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. 2018. 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

1 Introduction

In recent years there has been an increased interest in and use of graph technologies. Application areas are wide reaching from healthcare to more traditional use cases such as social networks. Graph systems can be categorised into graph databases and graph processing frameworks. This paper focuses on graph databases, which at a high-level differ from relational databases in that they provide application programmers with a graph data model, as oppose to relations linked through foreign keys. Specifically, graph databases use the labelled property graph data model consisting of vertices and edges, which each can have properties and be group through labels.

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

As the sizes of graphs grow the limits of single node, replicated graph databases are being reached. This has lead to the development of horizontally scalable distributed graph databases. 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. In order to prevent this form of inconsistent one could implement a heavyweight coordination protocol across partitions providing Serializability or Snapshot Isolation, which have associated performance costs. 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 Distributed Graph Databases

In a graph database records represent vertices containing vertex properties and an adjacency list containing edge "pointers" to other vertices along with associated properties. Whilst edges typically have direction they are physically implemented as one would implement an undirected graph with meta-data indicating the direction and some reciprocal information regarding the edge. For example, an edge E joining two vertices A and B results in reciprocal entries for E being written in the adjacency lists for both A and B. If this edge represented the edge between an author and their book, the following reciprocal information would be stored A would store the information *wrote B* and B would store the information *written by A*. This permits bi-directional traversal of the edge, a query reading either A or B would be able to reify the edge correctly. When the adjacency lists for vertices are mutually compatible they are said to be *reciprocally consistent*. The challenge is maintaining reciprocally consistency in a distributed graph database.

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**. That is the number of entries in a vertex's adjacency list that point to vertices on remote servers is minimised. However, regardless of the graph partitioning algorithm used there will exist a non-negligible number of distributed edges. Without sufficient isolation between concurrent updates to a distributed edge, reciprocal consistency can be violated.

To preserve reciprocal consistency distributed edges could be updated through a protocols such as 2-Phase Commit with isolation provided via 2-Phase Locking. However, such an approach is well known to negatively impact scalability and performance. The approach taken here is provide a weaker isolation model whilst still preserving reciprocal consistency.

3 Preserving Reciprocal Consistency

Assume a graph database partitioned across servers in a cluster, with each partition being a Raft cluster providing linearizable reads and writes to the data objects it is responsible for. One strawman approach to preserving reciprocal consistency would be for update operations to take an exclusive lock on both records that make up the distributed edge, releasing locks when the update has been completed. Locking is a pessimistic approach and has the implicit assumption that concurrent updates will be common, this is not the case in graph databases. A second approach would be having a read operation read both records to verify the information is consistent. The problem here is that read operations dominant graph workloads and every read of a distributed edge would incur the penalty of traversing the network twice.

The approach developed leverages 1) the frequency of concurrent update operations to the same edge is low 2) a logical write of an edge consists of writes to two physical records 3) read operations only access a single record. We now outline a variant of the **Collision Detection** mechanism presented in REF and discuss it's combination with Raft.

3.1 Collision Detection Mechanism

Initially, each write to a distributed edge is provisional. They become permanent only if, and when, the transaction that contains them is allowed to commit.

Provisional writes on a given record can occur only one at a time and each is time-stamped using a combination of the transaction ID and the local host's clock - used to uniquely identify the operation. This is stored in a order-preserving provisional write list. Thus, when a transaction attempts to write a given record, it can identify all other transactions, called predecessors (if any), that have earlier wrote that record provisionally along with the transaction responsible for latest committed value. Currently no higher information regarding specific operation type is assumed. Read operations receive the latest committed version of a record and ignore any provisionally updated values.

Collision detection: An edge has a *tail record* and *head record* - a write operation by a transaction must write to *both* records. The order in which updates occur is not constrained, that is, one update may update the head and then after a network delay the tail, the opposite can also occur with equal chance.

In order to distinguish between which write a transaction they are given labels 1 and 2, which act as 'history' meta-data indicating the partition where a transaction started and completed its write. The meta-data consists of 1) local timestamp, 2) transaction ID and 3) order label.

The following rule is applied: **Cancellation rule:** If, by the time an operation is performing it's second update, if a previous provisional write labeled 1 has been observed, but the corresponding label 2 has not been observed, then this write is cancelled. In other words, a write is cancelled if it has observed the start of a previous attempt, but not its completion. This prevents writes happening out-of-order. When an update is cancelled, the transaction containing it is aborted and all its provisional records are erased.

3.2 Raft

Raft is a leader-based consensus algorithm for managing a replicated log. We use it for managing our write-ahead-log to provide durability. In our protocol each partition is operating their own Raft cluster consisting of N servers. Whilst operations in the Raft log are totally ordered per partition this is not required across all partitions. In fact, in order to preserve reciprocal consistency the property required is that each write operation for a given distributed edge appears in the same order across partition's Raft logs.

3.3 Combining protocols

The steps path describe the happy path when updating tail then head records:

1. Client issues write operation to distributed edge
2. Arrives at tail of edge, appends write to in-memory provisional write chain for given edge and collect predecessors
3. Issues RPC to head of edge
4. Arrives at head of edge
5. Check cancellation rule
6. If success,
   a. Commit at head with respect to predecessors, decision is made durable both updating committed value.
   b. Issue RPC to tail of edge
   c. Commit at tail with respect to predecessors
   d. Issue success response to client
7. Else abort,
   a. Abort decision is made durable

b. Issue RPC to tail of edge
c. Remove write from in-memory provisional write chain
d. Issue abort response to client

8. Response arrives at the client

At any given time the leader of the Raft cluster has a log containing commit/abort decisions, an in-memory provisional write chain containing write with labels 1 and 2 and some log entries that are not yet safe to apply to the raft state machine. When the leader fails all in-memory information and unreplicated log entries are lost.

3.4 Extensions

So far all operation that mutate a distributed edge have been encapsulated by a "write".

3.5 Alternative Methods

4 Evaluation & Results

In order to test the throughput and latency of the proposed mechanism a discrete event orientated simulation has been performed. Collision detection has been tested against the simple approach defined in Section 3.

5 Related Work

6 Conclusions