Iceberg to TigerGraph Data Integration Pipeline

Design Proposal and Architecture Analysis

Rahul Khanna

January 9, 2026

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 Catalogue, pulls the table metadata from Iceberg snapshots and fetches data from S3 buckets. Internally, it maps the Iceberg data into TigerGraph compatible data format through a Mapping Engine. Furthermore, I use the REST++ APIs (mock) provided by the TigerGraph-hosted clients and perform batch or micro-batch operations to push the data into the graph database. In the end, I discuss in detail different alternatives we can take for this data integration pipeline, and evaluate their use-cases, fault tolerance, 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 unstructured data stored in data lakes into highly-structured graph data format for advanced analytics such as social media relations, 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 Data lakes such as Apache Iceberg tables (ACID-compliant table format)
- Leverage Unity Catalog for metadata sync, management and governance
- Transform relational data into graph structures
- Load efficiently into TigerGraph according to scales
- Support both bulk loads and near-real-time 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 or Micro-Batch processing with Apache Spark** (allows parallel data processing)
2. **Two loading mechanisms**:
   - REST++ API for near-real-time incremental updates
   - GSQL loading jobs for bulk loads (backfill millions of records or initial bulk loading of Iceberg data snapshots)
3. **Push architecture** (Spark reads from Iceberg and pushes to TigerGraph using REST++ API calls or loading jobs)
4. **Configuration-driven schema mapping** (allowing flexibility to add new schema changes if Iceberg snapshot is changed)

1.3 My Design Choices

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

Table 1: Design Decision

1.4 Performance Benefits

- **Performance**: Upload millions of records in a few hours (GSQL loading jobs)
- **Micro-Batching**: Achieve near real-time sync of records (REST++ upsert operations) with low latencies
- **Flexibility**: Easy addition of new tables via configuration
- **Scalability**: Horizontal scaling via Spark partitioning and parallelization while both reading and writing.

2 System Architecture

2.1 High-Level Architecture

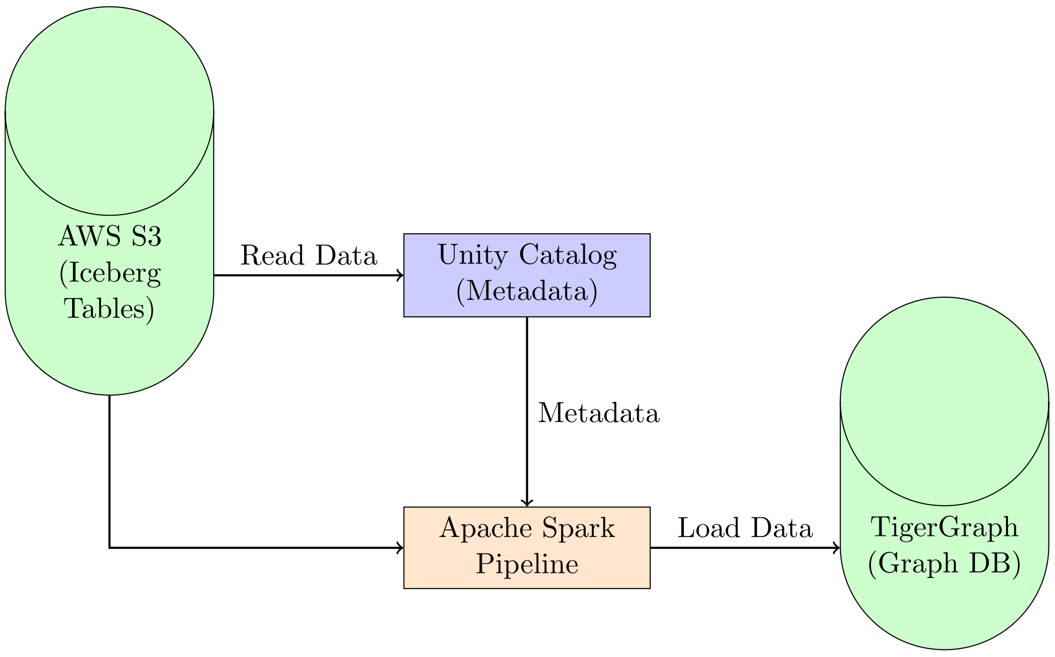

Figure 1: High-Level System Architecture

On a high level, the data pipeline Django project has an Apache Spark pipeline app. This service would integrate with Apache Iceberg through Databricks Unity Catalog to constantly sync Iceberg metadata (snapshots, file locations, authentication etc.). On any new data uploads to Iceberg, the pipeline fetches the data sets from the exact file locations given by Unity Catalog. I transform the datasets to be compatible with the TigerGraph graph-based data structure, and then push or upsert the new datasets as vertices or edges into the TigerGraph DB.

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 should go with Apache Spark as the middleware because of several performance and cost benefits.

First of all, Iceberg has recently been upgraded for **streaming or event-driven workflows**. For a long duration, its architecture is ideal for batch processing. Within its core, on updates and additions, Iceberg has to create a new table snapshot and copy all the data files (Copy on Write or Merge on Read), which used to prove to be highly inefficient for highly-frequent event-driven systems.

Although now streaming is supported by Apache Iceberg, large native development already exists for batch or bulk processing and integrating a streaming connector like Apache Kafka Connect, Flink or Spark streaming may need additional computing complexity and costs. Therefore, if not strictly required to be real-time, batch and micro-batch processing provided natively by both Apache Spark and Apache Iceberg will be ideal.

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 and parallelization support
- Fault tolerance through RDDs, lineage tracking 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 and joins 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 through advanced graph algorithms, and can sync millions of records per second. It has been widely used across different platforms for enabling graph-based analytics and real-time insights.

2.3 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 (< 10000).
- Can work well for event-driven or real-time uploads.
- Can also work for mini-batching, achieving close event-driven integration.
- Needs a lot of HTTP overhead for each upsert call.
- Can be batched and performed in parallel for different datasets.

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 such as notification systems, real-time messaging, social media etc. These **upsert** API calls can push incremental updates coming from event-driven data-pipelines example, Change-Data-Capture from Debezium. To reduce HTTP overheads, we can also micro-batch or batch upload datasets into a single REST++ upsert API call.

Method 2: GSQL Loading Jobs

```
USE GRAPH EcommerceGraph

CREATE LOADING JOB load_users FOR GRAPH EcommerceGraph {
    DEFINE FILENAME user_file;
    LOAD user_file TO VERTEX User VALUES ($1, $2, $3) USING SEPARATOR = ",";
}
RUN LOADING JOB load_users USING user_file = 's3://bucket/users.csv'
```
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.

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

As mentioned in official documentation, GSQL loading jobs can achieve a throughput of at least a million records per second. REST++ can handle 10000 records a second, which means it will take roughly 100 seconds ($\sim$ 2 minutes) to upload a million records if not considering **parallelization**.

In Table 2, we see that REST++ API calls will take significantly more time to upload 10

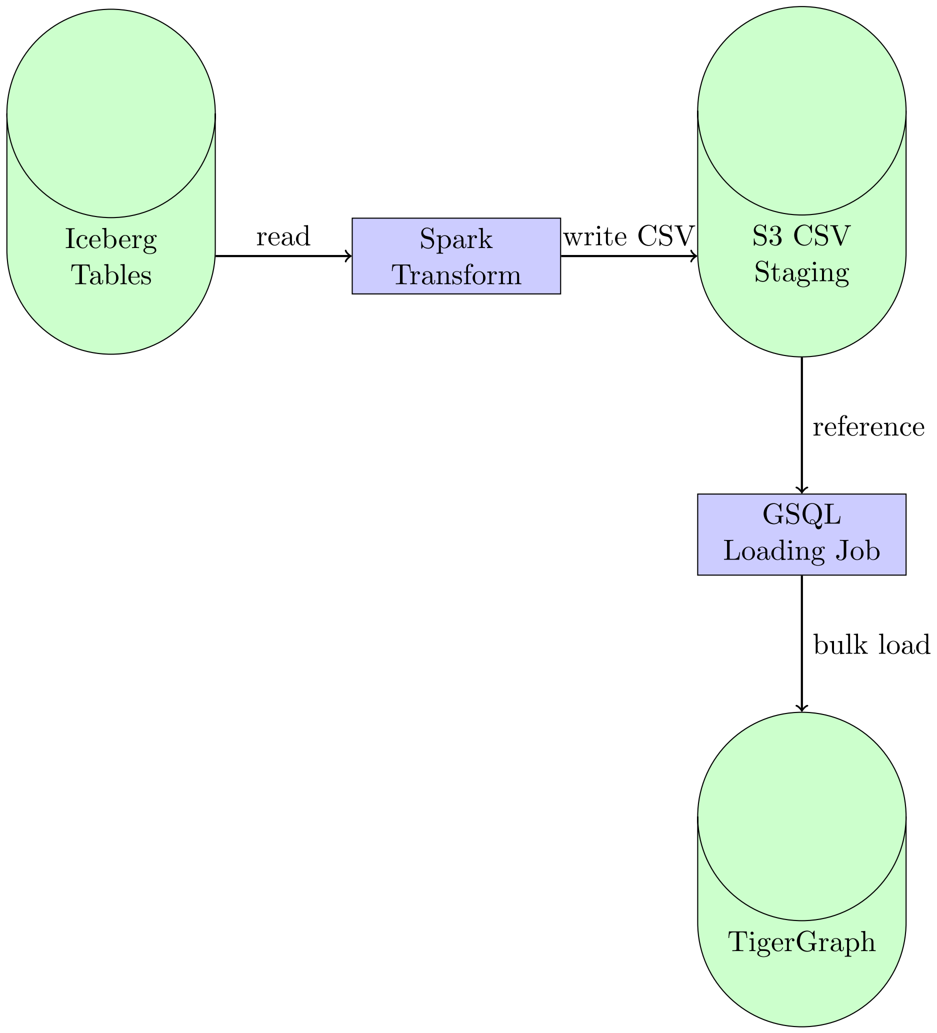

Figure 2: GSQL Loading Jobs Data Flow

million or more records than GSQL loading jobs. We can simply create threads to parallelize that many batches of 10000 records, as it will be a highly delicate integration prone to constant failures and retries, reducing fault-tolerance and consistency upon failures. Also, the Tiger-Graph client can become a bottleneck if it is not able to process that many API calls.

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 S3 or local file partitions and use GSQL loading jobs to perform highly-performant data ingestion.

2.5 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 enable the data-integration pipeline, which utilizes **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. E-commerce datasets are huge (Petabyte-scale), and they can be bulk uploaded using a combination of Unity Catalogue, Apache Spark, S3 storage, and GSQL loading jobs.

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

3 Design Alternatives and Tradeoffs

3.1 *Processing Model: Batch vs Streaming*

Disadvantages of Stream Processing through Kafka Connect when connecting with Iceberg

- **Iceberg is Snapshot Based**: Iceberg mainly appends or rewrites files and creates new snapshots based on updates, and does not track row-level changes. Iceberg recently upgraded for streaming or event-driven workflows in its implementation and has suggested in their official documentation that it needs additional complexity to maintain stream writes, and continuous writes can lead to a lot of table metadata because of new snapshots which is extremely hard to manage and maintain [Apache Iceberg maintaining Stream writes].

- **Kafka Connect is Row-level Based** Kafka is a technology specifically made to track continuous changes at smallest-scales like row updates, creates or deletes in relational DB and low-latency sync. This is quite opposite in architecture to Iceberg, which is built to deal with changes at the file-level and does not track row-level changes natively.

Criterion	Batch Processing	Streaming Processing
Latency	seconds to hours	milliseconds to seconds
Complexity	Low (scheduled jobs)	High (state management)
Cost [ChatGPT estimated]	low-moderate	high (around 40x more)
Iceberg Support	Excellent	Limited (improving in v2.0)
Fault Tolerance	Simple (retry job)	Complex (checkpointing)

Table 3: Batch vs Streaming Comparison

- **No offsetting mechanism like Kafka**: Iceberg does not support monotonic offsetting as Kafka does, and we need to manually implement offsetting till row-level for file-level changes, which could be highly complex to implement and may lead to data loss.
- **No Data Unity support by Kafka connect** Kafka connect does not implement Databricks Unity catalog which reduces many complexities like AWS permissions, tokens, etc. This also needs to be manually implemented.
- **TigerGraph bottleneck** - As seen in previous sections, although TigerGraph does implement a near-real event-driven architecture through REST++ API calls, it is not made for a continuous flow of streaming datasets. I observed that there are significant cost and resource benefits when GSQL loading jobs are used. There are many overheads involved in REST++ API calls, including HTTP configurations, permissions, retry mechanisms, failure management, and idempotency challenges on large-scales. To support continuous data flow, we need to keep several TigerGraph clients actively running and consuming new Kafka messages.

My Decision: I recommend to start with batch or micro-batch processing and implementing streaming (through Spark streaming potentially) only when low-latency requirements are the main business needs (< 2 minute).

3.2 *Data Flow: Push vs Pull*

Push Mode (Selected)

- **TigerGraph is PUSH-based**: TigerGraph is natively built to handle push loads. REST++ upsert API calls and GSQL loading jobs are inherently mechanisms to push data to TigerGraph.
- **Parallel processing via Spark executors**: Pushing to TigerGraph DB clusters would be extremely efficient because of the partitioning and parallelization capabilities provided by Apache Spark inherently.
- **Inherent Fault Tolerance** : Spark supports automatic retry mechanisms for failed jobs and ensures the successful reexecution of a particular sync job. This gives confidence that the Spark jobs will be able to sync near-real-time events eventually.

Criterion	**Push Mode (Spark)**	**Pull Mode (Kafka Connector)**
Data Model	snapshot-based	event-based
Iceberg Fit	native support	no native support
Scalability	horizontally scalable	limited scalability
Operational Complexity	moderate	high
Fault Tolerance	built-in retries	custom retry logic
TigerGraph Fit	bulk loading	small writes
Latency	medium	low (theoretical)
Cost [ChatGTP Estimated]	cost-efficient	expensive at scale

Table 4: Push vs Pull Ingestion Model Comparison

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

Disadvantages:

- **TigerGraph is not PULL-based**: TigerGraph loading methods are push-based, and it will be hard to implement pull-based mechanisms.
- **Complex Code**: For frequent updates, the Kafka connector needs to calculate the difference between the last two snapshots and send updates based on findings. This is highly complex and may increase computational and development complexity
- **Complicated failure recoveries** Although Kafka Connect is highly parallel, we need to ensure manual offsetting to handle failed message sync, which needs a separate orchestrator service like Zookeeper to keep track of each consumer and their offsets.

My Decision: I recommend going with PUSH based mechanism, which is inherently supported by both Iceberg and TigerGraph, and Apache Spark fits perfectly in between, allowing fault-tolerant, reliable and scalable data ingestion.

4 Architectural Benefits

This section discusses the architectural benefits of the proposed push-based data pipeline built using Apache Spark, Apache Iceberg, and TigerGraph, orchestrated through a Django-Spark-based control layer. The analysis focuses on fault tolerance, reliability, scalability, and validation, which are critical for large-scale data ingestion systems.

4.1 Fault Tolerance

Fault tolerance is a primary design goal of the pipeline due to the large data volumes and distributed execution environment.

Current Guarantees through Django Project

- Apache Spark provides task-level fault tolerance through automatic retries and recomputation of failed tasks.
- Iceberg's snapshot-based design ensures consistent reads even in the presence of partial failures.
- Failures during transformation do not corrupt source data, as Iceberg snapshots are immutable.
- TigerGraph bulk ingestion jobs can be retried safely due to unique vertex and edge primary keys.

Important Future Architectural Considerations:

- All Spark jobs must be idempotent, producing identical outputs when re-executed on the same snapshot.
- Snapshot identifiers or ingestion timestamps should be persisted externally to track successful ingestion checkpoints.
- REST++ ingestion should be limited to micro-batches to avoid partial ingestion failures.

Required Upgrades for Improved Fault Tolerance:

- Persist the last successfully processed Iceberg snapshot ID in a metadata store.
- Introduce structured retry policies with exponential backoff.

4.2 Reliability

Reliability refers to the system's ability to consistently deliver correct and complete data to TigerGraph.

Current Guarantees in Design Decision:

- Snapshot isolation ensures that each Spark job processes a consistent view of data.
- Primary keys prevent duplicate vertices and edges during retries.
- Push-based ingestion avoids race conditions common in pull-based architectures, especially in Kafka-based workflows.

Important Future Architectural Considerations:

- Strong schema validation must be enforced at the transformation layer.
- All ingestion operations should be designed to be restartable at the Django-Spark layer. If needed, S3 checkpointing can be integrated.

Required Upgrades for Improved Reliability:

- Introduce schema compatibility checks during mapping.
- Implement alerting for partial or failed ingestion runs.

4.3 Scalability

The pipeline is designed to scale horizontally to handle billions of records.

Current Guarantees:

- Spark distributes computation across multiple executors.
- Iceberg enables partition pruning and parallel file reads.
- TigerGraph bulk loading jobs ingest data in parallel.

Architectural Considerations:

- REST++ ingestion should be used only for small micro-batches.
- Bulk ingestion through GSQL loading job must be preferred for large datasets.
- Data should be partitioned by vertex or composite edge keys before loading.

Required Upgrades for Improved Scalability:

- Increase Spark executor count and memory allocation.
- Use object storage (e.g., S3) as an intermediate staging layer for graph data.

4.4 Validation and Data Quality

Validation ensures correctness, completeness, and trustworthiness of ingested data.

Current Guarantees:

- Snapshot-based reads prevent partial data ingestion.
- Strict mapping ensures repeatable results.
- Django API endpoints allow post-ingestion verification.

Future Architectural Considerations:

- Validation must occur both before and after ingestion.
- Metrics should be collected for each ingestion stage.
- Schema evolution must be explicitly handled.

Required Upgrades for Improved Validation:

- Add data-count reconciliation between Iceberg and TigerGraph.

My Decision: We should go with **Push mode** supported by Apache Spark.

5 Key Benefits from the Spark-based data ingestion

5.1 *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

5.2 *Parallel Processing with Spark*

Partitioning Strategy

```python
# 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 3: Spark Parallel Processing

Data Parallelism
Spark splits large datasets into many independent partitions, which are processed simultaneously.
Data Pipeline Project: Huge Iceberg table stored as 10,000 Parquet files can be read by hundreds of Spark tasks in parallel, reducing read time from hours to minutes.

Task-Level Parallelism
Each transformation is broken into multiple tasks that run concurrently across the Spark cluster which provides faster execution and isolated failure recovery.
Data Pipeline Project: Mapping user records to graph vertices is executed independently for each partition, allowing thousands of records to be transformed at the same time.

Executor-Based Parallelism
Spark assigns multiple tasks to long-running executors, each utilizing multiple CPU cores providing high-throughput and concurrency.

Partition-Aware Parallelism
Data can be explicitly partitioned by keys such as user ID or product ID to reduce skew.
Data Pipeline Project: Edges partitioned by source vertex ensure even load during graph ingestion.

Fault-Tolerant Parallelism
Failed tasks are retried independently without restarting the entire job ensuring high reliability.
Example: If one executor fails while processing 1% of the data, only that portion is recomputed.

6 Cost Analysis

6.1 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 5: Monthly Infrastructure Cost Comparison

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

7 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 (assuming no parallelization)
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:

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

8 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 (Spark streaming or 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)

9 Conclusion

1. **Processing Model**: Batch + Micro-Batch processing/
2. **Data Flow**: Push architecture using Spark.
3. **Loading Method**: Hybrid approach
   - GSQL Loading Jobs for bulk loads (> 1M records).
   - REST++ API for incremental updates (< 1M records).
4. **Deployment**: CronJob for regular execution of sync job.

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`