CURSO DE PÓS-GRADUAÇÃO EM BIG DATA

TITAN:DB

UM BANCO DE DADOS TRANSACIONAL ORIENTADO A GRAFOS

Antonio Cavalcante de Paula Filho

Guilherme Moreira dos Reis

Luciano Bonfim de Azevedo

Rio de Janeiro - RJ

2016

RESUMO

Dados não estruturados geram muitos desafios e os bancos de dados baseados em grafos vêm se mostrando uma boa opção para modelagem de dados, principalmente em casos onde interconectividade e a escalabilidade são aspectos relevantes. Além de proporcionar uma representação de dados complexos, esse tipo de banco de dados tem a capacidade de executar consultas eficientes por eliminar o custo de operações de Junções, que são muito onerosas para os bancos relacionais. Este documento apresenta o TitanDB, distribuido no modelo open source, construído sobre os sistemas de gerenciamento de armazenamento de dados: cassandra, Hbase e berkeleyDB, conseguindo provê em tempo real, milhares de usuários simultâneos pesquisando sobre grafos de bilhões de arestas.

ABSTRACT

Unstructured data generate many challenges, and graph databases have shown to be a good option for data modeling, especially in cases where interconnectivity and scalability are important aspects. In addition to providing a complex data representation, this type of database has the ability to perform efficient queries because eliminates the cost of join operations, which can be quite expensive for relational databases. This document presents the TitanDB, distributed in open source model, built on the data storage backend systems: cassandra, Hbase and BerkeleyDB, which enables thousands of simultaneous users to search through graphs that features billions of edges.

ÍNDICE

1. INTRODUÇÃO E UM BREVE HISTÓRICO

O Titan é um sistema de banco de dados transacional, orientado a grafos, desenvolvido pela startup Thinkaurelius sob a licença Apache 2.

Como todo SGBD desse tipo, utiliza o conceito da teoria dos grafos na forma como organiza e armazena as informações.

Utiliza nós, arestas e propriedades para representar e armazenar as informações.

Esse tipo de estrutura permite uma modelagem versátil e intuitiva dos dados, possibilitando a captura e análise de estruturas relacionais ricas e complexas (BROECHELER, 2012).

Navathe (2005) diz que esses SGBDs são semelhantes aos implementados em árvores, porém, baseado na teoria dos grafos. Uma árvore é um grafo conexo (existe caminho entre quaisquer dois de seus vértices) e acíclico (não possui ciclos).

Navathe (2005) diz também que esse tipo de estrutura pode ser usada para o mapeamento de diversos problemas complexos, processos industriais, logística, sistemas de comunicação, fluxo de rede, etc.

Outros casos de uso que se beneficiam desses conceitos são: otimização de rotas, detecção de fraude, mecanismos de recomendação baseadas em redes sociais, otimização de anúncios, representação de conhecimento, cuidados relativos a saúde das pessoas, educação, segurança, etc.

O desenvolvimento do Titan foi um esforço contínuo de vários desenvolvedores, colaboradores e organizações.

Colaboradores:

- Matthias Broecheler: desenvolvedor líder e teórico.
- Dan LaRocque: desenvolvedor líder com foco especial nos adaptadores de armazenamento de backend e implantação em ambientes de nuvem.
- Marko A. Rodriguez: suporte ao TinkerPop, testes e documentação.
- Pavel Yaskevich: Desenvolvedor com grandes contribuições com adaptadores backend do Titan, Cassandra e HBase.
- Stephen Mallette: suporte ao TinkerPop, testes e documentação.
- Daniel Kuppitz: testes e feedback geral.
- Ketrina Yim: criador do logo do Titan.

Organizações:

- Aurelius: equipe de desenvolvimento e manutenção da base de código do Titan.

- Pearson: desenvolvimento de caso de uso, feedback dos clientes e pioneiro no uso da tecnologia.
- Intel: desenvolvimento de recursos individualmente.
- Digital Reasoning: desenvolvimento de recursos individualmente e teste de caso de uso.

O Titan começou a ser concebido em meados de 2011, quando a Pearson, uma empresa ligada a área da educação, contatou a Aurelius propondo que eles desenvolvessem uma solução para apoiar e melhorar os seus serviços de cursos online, aplicando a teoria dos grafos e a ciência de redes (RODRIGUEZ et al., 2013).

Nessa época, não havia no mercado tecnologia de banco de dados orientados a grafos, escalável e de código aberto. Nesse projeto, o Titan teve a responsabilidade de representar todas as universidades, estudantes, cursos, recursos, etc. Tudo em um simples e unificado grafo (RODRIGUEZ et al., 2013).

A figura 01 a seguir, mostra de forma simplificada o grafo do projeto da Pearson.

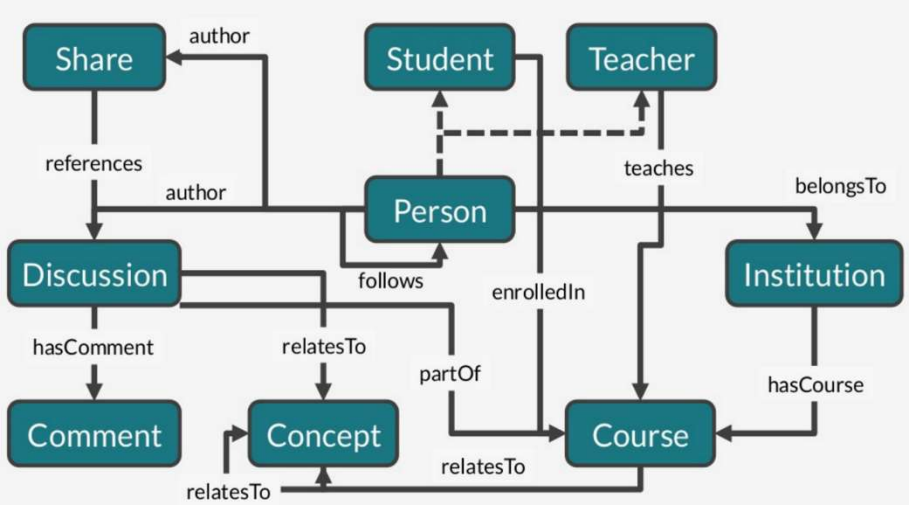

Figura 01: Grafo da Pearson (MATTHIAS, 2013)

Em fevereiro de 2015, o Titan foi adquirido pela DataStax, segundo Wolpe (2015), o grande valor na aquisição do Titan pela empresa não foi o banco de dados Titan em si, mas sim, os conhecimentos adquiridos a partir de sua engenharia no intuito de aplicá-los em novos projetos, como por exemplo, o DSE Graph, lançado em 12 de abril de 2016.

O DSE (DataStax Enterprise) foi desenvolvido a partir da tecnologia do Titan, como uma solução comercial de banco de dados orientados a grafos, com uma melhor escalabilidade e uma melhor integração com o Cassandra, que é disponibilizado também comercialmente pela empresa.

Além da Pearson, o Titan:db está sendo usado por: magazineluiza, Cisco, PayPal, Amazon entre outros (DIAS, 2016).

Segundo (FATALA et al., 2014), o Neo4J, concorrente do Titan, é mais fácil de utilizar, manter e de gerenciar, mas, apresenta alguns problemas, quando o banco começa a escalar muito, sua alta disponibilidade fica comprometida, pois é baseada em replicação master slave, isto é, tem um único ponto de falha.

Ainda segundo (FATALA et al., 2014), o Neo4J também não lida bem com o problema dos super nodes, que são vértices que tem tantas arestas conectadas, que começam a não caber mais na memória, as consultas ficam lentas e inviabilizam ordenação em cima das arestas.

Já o Titan escala muito fácil, utiliza replicação master master, através de uma camada de armazenamento, tem facilidades para se trabalhar com super nodes e possui inúmeras configurações para ajuste de desempenho (TITAN:DB).

Índices centrado nos vértices (Vertex-centric), fornecem consultas em nível de vértice, aliviando o problema do super nó. Cada vértice tem seu próprio índice.

2. CAPACIDADES E BENEFÍCIOS (TITAN:DB)

Um grafo Titan (TITAN:DB) pode ser escalável através de vários servidores e dispositivos de armazenamento diferentes, para isso ele precisa de um sistema de gerenciamento de armazenamento de dados também escalar, como o Cassandra ou HBase, suportando processamento de grafos tão grandes, que exigem armazenamento e capacidades computacionais que vão além do que uma única máquina pode proporcionar.

Ele é capaz de lidar com aplicações com mais de 100 bilhões de nós e dezenas de milhares de usuários concorrentes executando consultas complexas, dando suporte à múltiplas transações simultâneas e processamento operacional.

Ele trabalha com particionamento e distribuição de dados em clusters de computadores, utiliza redundância de dados, replicando-os para contornar problemas de falhas em máquinas do cluster e também para ter um melhor desempenho (AHOMOLYA, 2014).

É compatível com o Tinkerpop, um framework, isto é, um conjunto de ferramentas para se trabalhar com grafos, que permite aos usuários modelar o seu domínio como um grafo, e analisá-lo.

A pilha do Tinkerpop fornece bibliotecas padrão e interfaces para diferentes fornecedores de banco de dados orientados a grafos (NONNEN, 2016), entre eles o Titan:db.

Os componentes do Tinkerpop são:

- **Blueprints API**: como se fosse um JDBC para grafos, ele define várias interfaces, algumas implementações e uma suíte de testes para quem está desenvolvendo um banco de dados orientado a grafo implementar;
- **Pipes**: estrutura de processamento de fluxo de dados, monta um pipeline para as queries;
- **Gremlin**: linguagem padrão para lidar com grafos, permitindo a navegação de forma simples pelos dados do grafo, o Gremlin é um shell Groovy que por sua vez é um super set do Java;
- **Frames**: trabalha como hibernate, mapeia objetos do grafo;
- **Furnace**: é um conjunto de algoritmos para lidar com grafos que pode ser aplicado se usar também o Frames.
- **Rexster**: permite você montar uma API rest completa para interagir com seu banco de dados sem mexer com código.

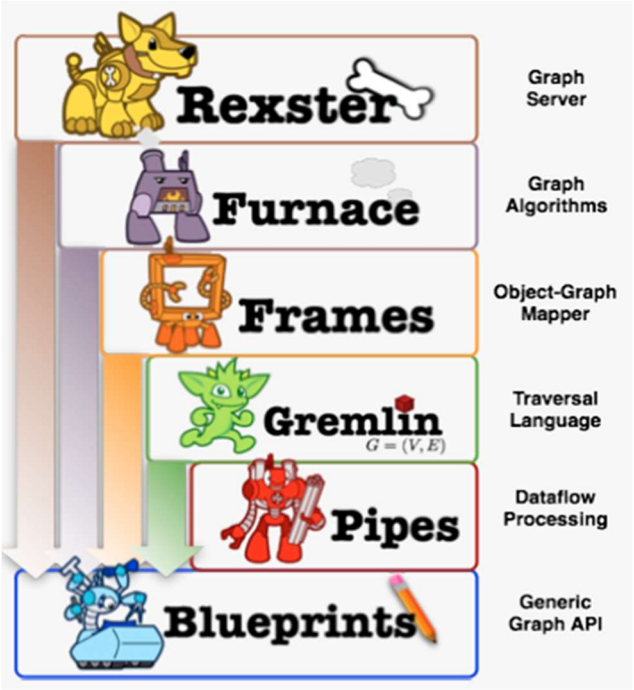

Figura 02: pilha do Tinkerpop

2.1. BENEFÍCIOS DO TITAN COM O CASSANDRA

- Continuamente disponível, não há pontos únicos de falha;
- Sem gargalos de read / write, não utiliza arquitetura master / slave;
- A escalabilidade elástica permite a introdução e remoção de máquinas;
- Camada de cache garante que os dados continuamente acessados estejam disponíveis na memória;
- Aumentar o tamanho do cache, adicionando mais máquinas ao cluster;
- Integração com o Hadoop;
- Open source sob a licença Apache 2.

2.2. BENEFÍCIOS DO TITAN COM O HBASE

- A forte integração com o ecossistema Hadoop;
- O suporte nativo para a consistência forte;
- Escalabilidade linear com a adição de mais máquinas;
- Leituras e escritas rigorosamente consistentes;
- Classes base conveniente para apoiar os trabalhos Hadoop MapReduce com tabelas do HBase;
- Suporte para exportar métricas via JMX;

- Open source sob a licença Apache 2.

2.3. TITAN E O TEOREMA CAP

Para conseguir melhor desempenho do sistema, alta escalabilidade, alta disponibilidade e tolerância a falhas, abre-se mão de uma das três qualidades mais importantes que um banco de dados deve ter, só é possível garantir duas e será necessário fazer uma escolha, esta decisão deverá ser baseada em requisitos de negócios.

Por isso, em qualquer projeto de banco de dados, o teorema CAP deve ser cuidadosamente considerado (C = Consistência, A = Disponibilidade, P = Partitionabilidade).

O Titan é distribuído com 3 ferramentas de apoio, são as engines de armazenamento, elas garantem duas dessas qualidades: Cassandra, HBase e BerkeleyDB. As suas vantagens e desvantagens no que diz respeito ao teorema CAP são representados no diagrama abaixo. O BerkeleyDB é um banco de dados não-distribuído e, como tal, é normalmente usado com o Titan apenas para fins de teste e de exploração.

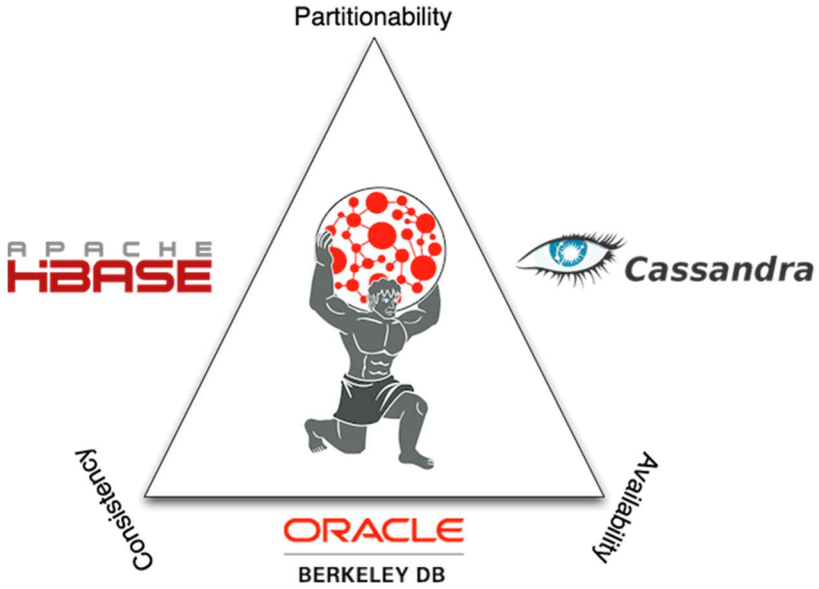

Figura 03: Diagrama do teorema CAP

O HBase dá preferência a consistência e particionamento em relação a disponibilidade, Cassandra dá preferência a disponibilidade e particionamento e o Oracle Berkeley DB a consistência e a disponibilidade, mas, sem particionamento.

3. ARQUITETURA (TITAN:DB)

O Titan está focado em: serialização compacta de grafos, modelagem de dados em grafos e execução de consulta eficiente (TITAN:DB).

O foco do Titan não está em coisas como: replicação, backup e snap shots, porque tudo isso é tratado pelo storagebackend (Cassandra, HBase...) (BROECHELER, 2013).

Entre o Titan e os discos existem um ou mais adaptadores de armazenamento e indexação. Como dito antes, o Titan vem com (TITAN:DB):

- Armazenamento de dados
    - Cassandra
    - HBase
    - BerkeleyDB
- Mecanismos de indexação que aceleram e permitem consultas mais complexas
    - Elasticsearch
    - Lucene

A arquitetura modular do Titan suporta também adaptadores de terceiros.

Em termos gerais, os aplicativos podem interagir com Titan de duas maneiras:

- Titan embarcado dentro da aplicação, executando consultas com Gremlin diretamente no grafo na mesma JVM. Execução de consulta, caches do Titan e manipulação de transações, tudo acontece na mesma JVM da aplicação, enquanto a recuperação de dados a partir do armazenamento backend pode ser local ou remoto;
- Interagir com uma instância local ou remota do Titan, através da solicitação de consultas Gremlin para o servidor. O Titan suporta nativamente os componentes servidores Gremlin da pilha de ferramentas Tinkerpop.

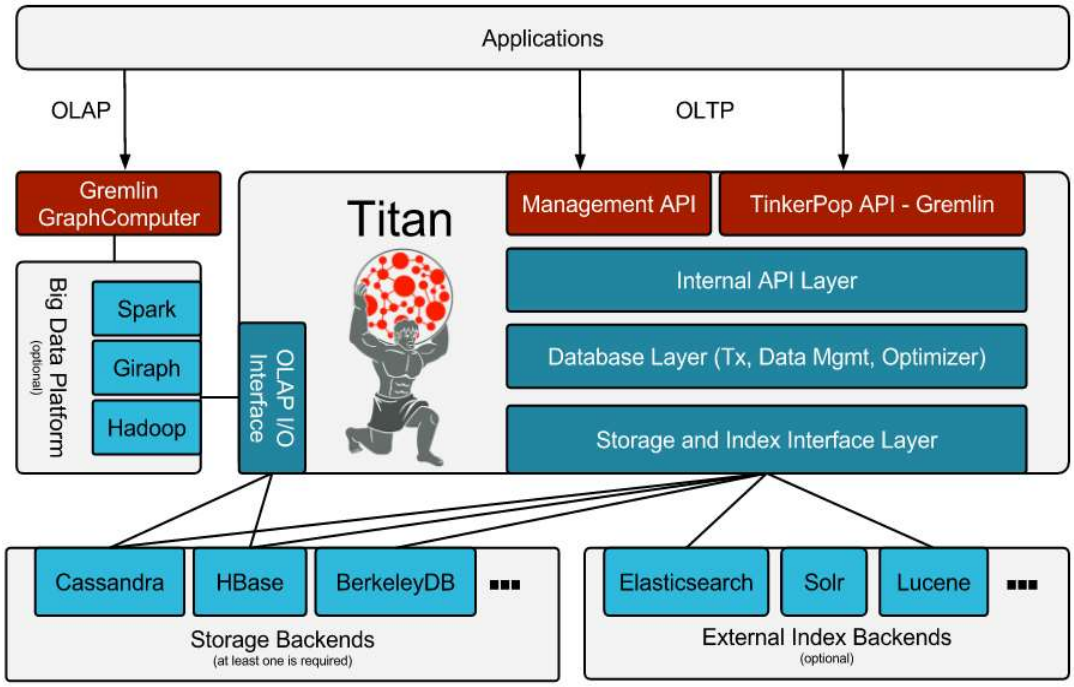

Figura 04: Diagrama da arquitetura do Titan (TITAN:DB)

Os dados dos grafos no Titan são armazenados em lista de adjacência, o que significa que os dados são armazenados como uma coleção de vértices junto com a lista de adjacência de cada vértice. A lista de adjacência de um vértice contém todas as arestas incidentes no vértice (e as propriedades).

Ao armazenar um grafo em formato de lista de adjacência, o Titan garante que todas as arestas incidentes em um vértice e suas propriedades sejam armazenadas de forma compacta no backend de armazenamento, o que acelera as consultas. A desvantagem é que cada aresta deve ser armazenada duas vezes - uma vez para cada vértice nas extremidades da aresta.

O Titan armazena a lista de adjacência de um grafo no formato Big Table, onde cada tabela é uma coleção de linhas, cada linha identificada por uma chave, cada linha formada por um número arbitrário (grande, mas limitado) de células. Uma célula é composta por uma coluna e o seu valor. Uma célula é identificada de forma única por uma coluna dentro de uma dada linha. Linhas no modelo BigTable são chamadas "wide rows", porque suportam um grande número de células e as colunas dessas células não têm que ser definidas antes, como é exigido nas bases de dados relacionais.

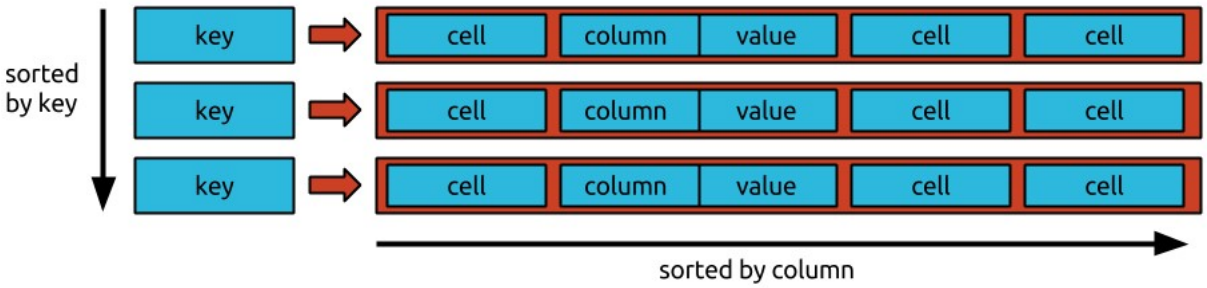

Figura 05: modelo big table de dados (TITAN:DB)

O Titan tem uma exigência adicional para o modelo de dados BigTable, as células devem ser classificadas por suas colunas e um subconjunto de células especificadas por um intervalo de coluna deve ser eficientemente recuperáveis (por exemplo, usando estruturas de índice, skip lists, ou busca binária).

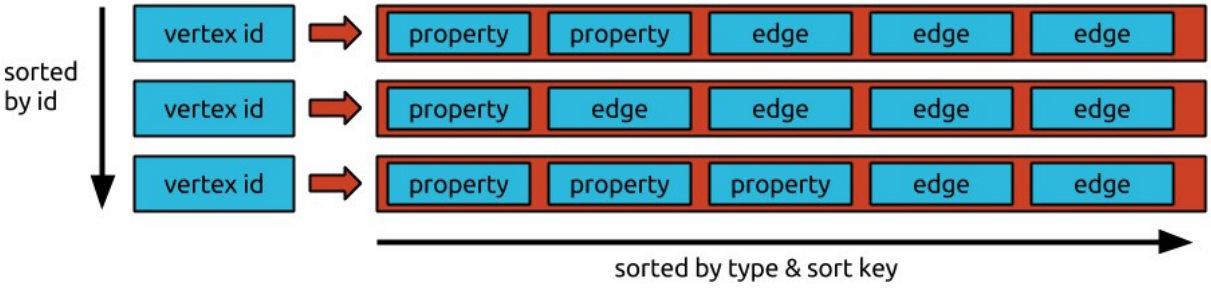

Figura 06: modelo Titan:DB de dados (TITAN:DB)

4. INSTALAÇÃO

Titan pode ser baixado na seção de downloads do repositório do projeto, no endereço:

https://github.com/thinkaurelius/titan/wiki/Downloads

A versão titan-1.0.0-hadoop1.zip é a recomendada segundo o site oficial,

Titan requer Java 8 (Standard Edition).

Instalando o Titan:db no Windows:

1. Descompactar o arquivo titan-1.0.0-hadoop1.zip.
   Suponha para o exemplo, a descompactação no C:\
2. Entre na pasta do Titan com o comando:
   cd C:\ titan-1.0.0-hadoop1\
3. Executar então o gremlin shell com o comando:
   .\bin\gremlin.bat
4. Para sair:
   Ctrl d

Instalando o Titan:db no GNU/Linux:

1. Unzipar o arquivo titan-1.0.0-hadoop1.zip, suponha que tenha sido descompactado no /home/usuario/Downloads/
2. Entrar na pasta bin do Titan, com o comando:
   cd /home/usuario/Downloads/titan-1.0.0-hadoop1
3. Executar então o gremlin shell com o comando:
   ./bin/gremlin.sh
4. Para sair:
   Ctrl d

5. LINGUAGEM DE CONSULTA GREMLIN

Gremlin é uma DSL (Linguagem de domínio específico), especializada em transitar em grafos (FATALA et al., 2014), construída em Groovy, que é um super set do Java, orientada a caminho, ele permite expressar sucintamente operações e caminhos em grafos complexos.

Gremlin é independente do Titan e suporta a maioria dos bancos de dados orientado grafo. Construir aplicações em cima do Titan através do uso do Gremlin, evita ficar preso a um fabricante, porque a sua aplicação pode ser migrada para outros bancos de dados orientados a grafo suportados pelo Gremlin, reaproveitando scripts, recomendações, etc. Que já existam (FATALA et al., 2014).

5.1. CARREGANDO O GRAFO DOS DEUSES (TITAN:DB)

O grafo da figura a seguir, mostra o exemplo que vem com o Titan:DB, o grafo dos deuses.

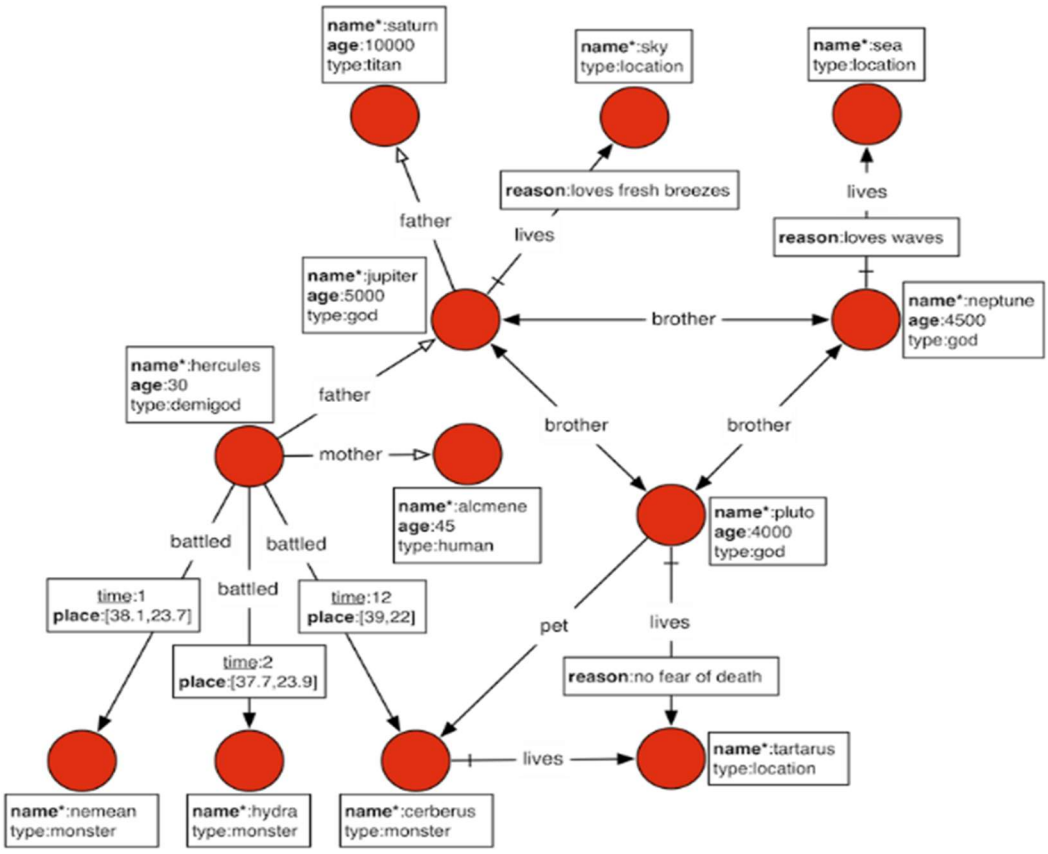

Figura 07: Grafo dos Deuses (TITAN:DB)

Para carregar o grafo dos deuses são usados os comandos:

//Instancia o grafo passando a configuração do berkeleydb como parâmetro.

gremlin> graph = TitanFactory.open('conf/titan-berkeleyje-es.properties')

==>standardtitangraph[berkeleyje:../db/berkeley]

//Cria o grafo recém instanciado no storagebackend

```
gremlin> GraphOfTheGodsFactory.load(graph)
==>null
//Visita exatamente uma vez cada vértice no grafo, verificando e/ou atualizando cada vértice
gremlin> g = graph.traversal()
==>graphtraversalsource[standardtitangraph[berkeleyje:../db/berkeley], standard]
```

Alguns comandos básicos:

```
//Retorna a quantidade de vértices
gremlin> g.V( ).count( )
==>12

//Retorna a quantidade de arestas
gremlin> g.E( ).count()
==>17
```

O código abaixo, associa à variável saturn, as propriedades do vértice que lhe representa, retornando seu índice:

```
gremlin> saturn = g.V().has('name', 'saturn').next()
==>v[256]
```

Nesse comando abaixo, o valueMap( ) retorna as propriedades do elemento saturn:

```
gremlin> g.V(saturn).valueMap()
==>[name:[saturn], age:[10000]]
```

A próxima consulta recupera o neto de satun. A aresta do filho de saturn (jupiter), aponta para ele (saturn), assim como a aresta de hercules, neto de saturn, aponta para o pai dele, Júpiter.

```
gremlin> g.V(saturn).in('father').in('father').values('name')
==>hercules
```

Veja a figura:

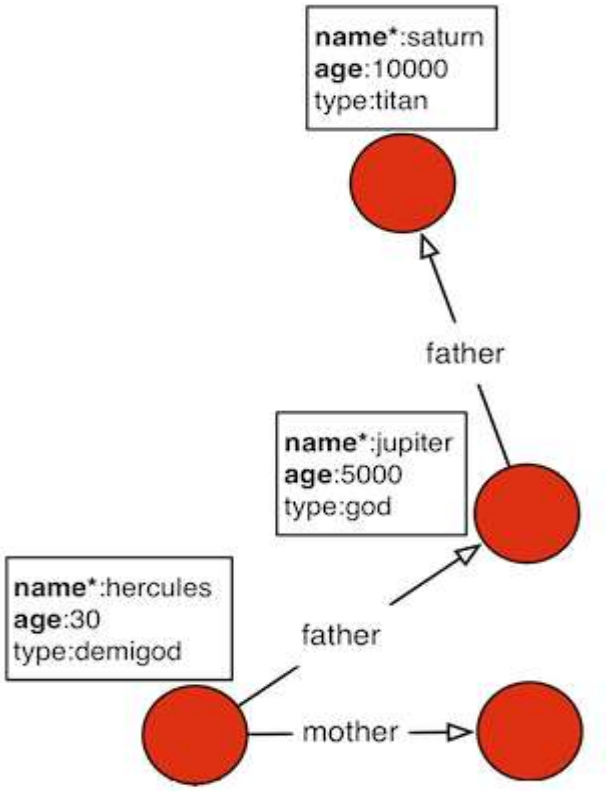

Figura 08: Grafo dos Deuses relações (TITAN:DB)

A propriedade Place é também um índice no grafo, ela é uma propriedade de aresta.

É possível buscar no grafo dos deuses por todos os eventos que aconteceram dentro de 50km de Atenas, latitude 37.97 e longitude 23.72

gremlin> g.E().has('place', geoWithin(Geoshape.circle(37.97, 23.72, 50)))

Retorna os vértices envolvidos nesses eventos:

==>e[6qm-9o0-9hx-36w][12528-battled->4136]

==>e[74u-9o0-9hx-3bc][12528-battled->4296]

Vimos anteriormente, que o neto de Saturno era Hércules. Isso pode ser expresso através de um loop. Em essência, Hercules é o vértice que está a 2 passos de distância de Saturno ao longo do caminho com arestas father ("pai").

gremlin> hercules = g.V(saturn).repeat(__.in('father')).times(2).next()

==>v[1536]

Retorna o índice do vértice hercules e atribui à variável hercules.

Hercules é um semideus. Para provar que Hercules é metade humano e metade deus, as origens de seus pais devem ser examinadas. É possível percorrer a partir do vértice hercules a sua mãe e pai. Finalmente, é possível determinar o tipo de cada um deles. Produzindo "god" e "human."

gremlin> g.V(hercules).out('father', 'mother')

```
==>v[1024]
==>v[1792]
gremlin> g.V(hercules).out('father', 'mother').values('name')
==>jupiter
==>alcmene
gremlin> g.V(hercules).out('father', 'mother').label()
==>god
==>human
gremlin> hercules.label()
==>demigod
```

Os exemplos até agora têm sido em relação aos ascendentes dos diversos personagens no panteão romano. O Modelo de propriedade do grafo é expressivo o suficiente para representar vários tipos de coisas e relacionamentos. Desta forma, o grafo dos deuses também identifica várias façanhas heroicas de Hércules, ele estava envolvido em duas batalhas próximas a Atenas. É possível explorar esses eventos atravessando arestas battled (batalha) que sai do vértice hercules.

```
gremlin> g.V(hercules).out('battled')
==>v[2304]
==>v[2560]
==>v[2816]
gremlin> g.V(hercules).out('battled').valueMap()
==>[name:[nemean]]
==>[name:[hydra]]
==>[name:[cerberus]]
gremlin> g.V(hercules).outE('battled').has('time', gt(1)).inV().values('name')
==>cerberus
==>hydra
```

A propriedade de aresta time nas arestas battled é indexada pelos índices centrado no vértice. Recuperar as arestas battled incidentes em hercules, de acordo com a restrição do filtro time, é mais rápido do que fazer uma varredura linear de todas as arestas e filtrar. (Tipicamente O (log n), onde n é o número de arestas incidentes). Titan é suficientemente inteligente para usar índices centrado no vértice (vertex-centric), quando disponível. O toString ( ) de uma expressão Gremlin mostra uma decomposição em etapas individuais.

```
gremlin> g.V(hercules).outE('battled').has('time', gt(1)).inV().values('name').toString()
```

==>[GraphStep([v[24744]],vertex), VertexStep(OUT,[battled],edge), HasStep([time.gt(1)]), EdgeVertexStep(IN), PropertiesStep([name],value)]

Uma consulta Gremlin é uma cadeia de operações / funções que são avaliadas da esquerda para a direita. Uma consulta para saber quem é o avô de hercules é mostrada abaixo sobre o grafo do conjunto de dados Deuses.

```
gremlin> g.V( ).has('name', 'hercules').out('father').out('father').values('name')
==>saturn
```

A consulta pode ser lida:

- g: o grafo atual que será atravessado pelo Gremlin.
- V: para todos os vértices no grafo.
- has('name', 'Hercules'): filtra os vértices para obter aquele com nome da propriedade "Hercules" (há apenas um).
- out('father'): atravessa a aresta pai que sai de Hércules.
- out('father'): atravessa a aresta pai que sai do vértice do pai de Hércules (Júpiter).
- name: obtém a propriedade nome do vértice do avô de "Hercules".

Agora o grafo será criado com o Cassandra de backend de armazenamento e o elasticsearch como index backend, já que no exemplo anterior, foi usado o BerkeleyDB.

Primeiro deve-se iniciar o Cassandra, para esse exemplo, será usado o apache-cassandra-3.7, na pasta, /home/vagrant/apache-cassandra-3.7:

Entra na pasta do cassandra

```
cd /home/vagrant/apache-cassandra-3.7
//Inicia o cassandra
./bin/cassandra -f &
//habilita explicitamente o thrift para que o Titan possa conectar
./bin/nodetool enablethrift
//Inicia o elasticsearch
./bin/elasticsearch -f &
//Local Server Mode
//Cria um grafo vazio do Titan com o cassandra e elasticsearch
gremlin> TitanGraph g = TitanFactory.build().
set("storage.backend", "cassandra").
set("storage.hostname", "127.0.0.1").
open();
```

```
==>standardtitangraph[cassandra:[127.0.0.1]]
```

Ou para abrir o grafo dos deuses

```
gremlin> graph = TitanFactory.open('conf/titan-cassandra-es.properties')
==>standardtitangraph[cassandrathrift:[127.0.0.1]]
gremlin> GraphOfTheGodsFactory.load(graph)
==>null
gremlin> g = graph.traversal()
==>graphtraversalsource[standardtitangraph[cassandrathrift:[127.0.0.1]], standard]
```

Entra no grafo dos deuses pelo vértice saturn

```
gremlin> saturn = g.V().has('name', 'saturn').next()
==>v[4112]
```

Continuando a explorar o grafo dos deuses

```
==>graphtraversalsource[titangraph[cassandrathrift:127.0.0.1], standard]
gremlin> g.V().has('name', 'hercules')
==>v[24]
gremlin> g.V().has('name', 'hercules').out('father')
==>v[16]
gremlin> g.V().has('name', 'hercules').out('father').out('father')
==>v[20]
gremlin> g.V().has('name', 'hercules').out('father').out('father').values('name')
==>saturn
```

5.2. GREMLIN VS SQL

Abrindo e carregando o grafo clássico que vem com o tinkerpop, veja o diagrama:

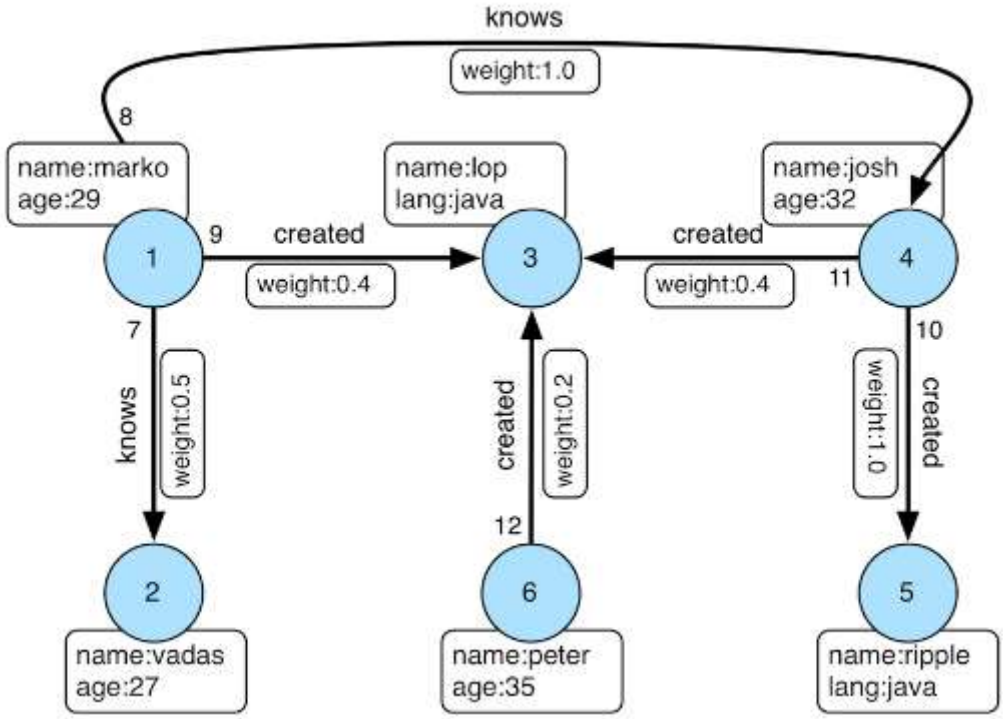

Figura 09: Grafo classic do tinkerpop

Para abrir e criar:

```
//Cria o grafo padrão do tinkerpop
gremlin> g = TinkerFactory.createClassic()

==>tinkergraph[vertices:6 edges:6]

// Atravessa o grafo lendo e atualizando
gremlin> grafo = g.traversal( )

==>graphtraversalsource[tinkergraph[vertices:6 edges:6], standard]

//Retorna todos os vértices
gremlin> grafo.V( )

==>v[1]

==>v[2]

==>v[3]

==>v[4]

==>v[5]

==>v[6]
```

```
gremlin> marko = grafo.V().has('name', 'marko').next()
==>v[1]
//A consulta abaixo equivale a: select * from users
gremlin> grafo.V().valueMap()
==>[name:[marko], age:[29]]
==>[name:[vadas], age:[27]]
==>[name:[lop], lang:[java]]
==>[name:[josh], age:[32]]
==>[name:[ripple], lang:[java]]
==>[name:[peter], age:[35]]
```

Se quiser pegar especificamente o primeiro vértice:

```
gremlin> grafo.V(1).valueMap( )
==>[name:[marko], age:[29]]
//Pegando só o nome do primeiro vértice
gremlin> grafo.V(1).valueMap('name')
==>[name:[marko]]
//Pegando todos os nomes (select name from tbl_users)
gremlin> grafo.V( ).valueMap('name')
==>[name:[marko]]
==>[name:[vadas]]
==>[name:[lop]]
==>[name:[josh]]
==>[name:[ripple]]
==>[name:[peter]]')
//Pegando todos os nomes e idades (select name, age from tbl_users)
gremlin> grafo.V().valueMap('name', 'age')
```

==>[name:[marko], age:[29]]

==>[name:[vadas], age:[27]]

==>[name:[lop]]

==>[name:[josh], age:[32]]

==>[name:[ripple]]

==>[name:[peter], age:[35]]

//Removendo o vértice de índice 1 (marko)
gremlin> grafo.V(1).drop()

//Adiciona de volta o vértice removido
gremlin> g.addVertex(T.label, "person", T.id, 1, "name", "marko", "age", 29)

Tabela 01: SQL vs Gremlin

QUERY	SQL	GREMLIN
Todos os usuários	select * from tbl_users	grafo.V().hasLabel().valueMap( )
O nome de todos os usuários	select name from tbl_users	grafo.V().valueMap('name')
Nome e idade de todos	select name, age from users	grafo.V().valueMap('name', 'age')
Com distinct	select distinct (lang) From tbl_users	grafo.V().values('lang').dedup()
Usuário mais velho	select max(age) from users	grafo.V().values('age').max()
Select por igualdade	select * from tbl_users where age = 35	grafo.V().has('age', 35).valueMap()
Select por comparação	select * from tbl_users where age > 21	grafo.V().has('age', gt(21)).valueMap()

6. TITAN SERVER (TITAN:DB)

Titan utiliza o gremlin server como o componente de servidor para processar e responder às consultas clientes.

Gremlin server fornece uma maneira de executar remotamente scripts Gremlin em uma ou mais instâncias do Titan. Por padrão, os aplicativos clientes podem se conectar a ele através de WebSockets com um subprotocolo personalizado (há um número de clientes desenvolvidos em diferentes linguagens para ajudar a suportar o subprotocolo).

6.1. INICIANDO O TITAN

```
$ ./bin/titan.sh start
Forking Cassandra...
Running `nodetool statusthrift`.. OK (returned exit status 0 and printed string "running").
Forking Elasticsearch...
Connecting to Elasticsearch (127.0.0.1:9300)... OK (connected to 127.0.0.1:9300).
Forking Gremlin-Server...
Connecting to Gremlin-Server (127.0.0.1:8182)... OK (connected to 127.0.0.1:8182).
Run gremlin.sh to connect.
```

6.2. CONECTANDO COM O GREMLIN SERVER

O servidor Gremlin estará pronto para escutar conexões WebSocket quando iniciado. Uma forma fácil de testar a conexão é com o console do Gremlin.

Abaixo o comando bin/gremlin.sh inicia o console Gremlin, depois o :remote para conectar e por último o comando :> que é o submit.

```
$  bin/gremlin.sh
         \,,,/
         (o o)
-----oOOo-(3)-oOOo-----
plugin activated: tinkerpop.server
plugin activated: tinkerpop.hadoop
plugin activated: tinkerpop.utilities
plugin activated: aurelius.titan
plugin activated: tinkerpop.tinkergraph
gremlin> :remote connect tinkerpop.server conf/remote.yaml
==>Connected - localhost/127.0.0.1:8182
gremlin> :> graph.addVertex("name", "stephen")
```

```
==>v[256]
gremlin> :> g.V().values('name')
==>stephen
```

O comando :remote diz para o console configurar uma conexão remota com o Gremlin server usando o arquivo conf/remote.yaml para conectar. Esse arquivo aponta para uma instância do Gremlin sendo executada em localhost. O comando :> é o submit, que envia Gremlin nessa linha, para o atual remoto ativo.

Dica:

Para iniciar o Titan Server com a API REST encontre o arquivo conf/gremlin-server/gremlin-server.yaml na distribuição e edite-o. Modifique a configuração do channelizer para org.apache.tinkerpop.gremlin.server.channel.HttpChannelizer e em seguida inicie o Titan server.

7. CONFIGURAÇÕES (TITAN:DB)

Um cluster de um grafo do Titan consiste de uma ou múltiplas instâncias do Titan.

Para abrir uma instância do Titan, uma configuração tem que ser especificada.

Uma configuração do Titan especifica quais componentes o Titan pode utilizar, controla todos os aspectos operacionais da implantação do Titan e fornece algumas opções de tuning para conseguir um melhor desempenho do cluster.

No mínimo, uma configuração deve definir o mecanismo (engine) de persistência que o Titan deve usar como armazenamento backend.

Exemplo de configuração:

7.1. CASSANDRA

O código abaixo configura o Titan para usar o cassandra como engine de persistência rodando localmente e o sistema de indexação elasticsearch rodando remotamente:

```
storage.backend=cassandra
storage.hostname=localhost
index.search.backend=elasticsearch
index.search.hostname=100.100.101.1, 100.100.101.2
index.search.elasticsearch.client-only=true
graph = TitanFactory.open("conf/titan-cassandra.properties")
```

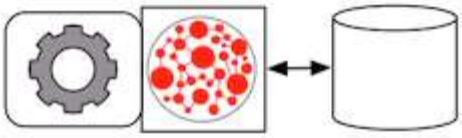

Figura 10: servidor local

Quando o grafo precisa ser escalado para além dos limites de uma única máquina, então o Cassandra e o Titan provavelmente estarão logicamente separados em diferentes máquinas. O cluster Cassandra mantém a representação do grafo e qualquer número de instâncias do Titan para o acesso de leitura / gravação via socket para o cluster do Cassandra.

Por exemplo, suponha que temos um cluster Cassandra rodando, onde uma das máquinas tem o endereço IP 77.77.77.77, outra 77.77.77.78, então, a conexão do Titan com o cluster é realizado da seguinte forma (endereços IP são separados por vírgula para fazer referência a mais de uma máquina):

```
TitanGraph graph = TitanFactory.build().
set("storage.backend", "cassandra").
set("storage.hostname", "77.77.77.77, 77.77.77.78")
open()
```

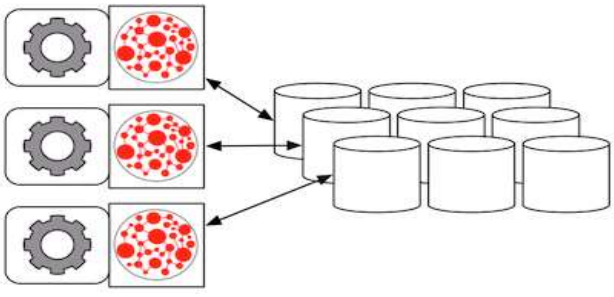

Figura 11: servidor remoto

O servidor Gremlin pode ser atrelado a cada instância Titan definida no código anterior. Desta forma, a aplicação do usuário final não precisa ser um aplicativo baseado em Java, pois ele pode se comunicar com Gremlin Server como um cliente. Esse tipo de implantação é ótimo para arquiteturas poliglotas, onde vários componentes escritos em linguagens diferentes precisam refereciar e fazer cálculos no grafo.

Para iniciar o servidor gremlin é usado o comando:

```
./bin/gremlin-server.sh
```

E

ntão, em uma sessão gremlin.sh externa, poderá ser enviado comandos gremlin através do cluster:

```
:plugin use tinkerpop.server
:remote connect tinkerpop.server conf/remote.yaml
:> g.addV()
```

Neste caso, cada servidor Gremlin seria configurado para se conectar ao cluster Cassandra. O código logo abaixo mostra o fragmento específico da configuração do servidor Gremlin.

```
...
graphs: {
  g: conf/titan-cassandra.properties}
plugins:
  - aurelius.titan
...
```

Finalmente, Cassandra pode ser incorporado no Titan, o que significa que o Titan e o Cassandra são executados na mesma JVM e comunicam-se via processo de chamada e não através da rede. Isso remove a (de) serialização e protocolo de rede, levando a melhorias consideráveis de desempenho. Neste modo de implantação, o Titan internamente inicia um daemon Cassandra e o Titan não se conecta a um cluster existente, mas, ele é o seu próprio cluster.

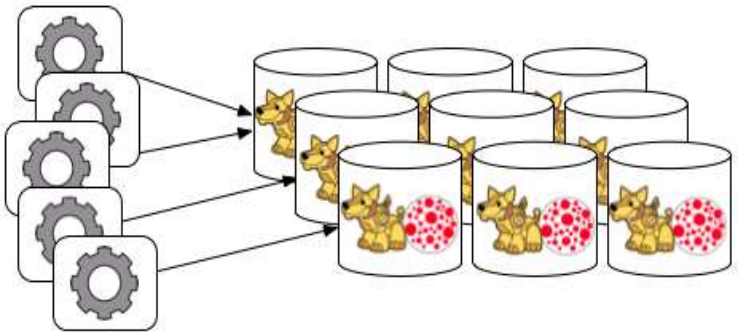

Figura 12: embarcado (embedded)

Para usar Titan no modo embarcado, basta configurar embeddedcassandra como o backend de armazenamento. As opções de configuração listados abaixo também se aplicam a Cassandra incorporado. Ao executar Titan no modo embarcado, o arquivo cassandra.yaml é configurado usando a opção de configuração adicional storage.conf-file, que especifica o arquivo yaml como uma URL completa, por exemplo:

storage.conf-file = file:///home/cassandra.yaml

Ao configurar o Cassandra, recomenda-se considerar as seguintes opções de configuração específica do Cassandra:

- read-consistency-level: nível de consistência do Cassandra para operações de leitura
- write-consistency-level: nível de consistência do Cassandra para operações de gravação
- replication-factor: fator de replicação usado (valor recomendado é 3)
- thrift.frame_size_mb: O tamanho máximo a ser usado pelo thrift. Aumentar este valor ao recuperar conjuntos muito grandes de resultados. Aplicável apenas quando storage.backend = cassandrathrift
- keyspace: O nome do keyspace para armazenar o grafo Titan. Permite que vários grafos Titan coexistam no mesmo cluster Cassandra.

7.2. HBASE

Configurando o Titan para usar o mecanismo de persistência HBase funcionando remotamente e usa componente de caching do Titan para um melhor desempenho.

storage.backend=hbase

storage.hostname=100.100.101.1

storage.port=2181

cache.db-cache = true

cache.db-cache-clean-wait = 20

```
cache.db-cache-time = 180000
cache.db-cache-size = 0.5
```

HBase pode ser executado como um banco de dados autônomo no mesmo localhost. Neste modelo, Titan e HBase comunicam-se através de um socket localhost. Executar o Titan sobre o HBase requer as seguintes etapas:

Para iniciar o HBase é usado:

```
./bin/start-hbase.sh
```

Para parar o HBase é usado:

```
stop-hbase.sh
```

Com o HBase rodando, agora é possível criar um grafo Titan com HBase:

```
TitanGraph graph = TitanFactory.build()
  .set("storage.backend", "hbase")
  .open();
```

Note que não precisou especificar um nome de host, a conexão localhost é tentada por padrão.

Para usar o Titan com HBase de forma escalar, em modo Remote Server, é usado:

```
TitanGraph g = TitanFactory.build()
  .set("storage.backend", "hbase")
  .set("storage.hostname", "77.77.77.77, 77.77.77.78, 77.77.77.79")
  .open();
```

Em modo Remote Serve com o Gremlin Server:

```
http://rexster.titan.machine1/mygraph/vertices/1
http://rexster.titan.machine2/mygraph/tp/gremlin?script=g.v(1).out('follows').out('created')
```

Neste caso, cada servidor Gremlin seria configurado para se conectar ao cluster HBase. O seguinte fragmento mostra a configuração do servidor Gremlin.

```
...
graphs: {
  g: conf/titan-hbase.properties}
plugins:
  - aurelius.titan
...
```

7.3. BERKELEYDB

Configura o Titan para usar o BerkeleyDB como o engine de persistência embutido e o elasticsearch como sistema de indexação embutido.

```
storage.backend=berkeleyje
storage.directory=/tmp/graph
index.search.backend=elasticsearch
index.search.directory=/tmp/searchindex
index.search.elasticsearch.client-only=false
index.search.elasticsearch.local-mode=true
```

O Berkeley DB é executado na mesma JVM com o Titan, por isso, requer apenas uma configuração simples e nenhuma configuração adicional:

```
TitanGraph g = TitanFactory.build().
set("storage.backend", "berkeleyje").
set("storage.directory", "/tmp/graph").
open();
```

Se um cluster de um grafo do Titan foi configurado previamente e/ou apenas o armazenamento backend precise ser definido, o TitanFactory( ) aceita uma string representando o nome do backend e o hostname ou diretório:

```
graph = TitanFactory.open('cassandra:localhost')
graph = TitanFactory.open('berkeleyje:/tmp/graph')
```

7.4. TITAN SERVER

Para interagir com o Titan remotamente ou em outro processo através de um cliente, o servidor Titan precisa ser configurado e iniciado.

Internamente, o Titan usa o GremlinServer da pilha TinkerPop para serviços de solicitações clientes e, portanto, a configuração é feita através do arquivo de configuração do GremlinServer.

Para configurar o GremlinServer como uma instância de um grafo Titan, o arquivo de configuração do GremlinServer requer a seguinte especificação:

```
...
graphs: {
  graph: conf/titan-berkeleyje.properties
}
plugins:
```

- aurelius.titan

...

Graphs: define a ligação para uma configuração específica do TitanGraph, no caso acima, a configuração liga o grafo a uma configuração do BerkeleyDB (conf/titan-berkeleyje.properties).

Isso significa que quando referenciamos o TitanGraph em um contexto remoto, esse grafo pode ser referenciado por g no script enviado para o servidor.

Plugin: habilita o plugin Gremlin, o qual permite auto importar classes do Titan, assim, podem ser referenciadas em scripts submetidos remotamente.

7.5. CONFIGURAÇÃO GLOBAL

O Titan tem duas opções de configurações distintas, a Local e a Global.

A Local é aplicada a uma instância individual do Titan.

A Global é aplicada a todas as instâncias em um cluster.

O Titan tem cinco escopos de opções de configuração.

- LOCAL: aplicada a uma instância individual do Titan e são especificadas na configuração fornecida na inicialização da instância do Titan
- MASKABLE: essa opção de configuração pode ser sobrescrita por uma instância individual do Titan pelo arquivo de configuração local. Se o arquivo de configuração local não especificar a opção, seu valor é lido da configuração global do cluster Titan
- GLOBAL: essa opção é sempre lida da configuração do cluster e não podem ser sobrescrita por uma instância básica
- GLOBAL_OFFLINE: como a GLOBAL, mas, mudar essa opção requer a reinicialização do cluster para assegurar que o valor é o mesmo através de todo o cluster;
- FIXED: como a GLOBAL, mas, o valor não pode ser mudado depois que o cluster tiver inicializado

Quando a primeira instância de um cluster Titan é iniciada, as opções de configuração global são inicializadas a partir do arquivo de configuração local fornecido. Posteriormente, alterar as opções de configuração global, poderá ser feita através da API de gerenciamento do Titan. Para acessar a API de gerenciamento, é necessário chamar g.getManagementSystem () em uma instância do Titan.

8. ESQUEMA E MODELAGEM DE DADOS (TITAN:DB)

Cada grafo Titan tem um esquema composto de label (rótulos) de aresta, propriedade chave e label de vértice.

8.1. DEFININDO LABELS DE ARESTAS

Cada aresta conectando dois vértices tem um label que define a semântica do relacionamento. Por exemplo, uma aresta com o rótulo amigo entre os vértices A e B, codifica uma amizade entre os dois indivíduos.

Para definir um label de aresta, é preciso chamar makeEdgeLabel (String) em um grafo já aberto ou gerenciamento de transação e fornecer o nome do rótulo da aresta como argumento.

A multiplicidade de um label de aresta define a restrição de multiplicidade em todas as arestas de determinado rótulo, isto é, o número máximo de arestas entre pares de vértices.

8.1.1. MULTIPLICIDADE DE ARESTA

Configurações de multiplicidade

MULTI: Permite múltiplas arestas do mesmo label entre qualquer par de vértices.

SIMPLE: Permite no máximo uma aresta de determinado rótulo entre qualquer par de vértices.

MANY2ONE: Permite no máximo uma aresta de saída de um label em qualquer vértice no grafo, mas não coloca nenhuma restrição sobre arestas de entrada.

ONE2MANY: Permite no máximo uma aresta de entrada de um label em qualquer vértice no grafo, mas não coloca nenhuma restrição sobre arestas de saída.

ONE2ONE: Permite no máximo uma aresta de entrada e uma aresta de saída de determinado label em qualquer vértice no grafo.

O código abaixo mostra como configurar multiplicidade de arestas.

```
mgmt = graph.openManagement()
follow = mgmt.makeEdgeLabel('follow').multiplicity(MULTI).make()
mother = mgmt.makeEdgeLabel('mother').multiplicity(MANY2ONE).make()
mgmt.commit( )
```

8.2. DEFININDO PROPRIEDADES

Propriedades em vértices e arestas são pares de chave-valor. Por exemplo, a propriedade nome = 'Daniel' tem a chave nome e o valor 'Daniel'. Propriedade chave-valor faz parte do esquema Titan e pode restringir os tipos de dados permitidos e a cardinalidade de valores.

8.1.2. TIPOS DE DADOS DE PROPRIEDADES CHAVE

A classe dataType é usada para definir o tipo de dado de uma propriedade chave. Titan irá garantir que todos os valores associados com a chave tenham o tipo de dado configurado e, assim, garante que o dado adicionado ao grafo seja válido. Por exemplo, pode-se definir que a chave nome tenha um tipo de dado String.

Titan suporta nativamente os seguintes tipos de dados.

- String - sequência de caracter
- Character - um caracter individual
- Boolean - true ou false
- Byte - valor em byte
- Short - valor em short
- Integer - valor em Integer
- Long - valor em Long
- Float - ponto flutuante de 4 byte
- Double - ponto flutuante de 8 byte
- Decimal - número com três dígitos decimais
- Precision - número com seis dígitos decimais
- Date – data
- Geoshape - Geographic shape like ponto, círculo ou retângulo (box)
- UUID - UUID (identificador único universal)

8.1.3. CARDINALIDADE DE PROPRIEDADE CHAVE

Cardinality é usada para definir a cardinalidade permitida dos valores associados com a chave em qualquer vértice.

Configurações de cardinalidade:

- SINGLE: Permite no máximo um valor por elemento para essa chave. Em outras palavras, o mapeamento chave → valor é único para todos os elementos no grafo. A propriedade chave birthDate é um exemplo de cardinalidade SINGLE, já que cada pessoa tem exatamente uma data de nascimento
- LIST: Permite um número arbitrário de valores por elemento para essa chave. Em outras palavras, a chave é associada com uma lista de valores, permitindo valores duplicados. Modelar sensores como vértices em um grafo, a propriedade chave

sensorReading seria um exemplo com cardinalidade LIST, permitindo que lotes de sensores de leitura (potencialmente duplicados) sejam gravados

- SET: Permite vários valores, mas sem valores duplicados por elemento para essa chave. A propriedade chave nome pode ter cardinalidade SET, se queremos capturar todos os nomes de um indivíduo (incluindo apelido, nome de solteiro, etc)
- A configuração padrão da cardinalidade é SINGLE. Perceba que as propriedades chave usadas nas arestas e as propriedades têm cardinalidade SINGLE. Colocar vários valores para uma única chave em uma aresta ou propriedade, não é suportado

```
mgmt = graph.openManagement()
birthDate=mgmt.makePropertyKey('birthDate').dataType(Long.class).
cardinality(Cardinality.SINGLE).make()
name = mgmt.makePropertyKey('name').dataType(String.class).
cardinality(Cardinality.SET).make( )
sensorReading = mgmt.makePropertyKey('sensorReading').dataType(Double.class).
cardinality(Cardinality.LIST).make()
mgmt.commit()
```

8.3. TIPOS DE RELAÇÕES

Labels de aresta e propriedades chave em conjunto representam um tipo de relação. Os nomes dos tipos de relação devem ser únicos no grafo, o que significa que as propriedades chave e labels de aresta não podem ter o mesmo nome. Existem métodos na API do Titan para consultar a existência ou recuperar os tipos de relações que englobam as propriedades chave e labels de arestas.

```
mgmt = graph.openManagement()
if (mgmt.containsRelationType('name'))
   name = mgmt.getPropertyKey('name')
mgmt.getRelationTypes(EdgeLabel.class)
mgmt.commit()
```

8.4. DEFININDO LABELS DE VÉRTICES

Ao contrário dos labels de aresta, os labels dos vértices são opcionais. Labels de vértices são úteis para distinguir diferentes tipos de vértices, exemplo, vértices de usuário e vértices de produto.

Para compatibilidade com modelos, Titan fornece métodos diferentes, com nomes diferentes, para adicionar vértices rotulados e não rotulados:

- addVertexWithLabel
- addVertex

Mesmo os labels sendo opcionais no nível do modelo conceitual e de dados, o Titan atribui a todos os vértices um rótulo como um detalhe de implementação interna. Vértices criados pelos métodos addVertex usam o label padrão do Titan.

Para criar um label, é preciso chamar o makeVertexLabel(String).make() em um grafo aberto ou gestão de transação e fornecer o nome do label do vértice como argumento. Nome de label de vértice deve ser único no grafo.

O código abaixo mostra como criar um label de vértice.

```
mgmt = graph.openManagement()
person = mgmt.makeVertexLabel('person').make()
mgmt.commit()
// Cria um vértice rotulado
person = graph.addVertex(label, 'person')
// Cria um vértice sem rótulo
v = graph.addVertex()
graph.tx().commit()
```

8.5. ALTERAR ELEMENTOS DO ESQUEMA

A definição de um label de aresta, propriedade chave ou label de vértice, não pode sofrer alteração uma vez que estão comitados no grafo, no entanto, os nomes dos elementos do esquema podem ser alterados através TitanManagement.changeName(TitanSchemaElement, String), como mostrado no exemplo a seguir, onde a propriedade chave place é renomeado para location.

```
mgmt = graph.openManagement()
place = mgmt.getPropertyKey('place')
mgmt.changeName(place, 'location')
mgmt.commit()
```

Note que as mudanças de nome no esquema podem não ser imediatamente visível na instância atual em execução de transação e outras instâncias no cluster do grafo Titan. Enquanto mudanças de nome de esquema são anunciadas a todas as instâncias Titan através do backend de armazenamento, esse processo pode demorar algum tempo para que as alterações tenham

efeito e poderá exigir a reinicialização, por exemplo, no caso de certas condições de falha – problema na conexão de rede – coincidirem com a renomeação. Assim, o usuário deve assegurar que em qualquer mudança:

- O label ou chave renomeado não esteja atualmente em uso ativo (ou seja, escritos ou lidos) e não estará em uso até que todas as instâncias Titan estejam cientes da mudança de nome
- Transações em execução acomodem ativamente o breve período intermediário em que o nome do velho ou novo é validado na instância específica do Titan e o status da mudança de nome é anunciado. Por exemplo, poderia ter uma consulta para ambos os nomes em simultâneo, o novo e o velho

Caso surja a necessidade de redefinir um tipo de esquema existente, recomenda-se mudar o nome deste tipo para um nome que não está neste momento em uso.

Depois disso, um novo label ou a chave pode ser definido com o nome original, assim, efetivamente substituindo o antigo. No entanto, note que isso não afetaria vértices, arestas ou propriedades previamente escritas com o tipo existente. Redefinir elementos do grafo existente não é suportado on-line e deve se realizado por meio de uma transformação em lote do grafo.

9. INDEXANDO PARA UM MELHOR DESEMPENHO (TITAN:DB)

Titan suporta dois diferentes tipos de indexação para acelerar o processamento de consultas:

- Graph Index
- vertex-centric index

A maioria das consultas ao grafo começam a atravessá-lo através de uma lista de vértices ou arestas que são identificadas por suas propriedades. Graph Index faz com que operações de recuperação global sejam eficientes em grafos muito grandes. Índice Vertex-centric acelera o processo de percorrer o grafo, sobretudo quando a travessia será através de vértices com muitas arestas incidentes.

9.1. ÍNDICES DO GRAFO

Índices do grafo (Graph Index) é uma estrutura de índice global abrangendo todo o grafo, permitindo a recuperação eficiente de vértices e arestas, através de suas propriedades, aplicando filtros seletivos. Exemplo:

```
g.V().has('name', 'hercules')
g.E().has('reason', textContains('loves'))
```

A primeira consulta procura em todos os vértices aquele com o nome hercules;

A segunda consulta procura em todas as arestas aquelas em que a propriedade chave reason tenha a palavra “loves”. Sem um índice no grafo que responda a essas consultas, será necessário percorrer o grafo inteiro, por todos os vértices e arestas até encontrar o que satisfaça a condição da consulta. Isso seria muito ineficiente para grafos muito grandes.

O Titan tem dois tipos de índices de grafo (Graph Index): composite e mixed.

O composite é rápido e eficiente, mas, limitado a casos particulares onde às combinações de propriedades chave foram definidas previamente.

O mixed pode ser usado em qualquer combinação de chaves indexadas e suporta múltiplas condições predicadas, além de depender de um sistema armazenamento de índice externo.

Os dois tipos de índices são criados através do sistema de gerenciamento do Titan, que retorna um construtor de índice.

```
TitanManagement.buildIndex(String, Class)
```

O primeiro argumento define o nome do índice e o segundo argumento especifica o tipo de elemento a ser indexado.

O nome do índice do grafo deve ser único.

Grafos indexados por propriedades chaves recém definidas, na mesma transação da indexação, por exemplo, estarão imediatamente disponíveis.

Grafos indexados por propriedades chaves que já estão em uso necessitará a execução de um procedimento de reindexação, para assegurar que o índice tenha todos os elementos previamente adicionados. Até que o procedimento de reindexação esteja concluído, o índice não estará disponível.

O mais indicado é definir os índices do grafo na mesma transação, como no esquema inicial.

Nota:

Na falta de um índice, o Titan percorrerá todo grafo até encontrar o que foi solicitado, isso é muito ineficiente. Habilitar a configuração force-index em produção, evita essa ação ineficiente e lenta de scannear o grafo todo, forçando a criação de um índice.

9.1.1. ÍNDICE COMPOSITE

O índice composite recupera vértices e arestas por uma ou várias chaves (key).

Considere o seguinte índice composite:

```
graph.tx().rollback() //Nunca cria novos índices enquanto uma transação está ativa
mgmt = graph.openManagement()
name = mgmt.getPropertyKey('name')
age = mgmt.getPropertyKey('age')
mgmt.buildIndex('byNameComposite',
Vertex.class).addKey(name).buildCompositeIndex()
mgmt.buildIndex('byNameAndAgeComposite',
Vertex.class).addKey(name).addKey(age).buildCompositeIndex()
mgmt.commit()
//Espera pelo índice para ficar disponível
mgmt.awaitGraphIndexStatus(graph, 'byNameComposite').call()
mgmt.awaitGraphIndexStatus(graph, 'byNameAndAgeComposite').call()
//Reindexa os dados existentes
mgmt = graph.openManagement()
mgmt.updateIndex(mgmt.getGraphIndex("byNameComposite"),
SchemaAction.REINDEX).get()
mgmt.updateIndex(mgmt.getGraphIndex("byNameAndAgeComposite"),
SchemaAction.REINDEX).get()
```

```
mgmt.commit()
```

Primeiro, duas propriedades chaves já definidas são atribuídas as variáveis name e age, depois um índice simples, composite, é construído em cima da propriedade chave nome.

O Titan irá usar esse índice para responder a consultas como:

```
g.V().has('name', 'hercules')
```

O segundo índice composite inclui ambas chaves (name, age).

O Titan irá usar esse índice para responder a consultas como:

```
g.V().has('age', 30).has('name', 'hercules')
```

Note que todas as chaves de um índice de grafo composite deve ser encontrada na consulta para que o índice seja usado. Por exemplo, a consulta a seguir não pode ser respondida com nenhum dos dois index criados, porque a consulta só tem a constrain age, mas não, name.

```
g.V().has('age', 30)
```

Note também que índices de grafo composite podem somente ser usado em constraints de igualdade como nas queries anteriores. A seguinte consulta pode ser respondida só com índice definido na chave nome, porque a constraint age não é uma restrição de igualdade.

```
g.V().has('name', 'hercules').has('age', inside(20, 50))
```

Índices composite não requer configuração de um indexador externo de backend e são suportados pelo sistema de armazenamento backend primário. Consequentemente, modificações no índice composite são persistidas através da mesma transação, assim como as modificações do grafo, o que significa que essas mudanças são atômicas e/ou consistentes se o sistema de armazenamento rodando atrás suporta atomicidade e consistência.

Nota:

Um índice composite pode ter uma ou múltiplas chaves. Um índice composite com apenas uma chave, é chamado de: key-index.

9.1.1.1. ÍNDICE DE UNICIDADE

Índices compostos também podem ser utilizados para reforçar a exclusividade da propriedade no grafo. Se um índice composto é definido como unique( ), pode haver no máximo um vértice ou aresta, para qualquer concatenação de valores de propriedades associadas com as chaves do índice. Por exemplo, para impor que os nomes sejam únicos em todo o grafo, o seguinte index composite poderia ser definido:

```
graph.tx().rollback()  //Nunca cria novos índices enquanto uma transação está ativa
mgmt = graph.openManagement()
name = mgmt.getPropertyKey('name')
```

```
mgmt.buildIndex('byNameUnique',
Vertex.class).addKey(name).unique().buildCompositeIndex()
mgmt.commit()
//Espera pelo índice para ficar disponível
mgmt.awaitGraphIndexStatus(graph, 'byNameUnique').call()
//Reindexa os dados existentes
mgmt = graph.openManagement()
mgmt.updateIndex(mgmt.getGraphIndex("byNameUnique"),
SchemaAction.REINDEX).get()
mgmt.commit()
```

Nota:

Para reforçar a unicidade contra uma eventual consistência de armazenamento backend, a consistência do índice deve ser explicitamente definida para ativar o bloqueio.

9.1.2. ÍNDICE MIXED

Índices mixed recuperam vértices ou arestas por qualquer combinação de propriedades chave previamente adicionadas. Índices mixed fornecem mais flexibilidade do que índices composite e apoiam predicados condicional adicional para além da igualdade. Por outro lado, os índices mixed são mais lentos para a maioria das consultas de igualdade do que índices composite.

Ao contrário de índices composite, índices mixed exigem uma configuração de um backend de indexação e usa essa infraestrutura de indexação para executar operações de pesquisa. Titan pode suportar vários backends de indexação em uma única instalação. Cada backend de indexação deve ser identificado exclusivamente pelo nome na configuração do Titan, que é chamado o nome do backend de indexação.

```
graph.tx().rollback()  //Nunca cria novos índices enquanto uma transação está ativa
mgmt = graph.openManagement()
name = mgmt.getPropertyKey('name')
age = mgmt.getPropertyKey('age')
mgmt.buildIndex('nameAndAge',Vertex.class).
addKey(name).addKey(age).buildMixedIndex("search")
mgmt.commit()
//Espera pelo índice para ficar disponível
mgmt.awaitGraphIndexStatus(graph, 'nameAndAge').call()
```

```
//Reindexa os dados existentes
mgmt = graph.openManagement()
mgmt.updateIndex(mgmt.getGraphIndex("nameAndAge"),
SchemaAction.REINDEX).get()
mgmt.commit()
```

O exemplo acima define um índice mixed contendo as propriedades chave nome e idade. A definição refere-se à indexação backend de nome search, assim o Titan sabe qual backend configurado de indexação ele deve usar para esse índice particular. O parâmetro search especificado na chamada buildMixedIndex, deve coincidir com a segunda cláusula na definição de configuração do Titan, assim: index.search.backend.

Se o índice foi o solrsearch, então a definição de configuração apareceria assim: index.solrsearch.backend.

O exemplo mgmt.buildIndex especificado acima, usa o texto search como seu comportamento padrão. A declaração de um índice que o define explicitamente como um índice de texto pode ser escrito da seguinte forma:

```
mgmt.buildIndex('nameAndAge',Vertex.class).addKey(name,Mapping.TEXT.
getParameter()).addKey(age,Mapping.TEXT.getParameter()).
buildMixedIndex("search")
```

Embora a definição do índice do exemplo acima é semelhante ao índice composite, ele fornece maior apoio a consulta e pode responder a qualquer uma das seguintes consultas.

```
g.V().has('name', textContains('hercules')).has('age', inside(20, 50))
g.V().has('name', textContains('hercules'))
g.V().has('age', lt(50))
```

Índices mixed suportam pesquisa de texto completo, pesquisa de intervalo, pesquisa geo , etc.

Nota:

Ao contrário de índices composite, índices mixed não suportam unicidade.

9.1.2.1. ADICIONANDO PROPRIEDADES CHAVE

Propriedades chave podem ser adicionadas a um índice mixed existente, o que permite consultas subsequentes incluir esta chave na condição de consulta.

```
graph.tx().rollback()  //Nunca cria novos índices enquanto uma transação está ativa
mgmt = graph.openManagement()
location = mgmt.makePropertyKey('location').dataType(Geoshape.class).make()
```

```
nameAndAge = mgmt.getGraphIndex('nameAndAge')
mgmt.addIndexKey(nameAndAge, location)
mgmt.commit()
//Espera pelo índice para ficar disponível
mgmt.awaitGraphIndexStatus(graph, 'nameAndAge').call()
//Reindexa os dados existentes
mgmt = graph.openManagement()
mgmt.updateIndex(mgmt.getGraphIndex("nameAndAge"),
SchemaAction.REINDEX).get()
mgmt.commit()
```

Para adicionar uma chave recém-definida, primeiro recupera-se o índice existente a partir da operação de gestão de transação pelo seu nome e, em seguida, chama o método addIndexKey para adicionar a chave a este índice.

Se a chave adicionada está definida na mesma operação de gestão, ela estará imediatamente disponível para consulta. Se a propriedade chave já está em uso, adicionar chave requer a execução de um procedimento de reindexação para assegurar que o índice contenha todos os elementos adicionados anteriormente. Até que o procedimento de reindexação esteja completo, a chave não se encontra disponível no índice mixed.

9.1.2.2. MAPEANDO PARÂMETROS

Ao adicionar uma propriedade chave a um índice mixed - quer através do construtor do índice ou o método addIndexKey - uma lista de parâmetros pode ser especificada opcionalmente para ajustar como o valor da propriedade é mapeada para o backend de indexação.

9.1.3 ORDENAÇÃO

A ordem em que os resultados de uma consulta ao grafo são devolvidos, pode ser definida utilizando a diretiva order ( ).by ( ). O método order ( ).by ( ) espera dois parâmetros:

- O nome da propriedade chave segundo a qual ordenará os resultados. Os resultados serão ordenados pelo valor dos vértices ou arestas para esta propriedade chave
- A ordem de classificação: pode ser crescente (icr) ou decrescente (decr)

Por exemplo, a consulta g.V().has('name', textContains('hercules')).order().by('age', decr).limit(10) recupera os dez indivíduos mais velhos com hercules em seu nome.

Quando se utiliza order( ).by( ), é importante notar que:

- Um índice composite não suporta nativamente ordenar resultados de pesquisa. Todos os resultados serão recuperados e em seguida, classificadas na memória. Para grandes conjuntos de resultados, pode ser muito dispendioso
- O mixed suporta ordenação nativamente e eficientemente. No entanto, a propriedade chave utilizada no order( )by( ), deve ter sido previamente adicionados ao índice mixed para suportar resultados ordenados nativo. Isto é importante nos casos em que a chave utilizada no order( ).by( ) é diferente das chaves da consulta. Se a propriedade chave não faz parte do índice, então a ordenação requer carregar todos os resultados na memória

9.1.4 RESTRIÇÃO DE LABEL

Em muitos casos é desejável apenas índices de vértices ou arestas com um label específico. Por exemplo, pode-se querer indexar apenas deuses por seu nome e não todos os vértices que tenham a propriedade nome. Ao definir um índice é possível restringir o índice para um vértice ou aresta de label particular, utilizando o método indexOnly do construtor de índices. Em seguida cria um índice composite para o nome da propriedade chave que indexa apenas vértices marcados com deus.

```
graph.tx().rollback()  //Never create new indexes while a transaction is active
mgmt = graph.openManagement()
name = mgmt.getPropertyKey('name')
god = mgmt.getVertexLabel('god')
mgmt.buildIndex('byNameAndLabel',
Vertex.class).addKey(name).indexOnly(god).buildCompositeIndex()
mgmt.commit()
//Wait for the index to become available
mgmt.awaitGraphIndexStatus(graph, 'byNameAndLabel').call()
//Reindex the existing data
mgmt = graph.openManagement()
mgmt.updateIndex(mgmt.getGraphIndex("byNameAndLabel"),
SchemaAction.REINDEX).get()
mgmt.commit()
```

Semelhante restrição de label se aplica a índices mixed. Quando um índice composite com restrição de label é definido como único, a restrição de exclusividade só se aplica a propriedades em vértices ou arestas para o rótulo especificado.

9.1.5. COMPOSITE VS MIXED

Use um índice composite para recuperar a combinação exata do índice. Índices composite não requerem configuração ou um sistema de índice externo, e muitas vezes são significativamente mais rápidos do que os índices mixed.

Como uma exceção, utiliza-se um índice mixed para correspondências exatas, quando o número de valores distintos de restrição de consulta é relativamente pequeno ou quando se espera que um valor seja associado com muitos elementos no grafo (isto é, no caso de uma baixa seletividade).

Use índice mixed para intervalo numérico, full-text ou indexação geo-spatial. Além disso, usar um índice mixed pode acelerar consultas order( ).by( ).

9.2 ÍNDICE VERTEX-CENTRIC

Índices centrados nos vértices (vertex-centric) são estruturas de índice local, construídas individualmente por vértice. Em grandes grafos, vértices podem ter milhares de arestas incidentes. Transitar através desses vértices pode ser muito lento, porque um grande subconjunto das arestas incidentes tem que ser recuperadas e em seguida filtradas na memória para coincidir com as condições da travessia. Índices vertex-centric pode acelerar essas travessias usando estruturas localizadas de índice para recuperar apenas as arestas que precisam ser percorridas.

Suponhamos que Hércules lutou contra centenas de monstros, além dos três capturado no Grafo introdutório dos Deuses. Sem um índice vertex-centric, uma consulta pedindo esses monstros que lutaram entre o ponto de tempo 10 e 20, exigiria a recuperação de todas as arestas battled, mesmo que haja apenas algumas arestas correspondentes.

```
h = g.V().has('name', 'hercules').next()
g.V(h).outE('battled').has('time', inside(10, 20)).inV()
```

Construir um índice vertex-centric pelo tempo, acelera essas consultas.

```
graph.tx().rollback()  //Never create new indexes while a transaction is active
mgmt = graph.openManagement()
time = mgmt.getPropertyKey('time')
battled = mgmt.getEdgeLabel('battled')
mgmt.buildEdgeIndex(battled, 'battlesByTime', Direction.BOTH, Order.decr, time)
```

```
mgmt.commit()
//Wait for the index to become available
mgmt.awaitGraphIndexStatus(graph, 'battlesByTime').call()
//Reindex the existing data
mgmt = graph.openManagement()
mgmt.updateIndex(mgmt.getGraphIndex("battlesByTime"),
SchemaAction.REINDEX).get()
mgmt.commit()
```

Este exemplo constrói um índice vertex-centric pelas arestas battled em ambos as direções pelo time e em ordem decrescente. Um índice de vertex-centric é construído em cima do label específico da aresta que é o primeiro argumento para o método TitanManagement.buildEdgeIndex de construção index ( ). O índice só se aplica a arestas com esse label - battled no exemplo acima. O segundo argumento é o nome único para o índice. O terceiro argumento é a direção da aresta em que o índice é construído. O índice só se aplica a percursos ao longo das arestas no sentido que foi definido. Neste exemplo, o índice vertex-centric é construído em ambos os sentidos, o que significa que o tempo restringe percursos ao longo das arestas battled tanto no sentido IN como no OUT. Titan irá manter um índice de vértice central em ambos os vértices in e out das arestas battled. Alternativamente, pode-se definir o índice a ser aplicado à direção OUT somente, que iria acelerar travessias de hercules para os monstros, mas não no sentido inverso. Nesse caso, só seria necessário manter um índice, e portanto, metade do custo de manutenção e armazenamento do índice. Os dois últimos argumentos são a ordem de classificação do índice e uma lista de propriedades chaves do índice. A ordem de classificação é opcional e o padrão é ordem crescente (Order.ASC). A lista de propriedades chaves deve ser não-vazia e definir as chaves através da qual a indexação das arestas de um dado label será feita. Um índice vertex-centric pode ser definido com várias chaves.

```
graph.tx().rollback()  //Never create new indexes while a transaction is active
mgmt = graph.openManagement()
time = mgmt.getPropertyKey('time')
rating = mgmt.makePropertyKey('rating').dataType(Double.class).make()
battled = mgmt.getEdgeLabel('battled')
mgmt.buildEdgeIndex(battled, 'battlesByRatingAndTime', Direction.OUT, Order.decr,
rating, time)
mgmt.commit()
```

```
//Wait for the index to become available
mgmt.awaitRelationIndexStatus(graph, 'battlesByRatingAndTime', 'battled').call()
//Reindex the existing data
mgmt = graph.openManagement()
mgmt.updateIndex(mgmt.getRelationIndex(battled,'battlesByRatingAndTime'),
SchemaAction.REINDEX).get()
mgmt.commit()
```

Este exemplo estende o esquema por uma propriedade de classificação nas arestas battled e constrói um índice vertex-centric que indexa as arestas battled na direção out, pelo rating e pelo time em ordem decrescente. Note que a ordem em que as propriedades chaves são especificadas é importante, porque índices de vertex-centric são índices de prefixo. Isto significa que, arestas battled são indexadas primeiro pelo rating (ranking) e em segundo pelo time.

```
h = g.V().has('name', 'hercules').next()
g.V(h).outE('battled').property('rating', 5.0) //Add some rating properties
g.V(h).outE('battled').has('rating', gt(3.0)).inV()
g.V(h).outE('battled').has('rating', 5.0).has('time', inside(10, 50)).inV()
g.V(h).outE('battled').has('time', inside(10, 50)).inV()
```

Assim, o índice battlesByLocationAndTime pode acelerar as duas primeiras, mas não a terceira consulta.

Vários índices vertex-centric podem ser construídos para o mesmo label de aresta, a fim de suportar diferentes restrições de percursos. O otimizador de consulta do Titan tenta escolher o índice mais eficiente para qualquer travessia. Índices vertex-centric apenas suportam restrições de igualdade e de intervalo.

9.2.1. PERCURSOS ORDENADOS

As seguintes consultas especificam uma ordem na qual as arestas incidentes serão atravessadas. Usar o comando localLimit para recuperar um subconjunto das arestas (em uma dada ordem) para cada vértice que é atravessado.

```
h = g..V().has('name', 'hercules').next()
g.V(h).local(outE('battled').order().by('time', decr).limit(10)).inV().values('name')
g.V(h).local(outE('battled').has('rating',5.0).order().by('time',decr).limit(10)).values('plac
e')
```

A primeira consulta pede os nomes dos 10 monstros que mais recentemente lutaram com Hércules. A segunda consulta pede os 10 lugares mais recentes das batalhas de Hércules que são classificados como 5 estrelas. Em ambos os casos, a consulta é limitada por uma ordem em uma propriedade chave com um limite para o número de elementos a serem devolvidos.

Tais consultas também podem ser eficientemente respondidas pelos índices vertex-centric se a chave de ordem corresponde à chave do índice e a ordem requerida (isto é, crescente ou decrescente) é a mesma definida para o índice. O índice battlesByTime seria usado para responder à primeira pergunta e battlesByLocationAndTime se aplica ao segundo. Note-se, que o índice battlesByLocationAndTime não pode ser usado para responder à primeira consulta porque uma restrição de igualdade na classificação deve estar presente para a segunda chave no índice para ser eficaz.

10. TRANSAÇÃO (TITAN:DB)

Quase toda interação com a Titan está associada com uma transação. Transações Titan são seguras para o uso concorrente de multi-thread.

Métodos em uma instância de grafo do Titan como graph.v (...) e graph.commit( ) executam uma pesquisa ThreadLocal para recuperar ou criar uma transação associada com a thread chamada. Quem chama pode alternativamente deixar de usar um gerenciamento de transação ThreadLocal em favor de chamar graph.newTransaction( ), que retorna uma referência a um objeto de transação com métodos para ler / escrever dados no grafo e confirmar (commit) ou anular (rollback).

Transações Titan não são necessariamente ACID. Elas podem ser assim configuradas no BerkleyDB, mas eles geralmente não são assim no Cassandra ou HBase, onde o sistema de armazenamento subjacente não fornece isolamento serializável ou escritas atômicas multi-row e o custo de simular essas propriedades são substanciais.

10.1. MANIPULAÇÃO DE TRANSAÇÕES

Cada operação em um grafo Titan, ocorre dentro do contexto de uma transação. De acordo com a especificação do Blueprints, cada thread abre sua própria transação no banco de dados de grafo com a primeira operação (isto é, a recuperação ou mutação) no grafo.

```
graph = TitanFactory.open("berkeleyje:/tmp/titan")
juno = graph.addVertex() //Automatically opens a new transaction
juno.property("name", "juno")
graph.tx().commit() //Commits transaction
```

Neste exemplo, uma base de dados de um grafo Titan local está aberto. Adicionar o vértice "Juno" é a primeira operação (neste segmento) que abre automaticamente uma nova transação. Todas as operações subsequentes ocorrem no contexto dessa mesma transação até que a transação seja explicitamente parada ou o banco de dados for desligado shutdown ( ), isto é, fechado. Se as transações estão ainda abertas quando shutdown( ) é chamado, então o comportamento das operações em aberto é tecnicamente indefinido. Na prática, todas as transações não vinculadas a thread, geralmente serão eficazmente revertidas (rollback), mas a transação vinculadas a thread pertencente ao segmento que chamou shutdown( ), será primeiro commitada. Note que as duas operações de ler e escrever ocorrem dentro do contexto de uma transação.

10.2. ESCOPO DE TRANSAÇÃO

Todos os elementos do grafo (vértices, arestas e tipos) estão associados com o escopo da transação em que foram recuperados ou criados. Sob padrão semântico transacional do Blueprint, as transações são criadas automaticamente com a primeira operação no grafo e fechado explicitamente usando commit ( ) ou rollback ( ). Uma vez que a transação é fechada, todos os elementos do grafo associado a essa transação se torna obsoleto e indisponível. No entanto, o Titan fará a transição automaticamente de vértices e tipos para o novo escopo transacional, como mostrado neste exemplo.

```
graph = TitanFactory.open("berkeleyje:/tmp/titan")
juno = graph.addVertex() //Automatically opens a new transaction
graph.tx().commit() //Ends transaction
juno.property("name", "juno") //Vertex is automatically transitioned
```

Arestas, por outro lado, não são transferidas automaticamente e não pode ser acessada fora da sua transação original. Eles devem ser explicitamente transicionada.

```
e = juno.addEdge("knows", graph.addVertex())
graph.tx().commit() //Ends transaction
e = g.E(e).next() //Need to refresh edge
e.property("time", 99)
```

10.3. FALHAS DE TRANSAÇÂO

Quando uma transação é commitada, o Titan tentará persistir todas as alterações no backend de armazenamento. Isto pode não ser sempre bem-sucedido devido a exceções IO, erros de rede, falhas de máquina ou indisponibilidade de recursos. Por isso, as transações podem falhar. Na verdade, as transações acabarão por falhar em sistemas grandes. Portanto, é altamente recomendável que o seu código espere e trate tais falhas.

```
try {
   if (g.V().has("name", name).iterator().hasNext())
      throw new IllegalArgumentException("Username already taken: " + name)
   user = graph.addVertex()
   user.property("name", name)
   graph.tx().commit()
} catch (Exception e) {
   //Recover, retry, or return error message
   println(e.getMessage())
```

```
}
```

O exemplo acima demonstra uma implementação de inscrição de usuário simplificada, onde nome é o nome do usuário que deseja registrar. Em primeiro lugar, é verificado se um usuário com esse nome já existe. Se não, um novo vértice de usuário é criado e o nome atribuído. Finalmente, a transação é confirmada.

Se a transação falhar, uma TitanException é lançada. Há uma variedade de razões pelas quais uma transação pode falhar. Titan diferencia entre falhas potencialmente temporárias e permanentes.

Falhas potencialmente temporárias são aquelas relacionadas a indisponibilidade de recursos IO (por exemplo, network timeout). O Titan tenta automaticamente recuperar-se de falhas temporárias repetindo a persistência do estado transacional depois de algum atraso. O número de tentativas de repetição e o atraso da repetição são configuráveis.

Falhas permanentes podem ser causadas por perda de conexão completa, falha de hardware ou contenção de bloqueio. Para entender a causa da contenção de bloqueio, considere o exemplo de inscrição acima e suponha que um usuário tenta a inscrição com nome de utilizador "juno". Este nome de usuário ainda poderá estar disponível no início da transação, mas com tempo a transação é confirmada, outro usuário pode tentar simultaneamente se registrado com "Juno", por isso a transação mantém o bloqueio no nome do usuário, fazendo, portanto, com que a outra transação falhe. Dependendo da semântica da transação, pode-se recuperar de uma falha de contenção de bloqueio executando novamente toda a transação.

Exceções permanentes que podem falhar em uma transação incluem:

- Exceção de bloqueio permanente (contenção de bloqueio Local): a outra thread local já tenha sido concedido um bloqueio em conflito
- PermanentLockingException (incompatibilidade de valor esperado para o X: esperado = Y vs atual =Z): A verificação de que o valor lido nesta transação é a mesma que a do armazenamento de dados após a aplicação para o bloqueio que falhou. Em outras palavras, outra operação modificau o valor depois de ter sido lido e modificado.

10.4. TRANSAÇÕES MULTI-THREADED

Titan suporta transações multi-threaded através da interface ThreadedTransactionalGraph do Blueprint. Assim, para acelerar o processamento de transações e utilizar a arquitetura multi-core, várias threads podem ser executadas simultaneamente em uma única transação.

Com manipulação de transações padrão do Blueprints, cada thread abre automaticamente sua própria transação no banco de dados de grafo. Para abrir uma transação independente do segmento, use o método newTransaction ( ).

```
tx = graph.newTransaction();
threads = new Thread[10];
for (int i=0; i<threads.length; i++) {
   threads[i]=new Thread({
      println("Do something");
   });
   threads[i].start();
}
for (int i=0; i<threads.length; i++) threads[i].join();
tx.commit();
```

O método newTransaction () retorna um novo objeto TransactionalGraph que representa esta transação recém-iniciada. O objeto grafo tx suporta todos os métodos que o grafo original suporta, mas, o faz sem abrir novas operações para cada thread. Isso permite iniciar várias threads, todas trabalhando simultaneamente na mesma transação e uma delas finalmente confirma (commit) a transação quando todas as threads concluídrem seu trabalho.

Titan se baseia em estruturas de dados otimizadas para uso concorrente, para suportar centenas de threads simultâneas em execução de forma eficiente em uma única transação.

10.5. ALGORITMOS CONCORRENTES

Threads independentes iniciada através newTransaction ( ) são particularmente úteis na implementação de algoritmos de grafo concorrentes. A maioria das travessias ou passagem de mensagens (ego-centric) como algoritmos de grafo são execuções paralelizadas e complexas, ou seja, executadas através de várias threads, através de newTransaction, é possível fazer isso com menos esforço. Cada ums dessas threads podem operar em um único objeto TransactionalGraph retornado por newTransaction sem bloquear o outro.

10.6. TRANSAÇÕES ANINHADAS

Outro caso de uso para transações de threads independentes, é transação aninhada, que deve ser independente da operação que a envolve.

Por exemplo, suponha um trabalho transacional de longa duração tem para criar um novo vértice com um nome único. Impor nomes exclusivos exige um bloqueio e uma vez que a transação será executada por um longo tempo, o congestionamento de bloqueio e as falhas transacionais serão prováveis e caras.

```
v1 = graph.addVertex()
//Do many other things
v2 = graph.addVertex()
v2.property("uniqueName", "foo")
v1.addEdge("related", v2)
//Do many other things
// This long-running tx might fail due to contention on its uniqueName lock
graph.tx().commit()
```

Uma maneira de contornar isso é criar o vértice em uma thread de transação curta, independente e aninhada, como demonstrado pelo seguinte código:

```
v1 = graph.addVertex()
//Do many other things
tx = graph.newTransaction()
v2 = tx.addVertex()
v2.property("uniqueName", "foo")
tx.commit() // Any lock contention will be detected here
v1.addEdge("related", g.V(v2).next()) // Need to load v2 into outer transaction
//Do many other things
graph.tx().commit() // Can't fail due to uniqueName write lock contention involving v2
```

10.7. MANIPULAÇÃO COMUM DE PROBLEMAS TRANSACIONAIS

As transações são iniciadas automaticamente com a primeira operação executada no grafo. Não é necessário iniciar uma transação manualmente. O método newTransaction é usado para começar apenas as operações multi-threaded.

As transações são automaticamente iniciadas sob a semântica Blueprints mas não automaticamente terminadas. As transações têm que ser encerradas manualmente com g.commit( ) se bem sucedido ou g.rollback( ) se não. A finalização manual de transações é necessária porque somente o usuário conhece o limite transacional. Uma transação tentará manter o seu estado a partir do início da transação. Isso pode levar a um comportamento inesperado em aplicações multi-threaded, como ilustrado no exemplo artificial seguinte:

```
v = g.V(4).next() // Retrieve vertex, first action automatically starts transaction
g.V(v).bothE()
>> returns nothing, v has no edges
//thread is idle for a few seconds, another thread adds edges to v
g.V(v).bothE()
>> still returns nothing because the transactional state from the beginning is maintained
```

Tal comportamento inesperado é provável de ocorrer em aplicações cliente-servidor onde o servidor mantém vários segmentos para responder a pedidos de clientes. Portanto, é importante terminar a transação após uma unidade de trabalho (por exemplo, fragmento de código, consulta, etc). Dessa forma, o exemplo acima deve ser:

```
v = g.V(4).next() // Retrieve vertex, first action automatically starts transaction
g.V(v).bothE()
graph.tx().commit()
//thread is idle for a few seconds, another thread adds edges to v
g.V(v).bothE()
>> returns the newly added edge
graph.tx().commit()
```

Ao usar transações multi-threaded através newTransaction todos os vértices e arestas recuperados ou criados no escopo da referida transação, não estão disponíveis fora do âmbito da referida transação. Acesso a esses elementos depois que a transação foi fechada resultará em uma exceção. Tal como demonstrado no exemplo acima, tais elementos têm de ser recarregados(refreshed) explicitamente na nova transação utilizando g.V (existingVertex) ou g.E (existingEdge).

10.8. CONFIGURAÇÃO DE TRANSAÇÕES

O método do Titan TitanGraph.buildTransaction( ) dá ao usuário a capacidade de configurar e iniciar uma nova transação multi-threaded em um grafo. Por isso, é idêntico ao TitanGraph.newTransaction( ) com opções de configuração adicionais.

O buildTransaction() retorna uma TransactionBuilder que permite que os seguintes aspectos de uma transação sejam configurados:

- readOnly( ) - faz com que operação read-only e qualquer tentativa de modificação do grafo resulte em uma exceção.
- enableBatchLoading( ) - permite batch-loading para uma transação individual. Esta configuração resulta em eficiências semelhantes como a configuração graph-

wide, storage.batch-loading, devido à desativação de verificações de consistência e outras otimizações. Ao contrário storage.batch-loading esta opção não irá alterar o comportamento do backend de armazenamento.

- setTimestamp(long) - define o timestamp para esta transação e comunica ao backend de armazenamento de persistência. Dependendo da infra-estrutura de armazenamento, essa configuração pode ser ignorada. Para backends, eventualmente consistentes, este é o timestamp usado para resolver conflitos de gravação. Se essa configuração não for especificada explicitamente, Titan usa a hora atual.
- setVertexCacheSize(long size) - O número de vértices que esta transação armazena na memória. Quanto maior este número, mais memória uma transação potencialmente pode consumir. Se esse número é muito pequeno, uma transação poderá ter de voltar a buscar dados, o que provoca atrasos em particular para transações de longa execução.
- checkExternalVertexExistence(boolean) - Se esta transação deve verificar a existência de vértices para o utilizador desde ids de vértice. Essas verificações requerem acesso ao banco de dados, o que leva tempo. A verificação de existência só deve ser desativada se o usuário está absolutamente certo de que o vértice deve existir - caso contrário, a corrupção de dados pode acontecer.
- checkInternalVertexExistence(boolean) - Se esta transação deve verificar a existência de vértices durante a execução da consulta. Isto pode ser útil para evitar vértices fantasmas em infra-estruturas de armazenagem, eventualmente consistentes. Desativada por padrão. A ativação dessa configuração pode retardar o processamento de consultas.
- consistencyChecks(boolean) - Se o Titan deve reforçar o nível de esquema de restrições de consistência (por exemplo, restrições de multiplicidade). Desativando verificações de consistência leva a um melhor desempenho, mas requer que o usuário garanta a confirmação de consistência no nível do aplicativo para evitar inconsistências. USE com muito cuidado!

Uma vez que, as opções de configuração desejadas foram especificadas, a nova transação é iniciada via start( ), que retorna uma TitanTransaction.

11. CACHE (TITAN:DB)

11.1. CACHING

Titan emprega múltiplas camadas de armazenamento de dados em cache, para facilitar o processo transitar o grafo. As camadas de armazenamento em cache são listadas aqui na ordem em que são acessados de dentro de uma transação Titan.

11.2. TRANSACTION-LEVEL CACHING

Dentro de uma transação aberta, Titan mantém dois caches:

- Vertex cache: caches de vértices acessados e sua lista de adjacência (ou subconjuntos dos mesmos), para que o acesso posterior seja significativamente mais rápido dentro da mesma transação. Assim, este cache acelera a iteração.
- Index Cache: armazena em cache os resultados de consultas de índice para que as chamadas subsequentes ao índice possa acessar a partir da memória.

O tamanho de ambos é determinado pelo tamanho do cache de transação. O tamanho do cache de transação pode ser configurado via cache.tx-cache-size ou abrindo uma transação através do construtor de transação graph.buildTransction( ) utilizando o método setVertexCacheSize (int).

11.3. VERTEX CACHE

O cache de vértice contém vértices e o subconjunto de sua lista de adjacência que foi recuperado em uma transação particular. O número máximo de vértices mantido nesta memória cache é igual ao tamanho do cache de transação. Se a carga de trabalho da transação é uma travessia iterativa, o cache de vértice irá acelerar significativamente.

O tamanho do cache de vértice na pilha não é apenas determinado pelo número de vértices que ele possa conter, mas também pelo tamanho da sua lista de adjacência. Em outras palavras, os vértices com listas de adjacência grandes (isto é, muitas arestas incidentes) irá consumir mais espaço no cache do que aqueles com listas menores.

11.4. INDEX CACHE

O cache de índice contém os resultados de consultas de índice executadas no contexto de uma transação. Subsequentes chamadas a estes índices serão acessadas a partir deste cache e são, portanto, significativamente mais baratas.

11.5. DATABASE LEVEL CACHE

O database level cache mantém listas de adjacência (ou subconjuntos dos mesmos) em várias operações, além da duração de uma única transação. O cache de nível de banco de dados é compartilhado por todas as transações através de um banco de dados. É mais eficiente em espaço do que caches de nível de transação, mas, um pouco mais lento para acesso. Em contraste com as caches de nível de transação, as caches de nível de banco de dados não expiram imediatamente depois de fechar uma transação. Assim, o cache de nível de banco de dados acelera significativamente percursos no grafo.

Importante: o cache de nível de banco de dados está desabilitado por padrão na versão atual do Titan. Para ativá-la, defina cache.db-cache = true.

11.5.1. TEMPO DE EXPIRAÇÃO DO CACHE

A configuração mais importante para o comportamento de desempenho e de consulta é o tempo de expiração do cache que é configurado através cache.db-cache-time. O cache vai guardar elementos do grafo por alguns milissegundos. Se um elemento expirar, no próximo acesso, os dados serão lidos novamente a partir do backend de armazenamento.

11.5.2 TAMANHO DO CACHE

A opção de configurações cache.db-cache-size controla a quantidade de espaço na pilha que o Titan poderá consumir para armazenar o cache. Quanto maior o cache, mais eficaz ele será.

11.5.3 CLEAN UP WAIT TIME

Quando um vértice é modificado localmente (por exemplo, uma aresta é adicionada) todas as entradas de cache de nível de banco de dados relacionadas a esse vértice são marcadas como expiradas e eventualmente despejadas. Isso fará com que Titan atualize os dados do vértice no backend de armazenamento no próximo acesso e irá repopular o cache.

No entanto, quando a infra-estrutura de armazenamento é, eventualmente consistente, as modificações que desencadearam o despejo podem não ser ainda visível. Configurando cache.db-cache-clean-wait, o cache irá esperar pelo menos um número de milissegundos antes de preencher o cache com a entrada recuperada do backend de armazenamento.

Se Titan é executado localmente ou em um backend de armazenamento que garante visibilidade imediata de modificações, esse valor pode ser definido como 0.

11.5.4 STORAGE BACKEND CACHING

Cada backend de armazenamento mantém a sua própria camada de cache de dados. Estes caches se beneficiam de compressão, compactação de dados, expiração coordenada e muitas vezes são mantidos fora do heap, o que significa que grandes caches podem ser usados sem se preocupar com questões de coleta de lixo de memória. Embora estes caches possam ser significativamente maiores do que o cache de nível de banco de dados, eles também são mais lentos para acesso.

O tipo exato de caching e suas propriedades dependem da infraestrutura de armazenamento.

12. LOG DE TRANSAÇÃO (TITAN:DB)

O Titan pode automaticamente guardar as alterações transacionais para processamento adicional ou como um registro de mudança em um log. Para ativar o log para uma transação particular, é necessário especificar o nome do log alvo durante o início da operação

```
tx = graph.buildTransaction().logIdentifier('addedPerson').start()
u = tx.addVertex(label, 'human')
u.property('name', 'proteros')
u.property('age', 36)
tx.commit()
```

Após o commit, todas as alterações feitas durante a transação são registradas no sistema de registro de usuário em um log chamado addedPerson. O sistema de registro de usuário é um backend de registro configurável com uma interface de log compatível com o Titan. Por padrão, o log é escrito em um armazenamento separado no backend de armazenamento primário, que pode ser configurado conforme descrito abaixo. O identificador de log especificado durante o início da transação define o log em que as mudanças são registradas, permitindo assim que diferentes tipos de mudanças sejam registradas em logs separados para processamento individual.

```
tx = graph.buildTransaction().logIdentifier('battle').start()
h = tx.traversal().V().has('name', 'hercules').next()
m = tx.addVertex(label, 'monster')
m.property('name', 'phylatax')
h.addEdge('battled', m, 'time', 22)
tx.commit()
```

O Titan fornece um framework para processamento de log de transações do usuário para processar as alterações transacionais que foram feitas. O processador de log de transações é aberto através TitanFactory.openTransactionLog (TitanGraph) no grafo dos deuses. Então, pode-se adicionar processadores para um log específico que detém alterações transacionais.

```
import java.util.concurrent.atomic.*;
import com.thinkaurelius.titan.core.log.*;
import java.util.concurrent.*;
logProcessor = TitanFactory.openTransactionLog(g);
totalHumansAdded = new AtomicInteger(0);
totalGodsAdded = new AtomicInteger(0);
logProcessor.addLogProcessor("addedPerson").
```

```
        setProcessorIdentifier("addedPersonCounter").
        setStartTime(System.currentTimeMillis(), TimeUnit.MILLISECONDS).
        addProcessor(new ChangeProcessor() {
          @Override
          public void process(TitanTransaction tx, TransactionId txId, ChangeState
changeState) {
            for (v in changeState.getVertices(Change.ADDED)) {
              if (v.label().equals("human")) totalHumansAdded.incrementAndGet();
            }
          }
        }).
        addProcessor(new ChangeProcessor() {
          @Override
          public void process(TitanTransaction tx, TransactionId txId, ChangeState
changeState) {
            for (v in changeState.getVertices(Change.ADDED)) {
              if (v.label().equals("god")) totalGodsAdded.incrementAndGet();
            }
          }
        }).
        build();
```

Neste exemplo, um processador de log é construído para o log de transações usuário chamado addedPerson para processar as alterações feitas em transações que usaram o identificador de log addedPerson. Dois processadores de mudança são adicionados a este processador log. O primeiro processador conta o número de seres humanos adicionados e a segunda conta o número de deuses adicionados ao grafo.

Quando um processador de log é construído em cima de um registro específico, como o registro de addedPerson no exemplo acima, ele vai começar a ler registros de alterações transacionais do log imediatamente após a construção bem-sucedida e inicialização até o head do log. O horário de início especificado no construtor marca o ponto de tempo no log onde o processador de log irá começar a ler. Opcionalmente, pode-se especificar um identificador para o processador de log no construtor. O processador de log irá utilizar o identificador para persistir regularmente o seu estado de processamento, ou seja, ele vai manter um marcador no registro de log da última leitura. Se o processador de log é reiniciado mais tarde com o mesmo

identificador, ele vai continuar a leitura a partir do último registro de leitura. Isto é particularmente útil quando o processador de log precisa funcionar durante longos períodos de tempo e é, portanto, susceptível a falhas. Em tais situações de falha, o processador de log pode simplesmente ser reinicializado com o mesmo identificador. Deve-se assegurar que os identificadores de processador log são únicos em um cluster Titan, a fim de evitar conflitos nas marcas de leitura persistida.

Um processador de mudança deve implementar a interface ChangeProcessor. Seu método process( ) é chamado para cada registro de mudança lido a partir do log tratado pelo TitanTransaction, o ID da transação que causou a mudança e um recipiente ChangeState que detém as alterações transacionais. O container de mudança de estado pode ser consultado para recuperar elementos individuais que faziam parte da mudança de estado. No exemplo, todos os vértices adicionados são recuperados. O ID da transação fornecido pode ser usado para investigar a origem da transação que é identificada exclusivamente pela combinação do ID da instância Titan que executou a transação (txId.getInstanceId ( )) e a identificação de instância de transação específica (txId.getTransactionId ( )). Além disso, o tempo da operação está disponível através txId.getTransactionTime ( ).

Processadores de mudança são executados individualmente e em múltiplas threads. Se um processador de mudança acessa um estado global, deve assegurar-se de que tal estado permita o acesso simultâneo. Enquanto o processador de log lê registros de log sequencialmente, as alterações são processadas em vários segmentos, de modo que não se pode garantir que a ordem de registro é preservada nos processadores de mudança.

Note que os processadores de log executam cada processador de mudança registrada pelo menos uma vez para cada escrita no log de registros, o que significa que um único registro de mudança transacional pode ser processado várias vezes sob certas condições de falha. Não se pode adicionar ou remover processador de mudança de um processador de log em execução. Em outras palavras, um processador de log é imutável depois que ele é construído. Para alterar processamento de registro, tem que iniciar um novo processador de log e desligar um já existente.

```
logProcessor.addLogProcessor("battle").
    setProcessorIdentifier("battleTimer").
    setStartTime(System.currentTimeMillis(), TimeUnit.MILLISECONDS).
    addProcessor(new ChangeProcessor() {
      @Override
```

```
            public void process(TitanTransaction tx, TransactionId txId, ChangeState
changeState) {
                h = tx.V().has("name", "hercules").toList().iterator().next();
                for (edge in changeState.getEdges(h, Change.ADDED, Direction.OUT,
"battled")) {
                    if (edge.<Integer>value("time")>1000)
                        h.property("oldFighter", true);
                }
            }
        }).
        build();
```

O processador de log acima processa transações para o identificador de log battle com um único processador de mudança que avalia arestas battled que foram adicionadas a hercules. Este exemplo demonstra que o manipulador de transação passou para o processador de mudança uma TitanTransaction normal, a qual consulta o grafo Titan e faz alterações a ele.

12.1. CASOS DE USO DE LOG DE TRANSAÇÃO

12.1.1. REGISTRO DE MUDANÇA

O log de transações do usuário pode ser usado para manter um registro de todas as alterações feitas em um grafo. Ao usar identificadores de log separados, as mudanças podem ser gravadas em diferentes registos para distinguir tipos de transações separadas.

A qualquer momento, um processador de log pode ser construído, o qual pode processar todas as alterações registradas a partir da hora pretendida de início. Isto pode ser utilizado para a análise forense, para reproduzir as alterações em um grafo diferente, ou para computar um agregado.

12.1.2. ATUALIZAÇÕES DOWNSTREAM

É muitas vezes o caso de quando um cluster de um grafo Titan é parte de uma arquitetura maior. O log de transações do usuário e do framework de processador de log fornecem as ferramentas necessárias para transmitir alterações em outros componentes do sistema global sem afetar as operações originais causando a mudança. Isto é particularmente útil quando a latência de transação necessita ser baixa e / ou existem um número de outros sistemas que necessitam ser alertado quando ocorrer mudança no grafo.

12.1.3. TRIGGERS

O log de transações do usuário fornece a infraestrutura básica para implementar gatilhos que podem ser dimensionados para um grande número de transações simultâneas em grafos muito grandes. Um gatilho é registrado com uma mudança particular de dados disparando um evento em um sistema externo ou alterações adicionais ao grafo. Em escala, não é aconselhável implementar gatilhos na operação original, mas sim, um processo que desencadeia com um ligeiro atraso através do framework de processador de log. O segundo exemplo mostra como alterações no grafo podem ser avaliadas e provocar modificações adicionais.

12.1. CONFIGURAÇÃO DO LOG

Há uma série de opções de configuração para ajustar a forma como o processador de log lê a partir do log. Para configurar o log de transações do usuário, use o namespace log.user. As opções listadas não permitem a: configuração do número de threads a ser usada, o número de registros de log lido em cada lote, o intervalo de leitura, e se a transação de registros de mudança deve expirar automaticamente e ser removido do log após um período de tempo configurável (TTL).

13. PARTICIONAMENTO DE GRAFO (TITAN:DB)

Quando o cluster Titan consiste em várias instâncias de armazenamento de backend, o grafo pode ser dividido entre várias máquinas. Uma vez que o Titan armazena o grafo em uma representação de lista de adjacência, a atribuição de vértices para máquinas determina o particionamento.

Por padrão, o Titan utiliza uma estratégia de particionamento aleatório que atribui aleatoriamente vértices para máquinas. O Particionamento aleatório é muito eficiente, não requer nenhuma configuração e resulta em partições balanceadas. No entanto, particionamento aleatório resulta em um processamento de consultas menos eficiente quando o cluster do Titan cresce para acomodar mais dados, fazendo aumentar a necessidade de comunicação entre as instâncias para recuperar conjunto de resultados da consulta. Particionamento explícito de grafos pode garantir que subgrafos frequentemente percorridos sejam armazenados na mesma instância, reduzindo assim a sobrecarga de comunicação, significativamente.

Para ativar o particionamento grafo explícito no Titan, as seguintes opções de configuração devem ser definidas quando o cluster Titan é inicializado.

```
cluster.partition = true
cluster.max-partitions = 32
ids.flush = false
```

A opção de configuração max-partitions controla quantas partições virtuais o Titan cria. Este número deve ser cerca de duas vezes o número de instâncias de armazenamento de backend. Como certamente o cluster do Titan crescerá, é necessário estimar o tamanho do cluster no futuro e usar essa proporção como uma base. Definir esse número muito grande irá fragmentar desnecessariamente o cluster, o que pode levar a um mau desempenho.

Particionamento explícito só pode ser habilitado em infraestruturas de armazenamento que suporte armazenamento de chave ordena.

- HBase: suporta particionamento de grafos explícito
- Cassandra: Deve ser configurado para usar ByteOrderedPartitioner para suportar o particionamento de grafo explícito

Quando o grafo é pequeno ou acomodado em poucas instâncias de armazenamento, é melhor usar o particionamento aleatório pela sua simplicidade. Como regra geral, deve-se considerar fortemente permitido o particionamento explícito e configurar uma heurística de particionamento adequada, quando o grafo cresce em 10s de bilhões de arestas.

14. CRUD COM O EXEMPLO LUIZALABS (FATALA et al., 2014)

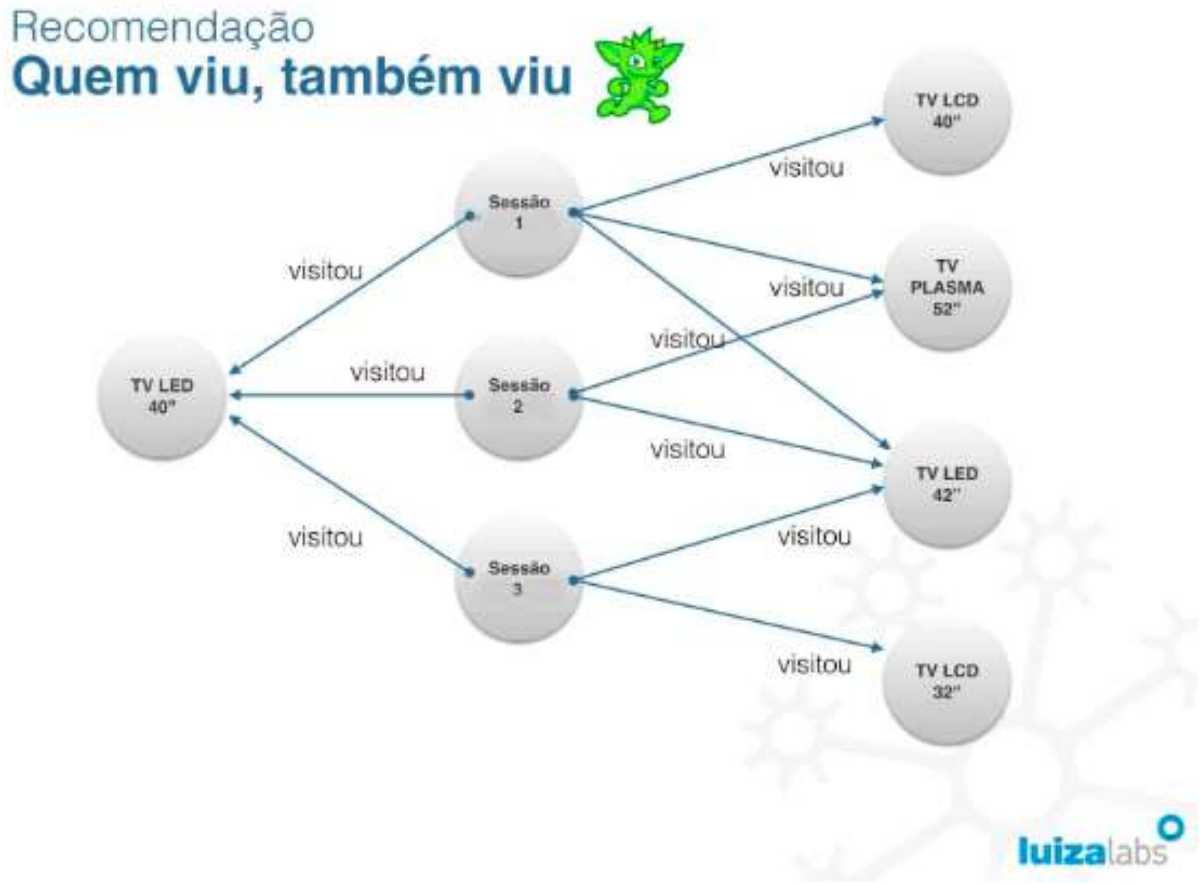

Figura 13: Grafo de exemplo LuizaLabs(FATALA et al., 2014)

Esse é um exemplo que ilustra um sistema de recomendação de produtos baseado em grafo (FATALA et al., 2014).

Partindo do vértice TV LED 40, o sistema gera uma recomendação de outros produtos, baseado no número de visitas a outros itens que as mesmas sessões também visualizaram.

```
//inicia o gremlin-server, cassandra e o elasticsearch
./bin/titan.sh start
//Inicia o gremlin shell
./bin/gremlin.sh
//Cria um grafo vazio usando cassandra como storage backend
gremlin>g = TitanFactory.open('cassandra:localhost')
==>standardtitangraph[cassandra:[localhost]
```

Outra opção de criação do grafo com o Cassandra:

```
gremlin>g = TitanFactory.build().set('storage.backend','cassandra').open()
==>standardtitangraph[cassandra:[127.0.0.1]]
```

```
//Adiciona os vértices
gremlin>sessao1 = g.addVertex(T.label, "sessao", "name", "sessao1")
gremlin>sessao2 = g.addVertex(T.label, "sessao", "name", "sessao2")
gremlin>sessao3 = g.addVertex(T.label, "sessao", "name", "sessao3")
gremlin>produto1 = g.addVertex(T.label, "produto", "name", "TV LED 40")
gremlin>produto2 = g.addVertex(T.label, "produto", "name", "TV LCD 40")
gremlin>produto3 = g.addVertex(T.label, "produto", "name", "TV PLASMA 52")
gremlin>produto4 = g.addVertex(T.label, "produto", "name", "TV LED 42")
gremlin>produto5 = g.addVertex(T.label, "produto", "name", "TV LCD 32")
//Carrega todo o grafo e o associa  a variável grafo
gremlin>grafo = g.traversal()
//Definindo as variáveis
gremlin>sessao1 = grafo.V().has('name', 'sessao1').next()
==>v[4320]
gremlin>sessao2 = grafo.V().has('name', 'sessao2').next()
==>v[4256]
gremlin>sessao3 = grafo.V().has('name', 'sessao3').next()
==>v[8312] //Vértices produtos
gremlin>tvLed40 = grafo.V().has('name', 'TV LED 40').next()
==>v[4192]
gremlin>tvLcd40 = grafo.V().has('name', 'TV LCD 40').next()
==>v[8288]
gremlin>tvPlasma52 = grafo.V().has('name', 'TV PLASMA 52').next()
==>v[4304]
gremlin>tvLed42 = grafo.V().has('name', 'TV LED 42').next()
==>v[4328]
gremlin>tvLcd32 = grafo.V().has('name', 'TV LCD 32').next()
==>v[4216]
//Checando os vértices
gremlin>grafo.V(tvLed40)
==>v[4192]
gremlin>grafo.V(tvLed40).valueMap()
==>[name:[TV LED 40]]
gremlin>grafo.V(tvPlasma52).valueMap()
```

```
==>[name:[TV PLASMA 52]]
//Adicionando arestas que saem do vertice sessao1
gremlin>sessao1.addEdge('visitou', tvLed40)
==>e[odxcs-3c0-27th-38g][4320-visitou->4192]
gremlin>sessao1.addEdge('visitou', tvLcd40)
==>e[1cruak-3c0-27th-6e8][4320-visitou->8288]
gremlin>sessao1.addEdge('visitou', tvPlasma52)
==>e[1cruos-3c0-27th-3bk][4320-visitou->4304]
gremlin>sessao1.addEdge('visitou', tvLed42)
==>e[1crv30-3c0-27th-3c8][4320-visitou->4328]
//Arestas da sessao2
gremlin>sessao2.addEdge('visitou', tvLed40);
==>e[odxck-3a8-27th-38g][4256-visitou->4192]
gremlin>sessao2.addEdge('visitou', tvLed42);
==>e[odxqs-3a8-27th-3c8][4256-visitou->4328]
gremlin>sessao2.addEdge('visitou', tvPlasma52);
==>e[ody50-3a8-27th-3bk][4256-visitou->4304]
//Arestas da sessao3
gremlin>sessao3.addEdge('visitou', tvLed40);
==>e[odxcf-6ew-27th-38g][8312-visitou->4192]
gremlin>sessao3.addEdge('visitou', tvLed42);
==>e[odxqn-6ew-27th-3c8][8312-visitou->4328]
gremlin>sessao3.addEdge('visitou', tvLcd32);
==>e[ody4v-6ew-27th-394][8312-visitou->4216]
//Grafo completo 8 vértices e dez arestas
gremlin>grafo.V().valueMap();
==>[name:[sessao1]]
==>[name:[sessao2]]
==>[name:[sessao3]]
==>[name:[TV PLASMA 52]]
==>[name:[TV LED 40]]
==>[name:[TV LCD 32]]
==>[name:[TV LED 42]]
==>[name:[TV LCD 40]]
```

```
gremlin>grafo.V().count()
==>8
gremlin>grafo.E().count()
==>10
//Quem visitou tvLed40?
gremlin> grafo.V(tvLed40).in('visitou').valueMap()
==>[name:[sessao2]]
==>[name:[sessao1]]
==>[name:[sessao3]]
//Quais produtos a sessao1 visitou?
gremlin> grafo.V(sessao1).out('visitou').valueMap()
==>[name:[TV LED 40]]
==>[name:[TV PLASMA 52]]
==>[name:[TV LED 42]]
==>[name:[TV LCD 40]]
//Recomendação
gremlin>grafo.V(tvLed40).in('visitou').out('visitou').where(is(neq(tvLed40)))valueMap().groupCount()
==>[[name:[TV LED 42]]:3, [name:[TV LCD 40]]:1, [name:[TV LCD 32]]:1, [name:[TV PLASMA 52]]:2]
//Para exportar o grafo em JSON
gremlin> g.io(graphson()).writeGraph('/tmp/grafo_magazine.json')
//Para importar de JSON
gremlin> novoGrafo.io(IoCore.graphson()).readGraph('/vagrant/grafo_magazine.json');
```

15. CONCLUSÃO

Embora o Titan DB seja bastante utilizado hoje, a Datastax, empresa que o comprou em 2015 descontinuou o projeto, hoje ele é a base para outros projetos da Datastax(CHAN, 2016).

A partir do conhecimento tecnológico adquirido com o Titan e o time de desenvolvedores, a Datastax desenvolveu o DSE (DataStax Enterprise), um banco comercial orientado a grafo, que na verdade é o próprio Titan DB modificado, com algumas facilidades de configurações e uma integração maior com o cassandra, mas, com limitações também em relação ao Titan, que oferece mais flexibilidade e liberdade em relação as engines de backend.

O projeto ainda é aberto e pode ser usado e modificado pelos desenvolvedores, de acordo com as conveniências dos projetos.

REFERÊNCIAS

AHOMOLYA, 27 de novembro de 2014, às 10:36 horas, **Hack Day Story: Introducing Titan DB**. Disponível em: <http://tech.kinja.com/hack-day-story-introducing-titan-db-1664113385>. Acesso em: 10 de julho de 2016, às 17:20 horas.

BROECHELER, MATTHIAS, 08 de agosto de 2012, Diretor-Chefe de Tecnologia, CTO, e co-fundador da DataStax, **TITAN BIG GRAPH DATA WITH CASSANDRA**.

BROECHELER, MATTHIAS, 14 de junho de 2013, Diretor-Chefe de Tecnologia, CTO, e co-fundador da DataStax, **C* Summit 2013: Distributed Graph Computing with Titan and Faunus by Matthias Broecheler**. Disponível em: <http://pt.slideshare.net/planetcassandra/distributed-graph-computing-with-titan-and-faunus>. Acesso em: 02 de agosto de 2016.

BROECHELER, MATTHIAS, 10 de maio de 2013, Diretor-Chefe de Tecnologia, CTO, e co-fundador da DataStax, **5 Minute C* Interview | Titan Graph Database**, Disponível em: <http://www.planetcassandra.org/blog/5-minute-c-interview-titan-graph-database/>. Acesso em: 15 de setembro de 2016.

CHAN, WAYNE, 13 de março de 2015, **DataStax Blog - Q&A – Cassandra and TitanDB Insights into DataStax's Graph Strategy**. Disponível em: <http://www.datastax.com/2015/03/qa-cassandra-and-titandb-insights-into-datastaxs-graph-strategy>. Acesso em: 20 de setembro de 2016, às 13:03 horas.

NONNEN, JAN, 13 de janeiro de 2016, **INTRODUCTION TO THE TITAN GRAPH DATABASE** . Disponível em: < http://www.viaboxx.de/uncategorized/introduction-to-the-titan-graph-database/ >. Acesso em: 11 de julho de 2016, às 22:39 horas.

TITAN:DB, **Documentação oficial do Titan.** Disponível em: http://s3.thinkaurelius.com/docs/titan/>. Acesso em: Julho de 2016.

RODRIGUEZ, MARKO A.; LAROCQUE, DAN; BROECHELER, MATTHIAS, 13 de maio de 2013, **Educating the Planet with Pearson.** Disponível em: < https://thinkaurelius.com/2013/05/13/educating-the-planet-with-pearson/ >. Acesso em: 10 de julho de 2016.

NAVATHE, SHAMKANT B.; ELMASRI, RAMEZ, 2005**, Sistemas de Banco de Dados** - 4°edição, Editora: Pearson.

WOLPE, T., 3 de fevereiro de 2015, **ZDNet Articles,** Disponível em: <http://www.zdnet.com/article/datastax-snaps-up-aurelius-and-its-titan-team-to-build-new-graph-database/>. Acesso em: 10 de julho de 2016, arquivado no curso de Engenharia de Produção na UNIMAR.

DIAS, MIKE; 8 de julho de 2016; **Grafos com TinkerPop3 e TitanDB - TDC2016**, Disponível em: <https://speakerdeck.com/mikedias/grafos-com-tinkerpop3-e-titandb-tdc2016>. Acesso em: 02 de agosto de 2016.

FATALA, ANDRÉ; PEDIGONE, RENATO; 27 de julho de 2014; **Um Sistema de recomendação de produtos baseado em grafos: Titan, Cassandra, Redis e Hadoop em produção**, Disponível em: <https://www.infoq.com/br/presentations/sistema-de-recomendacao-baseado-em-grafos>. Acesso em: 02 de agosto de 2016.