan.thinkaurelius.com.
[5] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 67–78.
[6] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep-Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). arXiv:2001.02299 http://arxiv.org/abs/2001.02299
[7] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2014. Scalable atomic visibility with RAMP transactions. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 27–38. https://doi.org/10.1145/2588555.2588562
[8] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 1–10. https://doi.org/10.1145/223784.223785
[9] 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. (2019). arXiv:cs.DB/http://arxiv.org/abs/1910.09017v1
[10] Emil Eifrem. 2016. Graph databases: The key to foolproof fraud detection? *Computer Fraud & Security* 2016 (03 2016), 5–8. https://doi.org/10.1016/S1361-3723(16)30024-0
[11] Paul D. Ezhilchelvan, Isi Mitrani, and Jim Webber. 2018. On the Degradation of Distributed Graph Databases with Eventual Consistency. In *Computer Performance Engineering - 15th European Workshop, EPEW 2018, Paris, France, October 29-30, 2018, Proceedings (Lecture Notes in Computer Science)*, Rena Bakhshi, Paolo Ballarini, Benoît Barbot, Hind Castel-Taleb, and Anne Remke (Eds.), Vol. 11178. Springer, 1–13. https://doi.org/10.1007/978-3-030-02227-3_1
[12] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *PVLDB* 10, 5 (2017), 553–564. https://doi.org/10.14778/3055540.3055548
[13] Jiewen Huang and Daniel Abadi. 2016. LEOPARD: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs. *PVLDB* 9, 7 (2016), 540–551. https://doi.org/10.14778/2904483.2904486
[14] Dan Pritchett. 2008. BASE: An Acid Alternative. *ACM Queue* 6, 3 (2008), 48–55. https://doi.org/10.1145/1394127.1394128
[15] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data.* " O'Reilly Media, Inc.".
[16] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing. *Proceedings of the VLDB Endowment* 11, 4 (2017), 420–431.
[17] Jim Webber, Paul Ezhilchelvan, and Isi Mitrani. 2. Modeling Corruption in Eventually-Consistent Graph Databases. (2). arXiv:cs.DB/http://arxiv.org/abs/1904.04702v1