UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
CURSO DE GRADUAÇÃO EM TECNOLOGIA DA INFORMAÇÃO

Cleiane Clementino Bondade

Gabriel Guilherme Cavalcanti da Costa

Moisés Sousa Araújo

Raphael Cezar Sabbado

Relatório de desenvolvimento de rotinas de suporte ao modelo de programação MapReduce.

Natal
2024

SUMÁRIO

1 INTRODUÇÃO

Este relatório apresenta a implementação do modelo de programação MapReduce em Elixir, uma linguagem funcional conhecida por sua capacidade de lidar com concorrência de forma eficiente. O modelo MapReduce foi introduzido pelo Google em 2004 e auxilia no processamento de volumes massivos de dados através da paralelização de tarefas de maneira eficiente e escalável.

O modelo de programação MapReduce foi pensado para o tratamento de uma grande quantidade de dados, utilizando processamento paralelo e distribuído para otimizar o desempenho, provendo, ainda, uma abstração para facilitar o uso dessa estrutura, sem ser necessário para seu usuário compreender os aspectos relacionadas ao paralelismo e processamento distribuído que o atende.

Nesse contexto, tem-se que o usuário é o responsável por fornecer ao MapReduce, além dos dados, uma função map, que será aplicada a cada elemento do conjunto, gerando um conjunto de dados no formato chave-valor, e uma função reduce que deve operar sobre uma lista de valores associados a cada chave para produzir o resultado final.

De modo geral, as rotinas de suporte irão receber o conjunto de dados e particioná-lo em uma determinada quantidade de subconjuntos. Cada um desses subconjuntos é processado de maneira independente, aplicando o map fornecido pelo usuário para cada elemento. O resultado do map é agrupado pela chave, de forma que cada chave é associada a uma lista de valores. E, finalmente, aplica-se a função reduce do usuário para cada chave e lista associada, novamente com recursos de paralelismo.

A flexibilidade da solução possibilita sua implementação em qualquer linguagem de programação que suporte paralelismo. Isso abre portas para uma ampla gama de aplicações em diferentes domínios, desde análise de dados em tempo real até processamento de logs e indexação de documentos. No presente projeto, simulou-se a utilização do programa desenvolvido para a contagem de palavras de um texto, o qual é um problema comumente utilizado para exemplificar os conceitos desse modelo.

1.1 Tabela de participação

A implementação do programa foi dividida entre os membros do grupo, com a participação representada na tabela a seguir.

Nome	Participação	Exemplos de participação
Cleiane Clementino Bondade	10	Particionamento, Concorrência com Task,
Gabriel Guilherme Cavalcanti da Costa	10	Particionamento, Agrupamento por chave
Moisés Sousa Araújo	10	Particionamento, suporte ao Reduce do usuário
Raphael Cezar Sabbado	10	Particionamento, suporte ao Reduce do usuário

2 IMPLEMENTAÇÃO

O modelo MapReduce proposto inicialmente pelo Google e o modelo implementado nesse projeto possuem algumas diferenças relevantes. Além de ser uma versão muito simplificada de toda a estrutura proposta pela companhia, o modelo aqui retratado não fornece suporte específico para, por exemplo, o processamento paralelo e distribuído, se limitando a utilizar mecanismos de concorrência. Ademais, não há implementação de tolerância a falhas e outras medidas de segurança que precisariam estar presentes em uma solução completa.

Outra diferença que cabe ser apontada é que a função map fornecida pelo usuário recebe somente um parâmetro, diferentemente da proposta do Google na qual a função recebe dois, chave e valor. A empresa utiliza o parâmetro da chave para, por exemplo, identificar os arquivos que contém os dados a serem trabalhados, o que é necessário para a utilização de paralelismo e distribuição da gigantesca quantidade de dados que se propôs a tratar, os quais precisam estar acessíveis por meio de arquivos. Para esta implementação simplificada não se considerou necessária a utilização desse parâmetro.

Questões de complexidade e insuficiência de recursos são os principais empecilhos para o desenvolvimento dessa estrutura ideal proposta por esta

organização e outros serviços. Assim, para o presente projeto, o objetivo é fornecer uma simplificação do modelo tradicional, admitindo, com isso, maiores limitações e assumindo as diferenças inerentes à essa escolha.

2.1 Principais funcionalidades

Após o entendimento das especificações do modelo MapReduce e especificidades do modelo a ser realmente implementado, o problema foi subdividido em cinco principais funcionalidades a serem desenvolvidas para fornecer o suporte necessário ao modelo: particionamento do conjunto inicial de dados, aplicação da função map fornecida pelo usuário para cada elemento de cada partição, agrupamento dos dados por chave, aplicação da função reduce fornecida pelo usuário para cada conjunto associado a uma chave, incorporação de mecanismos de concorrência nos processamentos realizados pelas funções map e reduce.

2.1.1 Particionamento

O particionamento é a operação inicial de suporte ao modelo MapReduce. Essa funcionalidade foi projetada de modo a receber o conjunto de dados que será alvo do processamento do MapReduce, em formato de lista, e a quantidade de partições desejadas, retornando uma lista de listas como resultado. A função é independente do tipo de dado do conjunto, assim, tipos primitivos ou estruturas são tratados igualmente como um elemento da lista. O particionamento resultante deve ser entendido como cada elemento da lista representando uma partição, sendo cada partição uma lista por si própria.

Considerando a quantidade de partições definida e o tamanho da lista, calcula-se a quantidade de elementos que as partições terão. No caso de uma divisão não exata, os elementos restantes serão distribuídos um para cada partição, de forma que a diferença máxima na quantidade de elementos entre duas partições é de 1. No caso de o tamanho da lista ser menor que a quantidade de partições definida, o MapReduce irá utilizar como quantidade de partições o tamanho da lista. Assim, atribui-se à cada partição a quantidade de elementos pré-definida.

A implementação dessa funcionalidade se deu no módulo Particao, com os respectivos testes sendo criados simultaneamente ao desenvolvimento da função.

Como se tratou do contato inicial da equipe com a implementação do projeto, o que ainda envolveu o planejamento inicial das estruturas de dados a serem utilizadas e outros detalhes técnicos, toda a equipe esteve ativamente envolvida nessa tarefa.

A versão inicial da implementação utilizou uma abordagem recursiva, na qual a lista final era composta por uma partição inicial, gerada a partir da alocação da quantidade de elementos determinada para a partição, e aplicação recursiva da função ao restante da lista. Essa versão foi preterida em detrimento de uma que não utiliza recursão explicitamente, mas a função chunk_every do módulo Enum nativo do Elixir, obtendo ganho de desempenho nos testes realizados.

2.1.2 Suporte ao Map

O suporte ao map do usuário trata-se, tão somente, de uma funcionalidade que recebe a lista com as partições, gerada na operação anterior, e executa a função map do módulo Enum duas vezes, uma para acessar cada partição, e outra vez para acessar cada elemento dessa partição, ao qual o map do usuário é aplicado. Com a adição do recursos de concorrência os Enum.map/2 que irão percorrer os elementos de cada partição são executados simultaneamente.

Espera-se que a aplicação da função map do usuário transforme cada elemento em uma estrutura do tipo chave-valor, pois essa é a estrutura prevista para se realizar o agrupamento posterior dos elementos. Nesse sentido, considerando as estruturas suportadas pelo Elixir, a função map do usuário deve receber um elemento do conjunto de dados e retornar uma tupla com chave e valor. Assim, o resultado dessa funcionalidade é uma lista de partições em que, presume-se, cada elemento é uma 2-tupla. Essa funcionalidade foi implementada diretamente no módulo App.

Os testes dessa funcionalidade dependem da existência de uma função map do usuário, assim, os testes foram realizados simulando um map do usuário que recebe uma palavra e retorna uma 2-tupla na qual a chave é a própria palavra e o valor é inteiro 1, como em {“casa”,1}. A função que simula o map do usuário foi a map do módulo Usuario.

2.1.3 Concorrência

Os mecanismos de concorrência, embora devam estar presentes na operação do suporte ao map do usuário, foram incluídos somente após a fase de testes e validação dessa operação.

Com o suporte ao map concretizado, iniciou-se o processo de implementação dos recursos de concorrência. Nessa etapa foram identificados como possíveis soluções a função spawn, que cria novos processos, e a async do módulo Task, sendo essa última a escolhida por se tratar de uma abstração que simplifica a inicialização de novos processos (utilizando spawn) e a espera pela finalização das tarefas assíncronas.

Para adicionar a concorrência ao suporte ao map já implementado foi necessário apenas ajustar o map que percorre cada elemento de uma partição para, em vez de ser executado diretamente, ser executado por meio do Task.async/3.

Com esse ajuste, cada partição é tratada de forma independente, o que colabora no ganho de desempenho do programa, visto que elementos de diferentes partições podem ser processados simultaneamente.

As máquinas nas quais o programa foi testado não dispunham de uma estrutura de hardware com muitos núcleos de processamento (máquinas com 8 e 4 núcleos), portanto, o processamento dos dados das partições ocorreu majoritariamente de maneira concorrente, e não realmente paralela (ou simultânea). No entanto, mesmo com essa limitação, testes de benchmark realizados comparando as versões com e sem async apontaram uma melhoria no desempenho quando usando async.

Para as próximas operações do programa é necessário que todas as partições tenham terminado de executar, isso porque, nos passos seguintes, dados que devem ser processados juntos podem estar espalhados nas diversas partições. Nesse sentido, a etapa final dessa funcionalidade é aguardar a finalização das tarefas assíncronas, o que foi feito com a função await_many também do módulo Task. Essa funcionalidade foi implementada diretamente no módulo App.

Não foi exequível conceber testes automatizados que verificassem especificamente o sucesso da implementação dos recursos de concorrência: para esta funcionalidade considera-se apenas o resultado da aplicação da função map do usuário aos elementos das partições como o indicador da corretude do programa até esta etapa.

2.1.4 Agrupamento de chaves

Finalizada a execução do map, considerando que nesse ponto tem-se uma lista de listas (partições) nas quais cada elemento é uma tupla, o passo seguinte é realizar um agrupamento dos elementos com mesma chave.

Nesse momento, todas as partições são desfeitas, de forma que todas tuplas passam a ser elementos de uma única lista, para, então, serem agrupadas. O agrupamento vai resultar em uma lista de 2-tuplas, com o primeiro valor da tupla representando a chave da tupla do passo anterior, e o segundo elemento será uma lista que contém todos os valores das tuplas do passo anterior que compartilham a mesma chave. Dessa forma, não há mais repetição de tuplas com a mesma chave, cada tupla terá uma chave diferente, e uma lista de valores associados a ela.

Assim, uma entrada do tipo: **[[** {casa,1}, {bola,1}, {gato,1} **]** , **[** {gato,1}, {mar,1}, {gato,1} **]]** , ou seja, uma lista de partições (lista) onde cada elemento é uma 2-tupla, resultaria em: **[** { casa, [1] }, { bola, [1] }, { gato, [1,1,1] }, { mar, [1] } **]**, não havendo, no entanto, garantia de ordenação específica, visto que durante o processo de agrupamento por chave existe uma transformação da lista em um map, o que impede a garantia de ordenação.

A implementação dessas operações foi efetuada pela utilização de funções de bibliotecas padrão do Elixir, como o List.flatten/1, Enum.group_by/3 e Map.to_list/1, na função agrupar do módulo App. Os testes automatizados dessa funcionalidade consistiram basicamente em verificar se, para o problema simulado da contagem de palavras, dada uma entrada correta, a saída gerada apresenta a estrutura pretendida.

2.3.5 Suporte ao Reduce

A funcionalidade de suporte ao reduce foi iniciada após a conclusão da função map (embora dependa da função de agrupamento) e por isso o mesmo embasamento foi utilizado em sua criação, o processo foi concretizado inicialmente com uma função específica, e depois foi adaptado para aceitar qualquer função, cumprindo a generalidade esperada do programa.

Após a finalização do agrupamento é realizado um novo particionamento dos dados, para que as partições possam ser processadas de maneira simultânea pela reduce. Assim, a funcionalidade acessa cada partição e aplica a função inserida pelo usuário para agregar, combinar ou consolidar cada elemento, concretizando o papel de reduce no modelo.

No exemplo simulado, de contagem de palavras, a funcionalidade de suporte ao reduce receberia uma partição no formato **[** { casa, [1] }, { gato, [1,1,1] } **]**, e aplicaria a função do usuário para cada uma das tuplas (enquanto outras partições estão da mesma forma sendo processadas em paralelo), resultando em **[** { casa, 1}, {gato, 3} **]**.

Assim como no suporte ao map, os testes dessa funcionalidade dependem da existência de uma função reduce do usuário, dessa forma, os testes foram realizados simulando um reduce do usuário que recebe dois parâmetros (chave e valor), com chave e valor correspondendo à chave e valor do elemento (2-tupla) a ser processado, ou seja, a chave sendo uma palavra, e o valor uma lista de números 1, que representa a quantidade de vezes que a palavra aparece no conjunto de dados. Essa função simulada retorna uma 2-tupla com a palavra como chave e a contagem do tamanho da lista do valor.

Essa funcionalidade foi implementada, principalmente, na função reduce do módulo NossoReduce, enquanto que a função que simula a reduce do usuário é a reduce do módulo Usuario.

2.2 Outras funcionalidades

Além das funcionalidades principais mencionadas outras auxiliares precisam ser disponibilizadas para que o programa pudesse ser efetivamente demonstrado. É o caso, por exemplo, do módulo ES (arquivo io.ex), que foi desenvolvido para testar o problema da contagem de palavras de um texto, sendo responsável por ler o conteúdo de uma arquivo e gerar uma lista contendo todas as palavras. Outro exemplo é o módulo Usuario que, como já foi mencionado, foi utilizado para simular o que poderiam ser funções de map e reduce do usuário.

Cabe destacar, no entanto, o módulo TesteDesempenho. As funções desse módulo foram criadas para processar os dados de maneira semelhante ao apresentado durante esse relatório, excluindo, no entanto, todos os recursos relacionados à concorrência e paralelismo, como as operações de particionamento e execução do Task.async/3. O objetivo, com isso, era possibilitar a comparação das duas versões e verificar se, no contexto dos testes efetuadas, a utilização da estrutura concorrente traria benefícios para o desempenho do programa.

Essas comparações foram realizadas com o auxílio da biblioteca benchee, que permite a execução de testes de desempenho e uso de memória, constando no arquivo benchmark.exs as configurações desses testes.

3 UTILIZAÇÃO

O código-fonte do programa pode ser baixado em: https://github.com/gabriel-guilherme/map-reduce-elixir .

O programa foi desenvolvido utilizando a versão 1.14 do Elixir, portanto, é necessário possuir essa versão instalada na máquina onde for ser executado. Além disso, considerando o uso da biblioteca benchee, recomenda-se a instalação da dependência com o comando mix deps.get ou remoção dessa biblioteca do mix.exs.

Após a compilação do programa com o mix compile é possível rodar o benchmark usando o comando mix run benchmark.exs, e rodar os testes (7 no total) usando mix test. Os arquivos de entrada utilizados nos testes são o input.txt e

sherk.txt, portanto, recomenda-se não removê-los nem alterá-los. O conteúdo de ambos simula um texto que seria utilizado como base para o problema da contagem de palavras.

Para executar o programa no modo iterativo utiliza-se a função App.executar/4 que recebe como parâmetros os seguintes argumentos em ordem:

- Arquivo de dados: arquivo de onde serão lidos os dados de entrada para posterior uso do “map” fornecido pelo usuário. Exemplo: “input.txt”
- Quantidade de partições: quantidade de partições escolhidas pelo usuário para serem executadas em paralelo.
- Função map do usuário: função “map” definida pelo usuário para o uso no modelo.
- Função reduce do usuário: função “reduce” definida pelo usuário para o uso no modelo.

Assim, pode-se executar, por exemplo:

```
iex -S mix

App.executar(“sherk.txt”, 4, &NovoModulo.map/1, &NovoModulo.reduce/2)
```

Considerando que a execução real do programa pela função indicada anteriormente demanda que o usuário crie e passe para o App.executar/4 como parâmetro as funções map e reduce a serem utilizadas, foi pensada em uma alternativa para facilitar a demonstração do programa. Assim, foram disponibilizados os métodos App.demo/2 e App.demo/1, os quais utilizam as funções map e reduce do módulo Usuario, que simulam a contagem de palavras de um texto. App.demo/1 recebe por parâmetro somente a quantidade de partições desejadas, e usa como arquivo de dados o sherk.txt, já App.demo/2 recebe, como primeiro parâmetro, um arquivo de dados a ser utilizado. Logo, para uma demonstração de contagem de palavras é possível executar simplesmente: App.demo(5).

4 CONSIDERAÇÕES FINAIS

A implementação do modelo MapReduce em Elixir, apesar das limitações impostas pelo escopo simplificado do projeto, mostrou, além das características do modelo, a possibilidade de se utilizar Elixir para o processamento de grandes volumes de dados.

Durante todo o desenvolvimento, foram exploradas e implementadas as principais funcionalidades do modelo em questão: particionamento do conjunto de dados, aplicação das funções map e reduce fornecidas pelo usuário e a incorporação de mecanismos de concorrência para melhorar o desempenho do processamento.

A principal diferença entre o modelo proposto pelo Google e da implementação aqui apresentada reside na simplificação e, consequentemente, limitações do projeto, que não oferece suporte para processamento distribuído nem tolerância a falhas. Apesar disso, foi possível criar uma versão funcional do MapReduce que pode servir como base para futuras extensões, visando maior robustez e atender uma complexidade de demandas.

Com base nos resultados obtidos, observou-se que Elixir é uma linguagem adequada para a implementação de modelos concorrentes de processamento de dados, e que o MapReduce pode ser adaptado e implementado de maneira eficiente mesmo em contextos com restrições de recursos. Essa implementação básica abre caminho para aprimoramentos e a inclusão de funcionalidades a fim de torná-la uma ferramenta ainda mais poderosa na manipulação de grandes volumes de dados.

REFERÊNCIAS

https://static.googleusercontent.com/media/research.google.com/pt-BR//archive/mapreduce-osdi04.pdf

https://stackoverflow.com/questions/12375761/good-mapreduce-examples

https://elixirschool.com/pt/lessons/intermediate/concurrency

https://www.geeksforgeeks.org/mapreduce-understanding-with-real-life-example/

https://en.wikipedia.org/wiki/MapReduce