

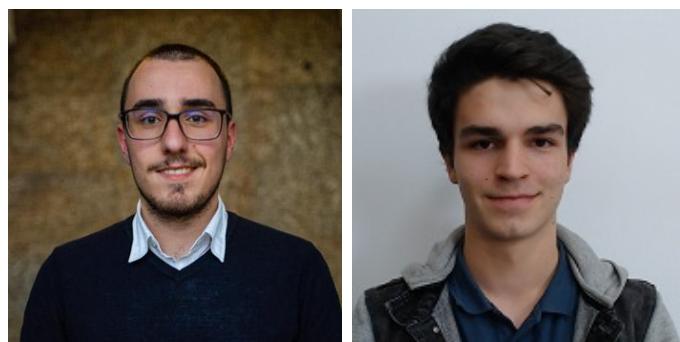
UNIVERSIDADE DO MINHO
MESTRADO INTEGRADO EM ENGENHARIA
INFORMÁTICA

Análise de Aplicações Big Data

Grupo 4

Diogo Pinto Ribeiro, A84442

Rui Filipe Moreira Mendes, A83712



Laboratórios de Engenharia Informática
4º Ano, 2º Semestre
Departamento de Informática

17 de junho de 2021

Índice

1	Introdução	1
1.1	Contextualização	1
1.2	Objetivos	1
2	Desenho	3
2.1	<i>System Calls</i>	3
2.1.1	mknod	3
2.1.2	access	3
2.1.3	open	3
2.1.4	read	4
2.1.5	pread	4
2.1.6	write	4
2.1.7	pwrite	4
2.1.8	truncate	4
2.1.9	lseek	5
2.1.10	fsync	5
2.2	Arquitetura	5
3	Implementação	7
3.1	Tecnologias	7
3.2	Sistema FUSE	8
3.3	Servidor	8
3.4	Elasticsearch e Kibana	10
3.5	Otimizações	13
4	Análise	14
4.1	<i>Benchmarks</i>	14
4.1.1	RocksDB	14
4.1.2	PostgreSQL	18
4.2	Análise de Resultados	20
4.2.1	RocksDB	20
4.2.2	PostgreSQL	34

5 Conclusões	38
5.1 Balanço	38
5.2 Trabalhos Futuros	39

Lista de Figuras

1	Desenho do Sistema	6
2	Arquitetura do Sistema	7
3	Total, proporção e duração média das <i>system calls</i>	11
4	10 caminhos mais acedidos	11
5	Métricas mistas sobre operações	12
6	Métricas instantâneas do Metricbeat	12
7	Métricas temporais sobre CPU, memória e disco	12
8	Métricas relativas à rede e processos	13
9	Comparação dos resultados em operações por segundo	16
10	Comparação dos tempos de indexação	17
11	Escritas Sequenciais - Tipo e Duração das <i>system calls</i>	21
12	Escritas Sequenciais - Tamanho médio em <i>bytes</i> escrito e lido	21
13	Escritas Sequenciais - Caminhos mais acedidos	22
14	Escritas Sequenciais - Linha temporal dos caminhos acedidos	22
15	Escritas Sequenciais - Linha temporal da utilização de recursos	23
16	Escritas Sequenciais - Utilização de recursos por processos	23
17	Escritas Aleatórias - Tipo e Duração das <i>system calls</i>	24
18	Escritas Aleatórias - Tamanho médio em <i>bytes</i> escrito e lido	25
19	Escritas Aleatórias - Caminhos mais acedidos	25
20	Escritas Aleatórias - Linha temporal dos caminhos acedidos	25
21	Escritas Aleatórias - Linha temporal da utilização de recursos	26
22	Escritas Aleatórias - Utilização de recursos por processos	27
23	Leituras Sequenciais - Tipo e Duração das <i>system calls</i>	27
24	Leituras Sequenciais - Tamanho médio em <i>bytes</i> escrito e lido	28
25	Leituras Sequenciais - Caminhos mais acedidos	29
26	Leituras Sequenciais - Linha temporal dos caminhos acedidos	29
27	Leituras Sequenciais - Linha temporal da utilização de recursos	30
28	Leituras Sequenciais - Utilização de recursos por processos	30
29	Leituras Aleatórias - Tipo e Duração das <i>system calls</i>	31
30	Leituras Aleatórias - Tamanho médio em <i>bytes</i> escrito e lido	31
31	Leituras Aleatórias - Caminhos mais acedidos	32

32	Leituras Aleatórias - Linha temporal dos caminhos acedidos	32
33	Leituras Aleatórias - Linha temporal da utilização de recursos	33
34	Leituras Aleatórias - Utilização de recursos por processos	33
35	TPC-C - Tipo e Duração das <i>system calls</i>	34
36	TPC-C - Tamanho médio em <i>bytes</i> escrito e lido	35
37	TPC-C - Caminhos mais acedidos	36
38	TPC-C - Linha temporal dos caminhos acedidos	36
39	TPC-C - Linha temporal da utilização de recursos	37
40	TPC-C - Utilização de recursos por processos	37

Lista de Tabelas

1	Resultados do db_bench para o sistema FUSE puro	15
2	Resultados do db_bench para a versão não otimizada	15
3	Resultados do db_bench para a versão otimizada	15
4	Resultados dos tempos de indexação	17
5	Resultados do TPC-C para o sistema FUSE puro	19
6	Resultados do TPC-C para a versão otimizada	19

1 Introdução

No presente relatório iremos detalhar o desenvolvimento e análise de um sistema de *tracing* para aplicações *Big Data* realizado no âmbito da Unidade Curricular de Laboratórios de Engenharia Informática.

1.1 Contextualização

Um sistema de *tracing* consiste num sistema capaz de efetuar registos(*logs*) da execução de um determinado programa com o objetivo de posteriormente poderem ser utilizados para perceber eventuais falhas de performance ou padrões na execução do programa.

A análise do resultado de um sistema de *tracing* pode ser feita após a execução de um programa, ou em tempo real. O nosso foco neste projeto serão as análises em tempo real, dado que estas apresentam algumas características interessantes.

Apesar de existir um acréscimo de processamento durante a execução do programa, sermos capazes de analisar o comportamento de uma aplicação em tempo real é de extrema utilidade. Não ter de esperar pelo fim da execução de um programa, ou não ter de efetuar interrupções para posteriormente analisar os dados é uma mais valia para qualquer sistema de *tracing*.

Os principais desafios na construção de um sistema de *tracing* em tempo real passam por sermos capazes de conjugar a recolha e análise de dados em tempo real, com o acréscimo de processamento necessário, que poderá tornar as aplicações que correm sobre o nosso sistema desnecessariamente lentas. Outro desafio inerente é o fator tempo real. Será sempre um desafio conseguir recolher os dados, processar os mesmos, e exibir o conteúdo, tendo o mínimo de desfazamento temporal possível.

1.2 Objetivos

Os nossos objetivos com a criação deste sistema passam pelo desenvolvimento de um sistema de *tracing* em tempo real, baseado em FUSE, que nos permita coletar as *system calls* efetuadas por aplicações, nomeadamente aplicações *Big*

Data. Estas textitsystem calls serão guardadas utilizando as ferramenta Elastic-Search e Kibana de forma a permitir uma análise mais detalhada. Tal análise é fundamental para perceber potenciais problemas de desempenho e funcionamento destas aplicações. Outros dos nossos objetivos passa por efetuar a análise em tempo real com o menos desfasamento temporal possível, sendo que considerámos que poderá ser um dos objetivos mais difíceis de cumprir. É também um objetivo nosso que o sistema a desenvolver cause o menor impacto possível nas aplicações que sobre ele correm.

2 Desenho

Neste capítulo iremos apresentar o processo de desenho do nosso sistema de *tracing*, detalhando as decisões por nós tomadas.

2.1 *System Calls*

O primeiro passo no desenho do sistema consistiu em definir quais as *system calls* que pretendemos analisar. Dado que estamos a utilizar um sistema de ficheiros FUSE como base para o nosso trabalho, optámos por selecionar um subconjunto das operações disponibilizadas.

De seguida são apresentadas as *system calls* por nós selecionadas bem como os respetivos valores que pretendemos armazenar para cada uma delas.

2.1.1 mknod

Para a *system call mknod* consideramos que seria relevante guardar o seu **tipo**(mknod), o **pid** do processo que a invoca, o **timestamp** de quando foi invocada, a sua **duração**, o **código de erro** retornado, o **caminho** associado, e o **modo de criação**.

2.1.2 access

Para a *system call access* consideramos que seria relevante guardar o seu **tipo**(access), o **pid** do processo que a invoca, o **timestamp** de quando foi invocada, a sua **duração**, o **código de erro** retornado, o **caminho** associado, e o **modo de acesso**.

2.1.3 open

Para a *system call open* optamos por guardar o seu **tipo**(open), o **pid** do processo que a invoca, o **timestamp** de quando foi invocada, a sua **duração**, o **código de erro** retornado, o **caminho** associado, e as respetivas **flags**.

2.1.4 read

Para a *system call* **read**, optamos por guardar o seu **tipo**(read), o **pid** do processo que a invoca, o **timestamp** de quando foi invocada, a sua **duração**, o **código de erro** retornado, o **caminho** associado, e a quantidade de **bytes lidos**.

2.1.5 pread

Para a *system call* **pread**, optamos por guardar o seu **tipo**(pread), o **pid** do processo que a invoca, o **timestamp** de quando foi invocada, a sua **duração**, o **código de erro** retornado, o **caminho** associado, a quantidade de **bytes lidos** e o **offset** associado.

2.1.6 write

Para a *system call* **write**, de forma semelhante ao *read*, optamos por guardar o seu **tipo**(write), o **pid** do processo que a invoca, o **timestamp** de quando foi invocada, a sua **duração**, o **código de erro** retornado, o **caminho** associado, e a quantidade de **bytes escritos**.

2.1.7 pwrite

Para a *system call* **pwrite**, de forma semelhante ao *pread*, optamos por guardar o seu **tipo**(pwrite), o **pid** do processo que a invoca, o **timestamp** de quando foi invocada, a sua **duração**, o **código de erro** retornado, o **caminho** associado, a quantidade de **bytes escritos** e o **offset** associado.

2.1.8 truncate

Para a *system call* **truncate**, optamos por guardar o seu **tipo**(truncate), o **pid** do processo que a invoca, o **timestamp** de quando foi invocada, a sua **duração**, o **código de erro** retornado, o **caminho** associado, e o **novo tamanho** utilizado pela operação.

2.1.9 lseek

Para a *system call* **lseek**, optamos por guardar o seu **tipo**(lseek), o **pid** do processo que a invoca, o **timestamp** de quando foi invocada, a sua **duração**, o **código de erro** retornado, o **descritor do ficheiro** associado, e o **offset** associado à operação.

2.1.10 fsync

Para a *system call* **lseek**, optamos por guardar o seu **tipo**(fsync), o **pid** do processo que a invoca, o **timestamp** de quando foi invocada, a sua **duração**, o **código de erro** retornado, e o **descritor do ficheiro** associado.

O conjunto de *system calls* apresentado representa uma iteração inicial do sistema, sendo que em versões futuras seria interessante extender este conjunto de operações de modo a aumentar a quantidade de dados analisados e assim melhorar a qualidade das análises efetuadas posteriormente.

2.2 Arquitetura

Antes mesmo de avançar para a implementação, foi necessário entender quais os componentes que iremos necessitar para a construção do sistema, bem como entender a forma como se interligam. Nesta fase do projeto não é o nosso foco especificar tecnologias, mas sim moldar o sistema.

Dado que o nosso principal foco consiste em captar *system calls* e criar registos acerca dos mesmos, entendemos que será necessário criar ou pesquisar acerca de algum tipo de sistema de ficheiros que nos permita interceptar *system calls*. Além disso, é necessário armazenar os registos gerados, sendo que faz sentido a utilização de algum sistema de armazenamento, mais concretamente, sob a forma de uma base de dados. Será também importante considerar a comunicação entre o sistema de ficheiros e o sistema de armazenamento, dado que questões de performance serão críticas. Por fim, iremos necessitar de alguma ferramenta para a visualização dos dados armazenados. Para esse efeito é do nosso interesse a utilização de uma ferramenta que não se limite a listar dados, mas que forneça mecanismos para

correlacionar os mesmos e se possível, permita gerar gráficos e outros elementos semelhantes, de modo a facilitar a análise dos dados.

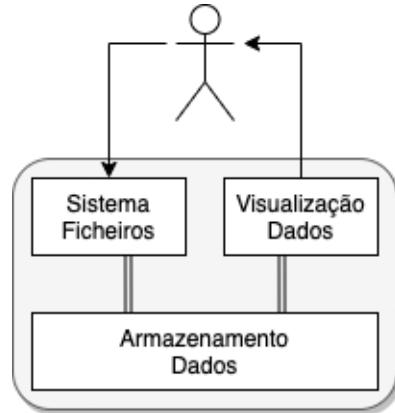


Figura 1: Desenho do Sistema

Tal como podemos verificar na Figura 1, trata-se de uma arquitetura relativamente simples e com poucas componentes. Apesar da sua simplicidade, podemos desde já constatar que os canais de comunicação entre o sistema de ficheiros e o armazenamento, entre o sistema de armazenamento e a componente de visualização de dados devem ser rápidos e fiáveis. Do mesmo modo, a existência de um número reduzido de componentes não replicados também nos pode indicar que cada um poderá estar sujeito a uma carga de trabalho elevada e ser um gargalo no sistema. Estas considerações serão tidas em conta na fase de implementação do sistema que iremos ver de seguida.

3 Implementação

Neste capítulo iremos apresentar o processo de desenvolvimento do nosso sistema de *tracing*, bem como melhorias e alterações que considerámos relevantes.

3.1 Tecnologias

Antes de iniciarmos o processo de implementação é necessário definir quais as tecnologias que iremos utilizar em cada componente do sistema.

Para o sistema de ficheiros iremos utilizar o **FUSE** (*Filesystem in Userspace*). Mais concretamente, optámos pela implementação *libfuse* que é a implementação de referência para comunicar com o módulo FUSE do *kernel*. Com o FUSE torna-se possível interceptar as *system calls* efetuadas ao sistema operativo, sendo essa uma funcionalidade por nós desejada.

Para o armazenamento de dados iremos utilizar o **Elasticsearch**. O Elasticsearch torna-se uma ferramenta interessante dado que nos permite armazenar dados, e complementar o serviço com ferramentas analíticas, nomeadamente o **Kibana**. Desse modo podemos armazenar e visualizar dados simultâneamente.

Após definirmos quais as *system calls* que iremos recolher, após desenharmos a arquitetura que pretendemos implementar, e após escolhermos as tecnologias a utilizar, focamos a nossa atenção em aspectos mais técnicos, nomeadamente a interligação entre um sistema de ficheiros em FUSE e o Elasticsearch, procurando soluções existentes e quais as restrições inerentes.

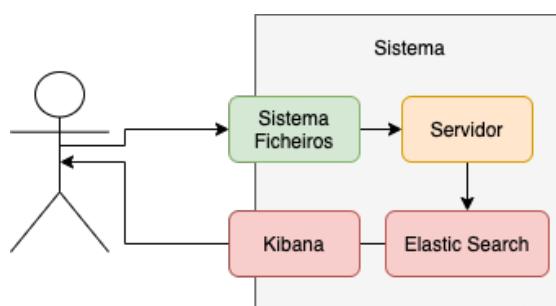


Figura 2: Arquitetura do Sistema

Uma das principais restrições existentes na nossa arquitetura consiste na ausência de um cliente Elasticsearch em C. Como a implementação FUSE por nós utilizada foi desenvolvida em C, surgiu a necessidade da criação de um servidor intermediário entre o sistema de ficheiros e o Elasticsearch.

Na Figura 2 podemos ver a tradução do desenho do nosso sistema para uma arquitetura que inclui as tecnologias por nós escolhidas e o efeito que estas tiveram no desenho original.

3.2 Sistema FUSE

Para sermos capazes de capturar informação sobre as *system calls* precisamos de implementar então um sistema que fosse capaz de as intercetar e nos permitisse ir guardando informação que achamos relevante.

Na implementação optamos por começar com um exemplo que já é fornecido pelos desenvolvedores do **FUSE** no repositório da implementação, o *passthrough*.[1] O *passthrough* apresenta suporte para lidar com várias operações e mostra de uma forma muito simplista como é que procedemos à realização efetiva das operações depois de as observarmos. Isto foi algo que auxiliou o nosso trabalho visto que oferecia uma base de trabalho pela qual começar.

Uma vez estudado o exemplo do *passthrough* a implementação do sistema ficou mais simples e começamos por fazer a ligação entre o sistema **FUSE** e o servidor através de *sockets TCP* o que permitiu enviar a informação relevante captada para processamento posterior.

Posteriormente para cada operação analisada implementamos o formato dos dados a enviar, que se encontra especificado na secção 2.1 do relatório. Os campos por nós selecionados são agrupados numa string, e separados por espaços, sendo posteriormente enviados ao Servidor.

3.3 Servidor

Com o intuito de servir como intermediário entre o sistema de ficheiros e o Elasticsearch, foi criado um servidor em **Python**. Optámos por esta linguagem

dado que esta possui um cliente para o Elasticsearch e por nos permitir rapidamente obter uma solução.

Este servidor tem como principal função receber conteúdo por parte do sistema de ficheiros, efetuar as alterações necessárias, e inserir documentos num Índice. Inicialmente considerámos a utilização do gRPC dado que é uma solução bastante eficiente e nos permitiria poupar tempo. No entanto, esta também não possui uma suporte para a linguagem C. Com esta restrição em mente, implementámos um protocolo de comunicação simples sobre **canais TCP**. A utilização de canais de comunicação TCP deveu-se principalmente a estes serem fiáveis e garantirem ordem na entrega.[2]

O servidor utiliza os canais de comunicação TCP para ler o conteúdo que lhe é enviado e processar o mesmo. Para esse efeito é utilizado o campo **tipo** presente em todas as *system calls*. Após a construção do objeto a inserir no Elasticsearch, é invocado o cliente para Python.

Após implementarmos o servidor, rapidamente percebemos que existiam algumas melhorias importantes. A principal melhoria que implementamos consiste na alteração do tipo de execução do programa. Até agora o servidor executava sequencialmente, o que trazia bastantes atrasos e é ineficiente. Para reverter a situação, recorremos a um **ThreadPoolExecutor** que nos permite alocar um determinado número de *threads* para executarem funções assincronamente. Deste modo, o servidor pode seguir a sua execução enquanto um documento é inserido no Elasticsearch, ficando a *thread* responsável pela inserção disponível após terminar a operação.[3]

No servidor existem 2 parâmetros de configuração que podem ser alterados: o **número de threads** no *ThreadPoolExecutor*, e o **tamanho do buffer** usado na leitura dos *sockets* TCP.

Após alguns testes, chegámos a uma conclusão acerca de quais os valores ótimos a ser usados na configuração do sistema. A configuração ótima consistem em **16 threads** e **128 KB(131072 bytes)** para o tamanho do *buffer* de leitura dos *sockets*. Apesar de ser um valor relativamente grande para ser usado em comunica-

ções sobre redes, como o emissor e destinatário se encontram na mesma máquina, decidimos que essa questão não seria muito relevante para a nossa implementação, apesar de necessitar de revisão no caso do servidor se encontrar numa máquina diferente.

3.4 Elasticsearch e Kibana

Antes de armazenarmos dados no Elasticsearch, dado que se trata de uma ferramenta diferente daquelas que utilizámos previamente, efetuamos alguma pesquisa acerca da estrutura dos dados e como estes são armazenados, tendo chegado à proposta a seguir apresentada.

Os documentos por nós inseridos, fazem parte de um índice denominado de *calls*. Dentro desse índice encontram-se os diversos **tipos** de *system calls*, sendo que em cada tipo podemos encontrar os documentos associados ao mesmo. Comparando a estrutura do Elasticsearch com a do MySQL, podemos relacionar um índice como sendo o equivalente de uma base de dados, um tipo como sendo o equivalente de uma tabela, e um documento com propriedades como sendo o equivalente de colunas e linhas de uma tabela.[4]

Dado que no armazenamento dos dados provenientes do sistema de ficheiro está a ser utilizada a ferramenta Elasticsearch, considerámos que para a visualização dos mesmos faria sentido a utilização do **Kibana**. Deste modo conseguimos criar *dashboards* com gráficos e elementos intuitivos para a exibição de métricas ou registos.

A *dashboard* será uma parte fundamental do nosso sistema, dado que toda a análise será proveniente da mesma. Apesar de mencionadas agora, as métricas disponíveis na *dashboard* ajudaram a definir quais os campos que necessitámos de recolher para cada *system call*. Com isso em mente considerámos os seguintes gráficos/métricas úteis:

- Total de *System Calls*
- Proporção de cada tipo de *system call*
- Duração média de cada tipo de *system call*

- Análise temporal dos caminhos acedidos
- 10 caminhos mais acedidos
- Análise temporal dos processos ativos
- 50 processos mais ativos
- Tamanho médio (em KB) das escritas e leituras
- Duração média de todas as *system calls*

De seguida apresentámos alguns exemplos de gráficos e métricas retirados da *dashboard* após uma execução.

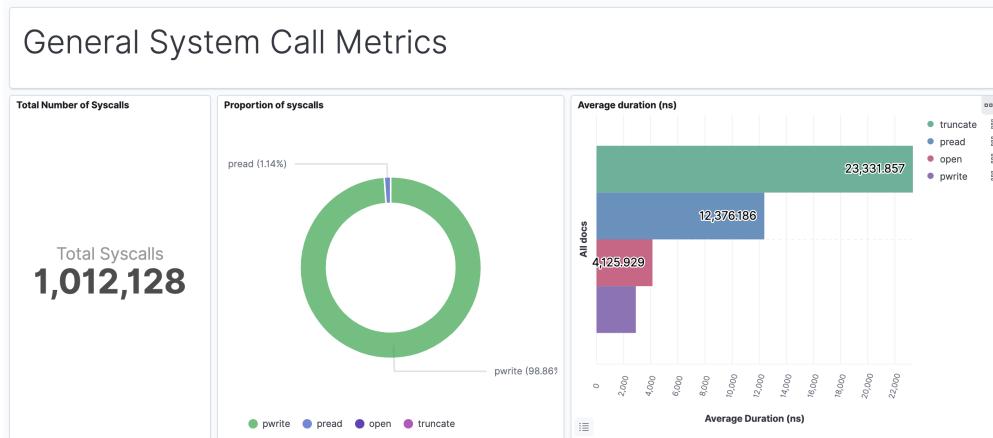


Figura 3: Total, proporção e duração média das *system calls*

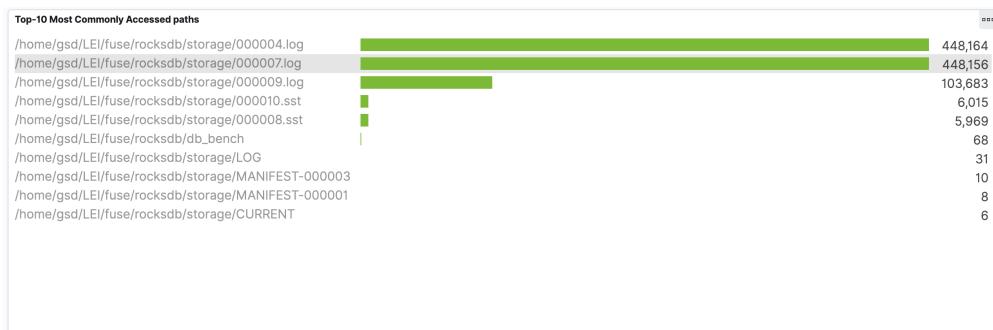


Figura 4: 10 caminhos mais acedidos

Miscellaneous Statistics



Figura 5: Métricas mistas sobre operações

Após a implementação desta *dashboard* no Kibana, foi-nos sugerida a utilização do **Metricbeat** para recolher métricas acerca da máquina onde corre o nosso sistema. O Metricbeat é uma ferramenta leve com integração no Kibana. Alguns dos dados recolhidos passam pela utilização do CPU e Memória ao longo do tempo e pela identificação dos processos que mais recursos consomem. Este tipo de métricas é extremamente interessante dado que podem ser correlacionadas com as métricas do nosso sistema de *tracing*.

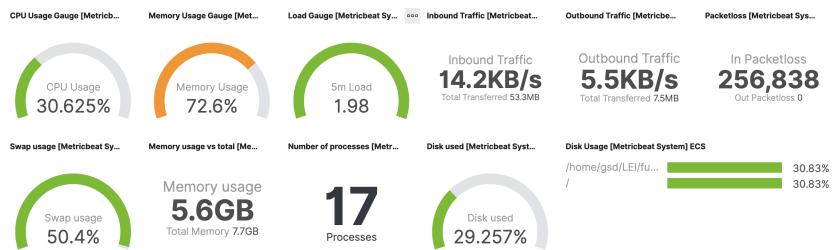


Figura 6: Métricas instantâneas do Metricbeat

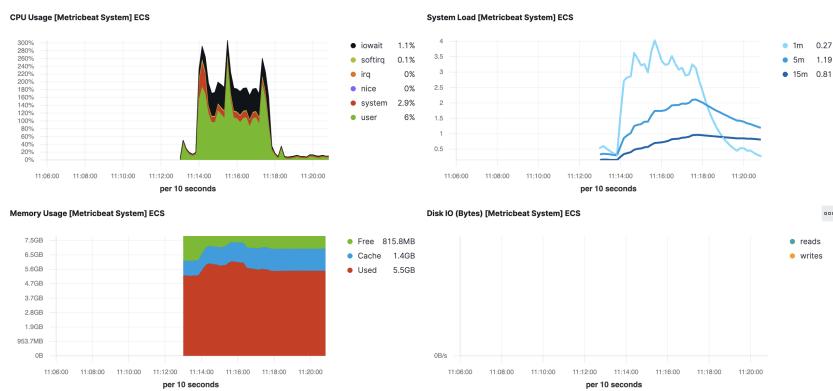


Figura 7: Métricas temporais sobre CPU, memória e disco

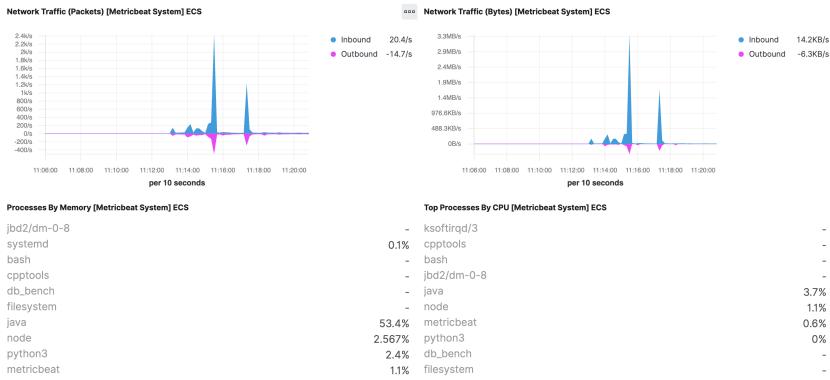


Figura 8: Métricas relativas à rede e processos

3.5 Otimizações

Após a implementação inicial, considerámos que seria importante avaliar o nosso sistema em busca de falhas de *performance* críticas. Após algum debate e após notarmos que a performance na inserção de documentos era muito baixa, chegamos à conclusão de que a inserção de 1 documento no Elasticsearch seria uma abordagem extremamente ineficiente. Com alguma pesquisa fomos capazes de encontrar uma ferramenta capaz de resolver o problema denominada de **Bulk API** fornecida pelo próprio Elasticsearch.[5] Esta API permite que múltiplas operações de indexação sejam agrupadas numa única invocação. Com isso, somos capazes de reduzir o *overhead* associado, e aumentar drasticamente a velocidade de indexação.

Apesar de ser uma estratégia para melhorar a performance na indexação, com alguma pesquisa fomos capazes de encontrar alguns parâmetros que poderiam ser ajustados para alcançar esse objetivo.[6] Um dos parâmetros que ajustamos foi o **intervalo de atualização** do Elasticsearch. Este encontrava-se a 1 segundo, tendo sido aumentado para **5 segundos**. Aquando a indexação, invocar a operação de atualização pode ser custoso e prejudicar a performance. Tendo em conta que o objetivo é ver dados em tempo real, este intervalo não poderá ser demasiado elevado, sendo os 5 segundos uma janela ideal.

Doravante, iremos denominar a versão prévia do nosso sistema como a versão não otimizada, e a versão que inclui as afinações no Elasticsearch como a versão otimizada.

4 Análise

Tal como mencionado previamente, a análise dos dados recolhidos no nosso sistema é um ponto crucial deste projeto. A análise dos dados recolhidos permitirá tirar conclusões acerca do comportamento das aplicações que executámos no nosso sistema, bem como acerca da performance das mesmas. Além de analisar os registos, poderá também ser relevante entender qual o peso das funcionalidades por nós adicionadas e quais os ganhos ou alterações entre as versões não otimizada e otimizada.

4.1 *Benchmarks*

De modo a responder a estas questões, corremos um conjunto de *benchmarks* sobre o nosso sistema, e utilizamos o resultado dos mesmos para efetuar uma análise das aplicações utilizadas. Para o efeito considerámos a utilização de 2 aplicações: o RocksDB e o PostgreSQL. Para todas as execuções foi utilizada uma máquina com as seguintes especificações de *hardware*: **8 GB** de memória, **128 GB** de armazenamento num disco SSD, e um processador Intel **i3 Dual Core com 3.7 GHz** de frequência.

4.1.1 RocksDB

Com este *benchmark*, o nosso objetivo é quantificar o desempenho que permimos ao utilizar uma *key-value store* no nosso sistema de *tracing*. Para executar os *benchmarks* utilizámos o **db_bench**[7] dado que este foi desenvolvido pelos criadores do RocksDB. A metodologia dos testes passa por correr os *benchmarks* num sistema de ficheiros FUSE simples, e no nosso sistema (versões não otimizada e otimizada). Para garantir alguma coerência nos resultados, é necessário um processo inicial que consiste em limpar a memória da máquina e partir de um estado limpo, isto é, não possuir registos no Elasticsearch e não possuir dados no RocksDB.

Para simular diferentes cargas de trabalho iremos correr 4 *workloads* existentes no db_bench: escritas sequenciais e aleatórias, e leituras sequenciais e aleatórias. Estas cargas de trabalho irão gerar dados que poderão em parte ser previ-

siveis, dado que se tratam de grupos de operações restritas. De modo a garantir alguma precisão nos dados, iremos executar cada *workload* 3 vezes e apresentar a média dos resultados, sendo que as métricas retiradas são o **número de operações por segundo**, a duração em **microsegundos por operação**, e a velocidade em **MB/s**.

Workload	FUSE		
	micros/op	ops/sec	MB/s
Sequential Read	0,6	1637912	181,2
Random Read	14,0	71348	7,9
Sequential Write	14,8	67780	7,5
Random Write	17,6	56904	6,3

Tabela 1: Resultados do db_bench para o sistema FUSE puro

Workload	Versão Não Otimizada		
	micros/op	ops/sec	MB/s
Sequential Read	1,3	761100	84,2
Random Read	52,7	19679	2,2
Sequential Write	23,3	42949	4,8
Random Write	27,0	37041	4,1

Tabela 2: Resultados do db_bench para a versão não otimizada

Workload	Versão Otimizada		
	micros/op	ops/sec	MB/s
Sequential Read	1,0	1053473	116,5
Random Read	20,9	48230	5,4
Sequential Write	23,2	43198	4,8
Random Write	27,7	36042	4,0

Tabela 3: Resultados do db_bench para a versão otimizada

A principal conclusão que podemos tirar acerca da perda de performance entre um sistema FUSE puro (Tabela 1) e o nosso sistema de *tracing* (Tabela 2) é que existe uma degradação significativa nos valores obtidos. No entanto, com a versão otimizada (Tabela 3), podemos notar que as velocidades de leitura melhoraram significativamente, e as velocidades de escrita se mantêm relativamente

semelhantes. Para ilustrar as diferenças entre as 3 execuções, elaborámos o seguinte gráfico (Figura 9).

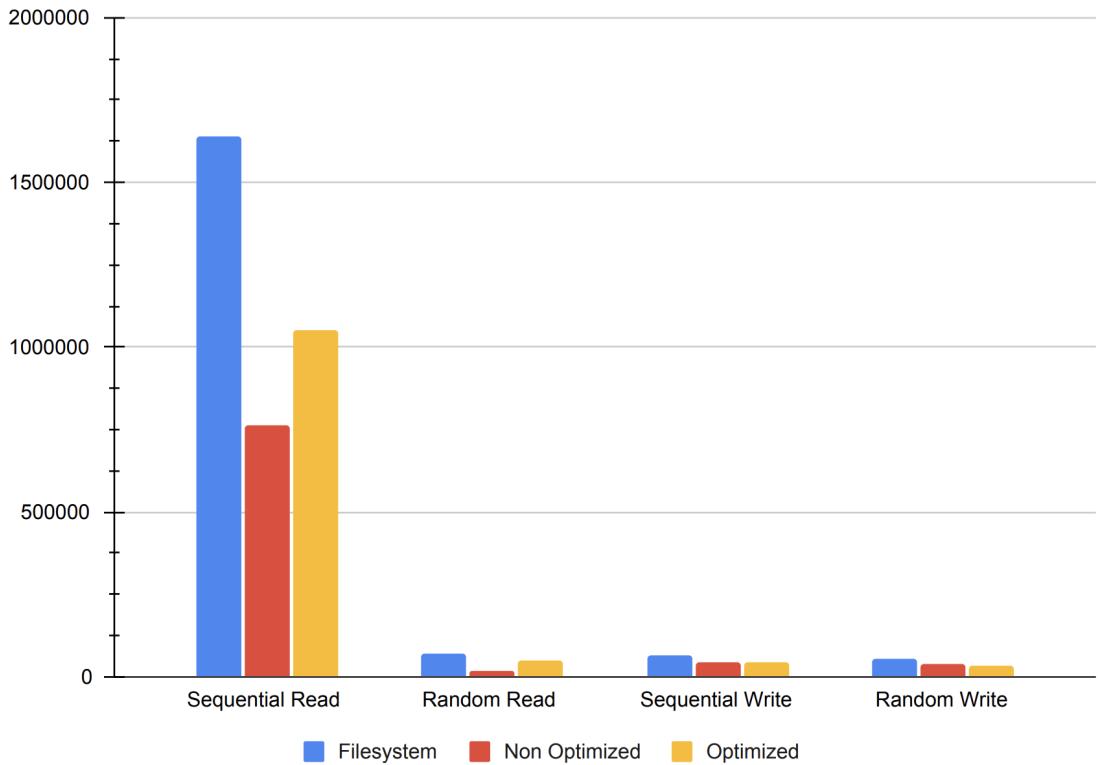


Figura 9: Comparaçāo dos resultados em operaçāes por segundo

Resumidamente, podemos afirmar que as funcionalidades por nós adicionadas causaram um impacto negativo na *performance* comparativamente a um sistema FUSE puro. No entanto, com a versão otimizada conseguimos recuperar alguma da *performance* perdida. Outra conclusão que retirámos foi de que com sucessivas iterações e melhorias sobre a nossa solução, conjugado com uma máquina com maior poder de processamento e uma maior quantidade de memória, seríamos capazes de nos aproximar significativamente dos valores ideias da versão do FUSE puro.

Com as experiências que realizamos, concluímos que com mais recursos do sistema, o número de *threads* alocadas ao *ThreadPoolExecutor* poderia ser maior, permitindo aumentar a concorrência no processo, e desse modo servir mais pedidos

em simultâneo. Do mesmo modo, o Elasticsearch teria mais recursos para poder efetuar a indexação. Dado isto, tudo nos leva a crer que as limitações de *hardware* são reais.

Tal como mencionado anteriormente, o objetivo do nosso sistema de *tracing* consiste em permitir efetuar uma análise das *system calls* invocadas ao sistema operativo em tempo real. Iremos agora focar a nossa atenção na questão dos dados recolhidos serem apresentados no Elasticsearch em tempo útil para podermos afirmar que é possível efetuar uma análise em tempo real, e para perceber se a versão otimizada teve algum impacto nessa vertente.

Com isso em mente, decidimos recorrer à *workload Sequential Write* para efetuar as medições entre o **início** e o **fim** do **processo de indexação** do Elasticsearch. Optámos por apenas verificar para uma *workload* dado que todas efetuam um número semelhante de operações (aproximadamente 1 milhão). Para a apresentação dos valores medidos, optámos por efetuar a **média** de **3 execuções**, de modo a obter um valor com menos flutuação, e efetuamos medições para ambas as versões do nosso sistema(Não Otimizada e Otimizada). Os tempos de indexação correspondem ao tempo necessário para inserir todos os índices no Elasticsearch.

Workload	Versão Não Otimizada	Versão Otimizada
Sequential Write	19 min, 59 seg	03 min, 31 seg

Tabela 4: Resultados dos tempos de indexação

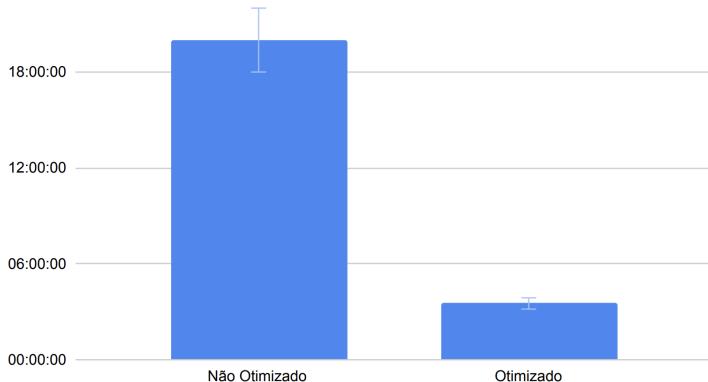


Figura 10: Comparaçao dos tempos de indexação

Considerando a Figura 10, podemos notar que com a versão Otimizada fomos capazes de obter uma melhoria significativa nos tempo necessário para efetuar a indexação. Apesar de se tratar de uma diferença acentuada, não significa que o resultado seja satisfatório, sendo que o nosso desejo é de que o processo seja o mais rápido possível. Apesar de reconhecermos que não são valores ideais, consideramos que para o *hardware* em questão e com a carga de trabalho suportada, os valores da versão Otimizada são os mais baixos que somos capazes de obter.

Desse modo, a redução no tempo de indexação em conjunto com as melhorias nas *workloads* anteriormente apresentadas, permitem-nos afirmar que nos caso de estudo do RocksDB, obtivemos sucesso na implementação das otimizações.

4.1.2 PostgreSQL

Com este *benchmark*, o nosso objetivo é quantificar o desempenho que permimos ao utilizar uma base de dados relacional no nosso sistema de *tracing*. Para executar os *benchmarks* utilizámos o **TPC-C**[8] dado que este simula uma carga transacional sobre o PostgreSQL e dado que é utilizado como *benchmark*. A metodologia dos testes passa por correr o *benchmark* num sistema de ficheiros FUSE simples, e no nosso sistema (versão otimizada). Para garantir alguma coerência nos resultados, é necessário um processo inicial que consiste em limpar a memória da máquina e partir de um estado limpo, isto é, não possuir registos no Elasticsearch e não possuir dados no PostgreSQL.

O TPC-C permite que sejam feitas configurações no ficheiro *workload-config.properties* sendo que considerámos **2 armazéns** como fator de escala e um tempo de execução de **5 minutos**. Relativamente a parâmetros da base de dados, considerámos os seguintes valores:

- tpcc.numclients = 10
- tpcc.numcustomers = 3000
- tpcc.numitems = 100000
- tpcc.numdistricts = 10
- tpcc.numnames = 999

De modo a garantir alguma precisão nos dados, iremos executar a carga transacional do TPC-C 3 vezes e apresentar a média dos resultados, sendo que as métricas retiradas são o **número de transações por segundo**, o **tempo de resposta em segundos**, e a **taxa de erro** em percentagem.

Workload	FUSE		
	Throughput	Tempo Resposta	Taxa de Erro
TPC-C	1,677 tx/s	0,006 s	0,0 %

Tabela 5: Resultados do TPC-C para o sistema FUSE puro

Workload	Versão Otimizada		
	Throughput	Tempo Resposta	Taxa de Erro
TPC-C	1,768 tx/s	0,007 s	0,002 %

Tabela 6: Resultados do TPC-C para a versão otimizada

Analisando os dados das Tabelas 5 e 6 consideramos que não podemos retirar muitas conclusões. A nossa opinião baseia-se no facto de com o nosso sistema, mesmo tratando-se da versão otimizada, seria expectável uma ligeira subida nos tempos de resposta, e uma ligeira descida nas transações realizadas por segundo. No entanto, tal não se verifica na totalidade. Os tempos de resposta subiram ligeiramente, mas o número de transações por segundo também subiu. Para poder tirar conclusões seria necessário perceber se a subida no número de transações por segundo se deve à base de dados, se é causada pelo aumento na taxa de erro, ou se seria necessário correr os testes de forma mais extensiva para diminuir o desvio dos valores.

Apesar dessa questão, podemos afirmar que com a versão do nosso sistema, os valores obtidos são satisfatórios, dado que tanto os tempos de resposta como a taxa de erro mantêm-se em valores aceitáveis e foram alvo de uma subida mínima, e, dado que o número de transações por segundo melhorou ligeiramente.

4.2 Análise de Resultados

Até agora, vimos detalhes acerca da implementação do nosso sistema de *tracing* em tempo real, bem como algumas das melhorias que foram implementadas. Foram também executadas diversas cargas de trabalho com o intuito de recolher diversas métricas de *performance*. No entanto, ainda não nos focamos na parte mais importante de todo o projeto, que é a análise das aplicações que correm sobre o nosso sistema de *tracing*.

Nesta secção iremos rever os resultados dos *benchmarks* executados anteriormente, e iremos efetuar uma análise de modo a perceber alguns detalhes acerca do funcionamento das cargas de trabalho, o seu impacto na máquina em que foram executadas, e se existem algumas falhas ou quebras evidentes na performance.

Com esse objetivo em mente, iremos tentar responder pelo menos às seguintes questões para cada um dos *benchmarks* e *workloads*.

- Qual a utilização de CPU e RAM durante a execução?
- Quais as *system calls* executadas e a sua duração média?
- Quais os caminhos mais acedidos?
- Nas *workloads* de escrita e leitura, qual o número médio de *bytes* escritos e lidos?

4.2.1 RocksDB

Relativamente ao RocksDB, iremos analisar os resultados de todas as *workloads* individualmente, começando pelas **escritas sequenciais**.

Na Figura 11 podemos analisar que na *workload* de escritas sequenciais é invocada quase exclusivamente a *system call pwrite*. Dada a natureza da carga de trabalho, considerámos que isso é bastante expectável. Outro ponto que considerámos interessante rever foi a disparidade entre duração de *system calls*. No casos daquelas que nos interessam mais, nomeadamente o **pwrite**, temos durações extremamente baixas, o que é excelente para a *workload* em questão. A duração média de um **pread** pareceu-nos um pouco alta, sendo que antes de tirar mais

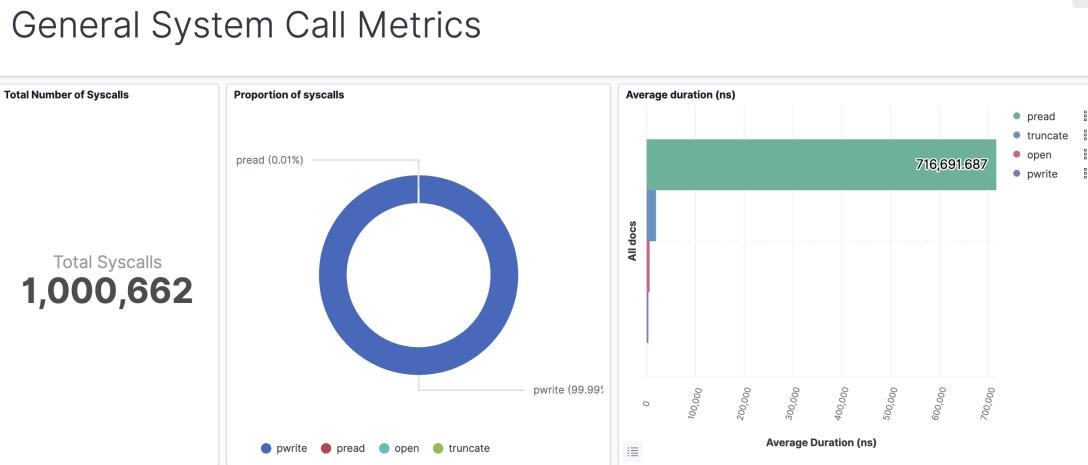


Figura 11: Escritas Sequenciais - Tipo e Duração das *system calls*

conclusões, fomos verificar o **tamanho médio em bytes lido e escrito** de cada uma das operações.

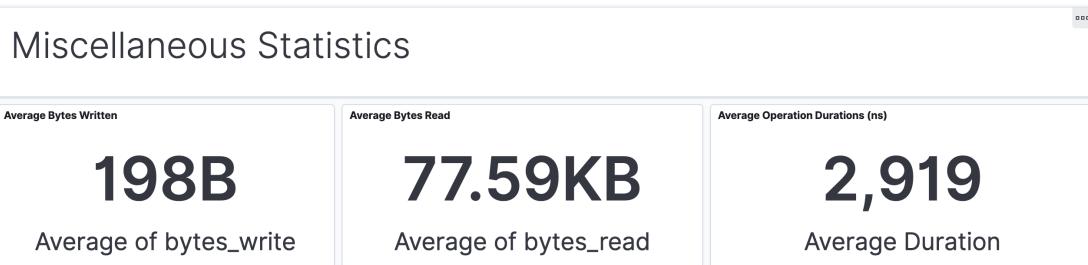


Figura 12: Escritas Sequenciais - Tamanho médio em *bytes* escrito e lido

Olhando para os valores da Figura 12, verificámos que em média são feitas escritas muito pequenas, e leituras consideravelmente grandes, o que ajuda a explicar a disparidade nas durações de cada operação.

O passo seguinte passou por analisar quais os caminhos mais acedidos no nosso sistema e aquilo que verificámos foi algo que já esperávamos: na Figura 13 os caminhos mais acedidos correspondem aos ficheiros onde foram realizadas as escritas pelo db_bench.

Ainda sobre a questão dos caminhos acedidos, analisando a Figura 14 conseguimos visualizar os 3 ficheiros de *log* referidos anteriormente e conseguimos

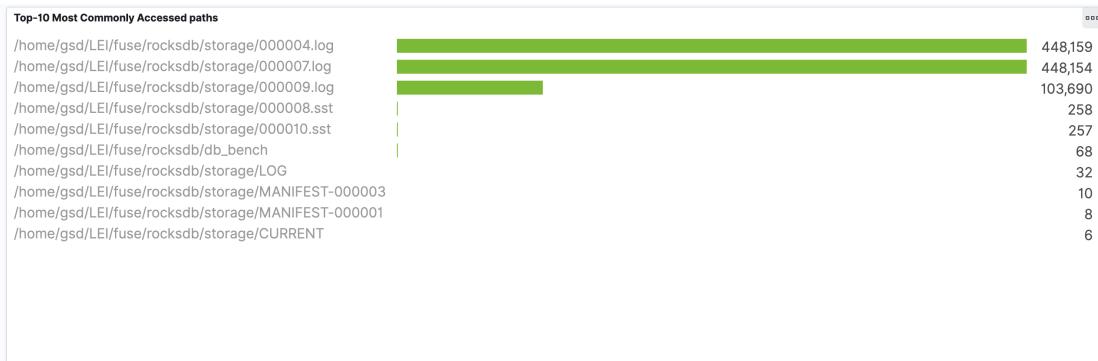


Figura 13: Escritas Sequenciais - Caminhos mais acedidos

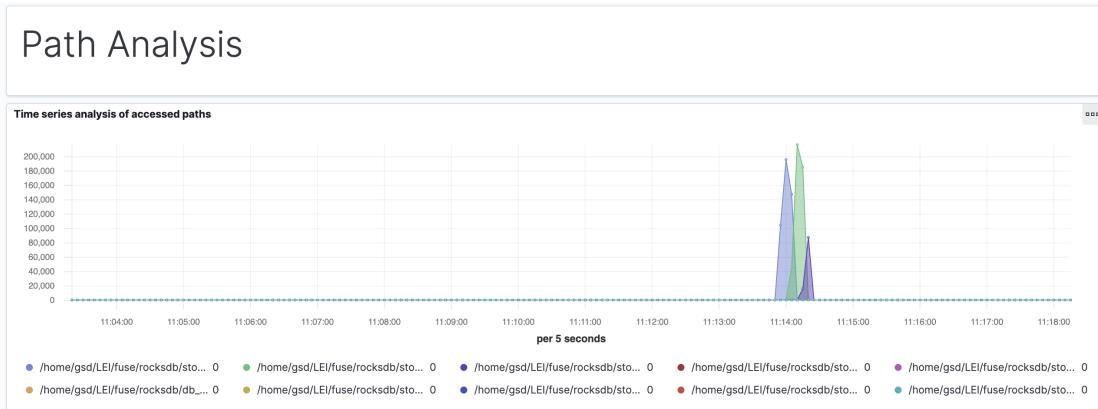


Figura 14: Escritas Sequenciais - Linha temporal dos caminhos acedidos

perceber que estes são acedidos sequencialmente, apenas existindo uma ligeira sobreposição temporal, e conseguimos perceber que estes acessos ocorreram entre as 11 horas, 13 minutos e aproximadamente 30 segundos e as 11 horas, 14 minutos e aproximadamente 20 segundos.

Recorrendo ao Metricbeats, e olhando para a Figura 15, conseguimos notar que pouco antes das 11 horas e 14 minutos existe uma subida acentuada quer na utilização do CPU, quer na utilização de memória, sendo que o aumento da utilização de CPU é maioritariamente causado por aplicações do utilizador. O primeiro pico visível pode ser associado ao db_bench, sendo que os restantes 2 picos podem ser justificados pelo processo de indexação do Elasticsearch. Relativamente à utilização de memória, notámos que existiu um aumento de aproximadamente

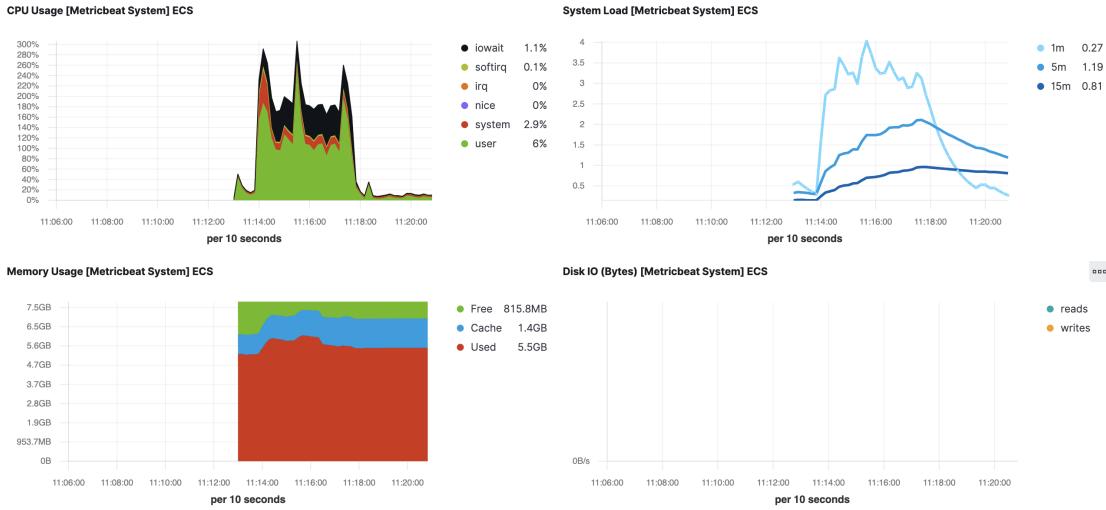


Figura 15: Escritas Sequenciais - Linha temporal da utilização de recursos

1 GB de memória quando é iniciado o db_bench e que também existe um ligeiro aumento na utilização da cache.

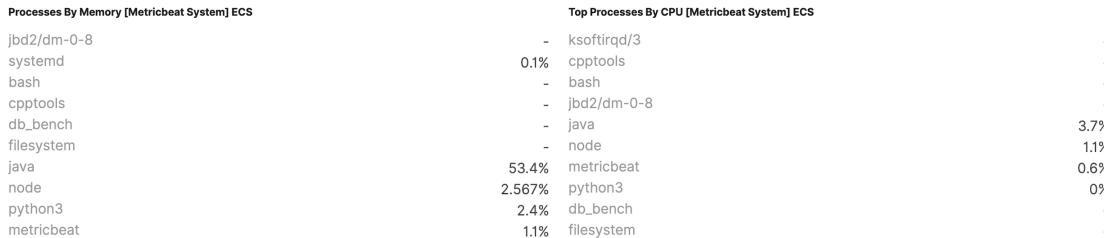


Figura 16: Escritas Sequenciais - Utilização de recursos por processos

Outra questão que é importante analisar consiste na utilização dos recursos da máquina por parte de processos, sendo que efetuámos descobertas interessantes. Tal como referido anteriormente, consideramos que os recursos da máquina eram um fator limitante, sendo que se verificou que a utilização de memória tem sido significativa. No entanto, para nossa surpresa, verificou-se que perto de **metade da memória utilizada** estava alocada a 1 único processo (java). O processo java encontra-se associado ao Elasticsearch, sendo que não o podemos simplesmente terminar. Dado que se trata de uma imagem, apenas são exibidos os valores após o término da execução, sendo que durante a mesma, os valores foram ainda mais elevados, estando a memória quase sempre ocupada na sua totalidade. Com

isto podemos afirmar que a presença do Elasticsearch na máquina afeta imenso a performance da mesma. Isto levou-nos a considerar a hipótese de numa iteração futura considerar colocar o Elasticsearch numa máquina diferente da que contém o sistema e perceber se existem ganhos ou perdas na *performance*. Convém notar que as métricas da Figura 28 são instantâneas, isto é, refletem valores no momento. Dado que as capturas foram efetuadas no final da execução, os valores não refletem todo o processo de execução, apenas as etapas finais.

A próxima *workload* que iremos analisar é a de **escritas aleatórias**.

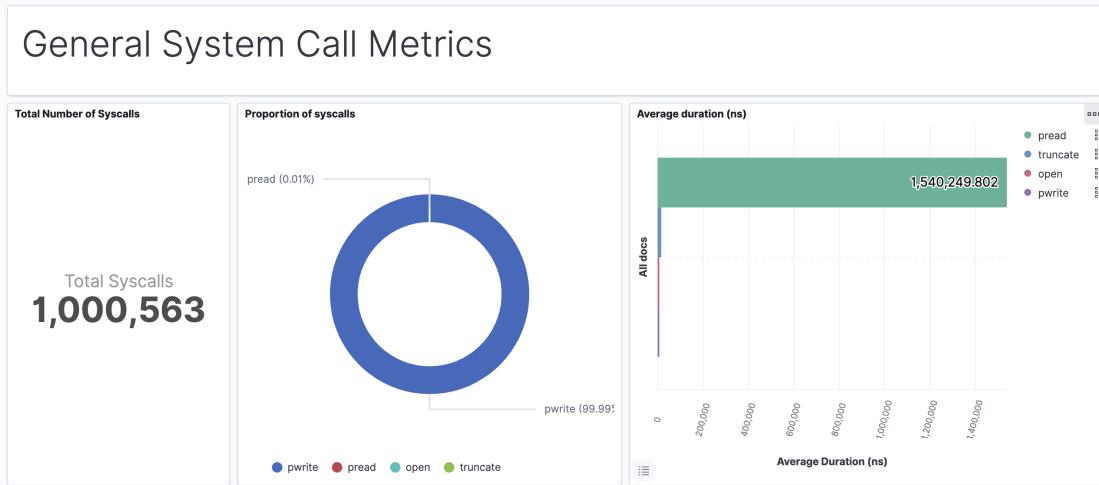


Figura 17: Escritas Aleatórias - Tipo e Duração das *system calls*

Na Figura 17 podemos analisar que na *workload* de escritas aleatórias é invocada quase exclusivamente a *system call* **pwrite**. Dada a natureza da carga de trabalho, considerámos que isso é bastante expectável. Tal como verificado anteriormente, no **pwrite**, temos durações extremamente baixas, o que é excelente para a *workload* em questão. A duração média de um **pread** é mais uma vez um pouco alta, sendo que antes de tirar mais conclusões, efetuámos a mesma verificação do **tamanho médio em bytes lido e escrito** de cada uma das operações.

Olhando para os valores da Figura 18, verificámos mais uma vez que em média são feitas escritas muito pequenas, e leituras consideravelmente grandes, o que explica mais uma vez a disparidade nas durações de cada operação.

De seguida voltamos a analisar quais os caminhos mais acedidos durante a

Miscellaneous Statistics

Average Bytes Written

188B

Average of bytes_write

Average Bytes Read

78.81KB

Average of bytes_read

Average Operation Durations (ns)

3,326

Average Duration

Figura 18: Escritas Aleatórias - Tamanho médio em *bytes* escrito e lido

execução do *benchmark* no nosso sistema e aquilo que verificámos foi algo que já esperávamos: na Figura 19 os caminhos mais acedidos correspondem aos ficheiros onde foram realizadas as escritas pelo db_bench.

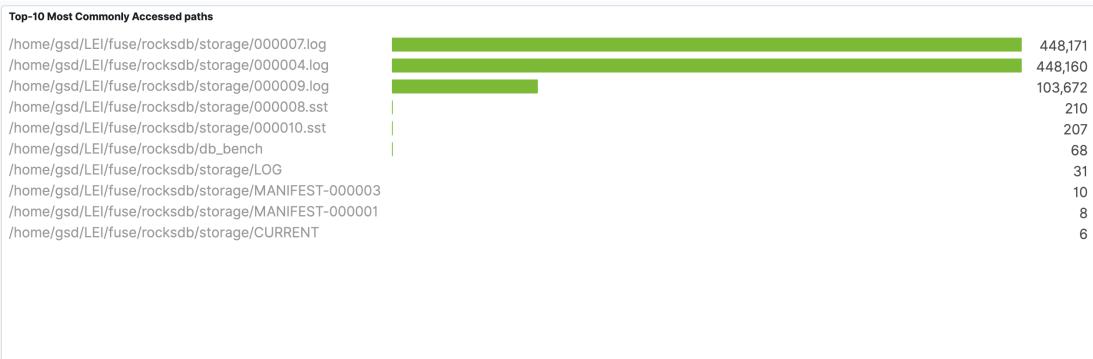


Figura 19: Escritas Aleatórias - Caminhos mais acedidos

Path Analysis

Time series analysis of accessed paths

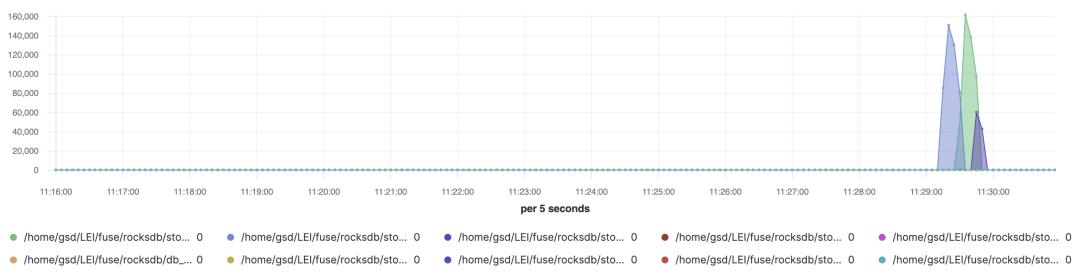


Figura 20: Escritas Aleatórias - Linha temporal dos caminhos acedidos

Ainda sobre a questão dos caminhos acedidos, analisando a Figura 20 con-

seguimos visualizar os 3 ficheiros de *log* referidos anteriormente e conseguimos perceber que estes são acedidos sequencialmente, apenas existindo uma ligeira sobreposição temporal, e conseguimos perceber que estes acessos ocorreram entre as 11 horas, e aproximadamente 29 minutos e as 11 horas e 30 minutos.

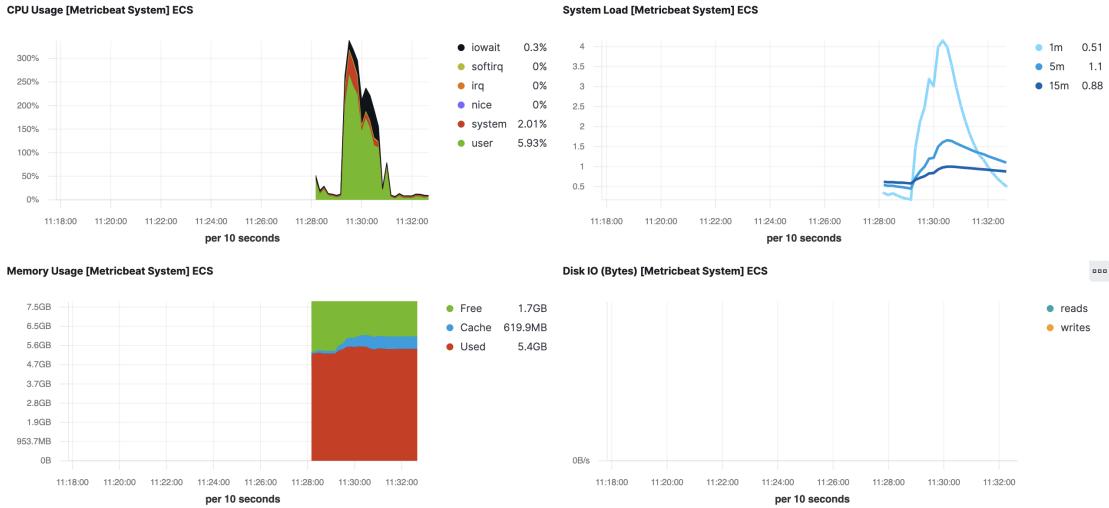


Figura 21: Escritas Aleatórias - Linha temporal da utilização de recursos

Recorrendo ao Metricbeats, e olhando para a Figura 21, conseguimos notar que por volta das 11 horas e 29 minutos existe uma subida acentuada quer na utilização do CPU, quer na utilização de memória. O aumento da utilização de CPU visível é maioritariamente causado por aplicações do utilizador. O pico visível pode ser associado ao db_bench, sendo que coincide temporalmente com o acesso aos caminhos utilizados pelo db_bench. Relativamente à utilização de memória, notámos que existiu um aumento de aproximadamente **1 GB** de memória quando é iniciado o db_bench.

Outra questão que é importante analisar consiste na utilização dos recursos da máquina por parte de processos, sendo que voltamos a constatar a mesma situação da *workload* anterior. Aproximadamente **metade da memória utilizada do sistema é consumida por um processo java**, sendo que já constatamos que este pertence ao Elasticsearch. Analisando também o consumo do CPU por parte de processos, podemos verificar que tanto um processo java (associado ao Elasticsearch) como um processo python3 (associado ao nosso servidor intermediário)

Processes By Memory [Metricbeat System] ECS		Top Processes By CPU [Metricbeat System] ECS	
-	kworker/0:0H	-	-
ktreadd	ktreadd	-	-
ksoftirqd/0	ksoftirqd/0	-	-
systemd	-	-	-
db_bench	-	-	-
filesystem	-	-	-
java	node	6.8%	-
python3	metricbeat	0.7%	-
node	53.433% jbd2/dm-0-8	0.65%	-
metricbeat	4.433% filesystem	-	-
jbd2/dm-0-8	3.267% db_bench	-	-
	1.1% python3	62.633%	-
	0% java	51.567%	-

Figura 22: Escritas Aleatórias - Utilização de recursos por processos

rio) consomem percentagens significativas do CPU. Isto comprova uma mais vez o nosso interesse em correr o Elasticsearch numa máquina distinta, ou de possuir uma máquina com mais recursos de memória e CPU.

A próxima *workload* que iremos analisar é a de **leituras sequenciais**. Antes de partirmos para a análise dos resultados desta *workload*, convém rever o modo como esta se processa. As *workloads* de leitura no db_bench têm obrigatoriamente de ser precedidas por escritas, de modo a existir conteúdo para ler, dado que não é feito um populamento inicial automático. Desse modo, para ambas as *workloads* de leitura são corridas *workloads* de escritas sequenciais, sendo que isso se irá refletir nos resultados.

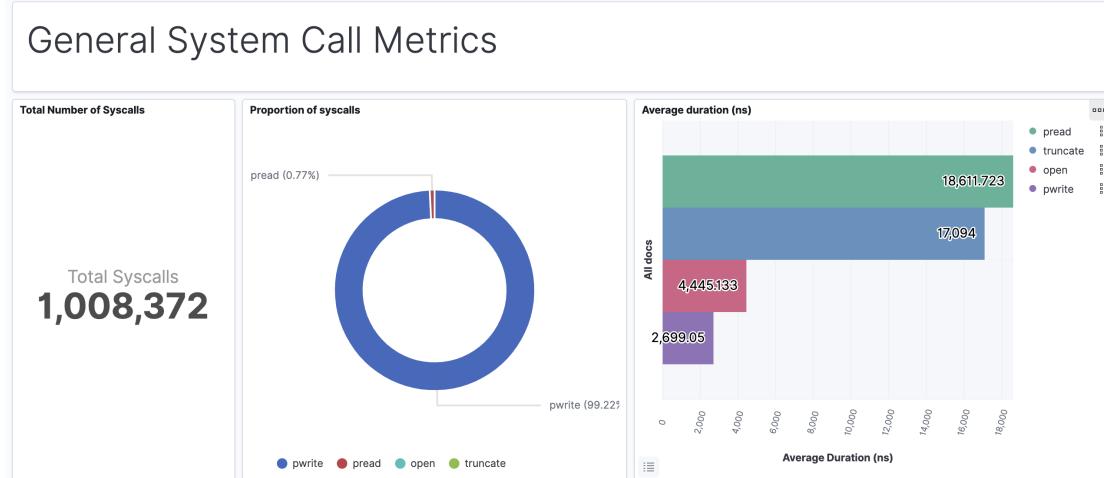


Figura 23: Leituras Sequenciais - Tipo e Duração das *system calls*

Na Figura 23 podemos verificar que tal como mencionado, são efetuadas escritas e leituras. O número total de *system calls* aumentou um pouco em relação

às *workloads* de escrita, e as proporções são ligeiramente diferentes. Tratando-se de 1 milhão de operações, uma diferença de 0,5% pode parecer insignificante, apesar de se tratarem de 5000 operações. As durações das leituras face às escritas continuam a ser mais elevadas(aproximadamente 9 vezes mais longas), sendo que iremos voltar a analisar o tamanho das escritas e leituras.

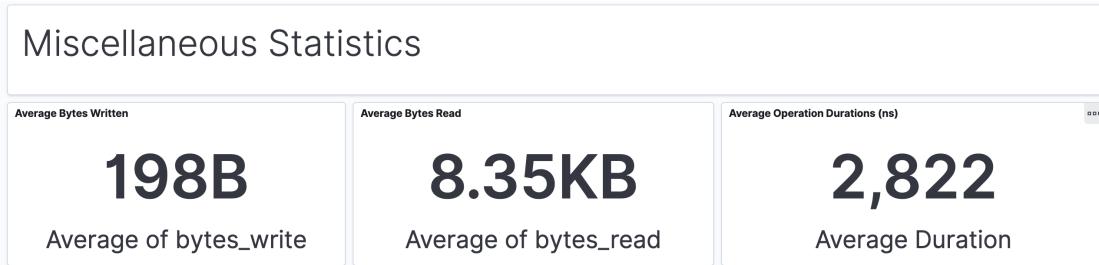


Figura 24: Leituras Sequenciais - Tamanho médio em *bytes* escrito e lido

Na Figura 24 podemos verificar que o tamanho das leituras é significativamente maior que o tamanho das escritas. No entanto, a diferença não é tão acen-tuada como nas *workloads* de escrita, o que explica a diminuição do fosso entre durações médias de *system calls*. O facto do tamanho das leituras ser superior ao das leituras ajuda-nos a explicar o porquê do número de escritas ser muito superior ao de leituras. Se multiplicarmos o total de *system calls* pela proporção de escritas, obtemos aproximadamente 1000506 escritas. Multiplicando o número de escritas pelo tamanho médio, podemos dizer que foram escritos aproximadamente 189 MB. Efetuando o mesmo processo para as leituras, chegamos à conclusão que foram feitas aproximadamente 7764 leituras. Com estes dados, chegámos à conclusão que foram lidos aproximadamente 63 MB.

O passo seguinte passou por analisar quais os caminhos mais acedidos no nosso sistema e aquilo que verificámos foi semelhante aos casos anteriores: na Figura 25 os caminhos mais acedidos correspondem aos ficheiros onde foram realizadas as escritas pelo db_bench.

Ainda sobre a questão dos caminhos acedidos, analisando a Figura 26 conseguimos visualizar os 3 ficheiros de *log* referidos anteriormente e conseguimos perceber que estes são acedidos sequencialmente, apenas existindo uma ligeira sobreposição temporal, e, conseguimos perceber que estes acessos ocorreram entre

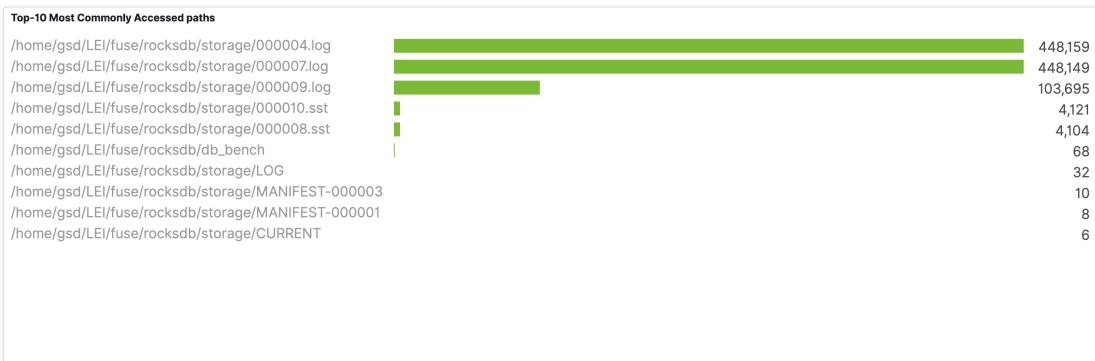


Figura 25: Leituras Sequenciais - Caminhos mais acedidos

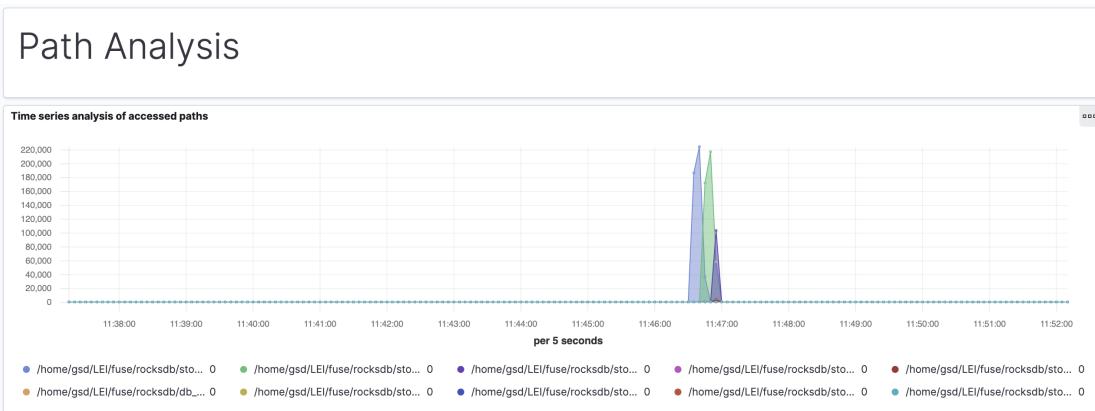


Figura 26: Leituras Sequenciais - Linha temporal dos caminhos acedidos

as 11 horas, 46 minutos e aproximadamente 30 segundos e as 11 horas e aproximadamente 47 minutos.

Recorrendo ao Metricbeats, e olhando para a Figura 27, conseguimos notar que por volta 11 horas e 46 minutos existe uma subida acentuada quer na utilização do CPU, quer na utilização de memória, sendo que o aumento da utilização de CPU é maioritariamente causado por aplicações do utilizador. Na utilização do CPU, o primeiro pico visível pode ser associado ao db_bench, sendo que os restantes 2 picos podem ser justificados pelo processo de indexação do Elasticsearch. Relativamente à utilização de memória, notámos que existiu um aumento de aproximadamente **1 GB** de memória quando é iniciado o db_bench e que também existe um ligeiro aumento na utilização da cache.

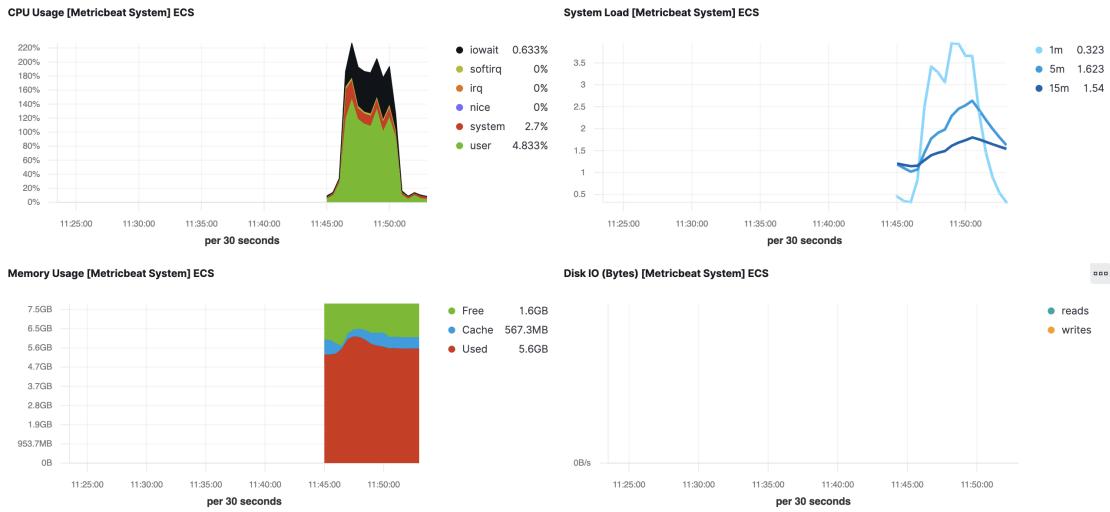


Figura 27: Leituras Sequenciais - Linha temporal da utilização de recursos

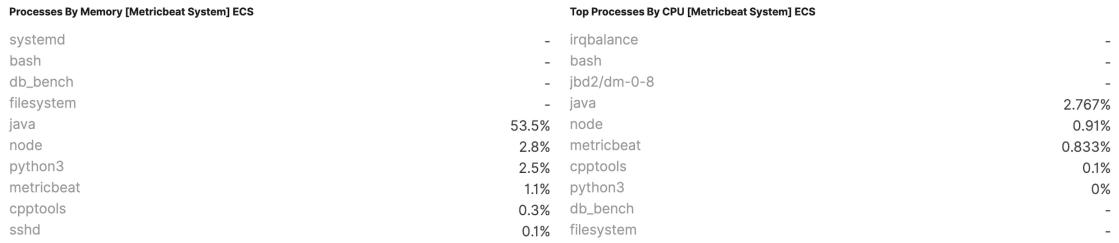


Figura 28: Leituras Sequenciais - Utilização de recursos por processos

Outra questão que é importante analisar consiste na utilização dos recursos da máquina por parte de processos. Tal como verificado anteriormente, voltamos a ter uma percentagem significativa dos recursos a ser consumida por um processo java associado ao Elasticsearch.

A última *workload* que iremos analisar é a de **leituras aleatórias**.

Na Figura 29 podemos verificar que tal como esperado, são efetuadas escritas e leituras. O número total de *system calls* aumentou um pouco em relação às *workloads* de escrita, e as proporções são ligeiramente diferentes. As durações das leituras face às escritas continuam a ser mais elevadas(aproximadamente 5 vezes mais longas), sendo que iremos voltar a analisar o tamanho das escritas e leituras.

Na Figura 30 podemos verificar que o tamanho das leituras é significativa-

General System Call Metrics



Figura 29: Leituras Aleatórias - Tipo e Duração das *system calls*

Miscellaneous Statistics



Figura 30: Leituras Aleatórias - Tamanho médio em *bytes* escrito e lido

mente maior que o tamanho das escritas. No entanto, a diferença não é tão acen-tuada como nas *workloads* anteriores, o que explica a diminuição do fosso entre durações médias das *system calls*. O facto do tamanho das leituras ser superior ao das leituras ajuda-nos a explicar o porquê do número de escritas ser muito su-perior ao de leituras. Se multiplicarmos o total de *system calls* pela proporção de escritas, obtemos aproximadamente 1000583 escritas. Multiplicando o número de escritas pelo tamanho médio, podemos dizer que foram escritos aproximadamente 189 MB. Efetuando o mesmo processo para as leituras, chegamos à conclusão que foram feitas aproximadamente 11538 leituras. Com estes dados, chegámos à con-clusão que foram lidos aproximadamente 63 MB. Estes valores coincidem com a *workload* de escritas sequenciais.

O passo seguinte passou por analisar quais os caminhos mais acedidos no nosso

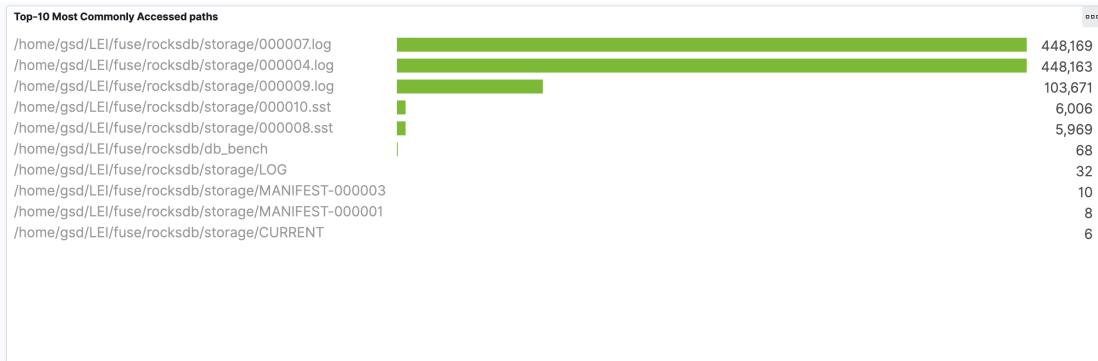


Figura 31: Leituras Aleatórias - Caminhos mais acedidos

sistema e aquilo que verificámos foi semelhante aos casos anteriores: na Figura 31 os caminhos mais acedidos correspondem aos ficheiros onde foram realizadas as escritas pelo db_bench.

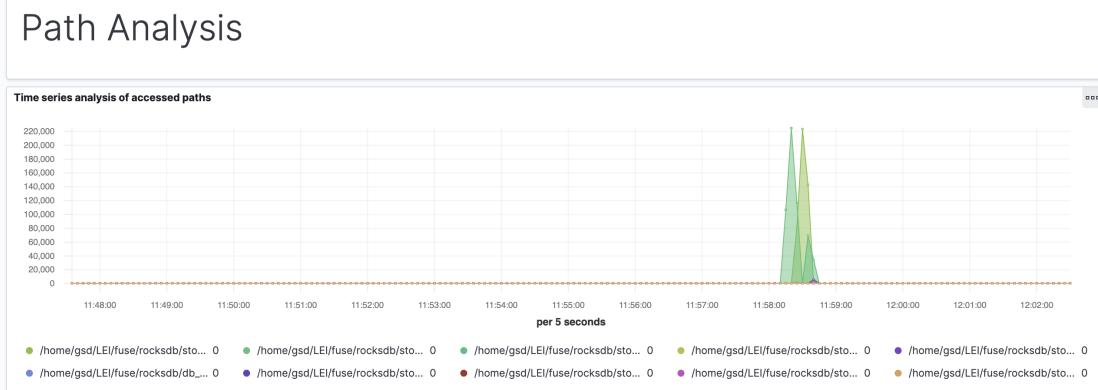


Figura 32: Leituras Aleatórias - Linha temporal dos caminhos acedidos

Ainda sobre a questão dos caminhos acedidos, analisando a Figura 32 conseguimos visualizar os 3 ficheiros de log referidos anteriormente e conseguimos perceber que estes são acedidos sequencialmente, apenas existindo uma ligeira sobreposição temporal, e, conseguimos perceber que estes acessos ocorreram entre as 11 horas e aproximadamente 58 minutos e as 11 horas e aproximadamente 59 minutos.

Recorrendo ao Metricbeats, e olhando para a Figura 33, conseguimos notar que por volta 11 horas e 58 minutos existe uma subida acentuada na utilização do

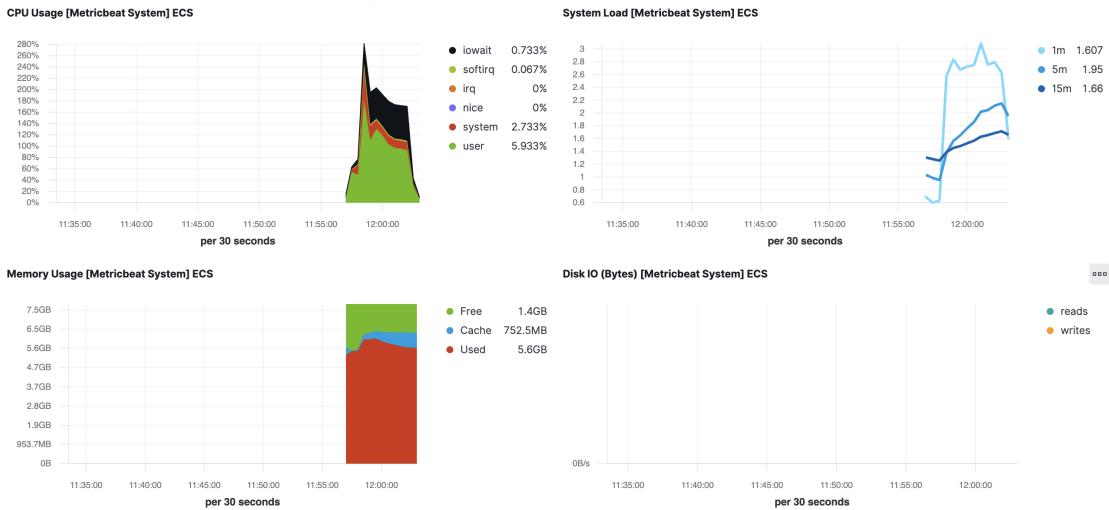


Figura 33: Leituras Aleatórias - Linha temporal da utilização de recursos

CPU e um aumento na utilização de memória, sendo que o aumento da utilização de CPU é maioritariamente causado por aplicações do utilizador. Na utilização do CPU, podemos ver um pico inicial que pode ser associado ao db_bench, sendo que o comportamento restante pode ser justificados pelo processo de indexação do Elasticsearch. Relativamente à utilização de memória, notámos que existiu um aumento de aproximadamente **1 GB** de memória por volta das 11 horas e 58 minutos, que coincide com o momento em que é iniciado o db_bench.

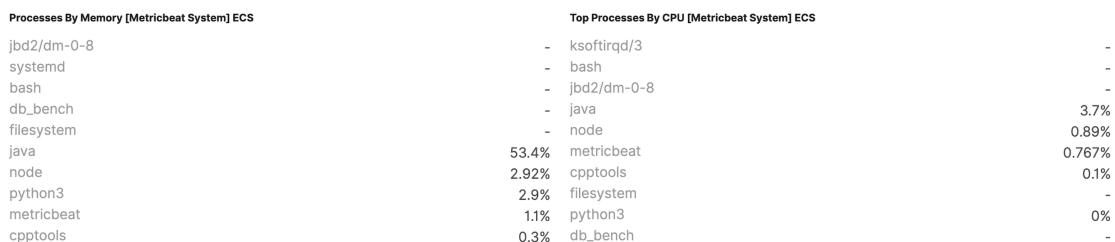


Figura 34: Leituras Aleatórias - Utilização de recursos por processos

A última questão que é importante analisar consiste na utilização dos recursos da máquina por parte de processos. Mais uma vez, voltamos a ter uma percentagem significativa dos recursos a ser consumida por um processo java associado ao Elasticsearch.

Após terminarmos a análise ás 4 *workloads* podemos tirar algumas conclusões. Todas as *workloads* efetuam as escritas e leituras por 3 ficheiros de *log*. Conseguimos também corelacionar a utilização de CPU e memória com a execução do db_bench e com o processo de indexação do Elasticsearch. Conseguimos perceber que o db_bench se comporta de uma forma relativamente expectável e semelhante em todos os casos, ao contrário do processo de indexação, que não tem sempre o mesmo impacto na utilização de recursos. Fomos também capazes de perceber a razão da disparidade dos tempos de execução de leituras e escritas com a quantidade de *bytes* lidos. Por fim, conseguimos também perceber o impacto quer do nosso servidor, quer do db_bench no sistema, e, fomos capazes de entender para onde estão a ser desviados a maioria dos recursos da máquina(processo java relacionado com o Elasticsearch).

4.2.2 PostgreSQL

A outra aplicação que optamos por analisar foi o PostgreSQL e como *workload* utilizamos o **TPC-C** que nos permite executar várias transações por vários clientes em armazéns, colocando assim pressão sobre a base de dados.

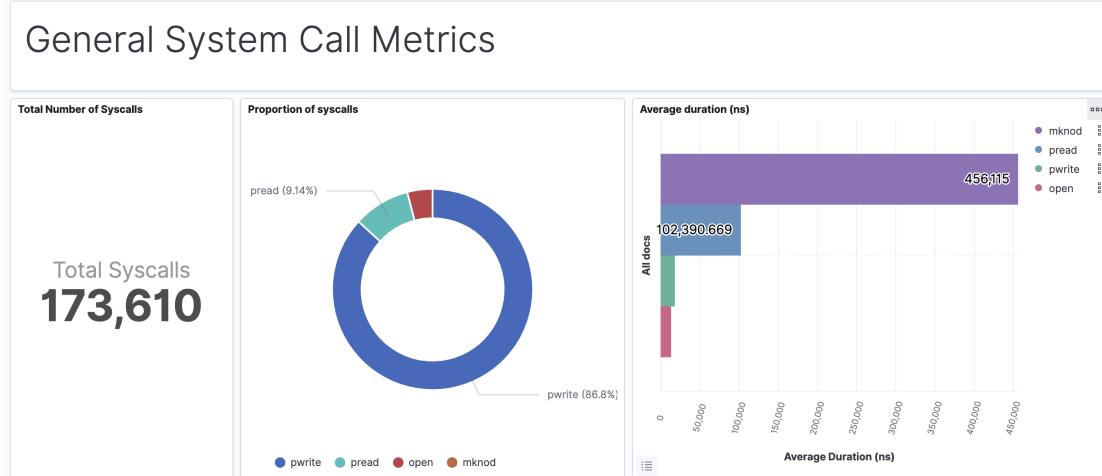


Figura 35: TPC-C - Tipo e Duração das *system calls*

Na Figura 35 vemos então as várias *system calls* que são realizadas pela *workload* verificando que a maioria são escritas como seria de prever, uma vez que primeiro é preciso povoar a base de dados, e, mesmo a maior parte das transações realizadas

depois fazem também escritas. Quanto às durações das operações, são semelhantes às que observamos no **RocksDB** onde temos as leituras a demorarem em média mais que as escritas e embora se tenha realizado com muito pouco frequência, o *mknod* foi a operação que demorou mais tempo. No entanto, não terá grande impacto no desempenho visto que ocorreu um número muito reduzido de vezes.

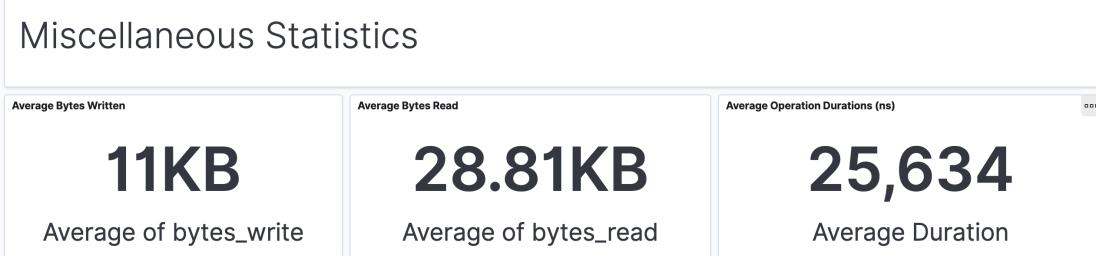


Figura 36: TPC-C - Tamanho médio em *bytes* escrito e lido

Na Figura 36 é visível que as escritas têm um tamanho reduzido quando comparadas com as leituras (cerca de três vezes mais). Contudo, quando comparado com o tamanho médio destas operações realizadas pelo **RocksDB** observamos que o PostgreSQL faz as operações com tamanhos bastante superiores, cerca de 7 vezes mais tamanho nas escritas e 5 vezes mais tamanho nas leituras o que pode também justificar o número mais reduzido de *system calls* captadas sendo que como a *workload* não é a mesma, esta pode não ser a única razão. Contudo, com o aumento do tamanho em cada uma das operações individuais seria de esperar que a duração das mesmas aumentasse e foi o que se verificou. O PostgreSQL tem uma média de duração das operações cerca de 8 vezes mais elevada. Como já foi referido, as *workloads* para ambas as aplicações são diferentes, mas, mesmo assim achamos relevante a diferença em algumas destas métricas, o que nos permite estabelecer uma comparação entre as duas.

Na análise de quais os caminhos mais acedidos no nosso sistema verificámos o seguinte: na Figura 37 o caminho mais acedido corresponde ao ficheiro de *log* que o PostgreSQL estava a utilizar. As outras escritas são feitas na diretoria *base* que é onde estão guardados os dados e por isso são os dados contidos efetivamente na base de dados.

Podemos também verificar na Figura 38 que os ficheiros são acedidos por uma

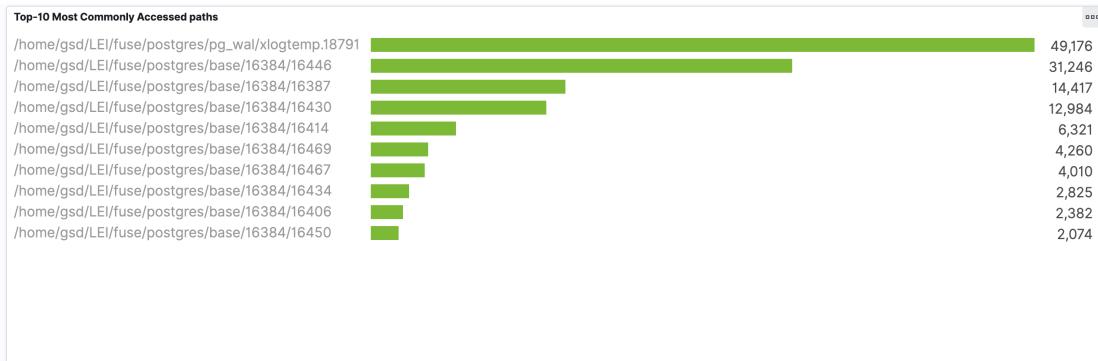


Figura 37: TPC-C - Caminhos mais acedidos

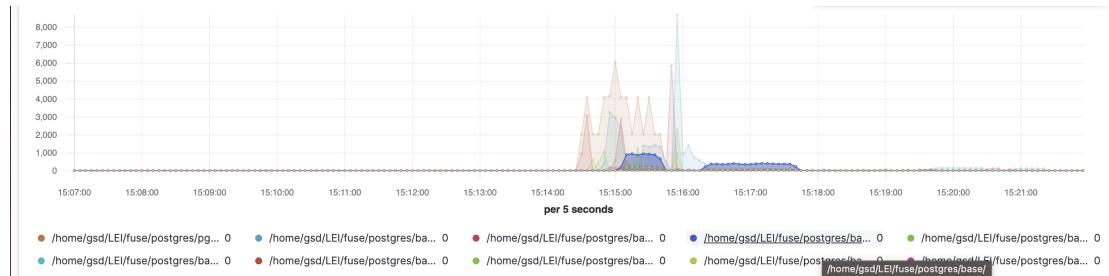


Figura 38: TPC-C - Linha temporal dos caminhos acedidos

forma relativamente aleatória e que varia conforme as transações que vão sendo realizadas, tendo em conta que o ficheiro de *log* é acedido principalmente numa primeira fase de execução da aplicação que é quando é feito o povoamento da base de dados.

Utilizando mais uma vez o Metricbeats, na Figura 39, vemos que por volta 15 horas e 14 minutos há um aumento da utilização do CPU coincidente com a altura em que foi realizado o processo de carregamento, uma vez concluído esse processo vemos uma utilização bem mais reduzida. A utilização de memória aumentou principalmente após a fase mais intensa de CPU ter terminado, o que pode ser justificado por ser apenas depois que são realizadas leituras e por isso tiram melhor proveito deste tipo de memória e por isso é mais usado.

Quanto à utilização dos recursos da máquina durante a execução, a maior parte continua a ser utilizado por um processo java que associamos ao Elasticsearch, sendo também visível alguma utilização de CPU associada ao servidor *python*.

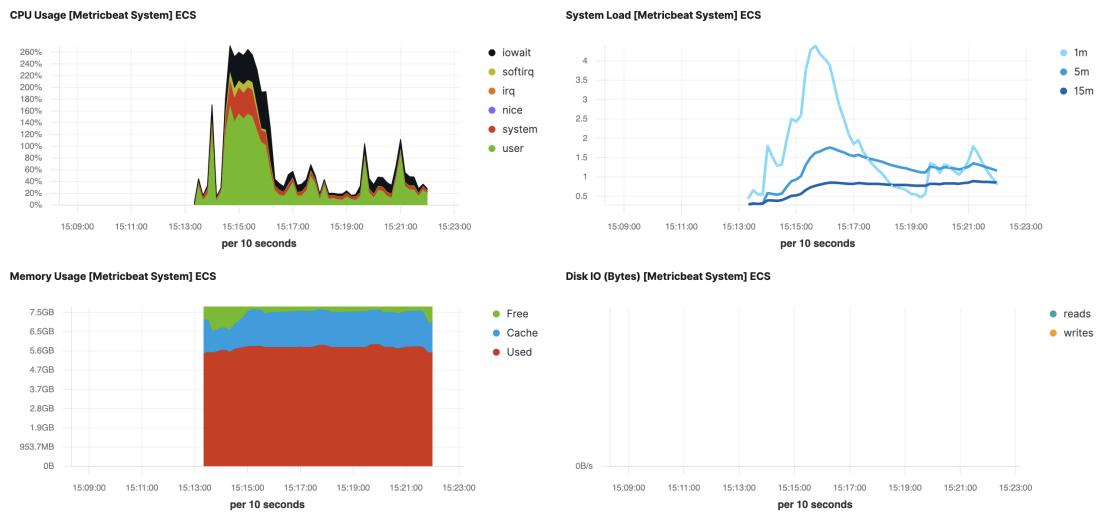


Figura 39: TPC-C - Linha temporal da utilização de recursos

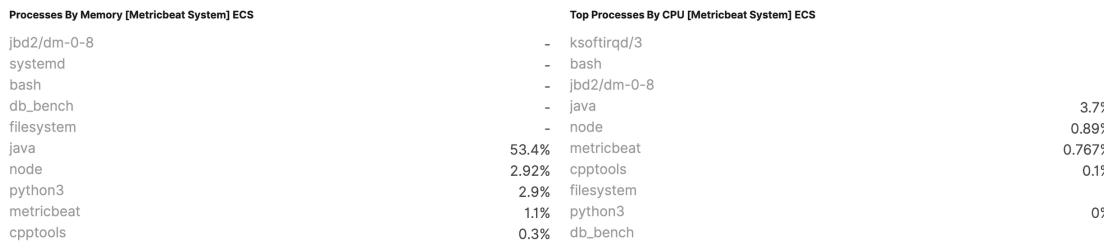


Figura 40: TPC-C - Utilização de recursos por processos

No entanto, convém mencionar que as capturas foram efetuadas após o término da execução, sendo que os valores instantâneos como por exemplo percentagens de utilização de processos, refletem o estado após a execução da carga de trabalho, ou já em fases finais.

5 Conclusões

Nesta secção iremos fazer um pequeno balanço acerca do sistema desenvolvido, considerando os objetivos inicialmente traçados, e comparando os mesmos com os resultados obtidos. Iremos também mencionar algumas sugestões nossas para melhorias ou adições interessantes para iterações futuras deste sistema.

5.1 Balanço

Nas fases iniciais do desenvolvimento deste projeto, foram propostos alguns objetivos para o desenvolvimento deste sistema. Analisando esses mesmos objetivos, consideramos que conseguimos alcançar todos com um grau de sucesso satisfatório. No entanto, isto não significa que não existem pontos de melhoria, especialmente em questões de *performance*.

Os nossos objetivos de causar o menor impacto possível nas aplicações que correm sobre o nosso sistema e de ter o menor desfasamento possível na análise em tempo real dos dados, são algo complexos de alcançar, sendo que verificamos isso na secção dos *benchmarks*. Tal como já foi visto anteriormente, foi criada uma versão otimizada do nosso sistema, sendo que com essa versão fomos capazes de melhorar tanto a *performance* das aplicações que correm sobre o nosso sistema (como por exemplo nas leituras do db_bench), como de diminuir drasticamente os tempos de indexação. Esta versão, apesar de ir de encontro com os objetivos traçados, não nos deixa completamente satisfeitos, sendo que consideramos que existem melhorias a efetuar para diminuir o fosso entre um sistema FUSE puro, e a nossa versão otimizada. Essas melhorias poderão ser tanto sob a forma de alterações na arquitetura da solução, como na utilização de máquinas com mais recursos de memória e maior poder de processamento.

Com as conclusões a que chegámos ao efetuar a análise de resultados, surgiu uma sugestão que poderá ser interessante explorar. Dado que em todas as execuções se notou a presença de um processo java a consumir uma quantidade elevada de recursos, especialmente memória, seria uma ideia interessante explorar uma alteração na arquitetura. Esta alteração passaria por ter o Elasticsearch numa

máquina separada, sendo que como o servidor que construímos possui suporte para o cliente Elasticsearch, se trara de uma solução possível. Seria interessante perceber se existiriam ganhos de *performance* ao libertar a máquina da carga do Elasticsearch, e até que ponto esta separação introduz atrasos de comunicação adicionais.

5.2 Trabalhos Futuros

Apesar de termos mencionado que nos encontramos satisfeitos com o nosso sistema, e que este cumpre os objetivos definidos, não significa que não possam existir melhorias ou alterações que fossem interessantes.

Tal como mencionamos anteriormente, numa iteração futura seria interessante colocar o Elasticsearch numa máquina distinta e perceber se os ganhos que obtemos ao libertar a máquina onde correm as aplicações, se sobreponem aos atrasos inerentes à comunicação com uma segunda máquina. Paralelamente a esta questão, seria do nosso interesse continuar a melhorar a *performance* do nosso sistema, de modo a minimizar o impacto em aplicações e a reduzir os tempos de indexação.

Outra melhoria que acrescentaria utilidade ao nosso sistema seria a adição ou reformulação da *dashboard* do Elasticsearch de modo a ter métricas adicionais que sejam relevantes, ou trocar algumas menos relevantes. Na análise de resultados chegámos à conclusão que os dados que apresentámos de momento são um pouco monótonos e permitem uma análise limitada.

Por fim, a última questão que discutimos e que seria relevante, passa pela adição de mais *system calls*. Apesar de termos um conjunto satisfatório de *system calls*, a adição de mais algumas apenas acresce valor ao sistema e permite efetuar análises de maior qualidade.

Referências

- [1] *libfuse*, "<https://github.com/libfuse/libfuse>", Acedido: 21-03-2020.
- [2] *Socket Programming HOWTO*, "<https://docs.python.org/3/howto/sockets.html>", Acedido: 12-04-2020.
- [3] *Launching parallel tasks*, "<https://docs.python.org/3/library/concurrent.futures.html>", Acedido: 02-05-2020.
- [4] *What is an Elasticsearch Index?* "<https://www.elastic.co/pt/blog/what-is-an-elasticsearch-index>", Acedido: 28-03-2020.
- [5] *Bulk API*, "<https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-bulk.html>", Acedido: 14-04-2020.
- [6] *Tune for indexing speed*, "<https://www.elastic.co/guide/en/elasticsearch/reference/master/tune-for-indexing-speed.html>", Acedido: 15-04-2020.
- [7] *db_bench*, "<https://github.com/facebook/rocksdb/wiki/Benchmarking-tools>", Acedido: 12-05-2020.
- [8] *TPC-C*, "<http://www.tpc.org/tpcc/>", Acedido: 06-06-2020.