Telegraf Split Data into different DBs

explain me this regex: contrude/(?P<ship>[^/]+)/(?P<container>[^/]+)/(?P<sensor>[^/]+)

This regular expression is designed to match a URL or path structure containing three segments after `contrude`. Here's how it works:

Regex Breakdown

- `contrude/`: Matches the literal string `contrude/`.
- `(?P<ship>[^/]+)`: Captures a sequence of one or more characters that are **not** a `/` character (meaning it matches up to the next `/`). The `(?P<ship> ...)` syntax gives this segment the name **ship**.
- `/(?P<container>[^/]+)`: Similarly, captures a sequence of characters that are **not** `/`, assigns it the name **container**, and matches up to the next `/`.
- `/(?P<sensor>[^/]+)`: Finally, captures another sequence of characters that are **not** `/`, assigns it the name **sensor**, and captures until the end of this segment.

Example Match

If the path is `contrude/alpha123/cont567/sens89`, this regex would parse it as:

- `ship` = `"alpha123"`
- `container` = `"cont567"`
- `sensor` = `"sens89"`

Applications

This regex could be useful for parsing URL segments or path strings that contain dynamic elements you want to capture and label for further use, such as in a web application routing system.

write me a regex that excepts this string:
contrude/n/n/sensor

n: can be any whole number / integer
sensor: can be d

To match the string format `contrude/n/n/sensor` where `n` is any whole number (integer) and `sensor` can be exactly `d`, here's the regex:

```
regex

contrude/\d+/\d+/d
```

Explanation

- `contrude/`: Matches the literal string `contrude/`.
- `\d+`: Matches one or more digits, representing any whole number (integer).
- `/\d+`: Matches a second segment of one or more digits.
- `/d`: Matches the literal character `d`.

Example Matches

- `contrude/1/5/d`
- `contrude/123/456/d`

This regex ensures that both middle segments are integers and the final segment is exactly `d`.

write me a regex that excepts this string:
contrude/n/n/sensor

n: can be any whole number / integer
sensor: can be any string

To match the string format `contrude/n/n/sensor` where `n` is any integer and `sensor` can be any string, you can use this regex:

```
regex

^contrude/\d+/\d+/.*$
```

Explanation

- `^contrude/`: Asserts that the string starts with the literal `contrude/`.
- `\d+`: Matches one or more digits, representing any whole number (integer).
- `/\d+`: Matches the second segment, which is also one or more digits.
- `/.*`: Matches `/` followed by any sequence of characters (this allows `sensor` to be any string, including empty if needed).
- `$`: Asserts the end of the string.

Example Matches

- `contrude/1/5/temp`
- `contrude/123/456/any_string_here`
- `contrude/0/9999/sensor_data`

This regex ensures that the first two segments are integers and the last segment is any string.

the string should not be empty

If the `sensor` part should not be empty, you can adjust the regex as follows:

```
regex

^contrude/\d+/\d+/[^/]+$
```

Explanation

- `^contrude/`: Matches the string beginning with `contrude/`.
- `\d+`: Matches one or more digits, representing any whole number (integer).
- `/\d+`: Matches a second segment of one or more digits.
- `/[^/]+`: Matches a `/` followed by one or more characters that are not `/`, ensuring that `sensor` is not empty.
- `$`: Asserts the end of the string.

Example Matches

- `contrude/1/5/temp`
- `contrude/123/456/sensor_data`
- `contrude/0/9999/status`

This regex now enforces that the `sensor` segment contains at least one character and is not empty.

now format me this regex: "^contrude/\d+/\d+/[^/]+$" into a valid telegraf (TOML) regex string using the processors.reges plugin

To format this regex for use in a Telegraf `TOML` configuration with the `processors.regex` plugin, you'll need to escape the backslashes and enclose it in double quotes. Here's how it would look:

```
toml

[[processors.regex]]
  [[processors.regex.fields]]
    pattern = "^contrude/\\d+/\\d+/[^/]+$"
    replacement = "<your_replacement>"
```

Explanation

- Each `\` in the regex needs to be escaped with an additional `\` for TOML syntax, so `\d` becomes `\\d`.
- Enclose the entire regex pattern in double quotes.

Replace `<your_replacement>` with whatever you need the `processors.regex` plugin to replace matched strings with, or configure other settings as needed.

explain me, what the replacement is for

In Telegraf's `processors.regex` plugin, the `replacement` field is used to define what the matched text should be transformed into. When a field value matches the specified `pattern` (your regex), it can be replaced or modified according to the `replacement` value.

How `replacement` Works

- **Without** `replacement`: If you just want to check whether data matches a certain pattern, `replacement` can be omitted. This is common for validation-only purposes.
- **With** `replacement`: If you want to transform data, `replacement` specifies what the new value will be after a match is found. You can replace part of the field value, the entire field value, or format it in a new way.

Example Use Case

Suppose you want to rename any paths matching `contrude/\d+/\d+/[^/]+` to a standardized format like `standard_path`:

```toml
[[processors.regex]]
  [[processors.regex.fields]]
    pattern = "^contrude/\\d+/\\d+/[^/]+$"
    replacement = "standard_path"
```

With this configuration:

- Any field values matching `contrude/<number>/<number>/<sensor>` will be replaced with `"standard_path"`.

This feature is particularly useful in cases where you want to clean, anonymize, or standardize field data in your metrics.

is this valid:
[[processors.regex]]

 [[processors.regex.tags]]
  key = "topic"
  pattern = "^contrude/\\d+/\\d+/[^/]+$"

  replacement = "${ship},${container},${sensor}"

The configuration you provided is close, but it needs a few adjustments. In Telegraf's `processors.regex` plugin, capturing groups from the regex pattern (like `(?P<name>...)`) are generally not directly accessible in `replacement` fields as variables. So `${ship}`, `${container}`, and `${sensor}` will not work as-is.

To extract values into tags or fields based on capturing groups, you need to use **multiple regex processors** for each segment, or adapt the regex and processing to work within Telegraf's capabilities.

Here's an alternative that keeps `topic` as a single field but also captures specific groups. However, Telegraf's regex processor may not support named capturing groups directly. Instead, consider extracting each part in sequence if possible.

Updated Configuration without Named Groups

To proceed with this in a more compatible way:

```toml
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$1,$2,$3"  # $1, $2, and $3 are for each capture group
```

Explanation:

- `pattern`: The pattern uses `()` for capturing groups, without names. Each group represents a different component (`ship`, `container`, `sensor`).
- `replacement`: Uses `$1`, `$2`, and `$3` to refer to each capturing group.

This setup would replace a tag's `topic` value matching `contrude/n/n/sensor` with a comma-separated string of the values for `ship`, `container`, and `sensor`.

If you specifically need the extracted values to be assigned to separate tags (`ship`, `container`, `sensor`), you'll need additional steps, like a custom plugin or using multiple regex processors, as direct capture into multiple tags is not natively supported in this configuration.

and what if i used this:
[[processors.regex]]

 [[processors.regex.tags]]
  key = "topic"

  pattern = "contrude/(?P<ship>[^/]+)/(?P<container>[^/]+)/(?P<sensor>[^/]+)"
  replacement = "${ship},${container},${sensor}"

In Telegraf's `processors.regex` plugin, using named capturing groups (like `(?P<ship>...)`) and referencing them directly in `replacement` (e.g., `${ship}`) is **not supported** out of the box. Unfortunately, Telegraf doesn't natively recognize named groups within the `replacement` string.

Instead, Telegraf's `replacement` only supports basic positional references (`$1`, `$2`, etc.), where each `$n` corresponds to the order of the capturing group in the regex.

How to Work Around This Limitation

To handle this within Telegraf, you can use a sequence of `regex` processors to split out each part individually. Here's one way to approach it, by first capturing the entire string and then extracting each segment.

Example Configuration Using Multiple Processors

This example assumes you need separate tags for `ship`, `container`, and `sensor`:

```toml
# First, extract the whole `topic` field into a temporary tag
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/([^/]+)/([^/]+)/([^/]+)$"
    replacement = "$1,$2,$3"  # Comma-separated values for temporary processing

# Then split each part from the temporary tag
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"  # Matches the comma-separated format
    replacement = "$1"
    result_key = "ship"  # Creates the `ship` tag

[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"
    replacement = "$2"
    result_key = "container"  # Creates the `container` tag

[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"
    replacement = "$3"
    result_key = "sensor"  # Creates the `sensor` tag
```

Explanation

1. **First processor**: Extracts `ship`, `container`, and `sensor` as comma-separated values in the `topic` tag.
2. **Subsequent processors**: Each one extracts a specific value by using the correct positional reference (`$1`, `$2`, or `$3`) and assigns it to a new tag (`ship`, `container`, or `sensor`).

Result

After running this sequence, you'll have three separate tags (`ship`, `container`, and `sensor`) populated with the values extracted from `topic`.

but i want to only recieve topics with the discussed mqtt topic of "contrude/n/n/sensor"

where n is any integer and sensor can be any string which is not empty

To ensure you only receive topics matching the specific MQTT topic format of `contrude/n/n/sensor` (where `n` is any integer and `sensor` is a non-empty string), you can configure the `processors.regex` plugin in Telegraf to filter out the topics accordingly. Here's how to set it up:

Telegraf Configuration

You want to use the regex processor to match the desired pattern and discard any other topics. Here's how to do that:

```toml
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$0"  # Keep the original topic format if matched
    result_key = "filtered_topic"  # Store the filtered topic in a new key

[[processors.regex]]
  namepass = ["filtered_topic"]
```

Explanation

1. **First Processor**:
   - `key = "topic"`: This specifies the field in your incoming data where the topic will be matched.
   - `pattern`: The regex `^contrude/(\\d+)/(\\d+)/([^/]+)$` ensures that:
     - `contrude/` is the starting literal.
     - The first `\\d+` matches any integer.
     - The second `\\d+` matches another integer.
     - `([^/]+)` ensures that the sensor is a non-empty string (as it matches one or more characters that are not `/`).
   - `replacement = "$0"`: Keeps the matched topic as is if it conforms to the pattern.
   - `result_key = "filtered_topic"`: The matched and filtered topic is stored in a new field named `filtered_topic`.
2. **Second Processor**:
   - `namepass = ["filtered_topic"]`: This configuration tells Telegraf to only pass through metrics that have the `filtered_topic` field, effectively filtering out any messages that do not match your specified pattern.

Outcome

With this configuration, only topics that match the format `contrude/n/n/sensor` (where `n` is any integer and `sensor` is a non-empty string) will be processed further in Telegraf. All other topics will be ignored, ensuring that only valid topics are received and handled in your data pipeline.

i now have this telegraf configuration:
Extract the whole topic field into temporary tags
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$1,$2,$3" # Comma-separated values for temporary processing

Split each part from the temporary tag
[[processors.regex]]

```
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"  # Matches the comma-separated format
    replacement = "$1"
    result_key = "ship"  # Creates the ship tag

# Split each part from the temporary tag
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$" # Matches the comma-separated format
    replacement = "$2"
    result_key = "container"  # Creates the container tag

# Split each part from the temporary tag
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$" # Matches the comma-separated format
    replacement = "$3"
    result_key = "sensor"  # Creates the sensor tag
```

is there any way to make this simpler than you described it?

Yes, you can simplify your Telegraf configuration by consolidating the regex processing steps into a single processor that extracts the `ship`, `container`, and `sensor` in one go. Instead of processing the `topic` multiple times, you can directly create multiple tags from a single match using one regex pattern.

Here's how you can streamline your configuration:

Simplified Telegraf Configuration

```toml
# Extract ship, container, and sensor from the topic in one go
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$1,$2,$3"  # Comma-separated values for processing

[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"  # Matches the comma-separated format
    replacement = "$1"  # Creates the ship tag
    result_key = "ship"

[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"  # Matches the comma-separated format
    replacement = "$2"  # Creates the container tag
    result_key = "container"

[[processors.regex]]
  [[processors.regex.tags]]
```

```
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"  # Matches the comma-separated format
    replacement = "$3"  # Creates the sensor tag
    result_key = "sensor"
```

Consolidated Configuration

To make it even simpler and avoid redundant regex patterns, you can use just one processor to directly extract the tags without needing the intermediate step. Here's a cleaner approach:

```
toml

# Extract ship, container, and sensor from the topic in one go
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$1,$2,$3"  # Comma-separated values for processing
    result_key = "topic"

# Now split the combined topic into individual tags
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"  # Matches the comma-separated format
    replacement = "$1"  # Extracts ship
    result_key = "ship"

[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"  # Matches the comma-separated format
    replacement = "$2"  # Extracts container
    result_key = "container"

[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"  # Matches the comma-separated format
    replacement = "$3"  # Extracts sensor
    result_key = "sensor"
```

Final Alternative: Single Extraction

You can simplify this to a single extraction step without intermediate replacements by directly assigning the values to separate tags using the same pattern:

```
toml

[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$1,$2,$3"  # Comma-separated values for processing

# Creating individual tags from the initial match
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$1"
    result_key = "ship"  # Creates the ship tag

[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
```

```
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$2"
    result_key = "container"  # Creates the container tag

[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$3"
    result_key = "sensor"  # Creates the sensor tag
```

Summary

- This approach eliminates the need for multiple passes or temporary comma-separated values.
- Each processor directly captures and assigns the relevant parts of the topic to their respective tags, which makes the configuration cleaner and easier to maintain.
- Ensure that your version of Telegraf supports this approach, as features may vary slightly between versions.

in following questions use this processors.regex part:

```
# Extract the whole topic field into temporary tags
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$1,$2,$3"  # Comma-separated values for temporary processing

# Split each part from the temporary tag
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"  # Matches the comma-separated format
    replacement = "$1"
    result_key = "ship"  # Creates the ship tag

# Split each part from the temporary tag
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$" # Matches the comma-separated format
    replacement = "$2"
    result_key = "container"  # Creates the container tag

# Split each part from the temporary tag
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$" # Matches the comma-separated format
    replacement = "$3"
    result_key = "sensor"  # Creates the sensor tag
```

Got it! I'll use that specific `processors.regex` configuration in any related questions you have. Feel free to ask anything!

> i use influxdb v2 as a database. now i want to save this data into the database.

To save data extracted from your MQTT topics into InfluxDB v2 using Telegraf, you can configure the `outputs.influxdb_v2` section in your Telegraf configuration. Below is an example of how you might set it up based on your previous regex extraction configuration.

Example Telegraf Configuration to Output to InfluxDB v2

```toml
# Configuration to write data to InfluxDB v2
[[outputs.influxdb_v2]]
  ## The URLs of the InfluxDB instance
  urls = ["http://<your_influxdb_host>:<your_influxdb_port>"]

  ## Token for authentication
  token = "<your_influxdb_token>"

  ## The organization name
  organization = "<your_organization>"

  ## The bucket to write to
  bucket = "<your_bucket>"

  ## Optional: Customize the write timeout
  timeout = "5s"

  ## Optional: Set the precision of timestamps
  precision = "s"

  ## Optional: Include tag keys
  # tags = ["ship", "container", "sensor"]
```

Explanation of Key Configuration Fields

- `urls`: Specify the URL of your InfluxDB instance, including the host and port.
- `token`: Your InfluxDB authorization token, which is required to write data.
- `organization`: The organization name that you have set up in InfluxDB.
- `bucket`: The name of the bucket where you want to store the data.
- `timeout`: Adjust the timeout for writes to the database (default is usually fine).
- `precision`: Set the precision of the timestamps; you can use `ns`, `us`, `ms`, or `s` depending on your needs.

Data Format

When you configure the output like this, Telegraf will write the data using the following points structure:

- Each measurement corresponds to the extracted fields (e.g., `ship`, `container`, and `sensor`).
- Tags (like `ship`, `container`, and `sensor`) can be added to enrich the data.

Full Example Configuration

Combining both the regex extraction and the InfluxDB output, your full Telegraf configuration might look like this:

```toml
# Extract the whole topic field into temporary tags
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$1,$2,$3"  # Comma-separated values for temporary processing

# Split each part from the temporary tag
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"  # Matches the comma-separated format
    replacement = "$1"
    result_key = "ship"  # Creates the ship tag

# Split each part from the temporary tag
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$" # Matches the comma-separated format
    replacement = "$2"
    result_key = "container"  # Creates the container tag

# Split each part from the temporary tag
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$" # Matches the comma-separated format
    replacement = "$3"
    result_key = "sensor"  # Creates the sensor tag

# Configuration to write data to InfluxDB v2
[[outputs.influxdb_v2]]
  urls = ["http://<your_influxdb_host>:<your_influxdb_port>"]
  token = "<your_influxdb_token>"
  organization = "<your_organization>"
  bucket = "<your_bucket>"
  timeout = "5s"
  precision = "s"
```

Steps to Implement

1. Replace placeholders (like `**<your_influxdb_host>**`, `**<your_influxdb_token>**`, etc.) with your actual values.
2. Save your Telegraf configuration file.
3. Restart the Telegraf service to apply the changes.

With this setup, Telegraf will process your MQTT data and write the extracted `**ship**`, `**container**`, and `**sensor**` values into your specified InfluxDB v2 bucket.

i now use this output config:
Output Plugin Configuration
[[outputs.influxdb_v2]]
  urls = ["http://db_sensor:8086"]
  token = "${INFLUX_TOKEN}"
  organization = "${INFLUX_ORG}"

```
bucket = "${INFLUX_BUCKET}"
precision = "s"
```

Your InfluxDB v2 output configuration for Telegraf looks good! Using environment variables like `${INFLUX_TOKEN}`, `${INFLUX_ORG}`, and `${INFLUX_BUCKET}` is a great practice for managing sensitive information and making your configuration more flexible.

Steps to Ensure Everything Works

1. **Environment Variables**: Make sure that the environment variables are correctly set in your environment where Telegraf is running. You can check this by running:

   ```bash
   echo $INFLUX_TOKEN
   echo $INFLUX_ORG
   echo $INFLUX_BUCKET
   ```

   Each of these commands should return the expected values.

2. **Telegraf Service**: After modifying the configuration file, restart the Telegraf service to apply the changes:

   ```bash
   sudo systemctl restart telegraf
   ```

3. **Logs**: Monitor Telegraf logs for any errors or warnings that may indicate issues with connecting to InfluxDB or writing data:

   ```bash
   sudo journalctl -u telegraf -f
   ```

4. **InfluxDB Configuration**: Ensure that your InfluxDB instance is running and accessible at `http://db_sensor:8086`. You can verify this by trying to access the InfluxDB UI or using a curl command:

   ```bash
   curl -G http://db_sensor:8086/api/v2/buckets \
        --header "Authorization: Token ${INFLUX_TOKEN}"
   ```

5. **Data Verification**: After everything is set up, verify that data is being written to your specified bucket in InfluxDB. You can use the InfluxDB UI or a client library to query the data.

Sample Query to Verify Data

After sending data, you can run a query to check if the expected measurements are present in your bucket:

```
flux

from(bucket: "${INFLUX_BUCKET}")
  |> range(start: -1h)  // Adjust the time range as needed
  |> filter(fn: (r) => r._measurement == "<your_measurement_name>")
```

Replace `<your_measurement_name>` with the name of the measurement you're expecting to see.

If you have any specific questions or run into issues, feel free to ask!

In contrude/n/n/sensor, the sensor value can be any string which is not empty. In conclusion the sensor tag can have different values. For example temperature, humidity, etc.. I already persisted all the data into one influxdb v2 database.
Now I want to persist the data into different databases depending on the value of the sensor tag. For example if the sensor tags value is temperature, the data from this input should be saved into the database temperature. The same thing applies for humidity. If the sensor tag is humidity, the value of this input should be saved into the humidity database. Is this possible?

Yes, it's possible to write data to different databases in InfluxDB based on the value of the sensor tag. However, to achieve this functionality using Telegraf alone can be somewhat limited since Telegraf's output configuration does not support conditional routing natively.

Possible Approaches

Here are a few approaches you could consider:

1. Use Multiple Telegraf Instances

You can run multiple instances of Telegraf, each configured to listen for specific sensor types and write to the corresponding database. For instance:

- **Instance for Temperature**: Configure one Telegraf instance that only processes messages with the sensor tag value `temperature` and writes to the `temperature` database.
- **Instance for Humidity**: Configure another Telegraf instance that processes messages with the sensor tag value `humidity` and writes to the `humidity` database.

Each instance would use the `[[outputs.influxdb_v2]]` section pointing to different databases based on the sensor.

Example Configuration

For the temperature instance:

```
toml

[[outputs.influxdb_v2]]
  urls = ["http://db_sensor:8086"]
  token = "${INFLUX_TOKEN}"
  organization = "${INFLUX_ORG}"
  bucket = "temperature"  # Point to temperature database
```

For the humidity instance:

```toml
[[outputs.influxdb_v2]]
  urls = ["http://db_sensor:8086"]
  token = "${INFLUX_TOKEN}"
  organization = "${INFLUX_ORG}"
  bucket = "humidity"  # Point to humidity database
```

2. Use a Stream Processor

Another approach is to use a stream processing tool (like Apache Kafka with Kafka Connect) to route messages based on their content. You would:

1. Publish all MQTT messages to a Kafka topic.
2. Use a Kafka consumer that reads messages and writes them to the appropriate InfluxDB database based on the sensor type.

This setup adds complexity but allows for more flexibility in routing.

3. Use `[[processors.regex]]` with Dynamic Tags

Telegraf doesn't support direct conditional logic for output databases, but you can use dynamic tags in the `[[processors.regex]]` to filter data that goes to specific outputs. However, you would still need separate output configurations.

Here's a conceptual example of how to set it up using the processors, but you would still need to define multiple outputs manually:

```toml
[[processors.regex]]
  [[processors.regex.tags]]
    key = "sensor"
    pattern = "^(temperature|humidity)$"
    result_key = "sensor_type"

[[outputs.influxdb_v2]]
  urls = ["http://db_sensor:8086"]
  token = "${INFLUX_TOKEN}"
  organization = "${INFLUX_ORG}"
  bucket = "temperature"
  namepass = ["sensor_type:temperature"]

[[outputs.influxdb_v2]]
  urls = ["http://db_sensor:8086"]
  token = "${INFLUX_TOKEN}"
  organization = "${INFLUX_ORG}"
  bucket = "humidity"
  namepass = ["sensor_type:humidity"]
```

In this setup, you'd use `namepass` to filter data based on the `sensor_type` tag. However, keep in mind that you need to ensure your Telegraf version supports this.

Conclusion

Using multiple Telegraf instances or a streaming platform like Kafka can give you the flexibility you need to route data to different databases based on sensor values. The first option (multiple Telegraf instances) is simpler to implement and maintain if you're primarily using Telegraf. If your setup grows in complexity, consider integrating a message queue or stream processor for more advanced routing and handling.

> i would like to use the thirds option with just one telegraf and processor.regex. explain this method to me

Using a single Telegraf instance with `processors.regex` to dynamically route data based on the `sensor` tag requires a bit of configuration trickery, as Telegraf does not natively support conditional routing within a single output plugin. However, you can use multiple `[[outputs.influxdb_v2]]` configurations and selectively pass data to each based on the `sensor` tag's value.

Here's a step-by-step breakdown of this approach:

Configuration Setup

1. **Extract the `sensor` Tag**: Use `processors.regex` to ensure the `sensor` value (like `temperature` or `humidity`) is extracted as a tag from your MQTT topic.
2. **Apply Tag-Based Filtering**: With the `sensor` tag set up, you can then define multiple `outputs.influxdb_v2` plugins, each configured to write to a different database (or bucket in InfluxDB v2), and apply a filter to each output to match only certain `sensor` values. This filtering is done using the `namepass` or `tagexclude` option.

Example Configuration

Here's an example configuration to illustrate how this works. Let's assume the `sensor` tag is extracted from your MQTT topic with values like `temperature` or `humidity`.

1. Extract the Sensor Information with `processors.regex`

To extract `ship`, `container`, and `sensor` values, the following regex processor example will work with your existing topic structure. This will allow us to create the `sensor` tag, which we will use to conditionally write to different InfluxDB databases.

```toml
# Step 1: Extract topic components
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^contrude/(\\d+)/(\\d+)/([^/]+)$"
    replacement = "$1,$2,$3"

# Step 2: Set ship, container, and sensor tags individually
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"
    replacement = "$1"
    result_key = "ship"
```

```
[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"
    replacement = "$2"
    result_key = "container"

[[processors.regex]]
  [[processors.regex.tags]]
    key = "topic"
    pattern = "^([^,]+),([^,]+),([^,]+)$"
    replacement = "$3"
    result_key = "sensor"
```

2. Configure Multiple Outputs with Filters

Next, we'll set up two different `[[outputs.influxdb_v2]]` configurations, each one configured to filter data based on the `sensor` tag value.

Temperature Output

This output writes only data where `sensor` equals `temperature`:

```toml
[[outputs.influxdb_v2]]
  urls = ["http://db_sensor:8086"]
  token = "${INFLUX_TOKEN}"
  organization = "${INFLUX_ORG}"
  bucket = "temperature"  # Bucket for temperature data
  precision = "s"

  # Only pass data where the sensor tag is "temperature"
  [outputs.influxdb_v2.tagpass]
    sensor = ["temperature"]
```

Humidity Output

This output writes only data where `sensor` equals `humidity`:

```toml
[[outputs.influxdb_v2]]
  urls = ["http://db_sensor:8086"]
  token = "${INFLUX_TOKEN}"
  organization = "${INFLUX_ORG}"
  bucket = "humidity"  # Bucket for humidity data
  precision = "s"

  # Only pass data where the sensor tag is "humidity"
  [outputs.influxdb_v2.tagpass]
    sensor = ["humidity"]
```

Explanation

- **Tag Extraction**: Using the `processors.regex` processor, we extract `ship`, `container`, and `sensor` from the `topic`, so each measurement now includes these tags.
- **Output Filtering**: The `tagpass` option in each `outputs.influxdb_v2` block filters data based on the `sensor` tag. Data points with a `sensor` value of `temperature` go only to the `temperature`

bucket, while those with a `sensor` value of `humidity` go to the `humidity` bucket.
- **Single Telegraf Instance**: Since you're only running one instance of Telegraf, all of this configuration is managed within one Telegraf service.

Additional Considerations

- **Unmatched Data**: If a `sensor` tag doesn't match either `temperature` or `humidity`, it won't be sent to any output bucket due to the `tagpass` filters.
- **Scalability**: If you have more sensor types, you'll need to add more `outputs.influxdb_v2` blocks, each with a corresponding `bucket` and `tagpass` filter.

This method provides flexibility to route data within a single Telegraf instance and ensures each sensor type can go to its own database bucket as needed. Let me know if you have further questions or need any adjustments!

This conversation is archived. To continue, please unarchive it first.

Unarchive