Soluciones de almacenamiento de NetApp para Apache Spark

NetApp artificial intelligence solutions

NetApp
February 12, 2026

Tabla de contenidos

Soluciones de almacenamiento de NetApp para Apache Spark

TR-4570: Soluciones de almacenamiento de NetApp para Apache Spark: Arquitectura, casos de uso y resultados de rendimiento

Rick Huang, Karthikeyan Nagalingam, NetApp

Este documento se centra en la arquitectura Apache Spark, los casos de uso de los clientes y la cartera de almacenamiento de NetApp relacionada con el análisis de big data y la inteligencia artificial (IA). También presenta varios resultados de pruebas utilizando herramientas de inteligencia artificial, aprendizaje automático (ML) y aprendizaje profundo (DL) estándar de la industria contra un sistema Hadoop típico para que pueda elegir la solución Spark adecuada. Para comenzar, necesita una arquitectura Spark, componentes apropiados y dos modos de implementación (clúster y cliente).

Este documento también proporciona casos de uso de clientes para abordar problemas de configuración y analiza una descripción general de la cartera de almacenamiento de NetApp relevante para análisis de big data e IA, ML y DL con Spark. Luego finalizamos con los resultados de las pruebas derivadas de los casos de uso específicos de Spark y la cartera de soluciones NetApp Spark.

Desafíos del cliente

Esta sección se centra en los desafíos de los clientes con el análisis de big data y la IA/ML/DL en industrias de crecimiento de datos, como el comercio minorista, el marketing digital, la banca, la fabricación discreta, la fabricación de procesos, el gobierno y los servicios profesionales.

Rendimiento impredecible

Las implementaciones tradicionales de Hadoop generalmente utilizan hardware básico. Para mejorar el rendimiento, debe ajustar la red, el sistema operativo, el clúster Hadoop, los componentes del ecosistema como Spark y el hardware. Incluso si ajusta cada capa, puede ser difícil lograr los niveles de rendimiento deseados porque Hadoop se ejecuta en hardware básico que no fue diseñado para un alto rendimiento en su entorno.

Fallos de medios y nodos

Incluso en condiciones normales, el hardware comercial es propenso a fallar. Si falla un disco en un nodo de datos, el maestro Hadoop considera, por defecto, que ese nodo no está en buen estado. Luego, copia datos específicos de ese nodo a través de la red desde réplicas a un nodo en buen estado. Este proceso ralentiza los paquetes de red para cualquier trabajo de Hadoop. Luego, el clúster debe volver a copiar los datos y eliminar los datos sobre replicados cuando el nodo en mal estado vuelva a un estado correcto.

Dependencia del proveedor de Hadoop

Los distribuidores de Hadoop tienen su propia distribución de Hadoop con su propia versión, lo que limita al cliente a esas distribuciones. Sin embargo, muchos clientes requieren soporte para análisis en memoria que no vincule al cliente a distribuciones específicas de Hadoop. Necesitan la libertad de cambiar distribuciones y

aún así llevar consigo sus análisis.

Falta de soporte para más de un idioma

Los clientes a menudo necesitan soporte para varios idiomas además de los programas Java MapReduce para ejecutar sus trabajos. Opciones como SQL y scripts brindan más flexibilidad para obtener respuestas, más opciones para organizar y recuperar datos y formas más rápidas de mover datos a un marco de análisis.

Dificultad de uso

Desde hace algún tiempo, la gente se ha quejado de que Hadoop es difícil de usar. Aunque Hadoop se ha vuelto más simple y más poderoso con cada nueva versión, esta crítica ha persistido. Hadoop requiere que usted comprenda los patrones de programación Java y MapReduce, un desafío para los administradores de bases de datos y personas con habilidades de programación tradicionales.

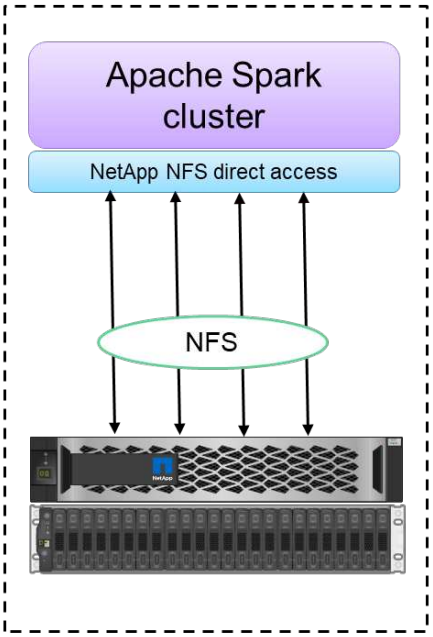
Marcos y herramientas complicados

Los equipos de IA empresariales enfrentan múltiples desafíos. Incluso con un conocimiento experto en ciencia de datos, las herramientas y los marcos para diferentes ecosistemas de implementación y aplicaciones podrían no ser fácilmente trasladables de uno a otro. Una plataforma de ciencia de datos debe integrarse perfectamente con las plataformas de big data correspondientes creadas en Spark con facilidad de movimiento de datos, modelos reutilizables, código listo para usar y herramientas que respalden las mejores prácticas para crear prototipos, validar, controlar versiones, compartir, reutilizar e implementar rápidamente modelos en producción.

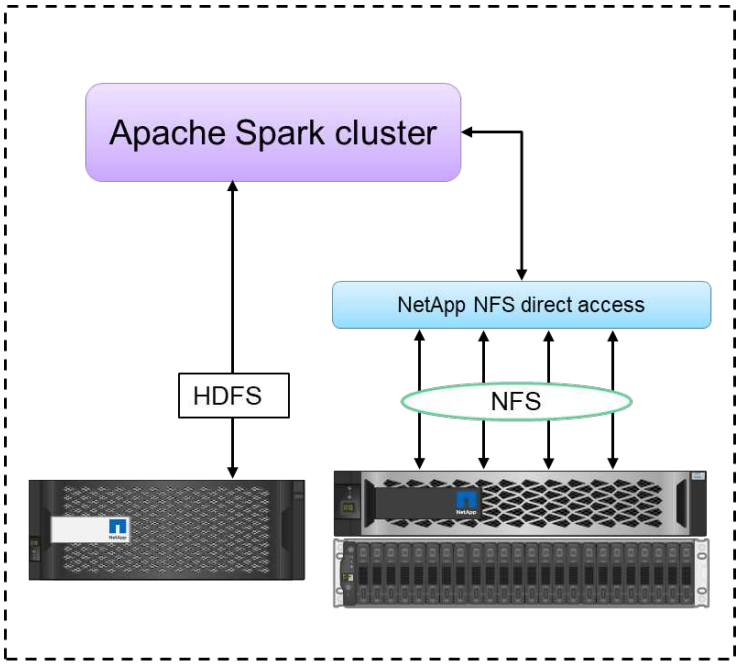
¿Por qué elegir NetApp?

NetApp puede mejorar su experiencia con Spark de las siguientes maneras:

- El acceso directo a NFS de NetApp (que se muestra en la figura a continuación) permite a los clientes ejecutar trabajos de análisis de big data en sus datos NFSv3 o NFSv4 existentes o nuevos sin mover ni copiar los datos. Evita copias múltiples de datos y elimina la necesidad de sincronizar los datos con una fuente.
- Almacenamiento más eficiente y menos replicación de servidores. Por ejemplo, la solución NetApp E-Series Hadoop requiere dos en lugar de tres réplicas de los datos, y la solución FAS Hadoop requiere una fuente de datos pero no replicación ni copias de datos. Las soluciones de almacenamiento de NetApp también producen menos tráfico de servidor a servidor.
- Mejor comportamiento de los trabajos y clústeres de Hadoop durante fallas de unidades y nodos.
- Mejor rendimiento en la ingesta de datos.



Configuration 1: NFS as primary storage



Configuration 2: HDFS and NFS in single Spark cluster

Por ejemplo, en el sector financiero y sanitario, el traslado de datos de un lugar a otro debe cumplir obligaciones legales, lo que no es una tarea fácil. En este escenario, el acceso directo de NetApp NFS analiza los datos financieros y de atención médica desde su ubicación original. Otro beneficio clave es que el uso del acceso directo NFS de NetApp simplifica la protección de los datos de Hadoop mediante el uso de comandos nativos de Hadoop y la habilitación de flujos de trabajo de protección de datos con la amplia cartera de gestión de datos de NetApp.

El acceso directo NFS de NetApp ofrece dos tipos de opciones de implementación para clústeres Hadoop/Spark:

- De forma predeterminada, los clústeres Hadoop o Spark utilizan el sistema de archivos distribuido Hadoop (HDFS) para el almacenamiento de datos y el sistema de archivos predeterminado. El acceso directo NFS de NetApp puede reemplazar el HDFS predeterminado con almacenamiento NFS como sistema de archivos predeterminado, lo que permite el análisis directo de datos NFS.
- En otra opción de implementación, el acceso directo NFS de NetApp admite la configuración de NFS como almacenamiento adicional junto con HDFS en un solo clúster Hadoop o Spark. En este caso, el cliente puede compartir datos a través de exportaciones NFS y acceder a ellos desde el mismo clúster junto con los datos HDFS.

Los beneficios clave de utilizar el acceso directo NFS de NetApp incluyen los siguientes:

- Analizar los datos desde su ubicación actual, lo que evita la tarea, que consume mucho tiempo y rendimiento, de mover datos analíticos a una infraestructura Hadoop como HDFS.
- Reducir el número de réplicas de tres a una.
- Permitir a los usuarios disociar el procesamiento y el almacenamiento para escalarlos de forma independiente.
- Proporcionar protección de datos empresariales aprovechando las ricas capacidades de gestión de datos de ONTAP.
- Certificación con la plataforma de datos Hortonworks.
- Habilitación de implementaciones de análisis de datos híbridos.

- Reducir el tiempo de backup aprovechando la capacidad multihilo dinámico.

Ver"TR-4657: Soluciones de datos en la nube híbrida de NetApp : Spark y Hadoop, basadas en casos de uso de clientes" para realizar copias de seguridad de datos de Hadoop, realizar copias de seguridad y recuperación ante desastres desde la nube a las instalaciones locales, habilitar DevTest en datos de Hadoop existentes, protección de datos y conectividad multicloud, y acelerar las cargas de trabajo de análisis.

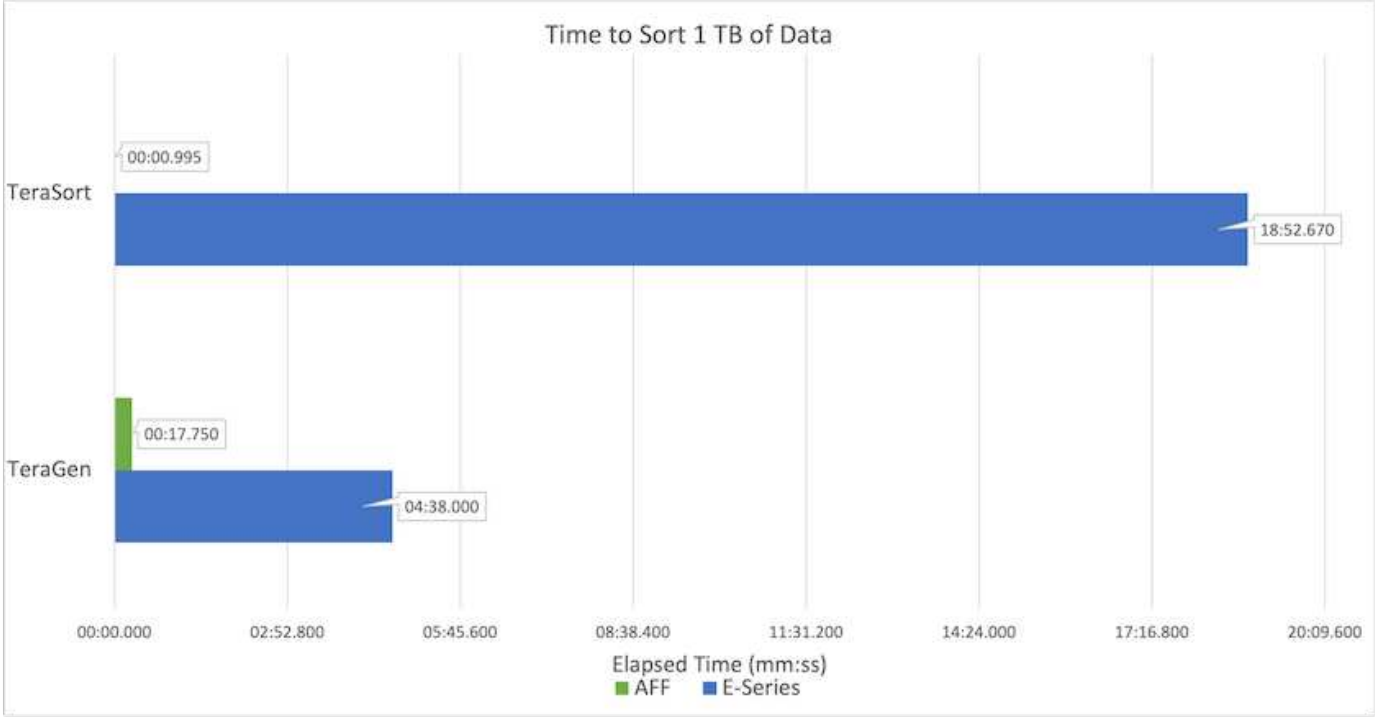
Las siguientes secciones describen las capacidades de almacenamiento que son importantes para los clientes de Spark.

Nivelación de almacenamiento

Con los niveles de almacenamiento de Hadoop, puede almacenar archivos con diferentes tipos de almacenamiento de acuerdo con una política de almacenamiento. Los tipos de almacenamiento incluyen `hot` , `cold` , `warm` , `all_ssd` , `one_ssd` , y `lazy_persist` .

Realizamos la validación de la clasificación en niveles del almacenamiento de Hadoop en un controlador de almacenamiento NetApp AFF y un controlador de almacenamiento E-Series con unidades SSD y SAS con diferentes políticas de almacenamiento. El clúster Spark con AFF-A800 tiene cuatro nodos de trabajo de cómputo, mientras que el clúster con E-Series tiene ocho. Esto es principalmente para comparar el rendimiento de las unidades de estado sólido (SSD) frente a los discos duros (HDD).

La siguiente figura muestra el rendimiento de las soluciones NetApp para un SSD Hadoop.



- La configuración básica de NL-SAS utilizó ocho nodos de cómputo y 96 unidades NL-SAS. Esta configuración generó 1 TB de datos en 4 minutos y 38 segundos. Ver "Solución NetApp E-Series TR-3969 para Hadoop" para obtener detalles sobre la configuración del clúster y del almacenamiento.
- Con TeraGen, la configuración SSD generó 1 TB de datos 15,66 veces más rápido que la configuración NL-SAS. Además, la configuración SSD utilizó la mitad del número de nodos de cómputo y la mitad del número de unidades de disco (24 unidades SSD en total). Según el tiempo de finalización del trabajo, fue casi el doble de rápido que la configuración NL-SAS.

- Con TeraSort, la configuración SSD ordenó 1 TB de datos 1138,36 veces más rápido que la configuración NL-SAS. Además, la configuración SSD utilizó la mitad del número de nodos de cómputo y la mitad del número de unidades de disco (24 unidades SSD en total). Por lo tanto, por unidad, fue aproximadamente tres veces más rápido que la configuración NL-SAS.
- La conclusión es que la transición de los discos giratorios a la tecnología flash mejora el rendimiento. El número de nodos de cómputo no fue el cuello de botella. Con el almacenamiento all-flash de NetApp, el rendimiento en tiempo de ejecución escala bien.
- Con NFS, los datos eran funcionalmente equivalentes a estar agrupados todos juntos, lo que puede reducir la cantidad de nodos de cómputo según su carga de trabajo. Los usuarios del clúster Apache Spark no tienen que reequilibrar manualmente los datos al cambiar la cantidad de nodos de cómputo.

Escalado del rendimiento - Escalamiento horizontal

Cuando necesita más potencia de procesamiento de un clúster Hadoop en una solución AFF , puede agregar nodos de datos con una cantidad adecuada de controladores de almacenamiento. NetApp recomienda comenzar con cuatro nodos de datos por matriz de controlador de almacenamiento y aumentar la cantidad a ocho nodos de datos por controlador de almacenamiento, según las características de la carga de trabajo.

AFF y FAS son perfectos para análisis in situ. Según los requisitos de cálculo, puede agregar administradores de nodos, y las operaciones no disruptivas le permiten agregar un controlador de almacenamiento a pedido sin tiempo de inactividad. Ofrecemos funciones avanzadas con AFF y FAS, como compatibilidad con medios NVME, eficiencia garantizada, reducción de datos, calidad de servicio, análisis predictivo, niveles de nube, replicación, implementación de nube y seguridad. Para ayudar a los clientes a satisfacer sus necesidades, NetApp ofrece funciones como análisis del sistema de archivos, cuotas y equilibrio de carga integrado sin costos de licencia adicionales. NetApp tiene un mejor rendimiento en cantidad de trabajos simultáneos, menor latencia, operaciones más simples y mayor rendimiento de gigabytes por segundo que nuestros competidores. Además, NetApp Cloud Volumes ONTAP se ejecuta en los tres principales proveedores de nube.

Escalado del rendimiento: escalar hacia arriba

Las funciones de ampliación le permiten agregar unidades de disco a los sistemas AFF, FAS y E-Series cuando necesita capacidad de almacenamiento adicional. Con Cloud Volumes ONTAP, escalar el almacenamiento al nivel de PB es una combinación de dos factores: agrupar los datos poco utilizados en el almacenamiento de objetos desde el almacenamiento en bloque y apilar licencias de Cloud Volumes ONTAP sin procesamiento adicional.

Múltiples protocolos

Los sistemas NetApp admiten la mayoría de los protocolos para implementaciones de Hadoop, incluidos SAS, iSCSI, FCP, InfiniBand y NFS.

Soluciones operativas y soportadas

Las soluciones Hadoop descritas en este documento son compatibles con NetApp. Estas soluciones también están certificadas con los principales distribuidores de Hadoop. Para obtener más información, consulte la "Hortonworks" sitio y Cloudera "proceso de dar un título" y "pareja" sitios.

Público objetivo

El mundo del análisis y la ciencia de datos afecta a múltiples disciplinas en TI y negocios:

- El científico de datos necesita la flexibilidad de utilizar las herramientas y bibliotecas que elija.

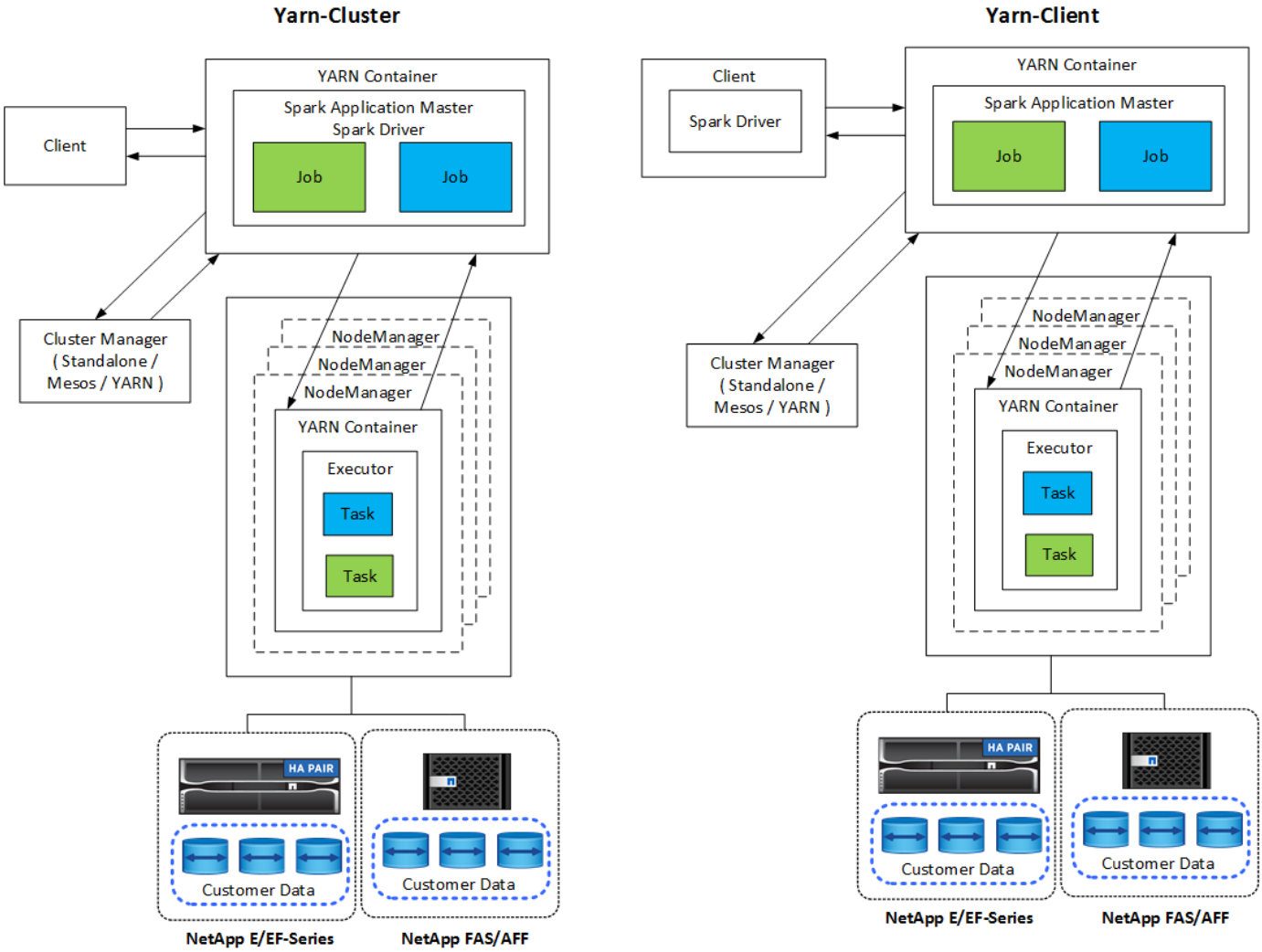
- El ingeniero de datos necesita saber cómo fluyen los datos y dónde residen.
- Un ingeniero de DevOps necesita las herramientas para integrar nuevas aplicaciones de IA y ML en sus canales de CI y CD.
- Los administradores y arquitectos de la nube deben poder configurar y gestionar recursos de nube híbrida.
- Los usuarios comerciales quieren tener acceso a aplicaciones de análisis, inteligencia artificial, aprendizaje automático y aprendizaje automático.

En este informe técnico, describimos cómo NetApp AFF, E-Series, StorageGRID, acceso directo NFS, Apache Spark, Horovod y Keras ayudan a cada uno de estos roles a aportar valor al negocio.

Tecnología de soluciones

Apache Spark es un marco de programación popular para escribir aplicaciones Hadoop que funciona directamente con el sistema de archivos distribuidos Hadoop (HDFS). Spark está listo para producción, admite el procesamiento de datos de transmisión y es más rápido que MapReduce. Spark tiene almacenamiento en caché de datos en memoria configurable para una iteración eficiente, y el shell de Spark es interactivo para aprender y explorar datos. Con Spark, puedes crear aplicaciones en Python, Scala o Java. Las aplicaciones Spark constan de uno o más trabajos que tienen una o más tareas.

Cada aplicación Spark tiene un controlador Spark. En el modo YARN-Client, el controlador se ejecuta en el cliente localmente. En el modo YARN-Cluster, el controlador se ejecuta en el clúster en el maestro de aplicaciones. En el modo de clúster, la aplicación continúa ejecutándose incluso si el cliente se desconecta.



Hay tres administradores de clúster:

- **Autónomo.** Este administrador es parte de Spark, lo que facilita la configuración de un clúster.
- **Mesos Apache.** Este es un administrador de clúster general que también ejecuta MapReduce y otras aplicaciones.
- **HILO DE HADOOP.** Este es un administrador de recursos en Hadoop 3.

El conjunto de datos distribuidos resilientes (RDD) es el componente principal de Spark. RDD recrea los datos perdidos y faltantes a partir de los datos almacenados en la memoria del clúster y almacena los datos iniciales que provienen de un archivo o se crean mediante programación. Los RDD se crean a partir de archivos, datos en la memoria u otro RDD. La programación Spark realiza dos operaciones: transformación y acciones. La transformación crea un nuevo RDD basado en uno existente. Las acciones devuelven un valor de un RDD.

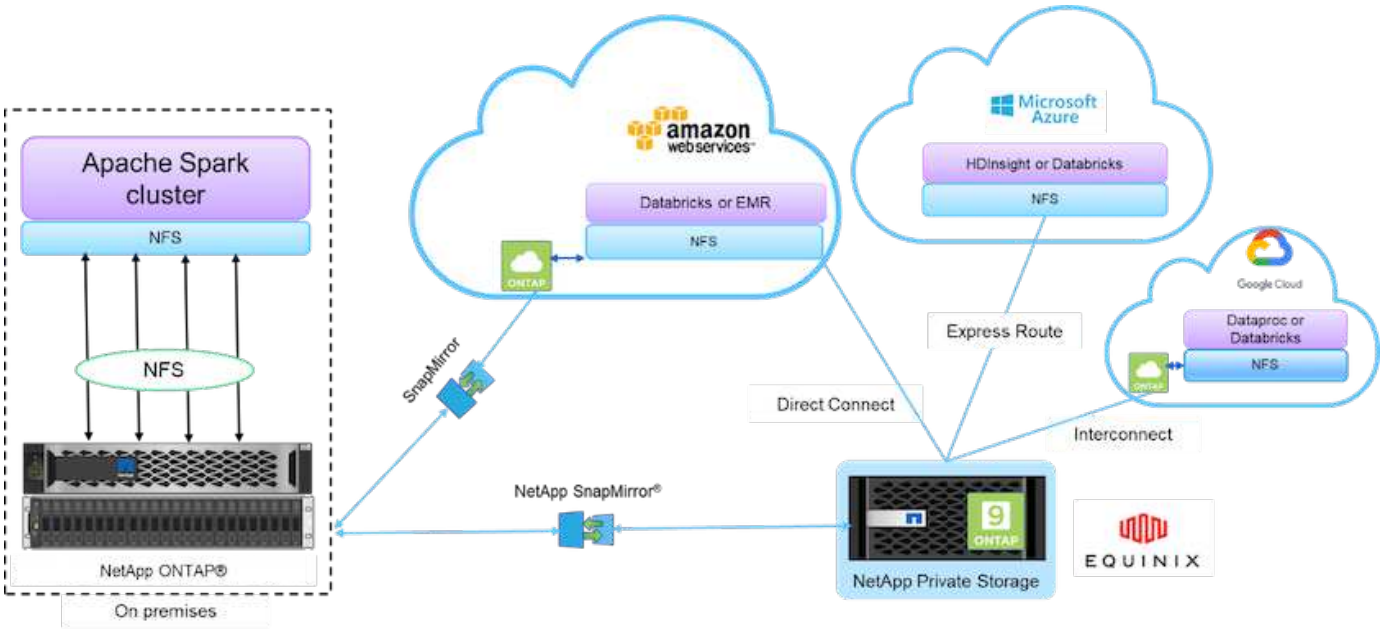
Las transformaciones y acciones también se aplican a los conjuntos de datos y marcos de datos de Spark. Un conjunto de datos es una colección distribuida de datos que proporciona los beneficios de los RDD (tipificación fuerte, uso de funciones lambda) con los beneficios del motor de ejecución optimizado de Spark SQL. Se puede construir un conjunto de datos a partir de objetos JVM y luego manipularlo mediante transformaciones funcionales (mapa, flatMap, filtro, etc.). Un DataFrame es un conjunto de datos organizado en columnas con nombre. Es conceptualmente equivalente a una tabla en una base de datos relacional o un marco de datos en R/Python. Los DataFrames se pueden construir a partir de una amplia variedad de fuentes, como archivos de datos estructurados, tablas en Hive/HBase, bases de datos externas locales o en la nube, o RDD existentes.

Las aplicaciones Spark incluyen uno o más trabajos Spark. Los trabajos ejecutan tareas en ejecutores, y los ejecutores se ejecutan en contenedores YARN. Cada ejecutor se ejecuta en un solo contenedor y los ejecutores existen durante toda la vida de una aplicación. Un ejecutor se fija después de que se inicia la aplicación y YARN no redimensiona el contenedor ya asignado. Un ejecutor puede ejecutar tareas simultáneamente en datos en memoria.

Descripción general de las soluciones Spark de NetApp

NetApp tiene tres carteras de almacenamiento: FAS/ AFF, E-Series y Cloud Volumes ONTAP. Hemos validado AFF y el sistema de almacenamiento E-Series con ONTAP para soluciones Hadoop con Apache Spark.

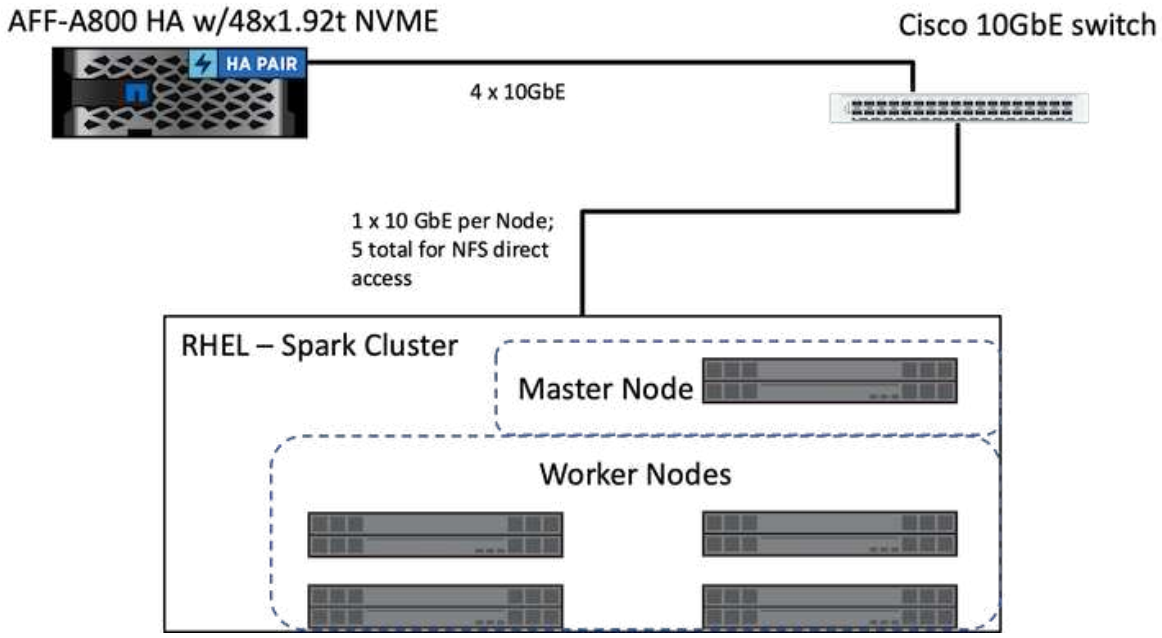
La estructura de datos impulsada por NetApp integra servicios y aplicaciones de gestión de datos (bloques de construcción) para el acceso, control, protección y seguridad de los datos, como se muestra en la siguiente figura.



Los componentes básicos de la figura anterior incluyen:

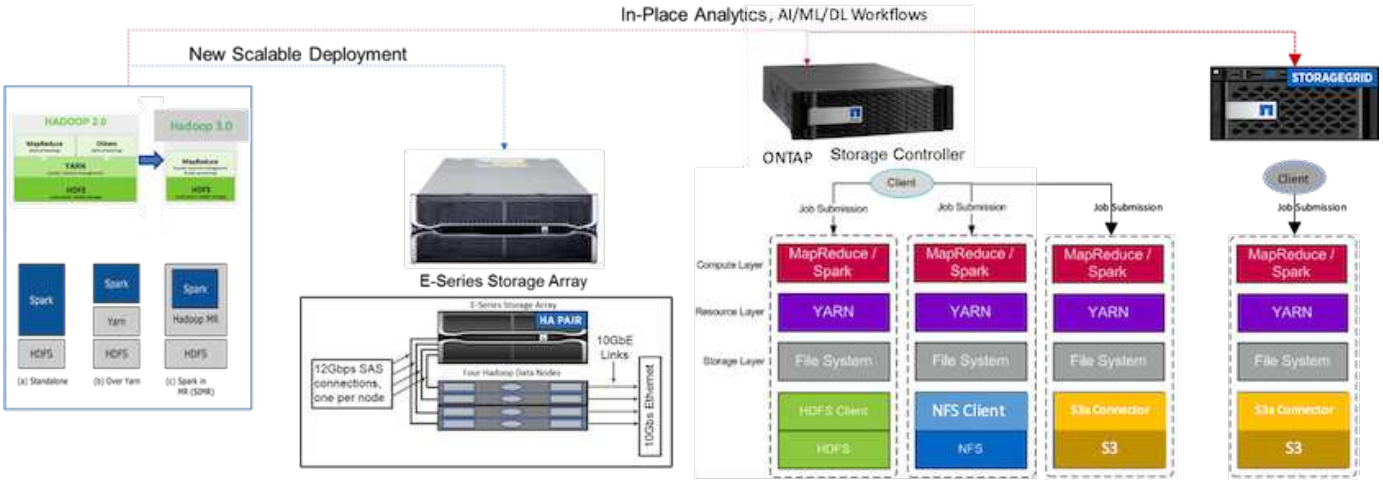
- **Acceso directo a NFS de NetApp .** Proporciona los últimos clústeres Hadoop y Spark con acceso directo a volúmenes NFS de NetApp sin requisitos de software o controladores adicionales.
- * Cloud Volumes ONTAP NetApp Cloud ONTAP y Google Cloud NetApp Volumes.* Almacenamiento conectado definido por software basado en ONTAP que se ejecuta en Amazon Web Services (AWS) o Azure NetApp Files (ANF) en los servicios de nube de Microsoft Azure.
- **Tecnología NetApp SnapMirror .** Proporciona capacidades de protección de datos entre las instancias locales y las de ONTAP Cloud o NPS.
- **Proveedores de servicios en la nube.** Estos proveedores incluyen AWS, Microsoft Azure, Google Cloud e IBM Cloud.
- **PaaS.** Servicios de análisis basados en la nube como Amazon Elastic MapReduce (EMR) y Databricks en AWS, así como Microsoft Azure HDInsight y Azure Databricks.

La siguiente figura muestra la solución Spark con almacenamiento NetApp .

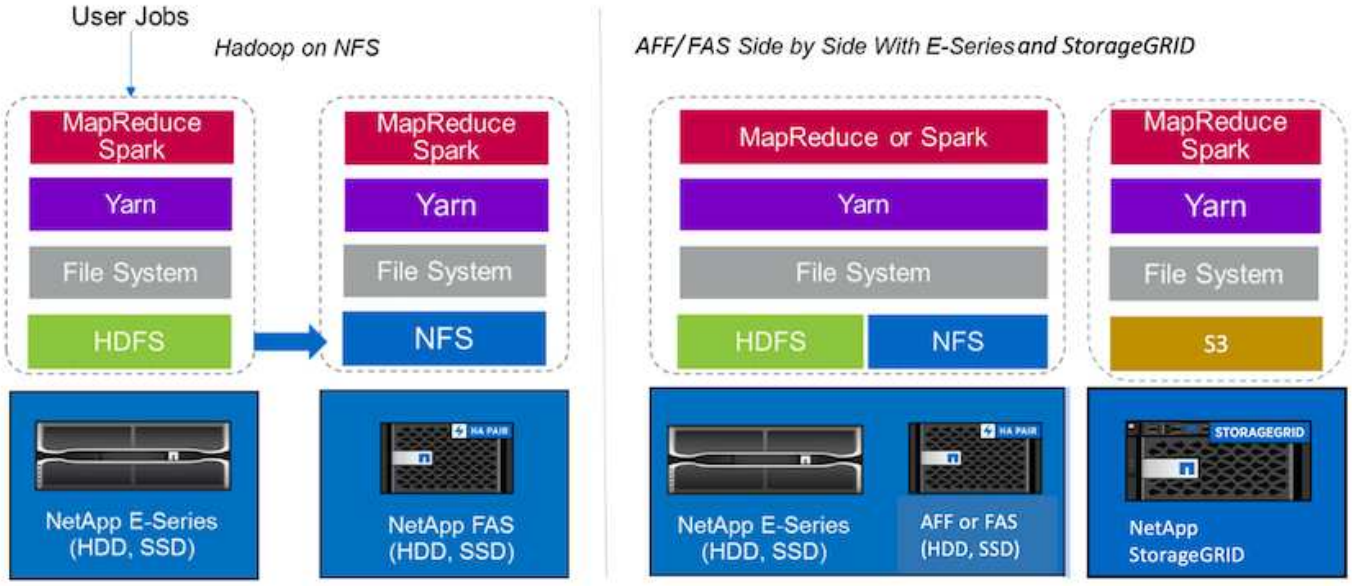


La solución ONTAP Spark utiliza el protocolo de acceso directo NFS de NetApp para análisis locales y flujos de trabajo de IA, ML y DL mediante el acceso a datos de producción existentes. Los datos de producción disponibles para los nodos Hadoop se exportan para realizar trabajos de análisis y de inteligencia artificial, aprendizaje automático y aprendizaje automático en el lugar. Puede acceder a los datos para procesarlos en los nodos Hadoop con acceso directo a NetApp NFS o sin él. En Spark con el independiente o `yarn` Administrador de clústeres, puede configurar un volumen NFS mediante `file://<target_volume>` . Validamos tres casos de uso con diferentes conjuntos de datos. Los detalles de estas validaciones se presentan en la sección "Resultados de las pruebas". (referencia cruzada)

La siguiente figura muestra el posicionamiento del almacenamiento Apache Spark/Hadoop de NetApp .



Identificamos las características únicas de la solución E-Series Spark, la solución AFF/ FAS ONTAP Spark y la solución StorageGRID Spark, y realizamos pruebas y validaciones detalladas. Con base en nuestras observaciones, NetApp recomienda la solución E-Series para instalaciones nuevas y nuevas implementaciones escalables, y la solución AFF/ FAS para análisis locales, IA, ML y cargas de trabajo de DL utilizando datos NFS existentes, y StorageGRID para IA, ML y DL y análisis de datos modernos cuando se requiere almacenamiento de objetos.



Un lago de datos es un repositorio de almacenamiento para grandes conjuntos de datos en formato nativo que pueden usarse para trabajos de análisis, inteligencia artificial, aprendizaje automático y aprendizaje automático. Creamos un repositorio de lago de datos para las soluciones Spark E-Series, AFF/ FAS y StorageGRID SG6060. El sistema E-Series proporciona acceso HDFS al clúster Hadoop Spark, mientras que se accede a los datos de producción existentes a través del protocolo de acceso directo NFS al clúster Hadoop. Para los conjuntos de datos que residen en el almacenamiento de objetos, NetApp StorageGRID proporciona acceso seguro S3 y S3a.

Resumen del caso de uso

En esta página se describen las diferentes áreas en las que se puede utilizar esta solución.

Transmisión de datos

Apache Spark puede procesar datos de streaming, que se utilizan para procesos de extracción, transformación y carga (ETL) de streaming, enriquecimiento de datos, detección de eventos de activación y análisis de sesiones complejas:

- **Transmisión ETL.** Los datos se limpian y agregan continuamente antes de ingresarlos en los almacenes de datos. Netflix utiliza Kafka y Spark Streaming para crear una solución de recomendación de películas en línea y monitoreo de datos en tiempo real que puede procesar miles de millones de eventos por día desde diferentes fuentes de datos. Sin embargo, el ETL tradicional para el procesamiento por lotes se trata de manera diferente. Estos datos se leen primero y luego se convierten a un formato de base de datos antes de escribirse en la base de datos.
- **Enriquecimiento de datos.** Spark Streaming enriquece los datos en vivo con datos estáticos para permitir un análisis de datos más en tiempo real. Por ejemplo, los anunciantes en línea pueden ofrecer anuncios personalizados y específicos basados en información sobre el comportamiento del cliente.
- **Detección de eventos desencadenantes.** Spark Streaming le permite detectar y responder rápidamente a comportamientos inusuales que podrían indicar problemas potencialmente graves. Por ejemplo, las instituciones financieras utilizan desencadenadores para detectar y detener transacciones fraudulentas, y los hospitales utilizan desencadenadores para detectar cambios peligrosos para la salud detectados en los signos vitales de un paciente.

- **Análisis de sesión complejo.** Spark Streaming recopila eventos como la actividad del usuario después de iniciar sesión en un sitio web o aplicación, que luego se agrupan y analizan. Por ejemplo, Netflix utiliza esta funcionalidad para ofrecer recomendaciones de películas en tiempo real.

Para obtener más información sobre la configuración de datos de transmisión, la verificación de Confluent Kafka y las pruebas de rendimiento, consulte"TR-4912: Pautas recomendadas para el almacenamiento en niveles de Confluent Kafka con NetApp" .

aprendizaje automático

El marco integrado de Spark le ayuda a ejecutar consultas repetidas en conjuntos de datos utilizando la biblioteca de aprendizaje automático (MLlib). MLlib se utiliza en áreas como agrupamiento, clasificación y reducción de dimensionalidad para algunas funciones comunes de big data, como inteligencia predictiva, segmentación de clientes para fines de marketing y análisis de sentimientos. MLlib se utiliza en seguridad de red para realizar inspecciones en tiempo real de paquetes de datos en busca de indicios de actividad maliciosa. Ayuda a los proveedores de seguridad a conocer nuevas amenazas y mantenerse a la vanguardia de los piratas informáticos mientras protegen a sus clientes en tiempo real.

aprendizaje profundo

TensorFlow es un marco de aprendizaje profundo popular utilizado en toda la industria. TensorFlow admite el entrenamiento distribuido en un clúster de CPU o GPU. Este entrenamiento distribuido permite a los usuarios ejecutarlo en una gran cantidad de datos con muchas capas profundas.

Hasta hace poco, si queríamos usar TensorFlow con Apache Spark, necesitábamos realizar todo el ETL necesario para TensorFlow en PySpark y luego escribir los datos en un almacenamiento intermedio. Luego, esos datos se cargarían en el clúster TensorFlow para el proceso de entrenamiento real. Este flujo de trabajo requería que el usuario mantuviera dos clústeres diferentes, uno para ETL y otro para el entrenamiento distribuido de TensorFlow. Normalmente, ejecutar y mantener varios clústeres era una tarea tediosa y que consumía mucho tiempo.

Los DataFrames y RDD en versiones anteriores de Spark no eran adecuados para el aprendizaje profundo porque el acceso aleatorio era limitado. En Spark 3.0 con el proyecto Hydrogen, se agrega soporte nativo para los marcos de aprendizaje profundo. Este enfoque permite la programación no basada en MapReduce en el clúster Spark.

Análisis interactivo

Apache Spark es lo suficientemente rápido para realizar consultas exploratorias sin muestrear con lenguajes de desarrollo distintos de Spark, incluidos SQL, R y Python. Spark utiliza herramientas de visualización para procesar datos complejos y visualizarlos de forma interactiva. Spark con transmisión estructurada realiza consultas interactivas sobre datos en vivo en análisis web que le permiten ejecutar consultas interactivas sobre la sesión actual de un visitante web.

Sistema de recomendación

A lo largo de los años, los sistemas de recomendación han traído enormes cambios a nuestras vidas, a medida que las empresas y los consumidores han respondido a cambios dramáticos en las compras en línea, el entretenimiento en línea y muchas otras industrias. De hecho, estos sistemas se encuentran entre las historias de éxito más evidentes de la IA en la producción. En muchos casos de uso práctico, los sistemas de recomendación se combinan con IA conversacional o chatbots interconectados con un backend de PNL para obtener información relevante y producir inferencias útiles.

Hoy en día, muchos minoristas están adoptando modelos de negocio más nuevos, como comprar en línea y

recoger en la tienda, recoger en la acera, autopago, escanear y listo, y más. Estos modelos han cobrado relevancia durante la pandemia de COVID-19 al hacer que las compras sean más seguras y cómodas para los consumidores. La IA es crucial para estas tendencias digitales crecientes, que están influenciadas por el comportamiento del consumidor y viceversa. Para satisfacer las crecientes demandas de los consumidores, aumentar la experiencia del cliente, mejorar la eficiencia operativa y aumentar los ingresos, NetApp ayuda a sus clientes empresariales y empresas a utilizar algoritmos de aprendizaje automático y aprendizaje profundo para diseñar sistemas de recomendación más rápidos y precisos.

Existen varias técnicas populares que se utilizan para proporcionar recomendaciones, incluido el filtrado colaborativo, los sistemas basados en contenido, el modelo de recomendación de aprendizaje profundo (DLRM) y las técnicas híbridas. Los clientes utilizaron anteriormente PySpark para implementar el filtrado colaborativo para crear sistemas de recomendación. Spark MLlib implementa mínimos cuadrados alternos (ALS) para el filtrado colaborativo, un algoritmo muy popular entre las empresas antes del surgimiento de DLRM.

Procesamiento del lenguaje natural

La IA conversacional, posible gracias al procesamiento del lenguaje natural (PLN), es la rama de la IA que ayuda a las computadoras a comunicarse con los humanos. La PNL prevalece en todos los sectores industriales y en muchos casos de uso, desde asistentes inteligentes y chatbots hasta búsquedas de Google y texto predictivo. Según un "Gartner" Predicción: para 2022, el 70% de las personas interactuarán con plataformas de IA conversacional a diario. Para una conversación de alta calidad entre un humano y una máquina, las respuestas deben ser rápidas, inteligentes y que suenen naturales.

Los clientes necesitan una gran cantidad de datos para procesar y entrenar sus modelos de PNL y reconocimiento automático de voz (ASR). También necesitan mover datos a través del borde, el núcleo y la nube, y necesitan el poder de realizar inferencias en milisegundos para establecer una comunicación natural con los humanos. NetApp AI y Apache Spark son una combinación ideal para computación, almacenamiento, procesamiento de datos, entrenamiento de modelos, ajuste e implementación.

El análisis de sentimientos es un campo de estudio dentro de la PNL en el que se extraen sentimientos positivos, negativos o neutrales del texto. El análisis de sentimientos tiene una variedad de casos de uso, desde determinar el desempeño de los empleados del centro de soporte en conversaciones con las personas que llaman hasta brindar respuestas de chatbot automatizadas apropiadas. También se ha utilizado para predecir el precio de las acciones de una empresa basándose en las interacciones entre los representantes de la empresa y la audiencia en las conferencias de ganancias trimestrales. Además, el análisis de sentimientos se puede utilizar para determinar la opinión de un cliente sobre los productos, servicios o soporte proporcionado por la marca.

Usamos el "Spark PNL" biblioteca de "Laboratorios John Snow" para cargar tuberías entrenadas previamente y modelos de Representaciones de Codificador Bidireccional de Transformadores (BERT), incluidos "sentimiento de las noticias financieras" y "FinBERT" , realizando tokenización, reconocimiento de entidades nombradas, entrenamiento de modelos, ajuste y análisis de sentimientos a escala. Spark NLP es la única biblioteca de PNL de código abierto en producción que ofrece transformadores de última generación como BERT, ALBERT, ELECTRA, XLNet, DistilBERT, RoBERTa, DeBERTa, XLM- RoBERTa, Longformer, ELMO, Universal Sentence Encoder, Google T5, MarianMT y GPT2. La biblioteca funciona no solo en Python y R, sino también en el ecosistema JVM (Java, Scala y Kotlin) a escala al extender Apache Spark de forma nativa.

Principales casos de uso y arquitecturas de IA, ML y DL

Los principales casos de uso y metodología de IA, ML y DL se pueden dividir en las siguientes secciones:

Canalizaciones de Spark NLP e inferencia distribuida de TensorFlow

La siguiente lista contiene las bibliotecas de PNL de código abierto más populares que han sido adoptadas por la comunidad de ciencia de datos en diferentes niveles de desarrollo:

- "Kit de herramientas de lenguaje natural (NLTK)" . El kit de herramientas completo para todas las técnicas de PNL. Se mantiene desde principios de la década del 2000.
- "TextoBlob" . Una API de Python de herramientas de PNL fácil de usar construida sobre NLTK y Pattern.
- "PNL de Stanford Core" . Servicios y paquetes de PNL en Java desarrollados por Stanford NLP Group.
- "Gensim" . Topic Modelling for Humans comenzó como una colección de scripts de Python para el proyecto de la Biblioteca Checa de Matemáticas Digitales.
- "SpaCy" . Flujos de trabajo de PNL industrial de extremo a extremo con Python y Cython con aceleración de GPU para transformadores.
- "Texto rápido" . Una biblioteca de PNL gratuita, liviana y de código abierto para el aprendizaje de incrustaciones de palabras y la clasificación de oraciones creada por el laboratorio de investigación de inteligencia artificial (FAIR) de Facebook.

Spark NLP es una solución única y unificada para todas las tareas y requisitos de PNL que permite un software escalable, de alto rendimiento y alta precisión impulsado por PNL para casos de uso de producción reales. Aprovecha el aprendizaje por transferencia e implementa los últimos algoritmos y modelos de última generación en la investigación y en todas las industrias. Debido a la falta de soporte completo por parte de Spark para las bibliotecas anteriores, Spark NLP se creó sobre "Spark ML" aprovechar el motor de procesamiento de datos distribuido en memoria de propósito general de Spark como una biblioteca de PNL de nivel empresarial para flujos de trabajo de producción de misión crítica. Sus anotadores utilizan algoritmos basados en reglas, aprendizaje automático y TensorFlow para impulsar implementaciones de aprendizaje profundo. Esto cubre tareas comunes de PNL que incluyen, entre otras, tokenización, lematización, derivación, etiquetado de partes del discurso, reconocimiento de entidades nombradas, corrección ortográfica y análisis de sentimientos.

Representaciones de codificador bidireccional a partir de transformadores (BERT) es una técnica de aprendizaje automático basada en transformadores para PNL. Popularizó el concepto de preentrenamiento y ajuste fino. La arquitectura del transformador en BERT se originó a partir de la traducción automática, que modela las dependencias a largo plazo mejor que los modelos de lenguaje basados en redes neuronales recurrentes (RNN). También introdujo la tarea de modelado de lenguaje enmascarado (MLM), donde un 15% aleatorio de todos los tokens se enmascaran y el modelo los predice, lo que permite una verdadera bidireccionalidad.

El análisis del sentimiento financiero es un desafío debido al lenguaje especializado y la falta de datos etiquetados en ese dominio. FinBERT, un modelo de lenguaje basado en BERT preentrenado, fue adaptado al dominio en "Reuters TRC2" , un corpus financiero, y ajustado con datos etiquetados ("Frase financiera del banco") para la clasificación del sentimiento financiero. Los investigadores extrajeron 4.500 frases de artículos de noticias con términos financieros. Luego, 16 expertos y estudiantes de maestría con experiencia en finanzas etiquetaron las oraciones como positivas, neutrales y negativas. Creamos un flujo de trabajo Spark de extremo a extremo para analizar el sentimiento de las transcripciones de las llamadas de ganancias de las 10 principales empresas del NASDAQ de 2016 a 2020 utilizando FinBERT y otras dos canalizaciones entrenadas previamente. "Explicar el documento DL") de Spark NLP.

El motor de aprendizaje profundo subyacente para Spark NLP es TensorFlow, una plataforma de código abierto de extremo a extremo para el aprendizaje automático que permite la creación sencilla de modelos, la producción de ML sólida en cualquier lugar y la experimentación potente para la investigación. Por lo tanto, al ejecutar nuestros pipelines en Spark `yarn cluster` En este modo, básicamente estábamos ejecutando TensorFlow distribuido con paralelización de datos y modelos en un nodo maestro y varios nodos de trabajo,

así como almacenamiento conectado a la red montado en el clúster.

Capacitación distribuida de Horovod

La validación central de Hadoop para el rendimiento relacionado con MapReduce se realiza con TeraGen, TeraSort, TeraValidate y DFSIO (lectura y escritura). Los resultados de la validación de TeraGen y TeraSort se presentan en "Solución NetApp E-Series para Hadoop" y en la sección "Niveles de almacenamiento" para AFF.

Basándonos en las solicitudes de los clientes, consideramos que la capacitación distribuida con Spark es uno de los casos de uso más importantes. En este documento, utilizamos el "Hovorod en Spark" para validar el rendimiento de Spark con soluciones locales, nativas de la nube e híbridas de NetApp mediante controladores de almacenamiento NetApp All Flash FAS (AFF), Azure NetApp Files y StorageGRID.

El paquete Horovod en Spark proporciona un envoltorio conveniente alrededor de Horovod que simplifica la ejecución de cargas de trabajo de entrenamiento distribuidas en clústeres Spark, lo que permite un ciclo de diseño de modelo ajustado en el que el procesamiento de datos, el entrenamiento del modelo y la evaluación del modelo se realizan en Spark, donde residen los datos de entrenamiento e inferencia.

Hay dos API para ejecutar Horovod en Spark: una API de estimación de alto nivel y una API de ejecución de nivel inferior. Aunque ambos utilizan el mismo mecanismo subyacente para ejecutar Horovod en los ejecutores Spark, la API Estimator abstrae el procesamiento de datos, el ciclo de entrenamiento del modelo, los puntos de control del modelo, la recopilación de métricas y el entrenamiento distribuido. Utilizamos Horovod Spark Estimators, TensorFlow y Keras para un flujo de trabajo de preparación de datos de extremo a extremo y entrenamiento distribuido basado en "Ventas en tiendas Kaggle Rossmann" competencia.

El guión `keras_spark_horovod_rossmann_estimator.py` se puede encontrar en la sección"Scripts de Python para cada caso de uso principal." Consta de tres partes:

- La primera parte realiza varios pasos de preprocesamiento de datos sobre un conjunto inicial de archivos CSV proporcionados por Kaggle y recopilados por la comunidad. Los datos de entrada se separan en un conjunto de entrenamiento con un `Validation` subconjunto y un conjunto de datos de prueba.
- La segunda parte define un modelo de red neuronal profunda (DNN) Keras con función de activación sigmoidea logarítmica y un optimizador Adam, y realiza un entrenamiento distribuido del modelo utilizando Horovod en Spark.
- La tercera parte realiza una predicción en el conjunto de datos de prueba utilizando el mejor modelo que minimiza el error absoluto medio general del conjunto de validación. Luego crea un archivo CSV de salida.

Ver la sección"Aprendizaje automático" para varios resultados de comparación de tiempo de ejecución.

Aprendizaje profundo multitrabajador con Keras para la predicción del CTR

Con los recientes avances en plataformas y aplicaciones de ML, ahora se presta mucha atención al aprendizaje a escala. La tasa de clics (CTR) se define como el número promedio de clics por cada cien impresiones de anuncios en línea (expresado como porcentaje). Se adopta ampliamente como una métrica clave en varios sectores industriales y casos de uso, incluidos el marketing digital, el comercio minorista, el comercio electrónico y los proveedores de servicios. Para obtener más detalles sobre las aplicaciones de CTR y los resultados del rendimiento del entrenamiento distribuido, consulte"Modelos de aprendizaje profundo para el rendimiento de la predicción de CTR" sección.

En este informe técnico utilizamos una variación del "Conjunto de datos de registros de clics de Criteo en terabytes" (ver TR-4904) para el aprendizaje profundo distribuido de múltiples trabajadores que utiliza Keras para crear un flujo de trabajo Spark con modelos de redes profundas y cruzadas (DCN), comparando su

desempeño en términos de función de error de pérdida de registro con un modelo de regresión logística Spark ML de referencia. DCN captura de manera eficiente interacciones de características efectivas de grados limitados, aprende interacciones altamente no lineales, no requiere ingeniería de características manual ni búsqueda exhaustiva y tiene un bajo costo computacional.

Los datos para los sistemas de recomendación a escala web son en su mayoría discretos y categóricos, lo que genera un espacio de características grande y escaso que dificulta la exploración de características. Esto ha limitado la mayoría de los sistemas a gran escala a modelos lineales como la regresión logística. Sin embargo, la clave para hacer buenas predicciones es identificar características frecuentemente predictivas y, al mismo tiempo, explorar características cruzadas poco comunes o no observadas. Los modelos lineales son simples, interpretables y fáciles de escalar, pero tienen un poder expresivo limitado.

Por otra parte, se ha demostrado que las características cruzadas son significativas para mejorar la expresividad de los modelos. Lamentablemente, a menudo se requiere ingeniería de características manual o una búsqueda exhaustiva para identificar dichas características. Generalizar a interacciones de características invisibles suele ser difícil. El uso de una red neuronal cruzada como DCN evita la ingeniería de características específicas de la tarea al aplicar explícitamente el cruce de características de manera automática. La red cruzada consta de múltiples capas, donde el mayor grado de interacciones está determinado probablemente por la profundidad de la capa. Cada capa produce interacciones de orden superior basadas en las existentes y conserva las interacciones de las capas anteriores.

Una red neuronal profunda (DNN) promete capturar interacciones muy complejas entre características. Sin embargo, en comparación con DCN, requiere casi un orden de magnitud más de parámetros, no puede formar características cruzadas de manera explícita y puede fallar en el aprendizaje eficiente de algunos tipos de interacciones de características. La red cruzada utiliza eficientemente la memoria y es fácil de implementar. El entrenamiento conjunto de los componentes cruzados y DNN captura de manera eficiente las interacciones de características predictivas y brinda un rendimiento de última generación en el conjunto de datos CTR de Criteo.

Un modelo DCN comienza con una capa de incrustación y apilamiento, seguida de una red cruzada y una red profunda en paralelo. A estas, a su vez, les sigue una capa de combinación final que combina las salidas de las dos redes. Los datos de entrada pueden ser un vector con características dispersas y densas. En Spark, las bibliotecas contienen el tipo `SparseVector` . Por lo tanto, es importante que los usuarios distingan entre ambos y tengan cuidado al llamar a sus respectivas funciones y métodos. En los sistemas de recomendación a escala web, como la predicción de CTR, las entradas son principalmente características categóricas, por ejemplo `'country=usa'` . Estas características suelen codificarse como vectores one-hot, por ejemplo, `'[0,1,0, …]'` . Codificación one-hot (OHE) con `SparseVector` es útil cuando se trabaja con conjuntos de datos del mundo real con vocabularios en constante cambio y crecimiento. Modificamos los ejemplos en "CTR profundo" para procesar vocabularios grandes, creando vectores de incrustación en la capa de incrustación y apilamiento de nuestro DCN.

El "Conjunto de datos de anuncios de display de Criteo" predice la tasa de clics de los anuncios. Tiene 13 características enteras y 26 características categóricas en las que cada categoría tiene una alta cardinalidad. Para este conjunto de datos, una mejora de 0,001 en la pérdida logarítmica es prácticamente significativa debido al gran tamaño de entrada. Una pequeña mejora en la precisión de la predicción para una gran base de usuarios puede conducir potencialmente a un gran aumento en los ingresos de una empresa. El conjunto de datos contiene 11 GB de registros de usuarios de un período de 7 días, lo que equivale a alrededor de 41 millones de registros. Usamos Spark `dataFrame.randomSplit()function` Dividir aleatoriamente los datos para entrenamiento (80%), validación cruzada (10%) y el 10% restante para pruebas.

DCN se implementó en TensorFlow con Keras. Hay cuatro componentes principales en la implementación del proceso de entrenamiento de modelos con DCN:

- **Procesamiento e incrustación de datos.** Las características de valor real se normalizan aplicando una transformación logarítmica. Para las características categóricas, integramos las características en vectores

densos de dimensión 6×(cardinalidad de categoría)1/4. La concatenación de todas las incrustaciones da como resultado un vector de dimensión 1026.

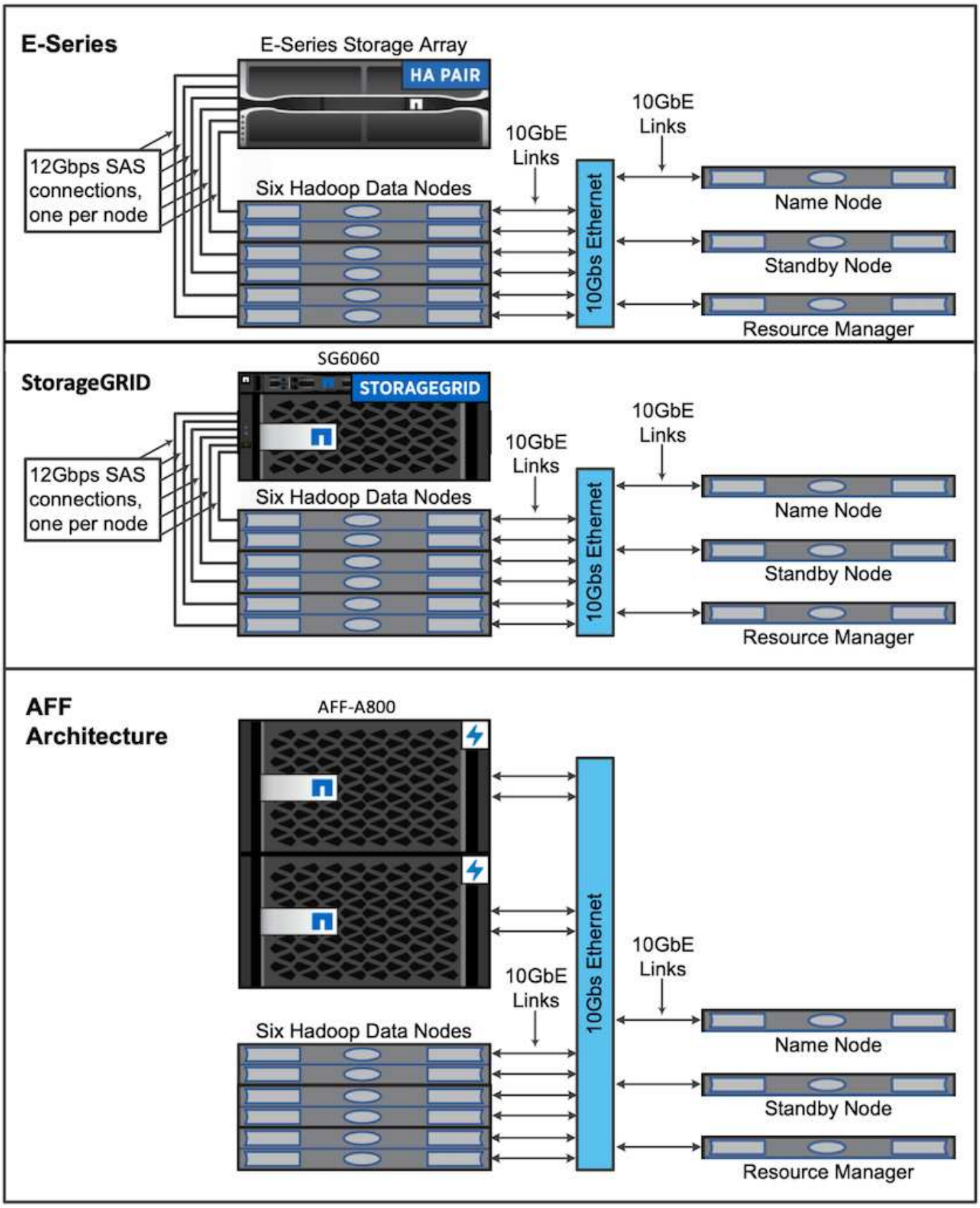
- **Mejoramiento.** Aplicamos optimización estocástica de minilotes con el optimizador Adam. El tamaño del lote se estableció en 512. Se aplicó la normalización por lotes a la red profunda y la norma de recorte de gradiente se estableció en 100.
- **Regularización.** Utilizamos la detención temprana, ya que no se encontró que la regularización o el abandono de L2 fueran efectivos.
- **Hiperparámetros.** Informamos los resultados basados en una búsqueda en cuadrícula sobre el número de capas ocultas, el tamaño de la capa oculta, la tasa de aprendizaje inicial y el número de capas cruzadas. El número de capas ocultas varió entre 2 y 5, con tamaños de capas ocultas que variaron entre 32 y 1024. Para DCN, el número de capas cruzadas fue de 1 a 6. La tasa de aprendizaje inicial se ajustó de 0,0001 a 0,001 con incrementos de 0,0001. Todos los experimentos se detuvieron anticipadamente en el paso de entrenamiento 150 000, más allá del cual comenzó a producirse un sobreajuste.

Además de DCN, también probamos otros modelos populares de aprendizaje profundo para la predicción de CTR, incluidos "DeepFM" , "AutoInt" , y "DCN v2" .

Arquitecturas utilizadas para la validación

Para esta validación, utilizamos cuatro nodos de trabajo y un nodo maestro con un par AFF-A800 HA. Todos los miembros del clúster estaban conectados a través de conmutadores de red 10GbE.

Para esta validación de la solución NetApp Spark, utilizamos tres controladores de almacenamiento diferentes: el E5760, el E5724 y el AFF-A800. Los controladores de almacenamiento de la Serie E se conectaron a cinco nodos de datos con conexiones SAS de 12 Gbps. El controlador de almacenamiento de par HA AFF proporciona volúmenes NFS exportados a través de conexiones de 10 GbE a nodos de trabajo de Hadoop. Los miembros del clúster Hadoop se conectaron a través de conexiones 10GbE en las soluciones Hadoop E-Series, AFF y StorageGRID .



Resultados de las pruebas

Utilizamos los scripts TeraSort y TeraValidate en la herramienta de evaluación

comparativa TeraGen para medir la validación del rendimiento de Spark con configuraciones E5760, E5724 y AFF-A800. Además, se probaron tres casos de uso principales: pipelines de Spark NLP y capacitación distribuida de TensorFlow, capacitación distribuida de Horovod y aprendizaje profundo de múltiples trabajadores usando Keras para predicción de CTR con DeepFM.

Para la validación de E-Series y StorageGRID , utilizamos el factor de replicación de Hadoop 2. Para la validación de AFF , solo utilizamos una fuente de datos.

La siguiente tabla enumera la configuración de hardware para la validación del rendimiento de Spark.

Tipo	Nodos de trabajo de Hadoop	Tipo de unidad	Unidades por nodo	Controlador de almacenamiento
SG6060	4	SAS	12	Par único de alta disponibilidad (HA)
E5760	4	SAS	60	Par de HA único
E5724	4	SAS	24	Par de HA único
AFF800	4	Unidad de estado sólido	6	Par de HA único

La siguiente tabla enumera los requisitos de software.

Software	Versión
RHEL	7,9
Entorno de ejecución de OpenJDK	1.8.0
Máquina virtual de servidor OpenJDK de 64 bits	25,302
Git	2.24.1
GCC/G++	11.2.1
Chispa	3.2.1
PySpark	3.1.2
SparkNLP	3.4.2
Flujo de tensor	2.9.0
Keras	2.9.0
Horovod	0.24.3

Análisis del sentimiento financiero

Nosotros publicamos"TR-4910: Análisis de sentimientos de las comunicaciones de los clientes con NetApp AI" , en el que se construyó una canalización de IA conversacional de extremo a extremo utilizando "Kit de herramientas DataOps de NetApp" , almacenamiento AFF y sistema NVIDIA DGX. El pipeline realiza procesamiento de señales de audio por lotes, reconocimiento automático de voz (ASR), aprendizaje por transferencia y análisis de sentimientos aprovechando el kit de herramientas DataOps. "SDK de NVIDIA Riva" , y el "Marco del Tao" . Al ampliar el caso de uso del análisis de sentimientos a la industria de servicios financieros, creamos un flujo de trabajo SparkNLP, cargamos tres modelos BERT para varias tareas de PNL,

como el reconocimiento de entidades nombradas, y obtuvimos sentimientos a nivel de oración para las llamadas de ganancias trimestrales de las 10 principales empresas del NASDAQ.

El siguiente script `sentiment_analysis_spark. py` utiliza el modelo FinBERT para procesar transcripciones en HDFS y producir recuentos de sentimientos positivos, neutrales y negativos, como se muestra en la siguiente tabla:

```
-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
```

La siguiente tabla enumera el análisis de sentimiento a nivel de oración, tras la presentación de resultados, de las 10 principales empresas del NASDAQ de 2016 a 2020.

Recuentos y porcentajes de sentimientos	**Las 10 empresas**	**AAPL**	**AMD**	**Amazon**	**Director Ejecutivo**	**GOOGL**	**INTC**	**MSFT**	**NVDA**
Recuentos positivos	7447	1567	743	290	682	826	824	904	417
Conteos neutrales	64067	6856	7596	5086	6650	5914	6099	5715	6189
Recuentos negativos	1787	253	213	84	189	97	282	202	89
Recuentos sin categorizar	196	0	0	76	0	0	0	1	0
(recuentos totales)	73497	8676	8552	5536	7521	6837	7205	6822	6695

En términos de porcentajes, la mayoría de las frases pronunciadas por los directores ejecutivos y directores

financieros son factuales y, por lo tanto, transmiten un sentimiento neutral. Durante una conferencia telefónica sobre ganancias, los analistas hacen preguntas que pueden transmitir un sentimiento positivo o negativo. Vale la pena investigar más a fondo cuantitativamente cómo el sentimiento negativo o positivo afecta los precios de las acciones el mismo día o el siguiente de negociación.

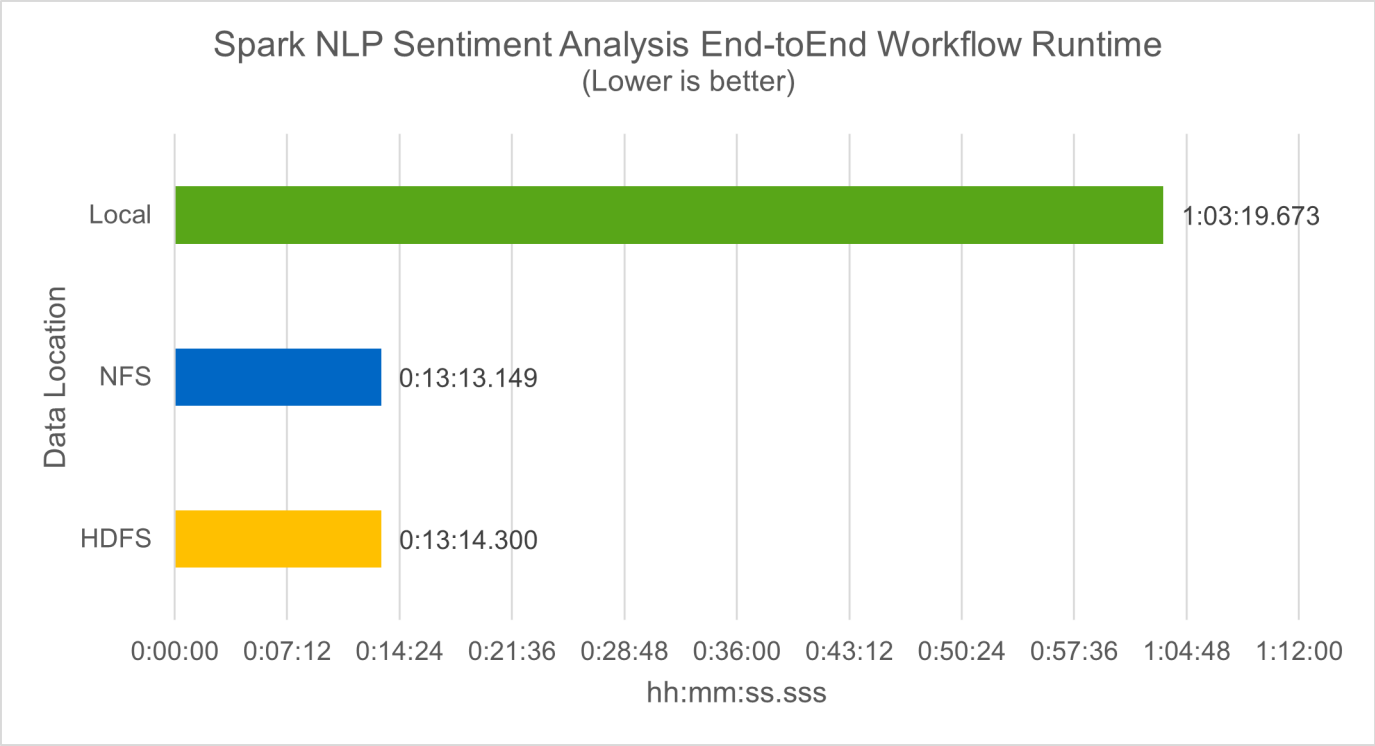
La siguiente tabla enumera el análisis de sentimiento a nivel de oración para las 10 principales empresas del NASDAQ, expresado en porcentaje.

Porcentaje de sentimiento	Las 10 empresas	AAPL	AMD	Amazon	Director Ejecutivo	GOOGL	INTC	MSFT	NVDA
Positivo	10.13%	18.06%	8.69%	5.24%	9.07%	12.08%	11.44%	13.25%	6.23%
Neutral	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%
Sin categorizar	0.27%	0%	0%	1.37%	0%	0%	0%	0.01%	0%

En términos del tiempo de ejecución del flujo de trabajo, vimos una mejora significativa de 4,78x `local` modo a un entorno distribuido en HDFS y una mejora adicional del 0,14 % al aprovechar 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 muestra la siguiente figura, el paralelismo de datos y modelos mejoró el procesamiento de datos y la velocidad de inferencia del modelo distribuido de TensorFlow. La ubicación de datos en NFS produjo un tiempo de ejecución ligeramente mejor porque el cuello de botella del flujo de trabajo es la descarga de modelos previamente entrenados. Si aumentamos el tamaño del conjunto de datos de transcripciones, la ventaja de NFS es más obvia.



Entrenamiento distribuido con rendimiento de Horovod

El siguiente comando produjo información de tiempo de ejecución y un archivo de registro en nuestro clúster Spark usando un solo `master` nodo con 160 ejecutores cada uno con un núcleo. La memoria del ejecutor se limitó a 5 GB para evitar errores de falta de memoria. Ver la sección"Scripts de Python para cada caso de uso principal" Para obtener más detalles sobre el procesamiento de datos, el entrenamiento del modelo y el cálculo de la precisión del modelo en `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
```

El tiempo de ejecución resultante con diez épocas de entrenamiento fue el siguiente:

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

Se necesitaron más de 43 minutos para procesar datos de entrada, entrenar un modelo DNN, calcular la precisión y producir puntos de control de TensorFlow y un archivo CSV para los resultados de la predicción. Limitamos el número de épocas de entrenamiento a 10, que en la práctica suele establecerse en 100 para garantizar una precisión satisfactoria del modelo. El tiempo de entrenamiento normalmente se escala linealmente con el número de épocas.

A continuación, utilizamos los cuatro nodos de trabajo disponibles en el clúster y ejecutamos el mismo script en `yarn` modo con datos en 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
```

El tiempo de ejecución resultante se mejoró de la siguiente manera:

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

Con el modelo de Horovod y el paralelismo de datos en Spark, vimos una aceleración del tiempo de ejecución de 5,29x `yarn` versus `local` Modo con diez épocas de entrenamiento. Esto se muestra en la siguiente figura con las leyendas. `HDFS` y `Local` . El entrenamiento del modelo DNN de TensorFlow subyacente se puede acelerar aún más con GPU si están disponibles. Planeamos realizar estas pruebas y publicar los resultados en un futuro informe técnico.

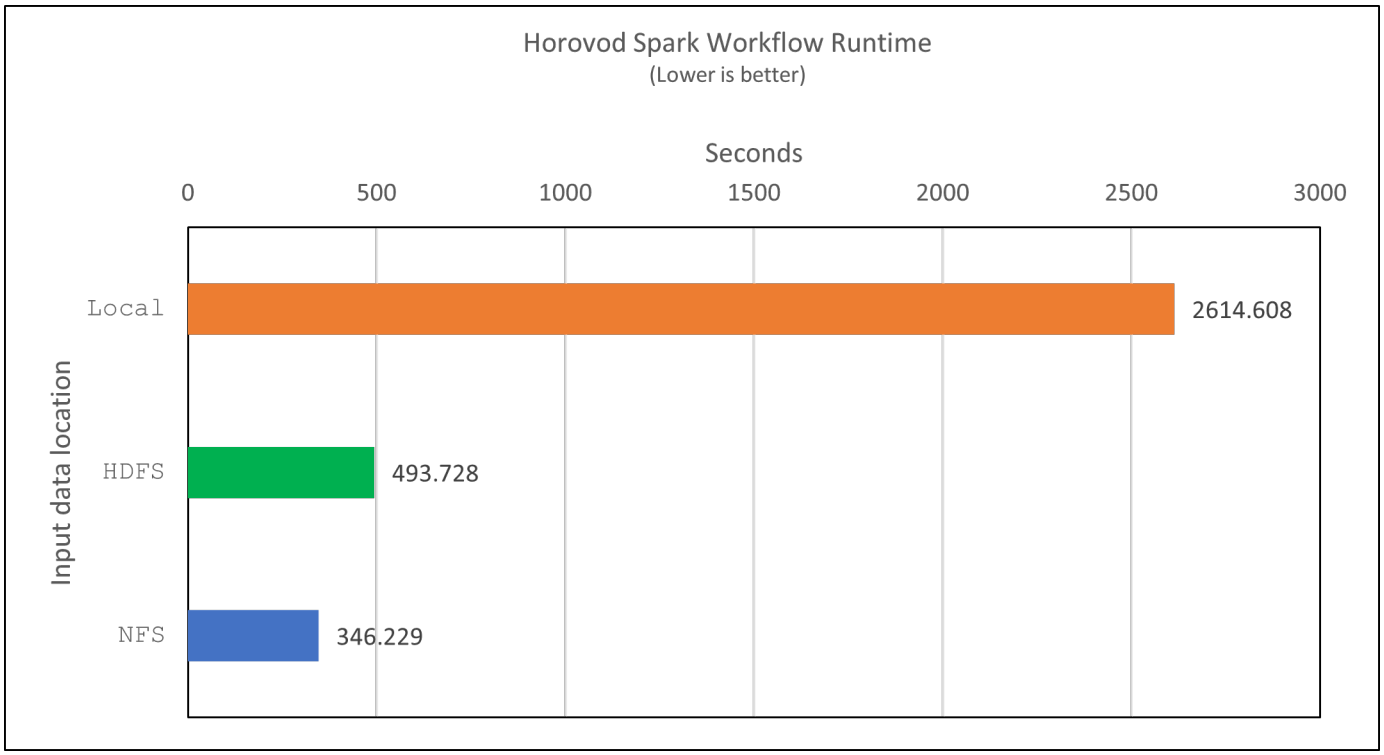
Nuestra siguiente prueba comparó los tiempos de ejecución con datos de entrada que residen en NFS versus HDFS. El volumen NFS en el AFF A800 se montó en `/sparkdemo/horovod` en los cinco nodos (uno maestro y cuatro trabajadores) de nuestro clúster Spark. Ejecutamos un comando similar al de las pruebas anteriores, con el `--data- dir` parámetro que ahora apunta al montaje 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
```

El tiempo de ejecución resultante con NFS fue el siguiente:

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

Hubo una aceleración adicional de 1,43x, como se muestra en la siguiente figura. Por lo tanto, con un almacenamiento all-flash de NetApp conectado a su clúster, los clientes disfrutan de los beneficios de la transferencia y distribución rápida de datos para los flujos de trabajo de Horovod Spark, logrando una aceleración de 7,55 veces en comparación con la ejecución en un solo nodo.



Modelos de aprendizaje profundo para el rendimiento de la predicción de CTR

Para los sistemas de recomendación diseñados para maximizar el CTR, es necesario aprender interacciones de características sofisticadas detrás de los comportamientos de los usuarios que se puedan calcular matemáticamente desde el orden bajo hasta el orden alto. Las interacciones de características de orden bajo y de orden alto deberían ser igualmente importantes para un buen modelo de aprendizaje profundo sin sesgarse hacia una u otra. Deep Factorization Machine (DeepFM), una red neuronal basada en máquinas de factorización, combina máquinas de factorización para recomendación y aprendizaje profundo para el aprendizaje de características en una nueva arquitectura de red neuronal.

Aunque las máquinas de factorización convencionales modelan interacciones de características por pares como un producto interno de vectores latentes entre características y teóricamente pueden capturar información de alto orden, en la práctica los profesionales del aprendizaje automático usualmente solo usan interacciones de características de segundo orden debido a la alta complejidad de cálculo y almacenamiento. Variantes de redes neuronales profundas como la de Google "Modelos anchos y profundos" Por otro lado, aprende interacciones de características sofisticadas en una estructura de red híbrida combinando un modelo lineal amplio y un modelo profundo.

Hay dos entradas para este modelo amplio y profundo: una para el modelo amplio subyacente y otra para el profundo; la última parte aún requiere ingeniería de características experta y, por lo tanto, hace que la técnica sea menos generalizable a otros dominios. A diferencia del modelo ancho y profundo, DeepFM se puede entrenar de manera eficiente con características sin procesar sin ninguna ingeniería de características porque su parte ancha y su parte profunda comparten la misma entrada y el vector de incrustación.

Primero procesamos el Criteo `train.txt` (11 GB) en un archivo CSV llamado `ctr_train.csv` almacenado en un montaje NFS `/sparkdemo/tr-4570-data` usando `run_classification_criteo_spark.py` de la sección"Scripts de Python para cada caso de uso principal." Dentro de este script, la función `process_input_file` Realiza varios métodos de cadena para eliminar tabulaciones e insertar `','` como delimitador y `'\n'` como nueva línea. Tenga en cuenta que solo necesita procesar el original. `train.txt` una vez, para que el bloque de código se muestre como comentarios.

Para las siguientes pruebas de diferentes modelos DL, utilizamos `ctr_train.csv` como archivo de entrada. En ejecuciones de prueba posteriores, el archivo CSV de entrada se leyó en un Spark DataFrame con un esquema que contenía un campo de `'label'` , características densas de números enteros `['I1', 'I2', 'I3', …, 'I13']` , y características dispersas `['C1', 'C2', 'C3', …, 'C26']` . La siguiente `spark-submit` El comando toma un CSV de entrada, entrena los modelos DeepFM con una división del 20 % para la validación cruzada y elige el mejor modelo después de diez épocas de entrenamiento para calcular la precisión de la predicción en el conjunto de prueba:

```
(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
```

Tenga en cuenta que dado que el archivo de datos `ctr_train.csv` Si tiene más de 11 GB, debe establecer un espacio suficiente `spark.driver.maxResultSize` mayor que el tamaño del conjunto de datos para evitar errores.

```
 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()
```

En lo anterior `SparkSession.builder` configuración que también habilitamos "Flecha apache" , que convierte un Spark DataFrame en un Pandas DataFrame con el `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.
```

Después de la división aleatoria, hay más de 36 millones de filas en el conjunto de datos de entrenamiento y 9 millones de muestras en el conjunto de prueba:

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

Debido a que este informe técnico se centra en las pruebas de CPU sin utilizar ninguna GPU, es imperativo que cree TensorFlow con los indicadores de compilador adecuados. Este paso evita invocar bibliotecas aceleradas por GPU y aprovecha al máximo las extensiones vectoriales avanzadas (AVX) y las instrucciones AVX2 de TensorFlow. Estas características están diseñadas para cálculos algebraicos lineales como suma vectorizada, multiplicaciones de matrices dentro de un entrenamiento DNN de propagación hacia adelante o hacia atrás. La instrucción FMA (Multiplicación y Suma Fusionada) disponible con AVX2 que utiliza registros de punto flotante (FP) de 256 bits es ideal para códigos enteros y tipos de datos, lo que da como resultado una aceleración de hasta 2x. Para los tipos de datos y códigos FP, AVX2 logra una aceleración del 8 % con respecto a 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 crear TensorFlow desde la fuente, NetApp recomienda usar "Bazel" . Para nuestro entorno, ejecutamos los siguientes comandos en el indicador de shell para instalar `dnf` , `dnf-plugins` y Bazel.

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

Debe habilitar GCC 5 o una versión más reciente para usar las características de C++17 durante el proceso de compilación, que proporciona RHEL con la Biblioteca de colecciones de software (SCL). Los siguientes comandos instalan `devtoolset` y GCC 11.2.1 en nuestro clúster 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
```

Tenga en cuenta que los dos últimos comandos habilitan `devtoolset-11` , que utiliza `/opt/rh/devtoolset-11/root/usr/bin/gcc` (CCG 11.2.1). Además, asegúrese de que su `git` La versión es mayor que 1.8.3 (viene con RHEL 7.9). Consulte esto "artículo" para actualizar `git` a 2.24.1.

Suponemos que ya ha clonado el último repositorio maestro de TensorFlow. Luego crea un `workspace` directorio con un `WORKSPACE` archivo para compilar TensorFlow desde la fuente con AVX, AVX2 y FMA. Ejecutar el `configure` archivo y especifique la ubicación binaria de Python correcta. "CUDA" está deshabilitado para nuestras pruebas porque no usamos una GPU. A `.bazelrc` El archivo se genera según su configuración. Además, editamos el archivo y lo configuramos. `build --define=no_hdfs_support=false` para habilitar la compatibilidad con HDFS. Referirse a `.bazelrc` en la sección"Scripts de Python para cada caso de uso principal," para obtener una lista completa de configuraciones y banderas.

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

Después de crear TensorFlow con los indicadores correctos, ejecute el siguiente script para procesar el

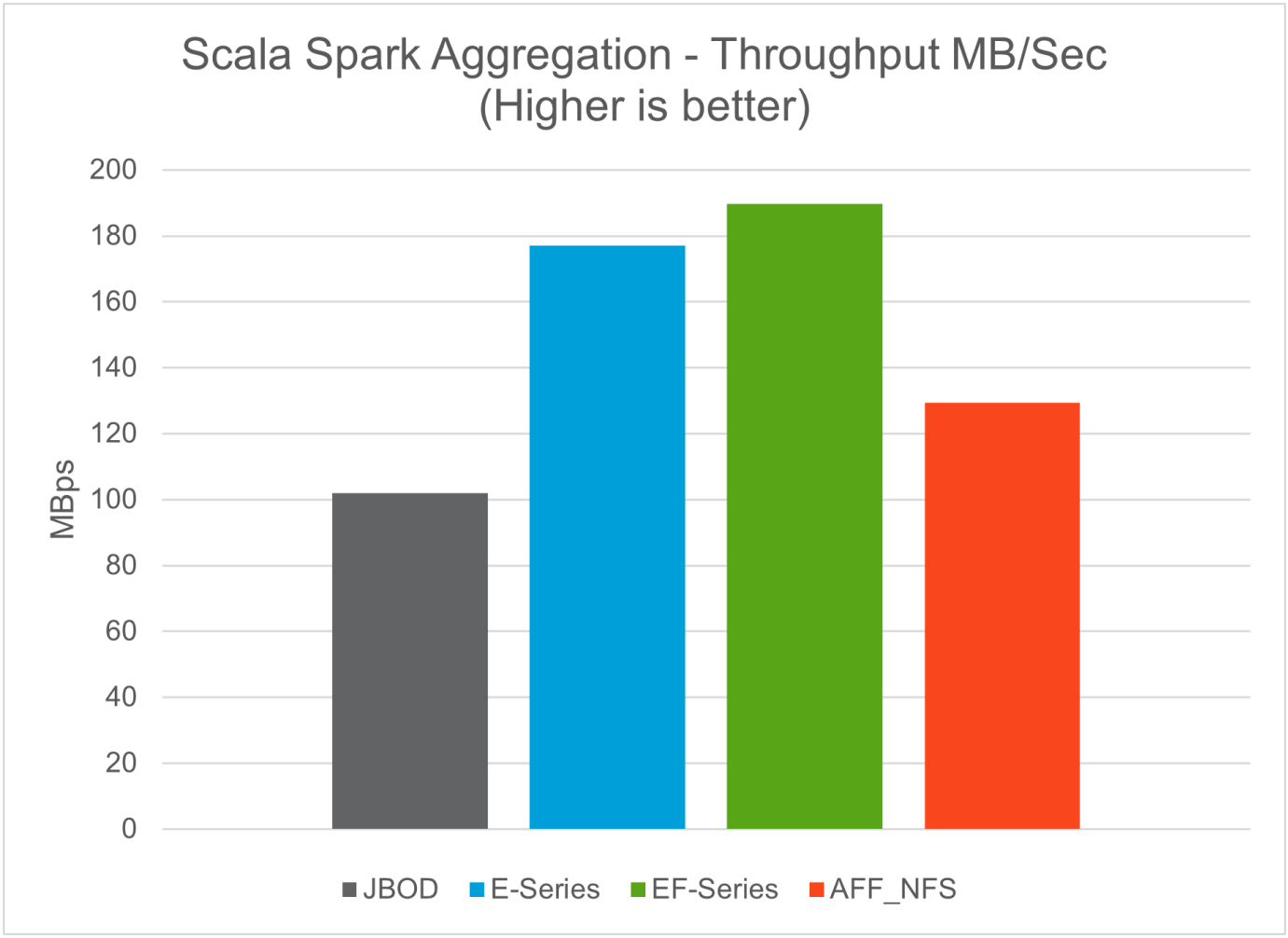
conjunto de datos de anuncios de Criteo Display, entrenar un modelo DeepFM y calcular el área bajo la curva característica operativa del receptor (ROC AUC) a partir de los puntajes de predicción.

```
(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
```

Después de diez épocas de entrenamiento, obtuvimos la puntuación AUC en el conjunto de datos de prueba:

```
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 manera similar a los casos de uso anteriores, comparamos el tiempo de ejecución del flujo de trabajo de Spark con datos que residen en diferentes ubicaciones. La siguiente figura muestra una comparación de la predicción de CTR de aprendizaje profundo para un tiempo de ejecución de flujos de trabajo de Spark.

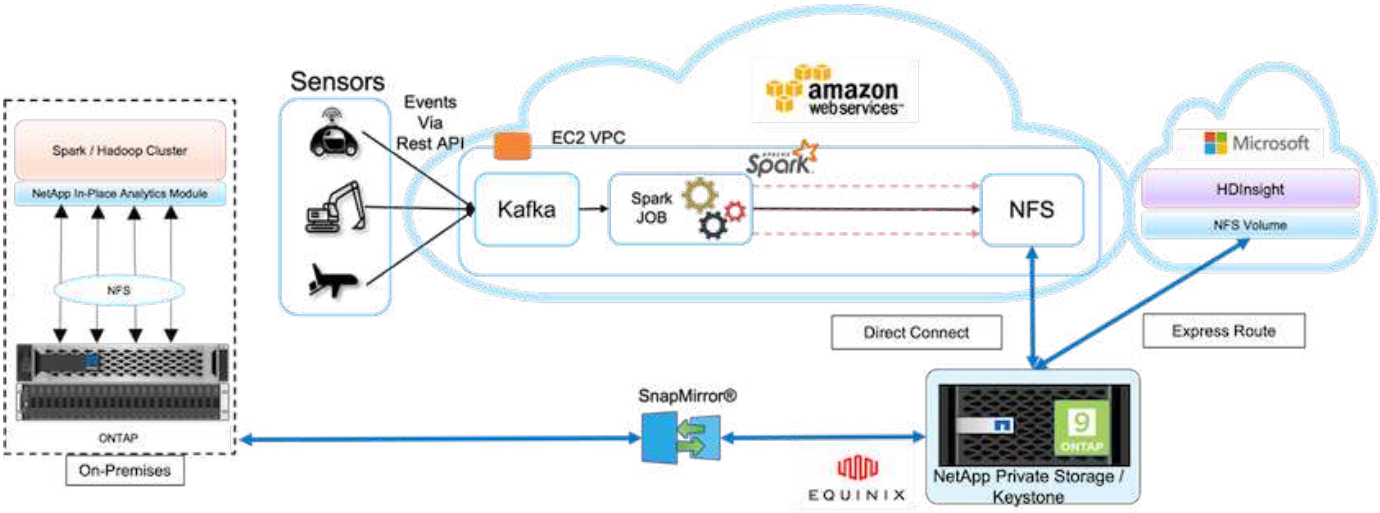


Solución de nube híbrida

Un centro de datos empresarial moderno es una nube híbrida que conecta múltiples entornos de infraestructura distribuida a través de un plano de gestión de datos continuo con un modelo operativo consistente, en las instalaciones y/o en múltiples nubes públicas. Para aprovechar al máximo una nube híbrida, debe poder mover datos sin problemas entre sus entornos locales y de múltiples nubes sin necesidad de realizar conversiones de datos ni refactorizar aplicaciones.

Los clientes han indicado que comienzan su viaje a la nube híbrida moviendo almacenamiento secundario a la nube para casos de uso como protección de datos o moviendo cargas de trabajo menos críticas para el negocio, como desarrollo de aplicaciones y DevOps a la nube. Luego pasan a cargas de trabajo más críticas. El alojamiento web y de contenido, el desarrollo de aplicaciones y DevOps, las bases de datos, los análisis y las aplicaciones en contenedores se encuentran entre las cargas de trabajo de nube híbrida más populares. La complejidad, el costo y los riesgos de los proyectos de IA empresarial han obstaculizado históricamente la adopción de IA desde la etapa experimental hasta la producción.

Con una solución de nube híbrida de NetApp , los clientes se benefician de herramientas integradas de seguridad, gobernanza de datos y cumplimiento con un único panel de control para la gestión de datos y flujo de trabajo en entornos distribuidos, al tiempo que optimizan el costo total de propiedad en función de su consumo. La siguiente figura es un ejemplo de solución de un socio de servicios en la nube encargado de proporcionar conectividad multi-nube para los datos de análisis de big data de los clientes.



En este escenario, los datos de IoT recibidos en AWS desde diferentes fuentes se almacenan en una ubicación central en NetApp Private Storage (NPS). El almacenamiento NPS está conectado a clústeres Spark o Hadoop ubicados en AWS y Azure, lo que permite que las aplicaciones de análisis de big data que se ejecutan en múltiples nubes accedan a los mismos datos. Los principales requisitos y desafíos para este caso de uso incluyen lo siguiente:

- Los clientes quieren ejecutar trabajos de análisis en los mismos datos utilizando múltiples nubes.
- Los datos deben recibirse de diferentes fuentes, como entornos locales y en la nube, a través de diferentes sensores y concentradores.
- La solución debe ser eficiente y rentable.
- El principal desafío es construir una solución rentable y eficiente que ofrezca servicios de análisis híbridos entre diferentes entornos locales y en la nube.

Nuestra solución de protección de datos y conectividad multicloud resuelve el problema de tener aplicaciones de análisis de nube en múltiples hiperescaladores. Como se muestra en la figura anterior, los datos de los sensores se transmiten y se incorporan al clúster de AWS Spark a través de Kafka. Los datos se almacenan en un recurso compartido NFS que reside en NPS, que se encuentra fuera del proveedor de la nube dentro de un centro de datos de Equinix.

Dado que NetApp NPS está conectado a Amazon AWS y Microsoft Azure a través de conexiones Direct Connect y Express Route respectivamente, los clientes pueden aprovechar el módulo de análisis local para acceder a los datos de los clústeres de análisis de Amazon y AWS. En consecuencia, dado que tanto el almacenamiento local como el NPS ejecutan el software ONTAP , "SnapMirror" Puede reflejar los datos de NPS en el clúster local, lo que proporciona análisis de nube híbrida en las instalaciones locales y en múltiples nubes.

Para obtener el mejor rendimiento, NetApp generalmente recomienda utilizar múltiples interfaces de red y conexiones directas o rutas expresas para acceder a los datos desde las instancias de la nube. Contamos con otras soluciones de transferencia de datos, incluidas: "XCP" y "Copia y sincronización de BlueXP" para ayudar a los clientes a construir clústeres Spark de nube híbrida que sean rentables, seguros y conscientes de las aplicaciones.

Scripts de Python para cada caso de uso principal

Los siguientes tres scripts de Python corresponden a los tres casos de uso principales probados. Primero es `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}")
```

El segundo guión es `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()
```

El tercer guión es `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))
```

Conclusión

En este documento, analizamos la arquitectura de Apache Spark, los casos de uso de los clientes y la cartera de almacenamiento de NetApp en relación con el big data, el análisis moderno y la inteligencia artificial, el aprendizaje automático y el aprendizaje automático. En nuestras pruebas de validación de rendimiento basadas en herramientas de evaluación comparativa estándar de la industria y la demanda de los clientes, las soluciones NetApp Spark demostraron un rendimiento superior en relación con los sistemas Hadoop nativos. Una combinación de los casos de uso de clientes y los resultados de rendimiento presentados en este informe pueden ayudarlo a elegir una solución Spark adecuada para su implementación.

Dónde encontrar información adicional

En este TR se utilizaron las siguientes referencias:

- Arquitectura y componentes de Apache Spark

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

- Casos de uso de Apache Spark

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

- Spark PNL

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

- BERT

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

- Red profunda y cruzada para predicciones de clics en anuncios

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

- FlexGroup

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

- ETL de transmisión

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

- Soluciones NetApp E-Series para Hadoop

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

- Soluciones de análisis de datos modernos de NetApp

  "Soluciones de análisis de datos"

- 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

- Copia y sincronización de BlueXP

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

- Kit de herramientas DataOps

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