Data Streams

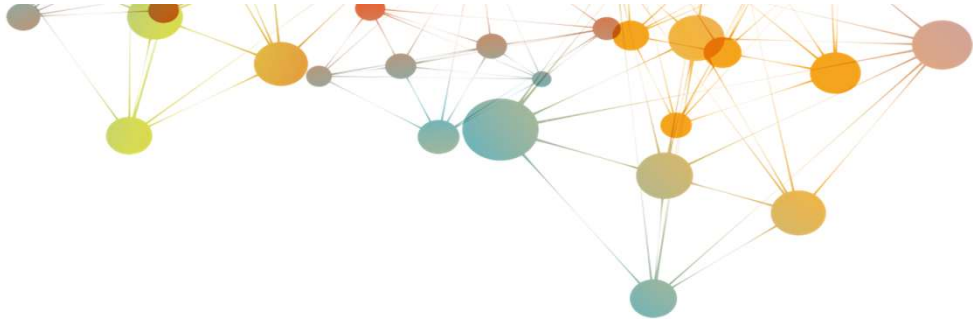
고급데이터베이스 7주차

Apache Spark 아키텍처의 이해 & 버전별 주요 기능

데이터스트림즈 기술연구소
길기범 수석연구원

목차

Section 1.



Apache Spark 아키텍처

작업 배포 방식과 리소스 설정 방식을 중심으로 Spark의 핵심 컴포넌트와 동작 구조를 심층 분석합니다

Apache Spark 개요

Apache Spark란?

- 대규모 데이터 처리를 위한 통합 분석 엔진으로, 인메모리 기반의 빠른 연산 속도가 특징
- 2009년 UC 버클리 대학의 AMPLab에서 시작되어 2013년 아파치 재단으로 이관된 오픈소스 프로젝트

주요 특징

- 속도: 인메모리 처리로 Hadoop MapReduce보다 최대 100배 빠른 연산
- 다양한 언어 지원: Scala, Java, Python, R 등 여러 프로그래밍 언어 API 제공
- 통합성: 배치, 스트리밍, 머신러닝, SQL 분석 등 다양한 워크로드를 단일 엔진에서 처리

핵심 개념

- 분산 처리: 여러 노드에 작업을 분산하여 대규모 데이터 병렬 처리
- 지연 평가(Lazy Evaluation): 실제 결과가 필요할 때까지 연산 최적화 및 지연

Spark 아키텍처의 핵심 요소

Driver

- 애플리케이션의 메인 프로그램으로 SparkContext 객체를 생성하고 모든 Spark 작업 제어
- 작업(Job) 분할, 태스크(Task) 스케줄링, 클러스터와의 통신 담당

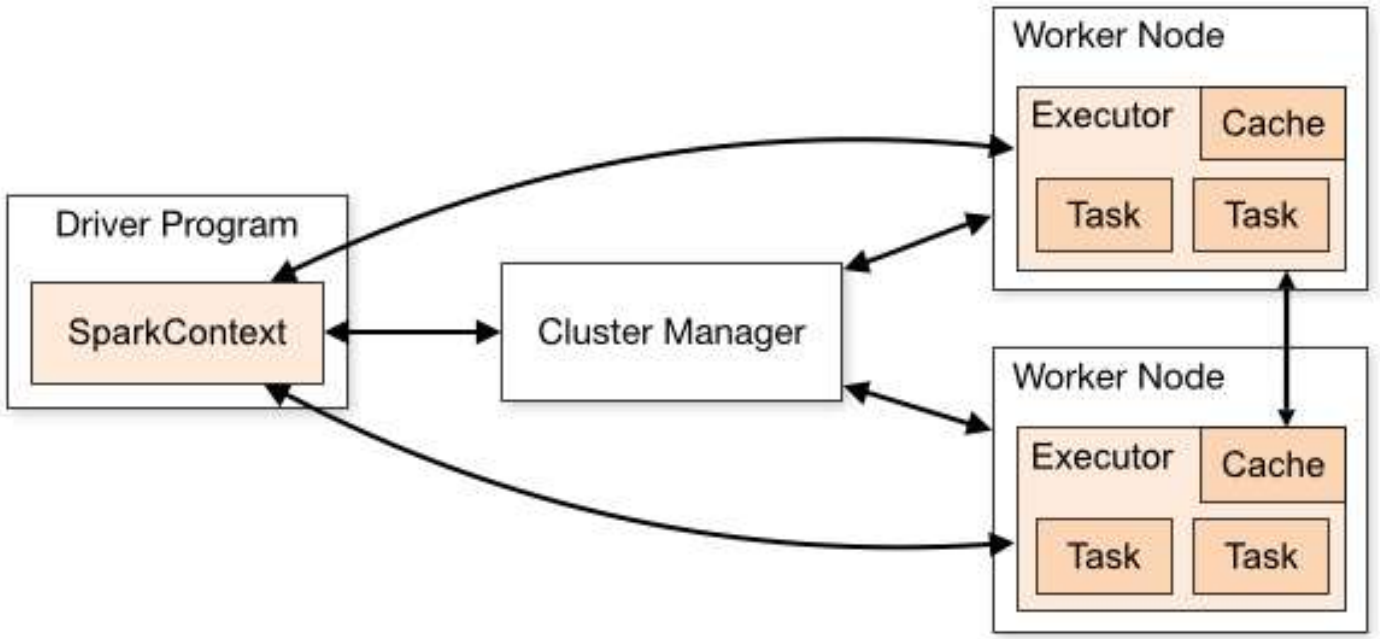
Executor

- 워커 노드에서 실행되는 프로세스로 실제 계산을 수행하고 데이터 저장
- 각 애플리케이션마다 독립적인 Executor 프로세스 집합이 할당됨
- 다중 스레드로 태스크 실행 및 RDD 데이터 캐싱 담당

Cluster Manager

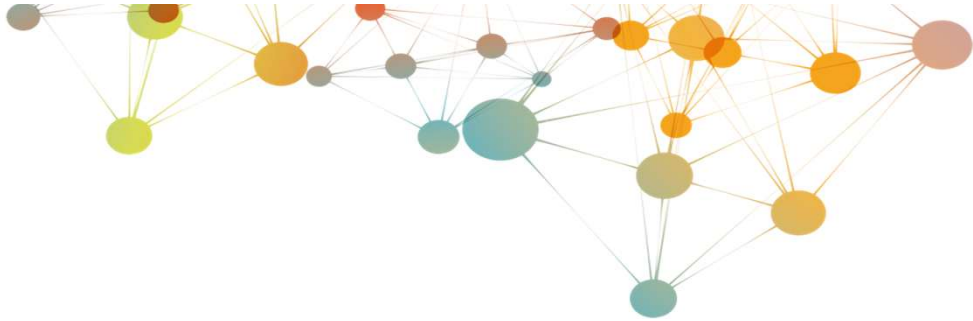
- 클러스터의 자원 관리자로 Executor 실행을 위한 리소스 할당
- Standalone, YARN, Kubernetes 등 다양한 유형 지원
- 클러스터 노드 간 자원 스케줄링 및 모니터링 수행

Spark 동작 구조



Data Streams

SparkContext와 애플리케이션 실행 구조



SparkContext란?

• Spark 애플리케이션과 클러스터 사이의 연결점(Connection Point)으로, 모든 Spark 작업의 시작점

• Spark 2.0부터는 SparkSession이 통합 진입점으로 제공되며, SparkContext를 내부적으로 포함

주요 역할

• 클러스터 연결 및 리소스 협상: 클러스터 매니저(YARN, Kubernetes 등)와 통신하여 자원 할당

• Job 스케줄링: DAG(Directed Acyclic Graph) 스케줄러를 통한 작업 최적화 및 실행 계획 수립

• RDD 생성 및 관리: 데이터의 분산 처리를 위한 RDD(Resilient Distributed Dataset) 관리

애플리케이션 실행 구조

• SparkContext 초기화 → 클러스터 리소스 확보 → 작업(Job) 제출 → 태스크(Task) 실행 → 결과 수집

• 하나의 JVM에는 동시에 하나의 활성화된 SparkContext만 존재 가능

작업 배포 방식 개요

배포 방식이란?

- Spark 애플리케이션의 Driver와 Executor가 어디에서 실행되는지를 결정하는 방식
- 리소스 활용, 성능, 관리 측면에서 각 배포 방식마다 장단점 존재

Local Mode

단일 머신에서 Driver와 Executor가 모두 하나의 JVM에서 실행됨

개발 및 테스트 용도로 적합

Client Mode

Driver는 클라이언트 머신에서, Executor는 클러스터에서 실행됨

대화형 작업이나 디버깅에 적합

Cluster Mode

Driver와 Executor 모두 클러스터 내에서 실행됨

프로덕션 환경에 가장 적합

- 배포 방식은 --deploy-mode 옵션으로 지정 (예: spark-submit --deploy-mode cluster)

Local Mode란?

개념 및 특징

- 단일 머신 실행 방식: 클러스터 환경이 아닌 단일 JVM 내에서 모든 Spark 컴포넌트(Driver와 Executor)가 실행됨
- 간편한 설정: 별도의 클러스터 매니저 없이 로컬 PC에서 즉시 실행 가능
- 스레드 활용: --master local[N] 옵션으로 CPU 코어 수(N) 지정 가능

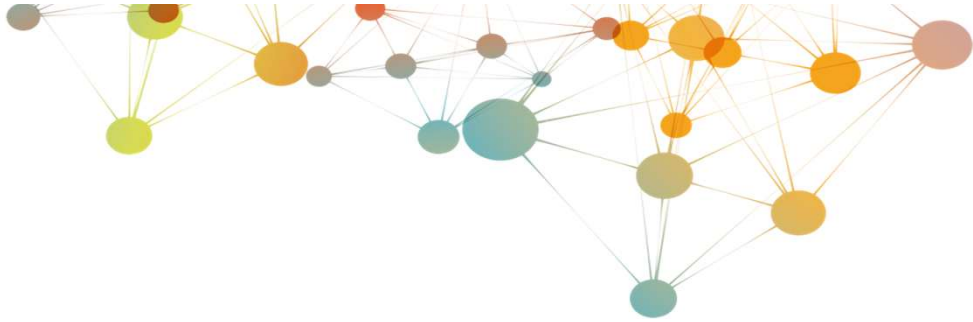
장단점

- 장점: 설정 간소화, 빠른 개발 및 디버깅, 리소스 사용 효율성
- 단점: 단일 노드 메모리/CPU 한계, 확장성 제한, 실제 분산 환경과 차이

적합한 사용 사례

- 개발 및 테스트: 코드 작성, 알고리즘 테스트, 디버깅
- 학습 및 교육: Spark 기능 학습, 예제 실행
- 소규모 데이터 분석: 단일 머신으로 처리 가능한 데이터셋 분석

Local Mode 실행 예시



spark-shell 실행 예제

• 기본 실행 (로컬 모드, 모든 코어 사용)

```
$ spark-shell --master local[*]
```

• 메모리 설정 및 특정 코어 수 지정

```
$ spark-shell --master local[4] --driver-memory 4g
```

주요 옵션 설명

• --master local[n]: 로컬 모드에서 사용할 스레드 수 지정 (n은 숫자 또는 * 사용)

• --driver-memory, --executor-memory: Driver/Executor에 할당할 메모리 지정

pyspark 실행 예제

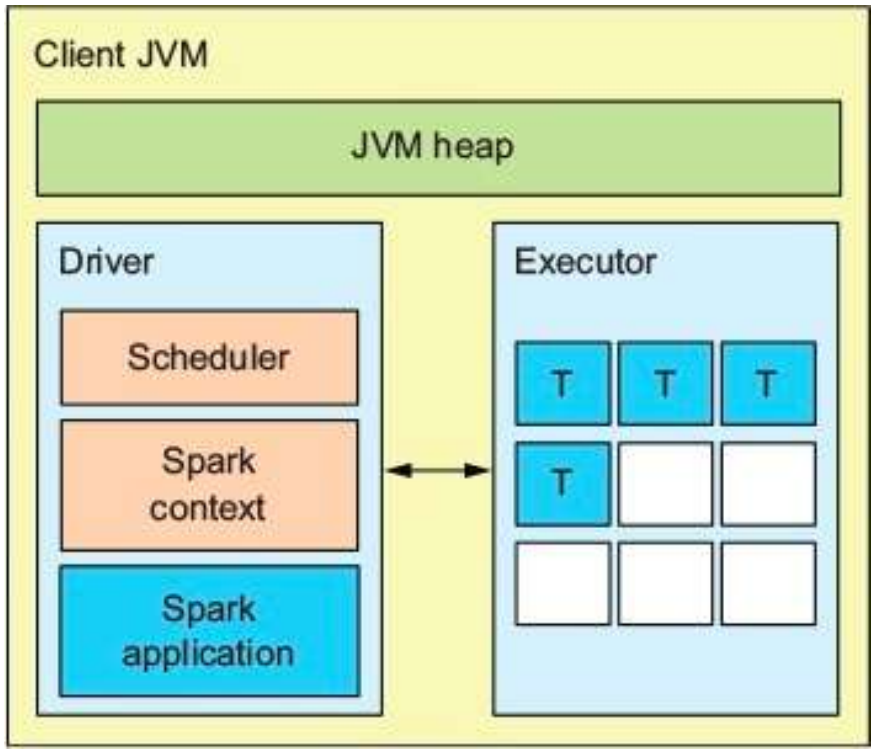
• Python 인터페이스로 로컬 모드 실행

```
$ pyspark --master local[*] --driver-memory 2g
```

• 패키지 추가 및 환경 변수 설정

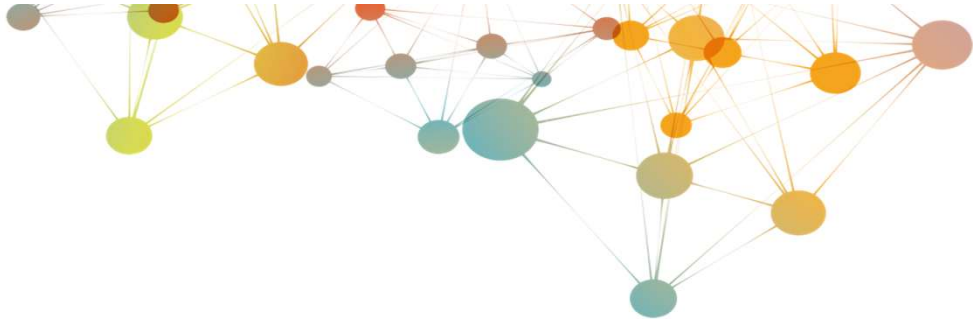
```
$ pyspark --master local[2] --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.3.0
```

Local Mode 구조



Data Streams

Client Mode란?



Client Mode의 기본 개념

- 드라이버(Driver)는 클라이언트의 로컬 머신에서 실행되고, 실행자(Executor)는 클러스터 내에서 실행되는 방식
- 사용자 애플리케이션의 메인 프로세스와 SparkContext가 클라이언트에서 구동됨

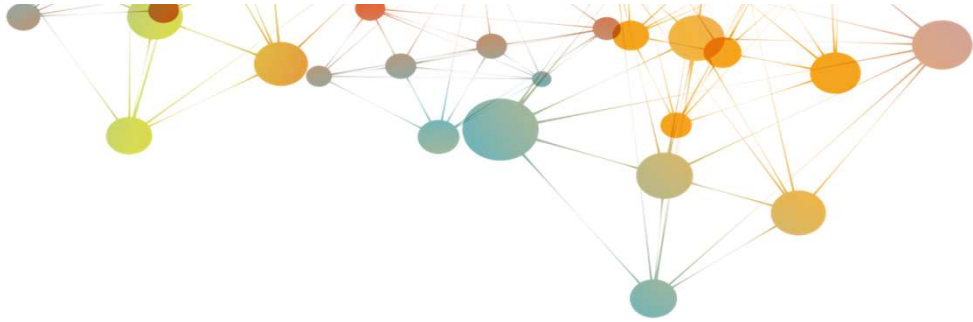
주요 특징

- 직접 결과 확인: 드라이버가 로컬에서 실행되어 실시간 로그 확인 및 디버깅 용이
- 네트워크 요구사항: 클라이언트와 클러스터 간 지속적인 네트워크 연결 필요
- 리소스 사용: 드라이버는 클라이언트 머신의 리소스를 사용, 클러스터 리소스는 Executor에만 할당

적합한 사용 사례

- 대화형 작업: Spark shell, Jupyter notebook과 같은 대화형 환경에 적합
- 개발 및 디버깅: 실시간 피드백이 필요한 개발 단계에 효과적

Client Mode 실행 예시



기본 명령어 구조

• --deploy-mode client 옵션을 사용하여 Driver를 로컬 머신에서 실행

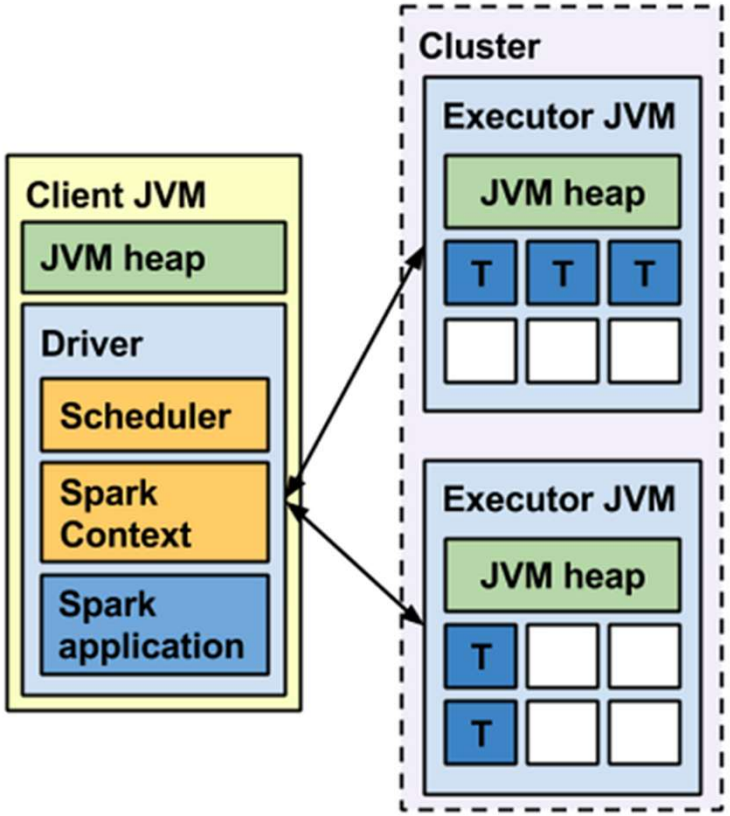
```
spark-submit --master yarn \
  --deploy-mode client \
  --class org.example.SparkApp \
  --executor-memory 4g \
  app.jar [arguments]
```

실전 설정 예시

• 대화형 데이터 분석 작업 예시

```
spark-submit --master yarn \
  --deploy-mode client \
  --num-executors 10 \
  --executor-cores 4 \
  --executor-memory 8g \
  --conf spark.yarn.am.memory=2g \
  --conf spark.driver.memory= 4g \
  example.py
```

Client Mode 구조



Data Streams

Cluster Mode란?

Cluster Mode 정의

- Driver와 Executor 모두가 클러스터 내부에서 실행되는 Spark 배포 방식
- 클라이언트가 애플리케이션을 제출하면 Cluster Manager가 Driver 프로그램을 워커 노드 중 하나에 할당하여 실행

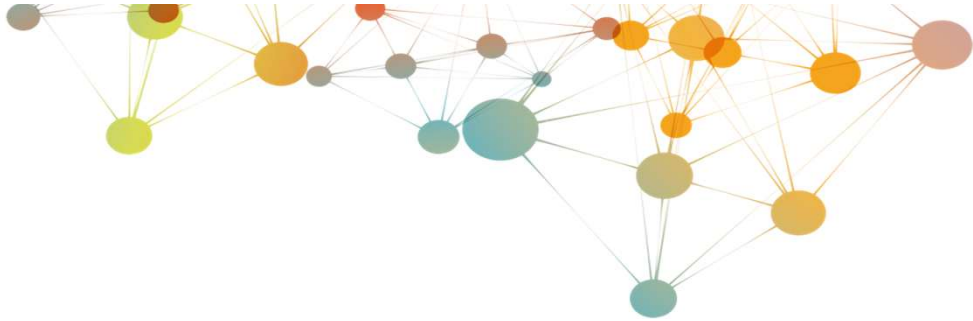
Cluster Mode의 장점

- 높은 가용성: 제출 클라이언트가 종료되어도 애플리케이션이 계속 실행됨
- 향상된 네트워크 성능: Driver와 Executor 간 통신이 클러스터 내부 네트워크로 이루어짐
- 프로덕션 환경에 적합: 장시간 실행되는 배치 작업과 스케줄링된 작업에 이상적

적합한 사용 사례

- 프로덕션 워크로드 및 자동화된 파이프라인
- 클라이언트 연결 독립적으로 실행되어야 하는 장기 실행 작업

Cluster Mode 실행 예시



기본 명령어 구조

• --deploy-mode cluster 옵션을 사용하여 Driver를 설정 된 클러스터 내부 에서 실행

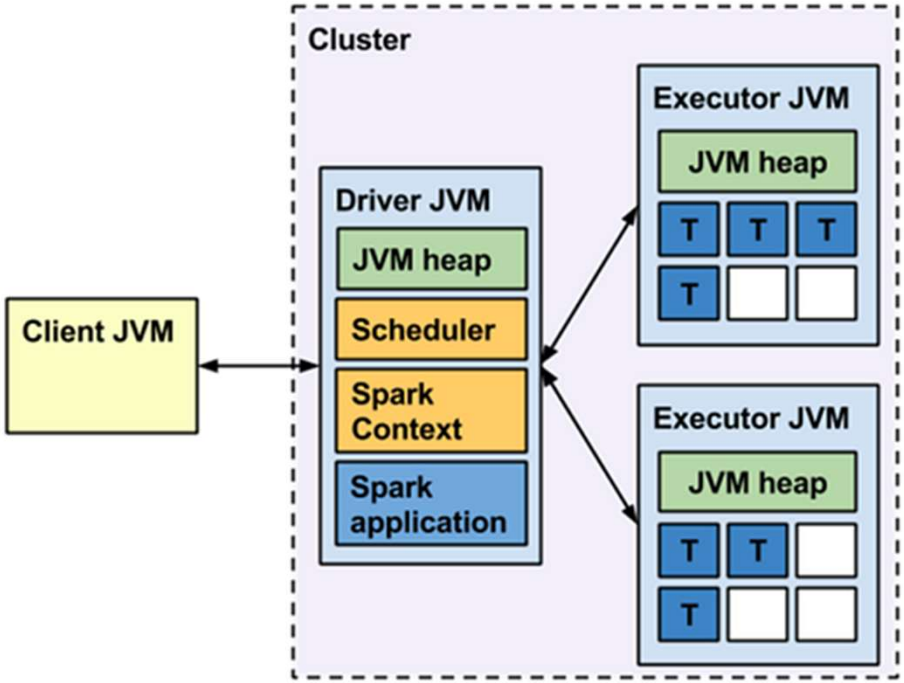
```
spark-submit --master yarn \
  --deploy-mode cluster \
  --class org.example.SparkApp \
  --executor-memory 4g \
  app.jar [arguments]
```

실전 설정 예시

대량 데이터 배치 작업 예시

```
spark-submit --master yarn \
  --deploy-mode cluster \
  --num-executors 10 \
  --executor-cores 4 \
  --executor-memory 8g \
  --conf spark.yarn.am.memory=2g \
  --conf spark.driver.memory= 4g \
  example.py
```

Cluster Mode 구조



Data Streams

작업 배포 방식별 특징 비교

배포 모드	설명	장점	단점	활용 사례
Local Mode	단일 JVM에서 Driver와 Executor가 모두 실행되는 방식	• 간단한 설정 • 빠른 개발/테스트 • 디버깅 용이	• 단일 머신 자원 한계 • 대용량 처리 제한	• 개발 환경 • 소규모 데이터 • 학습/실습
Client Mode	Driver는 클라이언트(제출자) 머신에서, Executor는 클러스터 노드에서 실행	• 대화형 분석 적합 • 실시간 출력 확인 • 애플리케이션 제어 용이	• Driver 장애 시 전체 영향 • 네트워크 대역폭 제약	• 대화형 분석 • 디버깅 • Spark-shell
Cluster Mode	Driver와 Executor 모두 클러스터 노드에서 실행되는 방식	• 높은 안정성 • 자원 활용 최적화 • 장애 대응 우수	• 대화형 실행 어려움 • 로그 확인 복잡	• 프로덕션 환경 • 배치 작업 • 장시간 작업

리소스 설정 방식 개요

클러스터 매니저의 역할

- Spark 애플리케이션의 자원 할당 및 관리를 담당하는 중요 컴포넌트
- Driver와 Executor에게 컴퓨팅 리소스를 제공하고 작업 스케줄링 지원

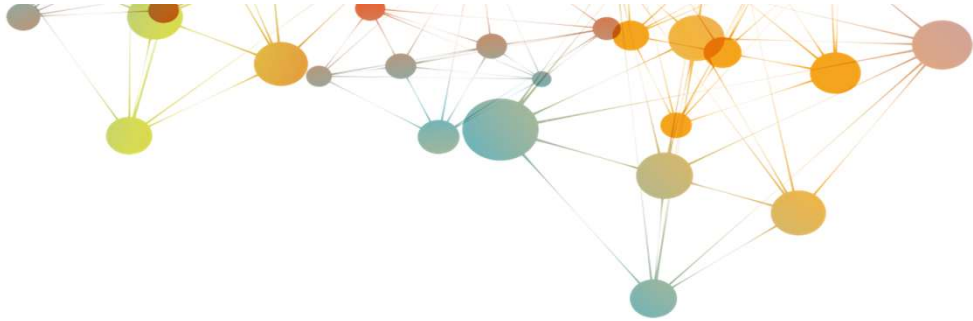
Spark 지원 클러스터 매니저 유형

- Standalone: Spark 자체 내장된 클러스터 관리자, 간단한 설정과 빠른 배포 가능
- YARN (Yet Another Resource Negotiator): Hadoop 3의 리소스 관리자, 대규모 클러스터에 적합
- Kubernetes: 컨테이너 기반 오케스트레이션 플랫폼, 동적 리소스 할당과 확장성 제공

클러스터 매니저 선택 고려사항

- 운영 환경 통합: 기존 인프라(Hadoop, 컨테이너 등)와의 연계성
- 관리 복잡성: 설정, 유지보수, 모니터링의 용이성

Standalone Mode란?



개념

• Apache Spark에 내장된 클러스터 매니저로, Hadoop과 같은 외부 리소스 관리자 없이도 독립적으로 동작

• Master-Worker 아키텍처를 기반으로 간단한 분산 환경 구축이 가능

주요 특징

• 설치 용이성: 별도의 분산 시스템 설치 없이 Spark만으로 클러스터 구성

• 고가용성(HA) 지원: ZooKeeper를 통한 마스터 노드 이중화 구성 가능

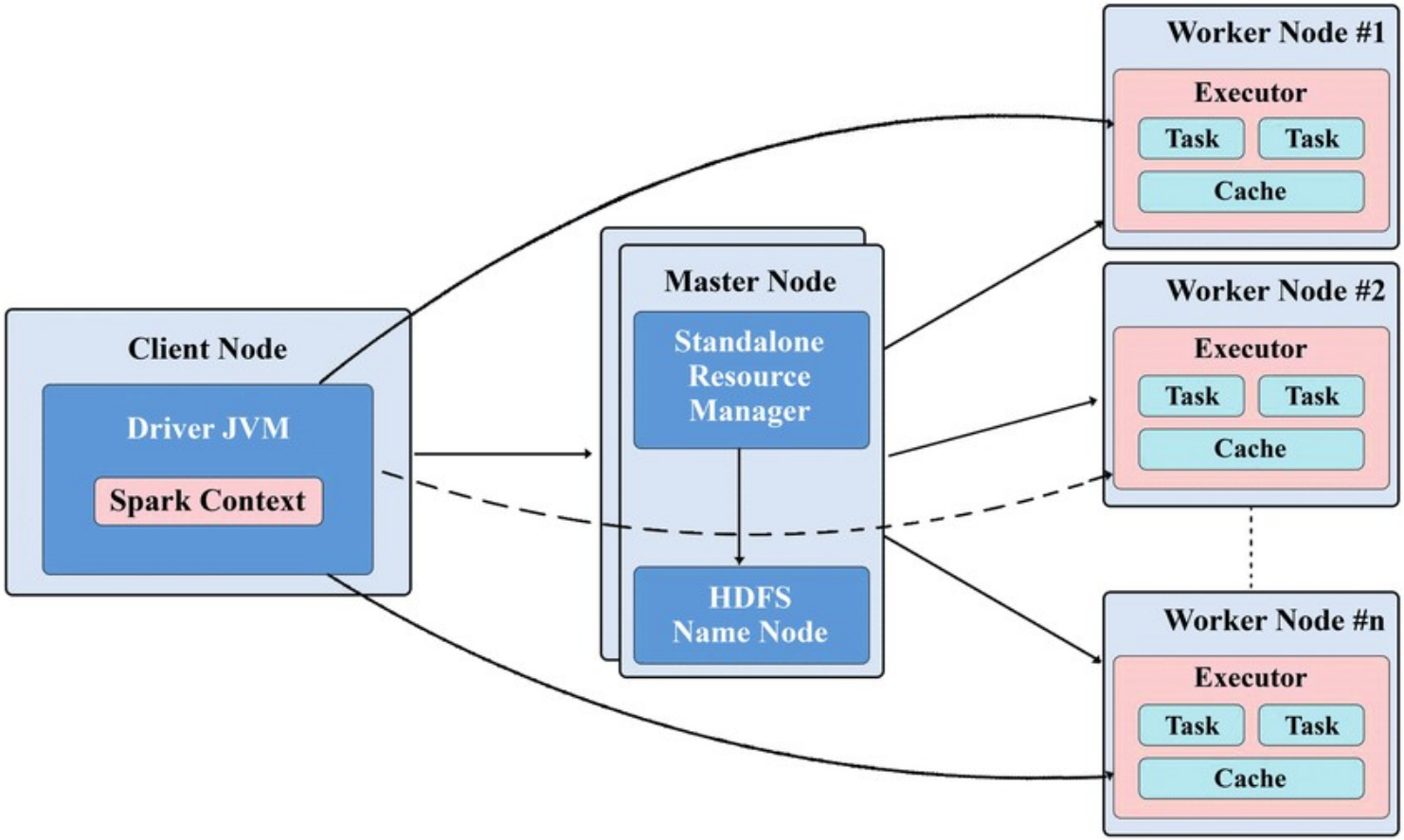
• 웹 UI 제공: 클러스터 상태 및 작업 모니터링을 위한 대시보드

구성 방법

• 시작 스크립트: start-master.sh와 start-worker.sh를 통한 간편한 실행

• 설정 파일: spark-env.sh와 spark-defaults.conf에서 리소스 할당 및 환경 설정

Standalone Mode 구조



Data Streams

YARN Mode란?

Hadoop YARN 개요

• YARN(Yet Another Resource Negotiator)은 Hadoop 2.0에서 도입된 클러스터 리소스 관리 시스템

• MapReduce와 분리된 범용 자원 관리자로, Spark를 포함한 다양한 분산 애플리케이션 지원

YARN의 주요 구성요소

• ResourceManager: 클러스터 전체 리소스를 관리하는 마스터 프로세스

• NodeManager: 각 노드의 리소스를 관리하는 에이전트

• ApplicationMaster: Spark 작업별 프로세스로, 리소스 요청 및 작업 조율 담당

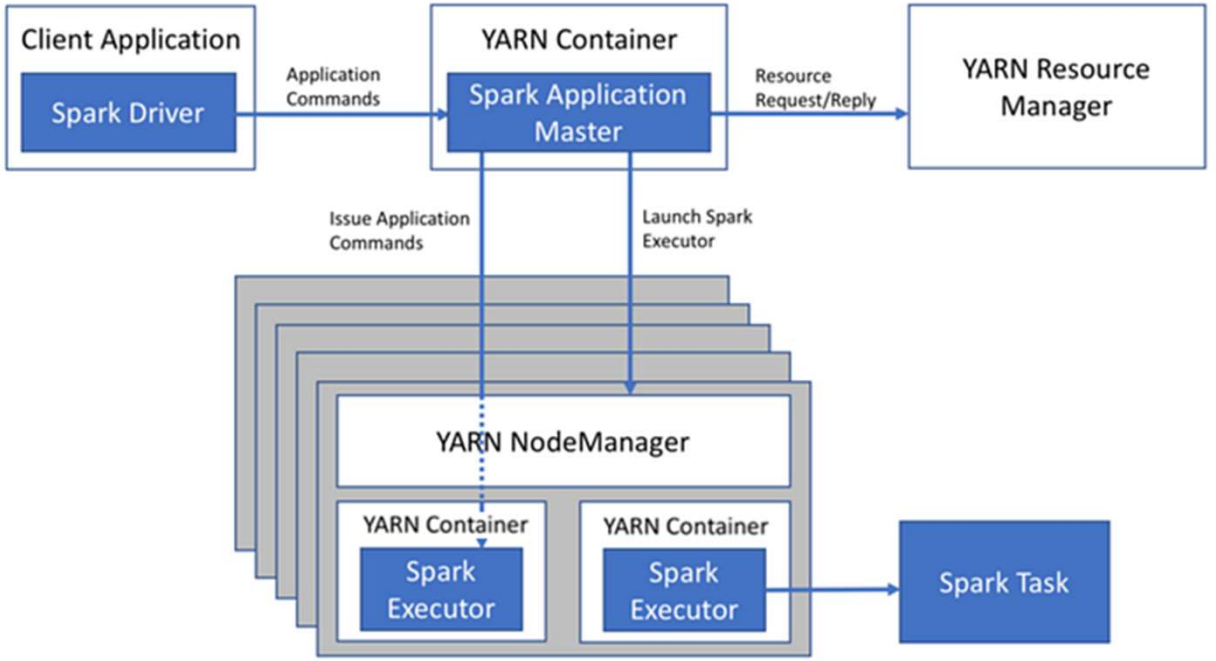
YARN Mode의 장점

• 다중 워크로드 지원: 하나의 클러스터에서 Spark, Hadoop, Hive 등 다양한 워크로드 실행 가능

• 동적 리소스 할당: 애플리케이션 요구에 따라 자원을 유연하게 조정

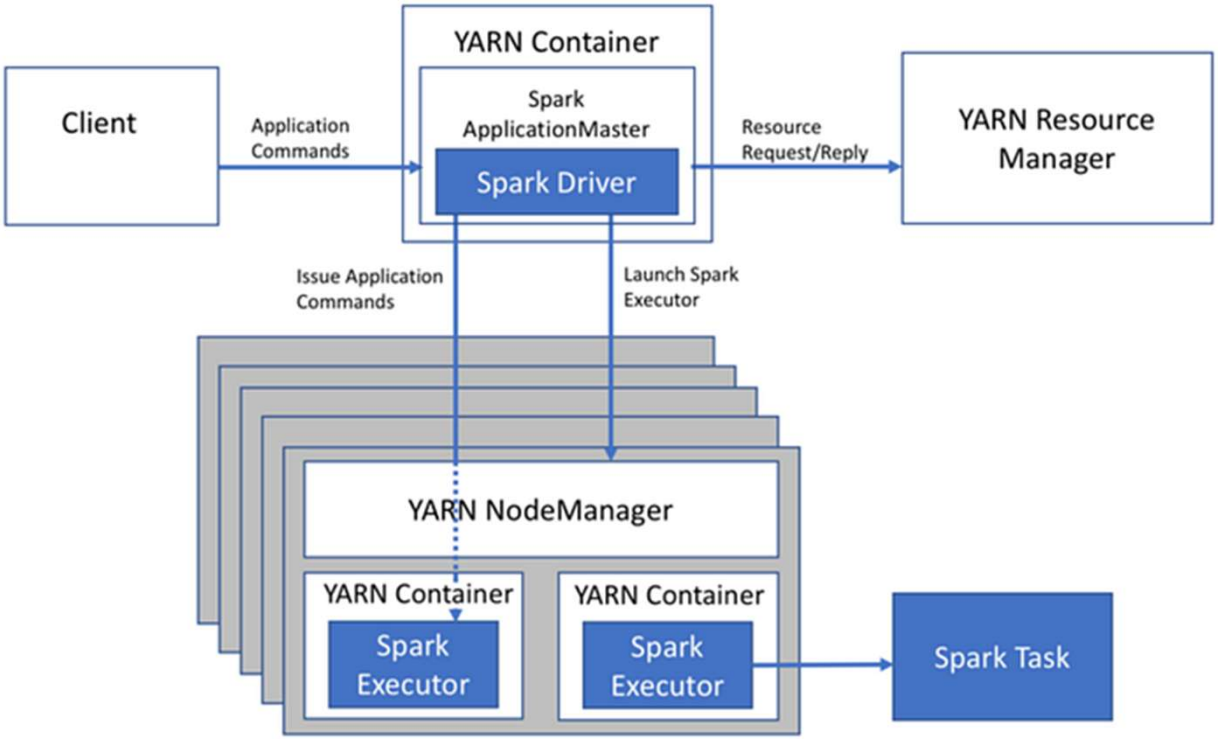
• 기존 Hadoop 인프라 활용: 별도 클러스터 구축 없이 기존 환경에서 Spark 실행

YARN Mode 구조(Client-mode)



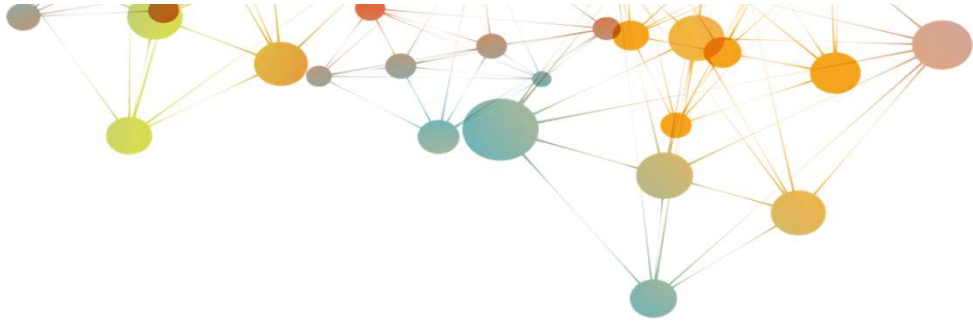
Data Streams

YARN Mode 구조(Cluster-mode)



Data Streams

Kubernetes Mode란?



Kubernetes 기반 Spark 실행

- 컨테이너 기반 오케스트레이션: Spark 작업이 Kubernetes 환경에서 컨테이너로 실행되는 방식
- Driver와 Executor가 각각 독립된 Pod로 실행되며, Kubernetes API로 리소스 할당 및 관리

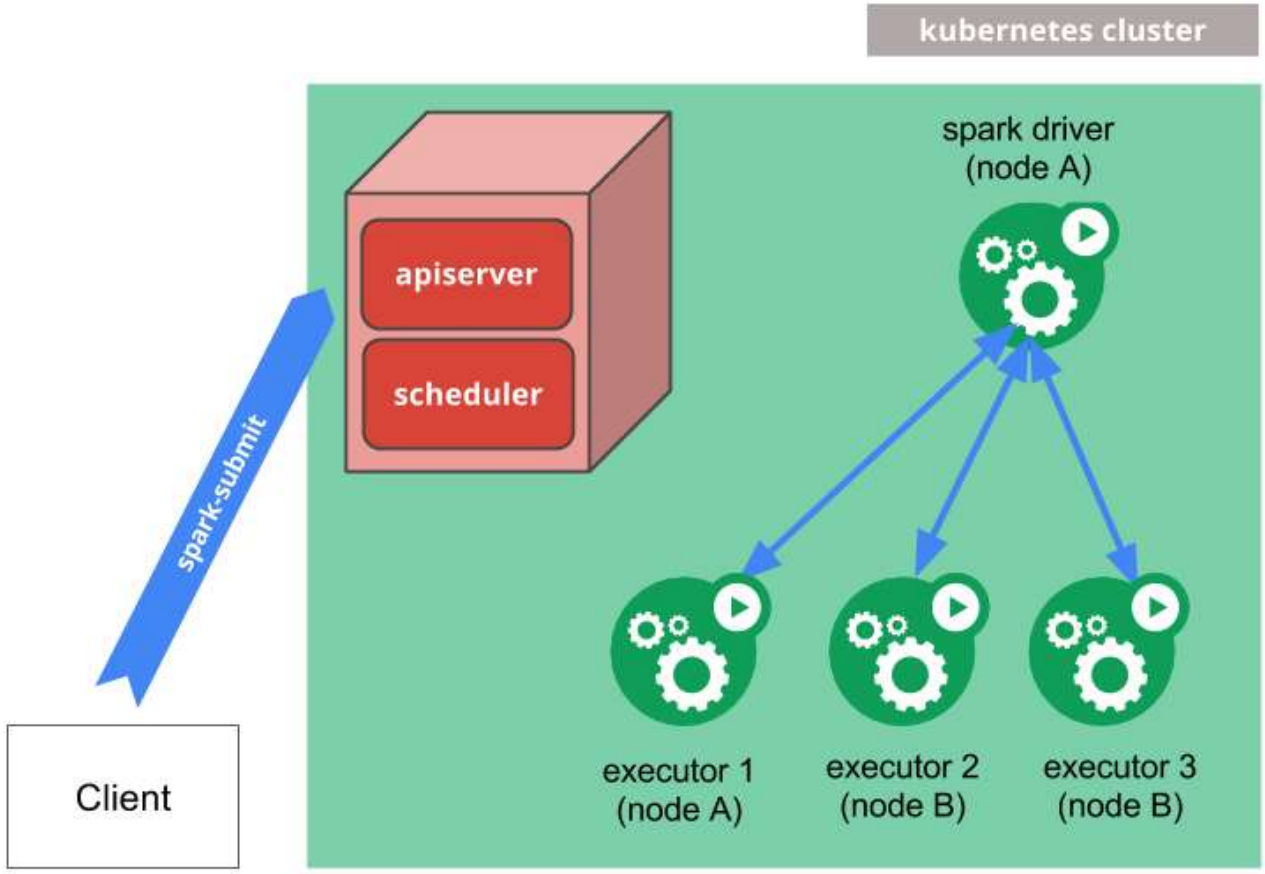
주요 특징

- 동적 리소스 할당: 애플리케이션 요구에 따라 컨테이너 자원을 탄력적으로 확장/축소
- 배포 자동화: 컨테이너 이미지 기반 일관된 환경 제공 및 자동화된 배포 파이프라인
- 인프라 독립성: 온프레미스, 클라우드 등 다양한 환경에서 일관된 방식으로 실행 가능

활용 사례

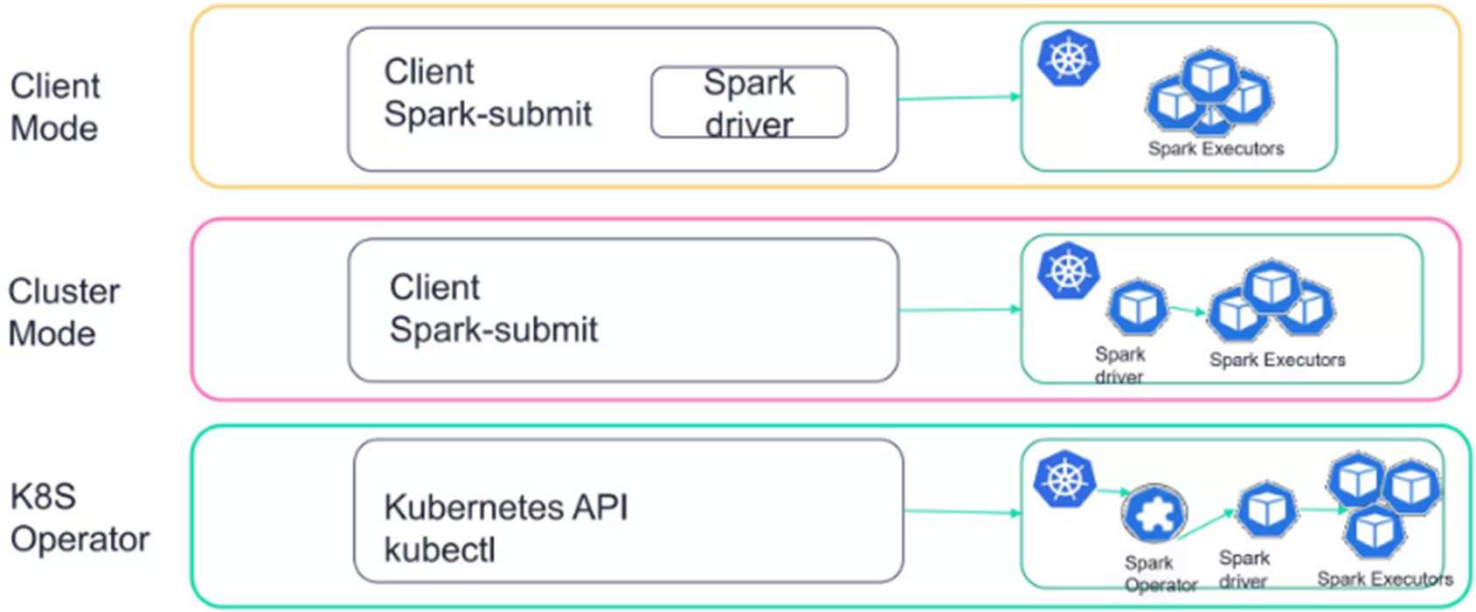
- 클라우드 네이티브 환경에서 멀티테넌트 Spark 클러스터 운영
- DevOps 파이프라인에 CI/CD 통합을 통한 데이터 애플리케이션 배포

Kubernetes Mode 구조



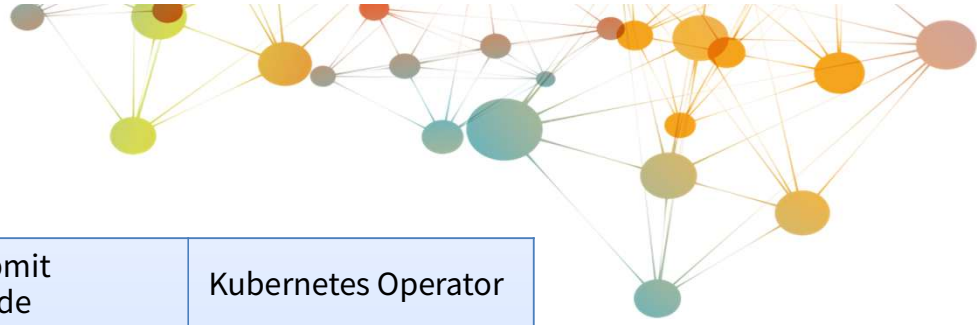
Data Streams

Kubernetes Mode 구조



Data Streams

Kubernetes Mode 구조

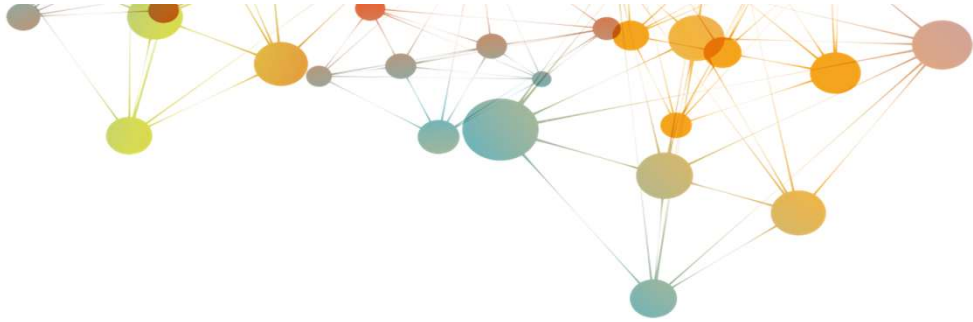


항목	spark-submit Cluster Mode	spark-submit Client Mode	Kubernetes Operator
배포 명령	spark-submit --deploy-mode cluster	spark-submit --deploy-mode client	kubectl apply -f app.yaml
Driver 위치	K8s Pod	클러스터 외부/별도 Pod	K8s Pod
제출 후 동작	백그라운드 실행 가능	계속 실행 필요 ⚠️	즉시 종료
재시작 정책	Pod restartPolicy	없음 ❌	고급 재시작 정책 ✅
상태 추적	kubectl만 가능	kubectl만 가능	SparkApplication Status
네트워크 설정	자동	수동 (복잡)	자동
GitOps 통합	어려움	어려움	용이 ✅
모니터링	수동 설정	수동 설정	Prometheus 통합
스케줄링	외부 도구 필요	외부 도구 필요	CronSchedule 지원
의존성 관리	수동	수동	Dependencies 섹션
학습 곡선	낮음	중간	높음
프로덕션 적합	✅ 적합	⚠️ 부적합	✅✅ 매우 적합

리소스 설정 방식별 특징 비교

리소스 관리자	설명	장점	단점	활용 사례
Standalone	Spark에 내장된 자체 클러스터 관리자로, 독립 실행 환경 제공	• 별도 설치 불필요 • 간편한 설정 • 낮은 의존성	• 제한적 스케줄링 • 동적 할당 한계 • 멀티 테넌시 제한	• 소규모 팀 • 전용 클러스터 • 간단한 워크로드
YARN	Hadoop의 리소스 관리자로, 다양한 워크로드와 자원 공유 지원	• 중앙집중식 관리 • 세밀한 자원 제어 • 다양한 스케줄러	• Hadoop 의존성 • 복잡한 설정 • 컨테이너 제약	• 기업 환경 • 대규모 데이터 • 기존 Hadoop
Kubernetes	컨테이너 오케스트레이션 플랫폼 기반 자원 관리 방식	• 클라우드 네이티브 • 동적 확장성 • 격리성 우수	• 설정 복잡성 • 러닝커브 높음 • 스토리지 관리	• 마이크로서비스 • 클라우드 환경 • DevOps 조직

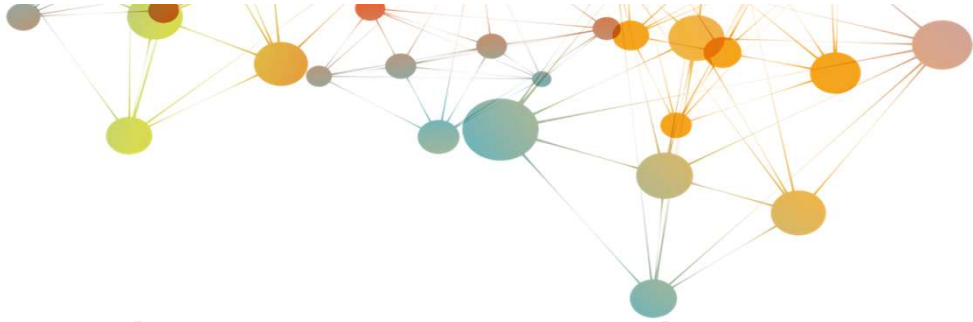
Section 2.



Apache Spark 출시 타임라인

Spark 1.0부터 최신 버전까지의 주요 릴리즈 및 핵심 기능 변화 과정을 시간순으로 살펴봅니다

Apache Spark 출시 타임라인



Spark 1.x

2014-2016

Spark 1.0

2014년 5월

최초 안정화 버전, RDD 기반 API

Spark 1.2

2014년 12월

DataFrame API 최초 도입

Spark 1.6

2015년 12월

Dataset API 도입, 머신러닝 파이프라인

Spark 2.x

2016-2019

Spark 2.0

2016년 7월

Structured Streaming, 통합 API

Spark 2.2

2017년 7월

Structured Streaming 프로덕션 지원

Spark 2.4

2018년 11월

Kubernetes 지원, Barrier 실행 모드

Spark 3.x

2020-2024

Spark 3.0

2020년 6월

Adaptive Query Execution, ANSI SQL

Spark 3.2

2021년 10월

PySpark pandas API 개선

Spark 3.5

2023년 9월

Delta Lake 통합 확장

Spark 4.x

2025-현재

Spark 4.0

2025년 5월

Columnar Execution, 개선된 Spark Connect

Spark 4.1

2025년 9월

최신 릴리스, SQL 기능 확장

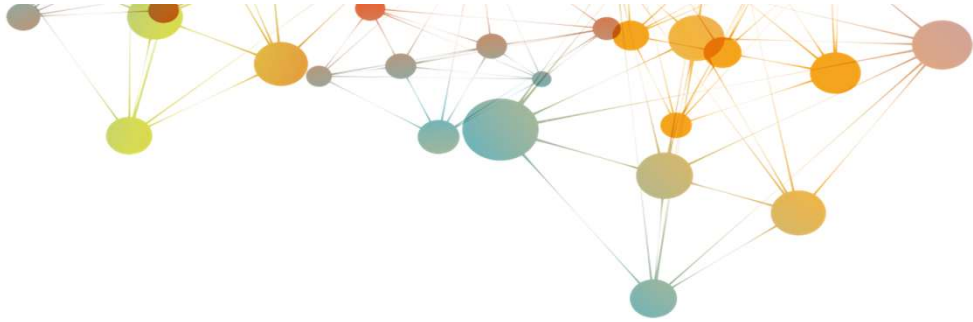
버전별 대표 기능 요약 #1

버전	출시일	대표 기능	핵심 개선점
Spark v1.x (1.0 - 1.6)	2014년 5월 - 2016년 1월	• RDD (Resilient Distributed Dataset) • Spark SQL • Mllib • GraphX	• 인메모리 분산 컴퓨팅 기반 • Hadoop MapReduce 대비 10-100배 성능 향상 • DataFrame API (v1.3부터) • R 언어 지원 (v1.4부터)
Spark v2.x (2.0 - 2.4)	2016년 7월 - 2019년 11월	• Dataset API • Structured Streaming • Catalyst Optimizer • Tungsten 엔진	• 쿼리 최적화 엔진 도입 • 강타입 API와 런타임 타입 안정성 • 코드 생성으로 실행 속도 향상 • 메모리 관리 최적화

버전별 대표 기능 요약 #2

버전	출시일	대표 기능	핵심 개선점
Spark v3.x (3.0 - 3.5)	2020년 6월 - 2023년 10월	• Adaptive Query Execution (AQE) • Dynamic Partition Pruning (DPP) • Python API 확장 • ANSI SQL 호환성	• 런타임 통계 기반 쿼리 최적화 • 불필요한 파티션 스캔 제거로 성능 향상 • Python UDF 가속화 • GPU 지원 강화
Spark v4.x (4.0 - 현재)	2025년 5월 - 현재	• Columnar Execution • Spark Connect • Catalyst 개선 • Python 생태계 통합	• Arrow 기반 컬럼 지향 처리 • 클라이언트-서버 아키텍처 분리 • PySpark 성능 20-50% 향상 • SQL 언어 확장 및 호환성 강화

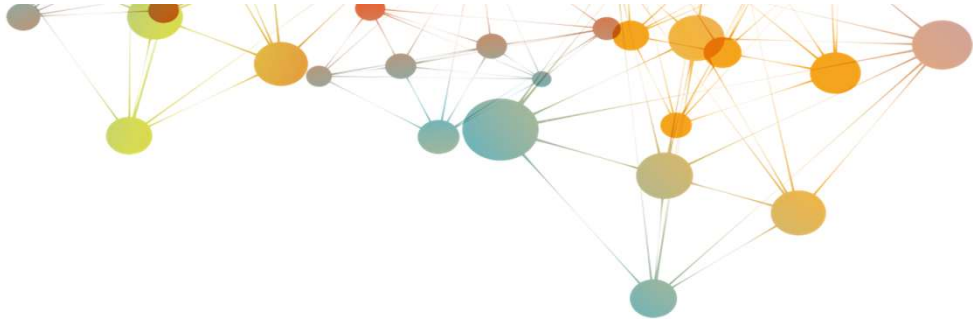
Section 3.



Apache Spark v1

RDD, Spark SQL, MLlib, GraphX 등 Spark 1.x의 핵심 컴포넌트와 기능적 특징을
살펴보고 기초 분산처리 패러다임을 이해합니다

Spark v1 핵심 기능



Apache Spark v1 주요 모듈

RDD (Resilient Distributed Dataset)

Spark의 기본 데이터 추상화로, 클러스터에 분산된 변경 불가능한(immutable) 데이터 컬렉션. 장애 복구 기능과 병렬 처리 기반

Spark SQL

정형 데이터 처리를 위한 인터페이스 제공. SQL과 DataFrame API 지원으로 대규모 데이터셋 처리 및 분석

MLlib

분산 머신러닝 라이브러리로 분류, 회귀, 군집화, 협업 필터링 등 다양한 알고리즘과 유틸리티 제공

GraphX

그래프 처리 및 그래프 병렬 연산을 위한 API 제공. 소셜 네트워크, 경로 최적화 등의 그래프 분석에 활용

RDD란 무엇인가?

RDD(Resilient Distributed Dataset)의 정의

- Spark의 핵심 데이터 추상화 단위로, 클러스터에 분산된 요소들의 집합
- 여러 노드에 분산된 데이터셋으로, 장애 발생 시에도 스스로 복구 가능한 구조

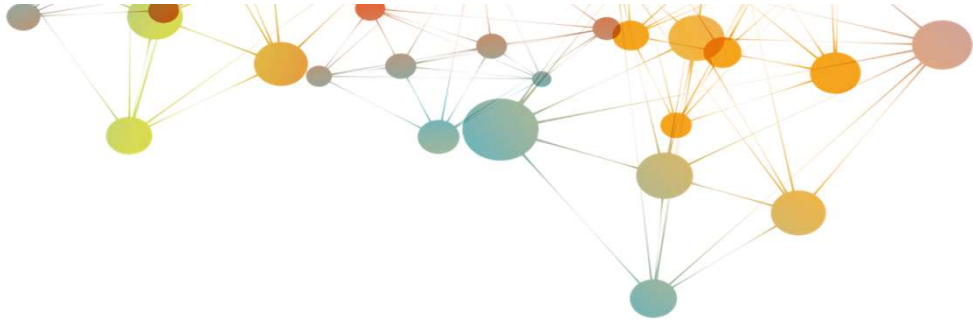
RDD의 주요 특징

- 복원성(Resilience): 노드 장애 시 데이터 손실 없이 자동 복구 가능
- 분산처리(Distribution): 클러스터 전체에 데이터가 분산되어 병렬 처리
- 불변성(Immutability): 생성 후 변경 불가능, 변환 시 새로운 RDD 생성
- 지연 연산(Lazy Evaluation): 액션 연산이 호출될 때까지 실제 계산 지연

RDD 생성 방법

- 외부 데이터 소스(파일, HDFS)로부터 로드하여 생성
- 기존 RDD에 변환 연산(Transformation)을 적용하여 생성

RDD 작동 원리



Transformation 연산

- 새로운 RDD를 생성하는 연산으로, 원본 RDD를 변경하지 않음
- 예: map(), filter(), flatMap(), groupByKey(), reduceByKey() 등

Action 연산

- 결과값을 반환하거나 저장하는 연산으로, 실제 계산 트리거
- 예: collect(), count(), first(), take(), saveAsTextFile() 등

Lazy Evaluation (지연 평가)

- Transformation은 즉시 수행되지 않고 Action이 호출될 때까지 지연됨
- 이를 통해 실행 계획 최적화와 불필요한 계산 방지 가능
- DAG(Directed Acyclic Graph)를 구성하여 효율적인 연산 경로 설계

DataFrame / Dataset

DataFrame?

- 데이터를 테이블 형태(행/열)로 표현하는 분산 데이터 컬렉션
- RDBMS의 테이블 혹은 R/Python의 DataFrame과 유사한 개념

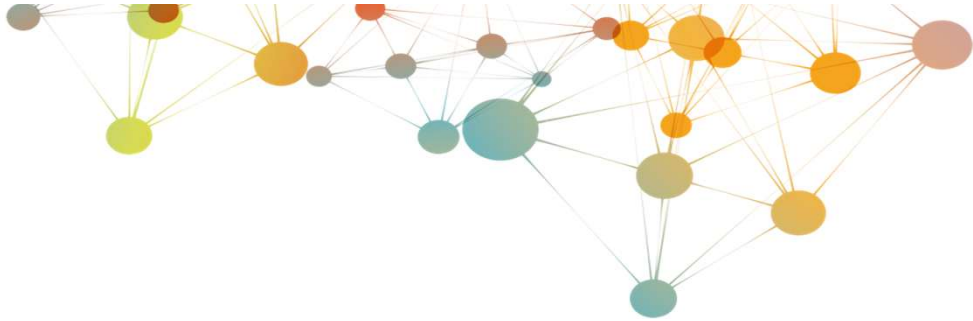
Dataset?

- 타입의 다양성을 제공해주는 분산 데이터 컬렉션
- Dataset<Row> = DataFrame

주요 특징

- 구조화 된 스키마 기반 데이터 처리
- Catalyst 옵티마이저를 통한 실행 계획 최적화
- 향상 된 개발 생산성

Spark SQL



데이터 질의를 위한 SQL 인터페이스

- Spark SQL은 Spark 1.0에서 처음 도입된 SQL 기반 데이터 처리 모듈
- 분석가를 위한 대화형 유틸리티

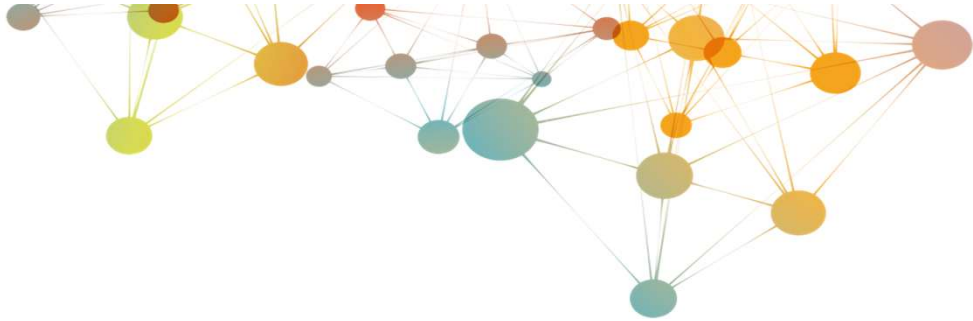
주요 기능

- SchemaRDD 도입: 초기 DataFrame의 전신, 데이터에 스키마 정보 부여
- Hive 메타스토어 연동: 기존 Hive 테이블 접근 및 쿼리 가능
- JDBC/ODBC 서버: 외부 BI 도구에서 Spark 데이터 접근 지원

한계점

- 제한적인 최적화: Catalyst 최적화가 아직 초기 단계

MLlib



MLlib 개요

- Spark v1에서 제공하는 기본 머신러닝 라이브러리로 분산 환경에서 ML 알고리즘 실행
- 대용량 데이터에 대한 스케일링 가능한 머신러닝 기능 지원

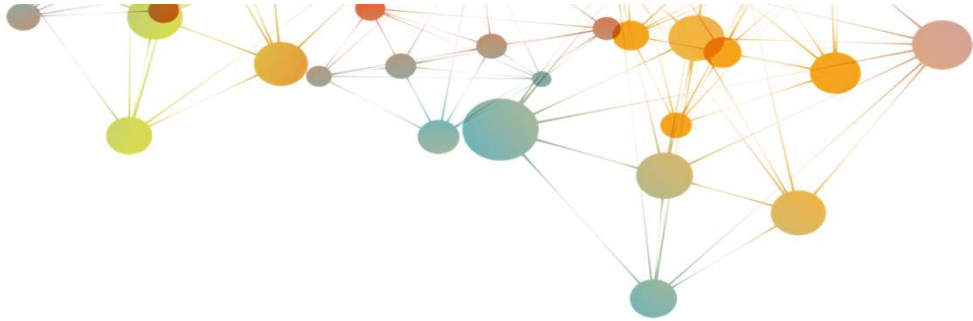
주요 알고리즘

- 분류 및 회귀: 로지스틱 회귀, 선형 회귀, SVM, 나이브 베이즈
- 클러스터링: K-means, 가우시안 혼합, Power Iteration 클러스터링
- 추천: ALS(Alternating Least Squares) 기반 협업 필터링
- 차원 축소: SVD(Singular Value Decomposition), PCA

특징과 한계

- RDD 기반 API를 사용하여 분산 메모리에서 효율적으로 동작
- v2에서는 DataFrame 기반의 spark.ml 패키지로 발전 (더 확장된 기능)

GraphX



GraphX란?

• Spark v1에서 도입된 그래프 데이터 처리를 위한 분산 그래프 프로세싱 프레임워크

• RDD 기반으로 구축되어 정점(Vertex)과 간선(Edge)으로 구성된 방향성 그래프 구조 제공

주요 특징

• ETL 프로세스: 그래프를 표 형식과 그래프 형식 간 변환 가능

• 내장 알고리즘: PageRank, 연결 구성요소, 최단 경로, 삼각형 카운팅 등 제공

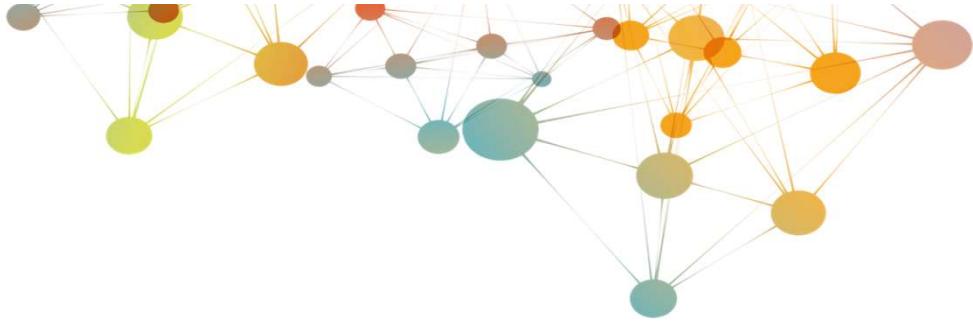
• Pregel API: 반복적 그래프 계산을 위한 Google의 Pregel 모델 구현 지원

활용 분야

• 소셜 네트워크 분석: 영향력 있는 사용자 식별, 커뮤니티 감지

• 추천 시스템: 사용자-아이템 관계를 그래프로 모델링한 추천 알고리즘

생태계의 확장



초기 Spark 생태계 확장

• Spark 1.x 시리즈는 핵심 API 및 인터페이스 표준화를 통해 다양한 확장 모듈과 외부 연동을 지원

• Hadoop 에코시스템과의 강력한 통합으로 기존 빅데이터 인프라와 원활한 연동

주요 생태계 확장 솔루션

• Zeppelin: 대화형 노트북 환경에서 Spark 분석 워크플로우 지원

• Alluxio(구 Tachyon): 메모리 중심 분산 스토리지 시스템과의 통합

• Kafka 연동: 실시간 데이터 파이프라인 구축을 위한 Receiver 기반 통합

산업계 영향

• 다양한 BI 도구 및 ETL 솔루션이 Spark 연동 기능을 추가하기 시작

• 오픈소스 생태계와 상용 솔루션의 조화로운 발전으로 기업 도입 가속화

코드 예제

이름별 나이의 평균 구하기

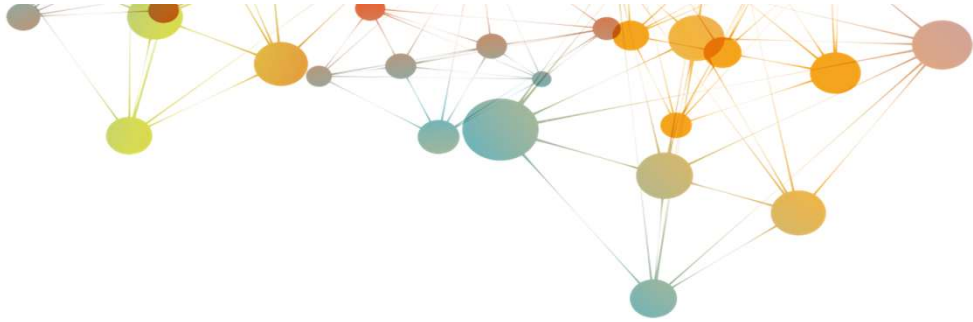
```
# rdd example
dataRDD = sc.parallelize([("영희", 20), ("철수", 30), ("민수", 22), ("재원", 28)])

agesRDD = (dataRDD.map(lambda x: (x[0], (x[1], 1)))\
    .reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) \
    .map(lambda x: (x[0], x[1][0] / x[1][1])))
```

- RDD 예시의 경우, 람다 표현식이 어떻게 키를 집계하고 평균 계산을 하는지 직관적으로 알기 쉬움

```
# dataframe example
data_df = ss.createDataFrame([("영희", 20), ("철수", 30), ("민수", 22),
                              ("재원", 28)], ["name", "age"])
avg_df = data_df.groupBy("name").agg(avg("age"))
```

- DataFrame API 예시의 경우는 스파크가 **무엇을 하는지**가 명확하게 보임

Section 4.



Apache Spark v2

DataFrame과 Dataset API, Catalyst 옵티마이저, Tungsten 엔진, Structured S
treaming 등 Spark 2.x의 혁신적 기능과 성능 개선 사항을 살펴봅니다

Spark v2 핵심 기능

Spark v2 주요 혁신

• Spark 2.0(2016년)부터 성능과 사용성에 집중한 구조적 혁신 도입

핵심 기능

• Dataset/DataFrame API: 강력한 타입 안정성과 성능을 모두 갖춘 고수준 API

• Catalyst 최적화 엔진: 쿼리 플랜을 자동으로 최적화하는 고급 쿼리 옵티마이저

• Tungsten 실행 엔진: 메모리 관리와 CPU 효율성을 극대화한 코드 생성 엔진

• Structured Streaming: 배치 처리와 동일한 API로 스트림 데이터 처리 지원

성능 향상

• Spark 1.x 대비 최대 10배 빠른 연산 속도 (특히 SQL 및 DataFrame 연산)

• 런타임 코드 최적화를 통한 효율적인 CPU와 메모리 사용

DataFrame & Dataset 통합

DataFrame & Dataset 개념

• Spark 1.3에서 DataFrame이, Spark 1.6에서 Dataset이 도입되어 Spark 2.0에서 통합 API로 완성

• RDB의 테이블과 유사한 구조화된 데이터 처리 모델 제공

주요 특징

• 스키마 정보를 기반으로 한 최적화된 데이터 처리

• Catalyst 최적화 엔진을 통한 자동 쿼리 최적화

• 다양한 타입 사용 가능

RDD와의 차이점

• 선언적 프로그래밍 방식으로 구현 세부사항 추상화 (RDD는 명령형)

• SQL 쿼리 실행과 표준 데이터 형식 지원 강화

Catalyst Optimizer란?

개념

- Spark SQL에 도입된 쿼리 최적화 프레임워크로, DataFrame과 SQL 쿼리를 자동으로 최적화
- 사용자 코드를 분석하여 실행 계획을 최적화하는 역할을 수행하는 핵심 엔진

논리적 최적화

- 규칙 기반 최적화: 조건절 푸시다운(Predicate Pushdown), 불필요한 연산 제거 등 적용
- 연산자 순서 재정렬과 불필요한 표현식 제거를 통해 최적의 논리 계획 도출

물리적 최적화

- 논리적 계획을 실제 실행 가능한 코드로 변환하는 단계
- 비용 기반 최적화: 여러 물리적 실행 방식 중 통계 정보를 활용해 가장 효율적인 방식 선택
- Spark 2.x부터 도입된 전체 단계 코드 생성(Whole-Stage CodeGen)으로 성능 극대화

Tungsten 엔진 개요

Tungsten 프로젝트란?

• Spark 2.0에서 도입된 하드웨어 최적화 프로젝트로, 모던 하드웨어에서 Spark의 성능을 극대화

• JVM의 한계를 뛰어넘는 메모리 관리 및 CPU 효율성 개선에 초점

메모리 관리 최적화

• 오프-힙 메모리 관리: Java 가비지 컬렉션의 영향을 받지 않는 직접 메모리 관리

• 바이너리 포맷: 객체 직렬화/역직렬화 오버헤드 제거를 위한 바이너리 메모리 표현

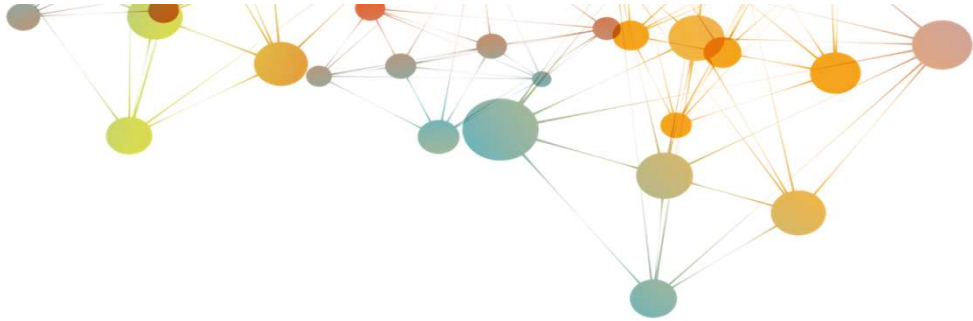
• 캐시 인지 연산: CPU 캐시 지역성을 고려한 데이터 구조 설계

CPU 효율성 향상

• 전체 스테이지 코드 생성: 여러 연산자를 단일 함수로 합쳐 가상 함수 호출 오버헤드 제거

• 벡터화 처리: SIMD 명령어를 활용한 데이터 병렬 처리 지원

Structured Streaming 등장



Structured Streaming이란?

- Spark 2.0에서 처음 도입된 새로운 스트리밍 데이터 처리 엔진
- 스트리밍 데이터를 계속 추가되는 테이블로 간주하는 획기적인 패러다임 제시
- RDD, DStreams -> DataFrame, Dataset, Structured Streaming

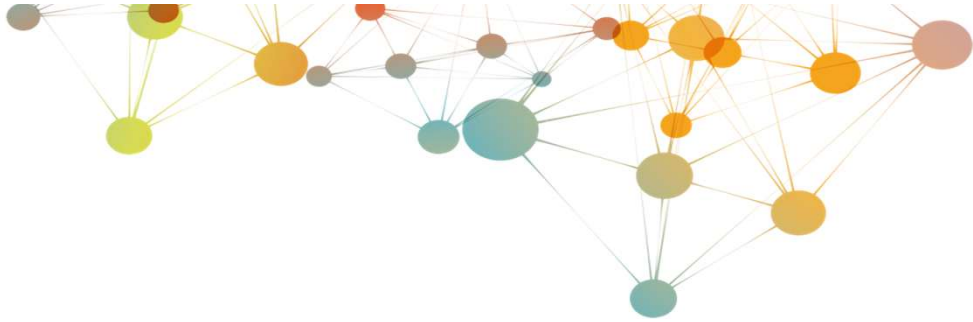
마이크로배치 처리 방식

- 데이터를 작은 배치(micro-batch) 단위로 처리하여 처리량과 지연시간의 균형 확보
- 스트림 처리 결과가 일괄 처리(batch)와 정확히 동일한 결과 보장

주요 장점

- SQL, DataFrame, Dataset API를 사용한 통합 개발 경험
- Catalyst 옵티마이저를 통한 스트리밍 쿼리 최적화 가능

Section 5.



Apache Spark v3

Adaptive Query Execution(AQE), Dynamic Partition Pruning(DPP) 등 고도화 된 쿼리 최적화와 ANSI SQL 지원을 통한 성능 혁신

Spark v3 핵심 기능

Adaptive Query Execution (AQE)

- 런타임 통계 기반으로 쿼리 실행 계획을 동적으로 최적화하는 기능
- 주요 최적화: 동적 셔플 파티션 병합, 스큐 조인 최적화, 동적 조인 전략 전환

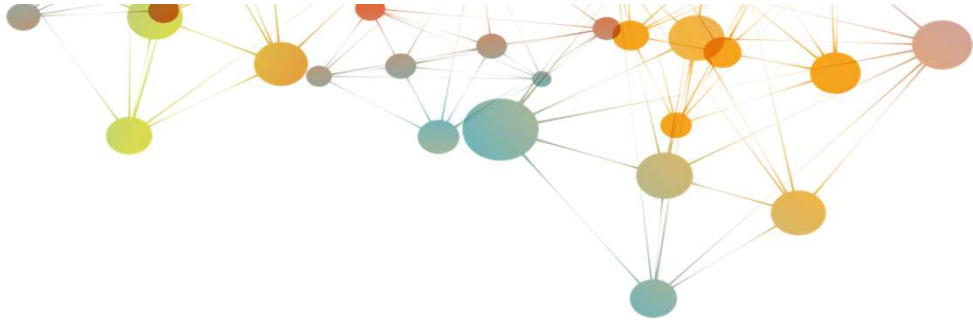
Dynamic Partition Pruning (DPP)

- 조인 연산 시 불필요한 파티션 스캔을 동적으로 제거하여 성능 향상
- 실행 시점에 필터 조건을 통해 읽어야 할 파티션을 결정하는 지능형 최적화

ANSI SQL 표준 지원

- SQL 표준 문법과 기능을 확장 지원하여 호환성 강화
- 향상된 타입 변환 규칙과 NULL 처리로 엔터프라이즈 SQL 호환성 개선

Adaptive Query Execution



AQE란?

- Spark 3.0에서 도입된 런타임 쿼리 최적화 기능으로, 실행 중에 수집된 통계 정보를 활용해 쿼리 플랜을 동적으로 조정
- 정적 계획이 아닌 실행 시간에 최적화함으로써 쿼리 성능 향상

AQE의 주요 기능

- 동적 셔플 파티션 통합: 작은 파티션들을 더 큰 파티션으로 병합하여 셔플 효율성 개선
- 조인 전략 전환: 런타임 통계에 따라 SortMerge Join에서 Broadcast Join으로 동적 전환
- 스큐 데이터 최적화: 데이터 치우침이 있는 파티션을 감지하고 분할하여 병렬 처리 효율 향상

활성화 방법

- Spark 3.0부터 기본적으로 활성화 (spark.sql.adaptive.enabled=true)
- 다양한 세부 설정을 통해 개별 최적화 기능 조정 가능

Dynamic Partition Pruning

개념

• Dynamic Partition Pruning(DPP)은 쿼리 실행 중 조인 조건에 따라 불필요한 파티션 스캔을 동적으로 제거하는 기술

• Spark 3.0에서 도입된 핵심 성능 최적화 기능 중 하나

작동 방식

• 조인 작업 시 필터링 조건을 파티션 테이블로 푸시다운하여 스캔 대상 파티션을 최소화

• 런타임에 작은 테이블의 결과를 먼저 계산하고, 그 결과를 사용하여 대용량 파티션 테이블의 필요한 파티션만 스캔

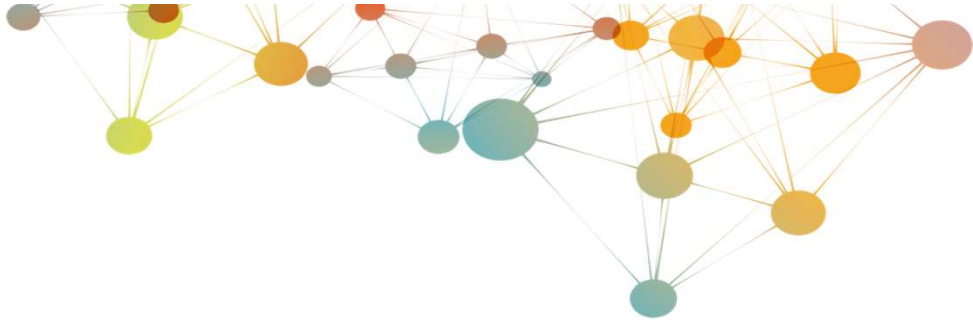
주요 이점

• I/O 비용 대폭 감소: 불필요한 데이터를 읽지 않아 디스크 I/O 최소화

• 쿼리 실행 시간 단축: 대규모 파티션 테이블 조인 작업에서 성능이 크게 향상 (최대 10배)

• 자동 활성화: spark.sql.optimizer.dynamicPartitionPruning.enabled=true (기본값)

데이터 레이크 통합



데이터 레이크 개념

- 데이터 레이크는 원시 형태의 대규모 데이터를 저장하고 분석하는 중앙 저장소 아키텍처
- 스키마 정의 없이 유연한 저장이 가능하지만, 메타데이터 관리가 중요한 과제로 부각

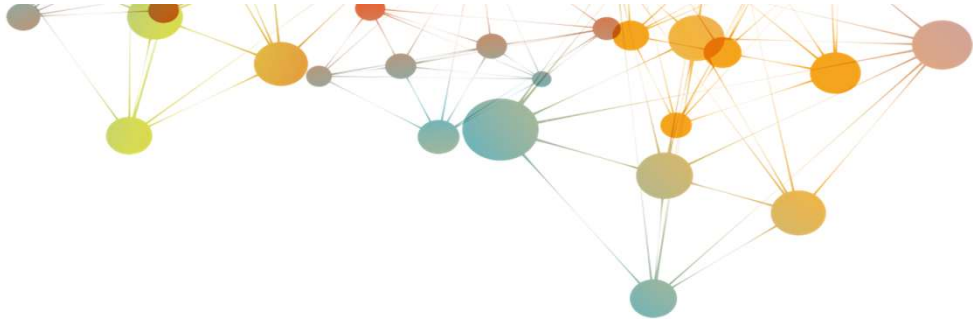
Spark v3의 데이터 레이크 통합 기능

- Delta Lake 지원: ACID 트랜잭션, 스키마 진화, 타임 트래블 기능 통합 및 최적화
- Apache Iceberg 지원: 대규모 테이블의 스키마 변경과 파티셔닝 관리를 위한 통합
- Catalog API 개선: 멀티 카탈로그 지원으로 다양한 메타스토어 연동 강화

주요 개선사항

- 파티션 관리: 동적 파티션 관리와 효율적인 메타데이터 처리 기능 강화
- 일관성 보장: 대규모 읽기/쓰기에서도 데이터 일관성 보장을 위한 최적화

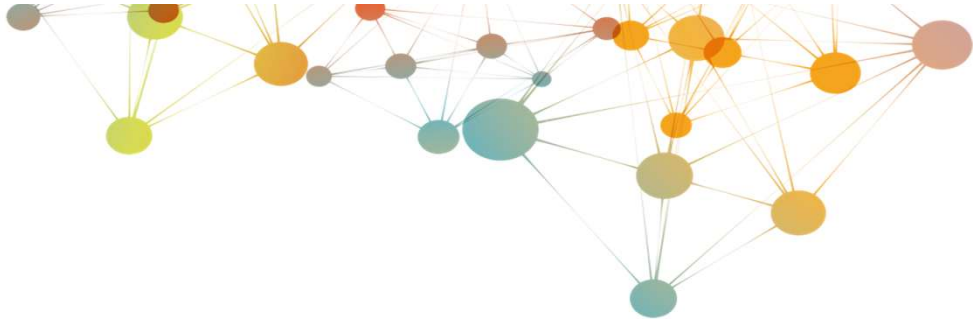
Section 6.



Apache Spark v4

Spark 4.x의 주요 혁신: Columnar Execution, Catalyst 엔진 고도화, 다양한 성능 향상 기능을 중심으로 살펴봅니다

Spark v4 핵심 기능



주요 혁신 기능

• Columnar Execution: Apache Arrow 기반의 컬럼 형식 처리로 20-50% 성능 향상과 메모리 효율성 개선

• Spark Connect: 서버-클라이언트 아키텍처로 분리하여 유연한 원격 API 연결 및 다중 프론트엔드 지원

• Catalyst 최적화 엔진 개선: 복잡한 조인과 셔플 연산의 성능 향상, 적응형 쿼리 실행(AQE) 고도화

추가 중요 개선사항

• SQL 기능 강화: ANSI SQL 호환성 확대, 복잡한 분석 함수 지원, 윈도우 함수 성능 향상

• Python 지원 향상: PySpark 성능 최적화, Pandas API 확장, Python UDF 가속화

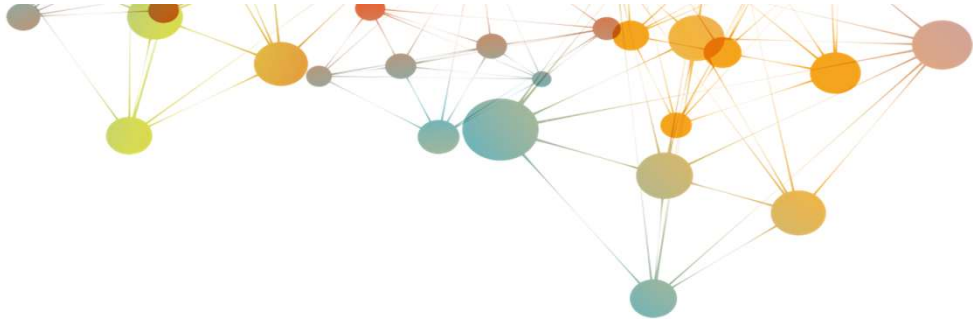
• 스트리밍 개선: 레이트 스트림 처리, 처리량 향상, 더욱 안정적인 체크포인팅 메커니즘

운영 및 배포 개선

• Kubernetes 통합 강화: 동적 리소스 할당, 오토스케일링, 운영 편의성 향상

• 모니터링 도구 개선: 상세 실행 메트릭, 향상된 웹 UI, 진단 기능 강화

Columnar Execution 개념



Columnar Execution이란?

• Spark 4.0에서 도입된 Apache Arrow 기반의 컬럼 지향 데이터 처리 방식

• 행 기반(Row-based) 방식이 아닌 열 기반(Column-based) 데이터 처리 패러다임 채택

Arrow 기반 컬럼형 엔진의 장점

• 메모리 효율성: 동일 타입 데이터 연속 저장으로 압축률과 캐시 효율성 향상

• 벡터화된 연산: CPU의 SIMD(Single Instruction Multiple Data) 명령어 활용

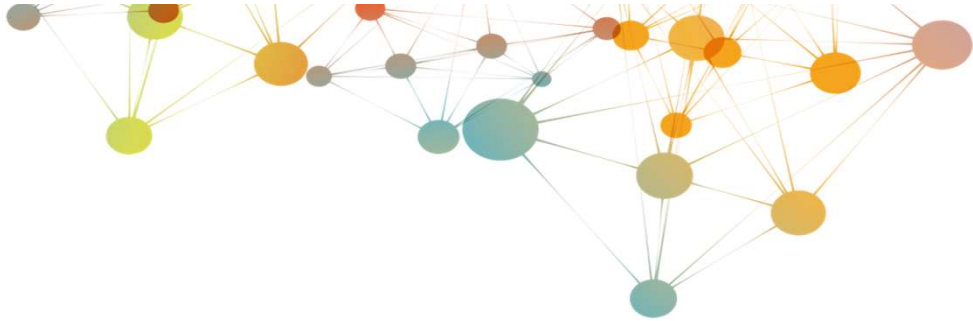
• 언어 간 효율적 전송: JVM과 Python/R 간 변환 오버헤드 대폭 감소

Spark 4.0 성능 개선

• 특히 PySpark 워크플로우에서 최대 20-50%의 성능 향상

• 분석 쿼리와 데이터 교환 시나리오에서 병목 현상 해소

Spark Connect란?



Spark Connect 개념

- Spark 4.0에서 도입된 클라이언트-서버 아키텍처 기반 원격 연결 프레임워크
- 데이터 처리 엔진과 클라이언트 애플리케이션을 완전히 분리하는 새로운 패러다임

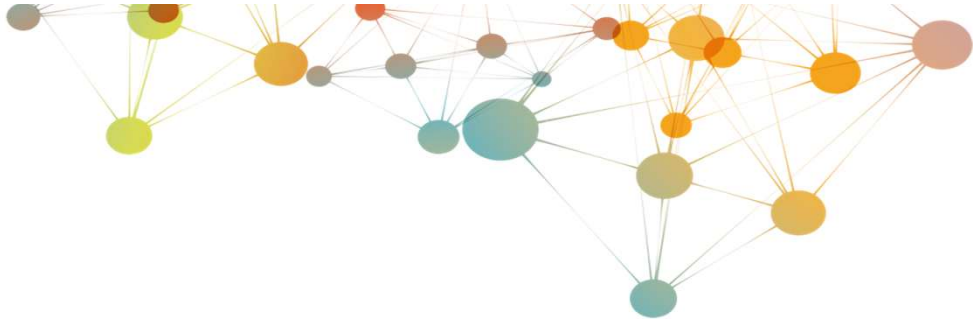
주요 기능 및 특징

- 유연한 API 연결 - 다양한 언어와 환경에서 원격 Spark 세션 접근 가능
- 리소스 격리 - 클라이언트와 Spark 클러스터 간 메모리/CPU 자원 분리
- 안정적인 연결 관리 - 네트워크 장애 대응 및 세션 복구 메커니즘 지원

지원하는 Frontend

- 다양한 언어 인터페이스 - Python, R, SQL, Java, Scala 등 지원
- 개발 환경 통합 - Jupyter, RStudio, VS Code 등 노트북/IDE 환경과 연동
- BI 도구 및 커스텀 애플리케이션 연동 인터페이스 제공

Catalyst 개선점



Spark v4에서의 Catalyst 옵티마이저 고도화

- 쿼리 계획 최적화: 복잡한 쿼리의 계획 생성 속도 향상 및 정교한 비용 모델 도입
- 조인 재정렬: 다중 테이블 조인에서 최적의 실행 순서 자동 결정 기능 강화

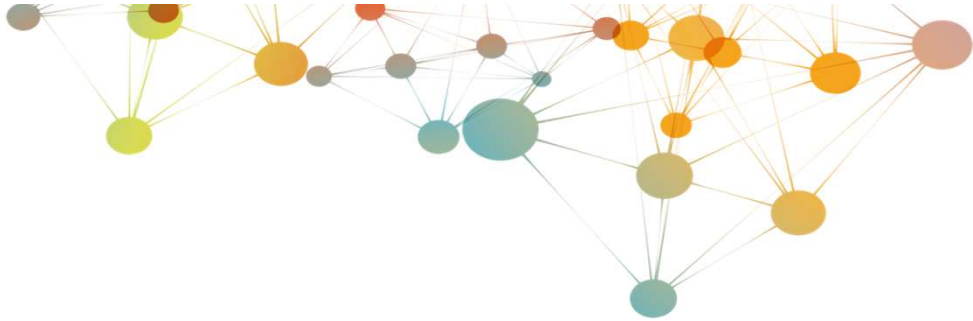
컬럼형 처리와의 통합

- Apache Arrow 통합: Catalyst와 컬럼형 실행 엔진 간의 긴밀한 통합으로 메모리 효율성 향상
- 벡터화 연산: CPU 캐시 친화적인 데이터 처리로 분석 쿼리 성능 20-30% 향상

성능 향상 포인트

- 코드 생성 최적화: 더 효율적인 런타임 코드 생성으로 CPU 사용률 개선
- 메모리 관리 개선: 오프힙 메모리 활용 최적화와 GC 부담 감소

PySpark, Pandas API on Spark 확장



Python 환경 주요 개선점

- Arrow 기반 컬럼 처리: Python UDF 실행 시 데이터 전송 병목 현상 대폭 감소
- Spark Connect를 통한 Python 클라이언트 격리로 안정성 및 확장성 개선
- JVM과 Python 프로세스 간 직렬화/역직렬화 성능 최적화

Pandas API on Spark 향상

- Pandas API 호환성 90% 이상으로 확대: 기존 Pandas 코드 재사용 용이
- 파이썬 네이티브 코드 실행 성능 향상: 이전 버전 대비 최대 40% 속도 개선
- 복잡한 그룹화 연산 및 윈도우 함수 지원 확대

사용자 경험 개선

- 디버깅 용이성: 개선된 에러 메시지와 트레이스백 정보 제공
- MLflow 통합 강화: Python 기반 ML 워크플로우 간소화

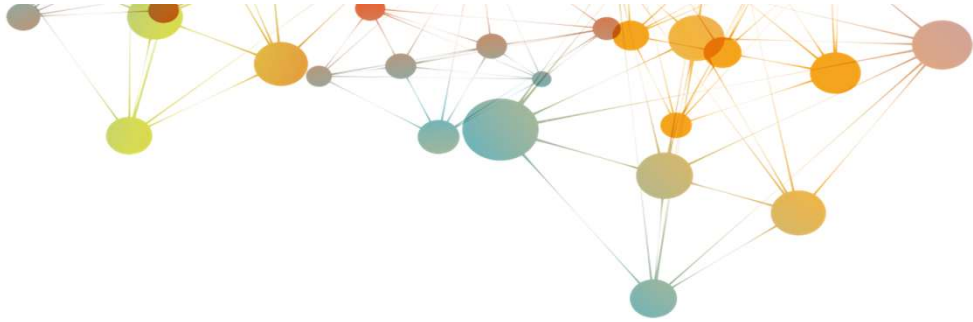
버전별 핵심 비교표

버전	출시 시기	대표 기능	핵심 키워드	성능
Spark v1 (1.0-1.6)	2014-2016	• RDD (Resilient Distributed Dataset) • Spark SQL • MLlib (머신러닝) • GraphX (그래프 처리)	• 분산 데이터 추상화 • Map-Reduce 패러다임 • 인메모리 처리	• Hadoop 대비 10-100배 성능 향상 • 인메모리 처리 기반
Spark v2 (2.0-2.4)	2016-2019	• DataFrame/Dataset API • Structured Streaming • Catalyst 옵티마이저 • Tungsten 실행 엔진	• 구조화된 데이터 처리 • 타입 안정성 • 코드 최적화	• Spark 1.x 대비 2-10배 성능 향상 • 메모리 사용 효율화

버전별 핵심 비교표

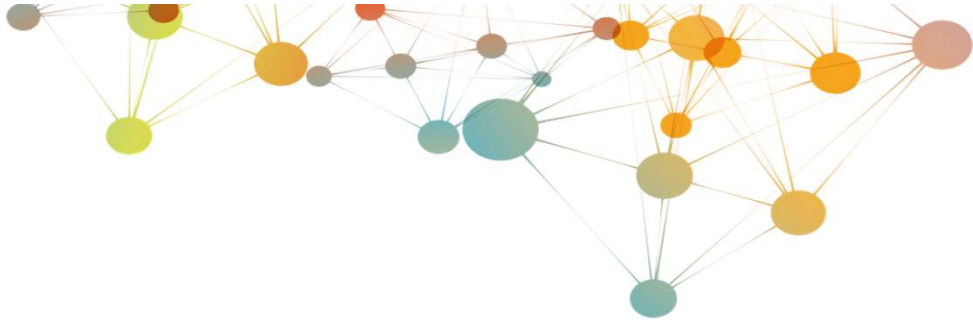
버전	출시 시기	대표 기능	핵심 키워드	성능
Spark v3 (3.0-3.5)	2020-2024	• Adaptive Query Execution (AQE) • Dynamic Partition Pruning (DPP) • ANSI SQL 표준 지원 • Python 타입 힌트 지원	• 런타임 쿼리 최적화 • 동적 파티션 관리 • SQL 표준화	• Spark 2.x 대비 2-3배 성능 향상 • 대규모 조인 연산 최적화
Spark v4 (4.0+)	2025-현재	• Columnar Execution • Spark Connect • 향상된 Catalyst 엔진 • 고도화된 Python API	• Arrow 기반 데이터 처리 • 클라이언트-서버 분리 • 통합 API	• Spark 3.x 대비 20-50% 성능 개선 • Python 워크로드 가속화

버전별 API 성능 비교표



Spark 버전	Scala API	Java API	Python API (PySpark)	비고
v0.9 (1.0 이전)	**1.0**	**0.65 - 0.75**	**0.40 - 0.50**	Java API 초기, Python 제한적
v1.0	**1.0**	**0.85 - 0.92**	**0.45 - 0.60**	Java API 안정화 시작
v2.0	**1.0**	**0.95 - 0.98**	**0.70 - 0.85**	Catalyst/Tungsten 최적화
v3.0	**1.0**	**0.97 - 1.0**	**0.80 - 0.90**	AQE, Pandas UDF 개선
v4.0	**1.0**	**0.98 - 1.0**	**0.85 - 0.95**	Arrow 통합, 성능 격차 최소화

Section 7.



Apache Spark의 활용

실제 Spark 환경에서의 다양한 활용 방법과 응용 사례를 살펴봅니다. 내장 유틸리티, Thrift Server, Livy Server, Airflow 등 활용 방안을 비교 분석합니다.

내장 유틸리티 활용 개요

Spark 내장 유틸리티란?

- Spark는 다양한 인터페이스를 통해 대화형 분석과 개발 환경을 제공
- 각 언어별 특화된 shell 환경으로 빠른 프로토타이핑과 데이터 탐색 가능

주요 내장 유틸리티

- spark-shell: Scala 기반 대화형 쉘, 강력한 타입 안정성과 함수형 프로그래밍 지원
- pyspark: Python 인터페이스, NumPy/Pandas 통합, 풍부한 라이브러리 생태계 활용 가능
- sparkR: R 언어 인터페이스, 통계 분석 및 시각화에 특화된 기능 제공

공통 기능

- 모든 유틸리티는 동일한 배포 옵션(--master, --deploy-mode) 지원
- 대화형(REPL) 환경으로 즉각적인 피드백과 반복적 분석 가능

spark-shell 옵션 소개

클러스터 설정 옵션

- --master: 클러스터 매니저 연결 URL 설정 (예: local[4], yarn, spark://host:7077)
- --deploy-mode: 드라이버 실행 위치 지정 (client 또는 cluster)

리소스 할당 옵션

- --executor-memory: 실행기당 메모리 할당 (예: 4g, 2048m)
- --driver-memory: 드라이버 프로세스 메모리 (예: 2g)
- --executor-cores: 실행기당 코어 수

환경 및 설정 옵션

- --conf: Spark 설정 속성 지정 (예: spark.sql.shuffle.partitions=100)
- --jars: 클러스터에 배포할 추가 JAR 파일
- --packages: Maven 좌표로 지정된 의존성

사용 예시

- 로컬 4코어 실행: spark-shell --master local[4] --driver-memory 2g
- YARN 클러스터 연결: spark-shell --master yarn --deploy-mode client --executor-memory 4g --executor-cores 2

Spark Thrift Server란?

개념 및 정의

• JDBC/ODBC 기반 SQL 엔진으로, 외부 시스템이 Spark SQL을 쉽게 활용할 수 있게 해주는 서비스

• Apache Hive의 HiveServer2를 기반으로 구축된 인터페이스로, Hive와 호환되는 API 제공

주요 기능

• 멀티 세션 지원: 여러 클라이언트가 동시에 연결하여 SQL 쿼리 실행 가능

• 장기 실행 서비스: 한 번 시작하면 여러 쿼리를 처리할 수 있도록 계속 실행됨

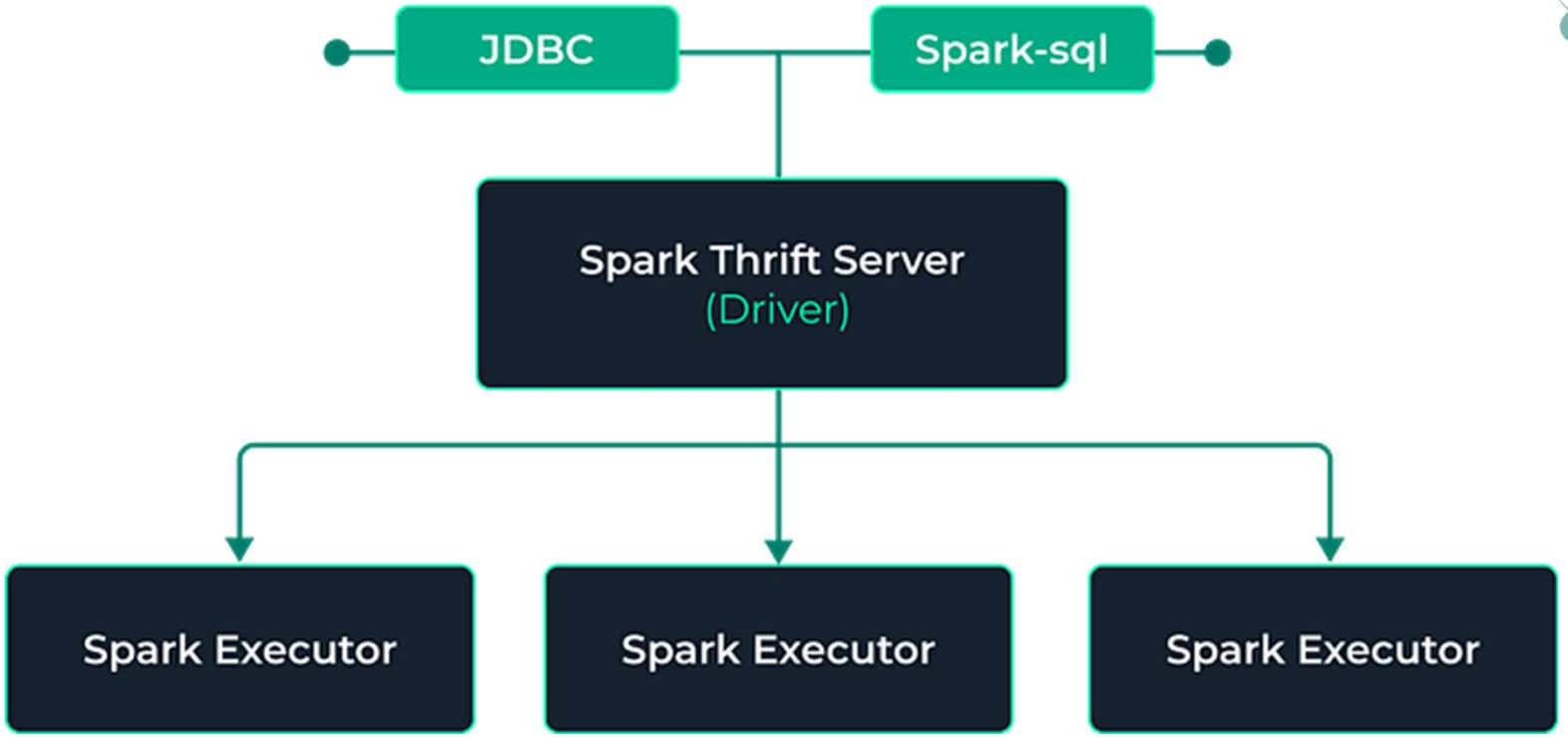
• BI 도구 연동: Tableau, Power BI 등 표준 JDBC/ODBC 인터페이스를 사용하는 도구와 연동 가능

활용

• 데이터 분석 환경: SQL 기반 대화형 분석 도구에서 Spark 성능 활용

• 레거시 시스템 통합: 기존 JDBC/ODBC 기반 애플리케이션을 Spark에 연결

Spark Thrift Server 아키텍처



Data Streams

Spark Thrift Server 주요 설정

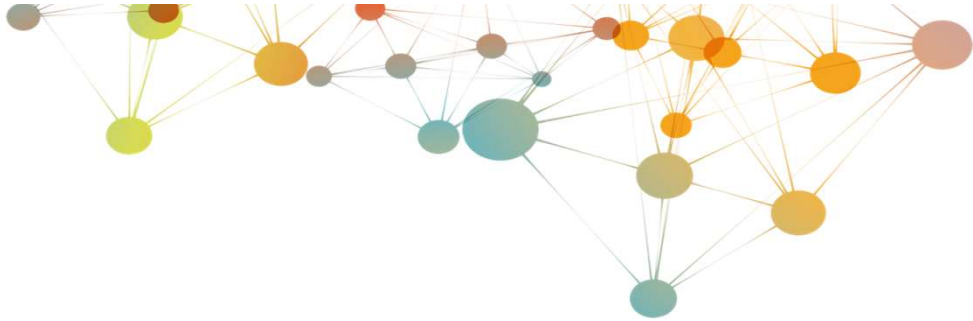
start-thriftserver.sh 스크립트 옵션

- 기본 실행 방법: ./sbin/start-thriftserver.sh
- 포트 지정: --hiveconf hive.server2.thrift.port=10001 (기본값: 10000)
- 호스트 지정: --hiveconf hive.server2.thrift.bind.host=hostname

hive-site.xml 주요 설정

- 전송 모드: hive.server2.transport.mode - binary 또는 http
- HTTP 모드 설정: hive.server2.thrift.http.port 및 hive.server2.http.endpoint
- 인증 설정: hive.server2.authentication - NONE, LDAP, KERBEROS, CUSTOM
- 메타스토어 연결: hive.metastore.uris - 외부 Hive 메타스토어 연결 시 사용

Thrift Server 연동 가능 BI



BI 도구 연동 개요

- Spark Thrift Server는 JDBC/ODBC 프로토콜을 통해 다양한 BI(Business Intelligence) 도구와 연결 가능
- 데이터 과학자와 분석가가 익숙한 인터페이스로 대규모 Spark 데이터를 시각화하고 분석할 수 있음

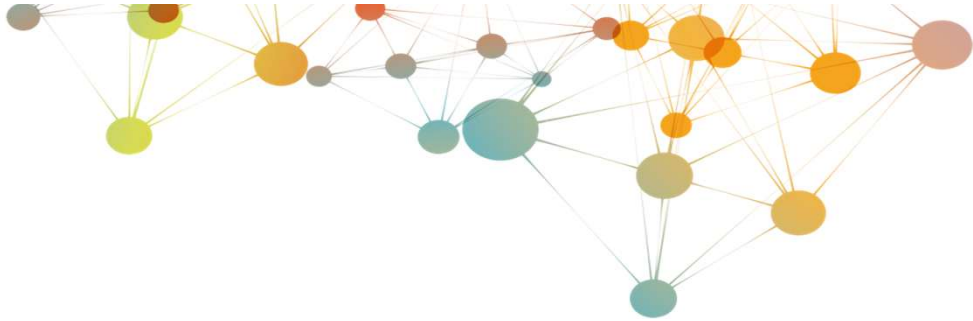
주요 연동 BI 도구

- Tableau: JDBC 드라이버를 통한 연결, 드래그-앤-드롭 인터페이스로 Spark 데이터를 시각화
- Power BI: Microsoft의 BI 도구, ODBC 연결을 통해 대시보드 및 보고서 생성 지원
- Excel: ODBC 데이터 소스로 연결하여 피벗 테이블, 차트 작성 가능
- 기타 도구: MicroStrategy, QlikView, Looker 등 JDBC/ODBC 지원 BI 도구

연동의 이점

- 전문적 시각화: 복잡한 코딩 없이도 전문적인 데이터 시각화 가능
- 셀프 서비스 분석: 비 개발자도 빅데이터를 직접 분석할 수 있는 환경 제공

Spark Livy Server란?



Livy Server 개념

• Apache Spark 클러스터와의 상호작용을 위한 REST 인터페이스를 제공하는 서비스

• Spark 작업의 원격 제출 및 관리를 웹 API를 통해 간편하게 처리

주요 기능

• 인터랙티브 세션: Scala, Python, R, SQL 쉘 세션 생성 및 코드 실행

• 배치 작업: JAR, Python 스크립트 등을 클러스터에 제출 및 모니터링

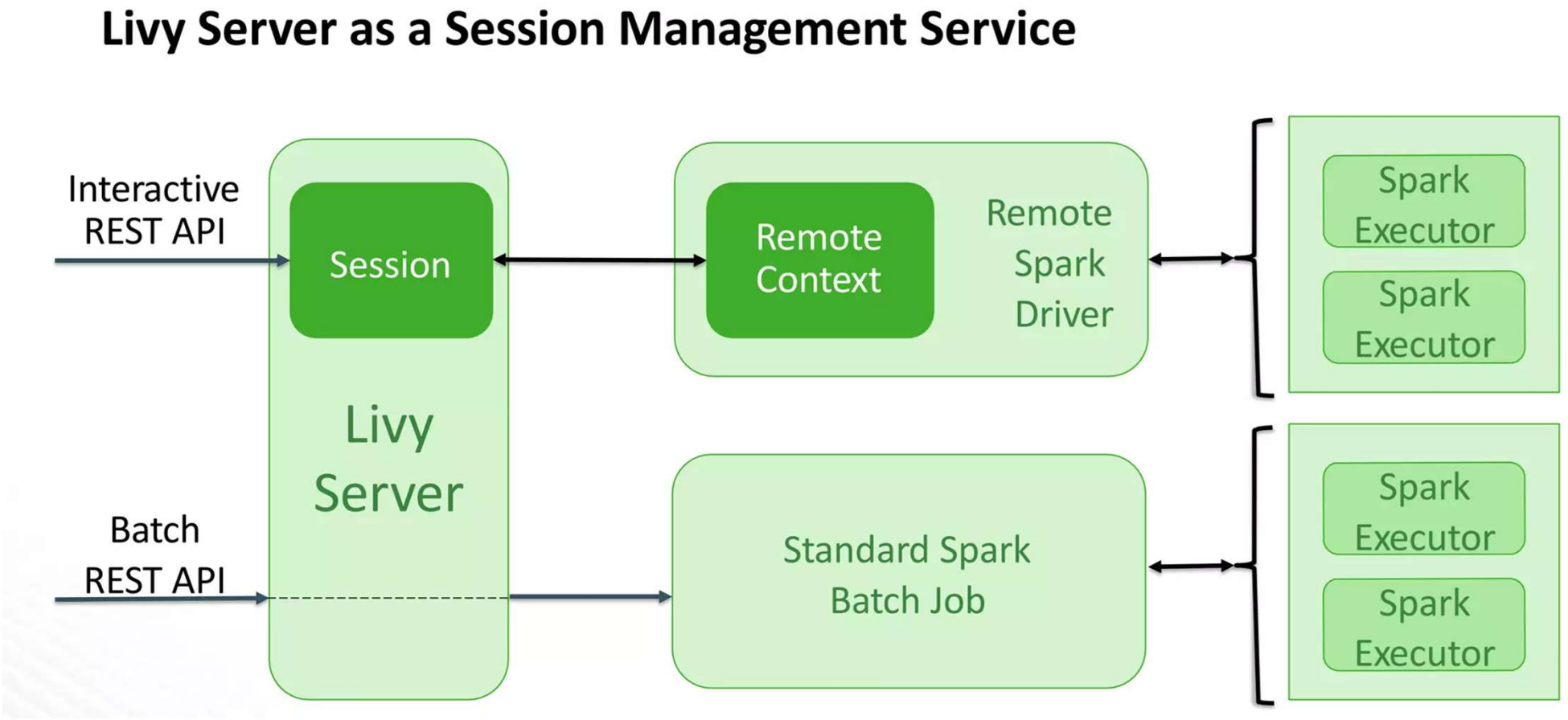
• 다중 사용자 지원: 여러 클라이언트가 동시에 하나의 Spark 클러스터 공유

활용

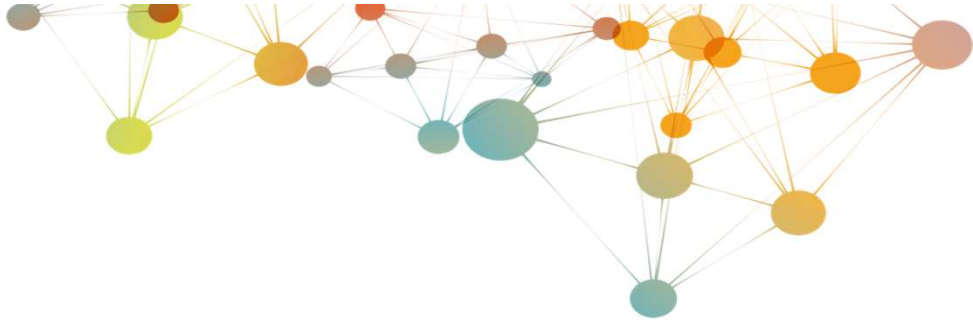
• 웹 애플리케이션, 대시보드에서 언어 독립적 Spark 연동 가능

• Spark 코드를 포함하지 않는 서비스에서도 REST API로 Spark 기능 활용

Livy Server 아키텍처 설명



Livy Server 설정 예시



REST API 기본 엔드포인트

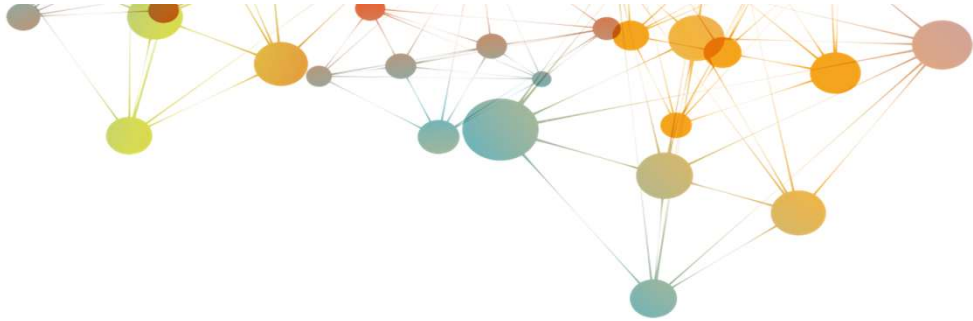
- 세션 관리: /sessions, /sessions/{sessionId}
- 명령어 실행: /sessions/{sessionId}/statements
- 배치 작업: /batches, /batches/{batchId}

세션 생성 예시

```
curl -X POST -H "Content-Type: application/json" \
-d '{"kind": "pyspark", "numExecutors": 4, "executorCores": 2,
"executorMemory": "2g", "driverMemory": "2g"}' \
http://<livy-server>:8998/sessions
```

세션 관리 작업

- 세션 목록 조회: GET /sessions
- 세션 상태 확인: GET /sessions/{sessionId}/state
- 세션 종료: DELETE /sessions/{sessionId}



Livy 연동 가능 솔루션

Jupyter Notebook/Lab

• Sparkmagic 확장을 통해 Livy REST API 연결 지원

• 대화형 개발 환경에서 원격 Spark 클러스터를 Livy를 통해 활용 가능

Apache Zeppelin

• Livy 인터프리터 통해 Spark 작업 원격 제출 및 관리

• 다양한 언어 지원: Scala, Python, R 등 한 노트북에서 혼합 사용

Hue

• Cloudera의 웹 기반 인터페이스로, Livy를 통한 Spark SQL 쿼리 실행 지원

• 데이터 브라우저, 노트북, 워크플로우 편집기 등 통합 환경 제공

기타 연동 가능 도구

• RStudio: sparklyr 패키지로 Livy 연결 지원

• Visual Studio Code: Spark & Hive Tools 확장으로 Livy 연동

Apache Airflow

Airflow?

- DAG(Directed Acyclic Graph) 기반의 워크플로우 오케스트레이션 오픈소스
- 워크플로우 작성, 스케줄링, 모니터링 통합 플랫폼

주요 기능

- 워크플로우: 각 작업의 관계(순서, 의존성)을 정의
- 스케줄링: 다양한 주기로 정의된 작업 실행(ex> cron)
- 모니터링: 자체 Web UI를 통하여 작업 관리(상태 모니터링, 시작, 재시작 등)
- 실행: 작업 인터페이스 제공(BashOperator, PythonOperator, SparkSubmitOperator 등)

활용

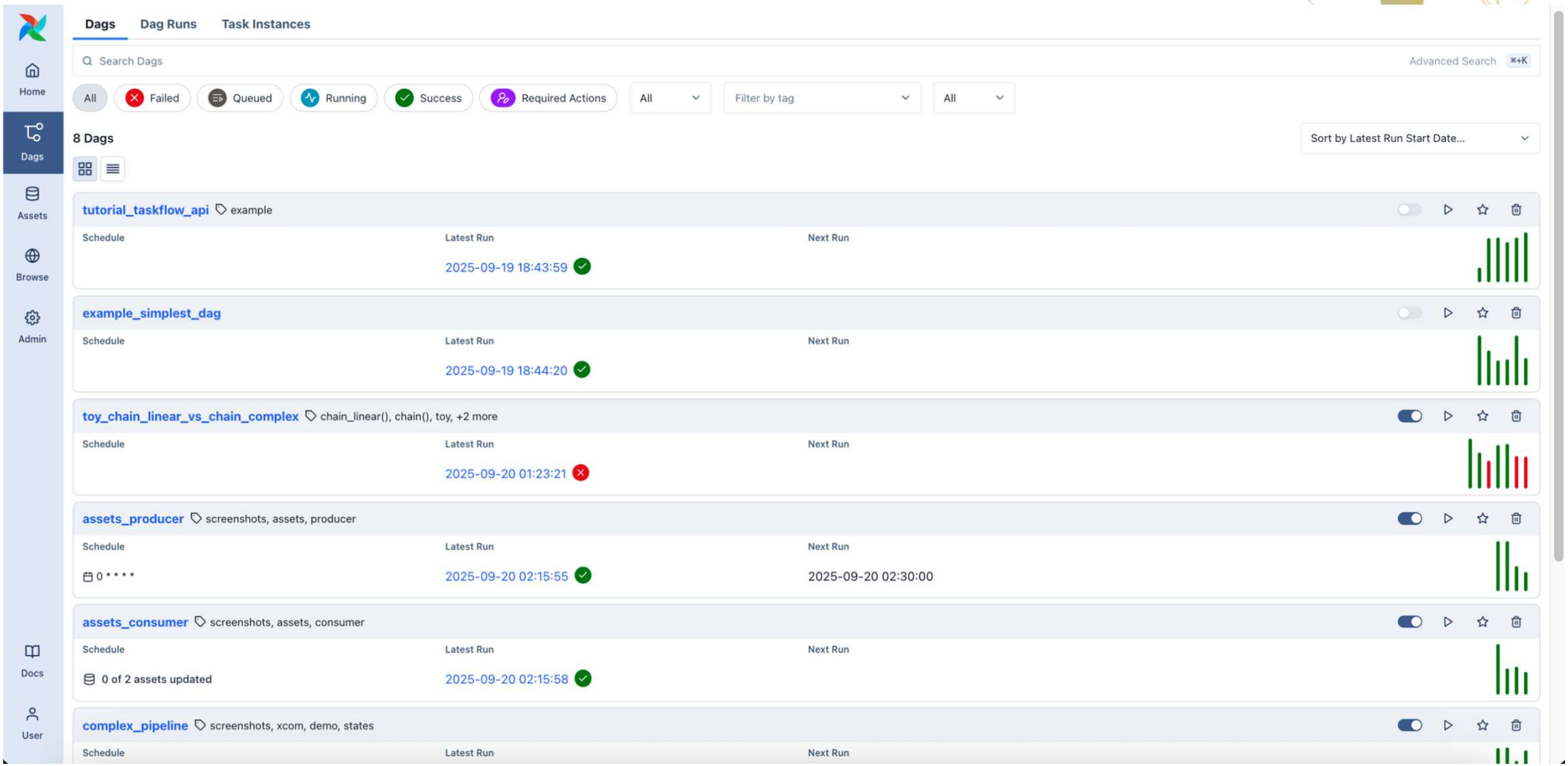
- 대량의 배치 작업 스케줄러로 많이 사용
- 특히 SparkJDBCOperator, SparkSQLOperator, SparkSubmitOperator 제공으로 Apache Spark 배치 작업과의 궁합이 최적화 되어 있음

Apache Airflow – Operator 예제

```python
from airflow import DAG
from airflow.providers.apache.spark.operators.spark_submit import SparkSubmitOperator
from datetime import datetime

# DAG 정의
default_args = {
    'owner': 'airflow',
    'start_date': datetime(2025, 10, 21),
    'retries': 1,
}

with DAG(
    dag_id='spark_submit_example',
    default_args=default_args,
    schedule_interval='0 3 * * *',  # 매일 새벽 3시 실행 (cron 형식)
    catchup=False,
    description='Airflow Spark Submit Operator Example',
) as dag:

    spark_submit_task = SparkSubmitOperator(
        task_id='spark_pi_job',
        application='/opt/spark/examples/src/main/python/pi.py',  # Spark 작업 경로
        conn_id='spark_default',  # Airflow connection 설정 필요
        verbose=True,
        conf={
            "spark.executor.memory": "2g",
            "spark.executor.cores": "2",
            "spark.driver.memory": "1g"
        },
        name='airflow_spark_pi',
        application_args=['1000'],  # Spark 애플리케이션 인자
        execution_timeout=None,
    )
```

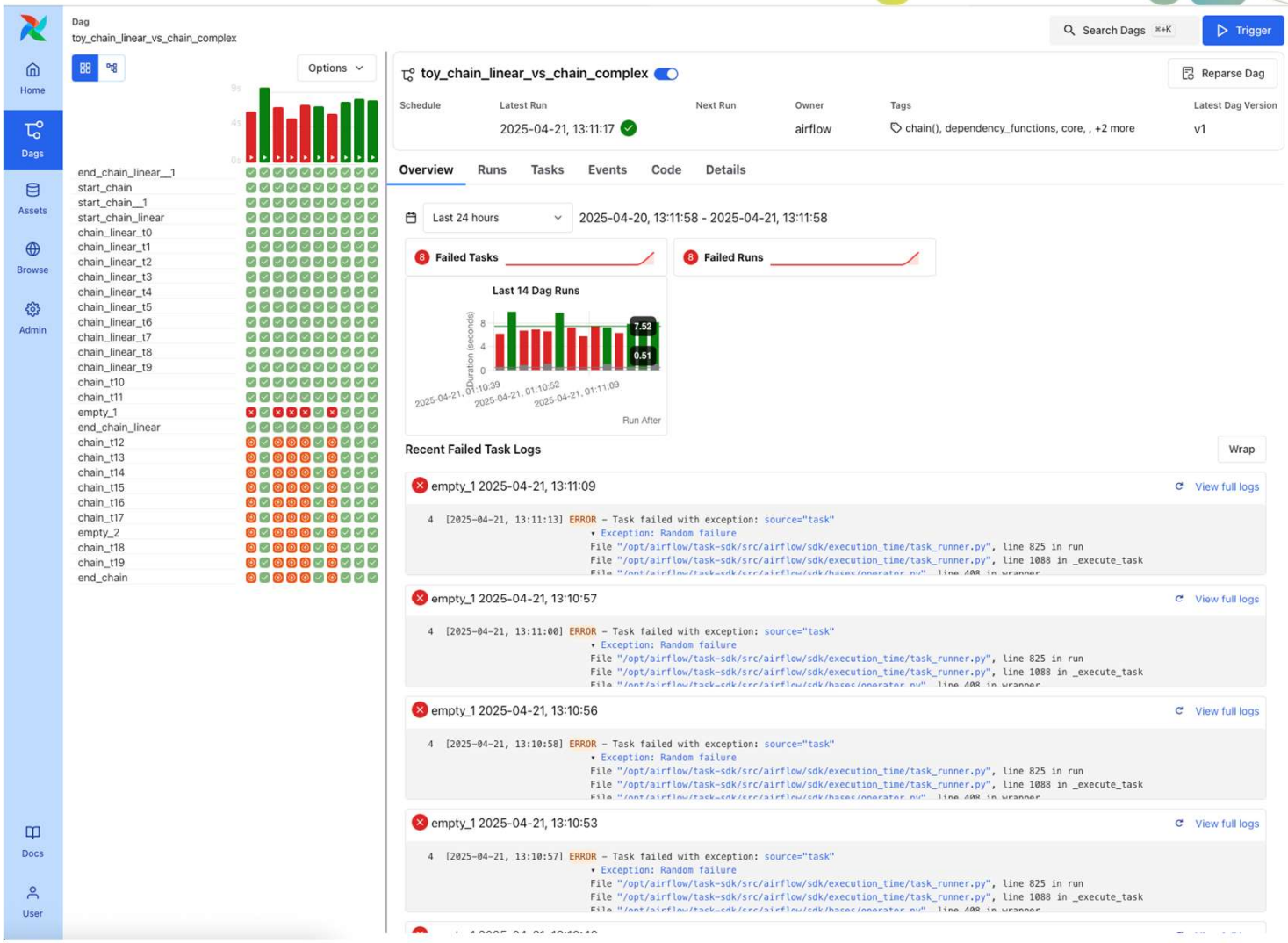

Data Streams

Apache Airflow – Web UI



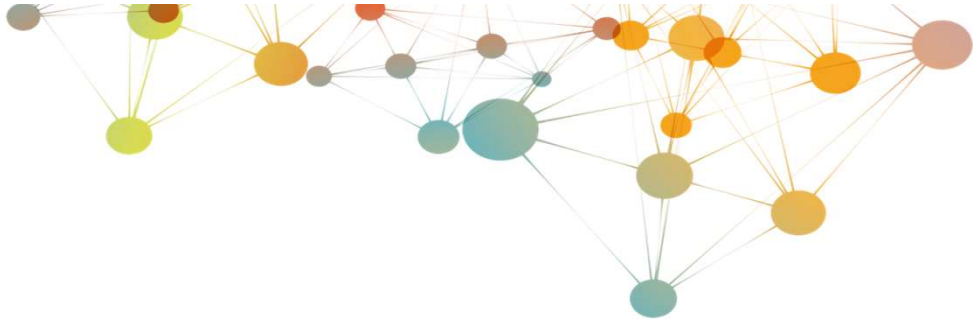
Data Streams

Apache Airflow – Web UI



Data Streams

Section 8.



요약 및 정리

Apache Spark 아키텍처, 버전별 특징, 그리고 다양한 활용 방식에 대한 핵심 내용을 정리합니다

Spark 아키텍처 핵심 요약

작업 배포 모드

- Local Mode: 단일 머신에서 모든 컴포넌트 실행, 개발 및 테스트에 적합
- Client Mode: Driver는 로컬, Executor는 클러스터에서 실행, 대화형 분석에 최적화
- Cluster Mode: Driver와 Executor 모두 클러스터에서 실행, 프로덕션 환경 적합

리소스 관리 방식

- Standalone: Spark 자체 클러스터 관리자, 간단한 구성과 독립적 운영
- YARN: Hadoop 생태계 통합, 리소스 공유와 다중 워크로드 지원
- Kubernetes: 컨테이너 기반 오케스트레이션, 클라우드 네이티브 환경 최적화

핵심 아키텍처 요소

- Driver: 애플리케이션 실행 흐름 제어, SparkContext 생성, 작업 계획 및 조율
- Executor: 작업 실행 주체, 데이터 처리 및 캐싱, 복원성 제공
- Cluster Manager: 리소스 할당, 자원 관리, 노드 모니터링

Spark 버전별 혁신 요약

Spark v1 (2014-2016)

- RDD(Resilient Distributed Dataset): 분산 데이터의 기본 추상화 모델 도입
- Spark SQL: 구조화된 데이터 처리와 SQL 인터페이스 지원
- MLlib & GraphX: 머신러닝과 그래프 처리 라이브러리 기반 마련

Spark v2 (2016-2019)

- DataFrame & Dataset API: 타입 안정성과 최적화된 실행 플랜 제공
- Catalyst Optimizer: SQL 및 DataFrame 질의 최적화 엔진 도입
- Structured Streaming: 일관된 API로 배치/스트리밍 통합 처리 지원

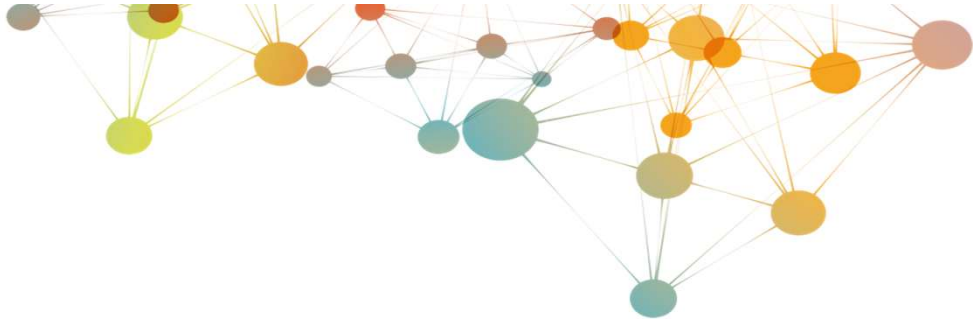
Spark v3 (2020-2024)

- Adaptive Query Execution(AQE): 런타임 통계 기반 쿼리 동적 최적화
- Dynamic Partition Pruning(DPP): 조인 시 불필요한 파티션 자동 제거
- Python 지원 강화: Pandas API 통합 및 성능 개선

Spark v4 (2025~)

- Columnar Execution: Arrow 기반 컬럼형 실행으로 성능 20-50% 향상
- Spark Connect: 분리된 클라이언트-서버 아키텍처로 유연한 연결성 확보
- 클라우드 네이티브: Kubernetes 통합 강화 및 컨테이너 최적화

적합 활용법 및 미래 전망



실무 적용 시 고려사항

- 활용 목적에 맞는 배포 모드 선택: 개발/테스트에는 Local, 대화형 분석에는 Client, 프로덕션에는 Cluster 모드
- 리소스 관리 전략: 데이터 크기, 처리 복잡성, SLA에 따른 메모리, CPU 코어 최적 할당
- 버전 선택 기준: 기존 시스템과의 통합성, 필요한 기능, 성능 요구사항을 종합적으로 고려

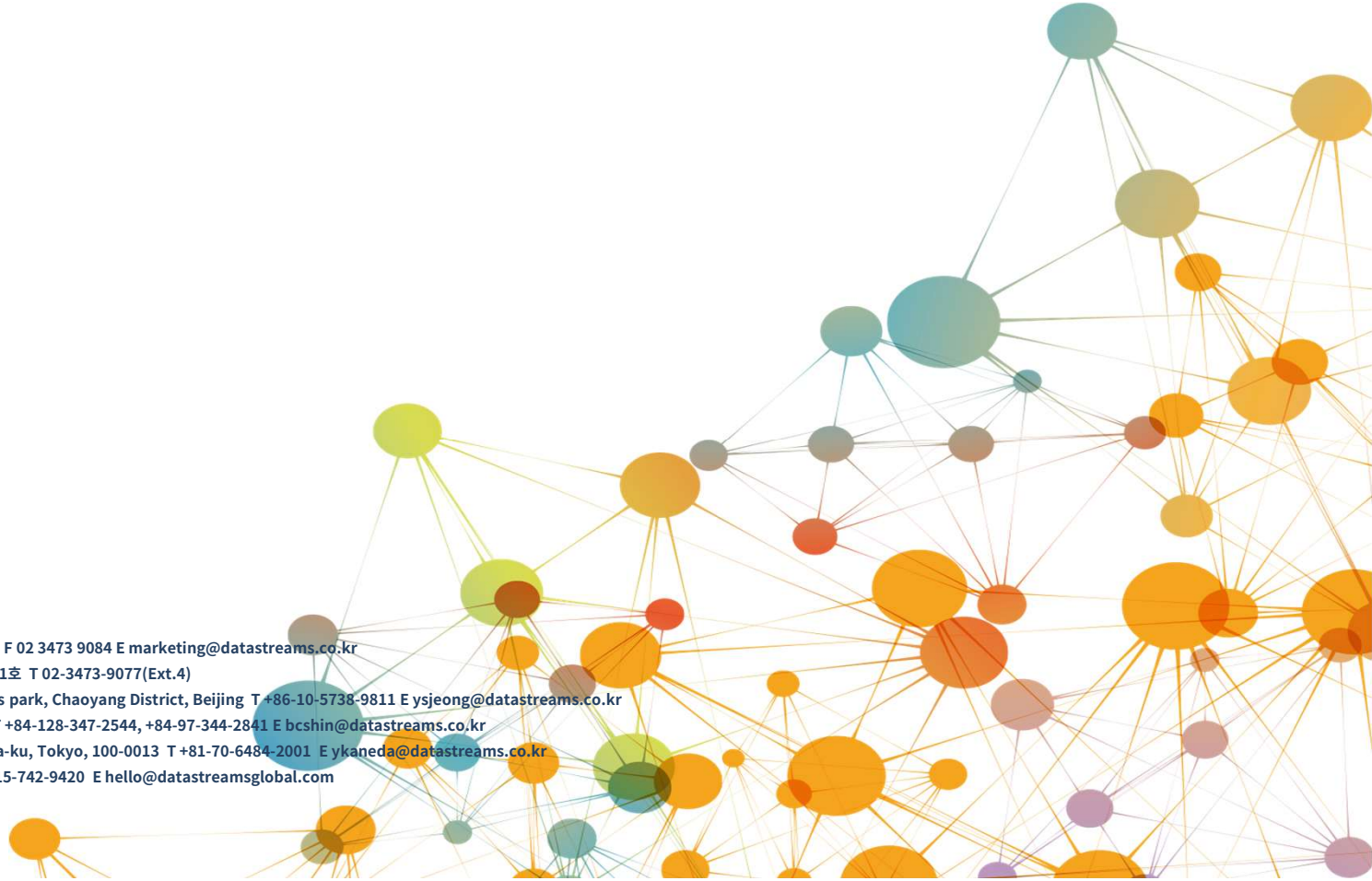
산업 동향 및 미래 방향

- 클라우드 네이티브 통합 강화: AWS EMR, Azure Synapse, GCP Dataproc과의 완전한 통합
- 실시간 처리의 진화: 배치 중심에서 스트리밍 처리 중심으로 패러다임 전환
- AI/ML과의 융합 가속화: 딥러닝 프레임워크와의 통합, 분산 학습 최적화

권장 적용 전략

- 비용 효율적 설계: 오토스케일링, 임시 클러스터, 적절한 인스턴스 타입 선택
- 최신 버전 활용: AQE, DPP, Columnar Execution 등 성능 향상 기능 활용
- 데이터 레이크 아키텍처 연계: Delta Lake, Iceberg와 같은 현대적 레이크 포맷 통합

Thank you !



(주)데이터스트림즈 본사 서울시 서초구 사임당로 28 청호나이스빌딩 6층 T 02 3473 9077 F 02 3473 9084 E marketing@datastreams.co.kr
(주)데이터스트림즈 R&D센터 경기도 성남시 분당구 대왕판교로 670 유스페이스몰2 B동 601호 T 02-3473-9077(Ext.4)
DataStreams China 100-102 Pohang center 28F, Wangjing technology business park, Chaoyang District, Beijing T +86-10-5738-9811 E ysjeong@datastreams.co.kr
DataStreams Vietnam 1806, CMC Building, Duy Tan St., Cau Giay Dist, Hanoi T +84-128-347-2544, +84-97-344-2841 E bcshin@datastreams.co.kr
DataStreams Japan 18F, Shinkasumigaseki Bldg., 3-3-2 Kasumigaseki Chiyoda-ku, Tokyo, 100-0013 T +81-70-6484-2001 E ykaneda@datastreams.co.kr
DataStreams USA Contact 1229 2nd Avenue, San Francisco, CA 94122 T +1-415-742-9420 E hello@datastreamsglobal.com