Apache Spark

Carlos Eduardo Martins Relvas 5893531

Junho 2015

1 Introdução

O surgimento do hadoop no final da década passada inovou na forma que armazenamos e processamos os dados. Baseado em processamento paralelo, baixo custo de hardware e à prova de falhas, este projeto open source expandiu consideravelmente a capacidade de processamento sem grande elevação do custo final para o manuseio destes dados. Umas das grandes motivações para o surgimento do hadoop foi o aumento do número de dados para serem armazenados. No cenário da época, início dos anos 2000, tínhamos memória em disco barata e memória RAM era muito mais cara. Assim o hadoop surgiu com foco em dados em disco. Já no cenário atual, a memória RAM se tornou mais barata. Neste contexto, recentemente foi criado o projeto Spark, que tem como principal caracteríctica o uso extensivo da memória RAM, melhorando a performance computacional em relação ao Hadoop.

Neste trabalho discutiremos algumas das principais características e propriedades do Spark. Para isso, inicialmente, na Seção 2, faremos uma breve introdução ao Hadoop para motivarmos na Seção 3 o Spark. Nesta seção, falaremos com mais detalhes deste projeto. Dentro desta seção, discutiremos os projetos pertencentes ao ecossistema Spark, como o Spark SQL, Spark Streaming, MLlib e GraphX, em diferentes subseções. Por fim, na Seção 4, apresentamos uma breve conclusão do trabalho e na Seção 5 mostramos a bibliografia utilizada.

2 Apache Hadoop

Em 2003, o Google publicou um paper sobre o Google File System, um sistema de arquivos distribuídos em um cluster de commodity hardware. Já em 2004, aquela instituição publicou um novo paper sobre o uso do modelo de programação Map Reduce em clusters de computadores. Apesar de publicar todo o conhecimento, as ferramentas em si não foram liberadas para o público. Neste contexto, a comunidade científica começou a trabalhar para criar ferramentas open parecidas. Doug Cutting, com forte patrocínio do Yahoo, foi o criador do Hadoop em 2007, baseando o seu trabalho nos dois papers publicados pelo Google.

O Hadoop é uma ferramenta open source escrita em Java para armazenamento e processamento distribuído, consistindo basicamente no módulo de armazenamento (HDFS, hadoop distributed file system) e no módulo de processamento (Map Reduce).

O Hadoop trouxe interessantes propriedades para um cluster de computadores commodity. Umas dessas propriedades é a tolerância a falhas. Por ser voltado para computadores comuns, falhas acontecem o tempo todo. Para contornar isso, o Hadoop distribui os dados quando armazenados e replica-os, geralmente em nós de computadores diferentes. Usualmente são 3 réplicas, no entanto, este número pode ser parametrizado. Além de ter várias réplicas, quando um processo que está sendo executado em um nó demora para responder ou falha, o Hadoop automaticamente aloca um outro nó para realizar este processo.

Outra propriedade interessante presente no Hadoop é trazer a computação para onde o dado está. Em computação distribuída, um dos maiores gargalos é sempre a transferência de dados entre computadores diferentes. Para este problema, o Hadoop é baseado no modelo de programação Map Reduce. Neste modelo, os dados que são particionados e armazenados em diferentes nós são processados por duas funções, a função Mapper e a função Reducer. A função Mapper irá ser executada em cada um dos pedaços e irá emitir várias tuplas de chave - valor. Após a conclusão do processo de Mapper, há o processo de Shuffle, responsável por ordenar as tuplas de todos os processos que executaram os mappers e distribuir todas as chaves idênticas para o mesmo reducer, que irá receber estes valores e executar uma outra função nestes dados. A ideia aqui é apenas dar uma breve descrição do modelo de Map Reduce, sem entrar em muitos detalhes. Vale ressaltar que o Hadoop utiliza apenas programação Map Reduce.

O modelo de programação em Map Reduce é acíclico, o que faz com que seja muito bom para gerenciamento de falhas, visto que se uma função mapper ou reducer falhar, basta executar novamente este pedaço. No entanto, este modelo não é indicado para reuso de dados, como processos iterativos, que são muitos comuns em análise de dados (diversos algoritmos de machine learning). Para executar estes processos iterativos usando Map Reduce, temos que encadear diversos processos de Map Reduce, no entanto, não conseguimos aproveitar a saída de um reducer para a entrada em um mapper de forma automática, teríamos que inicialmente salvar a saída do reducer no HDFS para depois o próximo mapper conseguir acessar estes dados, o que implica em enfatizar o problema de alta latência presente no Hadoop.

Como o modelo de Map Reduce é para processo Batch e não consegue atender de forma satisfatória diversas demandas, foram surgindo dentro do ecossistema Hadoop diversos sistemas especializados, como por exemplo, o Storm e o Kafka para streaming, o Giraph para algoritmos de grafos e o Hive para facilitar a manipulação de dados estruturados por meio de SQL. Estes vários ecossistemas dificultam a vida do usuário que tem que aprender diferentes sistemas para resolver um problema maior.

Aproveitando as propriedades interessantes do Hadoop e tentando melhorar o que não era tão agradável, surgiu o projeto Apache Spark.

A principal característica do Spark é a propriedade de distribuir os dados em memória. Na próxima seção, discutiremos com mais detalhes como o Spark funciona.

3 Apache Spark

De forma geral, o Spark é uma engine rápida, escrita em Scala, para processamento de grandes volumes de dados em um cluster de computadores. Scala é uma liguaguem funcional que roda na JVM. Surgiu como um projeto da AMPLab de Berkeley em 2009, tendo como principal criador Matei Zaharia em sua tese de doutorado. Em 2010, foi lançada a primeira versão, a alpha-0.1. Em junho de 2015 está sendo lançada a versão 1.4, com muito mais propriedades interessantes. Apesar de ter surgido em Berkeley, mais tarde, os criadores de Spark fundaram uma empresa, DataBricks para suportar e fornecer consultoria ao Spark.

No início de 2014, o Spark se tornou um projeto top level Apache. Atualmente é um dos projetos mais ativos da comunidade open source

com mais de 400 desenvolvedores de 50 empresas diferentes, apresentando committers de mais de 16 organizações, incluindo grandes empresas como Yahoo, Intel, DataBricks, Cloudera, HortonWorks, etc.

Atualmente já é possível perceber diversas empresas, grandes ou pequenas, utilizando Spark nos seus ambientes internos ou na própria nuvem, visto que é simples subir o Spark na cloud da Amazon, o que faz com que o Spark seja uma grande tendência para os próximos anos, assim como a necessidade de profissionais qualificados em Spark.

Definindo um pouco a notação utilizada neste trabalho, um nó é um computador do cluster e se divide em master, responsável por gerenciar todo o cluster, e workers, responsáveis por executar os processos. O Spark driver é o programa principal que cria um spark context e comunica-se com o cluster manager para distribuir tarefas aos executores.

O Spark, assim como o Hadoop, também foi pensado para ser escalável. Ou seja, assim quando seu problema for muito grande, basta aumentar o número de computadores disponíveis para conseguir processar e reduzir o tempo. É esperada uma redução quase linear do tempo com mais computadores. No entanto, conforme discutiremos adiante, o Spark ainda não escala tão bem com muitos computadores, mais de quinhentos mil computadores.

Umas das grandes vantagens do Spark em relação ao Hadoop são as High Level API's de programação. Enquanto que o Hadoop é nativamente apenas Java, no Spark temos a disposição API's em Scala, Java e Python, sendo este último o mais utilizado atualmente. Uma nova API com suporte para R e Dataframes será disponibilizada no Spark 1.4, lançado nas próximas semanas. Por ser uma API de alto nível, os programadores podem focar na lógica do problema, fazendo com que o desenvolvimento seja mais fácil e rápido. Vale ressaltar que estas API's não estão disponíveis para todos os módulos do Spark. Por exemplo, a programação no Spark Streaming só está disponível em Scala atualmente.

Outra vantagem do Spark em relação ao Hadoop é a baixa latência, considerada near real time. Isto se deve ao Hadoop persistir dados intermediários em disco, ao contrário do Spark, como veremos adiante, que tem diversos tipos de persistência. Outra propriedade interessante no Spark é a possibilidade de construção de uma arquitetura Lambda usando apenas Spark.

O Spark pode acessar dados de diferentes fontes, como por exemplo, do HDFS, o armazenamento distribuído presente no Hadoop. Além disso,

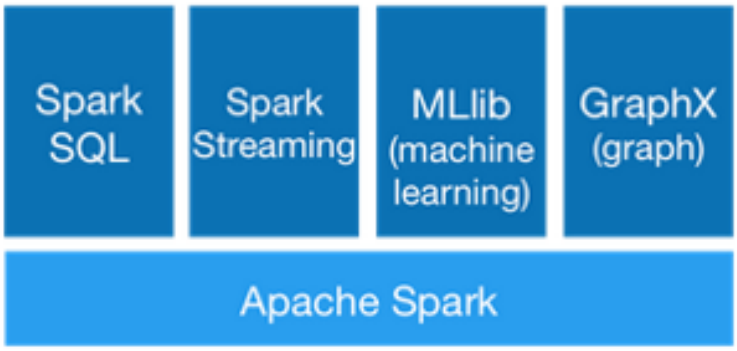
ele pode acessar dados de banco de dados NOSQL como Cassandra ou Hbase, ou até dados utilizados no Hive. Há muitas outras opções e sempre são feitas novas conexões do Spark com outros bancos de dados.

Além de poder fazer com que o Spark utilize dados armazenados em diferentes lugares, podemos rodar o Spark de diferentes maneiras. A primeira possibilidade, que é muito utilizada para testes, é executar o Spark local sem processamento distribuído. Há também a possibilidade de rodar o Spark local com múltiplos processadores (threads). O modo mais comum, entretanto, é utilizar o Spark em um cluster de computadores, podendo ser controlado por diferentes gerenciadores. Se não houver nenhum gerenciador no seu cluster, é possível utilizar o gerenciador do próprio Spark, Spark Standalone. Também, jobs de Spark podem ser gerenciados pelo Hadoop Yarn, Apache Mesos ou pelo gerenciador da Amazon (EC2). Mesos também é um projeto Top Level Apache de Berkeley e que tem como um dos seus criadores, Matei Zaharia, o mesmo criador do Spark, fazendo com que fosse muito utilizado no começo do desenvolvimento de Spark.

O Spark pode ser utilizado interativamente por meio da Spark Shell, disponível em Python ou Scala. A programação em Spark é muito baseada em programação funcional, sempre passando funções como parâmetros.

Um dos problemas do Hadoop que apontamos anteriormente é que o modelo de programação Map Reduce não era a solução para tudo, o que fez com diversos sistemas especializados surgissem. O Spark tenta solucionar isso sendo uma ferramenta mais genérica que substitui muito destes sitemas criados em cima do Hadoop. Por exemplo, o Hive foi criado para a execução de queries em dados estruturados sem a necessidade de escrevermos função Map Reduce em Java. Para isso, um dos módulos em Spark é o SparkSQL. Para algoritmos de machine learning, surgiu no Hadoop uma biblioteca de código em Java, denominada Mahout, já no Spark há o módulo do MLLib com o mesmo objetivo. Da mesma forma o Spark streaming tem como objetivo processamento de dados em tempo real, assim como o A pache Storm. Já o módulo GraphX do Spark surgiu como alternativa ao Apache Giraph.

A Figura 1 abaixo nos mostra os módulos construídos no Spark. Falaremos nas subseções adiante um pouco de cada um destes módulo.



(a) *Spark módulos*

A programação em Spark se baseia no conceito de RDD's. RDD's são a unidade fundamental de dados no Spark e tem como principal característica a propriedade de ser imutável. Após ser criada, não podemos alterá-la, apenas criar um nova RDD baseada nesta RDD. RDD significa resilient distribute dataset. A propriedade de resiliência vem da tolerância a falhas, se os dados na memória forem perdidos, podem ser recriados. Explicaremos mais para frente como o Spark trata a questão da resiliência. O distributed, como o próprio nome já diz, significa que os dados são armazenados na memória de todo o cluster. Por fim o dataset indica que os dados iniciais podem vir de um arquivo ou ser criado programaticamente. Logo, RDD's são criadas por meio de arquivos, de dados na memória ou de outras RDD's. A maioria das aplicações em Spark consistem em manipular RDD's.

RDD's podem armazenar qualquer tipo de elemento, como tipos primitivos (inteiros, caracteres, booleanos, etc), sequências (strings, listas, arrays, tuples, etc), assim como tipos mistos e objetos Scala ou Java. Alguns tipos de RDD ainda possuem outras características, como o Pair RDD's, que armazenam na forma de chave valor e os Double RDD's, que armazenam dados numéricos.

As RDD's são particionadas e divididas em todo o cluster. O Spark constrói estas partições de forma automática, no entanto, o usuário também pode escolher o número de partições desejado.

As operações em RDD são divididas em dois tipos, de transformação e de ação. Há diversas funções de transformação e de ação. As operações de transformação retornam um novo RDD, enquanto que operações de ação retornam um resultado para o driver ou escreve na camada de armazenamento. Abaixo apresentamos alguns exemplos destas funções.

A lista mais extensa pode ser consultada na documentação do Spark. Abaixo, segue alguns exemplos de operações de transformação:

- map (func) - retorna um novo RDD aplicando a função func em cada elemento.
- filter (func) - retorn um novo RDD aplicando o filtro func.
- flatMap (func) - similar ao map, mas retornando mais itens ao invés de apenas um.
- sample(withReplacement, fração, semente) - amostra aleatoriamente uma fração dos dados com ou sem reposição usando a semente para gerar os números aleatórios.
- union(rdd) - retorna um novo RDD que contém a união dos elementos do RDD original e do RDD passado como argumento.
- distinct() - retorna um novo dataset contendo os valores distintos do RDD original.
- groupByKey() - aplicado em pair RDD's da forma (K, V), retornando um novo pair RDD da forma (K, iterable<V>).
- reduceByKey(func) - aplicado também em um pair RDD (K, V), agregando os valores de V pela função func para cada chave K.
- pipe(command) - aplica para cada partição do RDD um comando shell. Qualquer linguagem com stdin, stdout pode ser utilizada.

Algumas operações de ação são:

- collect() - retorna todos os elementos do RDD como um array para o driver. Usado principalmente após uma operação de filtro para retornar poucos dados.
- count() - retorno o número de elementos no RDD.
- first() - retorna o primeiro elemento do dataset.
- take(n) - retorna os primeiros n elementos do dataset.
- saveAsTextFile(file) - salva o RDD em um arquivo de texto.

Uma propriedade interessante das RDD's é a lazy evaluation, que significa que nada é processado até uma operação de ação. Por exemplo, imagine que executamos o código abaixo:

```
> data = sc.textfile("test.txt")

> data_up = data.map(lambda line: line.upper())

> data_filtr = data_up.filter(lambda line: line.startswith("E"))

> data_filtr.count()

 3
```

(b) *Exemplo de programa em Spark usando API em Python.*

A primeira linha lê um arquivo texto e cria o RDD data. No entanto, este RDD não é criado até utilizarmos uma operação de ação. A segunda linha aplica uma função em cada elemento do RDD data, criando o RDD data_up transformando todas as letras em maiúsculas. Mais uma vez, na verdade, o Spark não cria nada, apenas salva como criar. A terceira linha pedimos para criar o RDD data_filtr filtrando data_up para linhas que comecem com E. Novamente nada é criado. Na quarta linha aplicamos uma ação, pedindo para ele retornar o número de linhas do RDD data_filtr. Apenas nesta etapa, o Spark que guardou toda a linhagem dos dados, irá criar todos os RDD's e executar a contagem final.

A linhagem dos dados (data lineage) também é algo importante dentro do Spark, visto que é exatamente o que ele utiliza para ser tolerante à falha. Assim, se um processo falha, o Spark utiliza esta linhagem para reconstruir tudo que falhou e processar novamente.

Uma das propriedades mais interessantes do Spark é a persistência de dados, podendo deixar algumas RDD's na memória RAM, para reuso, diminuindo consideravelmente o tempo de execução de processos iterativos, visto que não precisarão, a cada etapa, escrever os resultados no disco para depois ler novamente. O Spark apresenta diversos níveis de persistência. A Figura abaixo, retirada de Zaharia et al. (2015), apresenta todas as possibilidades de persistência.

Level	Space Used	CPU time	In memory	On Disk	Nodes with data	Comments
MEMORY_ONLY	High	Low	Y	N	1	
MEMORY_ONLY_2	High	Low	Y	N	2	
MEMORY_ONLY_SER	Low	High	Y	N	1	
MEMORY_ONLY_SER_2	Low	High	Y	N	2	
MEMORY_AND_DISK	High	Medium	Some	Some	1	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_2	High	Medium	Some	Some	2	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	1	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER_2	Low	High	Some	Some	2	Spills to disk if there is too much data to fit in memory.
DISK_ONLY	Low	High	N	Y	1	
DISK_ONLY_2	Low	High	N	Y	2	

(c) *Níveis de persistência no Spark.*

A opção default é a opção memory_only, que armazena o RDD como um objeto Java deserializado na JVM. Se o RRD não cabe na memória, algumas partições não serão armazenadas e serão recalculadas quando necessário. A opção memory_and_disk é similar, mas quando o RDD não cabe na memória, algumas partições são armazenadas em disco. Outra opção é memory_only_ser que armazena o RDD como um objeto serializado (um byte por partição). Já a opção memory_only_disk_ser é similar, mas faz o spill para disco se não couber tudo na memória invés de reconstruir no momento da execução como no passo anterior. Por fim, a opção disk_only armazena as partições do RDD apenas usando disco. O número 2 ao lado das opções apenas indica que cada partição é replicada em dois nós do cluster.

Para entender como funciona o processo de cache, fizemos uma pequena alteração no código utilizado anteriormente, descrito na Figura abaixo.

```
> data = sc.textfile("test.txt")

> data_up1 = data.map(lambda line: line.upper())

> data_up1.cache

> data_filtr = data_up1.filter(lambda line: line.startswith("E"))

> data_filtr.count()
 3
> data_filtr.count()
 3
```

(d) *Exemplo de persistência de programa em Spark usando API em Python.*

Este código tem como objetivo realizar o mesmo procedimento. No entanto, agora, deixamos na memória o RDD data_up, por meio do comando data_up.cache. Ao executarmos a primeira operação de ação, o count, o processo de lazy evaluation constrói todos os RDD's, deixando em memória o data_up. Quando executamos o segundo count, ele não precisará recalcular todos os RDD's, visto que o data_up já estará pronto, logo iniciará a construção a partir deste RDD. Se por acaso, algum processo de alguma partição falhar, o Spark, que armazenou toda a lineage, irá reconstruir as partes faltantes e executará novamente.

O Spark pode apresentar problemas quando a linhagem dos dados começar a ficar muito grande, visto que se for preciso reconstruir os RDD caso ocorra alguma falha, demorará bastante tempo. Uma maneira de tentar evitar este problema é a opção de checkpoint, que salva os dados direto no HDFS, eliminando toda a linhagem armazenada e tendo ainda tolerância a falhas.

Na notação de Spark, um job é dividido em várias tarefas (tasks). Uma tarefa corresponde a uma sequência de operações em uma mesma partição nos dados. Um conjunto de tarefas compreende um stage. Um job também corresponde a vários stages. Adiante, exemplificaremos estes detalhes de notação.

Assim como no Hadoop, os dados são armazenados em diferentes nós. Quando possível, as tarefas (tasks) são executadas nos nós onde os dados estão na memória.

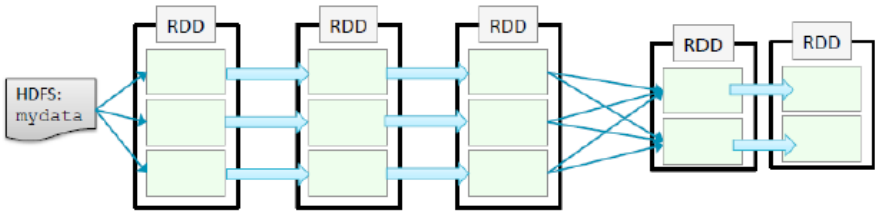
Com o código abaixo, iremos exemplificar estes conceitos definidos

aqui. O objetivo deste código é calcular o tamanho médio das linhas que cada palavra aparece em um arquivo texto. A primeira coisa que ele fez foi ler os dados do HDFS. Depois, aplicamos uma função flatMap para quebrar a linha em palavras, após este passo aplicamos a função Map que retorna cada palavra e o tamanho da linha em que ela aparece. Depois agrupamos todos os elementos de chave e valor pela mesma chave por meio da função groupByKey. Por fim, devolvemos para cada palavra o tamanho médio de todas as linhas em que a palavra aparece, utilizando a função Map novamente.

```
> media_palavra = sc.textfile("/user/hive/warehouse/mydata/") \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0], len(word))) \
    .groupByKey() \
    .map(lambda (k, values): (k, sum(values)/len(values)))
```

(e) *Exemplo de programa em Spark usando API em Python.*

Para exemplificar o que discutimos aqui e o que discutiremos adiante, suponha que executamos este código e que a tabela mydata esteja particionada em três nodes diferentes. Assim, o Spark executará este código do seguinte modo:



(f) *Construção dos Stages e Tasks para execução do código acima.*

O primeiro retângulo da esquerda representa o momento em que lemos os dados do HDFS e armazenamos os três pedaços em três nós diferentes. O segundo representa a execução da função flatMap enquanto que o terceiro representa a execução da função Map. Repare que estas três etapas são feitas de modo independente entre cada pedaço. Depois há uma etapa de shuffle que ordena por chave e distribui as mesmas chaves para as etapas finais. Já na quarta etapa, o groupByKey consolida os resultados de cada chave, no caso cada palavra. Por fim, é calculada a média por meio da função Map.

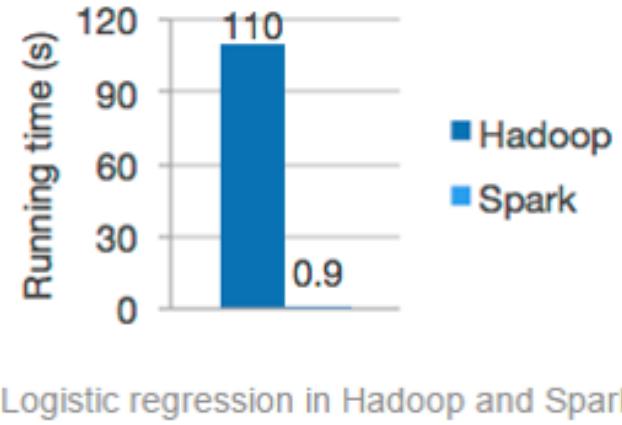
Para cada partição, as execuções das três etapas representa uma task. Logo, temos três tasks nestas três primeiras etapas, definindo o primeiro stage. Repare que se um processo dentro da task falhar, toda a task será recalculada. Depois, entre a etapa 4 e 5, temos que os dois nós podem atuar de forma independente, criando mais duas tasks e o segundo stage.

Para cada uma destas etapas, o Spark constrói um grafo acíclico direcionado (DAG) das dependências entre a RDD, para assim ter todo o mapa de execução caso ocorra alguma falha, bem como para paralelizar as execuções entre as diferentes partições. Há dois tipos de relações entre as RDD's, operações Narrow e operações Wide. Nas operações Narrow, apenas um filho depende da RDD, não é necessário o processo de Shuffle e pode ser colapsada em um estágio único. Já nas operações Wide, múltiplos filhos dependem da mesma RDD e há sempre a definição de um novo stage.

Com a construção do DAG, o Spark paraleliza as execuções, inicialmente com todas as tasks do primeiro stage. Quando elas terminam, fazem o shuffle e executam todas as tasks do segundo stage. Em caso de falhas, ele executa novamente as tasks necessárias.

A Spark shell possibilita a exploração interativa dos dados. Além disso, o Spark também disponibiliza as aplicações em Spark, que rodam como se fossem programas independentes. Estas aplicações em Java ou Scala devem ser compiladas em arquivo JAR e passadas para todos os computadores onde irão ser executadas. Com esta opção, é possível a programação de diversas funções que serão executadas várias vezes por usuários diferentes com diferentes dados.

Com todas estas propriedades, diversos benchmarks surgiram comparando os tempos de execução do Spark com os tempos de execução do Hadoop. Quando os dados cabem na memória, os números apontam que o Spark consegue resultados cerca de cem vezes mais rápidos. Já quando o Spark precisa fazer uso do disco também, os números indicam um melhor em torno de dez vezes. A figura abaixo, retirada do próprio site do projeto, compara o tempo de execução de uma regressão logística no Spark e no Hadoop.



(g) *Regressão Logística em Spark e Hadoop.*

Outro benchmark conhecido é o problema de ordenar um conjunto de 100 terabytes. O Hadoop em 2013 ordenou 103 terabytes com 2100 nós em 72 minutos, já o Spark em 2014 ordenou 100 terabytes com 206 nós em 23 minutos.

Apesar do Spark apresentar propriedades bastante interessantes e ser o assunto do momento no mundo da computação distribuída, nem tudo são flores, apresentando alguns problemas. O que talvez mais esteja impactando as empresas atualmente é a sua imaturidade em razão de ser um projeto bastante novo, com vários bugs sendo descobertos recentemente. Outro fator conhecido é que ainda não é tão escalável quanto o Hadoop. Outro ponto é que por poder apresentar um job com tasks formada por vários processos, sob uma falha, pode demorar um tempo maior para esta falha ser refeita. Como era de esperar, outro ponto é que vários jobs de Spark consomem muita memória RAM, o que requer um gerenciador melhor do cluster. Por fim, outro ponto de desvantagem ao ser comparado com o Hadoop, é que sua configuração é mais difícil, bem como a otimização de todos os parâmetros de configuração para melhorar o desempenho do seu cluster para algum caso específico.

Nas próximas seções, iremos entrar nos detalhes nos diferentes módulos existentes no Spark.

3.1 Spark SQL

Como SQL é uma liguaguem muito aceita e utilizada para trabalhar com dados estruturados, foi criado o módulo Spark SQL. Funciona de maneira similar ao Hive, que converte códigos SQL para Map Reduce Java. Este módulo funciona de maneira similar, convertendo queries em jobs Spark. Com esta solução é muito fácil integrar comando SQL com comando Spark, armazenando os resultados do SQL em um RDD e o utilizando com novos comandos Spark, podendo criar soluções e aplicações bem interessantes. É importante salientar que este módulo está disponível para as API's de Java, Scala e Python. Outro ponto importante é que este módulo é uma evolução do projeto Shark, também iniciado em Berkeley.

Este módulo também tem a propriedade de poder trabalhar com dados de diversas fontes como, por exemplo, do Hive, arquivos Parquet ou JSON, entre outros. Com isso é possível até mesmo juntar dados do Hive com dados de outras fontes no próprio Spark SQL. Além disso, o Spark SQL também possui uma integração bastante detalhada com o Hive, podendo fazer uso do metastore armazenado no Hive, bem como o frontend, o que cria compatibilidade com todas as queries utilizadas anteriormente. Também é possível o uso de todas as UDF's (user defined functions) construídas anteriormente.

A imagem abaixo nos apresenta um exemplo do uso do Spark SQL em que o resultado de uma query, agora um RDD, é utilizado com comandos spark.

```
sqlCtx = new HiveContext(sc)
results = sqlCtx.sql(
  "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

(h) *Spark SQL combinado com comandos Spark.*

Por fim, outra possibilidade do Spark SQL é fazer uma conexão JDBC ou ODBC para acessar dados de outros bancos de dados.

Aqui apresentamos apenas uma breve descrição do Spark SQL, assim como faremos para todas as extensões disponíveis no Spark. Para maiores detalhes, as referências bibliográficas podem ser consultadas.

3.2 Spark Streaming

Spark Streaming é uma outra extensão do Spark, voltada para processamento de dados em tempo real, apresentando diversas propriedades interessantes, entre elas, podemos destacar a escalabilidade e a tolerância a falhas, processamento único e a possível integração entre processos batch e em tempo real. Infelizmente, até o momento, só está disponível em Java ou Scala.

Funciona, basicamente, dividindo o stream de dados em lotes de n milissegundos, segundos ou minutos e processando cada um destes lotes no Spark como um RDD. Por fim, o resultado é retornado também em lotes. A quantidade de tempo que cada lote será processado é totalmente parametrizável.

Para a criação do streaming de dados, o Spark se baseia no contexto de DStreams (Discretized Stream). Estes objetos podem ser criados por meio de uma Unix Sockets ou de outras fontes, como por meio do Flume, Kafka, Twitter, entre outros. Assim como antes, há dois tipos de operações nas DStreams, a de transformação e a de saída, utilizada para salvar os resultados e funcionando de maneira parecida com as de ação para as RDD's.

Outro conceito interessante presente no Spark é o conceito de operações em sliding window, que possuem algumas operações interessantes como, por exemplo, countByWindow, que irá contar a quantidade de observações sempre de tempos em tempos, definido como parâmetro. A figura abaixo apresenta um exemplo que de 5 em 5 segundos, todo o post do twitter que contém a palavra Spark é contado.

```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

(i) *Scala Streaming, contagem em sliding window.*

Para se prevenir de falhas, o Spark streaming automaticamente, por default, persiste e replica os dados duas vezes. Assim, se algum computador cair ou falhar, todo o processamento poderá ser refeito.

3.3 MLlib (Machine Learning)

Um dos grandes objetivos da construção do Spark foi a melhor perfomance para processos iterativos. Quando falamos de processos iterativos, estamos falando de algoritmos de aprendizado de máquina, que usualmente apresentam processos de otimização iterativos. Vale ressaltar que o MLLib está disponível em Python, Scala e Java.

O MLLib consiste em uma biblioteca de códigos de machine learning prontos e disponíveis para uso, funcionando de forma muito parecida aos pacotes do R ou ao numpy e ao scikit-learn do python. Benchmarks indicam que estes algoritmos são cerca de 100 vezes mais rápidos do que suas respectivas versões em Map Reduce no Hadoop. Além de serem mais rápidos, muitos algoritmos escritos em Map Reduce dentro do Hadoop continuam algumas simplificações para poderem ser executados em Map Reduce, o que não é mais necessário no Spark e o que de fato não acontece.

A versão 1.3 do MLLib, presente também na versão 1.3 do Spark, já possui diversos algortimos. No site oficial do Spark é possível ter uma lista de todos os algoritmos disponíveis e como executá-los. Podemos destacar o algoritmo de SVM, regressão logística, regressão linear, árvores de decisão, sistemas de recomendação, agrupamento por diversas técnicas, SVD e naive Bayes.

Abaixo, podemos visualizar um exemplo do uso do MLLib por meio da construção de um método de agrupamento usando a técnica K-Means com 10 grupos.

```
points = spark.textFile("hdfs://...")
               .map(parsePoint)

model = KMeans.train(points, k=10)
```

(j) *Clustering usando K-Means com o MLLib*

3.4 GraphX

O último módulo do Spark que discutiremos neste trabalho é o GraphX, desenvolvido com o intuito de substituir os sistemas especializados de grafos que foram feitos para Hadoop, permitindo a análise e processamento de grafos em paralelo. Atualmente, vem sendo bastante utilizado para o estudo das diferentes relações em redes sociais. Um ponto importante é que ainda não estão disponíveis para o API em python.

É importante ressaltar que o GraphX tem os mesmos benefícios do Spark, como a flexibilidade e a tolerância a falhas. Além disso, apresenta uma performance similar aos sistemas de grafos mais conhecidos como o Giraph e o GraphLab. No entanto, o grande mérito deste módulo é a unificação de ferramentas para trabalhar com grafos, sendo possível a execução de processos de ETL, análise exploratória e algoritmos mais complexos de análise de grafo. Outra possibilidade que o Spark fornece é a execução de algoritmos usando a API de Pregel, arquitetura para grafos desenvolvida pelo Google.

Assim como no MLLib, o GraphX já disponibiliza diversos algoritmos de grafos, como o famoso algoritmo de PageRank, utilizado pelo Google para ordenar as buscas. Outros algoritmos também estão disponíveis, como Label Propagation, SVD, contagem de triângulos, componentes conectados, entre outros.

Veja abaixo um exemplo de como construir um grafo utilizando este módulo

```
graph = Graph(vertices, edges)
messages = spark.textFile("hdfs://...")
graph2 = graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

(k) *Construção de um grafo com GraphX.*

4 Conclusão

O Spark vem recebendo grande atenção do mundo, tendo muitos desenvolvedores atualmente e sendo implantado em diversas empresas, desde grandes até pequenas. Considerado um dos sucessores do Hadoop, vem ganhando espaço no mundo acadêmico e profissional de forma rápida.

Percebemos ao longo deste trabalho que o Spark apresenta propriedades interessantes que já existiam no Hadoop, como tolerância a falhas e levar o processamento aonde os dados estão. Além destas, ele trouxe novas propriedades que também chamaram a atenção, como a habilidade in memory, a possibilidade do uso de mais linguagens de programação, processamento quase em tempo real, não é limitado apenas a Map Reduce, otimizações de processamento por apresentar lazy evaluation, além

é claro de tentar unificar diversos sistemas especializados.

De tudo que apresentamos aqui, percebemos que o Spark é realmente muito interessante e tem tudo para cada dia mais ser mais utilizado, no entanto, não podemos esquecer que ainda é um ecossistema imaturo, em que diversos bugs ainda são encontrados, além da necessidade de evolução em alguns pontos como segurança. Outro ponto que ainda precisa ser explorado é a grande dificuldade de otimizações de processos, com vários parâmetros para serem configuráveis.

5 Bibliografia

- Zaharia, Matei. An Architecture for Fast and General Data Processing on Large Clusters, EECS Department, University of California, Berkeley, 2014.
- Documentação do Apache Spark: https://spark.apache.org/documentation.html
- Tom White. Hadoop: The Definitive Guide. Yahoo Press.
- Chuck Lam. Hadoop in Action. Manning Publications
- Introduction to Spark Developer Training, http://pt.slideshare.net/cloudera/spark-devwebinarslides-final
- Introduction to Spark Training, http://pt.slideshare.net/datamantra/introduction-to-apache-spark
- Holden Karau, Andy Konwinski, Patrick Wendell and Matei Zaharia. Learning Spark. O'Reilly Media.
- Marko Bonaci e Petar Zecevic. Spark in Action. Manning Publications.
- Sandy Ryza, Uri Laserson, Sean Owen e Josh Wills. Advanced Analytics with Spark. O'Reilly Media.
- Michael Malak. Spark GraphX in Action. Manning Publications.
- Holden Karau. Fast Data Processing with Spark. Packt Publishing.

- Reynold Xin, Joshua Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, Ion Stoica. Shark: SQL and Rich Analytics at Scale. Technical Report UCB/EECS-2012-214, 2012.

- Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, Ion Stoica. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. HotCloud, 2012

- Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Haoyuan Li, Scott Shenker, Ion Stoica. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory (demo). SIGMOD, 2012. Best Demo Award.

- Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI, 2012. Best Paper Award and Honorable Mention for Community Award.

- Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. Technical Report UCB/EECS-2011-82, 2011.

- Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. Spark: Cluster Computing with Working Sets. HotCloud, 2010.