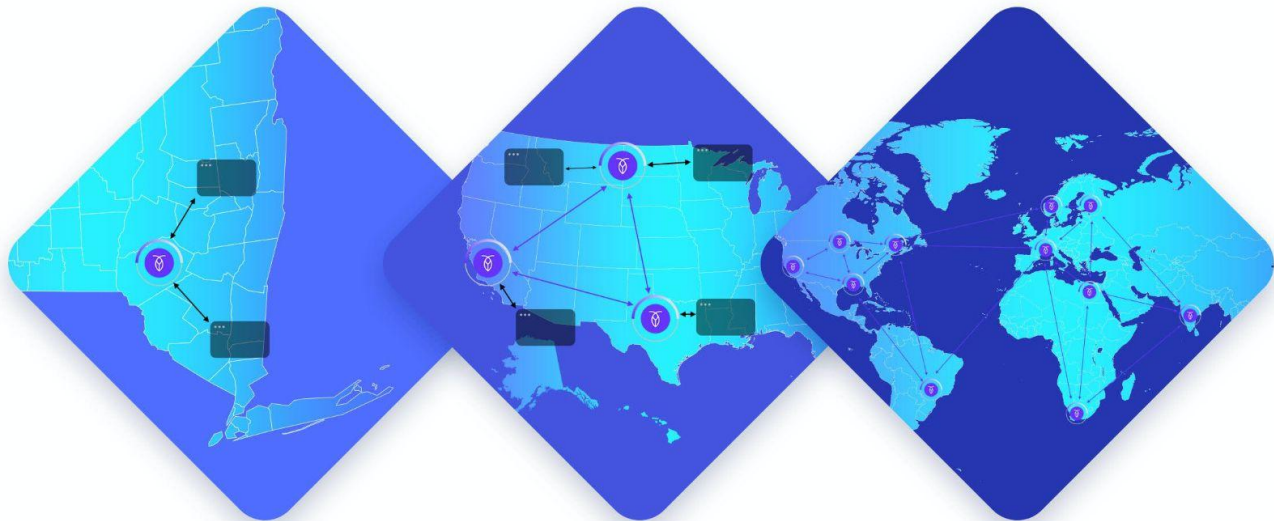




# Fearless Global Transactions with CockroachDB

Nathan VanBenschoten (@natevanben)



# Challenge

# Challenge

```
> INSERT INTO Orders VALUES (gen_random_uuid(), 123, 3, 789)
```

# Challenge

```
> INSERT INTO Orders VALUES (gen_random_uuid(), 123, 3, 789)
```

Products

Product		
PK	ProductID	UUID
	Name	STRING(256)
	Price	FLOAT64

Orders

Order		
PK	OrderID	UUID
FK	ProductID	UUID
	Quantity	INT64
FK	CustomerID	UUID

Customers

Customer		
PK	CustomerID	UUID
	FirstName	STRING(256)
	LastName	STRING(256)

# Challenge

```
> INSERT INTO Orders VALUES (gen_random_uuid(), 123, 3, 789)
```

Products

Product		
PK	ProductID	UUID
	Name	STRING(256)
	Price	FLOAT64

Orders

Order		
PK	OrderID	UUID
FK	ProductID	UUID
	Quantity	INT64
FK	CustomerID	UUID

Customers

Customer		
PK	CustomerID	UUID
	FirstName	STRING(256)
	LastName	STRING(256)

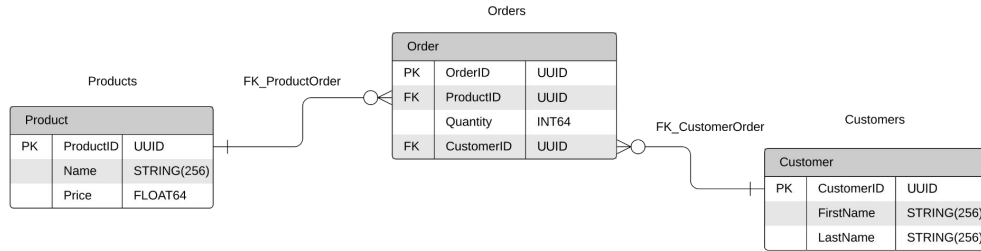
## REQUIREMENTS

### Consistency

Referential integrity across tables

# Challenge

```
> INSERT INTO Orders VALUES (gen_random_uuid(), 123, 3, 789)
```



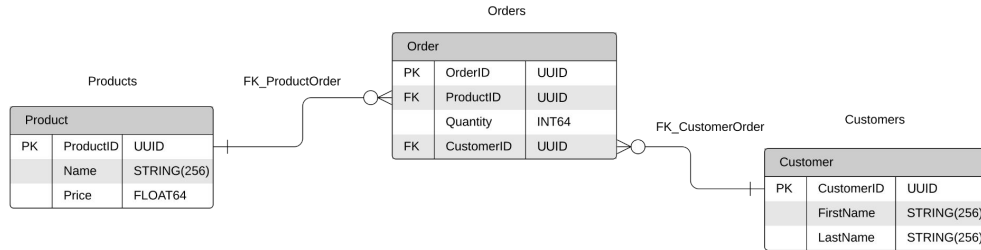
## REQUIREMENTS

### Consistency

Referential integrity across tables

# Challenge

```
> INSERT INTO Orders VALUES (gen_random_uuid(), 123, 3, 789)
```



## REQUIREMENTS

### Consistency

Referential integrity across tables

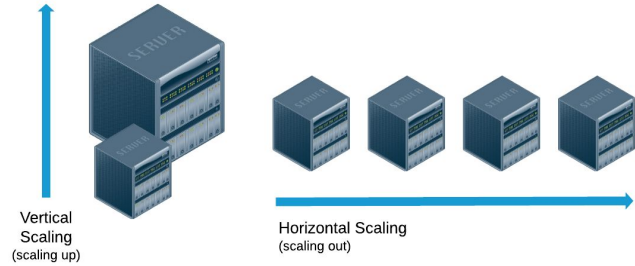
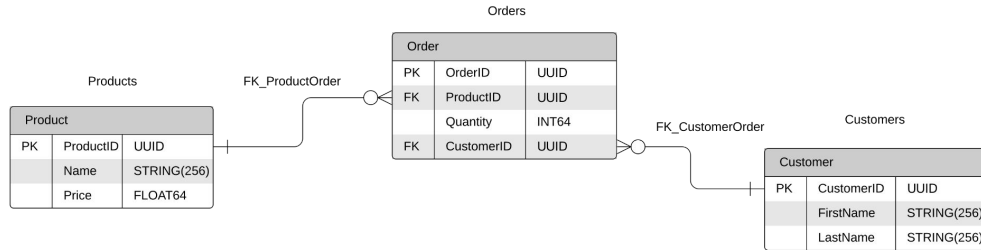
### Scalability

100k+ orders per second



# Challenge

```
> INSERT INTO Orders VALUES (gen_random_uuid(), 123, 3, 789)
```



## REQUIREMENTS

### Consistency

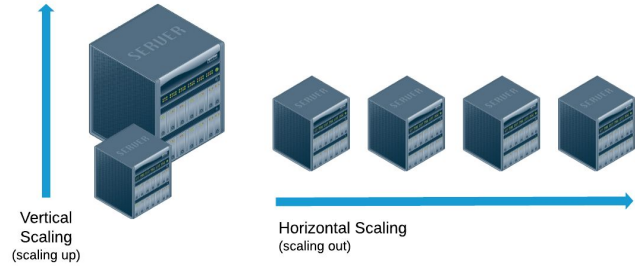
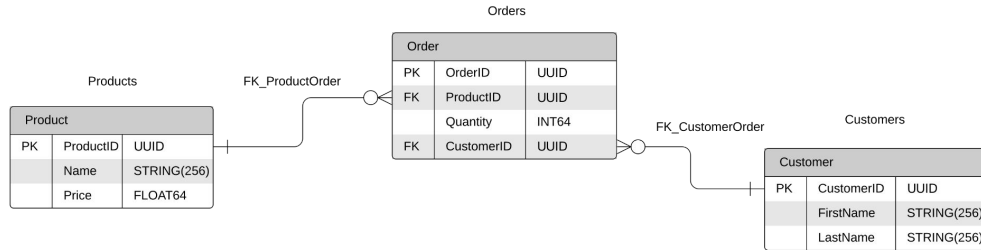
Referential integrity across tables

### Scalability

100k+ orders per second

# Challenge

```
> INSERT INTO Orders VALUES (gen_random_uuid(), 123, 3, 789)
```



## REQUIREMENTS

### Consistency

Referential integrity across tables

### Scalability

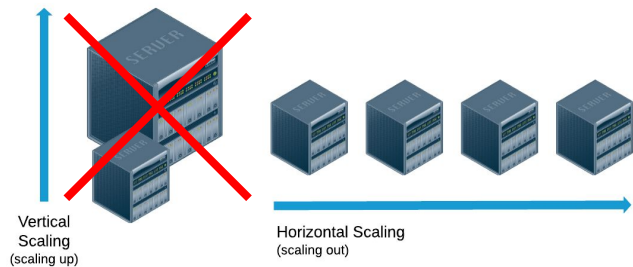
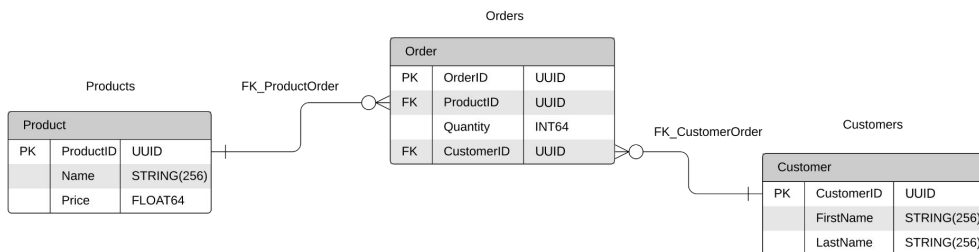
100k+ orders per second

### High availability

Survive node/zone/region failure

# Challenge

```
> INSERT INTO Orders VALUES (gen_random_uuid(), 123, 3, 789)
```



## REQUIREMENTS

### Consistency

Referential integrity across tables

### Scalability

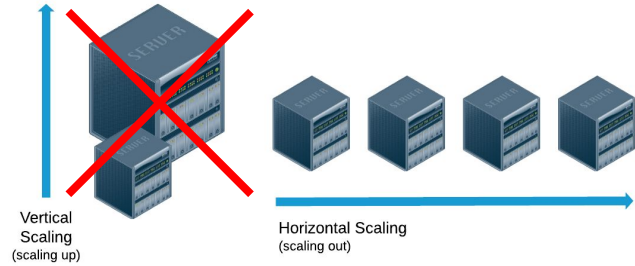
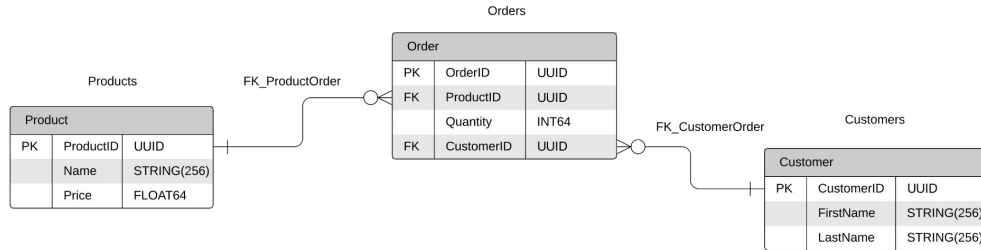
100k+ orders per second

### High availability

Survive node/zone/region failure

# Challenge

```
> INSERT INTO Orders VALUES (gen_random_uuid(), 123, 3, 789)
```



## REQUIREMENTS

### Consistency

Referential integrity across tables

### Scalability

100k+ orders per second

### High availability

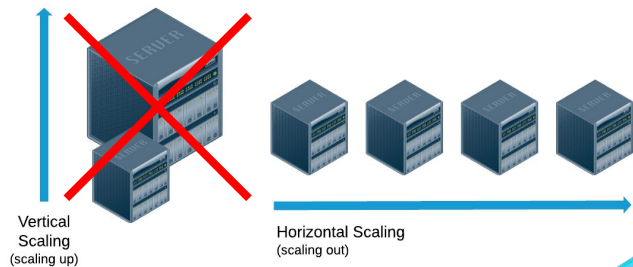
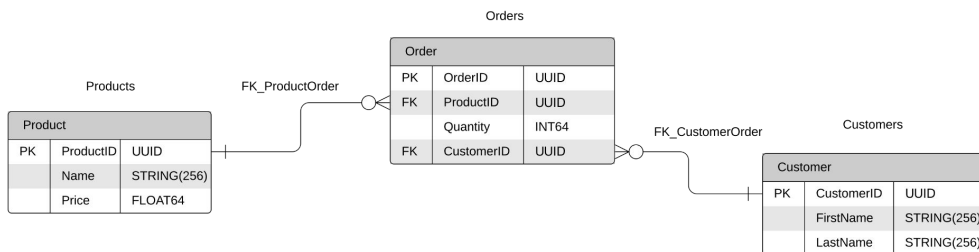
Survive node/zone/region failure

### Low latency

Sub 20ms end-to-end

# Challenge

```
> INSERT INTO Orders VALUES (gen_random_uuid(), 123, 3, 789)
```



## REQUIREMENTS

### Consistency

Referential integrity across tables

### Scalability

100k+ orders per second

### High availability

Survive node/zone/region failure

### Low latency

Sub 20ms end-to-end

# NewSQL

What's changed?

## SQL REQUIREMENTS

ACID

Secondary indexes

Foreign key constraints

Joins

ORDER BY / LIMIT



## SYSTEM REQUIREMENTS

Strong consistency

Strong isolation

Cross-partition transactions

Range partitioning

# NewSQL

What's changed?

## SYSTEM REQUIREMENTS

Strong consistency  
Strong isolation  
Cross-partition transactions  
Range partitioning



## SYSTEM IMPLEMENTATION

Leader-based consensus protocols  
Synchronous replication  
Partitioned consensus + clock sync  
Serializable isolation + MVCC  
Stale reads, not inconsistent reads

# NewSQL

## What's changed?

### SYSTEM IMPLEMENTATION

Leader-based consensus protocols

Synchronous replication

Partitioned consensus + clock sync

Serializable isolation + MVCC

Stale reads, not inconsistent reads



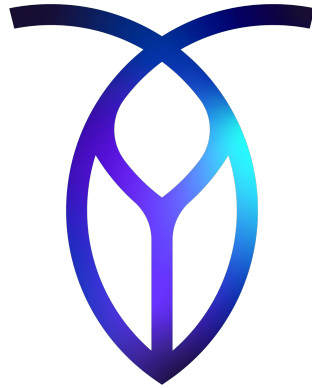
### GLOBAL CHALLENGES

Data needs a leader, only leader can write

High consensus latency

Global serializable transaction history





**CockroachDB**



**IN LATEST RELEASE — VERSION 21.1**

First-class region management

Goal-oriented data placement policies

Non-voting replicas

Implicit table partitioning

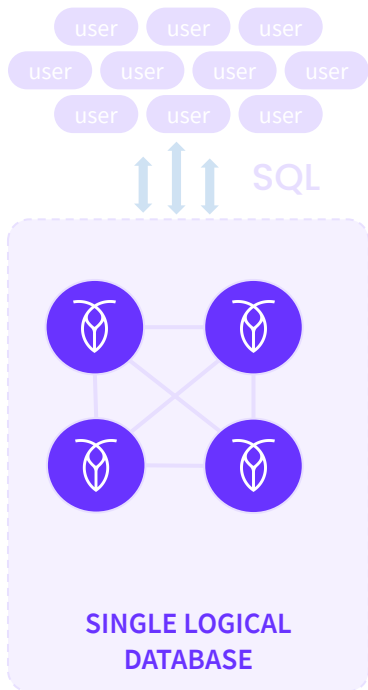
Auto row-level data homing

Locality aware cost-based SQL optimizations

Non-blocking extension to transaction model

# CockroachDB: Scalability

Elastic & efficient scale for applications with a relational database



## Scalability

CockroachDB is a distributed, **relational database** that can be used for the most straightforward, common and high value workloads and gives your developers, **familiar standard SQL**

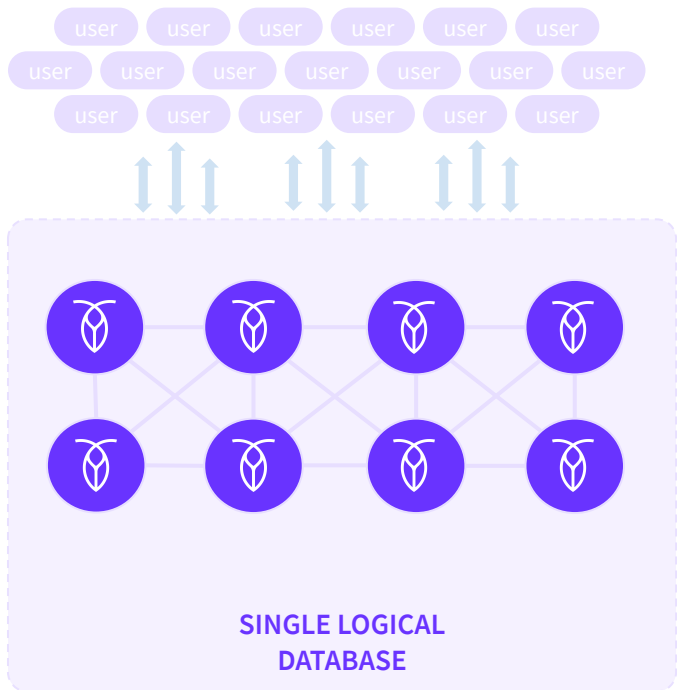
It is a database cluster that is comprised of nodes that appear as a single logical database

## Survivability

## Localization

# CockroachDB: Scalability

Elastic & efficient scale for applications with a relational database



## Scalability

**Scale** the database by simply adding more nodes

CockroachDB auto-balances to incorporate the new resource. No manual work is required

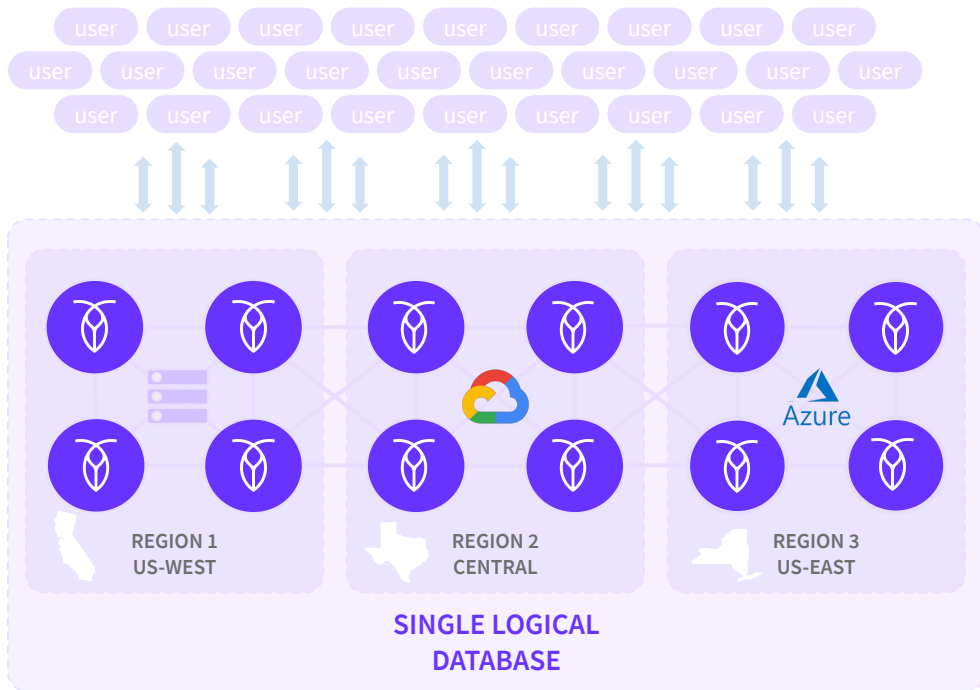
- Easy scale for increase in volume of data in the database
- Every node accepts reads & writes so you also scale transactional volume (writes)

## Survivability

## Localization

# CockroachDB: Scalability

Elastic & efficient scale for applications with a relational database



## Scalability

**Scale even further** across regions and even clouds, yet still deliver a single logical database

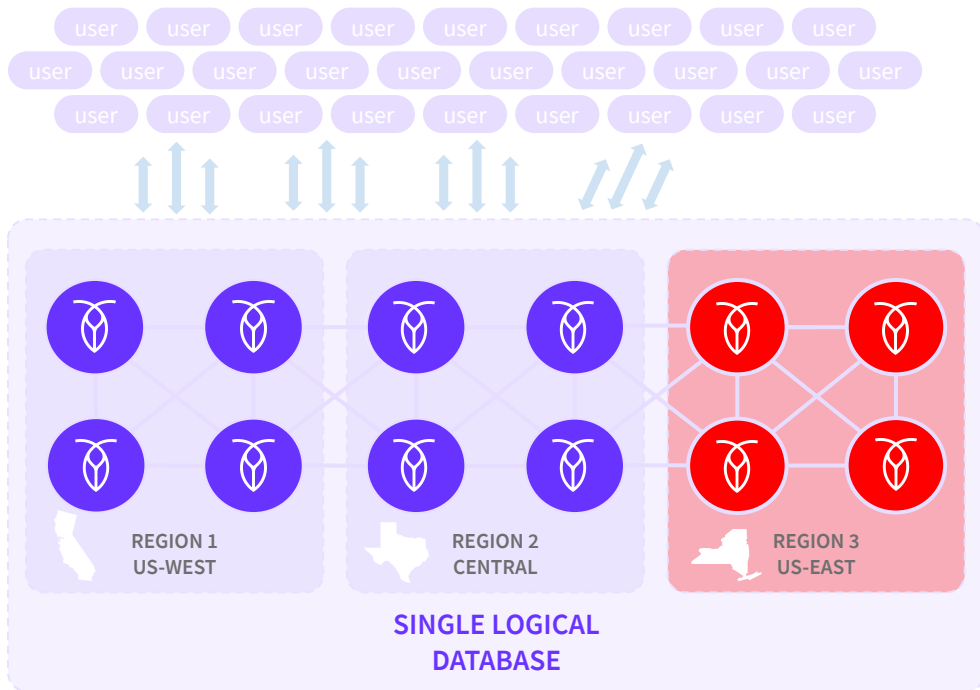
CockroachDB excels when deployed across multiple data centers in multiple regions

## Survivability

## Localization

# CockroachDB: Survivability

A database that is always on & build to survive failures



## Scalability

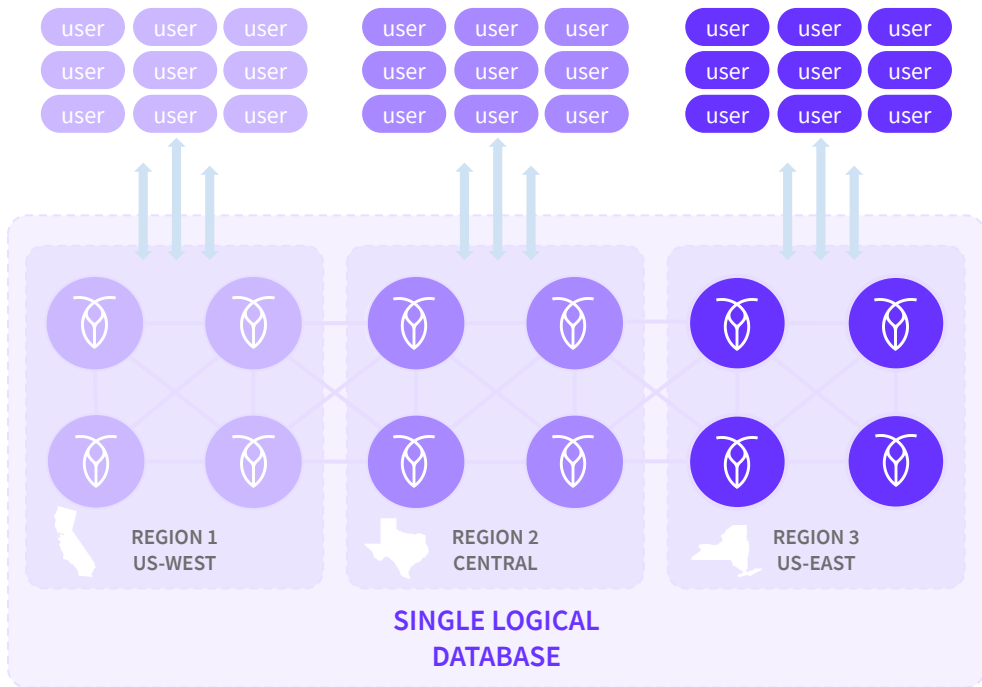
## Survivability

CockroachDB is naturally **resilient** so you can survive failure of a node or even an entire region without service disruption

- Always-on and with zero RPO
- Allows for no downtime rolling upgrades
- Online schema changes

# CockroachDB: Localization

Low-latency access even across broadly dispersed clients



Scalability

Survivability

## Localization

CockroachDB allows you to tie each row to a physical location based on data within each record

- reduce read/write latencies
- comply with regulations
- ensure customer data privacy

# Schema-Level Primitives

SQL tools for global databases

**Database Regions**

**Survival Goals**

**Table Locality**



# Schema-Level Primitives

SQL tools for global databases

**Database Regions**

Survival Goals

Table Locality

# Database Regions

Which regions does a database live in?

```
> ALTER DATABASE <db> ADD REGION "europe-west1"
```

```
> SHOW REGIONS FROM DATABASE <db>
```

region	primary	zones
europe-west1	false	{b,c,d}
us-east1	false	{b,c,d}
us-west1	true	{a,b,c}

# Database Regions

Which regions does a database live in?

```
> SHOW REGIONS FROM CLUSTER
```

region	zones
-----+-----	
europa-west1	{b,c,d}
us-east1	{b,c,d}
us-west1	{a,b,c}

# Database Regions

Which regions does a database live in?

```
> SHOW ENUMS FROM <db>
```

name	values
crdb_internal_region	{europe-west1,us-east1,us-west1}

# Schema-Level Primitives

SQL tools for global databases

Database Regions

**Survival Goals**

Table Locality

# Region Survival

Cross-Region Consensus

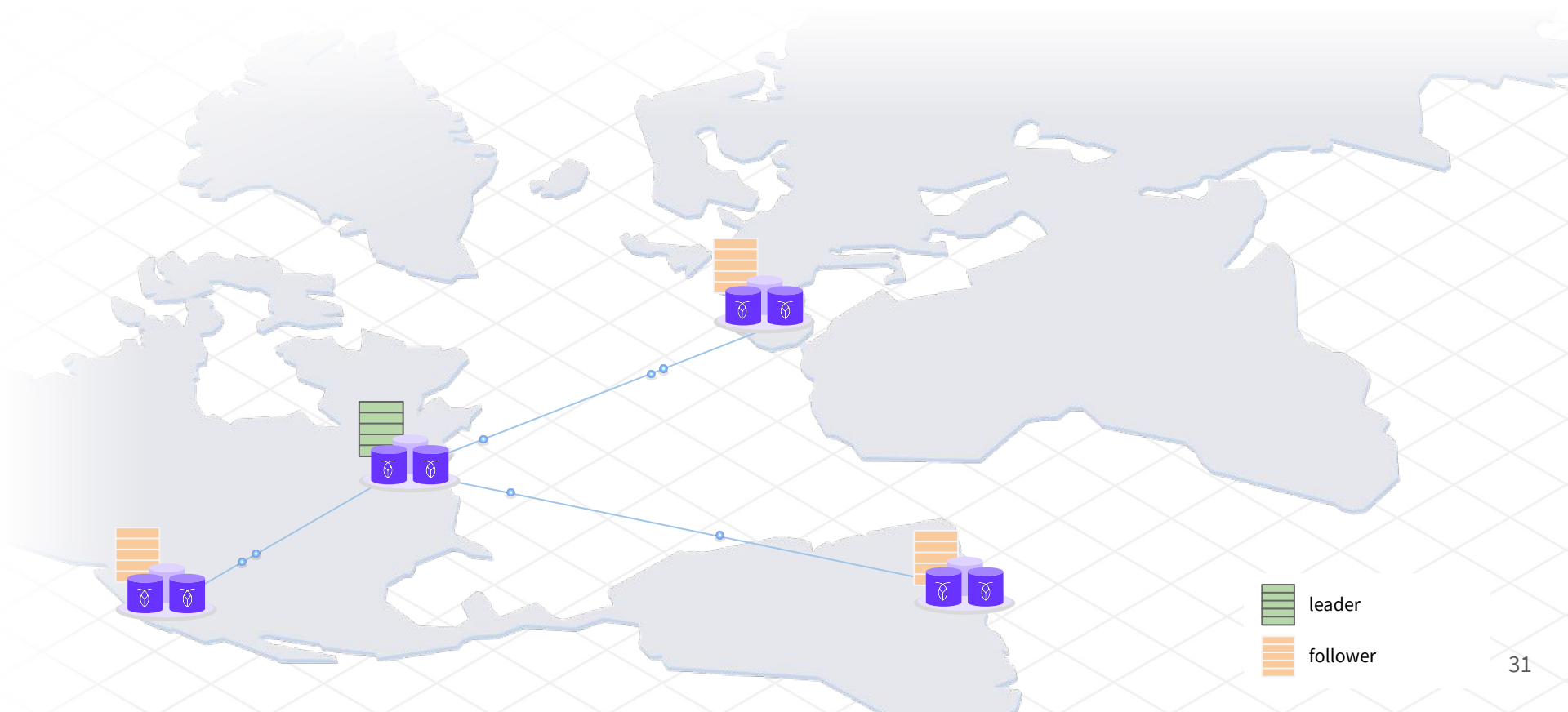
```
> ALTER DATABASE <db> SURVIVE REGION FAILURE
```



# Region Survival

Cross-Region Consensus

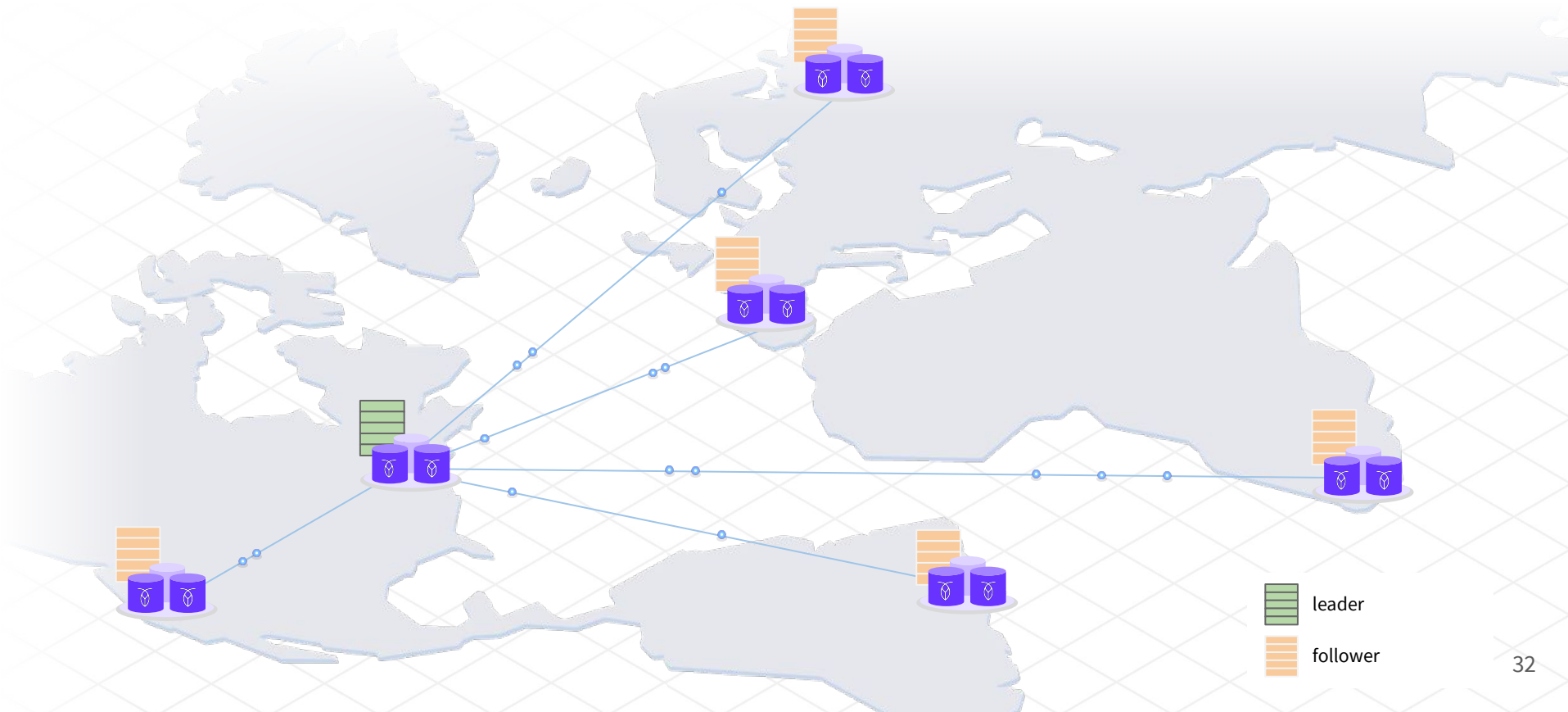
```
> ALTER DATABASE <db> SURVIVE REGION FAILURE
```



# Region Survival

## Cross-Region Consensus

```
> ALTER DATABASE <db> SURVIVE REGION FAILURE
```





# Region Survival

Cross-Region Consensus with Non-Voting Replicas

```
> ALTER DATABASE <db> SURVIVE REGION FAILURE
```



# Region Survival

Cross-Region Consensus with Non-Voting Replicas

```
> ALTER DATABASE <db> SURVIVE REGION FAILURE
```



# Zone Survival

Intra-Region Consensus with Non-Voting Replicas

```
> ALTER DATABASE <db> SURVIVE ZONE FAILURE
```



- leader
- voting follower
- non-voting follower

# Schema-Level Primitives

SQL tools for global databases

Database Regions

Survival Goals

**Table Locality**

# Table locality

Tune tables based on access locality

REGIONAL

GLOBAL

# REGIONAL tables

Meant for localized data

```
> ALTER DATABASE <db> SET PRIMARY REGION 'us-east1'
```

```
> ALTER TABLE <t> SET LOCALITY REGIONAL
```



# Table locality

What if I don't configure anything?

Default **survival goal**, default **table locality**

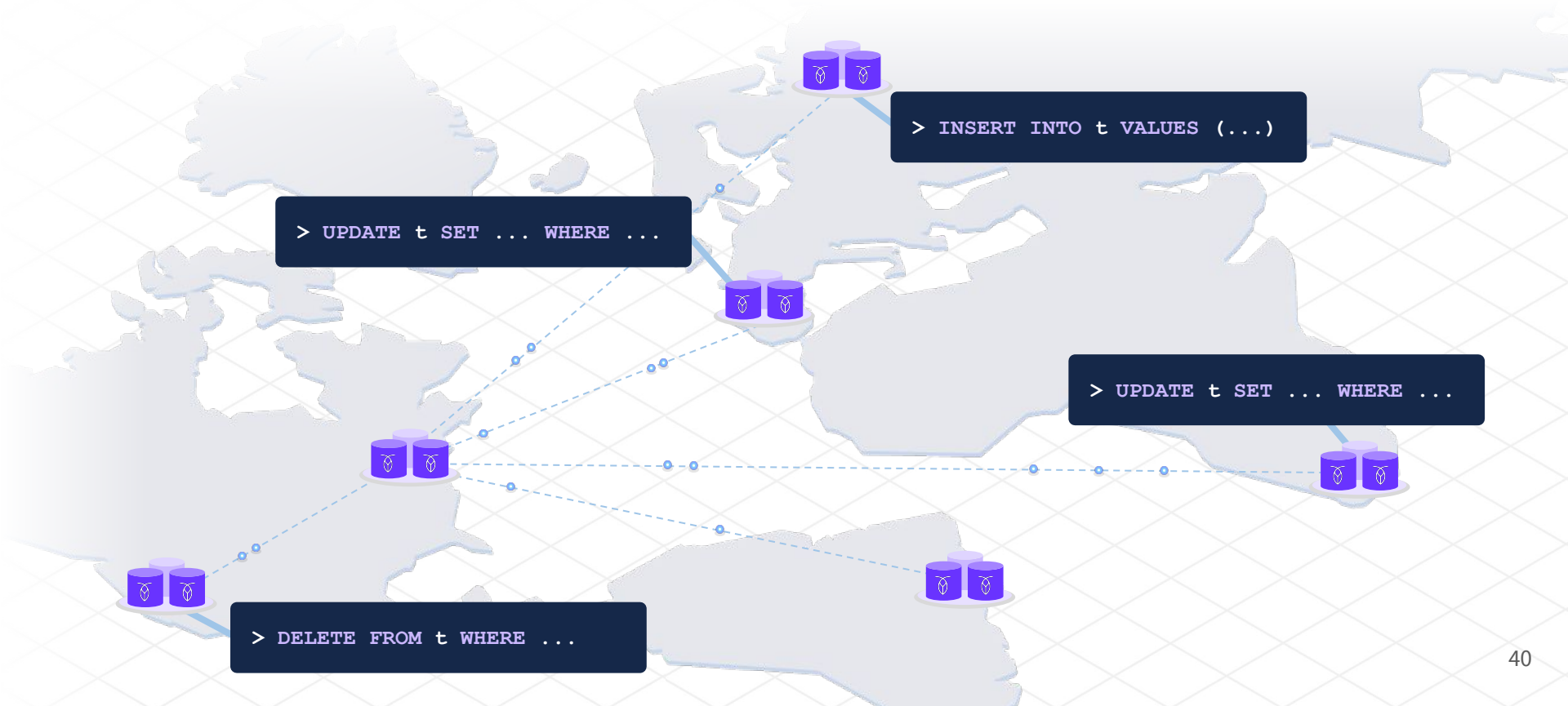
- **Failure tolerance:** Availability zone failover with *no data loss*  
Region failover with *some data loss* (in development)
- **Reads:** All regions
- **Writes:** Single region

Configured **REGION survival goal**, default **table locality**

- **Failure tolerance:** Region failover with *no data loss*
- **Reads:** All regions
- **Writes:** Single region, increased latency

# REGIONAL tables

Meant for localized data





# REGIONAL tables

Meant for localized data

```
> CREATE TABLE t (k INT PRIMARY KEY)
```

```
> SHOW TABLES
```

table_name	locality
t	REGIONAL BY TABLE IN PRIMARY REGION

```
> ALTER TABLE t SET LOCALITY REGIONAL IN "europe-west1"
```

```
> SHOW TABLES
```

table_name	locality
t	REGIONAL BY TABLE IN "europe-west1"

# REGIONAL tables

Meant for localized data

```
> ALTER TABLE <t> SET LOCALITY  
REGIONAL IN "europe-west1"
```



# REGIONAL tables

Meant for localized data

```
> ALTER TABLE users_na  
  SET LOCALITY REGIONAL IN "northamerica-west1"
```

```
> ALTER TABLE users_sa  
  SET LOCALITY REGIONAL IN "southamerica-east1"
```

```
> SHOW TABLES
```

table_name	locality
users_na	REGIONAL BY TABLE IN "northamerica-west1"
users_sa	REGIONAL BY TABLE IN "southamerica-east1"

# REGIONAL tables

But why do we need them?

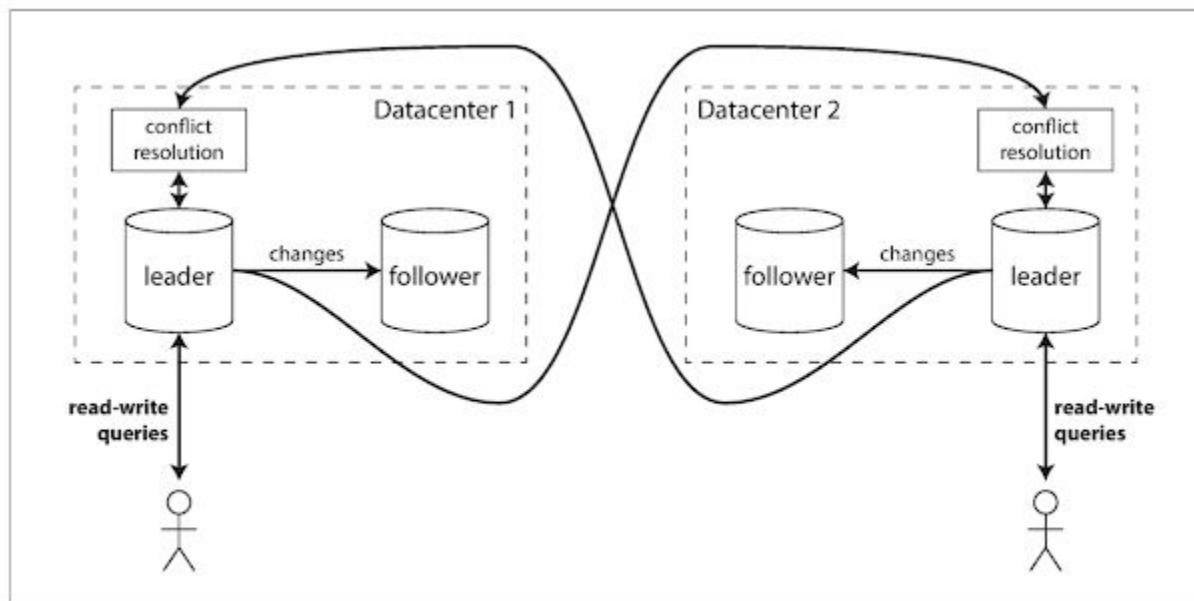


Figure 5-6. Multi-leader replication across multiple datacenters.

# REGIONAL tables

But why do we need them?

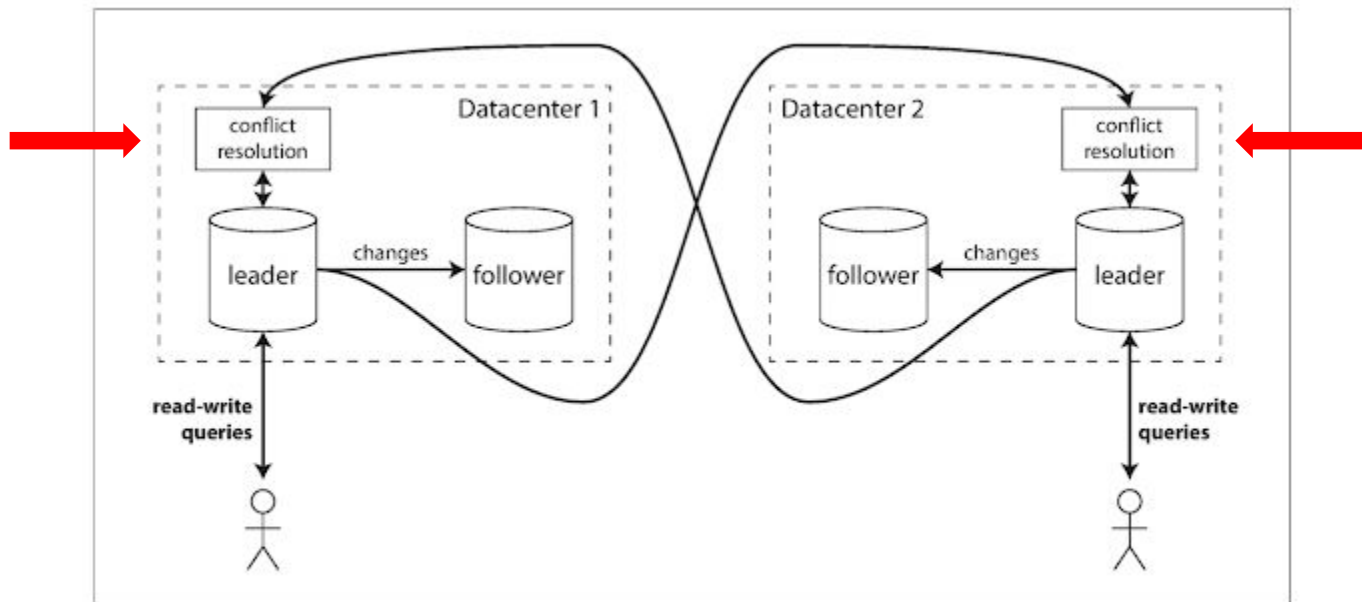


Figure 5-6. Multi-leader replication across multiple datacenters.

# REGIONAL BY ROW tables

Localization, at a row-level

```
> CREATE TABLE orders (  
  id INT PRIMARY KEY,  
  item STRING,  
  price FLOAT  
) LOCALITY REGIONAL IN "us-west1"  
  
> INSERT INTO orders VALUES (...)
```

id	item	price
1	Bat	1.11
2	Ball	2.22
3	Glove	3.33

# REGIONAL BY ROW tables

Localization, at a row-level

```
> CREATE TABLE orders (  
  id INT PRIMARY KEY,  
  item STRING,  
  price FLOAT  
) LOCALITY REGIONAL BY ROW  
  
-- ALTER TABLE orders ADD COLUMN  
  crdb_region crdb_internal_region  
  NOT NULL NOT VISIBLE  
  
-- ALTER TABLE orders  
  PARTITION BY LIST (crdb_region)
```

<i>crdb_region</i> (hidden)	id	item	price
<i>us-west1</i>	1	Bat	1.11
<i>us-east1</i>	2	Ball	2.22
<i>europa-west1</i>	3	Glove	3.33

# REGIONAL BY ROW tables

Localization, at a row-level

```
> SELECT * FROM orders
```

id	item	price
1	Bat	1.11
2	Ball	2.22
3	Glove	3.33

```
> INSERT INTO orders  
VALUES (4, 'Helmet', 4.44)
```

```
> SELECT *, crdb_region FROM orders
```

id	item	price	crdb_region
1	Bat	1.11	us-west1
2	Ball	2.22	us-east1
3	Glove	3.33	eu-west1

```
> INSERT INTO  
orders (id, item, price, crdb_region)  
VALUES (5, 'Hat', 5.55, 'eu-west1')
```

*“Users can interact with the hidden crdb\_region column to control homing, but they do not need to”*



# REGIONAL BY ROW tables

Auto-homing and Re-homing

```
> SHOW COLUMNS FROM orders
```

column_name	data_type	is_hidden	column_default
id	INT8	false	NULL
item	STRING	false	NULL
price	FLOAT8	false	NULL
crdb_region	crdb_internal_region	true	gateway_region()

```
> INSERT INTO orders VALUES (6, 'Pants', 6.66) RETURNING crdb_region
```

```
crdb_region
```

```
-----  
us-east1
```

# REGIONAL BY ROW tables

Auto-homing and Re-homing

```
> INSERT INTO orders VALUES (6, 'Pants', 6.66) RETURNING crdb_region
```

```
crdb_region
```

```
-----
```

```
us-east1
```

```
> UPDATE orders SET crdb_region = 'europe-west1' WHERE id = 6
```

```
> SELECT *, crdb_region FROM orders WHERE id = 6
```

```
id | item  | price | crdb_region
```

```
-----+-----+-----+-----
```

```
6 | Pants | 6.66 | europe-west1
```

# REGIONAL BY ROW tables

Preserving global uniqueness

```
UNIQUE (email) != UNIQUE (crdb_region, email)
```

But we are indexing on (crdb\_region, email) due to **partitioning!**

How do we enforce uniqueness of emails?

**Trick:** crdb\_region is an **enum** → all possible values known

```
SQL optimizer turns SELECT count(*) = 1 FROM t WHERE email = $1
```

```
into SELECT count(*) = 1 FROM t WHERE (crdb_region, email)
      IN (('us-east1', $1), ('us-west1', $1), ...)
```

# REGIONAL BY ROW tables

Preserving global uniqueness, quickly

Tables with **no UNIQUE constraints**

```
Ex. CREATE TABLE logs (ts TIMESTAMP, msg STRING)
```

UNIQUE constraints on **UUID column** generated with **gen\_random\_uuid()**

Builtin often hidden in a **DEFAULT expression**

```
Ex. CREATE TABLE rides (  
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),  
    start_loc GEOGRAPHY, end_loc GEOGRAPHY)
```

Specialized **sequence-like data type** that rejects user-supplied values (in development)

# REGIONAL BY ROW tables

Locality optimized search

```
> EXPLAIN SELECT * FROM orders WHERE id = 5
```

```
-----  
• union all  
  | limit: 1  
  |  
  | • scan  
  |   table: orders@primary  
  |   spans: [/'us-east1'/5 - /'us-east1'/5]  
  |  
  | • scan  
  |   table: orders@primary  
  |   spans: [/'europe-west1'/5 - /'europe-west1'/5]  
  |           [/'us-west1'/5 - /'us-west1'/5]
```

# REGIONAL BY ROW tables

Locality optimized search for mutations

```
> EXPLAIN UPDATE orders SET price = 9 WHERE id = 5
```

```
-----  
• update  
  | table: orders  
  |  
  | • union all  
  |   limit: 1  
  |  
  |   • scan  
  |     spans: [/'us-east1'/5 - /'us-east1'/5]  
  |  
  |   • scan  
  |     spans: [/'europe-west1'/5 - /'europe-west1'/5]  
  |           [/'us-west1'/5 - /'us-west1'/5]
```

# REGIONAL BY ROW tables

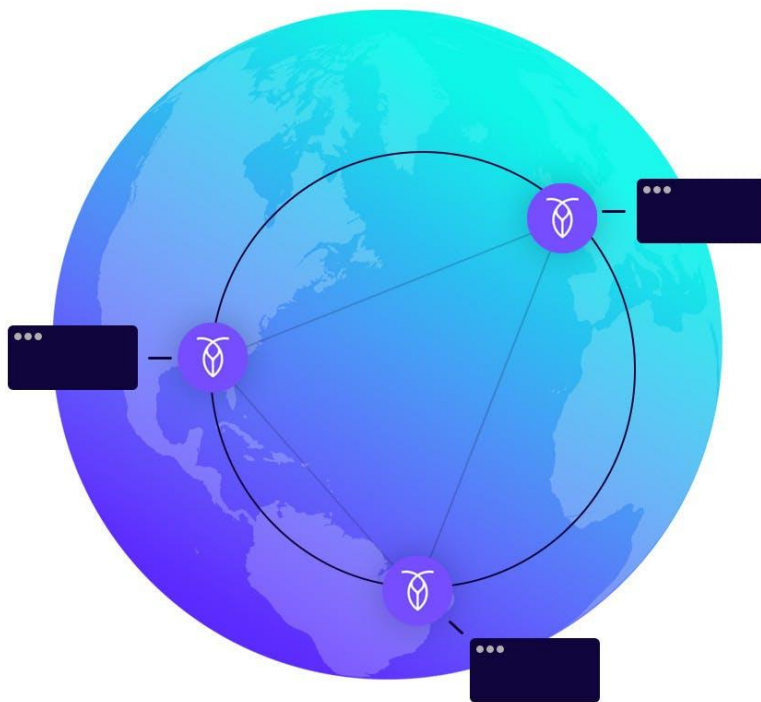
## Latency Profile

	Locally homed	Remotely homed
INSERT	<b>Local</b> if no uniqueness check, otherwise <b>Remote</b>	N/A
UPDATE	<b>Local</b>	<b>Remote</b>
DELETE	<b>Local</b>	<b>Remote</b>
SELECT (lookup)	<b>Local</b>	<b>Remote</b>
SELECT (scan)	<b>Remote</b>	<b>Remote</b>
Stale SELECT (lookup/scan) *	<b>Local</b>	<b>Local</b>

Local = 1 - 3ms

Remote = 30 - 120ms

\* Stale reads only possible in read-only transactions



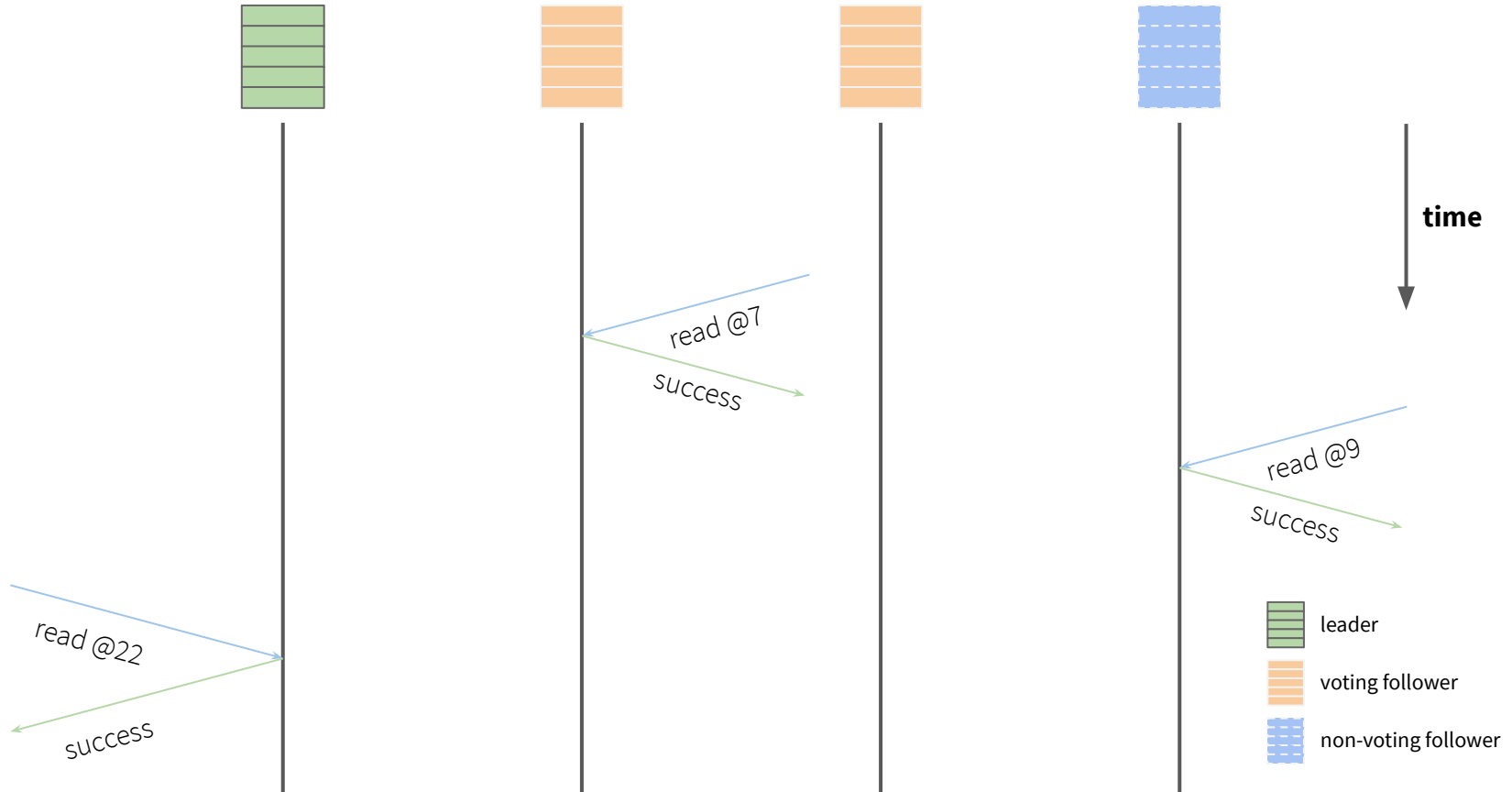
# GLOBAL tables

Meant for global data



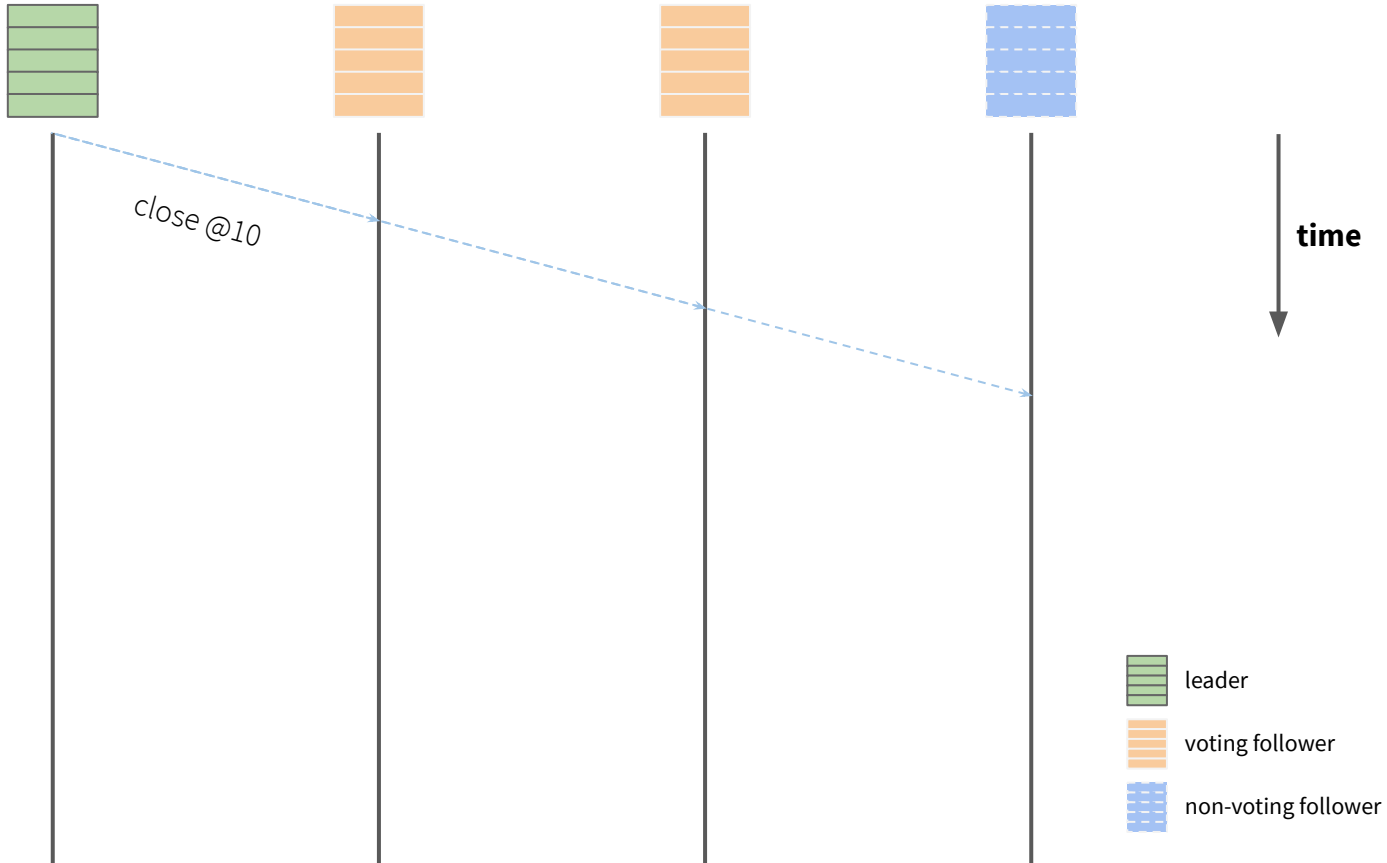
# Stale Reads

Low-Latency Reads, Everywhere



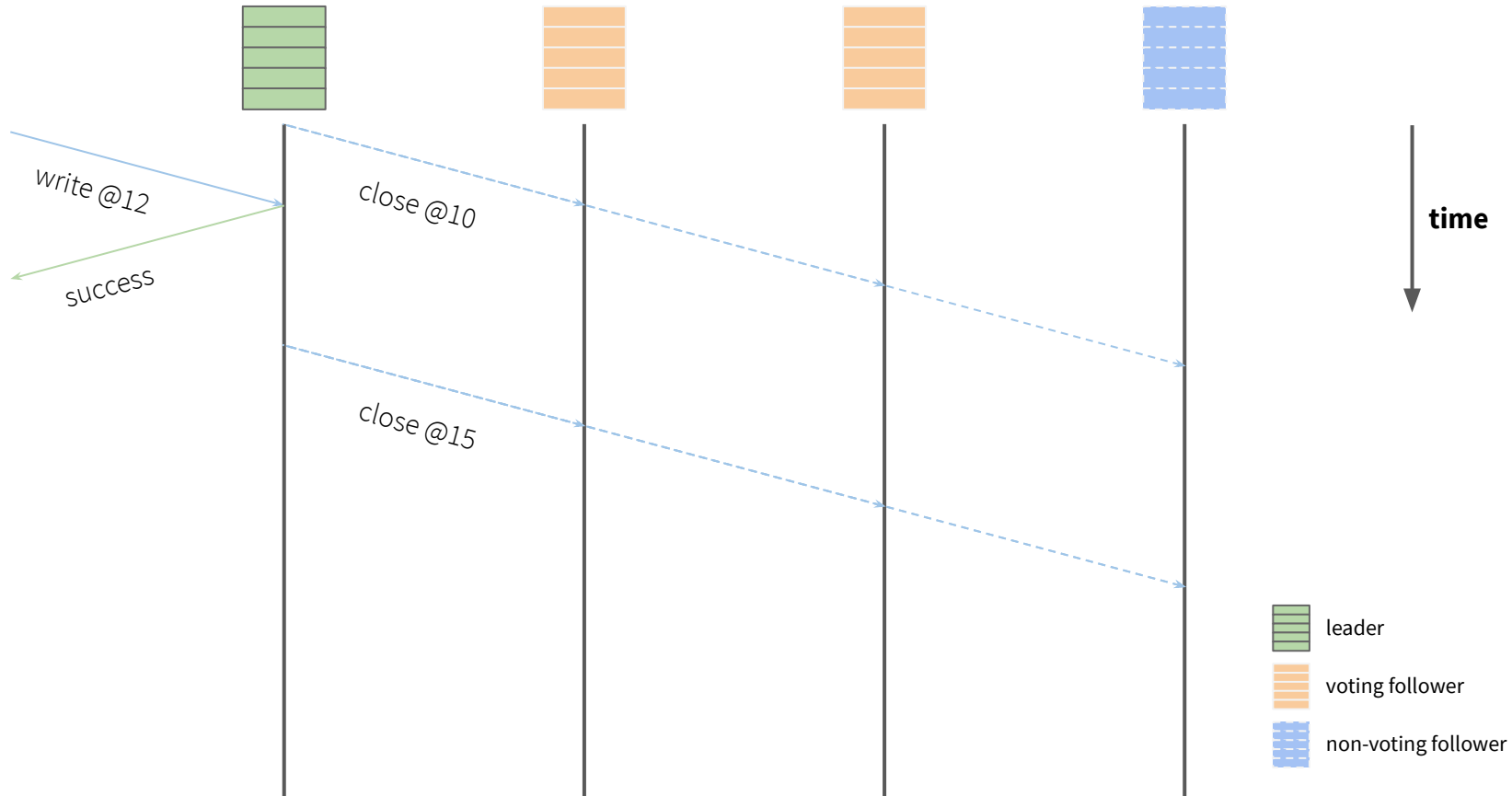
# Stale Reads

Low-Latency Reads, Everywhere



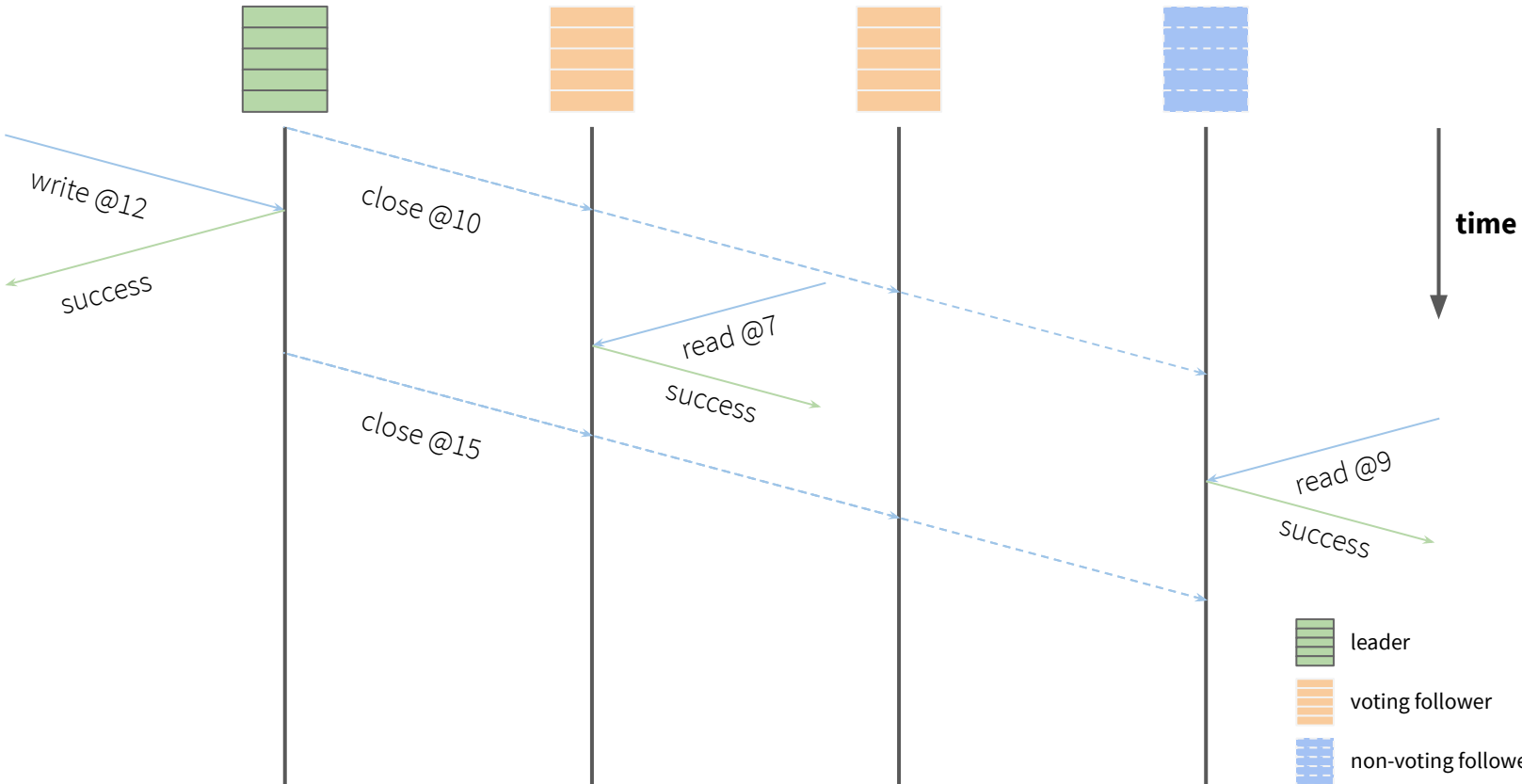
# Stale Reads

Low-Latency Reads, Everywhere



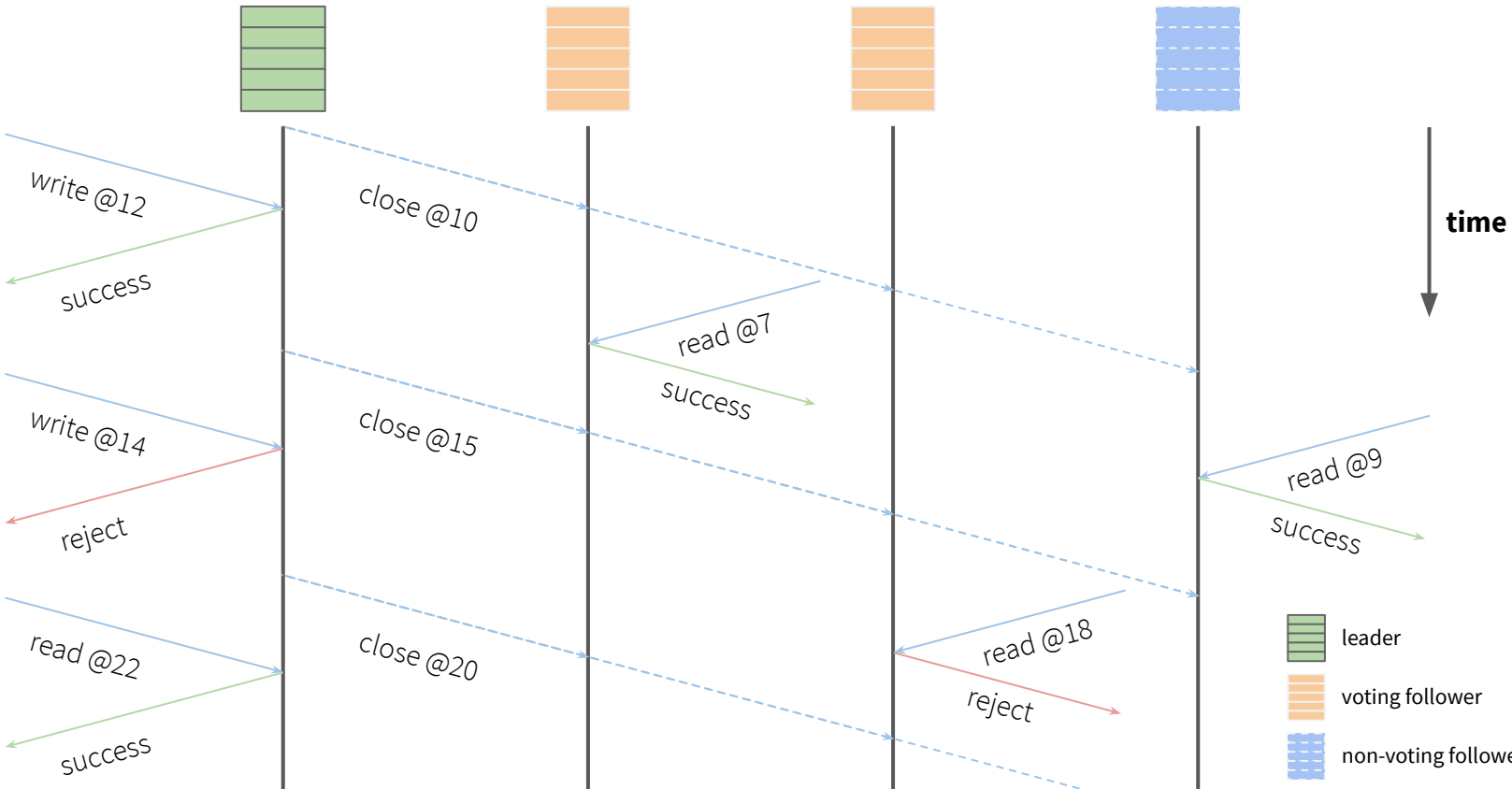
# Stale Reads

Low-Latency Reads, Everywhere



# Stale Reads

Low-Latency Reads, Everywhere



# Stale Reads

Low-Latency Reads, Everywhere

**Exact staleness** - Client-provided staleness

**Bounded staleness** (in development) - Client-provided staleness limit, dynamic staleness

## Benefits

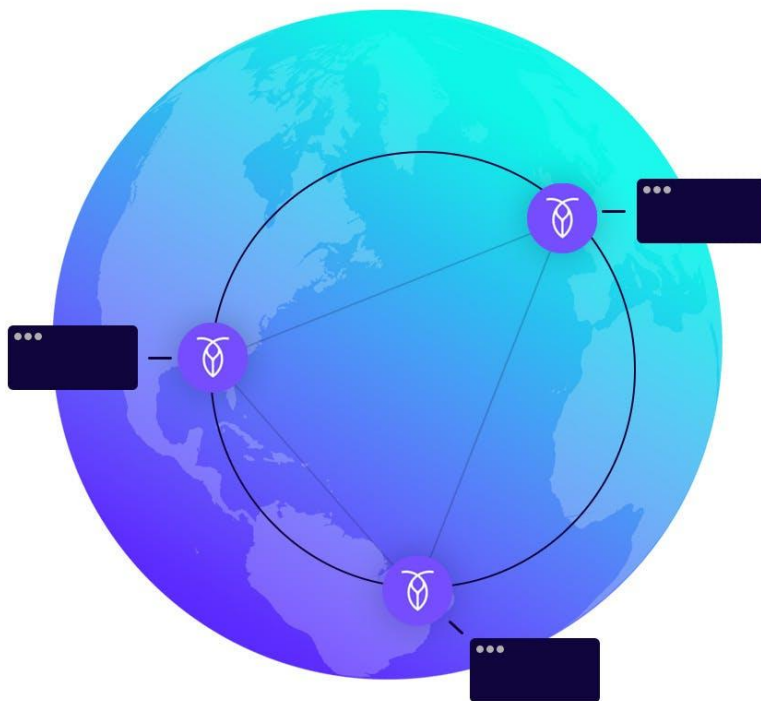
- Low-latency reads from all regions

## Limitations

- Stale results (~3 seconds)
- Can only be used in **read-only transactions!**

```
> SELECT * FROM orders
  AS OF SYSTEM TIME '-3s'
```

id	item	price
1	Bat	1.11
2	Ball	2.22
3	Glove	3.33

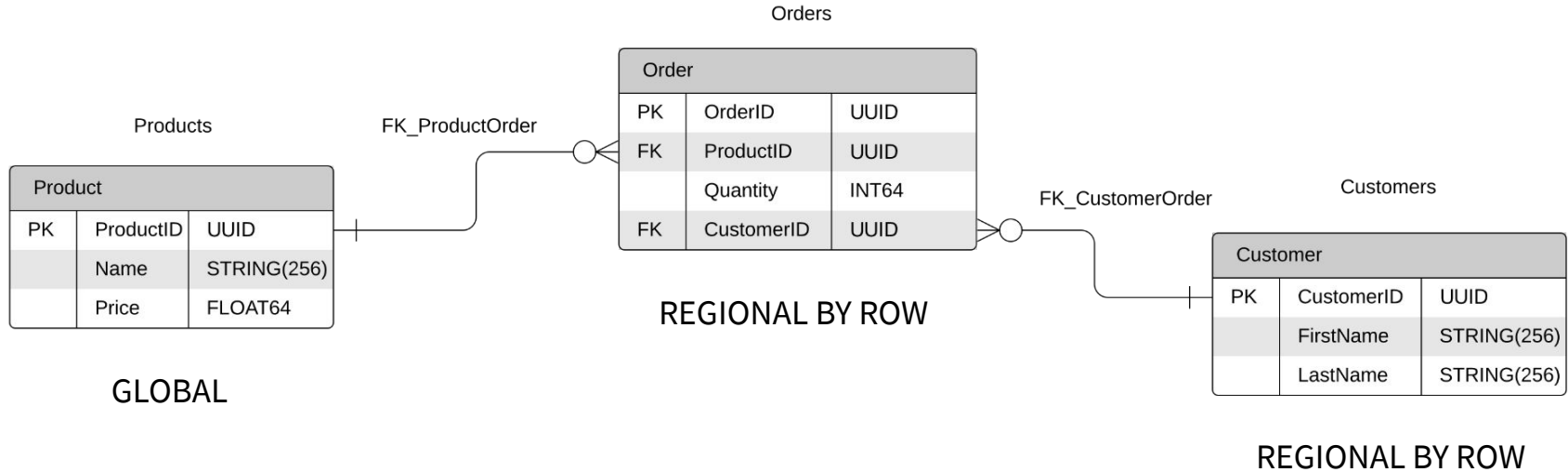


# GLOBAL tables

Meant for global data

# GLOBAL tables

Meant for global data

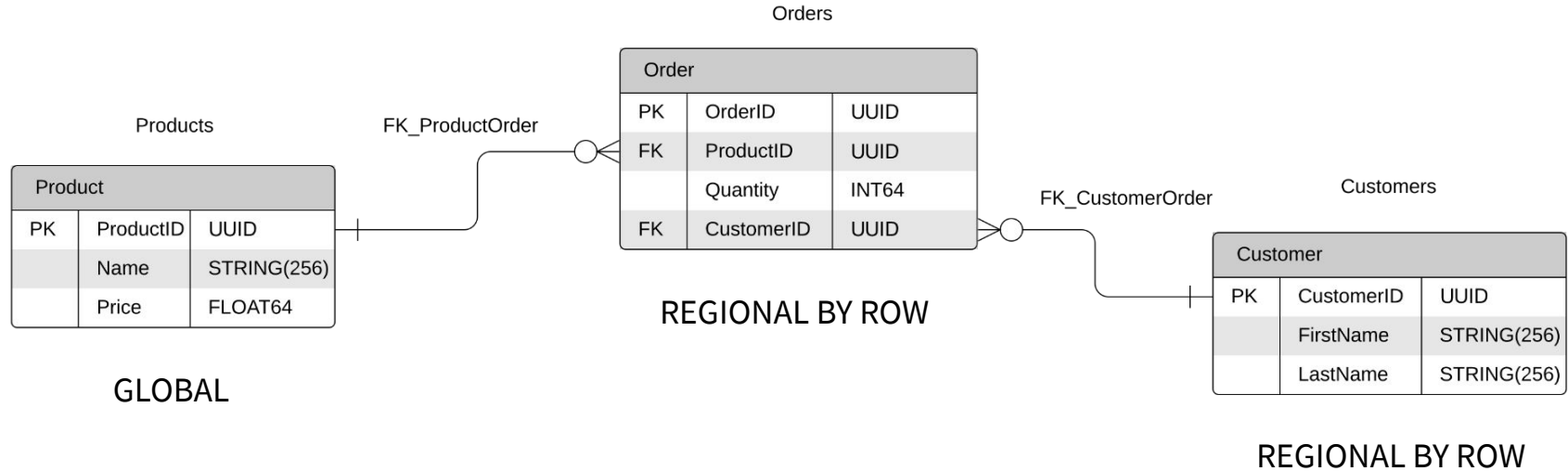


```
> ALTER TABLE Products SET LOCALITY GLOBAL
```



# GLOBAL tables

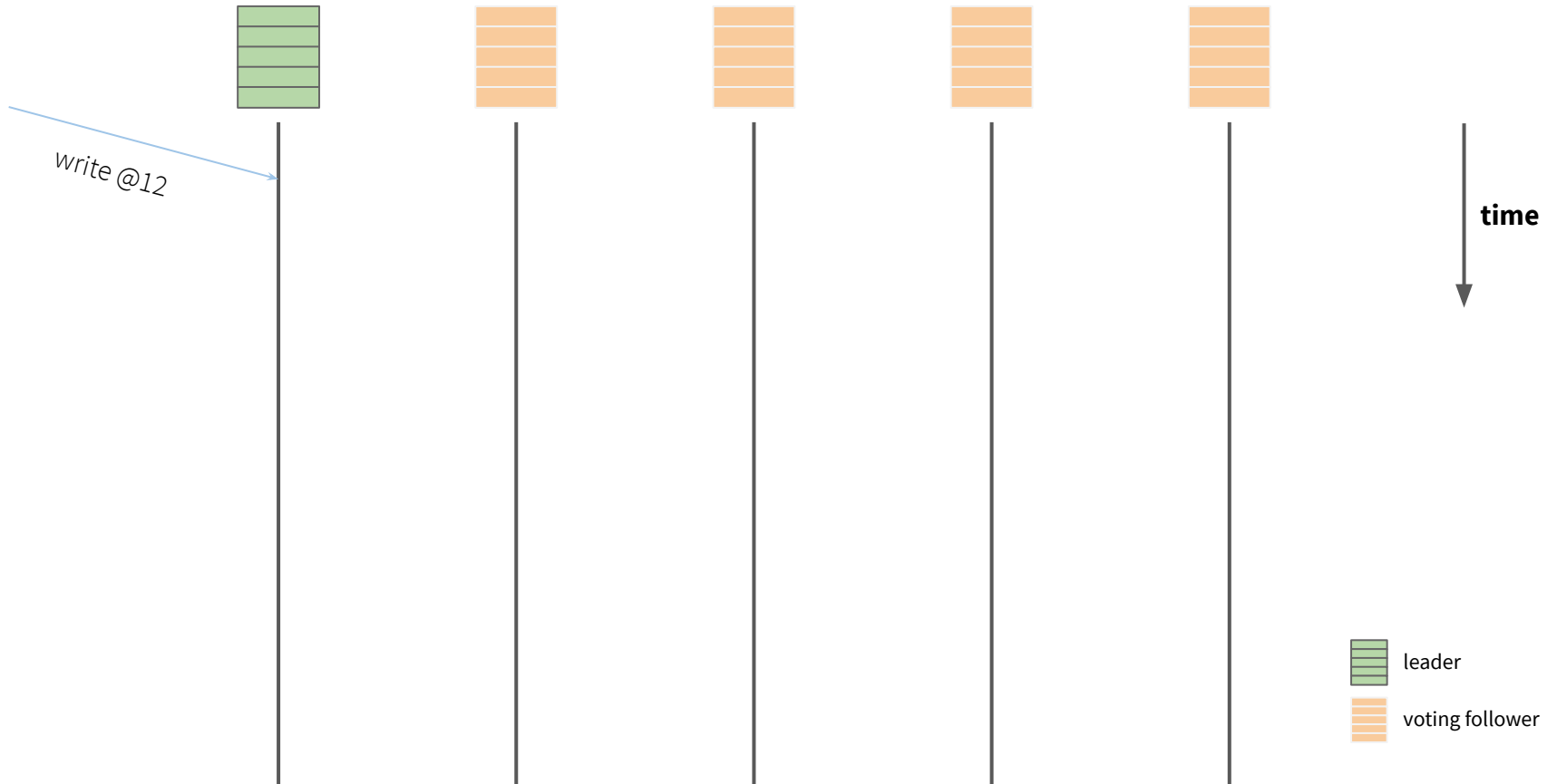
Meant for global data



```
> INSERT INTO Orders VALUES (gen_random_uuid(), 123, 3, 789)
```

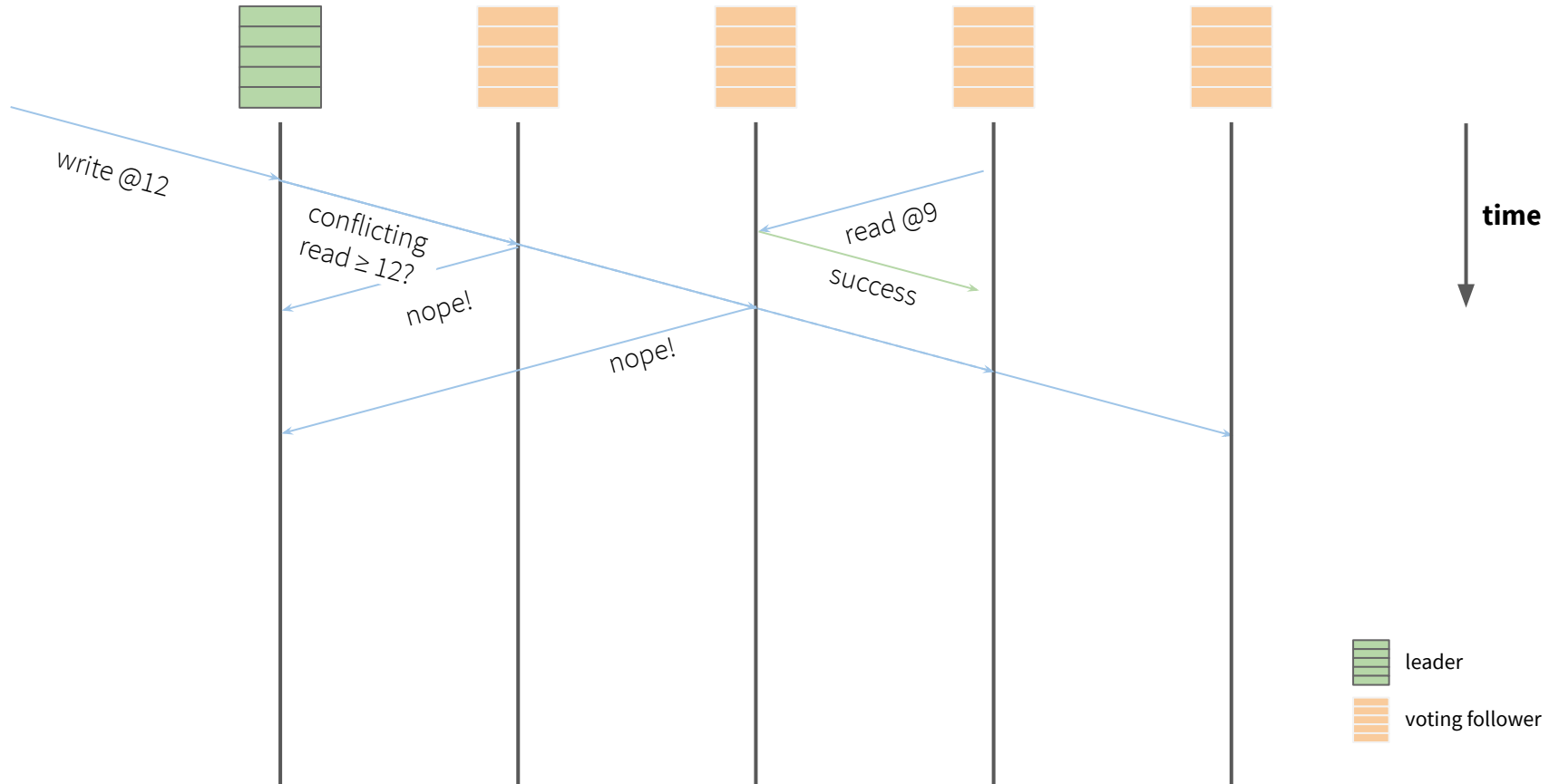
# GLOBAL tables

Meant for global data



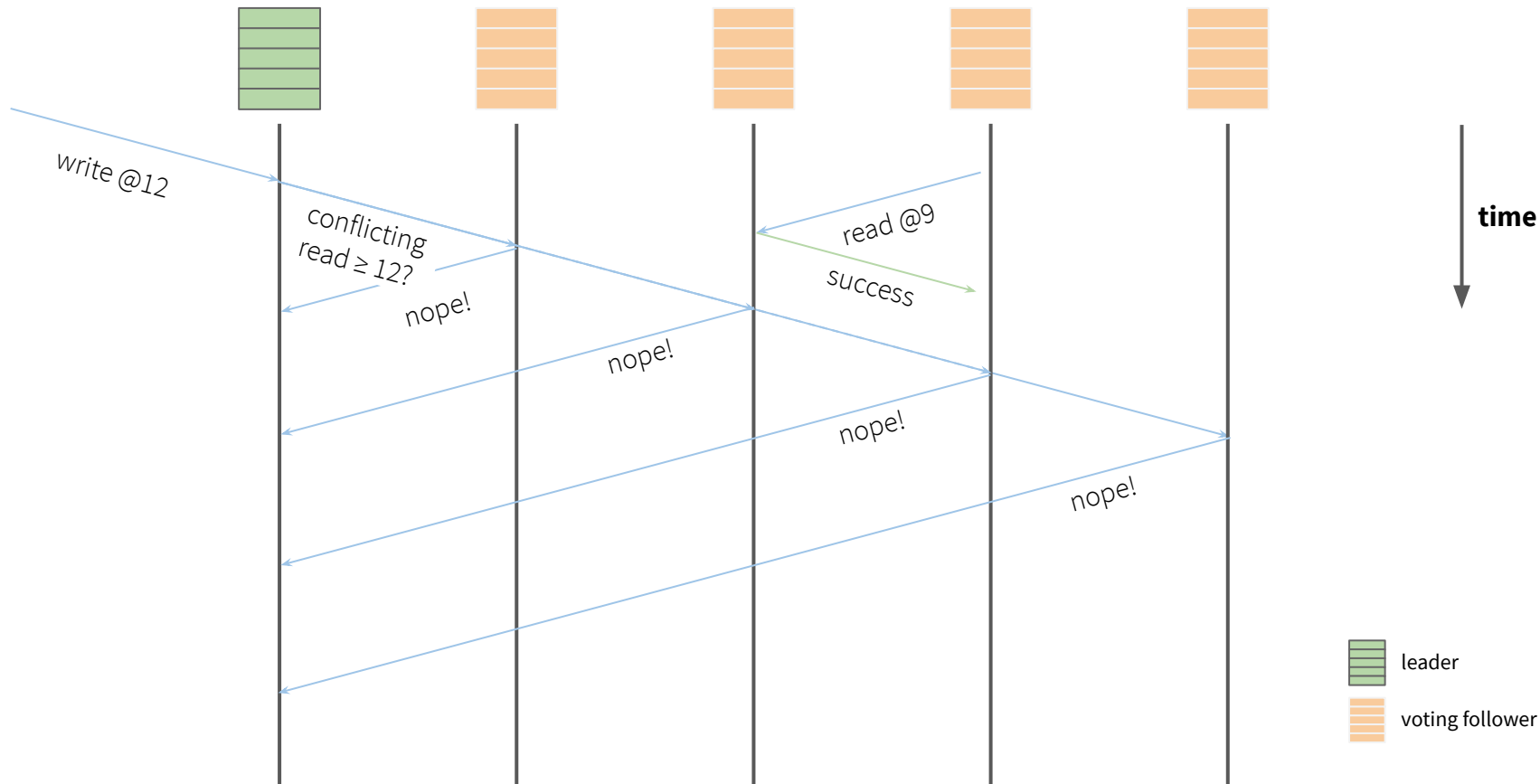
# GLOBAL tables

Meant for global data



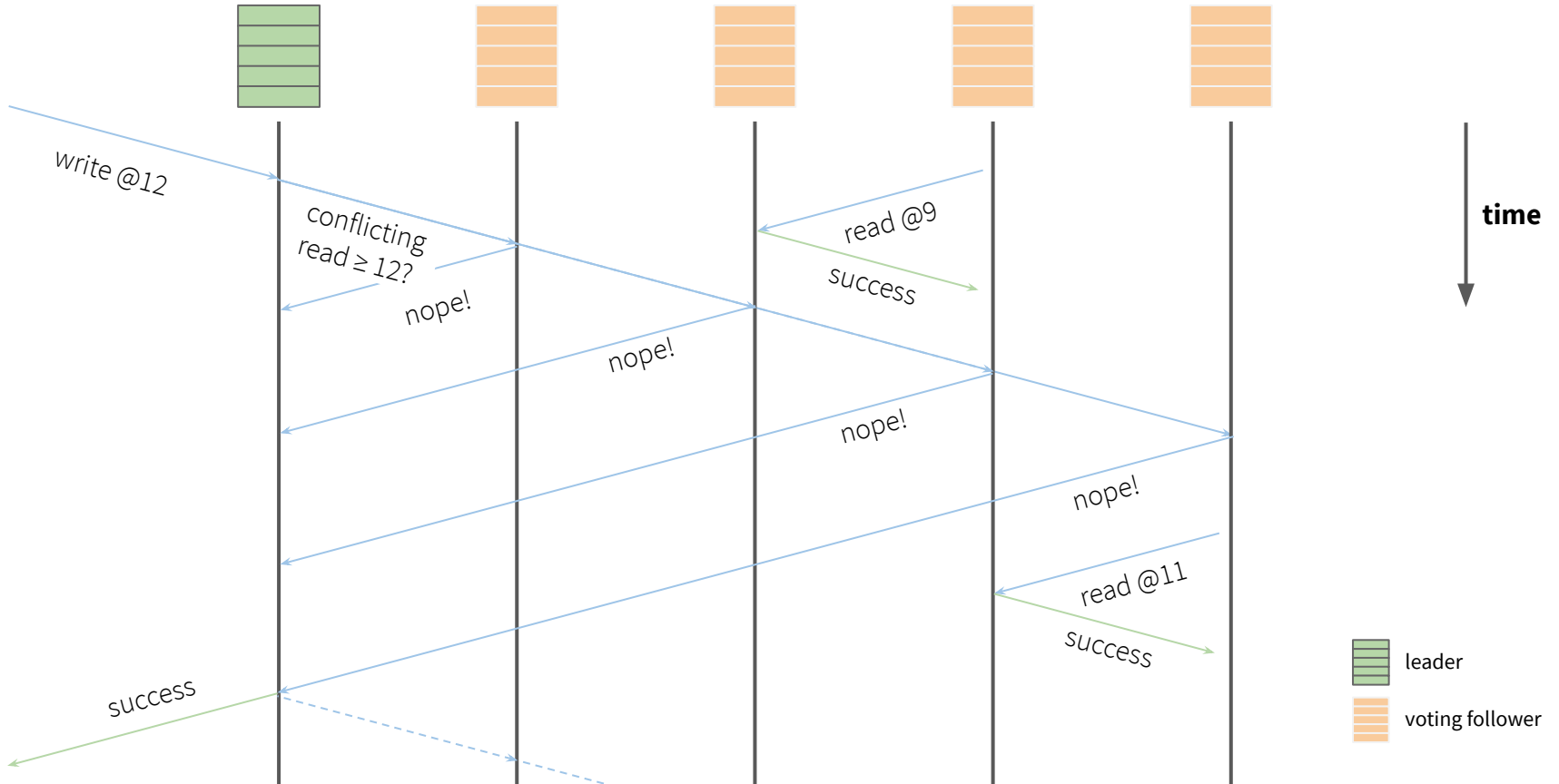
# GLOBAL tables

Meant for global data



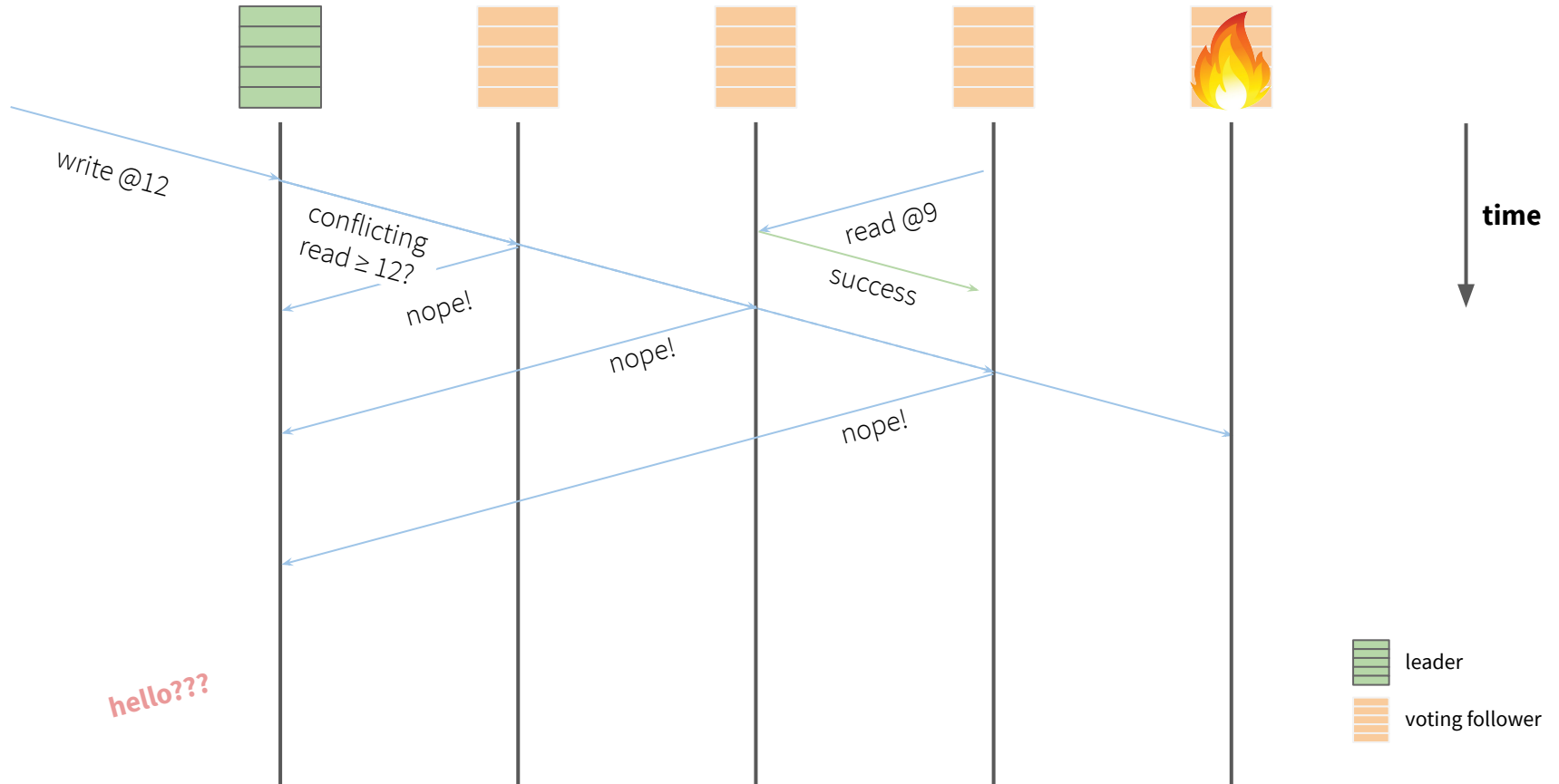
# GLOBAL tables

Meant for global data



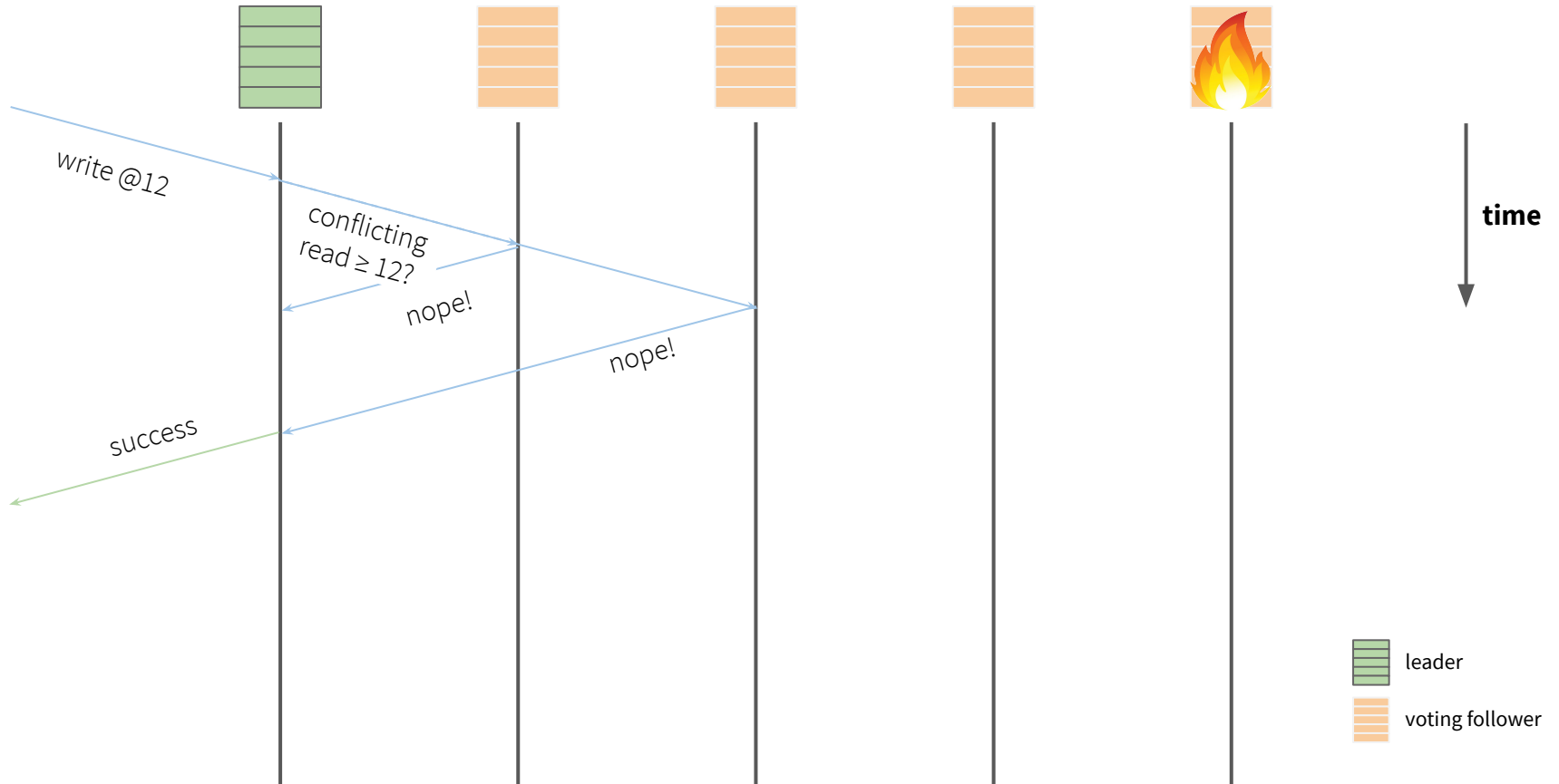
# GLOBAL tables

Meant for global data



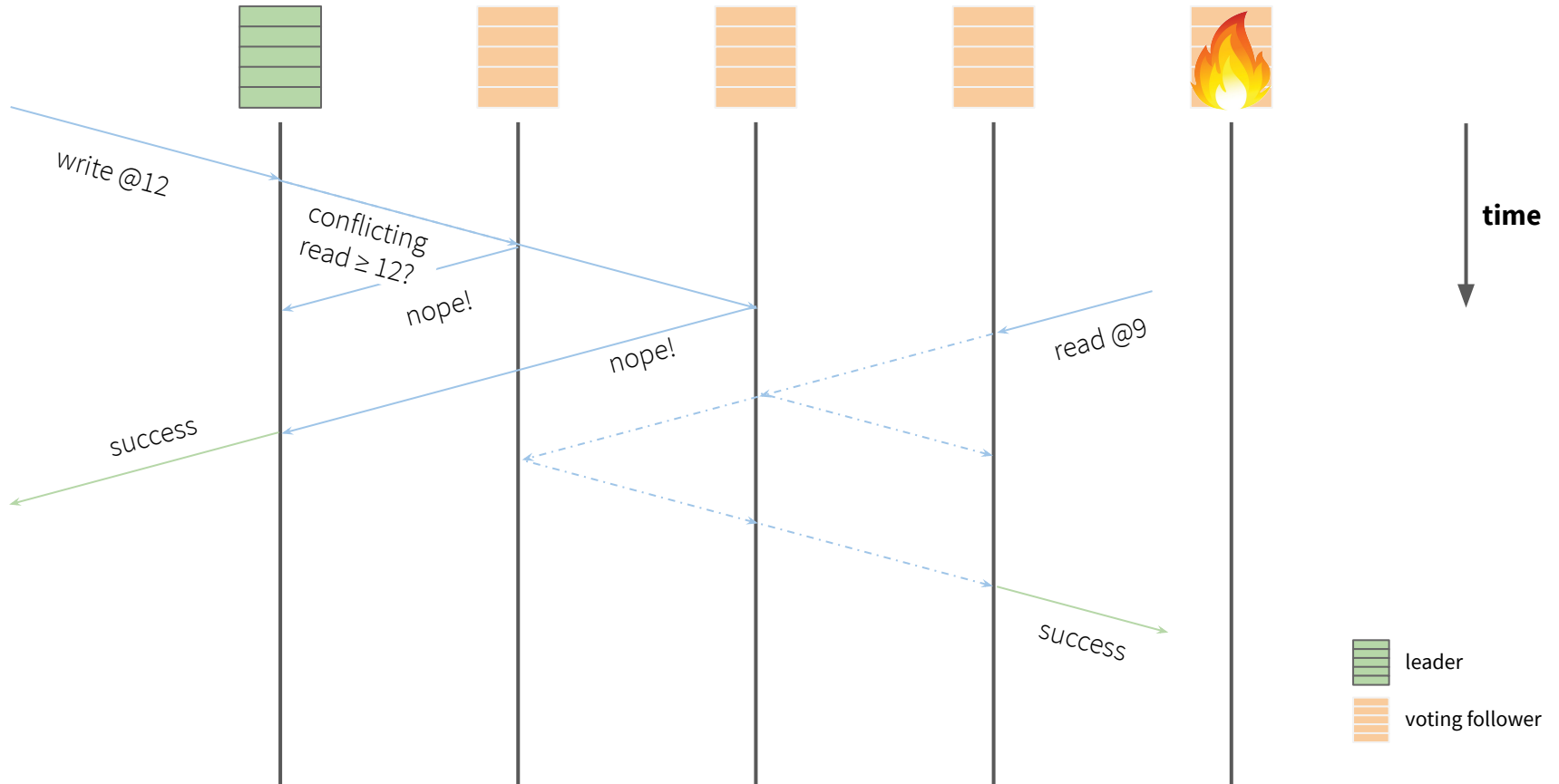
# GLOBAL tables

Meant for global data



# GLOBAL tables

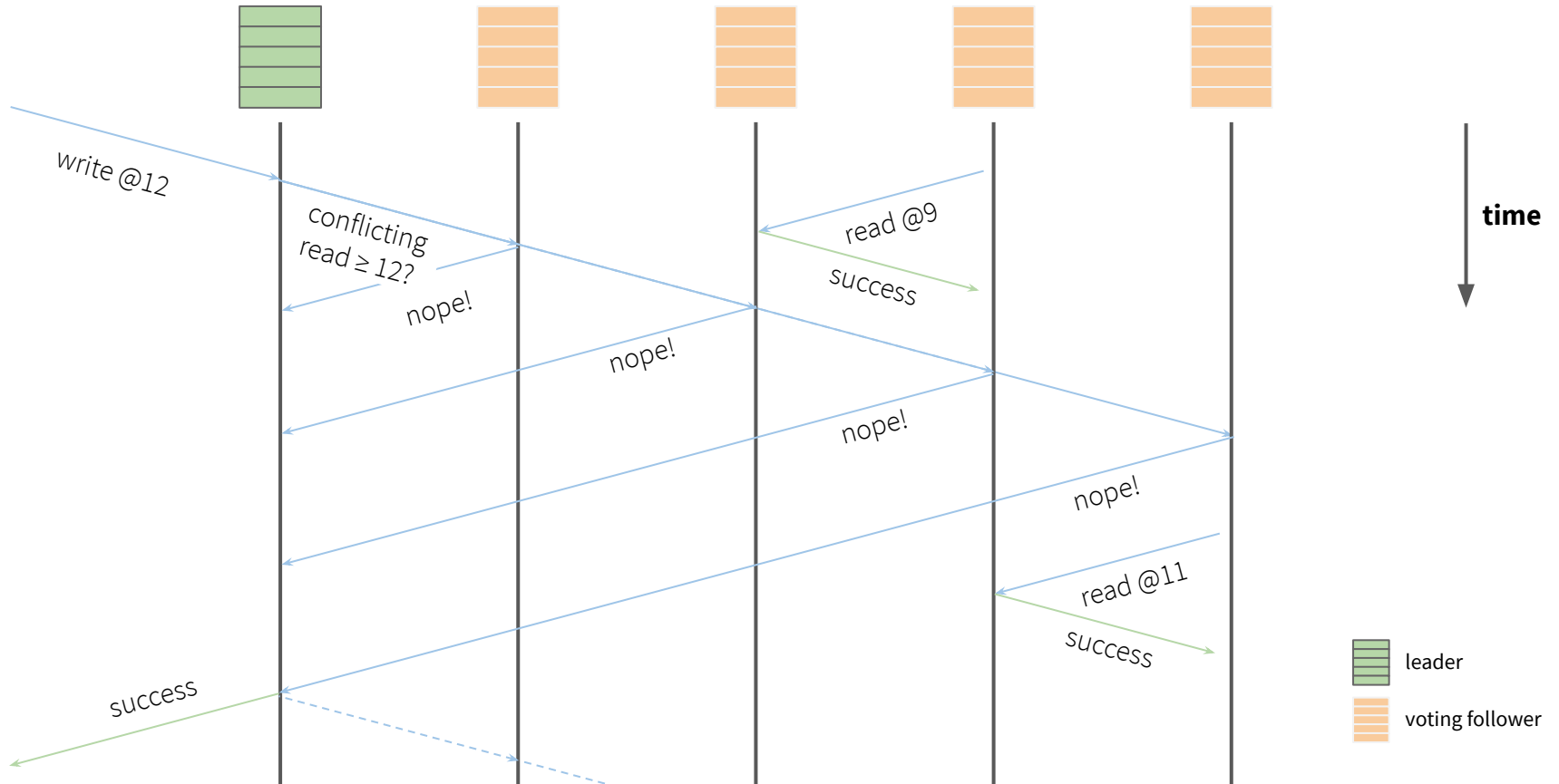
Meant for global data





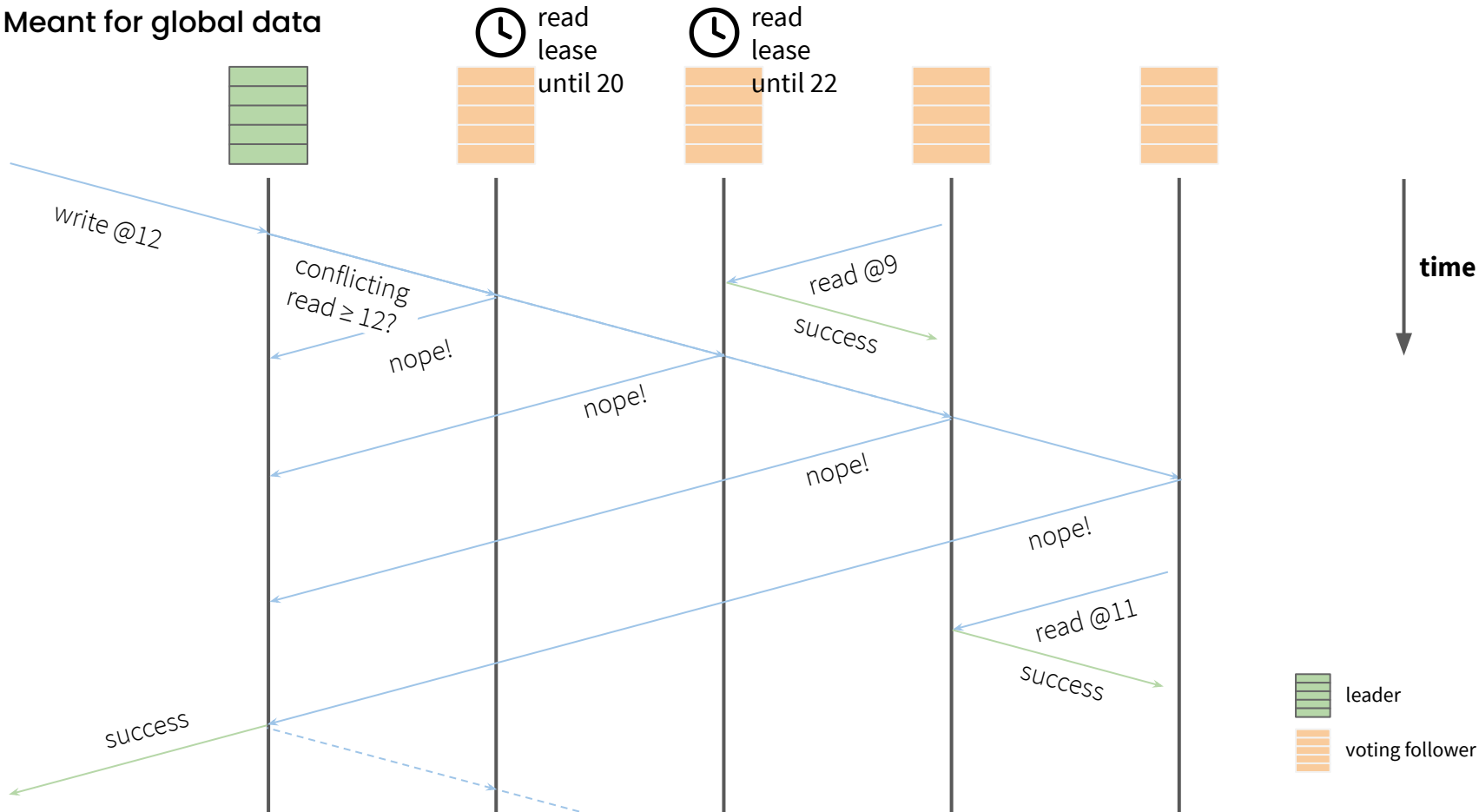
# GLOBAL tables

Meant for global data



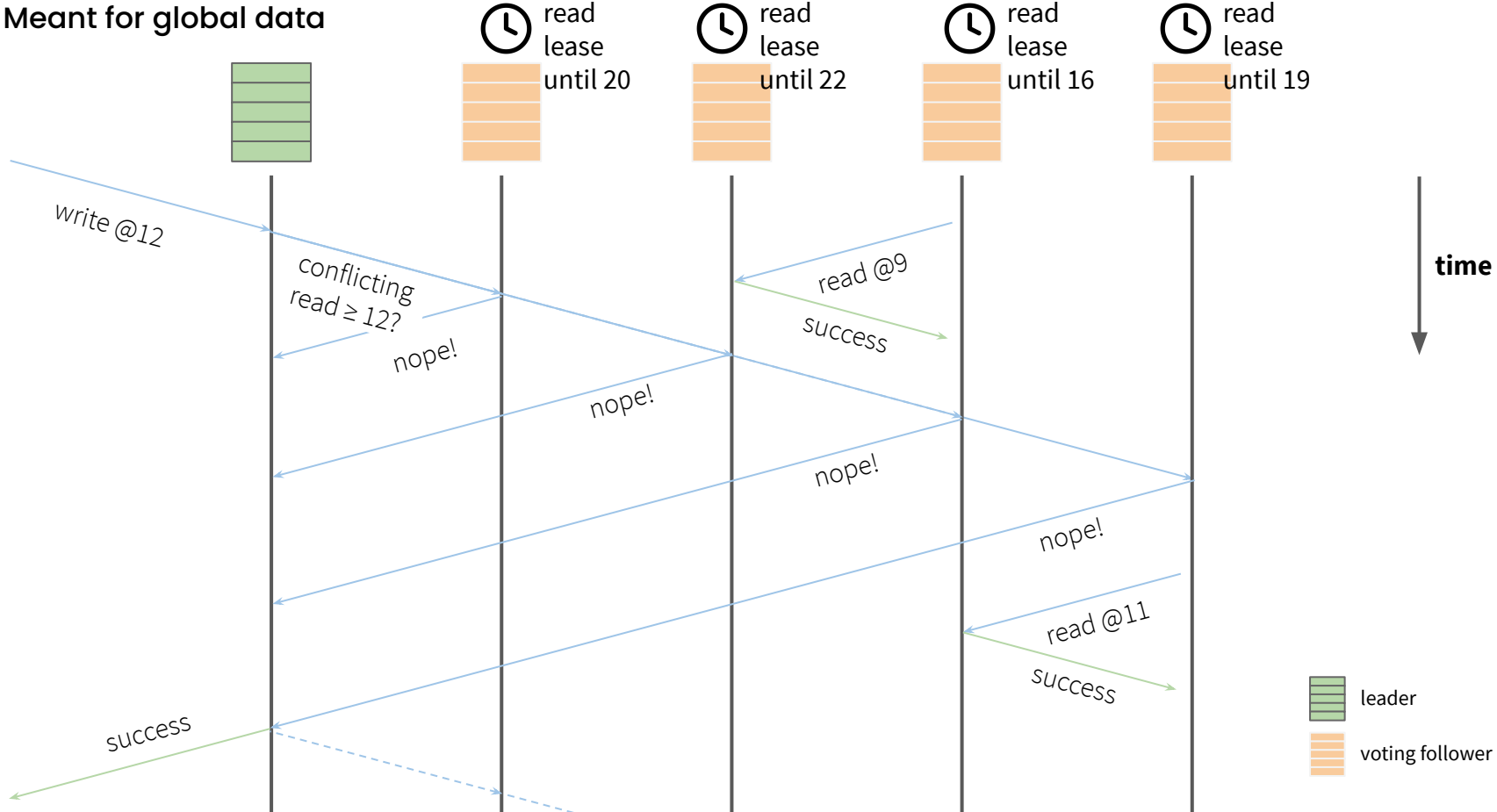
# GLOBAL tables

Meant for global data



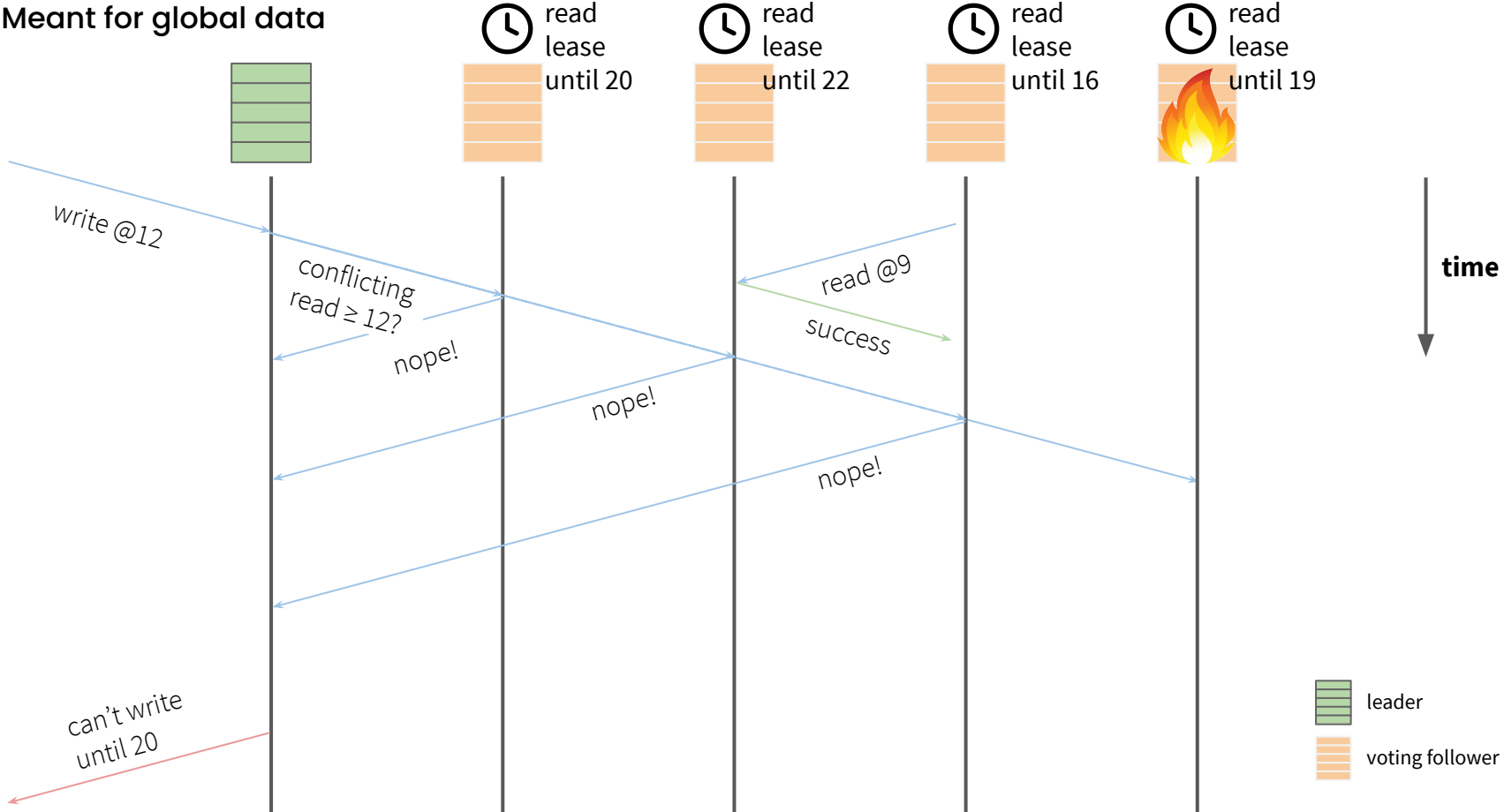
# GLOBAL tables



Meant for global data



# GLOBAL tables

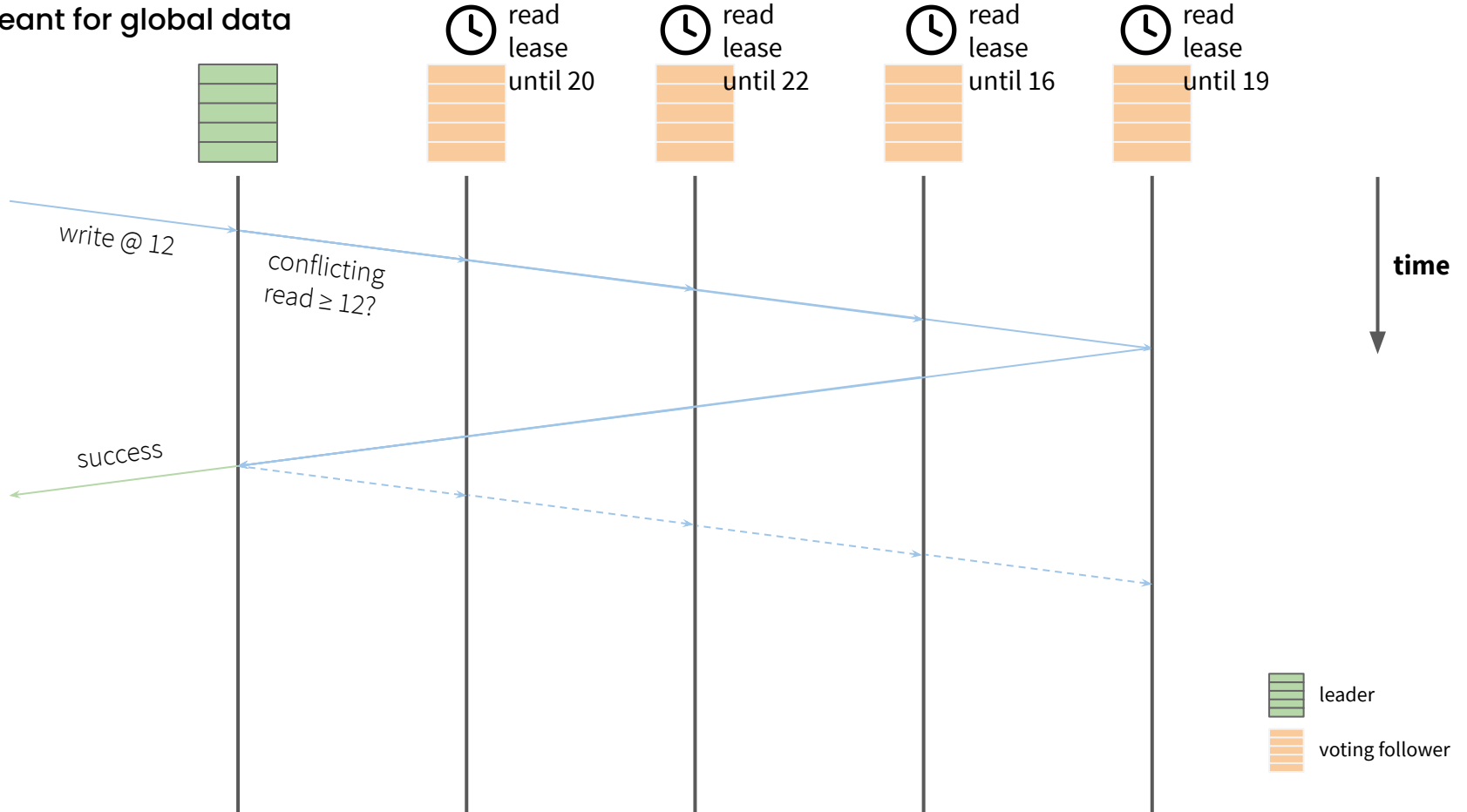
Meant for global data



 leader  
 voting follower

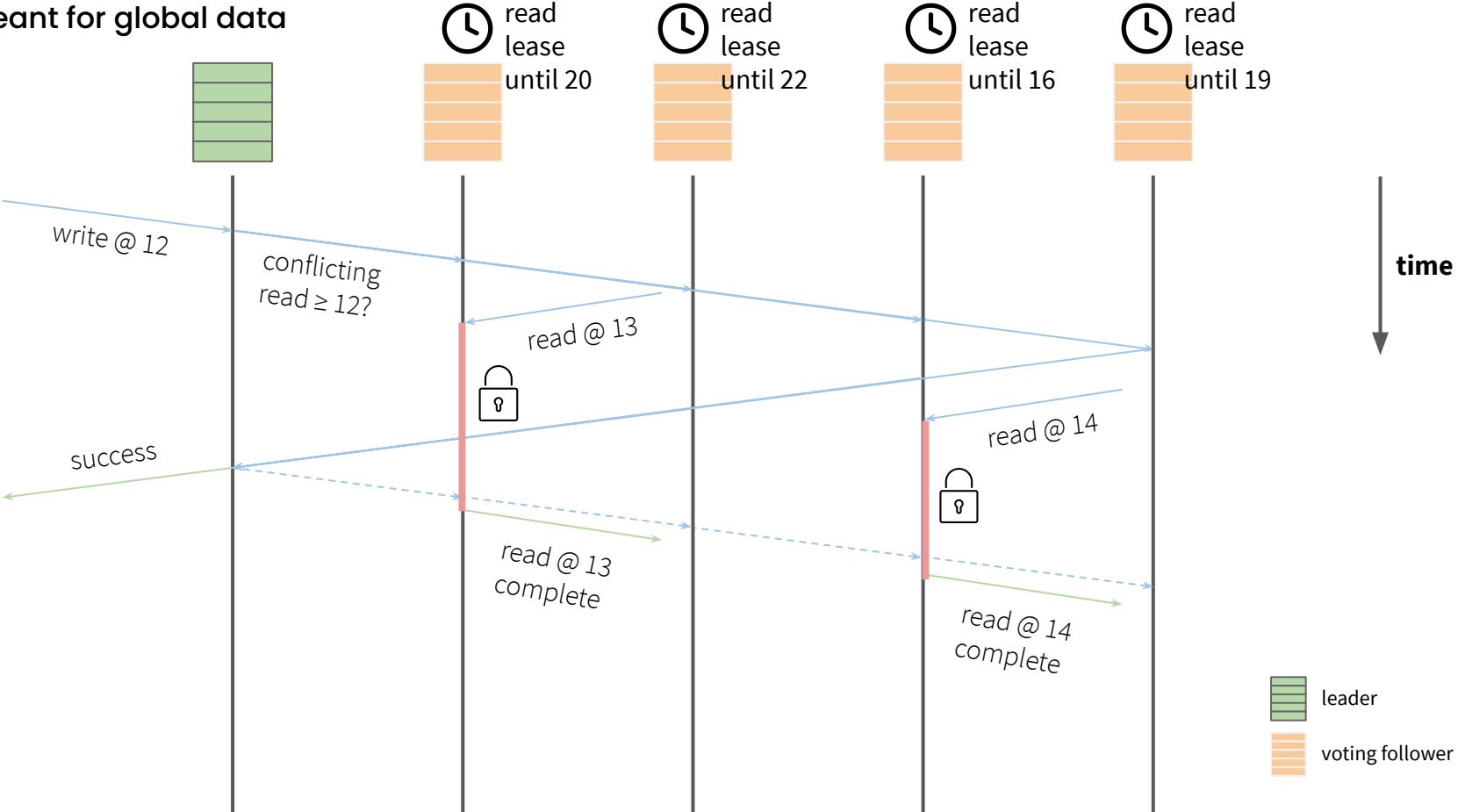
# GLOBAL tables

Meant for global data



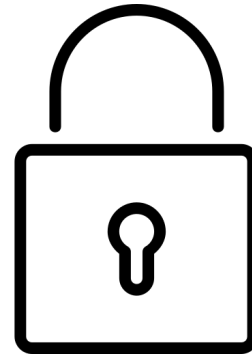
# GLOBAL tables

Meant for global data



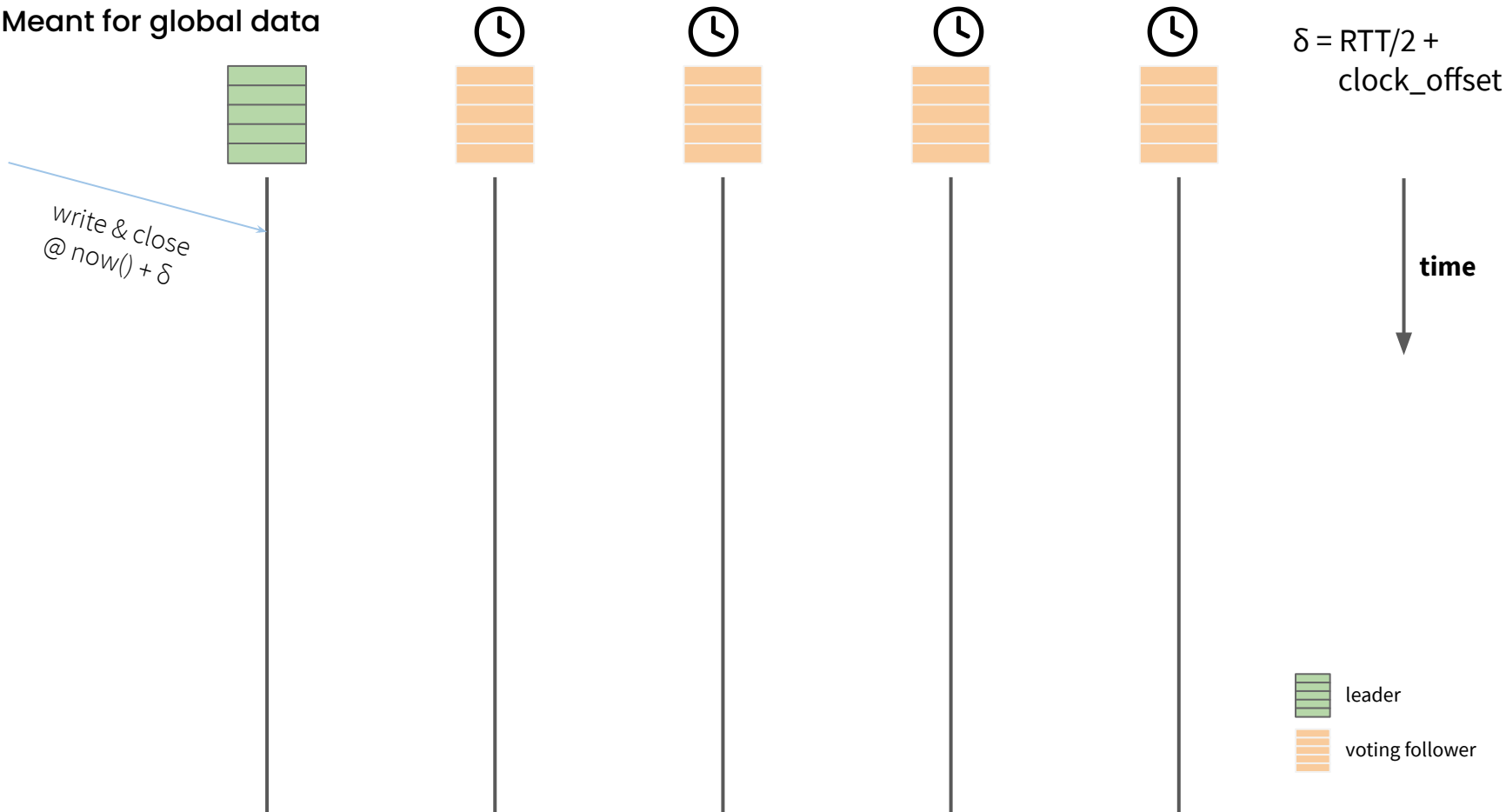
# GLOBAL tables

Meant for global data



# GLOBAL tables

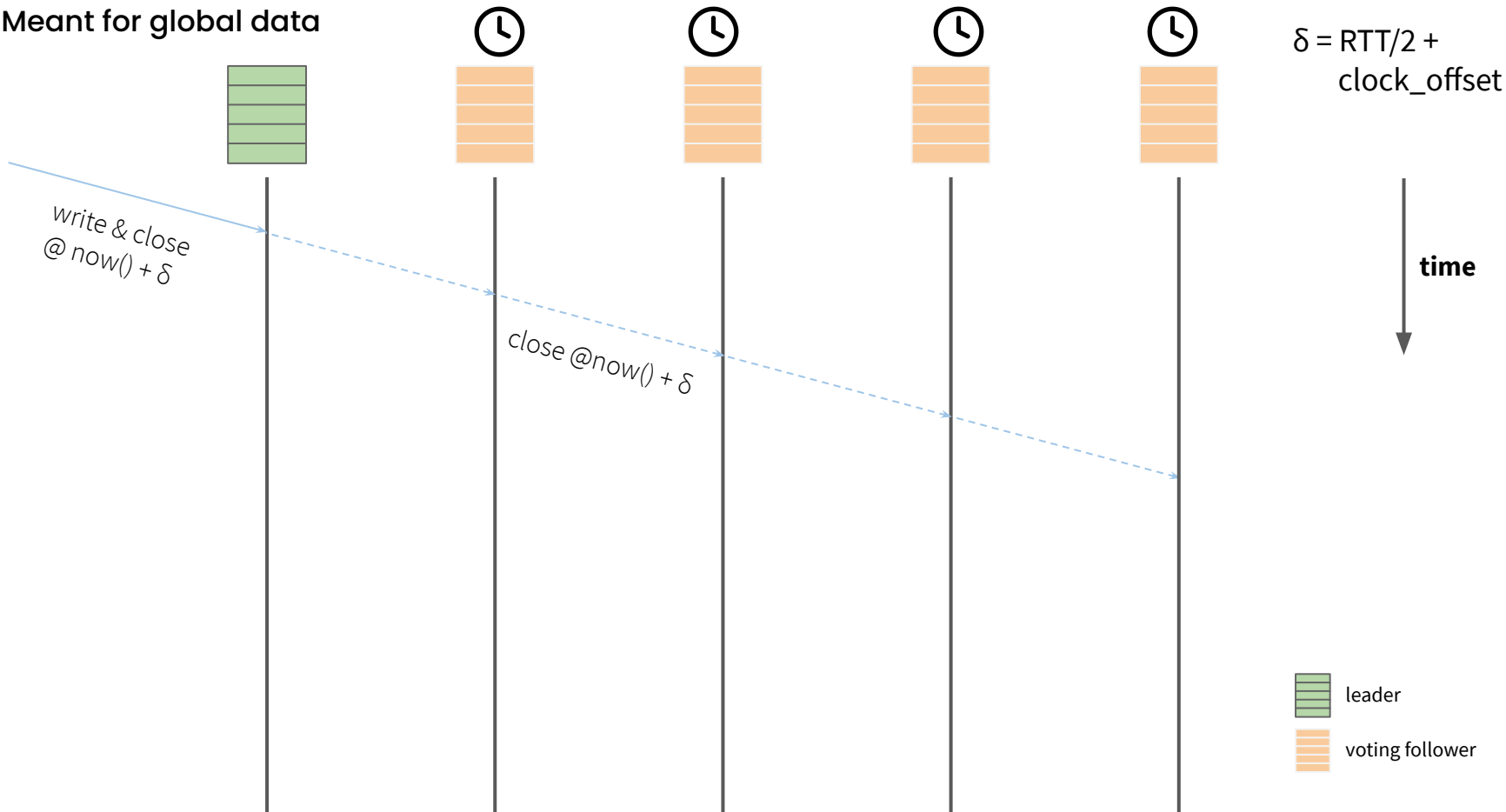
Meant for global data





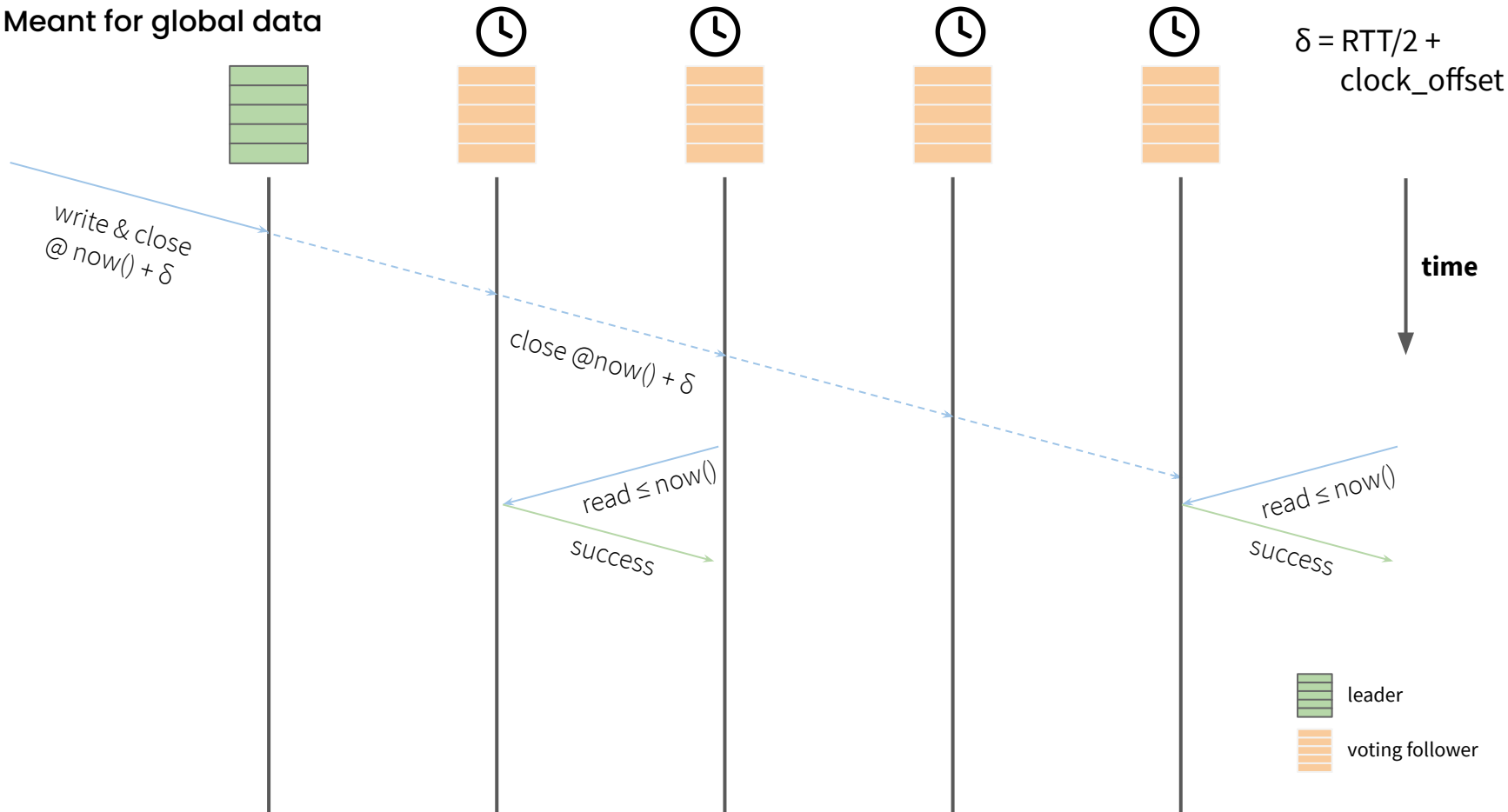
# GLOBAL tables

Meant for global data



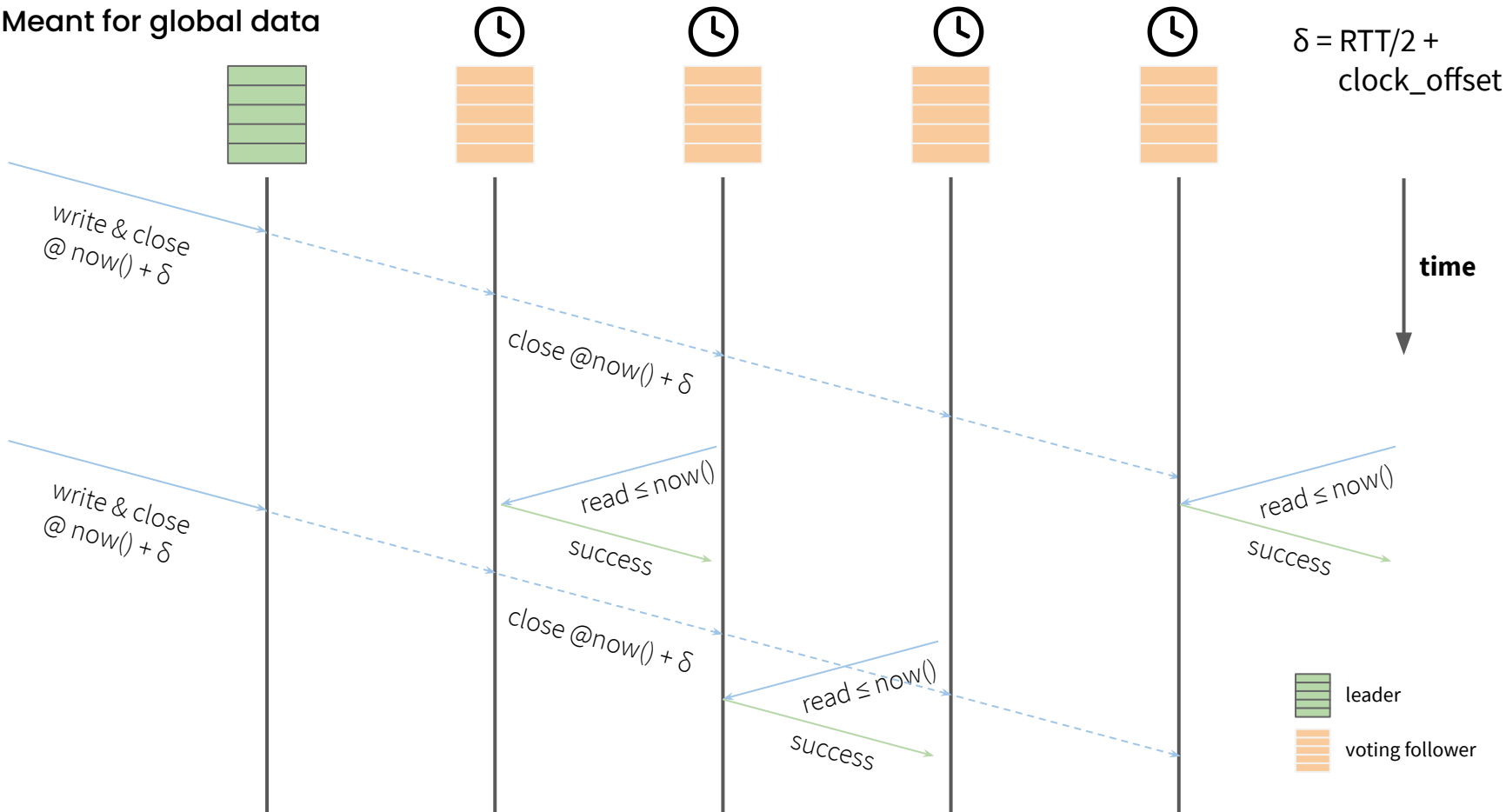
# GLOBAL tables

Meant for global data



# GLOBAL tables

Meant for global data



# GLOBAL tables

Meant for global data

Consistency (“linearizability”) = **read-your-writes** + **monotonic-reads**

1. After committing, writes **commit wait** until visible on all followers
2. When reads see write in **uncertainty interval**, wait before retrying

# GLOBAL tables

Meant for global data

Hazard of clock reliance? **Stale reads**

If clock skew bounds exceeded before detection, what happens?

- Loss of causality
- + No loss of isolation

Not a new concern — already present due to **read lease mechanism**

# GLOBAL tables

Meant for global data

## Benefits

- **Fast reads** from all regions
- Bounded tail-latency, below WAN communication latency

## Limitations

- **Slower writes**, must wait for clock sync and communication latency

# Table-Locality Settings

## Latency Profile Comparison

	<b>REGIONAL</b>	<b>GLOBAL</b>
Access locality	High	Low
Access patterns	Read-often, Write-often	Read-often, Write-rarely
Read latency (local home)	<b>Fast</b>	<b>Fast</b>
Read latency (remote home)	<b>Slow</b>	<b>Fast</b>
Read latency (stale)	<b>Fast</b>	<b>Fast</b>
Write latency (local home)	<b>Fast</b>	<b>Slow</b>
Write latency (remote home)	<b>Slow</b>	<b>Slow</b>

# Challenge Completed

```
> ALTER DATABASE <db> ADD REGION "us-east1",  
  "europe-east2", "asia-east1", ...  
  
> ALTER DATABASE <db> SURVIVE REGION FAILURE  
  
> ALTER TABLE Orders LOCALITY REGIONAL BY ROW  
> ALTER TABLE Customers LOCALITY REGIONAL BY ROW  
> ALTER TABLE Products LOCALITY GLOBAL  
  
> INSERT INTO Orders VALUES  
  (gen_random_uuid(), 123, 3, 789)
```

## REQUIREMENTS

### Consistency

Referential integrity across tables

### Scalability

100k+ orders per second

### High availability

Survive node/zone/region failure

### Low latency

Sub 20ms end-to-end





# CockroachDB 21.1

**Database Regions**

**Survival Goals**

**Table Locality**

# Summary

Database Regions

Survival Goals

Table Locality

Goal-oriented **data placement policies**

**Non-blocking** extension to transaction model

First-class region management

**Implicit table partitioning**

Auto row-level data homing

**Non-voting replicas**

Locality aware cost-based **SQL optimizations**