Universidade do Minho

Laboratório em Engenharia Informática

Base de Dados NoSQL Orientada ao Modelo Serverless

José Ferreira - a78452 Jorge Oliveira - a78660

Supervisor: **Ricardo Vilaça**

23 de Junho de 2019

Resumo

Neste relatório vamos apresentar a nossa proposta para uma Base de Dados *NoSQL* orientada ao modelo *Serverless*.

Iniciamos com uma contextualização e fundamentação do problema em seguida identificamos os objetivos do projeto. De seguida, expomos a arquitetura do sistema, todos os passos de desenvolvimento e ainda a solução proposta pelo grupo com alguns detalhes da implementação. Posteriormente, apresentamos os testes de avaliação de desempenho da base de dados comparando a sua evolução ao longo das etapas.

No final identificamos quais foram os maiores desafios enfrentados durante a realização do projeto e ainda alguns problemas da nossa aplicação, propondo melhorias para serem implementadas numa versão futura.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Contextualização

O mais comum no desenvolvimento de aplicações é que os pedidos efetuados são direcionados para o servidor responsável por correr a nossa aplicação. Isto faz com que sejamos responsáveis pela gestão dos recursos da máquina, o que vai acarretar alguns problemas.

O servidor estará sempre a correr mesmo que não possua qualquer tipo de utilização o que vai aumentar os custos, já que poderemos estar a pagar por um serviço que não está a ser utilizado. É nossa a responsabilidade para efetuar a manutenção de servidores e de o manter online, bem como pelo controlo ao nível de segurança do mesmo. Caso exista um grande número de pedidos e seja necessário aumentar a capacidade do servidor tal é conseguido aumentando os recursos do servidor (como por exemplo efetuando uma melhoria do *hardware*).

Para os desenvolvedores comuns e pequenas empresas todas estas tarefas acarretam custos elevados, tanto a nível monetário como humano. É necessário perder tempo a efetuar todas estas configurações

e a garantir todas as funcionalidades o que faz com que se desviem do principal objetivo da aplicação que estão a desenvolver. Nas grandes empresas podem não "enfrentar"este problema devido ao capital que possuem para poderem contratar uma equipa especializada, no entanto pode sempre atrasar o desenvolvimento aplicacional e tal não é desejado.

A **arquitetura serverless** promete uma mudança de paradigma resolvendo estes problemas, mais concretamente num modelo *Function as a Service* - **FaaS**. Os provedores de serviço são responsáveis por executar pedaços de código, tendo que ser os mesmos responsáveis pela gestão de recursos e só se paga o tempo de utilização e nada mais. No FaaS este código é "enviado"através da escrita de funções que podem suportar inúmeras linguagens.

1.2 Motivação e Objetivos

Todas as aplicações possuem uma base de dados para permitir guardar os dados persistentemente. Para tal, necessitam de um servidor de base de dados que estará permanentemente em execução.

O objetivo é a construção de uma base de dados apta para um modelo *serverless* para que por intermédio de funções as aplicações possam ter acesso à mesma sem que seja necessário possuir um servidor de base de dados sempre ativo. Permitimos assim a redução de custos e uma maior simplicidade para aceder aos dados guardados já que as funções são codificadas na linguagem que os desenvolvedores se sintam mais à vontade (desde que possua suporte para a mesma).

A base de dados deve conter as seguintes caraterísticas:

Key-Document - o modelo de dados deveria ser um par chave-valor. Sendo que a chave possa ser genérica e o valor é um documento

sendo que os mesmos devem de ser JSON

Serverless - respondendo a eventos referentes a funções, tratando os mesmos

Distribuída e Tolerante a falhas - evitando que os dados fiquem guardados todos num mesmo local (georeplicação)

API Simples - com as operações fundamentais: *get, put, remove e scan*

Possuindo a base de dados com estas caraterísticas permitíamos desta forma que a mesma pudesse ser acedida a qualquer momento e de qualquer lugar sem que o utilizador se precisasse de preocupar com a gestão dos dados, bem como a localização dos mesmos, problemas relacionados ao nível de segurança e que só pagasse quando utilizasse a mesma.

Capítulo 2

Desenvolvimento Aplicacional

2.1 Arquitetura

A nossa base de dados é constituída por dois componentes:

Master - tem o papel de coordenar todos os *workers* da base de dados, efetuar a interação com o cliente e ainda efetuar a gestão ao nível dos recursos para garantir que a mesma seja elástica e consiga responder em períodos de grande sobrecarga

Slave - é o *worker* da base de dados, responsável por tratar efetivamente dos pedidos dos clientes e possui um conjunto de chaves pelo qual o mesmo é responsável

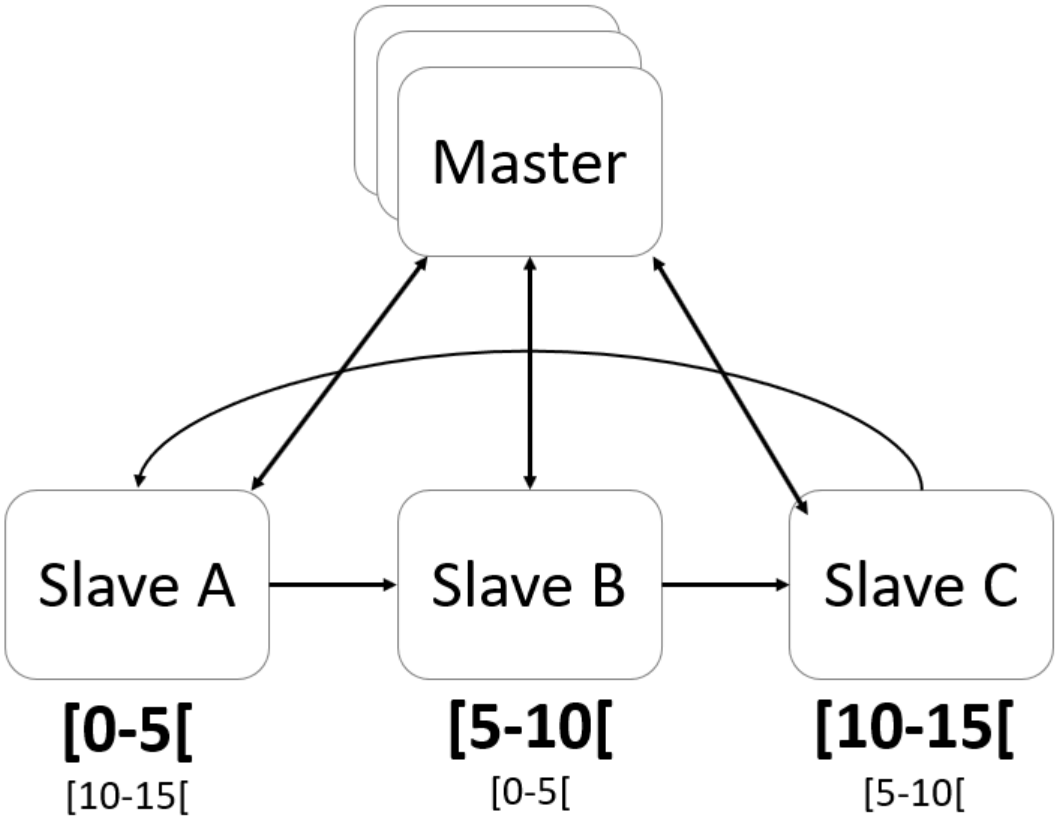

Figura 2.1: Arquitetura Base de Dados

O *master* e os *slaves* estão organizados da forma como é apresentado na figura 2.7. O objetivo é que o *master* estabeleça uma ligação para cada um dos *slaves* existentes, sendo que a comunicação se pode dar nos dois pedidos já que algumas demandas do *master* exigem respostas ou confirmações. Como o *master* é um componente importante na nossa aplicação é imperativo que o mesmo não falhe, mas como tal não é possível controlar a única solução é introduzir replicação para contornar a falha do mesmo (caso ocorra). Para tal, decidimos efetuar várias cópias exatas do mesmo aplicando assim uma técnica de replicação ativa. De salientar, que devido a ao uso desta técnica todas as

operações que o *master* realiza tinham de ser determinísticas o que impede, por exemplo, o uso de números aleatórios para gerar identificadores para as mensagens.

Os *slaves* também são um ponto crucial na nossa implementação, já que os mesmos é que vão guardar persistentemente os dados relativos aos conjuntos de chave que são responsáveis. Desta forma, percebe-se que o mais importante é evitar a "falha"dos conjuntos de chave e não propriamente dos *slaves*. Devido a este pormenor a técnica de replicação utilizada é diferente da do *master*. Neste ponto, decidimos guardar cópias exatas do conjunto de chaves ao invés de possuir cópias exatas de cada um dos *slaves*. Para tal, decidimos reutilizar os recursos que temos disponíveis, já que para além de cada um dos *slaves* ser responsável por um conjunto de chaves possui também outros conjuntos de chaves que são responsabilidade de um outro *worker* do sistema. Na figura 2.7 está representado um fator de replicação de 1, no entanto este pode ser aumentado, sendo que no máximo o fator de replicação pode chegar a **N-1**, sendo N o número de *slaves*.

A replicação serve não só para tornar o nosso sistema tolerante a faltas, mas também poderá melhorar em certo ponto a performance do mesmo. Como temos várias réplicas do conjunto de chaves podemos fazer um balanceamento de carga nas leituras da base de dados. No entanto, isto não significa que quanto mais réplicas se acrescentem mais ganhos vamos possuir, já que vamos ter mais custos para garantir a consistência entre as diferentes réplicas e é necessário efetuar a operação de escrita para todas o que torna o sistema mais lento nestas operações.

O objetivo é que cada *slave* guarde um conjunto de chaves que pode ser de qualquer tipo. A base de dados está preparada para receber chaves genéricas, já que tanto os *slaves* como o *master* apenas lidam com *bytes*. No entanto, não é possível, por exemplo, uma mesma instância da base de dados utilizar Inteiros e Strings, toda a instância deve conter o mesmo tipo de chave.

De salientar que todos os componentes são *"dockerized"* para que tenhamos uma estrutura genérica e mais robusta.

2.2 Incorporação com o Apache OpenWhisk

Para que a nossa base de dados seguisse o modelo *serverless* era necessário efetuar a incorporação da mesma com uma plataforma *Function as a Service*. Decidimos que a mais viável seria o Apache Openwhisk. O objetivo era integrar a nossa base de dados com a plataforma de maneira a que fosse possível utilizar uma aplicação na mesma sendo que os dados estariam guardados na base de dados por nós implementada.

No entanto à medida que investigávamos mais sobre a plataforma verificamos que o principal objetivo não era permitido e que a mesma não deixava, então, integrar qualquer base de dados. s únicas instâncias de base de dados locais criadas por desenvolvedores que esta plataforma deixava integrar tinham de ser da *CouchDB* ou então *Cloudant*. **Desta forma vimo-nos impedidos de tornar a nossa base de dados *serverless* tal como era o objetivo principal do projeto.**

Na verdade o que o Apache OpenWhisk permite, bem como todas as outras plataformas similares a esta, é o desenvolvimento de aplicações. Isto é, o programador implementa a sua aplicação apenas com o foco total no que a mesma deve executar e em seguida "instala"na plataforma definindo uma série de funções que permitem o acesso às funcionalidades da mesma, sem que para tal tenha de possuir um servidor sempre ativo. Uma solução proposta era "ver"a nossa base de dados como uma aplicação deste género, onde definíamos funções que executavam o conjunto de operações permitido pela nossa base de dados. No entanto esta solução não fazia tanto sentido, já que a nossa aplicação seria uma base de dados onde por detrás o que estaria era outra base de dados a efetuar o mesmo trabalho que aplicação por nós implementada.

Desta forma tivemos de mudar um pouco o foco da importância das caraterísticas da nossa base de dados já que a orientação ao modelo *serverless* ficou impossível de fazer. Desta forma, decidimos nos focar mais na **distribuição** e *replicação* da mesma, garantindo que a nossa base de dados possuía todos os condimentos para poder ser utilizada numa plataforma FaaS.

2.3 Estratégia de Desenvolvimento

Inicialmente foram traçadas um conjunto de etapas que deveriam de estar totalmente completadas para que se pudesse avançar para a próxima. Estas etapas na prática foram traduzidas em **protótipos** da nossa base de dados. Utilizámos uma estratégia incremental, sendo que a primeira etapa é a mais simples de toda e à medida que vamos avançando aplicamos também uma maior complexidade na base de dados ficando esta cada vez mais robusta.

2.3.1 Protótipo 0

Numa primeira fase o objetivo era construir o sistema o mais simples possível, sem replicação sem elasticidade, sem cache. Implementar as operações de *get, put, scan* e *remove* diretamente no *worker*, mais uma vez, na sua versão mais simples.

A interação entre o cliente e a base de dados era feita por intermédio de um *stub* que foi codificado de raíz. Apesar de não ser parte integrante da base de dados era necessário para que nesta fase fosse possível estabelecer comunicação entre um cliente e a nossa base de dados. Todos os pedidos deveriam de ser relacionados para o *master* e este reencaminhava para o *slave* correspondente. O número de *slaves* era pré-definido e fixo durante toda a execução.

2.3.2 Protótipo 1

Nesta fase decidimos aumentar a capacidade das nossas operações acrescentando **projeções** e **seleções**. As projeções são conseguidas facilmente devido à utilização de documentos no formato JSON, sendo que era necessário apenas extrair as chaves correspondetes dos documentos. Para que as seleções funcionassem perfeitamente era necessário produzir uma linguagem própria para interpretação da mesma, como este não era o foco principal do nosso projeto, utilizamos o *Predicate* do JAVA que nos permite definir filtros.

Como já foi indicado na secção da Arquitetura o *Slave* é responsável por um conjunto de chaves. De modo a facilitar na distribuição e na robustez do *slave* decidimos criar *chunks* para dividir este conjunto de chaves. O *slave* fica responsável pelo mesmo conjunto no entanto este é dividido em conjuntos mais pequenos.

Melhorámos ainda a operação de *scan*. No protótipo anterior toda a informação da base de dados era carregada para memória. No entanto se fossem guardados gigas de informação tal poderia não ser possível de ser carregada totalmente para a memória. Nesta fase foi então efetuado um carregamento parcial da informação, sendo disponibilizado ao utilizador um iterador para conseguir percorrer todas as chaves da base de dados.

De modo a evitar um *bottleneck* no *master*, já que o mesmo iria receber sempre um pedido do cliente, implementamos cache do lado do cliente. Era guardada a informação sobre que conjuntos de chaves é que pertenciam ao *slave* que foi interrogado.

2.3.3 Protótipo 2

Nas fases anteriores caso algum componente falhasse o nosso serviço deixaria de funcionar. Desta modo, nesta fase decidimos aplicar a re-

plicação tanto ao nível do *master* como ao nível dos *slaves*. Na secção da *Arquitetura* já foram dados os traços gerais para implementar a replicação.

2.3.4 Protótipo 3

Neste protótipo foi implementado a redistribuição de chaves para tratar da elasticidade da nossa base de dados. Desta forma, caso fosse detetado a sobreutilização ou subutilização de um servidor, eram acrescentados ou retirados, respetivamente, *slaves* para adequar à carga de pedidos que estavam a ser realizados para a nossa base de dados.

2.3.5 Protótipo 4

Até esta fase todas as chaves eram associadas a um tipo especifico, neste caso *Long*. No entanto um dos objetivos é que as chaves pudessem ser genéricas. Desta forma foi alterada a chave para que tanto o *Master* como os *Slaves* apenas trabalhassem com *bytes*. No entanto, a base de dados só funciona com chaves de um só tipo, se uma for um inteiro todas as restantes deverão ser inteiros, como também, se uma chave for uma String então todas as restantes devem de ser String. Decidimos apenas implementar suporte para dois tipos, mas é facilmente extensível para os restantes.

2.4 Solução Proposta e Detalhes de Implementação

A solução proposta pelo grupo contém as seguintes características:

- Modelo com um par: chave -> documento *json*
- Distribuída
- Replicada
- Elástica (adapta-se às cargas no sistema, aumentando/diminuindo o número de *slaves*

De seguida vamos referir alguns detalhes de implementação da solução a nível das operações, replicação e elasticidade.

2.4.1 Operações

Como já foi referido decidimos implementar 4 operações distintas:

get - retorna o documento associado a uma chave. Podem ser aplicados filtros e projeções;

put - inserção de um documento correspondente a uma chave definida pelo utilizador. Caso a chave já exista na base de dados, então esta operação possui caraterísticas de *update* sendo que o documento associado à chave introduzida pelo utilizador será substituído na totalidade;

remove - remoção de um documento associado a uma chave. Podem ainda ser introduzidas seleções e projeções, sendo que nas projeções apenas serão eliminados do documentos os campos correspondentes;

scan - faz um leitura total à base de dados que deve de ser ordenada pelas chaves das mesmas. Pode ainda ser aplicado filtros e projeções ao *scan*;

As três primeiras operações são bastante simples e a estratégia de implementação é similar nas mesmas. Sendo necessário verificar sempre qual o *slave* que trata a chave que queremos atualizar/ler. Para isso o *stub* (que faz a ligação do cliente ao sistema) deve verificar se contém na **cache** o *slave* correspondente e em caso afirmativo envia mensagem para o mesmo. Caso não contenha esse elemento na **cache** deve interrogar o *master* sobre qual o *slave* responsável da chave em questão, atualizando posteriormente a **cache**.

Como introduzimos redistribuição das chaves, a **cache** pode ficar desatualizada. Desta forma pode ser enviado um pedido para um *slave* que já não é responsável pelo conjunto de chaves, pelo que o *stub* é responsável uma nova tentativa da operação, caso esta ainda não tenha terminado, perguntando novamente ao *master* qual o *slave* responsável pelo tratamento daquela chave.

Com a introdução de replicação existe mais do que um *slave* capaz de responder a leituras, sendo que não é necessário que este tipo de operação passe por todas as réplicas responsáveis por um conjunto de chaves já que não vai alterar o estado da base de dados. Na operação de **get** é realizado um *round-robin* nos *slaves* que tratam um determinado conjunto de chaves. Tanto o **put** como o **remove** têm de ser enviados à réplica primária que trata um conjunto de chaves, já que são escritas e alteram o estado da base de dados.

O **scan** foi a operação mais trabalhosa, visto que a base de dados pode conter bastantes dados e não é possível transportar todos para memória do lado do cliente. Para ultrapassar esta dificuldade esta operação é realizada utilizando um iterador, que vai pedindo quantidades de chaves aos diferentes *slaves* (por exemplo 20 chaves de cada vez). Inicia realizando um pedido ao *slave* com o primeiro conjunto de chaves e quando o iterador atual já está terminado faz um novo pedido iniciando na última chave que continha, sendo este processo recursivo até não existirem mais chaves. Para que esta operação seja ordenada os conjuntos de chaves têm uma ordem definida e os pedidos são realizados por essa ordem. O *stub* é responsável por manter o estado da

posição de leitura do *scan*.

2.4.2 Replicação

Para que o nosso sistema seja tolerante a faltas decidimos introduzir replicação, tanto no *master* como no *slave*. Para a replicação decidimos utilizar o *toolkit* de comunicação em grupo *Spread*.

Replicação ao Nível do Master

A técnica de replicação utilizada no *master* é replicação ativa e o fator de replicação, por defeito, possui o valor dois (ou seja existem 3 nós). Decidimos utilizar replicação ativa pelo facto das operações serem todas determinísticas (apenas são de controlo), sendo necessário que as mensagens a ele enviadas sejam com ordem total (*total order multicast*). Relativamente à recuperação de estado é enviado todo o estado de gestão dos *slaves* e as meta-informações guardadas sobre os mesmos.

De salientar que como temos várias réplicas exatas do *master* foi necessário lidar tanto nos *workers* como no lado do Cliente com mensagens repetidas. Já que os mesmos quando comunicam com o *master* iriam sempre receber N mensagens, sendo N igual ao número de *masters* que funcionam corretamente.

Replicação ao Nível do Slave

Relativamente ao *slave* decidimos utilizar replicação passiva, como já foi referido anteriormente. Neste caso o protocolo de replicação passiva é um pouco diferente do que estávamos habituados, visto que quando existe a falha de um primário, o *master* tem de detetar essa

falha e voltar a iniciar outro *slave* com o mesmo identificador, entrando este como primário em alguns conjuntos de chaves. Esta entrada como primário pode levar a que alguns pedidos sejam atrasados, tendo que ser retransmitidos para terem sucesso. Como estamos a usar replicação passiva é necessário que as escritas sejam propagadas do primário para os secundários. Neste caso decidimos esperar por todos os *acks* para garantir consistência entre as réplicas.

Para que uma réplica seja reintegrada é necessário que recupere o estado atual do grupo. Para realizar essa recuperação, visto que usamos *containers*, decidimos passar todo o estado atual já que os mesmos são criados a "partir do zero"não tendo portanto qualquer informação guardada de outro momento. O estado transferido são todas as chaves e os seus documentos que o *slave* guarda.

Ora à semelhança da operação *scan* esta informação pode não ser passível de ser carregada totalmente para memória, desta forma decidimos repartir a transferência de estado relativa aos conjuntos de chaves entre os diferentes *slaves*. Caso a réplica que entrou no grupo detete a falha da réplica que lhe estava a enviar estado pede a outra réplica que já se encontrava no grupo antes desta ter entrado inicialmente. É importante que seja uma réplica que já estivesse no grupo para ter a certeza que a mesma possui todo o estado. Esta é uma situação improvável de acontecer, mas que caso aconteça necessita de ser tratada. As mensagens de *ACK* não são mais consideradas quando há mudança de primário visto que o *toolkit Spread* com a garantia de *View Synchrony* garantem que quando existe uma mudança de vista todas as mensagens da vista anterior são entregues a todos os processos corretos.

2.4.3 Elasticidade

Para que o nosso sistema seja adaptável a cargas dinâmicas é necessário que o mesmo seja elástico, ou seja, que seja de capaz de efetuar um aumento do número de *slaves* quando a carga é elevada e diminua

quando é baixa. Para verificar a carga no sistema cada um dos *slaves* envia, periodicamente, uma mensagem para o *master* com as seguintes informações:

- percentagem de **cpu** utilizada
- quantidade de memória livre
- quantidade de leituras e escritas ocorridas no intervalo de tempo periódico

Quando o *master* deteta que a percentagem de *cpu* está acima de um determinado valor ou a quantidade de memória está abaixo de outro e o número de *slaves* atuais no sistema não é igual ao número de conjuntos então decide criar mais um *slave* para suportar a sobrecarga. Se existirem o mesmo número de conjuntos que o número de *slaves* então não faz sentido introduzir um novo *slave*.

Quando o *master* deteta que uma média ponderada entre a quantidade de escritas e leituras está abaixo de um determinado valor e o número de *slaves* é superior ao número mínimo (número inicial de *slaves*), então decide eliminar o *slave* que lhe enviou a informação tratando assim a sub-carga do sistema.

2.4.4 Algoritmo de Criação de um Novo Slave

Quando o *master* deteta a sobrecarga de um *slave* deve proceder para a criação de um novo *slave*, como já referido anteriormente.

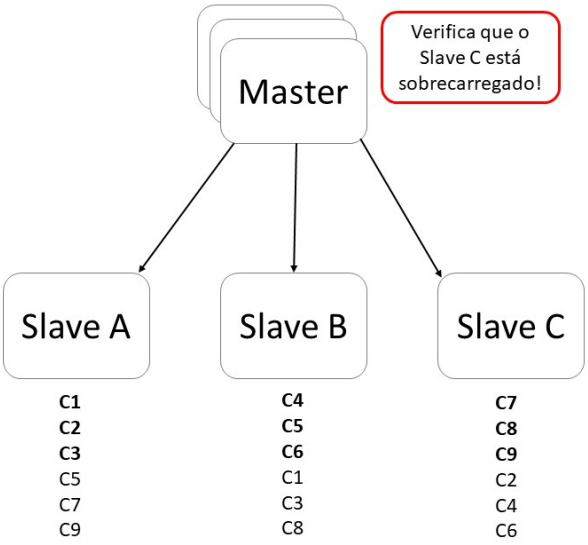

Figura 2.2: Deteção de Sobrecarga

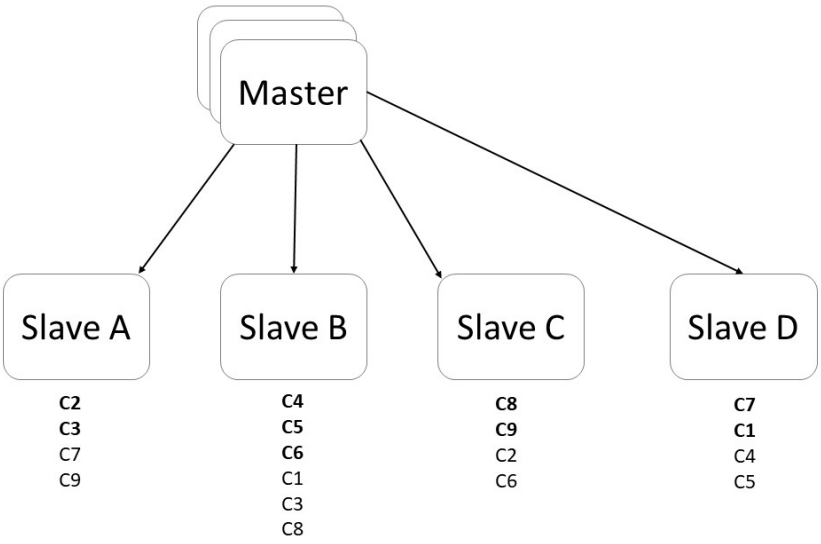

Figura 2.3: Aumento do Número de Slaves

Para criar um novo *slave* o *master* verifica qual o conjunto de chaves que se encontra mais sobrecarregado do *slave* que detetou como estando com carga excessiva. Porém, para que o sistema continue balanceado, ao nível da quantidade de chaves que cada *worker* é responsável, poderá ser necessário transferir mais conjuntos para o mesmo. Depois de ter sido detetado qual o conjunto "problemático"é selecionado, utilizando um *round-robin*, dos restantes *slaves* outros conjuntos, com menos carga, até que o novo *slave* contenha o número de conjuntos necessários.

2.4.5 Algoritmo de Eliminação do Slave

Quando o *master* deteta a subcarga de um determinado *slave* decide efetuar a excluisão do mesmo, distribuindo as suas chaves equitativamente pelos restantes *slaves* corretos presentes no sistema.

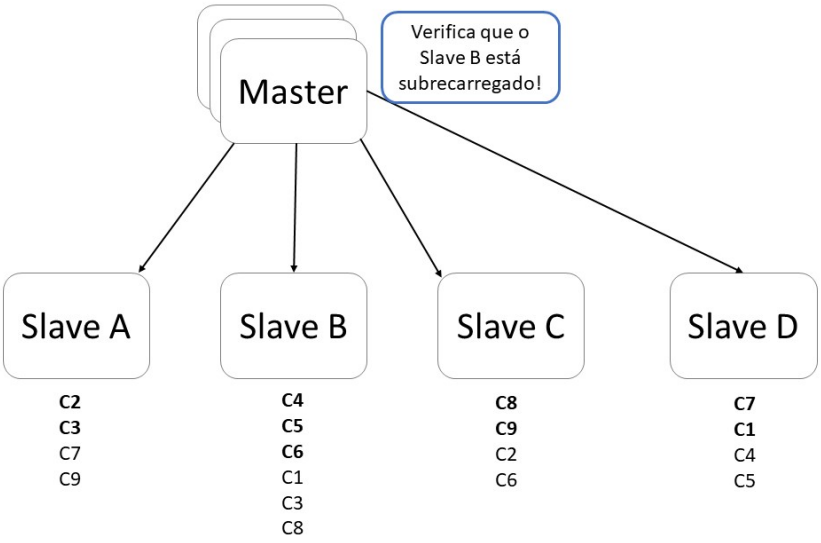

Figura 2.4: Deteção de Subcarga

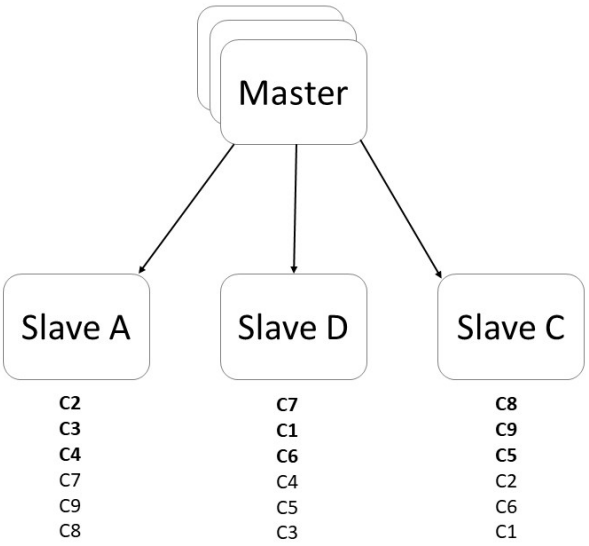

Figura 2.5: Diminuição do Número de Slaves

Por cada conjunto de chaves primários são selecionados aproximadamente N secundários, sendo N o **fator de replicação**.

O objetivo deste algoritmo é manter um balanceamento entre o número de conjuntos de chaves primários e secundários que cada *slave* trata e não permitir que um determinado *slave* contenha conjuntos de chaves repetidos (primários e secundários).

2.4.6 Redistribuição das Chaves

Após o aumento e diminuição no número de *slaves* é necessário garantir que o novo contém um estado atualizado dos grupos a que se vai juntar, para garantir consistência. Decidimos para isso tirar proveito da transferência de estado da replicação passiva dividindo em 2 casos diferentes:

- Quando se junta ao grupo com o papel de secundário para o conjunto de chaves
- Quando se junta ao grupo com o papel de primário para o conjunto de chaves

Caso entre como secundário, é garantido que o identificador com que entra no grupo é sempre crescente e diferente dos outros pelo que pode se juntar ao grupo sem qualquer problema associado e é "tratado"como uma falha de um secundário.

Caso entre com o papel de primário tem de entrar com o mesmo identificador do primário anterior. Desta forma antes de se poder juntar ao grupo tem de esperar uma confirmação do *master*, que por sua

vez tem de receber uma confirmação dos *slaves* que eram primários a indicar que já saíram dos grupos. Após esta confirmação então o *master* introduz a diretiva, para o "novo primário", de que o mesmo pode se juntar ao grupo.

Continua a utilizar a transferência de estado da replicação passiva, visto que contemplamos o caso em que um novo *slave* entra como primário.

2.5 Avaliação de Desempenho

Para avaliar o desempenho da nossa base de dados utilizamos um serviço padrão de *benchmarking*, o *YCSB*. Inicialmente foi necessário realizar a ligação entra a nossa base de dados e o serviço o que nos criou algumas dificuldades. No entanto após uma observação do exemplo do *RocksDB* e com a configuração correta conseguimos realizar a ligação.

Para termos alguma capacidade de avaliação da performance da nossa base de dados, decidimos comparar os resultados obtidos com os do *RocksDB*. Naturalmente não esperávamos que os nossos testes fossem melhores, já que cada *slave* utiliza o *RocksDB* para guardar a informação associada às chaves.. Seria adequado, no futuro, comparar com uma base de dados que também seja replicada e distribuída como por exemplo *Cassandra*. Os testes forem então realizados sobre o *RocksDB* e a nossa base de dados nas suas diferentes versões:

- Sem replicação e sem redistribuição;
- Com replicação;
- Com replicação e redistribuição;
- Com replicação, redistribuição e chaves genéricas;

De seguida decidimos os *workloads* a utilizar, escolhendo os seguintes:

- A - com 50% de *reads* e 50% de *updates*
- E - com 30% de *scans* e 70% de *inserts*

Posteriormente definimos que o *load* seria sempre realizado com 25000 registos e que para a avaliação seriam utilizados os seguintes números de operações:

- 500;
- 1000;
- 1500;
- 2000;
- 2500;
- 3000;
- 3500;
- 4000;

Para gerar um gráfico com os resultados, decidimos utilizar o exemplo *ycsb2graph*, obtendo os seguintes resultados.

2.5.1 Workload A

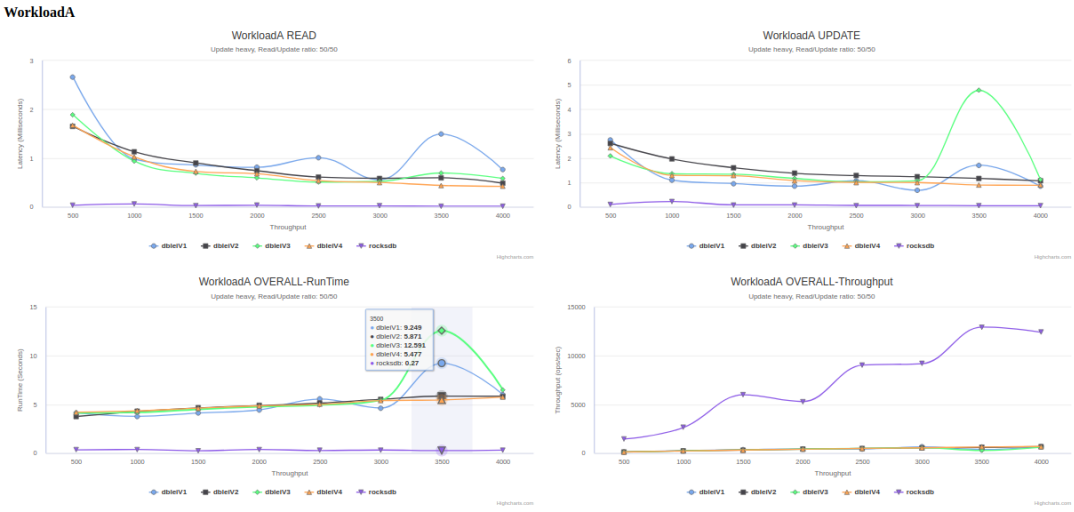

Figura 2.6: Resultados do Workload A

Numa primeira análise dos resultados obtidos verificámos que o *RocksDB* possui uma melhor performance, quando comparada com qualquer das versões da nossa base de dados.

Relativamente às quatro versões diferentes da nossa base de dados podemos verificar que as leituras se comportam aproximadamente da mesma forma, sendo até ligeiramente melhores nos casos em que existe replicação e redistribuição comparado com a versão inicial. Respetivamente aos *updates* podemos verificar que a latência é ligeiramente superior nos casos em que existe replicação, o que é normal pelo facto de ser necessário que as réplicas se mantenham consistentes. Podemos no entanto, verificar que esta diferença não é muito significativa o que é bastante satisfatório.

Relativamente aos resultados gerais podemos verificar que tanto no tempo de execução como no débito o *RocksDB* é bastante mais eficiente do que a nossa base de dados.

Alguns aspetos positivos que podemos verificar é que a latência, em média, tem uma tendência para diminuir tanto nas leituras como nos *updates* à medida que são realizadas mais operações. Para além disso podemos também verificar que o débito tem tendência para aumentar, apesar de muito lentamente.

2.5.2 Workload E

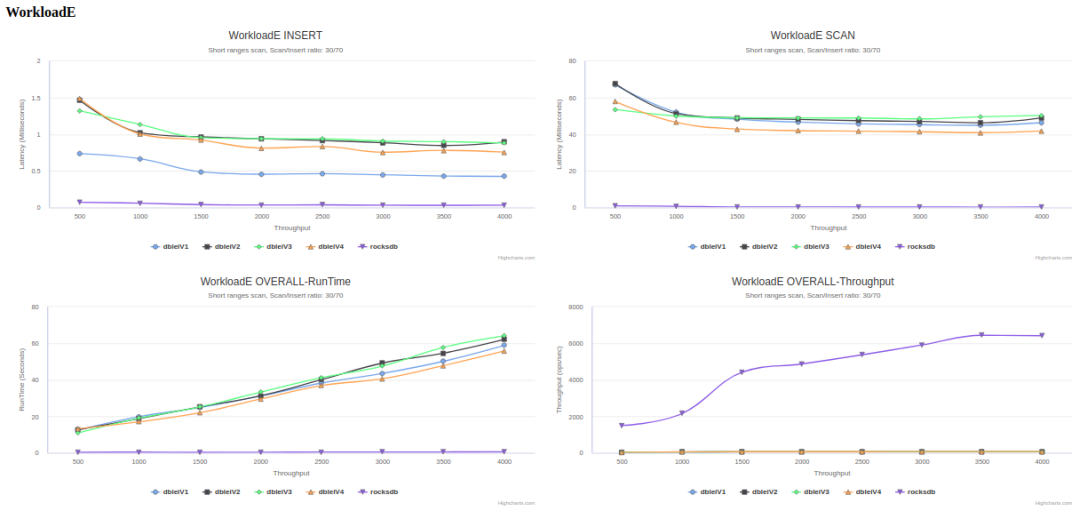

Figura 2.7: Resultados do Workload E

Neste *workload* podemos verificar que já existe uma diferença significativa entre os *inserts* nas versões que contêm replicação quando comparadas com a versão inicial.

As quatro versões da nossa base de dados têm um desempenho bastante similar, tanto nas operações de *scan* como no global. A diminuição da latência nos *scans* é bastante mais reduzida quando comparada com os casos anteriores, e no global o débito praticamente mantém-se constante.

Um dos aspetos positivos que podemos retirar dos testes é o facto de todos terem sido efetuados com sucesso, o que nos indica que não houve erros.

Capítulo 3

Conclusão

3.1 Trabalho Futuro

Obviamente um dos pontos que ficou por executar na nossa implementação foi a incorporação com uma plataforma FaaS. Desta forma este seria um dos pontos a melhorar numa versão mais avançada da nossa base de dados, desde que seja encontrada/criada uma nova plataforma que permita então a instalação da mesma. É um ponto que não está só dependente de nós para o conseguir.

Um dos pontos que podem ser melhorados é a forma como disponibilizamos as operações que envolvem filtros e projeções. A forma encontrada é bastante simples, no entanto não é *user friendly*, já que utiliza apenas a sintaxe do JAVA tanto para as projeções como para os filtros. O ideal seria implementar uma linguagem própria para realizar este tipo de operações. No entanto, para conseguir tal coisa era necessário codificar um *parser* para que fosse validado toda a sintaxe inserida pelo utilizador. Desta forma, cada usuário de uma forma muito mais cómoda seria capaz de realizar projeções e seleções nas suas *queries*.

Para a distribuição é necessário validar se os *workers* estão sobrecarregados ou não. A estratégia utilizada foi bastante simples, sendo que os próprios *slaves* efetuavam uma monitorização e periodicamente enviavam os dados para o *master* que iria analisá-los e caso fosse necessário acrescentava ou diminuía o número de *slaves*. No entanto esta abordagem é sensível a picos, sendo que apenas vamos alterar a elasticidade da nossa base de dados em momentos de uma grande afluência de pedidos ou então quando a mesma quase não os recebe, no entanto esta afluência também tem de coincidir com os períodos de monitorização. Durante os períodos pode existir uma grande quantidade de pedidos que não é detetada e dessa forma pode não existir uma atuação do *master* para aumentar a flexibilidade da base de dados.

Para resolver esta situação poderíamos usar médias ao invés de pontos de pico, os *slaves* poderiam efetuar medições mais frequentemente e serem os mesmos a indicar ao *master* que estavam sobrecarregados através de um algoritmo mais robusto e ainda acrescentar *machine learning* já que permite implementar um sistema de conhecimento que possa fazer aumentar ou diminuir o número de *slaves* nas alturas mais pertinentes.

Outro aspeto importante de ser melhorado era a replicação. A própria ferramenta de comunicação utilizada, *Spread*, não se revelou a ideal para este tipo de replicação, sendo que com uma pesquisa e orientação poderíamos utilizar uma nova ferramenta que nos permitisse diminuir a complexidade gerada no processo de replicação.

Cada componente da nossa arquitetura foi "*dockerized*" o que acrescenta algum grau de independência da nossa aplicação. No entanto, devido ao uso do *Spread*, não conseguimos executar os componentes em máquinas diferentes. Uma melhoria assinalável na nossa base de dados é a introdução da georeoplicação que pode ser feita por intermédio do *kubernetes*. Já nesta versão tentámos utilizá-lo, mas verificámos que tal não era possível fazer de momento, pelo que teve de ser totalmente o *master* responsável pela criação e lançamento de *slaves*. Numa versão posterior onde será implementado o *kubernetes* teremos uma solução

mais robusta e genérica sem que seja necessário utilizar comandos *sh* e afins.

Um problema que temos é que como na redistribuição das chaves utilizamos a transferência de estado do protocolo de replicação passiva e retiramos uma réplica do grupo temos de ter no mínimo 2 réplicas com o estado recuperado. Ou seja, o *master* apenas podia iniciar um aumento/diminuição no número de *slaves* quando tem a confirmação de que todos os grupos que vão ser alterados contêm no mínimo 2 réplicas com o estado recuperado. Uma outra possível solução seria que o novo primário entrasse com um identificador superior ao do anterior mas inferior a todos os secundários (visto que a eleição do primário é pelo menor identificador) e que o primário anterior apenas saísse do grupo quando tivesse enviado todo o estado ao novo primário. Futuramente este é um aspecto chave a ser melhorado para que a elasticidade seja o mais transparente possível.

O nosso algoritmo de distribuição pode ter o problema de não balancear corretamente o sistema, pelo facto de tentar colocar sempre um conjunto de chaves num *slave* em vez de verificar quais os *slaves* que podem ceder/receber um conjunto de chaves, ou seja tendo um panorama global e apenas fazendo a redistribuição depois. Tanto o algoritmo de aumento como diminuição podem ser melhorados no futuro para garantir uma elasticidade quase perfeita.

3.2 Desafios de Implementação

A ligação ao Apache OpenWhisk foi uma dificuldade encontrada durante o decorrer da codificação da nossa base de dados. E foi uma sobre a qual não conseguimos implementar, visto que não existia uma plataforma que nos permitia realizar o que era desejado. Esta dificuldade não está relacionada a nível de código, já que não foi possível efetuar, mas sim pelo facto de não existir grande documentação sobre o que

queríamos fazer (já que quase tudo o encontramos tinha a haver com a incorporação de uma aplicação e não com a "instalação"de uma base de dados) e pelo facto de o grupo ter disponibilizado bastante tempo para tentar encontrar uma solução, em que a que foi encontrada posteriormente verificamos que não fazia sentido implementá-la de acordo com as caraterísticas da nossa base de dados.

A replicação também apresentou algumas dificuldades devido aos conhecimentos por nós adquiridos e também à única ferramenta que conhecíamos para implementar replicação.

Foi nos incutido conhecimento sobre como é que é possível replicar máquinas, mas neste projeto a replicação não estava relacionada mesmo com a máquina, no caso dos *slaves*, mas sim com o conjunto de chaves existentes. Os modelos de replicação por nós aprendidos foram a replicação ativa e passiva, no entanto nenhuma das duas se adequava na perfeição a que queríamos fazer. Usamos uma implementação bem parecida com a passiva, no entanto pelo facto de o primário poder mudar sem que haja a falha do mesmo dificultou a sua implementação já que era necessário encontrar uma estratégia que permitisse que todos os *slaves* envolvidos num conjunto de chaves possuísse a mesma visão sobre qual deles é que era o primário.

A única ferramenta conhecida por nós para a implementação de replicação é o *Spread Toolkit*. No entanto, verificámos que não era a melhor forma para implementar a replicação ao nível de conjunto de chaves, onde foi necessário criar um grupo por cada conjunto de chaves, depois um global para que os *masters* tenham a noção de quando os *slaves* falham e ainda um para enviar mensagem para todos os *masters*. Eram muitas conexões e grupos para gerir o que tornou algo mais complexa a codificação da replicação da base de dados. O facto de usar o *Toolkit* de comunicação *Spread* faz com que seja obrigatório que todos os componentes tenham acesso a uma mesma conexão para que se possam conhecer totalmente, o que nos impediu de efetuar a georeplicação que seria conseguida com o *kubernetes*.

O *parsing* das mensagens relacionados com a replicação é algo complexo com a utilização do *Toolkit* de comunicação *Spread* já que era necessário o uso de *ifs* encadeados o que dificultava a perceção do código e diminui a performance da nossa aplicação.

3.3 Conclusão

Em suma, com exceção do problema encontrado sobre a incorporação da nossa base de dados à plataforma *Apache OpenWhisk* que se revelou impossível de resolver com uma solução viável, alheia ao nosso grupo, conseguimos implementar todas as caraterísticas traçadas inicialmente para a base de dados NoSQL. Desta forma, fez com que a nossa base de dados fosse uma "base de dados normal"mas com elasticidade e replicação já incorporadas sem que o cliente que a utilize se tenha de preocupar com estes problemas.

Não podemos afirmar que ficamos completamente satisfeitos com o trabalho realizado, visto que não conseguimos cumprir o principal objetivo e este é que traria algo de diferente à nossa implementação, no entanto face às adversidades e ao conhecimento adquirido durante a investigação e execução do nosso projeto ficamos agradados com a experiência adquirida e com o resultado obtido.

Bibliografia

[1] Amazon Aurora Serverless: https://aws.amazon.com/pt/rds/aurora/serverless/

[2] Microsoft Azure: https://docs.microsoft.com/en-us/azure/cosmos-db/serverless-computing-database

[3] Fauna - base de dados serverless: https://fauna.com/blog/escape-the-cloud-database-trap-with-serverless

[4] Novkovic, Nemanja (2018) *What Is a Serverless Database? (Overview of Providers, Pros, and Cons)*. Disponível em https://dzone.com/articles/what-is-a-serverless-database-overview-of-provider

[5] Apache OpenWhisk: https://openwhisk.apache.org/

[6] Github do Apache OpenWhisk: https://github.com/apache/incubator-openwhisk/tree/master/common/scala/src/main/scala/org/apache/openwhisk/core/database

[7] Filters HBase: https://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/FilterList.html

[8] GitHub do YCSB: https://github.com/brianfrankcooper/YCSB

[9] Github ycsb2graph: https://github.com/chinglinwen/ycsb2graph

[10] Guia Kubernetes: https://shekhargulati.com/2019/02/02/the-kubernetes-guide-part-1-learn-kubernetes-by-deploying-a-real-world-app

[11] Tutorial Kubernetes: https://blog.couchbase.com/databases-on-kubernetes/

[12] Rahm, Erhard; Stiihr, Thomas *Analysis of Parallel Scan Processing in Shared Disk Database Systems*