Estudo Comparativo de Bancos de Dados NoSQL

Dinei A. Rockenbach[1], **Nadine Anderle**[1], **Dalvan Griebler**[1,2], **Samuel Souza**[1]

[1] Laboratório de Pesquisas Avançadas para Computação em Nuvem (LARCC)
Faculdade Três de Maio (SETREM) – Três de Maio – RS – Brasil

[2]Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS/PPGCC)
Porto Alegre – RS – Brasil

{dineiar,nadianderle}@gmail.com,dalvan.griebler@acad.pucrs.br,
samuel@samuelsouza.com

Abstract. *NoSQL databases emerged to fill limitations of the relational databases. The many options for each one of the categories, and their distinct characteristics and focus makes this assessment very difficult for decision makers. Most of the time, decisions are taken without the attention and background deserved due to the related complexities. This article aims to compare the relevant characteristics of each database, abstracting the information that bases the market marketing of them. We concluded that although the databases are labeled in a specific category, there is a significant disparity in the functionalities offered by each of them. Also, we observed that new databases are emerging even though there are well-established databases in each one of the categories studied. Finally, it is very challenging to suggest the best database for each category because each scenario has its requirements, which requires a careful analysis where our work can help to simplify this kind of decision.*

Resumo. *Bancos de dados NoSQL surgiram para suprir limitações dos bancos de dados relacionais. A miríade de opções em cada uma das categorias, cada qual com características e foco distintos, torna muito custosa esta avaliação para tomadores de decisões. Na maior parte das vezes, as decisões são tomadas sem a atenção e o embasamento merecidos devido à complexidade relatada. Este artigo tem como objetivo comparar as características de cada banco de dados, abstraindo as informações que embasam o marketing de cada um. Ao final, conclui-se que apesar dos bancos serem rotulados em determinada categoria, existe uma disparidade muito grande nas funcionalidades oferecidas por cada um deles. Além disso, notou-se que novos bancos estão surgindo mesmo havendo líderes bem estabelecidos em cada uma das categorias estudadas. Sendo assim, é difícil indicar o melhor banco de cada categoria, pois é preciso avaliar as necessidade de cada cenário de aplicação, o que requer uma análise cuidadosa, onde este trabalho pode ajudar a simplificar a tomada de decisão.*

1. Introdução

As necessidades de alta disponibilidade e velocidade, bem como o aumento do volume de dados e consumo destes nos sistemas e aplicações atuais, influenciaram no desenvolvimento de novas tecnologias de bancos de dados para o armazenamento de informações. Estas tecnologias vão além dos bancos de dados relacionais, impulsionando o movimento

NoSQL (*Not only SQL*) [Fowler 2015]. Não obstante, há uma consequente explosão no número de sistemas de armazenamento disponíveis, o que dificulta a tomada de decisão quanto à opção que melhor supre as necessidades organizacionais. Escolher o banco dados adequado para as demandas do sistema, não é uma tarefa trivial. Isso requer muito estudo e pesquisa para que as características possam ser comparadas e evidenciadas.

Com o objetivo de solucionar problemas específicos, os bancos NoSQL foram categorizados de acordo com suas características e otimizações. Por exemplo, a Amazon utiliza seu sistema chave-valor (*key-value*) Dynamo [DeCandia et al. 2007] para gerenciar as listas de mais vendidos, carrinhos de compras, preferências do consumidor, gerenciamento de produtos, entre outras aplicações. Existem também sistemas de família de colunas (*column family* ou *columnar*) que foram influenciados pelo Bigtable da Google [Chang et al. 2008]. Enquanto isso, os assim chamados de sistemas de documentos (*document*) resultaram, por exemplo, no MongoDB[1]. Por fim, do chamado banco triplo (*graph database* ou *triple*), tem-se como exemplo o Neo4j[2]. Cada uma destas categorias traz sistemas que cobrem diferentes limitações dos bancos relacionais tradicionais.

Esta pesquisa tem como objetivo fazer um estudo comparativo do estado da arte sobre os bancos de dados NoSQL. Para isso, foi realizado um levantamento das tecnologias que seriam estudadas, procurando mesclar bancos de dados já considerados em outros estudos com bancos não considerados. A fim de analisar as funcionalidades dos bancos, as principais e importantes características foram tabuladas com o propósito de identificar as diferenças e vantagens em cada uma das suas categorias.

Tendo em vista que o foco das quatro categorias dos bancos NoSQL já foi explorado na academia em [Moniruzzaman and Hossain 2013, Pokorny 2013, Leavitt 2010], dentre outros autores, entende-se que identificar a aplicação de cada uma delas (chave-valor, família de colunas, documentos e triplo) não é mais um problema sem resposta. Porém, permanece a carência de um estudo do estado da arte, sobre bancos de dados NoSQL, que compare as funcionalidades dos bancos dentro de cada uma das categorias de forma isolada, considerando bancos já abordados por outros pesquisadores e complementando com bancos de dados emergentes. Desta forma, adicionando evidências ao conhecimento existente em cada uma das categorias e aprimorando as possibilidades de comparação entre eles.

Este artigo está organizado em 5 seções, incluindo esta seção introdutória. Na Seção 2 estão os trabalhos relacionados. A Seção 3 traz um embasamento sobre as categorias de bancos estudados e o estudo comparativo dos bancos de dados pesquisados encontra-se na Seção 4. Por fim, na Seção 5 estão as conclusões e propostas para trabalhos futuros.

2. Trabalhos Relacionados

Esta seção apresenta as pesquisas relacionadas que antecederam e basearam este estudo. Na primeira coluna da Tabela 1 é apresentada a referência do trabalho e na segunda coluna as tecnologias abordadas. Na sequência, está exposto o foco do trabalho dentro do espectro de bancos de dados NoSQL e por fim algumas observações.

Tanto [Hecht and Jablonski 2011] quanto [Han et al. 2011] se propõem a fazer

[1]https://www.mongodb.com
[2]https://neo4j.com

Tabela 1. Trabalhos relacionados

Estudo	Tecnologias	Foco	Observações
[Han et al. 2011]	Redis, Tokyo, Flare	NoSQL key-value, columnar e document	Não oferece comparativo.
[Hecht and Jablonski 2011]	Voldemort, Redis, Membase	NoSQL	Avalia API, concorrência, replicação e consistência.
[Deka 2014]	Hypertable, Voldemort, Dynomite, Redis, Dynamo	NoSQL	Apenas características mercadológicas.
[Zhang et al. 2015]	MemepiC, RAMCloud, Redis, Memcached, MemC3, TxCache	IMDB	Avalia o projeto, modelo, índices, concorrência, e mais.
[Abramova et al. 2014]	Cassandra, Hbase, MongoDB, OrientDB, Redis	NoSQL	Desempenho de leitura, modificação e inserção. Não considera o ambiente.
[Gessert et al. 2017]	Redis, Riak, Cassandra, HBase, MongoDB	NoSQL key-value, columnar e document	Características de projeto, mas pouco elaboradas.
[Moniruzzaman and Hossain 2013]	Redis, Riak, MongoDB, CouchDB, DynamoDB, HBase, Cassandra, Accumulo, Neo4j	Categorias NoSQL	Foco não envolve análise detalhada de bancos individuais.
Este estudo	Redis, Memcached, Voldemort, Aerospike, Hazelcast, Riak KV, BigTable, HBase, Hypertable, Accumulo, MongoDB, CouchDB, Couchbase, MarkLogic, OrientDB, Neo4j, JanusGraph, Graph Engine, Bitsy	NoSQL	Características mercadológicas, de projeto e de manutenção.

um estudo e avaliação dos bancos de dados NoSQL, tendo o mesmo objetivo principal: prover informações para auxiliar na escolha do banco NoSQL que melhor atende às necessidades. No ponto de intersecção entre este trabalho e os citados, o trabalho [Hecht and Jablonski 2011] inclui os bancos Project Voldemort, Redis e Membase, e [Han et al. 2011] avalia Redis, Tokyo Cabinet-Tokyo Tyrant e Flare. Quanto a descrição das características de cada tecnologia [Han et al. 2011] descreve brevemente aquelas que o banco contempla, porém não compara as tecnologias abordadas entre si. Já [Hecht and Jablonski 2011] apresenta uma análise mais profunda permitindo a comparação das mesmas. Diferentemente, esse trabalho abrange um número maior de tecnologias e também mais características que os demais conforme pode ser observado na tabela 1.

Em [Deka 2014] é apresentada uma visão geral de quinze sistemas NoSQL, tais como, Cassandra, BigTable, Pnuts, Redis, entre outros. Nota-se a falta, porém, de uma visão comparativa mais clara sobre aspectos de garantias de durabilidade, disponibilidade, protocolos suportados, e outras informações que podem vir a influenciar significativamente na escolha de um banco de dados. Então esses aspectos não considerados por [Deka 2014], foram inclusos neste estudo com o intuito de agregar mais conhecimento e conteúdo.

Já [Zhang et al. 2015], traz uma visão bem estruturada dos objetivos que nortearam o projeto de cada um dos sistemas descritos na pesquisa, o qual foca em sistemas com gerenciamento e processamento de dados em memória. Dentre os sistemas estudados, os representantes NoSQL são MemepiC, RAMCloud, Redis, Memcached, MemC3 e TxCache. Os autores descrevem no artigo as cargas de dados mais adequadas aos sistemas, a estratégia para construção de índices, o controle de concorrência, tolerância a falhas, tratamentos para conjuntos de dados maiores do que a memória disponível e o suporte a consultas personalizadas em baixo nível (como *stored procedures* e *scripts* em linguagem nativa), porém com pouca abordagem de alto nível que auxilie na interpretação dos resultados e consequentemente na escolha de um sistema em favor de outro.

O artigo [Gessert et al. 2017] apresenta uma comparação semelhante à que foi realizada para esta pesquisa, além de técnicas e requisitos funcionais e não funcionais atendidos pelos bancos. Porém, ainda que os teoremas CAP e PACELC e as técnicas estejam bem definidas, pouco se fala sobre a implementação destas técnicas em cada um dos bancos apresentados, ou seja, não está claramente descrito quais das garantias que cada um dos bancos oferece. Enquanto isso, essa pesquisa apresenta as resguardas do teorema CAP que os sistemas estudados disponibilizam.

Uma vez que o foco de [Moniruzzaman and Hossain 2013] é categorizar os bancos de dados, o trabalho não consegue elaborar detalhes sobre as funcionalidades oferecidas pelos bancos, ainda que estas estejam bem sumarizadas em uma tabela. Muito devido a isso, também falta diversidade no comparativo, onde figuram apenas os principais representantes de cada categoria. Ao contrário disto, nosso estudo categoriza cada uma das tecnologias abordadas de acordo com os quatro modelos de NoSQL, e estrutura os resultados em tabelas com o objetivo de facilitar a comparação dos bancos.

3. Bancos de dados NoSQL

De acordo com [Fowler 2015], os bancos de dados NoSQL são aqueles bancos que não requerem um rigoroso esquema para os registros, que possam ser utilizados de forma distribuída em *hardware* comum, e que não utilizem o modelo matemático dos bancos de dados relacionais. Segundo [Kabakus and Kara 2017], bancos de dados relacionais (RDBMS) se baseiam no modelo ACID (*Atomicity, Consistency, Isolation, Durability*) para garantir a consistência e manter a integridade dos dados, enquanto que os bancos NoSQL partem do princípio BASE (*Basically Available, Soft-state, Eventually consistent*) para atingir melhor desempenho, disponibilidade e escalabilidade. Porém, vários bancos NoSQL têm adicionado suporte ao modelo ACID nos últimos anos [Fowler 2015].

Os bancos de dados NoSQL podem possuir armazenamento primário tanto na memória RAM quanto no modelo tradicional em discos. Ambos modelos de armazenamento possuem suas vantagens e desvantagens, e podem ser utilizados em complemento um ao outro.

Os bancos de dados em memória ou *In-Memory DataBases* (IMDB), como seu próprio nome sugere, são aqueles bancos de dados que utilizam a memória principal do computador ou memória RAM como dispositivo primário de armazenamento de dados [Lake and Crowther 2013]. O desempenho destes sistemas de armazenamento é significantemente melhor do que sistemas tradicionais devido ao uso da memória volátil para o mapeamento de registros [Abramova et al. 2014].

O principal motivo para o uso de um sistema de armazenamento de dados em memória, tem relação com o fato de que a gravação e leitura de dados em discos rígidos (dispositivo principal de armazenamento dos bancos de dados tradicionais) é milhares de vezes mais lenta do que a mesma operação utilizando a memória do computador [Lake and Crowther 2013]. Os bancos de dados em memória, neste caso, focam na redução do acesso a disco com o objetivo de melhorar o desempenho de sistemas que os utilizem.

Quanto à sua classificação, os bancos de dados NoSQL podem ser categorizados em quatro classes conforme suas otimizações: chave-valor (*key-value*), orietados a documentos (*document*), família de colunas (*column family* ou *columnar*) e banco triplo (*graph*

database ou *triple*). Cada uma dessas classes possui sistemas que atendem a diferentes limitações dos bancos relacionais tradicionais e se complementam entre si [Moniruzzaman and Hossain 2013]. As próximas subseções trazem um resumo sobre cada uma das categorias.

3.1. Bancos de dados chave-valor

É possível classificar como do tipo chave-valor, aqueles bancos de dados cuja a informação armazenada possui a sua respectiva chave. A Amazon utiliza seu sistema *key-value* Dynamo [DeCandia et al. 2007] para gerenciar funcionalidades tais como: listas de mais vendidos, carrinhos de compras, preferências do consumidor, gerenciamento de produtos, entre outras aplicações. A maioria dos bancos de dados que possuem armazenamento em memória RAM são da categoria chave-valor especialmente pela simplicidade de sua estrutura de armazenamento dos dados.

Os bancos de dados NoSQL da categoria chave-valor são os representantes com as estruturas mais simples dentre os existentes [Pokorny 2013] e possibilita a visualização da base de dados como uma tabela *hash*. No entanto, cabe ressaltar que eles possuem um amplo espectro de casos de uso, sendo largamente adotados, tanto como armazenamento primário, quanto para auxiliar outros sistemas de armazenamento [Carlson 2013].

Neste modelo o armazenamento dos dados é a combinação de um conjunto de chaves, e estas estão ligadas a um valor, *string* ou binário. Na Figura 1 é possível visualizar exemplos de dois comandos básicos em um banco de dados chave-valor.

```
> SET mykey "Hello"
"OK"
> GET mykey
"Hello"
```

Figura 1. Exemplos dos comandos GET e SET

Esse modelo pode ser considerado de fácil integração, possibilitando assim que os dados sejam rapidamente acessados através da sua chave, contribuindo para aumentar a disponibilidade de acesso as informações [Fowler 2015]. A manipulação dos dados de modo geral é muito simples, geralmente sendo baseada em comandos como o *get*() e o *set*(), os quais tem por função obter e entrar valores. Uma desvantagem deste modelo é que o mesmo não permite a recuperação de objetos através de consultas mais complexas, como pesquisas com *join*, por exemplo.

3.2. Bancos de dados de famílias de colunas

Os bancos de dados de família de colunas (*column family* ou *columnar*), possuem a sua estrutura similar à estrutura tradicional de tabelas dos bancos de dados relacionais, porém, otimizados de modo que as informações armazenadas estão organizadas em colunas ao invés de em linhas. Os bancos classificados nesse modelo foram influenciados pelo BigTable da Google [Chang et al. 2008], e alguns exemplos dessa categoria são o Cassandra [Apache Software Foundation 2008a] e HBase [Apache Software Foundation 2008b]. O Cassandra é utilizado pelo eBay para diversas funcionalidades, tais como: armazenar as

notificações de usuários, gerir os itens das páginas, indicação de favoritos entre outras aplicações [Bradberry and Lubow 2013].

Embora o modelo de colunas compartilhe o conceito de armazenamento dos sistemas relacionais, a diferença é o modo em que os dados não ficam armazenados em tabelas, mas sim em arquiteturas que estão distribuídas massivamente. No armazenamento em colunas, cada chave está associada a um ou mais atributos (colunas) e as informações ficam guardadas de tal modo que as informações possam ser agregadas rapidamente com menor atividade de E/S [Nayak et al. 2013].

Os bancos colunares encorajam o uso da desnormalização através de duplicidade dos dados. Com isso é possível obter um ganho na velocidade de leitura, pois não são utilizados relacionamentos que demandam um trabalho de reconstituição dos dados [Fowler 2015], porém uma desvantagem é que com isso há perda na velocidade de alteração dos dados [Nayak et al. 2013].

3.3. Bancos de dados orientados a documentos

Os bancos orientados a documentos (*documents*) resultaram, por exemplo, no MongoDB [MongoDB 2009] e CouchDB [CouchDB 2005], onde as informações são armazenadas sobre uma estrutura de árvore, em formatos como XML ou JSON. O governo de Chicago utiliza uma plataforma para gerenciar operações inteligentes construída em cima do MongoDB denominada *WindyGrid* [Goldsmith and Crawford 2014]. Essa ferramenta foi projetada para trabalhar com o grande volume dados gerados pela cidade, onde através de um mapa de Chicago, é possível visualizar informações como chamadas para os serviços 911 e 311, trânsito, informações de construções, *tweets* públicos e outros dados críticos [Thornton 2013].

Os bancos de documentos são semelhantes aos bancos de chave-valor, porém neste caso o valor é estruturado e possui hierarquia. Diferentemente dos bancos tradicionais, estes não possuem um esquema ou estrutura pré-definida [Fowler 2015].

É bastante comum a utilização dos formatos JSON e XML para representar os documentos nestes tipos de bancos de dados [Pokorny 2013] e o suporte ao processamento e análise de dados é bastante variado dentre os representantes da categoria. Uma característica interessante é o método *append-only*, que é utilizado em alguns bancos de documentos para armazenar os dados e oferece operações de escrita em tempo constante.

3.4. Bancos de dados de grafos ou triplos

Os bancos de dados de grafos ou triplos (*graph database* ou *triple*) armazenam as informações compostas por três elementos: sujeito, propriedade ou relacionamento e valor [Fowler 2015, Kabakus and Kara 2017]. Deste modelo é possível citar como exemplo o Neo4j [Neo4j 2007] e o JanusGraph [JanusGraph 2017].

Os bancos triplos possuem foco na criação de relacionamentos entre os dados. Porém, diferentemente dos bancos relacionais tradicionais, os dados não são tabulares, e sim armazenados em registros triplos compostos por sujeito, predicado e objeto. Outra nomenclatura adotada para estes bancos é a sigla RDF ou *Resource Description Framework*, onde os dados são compostos por recurso, propriedade e indicação ou valor.

Com estes bancos é possível construir redes complexas de relacionamentos e revelar novos dados através da inferência. Bancos triplos com suporte a consultas complexas,

tais como análise de caminhos e distâncias entre dois nós, são conhecidos como bancos de grafos [Fowler 2015].

O projeto Apache TinkerPop [Apache TinkerPop 2008] tem buscado padronizar a camada de comunicação entre aplicações e bancos de grafos, através da oferta de um *framework* que inclui uma linguagem de navegação em grafos chamada Gremlin.

4. Estudo Comparativo de Bancos de Dados NoSQL

Nesta seção estão apresentados os estudos comparativos entre os bancos de dados NoSQL das categorias apresentadas anteriormente. O estudo englobou características mercadológicas, do projeto e de manutenção de cada um dos bancos de dados.

Nas características mercadológicas são explorados aspectos sem relação direta com funcionalidades ou o funcionamento do banco, tais como o ano em que a primeira versão do sistema foi lançada, os licenciamentos sob os quais o software é distribuído, a linguagem na qual o sistema foi desenvolvido, os sistemas operacionais suportados, as linguagens nas quais são oferecidos clientes para comunicação e os protocolos de comunicação suportados. A partir destes dados os interessados podem avaliar a maturidade do sistema, se a licença está alinhada com as necessidades empresariais, e se a infraestrutura disponível e o esforço de implementação estão dentro do esperado.

Nas características do projeto são descritas definições decididas no projeto do sistema, tais como as opções para escalabilidade horizontal (ou clusterização), a classificação do sistema segundo o teorema CAP (explicado na Seção 4.1), como é feito o controle de concorrência, o suporte à transações ACID, as opções de persistência dos dados em disco, o suporte a dados complexos e as opções para autenticação do cliente. Analisando-se estas características é possível avaliar se as funcionalidades e características do banco de dados atendem às demandas e características referentes aos dados que se pretende armazenar nos mesmos.

Nas características de manutenção são explanados aspectos que tem relação direta com a manutenção e suporte ao sistema, tais como a existência de interfaces de gerenciamento, ferramentas de monitoramento e benchmarks embutidos (que são disponibilizados junto com o sistema e cujo principal objetivo é avaliar o desempenho do sistema em determinada infraestrutura) para os sistemas estudados. É importante notar que foram avaliadas apenas as ferramentas oficiais dos desenvolvedores dos sistemas, portanto muitas destas ferramentas, ainda que não estejam declaradas aqui, já foram desenvolvidas pela comunidade e estão disponíveis. Estas informações são particularmente interessantes para avaliar o esforço de manutenção que será despendido após a implantação do sistema.

4.1. Teorema CAP

O teorema CAP (*Consistency*, *Availability* e *Partition tolerance*) foi proposto por Eric Brewer em [Brewer 2000] e verificado em [Gilbert and Lynch 2002], desde então tem sido largamente aplicado na academia. No contexto do teorema, *Consistency* é sinônimo de linearizabilidade ([Herlihy and Wing 1990]), portanto um sistema é considerado consistente se as operações distribuídas possam ser vistas como se estivessem executando em um único nó, diferindo do conceito de Consistência como definida na sigla ACID. A definição de *Availability* impõe que cada requisição recebida por um nó que não esteja em estado de falha deva resultar em uma resposta (que não seja um erro) [Gilbert and

Lynch 2002], e [Kleppmann 2015] ressalta que qualquer nó do cluster deve ser capaz de responder à requisição de forma independente de outros nós. Por fim, *Partition tolerance* afirma que redes assíncronas podem perder pacotes, e que uma rede está particionada quanto todas as mensagens entre dois componentes de rede são perdidas. [Kleppmann 2015] critica que esta definição ignora latência e tempos de resposta.

O teorema afirma que na existência de uma falha de comunicação (*partition*) cada nó de um sistema distribuído deve escolher entre responder requisições, mantendo a disponibilidade (*availability*) e assumindo o risco de não retornar os dados mais atuais, ou rejeitar requisições para garantir a consistência dos dados (*consistency*). Sistemas classificados como AP priorizam a disponibilidade, enquanto que sistemas classificados como CP priorizam a consistência. O teorema tem sido alvo de muitas críticas e Brewer explora algumas de suas limitações em [Brewer 2012], enquanto Abadi propõe o teorema PACELC como alternativa em [Abadi 2012].

4.2. Bancos de dados chave-valor

Esta seção apresenta uma comparação entre as características mercadológicas (Tabela 2), de projeto (Tabela 3) e de manutenção (Tabela 4) dos bancos chave-valor.

Vale ressaltar que o Redis não suporta oficialmente Windows, mas uma versão para Windows x64 é mantida pela equipe da MS Open Tech (Microsoft Open Technologies). Quanto ao Voldemort e ao Hazelcast, como os mesmos rodam na JVM (Java Virtual Machine), podem-se considerar os sistemas operacionais suportados por esta tecnologia. Tanto Aerospike quanto Riak KV oferecem pacotes para sistemas baseados nas distribuições Linux Red Hat, Debian e Ubuntu. O Aerospike ainda oferece sua execução no OS X e Windows através de máquinas virtuais.

Tabela 2. Características mercadológicas

	Redis	Memcached	Voldemort	Aerospike	Hazelcast	Riak KV
Lançamento	2009	2003	2009	2012	2009	2009
Licenciamento	BSD-3 e comercial	BSD-3	Apache 2	AGPL e comercial	Apache 2 e comercial	Apache 2 e comercial
Desenvolvido	C	C	Java	C	Java	Erlang
SO Suporte	Linux, BSD, OSX e Windows	Debian/Ubuntu e Windows	JVM	Linux, OS X e Windows	JVM	Linux
Clientes	48 linguagens	Não existe listagem oficial	4 linguagens	12 linguagens	6 linguagens	21 linguagens
Protocolos	Próprio (RESP)	Próprio	HTTP, Socket, NIO	Próprio e JDBC	Próprio e Memcached	API HTTP e próprio

Quanto ao item linguagens com cliente, é importante notar que foram considerados apenas as linguagens e clientes listados no site oficial de cada sistema e que o site do Memcached não oferece uma listagem oficial das linguagens suportadas. Nota-se também que as linguagens C++, Java e Python são as únicas para as quais todos os sistemas possuem clientes. Os protocolos de comunicação são o meio de comunicação entre o cliente e o servidor, porém, na maioria dos casos a comunicação é feita através de um dos clientes já construídos e a empresa não precisa se preocupar com o protocolo utilizado pelo cliente.

Na Tabela 3 é possível avaliar: as opções para escalabilidade horizontal (ou *clusterização*), a classificação do sistema segundo o teorema CAP [Brewer 2000], como é

feito o controle de concorrência, o suporte à transações ACID, as opções de persistência dos dados em disco, o suporte a dados complexos e as opções para autenticação do cliente. Analisando a Tabela 3 é possível avaliar se as funcionalidades e características do banco de dados atendem às demandas e características referentes aos dados que se pretende armazenar nos mesmos.

Quanto à escalabilidade, todos os bancos exceto o Memcached incluem suporte a *sharding* e replicação, sendo que a maioria (a exemplo do Dynamo [DeCandia et al. 2007]) oferecem fator de replicação configurável para leituras e escritas. O Redis utiliza o modelo *master/slave*, comumente utilizado nas bases de dados relacionais tradicionais.

Quanto à classificação do Redis no espectro do teorema CAP, é importante mencionar que o ele não atende todos os requisitos de um sistema CP de [Brewer 2000], por usar replicação assíncrona entre os nós do *cluster*. O teorema CAP também não se aplica ao Memcached pelo fato de ele não suportar a criação de *clusters* e, portanto, não haver comunicação entre nós. O Riak KV possui uma configuração onde é possível definir qual dos atributos devem ser preservados (disponibilidade ou consistência).

Tabela 3. Características do projeto

	Redis	Memcached	Voldemort	Aerospike	Hazelcast	Riak KV
Escalabilidade horizontal	Mestre - Escravo	Não	Fator de Replicação	Fator de Replicação	Fator de Replicação	Fator de Replicação
Teorema CAP	CP	N/A	AP	AP	AP	Config.
Controle Concorrência	Single-thread	Mutex lock	MVCC	Test-and-set	Multi-single-thread	MVCC
Transações	Parcial	Não	Não	Parcial	Sim	Não
Persistência em disco	RDB e AOF	Não	Config.	Assínc.	Banco auxiliar	Banco auxiliar
Suporte a dados complexos	Sim	Não	Sim	Sim	Sim	Sim
Autenticação	Simples	SASL	Kerberos	Somente comercial	Simples, SSL, Kerberos,IP	Sim, e autorização

O controle de concorrência é implementado de diferentes maneiras. No Redis a execução é *single-thread* e as requisições são processadas de forma assíncrona internamente [Zhang et al. 2015], sendo que apenas um *master* responde por uma determinada chave, portanto não há concorrência. O Memcached utiliza *mutex* (*mutual exclusive*) *lock*. Tanto Voldemort quanto Riak KV seguem a implementação do Dynamo [DeCandia et al. 2007] e utilizam *vector clocks*, uma implementação do versionamento baseado em locking otimista conhecida por MVCC (*Multi Version Concurrency Control*). O Aerospike utiliza o método conhecido como *test-and-set* ou *check-and-set* (CAS), uma operação atômica implementada a baixo nível que escreve em um local de memória e retorna o valor antigo. No Hazelcast, é criada uma *thread* para atender cada uma das partições internas de dados, portanto ainda que ele seja *multi-thread*, uma chave específica está num contexto *single-thread* e, portanto, não há concorrência.

Quanto às transações, há um nível bastante variado de suporte oferecido pelos sistemas. Ainda que o Redis tenha suporte básico a transações, estas não possuem opção de rollback e a durabilidade da mesma depende da persistência em disco. Memcached, Voldemort e Riak KV declaram não suportarem transações, enquanto que o Aerospike suporta transações que envolvam uma única chave ou que sejam somente leitura, no caso de envolverem múltiplas chaves. O Hazelcast se destaca sendo o único a oferecer transações ACID completas.

A persistência em disco é oferecida por todos os bancos, exceto o memcached. No Redis são oferecidas duas formas complementares: *snapshot* (RDB), onde todos os dados na memória são gravados em disco, e *append-only file* (AOF), onde cada operação é gravada em um arquivo de log e o arquivo é reescrito quando chega em um tamanho pré-determinado. O Voldemort oferece opções para configurar a persistência como síncrona (*write through*, onde a operação é persistida antes do retorno ao cliente) ou assíncrona (*write behind*, onde o cliente recebe a confirmação e posteriormente a operação é persistida), enquanto que o Aerospike faz a persistência de forma assíncrona. Vale mencionar que o Aerospike tem melhorias focadas no uso de SSD como dispositivo de armazenamento permanente. Tanto Hazelcast como Riak KV oferecem persistência através do acoplamento de um banco de dados auxiliar e a assincronicidade é configurável.

Referente ao suporte a dados complexos, vale notar que todos os sistemas, exceto o Memcached, suportam chaves do tipo lista e hashtable (ainda que com nomes diferentes). Hazelcast e Voldemort se baseiam fortemente nas classes do Java, Redis e Riak KV ainda oferecem suporte ao tipo HyperLogLogs, enquanto Redis e Aerospike oferecem suporte a tipagem ou comandos relativos a georreferenciamento.

Finalizando, enquanto que Redis oferece autenticação simples através de credenciais pré-configuradas, o Memcached oferece autenticação através do protocolo SASL (*Simple Authentication and Security Layer*). O Voldemort é integrado ao protocolo Kerberos. O Aerospike oferece autenticação apenas em sua versão com licenciamento comercial. O Hazelcast oferece todos os protocolos supracitados e o SSL. Por último, o Riak KV oferece um sistema próprio de usuários e grupos, com autenticação e autorização baseada em diversos mecanismos, incluindo senhas e certificados digitais.

Na Tabela 4 está descrita a existência de interfaces de gerenciamento, ferramentas de monitoramento e benchmarks para os sistemas estudados. É importante notar que foram avaliadas apenas as ferramentas oficiais dos desenvolvedores dos sistemas. Portanto, muitas destas ferramentas, ainda que não estejam declaradas aqui, já foram desenvolvidas pela comunidade e estão disponíveis. Estas informações são particularmente interessantes para avaliar o esforço de manutenção que será despendido após a implantação do sistema.

Tabela 4. Características de manutenção

	Redis	Memcached	Voldemort	Aerospike	Hazelcast	Riak KV
Interface de Gerenciamento	Não	Não	Básica	À parte	Comercial	Sim
Ferramentas de Monitoramento	'INFO'	'stats'	JMX	'asadm'	JMX	'stats'
Benchmark embutido	Sim	Não	Sim	Sim	Não	Sim

Quanto às interfaces de gerenciamento oferecidas pelos sistemas avaliados, o Redis e o Memcached são os únicos que não as oferecem nativamente (ainda que existam opções na comunidade) e o Voldemort oferece uma interface básica à parte escrita em Ruby, que está sem manutenção. O Aerospike oferece uma interface que deve ser instalada à parte. No Hazelcast, esta funcionalidade está disponível apenas na versão comercial. O Riak KV é o único onde a interface de gerenciamento já está integrada ao código principal do programa, não requerendo nenhuma instalação extra.

A respeito de ferramentas de monitoramento, o Redis oferece comandos como

INFO, *MEMORY* e *LATENCY*, enquanto que o Memcached oferece o comando *stats* e o Aerospike oferece o comando *asadm*. O Voldemort possui uma interface completa de monitoramento exposta através de *Java Management Extensions* (JMX). Esta é a mesma estratégia utilizada pelo Hazelcast. O Riak KV oferece os comandos *stat* e *stats* em sua interface de linha de comando (CLI) *riak-admin* e a URL */stats* em sua API HTTP.

No item *benchmark* embutido foi avaliado se são disponibilizados *benchmarks* junto com o sistema, cujo principal objetivo é avaliar o desempenho do sistema em determinada infraestrutura. Enquanto que Memcached e Hazelcast não oferecem ferramentas próprias para realização de *benchmark*, no Redis existe a ferramenta *redis-benchmark* e o Voldemort oferece a *voldemort-performance-tool*. No Aerospike os *benchmarks* estão nos clientes disponibilizados e no Riak KV o nome dado ao *benchmark* é *Basho Bench*.

Após comparar as características dos bancos de dados chave-valor, Redis, Memcached, Voldemort, Aerospike, Hazelcast e Riak KV, é fácil entender o motivo pelo qual o Memcached não é considerado um banco de dados, pois seu foco destoa bastante de seus semelhantes. É possível perceber também que, mesmo que o Redis tenha oferecido suporte à clusterização em suas versões mais recentes, os outros sistemas ainda estão à frente quando o assunto é funcionalidades para execução distribuída. É possível perceber também como Voldemort e Hazelcast se utilizam do ecossistema Java para prover funcionalidades interessantes e como o paper do Dynamo [DeCandia et al. 2007] influencia principalmente Voldemort e Riak KV.

4.3. Bancos de dados de famílias de colunas

Esta seção apresenta uma comparação entre as características mercadológicas (Tabela 5), de projeto (Tabela 6) e de manutenção (Tabela 7) dos bancos de famílias de colunas ou columnar.

Ainda que o Cassandra seja o representante mais famoso desta categoria, o Bigtable [Chang et al. 2008] é considerado o grande influenciador no modelo de dados dos bancos de famílias de colunas. Enquanto que o Cassandra trabalha com uma arquitetura distribuída semelhante ao Dynamo [DeCandia et al. 2007] e um modelo de dados semelhante ao Bigtable, o HBase pode ser considerado uma implementação do Bigtable baseado no Hadoop e o Accumulo é mais integrado com o ecossistema da Apache, fazendo uso de Hadoop, ZooKepper e Thrift.

Tabela 5. Características mercadológicas

	Bigtable	HBase	Cassandra	Accumulo
Lançamento	2005	2008	2008	2008
Licenciamento	Proprietária	Apache 2	Apache 2	Apache 2
Desenvolvido	C++, Java, Python, Go, Ruby	Java	Java	Java
SO Suporte	Google Cloud	JVM	JVM	JVM
Clientes	3 oficiais	1 oficial e 4 da comunidade, além das 18 oficiais do Thrift	12 da comunidade	18 oficiais do Thrift
Protocolos	API RCP e API HBase	Thrift e API HTTP REST	Próprio (CQL)	Thrift

O Bigtable se diferencia dos outros bancos estudados por ser oferecido pela Google apenas com licenciamento proprietário e em modelo SaaS, no ambiente de cloud da própria empresa. A implementação do mesmo segue as mesmas linguagens utilizadas em outros sistemas da empresa, com foco majoritário em C++ e Java.

Os bancos HBase, Cassandra e Accumulo, por sua vez, são todos projetos da fundação Apache, distribuídos sob a licença Apache 2. Os mesmos são implementados em Java e rodam na JVM.

A respeito da comunicação com os bancos, o Bigtable oferece clientes em 3 linguagens e uma API baseada em RPC, além de suportar a API HTTP REST do HBase. Tanto HBase como Accumulo oferecem suporte ao framework Thrift, mas o primeiro oferece também uma API HTTP REST própria. O Cassandra possui clientes em várias linguagens e uma linguagem própria chamada Cassandra Query Language ou CQL. O suporte do Cassandra ao framework Thrift foi tornado obsoleto em favor da CQL, portanto ele não é considerado neste estudo.

Tabela 6. Características do projeto

	Bigtable	HBase	Cassandra	Accumulo
Escalabilidade horizontal	Mestre único	Mestre único	Shared-nothing, fator de replicação	Mestre único
Teorema CAP	CP	CP	Configurável	CP
Controle Concorrência	MVCC	MVCC	CAS e Paxos	MVCC
Transações	Não	Não	Sim	Básicas
Garantias da persistência em disco	Journal	Journal	Journal configurável por processo	Journal configurável por tabela e por sessão
Failover	Automático	Automático	Automático com hinted handoff	Automático
Autenticação e autorização	Google Cloud	SASL, Kerberos	Simples, plugável	SSL, Kerberos

O suporte a escalabilidade ou clusterização é implementada através da arquitetura mestre único no Bigtable, HBase e Accumulo, sendo que o primeiro funciona em cima do Colossus (antigamente chamado de Google File System ou GFS) e os dois últimos dependem do Hadoop Distributed File System (HDFS). O Cassandra segue a arquitetura focada em disponibilidade do Dynamo e funciona com nós idênticos, sem um ponto único de falha e com fator de replicação selecionável por operação.

No espectro do teorema CAP, Bigtable, HBase e Accumulo se classificam como sistemas consistentes (CP) por sua arquitetura de mestre único. Ainda que existam mestres de backup para assumir em caso de falha do mestre, o teorema CAP ressalta que qualquer nó deve ser capaz de responder uma requisição sem dependência de outro nó. O Cassandra, apesar de implementar uma arquitetura focada em disponibilidade (sendo, na configuração padrão, um sistema CA), pode ser configurado para prezar pela consistência, ao custo de redução do desempenho e disponibilidade.

Para o controle de concorrência, o Cassandra utiliza *compare-and-swap* (já explicado na Seção 4.2) para prover atomicidade e isolamento a nível de célula (duas das garantias ACID) e os nós utilizam o protocolo Paxos para obter alta consistência quando solicitado. Bigtable, HBase e Accumulo usam o método MVCC (também explicado na Seção 4.2).

Quanto às garantias da persistência em disco, todos os bancos estudados nesta categoria utilizam o método de *journal* (comumente utilizado pelos bancos tradicionais, é um log de operações em arquivo *append-only* também chamado de *Write Ahead Log* ou WAL) para garantir a durabilidade dos dados. Neste ponto, destaca-se a alta granularidade da configuração oferecida pelo Accumulo, pois é possível configurar a sincronização do journal para o disco a nível de tabela e até mesmo de sessão. O Cassandra também possui

uma opção global que permite configurar a forma como o *fsync* é feito.

O suporte à transações não é oferecido pelo Bigtable e HBase, e atomicidade e isolamento são garantidos apenas para operações em uma única célula. O Accumulo oferece transações básicas, no sentido de que é possível definir condições que devem ser atendidas para que operações sejam executadas, e ambas são executadas com um *lock*. O Cassandra vai além, oferecendo transações (chamadas de *lightweight transactions*) consistentes mesmo entre diferentes nós do cluster, utilizando o protocolo de consenso Paxos (bastante semelhante ao *two-phase commit* utilizado pelos bancos de dados tradicionais).

Todos os bancos oferecem *failover* automático, mas a implementação naturalmente diverge devido às diferenças de arquitetura. Bigtable, HBase e Accumulo, pela sua característica de mestre único, oferecem mestres de backup cuja principal função é assumir a liderança em caso de falha do mestre primário. O mestre primário, por sua vez, controla a distribuição de dados entre os nós do cluster e o *failover* dos nós. No Cassandra os nós comunicam-se uns com os outros através do protocolo *gossip* e usa-se *hinted handoff* (em que um nó responde por requisições de responsabilidade de outro nó enquanto este está inoperante e depois repassa as operações realizadas) para garantir a disponibilidade em caso de falha.

Os métodos de autenticação suportados pelos bancos são bastante variados, o Bigtable possui autenticação integrada com o Google Cloud, no chamado Identity Access Management (IAM), enquanto que HBase oferece os métodos SASL e integração com Kerberos. O Accumulo também oferece suporte ao Kerberos, mas oferece também autenticação via certificados SSL. Por fim, o Cassandra suporta nativamente autenticação e autorização simples baseadas no próprio banco de dados e oferece suporte a módulos plugáveis de autenticação para estender esta funcionalidade.

Tabela 7. Características de manutenção

	Bigtable	HBase	Cassandra	Accumulo
Interface de Gerenciamento	Google Cloud Platform Console	HBase Web UI	Não	Não
Ferramentas de Monitoramento	Stackdriver Monitoring	JMX e JSON na Web UI	JMX	Accumulo Monitor e Hadoop Metrics2 (JMX, Graphite, Ganglia)
Benchmark embutido	Não	LoadTestTool	cassandra-stress	Não

A respeito das interfaces de gerenciamento, enquanto que o Bigtable oferece o painel de gestão do Google (Google *Cloud Platform Console*), o HBase vem com o HBase Web UI que permite tanto a gestão quanto o monitoramento dos recursos computacionais (inclusive por uma API HTTP REST, reportando métricas em JSON). Cassandra e Accumulo não oferecem painéis de gestão nativos, ainda que o Accumulo ofereça uma interface visual para monitoramento.

Além do já mencionado HBase Web UI, o HBase, assim como o Cassandra, oferece integração com JMX para monitoramento da JVM. O Bigtable oferece monitoramento via API através do Stackdriver, outra tecnologia que compõe o Google Cloud. Por final, o Accumulo oferece uma interface visual para monitoramento chamada Accumulo Monitor e métricas através do Hadoop Metrics2, que possui integração com JMX, Graphite e Ganglia.

Por sua característica SaaS, não é de surpreender a ausência de um benchmark

embutido no Bigtable, mas chama atenção esta ausência no Accumulo. Para este fim o Cassandra oferece o cassandra-stress e o HBase oferece o LoadTestTool.

4.4. Bancos de dados orientados a documentos

Esta seção apresenta uma comparação entre as características mercadológicas (Tabela 8), de projeto (Tabela 9) e de manutenção (Tabela 10) dos bancos orientados a documentos.

O MarkLogic é considerado um dos precursores dos bancos NoSQL modernos e tem conseguido se manter dentre os bancos mais populares de sua categoria, porém o MongoDB é hoje considerado o banco NoSQL mais popular dentre todas as categorias [DB-Engines 2017]. CouchDB e Couchbase são fortes concorrentes, enquanto que o OrientDB é considerado uma opção mais viável quando há a necessidade de um banco de documento e de um banco de grafos [Yuhanna et al. 2016].

Tabela 8. Características mercadológicas

	MongoDB	CouchDB	Couchbase	MarkLogic	OrientDB
Lançamento	2009	2005	2010	2001	2010
Licenciamento	AGPL e proprietária	Apache 2	Apache 2 e proprietária	Proprietária	Apache 2
Desenvolvido	C++, C e JavaScript	Erlang	C++, C, Erlang e Go	C++ e JavaScript	Java
SO Suporte	Windows, OS X, Ubuntu, Debian, SUSE, RHEL, Amazon, Linux	Windows, OS X, FreeBSD, Unix-like	Windows, Ubuntu, Debian, Red Hat, SUSE, CentOS	Windows, OS X, Red Hat, SUSE, CentOS, Amazon Linux, Solaris	JVM
Clientes	10 oficiais e 32 da comunidade	11 oficiais	9 oficiais e 2 da comunidade	2 oficiais	3 oficiais e 12 da comunidade
Protocolos	Próprio	API HTTP	Memcached	XDBC, API HTTP, SQL (read-only)	Próprio, API HTTP REST

A respeito dos sistemas operacionais, vale ressaltar que MongoDB, Couchbase e MarkLogic não suportam mais sistemas 32 bits, oferecendo apenas instaladores para SO 64 bits em suas versões mais atuais. O suporte ao OS X do MarkLogic é apenas para fins de desenvolvimento, e quanto ao OrientDB, como o mesmo roda na JVM (*Java Virtual Machine*), podem-se considerar os sistemas operacionais suportados por esta tecnologia.

Nota-se o baixo número de linguagens suportadas pelos bancos MarkLogic e OrientDB pela tendência de utilização destes sistemas diretamente através das APIs HTTP REST, API esta que também é oferecida pelo CouchDB.

No MongoDB a escalabilidade é oferecida através do método conhecido como mestre único, onde todas as consultas e escritas são encaminhadas pelos nós para o mestre. Também é possível configurar para que seja possível ler dados das réplicas, diminuindo o nível de consistência da consulta. CouchDB e Couchbase oferecem uma arquitetura *shared-nothing*, onde cada nó é igual aos outros, com fator de replicação configurável. O MarkLogic implementa escalabilidade pela arquitetura *multi-master* ou mestre-mestre, onde existem vários servidores que atendem como mestres. O OrientDB também utiliza-se desta arquitetura, ainda aplicando o mecanismo de quórum, onde pelo menos um determinado quórum de mestres devem processar o comando com sucesso para que o mesmo seja aceito.

Dentro do teorema CAP explicado na Seção 4.1, MongoDB, Couchbase e MarkLogic são considerados sistemas CP enquanto que CouchDB e OrientDB são sistemas

Tabela 9. Características do projeto

	MongoDB	CouchDB	Couchbase	MarkLogic	OrientDB
Escalabilidade horizontal	Mestre único, fator de replicação	Shared-nothing, fator de replicação	Shared-nothing, fator de replicação	Mestre-mestre	Mestre-mestre e quórum
Teorema CAP	CP	AP	CP	CP	AP
Controle Concorrência	MGL	MVCC	Compare-and-swap	MVCC	MVCC
Transações	Não	Não	Parcial	Sim	Sim
Garantias da persistência em disco	Journal configurável por operação	Append-only configurável	Padrão é assíncrona, configurável por operação	Journal configurável por banco	Journal configurável por processo
Failover	Automático	Manual	Automático	Automático	Automático
Autenticação e autorização	Própria e SSL. Na comercial, Kerberos e LDAP	Própria, simples, OAuth	SASL, LDAP e PAM	Própria, LDAP e Kerberos	Própria e SSL

AP, quando configurados para tal. Todos eles oferecem configurações que violam algumas restrições do teorema, tais como configurar o OrientDB com *write quorum* maior que 1, o que reduz a disponibilidade em favor da consistência; ou configurar o Couchbase para permitir que nós de replicação respondam a leituras, aumentando a disponibilidade e reduzindo a consistência. Estas configurações, porém, não fazem com que o sistema mude de categoria dentro do teorema CAP.

Quanto ao controle de concorrência, o MongoDB aplica MGL (*Multiple Granularity Locking* ou bloqueio de granularidade múltipla), ou seja, o *lock* é feito no menor nível (global, banco de dados, coleção ou documento) possível para garantir a consistência de operações concorrentes. Todos os outros bancos utilizam métodos de bloqueio otimista (*optimistic locking*): o Couchbase usa CAS (*compare-and-swap*) e CouchDB, MarkLogic e OrientDB utilizam MVCC, ambos explicados na Seção 4.2.

Enquanto MongoDB e CouchDB não oferecem nenhum suporte à transações, Couchbase oferece suporte apenas a transações que envolvam um único documento e sugere em sua documentação que o cliente implemente uma forma de *two-phase commit* caso seja necessário. Tanto MarkLogic quando OrientDB oferecem transações mesmo em ambientes distribuídos, mas enquanto que o primeiro utiliza uma forma de *two-phase commit* onde a transação acontece simultaneamente nos diferentes nós do *cluster*, o segundo aplica uma validação de quórum, onde a transação é aceita quando a maioria dos nós entende que a transação ocorreu de forma satisfatória e o restante dos nós é sincronizado posteriormente.

A respeito das garantias da persistência em disco, o único banco que não se utiliza da técnica de *journal* é o Couchbase, fazendo a gravação em disco de forma assíncrona por padrão, porém permitindo a configuração de sincronicidade a nível de operação, para favorecer a consistência em favor do desempenho. O CouchDB se destaca por gravar o próprio banco de dados de forma append-only, diferentemente dos outros bancos que gravam um *journal* que é constantemente reescrito por uma operação de background. Nota-se, porém, que o CouchDB não oferece uma forma nativa de reescrita de seu arquivo de banco, portanto o mesmo cresce em um ritmo constante. O nível de configuração de sincronicidade oferecida pelos bancos no *journal* também é bastante variável, sendo por operação no MongoDB, por banco de dados no MarkLogic e por processo (instância do banco) no OrientDB (a escrita é considerada síncrona quando ocorre antes do retorno ao

cliente, e assíncrona quando o sistema não espera a escrita do *journal* para dar retorno ao cliente).

O único banco que não oferece a opção de *failover* automático é o CouchDB, enquanto que o restante se baseia no esquema de heartbeats (pequenos pacotes de dados enviados entre os nós do cluster em intervalos pré-configurados que definem se os nós estão disponíveis). Nos bancos baseados em nós mestres, como MongoDB, MarkLogic e OrientDB, quando algum mestre cai os nós de replicação promovem uma eleição no cluster para se autopromoverem a mestres.

Quanto a autenticação e autorização, ambas são suportadas por todos os bancos avaliados, e apenas o Couchbase não oferece uma API própria para gerenciamento de usuários dentro do próprio banco de dados. Autenticação baseada em certificados SSL é suportada pelo MongoDB e pelo OrientDB. Tanto MarkLogic quanto a versão comercial do MongoDB suportam autenticação pelo Kerberos e pelo protocolo LDAP, sendo que esta última também é oferecida pelo Couchbase.

Tabela 10. Características de manutenção

	MongoDB	CouchDB	Couchbase	MarkLogic	OrientDB
Interface de Gerenciamento	Apenas comercial	Sim	Sim	Sim	Sim
Ferramentas de Monitoramento	Comandos e ferramenta comercial	URL REST	Na interface e URL REST	Na interface e URL REST	JMX
Benchmark embutido	Sim	Não	Não	Não	Não

O MongoDB é o único dos bancos de documentos avaliados que não oferece uma interface de gerenciamento gratuita e embutida no sistema (ainda que existam várias criadas pela comunidade). Em todos os outros bancos existe uma interface de gerenciamento web que permite efetuar atividades de gerenciamento e monitoria dos mesmos.

Couchbase e MarkLogic se destacam por oferecerem métricas de monitoramento, incluindo históricos, na própria interface de gerenciamento da ferramenta, além de oferecerem web services REST com estas métricas. O CouchDB também oferece um web service, enquanto que o OrientDB expõe métodos de monitoramento através do JMX e o MongoDB oferece comandos internos e uma ferramenta comercial para monitoramento.

Por final, o MongoDB é o único banco a oferecer uma ferramenta de benchmark embutido no sistema, ainda que o OrientDB ofereça suporte à ferramenta de testes do framework TinkerPop.

4.5. Bancos de dados de grafos e triplos

Esta seção apresenta uma comparação entre as características mercadológicas (Tabela 11), de projeto (Tabela 12) e de manutenção (Tabela 13) dos bancos de grafos ou triplos.

O Neo4j é o banco de grafos mais popular atualmente e o OrientDB já foi apresentado anteriormente, na Seção 4.3, dentre os bancos de documentos, por ser um banco multi-modelo. O JanusGraph surgiu a partir do Titan, encerrado em 2015, e o Graph Engine era conhecido como Trinity em suas primeiras versões. O Bitsy, por fim, é um banco de grafos com arquitetura *serverless*, que roda embutido na aplicação cliente.

Enquanto que o Neo4j destaca-se por sua maturidade e presença de mercado, o OrientDB surge como um forte concorrente. Quanto ao JanusGraph, apesar de seu pouco

Tabela 11. Características mercadológicas

	Neo4j	OrientDB	JanusGraph	Graph Engine	Bitsy
Lançamento	2007	2010	2017	2017	2013
Licenciamento	GPLv3 e proprietária	Apache 2	Apache 2	MIT	Apache 2
Desenvolvido	Java e Scala	Java	Java	C# e C++	Java
SO Suporte	JVM	JVM	JVM	Windows e Ubuntu	JVM
Clientes	4 oficiais	3 oficiais e 12 da comunidade	8 oficiais do TinkerPop	1 oficial	1 oficial
Protocolos	Próprio (Bolt), API HTTP REST	Próprio, API HTTP REST	Próprio e TinkerPop	API HTTP REST	N/A

tempo de mercado, deve-se considerar que ele nasceu de um fork no código-fonte do Titan, que foi ativamente desenvolvido entre 2012 e 2015. O Graph Engine surge como uma boa alternativa para os adeptos do ecossistema da Microsoft, e o Bitsy destoa bastante de seus concorrentes pelo seu foco *serverless*.

O modelo de licenciamento do Neo4j é mais restritivo do que seus concorrentes, pois obriga a compra de uma licença proprietária para a distribuição de softwares comerciais. Nota-se a grande adoção do Java como linguagem de desenvolvimento e a consequente utilização da JVM como plataforma de execução, com exceção do Graph Engine, que roda na plataforma do .NET Framework.

A respeito das linguagens suportadas, vale destacar que o Graph Engine suporta apenas as linguagens do .NET, e que o Bitsy oferece suporte apenas ao Java, enquanto que o JanusGraph é completamente compatível com o framework TinkerPop e, portanto, suporta as 8 linguagens oficiais listadas no site deste framework.

Excetuando-se o Bitsy, que roda embutido e, portanto, não utiliza protocolos de comunicação via socket, todos os bancos de grafos estudados oferecem uma API HTTP REST (no JanusGraph, esta API é representada pelo componente Rexter do TinkerPop).

Tabela 12. Características do projeto

	Neo4j	OrientDB	JanusGraph	Graph Engine	Bitsy
Escalabilidade horizontal	2 formas na versão comercial	Mestre-mestre e quórum	Depende do storage	Mestre - Escravo	Não
Teorema CAP	CP	AP	Depende do storage	-	N/A
Controle Concorrência	Lock tradicional	MVCC	Depende do storage	Spinlock por registro	Optimistic locking
Transações	Sim	Sim	Sim	Sim	Sim
Garantias da persistência em disco	Journal síncrono	Journal configurável por processo	Depende do storage	Journal configurável por operação	Append-only síncrono
Failover	Apenas comercial, com quórum	Automático	Depende do storage	Automático, com heartbeats	N/A
Autenticação e autorização	Simples. Na comercial, LDAP e autorização	Própria e SSL	SSL e SASL do TinkerPop	Não	Não

Por seu foco mais comercial, o Neo4j oferece duas opções de escalabilidade horizontal, mas apenas em sua versão paga: *causal cluster*, onde existem múltiplos mestres e é utilizado um fator de replicação fixo; e *highly available cluster*, onde existe um único mestre e os outros nós do cluster são réplicas, mesmo método utilizado pelo MongoDB, conforme explicado na Seção 4.4. Nenhuma das duas formas oferece suporte a *sharding*,

portanto todos os nós contêm todos os dados, enquanto que o OrientDB suporta sharding e sua implementação de cluster está explicada na Seção 4.4.

O Bitsy não possui suporte à escalabilidade horizontal, por seu foco *serverless*, e o JanusGraph não suporta escalabilidade nativamente, delegando ao módulo de armazenamento o provimento desta funcionalidade. O Graph Engine implementa *sharding* através da divisão da memória RAM das máquinas em blocos de memória (*memory trunks*), e o método mestre-escravo para coordenação do cluster, onde o mestre mantêm um registro da localização de todos os dados dentro do cluster.

Quanto à abordagem ao teorema CAP explicado na Seção 4.1, o mesmo não se aplica ao Bitsy por este não prover escalabilidade horizontal, o Neo4j prioriza a consistência frente a partições (CP) em ambas as opções de clusterização, e o OrientDB prioriza a disponibilidade (AP), conforme explicado na Seção 4.4. Dentre os storage backends oferecidos pelo JanusGraph, o Cassandra preza pela disponibilidade (AP) e o HBase preza pela consistência (CP), enquanto que o BerkeleyDB não suporta escalabilidade horizontal. O Graph Engine não cumpre os requisitos de nenhuma das facetas do teorema CAP, pois não provê a linearizabilidade necessária para a classificação como CP devido à sua execução paralelizada, e nem a dissociação entre nós necessária para a classificação como AP (impossível em um esquema mestre - escravo, haja visto que o escravo depende do mestre para prover uma resposta).

Enquanto que o OrientDB utiliza-se do MVCC para controlar a concorrência, o Neo4j faz bloqueios (*locks*) de leitura (*read lock*, não exclusivos, onde múltiplas transações podem obter o mesmo *lock*) e de escrita (*write lock*, exclusivos, onde apenas uma transação pode obter o *lock*). O JanusGraph não faz nenhum controle adicional sobre a concorrência, deixando isso a cargo da camada de armazenamento. O Bitsy utiliza-se do método de bloqueio otimista, onde não são necessários *locks* perceptíveis pelo usuário e em caso de operações concorrentes a segunda falha. O controle de concorrência do Graph Engine baseia-se em *spinlocks* a nível de registro ou célula.

Todos os bancos estudados nesta categoria oferecem suporte à transações e oferecem garantias ACID, exceto o Graph Engine, que não provê serializabilidade (serializability) para threads concorrentes. Nota-se que as garantias ACID do JanusGraph mais uma vez dependem da camada de armazenamento escolhida.

Quanto às garantias da persistência em disco, tanto Neo4j quanto Graph Engine utilizam *journal*, porém no Neo4j ele é síncrono e no Graph Engine a sincronicidade pode ser configurada por operação. O Bitsy utiliza uma técnica semelhante ao CouchDB, estudado na Seção 4.4, onde o próprio banco é escrito na forma de um arquivo *append-only* no disco, porém neste caso a sincronização com o disco é sempre síncrona. Esta característica também depende do *storage backend* no JanusGraph.

No Neo4j o *failover* é oferecido de forma automatizada através de quórum entre os mestres, porém apenas na versão comercial. No Graph Engine o failover é automático através de *heartbeats* entre os nós do cluster, enquanto que no JanusGraph ele depende da camada de armazenamento e no Bitsy esta característica não se aplica.

A autenticação não é oferecida pelo Bitsy e pelo Graph Engine, e o JanusGraph oferece suporte aos protocolos SSL e SASL através do framework TinkerPop. Na versão comunitária do Neo4j é oferecida apenas autenticação simples, enquanto que na versão

comercial há integração com LDAP e suporte completo à autorização. O OrientDB já foi explanado na Seção 4.4.

Tabela 13. Características de manutenção

	Neo4j	OrientDB	JanusGraph	Graph Engine	Bitsy
Interface de Gerenciamento	Sim	Sim	The Dog House, da TinkerPop	Não	Não
Ferramentas de Monitoramento	Na comercial, suporte ao Graphite e exportação para CSV	JMX	Suporte a Graphite, CSV, JMX e outros	Não	JMX
Benchmark embutido	Não	Não	Não	Não	Não

O Neo4j oferece a interface de gerenciamento Neo4j Browser, enquanto que o JanusGraph não oferece nenhuma interface própria, mas pode-se utilizar a The Dog House, do framework TinkerPop. O Bitsy e o Graph Engine não oferecem interface de gerenciamento. Estas interfaces oferecem suporte não só à configuração e gerenciamento do servidor/cluster, como também execução de consultas e visualização de dados do grafo.

Assim como o OrientDB, o Bitsy e o JanusGraph oferecem suporte ao protocolo JMX para monitoramento e o JanusGraph ainda oferece suporte a logs no console, geração de arquivos CSV, e integração com Ganglia, Graphite e Slf4j, além da possibilidade de plugar módulos próprios de monitoramento. O Neo4j oferece suporte ao Graphite e exportação de arquivos CSV. O Graph Engine é o único que não oferece formas de monitoramento do cluster e seus nós.

Nenhum dos bancos estudados oferece um benchmark embutido, ainda que exista a ferramenta do framework TinkerPop que pode ser aplicada em todos eles, exceto no Graph Engine.

4.6. Discussão dos resultados

Dentre os bancos de dados chave-valor analisados, o Memcached oferece maturidade e simplicidade, com um leque relativamente pequeno de funcionalidades se comparado a seus concorrentes. O Redis possui foco semelhante, mas traz opções como suporte a dados estruturados e persistência em disco, suprindo várias demandas não atendidas pelo Memcached. O Voldemort se destaca por ter sido concebido e ainda ser mantido pelo LinkedIn como "uma cópia open-source do Dynamo" [Voldemort 2009], o que lhe confere certa segurança, além da persistência em disco configurável que o diferencia dos concorrentes.

O Aerospike traz um foco mais comercial, assim como Hazelcast e Riak KV, e destaca-se por ser o único a oferecer otimizações específicas para execução em SSD, que vêm ganhando espaço, além de dar suporte completo a todos os clientes das 12 linguagens de programação suportadas, pois o desenvolvimento dos mesmos é feito pela própria empresa. O destaque do Hazelcast fica por conta do suporte completo à transações ACID, característica rara nos bancos NoSQL, além do suporte ao protocolo do Memcached, o que facilita a migração de clientes deste. Por fim, o Riak KV herda muitas das características do Dynamo e oferece uma ampla gama de funcionalidades e configurações, destacando-se como uma das opções mais robustas.

Na análise dos bancos de dados de famílias de colunas, chama a atenção a aparente similaridade de três dos quatro bancos analisados: HBase, Cassandra e Accumulo são to-

dos projetos da fundação Apache e implementados em Java. Apesar disso, as respectivas arquiteturas diferem consideravalmente. O Cassandra se destaca em funcionalidades e garantias oferecidas, sendo o único com arquitetura *shared-nothing* (sem um ponto único de falha), podendo priorizar a disponibilidade em favor da consistência, e ainda assim o único a oferecer suporte a transações. HBase e Accumulo compartilham muitas características, mas o primeiro é mais maduro, possuindo uma comunidade maior e mais integração com o ambiente Hadoop, enquanto que o segundo depende menos do esquema dos dados, facilitando alterações na estrutura. O Bigtable difere bastante do restante, por ser uma ferramenta oferecida apenas comercialmente em formato SaaS.

Os bancos de dados orientados a documentos, por sua vez, divergem largamente em suas arquiteturas. Enquanto que o líder de popularidade MongoDB segue uma arquitetura mais tradicional, priorizando a consistência do dados, oferecendo escalabilidade com mestre único e utilizando *locks* para controle de concorrência, o CouchDB foca na disponibilidade, com arquitetura *shared-nothing* e *locking* otimista através do MVCC. O MarkLogic oferece um produto mais maduro e estável que os concorrentes, também priorizando a consistência mas usando *locking* otimista através do MVCC e uma arquitetura com múltiplos mestres, além de oferecer suporte à transações ACID. O Couchbase aparece como uma alternativa interessante, oferecendo funcionalidades comparáveis a estes e sendo lembrado principalmente pela sua alta performance [Yuhanna et al. 2016]. Por fim, o OrientDB é o representante do ecossistema Java dentre os bancos de documentos, apostando em sua arquitetura multi-modelo, o que parece ser uma tendência dentre os bancos NoSQL [Heudecker et al. 2016].

O destaque dos bancos de dados de grafos ou triplos fica por conta do Neo4j. Sendo o banco mais maduro de sua categoria, oferece uma abordagem mais tradicional com prioridade pela consistência e uso de *locks*, além de duas formas de escalabilidade horizontal, porém muitas funcionalidades apenas estão disponíveis na versão comercial. Também nota-se a predominância da linguagem Java e a variedade em relação ao método de controle de concorrência e ao teorema CAP. O Graph Engine destoa dos outros bancos por fazer uso exclusivamente da *stack* de tecnologias da Microsoft, o que é esperado quando considera-se que o mesmo é desenvolvido pela própria empresa.

5. Conclusões

Neste artigo, foi realizado um estudo comparativo sobre os bancos de dados NoSQL de diferentes categorias, detalhando suas características mercadológicas, de projeto e de manutenção. O trabalho realizado permite que sejam feitas comparações que auxiliam na escolha de um banco de dados mais apropriado e adequado ao cenário almejado. Diferente das comparações tradicionais, buscou-se aplicar uma análise de cunho científico sobre as características e funcionalidades de cada banco de dados. Isso porque existe um *marketing* agressivo adotado pelos fornecedores, que não esclarecem as limitações do banco de dados e buscam com viés positivo qualificar os mesmos.

A classificação em quatro categorias(chave-valor, sistemas de documentos, família de colunas e triplos) que representam a classe de problemas abordados pelos diversos bancos NoSQL, surgidos em um curto espaço de tempo, têm se mostrado insuficiente para a tomada de decisão sobre o banco ideal a se adotar em determinado cenário. A principal contribuição desse estudo é permitir a visão mais amplas das vantagens e limitações dos

mais diversos bancos que existem dentro dessas categorias. Além disso, comprovar que apesar desse buscarem atender o mesmo macro objetivo, eles são diferentes em funcionalidades e configurações entre si.

Também foi possível observar que o crescimento do volume de dados processados pelas aplicações modernas tem impulsionado o surgimento dos bancos de dados NoSQL. Pela promessa de adaptação às novas demandas de maneira elástica, oferecendo disponibilidade e velocidade além das oferecidas pelos bancos relacionais tradicionais. Porém, isso vem a um custo, geralmente sacrificando algumas das funcionalidades focadas em consistência oferecidas por estes.

A tendência natural é a gradual estabilização no crescimento do número de sistemas NoSQL emergentes. Na medida em que se definem os líderes das categorias, permanece a necessidade de manter um comparativo atualizado sobre as reais capacidades dos sistemas emergentes, considerando também a evolução dos bancos já estabelecidos. Tendo em vista os resultados apresentados e conhecendo os bancos oferecidos, o próximo passo para a escolha de uma tecnologia é conhecer o seu desempenho no cenário em que se deseja utilizá-la.

Uma vez que esse trabalho elaborou um estudo comparativo e identificou os bancos de dados com características semelhantes, uma proposta de trabalho futuro é realizar uma análise do desempenho dos bancos NoSQLs abordados nesse estudo. Esta análise pode ser feita em diferentes ambientes computacionais, tais como em nuvem e complementarmente em uma plataforma distribuída. Nota-se que a literatura também carece deste tipo de análise e discussão voltada ao desempenho, embora já existam algumas pesquisas para bancos de dados específicos [Zhang et al. 2015, Cao et al. 2016].

Referências

Abadi, D. (2012). Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45(2):37–42.

Abramova, V., Bernardino, J., and Furtado, P. (2014). Which nosql database? a performance overview. *Open Journal of Databases (OJDB)*, 1(2):17–24.

Apache Software Foundation (2008a). Accumulo. Access on <https://accumulo.apache.org/>.

Apache Software Foundation (2008b). Apache HBase. Access on <http://hbase.apache.org/>.

Apache TinkerPop (2008). Apache TinkerPop. Access on <http://tinkerpop.apache.org/>.

Bradberry, R. and Lubow, E. (2013). *Practical Cassandra: a developer's approach*. Addison-Wesley.

Brewer, E. (2000). Towards Robust Distributed Systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA. ACM.

Brewer, E. (2012). CAP twelve years later: How the "rules"have changed. *Computer*, 45(2):23–29.

Cao, W., Sahin, S., Liu, L., and Bao, X. (2016). Evaluation and Analysis of In-Memory Key-Value Systems. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 26–33.

Carlson, J. L. (2013). *Redis in Action*. Manning, Shelter Island, NY, USA.

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 Trans. on Computer Systems (TOCS)*, 26(2):4.

CouchDB (2005). Apache CouchDB. Access on <http://couchdb.apache.org/>.

DB-Engines (2017). DB-Engines - Knowledge Base of Relational and NoSQL Database Management Systems. Access on <https://db-engines.com/>.

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*, 41(6):205–220.

Deka, G. C. (2014). A survey of cloud database systems. *IT Professional*, 16(2):50–57.

Fowler, A. (2015). *NoSQL For Dummies*. John Wiley & Sons, 111 River Street, Hoboken, New Jersey, USA.

Gessert, F., Wingerath, W., Friedrich, S., and Ritter, N. (2017). NoSQL database systems: a survey and decision guidance. *Computer Science - Research and Development*, 32(3):353–365.

Gilbert, S. and Lynch, N. (2002). Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2):51–59.

Goldsmith, S. and Crawford, S. (2014). *The Responsive City: Engaging Communities Through Data-Smart Governance*. John Wiley & Sons, 111 River Street, Hoboken, New Jersey, USA.

Han, J., E, H., Le, G., and Du, J. (2011). Survey on NoSQL database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366.

Hecht, R. and Jablonski, S. (2011). NoSQL evaluation: A use case oriented survey. In *2011 International Conference on Cloud and Service Computing (CSC)*, pages 336–341.

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492.

Heudecker, N., Feinberg, D., Adrian, M., Palanca, T., and Greenwald, R. (2016). Magic Quadrant for Operational Database Management Systems. Technical report, Forrester Research, Inc., 56 Top Gallant Road, Stamford, CT 06902-7700 U.S.A.

JanusGraph (2017). JanusGraph: Distributed Graph Database. Access on <http://janusgraph.org/>.

Kabakus, A. T. and Kara, R. (2017). A performance evaluation of in-memory databases. *of King Saud University - Computer and Information Sciences*, 29(4):520–525.

Kleppmann, M. (2015). Please stop calling databases CP or AP. Access on <https://martin.kleppmann.com/2015/05/11/please-stop-calling-databases-cp-or-ap.html>.

Lake, P. and Crowther, P. (2013). *Concise guide to databases*. Springer-Verlag London, 111 River Street, Hoboken, New Jersey, USA.

Leavitt, N. (2010). Will NoSQL Databases Live Up to Their Promise? *Computer*, 43(2):12–14.

MongoDB (2009). MongoDB for GIANT ideas. Access on <https://www.mongodb.com/>.

Moniruzzaman, A. B. M. and Hossain, S. A. (2013). NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. *International Journal of Database Theory and Application*, 6(4).

Nayak, A., Poriya, A., and Poojary, D. (2013). Type of NOSQL Databases and its Comparison with Relational Databases. *International Journal of Applied Information Systems*, 5(4):16–19.

Neo4j (2007). Neo4j: The World's Leading Graph Database. Access on <https://neo4j.com/>.

Pokorny, J. (2013). NoSQL databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, 9(1):69–82.

Thornton, S. (2013). Chicago's windy grid: Taking situational awareness to a new level. *Data Smart City Solutions*.

Voldemort (2009). Project Voldemort. Access on <http://www.project-voldemort.com/>.

Yuhanna, N., Leganza, G., and Austin, C. (2016). The Forrester Wave™: Big Data NoSQL, Q3 2016. Technical report, Forrester Research, Inc., 60 Acorn Park Drive, Cambridge, MA 02140 USA.

Zhang, H., Chen, G., Ooi, B. C., Tan, K.-L., and Zhang, M. (2015). In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948.