Challenges and Choices in building a Replicated, High-Throughput, Distributed ACID Transactions Management System

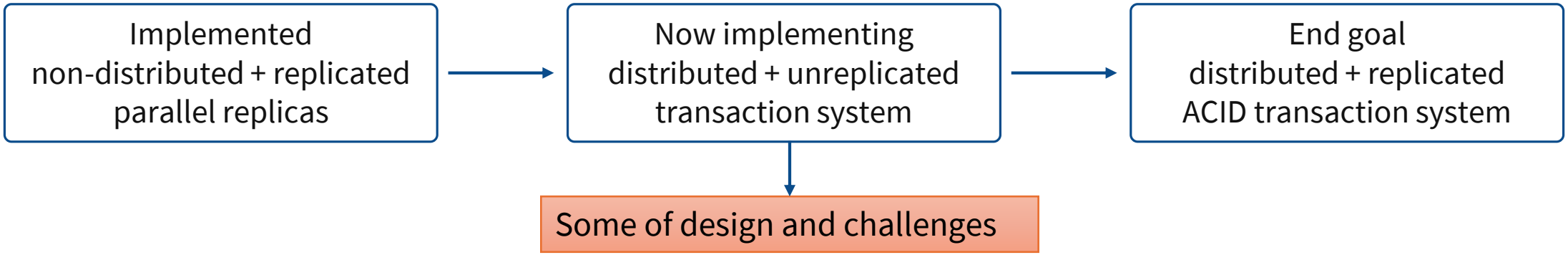
P. Ezhilchelvan[1], Y. Wang[1], and J. Webber[2]

April 2026

1. Newcastle University UK
2. Neo4j UK

Research trajectory

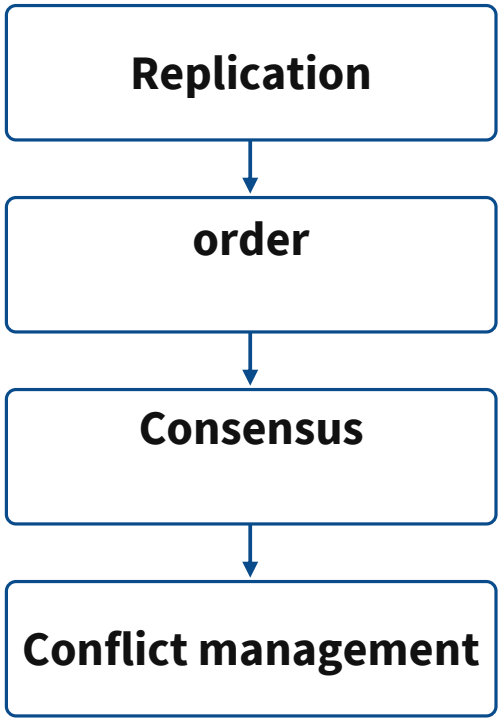
From a replicated centralized prototype to a distributed and unreplicated database system, with the final goal of combining distributed and replicated parallel processing.



The system challenge: aware of fundamentals to avoid duplication

A good design should avoid extra coordination overhead and ideally let one mechanism provide what another needs.

Example 1:



Ordering gives consensus “for free”
→ precedence-based conflict handling can use order, not global timestamps.

Avoid: timestamp dependence / global time base

Example 2:

2PC is essential for atomic commit – simpler form of consesnsus

But using centralised consensus for replication becomes a throughput killer

Needed: decentralised replication order that builds a 1-server abstraction and works cleanly with 2PC

Three choices that shape the architecture

The present research sits on a sequence of earlier design decisions and prototype systems.

1 No-Wait/Limited-Wait concurrency control

WHY

Selected **Instantaneous Abort** and **Hybrid version** through modelling, simulation, and implementations

PAYOFF

low overhead · low abort

(PhD thesis, 2023)

2 Ring-based Ordering with parallel processing

WHY

merge concurrent incoming streams into one global conflict-free transaction stream

PAYOFF

high-throughput ordering

(PhD thesis, 2025)

3 RAFT-inspired recovery adaptation

WHY

reuse mature recovery ideas with minor tweaks in a decentralised ring setting

PAYOFF

throughput + recoverability

(PhD thesis, 2024)

Background: **Limited-wait hybrid** concurrency control

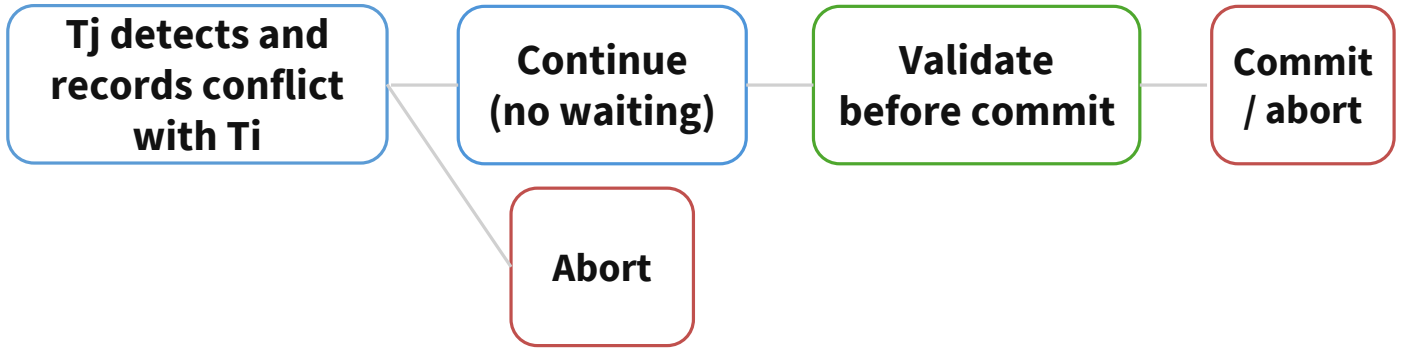
This **limited-wait hybrid** concurrency control enforces serializability by recording precedence relations and validating on conflicts before commit.

Ti → Tj means

Precedence edge

Tj depends on Ti

Ti is the predecessor; Tj is the successor.



Conflict Management

W-W conflicts: resolved instantaneously by aborting Tj to avoid multiple uncommitted writers.

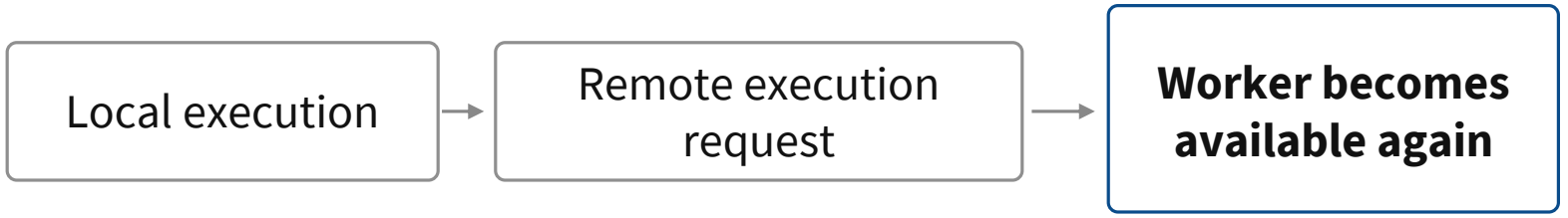
Read-related conflicts: recorded but not resolved until the validating

- R-W: wound any still active predecessor;
- W-R: abort if exists any uncommitted predecessor;

Challenge 1: what should a worker thread do while waiting for a remote shard?

When a distributed transaction needs remote execution, the local worker can either:

- Block for remote transaction response, or
- Continue by picking another queueing job



Candidate jobs:

- Fully executed transaction in / awaiting validation
- Partially executed transaction
- New transaction

Required priority on picking another queueing job

Required priority:

Wrong alternative A — prioritize new transactions first

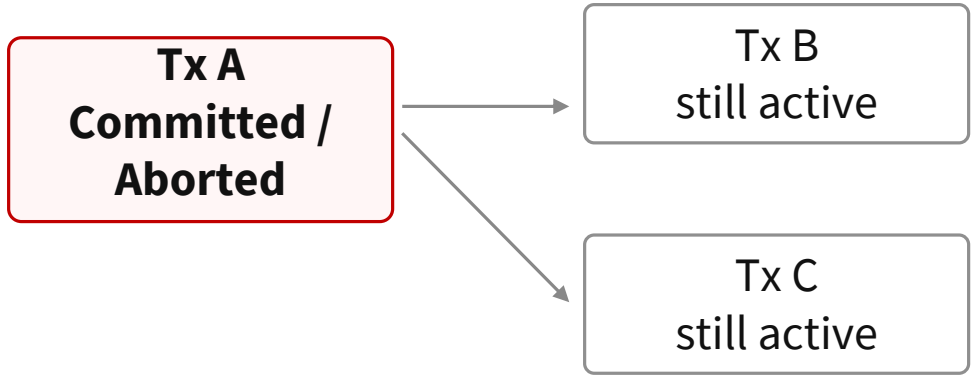
- Continuous arrivals keep starting fresh work
- Older transactions do not complete execution or 2PC
- This results a **livelock** risk under sustained load

Wrong alternative B — partial work above validation-ready work

- It inflates “concurrent” transactions
- Later transactions reach validation while conflicting predecessors are still unfinished
- More active predecessors result **in aborts and higher response time**

Challenge 2: when can completed transaction metadata be reclaimed safely?

In limited-wait hybrid concurrency control, transaction termination alone does not justify reclamation: completed transactions' metadata may still be needed by active successors.



- Immediate deletion is **unsafe** under delayed validation and asynchronous races.
- Keeping every finished context forever is too expensive.
- Reclamation therefore must be delayed, dependency-aware, and lightweight.

Tx A metadata must stay alive until all successors complete.

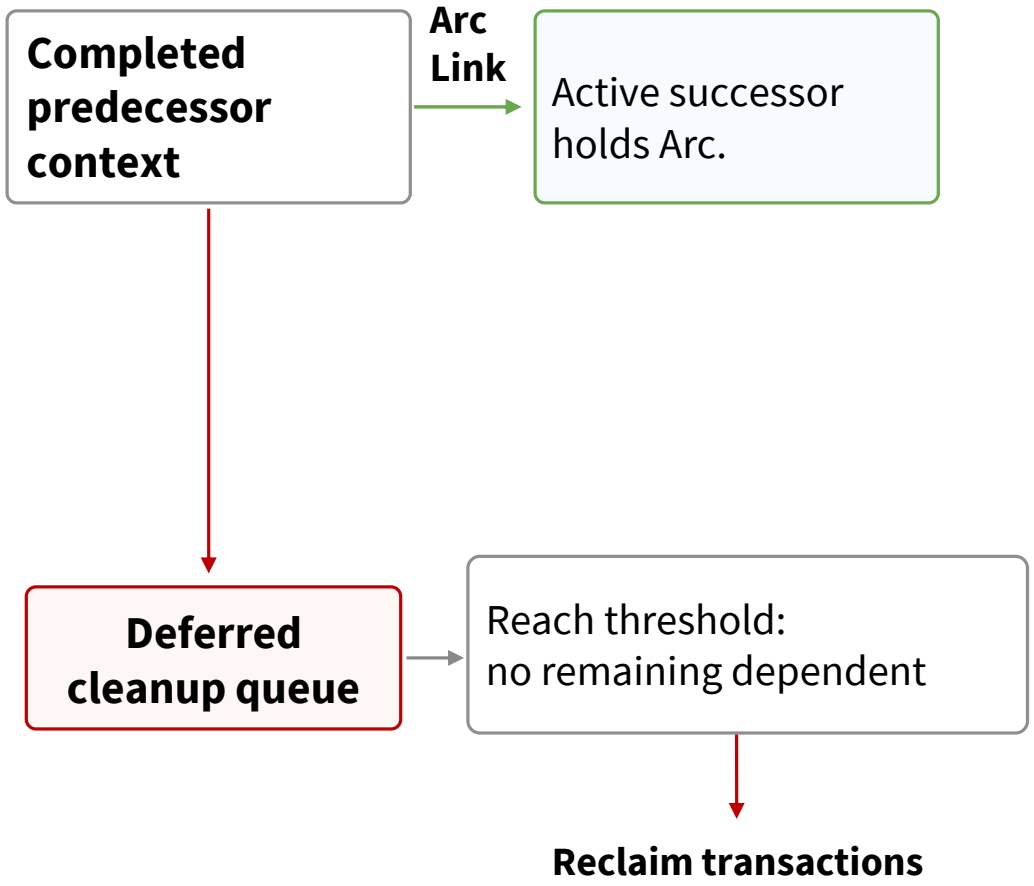
Arc-assisted delayed garbage collection

A completed transaction cannot be reclaimed while an active successor may still need its state.

Arc stands for **Atomically Reference Counted**, a Rust smart pointer that allows multiple threads or components to safely share ownership of the same data.

Arc-based retention rule

- Each transaction owns an **Arc** marker.
- A recorded predecessor edge gives the successor an **Arc**.
- After a transaction's termination, the context is reclaimed when no dependent remains.



Arc acts as a lightweight liveness mechanism: retention follows dependency structure, not fixed delays or explicit successor tracking.

Thank you for listening