RDBMS에서 Amazon DynamoDB로의 마이그레이션 모범 사례

적합한 워크로드에 NoSQL 성능 활용

Nathaniel Slater

2015년 3월

목차

요약

오늘날, 소프트웨어 아키텍트와 개발자들은 데이터 스토리지와 지속성을 다양하게 선택할 수 있습니다. 여기에는 기존 관계형 데이터베이스 관리 시스템(RDBMS)뿐 아니라 Amazon DynamoDB 같은 NoSQL 데이터베이스도 포함됩니다. 일부 워크로드는 NoSQL 솔루션을 사용하여 실행할 때 확장성과 비용 효율성이 높아집니다. 이 백서에서는 RDBMS에서 DynamoDB로 이러한 워크로드를 마이그레이션하는 모범 사례를 중점적으로 살펴봅니다. DynamoDB 같은 NoSQL 데이터베이스가 기존 RDBMS와 어떻게 다른지 설명하고, 분석, 데이터 모델링 및 RDBMS에서 DynamoDB로의 데이터 마이그레이션을 위한 프레임워크를 제안합니다.

서론

수십 년 동안 데이터 스토리지 및 지속성에서는 RDBMS가 사실상 유일한 선택지였습니다. 전자 상거래 웹사이트건 비용 보고 시스템이건 데이터 중심 애플리케이션이라면 십중팔구 관계형 데이터베이스를 사용하여 애플리케이션이 필요로 하는 데이터를 검색하고 저장했습니다. 여기에는 다음을 비롯한 수많은 이유가 있습니다.

- RDBMS는 완숙 단계에 접어든 안정된 기술입니다.
- 쿼리 언어인 SQL은 기능이 풍부하고 다양하게 활용할 수 있습니다.
- RDBMS 엔진을 실행하는 서버는 일반적으로 IT 인프라에서 가장 안정적이고 강력한 서버에 속합니다.
- 모든 주요 프로그래밍 언어에는 RDBMS와의 통신에 사용되는 드라이버 지원이 포함되어 있으며, 데이터베이스 중심 애플리케이션 개발을 간소화할 수 있는 풍부한 도구 세트를 갖추고 있습니다.

이러한 요인에 다른 많은 요인까지 더해져 RDBMS는 굳건한 위치를 지켜 왔습니다. 아키텍트와 소프트웨어 개발자들에게는 지금까지 데이터 스토리지와 지속성을 위한 합리적인 대안이 없었던 것입니다.

전자 상거래 및 소셜 미디어 같은 "인터넷 규모" 웹 애플리케이션의 성장, 스마트 폰과 태블릿 등 연결된 디바이스의 폭발적 증가, 빅 데이터의 부상으로 인해 전통적인 관계형 데이터베이스로는 처리하기 곤란한 새로운 워크로드가 등장했습니다. 트랜잭션 처리를 위해 설계된 시스템으로서 모든 RDBMS가 지원해야 하는 기본적 속성은 약어 ACID(Atomicity[원자성], Consistency[일관성], Isolation[격리성], Durability[내구성])로 정의됩니다. 원자성은 "전부 아니면 전무"를 뜻합니다. 즉, 트랜잭션이 완전히 실행되거나 전혀 실행되지 않습니다. 일관성은 트랜잭션 실행이 유효한 상태 변화를 초래함을 뜻합니다. 트랜잭션이 커밋되면 결과 데이터의 상태는 데이터베이스 스키마가 부과하는 제약을 준수해야 합니다. 격리성은 동시 트랜잭션들이 서로 분리되어 실행될 것을 요구합니다. 격리성 속성은 동시 트랜잭션들이 순차적으로 실행되는 경우, 데이터의 최종 상태가 동일함을 보장합니다. 내구성은 트랜잭션이 실행된 후 데이터 상태가 보존될 것을 요구합니다. 전원 차단이나 시스템 장애가 발생하는 경우, 데이터베이스는 알려진 마지막 상태로 복구될 수 있어야 합니다.

이러한 ACID 속성은 모두 바람직한 것이지만 이 네 가지 모두를 지원하려면 오늘날의 데이터 집약적 워크로드에서는 몇 가지 문제가 따르는 아키텍처가 필요합니다. 예를 들어 일관성에는 명확히 정의된 스키마가 필요하며, 데이터베이스에 저장된 모든 데이터는 이러한 스키마를 준수해야 합니다. 이는 임시 쿼리와 읽기 작업이 많은 워크로드에 매우 좋습니다. 게임 애플리케이션에서 게이머 상태 저장처럼 거의 전적으로 쓰기 작업으로 구성되는 워크로드의 경우, 이러한 스키마 적용은 스토리지 관점이나 컴퓨팅 관점에서 볼 때 비용이 많이 듭니다. 명확히 정의된 키 세트를 통해 서로 연관되는 행과 테이블에 이 데이터를 강제로 넣어서 게임 개발자가 얻는 이득은 거의 없습니다.

일관성은 또한 데이터를 수정하는 트랜잭션이 완료될 때까지 해당 데이터의 일부를 잠근 다음 변경 사항을 즉시 표시할 것을 요구합니다. 한 계좌에서 인출해서 다른 계좌에 입금하는 은행 트랜잭션에서는 이것이 필요합니다. 이런 유형의 트랜잭션을 "strongly consistent"라고 합니다. 반면 소셜 미디어 애플리케이션의 경우, 실제로 모든 사용자가 정확히 같은 시간에 데이터 피드 업데이트를 볼 수 있어야 한다는 요구 사항이 없습니다. 후자의 경우, 트랜잭션은 "eventually consistent"입니다. 소셜 미디어 애플리케이션은 데이터 변경 사항을 보게 되는 시간이 서로 달라지더라도 잠재적으로 수백만 명에 이르는 사용자를 처리할 수 있는 확장이 훨씬 중요합니다. strong consistency를 유지하면서 이런 수준의 동시성을 처리할 수 있게 RDBMS를 확장하려면 더욱 강력한(그리고 종종 독점인) 하드웨어로 업그레이드해야 합니다. 이를 "스케일링 업" 또는 "수직적 조정"이라고 하며, 대체로 엄청난 비용이 수반됩니다. 이런 수준의 동시성을 지원하도록 데이터베이스를 확장하는 보다 비용 효율적인 방법은 상용 하드웨어에서 실행되는 서버 인스턴스를 추가하는 것입니다. 이를 "스케일링 아웃" 또는 "수평적 조정"이라고 하며, 일반적으로 수직적 조정보다 훨씬 비용 효과적입니다.

Amazon DynamoDB와 같은 NoSQL 데이터베이스는 RDBMS에서 발견되는 확장 및 성능 문제를 해결합니다. “NoSQL”이라는 용어는 모든 현대 RDBMS의 기초가 되는, 1970년 논문 *'A Relational Model of Data for Large Shared Data Banks'*[1]에서 E.F Codd가 옹호한 관계형 모델을 따르지 않는 데이터베이스를 뜻합니다. 따라서 NoSQL 데이터베이스는 기존 RDBMS보다 기능과 특징 면에서 훨씬 다양합니다. SQL과 비슷한 공통 쿼리 언어가 없으며, 일반적으로 높은 I/O 성능과 수평적 확장성이 쿼리 유연성을 대신합니다. NoSQL 데이터베이스는 RDBMS와 같은 방식으로 스키마 개념을 적용하지 않습니다. 일부는 JSON 같은 반정형 데이터를 저장할 수 있고, 일부는 관련 값을 열 집합으로 저장할 수 있습니다. 단순히 키/값 쌍을 저장하는 경우도 있습니다.

그 결과, NoSQL 데이터베이스는 수평적으로 확장되는 훨씬 유연한 데이터 모델을 위해 RDBMS의 일부 쿼리 능력과 ACID 속성을 포기합니다. 이런 특징 때문에 비관계형 워크로드(앞서 언급한 게임 상태 예시와 같은)에 RDBMS를 사용하면 성능 병목 현상, 운영 복잡성, 비용 상승이 초래되는 상황에서는 NoSQL 데이터베이스가 탁월한 선택입니다. DynamoDB는 이 모든 문제에 대한 해법을 제공하며, RDBMS에서 이러한 워크로드를 마이그레이션하기에 탁월한 플랫폼입니다.

---

[1] http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf

Amazon DynamoDB 개요

Amazon DynamoDB는 AWS 클라우드에서 실행되는 완전 관리형 NoSQL 데이터베이스 서비스입니다. 엄청난 확장이 가능하고 분산된 NoSQL 데이터베이스를 실행하는 데 따른 복잡성은 서비스 자체에서 관리되므로 소프트웨어 개발자는 인프라 관리보다 애플리케이션 구축에 집중할 수 있습니다. NoSQL 데이터베이스는 확장을 염두에 두고 설계되었지만 아키텍처가 정교하기 때문에 대규모 NoSQL 클러스터 실행에는 상당한 운영 부담이 따를 수 있습니다. 개발자는 고급 분산 컴퓨팅 개념의 전문가가 될 필요 없이 원하는 프로그래밍 언어의 SDK를 사용하여 DynamoDB의 간단한 API를 배우기만 하면 됩니다.

DynamoDB는 사용이 쉬울 뿐 아니라 비용 효율적입니다. DynamoDB에서는 사용 중인 스토리지와 프로비저닝한 IO 처리량에 대해 요금을 지불합니다. DynamoDB는 탄력적으로 확장되도록 설계되었습니다. 애플리케이션의 스토리지 및 처리량 요구 사항이 낮으면 DynamoDB 서비스에서 소량의 용량만 프로비저닝하면 됩니다. 애플리케이션 사용자 수가 늘어나고 필요한 IO 처리량이 증가하면 그때그때 추가 용량을 프로비저닝할 수 있습니다. 이를 통해 애플리케이션이 원활하게 확장되어 데이터베이스에 초당 수천 건의 동시 요청을 하는 수백만의 사용자를 지원할 수 있습니다.

테이블은 DynamoDB에서 데이터를 구성하고 조직하기 위한 기본적 구성체입니다. 테이블은 항목으로 이루어집니다. 항목은 항목을 고유하게 식별하는 기본 키와 속성이라고 하는 키/값 쌍으로 구성됩니다. 항목은 RDBMS 테이블의 행과 비슷하지만 같은 DynamoDB 테이블의 모든 항목은 관계형 테이블의 모든 행이 같은 열을 공유하는 것처럼 같은 속성 세트를 공유할 필요가 없습니다. 그림 1은 DynamoDB 테이블의 구조와 테이블에 포함된 항목을 보여 줍니다. DynamoDB 테이블에는 열이라는 개념이 없습니다. 테이블의 각 항목은 최대 400K 크기의 임의 개수의 요소를 포함하는 튜플로 표현할 수 있습니다. 이 데이터 모델은 객체 직렬화와 분산 시스템에서의 메시징에 일반적으로 사용되는 형식으로 데이터를 저장하는 데 적합합니다. 다음 섹션에서 살펴보겠지만 이런 유형의 데이터가 포함된 워크로드는 DynamoDB로 마이그레이션하기에 좋습니다.

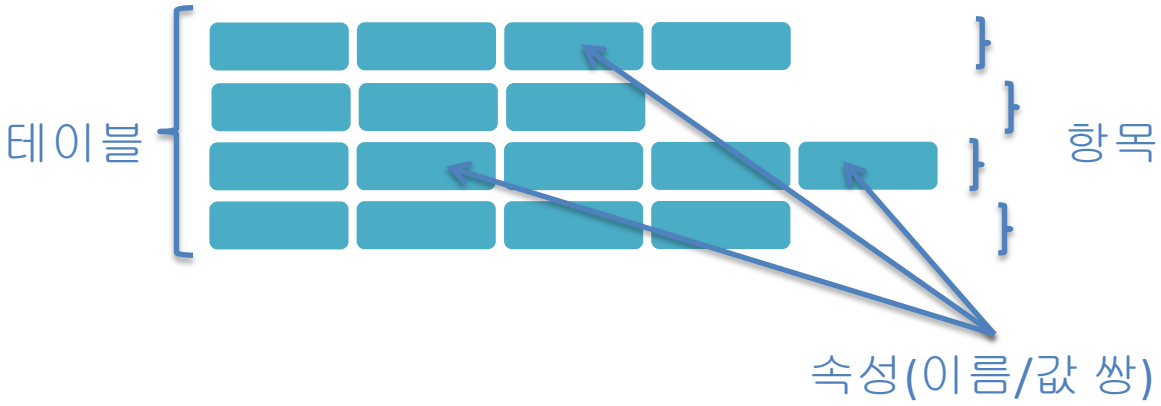

그림 1: DynamoDB 테이블 구조

테이블과 항목은 DynamoDB API를 통해 생성, 업데이트, 삭제됩니다. 관계형 데이터베이스 세계와 같은 표준 DML 언어 개념은 없습니다. DynamoDB에서의 데이터 조작은 객체 지향 코드를 통해 프로그래밍 방식으로 이루어집니다. DynamoDB 테이블에서 데이터를 쿼리할 수 있지만 이 역시 API를 통해 프로그래밍 방식으로 수행할 수 있습니다. SQL 같은 일반 쿼리 언어가 없기 때문에 DynamoDB를 가장 효과적으로 사용하려면 애플리케이션의 데이터 액세스 패턴을 이해하는 것이 중요합니다.

적합한 워크로드

DynamoDB는 NoSQL 데이터베이스이므로 비관계형 데이터가 포함된 워크로드에서 최고의 성능을 냅니다. 보다 일반적인 비관계형 워크로드 사용 사례는 다음과 같습니다.

- 광고 기술
    - 브라우저 쿠키 상태 캡처
- 모바일 애플리케이션
    - 애플리케이션 데이터 및 세션 상태 저장
- 게임 애플리케이션
    - 사용자 기본 설정 및 애플리케이션 상태 저장
    - 플레이어의 게임 상태 저장
- 소비자 "투표" 애플리케이션
    - 리얼리티 TV 경연, 슈퍼볼 광고
- 대규모 웹사이트
    - 세션 상태
    - 개인화에 사용되는 사용자 데이터
    - 액세스 제어
- 애플리케이션 모니터링
    - 애플리케이션 로그 및 이벤트 데이터 저장
    - JSON 데이터
- 사물 인터넷
    - 센서 데이터 및 로그 수집

이 모든 사용 사례는 NoSQL 데이터베이스를 강력하게 만드는 기능들의 몇 가지 조합을 통해 이득을 얻을 수 있습니다. 광고 기술 애플리케이션은 일반적으로 극히 짧은 지연 시간이 필요한데, DynamoDB의 5밀리초 이하의 읽기 및 쓰기 성능이 안성맞춤입니다. 모바일 애플리케이션과 소비자 투표 애플리케이션은 사용자가 수백만에 이르는 경우가 많고 초당 수천 건의 요청을 처리해야 합니다. DynamoDB는 수평적으로 확장되어 이런 부하를 충족할 수 있습니다. 끝으로 애플리케이션 모니터링 솔루션은 일반적으로 분당 수십만 개의 데이터 포인트를 수집하는데, DynamoDB의 스키마 없는 데이터 모델, 높은 IO 성능 및 기본 JSON 데이터 형식 지원은 이러한 유형의 애플리케이션에 매우 적합합니다.

워크로드가 DynamoDB와 같은 NoSQL 데이터베이스에 적합한지 판단할 때 고려할 또 다른 중요한 특징은 수평적 확장이 필요한지 여부입니다. 모바일 애플리케이션은 사용자가 수백만에 달할 수 있지만 각각의 애플리케이션 설치는 단일 사용자에 대한 세션 데이터를 읽고 쓰는 데 불과합니다. 이는 DynamoDB 테이블의 사용자 세션 데이터를 여러 스토리지 파티션에 분산시킬 수 있다는 뜻이 됩니다. 주어진 사용자에 대한 데이터 읽기 또는 쓰기는 단일 파티션으로 제한됩니다. 이 덕에 DynamoDB 테이블은 수평적 확장이 가능합니다. 즉, 더 많은 사용자가 추가될수록 더 많은 파티션이 생성됩니다. 이 데이터 읽기 및 쓰기 요청이 여러 파티션에 균일하게 분산되어 있으면 DynamoDB는 아주 많은 양의 동시 데이터 액세스를 처리할 수 있습니다. 이런 유형의 수평적 확장은 RDBMS에서는 "샤딩"을 사용하지 않고는 달성하기 어렵습니다. 게다가 샤딩을 사용하면 애플리케이션의 데이터 액세스 계층에 상당한 복잡성이 추가될 수 있습니다. RDBMS의 데이터가 "샤딩"되면 서로 다른 데이터베이스 인스턴스로 나누어집니다. 이로 인해 인스턴스의 인덱스와 인스턴스에 포함된 데이터 범위를 유지해야 합니다. 데이터를 읽고 쓰려면 클라이언트 애플리케이션은 어느 샤드에 읽거나 쓸 데이터 범위가 포함되어 있는지 알아야 합니다. 샤딩은 관리 부담과 비용도 추가합니다. 단일 데이터베이스 인스턴스 대신 이제 몇 개의 인스턴스를 계속 실행해야 하기 때문입니다.

워크로드가 DynamoDB에 적합한지 판단할 때는 애플리케이션의 데이터 일관성 요구 사항을 평가하는 것 역시 중요합니다. DynamoDB에서 실제로 지원하는 일관성 모델은 strong consistency와 eventual consistency의 두 가지로 전자가 후자보다 프로비저닝된 IO를 더 많이 필요로 합니다. 이런 유연성 덕에 개발자는 데이터베이스에서 가능한 최고의 성능을 뽑아내면서 애플리케이션의 일관성 요구 사항을 지원할 수 있습니다. 애플리케이션이 "strongly consistent" 읽기를 요구하지 않는 경우, 즉 한 클라이언트가 수행하는 업데이트가 다른 클라이언트에게 즉시 표시될 필요가 없는 경우, strong consistency를 강제 적용하는 RDBMS의 사용은 애플리케이션에는 순편익이 없이 성능에 부담만 초래할 수 있습니다. 그 이유는 strong consistency는 일반적으로 데이터 일부를 잠가야 하는데, 이것이 성능 병목 현상을 초래할 수 있기 때문입니다.

부적합한 워크로드

모든 워크로드가 DynamoDB 같은 NoSQL 데이터베이스에 적합한 것은 아닙니다. 이론상으로는 DynamoDB 테이블과 항목을 사용하여 고전적인 엔터티-관계 모델을 구현할 수 있지만 실제로는 작업이 대단히 까다롭습니다. 엔터티 간의 명확히 정의된 관계를 필요로 하는 트랜잭션 시스템은 여전히 기존 RDBMS를 사용하여 가장 잘 구현됩니다. 그 밖의 부적합한 워크로드는 다음과 같습니다.

- 임시 쿼리
- OLAP
- BLOB 스토리지

DynamoDB는 SQL 같은 표준 쿼리 언어를 지원하지 않고 테이블 조인이라는 개념이 없기 때문에 임시 쿼리 구성이 RDBMS에서만큼 효율적이지 않습니다. DynamoDB로 이러한 쿼리를 실행하는 것은 가능하지만 Amazon EMR과 Hive를 사용해야 합니다. 마찬가지로 OLAP 애플리케이션도 구현이 힘든데, 분석 처리에 사용되는 차원 데이터는 팩트 테이블을 차원 테이블에 조인해야 하기 때문입니다. 마지막으로 DynamoDB 항목의 크기 제한으로 인해 BLOB 저장이 실용적이지 않을 때가 많습니다. DynamoDB는 이진 데이터 형식을 지원하기는 하지만 이미지나 문서 같은 큰 이진 객체 저장에는 적합하지 않습니다. 다만 Amazon S3에 저장된 큰 BLOB에 대한 포인터를 DynamoDB 테이블에 저장하면 이 마지막 사용 사례를 쉽게 지원할 수 있습니다.

주요 개념

앞 섹션에서 설명한 것처럼 DynamoDB는 항목으로 구성된 테이블로 데이터를 조직합니다. DynamoDB 테이블의 각 항목은 임의의 속성 세트를 정의할 수 있지만 테이블의 모든 항목은 고유하게 항목을 식별하는 기본 키를 정의해야 합니다. 이 키는 "해시 키"라고 하는 속성을 포함해야 하며, 필요하다면 "범위 키"라고 하는 속성을 포함할 수 있습니다. 그림 2는 해시 키와 범위 키를 모두 정의하는 DynamoDB 테이블의 구조를 보여 줍니다.

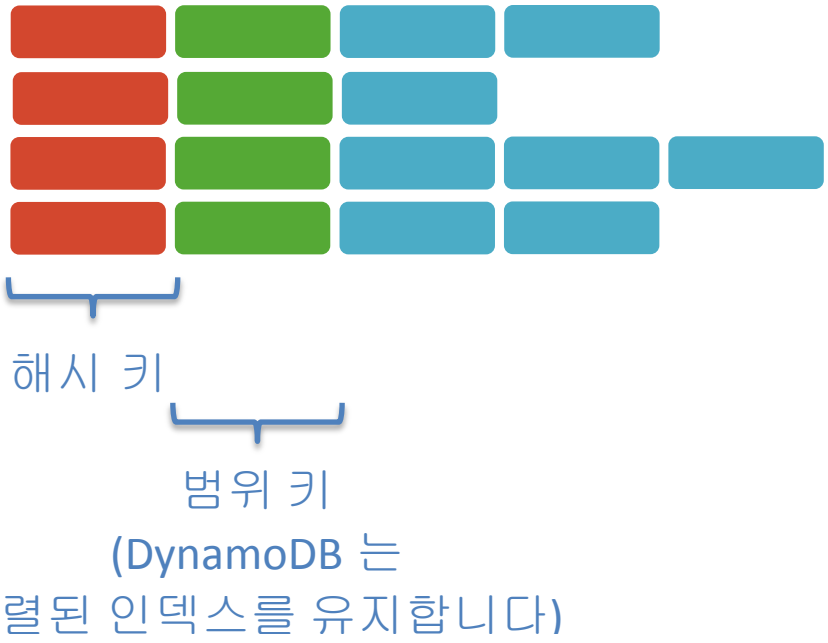

그림 2: 해시 키와 범위 키가 포함된 DynamoDB 테이블

단일한 속성 값으로 항목을 고유하게 식별할 수 있는 경우, 이 속성이 해시 키 역할을 할 수 있습니다. 그 밖의 경우, 2개의 값으로 항목을 고유하게 식별할 수 있습니다. 이 경우, 기본 키는 해시 키와 범위 키의 복합으로 정의됩니다. 그림 3은 이 개념을 보여 줍니다. 미디어 파일을 미디어 파일 트랜스코딩에 사용되는 코덱에 연결하는 RDBMS 테이블은 DynamoDB에서 해시 및 범위 키로 구성된 기본 키를 사용하여 단일 테이블로 모델링할 수 있습니다. 데이터가 DynamoDB 테이블에서 어떻게 비정규화되는지 눈여겨보십시오. 이것은 RDBMS에서 NoSQL 데이터베이스로 데이터를 마이그레이션할 때 흔한 관행이며, 이 백서 뒷부분에서 자세히 설명합니다.

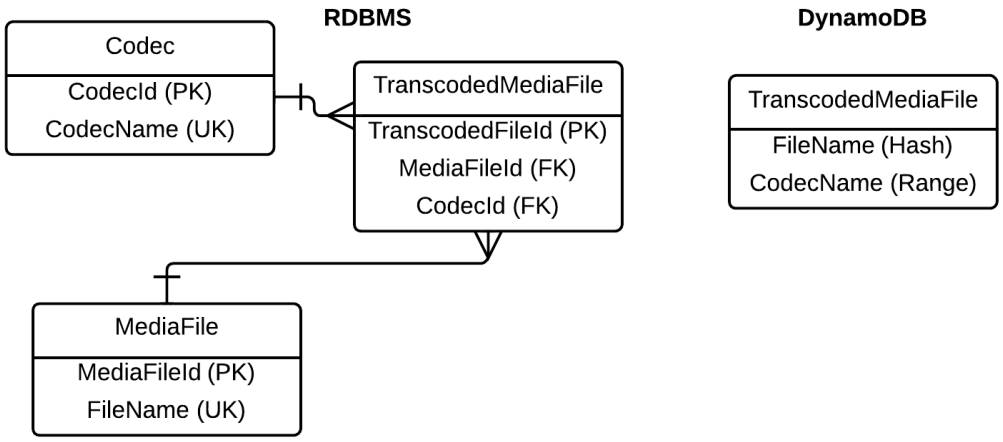

그림 3: 해시 키 및 범위 키의 예

이상적인 해시 키는 테이블의 항목들에 균일하게 분산된 수많은 개별 값을 포함합니다. 테이블의 항목들에 균일하게 분산되는 경향이 있는 속성의 좋은 예가 사용자 ID입니다. RDBMS에서 조회 값 또는 열거형으로 모델링되는 속성은 해시 키로 좋지 않습니다. 그 이유는 다른 값보다 훨씬 자주 발생하는 값이 있을 수 있기 때문입니다. 이러한 개념이 그림 4에 나와 있습니다. 사용자 ID의 개수는 균일한 반면 status-code는 그렇지 않음에 유의하십시오. DynamoDB 테이블에서 status_code를 해시 키로 사용하는 경우, 가장 자주 발생하는 값이 같은 파티션에 저장되며, 이 말은 대부분의 읽기 및 쓰기가 이 단일 파티션에 대해 이루어진다는 뜻입니다. 이를 "핫 파티션"이라고 하는데, 성능에 부정적 영향을 미칩니다.

`user-id`를 기준으로 `user_preference` 그룹의 합계로 `user_id, count(*)` 선택

user_id	합계
8a9642f7-5155-4138-bb63-870cd45d7e19	1
31667c72-86c5-4afb-82a1-a988bfe34d49	1
693f8265-b0d2-40f1-add0-bbe2e8650c08	1

```
status_code sc, log l의 합계로 status_code, count(*) 선택
여기서 l.status_code_id = sc.status_code_id
```

status_code	합계
400	125000
403	250
500	10000
505	2

그림 4: 잠재적 키 값의 균일 및 불균일 분산

기본 키를 사용하여 테이블에서 항목을 가져올 수 있습니다. 종종 해시 키 및 범위 키와는 다른 값 세트를 사용하여 항목을 가져오는 것이 유용합니다. DynamoDB는 로컬 및 글로벌 보조 인덱스를 통해 이러한 연산을 지원합니다. 로컬 보조 인덱스는 테이블에 정의된 것과 동일한 해시 키를 사용하지만 범위 키로는 다른 속성을 사용합니다. 그림 5는 로컬 보조 인덱스가 테이블에서 어떻게 정의되는지 보여 줍니다. 글로벌 보조 인덱스는 어떤 스칼라 속성도 해시 키 또는 범위 키로 사용할 수 있습니다. 보조 인덱스를 사용한 항목 가져오기는 DynamoDB API에서 정의된 쿼리 인터페이스를 사용하여 이루어집니다.

그림 5: 로컬 보조 인덱스

테이블당 존재할 수 있는 로컬 및 글로벌 보조 인덱스의 수에는 한도가 있기 때문에 영구 스토리지에 DynamoDB를 사용하는 애플리케이션의 데이터 액세스 요구 사항을 충분히 이해하는 것이 중요합니다. 또한 글로벌 보조 인덱스의 경우, 속성 값을 인덱스로 가져와야 합니다. 이것이 뜻하는 것은 인덱스가 생성될 때 상위 테이블의 속성 하위 집합을 선택해 인덱스에 포함시켜야 한다는 것입니다. 글로벌 보조 인덱스를 사용하여 항목이 쿼리되면 반환되는 항목에 채워지는 속성은 인덱스로 가져온 속성뿐입니다. 그림 6은 이 개념을 보여 줍니다. 원래의 해시 및 범위 키 속성이 글로벌 보조 인덱스에서 어떻게 승격되는지 눈여겨보십시오. 글로벌 보조 인덱스에서의 읽기는 항상 eventually consistent인 반면 로컬 보조 인덱스는 eventual consistency 또는 strong consistency를 지원합니다. 끝으로 로컬 보조 인덱스와 글로벌 보조 인덱스는 모두 인덱스에 대한 읽기 및 쓰기에 프로비저닝된 IO(아래에서 자세히 설명)를 사용합니다. 이 말은 항목이 메인 테이블에 삽입되거나 업데이트될 때마다 보조 인덱스가 인덱스 업데이트를 위해 IO를 사용한다는 뜻입니다.

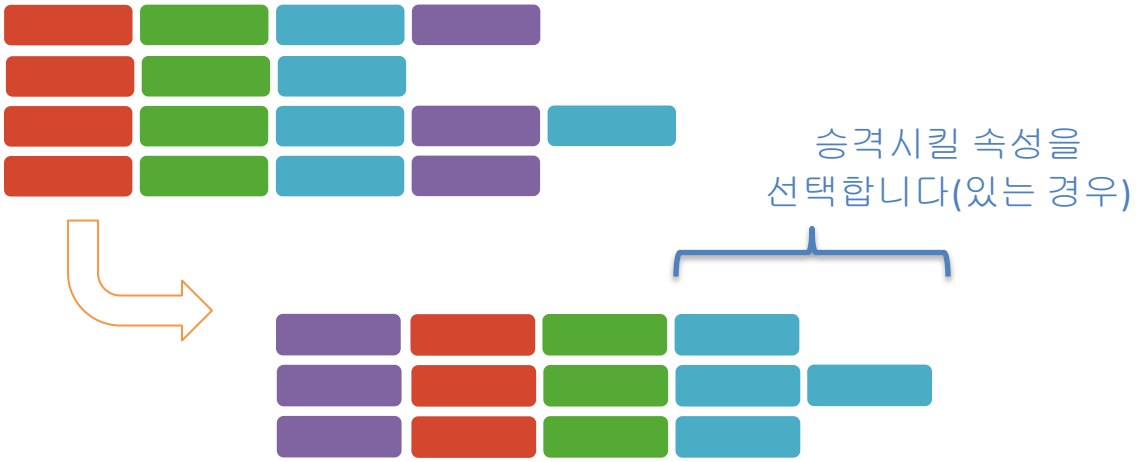

그림 6: 테이블에서 글로벌 보조 인덱스 생성

DynamoDB 테이블이나 인덱스에서 항목을 읽거나 쓸 때는 항상 읽기 또는 쓰기 작업 수행에 필요한 데이터 양이 "읽기 단위" 또는 "쓰기 단위"로 표현됩니다. 읽기 단위는 4K의 데이터로, 쓰기 단위는 1K의 데이터로 구성됩니다. 즉, 크기가 8K인 항목을 가져오려면 데이터 2 읽기 단위가 소비됩니다. 항목 삽입에는 데이터 8 쓰기 단위가 소비됩니다. 초당 허용되는 읽기 및 쓰기 단위 수를 테이블의 "프로비저닝된 IO"라고 합니다.
애플리케이션에서 초당 1,000개의 4K 항목을 써야 하는 경우, 테이블의 프로비저닝된 쓰기 용량은 초당 4,000 쓰기 단위 이상이어야 합니다. 테이블에 프로비저닝된 읽기 또는 쓰기 용량이 부족하면 DynamoDB 서비스는 읽기 및 쓰기 작업을 "제한(throttle)"합니다. 이는 성능 저하를 초래할 수 있고, 경우에 따라서는 클라이언트 애플리케이션에서 제한 예외를 초래할 수 있습니다. 이런 이유로 테이블을 설계할 때는 애플리케이션의 IO 요구 사항을 이해하는 것이 중요합니다. 다만 읽기 용량과 쓰기 용량은 기존 테이블에서 수정할 수 있으며, 애플리케이션이 갑자기 제한을 초래하는 사용 급증을 경험하는 경우, 프로비저닝된 IO를 늘려 새 워크로드를 처리할 수 있습니다. 같은 원리로 어떤 이유로든 부하가 감소하면 프로비저닝된 IO를 줄일 수 있습니다. 테이블의 IO 특성을 동적으로 수정하는 이 기능은 DynamoDB와 기존 RDBMS의 중요한 차이점입니다. RDBMS에서는 데이터베이스 엔진이 실행되는 기본 하드웨어를 기준으로 IO 처리량이 고정됩니다.

RDBMS에서 DynamoDB로 마이그레이션

앞 섹션에서 DynamoDB의 주요 기능 일부와 DynamoDB와 기존 RDBMS의 주요 차이점 일부를 살펴봤습니다. 이 섹션에서는 이러한 주요 기능과 차이점을 고려한, RDBMS에서 DynamoDB로의 마이그레이션 전략을 제안합니다. 데이터베이스 마이그레이션은 복잡하고 위험할 수 있으므로 단계적이고 반복적인 접근 방법을 따르는 것이 좋습니다. 새로운 기술 채택에서는 늘 그렇듯이 가장 쉬운 사용 사례에 먼저 집중하는 것이 좋습니다. 또한 이 절에서 제안하는 것처럼 DynamoDB로의 마이그레이션이 "모 아니면 도"식의 프로세스일 필요는 없다는 점도 명심해야 합니다. 일부 마이그레이션의 경우, DynamoDB와 RDBMS 모두에서 병렬로 워크로드를 실행하고 마이그레이션이 성공했음이 분명하고 애플리케이션이 제대로 작동하는 경우에만 DynamoDB로 마이그레이션하는 것이 합리적일 수 있습니다.

다음 상태 다이어그램은 우리가 제안하는 마이그레이션 전략을 표현한 것입니다.

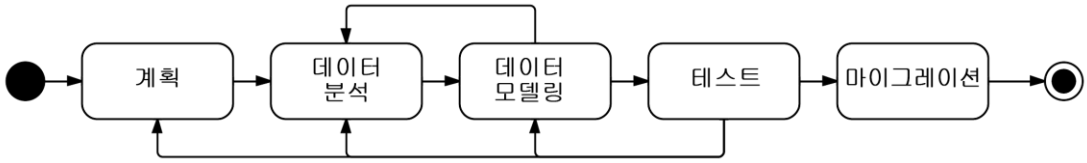

그림 7: 마이그레이션 단계

이 프로세스는 반복적이라는 점을 명심하십시오. 일부 상태의 결과는 이전 상태로의 복귀로 이어질 수 있습니다. 데이터 분석 및 데이터 모델링 단계에서의 실수가 테스트할 때까지 분명히 드러나지 않을 수 있습니다. 대부분의 경우, 최종 데이터 마이그레이션 상태에 도달하기 전에 이 단계들을 여러 번 반복해야 합니다. 각 단계에 대해서는 다음 섹션에서 자세히 설명합니다.

계획 단계

계획 단계의 첫 부분은 데이터 마이그레이션의 목적을 파악하는 것입니다. 목적에는 흔히 다음이 포함됩니다(하지만 이에 국한되지 않음).

- 애플리케이션 성능 향상
- 비용 절감
- RDBMS의 부하 감소

많은 경우, 마이그레이션의 목적은 위의 모두를 합한 것일 수 있습니다. 이러한 목적이 정의되고 나면 마이그레이션할 RDMBS 테이블의 ID를 DynamoDB에 알리는 데 사용할 수 있습니다. 앞서 언급한 것처럼 비관계형 데이터가 포함된 워크로드가 사용 중인 테이블은 DynamoDB로 마이그레이션하기에 아주 좋습니다. 이러한 테이블을 DynamoDB로 마이그레이션하면 애플리케이션 성능을 대폭 높이고 비용과 RDBMS의 부하를 줄일 수 있습니다. 마이그레이션에 적합한 테이블은 다음과 같습니다.

- 엔터티-속성-값 테이블
- 애플리케이션 세션 상태 테이블
- 사용자 기본 설정 테이블
- 로깅 테이블

테이블이 식별된 후에는 마이그레이션에 문제를 일으킬 수 있는 원본 테이블의 특징을 기록해야 합니다. 이 정보는 탄탄한 마이그레이션 전략을 선택하는 데 중요합니다. 마이그레이션 전략에 영향을 줄 수 있는 보다 일반적인 문제 일부를 살펴보겠습니다.

문제	마이그레이션 전략에 미치는 영향
마이그레이션 전 또는 도중에는 RDBMS 원본 테이블에 대한 쓰기를 묵인할 수 없습니다.	대상 DynamoDB 테이블의 데이터와 원본의 동기화는 어렵습니다. 원본과 대상 모두에 병렬로 데이터를 쓰는 마이그레이션 전략을 고려해 보십시오.
원본 테이블의 데이터 양이 기존 네트워크 대역폭으로 합리적으로 전송할 수 있는 양을 초과합니다.	원본 테이블에서 이동식 디스크로 데이터를 내보내고 AWS Import/Export 서비스를 사용하여 이 데이터를 S3의 버킷으로 가져오는 방법을 고려해 보십시오. S3에서 직접 DynamoDB로 이 데이터를 가져옵니다. 또는 최근 시점 이후에 생성된 기록만을 내보내 마이그레이션해야 하는 데이터 양을 줄이십시오. 이 시점보다 오래된 모든 데이터는 RDBMS의 레거시 테이블에 남게 됩니다.

문제	마이그레이션 전략에 미치는 영향
원본 테이블의 데이터를 DynamoDB로 가져오기 전에 변환해야 합니다.	원본 테이블에서 데이터를 내보내 S3로 전송합니다. 데이터 변환을 수행하는 데는 EMR 사용을 고려하고, 변환된 데이터를 DynamoDB로 가져옵니다.
원본 테이블의 기본 키 구조는 DynamoDB로 내보낼 수 없습니다.	가져온 항목의 해시 키 및 범위 키로 적합한 열을 식별하십시오. 또는 적절한 해시 키 역할을 할 대리 키(예: UUID)를 원본 테이블에 추가하는 것도 고려해 보십시오.
원본 테이블의 데이터는 암호화되어 있습니다.	암호화를 RDBMS가 관리하는 경우, 내보낼 때 데이터를 복호화하고, 가져올 때 기본 데이터베이스 엔진이 아니라 애플리케이션이 적용하는 암호화 스키마를 사용하여 다시 암호화해야 합니다. 암호화 키는 DynamoDB 외부에서 관리해야 합니다.

표 1: 마이그레이션 전략에 영향을 미치는 문제

마지막으로 어쩌면 가장 중요한 것은 계획 단계에서 백업 및 복구 프로세스를 정의하고 문서화해야 한다는 점입니다. 마이그레이션 전략이 RDBMS에서 DynamoDB로의 완전 전환을 필요로 하는 경우, 마이그레이션 실패 시 RDBMS를 사용하여 기능을 복구하는 프로세스를 반드시 정의해야 합니다. 위험을 줄이려면 일정 기간 동안 DynamoDB와 RDBMS에서 병렬로 워크로드를 실행하는 방법을 고려해 보십시오. 이 시나리오에서는 DynamoDB를 사용하는 프로덕션에서 워크로드가 충분히 테스트된 후에만 레거시 RDBMS 기반 애플리케이션을 비활성화할 수 있습니다.

데이터 분석 단계

데이터 분석 단계의 목적은 원본 데이터의 구성을 이해하고 애플리케이션이 사용하는 데이터 액세스 패턴을 파악하는 것입니다. 이 정보는 데이터 모델링 단계에 필요합니다. DynamoDB에서의 워크로드 실행 비용과 성능을 이해하는 것 역시 중요합니다. 원본 데이터 분석에는 다음이 포함되어야 합니다.

- DynamoDB로 가져올 항목 수의 추정
- 항목 크기의 분포
- 해시 키 또는 범위 키로 사용할 값의 다중성

DynamoDB 요금 체계에 포함되는 두 가지 주요 구성 요소는 스토리지와 프로비저닝된 IO입니다. DynamoDB 테이블로 가져올 항목의 수와 각 항목의 대략적 크기를 추정하면 테이블의 스토리지 및 프로비저닝된 IO 요구 사항을 계산할 수 있습니다. 일반적인 SQL 데이터 형식은 DynamoDB에서 문자열, 숫자, 이진 등 3가지 스칼라 형식 중 하나에 매핑됩니다. 숫자 데이터 형식의 길이는 가변적이며, 문자열은 이진 UTF-8을 사용하여 인코딩됩니다. 항목 크기를 추정할 때는 가장 큰 값을 가진 속성에 집중해야 합니다. 프로비저닝된 IOPS는 정수 1K 단위로 증가하기 때문입니다. DynamoDB에는 소수부 IO라는 개념이 없습니다. 항목 크기가 3.3K로 추정되는 경우, 단일 항목을 쓰고 읽으려면 각각 4개의 1K 쓰기 IO 단위와 1개의 4K 읽기 IO 단위가 필요합니다. 크기는 가장 가까운 킬로바이트로 올림하기 때문에 숫자 형식의 정확한 크기는 중요하지 않습니다. 대부분의 경우, 정밀도가 높은 큰 수라 하더라도 데이터는 작은 바이트를 사용하여 저장됩니다. 테이블의 각 항목에 포함된 속성 수는 가변적일 수 있으므로 항목 크기의 분포를 계산하고 백분위 값을 사용하여 항목 크기를 추정하는 것이 좋습니다. 예를 들어 95번째 백분위수에 해당하는 항목 크기를 선택하여 스토리지 및 프로비저닝된 IO 비용 추정에 사용할 수 있습니다. 원본 테이블에 개별적으로 검사해야 할 행이 너무 많을 경우, 원본 데이터에서 추출한 샘플을 사용해 항목 크기 분포를 계산합니다.

최소한 초당 단일 항목을 읽고 쓸 수 있는 프로비저닝된 읽기 및 쓰기 단위가 테이블에 충분해야 합니다. 예를 들어 크기가 95번째 백분위수 이하인 항목을 쓰는 데 4 쓰기 단위가 필요한 경우, 테이블에는 초당 4 쓰기 단위 이상의 프로비저닝된 IO가 있어야 합니다. 프로비저닝된 IO가 이보다 작으면 단일 항목 쓰기가 제한을 초래하고 성능이 저하됩니다. 실제로는 프로비저닝된 읽기 및 쓰기 단위 수가 필요한 최소값보다 훨씬 큽니다. 데이터 스토리지에 DynamoDB를 사용하는 애플리케이션은 일반적으로 읽기와 쓰기를 동시에 실행해야 합니다.

프로비저닝된 IO의 올바른 추정은 필요한 애플리케이션 성능을 보장하고 비용을 이해하기 위한 관건입니다. 해시 키 또는 범위 키가 될 수 있는 RDBMS 열 값의 분포 빈도를 이해하는 것 역시 최대 성능을 얻는 데 중요합니다. 균일하게 분포되지 않은(즉, 일부 값이 다른 값보다 훨씬 많이 발생) 값이 포함된 열은 좋은 해시 키 또는 범위 키가 아닙니다. 발생 빈도가 높은 키를 가진 항목에 대한 액세스는 같은 DynamoDB 파티션에 대해 이루어지고 이는 성능에 나쁜 영향을 미치기 때문입니다.

데이터 분석 단계의 두 번째 목적은 애플리케이션의 데이터 액세스 패턴을 분류하는 것입니다. DynamoDB는 SQL과 같은 일반 쿼리 언어를 지원하지 않으므로 테이블에서 데이터가 읽고 쓰여지는 방식을 문서화하는 것이 중요합니다. 이 정보는 테이블, 키 구조, 인덱스가 정의되는 데이터 모델링 단계에 매우 중요합니다. 몇 가지 일반적인 데이터 액세스 패턴은 다음과 같습니다.

- 쓰기 전용 – 항목이 테이블에 쓰여지고 애플리케이션은 이 항목을 읽지 않습니다.
- 개별 값 기준 가져오기 – 테이블에서 항목을 고유하게 식별하는 값을 기준으로 개별적으로 항목을 가져옵니다.
- 값 범위에 걸친 쿼리 – 이것은 시간 데이터에서 자주 볼 수 있습니다.

다음 섹션에서 살펴보겠지만 위에 설명한 범주를 사용한 애플리케이션의 데이터 액세스 패턴 설명서는 많은 데이터 모델링 결정 시 중심이 됩니다.

데이터 모델링 단계

이 단계에서는 테이블, 해시 및 범위 키, 보조 인덱스가 정의됩니다. 이 단계에서 만들어지는 데이터 모델은 데이터 분석 단계에서 설명한 데이터 액세스 패턴을 지원해야 합니다. 데이터 모델링의 첫 단계는 테이블의 해시 키 및 범위 키를 결정하는 것입니다. 해시 키만으로 구성되건 해시 키와 범위 키의 복합으로 구성되건 기본 키는 테이블의 모든 항목에 대해 고유해야 합니다. RDBMS에서 데이터를 마이그레이션할 때 원본 테이블의 기본 키를 해시 키로 사용하고 싶은 마음이 생깁니다. 하지만 실제로 이 키는 애플리케이션에는 의미상 무의미할 때가 많습니다. 예를 들어 RDBMS의 User 테이블은 숫자로 된 기본 키를 정의할 수 있지만 사용자 로그인을 담당하는 애플리케이션은 숫자로 된 사용자 ID가 아니라 이메일 주소를 묻습니다. 이 경우에는 이메일 주소가 "자연 키"이며, 해시 키 값을 기준으로 항목을 쉽게 가져올 수 있기 때문에 DynamoDB에서 해시 키로 더 적합합니다. 개별 값을 기준으로 항목을 가져오는 데이터 액세스 패턴에는 이런 해시 키 모델링 방식이 적합합니다. "쓰기 전용" 같은 다른 데이터 액세스 패턴의 경우, 무작위 생성되는 숫자 ID를 사용하는 것이 해시 키에 적합합니다. 이 경우, 애플리케이션이 테이블에서 항목을 가져오지 않기 때문에 키는 데이터 가져오기의 수단이 아니라 오로지 항목을 고유하게 식별하기 위해서만 사용됩니다.

2개의 키 값에서 고유한 인덱스를 포함하고 있는 RDBMS 테이블은 해시 키와 범위 키를 모두 사용하여 기본 키를 정의하기에 좋습니다. RDBMS에서 다대다 관계를 정의하는 데 사용되는 교차 테이블은 일반적으로 관계 양쪽의 키 값에서 고유한 인덱스를 사용하여 모델링됩니다. 다대다 관계의 데이터를 가져오려면 일련의 테이블 조인이 필요하므로 이러한 테이블을 DynamoDB로 마이그레이션하기 위해서는 데이터 비정규화도 필요합니다(아래에서 자세히 설명). 데이터 값은 종종 범위 키로 사용되기도 합니다. 임의의 날짜에 URL을 방문한 횟수를 계산하는 테이블은 URL을 해시 키로, 날짜를 범위 키로 정의할 수 있습니다. 해시 키만으로 구성된 기본 키의 경우, 복합 기본 키가 포함된 항목을 가져오려면 애플리케이션이 해시 키 값과 범위 키 값을 모두 지정해야 합니다. 해시 키 및/또는 범위 키로 대리 키 또는 자연 키가 더 나은 선택인지 여부를 평가할 때는 이 점을 고려해야 합니다.

키가 아닌 속성을 임의로 항목에 추가할 수 있으므로 DynamoDB 테이블 정의에서 지정해야 하는 속성은 해시 키와 (필요한 경우) 범위 키뿐입니다. 하지만 키가 아닌 속성에서 보조 인덱스를 정의하려는 경우, 테이블 정의에 키가 아닌 속성을 포함시켜야 합니다. 키가 아닌 속성을 테이블 정의에 포함시켜도 테이블의 모든 항목에 어떤 스키마도 적용되지 않습니다. 테이블의 각 항목은 기본 키 외에 임의의 속성 목록을 가질 수 있습니다.

RDBMS에서의 SQL 지원은 테이블의 어떤 열 값이든 사용하여 레코드를 가져올 수 있음을 뜻합니다. 이러한 쿼리가 늘 효율적이지 않을 수도 있습니다. 데이터를 가져오는 데 사용되는 열에 인덱스가 존재하지 않으면 일치하는 행을 찾기 위해 전체 테이블 스캔이 필요할 수 있습니다. DynamoDB API에 의해 노출되는 쿼리 인터페이스는 이런 방식의 테이블에서 항목 가져오기를 지원하지 않습니다. 전체 테이블 스캔을 할 수도 있지만 비효율적이며, 테이블이 큰 경우에는 상당한 읽기 단위를 소비합니다. 그 대신 테이블의 기본 키 또는 테이블에서 정의된 로컬 또는 글로벌 보조 인덱스의 키를 사용하여 DynamoDB 테이블에서 항목을 가져올 수 있습니다. RDBMS 테이블의 키가 아닌 열의 인덱스는 애플리케이션이 일반적으로 이 값에서 데이터를 쿼리함을 의미하므로 이러한 속성은 DynamoDB 테이블의 로컬 또는 글로벌 보조 인덱스로 좋습니다. DynamoDB 테이블에서 허용되는 보조 인덱스의 수에는 한도가 있으므로[2] 애플리케이션이 데이터 가져오기에 가장 자주 사용할 속성 값을 사용하여 이러한 인덱스에 대해 정의된 키를 선택하는 것이 중요합니다.

DynamoDB는 테이블 조인 개념을 지원하지 않으므로 RDBMS 테이블에서 데이터를 마이그레이션하려면 종종 데이터 비정규화가 필요합니다. RDBMS 사용에 익숙한 사용자에게는 이것이 낯설고 어쩌면 불편한 개념일 수 있습니다. RDBMS에서 DynamoDB로 마이그레이션하기에 가장 적합한 워크로드에는 비관계형 데이터가 포함되는 경향이 있으므로 비정규화가 관계형 데이터 모델에서와 같은 문제를 일으키는 경우는 드뭅니다. 예를 들어 관계형 데이터베이스에 UserID를 통해 연결되는 User 및 UserAddress 테이블이 포함된 경우, User 속성과 UserAddress 속성을 하나의 DynamoDB 테이블로 결합할 수 있습니다. 관계형 데이터베이스에서 UserAddress 정보를 정규화하면 주어진 사용자에 대해 여러 주소를 지정할 수 있습니다. 이것은 연락처 관리 또는 CRM 시스템의 요구 사항입니다. 하지만 DynamoDB에서는 User 테이블이 다른 용도에 사용될 가능성이 높습니다(아마도 모바일 애플리케이션의 등록 사용자 추적). 이 사용 사례에서는 Addresses에 대한 Users의 다중성보다는 확장성 및 사용자 레코드의 빠른 검색이 더 중요합니다.

[2] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html

데이터 모델링 예제

이 섹션과 앞 섹션에서 설명한 개념들을 결합하는 예제를 살펴보겠습니다. 이 예제는 효율적인 데이터 액세스를 위해 보조 인덱스를 사용하는 방법과 DynamoDB 테이블에서 항목 크기와 필요한 프로비저닝된 IO 양을 추정하는 방법을 보여 줍니다. 그림 8에는 전자 상거래 포털을 통한 온라인 주문 처리 시 이벤트 추적에 사용되는 스키마의 ER 다이어그램이 포함되어 있습니다. RDBMS 테이블 구조와 DynamoDB 테이블 구조가 모두 나와 있습니다.

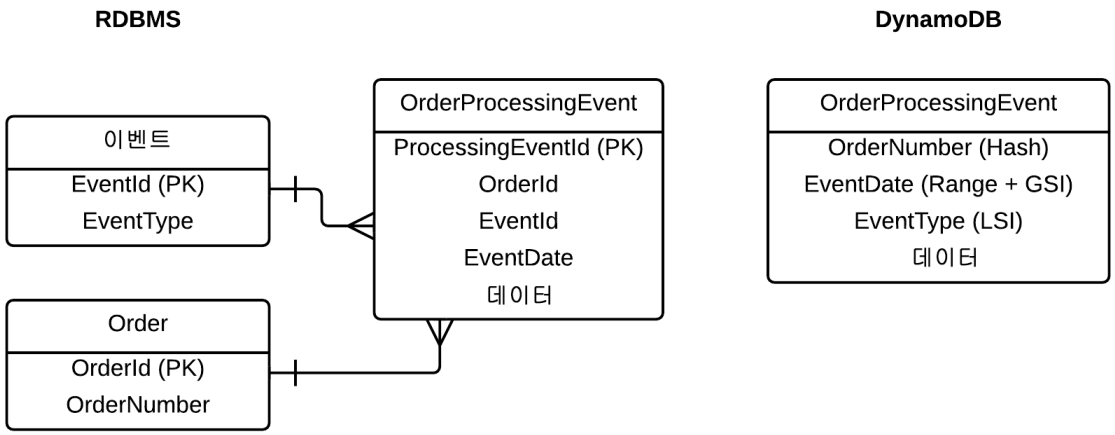

그림 8: 이벤트 추적을 위한 RDBMS 스키마와 DynamoDB 스키마

마이그레이션할 행의 수는 약 10^8이므로 항목 크기의 95번째 백분위수를 반복 계산하는 것은 실용적이지 않습니다. 그 대신 10^6 행을 대체하여 단순 무작위 샘플링을 수행합니다. 이렇게 하면 항목 크기 추정 목적에 적합한 정밀도를 얻게 됩니다. 이를 위해 DynamoDB 테이블에 삽입될 필드가 포함된 SQL 뷰를 생성합니다. 그러면 샘플링 루틴이 이 뷰에서 무작위로 10^6행을 선택하고 95번째 백분위수에 해당하는 크기를 계산합니다.

이 통계 샘플링에서 6.6KB라는 95번째 백분위수 크기가 나오며, 그 대부분은 "Data" 속성(최대 크기가 6KB)이 사용합니다. 단일 항목을 쓰는 데 필요한 최소 쓰기 단위 수는 다음과 같습니다.

$$ceiling(6.6KB\ per\ item/1KB\ per\ write\ unit) = 7\ write\ units\ per\ item$$

단일 항목을 읽는 데 필요한 최소 읽기 단위 수도 비슷하게 계산됩니다.

$$ceiling(6.6KB\ per\ item/4Kb\ per\ read\ unit) = 2\ read\ units\ per\ item$$

이 특정 워크로드는 쓰기 작업이 많으며, 하루 500건의 주문에 대한 1,000개의 이벤트를 쓰기에 충분한 IO가 필요합니다. 이것은 다음과 같이 계산됩니다.

$$500\ orders\ per\ day\ \times\ 1000\ events\ per\ order\ =\ 5\ \times 10^5\ events\ per\ day$$

$$5\ \times\ 10^5 events\ per\ day\ \times\ 86400\ seconds\ per\ day\ = 5.78\ events\ per\ second$$

$$ceiling(5.78\ events\ per\ second\ \times\ 7\ write\ units\ per\ item) = 41\ write\ units\ per\ second$$

테이블에서의 읽기는 시간당 한 번만 이루어지며, 이때 전 시간의 데이터를 ETL을 위해 Amazon Elastic Map Reduce 클러스터로 가져옵니다. 이 연산은 주어진 데이터 범위에서 항목을 선택하는 쿼리를 사용합니다(이것이 EventDate 속성이 범위 키이면서 동시에 글로벌 보조 인덱스인 이유입니다). 쿼리 결과를 검색하는 데 필요한 읽기 단위 수(글로벌 보조 인덱스에서 프로비저닝될)는 쿼리가 반환하는 결과에 따라 다릅니다.

$$5.78\ events\ per\ second\ \times\ 3600\ seconds\ per\ hour\ = 20808\ events\ per\ hour$$

$$\frac{20808\ events\ per\ hour\ \times\ 6.6KB\ per\ item}{1024KB}\ = 134.11MB\ per\ hour$$

단일 쿼리 연산에서 반환되는 데이터의 양은 최대 1MB이므로 페이지 매김이 필요합니다. 매시간 읽기 쿼리는 135페이지의 데이터 읽기가 필요합니다. strongly consistent 읽기의 경우, 한 번에 전체 페이지를 읽으려면 256 읽기 단위가 필요합니다(이 숫자는 eventually consistent 읽기의 절반입니다). 따라서 이 특정 워크로드를 지원하려면 256 읽기 단위와 41 쓰기 단위가 필요합니다. 실용적 관점에서 쓰기 단위는 48처럼 짝수로 표현될 가능성이 높습니다. 이제 이 워크로드의 DynamoDB 비용을 추정하는 데 필요한 모든 데이터가 갖춰졌습니다.

1. 항목 수(10^8)
2. 항목 크기(7KB)
3. 쓰기 단위(48)
4. 읽기 단위(256)

이 데이터를 Amazon 월 사용량 계산기를 통해 실행하여[3] 비용 추정치를 도출할 수 있습니다.

[3] http://calculator.s3.amazonaws.com/index.html

테스트 단계

테스트 단계는 마이그레이션 전략에서 가장 중요한 부분입니다. 테스트 단계에서 전체 마이그레이션 프로세스의 종단 간 테스트가 이루어집니다. 종합적 테스트 계획에는 최소한 다음이 포함되어야 합니다.

테스트 범주	목적
기본 승인 테스트	이러한 테스트는 데이터 마이그레이션 루틴이 완료되면 자동으로 실행되어야 합니다. 이 테스트의 주 목적은 데이터 마이그레이션의 성공 여부를 확인하는 것입니다. 이 테스트의 일반적 출력에는 다음이 포함됩니다. • 처리된 총 항목 수 • 가져온 총 항목 수 • 건너뛴 총 항목 수 • 총 경고 수 • 총 오류 수 테스트 결과 보고된 이러한 총 수치 중 하나라도 예상 값을 벗어나는 경우, 이는 마이그레이션이 성공하지 못했음을 나타내며, 프로세스의 다음 단계 또는 다음 테스트로 넘어가기 전에 이 문제를 해결해야 합니다.
기능 테스트	이 테스트는 데이터 스토리지로 DynamoDB를 사용하여 애플리케이션의 기능을 시험적으로 실행해 봅니다. 이 테스트에는 자동 테스트와 수동 테스트가 결합되어 있습니다. 기능 테스트의 주 목적은 RDBMS 데이터를 DynamoDB로 마이그레이션함으로써 발생하는 애플리케이션 문제를 식별하는 데 있습니다. 이 테스트 중에 데이터 모델의 허점이 종종 드러납니다.
비기능 테스트	이 테스트는 다양한 부하 수준에서의 성능과 애플리케이션 스택 모든 부분의 장애 복원력 등 애플리케이션의 비기능적 특징을 평가합니다. 이러한 테스트에는 가능성은 낮지만 애플리케이션에 악영향을 미칠 수 있는 경계 케이스 또는 엣지 케이스(예컨대 많은 클라이언트가 정확히 같은 시간에 동일한 레코드 업데이트를 시도하는 경우)도 포함될 수 있습니다. 계획 단계에서 정의한 백업 및 복구 프로세스도 비기능 테스트에 포함되어야 합니다.

테스트 범주	목적
사용자 승인 테스트	이 테스트는 최종 데이터 마이그레이션이 완료됐을 때 애플리케이션의 최종 사용자가 실행해야 합니다. 이 테스트의 목적은 애플리케이션이 조직에서의 주된 기능을 충족할 만큼 충분히 사용성이 있는지를 최종 사용자가 확인하는 데 있습니다.

표 2: 데이터 마이그레이션 테스트 계획

마이그레이션 전략은 반복적이므로 이 테스트들은 무수히 실행됩니다. 최대의 효율을 위해 마이그레이션할 데이터 총량이 큰 경우, 프로덕션 데이터의 샘플링을 사용하여 데이터 마이그레이션 루틴을 테스트하는 방법을 고려해 보십시오. 테스트 단계에서 나온 결과로 인해 프로세스의 이전 단계를 수정해야 하는 경우가 많습니다. 전체 마이그레이션 전략은 프로세스에서 반복할 때마다 점점 더 다듬어지며, 모든 테스트 실행에 성공했다면 다음 최종 단계인 데이터 마이그레이션을 실행할 때가 된 것입니다.

데이터 마이그레이션 단계

데이터 마이그레이션 단계에서는 원본 RDBMS 테이블의 프로덕션 데이터 전체를 DynamoDB로 마이그레이션합니다. 이 단계에 도달할 때쯤에는 종단 간 마이그레이션 프로세스가 테스트되고 철저히 검사된 상태일 것입니다. 프로세스의 모든 단계를 꼼꼼히 문서화해야 프로덕션 데이터 세트에서의 실행이 전에도 수없이 실행했던 절차를 따르는 것처럼 간단해집니다. 이 최종 단계에 대비하여 애플리케이션 사용자들에게 애플리케이션 유지 관리 및 (필요한 경우) 가동 중단이 있음을 통지하는 알림을 전송해야 합니다.

데이터 마이그레이션이 완료되면 이전 단계에서 정의한 사용자 수용 테스트를 마지막으로 한 번 실행하여 애플리케이션이 사용 가능한 상태인지 확인합니다. 어떤 이유로든 마이그레이션이 실패하는 경우, (이 시점에서 역시 철저히 테스트되고 검사되었을) 백업 및 복구 절차를 실행할 수 있습니다. 시스템이 다시 안정 상태가 되면 장애의 근본 원인 분석을 실시하고 근본 원인이 해결되는 대로 데이터 마이그레이션 일정을 다시 잡아야 합니다. 아무 문제가 없다면 애플리케이션의 정상 작동을 시사하는 충분한 데이터가 쌓일 때까지 다음 며칠 동안 애플리케이션을 면밀히 모니터링해야 합니다.

결론

적절한 워크로드에 DynamoDB를 활용하면 기존 RDBMS에 비해 비용과 운영 부담은 줄이고 성능, 가용성, 안정성은 높일 수 있습니다. 이 백서에서는 RDBMS에서 DynamoDB로 마이그레이션하기에 적합한 워크로드를 파악하고 마이그레이션하는 전략을 제안했습니다. 이러한 전략을 구현하려면 신중한 계획과 엔지니어링 노력이 필요하지만 DynamoDB와 같은 완전 관리형 NoSQL 솔루션으로 마이그레이션할 경우의 ROI는 노력에 수반되는 선행 비용을 크게 능가할 것이라고 확신합니다.

치트 시트

다음은 이 백서에서 논의한 주요 개념 일부와 이러한 개념이 자세히 설명된 섹션을 요약한 "치트 시트"입니다.

개념	섹션
적합한 워크로드 결정	*적합한 워크로드*
적합한 키 구조 선택	*주요 개념*
테이블 인덱싱	*데이터 모델링 단계*
읽기 및 쓰기 처리량 프로비저닝	*데이터 모델링 예제*
마이그레이션 전략 선택	*계획 단계*

참고 문헌

자세한 내용은 다음을 참조하십시오.

- DynamoDB 개발자 안내서[4]
- DynamoDB 웹사이트[5]

[4] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GettingStartedDynamoDB.html

[5] http://aws.amazon.com/dynamodb