yugabyteDB

PostgreSQL Compatibility

PostgreSQL Compatibility

Read Committed Isolation

yugabyteDB

PostgreSQL Compatibility

Read Committed Isolation & Pessimistic Locking

Why compatibility?

Why compatibility?

Scaling your existing app should be straightforward

NO to "our db scales flawlessly & is fast but we don't support this yet, can your app workaround this?"

Levels of compatibility

Wire - does the db work with existing Postgres drivers, same byte format, serialization etc.

Syntax - would the same syntax as PostgreSQL work?

Feature - parity in terms of different functionalities: triggers, stored procedures, gin indexes, etc

Runtime - matching execution semantics. An app shouldn’t be able to say whether the db point is PostgreSQL or something else (barring any theoretical performance differences that stem from distribution)

Compatibility at the transactional layer is even more important because……

Compatibility at the transactional layer is even more important because……

ACID transactions -> tricky

Compatibility at the transactional layer is even more important because……

ACID transactions -> tricky

Distributed ACID transactions -> trickier

Fault tolerance & always available

Clock skew

Sharding, automatic shard splitting

Load balancing

Horizontal scalability

and more ….

Compatibility at the transactional layer is even more important because……

ACID transactions -> tricky

Distributed ACID transactions -> trickier

=> Impossible for the app to workaround incompatibility at the transactional layer

Fault tolerance & always available

Clock skew

Sharding, automatic shard splitting

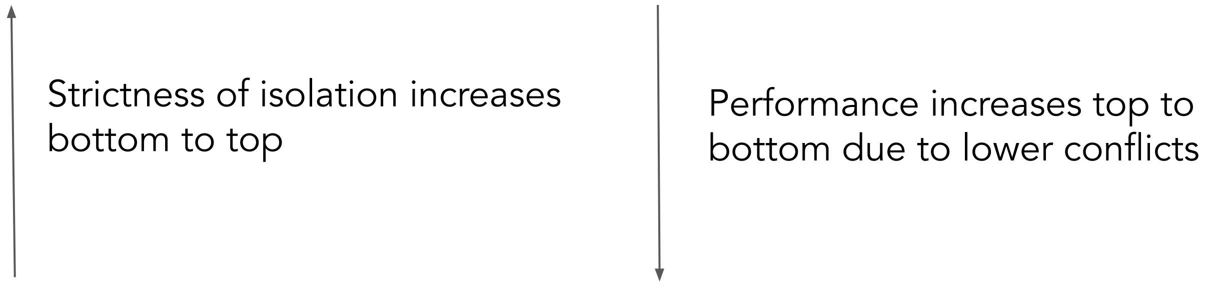
Load balancing

Horizontal scalability

and more ….

Isolation Levels in PostgreSQL

1. Serializable (supported by YB since long)
2. Repeatable Read (supported by YB since long)
3. Read Committed (new addition to YB!)
4. Read Uncommitted (same as Read Committed)



Strictness of isolation increases bottom to top

Performance increases top to bottom due to lower conflicts

Read Committed Isolation Level

Read Committed Isolation Level

Key ideas that define “read committed” -

1. From Postgres -
   a. New snapshot per statement in the transaction
   b. Apps never face serialization errors (40001), so don’t have to retry those
2. Read restart errors no more [stems from **clock-skew** due to YugabyteDB’s distributed nature]

Read Committed Isolation Level

Key ideas that define “read committed” -

1. From Postgres -
   a. New snapshot per statement in the transaction
   b. Apps never face serialization errors, so don’t have to retry those

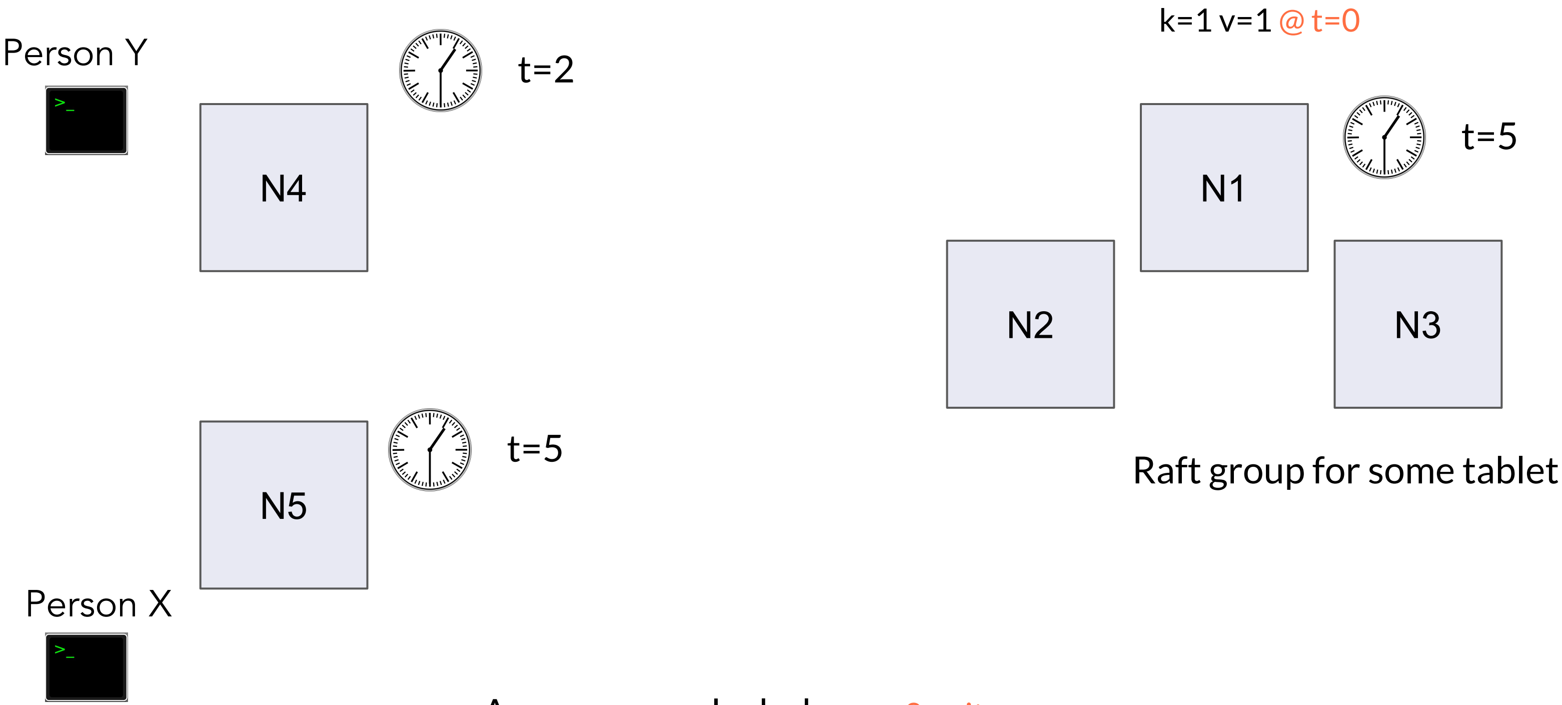
[Insert live terminal demo for point (1 & 2) here]

Read Committed Isolation Level

Key ideas that define “read committed” -

1. From Postgres -
   a. New snapshot per statement in the transaction
   b. Apps never face serialization errors, so don’t have to retry those
2. Read restart errors no more [specific to Yugabyte due to its distributed nature]

a 2 min detour to declutter read restarts …



Person Y

t=2

N4

k=1 v=1 @ t=0

N1

t=5

N2

N3

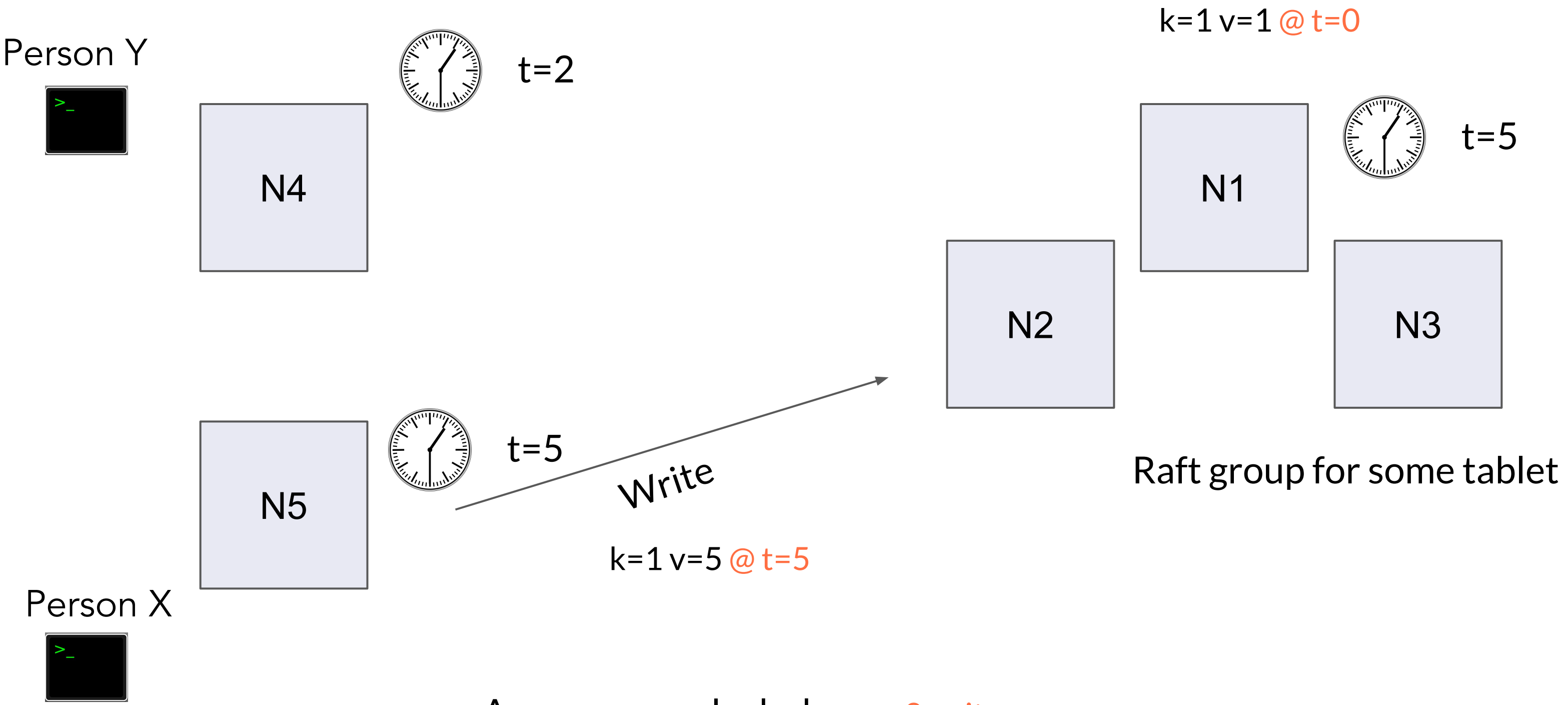
N5

t=5

Raft group for some tablet

Person X

Assume max clock skew = 8 units



Person Y

t=2

N4

k=1 v=1 @ t=0

N1

t=5

N2

N3

Raft group for some tablet

N5

t=5

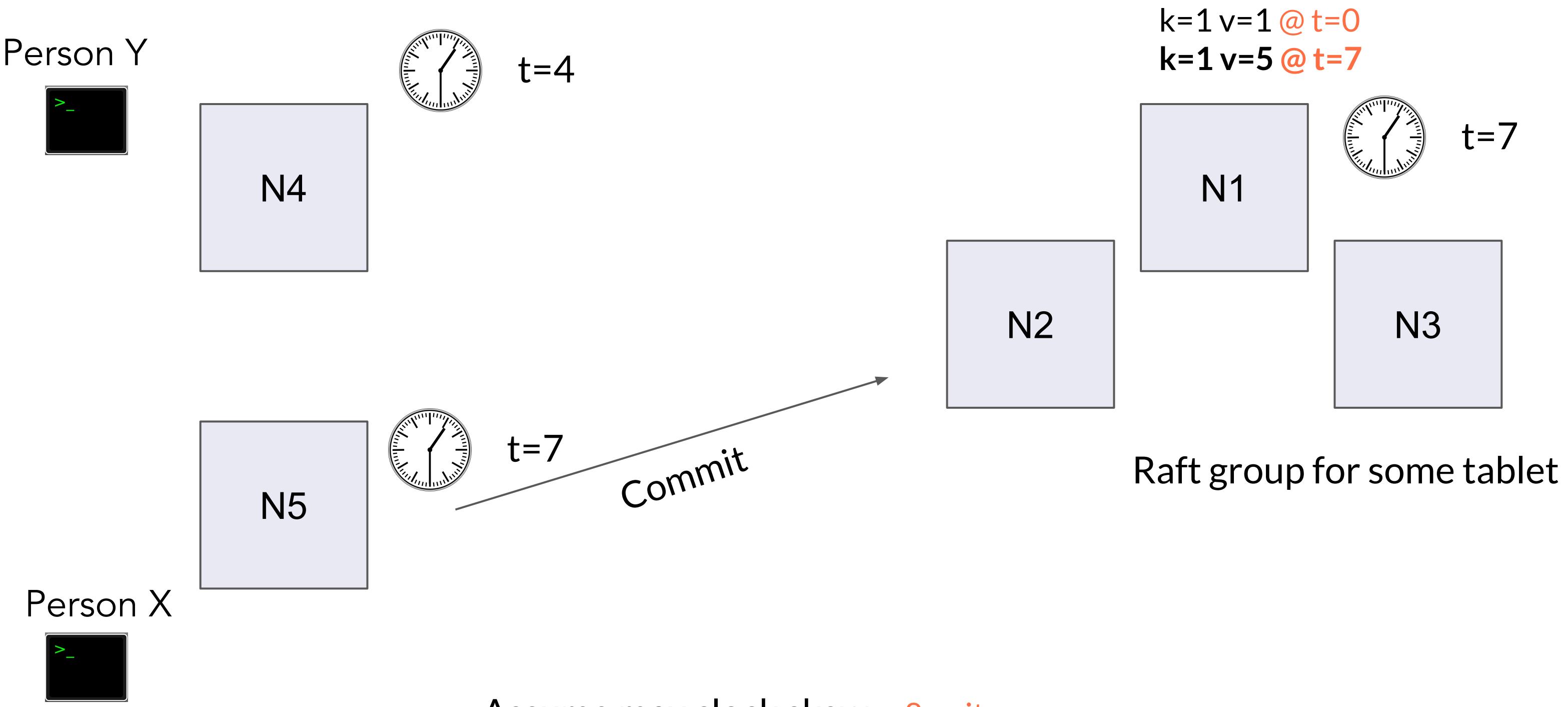
Write

k=1 v=5 @ t=5

Person X

Assume max clock skew = 8 units

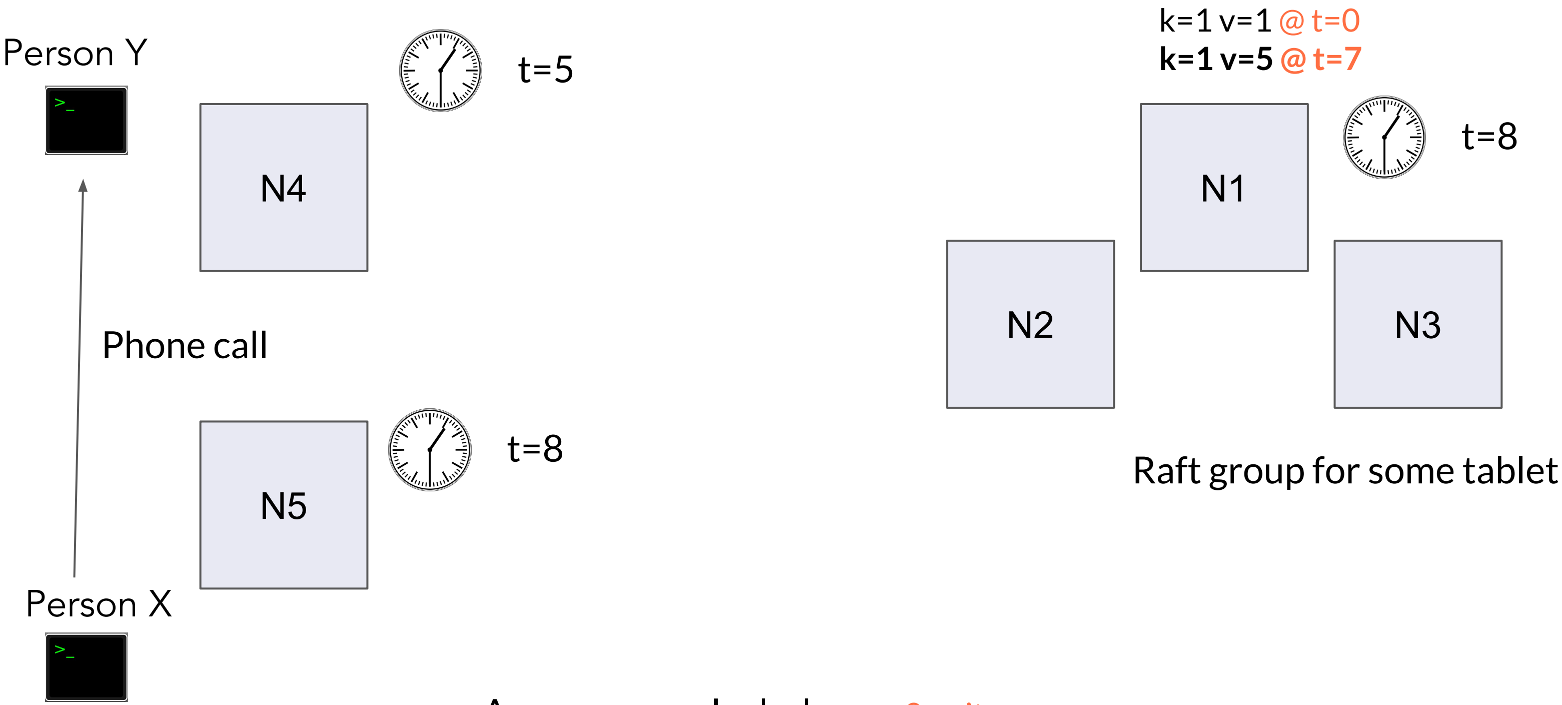
2 times units later .....



Raft group for some tablet

Assume max clock skew = 8 units

1 time unit later …..



Person Y

Person X

Phone call

N4

t=5

N5

t=8

k=1 v=1 @ t=0

k=1 v=5 @ t=7

N1

t=8

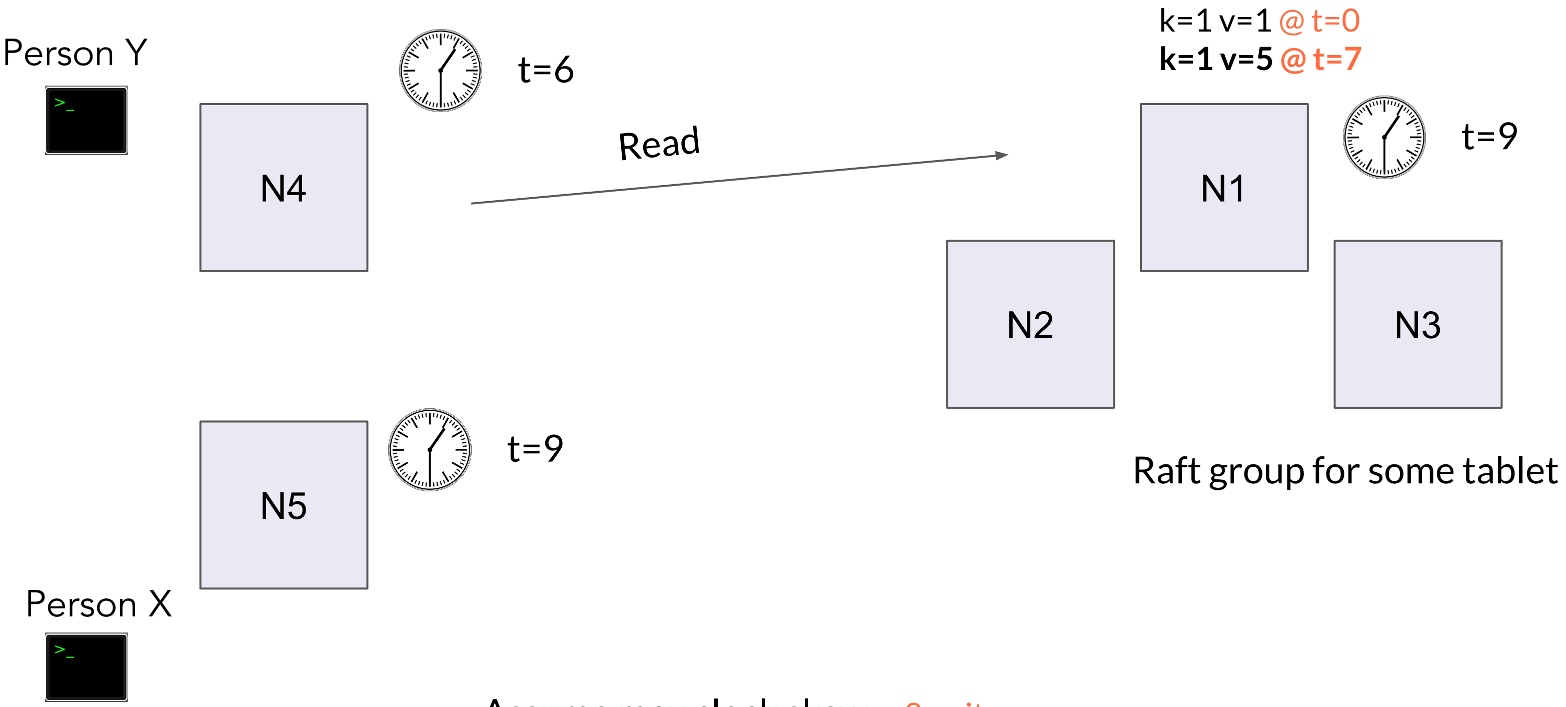
N2

N3

Raft group for some tablet

Assume max clock skew = 8 units

1 time unit later .....



Person Y

t=6

Read

k=1 v=1 @ t=0

k=1 v=5 @ t=7

t=9

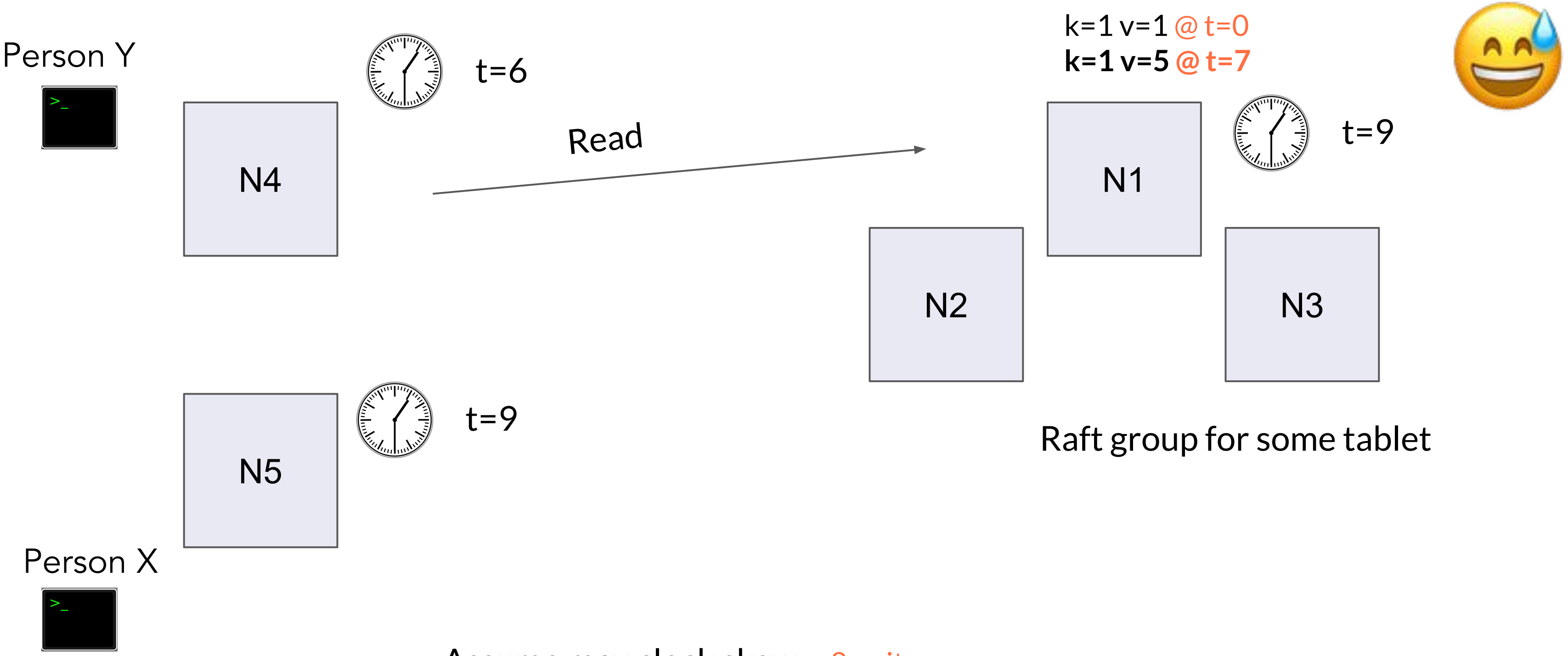
t=9

Raft group for some tablet

Person X

Assume max clock skew = 8 units

1 time unit later .....

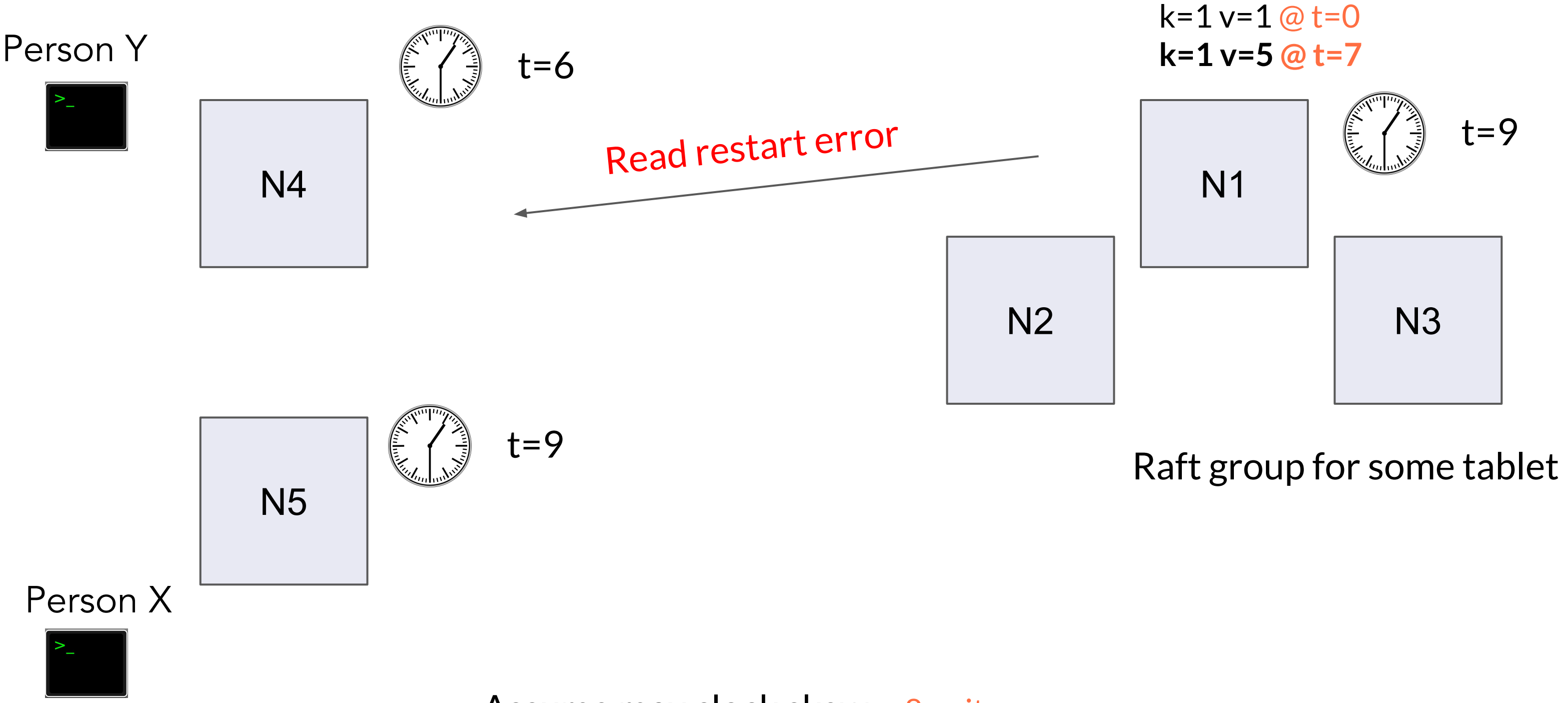


k=1 v=1 @ t=0
k=1 v=5 @ t=7

Raft group for some tablet

Assume max clock skew = 8 units

1 time unit later …..



Raft group for some tablet

Assume max clock skew = 8 units

[Insert live terminal demo for point (3) i.e., read restarts here]

The Temenos High Water Benchmark In (Big) Numbers [SKIP slide]

temenos

3000
Global Banking Customers

41/50
Of The Top Global Banks

1.2 Bn
Global Bank Customers

Investing
20%
Revenue in R&D

High Water Benchmark

102K
Business Transactions Per Second

100M
Customers
200M
Accounts

4.1x
More Efficient For A Smaller CO_2 Footprint

+40%
Better Performance

yugabyteDB

350K
Database Reads Per Second

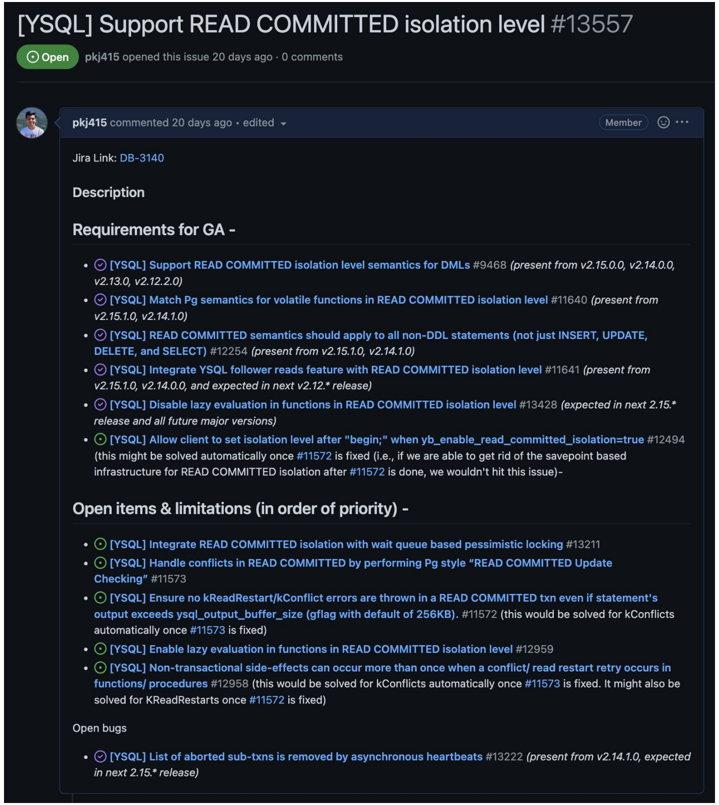
80K
Database Writes Per Second

Inserts **3** ms
Selects **1** ms
Deletes **1** ms

39/3
DB Nodes AWS AZ

Roadmap

Track further enhancements and their progress at https://github.com/yugabyte/yugabyte-db/issues/13557



[YSQL] Support READ COMMITTED isolation level #13557

pkj415 opened this issue 20 days ago · 0 comments

pkj415 commented 20 days ago • edited

Jira Link: DB-3140

Description

Requirements for GA -

- ✓ [YSQL] Support READ COMMITTED isolation level semantics for DMLs #9468 *(present from v2.15.0.0, v2.14.0.0, v2.13.0, v2.12.2.0)*
- ✓ [YSQL] Match Pg semantics for volatile functions in READ COMMITTED isolation level #11640 *(present from v2.15.1.0, v2.14.1.0)*
- ✓ [YSQL] READ COMMITTED semantics should apply to all non-DDL statements (not just INSERT, UPDATE, DELETE, and SELECT) #12254 *(present from v2.15.1.0, v2.14.1.0)*
- ✓ [YSQL] Integrate YSQL follower reads feature with READ COMMITTED isolation level #11641 *(present from v2.15.1.0, v2.14.0.0, and expected in next v2.12.* release)*
- ✓ [YSQL] Disable lazy evaluation in functions in READ COMMITTED isolation level #13428 *(expected in next 2.15.* release and all future major versions)*
- ⊙ [YSQL] Allow client to set isolation level after "begin;" when yb_enable_read_committed_isolation=true #12494 (this might be solved automatically once #11572 is fixed (i.e., if we are able to get rid of the savepoint based infrastructure for READ COMMITTED isolation after #11572 is done, we wouldn't hit this issue)-

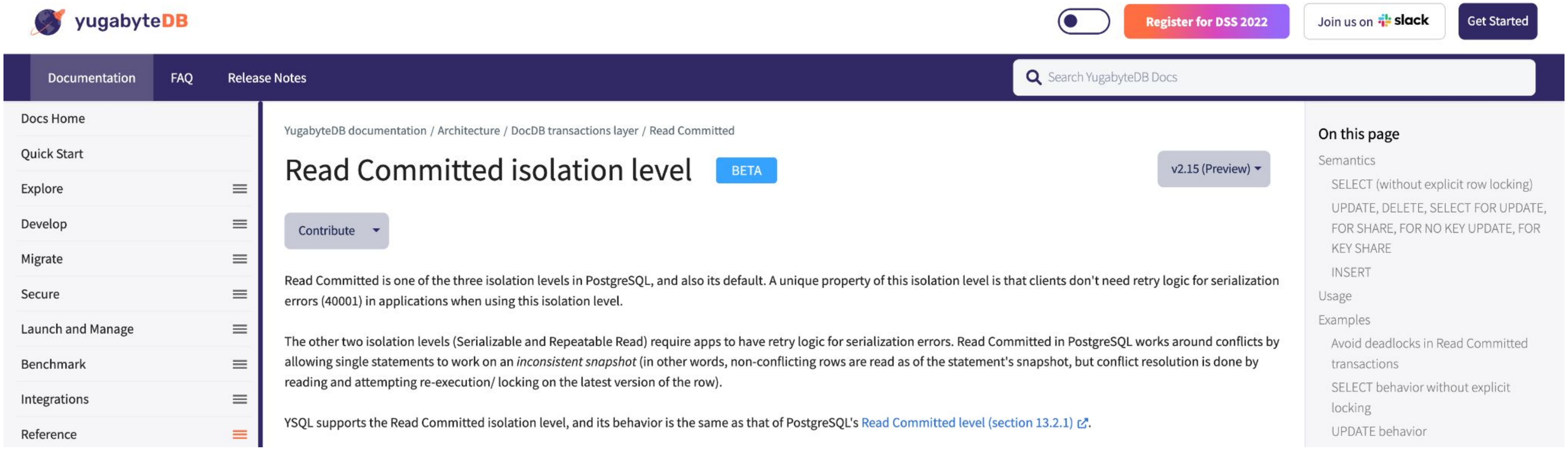
Open items & limitations (in order of priority) -

- ⊙ [YSQL] Integrate READ COMMITTED isolation with wait queue based pessimistic locking #13211
- ⊙ [YSQL] Handle conflicts in READ COMMITTED by performing Pg style “READ COMMITTED Update Checking” #11573
- ⊙ [YSQL] Ensure no kReadRestart/kConflict errors are thrown in a READ COMMITTED txn even if statement's output exceeds ysql_output_buffer_size (gflag with default of 256KB). #11572 (this would be solved for kConflicts automatically once #11573 is fixed)
- ⊙ [YSQL] Enable lazy evaluation in functions in READ COMMITTED isolation level #12959
- ⊙ [YSQL] Non-transactional side-effects can occur more than once when a conflict/ read restart retry occurs in functions/ procedures #12958 (this would be solved for kConflicts automatically once #11573 is fixed. It might also be solved for KReadRestarts once #11572 is fixed)

Open bugs

- ✓ [YSQL] List of aborted sub-txns is removed by asynchronous heartbeats #13222 *(present from v2.14.1.0, expected in next 2.15.* release)*

View the docs



YugabyteDB documentation / Architecture / DocDB transactions layer / Read Committed

Read Committed isolation level

Read Committed is one of the three isolation levels in PostgreSQL, and also its default. A unique property of this isolation level is that clients don't need retry logic for serialization errors (40001) in applications when using this isolation level.

The other two isolation levels (Serializable and Repeatable Read) require apps to have retry logic for serialization errors. Read Committed in PostgreSQL works around conflicts by allowing single statements to work on an *inconsistent snapshot* (in other words, non-conflicting rows are read as of the statement's snapshot, but conflict resolution is done by reading and attempting re-execution/ locking on the latest version of the row).

YSQL supports the Read Committed isolation level, and its behavior is the same as that of PostgreSQL's Read Committed level (section 13.2.1).

On this page

- Semantics
  - SELECT (without explicit row locking)
  - UPDATE, DELETE, SELECT FOR UPDATE, FOR SHARE, FOR NO KEY UPDATE, FOR KEY SHARE
  - INSERT
- Usage
- Examples
  - Avoid deadlocks in Read Committed transactions
  - SELECT behavior without explicit locking
  - UPDATE behavior

https://docs.yugabyte.com/preview/architecture/transactions/read-committed/

Pessimistic locking

Optimistic locking

On conflict, roll-back one of the conflicting txns based on priority

Deadlocks are avoided since transactions never wait

Aborts transactions unnecessarily in contentious workloads

Behavior is incompatible with PostgreSQL

Pessimistic locking

On conflict, wait for the blocking transaction to commit or rollback

Deadlocks are detected among waiting transactions

Aborts transactions minimally in contentious workloads

Behavior is compatible with PostgreSQL

Along with pessimistic locking, comes the problem of detecting distributed deadlocks.

Deadlock Detection

Detection happens quickly with detector triggering 1s after conflict

Deadlock Detection

Detection happens quickly with detector triggering 1s after conflict

Detection is guaranteed once a deadlock is created

Deadlock Detection

Edge Chasing Algorithm

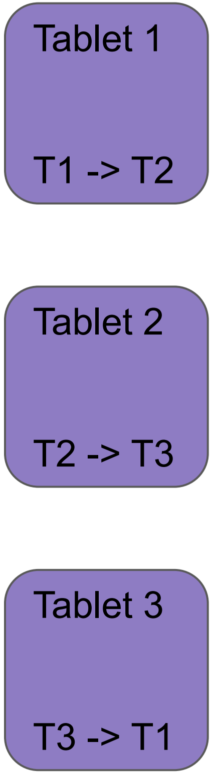
Chandy and Misra, 1982 - *A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems*

https://www.cs.utexas.edu/users/misra/scannedPdf.dir/ResourceDeadlock.pdf

Wait-for graph is formed by considering transactions as nodes and waiting-status as a directed edge

Probes are sent between transaction coordinators to detect cycles in this graph

Deadlock Detection



Tablet 1

T1 -> T2

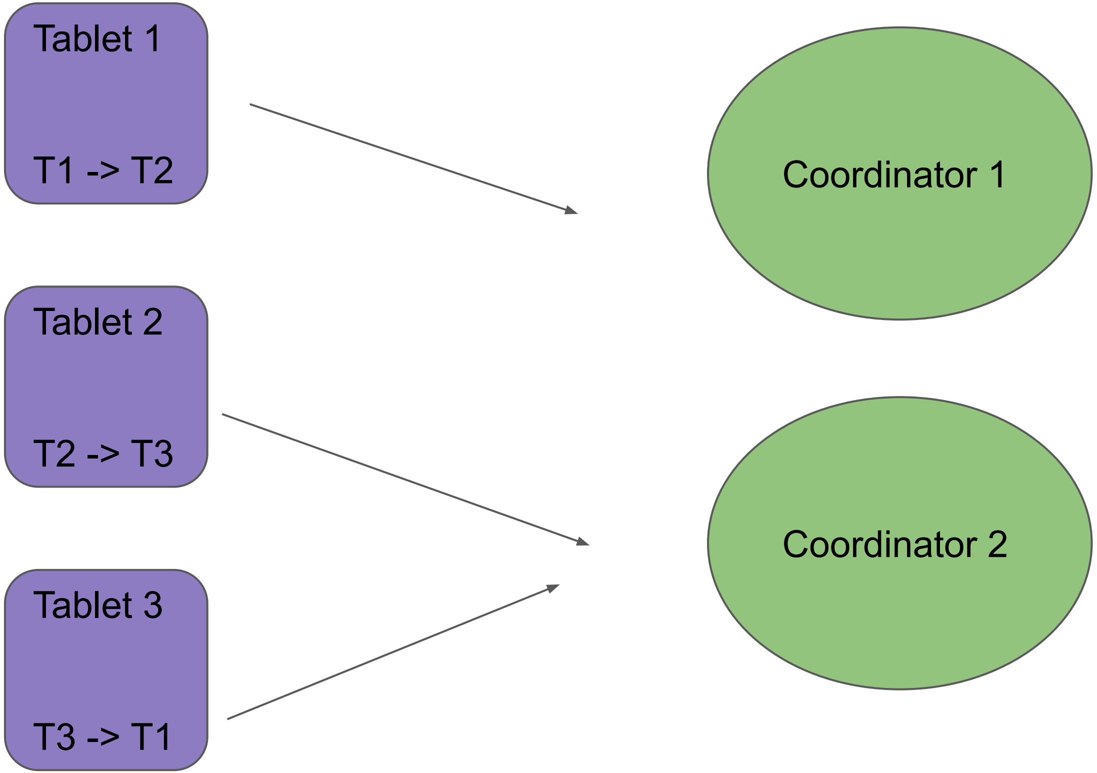
Tablet 2

T2 -> T3

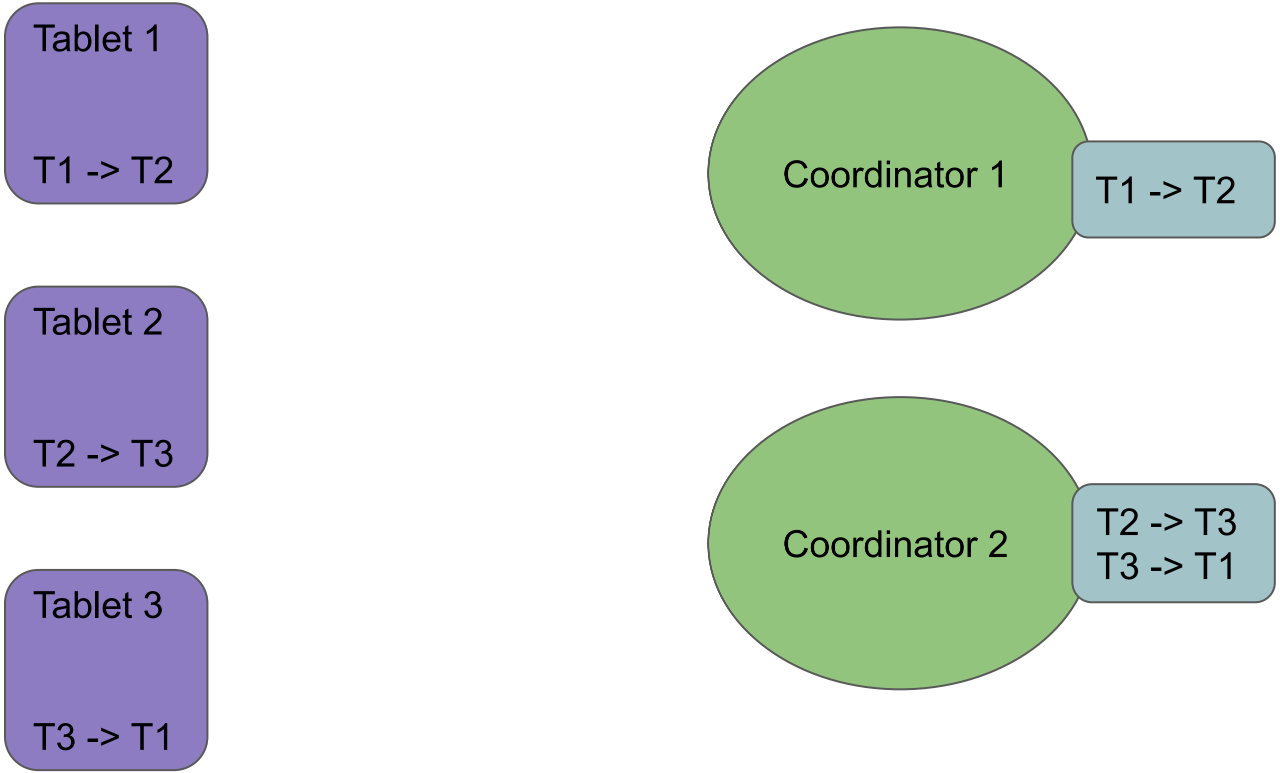
Tablet 3

T3 -> T1

Deadlock Detection



Deadlock Detection



Tablet 1

T1 -> T2

Tablet 2

T2 -> T3

Tablet 3

T3 -> T1

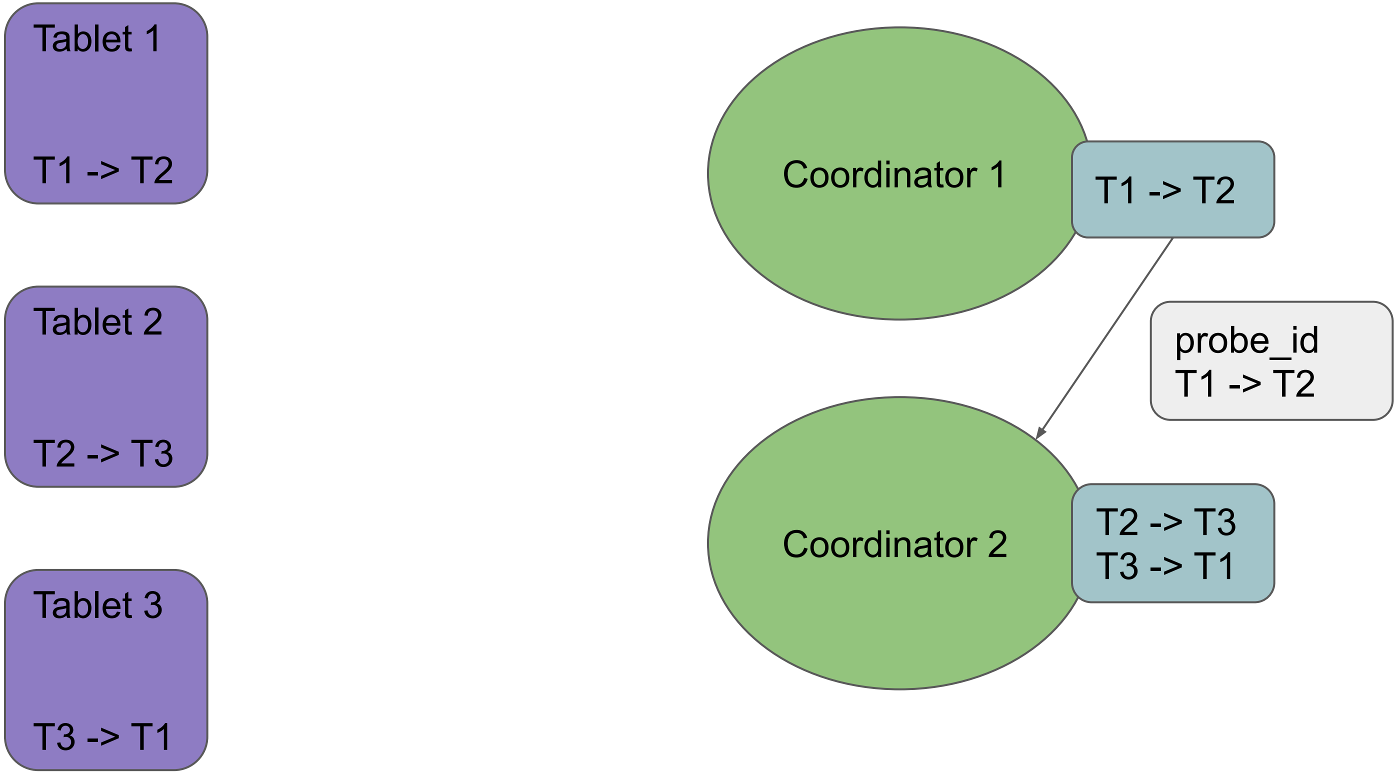
Coordinator 1

T1 -> T2

Coordinator 2

T2 -> T3
T3 -> T1

Deadlock Detection



Tablet 1

T1 -> T2

Tablet 2

T2 -> T3

Tablet 3

T3 -> T1

Coordinator 1

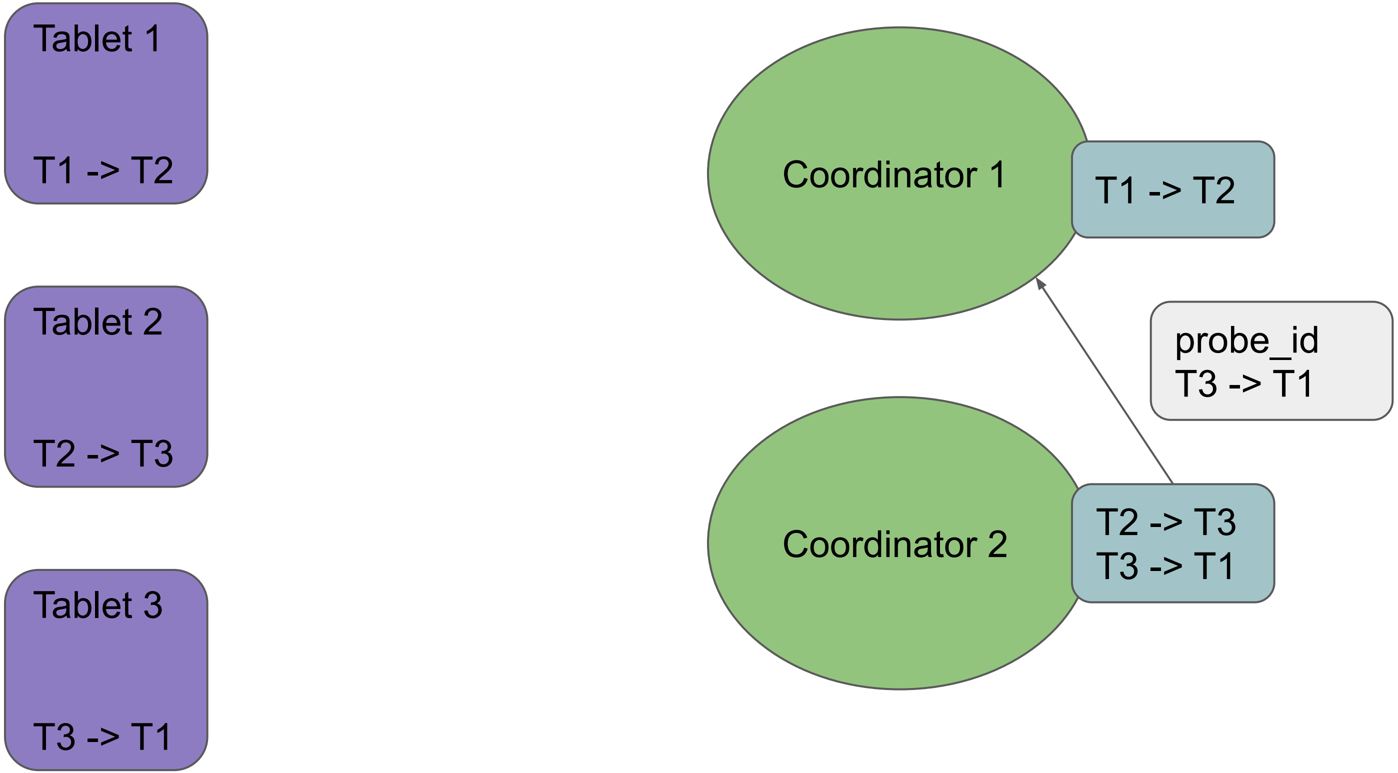
T1 -> T2

probe_id
T1 -> T2

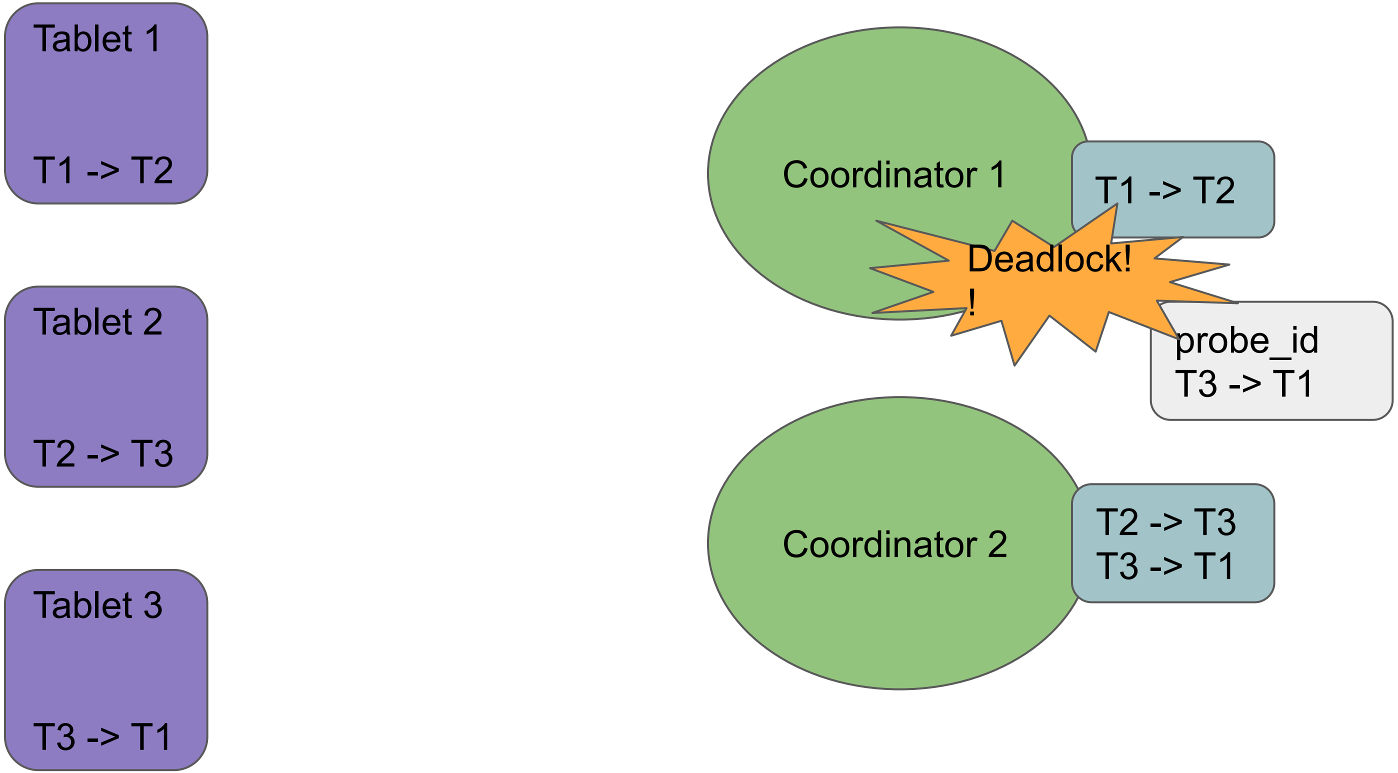
Coordinator 2

T2 -> T3
T3 -> T1

Deadlock Detection



Deadlock Detection



Deadlock Detection

Detection happens quickly with detector triggering 1s after conflict

Detection is guaranteed once a deadlock is created

Overhead is optimal requiring constant additional memory and only one probe per conflicting transaction. Probes could trigger many RPCs depending on wait-for graph

Additional Resources

View our docs page for Read Committed isolation level

View our spec document for pessimistic locking

See our roadmap for this area, more exciting Items coming your way!

Thank You

Join us on Slack: yugabyte.com/slack

Star us on Github: github.com/yugabyte/yugabyte-db