NetApp Storage-Lösungen für Apache Spark

NetApp artificial intelligence solutions

NetApp
February 12, 2026

Inhalt

NetApp Storage-Lösungen für Apache Spark

TR-4570: NetApp -Speicherlösungen für Apache Spark: Architektur, Anwendungsfälle und Leistungsergebnisse

Rick Huang, Karthikeyan Nagalingam, NetApp

Der Schwerpunkt dieses Dokuments liegt auf der Apache Spark-Architektur, Anwendungsfällen von Kunden und dem NetApp -Speicherportfolio im Zusammenhang mit Big Data-Analysen und künstlicher Intelligenz (KI). Darüber hinaus werden verschiedene Testergebnisse präsentiert, bei denen branchenübliche KI-, Machine-Learning- (ML) und Deep-Learning- (DL) Tools mit einem typischen Hadoop-System verglichen wurden, sodass Sie die geeignete Spark-Lösung auswählen können. Zu Beginn benötigen Sie eine Spark-Architektur, entsprechende Komponenten und zwei Bereitstellungsmodi (Cluster und Client).

Dieses Dokument enthält außerdem Anwendungsfälle von Kunden zur Lösung von Konfigurationsproblemen und bietet einen Überblick über das NetApp -Speicherportfolio, das für Big Data-Analysen sowie KI, ML und DL mit Spark relevant ist. Abschließend präsentieren wir Testergebnisse aus Spark-spezifischen Anwendungsfällen und dem NetApp Spark-Lösungsportfolio.

Kundenherausforderungen

Dieser Abschnitt konzentriert sich auf die Herausforderungen für Kunden im Zusammenhang mit Big Data Analytics und KI/ML/DL in Datenwachstumsbranchen wie Einzelhandel, digitales Marketing, Bankwesen, diskrete Fertigung, Prozessfertigung, Behörden und professionelle Dienstleistungen.

Unvorhersehbare Leistung

Bei herkömmlichen Hadoop-Bereitstellungen wird normalerweise Standardhardware verwendet. Um die Leistung zu verbessern, müssen Sie das Netzwerk, das Betriebssystem, den Hadoop-Cluster, Ökosystemkomponenten wie Spark und die Hardware optimieren. Selbst wenn Sie jede Ebene optimieren, kann es schwierig sein, das gewünschte Leistungsniveau zu erreichen, da Hadoop auf Standardhardware ausgeführt wird, die nicht für hohe Leistung in Ihrer Umgebung ausgelegt ist.

Medien- und Knotenfehler

Selbst unter normalen Bedingungen ist Standardhardware anfällig für Ausfälle. Wenn eine Festplatte auf einem Datenknoten ausfällt, betrachtet der Hadoop-Master diesen Knoten standardmäßig als fehlerhaft. Anschließend werden bestimmte Daten von diesem Knoten über das Netzwerk von Replikaten auf einen fehlerfreien Knoten kopiert. Dieser Prozess verlangsamt die Netzwerkpakete für alle Hadoop-Jobs. Der Cluster muss die Daten dann erneut zurückkopieren und die überreplizierten Daten entfernen, wenn der fehlerhafte Knoten wieder in einen fehlerfreien Zustand zurückkehrt.

Hadoop-Anbieterbindung

Hadoop-Distributoren verfügen über ihre eigene Hadoop-Distribution mit eigener Versionierung, wodurch der Kunde an diese Distributionen gebunden ist. Viele Kunden benötigen jedoch Unterstützung für In-Memory-Analysen, die den Kunden nicht an bestimmte Hadoop-Distributionen bindet. Sie müssen die Freiheit haben, die Verteilung zu ändern und trotzdem ihre Analysen mitzunehmen.

Fehlende Unterstützung für mehr als eine Sprache

Kunden benötigen zur Ausführung ihrer Aufgaben häufig zusätzlich zu MapReduce-Java-Programmen Unterstützung für mehrere Sprachen. Optionen wie SQL und Skripte bieten mehr Flexibilität beim Erhalten von Antworten, mehr Optionen zum Organisieren und Abrufen von Daten und schnellere Möglichkeiten zum Verschieben von Daten in ein Analyseframework.

Schwierigkeit der Verwendung

Seit einiger Zeit beschweren sich Leute, dass Hadoop schwierig zu verwenden sei. Obwohl Hadoop mit jeder neuen Version einfacher und leistungsfähiger geworden ist, hält sich diese Kritik hartnäckig. Hadoop erfordert, dass Sie die Programmiermuster von Java und MapReduce verstehen, was für Datenbankadministratoren und Personen mit herkömmlichen Skriptkenntnissen eine Herausforderung darstellt.

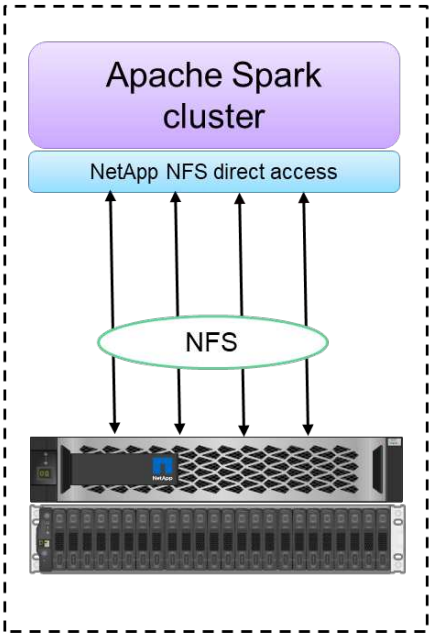
Komplizierte Frameworks und Tools

KI-Teams in Unternehmen stehen vor zahlreichen Herausforderungen. Selbst mit Expertenwissen im Bereich Data Science lassen sich Tools und Frameworks für unterschiedliche Bereitstellungsökosysteme und Anwendungen möglicherweise nicht einfach von einem zum anderen übertragen. Eine Data-Science-Plattform sollte sich nahtlos in entsprechende Big-Data-Plattformen integrieren lassen, die auf Spark basieren, und dabei einfache Datenbewegungen, wiederverwendbare Modelle, sofort einsatzbereiten Code und Tools bieten, die Best Practices für das Prototyping, Validieren, Versionieren, Teilen, Wiederverwenden und schnelle Bereitstellen von Modellen in der Produktion unterstützen.

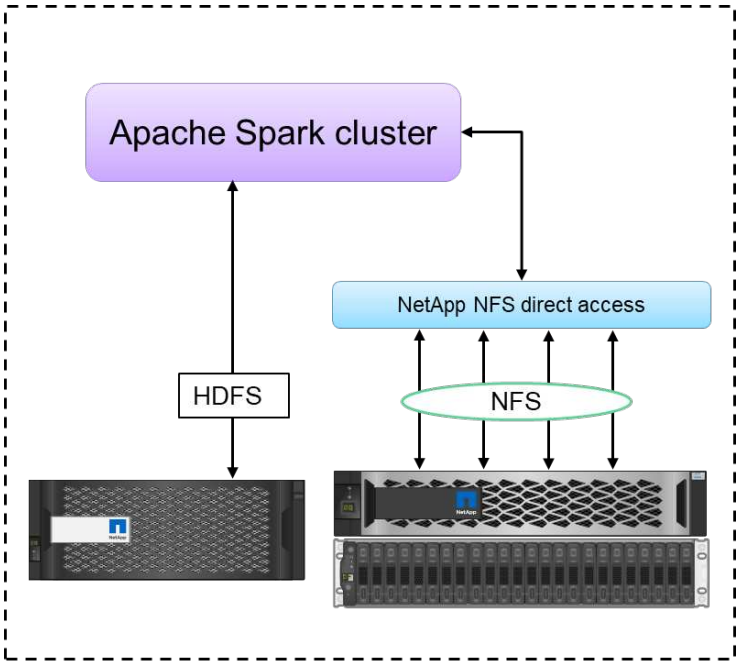
Warum NetApp?

NetApp kann Ihr Spark-Erlebnis auf folgende Weise verbessern:

- Der direkte NetApp NFS-Zugriff (siehe Abbildung unten) ermöglicht es Kunden, Big-Data-Analysejobs auf ihren vorhandenen oder neuen NFSv3- oder NFSv4-Daten auszuführen, ohne die Daten zu verschieben oder zu kopieren. Es verhindert mehrere Kopien der Daten und macht die Synchronisierung der Daten mit einer Quelle überflüssig.
- Effizientere Speicherung und weniger Serverreplikation. Beispielsweise erfordert die NetApp E-Series Hadoop-Lösung zwei statt drei Replikate der Daten, und die FAS Hadoop-Lösung erfordert eine Datenquelle, jedoch keine Replikation oder Kopien der Daten. NetApp -Speicherlösungen erzeugen außerdem weniger Server-zu-Server-Verkehr.
- Besseres Verhalten von Hadoop-Jobs und Clustern bei Laufwerk- und Knotenausfällen.
- Bessere Datenaufnahmeleistung.



Configuration 1: NFS as primary storage



Configuration 2: HDFS and NFS in single Spark cluster

Im Finanz- und Gesundheitssektor beispielsweise muss die Datenübertragung von einem Ort zum anderen gesetzlichen Verpflichtungen entsprechen, was keine leichte Aufgabe ist. In diesem Szenario analysiert der NetApp NFS-Direktzugriff die Finanz- und Gesundheitsdaten von ihrem ursprünglichen Speicherort aus. Ein weiterer wichtiger Vorteil besteht darin, dass die Verwendung des direkten NetApp NFS-Zugriffs den Schutz von Hadoop-Daten durch die Verwendung nativer Hadoop-Befehle vereinfacht und Datenschutz-Workflows mit dem umfangreichen Datenverwaltungsportfolio von NetApp ermöglicht.

Der direkte NetApp NFS-Zugriff bietet zwei Arten von Bereitstellungsoptionen für Hadoop/Spark-Cluster:

- Standardmäßig verwenden Hadoop- oder Spark-Cluster das Hadoop Distributed File System (HDFS) zur Datenspeicherung und als Standarddateisystem. Der direkte NetApp NFS-Zugriff kann das Standard-HDFS durch NFS-Speicher als Standarddateisystem ersetzen und so eine direkte Analyse von NFS-Daten ermöglichen.
- In einer weiteren Bereitstellungsoption unterstützt der direkte NetApp NFS-Zugriff die Konfiguration von NFS als zusätzlichen Speicher zusammen mit HDFS in einem einzelnen Hadoop- oder Spark-Cluster. In diesem Fall kann der Kunde Daten über NFS-Exporte freigeben und zusammen mit HDFS-Daten vom selben Cluster aus darauf zugreifen.

Zu den wichtigsten Vorteilen des NetApp NFS-Direktzugriffs zählen die folgenden:

- Analysieren der Daten von ihrem aktuellen Standort aus, wodurch die zeit- und leistungsintensive Aufgabe des Verschiebens von Analysedaten in eine Hadoop-Infrastruktur wie HDFS vermieden wird.
- Reduzierung der Anzahl der Replikate von drei auf eins.
- Ermöglicht Benutzern, Rechenleistung und Speicher zu entkoppeln, um sie unabhängig voneinander zu skalieren.
- Bietet Unternehmensdatenschutz durch Nutzung der umfassenden Datenverwaltungsfunktionen von ONTAP.
- Zertifizierung mit der Hortonworks-Datenplattform.
- Ermöglicht die Bereitstellung hybrider Datenanalysen.

- Verkürzung der Sicherungszeit durch Nutzung der dynamischen Multithread-Funktion.

Sehen"TR-4657: NetApp Hybrid Cloud-Datenlösungen – Spark und Hadoop basierend auf Kundenanwendungsfällen" zum Sichern von Hadoop-Daten, zur Sicherung und Notfallwiederherstellung von der Cloud vor Ort, zum Aktivieren von DevTest auf vorhandenen Hadoop-Daten, zum Datenschutz und zur Multicloud-Konnektivität sowie zum Beschleunigen von Analyse-Workloads.

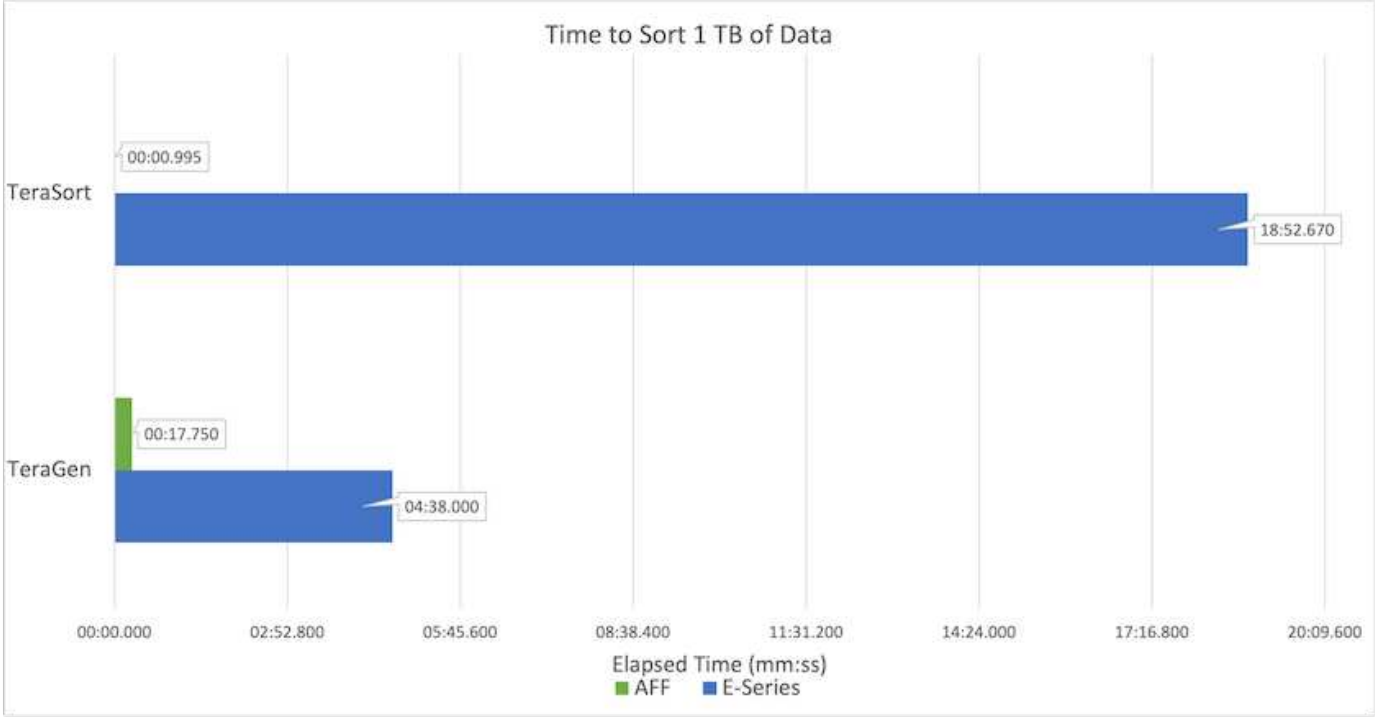
In den folgenden Abschnitten werden Speicherfunktionen beschrieben, die für Spark-Kunden wichtig sind.

Speicher-Tiering

Mit Hadoop Storage Tiering können Sie Dateien gemäß einer Speicherrichtlinie in verschiedenen Speichertypen speichern. Zu den Speichertypen gehören `hot` , `cold` , `warm` , `all_ssd` , `one_ssd` , Und `lazy_persist` .

Wir haben die Validierung der Hadoop-Speicherschichtung auf einem NetApp AFF Speichercontroller und einem E-Series-Speichercontroller mit SSD- und SAS-Laufwerken mit unterschiedlichen Speicherrichtlinien durchgeführt. Der Spark-Cluster mit AFF-A800 verfügt über vier Compute-Worker-Knoten, während der Cluster mit E-Series acht hat. Dabei geht es hauptsächlich darum, die Leistung von Solid-State-Laufwerken (SSDs) mit der von Festplatten (HDDs) zu vergleichen.

Die folgende Abbildung zeigt die Leistung von NetApp -Lösungen für eine Hadoop-SSD.



- Die NL-SAS-Basiskonfiguration verwendete acht Rechenknoten und 96 NL-SAS-Laufwerke. Diese Konfiguration generierte 1 TB Daten in 4 Minuten und 38 Sekunden. Sehen "TR-3969 NetApp E-Series-Lösung für Hadoop" für Details zur Cluster- und Speicherkonfiguration.
- Mit TeraGen generierte die SSD-Konfiguration 1 TB Daten 15,66-mal schneller als die NL-SAS-Konfiguration. Darüber hinaus verwendete die SSD-Konfiguration nur die Hälfte der Rechenknoten und die Hälfte der Festplattenlaufwerke (insgesamt 24 SSD-Laufwerke). Basierend auf der Zeit, die für die Auftragserledigung benötigt wurde, war es fast doppelt so schnell wie die NL-SAS-Konfiguration.
- Mit TeraSort sortierte die SSD-Konfiguration 1 TB Daten 1138,36-mal schneller als die NL-SAS-Konfiguration. Darüber hinaus verwendete die SSD-Konfiguration nur die Hälfte der Rechenknoten und die

Hälfte der Festplattenlaufwerke (insgesamt 24 SSD-Laufwerke). Daher war es pro Laufwerk ungefähr dreimal schneller als die NL-SAS-Konfiguration.

- Das Fazit ist, dass die Umstellung von rotierenden Festplatten auf reine Flash-Speicher die Leistung verbessert. Die Anzahl der Rechenknoten war nicht der Engpass. Mit dem All-Flash-Speicher von NetApp lässt sich die Laufzeitleistung gut skalieren.
- Mit NFS waren die Daten funktional gleichbedeutend mit einer gemeinsamen Bündelung, wodurch die Anzahl der Rechenknoten je nach Arbeitslast reduziert werden kann. Die Benutzer des Apache Spark-Clusters müssen die Daten nicht manuell neu ausbalancieren, wenn sie die Anzahl der Compute-Knoten ändern.

Leistungsskalierung – Scale-Out

Wenn Sie mehr Rechenleistung von einem Hadoop-Cluster in einer AFF Lösung benötigen, können Sie Datenknoten mit einer entsprechenden Anzahl von Speichercontrollern hinzufügen. NetApp empfiehlt, mit vier Datenknoten pro Speichercontroller-Array zu beginnen und die Anzahl je nach Arbeitslastmerkmalen auf acht Datenknoten pro Speichercontroller zu erhöhen.

AFF und FAS eignen sich perfekt für In-Place-Analysen. Basierend auf den Rechenanforderungen können Sie Knotenmanager hinzufügen und unterbrechungsfreie Vorgänge ermöglichen Ihnen, bei Bedarf und ohne Ausfallzeiten einen Speichercontroller hinzuzufügen. Wir bieten umfangreiche Funktionen mit AFF und FAS, wie z. B. NVME-Medienunterstützung, garantierte Effizienz, Datenreduzierung, QOS, prädiktive Analysen, Cloud-Tiering, Replikation, Cloud-Bereitstellung und Sicherheit. Um Kunden bei der Erfüllung ihrer Anforderungen zu unterstützen, bietet NetApp Funktionen wie Dateisystemanalyse, Kontingente und On-Box-Lastausgleich ohne zusätzliche Lizenzkosten. NetApp bietet eine bessere Leistung bei der Anzahl gleichzeitiger Jobs, eine geringere Latenz, einfachere Vorgänge und einen höheren Durchsatz in Gigabyte pro Sekunde als unsere Wettbewerber. Darüber hinaus läuft NetApp Cloud Volumes ONTAP auf allen drei großen Cloud-Anbietern.

Leistungsskalierung – Hochskalieren

Mithilfe der Scale-up-Funktionen können Sie Festplattenlaufwerke zu AFF, FAS und E-Series-Systemen hinzufügen, wenn Sie zusätzliche Speicherkapazität benötigen. Mit Cloud Volumes ONTAP ist die Skalierung des Speichers auf PB-Ebene eine Kombination aus zwei Faktoren: der Auslagerung selten verwendeter Daten aus dem Blockspeicher in den Objektspeicher und dem Stapeln von Cloud Volumes ONTAP -Lizenzen ohne zusätzliche Rechenleistung.

Mehrere Protokolle

NetApp -Systeme unterstützen die meisten Protokolle für Hadoop-Bereitstellungen, einschließlich SAS, iSCSI, FCP, InfiniBand und NFS.

Operative und unterstützte Lösungen

Die in diesem Dokument beschriebenen Hadoop-Lösungen werden von NetApp unterstützt. Diese Lösungen sind auch bei den wichtigsten Hadoop-Distributoren zertifiziert. Weitere Informationen finden Sie im "Hortonworks" Site und die Cloudera "Zertifizierung" Und "Partner" Websites.

Zielgruppe

Die Welt der Analytik und Datenwissenschaft berührt mehrere Disziplinen in IT und Wirtschaft:

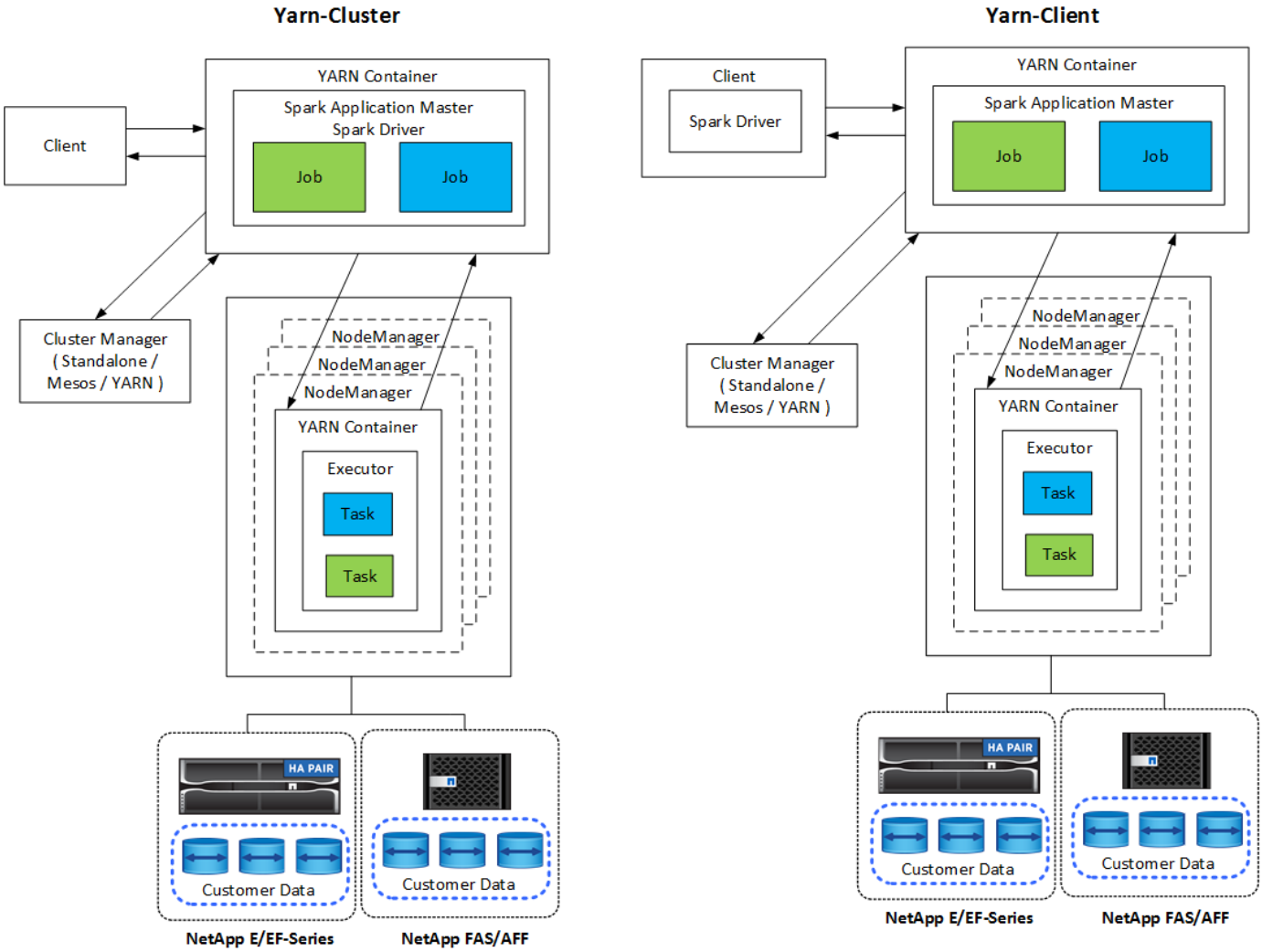
- Der Datenwissenschaftler benötigt die Flexibilität, die Tools und Bibliotheken seiner Wahl zu verwenden.
- Der Dateningenieur muss wissen, wie die Daten fließen und wo sie sich befinden.
- Ein DevOps-Ingenieur benötigt die Tools, um neue KI- und ML-Anwendungen in seine CI- und CD-Pipelines zu integrieren.
- Cloud-Administratoren und -Architekten müssen in der Lage sein, Hybrid-Cloud-Ressourcen einzurichten und zu verwalten.
- Geschäftsanwender möchten Zugriff auf Analyse-, KI-, ML- und DL-Anwendungen haben.

In diesem technischen Bericht beschreiben wir, wie NetApp AFF, E-Series, StorageGRID, NFS-Direktzugriff, Apache Spark, Horovod und Keras jeder dieser Rollen dabei helfen, einen Mehrwert für das Unternehmen zu schaffen.

Lösungstechnologie

Apache Spark ist ein beliebtes Programmierframework zum Schreiben von Hadoop-Anwendungen, das direkt mit dem Hadoop Distributed File System (HDFS) arbeitet. Spark ist produktionsbereit, unterstützt die Verarbeitung von Streaming-Daten und ist schneller als MapReduce. Spark verfügt über konfigurierbares In-Memory-Datencaching für effiziente Iteration und die Spark-Shell ist interaktiv zum Lernen und Erkunden von Daten. Mit Spark können Sie Anwendungen in Python, Scala oder Java erstellen. Spark-Anwendungen bestehen aus einem oder mehreren Jobs, die eine oder mehrere Aufgaben haben.

Jede Spark-Anwendung verfügt über einen Spark-Treiber. Im YARN-Client-Modus wird der Treiber lokal auf dem Client ausgeführt. Im YARN-Cluster-Modus läuft der Treiber im Cluster auf dem Anwendungsmaster. Im Clustermodus wird die Anwendung auch dann weiter ausgeführt, wenn die Verbindung zum Client getrennt wird.



Es gibt drei Cluster-Manager:

- **Eigenständig.** Dieser Manager ist Teil von Spark, wodurch die Einrichtung eines Clusters vereinfacht wird.
- **Apache Mesos.** Dies ist ein allgemeiner Cluster-Manager, der auch MapReduce und andere Anwendungen ausführt.
- **Hadoop YARN.** Dies ist ein Ressourcenmanager in Hadoop 3.

Der widerstandsfähige verteilte Datensatz (RDD) ist die Hauptkomponente von Spark. RDD erstellt die verlorenen und fehlenden Daten aus den im Speicher des Clusters gespeicherten Daten neu und speichert die ursprünglichen Daten, die aus einer Datei stammen oder programmgesteuert erstellt werden. RDDs werden aus Dateien, Daten im Speicher oder einem anderen RDD erstellt. Die Spark-Programmierung führt zwei Vorgänge aus: Transformation und Aktionen. Durch die Transformation wird ein neues RDD basierend auf einem vorhandenen erstellt. Aktionen geben einen Wert aus einem RDD zurück.

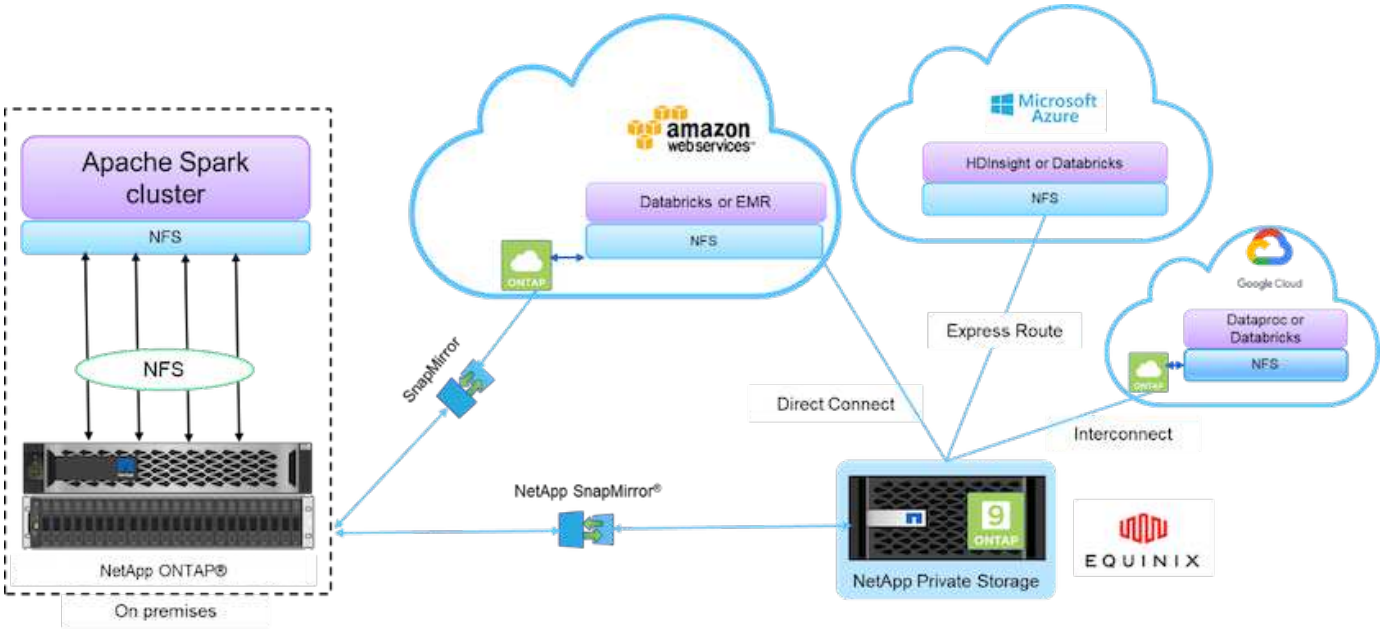
Transformationen und Aktionen gelten auch für Spark-Datasets und DataFrames. Ein Dataset ist eine verteilte Datensammlung, die die Vorteile von RDDs (starke Typisierung, Verwendung von Lambda-Funktionen) mit den Vorteilen der optimierten Ausführungs-Engine von Spark SQL kombiniert. Ein Datensatz kann aus JVM-Objekten erstellt und dann mithilfe funktionaler Transformationen (Map, FlatMap, Filter usw.) bearbeitet werden. Ein DataFrame ist ein in benannte Spalten organisierter Datensatz. Es ist konzeptionell gleichwertig mit einer Tabelle in einer relationalen Datenbank oder einem Datenrahmen in R/Python. DataFrames können aus einer Vielzahl von Quellen erstellt werden, beispielsweise strukturierten Datendateien, Tabellen in Hive/HBase, externen Datenbanken vor Ort oder in der Cloud oder vorhandenen RDDs.

Spark-Anwendungen umfassen einen oder mehrere Spark-Jobs. Jobs führen Aufgaben in Executoren aus und Executoren werden in YARN-Containern ausgeführt. Jeder Executor wird in einem einzelnen Container ausgeführt und Executoren existieren während der gesamten Lebensdauer einer Anwendung. Ein Executor wird nach dem Start der Anwendung fixiert und YARN ändert die Größe des bereits zugewiesenen Containers nicht. Ein Executor kann Aufgaben gleichzeitig auf Daten im Arbeitsspeicher ausführen.

Übersicht über die NetApp Spark-Lösungen

NetApp verfügt über drei Speicherportfolios: FAS/ AFF, E-Serie und Cloud Volumes ONTAP. Wir haben AFF und die E-Serie mit ONTAP Speichersystem für Hadoop-Lösungen mit Apache Spark validiert.

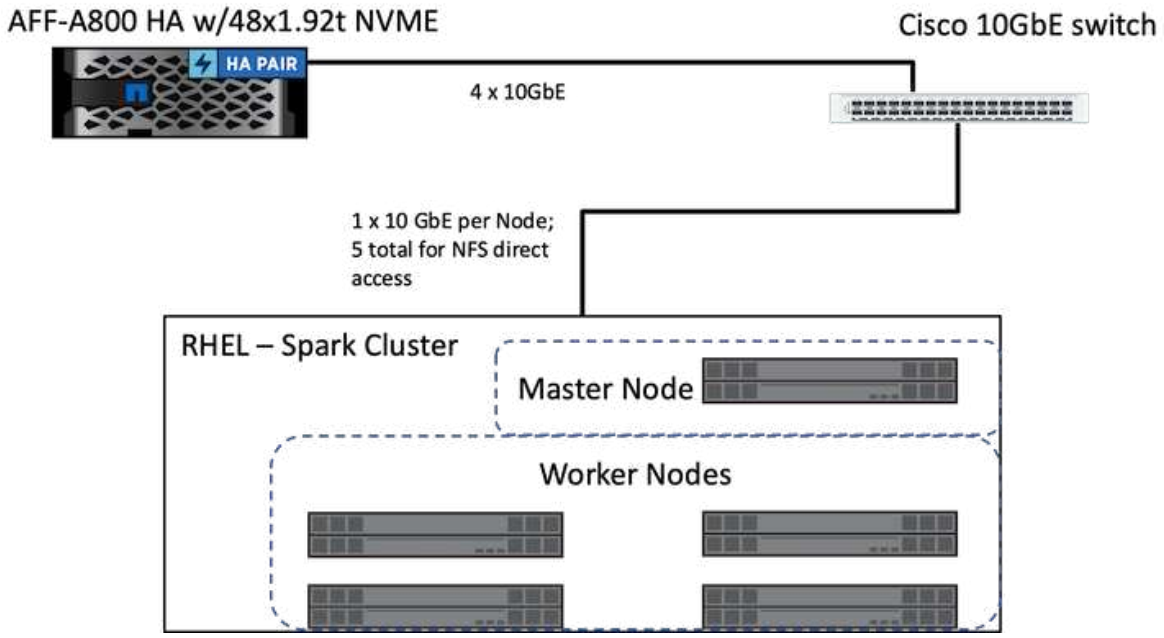
Das von NetApp betriebene Datengewebe integriert Datenverwaltungsdienste und Anwendungen (Bausteine) für Datenzugriff, -kontrolle, -schutz und -sicherheit, wie in der folgenden Abbildung dargestellt.



Zu den Bausteinen in der obigen Abbildung gehören:

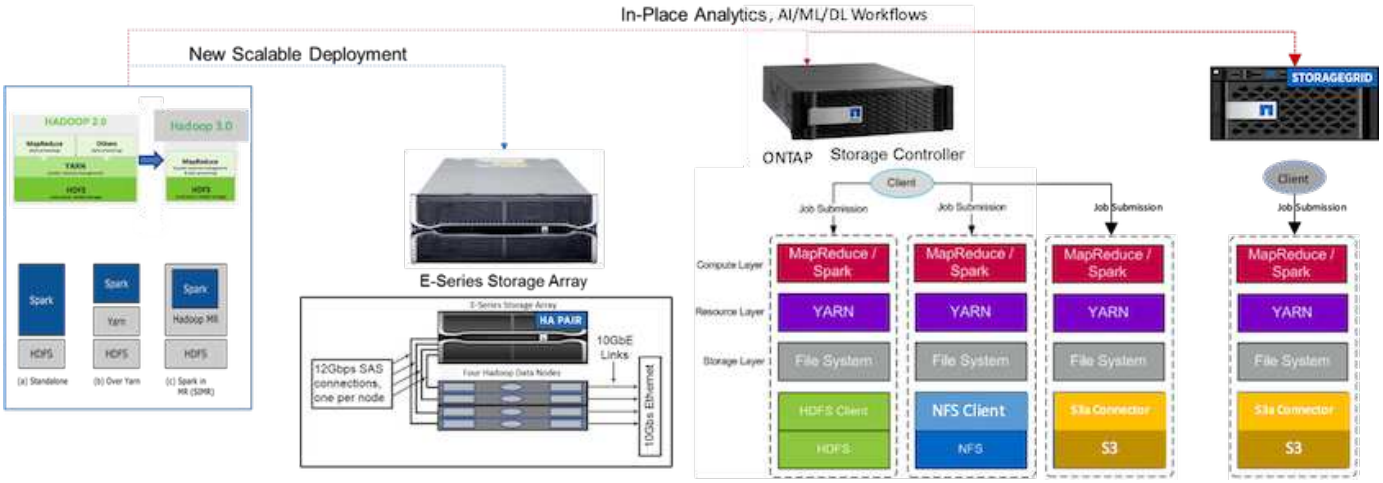
- * NetApp NFS-Direktzugriff.* Bietet den neuesten Hadoop- und Spark-Clustern direkten Zugriff auf NetApp NFS-Volumes ohne zusätzliche Software- oder Treiberanforderungen.
- * NetApp Cloud Volumes ONTAP und Google Cloud NetApp Volumes.* Softwaredefinierter verbundener Speicher basierend auf ONTAP , der in Amazon Web Services (AWS) oder Azure NetApp Files (ANF) in Microsoft Azure-Clouddiensten ausgeführt wird.
- * NetApp SnapMirror Technologie.* Bietet Datenschutzfunktionen zwischen lokalen und ONTAP Cloud- oder NPS-Instanzen.
- **Cloud-Dienstanbieter.** Zu diesen Anbietern gehören AWS, Microsoft Azure, Google Cloud und IBM Cloud.
- **PaaS.** Cloudbasierte Analysedienste wie Amazon Elastic MapReduce (EMR) und Databricks in AWS sowie Microsoft Azure HDInsight und Azure Databricks.

Die folgende Abbildung zeigt die Spark-Lösung mit NetApp Speicher.

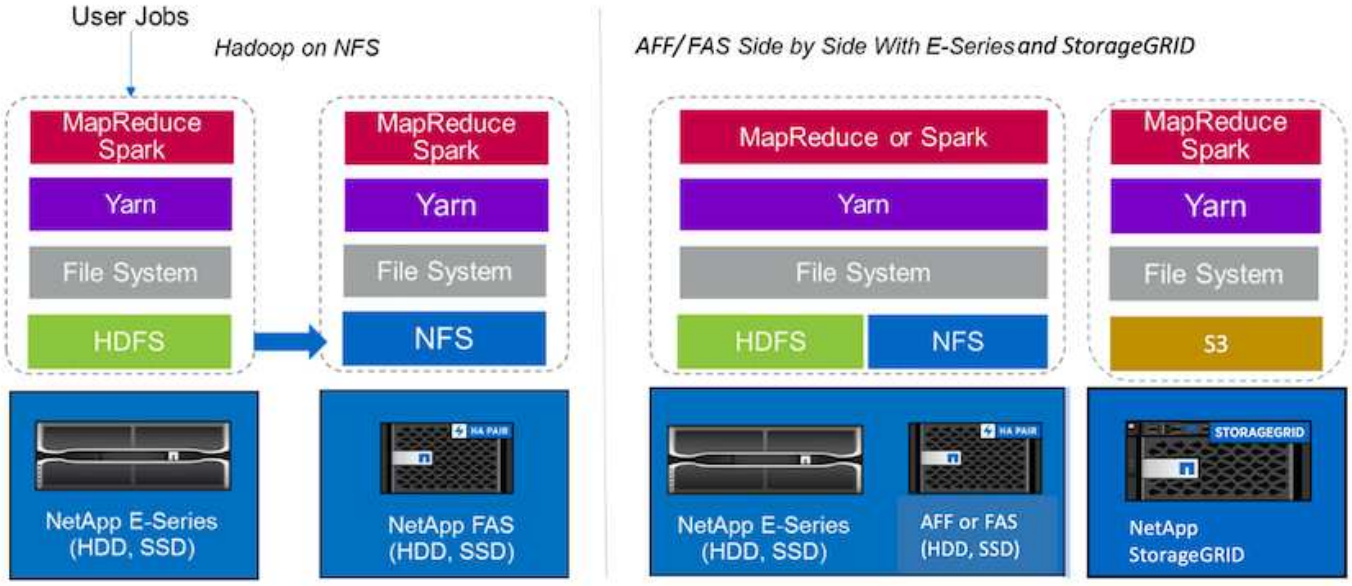


Die ONTAP Spark-Lösung verwendet das NetApp NFS-Direktzugriffsprotokoll für In-Place-Analysen und KI-, ML- und DL-Workflows unter Zugriff auf vorhandene Produktionsdaten. Für Hadoop-Knoten verfügbare Produktionsdaten werden exportiert, um vor Ort analytische sowie KI-, ML- und DL-Jobs auszuführen. Sie können auf die zu verarbeitenden Daten in Hadoop-Knoten entweder mit oder ohne direkten NetApp NFS-Zugriff zugreifen. In Spark mit dem Standalone- oder `yarn` Cluster-Manager können Sie ein NFS-Volume konfigurieren, indem Sie `file://<target_volume>` . Wir haben drei Anwendungsfälle mit unterschiedlichen Datensätzen validiert. Die Einzelheiten dieser Validierungen werden im Abschnitt „Testergebnisse“ vorgestellt. (xref)

Die folgende Abbildung zeigt die Speicherpositionierung von NetApp Apache Spark/Hadoop.



Wir haben die einzigartigen Funktionen der E-Series Spark-Lösung, der AFF/ FAS ONTAP Spark-Lösung und der StorageGRID Spark-Lösung identifiziert und detaillierte Validierungen und Tests durchgeführt. Basierend auf unseren Beobachtungen empfiehlt NetApp die E-Series-Lösung für Greenfield-Installationen und neue skalierbare Bereitstellungen sowie die AFF/ FAS -Lösung für In-Place-Analysen, KI-, ML- und DL-Workloads unter Verwendung vorhandener NFS-Daten und StorageGRID für KI-, ML- und DL- und moderne Datenanalysen, wenn Objektspeicher erforderlich ist.



Ein Data Lake ist ein Speicherrepository für große Datensätze in nativer Form, das für Analyse-, KI-, ML- und DL-Aufgaben verwendet werden kann. Wir haben ein Data Lake-Repository für die Spark-Lösungen E-Series, AFF/ FAS und StorageGRID SG6060 erstellt. Das E-Series-System bietet HDFS-Zugriff auf den Hadoop Spark-Cluster, während auf vorhandene Produktionsdaten über das NFS-Direktzugriffsprotokoll auf den Hadoop-Cluster zugegriffen wird. Für Datensätze, die sich im Objektspeicher befinden, bietet NetApp StorageGRID sicheren S3- und S3a-Zugriff.

Zusammenfassung des Anwendungsfalls

Auf dieser Seite werden die verschiedenen Bereiche beschrieben, in denen diese Lösung eingesetzt werden kann.

Streaming-Daten

Apache Spark kann Streaming-Daten verarbeiten, die für Streaming-Extract-, Transform- und Load-Prozesse (ETL), Datenanreicherung, Auslösen von Ereigniserkennung und komplexe Sitzungsanalysen verwendet werden:

- **Streaming ETL.** Daten werden kontinuierlich bereinigt und aggregiert, bevor sie in Datenspeicher übertragen werden. Netflix verwendet Kafka- und Spark-Streaming, um eine Echtzeit-Lösung für Online-Filmempfehlungen und Datenüberwachung zu erstellen, die täglich Milliarden von Ereignissen aus verschiedenen Datenquellen verarbeiten kann. Traditionelles ETL für die Stapelverarbeitung wird jedoch anders behandelt. Diese Daten werden zuerst gelesen und dann in ein Datenbankformat konvertiert, bevor sie in die Datenbank geschrieben werden.
- **Datenanreicherung.** Spark-Streaming reichert die Live-Daten mit statischen Daten an, um eine Datenanalyse in Echtzeit zu ermöglichen. Beispielsweise können Online-Werbetreibende personalisierte, zielgerichtete Anzeigen schalten, die auf Informationen zum Kundenverhalten basieren.
- **Ereigniserkennung auslösen.** Mit Spark-Streaming können Sie ungewöhnliches Verhalten, das auf potenziell schwerwiegende Probleme hinweisen könnte, schnell erkennen und darauf reagieren. Finanzinstitute verwenden beispielsweise Trigger, um betrügerische Transaktionen zu erkennen und zu stoppen, und Krankenhäuser verwenden Trigger, um gefährliche gesundheitliche Veränderungen anhand der Vitalfunktionen eines Patienten zu erkennen.
- **Komplexe Sitzungsanalyse.** Spark Streaming sammelt Ereignisse wie Benutzeraktivitäten nach der

Anmeldung bei einer Website oder Anwendung, die dann gruppiert und analysiert werden. Netflix nutzt diese Funktion beispielsweise, um Filmempfehlungen in Echtzeit bereitzustellen.

Weitere Informationen zur Konfiguration von Streaming-Daten, zur Confluent Kafka-Verifizierung und zu Leistungstests finden Sie unter"TR-4912: Best Practice-Richtlinien für Confluent Kafka Tiered Storage mit NetApp" .

Maschinelles Lernen

Das integrierte Spark-Framework unterstützt Sie beim Ausführen wiederholter Abfragen von Datensätzen mithilfe der Machine Learning-Bibliothek (MLlib). MLlib wird in Bereichen wie Clustering, Klassifizierung und Dimensionsreduktion für einige gängige Big-Data-Funktionen wie Predictive Intelligence, Kundensegmentierung für Marketingzwecke und Stimmungsanalyse verwendet. MLlib wird in der Netzwerksicherheit verwendet, um Datenpakete in Echtzeit auf Anzeichen böswilliger Aktivitäten zu überprüfen. Es hilft Sicherheitsanbietern, sich über neue Bedrohungen zu informieren, Hackern immer einen Schritt voraus zu sein und gleichzeitig ihre Kunden in Echtzeit zu schützen.

Tiefes Lernen

TensorFlow ist ein beliebtes Deep-Learning-Framework, das in der gesamten Branche verwendet wird. TensorFlow unterstützt das verteilte Training auf einem CPU- oder GPU-Cluster. Dieses verteilte Training ermöglicht es Benutzern, es auf einer großen Datenmenge mit vielen tiefen Schichten auszuführen.

Wenn wir TensorFlow mit Apache Spark verwenden wollten, mussten wir bis vor Kurzem alle erforderlichen ETL-Prozesse für TensorFlow in PySpark durchführen und dann die Daten in den Zwischenspeicher schreiben. Diese Daten würden dann für den eigentlichen Trainingsprozess in den TensorFlow-Cluster geladen. Dieser Workflow erforderte, dass der Benutzer zwei verschiedene Cluster verwaltete, einen für ETL und einen für das verteilte Training von TensorFlow. Das Ausführen und Warten mehrerer Cluster war normalerweise mühsam und zeitaufwändig.

DataFrames und RDD in früheren Spark-Versionen waren für Deep Learning nicht gut geeignet, da der wahlfreie Zugriff eingeschränkt war. In Spark 3.0 mit Project Hydrogen wird native Unterstützung für die Deep-Learning-Frameworks hinzugefügt. Dieser Ansatz ermöglicht eine nicht auf MapReduce basierende Planung auf dem Spark-Cluster.

Interaktive Analyse

Apache Spark ist schnell genug, um explorative Abfragen ohne Sampling mit anderen Entwicklungssprachen als Spark durchzuführen, darunter SQL, R und Python. Spark verwendet Visualisierungstools, um komplexe Daten zu verarbeiten und interaktiv zu visualisieren. Spark mit strukturiertem Streaming führt interaktive Abfragen für Livedaten in der Webanalyse durch, die es Ihnen ermöglichen, interaktive Abfragen für die aktuelle Sitzung eines Webbesuchers auszuführen.

Empfehlungssystem

Im Laufe der Jahre haben Empfehlungssysteme enorme Veränderungen in unser Leben gebracht, da Unternehmen und Verbraucher auf dramatische Veränderungen beim Online-Shopping, der Online-Unterhaltung und vielen anderen Branchen reagiert haben. Tatsächlich gehören diese Systeme zu den offensichtlichsten Erfolgsgeschichten der KI in der Produktion. In vielen praktischen Anwendungsfällen werden Empfehlungssysteme mit Konversations-KI oder Chatbots kombiniert, die mit einem NLP-Backend verbunden sind, um relevante Informationen zu erhalten und nützliche Schlussfolgerungen zu ziehen.

Heutzutage setzen viele Einzelhändler auf neuere Geschäftsmodelle wie Online-Kauf und Abholung im Geschäft, Abholung am Straßenrand, Self-Checkout, Scan-and-Go und mehr. Diese Modelle haben während

der COVID-19-Pandemie an Bedeutung gewonnen, da sie das Einkaufen für die Verbraucher sicherer und bequemer machen. KI ist für diese wachsenden digitalen Trends von entscheidender Bedeutung, die vom Verbraucherverhalten beeinflusst werden und umgekehrt. Um den wachsenden Ansprüchen der Verbraucher gerecht zu werden, das Kundenerlebnis zu verbessern, die Betriebseffizienz zu steigern und den Umsatz zu steigern, unterstützt NetApp seine Unternehmenskunden und Unternehmen dabei, mithilfe von Algorithmen für maschinelles Lernen und Deep Learning schnellere und präzisere Empfehlungssysteme zu entwickeln.

Es gibt mehrere gängige Techniken zum Bereitstellen von Empfehlungen, darunter kollaboratives Filtern, inhaltsbasierte Systeme, das Deep Learning Recommender Model (DLRM) und Hybridtechniken. Kunden nutzten PySpark zuvor, um kollaboratives Filtern zur Erstellung von Empfehlungssystemen zu implementieren. Spark MLlib implementiert Alternating Least Squares (ALS) für kollaboratives Filtern, einen in Unternehmen vor dem Aufkommen von DLRM sehr beliebten Algorithmus.

Verarbeitung natürlicher Sprache

Konversations-KI, die durch die Verarbeitung natürlicher Sprache (NLP) ermöglicht wird, ist der Zweig der KI, der Computern bei der Kommunikation mit Menschen hilft. NLP ist in allen Branchen und vielen Anwendungsfällen weit verbreitet, von intelligenten Assistenten und Chatbots bis hin zur Google-Suche und Textvorhersage. Laut einer "Gartner" Prognosen zufolge werden bis 2022 70 % der Menschen täglich mit Konversations-KI-Plattformen interagieren. Für eine qualitativ hochwertige Konversation zwischen Mensch und Maschine müssen die Antworten schnell, intelligent und natürlich klingen.

Kunden benötigen große Datenmengen, um ihre NLP- und automatischen Spracherkennungsmodelle (ASR) zu verarbeiten und zu trainieren. Sie müssen außerdem Daten zwischen Edge, Core und Cloud verschieben und benötigen die Fähigkeit, in Millisekunden Schlussfolgerungen zu ziehen, um eine natürliche Kommunikation mit Menschen herzustellen. NetApp AI und Apache Spark sind eine ideale Kombination für Computing, Speicherung, Datenverarbeitung, Modelltraining, Feinabstimmung und Bereitstellung.

Die Stimmungsanalyse ist ein Forschungsgebiet innerhalb der NLP, in dem positive, negative oder neutrale Stimmungen aus Texten extrahiert werden. Die Sentimentanalyse bietet vielfältige Anwendungsfälle, von der Ermittlung der Leistung von Supportcenter-Mitarbeitern in Gesprächen mit Anrufern bis hin zur Bereitstellung geeigneter automatisierter Chatbot-Antworten. Es wurde auch verwendet, um den Aktienkurs eines Unternehmens auf der Grundlage der Interaktionen zwischen Unternehmensvertretern und dem Publikum bei vierteljährlichen Telefonkonferenzen zu den Unternehmensergebnissen vorherzusagen. Darüber hinaus kann mithilfe der Stimmungsanalyse die Meinung eines Kunden zu den Produkten, Dienstleistungen oder dem Support der Marke ermittelt werden.

Wir nutzten die "Spark NLP" Bibliothek von "John Snow Labs" zum Laden vortrainierter Pipelines und Bidirectional Encoder Representations from Transformers (BERT)-Modelle, einschließlich "Stimmung in den Finanznachrichten" Und "FinBERT" , Durchführung von Tokenisierung, Named Entity Recognition, Modelltraining, Anpassung und Stimmungsanalyse im großen Maßstab. Spark NLP ist die einzige Open-Source-NLP-Bibliothek in der Produktion, die hochmoderne Transformatoren wie BERT, ALBERT, ELECTRA, XLNet, DistilBERT, RoBERTa, DeBERTa, XLM-RoBERTa, Longformer, ELMO, Universal Sentence Encoder, Google T5, MarianMT und GPT2 bietet. Die Bibliothek funktioniert nicht nur in Python und R, sondern auch im JVM-Ökosystem (Java, Scala und Kotlin) im großen Maßstab, indem sie Apache Spark nativ erweitert.

Wichtige Anwendungsfälle und Architekturen für KI, ML und DL

Die wichtigsten Anwendungsfälle und Methoden für KI, ML und DL können in die folgenden Abschnitte unterteilt werden:

Spark NLP-Pipelines und verteilte TensorFlow-Inferenz

Die folgende Liste enthält die beliebtesten Open-Source-NLP-Bibliotheken, die von der Data-Science-Community in unterschiedlichen Entwicklungsstufen übernommen wurden:

- "Toolkit für natürliche Sprache (NLTK)" . Das komplette Toolkit für alle NLP-Techniken. Es wird seit Anfang der 2000er Jahre gepflegt.
- "TextBlob" . Eine benutzerfreundliche Python-API für NLP-Tools, die auf NLTK und Pattern basiert.
- "Stanford Core NLP" . NLP-Dienste und -Pakete in Java, entwickelt von der Stanford NLP Group.
- "Gensim" . „Topic Modelling for Humans“ begann als Sammlung von Python-Skripten für das Projekt „Czech Digital Mathematics Library“.
- "SpaCy" . End-to-End-NLP-Workflows für die Industrie mit Python und Cython mit GPU-Beschleunigung für Transformatoren.
- "Fasttext" . Eine kostenlose, leichtgewichtige Open-Source-NLP-Bibliothek zum Lernen von Wort-Embeddings und zur Satzklassifizierung, die vom AI Research (FAIR)-Labor von Facebook erstellt wurde.

Spark NLP ist eine einzige, einheitliche Lösung für alle NLP-Aufgaben und -Anforderungen, die skalierbare, leistungsstarke und hochpräzise NLP-basierte Software für echte Produktionsanwendungsfälle ermöglicht. Es nutzt Transferlernen und implementiert die neuesten hochmodernen Algorithmen und Modelle in der Forschung und branchenübergreifend. Aufgrund der fehlenden vollständigen Unterstützung durch Spark für die oben genannten Bibliotheken wurde Spark NLP auf Basis von "Spark ML" um die Vorteile der universellen verteilten In-Memory-Datenverarbeitungs-Engine von Spark als NLP-Bibliothek der Enterprise-Klasse für unternehmenskritische Produktionsabläufe zu nutzen. Seine Annotatoren nutzen regelbasierte Algorithmen, maschinelles Lernen und TensorFlow, um Deep-Learning-Implementierungen zu unterstützen. Dies umfasst gängige NLP-Aufgaben, einschließlich, aber nicht beschränkt auf Tokenisierung, Lemmatisierung, Stemming, Part-of-Speech-Tagging, Named-Entity-Erkennung, Rechtschreibprüfung und Stimmungsanalyse.

Bidirectional Encoder Representations from Transformers (BERT) ist eine transformerbasierte maschinelle Lerntechnik für NLP. Es machte das Konzept des Vortrainings und der Feinabstimmung populär. Die Transformer-Architektur in BERT stammt aus der maschinellen Übersetzung, die langfristige Abhängigkeiten besser modelliert als auf rekurrenten neuronalen Netzwerken (RNN) basierende Sprachmodelle. Außerdem wurde die Masked Language Modelling (MLM)-Aufgabe eingeführt, bei der zufällig 15 % aller Token maskiert werden und das Modell sie vorhersagt, wodurch echte Bidirektionalität ermöglicht wird.

Aufgrund der Fachsprache und des Mangels an gekennzeichneten Daten in diesem Bereich ist die Analyse der Finanzstimmung eine Herausforderung. FinBERT, ein Sprachmodell basierend auf vortrainiertem BERT, wurde domänenangepasst auf "Reuters TRC2" , ein Finanzkorpus, und mit gekennzeichneten Daten fein abgestimmt ("Finanzielle PhraseBank") zur Klassifizierung der Finanzstimmung. Forscher extrahierten 4.500 Sätze aus Nachrichtenartikeln mit Finanzbegriffen. Anschließend bewerteten 16 Experten und Masterstudenten mit Finanzhintergrund die Sätze als positiv, neutral und negativ. Wir haben einen End-to-End-Spark-Workflow erstellt, um die Stimmung für die Transkripte der Telefonkonferenzen zu den Top-10-Gewinnzahlen der NASDAQ-Unternehmen von 2016 bis 2020 mithilfe von FinBERT und zwei weiteren vortrainierten Pipelines zu analysieren. "Dokument DL erklären") von Spark NLP.

Die zugrunde liegende Deep-Learning-Engine für Spark NLP ist TensorFlow, eine durchgängige Open-Source-Plattform für maschinelles Lernen, die eine einfache Modellerstellung, eine robuste ML-Produktion überall und leistungsstarke Experimente für die Forschung ermöglicht. Daher, wenn wir unsere Pipelines in Spark ausführen `yarn cluster` Im Modus führten wir im Wesentlichen verteiltes TensorFlow mit Daten- und Modellparallelisierung über einen Master- und mehrere Worker-Knoten sowie über einen auf dem Cluster montierten Netzwerkspeicher aus.

Horovod verteiltes Training

Die zentrale Hadoop-Validierung für die MapReduce-bezogene Leistung wird mit TeraGen, TeraSort, TeraValidate und DFSIO (Lesen und Schreiben) durchgeführt. Die Validierungsergebnisse von TeraGen und TeraSort werden in "NetApp E-Series-Lösung für Hadoop" und im Abschnitt „Storage Tiering“ für AFF.

Aufgrund von Kundenanfragen betrachten wir das verteilte Training mit Spark als einen der wichtigsten der verschiedenen Anwendungsfälle. In diesem Dokument haben wir die "Hovorod auf Spark" um die Spark-Leistung mit lokalen, Cloud-nativen und Hybrid-Cloud-Lösungen von NetApp unter Verwendung von NetApp All Flash FAS (AFF)-Speichercontrollern, Azure NetApp Files und StorageGRID zu validieren.

Das Horovod on Spark-Paket bietet einen praktischen Wrapper um Horovod, der die Ausführung verteilter Trainings-Workloads in Spark-Clustern vereinfacht und eine enge Modelldesignschleife ermöglicht, in der Datenverarbeitung, Modelltraining und Modellbewertung alle in Spark erfolgen, wo sich die Trainings- und Inferenzdaten befinden.

Es gibt zwei APIs zum Ausführen von Horovod auf Spark: eine Estimator-API auf hoher Ebene und eine Run-API auf niedrigerer Ebene. Obwohl beide denselben zugrunde liegenden Mechanismus zum Starten von Horovod auf Spark-Executoren verwenden, abstrahiert die Estimator-API die Datenverarbeitung, die Modelltrainingsschleife, die Modellprüfpunkte, die Metrikerfassung und das verteilte Training. Wir verwendeten Horovod Spark Estimators, TensorFlow und Keras für eine End-to-End-Datenaufbereitung und einen verteilten Trainings-Workflow basierend auf dem "Kaggle Rossmann Store Sales" Wettbewerb.

Das Drehbuch `keras_spark_horovod_rossmann_estimator.py` finden Sie im Abschnitt"Python-Skripte für jeden wichtigen Anwendungsfall." Es besteht aus drei Teilen:

- Der erste Teil führt verschiedene Schritte zur Datenvorverarbeitung für einen ersten Satz von CSV-Dateien durch, die von Kaggle bereitgestellt und von der Community gesammelt wurden. Die Eingabedaten werden in einen Trainingssatz mit einem `Validation` Teilmenge und ein Testdatensatz.
- Der zweite Teil definiert ein Keras Deep Neural Network (DNN)-Modell mit logarithmischer Sigmoid-Aktivierungsfunktion und einem Adam-Optimierer und führt ein verteiltes Training des Modells mit Horovod auf Spark durch.
- Der dritte Teil führt eine Vorhersage für den Testdatensatz durch, wobei das beste Modell verwendet wird, das den mittleren absoluten Gesamtfehler des Validierungssatzes minimiert. Anschließend wird eine CSV-Ausgabedatei erstellt.

Siehe den Abschnitt"Maschinelles Lernen" für verschiedene Laufzeitvergleichsergebnisse.

Multi-Worker-Deep-Learning mit Keras zur CTR-Vorhersage

Angesichts der jüngsten Fortschritte bei ML-Plattformen und -Anwendungen richtet sich der Fokus nun stark auf das Lernen im großen Maßstab. Die Klickrate (Click-Through-Rate, CTR) ist definiert als die durchschnittliche Anzahl von Klicks pro hundert Online-Anzeigenimpressionen (ausgedrückt als Prozentsatz). Es wird in zahlreichen Branchen und Anwendungsfällen, darunter digitales Marketing, Einzelhandel, E-Commerce und Dienstleister, als Schlüsselkennzahl eingesetzt. Weitere Einzelheiten zu den Anwendungen von CTR und verteilten Trainingsleistungsergebnissen finden Sie im"Deep-Learning-Modelle für die CTR-Vorhersageleistung" Abschnitt.

In diesem technischen Bericht verwendeten wir eine Variante des "Criteo Terabyte Click Logs-Datensatz" (siehe TR-4904) für verteiltes Deep Learning mit mehreren Workern unter Verwendung von Keras zum Erstellen eines Spark-Workflows mit Deep- und Cross-Network-Modellen (DCN), wobei die Leistung hinsichtlich der Log-Loss-Fehlerfunktion mit einem Basismodell der logistischen Regression von Spark ML verglichen wird. DCN erfasst effizient effektive Merkmalsinteraktionen begrenzten Grades, lernt hochgradig nichtlineare Interaktionen, erfordert keine manuelle Merkmalsentwicklung oder umfassende Suche und weist

einen geringen Rechenaufwand auf.

Daten für Empfehlungssysteme im Webmaßstab sind größtenteils diskret und kategorisch, was zu einem großen und spärlichen Merkmalsraum führt, der die Merkmalserkundung erschwert. Dies hat die meisten groß angelegten Systeme auf lineare Modelle wie die logistische Regression beschränkt. Der Schlüssel zu guten Vorhersagen liegt jedoch darin, häufig vorhersagbare Merkmale zu identifizieren und gleichzeitig ungesehene oder seltene Kreuzmerkmale zu untersuchen. Lineare Modelle sind einfach, interpretierbar und leicht skalierbar, ihre Ausdruckskraft ist jedoch begrenzt.

Andererseits hat sich gezeigt, dass Kreuzmerkmale für die Verbesserung der Ausdruckskraft der Modelle von Bedeutung sind. Leider ist zur Identifizierung solcher Features häufig eine manuelle Feature-Entwicklung oder eine umfassende Suche erforderlich. Die Verallgemeinerung auf unsichtbare Funktionsinteraktionen ist oft schwierig. Durch die Verwendung eines gekreuzten neuronalen Netzwerks wie DCN wird aufgabenspezifisches Feature Engineering vermieden, indem Feature-Crossing explizit und automatisch angewendet wird. Das Kreuznetzwerk besteht aus mehreren Schichten, wobei der höchste Grad an Interaktionen nachweislich durch die Schichttiefe bestimmt wird. Jede Schicht erzeugt Interaktionen höherer Ordnung auf der Grundlage bestehender Interaktionen und behält die Interaktionen der vorherigen Schichten bei.

Ein tiefes neuronales Netzwerk (DNN) verspricht, sehr komplexe Interaktionen zwischen Features zu erfassen. Im Vergleich zu DCN erfordert es jedoch fast eine Größenordnung mehr Parameter, kann keine Kreuzmerkmale explizit bilden und kann einige Arten von Merkmalsinteraktionen möglicherweise nicht effizient erlernen. Das Cross-Network ist speichereffizient und einfach zu implementieren. Durch das gemeinsame Training der Cross- und DNN-Komponenten werden prädiktive Feature-Interaktionen effizient erfasst und eine hochmoderne Leistung im Criteo CTR-Datensatz erzielt.

Ein DCN-Modell beginnt mit einer Einbettungs- und Stapelschicht, gefolgt von einem Quernetzwerk und einem tiefen Netzwerk parallel. Darauf folgt wiederum eine letzte Kombinationsschicht, die die Ausgaben der beiden Netzwerke kombiniert. Ihre Eingabedaten können ein Vektor mit spärlichen und dichten Merkmalen sein. In Spark enthalten die Bibliotheken den Typ `SparseVector` . Daher ist es für Benutzer wichtig, zwischen den beiden zu unterscheiden und beim Aufrufen der jeweiligen Funktionen und Methoden vorsichtig zu sein. In webbasierten Empfehlungssystemen wie der CTR-Vorhersage sind die Eingaben meist kategorische Merkmale, zum Beispiel `'country=usa'` . Solche Merkmale werden oft als One-Hot-Vektoren kodiert, zum Beispiel: `'[0,1,0, …]'` . One-Hot-Encoding (OHE) mit `SparseVector` ist nützlich, wenn Sie mit realen Datensätzen mit sich ständig änderndem und wachsendem Vokabular arbeiten. Wir haben Beispiele in "DeepCTR" um große Vokabulare zu verarbeiten und Einbettungsvektoren in der Einbettungs- und Stapelschicht unseres DCN zu erstellen.

Der "Criteo Display Ads-Datensatz" sagt die Klickrate der Anzeigen voraus. Es verfügt über 13 ganzzahlige Merkmale und 26 kategorische Merkmale, wobei jede Kategorie eine hohe Kardinalität aufweist. Für diesen Datensatz ist aufgrund der großen Eingabegröße eine Verbesserung des Logverlusts um 0,001 praktisch signifikant. Eine kleine Verbesserung der Vorhersagegenauigkeit für eine große Benutzerbasis kann möglicherweise zu einer erheblichen Steigerung des Umsatzes eines Unternehmens führen. Der Datensatz enthält 11 GB Benutzerprotokolle aus einem Zeitraum von 7 Tagen, was etwa 41 Millionen Datensätzen entspricht. Wir haben Spark verwendet `dataFrame.randomSplit()function` die Daten nach dem Zufallsprinzip für das Training (80 %), die Kreuzvalidierung (10 %) und die restlichen 10 % für Tests aufzuteilen.

DCN wurde auf TensorFlow mit Keras implementiert. Bei der Implementierung des Modelltrainingsprozesses mit DCN gibt es vier Hauptkomponenten:

- **Datenverarbeitung und -einbettung.** Realwertige Merkmale werden durch Anwenden einer Log-Transformation normalisiert. Für kategorische Merkmale betten wir die Merkmale in dichte Vektoren der Dimension 6 × (Kategoriekardinalität) 1/4 ein. Durch Verketten aller Einbettungen entsteht ein Vektor der Dimension 1026.

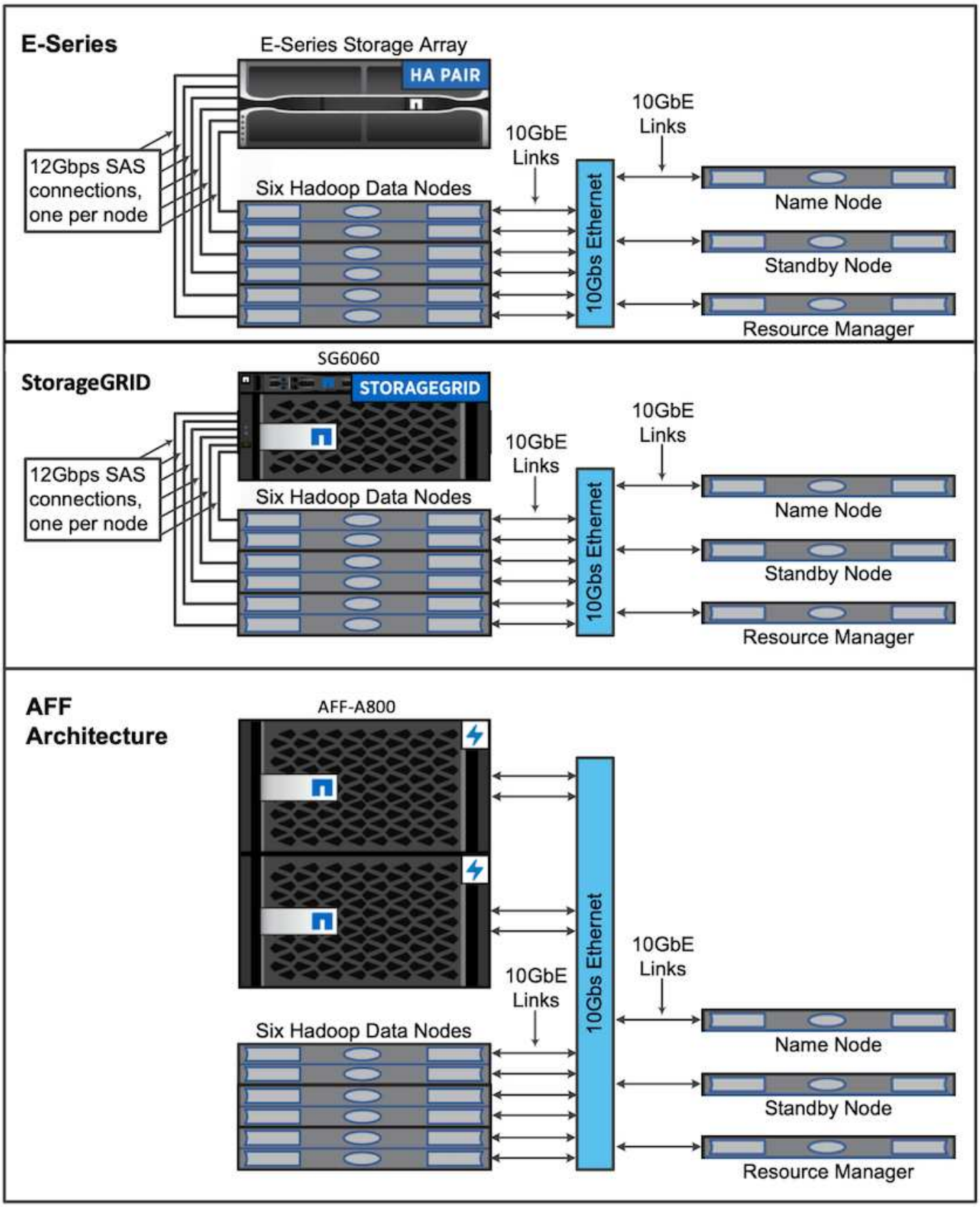
- **Optimierung.** Wir haben eine stochastische Mini-Batch-Optimierung mit dem Adam-Optimierer angewendet. Die Batchgröße wurde auf 512 festgelegt. Auf das tiefe Netzwerk wurde eine Batch-Normalisierung angewendet und die Gradienten-Clip-Norm auf 100 festgelegt.
- **Regularisierung.** Wir haben ein frühes Stoppen verwendet, da sich die L2-Regularisierung oder das Dropout als nicht wirksam erwiesen haben.
- **Hyperparameter.** Wir berichten über Ergebnisse, die auf einer Rastersuche über die Anzahl der verborgenen Schichten, die Größe der verborgenen Schichten, die anfängliche Lernrate und die Anzahl der Kreuzschichten basieren. Die Anzahl der verborgenen Schichten lag zwischen 2 und 5, wobei die Größe der verborgenen Schichten zwischen 32 und 1024 lag. Bei DCN lag die Anzahl der Querschichten zwischen 1 und 6. Die anfängliche Lernrate wurde in Schritten von 0,0001 von 0,0001 auf 0,001 eingestellt. Bei allen Experimenten wurde ein frühzeitiger Stopp bei Trainingsschritt 150.000 angewendet, da ab diesem Zeitpunkt eine Überanpassung eintrat.

Zusätzlich zu DCN haben wir auch andere beliebte Deep-Learning-Modelle zur CTR-Vorhersage getestet, darunter "DeepFM" , "AutoInt" , Und "DCN v2" .

Zur Validierung verwendete Architekturen

Für diese Validierung haben wir vier Worker-Knoten und einen Master-Knoten mit einem AFF-A800-HA-Paar verwendet. Alle Clustermitglieder waren über 10GbE-Netzwerk-Switches verbunden.

Für diese Validierung der NetApp Spark-Lösung haben wir drei verschiedene Speichercontroller verwendet: den E5760, den E5724 und den AFF-A800. Die Speichercontroller der E-Serie wurden mit 12-Gbit/s-SAS-Verbindungen an fünf Datenknoten angeschlossen. Der AFF HA-Paar-Speichercontroller stellt exportierte NFS-Volumes über 10-GbE-Verbindungen für Hadoop-Workerknoten bereit. Die Hadoop-Clustermitglieder wurden über 10-GbE-Verbindungen in den Hadoop-Lösungen E-Series, AFF und StorageGRID verbunden.



Testergebnisse

Wir haben die Skripte TeraSort und TeraValidate im Benchmarking-Tool TeraGen

verwendet, um die Spark-Leistungsvalidierung mit den Konfigurationen E5760, E5724 und AFF-A800 zu messen. Darüber hinaus wurden drei wichtige Anwendungsfälle getestet: Spark NLP-Pipelines und verteiltes TensorFlow-Training, verteiltes Horovod-Training und Multi-Worker-Deep-Learning mit Keras zur CTR-Vorhersage mit DeepFM.

Für die Validierung sowohl der E-Serie als auch der StorageGRID haben wir den Hadoop-Replikationsfaktor 2 verwendet. Für die AFF Validierung haben wir nur eine Datenquelle verwendet.

In der folgenden Tabelle ist die Hardwarekonfiguration für die Spark-Leistungsvalidierung aufgeführt.

Typ	Hadoop-Workerknoten	Antriebstyp	Laufwerke pro Knoten	Speichercontroller
SG6060	4	SAS	12	Einzelnes Hochverfügbarkeitspaar (HA)
E5760	4	SAS	60	Einzelnes HA-Paar
E5724	4	SAS	24	Einzelnes HA-Paar
AFF800	4	SSD	6	Einzelnes HA-Paar

In der folgenden Tabelle sind die Softwareanforderungen aufgeführt.

Software	Version
RHEL	7,9
OpenJDK-Laufzeitumgebung	1.8.0
OpenJDK 64-Bit-Server-VM	25,302
Git	2.24.1
GCC/G++	11.2.1
Funke	3.2.1
PySpark	3.1.2
SparkNLP	3.4.2
TensorFlow	2.9.0
Keras	2.9.0
Horovod	0.24.3

Finanzstimmungsanalyse

Wir veröffentlichten"TR-4910: Stimmungsanalyse aus der Kundenkommunikation mit NetApp AI" , in dem eine End-to-End-Konversations-KI-Pipeline mithilfe der "NetApp DataOps Toolkit" , AFF -Speicher und NVIDIA DGX-System. Die Pipeline führt Batch-Audiosignalverarbeitung, automatische Spracherkennung (ASR), Transferlernen und Stimmungsanalyse durch und nutzt dabei das DataOps Toolkit. "NVIDIA Riva SDK" und die "Tao-Rahmen" . Wir haben den Anwendungsfall der Stimmungsanalyse auf die Finanzdienstleistungsbranche ausgeweitet, einen SparkNLP-Workflow erstellt, drei BERT-Modelle für verschiedene NLP-Aufgaben wie die Erkennung benannter Entitäten geladen und die Stimmung auf Satzebene für die vierteljährlichen Gewinnaufrufe der Top 10-Unternehmen des NASDAQ ermittelt.

Das folgende Skript `sentiment_analysis_spark. py` verwendet das FinBERT-Modell, um Transkripte in HDFS zu verarbeiten und positive, neutrale und negative Stimmungszahlen zu erzeugen, wie in der folgenden Tabelle gezeigt:

```
-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
```

In der folgenden Tabelle ist die Stimmungsanalyse auf Satzebene zu den Gewinnaufrufen der Top 10-Unternehmen des NASDAQ von 2016 bis 2020 aufgeführt.

Stimmungszählung und Prozentsatz	**Alle 10 Unternehmen**	**AAPL**	**AMD**	**AMZN**	**CSCO**	**GOOGL**	**INTC**	**MSFT**	**NVDA**
Positive Zählungen	7447	1567	743	290	682	826	824	904	417
Neutrale Zählungen	64067	6856	7596	5086	6650	5914	6099	5715	6189
Negative Zählungen	1787	253	213	84	189	97	282	202	89
Nicht kategorisierte Zählungen	196	0	0	76	0	0	0	1	0
(Gesamtzahl)	73497	8676	8552	5536	7521	6837	7205	6822	6695

Prozentual gesehen sind die meisten Sätze der CEOs und CFOs sachlich und daher neutral. Während einer Telefonkonferenz zu den Quartalsergebnissen stellen Analysten Fragen, die eine positive oder negative Stimmung zum Ausdruck bringen können. Es lohnt sich, quantitativ weiter zu untersuchen, wie sich eine

negative oder positive Stimmung auf die Aktienkurse am selben oder am nächsten Handelstag auswirkt.

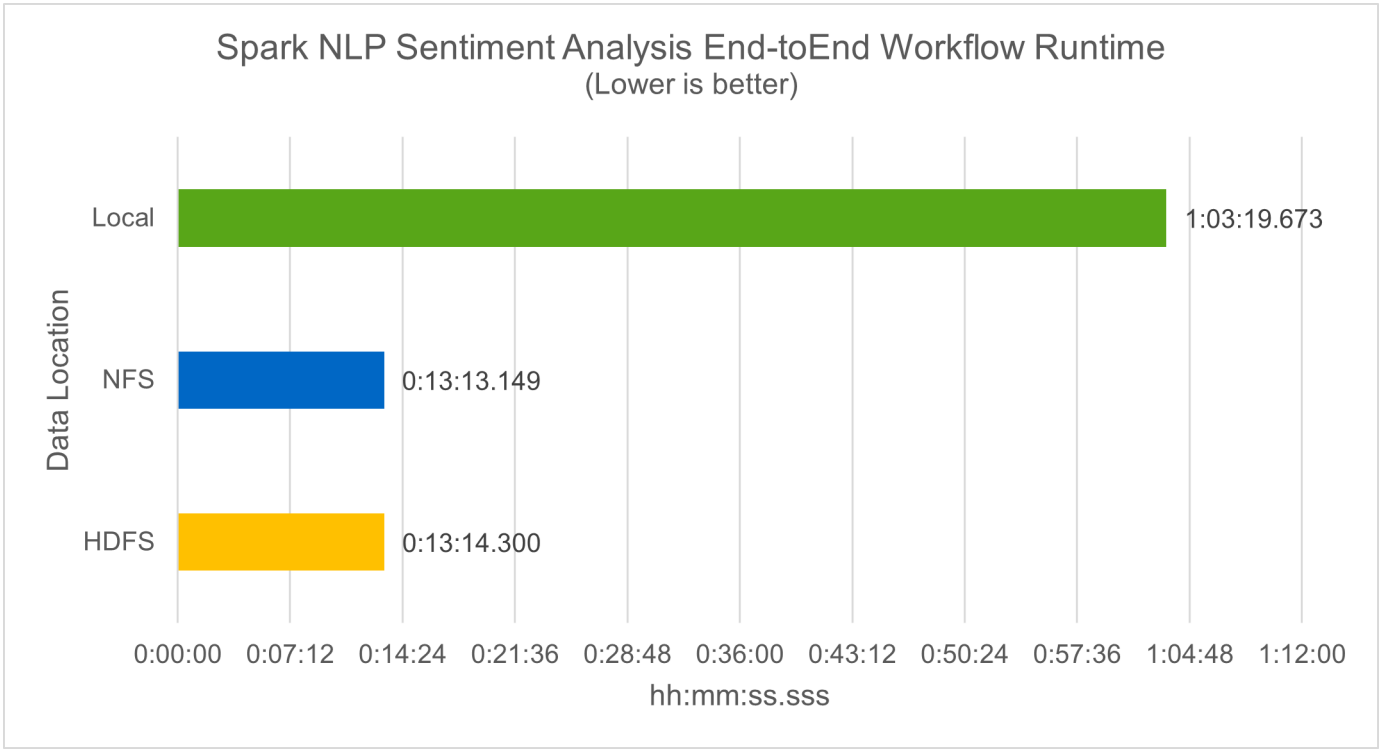
In der folgenden Tabelle ist die Stimmungsanalyse auf Satzebene für die Top 10-Unternehmen des NASDAQ in Prozent aufgeführt.

Stimmungsprozentsatz	Alle 10 Unternehmen	AAPL	AMD	AMZN	CSCO	GOOGL	INTC	MSFT	NVDA
Positiv	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 %
Negativ	2,43 %	2,92 %	2,49 %	1,52 %	2,51 %	1,42 %	3,91 %	2,96 %	1,33 %
Unkategorisiert	0,27 %	0 %	0 %	1,37 %	0 %	0 %	0 %	0,01 %	0 %

In Bezug auf die Workflow-Laufzeit konnten wir eine signifikante Verbesserung um das 4,78-fache gegenüber `local` Modus zu einer verteilten Umgebung in HDFS und eine weitere Verbesserung von 0,14 % durch Nutzung von 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
```

Wie die folgende Abbildung zeigt, verbesserte die Daten- und Modellparallelität die Geschwindigkeit der Datenverarbeitung und der verteilten TensorFlow-Modellinferenz. Die Datenspeicherung in NFS führte zu einer etwas besseren Laufzeit, da der Engpass im Workflow das Herunterladen vortrainierter Modelle ist. Wenn wir die Größe des Transkriptdatensatzes erhöhen, wird der Vorteil von NFS deutlicher.



Verteiltes Training mit Horovod-Leistung

Der folgende Befehl erzeugte Laufzeitinformationen und eine Protokolldatei in unserem Spark-Cluster unter Verwendung eines einzigen `master` Knoten mit 160 Executoren mit jeweils einem Kern. Der Executor-Speicher wurde auf 5 GB begrenzt, um Speicherfehler zu vermeiden. Siehe den Abschnitt"Python-Skripte für jeden wichtigen Anwendungsfall" Weitere Einzelheiten zur Datenverarbeitung, zum Modelltraining und zur Berechnung der Modellgenauigkeit in `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
```

Die resultierende Laufzeit mit zehn Trainingsepochen war wie folgt:

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

Es dauerte mehr als 43 Minuten, um Eingabedaten zu verarbeiten, ein DNN-Modell zu trainieren, die Genauigkeit zu berechnen und TensorFlow-Checkpoints und eine CSV-Datei für Vorhersageergebnisse zu erstellen. Wir haben die Anzahl der Trainingsepochen auf 10 begrenzt, in der Praxis wird sie jedoch oft auf 100 gesetzt, um eine zufriedenstellende Modellgenauigkeit zu gewährleisten. Die Trainingszeit skaliert normalerweise linear mit der Anzahl der Epochen.

Als nächstes nutzten wir die vier im Cluster verfügbaren Worker-Knoten und führten das gleiche Skript in `yarn` Modus mit Daten in 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
```

Die resultierende Laufzeit wurde wie folgt verbessert:

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

Mit Horovods Modell- und Datenparallelität in Spark konnten wir eine 5,29-fache Laufzeitbeschleunigung von `yarn` gegen `local` Modus mit zehn Trainingsepochen. Dies wird in der folgenden Abbildung mit den Legenden dargestellt `HDFS` Und `Local` . Das Training des zugrunde liegenden TensorFlow-DNN-Modells kann mit GPUs, sofern verfügbar, weiter beschleunigt werden. Wir planen, diese Tests durchzuführen und die Ergebnisse in einem zukünftigen technischen Bericht zu veröffentlichen.

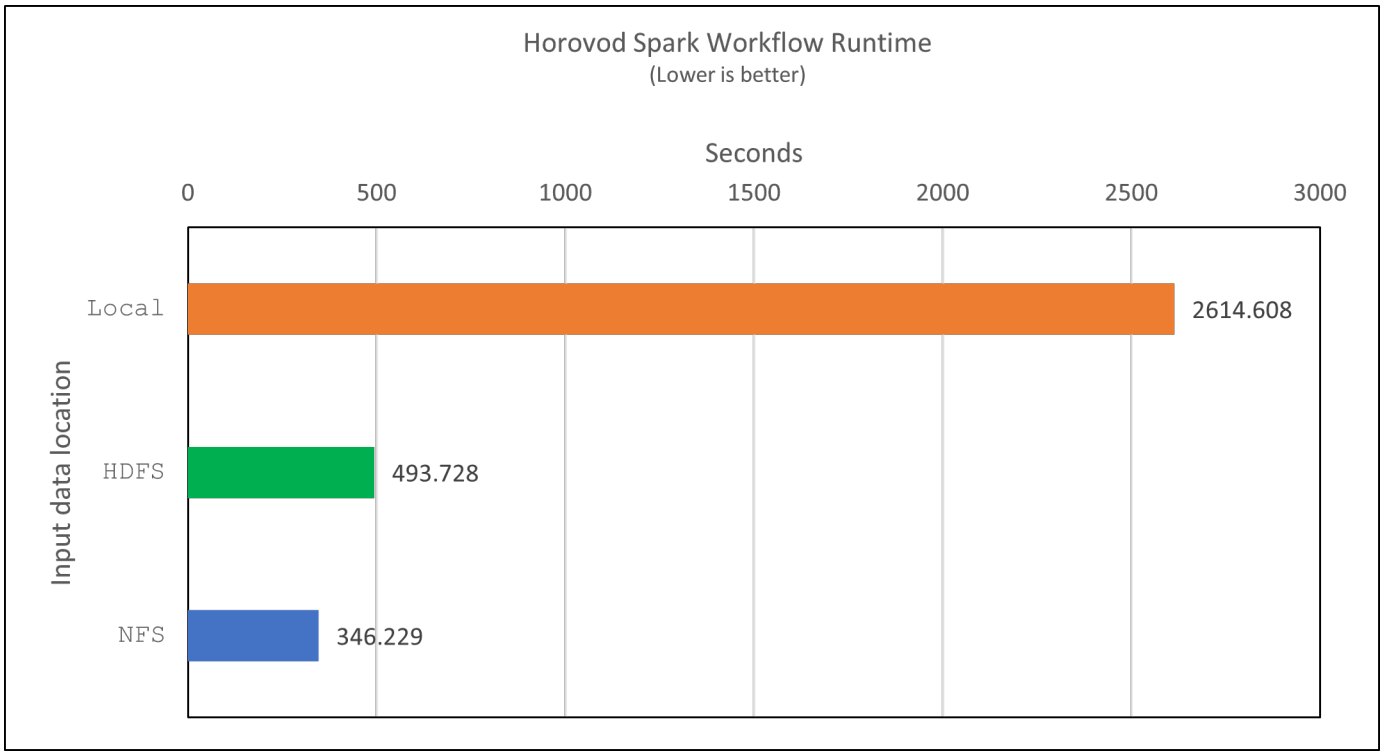
Unser nächster Test verglich die Laufzeiten mit Eingabedaten in NFS und HDFS. Das NFS-Volume auf der AFF A800 wurde gemountet auf `/sparkdemo/horovod` über die fünf Knoten (ein Master, vier Worker) in unserem Spark-Cluster. Wir haben einen ähnlichen Befehl wie bei den vorherigen Tests ausgeführt, mit dem `--data- dir` Parameter, der jetzt auf die NFS-Einbindung zeigt:

```
(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
```

Die resultierende Laufzeit mit NFS war wie folgt:

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

Es kam zu einer weiteren Beschleunigung um das 1,43-Fache, wie in der folgenden Abbildung gezeigt. Daher profitieren Kunden mit einem an ihren Cluster angeschlossenen NetApp All-Flash-Speicher von den Vorteilen einer schnellen Datenübertragung und -verteilung für Horovod Spark-Workflows und erreichen eine 7,55-fache Beschleunigung im Vergleich zur Ausführung auf einem einzelnen Knoten.



Deep-Learning-Modelle für die CTR-Vorhersageleistung

Für Empfehlungssysteme, die auf die Maximierung der Klickrate ausgelegt sind, müssen Sie die komplexen Funktionsinteraktionen hinter dem Benutzerverhalten erlernen, die sich mathematisch von der niedrigsten bis zur höchsten Ordnung berechnen lassen. Für ein gutes Deep-Learning-Modell sollten sowohl Merkmalsinteraktionen niedriger als auch höherer Ordnung gleichermaßen wichtig sein, ohne dass das eine oder das andere bevorzugt wird. Deep Factorization Machine (DeepFM), ein auf Faktorisierungsmaschinen basierendes neuronales Netzwerk, kombiniert Faktorisierungsmaschinen für Empfehlungen und Deep Learning für das Merkmalslernen in einer neuen neuronalen Netzwerkarchitektur.

Obwohl herkömmliche Faktorisierungsmaschinen paarweise Merkmalsinteraktionen als inneres Produkt latenter Vektoren zwischen Merkmalen modellieren und theoretisch Informationen höherer Ordnung erfassen können, verwenden Anwender des maschinellen Lernens in der Praxis aufgrund der hohen Rechen- und Speicherkomplexität normalerweise nur Merkmalsinteraktionen zweiter Ordnung. Varianten tiefer neuronaler Netzwerke wie die von Google "Breite und tiefe Modelle" lernt andererseits anspruchsvolle Merkmalsinteraktionen in einer hybriden Netzwerkstruktur durch die Kombination eines linearen breiten Modells und eines tiefen Modells.

Es gibt zwei Eingaben für dieses Wide & Deep-Modell, eine für das zugrunde liegende Wide-Modell und die andere für das Deep-Modell. Letzterer Teil erfordert noch immer eine fachmännische Feature-Entwicklung und macht die Technik daher weniger auf andere Domänen übertragbar. Anders als das Wide & Deep-Modell kann DeepFM effizient mit Rohmerkmalen trainiert werden, ohne dass ein Feature-Engineering erforderlich ist, da der breite und der tiefe Teil denselben Input und Einbettungsvektor verwenden.

Wir haben zunächst die Criteo `train.txt` (11 GB) in eine CSV-Datei mit dem Namen `ctr_train.csv` in einem NFS-Mount gespeichert `/sparkdemo/tr-4570-data` mit `run_classification_criteo_spark.py` aus dem Abschnitt"Python-Skripte für jeden wichtigen Anwendungsfall." Innerhalb dieses Skripts wird die Funktion `process_input_file` führt mehrere String-Methoden aus, um Tabs zu entfernen und einzufügen `','` als Trennzeichen und `'\n'` als Zeilenumbruch. Beachten Sie, dass Sie nur das Original verarbeiten müssen `train.txt` einmal, sodass der Codeblock als Kommentar angezeigt wird.

Für die folgenden Tests verschiedener DL-Modelle verwendeten wir `ctr_train.csv` als Eingabedatei. In nachfolgenden Testläufen wurde die CSV-Eingabedatei in einen Spark DataFrame mit Schema eingelesen, das ein Feld von `'label'` , ganzzahlige dichte Merkmale `['I1', 'I2', 'I3', …, 'I13']` und spärliche Merkmale `['C1', 'C2', 'C3', …, 'C26']` . Die folgende `spark-submit` Der Befehl nimmt eine CSV-Eingabe entgegen, trainiert DeepFM-Modelle mit 20 % Aufteilung für die Kreuzvalidierung und wählt nach zehn Trainingsepochen das beste Modell aus, um die Vorhersagegenauigkeit im Testsatz zu berechnen:

```
(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
```

Beachten Sie, dass die Datendatei `ctr_train.csv` über 11 GB liegt, müssen Sie eine ausreichende `spark.driver.maxResultSize` größer als die Datensatzgröße, um Fehler zu vermeiden.

```
 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()
```

Im obigen `SparkSession.builder` Konfiguration haben wir auch aktiviert "Apache-Pfeil" , das einen Spark DataFrame in einen Pandas DataFrame mit dem `df.toPandas()` Verfahren.

```
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.
```

Nach der zufälligen Aufteilung gibt es über 36 Millionen Zeilen im Trainingsdatensatz und 9 Millionen Stichproben im Testdatensatz:

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

Da sich dieser technische Bericht auf CPU-Tests ohne Verwendung von GPUs konzentriert, ist es zwingend erforderlich, dass Sie TensorFlow mit entsprechenden Compiler-Flags erstellen. Dieser Schritt vermeidet den Aufruf von GPU-beschleunigten Bibliotheken und nutzt die Advanced Vector Extensions (AVX) und AVX2-Anweisungen von TensorFlow voll aus. Diese Funktionen sind für lineare algebraische Berechnungen wie vektorisierte Addition, Matrixmultiplikationen innerhalb eines Feedforward- oder Backpropagation-DNN-Trainings konzipiert. Der mit AVX2 verfügbare Fused Multiply Add (FMA)-Befehl mit 256-Bit-Gleitkommaregistern (FP) ist ideal für ganzzahligen Code und Datentypen und führt zu einer bis zu zweifachen Beschleunigung. Bei FP-Code und Datentypen erreicht AVX2 eine um 8 % höhere Geschwindigkeit als 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.
```

Um TensorFlow aus dem Quellcode zu erstellen, empfiehlt NetApp die Verwendung "Bazel" . Für unsere Umgebung haben wir die folgenden Befehle in der Shell-Eingabeaufforderung ausgeführt, um zu installieren `dnf` , `dnf-plugins` und Bazel.

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

Sie müssen GCC 5 oder neuer aktivieren, um während des Build-Prozesses C++17-Funktionen zu verwenden, die von RHEL mit der Software Collections Library (SCL) bereitgestellt werden. Die folgenden Befehle installieren `devtoolset` und GCC 11.2.1 auf unserem RHEL 7.9-Cluster:

```
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
```

Beachten Sie, dass die letzten beiden Befehle `devtoolset-11` , das verwendet `/opt/rh/devtoolset-11/root/usr/bin/gcc` (GCC 11.2.1). Stellen Sie außerdem sicher, dass Ihre `git` Version ist höher als 1.8.3 (diese wird mit RHEL 7.9 geliefert). Siehe hierzu "Artikel" zur Aktualisierung `git` bis 2.24.1.

Wir gehen davon aus, dass Sie das neueste TensorFlow-Master-Repo bereits geklont haben. Erstellen Sie dann eine `workspace` Verzeichnis mit einem `WORKSPACE` Datei zum Erstellen von TensorFlow aus dem Quellcode mit AVX, AVX2 und FMA. Führen Sie den `configure` Datei und geben Sie den richtigen Python-Binärspeicherort an. "CUDA" ist für unsere Tests deaktiviert, da wir keine GPU verwendet haben. A `.bazelrc` Die Datei wird entsprechend Ihren Einstellungen generiert. Weiter haben wir die Datei bearbeitet und eingestellt `build --define=no_hdfs_support=false` um die HDFS-Unterstützung zu aktivieren. Siehe `.bazelrc` im Abschnitt"Python-Skripte für jeden wichtigen Anwendungsfall," für eine vollständige Liste der Einstellungen und Flags.

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

Nachdem Sie TensorFlow mit den richtigen Flags erstellt haben, führen Sie das folgende Skript aus, um den Criteo Display Ads-Datensatz zu verarbeiten, ein DeepFM-Modell zu trainieren und die Fläche unter der Receiver Operating Characteristic Curve (ROC AUC) aus den Vorhersagewerten zu berechnen.

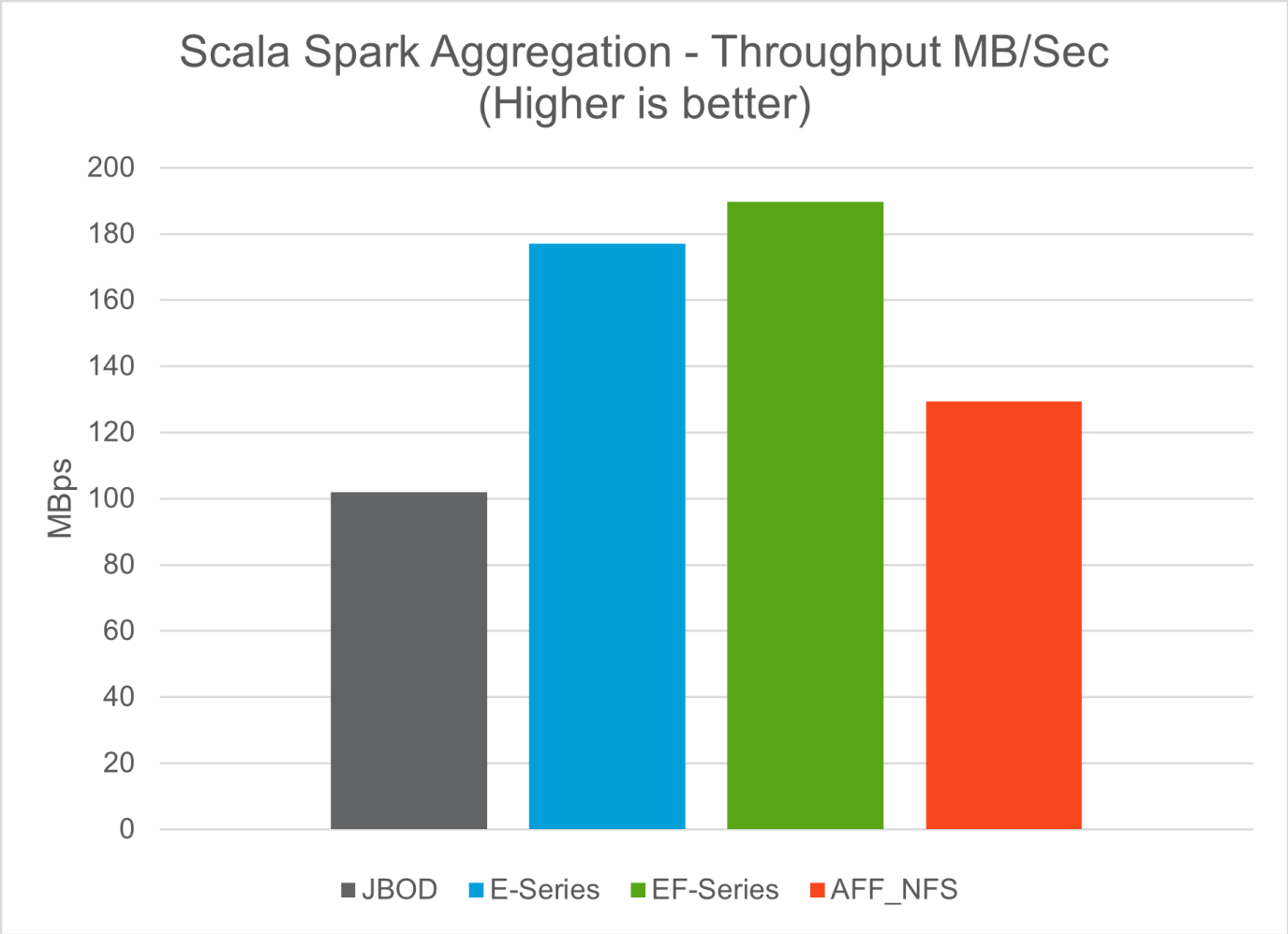
```
(base) [root@n138 examples]# ~/anaconda3/bin/spark-submit
--master yarn
--executor-memory 15g
--executor-cores 1
--num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py
--data-dir file:///sparkdemo/tr-4570-data
> . /run_classification_criteo_spark_nfs.log 2>&1
```

Nach zehn Trainingsepochen haben wir den AUC-Score für den Testdatensatz erhalten:

```
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
```

Ähnlich wie bei früheren Anwendungsfällen haben wir die Spark-Workflow-Laufzeit mit Daten verglichen, die

an verschiedenen Standorten gespeichert sind. Die folgende Abbildung zeigt einen Vergleich der Deep-Learning-CTR-Vorhersage für eine Spark-Workflow-Laufzeit.



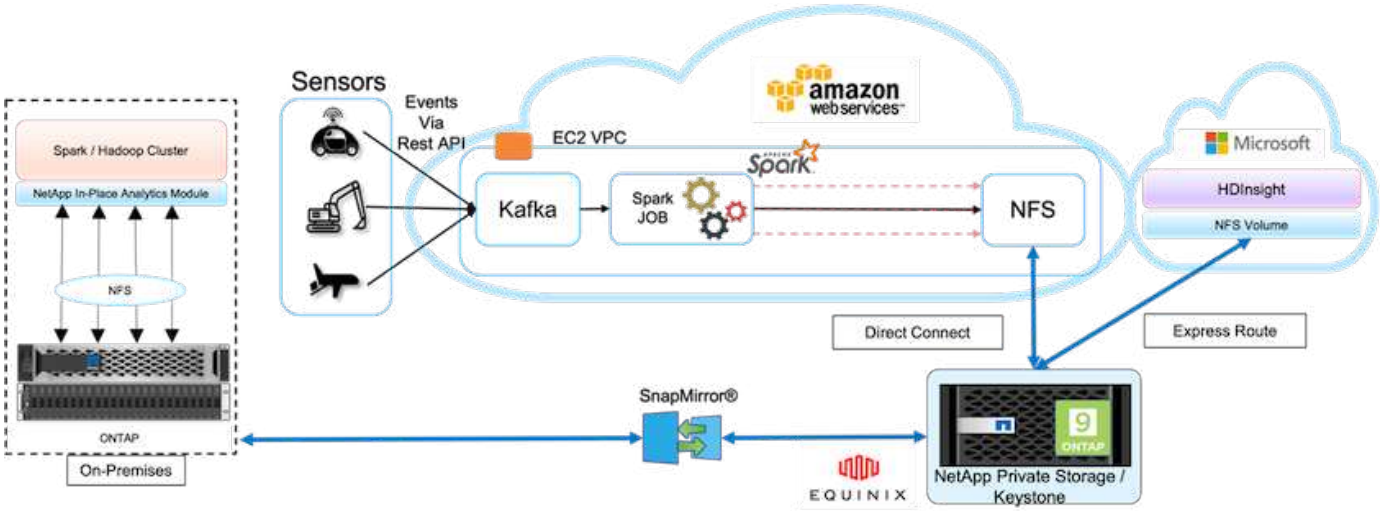
Hybrid-Cloud-Lösung

Ein modernes Unternehmensrechenzentrum ist eine Hybrid-Cloud, die mehrere verteilte Infrastrukturumgebungen über eine kontinuierliche Datenverwaltungsebene mit einem konsistenten Betriebsmodell vor Ort und/oder in mehreren öffentlichen Clouds verbindet. Um das Beste aus einer Hybrid Cloud herauszuholen, müssen Sie in der Lage sein, Daten nahtlos zwischen Ihren lokalen und Multi-Cloud-Umgebungen zu verschieben, ohne dass Datenkonvertierungen oder Anwendungs-Refactoring erforderlich sind.

Kunden haben angegeben, dass sie ihre Reise in die Hybrid Cloud entweder mit der Verlagerung von Sekundärspeichern in die Cloud für Anwendungsfälle wie Datenschutz oder mit der Verlagerung weniger geschäftskritischer Workloads wie Anwendungsentwicklung und DevOps in die Cloud beginnen. Anschließend wenden sie sich kritischeren Arbeitslasten zu. Zu den beliebtesten Hybrid-Cloud-Workloads gehören Web- und Content-Hosting, DevOps und Anwendungsentwicklung, Datenbanken, Analysen und containerisierte Apps. Die Komplexität, die Kosten und die Risiken von KI-Projekten in Unternehmen haben in der Vergangenheit die Einführung von KI von der experimentellen Phase bis zur Produktion behindert.

Mit einer NetApp Hybrid-Cloud-Lösung profitieren Kunden von integrierten Tools für Sicherheit, Datenverwaltung und Compliance mit einem einzigen Control Panel für das Daten- und Workflow-Management

in verteilten Umgebungen und optimieren gleichzeitig die Gesamtbetriebskosten basierend auf ihrem Verbrauch. Die folgende Abbildung zeigt eine Beispiellösung eines Cloud-Service-Partners, der die Aufgabe hat, Multi-Cloud-Konnektivität für die Big-Data-Analysedaten der Kunden bereitzustellen.



In diesem Szenario werden IoT-Daten, die in AWS aus verschiedenen Quellen empfangen werden, an einem zentralen Ort im NetApp Private Storage (NPS) gespeichert. Der NPS-Speicher ist mit Spark- oder Hadoop-Clustern in AWS und Azure verbunden, sodass Big-Data-Analyseanwendungen in mehreren Clouds ausgeführt werden können und auf dieselben Daten zugreifen. Zu den wichtigsten Anforderungen und Herausforderungen für diesen Anwendungsfall zählen die folgenden:

- Kunden möchten Analyseaufträge mit denselben Daten über mehrere Clouds ausführen.
- Daten müssen aus verschiedenen Quellen, beispielsweise lokalen und Cloud-Umgebungen, über verschiedene Sensoren und Hubs empfangen werden.
- Die Lösung muss effizient und kostengünstig sein.
- Die größte Herausforderung besteht darin, eine kostengünstige und effiziente Lösung zu entwickeln, die hybride Analysedienste zwischen verschiedenen lokalen und Cloud-Umgebungen bereitstellt.

Unsere Lösung für Datenschutz und Multicloud-Konnektivität löst das Problem, das entsteht, wenn Cloud-Analyseanwendungen über mehrere Hyperscaler verteilt sind. Wie in der obigen Abbildung gezeigt, werden Daten von Sensoren gestreamt und über Kafka in den AWS Spark-Cluster aufgenommen. Die Daten werden in einer NFS-Freigabe gespeichert, die sich in NPS befindet, das sich außerhalb des Cloud-Anbieters in einem Equinix-Rechenzentrum befindet.

Da NetApp NPS über Direct Connect- bzw. Express Route-Verbindungen mit Amazon AWS und Microsoft Azure verbunden ist, können Kunden das In-Place Analytics-Modul nutzen, um auf die Daten von Amazon- und AWS-Analyseclustern zuzugreifen. Da sowohl der lokale als auch der NPS-Speicher mit ONTAP -Software läuft, "SnapMirror" kann die NPS-Daten in den lokalen Cluster spiegeln und so Hybrid-Cloud-Analysen über lokale und mehrere Clouds hinweg bereitstellen.

Für eine optimale Leistung empfiehlt NetApp normalerweise die Verwendung mehrerer Netzwerkschnittstellen und Direktverbindungen oder Expressrouten für den Zugriff auf die Daten von Cloud-Instanzen. Wir haben andere Data Mover-Lösungen, darunter "XCP" Und "BlueXP Kopieren und Synchronisieren" um Kunden beim Aufbau anwendungsbewusster, sicherer und kostengünstiger Hybrid-Cloud-Spark-Cluster zu unterstützen.

Python-Skripte für jeden wichtigen Anwendungsfall

Die folgenden drei Python-Skripte entsprechen den drei getesteten Hauptanwendungsfällen. Erstens ist `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}")
```

Das zweite Skript ist `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()
```

Das dritte Skript ist `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))
```

Abschluss

In diesem Dokument besprechen wir die Apache Spark-Architektur, Anwendungsfälle von Kunden und das NetApp -Speicherportfolio im Zusammenhang mit Big Data, moderner Analytik sowie KI, ML und DL. In unseren Leistungsvalidierungstests auf Basis branchenüblicher Benchmarking-Tools und der Kundennachfrage zeigten die NetApp Spark-Lösungen eine höhere Leistung als native Hadoop-Systeme. Eine Kombination aus den in diesem Bericht vorgestellten Anwendungsfällen und Leistungsergebnissen von Kunden kann Ihnen bei der Auswahl einer geeigneten Spark-Lösung für Ihre Bereitstellung helfen.

Wo Sie weitere Informationen finden

Die folgenden Referenzen wurden in diesem TR verwendet:

- Apache Spark-Architektur und -Komponenten

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

- Anwendungsfälle für Apache Spark

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

- Spark NLP

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

- BERT

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

- Tiefes und netzwerkübergreifendes Netzwerk für Anzeigenklickvorhersagen

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

- FlexGroup

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

- Streaming-ETL

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

- NetApp E-Series-Lösungen für Hadoop

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

- Moderne Datenanalyselösungen von NetApp

  "Datenanalyselösungen"

- SnapMirror

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

- XCP

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

- BlueXP Kopieren und Synchronisieren

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

- DataOps-Toolkit

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