API ASSET JANUS

Technologies for Big Data Management

Students
Niccolò Francioni
Luca Mozzoni
Damiano Pasquini

Supervisor
Prof. Massimo Callisto De Donato

16th March 2023

Project overview

Asset management prototype tool based on **JanusGraph** database, able to run **CRUD operations** with focus on **search functionalities** over digital twin properties.

JanusGraph | Features

JanusGraph is a **graph-oriented** database:

- Uses graph structures with **nodes**, **edges** and **properties**
- **Graph Theory** concepts are applicable
- **Query language** is **Gremlin**, which chains together traversal operators to form **path-like expressions**
  - **e.g.** "*from node A traverse to node B and return its outgoing nodes*"
- Storage and index backends are **pluggable**
  - In our case, JanusGraph + Cassandra for storage

JanusGraph | Storage backend solutions

- JanusGraph is distributed with 3 supporting backends: **Apache Cassandra, Apache HBase and Oracle Berkeley DB**.
- The choice of backend solution falls according to specific implementation needs (e.g. **Availability over Consistency** and vice versa).
- In our case we chose **Cassandra** but, since our project is a prototype implementation we didn’t have any strict constraints to be met.

JanusGraph | Guarantees*

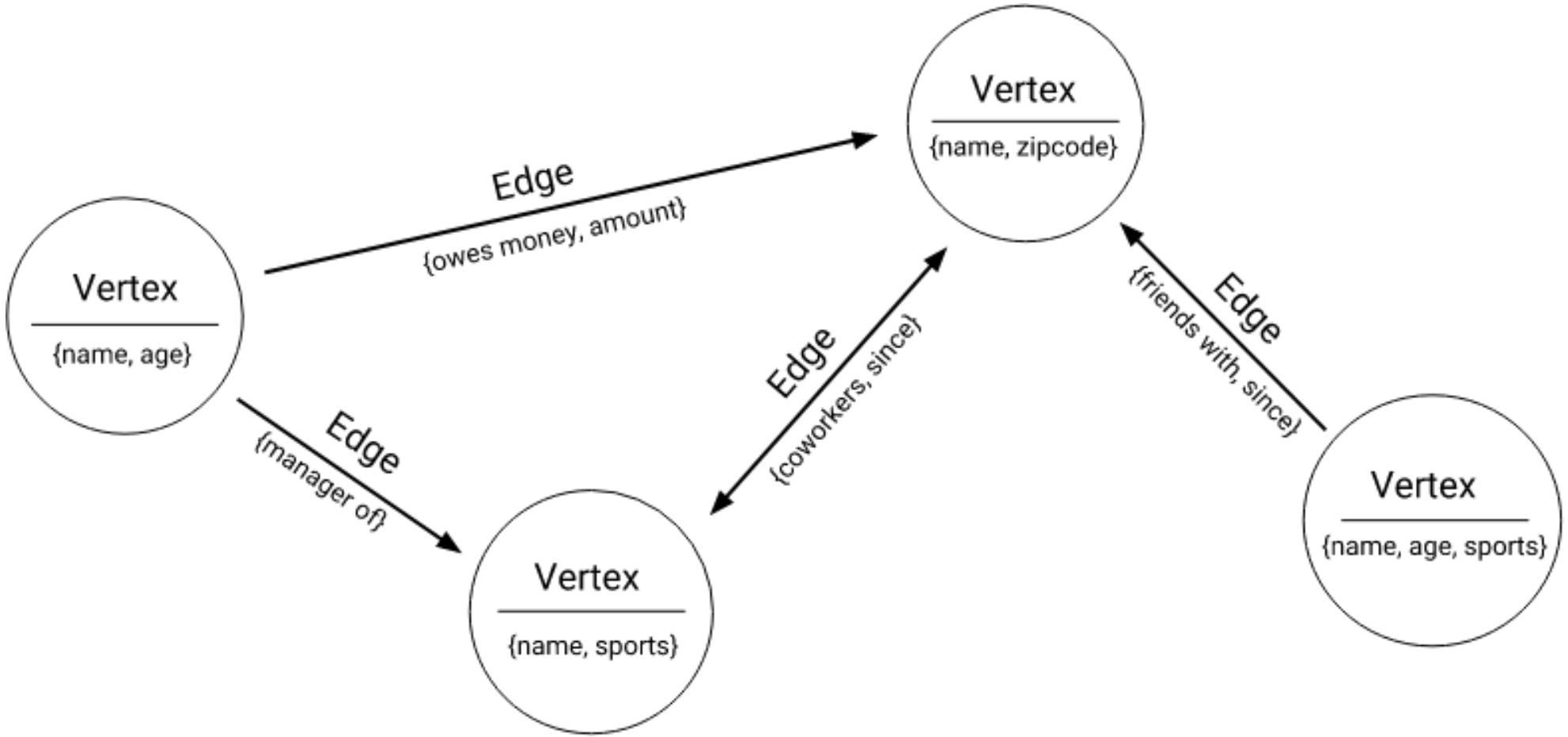
- When used with Cassandra, JanusGraph transactions are **not** ACID
  - simulating them is **costly** in terms of overhead
- Cassandra values **availability** and **partition tolerance** over **consistency** (CAP Theorem)
- Cassandra is an **eventually consistent** storage system
  - Tradeoff to achieve **high availability**
  - Involves some read and write **latencies**

***Guarantees made by JanusGraph depend on the storage backend**

JanusGraph | Schema and Data Modeling

- Each graph has a schema defined **explicitly** or **implicitly**
  - keys can be defined for **vertex/edge labels** and **property keys**
- A schema can be **evolved** over time:
  - it does **not** slow down query answering
  - it does **not** require database downtime
- Schemas allow to set property and connection **constraints:**
  - bound properties to specific vertex/edge labels
  - define which vertex labels can be connected by an edge

JanusGraph | Schema and Data Modeling



JanusGraph | Gremlin Query Language

- Gremlin is a graph traversal language used to **retrieve and modify** data
  - functional language
  - path-oriented language

- Gremlin is part of the **Apache TinkerPop** project
  - every Gremlin traversal is composed of a sequence of steps
  - a step perform an **atomic operation** on the data stream

e.g.

```
g.V().hasLabel('airport') // Find vertices that are airports
```

JanusGraph | Transactions

- Almost every interactions with JanusGraph is associated with transactions.
- Transactions are **safe for concurrent use** by multiple threads.
- Transactions are **not necessarily ACID** due to the underlying storage systems like Cassandra and HBase that do not provide such characteristics.

JanusGraph | Transactions

- Each thread **automatically opens** its own transaction
- Transactions are handled via ***commit()*** and ***rollback()*** methods
- **Thread-independent transactions** are also supported
  - multiple threads work on the same transaction
  - useful for concurrent algorithms or nested transactions
- JanusGraph automatically tries to rerun **failed transactions**
  - the number of attempts and delays are configurable

JanusGraph | Indexes

- Indexes are used to improve queries performance.
- JanusGraph currently supports:
  - Two types of indexes **graph indexes** and **vertex indexes**.
    - **composite indexes**
    - **mixed indexes**

JanusGraph | Graph Indexes

- Indexes are used to **speed up** the **random access** to the entire database.
- The goal is to get at the starting point of a query as **efficiently as possible** without having to first search the entire graph.
- Graph indexes should be established for **property keys** or **combination keys** that will be used regularly in the queries.

JanusGraph | Vertex Indexes

- Vertex indexes are associated with **vertices**.
- Typically used when the number of **incident edges** on a given vertex becomes significantly large such that it can impact on performance.
- While graph indexes are usually created at graph schema creation, vertex indexes are created as the need arises.

JanusGraph | Composite Indexes

- Composite indexes can be used to speed up queries where an **exact match** with the value for the given property key is sufficient.

```
g.V().has('type','class_room')
```

- For example the query above could take advantage of a composite index as we are only looking for exact matches where the value associated with ***type*** key is the value ***class_room***.

```
mgmt = graph.openManagement()  // Open a new management transaction
idx = mgmt.buildIndex('typeIndex',Vertex.class)
key = mgmt.getPropertyKey('type')  // Create a composite index for the type key for use with vertices
idx.addKey(key).buildCompositeIndex()
```

JanusGraph | Mixed indexes

- For queries that require more than a simple test for equality, like text predicates such as *textContains*, the creation of **mixed indexes is needed**.
- To use them an **indexing backend** is needed. The ones supported by JanusGraph are:
  - **Apache Solr**
  - **Apache Lucene**
  - **Elasticsearch**

Technical implementation | Objectives

Implement an asset management tool using **JanusGraph** able to:

- Run **CRUD operations**
- Run complex **search operations**
- Interact with a storage backend to **persist data**
- Expose **REST API endpoints**

Technical implementation | Technologies

- **JanusGraph**
  - as distributed **graph database**;
- **Apache Cassandra**
  - as **storage backend** for JanusGraph;
- **Java + Spring Boot**
  - run **Gremlin queries**
  - expose **REST APIs**

Technical implementation | Cassandra

- NOSQL column-oriented database
- Uses the **Cassandra Query Language (CQL)**
- **Distributed** and **scalable**
- Guarantees **high reliability** and **performance**

Technical implementation | Spring Boot

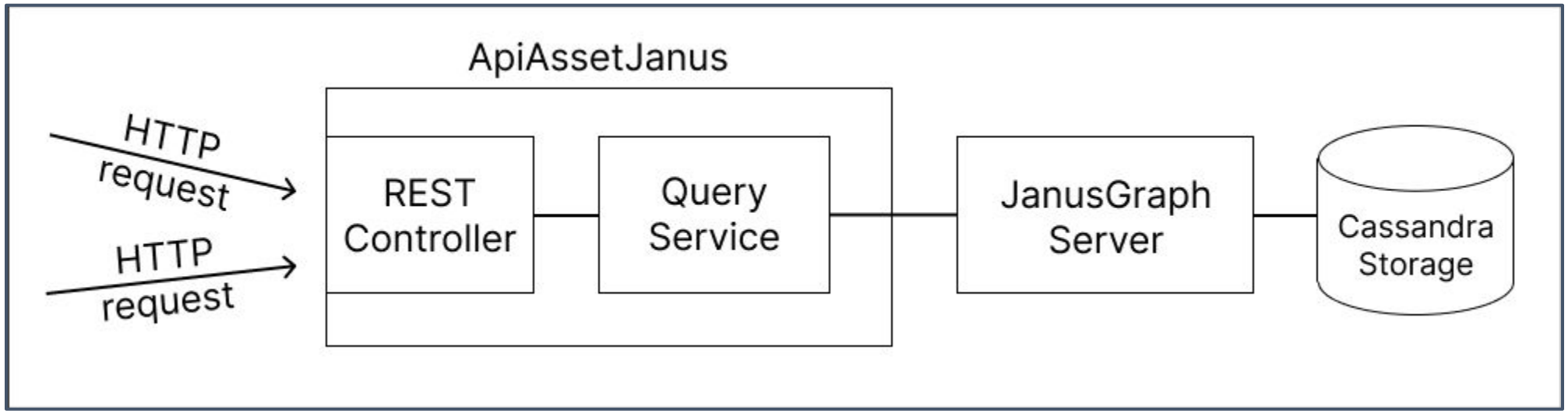
- Java framework to develop **web applications** and microservices
- Popular framework because it is focused on **speed**, **simplicity** and **productivity**, with a lot of the out of the box solutions
- It has been used in the project to develop **REST APIs**

Technical implementation | Steps

1. Installation of **JanusGraph** and **Cassandra**
2. Setting up JanusGraph to work with Cassandra
3. Creation of a Java application with the needed **dependencies**
4. **Code implementation** following an iterative approach
    a. **Writing** queries and REST APIs
    b. **Testing** with Postman

Technical implementation | Architecture

- **RESTController**: handles the incoming requests to the **QueryService**
- **QueryService:**
  - implements the code for executing queries
  - speaks directly to the JanusGraph server



Technical implementation | Queries

Gremlin-Java is considered the **reference implementation** of Gremlin

- allows interaction with a **Gremlin Server** (e.g. JanusGraph Server)
- implements Gremlin within the **Java** language (**same syntax**)
- allows **application-code** and **database-code** to be written in Java

```
public List<Map<Object, Object>> getVerticesByLabel(String label) {
        return this.g.V().hasLabel(label).elementMap().toList();
}
```

Achieved results

- Our software is capable of interacting with JanusGraph to
  - Run **CRUD operations**
  - Run complex **search operations**
  - Interact with a storage backend to **persist data**
  - Expose **REST API endpoints**

https://github.com/TBDM-Project/JanusGraph-API-example

Possible future improvements

- Implementation of an **index backend**
  - Improve search performance
  - One can add full-text search, geo search, …
  - e.g. **Elasticsearch**
- Improve the tool to work in a **real-world environment**

Thank you
for your attention!