#### 

|  |
| --- |
| Persistence Layer Design Document |
| SPIRE Solution Definition |
|  |
|  |
|  |

|  |
| --- |
| **Versions** |
| **ID** | **Revision Date** | **Author** | **Description of Changes** |
| 1.0 | 1/8/2018 | Naresh Dindi | Initial Version |
| 1.1 | 1/24/2018 | Naresh Dindi | Added details about NB and Endorsement XML processing |
| 1.2 | 1/25/2018 | Naresh Dindi | Added Details about Out Of Sequence XML processing |
| 1.3 | 1/26/2018 | Pradeep Katkojvala | Added Commercial Auto content |
| 1.4 | 2/12/2018 | Pradeep Katkojvala | Added Commercial Property content |
| 1.5 | 2/14/2018 | Naresh Dindi | Incorporated feedback comments from Ravi, Tarun |
| 1.6 | 3/8/2018 | Naresh Dindi | Incorporated feedback comments from Dan, Ravi, Tarun |
| 1.7 | 3/22/2018 | Naresh Dindi | Updated the document based on inputs from Hadoop team |
| 1.8 | 3/30/2018 | Naresh Dindi | Updated the document based on comments provided by Rajaram |
| 2.0 | 4/1/2018 | Rajaram Nataraj | Several new sections are added. New contents are added in several sections. |
| 2.1 | 06/08/2018 | Rajaram Nataraj | Document is updated as per the new design approach. |
| 2.2 | 06/29/2018 | Srinivas Pratap | Updated document based on comments provided by Dan and Tarun. |
| 3.0 | 07/15/2018 | Rajaram Nataraj | Updated based on the feedback provided by Dan and Tarun. |

|  |
| --- |
| **Contributors** |
| **Name** | | **Date** |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |

|  |
| --- |
| **Approvers** |
| **Name** | | **Approval Date** |
|  | |  |
|  | |  |

Acronyms used in this document:

|  |  |
| --- | --- |
| **SOR** | System of Record |
| **SOT** | System of Truth – Data that are prcessed and ready to share with Data consumers, example: Data Hub |
| **PL** | Persistence Layer |
| **HDFS** | Hadoop Distributed File System |
| **SRC** | Audit record from the Source system |
| **PLIN** | Audit record from Persistent Layer Input |
| **PLHIST** | Audit record from Persisten Layer History output |
| **PLTRF** | Audit record from Persisten Layer TRF Extract creation |
|  |  |
|  |  |

Contents

[SPIRE Solution Definition 1](#_Toc518055189)

[1. Introduction 5](#_Toc518055190)

[1.1. Scope of Document 5](#_Toc518055191)

[1.2. Buisness Objective/Description 5](#_Toc518055192)

[2. Solutions Architecture 6](#_Toc518055193)

[2.1. Conceptual architecture 6](#_Toc518055194)

[2.2. Contextual Architecture 6](#_Toc518055195)

[2.3. Logical Architecture 7](#_Toc518055196)

[2.4. Physical Architecture 9](#_Toc518055197)

[2.5. System Integrations 9](#_Toc518055198)

[2.6. Development Tools, Related Software and Hardware 9](#_Toc518055199)

[2.7. Environments 9](#_Toc518055200)

[2.8. Development Environment 10](#_Toc518055201)

[2.9. Test Environment (QA) 10](#_Toc518055202)

[2.10. Prod Enviornment 11](#_Toc518055203)

[2.11. Source Details: 11](#_Toc518055204)

[2.12. Storage File Formats: 11](#_Toc518055205)

[2.13. scheduling 12](#_Toc518055206)

[2.13.1. HDP Schedulers 12](#_Toc518055207)

[2.13.2. Enterprise Schedulers 12](#_Toc518055208)

[2.14. Error Handling and Recovery Strategy 12](#_Toc518055209)

[2.14.1. Error Handling 12](#_Toc518055210)

[2.15. Reusable Objects 13](#_Toc518055211)

[3. Security and User Authentication 13](#_Toc518055212)

[3.1. Security Configuration 13](#_Toc518055213)

[3.2. User authentication 13](#_Toc518055214)

[4. Extract, Transform and load process of xml 14](#_Toc518055215)

[4.1. Extract XML- process 14](#_Toc518055216)

[4.1.1. ExecuteSQL Processor 14](#_Toc518055217)

[4.1.2. ValidateXML - XML validation against XSD 14](#_Toc518055218)

[4.1.3. SplitAvro 14](#_Toc518055219)

[4.1.4. ConvertAvroToJSON 14](#_Toc518055220)

[4.1.5. EvaluateJSONPath 14](#_Toc518055221)

[4.1.6. AttributestoJSON: 15](#_Toc518055222)

[4.1.7. ExtractRawXML - Custom processor to convert JSON object to XML file. 15](#_Toc518055223)

[4.1.8. UpdateAttribute: 15](#_Toc518055224)

[4.1.9. ExecuteScript: 15](#_Toc518055225)

[4.1.10 PutHDFS: 15](#_Toc518055226)

[4.1.11 RouteonAttribute: 15](#_Toc518055227)

[4.1.12 ReplaceText: 16](#_Toc518055228)

[4.1.13 AttributestoJSON: 16](#_Toc518055229)

[4.2. Extract Structured Data 16](#_Toc518055230)

[4.2.1 ExecuteSQL: 16](#_Toc518055231)

[4.2.2 PutHDFS: 16](#_Toc518055232)

[4.3. End to end data processing of incoming XML Data 17](#_Toc518055233)

[4.3.1. Property File and Its Purpose 18](#_Toc518055234)

[4.3.2. Spark Implementation 19](#_Toc518055235)

[4.3.3. Spark Submit Command: 20](#_Toc518055236)

[4.3.4. ExecuteProcess 20](#_Toc518055237)

[4.3.5. Overall end-to-end process 21](#_Toc518055238)

[4.3.6. Micro Batching (Dependency check) 21](#_Toc518055239)

[4.3.7. Policy Duplicate Check 25](#_Toc518055240)

[4.3.8. Agency and Account Data from APPS 27](#_Toc518055241)

[4.3.9. Look-up table for Coverage-Coverage Parts 29](#_Toc518055242)

[4.3.10. Change Data Capture for Data Hub 30](#_Toc518055243)

[4.3.11. Creation of Keys for Data Hub Entities 31](#_Toc518055244)

[4.3.12. Restart and Audit 31](#_Toc518055245)

[4.3.13. Business Logic to handle all Transaction Types 32](#_Toc518055246)

[4.3.13.1 New Business 32](#_Toc518055247)

[4.3.13.2 Endorsement(MTC) 33](#_Toc518055248)

[4.3.13.3 Out of Sequence 33](#_Toc518055249)

[4.3.13.4 Data slicing logic for various transactions 34](#_Toc518055250)

[4.3.13.5 Extract file Input to DataHub 34](#_Toc518055251)

[4.3.14. Premium Stitching 34](#_Toc518055252)

[5 Testing Plan 35](#_Toc518055253)

[5.1 Unit Testing Plan Details 35](#_Toc518055254)

[5.2 Integration Testing Details 35](#_Toc518055255)

[5.3 Performance Testing Details 35](#_Toc518055256)

[6 Non Functional Requirements 35](#_Toc518055257)

[7 Deployment Plan 36](#_Toc518055258)

[7.1 Deployment Details 36](#_Toc518055259)

[7.2 Depolyment Component List 36](#_Toc518055260)

[8 Risks and Mitigation Plan 36](#_Toc518055261)

[9 References 36](#_Toc518055262)

[9.1 Data Model details 36](#_Toc518055263)

[9.2 Data Specifications 36](#_Toc518055264)

[10 Appendix 37](#_Toc518055265)

[10.1 High Level Architecture Document 37](#_Toc518055266)

[10.2 Appendix 2 37](#_Toc518055267)

[10.3 Appendix 3 37](#_Toc518055268)

[11 Place Holder 2 37](#_Toc518055269)

[12 Misc Placeholder 37](#_Toc518055270)

# Introduction

## Scope of Document

The purpose of this document is to describe the overall project focus for Persistent layer workstream from a technical perspective. This includes architecture, source data complexity, extract, transform and load, different risk levels inherent in moving the underlying data to satisfy Persistent layer requirements.

This document includes the technical details and specicification to derive the solution to the project and achieve the goal of the SPIRE program.

This document also describes their precise implementation details required to satisfy the requirements of the project.

As this project uses Hortonworks Hadoop platform, this document explains and provides the technical details on how tools technologies from Hadoop stack – Hadoop, Hive, NiFi, Spark, Scala are used to achieve the project goal.

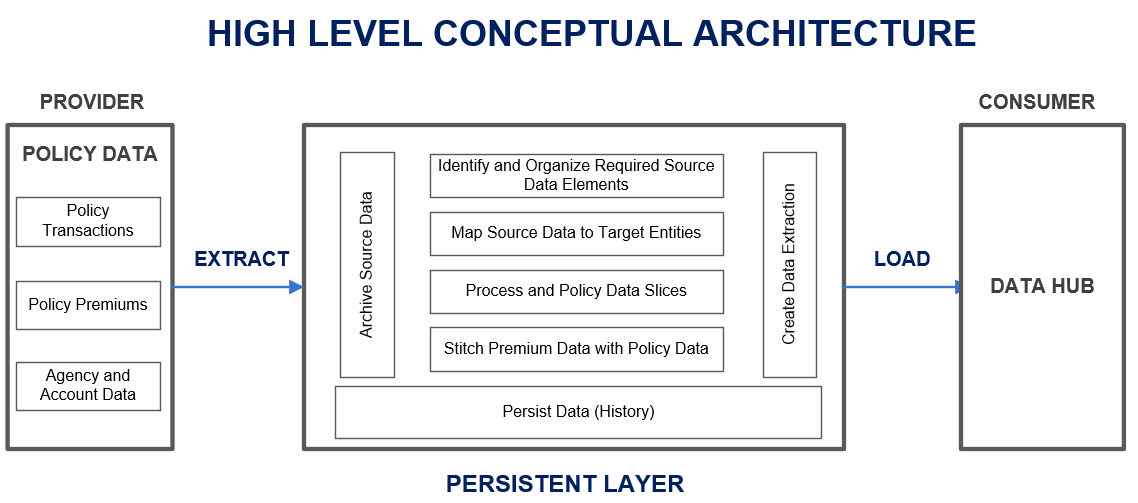
## Buisness Objective/Description

Please refer the following Project Charter document for the detail:

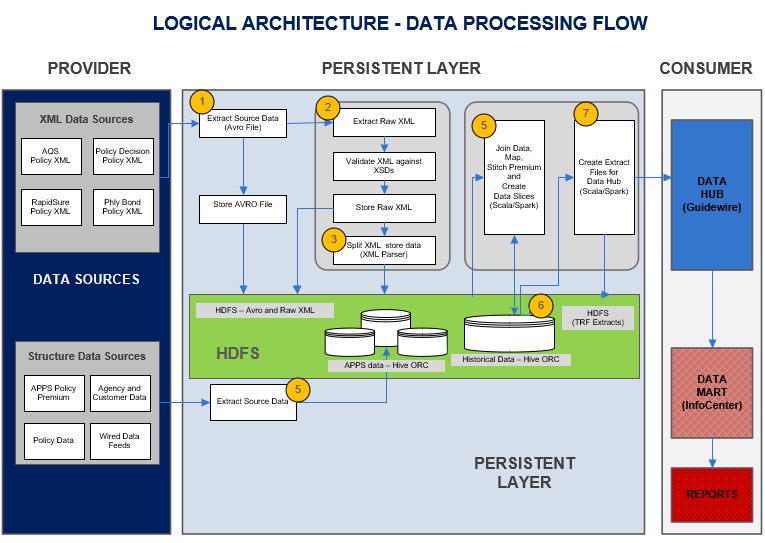
[SPIRE Project Charter Document](http://epmo.corp.tmnas.com/PWA/SPIRE%20Integration/Data%20%20Reporting/PHLY%20SPIRE%20-%20Project%20Charter%20-%20Data%20and%20Reporting.docx)

# Solutions Architecture

## Conceptual architecture

## Logical Architecture



1. Source data is extracted from **Policy Repository** Relational database and stored as Avro files thru NiFi. Refer the following section for details on NiFi flow.
2. Raw XML Data for each record is extracted from NiFi flow file. This XML data is validated against corresponding XSDs (XML Schema Definition) and stored in HDFS.
3. Raw XML is parsed, parsed data are organized and stored in HDFS for further processing using Scala and Spark-SQL.
4. APPS Data are extracted and stored in Hive ORC table thru NiFi.
5. Policy Data from HDFS are processed in Spark memory, mapped for target entity attributes and data is stitched with premium data from APPS Hive ORC.
6. Processed data (Histrical) stored in Hive ORC.
7. Data Hub TRF Extract files are created using Scala/Spark-SQL for each entities.

## Physical Architecture

Refer the High Level Architecture document in the Appendix for details.

(Add SharePoint link here)

## System Integrations

Persistent Layer is integrated with the Policy Repository, APPS and Data Hub.

## Development Tools, Related Software and Hardware

* Apache Nifi – Version 1.2.0.3.0.1.1-5 - Data Distribution and Processing System.
* HortonWorks Data Platform – version 2.6.2

Apache Ambari – 2.5.x

HDFS – 2.7.x

Hive version – 1.2.x

Spark version – 2.x

* SQL Server Drivers and Jar files:-

1. SQL Server Driver – com.microsoft.sqlserver.jdbc.SQLServerDriver
2. Jar file to enable Nifi connectivity to SQL Server – sqljdbc42.j

* Scala 2.10.x
* Databricks Jar - 2.x

## Environments

Below are the details for various environments and server names.

## Development Environment

Development is carried out in initial Sandbox (d00norapphad001).

**Hadoop Environment details:**

|  |  |
| --- | --- |
| HOST | D00NORAPPHAD001 |

**Ambari URL** :

<https://d00norapphad001.corp.tmnas.com:8080>

**Nifi URL:**

<https://d00norapphad001.corp.tmnas.com:9091/nifi/>

**Hadoop Configuration Folders details:**

Content will be updated once folder structure is finalized. Work is in progress.

This table will get updated once actual folders are created with Generic account s00hdpdev:

|  |  |  |
| --- | --- | --- |
| **Folder Name** | **HDFS/Local** | **What it contains** |
| data/xml/SOR/AQS  data/xml/SOR/RS  data/xml/SOR/PD  data/xml/SOR/PBS | HDFS | Contains XML’s of all sources. Inner directories can be segregated on source basis |
| XML Parsed Data/SOR | HDFS | XML parsed data will be in this folder. |
| /SOT/History | HDFS | Final History |
| data/APPS | HDFS | Data extracted from Historical tables would be placed inside this folder. |
| XPATH | Local | Xpaths of xml’s would be placed in this folder |
| Config File | Local | Config files would be placed in this folder |

**DB details:**

* Server Name: D01PISSQL014
* Database Name: Policy Repository

## Test Environment (QA)

**Hadoop Environment details:**

|  |  |
| --- | --- |
| HOST | Q00NORAPPHAD001/002 |

**Ambari URL** :

[https:// q00norapphad001.corp.tmnas.com:8080](https://d00norapphad001.corp.tmnas.com:8080)

**Nifi URL:**

[https:// q00norapphad001.corp.tmnas.com:9091/nifi/](https://d00norapphad001.corp.tmnas.com:9091/nifi/)

**Hadoop Configuration Folders details:**

/etc/hadoop/conf/core-site.xml

**DB details:**

* Server Name: Q01PISSQL006
* Database Name: Policy Repository

## Prod Enviornment

## <Looking for the link to the document. Can’t locate.>

## Source Details:

There are two different types of source data based on analysis on the source systems in scope. Details as below:

* **XML Data** - Source falling in this category initiate event driven message communication with the Enterprise Service Bus (ESB). There are Web Services which are invoked for this communication. The output of this data is in XML formats. This is transactional data which is net New or Changed data. The ESB also stores this XML into Policy Repository which is a SQL Server relational database. This data stream can be consumed directly from the ESB using Web Services and connectors or from the Policy Repository. It is proposed to consume this data from Policy Repository.
* **Structured Data** - for sources where data can be extracted directly from their relational databases. This data will be stored to HDFS using Nifi Dataflow in the form of JSON splits which is used for extract files creation.

## Storage File Formats:

Nifi after extracting data from heterogeneous sources like XMLs, Database tables etc. writes them to HDFS using various storage formats and Hive tables.

Below mentioned are the storage formats used as a part of solution:

1. **Avro**: Policy Repository data are extracted and stored in Avro format.
2. **XML**: Raw XML data extracted from Avro formatted data is stored. Nifi reads the XML messages along with MessageId and StatusDateTime to HDFS from SQL Server.
3. **ORC Formatted File**:
   1. **APPS** data extracted from the APPS DB are stored in Hive ORC format.
   2. **Historical Data**: Final processed historical data are stored in Hive ORC. This data will be used to create TRF Extract files.

## scheduling

## HDP Schedulers

Oozie will be used to schedule Hadoop jobs in Hortonworks platform.

|  |  |
| --- | --- |
| **Job Name** | **Description** |
| <Job 1> | Parses XMLs, make entries into Control table. |
| <Job 2> | Read Control table and select XMLs selectively using Micro Batching algorithm, Parse selected XMLs and store the data in HDFS. |
| <Job 3> | Read the parsed data from HDFS, Process (Map and create data slices) and create History records and insert into History table. |
| <Job 4> | Creates TRF Extracts. |
| <Job 5> | Create Audit and Control report |
| <Job 6> | ??? for NiFi |

## Enterprise Schedulers

Active Batch will be used to integrate with the systems outside of Hadoop platform:

|  |  |
| --- | --- |
| **Active Batch**  **Job Names** | **Job Description** |
| <Job A> | Creates TRF Extract files for all entities. |

## Error Handling and Recovery Strategy

## Error Handling

Nifi processors have a success and failure output relationship. For invalid data flowfiles are routed to a HDFS directory for analysis.

|  |  |
| --- | --- |
| **Processor Name** | **Error Occurrence Scenario and Handling.** |
| EvaluteJsonPath | To extract specific value from incoming json file we need to specify key’s absolute path. If absolute path is not correct it does not fetch the values. Error message is routed to a directory in HDFS. |
| RouteonAttribute | If incoming flow file’ attribute does not match with the attributes mentioned in the conditions of the  processor or if the condition is incorrect  then the incoming flow would be directed to HDFS error directory. |
| TransformXML | An XSLT file is applied to the flowfile XML payload. A new FlowFile is created with transformed content and is routed to the 'success' relationship. If the XSL transform fails, the original FlowFile is routed to the 'failure' relationship which is a HDFS directory. |
| AttributetoJSON | Attributes mentioned in this processor will be appended to the existing flow file .Failure to append those attributes or failure in conversion to JSON would cause the flow files to be directed to HDFS error  directory. |

## Reusable Objects

**Custom processor ExtractRawXML**– when we query a database to extract XML using Nifi processor QueryDatabaseTable the output of the resultant query (XML) will be converted to Avro format. So for further processing of XML (validate, transform and split) the input Avro must be converted back to XML format. This conversion from Avro to XML is done using a custom processor ExtractRawXML.

# Security and User Authentication

## Security Configuration

Refer the following document for details (Sharepoint link will be added to the following document. Currently can’t locate the document on Sharepoint):

**Persistent Layer Physical Architecture**

**NiFi Security capabilities:**

NiFi provides several different configuration options for security purposes. The most important properties are those under the "**security properties**" heading in the **nifi.properties** file. In order to run securely, the following properties must be set:

|  |  |
| --- | --- |
| **Property Name** | **Description** |
| nifi.security.needClientAuth | Set to true to specify that connecting clients must authenticate themselves. This property is used by the NiFi cluster protocol to indicate that nodes in the cluster will be authenticated and must have certificates that are trusted by the Truststores. |
| nifi.security.keystore | Filename of the Keystore that contains the server’s private key. |
| nifi.security.keystoreType | The type of Keystore. Must be either PKCS12 or JKS. JKS is the preferred type, PKCS12 files will be loaded with BouncyCastle provider. |
| nifi.security.keystorePasswd | The password for the Keystore. |
| nifi.security.keyPasswd | The password for the certificate in the Keystore. If not set, the value of  nifi.security.keystorePasswd will be used. |

## User authentication

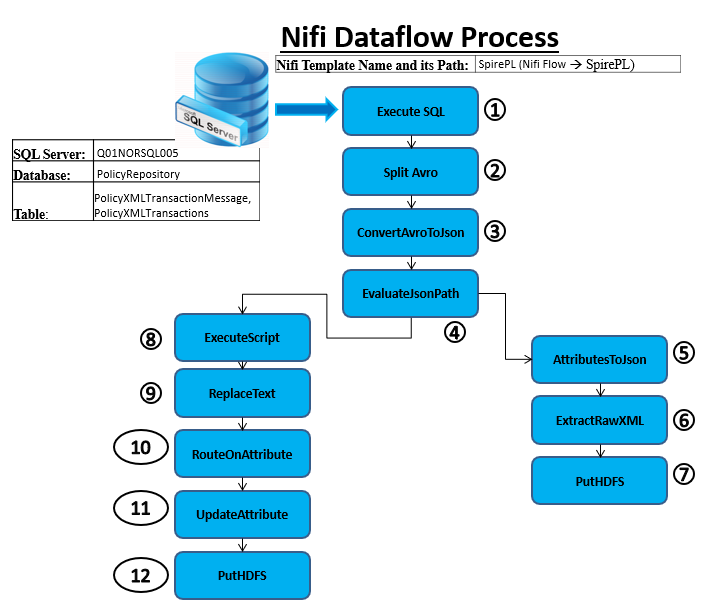
**<Ravi to fill this section.>**

# Extract, Transform and load process of xml

## Extract XML- process

Detailed configuration information for Nifi components used for data extraction:-

**Nifi Template Name – Nifi\SPIRE\_PL**



**Processors List**

### ExecuteSQL Processor

Establishes a connection to a database and pulls the required records from database by firing the SQL query mentioned in one of the properties of processors in AVRO format.

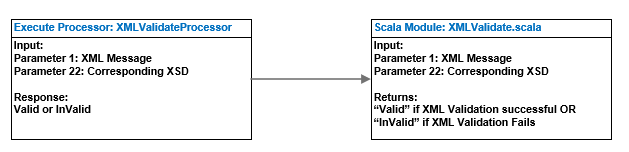
This processor makes use of a Database Connection Pooling Controller Service to obtain a connection to the database.

AVRO file contains a record-set with each record containing few string type columns and one xml content column.

Data will be pulled every hour from PR and APPS DB.

### ValidateXML - XML validation against XSD

NiFi Execute Processor will be used that calls the Scala module to validate the XML Message against the XSD Schema. XML Message and corresponding XSD will passed as an input to Scala module and Scala module returns “Valid” if XML validation is successful or returns “InValid” if XML validation fails.



### SplitAvro

Splits a binary encoded Avro datafile into smaller files based on the configured Output Size.(For handling the data record wise)

### ConvertAvroToJSON

Each AVRO record contains few string type columns and one xml content colum and this AVRO record is converted into a JSON object. This processor provides a direct mapping of an Avro field to a JSON field, such that the resulting JSON will have the same hierarchical structure as the Avro document. So the XML content will remain intact in one of the columns of the JSON flow file.Parsing will be done on XML content is the reason.

### EvaluateJSONPath

Evaluates one or more JsonPath expressions against the content of a FlowFile. The results of those expressions are assigned to FlowFile Attributes or are written to the content of the FlowFile itself, depending on configuration of the Processor.

### AttributestoJSON:

Generates a JSON representation of the input FlowFile Attributes. The resulting JSON can be written to either a new Attribute 'JSONAttributes' or written to the FlowFile as content.

### ExtractRawXML - Custom processor to convert JSON object to XML file.

For processing of XML (validate, transform and split) the input JSON object data must be converted back to XML format. This conversion from JSON object data to XML is done using a custom processor ExtractRawXML.

### UpdateAttribute:

Enables us to provide unique filenames to the FlowFiles along with the timestamp info:

### ExecuteScript:

Execute scipt is a processor is to execute ant script code.A code has been developed in Groovy Script to add CreatedDate and MessageTypeId tags into xml.These two fields are separate columns in source side but they are required to process XML’ and create an entity.

### PutHDFS:

Write FlowFile data to Hadoop Distributed File System (HDFS).Extracted RAW XML’s are stored into HDFS. The RAW XML files will be used for further processing.

### RouteonAttribute:

RouteonAttribute processor is used to segregate inflow files based on any attribute specified. For example based on source type, XML’s are routed to different flow as processing XML’s vary for each source system.

### ReplaceText:

ReplaceText processor is used to replace any text on incoming flow files with the specified text.

### AttributestoJSON:

AttributestoJSON converts all added attributes to json format files.

## Extract Structured Data

NiFi Template Name – NiFi\APPS\_QS\_BORDEREAU\_EXTRACT\_TABLES\APPS AND PATHNET TABLES

Dan’s suggestion to move APPS data to Hive ORC is pending. Currently, we are storing it on HDFS as per the current design. This is considered as a design change.

### ExecuteSQL:

Execute provided SQL select query. Query result will be converted to Avro format.

### PutHDFS:

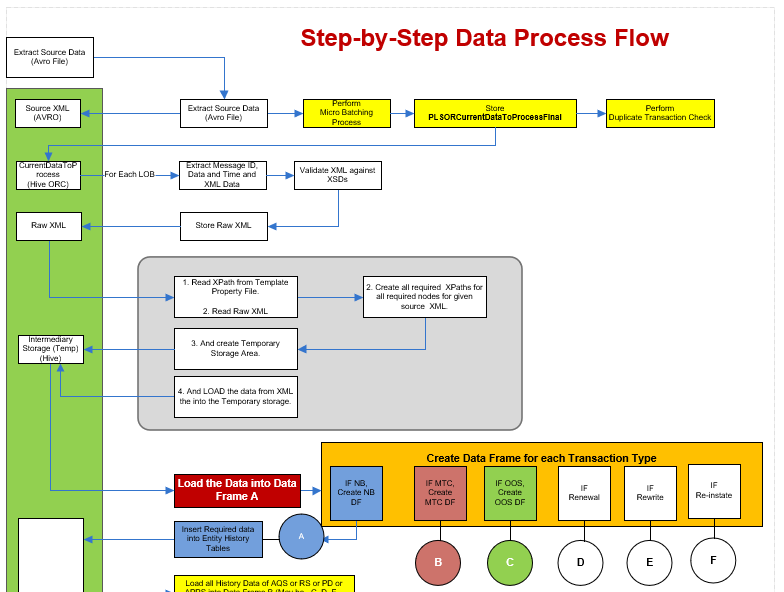
Write FlowFile data to Hadoop Distributed File System (HDFS).

Steps in structured data extraction:

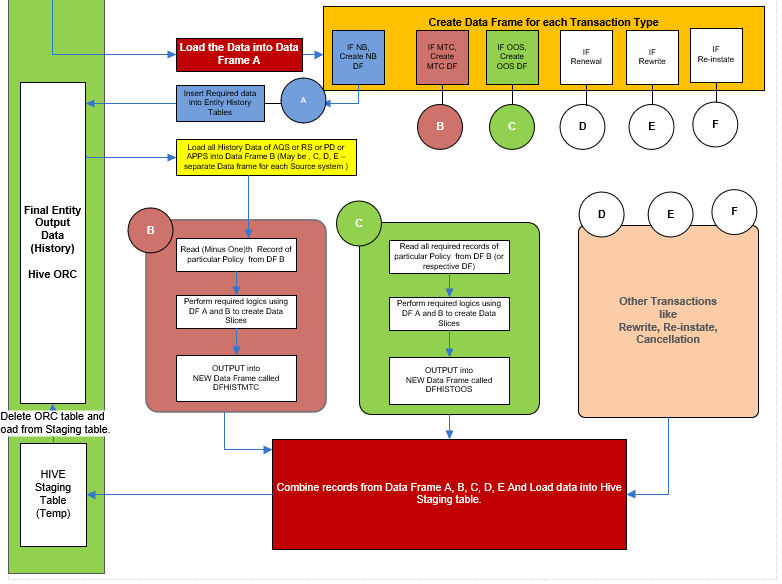
* + **ExecuteSQL**: Execute provided SQL select query. Query result will be converted to Avro format.
  + **SplitAvro**: Splits a binary encoded Avro datafile into smaller files based on the configured Output Size.
  + **ConvertAvroToJSON:** Converts a Binary Avro record into a JSON object.
  + **UpdateAttribute:** Enables us to provide unique filenames to the flowfiles.
  + **PutHDFS:** Write FlowFile data to Hadoop Distributed File System (HDFS).

## End to end data processing of incoming XML Data

Following diagram explains the end-to-end process once Micro Batching Process is performed. This process is followed as separate flow for each Line of Business.



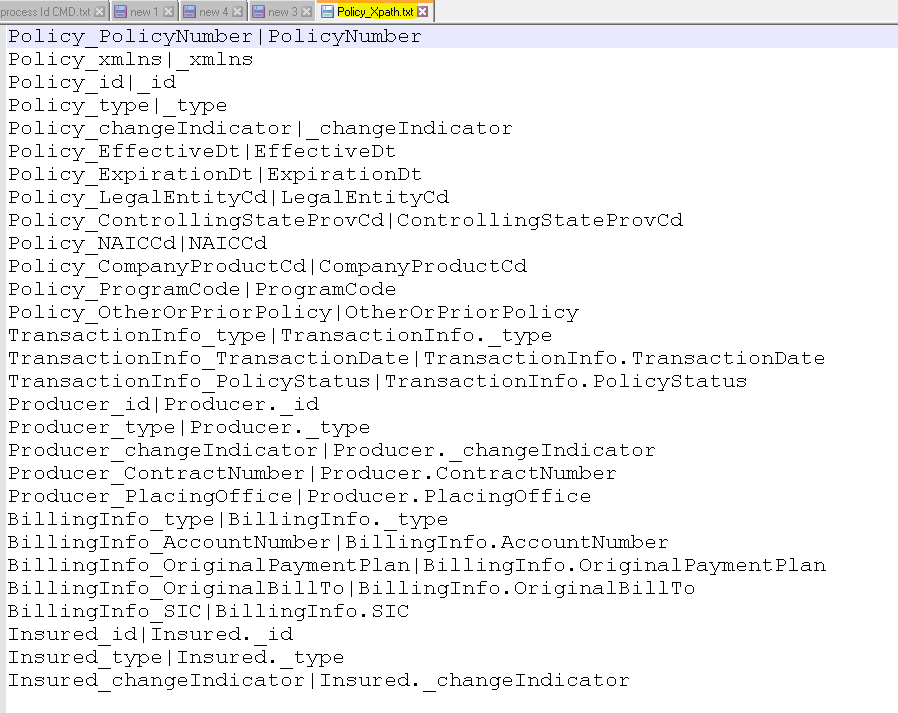
***Continuation of this diagram is given below:***

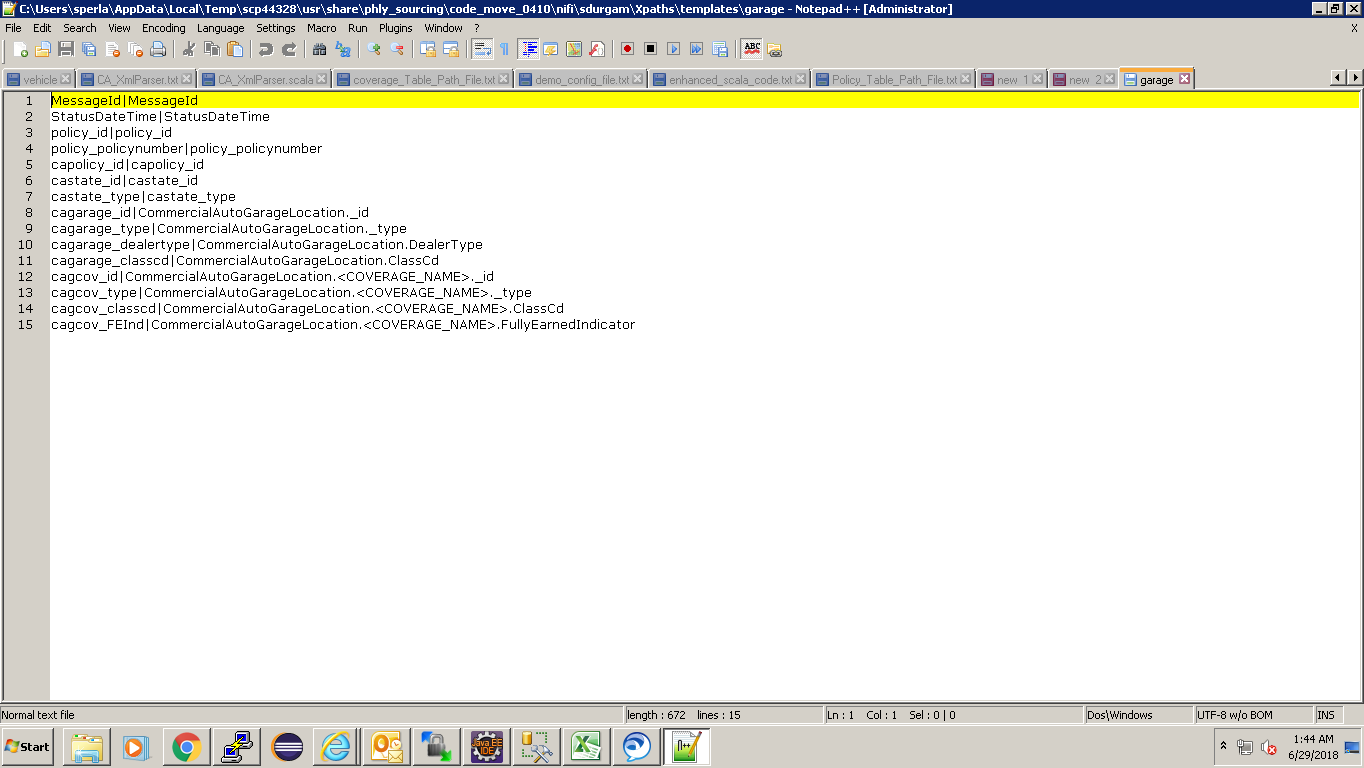


Note: Micro batching and Duplicate transaction check will be implemented in future LOBs. I will not be implemented for GL and CA.

## Property File and Its Purpose

Property file is a Excel file that contains Xpath of required attributes of an entity and based on Xpath data will be extracted from XML.





Property file would be placed in one of the folders in local file system of cluster, Spark code will read the property file to create entities.

## Spark Implementation

To generate entities for all the source systems Spark has been used.

The jar is created for the Spark code that creates entities on top of JSON splits generated from XML files by making use of the property file.

Jar file Location: **/usr/share/phly\_sourcing/spire\_pl/scripts/NewGlProj-0.0.1-SNAPSHOT-jar-with-dependencies.jar**

**Usage of Spark SQL**: Spark SQL is Spark’s interface for working with structured and semi structured data. Structured data is considered any data that has a schema such as JSON, Hive Tables and Parquet.  Schema means having a known set of fields for each record.  Semi structured data is when there is no separation between the schema and the data.

**Spark SQL Code Explanation**: Spark code reads the property file from its location then iterates all the sheets in the file. As one sheet is for one entity, Spark code will iterate a sheet row by row to create temporary tables based on the xpath. In each sheet the last line contains Spark SQL statement. Spark code will execute the Spark sql statement on temporary tables and generate a DataFrame for each entity. DataFrame of each entity will be stored in HDFS.

## Spark Submit Command:

Following command is used to execute the Spark code.

spark-submit --class <CLASS\_NAME><JAR\_NAME> <ARG0> <ARG1> <ARG2>

In above Spark Submit command following arguments are used.

args[0] : Input path of Prop file.

args[1]: XML input HDFS path.

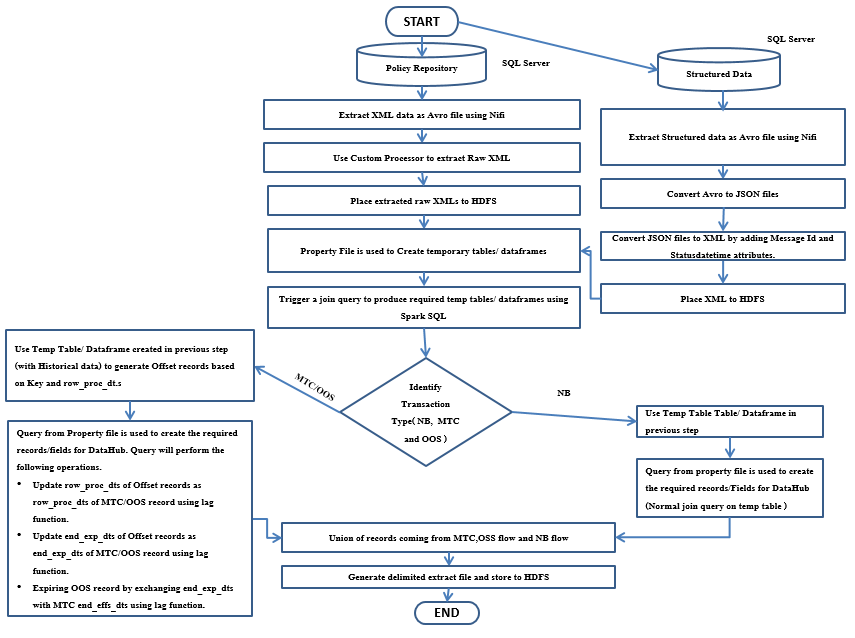
args[2]: Output path folder structure which contains final result.

## ExecuteProcess

Runs an operating system command specified by the user and writes the output of that command to a FlowFile. The whole end to end process is executed through a shell script.

|  |  |
| --- | --- |
| Processor Name: | **ExecuteGLSparkProcess** |
| Shell script executed by this processor: | Shell script that involks the Scala/Spark code using SPARK-SUBMIT |

## Overall end-to-end process – will be deleted



## Micro Batching (Dependency check)

## 

### Above diagram is explained in detailed manner below:

|  |
| --- |
| **Step by Step logic for Micro batching for incoming XML data**  This process explains how dependency check is performed on all records in the CURRENT BATCH of records to make sure that all records are sent by the source systems. If any one or more records are missing for any given transaction, respective records will not be processed and kept in the Hive table until all records become available.  *This process applies to source systems AQS, RS and PD, PBS only.* |
| XML Extracted from the sources will be stored in HDFS folder as follows:  /data/landing/SOR/AQS/418a5b74-5fa2-4ee2-9e35-342e70eac5a6\_PH12345.xml  /data/landing/SOR/RS  /data/landing/SOR/PD  /data/landing/SOR/PBS    **STEP 0:**  Back up **PLControlMaster** to use in case of failure or restart.  **STEP 1:**  Read XMLs along with other important key fields from Policy Repository, create a pipe delimiter txt file and store them in HDFS **//landing//SOR//** directory. Read XML with only good status, i.e., Status is null.  Ex.  File Name: **418a5b74-5fa2-4ee2-9e35-342e70eac5a6­\_PH12345.XML**  This file will be created every time batch of records are read from Policy Repository thru NiFi.  **STEP 2:**  Create a Hive Table (ORC) – **TMNAS.PLControlMaster.** Sample data are added to derive the logic for checking availability of all records that are required for particular transaction.  **STEP 2a:** Read all CURRENT RECORDS from HDFS SOR Raw folder and parse XML message and populate all the columns except Process flag column.  **STEP 2b:** Invoke “Duplicate check solution” to get the status from AQSInterfaceV3. If Status is Error, populate Process Flag to “X” and populate “N” to all other records.  **STEP 2c:** Insert all above records information into **TMNAS.PLControlMaster** table.  **STEP 3**:  Read all data from **TMNAS.PLControlMaster** table where Process Flag =” N” store it into a Data Set, **DataToProcessDS01**.  **STEP 4**:  Read all data from **TMNAS.PLControlMaster** table where Process Flag =” Y” or “X” store it into a Data Set, **DataAlreadyProcessedDS02.**  **STEP 5**:  Filter all data from Data set, **DataToProcessDS01** where TransType = “NB”, and set Process Flag field to “Y” and store them in a new dataset **DataToProcessDS0101**.  (Use Map function)  **STEP 6**:  6a. Filter all data from the dataset, **DataToProcessDS01** where **TransType NOT = “NB”** and **Group By** Policy number and Policy ID.  6b. For each group,   * Count the number of records in the group. * If number of records = 1, and if OOSType is Null, set Process Flag field to “Y” and store them in a new dataset **DataToProcessDS0102**. * If number of records Not = 1 (it means there are more than one transaction for a given Policy in the same batch),   + check there are no missing “Change Seq Nbr” AND   + check, there are all records (OOS Trans # = 1 of 3, 2 of 3 and 3 of 3) exits if OOSType is NOT NULL.  And then set Process Flag field to “Y” and store them in a new dataset **DataToProcessDS0103**. * Move rest of the records to dataset **DataDependedRecMissingDS0104**.   **STEP 7**:  Now combine datasets (apply Dataset UNION function) -  **DataToProcessDS0101,  DataToProcessDS0102,  DataToProcessDS0103 and** store them ina new dataset**, DataToProcessALLDS.**  NB: Now the dataset **DataToProcessALLDS** will have all required records to process in the given Batch.  **STEP 8**:  Read all CURRENT RECORDS from //landing/SOR folder and store them in a data frame, RawDataToProcessDF. And, then register this data frame as a Temp table, **PLSORRawCurrentTemp**.  Register the dataset, **DataToProcessALLDS as another Temp table, DataToProcessALLTemp.**  Use Spark-SQL to select all records from **PLSORRawCurrentTemp** where MessageTypeID matches the the MessageTypeID in **DataToProcessALLTemp** and store the final result in a data frame, **PLSORCurrentDataToProcessFinal**  Now, data frame, **PLSORCurrentDataToProcessFinal** will contain all REQUIRED DEPENDENT RECORDS to process in the current batch of records.  **STEP 9:**  UNION all DataSets (  **DataToProcessALLDS = all records with Flag “Y”**  **DataDependedRecMissingDS0104 = all current records with Flag “N”**  **DataAlreadyProcessedDS02 )**  **And**  Register as a new TEMP table called**, PLControlMasterTemp**  **STEP 10:**  Truncate **PLControlMaster.**  Select all from **PLControlMasterTemp and** INSERT INTO **PLControlMaster.**  **STEP 11**:  Perform Audit and Reconciliation for Records read, and Records written. |
|  |

## Policy Duplicate Check

|  |
| --- |
| Duplicates can exist when a policy has be approved and issued in AQS but has not process successfully to APPS. When this occurs, someone can re-process the policy with a change to AQS and reissue the policy. When this occurs, the keys are updated and are different but it is considered a duplicate policy.  AQS calls Policy Repository (PR) web services (Web method Service) whenever a new policy transaction is being issued by the user. With the call AQS supplies policy id and CL or SL indicator. Then PR web service calls getSLPolicyDetails() or getCLPolicyDetails() method to get policy transaction XML. The issued policies transactions also get processed through APPS ETL-Interface. In case of error, ETL-interface job flag the failed transactions. However, PR does not check for the error status. For more details, please refer to PR design document.  **Solution**    Configure DBCPConnectionPool for each environment.  Database Name: AqsInterfaceV3  Table Name: INT\_PoliciesToImportV3  On a daily basis, this solution requires at least five additional nifi processors for dup check within PL layer.  The following steps describe this solution:   1. The PL layer consumes policy transaction xml from PR and creates control record with status = “Pending”. This step is an existing step within PL layer. 2. After creating control record, the Nifi processors extracts key attributes from the control table. These key attributes are policy number, policy stat id ( STAT XML ) and policy id. 3. After extracting required attributes, the Nifi extract processors then sends these attributes to CL & SL specific Nifi processors. 4. The CL & SL Nifi processor performs SELECT SQL using DBCPConnectionPool Controller service. This service requires to be configured for AQS CL & AQS SL SQL Servers. 5. After executing SQL both processors hands over the results set to Nifi extract attribute processors. This processor extracts error and error\_desc fields. If both fields are not NULL then set status=”error”.    1. For OOS, the Nifi processor needs to evaluate “OOSControl\_StepType” attribute equal to “Change”. If error and error\_desc fields are not NULL then set status=”error” for all ( backoff, reapply and OOS) policy OOS records. 6. Spark component needs to ignore policies transactions with status=”error”. |

## Agency and Account Data from APPS

|  |
| --- |
| This section provides the solution to get the Agency Data from PATHNet DB and Account Data from AccessPhly DB (APPS DBs).  **Below is the reference diagram to read these data:**     * Create JDBC connection to APPS DBs. * Get Agency data thru the Stored Procedure, EIA\_Agency\_Sync\_Select: * **For Agency Full load:**   exec EIA\_Agency\_Sync\_Select   * **For Agency Incremental load based on your last successful datetime**   exec EIA\_Agency\_Sync\_Select @ChangesStartDate='2018-05-24 10:58:13.977'   * Get Account data thru the Stored Procedure, EIA\_Account\_Sync\_Select: * **For Account Full load**   exec EIA\_Account\_Sync\_Select   * **For Account  Incremental load based on your last successful datetime**   exec EIA\_Account\_Sync\_Select @ChangesStartDate='2018-05-24 10:58:13.977  **Following approaches are in consideration to read the AGENCY and ACCOUNT data:**   * *Reading data thru the Stored Procedure is not straight forward in NiFi.* * Following is a POC sample code that are planning to test. If POC is successful, we use this approach otherwise new approach need to be identified.   **Execute SQLSERVER stored procedure through SQLCMD commend .**   1. Install MsSqlCmdLnUtils  on your nifi server . 2. Create sample table .   CREATE TABLE test\_proc (track\_id int,username varchar(50),course\_no varchar(50),details varchar(50),status\_code int,status\_desc varchar(50));   1. INSERT Sample data.   INSERT INTO test\_proc (track\_id,username,course\_no ,details ,status\_code,status\_desc) SELECT '20585','A3843UP', '00009934','Testing','10000','calling proc through nifi';  cid:image007.jpg@01D3FFD9.CF77FE10   1. Create sample stored procedure for testing .Here we are passing three parameters as input and its return three output values.   CREATE Procedure testing\_proc @track\_id int,@username varchar(50),@course\_no varchar(50)  AS  SELECT details,status\_code,status\_desc FROM test\_proc  WHERE track\_id = @track\_id and username = @username and course\_no = @course\_no   1. Execute below command .   sqlcmd -S localhost -d sample -U admin -P admin -Q "EXEC testing\_proc @track\_id = '20585' ,@username = 'A3843UP' ,@course\_no = '00009934'"  cid:image008.jpg@01D3FFD9.CF77FE10  NOTE : Install **MsSqlCmdLnUtiles** on nifi server and call sqlcmd through nifi ExecuteProcess processor . |

## Look-up table for Coverage-Coverage Parts

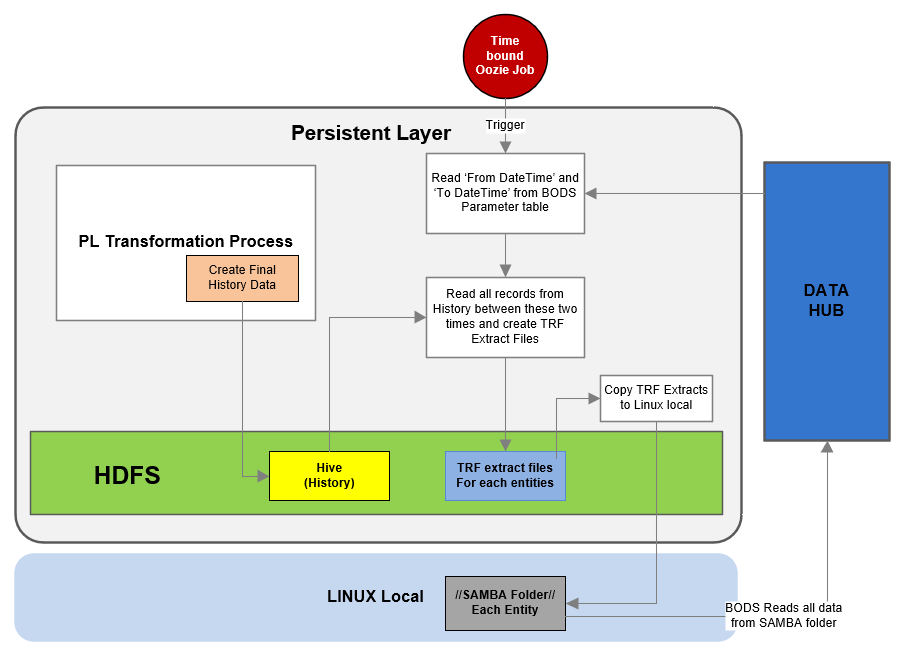
Static Look-up table will be created for Coverage-Coverage Parts in Hive ORC.

**Coverage-CoverageParts**:

|  |  |  |  |
| --- | --- | --- | --- |
| **LOB** | **SRCSystem** | **CoverageNode** | **Coverage Part** |
|  |  |  |  |
|  |  |  |  |

## Change Data Capture for Data Hub

Following steps explains the Change Data Capture logic for Data Hub (BODS):



1. Active Batch job will trigger Ozzie job.
2. PL continue to process all incoming data thoughout the day and populates History tables (Hive ORC) for all entities.
3. Time bound Oozie job will trigger TRF Extract file creation process once a day for a given time.
4. Read ‘From DataTime’ and ‘To DateTime’ from BODS Parameter table.
5. Read all records from History tables between these two times and create TRF Extract files into HDFS folder.
6. Copy all TRF Extract files from HDFS location into Linux local folder where SAMBA service is enabled.
7. BODS reads all files from **//Physical server name (or Static IP)//SAMBA//Entity folders.**
8. Once BODS completes reading all files from SAMBA folder, all these files will be deleted from this folder.

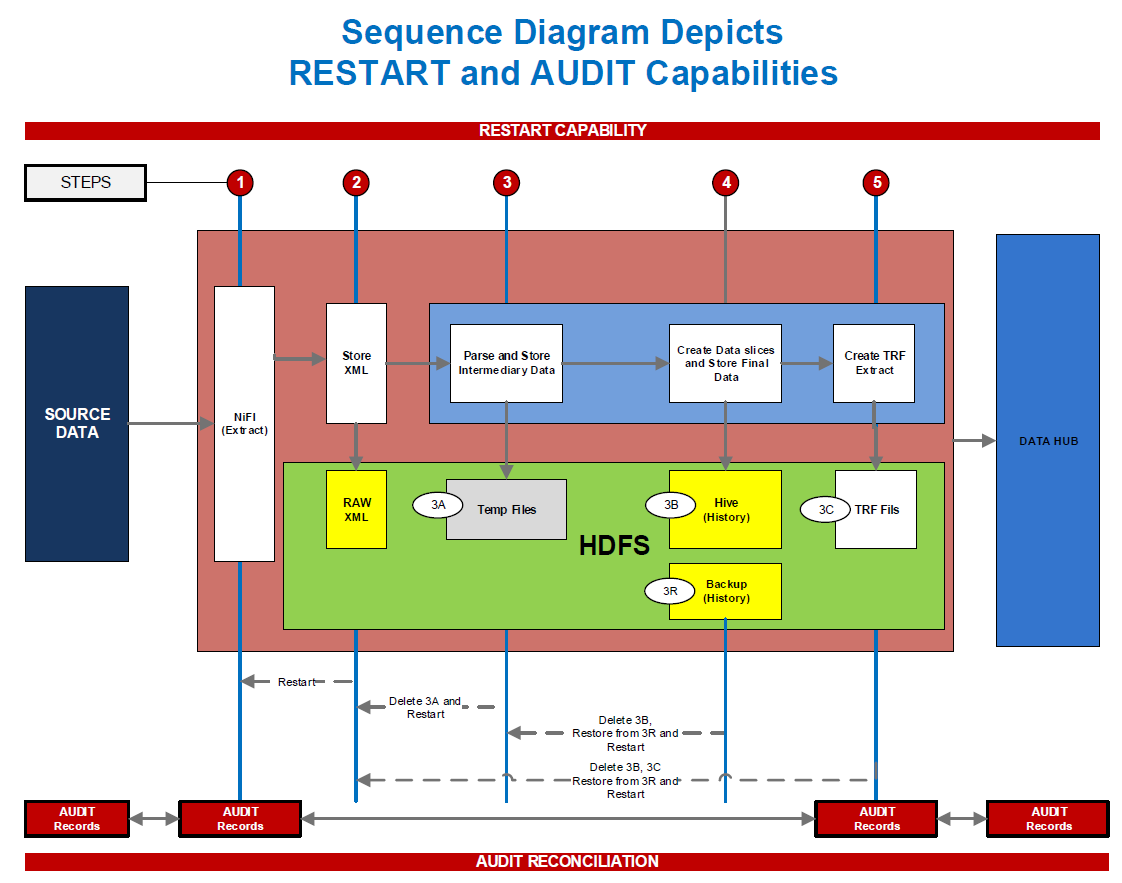
## Creation of Keys for Data Hub Entities

Refer the attached document for details:

[**DataHub Dupkey solution Design:**](http://epmo.corp.tmnas.com/PWA/SPIRE%20Integration/Data%20%20Reporting/DataHub%20Dupekey%20solution%20Recommendation.pdf)

## Restart and Audit

Following will explain the Restart capability in case of failure in the process or issue in the source code:



## Audit Control process

### 

**Audit Control Table (Hive ORC table):**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Audit**  **Source** | **DBName** | **Source**  **System** | **# of Records Read** | **Current DateTime** | **Temp1** |
| **SRC** | **PR** | **AQS** | **1233** | **06012018:142344** |  |
| **SRC** | **PR** | **RS** | **45** | **06012018:142344** |  |
| **PLIN** | **PR** | **AQS** | **1233** | **06012018:142344** |  |
| **PLHIST** | **PR** | **AQS** | **1277** | **06012018:142344** |  |
| **PLTRF** | **PR** | **AQS** | **1277** | **06012018:142344** |  |

* Audit source: Audit source is the source for which Audit record is created.   
  Example: SRC – PR or APPS or PATH.
* DBName: Data base from which record is read.   
  Example: PR – Policy Repository, APPS – APPS DB, PATH
* When record is read from the source system (PR or APPS) thru NiFi, insert the audit record for SRC.
* After records are inserted into Master control table, create a audit record for the Audit Source, PLIN.
* Audit will be performed between Audit Source, SRC and PLIN.
* Once records are processed and inserted into History table, create a audit record for PLHIST.
* Once TRF extract files are created, create aaudti record for PLTRF. BODS will use this record to compare and perform the audit between PL and BODS.
* Audit comparison can’t be performed between the Audit Sources PLIN and PlHIST.

## Restart Process

### 

### Restart will be performed as follows in case of failure at any given point in the ETL Process:

Entire ETL process is broken into Five steps and restart of jobs will be followed as described below:

**STEP 1**: Pulling the data from surcess thru NiFi.  
If any failure in this step, clear flow file and from content repository and restart this step after fixing the issue that caused the failure.

**STEP 2**: Proceed to this step only when step 1 is successful.   
Read XMLs from retrieved records and store them in HDFS location.   
If this step fails, fix the issue,

* + 1. Clear the HDFS location
    2. Restart from this step.   
       **OR**
    3. Update the date and time stamp appropriately to pull the right CDC records.
    4. Restart from Step 1.

**STEP 3:** Proceed to this step only when step 2 is successful.   
Partial XML Parse: Parse the XML to populate the Master Control Table.  
When this step fails,   
delete records that are inserted for this batch from the Master Control table and restart this step.

**STEP 4**: Proceed to this step only when step 3 is successful.   
Perform Duplicate Transaction Check.  
If this step fails, restart this step after correcting the issue.

**STEP 5**: Perfom Micro Batching – Dependency check.  
If this step fails, restart this step after correcting the issue.

**STEP 6:** Full XML Parse:Parse XML fully and store the data in intermediary (temporary) HDFS folder.  
If this step fails, delete all the contents from intermediary folder and restart this step after addressing the issue.

**STEP 7**: Process data from intermediary HDFS folder, create data slices and insert processed data into **History Hive ORC** table.  
If this step fails, delete recrods that are inserted for this batch from Hive table and restart this step.

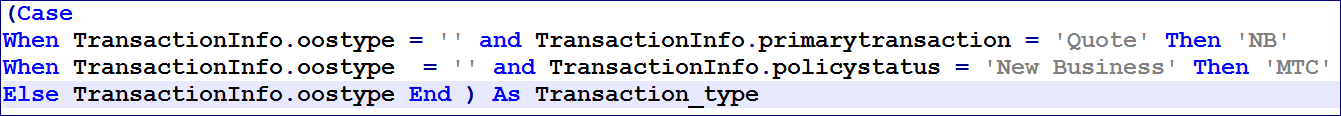
**STEP 8**: Create TRF Extract files.  
If this step fails, restart this step.

## Business Logic to handle all Transaction Types

**Note**:Only some Business transacitons for AQS system are covered in this section. Rest of the transactions for AQS and other systems will be addressed during the implementation time.

New Business, Renewal,Endorsement, Reinstament and Cancellation scenarios are identified based on the TransactionInfo node from the source XML. –

The below logic shows an example to identify particular transaction:



We create a temp table adding the above identified **transaction type** column in the base query which specifies whether the record is NB, MTC, Backoff, OutOfSequence or Reapply. The transaction type column is shown in the table for our understanding however it is not available in final entities.

Below table shows the temp table records for GL\_PolicyLine

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **pol\_line\_key** | **pol\_key** | **row\_proc\_dts** | **end\_eff\_dt** | **end\_exp\_dt** | **transaction\_type** |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:51:13.2505096-05:00 | 3/1/2018 | 3/1/2019 | NB |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:59:15.6729992-05:00 | 5/1/2018 | 3/1/2019 | MTC |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 15:08:09.4636803-05:00 | 5/1/2018 | 3/1/2019 | BackOff |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 15:08:17.8700892-05:00 | 4/1/2018 | 3/1/2019 | OutOfSequenceChange |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 15:08:22.7608475-05:00 | 5/1/2018 | 3/1/2019 | Reapply |

Below examples show logic for few of the transaction types

## New Business

SQL query from the property file will fetch the New Business Record from the temp table(s) based on **transaction\_type** ‘NB’. Query uses simple join conditions on various temp table(s) to get the required data which is available in various nodes of XML file. The results of the SQL query will be stored directly in delimited file which will be sent as a source to the extraction layer of DataHub.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **pol\_line\_key** | **pol\_key** | **row\_proc\_dts** | **end\_eff\_dt** | **end\_exp\_dt** | **transaction\_type** |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:51:13.2505096-05:00 | 3/1/2018 | 3/1/2019 | NB |

## Endorsement(Mid Term Change - MTC)

Once we get a MTC record the query will generate an offset record based on the previous transaction.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **pol\_line\_key** | **pol\_key** | **row\_proc\_dts** | **end\_eff\_dt** | **end\_exp\_dt** | **transaction\_type** |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:51:13.2505096-05:00 | 3/1/2018 | 3/1/2019 | NB |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:59:15.6729992-05:00 | 3/1/2018 | 5/1/2018 | OffSet |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:59:15.6729992-05:00 | 5/1/2018 | 3/1/2019 | MTC |

## Out of Sequence

Once we get an out of sequence record

This logic will be applied on the final entity, by taking the key columns and apply the LEAD function over the partition of the entity key column and create an offset record. This record will be inserted into the history table based on the particular day entity key column.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OOS OffSet |  |  |  |  |  |
| **pol\_line\_key** | **pol\_key** | **row\_proc\_dts** | **end\_eff\_dt** | **end\_exp\_dt** | **transaction\_type** |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:51:13.2505096-05:00 | 3/1/2018 | 3/1/2019 | NB |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:51:13.2505096-05:00 | 3/1/2018 | 4/1/2018 | OffSet |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 15:08:17.8700892-05:00 | 4/1/2018 | 3/1/2019 | OutOfSequenceChange |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 15:08:17.8700892-05:00 | 4/1/2018 | 5/1/2018 | OffSet |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:59:15.6729992-05:00 | 5/1/2018 | 3/1/2019 | MTC |

Expiring the OOS record

OOS is expiring based on latest MTC end\_eff\_dt

1. end\_eff\_dt of mtc record will become the end\_exp\_dt of OOS record

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Expiring OOS |  |  | |  |  |  |
| **pol\_line\_key** | **pol\_key** | **row\_proc\_dts** | | **end\_eff\_dt** | **end\_exp\_dt** | **transaction\_type** |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 15:08:17.8700892-05:00 | 4/1/2018 | | 3/1/2019 | OutOfSequenceChange |

## Data slicing logic for various transactions

Refer the document attached for the Data Slicing and Premium stitiching logic for various transactions:



**Note**:Only some Business transacitons for AQS system are covered in this section. Rest of the transactions for AQS and other systems will be addressed during the implementation time.

## Extract file Input to DataHub

Extract file is generated with all transactions types along with offset records using Spark code from the final table(having all history records) and sent to Datahub.

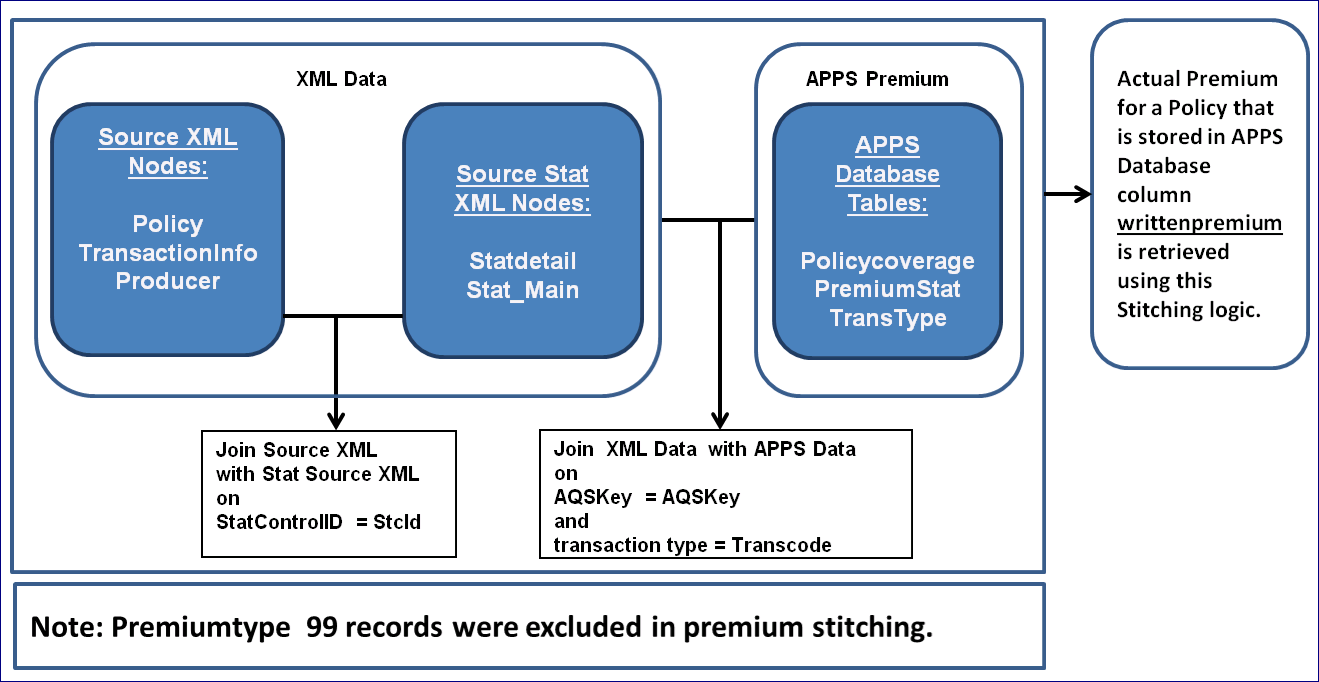
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **pol\_line\_key** | **Pol\_key** | **row\_proc\_dts** | **end\_eff\_dt** | **end\_exp\_dt** | **transaction\_type** |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:51:13.2505096-05:00 | 3/1/2018 | 3/1/2019 | NB |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:51:13.2505096-05:00 | 3/1/2018 | 5/1/2018 | OffSet |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:59:15.6729992-05:00 | 5/1/2018 | 3/1/2019 | MTC |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 14:51:13.2505096-05:00 | 3/1/2018 | 4/1/2018 | OffSet |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 15:08:17.8700892-05:00 | 4/1/2018 | 3/1/2019 | OutOfSequenceChange |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 15:08:17.8700892-05:00 | 4/1/2018 | 5/1/2018 | OffSet |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 15:08:22.7608475-05:00 | 5/1/2018 | 3/1/2019 | Reapply |
| AQS-PHPK1781060-716FF9BB-9388-43AF-A6B7-C32BCC02322D-GL | AQS-PHPK1781060 | 2018-03-01 15:08:09.4636803-05:00 | 5/1/2018 | 3/1/2019 | BackOff |

## Premium Stitching

Due to discrepancies/mismatch in regards to the premium data that is recalculated in APPS vs. current PAS it is required to bring in the historical premium details from both APPS and legacy PAS into GWDH to ensure the earnings process is kept whole. In the current environment, scenario associated with premium mismatches due to rounding are manual adjustments (including OOS), mid-term cancellations (especially if short-rated), and other scenarios. Using APPS as a pass-through for billing requires storing APPS and PAS data together because the proration techniques in APPS and AQS are quite close, but not the same, therefore it is important to land both systems premium and change dates for each policy. Also, APPS is the source for premium transactions.

Z\_APPS\_PREM\_AMT and Z\_PAS\_PREM\_AMT are the fields added into all the transaction tables of DataHub to store the premium data from APPS and PAS source systems respectively. PAS premium is obtained from Transaction Premium of source stat xml which is joined with source xml on statcontrolId by excluding premium type 99 records.

Below table shows the logic implemented in Premium Stitching



# Testing Plan

## Unit Testing Plan Details

### N/A

## Integration Testing Details

### N/A

## Performance Testing Details

### N/A

# Non Functional Requirements

Refer the attachment for details:

<http://teamtmnas.corp.tmnas.com/it/arch/Collaboration%20Library/Project%20Work/PHLY/SPIRE/Enterprise%20Architecture%20Design%20and%20Requirements_Spire_Group_1.docx>

# Deployment Plan

## Deployment Details

## Depolyment Component List

TBD

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.Nbr** | **Component Physical Name** | **Component Type** | **Deployment Location** | **Remarks** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

# Risks and Mitigation Plan

TBD

# References

## Data Model details

N/A

## Data Specifications

Refer the APPENDIX section for details.

# Appendix

## High Level Architecture Document

Refer the attachment for details:

[**High Level Architecture Document**](http://epmo.corp.tmnas.com/PWA/SPIRE%20Integration/Data%20%20Reporting/High%20Level%20Architecture%20Document.docx)

## Appendix 2

## Appendix 3

# Place Holder 2

# Misc Placeholder