Instituto Superior de Engenharia do Porto

Bases de Dados NoSQL

Ricardo Manuel Fonseca Cardoso

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Arquiteturas Sistemas e Redes

Orientador: Paulo Alexandre Gandra de Sousa

Júri:

Presidente:

Doutor Luís Miguel Moreira Lino Ferreira, Instituto Superior de Engenharia do Porto

Vogais:

Doutor Paulo Jorge Machado Oliveira, Instituto Superior de Engenharia do Porto

Doutor Paulo Alexandre Gandra de Sousa, Instituto Superior de Engenharia do Porto

Porto, Novembro 2012

Dedicatória

Aos meus pais, irmãs, família e amigos

Resumo

Com o advento da invenção do modelo relacional em 1970 por E.F.Codd, a forma como a informação era gerida numa base de dados foi totalmente revolucionada. Migrou-se de sistemas hierárquicos baseados em ficheiros para uma base de dados relacional com tabelas relações e registos que simplificou em muito a gestão da informação e levou muitas empresas a adotarem este modelo. O que E.F.Codd não previu foi o facto de que cada vez mais a informação que uma base de dados teria de armazenar fosse de proporções gigantescas, nem que as solicitações às bases de dados fossem da mesma ordem.

Tudo isto veio a acontecer com a difusão da internet que veio ligar todas as pessoas de qualquer parte do mundo que tivessem um computador. Com o número de adesões à internet a crescer, o número de *sites* que nela eram criados também cresceu (e ainda cresce exponencialmente). Os motores de busca que antigamente indexavam alguns *sites* por dia, atualmente indexam uns milhões de sites por segundo e, mais recentemente as redes sociais também estão a lidar com quantidades gigantescas de informação. Tanto os motores de busca como as redes sociais chegaram à conclusão que uma base de dados relacional não chega para gerir a enorme quantidade de informação que ambos produzem e como tal, foi necessário encontrar uma solução. Essa solução é NoSQL e é o assunto que esta tese vai tratar.

O presente documento visa definir e apresentar o problema que as bases de dados relacionais têm quando lidam com grandes volumes de dados, introduzir os limites do modelo relacional que só até há bem pouco tempo começaram a ser evidenciados com o surgimento de movimentos, como o *BigData*, com o crescente número de *sites* que surgem por dia e com o elevado número de utilizadores das redes sociais. Será também ilustrada a solução adotada até ao momento pelos grandes consumidores de dados de elevado volume, como o Google e o Facebook, enunciando as suas características vantagens, desvantagens e os demais conceitos ligados ao modelo NoSQL. A presente tese tenciona ainda demonstrar que o modelo NoSQL é uma realidade usada em algumas empresas e quais as principias mudanças a nível programático e as boas práticas delas resultantes que o modelo NoSQL traz.

Por fim esta tese termina com a explicação de que NoSQL é uma forma de implementar a persistência de uma aplicação que se inclui no novo modelo de persistência da informação.

Palavras-chave: NoSQL, Modelo Relacional, Bases de dados, BigData, Distribuição

Abstract

With the upcoming of the relation model, by E.F.Codd in 1970, the method of information management was totally revolutionized. The file base hierarchical systems were migrated to a relational database with tables, relationships and records that simplified a lot of information management and lead many corporations to use this model. What E.F.Codd does not foreseen, was the fact the information that a database would handle would be in gigantic proportions, neither that the number of requests made to the database would be in the same proportions.

All of this becomes a reality when the internet arrived and connected all people from any part of the world that owns a computer. With the number of internet users always growing, the number of websites hosted in the internet also grew and it still grows. Then, the search engines that in the old days only indexed a few sites per day nowadays index a few millions by hour, later on the so called social networks also see themselves dealing with enormous quantities of information. At that point, search engines and also social networks concluded that a relational database wasn't enough to manage the huge information that both have so, it was imperative to find a solution and that solution was NoSQL that is the main matter of this thesis.

The present document defines and presents the problem that relational databases have when dealing with huge data volumes. Announces the limits of the relational model that only now started to appear with the rising movements such as the BigData, the growing number of sites that are created daily and the continuously increasing number of social networks users. This thesis also shows the adopted solution until now by companies that deal with a lot of information, like Google and Facebook, explaining its characteristics, advantages, disadvantages and all the other concepts that the NoSQL model brings along. The present document also illustrates that NoSQL is a reality used in some corporations and the main changes from a programmatically view that come with the NoSQL model and the resulting good practices.

Finally this thesis ends explaining that NoSQL is a way of implementing an application's persistence that will be included in the next persistence model.

Keywords: NoSQL, Relational Model, Databases, Bigdata, Distribution

Agradecimentos

Em primeiro lugar gostaria de agradecer aos meus pais por apoiarem incondicionalmente durante todo o meu percurso académico e por nunca terem perdido a fé em mim. Agradeço também as minhas duas irmãs pois sem o seu apoio teria sido difícil para mim prosseguir com os meus estudos.

Agradeço em especial a todos os meus amigos e colegas de faculdade, a destacar o Bruno Garcês e o Nuno Rocha que me deram muito apoio na elaboração desta tese, aos meus companheiros de casa aqui no Porto Jorge Oliveira, Pedro Soares e Jorge Santos, aos meus colegas de grupo Bruno Saraiva, Ricardo Vieira, Paulo Taveira e Hugo Bastos, não só nos trabalhos de mestrado como também nos de licenciatura.

Agradeço a todos os meus professores, em especial ao Engenheiro Álvaro Ferreira e à professora Paula Miranda do Externato de Vila Meã que muito me ajudaram durante os meus anos de ensino secundário.

Finalmente, mas não menos importante, agradeço ao meu orientador Doutor Paulo Gandra de Sousa cuja orientação em muito me ajudou ao longo da realização desta tese. Sem ele todo este trabalho não seria possível.

Índice

Lista de Figuras

Lista de Tabelas

Acrónimos e Símbolos

Lista de Acrónimos

ACID	*Atomicity Consistency Isolation Duration*
ADO	*ActiveX Data Objects*
API	Interface de Programação de Aplicações
BASE	*Basically Available, Soft state, Eventual Consistency*
Blobs	*Binary Large Objects*
CURL	*Command Line URL*
DBA	*Database Administrator*
EDM	*Entity Data Model*
ELB	Elastic Load Balancer
HTTPS	Hypertext Transfer Protocol Secure
IDE	*Integrated Development Environment*
JDBC	*Java Database Connectivity*
JSON	*JavaScript Object Notation*
LDAP	*Lightweight Directory Access Protocol*
MVC	*Model View Controller*
MVCC	*Multi Version Concurrency Control*
NoSQL	*Not Only SQL*
OGM	*Object Grid Mapping*
ORM	*Object-Relational Mapping*
REST	*Representational State Transfer*
SQL	*Structured Query Language*
SSL	Secure Sockets Layer

1 Introdução

1.1 Enunciado do Problema

Com o rápido desenvolvimento da internet, as empresas viram uma nova oportunidade de negócio e como tal as aplicações informáticas migraram de sistemas fechados para a internet, um meio acedido por todos sob a forma de *websites*. De forma a organizar todos os *websites* que surgem foi criado um novo tipo de aplicação chamada motor de busca, que por sua vez tem como objetivo guardar informação sobre a página, nomeadamente palavras-chave, referências externas e mais recentemente meta-dados. Mais recentemente surgiu um novo tipo de aplicações denominadas de redes sociais. Estas redes sociais mantêm e gerem um grande número de utilizadores, cada um deles com diversas informações associadas a si mesmo, como por exemplo informações básicas (nome, data de nascimento, etc.), fotos, vídeos e ligações a outros utilizadores (amigos); toda esta informação gera um grande volume de dados. Quer no caso dos motores de busca, quer no caso das redes sociais, para armazenar toda a informação é necessário ter uma base de dados. Num cenário normal, as bases de dados relacionais são usadas para esse fim, no entanto o número de *sites* que aparecem por dia é elevado; de acordo com (Pingdom, 2011) é possível só num mês surgirem 255 milhões de novos *sites* e, segundo (Brandão, 2011), no caso de uma rede social como o Twitter que tem mais de 200 milhões de utilizadores registados e, por mês recebe 95.8 milhões de visitas. Deste modo, a informação subjacente à indexação de todos os *sites* que surgem diariamente e a de uma rede social é de tal forma enorme que, de acordo com (Jacobs, 2009), uma base de dados relacional tende a diminuir o desempenho à medida que tem muita informação. Uma possível solução é ter uma grande estrutura de processamento com milhares de servidores. Tomando o exemplo do Google, este lida com 24 Petabytes de informação por dia (Roe, 2012) e como tal, é necessário ter uma base de dados distribuída pronta a escalar rapidamente em função do grande volume de dados tratados pelo Google. Este nível de escalabilidade é difícil, senão impossível, de obter ao ser usada uma base de dados relacional, não só pelas restrições do modelo relacional em si, como também pela elevada complexidade da distribuição da base de dados.

Para solucionar este problema, os principais motores de busca e mais tarde as redes sociais, inventaram um novo modelo de base de dados NoSQL - segundo (Tiwari, 2011) é um termo que designa todas as bases de dados que não seguem os princípios das bases de dados relacionais e que estão relacionados com grandes volumes de dados.

1.2 Objetivos

Esta tese tem os seguintes objetivos:

- Compreender o modelo de bases de dados NoSQL:
    - Modo de funcionamento.
    - Diferentes tipos de bases de dados NoSQL.
    - Modelo de programação.
    - Vantagens e Desvantagens.
- Comparar o modelo relacional e o modelo NoSQL.
- Analisar o ponto de vista do programador sobre:
    - O que mudou.
    - Novas técnicas.
    - Ferramentas existentes.
- Comparar os diversos produtos de bases de dados NoSQL.
- Mostrar um exemplo prático.

1.3 Critérios de Sucesso

O sucesso da **compreensão do modelo de bases de dados NoSQL** virá com a análise concreta dos conceitos, especificidades, tipologias subjacentes ao modelo NoSQL, construída a partir do estudo das diferentes *API's* e metodologias.

O êxito da **confrontação do modelo relacional e do modelo NoSQL** deve-se à comparação dos dois modelos, salientando os principais pontos de divergência e as mudanças mais significativas com base nos fundamentos teóricos de cada modelo.

O resultado da **comparação dos diversos produtos de bases de dados NoSQL** será medido pelo estudo e demonstração dos diversos conceitos individuais de cada produto obtidos pela análise da documentação oficial.

Para concluir com sucesso **a análise do ponto de vista do programador e a montagem de um exemplo prático**, deve-se escolher e montar um dos diversos exemplos práticos que muitas empresas disponibilizam.

1.4 Contribuições Esperadas

As contribuições desta tese visam incluir:

- Um estudo sobre o conceito, técnicas, metodologias e *API's* das bases de dados NoSQL;
- Uma comparação entre o modelo relacional e o modelo NoSQL;
- Uma comparação entre os diversos produtos NoSQL;
- A montagem de um exemplo prático que fornecerá as bases para a análise do ponto de vista do programador.

1.5 Método de trabalho

Foi efetuada uma revisão conceptual do termo NoSQL, tendo decorrido desde Setembro de 2011 ate ao final Janeiro de 2012. Inicialmente, para obter uma primeira impressão sobre o tema, foram consultados os *sites* nosql-database[1] e highscalability[2] de forma a não só encontrar o significado do tema e também os conceitos a ele associados bem como o estado do mercado em termos de bases de dados NoSQL.

Tomando por base o que foi obtido na pesquisa inicial, foi feita uma pesquisa no motor de busca Google sobre livros que abordassem o tema de uma forma geral, tal como o Professional NoSQL de editora Wrox e, livros que abordassem o tema de uma forma mais específica, tal como os guias de um produto NoSQL, como por exemplo "Cassandra The Defenitive Guide", "Couchdb The Definitive Guide", etc. Posto isto e, depois de ter uma visão mais concreta da realidade do NoSQL, foram pesquisados artigos nos repositórios B-on e ACM que correspondessem às seguintes *queries* de pesquisa: "NoSQL Clusters" sendo que no primeiro foram encontrados 27 artigos e no segundo 101 artigos; "NoSQL Bigdata" que no repositório B-on devolveu 3 artigos e no repositório ACM devolveu 2 artigos; "NoSQL Social Network" que no repositório B-on devolveu 31 artigos e no repositório ACM devolveu 273 artigos; "NoSQL Search Engines" que no repositório B-on devolveu 39 artigos e no repositório ACM devolveu 65 artigos;

Para se ter uma ideia geral da implementação do NoSQL no mundo real, foram pesquisados os repositórios Infoq[3], Slideshare[4] e Dzone[5] bem como Google, com os nomes de cada produto: "Cassandra", "Windows Azure Table", etc, e que devido à imensidão de resultados não foi possível apurar a quantidade exata de artigos encontrados. É importante referir que dos resultados obtidos foram ignorados os artigos que estavam em duplicado, que eram sobre

[1] http://nosql-database.org/
[2] http://highscalability.com/
[3] http://www.infoq.com/
[4] http://www.slideshare.net/
[5] http://dzone.com/

projetos que ainda não vingaram ou que em pouco ou nada tinham a ver com o tema e os conceitos ou a implementação de NoSQL no mundo real.

Por fim, para a seleção de um exemplo apropriado, foi feita uma pesquisa no *google* pelos *sites* dos fabricantes e respetivos artigos e foi selecionado o exemplo mais completo, o do Microsoft Windows Azure elaborado segundo (Haridas, 2009).

1.6 Organização da tese

O presente documento encontra-se dividido em seis capítulos.

Neste **primeiro capítulo** é enunciado o problema, são definidos os objetivos e correspondentes critérios de sucesso, são enumeradas as contribuições esperadas, o método de trabalho e é introduzida a história do NoSQL.

No **segundo capítulo** é feita uma contextualização ao tema NoSQL enunciando os seus fundamentos teóricos, como por exemplo o teorema CAP e o movimento *BigData* e são apresentadas as limitações do modelo relacional.

No **terceiro capítulo** são apresentadas as características das bases de dados NoSQL, os diferentes tipos de bases de dados NoSQL, uma breve comparação entre os referidos tipos bem como o seu uso mais comum. Neste capítulo são também referidos conceitos intrínsecos a qualquer base de dados NoSQL como é o caso do *Map/Reduce* e do *Sharding*.

No **quarto capítulo** são apresentados três exemplos reais de empresas que utilizam NoSQL nos seus produtos.

No **quinto capítulo** são apresentados alguns dos produtos NoSQL existentes, é apresentado um exemplo de uma base de dados em quatro sistemas de bases de dados NoSQL (um de cada tipo) e no tradicional SQL. Neste capítulo são também referidas as boas práticas existentes de NoSQL bem como a modelação de relações e as alterações para o programador.

No **sexto capítulo** são apresentadas as conclusões e é referido o potencial trabalho futuro.

2 Contextualização

2.1 Introdução

Com a elevada expansão da Internet, o número de utilizadores tem vindo a aumentar constantemente, sendo que em Março de 2012, foram registados 2,280 milhões (Miniwatts Marketing Group, 2011) e como tal, o número de páginas da Internet e o número de utilizadores das redes sociais também cresceu como foi referido anteriormente. Com um aumento tão drástico, quer os motores de busca e mais tarde as redes sociais, tiveram de lidar com quantidades enormes de dados a processar, alcançando a ordem dos Petabytes (o Google lida com 24 Petabytes de informação por dia e uma rede social como o Twitter lida com 4 Petabytes de informação por ano (Boyce, 2010)).Uma base de dados por si só não consegue lidar com tanta informação, isto é, as bases de dados relacionais não foram feitas para lidar com um enorme volume de informação num curto espaço de tempo, nem preveem o aumento das suas capacidades de processamento, de um dia para o outro, sem ser necessário uma reconfiguração. Os motores de busca e as redes sociais quando começaram a lidar com um volume tão elevado de dados tentaram resolver o problema aumentando o número de servidores em que corriam as suas aplicações até se alcançar o desempenho esperado, mas, chegaram à conclusão que iria sempre existir um estrangulamento na base de dados prejudicando o desempenho do sistema. Como tal, numa segunda fase distribui-se a base de dados por vários nós e para o caso específico de uma base de dados relacional ter mais nós não só complica a *query* dos dados ao sondar os múltiplos nós da base de dados como também implica uma mudança significativa na arquitetura da aplicação.

2.2 Limitações do Modelo Relacional

As limitações do modelo relacional começaram a ser evidentes à medida que o volume de dados começou a aumentar. Com a expansão da internet e das redes sociais, as bases de dados relacionais começaram a ter que lidar com um grande volume de informação na ordem dos *Petabytes* e de um número elevado de pesquisas ao mesmo tempo. De forma a aumentar o desempenho de uma base de dados relacional, as únicas opções são: escalar verticalmente a base de dados, que consiste em substituir o *hardware* existente por um mais rápido; e escalar a base de dados horizontalmente que consiste em distribuir a base de dados por múltiplas instâncias. Das duas soluções anteriores a mais adotada era o escalamento vertical, mas, hoje em dia, esta solução tende a não ser a mais viável pois para processar informação na ordem dos Petabytes e a crescer, é necessário *hardware* extremamente caro e mesmo assim não é suficiente. Então, a solução passará por distribuir a base de dados por várias instâncias mas porém, esta solução também apresenta problemas. Segundo (Jacobs, 2009), à medida que o número de linhas numa base de dados relacional aumenta drasticamente, o desempenho da base de dados diminui, (a escalabilidade de uma base de dados relacional tem um limite de acordo com (Staveley, 2012)) pois, à medida que a base de dados é distribuída, a complexidade dos *joins* com tabelas que não estejam na mesma máquina aumenta. Do mesmo modo, à medida que o número de instâncias da base de dados aumenta, a garantia das propriedades ACID em transações distribuídas fica também muito difícil de gerir. Para agravar, pode ainda ser necessária a contratação de mais *DBA's* (Administradores de Base Dados) para gerirem toda esta complexidade.

Por isso, as bases de dados relacionais começaram a evidenciar os seguintes problemas quando lidam com volumes enormes de informação:

- Escalabilidade: de acordo com (Ferreira, 2010) as bases de dados escalam, mas não escalam facilmente. É possível escalar vertical e/ou horizontalmente mas a complexidade exigida para ter de gerir múltiplos nós da base dados ou o custo do *hardware* para suportar a distribuição são grandes entraves. Também as alterações, no caso de já existir uma aplicação a interagir com a base de dados, seriam de ordem elevada que poderia levar a profundas modificações da sua arquitetura.
- Um aumento na complexidade geral da base de dados e na sua gestão, quando esta é distribuída por varias instancias de acordo com (Staveley, 2012).
- As propriedades ACID de uma base de dados relacional obrigam a que esta tenha uma estrutura muito restrita.

Uma base de dados relacional também não lida bem com alterações no *schema* pois adicionar uma nova tabela ou relação pode mudar o comportamento da base de dados e das aplicações que são usadas por ela.

Uma das grandes limitações do modelo relacional é o facto de não suportar objetos nativamente, o que implica que seja necessário programar o mapeamento dos objetos da aplicação para a estrutura da base de dados ou recorrer a *frameworks* que agilizem este processo. Não é que o NoSQL remova a necessidade de mapeamento por completo, no entanto, devido à simplicidade de alguns dos seus modelos, ajuda a aligeirar o problema. No caso da gestão de documentos, existe um modelo de base de dados NoSQL preparado para essa estrutura: o modelo de Documento. O outro caso prende-se na dificuldade de manter e consultar grafos numa base de dados relacional mas, igualmente ao outro caso, dentro do NoSQL existe um modelo especificamente desenhado e otimizado para grafos.

Nos dois exemplos anteriores vai ter de existir um mapeamento ou um mecanismo semelhante, no entanto, a própria base de dados vai ajudar a simplificar esse mapeamento. No caso das bases de dados orientadas a Documento, segundo (Andrew J. Brust, 2011), os tipos de dados dentro dos documentos são os do *JavaScript*, como tal, não há necessidade de um mapeamento tão pesado como o de uma base de dados relacional.

2.3 Teorema CAP

O Teorema CAP de acordo com (Sousa, 2010), (Brewer, 2000) e (Lynch, 2002), assenta em três requisitos que afetam os sistemas distribuídos. Esses requisitos são:

- *Consistência*: que significa que um sistema fica pronto a ser usado logo após a inserção de um registo;
- *Disponibilidade*: que se traduz em o sistema estar ativo durante um período de tempo;
- *Tolerância a Falhas*: diz respeito à capacidade do sistema funcionar em caso de uma falha dos seus componentes;

Este teorema explica que não é possível, num sistema de dados distribuído, ter ao mesmo tempo estes três requisitos. O que é possível é ter dois deles ao mesmo tempo, como tal a escolha depende de qual dos requisitos não é tão necessário.

Deste teorema advém o modelo de consistência eventual (***B**asically **A**vailable, **E**ventual, **C**onsistency*) que as bases de dados NoSQL usam. Este modelo de consistência já não é novo e está normalmente associado a sistemas distribuídos. Estipula que se durante um determinado tempo não existirem atualizações, todas as atualizações pendentes serão propagadas por todos os nós do sistema ficando no fim todo o sistema consistente.

Este teorema foi fruto de um estudo realizado por (Brewer, 2000) da INKTOMI que depois foi provado por (Lynch, 2002). Neste estudo/demonstração, os autores explicam pormenorizadamente o porquê de nunca ser possível num sistema distribuído ter consistência, disponibilidade e tolerância a falhas ao mesmo tempo. A prova do teorema CAP segundo (Browne, 2009) confirma o que anteriormente (Brewer, 2000) e (Lynch, 2002) apresentaram.

Existem dois nós de rede (Nó 1 e Nó 2) e ambos partilham um valor V_0 como exemplifica a figura seguinte:

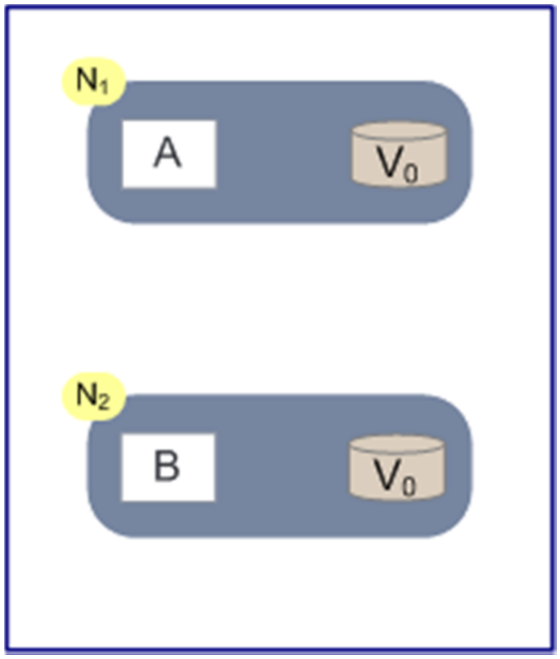

Figura 1 – Esquema do sistema retirado de (Browne, 2009)

Depois, no nó A corre um processo de escrita do valor e no nó B corre um processo de leitura de um valor como mostra a figura seguinte:

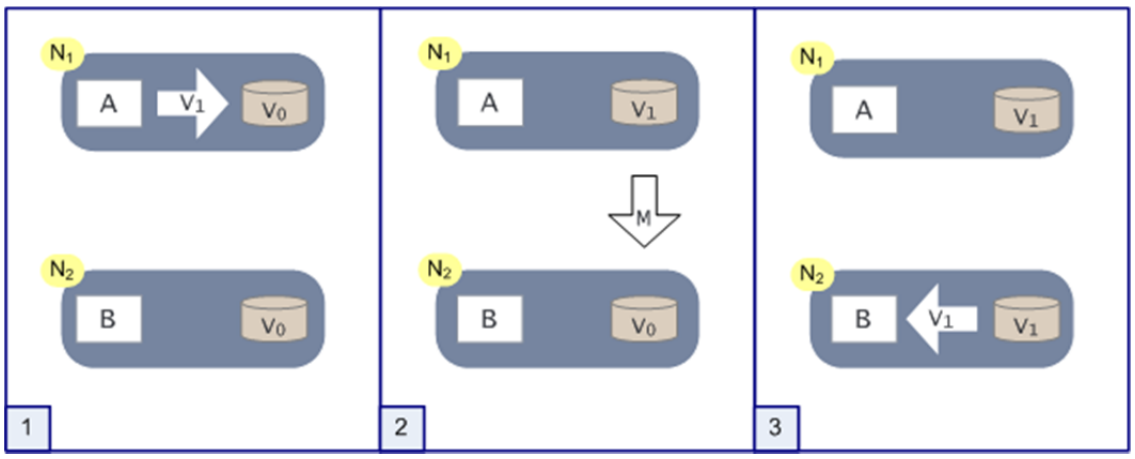

Figura 2- Diagrama dos processos do sistema retirado de (Browne, 2009)

Numa execução sem falhas, o processo de escrita no A iria fazer a escrita do valor V_1. De seguida, enviaria uma mensagem ao nó B com a atualização do valor V_0 para V_1. Depois, o processo de leitura no nó B iria ler o valor V_1 sem problemas. No entanto, se a mensagem de atualização do valor não chegar, por exemplo por falha na rede, e for feita a leitura nesse momento, vai existir uma inconsistência na base de dados como mostra a figura seguinte:

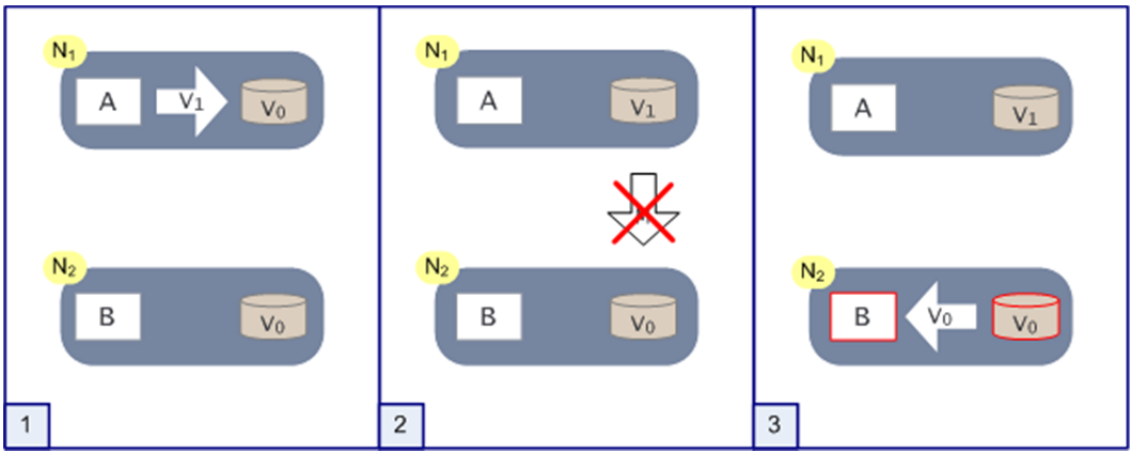

Figura 3 – Diagrama de erro de entrega da mensagem retirado de (Browne, 2009)

Admitindo que o método de passagem de mensagens seja síncrono ou assíncrono, o problema persistirá pois com um método assíncrono o nó A não tem garantias da entrega das mensagens. Com um método síncrono não existe forma de saber se a mensagem não chegou porque existe falha da rede ou atraso na rede.

Então, como só é possível escolher dois dos três requisitos, e de acordo com (Brewer, 2000) é possível obter as seguintes combinações:

- **Consistência e Disponibilidade -** Ao escolher Consistência e Disponibilidade abdica-se da tolerância a falhas de forma a suportar *commits* em duas fases e protocolos de validação de cache. É usado em sistemas de LDAP e sistemas de ficheiros xFS;
- **Consistência e tolerância a falhas -** Escolhendo a tolerância a falhas e consistência deixa-se de parte a disponibilidade de forma a ganhar mecanismos de *lock* pessimistas e uma melhor gestão das réplicas. Tem aplicações práticas em mecanismos de *lock* distribuídos;
- **Disponibilidade e tolerância a falhas -** Selecionando a tolerância a falhas e disponibilidade cede-se a consistência de forma a ganhar resolução de conflitos. Tem como aplicações práticas, *DNS* e *Caching* de *Web*;

2.4 O movimento BigData

BigData é um termo que está muito em voga nos dias de hoje mas é difícil encontrar uma definição em concreto. Observando algumas das marcas com bastante relevo na gestão de dados é possível chegar a uma definição com base em (Chris Eaton, 2012) e (Oracle, 2012) : *BigData* é um termo utilizado para descrever o enorme volume de dados que não pode ser processado com as ferramentas tradicionais a que as empresas hoje em dia têm acesso. Um

exemplo da aplicação deste termo é o Google que segundo os seus termos de utilização[6] recolhe dos seus utilizadores as seguintes informações:

- Informações pessoais (número de telefone, cartão de credito);
- Informações de *hardware* (modelo, versão do sistema operativo);
- Registo das pesquisas feitas e do endereço IP usado;
- Informações de localização, etc...;

Este movimento, não surgiu do nada mas ganhou bastante relevo quando as seguintes tecnologias começaram a ser desenvolvidas (Morgenthal, 2012):

- Baixo custo dos dispositivos de armazenamento que então obrigavam as empresas a armazenar apenas a informação essencial.
- Virtualização que aumentou a possibilidade de virtualizar quase todo o tipo de *hardware* necessário.
- *Cloud computing* que proporcionou às empresas uma solução de computação elástica, à medida das suas necessidades.
- Bases de dados NoSQL que já são desenhadas para lidar com largos volumes de informação em tempo útil e com suporte para processamento distribuído.

Depois de ter sido apresentada a definição do movimento *Big Data* e as tecnologias que impulsionaram este movimento, de seguida serão apresentados alguns casos de sucesso de uso do *Big Data* segundo (James Manyika, 2011):

- Atração de um maior número de clientes ao usar dados de localização de pessoas.

Ao usar um serviço de envio de *sms* com promoções e descontos para os telemóveis que se encontrem numa determinada área, é possível atrair mais clientes para uma determinada loja. Este sistema já existe, chama-se *ShopAlerts* e está a ser usado por grandes empresas de produtos tais como a Starbucks[7], North Face[8] e Sonic[9].

- Melhorar a qualidade dos serviços públicos (Sistema de Saúde dos Estados Unidos da América e Reino Unido).

Ao ligar os *datasets* de dados clínicos e de custos, o consórcio Kaiser Permanente conseguiu descobrir os efeitos secundários do medicamento Vioxx.

O Instituto Nacional do Sistema de Saúde e Excelência Clinica do Reino Unido, ao analisar custos e efetividade dos novos tratamentos e medicamentos (*datasets*), conseguiu negociar um melhor preço para os medicamentos e tratamentos com as empresas que os produzem e com o Sistema nacional de Saúde do Reino Unido que os distribuiu.).

- Melhoria no serviço de apoio aos desempregados.

A agência alemã Federal do Trabalho que ajuda os desempregados a procurarem emprego, ao criar e processar grandes quantidades de dados (*Big Data sets*) conseguiu perceber quais

[6] http://www.google.pt/intl/pt-PT/policies/privacy/
[7] http://www.starbucks.com/
[8] http://eu.thenorthface.com/tnf-eu-en/homepage
[9] http://www.sonicdrivein.com/home.jsp

os programas de procura de emprego que estavam a funcionar e quais os que não estavam. Ao extinguir os programas com menos sucesso, esta agência foi capaz de reduzir os gastos do orçamento em 10 mil milhões de euros.

2.5 Conclusões

Com vista a encontrar uma alternativa que consiga resolver os problemas acima mencionados, as grandes empresas, tais como a Google e mais recentemente o Facebook, reinventaram as bases de dados NoSQL que já existiam no tempo dos *Mainframes*. NoSQL, segundo (Tiwari, 2011), é um termo que designa as bases de dados de nova geração e que visam resolver os problemas das bases de dados relacionais quando estas lidam com grandes volumes de informação como já foi anteriormente confirmado por (Jacobs, 2009).

Estas bases de dados surgiram em 2004 quando a Google lançou o BigTable que, no fundo, é uma base dados NoSQL que visa colmatar as falhas do modelo relacional em termos de escalabilidade e disponibilidade, como é dito por (Fay Chang, 2006). Assim, ao contrário das bases de dados relacionais, as bases de dados NoSQL estão prontas a serem distribuídas e a escalarem facilmente. Ao mesmo tempo não cumprem as propriedades ACID de forma a representar objetos complexos do dia-a-dia e apresentam facilidades, quer para o programador, quer de gestão, sendo que muitas delas ainda incluem técnicas para simplificar a questão dos conflitos da distribuição da base de dados.

3 Características Bases de Dados NoSQL

3.1 Introdução

As bases de dados NoSQL foram desenvolvidas para serem fáceis de distribuir. Isso implica um diferente modelo de consistência em contrapartida ao comum ACID. Como tal, as bases de dados NoSQL fazem uso da consistência eventual pois este modelo de consistência foi desenvolvido para sistemas distribuídos mas, antes de analisar este modelo, convém referir um dos mais importantes fundamentos teóricos por trás deste modelo.

NoSQL é um conceito vasto que engloba todas as bases de dados que não seguem os princípios das bases de dados relacionais e que estão relacionadas com grandes volumes de dados, mas, qual é ao certo o significado do acrónimo NoSQL e como surgiu?

Observando o acrónimo e, segundo (Tiwari, 2011), conclui-se que é a junção de duas palavras inglesas, "*No*" e "*SQL*". "*No*" significa não em português e "*SQL*" significa *Structured Query Language* que é a linguagem usada para consultar e manipular as bases de dados relacionais. Posto isto, NoSQL deveria significar algo como NãoSQL mas na prática NoSQL significa uma coleção de produtos e conceitos sobre a manipulação de grandes volumes de dados sem usar unicamente *SQL*, ao invés de um produto único que contraria de alguma forma o *SQL*.

Bases de dados não relacionais não são novidade. Segundo (Haugen, 2010), já na década de 1960, a IBM tinha o sistema IMS[10], uma base de dados hierárquica para guardar e gerir as ordens de encomenda do projeto Apollo. Já na década de 70, a InterSystems lança o ISM[11] e surge também o DBM sendo melhorado ao longo da década seguinte até ao aparecimento do Lotus Notes[12] que era uma base de dados de documentos. Nos anos 90, com a revolução dos projetos *Open Source*, surge o GDBM que é um clone do DBM. Surge também nessa mesma

[10] http://www.ibm.com/developerworks/data/library/dmmag/DBMag_Issue408_IMSat40/
[11] http://labsoftnews.typepad.com/lab_soft_news/2010/10/more-on-mumps.html
[12] http://www.ibm.com/developerworks/lotus/library/ls-NDHistory/

década o InterSystems Caché[13] que era uma base de dados híbrida pós-relacional com acesso a objetos *MultiValue* e manipulação direta das estruturas de dados.

Toda esta evolução desde os primórdios das bases de dados até às bases de dados pós-relacionais culminou no desenvolvimento de vários projetos que originaram o verdadeiro acrónimo NoSQL, como por exemplo: Google BigTable[14], CouchDB[15], Neo4j[16], Cassandra[17], Amazon Dynamo[18], Microsoft Azure[19], etc. que através de novos conceitos e modelos de dados tenta mitigar o problema dos grandes volumes de dados.

Essa solução não passa por um só conceito global mas sim por um número elevado de tipos de soluções, isto é, o mundo NoSQL não tem só um tipo de base dados, tem vários. Estes podem ser classificados, segundo (Ferreira, 2010), quanto à arquitetura (distribuídos, não distribuídos), armazenamento (memória, disco, configurável) e modelo de dados (chave/valor, documento, colunas, grafo). O NoSQL também introduziu alguns novos conceitos. De acordo com nosql-database, esses conceitos são: *schema-free*, API simples e consistência eventual (BASE). Todos estes conceitos serão explicados com mais detalhe nas seções seguintes.

3.2 Características principais

3.2.1 Schema-Free

Schema free em português significa livre de esquema mas o que é ao certo um *schema*? Porque é que numa base de dados relacional é necessário um? Quais as vantagens e desvantagens que oferece? E porque é que a sua inexistência pode ser algo de bom? Esta seção visa responder a todas estas perguntas.

Schema por definição, segundo (Rybiński, 1987) e (Tomasz Imielinski and Witold Lipski, 1982), é um conjunto de fórmulas que descrevem a base de dados, relações e constrangimentos codificados na sintaxe dessa base de dados. Esta estrutura é fruto de um processo chamado normalização que significa, de acordo com (Codd, 1970), o processo de simplificação e organização dos campos e das tabelas de uma base de dados de forma a reduzir a redundância dos dados. Numa base de dados é necessário a normalização para garantir que nas tabelas possam ser inseridos dados sem anomalias (inconsistência de dados, redundância, etc.). Por outro lado, à medida que a quantidade de dados aumenta numa tabela, e segundo (Jacobs, 2009), a base de dados tende a ficar mais lenta. Então, num esforço para ganhar

[13] http://www.intersystems.com/cache/whitepapers/hybrid.html
[14] http://research.google.com/archive/bigtable.html
[15] http://couchdb.apache.org/
[16] http://neo4j.org/
[17] http://cassandra.apache.org/
[18] http://aws.amazon.com/dynamodb/
[19] https://www.windowsazure.com/en-us/

rapidez de leitura/obtenção de dados, segundo (Schrage, 2002), abdica-se da normalização. Ao obter-se esta rapidez também se ganha redundância nos dados e uma maior interdependência entre os mesmos. No entanto, ainda existem alguns problemas subjacentes à normalização ou à obrigação de ter um *schema*: A necessidade de, *A priori,* todos os dados obedecerem de alguma forma àquela estrutura e às modificações que por mais pequenas que sejam podem obrigar a uma restruturação tão grande que dê origem à restruturação da aplicação que usa a base de dados.

Após ter sido definido o que é um *schema* é necessário explicar o que significa *schema free*. *Schema* free não é nada mais do que ter uma base de dados que não tem uma estrutura fixa, isto é, ter uma base de dados que não obriga a que os dados obedeçam à rigidez de um *schema* definido *A Priori*. Passa a existir então uma base de dados não relacional mais relaxada que armazena dados não estruturados em qualquer formato sem uma normalização ou definição rígida prévia com um desempenho aceitável para as grandes exigências do *Big Data*.

As desvantagens desta característica que as bases de dados NoSQL partilham, são, de acordo com (Stainer, 2010): ao não ter um *schema* é possível ter dados redundantes ou até incoerentes; os dados passam a estar menos organizados. A garantia das propriedades ACID deixa de ser válida devido à não existência de um *schema* e disso deriva um modelo de consistência diferente, BASE, já abordado anteriormente que garante uma consistência eventual ao invés de uma consistência imediata como nas bases de dados relacionais. No entanto, todos estes "males", por assim dizer, são necessários porque, de acordo com (Brewer, 2000), autor do Teorema CAP que já foi analisado anteriormente, obriga a abrir mão, neste caso, da Consistência dos dados para ganhar em Tolerância a falhas, alta disponibilidade e distribuição dos dados - requisitos inerentes a grandes volumes de dados.

3.2.2 API Simples

O acrónimo *API* significa Interface de Programação Simples que na verdade são um conjunto de funções que um *software* disponibiliza para ser usado por outro (Palmer, 2010). Então, uma API Simples consiste num conjunto de funções que são simples. Mas será mesmo assim? Ao visitar o *site* nosql-database é possível encontrar uma extensa lista de produtos NoSQL e ao analisar os produtos mais sonantes é possível chegar a este quadro:

Tabela 1 – *API's* usadas por alguns *softwares* NoSQL de baseado em nosql-database

Produto	**API/Protocolo**
Hadoop/HBase	Java sem discriminar o método de acesso
Cassandra	Thrift/?
HyperTable	Thrift
Mongo	BSON sem discriminar o método de acesso
CouchDB	JSON/REST
RavenDB	.net/JSON
Azure Table Storage	.net/REST
RIAK	JSON/REST
Chrodless	Java/protocolo interno
Neo4J	Java/REST

Ao analisar a tabela é possível ver que as *API's* mais populares são: Thrift[20],JSON[21],Java[22] e por fim .net[23]. O protocolo mais usado é o REST, no entanto, a *API* pode também incluir o protocolo, como no caso do Thrift. O Apache Thrift, segundo (Mark Slee, 2007), é uma biblioteca e um motor de geração de código multiplataforma com o objetivo de permitir a comunicação entre múltiplas linguagens de programação. De acordo com (Mark Slee, 2007) e (Prunicki, 2008), a arquitetura é constituída por tipos de dados, protocolos, métodos de transporte e servidores, como mostra a figura da página seguinte:

[20] http://thrift.apache.org/
[21] http://www.json.org/
[22] http://www.java.com/
[23] http://www.microsoft.com/net

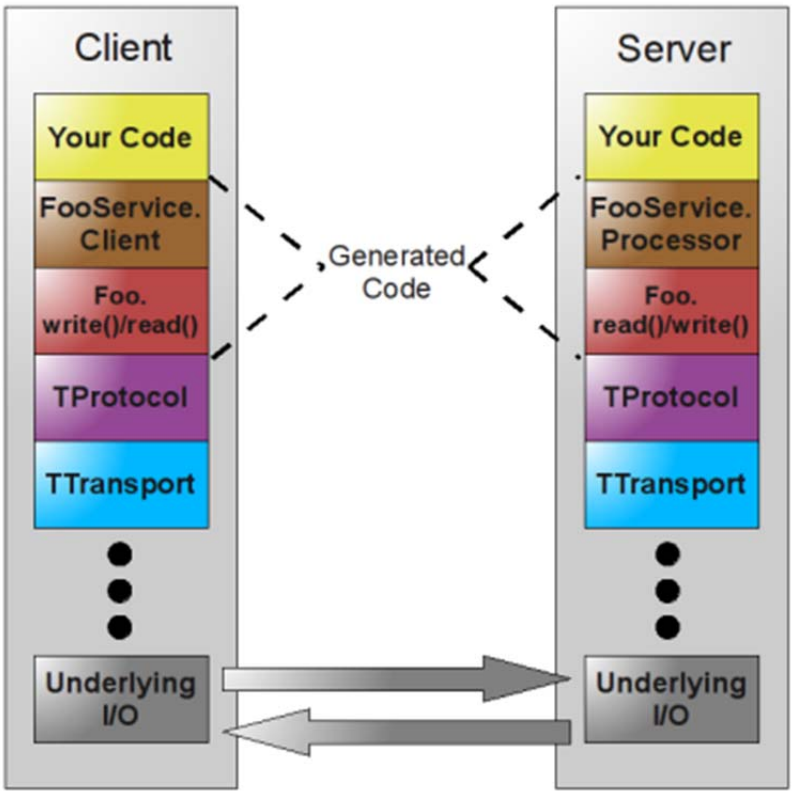

Figura 4 – Conjunto de módulos dentro do Apache Thrift retirado de (Prunicki, 2008)

O Apache Thrift permite gerar, através da junção das peças do conjunto, um cliente e um servidor para a linguagem mais adequada. Sobre este conjunto é possível encontrar algumas análises comparativas com: *Protocol Buffers*, *RMI* e *Rest*. Estas duas análises: (Gupta, 2011) e (Prunicki, 2008), atestam que Thrift tem um melhor desempenho do que *Rest* e *RMI*, no entanto, em alguns casos e devido à sua robustez, o *Protocol Buffers* pode superar o *Thrift*.

JSON (*JavaScript Oriented Notation*) é, de acordo com a página oficial[24], um formato de troca de dados, fácil de entender, de gerar e de analisar. É mais usado nos *softwares* NoSQL orientados a documento porque, de acordo com (Andrew J. Brust, 2011) e (J. Chris Anderson, 2010), estes softwares têm como base *JavaScript* e, ao usar um formato "intrínseco" desta linguagem, permite usar todos os objetos da base de dados como objetos de JavaScript, ou seja, o processo é uniformizado.

REST (*Representational State Transfer*), segundo (Fielding, 2000), é um estilo de arquitetura de *software* orientado ao documento. A ideia principal é usar os métodos do protocolo *HTTP* para gerir os documentos que são identificados sob a forma de *URL*. Daí que para obter um recurso ou uma coleção de recursos se faça com o auxílio do método *GET*. Por sua vez, com o método *PUT*, o recurso ou a coleção pode ser substituído.

[24] http://www.json.org/

Depois de terem sido analisadas as principais *API's*, o termo simples está relacionado com o facto de a *API* ser madura o suficiente e, como tal, já há bastante generalização; como é o caso do *REST*, *HTTP*, *Java*, *.net*, etc., ou com o facto de no site de cada *software* NoSQL existir quase sempre indicações para usar várias *API's* clientes que usam *Thrift* ou outro substituto mas que podem ser usadas com um número infindável de linguagens.

3.3 Tipos de Bases de Dados NoSQL

3.3.1 Par Chave/Valor

Este é talvez o modelo de dados mais importante de todo o mundo NoSQL sendo que os outros modelos derivam deste. Mas antes de se poder explicar ao certo o que é este modelo deve-se rever um tipo de dados presente nas linguagens de programação modernas: *collections* (dicionários, *arrays* associativos, etc.). Consistem em conjuntos de pares chave/valor, isto é, ter uma chave e logo a seguir um valor para ela. Um exemplo simples disto é uma agenda onde a cada dia (chave) corresponde um compromisso. Em *.net*, a título de curiosidade, uma das possíveis implementações seria com recurso a um dicionário e o código correspondente seria:

```
Dictionary<String, String> dicionario_teste = new Dictionary<String, String>();
```

Em NoSQL o modelo de dados chave/valor é igual ao das linguagens de programação consistindo na mesma num identificador seguido de um valor de tipo indiferenciado. De acordo com (Andrew J. Brust, 2011), este modelo é otimizado para ambientes em que seja necessário um bom mecanismo de cache tal como: categorias de produtos, listas de compras, top de produtos mais vendidos etc.. De seguida, é apresentada uma figura que mostra um exemplo mais realista de uma base de dados chave/valor:

Figura 5- Exemplo de Base de Dados Chave/Valor (Andrew J. Brust, 2011)

Ao observar este exemplo, é possível constatar que os tradicionais conceitos de tabela, coluna e linha foram alterados drasticamente. O conceito de linha mudou um pouco. No modelo tradicional uma linha continha um número fixo de colunas mas neste novo modelo isso deixou de acontecer, pois agora uma linha é um conjunto de pares chave/valor mas que não necessitam de ter os mesmos pares. O conceito de tabela ainda é o mesmo só que agora é um conjunto de linhas que tem vários pares chave/valor. Como agora é possível uma linha não ter o mesmo número de pares chave/valor, o conceito de coluna não existe porque existiriam posições em branco que não seriam benéficas para este modelo. As relações também deixaram de existir porque não deixou de ser obrigatório que uma linha tenha o mesmo número de colunas e, como tal, poderia acontecer que a coluna que fazia relação não existisse. Contudo, a obtenção da capacidade de armazenamento de grandes quantidades de dados foi o principal fator que levou à abolição das relações.

3.3.2 Documento

Um outro modelo de base de dados NoSQL é o de Documento em que cada entrada na base de dados corresponde a um documento. Estes documentos normalmente são decompostos num identificador e num valor. Nesse valor é possível introduzir mais identificadores seguidos de valores como por exemplo:

```
PrimeiroNome: “Ricardo”, Morada=”Mancelos”
```

Este modelo de dados codifica os documentos no formato em cima descrito com *XML*, *JSON*, *BSON* e os formatos binários mais conhecidos como *PDF*, *DOC*, *XLS*, etc. De forma a tornar a base dados mais funcional, os fabricantes não se esqueceram de suportar os anexos e suportar em alguns casos um sistema de controlo de versões para, se necessário, restaurar versões mais antigas. Com este modelo é também possível identificar um documento através de um *URL*, ou seja, através do uso de funções de *rendering* e um pouco de *REST*. De acordo com (J. Chris Anderson, 2010), é possível integrar um *website* inteiro dentro da própria base de dados, centralizando assim toda a aplicação.

Agora analisando um exemplo mais real, uma das muitas bases de dados NoSQL orientada a documentos, é o CouchDB. Tem todas as características acima mencionadas e mais algumas nomeadamente uma *API REST*. Na página seguinte, vai ser mostrado um exemplo da estrutura de um documento criado no CouchDB:

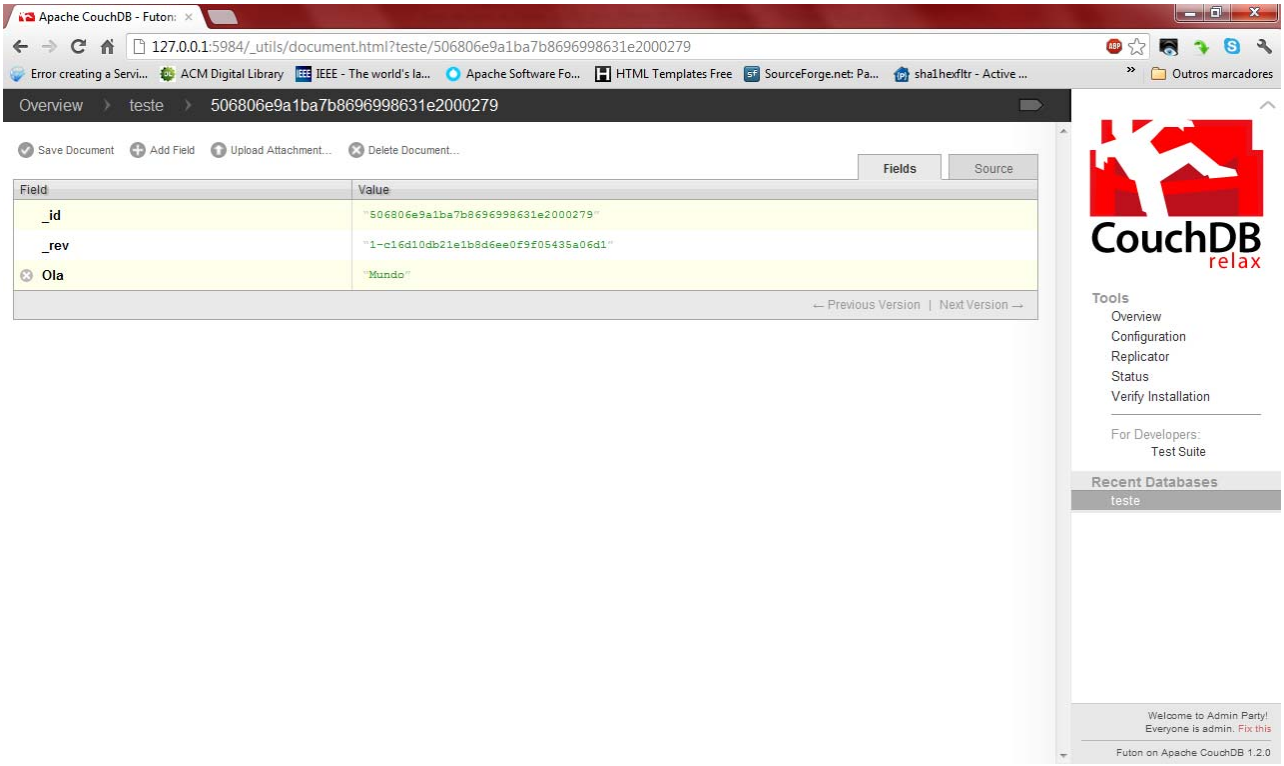

Figura 6 - Exemplo da estrutura de um documento criado no *CouchDb* pela interface Web

Como se pode observar, a estrutura do documento está em *JSON* e por omissão o *CouchDB* já cria um campo "_rev" com vista a um controlo de versões.

3.3.3 Colunas

Como já foi referido anteriormente, o modelo chave/valor é um modelo do qual alguns outros modelos derivam, o modelo de colunas é um exemplo disso mesmo. É uma evolução do modelo de chave/valor. Chegou-se à conclusão que ter um conjunto de pares chave sem um critério de organização suficientemente pequeno poderia ser um incómodo. Então como forma de agrupar os pares chave/valor, o Cassandra, segundo (Hewit, 2010), estipulou que os pares chave/valor (colunas) poderiam ser agrupados em linhas (*rows*). Também de forma a organizar ainda mais o modelo, surge o conceito de superfamília de colunas que na prática é par chave/valor constituído por um *id* e um conjunto de colunas. Ao organizar assim o modelo, ele deixou de ser totalmente *schema-free*, mesmo que na estrutura mais simples o par chave/valor possa ser completamente livre, os formatos hierárquicos como as superfamílias de colunas têm de ser declaradas. A razão disto é porque se se mantivesse a completa liberdade ao fazer alterações nas superfamílias de colunas poder-se-ia perder alguma funcionalidade importante para os clientes.

Pensando agora num exemplo mais real, como o do Facebook, vai ser mostrado na página um modelo de dados exemplo do Apache Cassandra:

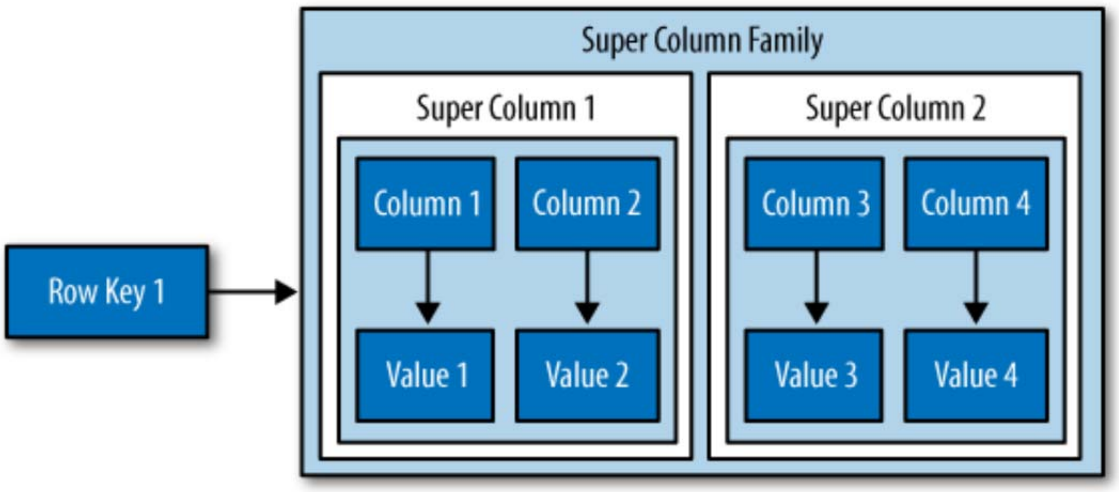

Figura 7- Modelo de dados exemplo do Apache Cassandra retirado de (Hewit, 2010)

Como se pode ver, os conceitos de coluna e superfamília de colunas existem associados a uma linha.

3.3.4 Grafo

Este é um modelo de base de dados NoSQL que se baseia na teoria dos Grafos. De acordo com (Allen, 2010), faz uso dos conceitos de vértices e ramos para representar toda a informação da base de dados em vez de usar tabelas ou pares chave-valor. Como principal vantagem às bases de dados relacionais, oferece a sua estrutura em grafo pois nela podem ser representados os objetos e propriedades com os quais os programadores estão familiarizados a trabalhar. Para além disso, a teoria dos grafos já é bastante madura e como tal já existem bastantes implementações dos algoritmos para percorrer grafos à procura de informações de forma bastante otimizada. Como tal, há mais facilidade em obter a informação e é possível adicionar e remover vértices ou ramos sem grandes conflitos. Este modelo parece encaixar muito bem no caso das redes sociais pois para, por exemplo, descobrir os amigos de um utilizador é mais fácil percorrer um grafo de associações do que fazer uma *query* a, possivelmente, mais do que uma tabela para os descobrir.

Na página seguinte, vai ser mostrado um exemplo de uma base de dados em grafo:

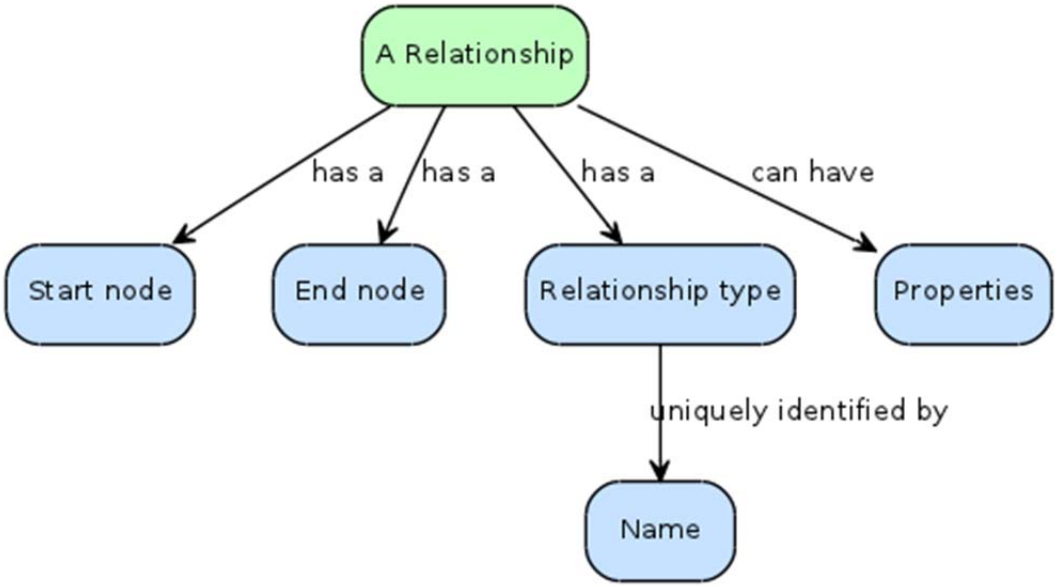

Figura 8 – Exemplo de uma base de dados em Grafo do Neo4j retirado da documentação do Neo4j[25]

Como é possível observar pela figura, os conceitos de vértices e ramos existem, da mesma forma que existem nos grafos, sendo possível atribuir valores aos ramos. É também o único modelo de base de dados NoSQL que permite controlar de uma forma simples as relações entre os vértices do grafo ao atribuir valores aos ramos, da mesma forma como se atribui pesos aos ramos de um grafo.

3.3.5 Comparação entre os diferentes tipos de Bases de Dados NoSQL

3.3.5.1 Chave/Valor

A maior vantagem deste modelo é a sua simplicidade de utilização e implementação de acordo com (Bernadette Farias Lóscio, 2011), pois é uma coleção de chaves com um único valor. O autor (Seeger, 2009) vai ainda mais longe ao especificar que este modelo é tão simples que os programadores demoram menos tempo a aprendê-lo porque já estão habituados aos conceitos de Dicionários e *Hashes*. De acordo com (Bernadette Farias Lóscio, 2011) este modelo é muito rápido no acesso aos dados e, segundo (Seeger, 2009), deixam de existir as *queries* complexas do SQL e deixa de haver a necessidade de fazer o *tunning* às *queries* como no modelo relacional.

Uma das principais desvantagens, de acordo com o *post* seguinte do blog SpecIndia[26], é que este modelo de base de dados deixa cair a integridade relegando-a para a aplicação. Outra

[25] http://docs.neo4j.org/chunked/milestone/index.html
[26] http://blog.spec-india.com/key-value-databasedoc

desvantagem, de acordo com (Bernadette Farias Lóscio, 2011) e com (Seeger, 2009), é que não permite consultas complexas, apenas *get's* e *set's* na sua forma mais purista. De acordo com (Seeger, 2009), existe ainda mais uma limitação, que é o facto de se perder a capacidade de fazer *queries ad-hoc* , que são as *queries* com um determinado fim, por exemplo: encontrar o nome do empregado que fez mais vendas.

3.3.5.2 Colunas

As vantagens deste modelo, de acordo com (Bernadette Farias Lóscio, 2011) e (Abadi, 2010) são: a consistência é forte, em que todas as leituras paralelas são vistas pelos processos paralelos pela mesma ordem (sequencial); numa *query* é possível escolher quais as colunas a retornar ao invés de todas; como em todas as linhas existe um *timestamp* é possível ter noção das várias versões da linha; este modelo está otimizado para as leituras e é disperso o que implica que as linhas pertencentes à mesma família de colunas possam ter diferentes colunas. De acordo com (Lehmann, 2012) este modelo tem ainda como vantagem o facto de ser desenhado para obter o maior desempenho pois suporta nativamente *views* persistentes e é muito eficiente na agregação de algumas colunas mas de várias linhas.

O principal ponto fraco deste modelo, de acordo com (Abadi, 2010), é o facto de não suportar relações, o que implica que os dados de cada pesquisa que se faça terem de estar obrigatoriamente na mesma família de colunas pois não é possível juntar os dados de uma família de colunas com outra. Segundo (Lehmann, 2012), este modelo oferece opções muito limitadas na filtragem dos dados de uma *query*. Requer um esforço de manutenção adicional ao apagar e atualizar os dados e é menos eficiente que um sistema orientado a linhas no acesso a um número elevado de colunas da mesma linha.

3.3.5.3 Documento

As grandes vantagens deste modelo, de acordo com (Bernadette Farias Lóscio, 2011), são: suporte a conjuntos de conjuntos de pares chave/valor, ao contrário do modelo chave/valor em que a base de dados por si só é um conjunto de pares chave/valor. Não depende de um esquema (*schema*) rígido e como tal suporta dados não estruturados, de acordo com (Lehmann, 2012). Este modelo é suficientemente flexível para suportar a adição de pares chave/valor em qualquer altura sem grandes repercussões. Ainda de acordo com (Lehmann, 2012) e (Henricsson, 2011), este modelo tem uma grande simplicidade permitindo assim aos utilizadores compreender facilmente o documento e aos programadores obter os documentos sob a forma de *arrays* associativos. Têm uma estrutura familiar à do *Lotus Notes* e a modelação das *queries* é simples e natural através do uso de técnicas *Map/Reduce* de acordo (Lehmann, 2012).

Os pontos menos fortes deste modelo e, de acordo com (Lehmann, 2012), são: uma necessidade de *hardware* mais poderoso por comparação com o modelo orientado a colunas pois as *queries* são mais dinâmicas. O futuro de alguns projetos que implementam este modelo é incerto, como no caso do CouchDB em que os programadores começaram a mudar para um outro produto. Outros projetos como o MongoDB não fornecem soluções para o

armazenamento de dados *off-line*. De acordo com (Henricsson, 2011) há uma clara falta de referenciação entre documentos que pode levar a uma redundância dos dados.

3.3.5.4 Grafo

Os principais pontos fortes deste modelo, segundo (Bernadette Farias Lóscio, 2011), são: um maior desempenho, por comparação com uma base de dados relacional, em consultas complexas que envolvam muitos *joins*, como por exemplo: "Quais as cidades visitadas anteriormente?", porque a estrutura de relações dos grafos simplificam em muito estas *queries* complexas, ao ponto de serem melhores do que numa base de dados relacional. Este modelo foi otimizado para o alto desempenho. A modelação de redes de dados é muito compacta, de acordo com (Lehmann, 2012). Finalmente de acordo com (Henricsson, 2011), têm uma facilidade elevada no armazenamento de informações sobre relações como por exemplo: redes sociais, itinerários, etc., quando comparadas com uma base de dados relacional.

A principal desvantagem, como se pode concluir ao analisar as vantagens, é o facto de este modelo não encaixar muito bem no modelo tradicional de negócio com tabelas. Existe também um problema ao fazer *queries* que implica que se percorra o grafo todo para responder a uma. Não é fácil criar grupos dentro de num grafo.

3.3.6 Usos mais comuns dos modelos de Dados NoSQL

Depois de terem sido apresentados os diversos modelos NoSQL, é necessário especificar quais as características/cenários de utilização para cada um destes modelos.

Num cenário em que sejam necessárias facilidades de *Upgrade*, mecanismos de *Cache* ou armazenar *Blobs* ou ainda com um número elevado de escritas mas um número pequeno de leituras e, de acordo com (Borkar, 2012), o modelo **Key/Value** é um bom candidato.

Numa aplicação que lide com muitos padrões de acesso e um enorme conjunto de tipos de dados que necessite de CRUD e que esteja a processar um *stream* de blocos contínuos em que a consistência não seja importante, o modelo que deve ser empregado é o de bases de dados de **Documento**, devido ao facto de este modelo possuir facilidades de acesso a dados complexos sem *joins*.

Para casos em que a distribuição seja um fator de elevada importância, que haja a necessidade de tipos de dados fluidos, que sejam necessários índices secundários e o conjunto dos dados esteja sempre a crescer, o modelo de **Colunas** deve ser o indicado pois tem uma boa flexibilidade e um *design* distribuído.

Caso a aplicação faça operações semelhantes a uma rede social, que seja necessário a construção de relações de forma dinâmica, em que haja uma elevada profundidade dos *joins* e em que as *queries* sejam de um elevado grau de complexidade, o modelo de **Grafo** é o que mais se adequa porque suporta a navegação rápida entre entidades. A estrutura em Grafo

facilita por si só a construção de relações e a não obrigatoriedade de um *schema* permite fazer alterações às relações quando for necessário.

3.4 Sharding e ServerFarm

Ao longo desta tese, já foi discutido o termo de escalabilidade horizontal ou na terminologia inglesa *sharding***.** A definição de *sharding*, segundo (Charles Bell, 2010), consiste em dividir os dados em fragmentos (*shards*) independentes por nós para que cada cliente que necessite dos dados seja redirecionado para o nó que os contém. De acordo com (Baron Shawrtz, 2008), é o processo de divisão dos dados em pequenos bocados (*shards*) e o seu pós armazenamento em diferentes nós. Da mesma forma, (Warden, 2011) defende que é o esquema usado para decidir em que máquina (nó), um dado pedaço de informação de uma base de dados vai residir. Depois de analisar estas três definições pode-se concluir que *sharding* traduz-se na forma como a informação de cada tabela em cada nó vai ser dividida, como mostra a figura seguinte:

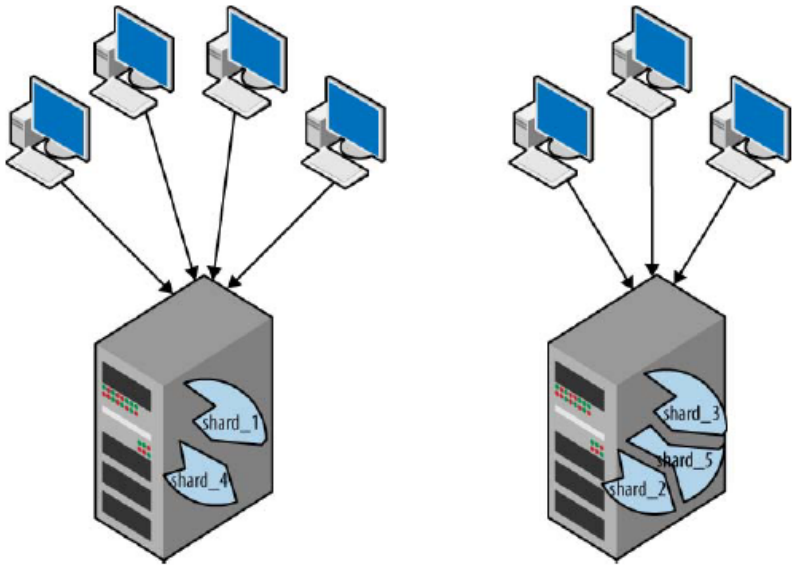

Figura 9 – Exemplo de *Sharding* retirado de (Charles Bell, 2010)

Há diversas razões para fazer o *sharding* à base de dados, de acordo com (Charles Bell, 2010), as mais comuns são:

- Colocar os dados geograficamente mais perto do utilizador.
- Reduzir o tamanho do *Dataset* de forma a facilitar a pesquisa.
- Equilibrar a carga do sistema ao ser possível dividir ainda mais os fragmentos que estejam a ser muito utilizados.

Mas existem também algumas desvantagens, entre elas a mais evidente é a mudança de um *dataset* que que vem de uma base de dados não distribuída para um que venha de uma base da dados distribuída em *shards*. Segundo (Baron Shawrtz, 2008), existem também algumas questões na mudança dos fragmentos de um nó para o outro.

Os tipos de *sharding* conhecidos segundo (Charles Bell, 2010) resumem-se a dois:

- *Sharding* estático, em que os dados de uma tabela são divididos em intervalos ou conjuntos restritos, como pode ser observado na figura seguinte:

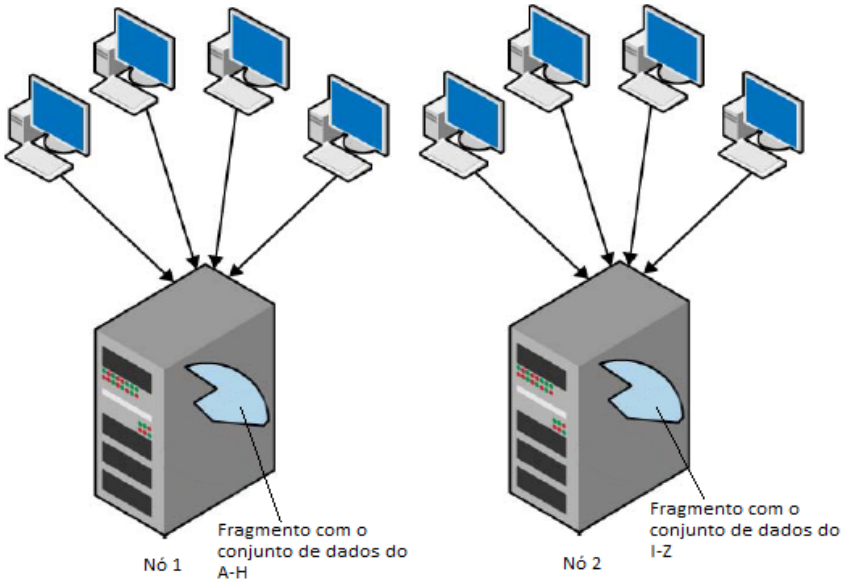

Figura 10 – Exemplo de *Sharding* estático por conjuntos adaptado de (Charles Bell, 2010)

- *Sharding* dinâmico, em que os dados de uma base de dados são divididos pelos nós e depois existe um nó que contém os mapeamentos de dados (dicionário), como mostra a figura seguinte:

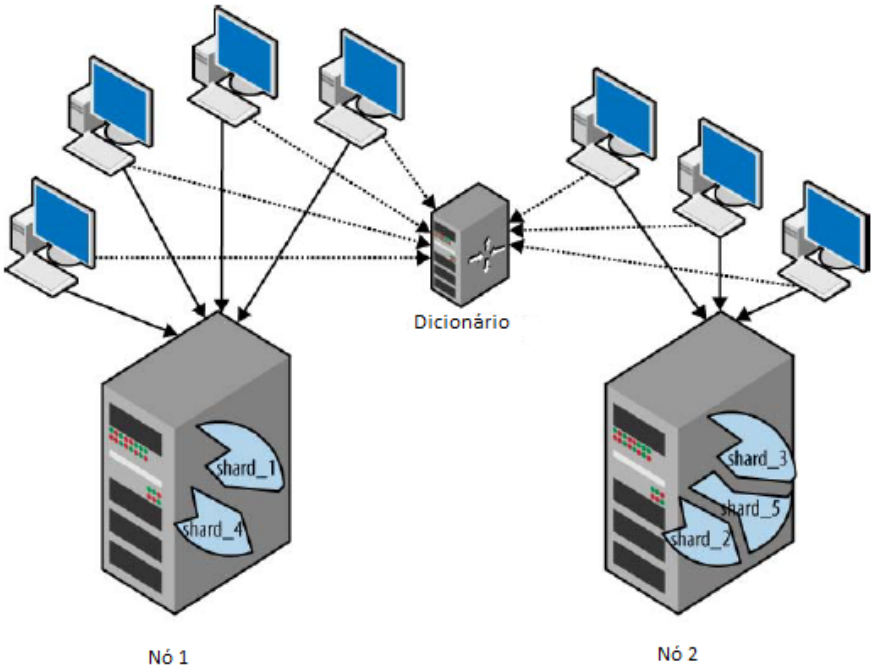

Figura 11 – Esquema de *Sharding* dinâmico adaptado de (Charles Bell, 2010)

Resumindo, são 3 os passos para fazer *sharding*, de acordo com (Charles Bell, 2010): a criação de uma *partition key*, que é um identificador da forma como as tabelas são partidas; uma função de partição que, com base na *partition key, separa* os dados e, um mapeamento de correspondência entre fragmentos e nós que, no caso de um *sharding* estático, pode ser feito pela função de partição e que no caso do *sharding* dinâmico pode ser feito com recurso à adição de uma tabela na base de dados que guarda os mapeamentos.

3.5 Map/Reduce

3.5.1 Definição

Map/Reduce é uma tecnologia que nasceu do Google para dar resposta à necessidade de tratamento do enorme volume de dados que o Google tem de tratar diariamente. Segundo (Ghemawat, 2004), é uma *framework* de programação para processar *datasets* de grande porte com *hardware* "normal" e de acordo com (White, 2011), é um modelo de dados para processamento paralelo. Ao analisar estas duas definições, é possível concluir que Map/Reduce não é mais do que uma forma de processar grandes volumes de informação.

O Map/Reduce assenta em dois grandes conceitos:

- Uma função de *Map*, que separa os grandes volumes de dados em pedaços mais pequenos;

- Uma função de *Reduce*, que por sua vez processa e agrupa todos os pequenos fragmentos dos dados.

Na prática e, segundo (Ghemawat, 2004), o que aconteceu na implementação do Google foi, pegar num *cluster* "vulgar" e escolher um nó como sendo o *Master* que vai receber o *dataset* gigantesco; depois este vai dividir os dados por algumas máquinas que tomaram o papel de *map workers* e executaram a função de *map* aos dados que lhe foram atribuídos. De seguida os *map workers* notificaram o *Master* de que a função de *map* está concluída para que depois o *Master* delegue num ou mais *reduce workers* o trabalho de obter os dados previamente mapeados pelo *map worker* e lhes aplique a função de *reduce*. Findo este processo, os dados estão processados. Para melhor ilustrar este processo, na página vai ser apresentada uma imagem com mais algum detalhe que sumariza todo este processo:

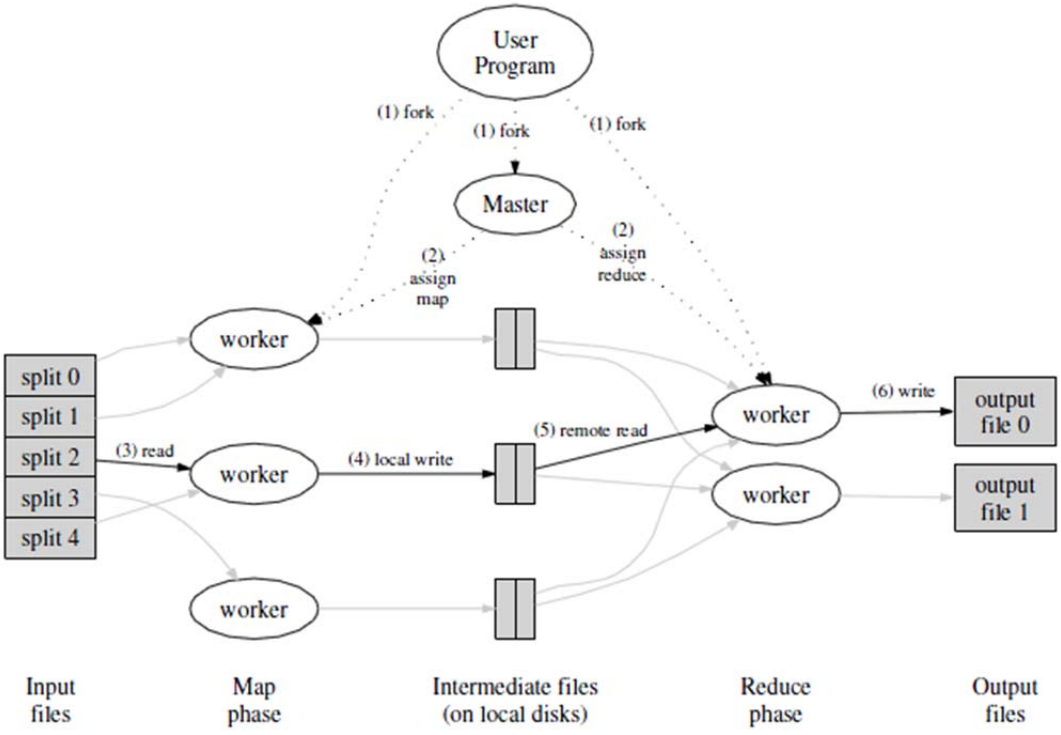

Figura 12 – Diagrama de execução de um processo de *Map/Reduce* no *cluster* do Google retirado de (Ghemawat, 2004)

3.5.2 Queries no Map/Reduce

Anteriormente foi feita uma introdução teórica ao tema do Map/Reduce. Dela pode-se concluir que o Map/Reduce assenta na função de *Map* e na função de *Reduce*. Em seguida, vai ser apresentado um exemplo de como fazer essas funções e um exemplo de uma *query* de um produto NoSQL que usa *Map/Reduce*. De acordo com (Ghemawat, 2004), a função de *Map* recebe como parâmetros um par chave/valor em que a chave é algum indentificador relevante para o processamento que vai ser feito e o valor são os dados em que vai ser feita a emissão do par chave/valor intermédio para a lista de pares chave/valor que esta função vai devolver. Posteriormente, a função de *Reduce* recebe como parâmetros uma chave e a lista de pares chave/valor retornados pela função de *Map* para posteriormente devolver o resultado final do processo sob a forma também de uma lista. No exemplo em Pseudo-código da página seguinte, que foi retirado do artigo de (Ghemawat, 2004), é feita uma contagem de palavras numa coleção de documentos:

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
EmitIntermediate(w, "1");
reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
result += ParseInt(v);
Emit(AsString(result));
```

Código 1 - Contagem das palavras de um documento em *Map/Reduce* de acordo com (Ghemawat, 2004)

Como é possível observar no pseudo-código, toma como argumentos o documento em si e os conteúdos. Faz sentido que neste caso o nome do documento seja a chave e, o valor seja o conteúdo do documento. Depois, para cada palavra, a função conta uma ocorrência e acrescenta este resultado à lista de resultados intermédios com a função *EmitIntermediate*. A função de *reduce* aceita como argumentos a lista de valores produzidos pela função de *map*, as suas chaves correspondentes e faz o somatório para cada palavra. Posto isto, vai ser apresentado um exemplo mais real, de (White, 2011), que exemplifica uma *query* propriamente dita feita para o Hadoop em Java:

```
public class NewMaxTemperature {
 static class NewMaxTemperatureMapper
 extends Mapper<LongWritable, Text, Text, IntWritable> {
        private static final int MISSING = 9999;
        public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
               String line = value.toString();
               String year = line.substring(15, 19);
               int airTemperature;
```

```
                if (line.charAt(87) == '+') { // parseInt doesn't like
leading plus signs
                        airTemperature = Integer.parseInt(line.substring(88,
92));
                } else {
                        airTemperature = Integer.parseInt(line.substring(87,
92));}
                String quality = line.substring(92, 93);
                if (airTemperature != MISSING && quality.matches("[01459]"))
{                       context.write(new Text(year), new
IntWritable(airTemperature));}}}
 static class NewMaxTemperatureReducer
 extends Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterable<IntWritable> values,
        Context context)
        throws IOException, InterruptedException {
                int maxValue = Integer.MIN_VALUE;
                for (IntWritable value : values) {
                        maxValue = Math.max(maxValue, value.get());
                }
                context.write(key, new IntWritable(maxValue));
        }
 }
 public static void main(String[] args) throws Exception {
        if (args.length != 2) {
                System.err.println("Usage: NewMaxTemperature <input path>
<output path>");
                System.exit(-1);
        }
        Job job = new Job();
        job.setJarByClass(NewMaxTemperature.class);
        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));
        job.setMapperClass(NewMaxTemperatureMapper.class);
        job.setReducerClass(NewMaxTemperatureReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);
        System.exit(job.waitForCompletion(true) ? 0 : 1);}}
```

Código 2 – Exemplo de uma *query* em *Map/Reduce* de acordo com (White, 2011)

Este exemplo visa processar dados meteorológicos processando as temperaturas referentes a cada ano que estão organizadas em ficheiros por estação meteorológica e divididos em pastas por ano. Logo, a função de *Map* vai simplificar os nossos dados, extraindo o ano e a temperatura das linhas do ficheiro e, a função de *Reduce*, vai encontrar a temperatura máxima de cada ano. Ao analisar este exemplo é possível ver que o básico do Map/Reduce mantém-se, isto é, criar duas funções, uma *Map* e outra *Reduce*, para processar os dados e, de acordo com a *API* do produto que se está a utilizar. A novidade é o método *main* que "lança" a *query* efetivamente. O que ressalta em primeiro plano é que o Hadoop obriga à criação de um *Job* para correr a *query* sendo que depois basta fazer as atribuições das classes que contêm os métodos *Map* e *Reduce*.

4 Casos Reais

Com este capítulo, pretende-se demonstrar de facto onde o NoSQL está a ser usado. Em ordem a isso, foram escolhidos e analisados três exemplos: Yotta[27], uma plataforma de optimização *web*, StudyBlue[28], uma plataforma de estudo *online* e NetFlix[29], um *website* de comércio eletrónico. Com esta análise pretende-se focar o ponto de vista do negócio e ponto de vista da arquitetura construída em cada exemplo.

4.1 NetFlix

A NetFlix é de acordo com o *site* oficial uma empresa de comércio eletrónico que vende subscrições para a visualização (*streaming*) de diversos conteúdos multimédia tais como filmes, programas de televisão etc.. que conta atualmente com mais de 27 milhões de membros.

De acordo com (Anand, 2010), a Netflix só tinha um *datacenter* e notou que o negócio ficava dependente de funcionamento do *datacenter* em questão. Qualquer intervenção no *datacenter* podia interromper o serviço e ter impactos negativos para o cliente. Com aumento dos clientes (subscritores) e a adoção do *streaming*, o referido *datacenter* não chegaria para satisfazer as necessidades de processamento e armazenamento necessárias, a Netflix. Para resolver o problema, a Netflix tinha duas soluções que eram: criar novos *datacenters,* o que implicaria um custo bastante elevado e mais recursos para os gerirem, ou, migrar todos os serviços de rede e de apoio para a *cloud*. A solução adotada foi a da migração para a *cloud* visto que assim a Netflix não gastaria tantos recursos a gerir os *datacenters* e podia concentrar-se mais no seu objetivo de entregar filmes e programas de TV.

[27] http://www.yottaa.com/home
[28] http://www.studyblue.com/
[29] https://signup.netflix.com/global

4.1.1 Ponto de Vista do Negócio

O negócio da Netflix segundo o *site* oficial é fornecer serviços de aluguer de filmes *online*. O aluguer em questão é efetuado através da página da Netflix e depois o cliente pode optar por ver o filme *online* através de *stream* no seu computador pessoal, consola ou dispositivo móvel ou pode ser enviada pelo correio uma cópia do filme para a morada do cliente.

Para além dos serviços mencionados anteriormente e de acordo com (Anand, 2010) , a Netflix fornece ainda serviços de classificação, crítica e recomendação de vídeos, login/gestão de utilizadores e filas de vídeos. Porém, para suportar todos estes serviços, a Netflix tem também serviços internos de gestão de DVD's, faturação e gestão de meta dados do utilizador, tal como o instante onde o utilizador pausou o filme para uma posterior reprodução.

4.1.2 Ponto de Vista da Arquitetura

O fornecedor de *cloud* escolhido pela NetFlix foi a Amazon e decidiu usar dois serviços, o Amazon SimpleDB e o Amazon S3. O motivo pelo qual foram usados estes 2 sistemas de bases de dados foi porque o SimpleDB tem limitações no que toca ao número de pares chave/valor (atributos) que pode guardar numa tabela (item). Para além disso o valor do par não pode exceder o tamanho de 1024 bytes, de acordo com (Anand, 2010). Para os datasets grandes (Big Data), segundo (Anad, 2011), só o log do Tomcat ocupava centenas de Petabytes para situações como essa e para o histórico antigo de alugueres, foi usado o S3 que não tem as limitações do SimpleDB.

A arquitetura da Netflix já com a base de dados NoSQL S3 é ilustrada na imagem da página seguinte:

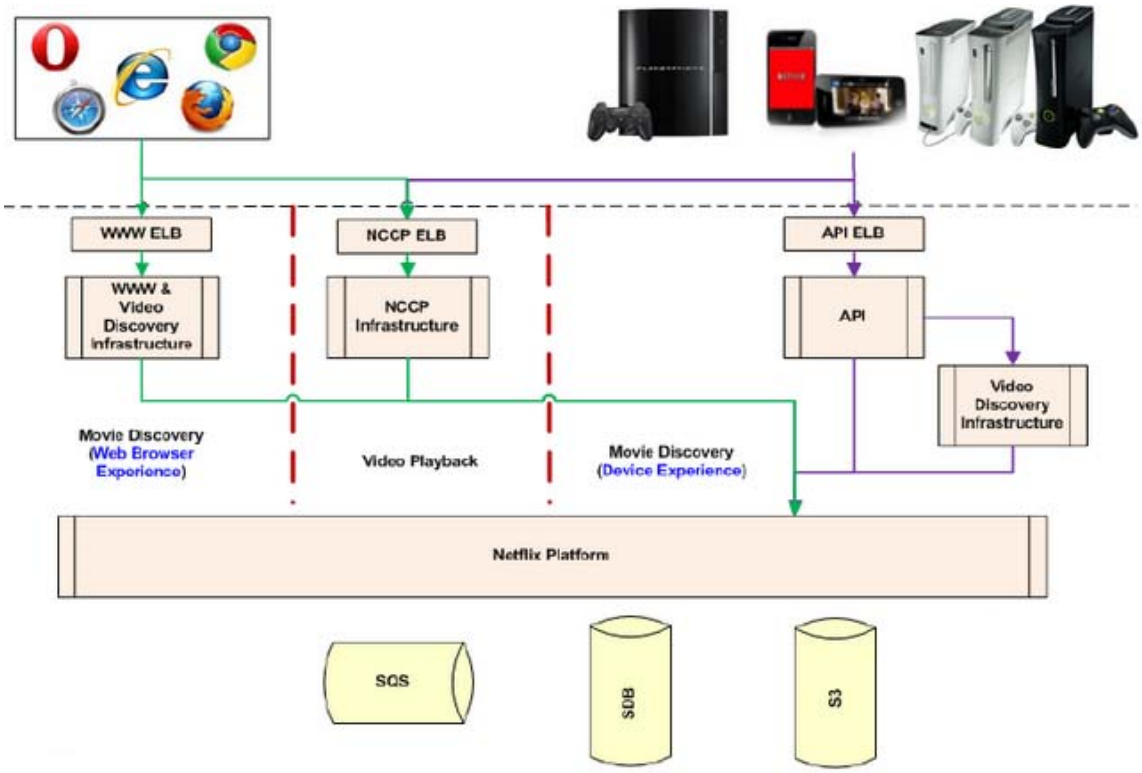

Figura 13 – Diagrama com a arquitetura da Netflix retirado de (Anad, 2011)

Como se pode observar, todos os clientes, inclusive as plataformas móveis, numa primeira fase passam por um balanceador de carga (*ELB – Elastic Load Balancer*) para, distribuir a carga e fazer autenticação. Depois, os clientes chegam à plataforma Netflix através de serviços de descoberta de vídeo. Nos pedidos em que seja apenas necessário a reprodução ou início do vídeo, a *framework* de NCCP trata desses pedidos. Finalmente dentro da plataforma Netflix é possível chegar aos dados estando divididos em três bases de dados: a de filas (*queues*), a do SimpleDB (SDB) e a do S3.

4.2 StudyBlue

A StudyBlue, de acordo com o *website* oficial, é um local que ajuda os estudantes a ter acesso a ferramentas de estudo de alta qualidade. É um local onde os estudantes podem partilhar cartões de estudo (*flash cards*), documentos e onde é possível interagir com pessoas das diversas escolas aderentes. De acordo com (Laurent, 2012), o StudyBlue é um serviço *online* para armazenar, estudar e partilhar os materiais do curso que o utilizador frequenta como sendo o papel digital dos estudantes modernos constituindo assim uma plataforma de estudo/aprendizagem *online*.

A estrutura inicial do sistema, de acordo com (Laurent, 2012), residia na *cloud*, mais concretamente no serviço de computação elástica da Amazon, o EC2, que continha apenas uma base de dados PostgreSQL para armazenar toda a informação relativa aos cartões de

estudo, utilizadores, aulas, documentos, pessoas, escolas, etc. Sendo que a StudyBlue tem um número elevado de cartões que advém do grande número de utilizadores que continua a crescer rapidamente e como a base de dados PostgreSQL não conseguia suportar toda aquela carga, a StudyBlue optou por usar uma base de dados NoSQL, o MongoDB, o qual vai ser abordado mais a frente, para gerir todos os seus dados.

4.2.1 Ponto de Vista do Negócio

Como já foi dito anteriormente e, de acordo com o *site* oficial da StudyBlue e (Laurent, 2012), o StudyBlue é um *site* de partilha de informação estudantil. Daí que o negócio da StudyBlue seja a gestão de toda a informação estudantil. Dentro do *site* e após o registo, é possível criar, editar, apagar e procurar cartões de estudo (*flash cards*), fazer o *upload* de documentos, criar aulas que armazenam esses documentos e procurar e trocar informação com pessoas.

4.2.2 Ponto de Vista da Arquitetura

A base de dados NoSQL escolhida pela StudyBlue foi o MongoDB. Esta escolha foi feita porque, de acordo com (Laurent, 2012), o MongoDB quando comparado com outros produtos NoSQL, mais concretamente com o SimpleDB, Cassandra, CouchDB e o Redis, apresenta uma manutenção mais simples, tem um maior grau de taxa de disponibilidade com as suas réplicas que elegem entre si qual é a réplica *master*, sendo assim possível quando um nó falha juntar um outro nó que fica configurado automaticamente e por fim, o facto de suportar uma distribuição da base de dados horizontalmente (escalamento horizontal) mantendo um bom desempenho de escrita sendo também fácil de acrescentar mais *shards*.

A arquitetura do StudyBlue incluindo já o MongoDB é descrita na figura da próxima página:

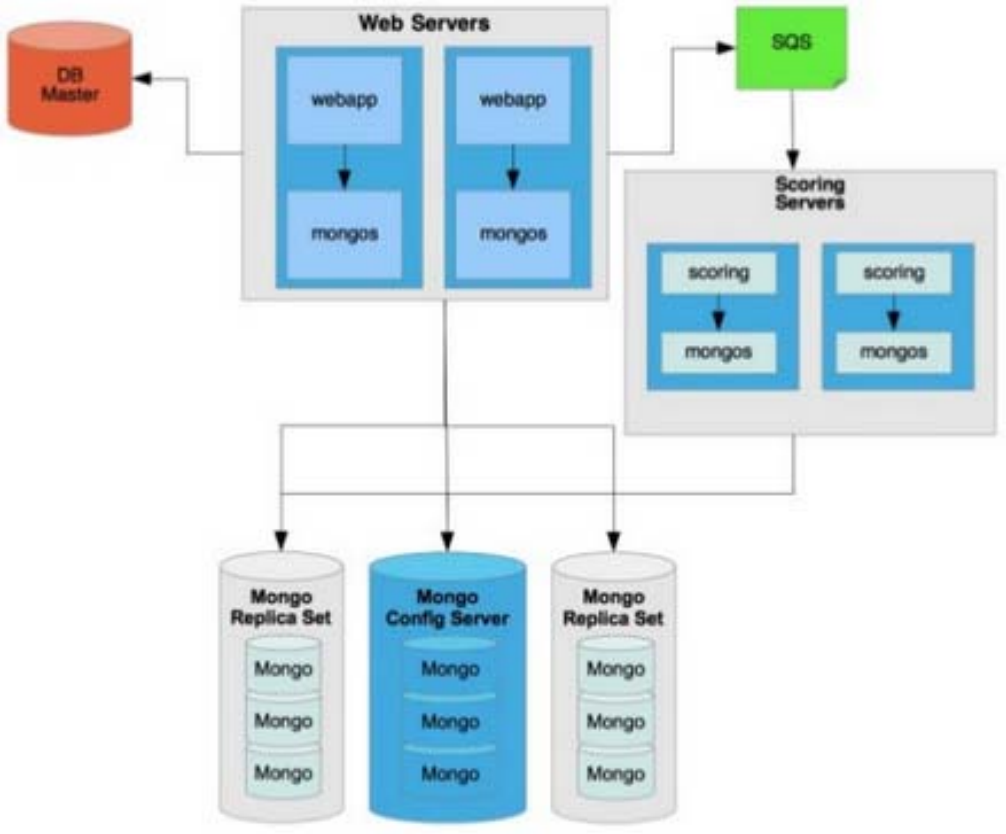

Figura 14 – Arquitetura do StudyBlue de acordo com (Laurent, 2012)

Como se pode observar pela figura, a arquitetura é bastante simples e pode ser resumida a qualquer aplicação, seja ela uma aplicação web alojada num servidor web ou uma operação sobre filas (*queues*). Qualquer pedido passa primeiro pelo processo mongos que, de acordo com o *site* (Merriman, 2009), é um processo que coordena, gere e redireciona os pedidos feitos ao *cluster* para os nós apropriados, de forma a parecer que o *cluster* é apenas um só servidor. Este processo não pode ser corrido em qualquer réplica e, como não tem dados de estado, necessita de comunicar com o servidor de configuração do MongoDB.

4.3 Yottaa

A Yottaa, segundo o *site* oficial, é um serviço de otimização e ou monitorização *web* que visa oferecer a qualquer *site,* uma otimização de forma a tornar o *site* mais rápido, seguro e escalável.

De forma a prestar o melhor serviço possível aos seus clientes, a Yottaa conta com uma tecnologia patenteada que corre sobre uma rede na *cloud* e que interliga mais de 20 *datacenters* que têm como função principal adaptarem-se aos *sites* que monitorizam e/ou otimizam.

4.3.1 Ponto de vista do Negócio

Como já foi dito anteriormente e, de acordo com o *site* oficial, a Yottaa é um fornecedor de serviços de monitorização e otimização de *websites.* O seu negócio assenta em três serviços chave como mostra a figura seguinte:

Figura 15 – Conceitos de negócio da Yottaa retirado do *site* oficial da Yotta

A Yottaa fornece aos seus clientes um serviço que **otimiza** o *website* dos clientes, tornando-o até duas vezes mais rápido, **protege** ao fazer a inspeção do tráfego a ele dirigido, ou seja, bloqueia o tráfego malicioso para deixar o canal de comunicação limpo para o tráfego legítimo e, **monitoriza** constantemente (24/7) alertando o cliente para eventuais problemas que possam surgir e ainda fornece uma visão em tempo real das operações do site em questão.

Com os seus diversos produtos, a Yottaa tenta otimizar o *site* do cliente de forma a que o tempo de carregamento das páginas e o número de pedidos ao servidor diminua. São também aplicadas otimizações específicas para cada *browser*, técnicas de aceleração *HTTPS* e *SSL* e mecanismos de execução de scripts em paralelo. Também com os seus serviços de *firewall*, a Yottaa bloqueia todo o trafego indesejado, ataques à rede, ataques de negação ao serviço e analisa o tráfego em tempo real garantindo a maior disponibilidade possível ao site. Inclui ainda serviços de monitorização que, 24 sobre 24 horas, monitorizam o *site* à procura de eventuais problemas que possam existir, alertando o utilizador em tempo real caso algum problema exista. Este mecanismo de monitorização ainda é capaz de definir automaticamente métricas de desempenho e monitorização bem como adaptar o *site* a essas métricas.

4.3.2 Ponto de vista da arquitetura

Como já foi dito anteriormente, para fornecer os referidos serviços de otimização, a Yottaa conta com uma infraestrutura de *cloud* que interliga mais de 20 *datacenters* distribuídos por vários continentes como mostra a figura seguinte:

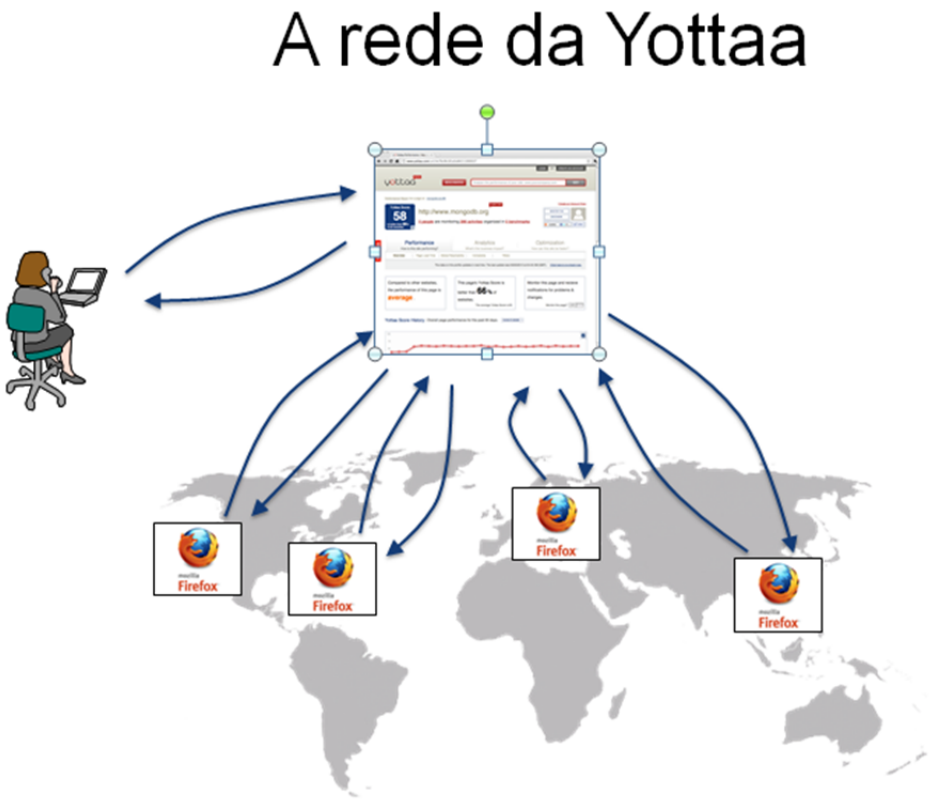

Figura 16 – Rede dos *datacenters* ligados por *cloud* da Yottaa adaptado de (Rosoff, 2010)

Todos estes *datacenters* suportam a tecnologia da Yottaa. Esta tecnologia e, de acordo com (Rosoff, 2010), reúne em si mais de 6000 URL's e 300 amostras por URL que ocupam aproximadamente 1 MB. Estes dados, por sua vez, são visualizados em tempo real sem atrasos. Todos estes dados constituem um grande *dataset* e, para agravar a situação, devido às tarefas de monitorização e dos requisitos de tempo real que obrigam à existência de atualizações à base de dados na ordem das centenas por segundo, uma base de dados relacional como a da figura da página seguinte não o consegue suportar:

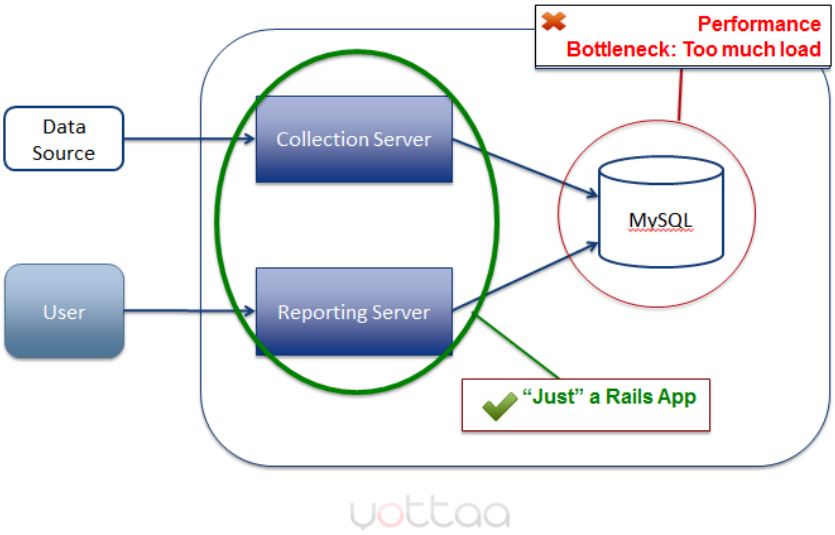

Figura 17 – Arquitetura de uma aplicação Rails da Yottaa retirado de (Rosoff, 2010)

Como é possível observar na imagem, a base de dados MySQL acaba por se tornar um estrangulamento, pois é utilizada por dois servidores que necessitam que a base de dados suporte os pedidos com um ótimo desempenho. Com o MySQL tal não foi possível no entanto foram-na mantendo e foram feitas tentativas com replicação e *sharding* mas, ou se ficava com um estrangulamento no acesso às réplicas ou era necessário, ao fragmentar a base de dados, modicar muito o código de acesso à base de dados.

Foi então que a Yottaa decidiu usar o MongoDB e a arquitetura ficou com o problema do escalamento resolvido como mostra a figura seguinte:

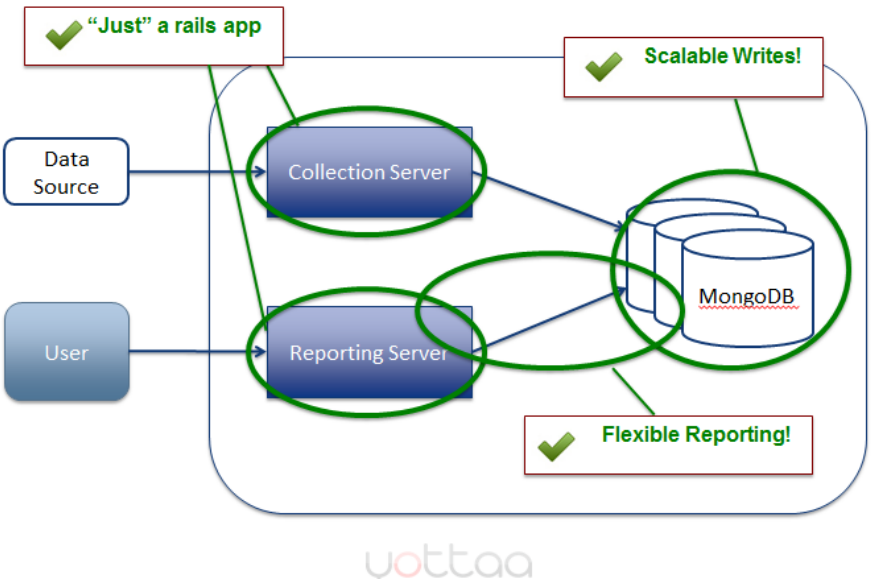

Figura 18 – Arquitetura com o MongoDB retirado de (Rosoff, 2010)

Finalmente com esta arquitetura, a Yottaa consegue facilmente escalar a base de dados só por adicionar um servidor.

5 Desenvolvimento de aplicações usando bases de dados NoSQL

5.1 Sistemas gestores de base de dados NoSQL

Após terem sido apresentados os vários modelos de bases de dados NoSQL e, para que se tenha uma visão completa, é necessário fazer uma análise comparativa a todos os modelos. No entanto, como o tema ainda é recente, não existe uma bibliografia que faça uma comparação desse género. Como tal, foi decidido, em primeiro lugar, analisar dois *softwares* de cada modelo para depois se obter as características comuns e generalizar.

Esta análise vai expor as vantagens e desvantagens de cada modelo baseadas na página do fabricante, nos guias e manuais de instruções que existem sobre cada *software* e em alguns artigos que fazem a comparação de um destes produtos com uma base de dados relacional.

Os *softwares* de base de dados NoSQL do modelo par chave/valor escolhidos foram: Microsoft Azure Table Storage[30] e Redis [31]; do modelo documento os escolhidos foram: CouchDB [32] e MongoDB[33]; do modelo de colunas foram: Cassandra[34] e Hadoop/HBase[35]; do modelo de grafo foi escolhido o Neo4j[36].

As razões que estiveram por trás da escolha destes *softwares* foram tanto a documentação existente, na qual apenas o Neo4j fica um pouco atrás, tanto as empresas que na realidade usam estes softwares (de acordo com o site de cada produto):

[30] https://www.windowsazure.com/en-us/
[31] http://redis.io/
[32] http://couchdb.apache.org/
[33] http://www.mongodb.org/
[34] http://cassandra.apache.org/
[35] http://hadoop.apache.org/
[36] http://neo4j.org/

- Microsoft Azure Table Storage: Nasa.
- Redis: Blizzard, Yahoo, Flickr.CouchDb: BBC, e aplicações do Facebook.
- MongoDb:SourceForge,MTV.
- Cassandra: Facebook,Twitter,Reddit .
- Hadoop/HBase:Adobe,AOL.
- Neo4j: Adobe,Cisco, Deutsche Telekom (de acordo com (Howard, 2012)).
- CouchDb:CreditSuisse,BerrySky (de acordo com (CouchDB, 2012))

5.1.1 Microsoft Azure Table Storage

O Azure Table Storage é uma dos serviços da Microsoft para armazenamento de dados e, de acordo com (Andrew J. Brust, 2011), consiste numa base de dados NoSQL na *cloud*.

As principais vantagens, de acordo com (Andrew J. Brust, 2011), são: concorrência otimista, que é a capacidade de a base de dados não fazer lock aos recursos que estão a ser atualizados, escalonamento automático, isto é, de acordo com (Haridas, 2009), o Microsoft Azure Table Storage é suficientemente autónomo para detetar partições (partes) de uma tabela que estão a ter muitos acessos e dividir a tabela, isolando numa máquina a partição da tabela que está a ter mais acessos e noutra máquina as outras partes da tabela que têm menos acessos. No entanto, esta automatização implica a escolha acertada da *partition key*.

A principal desvantagem é o facto de não suportar índices secundários, excluindo assim a capacidade de filtrar as entidades por valores das colunas ao invés das penosas análises de linhas, segundo (OakLeaf Systems, 2010). Um outro ponto que é importante referir, é que, de acordo com (Williams, 2010), com toda esta automatização de operações acaba-se por perder algum controlo para o lado da Microsoft. De facto, a perda de controlo pode ser um requisito muito negativo para algumas das empresas mas dependendo das necessidades de cada empresa.

5.1.2 Cassandra

As vantagens deste *software*, de acordo com (Hewit, 2010) são: a escalabilidade, que permite adicionar ou remover um nó sem perturbar o serviço; um maior grau de taxa de alta disponibilidade e tolerância a falhas, devido ao facto de distribuir a base de dados por várias réplicas, aumentando assim o número de réplicas da base de dados. Como a base de dados passa a estar distribuída por varias réplicas, a remoção de uma delas em nada interfere com o desempenho das outras. É possível ainda selecionar o nível de consistência especificando o quão forte deve ser a consistência e para isso é possível alterar os valores do nível de consistência e o fator de replicação da base de dados. Por último, é orientada às colunas, o

que significa que cada linha pode ter colunas diferentes, sendo assim útil quando os dados não seguem uma estrutura fixa.

As desvantagens deste *software*, de acordo com (Barber, 2010), são: não suporta *joins* e apenas tem chaves de indexação, o que implica não ter relações entre famílias de colunas. Obriga assim a que só se possam pesquisar dados dentro de uma família de colunas não os podendo juntar aos dados de outra família de colunas. Uma outra limitação e, de acordo com (Wilke, 2012), é que é obrigatório armazenar as linhas em disco, pois as chaves das linhas são usadas para determinar os nós responsáveis pela replicação dos dados. Um outro ponto menos favorável é o facto de a *API* ser construída em *Thrift*, não suportando assim capacidades de *streaming*, o que implica que cada valor escrito, ou lido, seja passível de ser armazenado em memória, limitando o suporte de objetos grandes por parte do Cassandra sem uma *API* específica. Um outro ponto negativo e, de acordo com (Datastax, 2012), é o facto dos índices das colunas do Cassandra não suportarem *queries* que necessitem de acesso a dados ordenados. Por fim e, de acordo com (Datastax, 2012), o facto de não ser possível usar índices secundários nas subcolunas de uma superfamília de colunas, restringindo assim o uso de superfamílias de colunas quando o número de subcolunas constitui também uma das desvantagens deste *software*.

5.1.3 CouchDb

Os pontos fortes deste modelo são baseados numa conferência do CouchDB e, de acordo com (Borkar, 2012), são: um *schema* flexível que permite ao utilizador alterar a estrutura da base de dados sem restruturar os dados existentes; elasticidade dinâmica que possibilita a consistência dos dados enquanto estão em movimento; desempenho, pois os dados são armazenados num documento único, permitindo assim uma baixa latência de acesso; flexibilidade nas *queries*, pois a indexação permite que se façam *queries* não só aos documentos em si como ao seu conteúdo.
Uma das desvantagens mais salientes e, de acordo com (Lennon, 2009), é o facto de o *schema* ser tao livre de maneira a que possa incorrer numa replicação dos dados desnecessária. Uma outra desvantagem e, segundo (J. Chris Anderson, 2010), cada alteração a um documento gera um novo documento ao estilo dos *softwares* de controlo de versões como CVS e SVN, consumindo assim muito espaço em disco.

5.1.4 Neo4j

O aspeto mais importante subjacente a este *software*, de acordo com (Kollegger, 2012), é a rapidez com que as *queries* são executadas. Foi feito um teste num cenário de 1000 pessoas que tinham 50 amigos cada uma e foi feita uma *query* para descobrir os amigos dos amigos até uma profundidade de quatro vértices. No mySQL a correr um cenário similar, a *query*

demorou dois segundos a ser executada, já no Neo4j demorou cerca de dois milissegundos. Depois e para tornar o exemplo mais real, foram usados 1000000 vértices e o tempo de execução da *query* no Neo4j manteve-se nos dois milissegundos enquanto no mySQL uma *query* com tantos vértices provavelmente não executaria, o que prova que o Neo4j é ideal para armazenar e tratar grafos de uma forma muito rápida e eficiente. Outros aspetos positivos deste *software* dignos de referencia e, de acordo com (Eifrem, 2009), são: a possibilidade de representar informação semiestruturada, ou seja, não requere uma estrutura previamente definida dos dados. Como tal o esquema da base de dados é fácil de evoluir, é possível estruturar o esquema de um grafo na própria base de dados sem recorrer a tabelas e *joins*. O desempenho de representação é maior do que outro tipo de base de dados pois, ao usar uma base de dados relacional, o *overhead* das relações teria custos de desempenho elevados. De acordo com (Howard, 2012), o Neo4j tem ainda a possibilidade de executar transações XA , ou seja, transações distribuídas que envolvem mais do que uma base de dados. Suporta também transações ACID que trazem alguns conceitos uteis das bases de dados relacionais como por exemplo o *commit* e o *rollback* de transações, a atomicidade, o isolamento, etc.

Os pontos menos favoráveis deste *software* e, segundo (Chad Vicknair, 2010), são: o grau de maturidade deste produto: como ainda há pouca adesão, é possível que ainda existam alguns bugs latentes por descobrir. Como usa uma indexação por texto, fazer uma pesquisa por um campo que não seja do tipo texto vai implicar *casts* e aumentar os custos de processamento. Não tem segurança a nível dos dados, de acordo com o manual do Neo4j, implicando assim que se recorra a ficheiros de configuração que bloqueiam o acesso à base de dados com base nos endereços de *IP* ou *URL's*. Outras alternativas para ter alguma segurança, envolvem remover plug-ins da *classpath* da aplicação e implementar regras através de classes de Java que não são difíceis de fazer, no entanto são mais trabalhosas. Acresce o facto de não suportar *joins* e ser otimizado para *queries* que vão de registo para registo, de acordo com (Kollegger, 2012).

5.1.5 Redis

O Redis é um servidor de dicionários remotos, implementa o modelo de pares chave/valor e é uma base de dados em memória. As principais vantagens que apresenta, de acordo com (Russo, 2010), são: tem um desempenho bastante elevado pois no *benchmark* que o autor faz referência, o Redis processou 100003 pedidos de set em pouco menos de dois segundos e atendeu a 100000 pedidos de *get* em dois segundos e meio. As operações são atómicas, ou seja, uma operação não pode ser dividida, no entanto o Redis também suporta atomicidade em grupos de operações. Tem um modelo de tipos orientado aos cientistas baseado em listas e conjuntos.

Como pontos menos positivos, o autor menciona que quanto maior for o *dataset* mais memória RAM (*Random Acess Memory*) é consumida, daí que quanto maior for o volume de dados com que o Redis lida mais RAM vai consumir e, dependendo da quantidade que temos

disponível, pode-se tornar num problema. A persistência é também um ponto menos positivo porque as operações de escrita e leitura em disco inerentes à persistência são muitos pesadas e em alguns casos podem implicar que o servidor bloqueie até que a escrita ou leitura em disco esteja concluída. A ocorrência de *Memory Bloats*, que são na verdade discrepâncias entre a quantidade de bytes entregue ao Redis e a quantidade de bytes usados em memória por esses dados. Na documentação oficial do Redis[37] foi possível encontrar mais uma desvantagem: o facto de o Redis correr numa *thread* única, não tirando assim partido da arquitetura *multicore* dos processadores atuais. Contudo é possível ter várias instâncias do Redis a correr na mesma máquina e tratá-las como servidores distintos mesmo que essas instâncias sejam sempre vistas como uma *thread*.

5.1.6 MongoDB

O MongoDB é uma base de dados orientada a documento e é mantido pela 10gen[38]. O nome deriva da palavra inglesa "humongous" que significa em português enorme.

As vantagens deste modelo, de acordo com (Lehmann, 2012), são: a linguagem de *querying* é tão dinâmica como o SQL; tem extensões *(add-ons)* que permitem o armazenamento de objetos binários, tais como imagens e vídeos que vão além dos esperados pares chave/valor; tem indexação por geo-referenciação incluída, o que permite *queries* como: "qual a estação de comboios mais próxima?" sejam fáceis de tratar. De acordo com (Dilley, 2012), tem uma maior facilidade na criação e execução de *queries* muito complexas e, de acordo com (Cattell, 2010), o MongoDB não tem *lock,* ou seja, é uma base de dados em que uma modificação a um recurso não bloqueia o acesso a esse recurso. Suporta *sharding* automático e operações atómicas como por exemplo a função *findAndReplace* que junta num só método a procura de um elemento e a sua modificação.

Como principais pontos menos bons e, de acordo com (Cattell, 2010), os índices do Mongo são árvores binárias que implicam que os critérios de uma *query* estejam na mesma ordem da árvore binária dos índices. Não devem ser usados campos em demasia na criação de índices. Só devem ser usados os necessários pois se forem usados demais, a base de dados pode ocupar mais espaço em disco do que era suposto. A diversidade dos tipos e ou linguagens usadas internamente pelo MongoDB pode causar um problema grave. Ao usar BSON para armazenamento dos dados, JSON para representação dos mesmos e JavaScript para administração originou o *bug* da comparação de dois longos[39]. Este *bug*, que passados dois anos ainda não foi resolvido, dá-se ao comparar dois números longos iguais e o resultado dessa comparação é falso ao invés de verdade.

[37] http://redis.io/topics/faq
[38] http://www.10gen.com/
[39] https://jira.mongodb.org/browse/SERVER-1672

5.1.7 Hadoop/HBase

O Hadoop, segundo a página oficial (Apache Hadoop, 2008), é uma *framework* que permite distribuir o processamento de grandes *datasets* através de um *cluster* de máquinas vulgares. Esta *framework* é composta por vários outros componentes, dos quais se destacam: o HDFS, que é um sistema de ficheiros distribuído baseado no Google Filesystem e, o HBase que, segundo a página oficial do HBase[40], é a base de dados distribuída do Hadoop.

De seguida vão ser apresentadas as vantagens e desvantagens do HBase e do Hadhoop pois ambos fazem parte de um todo.

5.1.7.1 Hadoop

As vantagens desta *framework,* de acordo com (Stone, 2011), são: distribuição não só dos dados como também do seu processamento/análise; as tarefas são independentes, o que permite lidar com falhas parciais mais facilmente; tem uma escalabilidade linear desenhada especialmente para o uso em máquinas vulgares; tem um modelo de programação simples pois a aplicação cliente só descreve funções de *Map/Reduce*.

Os pontos menos positivos desta *framework,* também de acordo com (Stone, 2011), são: é um *software* ainda em desenvolvimento e alguns dos seus módulos estão em constante atualização, como por exemplo o HDFS que só ainda há relativamente pouco tempo é que recebeu funções para fazer o *append* a ficheiros; o modelo de programação é restritivo pois uma *query* ao *dataset* implica a sua subdivisão em Map/*Reduce*, ou seja, de certa forma, tudo cai eventualmente no *Map/Reduce*; os *joins* de *datasets* são muito pesados pois são demasiado lentos e o facto de não existirem índices ainda aumenta mais o problema; as operações de gestão do *cluster* (*debug*, distribuição de software, *logs*) são complicadas de fazer; ainda só permite ter um nó *master,* que requere muita atenção para não deixar de funcionar e, limita a escalabilidade global do sistema; a gestão do fluxo de trabalhos não é fácil, quando os dados intermédios do processo de *Map/Reduce* são para manter; a configuração dos nós em termos de limites de memória e papel (*mapper* ou *reducer*) não é trivial.

5.1.7.2 HBase

As vantagens deste módulo, segundo (Varley, 2012), são: um modelo de dados semiestruturado que lida muito facilmente com vários tipos de dados diferentes na mesma tabela. É fácil de escalar pois apenas basta adicionar mais um servidor. De acordo com (Dhruba Borthaku, 2011), o HBase suporta também a distribuição dos dados de uma forma automática, removendo assim a necessidade da escolha da forma como os dados iam ser distribuídos. Por sua vez, a distribuição da carga é simples porque mais uma vez basta só adicionar mais uma máquina que automaticamente vai receber os dados que lhe foram atribuídos pelo master.

[40] http://hbase.apache.org/

Os aspetos negativos deste módulo, de acordo com (Varley, 2012), residem no facto de não existirem *joins*, ou seja, não é permitido juntar os dados de uma família de colunas com outra e no facto de não ter suporte para índices, o que dificulta um pouco as pesquisas à base de dados.

5.1.8 Sumário de Características dos produtos comparados

De forma a concluir esta comparação e a jeito de simplificar a leitura, foi decidido incluir quatro tabelas que classificam, entre outros, os *softwares* referidos anteriormente em termos de: *API's* suportadas para efetuar *queries*, mecanismos de controlo de concorrência, métodos de distribuição e modos de replicação, segundo (Robin Hecht, 2011):

Tabela 2 – Tabela Comparativa das API's de query suportadas pelos softwares NoSQL adaptado de (Robin Hecht, 2011)

Bases de Dados		**Opções de *query***			
		API REST	**API Java**	**Linguagem Própria**	**Suporte a *Map/Reduce***
Chave/Valor	Voldemort	-	+	-	-
	Redis	-	+	-	-
	Membase	+	+	-	-
Documento	Riak	+	+	-	+
	MongoDB	+	+	-	+
	CouchDB	+	+	-	+
Família de Colunas	Cassandra	-	+	+	+
	HBase	+	+	+	+
	Hypertable	-	+	+	+
Grafo	Sesame	+	+	+	-
	BigData	+	+	+	-
	Neo4J	+	+	+	-
	GraphDB	+	+	+	-
	FlockDB	-	+	-	-

Como se pode observar na tabela, todos os referidos produtos têm uma *API* para Java. Isto demonstra a popularidade da linguagem Java que, segundo (Singh, 2012), é a primeira linguagem mais popular entre os programadores. Outra *API* que também é suportada pela maioria das bases de dados da tabela anterior, é a *API REST* que, como já foi discutido anteriormente, tem as suas vantagens. As bases de dados da tabela anterior suportam em geral *Map/Reduce* e linguagens próprias da base de dados como o Cassandra CQL.

Tabela 3 – Tabela Comparativa dos Mecanismos de Controlo de Concorrência adaptado de (Robin Hecht, 2011)

Base de Dados		**Mecanismos de Controlo de Concorrência**		
		Mecanismo de *Lock* Normal	**Mecanismo de *Lock* Optimista**	**Mecanismo MVCC**
Chave/Valor	Voldemort	-	+	-
	Redis	-	+	-
	Membase	-	+	-
Documento	Riak	-	-	+
	MongoDB	-	-	-
	CouchDB	-	-	+
Família de Colunas	Cassandra	-	-	-
	HBase	+	-	-
	Hypertable	-	-	+
Grafo	Sesame	+	-	-
	BigData	-	-	-
	Neo4J	+	-	-
	GraphDB	-	-	+
	FlockDB	-	-	-

O mecanismo de *lock* normal de uma base de dados serve para garantir que apenas um e um só processo pode aceder a um pedaço da informação. Segundo (Microsoft, 2005), o mecanismo de *lock* serve para sincronizar o acesso por parte de múltiplos utilizadores à mesma informação ao mesmo tempo, ou seja, quando um utilizador efetua uma transação a uma linha da tabela, esta por sua vez fica indisponível para outras transações até que a transação do utilizador termine e liberte a linha.

O mecanismo de *lock* Optimista, como já foi ligeiramente abordado anteriormente, é, de acordo com a documentação do JBOSS[41] e com (Oracle, 2001), um mecanismo sem *lock*, ou seja, tudo está acessível a todos os utilizadores ao mesmo tempo, no entanto quando uma transação é feita e está prestes a terminar, a data de modificação dos dados é comparada com a data do início da transação e se a data de modificação for mais recente que a data da transação, ocorre um erro e a transação não é efetuada (*rollback*).

O mecanismo de MVCC (*Multiversion Concurrency Control*) é também um mecanismo que não envolve *lock.* Na prática e, segundo (Actian, 2010), todas as transações têm um *snapshot* dos dados fazendo com que os dados lidos estejam consistentes algures no tempo. De acordo com (Robin Hecht, 2011), o MVCC, em vez de usar *locks,* usa um conjunto de versões não modificadas ordenadas cronologicamente, garantindo assim sempre a última versão dos dados e, quando existe concorrência nas escritas, fica a cargo do cliente resolver estes conflitos pois com cada processo que escreve, um novo valor fica com uma ligação à versão antiga.

[41] http://docs.jboss.org/jbossas/docs/Server_Configuration_Guide/4/html/TransactionJTA_Overview-Pessimistic_and_optimistic_locking.html

Posto isto, é possível observar que as várias bases de dados NoSQL têm diversos modelos de concorrência ficando depois ao critério de cada um escolher o mais adequado ao fim que se destina a base de dados.

Tabela 4 - Tabela Comparativa dos Métodos de distribuição adaptado de (Robin Hecht, 2011)

Base de Dados		**Metodo de Distribuição**	
		Baseado em Intervalo (*range*)	**Baseado em *Hashing* Consistente**
Chave/Valor	Voldemort	-	+
	Redis	-	+
	Membase	-	+
Documento	Riak	-	+
	MongoDB	+	-
	CouchDB	-	+
Família de Colunas	Cassandra	-	+
	HBase	+	-
	Hypertable	+	-
Grafo	Sesame	-	-
	BigData	-	+
	Neo4J	-	-
	GraphDB	-	-
	FlockDB	-	+

O método de intervalo (range), segundo (Robin Hecht, 2011), consiste em dividir os dados em intervalos que por sua vez são distribuídos pelos vários nós existentes. Este esquema obriga à existência de um nó central que tenha conhecimento de quais intervalos guardam os diversos nós. Para além de coordenar o acesso aos intervalos, este nó pode ainda atuar como distribuidor de carga facilitando o trabalho dos outros nós. Esta estratégia é otimizada para *queries* que contenham intervalos, como tal, este tipo de *queries* são efetuadas de uma forma muito eficiente. No entanto, a existência de um nó central implica que toda a base de dados esteja dependente dele para funcionar corretamente. Se este nó falhar, todo o sistema deixa de funcionar porém, é possível replica-lo de forma a não ter um único ponto de falha nem uma dependência tão elevada.

Na segunda estratégia, onde existe uma dependência muito grande, através de uma função de *hash* é possível distribuir os dados por regiões de *hash*; os clientes podem calcular através da função de *hash*. Com isto, tem-se uma maior disponibilidade e simplicidade da arquitetura, no entanto, as *queries* de intervalo tem um desempenho menor nesta arquitetura.

Pelo que se pode ver pela tabela, as bases de dados do tipo chave/valor e documento usam o método de distribuição por *hash* consistente, por outro lado, as bases de dados do tipo coluna empregam o método de distribuição de intervalo. Um caso para o qual ainda não existe uma solução distinta, é o caso das bases de dados do tipo grafo porque um grafo não é fácil de

distribuir. As *queries* que são executadas sobre ele envolvem atravessá-lo e como tal, distribuir o grafo pode prejudicar o desempenho das *queries* devido ao tráfego de rede gerado pela distribuição. Daí que algumas bases de dados do tipo grafo, como por exemplo o Sesame, o Neo4J e o GraphDB, não tenham suporte à distribuição, no entanto, como o FlockDB não permite travessias de mais do que uma relação, é possível distribuir o grafo por vários nós.

Tabela 5 - Tabela Comparativa dos Modos de Replicação adaptado de (Robin Hecht, 2011)

Bases de Dados		**Modo de Replicação**		
		Leitura de uma Replica	**Escrita para uma Replica**	**Consistência**
Chave/Valor	Voldemort	+	-	+/-
	Redis	+	-	-
	Membase	-	-	+
Documento	Riak	+	-	+/-
	MongoDB	+	-	+/-
	CouchDB	+	+	-
Família de Colunas	Cassandra	+	-	+/-
	HBase	-	-	+
	Hypertable	-	-	+
Grafo	Sesame	-	-	+
	BigData	+	-	+
	Neo4J	+	+	-
	GraphDB	-	-	+
	FlockDB	+	+	+/-

Desta tabela e de acordo (Robin Hecht, 2011), é possível depreender que se podem dividir as bases de dados apresentadas em três tipos: bases de dados de consistência eventual, bases de dados de consistência otimista e bases de dados totalmente consistentes. O primeiro tipo já foi referido anteriormente e significa que a base de dados vai ficar consistente ao longo do tempo. No segundo modelo, a consistência e a replicação são configuráveis, conferindo à base dados a capacidade de lidar com diversas situações. No terceiro modelo não há replicação pois segundo o teorema CAP, já discutido anteriormente, não é possível num sistema distribuído ter consistência, alta disponibilidade, distribuição e tolerância a falhas, como tal, abdicou-se da distribuição e tolerância a falhas para ter a consistência e alta disponibilidade.

Exemplos de bases de dados que encaixam no primeiro modelo são: Redis, CouchDB e Neo4J. Do segundo modelo são: Voldmort, Riak, MongoDB e Cassandra. Do terceiro modelo são todos os demais excetuando o BigData que é a única base de dados que suporta consistência e replicação nativamente.

5.2 Aplicação Exemplo

O objetivo desta seção é mostrar um exemplo que faça uma ponte entre o SQL e um exemplo de cada *software* NoSQL. Para esse efeito foi criado o seguinte diagrama de dados exemplo com base em (Haridas, 2009):

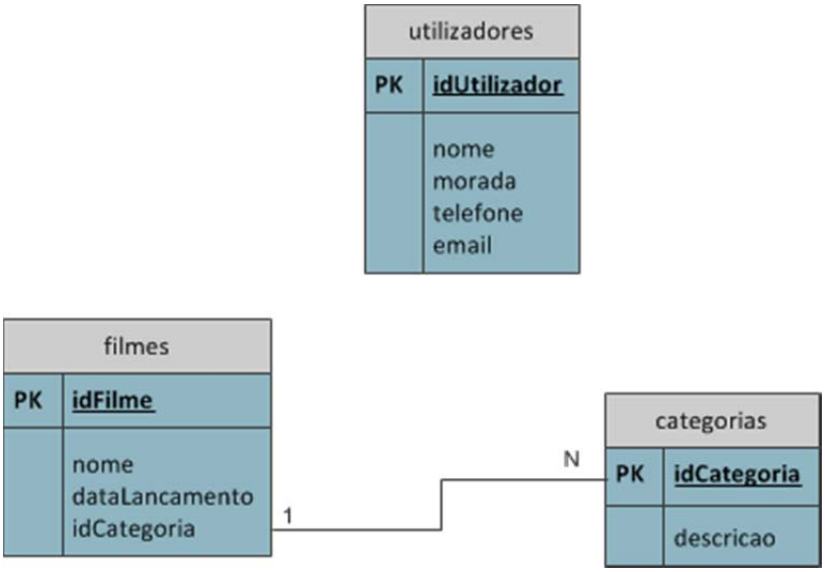

Figura 19 – Modelo de dados do exemplo do Windows Azure adaptado de (Haridas, 2009)

Em SQL o seguinte exemplo seria gerado através das seguintes instruções:

```
create table filmes(
 idFilme integer primary key,
 nome varchar(40),
dataLancamento DateTime,
idCategoria integer references categorias(idCategoria)
);
create table categorias(
 idCategoria integer primary key,
 descrição varchar(40)
);
create table utilizadores(
 idUtilizador integer primary key,
 morada varchar(40),
telefone integer,
email varchar(40)
);
```

Código 3 – Instruções SQL que permitiriam criar a base de dados do modelo anterior adaptado de (Haridas, 2009)

Programaticamente, em Windows Azure a base de dados seria criada com o código da página seguinte:

```
[DataServiceKey("PartitionKey", "RowKey")]
public class Movie
{
    /// <summary>
    /// A Categoria e a partition key
    /// </summary>
    public string PartitionKey { get; set; }
    /// <summary>
    /// O Titulo e a row key
    /// </summary>
    public string RowKey { get; set; }
    public DateTime Timestamp { get; set; }
    public int dataLancamento { get; set; }

}

[DataServiceKey("PartitionKey", "RowKey")]
public class Utilizador
{
    /// <summary>
    /// O numero de telefonr e partition key
    /// </summary>
    public int PartitionKey { get; set; }
    /// <summary>
    /// O email e a row key
    /// </summary>
    public string RowKey { get; set; }
    public string nome { get; set; }
    public string morada { get; set; }
}
```

Código 4 – Código que geraria a base de dados no Windows Azure Table Storage de acordo com (Haridas, 2009)

No caso do Cassandra e, de acordo com (Datastax, 2012) e com base em (Elis, 2011) e (Hewit, 2010), ter-se-iam as seguintes famílias de colunas:

Figura 20 – Exemplo das famílias de colunas resultantes no Cassandra do modelo de dados do exemplo adaptado de (Hewit, 2010)

Como se pode observar pela figura, as famílias de colunas, como anteriormente já foi dito, são parecidas com as tabelas, no entanto, uma das tabelas foi removida e inserida na tabela de filmes. Uma outra modificação foi criar um índex secundário na categoria do filme. De acordo com (Datastax, 2012) e (Hewit, 2010), o modelo de dados do Cassandra assenta na desnormalização, daí que a tabela categorias e a tabela filmes tenham formado uma só família de colunas. Segundo (Datastax, 2012), é possível criar índices secundários mas como já foi dito anteriormente, o Cassandra funciona melhor se estes se repetirem ao longo da família de colunas e como tal, a categoria é uma escolha aceitável como índice secundário.

Posto isto, vai ser apresentada a sintaxe que poderia criar o exemplo acima mencionado:

```
create keyspace Filmes
  with placement_strategy = 'SimpleStrategy'
  and strategy_options = {replication_factor : 1}
  and durable_writes = true;

use Filmes;
create column family utilizadores with comparator = UTF8Type;
create column family filmes with comparator = UTF8Type;

update column family utilizadores with
        column_metadata =
        [
        {column_name: nome, validation_class: UTF8Type},
        {column_name: morada, validation_class: UTF8Type},
        {column_name: telefone, validation_class: UTF8Type},
        {column_name: email, validation_class: UTF8Type}
        ];

assume utilizadores keys as utf8;

update column family filmes with
        column_metadata =
        [
        {column_name: nome, validation_class: UTF8Type},
        {column_name: dataLancamento, validation_class: UTF8Type},
        {column_name: categoria, validation_class: UTF8Type, index_type:
KEYS}
        ];

assume filmes keys as utf8;

set utilizadores['86']['nome'] = 'ricardoc';
set utilizadores['86']['morada'] = 'mancelos';
set utilizadores['86']['telefone'] = '939123363';
set utilizadores['86']['email'] = '1070482@isep.ipp.pt';

set filmes['filme_1']['nome']='Batman Begins';
set filmes['filme_1']['dataLancamento']='23-06-2005';
set filmes['filme_1']['categoria']='Accao';
```

Código 5 – Script do Cassandra que permitiria criar a base de dados exemplo adaptado de (Elis, 2011)

No caso do CouchDB, com base em (Borkar, 2012) e, usando o mesmo modelo de dados que foi usado no exemplo do Windows Azure, obter-se-iam os documentos seguintes:

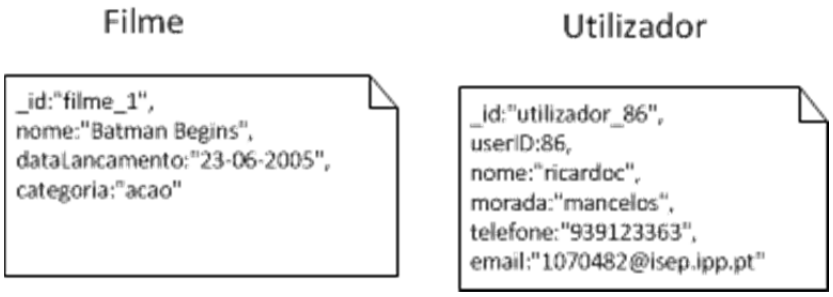

Figura 21 Exemplo dos documentos resultantes no CouchDB do modelo de dados exemplo baseado em (Borkar, 2012)

Uma das diferenças mais salientes neste tipo de base de dados é a total ausência de um *schema*, como tal não é necessário defini-lo inicialmente como no caso do SQL tradicional.

Uma outra diferença é que a base de dados pode ser vista como um grande saco de documentos, podendo armazenar documentos com estruturas radicalmente diferentes na mesma base de dados.

Os campos também mudaram um pouco, isto é, no caso dos filmes optou-se por criar um identificador com base num prefixo. É necessário o uso do prefixo filme de forma a possibilitar a distinção dos documentos que são filmes dos que não o são, sendo depois o resto do identificador um número sequencial como nas comuns bases de dados relacionais. Para além disso, as categorias que estavam numa tabela separada foram agora incorporadas no filme. Era possível ter documentos categoria que guardavam informação sobre a categoria e depois eram referenciados pelos documentos filmes, no entanto este esquema traz alguns inconvenientes como por exemplo se se quisesse descobrir informações sobre a categoria de um determinado filme, como não há *joins*, é necessário fazer duas *queries*, uma para retornar o filme em questão e outra para retornar a categoria do filme em questão. À medida que os objetos são distribuídos, mais complexa fica a camada da aplicação que tem de gerir toda a distribuição dos objetos. Quanto ao documento de utilizadores, foi mais uma vez usado um identificador com prefixo que atua como forma de distinguir os documentos que representam utilizadores dos demais documentos. Por fim, o documento tem um identificador do utilizador (userId) que é o identificador de um utilizador.

Posto isto, na página seguinte vai ser apresentada a sintaxe que permitiria criar a base de dados e os dois objetos acima mencionados através de pedidos *HTTP* com o uso de *cURL*:

```
curl –H “Content-Type: application/json” –X PUT
http://localhost:5984/movies
curl -H "Content-Type: application/json" -X PUT
http://localhost:5984/movies/filme_1 -d "{\"nome\":\"Batman
Begins\",\"dataLancamento\":\"23-06-2005\",\"categoria\":\"acao\"}"
curl -H "Content-Type: application/json" -X PUT
http://localhost:5984/movies/utilizador_86 -d
"{\"userId\":\"86\",\"nome\":\"ricardoc\",\"morada\":\"mancelos\",\"telefon
e\":\"939123363\",\"email:\":\"1070482@isep.ipp.pt\"}"
```

Código 6 – Pedidos cURL que permitiriam criar a base de dado exemplo no CouchDB baseado em (J. Chris Anderson, 2010)

No caso do Neo4j e, usando o mesmo modelo de dados que foi usado no exemplo do Windows Azure, obter-se-ia o seguinte grafo:

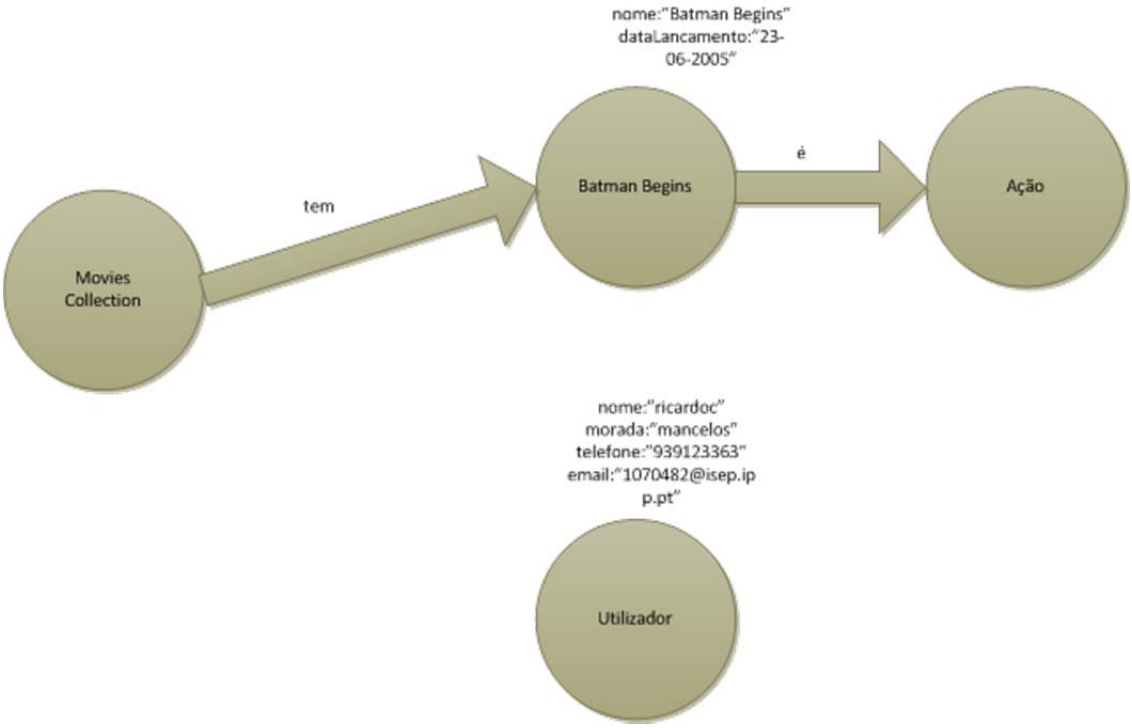

Figura 22 - Exemplo do grafo gerado no Neo4j do modelo de dados exemplo

O modelo de grafos obriga a algumas adaptações. De acordo com (Eifrem, 2012), as entidades em vez de surgirem sob a forma de tabelas, surgem sob a forma de nós do grafo e por sua vez, as relações surgem sob a forma de setas. Os dados propriamente ditos passam agora a residir em cada nó, sob a forma de pares chave valor. Uma das especificidades deste modelo é que ao deixar de ter tabelas, passa-se a ter apenas um conjunto de objetos que podem ou não estar ligados entre si. Outra alteração importante foi o abandono dos identificadores porque o Neo4j obriga a que a pesquisa sobre os dados seja feita de forma transversal, o que significa que para se obter um resultado é necessário percorrer todo o grafo, daí que os identificadores já não sejam necessários.

Posto isto, vai ser apresentada a sintaxe que permitiria criar o grafo, os quatro nós e respetivas propriedades acima mencionados através de pedidos *HTTP* com o uso de *cURL*:

```
curl -H Accept:application/json -H Content-Type:application/json -X POST -d
"{\"nome\": \"movies\"}" http://localhost:7474/db/data/node
curl -H Accept:application/json -H Content-Type:application/json -X POST -d
"{\"nome\": \"Batman Beggins\",\"dataLancamento\":\"29-06-2005\"}"
http://localhost:7474/db/data/node
curl -H Accept:application/json -H Content-Type:application/json -X POST -d
"{\"nome\": \"acao\"}" http://localhost:7474/db/data/node
curl -H Accept:application/json -H Content-Type:application/json -X POST -d
"{\"nome\":
\"ricardoc\",\"morada\":\"mancelos\",\"telefone\":\"939123363\",\"email\":\
"1070482@isep.ipp.pt\"}" http://localhost:7474/db/data/node
curl -H Accept:application/json -H Content-Type:application/json -X POST -d
"{\"to\": \"http://localhost:7474/db/data/node/6\", \"type\": \"TEM\"}"
http://localhost:7474/db/data/node/8/relationships
curl -H Accept:application/json -H Content-Type:application/json -X POST -d
"{\"to\": \"http://localhost:7474/db/data/node/7\", \"type\": \"E\"}"
http://localhost:7474/db/data/node/6/relationships
```

Código 7 – Pedidos *cURL* que permitiriam criar a base de dados exemplo no Neo4j de acordo com a documentação do Neo4j

O resultado final que pode ser visto usando a ferramenta de visualização do Neo4j é o seguinte:

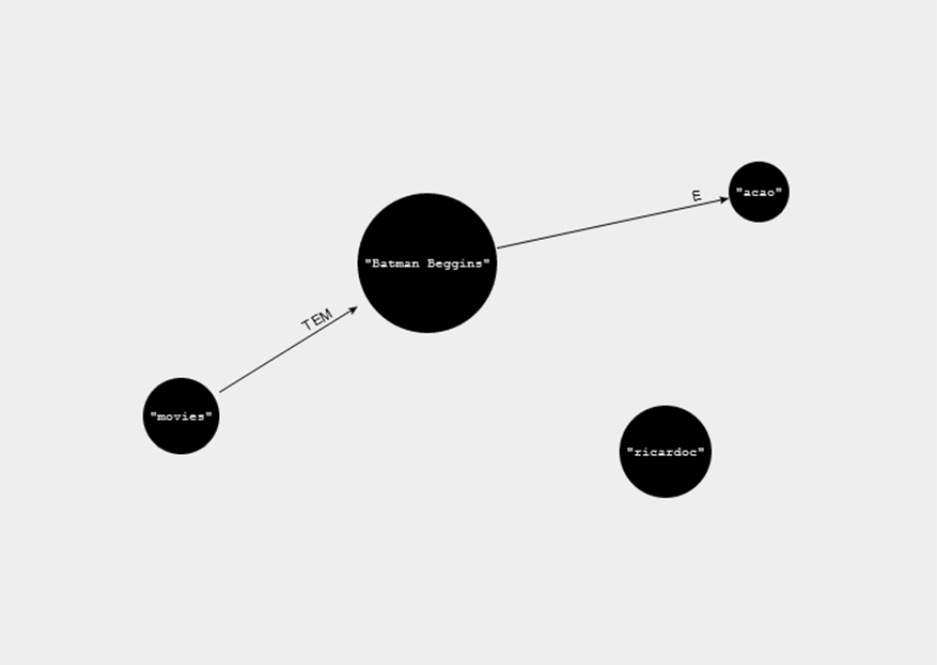

Figura 23 – Grafo final visto na interface web do Neo4j

5.3 Modelação de Relações

Uma relação numa base de dados relacional e, de acordo com (Microsoft, 2007), é um apontador de uma tabela para uma outra, por exemplo, tendo uma tabela de livros e uma tabela de autores, na tabela de livros colocar-se-ia um campo "id_autor" que por sua vez apontaria para um registo na tabela de autores correspondente a um autor. Estas relações servem em parte para não juntar todos os dados numa tabela, não repetir dados ao impor por meio de restrições (chaves primárias e secundárias) e para ligar os dados entre si.

No modelo NoSQL as relações são possíveis, no entanto o conceito de constrangimento não existe, existe sim, na mesma, os apontadores para outras tabelas que no modelo NoSQL tomam a forma de Documentos, Famílias de Colunas e Entidades. De acordo com (Holt, 2012), no CouchDB as relações podem ser feitas da seguinte maneira:

- Utilizar um campo como referência para o _id de outro documento.
- Utilizar campos que guardam *arrays* de dados em JSON.
- Utilizar um documento que tem os dois indentificadores de cada documento.
- Utilizar um documento que junte todos os campos dos dois documentos constituintes.

O problema das três primeiras formas e também de acordo com (Holt, 2012), é que com uma só *query* não nos é possível obter informação de documento e dos documentos por ele referenciados, ou seja, só se pode ter a informação de um documento e depois com o identificador do documento referenciado proceder à sua obtenção. Na quarta alternativa tem-se todos os dados aglutinados num documento, o que pode causar duplicação de informação mas, ao optar por não distribuir a informação por muitos objetos, o CouchDB não se vai tornar tão lento (Borkar, 2012).

No caso do AzureTable storage, as relações, segundo (Lerman, 2010), podem ser:

- Ter tudo numa tabela, configurando as *Partitionkeys* e *Rowkeys* da forma correta, ou seja, por exemplo, se tivermos a lidar com utilizadores e endereços, os endereços devem ter a mesma *PartitionKey* do utilizador mas *RowKeys* diferentes que sejam elegíveis o suficiente para distinguir entre um utilizador e um endereço.
- Utilizar um campo como referência para outra tabela.
- Utilizar campos que guardam dados em *strings* mas só podem ir até 64KB.

No primeiro caso, de acordo com (StackOverflow, 2011), temos uma solução que nos permite ter com uma só consulta os dados todos mas implica a duplicação de alguns dos dados. No segundo caso fica-se com uma maior complexidade na camada de aplicação que vai gerir todas estas relações e a possibilidade de não se obter todos os dados apenas com uma *query*. Finalmente no terceiro caso pode também levar a duplicação de dados e ter a limitação dos 64 KB.

No caso do Neo4j e, de acordo com a documentação do Neo4J, as relações, como fazem parte dos grafos, são triviais, sendo possível ter n nós com n relações. É possível ter um valor para essa relação e é possível também ter a noção da direção dessa relação.

No caso do Cassandra e, de acordo com (Poderi, 2010), é possível ter relações sob a forma de identificadores que apontam para outra linha (*row*) na mesma família de colunas como no exemplo seguinte:

```
People: {
  John Dow: {
    twitter: jdow,
    email: jdow@example.com,
    bio: bla bla bla,
    marriedTo: Mary Kay,
    ...
  },
  Mary Kay: {
    marriedTo: John Dow,
    ...
  },
  ...
}
```

Código 8 - Exemplo em *JSON* de uma relação com identificadores de acordo com (Poderi, 2010)

É também possível ter um campo numa família de colunas que aponta para uma coluna de uma outra família de colunas como por exemplo:

```
People: {
  John Dow: {
    twitter: jdow,
    email: jdow@example.com,
    bio: bla bla bla,
    marriedTo: Mary Kay,
    ...
  }
}
Children: {
  John Dow: {
    01/18/1976: John Dow Jr,
    05/27/1982: Kate Dow
  },
  ...
}
```

Código 9 - Exemplo em *JSON* de uma relação com outra família de colunas de acordo com (Poderi, 2010)

Para o caso de uma relação de muitos para muitos, tem-se duas famílias de colunas separadas contendo cada uma o identificador da primeira entidade e os dados da relação como na página seguinte:

```
Student:{
 Ricardo Cardoso:{
        fullName:Ricardo Manuel Fonseca Cardoso,
        Address:Rua conde de Avranches n56,
        phone:939123363,
        email:1070482@isep.ipp.pt
 }
 Jose Santos:{
        fullName:Jose Manuel dos Santos,
        Address:Rua conde de Avranches n56,
        phone:931234567,
        email:1070123@isep.ipp.pt
 }
}
Disciplines{
 INDES:{
        fullName:Interfaces and Design,
        department:Informatics Engeneering,
        branch:Arquitectures Systems and Networks,
        group:Industrial Informatics,
 }
 PSIDI:{
        fullName:Distributed Systems Programming
        department:Informatics Engeneering,
        branch:Arquitectures Systems and Networks,
        group:Computational Systems,
 }
 ESEGI:{
        fullName:Informatics Security Engeneering
        department:Informatics Engeneering,
        branch:Arquitectures Systems and Networks,
        group:Computational Systems,
 }
}
StudentDisciplines: {
  Ricardo Cardoso: {
    2010_1:PSIDI,
    2010_2: ESEGI,
 2012_1:INDES,
  }
   Jose Santos: {
    2008_1:PSIDI,
    2009_1: ESEGI,
 2010_1:INDES,
  }
}
DisciplineStudents{
 PSIDI:{
        1:Ricardo Cardoso,
        2:Jose Santos,
        ...
 }
 INDES:{
        1:Ricardo Cardoso,
        2:Jose Santos,
        ...
 }}
```

Código 10 – Exemplo de uma relação em *JSON* de muitos para muitos recorrendo a duas famílias de colunas separadas de acordo com (Poderi, 2010)

Finalmente, no caso de uma entidade "fraca", uma entidade que não pode ser identificada inequivocamente apenas pelos seus próprios atributos, há a possibilidade de usar uma superfamilia de Colunas:

```
Cars: {
  John Dow: {
    6CVW063: {
      model: chevrolet beat,
      color: blue,
      ...
    },
    4FYB066: {
      model: chevy equinox,
      color: white,
      ...
    },
  ...
}
```

Código 11 - Exemplo de uma entidade fraca em *JSON* representada por uma superfamilia de colunas de acordo com (Poderi, 2010)

5.4 Alterações para os Programadores que utilizam o Modelo Relacional

Hoje em dia o acesso aos dados guardados numa base de dados relacional é um processo simples do ponto de vista a que há várias formas de o fazer, no entanto essas formas de acesso podem ser resumidas em dois métodos de acesso:

- Uma *API* que pode ser fornecida pelos diversos fabricantes de *software,* tais como a Microsoft (ADO.Net,) ou Oracle (JDBC).
- Um ORM (*Object-Relational Maping*) que mapeia os objetos da aplicação para a base de dados, como por exemplo Hibernate e Entity Framework.

5.4.1 API's

O JDBC (*Java Database Conectivity*), segundo (Reese, 2000), é uma API de nível *SQL* que permite inserir comandos *SQL* em código Java abstraindo qual é a base de dados que vai ser usada, implicando assim que cada fabricante de base de dados forneça uma interface para o acesso à sua base de dados para posterior integração no JDBC. De acordo com (Maydene Fisher, 2003), JDBC é uma *API* que permite aceder a qualquer forma de dados tabular. É um conjunto de classes e interfaces Java com os quais se pode ler e escrever dados de qualquer base de dados com o uso da linguagem Java. Pode-se então concluir que JDBC é uma tecnologia de acesso a bases de dados relacionais através do uso de Java e *SQL*, no entanto, de forma a suportar qualquer base de dados e, de acordo com (Reese, 2000) e (Maydene Fisher, 2003), é necessário ter o driver específico do fabricante da base de dados, ou seja, o

JDBC inclui também diversos drivers para comunicar com os fabricantes de bases de dados mais comuns como mostra a imagem seguinte:

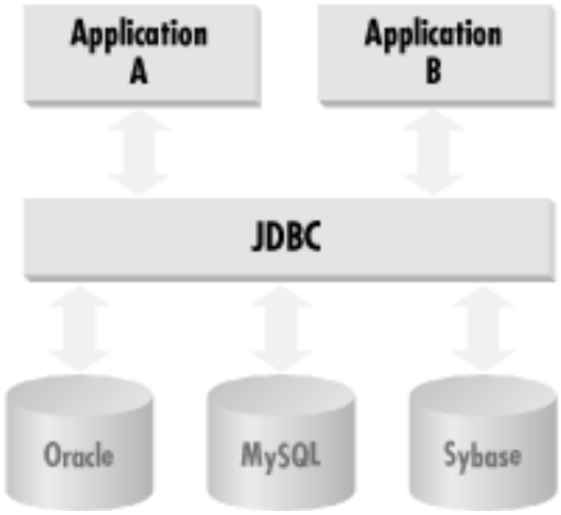

Figura 24 – Estrutura do JDBC retirado de (Reese, 2000)

O ADO.net, de acordo com (Patrick, 2010), é um conjunto de tecnologias que permite aceder e manipular dados estruturados (tabulares). Esses dados podem ser internos que são criados em memória e utilizados apenas na aplicação e externos que estão armazenados noutra localização afastada da aplicação, como por exemplo uma base de dados relacional. O ADO.net é constituído por:

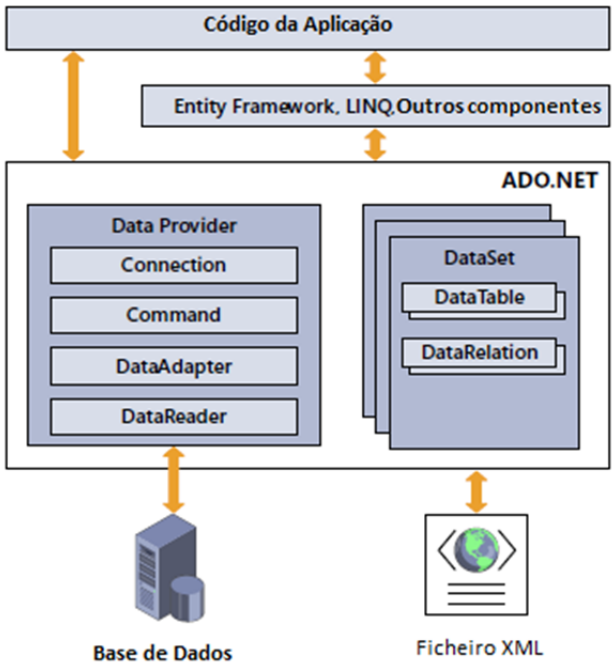

Figura 25 – Estrutura de Componentes ADO.net adaptado de acordo com (Patrick, 2010)

Com todos estes componentes é possível comunicar com qualquer base de dados suportada de forma automática, ou seja, com os controlos da *framework* .net preenchidos automaticamente ou manualmente, incluindo instruções SQL no código .net.

5.4.2 Object-Relational Maping

O Hibernate, de acordo com (Minter, 2010), é um motor de persistência que pode ser usado em qualquer aplicação Java. Permite mapear os objetos para a base de dados de uma forma simples (anotações ou XML) sem recorrer a grandes comandos SQL. Na prática, o *Hibernate* atua em conjunto com o JDBC de forma a otimizar a leitura e escrita de e para a base de dados como mostra o esquema da figura seguinte:

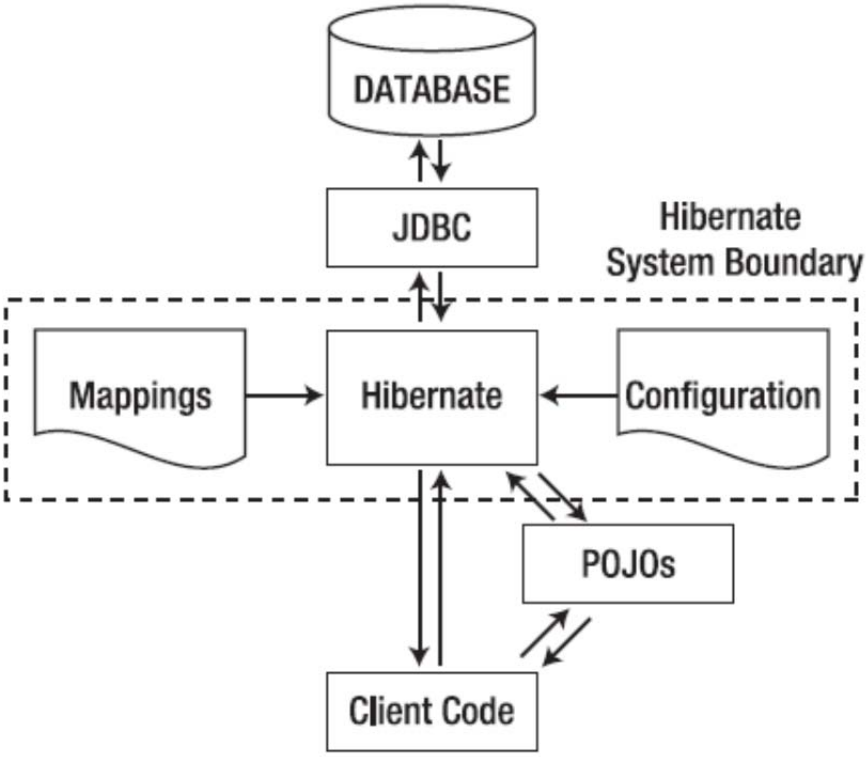

Figura 26 – Esquema do Hibernate retirado de (Minter, 2010)

Ao observar este esquema é possível notar que o Hibernate requere que o mapeamento dos objetos java para os objetos da base de dados seja feito, no entanto não é necessário escrever mais nenhum comando de *selec*t pois a *framework* faz o resto, com a vantagem da redução de código ser bastante aceitável quando comparado com o tradicional JDBC. De seguida vão ser apresentados dois exemplos de preenchimento de um objeto através de JDBC e Hibernate retirados de (Minter, 2010). O primeiro exemplo sem *Hibernate* e com o tradicional JDBC é ilustrado na página seguinte:

```
public static List getMessages(int messageId) throws MessageException {

Connection c = null;
 PreparedStatement p = null;

List list = new ArrayList();

try {
        Class.forName("org.postgresql.Driver");
        c = DriverManager.getConnection(
        "jdbc:hsqldb:testdb;shutdown=true",
        "hibernate",
        "hibernate");

        p = c.prepareStatement(

        "select message from motd");
        ResultSet rs = p.executeQuery();
        while(rs.next()) {
               String text = rs.getString(1);
               list.add(new Message(text));

        }
        return list;
 } catch (Exception e) {

        log.log(Level.SEVERE, "Could not acquire message", e);
        throw new MotdException(
        "Failed to retrieve message from the database.", e);

 } finally {
        if (p != null) {

               try {

                      p.close();

               } catch (SQLException e) {

                      log.log(Level.WARNING,
                      "Could not close ostensibly open statement.", e);

               }
        }

        if (c != null) {
               try {

                      c.close();

               } catch (SQLException e) {
                      log.log(Level.WARNING,
                      "Could not close ostensibly open connection.", e);
               }
        }
 }
}
```

Código 12 – Exemplo em Java do uso de ODBC retirado de (Minter, 2010)

O segundo exemplo já com *Hibernate* é o seguinte:

```
public static List getMessages(int messageId)
throws MessageException
{ SessionFactory sessions =
  new AnnotationConfiguration().configure().buildSessionFactory();
  Session session = sessions.openSession();
  Transaction tx = null;
  try {
        tx = session.beginTransaction();
        List list = session.createQuery("from Message").list();
        tx.commit();
        tx = null;
        return list;
  } catch ( HibernateException e ) {
        if ( tx != null ) tx.rollback();
        log.log(Level.SEVERE, "Could not acquire message", e);
        throw new MotdException(
        "Failed to retrieve message from the database.",e);
  } finally {
        session.close();
  }
}
```

Código 13 – Exemplo em *Hibernate* retirado de (Minter, 2010)

Como se pode observar, no primeiro exemplo é possível ver a instrução de SQL *select* para retornar a coluna *messages* da tabela motd, sendo necessário trinta linhas de código mas, ao comparar com a versão em *Hibernate* que tem metade das linhas de código, não necessita de nenhum comando de SQL puro, apenas foi precisa a coluna da qual era necessário retirar os dados. Um outro ponto a abordar é a questão dos mapeamentos, que podem ser feitos através de XML ou de anotações nas classes que definem os objetos da aplicação. De seguida vai ser apresentado um exemplo desse mapeamento adaptado de (Minter, 2010):

```
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
@Table(name=”Tbl_Messages”)
public class Message {
  private Message() {
  }
  public Message(String messageText) {
        this.messageText = messageText;}
  public String getMessageText() {
        return messageText;
  }
  public void setMessageText(String messageText) {
        this.messageText = messageText;}
  @Id
  public Integer getId() {
        return id;}
  public void setId(Integer id) {
        this.id = id;}
  private String messageText;
  private Integer id;}
```

Código 14 – Exemplo das anotações em *Hibernate* de acordo com (Minter, 2010)

Como se pode observar no exemplo anterior, as anotações surgem antecedidas de um @ que por sua vez indica ao compilador que o texto que se segue é uma anotação. A anotação @Entity em conjunto com a anotação @Table serve para indicar uma entidade e a tabela da base de dados onde essa entidade vai residir. É normal estar antes da declaração da classe e, por sua vez, a anotação @Id serve para indicar qual é o campo/atributo que funciona como chave primária. Existem diversas outras anotações tais como: @GeneratedValue, @TableGenerator, @IdClass, @Table,etc mas o Hibernate por si só não é o tema desta tese.

A Entity Framework, de acordo com (Klein, 2010), é uma estrutura que permite gerir dados relacionais como objetos. É também um modelo de programação concetual ao invés de um modelo de programação de armazenamento direto.

Ao fazer-se uso do tradicional ADO.Net de comandos SQL inseridos no código, os erros de SQL são de mais difícil perceção, acrescendo o facto de que quando um campo da base de dados mudar de nome é também necessário refletir essa mudança no código SQL que está inserido no código da aplicação. Isto implica que os programadores passem mais tempo a tratar das especificidades da base de dados ao invés de fazerem aquilo que é realmente importante, desenvolver a aplicação.

Numa tentativa de abstração da linguagem SQL, a Microsoft desenvolveu a Entity Framework, como um complemento ao ADO.Net, que faz o mapeamento dos objetos da aplicação para a base de dados com o uso de um EDM (Entity Data Model), um Modelo de Dados das Entidades, sem a necessidade dos tradicionais comandos SQL.

De seguida, vai ser apresentado um exemplo de um EDM:

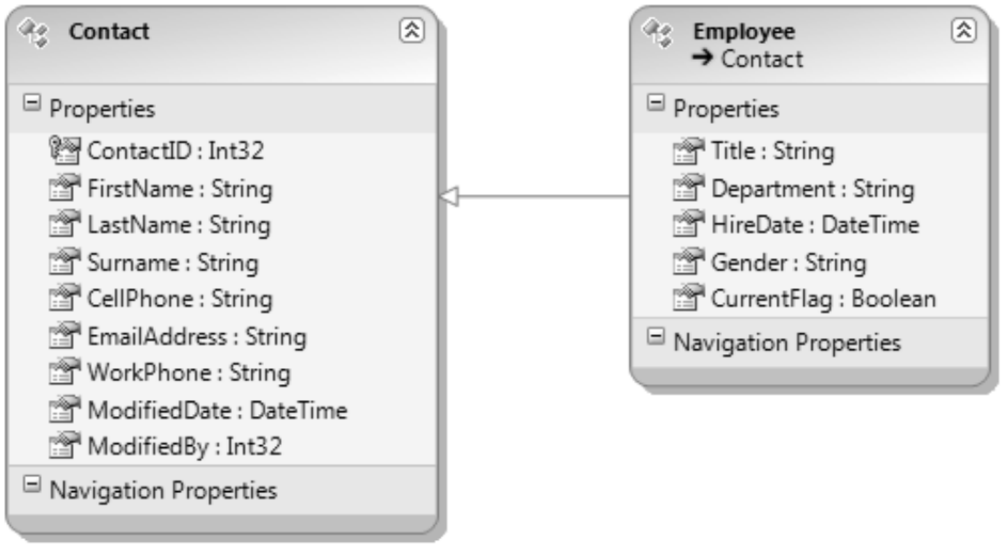

Figura 27 – Exemplo de um EDM retirado de (Klein, 2010)

Com este EDM, para obter o contacto dos empregados, basta só uma linha de código na linguagem C#:

```
From c in Contact.TypeOf<Employee> select c;
```

Código 15 – Instrução que devolveria o contacto dos empregados de acordo com (Klein, 2010)

Em SQL "puro" seriam necessárias pelo menos quatro linhas de código:

```
SELECT c.FirstName, c.LastName, e.Title, e.HireDate, aci.CellPhone,
aci.EmailAddress
FROM Contact c
INNER JOIN AdditionalContactInfo aci ON c.ContactID = aci.ContactID
INNER JOIN Employee e ON c.ContactID = e.ContactID
```

Código 16 - – Instruções SQL que devolveriam o contacto dos empregados de acordo com (Klein, 2010)

Segundo (Klein, 2010), a Entity Framework permite criar o EDM de raiz e depois gerar automaticamente a base de dados, os mapeamentos e classes correspondentes ou criar o EDM a partir de uma base de dados já existente ou ainda usar apenas o código para descrever o modelo de uma forma similar ao *Hibernate*.

5.4.3 No Caso Do NoSQL

Como já foi dito anteriormente, existem dois métodos para aceder a uma base de dados: por API ou por ORM. Todos os projetos NoSQL disponibilizam uma ou mais *API's* para acesso programático, são exemplos disso o Ektorp e Jrelax, que são clientes do CouchDB, Pycassa e Hector que por sua vez são clientes do Cassandra, py2neo e Neo4jPHP, clientes do Neo4j e por fim, Sider e Jedis, clientes do Redis. Mas a disponibilização de uma *API* para acesso programático seria de esperar pois uma das características de todas as bases de dados NoSQL consiste no facto de terem uma *API* simples como já foi referido num capítulo anterior. É também importante referir que cada linguagem tem formas especificas de definir *queries,* como por exemplo, o Cassandra com o CQL (*Cassandra Query Language*) ou o Redis com os seus comandos próprios. O CouchDB e o Neo4j por outro lado usam o formato JSON e REST para gerir os objetos da base de dados.

De acordo com (Hightower, 2011), já existe também um OGM (*Object Grid Mapper*), a *framework* Hibernate OGM que visa fornecer uma interface comum de acesso às bases de dados NoSQL mas, como ainda é relativamente recente, só suporta uma base de dados NoSQL, o Infinispan, que é uma base de dados NoSQL do tipo par chave/valor. O objetivo deste novo OGM foi tentar reutilizar os componentes do *Hibernate* que já existiam, como por exemplo o *Hibernate Core* e o *Hibernate Search* que foram modificadas para suportar o Infinispan. O JDBC também deixou de ser necessário porque como se está a usar uma base de dados NoSQL, é necessário fazer uso da API própria do Infinispan. De uma forma geral, as anotações e a forma de as fazer permanece a mesma, de forma a não modificar o uso do *Hibernate* em si. Na página seguinte, vai ser apresentado um modelo da arquitetura do *Hibernate OGM:*

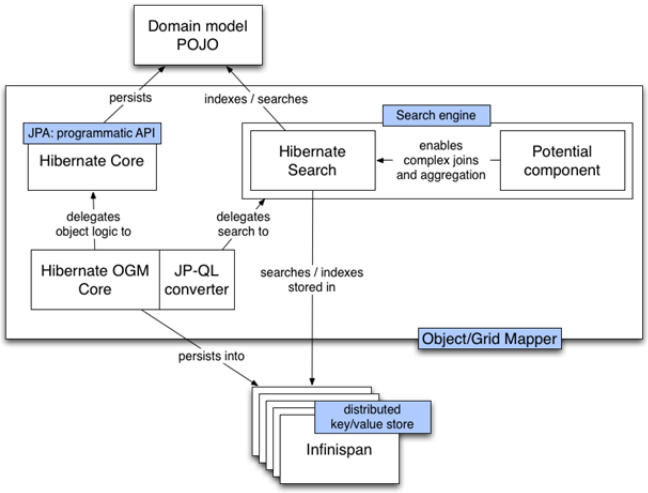

Figura 28 – Diagrama da arquitetura do *Hibernate* OGM retirado de (Hibernate, 2011)

Como é possível ver no diagrama, a estrutura não mudou muito, apenas foram acrescentados os componentes necessários, como o novo OGM Core que fará a persistência para o Infinispan.

De acordo com a fonte (Hightower, 2011), o *Hibernate OGM* tenciona suportar outros produtos NoSQL pois o Mongo, CouchDB e o Redis são contribuidores deste projeto e, como tal, o suporte para os seus produtos de base de dados NoSQL deve chegar em breve.

5.4.4 Diferenças

Como foi anteriormente referido, os métodos de acesso a uma base de dados NoSQL, quando comparados com os métodos de acesso a uma base de dados relacional, são iguais pois ambas as bases de dados têm uma *API* de acesso e um *ORM* ou *OGM*, se bem que este último ainda é relativamente recente. A principal diferença reside nas diferenças dos dois modelos, ou seja, o que mudou foi a forma como a base de dados é construída e organizada como por exemplo a normalização que no caso do NoSQL deixa de ser necessária (Penchikala, 2011).

A maneira de como as *queries* são construídas também mudou pois os *joins*, na maior parte dos produtos NoSQL, não são suportados e torna necessária a utilização de uma linguagem específica de cada produto ou uma *API REST* como já foi dito anteriormente. Em termos programáticos existem as mesmas formas de interagir com a base de dados, quer seja ela NoSQL ou relacional. Outra diferença reside também na sintaxe de cada *API* que é fornecida mas, mais uma vez, as diferenças seriam esperadas porque é uma tecnologia diferente e, como tal, tem as suas especificidades.

5.5 Padrões e boas práticas

5.5.1 CouchBase

De acordo com (CouchBase, 2012), os padrões a seguir subdividem-se em: **Modelo**, **Chaves Primárias**, **Edição e distribuição dos dados por documentos** e **Concorrência**.

No artigo de (CouchBase, 2012), é dito que a cada modelo deve corresponder um documento. Por exemplo, se um *blog* tem *posts* e comentários que são vistos como modelos diferentes na arquitetura MVC (*Model View Controller*) então, numa base de dados orientada a documentos, como é o caso do CouchBase, estes modelos devem ser documentos diferentes, ou seja, cada *post* deve corresponder a um documento e cada comentário deve corresponder a um outro documento. No caso do mapeamento dos objetos da aplicação, para o CouchBase, ao invés de se separar os objetos por tabelas basta apenas convertê-los em formato JSON e criar um id para eles, não sendo necessário mais nada para que o objeto fique armazenado no CouchBase.

No caso das Chaves Primárias, o artigo (CouchBase, 2012) afirma que só existe uma chave primária, o *document_id,* à margem do que acontece com as bases de dados relacionais. No entanto, é possível usar esta chave primária para criar referências em documentos relacionados como já foi referido anteriormente e, é possível optar por uma chave primária que é criada automaticamente ou especificada pelo utilizador. No primeiro caso, pode ocorrer que o *software* ordene os dados por data, possibilitando assim que dados com identificadores próximos sejam acedidos de uma forma mais eficiente; no segundo caso, o utilizador ao especificar a chave primária, pode recorrer ao uso de um prefixo de forma a agrupar os documentos, como por exemplo o prefixo já referido anteriormente: utilizador_numero de forma a reconhecer quais os documentos que representam utilizadores.

Quanto à Edição e Distribuição dos dados por documentos, o artigo (CouchBase, 2012) menciona que a edição dos dados de um documento está diretamente ligada à forma como os dados foram distribuídos por um documento, ou por vários, e se o número de documentos a ser atualizado não é muito elevado. No caso de se ter, por exemplo, uma foto e o seu título guardadas em documentos individuais por cada local em que essa foto aparece, para se proceder a atualização do título dessa foto, se o número de documentos a atualizar se situar entre 10 a 100 documentos e se a mudança não necessitar de ser síncrona, é possível ter uma tarefa a correr em *background* que atualiza o nome. No entanto, se vão ser atualizados um sem número de documentos e, ao invés de ter um documento com a foto e título por cada local em que a foto aparece, se optou por ter um documento foto, com a foto e o título, então todos os documentos em que a foto aparece de alguma, apontam para o documento foto. É possível apenas modificar o documento que contém a foto e as suas informações sem a necessidade de modificar outros documentos.

Sobre a concorrência, o artigo (CouchBase, 2012) atesta que para evitar atualizações concorrentes do mesmo documento é necessário tomar partido da utilização de múltiplos documentos. Num cenário que inclui artigos que são editados por um editor e quais podem

ser consultados e comentados por vários leitores, deve-se criar um documento que corresponda ao artigo em que o editor é o único a poder modificá-lo. Os comentários devem ser documentos à parte contendo uma referência ao artigo (document_id), o seu próprio identificador e comentário propriamente dito.

5.5.2 Cassandra

No Cassandra, um dos padrões mais importantes é talvez a forma de pensar orientada às *queries* (Hewit, 2010) e (Datastax, 2012). Num cenário relacional, primeiro desenha-se o modelo da base de dados, normaliza-se esse modelo e depois constroem-se *queries* sobre esse modelo.

No caso do Cassandra é diferente porque se faz ao contrário, isto é, de acordo com (Datastax, 2012), o primeiro passo para a modelação dos dados é pensar quais as *queries* que a aplicação vai executar e a partir daí devem ser estipuladas as famílias de colunas que vão servir essas *queries* tendo sempre em conta que deve ser feita uma família de colunas por cada *query* de forma a otimizar a base de dados para leituras. Na elaboração das famílias de colunas deve-se ter em conta que todas ou algumas colunas da família de colunas devem ser suficientes para responder a uma *query*. Finalmente, estas famílias de colunas residem num *KeySpace,* algo semelhante a uma base de dados relacional, e, segundo (Datastax, 2012), deve existir um *KeySpace* por cada aplicação que use o Cassandra.

Atualmente, o Cassandra já suporta nativamente índices primários e secundários. Os índices primários no Cassandra são as *rowkeys* de cada linha de uma família de colunas, o que possibilita fazer *queries* sobre estes índices que são mantidos por cada nó que gere uma partição dos dados. Quanto aos índices secundários, e, segundo (Datastax, 2012), estes já são suportados e devem ser usados apenas quando o valor do índice se repete várias vezes pelas diferentes linhas de uma família de colunas pois o Cassandra funciona melhor se assim for pois, por cada valor único do índice secundário, mais *overhead* vai existir na gestão e pesquisa do índice. Daí que os índices secundários devem ser usados em cenários em que se repitam várias vezes como é o caso de, numa família de colunas de utilizadores, quando se usa o distrito onde esses utilizadores residem. Ainda sobre os índices secundários, o autor de (Datastax, 2012) adverte que os índices secundários não devem ser usados em cenários de colunas que tenham muitos valores distintos, como no exemplo anterior, se em vez do distrito fosse usado o *email* de cada utilizador depois quando uma *query* fosse executada sobre valores deste índice, iria resultar em muitas procuras mesmo para resultados pequenos, tornando o sistema muito ineficiente. Como tal, nestes casos e a pensar numa maior eficiência, era possível criar uma família de colunas dinâmica de forma a atuar como índice secundário em vez de usar os que o Cassandra já fornece.

Quanto à seleção de chaves, o Cassandra não garante que as chaves naturais escolhidas pelo utilizador sejam únicas como tal, a aplicação que usa o *KeySpace* fica com esta tarefa

suplementar, garantir unicidade das chaves naturais. No entanto, o Cassandra fornece também as *surrogate keys* geradas automaticamente e com garantia de unicidade.

5.5.3 Microsoft Azure Table Storage

No caso do Micorsoft Azure Table Storage, uma das primeiras recomendações mencionadas em (Microsoft, 2010) é que não devem ser usadas datas como PartitionKeys pois, ao executar uma *query* que devolva dados ordenados por data, esses dados não vão estar ordenados corretamente. A solução para este cenário é definir uma data máxima, tal como referido no artigo (Microsoft, 2010), e posteriormente, subtrair a data dos dados a uma data máxima tendo assim os dados ordenados.

O Microsoft Azure Table Storage, de forma a manipular entidades (coleções de pares chave/valor), obriga a criar um objeto *DataServiceContext* para manter o registo dessas entidades. Para adicionar entidades a este objeto, existem vários métodos, no entanto, a forma como este objeto mantém o registo das entidades não é *thread safe,* como tal, é aconselhável um novo objeto por cada operação lógica (*update/delete/insert*).

Uma entidade só pode ser adicionada a um objeto *DataServiceContext* uma vez. Ao tentar adicionar a entidade a um objeto *DataServiceContext* uma segunda vez resultará numa Exceção.

Ao tentar obter uma entidade que ainda não existe, o Microsoft Azure Table Storage também vai devolver uma exceção mas, no entanto, é possível evitar essa exceção ao ativar uma propriedade que desativa as exceções associadas à não existência de entidades.

Ao usar a mesma instância do objeto Data *ServiceContext* para várias operações lógicas e, se ao salvar as alterações existir uma falha numa dessas operações, a operação falhada continua a ser registada pelo objeto *DataServiceContext*, ao voltar a guardar as alterações, essa operação vai ser de novo executada. Neste cenário deve ser usado um novo objeto *DataServiceContext* para ter um conjunto de operações limpas.

De forma a controlar se as várias entidades geridas pelo objeto *DataServiceContext* são ou não atualizadas, este objeto tem uma propriedade, *MergeOption*, que permite especificar o nível de atualização das entidades por ele geridas.

Como já foi dito anteriormente, a *PartitionKey* e a *RowKey* são fundamentais para manipulação dos dados, no entanto, o artigo (Microsoft, 2010) acerca desse assunto refere que a correta ou incorreta seleção da *PartitionKey* e a *RowKey* vai influenciar a Escalabilidade e a forma como as transações são feitas. Daí que a seleção deve ter em conta que, se é pretendido tirar partido das *Entity Group Transactions* (transações atómicas a entidades com a mesma *PartitionKey*), é necessário escolher um campo que permita agrupar os dados atendendo a um critério, como no exemplo do artigo (Microsoft, 2010), o género do filme. Este tipo de transações ao serem atómicas permitem obter dados que numa base de dados relacional estariam divididos entre várias tabelas, como tal, esta escolha deve ser feita para

tentar recuperar todos os dados necessários com uma só *query*. Ainda sobre este assunto, o artigo (Microsoft, 2010) refere as seguintes escolhas de *PartitionKeys*:

- *Single Partition Value*: o mesmo valor de *PartitionKey* para todas as entidades permite ter *Entity Group Transactions* que funcionam em *Batch*. Ao permitir ate 100 operações de cada vez, reduz os custos e permite operações atómicas por cada lote de entidades. Como desvantagens, não escala para além de um servidor, limitando assim esta opção para cenários em que o volume de informação não seja muito elevado.
- *New PartitionKey for Every Entity*: uma *Partition Key* para cada entidade. Como vantagens, escala muito bem porque é possível distribuir por um servidor cada entidade e depois ter um *loadBalancer* que gere o acesso a cada servidor que guarda a entidade. No entanto, as *Entity Group Transactions* ficam limitadas a apenas uma partição (um grupo de entidades num servidor) de um servidor e a uma *query* com intervalos (range *query*) o que neste cenário resulta numa *query* que percorra toda a tabela.

Para a escolha da *PartitionKey*, de forma a ter as *queries* o mais eficientes possível, a (Microsoft, 2010) recomenda que se determinem quais as propriedades mais importantes duma tabela pois estas vão ser usadas como filtros. De seguida, devem ser escolhidas as propriedades que sejam potenciais candidatas a serem *PartitionKeys*. Depois, deve ser identificada a *query* dominante (o tipo de *query* mais executada) e a partir dela devem ser escolhidas as propriedades que vão ser usadas como filtros, construindo assim o conjunto inicial de chaves (*keys*). Este conjunto deve ser ordenado por importância da *query* e, se até estas propriedades não identificarem inequivocamente uma entidade, deve ser incluída uma outra propriedade que o faça. Posto isto, se só restar uma propriedade, esta deve ser usada como *PartitionKey*. Se restarem duas, a primeira deve ser usada como PartitionKey e a segunda como *RowKey*. Se porventura restarem mais do que duas propriedades, então as propriedades restantes devem ser divididas em dois grupos e, grupo a grupo, concatenadas por um hífen. O primeiro grupo deve ser usado como *PartitionKey* e o segundo deve ser usado como *RowKey*, implicando assim que a aplicação cliente lide com as *keys* concatenadas. Ao usar esta metodologia, continua a ser possível agrupar as entidades, no entanto passa-se a usar um prefixo em vez de uma *PartitionKey* simples que facilita as *queries* de intervalo (range *queries*).

6 Conclusões

6.1 NoSQL como uma tendência

O modelo relacional é um dos modelos mais usados hoje em dia. Desde que foi inventado em 1970 e até aos dias de hoje, é a solução mais amplamente adotada em termos de persistência. Tem como vantagens ser um modelo robusto, fiável, ideal para dados tabulares e para sistemas cujo *schema* não varie muito. É um modelo bastante testado e com provas dadas e, fornece garantias de atomicidade, consistência, isolamento e durabilidade.

No entanto, quando é necessário escalar a base de dados devido à carga do sistema, quando se chega aos limites da escalabilidade vertical, ou seja, quando não se pode comprar máquinas melhores, ou quando se chega ao limite da escalabilidade horizontal porque essa modificação obriga a fazer tremendas mudanças na aplicação cliente, ou o número de conflitos nas várias instâncias da base é complicado de gerir e obriga à contratação de mais *DBA's*, ou para aumentar o desempenho é necessário recorrer a desnormalizarão, o modelo relacional parece não ser suficiente para solucionar o problema.

Com o aparecimento do movimento *Big Data* que afirma que cada vez mais a quantidade de dados que uma base de dados tem de armazenar é elevadíssima, com o aparecimento Teorema CAP que afirma que num sistema é impossível ter ao mesmo tempo consistência, alta disponibilidade e distribuição/tolerância a falhas, o modelo relacional devido às suas limitações não é de todo adequado para lidar com a realidade atual.

A solução abordada e estudada no presente documento é o NoSQL. Este, tenta resolver o problema rompendo com o modelo relacional introduzindo novos modelos e melhores técnicas de distribuição da base de dados. As bases de dados NoSQL apresentam como características principais um *schema free*, um esquema totalmente livre que permite modificações sem comprometer as aplicações clientes já desenvolvidas e uma *API* simples, ou seja, uma forma de acesso à base de dados programaticamente simples, fácil de usar e com grande suporte às principais linguagens e métodos de acesso. São bases de dados prontas a escalar horizontalmente e preparadas para trabalharem num *cluster* de dados, cuja

configuração não requere mais *DBA's*; algumas inclusive são capazes de automaticamente configurarem uma nova instância da base de dados quando uma nova máquina é introduzida (Cassandra) ou, que por exemplo, a replicação entre o *cluster* de máquinas é feita automaticamente (*MongoDB*). Muitas destas bases de dados NoSQL já suportam, ou têm de raiz, mecanismos de *Map/Reduce* e *Sharding*. Com os novos modelos chave/valor, documento, colunas e grafo, estas bases de dados estão aptas a lidarem com grandes volumes de dados pois o modelo em si, como não tem tantas restrições como modelo relacional, e, em alguns casos é um modelo propício à desnormalizarão dos dados, torna rápido o acesso aos dados. No entanto, há um preço a pagar. A consistência muda um pouco e passa-se a ter garantias de que ao longo do tempo o sistema irá ficar consistente (consistência eventual) ao invés de uma consistência forte e imediata como é o caso do modelo relacional mas, o teorema CAP assim o obriga.

Mas o presente documento não pretende afirmar que o modelo relacional está para cair em desuso ou que deve ser abandonado pois as milhares de empresas que utilizam este modelo não estão prontas para abdicarem dele assim tao rápido porque, como já foi dito anteriormente, o modelo relacional funciona bem até quando os dados começam a crescer muito. Daí que a tendência e, de acordo com (Fowler, 2011), é para haver uma persistência poliglota em que mediante o tipo e quantidade de dados que existem, tem-se uma solução adequada para os dados: modelo relacional, modelo NoSQL ou ambos.

O grande fundamento por detrás desta nova técnica de persistência é: deve ser usada a ferramenta certa para o trabalho certo ao invés de usar "sempre o mesmo prego e martelo". De acordo com (Fowler, 2011), quando se usa o modelo relacional em casos que não se justifica, como por exemplo, ter uma aplicação que não necessita de transações, que a base de dados não é partilhada e são feitas apenas pesquisas por *id's* e, o modelo relacional pode prejudicar o desenvolvimento de uma aplicação, como tal, de acordo com (Fowler, 2011), antes de usar o modelo relacional como o único método de persistência, os requisitos da aplicação devem ser analisados e a persistência numa base de dados NoSQL deve ser tida em consideração. Segundo esta tendência, dependendo da natureza dos dados que a aplicação necessita de armazenar, a forma como a persistência será implementada poderá ser ou não relacional ou um misto das duas, como por exemplo o Twitter que, de acordo com (Finley, 2011), usa o MySQL como base de dados primária e soluções NoSQL, como por exemplo o Cassandra e o Hadoop, para casos em que o modelo relacional não se adequa.

6.2 Trabalho concluído

Esta tese tem como objetivos compreender o **modelo de bases de dados NoSQL**, desde os diferentes tipos de bases de dados, que resultou na elaboração do capítulo 3 subseção 3.3, feita através de uma extensa pesquisa sobre os modelos NoSQL, as vantagens e desvantagens de NoSQL explicadas resultante da elaboração do capítulo 3 subsecção 3.2.5 onde foi feita uma compilação das vantagens e desvantagens de cada modelo através da pesquiza nos livros da especialidade, artigos e etc., o modelo de programação que pode ser visto nos exemplos práticos montados no capítulo 5 subseção 2 resultantes de uma pesquiza/visualização dos recursos de cada fabricante posteriormente compilada nesse capítulo, modo de funcionamento que pode ser visto ao longo do capítulo 2 e 3.

Um outro objetivo desta tese é a comparação entre o **modelo relacional e o modelo NoSQL** que é feita ao longo do capítulo 2, onde se enunciam os fundamentos teóricos e os limites do modelo relacional e do capítulo 3 onde se enunciam as grandes características do modelo relacional.

Esta tese também se propôs a analisar o **ponto de vista do programador** nomeadamente no que mudou através da pesquisa realizada sobre o tema, resultando no capítulo 5 subsecção 5.4, as novas técnicas fruto de uma pesquisa no *site* dos fabricantes dos produtos NoSQL e de artigos da especialidade e, as ferramentas/produtos existentes enunciados no capítulo 5 subsecção 5.1 que resulta de uma pesquisa/leitura da documentação e artigos de opinião existentes.

A presente tese também se designou a **comparar os diversos produtos NoSQL**. Da pesquisa feita sobre o tema nos artigos da especialidade resultou a subsecção 3.3.5 do capítulo 3.

Por fim, esta tese também teve como objetivo a **apresentação de um exemplo prático** implementado em cada um dos modelos num *software* NoSQL adequado resultando na eleição de um exemplo prático, a partir dos existentes, que também fizesse o paralelismo com o modelo relacional. Depois, este exemplo foi adaptado aos outros modelos resultando assim na subsecção 5.2 do capítulo 5.

6.3 Trabalho futuro

O objetivo do presente documento é dar uma visão teórica sobre o conceito NoSQL, daí que não seja mencionada nenhuma configuração dos produtos de NoSQL. Um trabalho futuro seria instalar e configurar um *cluster* de cada produto, montar uns casos de teste e anotar os resultados em termos de desempenho, configuração automática das réplicas, redistribuição dos dados e modelação de um problema com o uso dos diversos modelos NoSQL. Seria também interessante, ao mesmo tempo que se monta este *cluster*, manter uma base de dados relacional e testá-la com os mesmos casos de teste para experimentar na realidade os limites do modelo relacional.

Um outro ponto interessante e que esta tese não foca, com exemplos “reais”, ou seja, tentar resolver um problema em concreto com o uso de tecnologias NoSQL, documentando todo o processo de desenvolvimento da aplicação, desde a escolha da linguagem, da base de dados NoSQL, da respetiva *API*. Também seria interessante desenvolver o mesmo exemplo com a mesma base de dados mas, em vez de ser usada uma *API,* ser usado o novo *OGM* da *Hibernate* ao fim de testar as suas potencialidades. Um outro ponto de vista que poderia ser interessante explorar, são as configurações inerentes a cada produto, como tal seria estimulante mostrar as configurações de cada produto em termos de sistemas e redes.

Por último, deveria ser testada a tendência da persistência poliglota, isto é, configurar um sistema com os dois tipos de persistência: relacional e NoSQL e, analisar e estudar quais as configurações necessárias, problemas encontrados, a arquitetura da aplicação que utilizaria os dois meios de persistência e se efetivamente num sistema é possível coexistirem os dois tipos de persistência sem nenhum problema de desempenho.

7 Referências

Abadi, D., 2010. *DBMS Musings.* [Online]
Available at: http://dbmsmusings.blogspot.pt/2010/03/distinguishing-two-major-types-of_29.html
[Acedido em 16 Julho 2012].
Actian, 2010. *MVCC - Ingres Community Wiki.* [Online]
Available at:
http://community.actian.com/w/index.php/MVCC#MVCC_.28Multi_Version_Concurrency_Control.29
[Acedido em 19 Setembro 2012].
Allen, J., 2010. *InfoQ: A Case fo Graph Databases.* [Online]
Available at: http://www.infoq.com/news/2010/09/Graph-Databases
[Acedido em 11 Dezembro 2011].
Anad, S., 2011. *OSCON Data 2011 -- NoSQL @ Netflix, Part 2.* [Online]
Available at: http://www.slideshare.net/r39132/oscon-data-2011-nosql-netflix-part-2
[Acedido em 24 Setembro 2012].
Anand, S., 2010. Netflix's Transition to High-Availability Storage Systems. Outubro, p. 9.
Andrew J. Brust, B. B. I. I., 2011. NoSQL and the Windows Azure platform. *Investigation of an Unlikely Combination*, 25 Abril, p. 28.
Apache Hadoop, 2008. *Hadoop.* [Online]
Available at: http://hadoop.apache.org/
[Acedido em 19 Julho 2012].
Barber, C., 2010. *Slideshare Cassandra Say Goodbye to the Relation Database.* [Online]
Available at: http://www.slideshare.net/cb1kenobi/cassandra-say-goodbye-to-the-relational-database-562010
[Acedido em 30 Maio 2012].
Baron Shawrtz, P. Z. T. J. D. Z. L. &. D. J. B., 2008. *High Performance MySQL.* 2ª ed. Sebastopol: O'Reilly Media, Inc.,.
Bernadette Farias Lóscio, H. R. d. O. C. d. S. P., 2011. NoSQL no desenvolvimento de aplicações Web. p. 17.
Borkar, D., 2012. *Navigating the Transition from Relational to NoSQL Technology.* [Online]
Available at: http://www.slideshare.net/Couchbase/navigating-the-transition-from-relational-to-nosql-technology?mkt_tok=3RkMMJWWfF9wsRokv6%2FKZKXonjHpfsX57OosWqawlMI%2F0ER3fOvrPUfGjI4ASstqI%2FqLAzICFpZo2FFWFfKBdJND%2B%2FlY
[Acedido em 23 Maio 2012].
Boyce, B., 2010. *Neowin.* [Online]
Available at: http://www.neowin.net/news/storing-tweets-requires-four-petabytes-of-data-a-year
[Acedido em 10 Outubro 2011].

Brandão, M., 2011. *Redes Sociais Factos e Estatisticas.* [Online]
Available at: http://www.maiswebmarketing.com/redes-sociais-factos-e-estatisticas-de-utilizacao/#
[Acedido em 07 Novembro 2011].
Brewer, E. A., 2000. *Towards Robust Distributed Systems.* [Online]
Available at: http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
[Acedido em 25 Fevereiro 2012].
Browne, J., 2009. *Brewer's CAP Theorem.* [Online]
Available at: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem
[Acedido em 29 Fevereiro 2012].
Cattell, R., 2010. Scalable SQL and NoSQL Data Stores. Dezembro, p. 16.
Chad Vicknair, M. M. Z. N. C. W., 2010. *A Comparison of a Graph Database and a Relational Database.* New York, ACM, p. 6.
Charles Bell, M. K. &. L. T., 2010. *MySQL High Availability.* 1ª ed. Sebastopol: O'Reilly Media, Inc..
Chris Eaton, D. D. D. L. Z., 2012. *Understanding Big Data:Analytics for Entreprise Class Hadoop and Streaming Data.* 1ª ed. United States of Ameria: McGrawHill.
Codd, E. F., 1970. A Relational Model of Data for Large Data Banks. Junho, p. 11.
CouchBase, 2012. RDBMS to NoSQL: Managing the Transition. 12 Fevereiro, p. 9.
CouchDB, 2012. *CouchDB In The Wild.* [Online]
Available at: http://wiki.apache.org/couchdb/CouchDB_in_the_wild
[Acedido em 23 Agosto 2012].
Datastax, 2012. *Datastax.* [Online]
Available at: http://www.datastax.com/docs/1.1/index
[Acedido em 28 Agosto 2012].
Dhruba Borthaku, K. M. R. R. S. S. S. M. S. G. K. M. M., 2011. Apache Hadoop Goes Realtime at Facebook. Junho, p. 10.
Dilley, B. C., 2012. *InfoQ.* [Online]
Available at: http://www.infoq.com/articles/mongodb-java-orm-bcd
[Acedido em 18 Julho 2012].
Eifrem, E., 2009. *InfoQ: Neo4j - The Benefits of Graph Databases.* [Online]
Available at: http://www.infoq.com/presentations/emil-eifrem-neo4j
[Acedido em 04 Junho 2012].
Eifrem, E., 2012. *Intro to Graph Databases.* [Online]
Available at: http://video.neo4j.org/rUD2/intro-to-graph-databases-126/
[Acedido em 20 Agosto 2012].
Elis, J., 2011. *Cassandra In Action.* [Online]
Available at: http://www.slideshare.net/mubarakss/cassandra-tutorial
[Acedido em 23 Julho 2012].
Fay Chang, J. D. S. G. W. C. H. D. A. W., 2006. Bigtable: A Distributed Storage System for Structured Data. Novembro, p. 14.
Ferreira, E., 2010. *Introdução ao NoSQL Parte 1.* [Online]
Available at: http://escalabilidade.com/2010/03/08/introducao-ao-nosql-parte-i/
[Acedido em 07 Novembro 2011].
Ferreira, E., 2010. *Introdução ao NoSQL parte II.* [Online]
Available at: http://escalabilidade.com/2010/04/06/introducao-ao-nosql-parte-ii/
[Acedido em 16 Novembro 2011].
Fielding, R. T., 2000. *Architectural Styles and.* [Online]
Available at: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
[Acedido em 09 Julho 2012].
Finley, K., 2011. *How Twitter Uses NoSQL.* [Online]
Available at: http://www.readwriteweb.com/cloud/2011/01/how-twitter-uses-nosql.php
[Acedido em 02 Outubro 2012].
Fowler, M., 2011. *PolyglotPersistence.* [Online]
Available at: http://martinfowler.com/bliki/PolyglotPersistence.html#footnote-example
[Acedido em 02 Outubro 2012].

Ghemawat, J. D. &. S., 2004. MapReduce: Simplified Data Processing on Large Clusters. 1 Dezembro, p. 13.
Gupta, D., 2011. *Thrift vs. Protocol Buffers.* [Online]
Available at: http://floatingsun.net/articles/thrift-vs-protocol-buffers/
[Acedido em 09 Julho 2012].
Haridas, J., 2009. *Windows Azure Tables and Queues Deep Dive.* Los Angeles, Microsoft.
Haugen, K., 2010. *A Brief Story of NoSQL.* [Online]
Available at: http://blog.knuthaugen.no/2010/03/a-brief-history-of-nosql.html
[Acedido em 12 Novembro 2011].
Henricsson, R., 2011. Document Oriented NoSQL Databases. 13 Junho, p. 44.
Hewit, E., 2010. *Cassandra: The Definitve Guide.* Gravenstein Highway North, Sebastopol: O'Reilly Media, Inc..
Hibernate, 2011. *Hibernate OGM Reference Guide.* [Online]
Available at: http://docs.jboss.org/hibernate/ogm/3.0/reference/en-US/html_single/
[Acedido em 18 Agosto 2012].
Hightower, R., 2011. Hibernate Object Mapping for NoSQL Datastores. *Hibernate Object Mapping for NoSQL Datastores*, 12 Julho, p. 3.
Holt, B., 2012. *Presentation: Entity Relationships in a Document Database.* [Online]
Available at: http://architects.dzone.com/articles/presentation-entity
[Acedido em 08 Agosto 2012].
Howard, P., 2012. *Bloor.* [Online]
Available at: http://www.bloorresearch.com/blog/IM-Blog/2012/5/neo4j.html
[Acedido em 04 Junho 2012].
J. Chris Anderson, J. L. &. N. S., 2010. *CouchDB The Definitive Guide.* 1ª ed. San Francisco: O'Reilly Media, Inc..
Jacobs, A., 2009. The Pathologies of Big Data. 1 Julho.
James Manyika, M. C. B. B. J. B. R. D. C. R. A. H. B., 2011. *Big data: The next frontier for innovation, competition, and productivity.* [Online]
Available at:
http://www.mckinsey.com/Insights/MGI/Research/Technology_and_Innovation/Big_data_The_next_frontier_for_innovation
[Acedido em 07 Março 2012].
Klein, S., 2010. *Pro Entity Framework 4.0.* 1ª ed. New york: Apress.
Kollegger, A., 2012. *0906 - Intro to Graph Databases.* [Online]
Available at: http://watch.neo4j.org/video/49039621
[Acedido em 06 Setembro 2012].
Laurent, S., 2012. *MongoDB Case Study at NoSQL Now 2012.* [Online]
Available at: http://www.slideshare.net/organicveggie/mongodb-case-study-at-nosql-now-2012
[Acedido em 25 Setembro 2012].
Lehmann, K., 2012. *NoSQL databases – an overview.* [Online]
Available at: http://blog.mindoo.com/web/blog.nsf/dx/EC12_Mindoo_T4S6-NoSQL_en.pdf/$file/EC12_Mindoo_T4S6-NoSQL_en.pdf
[Acedido em 20 Agosto 2012].
Lennon, J., 2009. *Begininng CouchDB.* 1st Edition ed. New York: Apress.
Lerman, J., 2010. Windows Azure Table Storage – Not Your Father's Database. *Windows Azure Table Storage – Not Your Father's Database*, Julho, p. 4.
Lynch, S. G. &. N., 2002. *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.* [Online]
Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.20.1495&rep=rep1&type=pdf
[Acedido em 28 Fevereiro 2012].
Mark Slee, A. A. a. M. K., 2007. Thrift: Scalable Cross-Language Services Implementation. 01 Abril, p. 8.
Maydene Fisher, J. E. J. B., 2003. *JDBC API Tutorial and Reference.* 3ª ed. New Jersey: Prentice Hall.
Merriman, D., 2009. *Sharding Introduction - MongoDB.* [Online]
Available at: http://www.mongodb.org/display/DOCS/Sharding+Introduction#ShardingIntroduction-

RoutingProcesses%28mongos%29
[Acedido em 25 Setembro 2012].
Microsoft, 2005. *Locking in the Database Engine.* [Online]
Available at: http://msdn.microsoft.com/en-us/library/ms190615(v=sql.90).aspx
[Acedido em 19 Setembro 2012].
Microsoft, 2007. *Definir relações entre tabelas numa base de dados do Microsoft Access.* [Online]
Available at: http://support.microsoft.com/kb/304466/pt
[Acedido em 08 Agosto 2012].
Microsoft, 2010. *How to get most out of Windows Azure Tables.* [Online]
Available at: http://blogs.msdn.com/b/windowsazurestorage/archive/2010/11/06/how-to-get-most-out-of-windows-azure-tables.aspx
[Acedido em 29 Agosto 2012].
Miniwatts Marketing Group, 2011. *Internet World Stats.* [Online]
Available at: http://www.internetworldstats.com/emarketing.htm
[Acedido em 07 11 2011].
Minter, J. L. a. D., 2010. *Beginning Hibernate.* 2ª ed. New York: Apress.
Morgenthal, J., 2012. *Evolution in Data Integration From EII to Big Data.* [Online]
Available at: http://www.infoq.com/articles/DataIntegrationFromEIItoBigData
[Acedido em 07 Março 2012].
OakLeaf Systems, 2010. *OakLeaf Systems.* [Online]
Available at: http://oakleafblog.blogspot.pt/2010/11/what-happened-to-secondary-indexes-for.html
[Acedido em 13 Julho 2012].
Oracle, 2001. *Optimistic Locking with Concurrency in Oracle.* [Online]
Available at: http://orafaq.com/papers/locking.pdf
[Acedido em 19 Setembro 2012].
Oracle, 2012. Oracle: Big Data for the Entreprise. Janeiro, p. 16.
Palmer, B., 2010. Why not try an API? Small software innovations can be powerful branding tools.. *Brandweek,* 1 February, Volume 51, p. 12.
Patrick, T., 2010. *Microsoft ADO.NET 4 Step by Step.* 1ª ed. Sebastopol: O'Reilly Media, Inc..
Penchikala, S., 2011. *James Phillips on Moving from Relational to NoSQL Databases.* [Online]
Available at: http://www.infoq.com/news/2011/12/relational-nosql-databases
[Acedido em 18 Agosto 2012].
Pingdom, 2011. *Royal Pingdom - Internet 2010 in numbers.* [Online]
Available at: http://royal.pingdom.com/2011/01/12/internet-2010-in-numbers/
[Acedido em 10 Outubro 2011].
Poderi, R., 2010. *Getting Started With Cassandra.* [Online]
Available at: http://blogs.nologin.es/rickyepoderi/index.php?/archives/9-Getting-Started-With-Cassandra.html
[Acedido em 19 Agosto 2012].
Prunicki, A., 2008. *Object Computing Inq, Java News Brief.* [Online]
Available at: http://jnb.ociweb.com/jnb/jnbJun2009.html
[Acedido em 09 Julho 2012].
Reese, G., 2000. *Database Programming with JDBC and Java.* 2ª ed. Sebastopol: O'Reilly & Associates, Inc..
Robin Hecht, S. J., 2011. NoSQL Evaluation A Use Case Oriented Survey. 12 Dezembro, p. 6.
Roe, C., 2012. *The Growth of Unstructured Data: What To Do with All Those Zettabytes?.* [Online]
Available at: http://www.dataversity.net/the-growth-of-unstructured-data-what-are-we-going-to-do-with-all-those-zettabytes/
[Acedido em 10 Abril 2012].
Rosoff, J., 2010. *Realtime Analytics with MongoDB.* [Online]
Available at: http://www.slideshare.net/jrosoff/scaling-rails-yottaa?from=ss_embed
[Acedido em 05 Outubro 2012].
Russo, M. J., 2010. *Redis, from the Ground Up.* [Online]
Available at: http://blog.mjrusso.com/2010/10/17/redis-from-the-ground-up.html#heading_toc_j_21
[Acedido em 18 Julho 2012].

Rybiński, H., 1987. On First-Order-Logic Databases. 3 Setembro, p. 25.
Schrage, D. B. B. a. J. F., 2002. Denormalization Guidelines for Base and Transaction Tables. Dezembro, p. 5.
Seeger, M., 2009. Key Value Stores: A practical overview. 21 Setembro, p. 21.
Singh, A., 2012. *The Top 10 Programming Languages.* [Online]
Available at: http://www.freerepublic.com/focus/f-chat/2880277/posts
[Acedido em 18 Setembro 2012].
Sousa, P., 2010. *Paulo Sousa's Blog.* [Online]
Available at: http://unrealps.wordpress.com/2010/12/28/o-teorema-cap/
[Acedido em 19 Novembro 2011].
StackOverflow, 2011. *How do I properly store data relationships with Microsoft Azure Table Storage?.* [Online]
Available at: http://stackoverflow.com/questions/1114951/how-do-i-properly-store-data-relationships-with-microsoft-azure-table-storage
[Acedido em 14 Agosto 2012].
Stainer, D., 2010. *NoSQL Databases.* [Online]
Available at: http://www.nosqldatabases.com/main/2010/7/9/nosql-databases-an-overview.html
[Acedido em 23 Julho 2012].
Staveley, A., 2012. *More on the Pros/Cons of SQL and NoSQL.* [Online]
Available at: http://architects.dzone.com/articles/more-proscons-sql-and-nosql
[Acedido em 01 Julho 2012].
Stone, Z., 2011. *CS 264 Massively Parallel Computing.* [Online]
Available at: http://www.cs264.org/lectures/files/cs_264_hadoop_lecture_2011.pdf
[Acedido em 19 Julho 2012].
Tiwari, S., 2011. *Professional NoSQL.* Estados Unidos da America: Wrox Programmer to Programmer.
Tomasz Imielinski and Witold Lipski, J., 1982. A systematic approach to relational database theory. p. 7.
Varley, I., 2012. *HBase Schema Design.* [Online]
Available at: http://www.slideshare.net/cloudera/5-h-base-schemahbasecon2012
[Acedido em 19 Julho 2012].
Warden, P., 2011. *Big Data Glossary.* 1ª ed. Sebastopol: O'Reilly Media, Inc..
White, T., 2011. *Hadhoop The Definitve Guide.* 2ª ed. Sebastopol: O'Reilly Media, Inc..
Wilke, A., 2012. *Cassandra Limitations.* [Online]
Available at: http://wiki.apache.org/cassandra/CassandraLimitations
[Acedido em 30 Maio 2012].
Williams, A., 2010. *ReadWriteWeb:For Partners: The 12 Benefits and Risks of Windows Azure.* [Online]
Available at: http://www.readwriteweb.com/cloud/2010/07/ray-wang-of-the-altimeter.php
[Acedido em 16 Julho 2012].