Quickstart

Docker

Binaries

Create a Cluster

Overview

Kubernetes Operator

M3DB

M3 Coordinator

M3 Query

How To Guides

Operational Guides

Integrations

FAQs

Github repo

Slack

Mailing list

Versions 1.0

Creating a Single Node M3 Cluster with Binaries

This guide shows how to install and configure M3, create a single-node cluster, and read and write metrics to it.

Warning

Deploying a single-node M3 cluster is a great way to experiment with M3 and get an idea of what it has to offer, but is not designed for production use. To run M3 in clustered mode, with a separate M3Coordinator read the clustered mode guide.

Prebuilt Binaries

M3 has pre-built binaries available for Linux and macOS. Download the latest release from GitHub.

Build From Source

Prerequisites

- Go 1.10 or higher
- Make

Build Source

```
make m3dbnode
```

Start Binary

By default the binary configures a single M3 instance containing:

- An M3DB storage instance for time series storage. It includes an embedded tag-based metrics index and an etcd server for storing the cluster topology and runtime configuration.
- An M3Coordinator instance for writing and querying tagged metrics, as well as managing cluster topology and runtime configuration.

It exposes three ports:

- `7201` to manage the cluster topology, you make most API calls to this endpoint
- `7203` for Prometheus to scrape the metrics produced by M3DB and M3Coordinator

The command below starts the node using the specified configuration file.

Pre-built binary | Self-built binary | Output

Download the example configuration file.

```
./m3dbnode -f /{FILE_LOCATION}/m3dbnode-local-etcd.yml
```

Info

Depending on your operating system setup, you might need to prefix the command with `sudo`.

Info

When running the command above on macOS you may see errors about “too many open files.” To fix this in your current terminal, use `ulimit` to increase the upper limit, for example `ulimit -n 10240`.

Configuration

This example uses this sample configuration file by default.

The file groups configuration into `coordinator` or `db` sections that represent the `M3Coordinator` and `M3DB` instances of single-node cluster.

Tip

You can find more information on configuring M3DB in the operational guides section.

Organizing Data with Placements and Namespaces

A time series database (TSDBs) typically consist of one node (or instance) to store metrics data. This setup is simple to use but has issues with scalability over time as the quantity of metrics data written and read increases.

As a distributed TSDB, M3 helps solve this problem by spreading metrics data, and demand for that data, across multiple nodes in a cluster. M3 does this by splitting data into segments that match certain criteria (such as above a certain value) across nodes into shards.

If you've worked with a distributed database before, then these concepts are probably familiar to you, but M3 uses different terminology to represent some concepts.

- Every cluster has **one** placement that maps shards to nodes in the cluster.
- A cluster can have **0 or more** namespaces that are similar conceptually to tables in other databases, and each node serves every namespace for the shards it owns.

For example, if the cluster placement states that node A owns shards 1, 2, and 3, then node A owns shards 1, 2, 3 for all configured namespaces in the cluster. Each namespace has its own configuration options, including a name and retention time for the data.

Create a Placement and Namespace

This quickstart uses the *http://localhost:7201/api/v1/database/create* endpoint that creates a namespace, and the placement if it doesn't already exist based on the `type` argument.

You can create placements and namespaces separately if you need more control over their settings.

In another terminal, use the following command.

Command | Output

```
#!/bin/bash
curl -X POST http://localhost:7201/api/v1/database/create -d '{
  "type": "local",
  "namespaceName": "default",
  "retentionTime": "12h"
}' | jq .
```

Placement initialization can take a minute or two. Once all the shards have the `AVAILABLE` state, the node has finished bootstrapping, and you should see the following messages in the node console output.

```
{"level":"info","ts":1598367624.0117292,"msg":"bootstrap marking all shards as
bootstrapped","namespace":"default","namespace":"default","numShards":64}
{"level":"info","ts":1598367624.0301404,"msg":"bootstrap index with bootstrapped
index segments","namespace":"default","numIndexBlocks":0}
{"level":"info","ts":1598367624.0301914,"msg":"bootstrap
success","numShards":64,"bootstrapDuration":0.049208827}
{"level":"info","ts":1598367624.03023,"msg":"bootstrapped"}
```

You can check on the status by calling the *http://localhost:7201/api/v1/services/m3db/placement* endpoint:

Command Output

```
curl http://localhost:7201/api/v1/services/m3db/placement | jq .
```

Tip

Read more about the bootstrapping process.

Ready a Namespace

Once a namespace has finished bootstrapping, you must mark it as ready before receiving traffic by using the *http://localhost:7201/api/v1/services/m3db/namespace/ready*.

Command Output

```
#!/bin/bash
curl -X POST http://localhost:7201/api/v1/services/m3db/namespace/ready -
d '{
  "name": "default"
}' | jq .
```

View Details of a Namespace

You can also view the attributes of all namespaces by calling the *http://localhost:7201/api/v1/services/m3db/namespace* endpoint

Command Output

```
curl http://localhost:7201/api/v1/services/m3db/namespace | jq .
```

Tip

Add `?debug=1` to the request to convert nano units in the output into standard units.

Writing and Querying Metrics

Writing Metrics

M3 supports ingesting statsd and Prometheus formatted metrics.

This quickstart focuses on Prometheus metrics which consist of a value, a timestamp, and tags to bring context and meaning to the metric.

You can write metrics using one of two endpoints:

- *http://localhost:7201/api/v1/prom/remote/write* - Write a Prometheus remote write query to M3DB with a binary snappy compressed Prometheus WriteRequest protobuf message.
- *http://localhost:7201/api/v1/json/write* - Write a JSON payload of metrics data. This endpoint is quick for testing purposes but is not as performant for production usage.

Prometheus | HTTP API

You can use the *http://localhost:7201/api/v1/json/write* endpoint to write a tagged metric to M3 with the following data in the request body, all fields are required:

- `tags`: An object of at least one `name`/`value` pairs
- `timestamp`: The UNIX timestamp for the data
- `value`: The float64 value for the data

Tip

The examples below use `__name__` as the name for one of the tags, which is a Prometheus reserved tag that allows you to query metrics using the value of the tag to filter results.

Tip

Label names may contain ASCII letters, numbers, underscores, and Unicode characters. They must match the regex `[a-zA-Z_][a-zA-Z0-9_]*`. Label names beginning with `__` are reserved for internal use. Read more in the Prometheus documentation.

Command 1 | Command 2 | Command 3

```
#!/bin/bash
curl -X POST http://localhost:7201/api/v1/json/write -d '{
  "tags":
    {
      "__name__": "third_avenue",
      "city": "new_york",
      "checkout": "1"
    },
    "timestamp": '\"$(date "+%s")\"',
    "value": 3347.26
}'
```

Querying metrics

M3 supports three query engines: Prometheus (default), Graphite, and the M3 Query Engine.

This quickstart uses Prometheus as the query engine, and you have access to all the features of PromQL queries.

To query metrics, use the *http://localhost:7201/api/v1/query_range* endpoint with the following data in the request body, all fields are required:

- `query`: A PromQL query
- `start`: Timestamp in `RFC3339Nano` of start range for results
- `end`: Timestamp in `RFC3339Nano` of end range for results
- `step`: A duration or float of the query resolution, the interval between results in the timespan between `start` and `end`.

Below are some examples using the metrics written above.

Return results in past 45 seconds

Linux | macOS/BSD | Output

```
curl -X "POST" -G "http://localhost:7201/api/v1/query_range" \
  -d "query=third_avenue" \
  -d "start=$(date "+%s" -d "45 seconds ago")" \
  -d "end=$( date +%s )" \
  -d "step=5s" | jq .
```

Values above a certain number

Linux | macOS/BSD | Output

```
curl -X "POST" -G "http://localhost:7201/api/v1/query_range" \
  -d "query=third_avenue > 6000" \
  -d "start=$(date "+%s" -d "45 seconds ago")" \
  -d "end=$( date +%s )" \
  -d "step=5s" | jq .
```

Values collected from Prometheus

If you followed the steps above for collecting metrics from Prometheus, the examples above work, but don't return any results. To query those results, use the following commands to return a sum of the values.

Linux | macOS/BSD | Output

```
curl -X "POST" -G "http://localhost:7201/api/v1/query_range" \
  -d "query=third_avenue_sum" \
  -d "start=$(date "+%s" -d "45 seconds ago")" \
  -d "end=$( date +%s )" \
  -d "step=500s" | jq .
```

Was this page helpful?

- Suggest edits to this page