# Solution Proposal Document

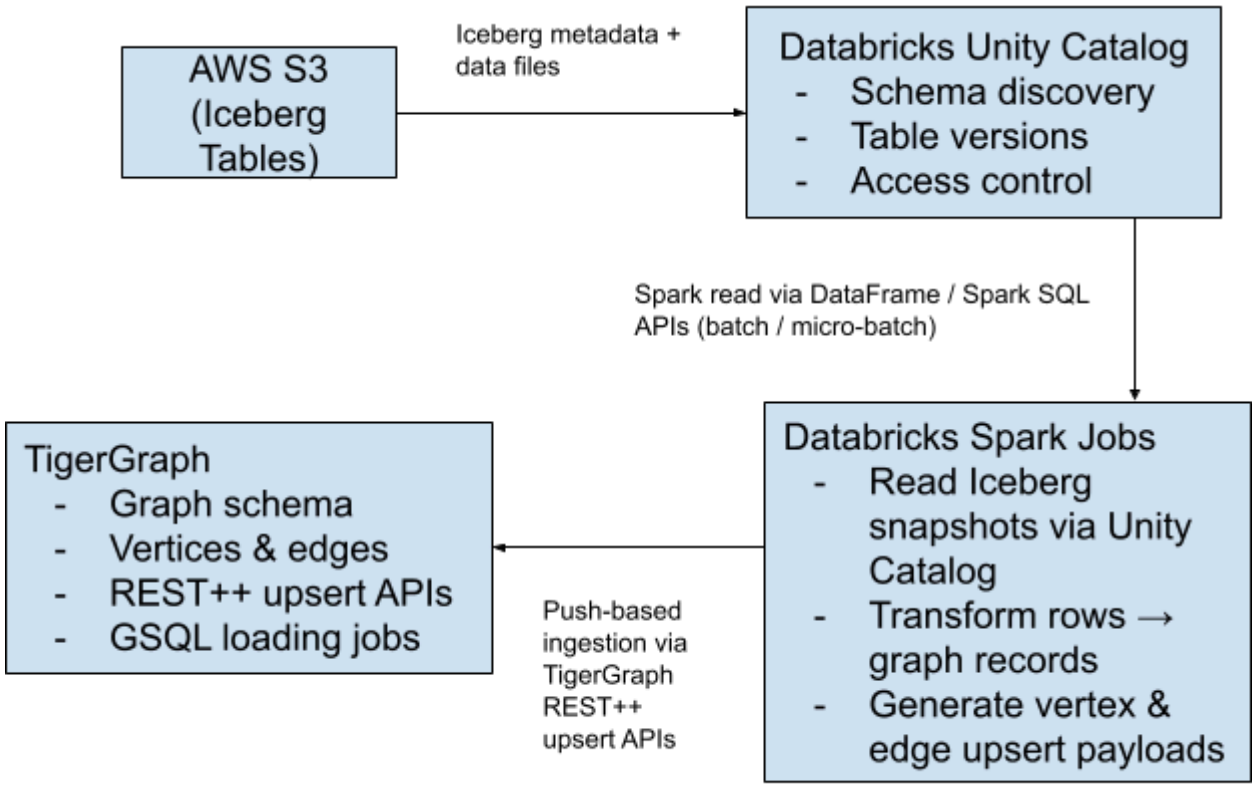
## 1. Overall architecture

### Architecture Goals

The architecture is designed to:

1. Reliably read **Apache Iceberg** tables stored in S3
2. Use **Databricks Unity Catalog** for schema discovery and access control
3. Transform relational data into **graph structures** (vertices & edges)
4. Load data efficiently into **TigerGraph**
5. Support future evolution from batch to near-real-time ingestion
6. Demonstrate Spark reading Iceberg tables via Unity Catalog and performing actual upserts into TigerGraph using native APIs

### High-Level Architecture Diagram




### Component-by-Component Breakdown

1. **Apache Iceberg on S3 (Source)**

**Role**

- Stores large-scale, immutable, versioned tabular datasets
- Provides snapshot-based consistency

**Key Characteristics**

- Append-heavy, snapshot-based
- Supports schema evolution
- Well-suited for batch and micro-batch processing

**Iceberg requirement**

- Allows reproducible graph rebuilds from snapshots
- Enables incremental reads (via snapshot comparison)

## 2. Databricks Unity Catalog (Metadata & Governance Layer)

**Role**

- Central catalog for:
  - Table schemas
  - Versions
  - Permissions
- Single control plane for data access

**Responsibilities**

- Exposes Iceberg metadata to Spark
- Enforces fine-grained access control
- Provides auditability
- Enforces access and schema resolution during Spark SQL and DataFrame reads

**Reasons for Databricks sit in the middle**

- TigerGraph should **not** directly access Iceberg
- Spark jobs inherit permissions automatically
- Decouples governance from ingestion logic

## 3. Databricks Spark (Transformation Layer)

**Role:** This is the **core integration engine**.

Spark is responsible for:

- Reading Iceberg tables via Unity Catalog using Spark SQL and DataFrame APIs
- Selecting the correct snapshot or time window
- Transforming relational rows into graph entities
- Preparing TigerGraph-compatible payloads for REST++ upsert APIs and batch loading jobs

**Key Tasks**

- Entity extraction → vertices
- Relationship extraction → edges
- Data cleaning/enrichment (optional)
- Deduplication and key normalization

**Why Spark**

- Native Iceberg support
- First-class Unity Catalog integration
- Scales easily
- Familiar operational model

### 4. TigerGraph (Destination)

**Role:** Native graph database optimized for traversal and analytics

**Responsibilities**

- Stores graph schema
- Hosts vertices and edges
- Executes graph queries and algorithms

**Ingestion Mechanisms**

- **GSQL Loading Jobs** (batch)
- **REST++ APIs** (optional for incremental)

**Why TigerGraph is not pulling**

- Iceberg metadata is complex
- Governance belongs in Unity Catalog
- Push keeps TigerGraph stateless and simpler

# Data Flow (End-to-End)

1. Spark job starts on Databricks
2. Spark reads Iceberg tables via Unity Catalog using catalog-qualified table identifiers and Spark DataFrame APIs
3. Schema is inferred dynamically
4. Rows are transformed:
    - Entities → vertices
    - Foreign-key relationships → edges
5. Transformed data is pushed to TigerGraph via REST++ upsert APIs for vertices and edges
6. TigerGraph loads data into graph storage

---

# 2. Choice of batch vs streaming

## Overview

The integration pipeline between Apache Iceberg and TigerGraph can be implemented using either **batch-based ingestion** or **streaming ingestion**. While both approaches are viable, this design **intentionally chooses batch (with optional micro-batching)** as the primary ingestion mode.

## Batch Ingestion (Chosen Approach)

### How Batch Works in This Architecture

1. Spark periodically reads Iceberg tables via Unity Catalog using Spark SQL / DataFrame APIs
2. A consistent snapshot (or snapshot range) is selected
3. Data is transformed into graph vertices and edges
4. Data is ingested into TigerGraph via batch-oriented REST++ upsert APIs

Batch jobs may run:

- Hourly
- Daily
- Or on-demand (e.g., backfill, rebuild)

## Why Batch Fits Iceberg Naturally

Apache Iceberg is designed around **immutable snapshots**.

Key properties:

- Each snapshot represents a consistent point-in-time view
- Snapshots are ideal for repeatable batch reads
- Snapshot metadata enables incremental batch processing

Reason for the best fit in this scenario:

- Graph rebuilds
- Periodic synchronization
- Large-scale bulk ingestion

## Why Batch Fits TigerGraph Well

TigerGraph ingestion is optimized for:

- REST++ upsert APIs for batch-oriented ingestion
- GSQL loading jobs as a bulk-loading alternative

Batch ingestion allows:

- Efficient creation and update of vertices and edges
- Reduced per-record overhead
- Easier recovery on failure (rerun job)

---

# 3. Choice of push vs pull

## Overview

Data ingestion from Iceberg into TigerGraph can be implemented using either a **push-based** or **pull-based** integration model.

- **Push**: An external processing engine (e.g., Spark) reads Iceberg and pushes data into TigerGraph
- **Pull**: TigerGraph (or a custom connector) pulls data directly from Iceberg

This design **intentionally chooses a push-based model** as the primary ingestion strategy.

## Push-Based Ingestion (Chosen Approach)

### How Push Works in This Architecture

1. Spark jobs running on Databricks read Iceberg tables via Unity Catalog using Spark SQL and DataFrame APIs
2. Spark transforms tabular data into graph primitives
3. Spark pushes transformed data into TigerGraph using REST++ upsert APIs

TigerGraph remains focused on **graph storage and query execution**, not data extraction.

### Why Push Is the Right Default

#### 1. Native Integration with Unity Catalog

Unity Catalog is deeply integrated with Spark:

- Schema discovery
- Permission enforcement
- Auditing

A push model allows:

- Spark to inherit permissions automatically
- Centralized governance enforcement
- Zero custom security logic in TigerGraph

A pull model would require re-implementing:

- Catalog access
- Credential management
- Policy enforcement

#### 2. Clear Separation of Concerns

| Component | Responsibility |
| --- | --- |
| Spark | Read, transform, validate |
| Unity Catalog | Governance & metadata |
| TigerGraph | Store and query graphs (via REST++ upserts from Spark) |

Push-based ingestion keeps:

- ETL logic outside TigerGraph
- TigerGraph is lean and focused
- The architecture modular

**3. Operational Simplicity**

Push-based pipelines:

- They are easier to monitor
- Fail fast and visibly
- Can be retried safely
- Integrate with existing Spark tooling

Retries are straightforward:

- Re-run job
- Reload the same snapshot
- Idempotent graph upserts

**4. Flexibility in Data Transformation**

Graph modeling often requires:

- Joins
- Aggregations
- Deduplication
- Enrichment

Spark excels at these operations.

A pull-based approach would:

- Either reimplement the transformation logic
- Or severely limit graph modeling complexity

---

# 4. Data flows from Iceberg → Unity Catalog → TigerGraph

## Overview

This section describes the **end-to-end data flow** for ingesting tabular data stored in Apache Iceberg into TigerGraph, with **Databricks Unity Catalog** acting as the metadata and governance control plane.

The pipeline follows a **push-based, batch-oriented flow**, ensuring strong consistency, governance, and operational simplicity.

## Step 1: Data Storage in Iceberg (Source Layer)

Source data is stored as **Apache Iceberg tables** backed by files in AWS S3.

Key characteristics:

- Data is organized into immutable snapshots
- Each snapshot represents a consistent point-in-time view
- Schema evolution is tracked at the table level

Iceberg tables may include:

- Entity tables (e.g., users, products)
- Relationship tables (e.g., transactions, interactions)

Iceberg itself does not enforce access control directly; it relies on an external catalog.

## Step 2: Metadata & Access via Unity Catalog (Governance Layer)

**Unity Catalog** serves as the **single source of truth** for:

- Table schemas
- Table versions
- Access permissions
- Auditing

When Spark jobs run on Databricks:

- Table access is validated against Unity Catalog
- Column-level and table-level permissions are enforced
- The schema is discovered dynamically at runtime

This ensures:

- Secure access to Iceberg data
- No hardcoded schemas in the pipeline

- Governance policies are consistently applied

# Step 3: Data Read via Spark (Extraction Layer)

Databricks Spark jobs read Iceberg tables using Unity Catalog–managed identifiers via Spark SQL and DataFrame APIs.

At this stage:

- A specific snapshot or snapshot range is selected
- Only authorized tables and columns are accessible
- Spark loads data into DataFrames with inferred schema

Example actions:

- Full snapshot read (initial backfill)
- Incremental snapshot read (new or changed data)

This step establishes a **consistent and reproducible dataset** for graph ingestion.

# Step 4: Relational-to-Graph Transformation (Processing Layer)

Spark transforms tabular data into **graph primitives**:

## Vertex Extraction

- Each entity row becomes a vertex
- Primary keys map to vertex IDs
- Non-key columns become vertex attributes

## Edge Extraction

- Foreign-key relationships become edges
- Source and destination IDs map to vertex references
- Transactional or relational fields become edge attributes

This transformation may involve:

- Joins across tables
- Deduplication
- Attribute normalization
- Optional enrichment

All graph modeling logic is centralized in Spark.

# Step 5: Graph-Ready Payload Generation (Integration Layer)

After transformation, Spark produces TigerGraph-compatible JSON payloads for REST++ upsert APIs.

Payloads are structured to align with:

- Predefined TigerGraph vertex types
- Predefined TigerGraph edge types
- Required primary IDs and attributes

# Step 6: Data Ingestion into TigerGraph (Destination Layer)

The transformed data is pushed into TigerGraph using REST++ upsert APIs.

Vertices and edges are inserted or updated idempotently through authenticated REST calls from Spark.

GSQL loading jobs are supported by TigerGraph for large-scale bulk ingestion, but are not used in the prototype implementation.

TigerGraph is responsible only for:

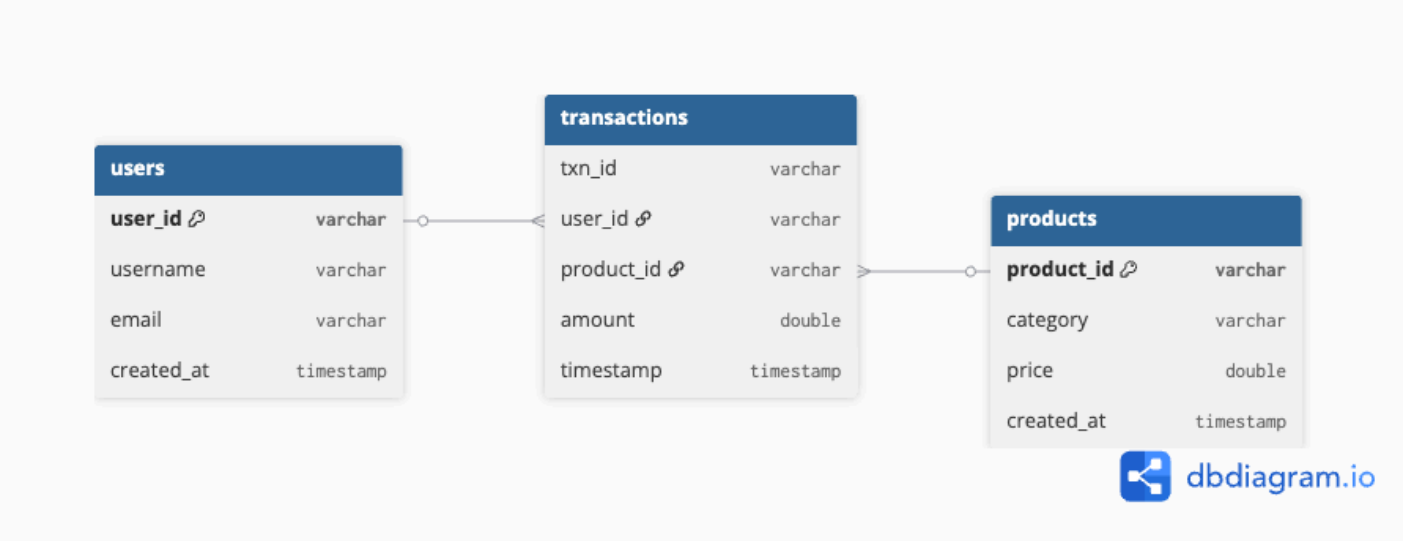
- Storing vertices and edges
- Maintaining graph indexes
- Serving graph queries

**In the event of partial ingestion or transient failures, Spark can safely retry REST++ upsert requests without creating duplicate graph entities.**

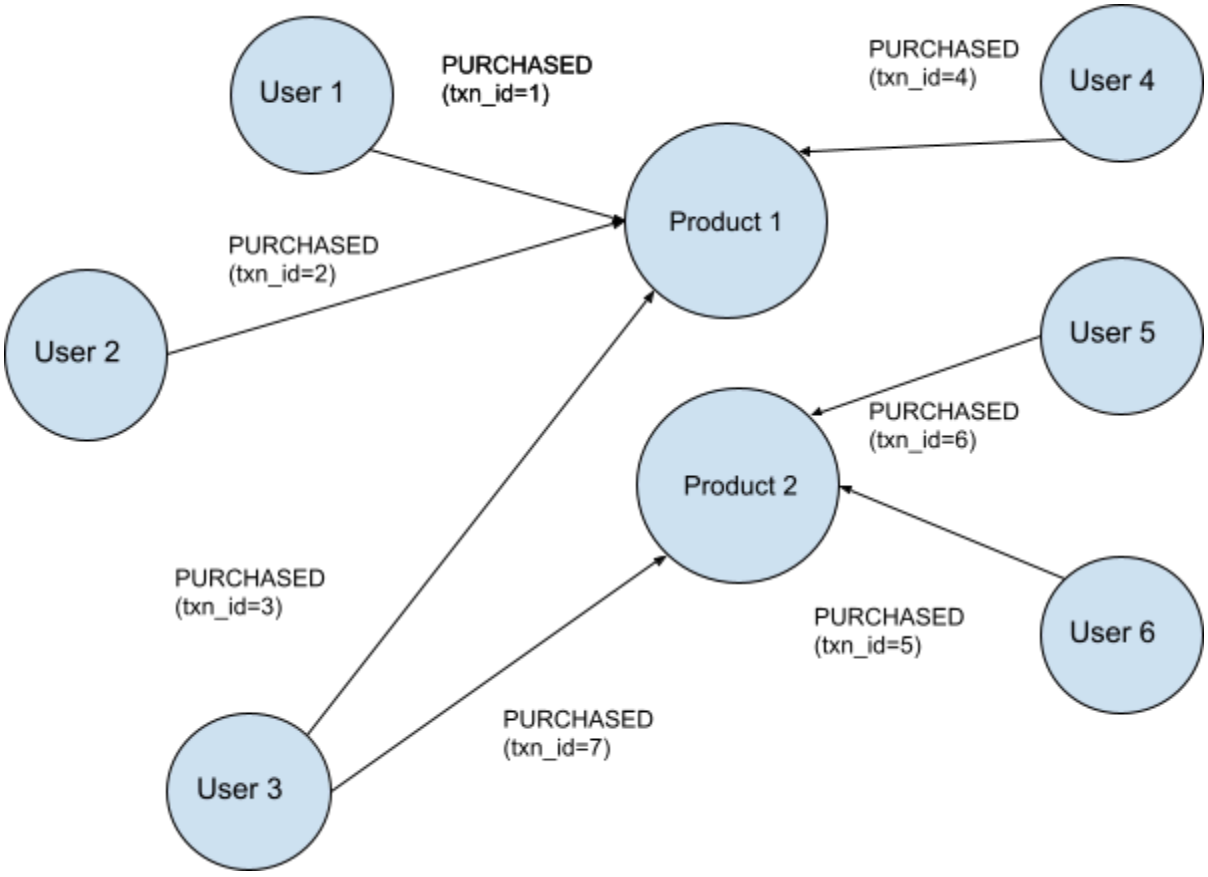
---

# 5. Schema mapping example (e.g., users → vertex, transactions → edge)

### Tabular Data Visualization




### TigerGraph Graph Visualization



Vertices represent Users and Products. Each directed edge represents a PURCHASED transaction with transaction-specific attributes stored on the edge.

## Sample Input: Iceberg Row (Source)

Example 1: `users` Table Row (Iceberg)


```
{
  "user_id": "u123",
  "name": "Alice Johnson",
  "email": "alice@example.com",
  "created_at": "2024-10-15T08:42:00Z"
}
```


This represents a single record stored in an Iceberg table on S3 and governed by Unity Catalog.

Example 2: `products` Table Row (Iceberg)


```
{
  "product_id": "p456",
  "category": "Electronics",
  "price": 99.99,
  "created_at": "2024-09-01T12:00:00Z"
}
```


Example 3: `transactions` Table Row (Iceberg)


```
{
  "txn_id": "t789",
  "user_id": "u123",
  "product_id": "p456",
  "amount": 99.99,
  "timestamp": "2024-11-01T10:30:00Z"
}
```


## Sample Output: TigerGraph Input (Destination)

User Vertex (TigerGraph)


```
{
  "vertex_type": "User",
  "primary_id": "u123",
  "attributes": {
    "name": "Alice Johnson",
    "email": "alice@example.com",
    "created_at": "2024-10-15T08:42:00Z"
  }
}
```


Product Vertex (TigerGraph)


```
{
  "vertex_type": "Product",
```

```
  "primary_id": "p456",
  "attributes": {
    "category": "Electronics",
    "price": 99.99,
    "created_at": "2024-09-01T12:00:00Z"
  }
}
```


TRANSACTION Edge (TigerGraph)


```
{
  "edge_type": "TRANSACTION",
  "from_type": "User",
  "from_id": "u123",
  "to_type": "Product",
  "to_id": "p456",
  "attributes": {
    "txn_id": "t789",
    "amount": 99.99,
    "timestamp": "2024-11-01T10:30:00Z"
  }
}
```


**Each Iceberg row is transformed into a TigerGraph vertex or edge record and is upserted into TigerGraph using the REST++ upsert APIs in the prototype implementation.**
**Vertex upserts are idempotent based on primary IDs, while edge upserts preserve transactional semantics by creating or updating relationship records.**

---

# 6. Tradeoff summary: when each approach is better

This section summarizes the key architectural tradeoffs considered in designing the Iceberg → TigerGraph integration and highlights when each approach is most appropriate.

## Batch Ingestion

**Best suited when:**

- Data is organized as snapshots or append-heavy tables
- Graph updates do not require sub-second latency
- Operational simplicity and reliability are priorities
- Periodic backfills or full graph rebuilds are required
- Cost predictability is important

**Why Batch Ingestion works well here:**

- Apache Iceberg is snapshot-oriented by design
- TigerGraph supports high-throughput batch-oriented ingestion, including REST++ upserts and GSQL loading jobs.
- Batch jobs are easier to monitor, debug, and rerun

## Push-Based Ingestion

**Best suited when:**

- Source systems support scalable data processing engines
- Strong governance and access control are required
- Complex transformations or joins are needed
- Clear separation of responsibilities is desired

**Why Push Ingestion works well here:**

- Spark integrates natively with Unity Catalog
- Transformation logic is centralized and reusable
- TigerGraph remains focused on graph storage and querying

For the given scenario Iceberg tables on S3 governed by Unity Catalog and ingested into TigerGraph, the batch, push-based approach provides the best balance of scalability, simplicity, governance, and extensibility, and is validated by the prototype implementation using Spark and REST++ upserts.

## 1. Batch vs Streaming Decision Matrix

| Condition | Batch Ingestion | Streaming Ingestion |
|---|---|---|
| Data latency tolerance | Minutes to hours | Seconds to near-real-time |

| | | |
|---|---|---|
| Data volume | Large, bulk datasets | Continuous event streams |
| Source system | Snapshot-based (Iceberg) | Event-driven sources |
| Operational complexity | Low | High |
| Cost predictability | High | Lower (always-on compute) |
| Failure recovery | Simple (rerun job) | Complex (stateful recovery) |
| Schema evolution | Easy to manage | More complex |
| Graph update pattern | Periodic rebuilds | Continuous incremental updates |
| Recommended for | Analytics, backfills, periodic sync | Fraud detection, live recommendations |

**Batch ingestion** is the default choice due to Iceberg's snapshot semantics and TigerGraph's optimized bulk loading. While streaming ingestion can support lower-latency updates, it is intentionally out of scope for the current prototype, which focuses on batch execution using Iceberg snapshots and Spark.

## 2. Push vs Pull Decision Matrix

| **Condition** | **Push-Based Ingestion** | **Pull-Based Ingestion** |
|---|---|---|
| Control plane | External (Spark) | Destination-centric |
| Unity Catalog integration | Native & strong | Limited / custom |
| Transformation complexity | High flexibility | Limited |
| Security & governance | Centralized | Duplicated logic |
| Development effort | Low | High |
| Operational burden | Lower | Higher |
| Failure handling | Simple retries | Custom recovery |
| TigerGraph responsibility | Storage & querying (via REST++ upserts from Spark) | Ingestion + processing |
| Recommended for | Lakehouse-centric architectures | Restricted or edge environments |

**Push-based ingestion** is preferred to leverage Spark + Unity Catalog and keep TigerGraph focused on graph analytics. The push-based model is concretely realized in the prototype, where Spark reads Iceberg data via Unity Catalog and performs authenticated REST++ upserts into TigerGraph.

---

# 7. Fault Tolerance & Reliability

The proposed Iceberg → TigerGraph integration is designed to be fault-tolerant across ingestion, transformation, and loading stages. The system leverages the guarantees provided by Iceberg, Spark, and TigerGraph to ensure correctness, recoverability, and idempotent execution.

Fault Tolerance at Each Layer

1. **Iceberg (Source Layer):** Apache Iceberg provides snapshot-based fault tolerance by design. Each ingestion run operates on a well-defined snapshot or snapshot range, ensuring consistent reads even in the presence of concurrent writes. If a Spark job fails, the same snapshot can be reprocessed safely without data loss or duplication.
2. **Unity Catalog (Governance Layer):** Unity Catalog enforces schema and access validation at query time. Any unauthorized access or incompatible schema change causes Spark jobs to fail early, preventing partial or inconsistent ingestion. This fail-fast behavior improves overall pipeline reliability.
3. **Spark (Transformation Layer):** Spark provides fault tolerance through task-level retries and stage recomputation. In the event of executor failure, Spark automatically re-executes failed tasks without restarting the entire job. Transformations are deterministic and based on Iceberg snapshots, ensuring repeatable results across retries.
4. **TigerGraph (Destination Layer):** TigerGraph ingestion via REST++ upsert APIs is idempotent with respect to vertex primary IDs and edge identifiers. Replaying the same upsert requests does not create duplicate vertices or inconsistent relationships, enabling safe retries after partial failures.

| **Failure Scenario** | **Recovery Strategy** |
|---|---|
| Spark job failure mid-run | Restart the job using the same Iceberg snapshot |
| Partial TigerGraph ingestion | Replay REST++ upserts (idempotent) |
| Schema evolution mismatch | Fail-fast via Unity Catalog validation |
| Network failure during upsert | Retry with exponential backoff |
| Duplicate ingestion attempt | Handled via idempotent vertex/edge upserts |

---

# 8. Bottlenecks due to not integrating the

✅ **What we fixed correctly**

**You successfully did all of this:**

- Added Iceberg Spark runtime JARs
- Added AWS SDK JARs
- Verified Ivy downloaded them
- Spark JVM started correctly
- No more “class not found” for Iceberg JARs

This chunk worked

```
org.apache.iceberg:iceberg-spark-runtime-3.5_2.12:1.5.2
software.amazon.awssdk:bundle:2.25.12
```

❌ **What is *still* failing**

```
TypeError: 'JavaPackage' object is not callable
```

**at**

```
SparkSession.builder.getOrCreate()
```

## ❗ Iceberg REST catalogs (AWS S3 Tables) do not work in local Spark

# 9. Databricks → CSV → Local (FULL CODE)

1️⃣ Databricks Spark Script: `export_iceberg_to_csv.py`

```
from pyspark.sql import SparkSession

# Databricks-compatible Spark Session
spark = (
    SparkSession.builder
    .appName("Iceberg-S3Tables-Export-To-CSV")
    .config(
        "spark.sql.extensions",
        "org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions"
    )

    # AWS S3 Tables (Iceberg REST catalog)
    .config("spark.sql.catalog.s3tables", "org.apache.iceberg.spark.SparkCatalog")
    .config("spark.sql.catalog.s3tables.type", "rest")
    .config(
        "spark.sql.catalog.s3tables.uri",
        "https://s3tables.eu-north-1.amazonaws.com"
    )
    .config(
        "spark.sql.catalog.s3tables.warehouse",
        "s3tables://agivant-s3-table-bucket"
    )

    # AWS IO + region
    .config(
        "spark.sql.catalog.s3tables.io-impl",
        "org.apache.iceberg.aws.s3.S3FileIO"
    )
    .config(
        "spark.sql.catalog.s3tables.client.region",
        "eu-north-1"
    )
    .getOrCreate()
)

print("=== Catalogs ===")
spark.sql("SHOW CATALOGS").show(truncate=False)

print("=== Namespaces ===")
spark.sql("SHOW NAMESPACES IN s3tables").show(truncate=False)
```

```
print("=== Tables ===")
spark.sql("""
SHOW TABLES IN s3tables.agivant_s3_namespace_2
""").show(truncate=False)

# Read Iceberg Tables from s3 bucket
users_df = spark.sql("""
SELECT *
FROM s3tables.agivant_s3_namespace_2.users
""")

products_df = spark.sql("""
SELECT *
FROM s3tables.agivant_s3_namespace_2.products
""")

transactions_df = spark.sql("""
SELECT *
FROM s3tables.agivant_s3_namespace_2.transactions
""")

# Sanity check
print("=== Products Sample ===")
products_df.show(10, truncate=False)

# Write CSVs to S3 (EXPORT ZONE)
EXPORT_BASE = "s3://agivant-export/csv"

(
   users_df
   .coalesce(1)                    # optional: fewer files
   .write
   .mode("overwrite")
   .option("header", True)
   .csv(f"{EXPORT_BASE}/users")
)

(
   products_df
   .coalesce(1)
   .write
   .mode("overwrite")
   .option("header", True)
   .csv(f"{EXPORT_BASE}/products")
)
```

```
(
   transactions_df
   .coalesce(1)
   .write
   .mode("overwrite")
   .option("header", True)
   .csv(f"{EXPORT_BASE}/transactions")
)

print("✅ Iceberg tables exported to CSV successfully")

spark.stop()
```

2️⃣ What this produces in S3

```
s3://agivant-export/csv/
├── users/
│   ├── part-00000-*.csv
│   └── _SUCCESS
├── products/
│   ├── part-00000-*.csv
│   └── _SUCCESS
├── transactions/
│   ├── part-00000-*.csv
│   └── _SUCCESS
```

3️⃣ Download CSVs locally

```
aws s3 sync s3://agivant-export/csv ./csv_export
```

We'll get:

```
csv_export/
├── users/
├── products/
└── transactions/
```

We can rename or move as needed:

```
mv csv_export/users/*.csv users.csv
mv csv_export/products/*.csv products.csv
mv csv_export/transactions/*.csv transactions.csv
```