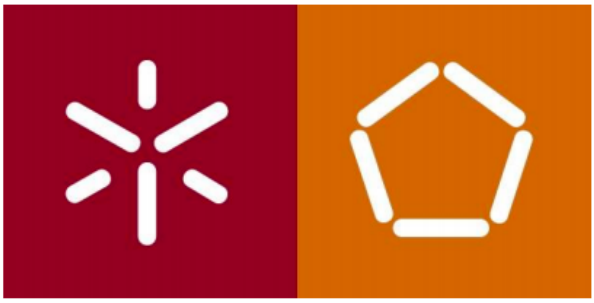

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Engenharia de Sistemas de Computação

Servidor de Bases de Dados em Larga Escala

Trabalho Prático 2

Eduardo Lourenço da Conceição (A83870)
Rui Nuno Borges Cruz Oliveira (A83610)

29/05/2021
Braga

Índice

1 Introdução

Para o segundo trabalho de Engenharia de Sistemas de Computação, foi-nos proposta a implementação de um sistema de Bases de Dados simples, utilizando o paradigma de cliente servidor, de modo a podermos analisar o comportamento do sistema, em particular estimando o seu comportamento num sistema de alto desempenho e de elevada procura, calculando o custo de instalação e manutenção do mesmo.

Neste relatório iremos expor as decisões que tomamos para criar o sistema, que aspetos tiveram particular atenção para que possamos melhor a *performance* do mesmo, que otimizações poderíamos ainda implementar e como é que esperamos que o sistema se comporte quando escalarmos a procura do mesmo.

2 Requisitos do Problema

O cerne do projeto é criar um servidor de bases de dados e entender como é que ele se comporta num contexto de larga escala. Mas como é que é definida, no contexto do enunciado, esta "larga escala"?

Para podermos estimar os custos de implementar o sistema em grande escala, devemos ter em consideração os seguintes requisitos:

- O volume inicial de dados que a base de dados alberga é de 1 *terabyte*, com a possibilidade de crescer até mais de 100 *terabytes* (iremos utilizar 100 TB como valor base);
- Teremos milhões de pedidos de leitura e de escrita por segundo, sendo que devemos tomar por base 10 000 000 de leituras por segundo e 500 000 escritas por segundo;
- A latência interna de resposta a um pedido não deverá ser maior do que 1 milissegundo;
- O número de clientes da aplicação estará entre 1000 e 100 000.

3 Implementação da Solução

3.1 Servidor de Base de Dados

O primeiro passo do projeto foi criar um servidor de base de dados simples, pelo que será o que descreveramos nesta secção.

3.1.1 Arquitetura e *Design*

O servidor foi implementado em C++, sendo o mesmo *multithreaded*, utilizando uma *thread* por cliente e um *socket* por cliente. Este é o modelo preferido quando tratamos de clientes que esperamos terem alguma longevidade na sua ligação ao servidor, pelo que diminuímos o *overhead* de estabelecimento de ligações assim, quando

comparado ao modelo de uma *thread* por pedido. Outros possíveis caminhos que poderíamos ter tomado neste aspeto serão descritos mais tarde. A utilização do C++ em detrimento do C também tem uma razão, que é a biblioteca de *threads* que podemos usar. As *threads* POSIX (*pthreads*) que temos no C são muito mais *hardware dependent* que as *std::threads*, cuja implementação está estritamente definida pelo *standart* de C++, desde o C++11, como vimos nas aulas da primeira parte da cadeira, pelo que o código se torna mais portátil quando utilizamos estas.

A base de dados funciona como uma espécie de *hash table* remota, sendo que as chaves da mesma são *long long* (8 *bytes*), e os valores são *strings* de tamanho fixo de 1024 *bytes*. Assim sendo, cada entrada na *hash table* terá 1032 *bytes* (excluíndo outros elementos que fazem parte da estrutura da *hash table*, nomeadamente apontadores).

Cada cliente liga-se à Base de Dados através de um *socket* TCP, e pode fazer pedidos do tipo *GET* ou do tipo *PUT* (consulta e inserção, por outras palavras). Os pedidos do tipo *GET* deverão vir com a chave do elemento que queremos procurar e os do tipo *PUT* deverão vir com a chave do elemento que queremos inserir e o seu valor. A mensagem que contém os pedidos deverá vir etiquetada com um identificador que nos diz qual o tipo da mensagem (um simples número inteiro, não um *bool* pois assim permitimos que sejam adicionados novos tipos de pedidos sem obrigar a grandes mudanças nesse aspeto).

De modo a que mantenhamos a pré-condição de uma *hash table* das operações de consulta e inserção serem de tempo constante ($O(1)$), a tabela em memória funciona como uma *look-up table*, com a sua implementação sendo um *array* cujos índices nos são dados a partir da função de *hash*. Em disco, o ficheiro em si é um ficheiro de texto em que cada registo tem exatamente o mesmo tamanho, sendo ele 20 caracteres para a chave, 1024 para o valor, um espaço a separar os dois e um '\n' a separar os registos. Estes dois últimos não são estritamente necessários, apenas os incluímos para propósitos de *debug*, e o sistema seria adaptado para remover isto no futuro. Como os registos têm tamanho fixo, a sua procura e escrita pode ser feita em tempo constante utilizando *seeks*. Estes *seeks* no entanto são problemáticos pois, nos sistemas Linux, um único *file pointer* existe para cada ficheiro num dado ponto no tempo, ou seja, não podemos aceder ao ficheiro em paralelo desta forma, o que implica que pedidos que obrigam a acessos ao ficheiro têm de ser controlados com um *lock* e, portanto, serializados, o que se prova um enorme *bottleneck* na *performance* da BD. Apesar de termos pensado em estratégias para dar a volta a este problema, que falaremos mais tarde, não as implementamos.

Em termos de controlo de contenção, falamos já do aspeto de acessos concorrentes ao ficheiro, mas outro aspeto em que temos cuidado é na tabela de *hash* em memória. A mesma também tem um *lock* associado. Isto é para prevenir que hajam acessos concorrentes à tabela que escrevam nos mesmos registos, causando *dirty writes*. Mais uma vez, ter apenas um *lock* associado à estrutura toda é dispendioso, uma vez que vai obrigar as *threads* a esperarem ela sua vez para escrever na tabela, mesmo que a sua escrita não afetasse a escrita atual, e obriga também a chamadas ao *kernel* para que possamos obter o *lock*, que é um *mutex*, uma operação cara.

3.1.2 Otimizações Futuras

De modo a que o sistema funcionasse no seu pleno num contexto de larga escala, existiriam outras otimizações que nos parece, em teoria, que provocariam efeitos positivos na *performance* do servidor, dos quais podemos destacar:

- O ficheiro que utilizamos para persistência de dados é um *bottleneck* enorme em várias vertentes, mas a serialização de acessos ao mesmo é o maior. Existem duas otimizações que poderíamos fazer ao mesmo. A primeira, e a maior, seria dividir o mesmo em vários ficheiros, em vez de um só, cada um contendo um *chunk* da Base de Dados, apenas uma fração da total. Utilizando, por exemplo, para o nosso caso de estudo em que o ficheiro tem 1GiB de dados, poderíamos dividir o mesmo em 100 ficheiros, cada um com aproximadamente 10MiB, mais ou menos mil registos, cada um com *locks* prórios. Assim poderíamos aceder a cada *chunk* em paralelo, levando a que possamos fazer até 100 acessos a memória persistente ao mesmo tempo, mas a um *chunk* ainda acedemos sequencialmente. Para resolvermos este problema deveríamos usar alguma forma de *Parallel I/O*[5], nomeadamente *MPI IO*, que nos permite ter vários *file descriptors* para o mesmo ficheiro, o que nos permitiria fazer acessos paralelos ao mesmo ficheiro. Mesmo para isto, seria necessário ter cuidado com acessos ao mesmo ficheiro, de modo a não haver escritas concorrentes no mesmo registo. Estas otimizações são estritamente necessárias, pois da forma que estamos a fazer acabamos por obrigar a que haja apenas 2 pedidos a ser executados no máximo a qualquer momento do tempo (um que vá a ficheiro e um que apenas aceda à memória), uma vez que utilizamos um *lock* global para cada. Esta otimização iria permitir que fossem executados tantos pedidos num dado instante de tempo quanto *shards* da base de dados, o que levaria a um *speedup* significativo a nível do tempo de execução dos pedidos;

- Outro problema que temos é o *lock* à estrutra inteira da *hash table*, que serializa acessos à mesma. Para resolver este problema, deveremos manter este *mutex*, mas adicionar um às entradas também. Desta forma, passaríamos a fazer *lock* da estrutura sempre que queremos aceder à mesma, retiramos o registo que queremos analisar, fazemos *lock* do mesmo e *unlock* da tabela, tratamos das operações necessárias com o registo e depois fazemos *unlock* do mesmo. Assim, a escrita e leitura dos registos deixa de ser parte da zona crítica protegida pelo *mutex* da estrutura, passando a ser uma zona crítica isolada, o que diminuiria o tempo em que cada *thread* tem o *lock*, mas aumentaria a memória necessária, uma vez que um *std::mutex* do C++ ocupa 40 *bytes*, uma quantidade que em larga escala não é negligenciável;

- Os dados que são guardados em memória são estáticos, o que é suficiente para um caso como o nosso em que testamos com pedidos para chaves aleatórias, mas não no contexto do mundo real em que o princípio de localidade temporal, que nos diz que dados que foram acedidos recentemente têm uma alta probabilidade de ser acedidos outras vez no futuro próximo, é algo que temos de ter em conta. É possível haver registos que apenas estão em ficheiro e que são acedidos com

frequência, o que não é ideal. Para resolver este desafio, passar a utilizar a tabela de *hash* com um esquema de substituição dos registos (nomeadamente *Least Recently Used*) deverá trazer efeitos positivos sobre o desempenho, mantendo não um conjunto estático e, de certa forma, aleatório de registos em memória mas sim um conjunto com alguns registos que em princípio serão mais populares;

- Outra das otimizações passa pela forma como os dados são comunicados entre o cliente e o servidor (e vice versa). De momento, o cliente usa *strings* para fazer pedidos ao servidor. No entanto, apesar de facilitar a legibilidade, não é a melhor opção quando falamos em *performance*, uma vez que a quantidade de *bytes* vai variar conforme os parâmetros de cada operação, por exemplo, para o *GET*, o número de *bytes* usados para guardar a *key* comunicada pelo cliente é o mesmo que o número de algarismos da mesma. Por outro lado, através da serialização de dados, o número de *bytes* utilizados na comunicação não vai variar, mais concretamente, para o *GET* 9 *bytes* (*char* com o valor inteiro da operação + *key* do registo) e para o *PUT* 1033 *bytes* (*char* com o valor inteiro da operação + *long long* com a *key* do registo + *string* com novo valor do registo).

- A procura de outros esquemas de alocação de *threads* aos clientes poderia ser interessante. A exploração de *thread pools*, em particular, poderiam ter alguns efeitos positivos pois escalaríam mais do que um número de *threads* igual ao número de clientes, uma vez que podem ser milhões. A isto associarmos *sockets assíncronos* e seguirmos um modelo de programação por eventos poderia levar a uma melhor utilização da mesma *thread pool*. Alternativamente, poderíamos envergar por uma exploração das *threads* verdes do C++ (*coroutines*), que em princípio escalariam melhor em termos de memória pela forma como diferem nas *threads* na utilização de uma *stack*, e que foram introduzidas no *standard* mais recente da linguagem, o C++20.

Estas são algumas das otimizações que seriam interessantes investigar no futuro. Em princípio, empregar algumas destas estratégias, ou mesmo todas, se possível, acabaria por ter um efeito bastante positivo na utilização de recursos e no débito do servidor.

3.1.3 Outras Considerações

Existem outros aspetos importantes para o bom funcionamento de uma base de dados que deveremos considerar para a sua implementação no mundo real que fogem um pouco à *scope* deste projeto, mas que achamos por bem mencionar nesta secção. A implementação de alguma forma de replicação, nomeadamente uma mistura de replicação passiva e ativa, como vemos em SGBDs como o *Postgres*, utilizando um protocolo de comunicação de grupo fiável ajudaria a construir um sistema que transparece para os clientes ser *always on*, mesmo quando uma das réplicas falha. Mais ainda, faria sentido termos cuidado com a ordenação dos pedidos, uma vez que estamos a trabalhar com um sistema assíncrono em que não sabemos com certeza qual a latência da rede.

Também seria importante termos alguma forma de proteger os dados que se encontram na base de dados, utilizando alguma forma de cifra para cifrar os dados, de modo

a que, caso haja uma *leak* dos conteúdos, os mesmos não servem de muito sem saber a cifra.

3.2 Cliente da Base de Dados

A implementação do cliente da base de dados é bem mais simples. É também em C++, e é apenas um programa *single threaded* que se conecta ao servidor através de um *socket* TCP e envia pedidos pelo mesmo.

Para os testes, foi utilizado um cliente que gera pedidos aleatórios (tanto a chave como o valor e o tipo) e envia-os até que o servidor pare de funcionar. Em particular, o tipo do pedido é gerado aleatoriamente com um rácio de 20 *GET* para 1 *PUT*, de forma a simular o rácio nas especificações dadas no enunciado do problema. Para testar com vários clientes em simultâneo, fazemos com que o programa do cliente corra em *background*, de modo a que o *script* que lança os clientes permita que sejam lançados vários sem esperarmos que um acabe.

Um cliente mais simples com uma interface pessoa-máquina também foi implementado, sendo essa a versão que foi inicialmente enviado para o professor. Esta versão funciona da mesma forma, mas tem uma interface para o cliente fazer pedidos à base de dados manualmente.

Cada pedido é bloqueante, ou seja, é necessário esperar pela resposta ao pedido antes de podermos enviar outro pedido. a utilização de *sockets* assíncronos juntamente com *completable futures* de modo a tornar os pedidos não bloqueantes poderia vir a ajudar no débito de um único cliente, mas a otimização do cliente não é o aspeto mais relevante deste trabalho.

4 Análise do Servidor

Para que possamos entender como o sistema se comporta num contexto do mundo real, especialmente para extrapolar o comportamento do mesmo num sistema de larga escala com milhares de pedidos por segundo, precisamos de entender como o sistema funciona num mundo muito mais pequeno e controlado. Para esse efeito, conduzimos testes para obter o perfil de execução do servidor.

4.1 Condições de Teste

Para testar o sistema, utilizamos o *cluster SeARCH6*[6] do Departamento de Informática da Universidade do Minho, especificamente o nodo *r641* do mesmo, pedindo sempre 1 *node* e 16 processos por nodo.

É de notar que, devido ao facto que o *cluster* da Universidade do Minho tem bastante tráfego, especialmente durante as horas da tarde, os testes foram feitos de noite, quando a rede de comunicação do *SeARCH* não é, em geral, tão concorrida, e poderemos ter o sistema mais próximo daquilo que seria uma infraestrutura dedicada apenas à utilização do servidor de base de dados. Obter as medições durante as horas da tarde mostrou resultados muito mais inconsistentes e *throughputs* consideravelmente piores.

4.2 Metodologia

Os testes foram feitos utilizando uma instância do servidor de base de dados, com um número variável de clientes. Este número está no conjunto {1,100,1000,2500}. Tentamos usar diferentes ordens de grandeza para o número de clientes ativos de uma só vez, sendo que o maior número que utilizamos foi 2500 devido ao limite que o *cluster SeARCH* impõe sobre o número de *sockets* ativos de uma só vez.

De modo a impormos uma condição de paragem para os testes, limitamos o envio de pedidos a um milhão por teste, sendo que, quando o servidor deteta que recebeu um milhão ou mais de pedidos, termina a sua execução, e o cliente, detetando que a sua conexão foi quebrada, também termina a sua execução, graciosamente.

No que toca à metodologia de instrumentação, esta é relativamente simples e direta: apenas fazemos uma injeção de código simples para medição de tempo utilizando as classes de C++ que melhor se adequam às medições, nomeadamente a classe *steady_clock*. Mais ainda, para obter os resulatdos que veremos mais à frente fazemos *sampling* em intervalos de um segundo numa das *threads*, nomeadamente na que corresponde ao primeiro cliente a ligar-se ao servidor. Periodicamente, a *thread* faz uma contangem do número de pedidos feitos desde a última medição e debita a informação necessária para o *standard output*. É de notar, como é claro, que esta medição é intrusiva e obriga essa *thread* a ocupar uma parte do seu tempo com medições, o que não é ideal mas, quando falamos de milhares e milhares de pedidos, acreditamos ser negligenciável, devido ao baixo número de instruções a mais que esta instrumentação acarreta, sendo ela apenas uma medição do relógio, uma escrita no *output* (que é admitidamente lenta, sem dúvida) e uma escrita de uma variável atómica. No entanto, como este processo não é contabilizado nas medições de tempos, não afetará os tempos de execução que veremos mais à frente.

O desenvolvimento do servidor requis um ciclo de otimização/medição de resultados, de modo a entendermos que direção deveríamos seguir para otimizar o servidor. Apenas as medições da versão final do sistema que criamos serão apresentadas aqui.

4.3 Débito

A primeira métrica que achamos importante retirar foi o débito de pedidos que o servidor consegue responder, por segundo. Esta métrica foi importante retirar de modo a podermos ter noção do comportamento de um único servidor, isolado, para com uma quantidade variada de clientes, o que nos dará uma ideia de qual deverá ser a cardinalidade ideal do grupo de comunicação a que cada instância do servidor pertenceria.

Para este efeito, testamos para um milhão de pedidos, com vários números de clientes, pelo que obtivemos os seguintes resultados:

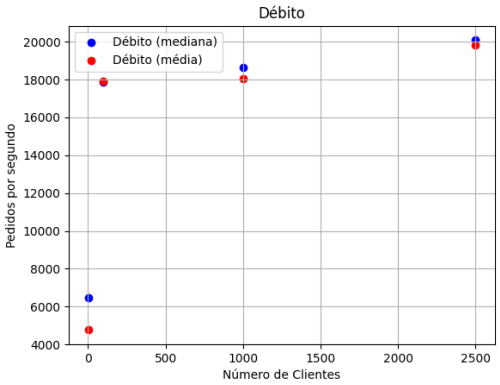

Figure 1: Débito do Servidor

A partir do gráfico da figura 1 podemos retirar algumas conclusões:

- Em primeiro lugar, podemos observar que o débito não aumenta linearmente com o número de pedidos por segundo. Isto seria de esperar devido ao elevado nível de contenção tanto sobre a estrutura de dados como sobre o ficheiro, que leva a que, quantos mais clientes existirem no sistema, mais tempo de espera sobre *locks* haverá. Mais ainda, todos os clientes são lançados pelo mesmo nodo (seria impensável atribuir a cada cliente um nodo ou até um *core* diferente) e partilham a largura de banda da rede, tanto uns com os outros, como com outros trabalhos que se encontrem no *cluster*, como referimos anteriormente;

- Apesar do aumento do débito ser cada vez menor, o maior número de clientes que lançamos provou ainda ter alguns ganhos em termos de débito, o que nos leva a crer que existe ainda um limite de clientes por servidor que não atingimos nos testes a partir do qual teremos *diminishing returns*, que seria o número ideal de clientes a estabelecer como limite de clientes por servidor.

A partir desta métrica já começa a surgir a seguinte ideia: ao estimarmos um valor para o custo do *hardware* necessário para manter este servidor em funcionamento, teremos que fazer uma estimativa pessimista. Tendo em conta que o ambiente de teste não é perfeito e altamente concorrido, e que existem certas otimizações que teriam de ser implementadas antes de fazer *deployment* do sistema, cujos efeitos produzidos apenas podemos teorizar, não podemos fazer uma estimativa exata de como seria o sistema real, mas sim apenas uma estimativa altamente pessimista.

4.4 Tempo de Computação no Servidor

Nesta secção iremos analisar o comportamento do servidor ao nível do tempo de execução dos pedidos que o mesmo recebe, distinguindo primeiro para cada um dos tipos e depois fazendo uma observação do tempo global dos pedidos.

4.4.1 *PUT*

Os *PUT*s são os pedidos mais sensíveis e que, em teoria, deverão produzir os piores tempos de execução, devido ao facto que precisam de escrever um KiB em memória em todos os acessos ao ficheiro. Mais ainda, são aqueles que obrigam absolutamente a utilização de *locks* de modo a não haver escritas concorrentes em posições iguais que levariam a comportamentos indefinidos.

Tanto para os *PUT*s como eventualmente para os *GET*s podemos fazer a distinção entre pedidos que vão ao ficheiro e os pedidos que não vão, sendo que podemos apresentar de seguida os valores que nós registamos:

	Memória		**Disco**		**Global**	
clientes	**Média**	**Mediana**	**Média**	**Mediana**	**Média**	**Mediana**
1	0.003	0.002	0.115	0.017	0.104	0.016
100	0.011	0.010	4.590	4.778	4.131	4.301
1000	0.012	0.010	47.458	42.467	42.713	38.221
2500	0.013	0.010	116.177	94.467	104.561	85.021

Table 1: Tempo de Execução de Pedidos do Tipo *PUT* (ms)

Analisando a tabela, podemos ver que os tempos estão maioritariamente dentro do esperado e querido, que é 1 milissegundo por pedido. No entanto, os *PUT*s que fazem acesso ao disco são bastante mais lentos, sendo que os tempos são maiores do que 1 ms por bastante. Isto é esperado visto que acessos a memória externa (neste caso, o disco) são mais caros em termos de tempo. Adicionando a isto o facto que temos de controlar acessos ao ficheiro com *locks*, é normal que isto aconteça e que tenda a piorar com o número de clientes a fazer pedidos em simultâneo. Este efeito será ainda mais sentido quanto menor for a percentagem de dados em RAM em proporção aos em disco.

Como os pedidos ao ficheiro serão em média 90% dos pedidos totais no nosso sistema, a latência está acima do suposto para estas quantidades de pedidos.

4.4.2 *GET*

No que toca aos pedidos da classe *GET*, é esperado que a sua *performance* seja ligeiramente melhor que os do tipo *PUT*, especialmente no acesso ao ficheiro, visto que, em princípio, operações de leitura deverão ser mais leves do que de escrita. Assim sendo, apresentamos os resulatados obtidos para os pedidos deste tipo na seguinte tabela:

	Memória		**Disco**		**Global**	
clientes	**Média**	**Mediana**	**Média**	**Mediana**	**Média**	**Mediana**
1	0.002	0.002	0.073	0.010	0.066	0.009
100	0.006	0.005	4.535	4.715	4.082	4.244
1000	0.005	0.005	47.079	42.515	42.372	38.264
2500	0.022	0.005	118.023	108.453	106.223	97.608

Table 2: Tempo de Execução de Pedidos do Tipo *GET* (ms)

Como podemos observar, o comportamento dos pedidos do tipo *GET* são algo semelhantes ao dos *PUT*s, sendo que, tal como o anterior, a latência média e mediana dos pedidos está bem acima do desejado.

4.4.3 Global

Olhando agora para o panorama geral da latência dos pedidos, como é que esta se comporta?

Olhando para a seguinte tabela:

	PUT		**GET**		**Total**	
clientes	**Média**	**Mediana**	**Média**	**Mediana**	**Média**	**Mediana**
1	0.104	0.016	0.066	0.009	0.068	0.009
100	4.131	4.301	4.082	4.244	4.084	4.248
1000	42.713	38.221	42.372	38.264	42.388	38.262
2500	104.561	85.021	106.223	97.608	106.144	97.009

Table 3: Tempo Global de Execução de Pedidos (ms)

O padrão que primeiro se destaca é que os tempos de execução são maioritariamente ditados pelos *GET*s, o que é de esperar, de certa forma, quando temos em conta que estes são vinte vezes mais comuns dos que os *PUT*s.

Os tempos globais estão muito aquém do que é esperado, pelo que o grande *bottleneck* está, sem dúvida, no problema do controlo de concorrência. Dentro do servidor, na realidade, só dois pedidos podem ser executados em paralelo. Isto deve-se ao facto que apenas fazemos distinção de duas zonas críticas disjuntas pela forma como estamos a gerir os *locks* (o ficheiro e a estrutura de dados), pelo que o nível de paralelismo é muito baixo. Aqui se revela a verdadeira utilidade das otimizações que mencionamos quanto à partição do ficheiro e da utilização de *locks* sobre as entradas da estrutura de dados e não apenas da estrutura em si. Ao dividirmos as duas zonas críticas em várias de grão mais fino podemos aproveitar melhor o paralelismo e temos a possibilidade de realizar várias dezenas ou até centenas de tarefas em paralelo. Estas tarefas não são computacionalmente intensivas, pelo que não iria escalar da melhor forma, mas seria esperado um ganho significativo, que nós cremos que poderia levar o tempo de execução à *threshold* pedida.

No entanto, apesar de aumentar o número de *cores* poder vir a melhorar a *performance* do programa, não achamos que seja boa ideia aumentar devido ao facto que pagaremos por cada *core* que utilizemos e queremos diminuir esse custo, pelo que manter o número de *cores* por instância a 16, ou até a menos, será imperativo.

4.5 *Overhead* Comunicacional

Na rede irão circular dois tipos de mensagens: mensagens de *request* do cliente para o servidor e mensagens de *reply* do servidor para o cliente. Ambas as mensagens são apresentadas sob a forma de uma *string*, não sendo serializadas num formato binário.

- A mensagem de *request* conterá o tipo de pedido (1 *byte*), a chave do pedido (um *long long* sob a forma de uma *string*, podendo conter até 20 *bytes*) e o valor do registo (1024 *bytes*), mais um espaço a separar os vários campos, e um "\n" no final da mensagem, para um máximo de 1048 *bytes* por mensagem de *request*.
- A mensagem de *reply* de um *GET* deverá vir o registo pedido, se existir, que contém o valor (1024 *bytes*) e o "\n" no final, para um máximo de 1025 *bytes*. Caso a chave não seja encontrada, ou o pedido seja um *PUT*, apenas é retornado um caracter (0 caso o pedido tenha falhado e 1 caso tenha tido sucesso).

Assumindo que a maior parte das mensagens de *reply* a um *GET* não irão incorrer em *miss* e vão retornar um registo, e sendo que estas são vinte vezes mais comuns que as de *reply* a *PUT*s, podemos chegar à largura de banda necessária para responder aos pedidos com a seguinte fórmula:

$$Bandwidth_{avg} = \frac{([\frac{20}{21} \times (1048+1025) + \frac{1}{21} \times (1048+1)] \times deb_{avg})}{1024^2} MiB/s$$

Tudo isto ignorando *headers* de protocolos de comunicação. Assim sendo, concluímos que estamos a utilizar a seguinte largura de banda no sistema atual, tendo em conta os débitos médios que apresentamos:

clientes	débito (pedidos/s)	largura de banda (MiB/s)
1	4759.93	9.189
100	17907.08	34.569
1000	18068.02	34.880
2500	19814.64	38.251

Table 4: Largura de Banda Utilizada, em média

A partir disto vemos que o sistema como foi implementado não é muito pesado, sendo que vários milhares de pedidos requerem apenas alguns *mebibytes* por segundo de largura de banda. No entanto, se as mensagens fossem enviadas num formato binário mais compacto, a largura de banda necessária poderia descer.

Aplicando agora a igualdade a que chegamos ao débito esperado de 10 000 000 de *GET*s mais 500 000 *PUT*s por segundo, chegamos à seguinte conclusão:

clientes	débito (pedidos/s)	largura de banda (MiB/s)
1000 - 100 000	10 500 000	20 269.2

Table 5: Largura de Banda Necessária, em média

Como é óbvio, o sistema real necessitará de uma largura de banda muito mais elevada de aproximadamente 20 GiB/s.

4.6 Memória e *Storage*

De modo a escalar o sistema para acarretar a quantidade de memória necessária para corresponder aos requisitos, teremos de ter o seguinte em consideração:

- Uma entrada em ficheiro contém 1048 *bytes* (como vimos anteriormente);
- Uma entrada em memória contém a chave (um *long long*, 8 *bytes*) e o valor (1024 *bytes*) para um total de 1032 *bytes*.

Assim sendo, teremos de ter acesso a alguma forma de *storage* em disco que seja capaz de armazenar 100TB de registos, que correspondem a aproximadamente a 95 000 000 000 de entradas.

Mais importante será como poderemos escalar os registos mantidos em memória rápida. Como dissemos, no nosso sistema temos apenas 100 MiB de registos em memória rápida, aproximadamente 101 000 registos. este valor é suficiente quando falamos de uma a base de dados que é relativamente pequena, mas fará sentido aumentar este número razoavelmente. Em proporção, este número corresponde a 10% da base de dados como ela está agora, mas apenas 0.0001% do sistema real. Na nossa ótica, para que os pedidos possam ser respondidos com alguma rapidez, guardando os valores mais populares em memória (utilizando o esquema de substituição por LRU, como já referimos), deverá ter pelo menos 1% dos registos em memória rápida.

Assim, o sistema precisaria de ter registos em RAM, o que é equivalente a aproxiamdamente 950 000 000 registos, o que equivale a 918 GiB em memória rápida.

Isto não é um número completamente impensável mas é bastante elevado, pelo que poderíamos diminuir para menos que isso, mas isso seria em detrimento da *performance* do programa. Diminuir para metade disso, ficando com 459GiB em memória rápida, o que equivale a 0.5% dos registos em memória rápida poderá ser tido em consideração. É muito importante frisar, no entanto, que este valor é realmente bem mais baixo e iria diminuir o débito de pedidos. No entanto, a melhor gestão dos acessos a ficheiro com as otimizações futuras que mencionamos (nomeadamente com o *sharding* do ficheiro e com o uso de *MPI IO*) é possível que o débito não fosse muito afetado por estas mudanças.

5 Implementação num Sistema Real

Nesta secção final, iremos explicar como teríamos de implementar o sistema no mundo real de modo a corresponder aos requisitos impostos no enunciado.

5.1 Requisitos Mínimos

Tendo em conta os requisitos que foram impostos no enunciado do trabalho, e que foram expostos no início deste documento, agregamos aqui as conclusões a que chegamos em relação às necessidades do sistema:

- *Storage*: mínimo de 100 TB, possível e provavelmente mais;
- RAM: mínimo de 918 GiB, assumindo 1% dos registos em memória;
- Rede: largura de banda de 21GiB/s;

Tendo em conta tudo o que referimos antes, acreditamos que uma infraestrutura que cumpra estes requisitos será suficiente para acarretar a quantidade de pedidos esperada, especialmente quando temos em consideração as otimizações extra que não pudemos fazer, mas que seriam obrigatórias para o correto *deployment* do sistema.

Outras considerações a ter seriam o número de instâncias do servidor que deveremos ter ativas. Uma vez que cada as instâncias que experimentamos operavam com 16 *cores*, achamos que poderemos escalar com esse número de *cores* por instância, sendo que o número vai variar consoante aquilo que tivermos disponível, mas será necessário ter mais do que uma instância sem dúvida. Isto não implica necessariamente ter mais do que 918 GiB de memória para guardar os registos, implica que cada uma das instâncias do servidor partilhará essa memória com os outros, sendo que cada um terá acesso a $\frac{918}{\#servidores}$GiB para guardar os registos, e partilharão *storage*.

Isto leva-nos ao seguinte requisito:

- *Cores*: 16 *cores*/instância de servidor.

A partir disto, podemos procurar uma infraestrutura que nos forneça as ferramentas necessárias para implementar este servidor de base de dados.

5.2 Infraestrutura Escolhida

De modo a simplificar a análise da infraestrutura, iremos utilizar as *shapes* da *Oracle Cloud Infrastructure*[4], proposta pelo professor.

Com base na oferta, optamos por escolher uma ***Bare Metal Dense I/O Shape***, mais especificamente a *shape* ***BM.DenseIO2.52***. As características da *shape* são as seguintes:

Shape	BM.DenseIO2.52
Processador	Intel Xeon Platinum 8167M [3]
OCPU	52
Memória (GiB)	725
Disco Local	51.2 TB NVMe SSD (8 drives)
Largura de Banda Máxima (GiB/s)	2×25
Frequência Base	2.0 GHz
Frequência Máxima	2.4 GHz

Table 6: Especificação da *Shape*

Esta *shape* foi escolhida por apresentar as seguintes características:

- Em primeiro lugar, apresenta uma elevada largura de banda, sendo que uma instância da *shape* já alberga a largura de banda mínima necessária;
- A sua *storage* é das maiores e mais rápidas disponíveis numa só *shape*, o que faz sentido, visto que a *shape* está desenhada especificamente para bases de dados em larga escala, de acordo com o fornecedor;
- A memória RAM que apresenta também é elevadíssima e pode albergar uma boa parte daquilo que necessitamos.

Tendo tudo isto em conta, podemos ver que esta é a *shape* que melhor se adequa ao sistema que queremos implementar. No entanto, uma instância só da *shape* não será suficiente, devido ao facto que apenas apresenta cerca de metade da *storage* necessária para albergar o sistema, pelo que precisaremos de duas instâncias. Isto obviamente acarretará um custo muito mais elevado, mas é necessário, e transforma o sistema num com:

- 104 *cores*;
- 1450 GiB de memória RAM;
- 102.4 TB de memória SSD;
- 4×25 GiB/s de Largura de Banda;

Estas características são suficientes ou, em alguns aspetos, excedem as necessidades do sistema. De modo a podermos fazer uma proporção do necessário com o

disponível, o sistema implementado terá características diferentes das planeadas de modo a podermos utilizar o máximo de recursos por que estaremos a pagar.

Em primeiro lugar, poderemos ter 6 instâncias do servidor ativas, cada uma com 16 *cores* e 1 instância apenas com 6 *cores*. Uma vez que esta terá menos capacidade computacional que as outras, poderá ser utilizada apenas como *backup* no grupo de comunicação dos *servidores*, que apenas é ativada no caso de uma das instâncias falhar. Como a memória que temos disponível é 1450GiB e não os 918GiB necessários, cada servidor terá acesso a $\frac{1450}{7}GiB = 207GiB$, que poderemos aproximar a 200 GiB. Tendo em conta que no grupo de comunicação a base de dados estará *sharded* e cada instância acederá a uma parte da base de dados e não a toda, 200GiB deverão ser necessários para manter a *performance* num nível aceitável, sendo que teremos 1.45% da base de dados em memória rápida, bem mais do que o 1% considerado inicialmente.

No que toca ao número de pedidos que podem circular na rede, nós temos acesso a uma largura de banda cinco vezes maior do que o necessário estimado, pelo que o número de pedidos será indubitavelmente correspondido, e até poderá escalar para um número maior, se necessário.

A *storage* é um aspeto em que o pedido está mais próximo do real, em que a infraestrutura que pedimos está bastante próxima do necessário, apenas com pouco mais de 2 TB extra. Escalar a base de dados de modo a precisarmos de mais *storage* obrigar-nos-ia a recorrer ao aluguer de um disco externo, o que iria acarretar um *penalty* em termos de *performance* e monetário.

Tendo tudo isto em conta, o sistema conseguiria utilizar bem os recursos a que tem acesso, e, mais ainda, corresponder aos requisitos que nos foram impostos no enunciado.

5.3 Custo Total Estimado

Quanto ao custo total do servidor, teremos de ter em conta o custo de duas *shapes BM.DenseIO2.52*, cada uma com 52 *cores*.

De acordo com a tabela de preços dada pela *Oracle*[2], o sistema acarreta o seguinte custo:

Produto	**Preço Unitário (€)**	**Métrica**
Compute - Bare Metal Dense I/O - X7	0.11450775	OCPU/hora

Table 7: Tabela de Preços da *Shape* pedida

Tendo tudo isto em conta, tendo em conta os preços disponíveis no dia 28 de maio de 2021 e a taxa de conversão do Dollar para Euro na mesma data, temos que o preço de manutenção anual do sistema será:

$$0.11450775 \times 2 \times 52 \times 365 \times 24 = \textbf{104321€}$$

Este é, assumidamente, pelo menos na nossa ótica, um valor algo elevado, mas que fará sentido quando falamos de um serviço em larga escala que deverá ser fiável e que lidará com quantidades enormes de dados e de transações a cada minuto.

Para a utilização do sistema será necessário um sistema operativo. No entanto, este é fornecido pela *Oracle*, sendo que podemos escolher entre várias distribuições do *Linux* (*Oracle Linux*, *CentOS* ou *Ubuntu*), sendo todas estas gratuitas, ou uma distribuição de servidor do *Windows*, que teria um custo acrescido. Por esta razão, escolheríamos qualquer uma das três distribuições do *Linux*, sendo que optaríamos por *Ubuntu*, devido ao facto que o *CentOS* deixou de receber *updates* desde dezembro de 2020[1] e que não estamos muito familiarizados com a distribuição do *Linux* da *Oracle*.

6 Conclusão

Em conclusão, acreditamos que atingimos os objetivos principais deste trabalho. Conseguimos desenvolver um sistema que responda à maior parte dos requisitos por si, e entendemos como poderíamos explicar a forma como o mesmo se comporta através de técnicas de instrumentação abordadas nas aulas.

O ponto em que admitimos que o nosso trabalho falha é na latência dos pedidos e na falta de implementações que nós conhecemos e descrevemos, que acreditamos que teriam um efeito genuinamente positivo no projeto e na projeção de custo do sistema.

Não obstante, ficamos contentes com o resultado final e cremos ter explicado devidamente o comportamento do sistema que nós próprios criamos.

7 Referências

[1] *CentOS Linux is dead—and Red Hat says Stream is "not a replacement"*. URL: https://arstechnica.com/gadgets/2020/12/centos-shifts-from-red-hat-unbranded-to-red-hat-beta/.

[2] *Compute Pricing*. URL: https://www.oracle.com/pt/cloud/compute/pricing.html.

[3] *Intel® Xeon® Platinum 8160M Processor*. URL: https://ark.intel.com/content/www/us/en/ark/products/120502/intel-xeon-platinum-8160m-processor-33m-cache-2-10-ghz.html.

[4] *OCI Compute Shapes*. URL: https://docs.oracle.com/en-us/iaas/Content/Compute/References/computeshapes.htm.

[5] *Parallel I/O*. URL: https://www.tacc.utexas.edu/documents/13601/900558/MPI-IO-Final.pdf/eea9d7d3-4b81-471c-b244-41498070e35d.

[6] *SeARCH Cluster*. URL: http://search6.di.uminho.pt/wordpress/.