|  |  |
| --- | --- |
|  | Data Processing |
|  |  |
|  | [Author name]  MOVE |

Logo

Description automatically generated with medium confidenceGraphical user interface, text

Description automatically generated with medium confidenceLogo

Description automatically generatedGraphical user interface, text

Description automatically generated with medium confidence

Table of Contents

[High-Level Overview 2](#_Toc47687242)

[Frontend vs. Backend Processing 2](#_Toc47687243)

[Frontend 2](#_Toc47687244)

[Backend 2](#_Toc47687245)

[Normalisation 3](#_Toc47687246)

[Processing 4](#_Toc47687247)

[Processing New Data 4](#_Toc47687248)

[Processing Existing Data 5](#_Toc47687249)

[To Database 5](#_Toc47687250)

[To CSV & XLSX 5](#_Toc47687251)

# High-Level Overview

Monnit sensors format their data into a single flat file regardless of the amount of measurands monitored by a sensor, the humidity sensors report four measurands concatenated into a single string resulting in the following; ‘Humidity|Celsius|DewPoint|GramsPerKilogram’.

This format allows the data to be easily stored, and transmitted, with minimal footprint and without any additional complexity arising from maintaining separate measurands but becomes an issue when visualising measurands and working with specific measurands.

The issues with visualisation can be seen on the test dashboard where data from the relative humidity sensors seems to continuously increase, and displays non-sensical data, because the dashboard is trying to read the concatenated measurands as a single value.

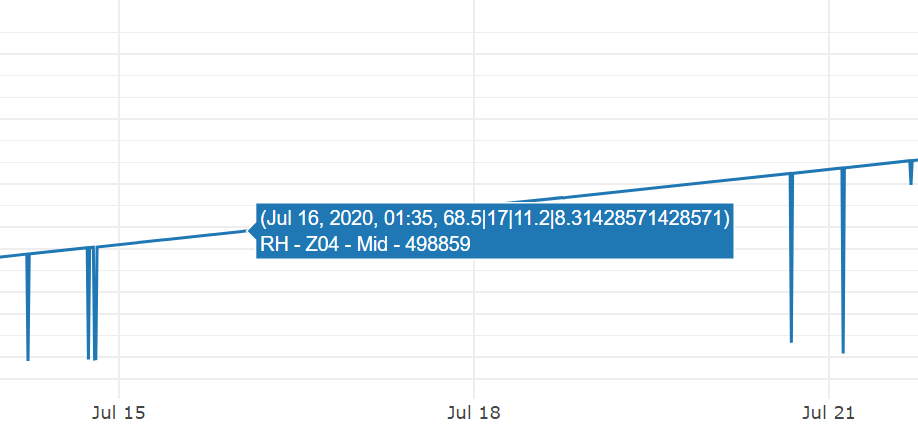


Fig.1. Data sample from a RH sensor showing concatenated values

## Frontend vs. Backend Processing

The concatenated data from the sensors can be dealt with two ways; on the frontend or on the backend.

### Frontend

Processing the data on the front end would require the dashboard solution to process and split the data as it is read from the database, and from there the processed data could either be written back to the database in it’s processed form or discarded and reprocessed if the data is needed again.

This would allow the platform to maintain a smaller footprint because a dedicated backend for data processing would not be required, but the additional load on the frontend can negatively impact the performance of the dashboard system.

### Backend

Processing the data on the backend would require a separate dedicated host for processing the data as it’s received from the Monnit servers, and in this case there is already a backend system that receives data from the Monnit servers and stores it into an SQL database so expanding the existing system to process the data into a more useable form will be the most efficient approach.

# Normalisation

The process of normalisation allows for the removal of redundant data from a dataset by breaking it down into multiple tables with key-dependencies allowing redundant data to be stored in one location and referenced in tables that require it.

Normalising the data that is received from Monnit will provide the template structure that the sensor data will be processed in to.

The normalisation table for the sensor data is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| UNF  Repeating attributes indented | 1NF  Remove repeating attributes and identify a primary key | 2NF  Remove partial dependencies | 3NF  Remove non-key dependencies |
| sensorID  sensorName  applicationID  dataMessageGUID  sensorState  messageDate  rawData  dataType  dataValue  plotValues  plotLabels  bateryLevel  signalStrength  pendingChange  voltage  gatewayID  gatewayName  accountID  networkID  messageType  gatewayPower  batteryLevel  gatewayDate  gatewayCount  signalStrength  pendingChange | **sensorID**  sensorName  applicationID  networkID | **sensorID**  sensorName  applicationID  networkID | **sensorID**  sensorName  applicationID\*  networkID\* |
| **dataMessageGUID**  rawData  dataValue  plotValues  messageDate | **dataMessageGUID**  rawData  dataValue  plotValues  messageDate | **readingID**  **dataMessageGUID**  sensorID**\***  dTypeID\*  plotLabelID\*  rawData  dataValue  plotValues  messageDate |
| dataType | dataType | **dTypeID**  dataType |
| plotLabels | plotLabels | **plotLabelID**  plotLabels |
| batteryLevel | batteryLevel | **readingID\***  **dataMessageGUID\***  batteryLevel |
| signalStrength | signalStrength | **readingID\***  **dataMessageGUID\***  signalStrength |
| pendingChange | pendingChange | **readingID\***  **dataMessageGUID\***  pendingChange |
| voltage | voltage | **readingID\***  **dataMessageGUID\***  voltage |
| **applicationID**  applicationName |
| **networkID**  networkName |
| **accountID**  accountName |
| **gatewayID**  gatewayName  accountID  messageType  gatewayPower  batteryLevel  gatewayDate  gatewayCount  signalStrength  pendingChange | **gatewayID**  gatewayName  accountID  messageType  gatewayPower  batteryLevel  gatewayDate  gatewayCount  signalStrength  pendingChange | **gatewayID**  gatewayName  accountID\*  messageType  gatewayPower  batteryLevel  gatewayDate  gatewayCount  signalStrength  pendingChange |

Additional details on the normalisation process, and the subsequent ERDs and data dictionaries, can be found in the document titled ‘SMAG Data Design’ accompanying this document.

Something to note about this normalisation process is that data relating to the Monnit IoT gateways is not normalised and the old scheme is still used, and this is because gateway data isn’t useful to the frontend dashboard so the additional time to normalise this data was unnecessary at this time.

# Processing

Processing the data into the new normalised form needs to be done in two stages; processing new data from the Monnit servers as it is received by the webhook listener, and processing the data that has already been collected and stored in the database into the new normalised form.

Much of the complexity from processing the data into its normalised form comes from data being written to, and retrieved from, the database in a specific order so the normalised form can be achieved, because the variables ‘sensorID’, ‘readingID’, ‘dTypeID’, and ‘plotLabelID’ are all generated by the database, or selected from the database, when the requisite data is sent to the database.

The order in which the data is sent is also strongly enforced because the foreign key dependencies of the database tables will not allow data to be written without respecting key dependencies as this would jeopardise the integrity of the stored data.

## Splitting Monnit Data

Splitting the concatenated data required the use of a function designed to read in a JSON entry and split specific entries using a list of prepared delimiters, and the currently identified list of delimiters are ‘%2c’, ‘|’, ‘%7c’, and ‘%7c0’.

It is unclear why the Monnit system uses multiple delimiters to separate data within a concatenated string and is especially unintuitive with regards to the ‘%7c0’ delimiter as it only occurs at the end of a concatenated string using ‘%7c’ as a delimiter.

This caused the split function to separate the ‘0’ from the ‘%7c0’ delimiter into a separate row, and thus polluting the processed data, as the delimiter was not expected.

It is unclear if including ‘%7c0’ in the delimiters list will negatively effect the data in other areas by incorrectly splitting data where a value of ‘0’ occurs after a ‘%7c’ delimiter in a valid context.

## Processing New Data

Processing the new data into its normalised form required a rewrite of the existing webhook script that received the data coming from the Monnit servers to be able to split the data into its normalised form.

Rewriting the webhook script also allowed some improvements to the original design of the webhook including the following:

* Switch from python defined SQL statements to Stored Procedures
* Obfuscate access credentials for the database and webhook POST by pulling credentials from hidden .JSON files within the ‘config’ directory

Transitioning the SQL transactions from traditional fixed SQL statements to the use of stored procedures in particular provides a significant benefit in terms of security as it prevents the SQL used in the script from being misappropriated if the SQL server is configured to disallow the use of SQL statements and forces the use of stored procedures.

## Processing Existing Data

Processing the data that has already been collected by the system works similarly to processing new data, and so much of the work done rewriting the webhook script will be used in the scrips for processing the old data.

There are currently two scripts for processing the old data; one that processed the data and saves it back to the database in it’s normalised form, and one that processes the data and exports it to CSV and XLSX files without sending the data back to the database server.

### To Database

The script that stores the data back in the database using its normalised form is intended to be run once, after the webhook listener is changed over to the new normalised system and will convert the old data to the new form and thus preserving the most complete and comprehensive version of the data.

#### Performance Issues

During the testing of the script it was observed that the processing speed of the script was low enough for the full execution of the script to be estimated in days, as the script was tested over a 24 hour period without it reaching a finished state.

The cause of this is that the script uses the same stored procedures and data structure as the webhook listener, but the webhook listener isn’t designed for processing throughput.

There are two solutions available to remedy this problem, the first is to run the script until completion regardless of the slow processing speed, and the second is to rewrite the processing script to process the different messages into their normalised before sending them back to the database and sending all the data back to the database in a single batch using SQL server’s ‘fast executemany’ function.

In theory this would significantly improve the throughput of the script, but it requires an extensive rewrite of the processing script and it’s associated stored procedures so the data that is generated by the database server is still generated and sent back to the processing script before writing the bulk of the data back to the database.

### To CSV & XLSX

The last script works in the same way as the previous and uses most of the same functions, but because it doesn’t send any data back to the database the data the is exported to the CSV and XLSX files aren’t fully normalised and don’t respect key dependencies.

This isn’t an issue because this script is intended for exporting sensor data in a convenient format for analysis, and most of the normalised data isn’t useful for analysis, so its absence isn’t relevant.

The aforementioned performance issues aren’t as prevalent with this script because the bottleneck with the previous script is with how the data is being written back to the database, and because this script doesn’t interact with the database outside of the initially retrieving the data the bottleneck isn’t encountered.