Apache Spark向け NetAppストレージソリューション

NetApp artificial intelligence solutions

NetApp
February 12, 2026

目次

Apache Spark向けNetAppストレージソリューション

TR-4570: Apache Spark 向けNetAppストレージ ソリューション: アーキテクチャ、ユースケース、パフォーマンス結果

Rick Huang、Karthikeyan Nagalingam、 NetApp

このドキュメントでは、Apache Spark アーキテクチャ、顧客のユースケース、およびビッグ データ分析と人工知能 (AI) に関連するNetAppストレージ ポートフォリオに焦点を当てています。また、一般的な Hadoop システムに対して業界標準の AI、機械学習 (ML)、ディープラーニング (DL) ツールを使用してさまざまなテスト結果を示し、適切な Spark ソリューションを選択できるようにします。まず、Spark アーキテクチャ、適切なコンポーネント、および 2 つのデプロイメント モード (クラスターとクライアント) が必要です。

このドキュメントでは、構成の問題に対処するための顧客の使用事例も提供し、ビッグ データ分析や Spark を使用した AI、ML、DL に関連するNetAppストレージ ポートフォリオの概要についても説明します。最後に、Spark 固有のユースケースとNetApp Spark ソリューション ポートフォリオから得られたテスト結果を紹介します。

顧客の課題

このセクションでは、小売、デジタル マーケティング、銀行、個別製造、プロセス製造、政府、専門サービスなどのデータ増加産業におけるビッグ データ分析と AI/ML/DL に関する顧客の課題に焦点を当てます。

予測不可能なパフォーマンス

従来の Hadoop の導入では、通常、市販のハードウェアが使用されます。パフォーマンスを向上させるには、ネットワーク、オペレーティング システム、Hadoop クラスター、Spark などのエコシステム コンポーネント、およびハードウェアを調整する必要があります。各レイヤーをチューニングしても、Hadoop は環境内での高パフォーマンス向けに設計されていない汎用ハードウェア上で実行されるため、望ましいパフォーマンス レベルを達成するのは難しい場合があります。

メディアとノードの障害

通常の状況でも、市販のハードウェアは故障しやすいものです。データ ノード上の 1 つのディスクに障害が発生した場合、Hadoop マスターはデフォルトでそのノードが正常でないと見なします。次に、そのノードの特定のデータをネットワーク経由でレプリカから正常なノードにコピーします。このプロセスにより、Hadoop ジョブのネットワーク パケットの速度が低下します。クラスターは、異常なノードが正常な状態に戻ったときに、データを再度コピーし、過剰に複製されたデータを削除する必要があります。

Hadoopベンダーロックイン

Hadoop ディストリビューターは独自のバージョン管理を備えた独自の Hadoop ディストリビューションを持っているため、顧客はそれらのディストリビューションに縛られてしまいます。ただし、多くの顧客は、特定の Hadoop ディストリビューションに縛られないインメモリ分析のサポートを必要としています。彼らには、配信を変更しながらも分析を持ち運べる自由が必要です。

複数の言語のサポートが不足している

多くの場合、顧客はジョブを実行するために MapReduce Java プログラムに加えて複数の言語のサポートを必要とします。 SQL やスクリプトなどのオプションにより、回答を得るための柔軟性が向上し、データを整理および取得するためのオプションが増え、データを分析フレームワークに移動する速度が速くなります。

使いにくさ

以前から、Hadoop は使いにくいという不満の声が上がっていました。 Hadoop は新しいバージョンが出るたびによりシンプルかつ強力になっているにもかかわらず、この批判は続いています。 Hadoop では、Java および MapReduce プログラミング パターンを理解する必要がありますが、これはデータベース管理者や従来のスクリプト スキルを持つ人にとっては難しい課題です。

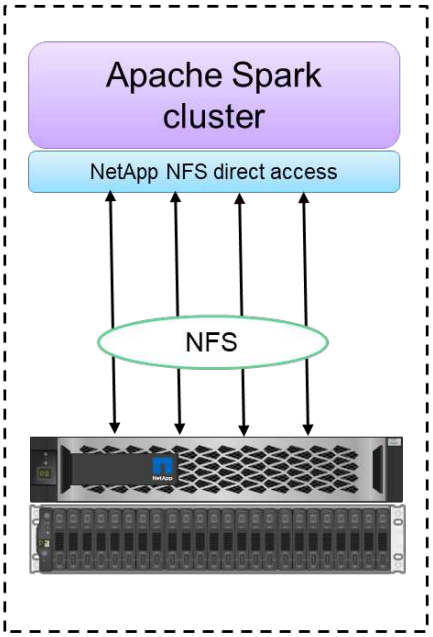
複雑なフレームワークとツール

企業の AI チームはさまざまな課題に直面しています。専門的なデータ サイエンスの知識があっても、さまざまなデプロイメント エコシステムおよびアプリケーション用のツールとフレームワークを、単純に相互に変換できるとは限りません。データ サイエンス プラットフォームは、データの移動が容易で、モデルを再利用可能、すぐに使用できるコード、モデルのプロトタイピング、検証、バージョン管理、共有、再利用、本番環境への迅速な導入に関するベスト プラクティスをサポートするツールを備え、Spark 上に構築された対応するビッグ データ プラットフォームとシームレスに統合される必要があります。

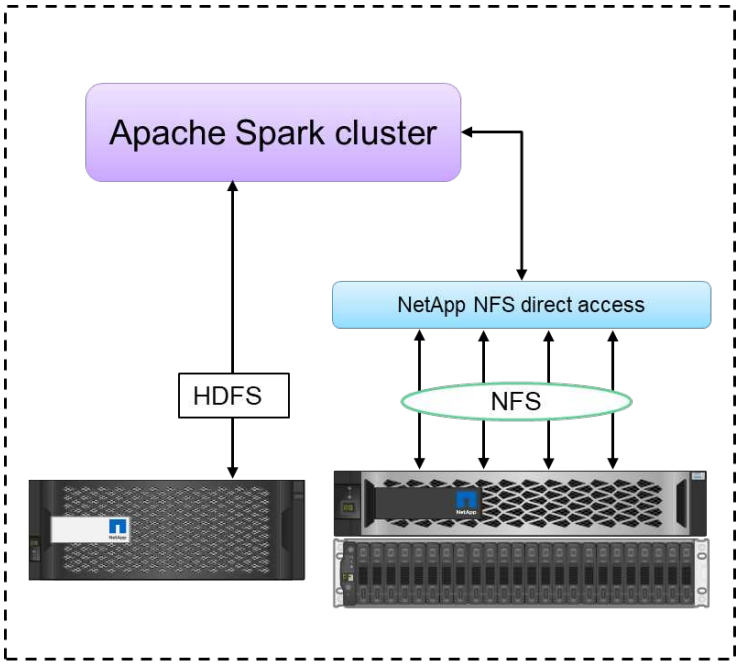
NetAppを選ぶ理由

NetApp は、次の方法で Spark エクスペリエンスを向上できます。

- NetApp NFS ダイレクト アクセス (下の図を参照) を使用すると、データを移動またはコピーすることなく、既存または新規の NFSv3 または NFSv4 データに対してビッグ データ分析ジョブを実行できます。データの複数のコピーを防ぎ、ソースとデータを同期する必要がなくなります。
- より効率的なストレージとより少ないサーバーレプリケーション。たとえば、 NetApp E シリーズ Hadoop ソリューションでは、データのレプリカが 3 つではなく 2 つ必要であり、 FAS Hadoop ソリューションではデータ ソースは必要ですが、データのレプリケーションやコピーは必要ありません。 NetApp ストレージ ソリューションでは、サーバー間のトラフィックも削減されます。
- ドライブおよびノード障害時の Hadoop ジョブとクラスターの動作が改善されました。
- データ取り込みパフォーマンスが向上します。



Configuration 1: NFS as primary storage



Configuration 2: HDFS and NFS in single Spark cluster

たとえば、金融や医療の分野では、ある場所から別の場所へのデータの移動は法的義務を満たす必要があり、これは簡単な作業ではありません。このシナリオでは、 NetApp NFS ダイレクト アクセスが、元の場所から財務データと医療データを分析します。もう 1 つの重要な利点は、 NetApp NFS ダイレクト アクセスを使用すると、ネイティブ Hadoop コマンドを使用して Hadoop データの保護が簡素化され、 NetAppの豊富なデータ管理ポートフォリオを使用してデータ保護ワークフローが実現されることです。

NetApp NFS ダイレクト アクセスは、Hadoop/Spark クラスターに 2 種類の導入オプションを提供します。

- デフォルトでは、Hadoop または Spark クラスターは、データ ストレージとデフォルトのファイル システムとして Hadoop 分散ファイル システム (HDFS) を使用します。 NetApp NFS ダイレクト アクセスでは、デフォルトの HDFS を NFS ストレージに置き換えてデフォルトのファイル システムとして使用できるため、NFS データの直接分析が可能になります。
- 別の導入オプションとして、 NetApp NFS ダイレクト アクセスでは、単一の Hadoop または Spark クラスター内の HDFS とともに NFS を追加ストレージとして構成することがサポートされています。この場合、顧客は NFS エクスポートを通じてデータを共有し、HDFS データと同じクラスターからデータにアクセスできます。

NetApp NFS ダイレクト アクセスを使用する主な利点は次のとおりです。

- 現在の場所からデータを分析することで、分析データを HDFS などの Hadoop インフラストラクチャに移動する、時間とパフォーマンスを消費するタスクを回避します。
- レプリカの数を 3 個から 1 個に減らします。
- ユーザーがコンピューティングとストレージを切り離して、個別に拡張できるようにします。
- ONTAPの豊富なデータ管理機能を活用して、エンタープライズ データ保護を提供します。
- Hortonworks データ プラットフォームの認定。
- ハイブリッド データ分析の展開を可能にします。
- 動的マルチスレッド機能を活用してバックアップ時間を短縮します。

見る"TR-4657: NetAppハイブリッド クラウド データ ソリューション - 顧客のユースケースに基づく Spark と Hadoop"Hadoop データのバックアップ、クラウドからオンプレミスへのバックアップと災害復旧、既存の Hadoop データでの DevTest の有効化、データ保護とマルチクラウド接続、分析ワークロードの高速化を実現します。

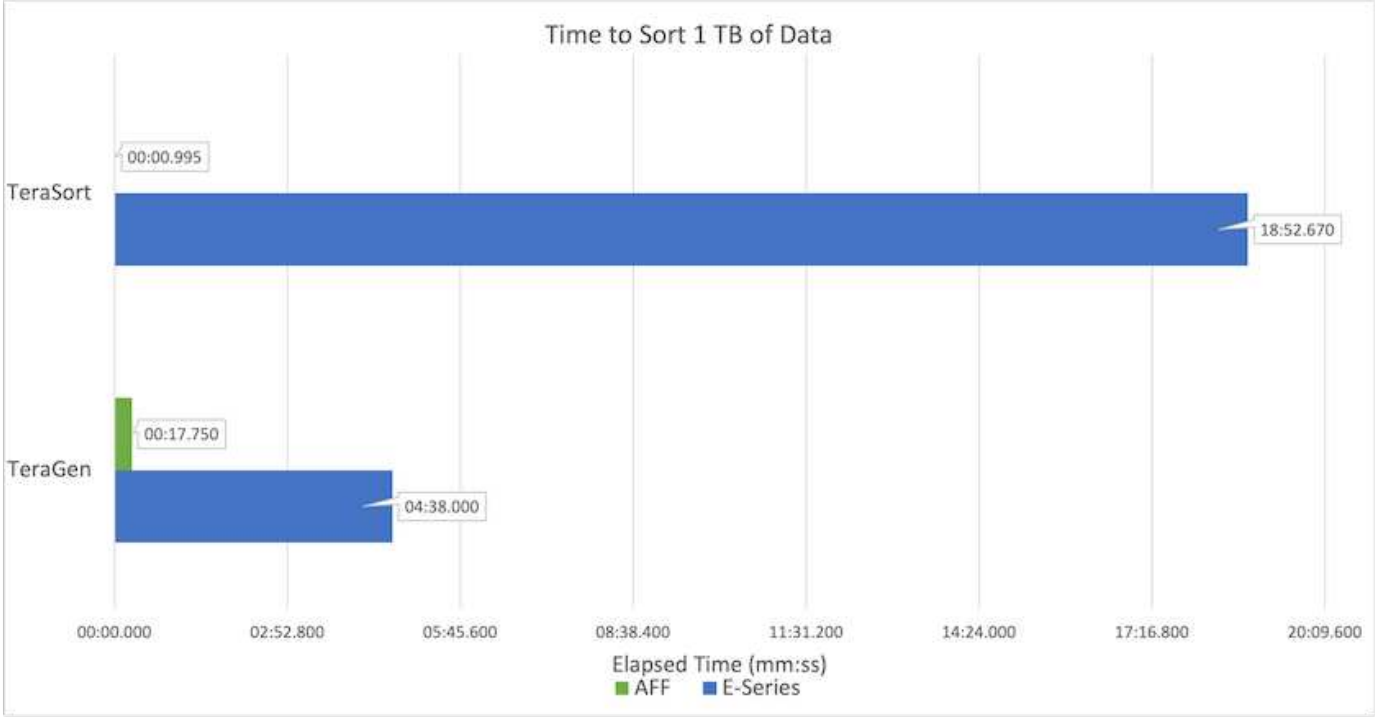
次のセクションでは、Spark のお客様にとって重要なストレージ機能について説明します。

ストレージ階層化

Hadoop ストレージ階層化を使用すると、ストレージ ポリシーに従って、異なるストレージ タイプでファイルを保存できます。ストレージの種類には以下が含まれます `hot`、 `cold`、 `warm`、 `all_ssd`、 `one_ssd`、 そして `lazy_persist`。

異なるストレージ ポリシーを持つ SSD および SAS ドライブを搭載したNetApp AFFストレージ コントローラと E シリーズ ストレージ コントローラで Hadoop ストレージ階層化の検証を実行しました。 AFF-A800 の Spark クラスターには 4 つのコンピューティング ワーカー ノードがありますが、E シリーズのクラスターには 8 つのコンピューティング ワーカー ノードがあります。これは主に、ソリッド ステート ドライブ (SSD) とハード ドライブ ディスク (HDD) のパフォーマンスを比較するためのものです。

次の図は、Hadoop SSD に対するNetAppソリューションのパフォーマンスを示しています。



- ベースライン NL-SAS 構成では、8 つのコンピューティング ノードと 96 個の NL-SAS ドライブが使用されました。この構成では、4 分 38 秒で 1 TB のデータが生成されました。見る "TR-3969 NetApp Eシリーズ Hadoop ソリューション"クラスターとストレージ構成の詳細については、こちらをご覧ください。
- TeraGen を使用すると、SSD 構成では NL-SAS 構成よりも 15.66 倍高速に 1 TB のデータを生成しました。さらに、SSD 構成では、コンピューティング ノードの数とディスク ドライブの数も半分になりました (合計 24 台の SSD ドライブ)。ジョブの完了時間に基づくと、NL-SAS 構成のほぼ 2 倍の速度でした。
- TeraSort を使用すると、SSD 構成では 1 TB のデータを NL-SAS 構成よりも 1138.36 倍速くソートできました。さらに、SSD 構成では、コンピューティング ノードの数とディスク ドライブの数も半分になりました (合計 24 台の SSD ドライブ)。したがって、ドライブあたりでは、NL-SAS 構成よりも約 3 倍高速になりました。

- 重要なのは、回転ディスクからオールフラッシュへの移行によりパフォーマンスが向上することです。コンピューティングノードの数はボトルネックではありませんでした。 NetApp のオールフラッシュ ストレージを使用すると、ランタイム パフォーマンスが適切に拡張されます。
- NFS では、データはすべて一緒にプールされることと機能的に同等であり、ワークロードに応じてコンピューティング ノードの数を削減できます。 Apache Spark クラスター ユーザーは、コンピューティング ノードの数を変更するときにデータを手動で再バランスする必要がありません。

パフォーマンスのスケーリング - スケールアウト

AFFソリューションの Hadoop クラスターからさらに計算能力が必要な場合は、適切な数のストレージ コントローラーを備えたデータ ノードを追加できます。 NetApp、ストレージ コントローラ アレイごとに 4 つのデータ ノードから開始し、ワークロードの特性に応じて、ストレージ コントローラごとに 8 つのデータ ノードまで増やすことを推奨しています。

AFFとFAS はインプレース分析に最適です。コンピューティング要件に基づいてノード マネージャーを追加でき、中断のない操作により、ダウンタイムなしでオンデマンドでストレージ コントローラーを追加できます。当社は、NVME メディア サポート、効率保証、データ削減、QOS、予測分析、クラウド階層化、レプリケーション、クラウド展開、セキュリティなど、 AFFおよびFASの豊富な機能を提供します。お客様が要件を満たせるよう、 NetApp は追加のライセンス費用なしで、ファイルシステム分析、クォータ、オンボックス負荷分散などの機能を提供します。 NetApp は、同時ジョブ数、低レイテンシ、シンプルな操作、および 1 秒あたりのギガバイト単位のスループットにおいて競合他社よりも優れたパフォーマンスを発揮します。さらに、NetApp Cloud Volumes ONTAP は3 つの主要クラウド プロバイダーすべてで実行されます。

パフォーマンスのスケーリング - スケールアップ

スケールアップ機能を使用すると、追加のストレージ容量が必要な場合に、 AFF、 FAS、および E シリーズ システムにディスク ドライブを追加できます。 Cloud Volumes ONTAPでは、ストレージを PB レベルに拡張するために、使用頻度の低いデータをブロック ストレージからオブジェクト ストレージに階層化し、追加のコンピューティングなしでCloud Volumes ONTAPライセンスをスタックするという 2 つの要素を組み合わせています。

複数のプロトコル

NetAppシステムは、SAS、iSCSI、FCP、InfiniBand、NFS など、Hadoop 展開のほとんどのプロトコルをサポートしています。

運用およびサポートソリューション

このドキュメントで説明されている Hadoop ソリューションは、 NetAppによってサポートされています。これらのソリューションは、主要な Hadoop ディストリビューターによって認定されています。詳細については、 "ホートンワークス"サイトとCloudera "認証"そして "partner"サイト。

対象

分析とデータ サイエンスの世界は、IT とビジネスの複数の分野に関係しています。

- データ サイエンティストには、選択したツールとライブラリを使用できる柔軟性が必要です。
- データ エンジニアは、データがどのように流れ、どこに存在するかを知る必要があります。
- DevOps エンジニアには、新しい AI および ML アプリケーションを CI および CD パイプラインに統合するためのツールが必要です。

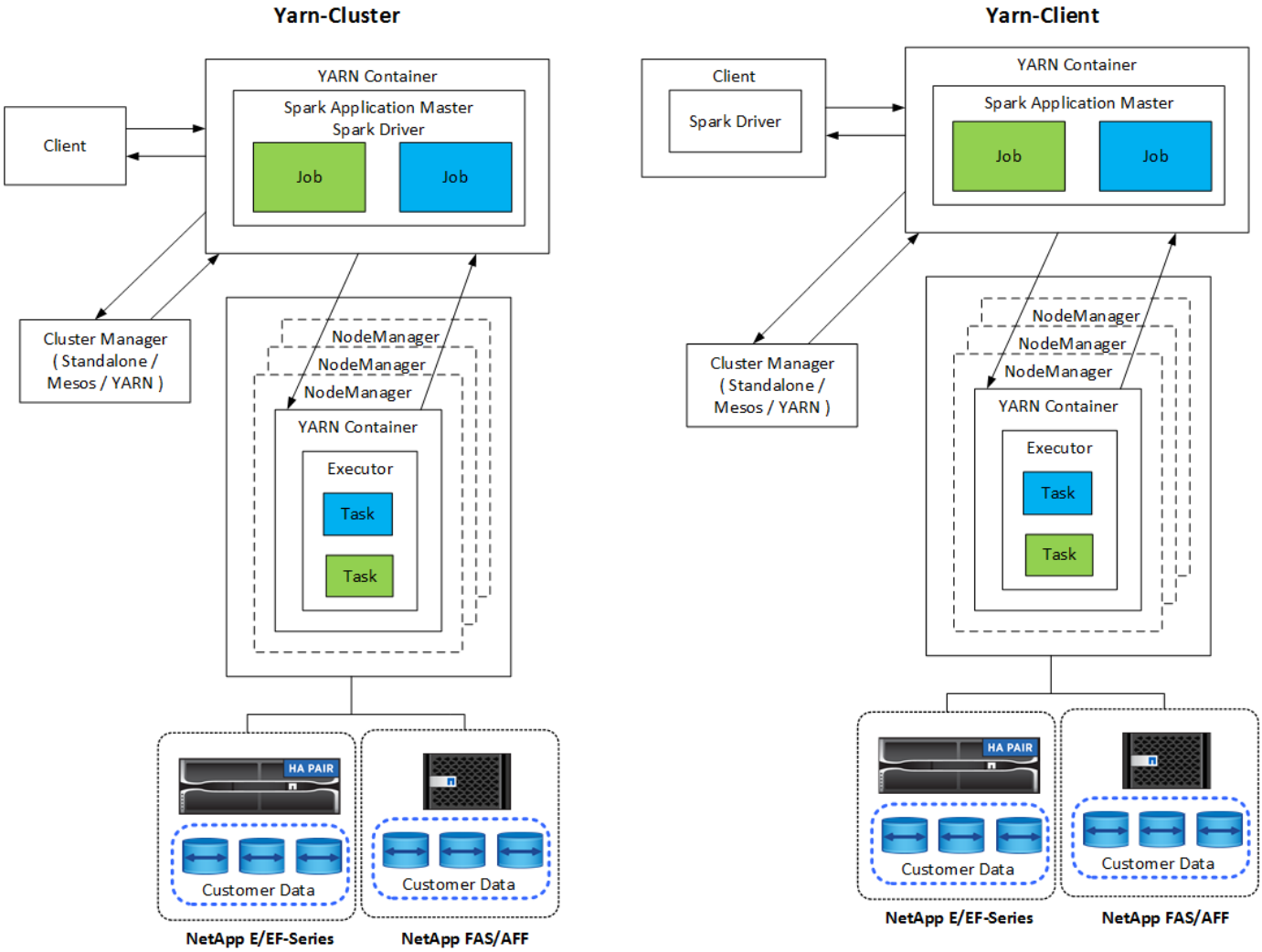
- クラウド管理者とアーキテクトは、ハイブリッド クラウド リソースをセットアップおよび管理できる必要があります。
- ビジネス ユーザーは、分析、AI、ML、DL アプリケーションにアクセスしたいと考えています。

この技術レポートでは、 NetApp AFF、E シリーズ、 StorageGRID、NFS ダイレクト アクセス、Apache Spark、Horovod、Keras が、これらのそれぞれの役割にどのように役立ち、ビジネスに価値をもたらすかについて説明します。

ソリューション技術

Apache Spark は、Hadoop 分散ファイル システム (HDFS) と直接連携する Hadoop アプリケーションを作成するための人気のプログラミング フレームワークです。 Spark は本番環境に対応しており、ストリーミング データの処理をサポートし、MapReduce よりも高速です。 Spark には、効率的な反復処理のために構成可能なメモリ内データ キャッシュがあり、Spark シェルはインタラクティブにデータを学習および探索できます。 Spark を使用すると、Python、Scala、または Java でアプリケーションを作成できます。 Spark アプリケーションは、1 つ以上のタスクを持つ 1 つ以上のジョブで構成されます。

すべての Spark アプリケーションには Spark ドライバーがあります。 YARN クライアント モードでは、ドライバーはクライアント上でローカルに実行されます。 YARN クラスター モードでは、ドライバーはアプリケーション マスター上のクラスターで実行されます。クラスター モードでは、クライアントが切断されてもアプリケーションは実行を継続します。



クラスター マネージャーは 3 つあります。

- *スタンドアロン*このマネージャーは Spark の一部であり、クラスターのセットアップを容易にします。
- *Apache Mesos。*これは、MapReduce やその他のアプリケーションも実行する一般的なクラスター マネージャーです。
- *Hadoop YARN。*これは Hadoop 3 のリソース マネージャーです。

復元力のある分散データセット (RDD) は、Spark の主要コンポーネントです。 RDD は、クラスター内のメモリに保存されたデータから失われたデータや欠落したデータを再作成し、ファイルから取得された初期データやプログラムによって作成された初期データを保存します。 RDD は、ファイル、メモリ内のデータ、または別の RDD から作成されます。 Spark プログラミングでは、変換とアクションという 2 つの操作を実行します。変換では、既存の RDD に基づいて新しい RDD が作成されます。アクションは RDD から値を返します。

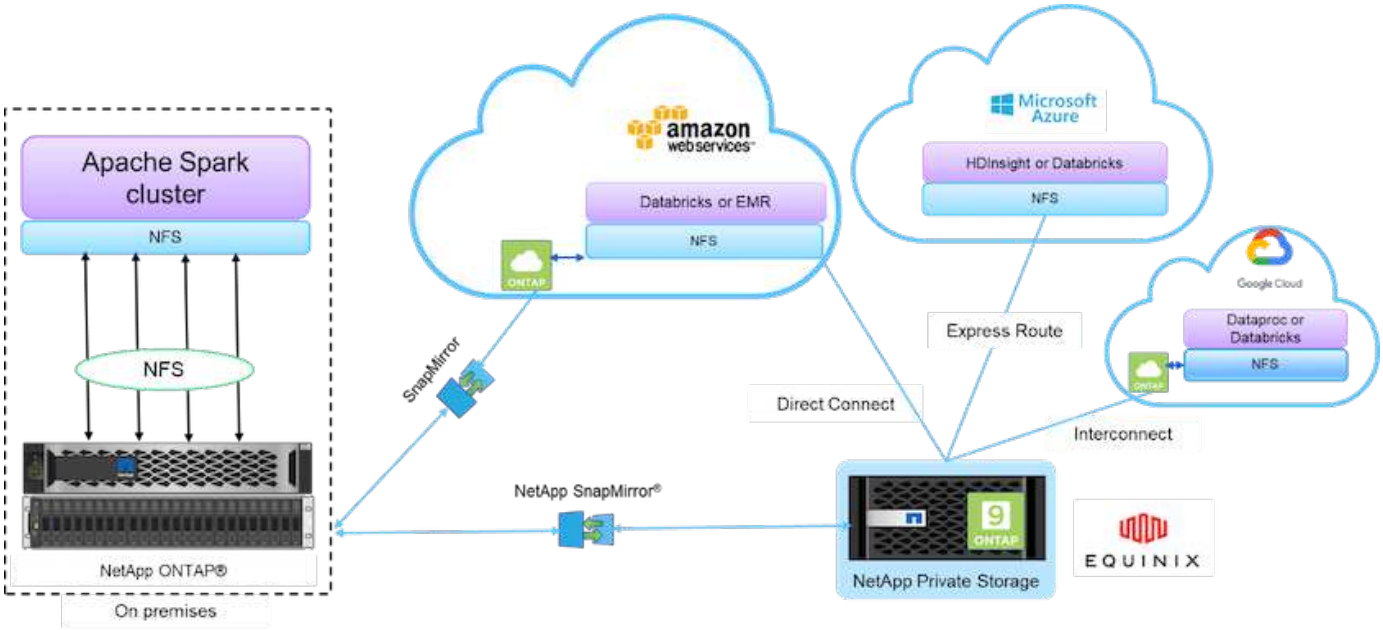
変換とアクションは、Spark データセットとデータフレームにも適用されます。データセットは、RDD (強力な型指定、ラムダ関数の使用) の利点と Spark SQL の最適化された実行エンジンの利点を兼ね備えた分散型データ コレクションです。データセットは JVM オブジェクトから構築し、関数変換 (map、flatMap、filter など) を使用して操作できます。 DataFrame は、名前付きの列に編成されたデータセットです。概念的には、リレーショナル データベースのテーブルまたは R/Python のデータ フレームと同等です。 DataFrames は、構造化データ ファイル、Hive/HBase 内のテーブル、オンプレミスまたはクラウド内の外部データベース、既存の RDD など、さまざまなソースから構築できます。

Spark アプリケーションには、1 つ以上の Spark ジョブが含まれます。ジョブはエグゼキュータ内でタスクを実行し、エグゼキュータは YARN コンテナ内で実行されます。各エグゼキュータは単一のコンテナ内で実行され、アプリケーションの存続期間中ずっと存在します。アプリケーションの起動後にエグゼキュータが固定され、YARN はすでに割り当てられているコンテナのサイズを変更しません。 Executor はメモリ内のデータに対してタスクを同時に実行できます。

NetApp Spark ソリューションの概要

NetApp には、 FAS/ AFF、E シリーズ、 Cloud Volumes ONTAP の3 つのストレージ ポートフォリオがあります。当社では、Apache Spark を使用した Hadoop ソリューション向けに、 AFFおよび E シリーズをONTAPストレージ システムで検証しました。

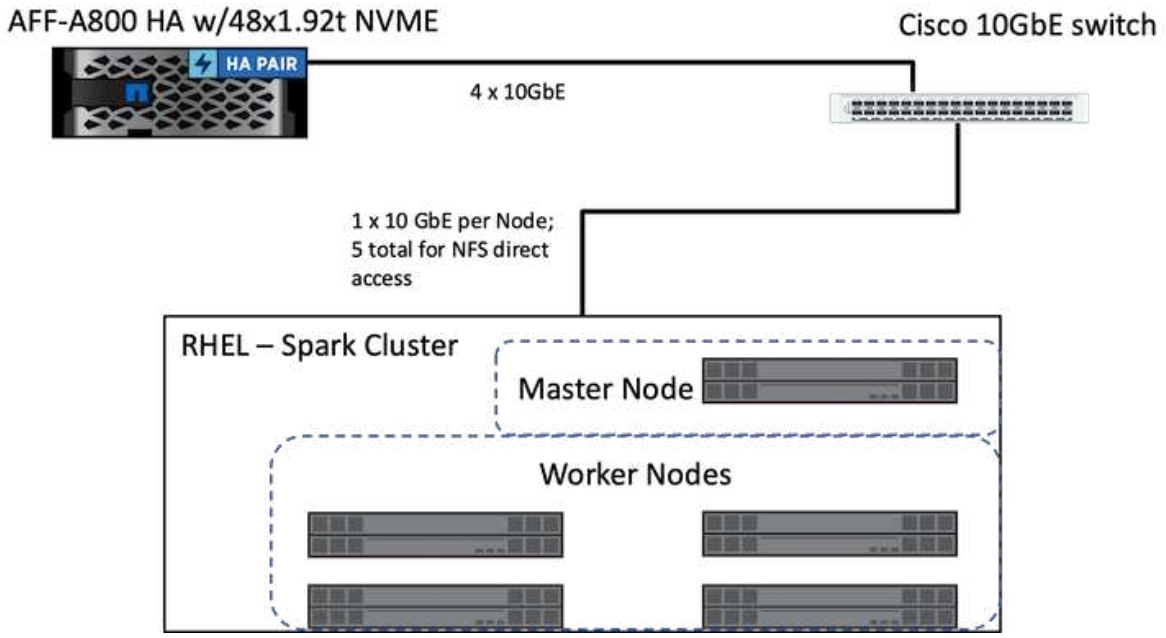
NetAppが提供するデータ ファブリックは、次の図に示すように、データ アクセス、制御、保護、セキュリティのためのデータ管理サービスとアプリケーション (ビルディング ブロック) を統合します。



上の図の構成要素は次のとおりです。

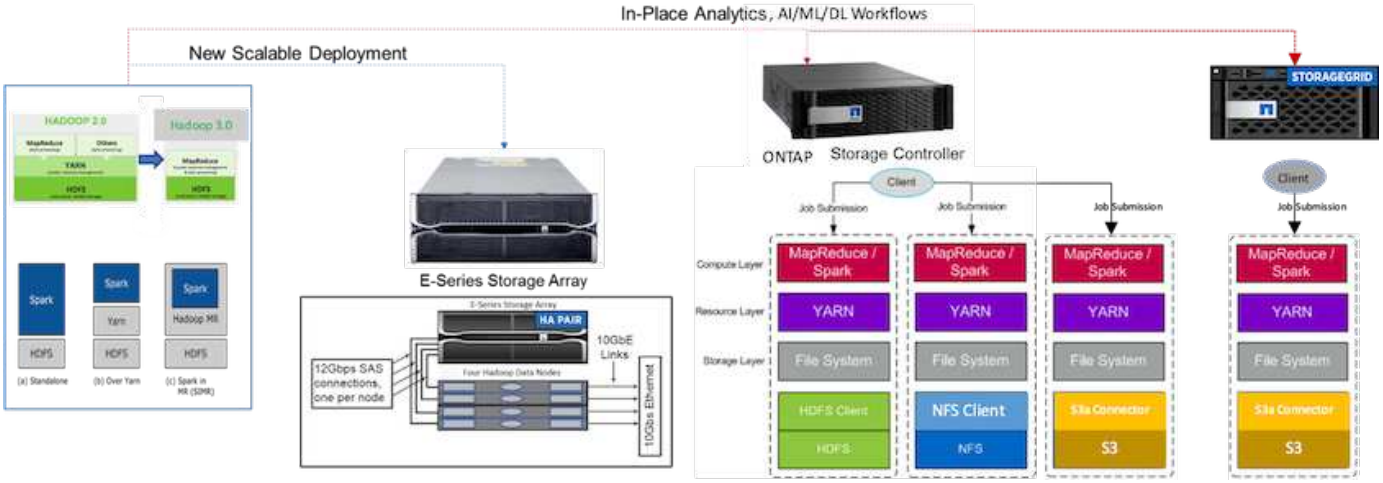
- * NetApp NFS ダイレクト アクセス。*追加のソフトウェアやドライバーを必要とせずに、最新の Hadoop および Spark クラスターにNetApp NFS ボリュームへの直接アクセスを提供します。
- * NetApp Cloud Volumes ONTAPとGoogle Cloud NetApp Volumes 。* Amazon Web Services (AWS) または Microsoft Azure クラウド サービスのAzure NetApp Files (ANF) で実行されるONTAPに基づくソフトウェア定義の接続ストレージ。
- * NetApp SnapMirrorテクノロジー。*オンプレミスとONTAP Cloud または NPS インスタンス間のデータ保護機能を提供します。
- *クラウド サービス プロバイダー*これらのプロバイダーには、AWS、Microsoft Azure、Google Cloud 、IBM Cloud が含まれます。
- **PaaS** AWS の Amazon Elastic MapReduce (EMR) や Databricks、Microsoft Azure HDInsight や Azure Databricks などのクラウドベースの分析サービス。

次の図は、 NetAppストレージを使用した Spark ソリューションを示しています。

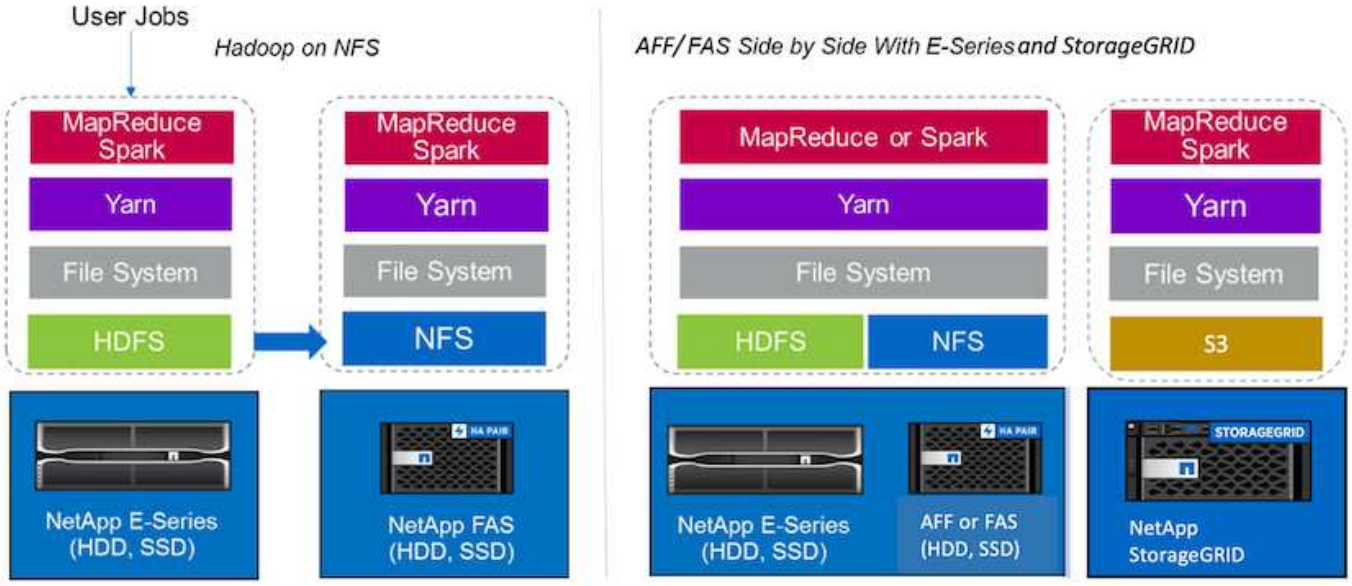


ONTAP Spark ソリューションは、既存の本番データへのアクセスを使用して、インプレース分析と AI、ML、DL ワークフローにNetApp NFS 直接アクセス プロトコルを使用します。 Hadoop ノードで利用可能な本番データは、インプレース分析および AI、ML、DL ジョブを実行するためにエクスポートされます。 Hadoop ノードで処理するデータには、 NetApp NFS 直接アクセスを使用しても使用しなくてもアクセスできます。 Sparkではスタンドアロンまたは `yarn`クラスタマネージャでは、`NFS`ボリュームを次のように設定できます。 `\file://<target_volume>` 。異なるデータセットを使用して 3 つのユースケースを検証しました。これらの検証の詳細は、「テスト結果」のセクションに記載されています。 （外部参照）

次の図は、 NetApp Apache Spark/Hadoop ストレージの位置付けを示しています。



E シリーズ Spark ソリューション、 AFF/ FAS ONTAP Spark ソリューション、 StorageGRID Spark ソリューションの独自の機能を特定し、詳細な検証とテストを実施しました。当社の観察に基づき、 NetApp は、グリーンフィールド インストールと新しいスケーラブルな導入には E シリーズ ソリューションを推奨し、既存の NFS データを使用したインプレース分析、AI、ML、DL ワークロードにはAFF/ FASソリューションを推奨し、オブジェクト ストレージが必要な場合の AI、ML、DL および最新のデータ分析にはStorageGRID を推奨しています。



データ レイクは、分析、AI、ML、DL ジョブに使用できるネイティブ形式の大規模なデータセットのストレージ リポジトリです。 E シリーズ、 AFF/ FAS、 StorageGRID SG6060 Spark ソリューション用のデータ レイク リポジトリを構築しました。 E シリーズ システムは Hadoop Spark クラスターへの HDFS アクセスを提供しますが、既存の運用データは Hadoop クラスターへの NFS 直接アクセス プロトコルを通じてアクセスされます。オブジェクト ストレージに存在するデータセットに対して、 NetApp StorageGRID はS3 および S3a の安全なアクセスを提供します。

ユースケースの概要

このページでは、このソリューションを使用できるさまざまな領域について説明します。

ストリーミングデータ

Apache Spark は、ストリーミング抽出、変換、ロード (ETL) プロセス、データ拡充、イベント検出のトリガー、複雑なセッション分析に使用されるストリーミング データを処理できます。

- *ストリーミング ETL*データは、データストアにプッシュされる前に継続的にクリーンアップされ、集約されます。 Netflix は、Kafka と Spark ストリーミングを使用して、さまざまなデータ ソースから毎日数十億のイベントを処理できるリアルタイムのオンライン映画推奨およびデータ監視ソリューションを構築しています。ただし、バッチ処理用の従来の ETL は異なる方法で処理されます。このデータは最初に読み取られ、データベースに書き込まれる前にデータベース形式に変換されます。
- データの拡充 Spark ストリーミングは、ライブ データを静的データで強化し、よりリアルタイムなデータ分析を可能にします。たとえば、オンライン広告主は、顧客の行動に関する情報に基づいて、パーソナライズされたターゲット広告を配信できます。
- トリガーイベント検出。 Spark ストリーミングを使用すると、潜在的に深刻な問題を示唆する異常な動作を迅速に検出して対応できます。たとえば、金融機関はトリガーを使用して不正な取引を検出して阻止し、病院はトリガーを使用して患者のバイタルサインで検出された危険な健康状態の変化を検出します。
- 複雑なセッション分析。 Spark ストリーミングは、Web サイトまたはアプリケーションにログインした後のユーザー アクティビティなどのイベントを収集し、それらをグループ化して分析します。たとえば、Netflix はこの機能を使用して、リアルタイムの映画推奨を提供しています。

ストリーミングデータの設定、Confluent Kafkaの検証、パフォーマンステストの詳細については、"TR-4912: NetAppを使用した Confluent Kafka 階層型ストレージのベストプラクティスガイドライン" 。

機械学習

Spark 統合フレームワークは、機械学習ライブラリ (MLlib) を使用してデータセットに対して繰り返しクエリを実行するのに役立ちます。 MLlib は、予測インテリジェンス、マーケティング目的の顧客セグメンテーション、感情分析などの一般的なビッグ データ機能のクラスタリング、分類、次元削減などの分野で使用されます。 MLlib は、ネットワーク セキュリティで使用され、悪意のあるアクティビティの兆候がないかデータ パケットをリアルタイムで検査します。セキュリティ プロバイダーが新しい脅威を把握し、ハッカーに先手を打つと同時にクライアントをリアルタイムで保護するのに役立ちます。

ディープラーニング

TensorFlow は、業界全体で使用されている人気のディープラーニング フレームワークです。 TensorFlow は、CPU または GPU クラスターでの分散トレーニングをサポートします。この分散トレーニングにより、ユーザーは多数の深いレイヤーを持つ大量のデータに対してトレーニングを実行できます。

つい最近まで、Apache Spark で TensorFlow を使用する場合は、PySpark で TensorFlow に必要なすべての ETL を実行し、データを中間ストレージに書き込む必要がありました。そのデータは、実際のトレーニング プロセスのために TensorFlow クラスターにロードされます。このワークフローでは、ユーザーは ETL 用と TensorFlow の分散トレーニング用の 2 つの異なるクラスターを維持する必要がありました。複数のクラスターの実行と維持は、通常、面倒で時間がかかります。

以前のバージョンの Spark の DataFrames と RDD は、ランダム アクセスが制限されていたため、ディープラーニングには適していませんでした。プロジェクト Hydrogen を使用した Spark 3.0 では、ディープラーニング フレームワークのネイティブ サポートが追加されます。このアプローチにより、Spark クラスター上で MapReduce ベース以外のスケジューリングが可能になります。

インタラクティブ分析

Apache Spark は、SQL、R、Python など、Spark 以外の開発言語でサンプリングせずに探索クエリを実行できるほど高速です。 Spark は視覚化ツールを使用して複雑なデータを処理し、インタラクティブに視覚化します。構造化ストリーミングを備えた Spark は、Web 分析のライブ データに対して対話型クエリを実行し、Web 訪問者の現在のセッションに対して対話型クエリを実行できるようにします。

レコメンデーションシステム

長年にわたり、企業や消費者がオンライン ショッピング、オンライン エンターテイメント、その他多くの業界における劇的な変化に対応するにつれ、レコメンデーション システムは私たちの生活に多大な変化をもたらしてきました。実際、これらのシステムは、生産における AI の最も明らかな成功事例の 1 つです。多くの実際の使用例では、レコメンデーション システムは、NLP バックエンドとインターフェースされた会話型 AI またはチャットボットと組み合わせられ、関連情報を取得して有用な推論を生成します。

今日、多くの小売業者は、オンラインで購入して店舗で受け取る、カーブサイドピックアップ、セルフチェックアウト、スキャンアンドゴーなどの新しいビジネスモデルを採用しています。これらのモデルは、消費者にとってショッピングをより安全で便利なものにすることで、COVID-19パンデミック中に注目を集めるようになりました。 AI は、消費者の行動に影響を受け、またその逆も起こる、こうした成長を続けるデジタル トレンドにとって極めて重要です。NetAppは、消費者の高まる需要に応え、顧客体験を強化し、運用効率を改善し、収益を増やすために、エンタープライズ顧客と企業が機械学習とディープラーニングのアルゴリズムを使用して、より高速で正確な推奨システムを設計できるよう支援します。

推奨事項を提供するために使用される一般的な手法としては、協調フィルタリング、コンテンツ ベース システム、ディープラーニング レコメンデーション モデル (DLRM)、ハイブリッド手法などがあります。これまで、顧客は PySpark を利用して、推奨システムを作成するための協調フィルタリングを実装していました。 Spark MLlib は、DLRM が登場する以前から企業の間で非常に人気があったアルゴリズムである協調フィルタリング用の交代最小二乗法 (ALS) を実装しています。

自然言語処理

自然言語処理 (NLP) によって可能になる会話型 AI は、コンピューターが人間とコミュニケーションするのを支援する AI の分野です。 NLP は、スマート アシスタントやチャットボットから Google 検索や予測テキストまで、あらゆる業界のさまざまなユース ケースで普及しています。ある "ガートナー"予測によると、2022 年までに 70% の人々が会話型 AI プラットフォームを日常的に利用するようになるでしょう。人間と機械の間で質の高い会話をするには、応答が迅速で、インテリジェントで、自然な響きでなければなりません。

顧客は、NLP および自動音声認識 (ASR) モデルを処理およびトレーニングするために大量のデータを必要とします。また、エッジ、コア、クラウド間でデータを移動する必要があり、人間との自然なコミュニケーションを確立するために、数ミリ秒単位で推論を実行する能力も必要です。 NetApp AI と Apache Spark は、コンピューティング、ストレージ、データ処理、モデル トレーニング、微調整、および導入に最適な組み合わせです。

感情分析は、テキストから肯定的、否定的、または中立的な感情を抽出する NLP の研究分野です。感情分析には、発信者との会話におけるサポート センターの従業員のパフォーマンスを判断することから、適切な自動チャットボット応答を提供することまで、さまざまな使用例があります。また、四半期ごとの収益報告の電話会議における企業代表者と聴衆とのやり取りに基づいて、企業の株価を予測するためにも使用されています。さらに、感情分析を使用すると、ブランドが提供する製品、サービス、またはサポートに対する顧客の見解を判断することもできます。

私たちは "スパークNLP"図書館から "ジョン・スノー・ラボ"事前学習済みのパイプラインとBERT（Bidirectional Encoder Representations from Transformers）モデルをロードする。 "金融ニュースの感情"そして "フィンバート"トークン化、固有表現認識、モデルトレーニング、フィッティング、感情分析を大規模に実行します。 Spark NLP は、BERT、ALBERT、ELECTRA、XLNet、DistilBERT、RoBERTa、DeBERTa、XLM-RoBERTa、Longformer、ELMO、Universal Sentence Encoder、Google T5、MarianMT、GPT2 などの最先端のトランスフォーマーを提供する、実稼働中の唯一のオープンソース NLP ライブラリです。このライブラリは、Apache Spark をネイティブに拡張することで、Python や R だけでなく、JVM エコシステム (Java、Scala、Kotlin) でも大規模に動作します。

主要なAI、ML、DLのユースケースとアーキテクチャ

主要な AI、ML、DL のユースケースと方法論は、次のセクションに分けられます。

Spark NLPパイプラインとTensorFlow分散推論

次のリストには、さまざまな開発レベルでデータ サイエンス コミュニティに採用されている最も人気のあるオープン ソース NLP ライブラリが含まれています。

- "自然言語ツールキット（NLTK）" 。すべての NLP テクニックに対応する完全なツールキット。 2000年代初頭から維持されてきました。
- "テキストブロブ" 。NLTK と Pattern をベースに構築された、使いやすい NLP ツール Python API。
- "スタンフォード・コアNLP" 。スタンフォード NLP グループによって開発された Java の NLP サービスとパッケージ。

- "ゲンシム" 。Topic Modelling for Humans は、チェコデジタル数学ライブラリ プロジェクト用の Python スクリプトのコレクションとして始まりました。
- "スパシー" 。トランスフォーマー向け GPU アクセラレーションを備えた Python と Cython を使用したエンドツーエンドの産業用 NLP ワークフロー。
- "ファストテキスト" 。Facebook の AI 研究 (FAIR) ラボによって作成された、単語埋め込みの学習と文分類のための無料の軽量オープンソース NLP ライブラリです。

Spark NLP は、すべての NLP タスクと要件に対応する単一の統合ソリューションであり、実際の運用ユースケースでスケーラブルで高性能、かつ高精度な NLP ベースのソフトウェアを実現します。転移学習を活用し、研究や業界全体で最新の最先端のアルゴリズムとモデルを実装します。 Sparkは上記のライブラリを完全にサポートしていないため、Spark NLPは "スパークML"ミッションクリティカルな本番ワークフロー向けのエンタープライズグレードの NLP ライブラリとして、Spark の汎用インメモリ分散データ処理エンジンを活用します。そのアノテーターは、ルールベースのアルゴリズム、機械学習、TensorFlow を活用して、ディープラーニングの実装を強化します。これには、トークン化、レマタイズ化、ステミング、品詞タグ付け、固有表現認識、スペルチェック、感情分析などを含む一般的な NLP タスクが含まれます。

BERT (Bidirectional Encoder Representations from Transformers) は、NLP 用のトランスフォーマー ベースの機械学習手法です。事前トレーニングと微調整の概念を普及させました。 BERT のトランスフォーマー アーキテクチャは機械翻訳から生まれたもので、リカレント ニューラル ネットワーク (RNN) ベースの言語モデルよりも長期的な依存関係をより適切にモデル化します。また、すべてのトークンのランダムな 15% がマスクされ、モデルがそれを予測することで真の双方向性を実現するマスク言語モデリング (MLM) タスクも導入されました。

金融感情分析は、専門的な言語とその分野のラベル付きデータが不足しているため、困難です。 FinBERT は、事前学習済みのBERTに基づく言語モデルであり、ドメイン適応された。 "ロイターTRC2" 、金融コーパス、ラベル付きデータで微調整された（ "金融フレーズバンク" ）を使用して金融センチメントを分類します。研究者らはニュース記事から金融用語を含む4,500の文を抽出した。その後、金融のバックグラウンドを持つ 16 人の専門家と修士課程の学生が、その文章を肯定的、中立的、否定的と分類しました。 FinBERTと他の2つの事前学習済みパイプラインを使用して、2016年から2020年までのNASDAQ上場企業トップ10社の決算説明会の記録の感情を分析するためのエンドツーエンドのSparkワークフローを構築しました。 "説明文書DL") を Spark NLP から取得します。

Spark NLP の基盤となるディープラーニング エンジンは TensorFlow です。これは、モデルの構築を容易にし、どこでも堅牢な ML を生成でき、研究のための強力な実験を可能にする、エンドツーエンドのオープンソース 機械学習プラットフォームです。したがって、Sparkでパイプラインを実行する場合 `yarn cluster`モードでは、基本的に、1 つのマスター ノードと複数のワーカー ノード、およびクラスターにマウントされたネットワーク接続ストレージにわたって、データとモデルの並列化を伴う分散 TensorFlow を実行していました。

Horovod分散トレーニング

MapReduce 関連のパフォーマンスに関するコア Hadoop 検証は、TeraGen、TeraSort、TeraValidate、および DFSIO (読み取りと書き込み) を使用して実行されます。 TeraGenとTeraSortの検証結果は、 "Hadoop向けNetApp Eシリーズソリューション" AFFの「ストレージ階層化」セクションを参照してください。

お客様のご要望に基づき、Spark を使用した分散トレーニングは、さまざまなユースケースの中でも最も重要なものの 1 つであると考えています。この文書では、 "SparkのHovorod" NetApp All Flash FAS (AFF) ストレージ コントローラー、 Azure NetApp Files、 StorageGRIDを使用して、 NetApp のオンプレミス、クラウド ネイティブ、ハイブリッド クラウド ソリューションで Spark のパフォーマンスを検証します。

Horovod on Spark パッケージは、Horovod の便利なラッパーを提供します。これにより、Spark クラスターでの分散トレーニング ワークロードの実行が簡単になり、データ処理、モデル トレーニング、モデル評価がすべてトレーニング データと推論データが存在する Spark で実行される、緊密なモデル設計ループが可能に

なります。

Spark で Horovod を実行するための API には、高レベルの Estimator API と低レベルの Run API の 2 つがあります。どちらも Spark 実行プログラムで Horovod を起動するために同じ基本メカニズムを使用しますが、Estimator API はデータ処理、モデル トレーニング ループ、モデル チェックポイント、メトリック収集、分散トレーニングを抽象化します。 Horovod Spark Estimators、TensorFlow、Kerasを使用して、エンドツーエンドのデータ準備と分散トレーニングワークフローを構築しました。 "Kaggle Rossmann ストア売上"競争。

脚本 `keras_spark_horovod_rossmann_estimator.py`セクションをご覧ください"それぞれの主要なユースケース向けの Python スクリプト。"3 つの部分から構成されます。

- 最初の部分では、Kaggle によって提供され、コミュニティによって収集された CSV ファイルの初期セットに対してさまざまなデータ前処理手順を実行します。入力データは、 `Validation`サブセットとテストデータセット。
- 2 番目の部分では、対数シグモイド活性化関数と Adam オプティマイザーを備えた Keras ディープ ニューラル ネットワーク (DNN) モデルを定義し、Spark 上の Horovod を使用してモデルの分散トレーニングを実行します。
- 3 番目の部分では、検証セット全体の平均絶対誤差を最小化する最適なモデルを使用して、テスト データセットの予測を実行します。次に、出力 CSV ファイルを作成します。

セクションを参照"機械学習"さまざまな実行時間の比較結果。

Keras を使用したマルチワーカー ディープラーニングによる CTR 予測

ML プラットフォームとアプリケーションの最近の進歩により、大規模な学習に多くの注目が集まっています。クリックスルー率 (CTR) は、オンライン広告の表示回数 100 回あたりの平均クリックスルー数 (パーセントで表されます) として定義されます。これは、デジタル マーケティング、小売、電子商取引、サービス プロバイダーなど、さまざまな業界の垂直分野やユース ケースで重要な指標として広く採用されています。CTRと分散トレーニングのパフォーマンス結果の適用の詳細については、"CTR予測パフォーマンスのためのディープラーニングモデル"セクション。

この技術レポートでは、 "Criteo テラバイトクリックログデータセット" (TR-4904 を参照) Keras を使用してマルチワーカー分散ディープラーニングを実施し、Deep and Cross Network (DCN) モデルを含む Spark ワークフローを構築し、ログ損失エラー関数の観点からそのパフォーマンスをベースライン Spark ML ロジスティック回帰モデルと比較します。 DCN は、制限された次数の有効な特徴相互作用を効率的にキャプチャし、高度に非線形な相互作用を学習し、手動の特徴エンジニアリングや徹底的な検索を必要とせず、計算コストが低くなります。

Web 規模の推奨システムのデータは大部分が離散的かつカテゴリ化されているため、特徴空間が大きくまばらになり、特徴の探索が困難になります。このため、ほとんどの大規模システムはロジスティック回帰などの線形モデルに制限されています。ただし、頻繁に予測される特徴を識別し、同時に目に見えない、またはまれなクロス特徴を探索することが、適切な予測を行うための鍵となります。線形モデルはシンプルで解釈しやすく、拡張も容易ですが、表現力には限界があります。

一方、クロス特徴はモデルの表現力の向上に重要であることが示されています。残念ながら、このような特徴を識別するには、多くの場合、手動の特徴エンジニアリングや徹底的な検索が必要になります。目に見えない機能の相互作用を一般化することは、多くの場合困難です。 DCN のようなクロスニューラルネットワークを使用すると、特徴の交差を明示的に自動的に適用することで、タスク固有の特徴エンジニアリングを回避できます。クロス ネットワークは複数のレイヤーで構成されており、相互作用の最高度はレイヤーの深さによって決定されると考えられます。各レイヤーは、既存の相互作用に基づいて高次の相互作用を生成し、前のレイヤーからの相互作用を維持します。

ディープ ニューラル ネットワーク (DNN) は、機能間の非常に複雑な相互作用をキャプチャできる可能性を秘めています。ただし、DCN と比較すると、ほぼ 1 桁多くのパラメータが必要となり、クロス フィーチャを明示的に形成できず、一部の種類のフィーチャの相互作用を効率的に学習できない可能性があります。クロスネットワークはメモリ効率が高く、実装が簡単です。クロスコンポーネントと DNN コンポーネントを共同でトレーニングすることで、予測機能のインタラクションを効率的にキャプチャし、Criteo CTR データセットで最先端のパフォーマンスを実現します。

DCN モデルは、埋め込みおよびスタッキング レイヤーから始まり、クロス ネットワークとディープ ネットワークが並列に続きます。次に、2 つのネットワークからの出力を結合する最終結合レイヤーが続きます。入力データは、スパースな特徴と密な特徴を持つベクトルにすることができます。 Sparkでは、ライブラリには次のような型が含まれています `SparseVector`。したがって、ユーザーはこれら 2 つを区別し、それぞれの関数やメソッドを呼び出すときに注意することが重要です。 CTR予測のようなWebスケールのレコメンデーションシステムでは、入力は主にカテゴリ特徴であり、例えば `'country=usa'`。このような特徴は、多くの場合、ワンホットベクトルとしてエンコードされます。たとえば、 `'[0,1,0, …]'` 。ワンホットエンコーディング（OHE） `SparseVector`常に変化し、増え続ける語彙を持つ現実世界のデータセットを扱うときに役立ちます。例を修正しました "ディープクリック率"大規模な語彙を処理し、DCN の埋め込みおよびスタッキング層に埋め込みベクトルを作成します。

その "Criteo ディスプレイ広告データセット"広告のクリック率を予測します。 13 個の整数特徴と 26 個のカテゴリ特徴があり、各カテゴリは高いカーディナリティを持っています。このデータセットでは、入力サイズが大きいため、logloss の 0.001 の改善は実質的に重要です。大規模なユーザーベースに対する予測精度のわずかな向上は、企業の収益の大幅な増加につながる可能性があります。データセットには、7 日間の 11 GB のユーザー ログが含まれており、これは約 4,100 万件のレコードに相当します。 Sparkを使用しました `dataFrame.randomSplit()function`データをランダムに分割し、トレーニング用（80%）、クロス検証用（10%）、残りの10%をテスト用にします。

DCN は、Keras を使用して TensorFlow に実装されました。 DCN を使用してモデル トレーニング プロセスを実装する場合、主なコンポーネントは 4 つあります。

- *データの処理と埋め込み。*実数値の特徴は、対数変換を適用することによって正規化されます。カテゴリ特徴量の場合、特徴量を6×(カテゴリカーディナリティ)1/4次元の稠密ベクトルに埋め込みます。すべての埋め込みを連結すると、次元 1026 のベクトルが生成されます。
- 最適化。 Adam オプティマイザーを使用してミニバッチ確率最適化を適用しました。バッチサイズは 512 に設定されました。ディープ ネットワークにバッチ正規化が適用され、勾配クリップ ノルムは 100 に設定されました。
- 正規化。 L2 正則化またはドロップアウトは効果的ではないことが判明したため、早期停止を使用しました。
- *ハイパーパラメータ*隠し層の数、隠し層のサイズ、初期学習率、およびクロス層の数に対するグリッド検索に基づいて結果を報告します。隠し層の数は 2 ～ 5 で、隠し層のサイズは 32 ～ 1024 でした。 DCN の場合、クロス レイヤーの数は 1 ～ 6 でした。初期学習率は 0.0001 から 0.001 まで 0.0001 ずつ増分して調整されました。すべての実験では、トレーニング ステップ 150,000 で早期停止が適用され、それを超えるとオーバーフィッティングが発生し始めました。

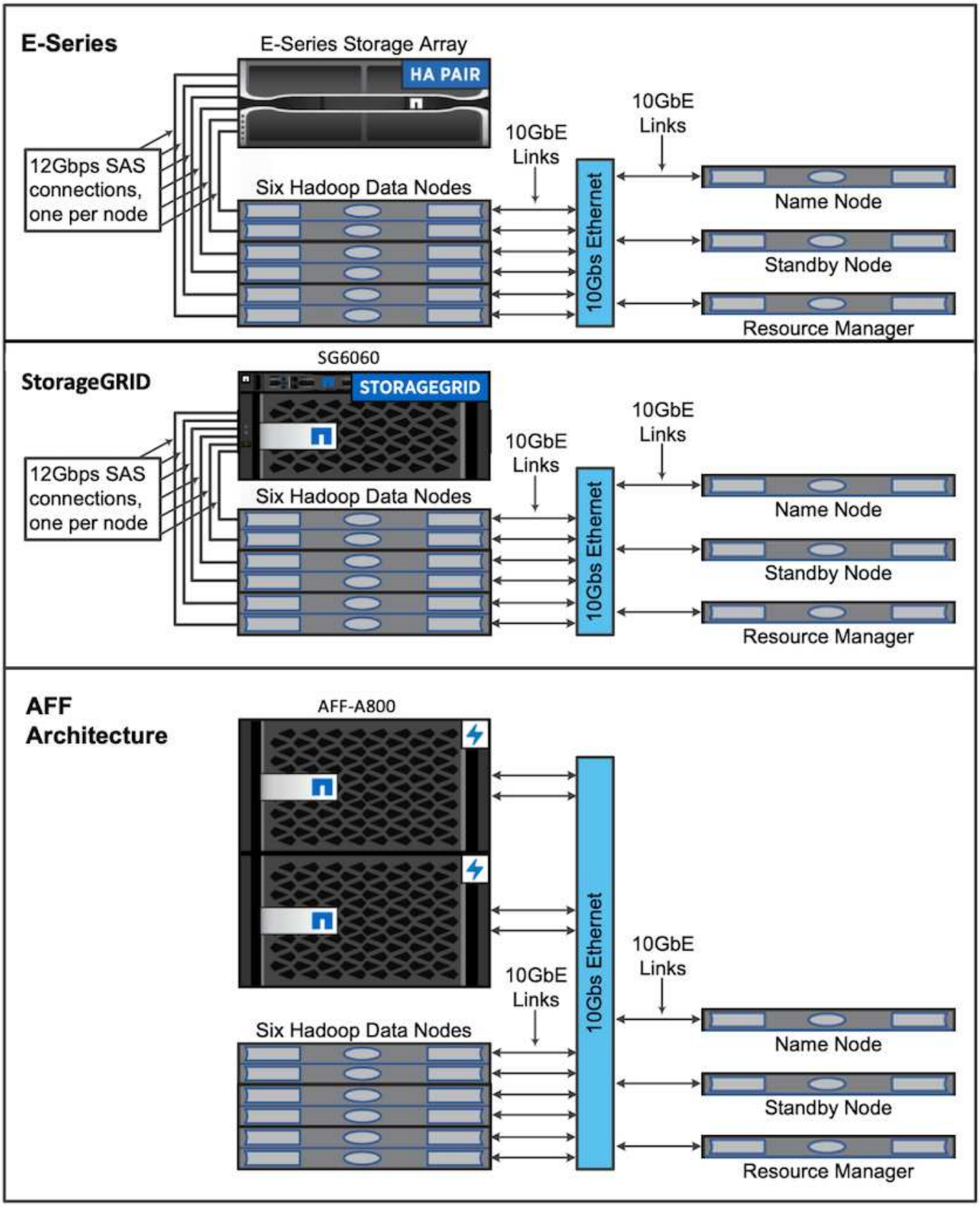
DCNに加えて、CTR予測のための他の一般的なディープラーニングモデルもテストしました。 "ディープFM" 、 "自動挿入" 、 そして "DCN v2"。

検証に使用されるアーキテクチャ

この検証では、 AFF-A800 HA ペアを持つ 4 つのワーカー ノードと 1 つのマスター ノードを使用しました。すべてのクラスター メンバーは 10GbE ネットワーク スイッチを介して接続されていました。

このNetApp Spark ソリューションの検証では、E5760、E5724、 AFF-A800 という 3 つの異なるストレージ

コントローラを使用しました。 E シリーズ ストレージ コントローラは、12Gbps SAS 接続で 5 つのデータノードに接続されていました。 AFF HA ペア ストレージ コントローラは、10GbE 接続を介してエクスポートされた NFS ボリュームを Hadoop ワーカー ノードに提供します。 Hadoop クラスター メンバーは、E シリーズ、 AFF、およびStorageGRID Hadoop ソリューション内の 10GbE 接続を介して接続されていました。



テスト結果

TeraGen ベンチマーク ツールの TeraSort および TeraValidate スクリプトを使用して、E5760、E5724、およびAFF-A800 構成での Spark パフォーマンス検証を測定しました。さらに、Spark NLP パイプラインと TensorFlow 分散トレーニング、Horovod 分散トレーニング、DeepFM による CTR 予測のための Keras を使用したマルチワーカー ディープラーニングという 3 つの主要なユース ケースがテストされました。

E シリーズとStorageGRID の両方の検証では、Hadoop レプリケーション ファクター 2 を使用しました。AFF検証では、1 つのデータ ソースのみを使用しました。

次の表は、Spark パフォーマンス検証のハードウェア構成を示しています。

タイプ	**Hadoop**ワーカーノード	ドライブ タイプ	ノードあたりのドライブ数	ストレージ コントローラ
SG6060	4	SAS	12	単一の高可用性（HA）ペア
E5760	4	SAS	60	単一のHAペア
E5724	4	SAS	24	単一のHAペア
AFF800	4	SSD	6	単一のHAペア

次の表にソフトウェア要件を示します。

ソフトウェア	**version**
RHEL	7.9
OpenJDK ランタイム環境	1.8.0
OpenJDK 64 ビット サーバー VM	25.302
ギット	2.24.1
GCC/G++	11.2.1
スパーク	3.2.1
パイスパーク	3.1.2
スパークNLP	3.4.2
テンソルフロー	2.9.0
ケラス	2.9.0
ホロヴォド	0.24.3

金融感情分析

我々は出版した"TR-4910: NetApp AIによる顧客コミュニケーションからの感情分析"エンドツーエンドの会話型AIパイプラインを構築した。 "NetApp DataOps ツールキット" 、 AFFストレージ、 NVIDIA DGX システム。パイプラインは、DataOpsツールキットを活用して、バッチオーディオ信号処理、自動音声認識（ASR）、転移学習、感情分析を実行します。 "NVIDIA Riva SDK" 、そして "タオフレームワーク"。感情分析のユ

ースケースを金融サービス業界に拡大し、SparkNLP ワークフローを構築し、名前付きエンティティの認識などのさまざまな NLP タスク用に 3 つの BERT モデルをロードし、NASDAQ トップ 10 企業の四半期決算発表の文章レベルの感情を取得しました。

次のスクリプト `sentiment_analysis_spark. py`FinBERT モデルを使用して HDFS 内のトランスクリプトを処理し、次の表に示すように、肯定的、中立的、否定的な感情カウントを生成します。

```
-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
```

次の表は、2016 年から 2020 年までの NASDAQ トップ 10 企業の収益報告の文章レベルの感情分析を示しています。

感情の数と割合	全**10**社	**AAPL**	**AMD**	アマゾン	**CSCO**	グーグル	**INTC**	マイクロソフト	**NVDA**
陽性数	7447	1567	743	290	682	826	824	904	417
中立カウント	64067	6856	7596	5086	6650	5914	6099	5715	6189
マイナスカウント	1787	253	213	84	189	97	282	202	89
分類されていない数	196	0	0	76	0	0	0	1	0
（合計数）	73497	8676	8552	5536	7521	6837	7205	6822	6695

パーセンテージで見ると、CEO や CFO が話した文章のほとんどは事実に基づいており、したがって中立的な感情を伝えています。決算説明会では、アナリストは肯定的または否定的な感情を伝える可能性のある質問をします。ネガティブまたはポジティブな感情が、取引当日または翌日の株価にどのような影響を与えるかを定量的にさらに調査する価値があります。

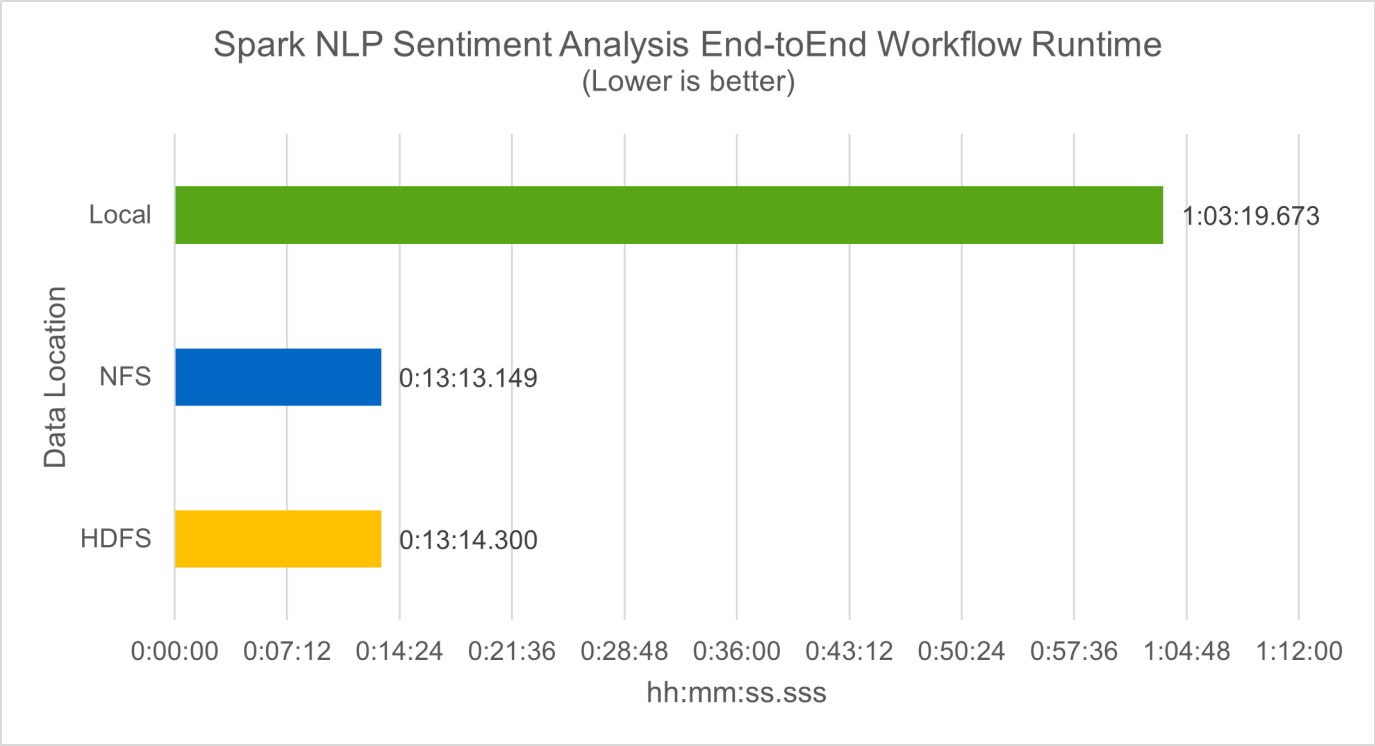
次の表は、NASDAQ 上位 10 社の文章レベルの感情分析をパーセンテージで示したものです。

感情の割合	全10社	AAPL	AMD	アマゾン	CSCO	グーグル	INTC	マイクロソフト	NVDA
ポジティブ	10.13%	18.06%	8.69%	5.24%	9.07%	12.08%	11.44%	13.25%	6.23%
中性	87.17%	79.02%	88.82%	91.87%	88.42%	86.50%	84.65%	83.77%	92.44%
ネガティブ	2.43%	2.92%	2.49%	1.52%	2.51%	1.42%	3.91%	2.96%	1.33%
未分類	0.27%	0%	0%	1.37%	0%	0%	0%	0.01%	0%

ワークフロー実行時間に関しては、4.78倍の大幅な改善が見られました。 `local`モードを HDFS の分散環境に移行し、NFS を活用することでさらに 0.14% の改善が実現しました。

```
-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
```

次の図に示すように、データとモデルの並列処理により、データ処理と分散 TensorFlow モデル推論速度が向上しました。ワークフローのボトルネックは事前トレーニング済みモデルのダウンロードであるため、NFS にデータを配置すると実行時間がわずかに改善されました。トランスクリプト データセットのサイズを増やすと、NFS の利点がより明らかになります。



Horovodパフォーマンスによる分散トレーニング

次のコマンドは、Sparkクラスタ内の実行時情報とログファイルを単一のコマンドで生成しました。 `master` それぞれ `1` つのコアを持つ `160` 個のエグゼキュータを持つノード。メモリ不足エラーを回避するために、実行メモリは `5 GB` に制限されました。セクションを参照"主要なユースケースごとの `Python` スクリプト"データ処理、モデルトレーニング、モデル精度計算の詳細については、 `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
```

10 回のトレーニング エポックで実行された結果は次のとおりです。

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

入力データの処理、DNN モデルのトレーニング、精度の計算、TensorFlow チェックポイントと予測結果の CSV ファイルの生成には 43 分以上かかりました。トレーニング エポックの数を 10 に制限しましたが、実際には、十分なモデル精度を確保するために 100 に設定されることが多いです。トレーニング時間は通常、エポック数に比例して増加します。

次に、クラスタ内で利用可能な4つのワーカーノードを使用して、同じスクリプトを実行しました。 `yarn` HDFS 内のデータを使用するモード:

```
(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
```

結果として、実行時間は次のように改善されました。

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

HorovodのモデルとSparkのデータ並列処理により、実行速度が5.29倍向上しました。 `yarn`対 `local`10 回のトレーニング エポックを含むモード。これは次の図に凡例とともに示されています。 `HDFS`そして `Local`。基盤となる TensorFlow DNN モデルのトレーニングは、利用可能な場合は GPU を使用してさらに高速化できます。私たちはこのテストを実施し、その結果を今後の技術レポートで公開する予定です。

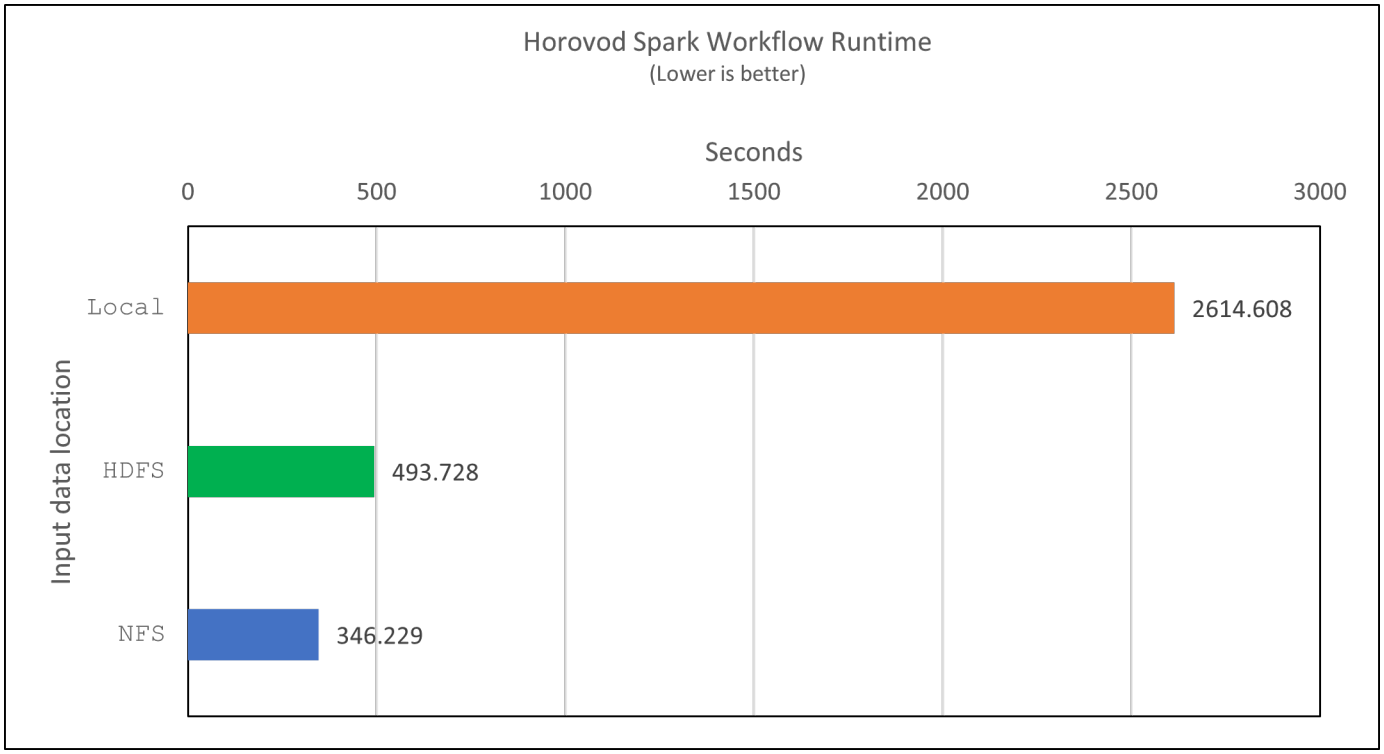
次のテストでは、NFS と HDFS にある入力データの実行時間を比較しました。 AFF A800のNFSボリュームは、 `/sparkdemo/horovod` Spark クラスター内の 5 つのノード (マスター 1 つ、ワーカー 4 つ) にわたって。前回のテストと同様のコマンドを実行したが、 `--data- dir`パラメータはNFSマウントを指すようになりました:

```
(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
```

NFS を使用した場合の実行時間は次のようになりました。

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

次の図に示すように、さらに 1.43 倍の高速化が実現しました。したがって、 NetAppオールフラッシュ ストレージをクラスターに接続することで、顧客は Horovod Spark ワークフローの高速データ転送と配信のメリットを享受でき、単一ノードで実行する場合と比較して 7.55 倍の高速化を実現できます。



CTR予測パフォーマンスのためのディープラーニングモデル

CTR を最大化するように設計されたレコメンデーション システムでは、低次から高次まで数学的に計算でき

るユーザー ビヘイビアの背後にある高度な機能の相互作用を学習する必要があります。低次の機能と高次の機能の相互作用は、どちらか一方に偏ることなく、優れたディープラーニング モデルにとって同等に重要です。因数分解マシン ベースのニューラル ネットワークである Deep Factorization Machine (DeepFM) は、推奨用の因数分解マシンと特徴学習用のディープラーニングを新しいニューラル ネットワーク アーキテクチャに組み合わせています。

従来の因数分解マシンは、特徴間の潜在ベクトルの内積としてペアワイズ特徴相互作用をモデル化し、理論的には高次の情報を取得できますが、実際には、機械学習の専門家は、計算とストレージの複雑さが高いため、通常、2 次特徴相互作用のみを使用します。 Googleのようなディープニューラルネットワークの変種 "ワイド＆ディープモデル"一方、線形ワイドモデルとディープモデルを組み合わせることで、ハイブリッドネットワーク構造における洗練された機能の相互作用を学習します。

このワイド＆ディープ モデルには 2 つの入力があります。1 つは基礎となるワイド モデル用、もう 1 つはディープ モデル用です。後者の部分では依然として専門家による特徴エンジニアリングが必要であり、そのためこの手法を他のドメインに一般化することは困難です。ワイド＆ディープ モデルとは異なり、DeepFM は、ワイド部分とディープ部分が同じ入力と埋め込みベクトルを共有するため、特徴エンジニアリングなしで生の特徴を使用して効率的にトレーニングできます。

まずCriteo `train.txt` （11GB）ファイルをCSVファイルに `ctr_train.csv`NFSマウントに保存`/sparkdemo/tr-4570-data`使用して `run_classification_criteo_spark.py`セクションから"それぞれの主要なユースケース向けの Python スクリプト。"このスクリプト内では、関数 `process_input_file`タブを削除して挿入するためのいくつかの文字列メソッドを実行します ','`区切り文字として '\n'`改行として。元のファイルのみを処理する必要があることに注意してください `train.txt`一度実行すると、コード ブロックがコメントとして表示されます。

さまざまなDLモデルの以下のテストでは、 `ctr_train.csv`入力ファイルとして。その後のテスト実行では、入力CSVファイルは、次のフィールドを含むスキーマを持つSpark DataFrameに読み込まれました。 `'label'`整数密な特徴 `['I1', 'I2', 'I3', …, 'I13']`、およびスパースな特徴 `['C1', 'C2', 'C3', …, 'C26']`。次の `spark-submit`コマンドは入力 CSV を受け取り、クロス検証のために 20% 分割して DeepFM モデルをトレーニングし、10 回のトレーニング エポック後に最適なモデルを選択して、テストセットでの予測精度を計算します。

```
(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
```

データファイルは `ctr_train.csv`11GBを超える場合は、十分な容量を設定する必要があります`spark.driver.maxResultSize`エラーを回避するには、データセットのサイズよりも大きくする必要があります。

```
 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()
```

上記において `SparkSession.builder`有効にした設定 "アパッチアロー"はSpark DataFrameをPandas DataFrameに変換します。 `df.toPandas()`方法。

```
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.
```

ランダムに分割した後、トレーニング データセットには 3600 万行以上、テスト セットには 900 万のサンプルが含まれます。

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

この技術レポートは GPU を使用せずに CPU テストに焦点を当てているため、適切なコンパイラ フラグを使用して TensorFlow をビルドすることが不可欠です。このステップでは、GPU アクセラレーション ライブラリの呼び出しを回避し、TensorFlow の Advanced Vector Extensions (AVX) と AVX2 命令を最大限に活用します。これらの機能は、ベクトル化された加算、フィードフォワード内の行列乗算、バックプロパゲーション DNN トレーニングなどの線形代数計算用に設計されています。 256 ビット浮動小数点 (FP) レジスタを使用する AVX2 で利用可能な Fused Multiply Add (FMA) 命令は、整数コードとデータ型に最適であり、最大 2 倍の速度向上をもたらします。 FP コードとデータ型の場合、AVX2 は AVX よりも 8% 高速化します。

```
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.
```

ソースからTensorFlowを構築するには、 NetAppは以下を使用することを推奨しています。 "バゼル" 。私たちの環境では、シェルプロンプトで次のコマンドを実行してインストールしました。 `dnf` 、 `dnf-plugins` 、そしてバゼル。

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

ビルド プロセス中に C++17 機能を使用するには、GCC 5 以降を有効にする必要があります。これは、RHEL によって Software Collections Library (SCL) とともに提供されます。次のコマンドはインストールします `devtoolset`RHEL 7.9 クラスター上の GCC 11.2.1:

```
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
```

最後の2つのコマンドは `devtoolset-11`、使用する `/opt/rh/devtoolset-11/root/usr/bin/gcc` （GCC 11.2.1）。また、 `git`バージョンは 1.8.3 より大きいです (RHEL 7.9 に付属)。こちらを参照してください "記事"更新用 `git`2.24.1 まで。

最新の TensorFlow マスター リポジトリのクローンがすでに作成されているものと想定します。次に、 `workspace`ディレクトリ `WORKSPACE`AVX、AVX2、FMA を使用してソースから TensorFlow をビルドするためのファイル。実行 `configure`ファイルを開き、正しい Python バイナリの場所を指定します。 "CUDA" GPU を使用していないため、テストでは無効になっています。あ `.bazelrc`ファイルは設定に従って生成されます。さらに、ファイルを編集して設定しました `build --define=no_hdfs_support=false`HDFS サポートを有効にします。参照 `.bazelrc`セクション内"主要なユースケースごとのPythonスクリプト"設定とフラグの完全なリストについては、こちらをご覧ください。

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

正しいフラグを使用して TensorFlow をビルドした後、次のスクリプトを実行して Criteo ディスプレイ広告データセットを処理し、DeepFM モデルをトレーニングし、予測スコアから受信者操作特性曲線の下の領域 (ROC AUC) を計算します。

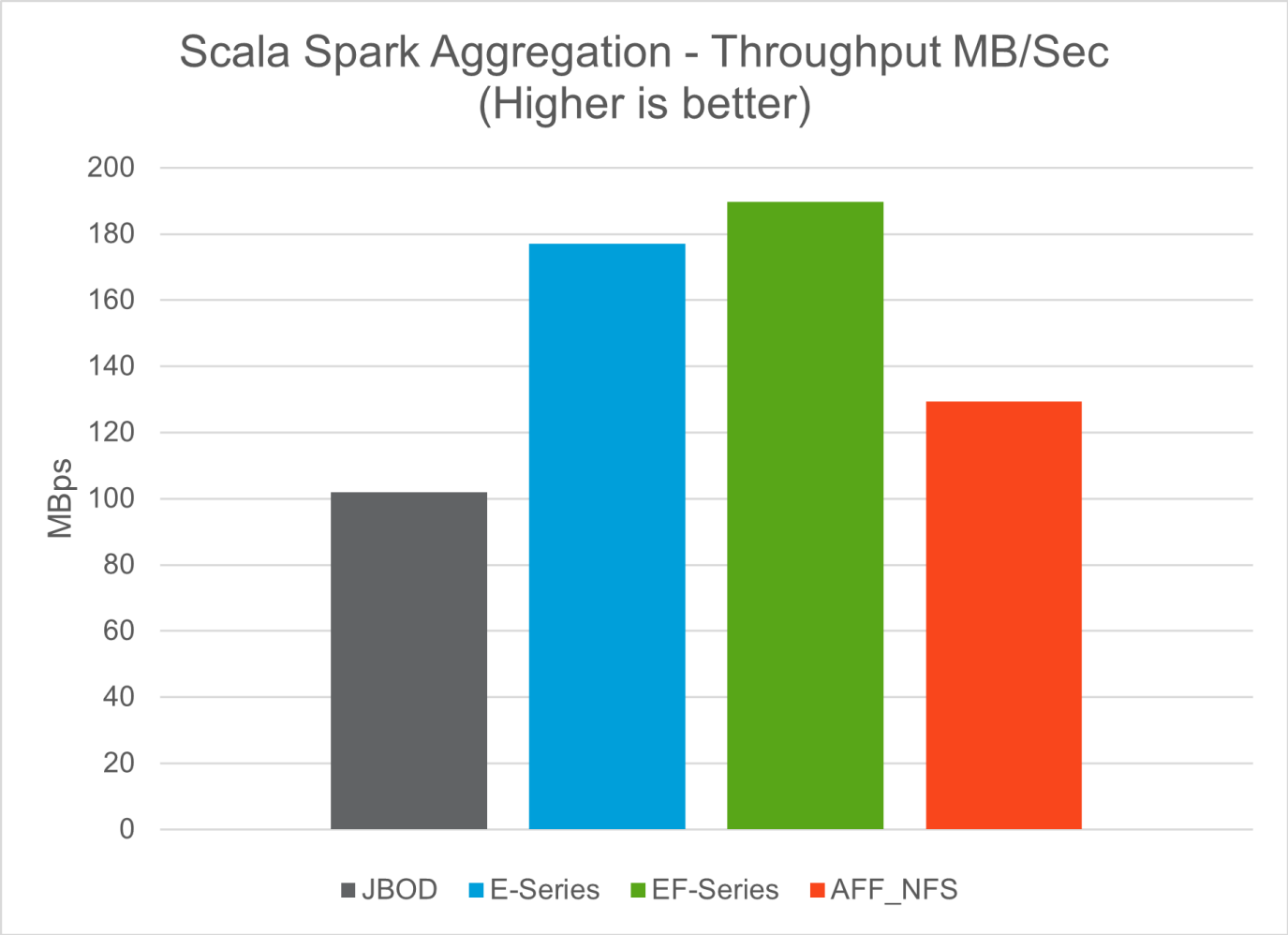
```
(base) [root@n138 examples]# ~/anaconda3/bin/spark-submit
--master yarn
--executor-memory 15g
--executor-cores 1
--num-executors 160
/sparkusecase/DeepCTR/examples/run_classification_criteo_spark.py
--data-dir file:///sparkdemo/tr-4570-data
> . /run_classification_criteo_spark_nfs.log 2>&1
```

10 回のトレーニング エポック後、テスト データセットの AUC スコアを取得しました。

```
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
```

以前のユースケースと同様に、Spark ワークフロー ランタイムをさまざまな場所に存在するデータと比較し

ました。次の図は、Spark ワークフロー ランタイムのディープラーニング CTR 予測の比較を示しています。



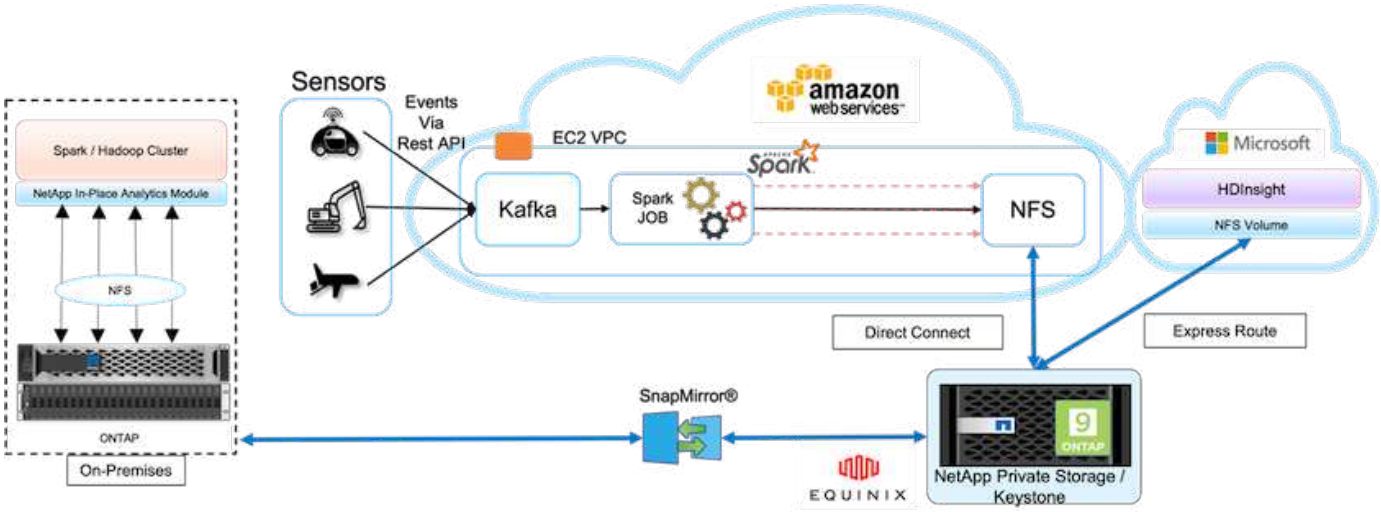
ハイブリッドクラウドソリューション

最新のエンタープライズ データ センターは、オンプレミスや複数のパブリック クラウドで、一貫した運用モデルを備えた継続的なデータ管理プレーンを通じて複数の分散インフラストラクチャ環境を接続するハイブリッド クラウドです。ハイブリッド クラウドを最大限に活用するには、データ変換やアプリケーションのリファクタリングを行わずに、オンプレミス環境とマルチクラウド環境間でデータをシームレスに移動できる必要があります。

顧客は、データ保護などのユースケースのためにセカンダリ ストレージをクラウドに移行するか、アプリケーション開発や DevOps などのビジネス クリティカル度の低いワークロードをクラウドに移行するかのいずれかの方法でハイブリッド クラウドの取り組みを開始すると述べています。その後、より重要なワークロードに移行します。最も人気のあるハイブリッド クラウド ワークロードには、Web およびコンテンツのホスティング、DevOps およびアプリケーション開発、データベース、分析、コンテナ化されたアプリなどがあります。これまで、企業の AI プロジェクトの複雑さ、コスト、リスクは、実験段階から実稼働段階までの AI 導入を妨げてきました。

NetApp のハイブリッド クラウド ソリューションを利用すると、お客様は、分散環境全体のデータとワークフロー管理を単一のコントロール パネルで統合されたセキュリティ、データ ガバナンス、コンプライアンス ツールから恩恵を受けられると同時に、消費量に基づいて総所有コストを最適化できます。次の図は、顧客の

ビッグデータ分析データにマルチクラウド接続を提供する役割を担うクラウド サービス パートナーのソリューション例です。



このシナリオでは、さまざまなソースから AWS に受信された IoT データは、 NetApp Private Storage (NPS) の中央の場所に保存されます。 NPS ストレージは、AWS および Azure にある Spark または Hadoop クラスターに接続され、複数のクラウドで実行されるビッグ データ分析アプリケーションが同じデータにアクセスできるようになります。このユースケースの主な要件と課題は次のとおりです。

- 顧客は複数のクラウドを使用して同じデータに対して分析ジョブを実行したいと考えています。
- データは、さまざまなセンサーやハブを介して、オンプレミスやクラウド環境などのさまざまなソースから受信する必要があります。
- ソリューションは効率的かつ費用対効果の高いものでなければなりません。
- 主な課題は、さまざまなオンプレミス環境とクラウド環境間でハイブリッド分析サービスを提供する、コスト効率が高く効率的なソリューションを構築することです。

当社のデータ保護およびマルチクラウド接続ソリューションは、複数のハイパースケーラーにクラウド分析アプリケーションを導入する際の問題点を解決します。上の図に示すように、センサーからのデータは Kafka を介してストリーミングされ、AWS Spark クラスターに取り込まれます。データは、Equinix データセンター内のクラウド プロバイダーの外部にある NPS にある NFS 共有に保存されます。

NetApp NPS は、それぞれ Direct Connect 接続と Express Route 接続を介して Amazon AWS と Microsoft Azure に接続されているため、お客様は In-Place Analytics モジュールを活用して、Amazon と AWS の両方の分析クラスターからデータにアクセスできます。その結果、オンプレミスとNPSストレージの両方でONTAPソフトウェアが実行されるため、 "SnapMirror" NPS データをオンプレミス クラスターにミラーリングし、オンプレミスと複数のクラウドにわたるハイブリッド クラウド分析を提供できます。

最高のパフォーマンスを得るために、 NetAppでは通常、複数のネットワーク インターフェイスと直接接続または高速ルートを使用してクラウド インスタンスからデータにアクセスすることを推奨しています。弊社には他にもデータムーバーソリューションがあります。 "XCP"そして "BlueXPコピーと同期"顧客がアプリケーション対応型で安全かつコスト効率に優れたハイブリッド クラウド Spark クラスターを構築できるように支援します。

主要なユースケースごとの **Python** スクリプト

次の 3 つの Python スクリプトは、テストされた 3 つの主要なユース ケースに対応して

います。まず 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}")
```

2番目のスクリプトは 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()
```

3番目のスクリプトは `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))
```

まとめ

このドキュメントでは、ビッグ データ、最新の分析、AI、ML、DL に関連する Apache Spark アーキテクチャ、顧客のユースケース、 NetAppストレージ ポートフォリオについて説明します。業界標準のベンチマーク ツールと顧客の要求に基づくパフォーマンス検証テストでは、 NetApp Spark ソリューションはネイティブ Hadoop システムに比べて優れたパフォーマンスを示しました。このレポートで紹介されている顧客のユースケースとパフォーマンス結果を組み合わせることで、導入に適した Spark ソリューションを選択するのに役立ちます。

詳細情報の入手方法

このTRでは次の資料を参照しています。

- Apache Spark のアーキテクチャとコンポーネント

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

- Apache Sparkのユースケース

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

- スパークNLP

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

- バート

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

- 広告クリック予測のためのディープネットワークとクロスネットワーク

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

- FlexGroup

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

- ストリーミングETL

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

- Hadoop向けNetApp Eシリーズソリューション

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

- NetApp の最新データ分析ソリューション

  "データ分析ソリューション"

- 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コピーと同期

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

- データオプスツールキット

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