aws

AWS Glue Apache Spark 작업의 성능 튜닝 모범 사례

: AWS Glue Apache Spark 작업의 성능 튜닝 모범 사례

Table of Contents

소개 1
주요 주제 2
아키텍처 2
복원력 있는 분산 데이터세트 3
지연 평가 4
Spark 애플리케이션 용어 5
Parallelism 6
Catalyst 옵티마이저 7
성능 문제 조사 10
Spark UI를 사용하여 병목 현상 식별 10
성능 조정을 위한닝 전략 12
성능 조정을 위한 기준선 전략 12
Spark 작업 성능에 대한 조정 사례 13
클러스터 용량 규모 조정 13
CloudWatch 지표 14
Spark UI 14
최신 버전 사용 16
데이터 스캔 양 감소 16
CloudWatch 지표 17
Spark UI 17
태스크 병렬화 26
CloudWatch 지표 26
Spark UI 27
셔플 최적화 32
CloudWatch 지표 33
Spark UI 33
계획 오버헤드 최소화 41
CloudWatch 지표 42
Spark UI 42
사용자 정의 함수 최적화 43
표준 Python UDF 45
벡터화된 UDF 45
Spark SQL 46
빅 데이터에 panda 사용 47

리소스 48
문서 기록 49
..... I

AWS Glue Apache Spark 작업의 성능 튜닝 모범 사례

Roman Myers, Takashi Onikura, Noritaka Sekiyama, Amazon Web Services(AWS)

2023년 12월(문서 기록)

AWS Glue 는 성능 튜닝을 위한 다양한 옵션을 제공합니다. 이 가이드에서는 Apache Spark의 튜닝 AWS Glue 에 대한 주요 주제를 정의합니다. 그런 다음 Apache Spark 작업에 AWS Glue 대해 이를 튜닝할 때 따라야 할 기준 전략을 제공합니다. 이 가이드를 사용하여 AWS Glue에서 사용할 수 있는 지표를 해석함으로써 성능 문제를 식별하는 방법을 알아봅니다. 그런 다음 이러한 문제를 해결하기 위한 전략을 통합하여 성능을 극대화하고 비용을 최소화합니다.

이 가이드에서는 다음과 같은 조정 사례를 다룹니다.

- 클러스터 용량 규모 조정
- 최신 AWS Glue 버전 사용
- 데이터 스캔 양 감소
- 태스크 병렬화
- 계획 오버헤드 최소화
- 셔플 최적화
- 사용자 정의 함수 최적화

Apache Spark의 주요 주제

이 섹션에서는 Apache Spark 성능 튜닝을 위한 AWS Glue Apache Spark 기본 개념과 주요 주제를 설명합니다. 실제 조정 전략을 논의하기 전에 이러한 개념과 주제를 이해하는 것이 중요합니다.

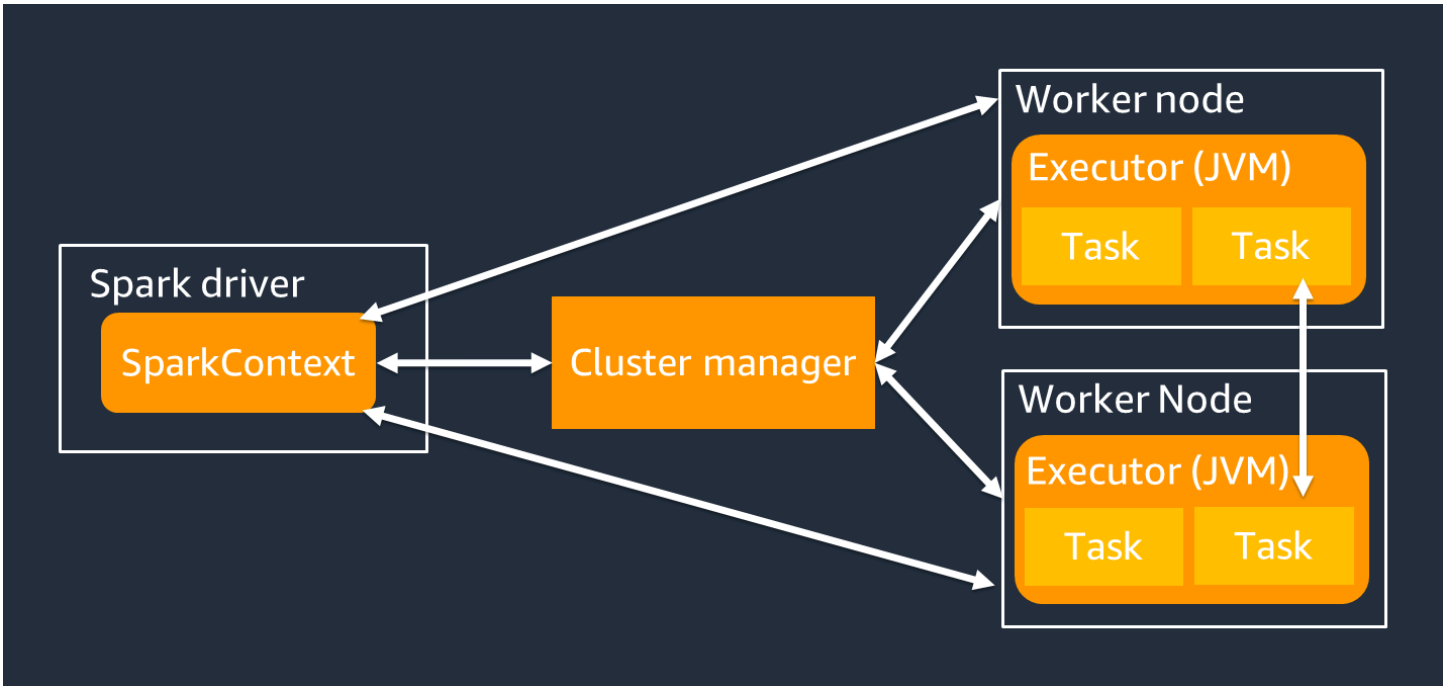
아키텍처

Spark 드라이버는 주로 Spark 애플리케이션을 개별 작업자에서 수행할 수 있는 태스크로 분할하는 작업을 담당합니다. Spark 드라이버에는 다음과 같은 책임이 있습니다.

- 코드에서 `main()` 실행
- 실행 계획 생성
- 클러스터의 리소스를 관리하는 클러스터 관리자와 함께 Spark 실행기 프로비저닝
- Spark 실행기에 대한 태스크 예약 및 태스크 요청
- 태스크 진행 상황 및 복구 관리

`SparkContext` 객체를 사용하여 작업 실행을 위해 Spark 드라이버와 상호 작용합니다.

Spark 실행기는 데이터를 보관하고 Spark 드라이버에서 전달되는 태스크를 실행하는 작업자입니다. Spark 실행기 수는 클러스터 크기에 따라 증가하거나 감소합니다.

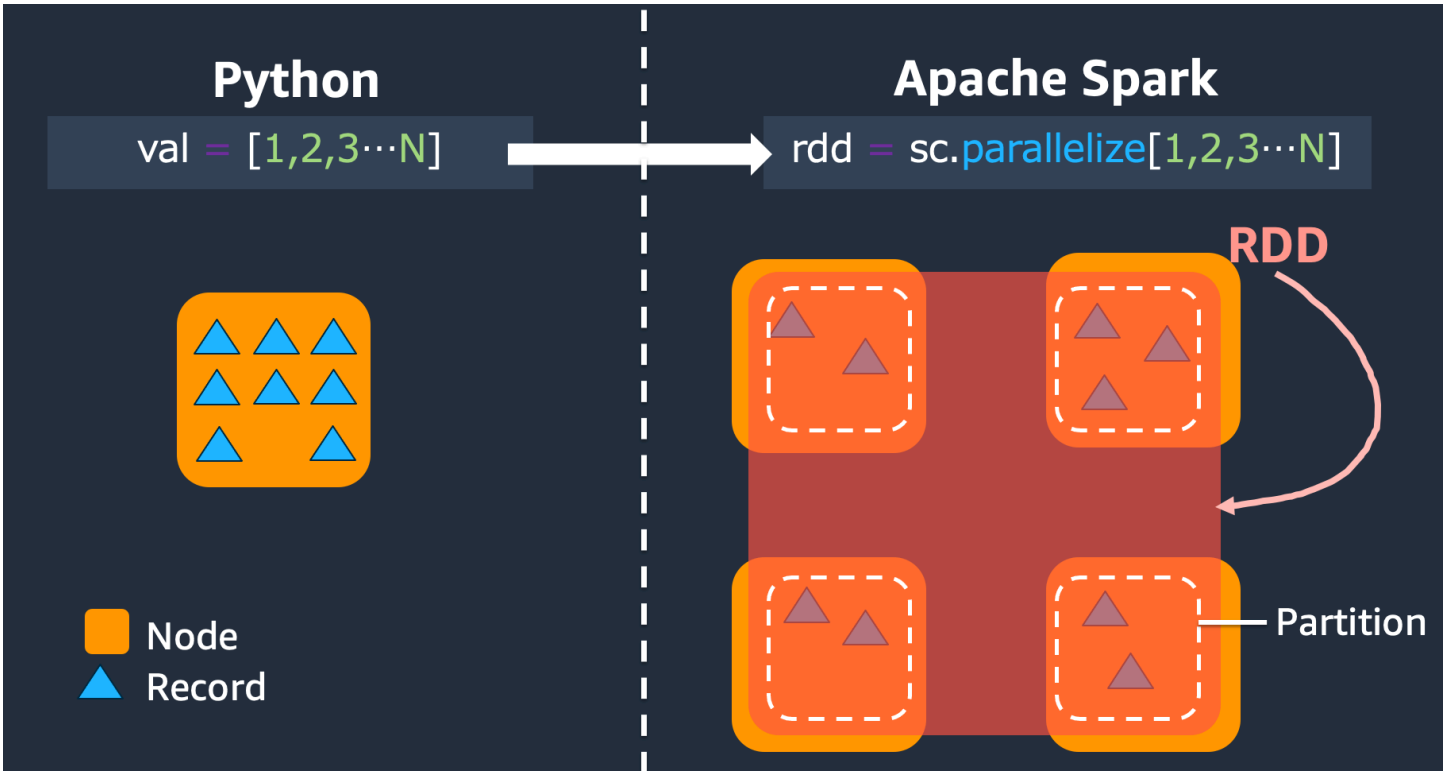


> **Note**
>
> Spark 실행기에는 여러 개의 슬롯이 있으므로 여러 태스크를 병렬로 처리할 수 있습니다. Spark는 기본적으로 각 가상 CPU(vCPU) 코어에 대해 하나의 태스크를 지원합니다. 예를 들어 실행기에 4개의 CPU 코어가 있는 경우 4개의 동시 태스크를 실행할 수 있습니다.

복원력 있는 분산 데이터세트

Spark는 Spark 실행기 전체에서 대규모 데이터세트를 저장하고 추적하는 복잡한 작업을 수행합니다. Spark 작업에 대한 코드를 작성할 경우 스토리지 세부 정보를 고려할 필요가 없습니다. Spark는 복원력이 뛰어난 분산 데이터세트(RDD) 추상화를 제공합니다. 이 추상화는 병렬로 작동하고 클러스터의 Spark 실행기에서에 분할할 수 있는 요소의 모음입니다.

다음 그림에서는 Python 스크립트가 일반적인 환경에서 실행될 때와 Spark 프레임워크(PySpark)에서 실행될 때 메모리에 데이터를 저장하는 방법의 차이를 보여줍니다.



- Python - Python 스크립트에서 `val = [1,2,3...N]`을 작성하면 코드가 실행 중인 단일 시스템의 메모리에 데이터가 유지됩니다.

- PySpark – Spark는 여러 Spark 실행기의 메모리에 분산된 데이터를 로드하고 처리하기 위해 RDD 데이터 구조를 제공합니다. `rdd = sc.parallelize[1,2,3...N]`와 같은 코드를 사용하여 RDD를 생성할 수 있으며, Spark는 여러 Spark 실행기에 데이터를 자동으로 배포하고 메모리에 보관할 수 있습니다.

  많은 AWS Glue 작업에서 DynamicFrames 및 Spark DataFrames를 통해 AWS Glue RDDs를 사용합니다. DataFrames RDD에서 데이터의 스키마를 정의하고 해당 추가 정보로 상위 수준 태스크를 수행할 수 있는 추상화입니다. RDD를 내부적으로 사용하기 때문에 데이터는 투명하게 배포되고 다음 코드의 여러 노드에 로드됩니다.

  - DynamicFrame

    ```
    dyf= glueContext.create_dynamic_frame.from_options(
        's3', {"paths": [ "s3://<YourBucket>/<Prefix>/"]},
        format="parquet",
        transformation_ctx="dyf"
    )
    ```

  - DataFrame

    ```
    df = spark.read.format("parquet")
        .load("s3://<YourBucket>/<Prefix>")
    ```

RDD에는 다음과 같은 기능이 있습니다.

- RDD는 파티션이라고 하는 여러 부분으로 분할된 데이터로 구성됩니다. 각 Spark 실행기는 하나 이상의 파티션을 메모리에 저장하고 데이터는 여러 실행기에 분산됩니다.
- RDD는 변경 불가능하므로 생성된 후에는 변경할 수 없습니다. DataFrame을 변경하려면 다음 섹션에 정의된 변환을 사용할 수 있습니다.
- RDD는 사용 가능한 여러 노드에서 데이터를 복제하므로 노드 장애로부터 자동으로 복구할 수 있습니다.

지연 평가

RDD는 기존 작업에서 새 데이터세트를 생성하는 변환과 데이터세트에서 계산을 실행한 후 드라이버 프로그램에 값을 반환하는 작업이라는 두 가지 유형의 작업을 지원합니다.

- 변환 - RDD는 변경 불가능하므로 변환을 사용해야만 변경할 수 있습니다.

예를 들어 map은 각 데이터세트 요소를 함수를 통해 전달하고 결과를 나타내는 새 RDD를 반환하는 변환입니다. map 메서드는 출력을 반환하지 않습니다. Spark는 사용자가 결과와 상호 작용할 수 있도록 하는 대신 향후 추상 변환을 저장합니다. Spark는 사용자가 작업을 직접 호출할 때까지 변환에 대해 작업하지 않습니다.

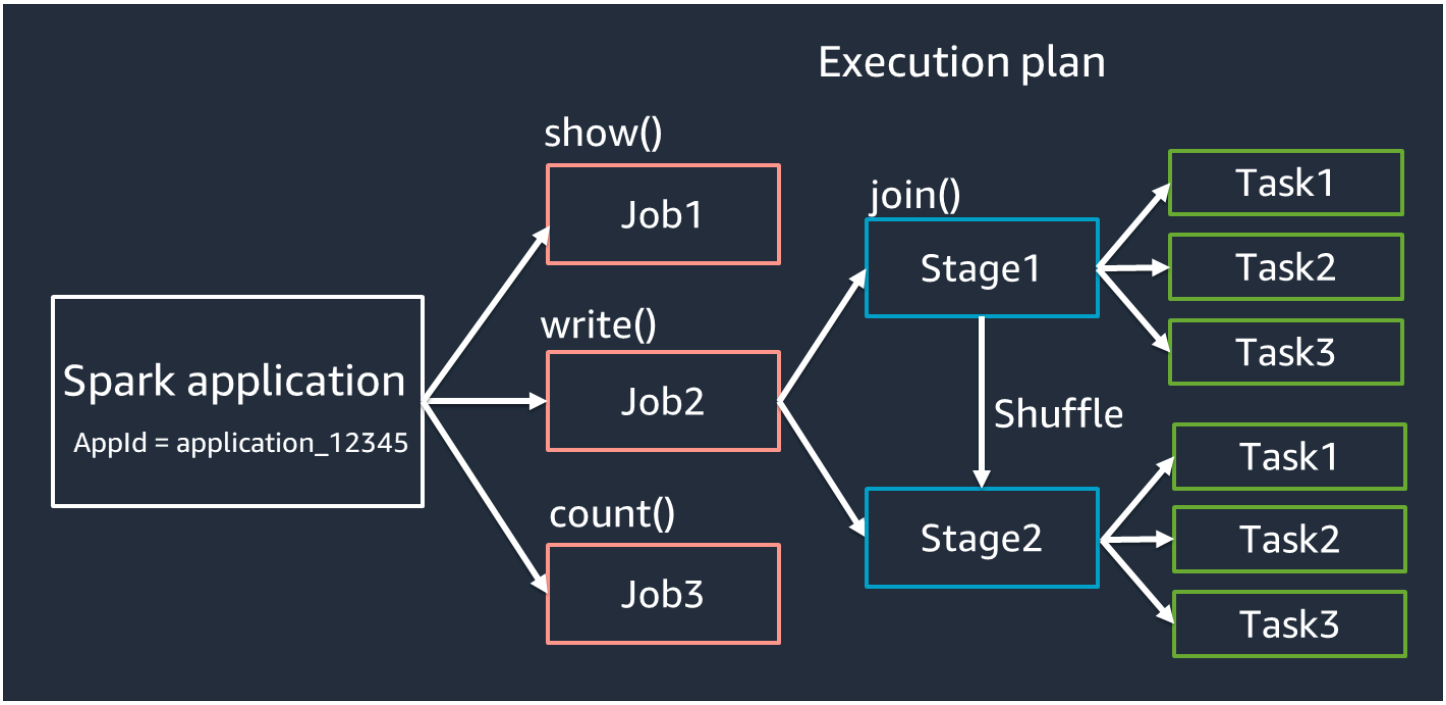
- 작업 - 변환을 사용하여 논리적 변환 계획을 수립합니다. 계산을 시작하려면 write, count show 또는 collect와 같은 작업을 실행합니다.

  Spark의 모든 변환은 결과를 즉시 계산하지 않는다는 점에서 지연됩니다. 대신 Spark는 Amazon Simple Storage Service(Amazon S3) 객체와 같은 일부 기본 데이터세트에 적용된 일련의 변환을 기억합니다. 변환은 작업에서 결과를 드라이버에 반환해야 하는 경우에만 계산됩니다. 이 설계를 통해 Spark를 더 효율적으로 실행할 수 있습니다. 예를 들어 map 변환을 통해 생성된 데이터세트가 reduce와 같이 행 수를 크게 줄이는 변환에 의해서만 소비되는 상황을 고려합니다. 그런 다음 매핑된 더 큰 데이터세트를 전달하는 대신 두 변환을 모두 거친 더 작은 데이터세트를 드라이버에 전달할 수 있습니다.

Spark 애플리케이션 용어

이 섹션에서는 Spark 애플리케이션 용어를 다룹니다. Spark 드라이버는 실행 계획을 생성하고 여러 추상화에서 애플리케이션의 동작을 제어합니다. 다음 용어는 Spark UI를 사용한 개발, 디버깅 및 성능 조정에 중요합니다.

- 애플리케이션 - Spark 세션(Spark 컨텍스트)에 기반합니다. 고유 ID(예: <application_XXX>)로 식별됩니다.
- 작업 - RDD에 대해 생성된 작업에 기반합니다. 작업은 하나 이상의 단계로 구성됩니다.
- 스테이지 - RDD에 대해 생성된 셔플에 기반합니다. 단계는 하나 이상의 태스크로 구성됩니다. 셔플은 데이터를 재분산하기 위한 Spark의 메커니즘으로, 이를 통해 여러 RDD 파티션에서 다르게 데이터가 그룹화됩니다. join()과 같은 특정 변환에는 셔플이 필요합니다. 셔플은 셔플 최적화 조정 사례에서 자세히 설명합니다.
- 태스크 - 태스크는 Spark에서 예약한 최소 처리 단위입니다. 각 RDD 파티션에 대해 태스크가 생성되며, 태스크 수는 단계의 최대 동시 실행 수입니다.

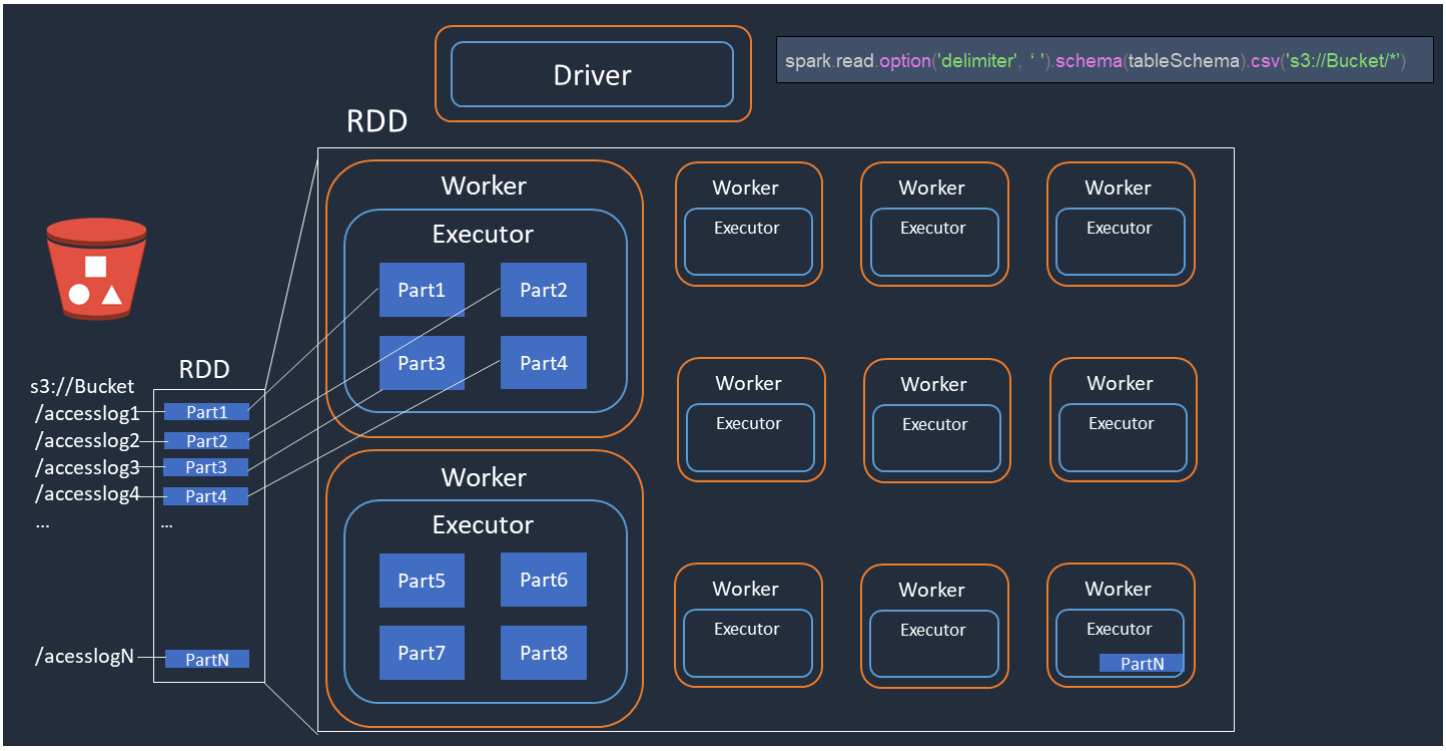


> **Note**
>
> 병렬화를 최적화할 경우 고려해야 할 가장 중요한 것은 태스크입니다. RDD 수에 따라 조정되는 태스크 수

Parallelism

Spark는 데이터 로드 및 변환 태스크를 병렬 처리합니다.

Amazon S3에서 액세스 로그 파일(`accesslog1 ... accesslogN`)의 분산 처리를 수행하는 예제를 고려합니다. 다음 다이어그램에서는 분산 처리 흐름을 보여줍니다.

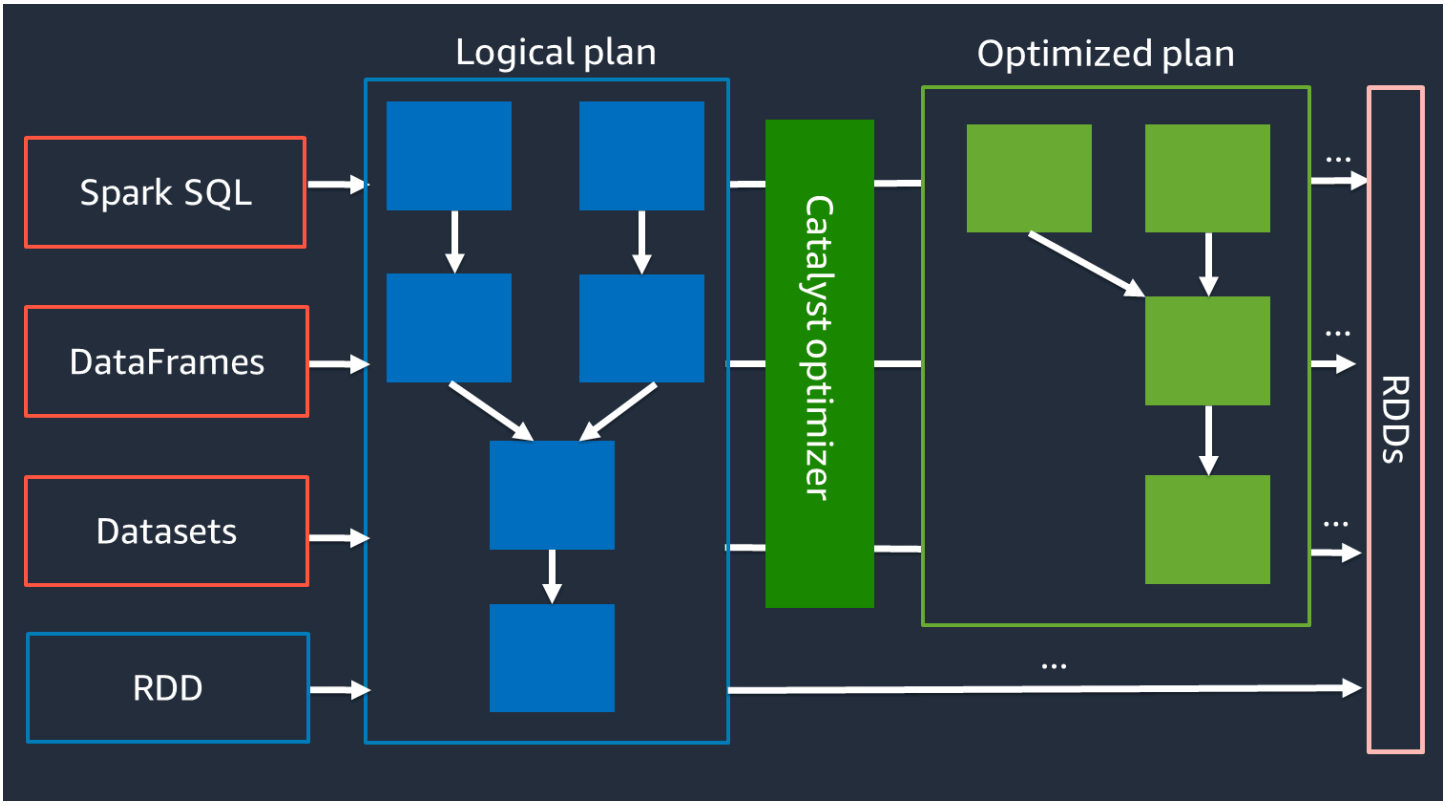


1. Spark 드라이버는 여러 Spark 실행기에서 분산 처리를 위한 실행 계획을 생성합니다.
2. Spark 드라이버는 실행 계획에 따라 각 실행기에 태스크를 할당합니다. 기본적으로 Spark 드라이버는 각 S3 객체(`Part1 ... N`)에 대해 RDD 파티션(각각 Spark 태스크에 해당)을 생성합니다. 그런 다음 Spark 드라이버는 각 실행기에 태스크를 할당합니다.
3. 각 Spark 태스크는 할당된 S3 객체를 다운로드하여 RDD 파티션의 메모리에 저장합니다. 이 방식으로 여러 Spark 실행기가 할당된 태스크를 병렬로 다운로드하고 처리합니다.

초기 파티션 수 및 최적화에 대한 자세한 내용은 태스크 병렬화 섹션을 참조하세요.

Catalyst 옵티마이저

내부적으로 Spark는 Catalyst 옵티마이저라는 엔진을 사용하여 실행 계획을 최적화합니다. Catalyst에는 다음 다이어그램에 설명된 대로 Spark SQL, DataFrame 및 Datasets와 같은 상위 수준 Spark API를 실행할 때 사용할 수 있는 쿼리 옵티마이저가 있습니다.



Catalyst 옵티마이저는 RDD API에서 직접 작동하지 않으므로 상위 수준 API는 일반적으로 하위 수준 RDD API보다 빠릅니다. 복잡한 조인의 경우 Catalyst 옵티마이저는 작업 실행 계획을 최적화하여 성능을 크게 개선할 수 있습니다. Spark UI의 SQL 탭에서 Spark 작업의 최적화된 계획을 볼 수 있습니다.

적응형 쿼리 실행

Catalyst 최적화 프로그램은 적응형 쿼리 실행이라는 프로세스를 통해 런타임 최적화를 수행합니다. 적응형 쿼리 실행은 런타임 통계를 사용하여 작업이 실행되는 동안 쿼리의 실행 계획을 다시 최적화합니다. 적응형 쿼리 실행은 다음 섹션에 설명된 대로 셔플 후 파티션 병합, 정렬 병합 조인을 브로드캐스트 조인으로 변환, 스큐 조인 최적화 등 성능 문제에 대한 여러 솔루션을 제공합니다.

적응형 쿼리 실행은 AWS Glue 3.0 이상에서 사용할 수 있으며 기본적으로 AWS Glue 4.0(Spark 3.3.0) 이상에서 활성화됩니다. 코드에서 `spark.conf.set("spark.sql.adaptive.enabled", "true")`를 사용하여 적응형 쿼리 실행을 켜거나 끌 수 있습니다.

사후 셔플 파티션 병합

이 기능은 map 출력 통계를 기반으로 각 셔플 후 RDD 파티션(병합)을 줄입니다. 쿼리를 실행할 때 셔플 파티션 번호의 조정을 단순화합니다. 데이터세트에 맞게 셔플 파티션 번호를 설정할 필요가 없습니다. Spark는 초기 셔플 파티션 수가 충분히 많으면 런타임에 적절한 셔플 파티션 번호를 선택할 수 있습니다.

`spark.sql.adaptive.enabled` 및 `spark.sql.adaptive.coalescePartitions.enabled`가 모두 true로 설정된 경우 셔플 후 파티션 병합이 활성화됩니다. 자세한 내용은 Apache Spark 설명서를 참조하세요.

정렬 병합 조인을 브로드캐스트 조인으로 변환

이 기능은 크기가 상당히 다른 두 데이터세트를 조인하는 시기를 인식하고 해당 정보를 기반으로 보다 효율적인 조인 알고리즘을 채택합니다. 자세한 내용은 Apache Spark 설명서를 참조하세요. 조인 전략은 셔플 최적화 섹션에서 설명합니다.

스큐 조인 최적화

데이터 스큐는 Spark 작업에서 가장 일반적인 병목 현상 중 하나입니다. 데이터가 특정 RDD 파티션(따라서 특정 태스크)으로 스큐되어 애플리케이션의 전체 처리 시간이 지연되는 상황을 설명합니다. 이 경우 종종 조인 작업의 성능이 저하될 수 있습니다. 스큐 조인 최적화 기능은 스큐된 태스크를 대략적으로 균등한 크기의 태스크로 분할하고 필요한 경우 복제하여 정렬 병합 조인의 스큐를 동적으로 처리합니다.

이 기능은 `spark.sql.adaptive.skewJoin.enabled`가 true로 설정된 경우 활성화됩니다. 자세한 내용은 Apache Spark 설명서를 참조하세요. 데이터 스큐는 셔플 최적화 섹션에서 자세히 설명합니다.

Spark UI를 사용하여 성능 문제 조사

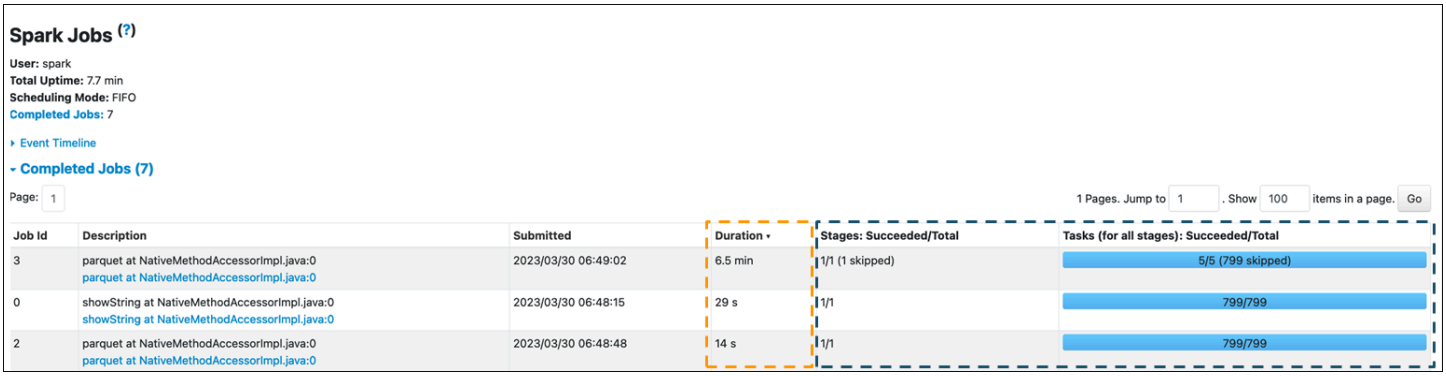
모범 사례를 적용하여 AWS Glue 작업의 성능을 조정하기 전에 성능을 프로파일링하고 병목 현상을 식별하는 것이 좋습니다. 이렇게 하면 올바른 업무에 집중할 수 있습니다.

빠른 분석을 위해 Amazon CloudWatch 지표는 작업 지표의 기본 보기를 제공합니다. Spark UI는 성능 조정을 위한 심층 보기를 제공합니다. 에서 Spark UI를 사용하려면 AWS Glue 작업에 대해 Spark UI를 활성화 AWS Glue해야 합니다. Spark UI에 익숙해지면 Spark 작업 성능을 조정하는 전략에 따라 조사 결과를 기반으로 병목 현상의 영향을 식별하고 줄입니다.

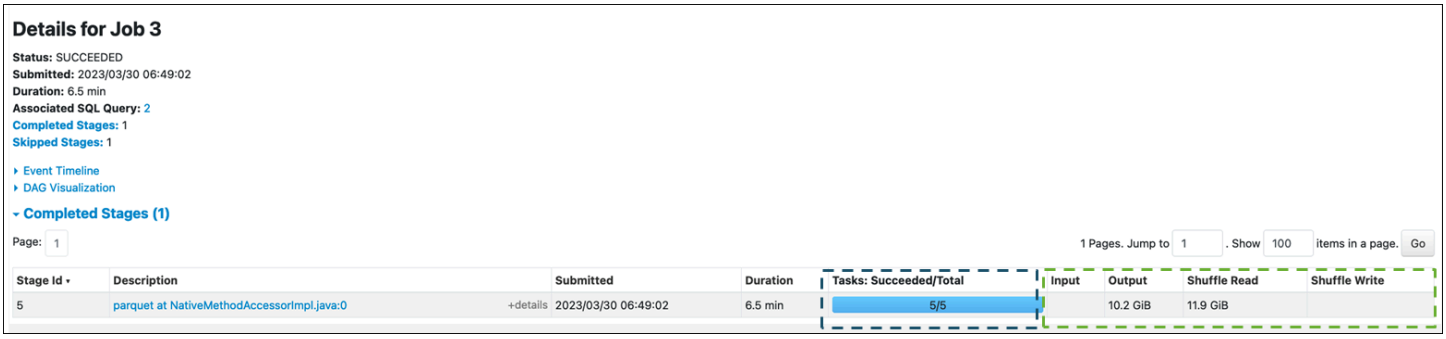
Spark UI를 사용하여 병목 현상 식별

Spark UI를 열면 Spark 애플리케이션이 테이블에 나열됩니다. 기본적으로 AWS Glue 작업의 앱 이름은 `nativespark-<Job Name>-<Job Run ID>`입니다. 작업 실행 ID를 기반으로 대상 Spark 앱을 선택하여 작업 탭을 엽니다. 스트리밍 작업 실행과 같은 불완전한 작업 실행은 불완전한 애플리케이션 표시에 나열됩니다.

작업 탭에는 Spark 애플리케이션의 모든 작업에 대한 요약이 표시됩니다. 단계 또는 태스크 실패를 확인하려면 총 작업 수를 확인합니다. 병목 현상을 찾으려면 기간을 선택하여 정렬합니다. 설명 열에 표시된 링크를 선택하여 장기 실행 작업의 세부 정보를 자세히 살펴봅니다.



작업 세부 정보 페이지에는 단계가 나열됩니다. 이 페이지에서는 지속 시간, 성공한 태스크 수 및 총 태스크 수, 입력 및 출력 수, 셔플 읽기 및 셔플 쓰기 양과 같은 전체 인사이트를 볼 수 있습니다.



Details for Job 3

Status: SUCCEEDED
Submitted: 2023/03/30 06:49:02
Duration: 6.5 min
Associated SQL Query: 2
Completed Stages: 1
Skipped Stages: 1

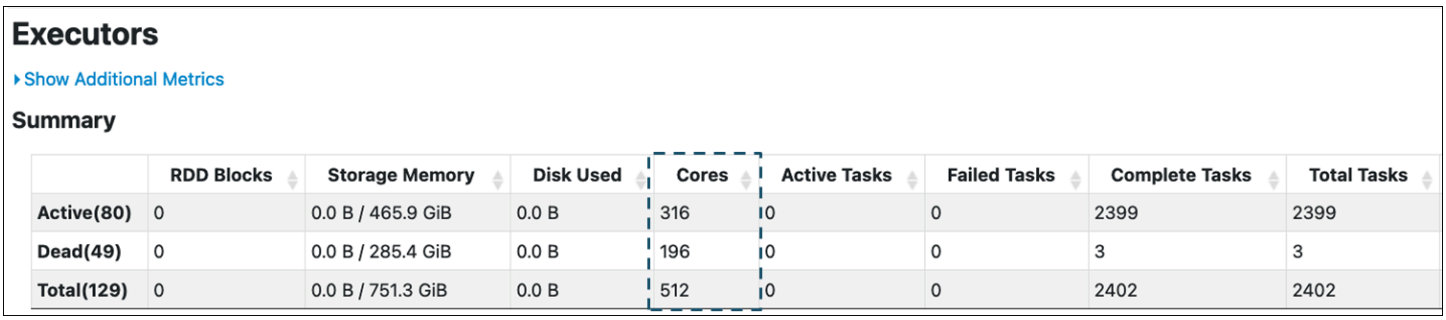
▸ Event Timeline
▸ DAG Visualization

▾ **Completed Stages (1)**

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
5	parquet at NativeMethodAccessorImpl.java:0 +details	2023/03/30 06:49:02	6.5 min	5/5		10.2 GiB	11.9 GiB	

실행기 탭에는 Spark 클러스터 용량이 자세히 표시됩니다. 총 코어 수를 확인할 수 있습니다. 다음 스크린샷에 표시된 클러스터에는 총 316개의 활성 코어와 512개의 코어가 포함되어 있습니다. 기본적으로 각 코어는 한 번에 하나의 Spark 태스크를 처리할 수 있습니다.



Executors

▸ Show Additional Metrics

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks
Active(80)	0	0.0 B / 465.9 GiB	0.0 B	316	0	0	2399	2399
Dead(49)	0	0.0 B / 285.4 GiB	0.0 B	196	0	0	3	3
Total(129)	0	0.0 B / 751.3 GiB	0.0 B	512	0	0	2402	2402

작업 세부 정보 페이지에 표시된 5/5 값에 따라 5단계가 가장 긴 단계이지만 512개 중 5개의 코어만 사용합니다. 이 단계의 병렬화는 너무 낮지만 시간이 많이 걸리기 때문에 병목 현상으로 식별할 수 있습니다. 성능을 개선하기 위해 이유를 파악하길 원합니다. 일반적인 성능 병목 현상의 영향을 인식하고 줄이는 방법에 대해 자세히 알아보려면 Spark 작업 성능 조정을 위한 전략을 참조하세요.

Spark 작업 성능 조정을 위한 전략

파라미터를 조정할 준비를 할 때는 다음과 같은 모범 사례를 고려하세요.

- 문제를 식별하기 전에 성능 목표를 결정합니다.
- 조정 파라미터를 변경하기 전에 지표를 사용하여 문제를 식별합니다.

작업을 조정할 때 가장 일관된 결과를 얻으려면 조정 작업에 대한 기준선 전략을 수립하세요.

성능 조정을 위한 기준선 전략

일반적으로 성능 조정은 다음 워크플로우에서 수행됩니다.

1. 성능 목표를 결정합니다.
2. 지표를 측정합니다.
3. 병목 현상을 식별합니다.
4. 병목 현상이 미치는 영향을 줄입니다.
5. 의도한 목표를 달성할 때까지 2~4단계를 반복합니다.

먼저 성능 목표를 결정합니다. 예를 들어 목표 중 하나는 3시간 이내에 AWS Glue 작업 실행을 완료하는 것일 수 있습니다. 목표를 정의한 후 작업 성능 지표를 측정합니다. 목표를 달성하기 위해 지표 및 병목 현상의 추세를 식별합니다. 특히 병목 현상 식별은 문제 해결, 디버깅 및 성능 조정에서 가장 중요합니다. Spark 애플리케이션을 실행하는 동안 Spark는 Spark 이벤트 로그에 각 태스크의 상태 및 통계를 기록합니다.

에서는 Spark 기록 서버에서 제공하는 Spark 웹 UI를 통해 Spark 지표를 볼 AWS Glue수 있습니다. AWS Glue for Spark 작업은 Spark 이벤트 로그를 Amazon S3에서 지정한 위치로 전송할 수 있습니다. AWS Glue 또한 Amazon EC2 인스턴스 또는 로컬 컴퓨터에서 Spark 기록 서버를 시작하기 위한 예제 AWS CloudFormation 템플릿 및 Dockerfile을 제공하므로 이벤트 로그와 함께 Spark UI를 사용할 수 있습니다.

성능 목표를 결정하고 이러한 목표를 평가하기 위한 지표를 식별한 후에 다음 섹션의 전략을 사용하여 병목 현상을 식별하고 해결할 수 있습니다.

Spark 작업 성능에 대한 조정 사례

Spark 작업에 대한 성능 튜닝에 다음 전략을 사용할 수 AWS Glue 있습니다.

- AWS Glue 리소스:
  - 클러스터 용량 규모 조정
  - 최신 AWS Glue 버전 사용
- Spark 애플리케이션:
  - 데이터 스캔 양 감소
  - 태스크 병렬화
  - 셔플 최적화
  - 계획 오버헤드 최소화
  - 사용자 정의 함수 최적화

이러한 전략을 사용하기 전에 Spark 작업에 대한 지표 및 구성에 액세스할 수 있어야 합니다. AWS Glue 설명서에서 이 정보를 확인할 수 있습니다.

AWS Glue 리소스 관점에서 AWS Glue 작업자를 추가하고 최신 AWS Glue 버전을 사용하여 성능을 개선할 수 있습니다.

Apache Spark 애플리케이션 관점에서 성능을 개선할 수 있는 여러 전략에 액세스할 수 있습니다. 불필요한 데이터가 Spark 클러스터에 로드되는 경우 이를 제거하여 로드된 데이터의 양을 줄일 수 있습니다. Spark 클러스터 리소스를 적게 사용하고 데이터 I/O가 적은 경우 병렬화할 태스크를 식별할 수 있습니다. 또한 상당한 시간이 걸리는 경우 조인과 같은 과중한 데이터 전송 작업을 최적화할 수 있습니다. 작업 쿼리 계획을 최적화하거나 개별 Spark 태스크의 계산 복잡성을 줄일 수도 있습니다.

이러한 전략을 효율적으로 적용하려면 지표를 참조하여 적용 가능한 시기를 식별해야 합니다. 자세한 내용은 다음의 각 섹션을 참조하세요. 이러한 기법은 성능 조정뿐만 아니라 메모리 부족(OOM) 오류와 같은 일반적인 문제를 해결하는 데도 효과적입니다.

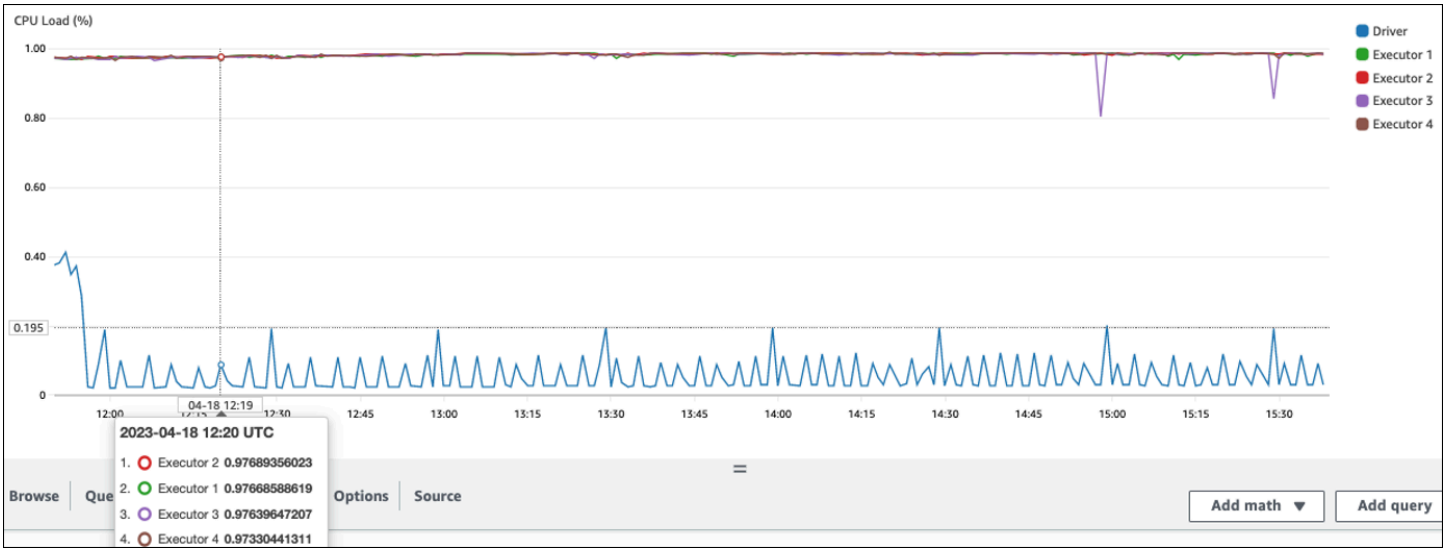
클러스터 용량 규모 조정

작업이 너무 오래 걸리지만 실행기가 충분한 리소스를 소비하고 Spark가 사용 가능한 코어에 비해 대량의 태스크를 생성하는 경우 클러스터 용량을 조정하는 방법을 고려합니다. 적절한지 평가하려면 다음 지표를 사용합니다.

CloudWatch 지표

- CPU 로드 및 메모리 사용률을 확인하여 실행기가 충분한 리소스를 사용하고 있는지 확인합니다.
- 작업 실행 시간을 확인하여 처리 시간이 너무 길어 성능 목표를 달성할 수 없는지 평가합니다.

다음 예제에서는 4개의 실행기가 97%를 초과하는 CPU 로드에서 실행되고 있지만 약 3시간 후에도 처리가 완료되지 않습니다.

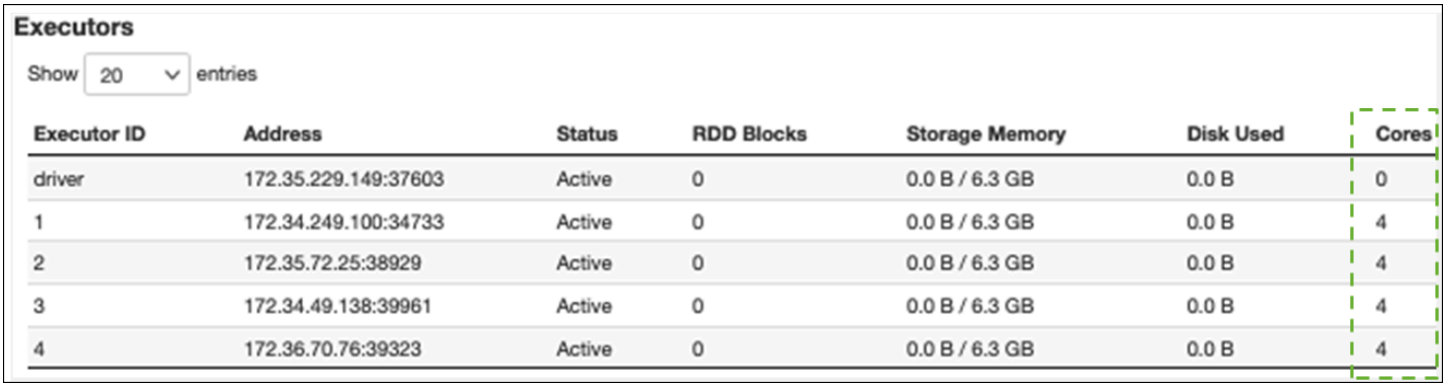


> **Note**
>
> CPU 부하가 낮으면 클러스터 용량을 조정해도 도움이 되지 않을 수 있습니다.

Spark UI

작업 탭 또는 단계 탭에서 각 작업 또는 단계의 태스크 수를 볼 수 있습니다. 다음 예제에서 Spark는 `58100` 태스크를 생성했습니다.

실행기 탭에서 총 실행기 및 태스크 수를 볼 수 있습니다. 다음 스크린샷에서 각 Spark 실행기에는 4개의 코어가 있으며 4개의 태스크를 동시에 수행할 수 있습니다.



Executors

Show 20 entries

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores
driver	172.35.229.149:37603	Active	0	0.0 B / 6.3 GB	0.0 B	0
1	172.34.249.100:34733	Active	0	0.0 B / 6.3 GB	0.0 B	4
2	172.35.72.25:38929	Active	0	0.0 B / 6.3 GB	0.0 B	4
3	172.34.49.138:39961	Active	0	0.0 B / 6.3 GB	0.0 B	4
4	172.36.70.76:39323	Active	0	0.0 B / 6.3 GB	0.0 B	4

이 예제에서 Spark 태스크 수(58100)는 실행기가 동시에 처리할 수 있는 16개의 태스크(4개의 실행기 × 4개의 코어)보다 훨씬 큽니다.

이러한 증상이 관찰되면 클러스터 규모 조정을 고려합니다. 다음 옵션을 사용하여 클러스터 용량을 조정할 수 있습니다.

- Enable AWS Glue Auto Scaling - Auto Scaling은 AWS Glue 버전 3.0 이상의 AWS Glue 추출, 변환, 로드(ETL) 및 스트리밍 작업에 사용할 수 있습니다.는 각 단계의 파티션 수 또는 작업 실행 시 마이크로배치가 생성되는 속도에 따라 클러스터에서 작업자를 AWS Glue 자동으로 추가하고 제거합니다.

  오토 스케일링이 활성화되어 있더라도 작업자 수가 증가하지 않는 상황이 관찰되면 작업자를 수동으로 추가하는 방법을 고려합니다. 그러나 한 단계에 대해 수동으로 규모를 조정하면 이후 단계에서 많은 작업자가 유휴 상태가 되어 성능이 개선되지 않으면서 비용이 더 많이 들 수 있습니다.

  오토 스케일링을 활성화하면 CloudWatch 실행기 지표에서 실행기 수를 볼 수 있습니다. 다음 지표를 사용하여 Spark 애플리케이션의 실행기 수요를 모니터링합니다.

  - `glue.driver.ExecutorAllocationManager.executors.numberAllExecutors`
  - `glue.driver.ExecutorAllocationManager.executors.numberMaxNeededExecutors`

  지표에 대한 자세한 내용은 Amazon CloudWatch 지표 AWS Glue 를 사용한 모니터링을 참조하세요.
- 스케일 아웃: AWS Glue 작업자 수 증가 - AWS Glue 작업자 수를 수동으로 늘릴 수 있습니다. 유휴 작업자를 관찰할 때까지만 작업자를 추가합니다. 이때 작업자를 더 추가하면 결과가 개선되지 않고 비용이 증가합니다. 자세한 내용은 태스크 병렬화를 참조하세요.
- 스케일 업: 더 큰 작업자 유형 사용 - 코어, 메모리 및 스토리지가 더 많은 작업자를 사용하도록 AWS Glue 작업자의 인스턴스 유형을 수동으로 변경할 수 있습니다. 작업자 유형이 클수록 메모리 집약적인 데이터 변환, 왜곡된 집계, 페타바이트 규모의 데이터와 관련된 엔터티 감지 검사 등 집약적인 데이터 통합 작업을 수직적으로 스케일링하고 실행할 수 있습니다.

작업 쿼리 계획이 상당히 크기 때문에 Spark 드라이버에 더 큰 용량이 필요한 경우에도 스케일 업이 도움이 됩니다. 작업자 유형 및 성능에 대한 자세한 내용은 AWS 빅 데이터 블로그 게시물 새로운 대형 작업자 유형 G.4X 및 G.8X로 Apache Spark 작업에 AWS Glue 맞게 조정을 참조하세요.

또한 더 큰 작업자를 사용하면 필요한 총 작업자 수를 줄일 수 있으므로 조인과 같은 집약적인 작업에서 셔플을 줄여 성능이 개선됩니다.

최신 AWS Glue 버전 사용

최신 AWS Glue 버전을 사용하는 것이 좋습니다. 작업 성능을 자동으로 개선할 수 있는 여러 최적화 및 업그레이드가 각 버전에 빌드되어 있습니다. 예를 들어 AWS Glue 4.0은 다음과 같은 새로운 기능을 제공합니다.

- 새로 최적화된 Apache Spark 3.3.0 런타임 - AWS Glue 4.0은 Apache Spark 3.3.0 런타임을 기반으로 구축되어 오픈 소스 Spark에 비교 가능한 성능 개선을 제공합니다. Spark 3.3.0 런타임은 Spark 2.x의 많은 혁신을 기반으로 빌드되었습니다.
- 향상된 Amazon Redshift 커넥터 - AWS Glue 4.0 이상 버전은 Apache Spark에 Amazon Redshift 통합을 제공합니다. 통합은 기존 오픈 소스 커넥터를 기반으로 빌드되며 성능과 보안을 위해 기능을 개선합니다. 통합을 통해 애플리케이션 성능이 최대 10배 빨라집니다. 자세한 내용은 Amazon Redshift integration with Apache Spark에 대한 블로그 게시물을 참조하세요.
- CSV 및 JSON 데이터를 사용한 벡터화된 읽기에 대한 SIMD 기반 실행 - AWS Glue 버전 3.0 이상 버전은 행 기반 리더에 비해 전체 작업 성능을 크게 높일 수 있는 최적화된 리더를 추가합니다. CSV 데이터에 대한 자세한 내용은 Optimize read performance with vectorized SIMD CSV reader를 참조하세요. JSON 데이터에 대한 자세한 내용은 Using vectorized SIMD JSON reader with Apache Arrow columnar format을 참조하세요.

각 AWS Glue 버전에는 커넥터, 드라이버 및 라이브러리 업데이트를 포함하여 이러한 정렬의 업그레이드가 포함됩니다. 자세한 내용은 AWS Glue 버전 및 버전 4.0으로 AWS GlueAWS Glue 작업 마이그레이션을 참조하세요.

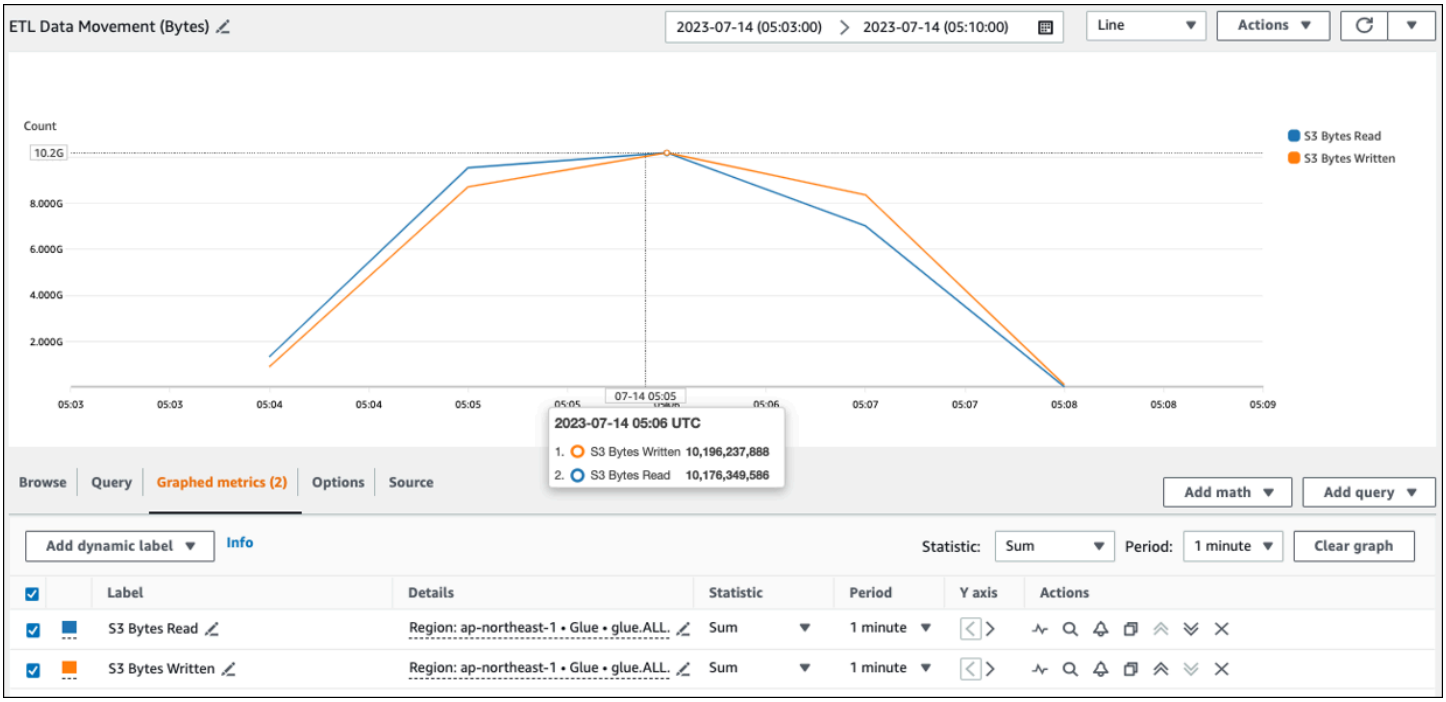
데이터 스캔 양 감소

시작할 때 필요한 데이터만 로드하는 방법을 고려합니다. 각 데이터 소스에 대해 Spark 클러스터에 로드되는 데이터의 양을 줄임으로써 성능을 개선할 수 있습니다. 이 접근 방식이 적절한지 평가하려면 다음 지표를 사용합니다.

Spark UI 섹션에 설명된 대로 CloudWatch 지표에서 Amazon S3의 읽기 바이트 및 Spark UI에서 추가 세부 정보를 확인할 수 있습니다.

CloudWatch 지표

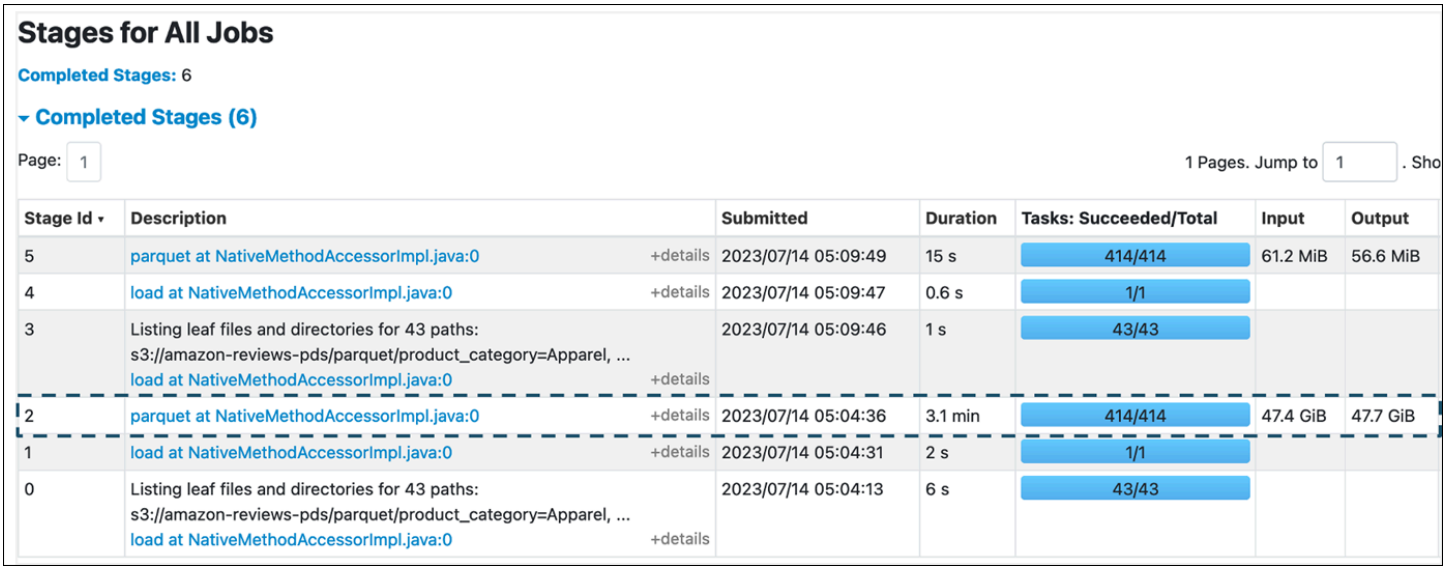
ETL 데이터 이동(바이트)에서 Amazon S3의 대략적인 읽기 크기를 확인할 수 있습니다. 이 지표는 이전 보고서 이후 모든 실행기가 Amazon S3에서 읽은 바이트 수를 보여줍니다. 이를 사용하여 Amazon S3의 ETL 데이터 이동을 모니터링할 수 있으며, 읽기와 외부 데이터 소스의 수집 속도를 비교할 수 있습니다.



예상보다 큰 S3 바이트 읽기 데이터 포인트가 관찰되는 경우 다음 솔루션을 고려합니다.

Spark UI

Spark UI AWS Glue 용의 스테이지 탭에서 입력 및 출력 크기를 볼 수 있습니다. 다음 예제의 경우 2단계에서는 47.4GiB 입력 및 47.7GiB 출력을 읽는 반면, 5단계에서는 61.2MiB 입력 및 56.6MiB 출력을 읽습니다.



Stages for All Jobs

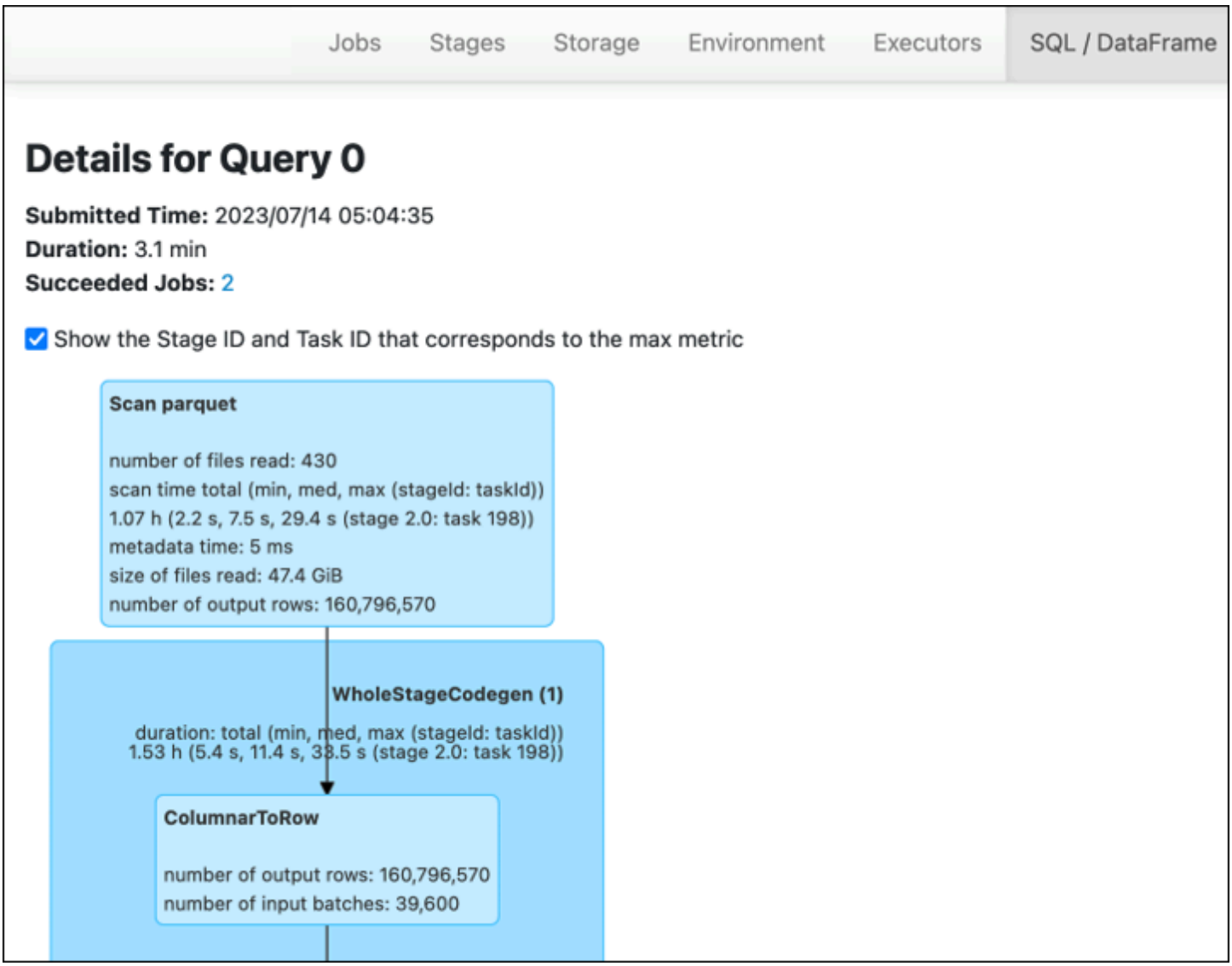
Completed Stages: 6

Completed Stages (6)

Page: 1 1 Pages. Jump to 1 . Sho

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output
5	parquet at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:09:49	15 s	414/414	61.2 MiB	56.6 MiB
4	load at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:09:47	0.6 s	1/1		
3	Listing leaf files and directories for 43 paths: s3://amazon-reviews-pds/parquet/product_category=Apparel, ... load at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:09:46	1 s	43/43		
2	parquet at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:04:36	3.1 min	414/414	47.4 GiB	47.7 GiB
1	load at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:04:31	2 s	1/1		
0	Listing leaf files and directories for 43 paths: s3://amazon-reviews-pds/parquet/product_category=Apparel, ... load at NativeMethodAccessorImpl.java:0 +details	2023/07/14 05:04:13	6 s	43/43		

AWS Glue 작업에서 Spark SQL 또는 DataFrame 접근 방식을 사용하면 SQL / DataFrame 탭에 이러한 단계에 대한 더 많은 통계가 표시됩니다. 이 경우 2단계에서 읽은 파일 수: 430, 읽은 파일 크기: 47.4GiB, 출력 행 수: 160,796,570이 표시됩니다.



읽고 있는 데이터와 사용 중인 데이터 사이에서 크기에 상당한 차이가 있는 경우 다음 솔루션을 시도해 보세요.

Amazon S3

Amazon S3에서 읽을 때 작업에 로드되는 데이터의 양을 줄이려면 데이터 세트의 파일 크기, 압축, 파일 형식 및 파일 레이아웃(파티션) 을 고려하세요. AWS Glue Spark 작업은 원시 데이터의 ETL에 자주 사용되지만 효율적인 분산 처리를 위해서는 데이터 소스 형식의 기능을 검사해야 합니다.

- 파일 크기 - 입력 및 출력의 파일 크기를 중간 범위(예: 128MB) 이내로 유지하는 것이 좋습니다. 너무 작은 파일과 너무 큰 파일은 문제가 될 수 있습니다.

  작은 파일이 많으면 다음과 같은 문제가 발생합니다.

- 많은 객체에 대한 요청(예: `List`, `Get` 또는 `Head`)을 수행하는 데 필요한 오버헤드로 인한 Amazon S3에서 과도한 네트워크 I/O 로드(동일한 양의 데이터를 저장하는 몇 개의 객체와 비교한 경우).
- Spark 드라이버에서 과도한 I/O 및 처리 로드가 발생하여 많은 파티션과 태스크가 생성되고 과도한 병렬 처리가 발생합니다.

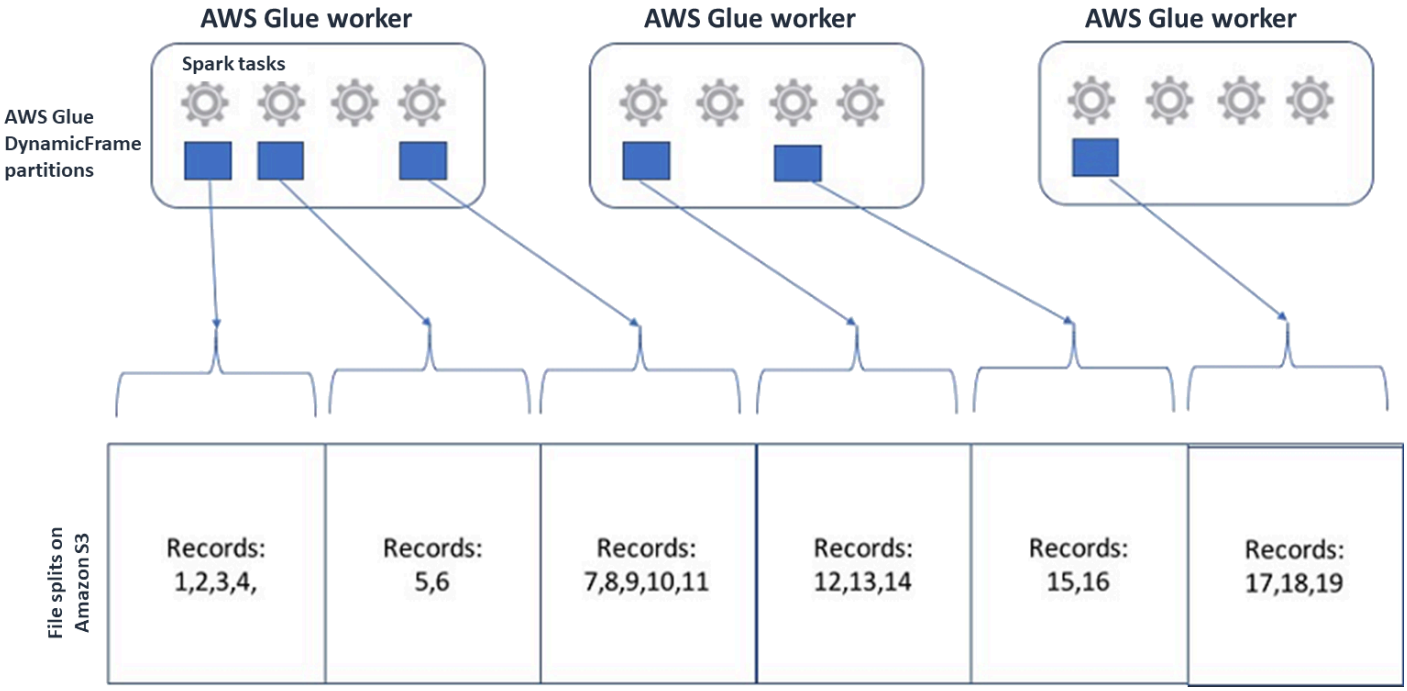
반면 파일 유형이 분할 가능하지 않고(예: gzip) 파일이 너무 큰 경우 단일 태스크가 전체 파일 읽기를 완료할 때까지 Spark 애플리케이션이 기다려야 합니다.

각 작은 파일에 대해 Apache Spark 태스크가 생성될 때 발생하는 과도한 병렬 처리를 줄이려면 DynamicFrames에 파일 그룹화를 사용합니다. 이 접근 방식은 Spark 드라이버에서 OOM 예외가 발생할 가능성을 줄입니다. 파일 그룹화를 구성하려면 `groupFiles` 및 `groupSize` 파라미터를 설정합니다. 다음 코드 예제에서는 이러한 파라미터와 함께 ETL 스크립트에서 AWS Glue DynamicFrame API를 사용합니다.

```
dyf = glueContext.create_dynamic_frame_from_options("s3",
    {'paths': ["s3://input-s3-path/"],
    'recurse':True,
    'groupFiles': 'inPartition',
    'groupSize': '1048576'},
    format="json")
```

- 압축 - S3 객체가 수백 메가바이트인 경우 압축하는 것이 좋습니다. 다양한 압축 형식이 있으며, 크게 두 가지 유형으로 분류할 수 있습니다.
  - gzip과 같은 분할할 수 없는 압축 형식을 사용하려면 한 작업자가 전체 파일의 압축을 풀어야 합니다.
  - bzip2 또는 LZO(인덱스)와 같은 분할 가능한 압축 형식은 병렬화할 수 있는 파일의 부분 압축 해제를 허용합니다.

  Spark 및 기타 일반적인 분산 처리 엔진의 경우 소스 데이터 파일을 엔진이 병렬로 처리할 수 있는 청크로 분할합니다. 이러한 단위를 종종 분할이라고 합니다. 데이터가 분할 가능한 형식이면 최적화된 AWS Glue 리더는 `GetObject` API에 특정 블록만 검색하는 `Range` 옵션을 제공하여 S3 객체에서 분할을 검색할 수 있습니다. 다음 다이어그램을 고려하여 실제로 어떻게 작동하는지 확인합니다.



압축된 데이터는 파일이 최적의 크기이거나 파일이 분할 가능한 한 애플리케이션의 속도를 크게 높일 수 있습니다. 데이터 크기가 작을수록 Amazon S3에서 스캔한 데이터와 Amazon S3에서 Spark 클러스터로의 네트워크 트래픽이 줄어듭니다. 반면 데이터를 압축 및 압축 해제하려면 더 많은 CPU가 필요합니다. 필요한 컴퓨팅 양은 압축 알고리즘의 압축 비율에 따라 조정됩니다. 분할 가능한 압축 형식을 선택할 경우 이 장단점을 고려합니다.

> **Note**
>
> gzip 파일은 일반적으로 분할할 수 없지만 개별 Parquet 블록을 gzip으로 압축할 수 있으며 이러한 블록은 병렬화할 수 있습니다.

- 파일 형식 - 열 형식을 사용합니다. Apache Parquet 및 Apache ORC는 널리 사용되는 열 기반 데이터 형식입니다. Parquet 및 ORC는 열 기반 압축을 사용하고 데이터 유형에 따라 각 열을 인코딩 및 압축하여 데이터를 효율적으로 저장합니다. Parquet 인코딩에 대한 자세한 내용은 Parquet encoding definitions를 참조하세요. Parquet 파일도 분할할 수 있습니다.

  열 형식은 열별로 값을 그룹화하고 블록에 함께 저장합니다. 열 기반 형식을 사용하는 경우 사용할 계획이 없는 열에 해당하는 데이터 블록을 건너뛸 수 있습니다. Spark 애플리케이션에서는 필요한 열만 검색할 수 있습니다. 일반적으로 압축률이 높거나 데이터 블록을 건너뛰면 Amazon S3에서 바이트를 더 적게 읽어 성능이 향상됩니다. 두 형식 모두 I/O를 줄이기 위해 다음과 같은 푸시다운 접근 방식도 지원합니다.

- 프로젝션 푸시다운 - 프로젝션 푸시다운은 애플리케이션에 지정된 열만 검색하는 기법입니다. 다음 예제와 같이 Spark 애플리케이션에서 열을 지정합니다.
  - DataFrame 예제: `df.select("star_rating")`
  - Spark SQL 예제: `spark.sql("select start_rating from <table>")`
- 조건자 푸시다운 - 조건자 푸시다운은 `WHERE` 및 `GROUP BY` 절을 효율적으로 처리하는 기법입니다. 두 형식 모두에는 열 값을 나타내는 데이터 블록이 있습니다. 각 블록에는 최댓값 및 최솟값과 같은 블록에 대한 통계가 들어 있습니다. Spark는 이러한 통계를 사용하여 애플리케이션에 사용되는 필터 값에 따라 블록을 읽거나 건너뛸지를 결정할 수 있습니다. 이 기능을 사용하려면 다음과 같이 다음 예제에 나온 대로 조건에 필터를 더 추가합니다.
  - DataFrame 예제: `df.select("star_rating").filter("star_rating < 2")`
  - Spark SQL 예제: `spark.sql("select * from <table> where star_rating < 2")`
- 파일 레이아웃 - 데이터 사용 방식에 따라 S3 데이터를 다른 경로의 객체에 저장하면 관련 데이터를 효율적으로 검색할 수 있습니다. 자세한 내용은 Amazon S3 설명서의 접두사를 사용한 객체 구성을 참조하세요. AWS Glue 는 `key=value` 형식의 Amazon S3 접두사에 키와 값을 저장하고 Amazon S3 경로를 통해 데이터를 분할할 수 있도록 지원합니다. 데이터를 분할하면 각 다운스트림 분석 애플리케이션이 스캔하는 데이터의 양을 제한하여 성능을 향상시키고 비용을 절감할 수 있습니다. 자세한 내용은 의 ETL 출력에 대한 파티션 관리를 참조하세요 AWS Glue.

  분할은 테이블을 여러 부분으로 나누고 다음 예제와 같이 year, month, day 등의 열 값을 기반으로 관련 데이터를 그룹화된 파일에 보관합니다.

```
# Partitioning by /YYYY/MM/DD
s3://<YourBucket>/year=2023/month=03/day=31/0000.gz
s3://<YourBucket>/year=2023/month=03/day=01/0000.gz
s3://<YourBucket>/year=2023/month=03/day=02/0000.gz
s3://<YourBucket>/year=2023/month=03/day=03/0000.gz
...
```

  AWS Glue Data Catalog의 테이블로 데이터세트를 모델링하여 데이터세트의 파티션을 정의할 수 있습니다. 그런 다음 다음과 같이 파티션 정리를 사용하여 데이터 스캔 양을 제한할 수 있습니다.

  - For AWS Glue DynamicFrame은 `push_down_predicate` (또는)를 설정합니다`catalogPartitionPredicate`.

```
dyf = Glue_context.create_dynamic_frame.from_catalog(
    database=src_database_name,
    table_name=src_table_name,
    push_down_predicate = "year='2023' and month ='03'",
```

```
)
```

- Spark DataFrame의 경우 파티션을 정리할 고정 경로를 설정합니다.

```
df = spark.read.format("json").load("s3://<YourBucket>/year=2023/month=03/*/*.gz")
```

- Spark SQL의 경우 데이터 카탈로그에서 파티션을 정리하도록 where 절을 설정할 수 있습니다.

```
df = spark.sql("SELECT * FROM <Table> WHERE year= '2023' and month = '03'")
```

- 데이터를 쓸 때 날짜별로 파티셔 AWS Glue닝하려면 다음과 같이 열의 날짜 정보로 DataFrame의 DynamicFrame 또는 partitionBy()에 partitionKeys를 설정합니다.
  - DynamicFrame

```
glue_context.write_dynamic_frame_from_options(
    frame= dyf, connection_type='s3',format='parquet'
    connection_options= {
        'partitionKeys': ["year", "month", "day"],
        'path': 's3://<YourBucket>/<Prefix>/'
    }
)
```

  - DataFrame

```
df.write.mode('append')\
    .partitionBy('year','month','day')\
    .parquet('s3://<YourBucket>/<Prefix>/')
```

이렇게 하면 출력 데이터의 소비자 성능이 향상될 수 있습니다.

입력 데이터세트를 생성하는 파이프라인을 변경할 수 있는 액세스 권한이 없는 경우 분할은 옵션이 아닙니다. 대신 glob 패턴을 사용하여 불필요한 S3 경로를 제외할 수 있습니다. DynamicFrame에서 읽을 때 제외를 설정합니다. 예를 들어 다음 코드는 2023년 1월~9월의 일수를 제외합니다.

```
dyf = glueContext.create_dynamic_frame.from_catalog(
    database=db,
    table_name=table,
    additional_options = { "exclusions":"[\"**year=2023/month=0[1-9]/**\"]" },
    transformation_ctx='dyf'
)
```

Data Catalog의 테이블 속성에서 제외를 설정할 수도 있습니다.

- 키: `exclusions`
- 값: `["**year=2023/month=0[1-9]/**"]`

- Amazon S3 파티션이 너무 많음 - 값이 수천 개인 ID 열과 같이 광범위한 값을 포함하는 열에서 Amazon S3 데이터를 분할하지 마세요. 그러면 가능한 파티션 수는 파티션을 분할한 모든 필드의 곱이므로 버킷의 파티션 수가 크게 증가할 수 있습니다. 파티션이 너무 많으면 다음과 같은 상황이 나타날 수 있습니다.
  - Data Catalog에서 파티션 메타데이터를 검색할 경우 지연 시간 증가
  - 작은 파일 수 증가(더 많은 Amazon S3 API 요청(`List`, `Get`, `Head`)이 필요함)

  예를 들어 `partitionBy` 또는 `partitionKeys`에서 날짜 유형을 설정하면 `yyyy/mm/dd`와 같은 날짜 수준 분할이 많은 사용 사례에 적합합니다. 그러나 `yyyy/mm/dd/<ID>`는 너무 많은 파티션을 생성하여 전반적으로 성능에 부정적인 영향을 미칠 수 있습니다.

  반면 실시간 처리 애플리케이션과 같은 일부 사용 사례에는 `yyyy/mm/dd/hh`와 같은 많은 파티션이 필요합니다. 사용 사례에 상당한 파티션이 필요한 경우 AWS Glue 파티션 인덱스를 사용하여 Data Catalog에서 파티션 메타데이터를 검색하는 데 걸리는 지연 시간을 줄이는 방법을 고려합니다.

데이터베이스 및 JDBC

데이터베이스에서 정보를 검색할 때 데이터 스캔을 줄이려면 SQL 쿼리에서 `where` 조건자(또는 절)를 지정할 수 있습니다. SQL 인터페이스를 제공하지 않는 데이터베이스는 쿼리 또는 필터링을 위한 자체 메커니즘을 제공합니다.

Java Database Connectivity(JDBC) 연결을 사용하는 경우 다음 파라미터에 대한 `where` 절과 함께 선택 쿼리를 제공합니다.

- DynamicFrame의 경우 sampleQuery 옵션을 사용합니다. `create_dynamic_frame.from_catalog`를 사용할 때 다음과 같이 `additional_options` 인수를 구성합니다.

```
query = "SELECT * FROM <TableName> where id = 'XX' AND"
datasource0 = glueContext.create_dynamic_frame.from_catalog(
  database = db,
  table_name = table,
  additional_options={
      "sampleQuery": query,
```

```
      "hashexpression": key,
      "hashpartitions": 10,
      "enablePartitioningForSampleQuery": True
    },
  transformation_ctx = "datasource0"
)
```

using create_dynamic_frame.from_options에서 다음과 같이 connection_options 인수를 구성합니다.

```
query = "SELECT * FROM <TableName> where id = 'XX' AND"
datasource0 = glueContext.create_dynamic_frame.from_options(
    connection_type = connection,
    connection_options={
        "url": url,
        "user": user,
        "password": password,
        "dbtable": table,
        "sampleQuery": query,
        "hashexpression": key,
        "hashpartitions": 10,
        "enablePartitioningForSampleQuery": True
    }
)
```

- DataFrame의 경우 쿼리 옵션을 사용합니다.

```
query = "SELECT * FROM <TableName> where id = 'XX'"
jdbcDF = spark.read \
.format('jdbc') \
.option('url', url) \
.option('user', user) \
.option('password', pwd) \
.option('query', query) \
.load()
```

- Amazon Redshift의 경우 Amazon Redshift Spark 커넥터에서 푸시다운 지원을 활용하려면 AWS Glue 4.0 이상을 사용합니다.

```
dyf = glueContext.create_dynamic_frame.from_catalog(
    database = "redshift-dc-database-name",
    table_name = "redshift-table-name",
```

```
    redshift_tmp_dir = args["temp-s3-dir"],
    additional_options = {"aws_iam_role": "arn:aws:iam::role-account-id:role/rs-role-
name"}
)
```

- 다른 데이터베이스의 경우 해당 데이터베이스에 대한 설명서를 참조하세요.

AWS Glue 옵션

- 모든 연속 작업 실행에 대한 전체 스캔을 방지하고 마지막 작업 실행 중에 존재하지 않은 데이터만 처리하려면 작업 북마크를 활성화합니다.
- 처리할 입력 데이터의 양을 제한하려면 작업 북마크를 사용하여 제한된 실행을 활성화합니다. 이렇게 하면 각 작업 실행에 대해 스캔한 데이터의 양을 줄일 수 있습니다.

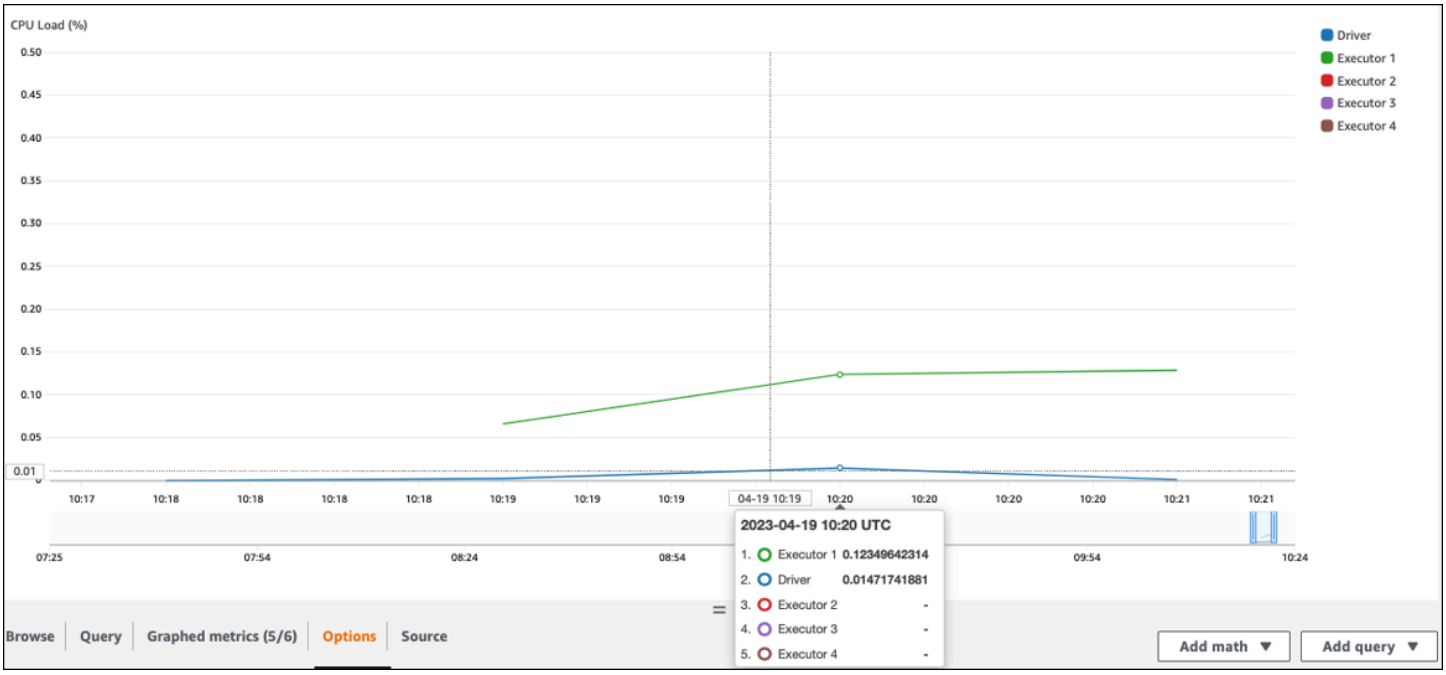
태스크 병렬화

성능을 최적화하려면 데이터 로드 및 변환에 대한 태스크를 병렬화하는 것이 중요합니다. Apache Spark의 주요 주제에서 설명한 것처럼 복원력이 뛰어난 분산 데이터세트(RDD) 파티션의 수는 병렬화 수준을 결정하기 때문에 중요합니다. Spark가 생성하는 각 태스크는 1:1 기준으로 RDD 파티션에 해당합니다. 최상의 성능을 얻으려면 RDD 파티션 수를 결정하는 방법과 해당 수를 최적화하는 방법을 이해해야 합니다.

병렬 처리가 충분하지 않은 경우 CloudWatch 지표 및 Spark UI에 다음 증상이 기록됩니다.

CloudWatch 지표

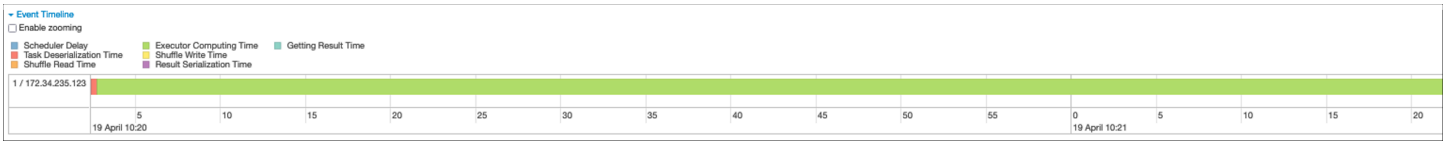
CPU 로드 및 메모리 사용률을 확인합니다. 작업 단계에서 일부 실행기가 처리 중이 아닌 경우 병렬 처리를 개선하는 것이 좋습니다. 이 경우 시각화된 기간에 실행기 1이 태스크를 수행했지만 나머지 실행기(2, 3, 4)는 수행하지 않았습니다. Spark 드라이버에서 해당 실행기에 태스크가 할당되지 않았다고 추론할 수 있습니다.



Spark UI

Spark UI의 Stage 탭에서 단계의 태스크 수를 볼 수 있습니다. 이 경우 Spark는 하나의 태스크만 수행했습니다.

또한 이벤트 타임라인에는 실행기 1이 하나의 태스크를 처리하는 것으로 표시됩니다. 즉, 이 단계의 작업은 하나의 실행기에서 완전히 수행되었고 다른 실행기는 유휴 상태였습니다.



이러한 증상이 관찰되면 각 데이터 소스에 대해 다음 솔루션을 시도해 보세요.

Amazon S3에서 데이터 로드 병렬화

Amazon S3에서 데이터 로드를 병렬화하려면 먼저 기본 파티션 수를 확인합니다. 그런 다음 대상 파티션 수를 수동으로 결정할 수 있지만 파티션이 너무 많지 않도록 해야 합니다.

기본 파티션 수 결정

Amazon S3의 경우 초기 Spark RDD 파티션 수(각각 Spark 태스크에 해당)는 Amazon S3 데이터세트의 기능(예: 형식, 압축 및 크기)에 따라 결정됩니다. Amazon S3에 저장된 CSV 객체에서 AWS Glue DynamicFrame 또는 Spark DataFrame을 생성하면 다음과 같이 초기 RDD 파티션(NumPartitions) 수를 대략적으로 계산할 수 있습니다.

- 객체 크기가 64MB 이하인 경우: `NumPartitions = Number of Objects`
- 객체 크기가 64MB를 초과하는 경우: `NumPartitions = Total Object Size / 64 MB`
- 분할 불가(gzip): `NumPartitions = Number of Objects`

데이터 스캔 양 감소 섹션에서 설명한 대로 Spark는 큰 S3 객체를 병렬로 처리할 수 있는 분할로 나눕니다. 객체가 분할 크기보다 크면 Spark는 객체를 분할하고 각 분할에 대해 RDD 파티션(및 태스크)을 생성합니다. Spark의 분할 크기는 데이터 형식과 런타임 환경에 기반하지만 이는 합리적인 시작 근사치입니다. 일부 객체는 gzip과 같은 분할할 수 없는 압축 형식을 사용하여 압축되므로 Spark는 이를 분할할 수 없습니다.

NumPartitions 값은 데이터 형식, 압축, AWS Glue 버전, AWS Glue 작업자 수 및 Spark 구성에 따라 다를 수 있습니다.

예를 들어 Spark DataFrame을 사용하여 단일 10GB `csv.gz` 객체를 로드하면 gzip은 분할할 수 없으므로 Spark 드라이버는 하나의 RDD 파티션(NumPartitions=1)만 생성합니다. 따라서 다음 그림에 설명된 대로 하나의 특정 Spark 실행기에 과도한 로드가 생기고 나머지 실행기에 태스크가 할당되지 않습니다.

Spark Web UI 단계 탭에서 단계의 실제 태스크 수(NumPartitions)를 확인하거나 코드에서 `df.rdd.getNumPartitions()`를 실행하여 병렬화를 확인합니다.

10GB gzip 파일이 생기면 해당 파일을 생성하는 시스템이 분할 가능한 형식으로 파일을 생성할 수 있는지 검사합니다. 옵션이 아닌 경우 클러스터 용량을 조정하여 파일을 처리해야 할 수 있습니다. 로드한 데이터에 대해 변환을 효율적으로 실행하려면 다시 분할을 사용하여 클러스터의 작업자 간에 RDD를 재조정해야 합니다.

수동으로 대상 파티션 수 확인

데이터의 속성과 Spark의 특정 기능 구현에 따라 기본 작업을 계속 병렬화할 수 있더라도 NumPartitions 값이 낮아질 수 있습니다. NumPartitions가 너무 작은 경우 `df.repartition(N)`을 실행하여 파티션 수를 늘리면 처리가 여러 Spark 실행기에 분산될 수 있습니다.

이 경우 df.repartition(100)을 실행하면 NumPartitions가 1에서 100으로 증가하고 각각 다른 실행기에 할당할 수 있는 태스크를 포함하는 100개의 데이터 파티션이 생성됩니다.

repartition(N) 작업은 전체 데이터를 균등하게 나누어(10GB/100개 파티션 = 100MB/파티션) 특정 파티션에 대한 데이터 스큐를 방지합니다.

> **Note**
>
> join과 같은 셔플 작업이 실행되면 spark.sql.shuffle.partitions 또는 spark.default.parallelism의 값에 따라 파티션 수가 동적으로 증가하거나 감소합니다. 이를 통해 Spark 실행기 사이에서 데이터를 보다 효율적으로 교환할 수 있습니다. 자세한 내용은 Spark 설명서를 참조하세요.

파티션의 목표 수를 결정할 때 목표는 프로비저닝된 AWS Glue 작업자의 사용을 극대화하는 것입니다. AWS Glue 작업자 수와 Spark 작업 수는 vCPUs. Spark는 각 vCPU 코어에 대해 하나의 태스크를 지원합니다. AWS Glue 버전 3.0 이상에서는 다음 공식을 사용하여 대상 파티션 수를 계산할 수 있습니다.

```
# Calculate NumPartitions by WorkerType
numExecutors = (NumberOfWorkers - 1)
numSlotsPerExecutor =
  4 if WorkerType is G.1X
  8 if WorkerType is G.2X
  16 if WorkerType is G.4X
  32 if WorkerType is G.8X
NumPartitions = numSlotsPerExecutor * numExecutors

# Example: Glue 4.0 / G.1X / 10 Workers
numExecutors = ( 10 - 1 ) = 9  # 1 Worker reserved on Spark Driver
numSlotsPerExecutor       = 4  # G.1X has 4 vCpu core ( Glue 3.0 or later )
NumPartitions = 9  * 4    = 36
```

이 예제에서 각 G.1X 작업자는 Spark 실행기(spark.executor.cores = 4)에 4개의 vCPU 코어를 제공합니다. Spark는 각 vCPU 코어에 대해 하나의 태스크를 지원하므로 G.1X Spark 실행기는 4개의 태스크(numSlotPerExecutor)를 동시에 실행할 수 있습니다. 태스크에 동일한 시간이 걸리는 경우 이 파티션 수는 클러스터를 완전히 사용합니다. 그러나 일부 태스크가 다른 태스크보다 오래 걸리므로 유휴 코어가 생성됩니다. 이 경우 병목 현상 태스크를 나누고 효율적으로 예약하려면 numPartitions에 2 또는 3을 곱하는 방법을 고려합니다.

파티션이 너무 많음

파티션 수가 너무 많으면 태스크 수가 너무 많습니다. 이로 인해 Spark 실행기 간의 관리 태스크 및 데이터 교환과 같은 분산 처리와 관련된 오버헤드 때문에 Spark 드라이버에 과부하가 발생합니다.

작업의 파티션 수가 목표 파티션 수보다 훨씬 큰 경우 파티션 수를 줄이는 방법을 고려합니다. 다음 옵션을 사용하여 파티션을 줄일 수 있습니다.

- 파일 크기가 매우 작은 경우 AWS Glue groupFiles를 사용합니다. Apache Spark 태스크 시작으로 인한 과도한 병렬화를 줄여 각 파일을 처리할 수 있습니다.
- `coalesce(N)`를 사용하여 파티션을 병합합니다. 이는 비용이 저렴한 프로세스입니다. 파티션 수를 줄이면 `repartition(N)`에서 셔플을 수행하여 각 파티션에 균등하게 레코드의 양을 분산하므로 `coalesce(N)`가 `repartition(N)`보다 선호됩니다. 이 경우 비용과 관리 오버헤드가 증가합니다.
- Spark 3.x 적응형 쿼리 실행을 사용합니다. Apache Spark 섹션의 주요 주제에서 설명한 대로 적응형 쿼리 실행은 파티션 수를 자동으로 병합하는 함수를 제공합니다. 실행을 수행할 때까지 파티션 수를 알 수 없는 경우 이 접근 방식을 사용할 수 있습니다.

JDBC에서 데이터 로드 병렬화

Spark RDD 파티션 수는 구성에 의해 결정됩니다. 기본적으로 `SELECT` 쿼리를 통해 전체 소스 데이터 세트를 스캔하기 위해 단일 태스크만 실행됩니다.

AWS Glue DynamicFrames과 Spark DataFrames 모두 여러 작업에서 병렬화된 JDBC 데이터 로드를 지원합니다. 이는 `where` 조건자를 사용하여 하나의 `SELECT` 쿼리를 여러 쿼리로 분할하여 수행됩니다. JDBC에서 읽기를 병렬화하려면 다음 옵션을 구성합니다.

- For AWS Glue DynamicFrame, 설정`hashfield`(또는 `hashexpression`) 및 `hashpartition`. 자세한 내용은 JDBC 테이블을 병렬로 읽기를 참조하세요.

```
connection_mysql8_options = {
    "url": "jdbc:mysql://XXXXXXXXXX.XXXXXXX.us-east-1.rds.amazonaws.com:3306/test",
    "dbtable": "medicare_tb",
    "user": "test",
    "password": "XXXXXXXXX",
    "hashexpression":"id",
    "hashpartitions":"10"
}
datasource0 = glueContext.create_dynamic_frame.from_options(
    'mysql',
    connection_options=connection_mysql8_options,
    transformation_ctx= "datasource0"
```

```
)
```

- Spark DataFrame의 경우 `numPartitions`, `partitionColumn`, `lowerBound`, `upperBound`를 설정합니다. 자세한 내용은 JDBC To Other Databases를 참조하세요.

```
df = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:mysql://XXXXXXXXXX.XXXXXXX.us-east-1.rds.amazonaws.com:3306/
test") \
    .option("dbtable", "medicare_tb") \
    .option("user", "test") \
    .option("password", "XXXXXXXXXX") \
    .option("partitionColumn", "id") \
    .option("numPartitions", "10") \
    .option("lowerBound", "0") \
    .option("upperBound", "1141455") \
    .load()

df.write.format("json").save("s3://bucket_name/Tests/sparkjdbc/with_parallel/")
```

ETL 커넥터를 사용할 경우 DynamoDB에서 데이터 로드 병렬화

Spark RDD 파티션 수는 `dynamodb.splits` 파라미터에 의해 결정됩니다. Amazon DynamoDB에서 읽기를 병렬화하려면 다음 옵션을 구성합니다.

- `dynamodb.splits`의 값을 늘립니다.
- for Spark의 ETL에 대한 연결 유형 및 옵션에 설명된 공식에 따라 파라미터를 최적화 AWS Glue 합니다.

Kinesis Data Streams에서 데이터 로드 병렬화

Spark RDD 파티션 수는 소스 Amazon Kinesis Data Streams 데이터 스트림의 샤드 수에 의해 결정됩니다. 데이터 스트림에 샤드가 몇 개뿐인 경우 Spark 태스크는 몇 개뿐입니다. 이로 인해 다운스트림 프로세스에서 병렬화가 낮아질 수 있습니다. Kinesis Data Streams에서 읽기를 병렬화하려면 다음 옵션을 구성합니다.

- Kinesis Data Streams에서 데이터를 로드할 경우 더 많은 병렬화를 얻으려면 샤드 수를 늘립니다.
- 마이크로 배치의 로직이 충분히 복잡한 경우 불필요한 열을 삭제한 후 배치 시작 시 데이터를 재분할하는 방법을 고려합니다.

자세한 내용은 AWS Glue 스트리밍 ETL 작업의 비용 및 성능을 최적화하는 모범 사례를 참조하세요.

데이터 로드 후 태스크 병렬화

데이터 로드 후 태스크를 병렬화하려면 다음 옵션을 사용하여 RDD 파티션 수를 늘립니다.

- 특히 로드 자체를 병렬화할 수 없는 경우 초기 로드 직후 더 많은 수의 파티션을 생성하기 위해 데이터를 다시 분할합니다.

  파티션 수를 지정하여 DynamicFrame 또는 DataFrame에서 `repartition()`을 직접 호출합니다. 일반적으로 사용 가능한 코어 수의 2~3배가 적당합니다.

  그러나 분할된 테이블을 작성할 경우 파일이 크게 증가할 수 있습니다(각 파티션이 잠재적으로 각 테이블 파티션에 파일을 생성할 수 있음). 이를 방지하기 위해 열을 기준으로 DataFrame을 다시 분할할 수 있습니다. 이렇게 하면 쓰기 전에 데이터가 구성되도록 테이블 파티션 열이 사용됩니다. 테이블 파티션에서 작은 파일을 가져오지 않고도 더 많은 수의 파티션을 지정할 수 있습니다. 그러나 일부 파티션 값이 대부분의 데이터로 끝나고 작업 완료가 지연되는 데이터 스큐를 피하도록 주의하세요.
- 셔플이 있는 경우 `spark.sql.shuffle.partitions` 값을 늘립니다. 또한 셔플 작업 시 메모리 문제를 해결하는 데 도움이 될 수 있습니다.

  2,001개가 넘는 셔플 파티션이 있는 경우 Spark는 압축된 메모리 형식을 사용합니다. 숫자가 이에 근사한 경우 보다 효율적인 표현을 얻기 위해 해당 한도를 초과하는 `spark.sql.shuffle.partitions` 값을 설정하려고 할 수 있습니다.

셔플 최적화

`join()` 및 `groupByKey()`와 같은 특정 작업에서는 Spark가 셔플을 수행해야 합니다. 셔플은 데이터를 재분산하기 위한 Spark의 메커니즘으로, 이를 통해 여러 RDD 파티션에서 다르게 데이터가 그룹화됩니다. 셔플 작업은 성능 병목 현상을 해결하는 데 도움이 될 수 있습니다. 그러나 셔플 작업에는 일반적으로 Spark 실행기 사이에서 데이터를 복사하는 작업이 포함되므로 셔플은 복잡하고 비용이 많이 드는 작업입니다. 예를 들어 셔플에서는 다음과 같은 비용이 발생합니다.

- 디스크 I/O:
  - 디스크에서 많은 수의 중간 파일을 생성합니다.
- 네트워크 I/O:
  - 많은 네트워크 연결이 필요합니다(연결 수 = `Mapper × Reducer`).

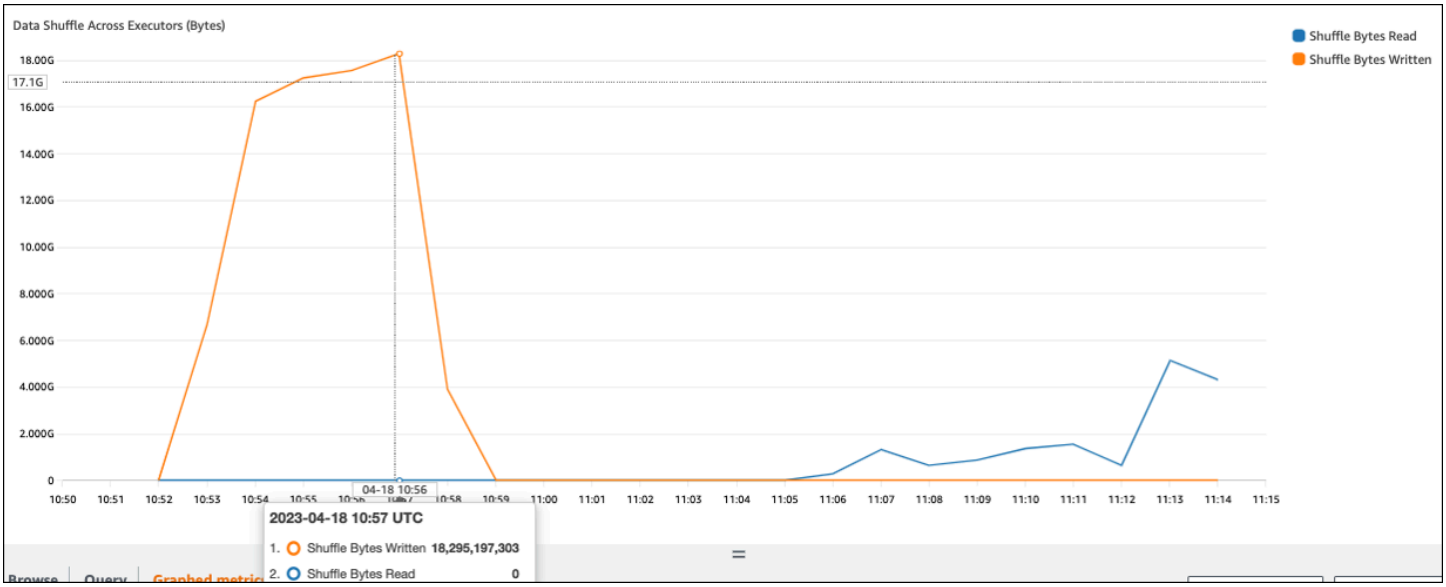
  - 레코드는 다른 Spark 실행기에서 호스팅될 수 있는 새 RDD 파티션으로 집계되므로 데이터세트의 상당 부분이 네트워크를 통해 Spark 실행기 사이에서 이동할 수 있습니다.
- CPU 및 메모리 로드:
  - 값을 정렬하고 데이터세트를 병합합니다. 이러한 작업은 실행기에서 계획되어 실행기에 과도한 로드를 부과합니다.

셔플은 Spark 애플리케이션의 성능 저하에서 가장 중요한 요인 중 하나입니다. 중간 데이터를 저장하는 동안 실행기의 로컬 디스크에서 공간을 소진하여 Spark 작업이 실패할 수 있습니다.

CloudWatch 지표와 Spark UI에서 셔플 성능을 평가할 수 있습니다.

CloudWatch 지표

셔플 바이트 쓰기 값이 셔플 바이트 읽기와 비교하여 높은 경우 Spark 작업은 `join()` 또는 `groupByKey()`와 같은 셔플 작업을 사용할 수 있습니다.



Spark UI

Spark UI의 스테이지 탭에서 셔플 읽기 크기/레코드 값을 확인할 수 있습니다. 실행기 탭에서도 볼 수 있습니다.

다음 스크린샷에서 각 실행기는 약 18.6GB/4,020,000개 레코드를 셔플 프로세스와 교환합니다(총 셔플 읽기 크기는 약 75GB).

셔플 유출(디스크) 열에는 디스크로의 많은 데이터 유출 메모리가 표시되며, 이로 인해 디스크가 가득 차거나 성능 문제가 발생할 수 있습니다.

Aggregated Metrics by Executor

Executor ID ▲	Address	Shuffle Read Size / Records	Shuffle Spill (Memory)	Shuffle Spill (Disk)
1	172.35.205.23:46731	18.6 GB / 40210300	98.1 GB	16.8 GB
2	172.35.195.173:46185	18.7 GB / 40246767	117.2 GB	17.3 GB
3	172.36.135.106:35913	18.6 GB / 40253921	101.6 GB	16.6 GB
4	172.34.131.223:46879	18.6 GB / 40190741	99.5 GB	16.4 GB

이러한 증상이 관찰되고 성능 목표에 비해 단계가 너무 오래 걸리거나 `Out Of Memory` 또는 `No space left on device` 오류로 인해 실패하는 경우 다음 솔루션을 고려합니다.

조인 최적화

테이블을 조인하는 `join()` 작업은 가장 일반적으로 사용되는 셔플 작업이지만 종종 성능 병목 현상이 되곤 합니다. 조인은 비용이 많이 드는 작업이므로 비즈니스 요구 사항에 필수적인 경우가 아니면 사용하지 않는 것이 좋습니다. 다음 질문을 통해 데이터 파이프라인을 효율적으로 사용하고 있는지 다시 확인합니다.

- 재사용할 수 있는 다른 작업에서도 수행되는 조인을 다시 계산하고 있나요?
- 출력의 소비자가 사용하지 않는 값으로 외래 키를 확인하기 위해 조인하고 있나요?

조인 작업이 비즈니스 요구 사항에 필수적임을 확인한 후라면 요구 사항을 충족하는 방식으로 조인을 최적화하기 위해 다음 옵션을 참조하세요.

조인 전에 푸시다운 사용

조인을 수행하기 전에 DataFrame에서 불필요한 행과 열을 필터링합니다. 이 경우 다음과 같은 이점이 있습니다.

- 셔플 중에 데이터 전송량 감소
- Spark 실행기에서 처리량 감소
- 데이터 스캔량 감소

```
# Default
df_joined  = df1.join(df2, ["product_id"])
```

```
# Use Pushdown
df1_select =
 df1.select("product_id","product_title","star_rating").filter(col("star_rating")>=4.0)
df2_select = df2.select("product_id","category_id")
df_joined  = df1_select.join(df2_select, ["product_id"])
```

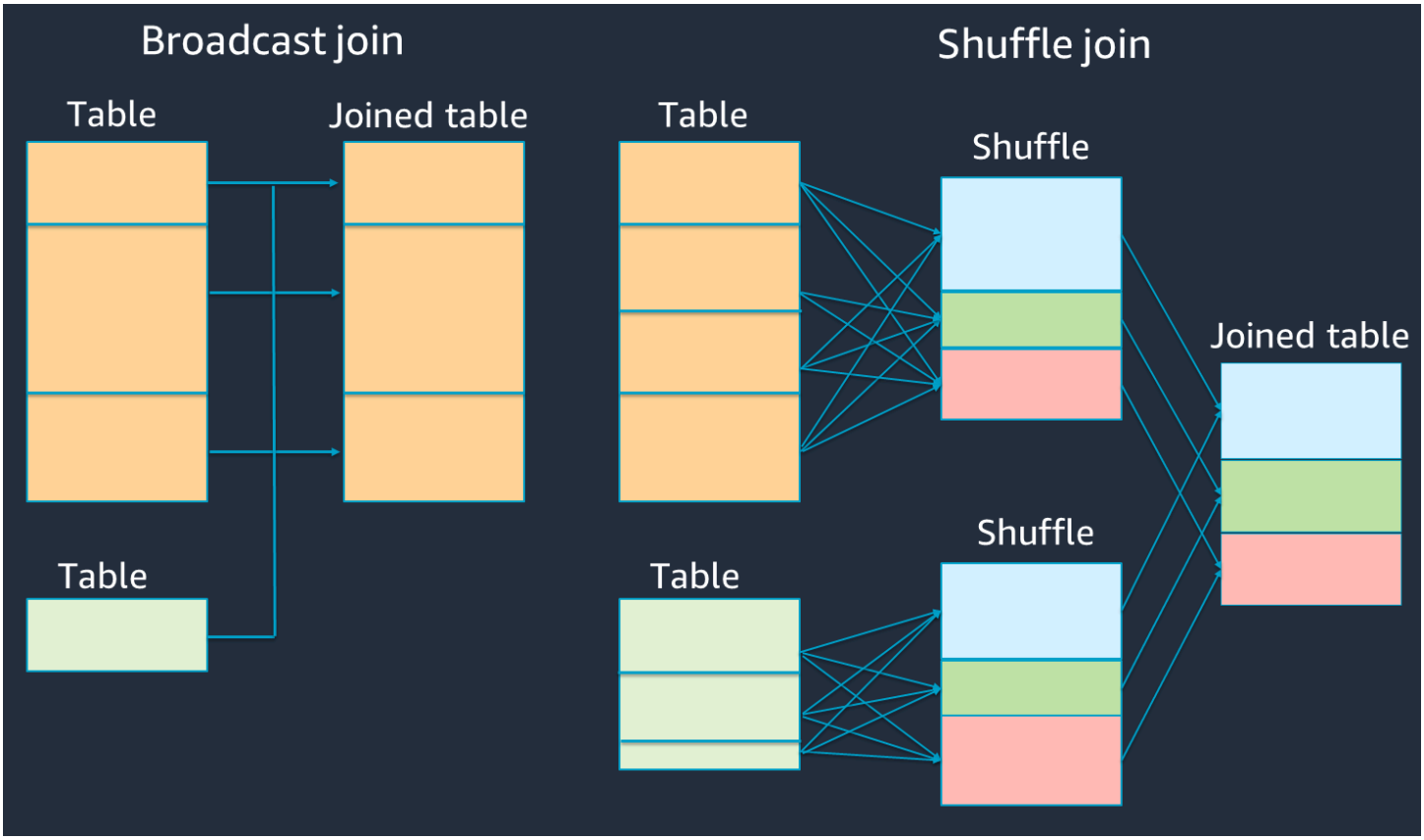
DataFrame 조인 사용

RDD API 또는 DynamicFrame 조인 대신 SparkSQL, DataFrame 및 Datasets와 같은 Spark 상위 수준 API를 사용해 보세요. `dyf.toDF()`와 같은 메서드 직접 호출을 사용하여 DynamicFrame을 DataFrame으로 변환할 수 있습니다. Apache Spark의 주요 주제 섹션에서 설명한 대로 이러한 조인 작업은 내부적으로 Catalyst 옵티마이저의 쿼리 최적화를 활용합니다.

셔플 및 브로드캐스트 해시 조인 및 힌트

Spark는 셔플 조인과 브로드캐스트 해시 조인이라는 두 가지 유형의 조인을 지원합니다. 브로드캐스트 해시 조인은 셔플 작업이 필요하지 않으며 셔플 조인보다 필요한 처리가 적을 수 있습니다. 그러나 작은 테이블을 큰 테이블에 조인할 때만 적용됩니다. 단일 Spark 실행기의 메모리에 적합할 수 있는 테이블을 조인할 경우 브로드캐스트 해시 조인을 사용하는 방법을 고려합니다.

다음 다이어그램에서는 브로드캐스트 해시 조인 및 셔플 조인에 대한 상위 수준 구조와 단계를 보여줍니다.



각 조인의 세부 정보는 다음과 같습니다.

- 셔플 조인:
  - 셔플 해시 조인은 정렬 없이 두 테이블을 조인하고 두 테이블 간에 조인을 분산합니다. Spark 실행기의 메모리에 저장할 수 있는 작은 테이블의 조인에 적합합니다.
  - 정렬 병합 조인은 조인할 두 테이블을 키별로 분산하고 조인하기 전에 정렬합니다. 큰 테이블의 조인에 적합합니다.
- 브로드캐스트 해시 조인:
  - 브로드캐스트 해시 조인은 더 작은 RDD 또는 테이블을 각 워커 노드로 푸시합니다. 그런 다음 더 큰 RDD 또는 테이블의 각 파티션과 맵 측 결합을 수행합니다.

    이 방법은 RDD 또는 테이블 중 하나가 메모리에 적합하거나 메모리에 적합하도록 만들 수 있는 조인에 적절합니다. 셔플이 필요하지 않으므로 가능하면 브로드캐스트 해시 조인을 수행하는 것이 좋습니다. 다음과 같이 조인 힌트를 사용하여 Spark에 브로드캐스트 조인을 요청할 수 있습니다.

```
# DataFrame
from pySpark.sql.functions import broadcast
```

```
df_joined= df_big.join(broadcast(df_small), right_df[key] == left_df[key],
 how='inner')

-- SparkSQL
SELECT /*+ BROADCAST(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
```

조인 힌트에 대한 자세한 내용은 Join hints를 참조하세요.

In AWS Glue 3.0 이상에서는 적응형 쿼리 실행 및 추가 파라미터를 활성화하여 브로드캐스트 해시 조인을 자동으로 활용할 수 있습니다. 적응형 쿼리 실행은 조인 측의 런타임 통계가 적응형 브로드캐스트 해시 조인 임계치보다 작을 경우 정렬-병합 조인을 브로드캐스트 해시 조인으로 변환합니다.

In AWS Glue 3.0에서는를 설정하여 적응형 쿼리 실행을 활성화할 수 있습니다`spark.sql.adaptive.enabled=true`. 적응형 쿼리 실행은 AWS Glue 4.0에서 기본적으로 활성화됩니다.

셔플 및 브로드캐스트 해시 조인과 관련된 추가 파라미터를 설정할 수 있습니다.

- `spark.sql.adaptive.localShuffleReader.enabled`
- `spark.sql.adaptive.autoBroadcastJoinThreshold`

관련 파라미터에 대한 자세한 내용은 Converting sort-merge join to broadcast join을 참조하세요.

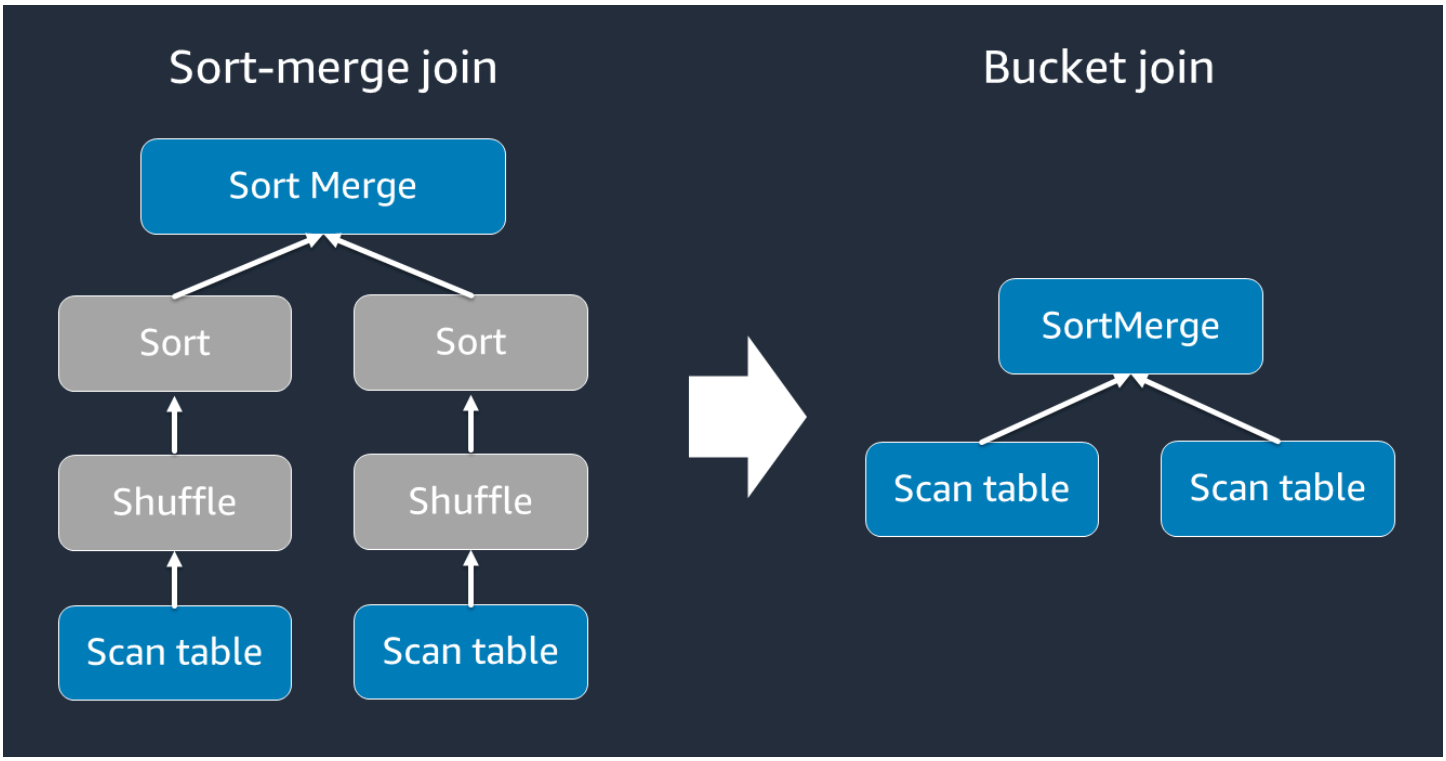
In AWS Glue 3.0 이상에서는 셔플에 다른 조인 힌트를 사용하여 동작을 조정할 수 있습니다.

```
-- Join Hints for shuffle sort merge join
SELECT /*+ SHUFFLE_MERGE(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGEJOIN(t2) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
SELECT /*+ MERGE(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;

-- Join Hints for shuffle hash join
SELECT /*+ SHUFFLE_HASH(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;

-- Join Hints for shuffle-and-replicate nested loop join
SELECT /*+ SHUFFLE_REPLICATE_NL(t1) /  FROM t1 INNER JOIN t2 ON t1.key = t2.key;
```

버킷팅 사용

정렬 병합 조인에는 셔플과 정렬 그리고 병합이라는 두 단계가 필요합니다. 이 두 단계로 인해 Spark 실행기에 과부하가 걸리고, 일부 실행기가 병합되고 다른 실행기가 동시에 정렬될 때 OOM 및 성능 문

제가 발생할 수 있습니다. 이러한 경우 버킷팅을 사용하여 효율적으로 조인할 수 있습니다. 버킷팅은 조인 키에 대한 입력을 사전 셔플 및 사전 정렬한 다음 정렬된 데이터를 중간 테이블에 씁니다. 정렬된 중간 테이블을 미리 정의하여 큰 테이블을 조인할 때 셔플 및 정렬 단계의 비용을 줄일 수 있습니다.



버킷 테이블은 다음과 같은 경우에 유용합니다.

- `account_id`와 같은 동일한 키를 통해 자주 조인되는 데이터
- 공통 열에서 버킷팅할 수 있는 기본 및 델타 테이블과 같은 일일 누적 테이블 로드

다음 코드를 사용하여 버킷팅된 테이블을 생성할 수 있습니다.

```
df.write.bucketBy(50, "account_id").sortBy("age").saveAsTable("bucketed_table")
```

조인 전 조인 키의 DataFrames 다시 분할

조인 전에 조인 키에서 두 DataFrames를 다시 분할하려면 다음 명령문을 사용합니다.

```
df1_repartitioned = df1.repartition(N,"join_key")
df2_repartitioned = df2.repartition(N,"join_key")
df_joined  = df1_repartitioned.join(df2_repartitioned,"product_id")
```

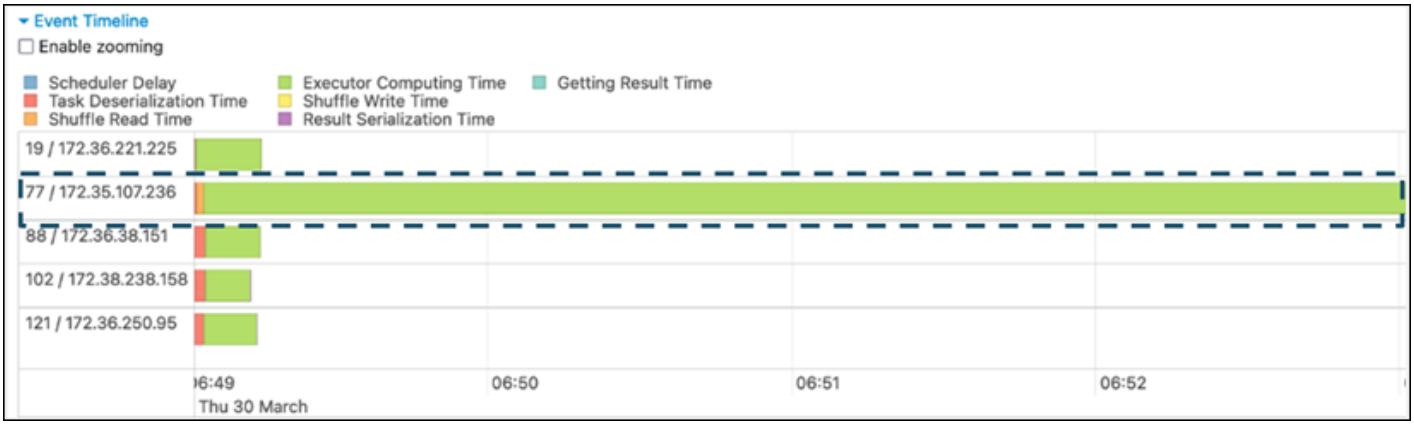
이렇게 하면 조인을 시작하기 전에 조인 키에서 두 개의 개별 RDD가 분할됩니다. 두 RDD가 동일한 분할 코드로 동일한 키에 분할되는 경우 조인에 대해 셔플을 수행하기 전에 함께 조인하려는 RDD 레코드가 동일한 작업자에 함께 존재할 가능성이 큽니다. 이렇게 하면 조인 중에 네트워크 활동 및 데이터 스큐를 줄여 성능이 향상될 수 있습니다.

데이터 스큐 재정의

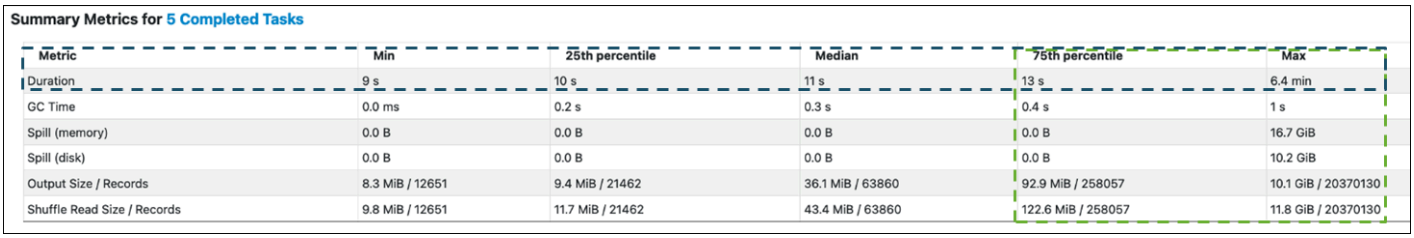
데이터 스큐는 Spark 작업에서 가장 일반적인 병목 현상 원인 중 하나입니다. 데이터가 RDD 파티션에 균일하게 분산되지 않을 경우 나타납니다. 이 경우 해당 파티션에 대한 태스크가 다른 태스크보다 훨씬 오래 걸리므로 애플리케이션의 전체 처리 시간이 지연됩니다.

데이터 스큐를 식별하려면 Spark UI에서 다음 지표를 평가합니다.

- Spark UI의 단계 탭에서 이벤트 타임라인 페이지를 검토합니다. 다음 스크린샷에서는 고르지 않은 작업 분산을 볼 수 있습니다. 균등하지 않게 분산되거나 실행하는 데 너무 오래 걸리는 태스크는 데이터 스큐를 나타낼 수 있습니다.



- 또 다른 중요한 페이지는 Spark 태스크에 대한 통계를 보여주는 요약 지표입니다. 다음 스크린샷은 지속 시간, GC 시간, 유출(메모리), 유출(디스크) 등에 대한 백분위수를 포함한 지표를 보여줍니다.



Metric	Min	25th percentile	Median	75th percentile	Max
Duration	9 s	10 s	11 s	13 s	6.4 min
GC Time	0.0 ms	0.2 s	0.3 s	0.4 s	1 s
Spill (memory)	0.0 B	0.0 B	0.0 B	0.0 B	16.7 GiB
Spill (disk)	0.0 B	0.0 B	0.0 B	0.0 B	10.2 GiB
Output Size / Records	8.3 MiB / 12651	9.4 MiB / 21462	36.1 MiB / 63860	92.9 MiB / 258057	10.1 GiB / 20370130
Shuffle Read Size / Records	9.8 MiB / 12651	11.7 MiB / 21462	43.4 MiB / 63860	122.6 MiB / 258057	11.8 GiB / 20370130

태스크가 균등하게 분산되면 모든 백분위수에서 비슷한 숫자가 표시됩니다. 데이터 스큐가 있는 경우 각 백분위수에 편향된 값이 표시됩니다. 이 예제에서 태스크 지속 시간은 최소, 25번째 백분위수, 중앙값, 75번째 백분위수에서 13초 미만입니다. 최대 태스크는 75번째 백분위수보다 100배 더 많은

데이터를 처리했지만 해당 지속 시간은 6.4분으로 약 30배 더 깁니다. 즉, 하나 이상의 태스크(또는 태스크의 최대 25%)이 나머지 태스크보다 훨씬 오래 걸렸습니다.

데이터 스큐가 표시되면 다음을 시도하세요.

- AWS Glue 3.0을 사용하는 경우를 설정하여 적응형 쿼리 실행을 활성화합니다`spark.sql.adaptive.enabled=true`. 적응형 쿼리 실행은 기본적으로 AWS Glue 4.0에서 활성화됩니다.

  다음 관련 파라미터를 설정하여 조인이 도입한 데이터 스큐에 적응형 쿼리 실행을 사용할 수도 있습니다.

  - `spark.sql.adaptive.skewJoin.skewedPartitionFactor`
  - `spark.sql.adaptive.skewJoin.skewedPartitionThresholdInBytes`
  - `spark.sql.adaptive.advisoryPartitionSizeInBytes=128m (128 mebibytes or larger should be good)`
  - `spark.sql.adaptive.coalescePartitions.enabled=true (when you want to coalesce partitions)`

  자세한 내용은 Apache Spark 설명서를 참조하세요.

- 조인 키에 대해 큰 범위의 값이 있는 키를 사용합니다. 셔플 조인에서 키의 각 해시 값에 대해 파티션이 결정됩니다. 조인 키의 카디널리티가 너무 낮으면 해시 함수가 여러 파티션에서 데이터를 분산하는 잘못된 작업을 수행할 가능성이 더 큽니다. 따라서 애플리케이션 및 비즈니스 로직이 이를 지원하는 경우 더 높은 카디널리티 키 또는 복합 키를 사용하는 방법을 고려합니다.

```
# Use Single Primary Key
df_joined = df1_select.join(df2_select, ["primary_key"])

# Use Composite Key
df_joined = df1_select.join(df2_select, ["primary_key","secondary_key"])
```

캐시 사용

반복 DataFrames 사용하는 경우 `df.cache()` 또는 `df.persist()`를 사용하여 각 Spark 실행기의 메모리 및 디스크에 계산 결과를 캐싱함으로써 추가 셔플 또는 계산을 피합니다. Spark는 디스크에 RDD를 유지하거나 여러 노드(스토리지 수준)에서 복제하는 작업도 지원합니다.

예를 들어 df.persist()를 추가하여 DataFrames를 유지할 수 있습니다. 캐시가 더 이상 필요하지 않은 경우 unpersist를 사용하여 캐싱된 데이터를 삭제할 수 있습니다.

```
df = spark.read.parquet("s3://<Bucket>/parquet/product_category=Books/")
df_high_rate = df.filter(col("star_rating")>=4.0)
df_high_rate.persist()

df_joined1 = df_high_rate.join(<Table1>, ["key"])
df_joined2 = df_high_rate.join(<Table2>, ["key"])
df_joined3 = df_high_rate.join(<Table3>, ["key"])
...
df_high_rate.unpersist()
```

불필요한 Spark 작업 제거

count, show 또는 collect와 같은 불필요한 작업을 실행하지 마세요. Apache Spark 섹션의 주요 주제에서 설명한 대로 Spark는 지연됩니다. 변환된 각 RDD는 작업을 실행할 때마다 다시 계산될 수 있습니다. Spark 작업을 많이 사용하면 각 작업에 대해 여러 소스 액세스, 태스크 계산 및 셔플 실행이 직접 호출됩니다.

상용 환경에서 collect() 또는 기타 작업이 필요하지 않은 경우 제거하는 것을 고려합니다.

> **Note**
>
> 상용 환경에서는 Spark collect()를 최대한 사용하지 마세요. collect() 작업은 Spark 실행기의 모든 계산 결과를 Spark 드라이버에 반환하므로 이로 인해 Spark 드라이버가 OOM 오류를 반환할 수 있습니다. OOM 오류를 방지하기 위해 Spark는 기본적으로 spark.driver.maxResultSize = 1GB를 설정합니다. 그러면 Spark 드라이버에 반환되는 최대 데이터 크기를 1GB로 제한합니다.

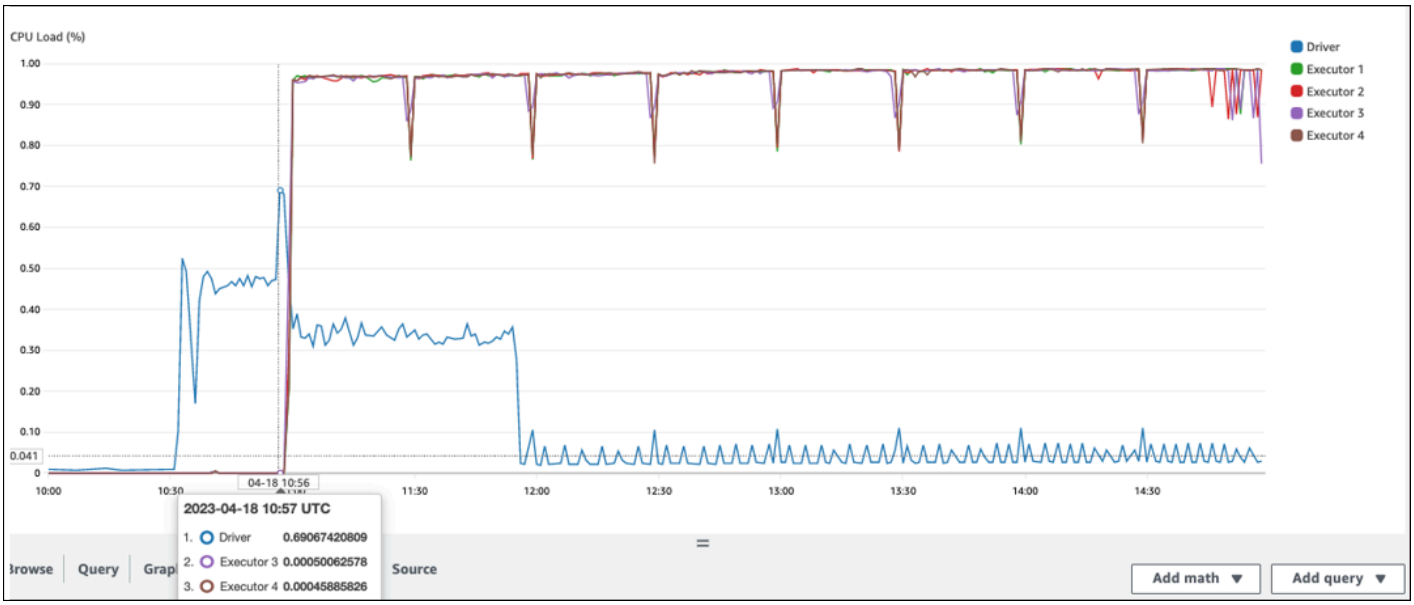
계획 오버헤드 최소화

Apache Spark의 주요 주제에서 설명한 대로 Spark 드라이버는 실행 계획을 생성합니다. 이 계획에 따라 분산 처리를 위해 Spark 실행기에 태스크가 할당됩니다. 그러나 작은 파일이 많거나 AWS Glue Data Catalog 에 파티션이 많은 경우 Spark 드라이버가 병목 현상의 원인이 될 수 있습니다. 높은 계획 오버헤드를 식별하려면 다음 지표를 평가합니다.

CloudWatch 지표

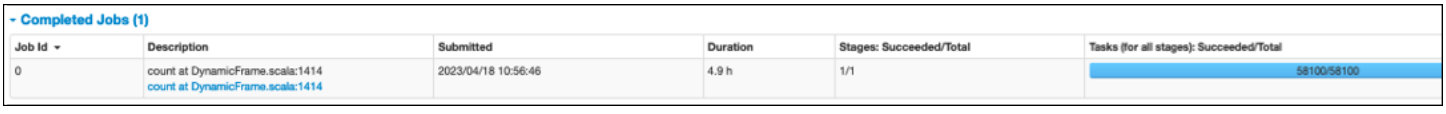
CPU 로드 및 메모리 사용률에서 다음 상황을 확인합니다.

- Spark 드라이버 CPU 로드 및 메모리 사용률은 높음으로 기록됩니다. 일반적으로 Spark 드라이버는 데이터를 처리하지 않으므로 CPU 로드 및 메모리 사용률이 급증하지 않습니다. 그러나 Amazon S3 데이터 소스에 작은 파일이 너무 많은 경우 모든 S3 객체를 나열하고 많은 수의 작업을 관리하면 리소스 사용률이 높아질 수 있습니다.
- Spark 실행기에서 처리가 시작되기 전에 긴 간격이 나타납니다. 다음 예제 스크린샷에서 Spark 실행기의 CPU 로드는 AWS Glue 작업이 10:00에 시작되더라도 10:57까지 너무 낮습니다. 이는 Spark 드라이버가 실행 계획을 생성하는 데 시간이 오래 걸릴 수 있음을 나타냅니다. 이 예제에서는 Data Catalog에서 많은 파티션을 검색하고 Spark 드라이버에서 많은 작은 파일을 나열하는 데 시간이 오래 걸립니다.



Spark UI

Spark UI의 작업 탭에서 제출 시간을 볼 수 있습니다. 다음 예제에서는 작업이 10:00:00에 시작되었더라도 Spark 드라이버가 10:56:46에 AWS Glue job0을 시작했습니다.



작업 탭에서 태스크(모든 단계): 성공/총계 시간을 볼 수도 있습니다. 이 경우 작업 수는 58100으로 기록됩니다. 태스크 병렬화 페이지의 Amazon S3 섹션에서 설명한 대로 태스크 수는 대략적으로 S3 객체 수에 해당합니다. 즉, Amazon S3에는 약 58,100개의 객체가 있습니다.

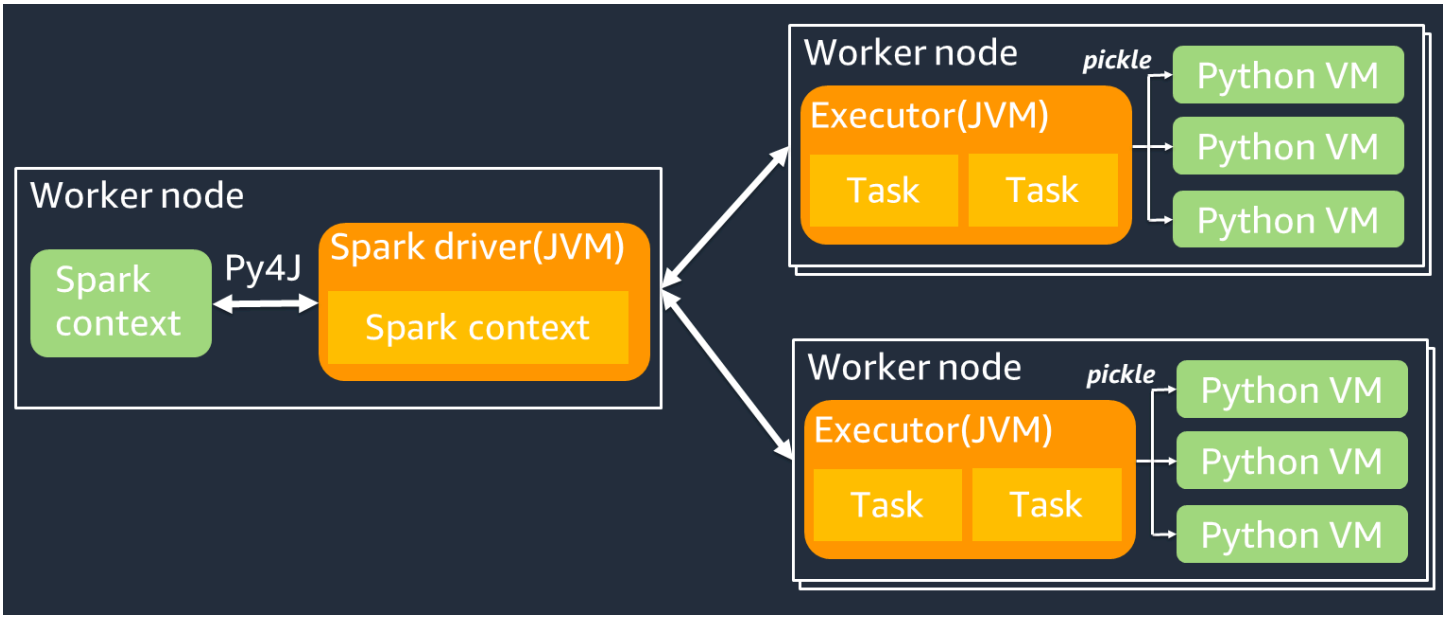
이 작업 및 타임라인에 대한 자세한 내용은 스테이지 탭을 검토하세요. Spark 드라이버에서 병목 현상이 관찰되면 다음 솔루션을 고려합니다.

- Amazon S3에 파일이 너무 많으면 작업 병렬화 페이지의 파티션이 너무 많음 섹션에 있는 과도한 병렬화에 대한 지침을 고려합니다.
- Amazon S3에 파티션이 너무 많으면 데이터 스캔 양 감소 페이지의 Amazon S3 파티션이 너무 많음 섹션에 있는 과도한 분할에 대한 지침을 고려합니다. Data Catalog에서 파티션 메타데이터를 검색하는 데 걸리는 지연 시간을 줄이기 위해 파티션이 많은 경우 AWS Glue 파티션 인덱스를 활성화합니다. 자세한 내용은 AWS Glue 파티션 인덱스를 사용하여 쿼리 성능 개선을 참조하세요.
- JDBC에 파티션이 너무 많으면 `hashpartition` 값을 낮춥니다.
- DynamoDB에 파티션이 너무 많으면 `dynamodb.splits` 값을 낮춥니다.
- 스트리밍 작업에 파티션이 너무 많으면 샤드 수를 줄입니다.

사용자 정의 함수 최적화

PySpark의 사용자 정의 함수(UDF) 및 `RDD.map`은 종종 성능을 크게 저하시킬 수 있습니다. Spark의 기본 Scala 구현에서 Python 코드를 정확하게 나타내는 데 필요한 오버헤드 때문입니다.

다음 다이어그램은 의 아키텍처입니다. PySpark를 사용하는 경우 Spark 드라이버는 Py4j 라이브러리를 사용하여 Python에서 Java 메서드를 직접 호출합니다. Spark SQL 또는 DataFrame 기본 제공 함수를 직접 호출할 때 함수는 최적화된 실행 계획을 사용하여 각 실행기의 JVM에서 실행되므로 Python과 Scala 간에 성능 차이가 거의 없습니다.



`map/ mapPartitions/ udf` 사용 등 자체 Python 로직을 사용하는 경우 태스크는 Python 런타임 환경에서 실행됩니다. 두 환경을 관리하면 오버헤드 비용이 발생합니다. 또한 JVM 런타임 환경의 기본 제공 함수에서 사용할 수 있도록 메모리의 데이터를 변환해야 합니다. Pickle은 JVM과 Python 런타임 간 교환에 기본적으로 사용되는 직렬화 형식입니다. 그러나 이 직렬화 및 역직렬화 비용은 매우 높으므로 Java 또는 Scala로 작성된 UDF는 Python UDF보다 더 빠릅니다.

PySpark에서 직렬화 및 역직렬화 오버헤드를 방지하려면 다음을 고려합니다.

- 기본 제공 Spark SQL 함수 사용 - 자체 UDF 또는 맵 함수를 Spark SQL 또는 DataFrame 기본 제공 함수로 대체하는 방법을 고려합니다. Spark SQL 또는 DataFrame 기본 제공 함수를 실행할 때 태스크는 각 실행기의 JVM에서 처리되므로 Python과 Scala 간에 성능 차이가 거의 없습니다.
- Scala 또는 Java에서 UDF 구현 - JVM에서 실행되므로 Java 또는 Scala로 작성된 UDF를 사용하는 방법을 고려합니다.
- 벡터화된 워크로드에 Apache Arrow 기반 UDF 사용 - Arrow 기반 UDF를 사용하는 방법을 고려합니다. 이 기능을 벡터화된 UDF(Pandas UDF)라고도 합니다. Apache Arrow는 JVM과 Python 프로세스 간에 데이터를 효율적으로 전송하는 데 사용할 AWS Glue 수 있는 언어에 구애받지 않는 인 메모리 데이터 형식입니다. 이는 현재 Pandas 또는 NumPy 데이터로 작업하는 Python 사용자에게 가장 유용합니다.

  Arrow는 열 형식(벡터화된 형식)입니다. 이 기능의 사용은 자동이 아니며 최대한 활용하고 호환성을 보장하기 위해 구성 또는 코드를 약간 변경해야 할 수 있습니다. 자세한 내용과 제한 사항은 Apache Arrow in PySpark를 참조하세요.

  다음 예제에서는 표준 Python, 벡터화된 UDF 및 Spark SQL에서 기본 증분 UDF를 비교합니다.

표준 Python UDF

예제 시간은 3.20(초)입니다.

예제 코드

```
# DataSet
df = spark.range(10000000).selectExpr("id AS a","id AS b")

# UDF Example
def plus(a,b):
    return a+b
spark.udf.register("plus",plus)

df.selectExpr("count(plus(a,b))").collect()
```

실행 계획

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[], functions=[count(pythonUDF0#124)])
+- Exchange SinglePartition, ENSURE_REQUIREMENTS, [id=#580]
+- HashAggregate(keys=[], functions=[partial_count(pythonUDF0#124)])
+- Project [pythonUDF0#124]
+- BatchEvalPython [plus(a#116L, b#117L)], [pythonUDF0#124]
+- Project [id#114L AS a#116L, id#114L AS b#117L]
+- Range (0, 10000000, step=1, splits=16)
```

벡터화된 UDF

예제 시간은 0.59(초)입니다.

벡터화된 UDF는 이전 UDF 예제보다 5배 빠릅니다. `Physical Plan`을 확인하면 `ArrowEvalPython`를 볼 수 있으며, 이는 이 애플리케이션이 Apache Arrow에 의해 벡터화되었음을 의미합니다. 벡터화된 UDF를 활성화하려면 코드에서 `spark.sql.execution.arrow.pyspark.enabled = true`를 지정해야 합니다.

예제 코드

```
# Vectorized UDF
from pyspark.sql.types import LongType
```

```
from pyspark.sql.functions import count, pandas_udf

# Enable Apache Arrow Support
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")

# DataSet
df = spark.range(10000000).selectExpr("id AS a","id AS b")

# Annotate pandas_udf to use Vectorized UDF
@pandas_udf(LongType())
def pandas_plus(a,b):
    return a+b
spark.udf.register("pandas_plus",pandas_plus)

df.selectExpr("count(pandas_plus(a,b))").collect()
```

실행 계획

```
== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[], functions=[count(pythonUDF0#1082L)],
 output=[count(pandas_plus(a, b))#1080L])
+- Exchange SinglePartition, ENSURE_REQUIREMENTS, [id=#5985]
+- HashAggregate(keys=[], functions=[partial_count(pythonUDF0#1082L)],
 output=[count#1084L])
+- Project [pythonUDF0#1082L]
+- ArrowEvalPython [pandas_plus(a#1074L, b#1075L)], [pythonUDF0#1082L], 200
+- Project [id#1072L AS a#1074L, id#1072L AS b#1075L]
+- Range (0, 10000000, step=1, splits=16)
```

Spark SQL

예제 시간은 0.087(초)입니다.

Python 런타임 없이 각 실행기의 JVM에서 태스크가 실행되므로 Spark SQL은 벡터화된 UDF보다 훨씬 빠릅니다. UDF를 기본 제공 함수로 교체할 수 있는 경우 교체하는 것이 좋습니다.

예제 코드

```
df.createOrReplaceTempView("test")
spark.sql("select count(a+b) from test").collect()
```

빅 데이터에 panda 사용

이미 pandas에 익숙하고 빅 데이터에 Spark를 사용하려는 경우 Spark. AWS Glue 4.0 이상에서 pandas API를 사용할 수 있습니다. 시작하기 위해 공식 노트북 Quickstart: Pandas API on Spark를 사용할 수 있습니다. 자세한 내용은 PySpark 설명서를 참조하세요.

리소스

- AWS Glue
- Performance Tuning(Spark SQL 가이드)
- AWS Glue Optimization Workshop

문서 기록

아래 표에 이 가이드의 주요 변경 사항이 설명되어 있습니다. 향후 업데이트에 대한 알림을 받으려면 RSS 피드를 구독하십시오.

변경 사항	설명	날짜
최초 게시	—	2024년 1월 2일

기계 번역으로 제공되는 번역입니다. 제공된 번역과 원본 영어의 내용이 상충하는 경우에는 영어 버전이 우선합니다.