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 the wild, 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. Then, a BASE database [14] is used for storage and adapted with a graph processing layer ([1], [4]). A caveat with this approach is BASE databases often eschew transactional guarantees in order to achieve higher performance, specifically there is often no concurrency control mechanism for operations that spans partitions.

Recent work [11] and [17] highlighted how reciprocal consistency of distributed edges can 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 of distributed edges requires some degree of coordination between partitions in the presence of concurrent modifications. This paper proposes and analyses a lightweight protocol that ensures reciprocal consistency of distributed edges in operational contexts where traditional concurrency control mechanisms have been 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 allows an edge to be traversed both directions at the same cost, facilitating improved 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.

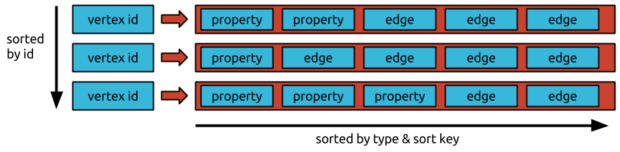

Figure 1. Database records representing vertices, containing data values and adjacency lists [1].

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 [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. 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% [13].

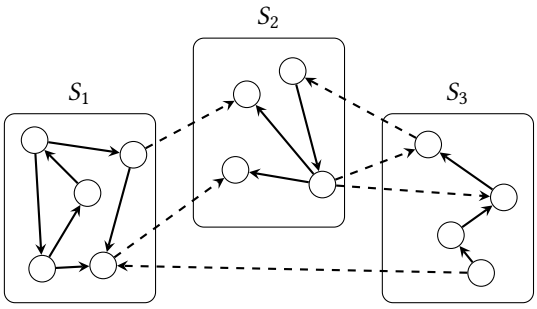

Figure 2. A graph partitioned across $k = 3$ servers in a cluster. Dashed lines represent distributed edges and solid lines represent local edges.

Adjacency lists can now contain edge pointers to vertices on remote servers. Maintaining reciprocal consistency for 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.

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 in BASE distributed graph databases can undermine reciprocal consistency of distributed edges, causing irreversible corruption that spreads at alarmingly rates ([11], [17]). This process is explained in Section 4.

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

[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.

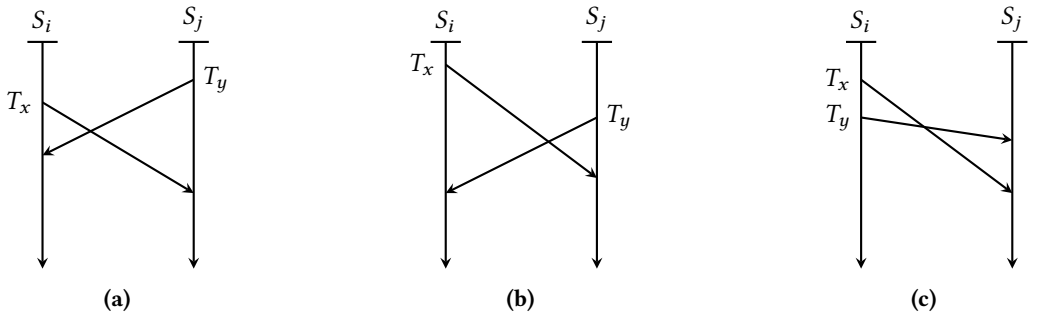

Figure 3. Possible interleavings of concurrent transaction's writes to a distributed edge spanning servers S_i and S_j by transactions T_x and T_y. In (a) T_y begins writing to the distributed edge before T_x, in (b) the converse is true, else they are equivalent. In (c) both transactions begin writing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

4 Corruption in BASE Distributed Graph Databases

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 sufficient 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, 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*.

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. 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 b.name = 'unknown'
CREATE (u)-[:WROTE]->(b)
```

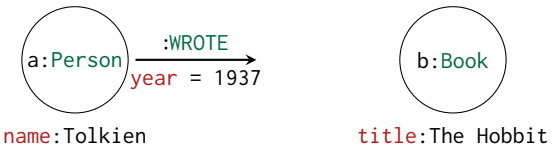

Figure 4. A half-corrupted edge.

5 *Delta* Protocol

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 3.

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].

This approach is problematic in a distributed graph database as a subset of distributed edges are traversed and modified with a high frequency e.g. critical sections of motorway in a road network, leading to high contention. In addition, graph transactions tend to be longer-lived than transactions in relational databases. Taking long duration locks on these frequently accessed edges significantly limits concurrency and reduces throughput. With these concerns in mind we

developed the *Delta* protocol, which aimed at preventing distributed edges becoming half-corrupted and hence quashing the seed of corruption. We also wanted to keep performance at levels that make using a BASE distributed graph database an attractive choice.[3]

The protocol leverages the fact that a transaction that writes at one end of a distributed edge, must then immediately write the other end. It is 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 the interleaving in Figure 3 (c), T_x overtakes T_y but is aborted when arriving at S_j as T_y has already wrote tentatively within Δ. 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.

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. However, if Δ is sufficiently large (larger than the duration of the transaction) the protocol produces outcomes equivalent to taking long-duration write locks on distributed edges with `NO_WAIT` deadlock detection.

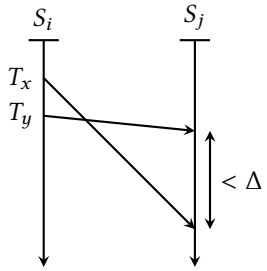

Figure 5. An example of the *Delta* protocol preserving reciprocal consistency.

Correctly choosing Δ is paramount, as if it 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. The network delay between S_i and S_j is sufficiently large the total ordering of writes to the distributed edge is lost.

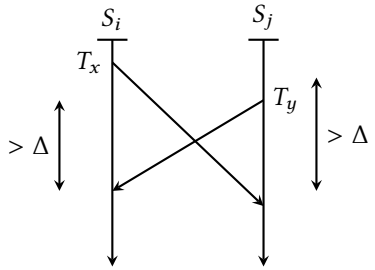

Figure 6. An example of the *Delta* protocol failing to preserve reciprocal consistency.

In summary, the Δ protocol attempts to prevent the occurrence of half-corrupted distributed edges by aborting transactions but if Δ is exceeded operational corruption 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 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.

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

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.

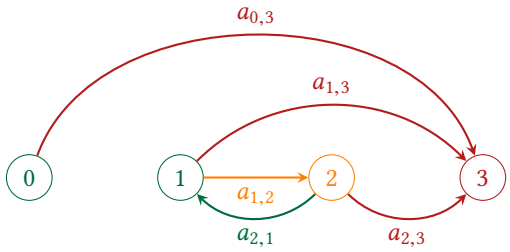

Figure 7. Edge transitions between clean, half-corrupted and semantically corrupt states.

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 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 8. 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.

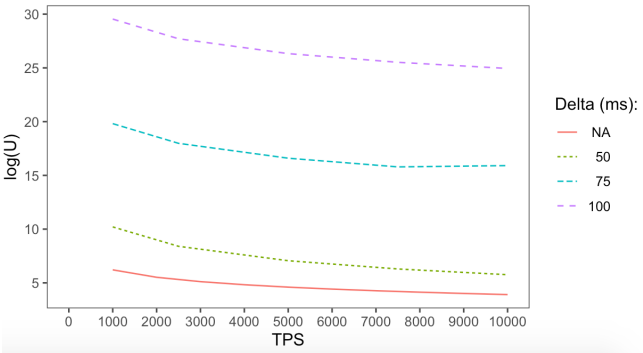

Figure 8. Time, log(U), until operational corruption ($\gamma = 0.1$) for $\Delta = 50, 75, 100ms$.

The abort rates for $\Delta = 50, 75, 100ms$ are given in Figure 9 for the most popular edge type, $N = 10^4$. The simulation was ran for 10 seconds for a range of transaction arrival rates, $\lambda = (1000, ..., 10000)$. 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.

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

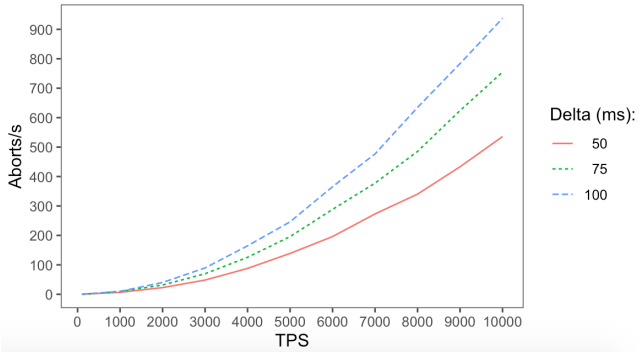

Figure 9. Aborts per seconds for $\Delta = 50, 75, 100ms$.

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 corruption can spread if left unchecked. Simulations indicate the protocol rules out corruption resulting from half-corrupted distributed edges in realistic database lifetime; the abort rate seems acceptable. 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. 2018. Modeling Corruption in Eventually-Consistent Graph Databases. (2018). arXiv:cs.DB/http://arxiv.org/abs/1904.04702v1