**Very Important Final Assessment**

**Incremental Data Processing with CDC (Concept)**

[**https://colab.research.google.com/drive/1QFHZse6MOalwfZ18VVCCJbT9Cne9EGo6?usp=sharing**](https://colab.research.google.com/drive/1QFHZse6MOalwfZ18VVCCJbT9Cne9EGo6?usp=sharing)

**Aim**

To continuously capture and apply only changes (inserts, updates, deletes) from a source system using Change Data Capture (CDC), and maintain analytics tables up to date with low latency.

**Procedure**

1. Enable CDC at the source database (MySQL binlog or PostgreSQL WAL).
2. Use a CDC tool like **Debezium** to capture change events and publish them into **Kafka topics**.
3. Read the Kafka topics in a **Spark/Flink** streaming job.
4. Parse each event to extract operation type (insert, update, delete).
5. Upsert data into target storage (Delta Lake, Hudi, or Iceberg) using primary keys.
6. Apply **idempotent merges** and maintain **watermarks** to handle late or out-of-order events.
7. Periodically compact data and validate record counts.
8. Expose the incremental tables for real-time BI or ML consumption.

**Implementation**

* Tools: **Debezium + Kafka + Spark Structured Streaming + Delta Lake**
* CDC events are read from Kafka, processed using Spark, and upserted into a Delta table.
* Example logic:
* MERGE INTO target\_table AS t
* USING updates AS s
* ON t.id = s.id
* WHEN MATCHED THEN UPDATE SET \*
* WHEN NOT MATCHED THEN INSERT \*
* Kafka ensures durability and order, while Delta maintains ACID transactions.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Result**

The system continuously updates the target analytics table in near real-time, ensuring data freshness and consistency without full reloads.

**Conclusion**

CDC-based incremental processing provides an efficient way to synchronize data between source and analytical systems with minimal latency, improving scalability and reducing resource costs.

**Credit Card Fraud Preprocessing (Spark + Kaggle)**

[**https://colab.research.google.com/drive/1ZsfXc9YTLqf6drKiGybbozC9ArFUJBx4?usp=sharing**](https://colab.research.google.com/drive/1ZsfXc9YTLqf6drKiGybbozC9ArFUJBx4?usp=sharing)

**Aim**

To clean, transform, and prepare the credit card fraud dataset for machine learning using Apache Spark.

**Procedure**

1. Download the dataset from **Kaggle** using kagglehub.
2. Load it into Spark using inferSchema=True.
3. Check and handle missing values:
   * Numeric columns → impute with mean or 0.
   * String columns → replace with “Unknown”.
4. Remove duplicates and cast numeric types to double.
5. Engineer new features like:
   * TransactionHour = (Time/3600) % 24
   * Amount bucketing or binning
6. Assemble features using VectorAssembler.
7. Scale features using StandardScaler.
8. Save the cleaned dataset as CSV for further analysis.

**Implementation**

* Tools: **PySpark, KaggleHub**
* Sample code:
* df = spark.read.csv("creditcard.csv", header=True, inferSchema=True)
* df = df.dropDuplicates()
* df = df.withColumn("TransactionHour", (col("Time")/3600) % 24)
* assembler = VectorAssembler(inputCols=feature\_cols, outputCol="features")
* df\_vec = assembler.transform(df)
* scaler = StandardScaler(inputCol="features", outputCol="scaled\_features")
* df\_scaled = scaler.fit(df\_vec).transform(df\_vec)
* df\_scaled.write.csv("cleaned\_creditcard.csv", header=True)

A computer screen shot of text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Result**

A cleaned and standardized dataset with new engineered features is successfully generated and stored for model training.

**Conclusion**

The preprocessing pipeline improved data quality and consistency, ensuring better input for downstream ML models and enhancing prediction accuracy for fraud detection.

**Real-Time Streaming Inference (Spark Structured Streaming + scikit-learn)**

[**https://colab.research.google.com/drive/1BXjE4gPdXsLgV\_Yd32KbxD-K9cx6QnZ-?usp=sharing**](https://colab.research.google.com/drive/1BXjE4gPdXsLgV_Yd32KbxD-K9cx6QnZ-?usp=sharing)

**Aim**

To perform real-time predictions on streaming data using a pre-trained RandomForest model integrated with Spark Structured Streaming.

**Procedure**

1. Train a RandomForest model using synthetic sensor data.
2. Save the trained model using **joblib**.
3. Set up directories: input/, output/, checkpoint/.
4. Load streaming data from input/ folder using Spark Structured Streaming.
5. In each micro-batch, convert Spark DataFrame to pandas and apply model prediction.
6. Write output predictions to CSV files in output/ directory.
7. Continuously simulate new incoming CSV files to generate real-time predictions.
8. Stop the query once predictions are complete.

**Implementation**

* Tools: **scikit-learn, Spark Structured Streaming, joblib**
* Code logic:
* def process\_batch(df, epoch):
* pdf = df.toPandas()
* model = joblib.load("rf\_model.pkl")
* pdf["prediction"] = model.predict(pdf[features])
* pdf.to\_csv(f"output/batch\_{epoch}.csv", index=False)
* df\_stream.writeStream.foreachBatch(process\_batch).start()

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Result**

Each new incoming batch of sensor data triggers a micro-batch inference process, and predictions are saved continuously to the output folder.

**Conclusion**

This experiment demonstrates the integration of batch ML models into a streaming environment, enabling real-time decision-making in scenarios like IoT and predictive analytics.

4. **In-Memory Batch Analytics & Caching (Spark)**

[**https://colab.research.google.com/drive/1nftBb73\_C9HMYlUgcz7O6u2Z2x3arhag?usp=sharing**](https://colab.research.google.com/drive/1nftBb73_C9HMYlUgcz7O6u2Z2x3arhag?usp=sharing)

**Aim**

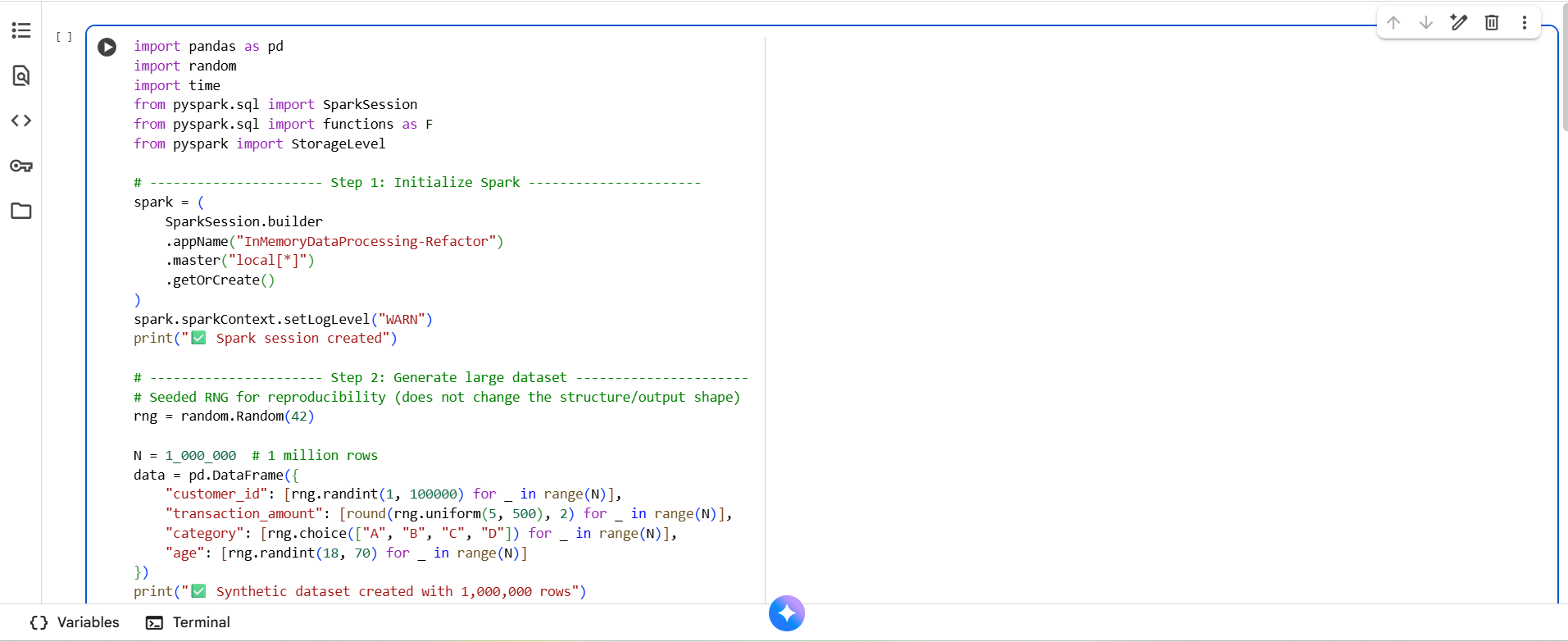
To compare the performance of Spark aggregations with and without in-memory caching on a large synthetic dataset.

**Procedure**

1. Start Spark Session.
2. Generate a dataset of 1 million records (customer, amount, category, age).
3. Perform baseline aggregations without caching and record time.
4. Apply .cache() to the DataFrame and materialize it using .count().
5. Re-run the same aggregation and note the reduced execution time.
6. Optionally, rank top customers by total amount spent.
7. Stop Spark session.

**Implementation**

* Tools: **PySpark, pandas, NumPy**
* Example snippet:
* df.groupBy("category").agg(avg("amount"), sum("amount")).show()
* df.cache()
* df.count()
* df.groupBy("category").agg(avg("amount"), sum("amount")).show()



A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**Result**

Execution time decreased significantly after caching, confirming the advantage of keeping data in memory during repeated operations.

**Conclusion**

In-memory caching in Spark greatly enhances performance for repetitive or iterative analytics tasks by reducing disk I/O and improving computation speed.

Top of Form

Bottom of Form