Escalando Spark no Mundo Real: Desempenho e Usabilidade

RESUMO

Apache Spark é uma das ferramentas abertas de processamento mais usadas em BigData, com uma grande quantidade de bibliotecas e ricas API's de integração de linguagem. Nos últimos dois anos, nosso grupo tem se dedicado a implantar o Spark em diversas organizações tanto por meio de empresas de consultoria como pelo nosso serviço hospedado, Databricks. Detalharemos os principais desafios e requerimentos que surgiram em uma grande quantidade de usuários ao se falar de Spark, e as melhorias de usabilidade e desempenho que desenvolvemos para contornar esses problemas.

1- INTRODUÇÃO

Interesses em MapReduce e processamento de dados em larga escala levaram ao surgimento de uma vasta gama de sistemas computacionais em cluster [3, 6, 7]. Esses sistemas usam uma variedade de novas API's, geramente baseadas em programação funcional, para suportar queries relacionais e tipos complexos de processamento (ex: extract-transform-load e Aprendizado de Máquina).

Desses sistemas, o Apache Spark [1] se tornou um dos mais utilizados, com, segundo nossos dados, mais de 500 implantações em produção, e a maior comunidade de colaboradores ativa no Apache (mais de 400 em 2014). Diferente de outros sistemas do tipo, o Spark oferece uma engine geral baseado em tarefas DAG e compartilhamento de dados, onde tarefas como batch Jobs, streaming, SQL e grafos analíticos possam rodar [14, 15, 2]. Também possui API's em Java, Scala, Python e R.

Enquanto o Spark se dissipava para uma plateia maior, tivemos a chance de ver onde sua API executava bem na prática, onde poderia ser melhorada e quais eram as necessidades desses novos usuários. Este artigo descreve as iniciativas que tomamos na Databricks para melhorar a usabilidade e desempenho do Spark. Trabalhamos tanto em melhorias da engine como no desenvolvimento de novas API's para tornar o Spark mais acessível a usuários não especializados, como a API DataFrame, orientada a tabelas [2].

2- DESAFIOS E REQUISITOS

Em geral, o Spark tem sucedido em seu objetivo de suportar cargas de trabalho analíticas: nós e outros temos conseguido implementar bibliotecas em cima de SQL, streaming, processamento de grafos e aprendizado de máquina, muitas vezes com desempenhos comparáveis a de engines especializadas [15, 2, 5]. A generalidade da engine do Spark é importante pois muitos usuários combinam diversos desses tipos de processamento em suas tarefas.

Mesmo assim, a API funcional do Spark apresentou alguns desafios tanto para os usuários quanto para o sistema, dada a heterogeneidade dos dados e cálculos suportados. Os desafios mais comuns que averiguamos foram os seguintes:

Semântica da API funcional. O núcleo da API do Spark é baseado em coleções de objetos Java/Python, da qual usuários rodam funções arbitrárias escritas nestas linguagens através de operadores como **map** ou **groupBy** [14]. Nós descobrimos que usuários tem dificuldade em escolher os melhores operadores para uma certa operação. Por exemplo, um problema comum é usar o operador do Spark **groupByKey**, o qual retorna uma coleção distribuída de pares (chave, lista de valores), e aplicar uma agregação em cada lista (ex.: Soma). O **groupByKey** tem que enviar cada lista de registro para uma máquina, já que esse é seu estilo de retorno, porém essa operação seria muito mais rápida usando um operador **reduceByKey**, que consegue fazer agregações parciais em cada nó.

Uma vez que as funções passadas para o Spark são códigos arbitrários em Java ou Python, fica difícil para a engine analisar e substituir os operadores automaticamente. Algumas pesquisas propuseram análises estáticas de UDFs [11], porém análises desse tipo podem ser frágeis para programas complexos orientados a objeto.

Depuração e perfil. Aplicações distribuídas são inerentemente difíceis de depurar, até mesmo com a API funcional do Spark, side-effect-free, porque os usuários devem se preocupar com a distribuição. Nós descobrimos que os problemas mais complicados estão em depuração de desempenho: usuários nem sempre percebem que seus trabalhos estão concentrados em poucas máquinas, ou que algumas de suas estruturas de dados são ineficientes em memória.

Gerenciamento de memória. O ‘‘Big Data’’ vêm em diferentes tamanhos e formatos, necessitando de um cuidadoso gerenciamento de memória pela engine. Enquanto operações externas para agregações e junções são bem entendidas, nó achamos outras fontes de alto uso de memória. Por exemplo, registros de dados em algumas aplicações (ex.: processamento de imagens) podem ter centenas de megabytes cada, necessitando de cuidadosas verificações enquanto cada registro é lido. Como outro exemplo, o Spark inicialmente assumia que os dados em cada bloco de um arquivo (tipicamente 128MB no HDFS) conseguia caber inteiro em memória simultaneamente, mas para alguns datasets extremamente comprimidos, cada bloco se descomprimia em 3-4GB de dados.

I/O em larga escala. Clusters de Spark e cargas de trabalho tem crescido significativamente, com o maior cluster tendo mais de 8000 nós e Jobs individuais processando mais de 1PB [13]. Nós investimos bastante em engenharia para que as camadas de redes e I/O do Spark consigam operar bem nessa escala.

Acessibilidade para não especialistas. Enquanto sistemas de computação em cluster como MapReduce foram desenvolvidos para engenheiros de software, em boa parte das empresas, ‘‘big data’’ precisa ser acessível para outros indivíduos, como domain experts (ex.: estatísticos e cientistas de dados) que não são desenvolvedores. Além disso, para todos os usuários, APIs de alto nível são importantes já que grande parte da análise de dados é experimental: usuários não tem tempo de desenvolver um programa distribuído e totalmente otimizado. Para resolver esses desafios, investimos bastante para fornecer APIs de alto nível em data science que espelham ferramentas de single-node, como Data Frames em R, em cima do Spark.

Descreveremos agora três importantes áreas que atacam estes desafios: Melhorias no núcleo da engine, ferramentas de depuração e uma nova API DataFrame.

3- MELHORIAS NA ENGINE

Nosso principal trabalho na engine de execução se resume em dois domínios: gerenciamento de memória e a camada de rede. Ambos focam no aumento do desempenho e robustez da engine para cargas de trabalho em larga escala.

3.1- Gerenciamento de Memória

Para melhorar o gerenciamento de memória, nós estudamos as causas dos problemas de memória baseado nos relatórios dos usuários, e implementamos um alocador por nó que gerência todos os usos de memória em cada nó. O Spark inicialmente tinha um gerenciador de memória que rastreava o tamanho dos dados em cache, que os usuários escolhiam em materializar em memória, descartando blocos antigos quando o limite era alcançado. O gerenciador original não rastreava explicitamente o uso de memória para o processamento de dados (ex.: espaços temporários para junções e agregações). Como resultado, uma grande fração dos usos excessivos de memória apareciam no processamento de grandes joins e agregações. Para resolver este problema, implementamos um segundo limite para rastrear tabelas hash de joins e agregações. Este limite era alocado dinamicamente entre os processos que rodavam estas operações enquanto suas tabelas cresciam, e processos que não podiam usar mais RAM começavam a usar o disco. Por último, um terceiro espaço foi reservado para o "desenrolar" dos blocos que eram lidos do disco, para ver se eles descomprimidos, cabiam em cache. Em todos estes casos, averiguávamos a memória a cada 16 segundos para lidar com tamanhos distorcidos de registros. Com estes controladores, a engine roda robustamente em uma vasta gama de cargas de trabalho.

3.2- Camada de rede

Na camada de rede do Spark, o maior desafio foi o suporte as operações de embaralhamento em vários nós. Operações de embaralhamento precisam mover dados de saída das tarefas de mapeamento para as tarefas de redução na rede, de forma que cada nó está enviando dados para todos os outros nós. Essas operações são difíceis de implementar, porque cada nó pode estar usando dados de vários discos, várias conexões geralmente são necessárias para saturar a rede, e cuidados devem ser tomados para se balancear essa carga (ex.: Se todos os redutores contatarem um nó, ele pode ficar sobrecarregado).

Nós escrevemos inicialmente um módulo de rede baseado no NIO do Java. Esse módulo usava diretamente a API de baixo nível de redes do NIO e necessitava manter estados complexos de máquina internamente. Além disso, criava uma pressão maior sobre o coletor de lixo da JVM e um maior uso de CPU, devido as cópias desnecessárias de buffers de rede.

No Apache Spark 1.2, trocamos o módulo de rede por uma nova implementação baseada em Netty (www.netty.io), um framework de rede de alto desempenho. Netty simplifica a programação em rede ao entregar uma abstração de alto nível, assíncrona e dirigida a eventos. Utilizando o Netty, conseguimos introduzir diversas funcionalidades para melhorar o desempenho e escalabilidade:

- I/O sem cópia: Instruir o kernel a copiar os dados diretamente dos arquivos em disco para o soquete, sem ir para o espaço de memória do usuário. Isso não apenas reduz o tempo de

CPU em trocas de contexto entre o kernel e o espaço do usuário, mas também a pressão sobre a memória na pilha da JVM.

- Gerenciamento do buffer de rede fora da pilha: O netty mantém uma coleção de páginas de memória explicitamente fora da pilha do Java, e como resultado elimina o impacto dos buffers de rede sobre o coletor de lixo da JVM.
- Diversas conexões: Cada nodo de trabalho do Spark mantém várias conexões paralelas ativas (5 por padrão) para busca de dados. Isso aumenta a vazão de busca e melhora o balanceamento de carga dos nós servindo dados.

Esta implementação é capaz de saturar uma rede com banda de bisecção completa entre 200 máquinas com links de 10Gbps cada. Nós a usamos para estabelecer o novo recorde na competição Daytona GraySort [12], ao ordenar 100 TB de dados em disco 3x mais rápido que o recorde anterior baseado em Hadoop, com 10x menos máquinas.

4- FERRAMENTAS DE DEPURAÇÃO

Para facilitar a depuração, adicionamos uma variedade de métricas para serem monitoradas pela interface da aplicação web do Spark. Essa interface mostra métricas como tempo levado para planejar, rodar e receber os resultados de cada tarefa; bytes de entrada e saída; e espaços usados em memória. Estas métricas são mostradas em tabelas, as quais o usuário pode ordenar por colunas e achar valores isolados. Elas também são cada vez mais agregadas em grafos, como um dashboard ao vivo com estatísticas das tarefas de streaming (Figura 2) e a visualização do operador DAG (Figura 3).

Além de métricas, uma funcionalidade muito útil que adicionamos foi o botão "stack trace", que pode ser usado para rastrear qualquer nó e ver quais funções ele está rodando no momento. Isso server como um simples perfil de amostragem e como um meio de verificar deadlocks.

Enfim, mesmo a interface atual fornecendo métricas para os operadores de baixo nível do Spark, muitos programas têm cada vez mais utilizado bibliotecas de alto nível como Spark SQL, DataFrames e a biblioteca de aprendizado de máquina do Spark (MLlib). Estamos também ampliando a interface de monitoramento para capturar estas operações de alto nível. Em nossa experiência, a visibilidade dentro do sistema, continua sendo um dos maiores desafios para usuários de computação distribuída.

5- API DATAFRAME

Para deixar o Spark mais acessível aos não especialistas e aumentar a quantidade de informações visíveis à engine, para otimizações automáticas, buscamos desenvolver uma API mais declarativa. Nós escolhemos uma API baseada em data frames, uma comum abstração para tabelas em Python e R. Data Frames suportam operações similares a álgebra relacional, mas as expõe como funções em linguagens procedurais (ex.: Python), para que desenvolvedores possam usar o controle de fluxo e características de abstração de linguagens acessíveis na construção de programas complexos.

Nossa API DataFrame implementa esta interface padrão, mas à compila utilizando o otimizador relacional do Spark SQL [2], possibilitando ricas otimizações lógicas e físicas na

computação como um todo[1]. Isso permite que o Spark DataFrames lide automaticamente com transformações da mesma forma que o problema do **groupBy**, descrito na Seção 2. Em nosso conhecimento, a implementação Spark do DataFrame é a primeira a utilizar um otimizador relacional por baixo—bibliotecas anteriores com as de R são imperativas.

O código abaixo mostra um curto exemplo de DataFrames em Python; a API é similar ao **pandas** (pandas.pydata.org):

means = users.where(users[“age”] > 20).groupBy(“City”).avg(“income”)

A API captura expressões users[“age”] > 21 como árvores sintáticas abstratas para permitir otimizações algébricas, diferente das funções opacas definidas pelo usuário passadas aos operadores funcionais do Spark, como **map**. Ainda assim, usuários ainda conseguem facilmente invocar UDFs quando necessário, passando funções em linha como na API Spark padrão.

Como mostrado na figura 4, as computações baseadas em DataFrames podem ser 2-5x mais rápidas que as da API funcional. Essas melhorias surgem tanto da geração de código em tempo de execução como das otimizações algébricas (ex.: predicate pushdowns).

Além de oferecer DataFrames para transformações básicas, estamos cada vez mais as utilizando para a entrada e saída em bibliotecas do Spark (ex.: A biblioteca de aprendizado de máquina [8]). Isso nos permite facilmente servir bibliotecas do Spark em diversas linguagens suportadas, sem ter que colocar especificidades de linguagens nelas. As recentemente adicionadas ligações em R ao Spark também suportam DataFrames e irão acessar outras bibliotecas desta maneira.

6- Trabalho em Desenvolvimento

Estamos continuamente otimizando o Spark tanto em desempenho como em usabilidade. Em usabilidade, nós e outros membros da comunidade estamos aumentando o Spark com uma grande quantidade de bibliotecas contendo versões escaláveis de algoritmos comuns de análise de dados. Por exemplo, a biblioteca de aprendizado de máquina do Spark, o MLlib, cresceu por um fator de 4 no último ano. Também desenvolvemos uma API plugável de fonte de dados, que facilita o acesso a fontes de dados externas de uma maneira uniforme, usando DataFrames ou SQL [2]. Juntas, essas APIs formam uma das maiores bibliotecas integradas para “big data”, e irá sem sombra de dúvidas conduzir a interessantes decisões de design, que permitirão eficientes composições de fluxo de trabalho.

Em desempenho, estamos, com um novo projeto de codinome Tungsten, implementando um gerenciador de memória fora da JVM e um gerador de código em tempo de execução, para trazer o desempenho do DataFrames e SQL no limite do hardware que roda por baixo [10]. Estas otimizações irão transparentemente acelerar os códigos do usuário e muitas das bibliotecas do Spark.

Temos também visto um crescente uso do Spark em projetos de pesquisa, incluindo agregações online [16], processamento de grafos [9], e neurociência em larga escala. Nós esperamos que o código conciso do Spark e a grande quantidade de funções o torne acessível para ambos o sistema e projetos orientados a aplicação.

Todas as funcionalidades descritas neste trabalho são de código aberto e disponíveis em **spark.apache.org**.

7- Referências

[1] Apache Spark project. http://spark.apache.org.
[2] M. Armbrust et al. Spark SQL: relational data processing in Spark. In SIGMOD, 2015.
[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In OSDI, 2004.
[4] J. Freeman, N. Vladimirov, T. Kawashima, Y. Mu, N. J. Sofroniew, D. V. Bennett, J. Rosen, C.-T. Yang, L. L. Looger, and M. B. Ahrens. Mapping brain activity at scale with cluster computing. Nature Methods, 11(9):941–950, Sept 2014.
[5] J. E. Gonzalez et al. GraphX: Graph processing in a distributed dataflow framework. In OSDI, 2014.
[6] M. Isard et al. Dryad: distributed data-parallel programs from sequential building blocks. Eurosys, 2007.
[7] G. Malewicz et al. Pregel: a system for large-scale graph processing. In SIGMOD, 2010.
[8] X. Meng et al. ML pipelines: a new high-level API for MLlib. http://tinyurl.com/spark-ml.
[9] F. A. Nothaft, M. Massie, T. Danford, Z. Zhang, U. Laserson, C. Yeksigian, J. Kottalam, A. Ahuja, J. Hammerbacher, M. Linderman, M. J. Franklin, A. D. Joseph, and D. A. Patterson. Rethinking data-intensive science using scalable analytics systems. In SIGMOD, 2015.
[10] Project Tungsten. https://databricks.com/blog/2015/04/28/.
[11] K. Tzoumas et al. Peeking into the optimization of data flow programs with mapreduce-style UDFs. In ICDE, 2013.
[12] R. Xin et al. GraySort on Apache Spark by Databricks. http://sortbenchmark.org/ApacheSpark2014.pdf.
[13] R. Xin and M. Zaharia. Lessons from running large scale Spark workloads. http://tinyurl.com/large-scale-spark.
[14] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In NSDI, 2012.
[15] M. Zaharia et al. Discretized streams: Fault-tolerant streaming computation at scale. In SOSP, 2013.
[16] K. Zeng et al. G-OLA: Generalized online aggregation for interactive analysis on big data. In SIGMOD, 2015.