DECOUPLED GRAPH COMPUTE

Analyzing Petabyte-Scale Data Without Migration

Lessons from building TigerGraph, Dgraph, and PuppyGraph

Weimo Liu
CEO of PuppyGraph

PuppyGraph

Query on the data. No migration required.

THE BOTTLENECK

Petabytes don't move easily

Traditional graph databases assume your data lives inside them. At petabyte scale, that assumption breaks first — long before query performance ever becomes the issue.

PBs

of data to ingest before a single query runs

2x

storage cost — one copy in the lake, one in the graph DB

Days

of ETL pipeline time before insights are current

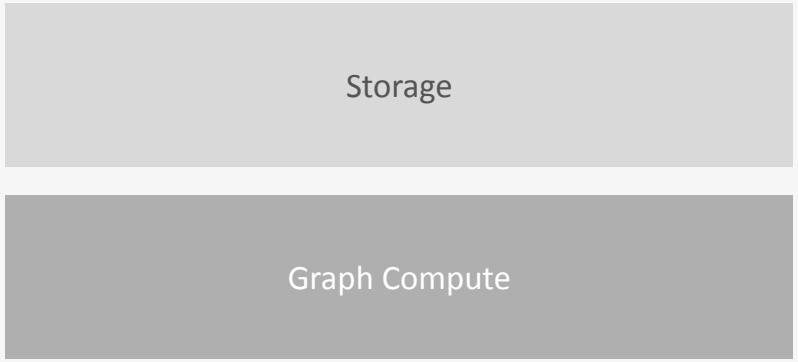
The migration tax

- Extract, transform, and load before any analysis starts
- Data goes stale the moment the copy is made
- Specialized storage engines can't scale as elastically as tables
- Every new source repeats the same costly pipeline

THE SHIFT

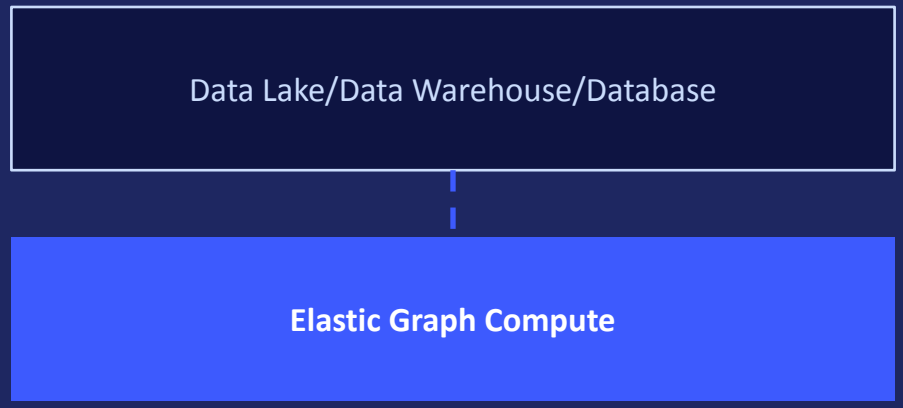
From coupled databases to decoupled engines

Coupled (traditional)



One engine, one box. Scale storage and compute together — whether you need to or not.

Decoupled (big data native)



Compute spins up against data in place, then scales to zero. Storage stays where it already is.

ARCHITECTURAL LESSONS

4 common implementation strategies for graph databases

Design	Core Idea	Advantages	Limitations	Examples
1. Key-value store + adjacency lists	Store each vertex as a key and its adjacency list as the value. Traversals retrieve neighbors through KV lookups.	Simple architecture, efficient local traversals, horizontally scalable.	Random I/O for deep traversals, traversals across partitions can be expensive.	JanusGraph, Dgraph
2. Query translation (Graph → SQL)	Translate Cypher/Gremlin/SPARQL into SQL executed by a relational database.	Reuses mature SQL engines, ACID transactions, no new storage layer.	Recursive traversals, variable-length paths, and graph optimizations are difficult to express efficiently in SQL.	AgensGraph, Apache AGE, Oracle PGQL, SQL Graph
3. In-memory graph engine	Load the entire graph (or active subgraph) into RAM for computation.	Extremely fast graph algorithms and traversals.	Limited by memory capacity and loading time; less suitable for petabyte-scale data.	Kuzu (primarily in-memory execution with disk support), NetworkX, Ligra, GraphMat
4. Native pointer-based graph storage	Store vertices and edges as linked objects (often doubly linked or pointer-based adjacency structures).	Constant-time neighbor access, efficient updates, ideal for OLTP workloads.	Pointer chasing hurts cache locality; difficult to distribute and compress.	Neo4j native record store, Memgraph, many academic graph databases

UNDER THE HOOD

Query-on-tables, step by step

1

Register

Point the engine at existing tables — Parquet, Iceberg, Delta, Databricks, Snowflake, Postgres — no copy, no load job.

2

Plan

The query planner turns a graph traversal into a set of node and edge operators.

3

Execute

Elastic compute workers pull only the data that the traversal actually touches.

4

Materialize

Results—subgraphs, paths, and aggregates—are returned to the client or written back to the underlying data store.

The result: complex queries such as 10-hop neighbor query on billions of edges in subsecond.

PuppyGraph Customers & Users

IN PRACTICE

What it takes to run this on your tables

1. **Tables -> Graph**

   Iceberg, Databricks, Snowflake, ClickHouse, Postgres tables with graph-shaped schemas (edges + vertices as tables).

2. **A cost-based graph planner**

   One that understands node and edge operators, not just single-table scans.

3. **Elastic compute**

   Container or serverless workers that scale to the traversal, not a fixed cluster size.

4. **Table-aware execution**

   Partition pruning and predicate pushdown at the storage layer.

Compute goes to the data.
Not the other way around.

Decoupled graph compute lets you run petabyte-scale graph analytics directly on the data storage you already have — no migration, no shadow copy, no stale results.

Thank you