Avaliação de desempenho de gerenciadores de bancos de dados multi-modelo em aplicações com persistência poliglota

Fábio Roberto Oliveira

Maio / 2017

Dissertação de Mestrado em
Ciência da Computação

Avaliação de desempenho de gerenciadores de bancos de dados multi-modelo em aplicações com persistência poliglota

Esse documento corresponde a Dissertação a ser apresentada à Banca Examinadora no curso de Mestrado em Ciência da Computação da Faculdade Campo Limpo Paulista.

Campo Limpo Paulista, **17 de Maio de 2017**.

Fábio Roberto Oliveira

Prof. Dr. Luis Mariano Del Val Cura (Orientador)

Ficha catalográfica

Dados Internacionais de Catalogação na Publicação (CIP)

Câmara Brasileira do Livro, São Paulo, Brasil.

Oliveira, Fábio Roberto

Avaliação de desempenho de gerenciadores de bancos de dados multi-modelo em aplicações com persistência poliglota / Fábio Roberto Oliveira. Campo Limpo Paulista, SP: FACCAMP, 2017.

Orientador: Profº. Dr. Luis Mariano Del Val Cura.
Dissertação (Programa de Mestrado em Ciência da Computação) – Faculdade Campo Limpo Paulista – FACCAMP.

1. Desempenho. 2. Banco de dados. 3. Multi-modelo. 4. NoSQL. 5. Persistência. 6. Poliglota. I. Del Val Cura, Luis Mariano. II. Campo Limpo Paulista. III. Título.

CDD-005.75

Folha de aprovação

Faculdade Campo Limpo Paulista – FACCAMP

Avaliação de desempenho de gerenciadores de bancos de dados multi-modelo em aplicações com persistência poliglota

Fábio Roberto Oliveira

Campo Limpo Paulista, 17 de Maio de 2017.

BANCA EXAMINADORA

Prof. Dr. Luis Mariano Del Val Cura

(Orientador – FACCAMP)

Prof. Dr. Eduardo Javier Huerta Yero

(FACCAMP)

Prof. Dra. Cristina Dutra de Aguiar Ciferri

(USP)

Agradecimentos

Agradeço a minha esposa Débora, meus pais Edison e Rosely, e meu irmão Fernando por todo o apoio e entusiasmo que sempre tiveram com meus estudos.

Agradeço ao meu orientador Dr. Mariano Del Val Cura pela atenção, paciência, e dedicação que conduziu minha orientação.

Agradeço o professor Dr. Osvaldo Luiz de Oliveira que lecionou várias disciplinas importantes quando eu ainda estava começando na graduação, além de sua excelente coordenação e iniciativa no curso de mestrado.

Agradeço a todos os meus amigos que fiz no decorrer do curso.

Resumo:

Os sistemas de gerenciamento de banco de dados NoSQL foram introduzidos recentemente como alternativas aos sistemas de gerenciamento de banco de dados relacionais tradicionais. Estes sistemas implementam modelos de dados mais simples e escaláveis que aumentam a eficiência e o desempenho de uma nova geração de sistemas com alta exigência de acesso e escalabilidade. Novos tipos de aplicações que modelam seus dados usando dois ou mais modelos de dados NoSQL são conhecidas como aplicações com persistência poliglota. Normalmente suas implementações são complexas porque elas devem gerenciar e armazenar seus dados utilizando simultaneamente vários sistemas de gerenciamento de bancos de dados. Recentemente, foi introduzida uma nova família de sistemas de gerenciamento chamados de multi-modelo que integram vários modelos de dados NoSQL em um único sistema. Esta dissertação apresenta uma avaliação do desempenho dos gerenciadores de bancos de dados multi-modelo quando utilizados em aplicações com persistência poliglota. Para essa avaliação, foram aplicados um conjunto de testes (benchmark) simulando uma aplicação com persistência poliglota executando operações básicas em bancos de dados. Estes testes foram aplicados nos gerenciadores de bancos de dados de modelo simples e multi-modelos. Os resultados demostram que em certos cenários os gerenciadores multi-modelo apresentam desempenho similar ou melhor que gerenciadores de modelo simples.

Abstract

NoSQL data store systems have recently been introduced as alternatives to traditional relational database management systems. These data stores systems implement simpler and more scalable data models that increase the performance and efficiency of a new kind of emerging scalability database application. Applications that model their data using two or more simple NoSQL models are known as applications with polyglot persistence. Usually, their implementations are complex because they must manage and store their data using several data store systems simultaneously. Recently, a new family of multi-model data stores was introduced, integrating simple NoSQL data models into a single unique system. This paper presents a performance evaluation of multi-model data stores used by an application with polyglot persistence. In this research, multi–model datasets were synthesized in order to simulate that application. We evaluate the performance of benchmarks based on a set of basic database operations on single model and multi-model data store systems. Experimental results show that in some scenarios multi-model data stores have similar or better performance than simple model data stores.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1. Introdução

Um dos mais importantes desafios para a comunidade de pesquisadores de bancos de dados tem sido o desenvolvimento da tecnologia para manipular expressivos volumes de dados heterogêneos, gerados por aplicações e pessoas (Stonebraker, 2010).

Atualmente os gerenciadores de bancos de dados relacionais apresentam dificuldades na escalabilidade e distribuição de dados para manipular aplicações que trabalham com *Big Data* (Paolo, et al., 2013). Para suprir essa demanda, uma nova classe de gerenciadores de bancos de dados que não trabalham com o modelo relacional surgiram, sendo classificados sobre o termo guarda-chuva NoSQL.

Um recurso comum destes novos gerenciadores é que eles suportam modelos de dados especializados, que são eficientes na escalabilidade e distribuição de dados.

Os principais modelos NoSQL são documentos, pares chave-valor, orientado a colunas e orientado a grafos (Stonebraker, 2010), mas estes modelos não são padronizados e formalizados como o modelo relacional

Atualmente há mais de duzentas implementações de bancos de dados NoSQL implementando diferentes tipos de modelos de dados (Edlich, 2016). A questão de escolha sobre qual destes gerenciadores adotar é uma decisão complexa para arquitetos de software, principalmente porque cada um trabalha com um modelo de dados, possuem linguagens de consulta que não são padronizadas e também possuem interfaces de acesso diferentes. Um fator de decisão frequente é o desempenho dos gerenciadores existentes para cada modelo. Por outro lado, é comum que uma aplicação não se adéque totalmente a um único modelo de dados. Como consequência, essas aplicações são mais facilmente modeladas e obtêm melhor desempenho se utilizarem vários modelos de dados e vários gerenciadores de bancos de dados simultaneamente. O termo *persistência poliglota* descreve as aplicações com este tipo de necessidades e requisitos. (Fowler, 2015)

A principal vantagem da adoção da persistência poliglota é ter o melhor desempenho nas aplicações. Este melhor desempenho é o resultado de modelar diferentes componentes ou requisitos do sistema segundo o modelo de dados que é mais apropriado para cada caso, ou seja, que executa mais eficientemente as requisições. Neste caso, a aplicação precisa acessar diferentes gerenciadores de bancos de dados que implementam os modelos de dados utilizados. Os problemas ou desvantagens do uso de persistência poliglota estão relacionados fundamentalmente com a complexidade que introduz no processo de gerenciamento e desenvolvimento destas aplicações. Dentre os problemas podemos citar os seguintes: replicação de dados entre vários gerenciadores de bancos de dados, gerenciamento complexo de transações que utilizam vários sistemas, uso de diferentes interfaces de aplicação e ferramentas para cada sistema, dentre outros.

Uma recente alternativa que simplifica estes problemas tem sido a utilização de uma nova classe de sistemas gerenciadores de bancos de dados que integram vários modelos de dados. Estes sistemas são chamados de *gerenciadores NoSQL multi-modelo.*

A introdução deste tipo de gerenciadores multi-modelo realmente simplifica o processo de gerenciamento e desenvolvimento de aplicações com persistência poliglota. No entanto, uma questão levantada é qual o impacto no desempenho destas aplicações da utilização destes gerenciadores como alternativa.

O objetivo desta dissertação é contribuir e pesquisar este impacto através da avaliação experimental do desempenho de aplicações poliglotas, com e sem o uso destes gerenciadores multi-modelo.

Para esta pesquisa foi desenvolvido um sistema que simula o comportamento de uma aplicação com persistência poliglota. Este sistema realiza operações de armazenamento e consulta de forma intensiva, sobre bancos de dados de testes que utilizam ambos os modelos de documentos e grafos. Estes bancos de dados são armazenados segundo duas alternativas: utilizando simultaneamente os gerenciadores do modelo de grafos Neo4j e do modelo de documentos MongoDB, ou utilizando unicamente os gerenciadores NoSQL multi-modelo ArangoDB e OrientDB. Os gerenciadores multi-modelo ArangoDB e OrientDB gerenciam ao mesmo tempo o modelo de documentos, chave-valor e o

de grafos. Eles foram escolhidos porque o código fonte destes produtos está disponível para inspeção, cada um deles implementa o modelo de grafos com abordagem diferente, e todos trabalham com acesso por interface webservice REST (Fielding, 2000).

Nesta dissertação é proposto um *benchmark* com uma série de testes para comparar o desempenho da aplicação poliglota simulada, quando utilizadas as duas abordagens apresentadas anteriormente. Para a realização dos experimentos foi utilizado um banco de dados de testes da empresa *Amazon* e foram gerados dois bancos de testes de forma sintética. Todos estes bancos armazenam seus dados segundo os modelos de documentos e de grafos.

A contribuição fundamental desta dissertação foi a avaliação experimental da conveniência do uso de gerenciadores de bancos de dados NoSQL multi-modelos em aplicações poliglotas. Os resultados permitiram levantar hipóteses sobre quais os cenários em que cada uma das alternativas propostas consegue oferecer o melhor desempenho. Adicionalmente, foi possível avaliar o impacto dos modelos de representação e implementação utilizados por estes gerenciadores no desempenho dos diferentes tipos de consulta.

O restante da dissertação está organizado conforme os capítulos a seguir:

No capítulo 2 será abordado o modelo relacional de dados, as novas aplicações Big Data e os problemas do modelo relacional para estas aplicações.

O capítulo 3 apresenta os modelos de dados NoSQL e os sistemas de gerenciamento que os implementam.

No capítulo 4 são apresentadas as aplicações com persistência poliglota, os gerenciadores de bancos da dados multi-modelos e exemplos de seu uso na implementação destas aplicações.

No capítulo 5 é apresentada a aplicação desenvolvida neste trabalho para realizar testes de desempenho de aplicações com persistência poliglota.

No capítulo 6 são apresentados os diferentes testes de desempenho *(benchmark)* de aplicações poliglotas, assim como os resultados destes testes e as principais conclusões da pesquisa realizada.

No capítulo 7 são apresentadas as considerações finais e as propostas de trabalhos futuros.

Capítulo 2. Modelo relacional de Banco de Dados e aplicações de Big Data

Este capítulo tem como foco o modelo relacional que tem sido o modelo dominante em aplicações de bancos de dados nos últimos 30 anos, bem como as aplicações Big Data. Serão abordados os seguintes tópicos: a origem do modelo relacional, o que é um sistema gerenciador de banco de dados, a integridade do modelo relacional, as vantagens e desvantagens deste modelo, as operações que esse modelo proporciona, aplicações Big Data, suas características, os problemas que o modelo relacional enfrenta ao trabalhar com Big Data e finalmente a necessidade de uma nova geração de bancos de dados.

2.1 Modelo relacional

A partir de 1960 os primeiros gerenciadores de bancos de dados começaram substituir a tecnologia disponível na época, que eram os primitivos sistemas de arquivos.

As primeiras versões dos gerenciadores de bancos de dados trabalhavam com modelos conhecidos como hierárquicos e de rede. O modelo hierárquico foi o primeiro a ser considerado como um modelo de dados, sendo que sua implementação somente foi possível devido a popularização dos dispositivos de discos de armazenamento, pois esses dispositivos possibilitaram a exploração de sua estrutura de endereçamento físico, tornando possível a representação hierárquica das informações.

O modelo em redes surgiu como uma melhora do modelo hierárquico, eliminando o conceito de hierarquia, permitindo que um mesmo registro estivesse envolvido em várias associações.

Considerando as deficiências dos modelos hierárquicos e de rede, na década de 1970, Edgar Codd (Codd, 1970) do Laboratório de Pesquisa da IBM fez uma proposta de um novo modelo de representação de dados, chamado modelo de dados relacional. Esse modelo teve uma ampla e rápida adoção por ser fundamentado na teoria matemática de relação e com formalismos para a consulta dos dados baseados em lógica de predicados e teoria dos conjuntos.

Em 1972 e 1977, dois gerenciadores de bancos de dados relacionais foram lançados: Ingres da Universidade de Berkley e System R da empresa IBM respectivamente. Como o modelo relacional apresentava benefícios e supria as

deficiências dos modelos anteriores, ele tornou-se um padrão de gerenciador de banco de dados para as próximas décadas, com muitos produtos sendo oferecidos a partir de 1980 (Ramakrishnan e Gehrke, 2008).

Apesar do modelo relacional se tornar um padrão, os modelos hierárquico e de rede, continuaram adotados em nichos específicos.

2.1.1 Definição do modelo relacional

No estudo apresentado por (Codd, 1970), três objetivos iniciais foram definidos para o modelo relacional:

- Independência dos dados: definição clara dos limites entre os aspectos físicos e lógicos de um gerenciador de banco de dados.
- Legibilidade: permitir um modelo estrutural simples de forma que qualquer usuário pudesse entender os dados e se comunicar com o banco de dados.
- Linguagem de alto nível: existência de uma linguagem que permitisse a manipulação de um conjunto de dados através de comandos padronizados.

Uma base de dados relacional é formada por relações estruturadas na forma de tabelas, e com relacionamentos entre essas tabelas (Silberschatz, et al., 2012). Uma *tabela (relação)* é uma estrutura formada por *linhas (tuplas)* e *colunas (atributos)*, onde são armazenados os dados. Cada linha representa um registro dentro da base de dados sendo formada por valores para um conjunto de atributos. Não é obrigatório um registro ter dados em todos os atributos (colunas), isto é, podem existir atributos com valor nulo, desde que não seja definida uma restrição. Para cada atributo também existe um conjunto de valores permitidos, denominado como *domínio.* É possível também identificar de forma única um registro numa tabela através do conjunto de atributos que são designados como *chaves candidatas*. Do conjunto de chaves candidatas mínimas, uma é eleita como *chave primária.* O relacionamento entre os registros (tuplas) de uma base de dados do modelo relacional é feito através de *chaves estrangeiras*, e essas chaves podem ser formadas por um ou mais atributos. Uma chave estrangeira na verdade faz referência a uma chave primária (ou chave candidata) de outra tabela.

Na figura 1 temos um exemplo de três tabelas (Produtos, Clientes e Movimentos) com a chave primária de cada tabela sendo o atributo id, com sua chave estrangeira relacionando com o id_produto e id_cliente.

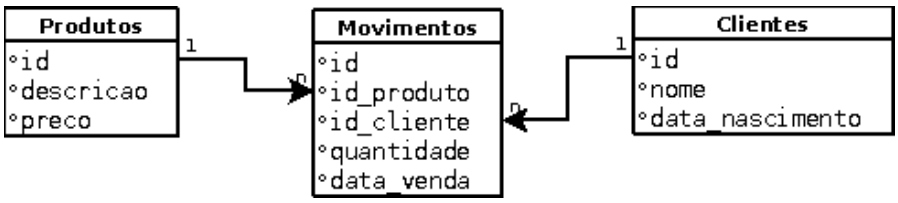

Figura 1. Exemplo de mapeamento relacional de três tabelas

2.1.2 Consultas no modelo relacional

As operações de consulta no modelo relacional são definidas através de formalismos matemáticos, utilizando a álgebra relacional e cálculo relacional.

A seguir serão abordados a álgebra relacional, o cálculo relacional de tuplas e a linguagem SQL que é um padrão para o modelo relacional.

a) Álgebra Relacional

No caso da álgebra relacional, ela é composta por um conjunto de operadores que podem ser unários ou binários (Date, 2015). A seguir são apresentadas as principais operações do modelo relacional, onde essas operações podem ser visualizadas melhor no exemplo da figura 2.

- **Seleção ou Restrição**

Obtém uma nova relação contendo todas as linhas (tuplas) de uma relação específica que satisfaçam a uma condição especificada, isto é, é aplicado uma condição na consulta ao ler as tuplas de uma relação.

Notação - σ predicado (relação)

Exemplo: σ atributo = 'conteúdo' (Relação)

- **Projeção**

Obtém uma nova relação contendo todas as tuplas que permaneçam em uma relação especificada, depois que atributos especificados foram descartados, isto é, apenas colunas selecionadas são apresentadas.

Notação: π lista_nome_atributos (Relação)

Exemplo: π atributo (Relação)

- **Produto**

Obtém uma nova relação contendo todas as possíveis tuplas que sejam uma combinação de duas tuplas, uma de cada duas relações especificadas. Esse operador também é conhecido como produto cartesiano ou produto cruzado.

Notação : relação1 x relação2 (R1 x R2)

Exemplo: π atributo1, atributo2, atributo3 (σ Relação1.atributo = Relação2.atributo (Relação1 x Relação2))

- **Intersecção**

Obtém uma relação contendo todas as tuplas que apareçam em ambas as relações determinadas.

Notação : relação1 ∩ relação2 (R1 ∩ R2)

Exemplo: Resultado : Relação1 ∩ Relação2

- **União**

Obtém uma relação contendo todas as tuplas que apareçam em qualquer uma das duas, ou em ambas as relações determinadas.

Notação: Relação1 ∪ Relação2 (R1 ∪ R2)

Exemplo: resultado: Relação1 ∪ Relação2

- **Diferença**

Obtém uma relação contendo todas as tuplas que apareçam na primeira, mas não na segunda das duas relações determinadas.

Notação : relação1 - relação2 (R1 - R2)

Exemplo: Resultado : Relação1 – Relação2

- **Junção**

Obtém uma relação contendo todas as tuplas possíveis que sejam uma combinação de duas tuplas, uma de cada duas relações especificadas, de forma

que as duas tuplas que contribuam para qualquer tupla de resultado tenham um valor comum para os atributos comuns das duas relações determinadas. Essa operação permite fazer a junção de várias relações na forma de colunas respeitando as chaves selecionadas para a operação de junção.

Notação: R1 |x| R2

Exemplo: σ <critério> (<relação1> X <relação2)

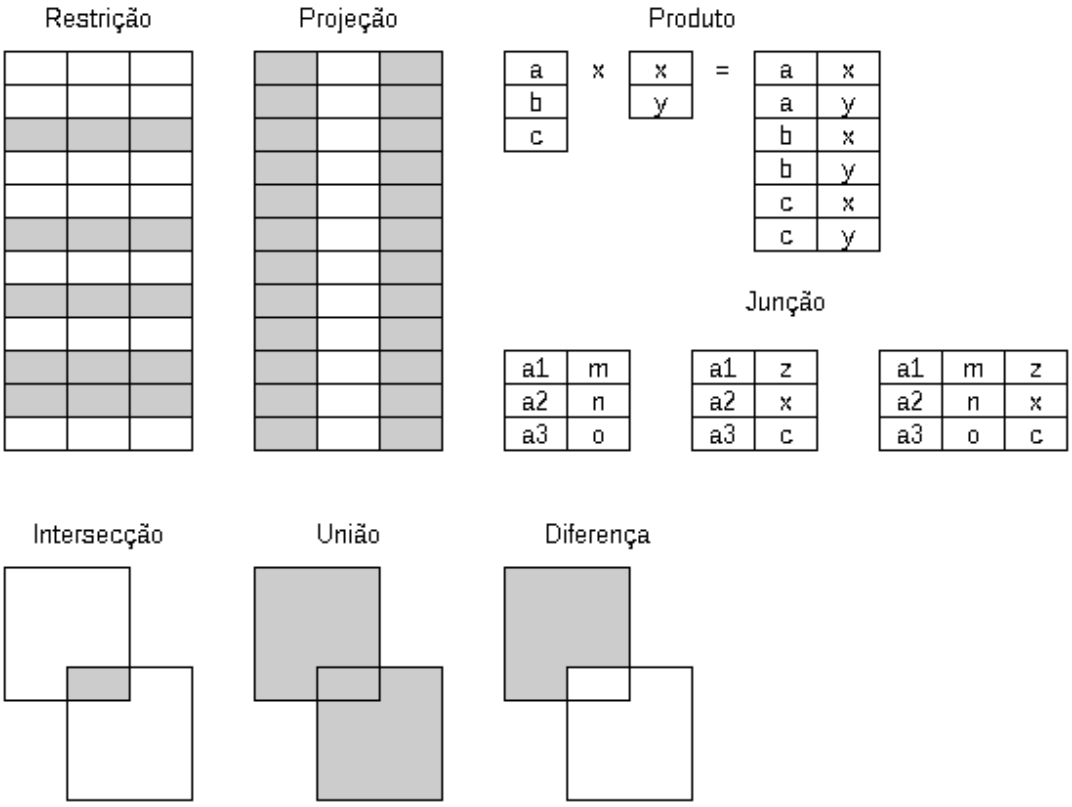

Figura 2. Operadores possíveis na álgebra relacional

b) Cálculo relacional de tuplas

O Cálculo relacional é uma linguagem de consulta que descreve a resposta desejada sobre um banco de dados sem especificar como obter esses dados, ou seja, trabalha como uma linguagem funcional.

Ele difere da Álgebra relacional que utiliza a abordagem de consulta procedural, mas ambas as abordagens conseguem os mesmos resultados. (Date, 2015)

- **Relação com a lógica de primeira ordem:**

Podemos definir uma fórmula com base a combinações de fórmulas atômicas.

Uma fórmula atômica é uma combinação de variáveis (tipo tupla ou tipo domínio) e atributos ou constantes, graças ao uso dos operadores: <, >, =,!=, <=, >=

Também é uma fórmula atômica variável ∈ Relação.

As combinações de fórmulas atômicas são geradas a partir do uso de operadores como NOT (¬), AND (∧), OR (∨), →.

Os qualificadores ∃, ∀ limitam uma variável.

- **Expressão geral**: { t | COND(t) }

  Onde t = variável de tupla e COND(t) = é uma expressão condicional envolvendo t

- **Resultado da consulta** = conjunto de todas as tuplas t que satisfazem COND(t)

Exemplo: encontrar todas as tuplas do funcionário que ganhe mais que 1750,00

Notação Funcionário: { t | empregado(t) AND t.salario > 1750}

Resultado: desejamos obter o conjunto de todas as tuplas t tal q exista uma tupla t na relação empregado, considerando que o atributo salário seja maior que 1750 reais.

c) Linguagem de consulta SQL

Um dos fatores que tem motivado o sucesso do modelo relacional, é a formalização matemática do processo de consulta através da álgebra e cálculo relacionais. Na década dos anos 70, surgiram múltiplas linguagens de consultas baseadas nestes formalismos, sendo que o SQL teve maior sucesso principalmente porque foi adotada por grandes corporações como a IBM. Ela se tornou padrão de linguagem de consulta e manipulação de bancos de dados relacionais. Segundo (Silberschatz, et al., 2012) a linguagem SQL é dividida em

grupos de operações, onde cada grupo é responsável por um conjunto de operações comuns. Os grupos são:

- **Linguagem de Definição de Dados (DDL)**

É o grupo que contém um conjunto de comandos dentro da linguagem SQL que tem como objetivo definir as estruturas do banco de dados. Essas operações permitem a criação, modificação e remoção das tabelas, assim como criação de índices. Estas operações SQL permitem definir a estrutura de uma base de dados, incluindo as linhas, colunas, tabelas, índices, e outros metadados.

Os principais comandos são: CREATE (Criar), DROP (deletar) e ALTER (alterar).

- **Linguagem de Manipulação de Dados (DML)**

É o grupo de comandos dentro da linguagem SQL utilizado para a recuperação, inclusão, remoção e modificação de informações em bancos de dados.

Os principais comandos são: SELECT (Seleção), INSERT (Inserção), UPDATE (Atualização) e DELETE (Exclusão).

- **Linguagem de Controle de Dados (DCL)**

É o grupo de comandos que permite ao administrador de banco de dados gerenciar o acesso aos dados deste banco. Alguns exemplos de comandos DCL são:

- GRANT: Permite dar permissões a um ou mais usuários em objetos do banco de dados.

- REVOKE: Retira as permissões dadas pelo comando GRANT.

As operações básicas que podemos conceder ou negar nas permissões são: SELECT, INSERT, UPDATE e DELETE.

2.1.3 Gerenciamento de transações

Uma importante funcionalidade de um sistema de gerenciamento de banco de dados, geralmente associada ao modelo relacional é o gerenciamento de transações. Segundo (Elmasri e Navathe, 2011) uma transação pode ser

definida como um conjunto de operações que possuem uma função específica para a aplicação do sistema de banco de dados, isto é, uma transação representa um conjunto de operações de leitura ou escrita que são realizadas de forma conjunta no banco de dados. A execução de transações deve obedecer a algumas propriedades a fim de garantir o sistema funcionando corretamente em relação à consistência dos dados. Estas propriedades são chamadas de propriedades ACID e são definidas a seguir:

- **Atomicidade:** Todas as operações da transação são executadas, ou seja, a transação é executada por completo, ou nada é executado.
- **Consistência:** Após uma transação ter sido finalizada, o banco de dados deve continuar em um estado consistente, ou seja, deve satisfazer as condições de consistência e restrições de integridade previamente assumidas.
- **Isolamento:** Se duas transações estão sendo executadas concorrentemente, o efeito de uma não pode interferir no efeito da outra, isto é, o resultado deve ser equivalente à execução isolada de uma transação com relação à outra. Esta propriedade está relacionada ao controle de concorrência do SGBD permitindo níveis de isolamento definidos na aplicação.
- **Durabilidade:** Uma vez que uma transação finalizou com sucesso, seu efeito não poderá mais ser desfeito, mesmo em caso de falha. Esta propriedade está relacionada à capacidade de recuperação de falhas do SGBD.

2.1.4 Bancos de dados relacionais distribuídos

Segundo (Casanova, 2012) pode-se definir bancos de dados distribuídos (BDD) como um conjunto de vários bancos de dados logicamente relacionados, mas separados em diferentes localidades físicas, distribuídos por uma rede de computadores. Um sistema de gerenciamento de bancos de dados distribuídos (SGBDD) tem como definição um sistema que possibilita o gerenciamento dos bancos de dados distribuídos e que abstrai a distribuição dos dados para os usuários, ou seja, é como se fosse um sistema centralizado. Um dos fatores de

sucesso de um banco de dados distribuído, é abstrair completamente todas as questões de particionamento de rede dos usuários.

Podemos citar três vantagens dos bancos de dados distribuídos. A primeira vantagem é a melhoria na confiabilidade, a segunda maior é a disponibilidade e a terceira a alta escalabilidade. A confiabilidade é a probabilidade de que um sistema continue operacional no decorrer do tempo. A disponibilidade é a probabilidade de um sistema estar disponível em um instante de tempo. A diferença entre disponibilidade e confiabilidade é que a disponibilidade leva em conta um instante de tempo, e a confiabilidade é medida em um intervalo de tempo. Quando os dados e o software do SGBD são distribuídos por várias localidades, uma localidade pode apresentar falhas, fazendo com que outras não possam ser acessadas, portanto a distribuição melhora a confiabilidade e a disponibilidade.

Como inverso do sistema distribuído, que é o ambiente centralizado, qualquer falha em um servidor torna o sistema inteiro indisponível para todos os usuários. Uma característica importante de um banco de dados distribuído é que se alguns dos servidores ficarem indisponíveis, outros usuários ainda podem acessar outras partes do banco de dados.

O termo escalabilidade tem como definição a capacidade de um sistema atender uma demanda crescente de carga de processamento, não comprometendo a performance ou o tempo de resposta para os usuários de forma severa, ou seja, conseguir atender essa variação de demanda de forma eficiente.

- **Tipos de bancos de dados distribuídos:**

Segundo (Fachin, 2007) podemos classificar os bancos de dados distribuídos em dois tipos, o SGBDD homogêneo e o SGBDD heterogêneo. No SGBDD homogêneo, os SGBDs locais são idênticos em todas as localidades com a mesma estrutura do banco de dados. Já no SGBDD heterogêneo, podem existir dois ou mais SGBDs diferentes espalhados entre as localidades, sendo que a estrutura do banco de dados também pode ser diferente. Podem existir diversos servidores e cada um deles pode conter um ou mais SGBDs locais.

Quando o banco de dados é heterogêneo, os dados são particionados, opção conhecida também por fragmentação

- **Tipos de fragmentação**

Segundo (Fachin, 2007), os fragmentos do banco de dados podem ser do tipo horizontal, vertical ou híbrido.

- Fragmentação horizontal:

Uma relação é dividida em subconjuntos de tuplas, sendo que cada tupla pertence a um fragmento, onde cada fragmento é associado a um servidor. Esse tipo de fragmentação divide as linhas de uma tabela em vários servidores.

- Fragmentação vertical:

Uma relação é dividida em subconjuntos de atributos diferentes resultando em relações menores que pertencem a um fragmento. Cada fragmento acaba sendo associado a um servidor. Esse tipo de fragmentação divide as colunas de uma tabela em vários servidores, sendo que é obrigatório que as colunas chaves da tabela estejam em todos os servidores também.

- Fragmentação híbrida:

Uma relação é dividida em vários fragmentos e cada fragmento é o resultado da fragmentação horizontal e vertical. Esse tipo de fragmentação divide as colunas de uma tabela em vários servidores, permitindo também que as linhas dessas tabelas também estejam particionadas nesses servidores, portanto é uma combinação de ambos tipos de fragmentação.

a) Controle de transações distribuídas (Protocolo Two-Phase Commit)

O gerenciamento de transações em sistemas de bancos de dados distribuídos precisa considerar a fragmentação dos dados explicada anteriormente. Para garantir as propriedades ACID em sistemas distribuídos foi introduzido o protocolo two-phase commit necessário para coordenar transações em múltiplos servidores.

O funcionamento desse protocolo trabalha obrigatoriamente em duas fases: preparação e resolução. Em cada transação, um processo age como coordenador. Esse processo coordenador supervisiona as atividades dos outros

participantes na transação com o objetivo de garantir um resultado consistente. Na fase de preparação, o coordenador manda uma mensagem para cada processo na transação, solicitando para cada processo preparar-se para confirmar. Quando um processo é preparado, ele garante que pode consolidar a transação criando um registro permanente de seu trabalho. Depois de garantir que pode confirmar, ele não pode mais optar de forma isolada por qualquer tentativa de abortar a transação. Se um processo não conseguir garantir que pode confirmar a transação, ele deverá abortar. Na fase de resolução, o coordenador registra as respostas. Se todos os participantes estiverem preparados para confirmar, a transação será confirmada. Caso contrário, a transação será abortada.

Em qualquer caso, o coordenador obrigatoriamente informa a todos os participantes do resultado. No caso de uma confirmação, os participantes reconhecem sua confirmação, fazendo com que as alterações confirmadas sejam persistidas. Isso garante que uma transação que foi bem sucedida seja refletida como uma alteração permanente. (Ramakrishnan e Gehrke, 2008)

- **Descrição dos passos do algoritmo:**

  - Primeira fase:

O coordenador envia uma mensagem requisitando o commit a todos os participantes. Cada participante responde com seu voto ao coordenador sobre sua opção de fazer commit. Se por algum motivo um participante não consegue executar a sua parte da transação localmente, ele envia uma mensagem abortando, senão envia uma mensagem confirmando.

- Segunda fase:

O coordenador obtém as respostas de todos os participantes. Se todas as respostas forem de confirmação, o coordenador envia uma mensagem solicitando o commit a todos os participantes. Se a mensagem de algum participante estiver abortando, o coordenador envia uma mensagem para abortar para todos os participantes que responderam confirmando.

Cada participante espera pela decisão do coordenador esperando a mensagem de resposta para finalizar a transação.

b) Problemas nos bancos de dados distribuídos

O gerenciamento de SGBDs distribuídos é mais complexo que em SGBD centralizado porque precisa resolver diversos problemas técnicos de particionamento. Esta complexidade pode influenciar a estabilidade e velocidade de um SGBDD. Destacam-se alguns problemas técnicos para distribuição de dados: (Ozsu e Valduriez, 2006)

- **Processamento distribuído de consultas**: a distribuição das informações em teoria poderia acelerar operações de consultas, mas recuperar dados fragmentados é uma tarefa custosa, principalmente devido a latência da rede. O custo nas atualizações também é alto no caso de dados fragmentados. O principal desafio é otimizar os fragmentos, paralelizando da melhor forma possível com o objetivo de melhorar o desempenho nas consultas e nas atualizações.

- **Controle distribuído da concorrência:** necessidade da sincronização de acessos entre os bancos de dados locais distribuídos, de forma que a seja mantida a integridade dos dados. O problema do controle da concorrência em um ambiente distribuído é mais complexo do que ocorre com a estrutura centralizada, principalmente porque há preocupações como manter a integridade das informações de um único banco de dados enquanto é necessário manter a consistência de várias cópias do banco de dados.

- **Gerenciamento de diretórios distribuídos**: O diretório do banco de dados contém informações como estrutura e localização sobre os itens das informações no banco de dados. Um destes diretórios pode permanecer centralizado em um único local ou distribuído por vários servidores, onde pode haver uma ou várias cópias do banco de dados, sendo mandatório que todos os servidores estejam com o diretório atualizado.

- **Gerenciamento distribuído de impasses**: caso o tradicional mecanismo de bloqueios seja utilizado para sincronização, a concorrência entre usuários pelo acesso a um conjunto de informações pode resultar em impasse. Como alternativa é necessário trabalhar com as técnicas de prevenção, anulação e detecção, que também se aplicam aos bancos de dados distribuídos.

- **Confiabilidade de gerenciadores de bancos de dados distribuídos**: uma das vantagens dos bancos de dados distribuídos é sua maior confiabilidade e disponibilidade. Caso ocorra uma falha que resulta em vários sites inoperantes, os bancos de dados existentes nos sites que permanecem operacionais devem continuar estáveis e atualizados. Quando o sistema de rede se recuperar da falha, o SGBDD deve ser capaz de se recuperar e manter atualizados os bancos de dados nos servidores em que ocorreram a falha, ou seja, é necessário sincronizar os dados com os servidores que ficaram ativos.

- **Suporte do sistema operacional**: os recursos oferecidos pelos sistemas operacionais para operações de bancos de dados não atendem aos requisitos do software de gerenciamento dos bancos de dados distribuídos. Os principais problemas são os sistemas de arquivos, o gerenciamento de memória, os métodos de acesso, a recuperação de falhas e o gerenciamento de processos.

2.1.5 Algumas desvantagens do modelo relacional

Apesar da grande revolução que o modelo relacional trouxe para a área de banco de dados, ainda é possível citar algumas limitações presentes nos dias de hoje.

- **Impedância objeto relacional**

  A incompatibilidade objeto / relacional (impedância) é o conjunto de problemas encontrados no mapeamento de tabelas a objetos e vice-versa. A incompatibilidade dos diferentes tipos de dados no banco de dados relacional e nas linguagens orientadas a objetos é um desses problemas. Além disso, os bancos de dados relacionais não suportam conceitos orientado a objetos como encapsulamento, acessibilidade (modificadores de acesso), composição, polimorfismo e herança. Na década de 1990 uma grande parte dos cientistas de computação acreditavam que os gerenciadores de bancos de dados relacionais fossem ser substituídos por gerenciadores orientados ao objetos, ou por algum gerenciador que suportasse estruturas de dados. Como já descrito anteriormente, o modelo relacional não suporta numa tupla qualquer tipo de estrutura de dados, como um registro aninhado ou uma lista, mas ele tem como vantagem a

simplicidade, velocidade e padronização da linguagem SQL, que é extremamente eficiente ao recuperar dados.

Os *frameworks* de mapeamento objeto-relacional (ORM) tentam resolver estes problemas de incompatibilidade ao funcionar como uma camada de tradução entre objetos e o modelo relacional, servindo como uma camada de que traduz e também persiste os objetos em tabelas. Para cada classe são geradas tabelas, e para cada atributo são geradas colunas, traduzindo os tipos de dados da linguagem orientada a objetos que mais se assemelham aos tipos de dados do modelo relacional. Quando há composição ou herança de objetos, os *frameworks* fazem mapeamentos cuidando dos relacionamentos entre as tabelas e suas chaves para simular a composição ou herança dos objetos. Para cada alteração na estrutura das classes, são disparados comandos DDL no banco de dados para suportar um novo esquema que seja compatível com a nova estrutura de objetos. Para cada nova instância de objeto, ou alteração de objetos existentes, são disparados comandos DML para armazenar e recuperar objetos em tabelas.

- **Representação e consulta de dados complexos com vários níveis de composição.**

  A representação de estruturas com múltiplos níveis de composição no modelo relacional exige grande quantidade de relacionamento entre tuplas através de chaves estrangeiras. Um caso típico deste tipo de estrutura é o caso dos grafos. A forma de representação mais comum de um grafo utiliza duas tabelas. A primeira tabela armazena em cada tupla um vértice junto com seus atributos. A segunda tabela armazena em cada tupla uma aresta do grafo contendo as chaves estrangeiras dos dois vértices que essa aresta relaciona. No caso de processamento de grafos, a representação no modelo relacional se mostra ineficiente devido ao alto custo de processamento das consultas. Para percorrer uma aresta de um vértice a outro do grafo, seria necessário executar uma junção entre as tabelas de vértices e arestas para encontrar seus relacionamentos. Quando realizamos o percurso de um grafo, por exemplo uma busca em profundidade, é necessário executar uma junção para cada nível de profundidade, isto é, é necessário executar uma sequência de operações de junção com os níveis de profundidade que é necessário percorrer. O alto

custo computacional de cada operação de junção torna o modelo relacional muito custoso para representação e processamento para grafos. Este mesmo problema aparece em estruturas hierárquicas como árvores. (Carneiro, 2013).

2.2 Aplicações Big Data

O processamento de grande volume de dados vem se tornando um desafio há pelo menos 50 anos, sendo que na última década se tornou vital para o progresso da humanidade e a concorrência nas organizações. Como exemplo de resultados do processamento de grandes volumes de dados, podemos citar os notáveis avanços na medicina, principalmente no reconhecimento de padrões de doenças através do cruzamento de bancos de dados com informações genéticas de pessoas. Nas empresas podemos citar a identificação de padrões em defeitos, tendências de vendas de produtos e hábitos de consumo. Na área da segurança podemos citar a análise das comunicações para prevenir o terrorismo. Em catástrofes com a detecção da quantidade e posicionamento de celulares para socorro, no caso de terremotos por exemplo. (Hekima, 2016)

Nos anos 1960, o filósofo canadense Herbert Marshall (Kaminski, 2016) cunhou o termo Aldeia Global, que afirmava que as novas tecnologias eletrônicas tinham a tendência de encurtar distâncias, fazendo com que o progresso da tecnologia reduziria o planeta a uma aldeia, ou seja, a um mundo onde todos estariam conectados. Hoje estamos vivendo exatamente a era que Marshall descreveu há 50 anos, principalmente com a popularização do acesso à internet e das redes sociais.

O uso do processamento do grande volume de dados foi ganhando importância conforme o poder de processamento aumentava e com ele surgiam ferramentas para usá-lo. O termo mais utilizado para descrever essas ferramentas ou técnicas é o de Inteligência de Negócios (BI - *business intelligence*), que podemos definir como sistemas que trabalham com um alto volume de dados e identificam padrões e tendências nos dados. Alguns autores estudaram o processo de adoção dessas tecnologias e o classificaram em fases e termos bem antes da popularização do termo Big Data. De acordo com (Tanaka, Silva e Paixão, 2015)

cada uma dessas fases correspondeu a um salto na tecnologia de processamento de dados de grande volume. Segundo (Chen, Chiang e Storey, 2012) é possível estabelecer uma divisão de três fases no uso de sistemas para suporte de decisões de negócios.

A primeira fase no desenvolvimento de BI se iniciou nos anos 1990 quando foram adotadas aplicações completas nas organizações para o gerenciamento de negócios. Estas aplicações, denominadas ERP (*Enterprise Resource Planning*), mapeavam os processos de gerência que antes eram manuais em processos automatizados e parametrizados. Como esses sistemas automatizavam praticamente todos os processos das empresas, a qualidade de informações armazenadas se mostrou muito importante nas decisões estratégicas. (Agrawal, 2014). A partir da adoção de sistemas multifuncionais (como os sistemas ERP), novas ferramentas de consultas aos bancos de dados surgiram, como o processamento analítico online (OLAP) e o Business Intelligence (BI) (Tanaka, Silva e Paixão, 2015). Descrevendo melhor, uma aplicação de BI é uma ferramenta de tomada de decisão estratégica, que basicamente extrai, processa, classifica e analisa os dados coletados, com o objetivo de detectar padrões fornecendo indicadores para tomada de decisão. Esta primeira fase do processamento de alto volume de dados é denominada “Business Intelligence and Analitics” pelos autores (Chen, Chiang e Storey, 2012) e se estende até meados dos anos 2000. Após essa fase inicial de consolidação de dados em sistemas ERP, uma segunda fase importante surgiu, no início dos anos 2000. O crescimento de aplicações de comércio eletrônico e o início das primeiras redes sociais, promoveu um novo tipo de necessidade de processamento de dados. A imensa quantidade de dados que passou a ser gerada era muito difícil de processar com as ferramentas existentes na época. A quantidade de usuários conectados, os padrões de navegação, compras efetuadas, mensagens trocadas, além de formatos não estruturados como imagens e vídeos, são alguns dos exemplos de informações que necessitavam de novas ferramentas para processamento. Essas novas necessidades de processamento de grandes volumes de informações geradas pela internet foram identificadas por (Chen, Chiang e Storey, 2012) e denominadas como a segunda fase do uso do Business Intelligence and Analitics. Como terceira fase, autores como (Berinato, 2014) citam o alto crescimento de novos dispositivos móveis conectados, além da migração de muitas das aplicações locais para a nuvem, conhecidas também como oferta de software por serviço (SaaS) ou

aplicações Cloud. Essa nova geração de dispositivos denominada de IoT “Internet of Things” (Dave, 2016) com sensores que geram uma imensa quantidade dados, exigiu um novo termo para demonstrar a necessidade de se processar um volume nunca antes visto de dados. Um novo termo foi cunhado para expressar essa quantidade de informação a ser processada com o nome de Big Data, sendo caracterizado por esse crescimento na geração de dados em múltiplos formatos e da necessidade de se processar e extrair valor de uma quantidade imensa de dados coletados. (Chede, 2013)

2.2.1 Definições e características do Big Data

O termo Big Data surgiu no ano de 2010, considerando a imensa taxa de crescimento de dados a ser processada como seu principal fator. Apenas para ter uma referência, de acordo com (Davenport, 2014), no ano de 2012 foram gerados no mundo cerca de 2,8 trilhões de gigabytes de dados, sendo possível constatar que essa taxa de crescimento é um bilhão de vezes maior que a quantidade de referência na década passada. A empresa IBM já demonstrou também que atualmente, todos os dias criamos cerca de 2.5 quintilhões de bytes de dados, e que 90% dos dados que existem hoje foram criados nos dois últimos anos. (Mcafee e Brynjolfsson, 2012). Conforme o volume de dados cresce, o formato dos dados também cresce, sendo na maioria das vezes sem estrutura fixa e coletados de múltiplas fontes. Podemos citar exemplos como: fotografias, vídeos, mensagens de e-mail, postagens, comentários em redes sociais e blogs e etc.

O termo Big Data também tem sido vinculado geração de dados contínuos (Chen, Chiang e Storey, 2012) e ao requisito da velocidade para eles serem processados. De acordo com a natureza de um dado, ele pode ficar obsoleto tão rapidamente, que processar o mesmo com algum atraso já o torna inútil. O processamento deve ser feito praticamente em tempo real para que a tomada de decisões tenha utilidade. Como um exemplo deste fato, podemos citar a análise de comentários de produtos ou empresas em redes sociais, também conhecida como análise de sentimento. Caso o processamento desses comentários não seja em tempo real, uma marca ou produto pode ser denegrida rapidamente, portanto a ação imediata de uma reclamação demanda um processamento em tempo real de dados de grande volume. (Matos, 2016)

Em (Stonebraker, 2010) conclui-se que as diferentes características do Big Data podem ser caracterizadas em três propriedades conhecidas como as três Vs:

- **Volume:** a larga escala de crescimento de dados não estruturados superam o armazenamento tradicional e o das soluções analíticas.
- **Variedade:** Big data é coletado a partir de novas fontes que não foram consideradas ou não existiam no passado. Processos tradicionais de gestão de dados precisam lidar com a variedade das fontes de dados como: redes sociais, imagens e vídeos, blogs, sensores de dispositivos, dentre outros.
- **Velocidade:** Os dados são produzidos em uma alta taxa de velocidade e precisam ser consumidos em tempo real.

Recentemente tem sido caracterizados outros dois fatores importantes de Big Data que são a veracidade e o valor. A veracidade é a comprovação que os dados coletados e processados tenham uma origem comprovada, e o valor é a acurácia dos dados, ou seja, se os dados são confiáveis. (Mcafee e Brynjolfsson, 2012)

2.2.2 Bancos de dados em aplicações Big Data

Em geral, os registros em um Banco de Dados podem ser considerados do tipo estruturado, semi-estruturado e não estruturado. Os dados estruturados possuem um esquema fixo, isto é, tem a mesma estrutura de atributos e seus domínios em cada registro, portanto essa estrutura deve ser mandatória. O tipo de dado semi-estruturado permite a definição de um esquema flexível, permitindo que nem todos os registros sigam o mesmo esquema, enquanto os dados não estruturados não possuem qualquer esquema definido. Como exemplos destes tipos podemos citar o modelo relacional como um tipo estruturado, os formatos XML, RDF e OWL como tipos semi-estruturados, e imagens, vídeos ou e-mails são dados não estruturados (Tanaka, Silva e Paixão, 2015)

Muitas das aplicações Big Data se relacionam com dados semi-estruturados e não estruturados, por causa da enorme variedade dos dados que foi mencionada anteriormente. A tabela 1 descreve as principais características dos dados estruturados e semi-estruturados de acordo com (Claro, 2012).

Caraterísticas dos dados estruturados e semi-estruturados

Tabela 1. Caraterísticas dos dados estruturados e semi-estruturados

Dados Estruturados	Dados Semi-estruturados
Esquema pré-definido	Nem sempre há um esquema
Estrutura regular	Estrutura irregular
Estrutura independente dos dados	Estrutura embutida nos dados
Estrutura reduzida	Estrutura extensa onde há particularidades de cada dado, visto que cada um pode ter uma organização própria
Fracamente evolutiva	Fortemente evolutiva (estrutura modifica-se com frequência)
Prescritiva (esquemas fechados e restrições de integridade)	Estrutura descritiva
Distinção entre estrutura e dados é clara	Distinção entre estrutura e dados não é clara

Como consequência das propriedades Big Data, uma aplicação com essas características exige um aumento crescente e contínuo da capacidade de processamento e armazenamento que acompanhe o crescimento do volume e velocidade destes dados. As opções para aumentar o desempenho e capacidade de um sistema gerenciador de banco de dados são duas:

- **Escalabilidade vertical:** Esta escalabilidade consiste em realizar de maneira sistemática a atualização do *hardware* do servidor de banco de dados de acordo com o aumento das exigências de velocidade de processamento e armazenamento da aplicação.
- **Escalabilidade horizontal***:* Esta escalabilidade consiste em distribuir ou replicar os dados em vários servidores e ir incrementando sua quantidade de acordo com a exigência da aplicação.

A solução mais simples é a escalabilidade vertical do sistema de banco de dados, mas essa solução fica inviável quando as exigências das aplicações atingem o limite de recursos em um único servidor, imposto pelo estado atual da tecnologia, o que é muito comum no caso do Big Data. A única solução viável nesse caso é distribuir a base de dados em vários servidores o que exige criar um sistema distribuído, escalável e com alta disponibilidade.

De modo geral para um sistema com essas características são importantes três requisitos:

- **Consistência**

  Num sistema distribuído de dados, este requisito significa que uma vez escrito um registro, este fica imediatamente disponível e pronto para ser utilizado. Podemos afirmar que para ter consistência, todos os servidores da rede necessitam ter a mesma versão dos registros. Os sistemas com uma forte consistência implementam as propriedades ACID definidas nos bancos de dados relacionais. (Brewer, 2000)

- **Disponibilidade**

  Este requisito se refere à concepção e implementação de um sistema que garanta que este esteja ativo de forma permanente durante um determinado período, isto é, o sistema é tolerante a falhas de software ou de hardware e se encontra igualmente disponível durante a atualização destes. Para ter disponibilidade, todos os clientes que necessitam recuperar ou gravar dados precisam encontrar pelo menos uma cópia dos dados solicitados. A disponibilidade oferece também o recurso de balanceamento de carga oferecendo alto desempenho para operações de leitura.

  Uma das formas mais comuns de implementação deste requisito num banco de dados relacional é fazer a replicação entre todos os servidores. Desta forma, a falha ou atualização de um servidor não gera a indisponibilidade do sistema. (Brewer, 2000)

- **Tolerância a particionamento de rede**

  Este requisito se refere à capacidade do sistema de continuar funcionando na presença de uma falha que provoca um particionamento de rede, ou seja, parte da rede fica dividida sem comunicação entre os servidores (Brewer, 2000). Um sistema com tolerância a partições, tem que continuar com seus dados, propriedades e características, ainda quando ocorre uma falha de comunicação na rede que gere um particionamento da mesma.

Em (Brewer, 2000) é relatado o teorema CAP que determina que não é possível para um sistema garantir simultaneamente as propriedades de Consistência (Consistency), Disponibilidade (Availability) e Tolerância a Particionamento (Partition Tolerance).

- **Teorema CAP**

O teorema de CAP afirma que apenas dois desses três requisitos podem ser mantidos ao mesmo tempo. Ele demonstra que para garantir sempre dois requisitos precisamos renunciar ao terceiro. Desta forma, é importante reconhecer, para cada aplicação e cada sistema, quais dos requisitos são importantes.

De modo geral, a maioria das aplicações Big Data precisam priorizar a garantia dos requisitos de Disponibilidade e Tolerância a Particionamento permitindo certa renuncia à Consistência.

2.2.3 Gerenciadores de bancos de dados relacionais em aplicações de Big Data

Considerando as características dos sistemas Big Data assim como os requisitos descritos anteriormente, vários problemas aparecem nos sistemas de bancos de dados relacionais distribuídos que limitam seu uso quando em aplicações Big Data

- **Prioridade na garantia da consistência através das propriedades ACID**

  Sistemas de Bancos Relacionais Distribuídos garantem sempre as propriedades ACID no gerenciamento de transações utilizando

algoritmos como o Protocolo *Two-Phase Commit,* apresentado no capítulo anterior. As propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade) garantem que transações sejam feitas com segurança, garantindo atomicidade de cada uma delas, consistência dos dados, isolamento e integridade durante todas as transações.

Como consequência da manutenção das propriedades ACID, nestes sistemas a prioridade é a garantia do requisito de Consistência. De acordo com o teorema CAP, estes sistemas não permitem garantir totalmente os requisitos de Disponibilidade e Tolerância a Particionamento prioritários em aplicações Big Data. Quando um banco de dados distribui seus dados, as garantias que o modelo relacional oferece através das propriedades ACID são sacrificadas, a fim de garantir a eficiência desta escalabilidade.

- **Escalabilidade limitada nas implementações atuais.**

  Um dos grandes problemas das implementações atuais de sistemas de gerenciamento baseados no modelo relacional é a capacidade de lidar com o alto crescimento nos dados, De modo geral estes sistemas começam apresentar degradação de desempenho quando o volume de dados começa a aumentar. Segundo (Jacobs, 2009), à medida que o número de linhas numa base de dados relacional distribuída aumenta rapidamente, o desempenho do sistema piora, porque a complexidade das operações de junções com tabelas que não estejam no mesmo servidor aumenta, e a velocidade acaba sendo afetada pelo desempenho da rede. Também ao aumentar o número de servidores, a garantia das propriedades ACID nas transações distribuídas reduz o desempenho, porque sincronizar todos os servidores é custoso.

- **Falta de flexibilidade no esquema de dados**

  O modelo relacional possui limitações quando é necessário trabalhar com dados que não possuem um esquema fixo. Em um banco de dados relacional as tabelas tem um esquema rígido com atributos fixos, portanto todas as tuplas de uma mesma tabela seguem o mesmo esquema da definição da tabela. Carregar dados não estruturados no modelo relacional é muito complicado, resultando numa fragmentação

de tabelas com múltiplos esquemas e correlações entre os dados carregados nessas tabelas. (Carneiro, 2013).

2 .2.4 Necessidade de novos gerenciadores de bancos de dados

As necessidades provenientes do Big Data e as limitações dos gerenciadores de bancos de dados relacionais provocaram o surgimento de alternativas para solucionar os problemas de desempenho de processamento e armazenamento. Grandes empresas como o Google e o Facebook, iniciaram uma extensa pesquisa e novos gerenciadores de bancos de dados surgiram. Estes gerenciadores surgiram em 2004 quando o Google lançou o BigTable, que é um modelo de dados alternativo ao modelo relacional, apresentando excelente escalabilidade e disponibilidade (Chang et al., 2008). Diferentemente dos gerenciadores de bancos de dados relacionais, essa nova geração escala horizontalmente mais facilmente. Adicionalmente muitos deles renunciam à manutenção das propriedades ACID, justamente para garantir o maior desempenho possível, substituindo as propriedades ACID por outro critério de consistência, que será discutida na próxima seção.

Estes sistemas de gerenciamento de dados junto com seus modelos de dados surgiram com o nome de Bancos de Dados NoSQL como uma forma de se distanciar do modelo relacional convencional e sua linguagem de consulta padrão SQL.

- **Propriedades ACID x BASE**

Os gerenciadores de bancos de dados relacionais garantem a integridade dos dados mantendo as propriedades ACID nas estruturas de armazenamento e gerenciamento de transações. Os novos modelos e sistemas de bancos de dados NoSQL trocam essas propriedades em favor do desempenho e da garantia dos requisitos de disponibilidade e tolerância a particionamento.(Pritchett, 2008)

No entanto, esses sistemas precisam manter integridade nos seus dados. Através dessa necessidade foram estabelecidos novas propriedades de integridade conhecidas como BaSE (*Basically Available, Soft state, Eventual consistency*) como contraposição às propriedades ACID. Estas propriedades focam na alta disponibilidade no sistema mas com uma definição de consistência

flexível chamada de consistência eventual. (Pritchett, 2008). As propriedades BaSE são as seguintes:

- BA (*Basically Available*): Toda requisição ao banco de dados deve ser atendida, portanto a disponibilidade é a prioridade.

- S (*Soft State*): O estado do banco de dados não precisa ser consistente o tempo todo, isto é, o estado do sistema pode ser alterado ao longo do tempo, mesmo sem novas operações, onde são sendo permitidas alterações propagadas com atraso.

- E (*Eventually Consistent*): O banco de dados irá se tornar consistente em momento futuro, mesmo que seja indeterminado.

A partir do conceito de BASE foram criados diferentes tipos de bancos de dados NoSQL, que não garantem consistência dos dados de uma aplicação do modo definido pelas propriedades ACID, mas trabalhando com o modo alternativo denominado consistência eventual. Empresas pioneiras nessa técnica como Google e a Amazon atendem as propriedades BASE, utilizando um modelo NoSQL com distribuição e escalabilidade horizontal. (Pritchett, 2008)

Como não existe garantia de consistência, torna-se mais fácil efetuar configurações de particionamento no modelo, resultando em mais escalabilidade, eficiência e desempenho, minimizando os custos de processamento e rede.

É importante ressaltar que a garantia das propriedades BASE não é padrão em todos os SGBDs NoSQL, portanto alguns SGBDs deles ainda implementam as propriedades ACID. Existem também SGBDs que permitem configurações optando entre um ou outro modelo de consistência no momento de programar uma transação. Os gerenciadores que permitem o tipo de seleção de consistência numa transação são mais flexíveis, gerando possibilidades como garantir a consistência em operações financeiras por exemplo.

2.3 Conclusão do capítulo

Neste capítulo foi apresentado o modelo relacional, a definição e as características de aplicações Big Data, bem como as desvantagens do modelo relacional para trabalhar com Big Data. Limitações do modelos relacional e suas

implementações acabaram gerando a necessidade de novos gerenciadores que foram brevemente mencionados neste capítulo.

No próximo capítulo serão apresentados com mais detalhes os motivos do surgimento dos gerenciadores NoSQL, os modelos e implementações destes gerenciadores suas vantagens para trabalhar com aplicações Big Data.

Capítulo 3. Modelos e implementações de SGBDs NoSQL

Este capítulo tem como objetivo descrever as características mais gerais dos modelos de dados denominados NoSQL. Serão apresentados cada um dos quatro tipos de modelos de dados considerados NoSQL, modelo de dados chave -valor, modelo de dados orientado a colunas, modelo de dados orientado a documentos e modelos de dados de grafos. Dada a ausência de padrões para estes modelos, serão apresentados exemplos utilizando as linguagens de consulta de alguns dos gerenciadores de bancos de dados mais relevantes baseados nestes modelos de dados.

3.1 Introdução

O termo NoSQL (*not only SQL*) tem sido utilizado como um termo guarda-chuva para agrupar uma série de projetos de implementação de modelos de banco de dados, com diferenças significativas quando comparados ao modelo relacional tradicional.

O termo foi utilizado pela primeira vez em 2009 por Carlo Strozzi, descrevendo um banco de dados criado por ele que não utilizava a linguagem SQL para consulta, nem implementava o modelo relacional. (Fowler, 2015)

Nesse mesmo ano, Johan Oskarsson organizou em São Francisco um evento para discutir novas tecnologias para armazenamento e processamento de dados. Os principais destaques do evento foram os novos produtos BigTable e Dynamo (Evans, 2009). Questionado sobre qual termo ia ser divulgado no Twitter nesse evento, Eric Evans da empresa RackSpace sugeriu o termo NoSQL. A partir desse evento, bancos de dados que utilizam modelos de dados alternativos ao modelo relacional e que tem como foco a escalabilidade, começaram a ser categorizados como bancos de dados NoSQL (Fowler, 2015). Como foi discutido no capítulo anterior, os fatores fundamentais para o desenvolvimento dos sistemas de bancos de dados NoSQL foram a necessidade de alta escalabilidade, disponibilidade e flexibilidade no esquema dos dados, permitindo estruturas de dados dinâmicas e adaptáveis. (Shashank, 2014)

Os problemas que os novos gerenciadores de bancos de dados NoSQL visam resolver são antigos. As primeiras empresas de ferramentas de busca como Excite, Yahoo, Lycos, Inktomi e Altavista, apresentaram em meados dos anos 1990

muitas demandas e reclamações sobre gerenciadores de bancos de dados (Nicholson, 1996). Essas empresas constataram que os sistemas de gerenciamento tradicionais baseados no modelo relacional apresentavam limitações para manipular quantidades massivas de dados, sendo obrigadas cada uma delas a implementar soluções próprias para armazenar e recuperar dados para suas aplicações.

Devido a essa necessidade de escalar e processar grande quantidade de dados, a empresa Google criou posteriormente um sistema de arquivos distribuído, um banco de dados orientado a colunas e um sistema distribuído de processamento denominado MapReduce (Dean & Ghemawat, 2004). Basicamente, Google criou um conjunto de tecnologias para resolver os problemas que as empresas pioneiras de busca enfrentavam. A partir dessas ferramentas internas, o Google iniciou a publicação de artigos descrevendo sua arquitetura e permitindo que desenvolvedores aprendessem seus produtos. Com base nessas publicações, funcionários do Yahoo criaram o Hadoop, que implementa toda essa arquitetura de forma open source. (Hadoop, 2007)

Após a criação do Hadoop, a Amazon também publicou artigos apresentando sua implementação de banco de dados NoSQL Dynamo (DeCandia, et al., 2007). A partir dessas iniciativas, empresas como Facebook, Netflix, eBay, dentre outras, resolveram também abrir suas soluções NoSQL internas para bancos de dados, resultando numa ampla gama de propostas e soluções para os novos problemas enfrentados por estas empresas.

O termo "NoSQL" tem origem absolutamente espontânea e não tem ainda uma definição universalmente aceita na comunidade científica. Em (Fowler, 2013) o autor tenta agrupar e organizar esses novos gerenciadores de bancos de dados.

Atualmente existem no mercado, cerca de duas centenas de bancos de dados NoSQL, dando suporte e utilizados em grandes portais na internet. (Edlich, 2016)

3.2 Características dos bancos de dados NoSQL

Como vimos no capítulo 2, os sistemas de gerenciamento NoSQL surgiram como alternativa aos sistemas de gerenciamento relacionais para as aplicações BigData. De modo geral existem quatro modelos de dados NoSQL que serão

apresentados na sequência. Estes modelos são: chave-valor, orientado a colunas, orientado a documentos e de grafos. Entre estes modelos existem diferenças mas todos eles compartilham características que serão descritas a seguir (Fowler, 2015):

• **Escalabilidade Horizontal:** Como vimos no capítulo 2, a escalabilidade horizontal é a solução que permite o processamento do grande volume de informação associado a aplicações Big Data adicionando novos servidores, fazendo com que a carga de requisições seja distribuída entre esses servidores. Uma das grandes vantagens dos bancos NoSQL é a ausência de bloqueios associada à manutenção das propriedades ACID, o que permite a escalabilidade horizontal com uma maior facilidade e eficiência.

• **Consistência eventual:** provavelmente o mais importante aspecto dos gerenciadores do tipo NoSQL é a consistência eventual que foi mencionada anteriormente. Esta característica pode ser explicada pelo teorema CAP assim como pelas propriedades BASE permitindo uma tolerância nas inconsistências temporárias com o objetivo de priorizar a disponibilidade.

• **Esquema flexível:** Outra característica comum em gerenciadores de bancos de dados NoSQL é a ausência parcial ou completa de esquema que define a estrutura do banco de dados, permitindo maior flexibilidade. Um aspecto de desvantagem é a falta de garantia de integridade dos dados, fato este que não ocorre no modelo relacional.

• **Desempenho na replicação:** Os gerenciadores de bancos de dados NoSQL geralmente possuem recursos de replicação de forma nativa, justamente para prover a melhor escalabilidade possível. Com esses recursos e algoritmos de balanceamento de carga, o tempo gasto para recuperar informações é reduzido.

• **Variedade de APIs para acesso ao banco de dados:** O acesso por parte das aplicações aos recursos dos gerenciadores do tipo NoSQL são variados cobrindo a maioria das plataformas e linguagens. Apesar de não ter uma padronização na linguagem de consulta como o SQL no modelo relacional, o padrão REST está sendo dominante no acesso via webservice.

Tabela 2. Caraterísticas do modelo relacional e do modelo NoSQL

	Banco de Dados Relacional	**Banco de Dados NoSQL**
Escalonamento	É possível adicionar novos servidores no modelo relacional, mas a adição de novos servidores impacta fortemente em operações de escrita, portanto o limite da escalabilidade é pequeno.	É possível adicionar muitos servidores dependendo do gerenciador utilizado, inclusive adição em tempo de execução.
Consistência	É uma das vantagens do modelo relacional porque as suas regras de consistência são bastante rigorosas.	É realizada eventualmente, apesar que alguns gerenciadores permitem especificar qual transação ou não deve ter consistência garantida em tempo de execução.
Disponibilidade	Tem um aspecto limitado porque não permite a adição de um número grande de servidores.	Permite a adição de muitos servidores oferecendo excelente disponibilidade e também particionamento automático.

Na tabela 2 é apresentado um resumo das características gerais dos modelos de dados NoSQL comparadas com o modelo relacional.

A seguir serão apresentados os principais modelos lógicos de dados NoSQL e os sistemas de gerenciamento mais destacados para cada um deles. Uma dificuldade para a apresentação dos modelos de dados NoSQL é a ausência de uma padronização como existe no modelo relacional. Por esta razão, cada modelo será apresentado informalmente e será ilustrado com algum dos sistemas que o implementam.

3.3 Modelo orientado a Chave-Valor (Key-Value)

- **Descrição do modelo:**

Um banco de dados no modelo chave-valor é composto por uma coleção de pares de elementos, sendo que cada par consiste de uma chave, que é única e

identifica o par, e um valor associado a essa chave. Segundo (Shashank, 2014) o modelo chave-valor armazena os dados como uma grande tabela *hash*.

O valor associado a uma chave pode ser de tipos diferentes como *String*, um dado semiestruturado baseado em documentos, uma lista de valores ou uma tabela *hash* contendo uma outra coleção de pares chave e valor. Os valores armazenados em um mesmo banco de dados podem ser de tipos diferentes. A figura 3 exibe as diferentes possibilidades de valores associados a uma chave. Note-se que os tipos e estruturas podem ser diferentes.

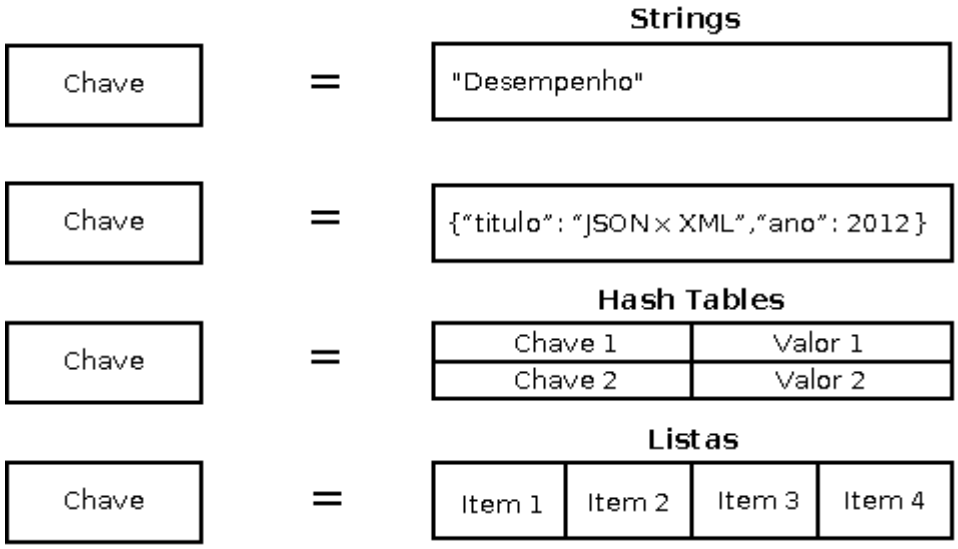

Figura 3. Possíveis valores armazenados por uma chave

Este modelo é fácil de implementar, bem escalável, e permite que os dados sejam rapidamente acessados pela chave. Isso contribui para aumentar a disponibilidade de acesso aos dados.

Uma característica deste modelo é que grande parte da lógica e o controle de integridade, é transferida diretamente para a aplicação, que é a responsável por manipular o valor associado a cada chave. Por exemplo, o valor associado a uma chave pode ser um documento, geralmente representados em formatos XML ou JSON, mas este documento será manipulado como um único valor, sem que o sistema de gerenciamento permita efetuar operações com os componentes do documento.

Neste modelo, o banco de dados geralmente não é normalizado. Isto implica que as informações podem ser armazenadas de forma redundante com

perigo de inconsistência e provocando a necessidade de maior espaço de armazenamento para os dados. No entanto, pelas características das aplicações prefere-se sacrificar estas propriedades para garantir maior desempenho na recuperação dos dados.

Como exemplo, considere-se uma aplicação em que são armazenados pedidos de compra. Cada pedido é identificado com uma chave junto com informações adicionais desse pedido, representadas em um documento JSON que inclui o nome do cliente e demais dados. Para utilizar a informação de um pedido, a aplicação é responsável por recuperar o documento JSON integralmente através da sua chave e decompô-lo. Caso um mesmo cliente esteja associado a vários pedidos, seu nome será replicado em cada um desses pedidos de forma redundante, mas facilitando as operações de armazenamento e recuperação dos pedidos. No entanto, caso o nome do cliente seja modificado, será necessário percorrer todos os pedidos para realizar a atualização do nome, isto é, a aplicação é responsável por garantir a consistência do nome do cliente no banco de dados.

- **Casos apropriados para uso**

  O modelo de chave-valor pode ser utilizado para armazenar informações que necessitem de rápido acesso, com uma pequena variedade de consultas, além de dados que serão descartados em breve por possuir um tempo de vida curto (Sadalage e Fowler, 2013). Por exemplo, o site de compras da Amazon utiliza o sistema de gerenciamento chave-valor Dynamo para armazenar as compras dos usuários. (Vogels, 2008)

- **Restrições**

  Dada as limitações deste modelo para realizar junção entre os dados, ele não é recomendável quando é necessário representar relacionamentos entre diferentes conjuntos de dados. Adicionalmente, este modelo apresenta limitações quando desejamos processar transações com múltiplas operações ou quando desejamos consultas complexas com múltiplas condições. (Sadalage, 2013)

- **Operações no banco de dados**

Em seguida são exibidas as operações de criação de banco de dados, inserção, recuperação, alteração e exclusão em um banco de dados chave-valor. Dada a ausência de um padrão, para ilustrar as operações, utilizaremos a linguagem do sistema de gerenciamento Redis.

Criação e uso de um banco de dados:

Para criar um banco de dados em geral basta se conetar no gerenciador atribuindo um identificador ao novo banco de dados. O banco de dados *default* com identificador 0 já está disponível para operações com dados. Não precisa ser fornecida nenhuma informação sobre a estrutura do banco de dados

No caso de Redis, o identificador de um banco de dados é numérico e o comando SELECT permite selecionar qual o banco que será utilizado.

Por exemplo, o comando:

```
SELECT 1
```

indica ao sistema que o banco de dados que será utilizando é aquele com identificador 1:

Inserção de dados:

Para adicionar um par chave-valor em Redis utilizamos o comando SET:

SET chave valor

Exemplo:

SET ano 2017

armazena no banco de dados o par com chave `ano` e valor `2017.`

Para inserir uma coleção de dados relacionados, de forma análoga a uma tabela no modelo relacional, associamos a uma chave uma tabela *hash* que contém um mapeamento de campos da tabela com valores. Para esta operação utilizamos o comando HSET:

```
HSET chave campo1 valor1
HSET chave campo2 valor2
```

Por exemplo, os comandos:

```
HSET 316412 nome João da Silva
HSET 316412 email joao@mail.com
HSET 316412 sexo masculino
```

permitem armazenar um usuário de RG `316412` com seu nome, e-mail e sexo.

Atualização de dados:

No caso de Redis, para atualizar dados podem ser utilizados os mesmos comandos para a inserção de dados, especificando uma chave já existente. Se a chave já existe, o valor associado a essa chave será automaticamente atualizado.

Por exemplo:

o comando

```
HSET 316412 email JoaoSilva@hotmail.com
```

atualiza o e-mail do usuário de RG 316412

Exclusão de dados:

Para excluir uma chave-valor utilizamos o comando DEL:

```
DEL chave
```

Para excluir um campo de uma tabela *hash* é necessário utilizar o comando HDEL especificando a chave:

```
HDEL chave campo
```

Por exemplo, o comando:

```
HDEL 316412 e-mail
```

elimina o campo `e-mail` da tabela associada ao usuário `316412`:

Recuperação de dados:

Para recuperar dados é necessário utilizar o comando apropriado selecionado chaves simples ou chaves com múltiplos campos e valores, que no caso é uma tabela *hash*.

Para recuperar uma chave-valor é necessário utilizar o comando GET:

```
GET chave
```

Exemplo recuperando a chave ano definida acima:

```
GET ano
```

Para recuperar uma chave que contém uma tabela *hash* utilizamos o comando HGET para um campo ou o comando HGETALL para todos os campos.

```
HGET chave campo
HGETALL chave
```

Por exemplo: o comando

```
HGET 316412 e-mail
```

recupera o valor do campo `e-mail` associado à tabela associada à chave `316412` enquanto o comando:

```
HGETALL 316412
```

recupera todos os campos da tabela associada à chave `316412`.

- **Exemplo de uso**

O modelo de chave-valor pode ser utilizado para armazenar os itens que estão sendo selecionados no carrinho de um site de compras.

É possível armazenar os itens escolhidos e suas quantidades para cada usuário utilizando chaves e valores no banco de dados Redis. No exemplo abaixo adicionamos um carrinho de compra com dois itens com suas quantidades. O comando HSET adiciona uma tabela *hash* que possui uma chave seguida de campos chave-valor, e o comando HGETALL recupera todas as chaves e valores de uma tabela *hash* com chave especificada.

O comando HSET tem como seu primeiro parâmetro o nome de uma chave, seguido de um mapeamento chave-valor que no SGBD Redis é denominado como *Fields* e *Values*. A chave neste caso é o nome de uma tabela *hash*.

No exemplo abaixo é criada uma cesta de compras de um usuário denominado “cart_Jose”, onde são adicionados duas compras de livros no formato

chave-valor, utilizando um delimitador “dois pontos” para separar o nome do livro e sua quantidade.

```
HSET cart_Jose compra1 “livro1:1”
HSET cart_Jose compra2 “livro2:5”
```

Para obter todas as chaves e seus valores, basta informar o nome da chave com o comando HGETALL.

```
HGETALL cart_Jose
```

Retorna:

```
“compra1” “livro1:1”
“compra2” “livro2:5”
```

- **Consistência dos dados:**

Os sistemas de bancos de dados baseados no modelo chave-valor geralmente não se preocupam com a consistência imediata dos dados, isto é, são eventualmente consistentes (Shashank, 2014)

Os bancos de dados Voldemort, Riak e Oracle NoSQL utilizam para a leitura de dados um algoritmo denominado *Read Repair* (Vogels, 2008). O algoritmo funciona em dois passos:

- No momento da leitura de um registro ocorre uma tentativa de determinar, através de um *timestamp*, qual dos vários valores disponíveis para a chave é o mais recente.
- Se o valor mais recente não pode ser decidido, então o cliente de banco de dados é apresentado com todas as opções de valores, deixando para decidir por si mesmo.

O banco de dados Redis trabalha com replicação assíncrona e o conceito de mestre e escravo. É possível permitir a leitura de servidores escravos, mas a leitura pode ser inconsistente e a aplicação precisa lidar com dados desatualizados. Para minimizar tal problema, existe um recurso que bloqueia os

clientes até que um número especificado de servidores escravos confirmem a atualização da informação, portanto esse recurso pode ser utilizado para informações críticas que não podem ficam inconsistentes.

Outros bancos de dados como o Aerospike trabalham com um conceito chamado *shared-nothing*, que permite uma replicação síncrona dos dados para manter os dados consistentes (Shashank, 2014). Com esse mecanismo, os servidores são dimensionados para formar um *cluster* que particiona os dados e paraleliza o processamento entre os servidores de forma transparente. Nos bancos de dados que trabalham com esse mecanismo uma chave alterada é replicada de forma síncrona em todos os servidores, garantindo consistência imediata dos dados.

3.4 Modelo orientado a Colunas

- **Descrição do modelo:**

No modelo orientado a colunas, um banco de dados é uma coleção de famílias de colunas. Uma família de colunas está composta por linhas que possuem uma chave e múltiplas colunas. Cada uma destas colunas associadas a uma linha consiste de um par chave-valor e um registro temporal (*timestamp)*. Cada valor do par pode ser de tipo atômico (inteiro, real, etc) ou de tipo *Map*. Um valor de tipo *Map* é uma coleção aninhada de pares chave-valor. O *timestamp* serve para expirar os dados e resolver conflitos de gravação.

Diferentes linhas não precisam ter as mesmas colunas e novas colunas podem sempre ser adicionadas a qualquer linha. O conceito de supercoluna, permite agrupar várias colunas de uma linha e associar a elas uma única chave, sendo considerado este agrupamento um contêiner de colunas.

A figura 4 mostra um exemplo de banco da dados baseado em colunas, Note-se que a linha contém seis colunas que são compostas por pares de chave-valor, e essas colunas foram agrupadas em duas super colunas.

Este modelo de dados surgiu associado o sistema BigTable (Chang, Fay et al. 2006) criado pelo Google, por isso é comum se referir a ele como o modelo de dados BigTable. Este modelo não suporta consultas com junção (*joins*) entre diferentes famílias de colunas. Por esta razão é necessário desnormalizar os

dados, isto é, armazenar os dados de forma redundante para garantir a escalabilidade. Para este modelo é desejável uma boa modelagem dos dados com foco nas consultas, visando a redundância destes dados para rápido acesso as consultas que serão realizadas. Comparado com o modelo de chave-valor, o modelo de colunas possui mais recursos porque permite associar a uma chave, uma família de colunas, que por sua vez tem colunas que contém um *timestamp* e um valor. (Fowler, 2015)

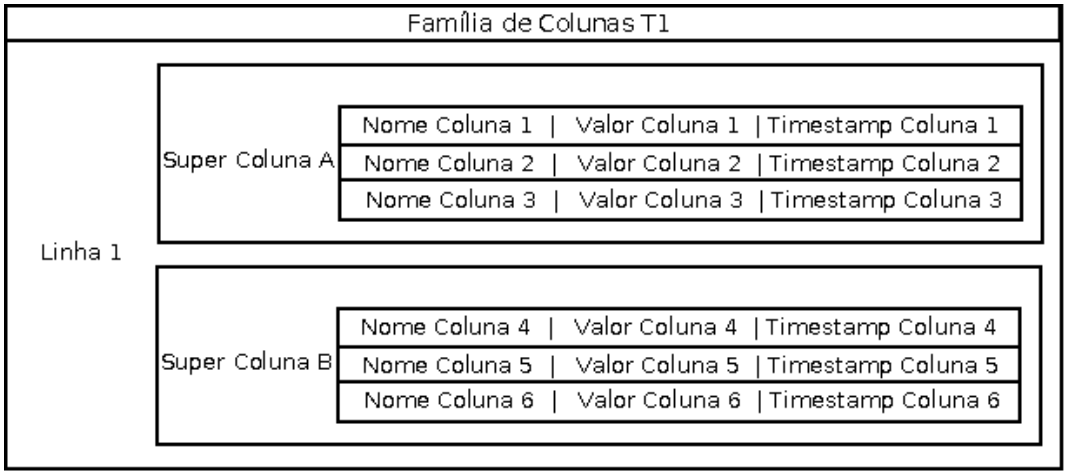

Figura 4. Organização do modelo de colunas

- **Casos apropriados para uso**

O modelo de colunas é ideal para dados que expiram após um período por possuir o recurso de *timestamp* nativo, aplicações com grande volume de dados e também para aplicações que necessitem particionar os dados em grandes *clusters* de computadores. (Sadalage e Fowler, 2013)

- **Restrições**

Não é recomendável utilizar quando for necessário processar operações de agregações ou quando as consultas forem dinâmicas. Quando não há certeza sobre os padrões de consulta, esse modelo não é ideal porque pode ser necessário alterar o formato das famílias de colunas para atender as consultas. (Sadalage e Fowler, 2013)

- **Operações no banco de dados orientado a colunas**

Em seguida são exibidas as operações básicas em um banco de dados baseado em colunas. Para ilustrar as operações utilizaremos a linguagem do sistema de gerenciamento Cassandra CQL.

Criação do banco de dados:

Um banco de dados é criado atribuindo a ele um nome e posteriormente são definidas as famílias de colunas. Em CQL um banco de dados é criado como um *Keyspace* através do comando:

```
CREATE KEYSPACE nome_do_banco_de_dados
```

Em seguida podem ser criadas famílias de colunas informando as colunas e tipos de dados que podem ser definidas para cada linha da família. Em CQL esta operação é realizada com o comando *CREATE TABLE*.

```
CREATE TABLE família (coluna1 tipo_de_dados, coluna2 tipo_de_dados)
```

Os tipos de dados associados a uma coluna podem ser de tipo atômicos (inteiro, real, texto, etc) ou pode ser de tipo *map.* O tipo *map* permite que uma coluna esteja formada por sua vez, de forma aninhada por uma coleção de colunas e tipos.

O exemplo abaixo cria uma família de colunas `usuarios` com colunas usuario_id, idade e status e telefones. Esta última coluna telefones é definida de tipo *map* o que significa que a ela pode ser atribuída uma coleção de colunas.

```
CREATE TABLE usuarios (
   usuario_id text PRIMARY KEY,
   idade int,
   status text,
   telefones  map<text,text>
    );
```

Inserção de dados:

Para inserir dados são fornecidos os valores das colunas em cada linha. Em CQL é utilizado o comando *INSERT INTO* que atribui a cada coluna seu valor.

Em particular nas colunas definidas de tipo *map* podem ser inseridas coleções de múltiplas colunas

```
INSERT INTO família values (campo1, campo2, campoMap
{'chave1':'valor1','chave2':'valor2'})
```

Neste comando a seguir é inserida uma linha na família `usuarios` contendo as colunas `usuario_id, idade, status e telefones` com seus valores. Note-se que no caso da coluna `telefones` é inserida uma coleção de colunas, cada uma para um número de telefone diferente.

```
INSERT INTO usuarios(usuario_id, idade,status)
VALUES("Joao",45, "A" ,
{ 'Comercial':'3973512" , 'Celular':'9951231'})
```

Recuperação de dados:

Para recuperar dados são fornecidas as colunas para as quais deseja-se os resultados. Em Cassandra é utilizado o comando *SELECT* que recupera as colunas dentro da família de coluna informada que satisfazem uma determinada condição. Este comando é similar ao *SELECT* de SQL mas não permite a realização de junções.

```
SELECT colunas from família_de_coluna
```

O exemplo abaixo recupera um usuário com coluna usuario_id1 da família de colunas usuários.

```
SELECT usuario_id, status FROM usuarios
where usuario_id=1 and status="A"
```

Atualização de dados:

Para atualizar dados é fornecida a família de colunas e as colunas que sofrerão a atualização com seus respectivos novos valores. Em Cassandra é utilizado o comando *UPDATE:*

```
UPDATE família_de_colunas
SET coluna = valor
```

O exemplo a seguir atualiza a coluna status das linhas da família de colunas `usuários` que satisfaz a condição de que o valor da coluna `usuario_id` seja 1.

```
UPDATE usuarios
SET status = "C"
where usuario_id=1
```

O comando *SET* permite adicionar dinamicamente novas colunas para uma linha. No exemplo abaixo uma nova coluna `passaporte` é adicionada dinamicamente apenas para a linha que contém a chave primária com coluna `usuario_id` de valor `1`..

```
SET usuarios[1][‘passaporte’] = ‘EXC9999’;
```

Exclusão de dados:

Para excluir dados do Cassandra é utilizado o comando *DELETE* informando a família de colunas selecionando o critério de exclusão na condição.

```
DELETE FROM família_de_coluna WHERE condição.
```

O exemplo abaixo exclui a linha que contém a coluna usuario_id=1

```
DELETE FROM usuarios
where usuario_id=1
```

- **Exemplos de uso**

Um exemplo conhecido de utilização deste modelo é a aplicação de entretenimento Netflix. Esta aplicação utiliza o sistema de gerenciamento de banco de dados Cassandra para armazenar as descrições e comentários dos filmes e shows. (Izrailevsky, 2011)

Consideremos o uso do modelo baseado em colunas para esta aplicação.

É possível criar uma estrutura onde as linhas armazenam os filmes e shows, e para cada uma destas linhas pode ser criada uma coleção de colunas de tipo *map* armazenando as descrições e comentários dos usuários

No comando abaixo é criado uma família de colunas filmes com esquema definido pelas colunas `chave`, `descrição` e `comentário`. As colunas `descrição` e `comentário` são do tipo *map*, e portanto permitem múltiplos valores .

```
CREATE TABLE filmes (
     chave text PRIMARY KEY,
     descricao map<text,text>,
     comentario map<text,text>,
);
```

O comando abaixo insere um novo filme atribuindo duas descrições na coluna descrição.

```
INSERT INTO filmes (chave, descricao) VALUES      (
'Uma Odisseia no Espaco',

{'ano':'1968','produtor':'Stanley Kubrick'},
)
```

No comando a seguir são adicionados comentários aos existentes na coluna *map*. Para isso é utilizado o operador de concatenar “+”.

```
UPDATE filmes SET comentarios = comentarios +
{'Usuario1':'Muito bom o filme','Usuario2':'Trilha
sonora excelente'} WHERE chave='Uma Odisseia no Espaco'
```

Abaixo são listados os filmes com o comando *SELECT*.

```
SELECT chave, descricao, comentario FROM FILMES;

chave                  | descricao          |
comentario
-----------------------+--------------------+-------
Uma Odisseia no Espaco |
{'ano':'1968','produtor':'Stanley Kubrick'} |
{'Usuario1':'Muito bom o filme','Usuario2':'Trilha
sonora excelente'}
```

- **Consistência dos dados:**

Nos bancos de dados orientados a colunas, geralmente são implementados com consistência eventual, portanto num determinado momento todas as réplicas nos diferentes servidores estarão consistentes.

Em algumas implementações como o Cassandra, existe a opção de alterar a consistência em tempo de execução. Com esse tipo de recurso, o desenvolvedor tem a possibilidade de definir para cada operação a qual o tipo de consistência desejada. Esta consistência pode ser definida estrita numa operação, e em outra operação com consistência fraca. (Shashank, 2014)

Algumas opções de consistência que o Cassandra oferece são as seguintes:

- One: Nesta opção, é suficiente que um único servidor responda a requisição, que a resposta já será enviada ao cliente.
- Quorum: Nesta opção a resposta a uma requisição será determinada por um “quórum” definido entre um número de servidores, ou seja, um número de servidores precisam responder para confirmar a resposta da requisição.
- All: Nesta opção a resposta a uma requisição somente é realizada quando todos os servidores são consultados.

O Cassandra também oferece um recurso chamado *lightweight transaction* que permite garantir que uma sequência de operações será executada sem interferência de outras, sendo bem semelhante ao nível de isolamento serializável oferecido pelos bancos de dados relacionais. Esse tipo de recurso deve ser evitado porque é uma operação custosa, principalmente porque obriga que todas as réplicas se atualizem pela rede. (Borsos, 2013)

Esse tipo de modelo também oferece recursos de indexação secundária para colunas.

3.5 Modelo orientado a Documentos

- **Descrição do modelo:**

Neste modelo os bancos de dados são coleções de documentos. Um documento, em geral, é um objeto com um identificador único e um conjunto de campos, que podem ser strings, listas ou documentos aninhados. Esta estrutura se assemelha à estrutura chave-valor apresentada anteriormente, mas permite aninhamento dos dados como é o caso do formato XML.

O formato mais utilizado no modelo de documentos é o JSON (JavaScript Object Notation) que é constituído por duas estruturas. A primeira é uma coleção de pares chave-valor, caracterizado por um objeto ou estrutura. A segunda é uma lista ordenada de valores, caracterizado por uma coleção ou vetor. Nesse formato os objetos são delimitados por chaves, onde os pares chave/valor são separados por vírgula (Fowler, 2015). Na figura 5 é possível verificar um exemplo de

documento JSON. O atributo `_id` é um atributo simples que recebe apenas um valor, que na notação JSON opera com o carácter dois pontos. O atributo `name` é composto pelo carácter chaves que permite uma estrutura aninhada de pares chave-valor, separadas por vírgula. Esta estrutura é denominada objeto. Também temos o atributo `contribs` que é composto pelo carácter colchete que permite uma lista de valores separados por vírgula. Essa estrutura é denominada array.

A mesma estrutura mapeada em JSON da figura 5 é demonstrada no formato XML com os elementos aninhados para base de comparação.

Uma característica importante deste modelo é que ele não depende de um esquema rígido, isto é, não obriga uma estrutura fixa como ocorre nos bancos relacionais. O formato do documento é livre permitindo assim que ocorra uma atualização na estrutura do documento, adicionando novos campos.

Um problema do modelo relacional descrito no capítulo 2 é a impedância objeto-relacional. Os objetos modelados nas aplicações segundo o paradigma de orientação a objetos, mas eles precisam ser convertidos em tabelas para ser armazenados em bancos de dados.

Este mapeamento pode ser complexo para determinado tipo de aplicações. Uma alternativa de armazenamento a projetos orientados a objetos tem sido o modelo de documentos. A estrutura hierárquica própria deste modelo é mais adequada para mapear uma coleção de dados modelados com orientação a objetos.

Nesse modelo também é comum a desnormalização, armazenando dados redundantes para otimizar o armazenamento e consulta. É possível num único documento salvar toda uma hierarquia de objetos e até mesmo criar ligações entre diferentes documentos, de forma semelhante as chaves estrangeiras do modelo relacional, visando facilitar as consultas. Estas ligações não são verificadas pelo sistema de banco de dados, portanto a aplicação deve manter controle sobre a integridade das ligações e dos dados redundantes.

Um gerenciador de banco de dados que adota esse modelo é o MongoDB (Suter, 2012).

Exemplo do formato JSON:

```
{
    '_id' : 1,
    'name' : { 'first' : 'John', 'last' : 'Backus' },
    'contribs' : [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
    'awards' : [
        {
            'award' : 'W.W. McDowell Award', 'year' : 1967, 'by' : 'IEEE
Computer Society'
        }, {
            'award' : 'Draper Prize', 'year' : 1993, 'by' : 'National Academy
of Engineering'
        }
    ]
}
```

Exemplo do formato XML:

```
<?xml version="1.0" encoding="UTF-8" ?>
       <_id>1</_id><name><first>John</first><last>Backus</last></name>
       <contribs>Fortran</contribs><contribs>ALGOL</contribs>
       <contribs>Backus-Naur Form</contribs><contribs>FP</contribs>
       <awards>
              <award>W.W. McDowell Award</award>
              <year>1967</year><by>IEEE Computer Society</by>
       </awards>
       <awards>
              <award>Draper Prize</award>
              <year>1993</year><by>National Academy of Engineering</by>
       </awards>
</xml>
```

Figura 5. Exemplo de mapeamento no modelo de documentos

- **Casos apropriados para uso**

  O modelo de documentos é bem flexível e pode atender aplicações em geral, desde aplicações comerciais até análise de tempo real pelo ótimo desempenho. (Sadalage e Fowler, 2013)

- **Restrições**

  Não é recomendável utilizar quando for necessário ter transações complexas entre múltiplos documentos ou quando é necessário impor um esquema muito rígido de consistência. (Sadalage e Fowler, 2013)

- **Operações no banco de dados MongoDB**

A seguir são apresentadas as operações de criação, inserção, recuperação, alteração e exclusão de dados no banco de dados.

Para ilustrar estas operações utilizaremos a linguagem do sistema de gerenciamento MongoDB o qual utiliza como padrão documentos JSON.

Criação do banco de dados:

Em MongoDB um banco de dados é composto por coleções e não é necessário criá-lo explicitamente.

O comando *USE* seleciona um banco de dados de trabalho que é criado automaticamente caso não exista.

```
use banco_de_dados
```

Inserção de dados:

Para inserir dados é necessário utilizar o comando *INSERT* informando qual coleção será inserida o documento, passando como parâmetro o documento JSON a ser inserido.

```
db.colecao.insert( documento_json )
```

Neste exemplo é inserido um documento na coleção `usuarios` que contém os atributos `usuario_id, idade e status`.

```
db.usuarios.insert(
   { usuario_id: "Joao", idade: 45, status: "A" }
)
```

Recuperação de dados:

Para recuperar dados é necessário utilizar o comando *FIND* na coleção que será procurado o documento, informando também os critérios de seleção.

```
db.colecao.find( critérios )
```

Exemplo recuperando um usuário com atributo usuario_id com valor 1 da coleção usuarios.

```
db.usuarios.find(
    { },
    { usuario_id: 1}
)
```

Atualização de dados:

Para atualizar dados é necessário utilizar o comando *UPDATE* selecionando a coleção que o documento pertence e também o filtro de seleção. É necessário informar o atributo a ser alterado com o parâmetro $set.

```
db.colecao.update ( filtro , $set: atributo:valor )
```

Exemplo atualizando a coluna status do usuário para conteúdo “C” com coluna usuario_id = 1. O parâmetro multi como *true* permite autorizar todos os documentos que atendam essa condição.

```
db.usuarios.update(
{ idade:{ $gt: 25 },
{ $set: { status: "C" } },
{ multi: true } )
```

Exclusão de dados:

Para excluir dados é necessário utilizar o comando *REMOVE* a partir da coleção desejada selecionando o critério de exclusão na condição.

```
db.colecao.remove( critério )
```

Neste exemplo são eliminados todos os documentos que tenham no atributo `status` o valor `"D”.`

```
db.usuarios.remove( { status: "D" } )
```

- **Exemplo de uso**

lustraremos o uso do modelo de documentos com o site de treinamento e colaboração Pearson que utiliza o banco de dados MongoDB para armazenar documentos de treinamentos. (Matin, 2014).

Os materiais de treinamento, estudo e comunicação são armazenados como uma estrutura em árvore no formato JSON.

Inicialmente, com o seguinte comando

```
use treinamento
```

é criado um banco de dados denominado `treinamento`.

Em seguida é utilizado o comando *INSERT* numa coleção chamada artigo.

Caso a coleção já exista é inserido um novo documento, caso não exista é criada a coleção. São informados após o comando *INSERT* todos os atributos de um documento json na estrutura chave-valor.

```
db.artigo.insert([        {“artigo”:”1”,"material":
{"sumário": "These are some of the most well-known
drugs and their generic drug counterparts. ","subject":
"These are some of the drug names you will hear the
most. More to be added later "},"company":
"memorize.com",”email": "test@test.com","phone": "(111)
1111111111","address": "example street", "drugs":
["Tamiflu":”Oselttamivir”,"Oxycontin":”Oxycodone”}] }
)
```

O comando *FINDONE* é utilizado para recuperar o documento com a condição de que o atributo `articl`e tenha valor 1. O documento é recuperado na variável `documento` e a seguir o atributo *`prescription`* deste documento é modificado com o valor *`headache`*.

```
documento = db.artigo.findOne({article:1})
documento.prescription = "headache”
```

Em seguida esta modificação é efetivada no banco de dados utilizando o comando *SAVE.*

```
db.artigo.save(documento)
```

- **Consistência dos dados:**

Nos bancos de dados orientado a documentos, a consistência geralmente é eventual. No caso do MongoDB, toda aplicação se conecta a um servidor primário para operações de escrita, mas existem servidores secundários replicando esses dados. Todas as modificações realizadas por essa aplicação são garantidas no servidor primário. No entanto, motivado por aumento de desempenho, as aplicações podem opcionalmente acessar os servidores secundários mas nada garante que os dados acessados estejam atualizados. Por padrão os clientes apenas podem acessar a dados do servidor primário, mas isso afeta a escalabilidade. (Shashank, 2014)

O MongoDB suporta transações no nível de documento único, ou seja, uma única transação não pode envolver múltiplos documentos.

3.6 Modelo orientado a Grafos

- **Descrição do modelo:**

De modo geral o modelo orientado a grafos possui como componentes principais três estruturas básicas. A primeira estrutura é denominada vértice que pode ser vista como um nó, a segunda é denominada aresta que interliga os vértices e a terceira são as propriedades chave-valor dos vértices e arestas. Uma das vantagens da utilização do modelo baseado em grafos é a possibilidade de consultas complexas que envolvem exploração de uma sequência de relacionamentos que é resolvido com o percorrimento de múltiplos níveis do grafo. Este tipo de consulta no modelo relacional geralmente envolve uma sequência de operações de junção entre várias tabelas. Neste sentido, o modelo orientado a grafos resolve essas consultas com maior desempenho. Um banco de dados orientado a grafos geralmente implementa algoritmos complexos em grafos como por exemplo: o Caminho de Custo Mínimo Dijkstra (Robinson, 2013).

Fazendo um paralelo com o modelo relacional, num grafo uma tupla é um vértice, cada coluna se torna uma propriedade do vértice, e cada chave estrangeira se torna uma aresta.

Ao contrário do modelo relacional que precisa executar as junções em cada consulta, num grafo os relacionamentos já estão armazenados como ponteiros entre os vértices, portanto aplicações que necessitam de muitos relacionamentos e também percorrer esses relacionamentos em múltiplos níveis, se tornam candidatas a utilizar o modelo de grafo.

Um gerenciador de banco de dados que implementa esse modelo é o Neo4j (Miller, 2013)

Na figura 6 é possível verificar a diferença entre vértices, arestas, propriedades e rótulos de um grafo.

Cada vértice possui um identificador numérico e pode pertencer a um rótulo, além de conter propriedades com chaves e valores.

Os rótulos servem para classificar os vértices, sendo que as propriedades chave-valor servem para armazenar dados nos vértices.

Nas arestas é possível também definir propriedades com chaves e valores para armazenar dados.

É possível percorrer o grafo utilizando condições utilizando essas propriedades chave-valor, além de ser permitido criar índices utilizando essas propriedades para agilizar o percorrimento do grafo.

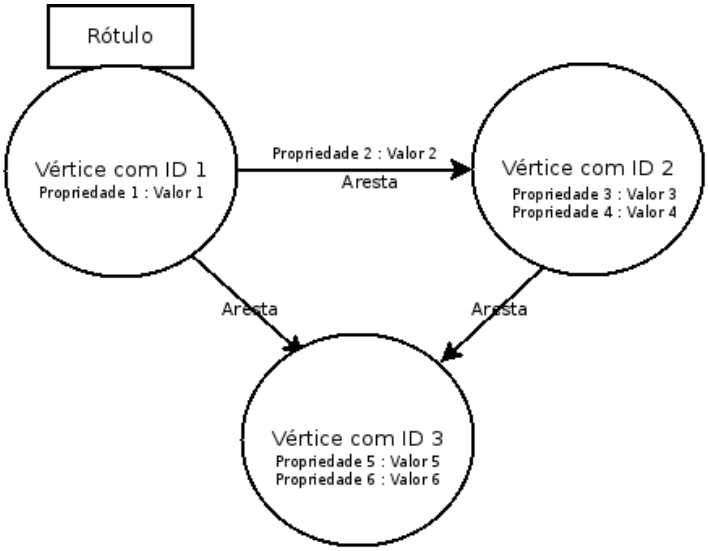

Figura 6. Modelo de grafo

- **Casos apropriados para uso**

  O modelo de grafo é ideal para dados conectados em múltiplos níveis com buscas dinâmicas, visando sempre a velocidade de acesso. (Sadalage e Fowler, 2013)

- **Restrições**

  Não é recomendável utilizar quando for necessário alterar todos os vértices ou um subconjunto deles porque é um processamento custoso em grafo, ou quando há particionamentos do grafo. (Sadalage e Fowler, 2013)

- **Operações no banco de dados Neo4j**

A seguir são apresentadas operações de criação de banco de dados, inserção, recuperação, alteração e exclusão de vértices e arestas em bancos de dados de grafos. Para ilustrar estas operações utilizaremos a linguagem Cypher do sistema de bancos de banco de dados Neo4j.

Criação do banco de dados:

Para criar um novo banco de dados no Neo4j é necessário fornecer um nome na conexão.

Inserção de dados:

Um novo vértice é criado no banco de dados fornecendo um rótulo e propriedades. Em Cypher é utilizado o comando *CREATE* informando qual é o rótulo a ser inserido no vértice seguido de uma sequência de propriedades chave-valor.

```
CREATE (variável:Rótulo { chave: valor }
```

No exemplo a seguir são criados dois vértices com rótulo `Usuário` e uma sequência de propriedades `nome, idade e status` com seus valores.

```
CREATE (n:Usuario { nome : 'Joao', idade : 45, status:
'A' })

CREATE (n:Usuario { nome : 'Jose', idade : 32, status:
'A' })
```

Para criar vértices e arestas ao mesmo tempo, ou seja, nunca única sentença Cypher, é necessário previamente definir quais os nós que serão conectados por essa aresta e a seguir utilizar o comando *`create`*.

```
CREATE (n)-[:ROTULO]→(m)
```

onde `n` e `m` são os vértices entre os quais está sendo criada uma aresta tendo como propriedade `ROTULO`

Caso os vértices já existam e seja necessário criar uma aresta entre eles, é necessário localizar os mesmos com o comando *MATCH* e em seguida criar a

aresta. A seguir é demonstrado como localizar dois vértices denominados a e b, criando em seguida uma aresta entre eles.

```
MATCH (a),(b)
WHERE a.propriedade = valor AND b.propriedade = valor
CREATE (a)-[r:ROTULO]→(b)
RETURN r
```

Neste exemplo, inicialmente são buscados os dois vértices que contenham propriedades `usuário` com valores `João` e `José.`

Em seguida é criada uma aresta entre esses vértices com um rótulo definido como `AMIGOS`.

```
MATCH (n),(m)
WHERE n.usuario = 'Joao'  AND m. usuario = 'Jose'
CREATE (n)-[r:AMIGOS]→(m)
RETURN r
```

Recuperação de dados:

A recuperação de dados em bancos de dados de grafos considera condições das propriedades dos vértices e das arestas. Para recuperar dados em Cypher é utilizado o comando *MATCH* onde são definidas as condições que devem satisfazer os vértices procurados. A cláusula *RETURN* informa a variável que na qual será retornado o resultado da recuperação.

```
MATCH variável:Rótulo WHERE condição RETURN variável
```

Neste exemplo é recuperado um vértice que possua rótulo `Usuario` e tenha um atributo nome com valor Joao. O vértice resultante da busca é armazenado na variável `p.`

```
MATCH (p:Usuario)
WHERE p.nome = 'Joao'
RETURN p
```

Neste outro exemplo são recuperados todos os vértices que tenham ambos como rótulo `Usuario,` e que estejam relacionados por uma aresta tal que o vértice inicial tenha um atributo nome com valor Joao.

```
MATCH (u:Usuario)-[:RELATED]→(urel:Usuario)
WHERE u.nome = 'Joao'
RETURN u,urel
```

Atualização de dados:

Para atualizar dados é necessário informar as condições dos vértices que serão atualizados assim como os atributos modificados com seus respectivos valores. Em Cypher é utilizado o comando *MATCH* junto com a cláusula *SET* que define os novos valores modificados.

```
MATCH variável:Rótulo WHERE condição SET atributo =
valor
```

No exemplo a seguir é atualizado o atributo `status` com novo valor `'C'` no vértice que tenha um atributo nome com valor Joao.

```
MATCH (n)
WHERE n.nome = 'Joao'
SET n.status = ‘C’
```

Exclusão de dados:

Para excluir dados é necessário definir a condição que devem satisfazer os vértices que serão excluídos. Em Cypher é utilizada a cláusula *DELETE* junto com o comando MATCH e a condição de exclusão pode estar tanto na cláusula *WHERE* ou no próprio filtro de condição da variável de vértice na cláusula *MATCH*.

```
MATCH variável:Rótulo WHERE condição DELETE variável
```

Neste exemplo são excluídos vértices e as arestas relacionadas a estes vértices que contenham o atributo status com valor ‘D’

```
MATCH (n { status: 'D' })-[r]-() DELETE n, r
```

Para excluir uma aresta é necessário informar os vértices conectados pela aresta, utilizando o comando DELETE na variável atribuída a aresta. No exemplo abaixo é excluída a aresta com rótulo RELATED entre os vértices que contenham os nomes Joao e Jose.

```
MATCH (n { nome: ‘Joao’ } )-[r:RELATED]-(m { nome:
‘Jose’ } )
DELETE r
```

- **Exemplo de uso**

Um exemplo de uso do modelo de grafos é um sistema de recomendação de produtos desenvolvido pela empresa Wallmart (Wada, 2013). Esta

aplicação exibe uma sugestão de produtos em tempo real e utiliza o banco de dados de grafos Neo4j para representar o relacionamento entre os produtos.

Para exemplificar como é possível implementar um sistema de recomendação com o Neo4j, vamos criar 3 produtos como vértices com arestas relacionando os mesmos.

Os comandos *CREATE* abaixo criam vértices com o rótulo `Produto` e a propriedade `desc` com o nome dos produtos.

```
CREATE (p1:Produto { desc : 'Mozart Vol. 05'})

CREATE (p2:Produto { desc : 'Wolfgang Amadeus
Mozart'})

CREATE (p3:Produto { desc : 'Jean Sibelius'})
```

Em seguida o comando *MATCH* é utilizado para buscar no grafo os vértices que contenham os nomes de produtos 'Mozart Vol. 05' e 'Wolfgang Amadeus Mozart' armazenando a referência aos vértices nas variáveis n e m respectivamente. Após localizar estes vértices, com o comando *CREATE* é criada uma aresta entre estes dois vértices encontrados, atribuindo a ela o rótulo *RELATED*.

```
MATCH (n),(m)
where n.desc = 'Mozart Vol. 05'
and m.desc = 'Wolfgang Amadeus Mozart'
CREATE (n)-[:RELATED]→(m)
```

Na sequência é realizada a mesma operação para criar uma aresta entre os vértices com os nomes de produtos 'Wolfgang Amadeus Mozart' e 'Jean Sibelius'

```
MATCH n, m
where n.desc = 'Wolfgang Amadeus Mozart'
and m.desc = 'Jean Sibelius'
CREATE (n)-[:RELATED]→(m)
```

Agora é possível percorrer o grafo buscando os produtos relacionados ao CD “`Mozart Vol. 05"` com o comando *MATCH*, obtendo o CD `'Wolfgang Amadeus Mozart'.` A busca filtra os rótulos *RELATED* ao percorrer as arestas.

```
MATCH (produto:Produto {desc:"Mozart Vol. 05"})-
[:RELATED]→(produtorel:Produto)
RETURN produto, produtorel
```

É possível realizar consultas percorrendo múltiplos níveis do grafo. No exemplo abaixo são obtidos os produtos relacionados ao CD “`Mozart Vol. 05"` num segundo nível de profundidade do grafo, recuperando neste caso o CD `'Jean Sibelius'.` Neste caso o operador “→” em Cypher indica qualquer direção de relacionamento. A partir do produto buscado pela condição, este operador permite buscar os vértices vizinhos conectados ao vértice produto, que no caso foi nomeado como produtorel. O resultado da execução do comando *MATCH* é retornado nas variáveis produto e produtorel, portanto exibe também o segundo nível de relacionamento entre os vértices.

```
MATCH (produto:Produto {desc:"Mozart Vol. 05"})-
[:RELATED]→(:Produto)←[:RELATED]-(produtorel:Produto)
RETURN produto, produtorel
```

- **Consistência dos dados:**

Os bancos de dados orientado a grafos também trabalham com consistência eventual ou fornecem opções de consistência total que geralmente implica em perda de desempenho.

O sistema de banco de dados Neo4j suporta as propriedades ACID apenas no servidor primário. Os clientes podem ler dados desatualizados caso sejam configurados para também acessar a servidores secundários até que todos os servidores estejam atualizados e consistentes. (Shashank, 2014)

O modelo de bancos de dados orientado a grafos é uma exceção entre os modelos NoSQL porque sua escalabilidade horizontal é mais complexa. Particionar diferentes partes do grafo em servidores diferentes provoca que as operações de percurso através das arestas sejam de alto custo. Devido à característica do modelo de alta interconexão entre os dados, para obter o melhor desempenho é preferível um menor número de servidores com maior poder de processamento.

(Jenson, 2015). No entanto, existem sistemas como Titan (Jenson, 2015) em que partes diferentes do grafo estão armazenadas em servidores diferentes. No caso do banco de dados Neo4j, cada servidor do *cluster* tem uma réplica completa do grafo.

3.7 Interface REST - Representational State Transfer

Cada gerenciador de banco de dados NoSQL oferece vários tipos de formas de acesso a seus dados. As principais interfaces são: API's de acesso para várias linguagens, drivers de banco de dados e webservices utilizando o padrão REST.

O acesso mais comum aos gerenciadores NoSQL é através da interface REST. O termo REST foi definido por (Fielding, 2000), e significa *Representational State Transfer*. Esta é uma arquitetura construída para servir aplicações em rede. A aplicação mais comum de REST é a própria *World Wide Web*, que utilizou a tecnologia REST como base para o desenvolvimento do HTTP.

Apesar de os SGBDs proverem outras interfaces, é recomendado o acesso através de webservices REST, principalmente porque os gerenciadores NoSQL podem ser acessados de qualquer dispositivo pela WEB, não tendo problemas com restrições de *firewall*. Outro fator importante é que os SGBDs NoSQL podem dispensar o uso de servidores de aplicação, podendo oferecer o acesso direto do *browser* ao *webservice* no próprio SGBD. (Shashank, 2014)

Podemos destacar as seguintes características dessa arquitetura:

- **Interface uniforme**

Cada recurso deve ter uma URI específica e coesa para poder ser acessado.

- **Representação dos recursos**

Nesta arquitetura os dados são devolvidos para a aplicação nos formatos HTML, XML, TXT e JSON, embora este último tornou-se um padrão devido a sua simplicidade e tamanho reduzido dos dados se comparado com o XML ou outros formatos.

- **Hypermedia**

Consiste no retorno de todas as informações necessárias na resposta para que o cliente consiga navegar, além de ter acesso a todos os recursos da aplicação.

- **Dispensável armazenamento de estado no servidor**

Nesta arquitetura as aplicações clientes podem trabalhar sem registro de sessões favorecendo a escalabilidade. Cada mensagem HTTP contém todas as informações necessárias para o pedido ser completado e compreendido. Assim os clientes e servidores não precisam gravar nenhum estado de suas comunicações (Shashank, 2014).

- **Menor volume de dados**

Os serviços WEB do antigo modelo SOAP são baseados em descritores WSDL, utilizam o protocolo SOAP e geralmente o formato XML para transporte e troca de dados. Esse formato é baseado na execução de operações.

Já no modelo REST, são recuperados recursos, e esses recursos são agrupados na forma de documentos, ligando todos os dados necessários entre si.

3.8 Conclusão do capítulo

Neste capítulo foram apresentados os modelos de dados NoSQL, ilustrados com alguns dos seus principais gerenciadores. Dada a especialização associada a estes modelos, existem aplicações que não se adéquam totalmente a um único deles. Frequentemente, estas aplicações modelam seus dados utilizando simultaneamente mais de um modelo de dados. A primeira geração dos gerenciadores NoSQL implementam apenas um modelo e, portanto, estas aplicações utilizam vários gerenciadores ao mesmo tempo. No próximo capítulo será apresentada esta classe de aplicações, denominadas de aplicações com persistência poliglota. Adicionalmente, serão introduzidos os gerenciadores de banco de dados NoSQL multi-modelos, que integram vários modelos de dados e são uma alternativa para aplicações com persistência poliglota.

Capítulo 4. Persistência poliglota e bancos de dados multi-modelos

A utilização simultânea de vários modelos de bancos de dados NoSQL trouxe desafios e problemas no desenvolvimento de um novo tipo de aplicações de bancos de dados, e essas aplicações são definidas como aplicações com persistência poliglota. Este capítulo caracteriza este tipo de aplicações assim como as abordagens para seu desenvolvimento e implementação. Como uma destas abordagens são apresentados os bancos de dados multi-modelos e dois dos gerenciadores para este tipo de banco.

4.1 Introdução

Atualmente há mais de duzentas implementações de bancos de dados NoSQL implementando diferentes tipos de modelos de dados. A questão de escolha sobre qual destes gerenciadores adotar é uma decisão complexa para arquitetos de software, principalmente porque cada um trabalha com um modelo de dados, possuem linguagens de consulta que não são padronizadas, e também possuem interfaces de acesso diferentes. Um fator de decisão frequente é o desempenho do modelo de dados que o gerenciador trabalha. É comum que uma aplicação necessite de mais de um modelo de dados ao mesmo tempo. Como consequência, essas aplicações trabalham com vários modelos de dados e vários gerenciadores de bancos de dados simultaneamente. O termo *persistência poliglota* foi introduzido para descrever esses requisitos.

A principal vantagem da adoção da persistência poliglota é ter mais desempenho no desenvolvimento e gerenciamento das aplicações, separando os requisitos da aplicação acessando diferentes gerenciadores de bancos de dados utilizando o modelo mais apropriado em cada caso.

Alguns dos gerenciadores de bancos de dados NoSQL surgiram como soluções para aplicações específicas, que tinham requisitos que não eram atendidos pelos bancos de dados relacionais. O Amazon DynamoDB e o Google BigTable são exemplos destas soluções. Um recurso comum de um gerenciador NoSQL é que eles implementam modelos simples e especializados de dados. Como consequência alguns problemas são melhores modelados por um modelo de dados do que outro.

Como um exemplo deste tipo de aplicação, consideremos um site que vende livros e outros produtos relacionados. Se o foco da aplicação são consultas relacionadas aos campos descrevendo os livros, então o modelo de documentos é adequado. Entretanto, se o foco da aplicação são consultas em diferentes níveis de similaridade entre livros, o maior desempenho pode ser conseguido utilizando o modelo de grafos. Se a aplicação requer as duas funcionalidades, o maior desempenho é conseguido utilizando simultaneamente o modelo de documentos e de grafos. A figura 7 descreve graficamente uma aplicação com persistência poliglota, na qual os dados precisam ser armazenados e gerenciados utilizando simultaneamente diferentes modelos de dados.

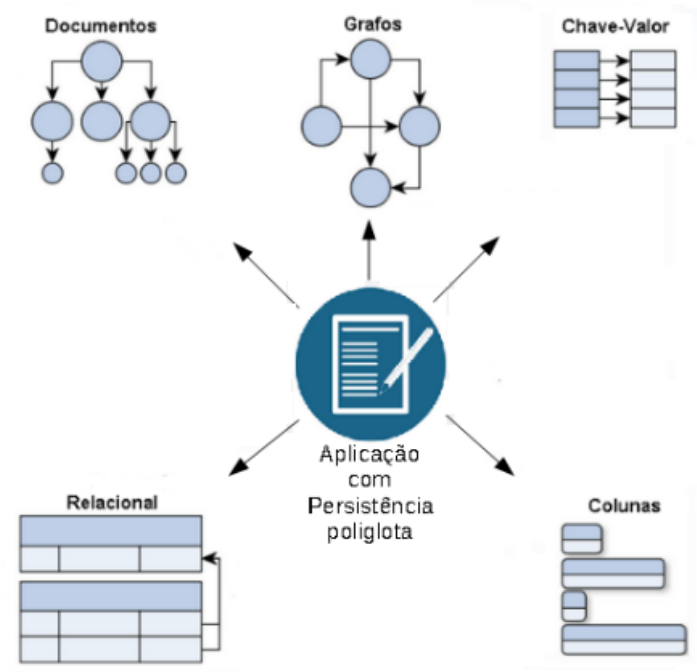

Figura 7. Persistência poliglota. A aplicação utiliza simultaneamente gerenciadores baseados em modelos de dados diferentes.

A necessidade das aplicações corporativas de trabalhar com múltiplos modelos de dados provoca um aumento significativo da complexidade de gerenciamento e manutenção operacional dos projetos. A causa fundamental é a necessidade de armazenar parte dos dados utilizando um gerenciador e parte em outro. As desvantagens desta solução são várias: perigos de inconsistência de dados, necessidade de replicação dos dados, complexidade no gerenciamento de

transações, uso de diferentes interfaces de acesso e ferramentas oferecidas por cada gerenciador, dentre outras.

A maioria dos gerenciadores NoSQL oferecem um modelo único de dados, que determina como eles serão organizados, armazenados e manipulados. Uma nova geração de SGBDs NoSQL foi criada para suportar múltiplos modelos de dados, em um único produto. Esses gerenciadores geralmente implementam os modelos de documentos, grafos e chave-valor num mesmo produto. (Aquila, 2016). Esses sistemas visam ser uma solução para aplicações de persistência poliglota e são denominados bancos de dados multi-modelo. Esses novos produtos focam na facilidade de desenvolvimento de aplicações que necessitam da persistência poliglota. No entanto, o uso destes sistemas também pode diminuir o desempenho das aplicações se comparados com as soluções tradicionais que utilizam gerenciadores NoSQL de modelos nativos, geralmente implementados com foco no desempenho. (Fowler, 2015)

4.2 Vantagens da persistência poliglota do ponto de vista da engenharia de software

A utilização da persistência poliglota pode ser vista com naturalidade quando é necessário selecionar qual é o meio mais produtivo, rápido e eficiente para armazenar e recuperar diferentes modelos de dados. Vários quesitos que merecem uma análise mais profunda, como mantenabilidade, desempenho e flexibilidade, são apresentados a seguir. Uma questão também importante que deve ser levada em conta, é a operação que vai manter o software desenvolvido, portanto é cada vez mais necessário pensar em Desenvolvimento e Operação como fatores interligados e indivisíveis, prática denominada recentemente como DevOps. (Freire, 2013)

A seguir são descritos alguns dos requisitos de engenharia de software que podem ajudar a identificar se a persistência poliglota é um caminho viável:

- M**antenabilidade**: é a propriedade que avalia se um software é fácil de ser mantido, melhorado, reparado e depurado. Mesmo que aplicações sejam escritas em linguagens já estabelecidas como Java ou C#, escolher um gerenciador NoSQL impacta muito nesta questão, ainda mais se a persistência poliglota for adotada. O acesso aos dados pode ocorrer por muitas interfaces diferentes, e com linguagens de consultas diferentes.

Escrever componentes que são executados diretamente nos gerenciadores NoSQL também é possível, portanto a persistência poliglota introduz também o código poliglota. Escolher um gerenciador NoSQL não é uma tarefa simples, ainda mais porque não há uma padronização nas ferramentas de administração ou linguagens de consulta. Avaliar se o produto tem maturidade, bom suporte, estabilidade e recursos suficientes em suas linguagens de consulta e componentes são fatores importantes para a mantenabilidade. Em (Dzhakishev, 2014) é avaliada a mantenabilidade de vários gerenciadores NoSQL em aplicações corporativas. O autor avalia que os gerenciadores MongoDB e Neo4j tem boas ferramentas, linguagens de consultas dotadas de muitos recursos, documentação e suporte, portanto são opções estáveis e maduras para aplicações corporativas. Alguns gerenciadores NoSQL adotam linguagens de consulta mais próximas ao SQL, como o Cassandra, Couchbase ou OrientDB, portanto oferecem uma transição mais amigável. Outros gerenciadores adotam sintaxes próprias em notação Javascript e chamada de métodos como o MongoDB, ou até mesmo encadeamento de comandos de leitura como o ArangoDB. Faz parte da mantenabilidade a produtividade que essas linguagens oferecem, e algumas são complexas ou muito imperativas, onde é necessário informar todas as operações necessárias num alto nível de detalhe. Alguns gerenciadores oferecem suporte ao Map-reduce permitindo explorar melhor o paralelismo, como no caso do MongoDB, CouchDB e Riak. É de grande importância analisar os recursos e produtividade destas linguagens antes de adotar um gerenciador NoSQL.

- **Flexibilidade**: um dos fatores que guiam para a persistência poliglota é a flexibilidade do esquema de dados. Não é incomum surgir a necessidade de uma aplicação mapear dados que não se adéquam perfeitamente adotando apenas um modelo de dados. Como exemplo podemos citar: dados do tipo financeiro sugerem o modelo relacional, por outro lado, dados com grande hierarquia e que não possuem esquema fixo, sugerem o modelo de documentos. Acessos rápidos de leitura e escrita para dados sugerem o modelo de chave-valor, e dados que são fortemente acoplados e que necessitam de busca em profundidade sugerem o modelo de grafos. Atividades como mineração de dados ou registro de logs de auditoria sugerem o modelo colunar. Podemos concluir que a estrutura dos dados é

um fator determinante na escolha de um modelo de dados e um gerenciador do tipo NoSQL.

- **Consistência**: A consistência nos gerenciadores NoSQL geralmente é chamada de eventualmente consistente. Isso significa que os desenvolvedores tem que trabalhar com a possibilidade de recuperar dados desatualizados, definir programaticamente quais dados são necessários ter uma consistência forte e até lidar com conflitos de gravação. Cada gerenciador NoSQL tem suas opções de consistência, e antes de adotar qualquer gerenciador é necessário estudar se ele apresenta o recurso necessário para a aplicação. Outro ponto de atenção é ter dados consistentes em gerenciadores NoSQL diferentes, com dados quem podem ficar replicados entre gerenciadores, como no caso de aplicações de persistência poliglota. No caso dos gerenciadores NoSQL, é muito mais fácil ter falhas de software com dados desatualizados, porque a premissa que o gerenciador de banco de dados garante esse requisito, não é mais válida, fazendo com que o desenvolvedor de software tenha mais responsabilidade ao gerenciador as operações de leitura e escrita dos dados.

- **Desempenho**: O desempenho é outro fator chave para a adoção da persistência poliglota. Processar dados que nem sempre são estruturados, requer modelos de dados especializados, e esses modelos são implementados por gerenciadores NoSQL também especializados em oferecer o melhor desempenho.

- **Disponibilidade**: a disponibilidade implica não apenas o particionamento e replicação de dados, mas também o tempo de recuperação de nós que possam ser comprometidos. Em (Nelubin, 2013) o autor avalia o tempo de recuperação de nós que falharam. O autor chega a conclusão que gerenciadores como o MongoDB e Aerospike possuem um bom tempo de recuperação, e que gerenciadores como o Couchbase e Cassandra demoraram mais para recuperar os nós que falharam.

- **Segurança**: Verificar se o gerenciador possui um histórico de muitas falhas de segurança também é importante. É interessante notar que esses gerenciadores NoSQL trabalham como servidores de aplicação, atendendo

requisições HTTP diretamente. Ataques no gerenciador MongoDB através de injeção de código Javascript já ocorreram, fazendo com que condições fossem anuladas ou alteradas, semelhante ao ataque de SQL Injection (Sullivan, 2011). Alguns gerenciadores NoSQL também vem por padrão com autenticação fraca, permitindo a exploração desta vulnerabilidade, como é o caso do MongoDB que permitiu ataques em larga escala. (Barry, 2017)

Após os requisitos iniciais serem identificados, questões de engenharia de software como modelo de dados, componentes, transformação de dados, testes automáticos e administração, devem ser questionados também:

- **Identificando qual gerenciador NoSQL utilizar**

  Os diferentes tipos de modelos de dados NoSQL apresentam semelhanças entre si e podem gerar dúvidas sobre qual adotar. Em (Bermbach, 2014) é proposto que o modelo de documentos ou o modelo de colunas tenha uma preferência ao mapear dados, pensando em consistência, flexibilidade e desempenho. Em (Abramova, 2014) é discutido que adotar o modelo de chave-valor realmente maximiza o desempenho, mas que o mapeamento dos dados pode ficar comprometido. Naturalmente o modelo de chave-valor vem se aproximando mais com o modelo de documentos para suprir essas deficiências, mas esta evolução pode impactar no desempenho. Um gerenciador NoSQL do tipo multi-modelo pode ser uma alternativa viável, mas é necessário verificar se o desempenho é adequado para a aplicação.

- **Micro serviços**

  A recomendação ao se trabalhar com persistência poliglota é o uso de micro serviços (Fowler, 2014). Um micro serviço deve oferecer os dados de um requisito funcional do sistema, e um micro serviço geralmente deve consultar apenas um gerenciador NoSQL. O acesso de um micro serviço ocorre pela interface REST, que já mapeia códigos de retornos HTTP e seu resultado é padronizado no formato JSON. A orquestração de chamadas de micro serviços deve ser feita um ESB (*Enterprise Service Bus*) que já lida

com chamadas e retornos, transformando e roteando os dados recuperados. Em (Baraiya, 2016) é descrito um novo orquestrador de micro serviços denominado Conductor. Este orquestrador foi implementado pela empresa Netflix, e o mesmo lida com fluxo de execução, gerenciamento de processos, sincronização, filas e canais de comunicação. É viável que um gerenciador NoSQL do tipo multi-modelo possa simplificar o modelo de micro serviços, tendo como vantagens:

- Armazenamento dos dados e lógica de negócio num único gerenciador, com uma única API.

- Melhor desempenho de rede, porque os clientes buscam os dados num único gerenciador, reduzindo o número de requisições necessárias para obter dados.

- Maior segurança aos dados sensíveis, já que os dados e serviços estão centralizados num único gerenciador.

- Documentação mais simples, porque apenas uma API e linguagem de consulta é utilizada, resultando em documentação gerada automaticamente através de anotações em alguns gerenciadores.

Um problema que pode ocorrer nos gerenciadores multi-modelo, é a falta de reusabilidade, porque como os dados estão todos disponíveis num único gerenciador, os desenvolvedores podem abusar da linguagem de consulta, não escrevendo componentes reutilizáveis. Não é difícil perder o conceito de micro serviços, colocando a lógica do negócio diretamente na linguagem de consulta, com consultas pesadas e difíceis de depurar.

- **Recuperação e transformação de dados de múltiplas fontes**

  É recomendado um orquestrador de serviços para recuperar os dados de diferentes gerenciadores, conforme relatado no item anterior. É possível manipular também esses retornos com linguagens de consultas JSON com a JsonPath (Baeldung, 2017). Um gerenciador NoSQL do tipo multi-modelo oferece linguagens de consultas que já recuperam e combinam os dados entre diferentes modelos, podendo facilitar o desenvolvimento de aplicações que necessitem de persistência poliglota.

- **Testes automáticos**

  Cada micro serviço deve ser testado de forma isolada, sendo que posteriormente é necessário fazer um teste de integração acessando todos os serviços. No caso dos gerenciadores multi-modelo, é necessário o foco da equipe em desenvolver micro serviços pequenos e direcionados, não colocando toda a lógica de negócio em linguagens de consulta, o que pode dificultar os testes automáticos, principalmente em unitários.

- **Administração de múltiplos gerenciadores NoSQL**

  Este é um dos principais problemas ao se adotar a persistência poliglota com múltiplos gerenciadores NoSQL. É necessário treinar desenvolvedores e administradores destes gerenciadores, portanto há um alto custo envolvido. Os gerenciadores NoSQL multi-modelo oferece esta vantagem de trabalhar com apenas um produto e linguagem de consulta.

  Uma alternativa é trabalhar com persistência poliglota utilizando uma solução integrada na nuvem. A Amazon Web Service ou Microsoft Azure, por exemplo, oferecem soluções integradas para persistência poliglota.

  Citando os serviços da Amazon, é possível mapear os dados utilizando os seguintes gerenciadores:

  - DynamoDB: trabalha com o modelo de chave-valor, documentos e até grafo (processado pelo Titan graph e armazenado com DynamoDB)

  - ElastiCache: trabalha com cache de dados de chave-valor compatível com o gerenciador Redis

  - S3: armazena qualquer tipo de arquivo

  - RDS: trabalha com banco de dados relacional (SQL Server, Oracle, MySQL ou PostgreSQL)

  - Redshift: trabalha com data warehouse de alto desempenho

A persistência poliglota introduz vários pontos de atenção na engenharia de software. É necessário avaliar com cuidado quais gerenciadores NoSQL utilizar, seus recursos, seu suporte, como garantir a consistência dos dados,

escalabilidade, como desenvolver testes automáticos e como manter toda essa infraestrutura.

Apesar do movimento NoSQL ter surgido há mais de uma década, sua adoção ainda é considerada lenta, principalmente pelos pontos de engenharia de software apresentados acima, além do tradicionalismo. Os gerenciadores NoSQL são muito populares em startups ou empresas que operam na modalidade Cloud, mas administradores de bancos de dados ainda evitam sua adoção em corporações mais tradicionais.

A empresa SAP com seu gerenciador de banco de dados HANA (Appleby, 2014) está quebrando este paradigma. Seu gerenciador trabalha com recursos comuns nos gerenciadores NoSQL, tais como: gerenciador com banco de dados em memória, modelo orientado a coluna, modelo de grafos, além do modelo relacional. Executar softwares complexos como um ERP num gerenciador que não implementa o modelo relacional tradicional é uma disruptura significativa. Gerenciadores relacionais como o Microsoft SQL Server e Oracle também estão inovando para trabalhar com documentos JSON, trabalhar com bancos de dados em memória, ou até mesmo oferecer recursos de grafos, portanto é possível que essa nomenclatura de gerenciadores NoSQL não seja mais viável no futuro.

As novas necessidades vão continuar introduzindo novos recursos nos gerenciadores de bancos de dados, mas a flexibilidade e desempenho demandados estão gerando mais complexidade e desafios na engenharia de software. É muito importante que os arquitetos e desenvolvedores se adéquem a essas novas necessidades, sendo capazes de selecionar quais são os melhores recursos ou produtos para cada caso, tendo como foco parâmetros como mantenabilidade, qualidade e desempenho.

4.3 Exemplo de aplicação com persistência poliglota

Um exemplo de aplicação com persistência poliglota que tem sua arquitetura publicada na internet, é o aplicativo *Squire* (Perdomo, 2013) disponível para *iPads*. Essa aplicação é um sistema de busca e recomendação de filmes e shows, permitindo os usuários contribuírem com comentários, sugestões, criação de grupos, classificação e votação de filmes, dentre outras facilidades.

A arquitetura de *backend* dessa aplicação utiliza o SGBD baseado em documentos MongoDB através da interface REST com transporte de dados JSON, e também utiliza o SGBD para grafos Neo4j.

No SGBD de documentos MongoDB são armazenados os usuários, sessões, filmes, shows, contatos, sugestões e recomendações. No SGBD orientado a grafos Neo4j são armazenados os relacionamentos dos filmes e shows, relacionamento dos usuários, e as notas permitindo as buscas entre os *ratings* dos filmes.

Conforme apresentado na figura 8, o aplicativo no iPad faz as requisições no servidor de aplicação principal, quando for trabalhar com dados no modelo documentos para pesquisar os filmes e shows. Quando for necessário buscar as recomendações de filmes, o servidor de aplicação principal coloca numa fila de mensagem uma requisição que é atendida pelo SGBD Neo4j, percorrendo o grafo.

Quando é necessário fazer atualização entre os dois SGBDs NoSQL, o de documentos e grafos, os dados também são colocados num serviço de fila que ambos produtos consomem e atualizam seus dados. Essa arquitetura pode ser eficiente, mas os dados transitam entre dois gerenciadores de bancos de dados NoSQL diferentes, cada um atendendo um propósito diferente. Provavelmente um SGBD multi-modelo ajudaria concentrar os dados e transações num único produto, podendo inclusive fazer o particionamento (*sharding*) para distribuir os dados se necessário.

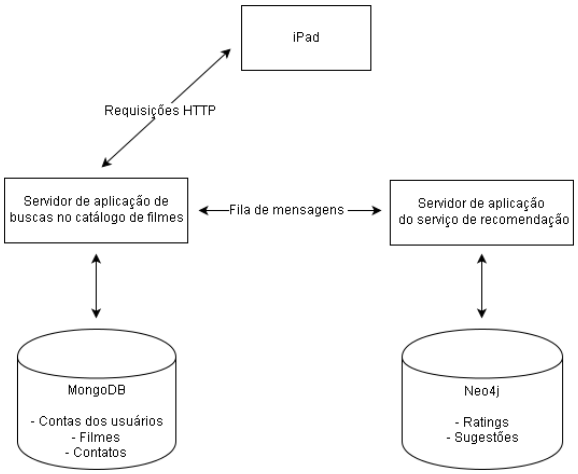

Figura 8. Squire - uma aplicação que utiliza persistência poliglota

4.4 Gerenciadores de bancos de dados multi-modelo

Os SGBDs multi-modelo visam suprir a necessidade de oferecer vários modelos de dados NoSQL ao mesmo tempo, num mesmo produto, permitindo ter alto desempenho, particionamento (*sharding*) e flexibilidade. Obviamente esse modelo é muito atrativo porque evita duplicação de dados, permite transações entre vários modelos de dados, proporciona o gerenciamento de um único produto, além de contar com o suporte de um único fabricante. Apesar da variedade de soluções ser benéfica permitindo o desenvolvimento e competição sadia entre produtos e empresas, manter os dados de uma aplicação entre vários produtos diferentes, com desempenho, disponibilidade e integridade, certamente não é uma tarefa fácil.

Geralmente, um gerenciador de banco de dados multi-modelo consegue trabalhar com documentos, grafos e chave/valor oferecendo recursos de relacionamento entre os modelos de dados (Fowler, 2015). Uma das diferenças fundamentais entre os diferentes gerenciadores multi-modelo é a estratégia diferente para a representação e implementação dos grafos. A seguir apresentaremos detalhes dos dois sistemas de gerenciamento multi-modelo que serão utilizados neste trabalho. São eles o OrientDB e o ArangoDB.

4.4.1 OrientDB

O OrientDB é um gerenciador de banco de dados multi-modelo de código aberto, escrito em Java, que implementa um modelo de dados com documentos, grafos, objetos e chave/valor. Ele suporta transações, particionamento (*sharding*) e replicação, funções do lado do servidor, como *stored procedures* que podem ser escritas em SQL ou JavaScript, uma API REST, uma linguagem de consulta muito semelhante ao SQL chamado OrientDB SQL que suporta as operações de grafos, filtros, junções, projeções, ordenações, agrupamento, funções de agregação, união e intersecção. (Garulli, 2012)

O OrientDB suporta todas as propriedades ACID, garantindo consistência completa em todos os servidores. Ele é implementado com um mecanismo onde vários servidores estão aptos para atender as requisições. No OrientDB a representação básica dos dados é baseada em um grafo implementado com

relações diretas entre os vértices. Esta estrutura de armazenamento é denominada "*Index Free Adjacency*". (Aquila, 2016). Um diagrama desta estrutura é apresentado na figura 9 onde vemos que para cada vértice, são armazenados ponteiros e chaves de todos os vértices adjacentes representados de V1 a V6 no exemplo. Em OrientDB (Garulli, 2012) é descrito que cada vértice possui um identificador de registro RID que corresponde a sua posição física de endereço de registro dentro do banco de dados. Com o RID é possível carregar um vértice rapidamente apenas com uma função de busca (*seek*) no endereço do banco de dados. Como em cada vértice estão armazenadas as arestas e em cada uma delas estão contidos os RID dos vértices adjacentes de entrada e saída (*in* e *out*), basta fazer uma operação de busca (*seek*) destes identificadores para localizar estes vértices adjacentes. No Anexo 1 é apresentada uma análise do código fonte do OrientDB

Esta estrutura tem o melhor desempenho porque as relações estão armazenadas nos próprios vértices. Este fato evita a busca dos vértices adjacentes em estruturas de índice, como acontece na representação de grafos em sistemas relacionais tradicionais, nos quais as relações entre vértices são consultadas através de operações de junção. Este tipo de estrutura garante bom desempenho ao percorrer um grafo em profundidade, mas pode consumir mais espaço no armazenamento em disco para armazenar as relações em cada vértice. Os gerenciadores que trabalham com essa estrutura de armazenamento são denominados de grafos nativo, ou grafos reais.

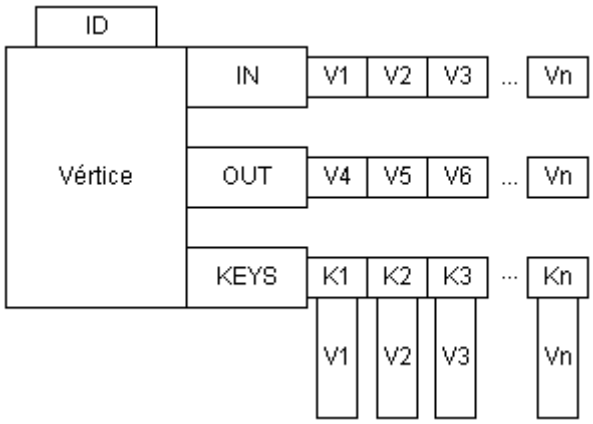

Figura 9. Estrutura de armazenamento de grafos nativo

O OrientDB possui várias classes de objetos de armazenamento, sendo que cada uma das classes é uma especialização da anterior definindo uma hierarquia.

Um *OIdentifiable* é a menor unidade de armazenamento do OrientDB, e ele pode ser classificado em 4 tipos:

- *Document*: é o tipo mais flexível que permite trabalhar com um esquema definido ou sem esquema. Ele permite uma estrutura em árvore de chave-valor chamada de JSON. Essa estrutura também suporta relacionamentos entre documentos.
- *RecordBytes*: Tipo de dados binário de armazenamento denominado BLOB
- *Vertex*: É um tipo especial de *OIdentifiable* para armazenar os vértices do grafo. Ele também é um documento permitindo o armazenamento de estrutura JSON.
- *Edge*: É um tipo especial de *OIdentifiable* representando arestas que conectam dois vértices do grafo.

O OrientDB suporta a definição de classes, que por sua vez estrutura o esquema de *Records*, que herda de *OIdentifiable*. Quando uma classe é definida, é criada uma estrutura chamada *Cluster*. Um *Cluster* é uma área de armazenamento que agrupam *Records*. É possível definir uma classe que armazena seus dados em diversos *Clusters*, conseguindo a vantagem de distribuir os dados de cada *Cluster* em servidores diferentes. Com esse recurso é possível particionar os dados de uma Classe, como numa estrutura de filiais de uma empresa, ou até mesmo um histórico de registros por ano, onde cada ano é salvo num *cluster* específico. É uma boa prática também dividir os *clusters* entre estruturas de discos diferentes para um melhor desempenho.

- **Operações no banco de dados OrientDB**

Em seguida são apresentadas as operações de criação de banco de dados, inserção, recuperação, alteração e exclusão de vértices e arestas no banco de dados OrientDB. O OrientDB suporta uma configuração que trabalha apenas com

documentos que não é o foco do estudo. Nas operações abaixo o foco é a configuração que gerencia grafos que suportam documentos.

Criação do banco de dados:

Para criar um novo banco de dados no OrientDB é necessário utilizar o comando *CREATE DATABASE* informando o novo nome do banco de dados.

```
CREATE DATABASE nome_banco_dados
```

Inserção de dados:

Para inserir vértices com os documentos é necessário utilizar o comando *CREATE* VERTEX informando a classe que é uma separação lógica do banco de dados, informando também as propriedades chave-valor individualmente, ou utilizando um documento JSON como parâmetro.

```
CREATE VERTEX nome_classe set chave = valor
```

```
CREATE VERTEX nome_classe CONTENT { "chave" : "valor", "chave" : "valor" }
```

Para criar arestas é utilizado o comando *CREATE* EDGE para relacionar dois vértices. Geralmente, os vértices que serão conectados pela aresta são definidos com buscas utilizando o comando *SELECT c*om as condições de busca

```
CREATE EDGE FROM (SELECT FROM nome_classe WHERE condição TO (SELECT FROM  nome_classe WHERE condição)
```

No exemplo a seguir são criados dois vértices com rótulo Usuário, definindo propriedades nome, idade e status.

```
CREATE VERTEX Usuario set Usuario='Joao', idade=45, status='A'
```

```
CREATE VERTEX Usuario set Usuario='Jose', idade=32, status='A'
```

A seguir, pode ser criada uma aresta a partir de duas consultas que recuperam estes vértices pela propriedade usuário com valores `João` e `José.` O rótulo definido na aresta foi denominado `AMIGOS`.

```
CREATE EDGE FROM (SELECT FROM Usuario WHERE
Usuario='Joao'  TO (SELECT FROM  Usuario WHERE
Usuario='Jose') SET label = ‘AMIGOS’
```

Recuperação de dados:

Para recuperar dados é necessário utilizar o comando *TRAVERSE* referenciando a classe que será lida, as condições para percorrer o grafo e o retorno.

```
TRAVERSE * FROM select_classe WHERE condição WHILE
condição_profundidade
```

Exemplo recuperando um vértice com atributo usuario_id com valor 1 com nível de profundidade ao percorrer o grafo <=3.

Note-se que este vértice recuperado realmente corresponde a um documento.

```
TRAVERSE * FROM (select from Usuario) where id = 1 while
$depth<=3
```

Atualização de dados:

Para atualizar dados é necessário utilizar o comando *UPDATE* informando a classe, condição de busca no grafo e os atributos que serão alterados com seus respectivos valores na cláusula SET.

```
UPDATE classe
SET campo = conteúdo
WHERE condição
```

Exemplo atualizando a coluna status do usuário para conteúdo “C” com propriedade id = 1

```
UPDATE Usuario
SET status = "C"
WHERE id = 1
```

Exclusão de dados:

Para excluir dados é necessário utilizar o comando *DELETE* a partir da classe desejada selecionando o critério de exclusão na condição *WHERE*.

```
DELETE FROM classe
WHERE condição
```

Exemplo de exclusão de vértices e arestas que contenham vértices com a propriedade status com valor ‘D’

```
DELETE FROM usuarios
WHERE status = "D"
```

- **Consistência dos dados:**

  O banco de dados OrientDB trabalha com consistência forte utilizando as propriedades ACID suportando transações distribuídas. Ele suporta o conceito de multi-master que permite que todas os servidores atuem também como escrita.

Como já citado anteriormente, os bancos de dados orientado a grafos tem limitações na sua escalabilidade. O OrientDB suporta o conceito de *sharding* (particionamento) por classe permitindo então a divisão do grafo entre servidores, mas obviamente isso impacta seriamente no desempenho.

4.4.2 ArangoDB

O ArangoDB é um banco de dados com sistema de código aberto escrito em C++ que implementa um modelo de banco de dados com documentos, grafos e chaves/valor. Ele suporta transações, particionamento (sharding) e replicação, possui uma linguagem denominada Foxx que é baseado em Javascript para desenvolver componentes do lado do servidor. Ele suporta a API REST, uma linguagem de consulta chamada AQL que permite junções, operações em grafos, iterações, filtros, projeções, ordenações, agrupamento, funções de agregação, união e intersecção. O ArangoDB suporta todas as propriedades ACID, mas apenas no servidor principal que é o *Master*, permitindo que leitura de dados inconsistentes ocorra em servidores desatualizados. No caso do ArangoDB, conforme demonstrado na figura 10, o armazenamento do grafo é feito no modelo de documentos, com documentos ligando outros documentos através de referências entre as coleções de documentos. Esses documentos especiais que armazenam as ligações entre os documentos vértices são chamados de documentos *Edge* (Arestas), que possuem atributos denominados _from e _to com os identificadores dos vértices. Os algoritmos de grafos foram adaptados para usar

essas estruturas, utilizando buscas com índices apropriados para acelerar o percorrimento. (Weinberger, 2016)

Para uma melhor utilização do ArangoDB foi necessário analisar seu código fonte. Como resultado dessa análise concluímos que o ArangoDB possui um mecanismo de armazenamento denominado *Shape* que permite compartilhar estruturas comuns de documentos, isto é, o mesmo esquema de atributos. Quando um documento é criado pela primeira vez é possível armazenar seu *Shape*. Os próximos documentos que contenham a mesma estrutura apenas criam ponteiros para o *Shape*, permitindo assim economia no armazenamento de dados. O resultado completo desta análise é apresentado no anexo 2.

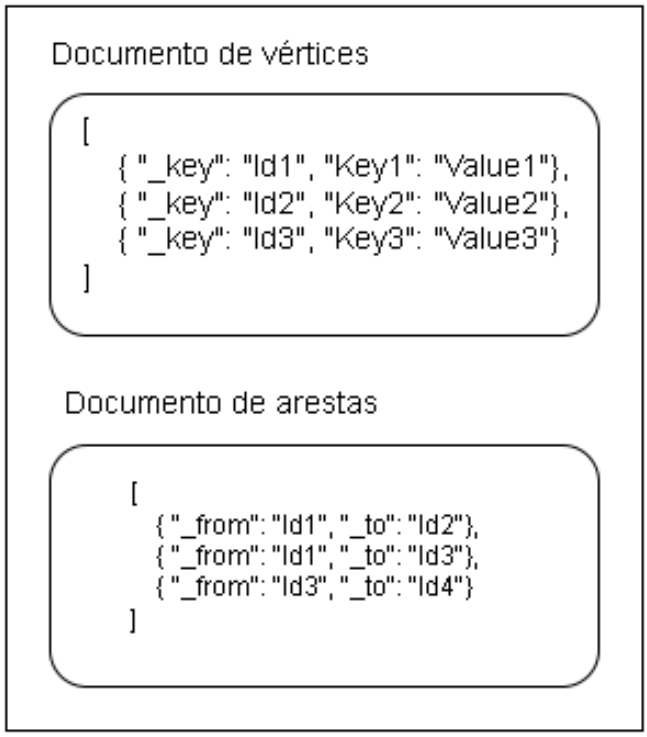

Figura 10. Estrutura de armazenamento de grafos utilizando documentos em ArangoDB

É possível classificar o OrientDB e o ArangoDB, como SGBDs que trabalham com grafos porque permitem armazenar vértices e arestas com propriedades, e oferecem funcionalidades para o uso de grafos. Apenas o

OrientDB possui um armazenamento de grafos do tipo nativo, porque ele utiliza o conceito de “*index-free adjacency*”, já o ArangoDB tem uma camada de grafos em cima do armazenamento nativo de documentos, que necessita pesquisar índices. (Fowler, 2015)

O OrientDB ocupa mais espaço em disco porque armazena os ponteiros para criar a estrutura “*index-free adjacency*”, e no caso do ArangoDB, o espaço ocupado é menor também porque salva os grafos em documentos, e através do mecanismo *Shape* consegue armazenar o esquema dos documentos apenas uma vez. (Fowler, 2015)

- **Operações no banco de dados ArangoDB**

Em seguida são exibidas as operações de criação de banco de dados, inserção, recuperação, alteração e exclusão de vértices e arestas no banco de dados ArangoDB. O ArangoDB trabalha com documentos sendo que seu *engine* de grafo foi adaptado ao modelo de documentos.

Criação do banco de dados:

Para criar um novo banco de dados no ArangoDB é necessário utilizar o comando *createDatabase* informando o novo nome do banco de dados.

```
db._createDatabase("banco_de_dados");
```

Inserção de dados:

Para inserir vértices com os documentos é necessário utilizar o comando *INSERT* informando a coleção de documentos que é uma separação lógica do banco de dados, informando também um documento JSON como parâmetro.

```
INSERT { Chave : 'Valor', Chave : 'Valor'} IN
Coleção_documentos
```

Para criar arestas é necessário buscar os vértices que serão conectados por uma aresta pelo comando *FOR*, informando as condições de busca e em seguida utilizando o comando *INSERT* na coleção especial *Edge* para relacionar ambos os vértices.

```
LET from = (FOR p IN Coleção_Documentos FILTER filtro
RETURN p._id) LET to=(FOR p IN Coleção_Documentos FILTER
filtro RETURN p._id) INSERT { from: from[0], to: to[0]}
INTO Coleção_Edge
```

Exemplo criando dois vértices com rótulo Usuário, definindo propriedades nome, idade e status.

```
INSERT { Usuario : 'Joao', idade : '45'} IN  Usuario

INSERT { Usuario : 'Jose', idade : '32'} IN  Usuario
```

Exemplo buscando dois vértices que contenham propriedades usuário João e José e em seguida criando uma aresta entre os vértices. O rótulo definido na aresta foi denominado AMIGOS.

```
LET from = (FOR p IN Usuario FILTER p.nome == 'Joao'
RETURN p._id) LET to=(FOR p IN Usuario FILTER p.nome ==
'Jose' RETURN p._id) INSERT { _from: from[0], _to:
to[0], type: 'AMIGOS' } INTO UsuariosRelated
```

Recuperação de dados:

Para recuperar dados é necessário utilizar o comando *FOR* com a cláusula *TRAVERSAL* referenciando a coleção que será lida, a coleção *Edge* que armazena as arestas, as condições para percorrer o grafo e o retorno.

```
LET from = (FOR u IN coleção_documentos FILTER
fitro_vértice_inicial' RETURN u._id) FOR p IN
TRAVERSAL(coleção_documentos, coleção_arestas, from[0],
'outbound', {  minDepth: 0,   maxDepth: 3,   paths: true
}) RETURN {   vertices: p.path.vertices[*]._id,
edges: p.path.edges[*]._id }
```

Exemplo recuperando um vértice com atributo usuario_id com valor 1 com nível de profundidade ao percorrer o grafo <=3

```
LET from = (FOR u IN Usuarios FILTER u.usuario_id ==
'1' RETURN u._id) FOR p IN TRAVERSAL(Usuarios,
UsuariosRelated, from[0], 'outbound', {  minDepth: 0,
maxDepth: 3,   paths: true }) RETURN {   vertices:
p.path.vertices[*]._id,   edges: p.path.edges[*]._id }
```

Atualização de dados:

Para atualizar dados é necessário utilizar o comando *UPDATE* informando a coleção de documentos, condição de busca e os atributos que serão alterados com seus respectivos valores na cláusula *SET*.

```
UPDATE {condição}
WITH { campo: "valor" }
IN coleção_documentos
```

Exemplo atualizando a coluna status do usuário para conteúdo “C” com propriedade id = 1

```
UPDATE {id = 1}
WITH { status: "C" }
IN Usuarios
```

Exclusão de dados:

Para excluir dados é necessário utilizar o comando *REMOVE* a partir da classe desejada selecionando o critério de exclusão.

```
REMOVE  {condição}
IN Coleção_documentos
```

Exemplo de exclusão de vértices e arestas que contenham vértices com a propriedade status com valor ‘D’

```
REMOVE  {status: “D”}
IN Usuarios
```

- **Consistência dos dados:**

O banco de dados ArangoDB trabalha com consistência forte utilizando as propriedades ACID em múltiplos documentos, mas apenas sem operar em *cluster*, ou seja, com apenas um servidor.

No modo *cluster* as transações que operam apenas num documento operam com as propriedades ACID, mas no modo multi-documento não, é suportado.

O ArangoDB trabalha muito bem com múltiplos servidores num *cluster* suportando *sharding* (particionamento).

4.5 Exemplo de mapeamento de persistência poliglota

No exemplo abaixo é mapeada uma aplicação que trabalha com persistência poliglota. Em cada vértice do grafo está armazenado um documento que descreve um cliente de restaurante ou um restaurante. Nos documentos dos clientes estão informados os dados do cliente e suas preferências culinárias. Nos documentos dos restaurantes estão descritos os dados dos restaurantes e suas especialidades culinárias.

No grafo estão relacionados os usuários que aprovam determinados restaurantes, além de usuários que seguem outros usuários por terem se identificados com as preferências de restaurantes.

Na Figura 11 há um exemplo desse mapeamento do grafo.

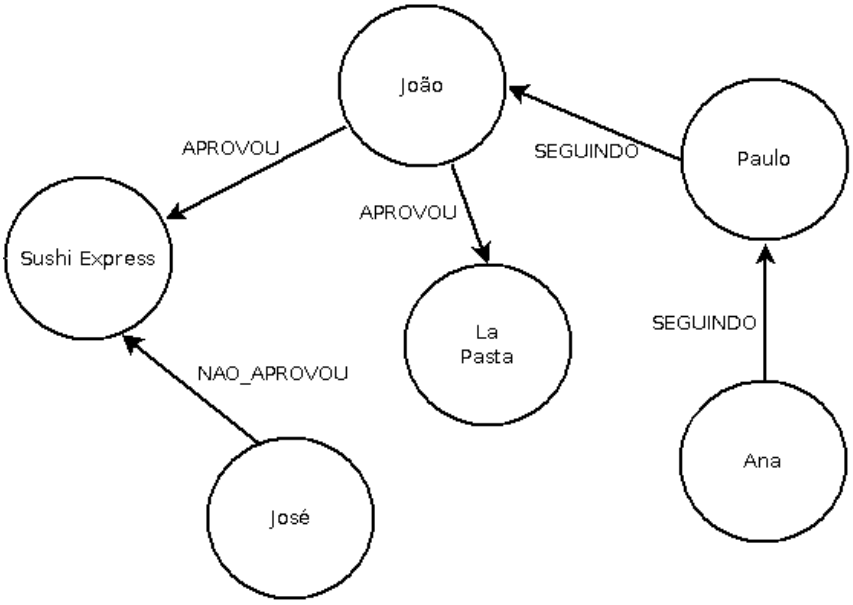

Figura 11. Exemplo de mapeamento de persistência poliglota

- Abaixo estão relacionados alguns exemplos de documentos que identificam os usuários e restaurantes.

```
{
  "nome": "José",
  "idade": "45",
  "email": "jose@email.com",
```

```
  "profissao": "arquiteto",
  "preferencias": [
    {"culinaria":"japonesa"},
    {"culinaria":"italiana"},
    {"vinho”:"seco"}
  ]
}

{
  "restaurante": "Sushi Express",
  "site": "www.sushiexpressfoodstest.com",
 "enderecos": [
        {
            "descricao": "Unidade 1",
            "rua": "Rua Liberdade 10",
            "cidade": "Sao Paulo",
            "estado": "SP”,
            "cep": “90000”
        },
        {
            "descricao": "Unidade 2",
            "rua": "Rua Liberdade 20",
            "cidade": "Sao Paulo",
            "estado": "SP”,
            "cep": “90001”
        }
    ],
    "telefones": [
        {
            "descricao": "Atendimento central",
            "numero": "555-1234"
        },
        {
            "descricao": "Financeiro",
            "numero": "555-1235"
        }
    ],
  "especialidades":  [
    {"prato":"Sushi"},
    {"prato":"Yakitori"},
    {"prato”:"Tonkatsu”},
    {"bebida”:"Saquê”},
    {"bebida”:"Ryokucha”}
]
}
```

Apresentamos uma sequência de consultas que ilustram uma aplicação de persistência poliglota que utiliza tanto a representação de grafos como de documentos.

Suponha que primeiramente são recuperados todos os documentos associados a restaurantes que oferecem como prato Sushi. Em seguida o grafo é percorrido filtrando aquelas arestas com o rótulo “APROVOU” de todos os restaurantes “Sushi Express”. Notes-se que serão recuperados e exibidos todos os documentos dos usuários que aprovaram este restaurante. Em uma terceira consulta, todos os usuários que estão seguindo o João são exibidos, portanto é uma operação de percorrimento no grafo até o nível 3.

A seguir são apresentados os comandos para essas consultas tanto em OrientDB como em ArangoDB .

- Consulta acima descrita no OrientDB:

```
select from restaurantes where prato containstext
'sushi'

traverse out('APROVOU') from (select from restaurante
where nome     ='Sushi Express')

traverse out('SEGUINDO') from (select from pessoa where
nome ='João') WHILE $depth <= 3
```

- Consulta acima descrita no ArangoDB:

```
FOR restaurante IN restaurantes
LET     p     =     (FOR     especialidade     IN
restaurantes.especialidades
FILTER LIKE(especialidade.prato, 'sushi', true) )
RETURN MERGE(
   restaurante,
   {especialidade: p}
)


LET from = (FOR u IN restaurantes FILTER u.nome ==
'Sushi    Express'  RETURN  u._id)  FOR  p  IN  FILTER
p.status = 'APROVOU'      TRAVERSAL(pessoas,
pessoasrelated, from[0], 'outbound',     {   minDepth:
0,   paths: true }) RETURN { vertices: p.path }


LET from = (FOR u IN pessoas FILTER u.nome == 'João'
RETURN    u._id) FOR p IN FILTER p.status = 'SEGUINDO'
TRAVERSAL(pessoas,   pessoasrelated,      from[0],
'outbound', {  minDepth: 0, maxDepth:    3,     paths:
true }) RETURN { vertices: p.path }
```

4.6 Conclusão do capítulo

Neste capítulo foram apresentadas as aplicações com persistência poliglota e os gerenciadores NoSQL multi-modelos como suporte para este tipo de aplicação. Uma questão levantada é qual o impacto no desempenho das aplicações implementadas usando estes gerenciadores quando comparado com a implementações utilizando gerenciadores de modelo único. No próximo capítulo será apresentada uma ferramenta de testes (benchmark) que permitirá comparar o desempenho das duas soluções simulando uma aplicação com persistência poliglota que é o objetivo fundamental desta dissertação

Capítulo 5. Benchmark para persistência poliglota

Este capítulo apresenta a proposta desta dissertação que corresponde à avaliação do desempenho de gerenciadores de bancos de dados multi-modelos em aplicações com persistência poliglota. São apresentados os objetivos fundamentais da dissertação e é introduzida a ferramenta implementada para realizar os testes de desempenho. Estes testes avaliam as principais operações de bancos de dados através de um programa que simula uma aplicação com persistência poliglota e que pode utilizar de forma flexível diferentes sistemas de gerenciamento de bancos de dados.

5.1 Objetivos

Como apresentado no capítulo 4, a introdução dos gerenciadores de bancos de dados NoSQL multi-modelo tem impactado positivamente a gestão e desenvolvimento das aplicações poliglotas. No entanto, não existem evidências de que estes gerenciadores impactam também o aumento do desempenho deste tipo de aplicações. Para contribuir a dar resposta a esta questão, esta dissertação propõe simular as operações de uma aplicação poliglota e comparar seu desempenho com e sem o uso de gerenciadores de bancos de dados multi-modelos. Nesta direção, os objetivos desta dissertação podem ser resumidos em três aspectos:

- Pesquisar os gerenciadores de bancos de dados NoSQL multi-modelos com foco em aplicações com persistência poliglota.
- Avaliar metodologias e ferramentas para testes de desempenho em aplicações com persistência poliglota.
- Comparar o desempenho de aplicações poliglotas implementadas com gerenciadores NoSQL multi-modelos e com gerenciadores baseados em modelos NoSQL tradicionais.

Para implementar uma aplicação com persistência poliglota sem o uso de gerenciadores de bancos de dados multi-modelo é necessário o acesso simultâneo a mais de um gerenciador de bancos NoSQL baseados em modelos diferentes. Nesta aplicação os dados tem componentes representados e armazenados segundo diferentes modelos de dados. Este fato exige a necessidade de

administrar vários gerenciadores ao mesmo tempo e implementar transações que realizam operações diferentes em cada gerenciador e depois integrar os resultados. Mesmo com uma maior complexidade para seu desenvolvimento, o desempenho desta implementação pode ser melhor que uma implementação utilizando um gerenciador de banco de dados multi-modelo.

Para simular uma aplicação poliglota sem o uso de gerenciadores de bancos de dados multi-modelo é necessária uma metodologia de testes e uma ferramenta que permita a interação simultânea com vários gerenciadores baseados em diferentes modelos de dados.

Como parte da pesquisa, foram analisados diferentes testes e métricas de desempenho em bancos de dados NoSQL na literatura assim como ferramentas de *benchmark* para estes testes. Os resultados fundamentais desta análise são apresentados a seguir.

5.2 Trabalhos correlatos

Existem vários trabalhos na literatura que avaliam o desempenho de gerenciadores baseados em modelos NoSQL. Em (Jouili e Vansteenberghe, 2013) os autores avaliam vários gerenciadores de bancos de dados NoSQL orientados a grafos, simulando operações com cargas de trabalhos diferentes, e tamanho de grafos diferentes, utilizando a interface de acesso *TinkerPop*, que é uma iniciativa de padronização no acesso a SGBDs orientados a grafos. Esta metodologia não é adequada diretamente para nossa pesquisa porque unicamente considera bancos de grafos, no entanto, este trabalho define métricas de avaliação de SGBDs orientados a grafos que utilizamos na nossa proposta de avaliação.

Em (Kolomicenko, 2013) o autor também avalia vários gerenciadores de bancos de dados NoSQL orientados a grafos, através da interface *TinkerPop*, utilizando grafos sintéticos. Como parte deste projeto foi desenvolvida uma ferramenta para a avaliação de desempenho denominada *BlueBench*. O foco deste trabalho é verificar apenas o desempenho em SGBDs orientados a grafos e portanto, a ferramenta proposta também não era adequada para os objetivos de nosso projeto. Este trabalho, no entanto, serviu como base para o desenvolvimento da ferramenta proposta nesta dissertação.

Em (Henricsson, 2011), são avaliados gerenciadores de bancos de dados orientados a documentos, simulando operações de leituras e escritas com cargas de trabalho diferentes, utilizando bibliotecas de acesso para a linguagem *Python*. Esta pesquisa teve como foco os SGBDs orientados a documentos e serviu como base para a definição das métricas de desempenho de um SGBD orientado a documentos que foram utilizadas em nosso projeto. Em (Narde, 2013), vários gerenciadores de bancos de dados NoSQL orientados a documentos são avaliados com foco em determinar em qual estrutura de Nuvens (*Cloud*) estes sistemas apresentam melhor desempenho. Neste trabalho foi utilizada a ferramenta Yahoo *Cloud Serving Benchmark* para avaliar o desempenho. Esta abordagem e ferramenta também não são adequados para nosso projeto porque não permite trabalhar com vários SGBDs ao mesmo tempo. Na pesquisa bibliográfica realizada apenas sistemas de gerenciamento do mesmo modelo foram avaliados. Não foram encontrados na literatura trabalhos de avaliação de desempenho envolvendo sistemas de gerenciamento multi-modelos.

Ferramentas para testes de desempenho de SGBD NoSQL

No ambiente computacional, um *benchmark* é um software que realiza um conjunto restrito e pré-definido de operações, e retorna um resultado em algum formato, definido como métricas. Estas métricas geralmente medem a velocidade da carga completada, ou a vazão, que é o número de cargas de trabalho por unidade de tempo que foram medidas. (Gray, 1993)

Ao executar o *benchmark* em múltiplos gerenciadores de bancos de dados, com diferentes *datasets*, é possível realizar comparações entre as métricas agrupando por operações.

Foram analisados dois softwares de *benchmark* para verificar se os mesmos atendiam os requisitos para avaliar o desempenho de aplicações poliglotas.

5.2.1 GraphDB-Bench

O GraphDB-Bench (Broecheler, 2013) é uma ferramenta de *benchmark* que compara o desempenho de bancos orientados a grafos. Ela é baseada na tecnologia *TinkerPop* utilizando o padrão *BluePrint* e linguagem *Gremlin*. Ela fornece uma interface para operações de testes definidos pelo usuário, mede o

tempo de execução e registra todos os resultados. O GraphDB-Bench também contém scripts para geração automática de gráficos sintéticos. Caso seja necessário carregar um *dataset* de grafos pronto, é possível importar grafos no padrão GraphML.

O projeto foi iniciado no ano de 2010 e a intenção original era tornar-se uma ferramenta de *benchmarking* extensível de forma tal que todos os resultados de *benchmark* seriam mantidos em um repositório público para referência. A ferramenta realiza testes de leitura e escrita em grafos, simulando também o percorrimento do grafo em várias profundidades. No entanto, o GraphDBBench não foi mais expandido desde 2011, tem sido dotado apenas pelos idealizadores para testes privados.

Esta ferramenta não permite testar aplicações de persistência poliglota porque apenas um sistema de gerenciamento e no modelo baseado em grafos pode ser testado de cada vez.

5.2.2 Yahoo! Cloud Serving Benchmark (YCSB)

O Yahoo! *Cloud Serving Benchmark* (YCSB) foi desenvolvido pela empresa Yahoo com o propósito de testar vários gerenciadores de bancos de dados NoSQL. É um projeto *open source* e extensível.

Ele trabalha com cargas de trabalho que podem ser carregados e executados utilizando um gerenciador de banco de dados. Cada carga de trabalho é uma mescla de operações de leitura e escrita baseadas em dados de tamanhos diferentes. (Abubakar, et al., 2014).

Nesta ferramenta é possível definir o número de *threads* que serão executadas com o objetivo de aumentar a carga computacional de um sistema de gerenciamento para medir o tempo de resposta das requisições.

Nesta ferramenta é possível adicionar novos SGBDs implementando uma interface específica com métodos de leitura, busca, inserção e alteração. No entanto, a ferramenta não é adequada para comparação de aplicações de persistência poliglota porque ela só permite testar um gerenciador de cada vez.

5.3 Ferramenta para avaliação do desempenho de aplicações poliglotas

Como foi analisado anteriormente, as ferramentas e metodologias encontradas na literatura permitem apenas avaliar o desempenho de diferentes sistemas de gerenciamento mas não permitem simular uma aplicação de persistência poliglota, que exige o acesso simultâneo a mais de um sistema de gerenciamento.

Por esta razão, foi necessário projetar e implementar uma ferramenta de análise de desempenho, que possibilite a simulação de uma aplicação com persistência poliglota. Essa nova ferramenta permite a distribuição de requisições para sistemas de gerenciamento de bancos de dados NoSQL baseados em modelos de dados diferentes.

A ferramenta desenvolvida foi projetada com os seguintes recursos e premissas:

- *Requisições de operações para os gerenciadores de bancos de dados através da interface REST:* A ferramenta deve acessar os gerenciadores NoSQL através da interface REST, padronizando os testes em todos os gerenciadores através desta interface.
- *Carga de trabalho configurável:* a ferramenta deve permitir flexibilidade nos testes e para isso a descrição destes testes e a carga de trabalho são definidos em arquivos em formato XML..
- *Geração de datasets sintéticos:* a ferramenta deve gerar *datasets* sintéticos do modelo de documentos JSON, criando atributos com valores aleatórios.
- *Geração de logs:* a ferramenta deve salvar em arquivos de log todas as requisições para análise.

5.3.1 Arquitetura da ferramenta

A ferramenta de testes foi desenvolvida como uma aplicação Java que produz as requisições para os SGBDs executarem.

Como apresentado na Figura 12, a aplicação de análise de desempenho possui um núcleo básico onde são disparadas as requisições que implementam os testes de carga (*workloads*).

Como definido nos requisitos, a aplicação é parametrizável através de arquivos de configuração XML que descrevem as requisições a ser disparadas para os SGBDs NoSQL que estão sendo executados em outros servidores.

A ferramenta foi idealizada com o objetivo de ser flexível e suportar qualquer gerenciador NoSQL que suporte a interface REST.

Os testes de cargas são organizados como trabalhos (*jobs*) que por sua vez possuem até duas tarefas (*tasks*).

Cada trabalho corresponde a uma requisição dentro da aplicação poliglota, isto é, uma requisição que utiliza dois modelos de dados diferentes. Um teste de carga pode executar várias *threads,* cada uma delas associada a um trabalho. Esta quantidade de threads é definida no arquivo de configuração do teste.

Por exemplo, no caso de uma requisição de recuperação de dados nos gerenciadores de modelo único, a primeira tarefa é uma consulta no modelo de grafos para recuperar os identificadores dos vértices de um grafo conectados em um determinado nível de profundidade. Na segunda tarefa são utilizados os identificadores dos vértices recuperados na primeira *tarefa* para consultas no modelo de documentos, recuperando neste caso todos os documentos com esses identificadores. Quando são testados SGBDs multi-modelos, esta mesma requisição é definida como um trabalho (*Job)* que possui apenas uma tarefa (*Task).*

Nesse tipo de gerenciador *a* tarefa realiza uma única consulta que percorre todos os vértices conectados e simultaneamente recupera os documentos destes vértices.

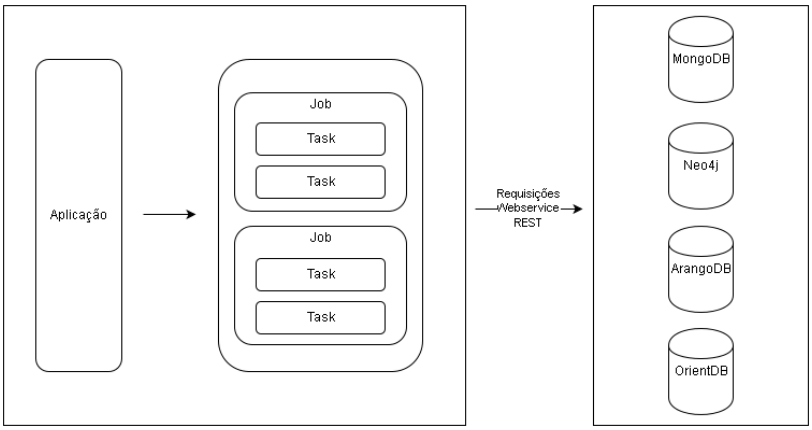

Figura 12: Arquitetura da aplicação desenvolvida

A Figura 13 apresenta um exemplo de configuração do arquivo XML da ferramenta desenvolvida.

O primeiro elemento denominado (*jobmanager)* contem a descrição do teste e o número de *Threads* que corresponde ao número de requisições concorrentes que o programa pode solicitar no servidor. Dentro do elemento (*jobmanager)* constam elementos (*job)* associados aos trabalhos que contém por sua vez os elementos (*task)* associados às tarefas. No caso dos gerenciadores multi-modelo, um (*job)* pode conter apenas um elemento (*task)*, porque em uma única consulta de persistência poliglota é percorrido o grafo e recuperados os documentos associados aos vértices percorridos.

No exemplo da Figura 13 há um (*job)* com duas (*task)*. A primeira (*task)* descreve um comando para ser executado no gerenciador Neo4j um comando para percorrer o grafo partindo do vértice com identificador com conteúdo “47000”. O gerenciador Neo4j retorna os vértices que estão ligados neste vértice inicial retornando os identificadores.

Para executar este comando *MATCH* do Neo4j, no elemento DATA é informado o comando a ser executado, e no elemento URL o caminho do webservice REST que vai atender a requisição.

Na segunda (*task)*, há um elemento denominado dependsOnFilter que processa o conteúdo retornado pela primeira (*task)*, preparando os dados para enviar para o segundo gerenciador, que no caso deste exemplo é o MongoDB.

Neste elemento dependsOnFilter é possível selecionar e filtrar os elementos JSON que foram retornados na primeira (*task)*, enviando os dados já filtrados para a segunda (*task)*.

Foram criados comandos próprios que configuram como é o processamento realizado sobre os resultados da primeira (task) para alimentar a segunda (task). Estes comandos são:

- `JSONPATH:` Através deste comando é possível selecionar e filtrar partes dos elementos JSON recebidos pelo processamento da primeira (task). Este comando é definido em uma linguagem similar a XPATH para XML.

- `REMOVE:` Remove caracteres ou strings do resultado.
- `EXTBETWORDS:` Seleciona dados entre duas palavras a partir do resultado.
- `TOKENIZE:` Divide os resultados utilizando um carácter específico.

No exemplo da Figura 13 o elemento dependsOnFilter remove *strings* e caracteres retornados pelo gerenciador Neo4j. Após a remoção de *strings,* são separados os vértices pelo delimitador vírgula para enviar para o gerenciador MongoDB, que receberá os vértices na variável [CONTENT].

São criadas várias requisições para o gerenciador MongoDB, uma para cada vértice recuperada do gerenciador Neo4j na primeira (*task)*.

Cada requisição vai recuperar um documento através do identificador. No arquivo XML completo de um teste, há muitos elementos (*job)* recuperando outros vértices, diferentemente do exemplo abaixo que apenas é recuperado o vértice com identificador “47000”.

```
<jobmanager>
  <title>Recupera do Neo4j os vertices acessando os documentos pelo id no MongoDB</title>
  <numberofThreads> 5 </numberofThreads>
   <job>
      <task>
         <data>{"query" : "MATCH path=(s:Sintetico {no:'47000'})-[:RELATED*1..1]-&gt;(x:Sintetico) return
ID(x);"}</data>
         <url>http://localhost:7474/db/data/cypher</url>
      </task>
      <task>
         <data></data>
         <dependsOnFilter>REMOVE  "columns" : [ "ID(x)" ],;REMOVE [;REMOVE ];REMOVE
data;REMOVE ";REMOVE :;REMOVE {;REMOVE };TOKENIZE ,</dependsOnFilter>
         <url>http://localhost:28017/amazon/sintetico/?filter_no=[CONTENT]</url>
      </task>
   </job>
</jobmanager>
```

Figura 13: Exemplo de arquivo XML de configuração da ferramenta de benchmark

5.4 Conclusão do capítulo

Neste capítulo foi apresentado o objetivo principal da dissertação correspondente à avaliação do desempenho de aplicações poliglotas utilizando gerenciadores de bancos de dados multi-modelos. Foi introduzida a arquitetura e

funcionalidade da ferramenta que foi desenvolvida para realizar os testes de desempenho. No próximo capítulo serão apresentados os detalhes destes testes propostos e seus resultados.

Capítulo 6. Resultados dos experimentos

Este capítulo apresenta os resultados experimentais obtidos para comparar o desempenho de uma aplicação poliglota com e sem sistemas de bancos de dados multi-modelos. Para obter estes resultados foi utilizada a ferramenta de testes apresentada no capítulo 5.

6.1 Metodologia dos testes de desempenho

Como apresentado no capítulo 5 esta dissertação tem como objetivo simular as operações de uma aplicação poliglota e comparar seu desempenho com e sem o uso de gerenciadores de bancos de dados multi-modelos. Neste trabalho consideramos uma aplicação poliglota que utiliza os modelos de bancos de dados orientados a documentos e de grafos. Outras combinações de modelos NoSQL poderiam ser pesquisadas mas está fora do escopo desta dissertação.

Neste trabalho propomos a criação de um *benchmark* para avaliar o desempenho baseado nos seguintes testes de carga em aplicações poliglotas:

- **Testes de armazenamento de grafos e documentos:**

  Estes testes simulam a criação ou carga de um banco de dados típico de uma aplicação poliglota. Nestes testes o banco de dados representa um conjunto de documentos relacionados através de um grafo.

  No caso do teste com gerenciadores distintos para cada modelo de dados o banco de dados é criado em duas etapas. Inicialmente são inseridos todos os documentos, em seguida são recuperados os identificadores destes documentos e é criado um grafo com os identificadores como vértices e o relacionamento entre os documentos como arestas.

  No caso do teste do gerenciador multi-modelo o banco é criado também em duas etapas. Na primeira etapa são inseridos os documentos sendo que cada um deles já é o vértice do grafo. Na segunda etapa são inseridos as arestas do grafo que relaciona os documentos/vértices.

- **Testes de recuperação de documentos**

  Estes testes não simulam propriamente uma aplicação poliglota. Eles avaliam o desempenho dos gerenciadores para recuperação de dados utilizando exclusivamente o modelo de documentos. Neste teste são recuperados os documentos a partir do uma lista de identificadores. Neste teste, a carga sobre os gerenciadores é aumentada progressivamente duplicando o número de clientes realizando requisições simultaneamente. São utilizados até 32 clientes independentes, cada um deles executando em uma máquina virtual diferente.

- **Testes de recuperação de documentos com percorrimento do grafo:**

  Estes testes simulam operações de consulta ou recuperação em uma aplicação poliglota. Nestes testes são recuperados todos documentos relacionados em vários níveis de profundidade com os documentos associados a uma lista de identificadores. Os níveis de profundidade são aumentados desde o valor 1 até o valor 6. Para cada nível de profundidade a carga sobre os gerenciadores é aumentada progressivamente duplicando o número de clientes realizando requisições simultaneamente. São utilizados até 32 clientes independentes, cada um deles executando em uma máquina virtual diferente.

  No caso do teste com gerenciadores distintos para cada modelo de dados, inicialmente são recuperados no grafo todos os identificadores relacionados com o identificador de consulta, percorrendo em seguida as arestas segundo o nível de profundidade definido. Posteriormente, no gerenciador de documentos são recuperados todos os documentos associados aos identificadores recuperados no grafo.

  No caso do teste do gerenciador multi-modelo, é executada uma única requisição que recupera todos os documentos associados aos vértices percorridos segundo o nível de profundidade definido.

Os experimentos avaliam as principais operações do modelo de documentos e grafos, nas implementações nativas e nas implementações multi modelo.

Como parte destes testes serão medidos outros parâmetros como o consumo de cpu, memória, disco e rede.

6.2 Gerenciadores de bancos de dados selecionados e ambiente para os testes

Os sistemas de gerenciamento de bancos NoSQL multi-modelo que foram selecionados são o ArangoDB e OrientDB. Ambos suportam os modelos de dados baseados em documento e grafo.

Como sistemas de gerenciamento baseados unicamente nos modelos de documentos e grafos foram selecionados os MongoDB e Neo4j, os quais são apresentados a seguir:

- **SGBD orientado a documentos MongoDB**

O MongoDB é um banco de dados orientado a documentos, criado em 2009, que trabalha sem esquemas. Foi escrito na linguagem de programação C++. Ele suporta particionamento (*sharding*), uma poderosa linguagem de consulta, documentos aninhados e referências. Ele não suporta as propriedades ACID completas, apenas suporta transações sobre um documento, não oferecendo transações que manipulam simultaneamente um conjunto de documentos.

- **SGBD orientado a grafo Neo4j.**

O Neo4j é um banco orientado a grafos *open source* criado em 2007. Ele oferece uma poderosa linguagem de consulta chamada Cypher, trabalha com REST API, não suporta particionamento (*sharding*) e não possui nenhum gerenciamento de permissões nos dados.

O Neo4j suporta o TinkerPop Blueprints permitindo usar a linguagem Gremlin para manipular grafos. Ele suporta todas as propriedades ACID.

6.2.1 Ambiente de execução dos experimentos

Todos os testes foram executados em um ambiente ilustrado na Figura 14. Os gerenciadores NoSQL foram executados em um servidor com processador AMD Opteron com 8 núcleos de 2.3ghz, 32Gb de RAM, duas HD's de 2Tb magnéticas SATA configuradas em Raid 1 por *hardware* com controladora PERC. A

vantagem de operar com Raid 1 é que os dados estão espelhados proporcionando redundância, além de oferecer o dobro de velocidade em operações de leitura porque a carga é dividida entre os dois discos, tendo a mesma velocidade de gravação se comparado com um único disco (Dell, 2017). A aplicação de teste desenvolvida que simula a persistência poliglota foi executada em até 32 máquinas virtuais com 1 vCPU com 4Gb de RAM.

Em alguns dos testes realizados a aplicação de teste foi executada como cliente inicialmente em apenas uma máquina virtual e, em seguida, foram sendo adicionadas progressivamente mais máquinas virtuais dobrando a capacidade, até chegar a 32 máquinas virtuais. O aumento foi na seguinte ordem: 1, 2, 4, 8, 16 até 32 máquinas virtuais. A decisão de trabalhar com 32 máquinas virtuais teve como objetivo ir aumentando o número de requisições simultâneas ao servidor a partir de vários clientes, o que implica em um aumento progressivo da carga de trabalho (teste de estresse). Para sincronizar as requisições entre as 32 máquinas virtuais, o serviço Cron foi usado para agendar o trabalho em uma data e hora específica, e via NTP os relógios foram sincronizados. As máquinas virtuais foram distribuídas em diferentes servidores, evitando ser hospedadas pelo mesmo servidor físico (requisito não funcional informado ao *datacenter*). Para garantir a velocidade de comunicação e isolamento de tráfego, os testes ocorreram em uma conexão *Gigabit* de rede com VLAN entre as 32 máquinas virtuais e o servidor.

Durante a execução dos testes todos os serviços foram desabilitados, deixando apenas em execução a aplicação de teste nos clientes e os gerenciadores de banco de dados testados no servidor.

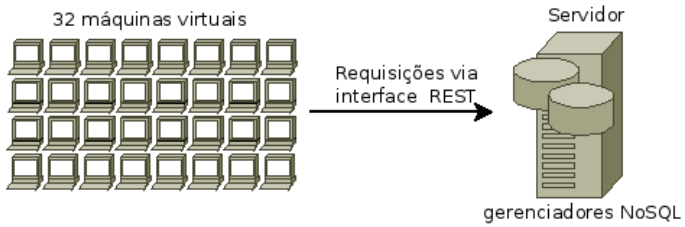

Figura 14: Descrição do ambiente de testes. De 1 até 32 clientes associados a máquinas virtuais realizam requisições nos servidores NoSQL

6.2.2 Métricas de análise

O objetivo da pesquisa é verificar se os gerenciadores de bancos de dados multi-modelo atingem um desempenho satisfatório quando comparados com gerenciadores de modelo único.

Os testes antes descritos foram configurados através de um arquivo XML de que descrevem o processamento do benchmark em cada uma das máquinas virtuais. No arquivo XML constam todas as consultas, URL's REST de acesso aos gerenciadores e parâmetros que serão enviados para as consultas.

Foi medido o tempo total de processamento entre as requisições das máquinas virtuais enviadas para o servidor, ou seja, quanto tempo o servidor demora para processar todas as requisições. O tempo decorrido para executar cada consulta no banco de dados foi coletado diretamente no servidor, sendo em seguida enviado como resposta ao cliente através de ferramentas de *profiler* nativas de cada gerenciador de banco de dados. O gerenciador Neo4j na versão testada não oferece esta funcionalidade de enviar o tempo de processamento da consulta de volta para o cliente, portanto o tempo calculado no Neo4j foi a diferença entre o início e fim da requisição, portanto é um tempo calculado diretamente no cliente. Neste caso do Neo4j, há uma penalidade adicionada ao tempo, porque é contabilizado o tempo da velocidade da rede entre os clientes e o servidor. Para estimar esse tempo médio de penalização foram comparados os tempos calculados pelos *profilers* dos gerenciadores ArangoDB, OrientDB e MongoDB, e os tempos calculados diretamente nos clientes entre o início e término da requisição, sendo que sua diferença foi em média 0.001 segundos (1 milissegundo).

No caso dos testes com uma única máquina virtual disparando as requisições para o servidor, o tempo total contabilizado foi a soma dos tempos de processamento de cada consulta enviada ao servidor.

No caso dos testes com várias máquinas virtuais, o total de consultas foi dividido igualmente entre as máquinas que iniciam suas requisições para o servidor de forma sincronizada.

Neste caso, o maior tempo de processamento dentre as máquinas virtuais é considerado como tempo total de processamento. Esse critério foi estabelecido

porque a interface utilizada para o teste é a webservice REST, portanto é importante medir o desempenho dos gerenciadores de bancos de dados ao responder consultas para múltiplos clientes, conforme ocorre numa aplicação web.

O teste foca na simulação de uma aplicação web.

- **Explicação de um teste exemplo e interpretação de seu desempenho:**

Vamos supor que uma máquina virtual requisite 10.000 consultas para o servidor. O tempo total contabilizado é a soma do tempo de processamento de todas as 10.000 requisições individuais. Para processar as mesmas 10.000 consultas com 2 máquinas virtuais, cada máquina virtual requisita 5.000 consultas para o servidor, e o tempo contabilizado é o da máquina virtual que demorou mais para ter o retorno das 5.000 consultas enviadas, porque o processamento no servidor é paralelo. Com esta abordagem é possível verificar a eficiência do servidor ao atender consultas ao mesmo tempo, simulando assim uma aplicação Web com muitas requisições REST paralelas. O tempo necessário para processar as 10.000 requisições vindas através de apenas uma máquina virtual é provavelmente maior porque o servidor está atendendo uma requisição por vez, com pouca utilização de sua capacidade de CPU. Quando duas máquinas virtuais estão enviando as requisições ao mesmo tempo, o servidor utiliza mais sua capacidade de processamento (CPU) para atender estas requisições de forma paralela e trabalhando com mais núcleos do processador, portanto processa as 10.000 requisições divididas entre duas máquinas virtuais com 5.000 requisições cada, num tempo bem menor.

- **Ferramenta dstat para coletar métricas:**

Através da ferramenta dstat (Wieers, 2005) disponível no sistema operacional Linux, foram coletadas métricas de consumo de cpu, disco, memória e rede no servidor para verificar o consumo de cada gerenciador de banco de dados ao atender as requisições. As métricas apresentadas nas tabelas dos testes correspondem ao momento do maior consumo (pico) de CPU ao executar um teste. A ferramenta foi configurada para salvar em arquivos csv as métricas coletando os dados a cada 1 segundo.

a) Informações coletadas pela ferramenta dstat:

As informações coletadas pela ferramenta em cada teste serão apresentadas como uma tabela com o seguinte formato:

```
----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total-
usr sys idl wai hiq siq| read  writ| used  buff  cach  free| recv  send
```

Em seguida são explicadas as métricas coletadas pela ferramenta.

Relacionadas com o uso de processador (CPU):

- Usr: Tempo gasto executando processos de usuários
- Sys: Tempo gasto executando processos de sistema
- Idl: Tempo disponível, sem processamento.
- Wai: Tempo gasto esperando recursos de I/O
- Hiq: Tempo gasto atendendo interrupções de hardware
- Siq: Tempo gasto atendendo interrupções de software

Relacionadas com o uso de disco externo:

- Read: Quantidade de bytes lidos no disco
- Writ: Quantidade de bytes escritos no disco

Relacionadas com o uso de memória interna:

- Used: Quantidade de memória utilizada
- Buff: Quantidade de memória utilizada para buffers de escrita do disco
- Cach: Quantidade de memória utilizada para cache de leituras do disco
- Free: Quantidade de memória não utilizada

Relacionadas com o uso de rede:

- Recv: Quantidade de bytes recebidos pela rede
- Send: Quantidade de bytes enviados pela rede

a) Exemplo de dados coletadas pela ferramenta

No exemplo abaixo constatamos que a carga de CPU está em 21%, que a leitura do disco é de 132Kbytes e escrita 33Megabytes, que a memória total ocupada é 30.66Gigabytes e que o consumo de rede é 1840bytes recebidos e 506bytes enviados.

A ferramenta permite especificar a quantidade de segundos de coleta, exibindo uma linha como esta abaixo a cada intervalo de tempo definido.

```
----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total-
usr sys idl wai hiq siq| read  writ| used  buff  cach  free| recv  send
 21  75   0   4   0   0| 132k   33M|30.6G 2248k  257M  193M|1840B  506B
```

As métricas mais importantes para nossa análise são: o consumo de CPU na coluna USR e a leitura e escrita do disco nas colunas READ e WRIT.

6.3 Bancos de testes (d*atasets*) utilizados

Para a realização dos testes de carga da aplicação poliglota foram utilizados três bancos de dados de teste. O primeiro banco de testes consistiu de dados reais de vendas fornecidos pela empresa *Amazon*, com características próprias de uma aplicação de persistência poliglota baseada em grafos e documentos. Os outros bancos de testes utilizados foram gerados de forma sintética simulando uma aplicação de persistência poliglota. Estes bancos de testes são descritos a seguir

- **Banco de teste de vendas da empresa Amazon**

A universidade de Stanford disponibiliza um *dataset* de vendas da empresa Amazon, que contem os produtos desta companhia descritos por diferentes atributos e adicionalmente o relacionamento entre esses produtos. (Stanford University, 2006)

Este banco de testes oferece um cadastro completo dos produtos da Amazon, suas categorias, produtos relacionados, compras e *reviews* com uma qualificação dos compradores. Esta informação pode ser representada utilizando documentos para os produtos e um grafo para o relacionamento entre os produtos.

- **Bancos de testes sintéticos**

Como parte da aplicação de teste foi implementado um gerador de dados sintético a fim de criar bancos de testes híbridos, ou seja, *datasets* que contenham simultaneamente uma estrutura de documentos e de grafos. Para gerar a estrutura de grafos sintéticos a aplicação utiliza o algoritmo Barabási-Albert (Barabási e Albert, 1999) com a biblioteca GraphStream (GraphStream, 2016). O algoritmo Barabási-Albert gera grafos sintéticos inicialmente com poucos vértices aleatórios, e posteriormente novos vértices são adicionados escolhendo como vértice pai os vértices aleatórios com mais arestas. No caso dos documentos, a aplicação gera documentos JSON aleatórios, com um nome de chave sequencial e valores aleatórios para essas chaves. O documento final tem apenas um nível hierárquico, com pares de chaves para facilitar o acesso de todas as linguagens de consulta. Foram gerados dois conjuntos de dados sintéticos com tamanhos diferentes.

A Tabela 3 descreve as dimensões de cada um dos três bancos de testes.

Tabela 3. Dimensões dos datasets

Tipo do dataset	Número de documentos / vértices	Número de campos em cada documento	Número de arestas
Amazon	548.552	8	1.788.725
Sintético Pequeno	100.000	10	547.993
Sintético Médio	500.000	50	6.500.237

A seguir são apresentados os resultados dos testes realizados.

6.4 Testes de armazenamento de documentos e grafos

Neste teste de *benchmark* o objetivo é avaliar o desempenho na criação de um banco de dados típico de uma aplicação poliglota, isto é, contendo documentos relacionados através de um grafo.

6.4.1 Teste de armazenamento de documentos e grafos no *dataset* Amazon

Este teste do *benchmark*, consiste do armazenamento do dataset da Amazon. Conforme a Figura 15 é possível verificar que o SGBD ArangoDB apresenta o melhor desempenho. Consideramos que este resultado se deve à estrutura de armazenamento de grafos do ArangoDB trabalhar com documentos para seu *engine* de grafos, além de trabalhar apenas com índices em memória.

No caso de Neo4j, na tentativa de melhorar os tempos de inserção de dados foi analisado o tutorial de desempenho em (Needham, 2014). Este tutorial sugere algumas técnicas como otimização de *buffers* de memória, criação de índices necessários e utilização da ferramenta de *Profiler,* mas não foi possível melhorar o tempo. Em (Mahajan, 2014) e (Pickhardt, 2012) sugere-se realizar as operações de inserção através da API diretamente, já que é constatado que o uso da linguagem Cypher para inserção provoca menor desempenho.

A Tabela 4 descreve o consumo de recursos computacionais durante o teste. As linhas 1 e 3 demonstram que não há muito consumo de CPU e disco nos gerenciadores ArangoDB e MongoDB. Já nas linhas 2 e 4, a quantidade de operações de I/O de disco do Neo4j e OrientDB foi superior ao ArangoDB na carga do grafo. Consideramos que a causa desta diferença está principalmente no custo de armazenar a estrutura “Index Free Adjancency”.

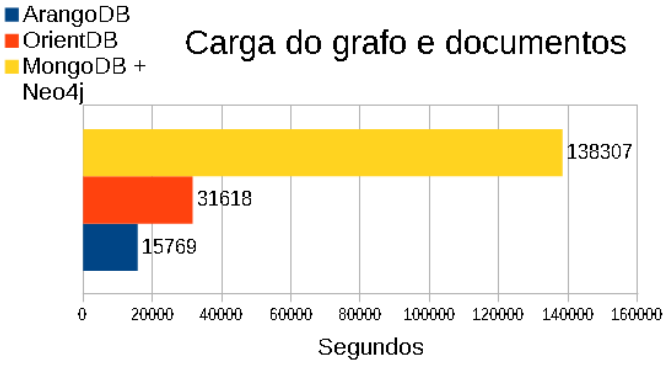

Figura 15: Resultados dos testes de armazenamento do dataset Amazon

Tabela 4. Consumo de recursos na carga de grafo e documentos com *dataset* Amazon

	SGBD	Vms	Recursos consumidos
1	ArangoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 21 75 0 4 0 0| 132k 33M|30.6G 2248k 257M 193M|1840B 506B
2	OrientDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 69 12 0 19 0 0|1816k 78M|30.7G 2248k 231M 193M|1716B 362B
3	MongoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 6 94 0 0 0 0|4096B 817k|24.1G 5836k 391M 6737M|1532B 362B
4	Neo4j	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 74 4 0 22 0 0|1564k 105M|30.7G 940k 181M 193M|1822B 416B

6.4.2 Teste de armazenamento de documentos e grafos com os *datasets* sintéticos

Neste teste de *benchmark* foram armazenados os *datasets* sintéticos pequenos e médios. Observamos nas Figuras 16 e 17 que novamente o ArangoDB apresentou melhor desempenho que os demais gerenciadores.

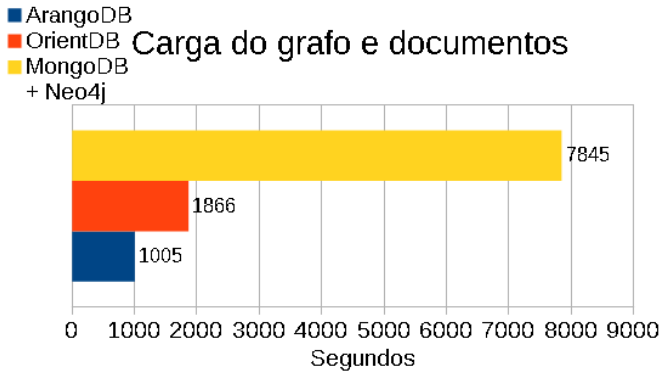

Figura 16. Resultados dos testes de armazenamento do dataset sintético pequeno

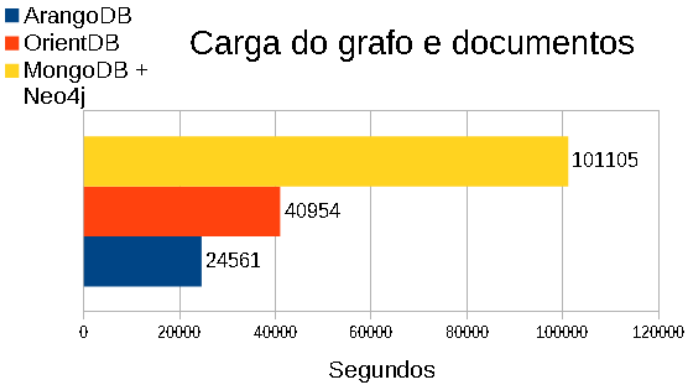

Figura 17. Resultados dos testes de armazenamento do dataset sintético médio

De modo geral os resultados mostram que ArangoDB apresenta o melhor desempenho na carga de grafos. Consideramos que este fato se deve à sua

implementação de estruturas de grafos baseada em documentos, na qual para ligar dois vértices apenas uma entrada é criada no documento que representa as arestas, além de sua estrutura denominada Shape que armazena conjuntamente estruturas iguais. Consideramos que os gerenciadores que trabalham com a estrutura "I*ndex Free Adjacency*" tem pior desempenho no armazenamento. O fato de que cada vértice tenha ponteiros para os outros vértices, provoca que a inserção de uma aresta resulte em maior processamento, maior utilização de espaço e mais operações I/O de disco para criar essa estrutura. É possível visualizar esse consumo maior de recursos devido a esta estrutura de armazenamento nas linhas 2 e 4 das tabelas 5 e 6.

Tabela 5. Consumo de recursos na carga de grafo e documentos sintético pequeno

	SGBD	Vms	Recursos consumidos
1	ArangoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 32 9 59 0 0 0| 132k 33M|30.6G 2248k 257M 193M|1840B 506B
2	OrientDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 67 1 32 0 0 0|1816k 78M|30.7G 2248k 231M 193M|1716B 362B
3	MongoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 22 5 73 0 0 0|4096B 817k|24.1G 5836k 391M 6737M|1532B 362B
4	Neo4j	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 76 11 13 0 0 0|1564k 105M|30.7G 940k 181M 193M|1822B 416B

Tabela 6. Consumo de recursos na carga de grafo e documentos sintético médio

	SGBD	Vms	Recursos consumidos
1	ArangoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 57 13 30 0 0 0| 132k 33M|30.6G 2248k 257M 193M|1840B 506B
2	OrientDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 79 13 8 0 0 0|1816k 78M|30.7G 2248k 231M 193M|1716B 362B
3	MongoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 32 24 44 0 0 0|4096B 817k|24.1G 5836k 391M 6737M|1532B 362B
4	Neo4j	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 87 13 0 0 0 0|1564k 105M|30.7G 940k 181M 193M|1822B 416B

6.5 Teste de recuperação de documentos.

Neste teste são avaliados os gerenciadores de bancos de dados que suportam o modelo de documentos.

6.5.1 Teste de recuperação de documentos no *dataset* Amazon

Neste teste de *benchmark*, foram realizadas 10.000 requisições para recuperar documentos cujos identificadores foram gerados aleatoriamente. A quantidade de requisições aumentaram progressivamente iniciando de 1 até 32 máquinas virtuais. Os resultados na Figura 18 mostram que o ArangoDB apresenta o melhor desempenho para recuperação de documentos, seguido do MongoDB que apresentou um desempenho semelhante. O OrientDB apresentou o pior desempenho demonstrando também, na Tabela 7 nas linhas 2, 5 e 8, que necessita de mais CPU para processar as requisições. Quando a quantidade de máquinas virtuais atingiu 32, o consumo de CPU de todos os gerenciadores chegou próximo de 100%, consumindo todos os núcleos de processador do servidor, conforme as linhas 7,8 e 9 da Tabela 7. Na Figura 18 visualizamos que o tempo total de processamento foi melhorando conforme foram adicionadas mais máquinas virtuais. O motivo é que mais núcleos de processador do servidor foram atendendo mais requisições paralelas, mas quando é atingido o número de 16 máquinas virtuais, esse ganho cessou.

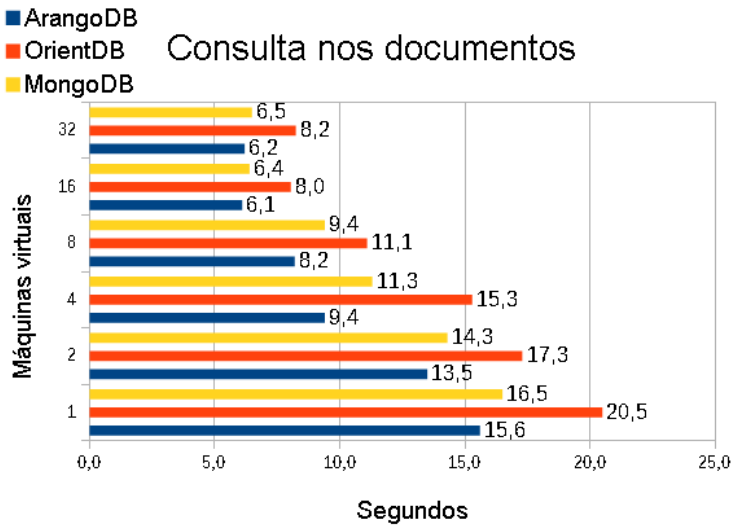

Figura 18: Resultados da recuperação de documentos do dataset Amazon.

Tabela 7. Consumo de recursos no teste de recuperação de documentos

	SGBD	Vms	Recursos consumidos
1	ArangoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 17 1 79 3 0 0|1056k 0 |30.6G 2248k 257M 193M|2458B 362B
2	OrientDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 26 1 70 3 0 0|2860k 9216B|30.7G 2248k 231M 193M|2122B 756B
3	MongoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 18 1 78 3 0 0|3196k 0 |24.1G 5836k 391M 6737M|2028B 212B
4	ArangoDB	8	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 59 7 30 4 0 0|3652k 51k|30.6G 2248k 257M 193M|1592B 272B
5	OrientDB	8	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 80 9 11 0 0 0|6140k 0 |30.7G 2248k 231M 193M|1846B 294B
6	MongoDB	8	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 57 7 36 0 0 0| 704k 223k|24.1G 5836k 391M 6737M|1592B 201B
7	ArangoDB	32	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 97 1 2 0 0 0|6284k 9216B|30.6G 2248k 257M 193M|1352B 272B
8	OrientDB	32	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 96 0 4 0 0 0|9444k 51k|30.7G 2248k 231M 193M|1426B 294B
9	MongoDB	32	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 93 1 6 0 0 0|2700k 9216B|24.1G 5836k 391M 6737M|1413B 201B

6.5.2 Teste de recuperação de documentos com os *datasets* sintéticos

Similarmente ao teste sobre o *dataset* Amazon, neste teste foram realizadas 10.000 requisições de documentos através das máquinas virtuais.

Os resultados para o *dataset* pequeno na Figura 19 mostra que ArangoDB tem um desempenho melhor do que MongoDB quando acessa documentos pequenos, inclusive consome menos CPU também conforme demonstrado nas linhas 1,4 e 7 da Tabela 8.

No caso do *dataset* médio, no qual os documentos tem maior quantidade de campos, os resultados na Figura 20 mostram que MongoDB tem melhor desempenho, superando o ArangoDB.

O OrientDB mantém o pior desempenho em ambos os *datasets* pequenos e médios. Nas linhas 2, 5 e 8 das tabelas 9 e 10 visualizamos que o OrientDB

consome mais CPU que os outros gerenciadores para trabalhar com documentos, portanto sua implementação de documentos sugere ser menos eficiente.

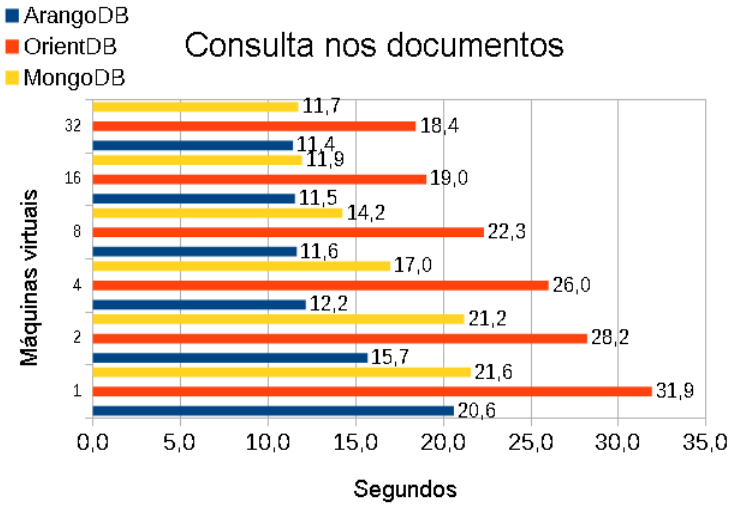

Figura 19. Consulta nos documentos do dataset sintético pequeno. Testes com 10000 consultas utilizando de 1 até 32 maquinas virtuais

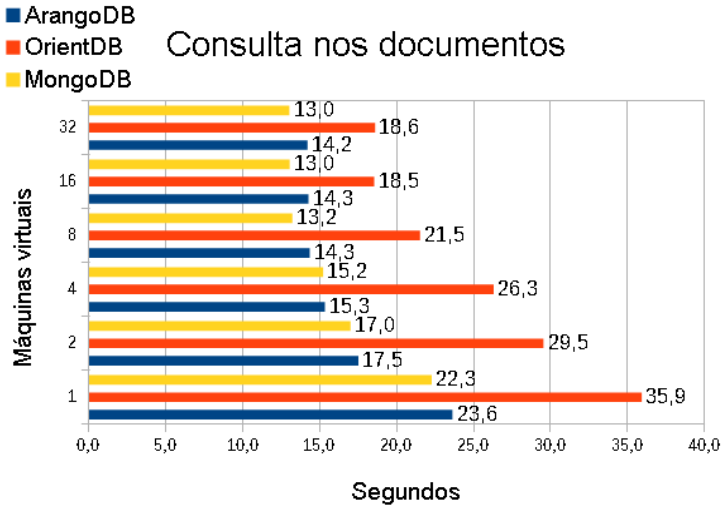

Figura 20. Consulta nos documentos do dataset sintético médio. Testes com 10000 consultas utilizando de 1 até 32 maquinas virtuais

Tabela 8. Consumo de recursos na consulta de documentos do dataset pequeno

	SGBD	Vms	Recursos consumidos
1	ArangoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 26 1 73 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2458B 362B
2	OrientDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 30 2 68 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|2122B 756B
3	MongoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 67 3 30 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|2028B 212B
4	ArangoDB	8	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 58 2 40 0 0 0|3652k 51k|30.6G 2248k 257M 193M|1592B 272B
5	OrientDB	8	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 97 3 0 0 0 0|6140k 0 |30.7G 2248k 231M 193M|1846B 294B
6	MongoDB	8	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 96 3 1 0 0 0| 704k 223k|24.1G 5836k 391M 6737M|1592B 201B
7	ArangoDB	32	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 91 2 7 0 0 0|6284k 9216B|30.6G 2248k 257M 193M|1352B 272B
8	OrientDB	32	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 98 1 1 0 0 0|9444k 51k|30.7G 2248k 231M 193M|1426B 294B
9	MongoDB	32	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 94 1 5 0 0 0|2700k 9216B|24.1G 5836k 391M 6737M|1413B 201B

Tabela 9. Consumo de recursos na consulta de documentos do dataset médio

	SGBD	Vms	Recursos consumidos
1	ArangoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 32 1 67 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2458B 362B
2	OrientDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 45 2 53 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|2122B 756B
3	MongoDB	1	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 48 1 51 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|2028B 212B
4	ArangoDB	8	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 88 1 11 0 0 0|3652k 51k|30.6G 2248k 257M 193M|1592B 272B
5	OrientDB	8	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 94 2 4 0 0 0|6140k 0 |30.7G 2248k 231M 193M|1846B 294B
6	MongoDB	8	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 96 1 3 0 0 0| 704k 223k|24.1G 5836k 391M 6737M|1592B 201B
7	ArangoDB	32	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 97 1 2 0 0 0|6284k 9216B|30.6G 2248k 257M 193M|1352B 272B
8	OrientDB	32	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 98 1 11 0 0 0|9444k 51k|30.7G 2248k 231M 193M|1426B 294B
9	MongoDB	32	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 93 1 6 0 0 0|2700k 9216B|24.1G 5836k 391M 6737M|1413B 201B

6.6 Testes de recuperação de documentos com percorrimento do grafo

Nestes próximos testes de *benchmark* o objetivo é comparar o desempenho dos gerenciadores em consultas ou recuperações típicas de uma aplicação com persistência poliglota. Neste tipo de consulta de aplicação poliglota, recuperam-se todos os documentos que estão relacionados através da estrutura de um grafo.

6.6.1 Teste percorrendo o grafo recuperando documentos no dataset Amazon

Neste teste de *benchmark*, foram realizadas 10.000 requisições para recuperar documentos por seu identificador junto com todos os documentos relacionados em vários níveis de profundidade através de um grafo. Esta consulta é característica de uma aplicação com persistência poliglota e implica o percorrimento do grafo através das suas arestas. O teste foi realizado com variação de 1 até 6 níveis de profundidade do grafo e utilizando desde 1 até 32 máquinas virtuais executando a aplicação cliente de benchmark. Quando são utilizados sistemas de banco de dados de modelo único, cada consulta deve ser executada em duas etapas. Na primeira etapa, uma consulta é disparada no banco de dados Neo4j para recuperar todos os ids dos documentos associados. No segundo passo, os documentos correspondentes são recuperadas a partir do banco de dados MongoDB usando o conjunto de chaves recuperados. O tempo total da consulta é calculada como uma soma de ambos os tempos do Neo4j e MongoDB. Através das figuras 21(a) até 21(f) é possível constatar que o ArangoDB tem um resultado bom em pesquisas de grafo com *datasets* de pequeno porte e também com pesquisas de profundidade até 3. A partir da profundidade 3, a solução poliglota utilizando simultaneamente os gerenciadores MongoDB e Neo4j apresentam um resultado melhor que com gerenciadores multi-modelo. Neste *dataset* da Amazon os gerenciadores multi-modelo provaram que são eficientes, mas com a tendência de o ArangoDB ser mais eficiente em documentos e recuperações com pouca profundidade no grafo. O OrientDB tende a ter um desempenho satisfatório em operações sobre ambos os modelos de dados, mas não apresenta os melhores resultados comparado com os gerenciadores de modelo único. Através da Tabela 10 constatamos que o consumo de CPU entre os gerenciadores é equilibrado, sendo que todos apresentaram um consumo elevado de CPU quando as requisições simultâneas atingem o número de núcleos da CPU do servidor. Quando esse evento ocorre, o paralelismo no servidor atinge o pico ideal consumindo toda a capacidade de CPU de todos os núcleos, conforme demonstrado na Tabela 10 a partir das linhas 4 a 9 e 13 a 18.

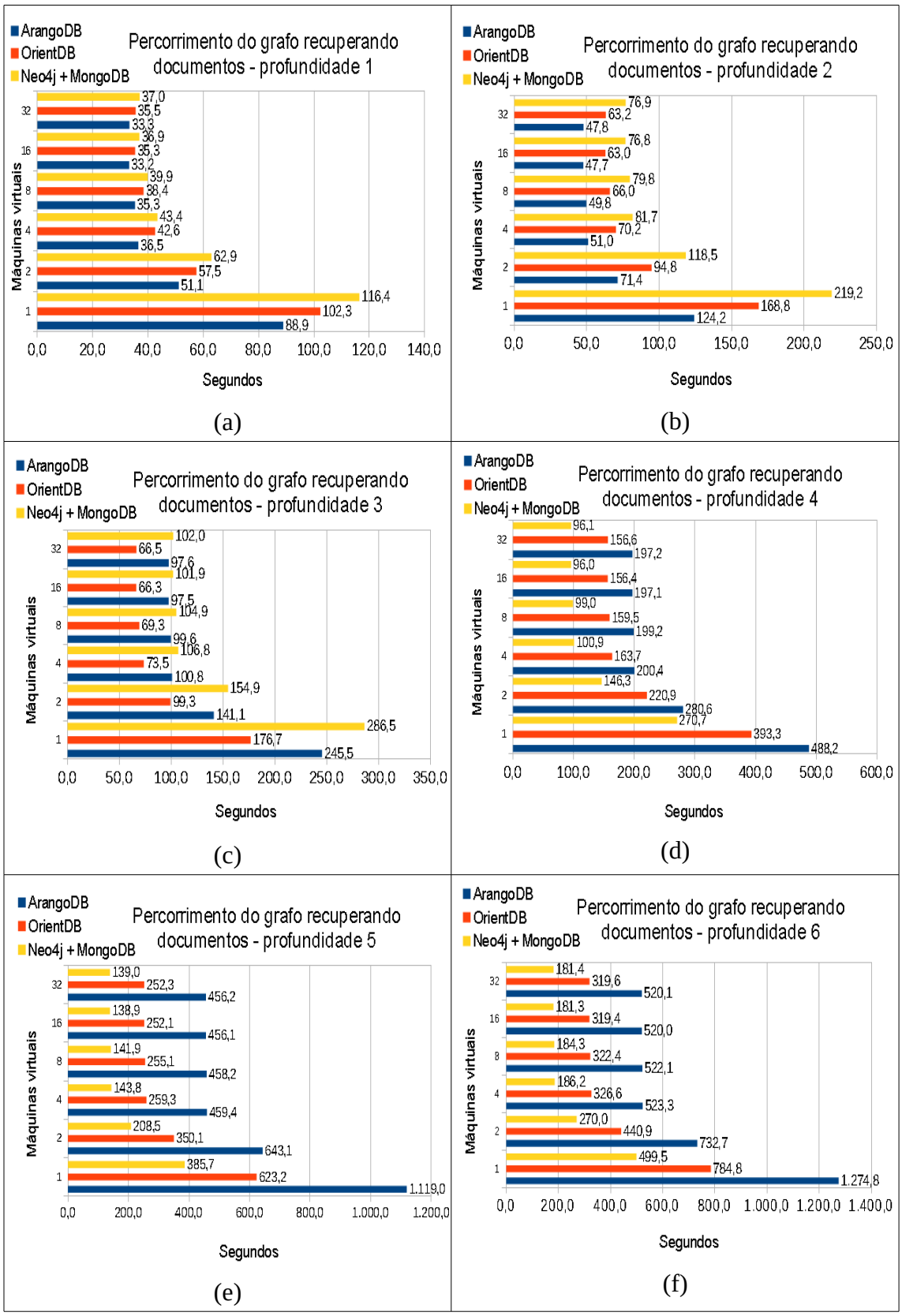

Figura 21: Resultados de desempenho de consultas de documentos com percorrimento do grafo no dataset Amazon. Os documentos estão associados a vértices do grafo e conectados por arestas. As consultas recuperam os documentos relacionados nos niveis (a) 1 até (f) 6 de profundidade. Testes com 10000 consultas utilizando de 1 até 32 maquinas virtuais

Tabela 10. Consumo de recursos no teste de percorrimento no grafo com *dataset* Amazon

	SGBD	Vms	Depth	Recursos consumidos
1	ArangoDB	1	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 28 1 68 3 0 0|1056k 0 |30.6G 2248k 257M 193M|3421B 362B
2	OrientDB	1	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 26 2 70 2 0 0|2860k 9216B|30.7G 2248k 231M 193M|3421B 756B
3	Neo4j	1	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 18 4 78 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|3214B 212B
4	ArangoDB	8	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 88 3 9 0 0 0|1056k 0 |30.6G 2248k 257M 193M|3138B 362B
5	OrientDB	8	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 86 4 7 3 0 0|2860k 9216B|30.7G 2248k 231M 193M|3722B 756B
6	Neo4j	8	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 92 4 2 2 0 0|3196k 0 |24.1G 5836k 391M 6737M|3520B 212B
7	ArangoDB	32	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2050B 362B
8	OrientDB	32	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 98 2 0 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|2024B 756B
9	Neo4j	32	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|2528B 212B
10	ArangoDB	1	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 38 3 58 1 0 0|1056k 0 |30.6G 2248k 257M 193M|2458B 362B
11	OrientDB	1	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 39 2 58 1 0 0|2860k 9216B|30.7G 2248k 231M 193M|2929B 756B
12	Neo4j	1	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 33 8 58 1 0 0|3196k 0 |24.1G 5836k 391M 6737M|2998B 212B
13	ArangoDB	8	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2788B 362B
14	OrientDB	8	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|2122B 756B
15	Neo4j	8	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|2828B 212B
16	ArangoDB	32	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2758B 362B
17	OrientDB	32	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|2526B 756B
18	Neo4j	32	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|2348B 212B

6.6.2 Teste percorrendo o grafo recuperando documentos nos *datasets* sintéticos

Este teste avalia o desempenho da consulta poliglota utilizando os bancos de testes sintéticos pequeno e médio. Como no teste anterior, foram realizadas 10.000 consultas que buscam documentos com seus identificadores gerados aleatoriamente e todos os documentos relacionados por um grafo em diferentes níveis de profundidade. O teste foi realizado com variação de 1 até 6 níveis de profundidade do grafo e utilizando progressivamente desde 1 até 32 máquinas virtuais executando aplicações cliente. Quando são utilizados sistemas de banco de dados de modelo único, cada consulta deve ser executado em duas etapas. Na primeira etapa, uma consulta é disparada no banco de dados Neo4j para recuperar todos os ids dos documentos associados. No segundo passo, os documentos correspondentes são recuperadas a partir do banco de dados MongoDB usando o conjunto de chaves recuperados. O tempo total da consulta é calculada como uma soma de ambos os tempos do Neo4j e MongoDB. As Figuras 22 e 23 mostra os resultados do teste de desempenho para os *datasets* pequeno e grande. Os resultados nas Figuras 22 (a) e 23 (b) mostram que progressivamente ArangoDB tem o melhor desempenho para consultas com percorrimento no grafo de no máximo profundidade 2. A Figura 22 (c) mostra que o desempenho de ArangoDB diminui e OrientDB atinge o melhor desempenho para consultas que requerem o percorrimento do grafo no nível de profundidade 3. A partir da profundidade 4 o uso combinado de sistemas de único modelo Neo4j mais MongoDB obtém o melhor desempenho. Nos testes com o *dataset* médio, conforme demonstrado na Figura 22, o ArangoDB não conseguiu percorrer o grafo com desempenho aceitável, sendo que a partir da profundidade 3 o SGBD consumiu 100% de CPU e não conseguiu processar as requisições conforme demonstrado nas linhas 10, 13 e 16 da Tabela 12. Consideramos que a causa deste comportamento é que como o ArangoDB trabalha com índices em memória, e a busca em grafos necessita percorrer estes índices, portanto há um limite de tamanho de grafo que pode ser representado em memória. O OrientDB tem um desempenho aceitável se comparado com os gerenciadores de modelo único, portanto é uma opção para aplicações com persistência poliglota que necessite percorrer o grafo em profundidade. Finalmente é possível concluir que para uma aplicação de persistência poliglota com percorrimento no grafo em níveis maiores que 3, o melhor desempenho é alcançado combinando Neo4j e MongoDB.

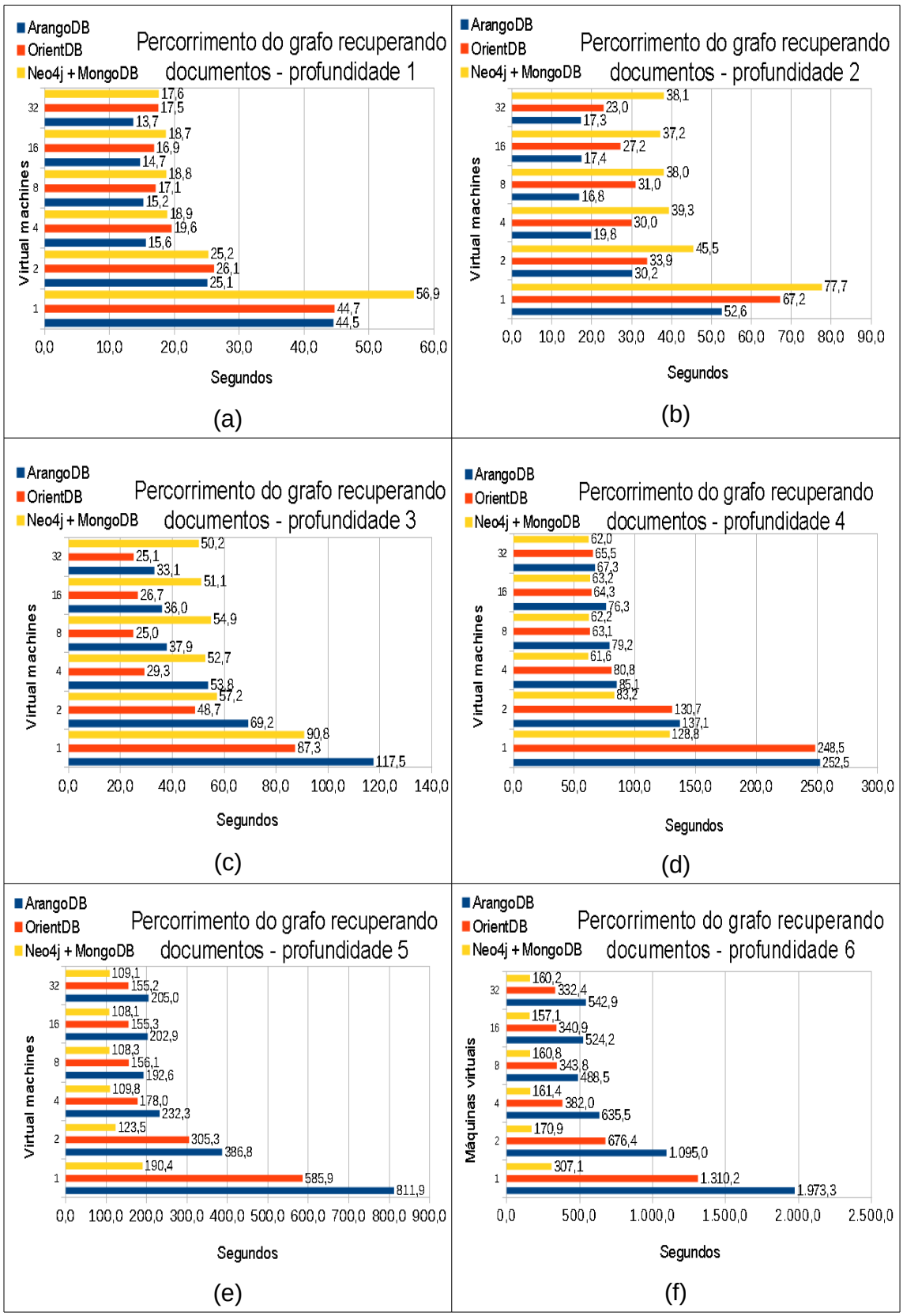

Figura 22. Resultados de desempenho de consultas de documentos com percorrimento do grafo no dataset Pequeno. Os documentos estão associados a vértices do grafo e conectados por arestas. As consultas recuperam os documentos relacionados nos níveis (a) 1 até (f) 6 de profundidade . Testes com 10000 consultas utilizando de 1 até 32 máquinas virtuais

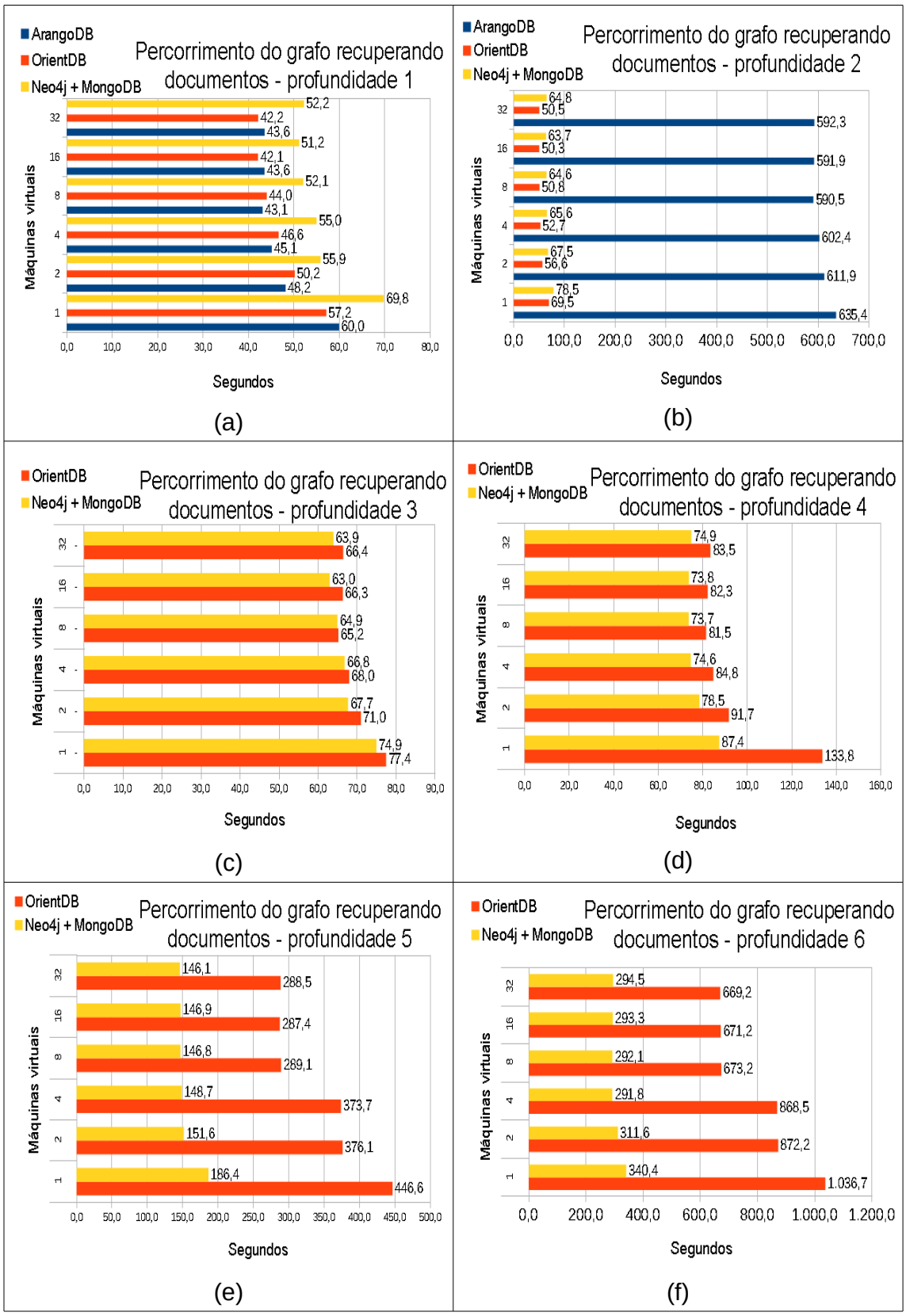

Figura 23. Resultados de desempenho de consultas de documentos com percorrimento do grafo no dataset Médio. Os documentos estão associados a vértices do grafo e conectados por arestas. As consultas recuperam os documentos relacionados nos níveis (a) 1 até (f) 6 de profundidade . Testes com 10000 consultas utilizando de 1 até 32 maquinas virtuais

Tabela 11. Consumo de recursos no teste de percorrimento no grafo com dataset pequeno

	SGBD	Vms	Depth	Recursos consumidos
1	ArangoDB	1	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 38 1 61 0 0 0|1056k 0 |30.6G 2248k 257M 193M|3421B 362B
2	OrientDB	1	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 42 2 56 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|3421B 756B
3	Neo4j	1	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 44 1 55 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|3214B 212B
4	ArangoDB	8	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 92 3 5 0 0 0|1056k 0 |30.6G 2248k 257M 193M|3138B 362B
5	OrientDB	8	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 94 1 5 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|3722B 756B
6	Neo4j	8	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 97 2 1 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|3520B 212B
7	ArangoDB	32	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 98 2 0 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2050B 362B
8	OrientDB	32	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|2024B 756B
9	Neo4j	32	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|2528B 212B
10	ArangoDB	1	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 48 3 48 1 0 0|1056k 0 |30.6G 2248k 257M 193M|2458B 362B
11	OrientDB	1	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 49 2 48 1 0 0|2860k 9216B|30.7G 2248k 231M 193M|2929B 756B
12	Neo4j	1	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 43 8 48 1 0 0|3196k 0 |24.1G 5836k 391M 6737M|2998B 212B
13	ArangoDB	8	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2788B 362B
14	OrientDB	8	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|2122B 756B
15	Neo4j	8	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|2828B 212B
16	ArangoDB	32	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2758B 362B
17	OrientDB	32	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|2526B 756B
18	Neo4j	32	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|2348B 212B

Tabela 12. Consumo de recursos no teste de percorrimento no grafo com dataset médio

	SGBD	Vms	Depth	Recursos consumidos
1	ArangoDB	1	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 27 1 69 3 0 0|1056k 0 |30.6G 2248k 257M 193M|3421B 362B
2	OrientDB	1	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 24 2 72 2 0 0|2860k 9216B|30.7G 2248k 231M 193M|3421B 756B
3	Neo4j	1	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 28 4 68 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|3214B 212B
4	ArangoDB	8	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 88 3 9 0 0 0|1056k 0 |30.6G 2248k 257M 193M|3138B 362B
5	OrientDB	8	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 86 4 7 3 0 0|2860k 9216B|30.7G 2248k 231M 193M|3722B 756B
6	Neo4j	8	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 92 4 2 2 0 0|3196k 0 |24.1G 5836k 391M 6737M|3520B 212B
7	ArangoDB	32	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2050B 362B
8	OrientDB	32	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 98 2 0 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|2024B 756B
9	Neo4j	32	3	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|2528B 212B
10	ArangoDB	1	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2788B 362B
11	OrientDB	1	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 51 2 46 1 0 0|2860k 9216B|30.7G 2248k 231M 193M|2929B 756B
12	Neo4j	1	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 43 8 48 1 0 0|3196k 0 |24.1G 5836k 391M 6737M|2998B 212B
13	ArangoDB	8	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2788B 362B
14	OrientDB	8	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|2122B 756B
15	Neo4j	8	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|2828B 212B
16	ArangoDB	32	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|1056k 0 |30.6G 2248k 257M 193M|2758B 362B
17	OrientDB	32	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|2860k 9216B|30.7G 2248k 231M 193M|2526B 756B
18	Neo4j	32	6	----total-cpu-usage---- -dsk/total- ------memory-usage----- -net/total- usr sys idl wai hiq siq| read writ| used buff cach free| recv send 99 1 0 0 0 0|3196k 0 |24.1G 5836k 391M 6737M|2348B 212B

6.7 Análise dos resultados

Com o resultado dos testes é possível concluir que os níveis de profundidade de percorrimento do grafo e o tamanho do grafo são os parâmetros que mais influenciam o desempenho das consultas em aplicações com persistência poliglota. O banco de dados orientado a grafo Neo4j combinado com o banco de dados de documentos MongoDB tem melhor desempenho em aplicações com consultas que exigem níveis mais profundos de percorrimento em grandes grafos. Como alternativa, o banco de dados multi-modelo OrientDB mostra desempenho semelhante, e em alguns casos até um melhor desempenho em aplicações com consultas em níveis mais profundos de grafos pequenos. Finalmente, o ArangoDB apresenta melhor desempenho em aplicações com consultas em até dois níveis de profundidade de percorrimento de grafos pequenos.

Os testes também sugerem que o modelo de implementação adotado pelos bancos de dados multi-modelo influenciam o desempenho das consultas. A estrutura de *Index Free Adjacency* adotado pelo OrientDB parece ser mais eficiente para consultas que exigem níveis mais profundos de percorrimento em grafos. No entanto, a camada de grafos baseada em documentos adotada pelo ArangoDB parece ser mais eficiente para consultas de grafos pequenos, mas apenas razoável em até dois níveis de profundidade de percorrimento de grafos médios porque necessita de índices em memória para percorrer o grafo.

Os resultados dos testes realizados sugerem também que o modelo de implementação de documentos do ArangoDB apresenta melhor desempenho que o modelo de documentos do OrientDB, portanto observamos uma tendência, nos bancos de dados multi-modelo, em apresentar melhor desempenho em grafos ou documentos, dependendo da estrutura de dados utilizada na implementação. Os testes sugerem que aplicações de persistência poliglota que trabalhem mais com documentos do que grafos, tem melhor desempenho na adoção do ArangoDB, e em aplicações que trabalhem mais com grafos do que documentos, tem melhor desempenho na adoção do OrientDB. Caso a aplicação necessite do melhor desempenho possível em ambos os modelos, a combinação dos gerenciadores MongoDB e Neo4j apresentam um melhor desempenho.

Capítulo 7. Conclusões e trabalhos futuros

Este trabalho teve como foco testar o desempenho de gerenciadores de bancos de dados multi-modelo, comparando os mesmos com gerenciadores de bancos de dados nativo. O objetivo era verificar a viabilidade na adoção desses gerenciadores, analisando também a diferença de implementação dos diferentes modelos. Como contribuição também deste trabalho, foi apresentada uma proposta de benchmark, além de uma ferramenta de benchmark desenvolvida para tal objetivo.

Foi possível concluir que os gerenciadores multi-modelo possuem desempenho suficiente para substituir vários gerenciadores de modelo único em aplicações que necessitam da persistência poliglota. Também foi possível identificar que os gerenciadores multi-modelo tem abordagens diferentes, principalmente na sua implementação do modelo de grafos, e essa abordagem gera uma diferença significativa no desempenho. Foi constatado que a partir dos *datasets* testados, o gerenciador OrientDB tem desempenho melhor que o gerenciador ArangoDB ao percorrer grafos de maior proporção, sendo até inviável percorrer grafos maiores com o gerenciador ArangoDB. Por outro lado, o gerenciador ArangoDB é mais rápido que o OrientDB ao manipular documentos, portanto o critério de escolha tende a ser a distribuição dos modelos de dados, isto é, caso seja necessário trabalhar mais com documentos do que grafos, o ArangoDB pode ser uma alternativa melhor, sendo que o OrientDB é uma opção inversa ao critério. O tamanho dos grafos tende a ser um fator importante também, porque o gerenciador ArangoDB falhou em percorrer grafos de maior porte, sendo que seu limitador é necessitar de índices para percorrer os grafos e que esses índices precisam ser alocados inteiramente na memória.

Apesar de não ser o objeto primário de estudo, foi possível constatar também que alguns gerenciadores são mais produtivos, fornecendo ferramentas e linguagens de consulta muito similares ao SQL tradicional, como no caso do OrientDB. A linguagem de consulta do gerenciador ArangoDB é muito flexível e trabalha de forma aninhada com buscas pelo comando FOR, mas tende a ser trabalhosa com sentenças muito grandes em consultas complexas. Um estudo das linguagens de consulta pode ser interessante também como trabalho futuro, focando na sua facilidade de utilização, sintáxe, desempenho e recursos.

É uma tendência que os gerenciadores de bancos de dados tradicionais também adotem esquemas multi-modelo. A maioria desses gerenciadores já estão suportando o modelo de documentos como XML e o JSON, sendo que alguns já adotaram também o modelo de grafos em versões específicas, ou em bibliotecas externas. É possível também comparar em trabalhos futuros o desempenho dos gerenciadores tradicionais em relação ao suporte de documentos.

Como pesquisa futura seria interessante comparar mais implementações de gerenciadores multi-modelo, analisando também algoritmos clássicos de grafos, além de recursos que afetam o desempenho: esquemas de consistência, gerenciamentos de *locks* e replicação de dados.

Há uma tendência também a que os gerenciadores multi-modelo suportem indexação do tipo *full-text-search* suporte geoespacial, portanto é possível comparar e analisar o desempenho e implementações destas tecnologias.

Outro fator que pode ser explorado é um estudo de desempenho de processamento distribuído nos gerenciadores multi-modelo. Já é comum a utilização de processamento distribuído para buscar os dados nos bancos de dados de forma particionada e otimizada, processando os dados de forma paralela. Em (Rubin, 2016) é demonstrado que a tecnologia *Apache Spark* aliada com o gerenciador MySQL pode acelerar as consultas em 10 vezes.

Uma possibilidade de estudo também poderia não estar vinculada aos gerenciadores multi-modelo, mas sim ao desempenho de processamento distribuído dos gerenciadores do tipo NoSQL. Como exemplo seria possível comparar o desempenho da tecnologia *Apache Phoenix* (Apache Phoenix, 2017) que implementa um banco de dados utilizando o gerenciador NoSQL Hbase e a tecnologia *Apache Spark*. Esta tecnologia parte de uma sentença SQL e compila a mesma para uma série de leituras no banco de dados Hbase, orquestrando um *cluster* fazendo leituras para produzir os resultados das consultas. Seria interessante comparar o desempenho do processamento distribuído desta abordagem com o processamento distribuído nativo de gerenciadores NoSQL do tipo *Map Reduce*.

É possível também comparar a implementação do recurso *Map Reduce* nativo através das consultas dos gerenciadores multi-modelo ArangoDB e

OrientDB, verificando seu desempenho em consultas com *datasets* de maior porte, utilizando um *cluster* de máquinas.

Finalmente é possível também testar o desempenho das consultas em dados particionados, recurso denominado *sharding* e que está disponível nos gerenciadores multi-modelo.

Referências:

Abramova, V. (2013) ‘Nosql databases: Mongodb vs cassandra’. *Proceedings of the International Conference on Computer Science and Software Engineering.* Páginas 14–22. ACM, Nova York, USA

Abubakar, Y., Adeyi, S., Auta, I. (2014) ‘Performance Evaluation of NoSQL Systems using YCSB in a resource Auestere Environment’, *International Journal of Applied Information Systems*, Volume 7, número 8.

Agrawal, D. (2014) ‘Analytics based decision making’. J*ournal of Indian Business Research*. Volume 6, Artigo 4, páginas 332-340

Andlinger, P. (2013) *RDBMS dominate the database market, but NoSQL systems are catching up*. Disponível em: http://db-engines.com/en/blog_post/23. Acessado em 7 Fevereiro 2017.

Apache Phoenix (2017) *OLTP and operational analyctics for Apache Hadoop.* Disponível em: https://phoenix.apache.org/. Acessado em 10 Fevereiro 2017.

Appleby, J. (2014) The Sap HANA. Disponível em: https://blogs.saphana.com/2014/09/09/the-sap-hana-faq. Acessado em 18 Julho 2017.

Aquila, L. (2016) *OrientDB the second generation of multi model NoSQL*. Disponível em: http://pt.slideshare.net/LuigiDellAquila/orientdb-the-2nd-generation-of-multimodel-nosql. Acessado em: 11 Julho 2016.

Atzeni P., Jensen C., Orsi G., Ram S., Tanca L., e Torlone R. (2013) 'The relational model is dead, SQL is dead and I don’t feel so good myself’. *SIGMOD Record 42, 2 (Junho 2013), páginas* 64-68

Baeldung, D. (2017) Introduction to JsonPath. Disponível em: http://www.baeldung.com/guide-to-jayway-jsonpath. Acessado em: 18 Maio 2017.

Barabási, A. L., Albert R. (1999) ‘Emergence of Scaling in Random Networks. Science’.

Baraiya, V. (2016) *Netflix Conductor: A microservices orchestrator.* Disponível em: https://medium.com/netflix-techblog/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40. Acessado em: 18 de Maio 2017.

Barry, C. (2017) *Ransomware attacks on MongoDB*. Disponível em: https://blog.barracuda.com/2017/01/09/ransomware-attacks-on-mongodb-picking-up-steam/. Acessado em: 15 de Julho de 2017.

Berinato, S. (2014). 'With big data comes big responsibility', *Harvard Business Review,* Novembro 2014.

Bilal, A. (2015) *Graph Databases and OrientDB*. Disponível em: http://cs.ulb.ac.be/public/_media/teaching/infoh415/student_projects/orientdb.pdf. Acessado em 1 Fevereiro 2017.

Bittencourt, R. G. (2004) *Aspectos básicos de banco de dados*. Disponível em http://www.marilia.unesp.br/Home/Instituicao/Docentes/EdbertoFerneda/BD-AspectosBasicos.pdf. (Acessado em: 19 Abril 2016).

Bloor, R. The Coming Database Revolution. Disponível em http://www.dataversity.net/archives/5638. Acesso em 06 de Junho de 2016.

Borsos , D. (2013) *New features in Cassandra 2.0 – More on Lightweight Transactions*. Disponível em https://opencredo.com/new-features-in-cassandra-2-0-more-on-lightweight-transactions. Acessado em 18 Novembro 2016.

Brewer, E. (2000) 'Towards Robust Distributed System', *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, página 7.

Broecheler , M. (2013) *Big Graph Data*. Disponível em http://www.slideshare.net/knowfrominfo/big-graph-data. Acessado em 1 Fevereiro 2017.

Carneiro, F. C. F. (2013) 'Repositório de dados relacional ou NoSQL?', *Revista Java Magazine,* v. 114, abr. 2013.

Casanova, M. A. (2012), *Banco de Dados.* Disponível em: http://www.inf.puc-rio.br/~casanova/INF1731-BD/modulo13.pdf. (Acessado em 6 Junho de 2016)

Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R.E. (2008) ‘Bigtable: A distributed Storage System for Structured Data’, *ACM Transactions on Computer Systems*, 26(2), volume 26, página 16

Chede, C. (2013) *Mitos sobre big data*. Disponível em: https://www.ibm.com/developerworks/community/blogs/ctaurion/entry/mitos_sobre_big_data (Acesso em: 01 Setembro 2015).

Chen, H., Chiang, R. H. L., Storey, V.C. (2012) ‘Business Intelligence and Analytics: From Big Data to Big Impact’, *Management Information Systems Quaterly*, páginas 1165-1188.

Claro, D. *Dados estruturados x Dados Semiestruturados x Dados não estruturados*. Disponível em: http://homes.dcc.ufba.br/~dclaro/download/mate04-20121/DadosEstruturadosxSemiEstruturadosxNaoEstruturados.pdf. Acessado em 22 Setembro 2016.

Codd, Edgar Frank ‘Is your DBMS Really Relational’. *Computerworld*, Outubro 1985.

Date, C. J. (2015) *Projeto de Banco de Dados e Teoria Relacional*. 1ª edição. O. Reilly.

Dean, J., Ghemawat S. *MapReduce: Simplified Data Processing on Large Clusters. Sixth Symposium on Operating System Design and Implementation*, Disponível em http://labs.google.com/papers/mapreduce.html. Acessado em: 19 Julho 2016.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W. (2007) Dynamo: Amazon’s Highly Available Key-value Store’, *ACM SIGOPS Operating Systems Review*, Volume 41, Artigo 6, p. 205.

Dell. *PowerEdge Raid Controllers*, Disponível em: https://www.dell.com/downloads/global/products/pvaul/en/perc-technical-guidebook.pdf. Página 23. Acessado em 1 Maio 2017.

Dzhakishev, D. (2014) *NoSQL Databases in the Enterprise. An Experience with Tomra s Receipt Validation System*. Tese de mestrado. Institutt for informatikk.

Edlich, S. *List of NoSQL Databases,* Disponível em: http://nosql-database.org. Acessado em 19 Julho 2016.

Elmasri, R. e Navathe, S. (2011) *Sistemas de bancos de dados*. São Paulo: Addison Wesley.

Evans, D. (2011) *A Internet das coisas,* Disponível em: http://www.cisco.com/web/BR/assets/executives/pdf/internet_of_things_iot_ibsg_0411final.pdf. (Acesso em: 01 Setembro 2015)

Evans, E. (2009) *NoSQL databases*. Disponível em: http://blog.symlink.com/2009/05/12/nosql_2009.html . Acessado em 1 Fevereiro 2017.

Fachin, A. (2007) *Banco de Dados Distribuídos*. Disponível em: http://pt.slideshare.net/andrefachin/banco-de-dados-distribuidos (Acessado: 9 Fevereiro 2017).

Fielding, R. (2000) *Architectural Styles and the Design of Network-based Software Architectures*. Tese de doutorado. Universidade da California.

Fowler, A. (2015) *NoSQL for Dummies,* Estados Unidos: Addison-Wesley

Fowler, M. (2014) *Microservices a new definition of this architectural term*. Disponível em: https://martinfowler.com/articles/microservices.html. Acessado em: 15 de julho de 2017.

Freire, F. (2013) O que é DevOps. Disponível em:https://www.ibm.com/developerworks/community/blogs/rationalbrasil/entry/o_que_devops?lang=en. Acessado em 18 Julho 2017.

Garulli, L. (2012) *Design your application using Persistent Graphs and OrientDB*. Disponível em: https://2012.nosql-matters.org/cgn/wp-content/uploads/2012/06/OrientDB-the-graph-database-2.0.pdf. Acessado em 19 Julho 2016.

Garulli, L. (2014) *Why relationships are cool*. Disponível em: http://pt.slideshare.net/orientdb/austin-data-geeks-why-relationships-are-cool-but-join-sucks. Acessado em 1 Fevereiro 2017.

GraphStream, *Dynamic Graph Library*. Disponível em: http://graphstream-project.org. (Acessado em: 9 Março 2016).

Gray, J. (1993). *The benchmark handbook for database and transaction processing systems*, Estados Unidos: Morgan Kaufmann.

Hadoop, *The Apache Software Foundation*. Disponível em http://hadoop.apache.org. Acessado em 19 Julho 2016.

Hekima, A. (2016) *Como utilizar Big Data para solucionar crises e reagir a desastres?*. Disponível em: http://www.bigdatabusiness.com.br/como-utilizar-big-data-para-solucionar-crises-e-reagir-a-desastres. Acessado em 18 Novembro 2016.

Henricsson, R. (2011) *Document Oriented NoSQL Databases*. Tese de mestrado. Blekinge Institute of Technology.

Intel (2012) *A vision for big data,* Disponível em: http://www.intel.com/content/dam/www/public/us/en/documents/reports/intel-corp-big-data-policy-position-paper.pdf. (Acesso em: 01 Setembro 2015)

Izrailevsky , Y. (2011) *NoSQL at Netflix*. Disponível em http://techblog.netflix.com/2011/01/nosql-at-netflix.html. Acessado em 1 Fevereiro 2017.

Jacobs, A. (2009). 'The pathologies of Big Data' *Magazine Communications of the ACM - A Blind Person's Interaction withTechnology*, Volume 52, número 8.

Jenkov , J. (2015) *Jackson - ObjectMapper*. Disponível em http://tutorials.jenkov.com/java-json/jackson-objectmapper.html. Acessado em 1 Fevereiro 2017.

Jenson, J. (2015) *Neo, Titan & Cassandra study*. Disponível em http://pt.slideshare.net/johnrjenson/no-sql-night-2. Acessado em 1 Fevereiro 2017.

Jouili, S. e Vansteenberghe, V. (2013) ‘An empirical comparison of graph databases’, *IEEE Computer Society: Social Computing*, páginas 708–715.

Kamada, C. M. (2004) *Bancos de dados relacionais*. Disponível em http://pt.slideshare.net/carloscastro2108/bancos-de-dados-relacionais (Acessado: 19 Abril 2015).

Kaminski, L. (2016) *A aldeia global e a contracultura: os meios tradicionais e alternativos de comunicação e a experiência histórica dos anos sessenta e setenta*. Disponível em: http://www.ufrgs.br/alcar/encontros-nacionais-1/9o-encontro-2013/artigos/gt-historia-da-midia-alternativa/a-aldeia-global-e-a-contracultura-os-meios-tradicionais-e-alternativos-de-comunicacao-e-a-experiencia-historica-dos-anos-sessenta-e-setenta. Acessado em: 18 Novembro 2016.

Kokay, M. C. (2016) *Banco de dados NoSQL: Um novo paradigma.* Disponível em: http://www.devmedia.com.br/websys.5/webreader.asp?cat=2&artigo=4773&revista=sqlmagazine_102#a-4773. (Acessado em: 19 Abril 2016)

Kolomicenko, V. (2013) *Analysis and Experimental Comparison of Graph Databases*. Tese de Mestrado. Charles University in Prague.

Korth, H. F. (1995) *Sistema de Banco de Dados,* São Paulo: Makron Books.

Mahajan, A. (2014) *Experiences using Graph database Neo4j for high volume application*. Disponível em: http://asakta.blogspot.com.br/2014/04/experiences-using-graph-database-neo4j.html. Acessado em: 11 Julho 2016.

Majumdar , D. (2016) *A quick survey of MVCC concurrency algorithms.* Disponível em http://gsf.hhg.to/mvcc-survey-1.0.pdf. Acessado em: 18 Novembro 2016.

Matin, A. (2014) *A Mobile-First, Cloud-First Stack at Pearson*. Disponível em https://www.mongodb.com/blog/post/mobile-first-cloud-first-stack-pearson?c=4fbed947c6. Acessado em 1 Fevereiro 2017.

Matos, D. (2016) *Análise de sentimentos e machine learning*. Disponível em http://www.cienciaedados.com/analise-de-sentimentos-e-machine-learning. Acessado em: 18 Novembro 2016.

McAfee, A. e Brynjolfsson, E. 2012, 'Big Data: The Management Revolution', *Harvard Business Review*, volume 90, número 10.

Meir-Huber, M. NoSQL – The Trend for Databases in the Cloud. Disponível em http://soa.sys-con.com/node/1615716. Acesso em 06 de Junho de 2016.

Menasce, D. (2002) *Planejamento de capacidade para serviço web; Métricas, modelos, métodos*. Editora Campus.

Miller, J. (2013) 'Graph Database Applications and Concepts with Neo4j', *AIS Electronic Library*, SAIS 2013 Proceedings paper 24.

Narde, R. (2013) *A Comparison of NoSQL systems*. Tese de mestrado. Rochester Institute of Technology.

Needham, M. (2014) *Optimizing cypher queries in neo4j*. Disponível em: http://pt.slideshare.net/markhneedham/optimizing-cypher?next_slideshow=1. Acessado em 1 Fevereiro 2017.

Nelubin, D. (2013) *Ultra-High Performance NoSQL Benchmarking: Analyzing Durability and Performance Tradeoffs*. Disponível em: http://www.thumbtack.net/solutions/documents/Ultra-HighPerformanceNoSQLBenchmarking.pdf. Acessado em 18 Julho 2017.

Nicholson, E. (1996) *Indexing and Abstracting on the World Wide Web: An Examination of Six Web Databases*. Disponível em: http://scottnicholson.com/pubs/iapaper.pdf. Acessado em 1 Fevereiro 2017.

Ozsu, M. T., Valduriez, P. (2006) *Principles of Distributed Database Systems*. Estados Unidos: Pearson.

Perdomo, A. (2013) *Squire: A polyglot application combining Neo4j, MongoDB, Ruby and Scala*. Disponível em: http://pt.slideshare.net/alberto.perdomo/squire (Acesso em: 9 Fevereiro 2015).

Pickhardt, R. (2012) *Get the Full Neo4j Power by using Core Java API instead of Cypher*. Disponível em: https://dzone.com/articles/get-full-neo4j-power-using. Acessado em 7 Fevereiro 2017.

Pritchett, D. (2008) *BASE: An Acid Alternative.* Disponível em: http://queue.acm.org/detail.cfm?id=1394128. Acessado em: 11 de Julho 2016.

Ramakrishnan R., Gehrke, J. (2008) *Sistemas de Gerenciamento de Bancos de Dados*. São Paulo: McGraw-Hill

Sadalage, P.J. e Fowler, M. (2013) *NoSQL distilled: A brief guide to the emerging world of polyglot persistence*. Estados Unidos: Addison-Wesley Educational Publishers.

Satishbabu, G. (2016) *Understand oracle real application cluster*. Disponível em http://pt.slideshare.net/satishbabugunukula/understand-oracle-real-application-cluster. Acessado em 1 Fevereiro 2017.

Shashank, T. (2014) *Professional NoSQL,* Estados Unidos: Wrox.

Silberschatz, A., Korth, F, Sudarshan, S. (2012) *Sistema de bancos de dados,* 6ª edição, Editora Campus.

Stanford University (2006) *Stanford large network Dataset collection*. Disponível em: https://snap.stanford.edu/data/#amazon (Acesso em: 9 Fevereiro 2015).

Stonebraker, M. (2010) ‘Big Data Means at Least Three Different Things’, *Database Group MIT Computer Science Lab.*

Stonebraker, M. (2016) *Facebook trapped in MySQL ‘fate worse than death’*. Disponível em https://gigaom.com/2011/07/07/facebook-trapped-in-mysql-fate-worse-than-death. Acessado em 1 Fevereiro 2017.

Strauch, C. (2012) ‘NoSQL Databases’, *Workshop do curso de Tecnologia de software*, Stuttgart Media University.

Sullivan, B. (2011) ‘NoSQL, but even less security’. Disponível em: http://blogs.adobe.com/security/files/2011/04/NoSQL-But-Even-Less-Security.pdf?file=2011/04/NoSQL-But-Even-Less-Security.pdf. Acessado em 15 de Julho de 2017.

Suter, R. (2012) ‘MongoDB An introduction and performance analysis’, *Seminário do mestrado em Ciências da Computação*, HSR Hochschule für Technik Rapperswil.

Szárnyas, G. (2017) *Formalizing openCypher Graph Queries*. Disponível em: hhttp://docs.inf.mit.bme.hu/ingraph/pub/btw2017-opencypher.pdf. Acessado em 7 Fevereiro 2017.

Tanaka, A., Silva, V., Paixão A. (2015) *De Business Intelligence a Data Science: um estudo comparativo entre áreas de conhecimento relacionadas*. Disponível em http://www.essentiaeditora.iff.edu.br/index.php/citi/article/view/6347. Acessado em 30 Setembro 2016.

Turatti , M. (2015) *Restheart - The REST API server for MongoDB*. Disponível em http://www.slideshare.net/mkjsix/restheart-the-rest-api-server-for-mongodb. Acessado em 1 Fevereiro 2017.

Vogel , L. (2016) *Apache HttpClient - Tutorial*. Disponível em http://www.vogella.com/tutorials/ApacheHttpClient/article.html. Acessado em 1 Fevereiro 2017.

Vogels , W. (2016) *Amazon DynamoDB – a Fast and Scalable NoSQL Database Service Designed for Internet Scale Applications*. Disponível em http://www.allthingsdistributed.com/2012/01/amazon-dynamodb.html/. Acessado em 18 Novembro 2016.

Vogels, W. (2008) ‘Eventually consistent’, *Communications of the ACM*, Volume 52, páginas 40-44.

Wada, M. (2013) *Walmart uses Neo4j to optimize customer experience with personal recommendations*. Disponível em http://info.neo4j.com/rs/neotechnology/images/neo4j-casestudy-walmart.pdf?_ga=1.139061814.444548602.1485968453. Acessado em 1 Fevereiro 2017.

Webber, J., Robinson, I. and Elfrem, E. (2015) *Graph databases*. Estados Unidos: O. Reilly.

Weinberger , C. (2016) *Index Free Adjacency or Hybrid Indexes for Graph Databases*. Disponível em: https://www.arangodb.com/2016/04/index-free-adjacency-hybrid-indexes-graph-databases. Acessado em: 11 de Julho 2016.

Wieers , D. (2005) Dstat – *Versatile Tool for Generating System Resource Statistics*. Disponível em https://linux.die.net/man/1/dstat. Acessado em 1 Fevereiro 2017.

Zheng, J. G (2012) *Relational databases review*. Disponível em http://jackzheng.net/teaching/it4153/files/database-basics-review.pdf. Acessado em: 11 Julho 2016.

Anexo 1 - Análise do código fonte do OrientDB

- Na classe OrientEdge, nas linhas 4 e 5, constatamos que cada aresta tem dois objetos diretos que representam um Vértice de entrada e um de saída. (Garulli, 2016)

Código fonte 1: Definição da classe OrientEdge

```
1 public class OrientEdge extends OrientElement implements Edge {
2  private static final long serialVersionUID = 1L;
3
4  protected OIdentifiable vOut;
5  protected OIdentifiable vIn;
6  protected String        label;
```

- Na classe OrientVertex.java há um método addEdge, que é chamado através de um objeto vértice (OrientVertex), que executa na linha 8 o método AddEdgeInternal com a lógica de persistência da classe OrientBaseGraph.

Código fonte 2: Método addEdge

```
1 public OrientEdge addEdge(String label, final OrientVertex inVertex, final String
iClassName, final String iClusterName,
      final Object... fields) {
2
3   if (inVertex == null)
4      throw new IllegalArgumentException("destination vertex is null");
5
6    final OrientBaseGraph graph = getGraph();
7    if (graph != null)
8      return graph.addEdgeInternal(this, label, inVertex, iClassName, iClusterName,
fields);

9    return OrientGraphNoTx.addEdgeInternal(null, this, label, inVertex, iClassName,
iClusterName, fields);
10  }
```

- Na classe OrientBaseGraph.java, são criados os objetos de vértices nas linhas 19 e 20. Em seguida nas linhas 72 e 74 os vértices são persistidos como objetos Oidentifiable (atributos to e from).

Código fonte 3: Método addEdgeInternal

```
1 OrientEdge addEdgeInternal(final OrientVertex currentVertex, String label, final
OrientVertex inVertex, final String iClassName,
      final String iClusterName, final Object... fields) {
2    if (currentVertex.checkDeletedInTx())
3      throw new ORecordNotFoundException("The vertex " + currentVertex.getIdentity() +
" has been deleted");
4
5    if (inVertex.checkDeletedInTx())
6      throw new ORecordNotFoundException("The vertex " + inVertex.getIdentity() + " has
been deleted");
7
8    autoStartTransaction();
```

```
9
10  final ODocument outDocument = currentVertex.getRecord();
11  if (outDocument == null)
12    throw new IllegalArgumentException("source vertex is invalid (rid=" +
currentVertex.getIdentity() + ")");

13  if (!ODocumentInternal.getImmutableSchemaClass(outDocument).isVertexType())
      throw new IllegalArgumentException("source record is not a vertex");

14  ODocument inDocument = inVertex.getRecord();
15  if (inDocument == null)
16    throw new IllegalArgumentException("destination vertex is invalid (rid=" +
inVertex.getIdentity() + ")");
17   if (!ODocumentInternal.getImmutableSchemaClass(outDocument).isVertexType())
18    throw new IllegalArgumentException("destination record is not a vertex");
19   OIdentifiable to;
20   OIdentifiable from;
21   label = OrientBaseGraph.encodeClassName(label);
22  if (label == null && iClassName != null)
23    // RETRO-COMPATIBILITY WITH THE SYNTAX CLASS:<CLASS-NAME>
24    label = OrientBaseGraph.encodeClassName(iClassName);
25   if (isUseClassForEdgeLabel()) {
26    final OrientEdgeType edgeType = getEdgeType(label);
27    if (edgeType == null)
28      // AUTO CREATE CLASS
29      createEdgeType(label);
30    else
31      // OVERWRITE CLASS NAME BECAUSE ATTRIBUTES ARE CASE SENSITIVE
32      label = edgeType.getName();
33  }
34   final String outFieldName = currentVertex.getConnectionFieldName(Direction.OUT,
label,
35      settings.isUseVertexFieldsForEdgeLabels());
36  final String inFieldName = currentVertex.getConnectionFieldName(Direction.IN, label,
settings.isUseVertexFieldsForEdgeLabels());
37  // since the label for the edge can potentially get re-assigned
38  // before being pushed into the OrientEdge, the
39  // null check has to go here.
40  if (label == null)
41    throw ExceptionFactory.edgeLabelCanNotBeNull();
42   OrientEdge edge = null;
43    if (currentVertex.canCreateDynamicEdge(outDocument, inDocument, outFieldName,
inFieldName, fields, label)) {
44    // CREATE A LIGHTWEIGHT DYNAMIC EDGE
45    from = currentVertex.rawElement;
46    to = inDocument;
47    if (edge == null) {
48      if (settings.isKeepInMemoryReferences())
49        edge = new OrientEdge(this, from.getIdentity(), to.getIdentity(), label);
50      else
51        edge = new OrientEdge(this, from, to, label);
52    }
53  } else {
54    if (edge == null) {
55      edge = new OrientEdge(this, label, fields);
56       if (settings.isKeepInMemoryReferences())
57        edge.getRecord().fields(OrientBaseGraph.CONNECTION_OUT,
currentVertex.rawElement.getIdentity(),
58            OrientBaseGraph.CONNECTION_IN, inDocument.getIdentity());
59      else
60        edge.getRecord().fields(OrientBaseGraph.CONNECTION_OUT,
currentVertex.rawElement, OrientBaseGraph.CONNECTION_IN,
              inDocument);
61    }
62     from = edge.getRecord();
63    to = edge.getRecord();
64  }
65  if (settings.isKeepInMemoryReferences()) {
66    // USES REFERENCES INSTEAD OF DOCUMENTS
67    from = from.getIdentity();
68    to = to.getIdentity();
69  }
70  edge.save(iClusterName);
71  // OUT-VERTEX ---> IN-VERTEX/EDGE
72  currentVertex.createLink(this, outDocument, to, outFieldName);
73  // IN-VERTEX ---> OUT-VERTEX/EDGE
74  currentVertex.createLink(this, inDocument, from, inFieldName);
```

```
75  outDocument.save();
76  inDocument.save();
77  return edge;
78}
```

- Na classe OrientVertex, nas linhas de 4 a 7 o método createLink cria uma coleção que armazena os vértices. Esse método é chamado através da classe OrientBaseGraph descrita no código fonte 3.

Código fonte 4: Método createLink

```
1 public static Object createLink(final OrientBaseGraph iGraph, final ODocument
iFromVertex, final OIdentifiable iTo,
      final String iFieldName) {

2 if (propType == OType.LINKLIST
3           || (prop != null &&
"true".equalsIgnoreCase(prop.getCustom(OrientVertexType.OrientVertexProperty.ORDERED))))
{
4         final Collection coll = new List(iFromVertex);
5         coll.add(iTo);
6         out = coll;
7         outType = Otype.LINKLIST;

8 return out;
```

- Finalmente na classe Otraverse.java ocorre o percorrimento do grafo sem consulta de índices, apenas varrendo a coleção de objetos Oidentifiable que estão ligados. O método next na linha 29 retorna objetos do tipo Oidentifiable, que no caso do grafo tem como implementação objetos do tipo OrientVertex. Nas classes ORecordID e ObinaryProtocol estão contidas a implementação da leitura do banco de dados de baixo nível recuperando bytes a partir de offsets. (cada Vértice possui um ORecordID que é seu Record ID, isto é, sua posição física de endereço de registro dentro do banco de dados. Com o RID é possível carregar um Vértice rapidamente apenas com uma função de busca a partir de deslocamento (offset) de um Stream.

Código fonte 5: Métodos execute, hasNext e next

```
1 /*
2    * Executes a traverse collecting all the result in the returning
List<OIdentifiable>.
3    *
4    * @see com.orientechnologies.orient.core.command.OCommand#execute()
5    */
6   public List<OIdentifiable> execute() {
7     final List<OIdentifiable> result = new ArrayList<OIdentifiable>();
8     while (hasNext())
9       result.add(next());
10     return result;
11  }
12 public boolean hasNext() {
13    if (limit > 0 && resultCount >= limit)
```

```
14      return false;
15
16    if (lastTraversed == null)
17      // GET THE NEXT
18      lastTraversed = next();
19
20    if (lastTraversed == null && !context.isEmpty())
21      throw new IllegalStateException("Traverse ended abnormally");
22
23    if (!OCommandExecutorAbstract.checkInterruption(context))
24      return false;
25
26    // BROWSE ALL THE RECORDS
27    return lastTraversed != null;
28  }
29 public OIdentifiable next() {
30    if (Thread.interrupted())
31      throw new OCommandExecutionException("The traverse execution has been
interrupted");
32
33    if (lastTraversed != null) {
34      // RETURN LATEST AND RESET IT
35      final OIdentifiable result = lastTraversed;
36      lastTraversed = null;
37      return result;
38    }
39
40    if (limit > 0 && resultCount >= limit)
41      return null;
42
43    OIdentifiable result;
44    OTraverseAbstractProcess<?> toProcess;
45    // RESUME THE LAST PROCESS
46    while ((toProcess = nextProcess()) != null) {
47      result = toProcess.process();
48      if (result != null) {
49        resultCount++;
50        return result;
51      }
52    }
53
54    return null;
55  }
```

Código fonte 6: Método fromStream que retorna um ORecordId

```
1 public ORecordId fromStream(final InputStream iStream) throws IOException {
2    clusterId = OBinaryProtocol.bytes2short(iStream);
3    clusterPosition = OBinaryProtocol.bytes2long(iStream);
4    return this;
5 }
```

Código fonte 7: Método fromStream que retorna um OrecordId e os métodos bytes2short e bytes2long

```
1 public ORecordId fromStream(final InputStream iStream) throws IOException {
2    clusterId = OBinaryProtocol.bytes2short(iStream);
3    clusterPosition = OBinaryProtocol.bytes2long(iStream);
4    return this;
5 }
6
7 public static int bytes2short(final InputStream iStream) throws IOException {
8    return (short) ((iStream.read() << 8) | (iStream.read() & 0xff));
9  }
10
```

```
11
12 public static long bytes2long(final byte[] b, final int offset) {
13    return ((0xff & b[offset + 7]) | (0xff & b[offset + 6]) << 8 | (0xff & b[offset +
14 5 ]) << 16
15        | (long) (0xff & b[offset + 4]) << 24 | (long) (0xff & b[offset + 3]) << 32 |
16(long) (0xff & b[offset + 2]) << 40
17        | (long) (0xff & b[offset + 1]) << 48 | (long) (0xff & b[offset]) << 56);
18  }
```

Anexo 2 - Análise do código fonte do ArangoDB

- No arquivo de header Graphs.h há a definição da classe Graph onde constatamos uma coleção de vértices e arestas nas linhas 6 e 10. A estrutura de arestas é um documento JSON completamente separado dos documentos de vértices.

Código fonte 6: Definição da classe Graph

```
1 class Graph {

2       private:
3 ////////////////////////////////////////////////////////////////////////////
4 ///  @brief the cids of all vertexCollections
5 ////////////////////////////////////////////////////////////////////////////
6        std::unordered_set<std::string> _vertexColls;
7 ////////////////////////////////////////////////////////////////////////////
8 ///  @brief the cids of all edgeCollections
9 ////////////////////////////////////////////////////////////////////////////
10       std::unordered_set<std::string> _edgeColls;
```

- No arquivo fonte Graphs.cpp, nas linhas 10, 14 e 17 constatamos a leitura do documento JSON onde é obtida a coleção das arestas através dos elementos “from” e dos elementos “to”, inserindo na coleção de vértices os vértices relacionados.

Código fonte 7: Definição do construtor da classe Graph

```
1 Graph::Graph(triagens::basics::Json const& j) :
2   _vertexColls(),
3   _edgeColls()
4 {
5   auto jsonDef = j.get(_attrEdgeDefs);
6
7   for (size_t i = 0; i < jsonDef.size(); ++i) {
8     Json def = jsonDef.at(i);
9     TRI_ASSERT(def.isObject());
10    Json e = def.get("collection");
11    TRI_ASSERT(e.isString());
12    std::string eCol = JsonHelper::getStringValue(e.json(), "");
13    addEdgeCollection(eCol);
14    e = def.get("from");
15    TRI_ASSERT(e.isArray());
16    insertVertexCollectionsFromJsonArray(e);
17    e = def.get("to");
18    TRI_ASSERT(e.isArray());
19    insertVertexCollectionsFromJsonArray(e);
20  }
21  auto orphans = j.get(_attrOrphans);
22  insertVertexCollectionsFromJsonArray(orphans);
23 }
```

- No arquivo fonte TraversalNode.cpp constatamos na linha 5 que antes de percorrer os vértices, é calculado o custo dessa operação utilizando um índice específico para a coleção de arestas.

Código fonte 7: Método estimateCost

```
1 double TraversalNode::estimateCost (size_t& nrItems) const {

2   TRI_ASSERT(collection != nullptr);
3
4    for (auto const& index : collection->getIndexes()) {
5      if (index->type == triagens::arango::Index::IndexType::TRI_IDX_TYPE_EDGE_INDEX)
6{
7        // We can only use Edge Index
8        if (index->hasSelectivityEstimate()) {
9          expectedEdgesPerDepth += 1 / index->selectivityEstimate();
10        }
11        else {
12          expectedEdgesPerDepth += 1000; // Hard-coded
13        }
14        break;
15      }
16    }
```

- Finalmente no arquivo TraversalBlock.cpp nas linhas 4, 7, 10 e 15 ocorre o percorrimento do Grafo retornando os vértices recuperados no método getReadDocuments. Nas linhas 15, 17 e 18 o índice é utilizado.

Código fonte 8: Método morePaths

```
1 bool TraversalBlock::morePaths (size_t hint) {
2
3 for (size_t j = 0; j < hint; ++j) {
4    std::unique_ptr<triagens::arango::traverser::TraversalPath> p(_traverser→next());
5
6 if ( usesVertexOutput() ) {
7      _vertices.emplace_back(p->lastVertexToJson(_trx, _resolver));
8    }
9    if ( usesEdgeOutput() ) {
10      _edges.emplace_back(p->lastEdgeToJson(_trx, _resolver));
11    }
12    if ( usesPathOutput() ) {
13      _paths.emplace_back(pathValue);
14    }
15    _engine->_stats.scannedIndex += p→getReadDocuments();
16}
17_engine->_stats.scannedIndex += _traverser->getAndResetReadDocuments();
18_engine->_stats.filtered += _traverser->getAndResetFilteredPaths();
19// This is only save as long as _vertices is still build
20return ! _vertices.empty();
```