Amazon Redshift에 대한 쿼리 모범 사례

AWS 권장 가이드

AWS 권장 가이드: Amazon Redshift에 대한 쿼리 모범 사례

Table of Contents

Amazon Redshift에 대한 쿼리 모범 사례

Ethan Stark, Amazon Web Services(AWS)

2024년 6월(문서 기록)

개요

이 가이드에서는 Amazon Redshift에서 쿼리 및 테이블 성능을 최적화하기 위한 권장 사항과 모범 사례를 제공합니다. Amazon Redshift를 사용하여 표준 SQL을 통해 데이터 웨어하우스와 데이터 레이크 전반에서 페타바이트 규모의 정형 및 반정형 데이터를 쿼리할 수 있습니다. 또한 이 가이드에서는 Amazon Redshift 데이터 웨어하우스의 핵심 아키텍처 구성 요소에 대한 개요도 제공합니다. 이 지식은 테이블 속성, 클러스터 구성 및 쿼리 구조와 같은 쿼리 성능 요인에 대한 이해와 함께 Amazon Redshift 데이터 웨어하우스를 위한 효율적이고 효과적인 테이블 및 쿼리를 설계하는 데 도움이 될 수 있습니다.

수강 대상

이 가이드는 Amazon Redshift에서 테이블 및 쿼리를 설계하거나 사용하는 데이터 엔지니어, 데이터 아키텍트 및 데이터 분석가를 대상으로 합니다.

목표

이 가이드는 사용자와 조직이 다음과 같은 목표를 수행하는 데 도움이 될 수 있습니다.

- 최적의 데이터 스토리지 및 검색 작업을 위한 테이블 설계
- 최적의 성능 및 비용 절감을 위한 쿼리 설계
- Amazon Simple Storage Service(Amazon S3)의 파일에서 직접 데이터를 쿼리하도록 Amazon Redshift Spectrum의 성능 최적화

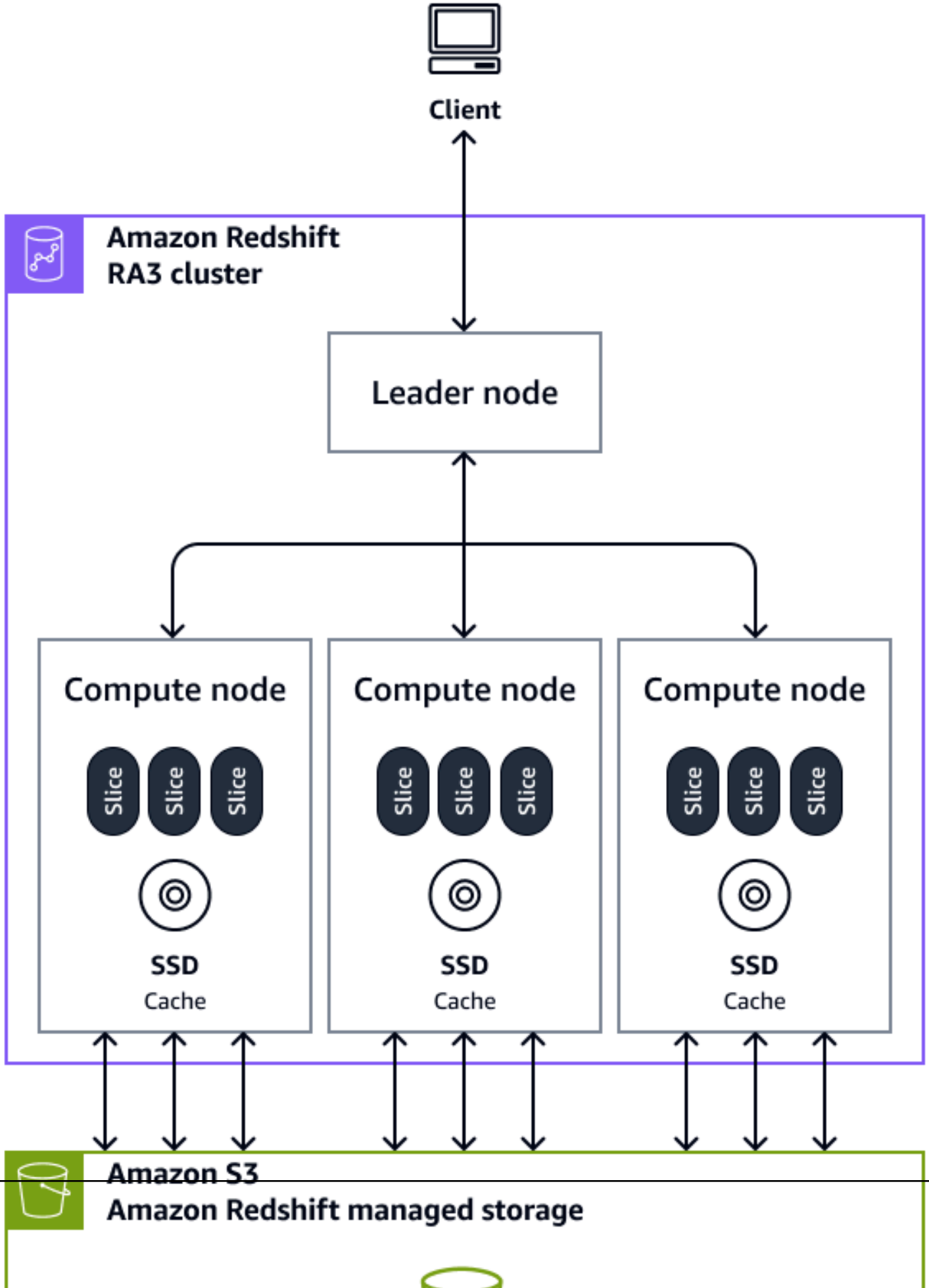
Amazon Redshift 데이터 웨어하우스 아키텍처 구성 요소

Amazon Redshift 데이터 웨어하우스의 핵심 아키텍처 구성 요소를 기본적으로 이해하는 것이 좋습니다. 이 지식은 최적의 성능을 위해 쿼리와 테이블을 설계하는 방법을 더 잘 이해하는 데 도움이 될 수 있습니다.

Amazon Redshift의 데이터 웨어하우스는 다음과 같은 핵심 아키텍처 구성 요소로 구성됩니다.

- 클러스터 - 하나 이상의 컴퓨팅 노드로 구성된 클러스터는 Amazon Redshift 데이터 웨어하우스의 핵심 인프라 구성 요소입니다. 컴퓨팅 노드는 외부 애플리케이션에 투명하지만 클라이언트 애플리케이션은 리더 노드와만 직접 상호 작용합니다. 일반적인 클러스터에는 두 개 이상의 컴퓨팅 노드가 있습니다. 컴퓨팅 노드는 리더 노드를 통해 조정됩니다.
- 리더 노드 - 리더 노드는 클라이언트 프로그램 및 모든 컴퓨팅 노드에 대한 통신을 관리합니다. 또한 리더 노드는 쿼리가 클러스터에 제출될 때마다 쿼리 실행 계획을 준비합니다. 계획이 준비되면 리더 노드는 코드를 컴파일하고 컴파일된 코드를 컴퓨팅 노드로 배포한 후 쿼리 결과를 처리하도록 각 컴퓨팅 노드에 데이터 조각을 할당합니다.
- 컴퓨팅 노드 - 컴퓨팅 노드에서 쿼리를 실행합니다. 리더 노드는 쿼리를 실행하는 계획의 개별 요소에 대해 코드를 컴파일하고 개별 컴퓨팅 노드에 코드를 할당합니다. 그러면 컴퓨팅 노드가 컴파일 코드를 실행한 후 최종 집계를 위해 중간 결과를 리더 노드에 다시 보냅니다. 각 컴퓨팅 노드에 전용 CPU와 메모리 그리고 연결된 디스크 스토리지가 있습니다. 그래도 워크로드가 증가하더라도 노드 수를 늘리거나, 노드 유형을 업그레이드하거나, 혹은 두 가지 방법 모두 사용하여 클러스터의 컴퓨팅 용량과 스토리지 용량을 늘릴 수 있습니다.
- 노드 조각 - 컴퓨팅 노드는 조각이라는 단위로 분할됩니다. 컴퓨팅 노드의 분할된 조각은 노드의 메모리 및 디스크 공간의 일부로 할당되고, 여기에서 노드에 할당되는 워크로드 일부를 처리합니다. 그러면 각 조각이 병렬 방식으로 실행되어 작업을 완료합니다. 데이터는 특정 테이블의 분산 스타일 및 분산 키를 기반으로 여러 조각에서 분산됩니다. 데이터를 균등하게 배포하면 Amazon Redshift가 조각에 워크로드를 균등하게 할당하고 병렬 처리의 이점을 극대화할 수 있습니다. 컴퓨팅 노드당 조각 수는 노드 유형에 따라 결정됩니다. 자세한 내용은 Amazon Redshift 설명서의 Amazon Redshift의 클러스터 및 노드를 참조하세요.
- 대량 병렬 처리(MPP) - Amazon Redshift는 MPP 아키텍처를 사용하여 복잡한 쿼리와 방대한 양의 데이터도 빠르게 처리합니다. 여러 컴퓨팅 노드가 데이터의 일부에서 동일한 쿼리 코드를 실행하여 병렬 처리를 극대화합니다.
- 클라이언트 애플리케이션 - Amazon Redshift는 다양한 데이터 로드, 데이터 추출, 전환, 적재(ETL), 비즈니스 인텔리전스(BI) 보고, 데이터 마이닝 및 분석 도구와 통합됩니다. 모든 클라이언트 애플리케이션은 리더 노드를 통해서만 클러스터와 통신합니다.

다음 다이어그램에서는 Amazon Redshift 데이터 웨어하우스의 아키텍처 구성 요소가 함께 작동하여 쿼리를 가속화하는 방법을 보여줍니다.


Client
Amazon Redshift
RA3 cluster
Leader node
Compute node
Slice
Slice
Slice
SSD
Cache
Compute node
Slice
Slice
Slice
SSD
Cache
Compute node
Slice
Slice
Slice
SSD
Cache
Amazon S3
Amazon Redshift managed storage

쿼리 수명 주기에는 7단계가 있습니다.

1. 쿼리 수신 및 구문 분석:
   - 리더 노드가 쿼리를 수신하고 SQL 구문을 분석합니다.
   - 구문 분석기에서 원래 쿼리에 대한 논리적 구조를 나타내는 초기 쿼리 트리를 생성합니다.
   - Amazon Redshift는 이 쿼리 트리를 쿼리 옵티마이저에 제공합니다.
2. 쿼리 최적화:
   - 옵티마이저가 쿼리를 평가하고 필요한 경우 효율성을 극대화하기 위해 쿼리를 재작성합니다.
   - 이 최적화 프로세스에는 여러 관련 쿼리를 생성하여 단일 쿼리를 대체하는 작업이 포함될 수도 있습니다.
3. 쿼리 계획 생성:
   - 옵티마이저는 실행을 위한 쿼리 계획(또는 필요한 경우 여러 계획)을 생성합니다.
   - 쿼리 계획에서는 조인 유형, 조인 순서, 집계 방법, 데이터 분산 요구 사항과 같은 실행 옵션을 지정합니다.
4. 실행 엔진 변환:
   - 실행 엔진이 쿼리 계획을 단계, 세그먼트 및 스트림으로 변환합니다.
     - 단계 - 쿼리 실행 중에 필요한 개별 작업을 나타냅니다. 컴퓨팅 노드에서 쿼리, 조인 또는 기타 데이터베이스 작업을 수행할 수 있도록 단계를 결합할 수 있습니다.
     - 세그먼트 - 단일 프로세스에서 실행할 수 있는 여러 단계를 결합합니다. 이는 컴퓨팅 노드 조각에서 실행할 수 있는 가장 작은 컴파일 단위입니다. (조각은 Amazon Redshift의 병렬 처리 단위입니다.)
     - 스트림 - 사용 가능한 컴퓨팅 노드 조각에 분산된 세그먼트 모음.
   - 실행 엔진은 이러한 단계, 세그먼트 및 스트림을 기준으로 컴파일된 코드를 생성합니다. 컴파일된 코드는 해석된 코드보다 실행 속도가 빠르고 소비하는 컴퓨팅 용량도 적습니다.
   - 리더 노드는 컴파일된 코드를 컴퓨팅 노드로 브로드캐스트합니다.
5. 병렬 실행:
   - 이 단계는 각 스트림에 대해 한 번 수행됩니다.
   - 컴퓨팅 노드 조각에서 쿼리 세그먼트를 병렬로 실행합니다.
   - 이때 Amazon Redshift는 최적화된 네트워크 통신, 메모리 사용, 디스크 관리를 최적화하여 한 쿼리 계획 단계의 중간 결과를 다음으로 전달합니다.
   - 이 최적화는 보다 빠른 쿼리 실행 속도에 기여합니다.
6. 스트림 처리:

- 이 단계는 각 스트림에 대해 한 번 수행됩니다.
- 엔진은 효율적인 병렬 처리를 위해 각 스트림에 대해 실행 가능한 세그먼트를 생성합니다.

7. 최종 정렬 및 집계:
   - 리더 노드는 쿼리에 필요한 모든 최종 정렬 또는 집계를 처리합니다.
   - 완료 후 리더 노드는 결과를 클라이언트로 반환합니다.

시스템 아키텍처에 대한 자세한 내용은 데이터 웨어하우스 시스템 아키텍처를 참조하세요.

Amazon Redshift의 쿼리 성능 요소

쿼리 성능에 영향을 미칠 수 있는 요인은 매우 많습니다. 다음에 나오는 데이터, 클러스터 및 데이터베이스 작업 측면은 모두 쿼리를 빠르게 처리하는 데 일조합니다.

- 테이블 속성
  - 정렬 키(Amazon Redshift Advisor)
  - 데이터 압축(자동화됨)
  - 데이터 분산(자동화됨)
  - 테이블 유지 관리(자동화됨)
- 클러스터 구성
  - 노드 유형
  - 노드 크기, 노드 수 및 조각
  - 워크로드 관리(자동화됨)
  - 단기 쿼리 가속화(자동화됨)
- SQL 쿼리
  - 쿼리 구조
  - 코드 컴파일

테이블 속성

Amazon Redshift 테이블은 Amazon Redshift에 데이터를 저장하는 기본 단위이며, 각 테이블에는 동작과 접근성을 결정하는 속성 세트가 있습니다. 이러한 속성에는 정렬, 분산 스타일, 압축 인코딩 등이 포함됩니다. Amazon Redshift 테이블의 성능, 보안 및 비용 효율성을 최적화하려면 이러한 속성을 이해해야 합니다.

정렬 키

Amazon Redshift는 테이블의 정렬 키에 따라 정렬된 순서대로 디스크에 데이터를 저장합니다. 쿼리 옵티마이저와 쿼리 프로세서는 컴퓨팅 노드 내 데이터 배치 정보를 사용하여 스캔해야 하는 블록 수를 줄입니다. 이를 통해 처리할 데이터의 양을 줄여 쿼리 속도가 크게 향상됩니다. `WHERE` 절의 필터를 쉽게 사용하기 위해 정렬 키를 사용하는 것이 좋습니다. 자세한 내용은 Amazon Redshift 설명서의 정렬 키 작업을 참조하세요.

데이터 압축

데이터 압축은 스토리지 요구 사항을 줄여 디스크 I/O를 감소시키고 쿼리 성능을 개선합니다. 쿼리를 실행하면 압축된 데이터를 메모리로 읽은 후 쿼리 실행 도중 압축 해제합니다. 적은 데이터를 메모리에 로드하면 Amazon Redshift는 더 많은 메모리를 할당하여 데이터를 분석할 수 있습니다. 원주형 스토리지는 유사한 데이터를 순차적으로 저장하기 때문에 Amazon Redshift는 원주형 데이터 형식에 특별히 연결된 적응형 압축 인코딩을 적용할 수 있습니다. 테이블 열에서 데이터 압축을 활성화하는 가장 좋은 방법은 데이터로 테이블을 로드할 때 Amazon Redshift에서 최적의 압축 인코딩을 적용하는 `AUTO` 옵션을 사용하는 것입니다. 자동 데이터 압축 사용에 대한 자세한 내용은 Amazon Redshift 설명서의 자동 압축을 사용하여 테이블 로드를 참조하세요.

데이터 분산

Amazon Redshift는 테이블의 배포 스타일에 따라 테이블 데이터를 컴퓨팅 노드에 저장합니다. 쿼리를 실행하면 쿼리 옵티마이저가 필요에 따라 데이터를 컴퓨팅 노드로 다시 분산시켜 조인 및 집계를 실행합니다. 테이블에 적합한 분산 스타일을 선택하면 조인에 앞서서 데이터를 필요한 것에 배치하여 재분산 단계의 영향을 최소화하는 데 효과적입니다. 가장 일반적인 조인을 쉽게 사용하기 위해 배포 키를 사용하는 것이 좋습니다. 자세한 내용은 Amazon Redshift 설명서의 데이터 분산 스타일 작업을 참조하세요.

테이블 유지 관리

Amazon Redshift는 대부분의 워크로드에 대해 바로 업계 최고의 성능을 제공하지만 Amazon Redshift 클러스터를 잘 실행하려면 유지 관리가 필요합니다. 데이터를 업데이트하고 삭제하면 배큠 처리해야 하는 데드 행이 생성되며 추가 순서가 정렬 키와 일치하지 않는 경우 추가 전용 테이블도 재정렬해야 합니다.

Vacuum

Amazon Redshift의 배큠 처리 프로세스는 Amazon Redshift 클러스터의 상태 및 유지 관리에 필수적입니다. 또한 쿼리 성능에도 영향을 미칩니다. 삭제 및 업데이트가 모두 이전 데이터에 플래그를 지정하지만 실제로 제거하지는 않으므로 이전 `UPDATE` 및 `DELETE` 작어에 의해 삭제할 항목으로 표시된 테이블 행이 차지하는 디스크 공간을 회수하기 위해 배큠 처리를 사용해야 합니다. Amazon Redshift는 자동으로 정렬하고 백그라운드의 테이블에서 `VACUUM DELETE` 작업을 수행할 수 있습니다.

로드 또는 일련의 증분적 업데이트 후 테이블을 정리하기 위해 전체 데이터베이스나 개별 테이블에 대해 `VACUUM` 명령을 실행할 수도 있습니다. 테이블에 정렬 키가 있고 삽입 시 정렬하도록 테이블 로드가 최적화되지 않은 경우 배큠을 사용하여 데이터를 재정렬해야 합니다(성능에 중요할 수 있음). 자세한 내용은 Amazon Redshift 설명서의 테이블 Vacuum을 참조하세요.

분석

ANALYZE 작업은 Amazon Redshift 데이터베이스의 테이블에 대한 통계 메타데이터를 업데이트합니다. 통계를 최신 상태로 유지하면 쿼리 플래너가 최적의 계획을 선택할 수 있으므로 쿼리 성능이 향상됩니다. Amazon Redshift는 데이터베이스를 지속적으로 모니터링하고 백그라운드에서 분석 작업을 자동으로 수행합니다. 시스템 성능에 대한 영향을 최소화하기 위해 ANALYZE 작업은 워크로드가 적은 기간에 자동으로 실행됩니다. ANALYZE를 명시적으로 실행하도록 선택한 경우 다음을 수행합니다.

- 쿼리를 실행하기 전에 ANALYZE 명령을 실행합니다.
- 정기적인 로드 또는 업데이트 사이클이 끝날 때마다 일상적으로 데이터베이스에 대해 ANALYZE 명령을 실행합니다.
- 새로 생성한 모든 테이블과 많이 변경된 모든 기존 테이블이나 열에 대해 ANALYZE 명령을 실행합니다.
- 쿼리에서의 사용 및 변경 경향에 따라 테이블과 열의 형식마다 일정을 달리하여 ANALYZE 작업을 실행하는 방법을 고려합니다.
- ANALYZE 명령을 실행할 때 시간과 클러스터 리소스를 절약하려면 PREDICATE COLUMNS 절을 사용합니다.

클러스터 구성

클러스터는 데이터의 실제 저장 및 처리를 수행하는 노드 모음입니다. 다음을 달성하려는 경우 Amazon Redshift 클러스터를 올바른 방식으로 설정하는 것이 중요합니다.

- 높은 확장성 및 동시성
- Amazon Redshift의 효율적인 사용
- 향상된 성능
- 비용 절감

노드 유형

Amazon Redshift 클러스터는 여러 노드 유형(RA3, DC2 및 DS2) 중 하나를 사용할 수 있습니다. 각 노드 유형마다 크기와 제한이 다르기 때문에 상황에 따라 클러스터를 확장하는 데도 좋습니다. 노드 크기는 스토리지 용량, 메모리, CPU, 그리고 클러스터의 각 노드 요금을 결정합니다. 비용 및 성능 최적화는 올바른 노드 유형과 크기를 선택하는 것으로 시작됩니다. 노드 유형에 대한 자세한 내용은 Amazon Redshift 설명서의 Amazon Redshift 클러스터 개요를 참조하세요.

노드 크기, 노드 수 및 조각

컴퓨팅 노드는 다수의 조각으로 분할됩니다. 노드가 많아질수록 프로세서와 조각의 수도 늘어난다는 것을 의미합니다. 그러면 쿼리를 다수의 조각으로 나눠 동시에 실행하기 때문에 쿼리의 처리 속도를 높일 수 있습니다. 그러나 노드가 많을수록 비용도 증가합니다. 즉, 시스템에 적합한 비용과 성능의 균형을 찾아야 합니다. Amazon Redshift 클러스터 아키텍처에 대한 자세한 내용은 Amazon Redshift 설명서의 데이터 웨어하우스 시스템 아키텍처를 참조하세요.

워크로드 관리

Amazon Redshift 워크로드 관리(WLM)를 통해 사용자는 우선순위가 지정된 워크로드 대기열을 유연하게 관리할 수 있으므로 빠른 단기 실행 쿼리가 대기열에서 장기 실행 쿼리 뒤에 모물지 않습니다. 자동 WLM은 기계 학습(ML) 알고리즘을 사용하여 쿼리를 프로파일링하고 적절한 리소스를 사용하여 적절한 대기열에 배치하는 동시에 쿼리 동시성 및 메모리 할당을 관리합니다. WLM에 대한 자세한 내용은 Amazon Redshift 설명서의 워크로드 관리 구현을 참조하세요.

단기 쿼리 가속화

단기 쿼리 가속화(SQA)는 선택한 단기 실행 쿼리를 장기 실행 쿼리보다 우선합니다. SQA 쿼리가 대기열에서 장기 쿼리 뒤에서 대기하지 않도록 SQA는 전용 공간에서 쿼리를 실행합니다. SQA는 단기 실행되고 사용자 정의 대기열에 있는 쿼리에만 우선순위를 부여합니다. SQA를 사용하는 경우 단기 실행 쿼리가 더 빠르게 실행되기 시작하며 사용자가 더 빨리 결과를 확인할 수 있습니다. SQA를 활성화하면 단기 쿼리 실행 전용 WLM 대기열을 축소하거나 제거할 수 있습니다. 또한 장기 실행 쿼리는 WLM 대기열의 슬롯을 두고 경합하지 않아도 됩니다. 즉, 쿼리 슬롯을 더 적게 사용하도록 WLM 대기열을 구성할 수 있습니다. 더 적은 동시성을 사용하면 대부분의 워크로드에 대해 쿼리 처리량이 증가되고 전체 시스템 성능이 향상됩니다. SQA에 대한 자세한 내용은 Amazon Redshift 설명서의 단기 쿼리 가속화 작업을 참조하세요.

SQL 쿼리

데이터베이스 쿼리는 데이터베이스의 데이터에 대한 요청입니다. 요청은 SQL을 사용하여 Amazon Redshift 클러스터에 들어와야 합니다. Amazon Redshift는 Java Database Connectivity(JDBC) 및 Open Database Connectivity(ODBC)를 통해 연결하는 SQL 클라이언트 도구를 지원합니다. JDBC 또는 ODBC 드라이버를 지원하는 SQL 클라이언트 도구라면 대부분 사용할 수 있습니다.

쿼리 구조

쿼리 작성 방식 또한 성능에 큰 영향을 미칩니다. 필요에 따라 필요한 최소의 데이터를 처리하고 반환하는 쿼리를 작성하는 것이 좋습니다. 쿼리를 구성하는 방법에 대한 자세한 내용은 이 가이드의 Amazon Redshift 쿼리 설계 모범 사례 섹션을 참조하세요.

코드 컴파일

Amazon Redshift는 각 쿼리 실행 계획에 최적화된 코드를 생성하고 컴파일합니다. 컴파일 코드는 인터프리터 사용에 따른 오버헤드를 제거하기 때문에 실행 속도가 더욱 빠릅니다. 컴파일된 코드의 성능 이점을 유지하면서 새 쿼리의 지연 시간을 최소화하기 위해 Amazon Redshift는 구성이라는 기술을 사용합니다. 구성은 새로운 쿼리를 즉시 처리하는 동시에 고도로 최적화된 쿼리별 코드를 백그라운드에서 컴파일하는 기존 로직의 간단한 배열을 생성합니다. 이렇게 하면 쿼리 실행의 중요한 경로에서 컴파일이 제거됩니다. 즉, 새 쿼리가 더 빠르게 시작되고 후속 실행과 일치하는 성능을 제공합니다.

또한 Amazon Redshift는 서버리스 컴파일 서비스를 사용하여 Amazon Redshift 클러스터의 컴퓨팅 리소스 이상으로 쿼리 컴파일을 확장합니다. 컴파일된 코드 세그먼트는 클러스터에 로컬로 캐시되고 클러스터 재부팅 후에도 유지되는 사실상 무제한의 원격 캐시에 캐시됩니다. 동일한 쿼리의 후속 실행은 컴파일 단계를 건너뛸 수 있기 때문에 더 빠르게 실행됩니다. Amazon Redshift는 확장 가능한 컴파일 서비스를 사용하여 코드를 병렬로 컴파일하여 일관되게 빠른 성능을 제공합니다.

Amazon Redshift 테이블 설계 모범 사례

이 섹션에서는 데이터베이스 테이블 설계에 대한 모범 사례 개요를 제공합니다. 최적의 쿼리 성능과 효율성을 달성하려면 다음 모범 사례를 따르는 것이 좋습니다.

정렬 키 작동 방식 이해

Amazon Redshift는 정렬 키에 따라 정렬된 순서로 디스크에 데이터를 저장합니다. Amazon Redshift 쿼리 옵티마이저는 최적의 쿼리 계획을 결정할 때 정렬 순서를 사용합니다. 정렬 키를 효과적으로 사용하려면 다음을 수행하는 것이 좋습니다.

- 테이블을 최대한 정렬된 상태로 유지합니다.
- VACUUM 정렬을 사용하여 최적의 성능을 복원합니다.
- 정렬 키 열은 압축하지 마세요.
- 정렬 키가 압축되고 sortkey1_skew 비율이 상당히 높으면 정렬 키에서 압축을 활성화하지 않고 테이블을 다시 생성합니다.
- 정렬 키 열에 함수를 적용하지 마세요. 예를 들어 다음 쿼리에서는 DATE로 캐스팅하는 경우 trans_dt : TIMESTAMPTZ 정렬 키 열이 사용되지 않습니다.

```
select order_id, order_amt
from sales
where trans_dt::date = '2021-01-08'::date
```

- 정렬 키 순서로 INSERT 작업을 수행합니다.
- 가능하면 GROUP BY 절에서 정렬 키를 사용합니다.

쿼리 조정 팁

쿼리를 조정하려면 다음을 수행하는 것이 좋습니다.

- 최적의 효과를 위해 항상 복합 정렬 키를 최저 카디널리티에서 최고 카디널리티로 정렬합니다.
- 복합 정렬 키의 선행 키가 상대적으로 고유한 경우(즉, 카디널리티가 높은 경우) 정렬 키에 열을 더 추가하지 마세요. 열을 더 추가하면 쿼리 성능에는 거의 영향을 미치지 않지만 유지 관리 비용이 추가됩니다.

정렬 키 효과 평가

쿼리를 최적화하려면 쿼리의 효과를 평가할 수 있어야 합니다. SVL_QUERY_SUMMARY 보기를 사용하여 쿼리 실행에 대한 일반 정보를 찾는 것이 좋습니다. 이 보기에서 IS_RRSCAN 속성을 사용하여 EXPLAIN 계획 단계에서 범위 제한 스캔을 사용하는지 확인할 수 있습니다. rows_pre_filter 속성을 사용하여 정렬 키의 선택성을 결정할 수도 있습니다.

v_my_last_query_summary라는 GitHub의 관리 보기를 사용할 수도 있습니다. 보기에는 실행된 마지막 쿼리에 대한 정보가 표시됩니다.

다음 명령문은 쿼리 실행에 대한 일반적인 정보를 찾는 방법을 보여줍니다.

```
select lpad(' ',stm+seg+step) || label as label,
       rows,
       bytes,
       is_diskbased,
       is_rrscan,
       rows_pre_filter
from svl_query_summary
where query = pg_last_query_id()
order by stm, seg, step;
```

이전 쿼리에서는 다음과 같은 출력 샘플을 반환합니다.

label	rows	bytes	is_diskbased	is_rrscan	rows_pre_filter
scan tbl=163860 name=orders	1500000	24000000	f	f	1500000
project	1500000	0	f	f	0
project	1500000	0	f	f	0
hash tbl=968	1500000	24000000	f	f	0
scan tbl=163852 name=lineitem	6001215	144029160	f	t	6001215
project	6001215	0	f	f	0
project	6001215	0	f	f	0
hjoin tbl=968	6001215	0	f	f	0
project	6001215	0	f	f	0
project	6001215	0	f	f	0

테이블 이해

테이블의 중요한 속성을 이해하는 것이 중요합니다. 테이블에 대해 자세히 알아보려면 다음을 수행합니다.

- PG_TABLE_DEF를 사용하여 테이블 열에 대한 정보를 봅니다.
- SVV_TABLE_INFO를 사용하여 데이터 분배 스큐, 키 분배 스큐, 테이블 크기, 통계 등 테이블에 대해 더욱 포괄적인 정보를 봅니다.

올바른 테이블 분배 스타일 선택

쿼리를 실행하면 쿼리 옵티마이저가 필요에 따라 쿼리 행을 컴퓨팅 노드로 다시 분산시켜 조인 및 집계를 실행합니다. 테이블 배포 스타일을 선택하는 목적은 쿼리 실행 이전에 데이터의 배포 위치를 지정하여 재배포 단계의 영향을 최소화하는 데 있습니다.

올바른 테이블 분배 스타일을 선택하려면 다음과 같은 접근 방식을 사용하는 것이 좋습니다.

- 동일한 노드 내에 행을 공동 배치하여 쿼리 실행 계획에서 브로드캐스팅 및 재분배를 방지합니다. 예를 들어 DISTKEY를 선택하면 팩트 테이블과 1차원 테이블을 공통 열에 분배할 수 있습니다. 필터링된 데이터 집합의 크기를 기준으로 가장 큰 차원을 선택합니다. 조인에 사용되는 행만 분산시켜야 하기 때문에 테이블 크기가 아닌 필터링 이후 데이터 세트의 크기를 고려합니다.
- 분배 키가 생성되는 열에 스큐가 없는지 확인합니다. 그렇지 않으면 한 컴퓨팅 노드가 다른 컴퓨팅 노드보다 더 많은 작업을 수행할 수 있습니다. 스큐 현상을 발견하면 분산 키 열 변경을 고려합니다. 열의 값이 균일하게 분배되거나 해당 카디널 값이 더 높은 경우 열을 분배 키의 후보로 간주할 수 있습니다.
- 조인 조건에 사용되는 테이블이 작으면(1GB 미만) ALL 분배 스타일을 고려합니다.
- 분배 키는 압축할 수 있지만 정렬 키 열(특히 정렬 키의 첫 번째 열)을 압축하지 않아야 합니다.

> **Note**
>
> 자동 테이블 최적화를 사용하는 경우 테이블의 분배 스타일을 선택할 필요가 없습니다. 자세한 내용은 Amazon Redshift 설명서의 자동 테이블 최적화 작업을 참조하세요. Amazon Redshift에서 적절한 배포 스타일을 선택하도록 하려면 배포 스타일에 AUTO를 지정합니다.

Amazon Redshift 쿼리 설계 모범 사례

이 섹션에서는 쿼리 설계에 대한 모범 사례 개요를 제공합니다. 최적의 쿼리 성능과 효율성을 달성하려면 이 섹션의 모범 사례를 따르는 것이 좋습니다.

SELECT * FROM 문을 사용하지 마세요.

SELECT * FROM 문을 사용하지 않는 것이 좋습니다. 대신 분석을 위해 항상 열을 나열합니다. 그러면 쿼리 실행 시간이 단축되고 Amazon Redshift Spectrum 쿼리에 대한 스캔 비용이 절감됩니다.

피해야 할 사항 예제

```
select *
from sales;
```

모범 사례 예제

```
select sales_date, sales_amt
from sales;
```

쿼리 문제 식별

STL_ALERT_EVENT_LOG 보기를 확인하여 쿼리에 발생할 수 있는 문제를 식별하고 수정하는 것이 좋습니다.

쿼리에 대한 요약 정보 가져오기

SVL_QUERY_SUMMARY 및 SVL_QUERY_REPORT 보기를 사용하여 쿼리에 대한 요약 정보를 가져오는 것이 좋습니다. 이 정보를 사용하여 쿼리를 최적화할 수 있습니다.

교차 조인 방지

반드시 필요한 경우가 아니라면 루트로 실행하는 것은 피하는 것이 좋습니다. 교차 조인에는 조인 조건이 없기 때문에 두 테이블의 카테시안 곱이 발생하는 원인이 됩니다. 교차 조인은 일반적으로 중첩 루프 조인으로 실행됩니다(가능한 조인 유형 중에서 속도가 가장 느린 조인).

피해야 할 사항 예제

```
select c.c_name,
          n.n_name
from tpch.customer c,
        tpch.nation n;
```

모범 사례 예제

```
select c.c_name,
           n.n_name
from tpch.customer c,
join tpch.nation n
  on n.n_nationkey = c.c_nationkey;
```

쿼리 조건자에서 함수 자제

쿼리 조건자에 함수를 사용하지 않는 것이 좋습니다. 쿼리 조건자에서 함수를 사용하면 일반적으로 함수가 각 행에 처리 오버헤드를 추가하고 쿼리의 전체 실행 속도를 늦출 수 있으므로 성능에 부정적인 영향을 미칠 수 있습니다.

피해야 할 사항 예제

```
select sum(o_totalprice)
from tpch.orders
where datepart(year, o_orderdate) = 1992;
```

모범 사례 예제

```
select sum(o_totalprice)
from tpch.orders
where o_orderdate between '1992-01-01' and '1992-12-31';
```

불필요한 캐스트 변환 자제

데이터 유형을 캐스팅하는 경우 시간과 리소스가 들어가고 쿼리 실행 속도가 느려지므로 쿼리에 불필요한 캐스팅 변환을 사용하지 않는 것이 좋습니다.

피해야 할 사항 예제

```
select sum(o_totalprice)
from tpch.orders
where o_ordertime::date = '1992-01-01';
```

모범 사례 예제

```
select sum(o_totalprice)
from tpch.orders
where o_ordertime between '1992-01-01 00:00:00' and '1992-12-31 23:59:59';
```

복잡한 집계에 CASE 표현식 사용

동일한 테이블에서 여러 번 선택하는 대신 CASE 표현식을 사용하여 복잡한 집계를 수행하는 것이 좋습니다.

피해야 할 사항 예제

```
select sum(sales_amt) as us_sales
from sales
where country = 'US';

select sum(sales_amt) as ca_sales
from sales
where country = 'CA';
```

모범 사례 예제

```
select sum(case when country = 'US' then sales_amt end) as us_sales,
           sum(case when country = 'CA' then sales_amt end) as ca_sales
from sales;
```

하위 쿼리 사용

쿼리에서 한 테이블만 조건자 조건에 사용되는 경우에는 하위 쿼리를 사용하는 것이 좋습니다. 그러면 하위 쿼리가 소수의 행(약 200개 미만)을 반환합니다.

피해야 할 사항 예제

하위 쿼리가 200개 미만의 행을 반환하는 경우:

```
select sum(order_amt) as total_sales
from sales
where region_key IN
        (select region_key
         from regions
         where state = 'CA');
```

모범 사례 예제

하위 쿼리가 200개 이상의 행을 반환하는 경우:

```
select sum(o.order_amt) as total_sales
from sales o
join regions r
  on r.region_key = o.region_key
  and r.state = 'CA';
```

조건자 사용

조건자를 사용하여 최대한 많은 데이터세트를 제한하는 것이 좋습니다. 조건자는 SQL에서 쿼리에 반환되는 데이터를 필터링하고 제한하는 데 사용됩니다. 조건자에 조건을 지정하여 지정된 조건을 기반으로 쿼리 결과에 포함할 행을 지정할 수 있습니다. 그러면 관심 있는 데이터만 검색할 수 있으며 쿼리의 효율성과 정확도가 향상됩니다. 자세한 내용은 Amazon Redshift 설명서에서 조건을 참조하세요.

조인으로 테이블을 필터링하는 조건자 추가

조건자에서 동일한 필터를 적용하더라도 조건자를 추가하여 조인에 참여하는 테이블을 필터링하는 것이 좋습니다. 조건자를 사용하여 SQL에서 조인으로 테이블을 필터링하면 처리해야 하는 데이터의 양을 줄이고 중간 결과 세트의 크기를 줄여 쿼리 성능을 개선할 수 있습니다. WHERE 절에서 조인 작업 조건을 지정하면 쿼리 실행 엔진이 조인되기 전에 조건과 일치하지 않는 행을 제거할 수 있습니다. 이렇게 하면 결과 세트가 작아지고 쿼리 실행이 빨라집니다.

피해야 할 사항 예제

```
select p.product_name, sum(o.order_amt)
from sales o
join product p
   on r.product_key = o.product_key
where o.order_date > '2022-01-01';
```

모범 사례 예제

```
select p.product_name, sum(o.order_amt)
from sales o
join product p
  on p.product_key = o.product_key
  and p.added_date > '2022-01-01'
where o.order_date > '2022-01-01';
```

조건자에 가장 저렴한 연산자 사용

조건자에서 가능한 가장 비용이 적게 드는 연산자를 사용합니다. 비교 조건 연산자는 LIKE 연산자보다 선호됩니다. LIKE 연산자는 여전히 SIMILAR TO 또는 POSIX 연산자보다 선호됩니다.

GROUP BY 절에서 정렬 키 사용

쿼리 플래너의 집계 효율을 높일 수 있도록 GROUP BY 절에 정렬 키를 사용합니다. GROUP BY 목록에 정렬 키 열만 포함되어 있으면(이중 하나는 분산 키임) 쿼리 실행 시 1단계 집계가 가능합니다. GROUP BY 목록의 정렬 키 열에는 1차 정렬 키, 정렬 키 순서에 사용할 다른 정렬 키를 순서대로 포함해야 합니다.

구체화된 뷰의 장점

가능하면 복잡한 코드를 구체화된 보기로 대체하여 쿼리를 다시 작성하면 쿼리 성능이 크게 향상됩니다. 자세한 내용은 Amazon Redshift 설명서의 Amazon Redshift에서 구체화된 뷰 만들기를 참조하세요.

GROUP BY 및 ORDER BY 절의 열에 주의하세요.

GROUP BY 및 ORDER BY 절을 모두 사용하는 경우 GROUP BY 및 ORDER BY 절 모두에서 동일한 순서로 열을 배치해야 합니다. GROUP BY는 암묵적으로 데이터를 정렬해야 합니다. ORDER BY 절이 다른 경우 데이터를 두 번 정렬해야 합니다.

피해야 할 사항 예제

```
select a, b, c, sum(d)
from a_table
group by b, c, a
```

```
order by a, b, c
```

모범 사례 예제

```
select a, b, c, sum(d)
from a_table
group by a, b, c
order by a, b, c
```

Amazon Redshift Spectrum 사용 모범 사례

이 섹션에서는 Amazon Redshift Spectrum 사용 모범 사례에 대한 개요를 제공합니다. Redshift Spectrum을 사용할 때 최적의 성능을 얻으려면 다음 모범 사례를 따르는 것이 좋습니다.

- 파일 유형은 Redshift Spectrum 쿼리 성능에 상당한 영향을 미친다고 가정합니다. 성능을 개선하려면 ORC 또는 Parquet과 같은 열 기반 인코딩 파일을 사용하고 매우 작은 차원 테이블에만 CSV 형식을 사용합니다.
- 접두사 기반 분할을 사용하여 파티션 정리를 활용합니다. 즉, 데이터 레이크의 파티션에 키가 지정된 필터를 사용해야 합니다.
- Redshift Spectrum은 대규모 요청을 처리하기 위해 자동으로 규모를 조정하므로 Redshift Spectrum에서 최대한 많은 작업을 수행합니다(예: 조건자 푸시다운).
- 자주 필터링되는 열의 파일을 분할할 때 주의하세요. 데이터가 하나 이상의 필터링된 열로 분할된 경우 Redshift Spectrum은 파티션 정리를 활용하고 불필요한 파티션 및 파일 스캔을 건너뛸 수 있습니다. 일반적인 관행은 시간을 기준으로 데이터를 파티셔닝하는 것입니다.
- 다음 쿼리를 사용하여 파티션의 효과와 Redshift Spectrum 쿼리의 효율성을 확인할 수 있습니다.

```
Select query,
           segment,
           max(assigned_partitions) as total_partitions,
           max(qualified_partitions) as qualified_partitions
From svl_s3partition
Where query=pg_last_query_id()
Group by 1,2;
```

  위의 쿼리는 다음을 보여줍니다.

  - total_partitions -에서 인식하는 파티션 수 AWS Glue Data Catalog
  - qualified_partitions - Redshift Spectrum 쿼리에 대해 액세스되는 Amazon Simple Storage Service(Amazon S3)의 접두사 수
- 또한 SVL_S3QUERY_SUMMARY 시스템 테이블을 확인하여 파티션의 효과와 Redshift Spectrum 쿼리의 효율성에 대해 알아볼 수도 있습니다. 이렇게 하려면 다음 문을 사용하면 됩니다.

```
Select *
From svl_s3query_summary
Where query=pg_last_query_id();
```

이전 쿼리는 파티션 정리의 효율성을 보여주는 파일 외에도 `is_partitioned`, `s3_scanned_rows/bytes`, `s3_returned_rows/bytes` 값을 포함하여 훨씬 더 많은 정보를 반환합니다.

Redshift Spectrum에서 조건자 푸시다운

조건자 푸시다운을 사용하면 Amazon Redshift 클러스터에서 리소스를 소비하지 않습니다. 많은 SQL 작업을 Redshift Spectrum 계층으로 푸시다운할 수 있습니다. 가능하면 이를 활용하는 것이 좋습니다.

다음 사항에 유의하세요.

- Redshift Spectrum 계층 내에서 다음을 포함한 일부 유형의 SQL 작업을 완전히 평가할 수 있습니다.
  - `GROUP BY` 절
  - 비교 및 패턴 일치 조건(예: `LIKE`)
  - 집계 함수(예: `COUNT`, `SUM`, `AVG`, `MIN`, `MAX`)
  - `regex_replace`, `to_upper`, `date_trunc` 및 기타 함수
- `DISTINCT` 및 `ORDER BY`를 포함하여 일부 작업을 Redshift Spectrum 계층에 푸시할 수 없습니다. 리더 노드에서 정렬이 수행되므로 가능하면 쿼리의 최상위 수준에서만 `ORDER BY`를 수행합니다.
- 쿼리 `EXPLAIN` 계획을 검사하여 조건자 푸시다운이 효과적인지 확인합니다. `EXPLAIN` 명령에서 Redshift Spectrum 부분을 찾으려면 다음 단계를 찾습니다.
  - S3 Seq Scan
  - S3 HashAggregate
  - S3 Query Scan
  - Seq Scan PartitionInfo
  - Partition Loop
- 쿼리에서 가장 적은 수의 열을 사용합니다. Redshift Spectrum은 데이터가 Parquet 또는 ORC 형식인 경우 스캔을 위해 열을 제거할 수 있습니다.
- 병렬 처리 및 파티션 제거를 위해 파티션을 광범위하게 사용하고 가능한 경우 파일 크기를 최소 64MB로 유지합니다.
- `CREATE EXTERNAL TABLE` 또는 `ALTER TABLE`을 사용하는 경우 `TABLE PROPERTIES 'numRows'='nnn'`을 설정합니다. Amazon Redshift는 쿼리 옵티마이저가 쿼리 계획을 생성하는 데 사용하는 테이블 통계를 생성하기 위해 외부 테이블을 분석하지 않습니다. 통계가 설정되지 않는

경우 Amazon Redshift는 외부 테이블이 더 큰 테이블이고 로컬 테이블이 더 작은 테이블이라고 가정합니다.

Redshift Spectrum에 대한 쿼리 조정 팁

쿼리를 조정할 경우 다음 사항에 유의하는 것이 좋습니다.

- Amazon Redshift 클러스터가 쿼리에 대해 참여할 수 있는 Redshift Spectrum 노드 수는 클러스터의 조각 수와 연결됩니다.
- 클러스터 크기를 조정하면 클러스터의 로컬 컴퓨팅 프로파일, 스토리지 프로파일 및 Amazon S3 데이터 레이크 쿼리의 쿼리 기능에 도움이 될 수 있습니다.
- Amazon Redshift 쿼리 플래너는 가능하면 항상 조건자 및 집계를 Redshift Spectrum 쿼리 계층으로 푸시합니다.
- Amazon S3에서 대량의 데이터가 반환되면 클러스터의 리소스에 의해 처리가 제한됩니다.
- Redshift Spectrum은 대규모 요청을 처리하기 위해 자동으로 확장되므로 처리를 Redshift Spectrum 계층으로 푸시할 수 있을 때마다 전반적인 성능이 향상됩니다.

리소스

- Amazon Redshift의 보안(Amazon Redshift 설명서)
- Amazon Redshift 쿼리 설계 모범 사례(Amazon Redshift 설명서)
- 쿼리 성능 튜닝(Amazon Redshift 설명서)
- 쿼리 계획(Amazon Redshift 설명서)
- Amazon Redshift Spectrum 쿼리 성능 개선(Amazon Redshift 설명서)
- Amazon Redshift의 쿼리 수명 주기 이해(AWS 권장 가이드)

문서 기록

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

변경 사항	설명	날짜
AQUA 제거됨	고급 쿼리 액셀러레이터(AQUA)에 대한 정보를 제거했습니다.	2024년 6월 14일
최초 게시	—	2023년 2월 3일

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