Titan Graph Database

CIS 4930/6930 Advanced Databases

Group 18
Thomas Baldwin
Jiayong Li
Zhaohe Xu
Jaswinder Sodhi
Outline

- Introductions to graph database
- Characteristic features
- Important implementation concepts
- Data model
- Queries and operations
Graph Concept

- In mathematics, the representation of graph is $G = (V, E)$.
- In computer science, Graph is an abstract data type that implements the math concepts.
- Graph have different attributes (weight, numeric attribute)
- Comes with different operations.
Operations

- adjacent($G, x, y$)
- neighbors($G, x$)
- add_vertex($G, x$)
- remove_vertex($G, x$)
- add_edge($G, x, y$)
- remove_edge($G, x, y$)
What is Graph Databases

- As name suggests, it is a database.
- Uses graph structures for semantic queries with nodes, edges and properties to represent and store data.
- The relationships allow data in the store to be linked together directly.
- contrasts with conventional relational databases.
Relational Data Model

- Relational tables, SQL and joins.
- Works pretty well at beginning.
- Join processing is expensive
- Inflexible data model.
Graph Data Model
But In Reality...

- Hybrid relations.
- Easy to change the current data model
- Flexible data model
- Handy in finding connections between entities
Overview of Architecture

- Titan itself is a graph database engine / database server / database management system.
- Titan itself is focused on compact graph serialization, rich graph data modeling, and query execution.
- Titan utilizes Hadoop for graph analytics and batch graph processing.
- Have multiple options for the backend storage system.
Introduction of Titan

- A powerful graph database
- Design for giant graph computing beyond what a single machine can provide
- Support real time traversals and analytical queries and other amazing features.
- Good choice for large scale Social Network applications (More examples later)
Overview of Architecture
Ecosystem

- Introductions to graph database
- Characteristic features
- Important implementation concepts
- Data model
- Queries and operations
What titan offers
Build for transactions

- High number concurrent Threads
- Incremental transactional capacity
- Answers complex queries
Consistency

Eventual consistency

Support for ACID
Dynamic Scalability

- In size of graph
- In number of vertices
- Infinite size graphs
- Unlimited users
- Multi data center replication.
Backend Support

Hbase, BerkeleyDB
Supports cassandra tables
Decentralized
Linear scalability
Fault tolerance
very high data volumes
Deployed in horizontal scale out fashion
Support for gremlin

Path oriented

Gremlin Console

Gremlin language

Gremlin server: Rexster
OLTP

Real time local traversals

Transactional systems

Multi threaded transactions
Data Analytics

- Global graph analytics
- Batch graph processing (Hadoop)
- Discover trends
- Apache Spark
Integration with tinkerpop stack

Graph computing framework

Allows gremlin

In memory vs distributed processing

The rest of the TinkerPop family:

- Pipes: dataflow framework. The basis of Gremlin
- Frames: Java bean framework for graphs
- Furnace: Property Graph algorithms
- Rexster: high-performance graph database server
The market titan aims at..

Do you value the connections?
Ready to scale?
Innovative queries?
Intuitive modeling
Inference
Ranking
Recommendation
Social networks
Sample use case

Movie graph with movies and actors.

See how old school RDBMS cannot run cool queries.
Data Model

- Schema and Data Modeling
- BigTable Data Model
- Titan Data Model
Schema and Data Modeling
BigTable Data Model

- Key -> vertex id
- Order in Titan
<table>
<thead>
<tr>
<th>Operation</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single vertex</td>
<td><code>g.V(4160)</code></td>
</tr>
<tr>
<td>Matching a property</td>
<td><code>g.V().has(&quot;name&quot;, &quot;Jupiter&quot;)</code></td>
</tr>
<tr>
<td>Range filtering</td>
<td><code>g.V().has(&quot;age&quot;, between(2000, 5000))</code></td>
</tr>
<tr>
<td>To other vertices</td>
<td><code>g.V().has(&quot;name&quot;, &quot;Jupiter&quot;).out()</code></td>
</tr>
<tr>
<td>To edges</td>
<td><code>g.V().has(&quot;name&quot;, &quot;Jupiter&quot;).outE()</code></td>
</tr>
<tr>
<td>Filtering with traversals</td>
<td><code>g.V().has(&quot;name&quot;, &quot;Jupiter&quot;).out().has(&quot;age&quot;, between(2000, 5000))</code></td>
</tr>
</tbody>
</table>
Java API

- Common Architecture
- Package Overview
- Create and Retrieve
Cluster cluster = Cluster.build("10.234.31.163").create();
Client client = cluster.connect();

...  
Client.submit("g.V()");
# API Packages

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.thinkaurelius.titan.core</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.core.attribute</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.core.log</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.core.schema</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.core.util</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.diskstorage</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.diskstorage.common</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.diskstorage.configuration</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.diskstorage.configuration.backend</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.diskstorage.idmanagement</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.diskstorage.indexing</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.diskstorage.keycolumnvalue</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.diskstorage.keycolumnvalue.cache</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.diskstorage.keycolumnvalue.inmemory</td>
<td></td>
</tr>
<tr>
<td>com.thinkaurelius.titan.diskstorage.keycolumnvalue.keyvalue</td>
<td></td>
</tr>
</tbody>
</table>
Create and Retrieve Example

```java
BaseConfiguration baseConfiguration = new BaseConfiguration();
baseConfiguration.setProperty("storage.backend", Cassandra);
baseConfiguration.setProperty("storage.hostname", "192.168.1.10");
TitanGraph titanGraph = TitanFactory.open(baseConfiguration);

Vertex rash = titanGraph.addVertex(null);
Vertex honey = titanGraph.addVertex(null);
rash.setProperty("userId", 1);
rash.setProperty("username", "rash");
rash.setProperty("birthday", 1990);
honey.setProperty("userId", 2);
honey.setProperty("username", "honey");
honey.setProperty("birthday", 1991);

Edge frnd = titanGraph.addEdge(null, rash, honey, "FRIEND");
frnd.setProperty("since", 2016);

titanGraph.commit();
```
Building Applications With Titan

AJAX Request

Keylines

Rexster

Titan:DB
Handling the Frontend...

AJAX Request

Java API (HTTP Client)

Keylines

Visually format the graph data returned

The best way to understand it is to visualize it.
Rexster on the Backend...

Rexster is a graph server that exposes graph through REST and a binary protocol called RexPro.

Provides standard low-level GET, POST, PUT, and DELETE methods

“The Dog House”
The Dog House
What does it look like all together?
Implementation Concepts

Gremlin Query Language
The Titan Server
Bulk Loading
Graph Partitioning
Gremlin Query Language

Titan’s query language used to retrieve data from and modify data in the graph

Path-oriented language which succinctly expresses complex graph traversals and mutation operations

Functional language whereby traversal operators are chained together to form path-like expressions
The Titan Server

Titan uses the Gremlin Server engine as the server component to process and answer client queries.

The Gremlin Server provides a way to remotely execute Gremlin scripts against one or more Titan instances hosted within it.

Client applications can connect to it via WebSockets with a custom subprotocol.

Can also be configured to serve as a REST-style endpoint.
Why the need for Bulk Loading?

Introducing Titan into an existing environment with existing data and migrating or duplicating this data into a new Titan cluster.

Adding an existing or external graph datasets to a running Titan cluster.

Updating a Titan graph with results from a graph analytics job.
What is Graph Partitioning?

When the Titan cluster consists of multiple storage backend instances, the graph must be partitioned across those machines.

Different ways to partition a graph:

- Random Graph Partitioning
- Explicit Graph Partitioning
Random Graph Partitioning

Pros

Very efficient

Requires no configuration

Results in balanced partitions

Cons

Less efficient query processing as the Titan cluster grows

Requires more cross-instance communication to retrieve the desired
Explicit Graph Partitioning

Pros

- Ensures strongly connected subgraphs are stored on the same instance
- Reduces the communication overhead significantly
- Easy to setup

Cons

- Only enabled against storage backends that support ordered key storage, e.g., HBase
Future of Titan

Byte order partitioner (partition graphs effectively so that data is available locally)
Ability to write hadoop jobs through gremlin
Loading subgraphs to run in-memory and running algorithms
DataStax (the firm behind the Cassandra DBMS for enterprise) acquired Aurelius (the team behind the Titan project) earlier this year. Work has started on a commercial, scalable graph database called DSE graph
90% of current data was created in past two years.