DATA LOAD ARCHITECTURE

DEVELOPER GUIDE CAN 6.0

FEBRUARY 25, 2023

AVANSEUS TECHNOLOGY PVT. LTD.

REVISION HISTORY

Version	Date	Change de-scription	Created By	Updated by	Reviewed by
V6.0	February 2023	Initial Release	Shlaghana	Raksha	Chiranjib

Table of Contents

1 Introduction

This document describes the sequence of steps to be followed for data load and architecture of data load process.

2 Table Description

This section includes reference table and transaction table with description.

Reference Table

Table Name	Description
FileCollectionConfiguration	Contains configured file Collection configuration
EventFileFormat	Contains configured record parser details
Preprocessor	Contains configured pre-processor
PostParsingProcessingLogic	Contains splitting and grouping screen configuration
Postprocessor	Contains configured postprocessor
DataCollectionConfigurationTemplate	Has master code template used for Data Collection & Configuration
3GPP5GConfigProperties	This contains 3GPP configuration properties
KafkaConfigProperties	Contains default configurations for Kafka
EventFileFormatTemplate	This contains template for record parser mapping code and template code for custom file type
PostParsingTemplate	This contains templates for Spitting and grouping screens of file-postprocessor
PreProcessorTemplate	Has master code template used for parser's pre-processor
PostProcessorTemplate	Has master code template used for parser's post- processor

Transaction Table

Data load takes place to the Datasource tables such as performance counter, Alarms, Tickets etc.

If resource is uploaded from Resource configuration screen, Resource Id is used as table for data load.

3 List of Indices

Master table indices: Master table refers to ids that are stored in DataSource table. Unique index for name field must be added to all master tables.

Example: db.EquipmentComponent.createIndex({name : 1}, {unique: true});

Data source table indices: For DataSource tables such as Alarms, Performance counter, tickets etc. unique field/set of fields is selected and index is added on those fields. This maintains uniqueness of data and avoid duplicate entries.

4 Configuration for Data Load from UI

Input Mapper Configuration

File Pre - Processor

File Pre-Processor screen is used to process the data before mapping it to CAN field. This is helpful when some data needs to be excluded from data load or some input data value needs to be modified.

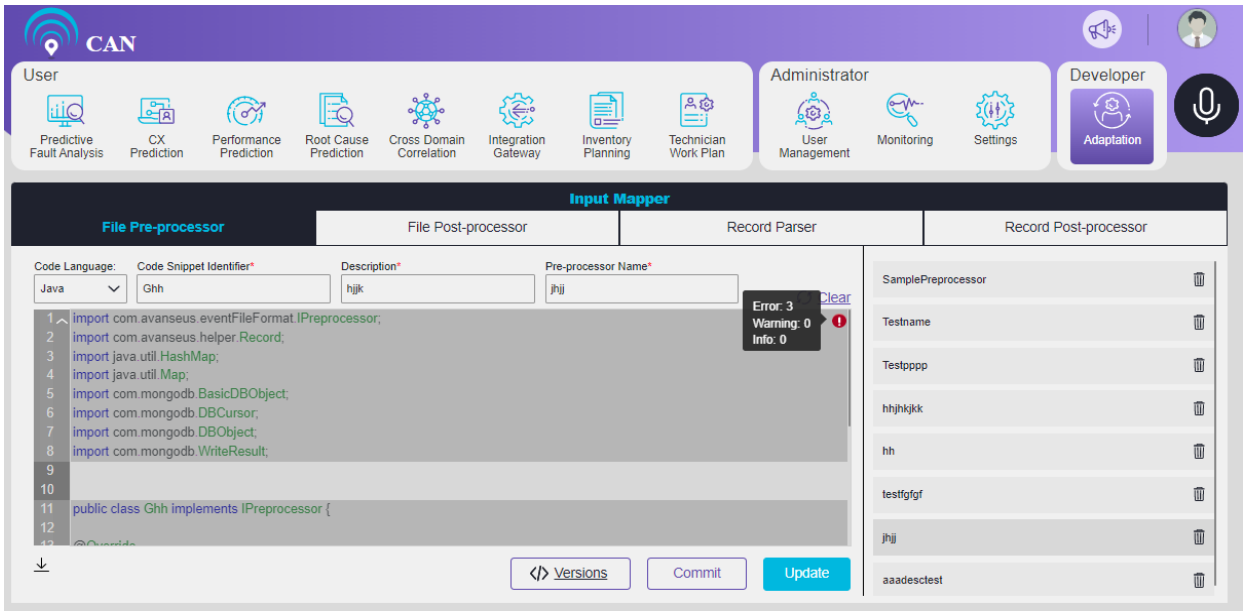

Figure 4-1 - File Pre-Processor Screen

File Post-processor

Splitting: It is defined as converting single record into multiple records.

- **Direct Mapping**

  Mapped attribute header requires list of columns to be split. The splitting takes place in such a way that number of newly added records is equal to size of mapped attribute. Two columns are newly added in the parsed record after split.

- **Custom Mapping**

  The custom mapping can be used if single column cannot be used as split logic. Criteria that are more complex can be configured by writing code in IDE.

Grouping: It is defined as grouping multiple records into single record.

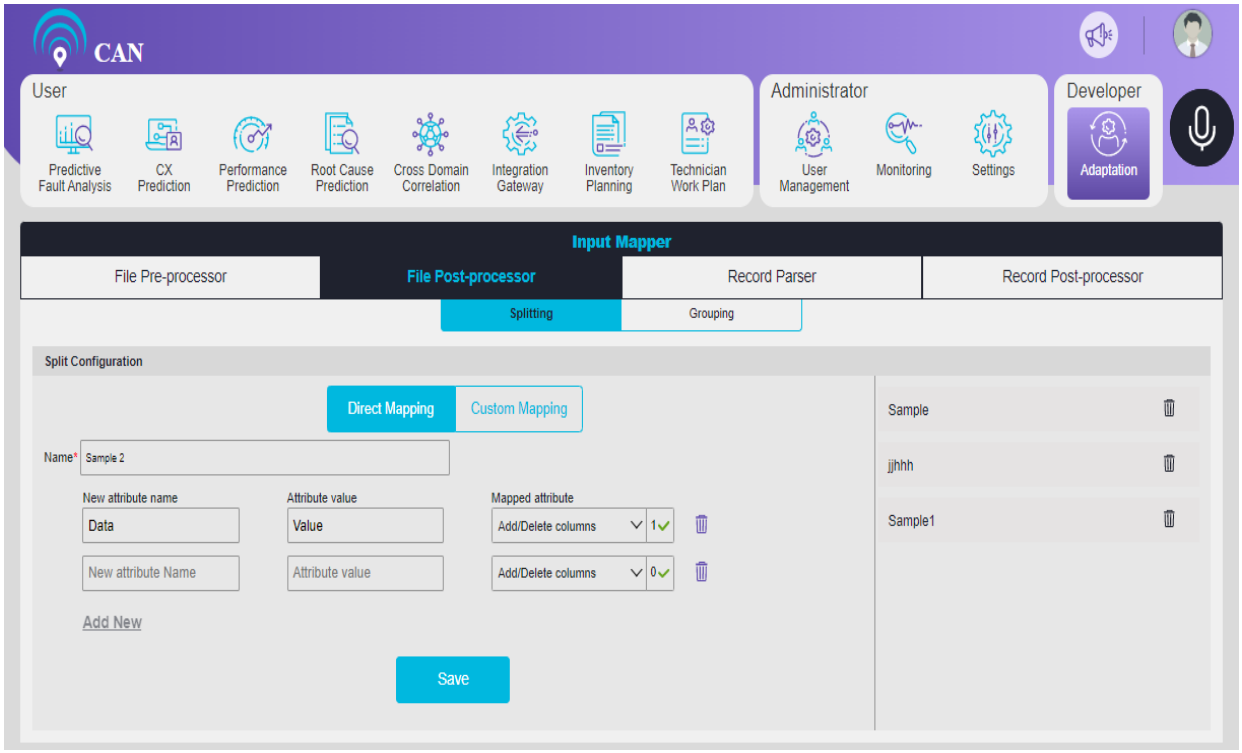

Figure 4-2 - File Post-Processor Screen

Record Parser

This configuration involves mapping of fields in file to CAN fields. Here mapping is done either directly or by using IDE code. Record processor uses this configuration to transform data and make it compactable with CAN module.

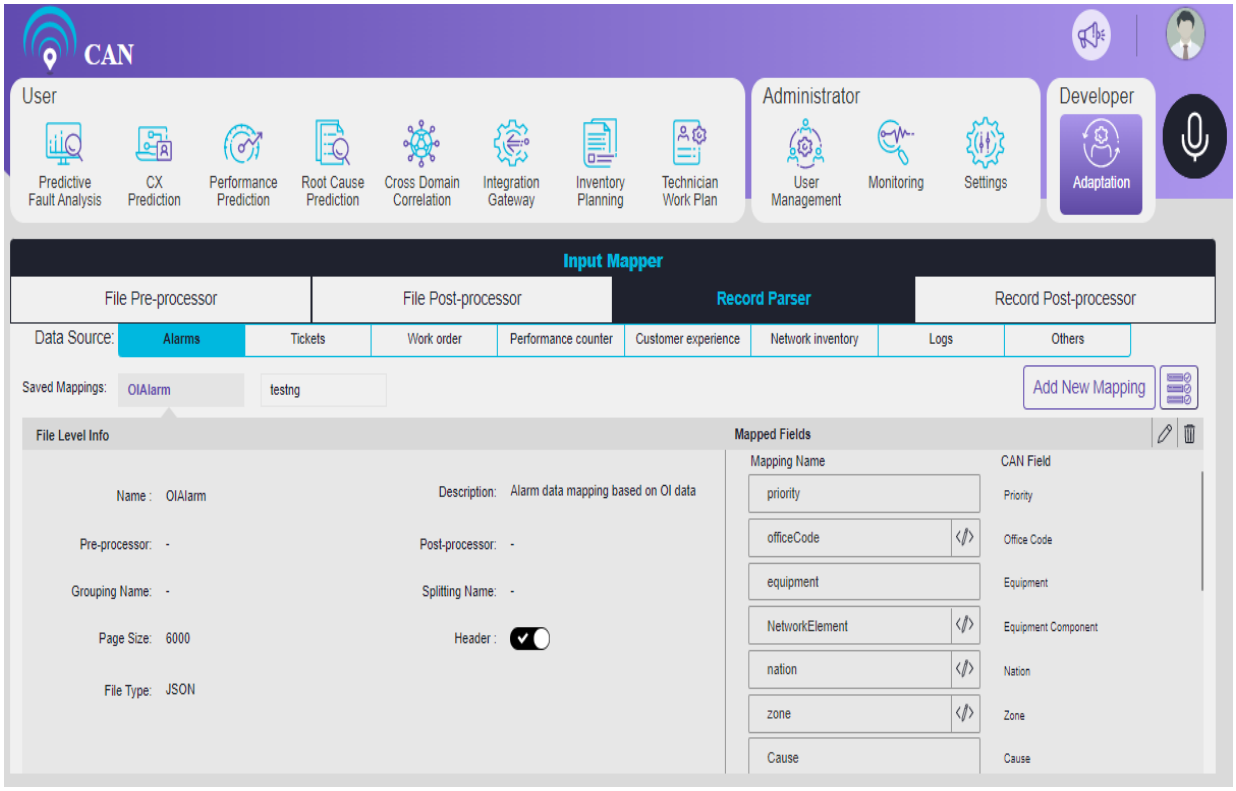

Figure 4-3 - Record Parser Screen

Record Post - Processor

Post-processor is used to modify or discard the data after parsing and just before loading of data. Data that comes as input here is transformed data.

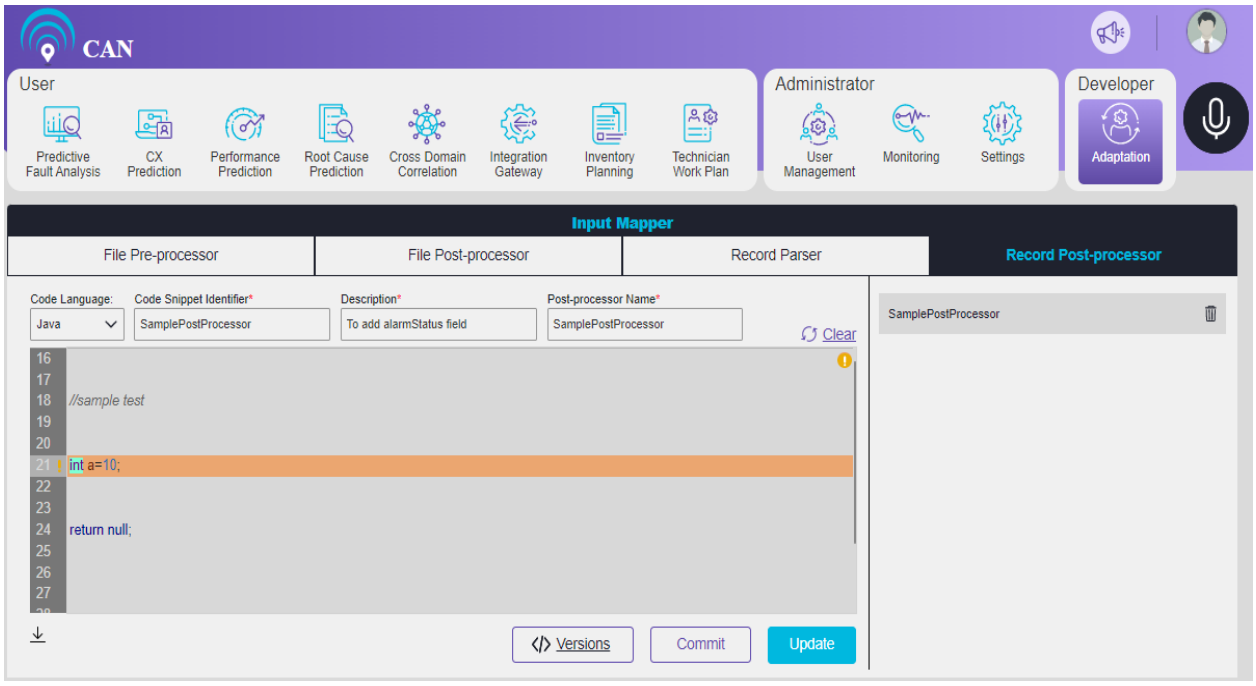

Figure 4-4 - Record Post-Processor Screen

Data Collection Configuration

Data Collection Configuration include configurations that are applicable for collecting data files from remote source. Configurations that are active in this tab are considered for data load.

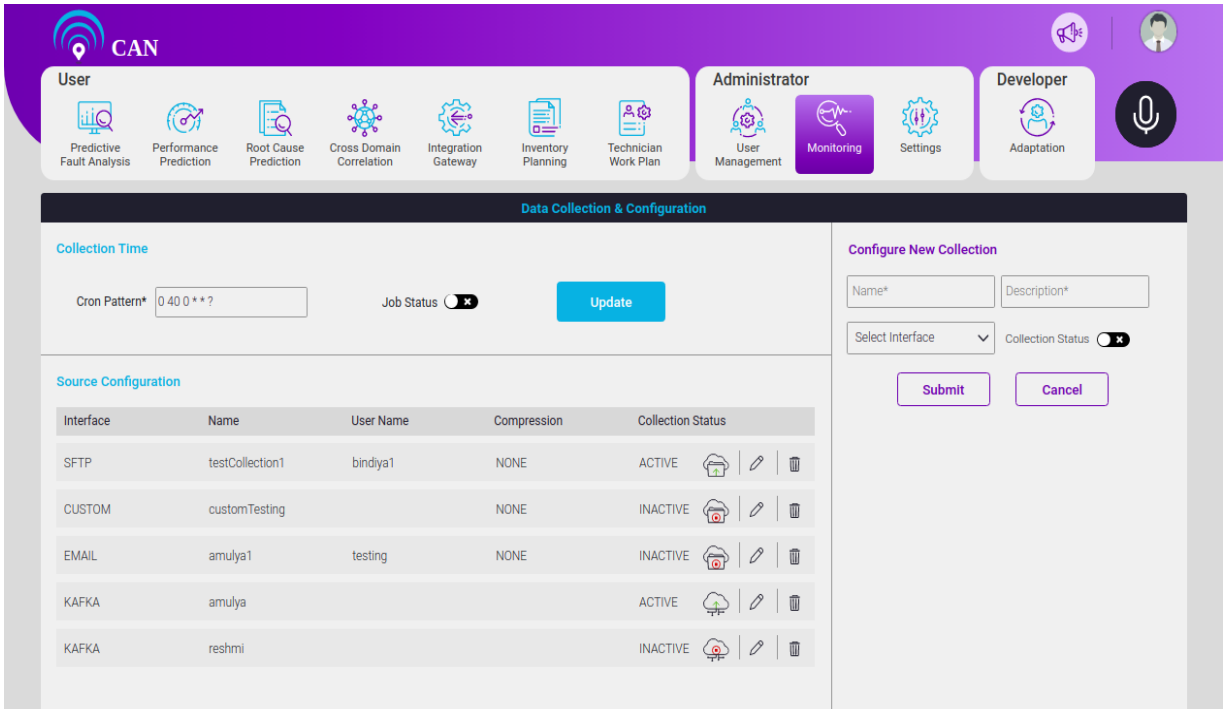

Figure 4-5 - Data Collection & Configuration

5 Data Load Process

Data load is a 3-phase process that involves data collection, data extraction and data transformation & loading.

Data Collection

Data collection takes place in CAN module, where data is collected in the form of batch or stream. Data collection include configurations that are applicable for collecting data files from remote source.

Data Extraction

Data extraction takes place in Batch Handler. Collected files from CAN module are passed to Batch Handler GRPC; where the file is converted into list of records after applying certain operations such as file pre-processor, file post-processor.

Data Transformation and Loading

Data transformation and loading takes place in Record Processor. List of records from Batch Handler is sent to Record processor GRPC, where record is converted into CAN compactable processed data and then stored in database.

Figure 5-1 - 3-Phase Data Load

Block Representation of Data Load

Block diagram shown below describes the data load process. It involves following steps.

1. CAN module collects data from remote sources, stores it in application and passes information to batch handler.
2. Batch Handler GRPC has one or more running instances where files from CAN module are equally distributed. Batch Handler processes files in batches and converts it to records.
3. Records from Batch Handler that are received by Record processor GRPC has one or more running instances which internally has threads equal to batch size specified in batch handler. Here records are transformed and loaded to database.

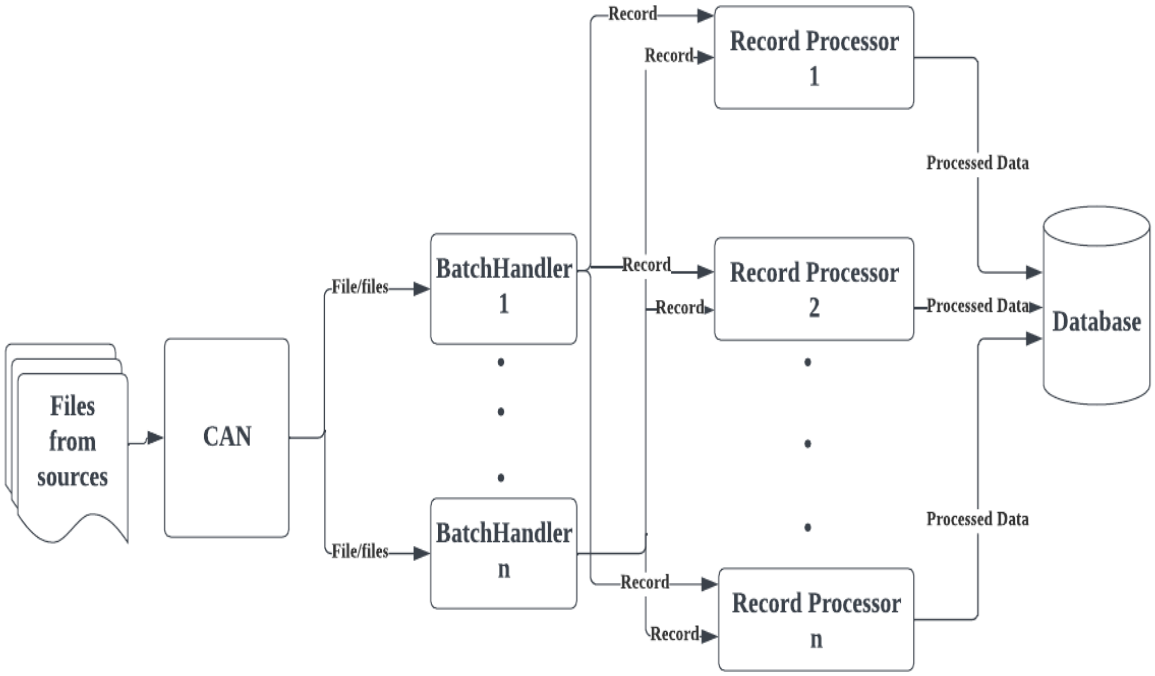

Figure 5-2 - Data Load Process

6 Data Collection in CAN

Data collection occurs as batches or streams in CAN module.

Data Collection as Batch

Collection of data is triggered at a specified time using cron pattern as a batch. Data collection as batches is achieved using following interfaces:

- SFTP
- FTP
- GITHUB
- EMAIL

Data Collection as Stream

Realtime streaming is achieved using following interfaces:

- Secure KAFKA
- 3GPP
- Prometheus

Data streaming from remote sources is achieved in CAN module using streaming interfaces. Data as stream entering CAN module is stored as small chunk files. Chunk files can hold up to 5 minutes data, later it is sent to data loading.

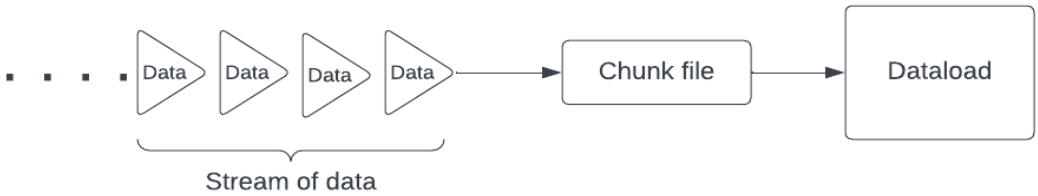

Figure 6-1 - Data Collection as Stream

7 Batch Handler GRPC

Batch Handler is a micro-service that is used to parse files by converting it to records in batches. Batch Handler is initiated from CAN. Output from batch handler is list of records.

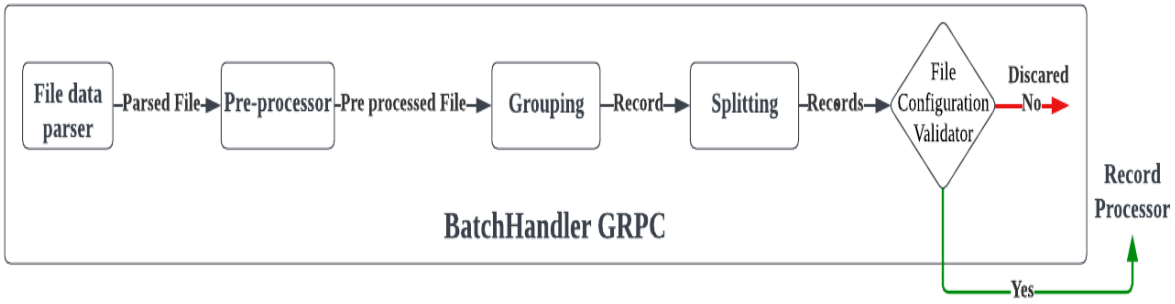

Figure 7-1 - Batch Handler GRPC

Batch Handler GRPC internally processes data through following pipeline.

File Data Parser:

File data parser is used to parse file obtained as input from CAN. Format of file that needs to be parsed is identified from file type configuration in parser screen. The file parsers are as follows.

1. Delimited/CSV file parser
2. Excel File parser
3. Custom delimited file parser
4. JSON file parser
5. XML
6. Custom

Pre-Processor Pipeline:

Configured Pre-processor is executed for the complete file. After execution, the processed data is sent to next pipeline. This is optional pipeline and is skipped if not configured.

Grouping Pipeline:

Processed data is consolidated as per logic configured in file post-processor. This is optional pipeline and is skipped if not configured.

Splitting Pipeline:

Here single record is split as list of records as per logic configured in file post-processor. This is optional pipeline and is skipped if not configured.

File Configuration Validator:

Data undergoes one level of validation where files for which incorrect columns are configured are discarded and not passed to record processor.

8 Caches in Batch Handler

Pre-processor cache: This cache contains all pre-processor classes and is refreshed in every 5 minutes. Any changes in pre-processor configuration needs 5 minutes to reflect.

Split and group cache: This cache contains all split and group classes and is refreshed in every 5 minutes. Any changes in file post-processor configuration needs 5 minutes to reflect.

9 Record Processor

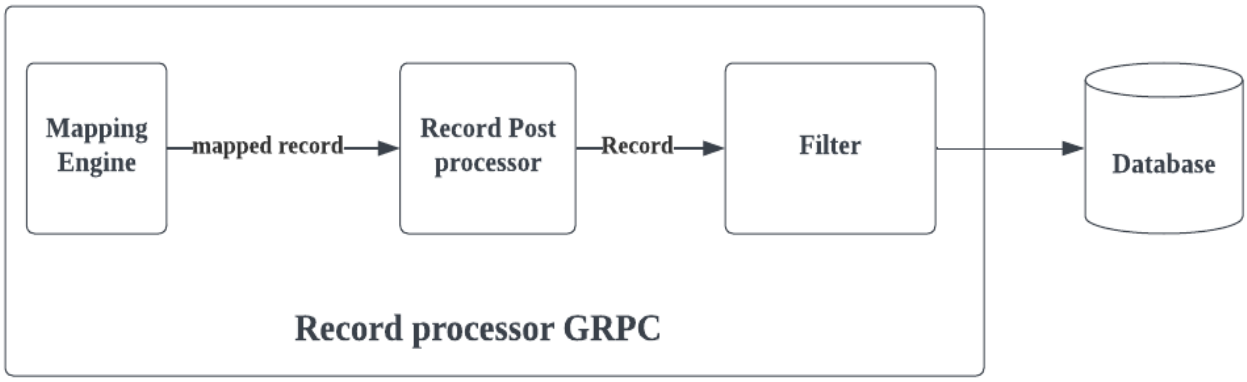

Figure 9-1 - Record Processor GRPC

Record processor GRPC is used to convert record to CAN compatible format. Input Mapper (record parser tab) configuration is used to convert raw data into CAN specific data and loaded into database. Record processor GRPC runs through pipelines.

Mapping Engine

Mapping engine involves various operations to convert raw data into CAN compatible data. Mapping resolution takes place in steps as shown below:

- **Direct mapping**: It maps the fields that are directly mapped from UI through either mapping field name or mapping field code.
- **Master mapping:** Master field ids are stored in DataSource tables. Initially default values are added to master tables. This transformed data is dumped into respective master tables and id is fetched. Data extraction mainly takes place from master table caches**. Hence, proper config entry for record processor cache, to avoid junk data from getting loaded and any data manipulation should follow restart of record processor**.
- **Aggregator Mapping:** It maps aggregator object fields. This is mainly used in prediction generation.
- **Dependent Mapping:** Mapping of dependent fields is done here, for which UI mapping is not provided.
- **Clean up:** Data clean up must be done before sending it to next pipeline.

Figure 9-2 - Mapping Engine

Record Post Processor

Record post processor that is configured is executed to modify/discard data just before input into database.

Filter

Alarm inclusion/exclusion is applied to filter out some alarms. This step is applicable only for Alarm DataSource. For rest of the data load, it is ignored.

10 Record Processor Cache:

Generic Caching:

- Record post-processor cache is refreshed in every 5 minutes. Any changes in record post-processor needs 5 minutes to reflect.
- Alarm Filter cache is refreshed every 1 hour.
- Classes of mapped field for which code is specified is stored in cache. This is refreshed every 5 minutes.
- Cause standardization cache has standard causes mapping or regex mapping. This is refreshed every 30 minutes.

Controlled Caching:

Record processor uses lazy population technique to populate cache.

Managing cache size is an important aspect of caching. CAN application has so much data stored, that it is impossible or unfeasible to store all of these data in the cache. Therefore, a mechanism is required to manage how much data is stored in the cache. Managing the cache size is typically done by **evicting** data from the cache, to make space for new data using **least accessed eviction technique**. Least accessed eviction means that the cache values that have been accessed the least number of times are evicted first. The caches below use controlled caching for storing data:

- Event file format cache is refreshed every 5 minutes.
- Respective master tables are also stored as cache that is **not refreshed, so data manipulation in collections of any master tables requires restart of Record processor.** By default, this cache contains 500000 entries and stores ids.
- Any other configurations to this cache can be done in recordProcessorCache field of Config table currently this is not visible in UI. Field size denotes size of cache and fields to be stored in cache are inside fields.
  Note: Fields denoted in this config entry is used for caching, to avoid generic default values and use existing master table entry fields. List must be checked thoroughly before starting data load. Any changes to **RecordProcessorCache entry in config table requires restart of record processor.**

  {

"_id":ObjectId("63861da42510e142e21b4ca5"),
"key":"recordProcessorCache",
"value":{
"default":{
"size":NumberInt(100000),
},
"Cause":{
"size":NumberInt(50000),
"fields":[
"serviceAffecting",
"domain",
"priority"
]
},
"OfficeCode":{
"size":NumberInt(50000),
"fields":[
"officeCodePriority"
]
},
"Equipment":{
"size":NumberInt(50000),
}
}
}

11 Audit of Parsed Files

To monitor data load process audit of each parsed file is maintained in ParserAudit table. Parser Audit has following fields:

1. **fileParsed**: Name of the file that is parsed.
2. **completed**: This denotes the state of file Parsing. It is false when parsing is not complete and true when parsing completes.
3. **startTime**: Time at which parsing has started.
4. **endTime**: Time at which parsing is complete (This time is current time when parsing is in progress).
5. **dataSource**: The DataSource of file that is loaded. *Example:* Alarms, Performance counter etc.
6. **subDataSource**: This shows the sub datasource.
7. **aggregatedRecords**: This is number of records inserted after the duplicate records and filtered records are discarded.

8. **discardedRecords**: Records that are discarded from rejections involving discarded category. *Example:* no Equipment, No cause etc.
9. **discardedRecordsCategory**: This contains list of error code and corresponding number of discarded record count for all category that are discarded.
10. **duplicateRecords**: To maintain the uniqueness of data load, unique index can be added to a field or set of fields on DataSource table. This count signifies records that are not inserted due to existing entry in table.
11. **filteredRecords**: This count is the filtered count based on alarm inclusion or exclusion added.
12. **groupedRecords**: The number of records after grouping/consolidation.
13. **splittedRecords**: The number of records after splitting single record into multiple records.
14. **inputRecords**: The number of lines or records in file.
15. **totalRecords**: Number of records in total after split and group operation is applied on the file.
16. **trueRecords**: Number of records that are correct after applying split and group and the ones which are not discarded.