Reciprocal Consistency in Distributed Graph Databases

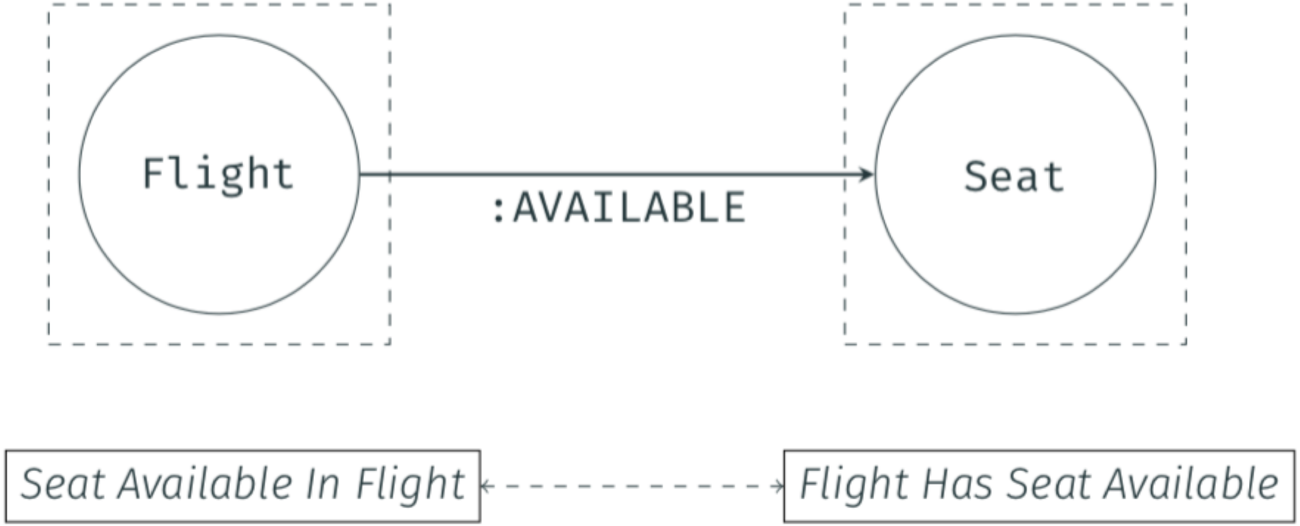
Jack Waudby

Supervisor: Paul Ezhilchelvan

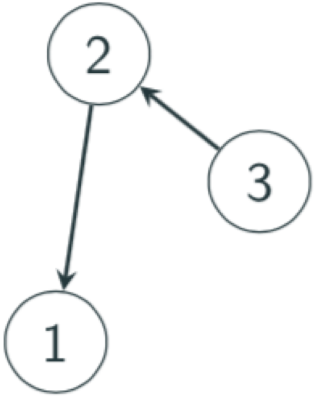
(in collaboration with Jim Webber, Neo4j)

Reciprocal Consistency

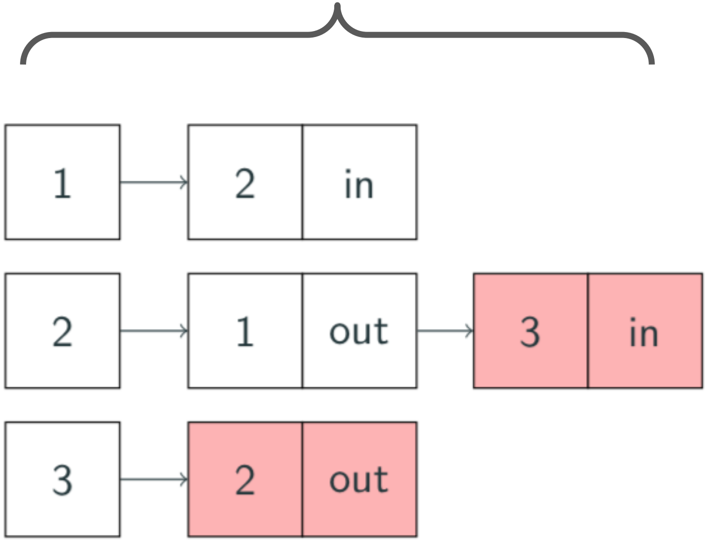
An edge is represented by **two physical records**



Reciprocal Consistency in Practice



Adjacency List



JanusGraph

Distributed Graph Databases

“Many graphs in practice are very large, often containing over a billion edges”

- 42 survey participants reported graphs with 1-10B edges
- 17 reported graphs with 10B-100B edges
- **7 reported graphs with 100B+ edges!**


The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing

Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, M. Tamer Özsu
David R. Cheriton School of Computer Science
University of Waterloo
{s3sahu,amine.mhedhbi,semih.salihoglu,jimmylin,tamer.ozsu}@uwaterloo.ca

ABSTRACT

Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, M. Tamer Özsu

The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing

VLDB 2017

Distributed Graph Databases

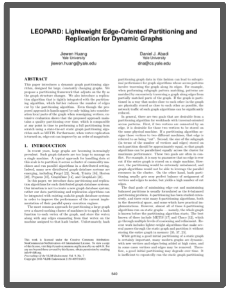
Partition data across multiple machines in a cluster

For key-value stores partitioning by keyspace does the trick

Graph partitioning is non-trivial!

k-Balanced Partitioning Problem

"The dual goals of minimizing edge cut and maintaining balanced partitions"



LEOPARD: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs

Jiewen Huang
Yale University
jiewen.huang@yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

1. INTRODUCTION

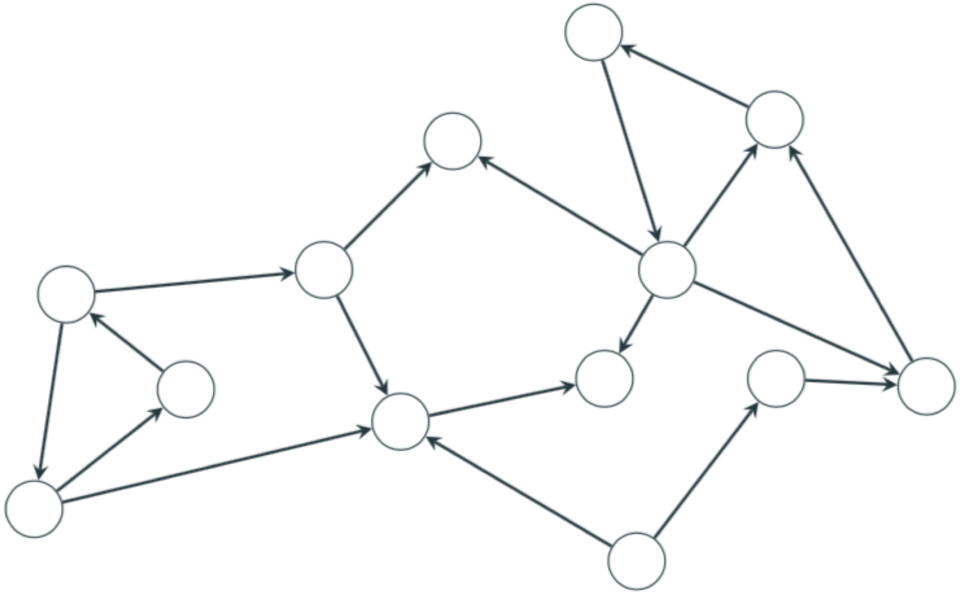
540

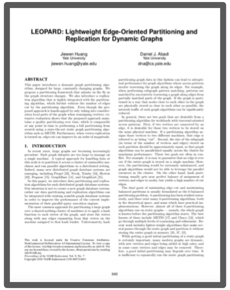
Jiewen Huang and Daniel J. Abadi

LEOPARD: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs

VLDB 2016

k-Balanced Partitioning Problem





LEOPARD: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs

Jiewen Huang
Yale University
jiewen.huang@yale.edu

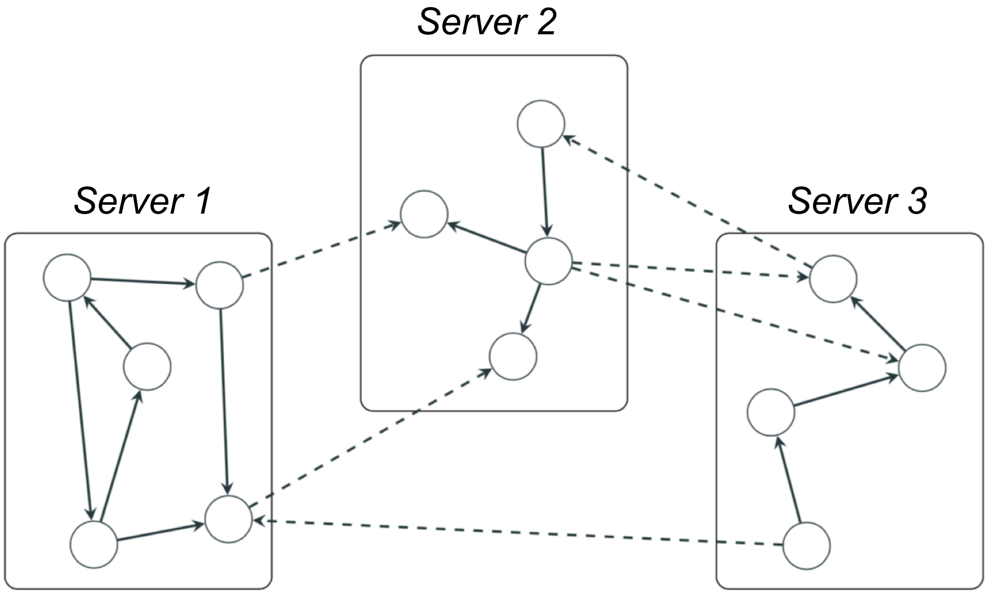
Daniel J. Abadi
Yale University
dna@cs.yale.edu

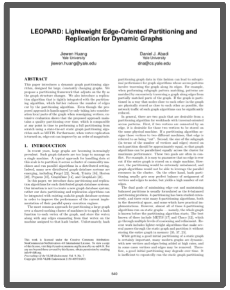
Jiewen Huang and Daniel J. Abadi

LEOPARD: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs

VLDB 2016

k-Balanced Partitioning Problem




LEOPARD: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs

Jiewen Huang, Yale University, jiewen.huang@yale.edu
Daniel J. Abadi, Yale University, dna@cs.yale.edu

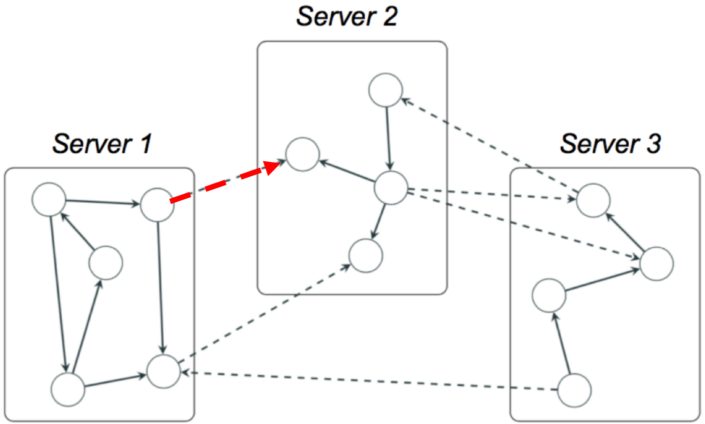
Jiewen Huang and Daniel J. Abadi

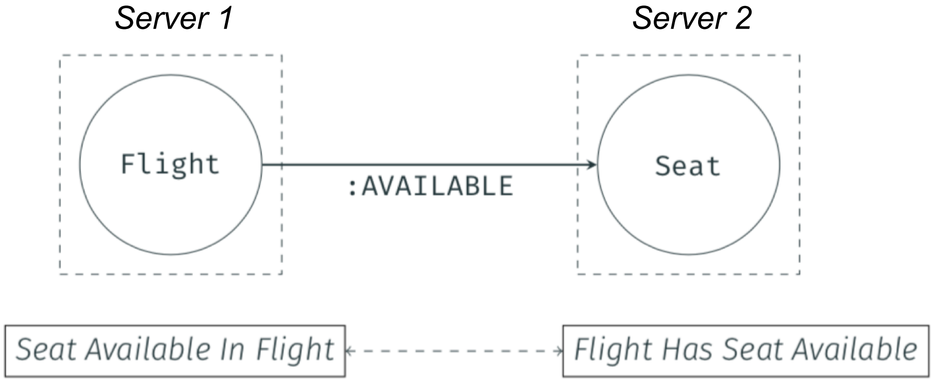
LEOPARD: Lightweight Edge-Oriented Partitioning and Replication for Dynamic Graphs

VLDB 2016

Distributed Edges

Reciprocal information resides on different servers





A Distributed Graph Database Architecture

JanusGraph + cassandra

Graph Layer

NoSQL Storage Engine

Reciprocal Consistency Violation

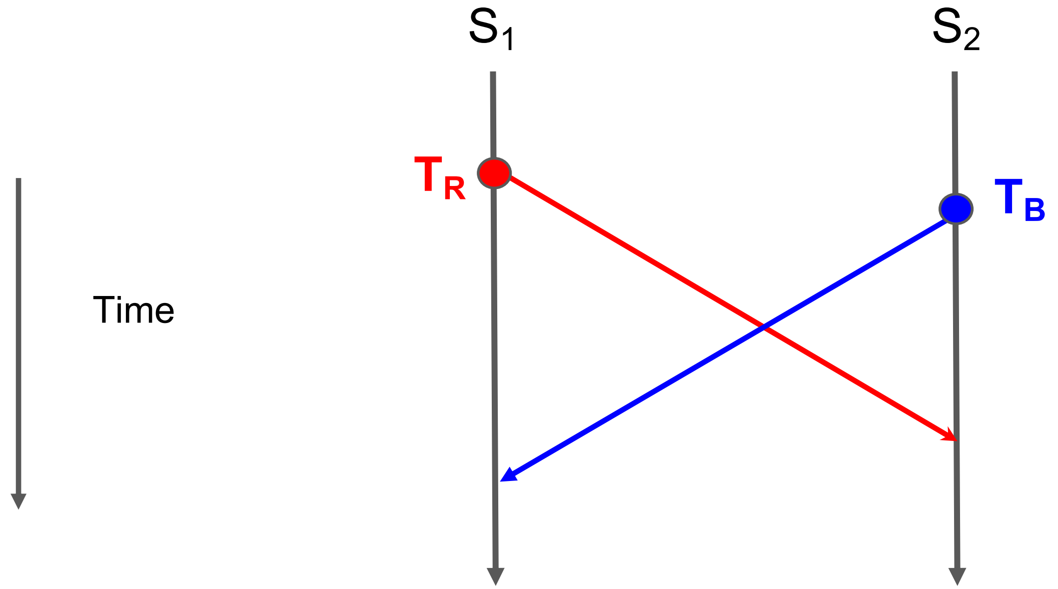
- NoSQL datastores provide weak isolation between concurrent transactions
- Distributed edges can become **reciprocal inconsistent**

Reciprocal Consistency Violation: Example

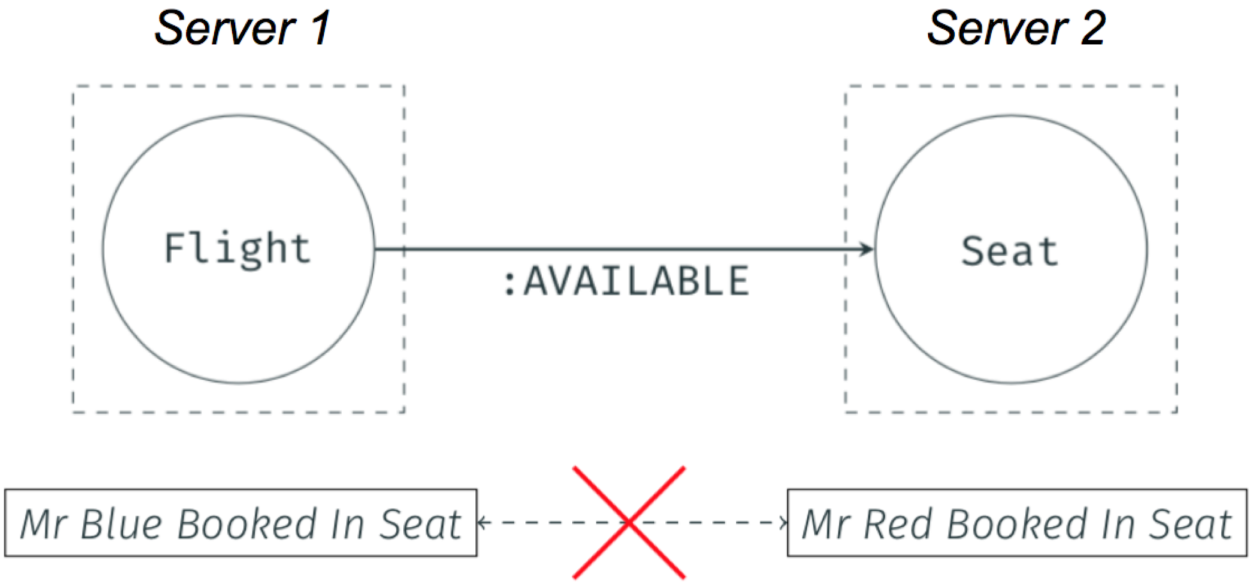
Two concurrent transactions:

- Mr. Red requests to book the seat, T_R
- Mr. Blue requests to book the seat, T_B

Reciprocal Consistency Violation



Reciprocal Consistency Violation



Spread of Corruption

1. Subsequent transactions read corrupt edge(s)
2. Performs write based on read
3. Corruption spreads through the database

How quickly does this happen?

Spread of Corruption

Model the rate of corruption in distributed graph databases with weak isolation

Parameters:

- 1 billion nodes, 11 billion edges (30% distributed)
- 5 edge types with varying access probabilities
- Reads per query geometrically distributed with avg. 15 edges
- 10% queries read-write transactions


On the degradation of distributed graph databases with eventual consistency

Paul Ezhilchelvan[1], Isi Mitrani[1], and Jim Webber[2]

[1] School of Computing, Newcastle University, NE4 5TG, UK
e-mail: paul.ezhilchelvan@ncl.ac.uk, isi.mitrani@ncl.ac.uk
[2] Neo4j UK, 41-45 Blackfriars Rd, London SE1 8NZ
e-mail: Jim.Webber@neo4j.com

Abstract. The 'eventual consistency' approach to updates in a distributed graph database leaves open the possibility that edge information may be corrupted. The process by which this occurs is modeled, with the aim of estimating the time that it takes for the database to become corrupted. A fluid approximation is developed and is evaluated for different parameter settings. Comparisons with simulations show that the results are very accurate.

1 Introduction

Managing distributed data typically means adopting either strongly or eventually consistent update policies. In the former approach, replicas of a data item are updated in an identical order at all servers [7]. When update requests are launched concurrently, users will see an identical update sequence, irrespective of the actual physical server they use for accessing the data. However, this single-server abstraction imposes a significant performance penalty, since update requests need to be ordered (and replicated) prior to being acted upon. Further, according to the CAP theorem (see [2, 6]), if the network partitions the servers, request ordering and access to data may be interrupted and the availability of services can be reduced.

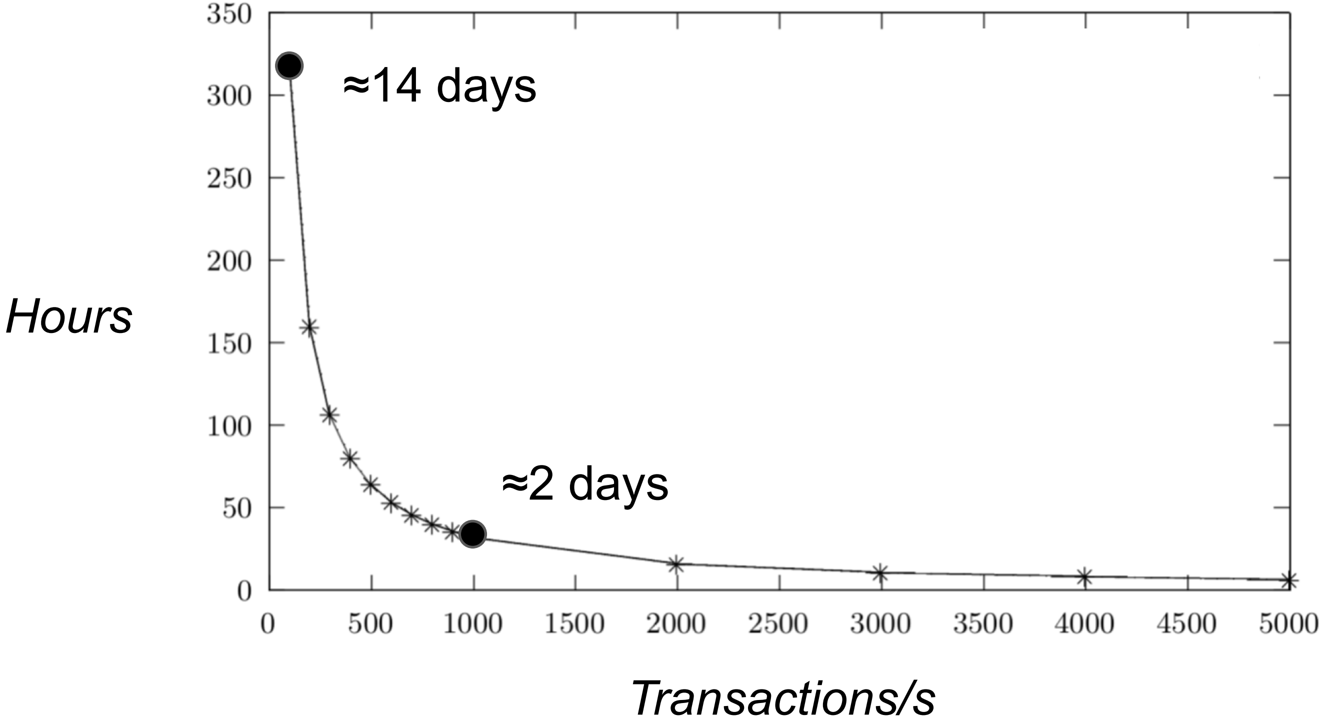
In view of these disadvantages, large distributed data stores such as Google Docs, Dynamo [5] and Cassandra [4], opt instead for an eventually consistent update policy (see [10]). Update requests are processed as soon as they arrive. This enhances system performance and availability but leaves a time window in which values of a replicated data item at different servers can be mutually inconsistent. These windows may occasionally lead to incorrect data operations. Eventual consistency is a viable model for applications where availability is paramount and where the consequences of any incorrect operations can be dealt with through compensations and state reconciliations.

For example, Bailis and Ghodsi [1] refer to an ATM service where eventual consistency can allow two users to simultaneously withdraw more money than their (joint) bank account holds; such an anomaly, on being detected, is reconciled by invoking exception handlers. Given that an ATM service is expected to be available 24/7, the eventually consistent approach is appropriate.

Paul Ezhilchelvan, Isi Mitrani, Jim Webber,
On the degradation of distributed graph databases with eventual consistency,
EPEW 2018

Simulation Results

Time taken until 10% edges were in a corrupted state



Preserving Reciprocal Consistency

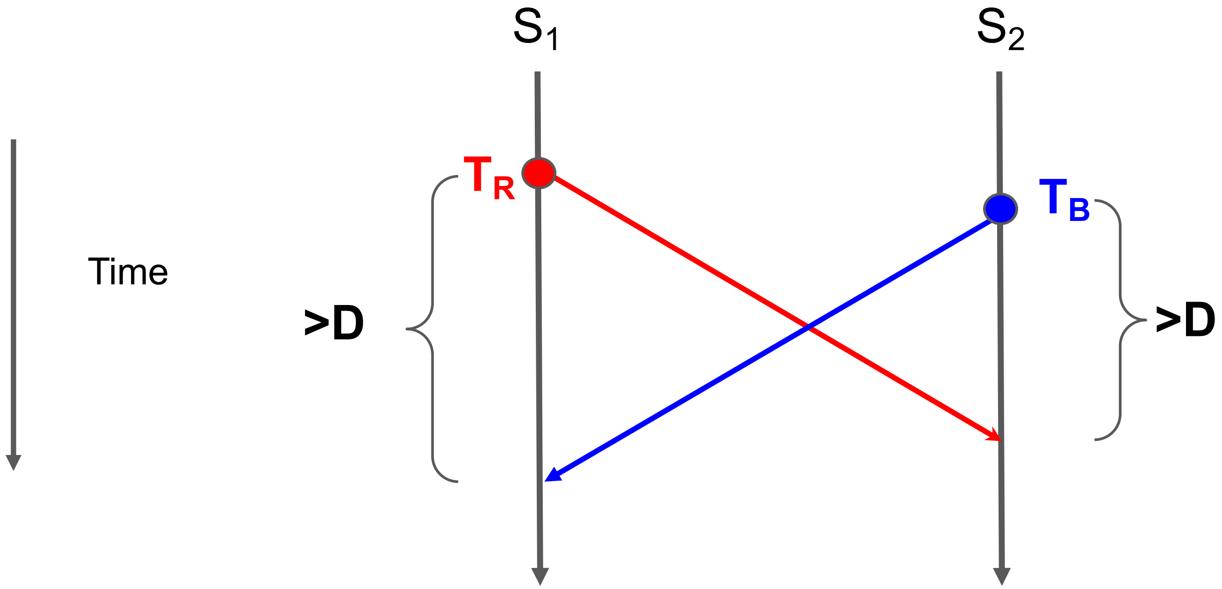
1. Delta Protocol
2. Collision Detection Protocol

Delta Protocol

- A write to a distributed edge consists of 2 writes
- Assume a bound D
- D reflects time taken by a transaction to write a distributed edge
- Rule: when attempting to write if another write exists within D, abort

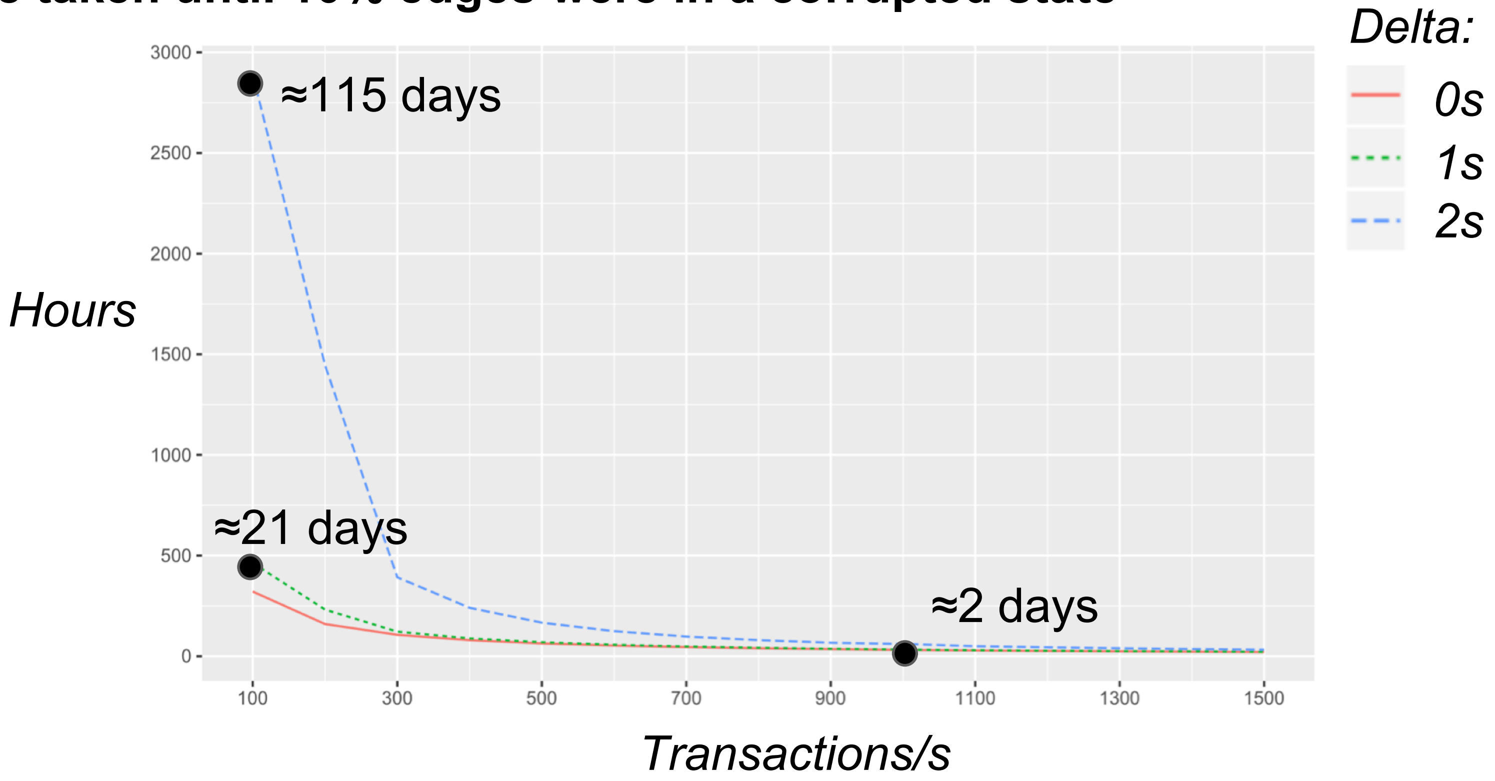
Scenarios exist when corruption can still occur

Reciprocal Consistency Violation



Delta Protocol: Simulation Results

Time taken until 10% edges were in a corrupted state



Collision Detection Protocol

- Introduce **tracking metadata**
- Transactions annotate their writes to distributed edges with a “1” and “2”

Rule: For any “1” seen, there must be a “2” in the opposite end.

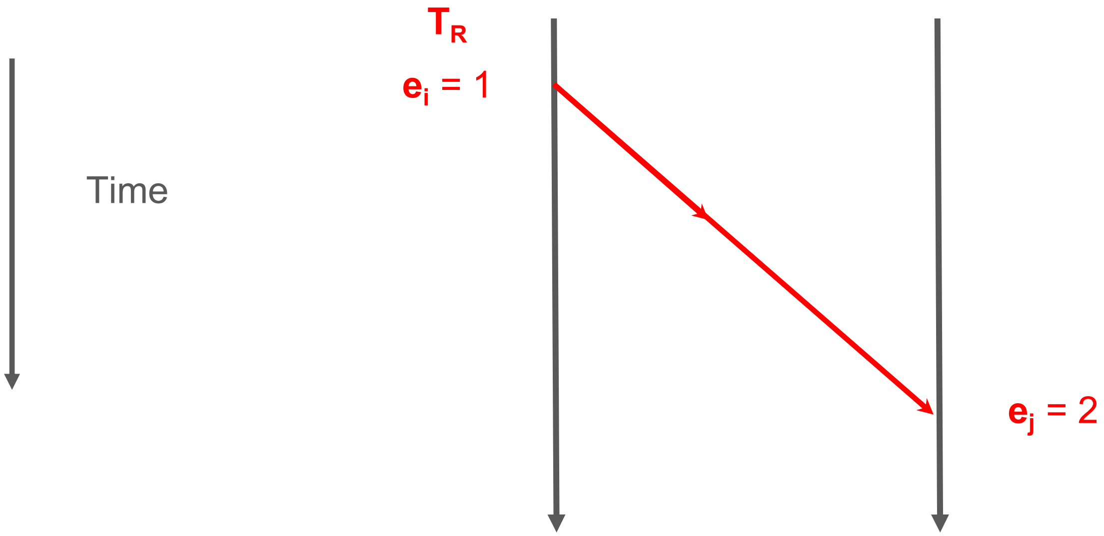

On the degradation of distributed graph databases with eventual consistency

Paul Ezhilchelvan[1], Isi Mitrani[1], and Jim Webber[2]

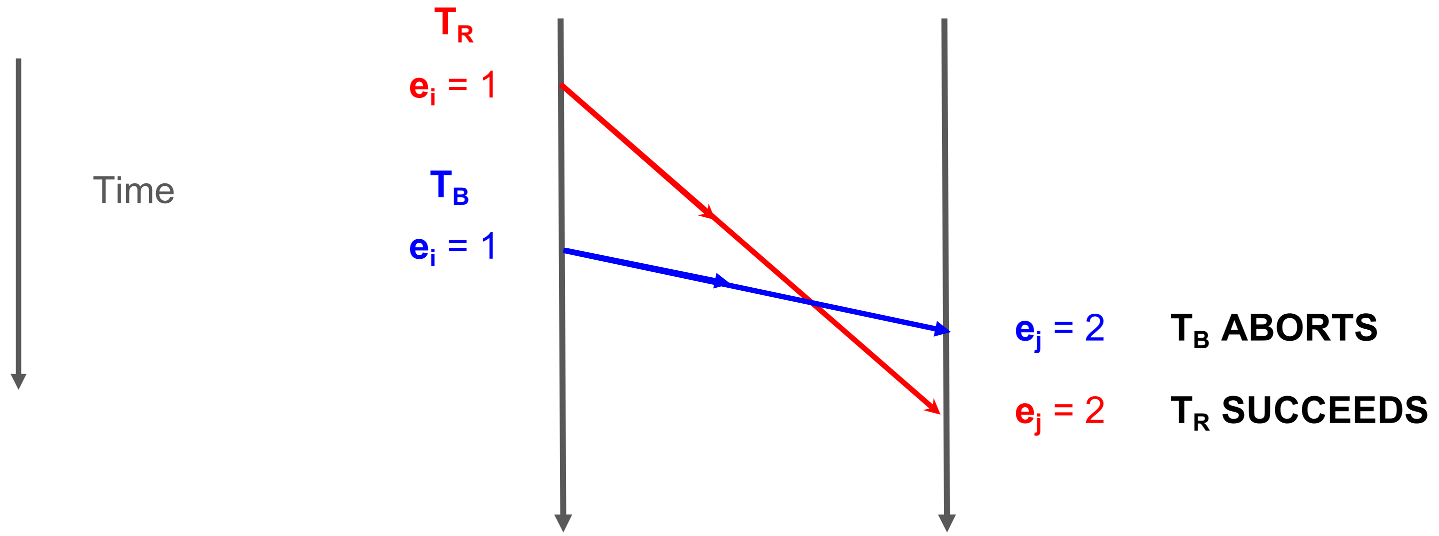
1 Introduction

Paul Ezhilchelvan, Isi Mitrani, Jack Waudby, Jim Webber,
Design and Evaluation of an Edge Concurrency Control Protocol for Distributed Graph Databases
EPEW 2019

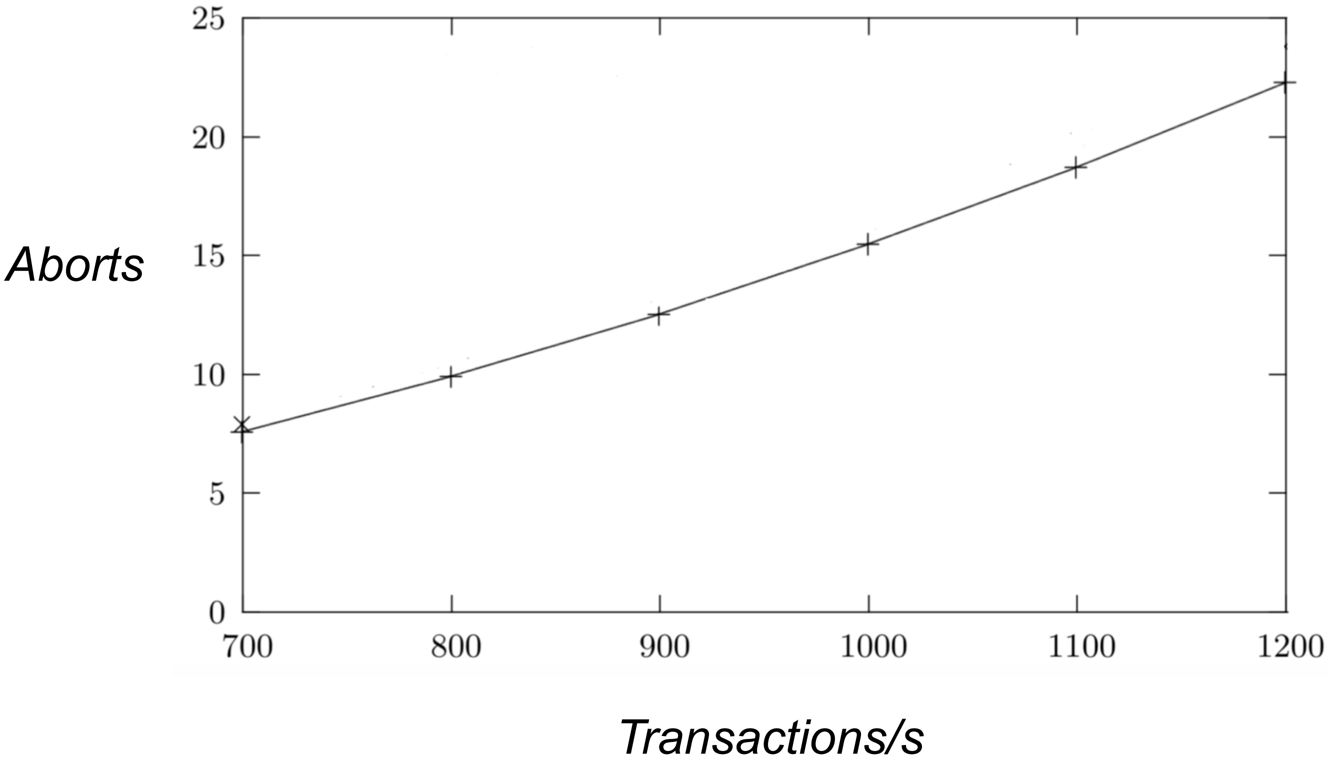
Collision Detection Protocol: Example



Collision Detection Protocol: Example



Collision Detection Protocol: Simulation Results



neo4j

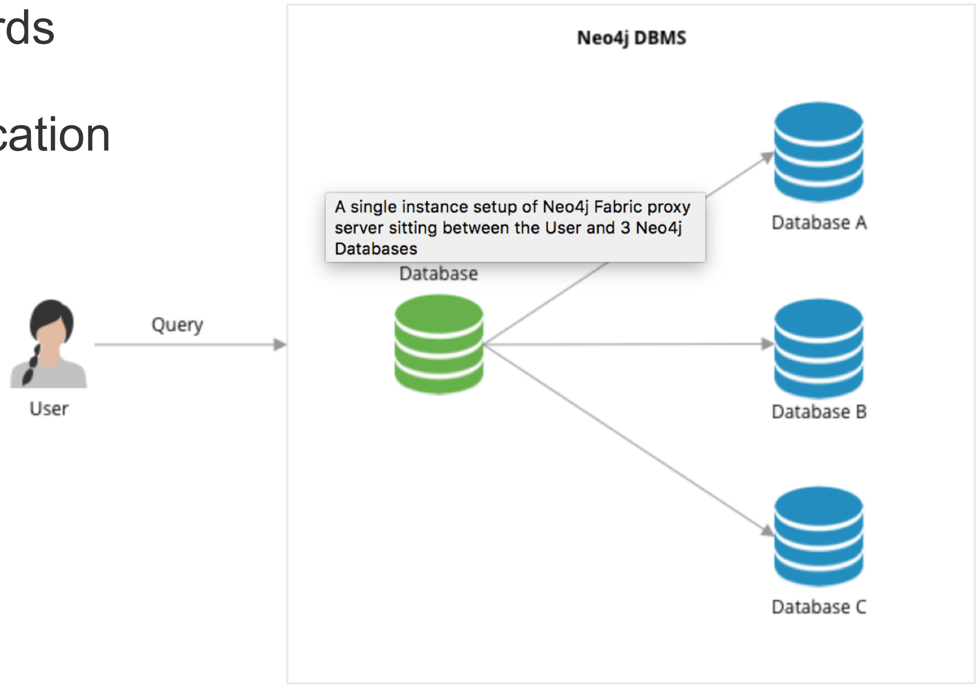
Neo4j Recent Developments

- **Fabric** introduced in 4.0 (January 2020)
- Store and retrieve data in multiple databases using a single Cypher query
- **Data Sharding**: the ability to access data available in distributed sources in the form of a **common graph partitioned on multiple databases**.

Neo4j Fabric

Queries cannot traverse across shards

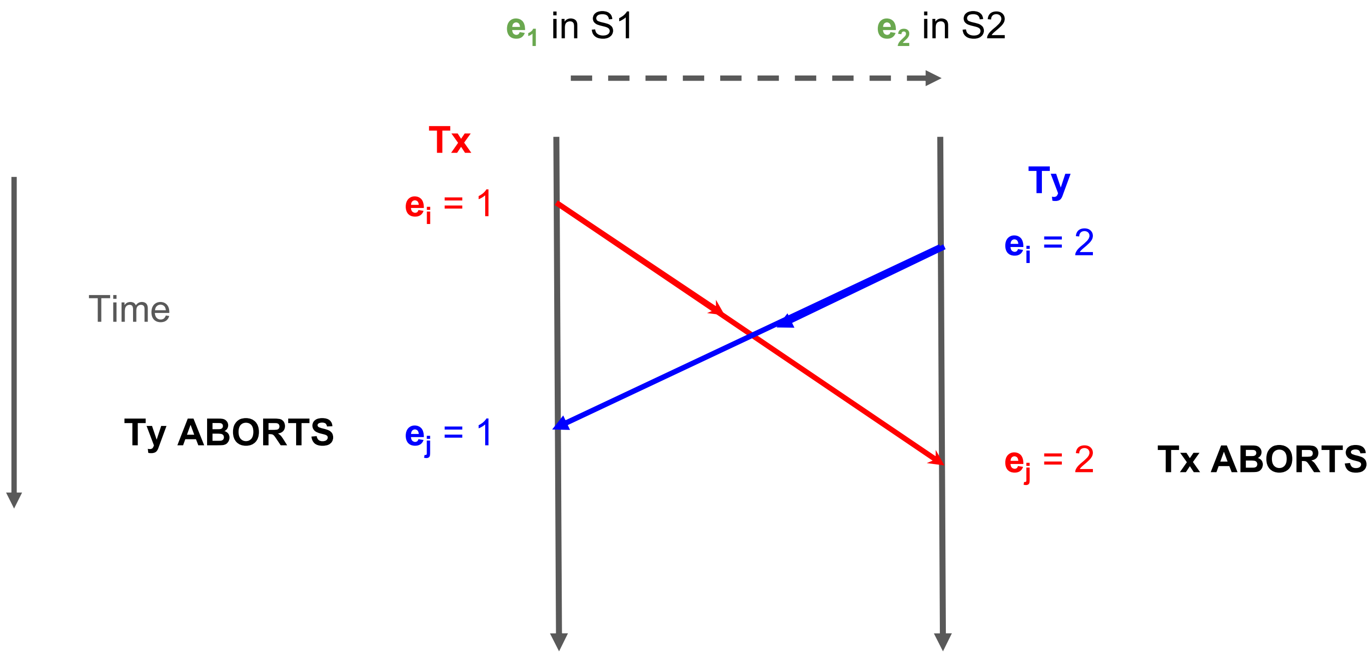
Needs a larger degree of data duplication



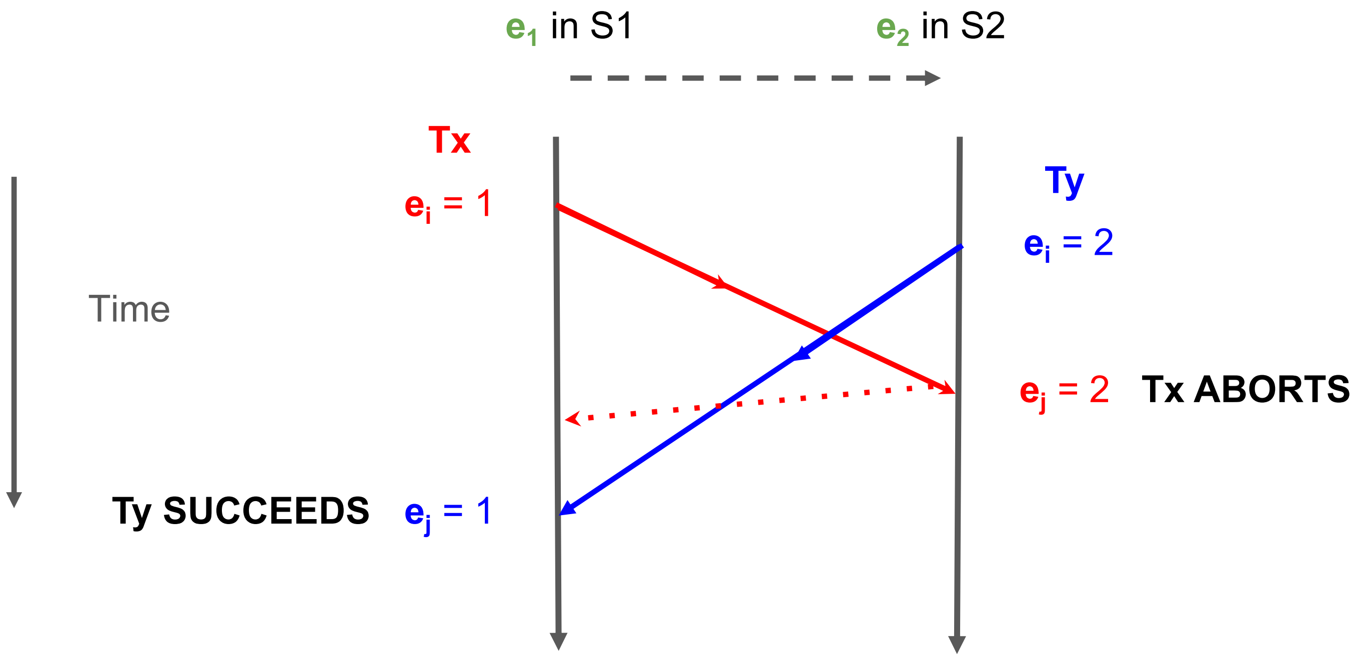
Summary

1. Data corruption happens and **spreads quick**
2. Probabilistic solutions reduce the time until corruption
3. **Imperative reciprocal consistency is preserved**
4. Deterministic mechanism that preserves reciprocal consistency at all times
5. Neo4j Fabric supports sharding at the cost of higher data redundancy

Collision Detection Protocol: Example 2



Collision Detection Protocol: Example 3



Neo4j Causal Clustering Architecture

- Default isolation: *Read Committed*
- Support for *Serializability* via explicit locking
- Clients get *read-your-own-writes* via bookmarking parameter for reads
- Replication semantics: *Causal Consistency*