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

A solution for maintaining reciprocal consistency in distributed graph databases is presented.

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

Keywords. reciprocal consistency, graph databases, collision detection

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, 2 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

In recent years there has been an increased interest in and use of graph technologies[REF]. Application areas are wide reaching from healthcare to more traditional use cases such as social networks [REF]. Graph databases model as a *labeled property graph*, vertices represent entities and edges represent the relationship between entities; properties can be stored on both vertices and edges [REF].

In practice graphs can be extremely large [REF], 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 [REF]. 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.

Recent work has investigated how violation of reciprocal consistency can lead to semantic corruption at alarmingly rates [REF]. In order to prevent this form of inconsistency, one could implement a heavyweight coordination protocol across partitions providing Serializability or Snapshot Isolation, which have associated performance costs [REF]. The approach taken in this paper is to build on earlier work and develop a protocol that provides weaker consistency guarantees whilst still preserving reciprocal consistency.

2 Reciprocal Consistency

In the labelled property graph data model edges have direction, there is a *source* and a *destination* vertex. However, for query performance, edge information is stored with **both** the source and destination vertices. This facilitates bi-directional edge traversal. Graphs are often implemented as an collection of adjacency lists where a list contains neighbors of a given vertex [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 AB is a directed edge at model level abstraction.

The reciprocal nature of information at the ends of an edge not only facilitates bidirectional traversal but also ensures operational correctness. Without it, the queries 'who wrote The Hobbit?' and 'what all did Tolkien write?' could yield in different results. So, an edge is said to be *reciprocally*

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

consistent, if the adjacency lists of the vertices it connects have mutually reciprocal information.

3 Distributed Graph Databases

A common approach to partitioning a graph is to use a balanced edge-cut. Vertices are evenly assigned to partitions in a manner that minimises the number of edges that span partitions - referred to as **distributed edges**. Regardless of the partitioning strategy used a non-negligible number of edges are distributed edges.

When concurrent queries update an edge, maintaining reciprocal consistency gets harder if the edge is distributed, i.e., its vertices are stored in distinct servers of a distributed database system. Our earlier work [REF] established that when edges are permitted to become reciprocally inconsistent, independent queries reading inconsistent information from different ends of such edges, do corrupt the database through their subsequent writes, and that a significant degree of corruption can result within durations that are worryingly small compared to database lifetime.

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. It proposes a suite of simple, coordination-free protocols and compares their performance through simulations.

4 Preserving Reciprocal Consistency

Both protocols described below leverage the same fact: at the logical level, updating a distributed edge is a single operation. However, it consists of two sub-operations, it must update information at the source and destination vertices an edge connects. Note, the order in which the sub-operations take place is not constrained. Updating the source then the destination is equally as likely as updating the destination then the source.

4.1 Probabilistic Protocol

In the probabilistic approach, an edge update is allowed if an earlier, temporary update is sufficiently early by D as per the local clock time and D is the bound on the server communication delays with probability (1-e). (Note that a more pessimistic estimation of D results in a smaller e) In this approach, the smaller the value chosen for D, the more is the extent of contentions permitted, hence the smaller is the number of expected aborts and, on the other hand, the larger is the risk of reciprocal consistency not being ensured and the database being corrupted. These are examined through simulations.

Algorithm 1: How to write algorithms

```
Result: Probabilistic Reciprocal Consistency
initialization;
while While condition do
    instructions;
    if condition then
        instructions1;
        instructions2;
    else
        instructions3;
    end
end
```

4.2 Deterministic Protocol

In the deterministic approach, queries updating distributed edges are expected to carry with them some state information as they traverse from one end server of a distributed edge to the other. Other than this overhead, there is no requirement for estimating bound on traversal delays. Our simulations will compare the number of aborts incurred in both approaches.

Algorithm 2: How to write algorithms

```
Result: Deterministic Reciprocal Consistency
initialization;
while While condition do
    instructions;
    if condition then
        instructions1;
        instructions2;
    else
        instructions3;
    end
end
```

5 Evaluation & Results

6 Related Work

7 Conclusions