Amazon Neptune 데이터베이스를 실행하는 ISVs 대한 다중 테넌시 지침

AWS 권장 가이드

AWS 권장 가이드: Amazon Neptune 데이터베이스를 실행하는 ISVs 대한 다중 테넌시 지침

Table of Contents

Amazon Neptune 데이터베이스를 실행하는 ISVs 대한 다중 테넌시 지침

Amazon Web Services(기여자)

2024년 8월(문서 기록)

다중 테넌시는 애플리케이션의 단일 인스턴스가 여러 고객에게 서비스를 제공하는 컴퓨터 시스템 아키텍처입니다. 각 고객을 테넌트라고 합니다. 다중 테넌트 아키텍처에서 애플리케이션의 이러한 인스턴스는 각 테넌트가 동일한 인프라에 물리적으로 공동 배치되지만 논리적으로 분리된 공유 환경에서 작동합니다.

독립 소프트웨어 공급업체(ISV)는 Amazon Neptune을 사용하여 고도로 연결된 데이터를 탐색해야 하는 애플리케이션을 구동할 수 있습니다. 계정에서 클라우드 기반 서비스형 소프트웨어(SaaS) 애플리케이션을 관리하고 테넌트에게 구독을 제공할 수 있습니다. 그러면 테넌트는 인터넷을 통해 또는를 통해 비공개로 서비스에 액세스할 수 있습니다 AWS PrivateLink. 이 모델의 경제성은 테넌트가 구매, 구축 및 유지 관리하는 것보다 저렴한 소프트웨어에 액세스할 수 있기 때문에 양 당사자 모두에게 효과적입니다. ISV는 소프트웨어를 생성하고 유지 관리하는 데 드는 비용보다 구독에 더 많은 비용을 청구할 수 있습니다. 문제는 비즈니스를 여러 테넌트로 확장하는 방법입니다.

다중 테넌시는 ISVs 중요한 경제적 및 운영상의 이점을 제공합니다. 멀티 테넌트 아키텍처는 조직에 더 나은 투자 수익률(ROI)을 제공합니다. 또한 다중 테넌시는 운영 요구 사항을 간소화하여 조직이 더 빠르게 이동하고 테넌트에게 소프트웨어를 제공하는 비용을 줄일 수 있도록 합니다.

이 문서는 Amazon Neptune을 사용하여 다중 테넌트 ISV 애플리케이션을 효과적으로 실행하는 방법에 대한 지침을 제공합니다. 이 지침은 ISVs가 SaaS 솔루션을 고객에게 성공적으로 제공할 수 있도록 지원하여 다년간 얻은 모범 사례를 기반으로 합니다. 조직의 목표 및 아키텍처 원칙의 맥락에서이 지침을 평가하면 솔루션을 최적화하는 방법을 찾는 데 도움이 됩니다.

> **Note**
>
> 이 문서는 모범 사례의 전체 목록을 제공하지 않습니다. 다중 테넌시 ISV 워크로드에 대한 구체적인 추가 지침을 제공하여 Amazon Neptune에 AWS Well-Architected Framework 적용 문서를 보완합니다. 솔루션을 설계할 때 두 문서의 고려 사항을 검토하는 것이 좋습니다.

SaaS 데이터 파티셔닝 모델

SaaS 개발자의 과제 중 하나는 다중 테넌트 환경에서 데이터를 표현하고 구성하기 위한 아키텍처 패턴을 설계하는 것입니다. 이러한 다중 테넌트 스토리지 메커니즘 및 패턴을 일반적으로 데이터 파티셔닝 이라고 합니다.

다중 테넌트 SaaS 환경에서는 데이터 파티셔닝과 테넌트 격리를 구분하는 것이 중요합니다. 이러한 개념은 관련이 있지만 동의어는 아닙니다. 데이터 파티셔닝은 각 테넌트에 대한 데이터를 저장하는 방법을 나타냅니다. 그러나 파티셔닝만으로 테넌트 격리를 보장할 수는 없습니다. 한 테넌트의 데이터에 다른 테넌트가 액세스할 수 없도록 하려면 추가 조치가 필요합니다.

다중 테넌트 SaaS 시스템의 세 가지 일반적인 데이터 파티셔닝 모델은 사일로, 풀 및 하이브리드입니다. 모델 선택은 다음과 같은 요인에 따라 달라집니다.

- 규정 준수
- 시끄러운 이웃
- 계층화 전략
- 운영 요구 사항
- 테넌트 격리 요구 사항

또한 에서 사용할 수 있는 각 데이터베이스 유형은 AWS 일반적으로 고유한 데이터 파티셔닝 및 테넌트 격리 모델 컬렉션을 제공합니다. 솔루션의 다양한 요구 사항을 지원하기 위해 테넌트 그래프를 구성하는 방법을 살펴볼 때 Amazon Neptune에서 제공하는 모델을 고려하세요.

많은 사용자가 다음 어설션 중 하나로 Neptune에서 설계를 ISVs 시작합니다.

- ISV 솔루션을 사용하려면 별도의 클러스터에서 고객을 물리적으로 분리해야 합니다.
- 이 ISV 솔루션에는 기존 관계형 데이터베이스 관리 시스템에서 찾을 수 있는 명명된 데이터베이스 또는 스키마와 같은 구성 요소가 필요합니다.

고려한 후에는 이러한 어설션이 사실이 아니라는 점을 ISVs 명심하세요. 거의 모든 워크로드에서 각 고객은 데이터베이스에 연결 해제된 그래프를 가지고 있기 때문입니다. 이 문서에서 설명하는 데이터 모델링 및 액세스 지침을 구현하면 이러한 데이터 경계가 교차되는 것을 방지하고 고객 데이터 프라이버시를 유지할 수 있습니다.

이 안내서에서는 사일로 모델과 풀 모델 을 모두 설명하지만 대부분 비용 및 운영 효율성을 위해 풀 모델을 ISVs 선택합니다. 이 가이드에서는 사일로 모델과 풀 모델의 측면을 모두 결합한 하이브리드 모델

에 대해 간략하게 설명합니다. 일부는 최대 고객을 위해 하이브리드 모델을 ISVs 사용하여 그래프 크기의 규제 또는 규정 준수 요구 사항을 수용합니다.

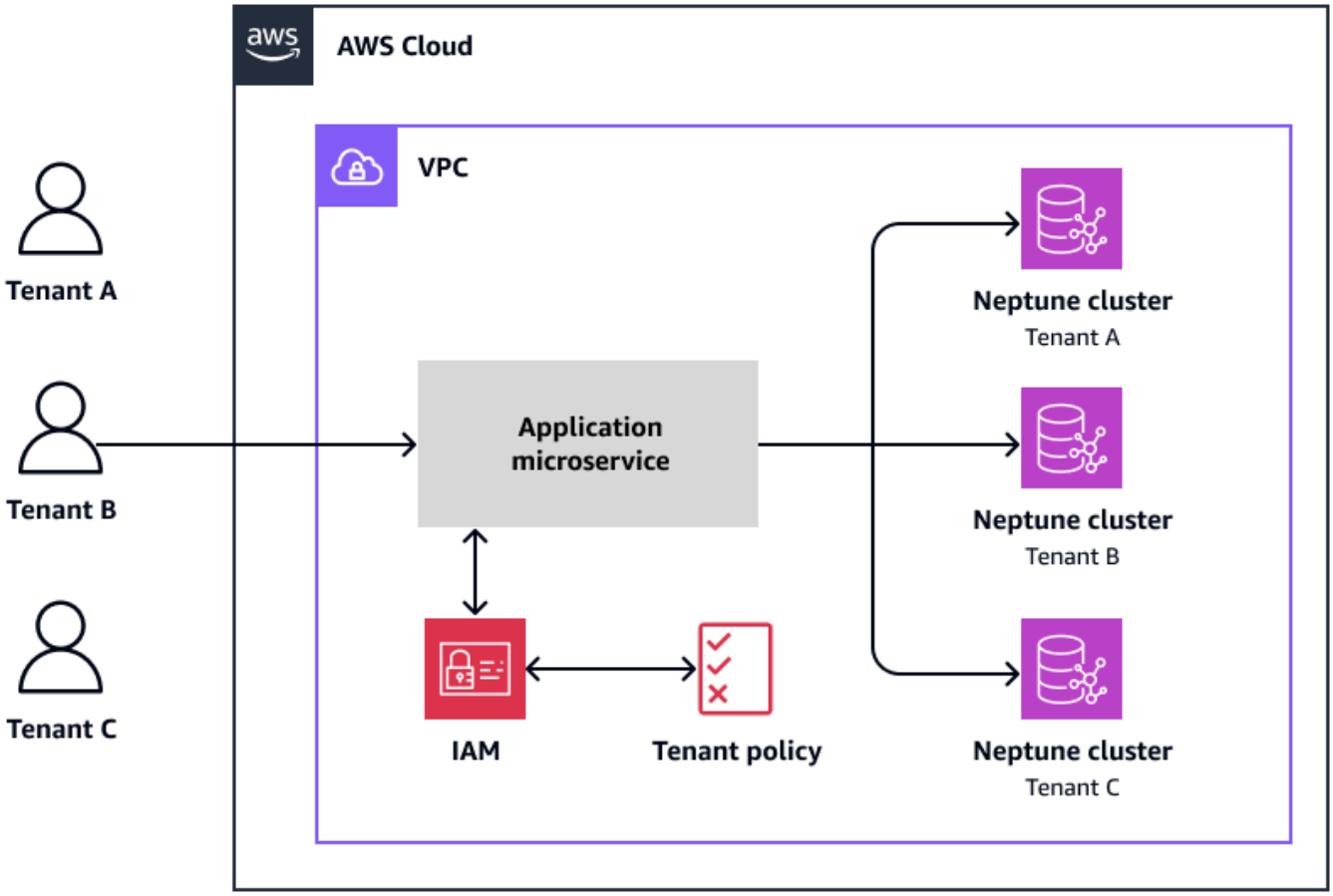
사일로 모델 다중 테넌시

일부 다중 테넌트 SaaS 환경에서는 규정 준수 및 규제 요구 사항으로 인해 테넌트의 데이터를 완전히 분리된 리소스에 배포해야 할 수 있습니다. 경우에 따라 대규모 고객에게는 시끄러운 이웃 영향을 줄이기 위한 전용 클러스터가 필요합니다. 이러한 상황에서는 사일로 모델을 적용할 수 있습니다.

사일로 모델에서 테넌트 데이터의 스토리지는 다른 테넌트 데이터와 완전히 격리됩니다. 테넌트의 데이터를 나타내는 데 사용되는 모든 구문은 해당 클라이언트에 대해 물리적으로 고유한 것으로 간주됩니다. 즉, 각 테넌트는 일반적으로 고유한 스토리지, 모니터링 및 관리를 갖게 됩니다. 또한 각 테넌트에는 암호화를 위한 별도의 AWS Key Management Service (AWS KMS) 키가 있습니다. Amazon Neptune에서 사일로는 테넌트당 하나의 클러스터입니다.

테넌트당 클러스터

클러스터당 하나의 테넌트를 보유하여 Neptune으로 사일로 모델을 구현할 수 있습니다. 다음 다이어그램은 각 테넌트에 대해 별도의 클러스터가 있는 Virtual Private Cloud(VPC)에서 애플리케이션 마이크로서비스에 액세스하는 세 개의 테넌트를 보여줍니다.



각 클러스터에는 효율적인 데이터 상호 작용 및 관리를 위해 고유한 액세스 포인트를 보장하는 데 도움이 되는 개별 엔드포인트가 있습니다. 각 테넌트를 자체 클러스터에 배치하면 테넌트 간에 잘 정의된 경계를 생성하여 고객이 다른 테넌트의 데이터와 데이터를 성공적으로 격리할 수 있도록 합니다. 이 격리는 엄격한 규제 및 보안 제약이 있는 SaaS 솔루션에도 유용합니다. 또한 각 테넌트에 자체 클러스터가 있는 경우 한 테넌트가 다른 테넌트의 경험에 부정적인 영향을 미칠 수 있는 부하를 부과하는 시끄러운 이웃에 대해 걱정할 필요가 없습니다.

cluster-per-tenant 사일로 모델에는 이점이 있지만 관리 및 민첩성 문제도 발생합니다. 이 모델의 분산 특성으로 인해 테넌트 활동 및 모든 테넌트의 운영 상태를 집계하고 평가하기가 더 어렵습니다. 이제 새 테넌트를 설정하려면 별도의 클러스터를 프로비저닝해야 하므로 배포도 더 어려워집니다. 클라이언트 업그레이드 및 버전이 데이터베이스 업그레이드와 긴밀하게 결합되면 공유 클라이언트 계층이 있는 환경에서 업그레이드가 더 어려워집니다.

Neptune은 서버리스 클러스터와 프로비저닝된 클러스터를 모두 지원합니다. 애플리케이션 워크로드가 서버리스 또는 프로비저닝된 인스턴스에서 더 잘 처리되는지 평가합니다. 일반적으로 워크로드의 수요 수준이 일정하면 프로비저닝된 인스턴스가 더 비용 효율적입니다. 서버리스는 짧은 기간 동안 데

이터베이스 사용량이 많고 장시간 가벼운 활동이 있거나 활동이 없는 까다롭고 가변적인 워크로드에 최적화되어 있습니다.

테넌트당 Neptune 프로비저닝 클러스터를 사용하는 경우 테넌트 수요의 최대 부하에 근접하는 인스턴스 크기를 선택해야 합니다. 서버에 대한 이러한 종속성은 SaaS 환경의 규모 조정 효율성과 비용에 계단식으로 영향을 미칩니다. SaaS의 목표는 실제 테넌트 부하에 따라 동적으로 크기를 조정하는 것이지만, Neptune 프로비저닝된 클러스터를 사용하려면 사용량이 많고 부하가 급증하는 경우를 고려하여 과도하게 프로비저닝해야 합니다. 오버프로비저닝은 테넌트당 비용을 높입니다. 또한 시간이 지남에 따라 테넌트 사용량이 변경되면 클러스터를 확장하거나 축소하려면 각 테넌트에 대해 별도로 적용해야 합니다.

Neptune 팀은 일반적으로 유휴 리소스로 인해 발생하는 더 높은 비용과 추가 운영 복잡성으로 인해 사일로 모델에 대해 조언합니다. 그러나 규제가 높거나 민감한 워크로드의 경우 이러한 추가 격리가 필요한 경우 고객은 추가 비용을 지불할 의향이 있을 수 있습니다.

사일로 모델에 대한 구현 지침

cluster-per-tenant 사일로 격리 모델을 구현하려면 AWS Identity and Access Management (IAM) 데이터 액세스 정책을 생성합니다. 이러한 정책은 테넌트가 자체 데이터가 포함된 Neptune 클러스터에만 액세스할 수 있도록 하여 테넌트의 Neptune 클러스터에 대한 액세스를 제어합니다. 각 테넌트에 대한 IAM 정책을 IAM 역할에 연결합니다. 그런 다음 애플리케이션 마이크로서비스는 IAM 역할을 사용하여 AWS Security Token Service () `AssumeRole` 메서드를 사용하여 세분화된 임시 자격 증명을 생성합니다AWS STS. 해당 테넌트의 Neptune 클러스터에만 액세스할 수 있는 이러한 자격 증명은 테넌트의 Neptune 클러스터에 연결하는 데 사용됩니다.

다음 코드 조각은 샘플 데이터 기반 IAM 정책을 보여줍니다.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "neptune-db:ReadDataViaQuery",
        "neptune-db:WriteDataViaQuery"
      ],
      "Resource": "arn:aws:neptune-db:us-east-1:123456789012:tenant-1-cluster/*",
      "Condition": {
        "ArnEquals": {
          "aws:PrincipalArn": "arn:aws:iam::123456789012:role/tenant-role-1"
```

```
        }
      }
    }
  ]
}
```

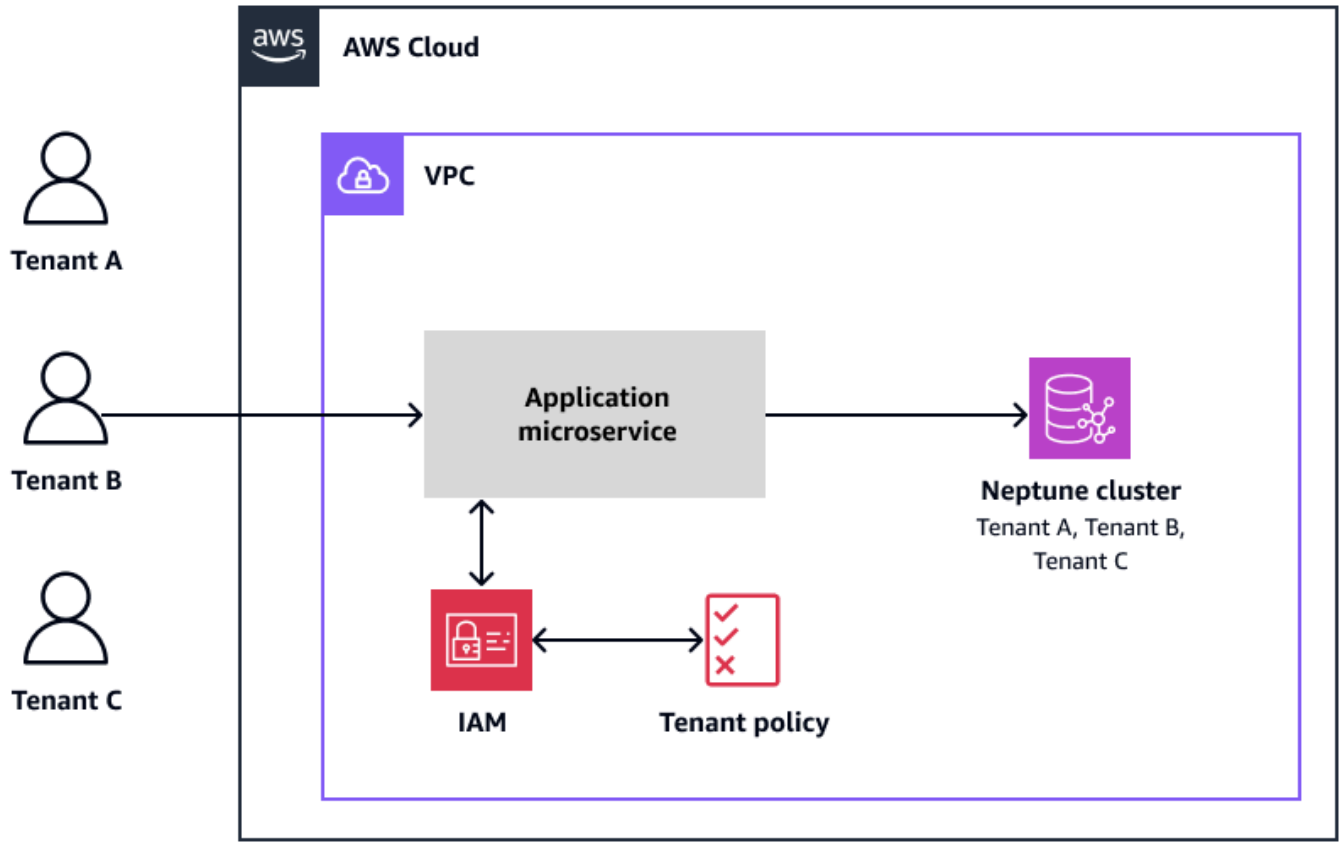
이 코드는 샘플 테넌트인에 해당 Neptune 클러스터에 대한 `tenant-1`읽기 및 쓰기 쿼리 액세스를 제공합니다. `Condition` 요소는 `tentant-1` IAM 역할()을 수임한 호출 엔터티(보안 주체`tenant-role-1`)만 `tenant-1`의 Neptune 클러스터에 액세스할 수 있도록 합니다.

풀 모델 다중 테넌시

비용이나 운영 오버헤드로 인해 사일로 모델을 구현할 필요나 실행이 불가능한 경우가 있습니다.

- 테넌트당 개별 클러스터를 유지 관리하는 리소스가 없을 수 있습니다.
- 각 테넌트의 데이터를 물리적으로 분리할 필요는 없으며 논리적 분리만으로도 요구 사항과 규정 준수 요구 사항을 충족할 수 있습니다.

다음 다이어그램은 테넌트 데이터가 단일 Amazon Neptune 클러스터에 배치되고 모든 테넌트가 공통 데이터베이스를 공유하는 풀 모델을 보여줍니다.



이 풀 격리 모델은 관리 오버헤드를 줄이고 관리할 클러스터가 적기 때문에 운영 효율성을 개선할 수 있습니다. 또한 컴퓨팅 리소스는 고객 비활성 기간 동안 유휴 상태를 유지하는 대신 여러 고객 간에 공유할 수 있습니다.

풀 모델을 사용하는 경우 데이터를 모델링하는 두 가지 방법이 있습니다. 접근 방식은 레이블이 지정된 속성 그래프(LPG)를 빌드하는지 아니면 리소스 설명 프레임워크(RDF)를 사용하여 그래프를 빌드하는지에 따라 달라집니다.

레이블이 지정된 속성 그래프의 풀 모델

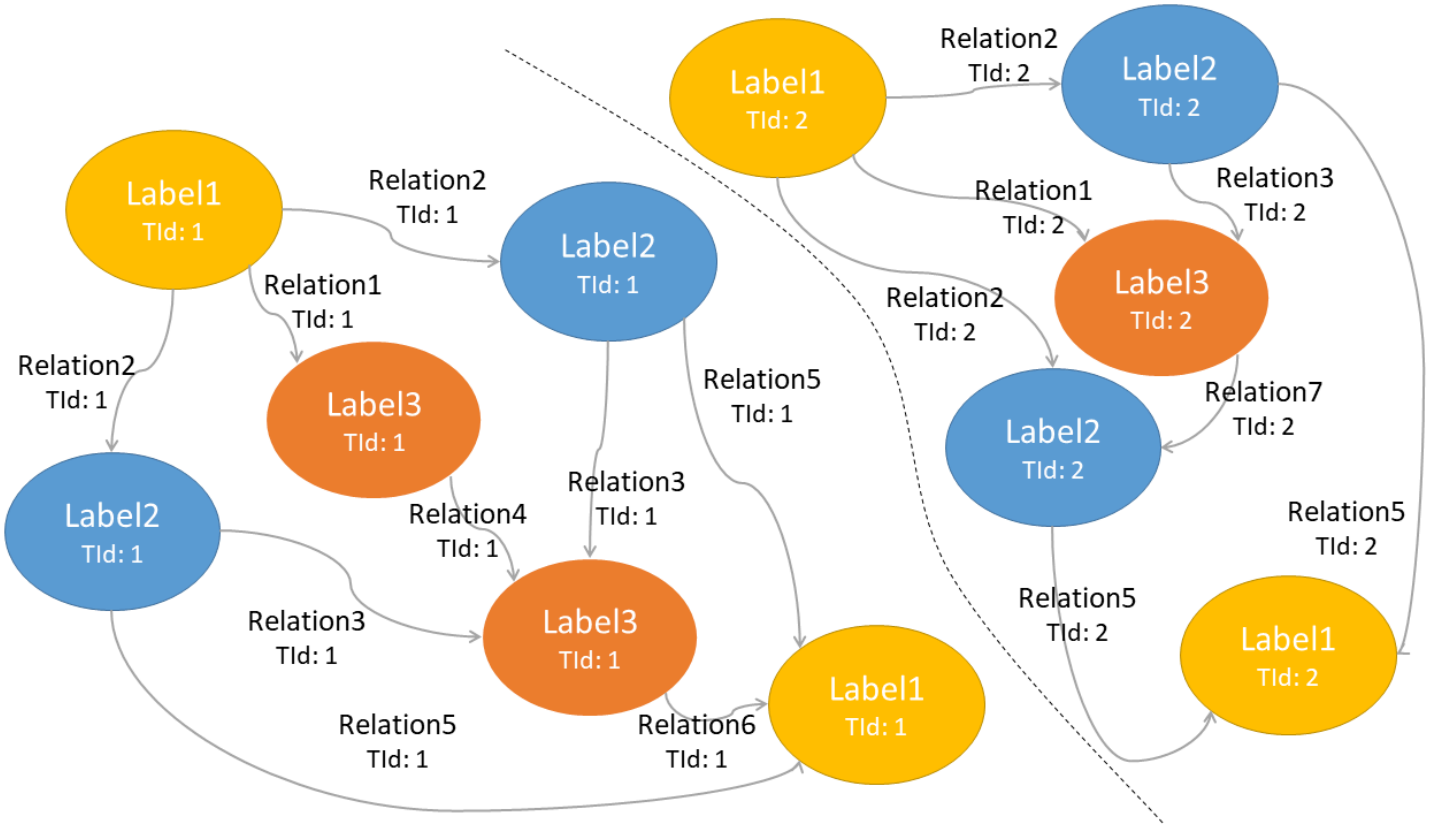
Amazon Neptune의 LPGs

- 속성 전략 – Apache TinkerPop Gremlin 언어의 PartitionStrategy와 같은 설정된 라이브러리 구문을 성능보다 우선적으로 사용해야 하는 경우 속성 전략을 선택합니다.
- 접두사 레이블 전략 – 성능 및 노이즈가 많은 이웃 효과를 제한하는 대부분의 시나리오에 접두사 레이블 전략을 사용하는 것이 좋습니다.
- 다중 레이블 전략 – 다중 레이블 전략은 접두사 레이블 전략의 성능이 향상되었습니다. 또한 클러스터의 모든 테넌트에 걸친 쿼리 실행을 지원합니다(예: 모든 테넌트에 대한 보고 또는 모니터링을 위한 ISV 쿼리).

속성 전략

LPGs 사용하면 사용자는 노드, 버텍스 및 엣지에 키-값 페어 속성을 추가할 수 있습니다. 논리적 분리를 달성하기 위해 대부분의 고객은 공통 테넌트 속성 키를 사용하여 이를 모든 노드 및 엣지에서 고유한 속성으로 직관적으로 모델링합니다. 테넌트 속성 키는 노드를 소유한 모든 테넌트를 나타냅니다. 테넌트 식별자는 개별 테넌트를 식별하는 고유한 값입니다.

다음 다이어그램은이 모델을 보여줍니다. 연결이 해제된 두 하위 그래프에는 레이블이 지정된 다양한 노드와 엣지가 있으며 테넌트 속성 키는 로 표시됩니다`TId`. 한 하위 그래프의 모든 노드와 엣지의 `TId` 값은 입니다`1`. 다른 하위 그래프에서는 모든 노드와 엣지의 `TId` 값이 입니다`2`.



레이블이 지정된 속성 그래프에는 이를 관리하는 두 가지 방법이 있습니다. Gremlin 쿼리 언어는 데이터의 데이터 파티셔닝을 관리하는 데 도움이 되는 PartitionStrategy 순회 라이브러리를 제공합니다 . 다음 예제의 코드는 모든 노드와 엣지에 라는 속성이 있을 것으로 예상합니다`TId`.

```
strategy1 = new PartitionStrategy(partitionKey: "TId", writePartition: "1",
 readPartitions: ["1"])
strategy2 = new PartitionStrategy(partitionKey: "TId", writePartition: "2",
 readPartitions: ["2"])
```

새 노드 또는 엣지가 작성되면 `"1"` 또는를 `strategy2` 선택했는지 여부에 `"2"`따라 `strategy1` 또는 값으로 속성`"TId"`이 추가됩니다. `"TId"`의를 사용하는 고객의 `"1"`경우를 사용합니다`strategy1`. 다음 예제에서는 해당 고객에 대한 데이터 쓰기를 보여줍니다.

```
g.withStrategies(strategy1).addV("Label1").property("Value", "123456").property(id,
 "Item_1")
```

읽기 쿼리의 경우 `"TId == '1'"` 또는에 대한 필터`"TId == '2'"`는 `strategy2`각각 `strategy1` 또는를 사용하여 모든 노드 또는 엣지 순회에 추가됩니다. 이러한 파티션 전략은 코드를 단순화하지만 필수는 아닙니다. 전략을 사용하면 권한 부여 수준에서 삽입하여 쿼리를 구성하는 하위 수준 코드로 전

달할 수 있다는 이점이 있습니다. 이렇게 하면 고객 식별자(TId)를 결정하는 코드가 쿼리의 로직과 분리됩니다.

다음 예제 코드는 데이터를 읽기 위한 Gremlin 쿼리를 보여줍니다.

```
g.withStrategies(strategy1).V().hasLabel("Label1")
```

위의 코드는 다음 예제와 동일합니다.

```
g.V().hasLabel("Label1").has("TId", "1")
```

마찬가지로 Gremlin을 사용하여 데이터를 작성할 때 다음 쿼리를 사용할 수 있습니다.

```
g.withStrategies(strategy1).addV("Label1").property("Value").property(id, "Item_1")
```

위의 코드는 파티션 전략을 사용하지 않으므로 "TId" 속성을 명시적으로 작성해야 하는 다음 예제와 동일합니다.

```
g.addV("Label1").property("TId", "1").property("Value").property(id, "Item_1")
```

openCypher에서는 이러한 라이브러리가 존재하지 않습니다. 노드 및 엣지에서 테넌트 식별자를 속성으로 추가하도록 쿼리를 작성하고 수정하는 것은 사용자의 책임입니다. 예제:

```
CREATE (n:Item {`~id`: 'Item_1', Value: '123456', TId: '1'})
CREATE (n:Item {`~id`: 'Item_2', Value: '123456', TId: '2'})
```

파티션 전략이 없는 Gremlin 코드 간의 유사성에 유의하세요. 그런 다음 다음 코드를 사용하여 첫 번째 CREATE 문에서 작성된 노드를 읽을 수 있습니다.

```
MATCH (n:Item {TId: '1'})
RETURN n
--or
MATCH (n:Item)
WHERE n.TId == '1'
RETURN n
```

PartitionStrategy와 같은 기본 TinkerPop Gremlin 구문을 사용하려는 경우 속성 전략을 선택할 수 있습니다. 그러나이 모델은 접두사 레이블 전략과 비교하여 Amazon Neptune에서 성능 단점이 있습니다. 이러한 성능 단점에 대한 자세한 내용은 LPG 모델의 성능 영향 섹션을 참조하세요.

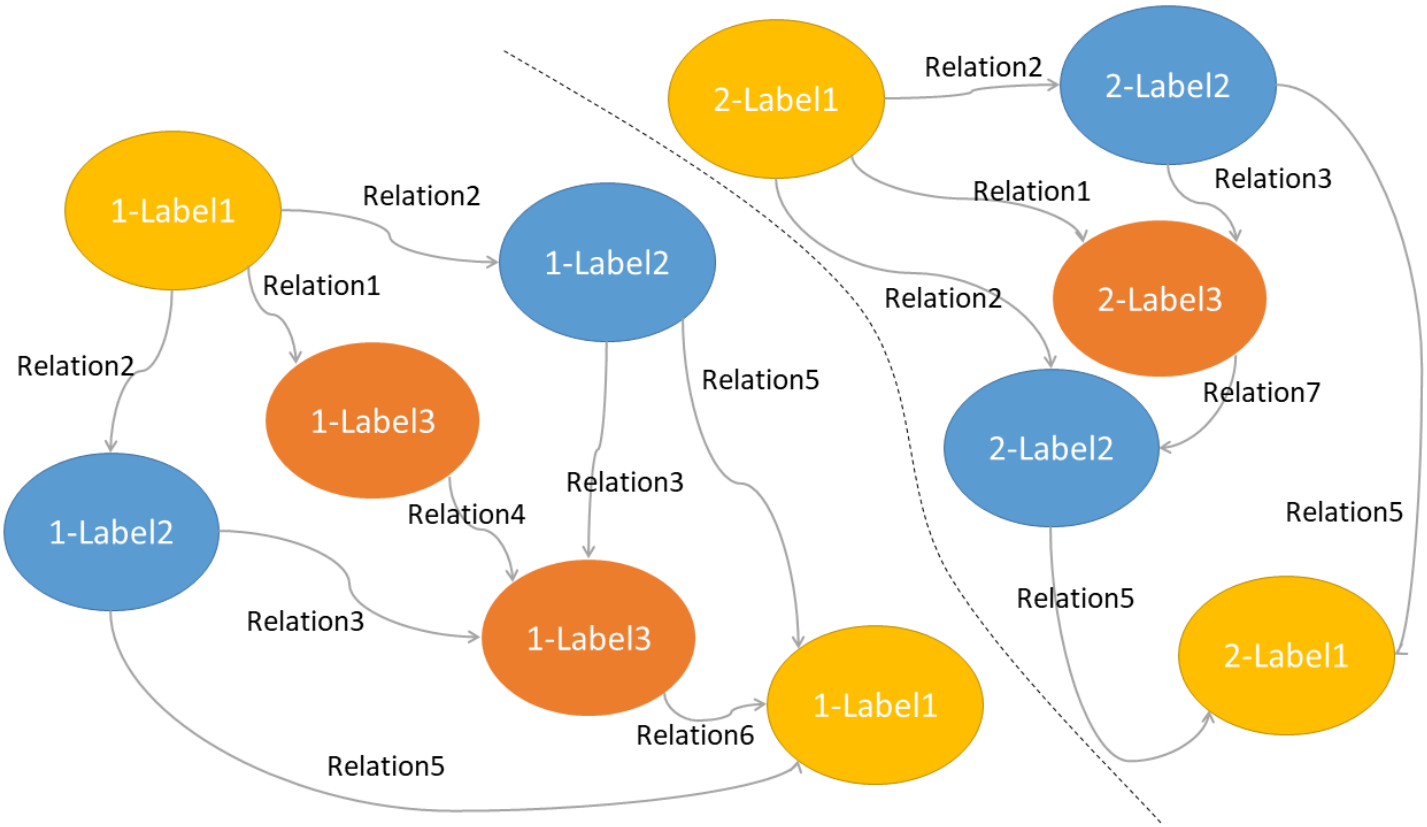
다음 조건이 적용되는 경우 엣지가 아닌 노드에서만 속성 전략을 모델링하는 것이 좋습니다.

- 그래프에는 레이블보다 훨씬 더 많은 엣지가 있습니다.
- 각 테넌트는 연결 해제된 그래프입니다.
- 노드를 레이블이 아닌 시작점으로 사용해야만 그래프에 액세스할 수 있습니다.

접두사 레이블 전략

성능이 가장 중요한 문제인 경우 속성 전략보다 접두사 레이블 전략을 고려하는 것이 좋습니다.

접두사 레이블 전략에서는 테넌트 식별자와 노드 레이블을 조합하여 각 노드에 레이블을 지정합니다. 예를 들어 테넌트의 식별자가 "1" 이고 노드 레이블이 인 경우 노드 레이블을 로 "Label1"지정합니다"1-Label1". 다음 다이어그램은이 모델을 사용하는 두 개의 연결 해제된 하위 그래프를 보여줍니다.



Gremlin에서 데이터를 쓸 때 노드의 레이블에 식별 번호를 추가할 수 있습니다.

```
g.addV("1-Label1")
g.addV("2-Label6")
```

이 그래프를 쿼리할 때 노드에이 접두사가 있는지 확인할 수 있습니다.

```
g.V().hasLabel("1-Label1")
```

openCypher에서는 CREATE 문을 사용하여 데이터를 작성할 수 있습니다.

```
CREATE (n:`1-Label1` {`~id`: 'Item_1', Value: 'XYZ123456'})
```

openCypher에서 작성한 데이터를 쿼리하려면 다음 코드를 사용합니다.

```
MATCH n= (:`1-Label1`)
RETURN n
```

접두사 레이블 전략은 모든 노드가 하나 이상의 테넌트에 할당되고 엣지 범위에서 권한이 할당되지 않는다고 가정합니다. 엣지 레이블에이 전략을 사용하지 마세요. 이렇게 하면 많은 수의 조건자가 발생하고 Neptune 성능에 부정적인 영향을 미치기 때문입니다.

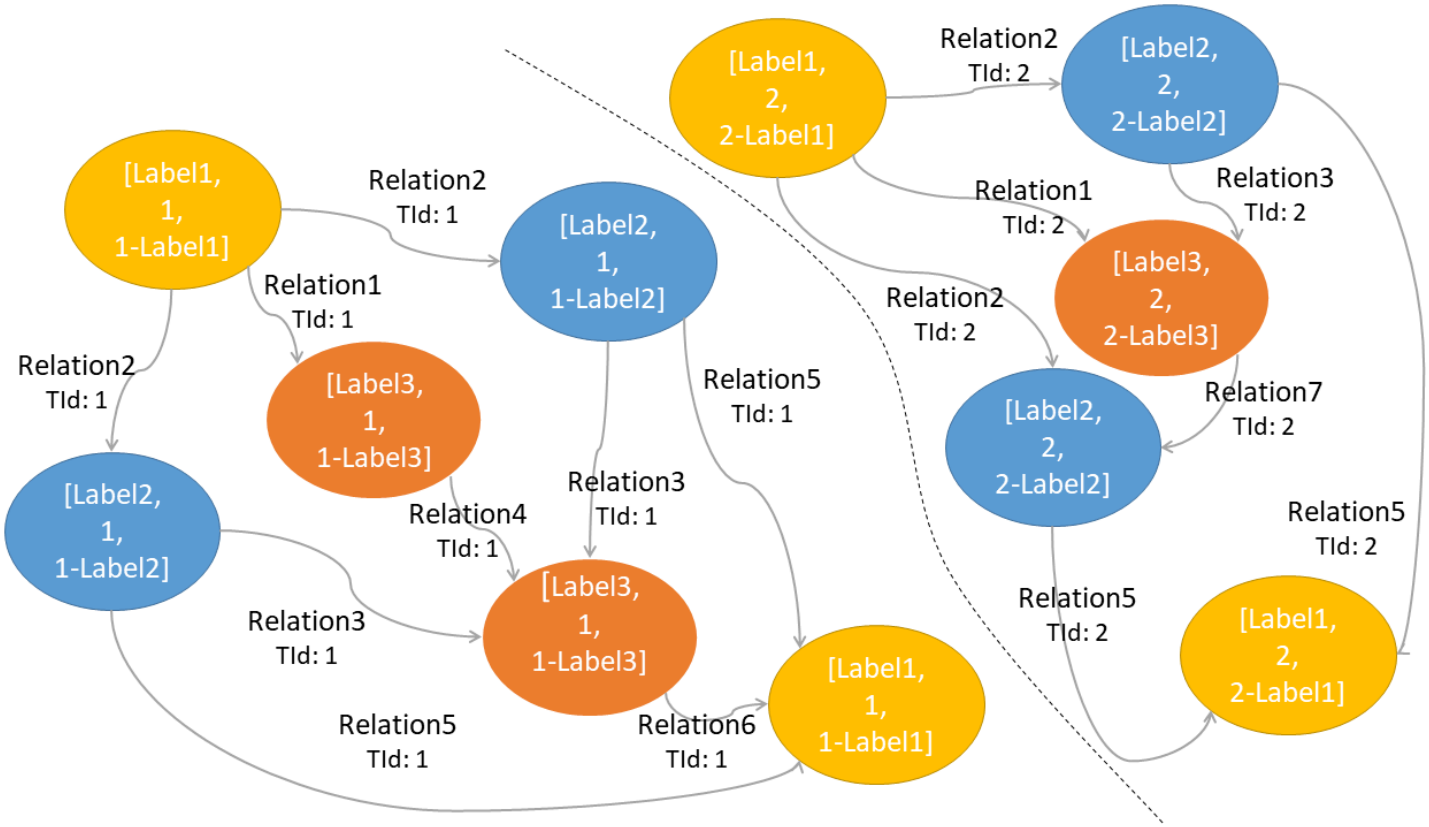
접두사 레이블 접근 방식에는 두 가지 주요 단점이 있습니다. 먼저 테넌트 전체에 걸쳐 있는 쿼리를 실행하기는 어렵습니다. 예를 들어 보고 또는 모니터링을 위해 지정된 레이블의 모든 노드를 계산하는 쿼리가 있습니다. 사용 사례인 경우이 전략을 다중 레이블 전략과 결합하는 것이 좋습니다. 전략 결합에 대한 자세한 내용은 하이브리드 모델 섹션을 참조하세요.

둘째, 접두사 레이블 전략에는 데이터 유출을 방지하기 위해 모든 쿼리에 적절한 접두사를 적절하게 적용하는 제어가 필요합니다. 그러나이 전략은 지연 시간이 짧은 쿼리가 필요한 워크로드에 가장 효율적인 옵션이므로 적극 권장합니다. LPG 모델의 성능 영향 섹션에서는 이것이 가장 효율적인 전략인 이유의 예를 제공합니다.

다중 레이블 전략

세 번째 옵션은 다중 레이블 전략을 사용하는 것입니다. 이 접근 방식의 경우 그래프의 모든 노드에 레이블을 추가합니다. 예를 들어 지정된 테넌트에 대한 모든 데이터를 필터링해야 하는 경우 테넌트 ID 레이블을 추가합니다. 테넌트에 관계없이 지정된 레이블에 대한 모든 데이터를 필터링해야 하는 경우 해당 레이블을 추가합니다. 다음 다이어그램은 각 노드에 대해 세 개의 레이블을 사용하여 적용되는 다중 레이블 전략을 보여줍니다.

이제 세 가지 패턴을 사용하여 그래프에 액세스할 수 있습니다.



- 모든 Label1 테넌트에서를 사용하여 모든 노드를 반환Label1하려면를 필터링합니다.
- 를 필터링1하여 테넌트 1의 모든 노드를 반환합니다.
- 레이블이 인 테넌트 1에 대한 모든 노드만 반환1-Label1하려면를 필터링합니다Label1.

LPGs 경우 이를 구현하는 두 가지 방법이 있습니다.

Gremlin에서는 SubgraphStrategy라는 순회 전략을 사용하여 "Label1"모든 쿼리의 범위를와 같은 특정 레이블이 있는 버텍스로만 제한할 수 있습니다.

```
g.withStrategies(
    new SubgraphStrategy(
        vertices=hasLabel("Label1")
    )
)
```

PartitionStrategy와 달리 SubgraphStrategy는 데이터 쓰기가 아닌 데이터 읽기에만 영향을 미칩니다. 데이터를 작성하려면 각 쿼리에 레이블을 수동으로 할당합니다.

```
g.addV("Label1").property("Value","XYZ123456")
```

```
.addV("Label2").property("Value","XYZ123456")
```

데이터를 읽을 때 SubgraphStrategy를 사용하여를 사용하여 모든 노드를 쿼리할 수 있습니다"Label1".

```
g.withStrategies(
new SubgraphStrategy(vertices=.hasLabel("Label1"))
).
V().has("Value","XYZ123456")
```

Neptune은 "Label1" 및 값이 인 첫 번째 레코드만 반환합니다"XYZ123456". SubgraphStrategy를 사용하지 않는 다음 쿼리와 동일합니다.

```
g.V().hasLabel("Label1").hasValue("XYZ123456")
```

이 기본 쿼리에서는 SubgraphStrategy가 사용하기 더 복잡한 것으로 보입니다. 라이브러리는 이미 정의된 전략을 g 사용하여의 인스턴스를 제공할 수 있습니다. 개발자는 적절한 필터가 적용되도록 할 필요가 없습니다.

```
def getGraphTraversal():
return g.withStrategies(new SubgraphStrategy(vertices=.hasLabel("Label1"))

getGraphTraversal().has("Value","XYZ123456")
```

openCypher 라이브러리에는 이러한 구문이 없으므로 각 노드에 대해 여러 레이블을 생성해야 합니다.

```
CREATE (n:`1`:`Label1`:`1-Label1` {`~id`: 'Item_1', Value: '12345'})
```

이러한 레이블을 사용하여 하위 그래프를 필터링할 때 찾고 있는 고객 레이블이 있거나 해당 레이블이 있는 다른 노드와 관계를 공유하는 노드를 반환할 수 있습니다.

```
MATCH n=(:Label1:`1`)
// or
MATCH n=(:`1-Label1`)
```

다중 레이블 전략은 유형(Label1) 또는 테넌트()별로 노드를 쿼리하거나 성능이 가장 중요한 경우(1) 보다 효율적인 접두사 레이블 전략을 사용할 수 있는 유연성을 제공합니다1-Label1.

이 전략의 주요 단점은 각 레이블이 그래프에 저장된 추가 객체라는 것입니다. 객체는 LPGs. 수집 속도는 초당 객체로 측정 및 제한되며 스토리지 비용은 소비되는 기가바이트 수에 따라 달라집니다. 즉, 추가 객체는 대규모로 측정 가능한 영향을 미칠 수 있습니다.

LPG 모델의 성능 영향

Amazon Neptune용 AWS Skill Builder 과정 데이터 모델링에서는 Neptune 데이터 모델 내부 및 모델링 영향에 대해 자세히 설명하지만, 여기에 이러한 설계에 대한 중요한 고려 사항을 요약하겠습니다. Amazon Neptune 단일 Neptune 클러스터에 세 개의 테넌트(T1, T2, T3)를 두는 것이 좋습니다. 이러한 테넌트에는 다음과 같은 속성이 있습니다.

- 테넌트 1(T1)에는 총 1억 개의 노드가 있고 1천만 개의 노드가 항목 유형입니다.
- 테넌트 2(T2)에는 총 1천만 개의 노드가 있고 1백만 개의 노드가 항목 유형입니다.
- 테넌트 3(T3)에는 총 1억 개의 노드가 있고 1백만 개의 노드가 항목 유형입니다.

속성 전략을 사용하여 테넌트 3의 항목을 검색할 쿼리를 실행합니다. Neptune은 통계에서 두 개의 인덱스 호출을 검사합니다.

- `tenant property key=T3`에서 1억 개의 결과가 있는 경우
- `label = Item` 1,200만 개의 결과(T1에서 1,000만 개 + T2에서 100만 개 + T3에서 100만 개)

Neptune 쿼리 옵티마이저는 후자의 쿼리가 먼저 가장 잘 적용된다고 판단한 다음(1,200만 결과) 각 항목의를 검사합니다`tenant property key=T3`. 1200만 개의 항목을 검색하여 100만 개의 결과를 찾습니다.

이 쿼리의 노이즈가 많은 이웃 영향을 확인합니다. 테넌트당 항목 노드가 1억 개 있는 경우 첫 번째 쿼리는 1,200만 개가 아닌 3억 개의 결과를 갖게 됩니다(이는 설명을 위해 지나치게 간소화되었습니다. Neptune 옵티마이저가 다른 작업 순서를 적용했을 수 있습니다).

다음으로 접두사 레이블 전략을 고려합니다. 1백만 개의 결과를 반환하는 `label=T3-Item`에서 단일 인덱스 호출을 수행합니다. 이렇게 하면 속성 전략과 동일한 결과를 얻지만 1,100만 개의 레코드를 더 적게 검색합니다. 또한 레이블이 인덱스에서 겹치지 않기 때문에 더 이상 시끄러운 이웃 문제가 없습니다.

다중 레이블 전략은 속성 전략에 대한 쿼리 성능을 직접 개선하지 않습니다. 속성 값을 기준으로 필터링하는 것은 검색 공간도 비슷할 때 레이블 값을 기준으로 필터링하는 것과 비슷합니다. 대신 다중 레이블 전략은 더 많은 유연성을 지원합니다. 다중 레이블 전략은 `label=T3` 또는 레이블에 대한 접두사

레이블 전략과 동등한 성능을 제공합니다`T3-Item`. 다중 레이블 전략은의 속성 전략과 동등한 성능을 제공합니다`label=Item`. 이점은 다양한 액세스 패턴을 지원하는 것입니다.

RDF용 풀 모델

리소스 설명 프레임워크(RDF)에는 데이터를 분리하는 논리적 방법을 제공하는 명명된 그래프 개념이 있습니다. Amazon Neptune에는 기본 명명된 그래프와 사용자 정의 명명된 그래프가 있습니다. 원하는 수만큼 명명된 그래프를 생성할 수 있습니다. 집합적으로 RDF 데이터 세트라고 합니다. 기본 또는 사용자 정의의 모든 명명된 그래프는 RDF 데이터 세트 내의 국제화된 리소스 식별자(IRI)로 정의됩니다. Neptune에서 사용자가 데이터를 쓸 때 명명된 그래프를 선언하지 않는 한 모든 트리플은 명명된 기본 그래프의 일부로 간주됩니다.

명명된 그래프에는 여러 사용 사례가 있습니다.

- 데이터 파티셔닝 및 데이터 격리
- 데이터 출처
- 버전 관리
- Inference

이 가이드는 데이터 파티셔닝 사용 사례에 중점을 둡니다. 각 테넌트에 대해 사용자 정의 명명된 그래프를 하나씩 생성하는 것이 좋습니다.

Graph Store HTTP 프로토콜을 사용한 SPARQL 쿼리 옵션

다음 예제 쿼리는 SPARQL 프로토콜 및 RDF 쿼리 언어(SPARQL) 및 그래프 스토어 HTTP 프로토콜을 사용하여 테넌트에 대한 명명된 그래프를 쿼리하거나 생성합니다.

- `HTTP GET` – 테넌트의 특정 그래프를 검색하려면:

```
curl --request GET 'https://your-neptune-endpoint:port/sparql/gsp/?graph=http%3A//
www.example.com/named/tenant1'
```

- `HTTP PUT` – 명명된 특정 그래프를 생성하거나 요청에 지정된 페이로드로 바꾸려면:

```
curl --request PUT -H "Content-Type: text/turtle" \  --data-raw "@prefix ex: http://
example.com/ . ex:subject ex:predicate ex:object ." \
'https://your-neptune-endpoint:port/sparql/gsp/?graph=http%3A//www.example.com/named/
tenant1'
```

RDF에서 객체는 트리플입니다.

- HTTP POST – 그래프가 없는 경우 명명된 새 그래프를 생성하거나 기존 그래프와 병합하려면:

```
curl --request POST -H "Content-Type: text/turtle" \
--data-raw "@prefix ex: http://example.com/ . ex:subject ex:predicate ex:object ." \

'https://your-neptune-endpoint:port/sparql/gsp/?graph=http%3A//www.example.com/named/tenant1'
```

RDF에 대한 테넌트 격리

애플리케이션 계층에 필요한 가드레일이 있는 데이터를 논리적으로 격리하려면 테넌트와 사용자 정의 명명된 그래프 간에 매핑을 생성합니다. RDF 데이터 세트에 대한 다중 테넌시를 설계할 때는 RDF 및 SPARQL의 다음 측면에 유의하세요.

- Neptune에서는 명명된 그래프를 지정하지 않고 쿼리하면 데이터베이스의 명명된 모든 그래프에서 패턴과 일치하는 모든 트리플을 검색합니다.
- RDF에는 이름이 지정된 그래프가 다른 노드 간의 연결에 대한 제약이 없습니다. 예를 들어 이전 다이어그램에서의 노드는 엣지를 통해 G2의 노드에 연결할 :G1 수 있습니다.

예를 들어 특정 테넌트의 최종 사용자가 API에 쿼리를 제출하는 경우 API는 Neptune 데이터베이스에 쿼리를 제출하기 전에 다음 요구 사항을 검증해야 합니다.

- 단일 테넌트로 범위가 지정된 모든 쿼리는 명명된 그래프를 지정해야 합니다. 그렇지 않으면 테넌트 간에 데이터가 유출될 위험이 있습니다.
- 쿼리 업데이트 또는 삭제는 항상 명명된 그래프를 지정해야 합니다.
- 엣지 또는 관계의 양쪽에 있는 노드는 항상 올바른 명명된 그래프에 속해야 합니다.

모범 사례에 대한 자세한 내용은 Neptune 설명서를 참조하세요.

성장 준비

풀 모델을 성공적으로 사용하면 결국 단일 Neptune 클러스터의 크기를 능가하게 됩니다. 테넌트가 증가하거나 테넌트 수가 증가하며 모든 고객에게 필요한 데이터 수집 속도가 클러스터의 기능을 초과합니다. 이 경우 고객을 여러 클러스터로 분할해야 합니다. 나중에 새로 고치려고 시도하는 대신이 구성

을 미리 설계합니다. 초기 규모가 단일 클러스터만 사용하는 경우에도 향후 해당 규모에 도달하면 여러 클러스터에서 테넌트를 라우팅하는 데 필요한 구성 요소를 모의합니다.

솔루션에 테넌트의 크기에 따라 더 많은 리소스가 필요한 경우 리소스 증가에도 대비하세요. 단일 클러스터의 여러 고객이 크게 성장하는 경우 해당 클러스터가 더 이상 요구 사항을 지원하지 않을 수 있습니다. Amazon Neptune DB 복제 기능을 사용하여 테넌트를 다른 클러스터로 이동하거나 기존 클러스터를 두 개로 분할하는 전략을 설계합니다.

DB 복제를 구현할 때 비용을 절감할 수 있는 Neptune Copy-on-Write 프로토콜을 숙지합니다., 수집 병목 현상으로 인해 클러스터를 분할하는 경우 정책에서 허용하는 경우 클러스터에서 데이터를 삭제하지 않는 것이 더 효율적일 수 있습니다. 두 클러스터는 변경되지 않은 경우 데이터 페이지를 공유하지만 데이터 페이지가 수정된 경우에는 공유하지 않습니다(일부 데이터가 삭제되었기 때문).

> **Note**
>
> 이 지침은이 작성 시점의 최신 Neptune 버전인 Neptune 버전 1.3.1에 적용됩니다. 이 지침은 Neptune 스토리지 계층이 발전함에 따라 향후 버전에서 변경될 수 있습니다.

다중 테넌시 시나리오에 대한 제한 사항

일부 Neptune 기능은 다중 테넌시 시나리오용으로 빌드되지 않습니다. 이러한 다중 테넌시 전략은 데이터베이스 수준에서 적용되지 않으므로 풀 모델의 Neptune 엔드포인트에 대한 직접 액세스 권한을 테넌트에게 부여해서는 안 됩니다. 항상 고객과 Neptune 엔드포인트 간에이 문서에 설명된 설계를 적용하는 일종의 프록시를 유지합니다. 이러한 프록시의 예는 다음과 같습니다.

- 클라이언트 계층에 레이블 필터 추가
- 인증 토큰을 테넌트 ID에 매핑하고이 필터를 쿼리에 삽입하는 API 보유

이 지침은 고객에게 Neptune 그래프 노트북, Neptune 그래프 탐색기 또는 Neptune 스트림과 같은 기능에 직접 액세스할 수 있는 권한을 부여하는 경우에도 적용됩니다.

하이브리드 모델 다중 테넌시

SaaS 솔루션은 사일로 모델과 풀 모델을 혼합하여 사용하는 경우가 많습니다. 다양한 요인이 동일한 환경 내에서 사일로 모델과 풀 모델을 모두 사용하는 시기와 방법을 결정하는 데 영향을 미칩니다.

이러한 요소 중 하나는 계층화로, SaaS 솔루션은 테넌트의 각 계층에 고유한 경험을 제공합니다. 예를 들어 티어가 무료, 표준 및 프리미엄인 경우 풀 모델을 사용하여 프리 티어 테넌트 데이터를 공유 Neptune 클러스터에 저장할 수 있습니다. Standard 및 Premium 계층 테넌트의 경우 cluster-per-tenant 사일로 모델을 사용할 수 있습니다.

또한 일부 SaaS 공급자는 공유 Amazon Neptune 클러스터에 풀 솔루션을 기반으로 구축할 수 있습니다. 그런 다음 규정 준수 및 규제 의무로 인해 격리된 스토리지가 필요한 테넌트를 위한 별도의 Neptune 클러스터를 생성할 수 있습니다.

이렇게 하면 데이터 액세스 계층 및 관리 프로파일에 복잡성이 증가할 수 있지만, 고객 요구 사항을 충족하기 위해 제안을 계층화할 수 있는 방법도 제공할 수 있습니다.

ISVs 운영 모범 사례

이 섹션의 많은 지침은 모든 고객에 대한 모범 사례이지만 ISVs.

최신 버전으로 Neptune 클러스터 업데이트

Amazon Neptune 릴리스 정보에서 모든 버전에 여러 버그 수정, 성능 개선 및 새로운 기능이 포함되어 있음을 확인할 수 있습니다. Neptune 클러스터를 가능한 한 최신 버전으로 유지합니다.

워크로드에서 이전에 검색되지 않은 버그를 발견하고 클러스터가 최신 버전인 경우 Neptune 엔지니어는 클러스터에 대한 프라이빗 패치를 생성할 수 있습니다(보증되고 원하는 경우). 패치는 해당 수정 사항을 정식으로 사용할 수 있는 다음 릴리스까지 연결될 수 있습니다. 클러스터를 최신 버전으로 업데이트하는 데 도움이 되도록 Neptune 블루/그린 솔루션을 사용합니다.

데이터 수집을 위해 삭제 및 교체 대신 델타 사용

여러 기술을 사용하여 Neptune에 데이터를 수집하거나 쓸 수 있습니다. 많은 고객이 피드에 변경 사항이 수신될 때마다 그래프를 삭제하고 다시 삽입하여 데이터 수집을 간소화하려고 합니다. 각 노드에 `last-modified` 속성을 추가하고 지정된 날짜 이후 수정되지 않은 노드를 주기적으로 스캔하여 삭제할 수 있습니다. 이러한 기술은 데이터 수집 프로세스를 단순화하지만 Neptune 클러스터에 장기적인 상태 및 확장성 영향을 미칩니다.

먼저 Neptune은 문자열의 사전 인코딩을 사용합니다. 노드 및 엣지의 IDs를 명시적으로 지정하지 않는 한 Neptune은 ID에 대한 문자열로 표시되는 GUID를 생성하고 해당 문자열을 사전에 저장합니다. 객체를 지속적으로 삭제하고 추가하는 경우 자동으로 생성된 IDs로 인해 사전에 팽창이 발생합니다.

둘째, Neptune은 최대 초당 약 120K 객체를 수집하도록 스케일 업합니다. 객체를 지속적으로 삭제하고 추가하는 경우 기본적으로 변경되지 않는 객체에서 해당 대역폭을 많이 소비합니다. 이렇게 하면 클러스터에서 호스팅할 수 있는 테넌트 수가 제한되고, 클러스터에 더 큰 라이터 인스턴스가 필요하며, 더 많은 I/O 작업이 필요합니다. 이러한 모든 요인으로 인해 비용이 증가합니다.

삭제 및 추가 메서드를 사용하는 대신 변경된 내용의 실제 델타를 계산하는 방법을 개발하는 것이 좋습니다. 그러나 일부 데이터 소스는 이에 도움이 되지 않습니다(예: 현재 상태를 반환하는 API 호출 또는 변경된 내용을 정확하게 추적하지 않는 이벤트). 원시 데이터 소스가 변경 사항을 식별하는 데 도움이 되지 않는 경우 추출, 변환 및 로드(ETL) 프로세스를 사용하여 델타를 계산합니다. 예를 들어 각 이전 데이터 캡처의 스냅샷을 Parquet 형식으로 유지하고, AWS Glue 를 사용하여 해당 스냅샷 간의 차이를 계산하고, 차이점만 Neptune에 푸시할 수 있습니다.

테넌트와 함께 Neptune 비용이 어떻게 발전할지 모델링합니다.

사일로, 풀 또는 하이브리드 모델을 사용하든 클라우드 비용은 테넌트의 크기에 따라 조정됩니다. 동시 연결이 더 많이 필요한 테넌트는 동시 연결이 적은 테넌트보다 더 큰 인스턴스 또는 더 많은 읽기 전용 복제본이 필요합니다. 더 빠른 데이터 수집이 필요한 테넌트에도 마찬가지입니다.

Neptune 클러스터 비용의 세 가지 구성 요소는 인스턴스 크기(및 수), 데이터 크기(GB-월), I/O 작업(백만 개당)입니다. 이러한 비용은 일반적으로 워크로드별로 다르지만 크기 및 데이터 볼륨에 따라 규모가 조정되지만 AWS 도구를 사용하여 측정할 수 있습니다. 시간이 지남에 따라 크기가 어떻게 달라지는지 포함하여 테넌트 크기의 주요 지표를 기준으로 규모 조정의 경제를 추적하고 이해합니다. I/O 요금의 예측 불가능이 마진에 영향을 미치는 경우 보다 예측 가능한 비용으로 Neptune I/O 최적화 스토리지를 선택하는 것이 좋습니다.

고객 수요에 맞게 클러스터 확장

Neptune 인스턴스 크기의 크기를 올바르게 조정하기 위한 시도된 공식이나 실제 공식은 없습니다. Neptune 설명서는 지침을 제공하지만 직접 매핑을 추천할 변수가 너무 많습니다. 이러한 변수에는 다음이 포함되지만 이에 국한되지는 않습니다.

- 데이터 모델
- 데이터 셰이프
- 쿼리 동시성
- 쿼리 복잡성

워크로드 및 테넌트 프로파일에 대한 최적의 크기를 결정하기 위한 테스트를 계획합니다. 일반적으로 비용 효율성과 예측 가능성을 위해 프로비저닝된 인스턴스를 사용하는 것이 좋습니다. 고객 경험 목표가 비용보다 최적의 규모 조정을 우선시하는 경우 Neptune Serverless 인스턴스를 사용하여 워크로드 변동에 관계없이 보다 일관된 경험을 보장하는 것이 좋습니다.

테넌트 읽기 워크로드의 피크와 트러프에 상당한 변동성이 있는 경우 Neptune Serverless 인스턴스를 Neptune Auto Scaling과 결합합니다. 새 읽기 전용 복제본이 초기화된 후 온라인 상태가 되는 데 보통 10~15분이 걸립니다. 즉, Auto Scaling만으로는 트래픽의 장기적인 변화를 처리할 수 있지만 빠르게 변화하는 활동 급증에는 충분하지 않습니다. Neptune Serverless와 Neptune Auto Scaling을 결합하여 인스턴스를 확장 또는 축소하고 읽기 전용 복제본 수를 확장 및 축소할 수 있습니다.

테넌트의 워크로드 프로파일 또는 서비스 수준 계약(SLAs)이 크게 다른 경우 사용자 지정 엔드포인트 및 전용 읽기 전용 복제본을 사용하여 트래픽을 해당 트래픽에 최적화된 인스턴스로 전달하는 것이 좋

습니다. 최적화에는 인스턴스의 다른 크기, 특정 쿼리 패턴 또는 버퍼 캐시 사전 워밍이 포함될 수 있습니다.

다음 단계

다중 테넌시 ISV 애플리케이션을 위해 Amazon Neptune을 구현하는 여정을 시작하는 경우 원하는 모델에 추가 고려를 하세요. 모델을 변경하면 여정 후반부에 비용이 더 많이 듭니다.

여정의 초반인 경우 필요에 가장 적합한 모델을 사용하고 있는지, 해당 모델의 지침을 따르고 있는지 확인합니다.

미리 계획을 세웁니다. 여정의 초반에는 정점과 엣지를 삭제하고 재추가하는 대신 변경 사항을 델타에 제공하도록 클러스터 간 고객 샤딩 작업을 연기하거나 ETL 프로세스를 최적화하는 것이 좋습니다. 규모를 조정하면 이러한 결정이 성능과 비용에 부정적인 영향을 미칠 수 있습니다.

마지막으로, 여정에 이미 잘 참여하고 있다면 이 지침은 아키텍처가 최적이라는 확신을 주거나 아키텍처를 개선하기 위한 변경 사항을 제공할 수 있습니다.

이 지침에 대한 질문이 있거나 추가 지원이 필요한 경우 AWS 계정 팀에 문의하여 Neptune 전문가와의 세션을 요청하세요.

리소스

- Amazon Neptune 설명서
- Amazon Neptune용 데이터 모델링(과정)
- Amazon Neptune에 AWS Well-Architected 프레임워크 적용
- SaaS 렌즈 Well-Architected 프레임워크
- 의 다중 테넌트 아키텍처에 대한 지침 AWS
- SaaS 테넌트 격리 전략: 다중 테넌트 환경에서 리소스 격리
- Apache TinkerPop 설명서
- SPARQL

기여자

이 가이드의 기여 요인은 다음과 같습니다.

- Brian O'Keefe, 보안 주체 WW SSA Neptune, AWS
- Veeresham Gande, 선임 기술 계정 관리자, AWS
- Dana Owens, Startup Solutions Architect, AWS
- Nima Seifi, Startup Solutions Architect, AWS

문서 이력

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

변경 사항	설명	날짜
최초 게시	—	2024년 9월 3일

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