Titan Graph Database

CIS 4930/6930 Advanced Databases

Group 18

Thomas Baldwin
Jiayong Li
Zhaohe Xu
Jaswinder Sodhi

Outline

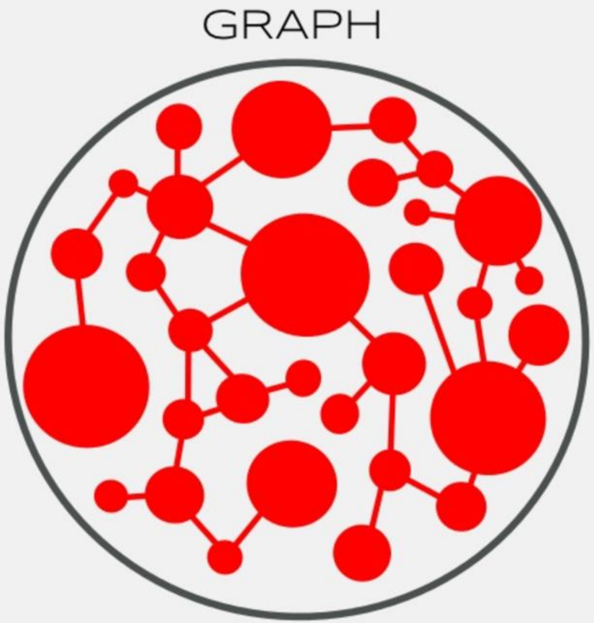
- Introductions to graph database
- Characteristic features
- Important implementation concepts
- Data model
- Queries and operations

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

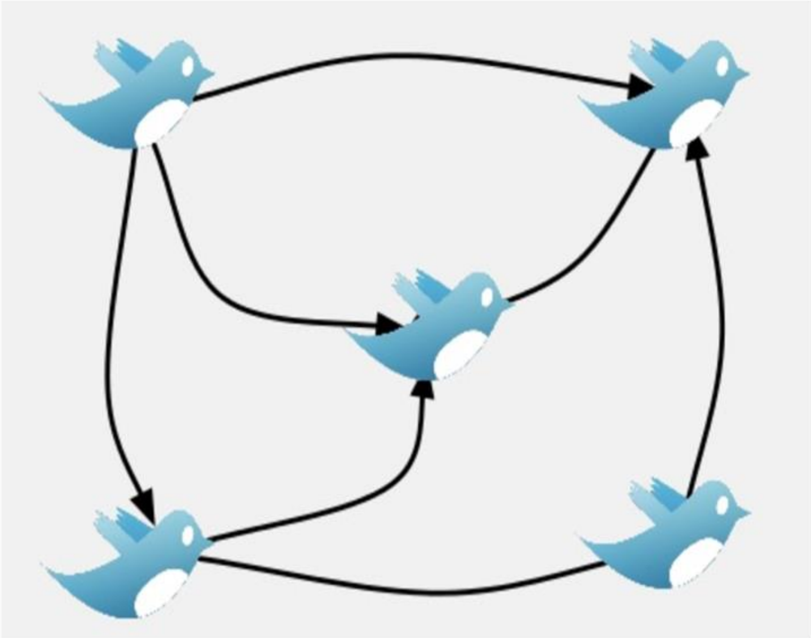
- As name suggests, it is a database.
- Uses graph structures for semantic queries with nodes, edges and properties to represent and store data
- The relationships allow data in the store to be linked together directly
- contrasts with conventional relational databases

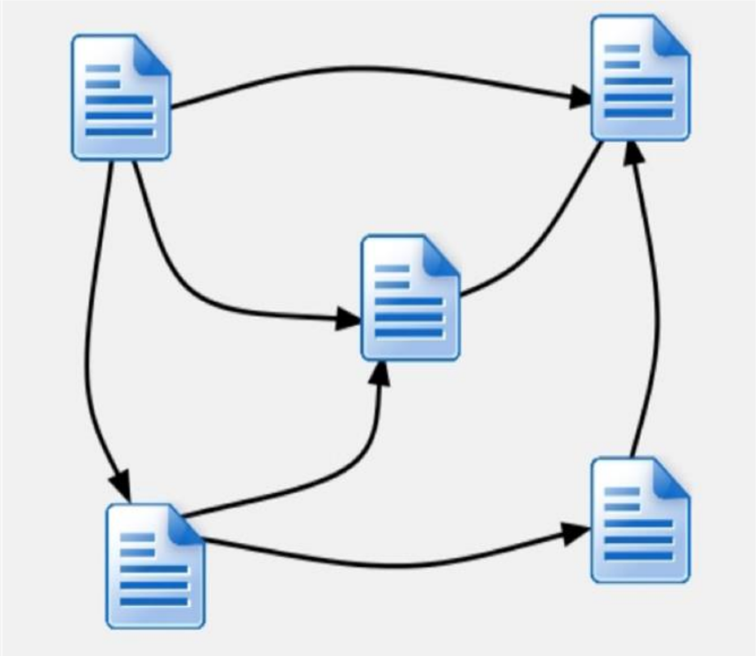
Relational Data Model

- Relational tables, SQL and joins.
- Works pretty well at beginning.
- Join processing is expensive
- Inflexible data model.

Name	Type
Hercules	demigod
Alcmene	human
Jupiter	god
Saturn	titan
Pluto	god
Neptune	god
Cerberus	monster

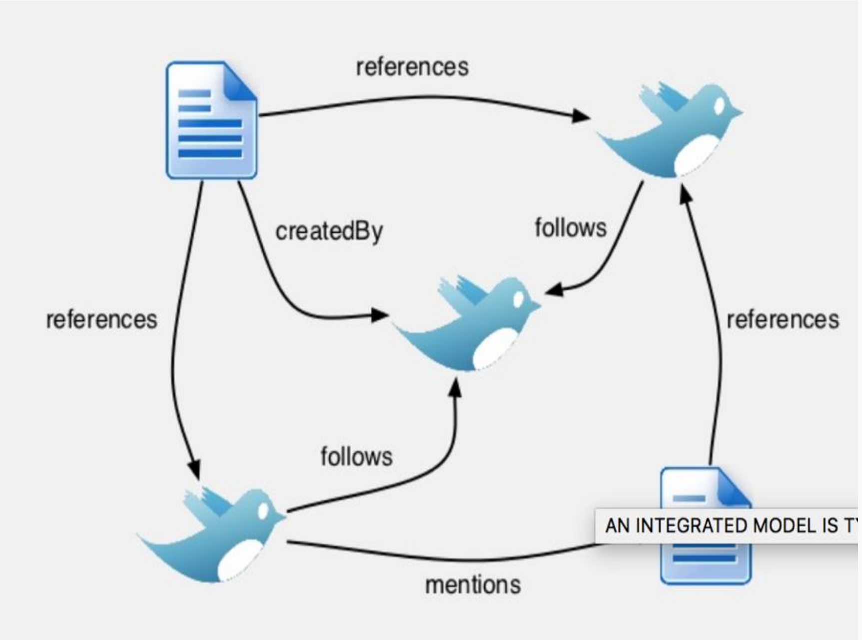
Graph Data Model





But In Reality...

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



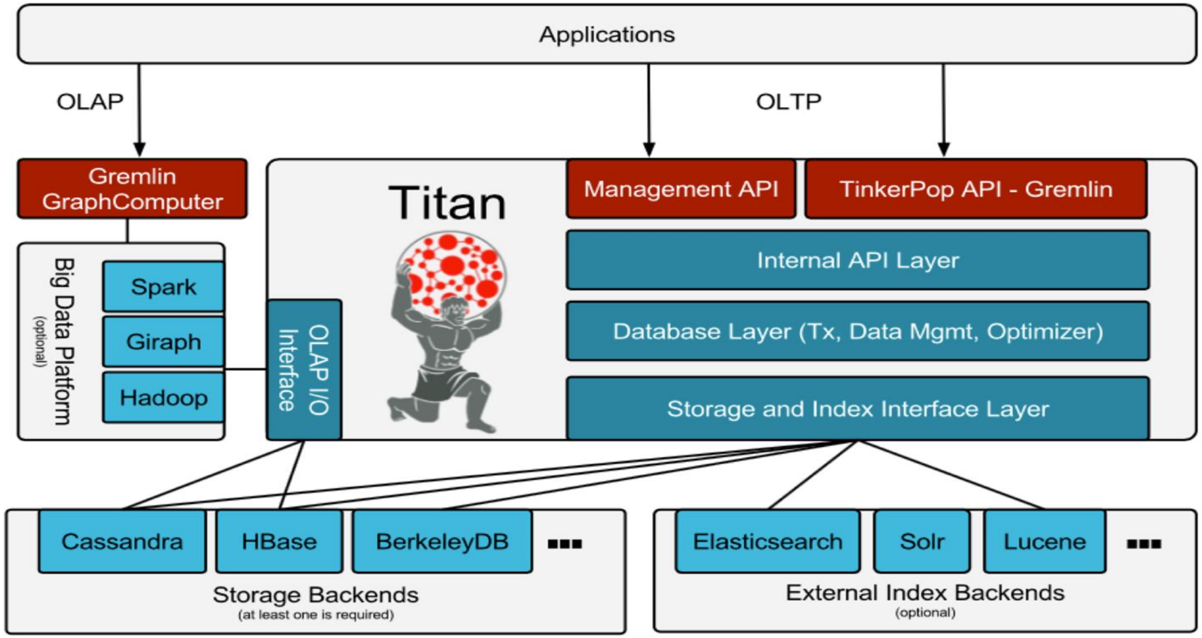
Overview of Architecture

- 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

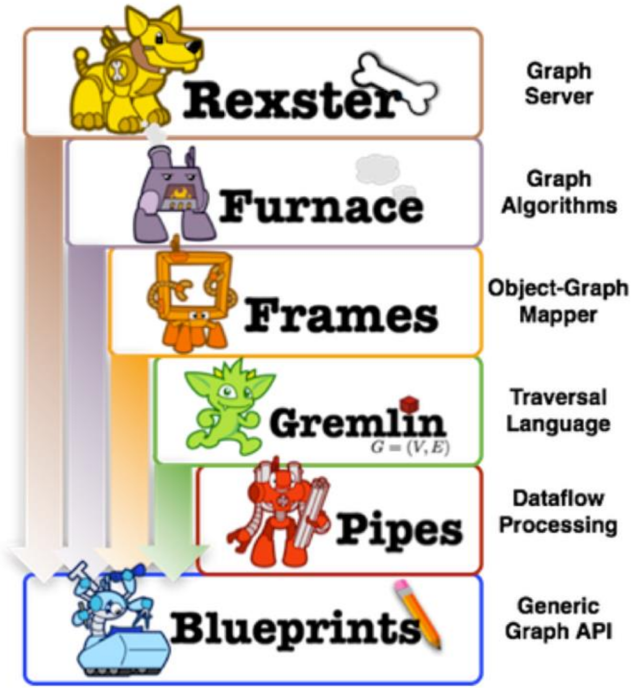
- A powerful graph database
- Design for giant graph computing beyond what a single machine can provide
- Support real time traversals and analytical queries and other amazing features.
- Good choice for large scale Social Network applications(More examples later)

Overview of Architecture

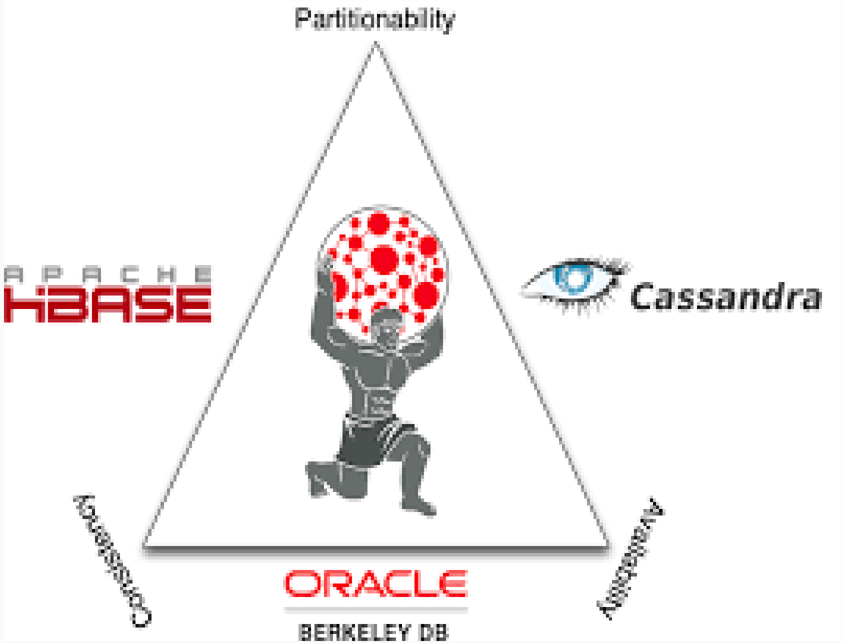


Ecosystem

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



What titan offers



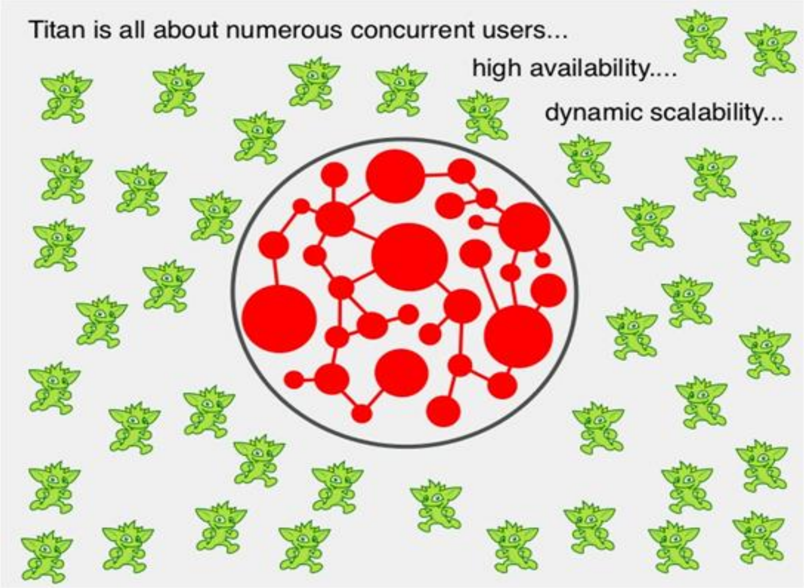
Build for transactions

High number concurrent

Threads

Incremental transactional capacity

Answers complex queries



Consistency

Eventual consistency

Support for ACID

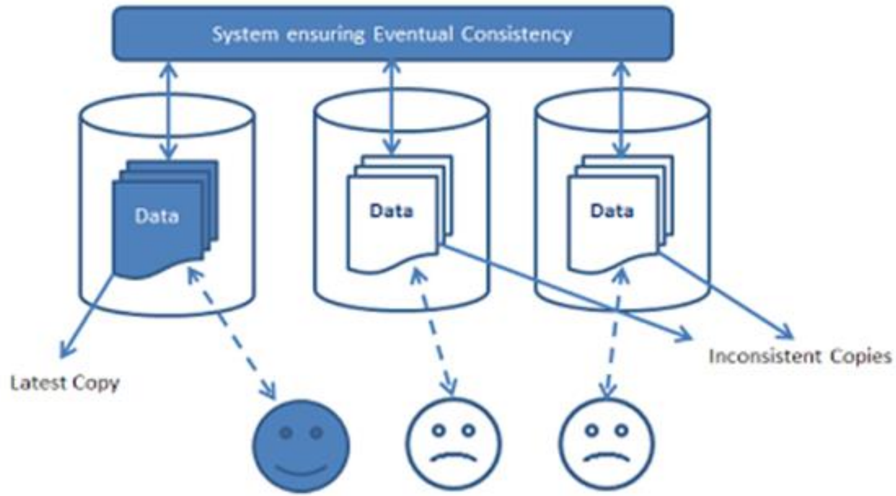


Figure showing a stale state where 2 copies of data are inconsistent with the latest one.

Dynamic Scalability

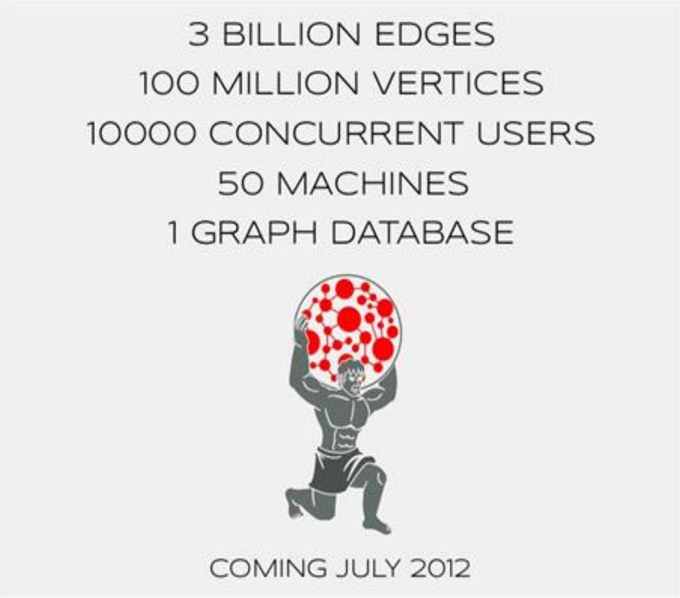
In size of graph

In number of vertices

Infinite size graphs

Unlimited users

Multi data center replication.



Backend Support

Hbase , BerkeleyDB

Supports cassandra tables

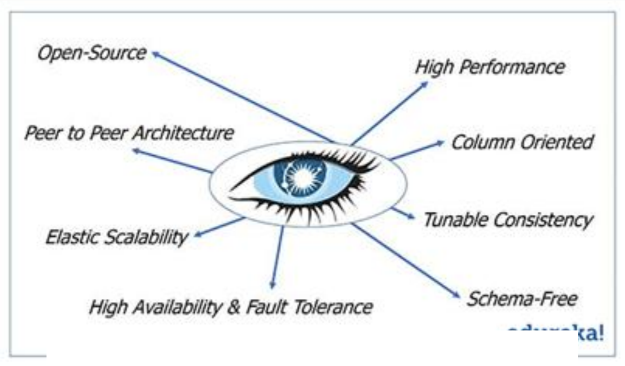
Decentralized

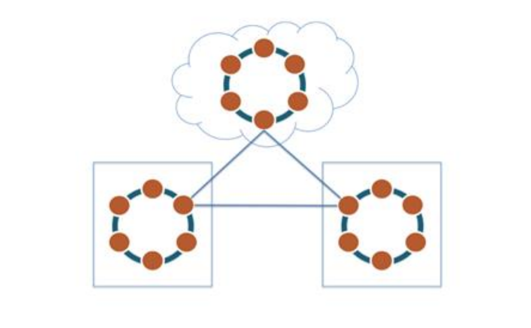
Linear scalability

Fault tolerance

very high data volumes

Deployed in horizontal scale out fashion





Support for gremlin

Path oriented

Gremlin Console

Gremlin language

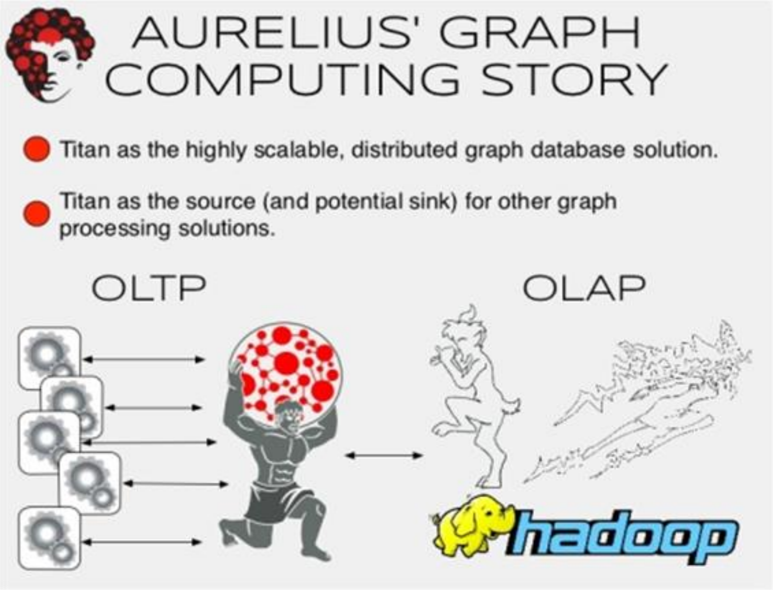
Gremlin server:Rexster

OLTP

Real time local traversals

Transactional systems

Multi threaded transactions



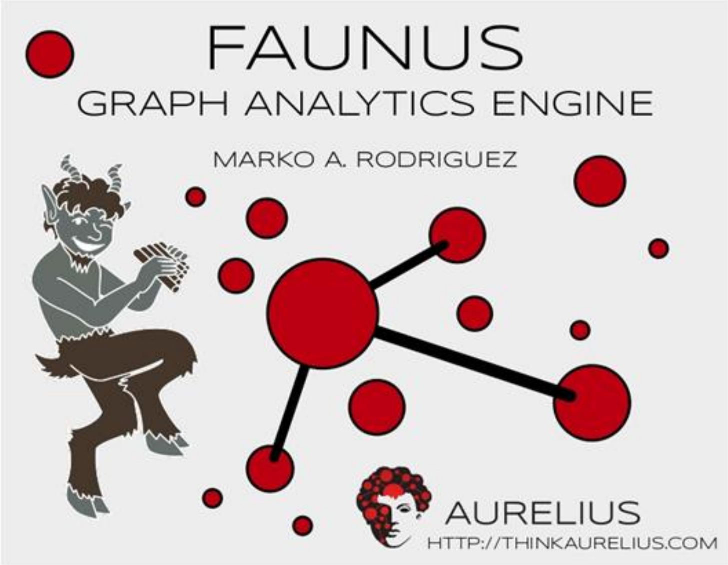
Data Analytics

Global graph analytics

Batch graph processing (Hadoop)

Discover trends

Apache Spark

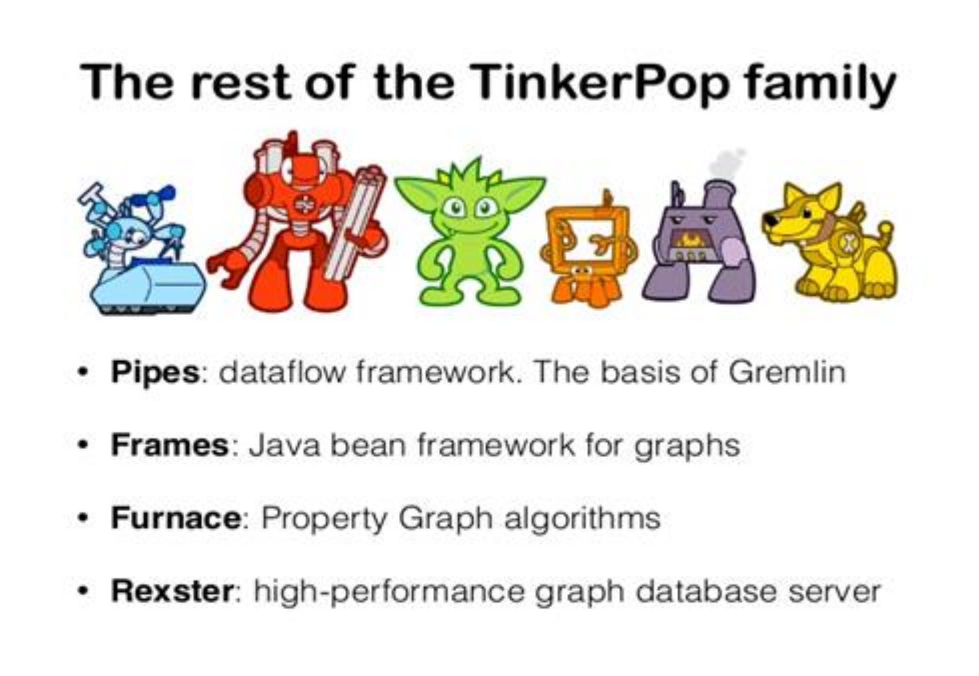


Integration with tinkerpop stack

Graph computing framework

Allows gremlin

In memory vs distributed processin



The market titan aims at..

Do you value the connections ?

Ready to scale ?

Innovative queries ?

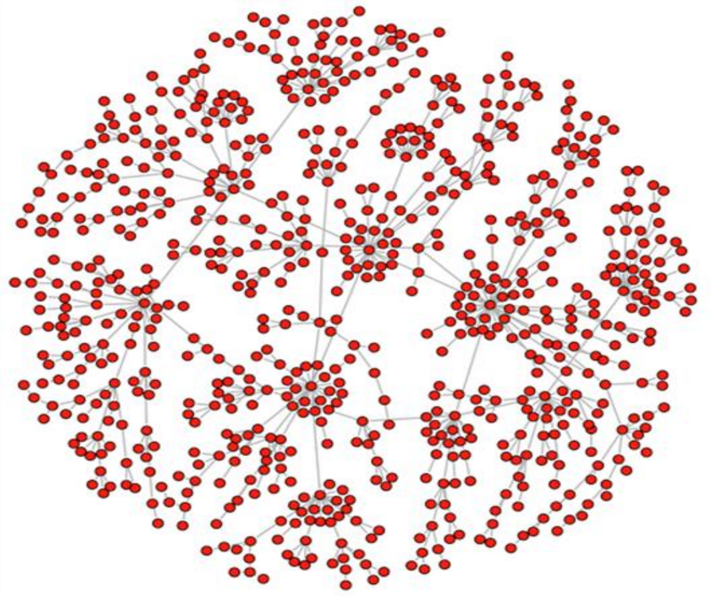
Intuitive modeling

Inference

Ranking

Recommendation

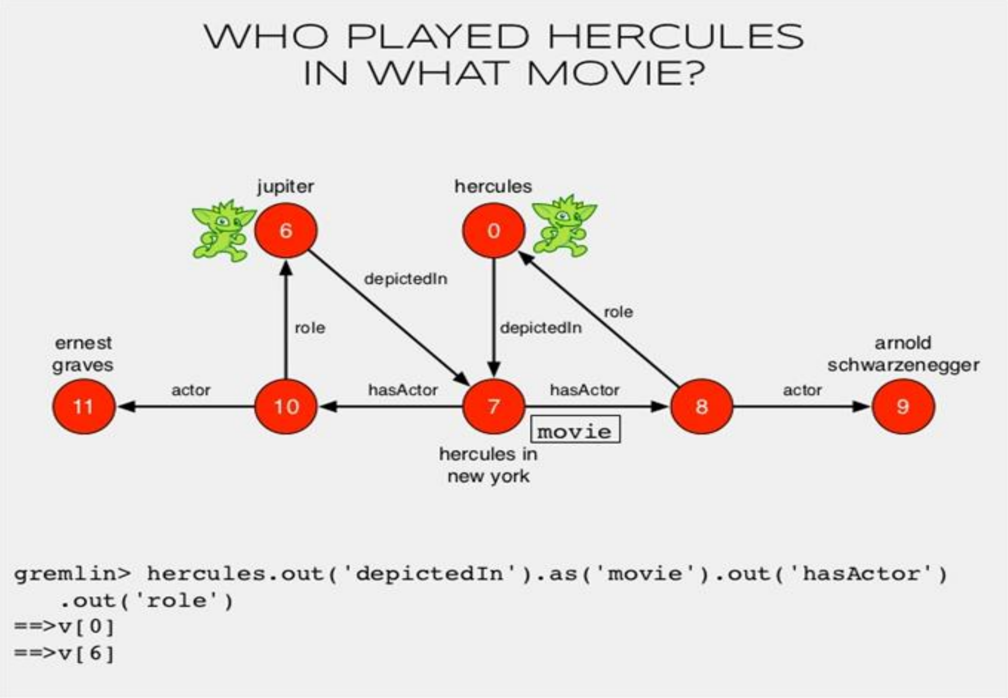
Social networks



Sample use case

Movie graph with movies and actors.

See how old school RDBMS cannot run cool queries



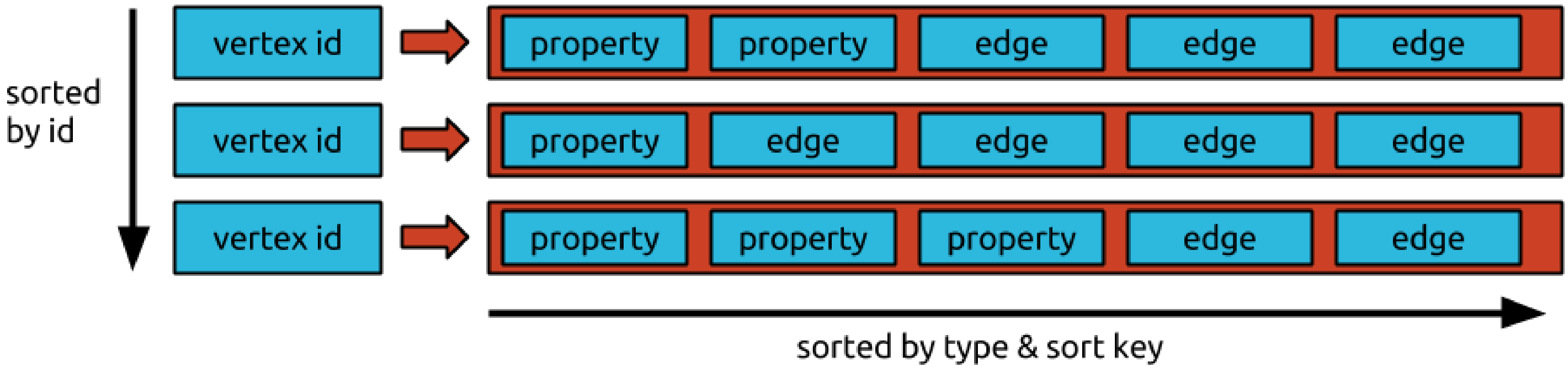
Data Model

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

Schema and Data Modeling

TITAN

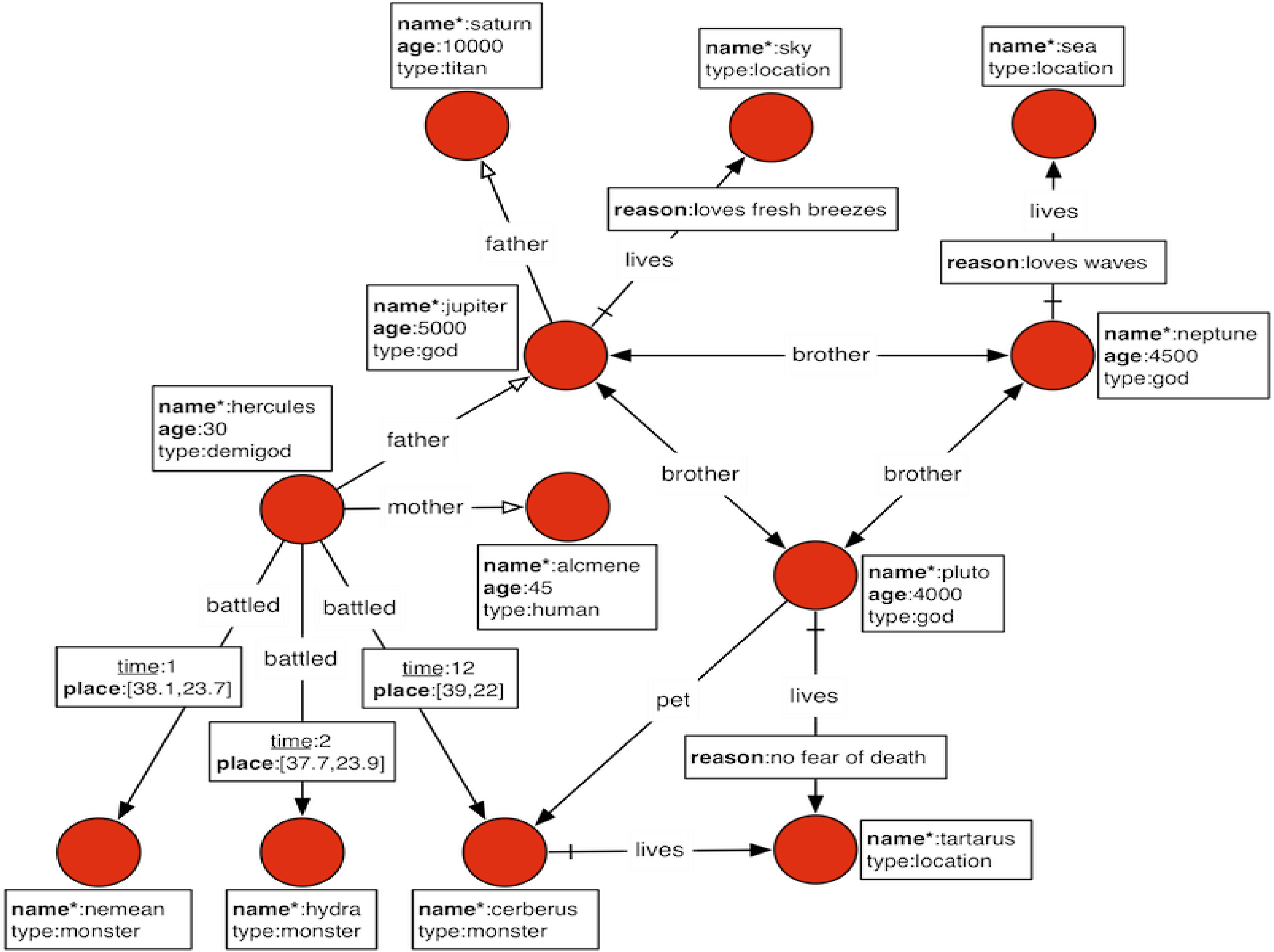
BigTable Data Model



- Key -> vertex id
- Order in Titan

Query Language: Gremlin

Operation	Query
Single vertex	g.V(4160)
Matching a property	g.V().has(“name”, “Jupiter”)
Range filtering	g.V().has(“age”, between(2000, 5000))
To other vertices	g.V().has(“name”, “Jupiter”).out()
To edges	g.V().has(“name”, “Jupiter”).outE()
Filtering with traversals	g.V().has(“name”, “Jupiter”).out(). has(“age”, between(2000, 5000))

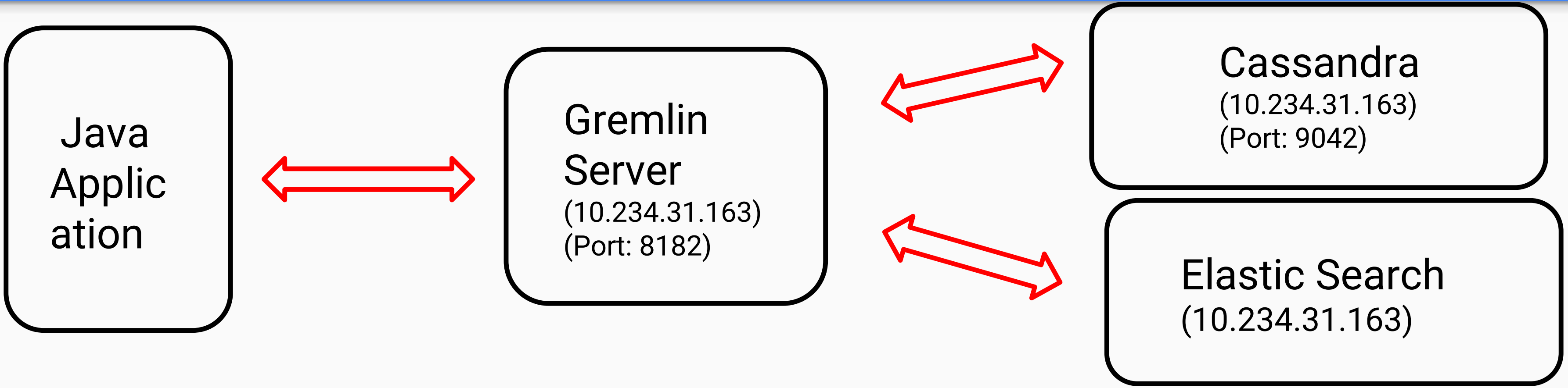


name*:saturn
age:10000
type:titan
name*:sky
type:location
name*:sea
type:location
father
reason:loves fresh breezes
lives
lives
reason:loves waves
name*:jupiter
age:5000
type:god
brother
name*:neptune
age:4500
type:god
name*:hercules
age:30
type:demigod
father
brother
brother
mother
name*:alcmene
age:45
type:human
name*:pluto
age:4000
type:god
battled
battled
battled
time:1
place:[38.1,23.7]
time:12
place:[39,22]
pet
lives
time:2
place:[37.7,23.9]
reason:no fear of death
lives
name*:tartarus
type:location
name*:nemean
type:monster
name*:hydra
type:monster
name*:cerberus
type:monster

Java API

- Common Architecture
- Package Overview
- Create and Retrieve

Common Architecture




```
Cluster cluster = Cluster.build(“10.234.31.163”).create();
Client client = cluster.connect();
…
Client.submit(“g.V()”);
```

API Packages

Package	Description
com.thinkaurelius.titan.core	
com.thinkaurelius.titan.core.attribute	
com.thinkaurelius.titan.core.log	
com.thinkaurelius.titan.core.schema	
com.thinkaurelius.titan.core.util	
com.thinkaurelius.titan.diskstorage	
com.thinkaurelius.titan.diskstorage.common	
com.thinkaurelius.titan.diskstorage.configuration	
com.thinkaurelius.titan.diskstorage.configuration.backend	
com.thinkaurelius.titan.diskstorage.idmanagement	
com.thinkaurelius.titan.diskstorage.indexing	
com.thinkaurelius.titan.diskstorage.keycolumnvalue	
com.thinkaurelius.titan.diskstorage.keycolumnvalue.cache	
com.thinkaurelius.titan.diskstorage.keycolumnvalue.inmemory	
com.thinkaurelius.titan.diskstorage.keycolumnvalue.keyvalue	

Create and Retrieve Example

```
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();
```

Configuration

Build the graph

Graph

Building Applications With Titan

AJAX Request

Keylines

Rexster

Titan:DB

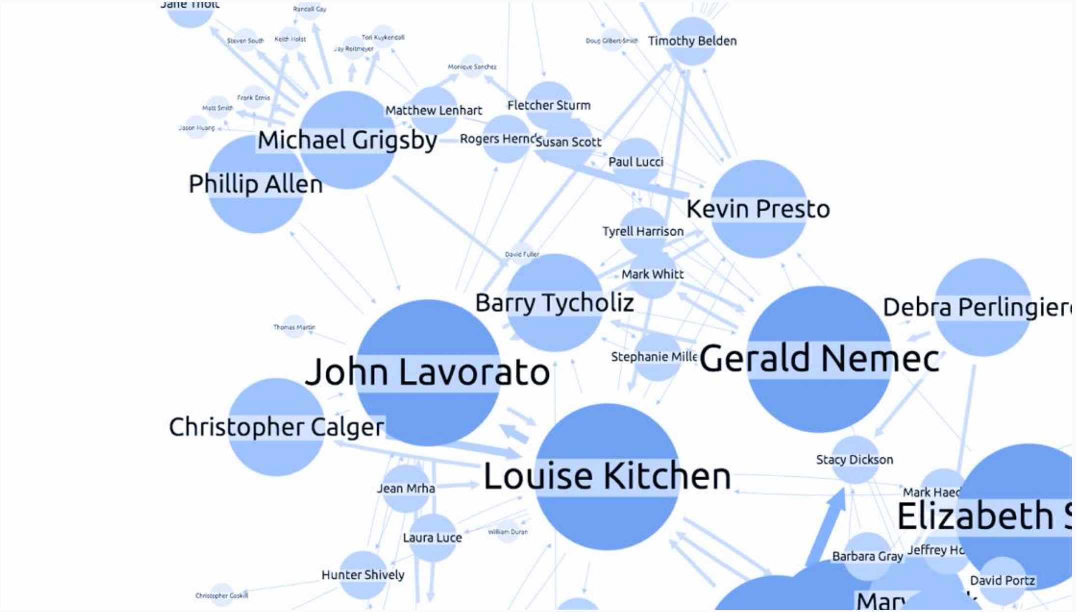
Handling the Frontend...

AJAX Request

- Java API (HTTP Client)

Keylines

- Visually format the graph data returned
- The best way to understand it is to visualize it.



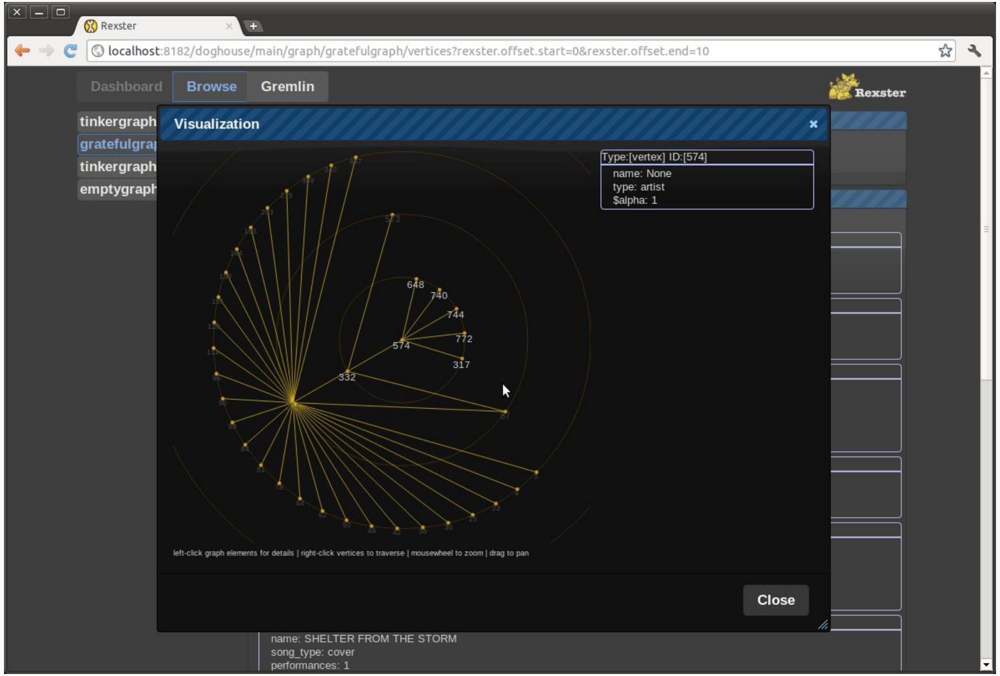
Rexster on the Backend...

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

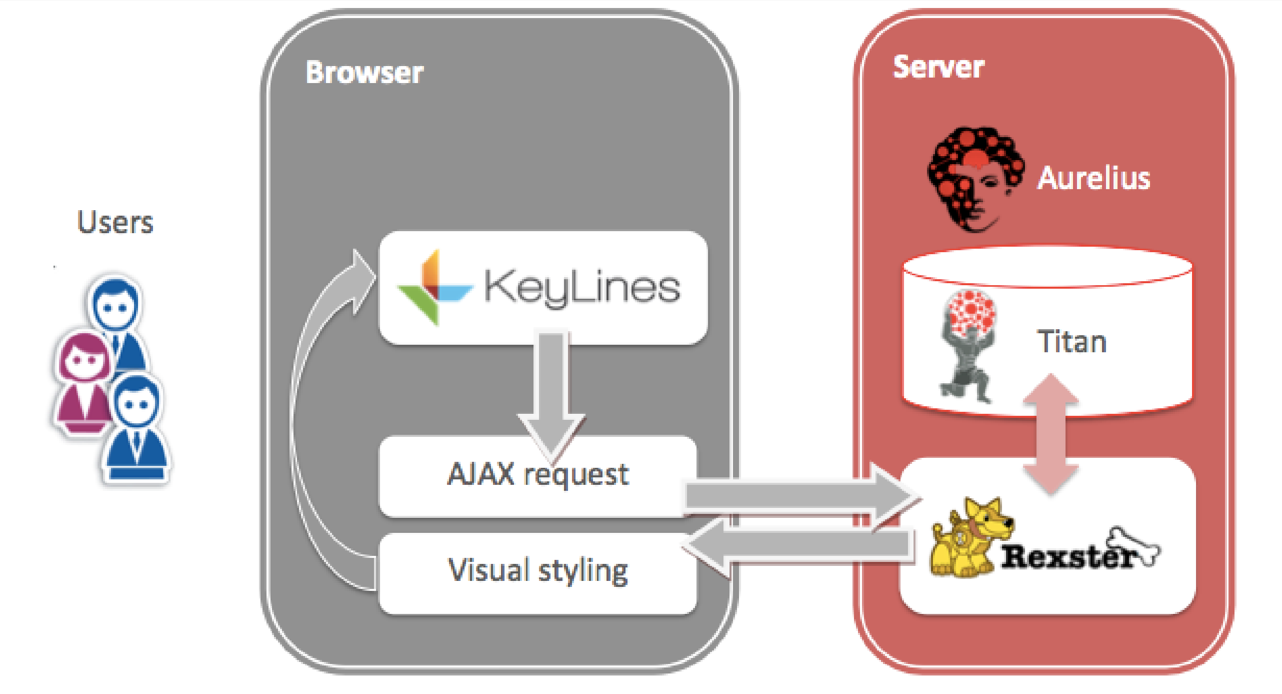
Provides standard low-level GET, POST, PUT, and DELETE methods

“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

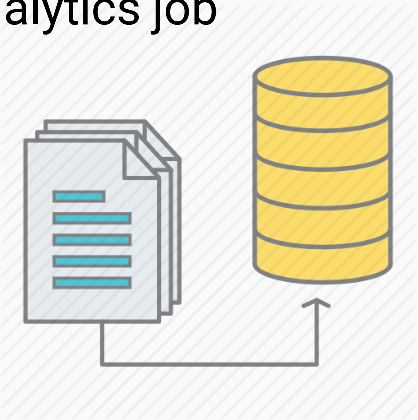
Can also be configured to serve as a REST-style endp

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

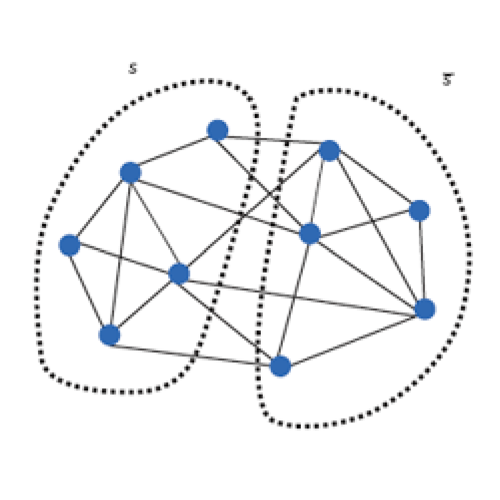


What is Graph Partitioning?

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
  - 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.