Suporte eficiente para pesquisas seletivas em MapReduce

Manuel Ferreira, João Paiva, Luís Rodrigues

INESC-ID, Instituto Superior Técnico, Universidade Técnica de Lisboa
{manuel.g.ferreira, joao.paiva, ler}@ist.utl.pt

Resumo Este artigo propõe e avalia um sistema destinado a suportar eficientemente pesquisas MapReduce que necessitam de processar apenas um sub-conjunto de todos os dados. O sistema proposto recorre a um formato de dados apropriado para suportar pesquisas seletivas e mecanismos de indexação de forma a melhorar a rapidez de acesso aos dados, encurtando significativamente a fase Map das execuções. Uma extensa avaliação experimental mostra que, ao evitar ler blocos irrelevantes, a nossa solução permite atingir uma melhoria de até 80 vezes quando comparada com a distribuição atual do Hadoop. Para além disso, o nosso sistema supera também outras concretizações do MapReduce que recorrem a variantes das técnicas propostas. O trabalho apresentado é um sistema de código-fonte aberto e está disponível para ser descarregado.

1 Introdução

Atualmente, existe uma crescente necessidade de analisar grandes volumes de dados, uma tarefa que requer infraestruturas de computação e armazenamento especializadas. Em particular, é necessário suportar programas paralelos, que se podem executar em centenas de servidores, de forma a produzir resultados úteis num intervalo de tempo aceitável. O paradigma MapReduce [2] tem-se tornado fundamental para paralelizar computações complexas sobre grandes volumes de dados. Concretizações do MapReduce, tais como o Hadoop [9], tornaram-se normas "de facto" para suportar o processamento de grandes volumes de dados.

Originalmente, o MapReduce foi concebido para trabalhos, como indexação de páginas da Internet, que processam o conjunto de dados na sua totalidade [17]. Contudo, à medida que o leque de aplicações para MapReduce cresce, este é frequentemente usado para executar pesquisas que dizem respeito apenas a uma fração do mesmo [3,4]. Por exemplo, fornecedores de serviços de telecomunicações podem obter uma fonte adicional de rendimento, recorrendo à venda da análise dos seus dados a terceiros para pesquisas de mercado [20].

Infelizmente, as concretizações MapReduce atuais não estão adaptadas para suportar este tipo de operações. De acordo com o modelo MapReduce, as entradas dos dados relevantes para a computação devem ser selecionadas durante a fase de Map, de forma a serem de seguida enviadas para os Reducers. Para tal, a tarefa Map é forçada a ler todos os dados, mesmo que os relevantes para a pesquisa sejam apenas uma pequena fração da totalidade. Este processo pode

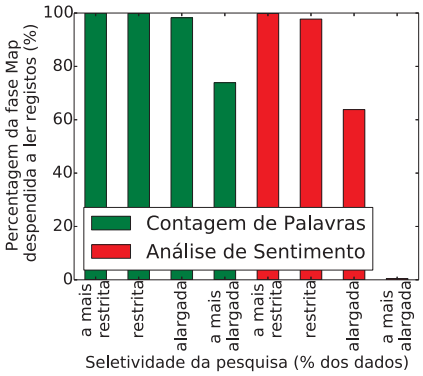

Figura 1: Custo da leitura de dados em função da duração da fase Map, dependendo do tipo de trabalho e da seletividade da pesquisa.

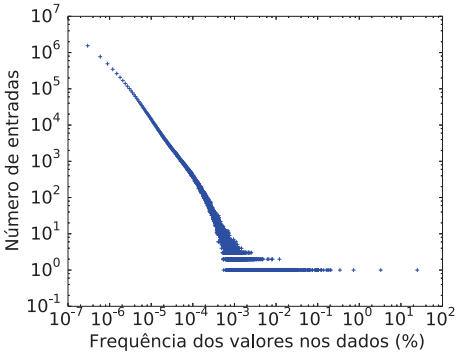

Figura 2: Distribuição de frequências dos valores do atributo "localização do utilizador" num conjunto de dados real do Twitter.

consumir uma porção significante de todo o processo MapReduce. A Figura 1 apresenta o custo de ler os dados iniciais, em comparação com o custo total da fase Map, para dois tipos de trabalhos e de pesquisas seletivas, usando o Hadoop sem alterações, e recorrendo a um conjunto de dados real do Twitter. O trabalho "contagem de palavras" é um exemplo comum de execuções MapReduce, e consiste em contar a frequência das palavras num texto; O trabalho "sentimento"calcula o grau de felicidade dos utilizadores do Twitter [1, 7], sendo um exemplo realista dos diversos tipos de programas que têm sido desenvolvidos para extrair informação a partir de dados do Twitter [13, 16]. Como se pode ver, ao pesquisar por qualquer item à exceção dos mais comuns, o tempo de ler registos ocupa entre 60 a 99% do tempo total da fase Map. No caso dos valores mais comuns, como a pesquisa é pouco seletiva, a maior parte do tempo é investido no processamento dos dados. De facto, no trabalho relacionado é possível identificar vários problemas cujas fases Map representam uma grande porção do processamento total e que portanto justificam a otimização desta tarefa como uma área de investigação interessante [3, 4, 12, 15, 18, 19, 21].

Uma característica comum aos conjuntos de dados reais de grande volume, e que motiva o nosso trabalho, é que a distribuição das frequências dos valores para qualquer atributo segue, tipicamente, uma distribuição fortemente enviesada tal como a distribuição Zipfiana [22]. Este facto é ilustrado na Figura 2, que apresenta a distribuição de frequências do atributo "localidade do utilizador" na amostra de um conjunto de dados real do Twitter usada na avaliação do sistema proposto. Este tipo de distribuições é particularmente favorável para ser indexado, uma vez que, apesar de uma pequena percentagem serem valores que aparecem muito frequentemente, a maioria é incomum. Isto significa que um índice criado usando estes valores permite selecionar apenas pequenas partes do conjunto de dados para cada pesquisa, resultando em pesquisas eficientes.

Neste artigo, mostramos como o tempo da fase Map pode ser substancialmente reduzido. Propomos e avaliamos um sistema que se baseia na combinação de técnicas que suportam eficientemente pesquisas seletivas de MapReduce. No-

meadamente, combinamos o uso de um formato de dados apropriado com ferramentas de indexação para melhorar a rapidez de acesso aos dados e encurtar significativamente a fase Map das execuções. Uma extensiva avaliação experimental do sistema mostra que, ao evitar ler blocos irrelevantes, é possível atingir uma melhoria até 80 vezes quando comparada com a distribuição atual do Hadoop. Para além disso, o nosso sistema supera também outras concretizações do MapReduce que recorrem a variantes das técnicas propostas.

O artigo está estruturado da seguinte forma. A Secção 2 contém uma curta introdução ao MapReduce. As Secções 3 e 4 descrevem a arquitetura e os algoritmos usados pelo sistema, enquanto que a Secção 5 descreve alguns pormenores relevantes de concretização. A Secção 6 fornece uma extensa avaliação experimental do sistema. Uma comparação entre o nosso sistema e o trabalho relacionado é fornecida na Secção 7 e a Secção 8 conclui o artigo.

2 Breve Descrição do MapReduce

Um programa MapReduce designa-se por "*trabalho*" (*job*) e é composto por um conjunto de tarefas, algumas das quais aplicam a função Map aos dados de entrada (tarefas Map), enquanto que outras aplicam a função Reduce aos resultados intermédios (tarefas Reduce). Alguns nós no sistema podem atuar como Mappers, Reducers ou ambos. Todo o processo é gerido por um mestre centralizado. Os passos principais de um programa MapReduce são os seguintes:

1. **Atribuição de tarefas**: Os dados iniciais são divididos em conjuntos de dados denominados *splits*, para cada um deles, o mestre cria uma tarefa Map e que de seguida atribui a um nó.
2. **Leitura dos Dados de Entrada**: O primeiro passo de uma tarefa Map é extrair os pares chave/valor dos dados contidos no split.
3. **Função Map**: Cada par chave/valor é passado para a função Map definida pelo utilizador e pode gerar zero, um ou mais pares chave/valor intermédios.
4. **Fase de Mistura**: Os pares chave/valor intermédios são atribuídos aos Reducers através de uma função de partição de tal maneira que todos os pares chave/valor intermédios com a mesma chave intermédia sejam processados pela mesma tarefa Reduce e, consequentemente, pelo mesmo Reducer.
5. **Ordenação & Agrupamento**: Cada Reducer ordena os dados intermédios de forma a agrupar todos os pares pela chave intermédia.
6. **Função Reduce**: Cada Reducer passa a chave intermédia e o conjunto dos valores intermédios correspondentes para a função Reduce definida pelo utilizador. O resultado dos Reducers é guardado no sistema de ficheiros global.

O Apache Hadoop [9] é a mais popular concretização com código-fonte aberto do mecanismo MapReduce. Inspirado na concretização da Google, o Hadoop também recorre a duas camadas diferentes: o HDFS, uma camada de armazenamento distribuído responsável por guardar os dados de forma persistente em vários nós; e o mecanismo MapReduce do Hadoop, responsável pela computação dos programas MapReduce.

Camada de Armazenamento: O Hadoop DFS (HDFS) é um sistema de ficheiros distribuído composto por três entidades: um NameNode, um SecondaryNameNode e um ou mais DataNodes. O NameNode é responsável por manter os meta-dados dos ficheiros guardados no sistema de ficheiros distribuído. De modo a recuperar esses meta-dados no caso de o NameNode falhar, o SecondaryNameNode mantém uma cópia do último ponto de controlo dos meta-dados. Cada ficheiro no HDFS é dividido em vários blocos de tamanho fixo (tipicamente configurados com 64MB cada), e cada bloco pode ser guardado em qualquer um dos nós de armazenamento. De modo a aumentar a disponibilidade, o Hadoop replica cada bloco usando um fator configurável (que por omissão é 3).

Camada de Processamento: As entidades envolvidas na camada de processamento são um mestre, chamado JobTracker, e um ou mais escravos, denominados TaskTrackers. O papel do JobTracker é coordenar todos as execuções MapReduce no sistema e atribuir tarefas aos TaskTrackers que, periodicamente, reportam ao JobTracker o progresso das tarefas que estão a executar em cada momento. O Hadoop recorre ao seguinte método de escalonamento de tarefas: Inicialmente, são atribuídas tarefas a cada escravo para processar os seus dados locais. No entanto, assim que o escravo terminar essas tarefas, podem-lhe ser atribuídas novas tarefas que o façam processar dados guardados noutros nós.

3 Mecanismos Principais do Sistema Proposto

Apresentamos agora o sistema proposto, que incorpora um conjunto coerente de técnicas com o intuito de melhorar a fase Map dos *trabalhos* que apenas manipulam parte dos dados. O sistema permite minimizar os dados lidos durante a fase de leitura dos dados de entrada, recorrendo aos seguintes mecanismos:

Formato dos Dados: Os dados são organizados por forma a promover a localidade. O sistema guarda as tabelas por colunas e não por linhas.

Agrupamento dos Dados: O sistema agrupa em cada nó dados semelhantes, para melhorar a eficácia do mecanismo de indexação.

Indexação: Cada nó cria índices locais dos dados que são guardados localmente. Os índices são criados e mantidos para os atributos mais relevantes.

Nas próximas secções apresentamos mais pormenores sobre cada um destes mecanismos.

3.1 Formato dos Dados

Num sistema MapReduce, os dados iniciais podem ser modelados como um conjunto de tabelas, onde cada tabela é composta por múltiplas colunas (também designadas por "atributos"). Um destes atributos, tipicamente o primeiro, é designado de chave, e identifica o registo, ou linha, da tabela. As tabelas são, normalmente, muito grandes e têm que ser particionadas e guardadas em múltiplos blocos de dados. Existem duas abordagens principais para projetar o conteúdo da tabela em blocos: orientação por linhas e orientação por colunas.

Figura 3: Ilustração dos dados organizados por grupos de linhas.

No formato com orientação por linhas, todos os atributos de um registo são guardados sequencialmente, e vários registos são guardados contiguamente em disco. Sistemas baseados em linhas são adequados para processar pesquisas onde é necessário processar todos os atributos dos registos ao mesmo tempo, uma vez que um registo inteiro pode ser lido através de um único acesso ao disco.

Por outro lado, trabalhos recentes [4, 6] têm demonstrado que um formato com orientação por colunas é adequado para *trabalhos* seletivos que acedem a um número reduzido de colunas, uma vez que as colunas que não são relevantes para o resultado pretendido não são carregadas do disco e filtradas, reduzindo, assim, o tempo de execução de um *trabalho* MapReduce.

Infelizmente, a cada coluna dos dados pode corresponder um tipo de dados diferente, o que implica que os valores para cada atributo poderão ocupar espaços muito diferentes em disco. Assim, uma partição simples dos dados por colunas poderia levar a que os blocos das colunas ficassem desalinhados, complicando o processo de reconstruir os registos a partir dos dados particionados. Nós evitamos este problema através de uma primeira divisão horizontal dos dados, criando *grupos de linhas* e, só depois, cada grupo de linhas é dividido verticalmente por colunas, cada uma delas guardada num ficheiro diferente (Figura 3). Desta forma, um ficheiro correspondente a um dado atributo é apenas lido quando uma pesquisa referencia o atributo correspondente, evitando ler os dados de outros atributos irrelevantes para a pesquisa. Uma vez que, por omissão o HDFS coloca os blocos de forma aleatória em todos os nós de dados, o sistema também inclui uma *Política de Colocação de Blocos* para garantir que todos os blocos correspondentes ao mesmo grupo de linhas são colocados no mesmo nó. Isto permite ao nosso sistema usar um formato de dados por colunas sem ter que ler dados de vários nós quando um *trabalho* necessita de registos inteiros e estes têm que ser reconstruidos a partir de várias colunas.

3.2 Agrupamento

A componente de agrupamento do sistema é responsável por reescrever os dados iniciais num formato que favorece os índices orientados por colunas. De facto, os índices por colunas podem ser bastante ineficientes sem o uso de algum tipo de agrupamento: Num volume de dados grande, a frequência da maior parte dos valores será significativamente maior que o número de grupos de linhas em que os dados foram partidos, o que significa que provavelmente, qualquer que seja o valor, este ocorrerá na maior parte dos grupos de linhas.

De forma a assegurar que qualquer que seja o valor, este esteja presente no menor número possível de blocos, o nosso sistema lê todos os dados, identifica os registos que partilham o mesmo valor para o atributo indexado, e retorna-os numa ordem tal que os registos com o mesmo valor fiquem contíguos em um ou mais blocos. Esta reordenação é feita *localmente*, para cada nó no sistema. O uso do agrupamento local tem duas vantagens significativas em relação a uma ordenação global: Faz com que a ordenação não seja cara, e preserva o bom balanceamento de carga alcançado pela distribuição inicial dos dados do HDFS.

3.3 Indexação

O nosso sistema usa indexação para evitar carregar todos os dados em memória durante a fase Map. Tal é conseguido usando índices como indicador se um bloco é relevante para o *job* que está a ser processado.

Os índices utilizados pelo nosso sistema são completamente descentralizados no sentido em que cada nó constrói os seus próprios índices, baseado apenas nos dados que guarda. Esta decisão de desenho simplifica a criação dos índices, e faz com que os acessos aos índices locais seja rápido. Em termos de estrutura, os índices são, na verdade, índices invertidos, em que cada entrada projeta o valor do atributo para a sua localização nos dados. Assim, usamos um par $\langle$*IDGrupoDeLinhas, [deslocação-coluna1, . . . , deslocação-colunaN]*$\rangle$. O primeiro elemento do par aponta para o primeiro grupo de linhas que contém esse valor do atributo. O segundo elemento é uma lista com tamanho igual ao número de colunas da tabela, em que cada entrada corresponde ao deslocamento desse atributo dentro do bloco correspondente a essa coluna.

De forma a poupar memória durante a execução de uma pesquisa, os índices são guardados em armazenamento persistente juntamente com os dados de forma particionada, de maneira a que apenas um pequeno conjunto de pares de índice sejam guardados em cada ficheiro. Assim, quando uma tarefa é atribuída a um nó, apenas os pares correspondentes ao valor pesquisado são carregados pela tarefa Map para memória de forma a determinar se os blocos correspondentes ao split da tarefa Map devam ser lidos. De modo a tornar a pesquisa dos índices ainda mais eficiente, o sistema mantém uma cache dos valores recentemente pesquisados usando uma política de limpeza baseada no princípio de menos recentemente usado. Este desenho simplifica a gestão dos índices, uma vez que permite a um nó manter os pares de índice em memória durante todo o *trabalho*, sem ter que pedir informação sobre a conclusão do *trabalho* a outros nós (já que, no MapReduce, esta informação só está disponível no mestre, o JobTracker).

4 Outros Aspetos do Sistema

Na secção anterior descrevemos os mecanismos principais que compõem o sistema proposto. Nesta secção discutimos questões complementares, tais como a replicação, a validação de blocos antes de os transferir, e o passo de pré-processamento necessário para transformar os dados para serem processados pelo MapReduce modificado.

Replicação: O Hadoop suporta replicação, permitindo que cada bloco seja replicado por (um número configurável de) R nós, de forma que quando um nó falha, os dados possam ser recuperados a partir dos restantes nós do sistema. O sistema proposto também permite que cada bloco seja replicado com semelhantes vantagens. Contudo, uma vez que cada nó agrupa os dados que guarda, a replicação no nosso sistema tem que ser feita ao nível do nó (i.e., todos os dados armazenados por um nó são replicados em conjunto). Para além disso, à semelhança de outros sistemas do estado da arte [4, 19], o nosso sistema também permite que cada nó agrupe os dados de acordo com um atributo diferente, podendo assim otimizar pesquisas sobre mais do que um atributo.

Pré-Validação na Transferência de Blocos: Relativamente aos índices, relembramos que o nosso sistema não recorre a um índice centralizado e que, para além disso, o modelo MapReduce permite que os nós leiam blocos remotos. Ao processar uma pesquisa seletiva, isto pode levar a que blocos sem informação relevante sejam transferidos pela rede. Na nossa concretização, introduzimos uma alteração ao Hadoop para evitar esta transferência supérflua. Antes de transferir um bloco para outro nó, o nó que aloja os dados carrega o seu índice e faz uma pesquisa de modo a determinar se o bloco é relevante ou não. Caso o bloco não seja relevante, a sua transferência é evitada. Os resultados que apresentamos na Secção 6 recorrem a esta otimização. No entanto, realçamos o facto de que o desenho proposto não está dependente desta decisão de concretização.

Pré-Processamento: O sistema proposto requer a execução de uma fase de pré-processamento dos dados para construir os índices e reformatar os blocos de dados, dividindo as linhas em colunas e agrupando dados semelhantes. Já que este pré-processamento não envolve quaisquer transferências de dados entre nós, pode ser feito eficientemente e até aproveitando a própria infraestrutura do MapReduce. Após os dados serem carregados para o HDFS, executamos um *trabalho* MapReduce de pré-processamento configurado de tal forma que cada nó atua como Mapper assim como Reducer. Este *trabalho* percorre todas as linhas de todos os ficheiros de entrada (i.e. todos os registos guardados pelo nó) e, na fase Map, retorna um par $\langle (ID, valor), Lista < campo > \rangle$, associando o próprio registo, formatado como uma lista de valores dos campos, com uma chave intermédia. Esta chave intermédia assegura (juntamente com uma função de partição que tem em conta o *ID* do nó) que existe uma relação 1:1 entre os Mappers e os Reducers, de forma a manter o balanceamento de dados inicialmente criado pelo HDFS. De modo a permitir que o Reducer faça o agrupamento dos registos, a chave intermédia contém também o valor do atributo contido no registo pelo qual os dados são agrupados. Uma vez que o atributo usado para o agrupamento dos dados deve ser definido na fase de pré-processamento, é da responsabilidade do utilizador selecionar qual o atributo mais relevante para agrupar e indexar os dados.

Na fase de Reduce, cada Reducer recebe vários conjuntos de registos, agrupados pelo valor do atributo. A tarefa do Reducer é emitir cada campo de cada registo para um bloco HDFS diferente. Note-se que, uma vez que cada coluna

pode conter um tipo de dados diferente, o tamanho de cada bloco (de coluna) pode também aumentar a um rácio diferente ao longo do pré-processamento. Assim, para garantir que todos os blocos correspondentes a um grupo de linhas ficam alinhados entre si, quando o conteúdo de uma coluna atingir a capacidade de um bloco HDFS, é criado um novo grupo de linhas. Este mecanismo segue uma ideia semelhante a outros sistemas com orientação por colunas do estado da arte [6,8,11]. Durante este processo, o sistema memoriza a posição onde cada valor diferente começa, de modo a ir preenchendo os índices.

Embora nesta fase de pré-processamento seja necessário transformar todos os dados do sistema, salientamos que este custo é pago apenas ao carregar os dados. Assim, assumimos que os ganhos obtidos após analisar os dados multiplas vezes serão superiores ao custo do pré-processamento.

5 Concretização do Sistema Proposto

O sistema proposto foi concretizado como um conjunto de extensões ao Hadoop e o protótipo está disponível para ser descarregado[1]. As extensões consistem em, aproximadamente, 6600 linhas de código. Nesta secção, realçamos as alterações mais importantes feitas ao Hadoop para suportar as funcionalidade propostas.

Relativamente ao carregamento dos dados para o HDFS, o sistema precisa de assegurar que os ficheiros relativos ao mesmo grupo de linhas são colocados no mesmo nó, de modo a permitir uma reconstrução dos registos local, caso seja necessário ler mais do que um atributo. Por omissão, o HDFS coloca os blocos de forma aleatória por todos os nós, portanto o nosso prototipo inclui uma política de colocação de blocos com conhecimento dos grupos de linhas, de modo a assegurar que os ficheiros relativos ao mesmo grupo de linhas sejam colocados no mesmo nó. Não foram necessárias quaisquer modificações ao núcleo do Hadoop para implementar esta funcionalidade, uma vez que o HDFS suporta a implementação de políticas de colocação de blocos personalizadas.

Os dados são divididos em splits logo após a submissão de um *trabalho*, de acordo com o `InputFormat` associado ao mesmo. Os splits gerados são depois processados pelas tarefas Map. Uma vez que o sistema proposto é baseado na manipulação de ficheiros, o `InputFormat` mais apropriado é o `FileInputFormat` do Hadoop, por ser a classe base para todos os `InputFormats` baseados em ficheiros. Contudo, o `FileInputFormat` considera todos os ficheiros de entrada, criando um split por cada bloco HDFS. Assim, definimos um `InputFormat` que tem informação acerca dos grupos de linhas, criando, por isso, apenas um split por cada grupo de linhas, em vez de um por cada bloco HDFS.

Simultaneamente, o `FileInputFormat` instancia também o `RecordReader`, responsável por gerar os pares chave/valor a partir dos dados iniciais e passá-los, um a um, para a função Map. Implementámos um `RecordReader` com a capacidade de permitir o acesso local ao índice para verificar se um dado grupo de linhas contido no split é relevante. Se tal se verificar, o `RecordReader` obtém

[1] O protótipo pode ser descarregado de: `https://github.com/shortmap/shortmap`

o posicionamento da primeira linha relevante de onde a leitura irá começar até chegar ao final do grupo de linhas ou até ler uma entrada que já não satisfaz a procura. Para suportar que o sistema evite transferências de blocos irrelevantes, adicionámos também uma nova rotina que é chamada pelo nó que está a processar um dado grupo de linhas. Esta rotina permite comunicar com o nó que guarda os dados, para que este consulte o seu índice local e devolva apenas a parte relevante do grupo de linhas, se existir. Caso contrário, o nó devolve apenas uma resposta negativa.

É relevante notar que, apesar destas alterações melhorarem significativamente o desempenho do MapReduce, (comparando com a implementação padrão do Hadoop), a maioria destas não requer quaisquer alterações ao core do Hadoop. Dado que o Hadoop é bastante extensível, a maioria destas capacidades podem ser implementadas apenas através do acrescento de novas classes e respetiva configuração nos ficheiros de configuração de sistema e do *trabalho*.

6 Avaliação

Nesta secção, começamos por mostrar como cada componente do sistema proposto contribui para o seu resultado como um todo, comparando com outras soluções do estado da arte. Na conclusão da secção apresentamos finalmente a comparação entre o nosso sistema e uma versão não modificada do Hadoop. Todas as experiências foram feitas usando um conjunto de 20 máquinas virtuais de Ubuntu Linux com kernel 2.6.32-33-server, executadas sobre 10 máquinas físicas a correr Xen. As máquinas físicas estão equipadas com dois processadores 2.13 GHz Quad-Core Intel(R) Xeon(R) E5506 e 40 GB de RAM, interligadas através de rede Ethernet Gigabit privada. Todos os valores apresentados foram obtidos através da média de pelo menos 3 execuções do sistema. Para o nosso conjunto de dados e configuração de hardware, o Hadoop sem modificações demora entre 1 e 3 horas a processar cada pesquisa.

Todos os resultados apresentados nesta secção recorreram a uma amostra de dados do Twitter [10], colecionada entre Maio e Setembro de 2013. Estes dados contêm 325,333,833 tweets, e que correspondem a 988GB de dados que, quando comprimidos com gzip, ocupa um espaço total de 161GB. Os tweets encontram-se num formato JSON, contendo cada um deles 23 atributos (e.g. um identificador, data de criação, *hashtags*, a mensagem de texto) entre eles um objeto JSON com mais 38 atributos sobre o utilizador que gerou o tweet, tais como o seu idioma, localidade, identificador, etc. Devido a restrições impostas pelo Twitter, não nos é possível tornar a amostra de dados pública, mas é possível obter um conjunto de dados semelhante recorrendo à API do Twitter.

Os padrões de acesso aos dados usado nesta avaliação (i.e. as pesquisas feitas no sistema) refletem um cenário em que um fornecedor de serviços disponibiliza a sua infraestrutura aos seus clientes para estes fazerem pesquisas seletivas. Exemplos destas pesquisas são procurar por um grupo demográfico ou um local em específico, ou por uma "hashtag" associada ao tweet. Usámos dois tipos diferentes de *trabalhos* seletivos MapReduce; Cada padrão de carga permite analisar

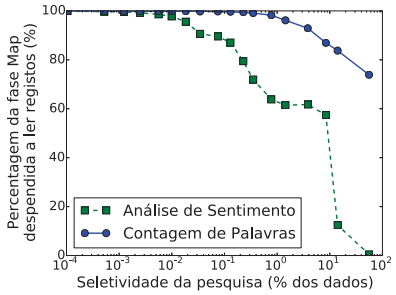

Figura 4: Custo relativo à leitura de dados, normalizado à duração da tarefa Map, dependendo da seletividade das pesquisas.

diferentes aspetos do sistema, de acordo com a quantidade de processamento necessária pela função Map correspondente. O primeiro tipo de *trabalho* consiste em aplicar uma pesquisa seletiva aos dados de modo a recolher apenas tweets que usem um idioma específico, aplicando depois uma simples contagem de palavras ao texto. Esta análise serve como base para a comparação com o segundo tipo de *trabalho*, mais complexo e realista, que consiste em calcular o sentimento dos utilizadores a partir dos seus tweets [1, 7]. Este *trabalho* ilustra o processamento de grandes volumes de dados necessário para extrair informação a partir de dados do Twitter [13, 16], tendo uma exigência significativamente maior em termos de processamento que a contagem de palavras.

Para ilustrar melhor a diferença entre os dois tipos de *trabalhos*, a Figura 4 mostra a percentagem do tempo da fase Map que o nosso sistema gasta na leitura dos registos de disco, executando a análise de sentimento e a contagem de palavras. Estes valores, à semelhança de outros nesta secção, são apresentados em função da seletividade da pesquisa, i.e., a que percentagem dos dados corresponde o valor pesquisado. Uma vez que o *trabalho* de contagem de palavras necessita de um menor processamento dos dados, gasta uma maior fração (de pelo menos 73%) da fase Map a ler os registos necessários. *Trabalhos* mais leves são, portanto, frequentemente limitados pelo tempo necessário para examinar os dados de entrada. Contrariamente, a análise de sentimento requer um maior processamento na função Map. Isto é particularmente visível ao processar o texto de utilizadores cujo idioma é o mais comum nos dados. Neste caso, uma vez que existe um grande número de mensagens de texto para processar, a maior parte do tempo da fase Map é gasta a processar os registos e não a lê-los de disco.

6.1 Comparação com Armazenamento por Linhas com Indexação

Uma das principais preocupações do nosso sistema é o formato dos dados. De modo a alcançar uma maior eficiência na leitura dos dados, usamos armazenamento por colunas. Para avaliar o efeito desta decisão de desenho, nesta secção comparamos o nosso sistema com sistemas que recorrem simultaneamente ao armazenamento por linhas e à utilização de índices.

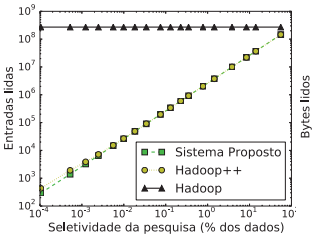

Figura 5: Número de entradas lido pelo Hadoop, Hadoop++ e o sistema proposto

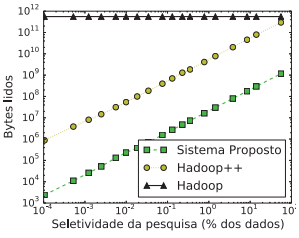

Figura 6: Número de bytes lidos pelo Hadoop, Hadoop++ e o sistema proposto.

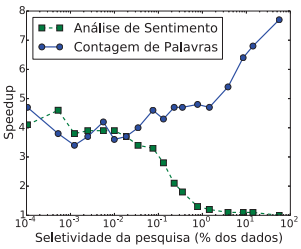

Figura 7: Speedup do nosso sistema sobre o Hadoop++.

Entre este tipo de sistema, uma solução do estado da arte comparável com o nosso sistema é o Hadoop++, [3]. O Hadoop++ não só usa um formato de dados orientado por linhas, assim como cria índices para cada item contido em cada bloco. O Hadoop++ ordena cada bloco internamente, e cria um índice por cada um, que depois é anexado ao bloco. Este índice é carregado em tempo de pesquisa, e usado para determinar que parte do bloco deve ser lida para responder a uma dada pesquisa. Uma vez que o código do Hadoop++ não está publicamente disponível, implementámos um prototipo deste sistema seguindo a especificação fornecida em [3].

Para melhor ilustrar as diferenças de desenho entre o nosso sistema, o Hadoop++ e o Hadoop, as Figuras 5 e 6 mostram o número de bytes e entradas lidas por cada solução. O Hadoop++ lê muito menos bytes e entradas que o Hadoop, e praticamente o mesmo número de entradas que o nosso sistema (uma vez que os índices permitem saltar blocos e entradas irrelevantes). Identicamente ao nosso sistema, o número de bytes lidos depende de quão comum é o valor procurado. Em termos de dados lidos, a principal diferença entre os dois sistemas é que, uma vez que o Hadoop++ usa armazenamento por linhas, tem que ler todas as colunas dos registos, enquanto que o sistema proposto apenas necessita ler os blocos correspondentes às colunas relevantes.

A Figura 7 mostra como estas decisões se refletem em termos de desempenho do sistema, capturando o *speedup* do sistema proposto sobre o Hadoop++ ao pesquisar por valores a que correspondem diferentes seletividades. Tal como seria de esperar dada a análise dos dados lidos, para todas as execuções o sistema proposto exibe speedups sobre o Hadoop++. Quando a pesquisa corresponde ao valor mais comum da amostra e é feita uma computação complexa (i.e. a análise de sentimento), o desempenho de ambos os sistemas tende a ser semelhante. Este facto deve-se à maior parte do tempo ser utilizado na computação do sentimento e não na leitura de dados. Por outro lado, para os *trabalhos* mais seletivos, o sistema apresenta um speedup de até 5 vezes, devido a ler menos dados de disco.

Curiosamente, ao executar a contagem de palavras, revela-se um efeito inesperado e o speedup do nosso sistema aumenta à medida que as pesquisas se tornam menos seletivas. Este resultado é devido ao facto do Hadoop++, para este tipo de pesquisas pouco seletivas, ser obrigado a carregar todos os blocos da amostra de dados. De facto, apesar de apenas cerca de 50% dos registos serem

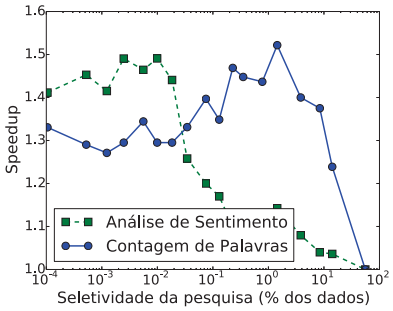

Figura 8: Speedup do nosso sistema sobre o SIOC.

relevantes para estas pesquisas, todos os blocos são relevantes uma vez que os registos estão distribuídos de forma uniforme pelos blocos. Globalmente, o nosso sistema alcança um speedup médio de 4.4 sobre o Hadoop++ na contagem de palavras e 2.4 para a análise de sentimento.

6.2 Comparação com Armazenamento por Colunas e Indexação

Vários sistemas do estado da arte optam por usar um formato por colunas juntamente com índices por bloco [4, 19]. Nesta secção, estudamos como o nosso sistema se compara com uma variante do Hadoop++ que, em lugar de guardar os dados por linhas, guarda-os em grupos de linhas partidos por colunas (como no nosso sistema), e indexa cada grupo de linhas individualmente (contrariamente ao sistema proposto, que agrupa os dados e indexa-os por nó). Esta avaliação permite perceber a influência do mecanismo de agrupamento do nosso sistema. Uma vez que o código do LIAH [19] ou do HAIL [4] não estão disponíveis, este protótipo representa uma versão simplificada destes sistemas; por comodidade, chamamos-lhe SIOC (Sistema com Indexação e Orientação a Colunas).

Uma vez que tanto o SIOC como o sistema proposto leem os mesmos dados de entrada, esses gráficos foram omitidos. A Figura 8 apresenta o speedup do sistema proposto sobre o SIOC. Como observado anteriormente, uma vez que as pesquisas menos seletivas da análise de sentimento são limitadas pelo tempo de processamento, o sistema proposto não apresenta uma vantagem significativa sobre as outras soluções para este cenário. Ainda assim, no global deste *trabalho*, o sistema atinge uma melhoria de até 50% em relação ao prototipo do SIOC, apresentando uma média de 25% de speedup. Dois fatores contribuem para este resultado. Em primeiro lugar, recorrer a um índice por bloco requer que o sistema carregue vários índices de disco durante a execução do *trabalho*, contrariamente ao nosso sistema que apenas carrega índices parciais na primeira tarefa associada ao *trabalho*. Em segundo lugar, e de forma mais relevante, uma vez que os dados não se encontram agrupados, vários blocos satisfazem a pesquisa, o que requer que o sistema os tenha que carregar do disco, e obter o primeiro valor relevante que poderá não estar no inicio do bloco. Por oposição, na maior parte dos casos o nosso sistema apenas necessita ler um único bloco.

Para a contagem de palavras, o custo de não usar agrupamento de dados torna-se mais visível. Contrariamente aos resultados com a análise de sentimento, o speedup do sistema proposto sobre o SIOC aumenta, de facto, à medida que a seletividade da pesquisa diminui. Estes resultados estão fortemente relacionados com o número de blocos que satisfazem a pesquisa. Para o nosso sistema, o número de blocos relevantes é proporcional à seletividade da pesquisa. Contudo, num sistema que não recorre a agrupamento de dados, o número de blocos relevantes cresce super-linearmente com a seletividade da pesquisa, uma vez que os registos que contêm o valor procurado estão distribuídos uniformemente por todos os blocos. Este efeito torna-se particularmente notório nas pesquisas menos seletivas, uma vez que o número de blocos relevantes no nosso sistema permanece muito pequeno, enquanto que no sistema sem agrupamento todos os blocos são relevantes. Isto também explica o porquê de quando pesquisamos pelo valor mais comum, o speedup do nosso sistema seja perto de 1: nesta situação, em ambos os sistemas, muitos blocos contêm elementos relevantes para a pesquisa. O custo de leitura dos muitos registos relevantes (igual em ambos os sistemas) é muito superior ao custo de saltar para a posição relevante dentro de cada bloco. Note-se também que para o item mais raro, o speedup do nosso sistema é menor do que nas pesquisas seguintes: isto deve-se ao facto de o valor estar presente em apenas um bloco em ambos os sistemas, resultando assim num desempenho semelhante para ambos (com uma pequena vantagem do sistema proposto, por ler apenas um único índice).

6.3 Comparação com o Hadoop

Nesta secção, avaliamos o desempenho global do sistema proposto, comparando execuções do nosso protótipo com uma versão não modificada do Hadoop. Contrariamente ao Hadoop, o nosso sistema não requer uma análise completa de todos os dados ao executar uma pesquisa seletiva. Como se pode ver na Figura 9, o sistema lê significativamente menos dados que o Hadoop. A Figura 10 mostra que o nosso sistema também lê consideravelmente menos entradas, o que contribui para reduzir o custo de processamento.

Mesmo quando pesquisando pelo valor mais comum, que corresponde a aproximadamente metade das entradas, o nosso sistema continua a ler uma menor quantidade de dados, uma vez que apenas lê as colunas relevantes para a pesquisa. O impacto desta melhoria é mais notório em *trabalhos* com uma maior fração de tempo despendido a ler os registos, uma vez que é esta a parte da fase Map que é encurtada.

A Figura 11 apresenta o speedup do sistema sobre o Hadoop, dependendo da seletividade da pesquisa. As consequências do sistema ler consideravelmente menos dados que o Hadoop podem ser facilmente observadas nestes resultados, uma vez que o nosso sistema atinge pesquisas mais rápidas de quase até duas ordens de magnitude. Como esperado pela análise dos dados lidos (Figura 9), a vantagem do nosso sistema é menos significativa ao pesquisar por valores mais comuns, uma vez que a diferença entre os dados lidos pelo sistema e pelo Hadoop

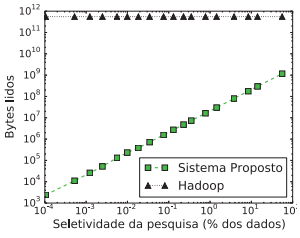

Figura 9: Comparação entre os dados lidos pelo Hadoop e pelo nosso sistema.

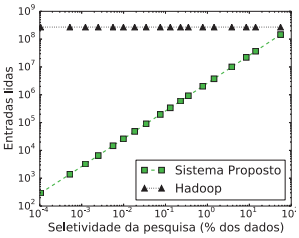

Figura 10: Comparação entre o número de registos lidos pelo Hadoop e pelo nosso sistema.

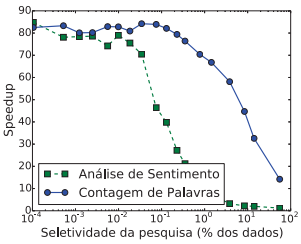

Figura 11: Speedup do nosso sistema sobre o Hadoop sem modificações.

encurta. Ainda assim, mesmo ao pesquisar pelo valor mais comum, para a contagem de palavras, o nosso sistema alcança um speedup de 14.2 vezes enquanto lê 460 vezes menos dados e 1.79 vezes menos entradas e processa o mesmo número de entradas. Neste cenário, uma vez que muitas das entradas precisam de ser processadas, os custos de processamento, apesar de reduzidos, limitam o speedup do sistema. Este facto é particularmente evidente quando olhamos para a análise de sentimento: o nosso sistema é apenas 1.2 vezes mais rápido que o Hadoop para o valor mais comum uma vez que, nessa situação, na prática o desempenho do sistema é limitado pelo custo de processamento. Para além disso, note-se que devido ao alto custo de processador associado à análise de sentimento, o speedup do sistema também decresce mais cedo (i.e. para valores menos comuns) do que na contagem de palavras. Estes resultados levam à conclusão de que, apesar de o sistema proposto ser particularmente adequado para *trabalhos* seletivos usando valores incomuns, o uso de um formato de dados por colunas permite-lhe atingir bons speedups em relação ao Hadoop mesmo ao pesquisar por valores comuns.

7 Trabalho Relacionado

De maneira semelhante ao sistema proposto, vários sistemas do estado da arte tomam partido de diferentes formatos de dados para melhorarem o desempenho do Hadoop. As três principais abordagens na literatura relacionadas com o formato dos dados são orientação por linhas [3,5,14], orientação por colunas [6,11] e o formato PAX [8,19]. Uma vez que o nosso sistema está desenhado para servir pesquisas sobre parte dos dados, a orientação por linhas não é o formato mais indicado para guardar os dados devido ao seu elevado custo ao ler apenas parte dos registos (ver Secção 6.1). Formatos com orientação por colunas, tal como o PAX, suportam leituras parciais dos dados. O nosso sistema faz uso de um formato por colunas, principalmente por o sistema estar desenhado para armazenar grandes volumes de dados comprimidos. Colocar todas as colunas num único ficheiro aumenta a entropia de informação, reduzindo assim os rácios de compressão [8]. Modificar o nosso sistema para usar o formato PAX envolveria uma modificação simples nos índices de modo a ter várias deslocações por cada bloco em vez de uma deslocação por cada bloco de colunas.

A indexação é uma técnica geralmente usada em SGBDs, que foi introduzida no MapReduce como uma forma para permitir ignorar registos ao ler os dados de entrada durante pesquisas seletivas. O Hadoop++ [3] propôs o Trojan Index, um tipo de índice criado por bloco e anexado a cada um destes. Isto permite ao MapReduce ler o índice antes de carregar o bloco do disco, e assim reduzir o número de registos recolhidos do disco, quando comparado com o Hadoop. Mais recentemente, trabalhos como o HAIL [4] e o LIAH [19] melhoraram este mecanismo reformatando cada bloco para o formato PAX, criando os índices durante a execução das pesquisas e de acordo com as mais efetuadas entre elas. Apesar de eficazes até um certo ponto, tal como mostramos na Secções 3.2 e 6.2, os sistemas tendem a não tirar completo partido dos índices quando estes não são combinados com uma reescrita dos blocos usando um agrupamento dos dados.

8 Conclusões

Neste artigo, descrevemos o desenho, implementação e avaliação experimental do um sistema que melhora significativamente o desempenho dos *trabalhos* MapReduce que apenas se interessam com um sub-conjunto dos dados. Os nossos resultados experimentais, obtidos com uma amostra de dados reais, mostram que o sistema proposto pode fornecer speedups de até 80 vezes sobre o Hadoop nas pesquisas sobre valores incomuns sem incorrer em penalização nos casos menos favoráveis. O sistema resultante supera também outros trabalhos relacionados que incorporaram algumas das técnicas adotadas. Este trabalho foi implementado como um sistema de código-fonte aberto e está disponível.

Como trabalho futuro, gostaríamos de tirar partido das frequências dos valores dos atributos disponíveis nos índices para melhorar o balanceamento de carga do sistema. Acreditamos que a informação das frequências pode ser aproveitada para aumentar o tamanho dos splits para pesquisas com alta seletividade, reduzindo, assim, o custo envolvido na criação e gestão das tarefas Map.

Agradecimentos: Os autores gostariam de agradecer ao A.P.Francisco e ao projeto "NetDyn: Understanding real large networks, from structure to dynamics"PTDC/EIA-EIA/118533/2010 por nos ter fornecido os dados do Twitter usados na avaliação deste trabalho. Este trabalho foi parcialmente suportado pela Fundação para a Ciência e Tecnologia (FCT) através do projeto PEPITA (PTDC/EEI-SCR/2776/2012), assim como através do financiamento pluri-anual do INESC-ID, via o programa PIDDAC, sob o projeto PEst-OE/EEI/LA0021/2013.

Referências

1. K. Bertrand, M. Bialik, K. Virdee, A. Gros, and Y. Bar-Yam. Sentiment in new york city: A high resolution spatial and temporal view. *arXiv preprint arXiv:1308.5010*, 2013.
2. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.
3. J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, September 2010.

4. J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only aggressive elephants are fast elephants. *CoRR*, abs/1208.0287, 2012.
5. M. Eltabakh, F. Özcan, Y. Sismanis, P. Haas, H. Pirahesh, and J. Vondrak. Eagle-eyed elephant: Split-oriented indexing in hadoop. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, pages 89–100, New York, NY, USA, 2013. ACM.
6. A. Floratou, J. M. Patel, E. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *Proc. VLDB Endow.*, 4(7):419–429, April 2011.
7. M. Frank, L. Mitchell, P. Dodds, and C. Danforth. Happiness and the patterns of life: a study of geolocated tweets. *Scientific reports*, 3, 2013.
8. Y. He, R Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ICDE '11, pages 1199–1208, Washington, DC, USA, 2011. IEEE Computer Society.
9. http://hadoop.apache.org.
10. https://dev.twitter.com/docs/platform objects/tweets.
11. A. Jindal, J.-A. Quiané-Ruiz, and J. Dittrich. Trojan data layouts: Right shoes for a running elephant. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 21:1–21:14, New York, NY, USA, 2011. ACM.
12. S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan. An analysis of traces from a production mapreduce cluster. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 94–103, Washington, DC, USA, 2010. IEEE Computer Society.
13. I. Kloumann, C. Danforth, K. Harris, C. Bliss, and P. Dodds. Positivity of the english language. *PloS one*, 7(1):e29484, 2012.
14. J. Lin, D. Ryaboy, and K. Weil. Full-text indexing for optimizing selection operations in large-scale data analytics. In *Proceedings of the Second International Workshop on MapReduce and Its Applications*, MapReduce '11, pages 59–66, New York, NY, USA, 2011. ACM.
15. M. Lin, L. Zhang, A. Wierman, and J. Tan. Joint optimization of overlapping phases in mapreduce. *Perform. Eval.*, 70(10):720–735, October 2013.
16. L. Mitchell, M. Frank, K. Harris, P. Dodds, and C. Danforth. The geography of happiness: Connecting twitter sentiment and expression, demographics, and objective characteristics of place. *PloS one*, 8(5):e64417, 2013.
17. L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Tech. Rep. 1999-66, Stanford InfoLab, November 1999.
18. B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: Locality-aware resource allocation for mapreduce in a cloud. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 58:1–58:11, New York, NY, USA, 2011. ACM.
19. S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead adaptive indexing in hadoop. *CoRR*, abs/1212.3480, 2012.
20. J. van der Lande. Big data analytics: Telecoms operators can make new revenue by selling data, http://www.analysysmason.com/, April 2013.
21. M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.
22. G. Zipf. *The Psychobiology of Language*. Houghton Mifflin, Boston, MA, 1935.