Preserving Reciprocal Consistency in Distributed Graph Databases

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

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

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

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

Abstract

In this paper we formalize the notion of *reciprocal consistency*, a important feature specific to graph databases. If reciprocal consistency is not enforced during transaction processing, a scale-free distributed graph database can become semantically corrupt within a short time period relative to database lifetime. Reciprocal consistency can be maintained as a part of enforcing any of several known strong isolation guarantees (e.g., snapshot isolation) incurring well established performance costs. However, in practice systems are often deployed with weaker consistency, this paper describes two protocols that provide only reciprocal consistency semantics. Catering for users that are interested only in ensuring reciprocal consistency to preserve the integrity of a distributed graph database. Both protocols presented are compared and contrasted using metrics such as number of aborts per second.

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

Keywords. Graph Databases, Reciprocal Consistency

ACM Reference Format:
Paul Ezhilchelvan, Isi Mitrani, Jack Waudby, and Jim Webber. 2020. Preserving Reciprocal Consistency in Distributed Graph Databases. In *PaPoC '20: Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, April 27, 2020, Heraklion, Crete, Greece.* ACM, New York, NY, USA, 3 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 technologies [4]. Application areas are wide reaching from healthcare, social networks and fraud detection [5]. Graph databases model data as a *labeled property graph*, vertices represent entities and edges represent the relationship between entities; properties can be stored on both vertices and edges [8].

In practice graphs can be extremely large [9], exceeding the storage capacity of a single-node database. An approach to storing large graphs is to partition graph data over several machines in a cluster. Moving from a single-node database to one partitioned and distributed across servers introduces several interesting problems regarding data consistency. Graphs place a high demand on the underlying database and maintaining structural integrity across partitions - called *reciprocal consistency* - in a distributed graph database is non-trivial.

In order to preserve reciprocal consistency, one could implement a heavyweight coordination protocol across partitions providing Serializability or Snapshot Isolation, which have associated performance costs. To avoid these costs database managements systems (DBMSs) provide several weak isolation models [3]. In addition, there are a number of systems that eschew transactional guarantees altogether, typically offering eventual consistency [2]. Recent work [6] investigated how violation of reciprocal consistency can occur under eventual consistency, leading to semantic corruption at alarmingly rates. The contribution of this paper is the description of two protocols that provide only reciprocal consistency, a consistency model between the known extremes that provides suitable performance. The performance of each protocol is measured using metrics such as aborts per second.

2 Reciprocal Consistency

In the labeled property graph data model edges have direction, there is a *source* and a *destination* vertex. 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 [REF]) is for records to represent vertices containing both data values and an adjacency list containing edge "pointers" to other vertices. In this representation an edge has "reciprocal" entries in the adjacency lists of the vertices the edge connects Figure 1. A query reading either the source or destination vertices should be able to reify the edge correctly, returning consistent results. When the adjacency lists for vertices are mutually compatible like this, we say they are *reciprocally consistent.*

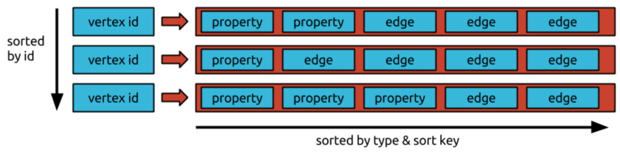

Figure 1. Graph representation [REF].

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). Adjacency lists of both A and B record information about that edge and this information is mutually reciprocal (or inverse) of each other: A's list will indicate 'A *wrote* B' while B's will have 'B *written* by A'. Thus, a query 'list all titles by the author who wrote The Hobbit' can be answered by landing at B and then traversing to A; even though the edge `(A {name:'Tolkien'})-[:WROTE]->(B {title:'The Hobbit'})` is a directed edge at model level abstraction.

3 Distributed Graph Databases

A distributed graph database employs a shared-nothing architecture, partitioning a graph between a number of loosely cooperating servers. Graph partitioning is non-trivial and a common approach is to use a k-balanced edge cut [7]. The objective of such an approach is to minimize the proportion of edge 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*, [FIG]. The proportion of distributed edges is highly data dependent. However, the number of such edges is always non-negligible ranging from 15-30%.

In a distributed graph database, adjacency lists can contain "pointers" to edges on remote servers. This makes maintaining distributed edge reciprocal consistency more challenging - especially given a common architecture employed by contemporary distributed graph databases. Often they use an existing eventually consistent database (e.g. Apache Cassandra [REF]) to store data, which has been adapted with a

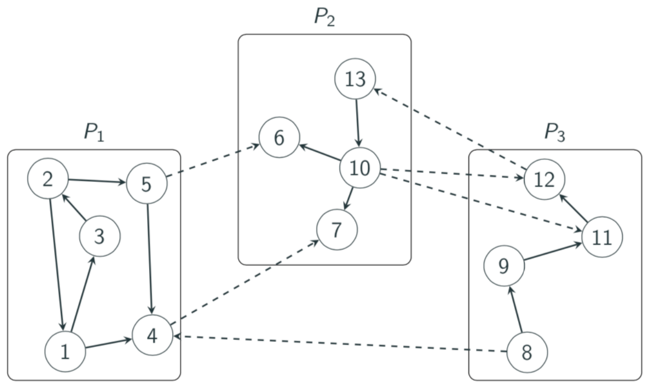

Figure 2. A partitioned graph assigned to servers in a cluster. Some edges are local and some distributed

programmatic API or query language expressed in terms of edges and vertices along with some gluecode to bind that interface to the underlying database. Superficially, opting for this design appears to be a good choice: the user has the modeling convenience of graphs with the operational characteristics from the underlying database. However, the problem with this design is the (lack of) transactional semantics are inherited from the underlying store. For example, Apache Cassandra provides no way of preventing updates from mutual interference in its normal multi-partition use case. Earlier work displayed how weak isolation undermines distributed edge reciprocal consistency, causing irreversible corruption that spreads at alarmingly rates [6].

Of course, reciprocal consistency can be maintained as a part of enforcing any of several known strong isolation guarantees (e.g., snapshot isolation) but that incurs a performance cost if users are interested only in ensuring reciprocal consistency to preserve the integrity of a distributed database. This paper addresses that challenge.

4 Preserving Reciprocal Consistency

A single logical write to a distributed edge consists of two sub-operations, updating reciprocal information at the source and the destination vertices. The order in which sub-operations take place are not constrained. For example, when updating an edge between A and B it is equally likely to update A then B as it is to update B then A. Without sufficient isolation, concurrent updates can interleave in a manner that violates reciprocal consistency [FIG], such an edge is known as a *half-corrupted* edge. Figure [FIG] outlines the interleavings that produce half-corrupted edges.

4.1 *Delta* Protocol

In order to prevent all interleaving in Figure [FIG] the *Delta* protocol adopts a probabilistic approach. It is assumed there is a known Δ, that reflects the bound on server communication delays, with a probability $(1 - \epsilon)$ that Δ is violated.

Informally, the protocol consists of two-phases. In phase 1 all writes by sub-operations are temporary and a sub-operation is allowed to proceed provided there is no other temporary sub-operation preceding it within Δ; measured as per the local clock time. If the attempt does not succeed, the transaction aborts, aborting any and every previous tentative write that it may have successfully completed. In phase 2, a transaction that succeeds at all its tentative write attempts, decides to commit when it completes its graph traversal; the transaction commits once all its tentative writes are committed. This protocol avoids all conflict scenarios in Figure [FIG] provided the bound is not violated. If the bound is violate corruption and its spread can still occur, Figure [FIG]. There exists an interesting relationship between Δ, the proportion of aborts and the spread of corruption.

Algorithm 1: *Delta* Protocol Phase 1

```
if sub-operation=1 then
    if $\Delta = \emptyset$ then
        Perform tentative write operation;
        Set $\Delta$;
        Issue sub-operation-2;
    else
        Abort transaction;
    end
else
    if $\Delta = \emptyset$ then
        Perform tentative write operation;
        Set $\Delta$;
        Continue transaction;
    else
        Abort transaction;
    end
end
```

5 Evaluation & Results

6 Related Work

7 Conclusions

References

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

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

[3] 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

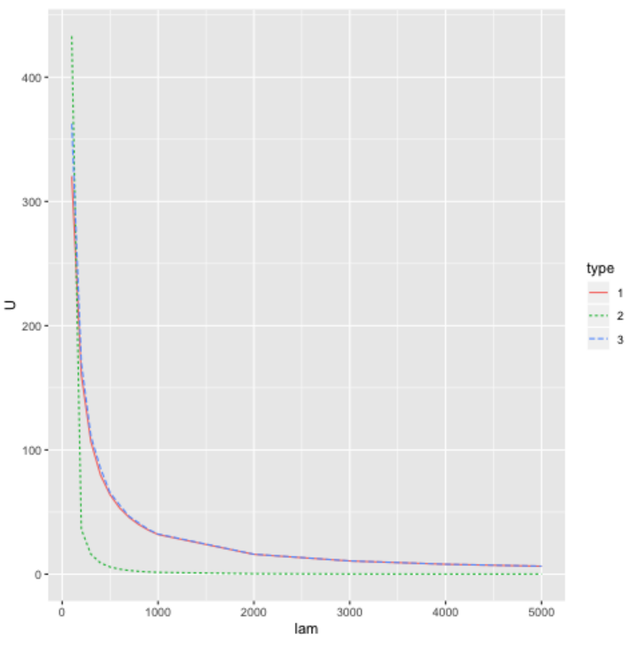

Figure 3. A partitioned graph assigned to servers in a cluster. Some edges are local and some distributed

[4] 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

[5] 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

[6] 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

[7] 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

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

[9] 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.