UNIVERSIDADE ESTADUAL DO NORTE DO PARANÁ

***CAMPUS* LUIZ MENEGHEL - CENTRO DE CIÊNCIAS TECNOLÓGICAS**

SISTEMAS DE INFORMAÇÃO

RAMON FREITAS DE OLIVEIRA

ANÁLISE DE DESEMPENHO DE BANCOS DE DADOS NOSQL PARA CONSTRUÇÃO DE APLICATIVOS NA NUVEM UTILIZANDO PAAS

Bandeirantes

2015

RAMON FREITAS DE OLIVEIRA

ANÁLISE DE DESEMPENHO DE BANCOS DE DADOS NOSQL PARA CONSTRUÇÃO DE APLICATIVOS NA NUVEM UTILIZANDO PAAS

Trabalho de Conclusão de Curso submetido à Universidade Estadual do Norte do Paraná, como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador: Prof. Ricardo Gonçalves Coelho

Bandeirantes

2015

RAMON FREITAS DE OLIVEIRA

ANÁLISE DE DESEMPENHO DE BANCOS DE DADOS NOSQL PARA CONSTRUÇÃO DE APLICATIVOS NA NUVEM UTILIZANDO PAAS

Trabalho de Conclusão de Curso submetido à Universidade Estadual do Norte do Paraná, como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

COMISSÃO EXAMINADORA

_____
Prof. M.e Ricardo Gonçalves Coelho
UENP – *Campus* Luiz Meneghel

_____
Prof. M.e Rodrigo Tomaz Pagno
UENP – *Campus* Luiz Meneghel

_____
Prof. M.ª Rafaella Aline Lopes da Silva
UENP – *Campus* Luiz Meneghel

Bandeirantes, __ de_____ de 2015

RESUMO

Este trabalho está inserido no contexto de bancos de dados NoSQL implantados em um serviço de nuvem do tipo PaaS. Propõe-se uma arquitetura abrangente para testes de desempenho de bancos de dados NoSQL, cujo principal diferencial é a inclusão do esquema de dados. Em seguida, o teste foi implementado no serviço de PaaS Google Appengine e executado. Os resultados do teste foram descritos na forma de gráficos e analisados.

Palavras-chave: Bancos de dados. Esquemas de dados. *Benchmark*.

ABSTRACT

This paper regards the performance of NoSQL databases that are implanted in PaaS cloud services. It proposes a broad architecture for performance tests, where the inclusion of the data schema is the main novelty. Also, the test architecture was implanted in the Google AppEngine PaaS cloud service and a series of tests was executed. The results were described in charts and analyzed.

LISTA DE FIGURAS

LISTA DE SIGLAS

ACID	Atomocidade, Consistência, Isolamento, Durabilidade
CAP	*Consistency, Availability, Partition tolerance*
JDO	*Java Data Objects*
JPA	*Java Persistence API*
NoSQL	*Not-only* SQL
PaaS	*Platform as a Service*
SGBD	Sistema Gerenciador de Banco de Dados
YCSB	*Yahoo! Cloud Serving Benchmark*

SUMÁRIO

1. INTRODUÇÃO

A tecnologia de sistemas de informação está em constante processo de mudança e aprimoramento. O número de usuários da Internet tem crescido rapidamente, assim como a demanda por espaço de armazenamento de dados e uso de recursos computacionais. Numa época em que a demanda era consideravelmente menor que a atual, foram desenvolvidas diversas soluções e conceitos de bancos de dados que se mostraram eficientes.

Com o advento do fenômeno conhecido hoje como Web 2.0, o qual segundo Murugesan (2007) “é uma coleção de tecnologias, estratégias de negócio e tendências sociais (…) mais dinâmica que a predecessora”, estudiosos passaram a procurar novas formas de ligar com um grande número de requisições a sistemas de bancos de dados e novas exigências. Para lidar com o contexto atual, foram desenvolvidas soluções de bancos de dados conhecidas como NoSQL. Estes bancos rompem com padrões já tradicionais e buscam atender a um grande número de usuários simultâneos, garantindo ao menos dois dos três seguinte fatores: consistência, disponibilidade e particionamento.

De acordo com Leavitt (2010) “o que os bancos NoSQL têm em comum é que não são relacionais. Sua maior vantagem é que (…) gerenciam dados não-estruturados e mídia social mais eficientemente”.

1.1 CONTEXTO E DELIMITAÇÃO DO TRABALHO

O contexto deste trabalho é o desempenho de bancos de dados NoSQL. Muitas vezes eles são disponibilizados em uma nuvem na forma de PaaS. Provedores de PaaS em geral disponibilizam acesso aos seu serviços por meio de linguagens de programa e *frameworks*. Neste trabalho, foi testado somente o PaaS Google AppEngine com as linguagens de programação Java e Python, e os *frameworks* JPA e JDO, ambos feitos para Java.

1.2 Formulação do Problema

Atualmente existem variadas soluções NoSQL disponíveis para uso, sendo algumas de código aberto ou fechado, disponíveis em uma PaaS ou prontas para serem implantadas em infraestrutra própria. Por ser uma tecnologia relativamente nova, ainda existe muito a ser explorado e são necessários trabalhos que apontem para caminhos a serem seguidos. Portanto, ao decidir fazer uso de um sistema NoSQL, o desenvolvedor pode se deparar com uma série de dúvidas.

O acesso ao banco de dados pelo aplicativo geralmente é feito com o uso de um *framework*, disponível para uma determinada linguagem de programação. O Google AppEngine possui suporte aos *frameworks* JPA e JDO, os quais fornecem maior transparência e agilidade ao desenvolvimento. Porém, a diferença de desempenho entre os dois pode ser considerável, como demonstrado por Saúde, Melchiori e Resende (2015).

Dentre outros fatores que podem afetar o desempenho de um aplicativo que faz uso do NoSQL, alguns são facilmente identificáveis e outros não. O esquema de dados é um fator geralmente negligenciado por desenvolvedores inexperientes. Scherzinger et al. (2013) mostraram que o esquema de dados implementado pode ter grande impacto na performance de um aplicativo que faz uso da tecnologia NoSQL, e o pior aspecto do problema é que ele é somente percebido quando o aplicativo está servindo uma grande quantidade de usuários.

Diante de tantos fatores em questão, este trabalho busca auxiliar o desenvolvedor e o analista no processo de tomada de decisões, possibilitando-os responder à seguinte pergunta: como escolher o que é melhor para meu aplicativo?

1.3 OBJETIVOS

1.3.1 Objetivo Geral

O objetivo deste trabalho é descrever a arquitetura de um teste e aplicá-lo a um banco de dados NoSQL, buscando identificar diferenças de desempenho quando o teste é executado em diferentes contextos, sendo a inclusão do esquema de dados seu principal diferencial.

1.3.2 Objetivos Específicos

- Elaborar uma arquitetura para o teste a ser realizado;
- Implementar a arquitetura no ambiente do Google AppEngine;
- Realizar testes comparativos entre os diferentes *frameworks* utilizados e esquemas de dados implementados;
- Analisar os resultados dos testes, chegando a conclusões e recomendações.

1.4 JUSTIFICATIVA

Ao contrário dos bancos de dados relacionais, para os quais existe um grande número de estudos, experiências compartilhadas e recomendações amplamente adotadas, ainda há diversas dúvidas quanto ao uso otimizado de sistemas NoSQL.

Segundo Lóscio, Oliveira e Pontes (2012), bancos relacionais se mostraram eficientes ao lidar com uma grande demanda de usuários e requisições durante décadas. Mas com a popularização da Internet e de serviços que se incluem na categoria Web 2.0, dos quais destacam-se as redes sociais, faz-se necessário o desenvolvimento de tecnologias capazes de lidar com uma demanda ainda maior. A comunidade de bancos de dados percebeu que as soluções tradicionais não são capazes de suportar tal expansão e identificou uma série de limitações.

Em um ambiente competitivo, a experiência do usuário, a qual é afetada pelo desempenho do aplicativo, se torna um fator determinante para o sucesso. Companhias

como Google, Microsoft e Facebook perceberam isto e desenvolveram suas próprias soluções.

Soluções NoSQL não estão disponíveis apenas para grandes empresas, mas também para pequenos desenvolvedores. O Google disponibiliza o Apache Cassandra, originalmente desenvolvido pela equipe do Facebook, e o Google Datastore em um serviço de nuvem PaaS (PRODAN; SPERK; OSTERMANN, 2011).

A Microsoft disponibiliza um serviço de PaaS com armazenamento de dados chamado de Windows Azure (CALDER et al., 2011).

Os serviços citados disponibilizam uma grande estrutura física, composta por milhares de servidores espalhados pelo mundo, possibilitando que pequenos desenvolvedores construam aplicativos altamente escaláveis e competitivos.

1.5 Organização do Trabalho

A seção 2 apresenta a fundamentação teórica. Nesta seção são explicitados conceitos que foram importantes para o desenvolvimento do trabalho, como Web 2.0, bancos de dados e teorema CAP. A seção 3 contém o desenvolvimento do trabalho. Nesta seção, a arquitetura de um teste explicada em detalhes. A seção 3.1.2 descreve suas etapa e a seção 3.1.3 sua arquitetura. Os detalhes da implementação do teste no ambiente do Google AppEngine estão na seção 3.2, que também inclui os resultados de um teste realizado e uma análise destes resultados, nas seções 3.2.3 e 3.2.4, respectivamente. A seção 4 contém a conclusão do trabalho.

2. FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta conceitos e trabalhos que serviram como base para este trabalho.

2.1 A *WEB* 2.0

O termo *Web* 2.0 foi criado para descrever o atual contexto da Internet, um ambiente repleto de serviços como redes sociais e *wikis*. Estes serviços devem ser capazes de ligar com um grande e crescente número de usuários simultâneos e seus dados. Esta demanda traz novos desafios para desenvolvedores, os quais não existiam no começo da popularização da Internet (O'REILLY, 2007).

Um famoso exemplo é o Facebook, a maior rede social do mundo, a qual possui mais de um bilhão de usuários ativos e precisa lidar com bilhões de requisições por dia (LAKSHMAN; MALIK, 2010).

2.2 BANCOS DE DADOS RELACIONAIS E SUAS LIMITAÇÕES

Há décadas, o paradigma relacional tem sido predominante em bancos de dados. Uma de suas características básicas e o processamento de transações, que faz uso das propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Pokorny (2011) define estas propriedades da seguinte forma:

- Atomicidade: ou todas as transações são realizadas, ou nada é;
- Consistência: o resultado das transações são dados válidos;
- Isolamento: transações são independentes;
- Durabilidade: o banco sobrevive a falhas do sistema.

Com o tempo, a comunidade de bancos de dados notou que estas propriedades são importantes apenas em alguns contextos, porém dispensáveis em outras. Em aplicativos que não lidam com dados sensíveis e importantes, algumas propriedades

ACID podem ser deixadas de lado, de forma a possibilitar ganho de desempenho. Por exemplo, Pokorny (2011) diz que ao se dispensar consistência, ganha-se mais disponibilidade e escalabilidade.

Leavitt (2010) diz que para que um aplicativo seja escalável além de um certo ponto, é necessário o uso da escalabilidade horizontal e diversos servidores; porém, bancos de dados relacionais não foram projetados para funcionar facilmente de maneira distribuída nem trabalhar com dados particionados.

Dentre outras limitações, Leavitt (2010) menciona:

- Complexidade: bancos relacionais devem ter seus dados convertidos para tabelas, mas às vezes os dados não se encaixam facilmente em tabelas;
- SQL: usar a linguagem SQL é conveniente para dados estruturados, mas pode produzir grandes quantidades de código complexo e não funciona bem com desenvolvimento ágil e moderno;
- Grande número de recursos: bancos relacionais oferecem uma grande gama de recursos e integridade de dados, mas muitas vezes desenvolvedores não necessitam de todos os recursos nem o custo de complexidade que eles adicionam.

2.3 Consistência, Disponibilidade e Particionamento

Sistemas de bancos de dados voltados à Web 2.0, geralmente buscam implementar soluções que garantam consistência, particionamento e disponibilidade, explicados por Pokorny (2011) da seguinte forma:

- Consistência significa que sempre que um dado é escrito, todos os que o lerem do mesmo banco de dados verão a última versão dele, diferentemente da noção usada pelo conceito de ACID.
- Disponibilidade é geralmente alcançada quando se tem um grande número de servidores físicos agindo como um único banco de dados pelo compartilhamento de dados entre vários nós e réplicas.
- Particionamento, ou tolerância ao particionamento, significa que o banco de dados ainda pode lido e escrito sobre, quando partes dele estão

completamente inacessíveis. Esta situação pode ocorrer quando uma ligação entre nós da rede é interrompida.

Tolerância ao particionamento pode ser alcançada com mecanismos que garantam que escritas destinadas a nós inacessíveis sejam redirecionadas para nós acessíveis. Então, quando os nós inacessíveis voltarem, eles receberam as escritas perdidas.

Quanto aos três items mencionados, Brewer (2000) formulou o teorema CAP, o qual foi provado por Gilbert e Lynch (2002). Segundo Gilbert e Lynch (2002), é impossível garantir consistência, disponibilidade e particionamento ao mesmo tempo.

A Figura 2.1, inserida abaixo, exibe este conceito visualmente.

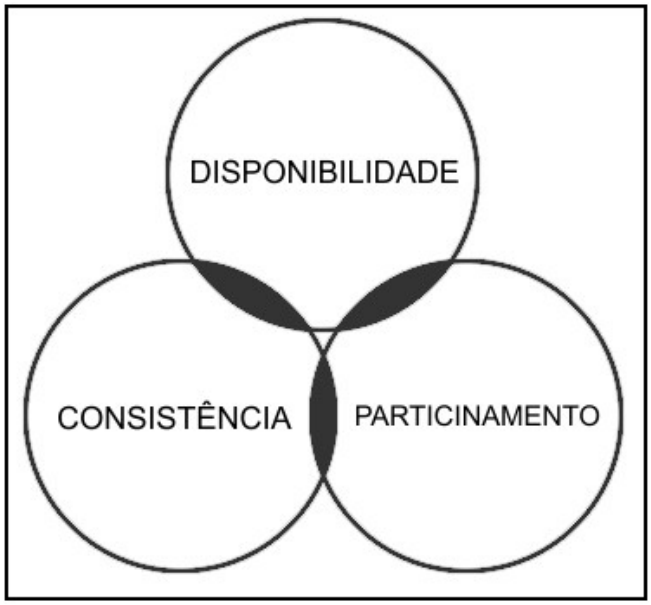

Figura 2.1 – Teorema CAP, em nenhum ponto todos os três círculos se tocam, mostrando que é impossível garantir os três itens ao mesmo tempo. Fonte: Autoria própria.

Ainda há controvérsias na comunidade acadêmica quanto a este teorema. Por exemplo, Calder et al. (2011) defendem que o *Windows Azure Storage* disponibiliza todos os três itens ao mesmo tempo.

2.4 NoSQL

Certas características garantidas por bancos de dados relacionais, como a alta consistência de dados são importantes em alguns contextos, como armazenagem de dados bancários; porém, dispensáveis em casos onde os dados armazenados não possuem grande importância, de forma a otimizar a disponibilidade e o particionamento dos dados (POKORNY, 2011).

Devido aos requisitos da Web 2.0, foram criados sistemas de bancos de dados não-relacionais conhecidos como NoSQL; estes bancos não seguem o padrão relacional e são preparados para atender às necessidades da Web 2.0 (HETCH; JABLONSKY, 2011).

Um grande número de desenvolvedores e usuários começaram a se voltar para bancos NoSQL. NoSQL significa “não somente SQL”, diferente de “não SQL”, como é comum de se pensar (LEAVITT, 2010).

Estas soluções se dividiram em quatro categorias quanto ao modelo de dados que utilizam (HECHT; JABLONSKI, 2011):

- Chave-valor: os dados são organizados em grandes tabelas onde cada linha possui uma chave, e pelo menos um valor, em um estrutura semelhante a mapas ou dicionários. São completamente livres de esquema, novos valores de qualquer tipo podem ser inseridos a qualquer tempo sem conflito com outros dados. O agrupamento de chaves e valores em coleções é a única possibilidade oferecida para se adicionar certo tipo de estrutura ao esquema de dados.
- Famílias de colunas: orientado ao armazenamento de muitos dados em muitas máquinas, onde colunas são organizadas por famílias de colunas;
- Orientados a grafos: implementam a estrutura de grafos para armazenar dados de maneira escalável, são orientados para grande quantidade de dados relacionados entre si;
- Orientados a documentos: basicamente utiliza documentos, os quais podem apontar para outros. Dentro de cada documento, as chaves devem ser únicas.

As soluções NoSQL têm algumas características em comum: escalabilidade horizontal, ausência de esquema ou esquema flexível, suporte nativo a replicação, consistência eventual (LÓSCIO; OLIVEIRA; PONTES, 2011).

Escalabilidade horizontal se trata da capacidade do banco NoSQL deve ter de agregar novas máquinas à sua infraestrutura, de forma a otimizar o processamento de requisições e se adaptar a uma crescente demanda. Geralmente, as soluções NoSQL são projetadas desde o esboço para possuírem esta capacidade.

O Apache Cassandra é capaz de lidar facilmente com a escalabilidade horizontal, implementando diversas técnicas já consolidadas para se propagar ao longo de uma rede de máquinas. O Facebook, desenvolvedor e maior usuário do Cassandra, possui clusters de servidores no mundo todo, prontos para servir às necessidades dos usuários em sua região (LAKSHMAN; MALIK, 2011).

A falta um esquema pré-definido e fixo que defina a organização dos dados é comum em soluções NoSQL. Geralmente, em bancos relacionais, a estrutura de tabelas e relações entre elas é definida após uma análise do contexto do aplicativo e geração de um esquema, e antes da inserção dos dados. Em bancos NoSQL, não é necessária uma definição prévia do esquema; porém os dados podem ser organizados em hierarquias e grupos, dependendo do sistema utilizado. O Google Megastore, por exemplo, possui esta capacidade, sendo que os dados são estruturados no momento da inserção no banco, seguindo um esquema de dados implementado em uma linguagem de programação (Baker el at., 2011). Suporte nativo a replicação é outra característica marcante do NoSQL, permitindo acesso mais rápido aos dados. O Apache Cassandra implementa a replicação utilizando estratégias bem consolidadas (LAKSHMAN; MALIK, 2010).

Bancos NoSQL são capazes de garantir consistência em alguns casos. No Google Megastore, um grupo de entidades é armazenado na mesma máquina e as transações envolvendo este grupo são submetidas a um controle. Desta forma, é a garantida a persistência dos dados neste grupo (BAKER et al., 2011).

Os bancos NoSQL têm se mostrado eficientes no tratamento de grandes quantidades de dados simultaneamente e consecutivamente. Cooper et al.(2010) provaram que os bancos de dados NoSQL, quando submetidos a certas cargas de operações, foram superiores ao MySQL, o banco relacional testado, pois conseguiram processar uma maior quantidade de requisições em menor tempo.

O Apache Cassandra, desenvolvido pela equipe do Facebook, tem sido capaz de lidar com a enorme demanda da rede social. É “um sistema de armazenamento de dados distribuído para armazenar grande quantidade de dados estruturados espalhados em vários servidores, proporcionando alta disponibilidade sem nenhum ponto de falha” (LAKSHMAN; MALIK, 2010).

O Google BigTable, solução NoSQL baseada em chave-valor do Google, tem sido usado por diversos serviços da empresa a muitos anos. Tem se mostrando eficiente ao atender a demanda. De acordo com Chang et al. (2008), até agosto de 2006 o Bigtable já era usado por mais de 60 projetos do Google.

2.5 GOOGLE DATASTORE E GOOGLE APPENGINE

Hoje existem diversas soluções disponíveis, sendo algumas na forma de PaaS. Em uma PaaS o usuário implanta seus aplicativos numa infraestrutura em nuvem utilizando bibliotecas, linguagens de programação, serviços e ferramentas disponibilizados pelo provedor da nuvem. O consumidor não gerencia ou controla a infraestrutura da nuvem, mas tem acesso aos aplicativos por ele implementados e suas configurações (MELL; GRANCE, 2009).

O Google Datastore (www.cloud.google.com/datastore) é um SGBD desenvolvido pelo Google acima do Megastore. Introduz recursos semelhantes aos SGBDs relacionais para gerenciamento, uso e manutenção dos dados. É disponibilizado no Google AppEngine.

O AppEngine é um ambiente completo em uma PaaS no qual qualquer desenvolvedor pode hospedar um aplicativo que faz uso dos recursos poderosos do Datastore. Suporta atualmente três linguagens de programação: Python, PHP, *Go* e Java. Para o acesso ao Datastore por um aplicativo, existem os seguintes *frameworks* disponíveis:

- Python: com biblioteca padrão do Google;
- Go: biblioteca padrão do Google;
- Java: biblioteca padrão do Google, JPA, JDO e *frameworks* de terceiros.

Aplicativos desenvolvidos com o AppEngine são hospedados por padrão numa página de Internet com o sufixo de domínio *.appspot.com*, sendo que cada um possui um banco de dados próprio.

2.6 Fatores que Afetam o Desempenho de Bancos NoSQL

Um banco de dados geralmente é implementado em uma infraestrutura que inclui servidores, redes, e software, tendo seus dados organizados de acordo com um esquema de dados não-fixo, geralmente implementado no código fonte. Desta forma, cada um dos fatores citados anteriormente podem ter impacto no aplicativo final.

Soluções NoSQL como o Google Datastore (www.cloud.google.com/datastore), MongoDB (www.mongodb.org) e Apache Cassandra (www.cassandra.apache.org) foram construídos utilizando-se técnicas conhecidas ou novas para lidar com escalabilidade, particionamento e disponibilidade. Porém, decisões erradas do analista ou programador podem prejudicar consideravelmente o desempenho do aplicativo a ser desenvolvido. Um analista inexperiente com NoSQL pode modelar um esquema de dados de forma a expressar uma óbvia relação de hirarquia entre entidades, seguindo convenções do modelo relacional; porém, isto não funcionaria bem no Google Datastore (SCHERZINGER et al., 2013).

Os fatores citados podem ser classificados em três camadas:

- Camada física - inclui os servidores, grupos de servidores e componentes de rede;
- Camada de sofware - inclui o banco de dados, *frameworks* e linguagens de programação utilizados;
- Camada lógica - inclui o esquema e esquema de dados utilizado.

2.7 Sobre o Impacto do Esquema de Dados no Desempenho de Aplicativos

O esquema não é um fator óbvio, mas é especialmente importante em NoSQL. Segundo Scherzinger et al.(2013), “o esquema de dados NoSQL utilizado tem impacto direto em aplicativos *web*. Especialmente para desenvolvedores com pouca experiência em NoSQL, os riscos inerentes em um esquema pobre podem ser incalculáveis. Pior ainda, os problemas vão somente se manifestar após a implantação do sistema, quando base crescente de usuários causa grande concorrência de escritas”.

Para provar o impacto sugerido, Scherzinger et al.(2013) criaram o seguinte contexto de aplicativo a ser estudado: um *blog* onde cada usuário pode postar ilimitado número de artigos e comentários, sendo que cada artigo pode conter um número ilimitado de comentários.

Scherzinger et al.(2013) criaram dois diferentes esquemas de dados: orientado a artigos e orientado a usuários.

O esquema orientado a artigos segue o paradigma relacional, no qual uma hierarquia é clara. Entidades do tipo *comments* são filhas de entidades do tipo *articles*, que por sua vez são filhas de entidades do tipo *users*. Desta forma são criados grupos de entidades *comments* e *articles*. A Figura 2.2 exibe o esquema visualmente:

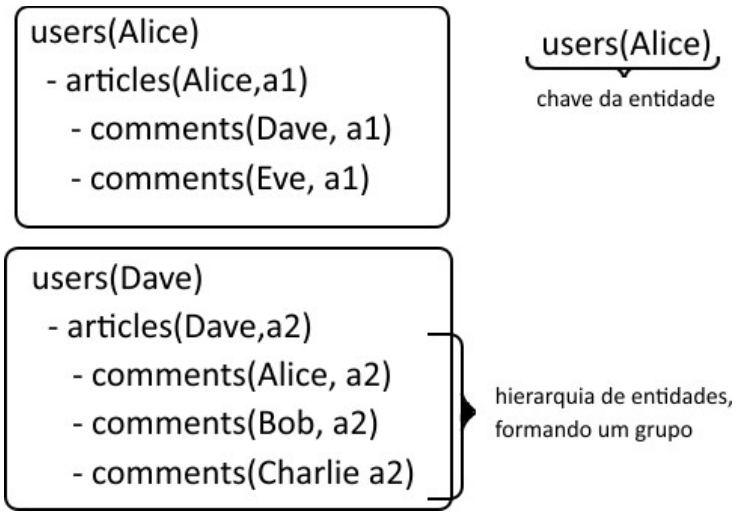

Figura 2.2 – Esquema de dados orientado a artigos. Fonte: SCHERZINGER et al., 2013.

No esquema orientado a usuários a hierarquia não é clara. Entidades *comments* e *articles* podem ambas ser filhas de *users*. A Figura 2.3 exibe a estrutura:

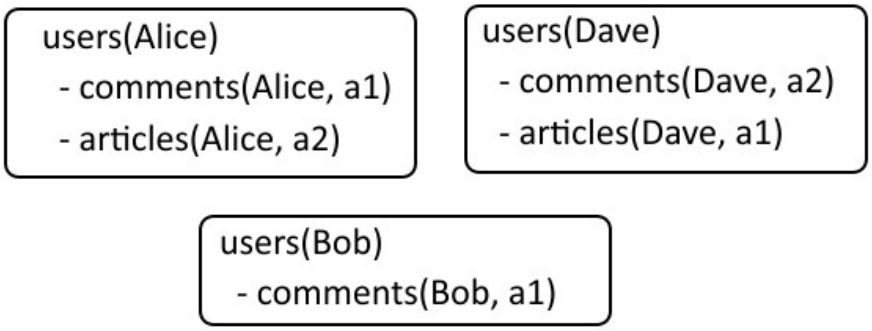

Figura 2.3 – Esquema de dados orientado a usuários. Fonte: SCHERZINGER et al., 2013.

Scherzinger et al.(2013) explicam que o Google Datastore garante consistência aos dados que pertencem a um grupo de entidades. Porém, grupos de entidades têm uma grande limitação no número de escritas concorrentes, por dois motivos principais: os dados do grupo são armazenados em uma só máquina e o sistema tenta garantir as propriedades ACID nas escritas sobre eles.

Este detalhe provavelmente passaria despercebido por analistas e desenvolvedores inexperientes, mas seu impacto pode ser considerável e percebido somente quando o aplicativo atingir um grande número de usuários.

2.8 *BENCHMARKING* DE BANCOS DE DADOS NOSQL

O YCSB é um aplicativo de *benchmark* que realiza requisições a um banco de dados, com o intuito de medir seu desempenho. De acordo com Cooper et al. (2010), o *Yahoo! Cloud Serving Benchmark* tem o objetivo de facilitar a comparação de desempenho dos sistemas de serviço de dados na nuvem.

Ele faz uso de *workload*s, que são um conjunto de operações que simulam um contexto. Segundo Cooper et al. (2010), “cada *workload* representa uma mistura particular de operações de escrita e leitura, tamanho de dados, distribuição de requisições [...] e pode ser usado para avaliar sistemas em um ponto específico da *performance*”.

Uma série de *workloads*, chamados *core workloads*, acompanham o YCSB por padrão, alguns deles são:

- A - Update heavy: 50% leituras, 50% escritas, pode ser utilizado para simular a gravação de ações recentes do usuário numa sessão, por exemplo;
- B - Read heavy: 95% leituras, 5% escritas, simula a marcação de fotos, por exemplo, pois cada nova marcação é uma escrita, sendo que a maioria das operações são de leitura;
- C - Read only: 100% leituras.

No YCSB, o tempo consumido por uma série de operações é um dos pontos utilizados para medir o desempenho. De acordo com Weiss et al. (2013), “de modo a avaliar o desempenho de certo aspecto de um programa (por exemplo, duas alternativas diferentes de esquema), um desenvolvedor deve executar medições que são adaptadas ao programa a ser testado”.

O algoritmo apresentado por Weiss et al. (2013) primeiro armazena o tempo atual da máquina utilizada, a ser tratado como tempo inicial, em seguida executa as operações a serem medidas e por fim apresenta o tempo total utilizado pelas operações, baseado no tempo atual menos o tempo inicial.

O YCSB testa somente um esquema de dados, o qual é fixo, não deixando a possibilidade de que o usuário do *benchmark* teste suas alternativas de esquemas (COOPER et al., 2010).

3. DESENVOLVIMENTO

3.1 UM TESTE ABRANGENTE PARA BANCOS DE DADOS NOSQL

Existem diversos fatores que podem afetar o desempenho de um banco NoSQL, sendo que o esquema de dados é um fator negligenciado pela maioria dos *benchmark*s atuais. Estre trabalho propõe um teste que engloba as camadas física, de *software* e lógica, tendo como principal diferencial a inclusão do esquema de dados em sua composição.

O objetivo é medir o desempenho de um código fonte já implementado, o qual possui as seguintes características: utiliza um *framework* específico para realizar a comunicação com o banco de dados; realiza uma série de operações no banco da dados e implementa um esquema.

Levando-se em consideração o YCSB e o *benchmark* adaptado proposto por Weiss et al. (2013), propomos um teste que faz uso de alguns de seus conceitos em conjunto, adaptando-os a uma nova arquitetura. O teste aqui apresentado possui, por padrão, os seguintes quatro tipos de *workloads,* cada um representando um contexto de aplicativo:

- A - *Update Heavy*: composto por 50% de leituras e 50% de escritas;
- B - *Read Mostly*: composto por 95% de leituras e 5% de escritas;
- C - *Read Only*: 100% leituras;
- W - *Write Only*: 100% escritas.

3.1.1 Medindo-se o Desempenho do Código

O código a ser testado deve implementar o algoritmo de *benchmark* adaptado proposto por Weiss et al. (2013). O trecho de código onde o tempo inicial é armazenado deve estar antes do trecho de código onde o esquema de dados é implementado e das requisições ao banco de dados.

Após a execução da série de requisições, é calculado o tempo total da execução do trecho em questão. Desta forma, se obtém o tempo total consumido pelo programa

para realizar as operações, possibilitando-se a resposta para algumas perguntas, como as seguintes:

- qual o impacto do *framework* utilizado no meu aplicativo?
- qual a melhor alternativa de esquema de dados devo utilizar?
- a linguagem de programação utilizada afetou o desempenho?
- o esquema de dados implementado é eficiente para o meu aplicativo?

```
#2. Armazena o tempo inicial
initialTime = time.clock()

#3. Realiza as operações de acordo com o esquema recebido por post
if(schema == "1"):

    user = User()
    userKey = user.put()
    article = Article(parent = userKey)
    articleKey = article.put()

    for writeOperations  in range(0, int(writes)):
        comment = Comment(parent = articleKey)
        comment.put()
    for readOperations  in range(0, int(reads)):
        query = Comment.query().get()

elif(schema == "2"):
    #No esquema 2, o número de escritas é dividido por 2 porque a cada
    #iteração, são inseridas 3 entidades
    for writeOperations  in range(0, int(writes)/3):
        user = User()
        userKey = user.put()
        article = Article(parent = userKey)
        article.put()
        comment = Comment(parent = userKey)
        comment.put()
    for readOperations  in range(0, int(reads)):
        query = Comment.query().get()
#calcula tempo final
totalTime = time.clock() - initialTime
```

Figura 3.1 – Trecho do código fonte escrito em Python. Fonte: Autoria própria.

3.1.2 Etapas do Teste

O teste pode ser dividido em três etapas:

- Entrada de parâmetros: antes de dar início ao teste, devem ser recebidas as seguintes variáveis de entrada:
  - número de operações: é o número total de operações a serem realizadas pelo algoritmo;
  - *workload*: o tipo de *workload* a ser submetido ao algoritmo (A, B, C, D ou W);
  - esquema: o esquema de dados a ser implementado pelo algoritmo;
  - *framework*: o *framework* e linguagem de programação a serem utilizados pelo algoritmo.
- Execução do algoritmo a ser testado: nesta etapa, o algoritmo a ser testado é executado, e seu desempenho medido;
- Exibição dos resultados: é exibido ao usuário do teste o tempo total consumido pelo algoritmo.

3.1.3 Arquitetura do Teste

O teste é composto por três partes:

- Aplicativo de entrada de parâmetros: recebe os parâmetros número de operações, *workload*, *framework* e esquema de dados, os quais serão enviados ao aplicativo seguinte;
- Aplicativo de teste: recebe os parâmetros enviados pelo aplicativo e implementa o esquema de dados selecionado, também realiza o número de operações de escrita e leitura, definidos pelo aplicativo anterior;
- Banco de dados: o banco é considerado uma parte distinta, pois cada aplicativo de teste deve utilizar seu próprio banco, esta restrição é necessária para que bancos não afetem os resultados de outros aplicativos de teste.

A Figura 3.1, abaixo, detalha a arquitetura visualmente, incluindo também o papel do usuário e a troca de mensagens entre as diferentes partes do teste.

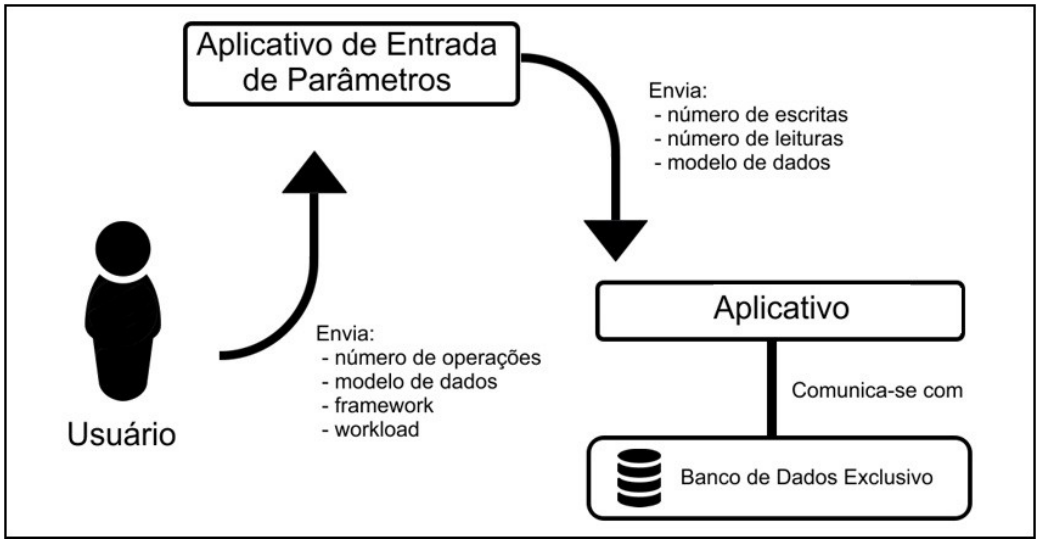

Figura 3.2 – Arquitetura do teste. Fonte: Autoria própria.

É importante levar em consideração que o aplicativo de teste não necessita receber o valor do número total de operações. A partir do número de operações definido pelo usuário e o *workload* selecionado, o aplicativo de entrada pode calcular o número de leituras e escritas, os quais serão enviados ao aplicativo de teste. Exemplo de caso: um usuário definiu 100 operações e *workload* A - *Update Heavy*, o qual é composto por 50% escritas e 50% leituras; o aplicativo de entrada deve calcular o número de escritas e leituras baseando-se nestas variáveis e enviá-los ao aplicativo de teste o qual receberá os valores 50 para leitura e 50 para escrita.

3.1.4 Sobre a Implementação do Esquema de Dados

O esquema de dados a ser testado pelo usuário deve ser implementado pelo aplicativo de teste. Neste trabalho, foram utilizados dois esquemas de dados, a serem detalhados mais adiante; porém o desenvolvedor do teste pode optar por programar um algoritmo dinâmico e flexível, o qual implementa o esquema de dados seguindo uma estrutura definida pelo usuário. Esta possibilidade é mais explorada adiante.

Como mencionado anteriormente, bancos de dados NoSQL não têm suporte a esquemas de dados fixos e pré-definidos, podendo ser de quatro diferentes categorias, entre outras variações. Portanto a maneira como o esquema será implementado varia consideravelmente de acordo com o banco utilizado.

O fato de o esquema de dados ser implementado no código-fonte facilita a possibilidade de se utilizar esquemas de dados dinâmicos no aplicativo de teste.

3.2 Implementando o Teste no Google AppEngine

Implementamos o teste proposto no ambiente de PaaS Google AppEngine (www.appengine.google.com), o qual já inclui todas a camadas a serem testadas. A infraestrutura utilizada é composta de milhares de servidores ao redor do mundo, pertencentes ao Google.

O sistema de banco de dados utilizado foi o Google Datastore. As linguagens e *frameworks* testados foram os seguintes:

- Python com bilbioteca padrão;
- Java com JPA;
- Java com JDO;
- Java com biblioteca padrão.

Optamos por testar dois esquemas de dados implementados previamente, ao invés de esquemas de dados dinâmicos gerados pelo aplicativo de teste baseados em uma estrutura definida pelo usuário.

Os dois esquemas de dados implementados são os esquemas sugeridos por Scherzinger et al.(2013), os quais foram detalhados anteriormente. Neste trabalho, o esquema de dados orientado a usuários é referenciado como esquema 1, e o outro como esquema 2.

Desta forma foram gerados 8 aplicativos, cada um com as seguintes características:

- Python com biblioteca de persistência padrão e esquema de dados 1;
- Python com biblioteca de persistência padrão e esquema de dados 2;
- Java com *framework* JPA e esquema de dados 1;
- Java com *framework* JPA e esquema de dados 2;
- Java com *framework* JDO e esquema de dados 1;
- Java com *framework* JDO e esquema de dados 2;
- Java com biblioteca de persistência padrão e esquema de dados 1;

- Java com biblioteca de persistência padrão e esquema de dados 2;.

3.2.1 Particularidades da Implementação no Google AppEngine

No ambiente utilizado, cada aplicativo está hospedado num domínio com o sufixo *.appspot.com*, sendo que cada aplicativo possui acesso a seu banco de dados particular.

Seguindo o princípio aqui sugerido que diz que cada aplicativo de teste deve utilizar seu próprio banco de dados, foi necessário criar um aplicativo para cada esquema e *framework* utilizado. Por exemplo: o aplicativo que implementa o esquema de dados 1 com a linguagem JAVA e o *framework* JPA será hospedado no domínio *javajpa1.appspot.com*. A Figura 3.2 detalha abaixo a implementação:

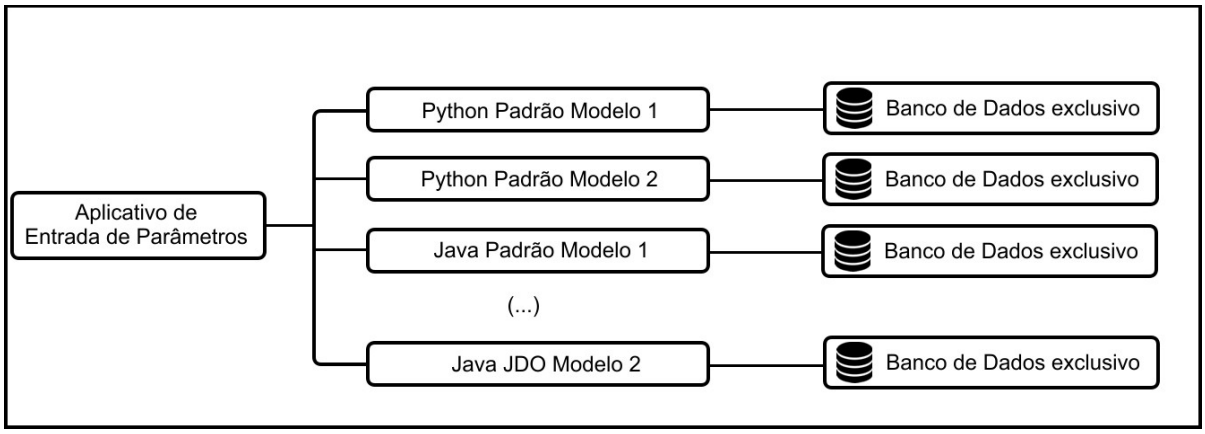

Figura 3.3 – Arquitetura da implementação do teste no Google AppEngine. Fonte: Autoria própria.

Os projetos possuem código aberto e estão disponíveis em *https://github.com/uenpbenchmark*. O Aplicativo de Entrada de Parâmetros, no dia 11/09/2015 estava disponível em *http://uenpnosql.appspot.com*.

3.2.2 Limitações do Google AppEngine

O Google AppEngine impõe uma série de limitações de uso, detalhadas na tabela abaixo. Além destas limitações, existe uma limitação de cerca de 400 operações por segundo, com o intuito de evitar abusos no uso do serviço. Portanto, testes com grande número de operações poderão falhar.

As limitações são diferentes para contas de desenvolvedor que possuem cobrança ativada e desativada. Os testes foram executados com uma conta com a cobrança desativada. Ao se executar 400 operações, os aplicativos de teste frequentemente resultavam em erro, devido a uma proteção do serviço contra abusos.

A tabela a seguir detalhas as limitações:

Tabela 3.1 – Limitações de uso do Google AppEngine.

Recurso	Limite diário padrão	Limite diário com cobrança ativada
Dados armazenados	1 GB*	1 GB grátis, sem limite
Número de índices	200*	200
Operações de escrita	50000	Ilimitado
Operações de leitura	50000	Ilimitado
Pequenas operações	50000	Ilimitado

*Não limite diário, mas limite total.

3.2.3 Resultados de um Teste Comparativo

Os testes a seguir foram executados da seguinte forma: cada *framework*/linguagem foi testado(a) 40 vezes em sequência, sendo que cada *workload* foi executado 10 vezes. Por exemplo: executou-se o aplicativo implementado com Python 40 vezes seguidas, sendo que a cada 10 execuções, trocou-se o *workload*. Realizados no dia 11/08/2015, entre as 17 e 21 horas, horário de Brasília.

A Figura 3.4 exibe abaixo uma comparação entre os esquemas de dados, quando executados em cada linguagem ou *framework*. A Figura 3.5 exibe o desempenho geral de cada linguagem/*framework.* O eixo vertical indica o tempo total consumido pelo teste.

Na Figura 3.4, nota-se que a pior combinação entre esquema de dados e framework foi esquema 2 com JDO. Não houve diferença significativa entre esquemas de dados 1 e 2 quando implementados em Python. Entre os frameworks, o JPA mostrou ser mais eficiente quanto ao desempenho e superior ao java com biblioteca padrão.

Na Figura 3.5 percebe-se que o Python se mostrou significativamente mais rápido que as outras linguagens e *frameworks*, no geral.

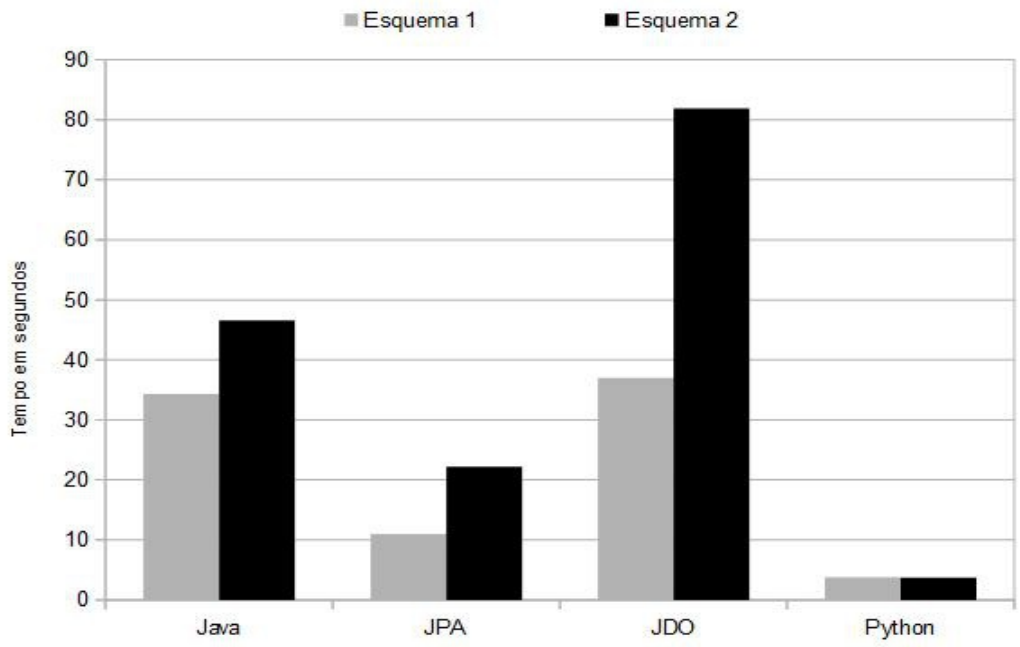

Figura 3.4 – Comparação de desempenho dos esquemas de dados, separados por linguagem ou *framework*. Fonte: Autoria própria.

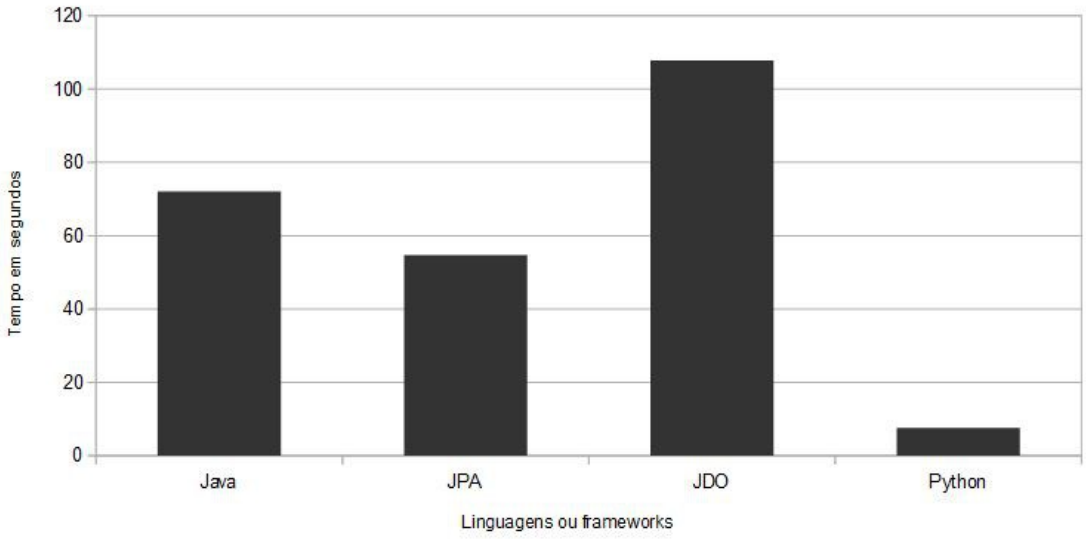

Figura 3.5 – Desempenho geral de cada linguagem ou *framework*. Fonte: Autoria própria.

A Figura 3.6 exibe uma comparação de desempenho das quatro linguagens e *framework* testados, quando separados tipo de *workload*, sem diferenciação entre linguagens e esquemas.

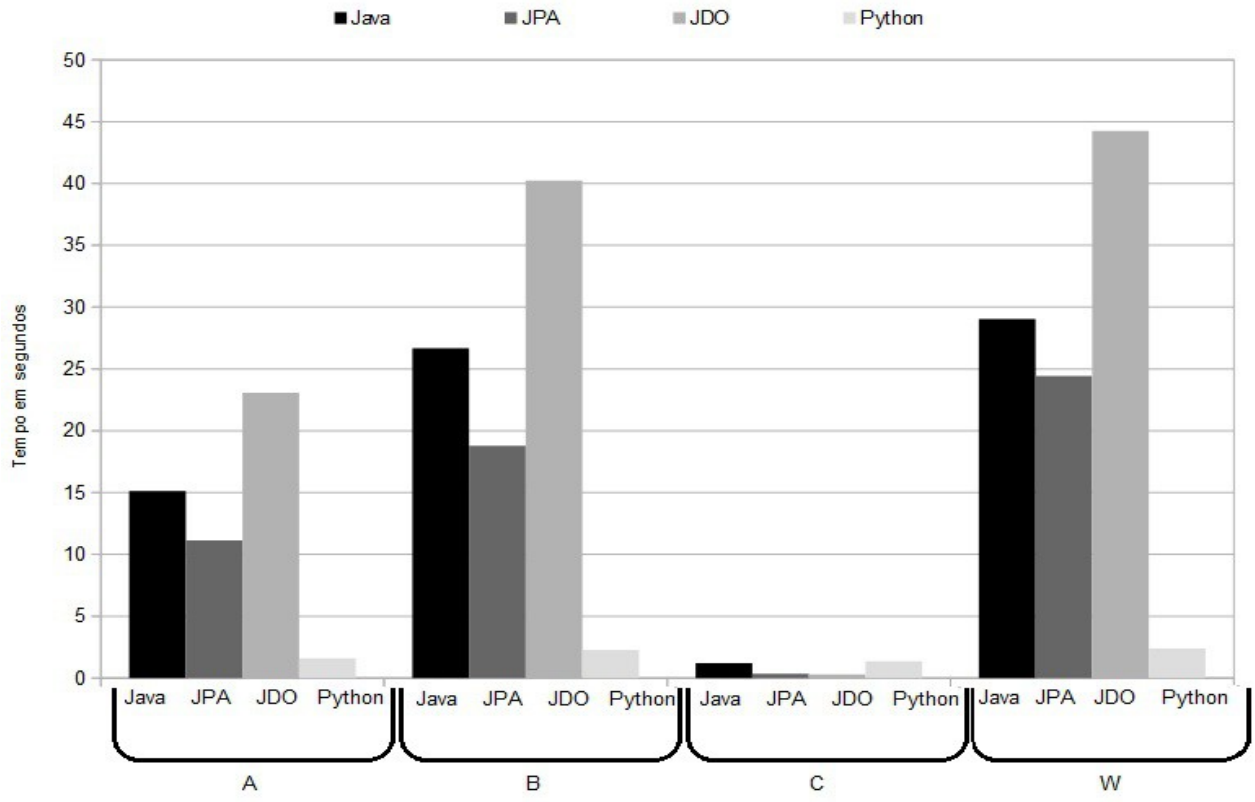

Figura 3.6 – Comparação de desempenho das linguagens e *frameworks*, separadas por tipo de *workload*. Fonte: Autoria própria.

A seguir, a Figura 3.7 exibe uma comparação de desempenho entre os quatro *workloads* testados, divididos por linguagem ou *framework*.

Na Figura 3.6 está claro que o workload C levou muito menos tempo para ser executado, o que é esperado, já que o workload realiza somente operações de leitura. Deste modo, o workload W é o mais lento, pois realiza somente operações de escrita. Mais uma vez o Python mostrou-se mais rápido ao executar todos os workloads, e o Java com biblioteca padrão mais lento que JPA.

O mesmo padrão se repete na Figura 3.7.

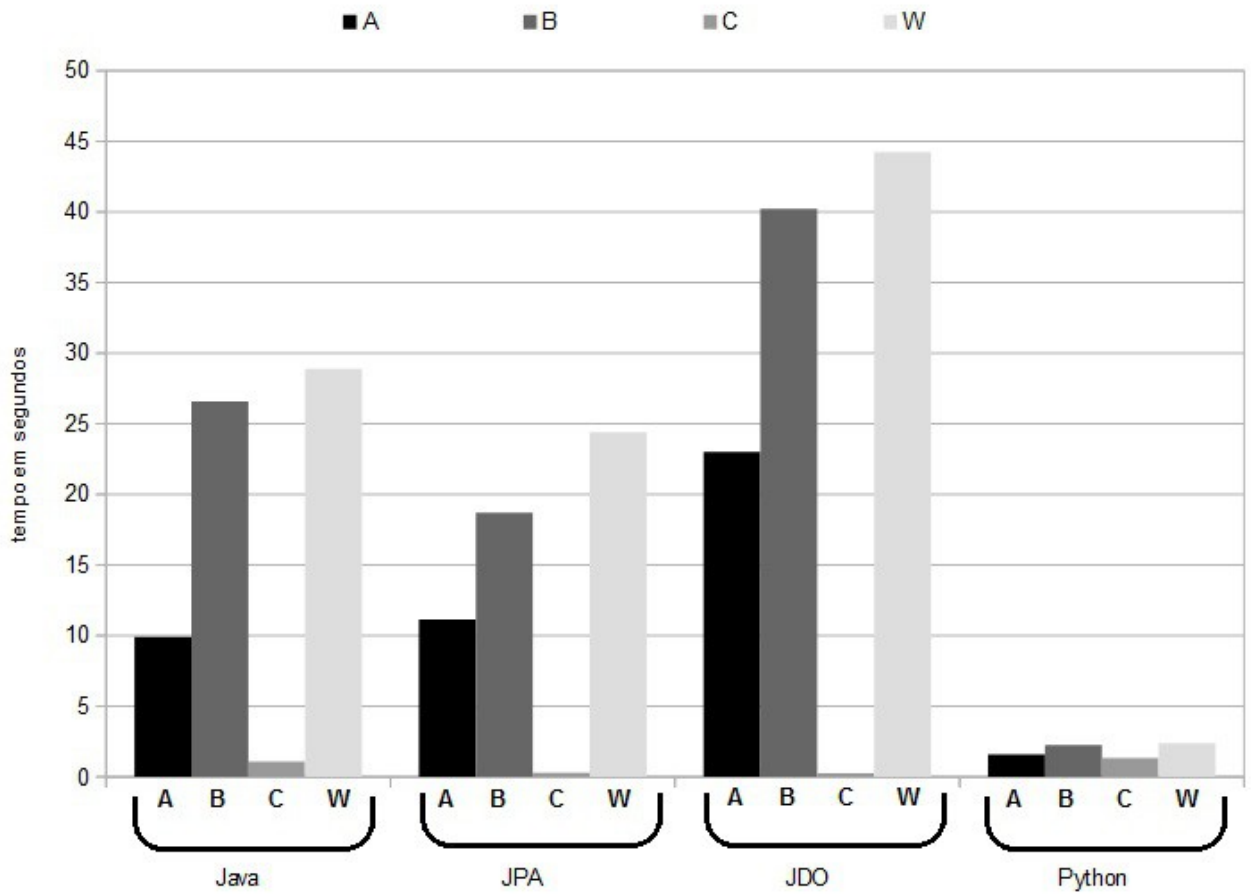

Figura 3.7 – Comparação de desempenho entre os *workloads*, separadas por linguagem ou *framework*. Fonte: Autoria própria.

A Figura 3.8 exibe abaixo uma comparação de desempenho entre os dois esquemas de dados testados, divididos por tipo de *workload*. A Figura 3.9 exibe o desempenho geral de cada esquema testado, levando em conta os *workloads* e linguagens/*frameworks*. O eixo vertical indica o tempo total consumido em segundos.

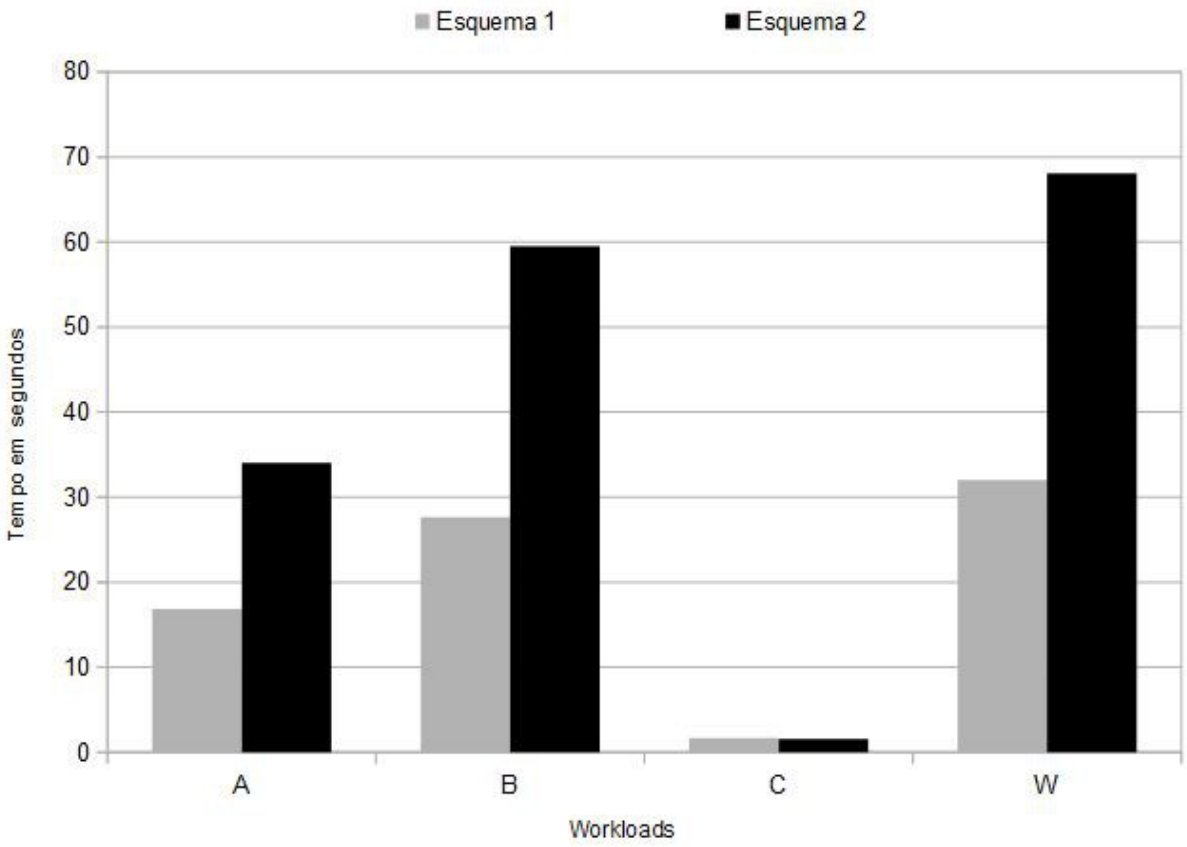

Figura 3.8 – Comparação de desempenho entre os esquemas de dados, separadas por tipo de *workload*. Fonte: Autoria própria.

Na Figura 3.8 percebe-se que o esquema de dados testado teve pouco impacto no desempenho do *workload* C, mas impacto considerável nos outros *workloads*.

Está evidente na Figura 3.9 que o esquema de dados 1 proporcionou um melhor desempenho de modo geral, quando considerados os frameworks e *workloads* testados.

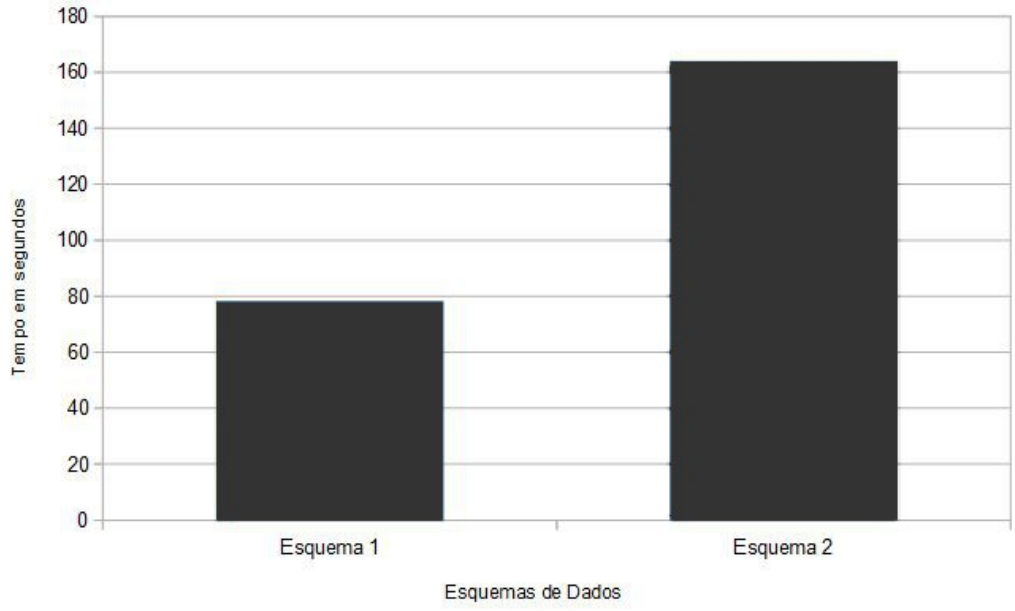

Figura 3.9 – Comparação geral de desempenho entre os esquemas de dados. Fonte: Autoria própria.

3.2.4 Análise dos Resultados do Teste Comparativo e Recomendações

A linguagem Python mostrou-se consideravelmente mais eficiente em relação às outras linguagens testadas. O *framework* JDO mostrou-se como o menos eficiente. Dentre os motivos, podemos destacar a falta de adequação ao Google Datastore. Durante a implementação, notou-se que não é possível implementar uma hierarquia de entidades com o JDO de forma simples, para se alcançar isto, é necessário adaptar a estrutura das classes de modo não convencional e não condizente com o paradigma da programação orientada a objetos.

O JPA mostrou-se menos eficiente que o Python, porém mais eficiente que a biblioteca padrão em Java. Apesar da menor eficiência, atualmente o JPA possui suporte adequado ao Google Datastore, tornando possível estruturar os objetos em hierarquias de forma relativamente simples. Aparentemente, o JPA está na frente do JDO atualmente.

Apesar de ser apenas a terceira mais rápida, a biblioteca padrão de Java permite um desenvolvimento mais rápido de aplicativos, pois não exige classes que expressem o esquema de dados ou entidades a serem persistidas no banco de dados. Um ponto negativo seria a difícil manutenção do código-fonte, uma vez que as estruturas das entidades não estão claras.

Os dois esquemas de dados apresentaram pouco diferença de desempenho quando executados em Python e Java com biblioteca padrão. Porém, o esquemas de dados 2 mostrou-se menos eficiente quando implementado com JPA e JDO.

Esta ineficiência pode ser explicada pelo alto custo de tempo consumido pelo aplicativo, ao implementar a hierarquia no código fonte. É importante lembrar que os *frameworks* JPA e JDO obrigam o uso de transações, as quais são implementadas pelo *framework* e podem ter impacto no desempenho.

Ao desenvolvedor iniciante, algumas recomendações podem ser feitas a partir dos resultados obtidos, como:

- o uso de frameworks é recomendado para equipes, uma vez que eles tornam a manutenção do código mais fácil;
- recomenda-se dar preferência ao *framework* JPA, ao invés de JDO, pois o JPA mostrou-se mais apropriado ao AppEngine e demonstrou melhor desempenho;
- para pequenos projetos, ou seja, aqueles com pequena complexidade, recomenda-se Python;
- quando Python não for uma linguagem viável para um pequeno projeto, e o desempenho não for um fator de grande importância, recomenda-se Java com biblioteca padrão, devido à sua menor complexidade; e
- quando facilidade de manutenção do código e desempenho forem fatores importantes em um projeto, recomenda-se Java com JPA.

Tabela 3.2 – Recomendações ao desenvolvedor iniciante quanto à que lingaugenm/*framework* utilizar.

Desempenho	**Facilidade de manutenção**	**Linguagem/Framework recomendado**
Importante	Não-importante	Python, ou Java com bilbioteca padrão
Importante	Importante	Java JPA
Não-importante	Importante	Java JPA

4. CONCLUSÕES

Este trabalho propôs uma arquitetura de teste de bancos de dados NoSQL. Detalhou uma implementação do teste no Google AppEngine e exibiu resultados obtidos após a realização de uma série de testes realizados nesta implementação.

Os testes realizados mostraram que é possível se obter números que expressem o desempenho de um aplicativo a ser desenvolvido ao se levar em consideração fatores como a infraestrutura, linguagem de programação, banco de dados e principalmente o esquema de dados.

O impacto do esquema de dados nos aplicativos de testes foi considerável, provando que este fator não deve ser negligenciado por desenvolvedores e analistas, especialmente para aplicativos voltados à Web 2.0. Desta forma, este teste abrangente provou ser essencial no processo de tomada de decisões e para a análise de desempenho de bancos de dados NoSQL.

Espera-se que o teste proposto sirva como modelo para futuras implementações e aprimorado. Trabalhos futuros podem implementar a capacidade de se criar um esquema de dados especificado pelo usuário, em tempo de execução, ao contrário da utilização de esquemas fixos, como realizado neste trabalho. Desta forma o teste se torna um *benchmark* flexível de grande utilidade para a comunidade de bancos de dados.

REFERÊNCIAS

BAKER et al. *Megastore: Providing Scalable, Highly Available Storage for Interactive Services*. Proceedings of the Conference on Innovative Data system Research. Asilomar, p. 223-234, 2011.

BREWER, A. *Towards robust distributed systems*. Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing. Portland, p. 7, 2000.

CALDER, B. et al*. Windows Azure Storage: a highly available cloud storage service with strong consistency.* New York*,* SOSP '11 Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, p. 143-157, 2011.

CHANG et al. *Bigtable: A Distributed Storage System for Structured Data*. ACM Transactions on Computer Systems. New York, v. 26, n. 4, 2008.

COOPER et al. *Benchmarking Cloud Serving Systems with YCSB.* Proceedings of the 1st ACM Symposium on Cloud Computing*.* New York, p.143-154, 2010.

GILBERT, S.; NANCY L*. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News. New York, v. 33, n. 2, p. 51-59, 2002.*

HETCH, R.; JABLONSKI, S. et al*. NoSQL evaluation: A use case oriented survey.* Cloud and Service Computing 2011 International Conference*.* Hong Kong, p. 336-341, 2011.

LAKSHMAN, A.; MALIK, P. *Cassandra: a decentralized structured storage system*. ACM SIGOPS Operating Systems Review. New York, v. 44, n. 2, p. 35-40, 2010.

LÓSCIO, B. F.; OLIVEIRA, H. R.; PONTES, J. C. S. *NoSQL no Desenvolvimento de Aplicações Web colaborativas.* VIII Simpósio Brasileiro de Sistemas Colaborativos, 2011*.*

MELL, P.; GRANCE, T*. The NIST definition of cloud computing. National Institute of Standards and Technology, v. 53, n. 6, p. 50, 2009.*

MURUGESAN, S. Understanding Web 2.0. IT Professional. IEEE, v. 9, n. 4, p. 34-41, 2007.

O'REILLY, T. *What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software.* Communications & Strategies, n. 1, p. 17, 2007*.*

SAÚDE, A. V.; MELCHIORI, A. P. P.; RESENDE, A. M. P*. Análise Comparativa de Frameworks de Persistência.* Lavras, 2015*.*

SCHERZINGER et al. On the necessity of model checking NoSQL database schemas when building SaaS applications. Proceedings of the 2013 International Workshop on Testing the Cloud*.* New York, p. 1-6, 2013.

WEISS et al.*Systematic performance evaluation based on tailored benchmark applications.* Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering. New York, p. 411-420, 2013.

LEAVITT, N. *Will NoSQL Databases Live Up to Their Promise?*. Computer, v.43, n.2, 2010.

POKORNY, J. *NoSQL databases: a step to database scalability in web environment*. Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services. New York, p.278 – 283, 2011.