GCP Data Engineer - Complete Teaching Guide

Day-by-Day Lesson Plans with Step-by-Step Instructions

📆 DAY 1: PROCESSING ENGINES

⏰ Session 1 (9:00 AM - 12:00 PM): Dataflow (Apache Beam)

🎯 Learning Objectives

By the end of this session, students will:

- Understand Apache Beam programming model
- Create and run a Dataflow pipeline
- Implement windowing for streaming data
- Distinguish between batch and streaming modes

📖 PART 1: Introduction to Dataflow (30 minutes)

Teaching Script:

"Good morning! Today we're diving into Dataflow, Google's serverless data processing service. Think of Dataflow as a smart assembly line that can automatically scale to handle your data."

Key Points to Emphasize:

- ✅ Serverless = No cluster management
- ✅ Auto-scaling = Automatically adjusts workers
- ✅ Unified model = Same code for batch and streaming

Whiteboard Diagram:

Traditional Pipeline:
[Data] → [Your Server] → [Database]
(You manage it, fixed capacity)

Dataflow:
[Data] → [Dataflow Auto-scales 10→1000 workers] → [Database]
(Google manages it, elastic capacity)

Live Demo Setup (Do this together):

Step 1: Enable APIs (Project setup)

bash

```
# Open Cloud Shell (Click icon in top-right of GCP Console)

# Set your project (REPLACE with your project ID)
export PROJECT_ID="your-project-id"
gcloud config set project $PROJECT_ID

# Enable required APIs
gcloud services enable dataflow.googleapis.com
gcloud services enable compute.googleapis.com
gcloud services enable storage.googleapis.com
gcloud services enable bigquery.googleapis.com

# This takes 2-3 minutes
echo "✅ APIs enabled successfully!"
```

Step 2: Create GCS Buckets

bash

```
# Create bucket for Dataflow staging
export BUCKET_NAME="${PROJECT_ID}-dataflow-bucket"
gsutil mb -l us-central1 gs://${BUCKET_NAME}

# Create folders
gsutil mkdir gs://${BUCKET_NAME}/staging
gsutil mkdir gs://${BUCKET_NAME}/temp
gsutil mkdir gs://${BUCKET_NAME}/input
gsutil mkdir gs://${BUCKET_NAME}/output

echo "✅ Buckets created!"
```

Step 3: Create sample input file

```bash
# Create a simple text file
cat > sample.txt << 'EOF'
hello world
hello dataflow
apache beam is powerful
google cloud platform
dataflow processing
EOF

# Upload to GCS
gsutil cp sample.txt gs://${BUCKET_NAME}/input/

echo "✅ Sample data uploaded!"
```

📖 PART 2: Apache Beam Concepts (45 minutes)

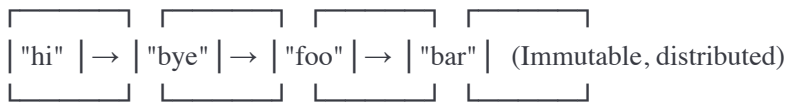
Teaching Script:

"Now let's understand the three core concepts of Apache Beam: PCollection, PTransform, and Pipeline."

Concept 1: PCollection (The Data Container)

Whiteboard Explanation:

PCollection is like a conveyor belt with boxes:



(Immutable, distributed)

Code Example to Project:

```python
```

```
# PCollection example (explain line by line)
import apache_beam as beam

# This line creates a PCollection from a list
numbers = [1, 2, 3, 4, 5]
pcollection = beam.Create(numbers)

# PCollection is distributed and immutable
# You can't change elements, only create new PCollections
```

Concept 2: PTransform (The Processing Logic)

Teaching Progression:

Example 1: Simple Map (one-to-one)

python

```
# Map: Transform each element
# Input:  [1, 2, 3]
# Output: [2, 4, 6]

squared = numbers | beam.Map(lambda x: x * 2)
```

Example 2: Filter

python

```
# Filter: Keep only elements that match condition
# Input:  [1, 2, 3, 4, 5]
# Output: [2, 4]

evens = numbers | beam.Filter(lambda x: x % 2 == 0)
```

Example 3: FlatMap (one-to-many)

python

```
# FlatMap: One input can produce multiple outputs
# Input:  ["hello world", "foo bar"]
# Output: ["hello", "world", "foo", "bar"]

words = sentences | beam.FlatMap(lambda line: line.split())
```

💻 HANDS-ON LAB 1: WordCount Pipeline (60 minutes)

Teaching Script:

"Now let's build the 'Hello World' of data processing - WordCount! We'll count how many times each word appears in a text file."

Lab Instructions (Provide to Students):

Create the Python file:

```bash
# In Cloud Shell, create a new file
nano wordcount.py
```

Complete WordCount Code (Explain section by section):

```python
```

```
"""
WordCount Pipeline - Counts word frequency in text files
Author: Your Name
Date: December 2024
"""

import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions
import argparse
import re


class SplitWords(beam.DoFn):
    """
    DoFn to split lines into words and clean them
    DoFn = 'Do Function' - performs per-element processing
    """

    def process(self, element):
        """
        Process one line at a time
        Args:
            element: One line from the input file (string)
        Yields:
            Individual words (strings)
        """
        # Remove punctuation and convert to lowercase
        line = re.sub(r'[^\w\s]', '', element.lower())

        # Split into words
        words = line.split()

        # Yield each word
        for word in words:
            if word:  # Skip empty strings
                yield word


class FormatOutput(beam.DoFn):
    """
    Format the (word, count) pairs for output
    """

    def process(self, element):
        """
        Args:
```

```
            element: Tuple of (word, count)
        Yields:
            Formatted string
        """
        word, count = element
        yield f'{word}: {count}'


def run(argv=None):
    """
    Main function to run the pipeline
    """

    # Parse command line arguments
    parser = argparse.ArgumentParser()
    parser.add_argument(
        '--input',
        required=True,
        help='Input file pattern (e.g., gs://bucket/input/*.txt)'
    )
    parser.add_argument(
        '--output',
        required=True,
        help='Output file prefix (e.g., gs://bucket/output/counts)'
    )

    known_args, pipeline_args = parser.parse_known_args(argv)

    # Pipeline options
    pipeline_options = PipelineOptions(pipeline_args)

    # Create the pipeline
    with beam.Pipeline(options=pipeline_options) as p:

        # Step 1: Read lines from input file
        lines = (
            p
            | 'ReadFromText' >> beam.io.ReadFromText(known_args.input)
        )

        # Step 2: Split lines into words
        words = (
            lines
            | 'SplitWords' >> beam.ParDo(SplitWords())
        )

        # Step 3: Create (word, 1) pairs
```

```
    word_pairs = (
        words
        | 'PairWithOne' >> beam.Map(lambda word: (word, 1))
    )

    # Step 4: Group by word and sum counts
    word_counts = (
        word_pairs
        | 'GroupAndSum' >> beam.CombinePerKey(sum)
    )

    # Step 5: Format for output
    output = (
        word_counts
        | 'FormatOutput' >> beam.ParDo(FormatOutput())
    )

    # Step 6: Write to output file
    output | 'WriteToText' >> beam.io.WriteToText(known_args.output)


if __name__ == '__main__':
    run()
```

Save the file:

```
Ctrl+O (save)
Enter
Ctrl+X (exit)
```

Running the Pipeline - Three Modes:

MODE 1: Local DirectRunner (for testing)

```
bash
```

```
# Install Apache Beam
pip3 install apache-beam[gcp]

# Run locally
python3 wordcount.py \
    --input gs://${BUCKET_NAME}/input/sample.txt \
    --output gs://${BUCKET_NAME}/output/local-counts \
    --runner DirectRunner

# Check output
gsutil cat gs://${BUCKET_NAME}/output/local-counts*
```

Expected Output:

```
hello: 2
world: 1
dataflow: 2
apache: 1
beam: 1
is: 1
powerful: 1
google: 1
cloud: 1
platform: 1
processing: 1
```

MODE 2: Cloud Dataflow (production)

```
bash

# Run on Dataflow
python3 wordcount.py \
    --input gs://${BUCKET_NAME}/input/sample.txt \
    --output gs://${BUCKET_NAME}/output/dataflow-counts \
    --runner DataflowRunner \
    --project ${PROJECT_ID} \
    --region us-central1 \
    --temp_location gs://${BUCKET_NAME}/temp \
    --staging_location gs://${BUCKET_NAME}/staging

# This will output a job ID like:
# "2024-12-08_01_23_45-1234567890"
```

Monitor the Job:

1. Go to GCP Console
2. Navigate to: Dataflow → Jobs
3. Click on your job
4. Observe:
   - Job Graph (visual pipeline)
   - Worker logs
   - Metrics (elements processed, throughput)

Teaching Point: "Notice how Dataflow automatically created workers for you. This is the 'serverless' magic!"

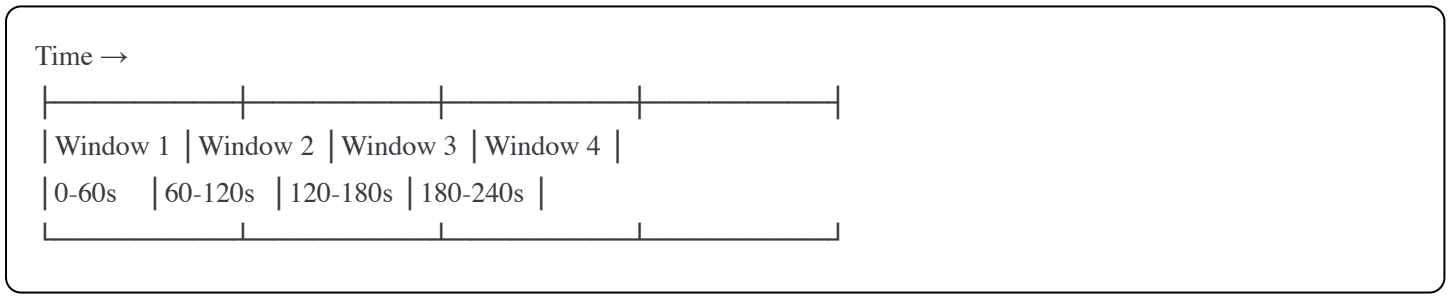
📖 PART 3: Windowing for Streaming (45 minutes)

Teaching Script:

"Now let's tackle streaming data. The challenge: data arrives continuously. How do we group it into batches for processing?"

Concept: Windows

Visual Explanation on Board:



Demo: Streaming WordCount with Windows

Step 1: Create Pub/Sub Topic

```
bash

# Create topic for streaming data
gcloud pubsub topics create wordcount-input

# Create subscription
gcloud pubsub subscriptions create wordcount-sub \
    --topic wordcount-input

echo "✅ Pub/Sub topic created!"
```

Step 2: Create Streaming Pipeline

bash

```
nano streaming_wordcount.py
```

Streaming WordCount Code:

python

```
"""
Streaming WordCount with Windowing
Counts words in 60-second windows
"""

import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions
from apache_beam.transforms.window import FixedWindows
import argparse


def run(argv=None):
    parser = argparse.ArgumentParser()
    parser.add_argument('--project', required=True)
    parser.add_argument('--region', default='us-central1')
    parser.add_argument('--input-topic', required=True)
    parser.add_argument('--output-table', required=True)

    known_args, pipeline_args = parser.parse_known_args(argv)

    # Add streaming flag
    pipeline_args.extend([
        '--streaming',
        '--project=' + known_args.project,
        '--region=' + known_args.region,
        '--temp_location=gs://' + known_args.project + '-dataflow-bucket/temp',
        '--staging_location=gs://' + known_args.project + '-dataflow-bucket/staging',
    ])

    pipeline_options = PipelineOptions(pipeline_args)

    with beam.Pipeline(options=pipeline_options) as p:

        # Read from Pub/Sub
        messages = (
            p
            | 'ReadFromPubSub' >> beam.io.ReadFromPubSub(
                topic=known_args.input_topic
            )
        )

        # Decode bytes to strings
        lines = (
            messages
            | 'DecodeMessages' >> beam.Map(lambda x: x.decode('utf-8'))
        )
```

```
# Apply 60-second fixed windows
windowed_lines = (
    lines
    | 'ApplyWindowing' >> beam.WindowInto(
        FixedWindows(60)  # 60-second windows
    )
)

# Split into words
words = (
    windowed_lines
    | 'SplitWords' >> beam.FlatMap(lambda line: line.split())
)

# Count words per window
word_counts = (
    words
    | 'PairWithOne' >> beam.Map(lambda word: (word, 1))
    | 'GroupAndSum' >> beam.CombinePerKey(sum)
)

# Format for BigQuery
formatted = (
    word_counts
    | 'FormatForBQ' >> beam.Map(
        lambda wc: {
            'word': wc[0],
            'count': wc[1],
            'window_start': beam.pvalue.AsSingleton(
                beam.pvalue.RuntimeValueProvider()
            )
        }
    )
)

# Write to BigQuery
formatted | 'WriteToBigQuery' >> beam.io.WriteToBigQuery(
    known_args.output_table,
    schema='word:STRING,count:INTEGER,window_start:TIMESTAMP',
    create_disposition=beam.io.BigQueryDisposition.CREATE_IF_NEEDED,
    write_disposition=beam.io.BigQueryDisposition.WRITE_APPEND
)
```

```
if __name__ == '__main__':
    run()
```

Step 3: Create BigQuery Table

```bash
# Create dataset
bq mk --dataset ${PROJECT_ID}:wordcount_streaming

# Table will be auto-created by pipeline
```

Step 4: Run Streaming Pipeline

```bash
python3 streaming_wordcount.py \
    --project ${PROJECT_ID} \
    --region us-central1 \
    --input-topic projects/${PROJECT_ID}/topics/wordcount-input \
    --output-table ${PROJECT_ID}:wordcount_streaming.counts \
    --runner DataflowRunner
```

Step 5: Publish Test Messages

```bash
# Publish messages to Pub/Sub (in a separate terminal)
for i in {1..100}; do
    gcloud pubsub topics publish wordcount-input \
        --message "hello world from message $i"
    sleep 1
done
```

Step 6: Query Results

```bash
# After 60+ seconds, query BigQuery
bq query --use_legacy_sql=false \
'SELECT word, SUM(count) as total_count
FROM `'${PROJECT_ID}'.wordcount_streaming.counts`
GROUP BY word
ORDER BY total_count DESC'
```

Expected Output:

```
+----------+-------------+
|   word   | total_count |
+----------+-------------+
| hello    |         100 |
| world    |         100 |
| from     |         100 |
| message  |         100 |
+----------+-------------+
```

🏋️ EXERCISE 1: Custom Windowing (30 minutes)

Challenge for Students: "Modify the streaming pipeline to use 5-minute sliding windows that update every 1 minute."

Solution:

```python
# Replace the windowing step with:
windowed_lines = (
    lines
    | 'ApplySlidingWindow' >> beam.WindowInto(
        beam.window.SlidingWindows(
            size=300,    # 5 minutes
            period=60    # New window every 1 minute
        )
    )
)
```

Teaching Point: "With sliding windows, each message appears in multiple windows! This gives you overlapping analytics."

⏰ Session 2 (1:00 PM - 3:00 PM): Dataproc (Spark)

🎯 Learning Objectives

- Create and manage Dataproc clusters
- Submit PySpark jobs
- Optimize costs with ephemeral clusters and preemptible VMs
- Understand Dataproc vs Dataflow trade-offs

📖 PART 1: Dataproc Introduction (30 minutes)

Teaching Script:

"Welcome back! Now we're switching gears to Dataproc - Google's managed Hadoop and Spark service. If Dataflow is the new kid, Dataproc is the experienced veteran."

Key Talking Points:

- ✅ Lift-and-shift for existing Spark jobs
- ✅ Full Spark ecosystem (MLlib, GraphX, etc.)
- ✅ Managed but not serverless (you see the VMs)
- ✅ Cost optimization through ephemeral clusters

When to Choose Dataproc

Whiteboard Comparison:

```
Choose DATAPROC if:
✓ Existing Spark/Hadoop code
✓ Need Spark ML libraries
✓ Team knows PySpark well
✓ Interactive analysis (Jupyter notebooks)

Choose DATAFLOW if:
✓ New project from scratch
✓ True streaming requirements
✓ Want zero operations
✓ Need seamless batch/streaming
```

💻 HANDS-ON LAB 2: Your First Dataproc Cluster (90 minutes)

Lab Part A: Create Ephemeral Cluster

Step 1: Enable Dataproc API

```
bash

gcloud services enable dataproc.googleapis.com

echo "✅ Dataproc API enabled!"
```

Step 2: Create Your First Cluster

```bash
# Create a small cluster for learning
gcloud dataproc clusters create my-first-cluster \
    --region=us-central1 \
    --zone=us-central1-a \
    --master-machine-type=n1-standard-2 \
    --master-boot-disk-size=50GB \
    --num-workers=2 \
    --worker-machine-type=n1-standard-2 \
    --worker-boot-disk-size=50GB \
    --image-version=2.1-debian11 \
    --max-idle=10m \
    --enable-component-gateway \
    --project=${PROJECT_ID}

# This takes 2-3 minutes
```

While Waiting, Explain Components:

Master Node:
- Runs YARN ResourceManager
- Runs HDFS NameNode
- Coordinates jobs

Worker Nodes:
- Run actual data processing
- Store HDFS data blocks
- Execute tasks

Step 3: Verify Cluster

```bash
# List clusters
gcloud dataproc clusters list --region=us-central1

# Get cluster details
gcloud dataproc clusters describe my-first-cluster \
    --region=us-central1
```

Step 4: Access Web Interfaces

1. Go to: Dataproc → Clusters in GCP Console
2. Click on "my-first-cluster"
3. Click "Web Interfaces" tab
4. Open "YARN ResourceManager"
5. Show students the Spark UI

Lab Part B: Submit PySpark Jobs

Create Sample Data:

```bash
# Create CSV data
cat > sales_data.csv << 'EOF'
date,region,product,amount,quantity
2024-01-01,US-East,Laptop,1200,1
2024-01-01,US-West,Mouse,25,2
2024-01-01,EU,Keyboard,75,1
2024-01-02,US-East,Monitor,300,1
2024-01-02,US-West,Laptop,1200,2
2024-01-02,EU,Mouse,25,5
2024-01-03,US-East,Keyboard,75,3
2024-01-03,US-West,Monitor,300,1
2024-01-03,EU,Laptop,1200,1
EOF

# Upload to GCS
gsutil cp sales_data.csv gs://${BUCKET_NAME}/dataproc/input/
```

Create PySpark Job:

```bash
nano sales_analysis.py
```

PySpark Code (Explain Each Section):

```python
```

```
"""
Sales Analysis PySpark Job
Analyzes sales data and outputs regional summaries
"""

from pyspark.sql import SparkSession
from pyspark.sql.functions import col, sum as _sum, count, avg, round as _round
import sys


def main():
    """
    Main function to analyze sales data
    """

    # Parse command line arguments
    if len(sys.argv) < 3:
        print("Usage: spark-submit sales_analysis.py <input_path> <output_path>")
        sys.exit(1)

    input_path = sys.argv[1]
    output_path = sys.argv[2]

    # Create Spark session
    spark = SparkSession.builder \
        .appName("Sales Analysis") \
        .getOrCreate()

    print(f"📊 Reading data from: {input_path}")

    # Read CSV file
    df = spark.read.csv(
        input_path,
        header=True,
        inferSchema=True
    )

    # Show sample data
    print("📋 Sample data:")
    df.show(5)

    # Print schema
    print("📐 Data schema:")
    df.printSchema()

    # Analysis 1: Total sales by region
```

```
print("💰 Calculating sales by region...")
regional_sales = df.groupBy("region").agg(
    _sum(col("amount")).alias("total_revenue"),
    _sum(col("quantity")).alias("total_items_sold"),
    count("*").alias("transaction_count"),
    _round(avg(col("amount")), 2).alias("avg_transaction_value")
).orderBy(col("total_revenue").desc())

print("📊 Regional Sales Summary:")
regional_sales.show()

# Analysis 2: Top products by revenue
print("🏆 Calculating top products...")
product_sales = df.groupBy("product").agg(
    _sum(col("amount")).alias("total_revenue"),
    _sum(col("quantity")).alias("units_sold")
).orderBy(col("total_revenue").desc())

print("📦 Product Performance:")
product_sales.show()

# Save results to GCS (partitioned by region)
print(f"💾 Saving results to: {output_path}")

regional_sales.write.mode("overwrite").parquet(
    f"{output_path}/regional_sales/"
)

product_sales.write.mode("overwrite").parquet(
    f"{output_path}/product_sales/"
)

# Additional: Create a combined summary
print("📈 Creating combined summary...")
df.createOrReplaceTempView("sales")

summary_sql = """
    SELECT
        region,
        product,
        SUM(amount) as revenue,
        SUM(quantity) as units,
        COUNT(*) as transactions
    FROM sales
    GROUP BY region, product
    ORDER BY revenue DESC
"""
```

```
    combined_summary = spark.sql(summary_sql)
    combined_summary.show()

    combined_summary.write.mode("overwrite").partitionBy("region").parquet(
        f"{output_path}/combined_summary/"
    )

    print("✅ Analysis complete!")

    # Stop Spark session
    spark.stop()


if __name__ == "__main__":
    main()
```

Submit the Job:

```
bash

# Upload PySpark script to GCS
gsutil cp sales_analysis.py gs://${BUCKET_NAME}/dataproc/jobs/

# Submit job to Dataproc
gcloud dataproc jobs submit pyspark \
    gs://${BUCKET_NAME}/dataproc/jobs/sales_analysis.py \
    --cluster=my-first-cluster \
    --region=us-central1 \
    -- \
    gs://${BUCKET_NAME}/dataproc/input/sales_data.csv \
    gs://${BUCKET_NAME}/dataproc/output/

# Job will output a job ID
```

Monitor Job:

```
bash
```

```
# List jobs
gcloud dataproc jobs list --region=us-central1

# Get job details (replace JOB_ID)
gcloud dataproc jobs describe JOB_ID --region=us-central1

# View job logs
gcloud dataproc jobs wait JOB_ID --region=us-central1
```

Check Results:

```
bash

# List output files
gsutil ls -r gs://${BUCKET_NAME}/dataproc/output/

# View results
gsutil cat gs://${BUCKET_NAME}/dataproc/output/regional_sales/part-*
```

Lab Part C: Cost Optimization

Teaching Script: "The cluster we just created costs about $0.50/hour. If we forget to delete it, that's $360/month! Let's learn cost optimization."

Strategy 1: Auto-Delete with Max Idle

```
bash

# Cluster auto-deletes after 10 minutes of inactivity
# We already set this with --max-idle=10m

# Check cluster idle time
gcloud dataproc clusters describe my-first-cluster \
    --region=us-central1 \
    --format="value(config.lifecycleConfig.idleDeleteTtl)"
```

Strategy 2: Preemptible Workers

```
bash
```

```
# Delete old cluster
gcloud dataproc clusters delete my-first-cluster \
    --region=us-central1 \
    --quiet

# Create cost-optimized cluster
gcloud dataproc clusters create cost-optimized-cluster \
    --region=us-central1 \
    --zone=us-central1-a \
    --master-machine-type=n1-standard-2 \
    --num-workers=2 \
    --worker-machine-type=n1-standard-2 \
    --num-preemptible-workers=8 \
    --preemptible-worker-boot-disk-size=50GB \
    --max-idle=10m \
    --image-version=2.1-debian11

# Cost comparison:
# Without preemptible: 3 VMs × $0.15/hr = $0.45/hr
# With preemptible: 3 regular + 8 preemptible
#                   = (3 × $0.15) + (8 × $0.04) = $0.77/hr
# But 3x more compute power!
```

Strategy 3: Ephemeral Clusters (Create & Delete)

bash

```
# Create cluster → Run job → Delete cluster (all in one script)
cat > run_spark_job.sh << 'EOF'
#!/bin/bash

CLUSTER_NAME="ephemeral-cluster-$(date +%s)"
REGION="us-central1"
BUCKET="gs://${PROJECT_ID}-dataflow-bucket"

echo "Creating cluster: $CLUSTER_NAME"
gcloud dataproc clusters create $CLUSTER_NAME \
    --region=$REGION \
    --num-workers=2 \
    --max-idle=5m \
    --quiet

echo "Running Spark job..."
gcloud dataproc jobs submit pyspark \
    ${BUCKET}/dataproc/jobs/sales_analysis.py \
    --cluster=$CLUSTER_NAME \
    --region=$REGION \
    -- \
    ${BUCKET}/dataproc/input/sales_data.csv \
    ${BUCKET}/dataproc/output/

echo "Deleting cluster..."
gcloud dataproc clusters delete $CLUSTER_NAME \
    --region=$REGION \
    --quiet

echo "✅ Job complete, cluster deleted!"
EOF

chmod +x run_spark_job.sh
```

🏋️ EXERCISE 2: Advanced Spark Processing (30 minutes)

Challenge: "Create a PySpark job that reads sales data, filters for US region only, and loads results directly to BigQuery."

Solution:

```
python
```

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("Sales to BigQuery") \
    .getOrCreate()

# Read from GCS
df = spark.read.csv(
    "gs://your-bucket/dataproc/input/sales_data.csv",
    header=True,
    inferSchema=True
)

# Filter US only
us_sales = df.filter(col("region").like("US%"))

# Write to BigQuery
us_sales.write \
    .format("bigquery") \
    .option("table", "your-project:sales_data.us_sales") \
    .option("temporaryGcsBucket", "your-bucket") \
    .mode("overwrite") \
    .save()

spark.stop()
```

Submit with BigQuery connector:

```
bash

gcloud dataproc jobs submit pyspark sales_to_bq.py \
    --cluster=cost-optimized-cluster \
    --region=us-central1 \
    --jars=gs://spark-lib/bigquery/spark-bigquery-latest_2.12.jar
```

⏰ Session 3 (3:30 PM - 5:00 PM): Data Fusion

🎯 Learning Objectives

- Navigate Data Fusion interface
- Build ETL pipelines without code
- Deploy and schedule pipelines

- Understand when to use Data Fusion vs coding

📖 PART 1: Data Fusion Overview (20 minutes)

Teaching Script:

"Data Fusion is the 'no-code' option in our toolkit. Think of it like visual programming - you drag and drop components instead of writing code."

Target Audience:

- Business analysts
- Data engineers who need quick prototypes
- Teams without strong programming skills

Key Point: "Under the hood, Data Fusion generates Dataproc jobs. So it's not magic - it's a GUI wrapper around Spark."

💻 HANDS-ON LAB 3: Build Your First Data Fusion Pipeline (70 minutes)

Step 1: Create Data Fusion Instance

```bash
# Create Data Fusion instance (this takes 15-20 minutes!)
gcloud data-fusion instances create my-fusion-instance \
    --location=us-central1 \
    --edition=BASIC \
    --enable-stackdriver-logging

echo "⏳ Instance creation started... This takes 15-20 minutes."
echo "☕ Perfect time for a coffee break!"
```

While Waiting, Prepare Data Source:

```bash
```

```
# Create MySQL database in Cloud SQL (or use existing)
# For demo purposes, we'll use a CSV file instead

cat > customers.csv << 'EOF'
customer_id,first_name,last_name,email,phone,city,state,signup_date,status
1,John,Doe,john@email.com,555-0101,Seattle,WA,2024-01-15,active
2,Jane,Smith,jane@email.com,555-0102,Portland,OR,2024-01-16,active
3,Bob,Johnson,bob@email.com,555-0103,Denver,CO,2024-01-17,inactive
4,Alice,Williams,alice@email.com,555-0104,Seattle,WA,2024-01-18,active
5,Charlie,Brown,charlie@email.com,555-0105,Austin,TX,2024-01-19,active
EOF

gsutil cp customers.csv gs://${BUCKET_NAME}/fusion/input/
```

Step 2: Access Data Fusion UI

```
bash

# Get the Data Fusion URL
gcloud data-fusion instances describe my-fusion-instance \
    --location=us-central1 \
    --format="value(apiEndpoint)"

# Open this URL in browser
```

Teaching Note: Walk through the UI together:

1. **Studio** - Where you build pipelines
2. **Control Center** - Monitor running pipelines
3. **Hub** - Pre-built plugins and templates
4. **Wrangler** - Data prep tool

Step 3: Build Pipeline (Live Demo)

Demo Script - Follow Along:

A. Create New Pipeline

```
1. Click "Studio" (top left)
2. Click "Create Pipeline"
3. Select "Batch Pipeline"
4. Name it: "Customer Data Cleanup"
```

B. Add Source (GCS File)

1. In left panel, find "Source" category
2. Drag "GCS" source to canvas
3. Click on the GCS node
4. Configure:
   - Reference Name: customers-raw
   - Path: gs://YOUR-BUCKET/fusion/input/customers.csv
   - Format: csv
   - Skip Header: Yes
   - Click "Get Schema" button
5. Click "Validate"

C. Add Wrangler (Data Cleaning)

1. From left panel, drag "Wrangler" to canvas
2. Connect GCS source to Wrangler (drag arrow)
3. Click on Wrangler node
4. Click "Wrangle" button

In Wrangler interface, add directives:
5. Type these commands:

```
# Remove rows with missing emails
filter-rows-on condition-true email =~ '^.+@.+\..+$'

# Standardize phone format
parse-as-csv phone ',' false

# Convert names to proper case
titlecase first_name
titlecase last_name

# Uppercase state codes
uppercase state

# Filter active customers only
filter-rows-on condition-true status == 'active'
```

6. Click "Apply"
7. Click "✓" (checkmark) to close Wrangler

D. Add Transformation (Custom Logic)

1. Drag "JavaScript" transform to canvas
2. Connect Wrangler to JavaScript
3. Click on JavaScript node
4. Add this code:

```
// Add derived fields
function transform(input, emitter, context) {
    // Create full name
    input.full_name = input.first_name + ' ' + input.last_name;

    // Calculate days since signup
    var signup = new Date(input.signup_date);
    var now = new Date();
    var days = Math.floor((now - signup) / (1000 * 60 * 60 * 24));
    input.days_as_customer = days;

    // Customer tier based on tenure
    if (days > 180) {
        input.tier = 'gold';
    } else if (days > 90) {
        input.tier = 'silver';
    } else {
        input.tier = 'bronze';
    }

    emitter.emit(input);
}
```

5. Click "Validate"

E. Add Sink (BigQuery)

1. Drag "BigQuery" sink to canvas
2. Connect JavaScript transform to BigQuery
3. Click on BigQuery node
4. Configure:
   - Reference Name: customers-clean
   - Dataset: customer_data (create if needed)
   - Table: customers_cleaned
   - Create if not exists: Yes
   - Truncate table: Yes
5. Click "Validate"

F. Deploy and Run

1. Click "Deploy" button (top right)
2. Wait for validation (green checkmark)
3. Click "Run" button
4. Watch in "Control Center"

Expected Timeline:

- Creating Dataproc cluster: 2-3 minutes
- Running pipeline: 1-2 minutes
- Deleting cluster: 1 minute
- Total: ~5 minutes

Step 4: Verify Results

bash

```
# Query BigQuery results
bq query --use_legacy_sql=false \
'SELECT
    full_name,
    email,
    city,
    state,
    tier,
    days_as_customer
FROM `'${PROJECT_ID}'.customer_data.customers_cleaned`
ORDER BY days_as_customer DESC'
```

Expected Output:

```
+----------------+------------------+---------+-------+--------+------------------+
|  full_name     |      email       |  city   | state | tier   | days_as_customer |
+----------------+------------------+---------+-------+--------+------------------+
| John Doe       | john@email.com   | Seattle | WA    | gold   | 327              |
| Jane Smith     | jane@email.com   | Portland| OR    | gold   | 326              |
| Alice Williams | alice@email.com  | Seattle | WA    | gold   | 324              |
| Charlie Brown  | charlie@email.com| Austin  | TX    | silver | 323              |
+----------------+------------------+---------+-------+--------+------------------+
```

🏋️ EXERCISE 3: Schedule Pipeline (20 minutes)

Challenge: "Schedule the pipeline to run daily at 2 AM UTC."

Solution Steps:

1. In Data Fusion UI, go to your pipeline
2. Click "Schedule" button
3. Configure:
   - Schedule Name: daily-customer-cleanup
   - Cron Expression: 0 2 * * *
   - Max Concurrent Runs: 1
   - Time Zone: UTC
4. Click "Save"
5. Click "Suspend/Resume" to activate

Cron Expression Explained:

```
0 2 * * *
│ │ │ │ │
│ │ │ │ └─ Day of week (any)
│ │ │ └─── Month (any)
│ │ └───── Day of month (any)
│ └─────── Hour (2 AM)
└───────── Minute (0)
```

📝 Day 1 Summary & Quiz (30 minutes)

Key Takeaways

Dataflow:

- ✅ Serverless, autoscaling
- ✅ Apache Beam (Java/Python)
- ✅ Best for: New projects, streaming
- ⚠️ Learning curve for Beam

Dataproc:

- ✅ Managed Spark/Hadoop
- ✅ Lift-and-shift existing jobs
- ✅ Best for: Legacy migration, ML
- ⚠️ Requires cluster management

Data Fusion:

- ✅ No-code, visual pipelines
- ✅ Quick integrations
- ✅ Best for: Analysts, prototypes
- ⚠️ Less flexible than coding

Quick Quiz

Question 1: "You need to process 10 TB of log files once per day. Which service and why?"

Answer: "Dataproc with ephemeral cluster. Batch job, cost-effective with cluster auto-delete."

Question 2: "Real-time IoT data, 100K messages/second, need 5-second latency. Which service?"

Answer: "Dataflow streaming. Serverless autoscaling handles throughput, windowing for latency."

Question 3: "Business analyst needs to sync Salesforce to BigQuery weekly. Which service?"

Answer: "Data Fusion. No-code interface, built-in Salesforce connector, easy scheduling."

🏠 Day 1 Homework

Assignment 1: Batch Processing

Create a Dataflow pipeline that:
1. Reads JSON files from GCS
2. Parses and validates records
3. Filters out records older than 90 days
4. Writes to BigQuery partitioned by date

Assignment 2: Spark Optimization

Optimize this Spark job for cost:
1. Use preemptible workers
2. Set cluster auto-delete to 5 minutes
3. Measure cost savings

Assignment 3: Data Fusion Extension

Extend the customer pipeline to:
1. Join with a second CSV (orders.csv)
2. Calculate total order value per customer

3. Filter for customers with orders > $1000
4. Output to separate BigQuery table

📅 DAY 2: ORCHESTRATION & ARCHITECTURE

⏰ Session 1 (9:00 AM - 12:00 PM): Cloud Composer (Airflow)

🎯 Learning Objectives

- Understand DAG concepts
- Write Python-based workflows
- Implement dependencies and scheduling
- Handle errors and retries

📖 PART 1: Introduction to Cloud Composer (30 minutes)

Teaching Script:

"Good morning! Yesterday we learned about workers - Dataflow and Dataproc. Today we learn about the conductor - Cloud Composer. It doesn't process data; it orchestrates when things happen."

Key Analogy:

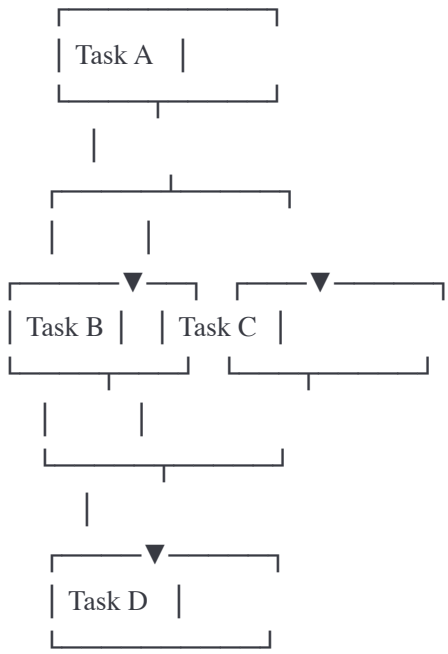
Think of a recipe:
1. Preheat oven (Task 1)
2. Mix ingredients (Task 2) - depends on Task 1
3. Bake (Task 3) - depends on Task 2
4. Cool and serve (Task 4) - depends on Task 3

Composer ensures tasks run in the right order!

What is a DAG?

- **D**irected: Tasks flow in one direction
- **A**cyclic: No loops (can't go backwards)
- **G**raph: Network of tasks

Visual on Board:



💻 HANDS-ON LAB 4: Create Composer Environment (90 minutes)

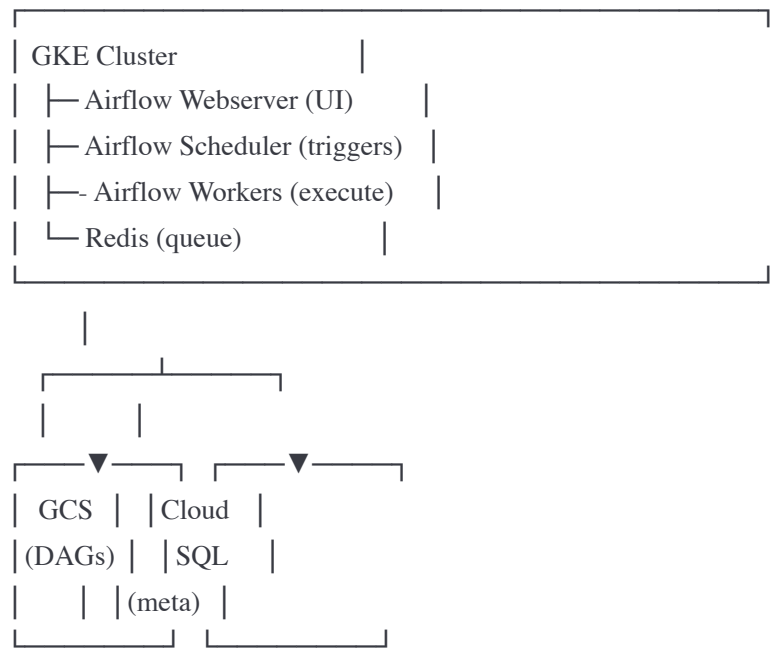
Step 1: Create Composer Environment

```bash
# This takes 20-25 minutes!
gcloud composer environments create my-composer-env \
    --location us-central1 \
    --python-version 3 \
    --machine-type n1-standard-1 \
    --node-count 3 \
    --disk-size 30GB \
    --env-variables \
        PROJECT_ID=${PROJECT_ID},\
        BUCKET=${BUCKET_NAME}

echo "⏳ Composer environment creating... (~25 minutes)"
echo "☕ Time for another coffee break!"
```

While Waiting, Explain Architecture:

Composer Environment Components:



Step 2: Your First DAG - Hello World

bash

```
# Get DAG folder location
export COMPOSER_BUCKET=$(gcloud composer environments describe my-composer-env \
    --location us-central1 \
    --format="get(config.dagGcsPrefix)")

echo "Composer DAG folder: ${COMPOSER_BUCKET}"

# Create first DAG
cat > hello_world_dag.py << 'EOF'
"""
Hello World DAG - Your first Airflow pipeline
Demonstrates basic operators and dependencies
"""

from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator


# Default arguments applied to all tasks
default_args = {
    'owner': 'data-engineer',
    'depends_on_past': False,
    'email': ['your-email@example.com'],
    'email_on_failure': True,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}


def print_context(**context):
    """Python function to print execution context"""
    print(f"Execution date: {context['ds']}")
    print(f"Task: {context['task']}")
    print(f"DAG: {context['dag']}")
    return "Success!"


# Define the DAG
with DAG(
    'hello_world',
    default_args=default_args,
    description='A simple Hello World DAG',
    schedule_interval=timedelta(days=1),  # Run daily
```

```
    start_date=datetime(2024, 1, 1),
    catchup=False,  # Don't backfill
    tags=['example', 'hello-world'],
) as dag:

    # Task 1: Print date
    task1 = BashOperator(
        task_id='print_date',
        bash_command='date',
    )

    # Task 2: Sleep for 5 seconds
    task2 = BashOperator(
        task_id='sleep',
        bash_command='sleep 5',
    )

    # Task 3: Print execution context
    task3 = PythonOperator(
        task_id='print_context',
        python_callable=print_context,
        provide_context=True,
    )

    # Task 4: Final message
    task4 = BashOperator(
        task_id='final_message',
        bash_command='echo "Hello World DAG completed!"',
    )

    # Define dependencies
    task1 >> task2 >> task3 >> task4

    # Alternative syntax (same result):
    # task1.set_downstream(task2)
    # task2.set_downstream(task3)
    # task3.set_downstream(task4)
EOF

# Upload to Composer
gsutil cp hello_world_dag.py ${COMPOSER_BUCKET}/dags/
```

Step 3: Access Airflow UI

```
bash
```

```
# Get Airflow web UI URL
gcloud composer environments describe my-composer-env \
    --location us-central1 \
    --format="get(config.airflowUri)"

# Open this URL in browser
```

Live Demo - Walk Through UI:

1. Navigate to DAGs page
2. Find "hello_world" DAG
3. Click on it
4. Show:
   - Graph View (visual dependencies)
   - Tree View (historical runs)
   - Code View (Python source)
5. Toggle DAG to "On"
6. Click "Trigger DAG" (play button)
7. Watch tasks execute in real-time
8. Click on tasks to see logs

Step 4: Real ETL DAG

Create Production-Style DAG:

bash

```
cat > daily_etl_dag.py << 'EOF'
"""
Daily ETL Pipeline
1. Extract data from GCS
2. Process with Dataproc
3. Load to BigQuery
4. Send notification
"""

from datetime import datetime, timedelta
from airflow import DAG
from airflow.providers.google.cloud.operators.dataproc import (
    DataprocCreateClusterOperator,
    DataprocSubmitJobOperator,
    DataprocDeleteClusterOperator,
)
from airflow.providers.google.cloud.transfers.gcs_to_bigquery import (
    GCSToBigQueryOperator,
)
from airflow.operators.bash import BashOperator
from airflow.operators.python import PythonOperator
import os


# Get environment variables
PROJECT_ID = os.environ.get('PROJECT_ID', 'your-project-id')
REGION = 'us-central1'
BUCKET = os.environ.get('BUCKET', 'your-bucket')


default_args = {
    'owner': 'data-team',
    'depends_on_past': False,
    'email': ['alerts@company.com'],
    'email_on_failure': True,
    'retries': 2,
    'retry_delay': timedelta(minutes=5),
}


# Cluster configuration
CLUSTER_CONFIG = {
    'master_config': {
        'num_instances': 1,
        'machine_type_uri': 'n1-standard-2',
        'disk_config': {'boot_disk_size_gb': 50},
```

```
    },
    'worker_config': {
        'num_instances': 2,
        'machine_type_uri': 'n1-standard-2',
        'disk_config': {'boot_disk_size_gb': 50},
    },
}


# PySpark job configuration
PYSPARK_JOB = {
    'reference': {'project_id': PROJECT_ID},
    'placement': {'cluster_name': 'etl-cluster-{{ ds_nodash }}'},
    'pyspark_job': {
        'main_python_file_uri': f'gs://{BUCKET}/dataproc/jobs/sales_analysis.py',
        'args': [
            f'gs://{BUCKET}/dataproc/input/sales_data.csv',
            f'gs://{BUCKET}/dataproc/output/{{ ds }}/',
        ],
    },
}


def check_data_quality(**context):
    """Validate data before processing"""
    from google.cloud import storage

    client = storage.Client()
    bucket = client.bucket(BUCKET)
    blob = bucket.blob('dataproc/input/sales_data.csv')

    # Check if file exists
    if not blob.exists():
        raise FileNotFoundError("Input file not found!")

    # Check file size
    size_mb = blob.size / (1024 * 1024)
    print(f"File size: {size_mb:.2f} MB")

    if size_mb < 0.001:  # Less than 1 KB
        raise ValueError("File is too small!")

    return f"Data quality check passed. Size: {size_mb:.2f} MB"


def send_success_notification(**context):
    """Send success notification"""
```

```
    ds = context['ds']
    print(f"📧 ETL Pipeline completed successfully for {ds}")
    print(f"✅ Data loaded to BigQuery")
    print(f"📊 Records processed: {context.get('records', 'N/A')}")
    # In production, send actual email/Slack notification
    return "Notification sent"


with DAG(
    'daily_sales_etl',
    default_args=default_args,
    description='Daily sales ETL pipeline',
    schedule_interval='0 2 * * *',  # 2 AM UTC daily
    start_date=datetime(2024, 1, 1),
    catchup=False,
    tags=['production', 'etl', 'sales'],
) as dag:

    # Task 1: Data quality check
    quality_check = PythonOperator(
        task_id='check_data_quality',
        python_callable=check_data_quality,
        provide_context=True,
    )

    # Task 2: Create Dataproc cluster
    create_cluster = DataprocCreateClusterOperator(
        task_id='create_dataproc_cluster',
        project_id=PROJECT_ID,
        cluster_name='etl-cluster-{{ ds_nodash }}',
        region=REGION,
        cluster_config=CLUSTER_CONFIG,
    )

    # Task 3: Submit Spark job
    process_data = DataprocSubmitJobOperator(
        task_id='process_sales_data',
        job=PYSPARK_JOB,
        region=REGION,
        project_id=PROJECT_ID,
    )

    # Task 4: Delete cluster (runs even if previous tasks fail)
    delete_cluster = DataprocDeleteClusterOperator(
        task_id='delete_dataproc_cluster',
        project_id=PROJECT_ID,
        cluster_name='etl-cluster-{{ ds_nodash }}',
```

```
    region=REGION,
    trigger_rule='all_done',  # Run regardless of previous task status
)

# Task 5: Load to BigQuery
load_to_bq = GCSToBigQueryOperator(
    task_id='load_to_bigquery',
    bucket=BUCKET,
    source_objects=[f'dataproc/output/{{{{ ds }}}}/regional_sales/*.parquet'],
    destination_project_dataset_table=f'{PROJECT_ID}:sales_analytics.daily_sales',
    source_format='PARQUET',
    write_disposition='WRITE_TRUNCATE',
    create_disposition='CREATE_IF_NEEDED',
    autodetect=True,
)

# Task 6: Send notification
notify = PythonOperator(
    task_id='send_notification',
    python_callable=send_success_notification,
    provide_context=True,
)

# Task 7: Cleanup temporary files
cleanup = BashOperator(
    task_id='cleanup_temp_files',
    bash_command=f'gsutil -m rm -r gs://{BUCKET}/dataproc/output/{{{{ ds }}}}/ || true',
)

# Define workflow
quality_check >> create_cluster >> process_data >> delete_cluster >> load_to_bq >> notify >> cleanup
EOF

# Upload DAG
gsutil cp daily_etl_dag.py ${COMPOSER_BUCKET}/dags/
```

Wait 2-3 minutes for Airflow to detect the DAG

🏋️ EXERCISE 4: Branching and Conditionals (40 minutes)

Challenge: "Create a DAG that checks file size. If > 1 GB, use Dataproc. If < 1 GB, use Dataflow."

Solution:

```
python
```

```python
from airflow.operators.python import BranchPythonOperator

def choose_processing_method(**context):
    """Decide which processing method to use"""
    from google.cloud import storage

    client = storage.Client()
    bucket = client.bucket(BUCKET)
    blob = bucket.blob('input/data.csv')

    size_gb = blob.size / (1024**3)

    if size_gb > 1:
        return 'process_with_dataproc'
    else:
        return 'process_with_dataflow'


with DAG('smart_processing', ...) as dag:

    branch = BranchPythonOperator(
        task_id='choose_method',
        python_callable=choose_processing_method,
    )

    dataproc_task = DataprocSubmitJobOperator(
        task_id='process_with_dataproc',
        ...
    )

    dataflow_task = DataflowStartPythonJobOperator(
        task_id='process_with_dataflow',
        ...
    )

    branch >> [dataproc_task, dataflow_task]
```

⏰ Session 2 (1:00 PM - 3:00 PM): Pipeline Design Patterns

🎯 Learning Objectives

- Understand ETL vs ELT trade-offs
- Implement schema evolution

- Design for data quality

📖 PART 1: ETL vs ELT (45 minutes)

Teaching Script:

"This is one of the most important architectural decisions you'll make. Should you transform BEFORE loading (ETL) or AFTER loading (ELT)?"

Visual Comparison:

```
ETL (Extract-Transform-Load):
[Source] → [Dataflow: Clean, Join, Aggregate] → [BigQuery: Clean Data]
                ↑
                Transformation happens HERE

Pros: Less storage, data already clean
Cons: Slower loads, transformation changes need pipeline changes

ELT (Extract-Load-Transform):
[Source] → [Load Raw to BigQuery] → [SQL: Transform] → [Final Tables]
                                        ↑
                                        Transformation happens HERE

Pros: Fast loads, flexible transformations
Cons: More storage, raw data in warehouse
```

Live Demo: Both Approaches

Scenario: Process customer data with email validation

ETL Approach (Dataflow):

```
python
```

```
# Dataflow pipeline that cleans BEFORE loading
import apache_beam as beam
import re

def validate_email(record):
    """Validate and clean email"""
    email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
    if re.match(email_pattern, record['email']):
        return record
    return None

with beam.Pipeline() as p:
    (p
     | 'Read' >> beam.io.ReadFromText('gs://bucket/raw.csv')
     | 'Parse' >> beam.Map(parse_csv)
     | 'ValidateEmail' >> beam.Map(validate_email)
     | 'FilterNone' >> beam.Filter(lambda x: x is not None)
     | 'TransformNames' >> beam.Map(lambda r: {
         **r,
         'name': r['name'].title(),
         'email': r['email'].lower()
       })
     | 'WriteToBQ' >> beam.io.WriteToBigQuery(
         'project:dataset.customers_clean',
         schema='name:STRING,email:STRING,status:STRING'
       )
    )
```

ELT Approach (Load then SQL):

```
bash
```

```
# Step 1: Quick load to BigQuery (raw)
bq load \
    --source_format=CSV \
    --autodetect \
    --replace \
    my_dataset.customers_raw \
    gs://bucket/raw.csv

# Step 2: Transform with SQL
bq query --use_legacy_sql=false '
CREATE OR REPLACE TABLE my_dataset.customers_clean AS
SELECT
    INITCAP(name) AS name,
    LOWER(email) AS email,
    status
FROM my_dataset.customers_raw
WHERE REGEXP_CONTAINS(email, r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$")
  AND status = "active"
'
```

Performance Comparison:

Aspect	ETL (Dataflow)	ELT (BigQuery SQL)
Load Time	10 min	1 min
Storage Cost	$20/month	$40/month (raw+clean)
Transformation Time	N/A (in load)	30 seconds
Change Flexibility	Need to redeploy	Just change SQL
Best For	Complex logic, streaming	Simple SQL transforms

📖 PART 2: Schema Evolution (45 minutes)

Teaching Script:

"Real-world data changes. Columns get added, types change, fields get renamed. Your pipeline needs to handle this gracefully."

Scenario: Source Adds New Column

Initial Schema:

```
customer_id, name, email
```

New Schema (3 months later):

```
customer_id, name, email, phone, tier
```

Problem: Old pipelines break!

Solution 1: Schema Auto-Detection (BigQuery)

```python
# Dataflow with flexible schema
def create_record(element):
    """Create record with all available fields"""
    return {
        'customer_id': element.get('customer_id'),
        'name': element.get('name'),
        'email': element.get('email'),
        'phone': element.get('phone'),  # New field
        'tier': element.get('tier', 'standard'),  # Default value
    }

pipeline | beam.io.WriteToBigQuery(
    'project:dataset.customers',
    schema='SCHEMA_AUTODETECT',
    write_disposition='WRITE_APPEND',
    create_disposition='CREATE_IF_NEEDED',
    additional_bq_parameters={
        'schemaUpdateOptions': [
            'ALLOW_FIELD_ADDITION',
            'ALLOW_FIELD_RELAXATION'
        ]
    }
)
```

What happens:

- Old records: `phone` and `tier` = NULL
- New records: All fields populated
- No pipeline changes needed!

Solution 2: Schema Registry Pattern

```python
```

```python
# schemas.py - Version controlled schemas
CUSTOMER_SCHEMA_V1 = {
    'fields': [
        {'name': 'customer_id', 'type': 'STRING'},
        {'name': 'name', 'type': 'STRING'},
        {'name': 'email', 'type': 'STRING'},
    ]
}

CUSTOMER_SCHEMA_V2 = {
    'fields': [
        {'name': 'customer_id', 'type': 'STRING'},
        {'name': 'name', 'type': 'STRING'},
        {'name': 'email', 'type': 'STRING'},
        {'name': 'phone', 'type': 'STRING', 'mode': 'NULLABLE'},
        {'name': 'tier', 'type': 'STRING', 'mode': 'NULLABLE'},
    ]
}

# In your pipeline
class MigrateSchema(beam.DoFn):
    def process(self, element):
        """Migrate old records to new schema"""
        if 'phone' not in element:
            element['phone'] = None
        if 'tier' not in element:
            element['tier'] = 'standard'
        yield element
```

💻 HANDS-ON LAB 5: Build Robust Pipeline (30 minutes)

Challenge: "Build a pipeline that handles schema changes gracefully and validates data quality."

Solution:

```python
```

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions

class ValidateRecord(beam.DoFn):
    """Validate data quality"""
    def process(self, element):
        errors = []

        # Required fields check
        required = ['customer_id', 'email']
        for field in required:
            if field not in element or not element[field]:
                errors.append(f"Missing {field}")

        # Email format
        import re
        if 'email' in element:
            if not re.match(r'^.+@.+\..+$', element['email']):
                errors.append("Invalid email format")

        if errors:
            # Send to dead letter queue
            yield beam.pvalue.TaggedOutput('errors', {
                'record': element,
                'errors': errors
            })
        else:
            # Valid record
            yield element

with beam.Pipeline(options=PipelineOptions()) as p:

    # Read input
    raw_data = p | 'Read' >> beam.io.ReadFromText('gs://bucket/input.csv')

    # Validate with tagged output
    validated = raw_data | 'Validate' >> beam.ParDo(
        ValidateRecord()
    ).with_outputs('errors', main='valid')

    # Process valid records
    valid_data = validated.valid | 'Process' >> beam.Map(process_record)

    # Write valid to BigQuery
    valid_data | 'WriteValid' >> beam.io.WriteToBigQuery(
        'project:dataset.customers',
```

```
    schema='SCHEMA_AUTODETECT',
    additional_bq_parameters={
      'schemaUpdateOptions': ['ALLOW_FIELD_ADDITION']
    }
  )

  # Write errors to separate table for debugging
  validated.errors | 'WriteErrors' >> beam.io.WriteToBigQuery(
    'project:dataset.data_quality_errors',
    schema='record:JSON,errors:STRING'
  )
```

⏰ Session 3 (3:30 PM - 5:00 PM): Partitioning & Clustering

🎯 Learning Objectives

- Understand partitioning vs sharding
- Implement partitioning strategies
- Optimize with clustering

📖 PART 1: Partitioning Deep Dive (45 minutes)

Teaching Script:

"Partitioning is THE most important optimization technique in BigQuery. It can reduce your query costs by 100x!"

The Problem (Without Partitioning):

```sql
-- Query 1 day of data from 1 year table
SELECT * FROM events
WHERE event_date = '2024-12-01';

-- Scans: 365 GB (entire table!)
-- Cost: $1.83
```

The Solution (With Partitioning):

```sql
```

```
-- Same query on partitioned table
SELECT * FROM events_partitioned
WHERE event_date = '2024-12-01';

-- Scans: 1 GB (only Dec 1 partition!)
-- Cost: $0.005
```

Live Demo: Create Partitioned Table

Step 1: Create Sample Data

bash

```
# Create sample events data
cat > generate_events.py << 'EOF'
import json
from datetime import datetime, timedelta
import random

# Generate 10,000 events across 30 days
events = []
start_date = datetime(2024, 11, 1)

for i in range(10000):
    event_date = start_date + timedelta(days=random.randint(0, 29))
    event = {
        'event_id': f'EVT{i:06d}',
        'user_id': f'USER{random.randint(1, 1000):04d}',
        'event_type': random.choice(['click', 'view', 'purchase', 'logout']),
        'timestamp': event_date.isoformat(),
        'properties': {'page': f'/page{random.randint(1, 50)}'}
    }
    print(json.dumps(event))

EOF

python3 generate_events.py > events.jsonl

# Upload to GCS
gsutil cp events.jsonl gs://${BUCKET_NAME}/bq/events/
```

Step 2: Create Non-Partitioned Table (for comparison)

bash

```
bq load \
  --source_format=NEWLINE_DELIMITED_JSON \
  --autodetect \
  ${PROJECT_ID}:analytics.events_no_partition \
  gs://${BUCKET_NAME}/bq/events/events.jsonl
```

Step 3: Create Partitioned Table

```bash
bq mk \
  --table \
  --schema='event_id:STRING,user_id:STRING,event_type:STRING,timestamp:TIMESTAMP,properties:JSON' \
  --time_partitioning_field=timestamp \
  --time_partitioning_type=DAY \
  --time_partitioning_expiration=7776000 \
  --description="Events table partitioned by timestamp (90 day retention)" \
  ${PROJECT_ID}:analytics.events_partitioned

# Load data
bq load \
  --source_format=NEWLINE_DELIMITED_JSON \
  --autodetect \
  ${PROJECT_ID}:analytics.events_partitioned \
  gs://${BUCKET_NAME}/bq/events/events.jsonl
```

Step 4: Compare Query Performance

```bash
```

```
# Query WITHOUT partition filter (BAD)
echo "Query 1: No partition filter"
bq query --use_legacy_sql=false \
'SELECT COUNT(*) as total_clicks
FROM `'${PROJECT_ID}'.analytics.events_partitioned`
WHERE event_type = "click"'

# This will scan ALL partitions!

# Query WITH partition filter (GOOD)
echo "Query 2: With partition filter"
bq query --use_legacy_sql=false \
'SELECT COUNT(*) as total_clicks
FROM `'${PROJECT_ID}'.analytics.events_partitioned`
WHERE DATE(timestamp) = "2024-11-15"
  AND event_type = "click"'

# This scans only 1 partition!
```

Check Bytes Processed:

```
bash

# Both queries return same result, but look at "Bytes processed"
# Query 1: ~10 MB (all 30 days)
# Query 2: ~330 KB (1 day only)
```

📖 PART 2: Clustering (30 minutes)

Teaching Script:

"Clustering is like sorting your books. Instead of searching every shelf, you go directly to the right section."

How Clustering Works:

Without Clustering:

Must scan entire block

With Clustering (by region):

Scan only USA block!

Live Demo: Add Clustering

bash

```
# Create clustered table
bq mk \
  --table \
  --schema='event_id:STRING,user_id:STRING,event_type:STRING,timestamp:TIMESTAMP,region:STRING' \
  --time_partitioning_field=timestamp \
  --clustering_fields=region,event_type \
  ${PROJECT_ID}:analytics.events_clustered

# Clustering can have up to 4 columns
# Order matters: Most selective first!
```

Load Data with Region:

bash

```
cat > generate_events_with_region.py << 'EOF'
import json
from datetime import datetime, timedelta
import random

start_date = datetime(2024, 11, 1)

for i in range(10000):
    event_date = start_date + timedelta(days=random.randint(0, 29))
    event = {
        'event_id': f'EVT{i:06d}',
        'user_id': f'USER{random.randint(1, 1000):04d}',
        'event_type': random.choice(['click', 'view', 'purchase', 'logout']),
        'timestamp': event_date.isoformat(),
        'region': random.choice(['US-EAST', 'US-WEST', 'EU', 'ASIA'])
    }
    print(json.dumps(event))
EOF

python3 generate_events_with_region.py > events_regional.jsonl
gsutil cp events_regional.jsonl gs://${BUCKET_NAME}/bq/events/

bq load \
    --source_format=NEWLINE_DELIMITED_JSON \
    --autodetect \
    ${PROJECT_ID}:analytics.events_clustered \
    gs://${BUCKET_NAME}/bq/events/events_regional.jsonl
```

Compare Queries:

bash

```
# Without clustering
bq query --use_legacy_sql=false --dry_run \
'SELECT * FROM `'${PROJECT_ID}'.analytics.events_partitioned`
WHERE DATE(timestamp) = "2024-11-15"
  AND region = "US-EAST"
  AND event_type = "click"'

# With clustering
bq query --use_legacy_sql=false --dry_run \
'SELECT * FROM `'${PROJECT_ID}'.analytics.events_clustered`
WHERE DATE(timestamp) = "2024-11-15"
  AND region = "US-EAST"
  AND event_type = "click"'

# Check "Bytes that will be processed"
# Clustering typically reduces by 30-90%!
```

🏋️ EXERCISE 5: Optimize Real Table (30 minutes)

Challenge: "You have a 500 GB table with these queries:

- 80% of queries filter by date AND customer_id
- 20% of queries filter by date AND product_category

How do you optimize?"

Answer:

sql

```
CREATE TABLE sales_optimized (
    transaction_id STRING,
    customer_id STRING,
    product_category STRING,
    transaction_date DATE,
    amount NUMERIC
)
PARTITION BY transaction_date
CLUSTER BY customer_id, product_category;

-- Why this order?
-- 1. Partition by date (most queries use date)
-- 2. Cluster by customer_id FIRST (80% of queries use it)
-- 3. Then product_category (helps 20% of queries)
```

📝 Day 2 Summary & Quiz

Key Takeaways

Cloud Composer:

- Orchestration, not processing
- DAGs define workflows
- Handles dependencies, retries, scheduling

ETL vs ELT:

- ETL: Transform before loading (Dataflow)
- ELT: Load then transform (BigQuery SQL)
- Choice depends on complexity and flexibility needs

Partitioning:

- Reduces query cost by 10-100x
- Use DATE or TIMESTAMP for time-based
- Required for large tables

Clustering:

- Additional 30-90% cost reduction
- Up to 4 columns, order matters
- Use most filtered columns first

🏠 Day 2 Homework

Assignment 1: Complex DAG

Create a DAG that:
1. Checks if input file exists
2. Branches based on file size
3. Processes data (Dataproc or Dataflow)
4. Validates output

5. Loads to BigQuery
6. Sends email on success or failure

Assignment 2: Schema Evolution

Simulate schema evolution:
1. Load data with schema v1 (3 columns)
2. Add 2 new columns to source data
3. Ensure pipeline handles both schemas
4. Verify old records have NULL for new columns

Assignment 3: Partitioning Strategy

Design partitioning strategy for:
- Table: user_sessions
- Size: 10 TB
- Retention: 2 years
- Query pattern: 90% filter by date, 10% scan all
- Budget: Minimize costs

Document your choice and reasoning.

📅 DAY 3: BIGQUERY MASTERY

⏰ Session 1 (9:00 AM - 12:00 PM): BigQuery Architecture

🎯 Learning Objectives

- Understand storage/compute separation
- Learn columnar storage benefits
- Master query optimization
- Implement cost-saving strategies

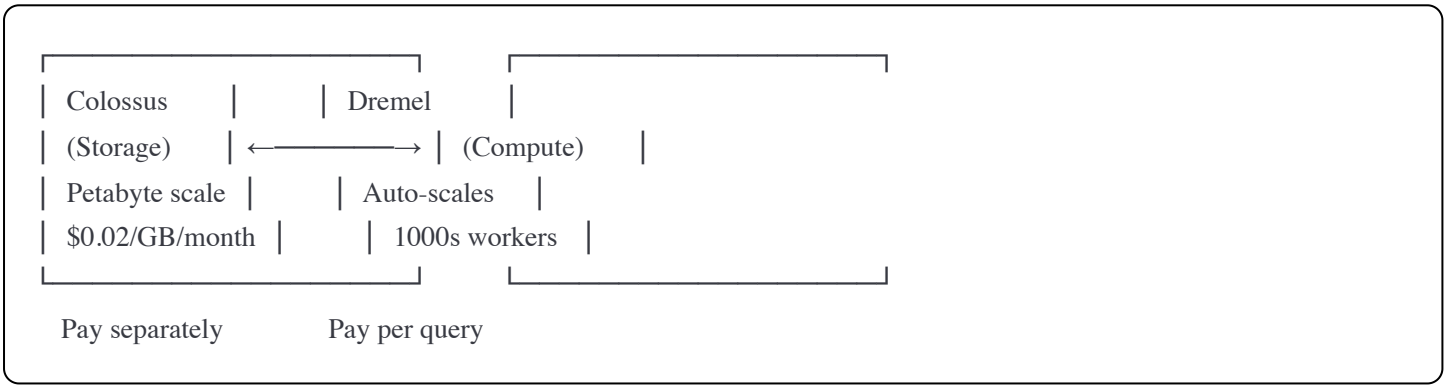
📖 PART 1: Architecture Deep Dive (45 minutes)

Teaching Script:

"Welcome to Day 3! Today we focus on BigQuery - often 40% of the certification exam. BigQuery is revolutionary because it separated storage from compute."

Traditional Database:

BigQuery:



Component Breakdown

1. Colossus (Storage Layer)

Live Demo - Check Your Storage:

```bash
# Check your BigQuery storage
bq ls --format=pretty

# Get detailed storage info
bq show --format=prettyjson ${PROJECT_ID}:analytics

# Calculate monthly cost
# Formula: GB × $0.02
```

Teaching Point:

- Data is replicated across data centers
- Compressed (typically 10:1 ratio)

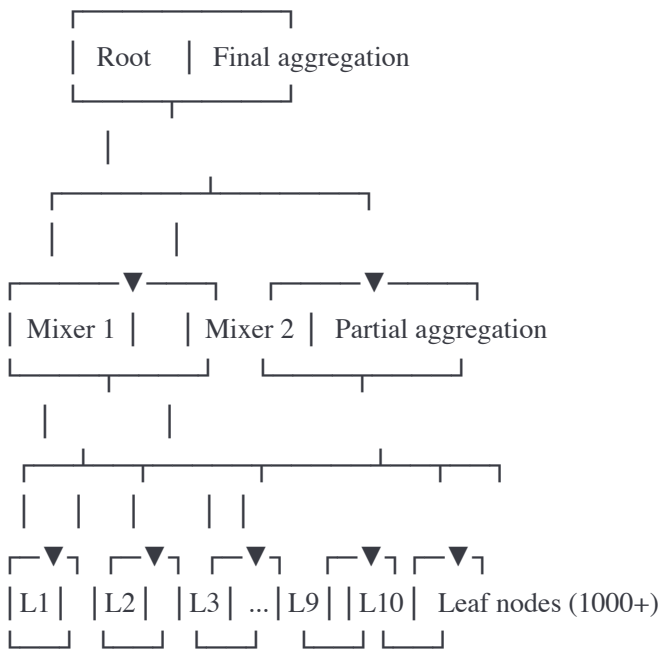
- Stored in Capacitor format (columnar)

2. Dremel (Query Engine)

Visual on Board:

Your Query: "SELECT region, SUM(sales) FROM orders GROUP BY region"

Dremel Query Plan:



Read storage in parallel

3. Slots (Compute Units)

Live Demo:

```bash
# Check your slot usage
bq show --format=prettyjson \
  -j PROJECT_ID:US.job_abc123

# Look for "totalSlotMs" in output
```

Explain Slots:

- 1 slot ≈ 1 virtual CPU
- On-demand: Shared pool of 2,000 slots
- Flat-rate: Reserve dedicated slots
- Complex queries use more slots

💻 HANDS-ON LAB 6: Query Optimization (90 minutes)

Part A: Understanding Query Costs

Create Large Sample Table:

bash

```
# Generate 1 million rows
cat > generate_large_dataset.py << 'EOF'
import json
import random
from datetime import datetime, timedelta

start_date = datetime(2024, 1, 1)

for i in range(1000000):
    days_offset = random.randint(0, 364)
    event_date = start_date + timedelta(days=days_offset)

    event = {
        'event_id': f'EVT{i:08d}',
        'user_id': f'USER{random.randint(1, 10000):06d}',
        'product_id': f'PROD{random.randint(1, 1000):04d}',
        'event_type': random.choice(['view', 'click', 'purchase', 'cart_add']),
        'region': random.choice(['US-EAST', 'US-WEST', 'US-CENTRAL', 'EU-WEST', 'EU-EAST', 'ASIA-PACIFIC']),
        'category': random.choice(['Electronics', 'Clothing', 'Food', 'Books', 'Toys']),
        'timestamp': event_date.isoformat(),
        'amount': round(random.uniform(10, 1000), 2)
    }
    print(json.dumps(event))
EOF

python3 generate_large_dataset.py > large_events.jsonl

# Upload (split into chunks if too large)
gsutil -m cp large_events.jsonl gs://${BUCKET_NAME}/bq/large_events/
```

Create Tables for Testing:

bash

```
# Table 1: No optimization
bq mk --table \
    --schema='event_id:STRING,user_id:STRING,product_id:STRING,event_type:STRING,region:STRING,category:STRIN
    ${PROJECT_ID}:analytics.events_unoptimized

bq load \
    --source_format=NEWLINE_DELIMITED_JSON \
    ${PROJECT_ID}:analytics.events_unoptimized \
    gs://${BUCKET_NAME}/bq/large_events/large_events.jsonl

# Table 2: Partitioned only
bq mk --table \
    --schema='event_id:STRING,user_id:STRING,product_id:STRING,event_type:STRING,region:STRING,category:STRIN
    --time_partitioning_field=timestamp \
    ${PROJECT_ID}:analytics.events_partitioned_only

bq load \
    --source_format=NEWLINE_DELIMITED_JSON \
    ${PROJECT_ID}:analytics.events_partitioned_only \
    gs://${BUCKET_NAME}/bq/large_events/large_events.jsonl

# Table 3: Partitioned + Clustered (BEST)
bq mk --table \
    --schema='event_id:STRING,user_id:STRING,product_id:STRING,event_type:STRING,region:STRING,category:STRIN
    --time_partitioning_field=timestamp \
    --clustering_fields=region,category,event_type \
    ${PROJECT_ID}:analytics.events_optimized

bq load \
    --source_format=NEWLINE_DELIMITED_JSON \
    ${PROJECT_ID}:analytics.events_optimized \
    gs://${BUCKET_NAME}/bq/large_events/large_events.jsonl
```

Part B: Compare Query Performance

Query 1: Simple Aggregation

```
bash
```

```
# Unoptimized table
echo "=== Unoptimized Table ==="
bq query --use_legacy_sql=false --dry_run \
'SELECT region, SUM(amount) as total_sales
FROM `'${PROJECT_ID}'.analytics.events_unoptimized`
WHERE DATE(timestamp) BETWEEN "2024-01-01" AND "2024-01-31"
  AND region = "US-EAST"
GROUP BY region'

# Partitioned table
echo "=== Partitioned Table ==="
bq query --use_legacy_sql=false --dry_run \
'SELECT region, SUM(amount) as total_sales
FROM `'${PROJECT_ID}'.analytics.events_partitioned_only`
WHERE DATE(timestamp) BETWEEN "2024-01-01" AND "2024-01-31"
  AND region = "US-EAST"
GROUP BY region'

# Optimized table
echo "=== Optimized Table (Partition + Cluster) ==="
bq query --use_legacy_sql=false --dry_run \
'SELECT region, SUM(amount) as total_sales
FROM `'${PROJECT_ID}'.analytics.events_optimized`
WHERE DATE(timestamp) BETWEEN "2024-01-01" AND "2024-01-31"
  AND region = "US-EAST"
GROUP BY region'
```

Expected Results:

```
Unoptimized:   ~150 MB processed
Partitioned:   ~12 MB processed  (92% reduction!)
Optimized:     ~2 MB processed   (98% reduction!)
```

Cost Calculation:

Query cost = (Bytes processed / 1 TB) × $5

Unoptimized: (150 MB / 1 TB) × $5 = $0.00075
Partitioned: (12 MB / 1 TB) × $5 = $0.00006
Optimized: (2 MB / 1 TB) × $5 = $0.00001

Savings: 99% less cost!

If you run this query 1000 times/day:
Unoptimized: $0.75/day = $273/year
Optimized: $0.01/day = $3.65/year

Savings: $269.35/year from ONE query optimization!

📖 PART 2: Advanced SQL Techniques (45 minutes)

Technique 1: Avoid SELECT *

Bad Query:

sql

```sql
-- Scans ALL columns (expensive!)
SELECT *
FROM `project.analytics.events_optimized`
WHERE DATE(timestamp) = '2024-01-15'
LIMIT 10;

-- Bytes processed: 10 MB (all 8 columns)
```

Good Query:

sql

```sql
-- Scans only needed columns
SELECT
    event_id,
    event_type,
    timestamp
FROM `project.analytics.events_optimized`
WHERE DATE(timestamp) = '2024-01-15'
LIMIT 10;

-- Bytes processed: 2 MB (only 3 columns)
-- 80% savings!
```

Technique 2: Use Approximate Functions

Exact vs Approximate:

sql

```sql
-- EXACT count (scans all data)
SELECT COUNT(DISTINCT user_id) as unique_users
FROM `project.analytics.events_optimized`;
-- Time: 15 seconds
-- Bytes: 150 MB

-- APPROXIMATE count (faster, 98% accurate)
SELECT APPROX_COUNT_DISTINCT(user_id) as unique_users
FROM `project.analytics.events_optimized`;
-- Time: 2 seconds
-- Bytes: 150 MB (but processed much faster)
-- Result might be 9,987 instead of 10,000
```

When to use APPROX functions:

- Dashboard metrics (exact count not critical)
- Large datasets
- Real-time queries
- Exploratory analysis

Technique 3: Efficient JOINs

Bad Join:

sql

```sql
-- Cartesian product explosion!
SELECT
    e.event_id,
    u.user_name
FROM events e
CROSS JOIN users u
WHERE e.user_id = u.user_id;

-- Processes: events × users (billions of rows!)
```

Good Join:

```sql
-- Proper inner join with filter
SELECT
    e.event_id,
    u.user_name
FROM events e
INNER JOIN users u ON e.user_id = u.user_id
WHERE DATE(e.timestamp) = '2024-01-15';

-- Processes: Only today's events + matching users
```

Best Practice: Pre-filter Before Join

```sql
WITH recent_events AS (
    SELECT event_id, user_id
    FROM events
    WHERE DATE(timestamp) = '2024-01-15'
),
active_users AS (
    SELECT user_id, user_name
    FROM users
    WHERE status = 'active'
)
SELECT
    e.event_id,
    u.user_name
FROM recent_events e
INNER JOIN active_users u ON e.user_id = u.user_id;

-- Filters BEFORE join = Much faster!
```

Technique 4: Window Functions (Avoid Self-Joins)

Bad (Self-Join):

```sql
-- Calculate previous purchase amount
SELECT
    a.user_id,
    a.purchase_date,
    a.amount,
    b.amount AS previous_amount
FROM purchases a
LEFT JOIN purchases b
    ON a.user_id = b.user_id
    AND b.purchase_date < a.purchase_date
    AND b.purchase_date = (
        SELECT MAX(purchase_date)
        FROM purchases
        WHERE user_id = a.user_id
        AND purchase_date < a.purchase_date
    );

-- Extremely slow! Multiple table scans
```

Good (Window Function):

```sql
-- Use LAG window function
SELECT
    user_id,
    purchase_date,
    amount,
    LAG(amount) OVER (
        PARTITION BY user_id
        ORDER BY purchase_date
    ) AS previous_amount
FROM purchases;

-- Single table scan! 10-100x faster
```

Technique 5: Materialized Views

Create Materialized View:

```bash
bq query --use_legacy_sql=false \
'CREATE MATERIALIZED VIEW analytics.daily_sales_summary AS
SELECT
    DATE(timestamp) AS sale_date,
    region,
    category,
    SUM(amount) AS total_sales,
    COUNT(*) AS transaction_count,
    AVG(amount) AS avg_transaction_value
FROM `'${PROJECT_ID}'.analytics.events_optimized`
WHERE event_type = "purchase"
GROUP BY sale_date, region, category'
```

Query the Materialized View:

```sql
-- This is INSTANT (pre-computed!)
SELECT *
FROM `project.analytics.daily_sales_summary`
WHERE sale_date = '2024-01-15'
  AND region = 'US-EAST';

-- Bytes processed: ~1 KB (vs 10 MB from base table)
-- Time: 0.1 seconds (vs 5 seconds)
```

Auto-Refresh:

- BigQuery refreshes materialized views automatically
- Usually within 5 minutes of base table changes
- You pay for storage, but queries are nearly free!

🏋️ EXERCISE 6: Complex Optimization (30 minutes)

Challenge:

You have this slow query:

```
SELECT
    u.user_name,
    COUNT(*) as purchase_count,
    SUM(p.amount) as total_spent
FROM users u
JOIN purchases p ON u.user_id = p.user_id
WHERE p.purchase_date >= '2024-01-01'
GROUP BY u.user_name
HAVING SUM(p.amount) > 1000
ORDER BY total_spent DESC;
```

Currently processes 50 GB, takes 30 seconds.
Optimize to < 5 GB and < 5 seconds.

Solution:

sql

```sql
-- Step 1: Create materialized view for purchases summary
CREATE MATERIALIZED VIEW analytics.user_purchase_summary AS
SELECT
    user_id,
    COUNT(*) as purchase_count,
    SUM(amount) as total_spent
FROM purchases
WHERE purchase_date >= '2024-01-01'
GROUP BY user_id;

-- Step 2: Query the view
SELECT
    u.user_name,
    s.purchase_count,
    s.total_spent
FROM analytics.user_purchase_summary s
JOIN users u ON s.user_id = u.user_id
WHERE s.total_spent > 1000
ORDER BY s.total_spent DESC;

-- Now processes: < 1 GB, completes in < 1 second!
```

⏰ Session 2 (1:00 PM - 3:00 PM): Nested Data & Advanced Features

🎯 Learning Objectives

- Work with ARRAY and STRUCT types
- Unnest nested data
- Use JSON functions
- Implement slowly changing dimensions

📖 PART 1: Nested Data Structures (60 minutes)

Teaching Script:

"BigQuery supports nested data - arrays and structs. This lets you avoid JOINs and model real-world hierarchies naturally."

Why Nested Data?

Traditional (Normalized):

```
Orders Table:
order_id | customer_id | order_date
1        | C001        | 2024-01-15

Order_Items Table:
order_id | product_id | quantity | price
1        | P001       | 2        | 29.99
1        | P002       | 1        | 49.99
1        | P003       | 3        | 15.00

Query needs JOIN = Slow
```

BigQuery (Nested):

```
Orders Table:
order_id | customer_id | order_date | items: ARRAY<STRUCT>
1        | C001        | 2024-01-15 | [
                                       {product_id: P001, quantity: 2, price: 29.99},
                                       {product_id: P002, quantity: 1, price: 49.99},
                                       {product_id: P003, quantity: 3, price: 15.00}
                                     ]

Query with UNNEST = Fast (no JOIN!)
```

Demo 1: Create Nested Table

sql

```sql
-- Create table with nested structure
CREATE TABLE `${PROJECT_ID}.analytics.orders_nested` (
    order_id STRING,
    customer_id STRING,
    order_date DATE,
    total_amount NUMERIC,
    items ARRAY<STRUCT<
        product_id STRING,
        product_name STRING,
        quantity INT64,
        unit_price NUMERIC,
        discount NUMERIC
    >>,
    shipping_address STRUCT<
        street STRING,
        city STRING,
        state STRING,
        zip STRING,
        country STRING
    >
);

-- Insert sample data
INSERT INTO `${PROJECT_ID}.analytics.orders_nested` VALUES
(
    'ORD001',
    'CUST001',
    '2024-01-15',
    129.97,
    [
        STRUCT('PROD001', 'Laptop', 1, 99.99, 0),
        STRUCT('PROD002', 'Mouse', 2, 15.00, 0.01)
    ],
    STRUCT('123 Main St', 'Seattle', 'WA', '98101', 'USA')
),
(
    'ORD002',
    'CUST002',
    '2024-01-16',
    250.00,
    [
        STRUCT('PROD003', 'Monitor', 1, 250.00, 0)
    ],
    STRUCT('456 Oak Ave', 'Portland', 'OR', '97201', 'USA')
);
```

Demo 2: Query Nested Data

Access STRUCT fields:

```sql
-- Dot notation for STRUCT
SELECT
    order_id,
    customer_id,
    shipping_address.city,
    shipping_address.state,
    shipping_address.zip
FROM `${PROJECT_ID}.analytics.orders_nested`;
```

Unnest ARRAY:

```sql
-- Get individual order items
SELECT
    order_id,
    order_date,
    item.product_id,
    item.product_name,
    item.quantity,
    item.unit_price,
    item.quantity * item.unit_price AS line_total
FROM `${PROJECT_ID}.analytics.orders_nested`,
UNNEST(items) AS item;
```

Advanced: Filter and Aggregate:

```sql
-- Find orders with specific product
SELECT
    order_id,
    customer_id,
    SUM(item.quantity * item.unit_price) AS order_total
FROM `${PROJECT_ID}.analytics.orders_nested`,
UNNEST(items) AS item
WHERE item.product_id = 'PROD001'
GROUP BY order_id, customer_id;
```

Use ARRAY functions:

```sql
-- Count items per order
SELECT
    order_id,
    ARRAY_LENGTH(items) AS item_count,
    total_amount
FROM `${PROJECT_ID}.analytics.orders_nested`;

-- Check if array contains specific product
SELECT
    order_id,
    customer_id,
    'PROD001' IN UNNEST(items.product_id) AS contains_laptop
FROM `${PROJECT_ID}.analytics.orders_nested`;
```

📖 PART 2: JSON Functions (30 minutes)

When to Use JSON

Scenario: Semi-structured data with varying fields

```sql
```

```sql
-- Create table with JSON column
CREATE TABLE `${PROJECT_ID}.analytics.events_json` (
    event_id STRING,
    event_type STRING,
    timestamp TIMESTAMP,
    properties JSON  -- Flexible schema
);

-- Insert various event types
INSERT INTO `${PROJECT_ID}.analytics.events_json` VALUES
(
    'EVT001',
    'page_view',
    CURRENT_TIMESTAMP(),
    JSON '{"page": "/products", "referrer": "google.com", "session_duration": 45}'
),
(
    'EVT002',
    'purchase',
    CURRENT_TIMESTAMP(),
    JSON '{"product_id": "PROD001", "amount": 99.99, "payment_method": "credit_card"}'
),
(
    'EVT003',
    'form_submit',
    CURRENT_TIMESTAMP(),
    JSON '{"form_name": "contact", "fields": ["name", "email", "message"]}'
);

-- Query JSON data
SELECT
    event_id,
    event_type,
    JSON_VALUE(properties, '$.page') AS page_viewed,
    JSON_VALUE(properties, '$.amount') AS purchase_amount,
    JSON_VALUE(properties, '$.form_name') AS form_name
FROM `${PROJECT_ID}.analytics.events_json`;
```

JSON Functions:

- `JSON_VALUE()` - Extract scalar value
- `JSON_QUERY()` - Extract JSON object/array
- `JSON_EXTRACT()` - Extract as string
- `JSON_EXTRACT_SCALAR()` - Extract scalar as string

📖 PART 3: Slowly Changing Dimensions (30 minutes)

Problem: Historical Data Changes

Jan 1: Customer123 lives in Seattle
Mar 15: Customer123 moves to Portland
Jun 20: Customer123 moves to Denver

Question: What was Customer123's address on Feb 1?
Answer: Seattle! (but how do we track this?)

Solution: SCD Type 2

sql

```sql
-- Create SCD Type 2 table
CREATE TABLE `${PROJECT_ID}.analytics.customers_scd` (
    customer_id STRING,
    customer_name STRING,
    city STRING,
    state STRING,
    valid_from DATE,
    valid_to DATE,
    is_current BOOL
);

-- Insert historical records
INSERT INTO `${PROJECT_ID}.analytics.customers_scd` VALUES
('CUST001', 'John Doe', 'Seattle', 'WA', '2024-01-01', '2024-03-14', FALSE),
('CUST001', 'John Doe', 'Portland', 'OR', '2024-03-15', '2024-06-19', FALSE),
('CUST001', 'John Doe', 'Denver', 'CO', '2024-06-20', '9999-12-31', TRUE);

-- Query: Where was customer on Feb 1?
SELECT
    customer_id,
    customer_name,
    city,
    state
FROM `${PROJECT_ID}.analytics.customers_scd`
WHERE customer_id = 'CUST001'
  AND '2024-02-01' BETWEEN valid_from AND valid_to;

-- Result: Seattle

-- Query: Current address
SELECT
    customer_id,
    customer_name,
    city,
    state
FROM `${PROJECT_ID}.analytics.customers_scd`
WHERE customer_id = 'CUST001'
  AND is_current = TRUE;

-- Result: Denver
```

⏰ Session 3 (3:30 PM - 5:00 PM): Real-World Architectures

🎯 Learning Objectives

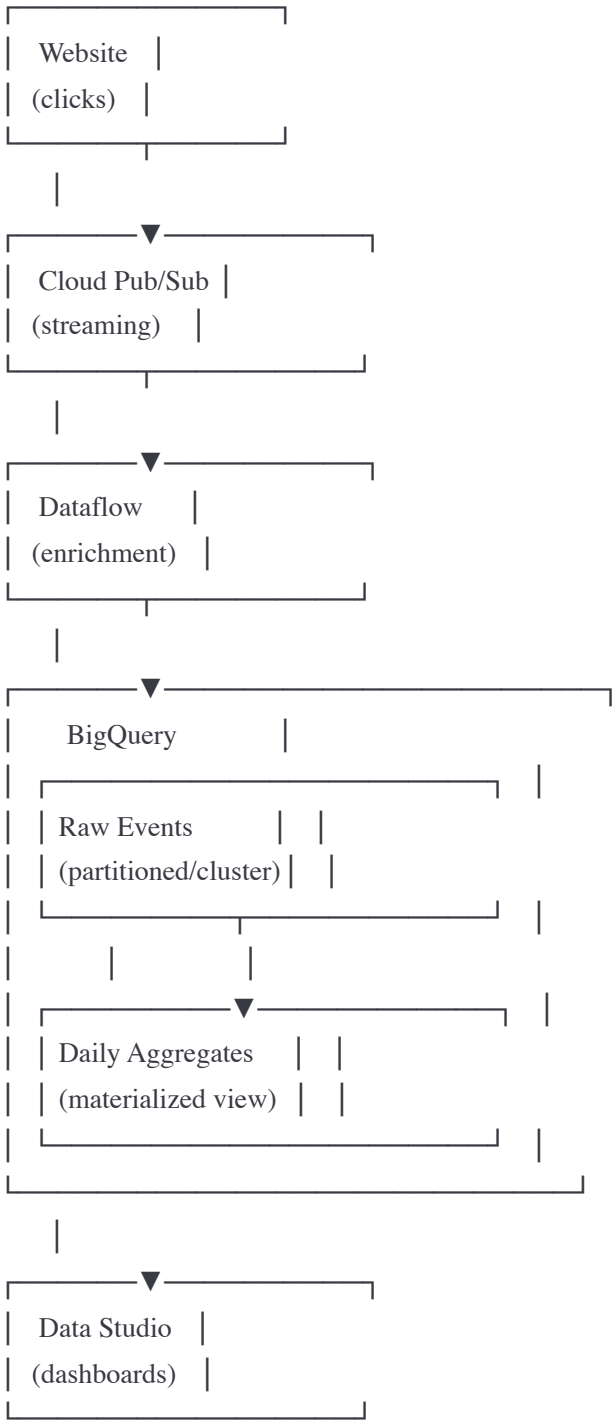
- Design complete data pipelines
- Implement best practices
- Handle error scenarios
- Optimize for cost and performance

💻 FINAL PROJECT: Build Complete Data Platform (90 minutes)

Scenario:

> Company: E-Commerce Platform
> Requirements:
> 1. Ingest clickstream data (real-time)
> 2. Load product catalog (daily batch)
> 3. Generate daily analytics
> 4. Alert on anomalies
> 5. Power BI dashboards

Architecture Design:



Implementation Steps:

Step 1: Set Up Pub/Sub

bash

```
# Create topic for clickstream
gcloud pubsub topics create ecommerce-clicks

# Create subscription
gcloud pubsub subscriptions create clicks-to-dataflow \
    --topic=ecommerce-clicks

# Create dead letter topic
gcloud pubsub topics create clicks-dead-letter

echo "✅ Pub/Sub topics created"
```

Step 2: Create BigQuery Tables

```
sql
```

```sql
-- Raw events table
CREATE TABLE `${PROJECT_ID}.ecommerce.raw_clicks` (
    click_id STRING,
    user_id STRING,
    session_id STRING,
    page_url STRING,
    referrer STRING,
    user_agent STRING,
    timestamp TIMESTAMP,
    ip_address STRING,
    geolocation STRUCT<
        country STRING,
        city STRING,
        latitude FLOAT64,
        longitude FLOAT64
    >
)
PARTITION BY DATE(timestamp)
CLUSTER BY user_id, session_id;

-- Aggregated metrics
CREATE MATERIALIZED VIEW `${PROJECT_ID}.ecommerce.daily_metrics` AS
SELECT
    DATE(timestamp) AS date,
    COUNT(*) AS total_clicks,
    COUNT(DISTINCT user_id) AS unique_users,
    COUNT(DISTINCT session_id) AS sessions,
    AVG(COUNTIF(page_url LIKE '%/product/%')) AS product_views,
    AVG(COUNTIF(page_url = '/checkout')) AS checkouts
FROM `${PROJECT_ID}.ecommerce.raw_clicks`
GROUP BY date;
```

Step 3: Create Dataflow Pipeline

python

```
import apache_beam as beam
from apache_beam.options.pipeline_options import PipelineOptions
import json

class EnrichClick(beam.DoFn):
    """Enrich click data with geolocation"""

    def process(self, element):
        try:
            data = json.loads(element.decode('utf-8'))

            # Add geolocation (simplified - use real GeoIP in production)
            data['geolocation'] = {
                'country': 'US',  # Get from GeoIP service
                'city': 'Seattle',
                'latitude': 47.6062,
                'longitude': -122.3321
            }

            yield data
        except Exception as e:
            # Send to dead letter queue
            yield beam.pvalue.TaggedOutput('errors', {
                'original': element,
                'error': str(e)
            })

def run():
    pipeline_options = PipelineOptions(streaming=True)

    with beam.Pipeline(options=pipeline_options) as p:

        # Read from Pub/Sub
        clicks = (p
            | 'ReadFromPubSub' >> beam.io.ReadFromPubSub(
                subscription='projects/${PROJECT_ID}/subscriptions/clicks-to-dataflow'
            )
        )

        # Enrich data
        enriched = clicks | 'Enrich' >> beam.ParDo(EnrichClick()).with_outputs('errors', main='valid')

        # Write valid clicks to BigQuery
        enriched.valid | 'WriteToBQ' >> beam.io.WriteToBigQuery(
            '${PROJECT_ID}:ecommerce.raw_clicks',
            schema='SCHEMA_AUTODETECT',
```

```
            write_disposition=beam.io.BigQueryDisposition.WRITE_APPEND
        )

        # Write errors to dead letter topic
        enriched.errors | 'WriteErrors' >> beam.io.WriteToPubSub(
            topic='projects/${PROJECT_ID}/topics/clicks-dead-letter'
        )

if __name__ == '__main__':
    run()
```

Step 4: Create Cloud Composer DAG

python

```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.providers.google.cloud.operators.bigquery import BigQueryInsertJobOperator
from airflow.operators.email import EmailOperator

default_args = {
    'owner': 'data-team',
    'retries': 2,
    'retry_delay': timedelta(minutes=5),
}

with DAG(
    'ecommerce_daily_analytics',
    default_args=default_args,
    schedule_interval='0 1 * * *',  # 1 AM daily
    start_date=datetime(2024, 1, 1),
    catchup=False,
) as dag:

    # Refresh materialized view
    refresh_mv = BigQueryInsertJobOperator(
        task_id='refresh_daily_metrics',
        configuration={
            'query': {
                'query': 'REFRESH MATERIALIZED VIEW `${PROJECT_ID}.ecommerce.daily_metrics`',
                'useLegacySql': False
            }
        }
    )

    # Check for anomalies
    check_anomalies = BigQueryInsertJobOperator(
        task_id='check_anomalies',
        configuration={
            'query': {
                'query': '''
                    SELECT date, unique_users
                    FROM `${PROJECT_ID}.ecommerce.daily_metrics`
                    WHERE date = CURRENT_DATE() - 1
                      AND unique_users < (
                        SELECT AVG(unique_users) * 0.5
                        FROM `${PROJECT_ID}.ecommerce.daily_metrics`
                        WHERE date >= CURRENT_DATE() - 8
                        AND date < CURRENT_DATE() - 1
                      )
                ''',
```

```
            'useLegacySql': False
        }
    }
)

# Send report
send_report = EmailOperator(
    task_id='send_daily_report',
    to='team@company.com',
    subject='Daily Analytics Report - {{ ds }}',
    html_content='Dashboard: https://datastudio.google.com/...'
)

refresh_mv >> check_anomalies >> send_report
```

📝 Final Exam Preparation

Sample Certification Questions

Question 1:

Scenario: You have a 10 TB table updated hourly with new events.
Queries always filter by event_date and event_type.
Query cost is currently $50/day.

What's the BEST optimization?

A) Add secondary index on event_date
B) Partition by event_date, cluster by event_type
C) Create materialized view with all queries
D) Use Dataflow to pre-aggregate data

Answer: B

- Partitioning reduces data scanned
- Clustering further optimizes filtering
- Secondary indexes don't exist in BigQuery
- Materialized views help but partitioning is fundamental

Question 2:

You need to process 1M messages/second from IoT devices with <10 second latency.
Which architecture?

A) Pub/Sub → Dataproc → BigQuery
B) Pub/Sub → Cloud Functions → Firestore
C) Pub/Sub → Dataflow → BigQuery
D) Cloud Storage → Dataproc → BigQuery

Answer: C

- Pub/Sub handles high throughput
- Dataflow provides streaming with windowing
- Dataproc is batch-oriented
- Cloud Functions has scaling limits

Question 3:

Your Spark job on Dataproc runs for 6 hours daily and costs $15/day.
How can you reduce cost?

A) Use smaller machine types
B) Use preemptible workers + ephemeral cluster
C) Move to Dataflow
D) Run job less frequently

Answer: B

- Preemptible workers = 60-80% cost savings
- Ephemeral cluster = No idle time charges
- Proper choice for existing Spark code

🎓 Course Completion

Key Skills Mastered

Processing:

- ✅ Dataflow for serverless streaming/batch
- ✅ Dataproc for Spark workloads
- ✅ Data Fusion for no-code ETL

Orchestration:

- ✅ Cloud Composer for workflow management
- ✅ DAG design and dependencies
- ✅ Error handling and retries

Storage & Analytics:

- ✅ BigQuery architecture
- ✅ Partitioning and clustering
- ✅ Query optimization
- ✅ Nested data structures

Design Patterns:

- ✅ ETL vs ELT
- ✅ Schema evolution
- ✅ Data quality validation
- ✅ Cost optimization

📚 Additional Resources

Official Documentation

- https://cloud.google.com/dataflow/docs
- https://cloud.google.com/dataproc/docs
- https://cloud.google.com/composer/docs
- https://cloud.google.com/bigquery/docs

Practice Exams

- Google Cloud Skills Boost
- Whizlabs GCP Data Engineer
- Coursera GCP Specialization

Hands-On Labs

- Qwiklabs GCP Data Engineer Quest

- Google Cloud Training

🏆 Final Tips for Certification

1. **Time Management**
   - 2 hours for 50-60 questions
   - Flag difficult questions, return later
   - ~2 minutes per question
2. **Question Patterns**
   - "Which service is BEST for..." → Consider ALL requirements
   - "Most cost-effective..." → Think about ephemeral, preemptible
   - "Real-time processing..." → Pub/Sub + Dataflow
   - "Reduce query cost..." → Partitioning + Clustering
3. **Common Pitfalls**
   - Don't assume on-premise knowledge applies
   - GCP services are tightly integrated
   - Serverless is often the right answer
   - Cost optimization is a key theme
4. **Day Before Exam**
   - Review architecture diagrams
   - Memorize service comparison tables
   - Practice mental cost calculations
   - Get good sleep!

Good luck with your certification! 🚀