Iceberg to TigerGraph Data Integration Pipeline

Design Proposal and Architecture Analysis

Rahul Khanna

December 15, 2025

Abstract

In this document, I have proposed a design for implementing a data integration pipeline from Apache Iceberg to the TigerGraph graph. Initially, I discuss my approach of using Apache Spark as the middleware that connects to Apache Iceberg through Databricks Unity Cataloge and pulls the data from S3 buckets. Further, it maps the Iceberg data into TigerGraph compatible data format. Furthermore, I use the REST++ API client provided by the TigerGraph-hosted clients and perform micro-batch operations to push the data into the TigerGraph database. In the end, I discuss in detail different alternatives we can take for this data integration pipeline, and evaluate their use-cases, performance and scalibility and costs and complexity.

1 Design Decision

1.1 Problem Statement

Upon my research, I found that organizations increasingly need to transform relational data stored in data lakes into graph structures for advanced analytics such as fraud detection, recommendation systems, and network analysis. The challenge presented to me, and many engineers, is building a scalable, maintainable pipeline that can:

- Read from Apache Iceberg tables (ACID-compliant table format)
- Leverage Unity Catalog for metadata management and governance
- Transform relational data into graph structures
- Load efficiently into TigerGraph at scale
- Support both initial bulk loads and ongoing incremental updates

1.2 Proposed Solution

After going through the documentation of possible cloud-infrastructural tools (Iceberg, Spark, Kafka Connect, Flink, TigerGraph) and their use cases, I have proposed the following architectural design solution:

1. **Batch processing with Apache Spark** (allows parallel data processing)
2. **Two loading mechanisms**:
   - REST++ API for almost-incremental updates
   - GSQL Loading Jobs for bulk loads (millions of records or initial bulk loading of Iceberg data snapshots)
3. **Push architecture** (Spark reads from Iceberg and pushes to TigerGraph)
4. **Configuration-driven schema mapping** (allowing flexibility to add new schema changes from Iceberg)

1.3 My Design Choices

Decision	Choice	Rationale
Processing Model	Batch + Micro-batch	Cost-effective, proven at scale
Data Flow	Push (Spark)	Leverages available infrastructure
Loading Method	REST++ & GSQL	Depending upon scales of data
Mapping Config	YAML-based	Flexible mapping to TigerGraph

Table 1: Design Decision

1.4 Expected Performance

- **Performance**: Upload millions of records in a few hours (GSQL loading jobs)
- **Micro-Batching**: Achieve almost real-time sync of records (REST++ upsert operations)
- **Flexibility**: Easy addition of new tables via configuration
- **Scalability**: Horizontal scaling via Spark partitioning

2 System Architecture

2.1 High-Level Architecture

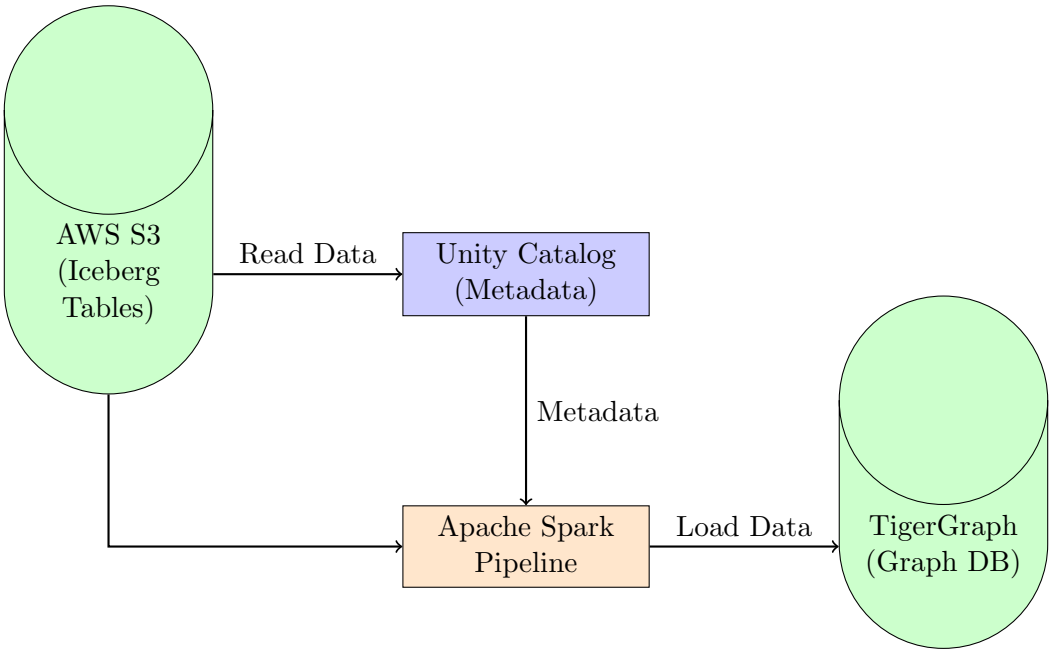

Figure 1: High-Level System Architecture

2.2 Component Overview

Source Layer: Apache Iceberg + Databricks Unity Catalog

Apache Iceberg provides ACID transactions on data lake tables. It also allows time travel and snapshot isolation, with schema evolution support.

While **Databricks Unity Catalog** provides centralized metadata management with fine-grained access control and data lineage tracking.

I assume that Amazon S3 acts as the data lake and object storage, storing files like parquet, avro or CSVs. Iceberg and Unity Catalog managing the data-table format and schema connected to S3 buckets.

Processing Layer: Apache Spark

Why did I choose Spark?

I read a lot about Iceberg and its compatibility and how efficiently Spark works with Iceberg. Even LLMs such as ChatGPT and Claude proposed that I go with Apache Spark as the middleware.

First of all, Iceberg is not built for **streaming or event-driven workflows**. Its architecture is ideal for batch processing. Within its core, on any update and appends, Iceberg has to create a new table snapshot and copy all the data files (Copy on Write or Merge on Read), which can prove to be highly inefficient for highly-frequent event-driven systems. Therefore,

pull-based integration mechanisms may not be ideal.

On the other hand, Apache Spark has been widely used with Apache Iceberg, and these are the following reasons:

- Native Iceberg integration via Spark DataSource APIs
- Unity Catalog strong support in Databricks APIs
- Horizontal scalability via Spark partitioning
- Fault tolerance through RDDs and automatic retries

Target Layer: TigerGraph

TigerGraph is a native Graph Database that can store entities in our relational databases as vertices and the entity relationships as edges.

Over the last week, I learned a lot from its official documentation [TigerGraph Architecture]. TigerGraph supports ACID transactions while also maintaining high parallelism of graph algorithms, achieving millions of records synced per second. It has been widely used across different platforms for enabling graph-based analytics and real-time insights.

TigerGraph Data Loading Methods

Method 1: REST++ API

```
POST /graph/{graph_name}/vertices/{vertex_type}
Body: {
    "vertices": {
        "vertex_id_1": {"attr1": "value1", ...},
        "vertex_id_2": {"attr2": "value2", ...}
    }
}
```

Listing 1: REST++ API Endpoint

Characteristics:
- Rest++ API calls are ideal for low-latency uploads (ms).
- Works well with small-medium uploads.
- Can work well for event-driven or real time uploads.
- Can also work for mini-batching achieving almost event-driven integration.
- Needs a lot of HTTP overhead for each upsert call.

From my understanding, REST++ API calls are ideal for data-pipelines that need **low-latency** and **real-time** updates mostly seen in event-driven systems involving Kafka. These **upsert** API calls can push incremental updates coming from event-driven data-pipelines example Change-Data-Capture from Debezium (or micro-batch payload to save network overhead).

Method 2: GSQL Loading Jobs

```
USE GRAPH EcommerceGraph
CREATE LOADING JOB load_users FOR GRAPH EcommerceGraph {
    DEFINE FILENAME user_file = "s3://bucket/users.csv";
    LOAD user_file TO VERTEX User VALUES(
        $"user_id", $"username", $"email"
    ) USING SEPARATOR=",", HEADER="true";
}
```

```
RUN LOADING JOB load_users
```

Listing 2: GSQL Loading Job Example

Characteristics:
- GSQL loading jobs are perfect when dealing with huge graph records (million to billion records).
- These uploads are not latency-friendly and are thus are not ideal for event-driven systems as seen for REST++.
- However, these jobs can upload as many records as we want through highly-parallelized ingestion performed at staging routines.
- It is also flexible and can read data from many sources such as local files (parquet, CSVs), S3 and Hadoop.
- No need to handle duplicate records and also no network HTTP overheads as seen in REST++ API calls.

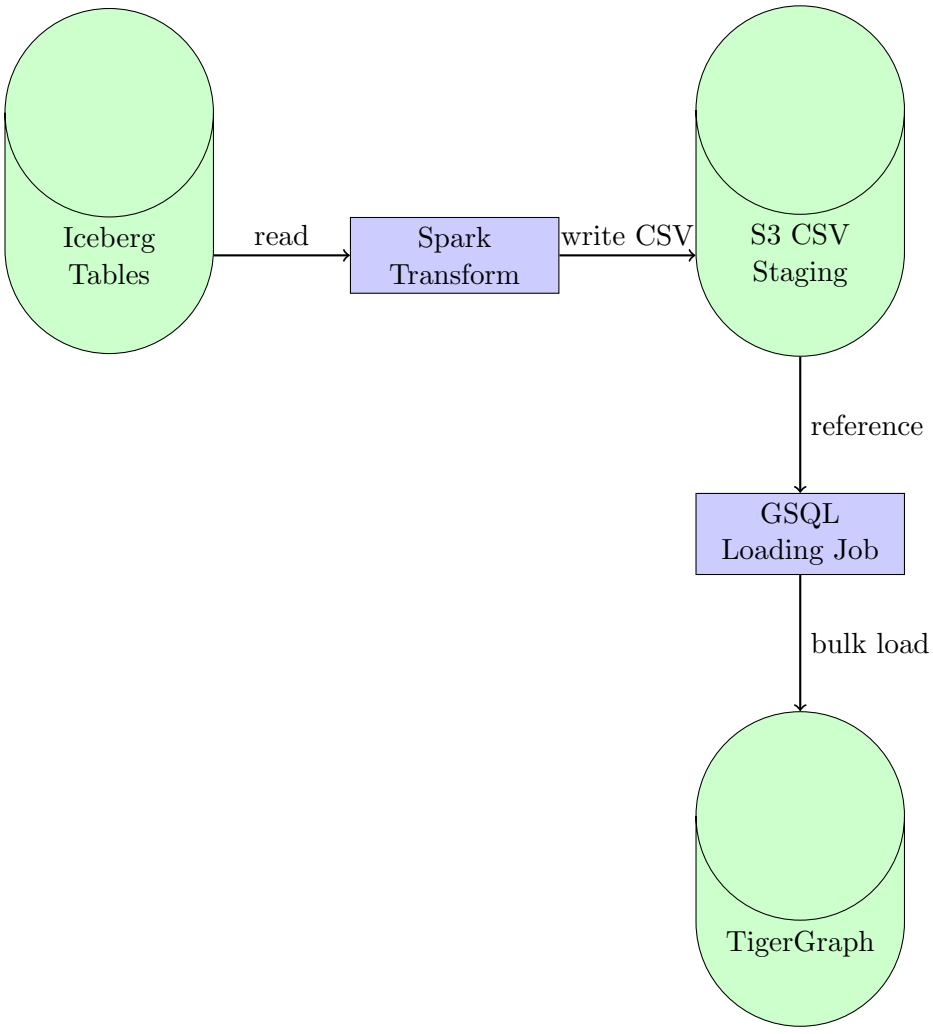

Figure 2: GSQL Loading Jobs Data Flow

Performance Comparison REST++ vs GSQL loading jobs for huge data volume

As mentioned in official documentation, GSQL loading jobs can achieve throughput of atleast a million records per second. REST++ can maximum handle 10000 records a second, which means it will take roughly 100 seconds ($\sim$ 2 minutes) to upload million records. Therefore, if I

scale this to more number of records we get

Records	REST++ Time	GSQL Time
1 Million	2 minutes	1 seconds
10 Million	20 minutes	10 seconds
100 Million	4 hours	2 minutes
1 Billion	2 days	20 minutes

Table 2: Time taken to load different data volumes

Therefore for large data volumes, GSQL loading job becomes much more resources, cost and performance efficient than compared to REST++. I recommend that we should have two separate frameworks in our data-integration pipelines.

- **Near-incremental or Low-Latency uploads**: Micro-Batch data sets from Iceberg, use Spark Partitioning and parallelization to then upload micro-batches into TigerGraph DB through REST++ upsert API calls.

- **Large Data ingestion or first time Data migration**: Use Spark to pull data from Iceberg table datasets and use partitioning and parallelization to store data into local or blob storage like S3. Then create GSQL loading jobs in our TigerGraph clusters to read different partitions and use graph-algorithms to perform parallelized data ingestion.

Hybrid Design and real-world simulation for pushing data to TigerGraph:

- GSQL Loading Jobs: Initial load, daily batch > 1M records
- REST++ API: Incremental updates, per-minute or hourly batch job < 1M records

This will allow the data-integration pipeline that uses **Apache Spark** and pushes data to TigerGraph to be flexible and extensible. Some use case for the given Ecommerce problem would be:

1. Transfer legacy ecommerce data from Iceberg into new TigerGraphDB:
For the initial migration our GSQL loading framework would be ideal. Ecommerce datasets are huge and they can be bulk uploaded using a combination of Unity Catalog, Apache Spark, S3 storage and GSQL loading jobs.

2. Continuous sync of new Ecommerce datasets into TigerGraphDB:
After intial successful data-migration to TigerGraph, we can open up an active data-integration pipeline that runs on cron basis (every minute or hour), batches new updates in Iceberg into less than 10000 records and then pushes the data in real-time into TigerGraphDB using REST++ API calls. Need to ensure idempotency of each upsert call and also enable retry logic on failures.

3 Design Alternatives and Tradeoffs

Processing Model: Batch vs Streaming

Disadvantages of Stream Processing through Kafka connect if Iceberg

- Iceberg currently doesn't efficiently support even-streams or change-data-capture in it's native implementation and APIs. This is major bottleneck to use stream-processing as our source layer which heavily relies on Iceberg.

Criterion	Batch Processing	Streaming Processing
Latency	Minutes to hours	Seconds to minutes
Complexity	Low (scheduled jobs)	High (state management)
Cost [ChatGPT estimated]	$150/month	$6,000/month (40x more)
Iceberg Support	Excellent	Limited (improving in v2.0)
Fault Tolerance	Simple (retry job)	Complex (checkpointing)

Table 3: Batch vs Streaming Comparison

- Inherently, Iceberg is snapshot-driven and although it supports high-concurrency reads and writes, but I am not sure how effective that workflow would be on huge-scales.

- Also, iceberg does not support monotonic offsetting like Kafka does and we need to manually implement offsetting till row-level which could be highly complex to implement and may lead to data loss.

- Kafka connect does not implement Databricks unity catalog which reduces many complexities like AWS permissions, tokens etc. This also needs to be manually implemented.

- Also as seen in previous sections, although TigerGraph does implement event-driven architecture through REST++ API calls, it largely becomes highly performant when I use GSQL loading jobs which it inherently support through highly parallelized graph algorithms.

My Decision: I recommend to start with batch or micro-batch processing and implement streaming only when low-latency requirements are needed (< 2 minute).

Data Flow: Push vs Pull

Push Mode (Selected)

```
# Spark reads Iceberg and pushes to TigerGraph
df = spark.read.format("iceberg").table("catalog.schema.table")

def push_partition(partition):
    tg_client = TigerGraphClient(...)
    for batch in chunks(partition, 10000):
        graph_data = transform(batch)
        tg_client.push(graph_data)

df.foreachPartition(push_partition)
```

Listing 3: Spark Push Pattern

Advantages:

- Leverages existing Databricks/Spark infrastructure
- Unity Catalog native integration

- Parallel processing via Spark executors
- Mature tooling and debugging

Pull Mode (Alternative)
Requires custom Kafka Connect Source Connector for Iceberg.

Disadvantages:

- No mature Iceberg source connector exists
- More custom code required
- Limited parallelism control
- Additional infrastructure (Kafka cluster)

My Decision: Use **Push mode** with Spark.

4 Few Implementation Details as done in my Mock Code

Micro-Batch Processing

Large datasets cannot fit in memory. Micro-batching provides:

- **Memory efficiency**: Process subset at a time
- **Error isolation**: Failed batch doesn't affect others
- **Progress visibility**: Track completion per batch
- **Restart capability**: Resume from last successful batch

Batch Size Selection

$$\text{Optimal Batch Size} \simeq \frac{\text{Available Memory}}{\text{Avg Record Size}} \tag{1}$$

Recommended sizes:

- Small records ($<$ 5KB): 10,000 - 50,000
- Medium records (5-50KB): 1,000 - 10,000
- Large records ($>$ 50KB): 100 - 1,000

Parallel Processing with Spark

Partitioning Strategy

```
# Read Iceberg table
df = spark.read.format("iceberg").table("catalog.schema.table")

# Repartition for parallelism
df = df.repartition(num_executors * cores_per_executor)

# Each partition processes independently
df.foreachPartition(lambda partition: process_partition(partition))
```

Listing 4: Spark Parallel Processing

5 Cost Analysis

5.1 Infrastructure Costs

NOTE: This has been estimated using LLMs like ChatGPT and Claude. So I am certain about the actual costs.

Component	Batch (Monthly)	Streaming (Monthly)
Databricks Cluster	$150	$5,760
S3 Storage (100GB)	$2.30	$2.30
S3 Requests	$0.50	$5.00
TigerGraph (Fixed)	$1,000	$1,000
Kafka Cluster	-	$200
Total	**$1,153**	**$6,967**

Table 4: Monthly Infrastructure Cost Comparison

From this analysis, streaming turns out to be **6x more expensive** than batch.

6 Performance Analysis

I present here rough calculations of time taken by each TigerGraph method to push1 billion records.

$$\text{Time} = \frac{\text{Total Records}}{\text{Throughput}} + \text{Overhead} \tag{2}$$

REST++ API Method
For 1 billion records at 1,0000 rec/sec:

$$\text{Time} = \frac{1,000,000,000}{1,0000} = 100,000 \text{ seconds} \approx 40 \text{ hours} \tag{3}$$

GSQL Loading Jobs Method
For 1 billion records at 1000,000 rec/sec:

$$\text{Time} = \frac{1,000,000,000}{1,000,000} = 1,000 \text{ seconds} \approx 20 \text{ minutes} \tag{4}$$

7 Decision Matrix

At last I present a decision matrix. This list down all my insights around the different approach we can take to sync data from Apache Iceberg to TigerGraphDB. I believe although most of the things could be feasible if I dig deeper but they seem too complex as of now. My decision would to go for a hybrid approach using a combination of batching for large-volume and micro-batching for near real-time requirements through a Apache Spark supported push mechanism for the data-pipeline.

Scenario	Processing	Loading	Records	Latency
Initial historical load (less complex)	Batch (Spark)	GSQL Jobs	10 million to 1 billion	Less strict (minutes to hours)
Daily batch updates (less complex)	Batch (Spark)	GSQL Jobs	1 million to 10 million	Less strict (seconds to minutes)
Hourly updates (HTTP overhead and state management)	Micro-batch (Spark/Flink)	REST++ API	10000 to 1 million	Medium-latency (seconds to minutes)
Real-time updates (highly complicated)	Streaming (Kafka Connect)	REST++ API	1000 - 10000	Low latency critical (seconds)
CDC from transactional DB (extremely complex)	Streaming (Kafka Connect)	REST++ API	1-1000	Sub-ms latencies (milliseconds)

8 Conclusion

1. **Processing Model**: Start with batch, add streaming if needed
2. **Data Flow**: Push architecture using Spark
3. **Loading Method**: Hybrid approach
   - GSQL Loading Jobs for bulk loads ($>$ 10M records)
   - REST++ API for incremental updates ($<$ 10M records)
4. **Deployment**: Kubernetes CronJob for scheduled execution

A References

1. Apache Iceberg Documentation: `https://iceberg.apache.org/docs/latest/`
2. Databricks Unity Catalog: `https://docs.databricks.com/data-governance/unity-catalog/`
3. TigerGraph Documentation: `https://docs.tigergraph.com/`
4. TigerGraph Loading Jobs: `https://docs.tigergraph.com/gsql-ref/current/ddl-and-loading/`
5. Apache Spark Iceberg Integration: `https://iceberg.apache.org/docs/latest/spark-getting-start`