Solution Proposal Document

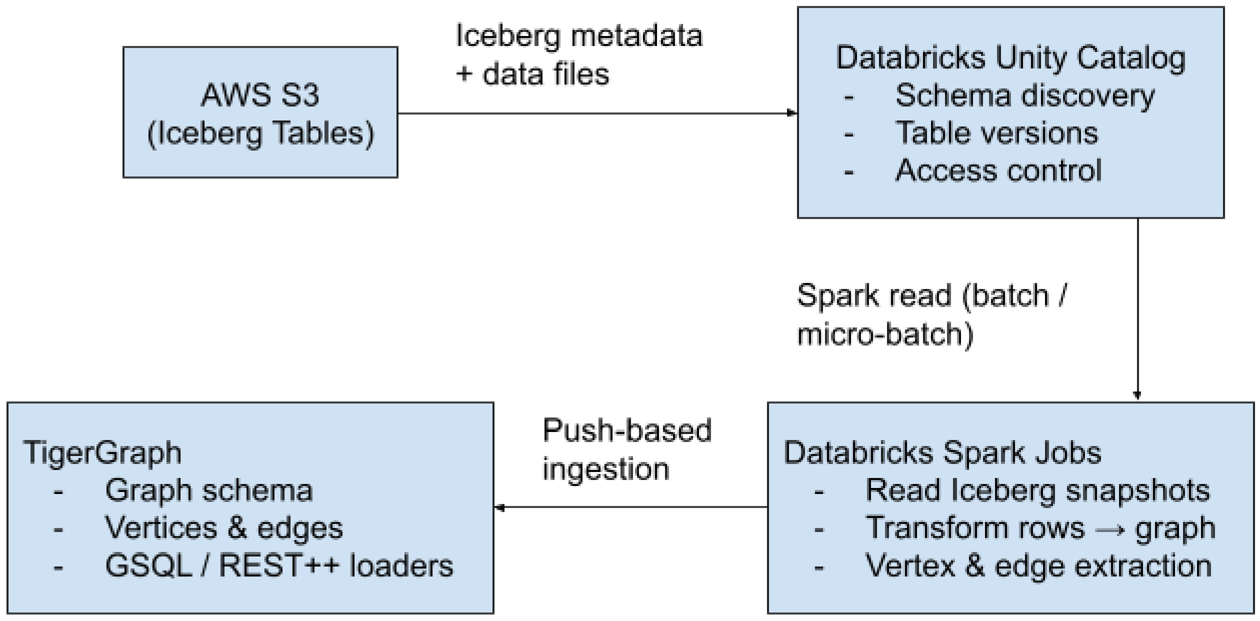
1. Overall architecture

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

High-Level Architecture Diagram (Conceptual)



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

Why Iceberg matters here

- 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

Why does it 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
- Selecting the correct snapshot or time window
- Transforming relational rows into graph entities
- Preparing TigerGraph-compatible payloads

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 Used

- **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
3. Schema is inferred dynamically
4. Rows are transformed:
    - Entities → vertices
    - Foreign-key relationships → edges
5. Transformed data is pushed to TigerGraph
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
2. A consistent snapshot (or snapshot range) is selected
3. Data is transformed into graph vertices and edges
4. Data is bulk-loaded into TigerGraph

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

This aligns well with:

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

Why Batch Fits TigerGraph Well

TigerGraph ingestion is optimized for:

- High-throughput bulk loading
- GSQL loading jobs
- Idempotent batch operations

Batch ingestion allows:

- Efficient creation 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
2. Spark transforms tabular data into graph primitives
3. Spark pushes transformed data into TigerGraph using:
    - GSQL loading jobs (batch)
    - REST++ APIs (incremental)

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

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 load

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.

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 payloads, such as:

- CSV files for bulk loading
- JSON records for REST-based ingestion

Payloads are structured to align with:

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

This ensures the ingestion layer is **schema-aware and deterministic**.

Step 6: Data Ingestion into TigerGraph (Destination Layer)

The transformed data is pushed into **TigerGraph** using native ingestion mechanisms:

Batch Ingestion

- GSQL loading jobs ingest large volumes efficiently
- Optimized for throughput and idempotency

Incremental Updates (Optional)

- REST++ APIs ingest smaller updates
- Used for near-real-time or micro-batch updates

TigerGraph is responsible only for:

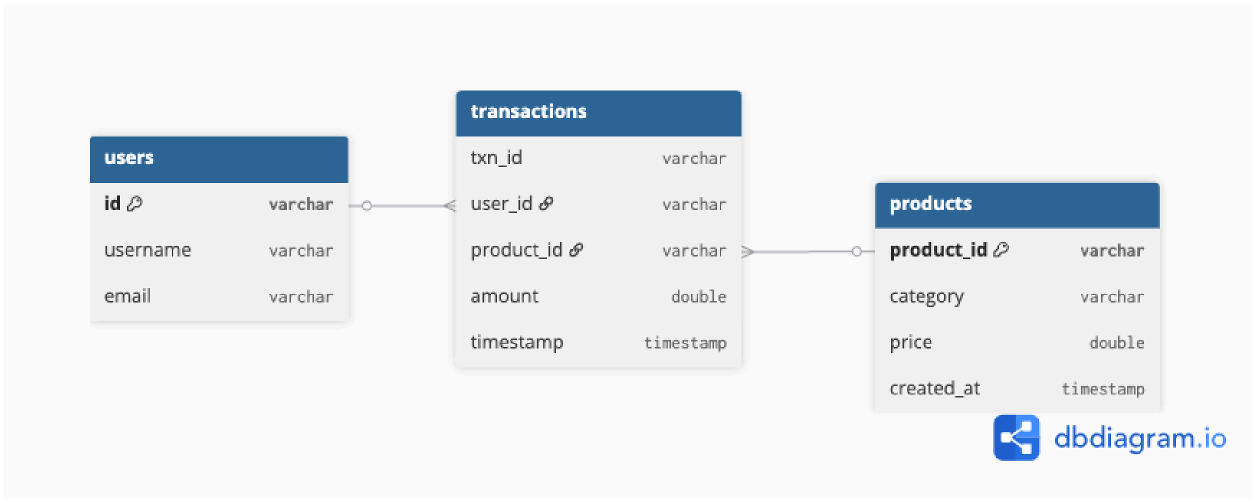
- Storing vertices and edges
- Maintaining graph indexes
- Serving graph queries

It does **not** directly access Iceberg or Unity Catalog.

---

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

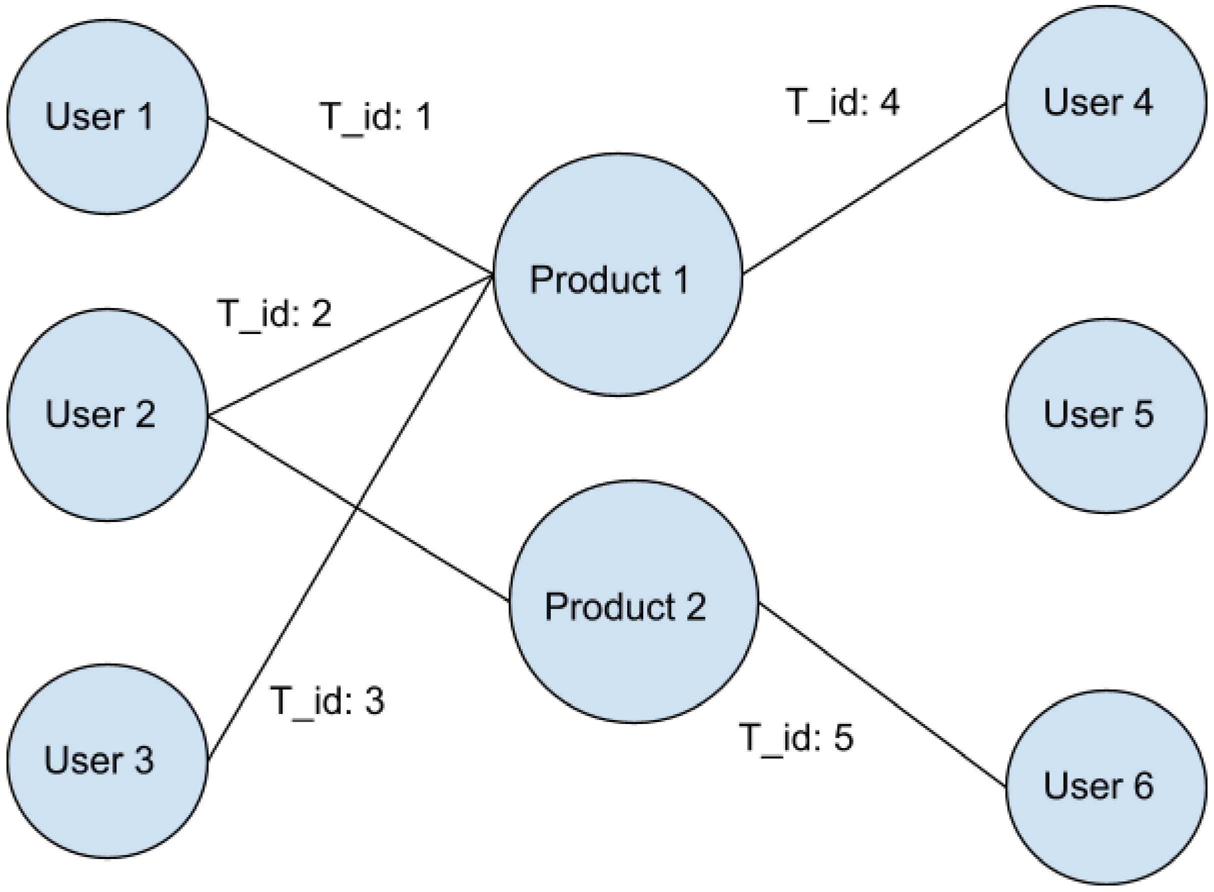
Source Tables (Iceberg)



users	
id	varchar
username	varchar
email	varchar

transactions	
txn_id	varchar
user_id	varchar
product_id	varchar
amount	double
timestamp	timestamp

products	
product_id	varchar
category	varchar
price	double
created_at	timestamp



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"
  }
}
```

PURCHASED Edge (TigerGraph)

```
{
  "edge_type": "PURCHASED",
  "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 before being ingested via GSQL loading jobs or REST++ APIs.

---

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 it works well here:

- Apache Iceberg is snapshot-oriented by design
- TigerGraph is optimized for bulk loading
- 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 it 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.

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.

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	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.