TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

NewSQL Database: Cockroach DB

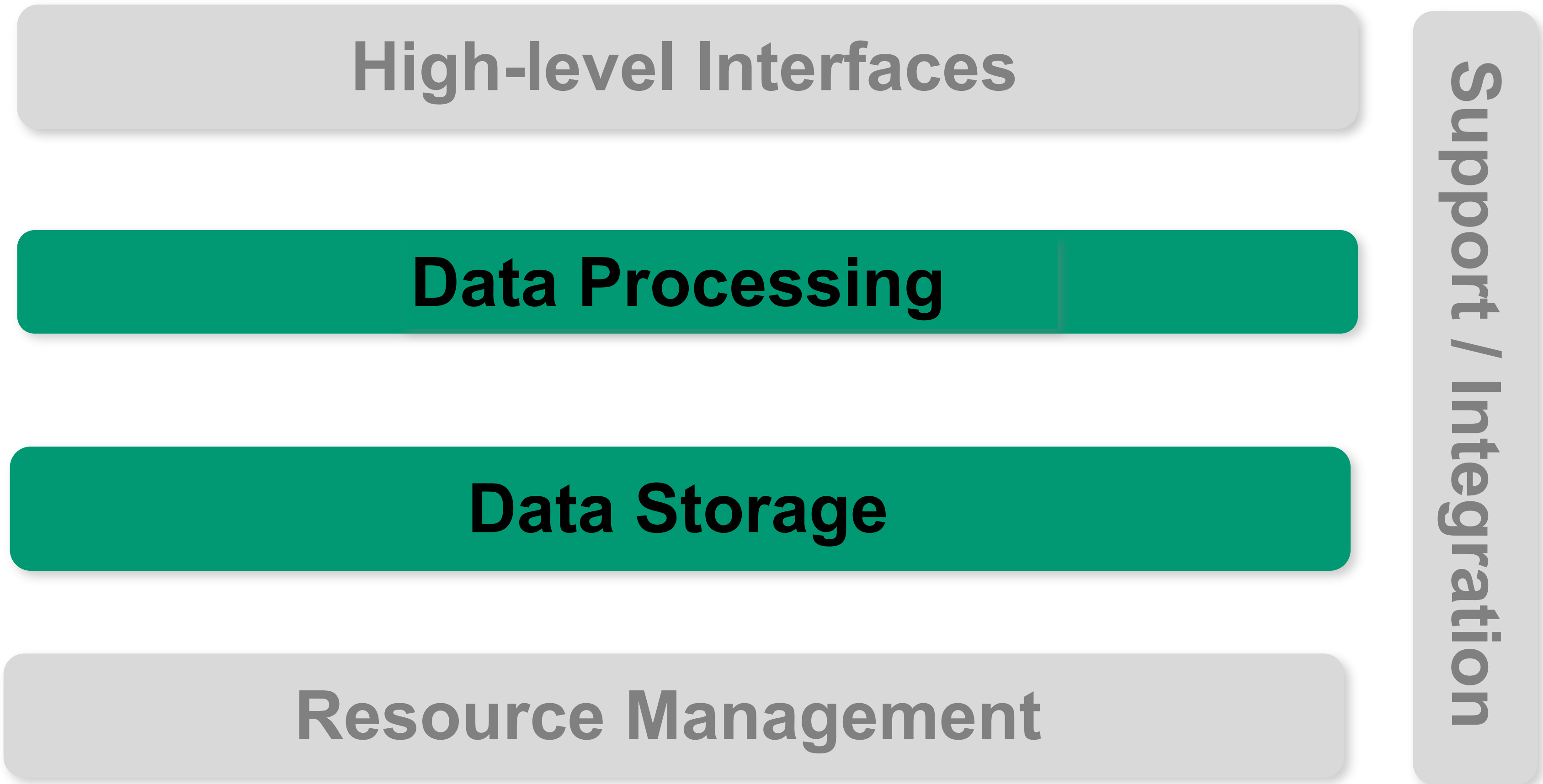
Corso di Sistemi e Architetture per Big Data

A.A. 2024/25

Matteo Nardelli

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack



Recalls on NewSQL

- RDBMS pros:
  - ACID transactions
  - Relational schemas (and schema changes without downtime)
  - SQL queries
  - Strong consistency

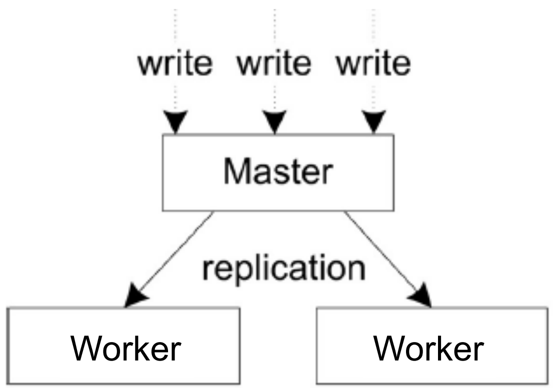
- RDBMS cons:
  - Lack of horizontal scalability (to 100s or 1000s of servers)

Recalls on NewSQL

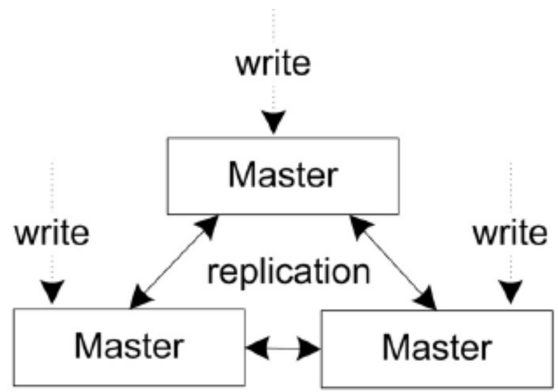
- **NewSQL**: a class of modern RDBMS
- Goals
  - Provide scalability of NoSQL systems for OLTP workloads, while maintaining ACID support of traditional RDBMS
  - Support SQL
- Examples (mostly closed source)
  - Google's Spanner
  - CockroachDB (Open-source, born as Spanner clone, then evolved differently)
  - VoltDB
  - MariaDB Xpand
  - NuoDB

Recalls on NewSQL: Replication

- Hot to scale? Multi-master (or master-less) schemes
  - Any node can receive data update statements



Primary-based replication



Multi-master replication

Cockroach DB: Overview

- NewSQL a.k.a. distributed SQL database
  - Scalability
  - Strong consistency
  - Survivability: tolerate disk, machine, rack, datacenter failures with minimal latency disruption and no manual intervention
- Multi-master architecture
  - **Each node acts as SQL gateway**:
    - Transforms and executes SQL statements to key-value (KV) operations;
    - Distributes KV operations across the cluster and returns results to the client
- CockroachDB is considered a CP system under the CAP theorem.

Read more: https://rcs.uwaterloo.ca/~ali/cs854-f23/papers/cockroachdb.pdf

Cockroach DB: Overview - Data Model

- Internal data model
  - Single, sorted map from key to value;
  - Map is divided into **ranges**;
  - Range is stored in a **local KV storage** engine (Pebble) and **replicated** to additional nodes;
    - Pebble is an embedded KV inspired by RocksDB and developed by Cockroach Labs
  - Ranges are **merged and split** to maintain target size (e.g., 64MB)

Cockroach DB: Key Features

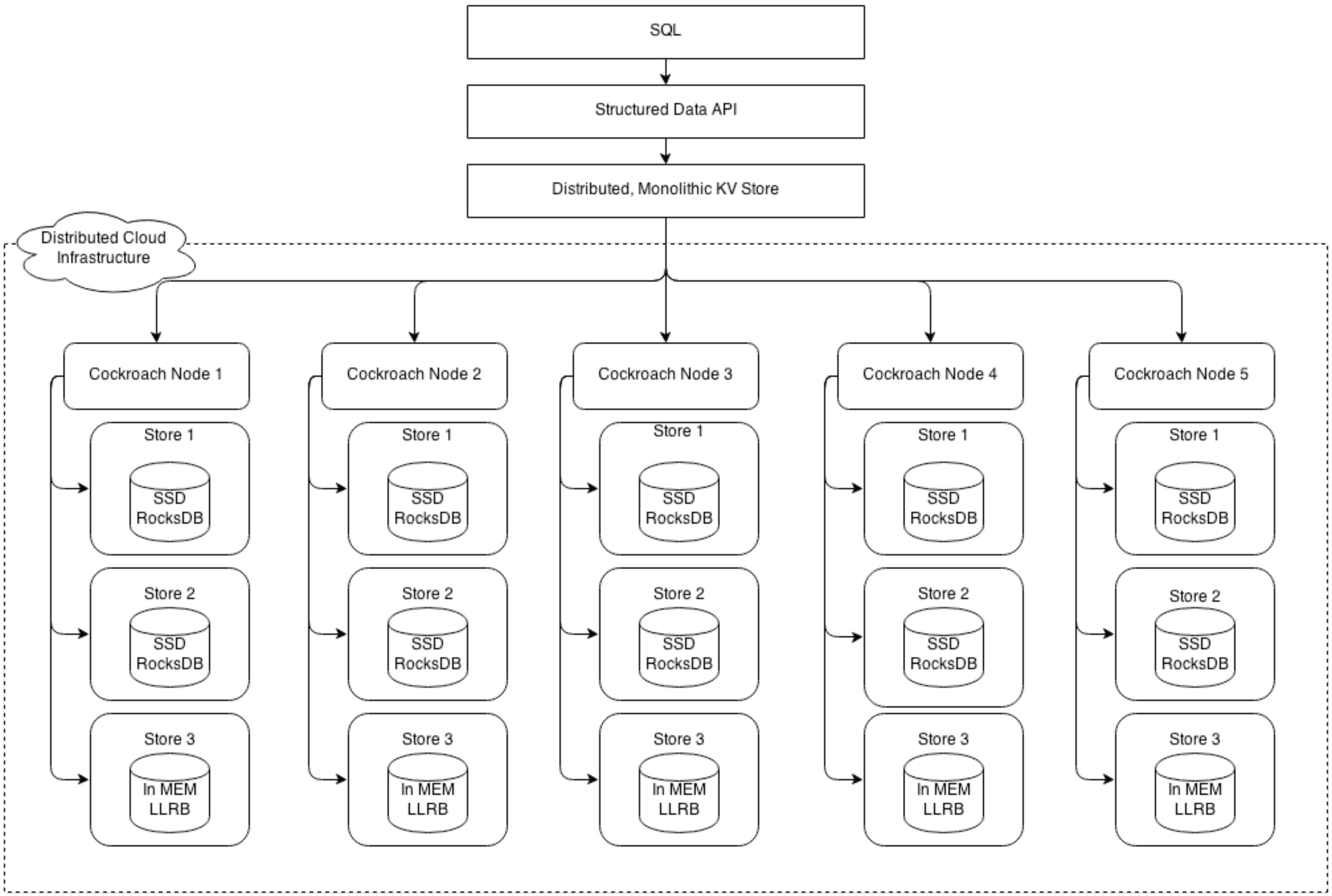
- Horizontal Scalability
  - Adding nodes increases storage capacity and overall throughput of queries;
  - Data partitioned: each node contains only part of the data (i.e., ranges);
- Fault-tolerance through data replication
  - Range replicas can be:
    - co-located within a single data center for low latency;
    - distributed across racks (survive to (some) network failures); or
    - distributed across different data centers.
- Strong consistency
  - Distributed consensus (Raft) for **synchronous replication** in each KV range;
  - **Mutations** across multiple ranges employ **distributed transactions**
  - Raft and distributed transactions guarantee **ACID** properties

Cockroach DB: Architecture Overview

- CockroachDB’s architecture is organized into layers:
  - **SQL**: translates SQL queries to KV operations;
  - **Transactional**: Allow atomic changes to multiple KV entries;
  - **Distribution**: Present replicated KV ranges as a single entity;
  - **Replication**: Consistently and synchronously replicate KV ranges across nodes;
  - **Storage**: read and write KV data on disk.

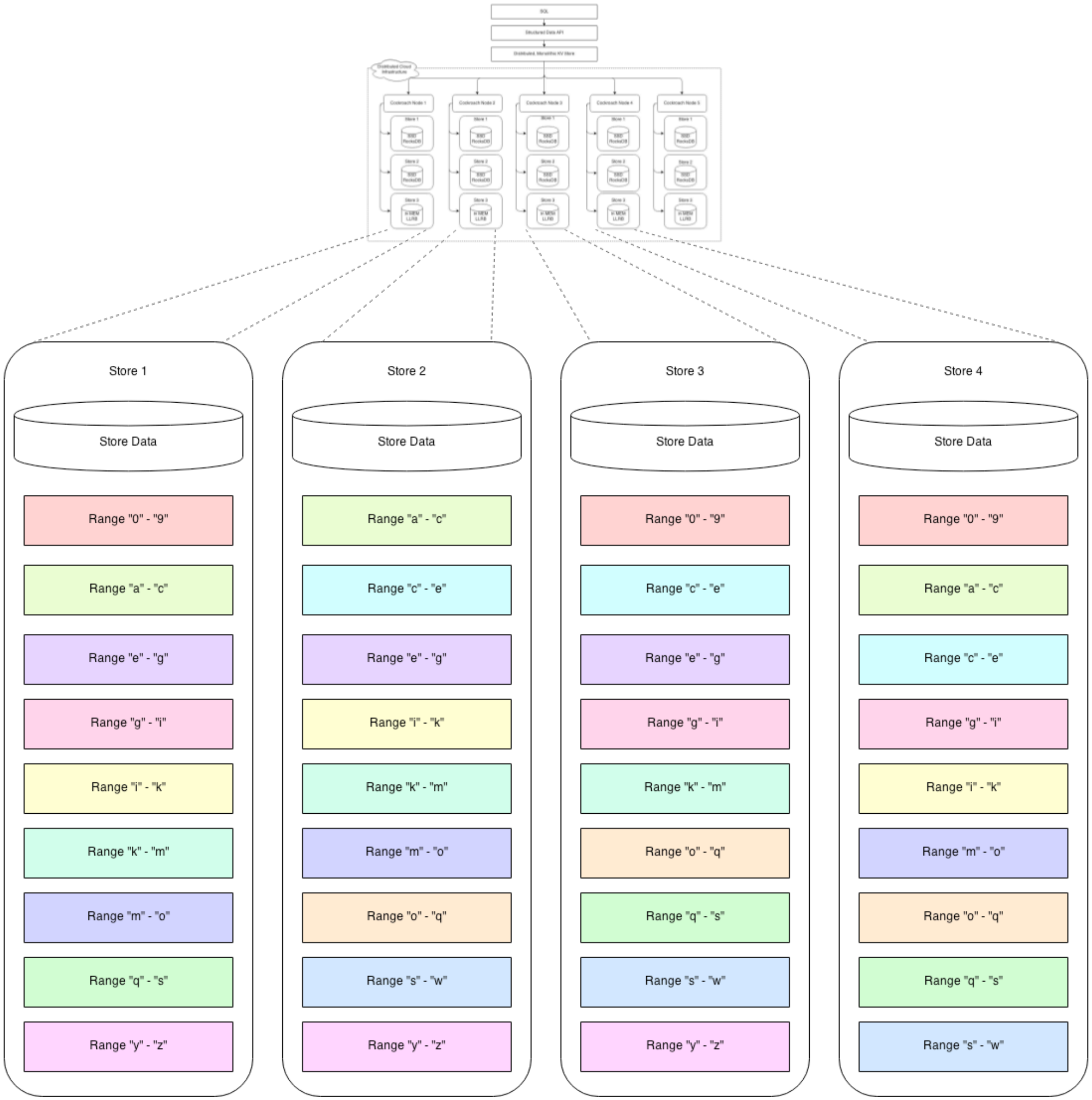
Read more: https://www.cockroachlabs.com/docs/stable/architecture/overview/

Cockroach DB: Architecture



- LLRB: Left-Leaning Red-Black tree, a special kind of self-balancing binary search tree

Cockroach DB: Architecture



Recall:
Raft is a crash fault-tolerant consensus protocol.

To tolerate F failures,
we need at least N = 2F + 1 nodes

Cockroach: Storage Layer

- Each cockroachDB node:
  - Contains one or more stores;
    - Each store should be placed on a unique disk;
    - Internally: an instance of Pebble (RocksDB);
  - All stores of the same node share a block cache;
  - Labelled with a "zone":
    - This zone concept enables to control the range's replication factor, adding constraints as to where the replicas can be located.
  - Stores gossip their descriptors periodically;
    - If a store appears to be failed, affected replicas will autonomously up-replicate themselves to other available stores to meet the replication factor
    - Self-healing property

Storage Layer: Versioned Data

- CockroachDB maintains multi-version data
  - Historical versions of values are stored with associated commit timestamps
  - Reads can specify a snapshot time to return data at a specific time;
  - Expiration interval: enables to garbage collect older versions of data;

- CockroachDB relies on **multi-version concurrency control** (MVCC)
  - To process concurrent requests and guarantee consistency
  - (Further details later)

Replication Layer

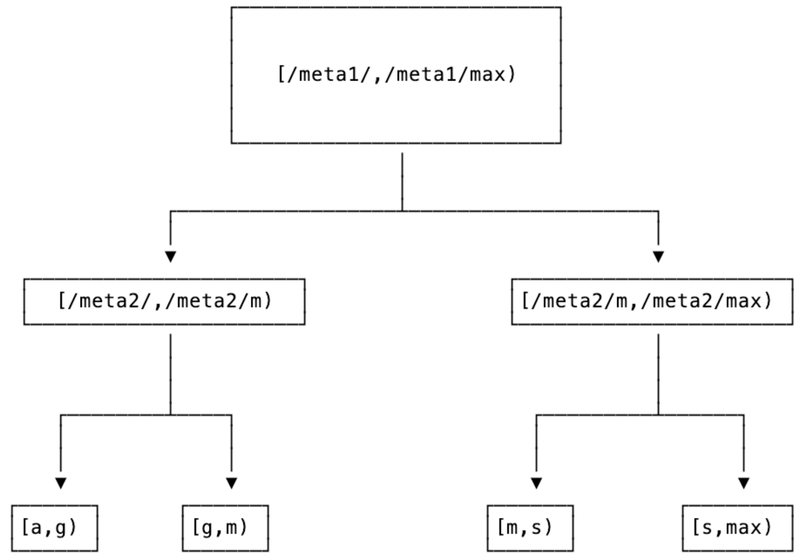
- CockroachDB uses a Raft instance at replica level
  - Non-voting replica:
    - Recently added to better support multi-region clusters;
    - Non-voting replicas follow the Raft log but do not participate in quorum;
    - Non-voting replicas can serve follower reads.
  - However, performing Raft consensus for all operations is expensive;
  - *Replica Lease:*
    - (Relatively) new concept in cockroachDB
    - A single node in the Raft group acts as the *leaseholder*, which is the only node that can serve reads or propose writes to the Raft group leader

Replication Layer

- *Replica Lease*
  - A lease held for a slice of time;
  - Lease acquisition:
    - A replica establishes itself as owning the lease on a range by committing a lease acquisition entry through Raft;
    - Soon after, the replica becomes the lease holder;
      - This guarantees that the replica has already applied all prior writes and can see them locally.
  - The replica holding the lease:
    - can satisfy reads locally (with no Raft consensus);
    - is in charge of handling range-specific maintenance (splitting, merging, rebalancing)
  - Although not mandatory, making the same node both Raft leader and leaseholder optimizes query performance

Distribution Layer

- CockroachDB stores data in a monolithic sorted map of key-value pairs; this enables:
  - *Simple lookups:* to identify nodes responsible for ranges;
  - *Efficient scans:* leveraging the order of data.



This meta range structure enables addressing up to 4EiB of user data by default

Transaction Layer: Hybrid Logical Clock (HLC)

- Each node maintains a hybrid logical clock (HLC)
- HLC allows us to track **causality** for related events
  - Similar to vector clocks, but with less overhead;
  - Incoming events inform the local HLC about the timestamp by the sender;
  - Outgoing events marked with an HLC timestamp;
- The HLC is updated by every read/write event on the node:
  - HLC timestamp consists of a physical and a logical component;
  - HLC time >= wall time;
- HLC is used to track versions of values and provide transactional isolation guarantees.

Transaction Layer: Distributed Transaction

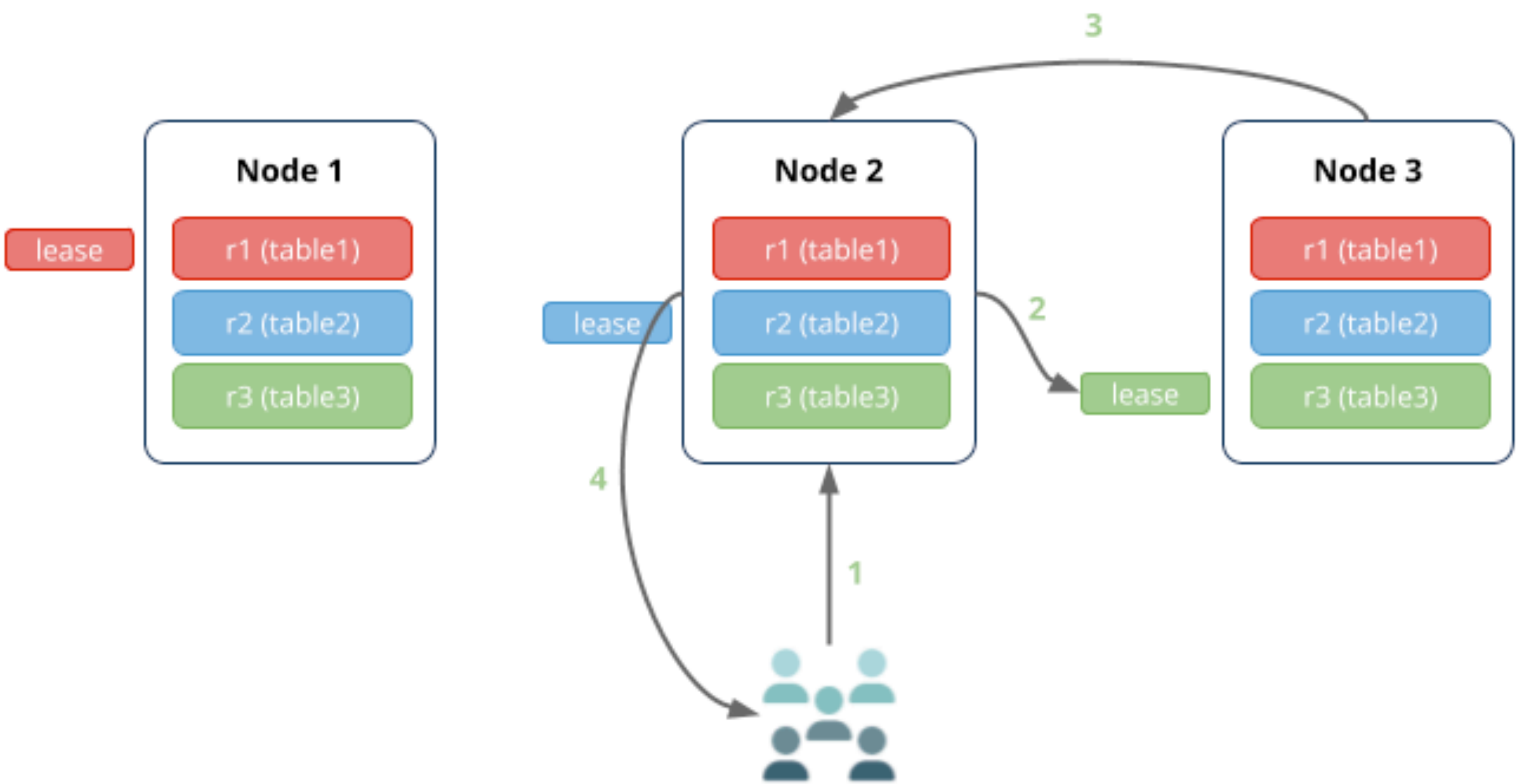
- Actors involved in a distributed transaction:
  1. SQL **Client** sends a query to the CockroachDB cluster;
     - CockroachDB keeps compatibility with Postgres clients
  2. **Load Balancing** routes the request to node in the cluster;
  3. **Gateway**: node that processes the SQL request and responds to the client;
  4. **Leaseholder**: node responsible for serving reads and coordinating writes of a specific range of keys in your query.
  5. **Raft leader**: node responsible for maintaining consensus among your CockroachDB replicas.

Read more: https://www.cockroachlabs.com/docs/stable/architecture/transaction-layer

Transaction Layer

- Read scenario

- There are 3 nodes in the cluster.
- There are 3 small tables, each fitting in a single range.
- Ranges are replicated 3 times (the default).
- A query is executed against **node 2** to read from **table 3**

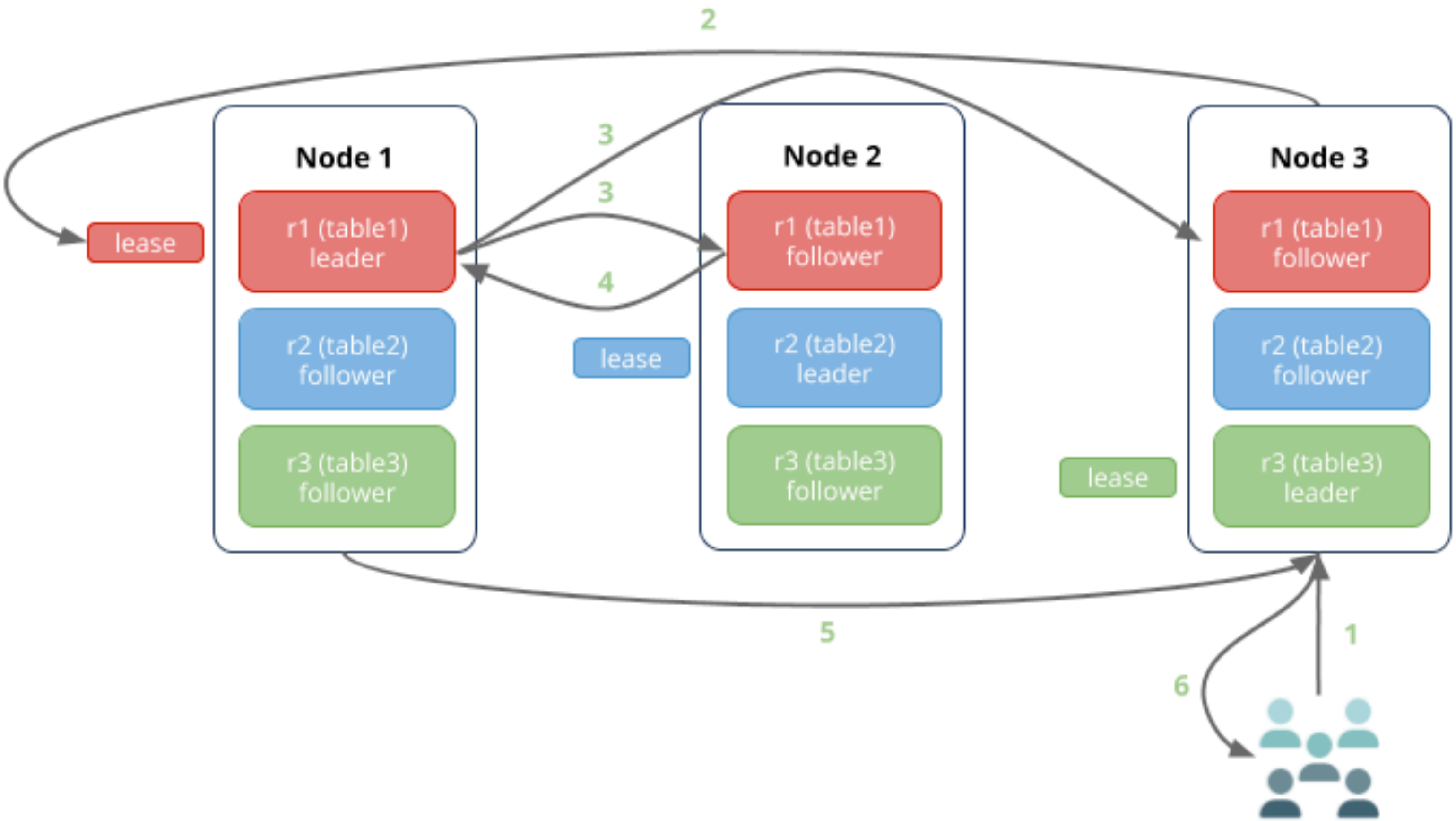


https://www.cockroachlabs.com/docs/v23.2/architecture/reads-and-writes-overview

Transaction Layer

- Write scenario

- There are 3 nodes in the cluster.
- There are 3 small tables, each fitting in a single range.
- Ranges are replicated 3 times (the default).
- A query is executed against **node 3** to write to **table 1**.



https://www.cockroachlabs.com/docs/v23.2/architecture/reads-and-writes-overview

Transaction Layer

- All statements are handled as transactions (autocommit mode);
- A write operation involves:
  - Write intents, replicated via Raft;
  - Unreplicated locks store in-memory, per-node;
  - Transaction record (with transaction current state).

- A read operation, can be of different types:
  - Strongly-consistent (default): through the leaseholder and sees all writes committed before the reading transaction (under Serializable isolation) or statement (under read committed isolation)
  - Stale reads (with AS OF SYSTEM TIME clause): from local replica;
- A read operation can optionally acquire locks:
  - Exclusive locks: block writes and locking read on a row;
    - SELECT … FOR UPDATE
  - Shared locks: block concurrent writes and exclusive locking reads on a row;
    - SELECT … FOR SHARE