기술 가이드

마이그레이션 가이드: 하둡에서 Databricks로

데이터 아키텍처 현대화

databricks

서문

하둡은 분산된 데이터 스토리지와 처리에 사용하는 오픈 소스 소프트웨어 프로젝트로 구성된 에코시스템입니다. Databricks는 클라우드 및 Apache Spark™ 기반 빅데이터 분석 서비스로, 대개 Amazon Web Services(AWS), Google Cloud Platform(GCP), Microsoft Azure에서 제공됩니다. Databricks는 Apache Spark를 개발하였으며, Spark의 최적화된 버전을 기반으로 하는 관리형 클라우드 플랫폼입니다. Databricks 플랫폼은 데이터 엔지니어링, 데이터 사이언스, BI 워크로드를 위한 협업, 스트리밍 및 배치 데이터 처리에 초점을 맞춘 개발 환경을 제공합니다. 또한, 노트북 환경을 제공하는 것은 물론이고 코드 개발과 테스트, 배포에 흔히 사용하는 다양한 IDE와도 통합됩니다. 이 가이드에서는 하둡에서 Databricks로 마이그레이션하는 방법을 설명합니다. 이 가이드에서 설명하는 모든 기능은 정식으로 출시(GA)되었고 프로덕션을 지원합니다.

이 가이드와 함께 참고할만한 Databricks 노트북 5개도 있습니다. 이 문서의 여러 섹션에 노트북 링크를 포함하였습니다. 모든 노트북이 들어 있는 폴더 다운로드 경로는 다음과 같습니다.

AWS AZURE

목차

챕터 01

개요

레이크하우스 아키텍처
하둡에서 Databricks로 구성 요소 매핑

databricks

챕터 1: 개요

레이크하우스 아키텍처

데이터 전략의 앞날을 위해 계획을 세우는 하둡 사용자들은 대부분 기존 하둡 플랫폼에 비용, 복잡성과 실용성 면에서 불만이 있습니다. 온프레미스 하둡 플랫폼은 데이터 사이언스 기능이 부족하고 운영 비용이 많이 드는 데다, 민첩성과 성능이 낮아 비즈니스 가치를 제공하지 못했습니다. 그래서 기업에서는 기존 하둡 플랫폼을 클라우드 데이터 플랫폼으로 현대화하고자 합니다.

Databricks 레이크하우스 플랫폼은 모든 데이터, 분석 및 AI 워크로드를 통합하는 클라우드 네이티브 플랫폼입니다. 레이크하우스 플랫폼은 데이터 레이크와 데이터 웨어하우스의 장점만을 결합했습니다. 데이터 웨어하우스에서 일반적으로 제공하는 데이터 관리와 성능을 제공하면서도 데이터 레이크의 저렴한 비용과 개체 스토어 유연성을 모두 갖추었습니다.

이 통합 플랫폼은 분석, 데이터 사이언스, 머신 러닝을 통합하지 못하게 방해하던 데이터 사일로를 제거하여 데이터 아키텍처를 단순화합니다. 그리고 오픈 소스 및 개방적 표준을 기반으로 유연성을 극대화합니다. 또한, 네이티브 협업 기능을 통해 팀 간 협업 능력을 강화하고 혁신의 속도를 향상합니다.

챕터 1: 개요

하둡에서 Databricks로 구성 요소 매핑

하둡 마이그레이션을 계획할 때는 레거시 하둡 기술을 최신 클라우드 기능으로 올바르게 매핑하는 것이 중요합니다. 아래 테이블은 하둡 플랫폼의 핵심적 기능을 Databricks 플랫폼으로 매핑합니다.

hadoop	databricks
데이터 스토리지 • 블록 스토리지 기반 HDFS • Kafka • HBase	**대응 제품** • 클라우드 오브젝트 스토리지: S3, ADLS, Azure Blob • 클라우드 네이티브 메시지 버스: Kinesis, Azure Event Hubs, Azure IoT Hub • 클라우드 네이티브 NoSQL: DynamoDB, CosmosDB
데이터 처리 • MapReduce • Pig • HiveQL • Spark	**대응 제품** • Databricks Delta Engine: 성능을 10~100배 개선한 최적화 Apache Spark • Databricks SQL: ANSI SQL 2003 준수 • 코드 없는 ETL: Azure Data Factory 매핑 플로우, Prophecy, Talend 등과 통합
작업 • Oozie(워크플로우 자동화)	**대응 제품** • Databricks 작업 스케줄러 • Apache Airflow 및 Azure Data Factory와 기본 통합 • Databricks API를 통해 외부 스케줄러 사용
코드 개발 • Apache Zeppelin 노트북 • Jupyter 노트북	**대응 제품** • Databricks 노트북 • Databricks API를 통해 Zeppelin, Jupyter, 모든 노트북 또는 IDE(예: Pycharm, IntelliJ 등) 지원
인터랙티브/즉석 쿼리 • HUE • Impala/Hive LLAP	**대응 제품** • Databricks SQL 워크스페이스 • Delta Engine/Photon

하둡은 데이터를 처리하기 위한 여러 가지 분산형 프로그래밍 프레임워크를 제공합니다. 여기에는 기존 저 수준 Apache MapReduce API 및 고 수준 프레임워크(Apache Pig, Apache Hive)를 포함합니다. 하둡은 Apache Spark도 지원합니다. Databricks Delta Engine은 데이터 처리가 간편합니다. Spark와 Databricks가 결합되어 최적화를 통해 오픈 소스 Spark에서 성능을 10~100배 향상하기 때문입니다. 그리고 Spark는 API를 사용하여 Java, Scala, Python, SQL, R로 코딩할 수 있습니다. Spark SQL은 ANSI SQL 2003을 준수합니다. Databricks 파트너 통합은 Azure Data Factory, Prophecy, Talend를 통해 코드 없이 데이터 파이프라인을 개발하도록 지원합니다.

하둡의 기본 워크플로우와 작업 오케스트레이션 도구는 Oozie입니다. Databricks는 더 높은 수준의 고급 예약 도구(예: Apache Airflow, Microsoft Azure Data Factory)와도 통합되며 작업 스케줄러도 제공합니다. Databricks에서 Databricks REST API를 통해 원하는 스케줄러를 사용할 수 있습니다.

하둡은 Apache Zeppelin 노트북을 클러스터에 연결하도록 지원하여 데이터와 시각적으로 상호작용하게 해줍니다. Databricks는 클라우드 내에 네이티브 노트북 인터페이스가 있습니다.

Databricks는 Zeppelin과 Jupyter 노트북도 지원하며, Databricks REST API를 통해 자주 사용하는 노트북이나 IDE를 연결할 수도 있습니다.

Databricks SQL 워크스페이스는 인터랙티브 SQL 및 즉석 쿼리에 사용할 수 있습니다. Databricks SQL은 레이크하우스에서 BI 및 SQL 쿼리를 실행하기 위한 네이티브 SQL 인터페이스로, 빠른 성능과 높은 동시성을 갖추었습니다. 스키마 브라우저가 있는 사용자 인터페이스, 자동 완성이 가능한 쿼리 편집기, 풍부한 시각화를 생성할 수 있는 대시보드로 구성됩니다. 사용자가 알림을 포함한 쿼리 예약을 설정할 수 있습니다. Databricks SQL은 여러 클러스터에서 쿼리의 부하를 자동으로 투명하게 분산시켜, 쿼리 응답에 높은 동시성과 낮은 지연을 제공합니다. Tableau, Microsoft Power BI 등 인기 있는 BI 도구는 네이티브 JDBC/ODBC 커넥터를 사용하여 플랫폼에 연결할 수 있습니다.

AWS Databricks SQL 가이드

Azure Databricks SQL 가이드

챕터 02

플랫폼 관리

Databricks 배포

네트워킹 및 사용자 정의 구성

클러스터

클러스터 풀

클러스터 리소스 관리

클러스터 모니터링

REST API 및 명령줄 인터페이스

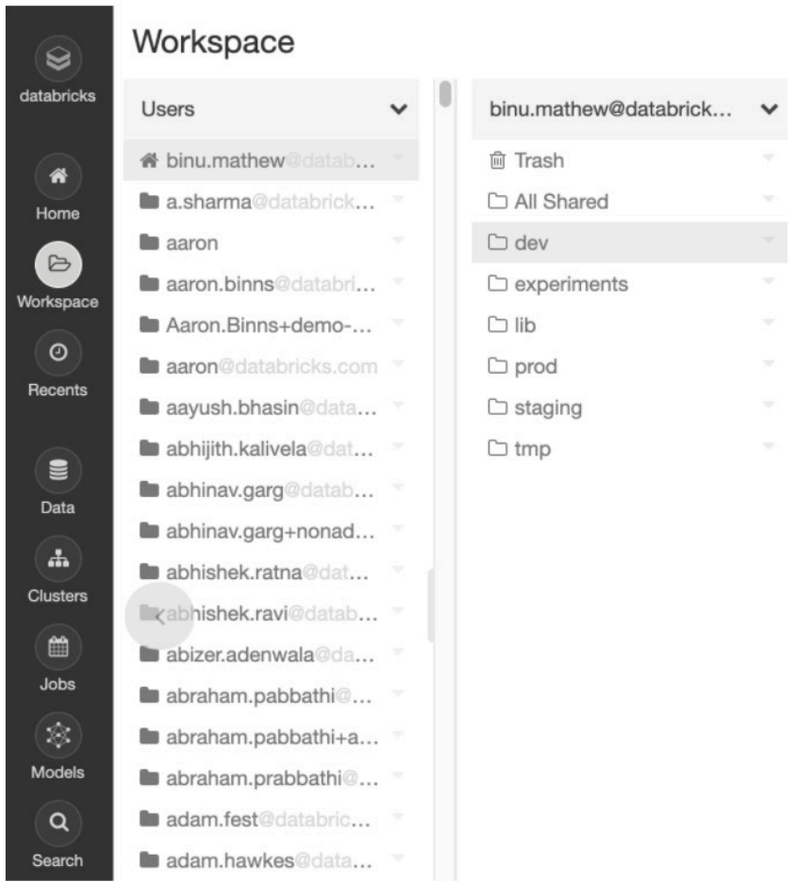
보안 및 거버넌스

데이터 탐색 및 감사

챕터 2: 플랫폼 관리

Databricks 배포

Databricks 배포는 워크스페이스라고도 하는데, 여기에는 웹 UI 포털이 있어 플랫폼을 관리, 운영할 수 있으며 애플리케이션 개발, 테스트와 배포에도 이를 이용할 수 있습니다. 포털 액세스는 조직 내 ID 제공자를 통해 다단계 인증(MFA)을 사용하여 SSO로 인증할 수 있습니다. Azure Databricks에서 포털 액세스는 Azure Active Directory(AAD) 계정으로 MFA를 사용하여 SSO로 인증합니다. AAD 계정이 있는 사용자만 포털에 액세스할 수 있습니다. 웹 UI 외에도 여러 가지 REST API와 명령줄 인터페이스(CLI)도 플랫폼 관리와 애플리케이션 개발, 테스트, 배포에 제공됩니다.



특정 조건에 따라 워크스페이스(배포)가 하나 이상일 수 있습니다.

- **환경의 분리:** 예를 들어 개발, 스테이징, 프로덕션 및 기타 환경에 각기 다른 워크스페이스를 제공합니다.
- **사업부의 분리:** 마케팅, 재무, 위험 관리 등의 부서에 다른 워크스페이스를 제공합니다.

Databricks에서 쓰는 몇 가지 기본 개념을 알아두는 것도 중요합니다. UI 왼쪽에 이런 개념들이 아이콘 형태로 표시됩니다 (다음 이미지 참조). 이 서비스는 웹 앱 UI와 REST API 및 CLI에도 모두 제공됩니다. 우선 이런 개념들을 간략히 소개한 다음, 이 가이드의 뒷부분에서 좀 더 자세히 다루도록 하겠습니다.

워크스페이스

워크스페이스는 Databricks에서 수행하는 모든 업무를 체계적으로 구성하는 데 유용합니다. 컴퓨터의 폴더 구조와 마찬가지로, 워크스페이스를 사용하면 노트북과 라이브러리를 저장하고 다른 사용자와 공유할 수도 있습니다. 조직 내 각각의 사용자에게 폴더가 하나씩 주어져 자신의 업무를 디렉터리 구조로 구성할 수 있게 됩니다. 워크스페이스에는 권한 설정이 있어 내 작업에 액세스할 권한이 있는 인물을 주도적으로 관리할 수 있습니다.

- Azure의 경우, 온라인 문서를 참조하여 워크스페이스에 대해 자세히 알아보세요.
- WS의 경우, 여기를 클릭하세요.

클러스터

클러스터는 데이터 워크로드를 처리하는 가상 머신(VM) 그룹을 말합니다. 클러스터를 통해 Java/Scala JAR 파일, Python 스크립트와 wheel/egg 파일에서 작성된 노트북, 라이브러리, 사용자 정의 코드에서 코드를 실행할 수 있습니다. 클러스터에는 세 가지 유형이 있습니다.

1. **표준:** 단일 사용자 워크로드 및/또는 단일 작업을 실행하는 데 사용하며 수명이 짧은 클러스터입니다.
2. **높은 동시성:** 여러 사용자가 공유하며, 원래 장기 실행되는 클러스터용입니다.
3. **단일 노드:** 경량 분석용(Spark를 사용하거나 사용하지 않음) 단일 VM 인스턴스입니다.

클러스터는 데이터를 저장하지 않습니다. 데이터는 항상 클라우드 스토리지 계정과 다른 데이터 소스에 저장됩니다.

자세한 내용은 이 가이드의 “클러스터” 섹션을 참조하세요.

노트북

노트북은 협업 환경을 제공하는 IDE로써 Scala, Python, R, SQL, Markdown으로 명령을 실행할 수 있습니다. 노트북에는 기본 언어가 있지만 각 셀을 재정의하여 다른 언어를 사용할 수도 있습니다. 노트북은 클러스터와 연결해야 명령을 실행할 수 있지만, 한 클러스터와 영구적으로 연결된 것은 아닙니다. 따라서 노트북은 웹에서 공유하거나 로컬 컴퓨터에 다운로드할 수 있습니다. 노트북에서 대시보드를 만들어, 셀을 생성한 코드 없이도 출력 셀을 표시할 수단으로 활용할 수 있습니다. 또한, 노트북은 클릭 한 번으로 작업을 예약하여 데이터 파이프라인을 실행하거나, 머신 러닝 모델을 업데이트하거나, 대시보드를 업데이트할 수 있습니다.

자세한 내용은 이 가이드의 “코드 개발을 위한 노트북 및 IDE” 섹션을 참조하세요.

라이브러리

라이브러리는 비즈니스 문제를 해결하는 데 필요한 추가적 기능을 제공하는 패키지나 모듈입니다. 직접 작성된 Scala/Java JARs, Python egg/wheel 파일이거나 직접 작성된 패키지일 수 있습니다. UI를 통해 직접 라이브러리를 작성하거나, Libraries API를 사용하거나, 패키지 관리 유틸리티(예: PyPi, Maven, CRAN)를 통해 직접 설치하여 라이브러리를 작성하고 업로드할 수 있습니다.

온라인 문서에서 라이브러리에 대해 자세히 알아보세요.

AWS AZURE

데이터

클라우드 스토리지에서 사용자와 상호작용을 주고받는 데이터는 테이블로 구성된 데이터베이스로 표현되는 정형 데이터로 구성할 수 있습니다. 이런 테이블에는 열 이름과 데이터 유형이 있는 스키마가 포함되어 있습니다. UI의 데이터 아이콘에 사용자의 조직 내 정형 데이터 자산이 목록으로 나열됩니다. Databricks는 정형, 반정형, 비정형 데이터 소스로 작업할 수 있습니다.

자세한 내용은 이 가이드의 “데이터 소스” 섹션을 참조하세요.

작업

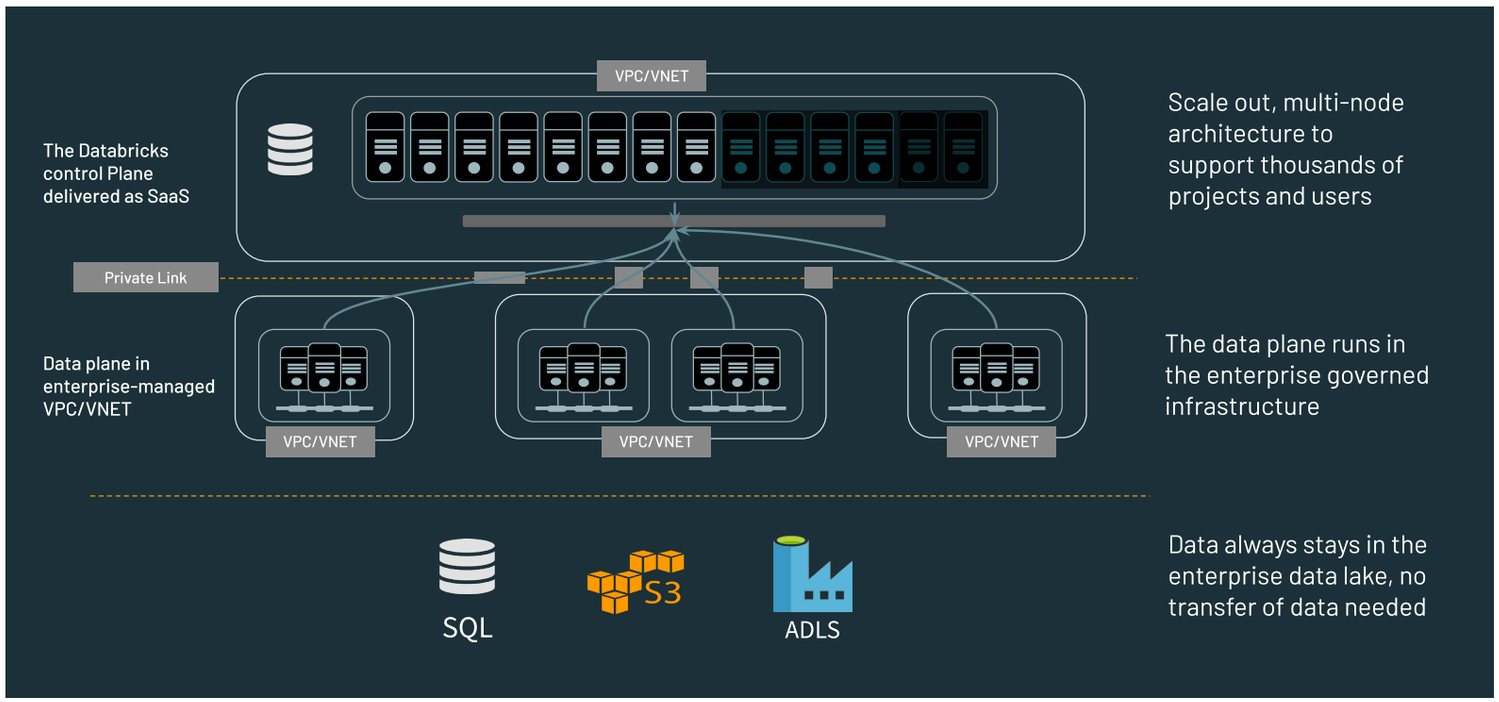
작업은 코드 실행을 예약하는 방식을 말합니다. 코드가 실행될 장소는 이미 기존에 존재하는 클러스터일 수도 있고, 자체적인 클러스터일 수도 있습니다. 작업은 노트북의 코드나 JAR 파일, Python 스크립트에서 실행할 수 있습니다. 작업은 UI를 통해 수동으로 생성하거나, REST API 또는 명령줄 인터페이스(CLI)를 통해 생성할 수 있습니다.

자세한 내용은 이 가이드의 “작업 예약 및 제출” 섹션을 참조하세요.

챕터 2: 플랫폼 관리

네트워킹 및 사용자 정의 구성

간략히 말하면, Databricks 배포 아키텍처는 Databricks 구독에서 실행되는 제어 영역(Control Plane)과 고객 구독에서 실행되는 데이터 영역(Data Plane)으로 구성됩니다. 제어 영역에는 Databricks가 자체 계정에서 관리하는 백엔드 서비스가 포함됩니다. 데이터 영역은 고객의 계정에서 관리하며, 이곳이 데이터 위치이며 데이터를 처리하는 곳이기도 합니다.



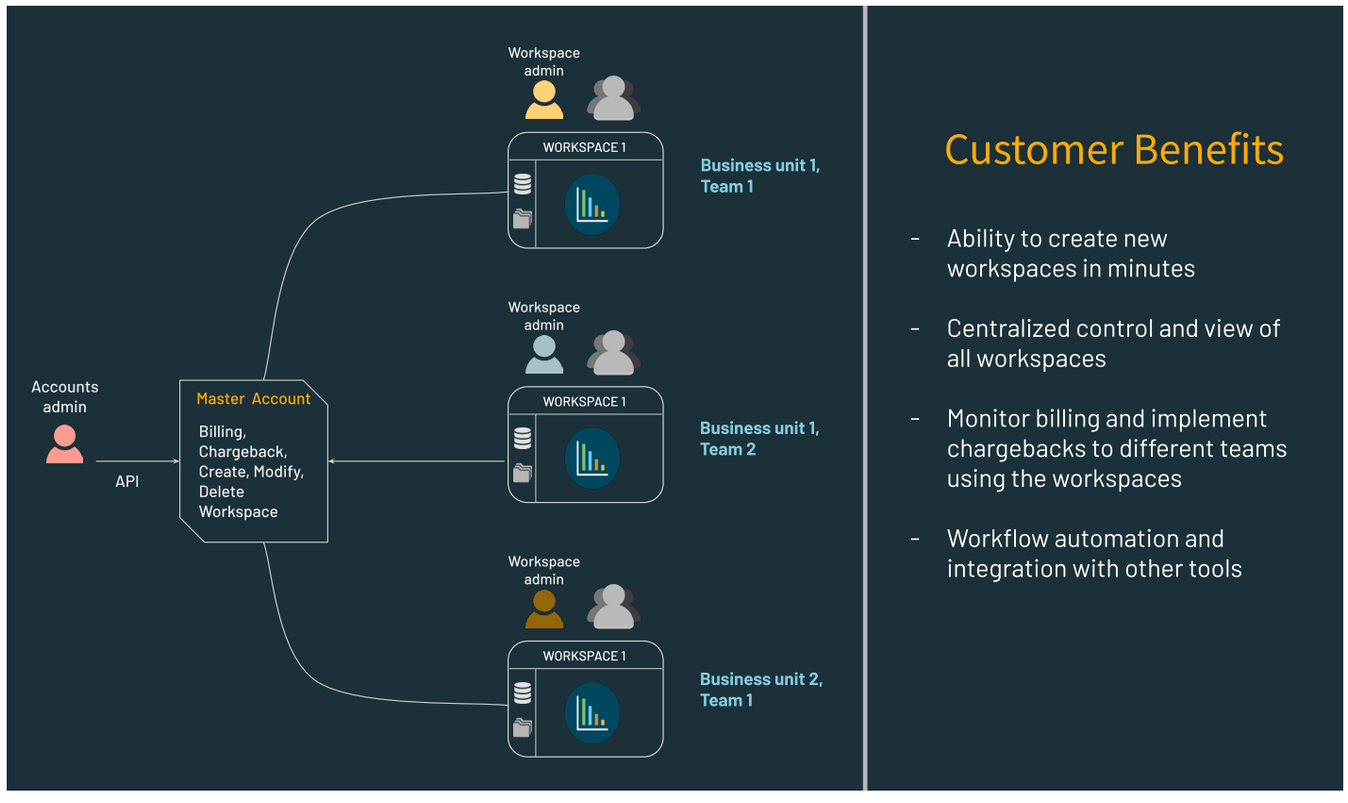
이 아키텍처에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

이 아키텍처의 핵심 기능 예시:

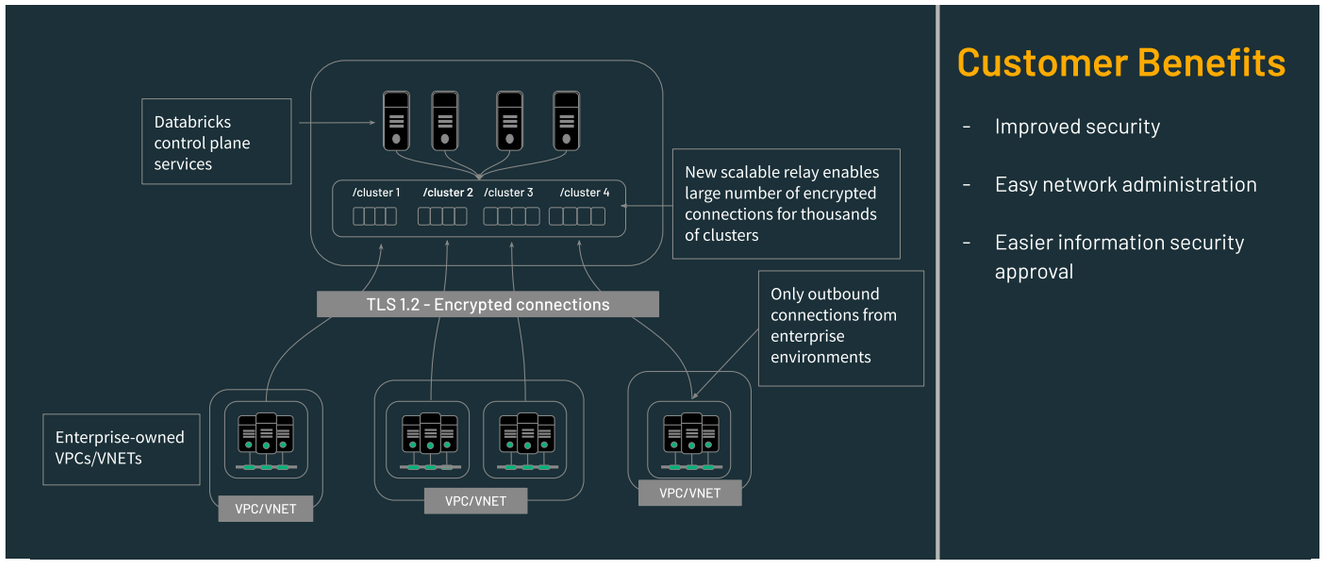
- 여러 개의 워크스페이스
- 자체 VPC/VNET 가져오기
- Bring your own key
- 공개 IP 없음
- IP 액세스 목록

여러 개의 워크스페이스



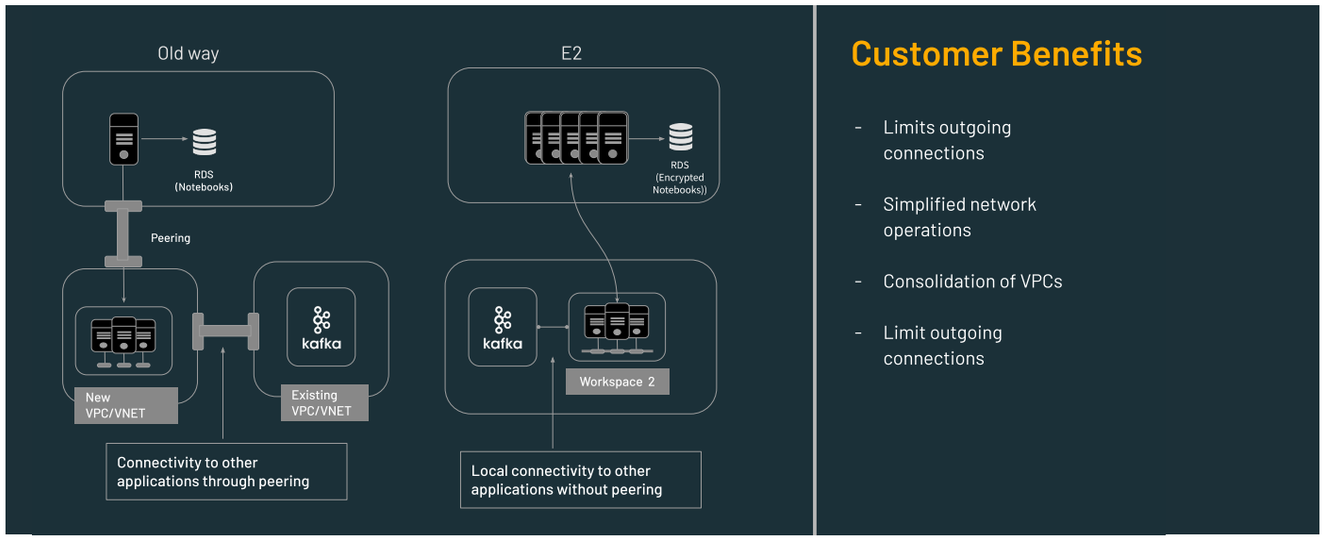
- AWS 문서
- Azure의 경우, Azure 계정 포털에서 여러 워크스페이스를 관리할 수 있습니다.

공개 IP 없음



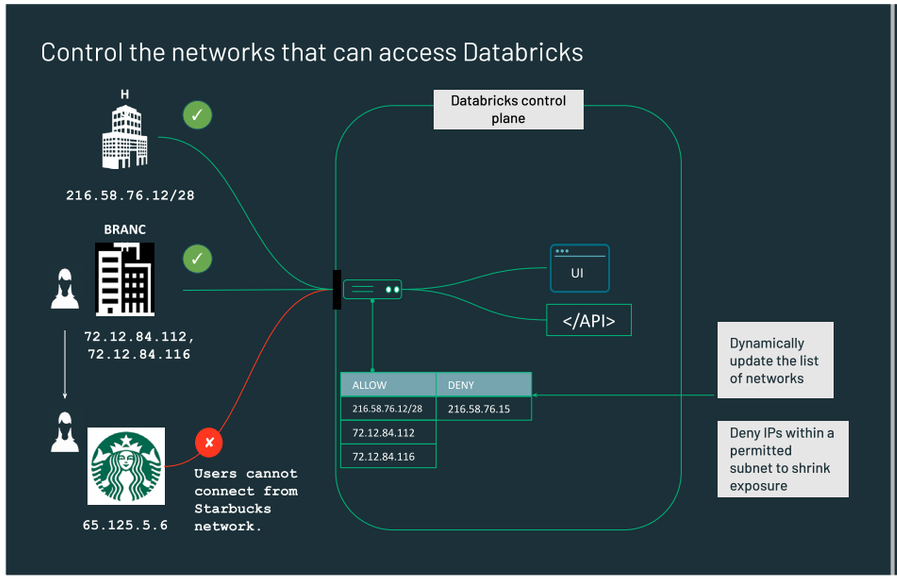
AWS 문서 Azure 문서

자체 VPC/VNET 가져오기



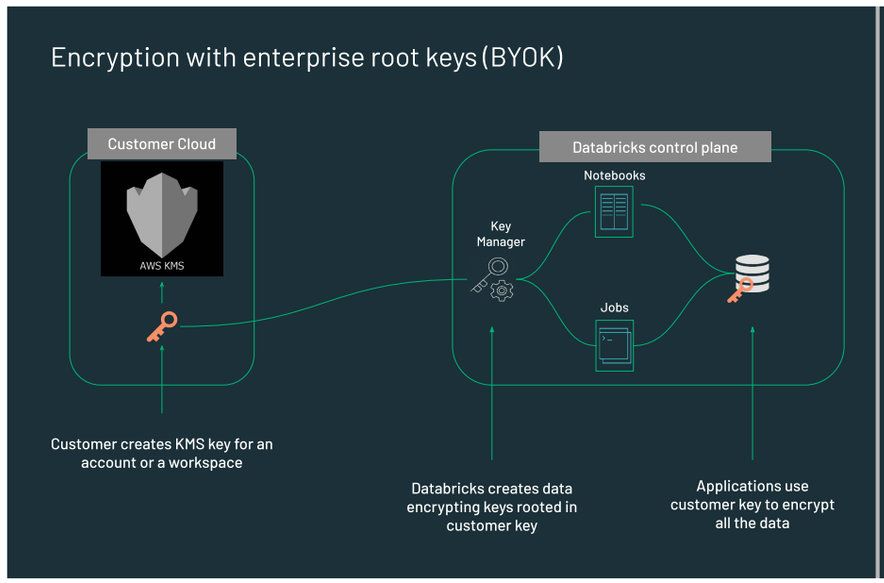
AWS 문서 Azure 문서

IP 액세스 목록



AWS 문서 Azure 문서

Bring your own key



AWS 문서 Azure 문서

챕터 2: 플랫폼 관리

클러스터

Databricks는 완전 관리형 PaaS 서비스로 인프라 관리, 운영, 유지관리가 필요하지 않습니다. 사용자와 프로세스는 VM 클러스터에서 코드를 실행하여 데이터 엔지니어링, 데이터 사이언스, 데이터 분석 워크로드에 사용합니다. 여기에는 배치, 실시간 프로덕션 ETL 파이프라인, 스트리밍 분석, 즉석 분석, 머신러닝, 딥러닝, 그래프 분석 등이 포함됩니다. 클러스터는 고정 크기(Fixed-Size) 또는 자동 크기 조정 등이 있으며, 기본적으로 120분 동안 아무런 활동이 없으면 자동 종료됩니다(이것은 구성 가능함).

- 크기 조정 클러스터의 경우, 클러스터 수명 주기가 끝날 때까지 일정한 상태를 유지합니다. 이것은 필요한 컴퓨팅 용량(CPU 코어와 RAM)을 정확히 알고 있을 때 탁월한 선택입니다. VM을 추가로 할당하여 시작하는 데 시간을 낭비할 일이 없기 때문입니다.
- 자동 크기 조정 클러스터의 경우, 최소 클러스터 VM 노드에서 최댓값(사용자가 구성)까지 동적으로 확장됩니다. 이 옵션은 필요한 컴퓨팅 용량(CPU 코어, RAM)을 쉽게 예측하기 어려울 때 권장합니다. 예를 들어, 데이터 볼륨이 증가하거나 데이터 편향(data skew)이 있을 경우에 해당합니다.

클러스터는 세 가지 유형으로 나뉩니다.

1. 표준 클러스터
2. 동시성이 높은 클러스터
3. 단일 노드 클러스터

표준 클러스터

표준 클러스터는 단일 사용자에게 추천합니다. 고정 크기일 수도, 자동 크기 조정 클러스터일 수도 있습니다. 표준 클러스터는 일반적으로 작업 실행에 사용하는 단기 클러스터지만, 스트리밍 작업의 경우 클러스터를 상시 운영하면서 장기 실행할 수도 있습니다. 표준 클러스터는 개발 언어에 관계없이 워크로드를 실행할 수 있습니다. Java, Python, R, Scala, SQL 등을 가리지 않습니다. 표준 클러스터는 개별적인 작업을 실행하는 데도 쓰입니다. 예를 들어 스트리밍, ETL이나 머신러닝이 대표적입니다. 표준 클러스터는 여러 사용자 및/또는 프로세스의 작업이 아니라 단일 사용자의 작업을 실행하므로, 강력한 리소스 분리, SLA 보증과 보안이 제공됩니다.

동시성이 높은 클러스터

동시성이 높은 클러스터는 여러 사용자가 하나의 클러스터에 액세스하여 인터랙티브/자동 작업을 실행할 때 이상적입니다. 클러스터 용량을 고정하거나 자동 확장할 수 있습니다. 기본적으로 동시성이 높은 클러스터는 자동으로 크기를 조정하도록 설정되어 있습니다. 이런 클러스터는 SQL, Python, R만 지원합니다. 동시성이 높은 클러스터의 주된 장점은 이 클러스터는 Apache Spark 네이티브 세분화 공유 기능을 제공하므로 리소스 활용량은 극대화하고, 쿼리 지연은 최소화하여 같은 클러스터의 모든 사용자가 작업을 실행할 수 있다는 점입니다. 해당 클러스터의 총 컴퓨팅 리소스(CPU와 RAM)를 모든 사용자가 공유하는 형태입니다. 동시성이 높은 클러스터를 이용하면 공유된 사용자 작업 환경, 실험, 일부 프로덕션 워크로드의 테스트와 실행에 대한 비용을 절감할 수 있습니다.

단일 노드 클러스터

단일 노드 클러스터는 Spark 드라이버 하나로 구성되어 있으며 Spark 작업자는 없습니다. 이 클러스터는 Spark 작업과 모든 Spark 데이터 소스를 지원합니다. 반면, 표준 클러스터는 Spark 작업을 실행하려면 하나 이상의 Spark 작업자가 필요합니다. 단일 노드 클러스터는 다음과 같은 용도에 유용합니다.

- 데이터를 로드하고 저장하려면 Spark를 이용해야 하는 단일 노드 머신 러닝 워크로드 실행
- 가벼운 탐색적 데이터 분석(EDA)

모든 클러스터 유형은 UI, REST API 또는 명령줄 인터페이스(CLI)를 통해 생성 및 구성할 수 있습니다. Databricks는 다양한 워크로드에 맞춰 다양한 VM 유형을 지원합니다. 메모리 최적화, CPU 최적화, 스토리지 최적화 및 GPU 가속 VM 등이 대표적입니다. 또한, Databricks는 사전 정의된 설정으로 클러스터를 시작하는 사용자 정의 컨테이너도 지원합니다. 예를 들어, 클러스터에 사용해야 하는 라이브러리가 여러 개인 경우 모든 종속성을 포함하는 사용자 정의 Docker 이미지를 만들고 이 이미지를 사용하여 Databricks 클러스터를 시작함으로써 클러스터 시작 시간을 단축할 수 있습니다.

클러스터에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

클러스터 모범 사례:

- 필요한 컴퓨팅 용량을 알 수 없을 경우 자동 크기 조정 클러스터 사용
- 자동 종료 설정(해당하는 경우)
- 최신 Databricks Runtime 버전을 사용하여 최신 성능 및 여타 여러 가지 최적화 활용
- 노트북 또는 BI 도구를 통해 사용자 팀이 데이터를 분석하거나, 또는 테이블 ACL을 통해 데이터 보호를 적용하고 싶을 경우 동시성이 높은 클러스터 모드 사용
- 프로젝트 또는 팀 기반 차지백에 클러스터 태그 사용
- 사용자 정의 Spark 구성 설정을 한 클러스터의 모든 노드에 적용 가능(필요한 경우)
- Cluster Event Log 및 Spark UI를 사용하여 클러스터 활동과 제출된 작업 성능 분석
- Cluster Log Delivery를 구성하여 Spark 드라이버와 작업자 로그를 클라우드 스토리지에 제공
- Cluster Access Control을 사용하여 사용자와 그룹 권한 구성
- Initialization Scripts 또는 Databricks Container Services를 사용하여 사전 설치된 소프트웨어와 라이브러리로 클러스터 시작 Databricks Container Services를 사용하면 직접 Docker 이미지를 생성하고, 이를 사용하여 Databricks 클러스터를 시작할 수 있습니다.
- User Cluster Policies를 사용하여 사용자가 시작할 수 있는 클러스터 유형 제한

챕터 2: 플랫폼 관리

클러스터 풀

클러스터 시작 시간과 자동 크기 조정 시간은 클라우드 제공업체로부터 VM 인스턴스를 가져오는 데 걸리는 시간에 따라 결정됩니다. 이 때문에 SLA에 영향이 발생한다면, Databricks 클러스터 풀을 사용하는 것이 좋습니다. 풀은 일련의 유휴, 바로 사용 가능한 인스턴스를 유지 관리하여 클러스터 시작과 자동 크기 조정 시간을 단축하는 효과가 있습니다. 풀과 연결된 클러스터에 VM이 더 필요한 경우, 이 VM은 우선 풀에 속한 유휴 VM 중 하나를 할당하려 합니다. 이렇게 하면 VM이 지연 없이 즉시 해당 클러스터에 연결되므로, SLA를 더욱 확실히 보증할 수 있습니다. 풀에 할당된 VM에 따라 관련 클라우드 비용을 부과합니다. 다만 Databricks는 그와 같은 인스턴스에 대한 요금을 부과하지는 않습니다.

클러스터에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

챕터 2: 플랫폼 관리

클러스터 리소스 관리

하둡 사용자들은 애플리케이션 리소스 관리, 작업 예약에 Apache YARN을 친숙하게 사용합니다. YARN을 사용하면 여러 사용자와 프로세스가 하나의 하둡 클러스터를 공유할 수 있으므로 CPU, RAM과 같은 클러스터 리소스도 공유하게 됩니다.

- 모든 작업에 리소스를 고르게 분배하는 공정한 스케줄링
- 용량 스케줄링으로 가중치가 부여된 비율에 따라 대기열을 정의하고, 사용자와 그룹을 해당 대기열에 할당하여 대기열 내에서 작업을 실행

YARN은 작업을 실행한 다음 지속되는 시간 동안 모니터링하므로 혹시 장애가 발생하는 경우, YARN이 그러한 작업을 다시 시작하려 시도하게 됩니다.

Databricks는 표준 클러스터와 동시성이 높은 클러스터의 애플리케이션 리소스 관리와 작업 스케줄링에 각기 다른 방식을 적용하며 자체적인 리소스 관리자를 사용하는데, 이는 Apache Spark의 독립형 리소스 관리자에 좀 더 가깝습니다. 하나의 SparkContext를 클러스터의 여러 세션에서 공유합니다. 클러스터상의 각 사용자마다 자체적으로 별도의 SparkSession이 있지만, SparkContex는 모든 사용자가 같은 것을 공유합니다. SparkContext를 사용하면 Spark 애플리케이션이 리소스 관리자의 도움을 받아 클러스터에 액세스할 수 있습니다.

다음의 Spark 구성 설정은 표준 클러스터와 동시성이 높은 클러스터에 모두 적용됩니다.

- Databricks 클러스터에는 기본적으로 모든 작업에 CPU와 RAM 컴퓨팅 리소스를 고르게 분배하는 공정한 스케줄링을 사용하는 기능이 있습니다.
  `spark.scheduler.mode FAIR`
- Databricks는 기본적으로 YARN과 유사한 용량 대기열이 없습니다
- Databricks는 실제로 선점 방식을 사용해 리소스 초과 할당을 방지합니다. 따라서 모든 작업이 리소스를 똑같이 나눠 가지도록 보장합니다.
  `spark.databricks.preemption.enabled true`
- 각 사용자가 자체 SparkSession 소유
  `spark.databricks.session.share false`

표준 클러스터 vs. 동시성 높은 클러스터의 리소스 공유

표준 클러스터는 원래 용도가 단일 처리 작업용입니다. 이는 사용자가 노트북을 통해 작업을 제출한 인터랙티브 세션일 수도 있고, 아니면 자동 작업일 수도 있습니다. 표준 클러스터를 공유하여 여러 사용자 및/또는 프로세스에서 작업을 실행하는 것은 권장하지 않습니다. 동시성이 높은 클러스터의 경우, 원래 공유에 적합합니다. 표준 클러스터와 동시성이 높은 클러스터는 둘 다 기본적으로 선취권을 사용합니다. 다만 동시성이 높은 클러스터의 경우, 각 사용자의 작업이 별도의 공정한 스케줄러 풀에서 실행되며 여기에 리소스를 공정하게 할당하도록 구성된 선점을 적용합니다. 동시성이 높은 클러스터는 사용자마다 장애 분리를 생성하기도 합니다. 장애를 분리하면 각 사용자의 환경이 서로 격리되므로 한 사용자의 프로세스가 전체 클러스터에 영향을 미칠 수 없습니다.

한 클러스터를 여러 사용자가 공유하는 방식의 보편적인 문제점은 한 사용자의 잘못된 코드 때문에 Spark 드라이버가 다운되어 모든 사용자의 클러스터가 중단될 수 있다는 것입니다. 이런 상황에서 Databricks 리소스 관리자는 샌드박싱을 통한 장애 분리를 제공하여 각 사용자에게 속한 드라이버 프로세스를 서로에게서 격리합니다. 따라서 한 사용자가 드라이버를 다운시킬 수도 있는 명령을 실행해도 안전하므로, 다른 사용자의 환경에 타격을 입힐 우려가 없습니다.

동시성이 높은 클러스터 선점의 경우 다음과 같이 Spark 설정을 구성하여 원하는 방식으로 여러 작업에 클러스터 하나를 공유할 수 있습니다.

`spark.databricks.preemption.threshold 0.5`
이것이 사용자 일인당 보장되는 공정한 공유 부분입니다. 이것을 1.0으로 설정하면 스케줄러가 능동적으로 작동하여 완벽하게 공정한 공유를 보장합니다. 이를 0.0으로 설정하면 사실상 선점이 비활성화됩니다. 기본 설정은 0.5입니다. 즉 최악의 경우에도 사용자는 공정한 점유율의 절반을 받을 수 있다는 뜻입니다.

`spark.databricks.preemption.timeout 30s`
선점이 적용될 때까지 사용자가 리소스 없이 버텨야 하는 시간입니다. 이 값을 낮게 설정하면 좀 더 인터랙티브한 반응 시간을 얻을 수 있지만, 대신 클러스터 효율이 떨어집니다. 권장값은 1~100초입니다.

`spark.databricks.preemption.interval 5s`
스케줄러가 태스크 선점을 검사하는 빈도입니다. 이 값은 선점 시간 초과(time-out)보다 작아야 합니다.

선점에 대한 자세한 내용은 온라인 문서를 참조하세요.
AWS Azure

표준 클러스터와 동시성이 높은 클러스터 둘 다, 기본적으로 모든 쿼리가 같은 공정 스케줄링 풀에서 실행되는 노트북에서 시작됩니다. 그러므로 노트북에 있는 모든 스트리밍 쿼리의 트리거에서 생성된 작업은 선착순(FIFO)으로 하나씩 실행됩니다. 이 때문에 쿼리에 불필요한 지연이 발생할 가능성이 있습니다. 클러스터 리소스를 효율적으로 공유하고 있지 않기 때문입니다. 모든 스트리밍 쿼리가 작업을 동시에 실행하고 클러스터를 효율적으로 공유하도록 하려면, 쿼리가 별도의 스케줄러 풀에서 실행되도록 설정하면 됩니다.

자세한 정보는 온라인 문서를 참조하세요.
AWS Azure

동시성이 높은 클러스터를 사용하면 여러 사용자가 서로 다른 다양한 SQL 쿼리를 제출할 수 있습니다. Databricks는 크기가 큰 쿼리의 가장 보편적인 원인을 살펴보고, 임계값을 넘는 쿼리를 종료하여 단독(rogue) 쿼리가 클러스터 로그를 독점하지 않도록 방지합니다. 이 기능을 Query Watchdog이라고 합니다.

`spark.databricks.queryWatchdog.enabled true`
이렇게 하면 Query Watchdog을 활성화합니다.

`spark.databricks.queryWatchdog.outputRatioThreshold 1000L`
쿼리가 입력 행 개수보다 너무 많은 출력 행을 생성하지 못하도록 예방하려면 Query Watchdog을 활성화하고 최대 출력 행 개수를 입력 행 개수의 배수로 설정해야 합니다. “1000L”은 특정 태스크가 입력 행 개수의 1,000배를 초과할 수 없다는 선언입니다.

Query Watchdog에 대한 자세한 내용은 온라인 문서를 참조하세요.
AWS Azure

챕터 2: 플랫폼 관리

클러스터 모니터링

Databricks는 Ganglia 지표를 제공하여 클러스터에서 작업이 실행될 때 그러한 클러스터를 모니터링합니다. 이 기능은 Databricks 플랫폼의 기본 제공하며 추가로 설정이나 통합하지 않아도 됩니다.

자세한 정보는 온라인 문서를 참조하세요.

AWS AZURE

AWS 고급 모니터링

AWS CloudWatch를 사용하여 한층 수준 높은 고급 애플리케이션 및 인프라 모니터링을 구현할 수도 있습니다. 이렇게 하려면 Databricks EC2 노드에 AWS CloudWatch 에이전트를 설치하여 지표가 CloludWatch로 전송되도록 하면 됩니다. Databricks는 타사 소프트웨어 설치를 허용하며, 타사 소프트웨어의 유지관리와 지원은 고객 책임입니다. Databricks 클러스터의 모든 노드에 CloudWatch 에이전트를 설치하려면 Init 스크립트나 Databricks Container Services를 사용하면 됩니다.

- init 스크립트에 대한 자세한 내용은 온라인 문서를 참조하세요.
- Databricks Container Services에 대한 자세한 내용은 온라인 문서를 참조하세요.

CloudWatch 에이전트는 Databricks 노드에 쉽게 설치할 수 있습니다. Databricks EC2 인스턴스는 모든 아웃바운드 트래픽을 허용하며, 따라서 CloudWatch 에이전트가 CloudWatch에 지표를 보내도록 구성할 수 있습니다. CloudWatch 에이전트를 설치하고 나면 이를 구성해야 합니다.

Init 스크립트를 사용한 CloudWatch 에이전트 설치

init 스크립트의 예시는 다음과 같습니다. 실제로 작동하는 예시입니다. CloudWatch 에이전트를 실행된 모든 Databricks 클러스터 노드에 다 설치해야 하는 경우, 이 init 스크립트는 전역 init 스크립트가 됩니다.

```
dbutils.fs.put("/databricks/init/cloud-watch-agent-install.sh","""
#!/bin/bash
# install CloudWatch agent
wget https://s3.amazonaws.com/amazoncloudwatch-agent/linux/amd64/latest/AmazonCloudWatchAgent.zip
unzip AmazonCloudWatchAgent.zip
sudo ./install.sh
# copy configuration files from root S3 bucket to local file system on Ubuntu nodes
cp /tmp/bmathew/test_script/common-config.toml /opt/aws/amazon-cloudwatch-agent/etc/common-config.toml
cp /tmp/bmathew/test_script/amazon-cloudwatch-agent-schema.json /opt/aws/amazon-cloudwatch-agent/doc/
amazon-cloudwatch-agent-schema.json
# start the CloudWatch Agent
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl -a fetch-config -m ec2 -c file:/
opt/aws/amazon-cloudwatch-agent/doc/amazon-cloudwatch-agent-schema.json -s
""", True)
```

CloudWatch 에이전트 구성

1. **선택 사항:** common-config.toml을 구성하여 CloudWatch 에이전트의 공통 구성과 이름이 지정된 프로필을 수정합니다. 이 파일은 프록시 설정을 지정해야 하는 경우, 또는 에이전트가 인스턴스 위치가 아닌 다른 리전의 CloudWatch로 지표를 보내야 하는 경우에만 수정합니다.

   모든 Databricks EC2 인스턴스는 아웃바운드 액세스 권한이 있습니다. 모든 Databricks 클러스터는 같은 리전으로 시작됩니다(단, 사용자가 리전 내 다른 가용 영역을 지정할 수는 있음). CloudWatch 를 Databricks 샤드(배포)와 같은 지역에 설치한다고 가정하면 이 파일을 수정하지 않아도 될 가능성이 있습니다. Databricks 클러스터에 연결된 IAM 역할에 적절한 권한이 있어야 CloudWatch에 지표를 전송할 수 있습니다.

   만일 이 파일을 수정해야 한다면, 이 작업은 init 스크립트를 이용해 수행하면 됩니다.

   init 스크립트가 루트 S3 버킷(dbfs — databricks file system)에서 이 구성 파일의 수정/최종 버전을 복사해 Databricks EC2 노드(로컬 Linux 파일 시스템)의 적절한 디렉터리에 넣습니다.

2. **필수 사항:** JSON 구성 파일이 필요합니다. 구성 파일부터 만든 다음에 어느 서버에서든 에이전트를 시작해야 합니다. 에이전트 구성 파일은 JSON 파일의 일종입니다. 이 파일이 에이전트가 수집할 지표와 로그를 지정합니다. init 스크립트가 루트 S3 버킷(dbfs - databricks file system)에서 에이전트 구성 파일의 수정/최종 버전을 복사하여 Databricks EC2 노드(로컬 Linux 파일 시스템)의 적절한 디렉터리에 넣습니다.

CloudWatch 에이전트 구성에 대한 자세한 내용은 공식 AWS 문서를 참조하세요.

Azure 고급 모니터링

Azure의 경우, Azure Monitor와도 통합됩니다. Azure 고객들에게는 이 솔루션이 더 친숙할 텐데, Azure 계정의 모든 활동은 이미 Azure Monitor를 사용해 관측하고 있기 때문입니다. Azure Databricks와 Azure Monitor를 통합하면 작업/애플리케이션 로그를 Azure Monitor로 보내 클러스터 지표를 상세하게 모니터링할 수 있습니다.

이 통합을 완료하는 방법은 온라인 문서를 참조하세요.

Azure Log Analytics 워크스페이스와 통합하여 지표 수집 데이터를 간편하게 분석할 수도 있습니다. 온라인 문서를 참조하세요.

챕터 2: 플랫폼 관리

REST API 및 명령줄 인터페이스

Databricks는 다양한 REST API와 강력한 명령줄(CLI)을 제공하여 플랫폼 상호작용이 다음과 같은 태스크를 실행할 수 있게 지원합니다.

- 클러스터 관리, 작업 제출, 라이브러리 관리, 사용자/그룹 관리
- JAR 파일(Java 또는 Scala) 및 Python 코드(scripts, egg, wheel 파일)를 제출하여 스트리밍/배치 작업으로 예약 및 실행
- 자주 사용하는 편집기/IDE로 로컬에서 개발하고 Databricks에 연결: Visual Studio, PyCharm, IntelliJ, RStudio, Jupyter, Zeppelin 등
- 소스 코드 관리(SCM) 통합: 휴대용 컴퓨터에서 로컬로 개발, Databricks로 가져오기/내보내기, SCM 도구를 사용한 체크인/체크아웃
- 지속적 통합/지속적 전달(CI/CD)
- 타사 스케줄러와 통합하여 고급 DAG/워크플로우 생성(예: Azure Data Factory, Apache Airflow)

사용자는 REST API 및 CLI에서 토큰을 사용하여 인증합니다. 토큰은 UI와 Tokens API를 사용하여 생성할 수 있습니다. 토큰은 .netrc 파일 및/또는 .databrickscfg 파일로 로컬에 저장할 수 있습니다. 사용자가 여러 배포에 액세스할 수 있으므로 파일에도 여러 개의 항목이 있을 수 있습니다. 예를 들어 다음과 같습니다.

.netrc

```
## Dev
machine eastus2.azuredatabricks.net
login token
password dapi[redacted]2b69ca
## Prod
machine eastus2.azuredatabricks.net
login token
password dapi[redacted]3d213
```

.databrickscfg

```
[DEV]
host = https://eastus2.azuredatabricks.net
username = token
password = dap[redacted]d97c64
[STAGING]
host = https://eastus2.azuredatabricks.net
username = token
password = dapi[redacted]b69ca
[PROD]
host = https://eastus2.azuredatabricks.net
username = token
password = dapi[redacted]8dca1
```

REST API에 사용에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

명령줄 인터페이스 설치 및 사용에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

챕터 2: 플랫폼 관리

보안 및 거버넌스

Databricks를 사용하면 UI 또는 REST API를 통해 플랫폼 액세스 권한을 중앙에서 관리, 제어하는 방법으로 사용자 액세스 권한을 부여, 거부, 취소할 수 있습니다.

- RBAC를 사용하여 데이터베이스, 테이블, 뷰에 대한 데이터 액세스 제어
- Cloud IAM 통합으로 파일 액세스
- 감사 로그를 사용하여 사용자 활동 모니터링
- 타사 데이터 카탈로그를 통합하여 마스터 데이터 관리, 리니지, 데이터 탐색, 데이터 프로파일링, 데이터 분류 제공 공급업체별 액세스, 오픈 소스 액세스.

Alation Amundsen IMMUTA

PRIVACERA Prophecy talend

하둡 에코시스템 경험이 있다면, 아마 RBAC(Role-Based Access Control, 역할 기반 액세스 제어)와 ABAC(Attribute-Based Access Control)는 이미 친숙한 개념일 것입니다. Databricks는 플랫폼과 데이터 양쪽 모두에 RBAC를 제공합니다. Databricks에서는 액세스 제어 목록(ACL)을 사용하여 워크스페이스, 노트북, 클러스터, 풀, 작업 및 Spark SQL 테이블 액세스 권한을 구성할 수 있습니다.

플랫폼 레벨에서 감사 로그를 활성화하여 사용자와 플랫폼의 모든 상호작용을 추적하는 것이 좋습니다.

자세한 정보는 온라인 문서를 참조하세요.

AWS AZURE

현재 Apache Ranger를 사용해서 데이터에 RBAC 및 ABAC 액세스를 제공하고 있을 수도 있습니다. Databricks에서는 데이터베이스 뷰를 사용하여 해당 뷰에 액세스 권한을 제공함으로써 RBAC와 유사한 기능을 제공합니다. 예를 들어, 민감한 정보가 포함된 기본 Spark SQL 테이블에 액세스할 권한이 있는 사람을 관리하고 싶다면 하나 이상의 Spark SQL 뷰를 생성할 수 있습니다. 이 뷰를 사용하면 기본 테이블에서 데이터를 여러 가지 방식으로 조회할 수 있습니다. 예를 들어, 로직이 포함된 뷰를 만들어 특정 필드를 해시 처리하고 특정 행을 필터링한 다음, 기본 테이블과 뷰에서 테이블 ACL을 적용하여 특정 그룹과 사용자의 액세스를 제한할 수 있습니다. 이 기능은 Python 및 SQL 워크로드를 실행하는 클러스터에 제공됩니다.

Databricks에서는 Unity Catalog를 출시할 예정입니다. 이를 통해 속성 기반 액세스 제어, 감사 및 리니지 정보를 제공합니다.

RBAC 및 ABAC 기능을 바로 사용하고 싶은 분에게는 Immuta 및 Privacera와의 파트너십을 통해 Ranger 통합을 제공합니다. 이 통합에 대한 자세한 내용은 Databricks 솔루션 아키텍트에게 상담하세요.

Databricks 보안에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

테이블 액세스 제어에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

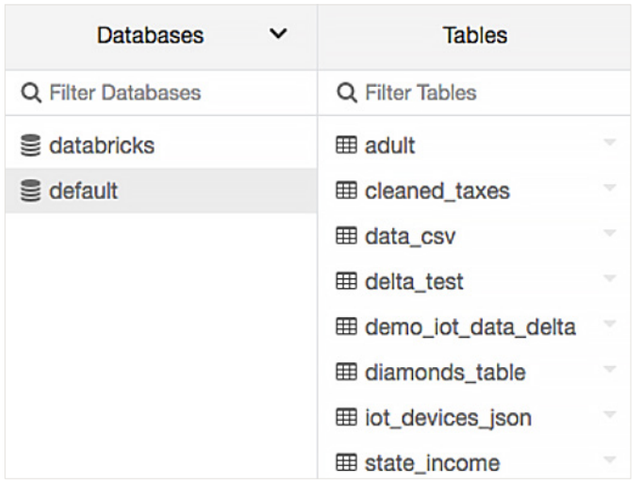
데이터 소스에 안전하게 액세스하는 방법은 이 가이드의 "데이터 소스" 섹션을 참조하세요.

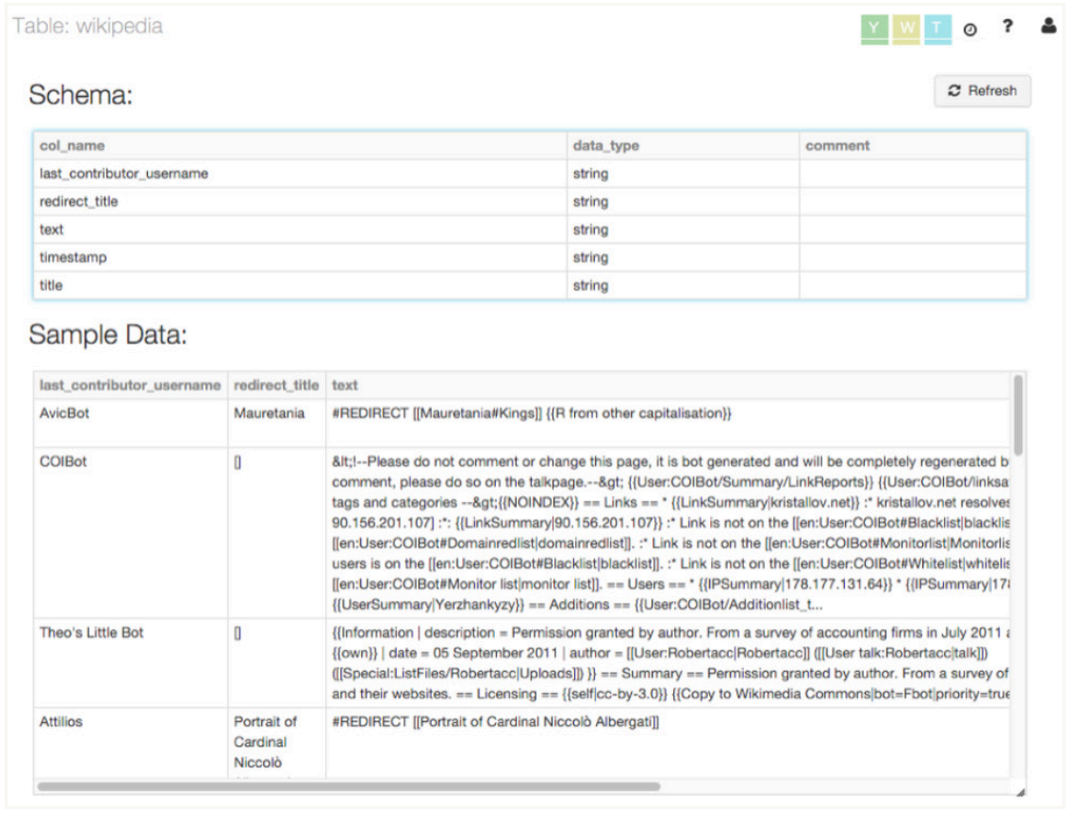
챕터 2: 플랫폼 관리

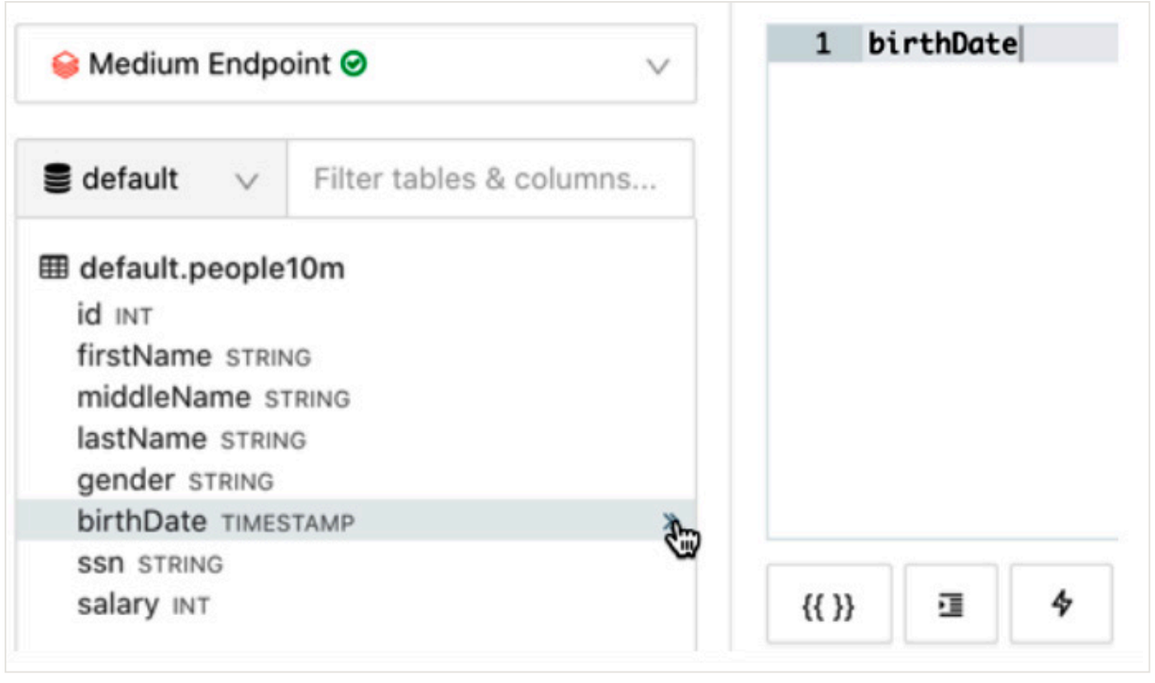
데이터 탐색 및 감사

Databricks에서는 Hive 메타스토어(임베디드 또는 외부)를 통해 데이터 카탈로그를 탐색할 수 있습니다. 이렇게 하면 여러 데이터베이스를 이동할 수 있고, 각각의 데이터베이스에 이용 가능한 테이블이 목록으로 표시되며 테이블마다 스키마와 데이터 샘플이 기재됩니다.

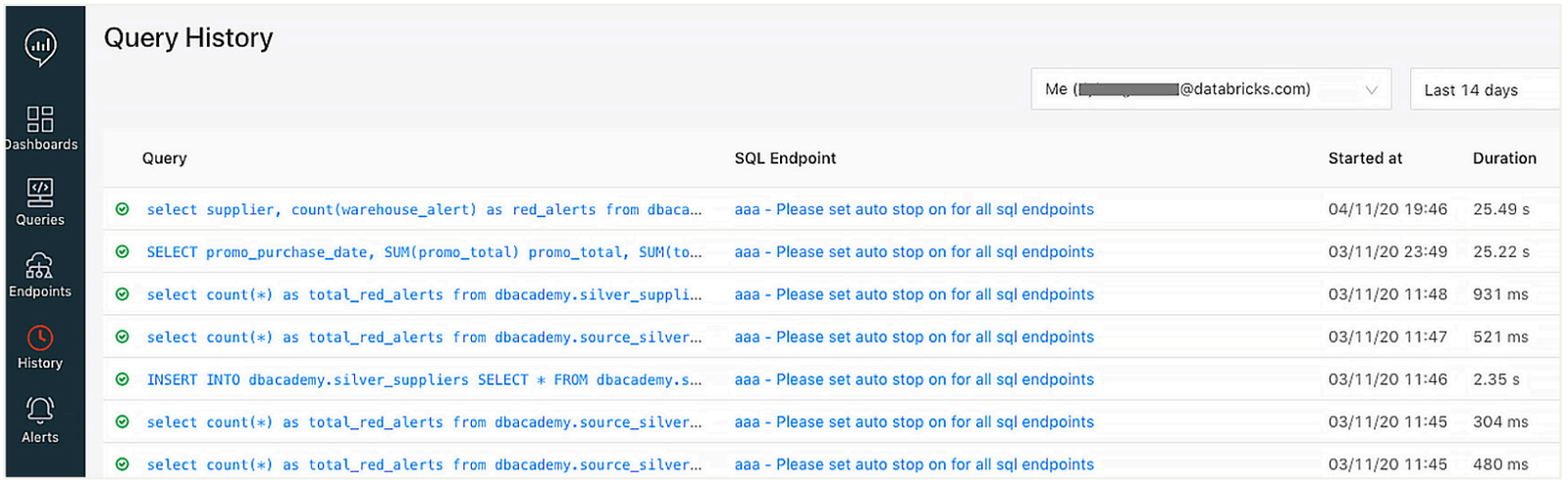
워크스페이스







Databricks SQL의 기록 탭에는 모든 실행 쿼리, 각 쿼리의 작성자와 실행 상태가 표시됩니다. 이 방법으로 데이터 액세스를 손쉽게 추적할 수 있습니다.



챕터 03

애플리케이션 개발, 테스트 및 배포

데이터 소스

데이터 마이그레이션

Hive 메타스토어

HiveQL vs. Spark SQL

Delta Lake를 사용한 데이터 파이프라인 최적화

사용자 정의 기능

Sqoop

Databricks 기반 Spark 코드 개발

코드 개발을 위한 노트북 및 IDE

소스 코드 관리 및 CI/CD

작업 예약 및 제출

챕터 3: 애플리케이션 개발, 테스트 및 배포

데이터 소스

Databricks는 클라우드 스토리지에 읽고 쓰는 데 최적화되어 있습니다. 예를 들어 S3, Azure Blob, Azure Data Lake Storage Gen 1 및 Azure Data Lake Storage Gen 2(ADLS) 등이 대표적입니다.

AWS에서 Databricks는 S3에 읽기와 쓰기에 최적화되어 있습니다. Databricks는 S3 클라우드 스토리지 외에 다른 스토리지 엔드포인트에도 읽고 쓸 수 있습니다. 예를 들어 관계형 데이터베이스(Oracle, Redshift, SQL Server, Teradata), HDFS, Apache Hive, NoSQL(HBase, Cassandra, MongoDB), 인메모리 캐시(Redis, RocksDB), S3 SQS, 메시지 버스(Kafka, Kinesis), 파일(구분된 텍스트 파일, JSON, Parquet, ORC, Avro) 등이 있습니다. 데이터 소스는 클라우드 내일 수도, 온프레미스일 수도 있습니다.

Databricks에서 지원하는 데이터 소스에 대한 자세한 내용은 온라인 문서를 참조하세요.

Azure에서 Databricks는 ADLS Gen 2를 사용해야 최적의 성능을 낼 수 있습니다. Azure Databricks는 Azure 클라우드 스토리지 외에 다른 스토리지 엔드포인트에도 읽고 쓸 수 있습니다. 예를 들어, 관계형 데이터베이스(Oracle, SQL Server, Teradata), HDFS, Apache Hive, NoSQL(HBase, Cassandra, MongoDB), 인메모리 캐시(Redis, RocksDB), 메시지 버스(Kafka), 파일(구분된 텍스트 파일, JSON, Parquet, ORC, Avro) 등이 있습니다. 또한, Azure Databricks는 여러 Azure 엔드포인트(예: SQL 데이터 웨어하우스, Cosmos DB, Event Hub, IoT Hub 등)와도 기본적으로 통합됩니다. 데이터 소스는 클라우드 내일 수도, 온프레미스일 수도 있습니다.

Azure Databricks에서 지원하는 데이터 소스에 대한 자세한 내용은 온라인 문서를 참조하세요.

하둡 사용자는 Optimized Row Columnar(ORC) 파일 형식에 친숙한데, Databricks에서도 이 파일 형식을 지원합니다.

다만 Databricks는 클라우드 스토리지에서 Parquet와 Delta에 최적화되어 있으며, 클라우드 스토리지에 성능, 안정성과 일관성을 부여하려면 Parque 기반 오픈 소스 스토리지 레이어인 Delta를 사용하는 편을 권장합니다. 데이터를 Spark DataFrame으로 읽어서 Delta 형식으로 저장하면 ORC 파일을 Delta로 쉽게 변환할 수 있습니다. 자세한 내용은 이 가이드의 “Delta” 섹션을 참조하세요.

Databricks는 Databricks File System, 이른바 DBFS라는 클라우드 스토리지를 기반으로 한 분산형 파일 시스템을 구축했습니다. DBFS는 Databricks Utilities(DBUtils)와 사용할 때는 클라우드 스토리지에 대한 추상화가 되며, 다음과 같은 기능을 제공합니다.

- 클라우드 스토리지에서 데이터를 마운트하여 자격 증명 없이 데이터에 액세스 가능(NFS 마운트와 유사)
- 클라우드별 API 대신 UNIX 디렉터리 및 파일 명령을 사용하여 클라우드 스토리지와 상호작용 가능
- 파일을 오브젝트 스토리지에 영구 저장하므로 클러스터를 종료해도 데이터 손실 방지
- Databricks 워크스페이스에서 클러스터에 마운트 지점을 생성하면, 기본적으로 그 워크스페이스의 모든 클러스터에서 해당 마운트에 액세스할 수 있습니다

자세한 내용은 이 가이드의 다음 섹션을 참조하세요.

- AWS 클라우드 스토리지 액세스
- Azure 클라우드 스토리지 액세스

S3 클라우드 스토리지 액세스

액세스 키

- 간편한 설정과 사용
- 사용 시 주의 필요 - 즉, 노트북 권한을 사용하여 자격 증명이 포함된 노트북을 아무도 읽을 수 없게 설정
- 키를 포함한 Secret API를 사용하여 실제 키 값 숨김
- 키 사용 방법

IAM 역할

- 보안 강화
- IAM 역할 하나만 Databricks 클러스터에 연결할 수 있다는 점에서 제한적
- IAM 역할을 사용하는 방법

IAM 자격 증명 패스스루

- 각 사용자가 자신의 자격 증명을 자기 AWS 계정과 동기화하여 어느 S3 버킷에 액세스 권한이 있는지 인증
- ID 제공자와의 통합 필요: SAML SSO를 사용한 AWS ID 연동
- IAM 자격 증명 패스스루 사용

Azure 클라우드 스토리지 액세스

공유된 키 또는 공유된 액세스 서명

- 간편한 설정과 사용
- 사용 시 주의 필요 - 즉, 노트북 권한을 사용하여 자격 증명이 포함된 노트북을 아무도 읽을 수 없게 설정
- 키를 포함한 Secret API를 사용하여 실제 키 값 숨김
- Azure Blob 스토리지로 키를 사용하는 방법
- Azure Data Lake 스토리지로 키를 사용하는 방법

AZURE ACTIVE DIRECTORY(AAD) 자격 증명

- AAD 자격 증명을 사용하여 ADLS Gen 2에서 직접 데이터에 액세스합니다

클라우드 스토리지 엔드포인트는 사용자가 소유하고 Databricks에서는 직접 액세스할 수 없습니다. 단, DBFS Root라는 공유 클라우드 스토리지 위치만은 예외로, Databricks도 여기에는 읽기/쓰기 권한이 있습니다. DBFS Root는 필수이고, Databricks에서 메타데이터와 로그, 데이터를 저장하는 데 사용합니다. 사용자 자체 보유 데이터는 DBFS Root에 저장하지 않는 것이 좋습니다. 기본적으로, 코드가 위치를 지정하지 않고 데이터를 쓰는 경우 이 데이터는 DBFS Root 스토리지 위치에 저장됩니다. 따라서 항상 데이터를 저장할 위치를 지정하는 것이 좋습니다. 예를 들어 다음의 코드는 위치를 지정하지 않고 파일을 작성합니다. 따라서 이 데이터는 DBFS Root에 저장됩니다.

```
1 df = spark.read.json("dbfs:/bmathew/data/clickstream_sample/file4.json")
2 df.write.mode("overwrite").parquet("file4.parquet")
```

▸ (2) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [channel: string, event_client_timestamp: string ... 15 more fields]

Command took 3.34 seconds -- by binu.mathew@databricks.com at 4/1/2020, 11:04:32 AM on bmathew-test

데이터를 DBFS Root에 저장하고 싶지 않다면 DBFS Root가 아닌 마운트 지점을 지정해야 합니다.

Cmd 1

```
1 df = spark.read.json("dbfs:/bmathew/data/clickstream_sample/file4.json")
2 df.write.mode("overwrite").parquet("/mnt/clickstream_data/file4.parquet")
```

▸ (2) Spark Jobs

▸ df: pyspark.sql.dataframe.DataFrame = [channel: string, event_client_timestamp: string ... 15 more fields]

Command took 3.03 seconds -- by binu.mathew@databricks.com at 4/1/2020, 11:07:20 AM on bmathew-test

Cmd 2

자세한 정보는 온라인 문서를 참조하세요.

DBFS 및 DBUtils

AWS AZURE

DBFS Root

AWS AZURE

클라우드 스토리지 연결 방법은 “클라우드 스토리지 연결” 노트북을 참조하세요. 이 노트북은 다음과 같은 문서와 함께 제출됩니다.

AWS AZURE

챕터 3: 애플리케이션 개발, 테스트 및 배포

데이터 마이그레이션

Databricks는 클라우드 오브젝트 스토리지에 최적화되어 있습니다. 예를 들어 Amazon Web Services의 S3, Microsoft Azure의 Blob 및 Azure Data Lake Storage Generation 2, Google Cloud Storage가 있습니다. Databricks는 클라우드 스토리지 외에 다른 스토리지 엔드포인트에도 읽고 쓸 수 있습니다. 예를 들어, 관계형 데이터베이스(Oracle, SQL Server, Teradata), HDFS, Apache Hive, NoSQL(HBase, Cassandra, Neo4j, MongoDB), 인메모리 캐시(Redis, RocksDB), 메시지 버스(Kafka), 파일(구분된 텍스트 파일, JSON, Parquet, ORC, Avro) 등이 있습니다. 데이터 소스는 클라우드 내일 수도 있고 온프레미스일 수도 있지만, Databricks는 클라우드에 최적화되어 있습니다.

데이터 마이그레이션을 시작하려면 먼저 이중 통합 전략을 확인해야 합니다. 이미 데이터를 하둡에 넣도록 정의한 프로세스가 있을 가능성도 있기 때문입니다. 이는 타사 수집 도구나 내부에서 구축한 프레임워크를 통해 구현할 수 있습니다. 간단한 방법은 대상을 아예 옮겨서 HDFS와 클라우드 스토리지에 데이터를 넣는 것입니다. 최초 데이터 피드를 가져오면 데이터에 추가적인 백업 위치가 생깁니다. 또한, Databricks로 클라우드 내에서 사용할 수 있는 새로운 고급 분석 기능도 제공합니다.

다음 단계는 과거 데이터를 마이그레이션하는 것입니다. 이 단계는 HDFS에 존재하는 데이터 용량에 따라 다소 시간이 걸릴 수 있습니다. 가능하다면, 하둡으로부터 마이그레이션해야 하는 사용 사례와 데이터 세트의 우선 순위를 일치하는 것이 좋습니다. 이렇게 하면 데이터를 클라우드로 옮겨야 하는 순서를 파악하는 데 도움이 됩니다.

HDFS에서 과거 데이터를 클라우드로 옮기는 방법은 두 가지가 있는데, 하나는 푸시 전략이고 또 하나는 풀 전략입니다. 일반적으로 풀 전략보다는 푸시 전략을 사용하는데, 데이터 소유자와 정보 보안팀이 데이터를 클라우드에 전송하는 방식과 시점을 더욱 주도적으로 관리할 수 있기 때문입니다. 고객에 따라 풀 전략을 택할 수도 있는데, 이 경우 워크플로우를 오직 클라우드에서만 관리해야 합니다.

HDFS에서 클라우드로 데이터를 마이그레이션하는 옵션은 다음과 같습니다.

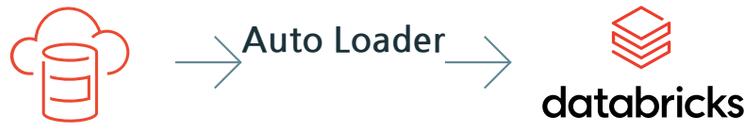
최대 몇백 테라바이트의 데이터 볼륨:

- 온프레미스와 클라우드 사이에 안전한 VPN 연결을 만들거나 클라우드 서비스 제공업체(CSP)에서 제공하는 사설 전용 연결 사용
- 메시지 버스로 IoT 데이터 스트리밍
- 클라우드 서비스 제공업체 유틸리티 및 기타 도구(예: DistCp)를 사용하여 하둡 스토리지(HDFS)에서 클라우드 스토리지로 파일 이동
- HDFS를 클라우드 스토리지와 동기화하는 기능이 있는 타사 도구(예: WANdisco) 사용

페타바이트급 이상의 데이터 볼륨:

- 디스크 우편 발송을 허용하는 CSP 서비스를 이용하여 해당 서비스 측에서 디스크를 클라우드 스토리지에 로드
- 사용자 데이터 센터에서 자사 데이터 센터로 디스크를 운송해 주는 CSP 서비스를 이용하여 이 서비스 측에서 디스크를 클라우드 스토리지에 로드
- 일부 CSP는 다른 데이터 마이그레이션 서비스를 제공할 수 있습니다. 각자 요구 사항에 가장 적절한 서비스를 찾아보세요.

데이터를 클라우드 스토리지로 로드하고 나면 Databricks에서 제공하는 Auto Loader 서비스를 사용하여 데이터를 Databricks로 빠르고 간편하게 가져올 수 있습니다.



- Auto Loader in AWS
- Auto Loader in Azure

데이터를 클라우드로 마이그레이션하는 것은 시간이 오래 걸리고 데이터 팀에게는 까다로울 과정일 수 있습니다. Databricks는 도구를 제공하는 파트너 공급업체와 협력하여 온프레미스 스토리지에서 클라우드 스토리지로의 데이터 마이그레이션을 안전하게 자동화합니다. 따라서 Databricks 클라우드로의 마이그레이션에 다음과 같은 중대한 경제적 효과가 발생합니다.

- 비용 절감
- 수동 방식 대비 마이그레이션 일정 2~3배 단축
- 전반적인 마이그레이션 일정 단축

자세한 정보는 Databricks에 문의하세요.

Azure의 경우, 데이터 전송에 적절한 솔루션을 선택하세요.
AWS의 경우 데이터 마이그레이션 서비스를 참조하세요.

챕터 3: 애플리케이션 개발, 테스트 및 배포

Hive 메타스토어

메타스토어 RDBMS는 Apache Hive 메타데이터의 중앙 리포지토리입니다. 스키마 정보(예: 열 이름, 데이터 유형, 데이터 파일 위치, 파티션, 기타 메타데이터)를 포함하는 테이블 메타데이터를 저장합니다. Databricks는 Hive 메타스토어를 사용하여 Spark SQL 테이블 메타데이터를 저장합니다. Databricks의 기본 구성은 관리형 Hive 메타스토어를 생성하여 유지관리하는 것입니다. 이는 Databricks에서 워크스페이스별로 관리합니다. 고객은 기존 Hive 메타스토어를 사용하거나, Databricks를 외부 Hive 메타스토어로 삼아 통합하거나 둘 중에서 선택할 수 있습니다. 다양한 버전의 Hive 메타스토어가 지원되며, 다양한 백엔드 RDBMS (예: MySQL, PostgreSQL, Oracle, SQL Server)를 사용할 수 있습니다. 기존 하둡 고객은 SQL Server를 백엔드 Hive 메타스토어로 사용하는 것이 일반적이며, Databricks는 최신 버전 Hive(가이드 작성 당시 기준 3.1)를 사용한 SQL Server를 지원합니다.

Databricks를 포함한 기본 Hive 메타스토어는 Databricks에서 호스팅하며, 현재 각 워크스페이스에서 당사 호스팅 Hive 메타스토어로의 연결에 250개의 제한이 있습니다. 각 Databricks 클러스터는 메타스토어 연결을 2개 이상 엽니다. 따라서 워크스페이스에 동시에 실행되는 클러스터가 125개 있으면 연결 한도인 250개에 도달합니다. 자체 외부 Hive 메타스토어를 사용하는 고객의 경우, 워크스페이스와 외부 Hive 메타스토어의 연결 개수는 알려진 제한이 없습니다. 고객이 외부 Hive 메타스토어를 관리하므로 데이터베이스 서버 설정을 조정하여 연결 개수를 조절할 수 있습니다.

자세한 내용은 이 가이드의 다음 섹션을 참조하세요.

AWS 외부 Hive 메타스토어 통합
Azure 외부 Hive 메타스토어 통합

Hive 메타스토어 마이그레이션

기존 Hive 메타스토어가 있고 여기에서 테이블 정의의 일부 또는 전체를 Databricks로 마이그레이션하고자 하는 경우, RDBMS 기능과 Hive 명령을 사용하면 됩니다. 이 작업을 수행하는 한 가지 방법으로 기존 Hive 메타스토어 전체를 새 RDBMS로 옮기는 방안이 있습니다.

1. 최신(기존) RDBMS에서 Hive 데이터베이스 덤프를 생성하여 이것을 파일에 씁니다. 예를 들어 MySQL을 사용하고 Hive 데이터베이스의 이름이 “hive”라면 mysqldump -u root hive >> my_dump_outputfile.sql이라고 명령을 작성합니다.

2. 파일 내에서 몇 가지 검색해서 바꾸기를 거쳐야 Databricks와 호환됩니다.

   - 테이블 데이터 파일의 위치가 Databricks 내의 위치와 상응해야 합니다. 위치 경로의 경우, 마운트 지점 위치(예: /path_to_ my_directory)를 사용하면 됩니다. (마운트 지점에 대한 자세한 내용은 “데이터 소스” 섹션을 참조하세요.) 또는, 위치를 지정할 때 클라우드 서비스 제공업체의 파일 시스템을 사용할 수도 있습니다(예: AWS에서 S3를 사용할 경우 s3a://my_bucket/path_to_my_files). 검색하여 바꾸기 함수를 작성하여 위치 경로를 조정합니다.
   - Databricks에서 ORC 파일을 계속 사용해도 되지만, Databricks는 오픈 소스 Parquet을 사용하는 Delta Lake 파일 형식에 최적화되어 있으므로 Delta Lake를 사용하는 것이 좋습니다. 이 형식에 대한 자세한 내용은 “Delta Lake” 섹션 참조하고 테이블 생성 명령에 대한 자세한 내용은 “Spark SQL” 섹션을 참조하세요.

- Delta Lake 대신 Parquet을 사용하고 순수 오픈 소스 Parquet 형식을 계속 사용하고자 한다면 Hive 스타일 구문 “STORED AS”를 Databricks에서 사용하지 않는 것이 좋습니다. 올바른 데이터 구문은 “USING”입니다. 검색하여 바꾸기 함수를 작성하여 변경합니다. 테이블 생성 명령은 “Spark SQL” 섹션을 참조하세요.

3. 몇몇 다른 검색하여 바꾸기를 실행해야 할 수도 있지만 Hive에서 Spark SQL을 호환할 때 흔히 발생하는 문제는 위의 예시와 같습니다.

4. 검색하여 바꾸기를 완료했으면 새 RDBMS에 Hive 데이터베이스를 만듭니다. 예를 들어 MySQL을 사용하는 경우, 명령은 create database hive입니다.

5. Hive 스키마 도구를 실행하여 기존 메타스토어 Hive 스키마의 정확한 버전에 대해 스키마를 초기화합니다. 예를 들어 MySQL을 사용하는 경우 이런 형태가 됩니다. $HIVE_HOME/bin/schematool-dbType mysql-initSchemaTo <hive_version> -userName <user_name>-passWord <password> -verbose

6. 파일에서 Hive 데이터베이스를 가져옵니다. 예를 들어 MySQL을 사용하는 경우 이것은 mysql -u root hive < my_ dump_ outputfile.sql이 됩니다.

7. 스키마를 최신 스키마 버전으로 업그레이드: $HIVE_HOME/bin/schematool -upgradeSchema-dbType mysql -userName <user_name> -passWord <password> -verbose

스키마(데이터베이스 및 테이블)만 마이그레이션하려면 다음의 Spark 코드를 사용하여 DDL 파일을 생성할 수 있습니다. 데이터 위치와 구문(필요한 경우)을 수정한 다음, DDL 파일을 새 메타스토어로 내보냅니다.

```
dbs = spark.catalog.listDatabases()
for db in dbs:
  f = open(“your_file_name_{}.ddl”.format(db.
name), “w”)
 tables = spark.catalog.listTables(db.name)
  for t in tables:
    DDL = spark.sql(“SHOW CREATE TABLE {}.{}”.
format(db.name, t.name))
    f.write(DDL.first()[0])
    f.write(“\n”)
f.close()
```

Delta 테이블로 마이그레이션되지 않는 테이블의 경우 MSCK REPAIR TABLE을 실행해야 합니다.

AWS 외부 Hive 메타스토어 통합

이미 AWS를 사용하고 있다면 AWS Glue 데이터 카탈로그도 사용하고 있을 가능성이 있습니다. Databricks에서는 Glue 데이터 카탈로그를 메타스토어로 사용할 수 있습니다. 이 통합을 구성하는 방법에 대한 자세한 내용은 온라인 문서를 참조하세요.

"외부 MySQL Hive 메타스토어와 Databricks.dbc 통합" 노트북에서 MySQL의 기존 Hive 3.1.0 메타스토어와 Databricks를 사용하는 방법의 예시를 참조하세요. 이 노트북은 이 문서와 함께 제출됩니다.

Databricks에서는 SQL Server를 Hive 메타스토어의 백엔드로 사용하여 최신 버전 Hive를 지원합니다. "외부 SQL Server Hive 메타스토어와 Databricks.dbc 통합" 노트북에서 SQL Server의 기존 Hive 3.1.0 메타스토어와 Databricks를 사용하는 방법의 예시를 참조하세요. 이 노트북은 이 문서와 함께 제출됩니다.

외부 Hive 메타스토어와 Databricks를 사용하는 방법에 대한 자세한 내용은 온라인 문서를 참조하세요.

Azure 외부 Hive 메타스토어 통합

"외부 SQL Server Hive 메타스토어와 Databricks.dbc 통합" 노트북에서 SQL Server의 기존 Hive 3.1.0 메타스토어와 Azure Databricks를 사용하는 방법의 예시를 참조하세요. 이 노트북은 이 문서와 함께 제출됩니다.

"외부 MySQL Hive 메타스토어와 Azure Databricks.dbc 통합" 노트북에서 MySQL의 기존 Hive 3.1.0 메타스토어와 Azure Databricks를 사용하는 방법의 예시를 참조하세요. 이 노트북은 이 문서와 함께 제출됩니다.

외부 Hive 메타스토어와 Azure Databricks를 사용하는 방법에 대한 자세한 내용은 온라인 문서를 참조하세요.

챕터 3: 애플리케이션 개발, 테스트 및 배포

HiveQL vs. Spark SQL

Apache Hive는 데이터 웨어하우스 소프트웨어 프로젝트로, 본래 하둡 에코시스템용으로 개발되었습니다. Hive는 온프레미스와 클라우드 내에서 다양한 스토리지 매체(예: HDFS, Azure 클라우드 스토리지, Amazon Web Services S3 오브젝트 스토리지, Google Cloud 스토리지 등)와 함께 사용할 수 있습니다. Hive는 데이터를 행, 열, 데이터 유형을 포함한 테이블로 표현하는 추상화 계층을 제공하며 SQL 인터페이스, HiveQL로 쿼리하고 분석합니다. Hive는 인메모리 분산형 엔진 Apache Tez로 데이터를 처리합니다.

Apache Hive는 Hive LLAP와의 트랜잭션(ACID)을 지원합니다. 트랜잭션을 이용하면 동시에 여러 사용자/프로세스가 데이터에 액세스하여 생성, 읽기, 업데이트 및 삭제(CRUD) 작업을 하는 환경에서도 데이터의 일관된 보기를 보장합니다. Databricks는 Delta를 제공합니다. Delta는 트랜잭션(ACID) 보증을 제공한다는 면에서 Hive LLAP와 비슷하지만 이외에도 여러 가지 다른 장점을 제공하므로 데이터에 액세스할 때 성능과 안정성에 도움이 됩니다. Delta는 오픈 소스 프로젝트입니다. Delta에 대한 자세한 정보는 이 가이드 후반부를 참조하세요.

Spark SQL은 행, 열, 데이터 유형으로 구성된 테이블로 표현되는 정형 데이터와 상호작용하는 Apache Spark 모듈입니다. Spark SQL은 SQL 2003과 호환되며 Apache Spark를 분산형 엔진으로 사용해 데이터를 처리합니다. Spark SQL 인터페이스 외에 DataFrames API를 사용해도 Java, Scala, Python 및 R을 사용해 데이터와 상호작용할 수 있습니다.

Spark SQL은 HiveQL과 유사합니다. 두 가지 모두 ANSI SQL 구문을 사용하고, Hive 함수는 대부분 Databricks에서 실행됩니다. 여기에는 날짜/시간 변환과 파싱, 수집, 문자열 조작, 수학 연산, 조건부 함수를 위한 Hive 함수도 포함됩니다. Hive 전용 함수도 있는데, 이는 Databricks의 Spark SQL에 존재하지 않으므로 Spark SQL 함수로 변환해야 합니다. 모든 HiveQL ANSI SQL 구문은 Databricks의 Spark SQL에서 작동한다고 해도 무방합니다. 여기에는 ANSI SQL 집계 및 분석 함수 등이 대표적인 예입니다.

Hive는 Optimized Row Columnar(ORC) 파일 형식에 최적화되었고 Parquet도 지원합니다. Databricks는 Parquet과 Delta에 최적화되었습니다. 오픈 소스 Parquet을 파일 형식으로 사용하는 Delta 사용을 권장합니다.

HDFS를 사용한 HiveQL 테이블 생성의 예시

```
CREATE EXTERNAL TABLE CUSTOMER_DB.CUSTOMER
(USER_ID INT, USER_NAME STRING) STORED AS
PARQUET
LOCATION ‘/data/customer’;
```

오브젝트 스토리지를 사용한 Spark SQL Databricks 기반 테이블 생성

```
CREATE TABLE CUSTOMER_DB.CUSTOMER (USER_ID INT,
USER_NAME STRING)
STORED AS PARQUET
LOCATION ‘/data/customer‘;
```

EXTERNAL 키워드는 사용할 필요가 없습니다. 위치만 지정하면 이 테이블이 자동으로 외부 테이블이 됩니다. 위치 경로의 경우, 마운트 포인트 위치(예: as /path_to_my_directory)를 사용할 수 있습니다. (마운트 지점에 대한 자세한 내용은 "데이터 소스" 섹션을 참조하세요). 또는, 위치를 지정할 때 클라우드 서비스 제공업체의 파일 시스템 클라이언트를 사용해도 됩니다(예: AWS에서 S3를 사용할 경우 s3a://my_bucket/path_ to_my_files).

또는,

```
DROP TABLE IF EXISTS CUSTOMER_DB.MY_TABLE;
CREATE TABLE BMATHEW.MY_TABLE (USER_ID INT,
USER_NAME STRING)
USING DELTA
LOCATION '/data/customer';
```

마찬가지로 EXTERNAL 키워드를 사용하지 않아도 됩니다. Databricks 기반 Spark SQL에서 테이블을 생성할 때 핵심적인 차이는 Hive 구문은 "stored as"를 사용하지만 Databricks는 "using"을 사용한다는 점입니다. 현재 Hive LLAP를 사용 중이고 Databricks로 마이그레이션하고자 하는 경우, Delta — "USING DELTA"를 사용하는 것이 좋습니다. Delta는 오픈 소스인 트랜잭션(ACID)을 제공하고, 데이터 엔지니어링, 데이터 사이언스, BI 워크로드에서 데이터 액세스 시 성능, 안정성, 일관성을 향상해줍니다.

테이블 구성에 설정할 수 있는 옵션은 여러 가지가 있습니다. Hive 사용자에게 친숙하면서도 Databricks의 Spark SQL에서 지원되는 몇 가지 옵션은 다음과 같습니다.

위치 — 클라우드 스토리지 위치로 여기에 데이터 파일이 저장됩니다. 기본 경로는 항상 /user/hive/warehouse/의 기본 루트 blob 스토리지 계정에 있습니다. 위치를 지정하지 않으면 관리형 테이블로 생성됩니다. 즉, 테이블을 제거하면 모든 데이터 파일도 삭제됩니다. 위치를 지정하면 이 테이블은 비관리형 테이블이나 외부 테이블이 됩니다 즉, 테이블을 제거해도 데이터는 디렉터리에 남아 있습니다. 위치 경로의 경우, 마운트 지점 위치(예: /path_to_ my_directory)를 사용할 수 있습니다. (마운트 지점에 대한 자세한 내용은 "데이터 소스" 섹션을 참조하세요.) 또는, 위치를 지정할 때 클라우드 서비스 제공업체의 파일 시스템을 사용할 수도 있습니다(예: AWS에서 S3를 사용할 경우 s3a://my_bucket/path_to_my_files).

파티션 기준 — 테이블을 하나 이상의 열로 분할합니다. 항상 각 값이 낮은 파티션 열을 선택하세요(낮은 기수성).

클러스터 기준 — For — 값이 많은 열의 경우(높은 기수성), 버킷을 사용하는 것이 성능에 도움이 됩니다.

TBL 속성 — Hive와 유사한 추가 설정으로 특정 구성을 지정할 수 있습니다.

위의 속성을 사용하여 Databricks에서 Parquet으로 테이블을 생성하는 예시는 다음과 같습니다.

```
DROP TABLE IF EXISTS BMATHEW.MY_TABLE;
CREATE TABLE BMATHEW.MY_TABLE (
USER_ID INT,
USER_NAME STRING,
TRANSACTION_DATE DATE)
USING PARQUET
PARTITIONED BY (TRANSACTION_DATE)
CLUSTERED BY (USER_ID) SORTED BY (USER_ID) INTO
32 BUCKETS
LOCATION '/tmp/bmathew/test_hive_data'
TBLPROPERTIES ('compression'='snappy',
'owner'='bmathew');
```

다만, Databricks에서 버킷은 Delta가 아니라 Parquet을 사용할 때만 지원됩니다.

스키마 정의를 포함한 테이블 속성을 확인하는 방법:

```
DESCRIBE FORMATTED BMATHEW.MY_TABLE;
```

Hive 스타일 구문은 Databricks에서도 지원됩니다.

```
CREATE TABLE my_table STORED AS PARQUET AS
(select 1 as user_id);
```

그러나 Hive 스타일 구문(**stored as parquet**)은 사용하지 않는 것이 좋습니다. **using parquet** 구문은 Spark SQL 전용입니다. 이 테이블은 항상 Spark SQL Catalyst optimizer용 오픈 소스 외의 최적화를 사용합니다. 반면, **stored as parquet**은 Spark와 Hive에서 모두 사용할 수 있지만, 일부 Databricks 전용 Spark SQL 최적화는 예상대로 작동하지 않을 수 있습니다. 따라서 Delta를 사용하고 싶지 않다면 "USING PARQUET"를 사용하는 것이 좋습니다.

자세한 정보는 온라인 문서를 참조하세요.

Databricks 데이터베이스 및 테이블

AWS AZURE

Spark SQL

AWS AZURE

Hive 호환성

AWS AZURE

Delta를 사용하여 데이터를 저장하는 것이 좋습니다. 자세한 정보와 노트북 예시는 다음 섹션 "Delta Lake를 사용한 데이터 파이프라인 최적화"를 참조하세요.

챕터 3: 애플리케이션 개발, 테스트 및 배포

Delta Lake를 사용한 데이터 파이프라인 최적화

하둡은 데이터를 처리하기 위한 여러 가지 분산형 프로그래밍 프레임워크를 제공합니다. 여기에는 기존 로우 레벨 MapReduce API와 하이 레벨 프레임워크(예: Pig, Hive(Tez 사용))가 포함됩니다. 하둡은 Spark도 지원합니다. Databricks Delta Engine은 Spark를 사용하여 데이터를 쉽게 처리할 수 있습니다. Spark와 Databricks를 결합하면 오픈 소스 Spark 대비 성능이 6배 향상되기 때문입니다. Delta Engine은 100% Apache Spark 호환 벡터화 쿼리 엔진으로, Delta Lake에서 쿼리 성능을 상당히 가속화하고 레이크하우스 아키텍처를 손쉽게 도입해 확장할 수 있습니다.

Apache Hive는 Hive LLAP와의 트랜잭션(ACID)을 지원합니다. 트랜잭션을 이용하면 동시에 여러 사용자와 프로세스가 데이터에 액세스하여 생성, 읽기, 업데이트 및 삭제(CRUD) 작업을 하는 환경에서 데이터의 일관된 보기를 보장합니다. Databricks는 Delta Lake를 제공합니다. Delta Lake는 트랜잭션(ACID) 보증을 제공한다는 면에서 Hive LLAP와 비슷하지만, 이외에도 여러 가지 다른 장점을 제공하므로 데이터에 액세스할 때 성능과 안정성에 도움이 됩니다. Delta는 오픈 소스 프로젝트입니다.

Databricks에서 Spark SQL 테이블을 생성할 때 Delta Lake를 사용하는 것이 좋습니다. Delta는 오픈 소스 스토리지 형식으로, 최적화된 Spark SQL 테이블을 생성하고 다음과 같은 장점을 제공합니다.

- 오픈 소스 Parquet를 기본 파일 형식으로 사용
- Parquet 파일의 트랜잭션 로그 생성
- 트랜잭션을 지원하기 위한 ACID 속성(관계형 데이터베이스에서 제공)
- 여러 사용자가 데이터에 동시에 액세스하여 생성, 읽기, 업데이트 및 삭제(CRUD) 작업을 수행하도록 허용하므로 데이터의 일관성과 안정성 보증
- 데이터 건너뛰기 인덱스를 적용하여 읽기 성능 개선
- VM의 로컬 SSD 드라이브에서 데이터를 캐싱함으로써, 나중에 클라우드 스토리지로 다시 연결하지 않고 VM 디스크에서 데이터를 가져와 읽을 수 있습니다. 이 기능을 사용하면 쿼리 성능이 대폭 향상되고, 스토리지 최적화 VM을 시작할 때만 작동합니다.
- 데이터 버전 관리를 제공하여 시간 이동(예: 롤백) 지원
- 스키마 적용
- 스키마 진화
- Z-Ordering이라는 클러스터 인덱스 사용
- 버킷을 지원하지 않지만 Delta에서 파티션, 데이터 건너뛰기, Z-Ordering을 사용하여 성능 향상

Delta를 사용하여 데이터 처리 워크로드를 최적화하는 방법은 “Databricks.dbc에서의 Delta” 노트북을 참조하세요. 이 노트북은 이 문서와 함께 제출됩니다.

AWS AZURE

Databricks 기반 Delta에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

기존 데이터를 Delta 형식으로 마이그레이션하는 방법에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

Databricks에서는 얼마 전 Delta Live Tables(DLT)를 공개했습니다. 이 기능을 사용하면 Delta Lake에서 고품질 데이터를 제공하는 안정적인 데이터 파이프라인을 손쉽게 구축하고 관리할 수 있습니다. DLT를 이용하면 데이터 엔지니어링 팀에서 선언적 파이프라인 개발, 자동 테스트 및 모니터링과 복구를 위한 심층적인 가시성을 얻어 ETL 개발과 관리를 간소화할 수 있습니다. 이 기술은 가시성, 데이터 품질 및 계통 정보가 나와 있지 않은 기존 하둡 데이터 파이프라인을 매우 간소화합니다.

자세한 정보는 Delta Live Tables 페이지를 참조하세요.

챕터 3: 애플리케이션 개발, 테스트 및 배포

사용자 정의 함수

고객은 Hive에 사용자 정의 함수(UDF)를 구현하여 기본 기능을 확장하는 경우가 많습니다. 개발자가 UDF를 사용하면 새 함수를 쓴 낮은 수준의 언어(예: Java, Scala)를 추상화하여 높은 수준의 언어(예: SQL 등)로 사용할 수 있습니다. Databricks 기반 Spark는 Spark SQL과 UDF를 통합하는 옵션이 있습니다.

조금만 변경하면 같은 Java UDF를 Databricks의 Hive에서도 사용할 수 있습니다(필요한 경우). 이를 위해서는 Java 소스 코드로 다음을 가져와야 합니다.

```
import org.apache.hadoop.hive.ql.exec.UDF;
```

위의 패키지는 항상 필수입니다. 가져오기가 필요할 수도 있습니다.

```
import org.apache.hadoop.io.*;
```

JAR 파일을 DBFS에 업로드하고, 클러스터에 연결된 JAR 파일로 클러스터를 시작한 다음, SQL 셀에 JAR 파일 경로를 추가하고 임시 함수를 생성해야 합니다. Azure의 UDF 예시를 참조하거나 AWS 아카이브 파일을 참조하세요. 이 노트북은 이 문서와 함께 제출됩니다.

또한, Databricks에서 Python과 Scala를 사용하여 UDF를 생성하고 Spark SQL을 통해 호출할 수 있습니다.

Python

AWS Azure

Scala

AWS Azure

이 가이드 앞에 제시한 링크에서 설명하였듯이 Databricks에서 UDF를 생성하면 SparkSession에서만 액세스할 수 있습니다. 예를 들어 동시성이 높고 공유된 클러스터에서 다른 사용자는 자체 SparkSession이 있어서 UDF에 직접 액세스할 수 없습니다. 공통 UDF를 공유하고 싶다면 모든 사람이 동일한 SparkSession을 공유해야 합니다. 다른 사용자가 UDF에 액세스하려면 클러스터에 다음의 구성 설정을 사용하여 SparkSession을 공유해야 합니다.

```
spark.databricks.session.share true
```

Java, Scala 또는 Python을 사용하여 코드를 작성하고 동일한 함수를 실행하는 라이브러리를 생성한 다음, 이 라이브러리를 클러스터에 연결할 수도 있습니다. 이렇게 하면 SparkSession을 공유할 필요가 없습니다.

여기서 한 가지 중요한 점을 지적하자면, UDF가 벡터화되어 있지 않으므로 한 번에 한 행씩 데이터를 다룬다는 사실입니다. Java와 Scala로 작성된 UDF는 Python으로 작성된 것보다 성능이 우수합니다. Java와 Scala로 작성된 함수는 Java Virtual Machine(JVM)에서 사용하지만, Python을 사용하면 Spark에서 JVM 프로세스의 데이터를 Python에서 판독 가능한 형식으로 직렬화해야 합니다. 이런 직렬화는 성능을 저하합니다. 성능이 허용치 이상으로 저하되면 Java 또는 Scala로 함수를 작성하는 것이 좋습니다. 그래도 Python에서 함수를 호출할 수 있는 것은 마찬가지입니다.

Databricks에서 pandas와 Apache Arrow를 사용하면 UDF를 벡터화할 수 있습니다. pandas UDF 사용에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

챕터 3: 애플리케이션 개발, 테스트 및 배포

Sqoop

Sqoop은 보이지 않는 곳에서 MapReduce를 실행합니다. 바로 이 Sqoop 때문에 MapReduce를 여전히 배포하고 있습니다. 많은 고객이 Sqoop을 떠나 Spark를 사용하여 관계형 시스템에서 직접 데이터를 읽기 시작했습니다. Spark의 데이터베이스에서 읽어오는 구문은 매우 간단하고, 데이터 처리 방식과 대상에 대한 유지 속성이 매우 유연합니다.

JSpark 코드에서는 JDBC 소스로 Sqoop 호출을 대체하면 됩니다. 이 Spark 코드는 Databricks 노트북에 저장되거나 코드 아티팩트(예: JAR, python whl)로 패키지화됩니다.

온라인 문서를 참조하세요. 예시 호출은 다음과 같습니다.

```
val jdbcDF = spark.read
  .format(“jdbc”)
.option(“url”, “jdbc:postgresql:dbserver”)
.option(“dbtable”, “schema.tablename”)
.option(“user”, “username”)
.option(“password”, “password”)
.load()
```

Databricks를 사용하면 Secrets를 활용하여 자격 증명이 코드에 노출되지 않게 방지할 수 있습니다.

Spark JDBC 소스에서도 Sqoop과 유사한 옵션을 지정할 수 있습니다. 예컨대 맞춤형 선택 쿼리, 읽기 및 배치 쓰기 용량 가져오기, 분리 설정 등이 있습니다.

Sqoop은 Sqoop Jobs를 통해 증분 로드를 제공하며, 내부적으로는 필드를 추적하여 새 데이터를 확인할 수 있습니다. 이 필드는 보통 타임스탬프이며, 계속 증가하는 시퀀스 ID일 수도 있습니다. Sqoop은 새 데이터를 대상 위치로 가져와서 가장 큰 값을 이 필드용으로 유지합니다. 그런 다음 이 값을 소스 테이블에서 새 데이터를 검색하는 데 사용합니다. Spark는 이 함수를 기본으로 지원하지 않습니다. 코드에서 “last modified timestamp” 필드나 시퀀스 ID를 추적하여 JDBC 소스로부터 데이터를 추출하는 데 사용한 SQL 쿼리를 수정해야 합니다.

챕터 3: 애플리케이션 개발, 테스트 및 배포

Databricks 기반 Spark 코드 개발

하둡 사용자가 JAR 파일과 스크립트를 통해 Spark 작업을 하둡 클러스터에 제출하는 경우, 각각 자체적인 SparkContext를 얻게 되지만 Databricks의 경우 Databricks 클러스터 하나에서 모든 사용자가 단 하나의 SparkContext를 공유합니다. 하둡과 Databricks 양쪽 모두 각각의 사용자에게 SparkSession을 하나씩 제공하는 것은 같습니다. Databricks에서 작업을 실행하면(Databricks 노트북을 통해서든, 자체적인 Java/Scala JAR 파일이나 Python 스크립트를 DBFS에 업로드해서든(개별적인 Python 스크립트 또는 wheel이나 egg 파일) 무관) 플랫폼이 사용자 대신 SparkContext를 만들어줍니다. Databricks는 SparkContext를 초기화하므로 새 SparkContext를 호출하면 코드가 실패합니다. 예를 들어 다음의 코드는 오류를 반환합니다.

```
Cmd 1

1  from pyspark import SparkConf, SparkContext
2
3  conf = (SparkConf()
4          .set("spark.executor.memory","2g"))
5  sc = SparkContext(conf = conf)

⊞ValueError: Cannot run multiple SparkContexts at once; existing SparkContext(app=Databricks Shell,
  master=spark://10.0.241.108:7077) created by __init__ at /local_disk0/tmp/1585629397371-0/PythonShe
  ll.py:1335

Command took 0.14 seconds -- by binu.mathew@databricks.com at 3/31/2020, 12:50:16 PM on bmathew-test
```

Databricks에서 생성한 공유 SparkContext를 사용하세요.

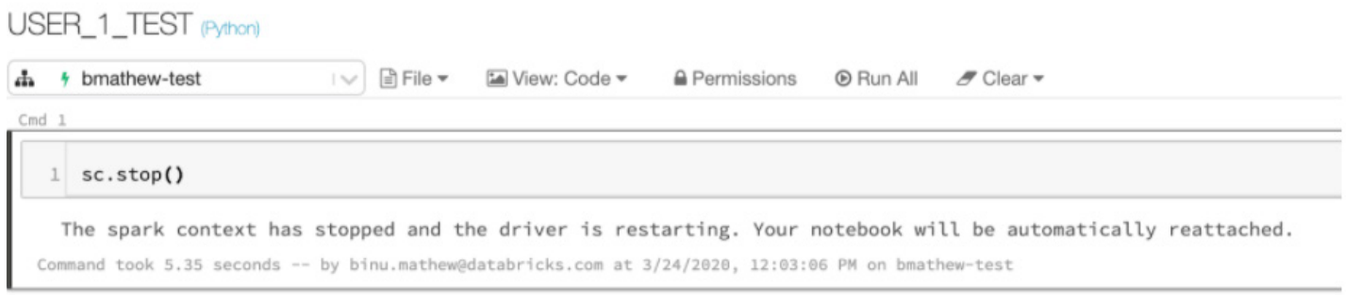
```
Cmd 1

1  %python
2  mySparkContext = SparkContext.getOrCreate()
3  mySparkSession = SparkSession.builder.getOrCreate()
```

예시로 돌아가자면, 코드를 이렇게 수정하면 되겠습니다.

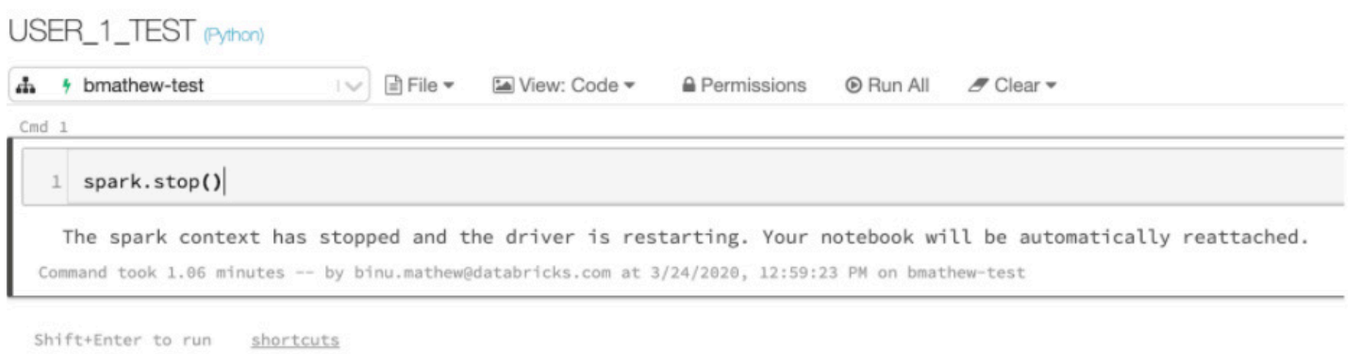
```
Cmd 2

1  from pyspark import SparkConf, SparkContext
2
3  conf = (SparkConf()
4          .set("spark.executor.memory","2g"))
5  sc = SparkContext.getOrCreate(conf = conf)

Command took 0.04 seconds -- by binu.mathew@databricks.com at 3/31/2020, 12:50:41 PM on bmathew-test
```

Databricks는 클러스터에 공유 SparkContext를 생성하므로 SparkContext를 종료해서는 안 됩니다. 동일한 클러스터에서 작업을 실행하는 다른 사용자에게 영향을 미칠 수 있기 때문입니다. 동일한 클러스터를 사용하는 실무자 2명의 예시를 들어보겠습니다.

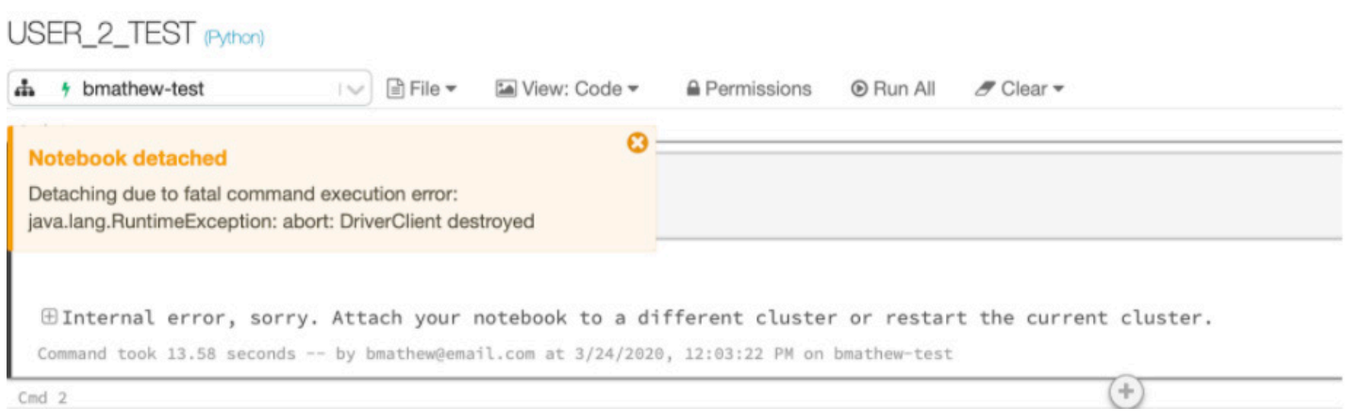
사용자 1이 다음의 명령을 입력하여 SparkContext를 종료합니다.



또는



사용자 2가 작업을 실행하려고 하지만 SparkContext가 중단되어 오류 메시지를 받습니다.



작업은 정상적으로 실행되지만 위의 오류가 반환되며 실패합니다.

자세한 내용은 다음의 문서를 참조하세요.

코드에서 SparkContext를 생성하는 방법의 예시를 살펴보겠습니다.

예시 1: 하둡에서 Databricks로 Spark RDD 코드 마이그레이션

예시 2: 하둡에서 Databricks로 Spark DataFrame 코드 마이그레이션

예시 1: 하둡에서 Databricks로 Spark RDD 코드 마이그레이션

Python 스크립트로 RDD API를 사용하는 기존 하둡 PySpark 코드는 Databricks에서 실행해야 합니다. 다음은 RDD API를 사용하여 Python 스크립트로 작성한 하둡 PySpark 코드로 데이터를 처리하는 예시입니다. 코드 샘플을 보면 SparkContext가 생성된 것을 알 수 있습니다. 이 작업은 지금도 사용자가 코드에서 하고 있는 작업일지 모릅니다. 이 예시에서는 Azure Blob 스토리지 계정에 액세스하여 자격 증명을 설정해야 합니다. 이렇게 하는 데는 여러 가지 방법이 있지만, 여기서는 자격 증명을 코드에서 설정하겠습니다. 이것은 개발자 자격으로 테스트할 때 사용 가능한 방법입니다. AWS와 Azure에서 동일한 코드를 살펴보겠습니다.

AWS:

```
from pyspark import SparkConf, SparkContext

## function to remove non-ascii characters from comment field
def remove_non_ascii(line):
    user_id = line[0]
    comment = line[1]
    return int(user_id),"".join(i for i in comment if ord(i)<128)

conf = (SparkConf()
         .set("spark.executor.memory","2g")
         .set("spark.hadoop.fs.s3a.access.key","<YOUR_ACCESS_KEY>")
         .set("spark.hadoop.fs.s3a.secret.key","<YOUR_SECRET_KEY>")
         .set("spark.hadoop.fs.s3a.impl","org.apache.hadoop.fs.s3a.S3AFileSystem"))
sc = SparkContext(conf = conf)

## read source file
lines = sc.textFile("s3a://<S3_BUCKET_NAME>/file.dat").map(lambda line: line.split('|'))

## remove first record which is header record
data_no_header = lines.zipWithIndex().filter(lambda row_index: row_index[1] > 0).keys()

## call function to remove non-ascii characters from comment field
output = data_no_header.map(lambda line: remove_non_ascii(line))

## save
output.saveAsTextFile("s3a://<S3_BUCKET_NAME>/output/")
```

Azure:

```
from pyspark import SparkConf, SparkContext

## function to remove non-ascii characters from comment field
def remove_non_ascii(line):
    user_id = line[0]
    comment = line[1]
    return int(user_id),"".join(i for i in comment if ord(i)<128)

## create Spark Context
conf = (SparkConf()
         .setAppName("Testing PySpark code")
         .set("spark.executor.memory","2g")
         .set("spark.hadoop.fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_ACCESS_KEY>"))
sc = SparkContext(conf = conf)

## read source file
lines = sc.textFile("wasbs://sample@bmathew.blob.core.windows.net/file.dat").map(lambda line: line.split('|'))

## remove first record which is header record
data_no_header = lines.zipWithIndex().filter(lambda row_index: row_index[1] > 0).keys()

## call function to remove non-ascii characters from comment field
output = data_no_header.map(lambda line: remove_non_ascii(line))

## save
output.saveAsTextFile("wasbs://sample@bmathew.blob.core.windows.net/output/")
```

이 코드를 Databricks에서 실행하려면 코드 실행 방식에 따라 변경 사항을 달리 적용해야 합니다.

- 기존 클러스터에서 Databricks 작업으로 실행
- 새 클러스터에서 Databricks 작업으로 실행

기존 클러스터에서 Databricks 작업으로 실행

이 Python 스크립트를 spark-submit을 사용해 Databricks 작업으로 기존 Databricks 클러스터에서 실행하려면, Python 스크립트를 수정하고 Databricks가 생성하는 기존 SparkContext(즉 SparkContext.getOrCreate)를 사용해야 합니다. 또한, 애플리케이션 이름을 설정할 때 YARN에서 Spark로 하둡을 실행하는 것과 같은 방식으로 설정하면 안 됩니다. 애플리케이션 이름을 설정해도 아무런 영향이 없습니다.

AWS와 Azure 기반에서 모두 Databricks에 실행할 수 있도록 동일한 코드를 수정한 실제 예시는 다음과 같습니다. Java와 Scala 코드도 똑같이 변경하면 됩니다. 코드 나머지 부분은 각종 Spark API를 사용할 때와 똑같이 그대로 유지됩니다.

AWS:

```
from pyspark import SparkConf, SparkContext

## function to remove non-ascii characters from comment field
def remove_non_ascii(line):
    user_id = line[0]
    comment = line[1]
    return int(user_id),"".join(i for i in comment if ord(i)<128)

conf = (SparkConf()
         .set("spark.executor.memory","2g")
         .set("spark.hadoop.fs.s3a.access.key","<YOUR_ACCESS_KEY>")
         .set("spark.hadoop.fs.s3a.secret.key","<YOUR_SECRET_KEY>")
         .set("spark.hadoop.fs.s3a.impl","org.apache.hadoop.fs.s3a.S3AFileSystem"))
sc = SparkContext(conf = conf)

## read source file
lines = sc.textFile("s3a://<S3_BUCKET_NAME>/file.dat").map(lambda line: line.split('|'))

## remove first record which is header record
data_no_header = lines.zipWithIndex().filter(lambda row_index: row_index[1] > 0).keys()

## call function to remove non-ascii characters from comment field
output = data_no_header.map(lambda line: remove_non_ascii(line))

## save
output.saveAsTextFile("s3a://<S3_BUCKET_NAME>/output/")
```

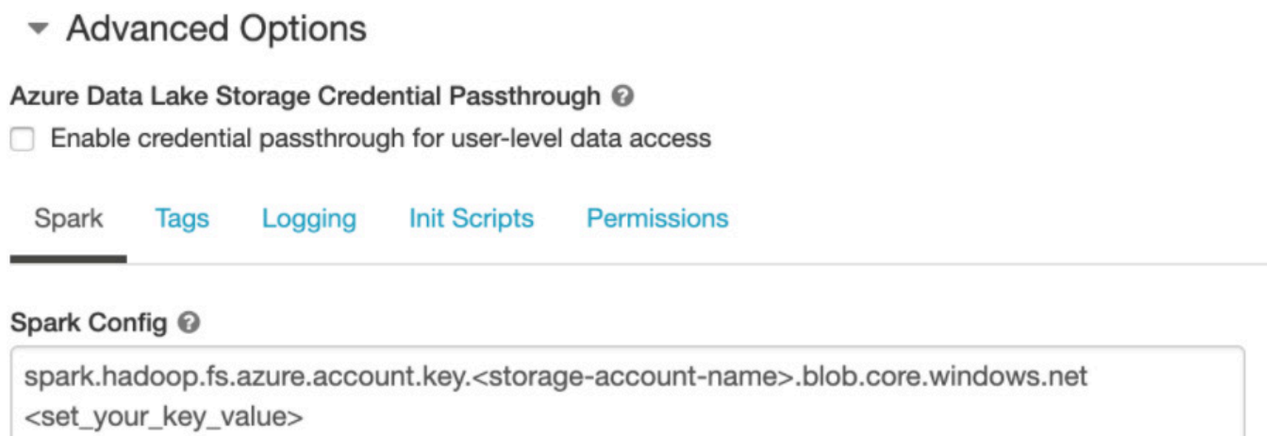
키를 사용하지 않고 Databricks에서 S3 데이터 소스에 액세스하는 방법은 여러 가지가 있습니다. 자세한 정보는 "데이터 소스" 섹션을 참조하세요. 앞의 예시에서는 키를 입력했습니다. Secrets API를 사용하여 실제 키 값이 표시되지 않도록 할 수 있습니다. Secret API에 대한 자세한 내용은 온라인 문서를 참조하세요.

S3 스토리지 액세스에 대한 자세한 내용은 온라인 문서를 참조하세요.

Azure:

```
from pyspark import SparkConf, SparkContext

## function to remove non-ascii characters from comment field
def remove_non_ascii(line):
    user_id = line[0]
    comment = line[1]
    return int(user_id),"".join(i for i in comment if ord(i)<128)

## use the existing Spark context
conf = (SparkConf()
        .set("spark.executor.memory","2g")
        .set("spark.hadoop.fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_ACCESS_KEY>"))
sc = SparkContext.getOrCreate(conf = conf)

## read source file
lines = sc.textFile("wasbs://sample@bmathew.blob.core.windows.net/file.dat").map(lambda line: line.split('|'))

## remove first record which is header record
data_no_header = lines.zipWithIndex().filter(lambda row_index: row_index[1] > 0).keys()

## call function to remove non-ascii characters from comment field
output = data_no_header.map(lambda line: remove_non_ascii(line))

## save
output.saveAsTextFile("wasbs://sample@bmathew.blob.core.windows.net/output/")
```

Databricks 클러스터에서 RDD API를 사용하여 spark-submit 작업으로 Java/Scala JAR 또는 Python 스크립트를 실행할 경우 앞의 예시와 같이 코드에서 자격 증명을 설정하면 됩니다(현재 하둡에서 사용하는 방식과 유사). 그러나 이는 spark-submit을 사용하여 작업을 실행할 경우에만 가능합니다. Spark Python 태스크를 사용하여 Python RDD 코드를 Databricks의 작업으로 제출하거나, 노트북 코드를 실행할 경우에는 코드에서 자격 증명을 설정할 수 없습니다. 이 사용 사례의 경우, 예시처럼 Azure 스토리지 자격 증명을 Databricks 클러스터의 Spark Config 설정으로 정의해야 합니다.



위의 예시에서는 키를 입력했습니다. Secrets API를 사용하여 실제 키 값이 표시되지 않도록 할 수 있습니다. Secret API Azure에 대한 자세한 내용은 온라인 문서를 참조하세요.

Python과 RDD API를 사용하여 노트북에서 직접 스토리지 자격 증명을 설정하려 하면 어떻게 되는지 알아보겠습니다. 설정이 무시되고, 다음과 같은 오류가 발생합니다.

스토리지 자격 증명을 설정하는 노트북 코드:

```
## we will use the existing Spark context
## use existing Spark Context
conf = (SparkConf()
         .setAppName("Testing PySpark code")
         .set("spark.executor.memory","2g")
         .set("spark.hadoop.fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_ACCESS_KEY>"))
sc = SparkContext.getOrCreate(conf = conf)
```

수신된 오류:

```
shaded.databricks.org.apache.hadoop.fs.azure.AzureException: shaded.databricks.org.apache.hadoop.fs.azure.AzureException: Container sample in account bmathew.blob.core.windows.net not found, and we can&#39;t create it using anoynomous credentials, and no credentials found for them in the configuration.
Command took 0.87 seconds -- by binu.mathew@databricks.com at 3/22/2020, 11:32:00 AM on bmathew-test
```

DBFS로 업로드된 Python RDD 코드를 실행하고 이를 Spark Python 태스크로 제출하려고 해도 동일한 오류가 발생합니다. DataFrames API와 Python을 사용할 경우, 코드와 노트북에서 모두 자격 증명을 설정할 수 있습니다. 다음의 예시에 그 방법이 나와 있습니다.

RDD API를 사용해 Databricks에서 Scala를 사용하는 경우, 스토리지 계정 자격 증명을 코드에서도 설정할 수 있고 노트북에서도 설정할 수 있습니다. 예를 들어 다음과 같습니다.

```
// Using an account access key
spark.sparkContext.hadoopConfiguration.set(
  "fs.azure.account.key.<storage-account-name>.blob.core.windows.net",
  "<storage-account-access-key>"
)
```

RDD API를 사용한 클라우드 스토리지 액세스 방법에 대한 자세한 내용은 온라인 문서를 참조하세요.

새 클러스터에서 Databricks 작업으로 실행

하둡 코드를 새 Databricks 클러스터에서 실행하되 이 클러스터는 이 작업에만 딱 한 번 쓰이고 종료되며, Databricks에서 spark-submit을 사용할 경우, SparkContext와의 상호작용 방법을 변경하고 새 SparkContext를 생성할 필요가 없습니다. 다시 말해 새 SparkContext를 생성한 원본 코드가 통한다는 말입니다. 단, Databricks에서 spark-submit을 사용하는 경우에 한합니다. Databricks에서 spark-submit을 사용하지 않는다면 기존 SparkContext를 사용하도록 코드를 수정해야 합니다. 예를 들어 이 코드를 Spark Python 태스크(spark-submit이 아니라)를 사용해 작업으로 실행하는 경우, 기존 SparkContext를 사용하도록 코드를 수정해야 합니다.

AWS에서는 코드를 다음과 같이 변경할 수 있고, Azure에서도 유사하게 변경합니다.

```
from pyspark import SparkConf, SparkContext

## function to remove non-ascii characters from comment field
def remove_non_ascii(line):
    user_id = line[0]
    comment = line[1]
    return int(user_id),"".join(i for i in comment if ord(i)<128)

conf = (SparkConf()
        .set("spark.executor.memory","2g")
        .set("spark.hadoop.fs.s3a.access.key","<YOUR_ACCESS_KEY>")
        .set("spark.hadoop.fs.s3a.secret.key","<YOUR_SECRET_KEY>")
        .set("spark.hadoop.fs.s3a.impl","org.apache.hadoop.fs.s3a.S3AFileSystem"))
sc = SparkContext(conf = conf)

## read source file
lines = sc.textFile("s3a://<S3_BUCKET_NAME>/file.dat").map(lambda line: line.split('|'))

## remove first record which is header record
data_no_header = lines.zipWithIndex().filter(lambda row_index: row_index[1] > 0).keys()

## call function to remove non-ascii characters from comment field
output = data_no_header.map(lambda line: remove_non_ascii(line))

## save
output.saveAsTextFile("s3a://<S3_BUCKET_NAME>/output/")
```

예시 2: 하둡에서 Databricks로 Spark DataFrame 코드 마이그레이션

Python 스크립트에서 DataFrame API를 사용하여 작성한 기존 하둡 PySpark 코드는 Databricks에서 실행해야 합니다. 다음은 Python 스크립트에서 DataFrame API를 사용하여 데이터를 처리하는 하둡 PySpark의 실제 예시입니다.

SparkSession 사용

다음의 코드 샘플과 유사한 하둡 코드에서 이미 SparkSession을 사용하고 있다면 코드를 변경할 필요가 없을 수도 있습니다. SparkSession은 DataFrame과 Data Set API를 사용하여 Spark와 상호작용할 수 있는 단일 진입점입니다. SparkSession을 사용할 경우, 명시적으로 SparkConf, SparkContext 또는 SQLContext를 생성할 필요가 없습니다. 모두 SparkSession에 캡슐화되기 때문입니다. AWS와 Azure에서 사용하는 코드의 예시 두 가지는 다음과 같습니다.

AWS:

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *

spark = (SparkSession
        .builder
        .appName("Testing PySpark code")
        .config("spark.executor.memory","2g")
        .config("spark.hadoop.fs.s3a.access.key", "<YOUR_ACCESS_KEY>")
        .config("spark.hadoop.fs.s3a.secret.key","<YOUR_SECRET_KEY>")
        .config("spark.hadoop.fs.s3a.impl","org.apache.hadoop.fs.s3a.S3AFileSystem")
        .getOrCreate())

csvSchema = StructType([
  StructField("product_id", LongType(), True),
  StructField("category", StringType(), True),
  StructField("brand", StringType(), True),
  StructField("model", StringType(), True),
  StructField("price", DoubleType(), True),
  StructField("processor", StringType(), True),
  StructField("size", StringType(), True),
  StructField("display", StringType(), True)
 ])

## create DataFrame
df = spark.read.option("header", "true").schema(csvSchema).csv("s3a://<S3_BUCKET_NAME>/product.csv")

# write the dataframe as a parquet file
df.write.mode("overwrite").parquet("s3a://<S3_BUCKET_NAME>/output/")
```

하둡 코드가 여기에 표시한 것과 같이 SpakrSession을 생성하는 경우, S3 자격 증명 설정 방식을 바꿔야 합니다. Java와 Scala 코드도 마찬가지입니다. 다만 Java 코드는 노트북에서 지원되지 않으므로 Databricks에서 작업으로 실행하기 위해 JAR 파일로 제출해야 합니다. 키를 사용하지 않고 Databricks에서 S3 데이터 소스에 액세스하는 방법은 여러 가지가 있습니다. 자세한 정보는 "데이터 소스" 섹션을 참조하세요. 다시 언급하지만, 사용자가 설정한 애플리케이션 이름은 Databricks에서 무시합니다. 코드 나머지 부분은 각종 Spark API를 사용할 때와 똑같이 그대로 유지됩니다.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *

spark = (SparkSession
        .builder
        .config("spark.executor.memory","2g")
        .getOrCreate())

spark._jsc.hadoopConfiguration().set("fs.s3n.awsAccessKeyId", "<YOUR_ACCESS_KEY>")
spark._jsc.hadoopConfiguration().set("fs.s3n.awsSecretAccessKey", "<YOUR_SECRET_KEY>")

csvSchema = StructType([
  StructField("product_id", LongType(), True),
  StructField("category", StringType(), True),
  StructField("brand", StringType(), True),
  StructField("model", StringType(), True),
  StructField("price", DoubleType(), True),
  StructField("processor", StringType(), True),
  StructField("size", StringType(), True),
  StructField("display", StringType(), True)
 ])

## create DataFrame
df = spark.read.option("header", "true").schema(csvSchema).csv("s3a://<S3_BUCKET_NAME>/product.csv")

# write the dataframe as a parquet file
df.write.mode("overwrite").parquet("s3a://<S3_BUCKET_NAME>/output/")
```

Azure:

```
from pyspark.sql.types import *
from pyspark.sql import SparkSession

spark = (SparkSession
         .builder
         .appName('Testing PySpark code')
         .config("spark.executor.memory","2g")
         .config("fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_ACCESS_KEY>")
         .getOrCreate())

csvSchema = StructType([
  StructField("product_id", LongType(), True),
  StructField("category", StringType(), True),
  StructField("brand", StringType(), True),
  StructField("model", StringType(), True),
  StructField("price", DoubleType(), True),
  StructField("processor", StringType(), True),
  StructField("size", StringType(), True),
  StructField("display", StringType(), True)
 ])

## create DataFrame from data in Blob storage using schema defined above
df = spark.read.option("header", "true").schema(csvSchema).csv("wasbs://initech@bmathew.blob.core.windows.net/product.csv")

# write the dataframe as a parquet file to Azure Blob Storage
df.write.mode("overwrite").parquet("wasbs://initech@bmathew.blob.core.windows.net/output/")
```

하둡 코드에서 이와 같이 SparkSession을 생성할 경우, 이 코드는 Databricks와 동일하게 실행됩니다. 즉, Python 스크립트와 Databricks 노트북에서 실행되는 코드로 실행됩니다. Java와 Scala 코드도 마찬가지입니다. Java 코드는 노트북에서 지원되지 않으므로 Databricks에서 작업으로 실행하기 위해 JAR 파일로 제출해야 합니다. 여기서 지적할 점은 Databricks에서는 애플리케이션 이름 설정을 지원하지 않으므로 설정해도 아무 영향이 없다는 것 한 가지뿐입니다.

하둡 Spark 코드에서 SparkContext를 생성하는 예시를 살펴보겠습니다.

SparkContext 사용

SparkContext를 명시적으로 생성하는 하둡 PySpark 코드의 실제 예시는 다음과 같습니다. 다음 예시는 Azure에서 실행했지만 다른 클라우드에도 이 내용이 그대로 해당합니다.

```
from pyspark.sql.types import *
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext

## Use existing SparkContext
conf = (SparkConf()
        .setAppName("Testing PySpark code")
        .set("spark.executor.memory","2g")
        .set("fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_ACCESS_KEY>"))
sc = SparkContext(conf = conf)
sqlContext = SQLContext(sc)

csvSchema = StructType([
  StructField("product_id", LongType(), True),
  StructField("category", StringType(), True),
  StructField("brand", StringType(), True),
  StructField("model", StringType(), True),
  StructField("price", DoubleType(), True),
  StructField("processor", StringType(), True),
  StructField("size", StringType(), True),
  StructField("display", StringType(), True)
 ])

## create DataFrame from data in Blob storage using schema defined above
df = sqlContext.read.option("header", "true").schema(csvSchema).csv("wasbs://initech@bmathew.blob.core.windows.net/product.csv")

# write the dataframe as a parquet file to Azure Blob Storage
df.write.mode("overwrite").parquet("wasbs://initech@bmathew.blob.core.windows.net/output/")
```

이 코드를 Databricks에서 실행하려면 코드 실행 방식에 따라 변경 사항을 달리 적용해야 합니다.

- 기존 클러스터에서 Databricks 작업으로 실행
- 새 클러스터에서 Databricks 작업으로 실행

기존 클러스터에서 Databricks 작업으로 실행

이 코드를 기존 Databricks 클러스터에서 작업으로 실행하려면 기존 SparkContext(SparkContext.getOrCreate)를 수정해서 사용해야 합니다. 애플리케이션 이름을 설정할 수도 있지만, Databricks에는 아무런 영향을 미치지 않고 Spark 기록 서버 UI에도 기록되지 않습니다.

```
from pyspark.sql.types import *
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext

## Use existing SparkContext
conf = (SparkConf()
         .set("spark.executor.memory","2g")
         .set("fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_ACCESS_KEY>"))
sc = SparkContext.getOrCreate(conf = conf)
sqlContext = SQLContext(sc)

csvSchema = StructType([
  StructField("product_id", LongType(), True),
  StructField("category", StringType(), True),
  StructField("brand", StringType(), True),
  StructField("model", StringType(), True),
  StructField("price", DoubleType(), True),
  StructField("processor", StringType(), True),
  StructField("size", StringType(), True),
  StructField("display", StringType(), True)
 ])

## create DataFrame from data in Blob storage using schema defined above
df = sqlContext.read.option("header", "true").schema(csvSchema).csv("wasbs://initech@bmathew.blob.core.windows.net/product.csv")

# write the dataframe as a parquet file to Azure Blob Storage
df.write.mode("overwrite").parquet("wasbs://initech@bmathew.blob.core.windows.net/output/")
```

이 변경 사항을 적용하고 나면 이 코드를 Python 스크립트로 실행할 수도 있고, Databricks 노트북 셀에서 직접 실행할 수도 있습니다. Java와 Scala 코드에도 똑같은 변경 사항을 적용해야 합니다. 코드 나머지 부분은 각종 Spark API를 사용할 때와 똑같이 그대로 유지됩니다.

소스 파일에서 읽는 작업에 DataFrame API를 사용하고 있으므로, SparkSession만 사용할 수 있습니다. 이것은 SparkConf, SparkContext 및 SQLContext도 캡슐화하기 때문입니다. 이 방식이 현재 사용되는 Spark 프로그래밍 방식과 더 일맥상통합니다.

```
from pyspark.sql.types import *
from pyspark.sql import SparkSession

spark = (SparkSession
         .builder
         .config("spark.executor.memory","2g")
         .config("fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_ACCESS_KEY>")
         .getOrCreate())

csvSchema = StructType([
  StructField("product_id", LongType(), True),
  StructField("category", StringType(), True),
  StructField("brand", StringType(), True),
  StructField("model", StringType(), True),
  StructField("price", DoubleType(), True),
  StructField("processor", StringType(), True),
  StructField("size", StringType(), True),
  StructField("display", StringType(), True)
 ])

## create DataFrame from data in Blob storage using schema defined above
df = spark.read.option("header", "true").schema(csvSchema).csv("wasbs://initech@bmathew.blob.core.windows.net/product.csv")

# write the dataframe as a parquet file to Azure Blob Storage
df.write.mode("overwrite").parquet("wasbs://initech@bmathew.blob.core.windows.net/output/")
```

새 클러스터에서 Databricks 작업으로 실행

하둡 코드를 새 Databricks 클러스터에서 실행하되 이 클러스터는 이 작업에만 딱 한 번 쓰이고 종료되며, Databricks에서 spark-submit을 사용할 경우, SparkContext와의 상호작용 방법을 변경하고 새 SparkContext를 생성할 필요가 없습니다. 새 SparkContext를 생성하는 원래의 코드는 Databricks에서 spark-submit을 사용하여 작업으로 제출할 경우에만 작동합니다.다시 말해 새 SparkContext를 생성한 원본 코드가 통한다는 말입니다. 단, Databricks에서 작업으로 제출하기 위해 spark-submit을 사용하는 경우에 한합니다. Databricks에서 spark-submit을 사용하지 않는다면 기존 SparkContext를 사용하도록 코드를 수정해야 합니다. 예를 들어 이 코드를 Spark Python 태스크(spark-submit이 아니라)를 사용해 작업으로 실행하는 경우, 기존 SparkContext를 사용하도록 코드를 수정해야 합니다.

이 코드를 작성할 때는 SparkSession을 사용하는 것이 낫습니다. SparkSession은 SparkConf, SparkContext와 SQLContext를 캡슐화합니다. SparkSession을 사용하는 방이 현재 사용되는 Spark 프로그래밍 관행과 더 일맥상통합니다. 코드는 다음과 같이 단순화할 수 있습니다.

```
from pyspark.sql.types import *
from pyspark.sql import SparkSession

spark = (SparkSession
         .builder
         .config("spark.executor.memory","2g")
         .config("fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_ACCESS_KEY>")
         .getOrCreate())

csvSchema = StructType([
  StructField("product_id", LongType(), True),
  StructField("category", StringType(), True),
  StructField("brand", StringType(), True),
  StructField("model", StringType(), True),
  StructField("price", DoubleType(), True),
  StructField("processor", StringType(), True),
  StructField("size", StringType(), True),
  StructField("display", StringType(), True)
 ])

## create DataFrame from data in Blob storage using schema defined above
df = spark.read.option("header", "true").schema(csvSchema).csv("wasbs://initech@bmathew.blob.core.windows.net/product.csv")

# write the dataframe as a parquet file to Azure Blob Storage
df.write.mode("overwrite").parquet("wasbs://initech@bmathew.blob.core.windows.net/output/")
```

챕터 3: 애플리케이션 개발, 테스트 및 배포

코드 개발을 위한 노트북 및 IDE

Apache Zeppelin은 하둡에서 코드 개발과 테스트, 데이터 쿼리에 사용하는 보편적인 노트북 개발 환경입니다. Databricks 노트북도 유사하지만 제공하는 기능이 더 많습니다.

데이터 액세스

사용 가능한 데이터 세트에 빠르게 액세스하거나 각종 데이터 소스에 연결합니다(온프레미스, 클라우드 내 불문).

다언어 지원

같은 노트북에서 여러 프로그래밍 언어를 지원하므로(R, Python, Scala 및 SQL 등) 인터랙티브 노트북을 사용해 데이터를 탐색할 수 있습니다.

자동 버전 관리

변경 사항과 버전을 자동으로 추적하여 중간에 그만둔 지점에서 작업을 계속하거나 변경 사항을 되돌립니다.

실시간 공동 작성

같은 노트북에서 실시간으로 작업하고 변경 사항은 상세한 수정 기록으로 추적합니다.

대시보드 및 시각화

자세한 대시보드를 생성하고 포인트-클릭 시각화를 사용하여 인사이트를 시각화하거나 matplotlib, ggplot, D3 등의 강력한 스크립트 작성 옵션을 사용합니다.

데이터 사이언스

노트북의 실험, 매개변수와 결과를 실행(run)으로 MLflow에 직접 자동 로깅하고, 사이드바에서 이전 실행과 코드 버전을 신속하게 확인하고 로드합니다.

노트북 워크플로우 및 작업 예약

다단계 파이프라인을 생성하고 특정 일정에 따라 프로덕션 파이프라인에 작업 형식으로 노트북을 실행합니다.

보안

각각의 노트북(또는 노트북 컬렉션)과 실험에 대한 액세스를 신속하게 관리하며, 여기에 단 하나의 공용 보안 모델을 적용합니다.

통합

Tableau, Looker, Power BI, RStudio, Snowflake 등에 연결되므로 데이터 사이언티스트와 엔지니어가 익숙한 도구를 사용할 수 있습니다.

자동 크기 조정 및 온디맨드 클러스터

노트북을 신속하게 연결하여 클러스터를 자동으로 관리하여 전례없이 큰 규모의 컴퓨팅을 능률적이고 비용 효율적인 방식으로 확장합니다.

Apache Zeppelin 노트북에서 작성하고 하둡에서 사용하는 코드는 대개 SparkContext를 생성할 필요가 없습니다. 이미 SparkContext가 생성되어 있기 때문입니다. 이는 Databricks 노트북도 마찬가지입니다. SparkContext 또는 SparkSession을 사용자가 명시적으로 인스턴스화할 필요가 없습니다. 이미 사용자를 대신해 완료해 놓았기 때문입니다. 따라서 Spark API 전용으로 Zeppelin 노트북에서 가져온 코드는 변경 없이 Azure Databricks에서 실행될 수도 있습니다.

Databricks에서 새 SparkContext를 생성하면 오류가 발생합니다.

Cmd 2

```
1  from pyspark import SparkConf, SparkContext
2
3  conf = (SparkConf()
4           .setAppName("Testing PySpark code")
5           .set("spark.executor.memory","2g")
6           .set("fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_AZURE_KEY>"))
7  sc = SparkContext(conf = conf)
```

```
ValueError: Cannot run multiple SparkContexts at once; existing SparkContext(app=Databricks Shell, master=spark://10.139.64.7:7077) created by __init__ at /local_disk0/tmp/1585189930999-0/PythonShell.py:1335
```

Command took 0.18 seconds -- by binu.mathew@databricks.com at 3/26/2020, 10:33:25 AM on bmathew-test

따라서 기존 SparkContext를 사용해야 합니다.

Cmd 2

```
1  from pyspark import SparkConf, SparkContext
2
3  ## Use existing SparkContext
4  conf = (SparkConf()
5           .setAppName("Testing PySpark code")
6           .set("spark.executor.memory","2g")
7           .set("fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_AZURE_KEY>"))
8  sc = SparkContext.getOrCreate(conf = conf)
```

Command took 0.15 seconds -- by binu.mathew@databricks.com at 3/26/2020, 10:31:30 AM on bmathew-test

Databricks 노트북 코드는 개발 과정에서 SparkSession을 이미 사용자 대신 생성해 놓고, SparkSession 자체가 SparkConf, SparkContext와 SQLContex를 캡슐화합니다. 예를 들어 다음 코드는 Databricks 노트북에서 작동하겠지만 꼭 그래야 하는 것은 아닙니다.

Cmd 4

```
from pyspark.sql import SparkSession

spark = (SparkSession
         .builder
         .config("spark.executor.memory","2g")
         .config("fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_ACCESS_K
         .getOrCreate())
```

Command took 0.03 seconds -- by binu.mathew@databricks.com at 3/26/2020, 10:43:53 AM on bmathew-test

대신 다음 작업만 해도 됩니다. SparkSession이 이미 인스턴스와되어 있기 때문입니다.

Cmd 5

```
spark.conf.set("spark.executor.memory","2g")
spark.conf.set("fs.azure.account.key.bmathew.blob.core.windows.net","<YOUR_ACCESS_KEY>")
```

Command took 0.22 seconds -- by binu.mathew@databricks.com at 3/26/2020, 11:27:50 AM on bmathew-test

Databricks는 다른 노트북과 IDE를 사용하여 플랫폼과 상호작용하는 옵션도 제공합니다. 예를 들어 Zeppelin, Jupyter, Visual Studio, PyCharm, IntelliJ, Eclipse, RStudio 등을 사용할 수 있습니다. 다른 노트북과 IDE 사용에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

전체 기능 목록과 Databricks 노트북 사용에 대한 자세한 내용은 온라인 문서를 참조하세요

AWS AZURE

챕터 3: 애플리케이션 개발, 테스트 및 배포

소스 코드 관리 및 CI/CD

Databricks 노트북에는 기본 버전 관리 기능이 내장되어 있습니다. 이 기능에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

노트북은 다음 소스 코드 관리(SCM) 시스템과도 연결할 수 있습니다.

GitHub

자세한 정보는 온라인 문서를 참조하세요.

AWS Azure

Bitbucket

자세한 정보는 온라인 문서를 참조하세요.

AWS Azure

GitLab

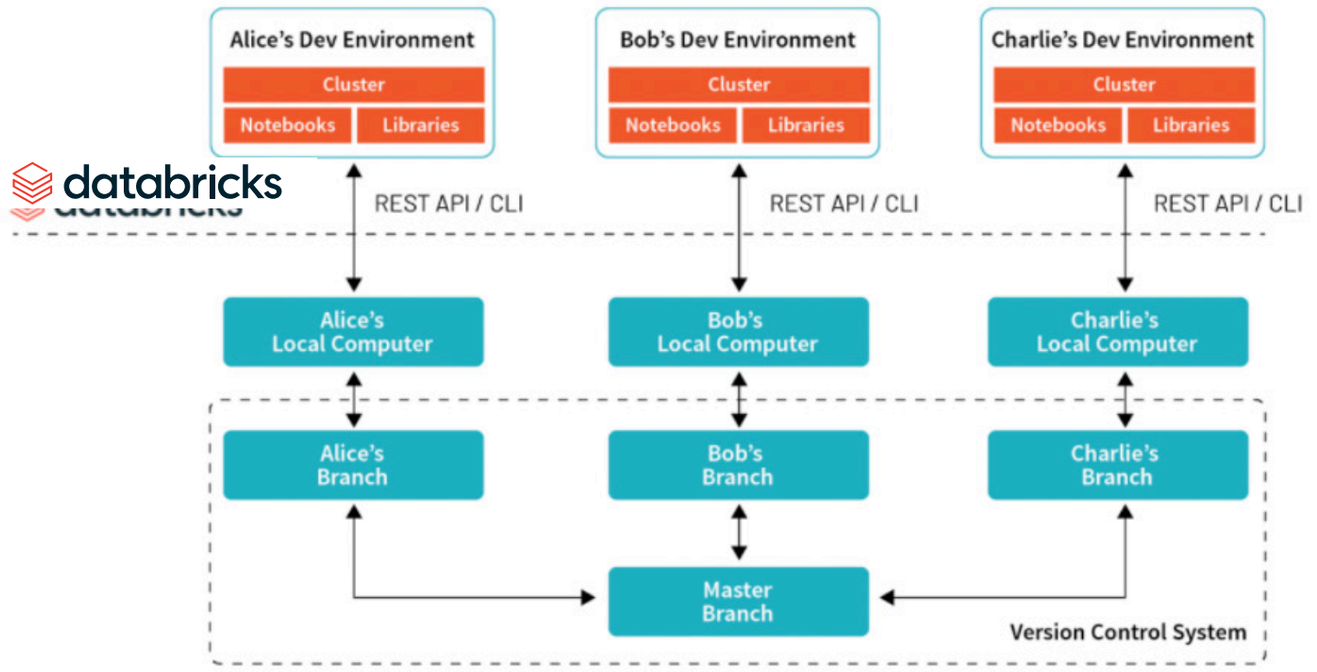
자세한 정보는 온라인 문서를 참조하세요.

AWS Azure

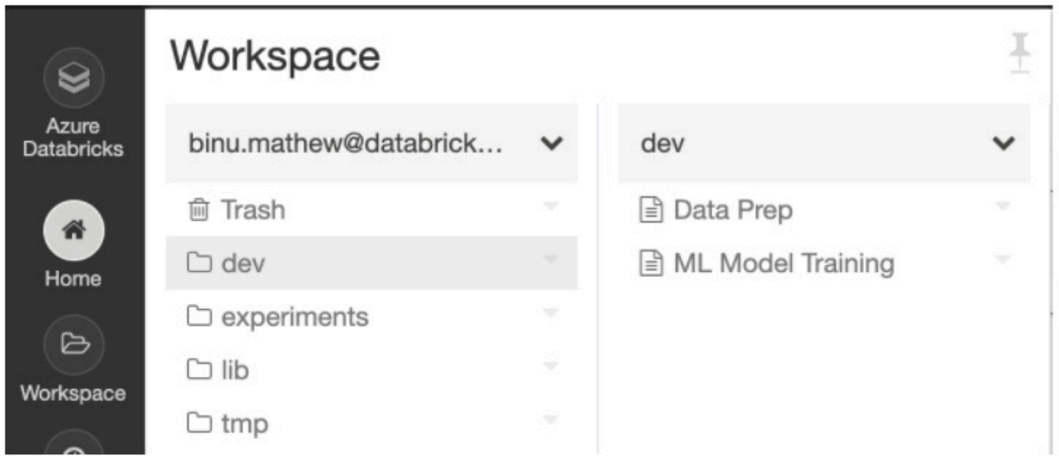
Azure DevOps

자세한 정보는 온라인 문서를 참조하세요.

또한 명령줄 인터페이스를 사용해도 노트북 코드와 비 노트북 코드(Java, Scala, Python 소스 코드)를 버전 관리 시스템과 동기화할 수 있습니다.



예를 들어봅시다. 어느 Databricks 사용자가 Python 스크립트와 노트북을 개발했는데, 이것을 현재 작업 중인 프로젝트용으로 회사 GitHub 리포지토리에 체크인해야 합니다. 여기서 노트북은 Databricks에서 개발했고, Python 스크립트는 휴대용 컴퓨터에서 로컬로 개발했습니다.




```
(base) C02WW0HYHTD5:dev bmathew$ ls -ltrh
total 0
-rw-r--r--  1 bmathew  staff     0B Mar 29 18:17 transform_and_parse_json.py
-rw-r--r--  1 bmathew  staff     0B Mar 29 18:18 extract_json_data.py
-rw-r--r--  1 bmathew  staff     0B Mar 29 18:18 load_json.py
(base) C02WW0HYHTD5:dev bmathew$
```

이 사용자는 휴대용 컴퓨터에서 로컬 분기를 작업 중인데, Python 스크립트와 노트북 코드를 체크인한 뒤 원격 SCM 서버로 푸시해야 합니다. 이 작업은 명령줄 인터페이스(CLI)와 SCM 도구 명령으로 쉽게 처리할 수 있습니다.

1. 사용자의 휴대용 컴퓨터에서, 노트북을 Databricks Workspace에서 내보내 로컬 휴대용 컴퓨터로 가져오겠습니다. 이 경우 Databricks CLI를 사용합니다.

```
databricks --profile AZURE_PROD workspace export /Users/binu.mathew@databricks.com/dev/”Data Prep” .
databricks --profile AZURE_PROD workspace export /Users/binu.mathew@databricks.com/dev/”ML Model Training” .
```

```
(base) C02WW0HYHTD5:recommendation_engine bmathew$ databricks --profile AZURE_PROD workspace export /Users/binu.mathew@databricks.com/dev/"Data Prep" .
(base) C02WW0HYHTD5:recommendation_engine bmathew$ databricks --profile AZURE_PROD workspace export /Users/binu.mathew@databricks.com/dev/"ML Model Training" .
(base) C02WW0HYHTD5:recommendation_engine bmathew$
```

로컬 휴대용 컴퓨터의 디렉터리를 보겠습니다. 노트북을 내보낸 것을 알 수 있습니다.

```
(base) C02WW0HYHTD5:recommendation_engine bmathew$ ls -ltrh
total 16
-rw-r--r--  1 bmathew  staff     0B Mar 29 18:24 transform_and_parse_json.py
-rw-r--r--  1 bmathew  staff     0B Mar 29 18:24 extract_json_data.py
-rw-r--r--  1 bmathew  staff     0B Mar 29 18:24 load_json.py
-rw-r--r--  1 bmathew  staff    29B Mar 29 18:26 Data Prep.py
-rw-r--r--  1 bmathew  staff    29B Mar 29 18:26 ML Model Training.py
(base) C02WW0HYHTD5:recommendation_engine bmathew$
```

2. 사용자 휴대용 컴퓨터에서 SCM 도구 명령을 사용해 로컬 분기를 생성합니다.

```
(base) C02WW0HYHTD5:recommendation_engine bmathew$ git checkout -b recommendation_engine
Switched to a new branch 'recommendation_engine'
(base) C02WW0HYHTD5:recommendation_engine bmathew$
```

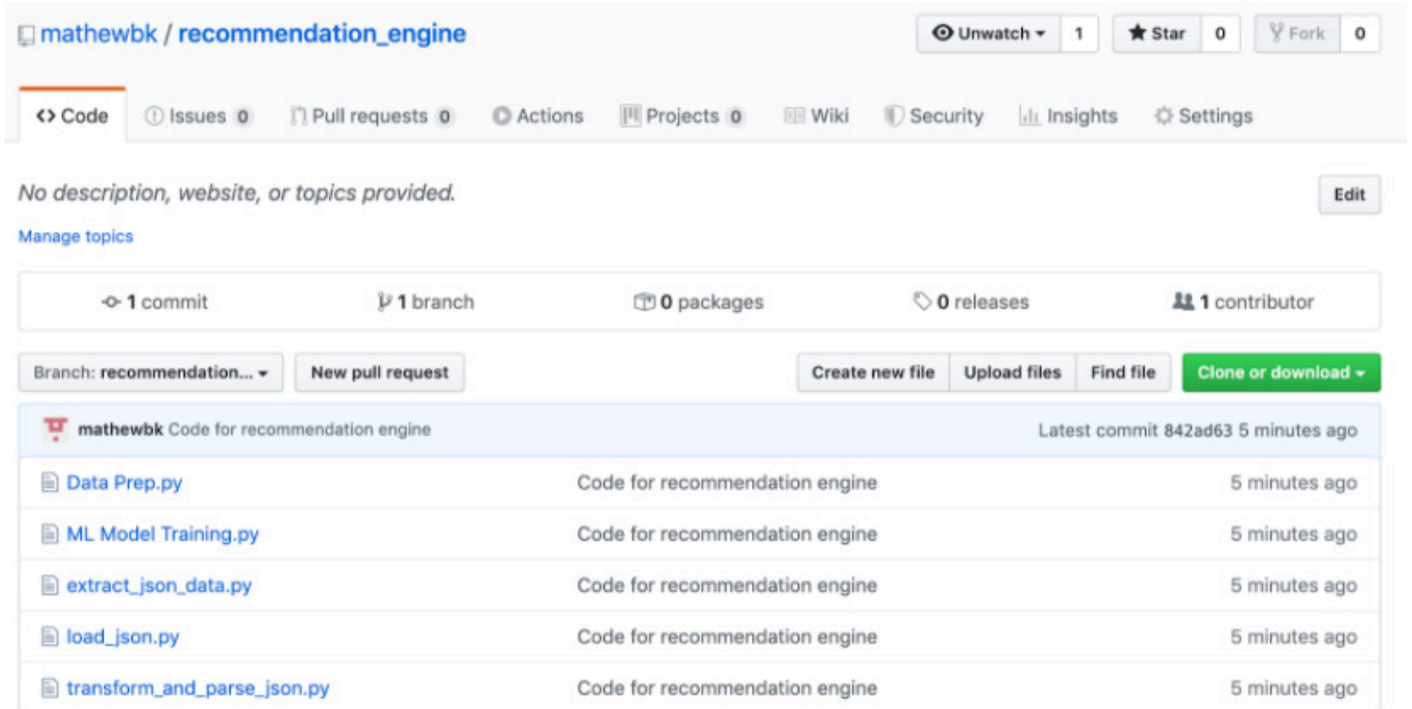
3. 이 로컬 분기에 모든 코드를 추가합니다. 여기에는 Python 스크립트와 노트북을 모두 포함합니다.

```
(base) C02WW0HYHTD5:recommendation_engine bmathew$ git add .
(base) C02WW0HYHTD5:recommendation_engine bmathew$ git commit -am "Code for recommendation engine"
[recommendation_engine (root-commit) 842ad63] Code for recommendation engine
 5 files changed, 2 insertions(+)
 create mode 100644 Data Prep.py
 create mode 100644 ML Model Training.py
 create mode 100644 extract_json_data.py
 create mode 100644 load_json.py
 create mode 100644 transform_and_parse_json.py
(base) C02WW0HYHTD5:recommendation_engine bmathew$
```

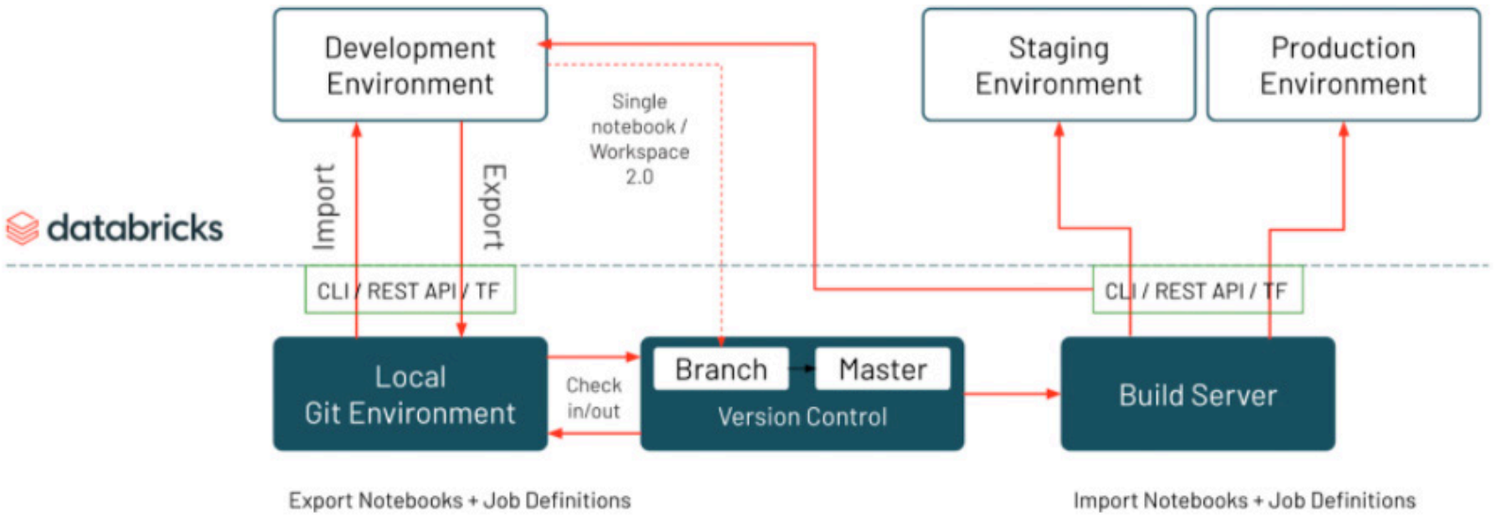
4. SCM 도구 명령을 사용해 로컬 분기를 푸시하여 SCM 서버를 제거합니다.

```
(base) C02WW0HYHTD5:recommendation_engine bmathew$ git remote  set-url origin git@github.com:mathewbk/recommendation_engine.git
(base) C02WW0HYHTD5:recommendation_engine bmathew$ git push origin recommendation_engine

Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 356 bytes | 356.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To github.com:mathewbk/recommendation_engine.git
 * [new branch]      recommendation_engine -> recommendation_engine
(base) C02WW0HYHTD5:recommendation_engine bmathew$
```

5. 로컬 분기가 SCM 서버로 푸시된 것을 확인할 수 있습니다.



코드를 소스 코드 관리 시스템에 체크인하고 나면 이를 빌드 서버로, 나아가 다양한 환경으로 보내 테스트와 최종 배포에 사용할 수 있습니다.



Databricks와 CI/CD 통합에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

챕터 3: 애플리케이션 개발, 테스트 및 배포

작업 예약 및 제출

Databricks를 사용하면 코드를 Java/Scala JAR 파일, Python 스크립트 및 wheel/egg 파일로 제출하거나, 노트북 코드로 제출하여 Databricks 클러스터에서 예약 작업이나 직접 실행 작업으로 실행할 수 있습니다. 플랫폼과 함께 기본 cron 스타일 작업 스케줄러가 제공되므로 이것을 사용해 작업을 예약하고 실행하면 됩니다. Databricks 작업 스케줄러는 UI, REST API 또는 명령줄 인터페이스를 통해 액세스할 수 있습니다. 타사 외부 작업 스케줄러에서 Databricks용 REST API 인터페이스를 사용하면 더욱 복잡한 워크플로우/DAG를 생성할 수 있습니다. 외부 작업 스케줄러에서 워크플로우를 생성하여 실행한 다음, 워크플로우의 각 태스크가 Databricks 클러스터(Java/Scala JAR 파일, Python 스크립트, wheel/egg 또는 노트북)에서 REST 호출을 통해 코드를 실행하도록 할 수 있습니다. 그런 다음, REST를 통해 반환 코드(종료 코드)를 가져와 워크플로우 진행 방식을 결정합니다(즉, 처리를 계속 진행할지 워크플로우 전체를 실패할지 결정).

Azure Data Factory(ADF)와 Apache Airflow는 작업을 예약하고 시작할 때 일반적으로 사용하는 도구입니다. Databricks는 ADF 및 Apache Airflow와 기본으로 통합되며, 작업 일정 관리를 지원합니다. 이 통합에 대한 자세한 내용은 온라인 문서를 참조하세요.

AWS AZURE

Databricks에서 작업 일정 관리 시 고려해야 할 중요한 사항:

- 워크스페이스 한 개당 UI에 표시되는 작업은 최대 1,000개입니다.
- 워크스페이스 하나가 한 시간에 생성할 수 있는 작업 수는 5,000개로 한정되어 있습니다("run now", "runs submit" 포함). 이 한도가 REST API와 노트북 워크플로우가 생성한 작업에도 영향을 미칩니다.
- 워크스페이스 한 개가 동시에 생성할 수 있는 실행 수는 150개로 제한됩니다.
- 위의 한도를 초과하는 경우, 워크스페이스를 여러 개 사용하면 됩니다.

Spark JAR 작업 vs. Spark Submit 작업

Databricks의 작업은 여러 가지 태스크 유형을 기반으로 합니다. 태스크 유형에 따라 실행할 코드 유형이 결정됩니다. 기존 spark-submit 구문도 JAR 파일에서 지원됩니다. 지원되는 태스크 유형은 다음과 같습니다.

- 노트북
- Spark JAR
- Spark Submit
- Python 스크립트

JAR 파일을 처리할 때는 Spark JAR 태스크 유형을 사용하는 것이 좋습니다. Spark Submit 태스크는 자동 크기 조정이나 Databricks 유틸리티 JAR 사용을 지원하지 않습니다. Databricks 유틸리티 JAR은 dbutils.* API를 사용하여 DBFS, 노트북, Secrets 등을 관리합니다.

기존 spark-submit 작업을 동급의 Spark JAR 버전으로 변환하고자 하는 경우, 다음과 같이 변경해야 합니다.

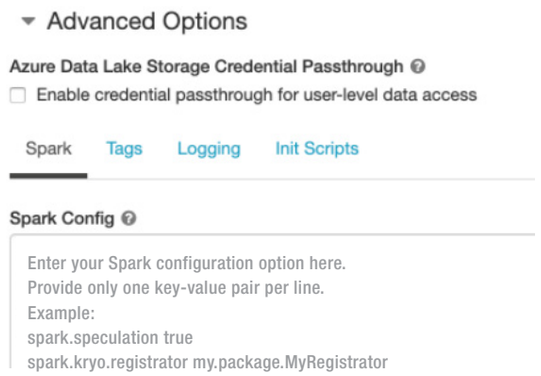
- 코드에 SparkContext.getOrCreate()를 사용하여 SparkContext를 가져옵니다.
- 매개변수가 JSON 형식의 태스크 매개변수를 통해 작업에 전달됩니다.
  AWS Azure
- —FILES를 통해 전달된 모든 파일은 클라우드 스토리지로 이동해야 합니다. 파일에 액세스하는 해당 코드는 클라우드 스토리지를 참조해야 합니다.
- 클러스터 구성에서 작업에 대한 숫자 실행자가 클러스터 구성의 Max Workers로 지정됩니다.
  AWS Azure
- 모든 Spark 구성 매개변수는 클러스터 구성에서 해당 작업에 대해 지정됩니다.

Spark Submit 작업

Spark를 친숙하게 사용하는 사용자라면 YARN을 통해 Spark 애플리케이션을 실행하는 데 이미 spark-submit을 사용하고 있을 가능성이 있습니다. Databricks에서는 현재 하둡에서와 같은 방식으로 spark-submit을 사용해 작업을 실행하도록 지원합니다.

Spark가 작동하는 방식은 YARN과 Databricks에서 몇 가지 다른 점이 있습니다.

- Databricks 클러스터의 기본 기능은 작업자 VM 한 개당 실행자를 하나 시작한 다음, 그 작업자 VM에 코어를 모두 사용하는 것입니다. RAM도 대부분 이 하나의 실행자에 쓰입니다. 단, OS와 여타 시스템 프로세스에 사용되는 RAM만은 예외입니다.
- VM에서 여러 개의 실행자를 시작하려면 Spark Config의 구성 설정에서 `spark.executor.cores`와 `spark.executor.memory`를 구성해야 합니다.



- 예를 들어 작업자 VM 두 개를 포함한 클러스터를 실행한다고 합시다. 각각의 VM에 61GB RAM과 코어 8개가 탑재되어 있습니다. 기본적으로 Databricks는 총 2개의 실행자를 시작합니다.

Executor ID	Address	Status
0	10.0.227.239:45117	Active
driver	10.0.250.76:44259	Active
1	10.0.243.105:36203	Active

- VM당 실행자를 두 개씩으로 하려면 Spark Config를 다음과 같이 구성하면 됩니다.

```
spark.executor.cores 4
spark.executor.memory 25g
```

- 이 예시에서 지적할 중요한 점은, 메모리를 30G(약 절반)로 설정해도 실행자가 여러 개 시작되지는 않았다는 사실입니다. 값을 낮춰 25G로 설정하자 효과가 있었습니다.
- 이제 작업자 VM 하나에 실행자 두 개가 설정되며, 각 실행자가 사용하는 IP는 같지만 포트는 다릅니다.

Executor ID	Address	Status
0	10.0.227.239:39565	Active
driver	10.0.250.76:44873	Active
1	10.0.227.239:41257	Active
2	10.0.243.105:38980	Active
3	10.0.243.105:44943	Active

YARN은 하둡 클러스터에서 리소스를 공유하므로 동일한 클러스터에 여러 프로덕션 작업을 제출할 수 있습니다. 하둡 클러스터는 상시 운영되는 클러스터인 반면, Databricks 클러스터는 단기 실행되고 자동 크기 조정됩니다. Databricks 클러스터는 작업 진행 내내 활성 상태로 유지되다가 작업이 끝나면 종료됩니다. Databricks의 경우, 각각의 프로덕션 작업을 자체적인 자동 크기 조정 클러스터에서 실행하는 편을 권장합니다. 작업이 완료되면 그 클러스터도 종료됩니다. 이 경우 분리 면에서 더 낫습니다. 즉 작업이 따로 실행되기 때문에 여러 작업이 리소스를 공유하며 경쟁하지 않아서 리소스 경합이 없고, 작업 완료를 더욱 확실히 보장할 수 있습니다.

자세한 정보는 온라인 문서를 참조하세요.

Databricks에서 작업 생성 및 제출

AWS Azure

하둡의 Apache Spark에서 Databricks로 프로덕션 워크로드 마이그레이션

AWS Azure

Spark 작업을 YARN에 제출하는 경우 spark-submit을 사용하면 됩니다. 현재 이 작업을 하고 있다면 같은 방식입니다. 예를 들어 다음은 Python 스크립트를 실행하기 위해 YARN에서 실행하는 spark-submit 명령입니다.

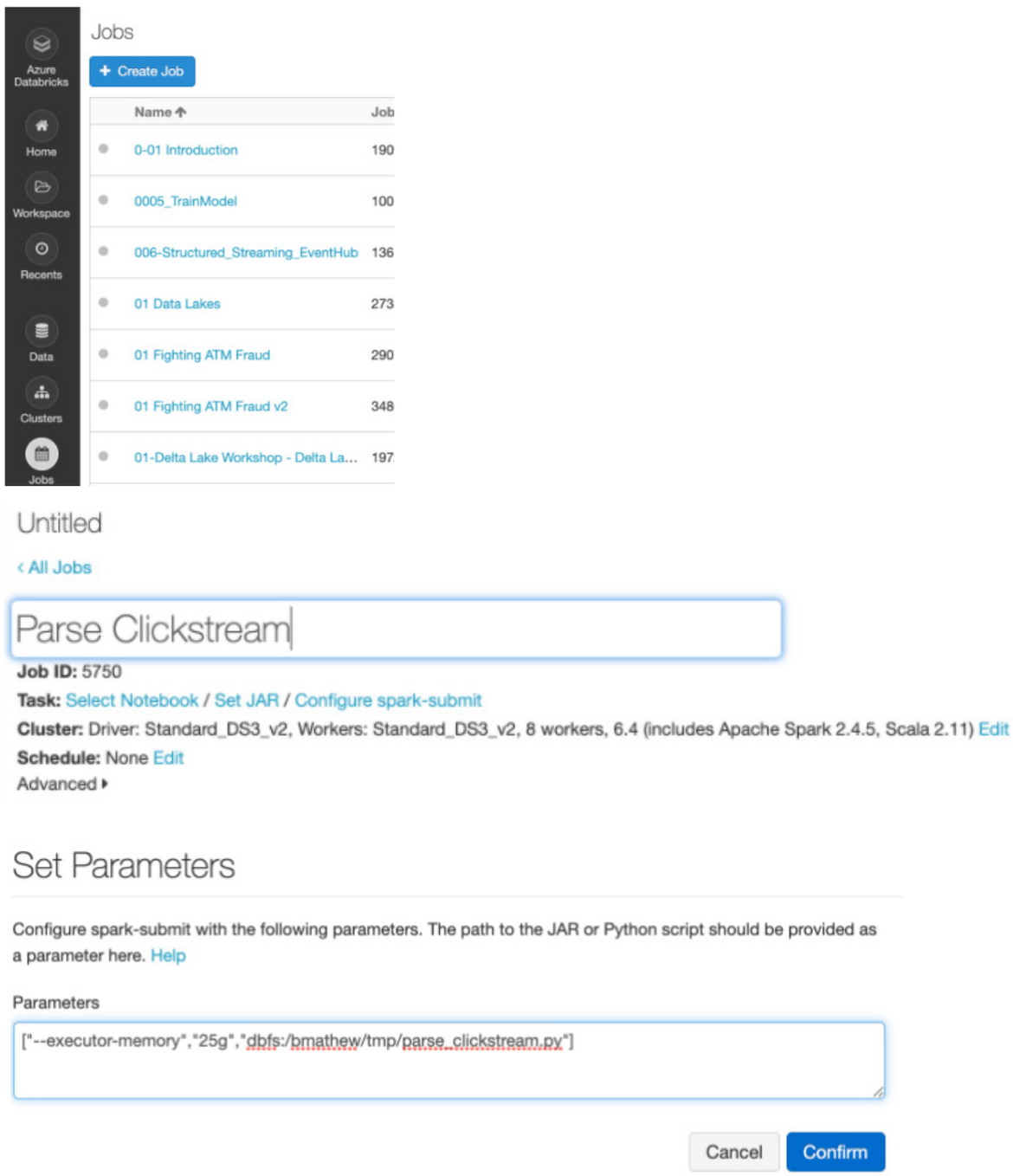
```
spark-submit --master=yarn --executor-memory=25g parse_clicksteam.py
```

Databricks에 작업을 제출할 때는 UI, REST API, 명령줄 인터페이스(CLI)를 사용합니다. 이제 예시를 살펴보겠습니다.

- UI를 사용한 작업 제출
- API 및 CLI를 사용한 작업 제출
- 기존 클러스터에 작업 제출

1. UI를 사용한 작업 제출

새 작업을 생성, spark-submit을 구성, 클러스터를 구성하고 작업을 예약하며 고급 속성을 설정합니다(필요한 경우).



2. API 및 CLI를 사용한 작업 제출

curl을 사용하여 REST API를 호출하고 작업을 생성할 수 있습니다.

```
curl -n -H "Content-Type: application/json" -X POST -d @- https://eastus2.azuredatabricks.net/api/2.0/jobs/create <<JSON
{
  "name": "Test Job Submission via API",
  "new_cluster": {
    "num_workers": 2,
    "spark_version": "6.4.x-scala2.11",
    "node_type_id": "Standard_D16_v3",
    "spark_env_vars": {
        "PYSPARK_PYTHON": "/databricks/python3/bin/python3"}
  },
  "spark_submit_task": {
        "parameters": [
            "--executor-memory",
            "25g",
            "dbfs:/bmathew/tmp/parse_clickstream.py"]}
}
JSON
```

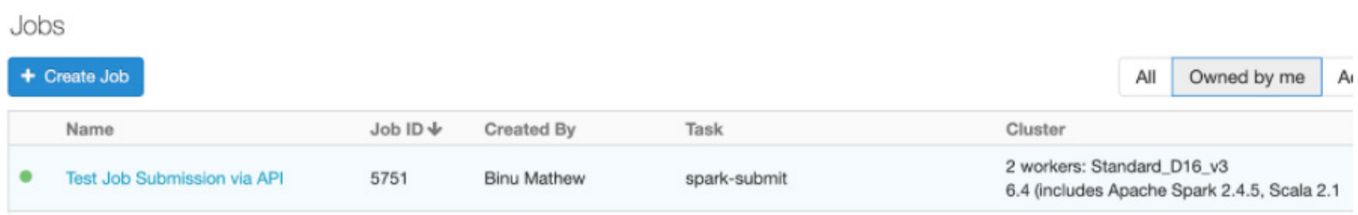
작업이 생성되도록 명령을 실행합니다. 이 예시의 경우, bash 스크립트로 실행될 명령을 포함한 파일을 생성합니다. 이렇게 하면 작업 ID가 반환됩니다.

```
(base) C02WW0HYHTD5:run_python bmathew$ bash ./create_spark_submit.sh
{"job_id":5751}
(base) C02WW0HYHTD5:run_python bmathew$
```

이 작업 ID를 사용하여 CLI에서 작업을 실행하세요. 이렇게 하면 실행 ID가 반환됩니다.

```
(base) C02WW0HYHTD5:run_python bmathew$ databricks --profile AZURE_PROD jobs run-now --job-id 5751
{
  "run_id": 8739,
  "number_in_job": 1
}
(base) C02WW0HYHTD5:run_python bmathew$
```

UI를 보면 작업이 실행 중인 것을 확인할 수 있습니다.



3. 기존 클러스터에 작업 제출

Databricks에서 spark-submit을 사용하여 작업을 제출하려면 기존 클러스터가 아니라 새 클러스터에서 실행해야 합니다. 기존 클러스터에서 작업을 실행하려면 spark_python_task로 작업을 생성하면 됩니다.

```
curl -n -H "Content-Type: application/json" -X POST -d @- https://eastus2.azuredatabricks.net/api/2.0/jobs/create <<JSON
{
  "name": "Test Job via API on existing cluster",
  "existing_cluster_id": "0407-061907-irk480",
  "spark_python_task": {
    "python_file": "dbfs:/bmathew/tmp/parse_clickstream.py"
  }
}
JSON
```

작업이 생성되도록 명령을 실행합니다. 이 예시에서는 명령이 포함된 파일을 생성했습니다. 이렇게 하면 작업 ID가 반환됩니다.

```
(base) C02WW0HYHTD5:run_python bmathew$ bash ./create_job_via_api_python_task.sh
{"job_id":5752}
(base) C02WW0HYHTD5:run_python bmathew$
```

작업 ID를 사용하여 CLI에서 작업을 실행합니다.

```
(base) C02WW0HYHTD5:run_python bmathew$ databricks --profile AZURE_PROD jobs run-now --job-id 5752
{
  "run_id": 8740,
  "number_in_job": 1
}
(base) C02WW0HYHTD5:run_python bmathew$
```

UI를 보면 작업이 실행 중인 것을 확인할 수 있습니다.

챕터 04

앞으로의 길잡이

다음 단계

챕터 4: THE PATH FORWARD

Next steps

Migration of your Hadoop environment to Databricks delivers significant business benefits, including:

- Reduction of operational cost
- Increased productivity of your data teams
- Unlocking of advanced AI and BI capabilities that drive top-line growth

Databricks, along with their preferred community of migration partners, is available to assist with your initiative by providing the following services:

- Inventorying your existing Hadoop landscape
- Developing a detailed future state reference architecture
- Quantifying the business benefits of migration
- Creating a joint implementation plan with your team
- Co-delivering a migration project
- Retiring your existing Hadoop environment

Please reach out to us at sales@databricks.com if you are interested in exploring Hadoop to Databricks migration.

MORE RESOURCES AVAILABLE AT DATABRICKS.COM/MIGRATION.

About Databricks

Databricks is the data and AI company. More than 5,000 organizations worldwide — including Comcast, Condé Nast, H&M and over 40% of the Fortune 500 — rely on the Databricks Lakehouse Platform to unify their data, analytics and AI. Databricks is headquartered in San Francisco, with offices around the globe. Founded by the original creators of Apache Spark™, Delta Lake and MLflow, Databricks is on a mission to help data teams solve the world's toughest problems. To learn more, follow Databricks on Twitter, LinkedIn and Facebook.

databricks