Preserving Reciprocal Consistency in Distributed Graph Databases

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

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

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

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

Abstract

In this paper the notion of *reciprocal consistency*, a important feature specific to graph databases, is formalized. If reciprocal consistency is not enforced during transaction processing, a distributed graph database can become operationally corrupt within a short time period relative to database lifetime. Reciprocal consistency can be maintained as a part of enforcing any of several known strong isolation guarantees (e.g., Snapshot Isolation) incurring well established performance costs. However, in practice systems are often deployed with weaker consistency, this paper describes a protocol that guarantees only reciprocal consistency. Catering for application programmers that are interested only in ensuring reciprocal consistency to preserve the integrity of a distributed graph database. The protocol is probabilistic ensuring reciprocal consistency provided certain conditions hold. The rate of corruption when these conditions no longer hold is investigated.

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

Keywords. Graph Databases, Reciprocal Consistency

ACM Reference Format:
Paul Ezhilchelvan, Isi Mitrani, Jack Waudby, and Jim Webber. 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 [7]. Application areas are wide reaching from healthcare, to social networks and fraud detection [8]. Graph databases model data as a *property graph* [11], vertices represent entities and edges represent the relationship between entities. In addition, properties can be stored on both vertices and edges.

In practice graphs can be extremely large, sometimes in the magnitude of 100 billion edges [12], exceeding the storage capacity of a single-node database management system (DBMS). An approach to storing large graphs is to partition graph data over several machines in a cluster. Moving from a single-node DBMS to one partitioned and distributed across servers introduces several interesting problems regarding data consistency. Graphs place a high demand on the underlying database and maintaining structural integrity across partitions - called *reciprocal consistency* - in a distributed graph database is non-trivial.

In order to preserve reciprocal consistency, one could implement a heavyweight coordination protocol across partitions providing Serializability or Snapshot Isolation, which have associated performance costs. To avoid these costs DBMSs provide application developers with the option of several weaker isolation models [6]. In addition, there are a number of systems that eschew transactional guarantees altogether, typically offering eventual consistency [5], in order to achieve higher performance.

Recent work [9] and [13] investigated the use of eventually consistent semantics in a distributed graph database and highlighted how violation of reciprocal consistency can occur under weak isolation. Also, due to the *Scale-Free* [3] nature of some graphs corruption can propagate through the database at alarmingly rates.

The contribution of this paper is the description of a probabilistic protocol that focuses only on ensuring reciprocal consistency in order to preserve structural integrity of a

partitioned graph database in a transaction processing context. The protocol's impact on the rate of corruption and its performance is evaluated and discussed.

The key finding is whilst initial corruption of clean records is a rare event, due to the scale-free topology corruption propagates alarmingly quick even for modest transaction arrival rates. This motivates the use of deterministic reciprocal consistency protocols or the use of protocols with stronger isolation guarantees, which subsume reciprocal consistency.

2 Reciprocal Consistency

In the property graph data model edges have direction, there is a *source* and a *destination* vertex. In the storage layer, edge information is stored with **both** the source and destination vertices. This facilitates bi-directional edge traversal and allows for better query performance.

A common approach to storing graphs (arising from JanusGraph [2] and TitanDB [4]) is for records to represent vertices containing both data values and an adjacency list containing edge *pointers* to other vertices, Figure 1. In this representation an edge has *reciprocal* entries in the adjacency lists of the vertices the edge connects. A query reading either the source or destination vertices should be able to reify the edge correctly, returning consistent results. When the adjacency lists for vertices are mutually compatible like this, they are said to be *reciprocally consistent.*

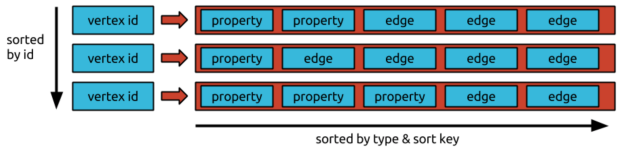

Figure 1. Vertex records containing data values and adjacency lists [2].

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).

```
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 mutually reciprocal (or inverse) of each other: *a*'s list will indicate '*a wrote b*' while *b*'s will have '*b written* by *a*'. Thus, a query 'list all titles by the author who wrote The Hobbit' can be answered by landing at *b* and then traversing to *a*; even though the edge is a directed edge at model level abstraction.

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 [10]. 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. Intra-partition edges are referred to as *local edges* and inter-partition edges are referred to as *distributed edges*, Figure 2. The proportion of distributed edges is highly data dependent. However, the number of such edges is always non-negligible ranging from 25-75% [10].

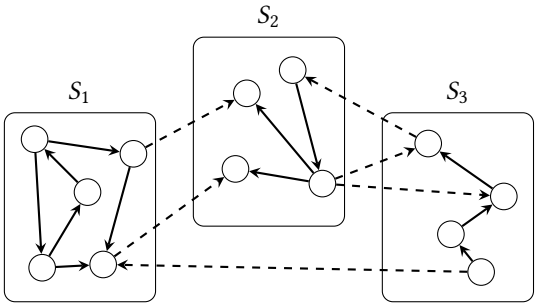

Figure 2. A graph partitioned across $k = 3$ servers in a cluster. Dashed lines represent distributed edges and solid lines represent local edges.

In a distributed graph database, adjacency lists can contain pointers to edges on remote servers. This makes maintaining distributed edge reciprocal consistency more challenging - especially given a common architecture employed by contemporary distributed graph databases. Often they use an existing eventually consistent database (e.g. Apache Cassandra [1]) to store data, which has been adapted with a programmatic API or query language expressed in terms of edges and vertices along with some gluecode to bind that interface to the underlying database. Superficially, opting for this design appears to be a good choice: the user has the modeling convenience of graphs with the operational characteristics from the underlying database. However, the problem with this design is the (lack of) transactional semantics are inherited from the underlying store. For example, Apache Cassandra provides no way of preventing updates from mutual interference in its normal multi-partition use case. Earlier work displayed how weak isolation undermines distributed edge reciprocal consistency, causing irreversible corruption that spreads at alarmingly rates [9], [13].

4 Database Corruption

We now explain how corruption occurs under weak isolation and propagates.

When a given transaction writes a distributed edge it performs two distinct sub-operations, writing reciprocal information at the source and the destination vertices, which say reside on servers S_i and S_j respectively.

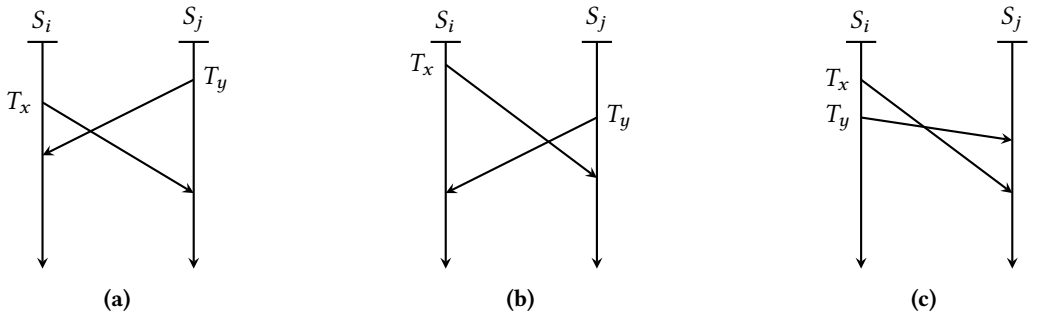

Figure 3. *Conflict Scenarios.* Possible interleavings between concurrent writes to a distributed edge spanning servers S_i and S_j by transactions T_x and T_y. In (a) T_y begins writing to the distributed edge before T_x, in (b) the converse is true, else they are equivalent. In (c) both transactions begin writing at the same server but overlap in the network and arrive out-of-order.

In a distributed graph database operating with weak isolation, transactions that concurrently modify the same distributed edge can *conflict*. That is transaction's sub-operations interleave producing a distributed edge in a *half-corrupted* state - reciprocal consistency has been violated. For such edges there exists a correct and a incorrect entry. Figure 3 outlines the conflict scenarios that produce half-corrupted edges. Note, the order in which sub-operations take place are 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.

If subsequent transactions read the incorrect entry 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 this, 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 3 will result in the half-corrupted edge displayed in Figure 4.

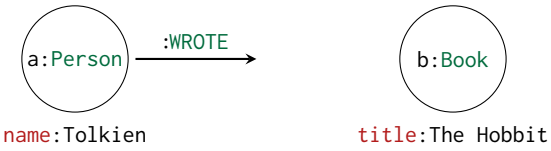

Figure 4. A half-corrupted edge resulting from conflicting transactions.

5 *Delta* Protocol

As stated earlier stronger isolation level such as serializability and snapshot isolation subsumes reciprocal consistency. The motivation between the *Delta* protocol was to develop a lightweight protocol permitting weaker isolation, reducing coordination and improving performance. It is imagined this protocol could be "bolted onto" a distributed graph database employing a NoSQL store with weak isolation for storage.

The *Delta* protocol is designed to mitigate the occurrence of half-corruption and as a result prevent operational corruption. It is assumed database servers accept transactions from clients and execute them on their behalf. Transactions may consist of a series of read and write operations to any object in the database. However, the *Delta* protocol is only invoked when a given transaction writes a distributed edge. Therefore, this protocol can be bolted onto existing mechanisms that provide guarantees within partitions but not across database partitions.

In order to prevent all interleavings in Figure 3 the *Delta* protocol adopts a probabilistic approach. It is assumed there is a known Δ, that reflects the bound on server communication delays, with a probability $(1 - \epsilon)$ that Δ is violated. All writes by sub-operations are temporary and a sub-operation is allowed to proceed provided there is no other temporary sub-operation preceding it within Δ; measured as per the local clock time. Figure 5 shows an instance of the protocol preventing Figure 3 (c), T_x overtakes T_y but is aborted when arriving at S_j. An aborted transaction, aborts any and every previous tentative write that it may have successfully completed. A transaction that succeeds at all its tentative write attempts, decides to commit when it completes its all operations; the transaction commits once all its tentative writes are committed. This protocol avoids all conflict scenarios in Figure 3 provided the bound Δ is not violated.

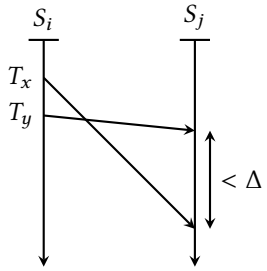

Figure 5. An example of the *Delta* protocol preserving reciprocal consistency.

However it is possible that the bound Δ is violated. If this happens all three conflict scenarios can still occur, resulting in half-corrupted edges and the spread of semantic corruption. Figure 6 displays an example when the *Delta* protocol fails to maintain reciprocal consistency.

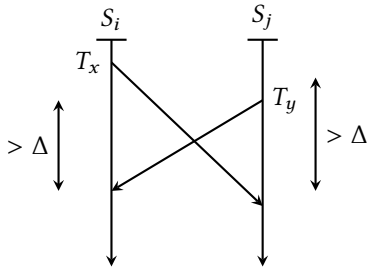

Figure 6. An example of the *Delta* protocol failing to preserve reciprocal consistency.

In short, the protocol attempts to avoid corruption by aborting transactions but cannot do so in all circumstances. Two questions naturally arise regards the performance of the protocol i) how does the protocol impact on the time until operational corruption? ii) how many transaction are aborted as a result of the protocol?

6 Modeling

In order to answer i) the model developed in [9] and fine-tuned in [13] was extended to measure the impact of the *Delta* protocol[1]. A summary of the model is now provided, before discussing the extensions.

Transactions arrive in a Poisson stream with rate λ per second. Each transaction contains a random number of read operations, K, followed by a single write. 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.

[1]An empirical evaluation of existing systems was not performed as such an evaluation would have been impractical (need to compare database state at end of experiment with the linearizable truth), slow (real time) and expensive (requiring many hours of storage and compute time)

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 interleaved updates.
4. Semantically corrupted.

The valid state transitions are given in Figure 7. Note, only distributed edges can be in state 2, but any edge, including local ones, can be in state 3.

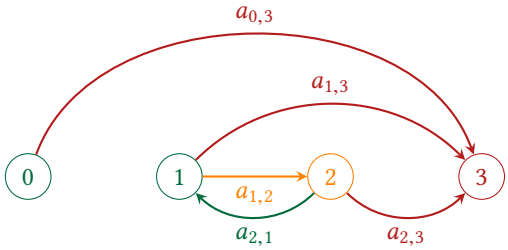

Figure 7. Edge transitions between clean, half-corrupted and semantically corrupt states.

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 conflict q_i, of 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 average time to corruption, U_γ. At time 0, all edges are clean (free from corruption). When a certain fraction, γ, of all edges become corrupted, the database itself is said to be operationally corrupt. Note, the model assumes crash-free hardware and bug-free software for simplicity. The reader is directed to [9] and [13] for a granular discussion of the initial model.

The Δ protocol influences the rate of corruption reducing the conflict probability, q_i. To simplify the derivation of the new conflict probability it was assumed. This leaves the Figure 6 as the only source of corruption. From this the following probability can be formulated:

$$q_i = P\left[(T_x > \Delta + X) \cap (T_y > \Delta - X)\right]$$

The arrival times of T_x, T_y are assumed exponentially distributed , $X \sim \exp(\frac{\lambda P_i}{2N_i})$. The transmission times $M_1, M_2 \sim \exp(\delta)$ and are *iid*. The complete derivation of q_i is provided in the Appendix.

Of interest, therefore, is: how large or small is the value of U_γ for a given value of γ under the *Delta* protocol? The answer depends on several parameters characterizing four systemic aspects:

- *Size and topology of graph database.* Size is expressed by the total number of edges N, and the fraction f of edges distributed across servers. We consider a common edge access patterns or topologies: a Scale-free topology, edges have different access probabilities and those that get accessed more frequently tend to be smaller in number.
- *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. To choose Δ one calculates $P(M > \Delta) = 1 - \epsilon$, e.g. $\epsilon = 0.01$ equates to a $\Delta = 2.3s$

To answer ii) an discrete-event simulation was performed. This simulation focused on a subset on the total database, the most popular edges to keep the runtime of the simulation to a reasonable time period. From the simulation metrics are collected measuring the number of aborts as a result of the *delta* protocol for a range of transaction arrival rates and Δ values.

7 Evaluation

The model assumes that the distributed graph database processes many reads-followed-by-write graph queries concurrently, processing both local and distributed edges. The simulation is performed on a Scale-Free graph, such as a social network, human brain, or road network, large enough to make processing non-trivial. There are approximately 77 million local edges, approximately 33 million distributed edges (in proportion to good graph partitioning algorithms); a graph of this size would have approximately 10 million nodes.

The graph consisted of five edge types, $N_1 = 10^4, N_2 = 10^5, N_3 = 10^6, N_4 = 10^7, N_5 = 10^8$ with access probabilities $p_1 = 0.5, p_2 = 0.26, p_3 = 0.13, p_4 = 0.07$ and $p_5 = 0.04$. The number of read operations per query is distributed geometrically starting at 2, with an average of 15. In all edge types, a fraction 0.3 are distributed, the remainder are local. The database is initial clean and considered to be corrupted when 10% ($\gamma = 0.1$) of all edges are corrupted. The time taken under operational corruption U, is measured in hours. U considered for a range of transaction arrival rates, $\lambda = (100, ..., 2000)$.

TODO: discuss results

The second simulation considers the most popular class of edges, $N = 10^4$, for 3 values of $\Delta = 1.2, 1.5, 2.3$, measured in seconds, which correspond to a 90,95 and 99 % chance a given message delivery exceeds Δ.

TODO: discuss results

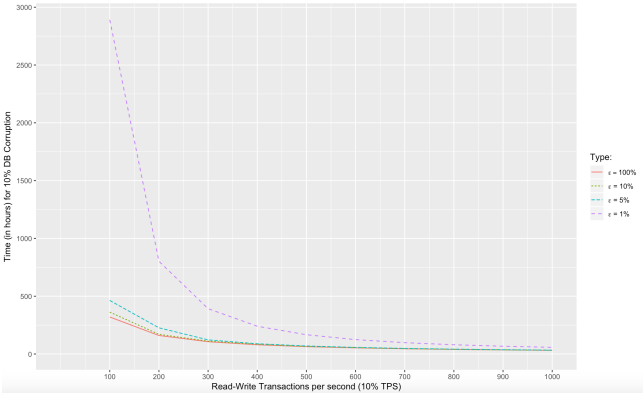

Figure 8. Corruption results

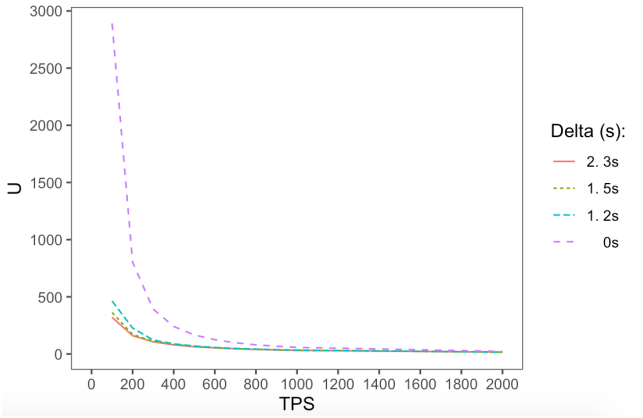

Figure 9. Abort results

8 Related Work

TODO

9 Conclusions

TODO

10 Acknowledgments

TODO

A Appendix

Figure 6 presents an interleaving when a distributed edge can become half-corrupted under the *Delta* protocol.

Transaction T_x arrives at S_i at time t_x (let $t = 0$) and writes tentatively, with the message delay between servers for T_x to write the edge at S_j being M_x. Transaction T_y arrives at S_j at time t_y ($t_y > t_x$) and writes the distributed edge, T_y then takes time M_y to write the edge at S_i. If T_x arrives at S_j after $t_y + \Delta$ and T_y arrives at S_i after $t_x + \Delta$ the edge can become half-corrupted. Letting $t_x = 0$, the probability that T_x and T_y conflict can be formulated as:

$$P\left[(M_x > \Delta + T_y) \cap (M_y > \Delta - T_x)\right]$$

.

The arrival times of T_x, T_y are assumed exponentially distributed , $T \sim \exp(\rho)$. Where, $\rho = \beta = \frac{\lambda P_i}{2N_i}$, the probability a given operation accesses the incorrect record of a half-corrupted edge of type i. The transmission times between servers $M_1, M_2 \sim \exp(\delta)$ and are *iid*.

Therefore,

$$\begin{aligned} q_i &= P\left[(T_1 > d + X) \cap (T_2 > d - X)\right] \\ &= \int_0^d \frac{\lambda P_i}{2N_i} e^{-\frac{\lambda P_i}{2N_i}x} e^{-\delta(d+x)} e^{-\delta(d-x)} dx \\ &\quad + \int_d^\infty \frac{\lambda P_i}{2N_i} e^{-\frac{\lambda P_i}{2N_i}x} e^{-\delta(d+x)} dx \\ &= e^{-2d\delta} - \left(\frac{\delta}{\frac{\lambda P_i}{2N_i} + \delta}\right) e^{-(\frac{\lambda P_i}{2N_i} + 2\delta)d} \end{aligned}$$

To choose to bound Δ consider the probability of the message delay exceeding Δ.

$$\begin{aligned} P\left[M > \Delta\right] &= 1 - e^{-\delta\Delta} \\ 1 - e^{-\delta\Delta} &= 1 - \epsilon \\ e^{-\delta d} &= \epsilon \\ \Delta &= -\frac{\ln(\epsilon)}{\delta} \end{aligned}$$

Substituting Δ into the above equation yields the conflict probability for a given ϵ. For example, $e = 0.001$ gives a 0.001 % probability the message delay exceeds Δ, for this $\Delta = 0.035s$

References

[1] [n.d.]. Apache Cassandra Documentation. http://cassandra.apache.org. Accessed: [2020-02-17].

[2] [n.d.]. JanusGraph Documentation. http://janusgraph.org/. Accessed: 2020-02-16.

[3] [n.d.]. Scale-Free Networks. https://en.wikipedia.org/wiki/Scale-free_network. Accessed:[19-02-20].

[4] [n.d.]. TitanDB Documentation. https://titan.thinkaurelius.com. Accessed: [2020-02-17].

[5] Peter Bailis and Ali Ghodsi. 2013. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM* 56, 5 (2013), 55–63. https://doi.org/10.1145/2447976.2447992

[6] 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

[7] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. [n.d.]. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. ([n. d.]). arXiv:cs.DB/http://arxiv.org/abs/1910.09017v1

[8] 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

[9] 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

[10] 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

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

[12] 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.

[13] Jim Webber, Paul Ezhilchelvan, and Isi Mitrani. [n.d.]. Modeling Corruption in Eventually-Consistent Graph Databases. ([n. d.]). arXiv:cs.DB/http://arxiv.org/abs/1904.04702v1