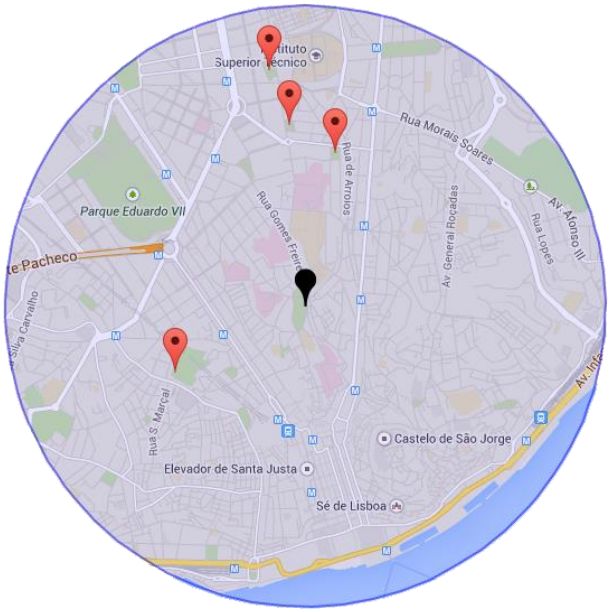
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia Electrónica e Telecomunicações e de Computadores

ISEL



SGBD de alta escalabilidade com suporte a dados georreferenciados

Ricardo Maranho
(Licenciado em Engenharia Informática e de Computadores)

Trabalho Final de Mestrado para obtenção do grau de Mestre em Engenharia Informática e de Computadores

Orientadores:
Doutor Walter Jorge Mendes Vieira
Licenciado António Carvalho (ACLSI)

Júri:
Presidente: Doutor Manuel Martins Barata
Vogais:
Mestre Nuno Miguel Soares Datia
Doutor Walter Jorge Mendes Vieira

Setembro de 2014

[DECLARAÇÕES]

Declaro que este trabalho de projecto é o resultado da minha investigação pessoal e independente. O seu conteúdo é original e todas as fontes consultadas estão devidamente mencionadas no texto, nas notas e na bibliografia.

O candidato,

_____

(Ricardo Soares Maranho)

Lisboa, ... de ..... de .....

O orientador,

_____

(Walter Vieira)

O orientador,

_____

(António Borges Carvalho)

Lisboa, ... de ..... de .....

Resumo

O desenvolvimento exponencial da tecnologia proporcionou a disponibilidade da mesma a toda e qualquer pessoa que tenha a possibilidade de a adquirir. A *internet* será provavelmente o melhor exemplo disso pois actualmente todas as pessoas que têm à sua disposição um *smartphone* têm acesso à internet. Com esta disponibilidade, que se tornou mundial, emergiu o conceito de rede social. Uma rede social é essencialmente uma comunidade de pessoas, ligadas entre si ou não, com um interesse comum e que utilizam uma determinada tecnologia para comunicar entre si com o objectivo de partilhar informação e/ou recursos.

Hoje em dia existem diversos tipos de redes sociais, sejam elas orientadas à partilha de informação empresarial, por exemplo *Linkedin*, ou orientadas à partilha de informação pessoal e comercial, por exemplo *Facebook*. No entanto, existem também redes sociais orientadas à partilha de informação georreferenciada, ou seja, informação associada a uma determinada coordenada geográfica (latitude e longitude). A informação associada a essa coordenada poderá ser algo tão simples como a entrada em determinado local, por exemplo a entrada num restaurante. Actualmente a rede social que melhor representa este conceito é o *Foursquare*. Uma das partes mais importantes, e potencialmente problemática caso não exista uma boa arquitectura, de um sistema deste género é o **SGBD** pois sendo o mesmo virado para a partilha de informação existe grande volume de dados em circulação uma vez que é necessário não só armazená-los como disponibilizá-los.

No decorrer deste projecto foram estudadas várias tecnologias, *open source* e próprias (proprietary), de forma a responder aos requisitos de uma rede social orientada à partilha de informação georreferenciada. Para as tecnologias que respondem a esses requisitos foram implementados protótipos funcionais de forma a estudar e testar as mesmas quanto a desempenhos de escrita e leitura num ambiente distribuído (*Sharding*). Os testes efectuados são constituídos por um elevado número de escritas e leituras, utilizando dados e interrogações geoespaciais, de forma a testar o desempenho de cada uma das tecnologias face aos requisitos de uma rede social.

Abstract

The exponential evolution of technology has allowed each and everyone, who's able to acquire it, an easy access to it. Internet is most likely the best example of it because nowadays anyone who has a smartphone can use it. With this global availability emerged the concept of social network. A social network is, in its essence, a community of people connected, or not, between themselves that share common interests and that use some sort of technology to communicate between themselves with the goal of sharing information and/or resources.

Nowadays there are many types of social networks, being enterprise oriented, e.g. Linkedin, or personal and commercial, e.g. Facebook. However, there are also social networks that share georeference information, i.e. information that’s related to geographic coordinates (latitude and longitude). The associated information might be something as simple as checking in at someplace, i.e. checking in at a restaurant. The best example of this kind of social network is Foursquare. One of the most important parts, and possibly troublesome if there's not a good architecture, of a system like this is the **DBMS** because being oriented to sharing georeferenced information there's a big volume of data to store and make available.

To answer the requisites of a social network oriented to georeferenced information, several open source and proprietary technologies were studied. Functional prototypes were implemented, for the technologies that answer those requisites, in order to study and test their performance of writes and reads on a sharded environment. The prototypes were tested on consecutive and concurrent writing and reading, using geospacial data and queries, in order to test the performance of each technology that fulfils the requirements of a social network.

Palavras-chave

Keywords

Palavras-chave

NoSQL, Modelo relacional, Rede social, Pontos de interesse, georreferenciação, Sistema de gestão de base de dados, MongoDB, SQL Server

Keywords

NoSQL, Relational model, Social network, Points of interest, georeference, Database management system, MongoDB, SQL Server

Agradecimentos

A toda a minha família pelo apoio e força que me deram e em particular aos meus pais por tudo o que me têm dado e por acreditarem em mim.

Aos meus orientadores Walter Vieira e António Costa pelo aconselhamento, disponibilidade e ajuda.

Ao colega André Costa pelas muitas horas de boa disposição no trabalho.

Acrónimos

GPS	*Global Positioning System*
SGBD	Sistema de Gestão de Base de Dados
DBMS	*Database Management System*
API	*Application Programming Interface*
REST	*Representational state transfer*
HTTP	*Hypertext Transfer Protocol*
URL	*Uniform Resource Locator*
BASE	*Basically Available, Soft state, Eventual consistency*
CAP	*Consistency, Availability, Partition tolerance*
ACID	*Atomicity, Consistency, Isolation, Durability*
CLR	*Common Language Runtime*
CSV	*Comma-separated values*

Convenções Tipográficas

Apresentam-se de seguida as convenções tipográficas utilizadas na escrita deste documento:

I. Aplica-se no texto a fonte *Calibri* com tamanho 12, espaçamento entre linhas de 1,5 cm e 12 pontos de espaçamento antes um parágrafo;

II. Aplicam-se números entre parêntesis rectos para representar uma referência bibliográfica;

Exemplo: R-TREE [2]

III. Aplica-se texto em itálico para identificar termos estrangeiros para os quais não foram encontradas traduções razoáveis;

Exemplo: *hardware*

IV. Aplica-se texto em negrito para identificar acrónimos.

Exemplo: **GPS**

Índice

Índice de figuras

Índice de código

Índice de tabelas

1 Introdução

A implementação de uma solução para uma rede social poderá ser uma tarefa complexa dependendo do tipo de rede social e do público-alvo. Este tipo de soluções é, normalmente, assente numa arquitectura distribuída composta por servidores e balanceadores de carga e, idealmente, distribuída geograficamente. Para gestão e utilização dessa arquitectura é necessário implementar uma, ou mais, aplicações cliente que possam ligar-se e operar sobre a mesma (seja para administração de toda ou parte da infra-estrutura ou para inserção e consulta de informação). Existem vários tipos de arquitecturas distribuídas, sejam elas orientadas a telecomunicações, a processamento em tempo real, por exemplo controlo aéreo de aviões, ou até mesmo para computação paralela, por exemplo em renderização gráfica. Além dos tipos enumerados existem também arquitecturas distribuídas de bases de dados, [1], em que uma base de dados está distribuída por múltiplos computadores. Uma vez que os dados são distribuídos entre vários computadores (nós), existe menos sobrecarga de cada um deles o que faz com que a utilização deste tipo de base de dados possibilite um aumento no desempenho das aplicações que as utilizam e, dependendo do tipo de replicação, poderá fornecer autonomia local de um nó. No entanto, a utilização e manutenção de uma base de dados deste tipo levanta vários problemas relacionados com a consistência da mesma, uma vez que havendo distribuição de dados, é, em geral, necessário que todos os computadores apresentem a mesma imagem dos dados. Numa base de dados distribuída, esta consistência obriga, em geral, a transacções distribuídas (portanto mais lentas) e a mecanismos de replicação síncrona da informação. Outro dos problemas inerentes é a escalabilidade da base de dados. A escalabilidade de uma base de dados pode tornar-se complexa uma vez que é necessário ter em conta a tecnologia da mesma, os requisitos da aplicação e os custos. Uma base de dados que permita a partilha de informação georreferenciada poderá ter problemas de escalabilidade uma vez que os índices criados sobre as colunas geográficas têm um grande peso tanto na escalabilidade como na distribuição da informação

1.1 A rede social de dados georreferenciados *Blipper*

A aplicação, chamada *Blipper*, que irá utilizar o cenário anteriormente descrito de uma base de dados distribuída, segue a filosofia de uma rede social orientada à partilha de informação georreferenciada, ou seja, informação associada a coordenadas, e tal como o *Foursquare* pretende fornecer uma forma fácil e intuitiva de partilhar actividades e interesses em pontos geográficos que o utilizador considere relevantes. Estas actividades e/ou interesses consistem em informação da coordenada geográfica (ponto), uma ou mais *tags* que descrevem o ponto, uma data de inserção no sistema e a relevância do novo ponto. A informação relativa aos pontos é disponibilizada ao utilizador numa interface semelhante à interface disponibilizada num navegador **GPS**. A data de inserção de um ponto no sistema é utilizada para atribuir tempo de vida aos pontos, ou seja, cada ponto tem um tempo de vida pré estabelecido de, por exemplo, 30 minutos o que irá fazer com que existam pontos que percam relevância com o passar do tempo. À medida que começam a existir um grande número de pontos que se sobreponham num determinado local, é criado um efeito semelhante ao existente nos mapas meteorológicos, como apresentado na Figura 1.

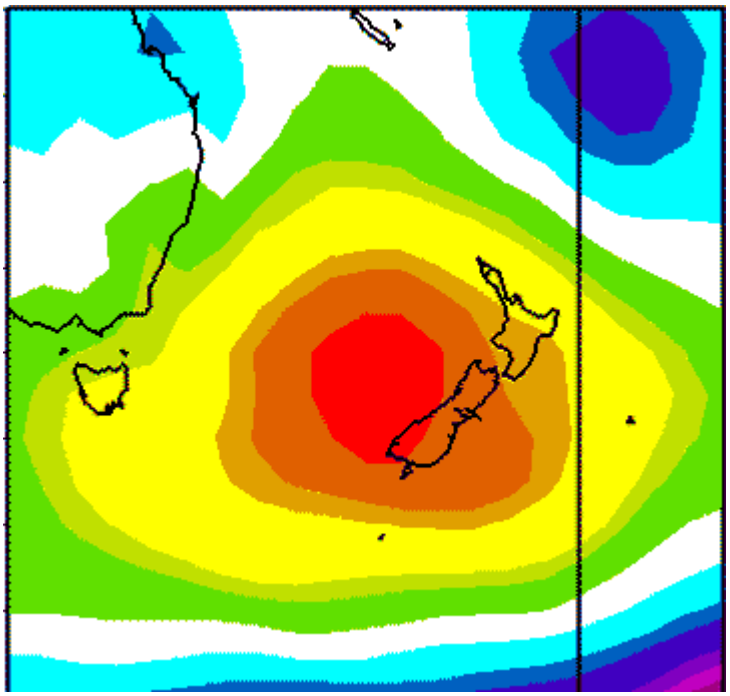


Figura 1 - Zona de alta pressão num mapa meteorológico

Além destas particularidades, a aplicação pode ser utilizada de uma forma anónima, ou seja, a identidade do utilizador que partilhou a informação não é partilhada com os outros utilizadores, no entanto a meta-informação associada ao ponto é partilhada.

1.2 Motivação

A implementação e manutenção de um **SGBD** altamente escalável, disponível e tolerante a falhas é um desafio que se pode tornar altamente complexo dependendo do objectivo para o qual o mesmo é desenvolvido. Tendo em conta que uma rede social é provavelmente um dos tipos de aplicações mais utilizadas nos dias de hoje, existem portanto muitos utilizadores em cada uma delas e existindo muitos utilizadores existe também muita informação. É fundamental garantir que um **SGBD** que dê suporte a uma aplicação deste género tenha possibilidade de crescer à medida das necessidades, portanto que tenha boa escalabilidade. Além da escalabilidade é importante que o **SGBD** tenha boa disponibilidade.

O cenário que serve de base à realização deste projecto é caracterizado por um **SGBD** que seja altamente escalável, distribuído e com grande disponibilidade para leituras e escritas de informação de forma a dar suporte a uma aplicação orientada à partilha de informação georreferenciada. Pelo facto da aplicação ser uma rede social, será potencialmente um sistema com muitos utilizadores e, portanto, terá muitos pontos de interesse registados e muitas pesquisas. Um aspecto crucial a tratar será a conciliação entre a necessidade de representação de informação georreferenciada e a necessidade de boas características de desempenho e escalabilidade. Além de desempenho e escalabilidade, também serão considerados aspectos relacionados com os níveis de disponibilidade, incluindo *disaster recovery*.

O objectivo deste trabalho é o desenvolvimento de protótipos funcionais que permitam estudar e testar as vantagens e desvantagens de vários **SGBD**s (*open source* e próprios) que permitam alta escalabilidade, suporte de dados georreferenciados e velocidade de acesso a dados e escrita dos mesmos.

Uma vez que os índices são fundamentais para a eficiência da interrogação a uma base de dados, é importante fornecer suporte a processamento distribuído de interrogações a dados georreferenciados.

Relativamente aos dados georreferenciados, as técnicas de indexação mais eficientes são específicas de modelos de representação geográfica e utilizam estruturas de dados do tipo R-TREE, [2], o que pode eventualmente provocar efeitos negativos na escalabilidade. Ao distribuir a informação com *Sharding*, está-se automaticamente a distribuir a carga de indexação e de pesquisa o que pode gerar problemas de escalabilidade num cenário em que também é necessário particionamento dos índices.

1.3 Organização do documento

Este documento está organizado em 6 capítulos organizados da seguinte forma:

1 No primeiro capítulo é descrito o cenário do trabalho bem como realizada uma introdução à aplicação *Blipper*;

2 No segundo capítulo são apresentados os principais modelos de bases de dados bem como os estudos efectuados;

3 No terceiro capítulo são apresentados os casos de utilização da aplicação, avaliados os modelos e algumas das tecnologias envolvidas no desenvolvimento bem como as arquitecturas propostas para a infra-estrutura de acesso a dados;

4 No quarto capítulo são apresentadas as implementações das arquitecturas propostas para a infra-estrutura de acesso a dados, da aplicação utilizada para testar os **SGBD**s e de um protótipo da aplicação *Blipper*;

5 No quinto capítulo são apresentados os resultados dos testes bem como as conclusões e trabalho futuro.

2 Estado da Arte

Neste capítulo são descritos e discutidos os principais modelos de bases de dados relacional e NoSQL. Sobre estes modelos são enumeradas as principais características de cada um deles e descritas as formas como cada um deles poderá responder aos requisitos da aplicação.

Como referido anteriormente, a escolha de um **SGBD** pode ser complexa tendo em conta os objectivos e os requisitos da aplicação que necessita do mesmo. Antes de ser sequer ponderado qualquer **SGBD**, devem ser primeiro analisados vários aspectos.

- Tipo de dados a armazenar:

  Diz respeito à informação que a aplicação irá gerar, que necessita de ser armazenada no **SGBD** e que pode influenciar a escalabilidade uma vez que a mesma é directamente afectada pelos índices criados sobre os dados. Aplicando a escalabilidade e os índices dos dados ao objectivo concreto deste trabalho, um **SGBD** que não suporte a criação de índices sobre dados geográficos pode fazer com que o desempenho de leitura dos dados seja pior, no entanto um **SGBD** que suporte esses índices deverá ter melhor desempenho para leituras mas pode criar dificuldades no escalamento de todo o sistema porque é necessário ter em conta o particionamento desses índices ao projectar o escalamento.

- Tipo de acessos:

  Define de que forma é que a informação pode ser acedida pela aplicação e podem ser classificados como:

  1 Leitura: Quando existem acessos ao **SGBD** para leitura de informação sem modificar a mesma;

  2 Escrita: Quando existem acessos ao **SGBD** para inserção ou actualização de informação;

No caso de existir concorrência no acesso aos dados, existem potenciais problemas de consistência. Estes problemas são tipicamente controlados através do nível de isolamento e isto é normalmente conseguido à custa de transacções e possíveis *locks* associados (dependendo do nível de isolamento da transacção). No entanto se existirem falhas na utilização deste mecanismo é possível criar *deadlocks* e originar a que parte dos dados, ou mesmo todos, fiquem inacessíveis posteriormente. A clara definição do tipo de acessos vai permitir definir se, no contexto da aplicação, são necessárias transacções ao nível do **SGBD**.

- Escalabilidade:

A escalabilidade de uma base de dados é um dos pontos mais importantes para permitir que uma aplicação possa expandir o seu armazenamento de informação, isto é, caso seja uma aplicação que necessite de muitos acessos para escritas/leituras poderá tornar-se essencial que o **SGBD** possa ser facilmente escalável. A escalabilidade é classificada como:

1. Horizontal (*Scale out*): Em que é aumentado o número de máquinas que suportam todo o **SGBD**;

2. Vertical (*Scale up*): Em que é feita uma actualização de *hardware* das máquinas que suportam o **SGBD**.

- Disponibilidade.

2.1 Escalabilidade vs Disponibilidade

A escalabilidade está relacionada com recursos e quanto mais recursos forem necessários mais difícil é escalar o sistema. A disponibilidade também está relacionada com recursos na medida em que as soluções de disponibilidade passam por aumentar o número de recursos, ou seja, por ter recursos redundantes. Pode dizer-se que tanto a escalabilidade como a disponibilidade são influenciadas pela replicação.

Em geral, existem ganhos na disponibilidade ao desenvolver-se uma solução com alta escalabilidade devido à replicação. No entanto, se a replicação for síncrona, pode-se comprometer a disponibilidade do sistema porque o processo de replicação só termina quando todos os nós se encontram no estado mais actual, ou seja, o sistema pode ficar bloqueado para escritas enquanto a replicação não terminar e isto tende a agravar-se ainda mais conforme o número de nós no sistema aumenta. Mas a disponibilidade é uma consequência da replicação tanto de *software*, como de *hardware* e de informação o que quer dizer que para aumentar a disponibilidade de um sistema, é necessário aumentar o número de réplicas. No entanto, este aumento pode ter um efeito adverso na escalabilidade caso o número de novas réplicas seja muito elevado.

2.2 Replicação

À medida que o volume de informação aumenta, torna-se cada vez mais difícil o escalamento vertical uma vez que o *hardware* tem limite e quanto mais recente for maior é o custo. Assim torna-se indispensável que uma solução de base de dados consiga escalar de forma fácil, horizontalmente e com baixo custo. Dependendo do modelo de distribuição adoptado é possível ter melhor desempenho de escritas e/ou leituras, maior capacidade para lidar com maiores volumes de informação ou maior disponibilidade.

A replicação pode ser classificada de duas formas:

- Síncrona (garante consistência): Quando existe uma inserção ou actualização de informação num nó, essa informação tem que ser propagada para os outros nós (réplicas). Neste tipo de replicação, só existe sucesso quando todas as réplicas forem actualizadas o que pode originar a que, caso exista indisponibilidade de um nó no sistema, a replicação não termine com sucesso. Para além disso, o tempo de escrita é aumentado devido à propagação da nova informação. Estas duas consequências deste tipo de replicação podem afectar para pior a

disponibilidade de todo o sistema. Este tipo de replicação geralmente envolve uma transacção distribuída.

- Assíncrona (não garante consistência): Neste tipo de replicação, a inserção ou actualização de informação num nó pode terminar com sucesso sem esperar pela sua escrita nos outros nós e a infra-estrutura é que fica responsável por propagar a nova informação pelos outros nós. Uma vez que o processo não é síncrono como o anterior, este tipo de replicação pode originar a que as réplicas apresentem valores diferentes durante períodos de tempo mais ou menos longos, ou seja, não garante a consistência da informação, no entanto favorece a disponibilidade do sistema.

2.3 Modelo Relacional

Existem duas características principais do modelo relacional:

- Desacoplamento entre a representação lógica e a representação física;
- Propriedades **ACID**.

Neste modelo, as ligações entre os elementos não são físicas mas sim lógicas (através de chaves) o que origina a uma separação das representações lógica, composta pela definição das tabelas, e física, composta pelos índices, particionamento, etc. Esta separação de representações tem como consequência uma grande adaptabilidade do modelo relacional pois a manipulação das características da representação física permite que a base de dados seja optimizada para um cenário em concreto e uma vez que o modelo relacional tem essa capacidade de optimizar estas características físicas, é possível melhorar o desempenho, em função das necessidades, para interrogações não conhecidas à priori (interrogações *ad-hoc*). Essas optimizações são feitas tendo por base ferramentas específicas e podem consistir na criação, alteração e remoção de índices. Assim, com a evolução e alteração das interrogações e manipulações de dados efectuadas sobre a base de dados, é possível optimizar a base de dados de forma a ter melhor desempenho para as interrogações mais críticas. A possibilidade de optimizar o

desempenho à medida das interrogações mais problemáticas, conforme as necessidades, torna o modelo relacional o mais adequado quando se desenvolve um sistema em que as interrogações não são conhecidas à priori.

Habitualmente o que é associado ao modelo relacional são as propriedades **ACID** mas é necessário ter em conta que as implementações actuais permitem que o programador controle quais destas propriedades pretende utilizar. Ao encapsular um determinado número de instruções numa transacção, garante-se a propriedade A (Atomicidade) [3], no entanto tal não é obrigatório e a não utilização de uma transacção origina a que cada instrução seja executada no modo *auto-commit* e portanto a atomicidade só é garantida ao nível de cada instrução. A propriedade C (Consistência), [4], está relacionada com a escrita em âmbito transaccional e é utilizada para garantir que um sistema transita de um estado consistente para outro consistente. Para garantir o isolamento entre instruções concorrentes é utilizada a propriedade I, [5].

2.4 NoSQL

NoSQL é um termo utilizado para definir as bases de dados que têm como características o facto de serem não relacionais, distribuídas, *open source* e facilmente escaláveis horizontalmente. A intenção original era a de criar bases dados modernas orientadas a um grande volume de informação, no entanto existem outras características associadas a **SGBD**s NoSQL como a não imposição de um esquema de dados, suporte fácil a replicação de informação, uma **API** simples, consistência eventual (não é imediata o que significa que a base de dados vai ficar eventualmente consistente com a passagem do tempo) – **BASE** (oposto de **ACID**) e grande volume de informação, [6].

O acrónimo **BASE** foi definido por Eric Brewer [7], também conhecido pela sua formulação do Teorema **CAP** (*Consistency, Availability, Partition Tolerance*) [8], e significa *Basically Available*, *Soft state*, *Eventual consistency*. Ou seja, *Basically Available* significa que um sistema deste género permite uma boa disponibilidade (*Available)*. *Soft state* significa que o estado do sistema pode ser alterado com o decorrer do tempo mesmo sem introdução de nova informação no mesmo. Por último, *Eventual consistency*

indica que o sistema irá ficar eventualmente consistente com o decorrer do tempo. Estes dois últimos pontos dizem respeito à consistência do sistema e, de acordo com o Teorema **CAP**, é o ponto que é sacrificado em prol dos outros dois, Disponibilidade (*Availability*) e Particionamento (*Partition Tolerance*).

Assim, o termo NoSQL é aplicado para agrupar um número de **SGBD**s que não utilizam um esquema de dados e abdicam da consistência (**BASE**) para favorecer outras características importantes, num sistema com grande volume de informação, como a replicação, escalabilidade e distribuição.

2.4.1 Principais modelos de dados NoSQL

Existem quatro principais modelos de dados NoSQL:

- Key-Value Stores (par chave-valor);
- Document Stores (orientado a documentos);
- Wide Column Stores (orientado a famílias de colunas);
- Graph (Grafo).

Cada um destes tipos tem características diferentes dos restantes o que faz com que cada um deles tenha propósitos diferentes.

Nos dois primeiros modelos de dados não existe o conceito de tabela, no entanto para tornar mais simples a explicação de ambos utiliza-se o termo tabela para identificar uma entidade semelhante a uma tabela do modelo relacional. [9]

2.4.1.1 Key-Value Stores (par chave-valor)

O modelo de dados Key-Value Store (KVP) é um modelo simples em que apenas é necessário uma chave (*key*) para aceder aos valores (*value*). Esta chave poderá ser algo

tão simples como o número do bilhete de identidade que identifica os dados relativos a uma pessoa.

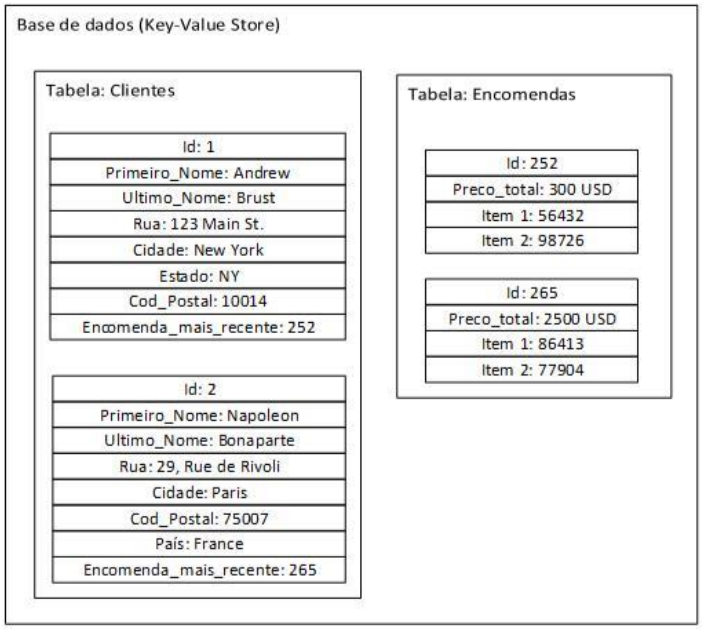


Figura 2 - Key-Value Store (adaptado de "Windows Azure No SQL White Paper")

A Figura 2 ilustra a estrutura de um possível modelo de dados KVP. Neste cenário existem duas tabelas, Clientes e Encomendas. Cada uma das tabelas tem o atributo ID como chave. O valor (informação) associado a cada chave, neste caso, corresponde à informação referente a um cliente, no caso da tabela Clientes, e à informação de uma encomenda, no caso da tabela Encomendas. Por ser um modelo em que apenas uma chave é suficiente para aceder aos dados, será, dependendo do cenário, o mais adequado quando não são necessárias interrogações ricas. Este tipo de modelo de dados é mais frequentemente utilizado quando não existe manipulação de dados. Um dos cenários para o qual é mais utilizado é a gestão de carrinhos de compras. Um exemplo deste tipo de modelo é o *Dynamo*, proprietário da *Amazon*, ou o *RIAK* que se trata duma implementação *open source* do *Dynamo*.

2.4.1.2 Document Stores (orientado a documentos)

Num modelo de dados deste tipo são armazenados documentos sendo cada um composto por vários pares chave-valor. Ao contrário do modelo relacional, é possível não só armazenar informação simples mas também informação complexa como outros documentos ou listas de valores. Normalmente estes documentos são persistidos no formato JSON. Habitualmente este tipo de modelos de dados fornece um **URL** de acesso ao documento e uma interface simples, de acesso ao mesmo, **REST** via **HTTP**, ou seja, é possível aceder ao conteúdo do documento através do **URL** do mesmo e de um *browser*.

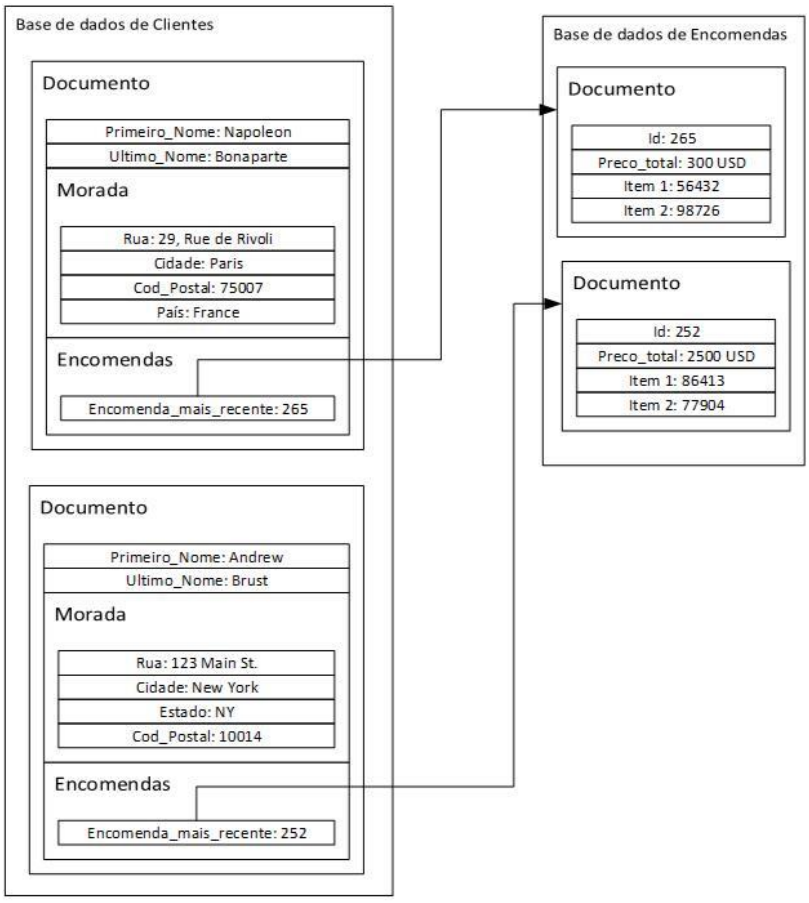


Figura 3 - Document Store (adaptado de "Windows Azure No SQL White Paper")

Na Figura 3 é ilustrado o exemplo de uma estrutura de um modelo de dados orientado a documentos. À semelhança do anterior, é composto pelas colecções de documentos Clientes e Encomendas. No entanto, neste exemplo, o campo Morada é composto por

uma lista de valores que identificam a rua, a cidade, o estado e o código postal (Cod_Postal). Neste exemplo existe ainda o campo Encomendas que contém um apontador para a Encomenda com o identificador correspondente, no entanto seria possível juntar as várias encomendas, sob a forma de uma lista, nesse campo caso fosse pretendido. Actualmente existem vários sistemas que seguem este modelo de dados, sendo o MongoDB, abordado mais à frente, um dos mais conhecidos e utilizados.

2.4.1.3 Wide Column Stores (orientado a famílias de colunas)

A maior parte dos **SGBD**s tem como unidade mínima de escrita uma linha (*row*) o que ajuda à performance de uma escrita. No entanto, por vezes existem situações em que não existem tantas escritas quanto leituras mas é necessário ler várias linhas de uma só vez. Nestas situações é melhor guardar grupos de colunas para todas as linhas. É nestes cenários que se torna útil um modelo de dados como este. Este tipo de modelo de dados organiza as colunas em famílias de colunas em que cada uma é parte de uma única família e é utilizada para acesso ao valor dessa coluna.

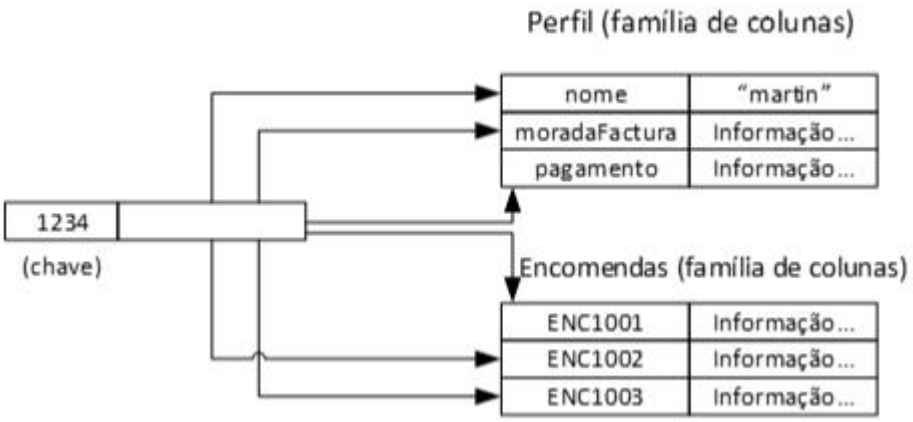


Figura 4 - Wide Column Store - Funcionamento (adaptado de "Windows Azure No SQL White Paper")

A Figura 4 ilustra um exemplo de acesso a um modelo de dados orientado a família de colunas. As famílias de colunas são indicadas como Perfil e Encomendas. Neste caso concreto, é possível não só obter um valor específico para uma coluna mas também obter todos os valores para toda a família. Por exemplo, ao executar a instrução

get('1234','nome') obtêm-se o nome do cliente com o número 1234 e ao executar a instrução *get('1234','encomendas')* obtêm-se todas as encomendas relativas a esse cliente (estas instruções são meramente utilizadas como exemplo apenas para explicar o funcionamento). [10]

Figura 5 - Wide Column Store - Estrutura (adaptado de "Windows Azure No SQL White Paper")

Duas tecnologias que utilizam este modelo de dados são o *BigTable* da *Google* e o *Cassandra* da *Apache*. No entanto o *Cassandra* utiliza uma terminologia equivalente mas diferente. Uma tabela é uma *Super Column Family* e uma família de colunas é uma *Super Column.*

A Figura 5 ilustra a possível estrutura de um modelo de dados deste género em que "T" simboliza uma tabela, "CF" uma família de colunas e "C" uma coluna. O equivalente no *Cassandra* é "SCF" uma *super column family*, "SC" uma *Super Column* e "C" uma coluna também.

As interrogações nestes sistemas são limitadas e pré-definidas sendo os relacionamentos entre os dados realizados pelas aplicações. Isto torna-se uma limitação relativamente ao modelo relacional.

2.4.1.4 Graph (Grafo)

No modelo de dados Graph de NoSQL, a informação é estruturada recorrendo a nós (nodes), relações (edges) e propriedades para representar e armazenar informação. Este tipo de dados tem a vantagem de não necessitar de índices uma vez que cada elemento contém um apontador directo para os seus elementos adjacentes. Neste tipo de modelo, a informação é tratada de uma forma semelhante a um modelo de dados RDF. Os nós podem ser compostos por uma ou várias propriedades e ao conjunto nó – relação – nó é dado o nome de asserção.

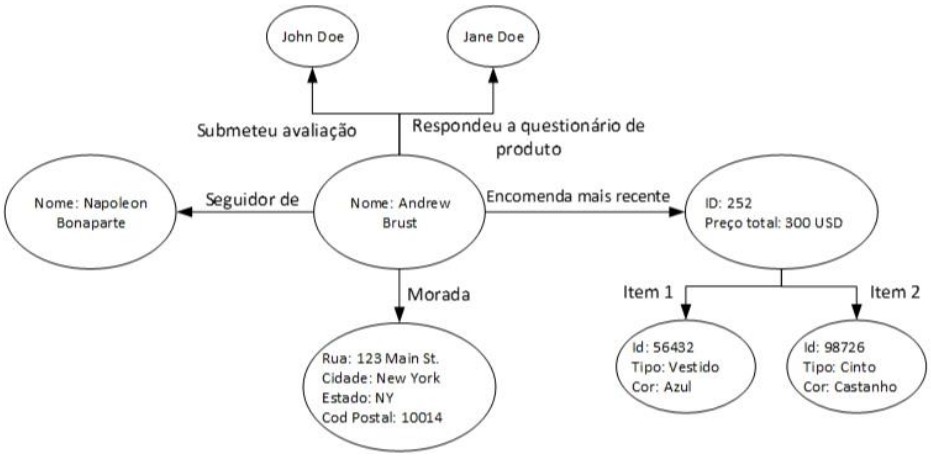


Figura 6 - Graph (adaptado de "Windows Azure No SQL White Paper")

A Figura 6 ilustra um exemplo de um possível modelo de dados do tipo Graph.

2.5 Distribuição de dados e escalabilidade

Existem duas formas de se utilizar a distribuição de dados de forma a se conseguir escalar a base de dados. A replicação permite copiar a mesma informação para outros nós no *cluster* e o *Sharding* permite distribuir a informação em diferentes nós de forma particionada. É possível utilizar apenas uma das duas ou ambas. Não existe uma forma pré definida de utilizar *Sharding* no modelo relacional, no entanto é possível fazê-lo tirando partido das características do mesmo (com particionamento horizontal, *linked servers* e *stored procedures* no caso do Microsoft SQL Server).

2.5.1 Sharding

Como descrito anteriormente, o *Sharding* permite distribuir a informação por diferentes nós.

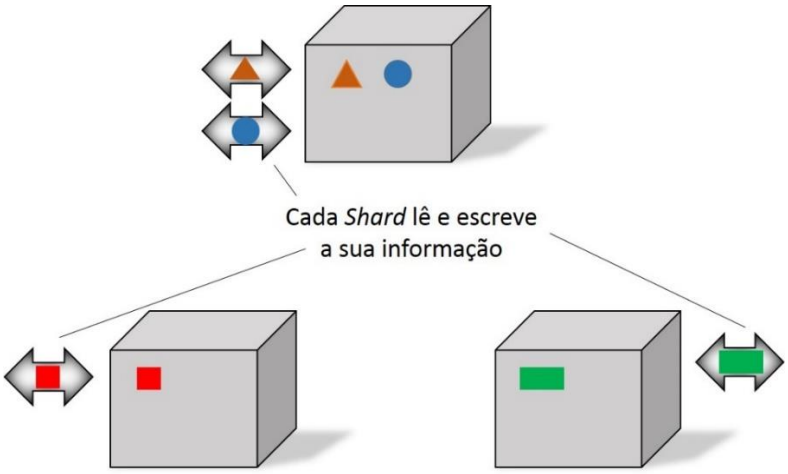


Figura 7 - Sharding (adaptado de "NoSQL Distilled")

No cenário ilustrado na Figura 7, cada utilizador tem uma ligação com servidores diferentes o que faz com que a carga de comunicações seja distribuída. Um dos factores a ter em conta neste tipo de distribuição é a igualdade da mesma, ou seja, tentar que a distribuição da informação seja feita de forma igual para todos os nós presentes no sistema de forma a distribuir também a sobrecarga pelo sistema. Muitos **SGBD**s NoSQL

"oferecem" *auto-sharding,* por ser mais fácil fazê-lo que no modelo relacional dada a diferença dos modelos de dados, em que a base de dados tem a responsabilidade de alocar a informação aos *Shards* e assegurar que o acesso é feito ao *Shard* correcto. Este tipo de distribuição é vantajoso tanto para leituras quanto para escritas no entanto não fornece tolerância a falhas quando não utilizado em conjunto com um modelo de replicação.

2.5.2 Replicação

A replicação tem vantagens e desvantagens, sendo a inconsistência a desvantagem mais evidente no caso da replicação assíncrona. A partir do momento em que existe replicação assíncrona de informação e existem clientes a efectuar leituras em nós diferentes do sistema, é possível que a informação seja diferente nesses nós o que irá originar a que os clientes estejam a visualizar informação diferente uma vez que a informação ainda não foi propagada na totalidade pelo sistema.

A replicação pode ser categorizada em:

- Replicação Master-Slave;
- Replicação P2P ou *multi-master*;

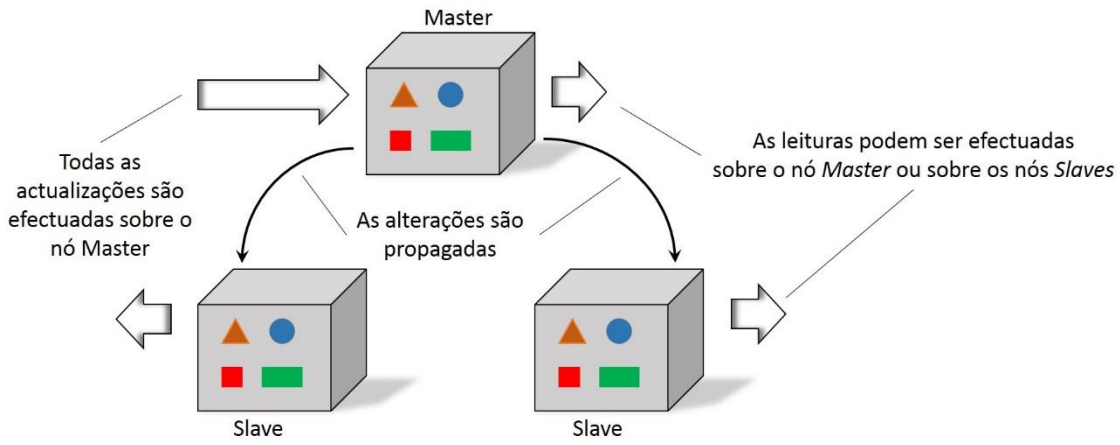


Figura 8 - Modelo de replicação Master-Slave (adaptado de "NoSQL Distilled")

No modelo *Master-Slave*, Figura 8, a informação é replicada por vários nós em que um é designado de *Master* e os outros *Slaves*. O nó *Master* é responsável por processar e replicar a informação.

Uma das vantagens deste tipo de replicação é a de facilitar o escalamento horizontal para leitura que é feito através da adição de mais nós *slaves*. No entanto, este tipo de replicação fica limitado à capacidade do *master* em lidar e propagar actualizações. Outra das vantagens é a resistência a falhas para leituras uma vez que em caso de falha do *master*, os *slaves* continuam a conseguir responder a interrogações. No entanto a falha do *master* pode fazer com que o sistema fique indisponível para escritas até que o *master* esteja novamente disponível mas neste caso é possível promover, manualmente ou automaticamente, um *slave* a *master.* A inconsistência pode ser uma grande desvantagem deste modelo uma vez que havendo falha do *master* qualquer actualização efectuada posteriormente poderá ser perdida o que significa que o *master* é o único ponto de falha.

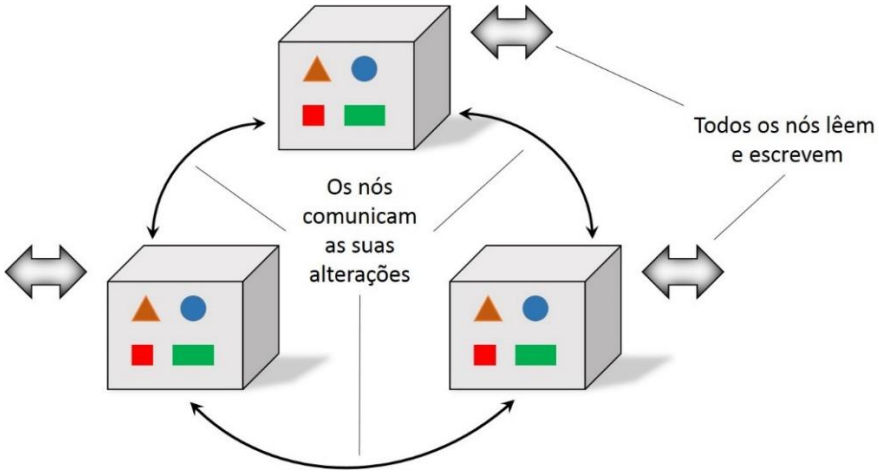


Figura 9 - Modelo de replicação P2P (adaptado de "NoSQL Distilled")

No tipo de replicação P2P, Figura 9, não existe a noção de *master* uma vez que todas as réplicas têm o mesmo estatuto, o que significa que podem aceitar escritas e a perda de uma delas não limita o acesso aos dados. Uma vez que todas as réplicas contêm, eventualmente, toda a informação, no caso de existir falha de uma o sistema continua a permitir o acesso à informação. Além disso, é possível adicionar novas máquinas para

aumentar a performance. No entanto, ao ser possível escrever em dois nós, é possível que dois utilizadores tentem actualizar a mesma informação em nós diferentes, o que pode levar, mais uma vez, à inconsistência. Mas existem cenários onde é possível fazer partição (funcional ou outra) de forma a evitar conflitos, por exemplo numa partição de acessos por fuso horário e existindo réplicas em continentes diferentes a funcionar em alturas diferentes, é possível configurar os acessos a cada nó de forma a que os mesmos não se sobreponham.

Actualmente os **SGBD**s que seguem o modelo relacional suportam replicação síncrona e assíncrona e sobre eles é também possível realizar *Sharding*.

3 Solução Proposta

Neste capítulo são apresentados os casos de utilização da aplicação, os modelos e algumas das tecnologias investigadas no desenvolvimento e apresentadas as arquitecturas implementadas.

Para apresentar possíveis soluções de **SGBD**s que possam satisfazer os requisitos da aplicação *Blipper*, foram estudadas várias tecnologias, tanto do modelo relacional como NoSQL, mas nem todas respondem a todos os requisitos da aplicação. Para que um **SGBD** possa ser considerado como possível solução para a aplicação *Blipper* é necessário que preencha os requisitos da aplicação, ou seja, que permita os tipos de dados, indexação e pesquisa correctos, que tenha facilidade de escalar horizontalmente com tolerância a falhas e que permita a utilização de técnicas de *Sharding*.

3.1 Casos de utilização da aplicação *Blipper*

A aplicação *Blipper* disponibiliza uma interface semelhante à interface de um navegador **GPS** onde é apresentado um mapa e informação relativa aos pontos de interesse ao redor do utilizador, tal como descrito no capítulo 1. Esta informação deverá ser actualizada periodicamente de forma a mostrar a informação mais recente em qualquer instante temporal. O facto de essa informação estar dependente da localização do utilizador e ser actualizada periodicamente, faz com que o desempenho de leituras do **SGBD** seja de grande importância. Além dessa actualização frequente, a aplicação disponibiliza uma forma de obter pontos de interesse, ao redor do utilizador, com base numa *tag*. Para além de acesso aos dados, o utilizador poderá também inserir um novo ponto, associado à sua localização actual, que consiste na informação da coordenada geográfica bem como o nome do ponto e a *tag*, ou conjunto de *tags,* que irão ser utilizadas para identificar o ponto.

Assim, existem três casos de utilização principais da aplicação:

1 – Obtenção de todos os pontos de interesse ao redor do utilizador;

2 – Obtenção de todos os pontos de interesse ao redor do utilizador através de uma *tag*;

3 – Inserção de um novo ponto.

Uma vez que na aplicação *Blipper* existe uma actualização frequente dos pontos de interesse em redor do utilizador, é esperado que existam muito mais leituras do que escritas.

3.2 Tipos de dados

No contexto da aplicação, existem apenas duas entidades representadas no **SGBD**:

- Utilizador: Composto por nome, data de nascimento e email (identificador);
- Ponto: Composto por nome, latitude, longitude, blip (latitude,longitude), tag, código do país, data e hora de inserção e o identificador do utilizador.

3.3 Indexação

Como a maior parte dos acessos à informação são baseados em coordenadas geográficas, é essencial que o **SGBD** suporte índices geoespaciais. Isto vai fazer com que exista uma melhor performance em interrogações sobre coordenadas geográficas. Uma vez que as interrogações são baseadas nos pontos de interesse em redor do utilizador, é também essencial que o **SGBD** permita a utilização de funções geoespaciais (por exemplo STDistance no caso do Microsoft SQL Server e GeoNear no caso de MongoDB).

3.4 Requisitos do SGBD

Tendo em conta as características da aplicação *Blipper,* a probabilidade de crescimento do número de utilizadores e sua informação é elevado. Por isso, é aceitável que existam

períodos temporais em que não exista consistência entre todos os nós mas é essencial que esses períodos sejam o mais curto possível.

Assim, com base nos cenários de utilização identificados anteriormente e no tipo de aplicação, os requisitos fundamentais para considerar a utilização de um **SGBD**:

- Alta escalabilidade;
- Alta disponibilidade;
- Disaster recovery;
- Índices e interrogações geoespaciais;

3.5 Modelo Relational vs NoSQL

Uma das grandes diferenças entre um **SGBD** relacional e um NoSQL é que o relacional obriga a uma definição de esquema de dados, ou seja, à definição explícita de todas as colunas, e seus tipos, de todas as tabelas. O mesmo não é verdade para um **SGBD** NoSQL que permite que cada "tabela" sirva para guardar qualquer coisa, ou seja, não é necessário que se defina um esquema de dados. As principais vantagens desta abordagem é a flexibilidade do **SGBD** para armazenar qualquer tipo de informação sem estar a obedecer a um esquema e a facilidade de adicionar campos sem fazer alterações na estrutura o que facilita a manipulação de informação não uniforme, no entanto pode obrigar a cuidados na implementação da camada de acesso a dados uma vez que poderá haver diferenças na estrutura de cada. [11]

Um **SGBD** relacional é desenhado para correr numa máquina mas habitualmente é mais económico que a computação de grandes pedaços de informação sejam tratados em blocos de várias máquinas mais baratas. Grande parte dos **SGBD**s NoSQL são desenhados especificamente para serem executados em *clusters* o que os torna numa melhor escolha para lidar com grandes pedaços de informação. O escalamento horizontal num **SGBD** NoSQL é também uma característica importante visto que é um processo mais fácil de executar do que num **SGBD** relacional porque apenas é necessário

adicionar mais nós ao *cluster* e a informação é automaticamente distribuída para o mesmo. [12]

A concorrência no acesso aos dados por parte de um conjunto grande de utilizadores é habitualmente um aspecto complexo de abordar e tratar. Nos **SGBD**s relacionais é possível controlar o acesso concorrente aos dados através da utilização de transacções. No entanto as transacções podem causar graves problemas num ambiente distribuído e particionado (clustered) pelo que grande parte dos **SGBD**s NoSQL não têm transacções mas sim outras características para lidar com a consistência e distribuição dos dados. [13]

A informação num modelo relacional é organizada sob a forma de relações e tuplos. Esta fundação em relações torna-o num modelo simples de compreender e utilizar, no entanto existe a limitação de que os valores contidos num tuplo têm de ser simples, ou seja, não é possível que sejam, por exemplo, uma lista. Isto pode originar uma discrepância entre a representação dos dados ao nível do **SGBD** e da camada aplicacional. O mesmo não é verificado num **SGBD** NoSQL uma vez que os dados existentes na memória podem ser persistidos e lidos *as is* (da forma que estão). [14]

Assim, pode-se concluir que as principais características do modelo NoSQL são:

- Não segue o modelo relacional;
- Facilidade de escalamento;
- Open-source;
- Sem definição de esquema de dados.

Uma comparação mais extensiva pode ser consultada na secção “NoSQL vs SQL Summary” em [15].

3.6 Teorema CAP

O teorema CAP (*Consistency*, *Availability*, *Partition Tolerance*) foi formulado por Eric Brewer e refere que, no evento de uma partição de rede, um sistema distribuído pode fornecer disponibilidade ou consistência mas nunca as duas. Operações consistentes fornecem garantias de que as operações de leitura reflectem a última actualização bem sucedida à base de dados e são uma característica de, por exemplo, transacções. A consistência é relativamente adquirida num sistema composto apenas por um servidor, o que é comum nos sistemas tradicionais de bases de dados relacionais.

Este teorema ganha novas proporções à medida que uma aplicação escala. Em valores transaccionais baixos, latências baixas permitem que as bases de dados fiquem consistentes, no entanto, à medida que a actividade aumenta irão existir limites de crescimento que poderão resultar em erros e assim prejudicar a consistência. [8]

ACID e **BASE** representam duas filosofias nos extremos do espectro consistência-disponibilidade. As propriedades **ACID** focam-se na consistência e são a abordagem tradicional das bases de dados. Hoje em dia os sistemas mais modernos de larga escala, incluindo a *cloud*, utilizam uma mistura das duas abordagens. Embora ambos os termos sejam mais mnemónicos do que exactos, o acrónimo **BASE** é um pouco mais estranho porque *Soft state* e *Eventual consistency* (consistência eventual) são técnicas que funcionam bem na presença de particionamento e que promovem a disponibilidade.

A relação entre **CAP** e **ACID** é mais complexa e frequentemente mal interpretada em parte porque as propriedades A e C representam conceitos diferentes que as mesmas letras em **CAP** e em parte porque escolher a disponibilidade afecta apenas algumas das garantias **ACID**. [16]

No contexto deste trabalho, é mais importante que o sistema esteja disponível para escritas e leituras do que esteja sempre consistente. Assim, a disponibilidade é um aspecto mais importante do que a consistência pelo que é aceitável que existam instantes temporais em que a informação não está completamente consistente.

3.7 Escalabilidade, distribuição e disponibilidade

Para acomodar o crescimento de utilizadores da aplicação, é fundamental que o **SGBD** possa ser escalável horizontalmente. Isto é resolvido com a adição de máquinas à infra-estrutura do **SGBD**. No entanto, dependendo da tecnologia, esse processo pode ser mais ou menos complexo, como se irá verificar mais à frente. Inerente à escalabilidade está o problema da distribuição. Quando são adicionados novos nós à infra-estrutura do **SGBD** é necessário que toda a infra-estrutura seja actualizada para que não existam problemas de distribuição da informação. As tecnologias NoSQL permitem não só uma facilidade de escalamento horizontal como uma distribuição automática através de *Sharding* automático ou manual. Dependendo da arquitectura do **SGBD**, a disponibilidade de todo o sistema pode ser melhorada caso exista mais que um nó para escritas e/ou leituras, ou seja, se existir mais que um nó para escritas existe melhor disponibilidade do sistema para escritas e o mesmo acontece para leituras. Caso não exista redundância entre os nós, de modo a que em caso de falha de um exista pelo menos um outro que consiga efectuar o mesmo trabalho, a disponibilidade do **SGBD** é prejudicada. Assim para que um sistema tenha alguma tolerância a falhas é necessário que exista replicação. A replicação no modelo NoSQL é assíncrona na maioria das tecnologias o que causa o problema da consistência eventual referido no subcapítulo 2.4. No modelo relacional geralmente é necessário uma reestruturação de código (*stored procedures* e *linked servers*) no **SGBD** ou na camada de acesso a dados de forma a utilizar os novos nós introduzidos na infra-estrutura, o que pode dificultar o escalamento horizontal. O mesmo acontece quanto a um esquema de distribuição baseado em *Sharding*. Embora seja possível utilizar técnicas de *Sharding* para distribuir a informação, as mesmas terão que ser feitas manualmente o que significa que, tal como no escalamento, possa ser necessário reestruturar código existente no **SGBD** ou na camada de acesso aos dados. Para tolerância a falhas, o modelo relacional disponibiliza formas de replicação síncrona ou assíncrona. A replicação síncrona poderá ser vantajosa num cenário em que todos os nós do sistema devem ter em todos os instantes temporais a informação mais recente, no entanto, dependendo do esquema de replicação, pode levar a indisponibilidade do sistema para escritas enquanto todos os nós não estiverem actualizados.

3.8 Tecnologias

Neste subcapítulo são detalhadas e é explicado de que forma as tecnologias utilizadas no desenvolvimento satisfazem os requisitos da aplicação *Blipper.*

3.8.1 MongoDB

MongoDB é um **SGBD** NoSQL *open source* orientado a documentos (*Document Store*) desenvolvido para facilitar o escalamento e que actualmente é utilizado pela rede social *Foursquare* [17]. A arquitectura de *Sharded Cluster* existente em MongoDB permite alta facilidade de escalamento, disponibilidade e tolerância a falhas.

A arquitectura mais utilizada num ambiente escalável desenvolvido em MongoDB é denominada de *Sharded Cluster* e trata-se de um conjunto de nós composto por três processos de configuração (*config server*), um ou mais *replica sets* e um processo de encaminhamento de pedidos (*router*). No subcapítulo seguinte são detalhados todos estes componentes que constituem a arquitectura de um *Sharded Cluster* em MongoDB.

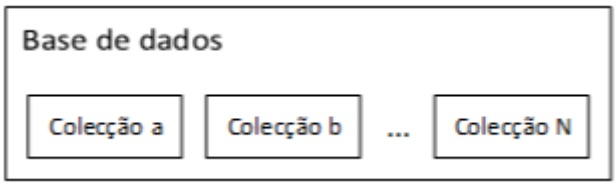


Figura 10 - Estrutura interna

A Figura 10 ilustra a estrutura interna de MongoDB. É composta por colecções (*collection*) dentro de bases de dados (*databases*) sendo que uma colecção é o equivalente a uma tabela do modelo relacional.

3.8.1.1 Arquitectura

Na Figura 11 é apresentada a arquitectura que define um *Sharded Cluster* em MongoDB. A arquitectura apresentada é composta por dois *routers* (aplicação mongos), três config servers e dois *Shards* configurados como *replica sets*. Um *replica set* é um *cluster* de instâncias MongoDB (aplicação mongod) que implementam, entre eles, replicação *master-slave* e mecanismos automáticos de *failover* baseados em *heartbeat* com ou sem *arbiter* (árbitro).

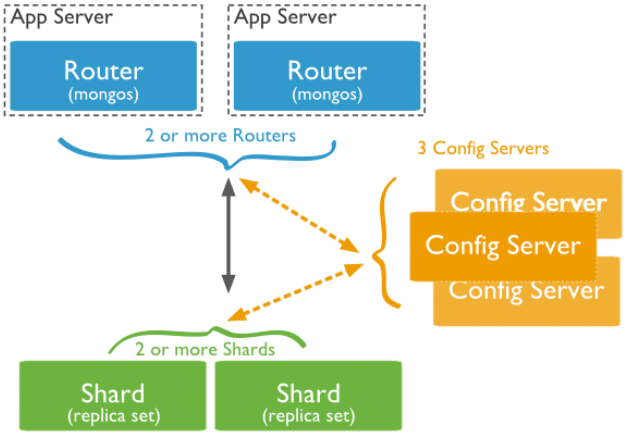


Figura 11 - Arquitectura Sharded Cluster (adaptado de "docs.mongodb.org")

3.8.1.2 Constituição do Sharded Cluster

Como descrito anteriormente, uma arquitectura *Sharded Cluster* é composta por *routers*, *Shards* e *config servers.* Cada uma destas partes tem responsabilidades diferentes no funcionamento de todo o *cluster*. Um *router* tem como responsabilidades encaminhar pedidos, processar as interrogações da aplicação, determinar o local da informação armazenada no *cluster* e agir como processo de *load balancing*. Numa arquitectura deste género, um *router* é a interface entre a aplicação e todo o cluster e, na perspectiva da aplicação, uma instância de mongos tem um comportamento idêntico a qualquer outra instância de mongod. Um *Shard* pode ser uma simples instância mongod ou um *replica set* e armazena parte da informação de todo o *Sharded Cluster.* No caso deste projecto, os *Shards* são configurados como *replica sets* para que exista redundância de informação em cada um deles. Esta redundância é garantida através de

replicação e o seu modo de funcionamento é abordado no próximo subcapítulo. Os *config servers* são instâncias mongod que armazenam a meta-informação de um *Sharded Cluster.* Utilizam o protocolo *two-phase commit* para assegurar consistência e fiabilidade imediata.

Num *Sharded Cluster* configurado para ambiente de produção é recomendado que existam duas instâncias *routers* (mongos), três instâncias *config server* (mongod) e pelo menos dois *Shards* (mongod) configurados como *replica sets*.

3.8.1.3 Replicação

Num ambiente de produção, é recomendado que cada *Shard* esteja configurado como *replica set.* A replicação numa arquitectura *Sharded Cluster* em MongoDB é feita ao nível dos *replica sets*. Num *replica set*, ilustrado na Figura 12, uma das instâncias mongod assume o papel de *primary* (primária). Esta instância primária recebe todas as operações de escrita e todas as outras instâncias recebem a nova informação da instância primária para que todas elas tenham o mesmo conjunto de dados. Para suportar a replicação, a instância primária guarda todas as alterações num ficheiro de *log*, denominado oplog, que é replicado assincronamente, sacrificando assim a consistência, pelas instâncias secundárias para aplicar as operações aos seus dados. As instâncias secundárias reflectem os dados da instância primária e em caso de falha da instância primária, o *replica set* irá promover uma das instâncias secundárias a primária. Por omissão as leituras são também feitas na instância primária mas é possível aceder a uma instância secundária para leituras.

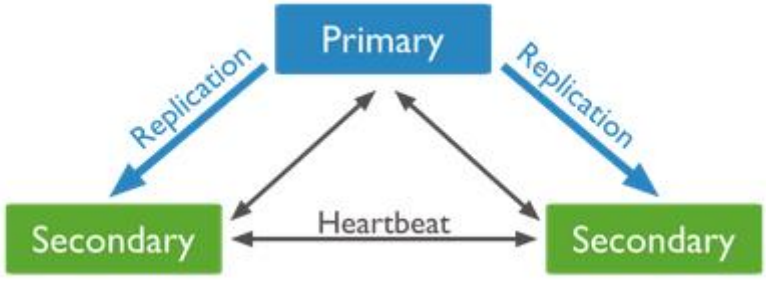


Figura 12 - Replica Set (adaptado de "docs.mongodb.org")

Para que uma instância secundária seja promovida a primária é necessário que ocorra uma eleição entre as instâncias. No caso em que existe um número par de instâncias é necessário adicionar um árbitro para que exista a maioria de votos. Estes árbitros só têm esse propósito e não armazenam informação nem necessitam de *hardware* dedicado. Na Figura 13 é apresentado um diagrama de um *replica set* composto por uma instância primária, uma secundária e um árbitro.

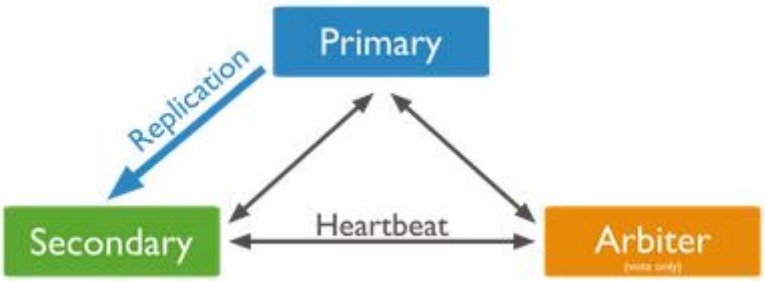


Figura 13 - Eleição de primary com árbitro (adaptado de "docs.mongodb.org")

A eleição de uma nova instância primária acontece quando existe uma falha na comunicação com a instância primária original. Esta falha acontece quando uma instância do *replica set* não consegue comunicar com a instância primária por mais de 10 segundos. Quando isto acontece, o *replica set* inicia o processo de eleição e a primeira instância secundária a obter a maioria dos votos torna-se a nova instância primária.

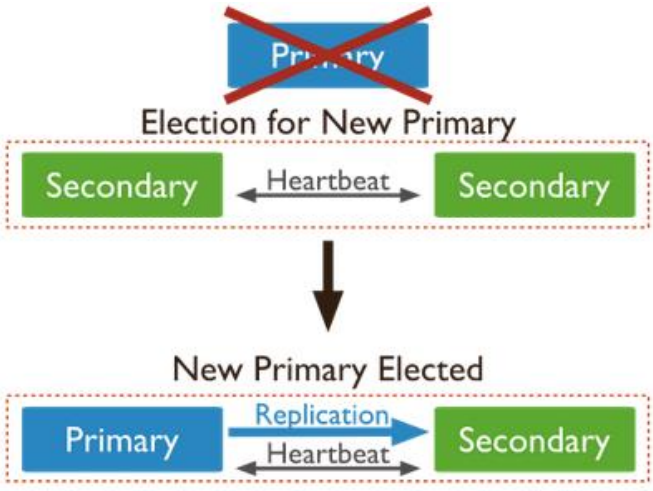


Figura 14 - Processo de eleição de nova instância primária (adaptado de "docs.mongodb.org")

Na Figura 14 é apresentado um diagrama que exemplifica o processo de eleição de uma nova instância primária num *replica set* com três instâncias em que duas são secundárias e a primária está indisponível.

3.8.1.4 Distribuição e escalamento

Na configuração de um *router* de um *Sharded Cluster*, o parâmetro *chunkSize* indica o tamanho que cada *chunk* (representa o tamanho de um conjunto de documentos a ser inserido num *Shard*) deve atingir para que exista uma operação de *split* (distribuição da informação pelos outros *Shards*). Por omissão o valor para esse parâmetro é 64MB.

A distribuição da informação pelos *Shards* é feita ao nível da colecção através de uma *Shard Key*. Uma *Shard Key* é um campo simples, ou composto, indexado que existe em todos os documentos da colecção. Para dividir a *Shard Key* em partes pode ser utilizado um esquema de particionamento baseado em gamas de valores ou em *hash.* No primeiro, ilustrado na Figura 15, a informação é dividida entre gamas determinadas pelos valores da *Shard Key*.

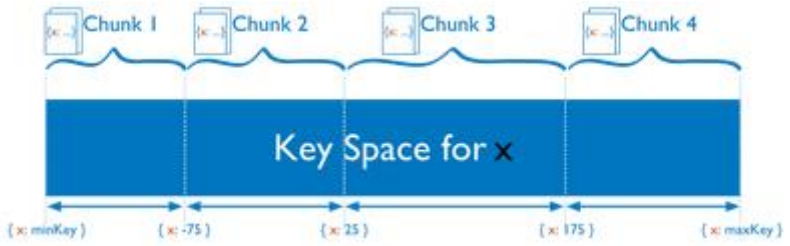


Figura 15 - Sharding por gama de valores (adaptado de "docs.mongodb.org")

Num esquema de particionamento baseado em gama de valores, é provável que os documentos com valores de *Shard key* próximos fiquem no mesmo *Shard*, no entanto tem como desvantagens a possibilidade de existirem *Shards* com mais informação que outros. Esta desvantagem pode ter como consequência o facto de apenas um pequeno número de *Shards* ter que conter toda ou a maior parte da informação porque está toda

na mesma gama de valores. Isto torna-se um problema para escalar posteriormente a arquitectura.

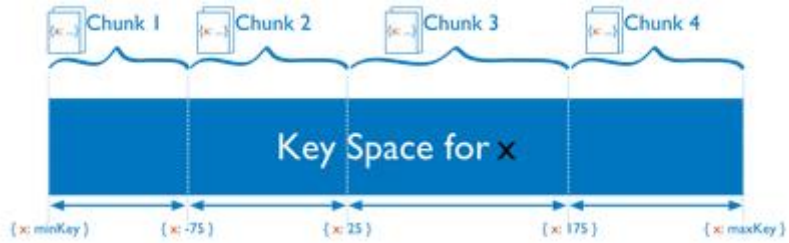


Figura 16 - Sharding baseado em hash (adaptado de "docs.mongodb.org")

Um esquema de particionamento baseado em *hash*, ilustrado na Figura 16, suporta interrogações de gamas de valores baseadas na *Shard Key* de forma mais eficiente pois o *router* consegue facilmente determinar em que *Shards* está contida essa informação e interroga unicamente esses *Shards.* Ao contrário do particionamento baseado em gama de valores, este particionamento garante uma distribuição equitativa de informação sobre todos os *Shards*.

Quando um novo *Shard* é inserido na arquitectura, a informação é redistribuída automaticamente tendo já em conta o novo *Shard.* A desvantagem desta redistribuição é que se existir uma grande quantidade de informação poderá demorar algum tempo e criar um grande volume de tráfego de dados na rede, no entanto através da utilização de *Shards* virtuais é possível atenuar este problema. *Shards* virtuais são *Shards* configurados no mesmo nó físico, o que significa que toda a redistribuição de informação é local ao nó.

Para o cenário concreto da aplicação *Blipper*, será utilizado um esquema de particionamento baseado em *hash* de forma a distribuir a informação da forma mais equitativa possível. Para *Shard Key* deveria ser utilizado o campo geográfico *blip* de forma a que a distribuição fosse efectuada por coordenadas, no entanto MongoDB não fornece a possibilidade de utilizar um campo geográfico como critério para *Shard Key*. Em alternativa será utilizado o campo referente ao código do país de forma a distribuir a informação baseada no país onde o ponto foi criado.

3.8.1.5 Indexação

Todas as colecções em MongoDB têm obrigatoriamente um índice no campo _id. Este campo existe por omissão e caso não seja especificado na inserção de nova informação, o driver da instância mongod irá criá-lo com o valor do ObjectId inserido. O ObjectId é um valor único (12 bytes) gerado pelo MongoDB. Para além desse índice criado por omissão, o MongoDB permite a criação de índices simples, sobre um campo, ou compostos, sobre vários campos, e que podem ser do tipo:

- *Multikey*: permitem a indexação de conteúdo armazenado em arrays. É criada uma entrada no índice por cada elemento do array;
- *Geospatial*: de forma a suportar interrogações eficientes envolvendo coordenadas geoespaciais são disponibilizados dois tipos de índices geoespaciais. Índices 2d que utilizam geometria plana e índices esféricos 2d que utilizam geometria esférica;
- *Text*: índices que suportam interrogações sobre conteúdos de texto (*string*) numa colecção;
- *Hashed*: este tipo de índice é utilizado em esquemas de particionamento baseado em *hash* e indexa o *hash* do valor de um campo.

De acordo com a documentação de MongoDB, [18], não é possível a utilização de um índice geoespacial como *Shard Key.*

3.8.2 Microsoft SQL Server 2012

O Microsoft SQL Server é um **SGBD** que segue o modelo relacional e que se encontra em permanente evolução. Possui um grande conjunto de funcionalidades e permite uma configuração extensiva de forma a poder ser optimizado para domínios de aplicação diferentes.

3.8.2.1 Arquitectura

O termo *Sharding* é um termo relativamente recente tendo sido introduzido pelas tecnologias NoSQL, no entanto é possível utilizar o SQL Server da mesma forma. Na definição da arquitectura a implementar para SQL Server tentou-se que esta fosse o mais semelhante possível à arquitectura de *Sharded Cluster* de MongoDB de forma a eliminar variáveis relacionadas com a arquitectura nos testes.

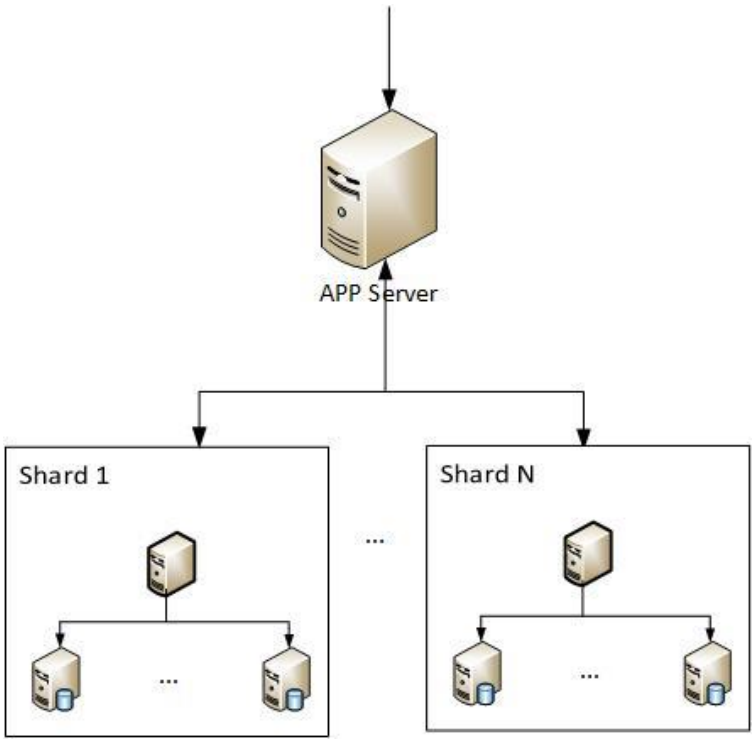


Figura 17 - Arquitectura SQL Server

A Figura 17 ilustra a arquitectura proposta para SQL Server. Ao contrário de MongoDB não existe uma instância SQL Server dedicada a fazer o encaminhamento de pedidos pois isso provoca mau desempenho geral da arquitectura. Assim o encaminhamento de pedidos para o *Shard* correcto é feito pela camada aplicacional que poderá consistir em vários servidores para maximizar a performance. Ao efectuar o encaminhamento de pedidos através da camada aplicacional existem ganhos na performance da arquitectura. Cada *Shard* é composto por um *load balancer* e várias instâncias SQL Server 2012.

3.8.2.2 Replicação

A replicação, tal como em qualquer **SGBD**, é essencial para introduzir redundância de dados de forma a fornecer tolerância a falhas por parte da arquitectura. Assim, cada *Shard* é composto por várias instâncias SQL Server 2012 que estão configuradas com um esquema de replicação, no entanto uma vez que os pedidos estão a ser encaminhados por um *load balancer* do *Shard* de forma a distribuir a capacidade de processamento, é fundamental que todas as instâncias permitam escritas e leituras. Para garantir que todos os nós do *Shard* possuem a capacidade de escrita e leitura o esquema de replicação deverá ser um esquema P2P.

3.8.2.3 Distribuição e escalamento

Uma vez que a distribuição da informação pelos *Shards* não é feita pela camada de dados mas sim pela camada aplicacional, é a mesma que fica encarregue de efectuar os cálculos para determinar o *Shard* correcto para escritas e leituras.

$$Shard = Abs(Hash(countryCode)\ \%\ Nr\_Shards)$$

Equação 1 - Cálculo do Shard

O cálculo do *Shard* acedido, para escrita ou leitura, é apresentado na Equação 1. O *hash* do valor campo do código do país é gerado pela aplicação e posteriormente é obtido o valor absoluto do resto da divisão do *hash* pelo número total de *Shards*. O resultado deste cálculo é um valor inteiro que varia de [0,N-1] sendo N o número total de *Shards* que existem na arquitectura.

A escalabilidade duma arquitectura deste tipo é mais complexa do que em MongoDB. Caso seja necessário escalar o número de máquinas da camada aplicacional apenas é necessário replicar o código para as novas máquinas mas o mesmo não se aplica na adição de um novo *Shard* à arquitectura. Quando um novo *Shard* é adicionado à

arquitectura, pode ser necessário reformular *stored procedures*, *views*, *triggers* e *linked servers* e, pelo facto de a aplicação estar a efectuar cálculos para poder determinar o *Shard* que será acedido, é necessário que seja também actualizada para que reflicta o novo número de *Shards* no cálculo.

3.8.2.4 Indexação

O SQL Server suporta os tipos de dados geoespaciais Geometry [19] e Geography [20], índices geoespaciais e permite a utilização de funções geoespaciais como STDistance que retorna um conjunto de pontos que estejam a uma distância máxima de um ponto recebido por parâmetro. Os tipos de dados Geometry e Geography são implementados como objectos **CLR** no SQL Server e são utilizados com propósitos diferentes. O primeiro é utilizado para representar informação num sistema de coordenadas Euclidiano e o segundo é utilizado para representar informação num sistema de coordenadas geográfico em que cada ponto é composto por latitude e longitude. Ao contrário de MongoDB em que toda a configuração é efectuada nos *routers* (processos mongos), nesta arquitectura proposta para SQL Server é necessário criar, em todas as instâncias, a base de dados e tabelas, os índices geoespaciais e os *stored procedures* o que torna esta arquitectura mais complexa de criar e gerir.

3.8.3 Outras tecnologias

No decorrer do projecto foram investigadas outras soluções NoSQL e do modelo relacional. Para NoSQL foram investigados o **SGBD** *Key-Value Pair* RIAK e o **SGBD** *document store* CouchDB. Para o modelo relacional foi investigado o **SGBD** MySQL Cluster. Para MySQL Cluster foram iniciados testes, no entanto devido aos *drivers Java* ClusterJ e ClusterJPA não darem suporte à inserção de coordenadas geográficas e o **SGBD** não suportar índices geoespaciais acabou por se abandonar esta tecnologia. Quanto às tecnologias NoSQL investigadas, a tecnologia RIAK parece bastante promissora no que toca a escalabilidade e disponibilidade, no entanto foi verificado que

para o cenário do trabalho não seria apropriado pois também não suporta índices e pesquisas geoespaciais. A tecnologia CouchDB é, tal como MongoDB, orientada a documentos e tem características semelhantes a MongoDB, no entanto no decorrer da investigação entre estas duas tecnologias deu-se prioridade à tecnologia MongoDB uma vez que a mesma já é utilizada por uma rede social orientada à partilha de informação georreferenciada.

3.9 Shard Key

O critério utilizado para distribuição da informação é o campo do código do país. Este campo, embora válido para efectuar os testes à arquitectura, torna-se desadequado num cenário real pois além de não permitir uma distribuição uniforme levanta problemas em pontos fronteira entre países. Como o cálculo do *Shard* onde a informação será escrita/lida depende deste campo, isto significa que no caso de uma interrogação sobre um ponto de Portugal que fique muito próximo de Espanha apenas irão ser retornados os pontos pertencentes a Portugal e o mesmo acontece quando o ponto pertence a Espanha e está muito próximo de Portugal em que apenas serão retornados os pontos de Espanha.

4 Implementação

Neste capítulo é detalhada a implementação das arquitecturas em MongoDB e em SQL Server 2012.

Ambas as arquitecturas foram configuradas numa infra-estrutura *cloud* composta por quatro máquinas dedicadas em que apenas três delas foram utilizadas para os *Shards*. Na outra máquina foram apenas configurados os *config servers* e o processo de encaminhamento de MongoDB.

4.1 Configuração da arquitectura Sharded Cluster MongoDB

Toda a configuração de MongoDB é efectuada através de linha de comandos e ficheiros de configuração. Foram criadas regras em cada *firewall*, de acordo com [21], em cada máquina para que fosse possível o tráfego de dados. Para implementação foi seguido o processo descrito em [22]. Todos os ficheiros de configuração são especificados em YAML.

4.1.1 Config Servers

É necessário que todos os processos dos *config servers* estejam em execução e disponíveis no início da configuração de um *Sharded Cluster*. Por omissão, um *config server* utiliza o porto 27019, no entanto é possível especificar o porto no momento da iniciação de cada *config server*.

```
bin\mongod --configsvr --dbpath data/configdb --port 27019
bin\mongod --configsvr --dbpath data/configdb --port 27020
bin\mongod --configsvr --dbpath data/configdb --port 27021
```

Listagem 1 - Iniciação dos três config servers MongoDB

Na Listagem 1 são apresentadas as três instruções, uma por cada processo, de iniciação dos três *config servers*.

4.1.2 Processo de encaminhamento mongos

É neste processo que são efectuadas as configurações do *Sharded Cluster*.

```
bin\mongos --configdb
"151.236.47.144:27019","151.236.47.144:27020","151.236.47.144:27021" --
config C:\mongodb1-mongos\mongodb.conf
```

Listagem 2 - Iniciação do mongos

Na Listagem 2 é apresentada instrução de iniciação do processo mongos. Os *endpoints* fornecidos nesta instrução correspondem aos três processos dos *config servers.*

```
systemLog:
   destination: file
   path: "C:/mongodb1-mongos/log/mongodb.log"
   logAppend: true
sharding:
   chunkSize: 64
```

Listagem 3 - Ficheiro inicial de configuração mongos

O conteúdo do ficheiro mongodb.conf utilizado como parâmetro na iniciação do mongos é apresentado na Listagem 3. Estas configurações apenas especificam a escrita da actividade do mongos num ficheiro de *log* pelo que não são obrigatórias na configuração do *Sharded Cluster*.

4.1.3 Shards

A iniciação das duas instâncias de cada *Shard* é também efectuada através de linha de comandos.

```
bin\mongod --config C:\1-mongodb-Shard1\mongodb.conf
bin\mongod --config C:\2-mongodb-Shard1\mongodb.conf
```

Listagem 4 - Iniciação das duas instâncias do Shard 1

```
bin\mongod --config C:\1-mongodb-Shard2\mongodb.conf
bin\mongod --config C:\2-mongodb-Shard2\mongodb.conf
```

Listagem 5 - Iniciação das duas instâncias do Shard 2

Na Listagem 4 e Listagem 5 são apresentadas as instruções de iniciação das duas instâncias dos *Shards*. Tal como na iniciação do mongos, foram especificados ficheiros de configuração.

```
systemLog:
   destination: file
   path: "C:/1-mongodb-Shard1/log/mongodb.log"
   logAppend: true
net:
   bindIp: 151.236.45.222
   port: 27017
replication:
   replSetName: "rsBlipper1"
   secondaryIndexPrefetch: "all"
   oplogSizeMB: 2048
storage:
   dbPath: "C:/1-mongodb-Shard1/data/db"
```

Listagem 6 - Ficheiro inicial de configuração do Shard 1

A diferença nos ficheiros de configuração dos *Shards* são os *endpoints* e os nomes dos *replica sets* especificados. Na Listagem 6 é apresentado o conteúdo do ficheiro de configuração da primeira instância do *Shard 1*. Para as duas instâncias do primeiro *Shard* apenas alteram os valores para os parâmetros *port* e *dbPath* tendo os valores 27017 e "C:/1-mongodb-Shard1/data/db" para a primeira instância e 27018 e "C:/2-mongodb-Shard1/data/db" para a segunda. O conteúdo dos ficheiros de configuração apenas difere no endereço ip e porto de cada nó (parâmetros bindIp e port, respectivamente), no nome do *replica set* (parâmetro replSetName), caminho para a pasta que contém a base de dados (parâmetro dbPath) e o caminho para o ficheiro de log.

Cada *Shard* foi configurado como um *replica set* e a primeira instância (porto 27017) de cada *replica set* foi configurada como sendo a primária. Para que uma instância seja configurada como primária, é necessário que a configuração do *replica set* seja feita sobre a mesma. Assim, em cada primeira instância foram executadas as instruções:

- rs.initiate(): inicia o *replica set*;
- rs.add(<ip>:<port>): adiciona um novo nó ao *replica set*. Na primeira instância do *Shard* 1 foi executado "rs.add("151.236.45.222:27018"), na primeira instância do *Shard* 2 foi executado "rs.add("151.236.45.20:27018")" e na primeira instância do *Shard* 3 foi executado "rs.add("85.234.141.86:27018")".

A instrução rs.status(), executada na primeira instância, permite confirmar que o *replica set* está correctamente configurado, tendo ficado a primeira instância como *PRIMARY* e a segunda como *SECONDARY*.

4.1.4 Configuração do Sharded Cluster

Após iniciados todos os processos e configurados os *replica sets* é necessário configurar o *Sharded Cluster*. Através de uma linha de comandos, efectuou-se a ligação ao processo mongos e foram executadas as instruções:

- use admin: selecciona a base de dados "admin". É nesta base de dados que são efectuadas as configurações sobre mongos;
- db.runCommand({ addShard : "rsBlipper1/151.236.45.222:27017, 151.236.45.222:27018"}): adiciona o *Shard* 1 ao *Sharded Cluster*;
- db.runCommand({ addShard : "rsBlipper2/151.236.45.20:27017, 151.236.45.20:27018"}): adiciona o *Shard* 2 ao *Sharded Cluster;*
- db.runCommand({ addShard : "rsBlipper3/ 85.234.141.86:27017, 85.234.141.86:27018"}): adiciona o *Shard* 3 ao *Sharded Cluster;*

- db.runCommand({ enableSharding : "Blipper" }): activa *Sharding* para a base de dados "*Blipper*";

Neste ponto, a arquitectura do *Sharded Cluster* está configurada, no entanto é necessário ainda definir o campo utilizado como *hashed Shard Key* e o campo utilizado para criar o índice geoespacial. Estes passos são efectuados na aplicação de testes.

4.2 Configuração da arquitectura SQL Server

Uma vez que não existe uma instância SQL Server dedicada ao encaminhamento de pedidos, na configuração desta arquitectura é necessário configurar apenas os *Shards.*

4.2.1 Shards

Para cada instância de cada *Shard* é necessário criar a base de dados, tabelas, índice geoespacial e *stored procedures*.

```
create table Blipper(
    id int not null primary key,
    name varchar(255) not null,
    latitude decimal(10,5) not null,
    longitude decimal(10,5) not null,
    blip geography not null,
    tag varchar(255) not null,
    countryCode varchar(8) not null
)
```

Listagem 7 - Criação da tabela Blipper em cada Shard

Na Listagem 7 é apresentada a definição da tabela *Blipper*. Esta instrução é executada em ambas as instâncias de cada *Shard*.

```
create spatial index idx_Blip on Blipper(blip) USING GEOGRAPHY_GRID
WITH
(
GRIDS=(LEVEL_1 = HIGH,LEVEL_2 = HIGH,LEVEL_3 = HIGH,LEVEL_4 = HIGH)
, CELLS_PER_OBJECT = 64
, PAD_INDEX = OFF
, SORT_IN_TEMPDB = OFF
, DROP_EXISTING = OFF
, ALLOW_ROW_LOCKS = ON
, ALLOW_PAGE_LOCKS = ON
)
```

Listagem 8 - Definição de índice geoespacial sobre a coluna blip

Na Listagem 8 é apresentada a definição do índice geoespacial. Esta instrução, tal como a anterior, é executada em ambas as instâncias de cada *Shard.*

```
CREATE PROCEDURE sp_getNearestBlips @latitude decimal(10,5),
@longitude decimal(10,5), @maxDistance int, @maxResults int
AS
    declare @fromPoint geography =
geography::STGeomFromText('POINT('+CAST(@latitude AS VARCHAR)+'
'+CAST(@longitude AS VARCHAR)+')',4326)
    SELECT TOP(@maxResults)[id]
      ,[name]
      ,[latitude]
      ,[longitude]
      ,[blip]
      ,[tag]
      ,[countryCode]
      ,blip.STDistance(@fromPoint) as distance
    FROM [Blipper_Shard1].[dbo].[Blipper]
    WHERE blip.STDistance(@fromPoint) IS NOT NULL and
blip.STDistance(@fromPoint) <= @maxDistance
GO
```

Listagem 9 - Stored Procedure utilizado para interrogações no Shard 1

Na Listagem 9 é apresentada a definição do *stored procedure* utilizado para interrogações ao *Shard* 1. A instrução para definição do *stored procedure* nos *Shards* 2 e 3 têm apenas como diferença o nome da base de dados. Caso o nome das bases de dados fosse o mesmo, esta instrução seria igual para todos os *Shards.* Este *stored procedure* recebe como parâmetros a latitude, longitude, a distância do ponto a pesquisar e o número máximo de registos a retornar.

4.2.2 Problemas

No início dos testes desta arquitectura foi utilizada uma instância SQL Server para o encaminhamento de pedidos. Esse encaminhamento era efectuado através da utilização de *linked servers* e *stored procedures* para escrita e leitura. Um acesso a qualquer um dos *Shards* implicava uma interrogação com a opção OPENQUERY, [23], o que tornou as interrogações muito complexas. Para agravar este facto, a utilização de uma coluna do tipo CLR Geography nestas interrogações obrigava à conversão do objecto Geography em texto no *Shard* para que fosse novamente convertido em Geography quando fosse recebido na resposta. Como solução a este problema optou-se pela criação de *stored procedures* para escrita e leitura em cada um dos *Shards* e a instância de encaminhamento apenas executava um ou outro dependendo do *Shard* que teria que utilizar e do tipo de operação. Esta solução verificou-se funcional, no entanto o seu desempenho não era aceitável. Assim, concluiu-se que o facto de utilizar uma instância cujo objectivo era encaminhar os pedidos para o *Shard* correcto não teria vantagens e optou-se por deixar essa responsabilidade na camada aplicacional o que melhorou drasticamente o desempenho da arquitectura. No entanto, como referido no subcapítulo 3.8.2.3, esta alteração tem como custo principal uma maior complexidade de escalamento.

4.3 Aplicação de testes

Para que seja possível testar as possíveis arquitecturas **SGBD** é necessário utilizar uma metodologia de testes igual para as mesmas. Assim, a metodologia de testes consiste em efectuar um elevado número de escritas seguido de um elevado número de leituras.

Para efectuar estes testes, foi desenvolvida uma aplicação que para além de efectuar as escritas e leituras também contabiliza o tempo total de cada uma destas operações. Estas operações são efectuadas de forma paralela com recurso a várias *Tasks* (tarefas). O número de tarefas é definido manualmente e o critério para decisão do número de tarefas foi o número de *Shards*, ou seja, para cada teste de escrita ou leitura são criadas

n tarefas. Desta forma, e principalmente no caso do SQL Server, garante-se a possibilidade de acessos paralelos ao **SGBD**.

Assim, o desempenho dos **SGBD**s é testada quanto a:

- Desempenho de inserção: O tempo demorado na inserção de todos os registos gerados;
- Desempenho de leitura: O tempo de leitura de um determinado número de registos através de uma interrogação geoespacial.

4.3.1 Funcionamento

No início da execução da aplicação é lido, de um ficheiro de texto separado por tabulações, um determinado número de registos por cada *Shard*, ou seja, se existirem três *Shards* será lido três vezes esse número de registos. A informação lida para cada *Shard* é diferente de forma a que sejam sempre efectuadas novas escritas e para aumentar o volume de informação na base de dados. Cada leitura é guardada em memória num objecto que é posteriormente utilizado por cada tarefa para escrita.

Após lido o ficheiro de texto e iniciados os objectos com a informação para escrita, são criadas n tarefas que irão proceder à escrita da sua informação. Para leitura são também criadas n tarefas que efectuam a interrogação geoespacial e processam a informação retornada. Uma vez que se está a testar o desempenho das arquitecturas, tendo por base o cenário de uma aplicação que suporta dados georreferenciados, é essencial que se monitorize o tempo total da execução de todas as operações. Assim, o tempo total da execução de todas as tarefas para escrita e para leitura é guardado e é posteriormente utilizado para cálculos de número médio de registos inseridos/lidos por segundo.

Após concluídas as operações, as estatísticas são geradas com base nas fórmulas apresentadas de seguida.

$$Total\ de\ registos\ inseridos = Nr\_Registos\_Inseridos\_Por\_Tarefa * Nr\_Tarefas$$

Equação 2 - Número total de registos inseridos

Na Equação 2 é apresentado o cálculo do número total de registos inseridos. Tendo em conta que existem n tarefas, o número total de registos inseridos é a soma do número de registos inseridos por cada tarefa.

$$Total\ de\ registos\ lidos = Nr\_Interrogações * Nr\_Maximo\_De\_Pontos\ * Nr\_Tarefas$$

Equação 3 - Número total de registos lidos

O cálculo do número total de registos lidos é apresentado na Equação 3. Cada interrogação geoespacial obtém um determinado número máximo de pontos. Assim, o número total de registos lidos corresponde ao produto do número de interrogações efectuado pelo número máximo de pontos e pelo número de tarefas.

$$Escritas\ (Média\ por\ segundo) = \frac{Total\ de\ registos\ inseridos}{Tempo\ total\ de\ execução}$$

Equação 4 - Escritas (Média por segundo)

O cálculo do número médio de escritas efectuadas por segundo, apresentado na Equação 4, é obtido através da divisão do número total de registos inseridos pelo tempo total de execução (segundos) das tarefas.

$$Leituras\ (Média\ por\ segundo) = \frac{Total\ de\ registos\ lidos}{Tempo\ total\ de\ execução}$$

Equação 5 - Leituras (Média por segundo)

Na Equação 5 é apresentado o cálculo do número médio de leituras por segundo. Este valor é obtido através da divisão do número total de registos lido pelo tempo total de execução (segundos) das tarefas.

4.3.2 Leitura dos dados

Para gerar os dados utilizados para preencher a base de dados foi criada a classe estática RecordsGenerator cuja responsabilidade é a de ler os mesmos de um ficheiro de texto, disponibilizado em [24]. Este ficheiro contém informação de cerca de 8.5 milhões de pontos, distribuídos pelo mundo, e respectiva meta-informação e está formatado como **CSV**.

Como referido no subcapítulo anterior, a informação lida do ficheiro é diferente para cada *Shard* de forma a que seja sempre escrita informação em todos os *Shards*.

O ficheiro de dados contém toda a informação agrupada por código de país. Uma vez que o cálculo do *Shard* acedido pelo SQL Server, para escrita ou leitura, depende directamente do resultado da função de *hash* do valor do código do país do ponto, como apresentado anteriormente pela Equação 1, a leitura sequencial de todos os pontos para escrita em dois *Shards* originava o mesmo valor de *hash* o que, num cenário com dois *Shards*, tinha como consequência que todos os pontos fossem inseridos no segundo *Shard* causando assim resultados irreais para leituras e escritas. Assim, a leitura da informação passou a ser efectuada através de blocos, parametrizados, em que existe sempre um intervalo de registos ignorados entre os mesmos.

$$Nr\_Registos\_Ignorados < \text{Nr\_Registos\_A\_Ler} * (\text{Nr\_Shard} * 3)$$

Equação 6 - Número de registos ignorados

Com o cálculo apresentado na Equação 6, são ignorados blocos na leitura da informação.

Recorrendo a um ciclo que ignora um determinado número de registos, através do cálculo apresentado na Equação 6, a informação é lida em blocos.

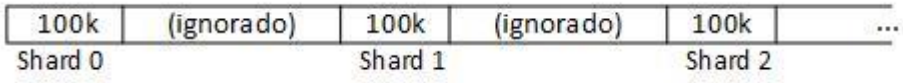


Figura 18 - Leitura de blocos de dados

Na Figura 18 é apresentado um cenário para leitura de 100 mil registos para três *Shards* sendo assim efectuadas as seguintes leituras:

1. Para o *Shard* 0 são lidos os registos [0;100000[;
2. Para o *Shard* 1 são lidos os registos [300000;400000[;
3. Para o *Shard* 2 são lidos os registos [600000;700000[.

Assim, garante-se que existe um maior número de valores distintos de códigos de países o que origina a uma melhor distribuição dos registos pelos *Shards*.

Na leitura da informação são guardados todos os códigos de países distintos lidos do ficheiro com o objectivo de serem utilizados posteriormente no processo de determinar o *Shard* a interrogar. Para tal, criou-se a classe estática RecordsInfo que contém um dicionário *key-value* em que a *Key* é o código do país e o *Value* um ponto pertencente a esse país. Este dicionário é actualizado por cada registo lido e, como dito anteriormente, é utilizado para determinar o *Shard* nas interrogações a cada um dos **SGBD**s.

4.3.3 Implementação

A aplicação de testes foi desenvolvida de forma a automatizar todo o processo de testes. Assim, são executadas n iterações de testes e escritos todos os resultados num ficheiro de texto. Através do ficheiro App.config da aplicação é possível redefinir a maior parte dos parâmetros utilizados nos testes entre iterações.

4.3.3.1 Classe DbTesterMain

A classe DbTesterMain, apresentada na Listagem 10, é responsável por criar instâncias, especificadas no ficheiro App.config, para testes a cada um dos **SGBD**s e gerar os dados para inserção recorrendo ao método GetRecordsBlipper() da classe estática RecordsGenerator, cujo funcionamento foi detalhado no subcapítulo 4.3.2.

```
class DbTesterMain
{
    public DbTesterMain(Type benchmarkType, TestMethod operacao, ResultsWriter
rw){...}
    private IEnumerable<KeyValuePair<long, RecordBlipper>>
GetRecordsBlipper(int idThread){...}
    private Task[] DoWriteBlipper(){...}
    private Task[] DoReadBlipper(){...}
    private IEnumerable<KeyValuePair<long, Record>> GetRecords(int
idThread){...}
    private Task[] DoRead(){...}
    private void Read(){...}
    private Task[] DoWrite(){...}
    public void Write(){...}
}
```

Listagem 10 - Classe DbTesterMain

Nos métodos DoWrite() e DoRead() são criadas n tarefas, uma por cada *Shard*, que irão executar escritas ou leituras, respectivamente, no **SGBD** instanciado.

O ficheiro App.config possui duas secções de configuração utilizadas pela aplicação: Testes e *appsettings*. Na secção Testes são parametrizados os **SGBD**s a testar bem como as respectivas operações e na secção *appsettings* são disponibilizadas configurações para o número de registos a inserir, o número de interrogações bem como o respectivo número máximo de pontos a obter por interrogação, o número de iterações de testes, as *connection strings* para cada nó da arquitectura e configurações relativas ao ficheiro de texto para leitura dos dados. As configurações são obtidas do ficheiro App.config através da classe estática AppSettingsParser.

4.3.3.2 Interface IBenchmark

Uma vez que as operações são as mesmas em ambos os **SGBD**s mas os comportamentos para cada um deles é diferente, criou-se a interface IBenchmark, apresentada na Listagem 11, que é implementada pelas classes MongoDB e MSSQLServer.

Esta interface contém quatro métodos principais:

- InitShards(): iniciação dos *Shards* que consiste em apagar toda a informação da tabela de testes, no caso de SQL Server, e na configuração de todo o *Sharded Cluster*, no caso de MongoDB;
- Write()/WriteBlipper(): recebe um conjunto de registos e insere-os na base de dados;
- Read()/ReadBlipper(): efectua interrogações à base de dados;

```
public interface IBenchmark
{
    void InitShards();
    void Read(TestResults ti);
    void ReadBlipper(TestResults ti);
    void Write(IEnumerable<KeyValuePair<long, Record>> recordsToInsert,
TestResults ti);
    void WriteBlipper(IEnumerable<KeyValuePair<long, RecordBlipper>>
recordsToInsert, TestResults ti);}
```

Listagem 11 - Interface IBenchmark

No início de cada teste de escrita é invocado o método InitShards() o que irá fazer com que todos os dados sejam apagados e posteriormente configurada a arquitectura.

Os métodos ReadBlipper() e WriteBlipper() são utilizados para os testes relacionados com o *Blipper*, sendo que no primeiro são efectuadas interrogações espaciais.

```
class TestResults
{
    public int IdShard { get; set; }
    //Resultados
    public int NumOperacoes { get; set; } //Numero de operacoes efectuadas
}
```

Listagem 12 - Classe TestResults

Na Listagem 12 é apresentada a classe TestResults. Esta classe é utilizada por cada tarefa para guardar os resultados das operações.

4.3.3.3 Classe MongoDB

A configuração da arquitectura *Sharded Cluster* é feita sobre o processo de encaminhamento mongos. Assim, o método InitShards() define a *Shard Key* e o índice geoespacial sobre a colecção.

```
var adminDB = server.GetDatabase("admin");

database.GetCollection<Row>(name).EnsureIndex(IndexKeys.GeoSpatial("Record
.Blip"));
var shardCollection = new CommandDocument { { "shardCollection", _dbName +
"." + name }, { "key", new BsonDocument("Record.CountryCode", "hashed") }
};

//Define como shard key Record.CountryCode
adminDB.RunCommand(shardCollection);
```

Listagem 13 - Método InitShards()

Na Listagem 13 é apresentada uma parte do código utilizado no método InitShards() em que é definido o índice geoespacial sobre o campo Blip e activado o *Sharding* ao nível da colecção definindo como *Shard Key* o campo CountryCode.

```
var numITeracoes = 0;
while (numITeracoes < AppSettingsParser.NUM_ITERACOES)
{
    var pontos = RecordsInfo.GetBlips();
    var randInd = RandomProvider.GetThreadRandom().Next(0, pontos.Length);
    var ponto = pontos[randInd];
    var collection =
database.GetCollection<RowBlipper>(AppSettingsParser.TABLENAME);
    var longitude = ponto[0];
    var latitude = ponto[1];
    var options = GeoNearOptions.SetMaxDistance(1).SetSpherical(true);
    GeoNearResult result = collection.GeoNear(Query.Null, latitude, longitude,
AppSettingsParser.NUM_RECORDS_A_LER, options);
    foreach (var elem in result.Hits)
    {
      var record = (RecordBlipper)((RowBlipper)elem.Document).Record;
      ti.NumOperacoes += 1;
    }
    ++numITeracoes;
}
```

Listagem 14 - Método ReadBlipper()

Na Listagem 14 é apresentada parte da implementação do método ReadBlipper(). Neste método são executadas as interrogações geoespaciais ao **SGBD** MongoDB. Na interrogação é escolhido aleatoriamente um ponto inserido, recorrendo ao método GetBlips() da classe estática RecordsInfo que retorna um array contendo pontos inseridos, e efectuada a interrogação geoespacial sobre o mesmo.

Os parâmetros que indicam o número de interrogações e o número de pontos a obter por interrogação são parametrizados no ficheiro App.config.

4.3.3.4 Classe MSSQLServer

Uma vez que não existe uma instância SQL Server responsável pelo encaminhamento de pedidos para o *Shard*, essa responsabilidade foi delegada na aplicação.

```
foreach (var item in recordsToInsert)
{
    var key = item.Key;
    var rec = item.Value;
    var query =
    string.Format(
"INSERT INTO {0} VALUES(@id,@name,@latitude,@longitude,@blip,@tag,@countryCode)",
    AppSettingsParser.TABLENAME);
    using (var conn = new SqlConnection(GetBlipperConnectionString(rec.CountryCode)))
    {
      conn.Open();
      using (var cmd = new SqlCommand(query, conn))
      {
         cmd.Parameters.Add("@id", SqlDbType.Int, 0).Value = key;
         cmd.Parameters.Add("@name", SqlDbType.VarChar, 255).Value = rec.Name;
         cmd.Parameters.Add("@latitude", SqlDbType.Decimal, 10).Value = rec.Latitude;
         cmd.Parameters.Add("@longitude", SqlDbType.Decimal, 10).Value = rec.Longitude;
         var blip = new SqlParameter("@blip", SqlDbType.Udt);
         blip.Value =
         SqlGeometry.STPointFromText(
            new SqlChars(
            ("POINT(" + rec.Latitude.ToString(CultureInfo.InvariantCulture) + " "
+rec.Longitude.ToString(CultureInfo.InvariantCulture) + ")").ToCharArray()),
            4326);
         blip.UdtTypeName = "Geography";
         cmd.Parameters.Add(blip);
         cmd.Parameters.Add("@tag", SqlDbType.VarChar, 255).Value = rec.Tag;
         cmd.Parameters.Add("@countryCode", SqlDbType.VarChar, 255).Value = rec.CountryCode;
         cmd.ExecuteNonQuery();
      }
    }
}
```

Listagem 15 - Método WriteBlipper()

Na Listagem 15 é apresentada uma parte do código do método WriteBlipper(). Recorrendo ao método GetBlipperConnectionString(), que recebe por parâmetro o código do país do ponto a ser inserido e retorna a *connection string* para o *Shard* a ser utilizado, é efectuada a ligação ao *Shard* correcto para escrita da nova informação. No método ReadBlipper() também é utilizado o método GetBlipperConnectionString() com o mesmo objectivo. O cálculo efectuado pelo método GetBlipperConnectionString() para determinar o *Shard* correcto é o apresentado na Equação 1.

Na interrogação à base de dados também é escolhido aleatoriamente um ponto, à semelhança do que foi implementado para MongoDB, e é executado o *stored procedure* sp_getNearestBlips no *Shard* interrogado.

4.3.3.5 Classe RecordBlipper

A classe RecordBlipper, apresentada na Listagem 16, representa um registo a ser inserido na base de dados e contém toda a informação associada a um ponto.

```
public class RecordBlipper
{
    public long Key { get; internal set; }
    public String Name { get; internal set; }
    public Double Latitude { get; internal set; }
    public Double Longitude { get; internal set; }
    public Double[] Blip { get; internal set; }
    public String Tag { get; internal set; }
    public String CountryCode { get; internal set; }

    public RecordBlipper(){}

    public RecordBlipper(long key, String name, Double latitude, Double longitude, String
tag, String countryCode)
    {
       Key = key;
       Name = name;
       Latitude = latitude;
       Longitude = longitude;
       Blip = new Double[2];
       Blip[0] = longitude;
       Blip[1] = latitude;
       Tag = tag;
       CountryCode = countryCode;
    }

    public override string ToString()
    {
       return String.Format("{0};{1};{2};{3};{4};", Name, Latitude, Longitude, CountryCode,
Tag);
}
```

Listagem 16 - Classe RecordBlipper

4.4 Protótipo da aplicação Blipper

Recorrendo à tecnologia ASP.NET MVC e à *framework* Google Maps, [25], foi desenvolvido um protótipo da aplicação *Blipper*. Não foram implementadas todas as funcionalidades da aplicação mas apenas as essenciais para que pudesse ser validado o seu funcionamento.

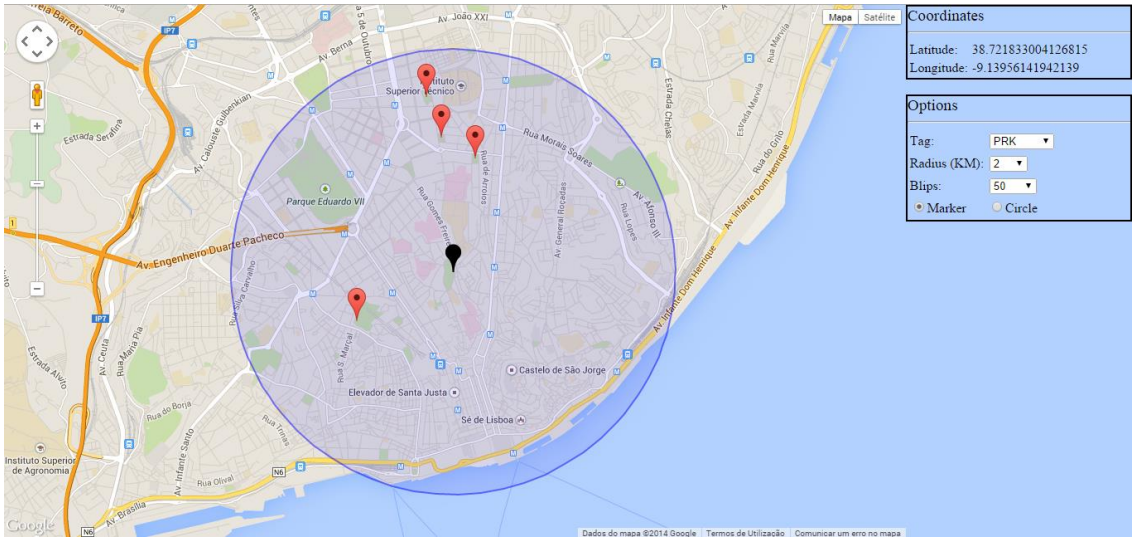


Figura 19 - Ecrã principal da aplicação Blipper

Na Figura 19 é ilustrado o ecrã principal da aplicação *Blipper*. A solução final seria desenvolvida tendo em conta dispositivos móveis o que significa que o ecrã principal da aplicação poderia ser diferente do apresentado.

O círculo azul, apresentado na figura, simboliza o alcance máximo da aplicação para as interrogações geoespaciais sendo o marcador preto o centro do círculo.

Os marcadores apresentados são actualizados sempre que é escolhida uma nova localização o que é conseguido arrastando o mapa com o cursor.

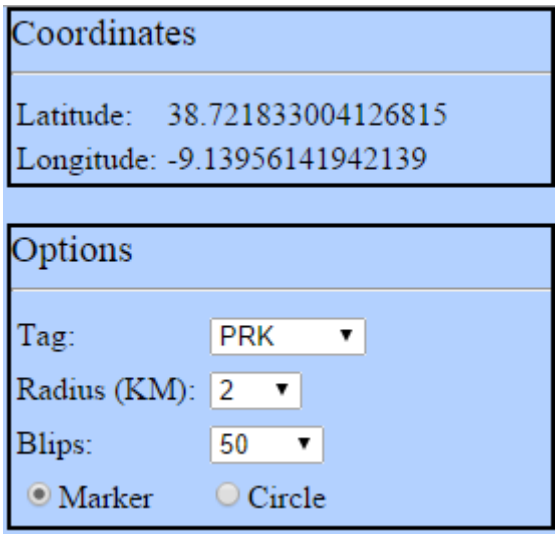


Figura 20 - Opções da aplicação Blipper

A Figura 20 ilustra as opções da aplicação. Estas opções são disponibilizadas no canto superior do lado direito. Na primeira secção são apresentadas as coordenadas do ponto central do círculo. Na secção abaixo é possível alterar a *tag* dos pontos que são visualizados, o alcance máximo das interrogações geoespaciais, o número máximo de pontos a mostrar e o tipo de marcador utilizado no mapa. Além destas opções, é possível adicionar um novo ponto de interesse através do duplo clique do rato no mapa o que causará a apresentação de uma nova secção onde poderá ser indicado a *tag* do novo ponto bem como o nome do mesmo.

Os pontos de interesse no mapa estão assinalados com os marcadores pré-definidos do Google maps, no entanto foi implementada a alternativa de serem visualizados como círculos com baixa opacidade para permitir visualizar no mapa a característica da sobreposição de pontos apresentado no subcapítulo 1.1.

```
public static class BlipperAL
{
    public static IEnumerable<Blip> GetNearestBlips(double latitude, double
longitude, string tag, int numBlips, double distance){...}
    public static void InsertBlip(Blip blip){...}
}
```

Listagem 17 - Classe estática BlipperAL

O acesso aos dados é efectuado através da classe estática BlipperAL, apresentada na Listagem 17. Aqui são disponibilizados os métodos GetNearestBlips() e InsertBlip() essenciais ao funcionamento da aplicação *Blipper*. No primeiro é efectuada uma interrogação geoespacial para obter um determinado número de pontos mais próximos do ponto recebido por parâmetro e no segundo é inserido um ponto na base de dados.

4.4.1 Intersecção de pontos de interesse

A Figura 21 ilustra a característica da sobreposição de pontos apresentado no subcapítulo 1.1. Neste cenário existem cerca de 10 pontos sobrepostos. Uma vez que cada círculo tem uma baixa opacidade, a sobreposição dos mesmos faz com que a opacidade da sua intersecção aumente o que se torna evidente pela cor mais escura.

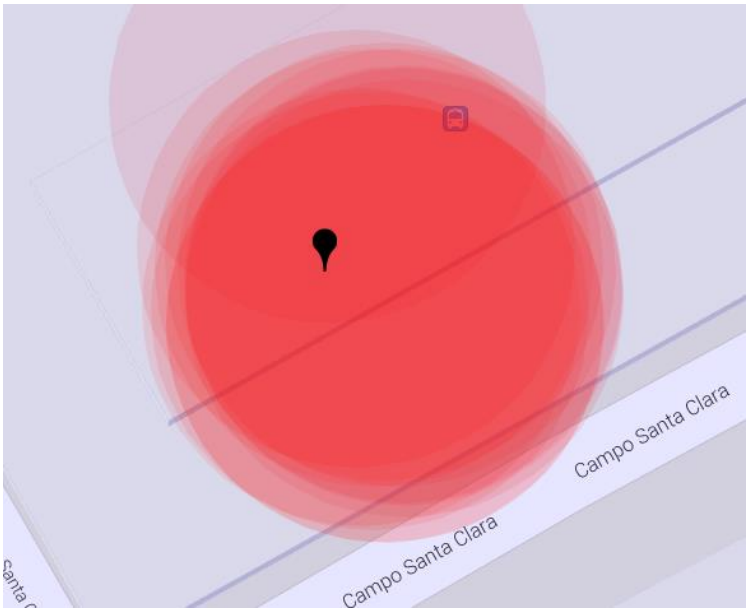


Figura 21 - Sobreposição de pontos de interesse

5 Resultados

Todos os testes foram executados numa infra-estrutura *cloud* composta por quatro máquinas com a mesma configuração de *hardware*. São caracterizadas por um processador Intel Core i3-2100, 4GB de Ram e disco rígido de 500GB Sata 2. Todas as máquinas têm uma ligação de rede de 100 Mbps e Microsoft Windows Server 2008 R2 SP1 como sistema operativo.

Foram configurados três *Shards* para cada **SGBD**. Cada *Shard* é configurado numa máquina e é composto por duas instâncias configuradas com replicação. A quarta máquina foi utilizada apenas para desenvolvimento da aplicação de testes e para execução dos processos de encaminhamento mongos e *config servers*.

As duas instâncias configuradas com replicação não deveriam ser executadas na mesma máquina pois em caso de falha da máquina é perdida toda a informação nas duas instâncias mas foram configuradas desta forma devido a limitações de *hardware*. Idealmente ter-se-ia uma máquina por cada instância de processo o que neste caso seriam onze máquinas para MongoDB (duas para as instâncias de mongos, três para as instâncias de *config servers* e duas por cada *Shard*) e sete no caso da arquitectura proposta para SQL Server 2012 (uma para a camada aplicacional e duas por cada *Shard*).

As versões das aplicações utilizadas são:

- Microsoft SQL Server 2012 v11.0.2100.60
- MongoDB v2.6.0 (driver v1.8.3.9)
- Visual Studio 2012 v11.0.50727.1 TRMREL

5.1 Metodologia de testes

A metodologia de testes consiste em efectuar dez vezes um determinado número de escritas seguido de leituras. Após cada execução dos testes são anotados o tempo total

de execução da operação e a média de operações por segundo. No final dos dez testes são efectuados cálculos tendo por base as equações apresentadas no subcapítulo 4.3.1.

Os testes foram divididos em três fases:

1. Primeira fase: Foram efectuados testes com um nó em que são inseridos 100 mil registos e lidos 1 milhão (mil interrogações em que são lidos mil pontos por cada);

2. Segunda fase: São criadas duas tarefas. Cada uma insere 100 mil registos e lê 1 milhão. No total são inseridos 200 mil registos e lidos 2 milhões;

3. Terceira fase: São criadas três tarefas. Cada uma insere 100 mil registos e lê 1 milhão. No total são inseridos 300 mil registos e lidos 3 milhões.

As estatísticas finais foram calculadas de forma a que fosse possível verificar a evolução de cada uma das arquitecturas configuradas ao ser adicionado um novo nó a cada uma delas.

Uma vez que as máquinas são todas iguais, espera-se que a adição de um nó à arquitectura não traga um benefício de mais de 100% (dobro do desempenho) e a adição de dois nós não traga um benefício de mais de 200% (triplo do desempenho).

Todos os valores apresentados dizem respeito à média dos dez testes de cada um dos parâmetros sendo o tempo total apresentado em segundos.

5.2 Testes com um nó

Nos testes com um nó não existe *Sharding* configurado pois não existe a necessidade de separar a informação. Este teste serviu para verificar o desempenho base das arquitecturas quando são constituídas por apenas um nó e, consequentemente, não existem perdas de desempenho com o processamento necessário para realizar o encaminhamento da informação para o *Shard* correcto.

		Tempo Total (Média)	Op/seg (Média)
MSSQL Server	Escrita	1205,069	83,103
	Leitura	414,827	2452,948
MongoDB	Escrita	4,078	24602,635
	Leitura	150,703	6676,762

Tabela 1 - Resultados para um nó

5.3 Testes com dois nós

Para os testes com dois nós já existe *Sharding* e consecutivamente o processamento inerente ao mesmo. Assim, é esperado que ambas as arquitecturas escalem com a adição de um novo nó à arquitectura.

		Tempo Total (Média)	Op/seg (Média)	Ganho (%)
MSSQL Server	Escrita	1256,273	159,327	91
	Leitura	408,911	4897,652	99
MongoDB (Com MongoS)	Escrita	13,561	15548,915	-37
	Leitura	291,490	6903,187	3

Tabela 2 - Resultados para dois nós

Como esperado, a adição de mais um nó aumenta consideravelmente o desempenho de SQL Server, estando muito próximo dos 100%, no entanto para a arquitectura *Sharded Cluster* de MongoDB o desempenho da leitura não apresenta muito benefício e o da escrita é prejudicada. Uma vez que as arquitecturas foram configuradas numa infra-estrutura *cloud*, foi considerada a hipótese de existir algum problema relacionado com a mesma, por exemplo sobrecarga de rede ou das máquinas. Assim, os testes para a arquitectura *Sharded Cluster* de MongoDB foram repetidos várias vezes e em alturas diferentes do dia sendo os resultados sempre negativos para escrita.

Tendo em conta que o processo de encaminhamento mongos é o responsável por processar e encaminhar toda a informação para o *Shard* correcto, foram efectuados mais testes mas desta vez eliminando este processo. Assim, à semelhança do que é feito

para SQL Server, a camada aplicacional ficou encarregue de encaminhar o pedido para o *Shard* correcto.

		Tempo Total (Média)	Op/seg (Média)	Ganho (%)
MongoDB (Sem MongoS)	Escrita	18,267	11005,075	-56
	Leitura	208,859	9762,330	46

Tabela 3 - Resultados para dois nós - sem mongos

Com estes resultados verifica-se que o desempenho da arquitectura melhorou bastante para leitura, no entanto mantem-se negativo para escrita. Sendo todas as ligações a MongoDB geridas internamente pelo *driver* c# de MongoDB, foi colocada a hipótese de efectuar a gestão das ligações através da aplicação. Assim, a aplicação de testes foi alterada para que fossem guardadas as ligações aos *Shards* em vez de abrir uma nova ligação por pedido e foram novamente executados os testes tendo em conta estas alterações. Assim, para gerir as ligações MongoDB foi criada a classe estática MongoDBCollections cujo objectivo é iniciar uma ligação para cada *Shard*. Cada uma destas ligações é depois utilizada, por cada tarefa, no acesso a cada um dos *Shards*.

		Tempo Total (Média)	Op/seg (Média)	Ganho (%)
MongoDB (Sem MongoS e com ligações reutilizadas)	Escrita	6,349	35697,577	45
	Leitura	190,918	10403,307	55

Tabela 4 - Resultados para dois nós - sem mongos e ligações reutilizadas

Com estas alterações verifica-se uma melhoria no desempenho, no entanto, o encaminhamento é feito pela camada aplicacional e as ligações aos *Shards* são reutilizadas.

5.4 Testes com três nós

Nos resultados destes testes são contabilizados dois ganhos: O ganho 2 para 3 que significa o ganho com a adição de mais um nó ao sistema e o ganho 1 para 3 que significa o ganho total dos três nós.

		Tempo Total (Média)	Op/seg (Média)	Ganho (%)	
				2 para 3	1 para 3
MSSQL Server	Escrita	1218,033	246,820	54	197
	Leitura	461,057	6520,613	33	165
MongoDB (Com MongoS)	Escrita	18,225	17374,802	11	-30
	Leitura	424,202	7091,537	2	6

Tabela 5 - Resultados para três nós

Mais uma vez, verifica-se que SQL Server tem um aumento no desempenho tanto no ganho 2 para 3 como no ganho 1 para 3 sendo que neste último se aproxima do máximo de 200%.

Relativamente à arquitectura *Sharded Cluster* de MongoDB, verifica-se mais uma vez que o ganho para leitura não apresenta muito benefício e para escrita é prejudicado. Assim, e pelo mesmo motivo que nos testes com dois nós, os testes foram repetidos sem o processo de encaminhamento mongos sendo a camada aplicacional responsável pelo encaminhamento de pedidos para o *Shard* correcto.

		Tempo Total (Média)	Op/seg (Média)	Ganho (%)	
				2 para 3	1 para 3
MongoDB (Sem MongoS)	Escrita	26,286	11403,790	3	-54
	Leitura	192,927	15564,584	59	133

Tabela 6 - Resultados para três nós - sem mongos

Tal como nos testes com dois nós, verifica-se que o desempenho da arquitectura melhorou bastante para leituras, no entanto mantem-se negativo para escrita. Assim,

repetiram-se os testes com a reutilização das ligações como efectuado nos testes com dois nós.

		Tempo Total (Média)	Op/seg (Média)	Ganho (%)	
				2 para 3	1 para 3
MongoDB (Sem MongoS e com ligações reutilizadas)	Escrita	8,258	36338,534	1	47
	Leitura	225,322	14003,506	34	109

Tabela 7 - Resultados para três nós - sem mongos e ligações reutilizadas

Mais uma vez, existe um aumento no desempenho ao remover o processo de encaminhamento mongos e ao reutilizar as ligações efectuadas a cada *Shard*.

5.5 Conclusões

Com os testes realizados foi possível verificar que a arquitectura *Sharded Cluster* de MongoDB tem problemas a escalar podendo mesmo piorar o seu desempenho com a adição de mais nós.

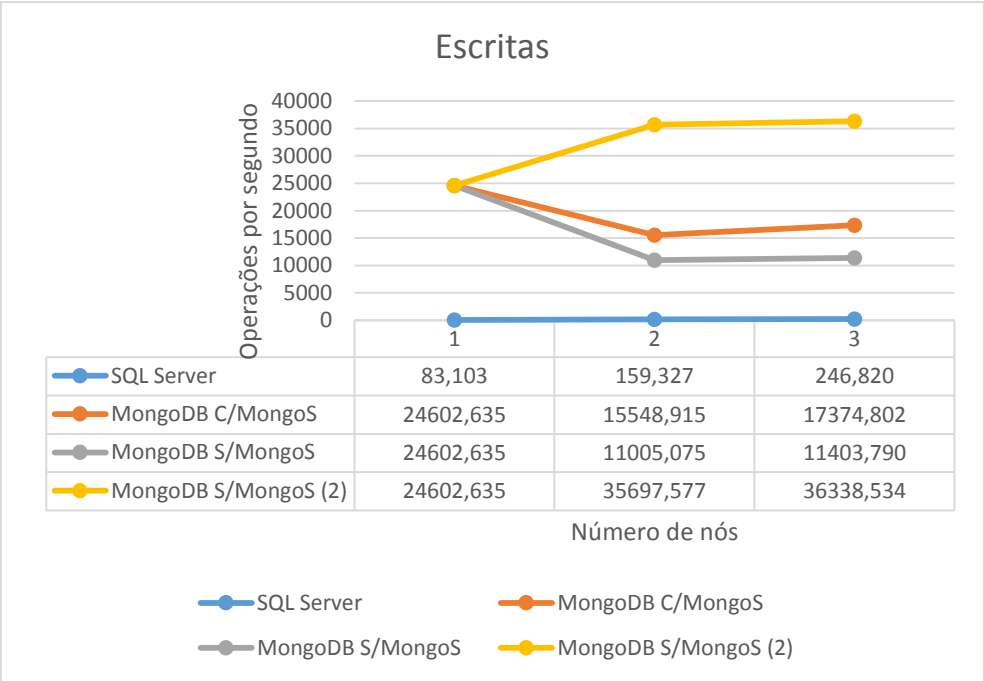


Figura 22 - Resultados dos testes de escrita

Na Figura 22 são ilustrados os resultados dos testes de escrita. Verifica-se que o desempenho da arquitectura *Sharded Cluster* de MongoDB escala negativamente para escritas e que a remoção do processo de encaminhamento e a reutilização das ligações ao MongoDB melhoram drasticamente o desempenho para escritas.

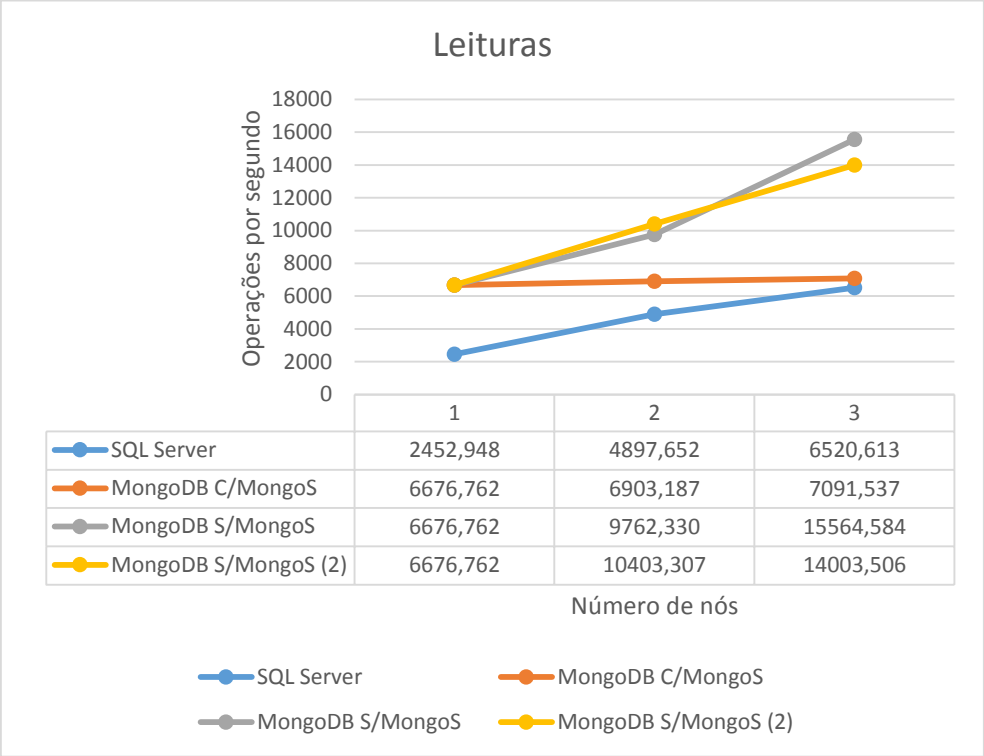


Figura 23 - Resultados dos testes de leitura

Na Figura 23 são ilustrados os resultados dos testes de leitura e é possível concluir que caso a adição de novos nós às arquitecturas continue com ganhos aproximados aos obtidos, é provável que com quatro ou cinco nós a arquitectura SQL Server se torne melhor que a arquitectura *Sharded Cluster* de MongoDB para leituras.

No desenvolvimento do projecto foram encontrados obstáculos que foram ultrapassados com sucesso, no entanto outros não foram ultrapassados sendo o critério para distribuição da informação o maior. O critério utilizado para distribuição da informação, o campo código do país do ponto, não é o mais adequado porque além de não permitir uma distribuição uniforme dos dados levanta o problema dos pontos

fronteira nas interrogações. Para além desses problemas, existe a possibilidade da solução não escalar dentro de cada país, por exemplo se existir apenas um país e muitos pedidos para o mesmo.

Relativamente a ambas as tecnologias, o facto de SQL Server escalar melhor do que MongoDB, neste cenário, não era de todo esperado, no entanto ainda menos esperado era o facto da arquitectura *Sharded Cluster* de MongoDB escalar negativamente para escritas. Contudo verificou-se que é possível melhorar o desempenho removendo o processo de encaminhamento mongos e reutilizando as ligações à custa de mais processamento ao nível aplicacional. Esse processamento inclui a gestão das ligações utilizadas a cada um dos *Shards* e a gestão do encaminhamento através do cálculo do *Shard* correcto. O facto de se terem obtido melhores resultados para leitura e escrita, nos testes de MongoDB, sem utilizar o *connection pooling* do *driver* c# sugere que pode existir um problema com o mesmo. No entanto, por ausência de documentação não é possível comprovar isso. Caso fosse implementada uma solução baseada em MongoDB, poder-se-ia apenas utilizar apenas os *Shards*, configurados como *replica set*, e realizar o encaminhamento, com reutilização de ligações, na camada aplicacional. Para além disso, ao se alterar o critério de distribuição para o ponto em vez do código do país estar-se-ia a aumentar a escalabilidade para ambas as soluções e ao mesmo tempo a resolver-se os problemas da distribuição não ser uniforme e da limitação de escalamento dentro de um país.

Embora para três nós a arquitectura *Sharded Cluster* de MongoDB escale negativamente para escritas, o desempenho da mesma é superior ao desempenho da arquitectura SQL Server. Assim, num cenário com três ou menos nós, a arquitectura *Sharded Cluster* de MongoDB seria a mais adequada para suportar a aplicação *Blipper* por apresentar melhor desempenho tanto para escrita como para leitura, no entanto caso seja necessário um maior escalamento horizontal a médio ou longo prazo, a arquitectura SQL Server pode revelar-se mais eficiente para leituras e provavelmente para escritas à medida que o número de nós aumenta.

5.6 Trabalho futuro

No desenvolvimento do projecto foram identificados os seguintes tópicos para evolução da solução:

- Utilização de outro critério para *Shard Key;*
- Testes com a plataforma Azure SQL;
- Adição de mais nós à arquitectura SQL Server e MongoDB.
- Configuração de uma arquitectura noutra tecnologia NoSQL.

Uma vez que o critério de distribuição de informação possui os problemas apresentados, a utilização de outro critério, de forma a que exista melhor distribuição da informação e que se consiga resolver o problema dos pontos fronteira, iria beneficiar a solução final.

Os testes sobre a tecnologia Azure SQL seriam interessantes não só para testar a arquitectura relativamente aos desempenhos para escritas e leituras geoespaciais mas também para verificar como será a escalabilidade de uma arquitectura desenvolvida nesta tecnologia.

Seria interessante também adicionar mais nós a ambas as arquitecturas desenvolvidas de forma a verificar se ambas continuam a escalar da forma observada nos testes realizados.

Existe um grande número de tecnologias NoSQL disponíveis mas devido ao tempo limitado e aos problemas iniciais encontrados na investigação e desenvolvimento de testes para as várias tecnologias investigadas não foi possível investigar possíveis soluções para outras tecnologias NoSQL, no entanto seria interessante essa investigação e desenvolvimento.

6 Bibliografia

[1] "Distributed database," [Online]. Available: http://en.wikipedia.org/wiki/Distributed_database. [Acedido em 08 03 2014].

[2] A. Guttman, "R-Trees: A dynamic index structure for spatial searching".

[3] "Atomicity," [Online]. Available: http://en.wikipedia.org/wiki/Atomicity_(database_systems). [Acedido em 26 02 2014].

[4] "Consistency," [Online]. Available: http://en.wikipedia.org/wiki/Consistency_(database_systems). [Acedido em 26 02 2014].

[5] "Isolation," [Online]. Available: http://en.wikipedia.org/wiki/Isolation_(database_systems). [Acedido em 26 02 2014].

[6] "List of NoSQL Databases," [Online]. Available: http://nosql-database.org/. [Acedido em 03 03 2014].

[7] "Eric Brewer (scientist)," [Online]. Available: http://en.wikipedia.org/wiki/Eric_Brewer_%28scientist%29. [Acedido em 04 03 2014].

[8] "CAP theorem," [Online]. Available: http://www.julianbrowne.com/article/viewer/brewers-cap-theorem. [Acedido em 04 03 2014].

[9] A. J. Brust, "Windows Azure NoSQL White Paper".

[10] P. J. Sadalage e M. Fowler, "Cap. 2.3 - Column-Family Stores," em *NoSQL Distilled*.

[11] P. J. Sadalage e M. Fowler, "Cap. 1.5. The Emergence of NoSQL," em *NoSQL Distilled*.

[12] P. J. Sadalage e M. Fowler, "Cap. 1.4. Attack of the Clusters," em *NoSQL Distilled*.

[13] P. J. Sadalage e M. Fowler, "Cap. 1.5. The Emergence of NoSQL e Cap. 1.1.2. Concurrency," em *NoSQL Distilled*.

[14] P. J. Sadalage e M. Fowler, "Cap. 1.2. Impedance Mismatch," em *NoSQL Distilled*.

[15] “NoSQL Databases Explained,” [Online]. Available: http://www.mongodb.com/learn/nosql.

[16] E. Brewer, “CAP Twelve Years Later: How the "Rules" Have Changed,” [Online]. Available: http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed. [Acedido em 30 09 2014].

[17] “MongoDB & Foursquare,” [Online]. Available: http://www.mongodb.com/customers/foursquare.

[18] “Geospatial Indexes and Sharding,” [Online]. Available: http://docs.mongodb.org/manual/applications/geospatial-indexes/.

[19] “geometry (Transact-SQL),” [Online]. Available: http://msdn.microsoft.com/en-us/library/cc280487.aspx.

[20] “geography (Transact-SQL),” [Online]. Available: http://msdn.microsoft.com/en-us/library/cc280766.aspx.

[21] “Configure Windows netsh Firewall for MongoDB,” [Online]. Available: http://docs.mongodb.org/manual/tutorial/configure-windows-netsh-firewall/.

[22] “Deploy a Sharded Cluster,” [Online]. Available: http://docs.mongodb.org/manual/tutorial/deploy-shard-cluster/.

[23] “OPENQUERY (Transact-SQL),” [Online]. Available: http://msdn.microsoft.com/en-us/library/ms188427.aspx.

[24] “GeoNames,” [Online]. Available: http://www.geonames.org/.

[25] “Google Maps Developers,” [Online]. Available: https://developers.google.com/maps/. [Acedido em 08 09 2014].