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 few or no isolation guarantees, profiting from increased performance but leaving them susceptible to rapid corruption. A lightweight 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*.

Keywords. Graph Databases, Reciprocal Consistency

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
© 2020 Association for Computing Machinery.
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 [8]. Application areas are wide reaching from healthcare, to social networks and fraud detection [9]. Graph databases model data as a *property graph* [14], 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. 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 [15], exceeding the storage capacity of a single-node database and motivating the need for distributed graph databases. A common design pattern is to partition graph data over several machines in a cluster and use a BASE database [13] for storage, adapted with a graph processing layer ([1], [4]). A caveat with this approach is BASE databases in practice eschew transactional guarantees in order to achieve higher performance.

Recent work [10] and [16] highlighted how reciprocal consistency can easily be violated in such a design introducing 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.

Preserving reciprocal consistency requires some degree of coordination between partitions in the presence of concurrent modifications. This paper proposes and analyses a light-weight protocol that ensures reciprocal consistency in operational contexts where concurrency control mechanisms are done away with for sake of performance.

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 facilitates bi-directional edge traversal and allows for better query performance.

A common approach to storing graphs (arising from JanusGraph [1] and TitanDB [4]) is for database 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 vertex should be able to reify the edge correctly, returning consistent results. 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.

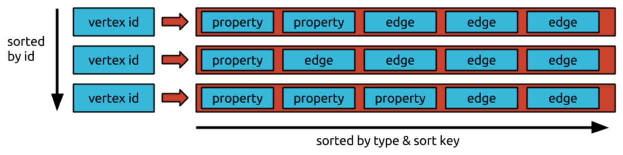

Figure 1. Database records representing vertices, containing data values and adjacency lists [1].

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 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 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.

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 [12]. 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 always non-negligible ranging from 25-75% [12].

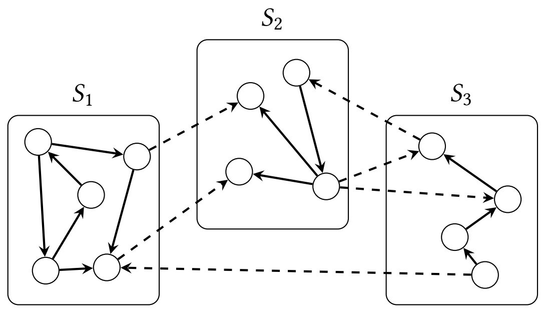

Figure 2. A graph partitioned across $k = 3$ servers in a cluster. Dashed lines represent distributed edges and solid lines represent local edges.

Adjacency lists can now contain edge pointers to vertices on remote servers. Maintaining reciprocal consistency for such distributed edges is challenging - especially given a common architecture employed by contemporary distributed graph databases. Often they use an existing BASE database 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. Using the primitives provided by BASE databases maintaining reciprocal consistency for local edges is straightforward, this is not true for distributed edges. The lack of concurrency control across partitions means it is possible that concurrent updates can interleave in a manner that violates reciprocal consistency.

4 Corruption in BASE Distributed Graph Databases

Earlier work investigated how weak isolation across partitions in BASE distributed graph databases can undermine reciprocal consistency of distributed edges, causing irreversible corruption that spreads at alarmingly rates ([10], [16]).

When a given transaction writes a distributed edge it must infact perform two writes, writing reciprocal information at the source and the destination vertices, which say reside on servers S_i and S_j respectively. Concurrent transaction's writes to a distributed edge can interleave producing a distributed edge in a *half-corrupted* state - reciprocal consistency has been violated, Figure 3. For such 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.*

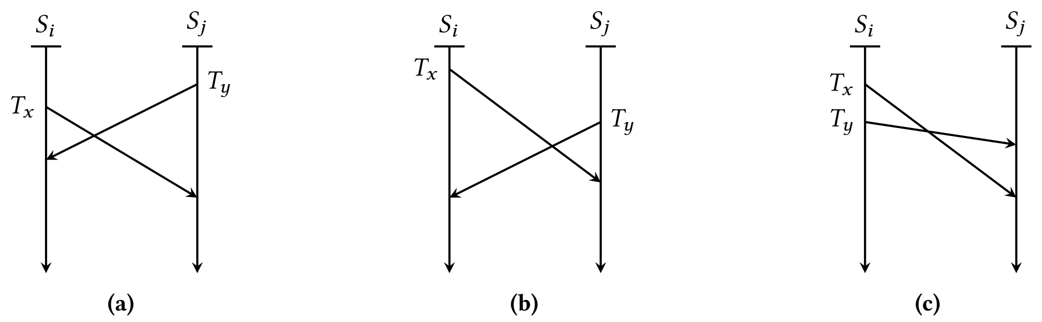

Figure 3. Possible interleavings of concurrent transaction's writes to a distributed edge spanning servers S_i and S_j by transactions T_x and T_y. In (a) T_y begins writing to the distributed edge before T_x, in (b) the converse is true, else they are equivalent. In (c) both transactions begin writing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

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 3 will result in the half-corrupted edge displayed in Figure 4.

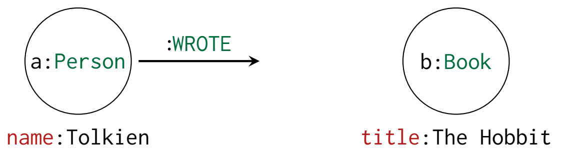

Figure 4. A half-corrupted edge resulting from conflicting transactions.

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.

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

5 *Delta* Protocol

A half-corrupted edge is an example of a *dirty write* (ANSI *P0* [7], 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 3. This is can be implemented by transactions taking long duration write locks[7], releasing them only once the acquiring transaction has committed or aborted. To prevent deadlock a policy such as `NO_WAIT` deadlock detection is used [11]. This approach is problematic in a distributed graph database as 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. Taking long duration locks on these edges significantly limits concurrency and throughput.

The motivation behind the *Delta* protocol was to develop a lightweight protocol that prevents half-corruption at a cheaper cost. One could imagine such a protocol be "bolted onto" a BASE distributed graph database. Note, this protocol is solely a concurrency control mechanism for distributed edges, guarantees about vertices, local edges are left for future work, along with issues surrounding replication and atomicity.

We leverage the fact that a transaction that writes at one end of a distributed edge, must then write the other. We then assumed that the network delay between two servers can be predicted to be some Δ. All writes are temporary and assumed to be made permanent by a later atomic commitment protocol once the transaction has completed. From this, a distributed edge write is allowed to proceed provided there is no other temporary write 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. This protocol avoids all conflict scenarios in Figure 3 provided Δ is not exceeded. It is helpful to think of this protocol as a locking protocol with a timer, in the extreme when Δ is set to infinity the protocol is equivalent to long-duration write locks with NO_WAIT deadlock detection.

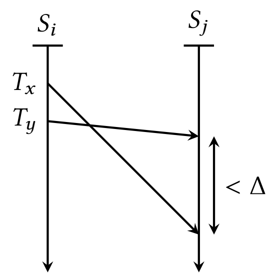

Figure 5. An example of the *Delta* protocol preserving reciprocal consistency.

If Δ is exceeded 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.

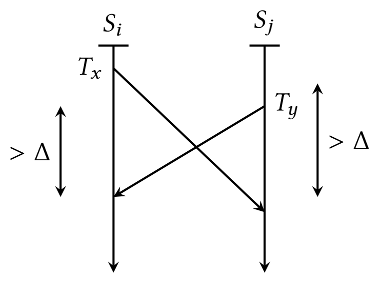

Figure 6. An example of the *Delta* protocol failing to preserve reciprocal consistency.

In summary, 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 for different values of Δ, i) how does the protocol impact on the time until operational corruption? ii) how many transactions are aborted as a result of the protocol?

6 Modeling

In order to answer i) the model developed in [10] and fine-tuned in [16] 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.

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.

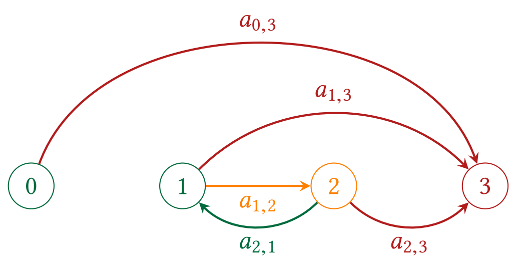

Figure 7. Edge transitions between clean, half-corrupted and semantically corrupt states.

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 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 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 [10] and [16] for a granular discussion of the initial model.

The *Delta* protocol influences the rate of corruption reducing the probability a transaction corrupts an edge, q_i^{new}. To simplify the derivation of the new conflict probability it was assumed that messages sent from the same servers do not arrive out of order, Figure 3(c). This leaves the Figure 6 as the only source of corruption. From this the following probability can be formulated:

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

The arrival times of T_x, T_y are assumed exponentially distributed , $X \sim \exp(\rho)$, where $\rho = \frac{\lambda P_i}{2N_i}$, the probability of

[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)

accessing a given end edge of type i. The transmission times $M_1, M_2 \sim \exp(\delta)$ and are *iid*. The complete derivation of q_i^{new} is provided in Appendix A.

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.

In order to answer (ii), calculating the number of aborts per second for a given Δ an discrete event-based simulation was constructed. The simulation, focuses on the subset of edges with the highest access probability.

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 7.7 billion local edges, approximately 3.3 million distributed edges (in proportion to good graph partitioning algorithms); a graph of this size would have approximately 1 billion nodes.

The graph consisted of seven edge types, $N_1 = 10^4, N_2 = 10^5, N_3 = 10^6, N_4 = 10^7, N_5 = 10^8, N_6 = 10^9, N_7 = 10^10$ 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 $p_7 = 0.01$. The number of read operations per query is distributed geometrically 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. 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 hours. 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 the range $\lambda = (10000, ..., 100000)$. We considered 3 values of $\Delta = 1, 5, 10ms$.

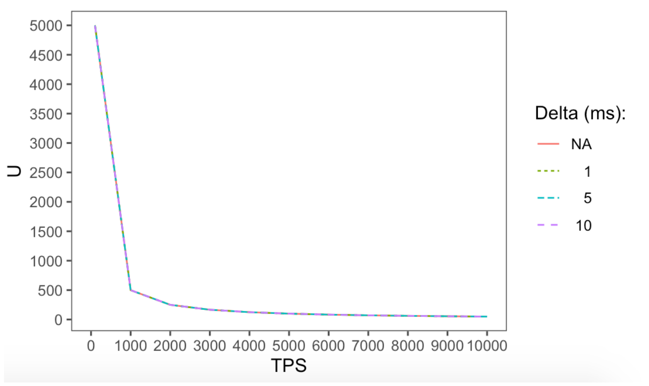

Figure 8. Time until operational corruption under delta protocol

The results for measuring the impact of time until operational corruption are given in Figure 8.

The abort rates for $\Delta = 1, 5, 10ms$ is given in Figure 9 for the most popular edge type, $N = 10^4$. The simulation was ran for 10 seconds for a range of transaction arrival rates, $\lambda = (1000, ..., 10000)$.

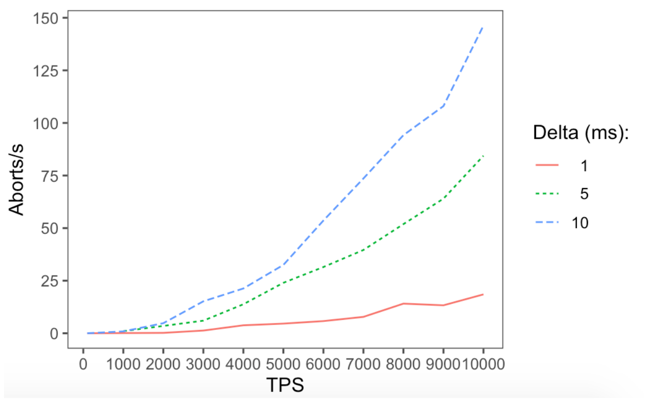

Figure 9. Aborts per seconds under protocol

8 Conclusions

In this paper a lightweight protocol for providing reciprocal consistency and mitigating the problem of high contention in a distributed graph database has been presented. The Δ protocol leverages the fact writes to edges always consists of two sequential writes. The result is guarantees much weaker than even Read Uncommitted isolation, the weakest ANSI isolation level. However, a mechanism to provide solely reciprocal consistency is believed to be valuable in practice. Concurrency control has been a long researched area, however to the best of our knowledge no other work has presented protocols bespoke to maintaining distributed graph databases.

TODO: More details regards protocol performance.

9 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:

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

.

The arrival times of T_x, T_y are assumed exponentially distributed , $T \sim \exp(\rho)$. Where, $\rho = \frac{\lambda P_i}{2N_i}$, 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,

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

References

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

[2] [n.d.]. openCypher Documentation. https://www.opencypher.org.

[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] 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] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] 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

[13] Dan Pritchett. 2008. BASE: An Acid Alternative. *ACM Queue* 6, 3 (2008), 48–55. https://doi.org/10.1145/1394127.1394128

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

[15] 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.

[16] 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