Data Streams

고급데이터베이스 6주차

Spark의 이해

데이터스트림즈 기술연구소
길기범 수석연구원

목차

Part 1: Spark의 탄생과 기본 개념

Spark: Cluster Computing with Working Sets(2010)

1. 강의 개요 및 배경
2. 저자 및 연구 배경 소개
3. MapReduce의 한계와 문제 정의
4. Spark의 핵심 아이디어와 RDD 초기 개념
5. Spark 프로그래밍 모델
6. 구체적 예제를 통한 이해
7. 초기 성능 평가 결과

Part 2: RDD 고급 개념

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing(2012)

1. 2012년 논문 개요 및 발전 사항
2. 저자진 확장과 연구 심화
3. RDD의 발전된 개념과 이론적 기반
4. 혁신적인 내결함성 메커니즘
5. 고급 성능 최적화 기법
6. 확장된 실험 결과와 검증
7. 실제 적용 사례와 영향

Part 1 - Spark의 탄생과 기본 개념

Spark이 왜, 어떻게 탄생했는지 살펴보고 기본 구조와 동작 원리를 이해합니다.

MapReduce의 한계와 2010년 당시 빅데이터 환경

Spark의 핵심 혁신: RDD와 메모리 기반 처리

데이터 처리 속도 향상과 반복/대화형 작업 지원

Spark 논문 저자 소개

2010년 논문 "Spark: Cluster Computing with Working Sets"의 저자들은 이후 빅데이터 생태계를 선도하는 인물들이 되었습니다.

- **Matei Zaharia**
  - UC Berkeley 박사과정 학생(당시 24세), 현재 Databricks CTO 이자 Stanford 교수
- **Mosharaf Chowdhury**
  - 현 미시간 대학교 교수, 네트워크 시스템 전문가
- **Michael J. Franklin**
  - UC Berkeley 교수, 현 시카고 대학 컴퓨터과학과 학장
- **Scott Shenker&Ion Stoica**
  - UC Berkeley 교수진, Databricks 공동 창업자

Matei Zaharia

Spark 프로젝트 창시자

UC Berkeley AMPLab

2010년대 빅데이터 연구의 메카

AMPLab에서 시작되어 산업계를 혁신한 프로젝트

AMPLab와 Spark 개발환경

UC Berkeley의 AMPLab에서 시작된 Apache Spark는 혁신적인 분산 컴퓨팅 시스템의 새로운 패러다임을 제시했습니다.

- **AMPLab**
  - 2009년 설립된 UC Berkeley의 빅데이터 연구소 (Algorithms, Machines, and People Laboratory)
- 빅데이터 처리의 **새로운 패러다임**을 연구하는 학제간 연구그룹
- Hadoop MapReduce의 한계를 해결하기 위한 **실용적 접근법** 추구
- Scala 언어 기반으로 개발, **함수형 프로그래밍**의 장점 활용
- 산학연 협력을 통한 **실질적 문제 해결**에 초점

UC Berkeley AMPLab - Spark의 탄생지

Spark은 무엇인가?

Apache Spark는 대규모 데이터 처리를 위한 통합 분석 엔진으로, 속도와 사용 편의성, 다양한 분석 기능을 모두 갖춘 차세대 빅데이터 프레임워크입니다.

- **메모리 기반 처리**를 통한 Hadoop 대비 100배 이상의 속도 향상
- 다양한 언어 지원 (Scala, Java, Python, R) 및 직관적인 API
- 하나의 엔진으로 **배치, 스트리밍, 머신러닝, SQL 쿼리** 처리 가능
- 기존 Hadoop 생태계와의 뛰어난 호환성 (HDFS, Hive 등)

Apache Spark 로고 - 빠른 속도와 확장성을 상징

빅데이터 환경의 변화(2010년)

2010년 당시 빅데이터 환경은 Hadoop/MapReduce가 지배적이었으나, 새로운 분석 요구사항들이 등장하며 변화의 조짐이 있었습니다.

- **Hadoop**의 폭발적 성장 - Yahoo, Facebook 등 대규모 채택
- 하드웨어 변화 - 메모리 가격 하락, 멀티코어 확산
- 분석 패러다임 변화 - **실시간 분석**과 **반복적 알고리즘** 수요 급증
- 기존 배치 중심 모델의 한계 직면

2010년 빅데이터 생태계는 변화의 기로에 서 있었습니다

MapReduce의 한계 1: 반복적 처리

머신러닝 알고리즘과 같은 반복적 계산 작업에서 MapReduce는 심각한 효율성 문제를 보입니다.

- 머신러닝 알고리즘(K-means, Logistic Regression 등)은 **동일 데이터에 대한 반복 작업**이 필수적
- MapReduce는 **매 반복마다 디스크에서 데이터를 재로드**해야 함
- 데이터 크기가 TB 단위라면 매 반복마다 수십~수백 초의 I/O 지연 발생
- 반복 횟수가 많아질수록 **디스크 I/O 비용이 기하급수적으로 증가**

MapReduce의 반복적 작업에서 매 반복마다 발생하는 디스크 I/O

MapReduce의 한계 2: 대화형 분석

MapReduce는 배치 처리에 최적화되어 있어 실시간/대화형 데이터 분석에 근본적인 한계가 있었습니다.

- 각 쿼리마다 **새로운 작업을 시작**해야 하는 부담
- 사용자 쿼리당 **수십~수백초**의 지연 시간
- 매 쿼리마다 **디스크 I/O 병목** 현상 발생
- 데이터 과학자의 **탐색적 분석**에 부적합한 응답성

쿼리 응답 시간 비교

MapReduce의 대화형 분석 성능 한계

Spark이 해결한 점

메모리 중심의 작업 처리로 반복/대화형 데이터 분석을 빠르게 제공하고, RDD 개념의 도입으로 성능과 개발 편의성을 크게 향상시켰습니다.

메모리 기반 처리로 디스크 I/O 병목 현상 해결

반복 알고리즘 실행 시 최대 10배 성능 향상

인터랙티브 쿼리와 대화형 데이터 분석 지원

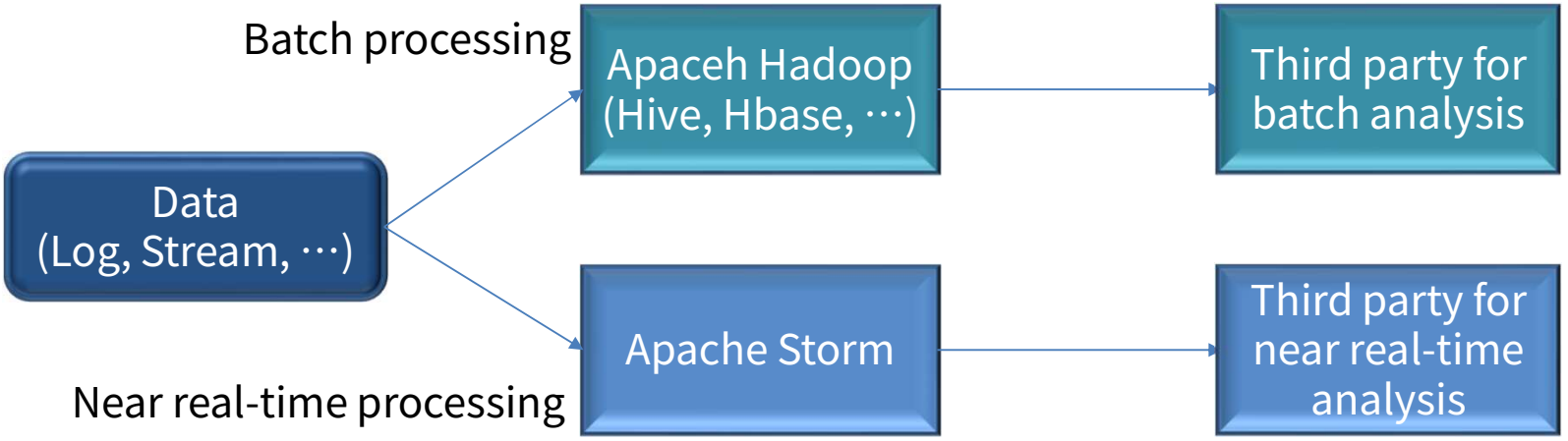
Spark의 출현 배경 요약

2010년대 빅데이터 처리 환경에서 Hadoop/MapReduce의 성공과 한계를 바탕으로 Spark가 등장하게 된 배경을 살펴봅니다.

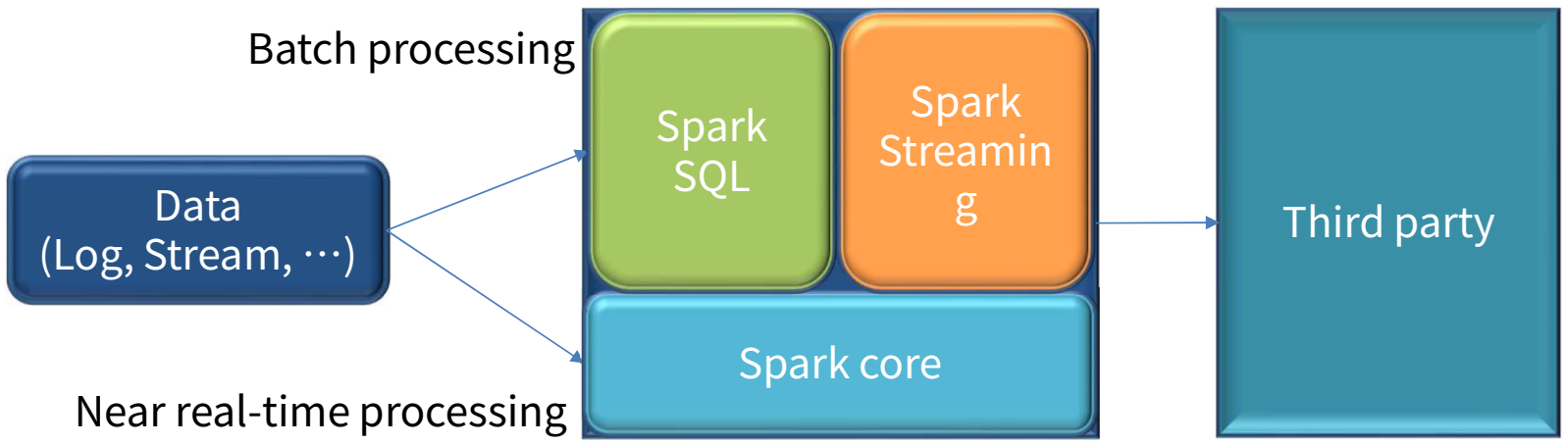
- Hadoop의 **성공 경험**과 대용량 배치 처리의 한계
- **반복적 연산** 작업 수요의 폭발적 증가
- 디스크 I/O 중심 모델의 성능 한계 인식
- **'Working Set'** 문제 해결의 필요성 대두

Spark 탄생의 역사적 맥락

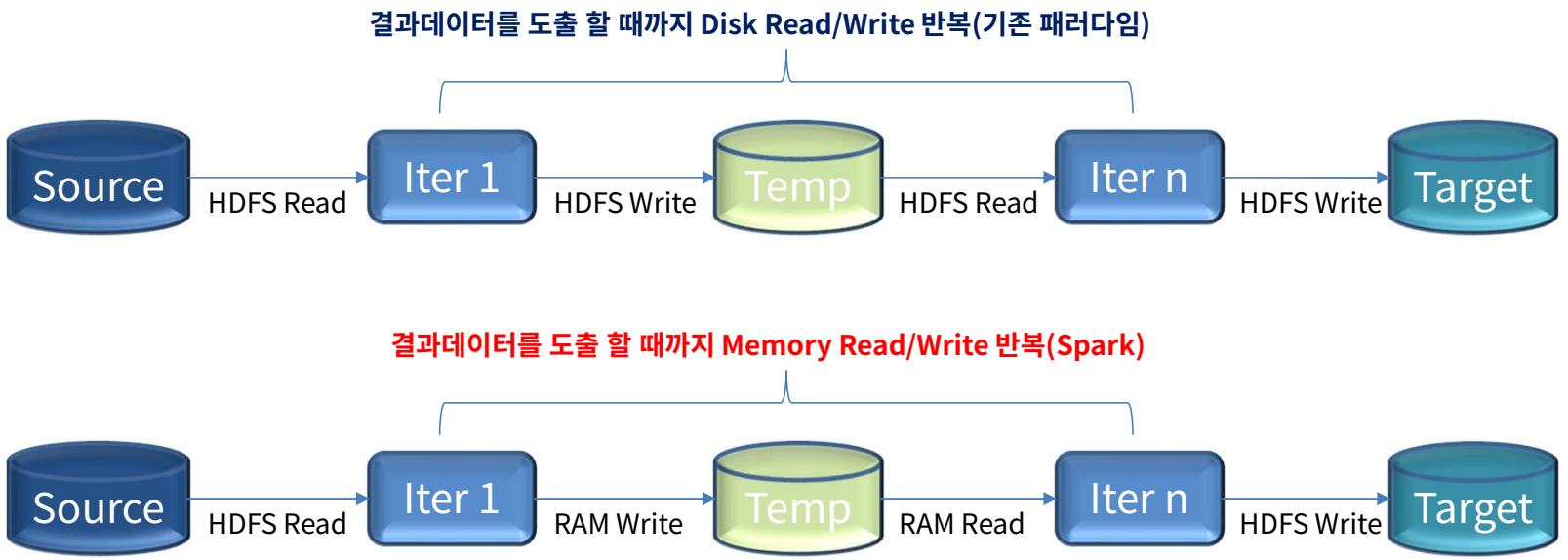
Spark의 출현 배경 – 업무 관점



Spark의 출현 배경 – 업무 관점



Spark의 출현 배경 – 성능 관점



Spark의 혁신적 철학

Spark은 기존 MapReduce의 한계를 넘어, 빅데이터 처리를 위한 혁신적인 패러다임을 제시했습니다. 빠른 성능과 유연성, 그리고 사용 편의성을 모두 갖춘 프레임워크입니다.

"Working Set" 개념 - 메모리에 데이터 캐싱으로 반복 접근 최적화

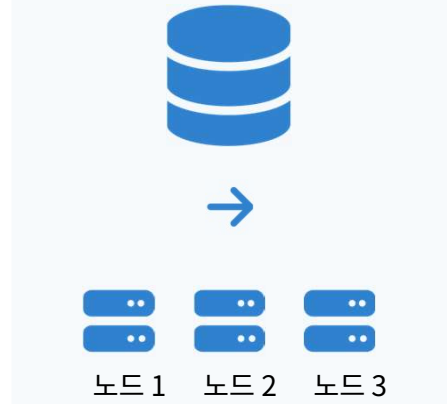
메모리 중심 아키텍처 - 디스크 I/O 최소화로 속도 혁신

내결함성과 확장성 - 장애 상황에서도 안정적 작동

RDD의 기본 개념

Resilient Distributed Dataset은 Spark의 핵심 추상화 개념으로,
클러스터 전체에 분산된 불변의 데이터 컬렉션입니다.

- **Dataset(데이터셋)** - 파일, 파라미터, 변환 결과 등 다양한 데이터
- **Resilient(복원력)** - 노드 장애 시에도 데이터 손실 없이 복구 가능
- **Distributed(분산)** - 클러스터의 여러 노드에 걸쳐 데이터 분산 저장
- **Immutable(불변성)** - 한 번 생성된 RDD는 수정할 수 없음



RDD: 클러스터 전체에 분산된 데이터셋

RDD 연산 - Transformation

Transformation은 기존 RDD에서 새로운 RDD를 생성하는 연산으로, 지연 평가(Lazy Evaluation) 방식으로 동작합니다.

- **map(func)** : 각 요소에 함수를 적용하여 새로운 RDD 생성
- **filter(func)** : 조건을 만족하는 요소만으로 새 RDD 생성
- **flatMap(func)** : map + flatten 결합 연산
- **union(), intersection(), distinct()** : 집합 연산
- **groupByKey(), reduceByKey()** : 집계 연산

Transformation의 지연 평가 개념

RDD 연산 - Action

Action 연산은 실제 계산을 수행하고 결과를 반환하는 RDD 연산입니다. Transformation과 달리 즉시 실행되며 Spark의 지연 평가(Lazy Evaluation)를 트리거합니다.

- **count()** : RDD의 요소 개수 반환
- **collect()** : 모든 요소를 배열로 드라이버에 반환
- **reduce(func)** : 요소를 병합하는 함수를 사용해 집계
- **take(n)** : 처음 n개 요소만 반환
- **foreach(func)** : 각 요소에 함수 적용(결과 반환 없음)
- **saveAsTextFile(path)** : 결과를 텍스트 파일로 저장

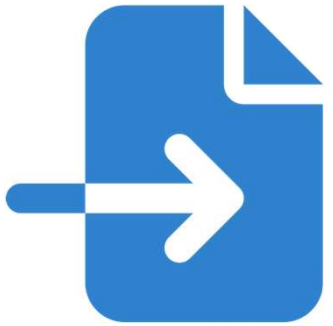
Action 연산은 계산을 즉시 실행하고 결과를 반환합니다

RDD 생성 방법 1: 파일 로딩

Spark에서는 다양한 파일 시스템의 데이터를 불러와 RDD를 생성할 수 있습니다.

- **sparkContext.textFile()** - 함수를 사용하여 텍스트 파일에서 RDD 생성
- 지원하는 파일 시스템 - **HDFS, S3, 로컬 파일시스템**
- 다양한 포맷 지원 - CSV, JSON, Parquet, ORC 등
- 자동 분산 처리 - 큰 파일도 파티션 단위로 분할 처리

```
// HDFS에서 파일 로딩
val hdfsFile = sc.textFile("hdfs://server/path/file.txt")


// 로컬 파일시스템에서 로딩
val localFile = sc.textFile("file:///path/to/file.txt")

// 와일드카드 패턴 사용
val logFiles = sc.textFile("/logs/2023/*.log")
```



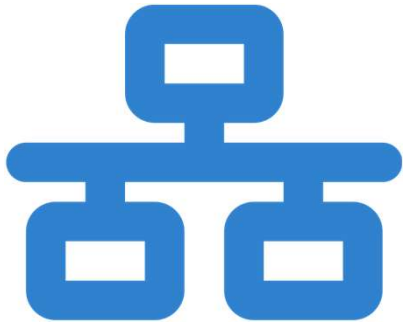
다양한 저장소에서 RDD 생성

RDD 생성 방법 2: 컬렉션 병렬화

Spark의 parallelize() 함수를 사용하여 기존 Scala 컬렉션을 RDD로 변환하는 방법입니다.

```
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
// 이제 distData는 클러스터에 분산된 RDD 입니다
```

- 로컬 메모리 데이터를 **분산 환경으로 빠르게 전환**할 때 유용
- List, Seq, Map 등 **다양한 Scala 컬렉션** 지원
- 파티션 수를 두 번째 파라미터로 지정 가능: **parallelize(data, 10)**



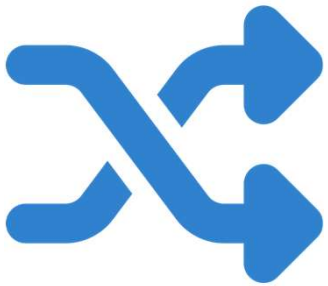
메모리 데이터가 분산 RDD로 변환되는 과정

RDD 생성 방법 3: 기존 RDD에서 변환

Spark의 가장 일반적인 RDD 생성 방법은 기존 RDD에 변환 연산(Transformations)을 적용하는 것입니다.

```
// 원본 RDD에서 변환 연산 적용
val words = textFile.flatMap(line => line.split(" "))
val pairs = words.map(word => (word, 1))
val counts = pairs.reduceByKey(_ + _)
```

- **map**: 각 요소를 새로운 요소로 변환 (1:1)
- **filter**: 조건을 만족하는 요소만 선택
- **flatMap**: 각 요소를 0개 이상의 요소로 확장
- **groupBy, join, union**등 다양한 변환 지원



변환 연산은 지연 평가(Lazy Evaluation) 방식으로 실행됩니다.

RDD의 캐싱: Persistence 레벨 변경

Spark에서는 RDD의 데이터를 메모리나 디스크에 유지하는 방식을 지정할 수 있습니다. 이를 통해 반복 작업에서 성능을 극대화할 수 있습니다.

- **.cache()** - 기본 메모리 저장 (MEMORY_ONLY와 동일)
- **.persist()** - 저장 레벨을 지정하여 RDD 유지

  저장 레벨: **MEMORY_ONLY**,**MEMORY_AND_DISK**,**DISK_ONLY 등**

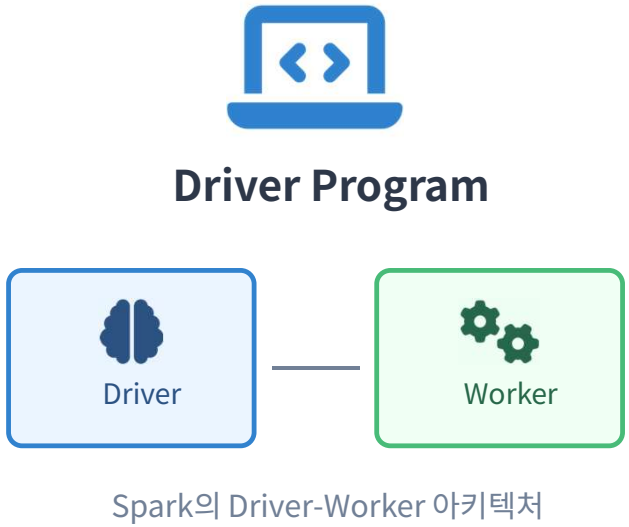
  저장 전략에 따라 메모리 사용량과 계산 속도 사이의 균형 조절

다양한 Persistence 레벨에 따른 저장 방식

Spark 프로그래밍 모델 개요

Spark는 Master-Worker 형태의 분산 처리 구조를 가지며, Driver 프로그램이 작업을 제어하고 Worker 노드가 실제 처리를 담당합니다.

- **Driver-Worker 구조**
  Driver가 애플리케이션 로직을 담당하고, Worker가 실제 데이터 처리를 수행
- **Scala 언어 채택**
  함수형 프로그래밍, JVM 호환성, 간결한 문법으로 빅데이터 처리에 적합
- **유연한 API**
  고수준 함수형 API로 복잡한 데이터 처리를 간결하게 표현 가능
- Python, Java, R 등 다양한 언어 지원으로 **접근성 향상**



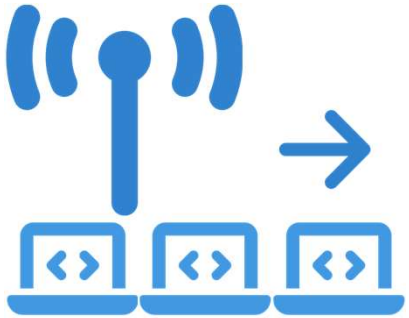
Spark의 Driver-Worker 아키텍처

Shared Variables: Broadcast

Broadcast 변수는 클러스터의 모든 노드에 읽기 전용 데이터를 효율적으로 배포하는 메커니즘입니다.

- 큰 데이터셋을 모든 태스크에 복사하는 대신, **한 번만 전송**하고 공유
- 대용량 조인(Join) 연산에서 **성능 대폭 향상**(최대 수십 배)
- 룩업 테이블, 피처 벡터, 머신러닝 모델 파라미터 공유에 활용

```
// Broadcast 변수 생성 예제
val broadcastVar = sc.broadcast(Array(1, 2, 3))

// 작업 내에서 사용
val result = rdd.map(x => x + broadcastVar.value(0))
```



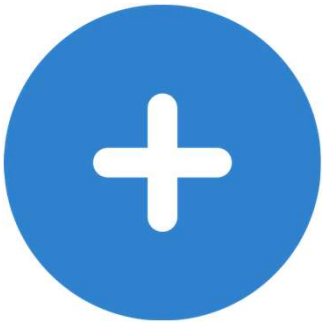
Broadcast 변수는 드라이버에서 모든 워커 노드로 효율적으로 데이터를 배포합니다

Shared Variables: Accumulator

Accumulator는 여러 노드에서 값을 모으는(집계하는) 변수로, 병렬 작업에서 안전하게 값을 누적할 수 있습니다.

- Worker 노드에서는 값을 **추가만 가능**(add-only)
- Driver 프로그램만 **최종 값을 읽을 수 있음**
- 주로 **카운터나 합계**용도로 사용
- Spark의 **내부 작업 통계**에도 활용됨

```
val accum = sc.longAccumulator("MyAccumulator")
sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))
println(accum.value) // 출력: 10
```



Accumulator - 분산 환경에서 안전한 값 누적

Spark vs Hadoop 성능 비교

실험 결과: Hadoop은 각 반복마다 127초, Spark는 첫 반복 174초, 이후 반복은 단 6초로 최대 20배 성능 향상 달성

메모리 캐싱으로 두 번째 이후 반복 시간 획기적 단축

반복적 머신러닝 알고리즘에서 압도적 성능 우위

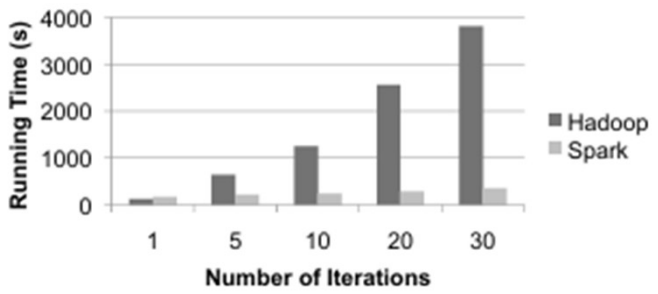


Figure 2: Logistic regression performance in Hadoop and Spark.

출처: "Spark: Cluster Computing with Working Sets" (2010) 실험 결과

Spark 실험환경 및 주요 결과

2010년 논문에서 Spark의 성능을 검증하기 위해 사용된 실험 환경과 그 결과를 살펴봅니다.

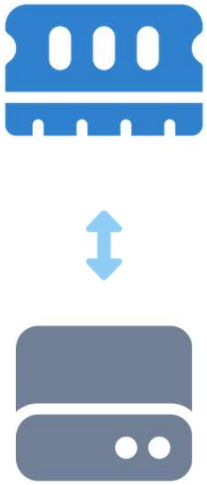
- **실험 환경:**20대의 m1.xlarge EC2 인스턴스 클러스터
- 각 노드: 4코어 CPU, 15GB 메모리, 1.7TB 디스크
- 데이터셋: 약**29GB의 텍스트 데이터**(Wikipedia 덤프)
- 벤치마크: 반복적 머신러닝, 텍스트 검색, 로그 마이닝

작업 유형	Hadoop	Spark	성능 향상
반복 (첫 회)	127초	174초	-
반복 (이후)	127초	6초	약 20배 ↑

EC2 클러스터 기반 성능 테스트 환경

실행 과정 비교(상세 분석)

단계	Spark	Hadoop MapReduce
데이터 로딩	HDFS에서 데이터를 로드하여 RDD 생성 (첫 번째 실행에서만)	HDFS에서 데이터를 로드 (매 작업마다 반복)
변환 단계	Transformation 연산 적용 (Lazy Evaluation)	Map 작업 수행 → 로컬 디스크에 중간 결과 저장
데이터 저장	메모리에 RDD 캐싱 (.cache() 또는 .persist())	Shuffle → 디스크 저장 → 네트워크 전송
집계/처리	캐시된 RDD에 Action 연산 적용 (즉시 결과 반환)	Reduce 작업 수행 → 결과 집계
반복 처리	캐시된 데이터 재사용 (디스크 I/O 없음)	데이터 로딩 부터 모든 과정 재시작 (디스크 I/O 발생)



Spark의 메모리 우선 처리 vs Hadoop의 디스크 기반 처리

Spark의 초기 성공 요인 요약

2010년 논문 발표 이후, Spark는 빅데이터 처리의 새로운 패러다임으로 빠르게 자리잡았습니다. 단순하면서도 강력한 API와 메모리 기반 처리 방식이 핵심 성공 요인이었습니다.

단순하고 직관적인 API 설계와 프로그래밍 모델

반복적/대화형 작업에서 최대 10-100배 성능 향상

실질적인 빅데이터 요구 충족과 활발한 커뮤니티

참고: Spark 개발 배경과 언어 선택의 철학

빅데이터 처리의 패러다임을 바꾼 기술적 결정

- 빅데이터 환경에서 기존 MapReduce의 한계와 다양한 분석 요구의 등장
- 다양한 워크로드(배치+실시간+인터랙티브)를 하나의 통합 프레임워크로 해결할 필요성
- Spark는 "통합(Unified) 데이터 처리 프레임워크"라는 철학 아래 개발됨
- 언어 선택 또한 이 철학을 반영: 빠른 실험, 학습, 분석, 생산성을 모두 지원해야 함

기반 언어로 Scala를 선택한 기술적 이유

간결한 문법과 높은 표현력

Scala의 간결하고 표현력 있는 문법은 빠른 프로토타이핑과 인터랙티브 분석에 최적화되어 있어 데이터 처리 코드를 직관적으로 작성 가능

함수형 프로그래밍 패러다임

함수형 프로그래밍 지원으로 RDD의 map, filter, reduce 등 분산 데이터 연산 추상화를 자연스럽게 표현할 수 있어 병렬 처리에 최적화

정적 타입 시스템

컴파일 타임 타입 검사를 통한 안정성과 실행 효율(성능) 보장, 오류를 사전에 발견하고 최적화된 코드 생성

JVM 호환성

Java Virtual Machine에서 실행되어 기존 Java/Hadoop 생태계와 높은 호환성을 제공하고, 기존 자바 라이브러리를 활용 가능

인터랙티브 셸(REPL) 지원

대화형 셸을 통해 대규모 데이터셋을 즉각적으로 탐색하고 분석 가능, 빠른 반복 개발과 데이터 과학 워크플로우에 최적화

Scala 기반 Spark의 실무적 장점과 영향

높은 생산성과 개발 효율

간결한 문법과 인터랙티브 환경을 통해 데이터 과학자와 엔지니어의 생산성이 크게 향상, 코드 작성과 테스트의 반복 주기 단축

복잡한 알고리즘 구현 용이

함수형 추상화와 강력한 타입 시스템 덕분에 머신러닝, 그래프 처리, 복잡한 데이터 분석 알고리즘을 간결하고 명확하게 구현 가능

빅데이터 처리 표준으로 자리매김

JVM 기반의 확장성과 범용성을 바탕으로 Spark는 전 세계적으로 빅데이터 처리의 표준 플랫폼으로 성장, 다양한 산업 분야에서 채택

다중 언어 API 확장성

Spark API는 Python, Java, R 등 다양한 언어로 확장되어 접근성을 높였지만, 최신 기능과 성능 최적화는 Scala에서 가장 먼저 구현되고 제공됨

활발한 생태계 형성

Scala 기반의 Spark는 활발한 오픈소스 커뮤니티와 광범위한 기업 지원을 받아 지속적인 혁신과 개선이 이루어지는 건강한 생태계 구축

Part 2 - RDD의 고급 개념

2012년 논문을 바탕으로 RDD 이론적 완성 및 내결함성, 최적화 전략 등 심화 개념을 학습합니다.

RDD 인터페이스 세부 구조와 의존성 (Narrow/Wide)

Lineage 기반 내결함성 메커니즘

고급 최적화 기법과 실제 성능 개선 사례

2년간 Spark의 진화

2010년 아이디어에서 2012년 완성된 시스템으로 진화한 Apache Spark의 발전 과정과 특징적 변화를 살펴봅니다.

- 2010년: **초기 아이디어와 프로토타입** 단계
- 2012년: **실제 워크로드 지원**이 가능한 안정적 시스템으로 발전
- 연구 목표의 확장: 단순 성능 향상 → **범용 분산 컴퓨팅** 플랫폼
- 연구진 확대: 5명 → 9명 (더 다양한 **전문가 참여**)
- 산업계의 관심 증대: 야후, 인텔 등 **실제 환경에서의 검증**

2010-2012: Spark의 발전과 성장 궤적

RDD의 3대 특성

RDD의 핵심 특성은 Spark의 성능과 안정성을 보장하는 근간이 됩니다. 이 세 가지 특성이 분산 환경에서 어떻게 작동하는지 이해하는 것이 중요합니다.

- **Immutability(불변성):**
  RDD는 생성 후 수정이 불가능한 읽기 전용 객체로, 일관성 유지와 병렬 처리의 안전성을 보장합니다.
- **Lazy evaluation(지연 평가):**
  변환 연산은 즉시 실행되지 않고, Action이 호출될 때까지 연산을 지연시켜 최적화 기회를 제공합니다.
- **Lineage(계보):**
  RDD는 자신이 어떻게 생성되었는지 변환 기록을 추적하여, 장애 발생 시 재계산을 통한 복구가 가능합니다.

RDD의 세 가지 핵심 특성이 Spark의 높은 성능과 내결함성을 지원합니다

Lazy evaluation & Lineage

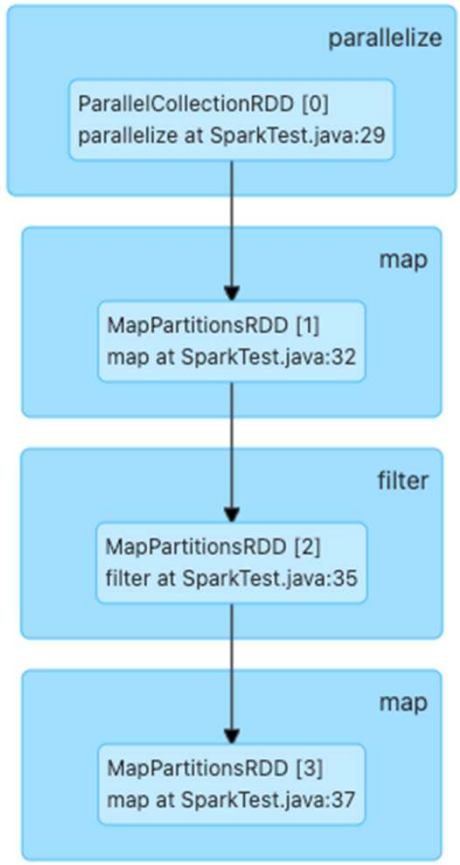
RDD는 Action 계열의 함수가 호출 되기 전 까지 수행 될 작업에 대해서 실행 전략을 미리 계획 합니다. 자주 사용 되는 RDD, 사용 예측이 되는 RDD 등을 미리 고려하여 최적의 자원으로 수행 할 수 있도록 전략을 세웁니다.

```
Int Num1 = 1;
Int Num2 = 2;
Int Tmp = Num1 + Num2;
Int Result = Tmp – Num1;

system.out.println(Result);
```

최적화 전

```
~~Int Num1 = 1;~~
Int Num2 = 2;
Int Tmp = ~~Num1 +~~ Num2;
Int Result = Tmp ~~– Num1~~;

System.out.println(Result);
```

최적화 후



RDD Lineage

Lineage와 내결함성(Fault Tolerance)

Spark의 혁신적인 내결함성 메커니즘: 데이터 자체가 아닌 데이터의 생성 과정(Lineage)만 기록하여 장애 발생 시 필요한 부분만 효율적으로 복구합니다.

체크포인트/복제 대신 RDD 변환 연산 기록만 저장

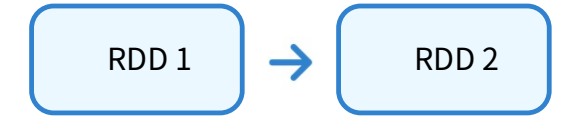
노드 장애 시 손실된 파티션만 선택적으로 재계산

저장 공간 최소화와 복구 속도 최적화 동시 달성

의존성 구조

RDD 간의 의존성(Dependency)은 RDD가 어떤 방식으로 다른 RDD로부터 파생되었는지를 정의하며, 스케줄링과 장애 복구에 결정적 영향을 미칩니다.

- **Narrow Dependency**
  : 부모 RDD의 각 파티션이 최대 하나의 자식 파티션에만 영향을 줌 (1:1 또는 N:1 관계)
- 예: map(), filter(), union() - 파티션 간 데이터 이동이 필요 없음
- **Wide Dependency**
  : 부모 RDD의 각 파티션이 여러 자식 파티션에 영향을 줌 (1:N 관계)
- 예: groupByKey(), join() - Shuffle 연산을 필요로 함
- 장애 발생 시: Narrow는 독립적 재계산 가능, Wide는 모든 부모 파티션 필요



Narrow Dependency (map, filter)

Wide Dependency (groupByKey, join)

의존성 구조

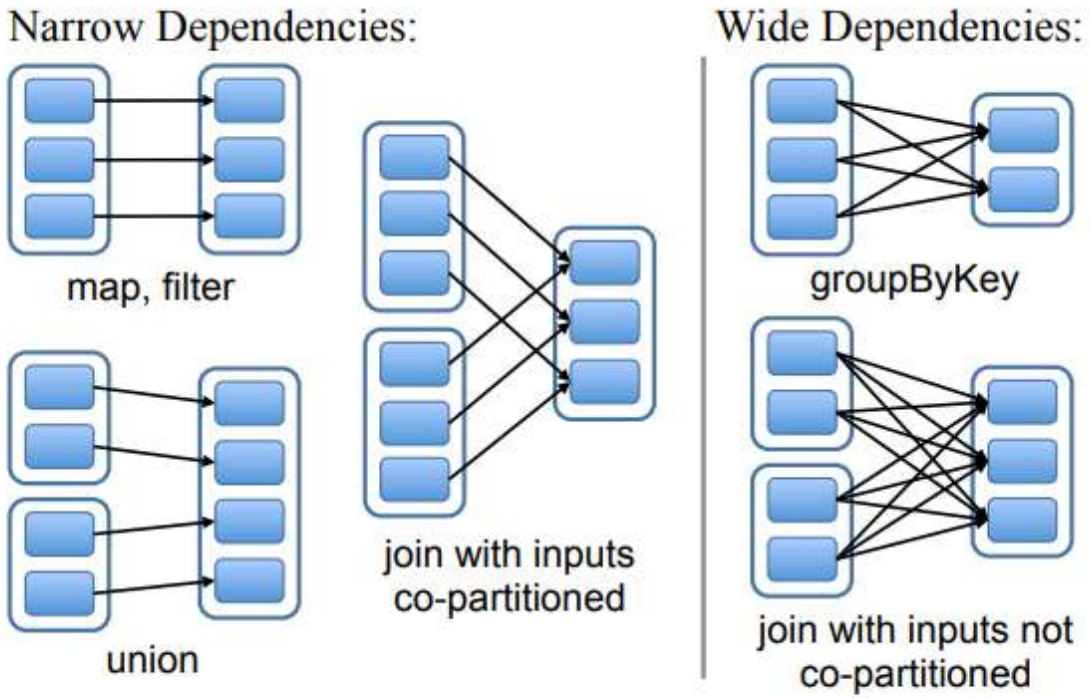


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

장애 복구 시나리오

Spark는 노드 장애가 발생했을 때 RDD의 계보 정보(Lineage)를 기반으로 손실된 파티션만 선택적으로 재계산하여 효율적으로 복구합니다.

- 특정 노드 장애 발생 → 해당 노드에 저장된 RDD 파티션 손실
- 파티션별**Lineage 그래프**활용하여 손실된 파티션만 재계산
- **Narrow Dependency**의 경우 병렬적으로 빠르게 복구 가능
- **Wide Dependency**의 경우 단계별 복구 필요 → 비용 증가
- 전체 어플리케이션 재시작 없이**부분적 재계산**으로 빠른 복구

Spark의 장애 복구 메커니즘 - 장애 발생 시 선택적 재계산

Narrow Dependency

```
var lines = load(“input_file”)
var errors = lines.filter(_.startsWith(“ERROR”))
var HDFS_errors = error.filter(_.contains(“HDFS”))
var time_fields = HDFS_errors.map(_.split(‘\t’)(3))
time_fields.save(“./error_time.dat”)
```

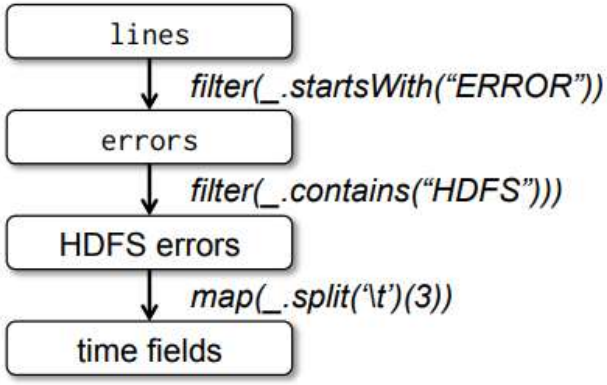


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

Narrow Dependency

예제 데이터 (HDFS 또는 로컬 파일)

임시 채팅

`logs.txt`

```pgsql
INFO User login successful
ERROR Database connection failed
INFO User viewed page
WARN Disk space low
ERROR Timeout occurred
```

1 Step 1. RDD 생성

```python
rdd1 = sc.textFile("logs.txt")
```

이 시점:

- `rdd1`은 `logs.txt`의 각 줄을 원소로 하는 **RDD**
- 아직 아무 데이터도 읽지 않음 (lazy)
- 파티션 단위로 분할되어 있음 (예: 2개의 파티션: P1, P2)

파티션	내용
P1	INFO User login successful ERROR Database connection failed
P2	INFO User viewed page WARN Disk space low ERROR Timeout occurred

2 Step 2. Transformation — narrow dependency

```python
rdd2 = rdd1.filter(lambda line: "ERROR" in line)
```

- `filter()`는 **narrow dependency**
- 각 파티션이 이전 RDD의 동일한 파티션(P1→P1', P2→P2')만을 참조함
- 데이터 셔플 없음

RDD	파티션	내용
rdd2	P1'	ERROR Database connection failed
rdd2	P2'	ERROR Timeout occurred

3 Step 3. Action (트리거)

```python
rdd2.collect()
```

→ 이 시점에서 Spark가 DAG를 실행함 (`textFile → filter → collect`)

→ 결과:

```css
["ERROR Database connection failed", "ERROR Timeout occurred"]
```

Narrow Dependency

4️⃣ Step 4. 장애 발생 가정 ⚡

예를 들어, **노드 2가 죽어서 RDD2의 P2′ (ERROR Timeout occurred)** 가 손실되었다고 가정합시다.

- Spark는 **P2′의 lineage**를 확인합니다.
  → "P2′는 `rdd1.P2`에서 filter(lambda line: 'ERROR' in line)으로 만들어졌네."
- 따라서 Spark는 P2′만 다시 계산합니다:
  1. `rdd1.P2` 데이터를 다시 읽음 (`INFO User viewed page`, `WARN Disk space low`, `ERROR Timeout occurred`)
  2. filter를 다시 적용함 → `"ERROR Timeout occurred"`
  3. P2′ 재생성 완료 ✅

5️⃣ Step 5. 결과 복원

이제 Spark는 다시 완전한 `rdd2`를 가짐:

```
css                                                    코드 복사

["ERROR Database connection failed", "ERROR Timeout occurred"]
```

📍 주의:

- RDD 전체를 다시 계산하지 않았습니다.
- **필요한 파티션(P2′)만 재계산**했습니다.
- narrow dependency 덕분에 재계산 경로가 짧고 비용이 적습니다.

Wide Dependency

```
var input = load(“input_file”)
var links = input.map(_.split(“,”))
var ranks0 = load(“rank_data”)
var contribs0 = ranks0.filter(_.contains(“A”))
var ranks1 = links.join(contribs0)
var contribs1 = ranks1.filter(_.contains(“B”))
var ranks2 = links.join(contribs1)
var contribs2 = ranks2.filter(_.contains(“C”))
var final_ranks = links.join(contribs2)
final_ranks.save(“./result.out”)
```

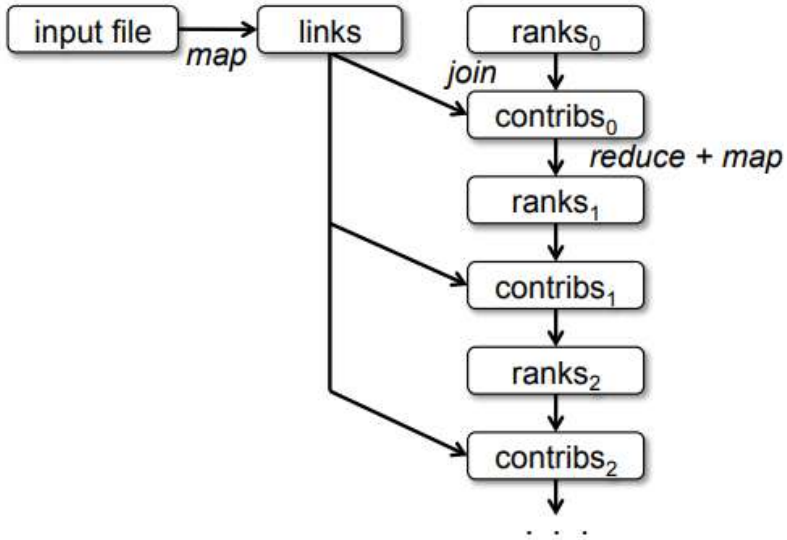


Figure 3: Lineage graph for datasets in PageRank.

Wide Dependency

```
var input = load(“input_file”)
var links = input.map(_.split(“,”))
var ranks0 = load(“rank_data”)
var contribs0 = ranks0.filter(_.contains(“A”))
var ranks1 = links.join(contribs0)
var contribs1 = ranks1.filter(_.contains(“B”))
var ranks2 = links.join(contribs1).persist
var contribs2 = ranks2.filter(_.contains(“C”))
var final_ranks = links.join(contribs2)
final_ranks.save(“./result.out”)
```

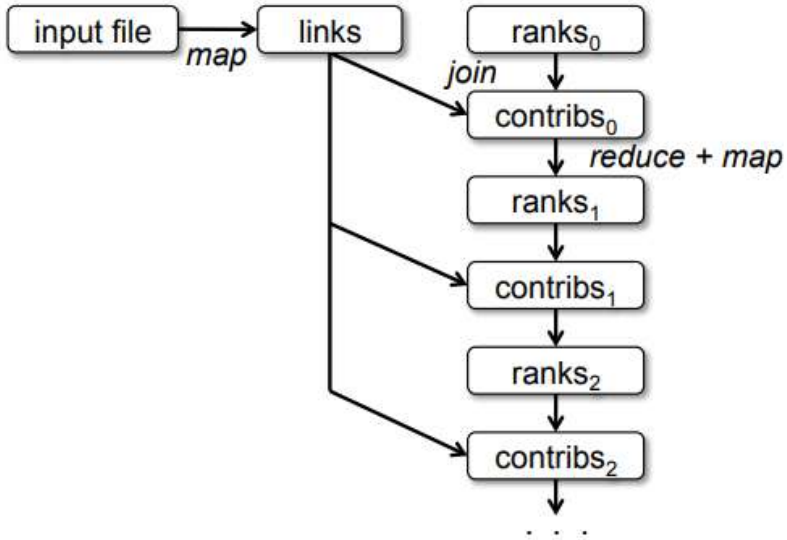


Figure 3: Lineage graph for datasets in PageRank.

Spark의 내결함성 실험 결과

Lineage 기반 복구 메커니즘의 효율성을 보여주는 실험 결과: 정상 실행 58초 → 노드 장애 발생 80초 → 복구 후 58초

정상 실행 시간: 58초

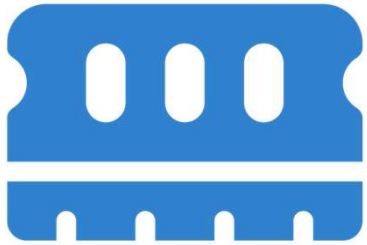
노드 장애 시간: 80초
(60% 성능 저하)

복구 후 시간: 58초

고급 메모리 관리 전략

Spark는 2012년 논문에서 메모리 관리를 위한 더 정교한 전략을 도입하여 제한된 메모리 환경에서도 효율적인 분산 처리를 가능하게 했습니다.

- **LRU 제거 정책**
  - 가장 오래 사용되지 않은 데이터를 메모리에서 우선 제거
- **영속성 우선순위**
  - 개발자가 지정한 RDD 중요도에 따라 캐싱 순위 결정
- **다양한 저장 레벨**
  - MEMORY_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER, OFF_HEAP

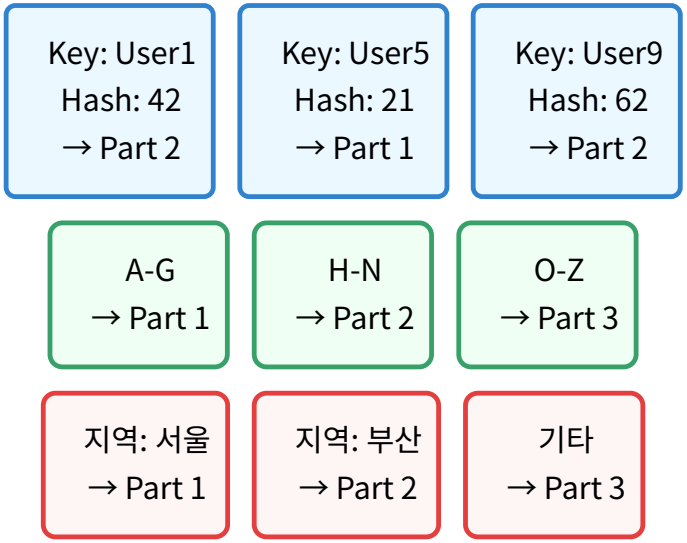


Spark의 메모리 계층 관리 구조

파티셔닝 전략 - Hash, Range, Custom

Spark에서는 데이터 분산 처리 효율을 높이기 위해 다양한 파티셔닝 전략을 제공합니다. 적절한 파티셔닝은 네트워크 트래픽과 셔플링을 최소화합니다.

- **Hash Partitioning**
  : 키의 해시값을 기준으로 데이터 분배, 균등한 분포가 장점이나 순서 보존이 안됨
- **Range Partitioning**
  : 키의 순서를 보존하여 정렬된 데이터에 효과적, 데이터 편향 가능성 있음
- **Custom Partitioner**
  : 특수 요구사항에 맞게 구현 가능, Partitioner 클래스 상속
- 적절한 파티셔닝은 **join 연산과 groupBy** 같은 셔플 연산의 성능을 크게 개선



Hash, Range, Custom 파티셔닝 예시

다양한 워크로드 성능평가

2012년 논문에서는 다양한 실제 워크로드에서 Spark의 성능을 평가하여 그 우수성을 증명했습니다. 특히 반복적 알고리즘과 대화형 쿼리에서 월등한 성능을 보였습니다.

- **Logistic Regression:** Hadoop 대비 최대 25.3배 성능 향상
- **K-means 클러스터링:** 데이터 크기에 따라 1.9~3.2배 성능 향상
- **PageRank:** 네트워크 분석 알고리즘에서 7.4배 성능 향상
- 실제 응용 사례: **텍스트 검색, 교통 데이터 분석, 로그 처리**
- 메모리 기반 처리의 이점이 반복적 계산이 많을수록 극대화됨

다양한 워크로드별 성능 향상 비교

대용량 인터랙티브 쿼리

Spark는 1TB 이상 대용량 데이터셋에서도 뛰어난 대화형 쿼리 성능을 보여줍니다. 2012년 논문에서 검증된 실험 결과를 살펴보겠습니다.

> 첫 쿼리: 170초 → 이후 쿼리: 5~7초

- 첫 번째 쿼리 실행 시 데이터를 메모리에**캐싱**
- 이후 동일 데이터셋 활용 쿼리는**초 단위**응답 시간
- 사용자가 데이터 탐색 시**실시간 분석**경험 제공
- 대화형 분석을 위한**Spark SQL**엔진의 기초 마련

첫 번째 쿼리: 170초

이후 쿼리: 5~7초

인터랙티브 쿼리 성능 향상 - 메모리 캐싱 효과

클러스터 확장성 검증

Spark는 클러스터 크기에 따른 성능 확장성과 리소스 제약 상황에서의 우아한 성능 저하(Graceful Degradation) 특성을 보여줍니다. 다양한 규모의 클러스터에서 일관된 내결함성을 유지합니다.

클러스터 크기별 성능 확장성: 노드 수 증가에 따른 선형적 성능 향상

메모리 부족 시 디스크로 우아하게 전환, 급격한 성능 저하 방지

대규모 클러스터에서도 5-7% 미만의 낮은 내결함성 오버헤드 유지

산업계 적용 사례

Apache Spark는 출시 이후 다양한 산업 분야에서 실제 비즈니스 문제 해결에 활발히 적용되고 있습니다. 특히 대용량 데이터 처리와 실시간 분석이 필요한 영역에서 두각을 나타냅니다.

- **Yahoo**- 스팸 탐지 및 개인화 추천 시스템에 Spark MLlib 활용
- **Intel**- 사물인터넷(IoT) 데이터 분석 및 실시간 처리
- **Amazon**- 대규모 로그 분석 및 추천 엔진 개발
- **금융권**- 실시간 부정거래 탐지 및 위험 분석
- **교통/물류**- 실시간 교통 예측 및 경로 최적화

다양한 산업 분야에서의 Spark 활용

Apache Spark의 오픈소스화

Spark는 UC Berkeley AMPLab에서 탄생한 후, 오픈소스 프로젝트로 급속히 성장하며 글로벌 커뮤니티와 함께 발전했습니다.

2010년

UC Berkeley AMPLab에서 첫 논문 발표 및 오픈소스 공개

2013년

Apache Software Foundation(ASF)에 **기증**되어 인큐베이터 프로젝트 시작

2014년 2월

Apache **Top-Level Project로 승격**, 커뮤니티 기반 거버넌스 확립

2014-2022년

연평균 **1,000명 이상의 기여자**가 참여하는 활발한 오픈소스 생태계 형성

Apache Software Foundation과 Spark의 만남

오픈소스의 핵심 가치

- 투명한 개발 과정
- 공동체 기반 의사결정
- 기여자 생태계 확장

Spark 발전의 의의와 현재

Apache Spark는 빅데이터/AI 시대의 핵심 기술로 진화하여 단순 처리 엔진에서 전방위적 데이터 플랫폼으로 발전했습니다. 다양한 에코시스템과 산업 표준으로서 그 가치가 지속적으로 확대되고 있습니다.

Spark SQL, MLlib, GraphX, Structured Streaming 등 다양한 확장 모듈로 성장

현대 기업 빅데이터 아키텍처의 표준 기술로 자리매김

AI/ML 워크로드와의 통합을 통한 지속적인 진화

핵심 시사점 & 마무리

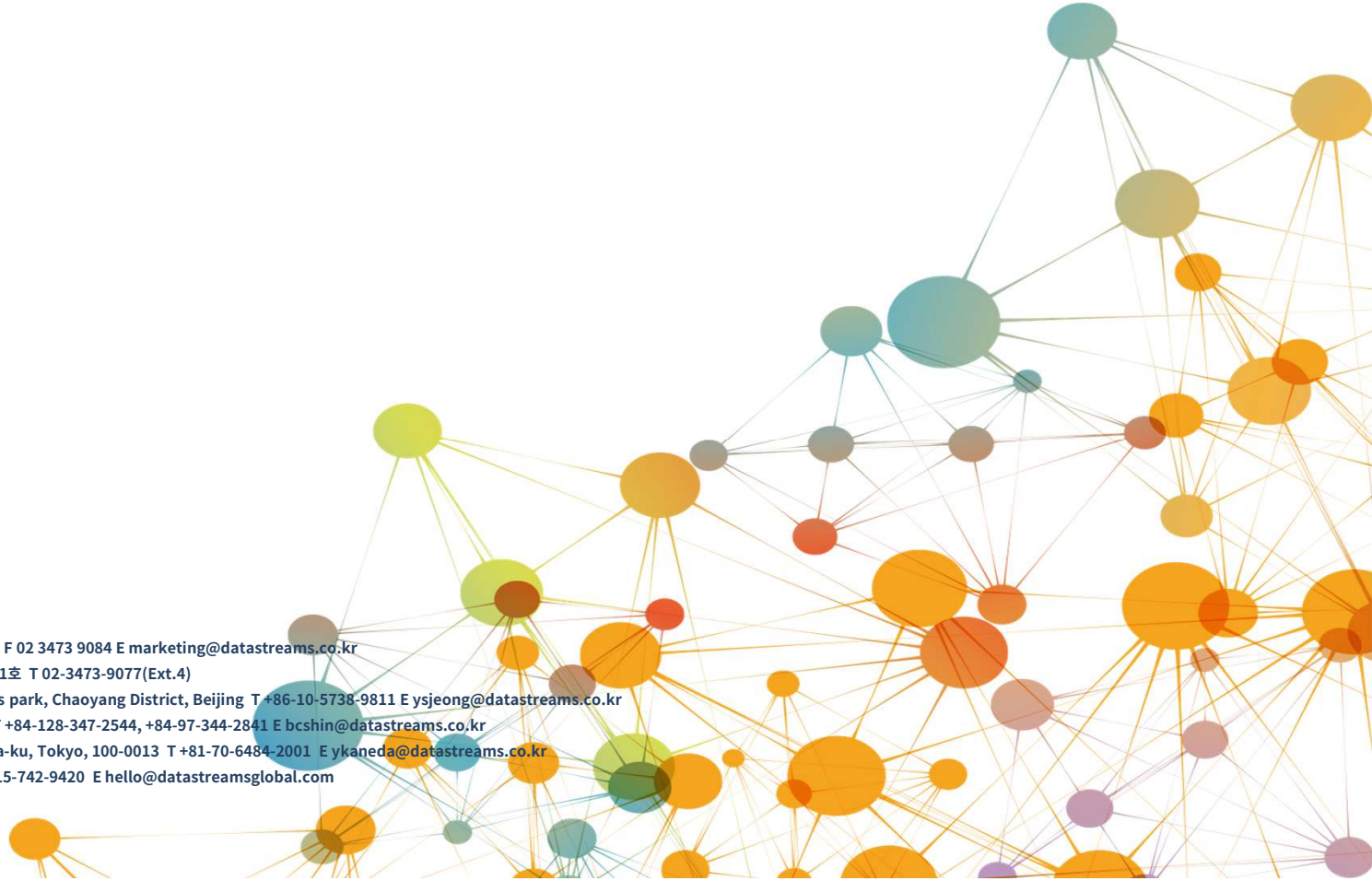
Apache Spark는 빅데이터 처리의 패러다임을 바꾸며 현대 데이터 분석과 AI 시대의 핵심 기술로 자리매김했습니다. 2010년 논문에서 시작된 아이디어가 오늘날의 데이터 생태계를 어떻게 변화시켰는지 돌아보겠습니다.

RDD의 혁신적 개념이 가져온 분산 컴퓨팅의 새로운 지평

학술적 아이디어가 산업 전반을 변화시킨 오픈소스 성공 사례

AI와 빅데이터 시대에 계속 진화하는 Spark 생태계의 미래

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