RIOT: A New Leaderless Consensus Protocol

Jim Webber
Chief Scientist, Neo4j

George Theodorakis
Research Engineer, Nvidia

Hugo Firth
Staff Software Engineer, Neo4j

Natacha Crooks
Assistant Professor, UC Berkeley

Empathy

Being nice to each other and your future self is a key part of building resilient systems

Paxos “made simple”

1 Introduction

The Paxos algorithm for implementing a fault-tolerant distributed system has been regarded as difficult to understand, perhaps because the original presentation was Greek to many readers [5]. In fact, it is among the simplest and most obvious of distributed algorithms. At its heart is a consensus algorithm—the “synod” algorithm of [5]. The next section shows that this consensus algorithm follows almost unavoidably from the properties we want it to satisfy. The last section explains the complete Paxos algorithm, which is obtained by the straightforward application of consensus to the state machine approach for building a distributed system—an approach that should be well-known, since it is the subject of what is probably the most often-cited article on the theory of distributed systems [4].

Neo4j with Paxos = $O(n^k)$ support load

Disclaimer: neither Lamport or I want to destroy you. This image is intended for comic light relief only.

Raft is humane

- Raft's authors cared about people
- Distributed systems are hard to understand, harder to debug
- Always chooses the simple design:
  - Clear roles
  - Shared log (inductively correct)
  - Everything decided by majority
  - Strong leader role for ordering
- Handles
  - Leader election
  - Log replication
  - Membership changes

In Search of an Understandable Consensus Algorithm (Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.

In Search of an Understandable Consensus Algorithm, Ongaro and Ousterhout (USENIX ATC '14)

Neo4j with Paxos = $O(n^k)$ support load

Neo4j with Raft = (log n) support load

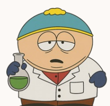
Ongaro and Ousterhout, 2014

What Raft taught us about distributed systems

“Be awesome to each other.” – Bill and Ted.

- E.g. smaller quorums *are* possible

- But simple majorities make for straightforward ops



  - Ask the 4am on-call engineer about it

- Simpler protocols have fewer places for bugs to hide
  - Bugs in distributed systems are a nightmare to find and solve

Fast Flexible Paxos: Relaxing Quorum Intersection for Fast Paxos

Heidi Howard
University of Cambridge
heidi.howard@cl.cam.ac.uk

Aleksey Charapko
University of New Hampshire
aleksey.charapko@unh.edu

Richard Mortier
University of Cambridge
richard.mortier@cl.cam.ac.uk

ABSTRACT

Paxos, the de facto standard approach to solving distributed consensus, operates in two phases, each of which requires an intersecting quorum of nodes. Multi-Paxos reduces this to one phase by electing a leader but this leader is also a performance bottleneck. Fast Paxos bypasses the leader but has stronger quorum intersection requirements.

In this paper we observe that Fast Paxos' intersection requirements can be safely relaxed, reducing to just one additional intersection requirement between phase-1 quorums and any pair of fast round phase-2 quorums. We thus find that the quorums used with Fast Paxos are larger than necessary, allowing alternative quorum systems to obtain new tradeoffs between performance and fault-tolerance.

performance by adjusting quorums depending on the phase of the algorithm [1, 4, 8, 9, 28, 35].

Paxos is usually implemented using Multi-Paxos [18, 19], an optimization that elects one node to be a *leader*. This single leader can then achieve distributed consensus in just one phase, but unfortunately also becomes a performance bottleneck.

Seeking to improve performance, a new family of *leaderless* consensus algorithms emerged, starting with Fast Paxos [23], which forms the basis for subsequent algorithms including Generalized Paxos [22] and Egalitarian Paxos [26]. Paxos uses the idea of *rounds* in which at most one value can be *proposed*. Fast Paxos introduced the notion of *fast rounds* where multiple values can be safely proposed in the same round. However, such fast rounds require stronger quorum intersection than classical rounds. Specifically,

Howard et al: Fast Flexible Paxos: Relaxing Quorum Intersection for Fast Paxos (ICDCN 2021)

If you ever have to choose between anything (except safety) and simple, always choose simple

The challenge of consensus for graphs

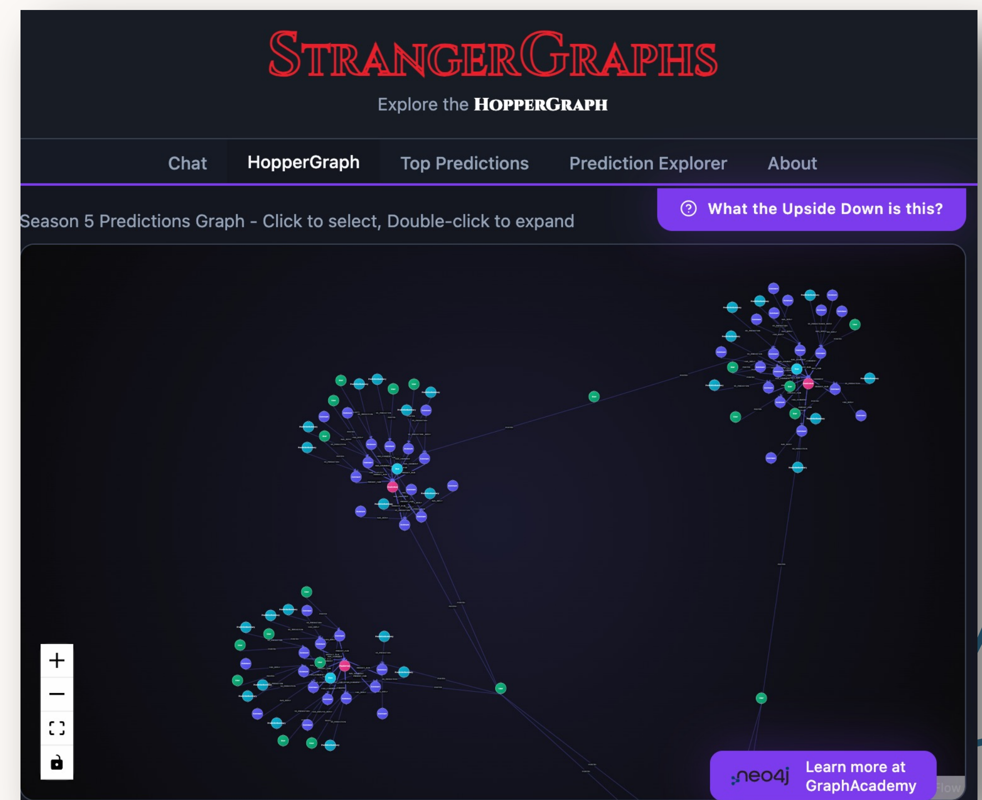
Joining the dots for safe, scalable graph databases

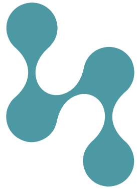
Robinson, Eifrém, and Webber, Graph Databases (O'Reilly) 2015

Labelled Property Graph Model

A humane graph model consisting of nodes, relationships, labels and property data.

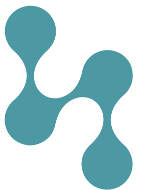
- Neo4j (and other graph databases) implement a humane and performant data model
  - Nodes represent entities
    - Have zero or more labels that describe their role
    - Have zero or more properties (key-value pairs) for entity data
  - Relationships connect nodes
    - Have type *and* direction
  - Model affordances
    - Any two nodes can be connected
    - By any number of relationships
    - Of any type
- **Cost of traversing** a relationship at query time is **constant, irrespective of type or direction**





Any two nodes can be connected
by any number of relationships
of any type in either direction.

This forms an implicit contract with users



Reciprocal Consistency

In the absence of coordination, your graph is garbage in a matter of months.

If `(rosa)-[:FOLLOWS]->(antonio)`

Then Antonio *is implicitly followed by* Rosa

Easy on a single server or leader-based replica

Just pointers

Very hard with sharded setup

The relationship record `-[:FOLLOWS]->` can cross shards

It must be carefully coordinated so that both shards always agree on its state

Queueing Models and
Service Management
Vol. 3, No. 2, page 235-253, 2020

QMSM
© PU 2018

Modeling the Gradual Degradation of Eventually-Consistent Distributed Graph Databases

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

[1]School of Computing, Newcastle University
NE4 5TG, United Kingdom
[2]Neo4j UK, Union House, 182-194 Union Street, London
SE1 0LH, United Kingdom

(Received March 2020 ; accepted June 2020)

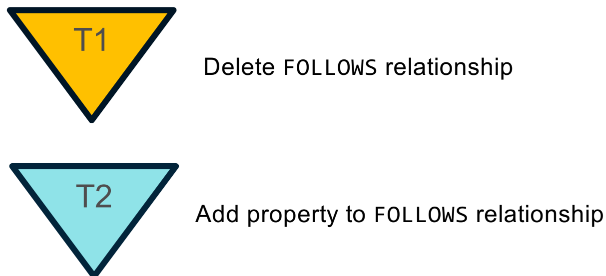
Abstract: Under the 'eventual consistency' approach to updates in a distributed graph database, it is possible that edge information may be corrupted. Errors may then be propagated to other parts of the database by subsequent queries. The process by which this occurs is modeled, with the aim of estimating the time that it takes for a clean database to become degraded to the point of being unusable. A fluid approximation is developed and two solution methods are proposed. The accuracy of those solutions is examined thoroughly, for databases with different sizes, structures and parameter settings, using simulations as a basis of comparison.

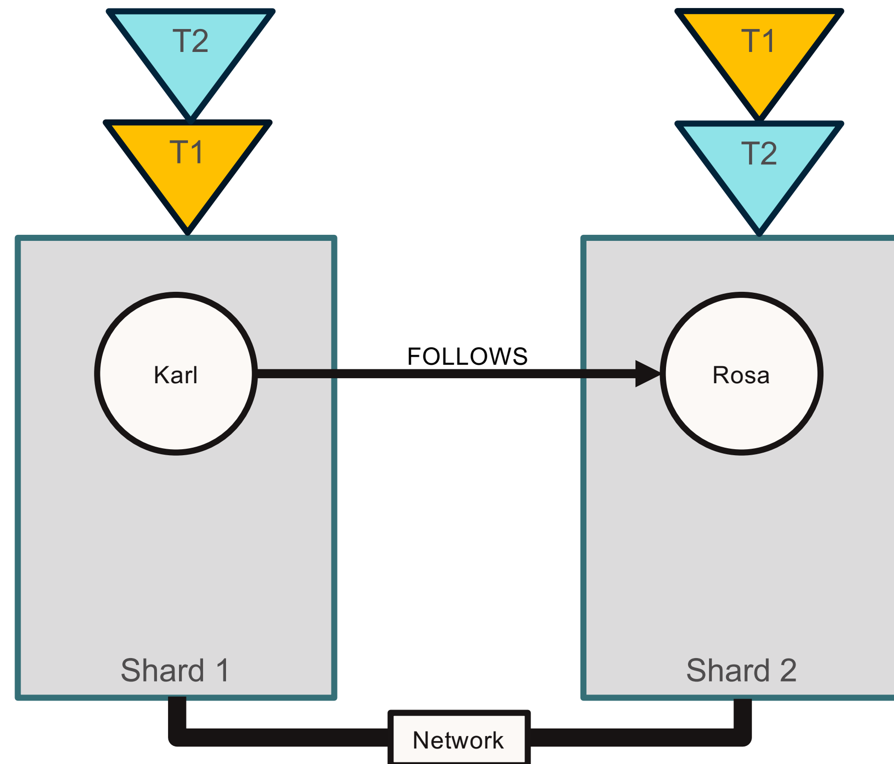
Keywords: Analytical evaluation, distributed edges, graph databases, half-corrupted edges, networked servers, simulations.

Ezhilchelvan et al: Modeling the Gradual Degradation of Eventually-Consistent Graph Databases (QMSM 2020)

A Sharded Graph Database

Puzzle: does the `FOLLOWS` relationship exist after both T1 and T2 have finished executing?





In this case, the answer is not defined which will lead to divergence and then to data corruption.

Do not let corruption take hold

Transaction protocol for distributed graphs

Since we can't fix divergent graphs, we have to prevent them from happening.

Cannot merge divergent graphs back together

Must prevent graphs diverging in the first place

If we can keep a majority of servers compatibly synchronised, then we **we can maintain correctness.**

But in doing so **we cannot make the system slow or impractical.**

RIOT

noun

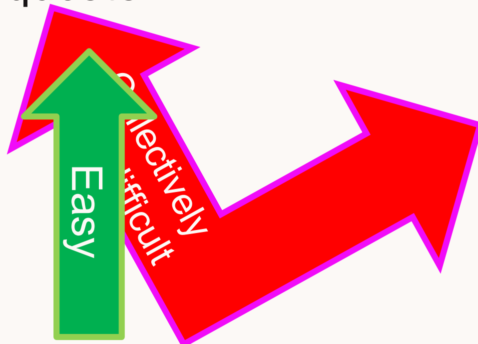
- A violent public disturbance caused by a group of people, often involving disorderly behaviour.
- A tortured acronym for *Replicated Independently Ordered Transactions.*

What do we need to build a safe and scalable graph database?

RIOT Requirements

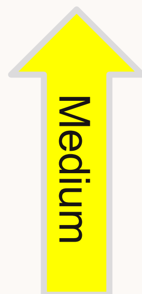
Scale

All servers need to be able to process read and write requests.



Correctness

Servers should not diverge even under challenging concurrency and fault conditions. It must be able to uphold reciprocal consistency.

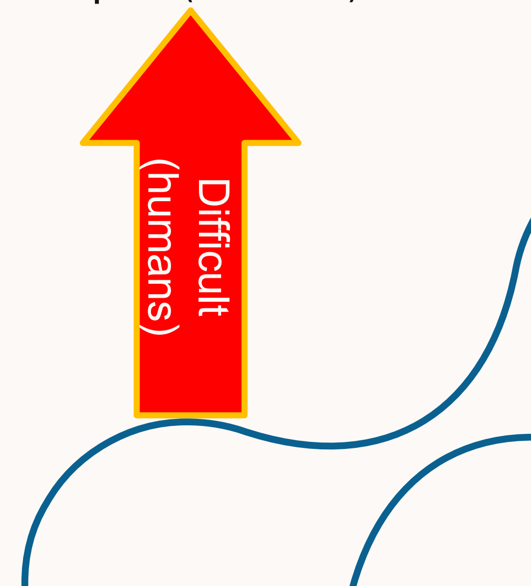


Redundancy

Multiple copies of the data need to be stored for fault-tolerance.

Practicality

The system should be as simple as possible, but no simpler (c.f. Raft).



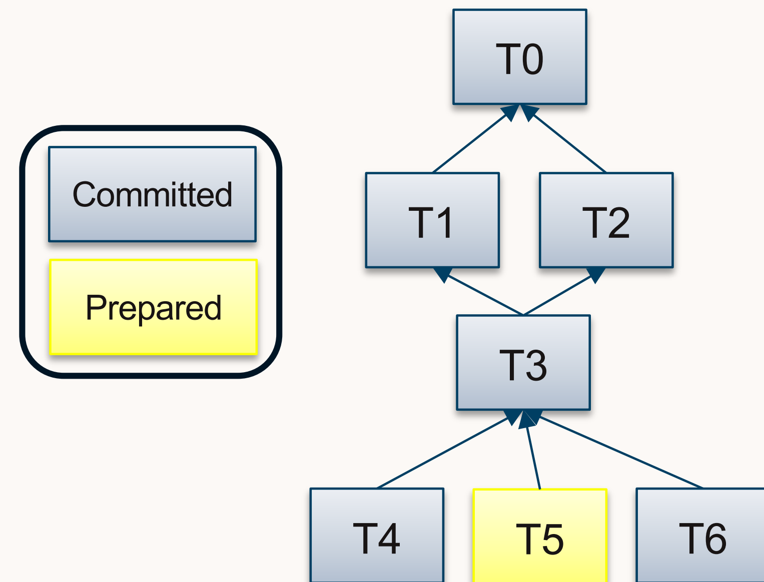
Let’s start a RIOT

Leaderless distributed transactions in a single shard

Servers make independent decisions based on their local state and incoming messages

RIOT Server State

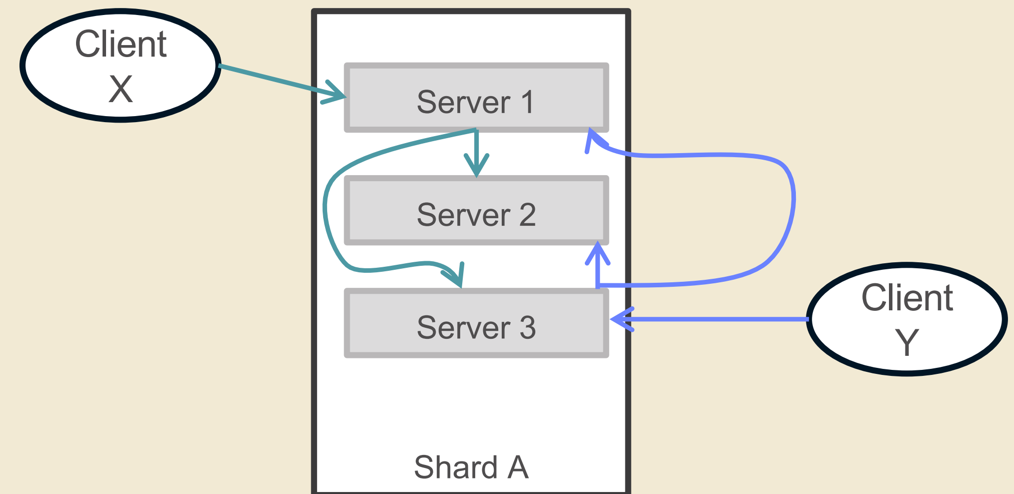
- All servers contain a DAG of transactions known as a TxDAG
- Two states: `PREPARED` and `COMMITTED`
- `PREPARED` transactions always leaf nodes
- The most recently `COMMITTED` transactions are leaf nodes
- Older `COMMITTED` transactions are non-leaf nodes
  - They become ancestors against which newer transactions prepare



Set of committed leaf nodes is called the **Leading Edge**
In this case: {T4, T6}

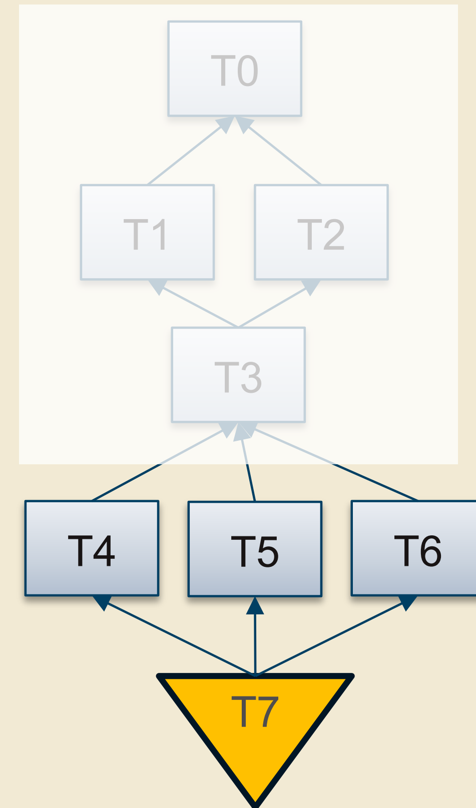
Single Shard Rules

- Allow any server to be a coordinator for scalability
- Use history metadata to determine compatible (not necessarily identical) histories for updates
- Voting model is simple majority within a shard



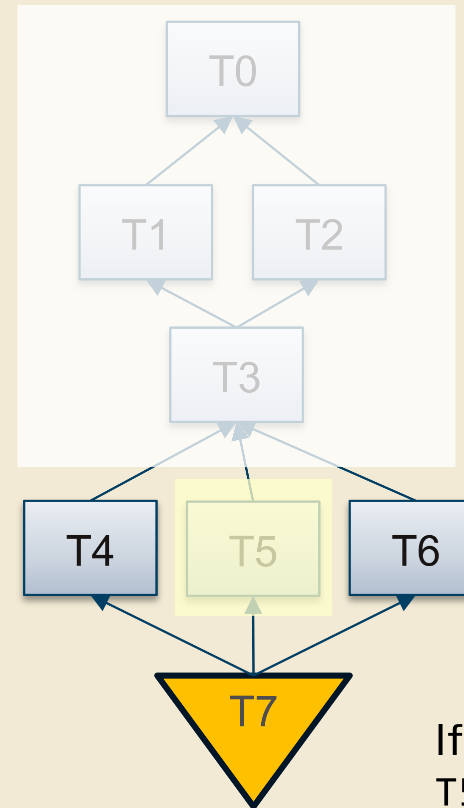
Single Shard Rules

- Allow any server to be a coordinator for scalability
- Use history metadata to determine compatible (not necessarily identical) histories for updates
- Voting model is simple majority within a shard



Single Shard Rules

- Allow any server to be a coordinator for scalability
- Use history metadata to determine compatible (not necessarily identical) histories for updates
- Voting model is simple majority within a shard



If T7->T5, we know that T5 must have been committed (by a majority) elsewhere, and so we can commit it here and let T7 continue.

Single Shard Rules

- Allow any server to be a coordinator for scalability
- Use history metadata to determine compatible (not necessarily identical) histories for updates
- Voting model is by simple majority within a shard

In Search of an Understandable Consensus Algorithm
(Extended Version)

Diego Ongaro and John Ousterhout
Stanford University

EASY TO UNDERSTAND

Abstract

Raft is a consensus algorithm for managing a re[illegible]d log. It produces a result equivalent to (multi-[illegible]axo[illegible] it is as efficient as Paxos, but its [illegible]ure i[illegible]diff[illegible] from Paxos; this makes R[illegible]ore [illegible]understand[illegible]han Paxos and also provi[illegible]s a be[illegible]r f[illegible]nda[illegible]on for building practical [illegible] order to [illegible]hance understandability, Ra[illegible]sepa[illegible]es the [illegible]y elements of consensus, such as [illegible]eader [illegible]ection, lo[illegible]rep[illegible]ation, and safety, and it enforces [illegible]on[illegible]g[illegible]of coherency to reduce the number of [illegible]tes th[illegible]t must be considered. Results from a user study d[illegible]monstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

[illegible]tate [illegible] red[illegible]ti[illegible] (relative to Paxos, Raft reduces the [illegible]gre[illegible] o[illegible] [illegible]eterminism and the ways servers can be in[illegible]c[illegible]ent with each other). A user study with 43 students [illegible]t two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [29, 22]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of

A (mostly) Healthy Shard

Consistent Majority
These two servers have applied the same transactions at this point in time.

Compatible minority
This server might just be a little slower (e.g. far away geographically), or it received messages in a slightly different, but compatible order.

A (mostly) Healthy Shard

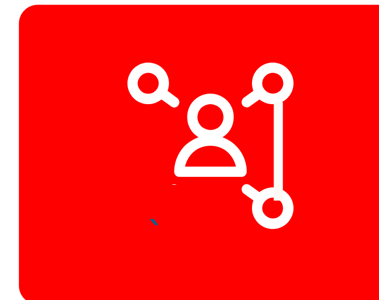
Consistent Majority

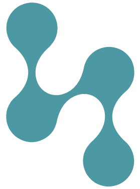
Safe to base future transactions from the state of either of these

Diverged minority

Cannot safely base future transactions on this state at this point in time

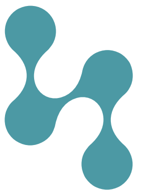
Repair (catch up from majority using a graph-diff algo) will happen as normal part of the protocol





In the following slides you will see identifiers represented as natural numbers because it looks nicer. In our system, the identifiers are UUIDs.

Remember to read them as IDs, not numbers otherwise it gets really confusing.



Happy path

Servers are fault-free and non-divergent.

Client initiates transaction on existing system

PD{}
LE{1, 2}

PD{}
LE{1, 2}

PD{}
LE{1, 2}

Client

Server 1

Server 2

Server 3

Begin(3)

Coordinator’s TxDAG Leading Edge equal to local leading edge

PD{}
LE{1, 2}

PD{}
LE{1, 2}

PD{}
LE{1, 2}

Client

Server 1

Server 2

Server 3

Prepare(3)->
{1, 2}

Prepare(3)->
{1, 2}

Prepare(3)->
{1, 2}

Majority confirmed by the coordinator for `TxID 3`

PD{3}
LE{1, 2}

PD{3}
LE{1, 2}

PD{3}
LE{1, 2}

Client

Server 1

Server 2

Server 3

Prepared(3)->
{1, 2}

Prepared(3)->
{1, 2}

Prepared(3)->
{1, 2}

Commit

PD{}
LE{3}

PD{}
LE{3}

PD{}
LE{3}

Client

Server 1

Server 2

Server 3

Commit(3)->{1, 2}

Commit(3)->{1, 2}

Commit(3)->{1, 2}

Prepared(3)->
{1, 2}

After prepare concludes, commit is inevitable (unless the system breaches its fault tolerance level, M = 2F +1)

Client receives outcome

PD{}
LE{3}

PD{}
LE{3}

PD{}
LE{3}

Client

Server 1

Server 2

Server 3

Committed(3)

Committed(3)

Committed(3)

Committed(3)

Normal operation under the laws of physics.

Servers are fault-free but are ahead or behind each other because of asynchrony.

Client initiates transaction on existing system

PD{}
LE{1, 2}

PD{}
LE{1, 2}

PD{2->0}
LE{1}

Client

Server 1

Server 2

Server 3

Begin(3)

Server 3 is slightly behind the others.
It has prepared but not yet committed Tx2

Chosen Coordinator Issues PREPARE in parallel

PD{}
LE{1, 2}

PD{}
LE{1, 2}

PD{2->0}
LE{1}

Client	Server 1	Server 2	Server 3

Prepare(3)->
{1, 2}

Prepare(3)->
{1, 2}

Prepare(3)->
{1, 2}

Coordinator's TxDAG Leading Edge compared to local version

PD{}
LE{1, 2}

PD{}
LE{1, 2}

PD{2->0}
LE{1}

Client

Server 1

Server 2

Server 3

Prepare(3)->
{1, 2}

Prepare(3)->
{1, 2}

Prepare(3)->
{1, 2}

TxDAG Leading Edge compared to local history

P{2->0}
C{1}

Server 3

P{3}
LE{1,2}

≠

PD{2->0}
LE{1}

But given LE{1,2} from S1

... can commit PD{2->0} to LE{1, 2}

Prepare(3) -> {1, 2}

NB: Safe because Tx2 by definition must have been committed by a majority elsewhere

Update local leading edge and respond

PD{3}->
LE{1, 2}

Server 3

S3 was able to learn some state from its peers and move forward

Prepared(3)->{1, 2}

Protocol completes normally from here

Divergent participant

Always abide by majority decisions, never equivocate.

Server 1 is up-to-date coordinator, Server 3 is stale/divergent

PD{}
LE{1, 2}

PD{}
LE{1, 2}

PD{}
LE{}

Client	Server 1	Server 2	Server 3

Begin(3)

Prepare(3)->
{1, 2}

Prepare(3)->
{1, 2}

Prepare(3)->
{1, 2}

Majority formed, coordinator observes divergence

PD{3}
LE{1, 2}

PD{3}
LE{1, 2}

PD{}
LE{}

Client

Server 1

Server 2

Server 3

Prepared(3)->
{1, 2}

Prepared(3)->
{1, 2}

Abort(3)->
{}

Coordinator can distinguish divergence from e.g. underlying database abort using response leading edges

Majority commits 3->{1, 2}, Server 3 needs to repair

Participant finds out the hard way that it is divergent

Divergent coordinator

Will not form a majority and therefore make no updates.

Server 3 is coordinator and is stale/divergent

Server 2 and 3 Respond Positively, with Qualified Votes

PD{}
LE{1, 2}

PD{}
LE{1, 2}

PD{}
LE{}

Client

Server 1

Server 2

Server 3

Commit(3)->
{}

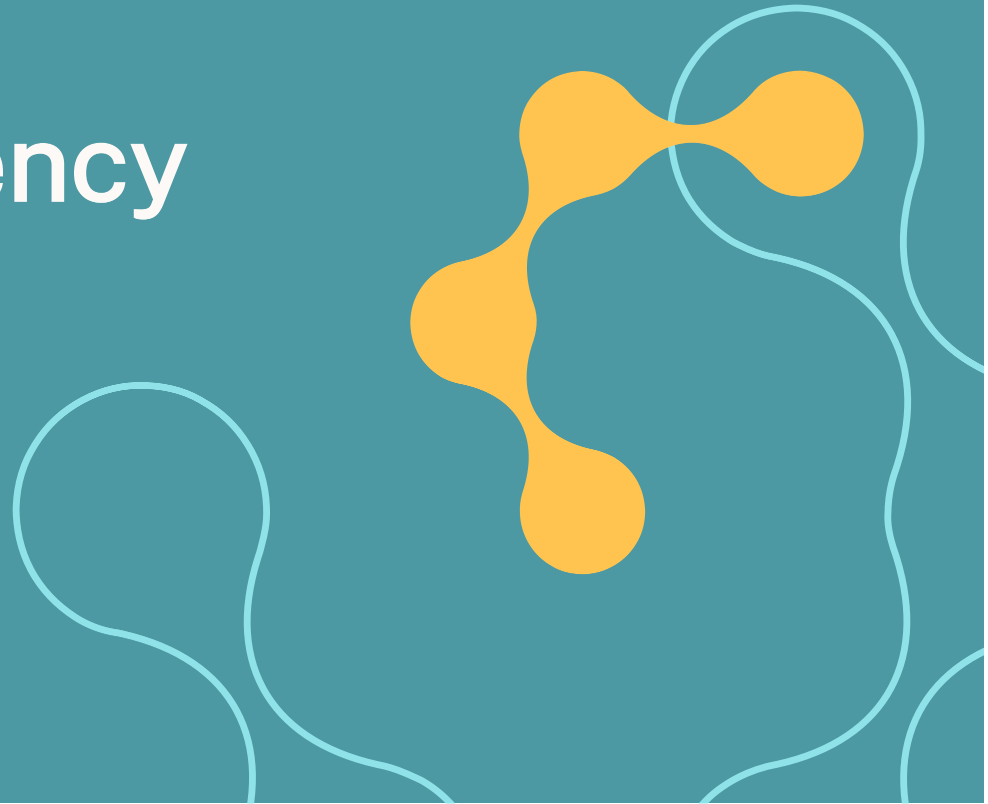
Prepared(3)->
{1, 2}

Prepared(3)->
{1, 2}

Server 3 realizes it is in a divergent minority and aborts

Improving concurrency and throughput

Can we loosen LE checks and maintain correctness?



Leading Edge Equality and Compatibility

***Equality* is simple, safe but has low throughput**

- Coordinator sends LE to participants
  - A bit like Etags on the Web
- Participants check is incoming leading edge is *equal*
- Simple implementation, maintain set of `COMITTED` leaf nodes atop the TxDAG
- Gets harder with number of concurrent transactions

***Compatibility* has better throughput, but is still abort heavy (30-40%)**

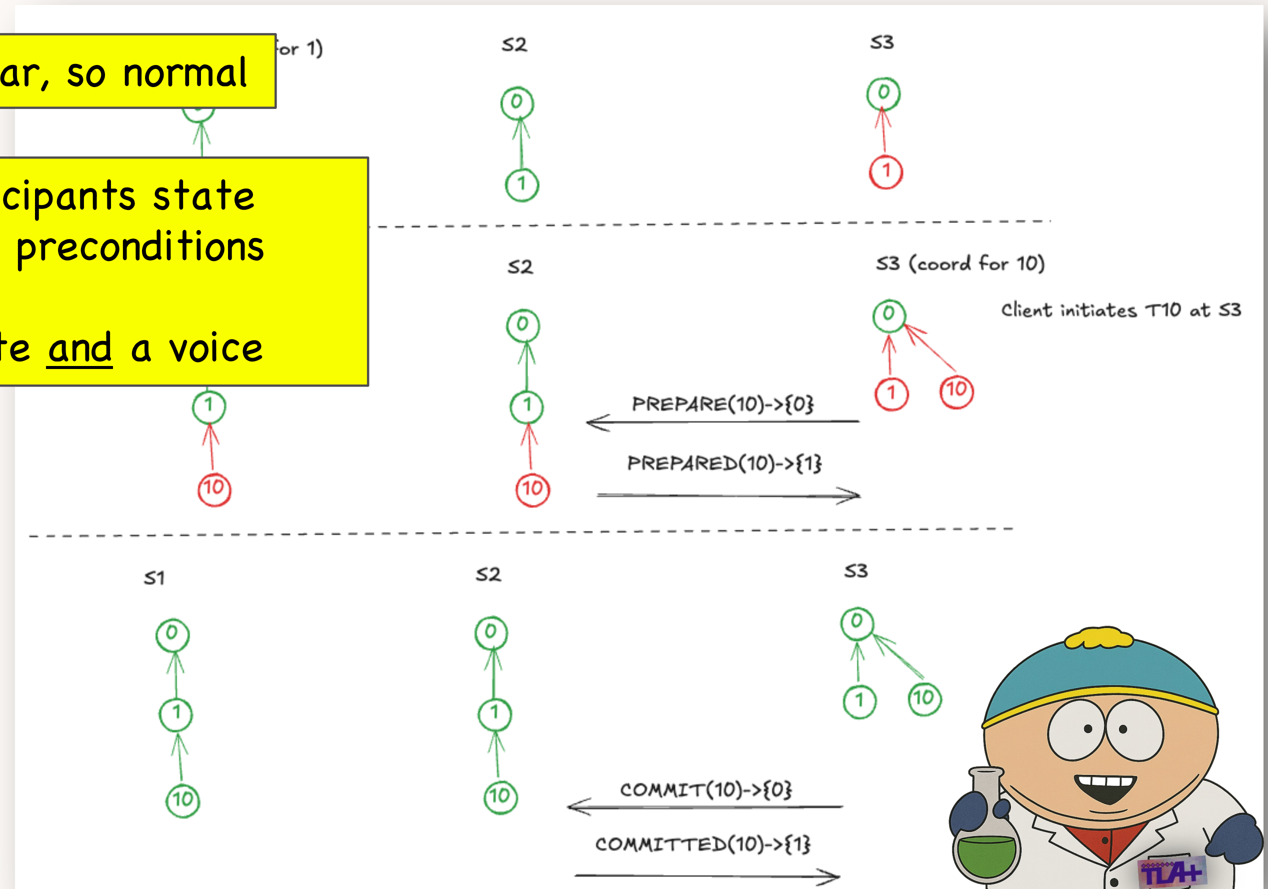
- Coordinator sends message with its leading edge
- Participants commit any transaction they have prepared that are in the incoming leading edge
- Participants check is incoming leading edge is *compatible*
  - Incoming leading edge can be matched onto local leading edge
- Gets harder with number of concurrent transactions

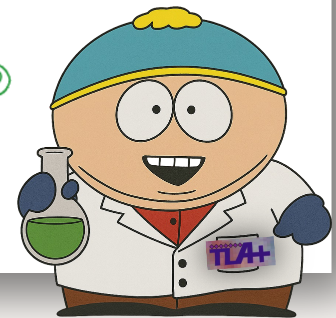
Bilateral Leading Edge History Checks

- Participant has seen all the coordinator's leading edge transactions *at some point*
- Coordinator has seen all the participant's leading edge transactions *at some point*
- They cannot significantly diverge because one side would abort
- But they can safely diverge (a little) where RMs determine that operations commute

So far, so normal

Participants state their preconditions too!
A vote <u>and</u> a voice



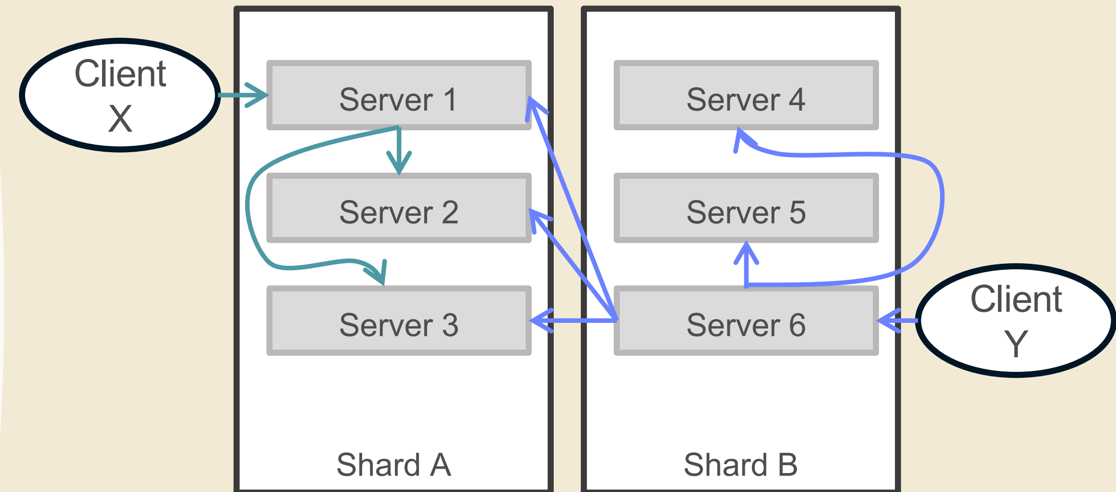


Full RIOT

Simple conjunction of majorities for transactions across many shards

Multi-Shard Rules

- Allow any server to be a coordinator for scalability
- Use metadata to determine compatible histories for updates within each shard
- *Across* involved shards a unanimous decision is required But only majority needed *within* each shard



Committable Conjunction of Majorities

All Involved Shards Vote Commit

Even though there are a mixture of unanimous and majority commits within the individual shards

Shard1

Shard2

Shard3

Non-Committable Outcome

Some shards vote to abort

Protocol needs unanimous agreement in prepare phase to continue to commit. In this case, we must abort.

Shard1

Shard2

Shard3

Optimising Cross-Shard Transactions

Interposed Coordinators

Shards

Each shard operates as an independent unit with independently operating servers just like in the single-shard protocol.

Within a shard, the coordinator appears to operate as normal.

Participants are unaware of the multi-shard nature of the transaction.

Coordinators

The first coordinator that the client contacts on any shard owns the whole transaction.

When the transaction flows to another shard, one of its servers is enrolled into the as a participant globally, but a coordinator locally.

Multi-Shard Prepare

Client	Shard 1	Shard 2	Shard 3

Begin(456)

Prepare(456)->
{401,402}

Prepare(3088->456)
->{3044, 3055, 3067,
3068}

Multi-Shard Commit

Client

Shard 1

Shard 2

Shard 3

Commit(456)->
{401,402}

Commit(456)

Commit(3088->456)
->{3044, 3055, 3067,
3068}

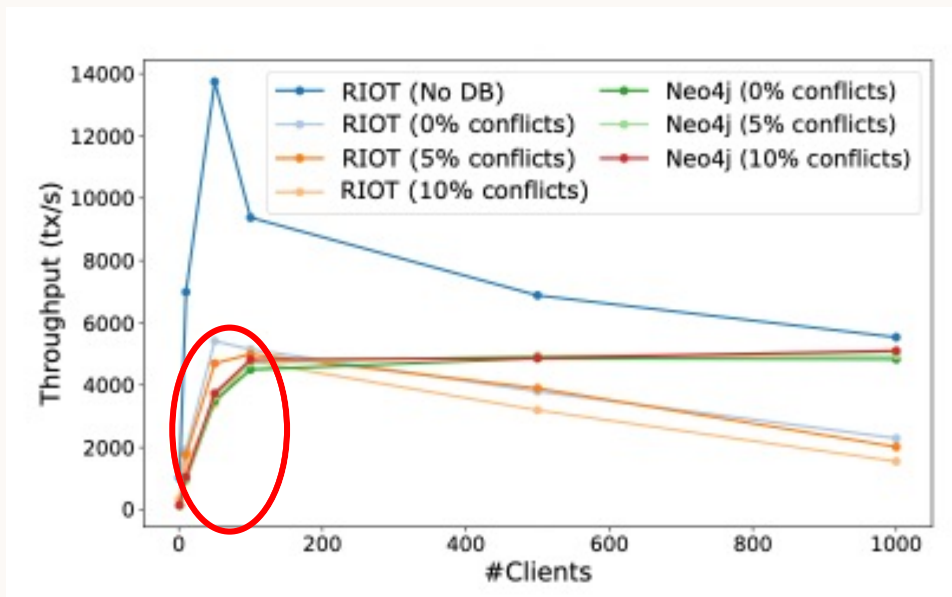
Evaluation

Does RIOT work? Is it useful?

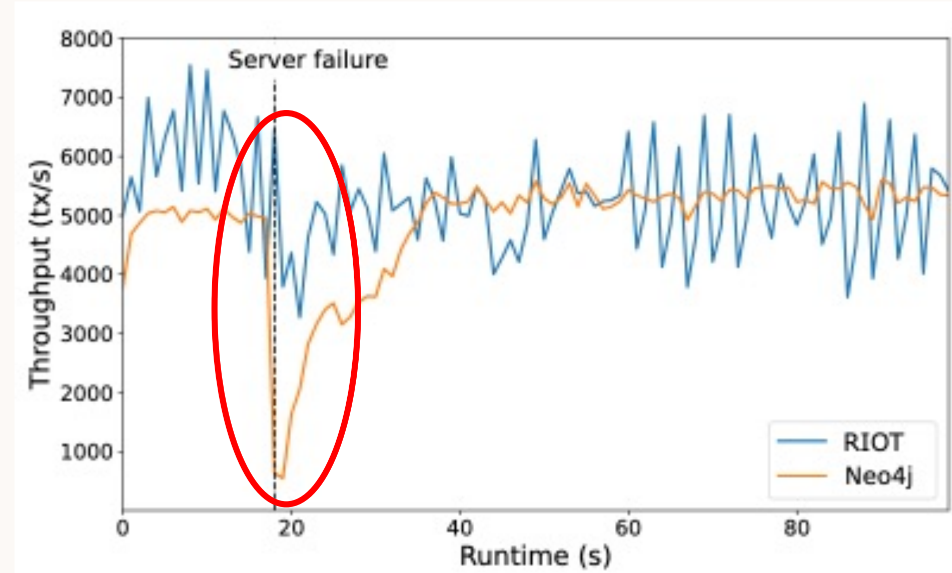
The Good Parts

Evaluation

Throughput vs Clients (5 servers)

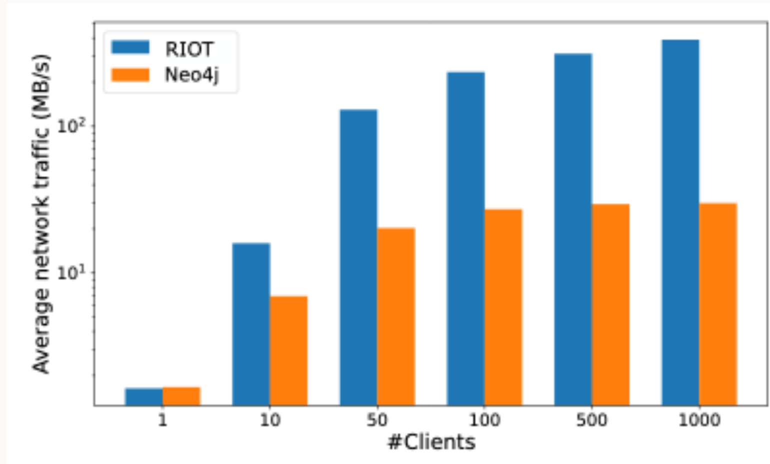


Failure Mode

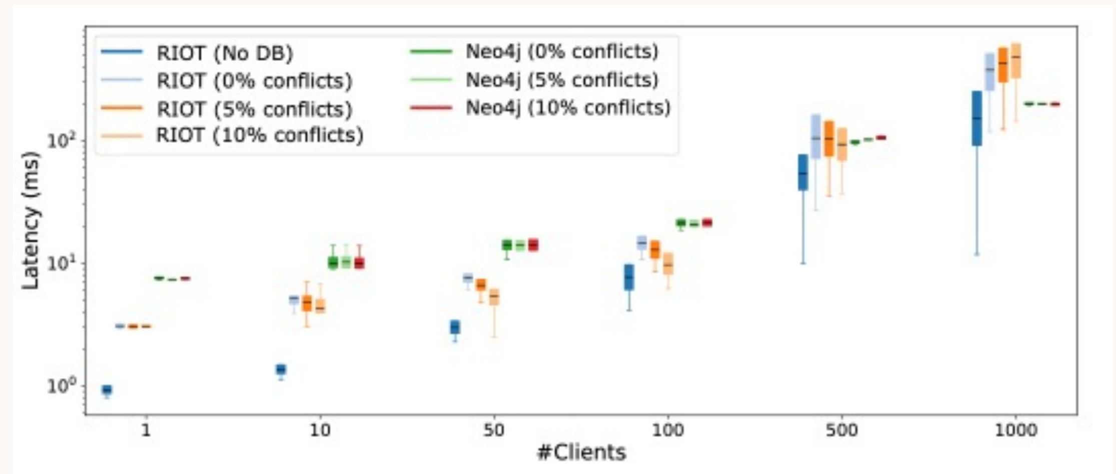


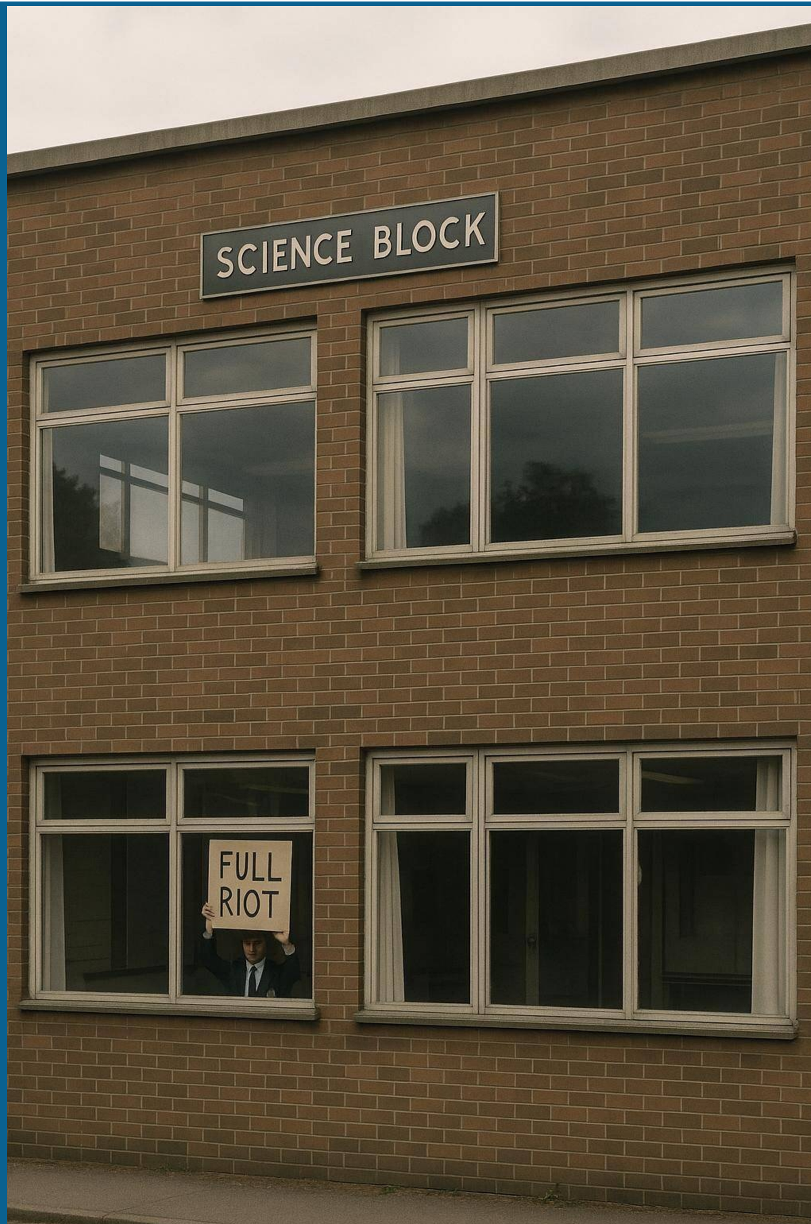
Evaluation

Traffic per machine



Latency (5 servers)





Collaborators

Tobias Lindåker
Relational AI

Paul Ezhilchelvan
Isi Mitrani
Newcastle University

Junhao Hu
Michael Cahill
Alan Fekete
University of Sydney

Any questions?

Jim Webber | Chief Scientist | Neo4j
jim@neo4j.com