Schema Design

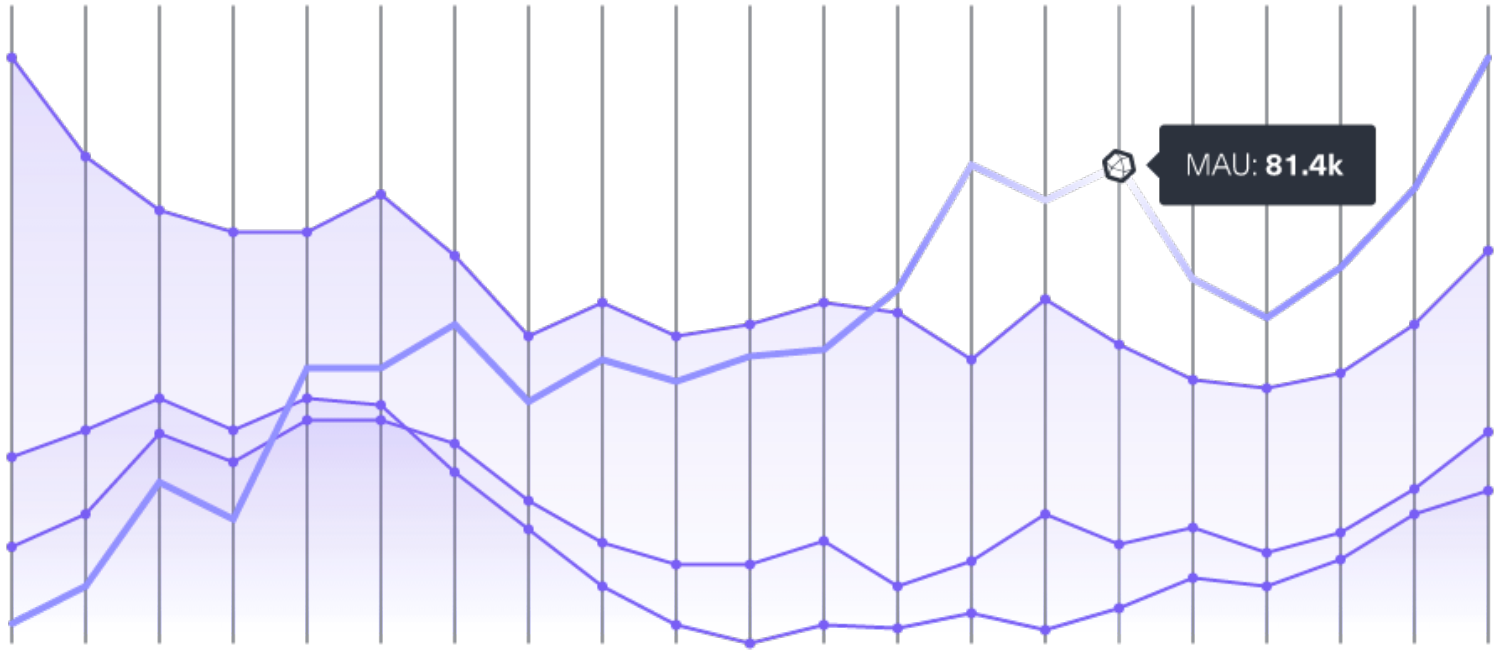
Michael Desa & Jack Zampolin

By the end of this section, participants will be able to...

1. Identify poorly designed schemas.
2. Design a basic schema for a common use case and query it efficiently.
3. Explain what a continuous query is and what they are used for.
4. Create their own continuous queries.
5. Describe what a retention policy is and its relation to databases and series
6. Create their own retention policies.
7. Combine retention policies and continuous queries in novel ways to manage their data's lifecycle.

influxdata

Important Things to Remember



- Tags are Indexed
- Fields are not
- All points are indexed by time
- You want to minimize the number of unique series

influxdata

What *not* to do

influxdata

DON'T ENCODE DATA INTO THE MEASUREMENT NAME

Measurement names like:

```
cpu.server-5.us-west value=2 1444234982000000000

cpu.server-6.us-west value=4 1444234982000000000

mem-free.server-6.us-west value=2500 1444234982000000000
```

Encode that information as tags:

```
cpu,host=server-5,region=us-west value=2 1444234982000000000

cpu,host=server-6,region=us-west value=4 1444234982000000000

mem-free,host=server-6,region=us-west value=2500 1444234982000000
```

What if my plugin sends data like that to InfluxDB?

Write something that sits between your plugin and InfluxDB to sanitize the data OR use one of our write plugins:

Example - Graphite Plugin:

Takes input like…

```
sensu.metric.net.server0.eth0.rx_packets 461295119435 1444234982
sensu.metric.net.server0.eth0.tx_bytes 1093086493388480 1444234982
sensu.metric.net.server0.eth0.rx_bytes 1015633926034834 1444234982
```

…and parses it with the following template…

```
["sensu.metric.* ..measurement.host.interface.field"]
```

…resulting in the following points in line protocol hitting the database:

```
net,host=server0,interface=eth0 rx_packets=461295119435 1444234982
net,host=server0,interface=eth0 tx_bytes=1093086493388480 1444234982
net,host=server0,interface=eth0 rx_bytes=1015633926034834 1444234982
```

DON'T OVERLOAD TAGS

BAD

```
cpu,server=localhost.us-west value=2 1444234982000000000

cpu,server=localhost.us-east value=3 1444234982000000000
```

GOOD

Separate out into different tags:

```
cpu,host=localhost,region=us-west value=2 1444234982000000000

cpu,host=localhost,region=us-east value=3 1444234982000000000
```

DON'T USE TAGS THAT HAVE HIGH VARIABILITY, E.G. UUIDS, HASHES, RANDOM STRINGS

BAD

```
response_time,session_id=33254331,request_id=3424347 value=340 14442349820000
```

You might need to do something like that, so consider:

- Vertical sharding across instances
- Use prefix tag groupings
- Use a cluster to horizontally shard series

DON'T HAVE A HUGE NUMBER OF INDEPENDENT TAGS

BAD

```
cpu,week=10,weekday=tues,hour=14,min=34,...,host=api-0 value=2 1444234982000
```

This increases the memory requirement of InfluxDB.

DON'T USE THE SAME NAME FOR A FIELD AND A TAG

BAD

This leads to unqueryable data:

```
login,user=admin user=2342,success=1 1444234982000
```

GOOD

Differentiate the names somehow:

```
login,user_type=admin user_id=2342,success=1 1444234982000
```

DON'T USE TOO FEW TAGS

BAD

```
cpu,region=us-west host="server1",value=0.5 1444234986000
cpu,region=us-west host="server2",value=4 1444234982000
cpu,region=us-west host="server2",value=1 1444234982000
```

Some problems you might run into:

- If points in the same series have the same timestamp, the last one will overwrite the ones before (LWW, last write wins).
- Not indexed means inefficient queries - will have to scan through all points to get to ones with specific field.
- Won't be able to GROUP BY host

DON'T WRITE DATA WITH THE WRONG PRECISION

BAD

Writing data using second precision when you need millisecond precision

- Timestamps will collide and you'll lose data.
- The database might think its 1970.
- The database might think its 2185.

BAD

Writing data using nanosecond precision when you need second precision

- More data over the wire.
- Decreased write throughput.
- Larger size on disk.

What should you do?

There's no single answer. Every InfluxDB use-case is a special flower.

However...

influxdata

HERE'S SOME GENERAL THINGS TO KEEP IN MIND

ASK

- What kind of queries do I want to run?
- Do I want to look things up by a particular value?
- Do I lose information by storing a value as a string?

GUIDELINES

- Anything in a GROUP BY clause must be a tag.
- Anything that you want to pass into a function must be a field.
- Few measurements with many tags is better than many measurements with few tags.
- If you lose information by storing as a string, use a field.

Designing a Run Tracking App

influxdata

THE APP SENDS BACK THE FOLLOWING INFORMATION EACH SECOND OF A RUN

- `user_id` Which user is running
- `run_id` Which run is being tracked (a unique identifier)
- `distance` The distance the runner had run in the last second
- `heart_rate` What the user's average heart rate was over the last second
- `speed` What the user's average speed was over the last second

THE APP REPORTS TO INFLUXDB EVERY TEN SECONDS

We only care about minute resolution.

Keep this in mind as we go forward.

SOME INFORMATION WE'D LIKE TO GET OUT OF THE DATA:

- What was the user's average speed during a run?
- How far did the user run?
- What was the user's average heart rate during a run?
- What was the users maximum heart rate during a run?

influxdata

QUESTION

How can we organize our data so that we can easily get the results that we want from the database?

Exercise

Why would it be a bad idea to make `distance`, `heart_rate`, or `speed` a tag instead of a field?

OPTION 1

Encode all of the information into the measurement name.

measurement(s):

`user_1.run_1`

`user_1.run_2`

`user_2.run_3`

tags:

fields:

`distance`

`speed`

`heart_rate`

OPTION 2

Everything as fields

measurement:

`run_stats`

tags:

fields:

`user_id`

`run_id`

`distance`

`speed`

`heart_rate`

OPTION 3

Use a single measurement, 2 tags, and the rest fields.

measurement:

`run_stats`

tags:

`user_id`

`run_id`

fields:

`distance`

`speed`

`heart_rate`

Exercise

Given that `run_id` is unique, does the inclusion of `user_id` as a tag increase the series cardinality?

Solution

Given that `run_id` is unique, does the inclusion of `user_id` as a tag increase the series cardinality?

No. Since each `run_id` has a unique user, having the tag `user_id` doesn't increase the number of unique series.

Schema for Run app

measurement:

`run_stats`

tags:

`user_id` `run_id`

fields:

`distance` `speed` `heart_rate`

Examples in LP

```
run_stats,user_id=1,run_id=3 distance=0.1,speed=10.1,heart_rate=170i 142309324834700

run_stats,user_id=1,run_id=3 distance=0.09,speed=9.3,heart_rate=150i 142309324834701

run_stats,user_id=1,run_id=3 distance=0.06,speed=7.4,heart_rate=140i 142309324834702
```

Information we'd like to get out of the data...

- What was the users average speed during a run?
- How far did the user run?
- What was the user's average heart rate during a run?
- What was the users maximum heart rate during a run?

REMINDER

We receive information with 10 second granularity.

We only care about 1 minute granularity.

Let’s create a database for the app

```
CREATE DATABASE runner
```

Exercise

In the last hour, what was the average speed for the run with `run_id=3` for each minute?

Solution

In the last hour, what was the average speed for the run with `run_id=3` for each minute?

```
SELECT mean(speed)
 FROM run_stats
 WHERE time > now() - 1h AND run_id='3'
 GROUP BY time(60s)
```

Exercise

In the last hour, what was the total distance per minute for the run with `run_id=3`?

Solution

In the last hour, what was the total distance per minute for the run with `run_id=3`?

```
SELECT sum(distance)

  FROM run_stats

  WHERE time > now() - 1h AND run_id='3'

  GROUP BY time(60s)
```

Exercise

In the last hour, what was the average `heart_rate` per minute for the run with `run_id=3`?

Solution

In the last hour, what was the average `heart_rate` per minute for the run with `run_id=3`?

```
SELECT mean(heart_rate)
  FROM run_stats
  WHERE time > now() - 1h AND run_id='3'
  GROUP BY time(60s)
```

Exercise

In the last hour, what was the max `heart_rate` per minute for the run with `run_id=3`?

Solution

In the last hour, what was the max `heart_rate` per minute for the run with `run_id=3`?

```
SELECT max(heart_rate)
    FROM run_stats
    WHERE time > now() - 1h AND run_id='3'
    GROUP BY time(60s)
```

Put it all together...

Combine the last 4 queries into a single query:

```
SELECT
  mean(speed) as speed,
  sum(distance) as distance,
  mean(heart_rate) as heart_rate,
  max(heart_rate) as max_hr
  FROM run_stats WHERE time > now() - 1h AND run_id='3'
  GROUP BY time(60s)
```

This is only for a specific run ID

We want it for each `run_id` in the time range.

Exercise

So let's group by `run_id` and `user_id`

```
SELECT
  mean(speed) as speed,
  sum(distance) as distance,
  mean(heart_rate) as heart_rate,
  max(heart_rate) as max_hr
  FROM run_stats WHERE time > now() - 1h
  GROUP BY time(60s), run_id, user_id
```

Questions

- What do we do with the results of the query?
- When do we run the query to downsample all of our data?

Continuous queries

Continuous queries are queries that will periodically run on data in InfluxDB.

They're somewhat similar to running queries with `cron`.

They're used to "downsample" data or pre-calculate common queries.

FORMAT OF A CONTINUOUS QUERY

```
CREATE CONTINUOUS QUERY [name_of_continuous_query]
 ON [name_of_db]
 [RESAMPLE [EVERY interval] [FOR interval]]
 BEGIN
  SELECT [inner_part_of_select]
    INTO [new_measurement]
    FROM [measurement]
    GROUP BY time([frequency]), [tags]
END
```

- The EVERY clause specifies how frequently the CQ will run.
- The FOR clause specifies how far back the CQ resamples.

Exercise

Turn the query we wrote earlier into a CQ:

```
SELECT
 mean(speed) as speed,
 sum(distance) as distance,
 mean(heart_rate) as heart_rate,
 max(heart_rate) as max_hr
 FROM run_stats WHERE time > now() - 1h
 GROUP BY time(60s), run_id, user_id
```

Solution

```
CREATE CONTINUOUS QUERY reduce_reso ON runner BEGIN
  SELECT
  mean(speed) as speed,
  sum(distance) as distance,
  mean(heart_rate) as heart_rate,
  max(heart_rate) as max_hr
  INTO reduced_run_stats
  FROM run_stats
  GROUP BY time(60s), run_id, user_id
END
```

Verify that the CQ was created:

```
SHOW CONTINUOUS QUERIES
```

NOW THAT WE'VE DOWNSAMPLED OUR DATA...

What do we do with our old data?

RETENTION POLICIES

- A retention policy describes how long the data should be stored in the database. (`DURATION`)
- A retention policy belongs to a database.

Creating a Retention Policy

```
CREATE RETENTION POLICY [name_of_policy]
 ON [name_of_database]
  DURATION [time_duration]
  REPLICATION [number] [DEFAULT]
```

Exercise

Create a retention policy for 1 hour with a replication of 1 that is the default retention policy for the database.

Solution

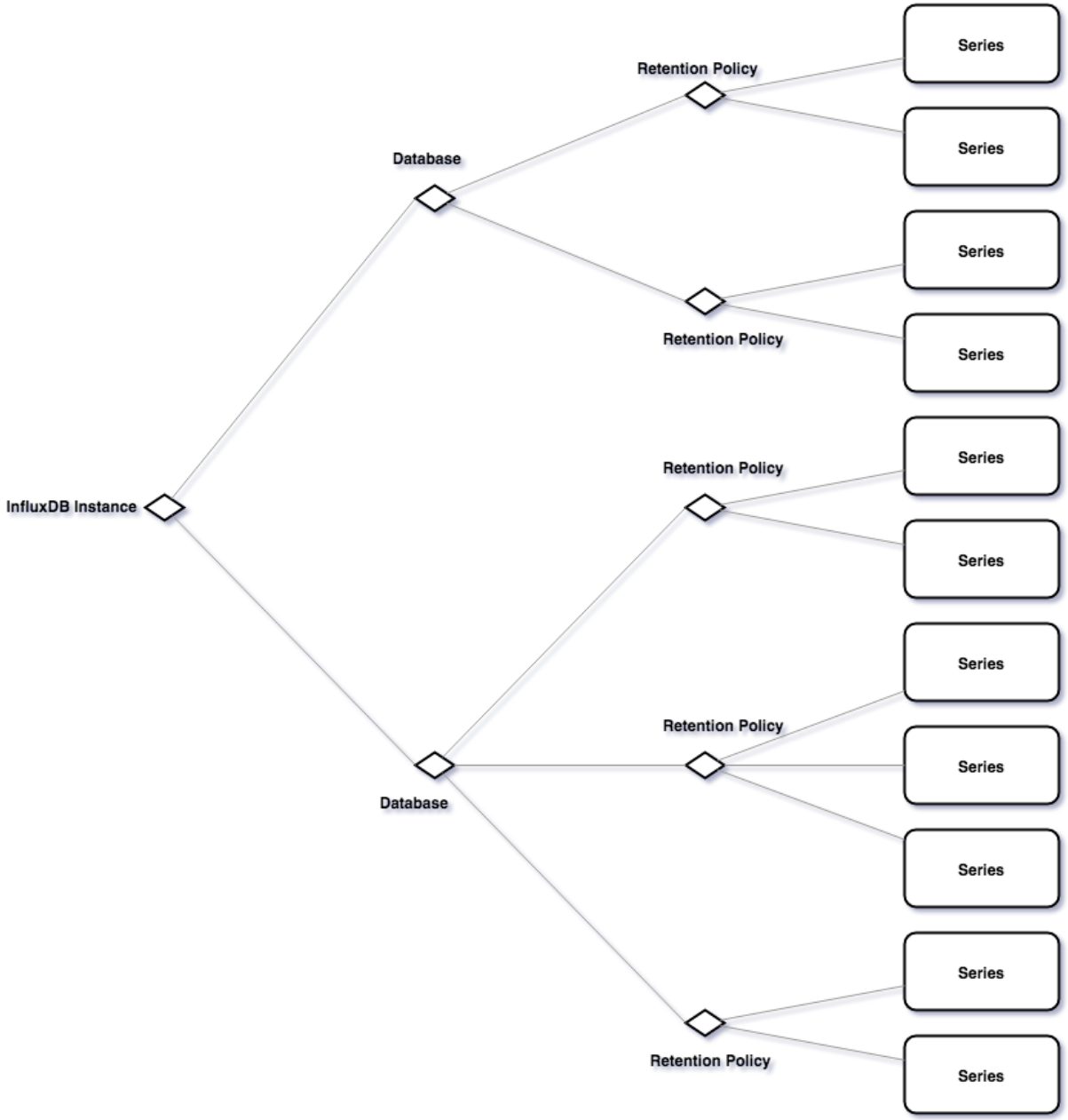
Create a retention policy for 1 hour with a replication of 1 that is the default retention policy for the database.

```
CREATE RETENTION POLICY one_hour ON runner
DURATION 1h REPLICATION 1 DEFAULT
```

Verify that the retention policy was created with:

```
SHOW RETENTION POLICIES ON runner
```

A Full Picture



influxdata

A fully qualified measurement

Measurements in InfluxDB are fully qualified by their database and retention policy.

```
"database"."retention_policy"."measurement"
```

Exercise

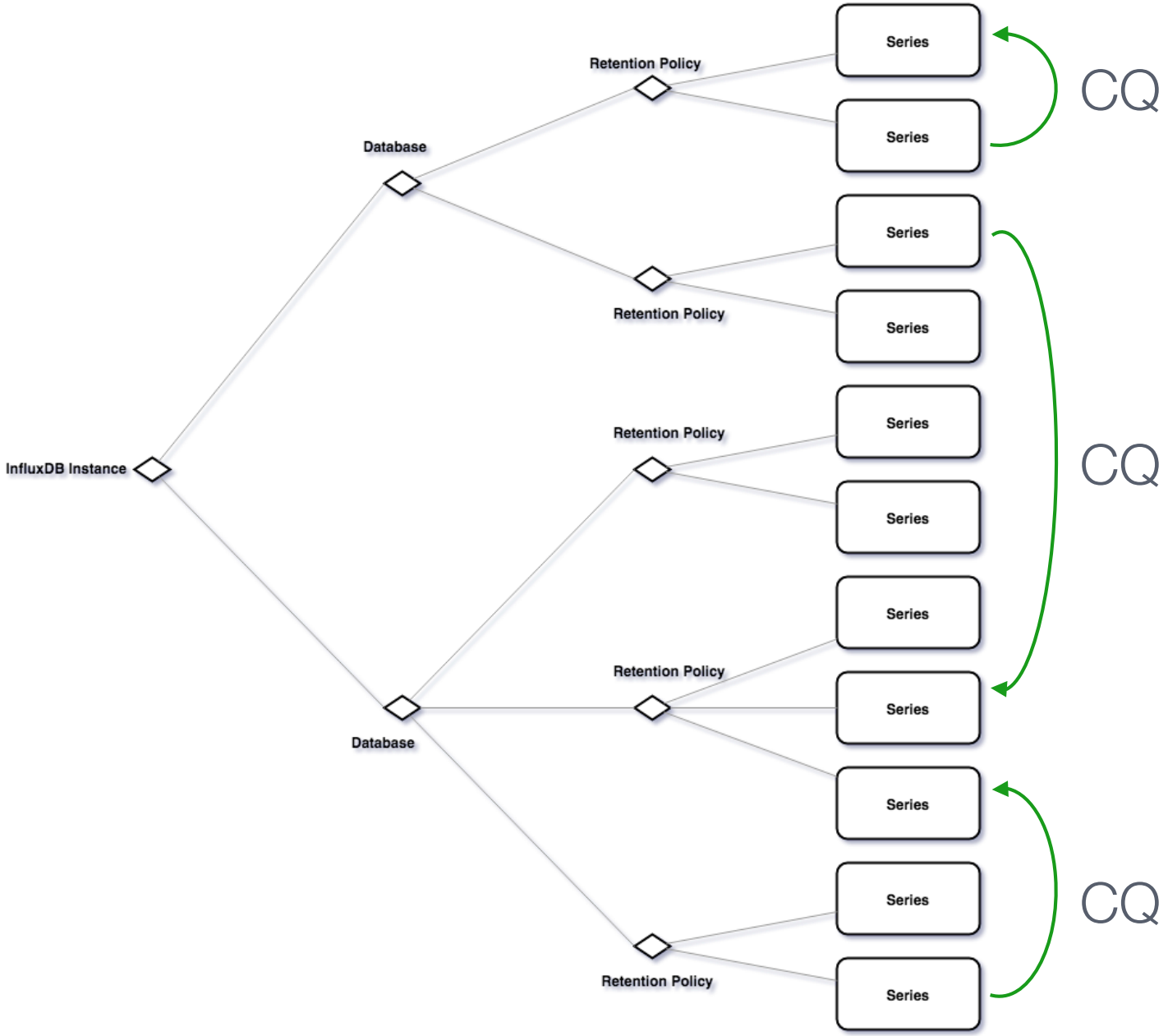
Modify the continuous query below so that the data from `run_stats` gets downsampled into a measurement in the `"default"` retention policy.

```
CREATE CONTINUOUS QUERY reduce_reso ON runner BEGIN
  SELECT
  mean(speed) as speed,
  sum(distance) as distance,
  mean(heart_rate) as heart_rate,
  max(heart_rate) as max_hr
  INTO reduced_run_stats
  FROM run_stats
  GROUP BY time(60s), run_id, user_id
```

Solution

```
CREATE CONTINUOUS QUERY reduce_reso ON runner BEGIN
  SELECT
  mean(speed) as speed,
  sum(distance) as distance,
  mean(heart_rate) as heart_rate,
  max(heart_rate) as max_hr
  INTO "runner"."default"."reduced_run_stats"
  FROM "runner"."one_hour"."run_stats"
  GROUP BY time(60s), run_id, user_id
END
```

A Complete Picture



influxdata

Load testing your system

influxdata

influx_stress -v2

- A utility that generates synthetic load on InfuxDB and comes bundled with every installation (written by Michael and Jack)
- Requires a working InfluxDB installation at localhost.
- This instance will be used for recording metrics as well as providing a target for the stress test.
- Many configuration options to increase load or present different load profiles
- Extensible syntax useful for testing different schemas from a write as well as a query perspective
- Useful for setting up CQ & RP environments

To run:

```
influx_stress -v2 -config file.iql
```

influxdata

Sample Config

$ cat file.iql

```
CREATE DATABASE IF NOT EXISTS thing

CREATE RETENTION POLICY ON thing DURATION 1h

SET Database [thing]

SET Precision [s]

GO INSERT devices
devices,
city=[str rand(8) 100],country=[str rand(8) 25],device_id=[str rand(10) 100]
lat=[float rand(90) 0],lng=[float rand(120) 0],temp=[float rand(40) 0]
100000 10s

GO QUERY devices
SELECT count(%f) FROM %m WHERE %t
DO 100

WAIT

DROP DATABASE thing
```