Lab 10: Monitoring your CockroachDB cluster

In addition to CockroachDB's built-in safeguards against failure, it is critical to actively monitor the overall health and performance of a cluster running in production and to create alerting rules that promptly send notifications when there are events that require investigation or intervention.

Start CockroachDB

Run the `cockroach demo` command:

```
cockroach demo
```

Now, create a user with a password, which you will need to access the DB Console:

```
CREATE USER max WITH PASSWORD 'roach';
```

This starts a single-node, temporary cluster with the `movr` dataset pre-loaded.

Built-in monitoring tools

CockroachDB includes several tools to help you monitor your cluster's workloads and performance.

DB Console

The `DB Console` collects time-series cluster metrics and displays basic information about a cluster's health, such as node status, number of unavailable ranges, and queries per second and service latency across the cluster. This tool is designed to help you optimize cluster performance and troubleshoot issues. The DB Console is accessible from every node at `http://<host>:<http-port>`, or `http://<host>:8080` by default.

The DB Console automatically runs in the cluster. The following sections describe some of the pages that can help you to monitor and observe your cluster.

Metrics dashboards

The Metrics dashboards, which are located within **Metrics** in DB Console, provide information about a cluster's performance, load, and resource utilization. The Metrics dashboards are built using time-series metrics collected from the cluster. By default, metrics are collected every 10 minutes and stored within the cluster, and data is retained at 10-second granularity for 10 days , and at 30-minute granularity for 90 days.

SQL Activity pages

The SQL Activity pages, which are located within **SQL Activity** in DB Console, provide information about SQL [statements], [transactions], and [sessions].

The information on the SQL Activity pages comes from the cluster's `crdb_internal` system catalog.

Health endpoints

CockroachDB provides two HTTP endpoints for checking the health of individual nodes.

These endpoints are also available through the [Cluster API] under `/v2/health/`.

If the cluster is unavailable, these endpoints are also unavailable.

/health

If a node is down, the `http://<host>:<http-port>/health` endpoint returns a `Connection refused` error:

```
curl http://localhost:8080/health
```

```
curl: (7) Failed to connect to localhost port 8080: Connection refused
```

Otherwise, it returns an HTTP `200 OK` status response code with an empty body:

```
{

}
```

The `/health` endpoint does not returns details about the node such as its private IP address. These details could be considered privileged information in some deployments. If you need to retrieve node details, you can use the `/_status/details` endpoint along with a valid authentication cookie.

/health?ready=1

The `http://<node-host>:<http-port>/health?ready=1` endpoint returns an HTTP `503 Service Unavailable` status response code with an error in the following scenarios:

- The node is in the [wait phase of the node shutdown sequence]. This causes load balancers and connection managers to reroute traffic to other nodes before the node is drained of SQL client connections and leases, and is a necessary check during [rolling upgrades].

  Tip:

  If you find that your load balancer's health check is not always recognizing a node as unready before the node shuts down, you can increase the `server.shutdown.drain_wait` [cluster setting] to cause a node to return `503 Service Unavailable` even before it has started shutting down.

- The node is unable to communicate with a majority of the other nodes in the cluster, likely because the cluster is unavailable due to too many nodes being down.

```
curl http://localhost:8080/health?ready=1
```

```
{
  "error": "node is not healthy",
  "code": 14,
  "message": "node is not healthy",
  "details": [
  ]
}
```

Otherwise, it returns an HTTP `200 OK` status response code with an empty body:

```
{

}
```

Raw status endpoints

Several endpoints return raw status metrics in JSON at `http://<host>:<http-port>/#/debug`. Feel free to investigate and use these endpoints, but note that they are subject to change.

Prometheus endpoint

Every node of a CockroachDB cluster exports granular time-series metrics at `http://<host>:<http-port>/_status/vars`. The metrics are formatted for easy integration with [Prometheus], an open source tool for storing, aggregating, and querying time-series data. The Prometheus format is human-readable and can be processed to work with other third-party monitoring systems such as [Sysdig] and [stackdriver]. Many of the [third-party monitoring integrations], such as [Datadog] and [Kibana], collect metrics from a cluster's Prometheus endpoint.

To access the Prometheus of a cluster running on `localhost:8080`:

```
curl http://localhost:8080/_status/vars
```

```
# HELP gossip_infos_received Number of received gossip Info objects
# TYPE gossip_infos_received counter
gossip_infos_received 0
# HELP sys_cgocalls Total number of cgo calls
# TYPE sys_cgocalls gauge
sys_cgocalls 3501
# HELP sys_cpu_sys_percent Current system cpu percentage
# TYPE sys_cpu_sys_percent gauge
sys_cpu_sys_percent 1.098855319644276e-10
# HELP replicas_quiescent Number of quiesced replicas
# TYPE replicas_quiescent gauge
replicas_quiescent{store="1"} 20
...
```

Critical nodes endpoint

The critical nodes status endpoint is used to:

- Check if any of your nodes are in a critical state. A node is *critical* if that node becoming unreachable would cause [replicas to become unavailable].
- Check if any ranges are [under-replicated or unavailable]. This is useful when determining whether a node is ready for [decommissioning].
- Check if any of your cluster's data placement constraints (set via [multi-region SQL] or direct [configuration of replication zones] are being violated. This is useful when implementing [data domiciling].

Examples

- Replication status - normal
- Replication status - constraint violation
- Replication status - under-replicated ranges
- Replication status - ranges in critical localities

Replication status - normal

The following example assumes you are running a newly started, local `cockroach demo` cluster started using the following command:

Note: Make sure to run `\q` in existing `cockroach demo` shell before running next command.

```
cockroach demo --geo-partitioned-replicas --insecure
```

```
curl -X POST http://localhost:8080/_status/critical_nodes
```

```
{
  "criticalNodes": [
  ],
  "report": {
    "underReplicated": [
    ],
    "overReplicated": [
    ],
    "violatingConstraints": [
    ],
    "unavailable": [
    ],
    "unavailableNodeIds": [
    ]
  }
}
```

Note:

You may have to wait a few minutes after starting the demo cluster before getting the 'all clear' output above. This can happen because it takes time for [replica movement] to occur in order to meet the constraints given by the [zone configurations] set by the `--geo-partitioned-replicas` flag].

Replication status - constraint violation

The following example assumes you are running a newly started, local `cockroach demo` cluster started using the following command:

Note: Make sure to run `\q` in existing `cockroach demo` shell before running next command.

```
cockroach demo --geo-partitioned-replicas --insecure
```

By default, this geo-distributed demo cluster will not have any constraint violations.

To introduce a violation that you can then query for, you'll modify the zone configuration of the `movr.users` table.

You can use `SHOW CREATE TABLE` to see what existing zone configurations are attached to the `users` table, so you know what to modify.

```
SHOW CREATE TABLE users;
```

```
  table_name |                                  create_statement
-------------+-------------------------------------------------------------------------
-----------------
  users      | CREATE TABLE public.users (
             |     id UUID NOT NULL,
             |     city VARCHAR NOT NULL,
             |     name VARCHAR NULL,
             |     address VARCHAR NULL,
             |     credit_card VARCHAR NULL,
             |     CONSTRAINT users_pkey PRIMARY KEY (city ASC, id ASC)
             | ) PARTITION BY LIST (city) (
```

```
              |     PARTITION us_west VALUES IN (('seattle'), ('san francisco'), ('los
angeles')),
              |     PARTITION us_east VALUES IN (('new york'), ('boston'), ('washington
dc')),
              |     PARTITION europe_west VALUES IN (('amsterdam'), ('paris'),
('rome'))
              | );
              | ALTER PARTITION europe_west OF INDEX movr.public.users@users_pkey
CONFIGURE ZONE USING
              |     constraints = '[+region=europe-west1]';
              | ALTER PARTITION us_east OF INDEX movr.public.users@users_pkey CONFIGURE
ZONE USING
              |     constraints = '[+region=us-east1]';
              | ALTER PARTITION us_west OF INDEX movr.public.users@users_pkey CONFIGURE
ZONE USING
              |     constraints = '[+region=us-west1]'
(1 row)
```

To create a constraint violation, use the `ALTER PARTITION` statement to tell the [ranges] in the `europe_west` partition that they are explicitly supposed to *not* be in the `region=europe-west1` locality:

```
ALTER PARTITION europe_west of INDEX movr.public.users@users_pkey CONFIGURE ZONE USING
constraints = '[-region=europe-west1]';
```

Once the statement above executes, the ranges currently stored in that locality will now be in a state where they are explicitly not supposed to be in that locality, and are thus in violation of a constraint.

In other words, this tells the ranges that "where you are now is exactly where you are *not* supposed to be". This will cause the cluster to rebalance the ranges, which will take some time. During the time it takes for the rebalancing to occur, the ranges will be in violation of a constraint.

The critical nodes endpoint should now report a constraint violation in the `violatingConstraints` field of the response, similar to the one shown below.

Tip:

You can also use the `SHOW RANGES` statement to find out more information about the ranges that are in violation of constraints.

```
curl -X POST http://localhost:8080/_status/critical_nodes
```

```
{
  "criticalNodes": [
  ],
  "report": {
    "underReplicated": [
    ],
    "overReplicated": [
    ],
    "violatingConstraints": [
      {
        "rangeDescriptor": {
          "rangeId": "89",
```

```
      "startKey": "8okSYW1zdGVyZGFtAAE=",
      "endKey": "8okSYW1zdGVyZGFtAAESszMzMzMzQAD/gAD/AP8A/wD/AP8A/yMAAQ==",
      "internalReplicas": [
        {
          "nodeId": 8,
          "storeId": 8,
          "replicaId": 5,
          "type": 0
        },
        {
          "nodeId": 4,
          "storeId": 4,
          "replicaId": 7,
          "type": 0
        },
        {
          "nodeId": 3,
          "storeId": 3,
          "replicaId": 6,
          "type": 0
        }
      ],
      "nextReplicaId": 8,
      "generation": "40",
      "stickyBit": {
        "wallTime": "0",
        "logical": 0,
        "synthetic": false
      }
    },
    "config": {
      "rangeMinBytes": "134217728",
      "rangeMaxBytes": "536870912",
      "gcPolicy": {
        "ttlSeconds": 14400,
        "protectionPolicies": [
        ],
        "ignoreStrictEnforcement": false
      },
      "globalReads": false,
      "numReplicas": 3,
      "numVoters": 0,
      "constraints": [
        {
          "numReplicas": 0,
          "constraints": [
            {
              "type": 1,
              "key": "region",
              "value": "europe-west1"
            }
          ]
```

```
            }
          ],
          "voterConstraints": [
          ],
          "leasePreferences": [
          ],
          "rangefeedEnabled": false,
          "excludeDataFromBackup": false
        }
      },
      ...
    ],
    "unavailable": [
    ],
    "unavailableNodeIds": [
    ]
  }
}
```

Replication status - under-replicated ranges

The following example assumes you are running a newly started, local `cockroach demo` cluster started using the following command:

Note: Make sure to run `\q` in existing `cockroach demo` shell before running next command.

```
cockroach demo --geo-partitioned-replicas --insecure
```

By default, this geo-distributed demo cluster will not have any [under-replicated ranges].

To put the cluster into a state where some [ranges] are under-replicated, issue the following `ALTER TABLE ... CONFIGURE ZONE` statement, which tells it to store 9 copies of each range underlying the `rides` table.

```
ALTER TABLE rides CONFIGURE ZONE USING num_replicas=9;
```

Once the statement above executes, the cluster will rebalance so that it's storing 9 copies of each range underlying the `rides` table. During the time it takes for the rebalancing to occur, these ranges will be considered under-replicated, since there are not yet as many copies (9) of each range as you have just specified.

The critical nodes endpoint should now report a constraint violation in the `underReplicated` field of the response.

Tip:

You can also use the `SHOW RANGES` statement to find out more information about the under-replicated ranges.

```
curl -X POST http://localhost:8080/_status/critical_nodes
```

```
{
  "criticalNodes": [
    {
      "nodeId": 2,
      "address": {
        "networkField": "tcp",
```

```
      "addressField": "127.0.0.1:26358"
    },
    "attrs": {
      "attrs": [
      ]
    },
    "locality": {
      "tiers": [
        {
          "key": "region",
          "value": "us-east1"
        },
        {
          "key": "az",
          "value": "c"
        }
      ]
    },
    "ServerVersion": {
      "majorVal": 23,
      "minorVal": 1,
      "patch": 0,
      "internal": 0
    },
    "buildTag": "v23.1.0-rc.2",
    "startedAt": "1683655799845426000",
    "localityAddress": [
    ],
    "clusterName": "",
    "sqlAddress": {
      "networkField": "tcp",
      "addressField": "127.0.0.1:26258"
    },
    "httpAddress": {
      "networkField": "tcp",
      "addressField": "127.0.0.1:8081"
    }
  },
  ...
],
"report": {
  "underReplicated": [
    {
      "rangeDescriptor": {
        "rangeId": "76",
        "startKey": "9A==",
        "endKey": "9IkSYW1zdGVyZGFtAAE=",
        "internalReplicas": [
          {
            "nodeId": 3,
            "storeId": 3,
            "replicaId": 4,
```

```
            "type": 0
          },
          {
            "nodeId": 4,
            "storeId": 4,
            "replicaId": 2,
            "type": 0
          },
          {
            "nodeId": 8,
            "storeId": 8,
            "replicaId": 3,
            "type": 0
          }
        ],
        "nextReplicaId": 5,
        "generation": "44",
        "stickyBit": {
          "wallTime": "0",
          "logical": 0,
          "synthetic": false
        }
      },
      "config": {
        "rangeMinBytes": "134217728",
        "rangeMaxBytes": "536870912",
        "gcPolicy": {
          "ttlSeconds": 14400,
          "protectionPolicies": [
          ],
          "ignoreStrictEnforcement": false
        },
        "globalReads": false,
        "numReplicas": 9,
        "numVoters": 0,
        "constraints": [
        ],
        "voterConstraints": [
        ],
        "leasePreferences": [
        ],
        "rangefeedEnabled": false,
        "excludeDataFromBackup": false
      }
    },
    ...
  ],
  "overReplicated": [
  ],
  "violatingConstraints": [
    {
      "rangeDescriptor": {
```

```
      "rangeId": "238",
      "startKey": "9IkSYW1zdGVyZGFtAAE=",
      "endKey": "9IkSYW1zdGVyZGFtAAESxR64UeuFQAD/gAD/AP8A/wD/AP8BgQAB",
      "internalReplicas": [
        {
          "nodeId": 9,
          "storeId": 9,
          "replicaId": 5,
          "type": 0
        },
        {
          "nodeId": 7,
          "storeId": 7,
          "replicaId": 4,
          "type": 0
        },
        {
          "nodeId": 8,
          "storeId": 8,
          "replicaId": 3,
          "type": 0
        }
      ],
      "nextReplicaId": 6,
      "generation": "48",
      "stickyBit": {
        "wallTime": "0",
        "logical": 0,
        "synthetic": false
      }
    },
    "config": {
      "rangeMinBytes": "134217728",
      "rangeMaxBytes": "536870912",
      "gcPolicy": {
        "ttlSeconds": 14400,
        "protectionPolicies": [
        ],
        "ignoreStrictEnforcement": false
      },
      "globalReads": false,
      "numReplicas": 9,
      "numVoters": 0,
      "constraints": [
        {
          "numReplicas": 0,
          "constraints": [
            {
              "type": 0,
              "key": "region",
              "value": "europe-west1"
            }
```

```
              ]
            }
          ],
          "voterConstraints": [
          ],
          "leasePreferences": [
          ],
          "rangefeedEnabled": false,
          "excludeDataFromBackup": false
        }
      },
      ...
    ],
    "unavailable": [
    ],
    "unavailableNodeIds": [
    ]
  }
}
```

Replication status - ranges in critical localities

The following example assumes you are running a newly started, local `cockroach demo` cluster started using the following command:

Note: Make sure to run `\q` in existing `cockroach demo` shell before running next command.

```
cockroach demo --geo-partitioned-replicas --insecure
```

By default, this geo-distributed demo cluster will not have any [nodes] in a critical state. A node is *critical* if that node becoming unreachable would cause [replicas to become unavailable].

The status endpoint describes which of your nodes (if any) are critical via the `criticalNodes` field in the response.

To artificially put the nodes in this demo cluster in "critical" status, we can issue the following SQL statement, which uses `ALTER TABLE ... CONFIGURE ZONE` to tell the cluster to store more copies of each range underlying the `rides` table than there are nodes in the cluster.

```
ALTER TABLE rides CONFIGURE ZONE USING num_replicas=128;
```

The critical nodes endpoint should now report that all of the cluster's nodes are critical by listing them in the `criticalNodes` field of the response.

```
curl -X POST http://localhost:8080/_status/critical_nodes
```

```
{
  "criticalNodes": [
    {
      "nodeId": 4,
      "address": {
        "networkField": "tcp",
        "addressField": "127.0.0.1:26360"
      },
```

```
      "attrs": {
        "attrs": [
        ]
      },
      "locality": {
        "tiers": [
          {
            "key": "region",
            "value": "us-west1"
          },
          {
            "key": "az",
            "value": "a"
          }
        ]
      },
      "ServerVersion": {
        "majorVal": 23,
        "minorVal": 1,
        "patch": 0,
        "internal": 0
      },
      "buildTag": "v23.1.0-rc.2",
      "startedAt": "1683656700210217000",
      "localityAddress": [
      ],
      "clusterName": "",
      "sqlAddress": {
        "networkField": "tcp",
        "addressField": "127.0.0.1:26260"
      },
      "httpAddress": {
        "networkField": "tcp",
        "addressField": "127.0.0.1:8083"
      }
    },
    ...
  ],
  "report": {
    "underReplicated": [
      {
        "rangeDescriptor": {
          "rangeId": "133",
          "startKey": "9A==",
          "endKey": "9IkSYW1zdGVyZGFtAAE=",
          "internalReplicas": [
            {
              "nodeId": 1,
              "storeId": 1,
              "replicaId": 5,
              "type": 0
            },
```

```
          {
            "nodeId": 4,
            "storeId": 4,
            "replicaId": 6,
            "type": 0
          },
          {
            "nodeId": 8,
            "storeId": 8,
            "replicaId": 3,
            "type": 0
          }
        ],
        "nextReplicaId": 7,
        "generation": "52",
        "stickyBit": {
          "wallTime": "0",
          "logical": 0,
          "synthetic": false
        }
      },
      "config": {
        "rangeMinBytes": "134217728",
        "rangeMaxBytes": "536870912",
        "gcPolicy": {
          "ttlSeconds": 14400,
          "protectionPolicies": [
          ],
          "ignoreStrictEnforcement": false
        },
        "globalReads": false,
        "numReplicas": 128,
        "numVoters": 0,
        "constraints": [
        ],
        "voterConstraints": [
        ],
        "leasePreferences": [
        ],
        "rangefeedEnabled": false,
        "excludeDataFromBackup": false
      }
    },
    ...
  ],
  "overReplicated": [
  ],
  "violatingConstraints": [
    {
      "rangeDescriptor": {
        "rangeId": "186",
        "startKey": "9IkSYW1zdGVyZGFtAAE=",
```

```
      "endKey": "9IkSYW1zdGVyZGFtAAESxR64UeuFQAD/gAD/AP8A/wD/AP8BgQAB",
      "internalReplicas": [
        {
          "nodeId": 7,
          "storeId": 7,
          "replicaId": 5,
          "type": 0
        },
        {
          "nodeId": 9,
          "storeId": 9,
          "replicaId": 4,
          "type": 0
        },
        {
          "nodeId": 8,
          "storeId": 8,
          "replicaId": 3,
          "type": 0
        }
      ],
      "nextReplicaId": 6,
      "generation": "52",
      "stickyBit": {
        "wallTime": "0",
        "logical": 0,
        "synthetic": false
      }
    },
    "config": {
      "rangeMinBytes": "134217728",
      "rangeMaxBytes": "536870912",
      "gcPolicy": {
        "ttlSeconds": 14400,
        "protectionPolicies": [
        ],
        "ignoreStrictEnforcement": false
      },
      "globalReads": false,
      "numReplicas": 128,
      "numVoters": 0,
      "constraints": [
        {
          "numReplicas": 0,
          "constraints": [
            {
              "type": 0,
              "key": "region",
              "value": "europe-west1"
            }
          ]
        }
```

```
          ],
          "voterConstraints": [
          ],
          "leasePreferences": [
          ],
          "rangefeedEnabled": false,
          "excludeDataFromBackup": false
        }
      },
      ...
    ],
    "unavailable": [
    ],
    "unavailableNodeIds": [
    ]
  }
}
```