Pig Ingestion Using XML

R Prasannavenkatesh
IMT2019063
prasannavenkatesh.ramkumar@iiitb.ac.in

Channamsetty Sivani
IMT2019020
channamsetty.sivani@iiitb.ac.in

***Abstract*—The project focuses on creating a data ingestion pipeline that can handle XML data with a predefined schema. The pipeline begins by taking in the XML script, which is then parsed using Python. The parsed XML data is used to generate a basic Pig script, which is a high-level scripting language designed for large-scale data processing. The Pig script is then executed on a Hadoop cluster or other distributed computing framework to process and analyze the data. The output of the Pig script and the Pig script itself will be returned to the user, allowing them to easily view and manipulate the data.**

***Index Terms*—xml, pig, react, fastapi, parser, abstraction**

I. Introduction

The ability to process and analyze large amounts of data has become increasingly important in today's world, with organizations of all sizes and across all industries relying on data-driven insights to make informed decisions. However, handling vast amounts of data can be a challenging and time-consuming task, particularly when dealing with data in a variety of formats.

This project aims to address these challenges by creating a data ingestion pipeline that can handle XML data with a predefined schema. The pipeline will use Python to parse the XML data and generate a Pig script, which is a high-level scripting language designed for large-scale data processing. The Pig script will then be executed on a distributed computing framework such as Hadoop, enabling the processing and analysis of large amounts of data in a scalable and efficient manner.

By automating the process of generating Pig scripts from XML data, this project aims to improve the efficiency and accuracy of large-scale data processing. The data ingestion pipeline and associated frontend and backend will allow users to quickly and easily process and analyze their data, saving time and reducing errors.

II. Problem Definition

The problem statement of the project is to create a data ingestion pipeline that automates the process of generating Pig scripts from XML data, enabling users to process and analyze their data quickly and easily. The pipeline will use Python to parse the XML data and generate a Pig script, which will be executed on a distributed computing framework such as Hadoop.

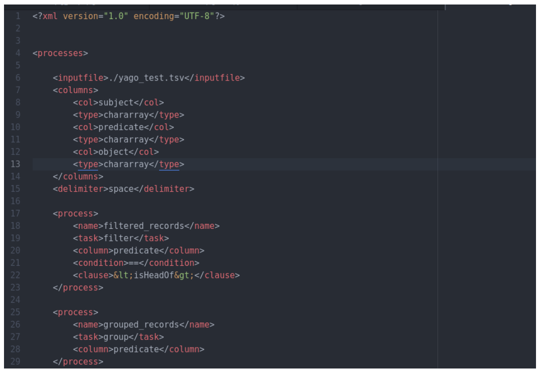
III. Approach

A. XML file

In the XML file we write a pig script workflow with multiple processes defined. Each process corresponds to a specific operation or transformation performed on the input data. We first specify XML version and encoding used. Below are the tags definitions, like for purpose we are using a particular tag which helps user to write xml files.

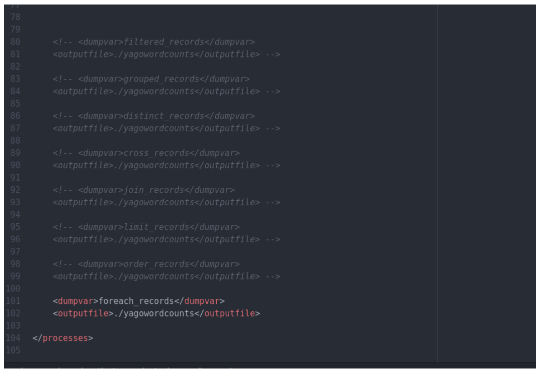
- <*processes*> : The root element that contains all the process definitions.
- <*inputfile*>./yago_test.tsv</inputfile> : Specifies the input datafile path. Here yago_test.tsv is dataset and we are importing it from current directory.
- <*columns*>: Defines the column names and their types for the input data.
- <*col*>: Specifies a column name of data.
- <*type*>: Specifies the type of the corresponding column.
- <*delimiter*>space</*delimiter*>: Specifies the delimiter used in the input file.
- <*process*>: Represents a specific operation or transformation process.
- <*name*>: Specifies the name of the process.
- <*task*>: Specifies the task to be performed, such as filter, group, distinct, cross, join, limit, order, or foreach.
- <*column*>: Specifies the column used in the process.
- <*column1*>: Specifies the column from the second table used in the join operation or can use for another column.
- <*condition*>: Specifies the condition for the filter operation.
- <*clause*>: Specifies the value or condition for the task.
- <*table*>: Specifies the table or input data used in the process.
- <*table1*>: Specifies the second table or input data used in the process (for cross or join operations).
- <*variable*>: Specifies a variable name for the output of the process or can use for any other variable present.
- <*variable1*>: Specifies if any other variable is present.
- <*dumpvar*>: Specifies the variable to be dumped or stored in an output file.
- <*outputfile*>: Specifies the output file path for the dumped data.

Each process represents a step in the Pig script, and the commented sections at the end show examples of how to dump the results to an output file. After we parse this xml file to parser, pigscript will be generated as shown in the following figure.



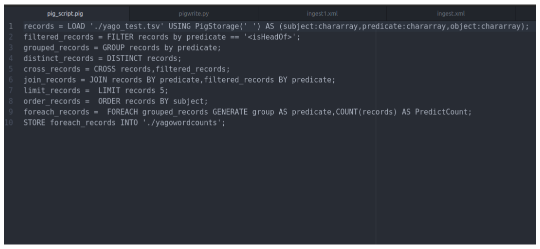
```xml
<?xml version="1.0" encoding="UTF-8"?>


<processes>

    <inputfile>./yago_test.tsv</inputfile>
    <columns>
        <col>subject</col>
        <type>chararray</type>
        <col>predicate</col>
        <type>chararray</type>
        <col>object</col>
        <type>chararray</type>
    </columns>
    <delimiter>space</delimiter>

    <process>
        <name>filtered_records</name>
        <task>filter</task>
        <column>predicate</column>
        <condition>==</condition>
        <clause>&lt;isHeadOf&gt;</clause>
    </process>

    <process>
        <name>grouped_records</name>
        <task>group</task>
        <column>predicate</column>
    </process>
```

Fig. 1. xml file



```xml


    <!-- <dumpvar>filtered_records</dumpvar>
    <outputfile>./yagowordcounts</outputfile> -->

    <!-- <dumpvar>grouped_records</dumpvar>
    <outputfile>./yagowordcounts</outputfile> -->

    <!-- <dumpvar>distinct_records</dumpvar>
    <outputfile>./yagowordcounts</outputfile> -->

    <!-- <dumpvar>cross_records</dumpvar>
    <outputfile>./yagowordcounts</outputfile> -->

    <!-- <dumpvar>join_records</dumpvar>
    <outputfile>./yagowordcounts</outputfile> -->

    <!-- <dumpvar>limit_records</dumpvar>
    <outputfile>./yagowordcounts</outputfile> -->

    <!-- <dumpvar>order_records</dumpvar>
    <outputfile>./yagowordcounts</outputfile> -->

    <dumpvar>foreach_records</dumpvar>
    <outputfile>./yagowordcounts</outputfile>

</processes>

```

Fig. 2. xml file



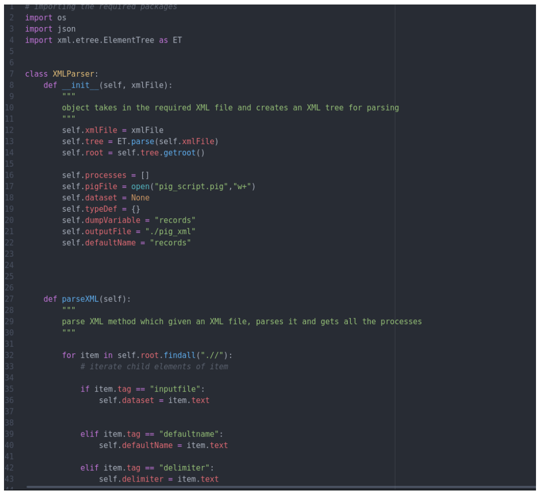
```
records = LOAD './yago_test.tsv' USING PigStorage(' ') AS (subject:chararray,predicate:chararray,object:chararray);
filtered_records = FILTER records by predicate == '<isHeadOf>';
grouped_records = GROUP records by predicate;
distinct_records = DISTINCT records;
cross_records = CROSS records,filtered_records;
join_records = JOIN records BY predicate,filtered_records BY predicate;
limit_records =  LIMIT records 5;
order_records =  ORDER records BY subject;
foreach_records =  FOREACH grouped_records GENERATE group AS predicate,COUNT(records) AS PredictCount;
STORE foreach_records INTO './yagowordcounts';
```

Fig. 3. Generated pif script

B. XML parser

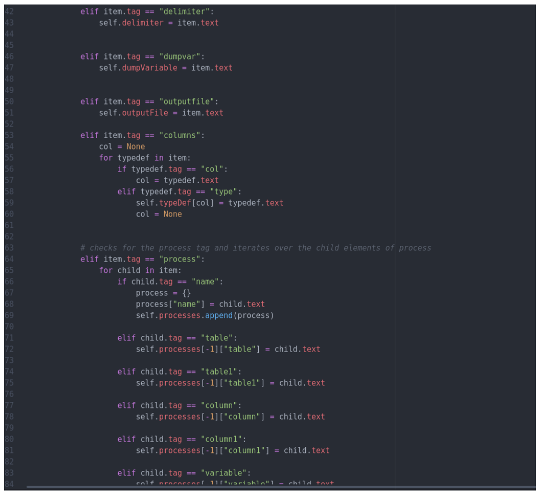
The python script performs XML parsing and generates a Pig script based on the parsed data. The script imports the required packages, including os, json, and xml.etree.ElementTree for XML parsing. It defines a class named XMLParser that takes an XML file as input and initializes various attributes and data structures. The parseXML method is responsible for parsing the XML file and extracting relevant information such as the dataset, column definitions, and processing instructions. The writeScript method generates the Pig script based on the parsed data. It constructs script lines for loading the dataset, applying filter conditions, performing grouping, distinct operations, cross products, joins, limits, ordering, and foreach operations. The generated Pig script is written to a file named ”pig_script.pig.” The executeScript method executes the generated Pig script using the os.system function. In the main section, an instance of the XMLParser class is created with the path to the XML file as an argument. The XML file is parsed, and the dataset, column definitions, and processes arPython script that performs XML parsing and generates a Pig script based on the parsed data.

This script provides a framework for parsing an XML file, extracting relevant information, and generating a Pig script based on the parsed data. It demonstrates an automated approach for processing XML data using Pig.e printed. Finally, the writeScript method is called to generate and execute the Pig script. It demonstrates an automated approach for processing XML data using Pig.



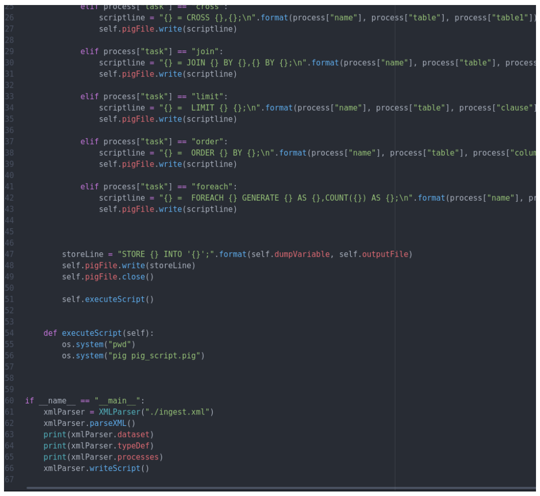
```python
# importing the required packages
import os
import json
import xml.etree.ElementTree as ET


class XMLParser:
    def __init__(self, xmlFile):
        """
        object takes in the required XML file and creates an XML tree for parsing
        """
        self.xmlFile = xmlFile
        self.tree = ET.parse(self.xmlFile)
        self.root = self.tree.getroot()

        self.processes = []
        self.pigFile = open("pig_script.pig","w+")
        self.dataset = None
        self.typeDef = {}
        self.dumpVariable = "records"
        self.outputFile = "./pig_xml"
        self.defaultName = "records"




    def parseXML(self):
        """
        parse XML method which given an XML file, parses it and gets all the processes
        """

        for item in self.root.findall(".//"):
            # iterate child elements of item

            if item.tag == "inputfile":
                self.dataset = item.text


            elif item.tag == "defaultname":
                self.defaultName = item.text

            elif item.tag == "delimiter":
                self.delimiter = item.text
```

Fig. 4. Xml parser



```python
            elif item.tag == "delimiter":
                self.delimiter = item.text


            elif item.tag == "dumpvar":
                self.dumpVariable = item.text


            elif item.tag == "outputfile":
                self.outputFile = item.text

            elif item.tag == "columns":
                col = None
                for typedef in item:
                    if typedef.tag == "col":
                        col = typedef.text
                    elif typedef.tag == "type":
                        self.typeDef[col] = typedef.text
                        col = None


            # checks for the process tag and iterates over the child elements of process
            elif item.tag == "process":
                for child in item:
                    if child.tag == "name":
                        process = {}
                        process["name"] = child.text
                        self.processes.append(process)

                    elif child.tag == "table":
                        self.processes[-1]["table"] = child.text

                    elif child.tag == "table1":
                        self.processes[-1]["table1"] = child.text

                    elif child.tag == "column":
                        self.processes[-1]["column"] = child.text

                    elif child.tag == "column1":
                        self.processes[-1]["column1"] = child.text

                    elif child.tag == "variable":
                        self.processes[-1]["variable"] = child.text
```

Fig. 5. Xml parser



```
            elif process["task"] == "cross":
                scriptline = "{} = CROSS {},{};\n".format(process["name"], process["table"], process["table1"])
                self.pigFile.write(scriptline)

            elif process["task"] == "join":
                scriptline = "{} = JOIN {} BY {},{} BY {};\n".format(process["name"], process["table"], process
                self.pigFile.write(scriptline)

            elif process["task"] == "limit":
                scriptline = "{} =  LIMIT {} {};\n".format(process["name"], process["table"], process["clause"]
                self.pigFile.write(scriptline)

            elif process["task"] == "order":
                scriptline = "{} =  ORDER {} BY {};\n".format(process["name"], process["table"], process["colum
                self.pigFile.write(scriptline)

            elif process["task"] == "foreach":
                scriptline = "{} =  FOREACH {} GENERATE {} AS {},COUNT({}) AS {};\n".format(process["name"], pr
                self.pigFile.write(scriptline)



        storeLine = "STORE {} INTO '{}';".format(self.dumpVariable, self.outputFile)
        self.pigFile.write(storeLine)
        self.pigFile.close()

        self.executeScript()


    def executeScript(self):
        os.system("pwd")
        os.system("pig pig_script.pig")



if __name__ == "__main__":
    xmlParser = XMLParser("./ingest.xml")
    xmlParser.parseXML()
    print(xmlParser.dataset)
    print(xmlParser.typeDef)
    print(xmlParser.processes)
    xmlParser.writeScript()
```

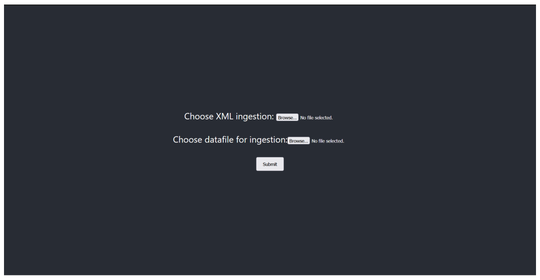
Fig. 6. Xml parser

C. Platform

The project involves several platforms and technologies that work together to create an efficient and user-friendly system for processing large amounts of XML data with a predefined schema.

The data ingestion pipeline will be developed using Python, a popular programming language for data processing and analysis. Python will be used to parse the XML data and generate a Pig script, which is a high-level scripting language designed for large-scale data processing. The Pig script will be executed on a distributed computing framework such as Hadoop, allowing the processing of large amounts of data in a scalable and efficient manner.

To provide a user-friendly interface, the project will include a frontend and backend. The frontend will be developed using React, a popular JavaScript library for building web interfaces. React provides a flexible and powerful platform for building user interfaces, allowing the creation of intuitive and responsive interfaces that can handle large amounts of data. The frontend will allow users to upload their XML files and view the generated Pig scripts and output, making it easy to process and analyze data. The frontend in the initial state takes the following look.

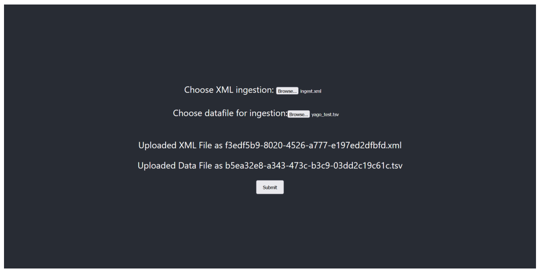


Choose XML ingestion: Browse... No file selected.

Choose datafile for ingestion: Browse... No file selected.

Fig. 7. The initial state of the frontend

The frontend has a form object that takes in 2 corresponding form entities, namely the xml file and data file. Once the both are taken, the submit button is pressed which then sends both of the files to the backend using the *ingest* route.



Choose XML ingestion: Browse... Ingest.xml

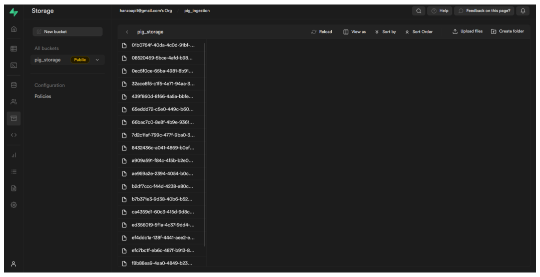
Choose datafile for ingestion: Browse... yago_test.tsv

Uploaded XML File as f3edf5b9-8020-4526-a777-e197ed2dfbfd.xml

Uploaded Data File as b5ea32e8-a343-473c-b3c9-03dd2c19c61c.tsv

Fig. 8. The initial state of the frontend

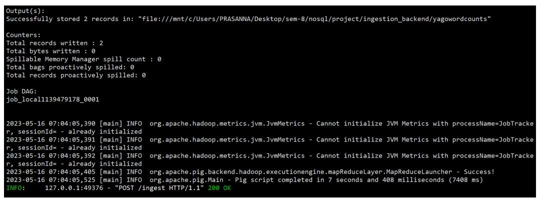
The file storage, path generation, and uploads are handled in a cloud bucket as opposed to local for ease of development. We are using supabase, a cloud platform that supports file buckets. In supabase, a project is created which contains



01b0764f-40da-4c0d-91bf-...
08520469-5bce-4afd-b98...
0ec5f0ce-65ba-4981-8b91...
32ace8f5-c1f5-4e71-94aa-3...
439f860d-8f66-4a5a-bbfe...
65eddd72-c5e0-449c-b60...
66bac7c0-8e8f-4b9e-9361...
7d2c11af-799c-477f-9ba0-3...
8432436c-a041-4869-b0ef...
a909a591-f84c-4f5b-b2e0...
ae959a2e-2394-4054-b0c...
b2df7ccc-f44d-4238-a80c...
b7b371e3-9d38-40b6-b52...
ca4359d1-60c3-415d-9d8c...
ed356019-5f1a-4c37-9dd4-...
ef4ddc1a-138f-4441-aee2-e...
efc7bc1f-eb6c-487f-b913-8...
f8b88ea9-4aa0-4849-b23...

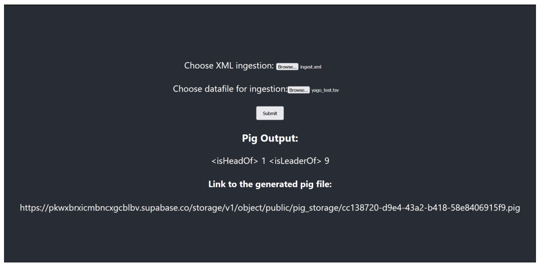
Fig. 9. The supabase bucket storage

The backend will be developed using FastAPI, a modern web framework for building APIs with Python. FastAPI provides a fast and efficient platform for building web applications, allowing for the rapid development of RESTful APIs that can handle large amounts of data. The backend will handle the communication between the frontend and the data ingestion pipeline, as well as the execution of the Pig scripts. The backend has an ingest route which will take in the file paths from the frontend, run the XML parser, generate the pig file, execute the pig file and return the results and the generated pig file back to the frontend.



```
Output(s):
Successfully stored 2 records in: "file:///mnt/c/Users/PRASANNA/Desktop/sem-8/nosql/project/ingestion_backend/yagowordcounts"

Counters:
Total records written : 2
Total bytes written : 0
Spillable Memory Manager spill count : 0
Total bags proactively spilled: 0
Total records proactively spilled: 0

Job DAG:
job_local1139479178_0001


2023-05-16 07:04:05,390 [main] INFO  org.apache.hadoop.metrics.jvm.JvmMetrics - Cannot initialize JVM Metrics with processName=JobTracke
r, sessionId= - already initialized
2023-05-16 07:04:05,391 [main] INFO  org.apache.hadoop.metrics.jvm.JvmMetrics - Cannot initialize JVM Metrics with processName=JobTracke
r, sessionId= - already initialized
2023-05-16 07:04:05,392 [main] INFO  org.apache.hadoop.metrics.jvm.JvmMetrics - Cannot initialize JVM Metrics with processName=JobTracke
r, sessionId= - already initialized
2023-05-16 07:04:05,405 [main] INFO  org.apache.pig.backend.hadoop.executionengine.mapReduceLayer.MapReduceLauncher - Success!
2023-05-16 07:04:05,525 [main] INFO  org.apache.pig.Main - Pig script completed in 7 seconds and 408 milliseconds (7408 ms)
INFO:     127.0.0.1:49376 - "POST /ingest HTTP/1.1" 200 OK
```

Fig. 10. Backend Ingestion route


Choose XML ingestion: Browse... ingest.xml
Choose datafile for ingestion: Browse... yago_test.tsv
Submit

Pig Output:

```
<isHeadOf> 1 <isLeaderOf> 9
```

Link to the generated pig file:

https://pkwxbrxicmbncxgcblbv.supabase.co/storage/v1/object/public/pig_storage/cc138720-d9e4-43a2-b418-58e8406915f9.pig

Fig. 11. Frontend final state

Overall, the project involves a range of platforms and technologies that work together to create an efficient and user-friendly system for processing and analyzing large amounts of XML data with a predefined schema. By leveraging the strengths of each platform, the project aims to provide a powerful and flexible solution for data processing and analysis that can be easily used by organizations dealing with large amounts of data.

IV. Evaluation and Results

The process of uploading input files from the frontend, parsing XML files, generating Pig scripts, and obtaining the output generally takes a comparable amount of time to running a regular Pig script. The additional steps involved in handling frontend interactions, XML parsing, and dynamic script generation are designed to be efficient and streamlined. Therefore, the overall execution time remains similar to the traditional Pig script execution. This allows for a seamless user experience without significant delays, ensuring that the workflow operates efficiently and delivers results in a timely manner. We also did this with large dataset successfully.

V. Future

In this project, we utilized Supabase, an online database solution, which has a storage limit of 50MB. While Supabase served our needs for this project, it is essential to consider potential alternatives for future work or projects that may require larger storage capacity. Several options like Cloud-based Relational Databases, NoSql databases, serverless databases or self-hosted databases can be explored as potential replacements for Supabase based on specific requirements and considerations. And we can also do visualizations for ouput file.

VI. Conclusion

In conclusion, the project aims to create a data ingestion pipeline for XML data with a predefined schema. The pipeline utilizes Python for parsing the XML data and generating Pig scripts, which are executed on a distributed computing framework. The goal is to enable efficient processing and analysis of large amounts of data in a scalable manner. By automating the process and providing an easy-to-use interface, the pipeline improves efficiency, accuracy, and accessibility for users working with XML data. The project addresses the challenges of handling diverse data formats and empowers organizations to gain valuable insights from their data quickly and easily.

VII. Links

- Github : Nosql-Project

References

[1] supabase, https://supabase.com/.
[2] fastapi, https://fastapi.tiangolo.com/lo/.
[3] React, https://react.dev/.
[4] XML, https://www.w3.org/standards/xml/core
[5] Apache Pig, https://pig.apache.org/
[6] Hadoop, https://hadoop.apache.org/