Soluções de armazenamento NetApp para Apache Spark

NetApp artificial intelligence solutions

NetApp
February 12, 2026

Índice

Soluções de armazenamento NetApp para Apache Spark

TR-4570: Soluções de armazenamento NetApp para Apache Spark: Arquitetura, casos de uso e resultados de desempenho

Rick Huang, Karthikeyan Nagalingam, NetApp

Este documento se concentra na arquitetura do Apache Spark, nos casos de uso do cliente e no portfólio de armazenamento da NetApp relacionado à análise de big data e inteligência artificial (IA). Ele também apresenta vários resultados de testes usando ferramentas de IA, aprendizado de máquina (ML) e aprendizado profundo (DL) padrão do setor em relação a um sistema Hadoop típico para que você possa escolher a solução Spark apropriada. Para começar, você precisa de uma arquitetura Spark, componentes apropriados e dois modos de implantação (cluster e cliente).

Este documento também fornece casos de uso do cliente para resolver problemas de configuração e discute uma visão geral do portfólio de armazenamento da NetApp relevante para análise de big data e IA, ML e DL com Spark. Em seguida, finalizamos com os resultados dos testes derivados de casos de uso específicos do Spark e do portfólio de soluções NetApp Spark.

Desafios do cliente

Esta seção se concentra nos desafios dos clientes com análise de big data e IA/ML/DL em setores de crescimento de dados, como varejo, marketing digital, bancos, manufatura discreta, manufatura por processos, governo e serviços profissionais.

Desempenho imprevisível

Implantações tradicionais do Hadoop normalmente usam hardware comum. Para melhorar o desempenho, você deve ajustar a rede, o sistema operacional, o cluster Hadoop, os componentes do ecossistema, como o Spark, e o hardware. Mesmo se você ajustar cada camada, pode ser difícil atingir os níveis de desempenho desejados porque o Hadoop é executado em hardware comum que não foi projetado para alto desempenho em seu ambiente.

Falhas de mídia e nó

Mesmo em condições normais, o hardware comum está sujeito a falhas. Se um disco em um nó de dados falhar, o mestre do Hadoop, por padrão, considera esse nó como não íntegro. Em seguida, ele copia dados específicos desse nó pela rede, de réplicas para um nó íntegro. Esse processo torna os pacotes de rede mais lentos para qualquer tarefa do Hadoop. O cluster deve então copiar os dados novamente e remover os dados replicados em excesso quando o nó com problemas retornar a um estado saudável.

Bloqueio de fornecedor do Hadoop

Os distribuidores do Hadoop têm sua própria distribuição do Hadoop com seu próprio controle de versão, o que prende o cliente a essas distribuições. No entanto, muitos clientes exigem suporte para análises na memória que não vinculem o cliente a distribuições específicas do Hadoop. Eles precisam da liberdade de

mudar as distribuições e ainda levar suas análises com eles.

Falta de suporte para mais de um idioma

Os clientes geralmente precisam de suporte para vários idiomas, além dos programas MapReduce Java, para executar seus trabalhos. Opções como SQL e scripts oferecem mais flexibilidade para obter respostas, mais opções para organizar e recuperar dados e maneiras mais rápidas de mover dados para uma estrutura de análise.

Dificuldade de uso

Já faz algum tempo que as pessoas reclamam que o Hadoop é difícil de usar. Embora o Hadoop tenha se tornado mais simples e poderoso a cada nova versão, essa crítica persiste. O Hadoop exige que você entenda os padrões de programação Java e MapReduce, um desafio para administradores de banco de dados e pessoas com conjuntos de habilidades tradicionais de script.

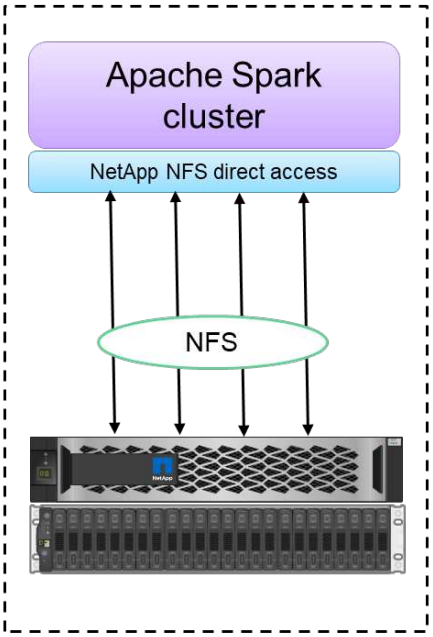
Estruturas e ferramentas complicadas

As equipes de IA corporativas enfrentam vários desafios. Mesmo com conhecimento especializado em ciência de dados, ferramentas e estruturas para diferentes ecossistemas e aplicativos de implantação podem não ser facilmente transferidas de um para outro. Uma plataforma de ciência de dados deve integrar-se perfeitamente com plataformas de big data correspondentes criadas no Spark, com facilidade de movimentação de dados, modelos reutilizáveis, código pronto para uso e ferramentas que oferecem suporte às melhores práticas para prototipagem, validação, controle de versão, compartilhamento, reutilização e implantação rápida de modelos na produção.

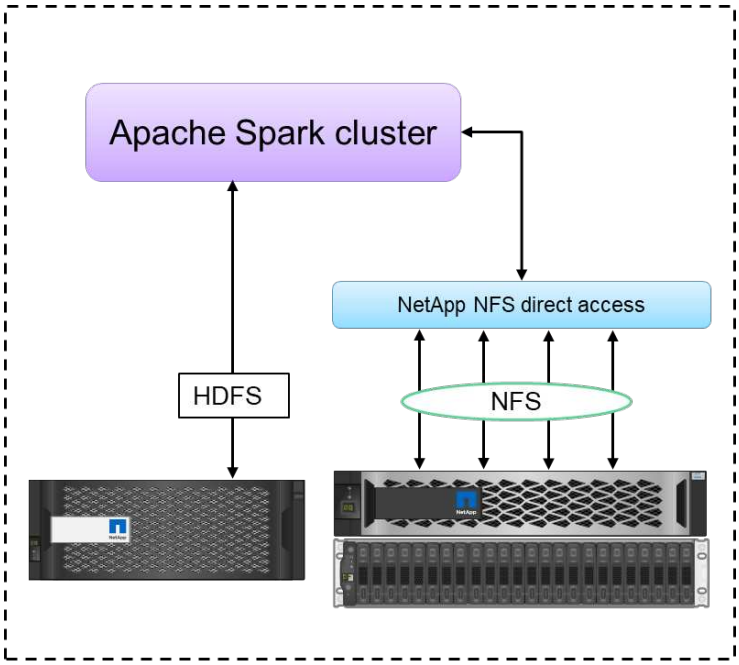
Por que escolher a NetApp?

A NetApp pode melhorar sua experiência com o Spark das seguintes maneiras:

- O acesso direto do NetApp NFS (mostrado na figura abaixo) permite que os clientes executem trabalhos de análise de big data em seus dados NFSv3 ou NFSv4 existentes ou novos sem mover ou copiar os dados. Ele evita múltiplas cópias de dados e elimina a necessidade de sincronizar os dados com uma fonte.
- Armazenamento mais eficiente e menos replicação de servidor. Por exemplo, a solução NetApp E-Series Hadoop requer duas, em vez de três réplicas dos dados, e a solução FAS Hadoop requer uma fonte de dados, mas nenhuma replicação ou cópias de dados. As soluções de armazenamento da NetApp também produzem menos tráfego de servidor para servidor.
- Melhor comportamento do cluster e do trabalho do Hadoop durante falhas de unidade e nó.
- Melhor desempenho de ingestão de dados.



Configuration 1: NFS as primary storage



Configuration 2: HDFS and NFS in single Spark cluster

Por exemplo, no setor financeiro e de saúde, a movimentação de dados de um lugar para outro deve atender a obrigações legais, o que não é uma tarefa fácil. Nesse cenário, o acesso direto do NetApp NFS analisa os dados financeiros e de saúde de seu local original. Outro benefício importante é que o uso do acesso direto do NetApp NFS simplifica a proteção de dados do Hadoop usando comandos nativos do Hadoop e permitindo fluxos de trabalho de proteção de dados com o rico portfólio de gerenciamento de dados da NetApp.

O acesso direto do NetApp NFS oferece dois tipos de opções de implantação para clusters Hadoop/Spark:

- Por padrão, os clusters Hadoop ou Spark usam o Hadoop Distributed File System (HDFS) para armazenamento de dados e o sistema de arquivos padrão. O acesso direto do NetApp NFS pode substituir o HDFS padrão pelo armazenamento NFS como o sistema de arquivos padrão, permitindo análises diretas em dados NFS.
- Em outra opção de implantação, o acesso direto do NetApp NFS oferece suporte à configuração do NFS como armazenamento adicional junto com o HDFS em um único cluster Hadoop ou Spark. Nesse caso, o cliente pode compartilhar dados por meio de exportações NFS e acessá-los do mesmo cluster junto com os dados HDFS.

Os principais benefícios de usar o acesso direto do NetApp NFS incluem o seguinte:

- Analisar os dados de seu local atual, o que evita a tarefa demorada e de alto desempenho de mover dados analíticos para uma infraestrutura Hadoop, como o HDFS.
- Reduzindo o número de réplicas de três para uma.
- Permitindo que os usuários dissociem a computação e o armazenamento para dimensioná-los de forma independente.
- Fornecendo proteção de dados empresariais aproveitando os recursos avançados de gerenciamento de dados do ONTAP.
- Certificação com a plataforma de dados Hortonworks.
- Habilitando implantações de análise de dados híbrida.
- Reduzindo o tempo de backup aproveitando a capacidade multithread dinâmica.

Ver"TR-4657: Soluções de dados em nuvem híbrida da NetApp - Spark e Hadoop com base em casos de uso do cliente" para fazer backup de dados do Hadoop, backup e recuperação de desastres da nuvem para o local, habilitar DevTest em dados do Hadoop existentes, proteção de dados e conectividade multicloud e acelerar cargas de trabalho de análise.

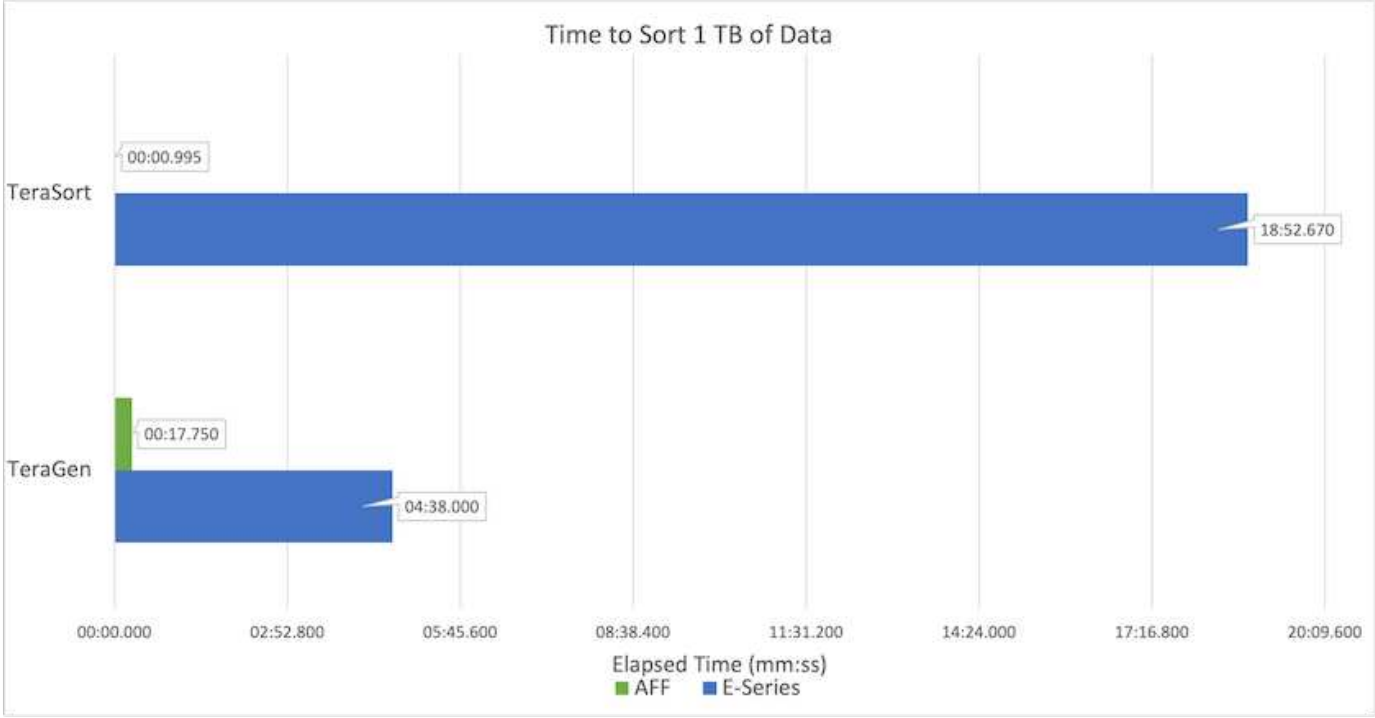
As seções a seguir descrevem recursos de armazenamento que são importantes para clientes do Spark.

Camadas de armazenamento

Com o armazenamento em camadas do Hadoop, você pode armazenar arquivos com diferentes tipos de armazenamento de acordo com uma política de armazenamento. Os tipos de armazenamento incluem `hot`, `cold`, `warm`, `all_ssd`, `one_ssd`, e `lazy_persist`.

Realizamos a validação da divisão em camadas de armazenamento do Hadoop em um controlador de armazenamento NetApp AFF e um controlador de armazenamento da série E com unidades SSD e SAS com diferentes políticas de armazenamento. O cluster Spark com AFF-A800 tem quatro nós de trabalho de computação, enquanto o cluster com E-Series tem oito. O objetivo principal é comparar o desempenho de unidades de estado sólido (SSDs) com o de discos rígidos (HDDs).

A figura a seguir mostra o desempenho das soluções NetApp para um SSD Hadoop.



- A configuração básica do NL-SAS usava oito nós de computação e 96 unidades NL-SAS. Esta configuração gerou 1 TB de dados em 4 minutos e 38 segundos. Ver "Solução NetApp E-Series TR-3969 para Hadoop" para obter detalhes sobre a configuração do cluster e do armazenamento.
- Usando o TeraGen, a configuração SSD gerou 1 TB de dados 15,66x mais rápido que a configuração NL-SAS. Além disso, a configuração SSD usou metade do número de nós de computação e metade do número de unidades de disco (24 unidades SSD no total). Com base no tempo de conclusão do trabalho, ele foi quase duas vezes mais rápido que a configuração NL-SAS.
- Usando o TeraSort, a configuração SSD classificou 1 TB de dados 1138,36 vezes mais rápido que a configuração NL-SAS. Além disso, a configuração SSD usou metade do número de nós de computação e metade do número de unidades de disco (24 unidades SSD no total). Portanto, por unidade, era aproximadamente três vezes mais rápido que a configuração NL-SAS.

- A conclusão é que a transição de discos giratórios para all-flash melhora o desempenho. O número de nós de computação não era o gargalo. Com o armazenamento all-flash da NetApp, o desempenho do tempo de execução é bem dimensionado.
- Com o NFS, os dados eram funcionalmente equivalentes a serem agrupados, o que pode reduzir o número de nós de computação dependendo da sua carga de trabalho. Os usuários do cluster Apache Spark não precisam rebalancear manualmente os dados ao alterar o número de nós de computação.

Escalonamento de desempenho - Escala horizontal

Quando você precisa de mais poder de computação de um cluster Hadoop em uma solução AFF , você pode adicionar nós de dados com um número apropriado de controladores de armazenamento. A NetApp recomenda começar com quatro nós de dados por matriz de controlador de armazenamento e aumentar o número para oito nós de dados por controlador de armazenamento, dependendo das características da carga de trabalho.

AFF e FAS são perfeitos para análises no local. Com base nos requisitos de computação, você pode adicionar gerenciadores de nós, e operações não disruptivas permitem adicionar um controlador de armazenamento sob demanda, sem tempo de inatividade. Oferecemos recursos avançados com AFF e FAS, como suporte de mídia NVME, eficiência garantida, redução de dados, QOS, análise preditiva, camadas de nuvem, replicação, implantação de nuvem e segurança. Para ajudar os clientes a atender às suas necessidades, a NetApp oferece recursos como análise de sistema de arquivos, cotas e balanceamento de carga on-box, sem custos adicionais de licença. A NetApp tem melhor desempenho no número de trabalhos simultâneos, menor latência, operações mais simples e maior taxa de transferência de gigabytes por segundo do que nossos concorrentes. Além disso, o NetApp Cloud Volumes ONTAP é executado em todos os três principais provedores de nuvem.

Escalonamento de desempenho - Escalonamento vertical

Os recursos de expansão permitem que você adicione unidades de disco aos sistemas AFF, FAS e E-Series quando precisar de capacidade de armazenamento adicional. Com o Cloud Volumes ONTAP, dimensionar o armazenamento para o nível PB é uma combinação de dois fatores: hierarquizar dados usados com pouca frequência para armazenamento de objetos a partir do armazenamento em bloco e empilhar licenças do Cloud Volumes ONTAP sem computação adicional.

Vários protocolos

Os sistemas NetApp oferecem suporte à maioria dos protocolos para implantações do Hadoop, incluindo SAS, iSCSI, FCP, InfiniBand e NFS.

Soluções operacionais e suportadas

As soluções Hadoop descritas neste documento são suportadas pela NetApp. Essas soluções também são certificadas pelos principais distribuidores do Hadoop. Para obter informações, consulte o "Hortonworks" site e o Cloudera "certificação" e "parceiro" sites.

Público-alvo

O mundo da análise e da ciência de dados abrange diversas disciplinas em TI e negócios:

- O cientista de dados precisa de flexibilidade para usar as ferramentas e bibliotecas de sua escolha.
- O engenheiro de dados precisa saber como os dados fluem e onde eles residem.

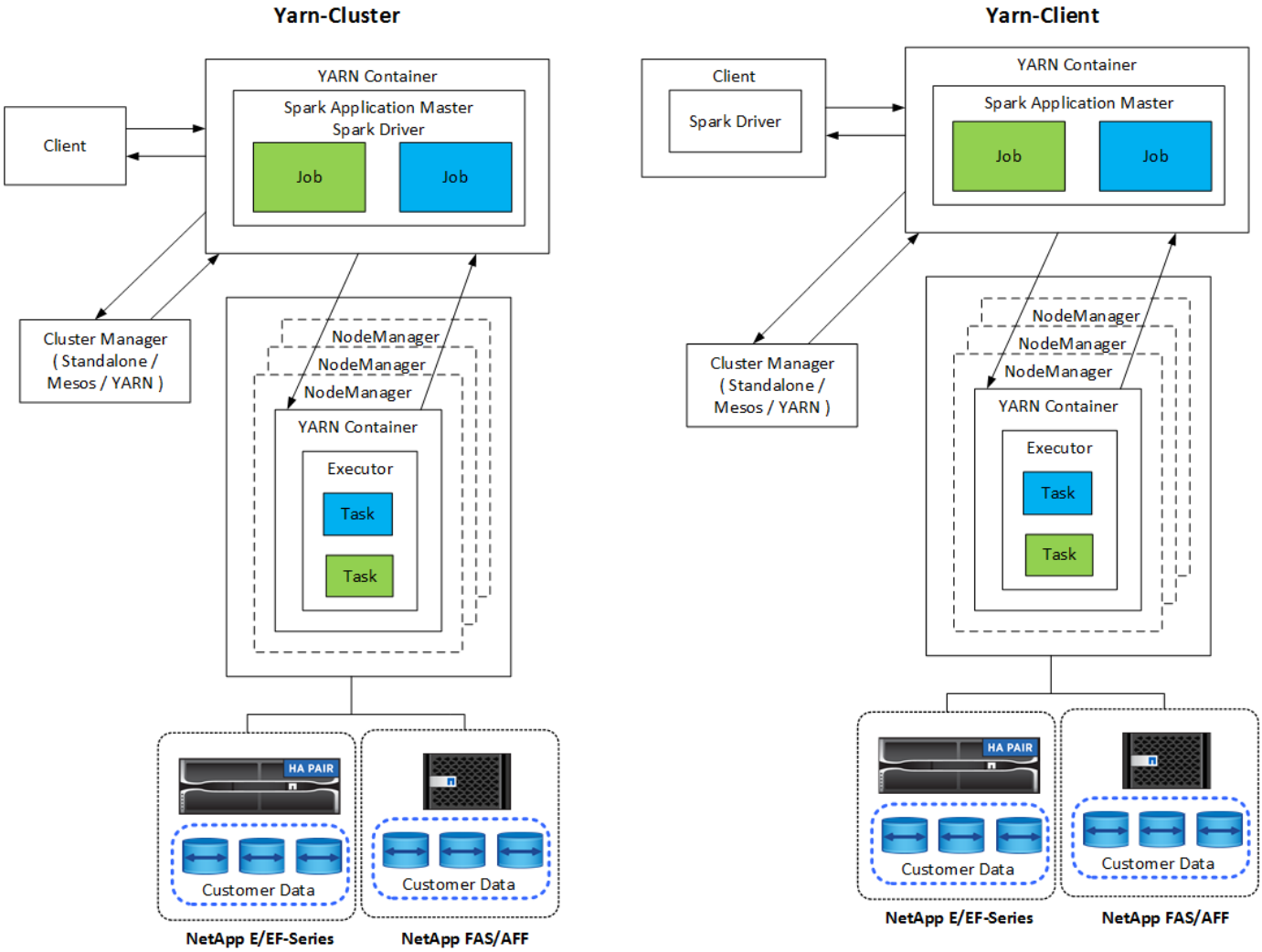
- Um engenheiro de DevOps precisa de ferramentas para integrar novos aplicativos de IA e ML em seus pipelines de CI e CD.
- Administradores e arquitetos de nuvem devem ser capazes de configurar e gerenciar recursos de nuvem híbrida.
- Usuários empresariais querem ter acesso a aplicativos de análise, IA, ML e DL.

Neste relatório técnico, descrevemos como NetApp AFF, E-Series, StorageGRID, NFS direct access, Apache Spark, Horovod e Keras ajudam cada uma dessas funções a agregar valor aos negócios.

Tecnologia de soluções

Apache Spark é uma estrutura de programação popular para escrever aplicativos Hadoop que funciona diretamente com o Hadoop Distributed File System (HDFS). O Spark está pronto para produção, suporta processamento de dados de streaming e é mais rápido que o MapReduce. O Spark tem cache de dados na memória configurável para iteração eficiente, e o shell do Spark é interativo para aprendizado e exploração de dados. Com o Spark, você pode criar aplicativos em Python, Scala ou Java. Os aplicativos Spark consistem em um ou mais trabalhos que têm uma ou mais tarefas.

Cada aplicativo Spark tem um driver Spark. No modo YARN-Client, o driver é executado no cliente localmente. No modo YARN-Cluster, o driver é executado no cluster no mestre do aplicativo. No modo de cluster, o aplicativo continua em execução mesmo se o cliente se desconectar.



Existem três gerenciadores de cluster:

- **Autônomo.** Este gerenciador faz parte do Spark, o que facilita a configuração de um cluster.
- **Apache Mesos.** Este é um gerenciador de cluster geral que também executa o MapReduce e outros aplicativos.
- **Hadoop YARN.** Este é um gerenciador de recursos no Hadoop 3.

O conjunto de dados distribuídos resilientes (RDD) é o principal componente do Spark. O RDD recria os dados perdidos e ausentes a partir dos dados armazenados na memória do cluster e armazena os dados iniciais que vêm de um arquivo ou são criados programaticamente. RDDs são criados a partir de arquivos, dados na memória ou outro RDD. A programação Spark realiza duas operações: transformação e ações. A transformação cria um novo RDD com base em um existente. Ações retornam um valor de um RDD.

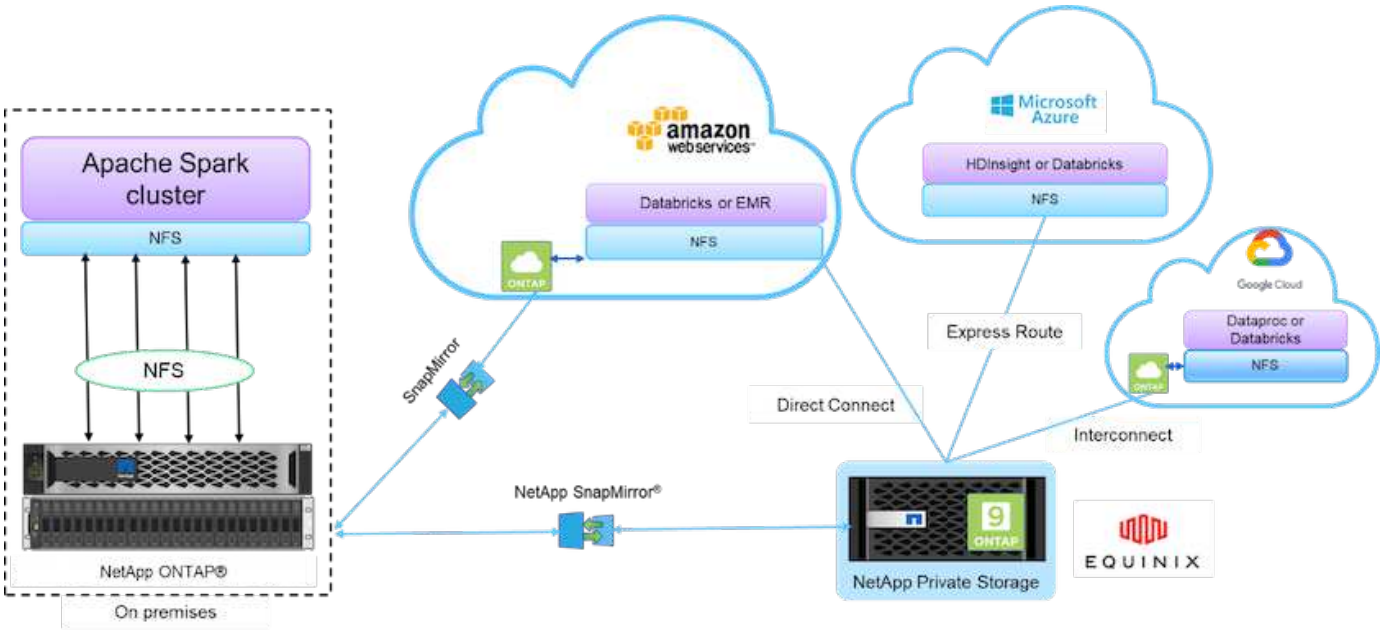
Transformações e ações também se aplicam a conjuntos de dados e quadros de dados do Spark. Um conjunto de dados é uma coleção distribuída de dados que fornece os benefícios de RDDs (tipagem forte, uso de funções lambda) com os benefícios do mecanismo de execução otimizado do Spark SQL. Um conjunto de dados pode ser construído a partir de objetos JVM e então manipulado usando transformações funcionais (mapa, flatMap, filtro e assim por diante). Um DataFrame é um conjunto de dados organizado em colunas nomeadas. É conceitualmente equivalente a uma tabela em um banco de dados relacional ou a um quadro de dados em R/Python. Os DataFrames podem ser construídos a partir de uma ampla gama de fontes, como arquivos de dados estruturados, tabelas no Hive/HBase, bancos de dados externos no local ou na nuvem, ou RDDs existentes.

Os aplicativos Spark incluem um ou mais trabalhos Spark. Os trabalhos executam tarefas em executores, e os executores são executados em contêineres YARN. Cada executor é executado em um único contêiner, e os executores existem durante toda a vida de um aplicativo. Um executor é corrigido depois que o aplicativo é iniciado, e o YARN não redimensiona o contêiner já alocado. Um executor pode executar tarefas simultaneamente em dados na memória.

Visão geral das soluções NetApp Spark

A NetApp tem três portfólios de armazenamento: FAS/ AFF, E-Series e Cloud Volumes ONTAP. Validamos o AFF e o E-Series com sistema de armazenamento ONTAP para soluções Hadoop com Apache Spark.

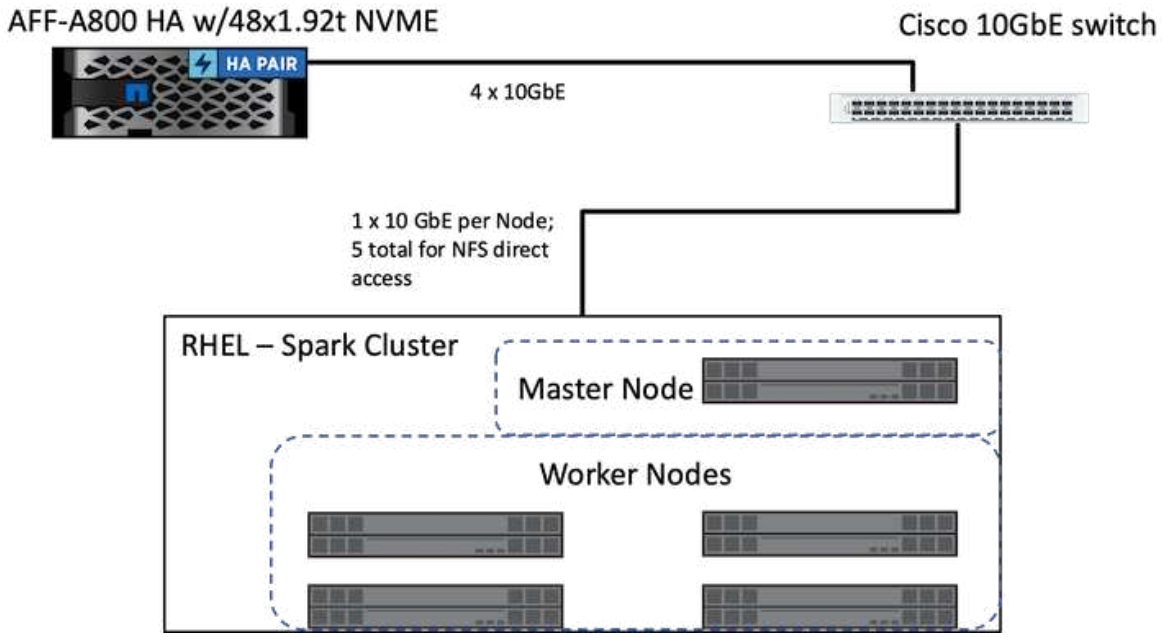
A estrutura de dados alimentada pela NetApp integra serviços e aplicativos de gerenciamento de dados (blocos de construção) para acesso, controle, proteção e segurança de dados, conforme mostrado na figura abaixo.



Os blocos de construção na figura acima incluem:

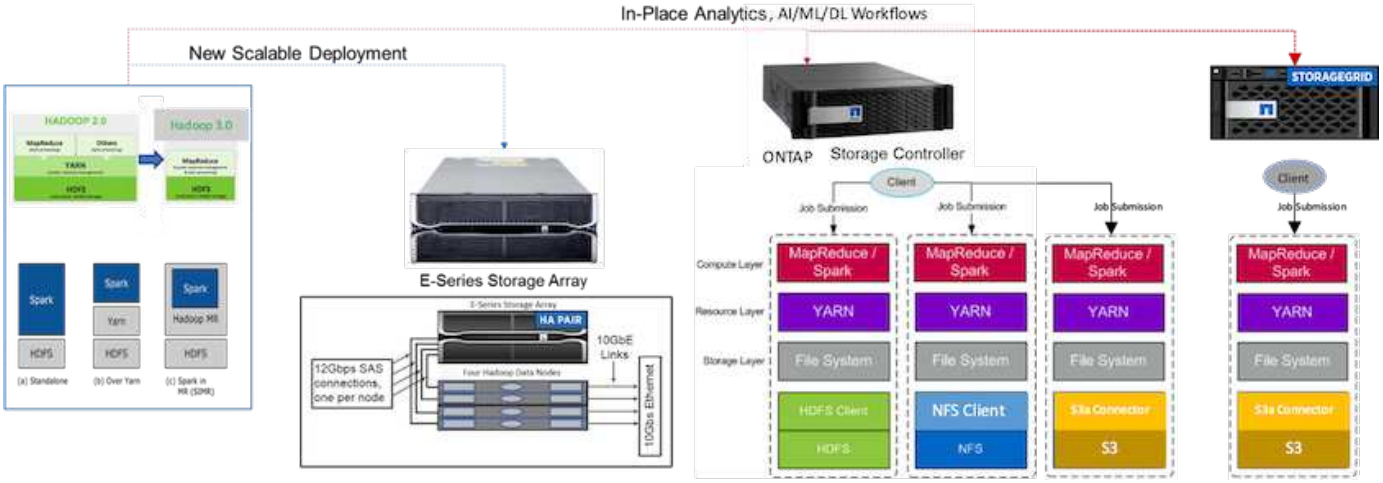
- * Acesso direto ao NetApp NFS.* Fornece os clusters Hadoop e Spark mais recentes com acesso direto aos volumes NetApp NFS sem requisitos adicionais de software ou driver.
- * NetApp Cloud Volumes ONTAP e Google Cloud NetApp Volumes.* Armazenamento conectado definido por software baseado em ONTAP em execução no Amazon Web Services (AWS) ou no Azure NetApp Files (ANF) nos serviços de nuvem do Microsoft Azure.
- * Tecnologia NetApp SnapMirror .* Fornece recursos de proteção de dados entre instâncias locais e ONTAP Cloud ou NPS.
- **Provedores de serviços em nuvem.** Esses provedores incluem AWS, Microsoft Azure, Google Cloud e IBM Cloud.
- **PaaS.** Serviços de análise baseados em nuvem, como Amazon Elastic MapReduce (EMR) e Databricks na AWS, bem como Microsoft Azure HDInsight e Azure Databricks.

A figura a seguir descreve a solução Spark com armazenamento NetApp .

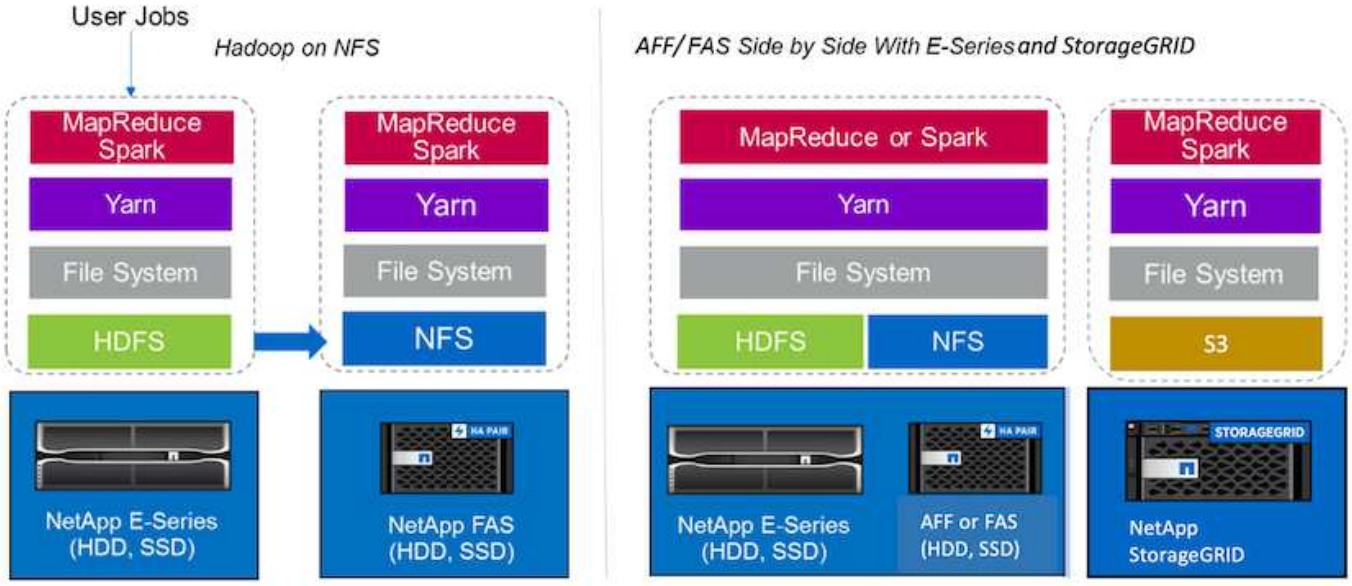


A solução ONTAP Spark usa o protocolo de acesso direto NetApp NFS para análises no local e fluxos de trabalho de IA, ML e DL usando acesso a dados de produção existentes. Os dados de produção disponíveis para os nós do Hadoop são exportados para executar tarefas analíticas e de IA, ML e DL no local. Você pode acessar dados para processar em nós do Hadoop com acesso direto ao NetApp NFS ou sem ele. No Spark com o autônomo ou `yarn` gerenciador de cluster, você pode configurar um volume NFS usando `file://<target_volume>` . Validamos três casos de uso com diferentes conjuntos de dados. Os detalhes dessas validações são apresentados na seção "Resultados dos testes". (referência externa)

A figura a seguir descreve o posicionamento do armazenamento do NetApp Apache Spark/Hadoop.



Identificamos os recursos exclusivos da solução E-Series Spark, da solução AFF/ FAS ONTAP Spark e da solução StorageGRID Spark, e realizamos validação e testes detalhados. Com base em nossas observações, a NetApp recomenda a solução E-Series para instalações greenfield e novas implantações escaláveis e a solução AFF/ FAS para análises no local, cargas de trabalho de IA, ML e DL usando dados NFS existentes, e StorageGRID para IA, ML e DL e análises de dados modernas quando o armazenamento de objetos for necessário.



Um data lake é um repositório de armazenamento para grandes conjuntos de dados em formato nativo que pode ser usado para tarefas de análise, IA, ML e DL. Criamos um repositório de data lake para as soluções Spark E-Series, AFF/ FAS e StorageGRID SG6060. O sistema E-Series fornece acesso HDFS ao cluster Hadoop Spark, enquanto os dados de produção existentes são acessados por meio do protocolo de acesso direto NFS ao cluster Hadoop. Para conjuntos de dados que residem no armazenamento de objetos, o NetApp StorageGRID fornece acesso seguro S3 e S3a.

Resumo do caso de uso

Esta página descreve as diferentes áreas nas quais esta solução pode ser usada.

Dados de streaming

O Apache Spark pode processar dados de streaming, que são usados para processos de extração, transformação e carregamento (ETL) de streaming; enriquecimento de dados; disparo de detecção de eventos; e análise de sessão complexa:

- **Streaming ETL.** Os dados são continuamente limpos e agregados antes de serem enviados aos armazenamentos de dados. A Netflix usa o streaming Kafka e Spark para criar uma solução de recomendação de filmes e monitoramento de dados online em tempo real que pode processar bilhões de eventos por dia de diferentes fontes de dados. No entanto, o ETL tradicional para processamento em lote é tratado de forma diferente. Esses dados são lidos primeiro e depois convertidos em um formato de banco de dados antes de serem gravados nele.
- **Enriquecimento de dados.** O Spark Streaming enriquece os dados ao vivo com dados estáticos para permitir uma análise de dados mais em tempo real. Por exemplo, os anunciantes on-line podem fornecer anúncios personalizados e segmentados, direcionados com base em informações sobre o comportamento do cliente.
- **Detecção de evento de gatilho.** O Spark Streaming permite que você detecte e responda rapidamente a comportamentos incomuns que podem indicar problemas potencialmente sérios. Por exemplo, instituições financeiras usam gatilhos para detectar e interromper transações fraudulentas, e hospitais usam gatilhos para detectar alterações perigosas de saúde detectadas nos sinais vitais de um paciente.
- **Análise complexa de sessão.** O Spark Streaming coleta eventos como atividade do usuário após login em um site ou aplicativo, que são então agrupados e analisados. Por exemplo, a Netflix usa essa

funcionalidade para fornecer recomendações de filmes em tempo real.

Para obter mais configurações de dados de streaming, verificação do Confluent Kafka e testes de desempenho, consulte"TR-4912: Diretrizes de práticas recomendadas para armazenamento em camadas do Confluent Kafka com NetApp" .

Aprendizado de máquina

A estrutura integrada do Spark ajuda você a executar consultas repetidas em conjuntos de dados usando a biblioteca de aprendizado de máquina (MLlib). O MLlib é usado em áreas como agrupamento, classificação e redução de dimensionalidade para algumas funções comuns de big data, como inteligência preditiva, segmentação de clientes para fins de marketing e análise de sentimentos. O MLlib é usado em segurança de rede para conduzir inspeções em tempo real de pacotes de dados em busca de indícios de atividade maliciosa. Ele ajuda os provedores de segurança a aprender sobre novas ameaças e a ficar à frente dos hackers, ao mesmo tempo em que protegem seus clientes em tempo real.

Aprendizado profundo

TensorFlow é uma estrutura de aprendizado profundo popular usada em todo o setor. O TensorFlow suporta treinamento distribuído em um cluster de CPU ou GPU. Este treinamento distribuído permite que os usuários o executem em uma grande quantidade de dados com muitas camadas profundas.

Até bem pouco tempo, se quiséssemos usar o TensorFlow com o Apache Spark, precisávamos executar todo o ETL necessário para o TensorFlow no PySpark e então gravar os dados no armazenamento intermediário. Esses dados seriam então carregados no cluster TensorFlow para o processo de treinamento real. Esse fluxo de trabalho exigia que o usuário mantivesse dois clusters diferentes, um para ETL e outro para treinamento distribuído do TensorFlow. Executar e manter vários clusters costumava ser tedioso e demorado.

DataFrames e RDD em versões anteriores do Spark não eram adequados para aprendizado profundo porque o acesso aleatório era limitado. No Spark 3.0 com projeto hidrogênio, o suporte nativo para as estruturas de aprendizado profundo é adicionado. Essa abordagem permite o agendamento não baseado em MapReduce no cluster Spark.

Análise interativa

O Apache Spark é rápido o suficiente para executar consultas exploratórias sem amostragem com linguagens de desenvolvimento diferentes do Spark, incluindo SQL, R e Python. O Spark usa ferramentas de visualização para processar dados complexos e visualizá-los interativamente. O Spark com streaming estruturado executa consultas interativas em dados ao vivo em análises da web, o que permite que você execute consultas interativas na sessão atual de um visitante da web.

Sistema de recomendação

Ao longo dos anos, os sistemas de recomendação trouxeram mudanças tremendas para nossas vidas, à medida que empresas e consumidores responderam a mudanças drásticas nas compras on-line, no entretenimento on-line e em muitos outros setores. De fato, esses sistemas estão entre as histórias de sucesso mais evidentes da IA na produção. Em muitos casos de uso prático, sistemas de recomendação são combinados com IA conversacional ou chatbots interligados a um backend de PNL para obter informações relevantes e produzir inferências úteis.

Hoje em dia, muitos varejistas estão adotando novos modelos de negócios, como comprar on-line e retirar na loja, retirada na calçada, autoatendimento, escanear e levar, e muito mais. Esses modelos se tornaram populares durante a pandemia da COVID-19, tornando as compras mais seguras e convenientes para os consumidores. A IA é crucial para essas crescentes tendências digitais, que são influenciadas pelo

comportamento do consumidor e vice-versa. Para atender às crescentes demandas dos consumidores, aprimorar a experiência do cliente, melhorar a eficiência operacional e aumentar a receita, a NetApp ajuda seus clientes e empresas empresariais a usar algoritmos de aprendizado de máquina e aprendizado profundo para projetar sistemas de recomendação mais rápidos e precisos.

Existem várias técnicas populares usadas para fornecer recomendações, incluindo filtragem colaborativa, sistemas baseados em conteúdo, modelo de recomendação de aprendizado profundo (DLRM) e técnicas híbridas. Anteriormente, os clientes utilizavam o PySpark para implementar filtragem colaborativa para criar sistemas de recomendação. O Spark MLlib implementa mínimos quadrados alternados (ALS) para filtragem colaborativa, um algoritmo muito popular entre empresas antes do surgimento do DLRM.

Processamento de linguagem natural

A IA conversacional, possibilitada pelo processamento de linguagem natural (PLN), é o ramo da IA que ajuda os computadores a se comunicarem com os humanos. A PNL é predominante em todos os setores e em muitos casos de uso, desde assistentes inteligentes e chatbots até pesquisa do Google e texto preditivo. De acordo com um "Gartner" previsão é que até 2022, 70% das pessoas estarão interagindo com plataformas de IA conversacional diariamente. Para uma conversa de alta qualidade entre um humano e uma máquina, as respostas devem ser rápidas, inteligentes e naturais.

Os clientes precisam de uma grande quantidade de dados para processar e treinar seus modelos de PNL e reconhecimento automático de fala (ASR). Eles também precisam mover dados pela borda, núcleo e nuvem, e precisam do poder de realizar inferências em milissegundos para estabelecer comunicação natural com humanos. O NetApp AI e o Apache Spark são uma combinação ideal para computação, armazenamento, processamento de dados, treinamento de modelos, ajuste fino e implantação.

Análise de sentimentos é um campo de estudo dentro da PNL em que sentimentos positivos, negativos ou neutros são extraídos do texto. A análise de sentimentos tem uma variedade de casos de uso, desde determinar o desempenho dos funcionários do centro de suporte em conversas com os chamadores até fornecer respostas automatizadas e apropriadas do chatbot. Ele também tem sido usado para prever o preço das ações de uma empresa com base nas interações entre representantes da empresa e o público em teleconferências de resultados trimestrais. Além disso, a análise de sentimentos pode ser usada para determinar a opinião do cliente sobre os produtos, serviços ou suporte fornecido pela marca.

Nós usamos o "Spark NLP" biblioteca de "Laboratórios John Snow" para carregar pipelines pré-treinados e modelos de Representações de Codificadores Bidirecionais de Transformadores (BERT), incluindo "sentimento de notícias financeiras" e "FinBERT" , realizando tokenização, reconhecimento de entidades nomeadas, treinamento de modelos, ajuste e análise de sentimentos em escala. Spark NLP é a única biblioteca NLP de código aberto em produção que oferece transformadores de última geração, como BERT, ALBERT, ELECTRA, XLNet, DistilBERT, RoBERTa, DeBERTa, XLM-RoBERTa, Longformer, ELMO, Universal Sentence Encoder, Google T5, MarianMT e GPT2. A biblioteca funciona não apenas em Python e R, mas também no ecossistema JVM (Java, Scala e Kotlin) em escala, estendendo o Apache Spark nativamente.

Principais casos de uso e arquiteturas de IA, ML e DL

Os principais casos de uso e metodologia de IA, ML e DL podem ser divididos nas seguintes seções:

Pipelines Spark NLP e inferência distribuída TensorFlow

A lista a seguir contém as bibliotecas de PNL de código aberto mais populares que foram adotadas pela comunidade de ciência de dados em diferentes níveis de desenvolvimento:

- "Kit de ferramentas de linguagem natural (NLTK)" . O kit de ferramentas completo para todas as técnicas de PNL. Ela vem sendo mantida desde o início dos anos 2000.
- "TextBlob" . Uma API Python de ferramentas de PNL fácil de usar, construída sobre NLTK e Pattern.
- "Stanford Core NLP" . Serviços e pacotes de PNL em Java desenvolvidos pelo Stanford NLP Group.
- "Gensim" . O Topic Modelling for Humans começou como uma coleção de scripts Python para o projeto da Biblioteca Matemática Digital Tcheca.
- "SpaCy" . Fluxos de trabalho de PNL industrial de ponta a ponta com Python e Cython com aceleração de GPU para transformadores.
- "Texto rápido" . Uma biblioteca de PNL gratuita, leve e de código aberto para aprendizado de incorporação de palavras e classificação de frases, criada pelo laboratório de pesquisa de IA (FAIR) do Facebook.

O Spark NLP é uma solução única e unificada para todas as tarefas e requisitos de PNL que permite software de PNL escalável, de alto desempenho e alta precisão para casos de uso de produção real. Ela aproveita a aprendizagem por transferência e implementa os algoritmos e modelos mais modernos em pesquisas e em todos os setores. Devido à falta de suporte total do Spark para as bibliotecas acima, o Spark NLP foi construído sobre "Spark ML" para aproveitar o mecanismo de processamento de dados distribuídos na memória de uso geral do Spark como uma biblioteca de PNL de nível empresarial para fluxos de trabalho de produção de missão crítica. Seus anotadores utilizam algoritmos baseados em regras, aprendizado de máquina e TensorFlow para impulsionar implementações de aprendizado profundo. Isso abrange tarefas comuns de PNL, incluindo, mas não se limitando a tokenização, lematização, definição de radicais, marcação de classes gramaticais, reconhecimento de entidades nomeadas, verificação ortográfica e análise de sentimentos.

Representações de codificador bidirecional de transformadores (BERT) é uma técnica de aprendizado de máquina baseada em transformadores para PNL. Popularizou o conceito de pré-treinamento e ajuste fino. A arquitetura do transformador no BERT originou-se da tradução automática, que modela dependências de longo prazo melhor do que modelos de linguagem baseados em Redes Neurais Recorrentes (RNN). Ele também introduziu a tarefa de Modelagem de Linguagem Mascarada (MLM), onde 15% aleatórios de todos os tokens são mascarados e o modelo os prevê, permitindo verdadeira bidirecionalidade.

A análise do sentimento financeiro é desafiadora devido à linguagem especializada e à falta de dados rotulados nesse domínio. FinBERT, um modelo de linguagem baseado em BERT pré-treinado, foi adaptado ao domínio em "Reuters TRC2" , um corpus financeiro e ajustado com dados rotulados ("Banco de Frases Financeiro") para classificação de sentimento financeiro. Pesquisadores extraíram 4.500 frases de artigos de notícias com termos financeiros. Em seguida, 16 especialistas e estudantes de mestrado com formação em finanças rotularam as frases como positivas, neutras e negativas. Construímos um fluxo de trabalho Spark de ponta a ponta para analisar o sentimento das transcrições de teleconferências sobre os lucros das 10 maiores empresas da NASDAQ de 2016 a 2020 usando FinBERT e dois outros pipelines pré-treinados, "Explicar Documento DL") do Spark NLP.

O mecanismo de aprendizado profundo subjacente ao Spark NLP é o TensorFlow, uma plataforma de ponta a ponta e de código aberto para aprendizado de máquina que permite a construção fácil de modelos, produção robusta de ML em qualquer lugar e experimentação poderosa para pesquisa. Portanto, ao executar nossos pipelines no Spark `yarn cluster` modo, estávamos essencialmente executando o TensorFlow distribuído com paralelismo de dados e modelos em um nó mestre e vários nós de trabalho, bem como armazenamento conectado à rede montado no cluster.

Treinamento distribuído Horovod

A validação principal do Hadoop para desempenho relacionado ao MapReduce é realizada com TeraGen, TeraSort, TeraValidate e DFSIO (leitura e gravação). Os resultados da validação do TeraGen e do TeraSort são apresentados em "Solução NetApp E-Series para Hadoop" e na seção "Storage Tiering" para AFF.

Com base nas solicitações dos clientes, consideramos o treinamento distribuído com Spark um dos mais importantes entre os vários casos de uso. Neste documento, utilizamos o "Hovorod no Spark" para validar o desempenho do Spark com soluções NetApp locais, nativas da nuvem e de nuvem híbrida usando controladores de armazenamento NetApp All Flash FAS (AFF), Azure NetApp Files e StorageGRID.

O pacote Horovod no Spark fornece um wrapper conveniente em torno do Horovod que simplifica a execução de cargas de trabalho de treinamento distribuídas em clusters Spark, permitindo um loop de design de modelo preciso no qual o processamento de dados, o treinamento do modelo e a avaliação do modelo são todos feitos no Spark, onde residem os dados de treinamento e inferência.

Há duas APIs para executar o Horovod no Spark: uma API Estimator de alto nível e uma API Run de nível inferior. Embora ambos usem o mesmo mecanismo subjacente para iniciar o Horovod nos executores do Spark, a API Estimator abstrai o processamento de dados, o loop de treinamento do modelo, o ponto de verificação do modelo, a coleta de métricas e o treinamento distribuído. Usamos Horovod Spark Estimators, TensorFlow e Keras para uma preparação de dados de ponta a ponta e um fluxo de trabalho de treinamento distribuído com base no "Vendas da loja Kaggle Rossmann" concorrência.

O roteiro `keras_spark_horovod_rossmann_estimator.py` pode ser encontrado na seção"Scripts Python para cada caso de uso principal." Ele contém três partes:

- A primeira parte executa várias etapas de pré-processamento de dados em um conjunto inicial de arquivos CSV fornecidos pelo Kaggle e coletados pela comunidade. Os dados de entrada são separados em um conjunto de treinamento com um `Validation` subconjunto e um conjunto de dados de teste.
- A segunda parte define um modelo de Rede Neural Profunda (DNN) de Keras com função de ativação sigmoide logarítmica e um otimizador de Adam, e realiza o treinamento distribuído do modelo usando Horovod no Spark.
- A terceira parte realiza a previsão no conjunto de dados de teste usando o melhor modelo que minimiza o erro absoluto médio geral do conjunto de validação. Em seguida, ele cria um arquivo CSV de saída.

Veja a seção"Aprendizado de máquina" para vários resultados de comparação de tempo de execução.

Aprendizado profundo multi-trabalhador usando Keras para previsão de CTR

Com os avanços recentes em plataformas e aplicativos de ML, muita atenção agora está voltada para o aprendizado em escala. A taxa de cliques (CTR) é definida como o número médio de cliques por cem impressões de anúncios on-line (expresso como uma porcentagem). Ela é amplamente adotada como uma métrica-chave em vários setores e casos de uso, incluindo marketing digital, varejo, comércio eletrônico e provedores de serviços. Para obter mais detalhes sobre as aplicações do CTR e os resultados do desempenho do treinamento distribuído, consulte o"Modelos de aprendizado profundo para desempenho de previsão de CTR" seção.

Neste relatório técnico, usamos uma variação do "Conjunto de dados de registros de cliques de terabytes da Criteo" (consulte TR-4904) para aprendizado profundo distribuído por vários trabalhadores usando Keras para criar um fluxo de trabalho Spark com modelos Deep and Cross Network (DCN), comparando seu desempenho em termos de função de erro de perda de log com um modelo de regressão logística Spark ML de base. O DCN captura eficientemente interações de recursos eficazes de graus limitados, aprende interações altamente não lineares, não requer engenharia de recursos manual ou pesquisa exaustiva e tem baixo custo computacional.

Os dados para sistemas de recomendação em escala da web são, em sua maioria, discretos e categóricos, o que leva a um espaço de recursos grande e esparso, o que é desafiador para a exploração de recursos. Isso limitou a maioria dos sistemas de larga escala a modelos lineares, como a regressão logística. No entanto, identificar características frequentemente preditivas e, ao mesmo tempo, explorar características cruzadas raras ou invisíveis é a chave para fazer boas previsões. Os modelos lineares são simples, interpretáveis e

fáceis de escalar, mas são limitados em seu poder expressivo.

Por outro lado, recursos transversais demonstraram ser significativos na melhoria da expressividade dos modelos. Infelizmente, muitas vezes é necessária engenharia de recursos manual ou pesquisa exaustiva para identificar tais recursos. Generalizar para interações de recursos invisíveis costuma ser difícil. Usar uma rede neural cruzada como a DCN evita a engenharia de recursos específicos da tarefa ao aplicar explicitamente o cruzamento de recursos de forma automática. A rede cruzada consiste em múltiplas camadas, onde o maior grau de interações é provavelmente determinado pela profundidade da camada. Cada camada produz interações de ordem superior com base nas existentes e mantém as interações das camadas anteriores.

Uma rede neural profunda (DNN) tem a promessa de capturar interações muito complexas entre recursos. Entretanto, comparado ao DCN, ele requer quase uma ordem de magnitude a mais de parâmetros, não é capaz de formar recursos cruzados explicitamente e pode falhar em aprender eficientemente alguns tipos de interações de recursos. A rede cruzada é eficiente em termos de memória e fácil de implementar. O treinamento conjunto dos componentes cruzados e DNN captura com eficiência interações de recursos preditivos e oferece desempenho de última geração no conjunto de dados CTR da Criteo.

Um modelo DCN começa com uma camada de incorporação e empilhamento, seguida por uma rede cruzada e uma rede profunda em paralelo. Estes, por sua vez, são seguidos por uma camada de combinação final que combina as saídas das duas redes. Seus dados de entrada podem ser um vetor com recursos esparsos e densos. No Spark, as bibliotecas contêm o tipo `SparseVector` . Portanto, é importante que os usuários diferenciem os dois e tenham cuidado ao chamar suas respectivas funções e métodos. Em sistemas de recomendação em escala web, como a previsão de CTR, as entradas são principalmente características categóricas, por exemplo `'country=usa'` . Tais características são frequentemente codificadas como vetores one-hot, por exemplo, `'[0,1,0, …]'` . Codificação one-hot (OHE) com `SparseVector` é útil ao lidar com conjuntos de dados do mundo real com vocabulários em constante mudança e crescimento. Nós modificamos exemplos em "DeepCTR" para processar vocabulários grandes, criando vetores de incorporação na camada de incorporação e empilhamento do nosso DCN.

O "Conjunto de dados de anúncios gráficos da Criteo" prevê a taxa de cliques dos anúncios. Ele tem 13 características inteiras e 26 características categóricas, nas quais cada categoria tem uma alta cardinalidade. Para este conjunto de dados, uma melhoria de 0,001 na perda logarítmica é praticamente significativa devido ao grande tamanho de entrada. Uma pequena melhoria na precisão da previsão para uma grande base de usuários pode potencialmente levar a um grande aumento na receita de uma empresa. O conjunto de dados contém 11 GB de registros de usuários de um período de 7 dias, o que equivale a cerca de 41 milhões de registros. Nós usamos Spark `dataFrame.randomSplit()function` dividir aleatoriamente os dados para treinamento (80%), validação cruzada (10%) e os 10% restantes para teste.

O DCN foi implementado no TensorFlow com Keras. Há quatro componentes principais na implementação do processo de treinamento de modelo com DCN:

- **Processamento e incorporação de dados.** Os recursos de valor real são normalizados pela aplicação de uma transformação logarítmica. Para recursos categóricos, incorporamos os recursos em vetores densos de dimensão 6×(cardinalidade da categoria)1/4. Concatenar todos os embeddings resulta em um vetor de dimensão 1026.
- **Otimização.** Aplicamos otimização estocástica de minilote com o otimizador Adam. O tamanho do lote foi definido como 512. A normalização em lote foi aplicada à rede profunda e a norma de corte de gradiente foi definida em 100.
- **Regularização.** Usamos a parada antecipada, pois a regularização ou o abandono da L2 não se mostraram eficazes.
- **Hiperparâmetros.** Relatamos resultados com base em uma pesquisa de grade sobre o número de camadas ocultas, o tamanho da camada oculta, a taxa de aprendizado inicial e o número de camadas cruzadas. O número de camadas ocultas variou de 2 a 5, com tamanhos de camadas ocultas variando de

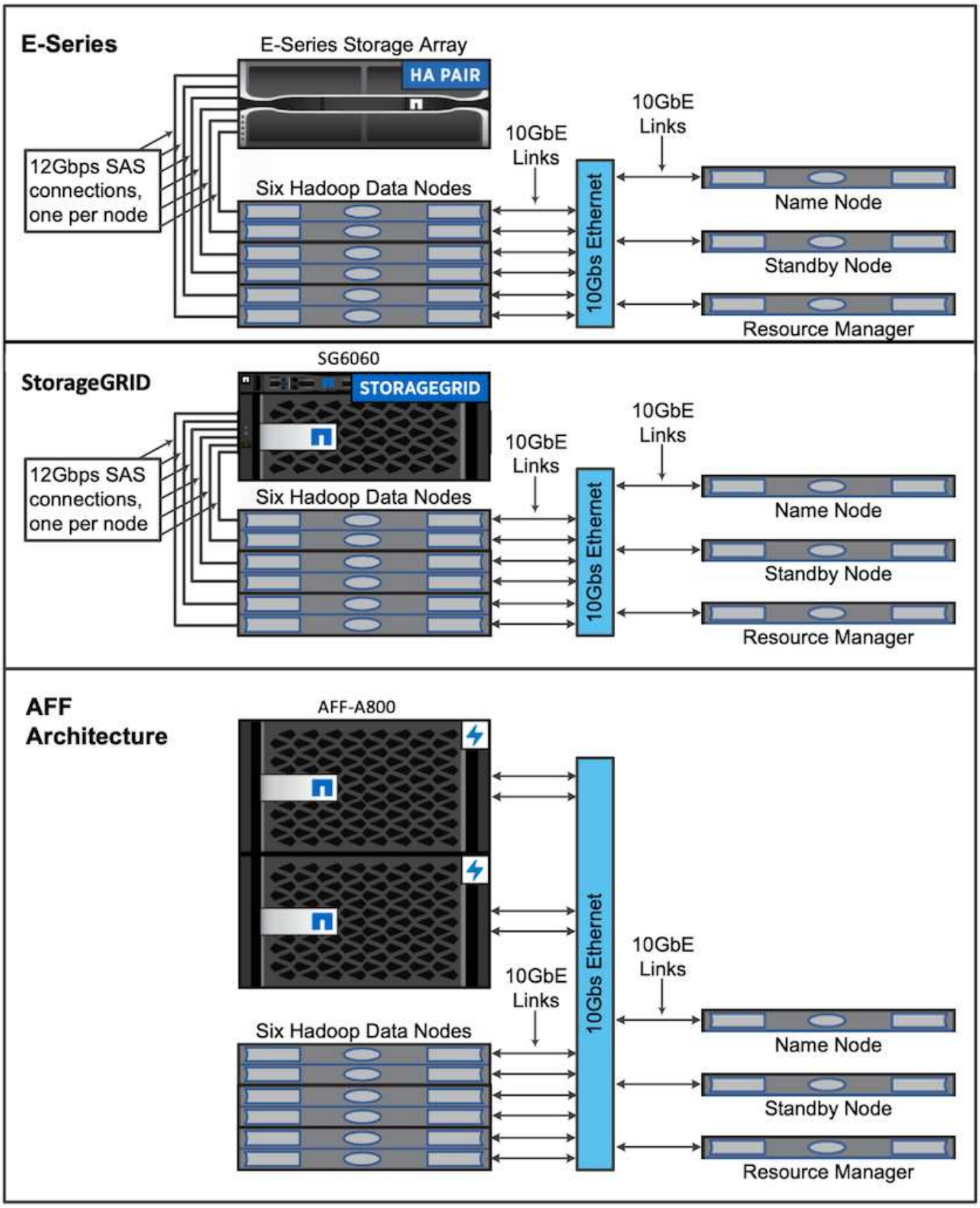
32 a 1024. Para DCN, o número de camadas cruzadas foi de 1 a 6. A taxa de aprendizagem inicial foi ajustada de 0,0001 para 0,001 com incrementos de 0,0001. Todos os experimentos aplicaram parada antecipada na etapa de treinamento 150.000, além da qual o overfitting começou a ocorrer.

Além do DCN, também testamos outros modelos populares de aprendizado profundo para previsão de CTR, incluindo "DeepFM" , "AutoInt" , e "DCN v2" .

Arquiteturas usadas para validação

Para esta validação, usamos quatro nós de trabalho e um nó mestre com um par AFF-A800 HA. Todos os membros do cluster estavam conectados por meio de switches de rede de 10 GbE.

Para esta validação da solução NetApp Spark, usamos três controladores de armazenamento diferentes: o E5760, o E5724 e o AFF-A800. Os controladores de armazenamento da série E foram conectados a cinco nós de dados com conexões SAS de 12 Gbps. O controlador de armazenamento AFF HA-pair fornece volumes NFS exportados por meio de conexões de 10 GbE para nós de trabalho do Hadoop. Os membros do cluster Hadoop foram conectados por meio de conexões de 10 GbE nas soluções Hadoop E-Series, AFF e StorageGRID .



Resultados dos testes

Usamos os scripts TeraSort e TeraValidate na ferramenta de benchmarking TeraGen para

medir a validação de desempenho do Spark com as configurações E5760, E5724 e AFF-A800. Além disso, três casos de uso principais foram testados: pipelines Spark NLP e treinamento distribuído TensorFlow, treinamento distribuído Horovod e aprendizado profundo de vários trabalhadores usando Keras para previsão de CTR com DeepFM.

Para validação do E-Series e do StorageGRID , usamos o fator de replicação 2 do Hadoop. Para validação do AFF , usamos apenas uma fonte de dados.

A tabela a seguir lista a configuração de hardware para a validação de desempenho do Spark.

Tipo	Nós de trabalho do Hadoop	Tipo de unidade	Unidades por nó	Controlador de armazenamento
SG6060	4	SAS	12	Par único de alta disponibilidade (HA)
E5760	4	SAS	60	Par único de HA
E5724	4	SAS	24	Par único de HA
AFF800	4	SSD	6	Par único de HA

A tabela a seguir lista os requisitos de software.

Software	Versão
RHEL	7,9
Ambiente de execução OpenJDK	1.8.0
VM do servidor OpenJDK de 64 bits	25,302
Git	2.24.1
GCC/G++	11.2.1
Fagulha	3.2.1
PySpark	3.1.2
SparkNLP	3.4.2
TensorFlow	2.9.0
Keras	2.9.0
Horovod	0.24.3

Análise de sentimento financeiro

Nós publicamos"TR-4910: Análise de sentimentos das comunicações com o cliente com a NetApp AI" , no qual um pipeline de IA de conversação de ponta a ponta foi construído usando o "Kit de ferramentas NetApp DataOps" , armazenamento AFF e sistema NVIDIA DGX. O pipeline executa processamento de sinal de áudio em lote, reconhecimento automático de fala (ASR), aprendizagem de transferência e análise de sentimentos, aproveitando o DataOps Toolkit, "NVIDIA Riva SDK" , e o "Estrutura do Tao" . Expandindo o caso de uso da análise de sentimentos para o setor de serviços financeiros, criamos um fluxo de trabalho SparkNLP, carregamos três modelos BERT para várias tarefas de PNL, como reconhecimento de entidade nomeada, e obtivemos sentimentos em nível de frase para as teleconferências de resultados trimestrais das 10 maiores empresas da NASDAQ.

O seguinte script `sentiment_analysis_spark. py` usa o modelo FinBERT para processar transcrições no HDFS e produzir contagens de sentimentos positivos, neutros e negativos, conforme mostrado na tabela a seguir:

```
-bash-4.2$ time ~/anaconda3/bin/spark-submit
--packages com.johnsnowlabs.nlp:spark-nlp_2.12:3.4.3
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
--conf spark.driver.extraJavaOptions="-Xss10m -XX:MaxPermSize=1024M"
--conf spark.executor.extraJavaOptions="-Xss10m -XX:MaxPermSize=512M"
/sparkusecase/tr-4570-nlp/sentiment_analysis_spark.py
hdfs:///data1/Transcripts/
> ./sentiment_analysis_hdfs.log 2>&1
real13m14.300s
user557m11.319s
sys4m47.676s
```

A tabela a seguir lista a análise de sentimento em nível de frase e teleconferência de resultados para as 10 maiores empresas da NASDAQ de 2016 a 2020.

Contagens de sentimentos e porcentagem	**Todas as 10 empresas**	**AAPL**	**AMD**	**AMZN**	**CSCO**	**GOOGL**	**INTC**	**MSFT**	**NVDA**
Contagens positivas	7447	1567	743	290	682	826	824	904	417
Contagens neutras	64067	6856	7596	5086	6650	5914	6099	5715	6189
Contagens negativas	1787	253	213	84	189	97	282	202	89
Contagens não categorizadas	196	0	0	76	0	0	0	1	0
(contagens totais)	73497	8676	8552	5536	7521	6837	7205	6822	6695

Em termos de porcentagem, a maioria das frases ditas pelos CEOs e CFOs são factuais e, portanto, carregam sentimentos neutros. Durante uma teleconferência de resultados, os analistas fazem perguntas que podem transmitir sentimentos positivos ou negativos. Vale a pena investigar mais quantitativamente como o sentimento negativo ou positivo afeta os preços das ações no mesmo dia ou no dia seguinte de negociação.

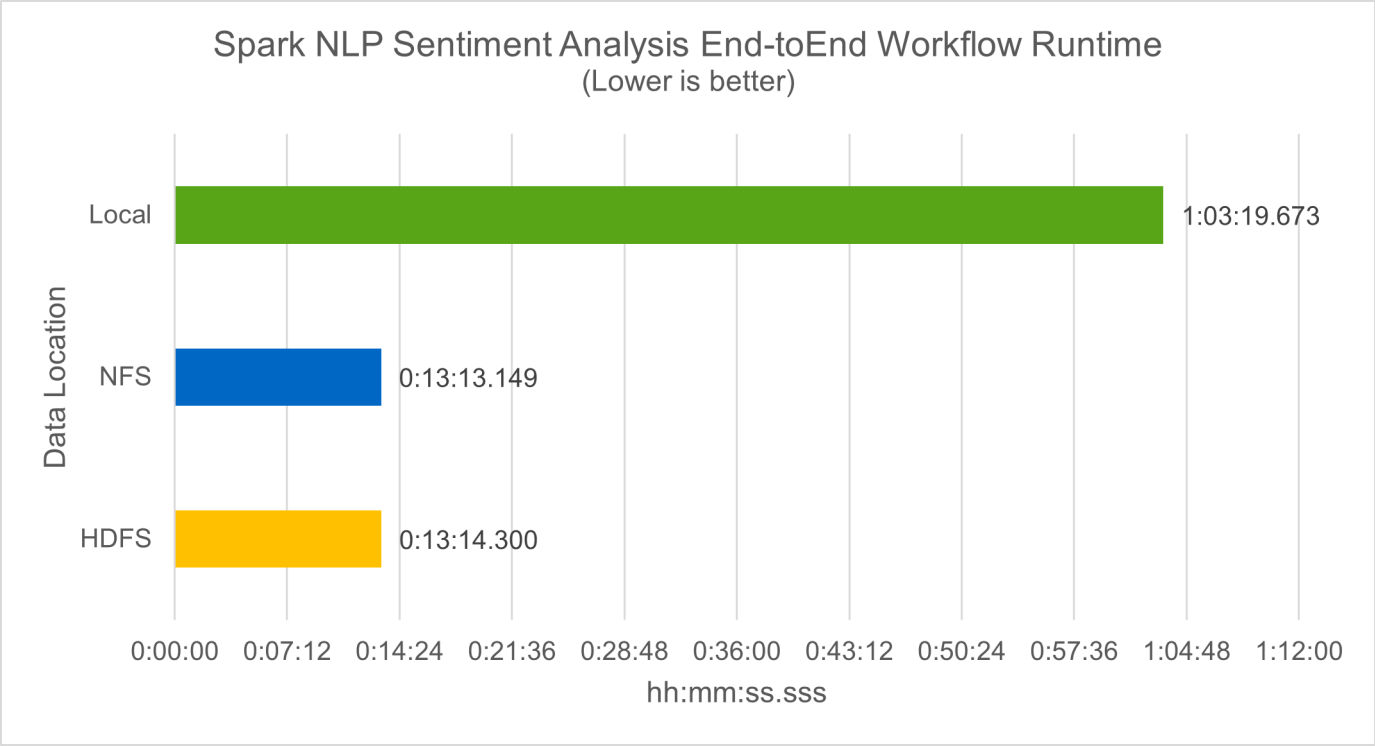
A tabela a seguir lista a análise de sentimento em nível de frase para as 10 maiores empresas do NASDAQ, expressa em porcentagem.

Porcentagem de sentimento	Todas as 10 empresas	AAPL	AMD	AMZN	CSCO	GOOGL	INTC	MSFT	NVDA
Positivo	10,13%	18,06%	8,69%	5,24%	9,07%	12,08%	11,44%	13,25%	6,23%
Neutro	87,17%	79,02%	88,82%	91,87%	88,42%	86,50%	84,65%	83,77%	92,44%
Negativo	2,43%	2,92%	2,49%	1,52%	2,51%	1,42%	3,91%	2,96%	1,33%
Sem categoria	0,27%	0%	0%	1,37%	0%	0%	0%	0,01%	0%

Em termos de tempo de execução do fluxo de trabalho, vimos uma melhoria significativa de 4,78x `local` modo para um ambiente distribuído no HDFS e uma melhoria adicional de 0,14% ao aproveitar o NFS.

```
-bash-4.2$ time ~/anaconda3/bin/spark-submit
--packages com.johnsnowlabs.nlp:spark-nlp_2.12:3.4.3
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
--conf spark.driver.extraJavaOptions="-Xss10m -XX:MaxPermSize=1024M"
--conf spark.executor.extraJavaOptions="-Xss10m -XX:MaxPermSize=512M"
/sparkusecase/tr-4570-nlp/sentiment_analysis_spark.py
file:///sparkdemo/sparknlp/Transcripts/
> ./sentiment_analysis_nfs.log 2>&1
real13m13.149s
user537m50.148s
sys4m46.173s
```

Como mostra a figura a seguir, o paralelismo de dados e modelos melhorou o processamento de dados e a velocidade de inferência do modelo distribuído do TensorFlow. A localização dos dados no NFS produziu um tempo de execução um pouco melhor porque o gargalo do fluxo de trabalho é o download de modelos pré-treinados. Se aumentarmos o tamanho do conjunto de dados de transcrições, a vantagem do NFS fica mais óbvia.



Treinamento distribuído com desempenho Horovod

O comando a seguir produziu informações de tempo de execução e um arquivo de log em nosso cluster Spark usando um único `master` nó com 160 executores, cada um com um núcleo. A memória do executor foi limitada a 5 GB para evitar erros de falta de memória. Veja a seção"Scripts Python para cada caso de uso principal" para mais detalhes sobre o processamento de dados, treinamento do modelo e cálculo de precisão do modelo em `keras_spark_horovod_rossmann_estimator.py`.

```
(base) [root@n138 horovod]# time spark-submit
--master local
--executor-memory 5g
--executor-cores 1
--num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir file:///sparkusecase/horovod
--local-submission-csv /tmp/submission_0.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_local. log 2>&1
```

O tempo de execução resultante com dez épocas de treinamento foi o seguinte:

```
real43m34.608s
user12m22.057s
sys2m30.127s
```

Foram necessários mais de 43 minutos para processar dados de entrada, treinar um modelo DNN, calcular a precisão e produzir pontos de verificação do TensorFlow e um arquivo CSV para resultados de previsão. Limitamos o número de períodos de treinamento a 10, que na prática geralmente é definido como 100 para garantir precisão satisfatória do modelo. O tempo de treinamento normalmente é escalonado linearmente com o número de épocas.

Em seguida, usamos os quatro nós de trabalho disponíveis no cluster e executamos o mesmo script em `yarn` modo com dados em HDFS:

```
(base) [root@n138 horovod]# time spark-submit
--master yarn
--executor-memory 5g
--executor-cores 1 --num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir hdfs:///user/hdfs/tr-4570/experiments/horovod
--local-submission-csv /tmp/submission_1.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_yarn.log 2>&1
```

O tempo de execução resultante foi melhorado da seguinte forma:

```
real8m13.728s
user7m48.421s
sys1m26.063s
```

Com o modelo de Horovod e o paralelismo de dados no Spark, vimos uma aceleração de tempo de execução de 5,29x `yarn` contra `local` modo com dez épocas de treinamento. Isso é mostrado na figura a seguir com as legendas `HDFS` e `Local` . O treinamento do modelo DNN subjacente do TensorFlow pode ser ainda mais acelerado com GPUs, se disponíveis. Planejamos realizar esses testes e publicar os resultados em um futuro relatório técnico.

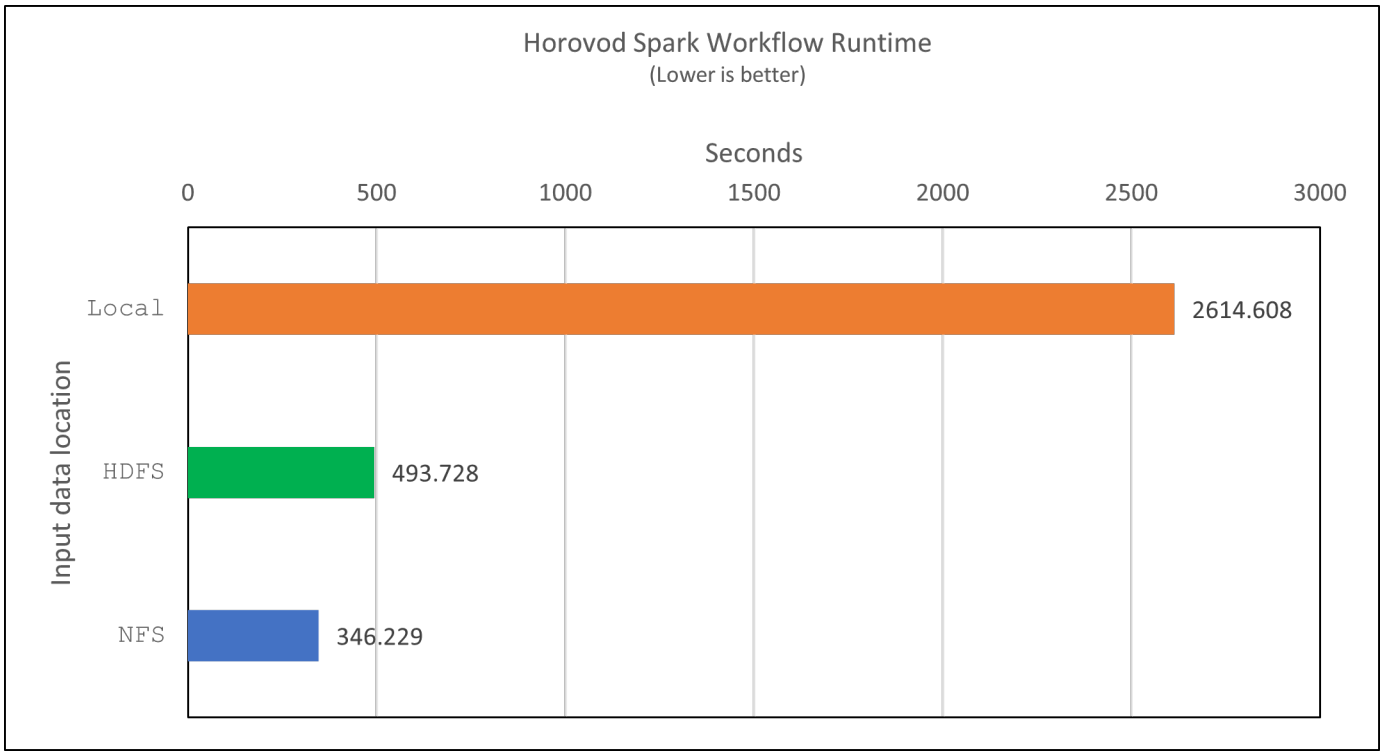
Nosso próximo teste comparou os tempos de execução com dados de entrada residindo em NFS versus HDFS. O volume NFS no AFF A800 foi montado em `/sparkdemo/horovod` nos cinco nós (um mestre, quatro trabalhadores) em nosso cluster Spark. Executamos um comando semelhante aos testes anteriores, com o `--data- dir` parâmetro agora apontando para a montagem NFS:

```
(base) [root@n138 horovod]# time spark-submit
--master yarn
--executor-memory 5g
--executor-cores 1
--num-executors 160
/sparkusecase/horovod/keras_spark_horovod_rossmann_estimator.py
--epochs 10
--data-dir file:///sparkdemo/horovod
--local-submission-csv /tmp/submission_2.csv
--local-checkpoint-file /tmp/checkpoint/
> /tmp/keras_spark_horovod_rossmann_estimator_nfs.log 2>&1
```

O tempo de execução resultante com NFS foi o seguinte:

```
real 5m46.229s
user 5m35.693s
sys  1m5.615s
```

Houve uma aceleração adicional de 1,43x, conforme mostrado na figura a seguir. Portanto, com um armazenamento all-flash da NetApp conectado ao seu cluster, os clientes aproveitam os benefícios da rápida transferência e distribuição de dados para fluxos de trabalho do Horovod Spark, alcançando uma aceleração de 7,55x em comparação à execução em um único nó.



Modelos de aprendizado profundo para desempenho de previsão de CTR

Para sistemas de recomendação projetados para maximizar o CTR, você deve aprender interações de recursos sofisticados por trás dos comportamentos do usuário que podem ser calculados matematicamente da ordem mais baixa para a mais alta. Tanto as interações de recursos de baixa quanto de alta ordem devem ser igualmente importantes para um bom modelo de aprendizado profundo, sem distorção em relação a uma ou outra. Deep Factorization Machine (DeepFM), uma rede neural baseada em máquina de fatoração, combina máquinas de fatoração para recomendação e aprendizado profundo para aprendizado de recursos em uma nova arquitetura de rede neural.

Embora as máquinas de fatoração convencionais modelem interações de recursos em pares como um produto interno de vetores latentes entre recursos e possam, teoricamente, capturar informações de alta ordem, na prática, os profissionais de aprendizado de máquina geralmente usam apenas interações de recursos de segunda ordem devido à alta complexidade de computação e armazenamento. Variantes de redes neurais profundas como as do Google "Modelos largos e profundos" por outro lado, aprende interações de recursos sofisticados em uma estrutura de rede híbrida combinando um modelo linear amplo e um modelo profundo.

Há duas entradas para este Modelo Amplo e Profundo, uma para o modelo amplo subjacente e outra para o profundo, sendo que esta última parte ainda requer engenharia de recursos especializada e, portanto, torna a técnica menos generalizável para outros domínios. Diferentemente do modelo amplo e profundo, o DeepFM pode ser treinado eficientemente com recursos brutos sem qualquer engenharia de recursos, porque sua parte ampla e parte profunda compartilham a mesma entrada e o mesmo vetor de incorporação.

Primeiro processamos o Criteo `train.txt` (11 GB) em um arquivo CSV chamado `ctr_train.csv` armazenado em uma montagem NFS `/sparkdemo/tr-4570-data` usando `run_classification_criteo_spark.py` da seção"Scripts Python para cada caso de uso principal." Dentro deste script, a função `process_input_file` executa vários métodos de string para remover tabulações e inserir `','` como delimitador e `'\n'` como nova linha. Observe que você só precisa processar o original `train.txt` uma vez, para que o bloco de código seja mostrado como comentários.

Para os testes seguintes de diferentes modelos DL, usamos `ctr_train.csv` como arquivo de entrada. Em execuções de testes subsequentes, o arquivo CSV de entrada foi lido em um Spark DataFrame com esquema contendo um campo de `'label'` , recursos densos inteiros `['I1', 'I2', 'I3', …, 'I13']` , e recursos esparsos `['C1', 'C2', 'C3', …, 'C26']` . A seguir `spark-submit` O comando recebe um CSV de entrada, treina modelos DeepFM com divisão de 20% para validação cruzada e escolhe o melhor modelo após dez períodos de treinamento para calcular a precisão da previsão no conjunto de teste:

```
(base) [root@n138 ~]# time spark-submit --master yarn --executor-memory 5g
--executor-cores 1 --num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py --data
-dir file:///sparkdemo/tr-4570-data >
/tmp/run_classification_criteo_spark_local.log 2>&1
```

Observe que, como o arquivo de dados `ctr_train.csv` for maior que 11 GB, você deve definir um espaço suficiente `spark.driver.maxResultSize` maior que o tamanho do conjunto de dados para evitar erros.

```
 spark = SparkSession.builder \
    .master("yarn") \
    .appName("deep_ctr_classification") \
    .config("spark.jars.packages", "io.github.ravwojdyla:spark-schema-
utils_2.12:0.1.0") \
    .config("spark.executor.cores", "1") \
    .config('spark.executor.memory', '5gb') \
    .config('spark.executor.memoryOverhead', '1500') \
    .config('spark.driver.memoryOverhead', '1500') \
    .config("spark.sql.shuffle.partitions", "480") \
    .config("spark.sql.execution.arrow.enabled", "true") \
    .config("spark.driver.maxResultSize", "50gb") \
    .getOrCreate()
```

No acima `SparkSession.builder` configuração também habilitamos "Flecha Apache" , que converte um Spark DataFrame em um Pandas DataFrame com o `df.toPandas()` método.

```
22/06/17 15:56:21 INFO scheduler.DAGScheduler: Job 2 finished: toPandas at
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py:96, took
627.126487 s
Obtained Spark DF and transformed to Pandas DF using Arrow.
```

Após a divisão aleatória, há mais de 36 milhões de linhas no conjunto de dados de treinamento e 9 milhões de amostras no conjunto de teste:

```
Training dataset size =  36672493
Testing dataset size =  9168124
```

Como este relatório técnico se concentra em testes de CPU sem usar GPUs, é fundamental que você crie o TensorFlow com sinalizadores de compilador apropriados. Esta etapa evita invocar quaisquer bibliotecas aceleradas por GPU e aproveita ao máximo as instruções Advanced Vector Extensions (AVX) e AVX2 do TensorFlow. Esses recursos são projetados para cálculos algébricos lineares, como adição vetorizada, multiplicação de matrizes dentro de um treinamento DNN de propagação direta ou retropropagação. A instrução Fused Multiply Add (FMA) disponível com AVX2 usando registradores de ponto flutuante (FP) de 256 bits é ideal para códigos inteiros e tipos de dados, resultando em um aumento de velocidade de até 2x. Para códigos FP e tipos de dados, o AVX2 atinge 8% de aceleração em relação ao AVX.

```
2022-06-18 07:19:20.101478: I
tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary
is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the
following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the
appropriate compiler flags.
```

Para construir o TensorFlow a partir do código-fonte, a NetApp recomenda usar "Bazel" . Para nosso ambiente, executamos os seguintes comandos no prompt do shell para instalar `dnf` , `dnf-plugins` , e Bazel.

```
yum install dnf
dnf install 'dnf-command(copr)'
dnf copr enable vbatts/bazel
dnf install bazel5
```

Você deve habilitar o GCC 5 ou mais recente para usar os recursos do C++17 durante o processo de compilação, que é fornecido pelo RHEL com a Software Collections Library (SCL). Os seguintes comandos instalam `devtoolset` e GCC 11.2.1 em nosso cluster RHEL 7.9:

```
subscription-manager repos --enable rhel-server-rhscl-7-rpms
yum install devtoolset-11-toolchain
yum install devtoolset-11-gcc-c++
yum update
scl enable devtoolset-11 bash
. /opt/rh/devtoolset-11/enable
```

Observe que os dois últimos comandos habilitam `devtoolset-11` , que usa `/opt/rh/devtoolset-11/root/usr/bin/gcc` (GCC 11.2.1). Além disso, certifique-se de que seu `git` a versão é maior que 1.8.3 (vem com o RHEL 7.9). Consulte isto "artigo" para atualização `git` para 2.24.1.

Presumimos que você já clonou o repositório mestre mais recente do TensorFlow. Em seguida, crie um `workspace` diretório com um `WORKSPACE` arquivo para compilar o TensorFlow a partir do código-fonte com AVX, AVX2 e FMA. Execute o `configure` arquivo e especifique o local binário correto do Python. "CUDA" está desabilitado para nossos testes porque não usamos uma GPU. UM `.bazelrc` o arquivo é gerado de acordo com suas configurações. Além disso, editamos o arquivo e configuramos `build --define=no_hdfs_support=false` para habilitar o suporte HDFS. Consulte `.bazelrc` na seção"Scripts Python para cada caso de uso principal," para uma lista completa de configurações e sinalizadores.

```
./configure
bazel build -c opt --copt=-mavx --copt=-mavx2 --copt=-mfma --copt=
-mfpmath=both -k //tensorflow/tools/pip_package:build_pip_package
```

Depois de criar o TensorFlow com os sinalizadores corretos, execute o script a seguir para processar o conjunto de dados Criteo Display Ads, treinar um modelo DeepFM e calcular a Área sob a Curva Característica Operacional do Receptor (ROC AUC) a partir das pontuações de previsão.

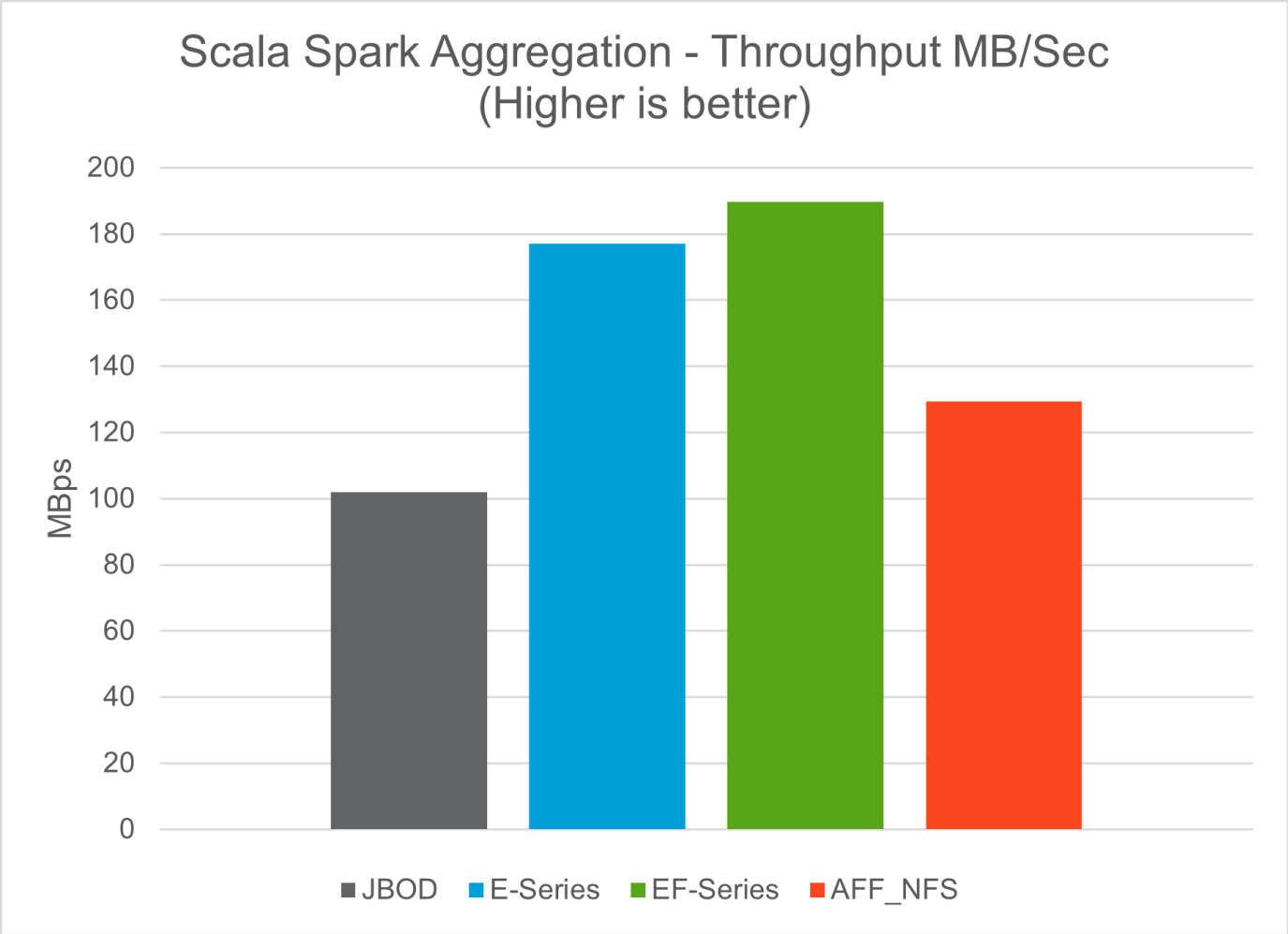
```
(base) [root@n138 examples]# ~/anaconda3/bin/spark-submit
--master yarn
--executor-memory 15g
--executor-cores 1
--num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py
--data-dir file:///sparkdemo/tr-4570-data
> . /run_classification_criteo_spark_nfs.log 2>&1
```

Após dez períodos de treinamento, obtivemos a pontuação AUC no conjunto de dados de teste:

```
Epoch 1/10
125/125 - 7s - loss: 0.4976 - binary_crossentropy: 0.4974 - val_loss:
0.4629 - val_binary_crossentropy: 0.4624
Epoch 2/10
125/125 - 1s - loss: 0.3281 - binary_crossentropy: 0.3271 - val_loss:
0.5146 - val_binary_crossentropy: 0.5130
Epoch 3/10
125/125 - 1s - loss: 0.1948 - binary_crossentropy: 0.1928 - val_loss:
0.6166 - val_binary_crossentropy: 0.6144
Epoch 4/10
125/125 - 1s - loss: 0.1408 - binary_crossentropy: 0.1383 - val_loss:
0.7261 - val_binary_crossentropy: 0.7235
Epoch 5/10
125/125 - 1s - loss: 0.1129 - binary_crossentropy: 0.1102 - val_loss:
0.7961 - val_binary_crossentropy: 0.7934
Epoch 6/10
125/125 - 1s - loss: 0.0949 - binary_crossentropy: 0.0921 - val_loss:
0.9502 - val_binary_crossentropy: 0.9474
Epoch 7/10
125/125 - 1s - loss: 0.0778 - binary_crossentropy: 0.0750 - val_loss:
1.1329 - val_binary_crossentropy: 1.1301
Epoch 8/10
125/125 - 1s - loss: 0.0651 - binary_crossentropy: 0.0622 - val_loss:
1.3794 - val_binary_crossentropy: 1.3766
Epoch 9/10
125/125 - 1s - loss: 0.0555 - binary_crossentropy: 0.0527 - val_loss:
1.6115 - val_binary_crossentropy: 1.6087
Epoch 10/10
125/125 - 1s - loss: 0.0470 - binary_crossentropy: 0.0442 - val_loss:
1.6768 - val_binary_crossentropy: 1.6740
test AUC 0.6337
```

De maneira semelhante aos casos de uso anteriores, comparamos o tempo de execução do fluxo de trabalho

do Spark com dados residentes em locais diferentes. A figura a seguir mostra uma comparação da previsão de CTR de aprendizado profundo para um tempo de execução de fluxos de trabalho do Spark.



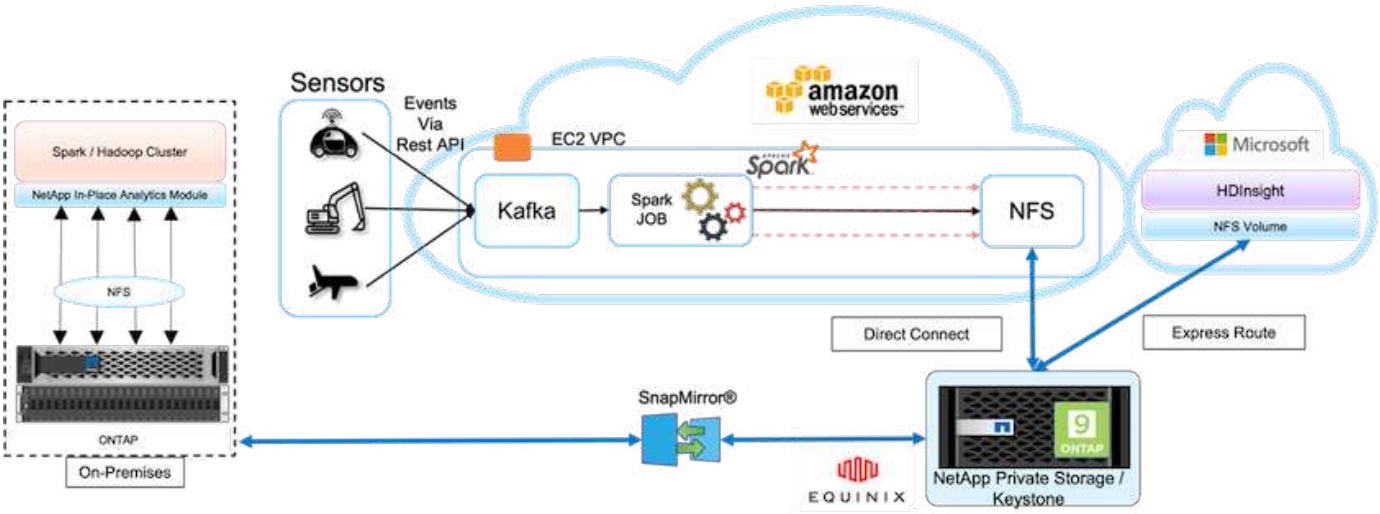
Solução de nuvem híbrida

Um data center empresarial moderno é uma nuvem híbrida que conecta vários ambientes de infraestrutura distribuída por meio de um plano de gerenciamento de dados contínuo com um modelo operacional consistente, no local e/ou em várias nuvens públicas. Para aproveitar ao máximo uma nuvem híbrida, você precisa ser capaz de mover dados facilmente entre seus ambientes locais e de várias nuvens, sem a necessidade de conversões de dados ou refatoração de aplicativos.

Os clientes indicaram que iniciam sua jornada para a nuvem híbrida movendo o armazenamento secundário para a nuvem para casos de uso como proteção de dados ou movendo cargas de trabalho menos críticas aos negócios, como desenvolvimento de aplicativos e DevOps para a nuvem. Eles então passam para cargas de trabalho mais críticas. Hospedagem de conteúdo e web, DevOps e desenvolvimento de aplicativos, bancos de dados, análises e aplicativos em contêineres estão entre as cargas de trabalho de nuvem híbrida mais populares. A complexidade, o custo e os riscos dos projetos de IA empresarial historicamente dificultaram a adoção da IA do estágio experimental até a produção.

Com uma solução de nuvem híbrida da NetApp , os clientes se beneficiam de ferramentas integradas de segurança, governança de dados e conformidade com um único painel de controle para gerenciamento de

dados e fluxo de trabalho em ambientes distribuídos, ao mesmo tempo em que otimizam o custo total de propriedade com base em seu consumo. A figura a seguir é um exemplo de solução de um parceiro de serviço de nuvem encarregado de fornecer conectividade multinuvem para dados de análise de big data dos clientes.



Neste cenário, os dados de IoT recebidos na AWS de diferentes fontes são armazenados em um local central no NetApp Private Storage (NPS). O armazenamento NPS é conectado a clusters Spark ou Hadoop localizados na AWS e no Azure, permitindo que aplicativos de análise de big data sejam executados em várias nuvens acessando os mesmos dados. Os principais requisitos e desafios para este caso de uso incluem o seguinte:

- Os clientes desejam executar tarefas analíticas nos mesmos dados usando várias nuvens.
- Os dados devem ser recebidos de diferentes fontes, como ambientes locais e de nuvem, por meio de diferentes sensores e hubs.
- A solução deve ser eficiente e econômica.
- O principal desafio é criar uma solução econômica e eficiente que ofereça serviços de análise híbridos entre diferentes ambientes locais e na nuvem.

Nossa solução de proteção de dados e conectividade multinuvem resolve o problema de ter aplicativos de análise de nuvem em vários hiperescaladores. Conforme mostrado na figura acima, os dados dos sensores são transmitidos e ingeridos no cluster do AWS Spark por meio do Kafka. Os dados são armazenados em um compartilhamento NFS que reside no NPS, que está localizado fora do provedor de nuvem, dentro de um data center da Equinix.

Como o NetApp NPS está conectado ao Amazon AWS e ao Microsoft Azure por meio de conexões Direct Connect e Express Route, respectivamente, os clientes podem aproveitar o In-Place Analytics Module para acessar os dados dos clusters de análise da Amazon e da AWS. Consequentemente, como tanto o armazenamento local quanto o NPS executam software ONTAP , "SnapMirror" pode espelhar os dados do NPS no cluster local, fornecendo análises de nuvem híbrida no local e em várias nuvens.

Para obter o melhor desempenho, a NetApp normalmente recomenda o uso de várias interfaces de rede e conexão direta ou rotas expressas para acessar os dados de instâncias de nuvem. Temos outras soluções de movimentação de dados, incluindo "XCP" e "BlueXP Copiar e Sincronizar" para ajudar os clientes a criar clusters Spark de nuvem híbrida, seguros, econômicos e com reconhecimento de aplicativos.

Scripts Python para cada caso de uso principal

Os três scripts Python a seguir correspondem aos três principais casos de uso testados. O primeiro é `sentiment_analysis_sparknlp.py` .

```
# TR-4570 Refresh NLP testing by Rick Huang
from sys import argv
import os
import sparknlp
import pyspark.sql.functions as F
from sparknlp import Finisher
from pyspark.ml import Pipeline
from sparknlp.base import *
from sparknlp.annotator import *
from sparknlp.pretrained import PretrainedPipeline
from sparknlp import Finisher
# Start Spark Session with Spark NLP
spark = sparknlp.start()
print("Spark NLP version:")
print(sparknlp.version())
print("Apache Spark version:")
print(spark.version)
spark = sparknlp.SparkSession.builder \
     .master("yarn") \
     .appName("test_hdfs_read_write") \
     .config("spark.executor.cores", "1") \
     .config("spark.jars.packages", "com.johnsnowlabs.nlp:spark-
nlp_2.12:3.4.3")\
     .config('spark.executor.memory', '5gb') \
     .config('spark.executor.memoryOverhead','1000')\
     .config('spark.driver.memoryOverhead','1000')\
     .config("spark.sql.shuffle.partitions", "480")\
     .getOrCreate()
sc = spark.sparkContext
from pyspark.sql import SQLContext
sql = SQLContext(sc)
sqlContext = SQLContext(sc)
# Download pre-trained pipelines & sequence classifier
explain_pipeline_model = PretrainedPipeline('explain_document_dl',
lang='en').model#pipeline_sa =
PretrainedPipeline("classifierdl_bertwiki_finance_sentiment_pipeline",
lang="en")
# pipeline_finbert =
BertForSequenceClassification.loadSavedModel('/sparkusecase/bert_sequence_
classifier_finbert_en_3', spark)
```

```
sequenceClassifier = BertForSequenceClassification \
        .pretrained('bert_sequence_classifier_finbert', 'en') \
        .setInputCols(['token', 'document']) \
        .setOutputCol('class') \
        .setCaseSensitive(True) \
        .setMaxSentenceLength(512)
def process_sentence_df(data):
    # Pre-process: begin
    print("1. Begin DataFrame pre-processing...\n")
    print(f"\n\t2. Attaching DocumentAssembler Transformer to the
pipeline")
    documentAssembler = DocumentAssembler() \
        .setInputCol("text") \
        .setOutputCol("document") \
        .setCleanupMode("inplace_full")
        #.setCleanupMode("shrink", "inplace_full")
    doc_df = documentAssembler.transform(data)
    doc_df.printSchema()
    doc_df.show(truncate=50)
    # Pre-process: get rid of  blank lines
    clean_df = doc_df.withColumn("tmp", F.explode("document")) \
        .select("tmp.result").where("tmp.end !=
-1").withColumnRenamed("result", "text").dropna()
    print("[OK!] DataFrame after initial cleanup:\n")
    clean_df.printSchema()
    clean_df.show(truncate=80)
    # for FinBERT
    tokenizer = Tokenizer() \
        .setInputCols(['document']) \
        .setOutputCol('token')
    print(f"\n\t3. Attaching Tokenizer Annotator to the pipeline")
    pipeline_finbert = Pipeline(stages=[
        documentAssembler,
        tokenizer,
        sequenceClassifier
        ])
    # Use Finisher() & construct PySpark ML pipeline
    finisher = Finisher().setInputCols(["token", "lemma", "pos",
"entities"])
    print(f"\n\t4. Attaching Finisher Transformer to the pipeline")
    pipeline_ex = Pipeline() \
        .setStages([
           explain_pipeline_model,
           finisher
           ])
    print("\n\t\t\t ---- Pipeline Built Successfully ----")
```

```
    # Loading pipelines to annotate
    #result_ex_df = pipeline_ex.transform(clean_df)
    ex_model = pipeline_ex.fit(clean_df)
    annotations_finished_ex_df = ex_model.transform(clean_df)
    # result_sa_df = pipeline_sa.transform(clean_df)
    result_finbert_df = pipeline_finbert.fit(clean_df).transform(clean_df)
    print("\n\t\t\t ----Document Explain, Sentiment Analysis & FinBERT
Pipeline Fitted Successfully ----")
    # Check the result entities
    print("[OK!] Simple explain ML pipeline result:\n")
    annotations_finished_ex_df.printSchema()
    annotations_finished_ex_df.select('text',
'finished_entities').show(truncate=False)
    # Check the result sentiment from FinBERT
    print("[OK!] Sentiment Analysis FinBERT pipeline result:\n")
    result_finbert_df.printSchema()
    result_finbert_df.select('text', 'class.result').show(80, False)
    sentiment_stats(result_finbert_df)
    return
def sentiment_stats(finbert_df):
    result_df = finbert_df.select('text', 'class.result')
    sa_df = result_df.select('result')
    sa_df.groupBy('result').count().show()
    # total_lines = result_clean_df.count()
    # num_neutral = result_clean_df.where(result_clean_df.result ==
['neutral']).count()
    # num_positive = result_clean_df.where(result_clean_df.result ==
['positive']).count()
    # num_negative = result_clean_df.where(result_clean_df.result ==
['negative']).count()
    # print(f"\nRatio of neutral sentiment = {num_neutral/total_lines}")
    # print(f"Ratio of positive sentiment = {num_positive / total_lines}")
    # print(f"Ratio of negative sentiment = {num_negative /
total_lines}\n")
    return
def process_input_file(file_name):
    # Turn input file to Spark DataFrame
    print("START processing input file...")
    data_df = spark.read.text(file_name)
    data_df.show()
    # rename first column 'text' for sparknlp
    output_df = data_df.withColumnRenamed("value", "text").dropna()
    output_df.printSchema()
    return output_dfdef process_local_dir(directory):
    filelist = []
    for subdir, dirs, files in os.walk(directory):
```

```
        for filename in files:
            filepath = subdir + os.sep + filename
            print("[OK!] Will process the following files:")
            if filepath.endswith(".txt"):
                print(filepath)
                filelist.append(filepath)
    return filelist
def process_local_dir_or_file(dir_or_file):
    numfiles = 0
    if os.path.isfile(dir_or_file):
        input_df = process_input_file(dir_or_file)
        print("Obtained input_df.")
        process_sentence_df(input_df)
        print("Processed input_df")
        numfiles += 1
    else:
        filelist = process_local_dir(dir_or_file)
        for file in filelist:
            input_df = process_input_file(file)
            process_sentence_df(input_df)
            numfiles += 1
    return numfiles
def process_hdfs_dir(dir_name):
    # Turn input files to Spark DataFrame
    print("START processing input HDFS directory...")
    data_df = spark.read.option("recursiveFileLookup",
"true").text(dir_name)
    data_df.show()
    print("[DEBUG] total lines in data_df = ", data_df.count())
    # rename first column 'text' for sparknlp
    output_df = data_df.withColumnRenamed("value", "text").dropna()
    print("[DEBUG] output_df looks like: \n")
    output_df.show(40, False)
    print("[DEBUG] HDFS dir resulting data_df schema: \n")
    output_df.printSchema()
    process_sentence_df(output_df)
    print("Processed HDFS directory: ", dir_name)
    returnif __name__ == '__main__':
    try:
        if len(argv) == 2:
            print("Start processing input...\n")
    except:
        print("[ERROR] Please enter input text file or path to
process!\n")
        exit(1)
    # This is for local file, not hdfs:
```

```
    numfiles = process_local_dir_or_file(str(argv[1]))
    # For HDFS single file & directory:
    input_df = process_input_file(str(argv[1]))
    print("Obtained input_df.")
    process_sentence_df(input_df)
    print("Processed input_df")
    numfiles += 1
    # For HDFS directory of subdirectories of files:
    input_parse_list = str(argv[1]).split('/')
    print(input_parse_list)
    if input_parse_list[-2:-1] == ['Transcripts']:
        print("Start processing HDFS directory: ", str(argv[1]))
        process_hdfs_dir(str(argv[1]))
    print(f"[OK!] All done. Number of files processed = {numfiles}")
```

O segundo script é `keras_spark_horovod_rossmann_estimator.py`.

```
# Copyright 2022 NetApp, Inc.
# Authored by Rick Huang
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
==========================================================================
====
# The below code was modified from: https://www.kaggle.com/c/rossmann-
store-sales
import argparse
import datetime
import os
import sys
from distutils.version import LooseVersion
import pyspark.sql.types as T
import pyspark.sql.functions as F
from pyspark import SparkConf, Row
from pyspark.sql import SparkSession
```

```
import tensorflow as tf
import tensorflow.keras.backend as K
from tensorflow.keras.layers import Input, Embedding, Concatenate, Dense,
Flatten, Reshape, BatchNormalization, Dropout
import horovod.spark.keras as hvd
from horovod.spark.common.backend import SparkBackend
from horovod.spark.common.store import Store
from horovod.tensorflow.keras.callbacks import BestModelCheckpoint
parser = argparse.ArgumentParser(description='Horovod Keras Spark Rossmann
Estimator Example',

formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--master',
                    help='spark cluster to use for training. If set to
None, uses current default cluster. Cluster'
                         'should be set up to provide a Spark task per
multiple CPU cores, or per GPU, e.g. by'
                         'supplying `-c <NUM_GPUS>` in Spark Standalone
mode')
parser.add_argument('--num-proc', type=int,
                    help='number of worker processes for training,
default: `spark.default.parallelism`')
parser.add_argument('--learning_rate', type=float, default=0.0001,
                    help='initial learning rate')
parser.add_argument('--batch-size', type=int, default=100,
                    help='batch size')
parser.add_argument('--epochs', type=int, default=100,
                    help='number of epochs to train')
parser.add_argument('--sample-rate', type=float,
                    help='desired sampling rate. Useful to set to low
number (e.g. 0.01) to make sure that '
                         'end-to-end process works')
parser.add_argument('--data-dir', default='file://' + os.getcwd(),
                    help='location of data on local filesystem (prefixed
with file://) or on HDFS')
parser.add_argument('--local-submission-csv', default='submission.csv',
                    help='output submission predictions CSV')
parser.add_argument('--local-checkpoint-file', default='checkpoint',
                    help='model checkpoint')
parser.add_argument('--work-dir', default='/tmp',
                    help='temporary working directory to write
intermediate files (prefix with hdfs:// to use HDFS)')
if __name__ == '__main__':
    args = parser.parse_args()
    # =============== #
    # DATA PREPARATION #
```

```
    # ================ #
    print('================')
    print('Data preparation')
    print('================')
    # Create Spark session for data preparation.
    conf = SparkConf() \
        .setAppName('Keras Spark Rossmann Estimator Example') \
        .set('spark.sql.shuffle.partitions', '480') \
        .set("spark.executor.cores", "1") \
        .set('spark.executor.memory', '5gb') \
        .set('spark.executor.memoryOverhead','1000')\
        .set('spark.driver.memoryOverhead','1000')
    if args.master:
        conf.setMaster(args.master)
    elif args.num_proc:
        conf.setMaster('local[{}]'.format(args.num_proc))
    spark = SparkSession.builder.config(conf=conf).getOrCreate()
    train_csv = spark.read.csv('%s/train.csv' % args.data_dir,
header=True)
    test_csv = spark.read.csv('%s/test.csv' % args.data_dir, header=True)
    store_csv = spark.read.csv('%s/store.csv' % args.data_dir,
header=True)
    store_states_csv = spark.read.csv('%s/store_states.csv' %
args.data_dir, header=True)
    state_names_csv = spark.read.csv('%s/state_names.csv' % args.data_dir,
header=True)
    google_trend_csv = spark.read.csv('%s/googletrend.csv' %
args.data_dir, header=True)
    weather_csv = spark.read.csv('%s/weather.csv' % args.data_dir,
header=True)
    def expand_date(df):
        df = df.withColumn('Date', df.Date.cast(T.DateType()))
        return df \
            .withColumn('Year', F.year(df.Date)) \
            .withColumn('Month', F.month(df.Date)) \
            .withColumn('Week', F.weekofyear(df.Date)) \
            .withColumn('Day', F.dayofmonth(df.Date))
    def prepare_google_trend():
        # Extract week start date and state.
        google_trend_all = google_trend_csv \
            .withColumn('Date', F.regexp_extract(google_trend_csv.week,
'(.*?) -', 1)) \
            .withColumn('State', F.regexp_extract(google_trend_csv.file,
'Rossmann_DE_(.*)', 1))
        # Map state NI -> HB,NI to align with other data sources.
        google_trend_all = google_trend_all \
```

```
            .withColumn('State', F.when(google_trend_all.State == 'NI',
'HB,NI').otherwise(google_trend_all.State))
        # Expand dates.
        return expand_date(google_trend_all)
    def add_elapsed(df, cols):
        def add_elapsed_column(col, asc):
            def fn(rows):
                last_store, last_date = None, None
                for r in rows:
                    if last_store != r.Store:
                        last_store = r.Store
                        last_date = r.Date
                    if r[col]:
                        last_date = r.Date
                    fields = r.asDict().copy()
                    fields[('After' if asc else 'Before') + col] = (r.Date
- last_date).days
                    yield Row(**fields)
            return fn
        df = df.repartition(df.Store)
        for asc in [False, True]:
            sort_col = df.Date.asc() if asc else df.Date.desc()
            rdd = df.sortWithinPartitions(df.Store.asc(), sort_col).rdd
            for col in cols:
                rdd = rdd.mapPartitions(add_elapsed_column(col, asc))
            df = rdd.toDF()
        return df
    def prepare_df(df):
        num_rows = df.count()
        # Expand dates.
        df = expand_date(df)
        df = df \
            .withColumn('Open', df.Open != '0') \
            .withColumn('Promo', df.Promo != '0') \
            .withColumn('StateHoliday', df.StateHoliday != '0') \
            .withColumn('SchoolHoliday', df.SchoolHoliday != '0')
        # Merge in store information.
        store = store_csv.join(store_states_csv, 'Store')
        df = df.join(store, 'Store')
        # Merge in Google Trend information.
        google_trend_all = prepare_google_trend()
        df = df.join(google_trend_all, ['State', 'Year',
'Week']).select(df['*'], google_trend_all.trend)
        # Merge in Google Trend for whole Germany.
        google_trend_de = google_trend_all[google_trend_all.file ==
'Rossmann_DE'].withColumnRenamed('trend', 'trend_de')
```

```
        df = df.join(google_trend_de, ['Year', 'Week']).select(df['*'],
google_trend_de.trend_de)
        # Merge in weather.
        weather = weather_csv.join(state_names_csv, weather_csv.file ==
state_names_csv.StateName)
        df = df.join(weather, ['State', 'Date'])
        # Fix null values.
        df = df \
            .withColumn('CompetitionOpenSinceYear',
F.coalesce(df.CompetitionOpenSinceYear, F.lit(1900))) \
            .withColumn('CompetitionOpenSinceMonth',
F.coalesce(df.CompetitionOpenSinceMonth, F.lit(1))) \
            .withColumn('Promo2SinceYear', F.coalesce(df.Promo2SinceYear,
F.lit(1900))) \
            .withColumn('Promo2SinceWeek', F.coalesce(df.Promo2SinceWeek,
F.lit(1)))
        # Days & months competition was open, cap to 2 years.
        df = df.withColumn('CompetitionOpenSince',
                           F.to_date(F.format_string('%s-%s-15',
df.CompetitionOpenSinceYear,

df.CompetitionOpenSinceMonth)))
        df = df.withColumn('CompetitionDaysOpen',
                           F.when(df.CompetitionOpenSinceYear > 1900,
                                  F.greatest(F.lit(0), F.least(F.lit(360 *
2), F.datediff(df.Date, df.CompetitionOpenSince))))
                           .otherwise(0))
        df = df.withColumn('CompetitionMonthsOpen',
(df.CompetitionDaysOpen / 30).cast(T.IntegerType()))
        # Days & weeks of promotion, cap to 25 weeks.
        df = df.withColumn('Promo2Since',
                           F.expr('date_add(format_string("%s-01-01",
Promo2SinceYear), (cast(Promo2SinceWeek as int) - 1) * 7)'))
        df = df.withColumn('Promo2Days',
                           F.when(df.Promo2SinceYear > 1900,
                                  F.greatest(F.lit(0), F.least(F.lit(25 *
7), F.datediff(df.Date, df.Promo2Since))))
                           .otherwise(0))
        df = df.withColumn('Promo2Weeks', (df.Promo2Days /
7).cast(T.IntegerType()))
        # Check that we did not lose any rows through inner joins.
        assert num_rows == df.count(), 'lost rows in joins'
        return df
    def build_vocabulary(df, cols):
        vocab = {}
        for col in cols:
```

```
            values = [r[0] for r in df.select(col).distinct().collect()]
            col_type = type([x for x in values if x is not None][0])
            default_value = col_type()
            vocab[col] = sorted(values, key=lambda x: x or default_value)
        return vocab
    def cast_columns(df, cols):
        for col in cols:
            df = df.withColumn(col,
F.coalesce(df[col].cast(T.FloatType()), F.lit(0.0)))
        return df
    def lookup_columns(df, vocab):
        def lookup(mapping):
            def fn(v):
                return mapping.index(v)
            return F.udf(fn, returnType=T.IntegerType())
        for col, mapping in vocab.items():
            df = df.withColumn(col, lookup(mapping)(df[col]))
        return df
    if args.sample_rate:
        train_csv = train_csv.sample(withReplacement=False,
fraction=args.sample_rate)
        test_csv = test_csv.sample(withReplacement=False,
fraction=args.sample_rate)
    # Prepare data frames from CSV files.
    train_df = prepare_df(train_csv).cache()
    test_df = prepare_df(test_csv).cache()
    # Add elapsed times from holidays & promos, the data spanning training
& test datasets.
    elapsed_cols = ['Promo', 'StateHoliday', 'SchoolHoliday']
    elapsed = add_elapsed(train_df.select('Date', 'Store', *elapsed_cols)
                          .unionAll(test_df.select('Date', 'Store',
*elapsed_cols)),
                          elapsed_cols)
    # Join with elapsed times.
    train_df = train_df \
        .join(elapsed, ['Date', 'Store']) \
        .select(train_df['*'], *[prefix + col for prefix in ['Before',
'After'] for col in elapsed_cols])
    test_df = test_df \
        .join(elapsed, ['Date', 'Store']) \
        .select(test_df['*'], *[prefix + col for prefix in ['Before',
'After'] for col in elapsed_cols])
    # Filter out zero sales.
    train_df = train_df.filter(train_df.Sales > 0)
    print('==================')
    print('Prepared data frame')
```

```
    print('==================')
    train_df.show()
    categorical_cols = [
        'Store', 'State', 'DayOfWeek', 'Year', 'Month', 'Day', 'Week',
'CompetitionMonthsOpen', 'Promo2Weeks', 'StoreType',
        'Assortment', 'PromoInterval', 'CompetitionOpenSinceYear',
'Promo2SinceYear', 'Events', 'Promo',
        'StateHoliday', 'SchoolHoliday'
    ]
    continuous_cols = [
        'CompetitionDistance', 'Max_TemperatureC', 'Mean_TemperatureC',
'Min_TemperatureC', 'Max_Humidity',
        'Mean_Humidity', 'Min_Humidity', 'Max_Wind_SpeedKm_h',
'Mean_Wind_SpeedKm_h', 'CloudCover', 'trend', 'trend_de',
        'BeforePromo', 'AfterPromo', 'AfterStateHoliday',
'BeforeStateHoliday', 'BeforeSchoolHoliday', 'AfterSchoolHoliday'
    ]
    all_cols = categorical_cols + continuous_cols
    # Select features.
    train_df = train_df.select(*(all_cols + ['Sales', 'Date'])).cache()
    test_df = test_df.select(*(all_cols + ['Id', 'Date'])).cache()
    # Build vocabulary of categorical columns.
    vocab = build_vocabulary(train_df.select(*categorical_cols)

.unionAll(test_df.select(*categorical_cols)).cache(),
                             categorical_cols)
    # Cast continuous columns to float & lookup categorical columns.
    train_df = cast_columns(train_df, continuous_cols + ['Sales'])
    train_df = lookup_columns(train_df, vocab)
    test_df = cast_columns(test_df, continuous_cols)
    test_df = lookup_columns(test_df, vocab)
    # Split into training & validation.
    # Test set is in 2015, use the same period in 2014 from the training
set as a validation set.
    test_min_date = test_df.agg(F.min(test_df.Date)).collect()[0][0]
    test_max_date = test_df.agg(F.max(test_df.Date)).collect()[0][0]
    one_year = datetime.timedelta(365)
    train_df = train_df.withColumn('Validation',
                                   (train_df.Date > test_min_date -
one_year) & (train_df.Date <= test_max_date - one_year))
    # Determine max Sales number.
    max_sales = train_df.agg(F.max(train_df.Sales)).collect()[0][0]
    # Convert Sales to log domain
    train_df = train_df.withColumn('Sales', F.log(train_df.Sales))
    print('==================================')
    print('Data frame with transformed columns')
```

```
    print('====================================')
    train_df.show()
    print('================')
    print('Data frame sizes')
    print('================')
    train_rows = train_df.filter(~train_df.Validation).count()
    val_rows = train_df.filter(train_df.Validation).count()
    test_rows = test_df.count()
    print('Training: %d' % train_rows)
    print('Validation: %d' % val_rows)
    print('Test: %d' % test_rows)
    # ============== #
    # MODEL TRAINING #
    # ============== #
    print('==============')
    print('Model training')
    print('==============')
    def exp_rmspe(y_true, y_pred):
        """Competition evaluation metric, expects logarithic inputs."""
        pct = tf.square((tf.exp(y_true) - tf.exp(y_pred)) /
tf.exp(y_true))
        # Compute mean excluding stores with zero denominator.
        x = tf.reduce_sum(tf.where(y_true > 0.001, pct,
tf.zeros_like(pct)))
        y = tf.reduce_sum(tf.where(y_true > 0.001, tf.ones_like(pct),
tf.zeros_like(pct)))
        return tf.sqrt(x / y)
    def act_sigmoid_scaled(x):
        """Sigmoid scaled to logarithm of maximum sales scaled by 20%."""
        return tf.nn.sigmoid(x) * tf.math.log(max_sales) * 1.2
    CUSTOM_OBJECTS = {'exp_rmspe': exp_rmspe,
                      'act_sigmoid_scaled': act_sigmoid_scaled}
    # Disable GPUs when building the model to prevent memory leaks
    if LooseVersion(tf.__version__) >= LooseVersion('2.0.0'):
        # See https://github.com/tensorflow/tensorflow/issues/33168
        os.environ['CUDA_VISIBLE_DEVICES'] = '-1'
    else:

K.set_session(tf.Session(config=tf.ConfigProto(device_count={'GPU': 0})))
    # Build the model.
    inputs = {col: Input(shape=(1,), name=col) for col in all_cols}
    embeddings = [Embedding(len(vocab[col]), 10, input_length=1,
name='emb_' + col)(inputs[col])
                  for col in categorical_cols]
    continuous_bn = Concatenate()([Reshape((1, 1), name='reshape_' +
col)(inputs[col])
```

```
                                    for col in continuous_cols])
    continuous_bn = BatchNormalization()(continuous_bn)
    x = Concatenate()(embeddings + [continuous_bn])
    x = Flatten()(x)
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dense(1000, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dense(500, activation='relu',
kernel_regularizer=tf.keras.regularizers.l2(0.00005))(x)
    x = Dropout(0.5)(x)
    output = Dense(1, activation=act_sigmoid_scaled)(x)
    model = tf.keras.Model([inputs[f] for f in all_cols], output)
    model.summary()
    opt = tf.keras.optimizers.Adam(lr=args.learning_rate, epsilon=1e-3)
    # Checkpoint callback to specify options for the returned Keras model
    ckpt_callback = BestModelCheckpoint(monitor='val_loss', mode='auto',
save_freq='epoch')
    # Horovod: run training.
    store = Store.create(args.work_dir)
    backend = SparkBackend(num_proc=args.num_proc,
                           stdout=sys.stdout, stderr=sys.stderr,
                           prefix_output_with_timestamp=True)
    keras_estimator = hvd.KerasEstimator(backend=backend,
                                         store=store,
                                         model=model,
                                         optimizer=opt,
                                         loss='mae',
                                         metrics=[exp_rmspe],
                                         custom_objects=CUSTOM_OBJECTS,
                                         feature_cols=all_cols,
                                         label_cols=['Sales'],
                                         validation='Validation',
                                         batch_size=args.batch_size,
                                         epochs=args.epochs,
                                         verbose=2,

checkpoint_callback=ckpt_callback)
    keras_model =
keras_estimator.fit(train_df).setOutputCols(['Sales_output'])
    history = keras_model.getHistory()
    best_val_rmspe = min(history['val_exp_rmspe'])
    print('Best RMSPE: %f' % best_val_rmspe)
    # Save the trained model.
```

```
    keras_model.save(args.local_checkpoint_file)
    print('Written checkpoint to %s' % args.local_checkpoint_file)
    # ================ #
    # FINAL PREDICTION #
    # ================ #
    print('================')
    print('Final prediction')
    print('================')
    pred_df=keras_model.transform(test_df)
    pred_df.printSchema()
    pred_df.show(5)
    # Convert from log domain to real Sales numbers
    pred_df=pred_df.withColumn('Sales_pred', F.exp(pred_df.Sales_output))
    submission_df = pred_df.select(pred_df.Id.cast(T.IntegerType()),
pred_df.Sales_pred).toPandas()
    submission_df.sort_values(by=['Id']).to_csv(args.local_submission_csv,
index=False)
    print('Saved predictions to %s' % args.local_submission_csv)
    spark.stop()
```

O terceiro script é `run_classification_criteo_spark.py`.

```
import tempfile, string, random, os, uuid
import argparse, datetime, sys, shutil
import csv
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.callbacks import EarlyStopping
from pyspark import SparkContext
from pyspark.sql import SparkSession, SQLContext, Row, DataFrame
from pyspark.mllib import linalg as mllib_linalg
from pyspark.mllib.linalg import SparseVector as mllibSparseVector
from pyspark.mllib.linalg import VectorUDT as mllibVectorUDT
from pyspark.mllib.linalg import Vector as mllibVector, Vectors as
mllibVectors
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.classification import LogisticRegressionWithSGD
from pyspark.ml import linalg as ml_linalg
from pyspark.ml.linalg import VectorUDT as mlVectorUDT
from pyspark.ml.linalg import SparseVector as mlSparseVector
from pyspark.ml.linalg import Vector as mlVector, Vectors as mlVectors
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import OneHotEncoder
from math import log
from math import exp  # exp(-t) = e^-t
```

```
from operator import add
from pyspark.sql.functions import udf, split, lit
from pyspark.sql.functions import size, sum as sqlsum
import pyspark.sql.functions as F
import pyspark.sql.types as T
from pyspark.sql.types import ArrayType, StructType, StructField,
LongType, StringType, IntegerType, FloatType
from pyspark.sql.functions import explode, col, log, when
from collections import defaultdict
import pandas as pd
import pyspark.pandas as ps
from sklearn.metrics import log_loss, roc_auc_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, MinMaxScaler
from deepctr.models import DeepFM
from deepctr.feature_column import SparseFeat, DenseFeat,
get_feature_names
spark = SparkSession.builder \
    .master("yarn") \
    .appName("deep_ctr_classification") \
    .config("spark.jars.packages", "io.github.ravwojdyla:spark-schema-
utils_2.12:0.1.0") \
    .config("spark.executor.cores", "1") \
    .config('spark.executor.memory', '5gb') \
    .config('spark.executor.memoryOverhead', '1500') \
    .config('spark.driver.memoryOverhead', '1500') \
    .config("spark.sql.shuffle.partitions", "480") \
    .config("spark.sql.execution.arrow.enabled", "true") \
    .config("spark.driver.maxResultSize", "50gb") \
    .getOrCreate()
# spark.conf.set("spark.sql.execution.arrow.enabled", "true") # deprecated
print("Apache Spark version:")
print(spark.version)
sc = spark.sparkContext
sqlContext = SQLContext(sc)
parser = argparse.ArgumentParser(description='Spark DCN CTR Prediction
Example',

formatter_class=argparse.ArgumentDefaultsHelpFormatter)
parser.add_argument('--data-dir', default='file://' + os.getcwd(),
                    help='location of data on local filesystem (prefixed
with file://) or on HDFS')
def process_input_file(file_name, sparse_feat, dense_feat):
    # Need this preprocessing to turn Criteo raw file into CSV:
    print("START processing input file...")
    # only convert the file ONCE
```

```
    # sample = open(file_name)
    # sample = '\n'.join([str(x.replace('\n', '').replace('\t', ',')) for
x in sample])
    # # Add header in data file and save as CSV
    # header = ','.join(str(x) for x in (['label'] + dense_feat +
sparse_feat))
    # with open('/sparkdemo/tr-4570-data/ctr_train.csv', mode='w',
encoding="utf-8") as f:
    #     f.write(header + '\n' + sample)
    #     f.close()
    # print("Raw training file processed and saved as CSV: ", f.name)
    raw_df = sqlContext.read.option("header", True).csv(file_name)
    raw_df.show(5, False)
    raw_df.printSchema()
    # convert columns I1 to I13 from string to integers
    conv_df = raw_df.select(col('label').cast("double"),
                            *(col(i).cast("float").alias(i) for i in
raw_df.columns if i in dense_feat),
                            *(col(c) for c in raw_df.columns if c in
sparse_feat))
    print("Schema of raw_df with integer columns type changed:")
    conv_df.printSchema()
    # result_pdf = conv_df.select("*").toPandas()
    tmp_df = conv_df.na.fill(0, dense_feat)
    result_df = tmp_df.na.fill('-1', sparse_feat)
    result_df.show()
    return result_df
if __name__ == "__main__":
    args = parser.parse_args()
    # Pandas read CSV
    # data = pd.read_csv('%s/criteo_sample.txt' % args.data_dir)
    # print("Obtained Pandas df.")
    dense_features = ['I' + str(i) for i in range(1, 14)]
    sparse_features = ['C' + str(i) for i in range(1, 27)]
    # Spark read CSV
    # process_input_file('%s/train.txt' % args.data_dir, sparse_features,
dense_features) # run only ONCE
    spark_df = process_input_file('%s/data.txt' % args.data_dir,
sparse_features, dense_features) # sample data
    # spark_df = process_input_file('%s/ctr_train.csv' % args.data_dir,
sparse_features, dense_features)
    print("Obtained Spark df and filled in missing features.")
    data = spark_df
    # Pandas
    #data[sparse_features] = data[sparse_features].fillna('-1', )
    #data[dense_features] = data[dense_features].fillna(0, )
```

```
    target = ['label']
    label_npa = data.select("label").toPandas().to_numpy()
    print("label numPy array has length = ", len(label_npa)) # 45,840,617
w/ 11GB dataset
    label_npa.ravel()
    label_npa.reshape(len(label_npa), )
    # 1.Label Encoding for sparse features,and do simple Transformation
for dense features
    print("Before LabelEncoder():")
    data.printSchema()  # label: float (nullable = true)
    for feat in sparse_features:
        lbe = LabelEncoder()
        tmp_pdf = data.select(feat).toPandas().to_numpy()
        tmp_ndarray = lbe.fit_transform(tmp_pdf)
        print("After LabelEncoder(), tmp_ndarray[0] =", tmp_ndarray[0])
        # print("Data tmp PDF after lbe transformation, the output ndarray
has length = ", len(tmp_ndarray)) # 45,840,617 for 11GB dataset
        tmp_ndarray.ravel()
        tmp_ndarray.reshape(len(tmp_ndarray), )
        out_ndarray = np.column_stack([label_npa, tmp_ndarray])
        pdf = pd.DataFrame(out_ndarray, columns=['label', feat])
        s_df = spark.createDataFrame(pdf)
        s_df.printSchema() # label: double (nullable = true)
        print("Before joining data df with s_df, s_df example rows:")
        s_df.show(1, False)
        data = data.drop(feat).join(s_df, 'label').drop('label')
        print("After LabelEncoder(), data df example rows:")
        data.show(1, False)
        print("Finished processing sparse_features: ", feat)
    print("Data DF after label encoding: ")
    data.show()
    data.printSchema()
    mms = MinMaxScaler(feature_range=(0, 1))
    # data[dense_features] = mms.fit_transform(data[dense_features]) # for
Pandas df
    tmp_pdf = data.select(dense_features).toPandas().to_numpy()
    tmp_ndarray = mms.fit_transform(tmp_pdf)
    tmp_ndarray.ravel()
    tmp_ndarray.reshape(len(tmp_ndarray), len(tmp_ndarray[0]))
    out_ndarray = np.column_stack([label_npa, tmp_ndarray])
    pdf = pd.DataFrame(out_ndarray, columns=['label'] + dense_features)
    s_df = spark.createDataFrame(pdf)
    s_df.printSchema()
    data.drop(*dense_features).join(s_df, 'label').drop('label')
    print("Finished processing dense_features: ", dense_features)
    print("Data DF after MinMaxScaler: ")
```

```
    data.show()

    # 2.count #unique features for each sparse field,and record dense
feature field name
    fixlen_feature_columns = [SparseFeat(feat,
vocabulary_size=data.select(feat).distinct().count() + 1, embedding_dim=4)
                              for i, feat in enumerate(sparse_features)] +
\
                             [DenseFeat(feat, 1, ) for feat in
dense_features]
    dnn_feature_columns = fixlen_feature_columns
    linear_feature_columns = fixlen_feature_columns
    feature_names = get_feature_names(linear_feature_columns +
dnn_feature_columns)
    # 3.generate input data for model
    # train, test = train_test_split(data.toPandas(), test_size=0.2,
random_state=2020) # Pandas; might hang for 11GB data
    train, test = data.randomSplit(weights=[0.8, 0.2], seed=200)
    print("Training dataset size = ", train.count())
    print("Testing dataset size = ", test.count())
    # Pandas:
    # train_model_input = {name: train[name] for name in feature_names}
    # test_model_input = {name: test[name] for name in feature_names}
    # Spark DF:
    train_model_input = {}
    test_model_input = {}
    for name in feature_names:
        if name.startswith('I'):
            tr_pdf = train.select(name).toPandas()
            train_model_input[name] = pd.to_numeric(tr_pdf[name])
            ts_pdf = test.select(name).toPandas()
            test_model_input[name] = pd.to_numeric(ts_pdf[name])
    # 4.Define Model,train,predict and evaluate
    model = DeepFM(linear_feature_columns, dnn_feature_columns,
task='binary')
    model.compile("adam", "binary_crossentropy",
                  metrics=['binary_crossentropy'], )
    lb_pdf = train.select(target).toPandas()
    history = model.fit(train_model_input,
pd.to_numeric(lb_pdf['label']).values,
                        batch_size=256, epochs=10, verbose=2,
validation_split=0.2, )
    pred_ans = model.predict(test_model_input, batch_size=256)
    print("test LogLoss",
round(log_loss(pd.to_numeric(test.select(target).toPandas()).values,
pred_ans), 4))
```

```
    print("test AUC",
round(roc_auc_score(pd.to_numeric(test.select(target).toPandas()).values,
pred_ans), 4))
```

Conclusão

Neste documento, discutimos a arquitetura do Apache Spark, casos de uso do cliente e o portfólio de armazenamento da NetApp em relação a big data, análise moderna e IA, ML e DL. Em nossos testes de validação de desempenho baseados em ferramentas de benchmarking padrão do setor e na demanda do cliente, as soluções NetApp Spark demonstraram desempenho superior em relação aos sistemas Hadoop nativos. Uma combinação dos casos de uso do cliente e dos resultados de desempenho apresentados neste relatório pode ajudar você a escolher uma solução Spark apropriada para sua implantação.

Onde encontrar informações adicionais

As seguintes referências foram utilizadas neste TR:

- Arquitetura e componentes do Apache Spark

  "http://spark.apache.org/docs/latest/cluster-overview.html"

- Casos de uso do Apache Spark

  "https://www.qubole.com/blog/big-data/apache-spark-use-cases/"

- Spark NLP

  "https://www.johnsnowlabs.com/spark-nlp/"

- BERTO

  "https://arxiv.org/abs/1810.04805"

- Rede profunda e cruzada para previsões de cliques em anúncios

  "https://arxiv.org/abs/1708.05123"

- FlexGroup

  https://www.netapp.com/pdf.html?item=/media/7337-tr4557pdf.pdf

- Transmissão ETL

  "https://www.infoq.com/articles/apache-spark-streaming"

- Soluções NetApp E-Series para Hadoop

  "https://www.netapp.com/media/16420-tr-3969.pdf"

- Soluções modernas de análise de dados da NetApp

  "Soluções de análise de dados"

- SnapMirror

  "https://docs.netapp.com/us-en/ontap/data-protection/snapmirror-replication-concept.html"

- XCP

  https://mysupport.netapp.com/documentation/docweb/index.html?productID=63942&language=en-US

- BlueXP Copiar e Sincronizar

  "https://cloud.netapp.com/cloud-sync-service"

- Kit de ferramentas DataOps

  "https://github.com/NetApp/netapp-dataops-toolkit"

Informações sobre direitos autorais

Informações sobre marcas comerciais