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

Our earlier work identifies *reciprocal consistency* as an important property that must be preserved in distributed graph databases. It also demonstrates that a failure to do so seriously undermines the integrity of the database itself in the long term. Reciprocal consistency can be maintained as a part of enforcing any known isolation guarantee and such an enforcement is also known to lead to reduction in performance. Therefore, in practice, distributed graph databases are often built atop BASE databases with no isolation guarantees, benefiting from good performance but leaving them susceptible to corruption due to violations of reciprocal consistency. This paper designs and presents a lightweight, locking-free protocol and then evaluates the protocol's abilities to preserve reciprocal consistency and also offer good throughput. Our evaluations establish that the protocol can offer both integrity guarantees and sound performance when the value of its parameter is chosen appropriately.

CCS Concepts. • **Information systems → Database transaction processing**; **Graph-based database models**.

Keywords. Graph Databases, Concurrency Control, Reciprocal Consistency

ACM Reference Format:
Jack Waudby, Paul Ezhilchelvan, Jim Webber, and Isi Mitrani. 2020. Preserving Reciprocal Consistency in Distributed Graph Databases. In *7th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '20), April 27, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3380787.3393675

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

ACM ISBN 978-1-4503-7524-5/20/04…$15.00
https://doi.org/10.1145/3380787.3393675

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* [16], 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 [16]. 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 [17], 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 [12] and [11] 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 [15] 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 the 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 distributed edge *must* immediately follow each other and the small interval between them is the sole *raison d'être* for reciprocal concurrency violations.

The paper is organized as follows: Section 2 describes how edges in a graph database are stored and the notion of reciprocal consistency. This notion is examined in the context of distributed graph databases in Section 3 and the root causes of consistency violations are explained in the following section. Section 5 presents the Delta protocol together with correctness reasoning. Sections 6 and 7 are devoted to evaluating the protocol using two metrics that measuring the protocol's ability to avoid inconsistencies (ensuring *safety*) and unnecessary aborts (enhancing *throughput*). Section 6 explains the strategies used for evaluating these metrics and the latter presents the results for various values chosen for the protocol parameter (Δ). Section 8 concludes the paper.

2 Reciprocal Consistency

In the property graph data model, edges have direction and each edge runs from a *source* vertex to a *destination* vertex. In the storage layer, however, edge directionality does not exist; **both** the source and the destination vertices store information about each other. This allows edge traversal to be bidirectional and speeds up query performance.

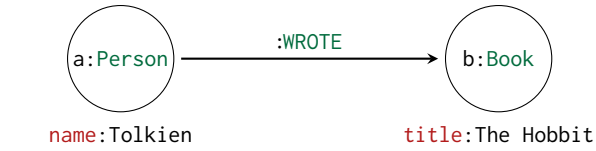
Consider, for example, the statement: "Tolkien *wrote* The Hobbit". It is expressed using vertex *a* for Tolkien and vertex *b* for The Hobbit, and an edge *wrote* running from *a* (source) to *b* (destination). Corresponding openCypher [2] code is given below and Fig 1(a) shows the model level view.

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

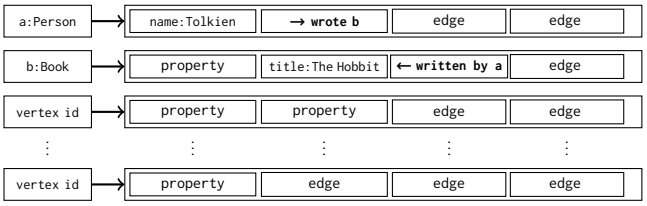
Fig 1(b) depicts the internal representation of a graph arising from JanusGraph [1] and TitanDB [4]. A vertex, such as *a*, is represented by a record that contains one or more properties of that vertex, followed by a sequence of *edge pointers* pointing to all those vertices to which this vertex is related either as a source or a destination. The sequence of edge pointers is also called the *adjacency list*.

It can be seen in Fig 1(b) that *a*'s adjacency list has an edge pointer entry that stores '*a wrote b*' while *b*'s list has a corresponding entry storing the reciprocal (or inverse) information '*b written* by *a*'. When the adjacency list entries for a given edge refer to each other in a complementary manner like this, that edge is said to be *reciprocally consistent*.

Consider a query: 'list all titles by the author who wrote The Hobbit'. This query needs to start from *b* which represents the only entity specified explicitly in it. Thanks to the reciprocal information in *b*, it can reach *a* from *b*, even though edge *ab* is "directed" from *a* to *b*, and then compile the necessary list from *a*. Note that reciprocal consistency is assumed to be prevail when a query reads only the source or destination vertex of an edge.



(a) Logical view.



(b) Storage view.

Figure 1. Logical and storage views of a reciprocally consistent edge *ab*.

3 Distributed Graph Databases

A distributed graph database employs a shared-nothing architecture, partitioning a graph among loosely cooperating servers. Graph partitioning is non-trivial and a common approach is to use a *k*-balanced edge cut [14]. The objective of such an approach is to minimize the proportion of edges that span partitions and also to balance the distribution of vertices to partitions. Fig 2 depicts a graph database partitioned across 3 servers, S_i, $i = 1, 2$ and 3.[1]

Intra-partition and inter-partition edges are respectively referred to as *local edges* and *distributed edges* (shown using dashed lines in Fig 2). The proportion of distributed edges is not negligible and can range from 25-75% [14].

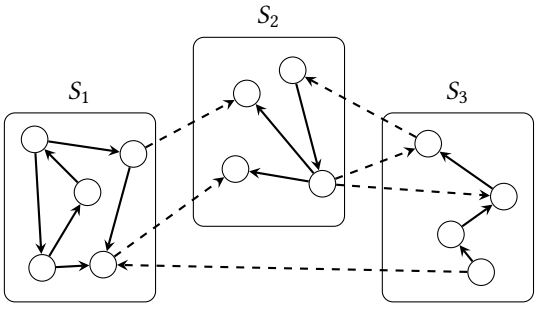


Figure 2. Local and distributed edges

Adjacency lists can now contain edge pointers to vertices on remote servers. Guaranteeing reciprocal consistency for distributed edges is challenging, especially when an existing BASE database is used for storage and is then adapted with a query language expressed in terms of edges and vertices

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

along with some gluecode to bind that interface to the underlying database. Opting for this design appears to be a good choice: it offers the application programmer the modeling convenience of graphs together with the operational characteristics of the underlying BASE database.

However, a major problem with this design option is the lack of transactional semantics from the underlying BASE databases. The latter seldom provide guarantees for multi-operation, multi-object transactions that span partitions. Without concurrency control across partitions, concurrent updates of distributed edges can interleave and violate reciprocal consistency.[2] Our earlier work showed that undermining reciprocal consistency in distributed edges can lead to irreversible corruption of the distributed database itself and the corruption rate can be worryingly high ([12], [18]). Origins of reciprocal inconsistency in distributed edges are explained in Section 4.

4 Reciprocally Inconsistent Distributed Edges

Suppose that the edge `(a)-[:WROTE]->(b)` is a distributed edge, with vertices a and b in servers S_i and S_j, $j \neq i$, respectively. When a transaction writes this edge ab,

1. two writes are performed: reciprocal entries in the adjacency lists of both a and b are updated, and
2. write order is unconstrained: a transaction is equally likely to write a then b as it is to write b then a.

Concurrent transactions T_x and T_y can interleave in the following three ways and each one is depicted in Fig 3:

(a) T_x starts before T_y; it writes a at S_i first and then proceeds to S_j across the network; T_y operates the other way round, beginning with S_j and proceeding to S_i (see Fig 3(a)). The net effect is: $T_x \rightarrow T_y$ at S_i and at $T_y \rightarrow T_x$ at S_j, where $T \rightarrow T'$ at S denotes that T *precedes* T' at server S.

(b) Same as the previous case, except that T_y starts earlier than T_x (see Fig 3(b)).

(c) Same net effect as in previous two cases, except that both T_x and T_y start their first writes at S_i, $T_x \rightarrow T_y$, but T_y overtakes T_x in reaching S_j where $T_y \rightarrow T_x$ (see Fig 3(c)).

We could envisage three more corresponding cases (a') - (c') where the roles of T_x and T_y in cases (a) - (c) are simply interchanged; e.g., in case (a'), T_y starts before T_x, writes a at S_i first and then proceeds to S_j across the network; T_x operates the other way round, beginning with S_j and proceeding to S_i. Thus, there are only 6 ways concurrent T_x and T_y can interleave. The arguments we make based on cases (a) - (c) of Fig 3 equally apply, by symmetry, to cases (a') - (c') and so, for brevity, we will not consider the latter.

At the end of each case in Fig 3, the last update on a is by T_y and that on b is by T_x. Unless updates of T_x and T_y are commutative, ab cannot be reciprocally consistent. As an example, suppose that T_x deletes the *wrote* edge while T_y concurrently 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
```

The interleaving patterns depicted in Fig 3 leave ab reciprocally inconsistent, as shown in Fig 4.

When T_x and T_y do not interleave, either $T_y \rightarrow T_x$ or $T_x \rightarrow T_y$ holds at both servers S_i and S_j; in that case, the last update on *both* a and b would be by either T_x or T_y, respectively. That is, when transactions update at both ends of ab in *some* arbitrarily chosen but identical order, ab is left reciprocally consistent after each transaction's update.

When interleaving updates leave ab reciprocally inconsistent, ab can be said to have become *half-corrupted* because if $T_y \rightarrow T_x$ is the chosen order between T_y and T_x, then the edge pointer in b of ab is in error; otherwise, the edge pointer in a is erroneous. Thus, a reciprocally inconsistent edge ab certainly has a corrupt half but the question of which half is corrupt has been left open.

Suppose that a future transaction T_w *first* reads the edge pointer of reciprocally in consistent ab, say, at vertex a. (Note that when T_w reads any edge, it does not check for reciprocally consistency). At that moment, T_w (implicitly) chooses the order $T_x \rightarrow T_y$ and thereby invalidates the other order $T_y \rightarrow T_x$ that prevails at vertex b. Thus, from that moment onward, the b end of edge ab becomes the corrupt end.

If no transaction ever reads the edge pointer at vertex b, then the order $T_x \rightarrow T_y$ effectively prevails and the half-corruption of ab remains invisible to the rest of the database. However, if T_z is to subsequently read the edge pointer at b and write another edge based on what it read, i.e., The Hobbit has unknown author, then it is introducing updates not consistent with what T_w read earlier; it thus introduces *semantic corruption* into the database. Further writes based on reading semantically corrupt data also spread corruption.

A database is said to be *operationally corrupt* when a significant proportion of its data records are in a semantically corrupt state. As stated earlier, we had shown that 10% of a distributed graph database can become semantically corrupt well within the system lifetime itself ([12], [18]).

Two more relevant remarks on past works: a half-corrupted edge is due to a *dirty write* (ANSI *P0* [8], Adya *G0* [5]) in the context of distributed graph databases. If the database provides the ANSI isolation level *Read Uncommitted*, it will identically order the writes of concurrent transactions and

[2]The concurrency control primitives provided by BASE databases are typically sufficient to ensure reciprocal consistency for local edges.

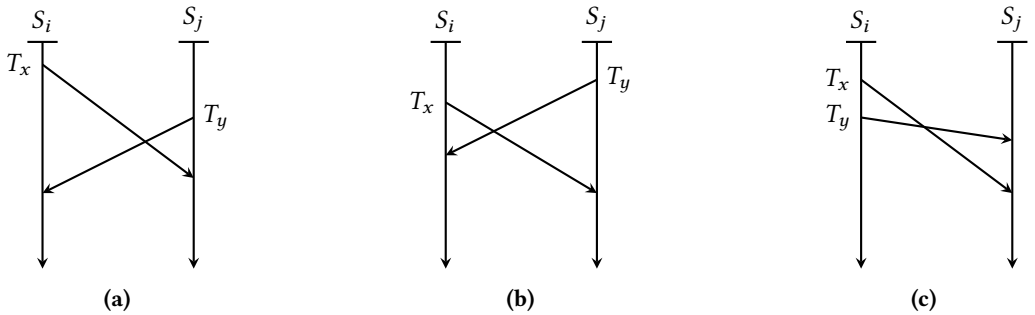
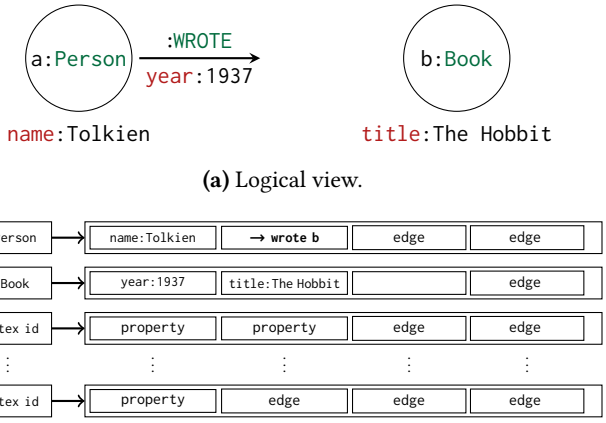


Figure 3. Interleavings of concurrent writes to a distributed edge by transactions T_x and T_y.



(a) Logical view.

(b) Storage view.

Figure 4. Logical and storage views of a reciprocally inconsistent edge *ab*.

this would prevent all interleaving patterns shown in Fig 3 and thus avert half-corruption altogether.

5 Delta Protocol

A straightforward solution to preventing dirty writes is for transactions to take long duration write locks [8], releasing them only after the acquiring transaction has committed or aborted. To prevent deadlock, a policy such as `NO_WAIT` deadlock avoidance is used, which was shown to be the optimal policy in a distributed, partitioned database [13].

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 which are characterized by the following two aspects. First, a subset of edges are traversed and modified with a very 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 those in other databases. So, having to wait for earlier writes to terminate while transactions are long lived, will severely limit the scope for concurrent processing and reduce throughput in a highly contentious environment.

With these concerns in mind we developed the Delta protocol, which aims at preventing edges from becoming 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 write 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 only if the end pointer is either in a committed state or the immediately preceding tentative write was done at least Δ time before (where time is measured as per local clock).
3. If a transaction performs all its tentative writes, then it is permitted to commit; otherwise, it must abort.
4. A transaction commits, when all its tentative writes are made permanent, e.g., by using an atomic commit protocol.
5. Tentative writes of an aborting transactions are ignored. An ignored tentative write can make a new transaction abort unnecessarily 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 completing its update at one end of a distributed edge at one server and starting its update at the other end of the same edge at another server.

Let Δ be chosen such that $\Delta > \delta$. Now consider the interleaving in Fig 3(b) and let t_x be the (global) time when T_x

[3] The Delta protocol is a concurrency control mechanism only for distributed edge updates; it does not concern itself with updates on vertices.

starts at server S_i; similarly t_y be the time T_y starts at server S_j. Since T_x starts after T_y in Fig 3(b), $t_x > t_y$, i.e. $(t_x - t_y) > 0$.

Say, T_y reaches S_i at time $t_y + d_y$, where d_y is the *actual* time elapsed between T_y completing a tentative write at one end and starting at another end, let us assume that $d_y \leq \delta$. When T_y arrives at S_i it will find a tentative write already done at time t_x. In this case, $t_y + d_y - t_x = d_y - (t_x - t_y) < d_y \leq \delta < \Delta$; so, T_y will abort, preventing writes from interleaving and half-corrupting the edge. Similar arguments can be made for the scenario in Fig 3(a), if $d_x \leq \delta$ where d_x is the *actual* time elapsed between T_x completing a tentative write at one end and starting at another end. Note that $t_y > t_x$ and T_x will abort because it will find out, on reaching server S_j for its next tentative write, that $t_x + d_x - t_y = d_x - (t_y - t_x) < d_x \leq \delta < \Delta$. Finally, for the interleaving in Fig 3(c), if $t_y - t_x > \Delta$, then T_y cannot overtake T_x at S_j, if $d_x \leq \delta$ holds.

Assume that $d_x > \delta$ or $d_y > \delta$ is possible, i.e., the estimate δ can fail to hold. In Fig 3(b), interleaving writes are avoided only if $t_y + d_y - t_x = d_y - (t_x - t_y) < d_y < \Delta$; otherwise, reciprocal inconsistency can occur. Similarly, interleaving writes of Fig 3(a) are avoided only if $t_x + d_x - t_y = d_x - (t_y - t_x) < d_x < \Delta$; otherwise, reciprocal inconsistency can occur. Thus, in the extreme case $t_x = t_y$ and reciprocal consistency is not guaranteed if $\Delta \leq \max\{d_x, d_y\}$.

For Fig 3(c), interleaving writes are avoided only if $(t_x + d_x) - (t_y + d_y) < \Delta$; given that $(t_y - t_x) < \Delta$, when d_x exceeds δ, the interleaving writes of Fig 3(c) are avoided only if $d_x \leq 2\Delta$; otherwise, reciprocal inconsistency can occur.

In summary, the Delta protocol eliminates interleaving of transactions during edge writes, so long as Δ remains larger than the interval d that elapses between a transaction completing its write at one end of a distributed edge and starting at the other end. Since the exact value of d taken by a transaction cannot be known in advance, its bound δ is estimated with the best effort and Δ is chosen to be $\Delta > \delta$.

The larger the value of Δ used, the more likely is that $\Delta > d$ holds and half-corruption and thereby operational corruption are averted; also, on the downside, the more likely is that non-interleaving transactions will find their tentative writes within Δ time of each other and the later ones choose to abort *unnecessarily*. In the extreme case, choosing a very large Δ ($\Delta \approx \infty$) totally eliminates any risk of reciprocal inconsistency but does not allow any tentative write until the preceding one is made permanent. It is equivalent to enforcing `NO_WAIT` policy, wherein a requesting transaction that finds the requested record being locked, must abort.

Our performance evaluation will therefore measure the following two metrics for various values of Δ:

- Time taken for 10% of a large database to be corrupt,
- Number of transactions aborted per second.

If d is exponentially distributed with mean $1/\mu$, the probability of d exceeding Δ is $e^{(-\mu\Delta)}$. The values of Δ chosen will explore a range of probabilities of Δ being exceeded.

6 Performance Evaluation Strategies

To assess the time to operational corruption, we reused the model that we developed in [12] and adapted it for the Delta protocol. For completeness, the model is briefly explained first before we discuss the adaptation. We note that an empirical evaluation of the time to operational corruption using a real system would be impractical due to the sheer length of time and the cost it would take to run the experiment. (For certain values of Δ, the model predicts anywhere between 1-75 years for operational corruption! Details are to follow.)

The model of [12] assumes that transactions arrive in a Poisson stream with rate λ transactions per second (TPS, for short). Each transaction performs a random number (K) of read operations and then updates a single edge.

To model a scale-free graph, edges in the distributed database are divided into T types, $i = 1, 2, \ldots, T$. Type-1 edges represent the most popular edges to be accessed by transactions and type-T edges are the least popular. Popular edges are smaller in number compared to the less popular ones. N denotes the total number of edges in the database. For all edge types, a fraction f are distributed.

At time 0, all edges are assumed to be clean (free from corruption); thereafter, an edge can be in one of four states:

1. Local and clean.
2. Distributed and clean.
3. Distributed and half-corrupted.
4. Local or distributed and semantically corrupted.

The delay d is the time interval that elapses between a transaction completing its tentative write at one end of a distributed edge and starting at another end. It constitutes the window for concurrent transactions to interleave as portrayed in Fig 3. It is assumed to be exponentially distributed with mean $1/\mu$ (i.e., at rate μ).

The model parameters now enable us to compute the probability q_i that a transaction updating a clean type-i distributed edge, leaves it half-corrupted, in the absence of any concurrency control mechanism. (Our earlier work in [12] assumes no concurrency control). These probabilities, q_i, $1 \leq i \leq T$, are then used to compute transition rates $a_{j,k}$ between state j to k, $1 \leq j, k \leq 4$. (Since local and distributed edges are fixed, $a_{1,2} = a_{2,1} = 0$.)

The rates $a_{j,k}$ are in turn used to simulate the spread of corruption within the database to estimate the first passage time U_γ for a fraction γ of N edges to enter the state 4. (When γN edges become semantically corrupt, the database becomes operationally corrupt.) The reader is directed to [12] and [18] for a granular discussion of the initial model.

Delta Protocol Adaptation. Our concurrency protocol reduces the probabilities q_i, $1 \leq i \leq T$ and we compute q_i^{new} as shown in Appendix A. The new probabilities q_i^{new} are used in the model and simulations of [12] to estimate U_γ.

Number of aborts per second. To evaluate this metric for various values of Δ, a second simulation that focuses specifically on the subset of most frequently accessed distributed edges was performed.

Note that both metrics that we set out to evaluate will be influenced by several parameters that characterize the database and other 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 following parameter choices are inline in with industry experiences. 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 before a write per query is geometrically distributed starting at 2, with an average of 15. In all edge types, $f = 0.3$ are distributed, the remainder are local; in proportion with a good graph partitioning algorithm.

The delay d between a transaction completing a tentative write at one end and starting at another end is exponential distributed with a mean of $5ms$. The database is initially clean and considered to be operationally corrupted when 10% ($\gamma = 0.1$) of all edges are semantically corrupted. The time taken until operational corruption, U, is measured in days. We consider 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 (10000, ..., 100000). The following Δ values were considered $\Delta = 50, 75, 100ms$. For each Δ the probability that d exceeds Δ is $P(d > \Delta) = 4.5^{-5}, 3.1 \times 10^{-7}, 2.1 \times 10^{-9}$ respectively.

The results for measuring the impact of Δ on the time until operational corruption are given in Fig 5 (where the y-axis in log scale). With no concurrency control, U ranges between 50-500 days. For $\Delta = 50ms$, U increases to 1-75 years. For $\Delta = 75, 100ms$ the time to corruption is significantly large as shown Fig 5.

To evaluate the number α of aborts occurred per second, simulations were run for 10 seconds for each arrival rate, $\lambda = (1000, ..., 10000)$. Fig 6 reports the fraction $\frac{\alpha}{\lambda}$ for various values of λ. This fraction is also the probability that an incoming transaction is aborted due to the Delta protocol. For $\Delta = 50$ ms, the abort probability is between $1 - 5\%$, this increases to between $1-7\%$ and $1-9\%$ for $\Delta = 75$ and $\Delta = 100$ respectively.

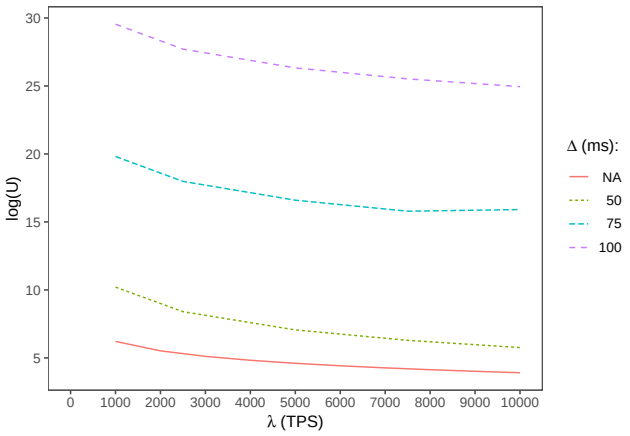


Figure 5. Time until operational corruption.

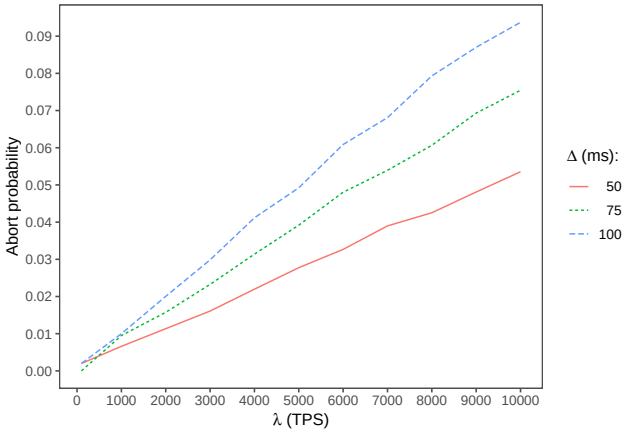


Figure 6. Fraction of aborts.

8 Conclusions

Database concurrency control has been a long researched area, however to the best of our knowledge this is first attempt at developing a 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 that writes to distributed edges always consist of two sequential writes to entries in the adjacency lists of vertices the edge connects. Since it is concerned only with edges (and not vertices) in a graph, it provides guarantees weaker than Read Uncommitted isolation (the weakest ANSI isolation level).

The protocol that is presented and performance-evaluated here, we believe, valuable in practice given the popularity of BASE distributed graph databases and the rate at which semantic corruption can spread if reciprocal consistency is left unchecked. Simulations indicate that when Δ values are chosen to be reasonably large, the protocol rules out corruption resulting from half-corrupted distributed edges while keeping the abort rate considerably small.

For future work, we intend on implementing the protocol to assess the validity of the simulations and to enable comparisons with traditional lock-based approaches. 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] P Ezhilchelvan, I Mitrani, J Waudby, and J Webber. 2019. Design and Evaluation of an Edge Concurrency Control Protocol for Distributed Graph Databases. In *16th European Performance Engineering Workshop (EPEW 2019)*. Newcastle University.

[12] 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

[13] 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

[14] 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

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

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

[17] 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.

[18] 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

A Derivation of Conflict Probability

Edges are divided into T types, $i = 1, 2, ..., T$. The number of edges of type i is represented by n_i. The probability an edge of type i is accessed by a given operation is represented by p_i. (Exact values taken by n_i and p_i are given in Section 7.)

The arrival time, a, of a transaction, T, is assumed exponentially distributed with rate ρ. Where $\rho = \frac{\lambda p_i}{2n_i}$ is the probability a given operation accesses the incorrect record of a half-corrupted edge of type i.

The time interval, d, that elapses between transaction T completing its tentative write at one end of a distributed edge and starting at another end, is assumed exponentially distributed with rate μ.

Consider the interleaving in Fig 3(a) and assume T_x arrives at S_i at time 0. Then, T_x arrives at S_j after d_x. Assume T_y arrives at S_j at some time a. Then, T_y arrives at S_i at time $a+d_y$. Half-corruption occurs under the following conditions:

(i) At S_j, $d_x > a + \Delta$

(ii) At S_i, $a + d_y > \Delta$

The conflict probability, q_i^{new}, for edge type i is given by,

$$
\begin{aligned}
q_i^{new} &= P\left[(d_x > a + \Delta) \cap (d_y > \Delta - a)\right] \\
&= \int_0^{\Delta} \frac{\lambda p_i}{2n_i} e^{-\frac{\lambda p_i}{2n_i} a} e^{-\mu(\Delta+a)} e^{-\mu(\Delta-a)} da \\
&\quad + \int_{\Delta}^{\infty} \frac{\lambda p_i}{2n_i} e^{-\frac{\lambda p_i}{2n_i} a} e^{-\mu(\Delta+a)} da \\
&= e^{-2a\mu} - \left(\frac{\mu}{\frac{\lambda p_i}{2n_i} + \mu}\right) e^{-(\frac{\lambda p_i}{2n_i} + 2\mu)a}
\end{aligned}
$$

From the perspective of computing q_i^{new}, Fig 3(b) is equivalent to Fig 3(a) and the above expression holds. Fig 3(c) is ignored as it can be avoided by having all first tentative updates on a given edge at a given server (e.g., at S_i) be numbered sequentially and having the second tentative attempts processed (at S_j) as per this sequence number.