ETL Pipeline (GCP) for Customer Data Processing - Sean Corzo 7/28/2023

ETL Pipeline for Customer Data Processing

Scenario:

You have been tasked with designing an ETL pipeline for a company that collects and processes customer data from various sources. The company wants to centralize the data and transform it into a unified format to perform analytics and generate insights. The pipeline should efficiently handle large volumes of data and be scalable to accommodate future growth.

Requirements:

Data Sources: The ETL pipeline should support extracting data from the following sources:

- **CSV files stored in an S3 bucket**
- **Real-time data streams from a Pub/Sub topic**

Data Transformation: The pipeline needs to perform the following transformations on the data:

Data cleansing (e.g., handling missing values, removing duplicates).

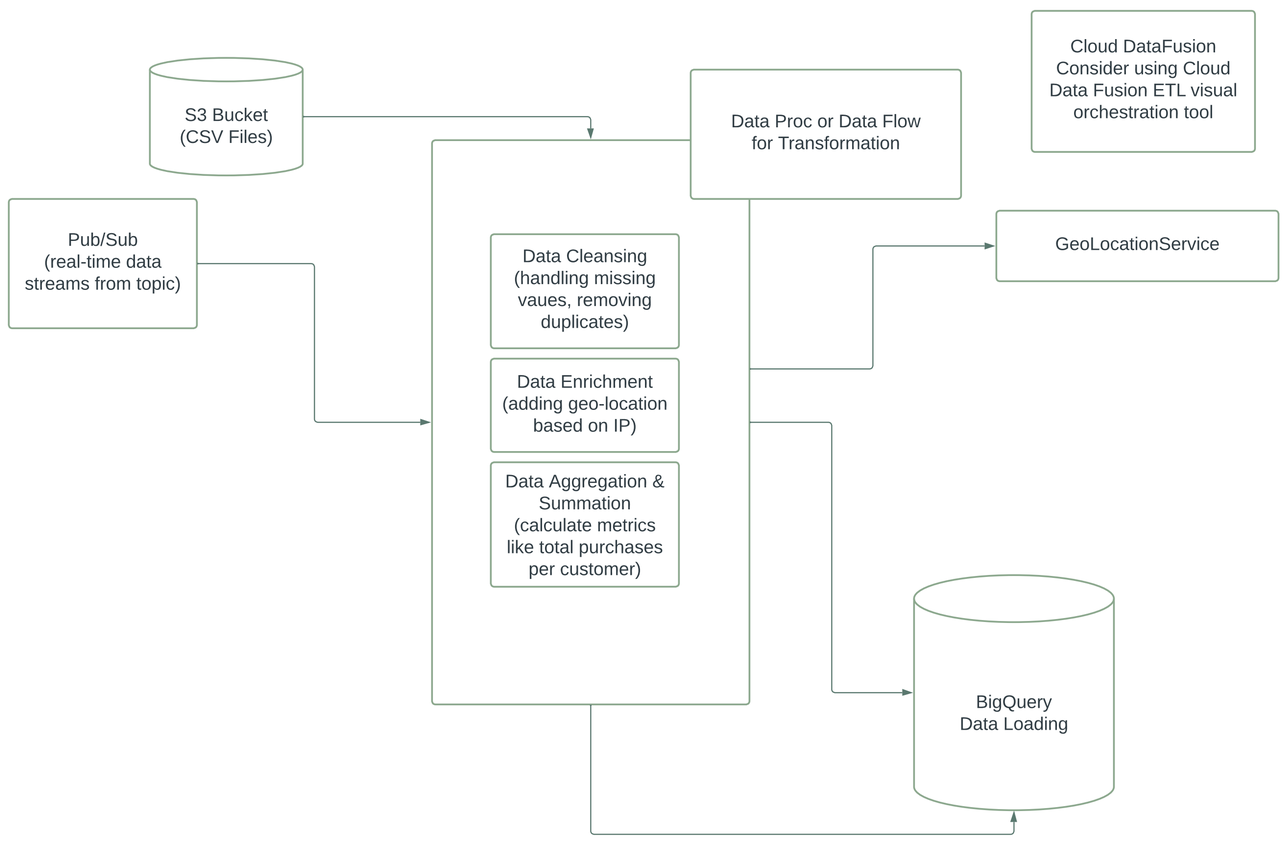
Data enrichment (e.g., adding geolocation information based on IP addresses).

Aggregation and summarization (e.g., calculating metrics like total purchases per customer).

Data Loading: The transformed data should be loaded into a centralized data warehouse for analytics and reporting purposes. The data warehouse of choice is BigQuery.

Scalability: The designed ETL pipeline should be scalable to handle increasing data volumes and processing demands over time.

Considerations:
Data integrity and validation.
Error handling and logging.
Performance optimization.
Security and access controls.



DataFlow

Goolge Cloud DataFlow

DataFlow Data Pipelines are managed end-to-end transformations utilizing Dataflow & Apache Beam. Select from a variety of inputs and outputs, select streaming or schedule batch repetitions, and define SLOs to meet your business needs

Use **Apache Beam**'s programming model and choose from a wide range of transformations to clean and manipulate the data.

ParDo Transform:
The ParDo transform in Apache Beam is often used for data cleansing tasks that require custom logic. It allows you to apply custom functions to individual elements in your data collection.

GroupByKey and Combine:
In cases where data cleansing requires aggregation (e.g., summing values by key), you can use GroupByKey and Combine transforms to perform aggregation operations on the data.

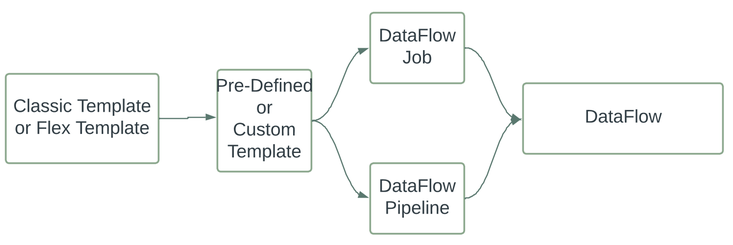
Output Data Sink:
After the data has been cleansed and transformed, you specify the output data sink, where the cleaned data will be written. This could be another file in Google Cloud Storage, a BigQuery table, or any other supported output destination.

Run the Dataflow Job:
Once the data cleansing pipeline is defined, you submit the Dataflow job to the Google Cloud Dataflow service for execution. Dataflow will take care of distributing the processing across a cluster of virtual machines, scaling as needed to handle the data volume, and managing failures and retries.

DataFlow Flex templates and classic templates

With a **Flex template**, the pipeline is packaged as a Docker image in Artifact Registry, along with a template specification file in Cloud Storage. The template specification contains a pointer to the Docker image. When you run the template, the Dataflow service starts a launcher VM, pulls the Docker image, and runs the pipeline. The execution graph is dynamically built based on runtime parameters provided by the user.

A **classic template** contains the JSON serialization of a Dataflow job graph.



DataFlow Jobs and Pipelines

DataFlow Job: a one-time execution of a pipeline
Use Cases: Batch processing

DataFlow Pipeline: a reusable entity that can be scheduled to run on a recurring basis using **Cloud Scheduler**
Use Cases: Streaming processing, batch processing

Simple example of word count using **Apache Beam for DataFlow**:

```
import apache_beam as beam

from apache_beam.options.pipeline_options import PipelineOptions

class WordExtractingDoFn(beam.DoFn):

    def process(self, element):
        return element.split()

def run_word_count_pipeline(input_file, output_path):
    options = PipelineOptions(["--runner=DataflowRunner", "--project=your-project-id"])
    with beam.Pipeline(options=options) as p:
        lines = p | 'ReadInput' >> beam.io.ReadFromText(input_file)
        words = lines | 'ExtractWords' >> beam.ParDo(WordExtractingDoFn())
        word_counts = words | 'CountWords' >> beam.combiners.Count.PerElement()
        formatted_output = word_counts | 'FormatOutput' >> beam.Map(lambda word_count: f"{word_count[0]}: {word_count[1]}")
        formatted_output | 'WriteOutput' >> beam.io.WriteToText(output_path)

if __name__ == '__main__':
    input_file = 'gs://your-bucket/input.txt'
    output_path = 'gs://your-bucket/output'
    run_word_count_pipeline(input_file, output_path)
```

DataFlow Sources and Sinks

Here are examples of some of the most common data sources and destinations (sinks) that Dataflow can work with:

Data Sources:

Google Cloud Storage (GCS):
Read data from GCS buckets using paths like `gs://your-bucket/your-data`.

Apache Kafka:
Read data from Apache Kafka topics for real-time stream processing.

Google BigQuery:
Read data from BigQuery tables for further processing.

Pub/Sub:
Read data from Google Cloud Pub/Sub topics for real-time streaming.

Text Files:
Read data from plain text files (CSV, JSON, XML, etc.) stored in various locations.

Avro Files:
Read data from Avro files, a compact binary data format.

Sinks (Destinations):

Google Cloud Storage (GCS):
Write processed data to GCS buckets using paths like `gs://your-bucket/your-output`.

Google BigQuery:
Write results directly to BigQuery tables for analytics and querying.

Apache Kafka:
Write processed data to Apache Kafka topics.

Pub/Sub:
Write processed data to Pub/Sub topics for real-time streaming.

Text Files:
Write data to plain text files in various formats (CSV, JSON, etc.).

Avro Files:
Write data to Avro files for optimized binary storage.

Google Cloud Firestore:
Write data to Firestore, Google's NoSQL document database.

Custom Sinks:
You can create custom sinks by implementing your own connectors to other data destinations.

DataProc

Google Cloud Dataproc Ecosystem

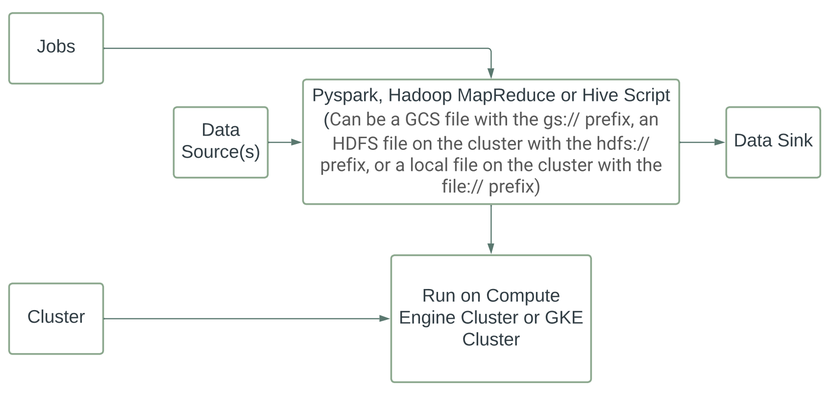
Google Cloud Dataproc is a managed service that provides a platform for running Apache Hadoop, Apache Spark, Apache Hive, and other big data frameworks on Google Cloud Platform. It simplifies the deployment and management of these frameworks, making it easier for users to process and analyze large datasets.

Here's a bit more about how Dataproc supports these frameworks:

Hadoop: Dataproc supports Apache Hadoop, which is a widely used open-source framework for distributed storage and processing of large datasets. Hadoop consists of the Hadoop Distributed File System (HDFS) for storage and the MapReduce programming model for processing data in parallel.

Spark: Apache Spark is another popular open-source framework for big data processing. It provides an in-memory processing engine that is faster than traditional MapReduce for certain workloads. Dataproc allows you to create Spark clusters and run Spark jobs for various data processing tasks, including batch and real-time data analysis.

Hive: Apache Hive is a data warehousing and SQL-like query language for Hadoop. It allows you to query and analyze data using a SQL-like syntax, making it more accessible for users who are familiar with relational databases and SQL. Dataproc supports Hive, allowing you to run Hive queries on your data stored in Hadoop Distributed File System (HDFS) or other data sources.



DataProc Deployment Options

Dataproc is a fully managed Hadoop and Spark service that makes it easy to process large datasets. It offers two deployment options:

- **Cluster on Compute Engine**: This option uses Google Compute Engine (GCE) VMs to create a **Hadoop YARN cluster.** This is the most common deployment option for Dataproc, as it offers the most flexibility and control.
- **Cluster on Kubernetes Engine**: This option uses Google Kubernetes Engine (GKE) to create a Dataproc cluster. This is a newer deployment option that offers some advantages over the Compute Engine option, such as better scalability and reliability.

Common Data Sources and Destinations for DataProc Jobs:

1. **Google Cloud Storage (GCS)**:
Read: Dataproc jobs often read input data from Google Cloud Storage buckets (GCS). You can specify the input path as a GCS URI (e.g., `gs://your-bucket/input`).
Write: Jobs can also write output data to GCS buckets. You can specify the output path as a GCS URI (e.g., `gs://your-bucket/output`).

2. **HDFS (Hadoop Distributed File System)**:
Read: If you have data stored in HDFS, Dataproc jobs can read data from HDFS as well. The path would be specified using an HDFS URI (e.g., `hdfs://your-hdfs-cluster/user/input`).
Write: Similarly, you can write output data to HDFS.

3. **Cloud Bigtable**:
Read: Dataproc jobs can read data from **Cloud Bigtable tables using the HBase API**.
Write: Jobs can also write results to Cloud Bigtable tables.

4. **External Databases**:
Read/Write: Dataproc jobs can read data from and write data to external databases using JDBC or other connectors. This enables interaction with databases like Google Cloud SQL, MySQL, PostgreSQL, and more.

5. **External APIs and Services**:
Read/Write: If your job needs to interact with external APIs or services, you can use the relevant libraries or connectors to read or write data.

6. **Custom Data Sources and Sinks**:
Read/Write: Dataproc jobs can be customized to read from or write to custom data sources or sinks by implementing the necessary connectors.

Simple example of a **PySpark script** that reads a text file, counts the occurrences of each word, and then displays the word count results:

```
from pyspark.sql import SparkSession

# Create a Spark session
spark = SparkSession.builder.appName("WordCount").getOrCreate()

# Load a text file from HDFS or local filesystem
text_file = "file:///path/to/your/textfile.txt"
lines = spark.read.text(text_file).rdd.map(lambda r: r[0])

# Split lines into words and count occurrences
word_counts = lines.flatMap(lambda line: line.split(" ")).countByValue()

# Display word count results
for word, count in word_counts.items():
    print(f"{word}: {count}")

# Stop the Spark session
spark.stop()
```

Simple example of word count script using **Hadoop's MapReduce framework**:

```
// WordCountMapper.java
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
public class WordCountMapper extends Mapper<Object, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException {
        String[] words = value.toString().split(" ");
        for (String w : words) {
            word.set(w);
            context.write(word, one);
        }
    }
}

// WordCountReducer.java
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Reducer;
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
            throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}

# Compile the Java files
javac -cp /path/to/hadoop-common.jar:/path/to/hadoop-mapreduce-client-core.jar WordCountMapper.java WordCountReducer.java

# Create a JAR file
jar -cvf wordcount.jar WordCountMapper.class WordCountReducer.class
```

Simple example of word count script using **Hive** (wordcound.hql file):

```
-- wordcount.hql
CREATE TABLE input_table (
    line STRING
);
LOAD DATA LOCAL INPATH 'input.txt' INTO TABLE input_table;
SELECT word, COUNT(*) AS count
FROM (
    SELECT EXPLODE(SPLIT(line, ' ')) AS word
    FROM input_table
) words
GROUP BY word;
```