Amazon Neptune データベースを実行する ISVs のマルチテナンシーガイダンス

AWS 規範ガイダンス

AWS 規範ガイダンス: Amazon Neptune データベースを実行する ISVs のマルチテナンシーガイダンス

Table of Contents

Amazon Neptune データベースを実行する ISVs のマルチテナンシーガイダンス

Amazon Web Services (寄稿者)

2024 年 8 月 (ドキュメント履歴)

マルチテナンシーは、アプリケーションの単一のインスタンスが複数のお客様にサービスを提供するコンピュータシステムアーキテクチャです。各顧客はテナントと呼ばれます。マルチテナントアーキテクチャでは、アプリケーションのこれらのインスタンスは、各テナントが同じインフラストラクチャに物理的に配置され、論理的に分離されている共有環境で動作します。

独立系ソフトウェアベンダー (ISV) として、Amazon Neptune を使用して、高度に接続されたデータ間のナビゲーションを必要とするアプリケーションを強化できます。アカウントでクラウドベースの Software as a Service (SaaS) アプリケーションを管理し、テナントにサブスクリプションを提供する場合があります。その後、テナントはインターネット経由で、または 経由でプライベートにサービスにアクセスできます AWS PrivateLink。このモデルの経済性は、テナントが購入、構築、保守するよりも安価なソフトウェアにアクセスできるため、両者にとって有効です。ISV として、サブスクリプションに対して、ソフトウェアの作成と保守にかかるコストよりも多くの料金を請求できます。問題は、ビジネスを複数のテナントにスケールする方法です。

マルチテナンシーはISVsに重要な経済的および運用上の利点を提供します。マルチテナントアーキテクチャにより、組織は投資収益率 (ROI) が向上します。マルチテナンシーは運用要件を簡素化し、組織がより迅速に移行し、テナントにソフトウェアを配信するコストを削減できるようにします。

このドキュメントでは、Amazon Neptune を使用してマルチテナント ISV アプリケーションを効果的に実行するためのガイダンスを提供します。このガイダンスは、ISVs による顧客への SaaS ソリューションの正常な提供を長年にわたってサポートしてきたベストプラクティスに基づいています。組織の目標とアーキテクチャ原則に照らしてこのガイダンスを評価することは、ソリューションを最適化する方法を見つけるのに役立ちます。

> **Note**
>
> このドキュメントでは、ベストプラクティスを網羅したリストは提供していません。マルチテナンシー ISV ワークロードに関する追加の具体的なガイダンスを提供することで、ドキュメントAmazon Neptune の AWS Well-Architected フレームワークの適用」を補足します。ソ

リューションを設計するときは、両方のドキュメントの考慮事項を確認することをお勧めします。

SaaS データパーティショニングモデル

SaaS デベロッパーにとっての課題の 1 つは、マルチテナント環境でデータを表現および整理するためのアーキテクチャパターンを設計することです。これらのマルチテナントストレージのメカニズムとパターンは、通常、データパーティショニング と呼ばれます。

マルチテナント SaaS 環境では、データパーティショニングとテナント分離を区別することが重要です。これらの概念は関連していますが、同義語ではありません。データパーティショニングとは、各テナントのデータを保存する方法を指します。ただし、パーティショニングだけではテナントの分離が保証されません。あるテナントのデータが別のテナントにアクセスできないようにするには、追加の対策が必要です。

マルチテナント SaaS システムの 3 つの一般的なデータパーティショニングモデルは、サイロ、プール、ハイブリッドです。どのモデルを選択するかは、次のような要因によって異なります。

- コンプライアンス
- ノイズの多い隣人
- 階層化戦略
- 運用要件
- テナント分離のニーズ

さらに、 で使用できる各データベースタイプには、 AWS 通常、データパーティショニングとテナント分離モデルの一意のコレクションが用意されています。ソリューションのさまざまなニーズをサポートするためにテナントグラフを整理する方法を検討するときは、Amazon Neptune が提供するモデルを検討してください。

多くの場合、次のいずれかのアサーションを使用して Neptune で設計ISVsを開始します。

- このISVソリューションでは、別々のクラスター間で顧客を物理的に分離する必要があります。
- このISVソリューションには、従来のリレーショナルデータベース管理システムにある名前付きデータベースやスキーマなどの構築が必要です。

検討後、これらのアサーションは正しくないISVsことを認識してください。これは、ほぼすべてのワークロードで、各顧客がデータベースに切断されたグラフを持っているためです。このドキュメントで説明されているデータモデリングとアクセスガイダンスを実装すると、これらのデータ境界が交差し、顧客のデータプライバシーが維持されます。

このガイドでは、サイロモデル とプールモデル の両方について説明しますが、ほとんどの場合、コストと運用効率のためにプールモデルISVsを選択します。このガイドでは、サイロモデルとプールモデルの両方の側面を組み合わせたハイブリッドモデルについて簡単に説明します。グラフサイズの規制またはコンプライアンス要件に対応するために、最大の顧客向けにハイブリッドモデルISVsを使用する人もいます。

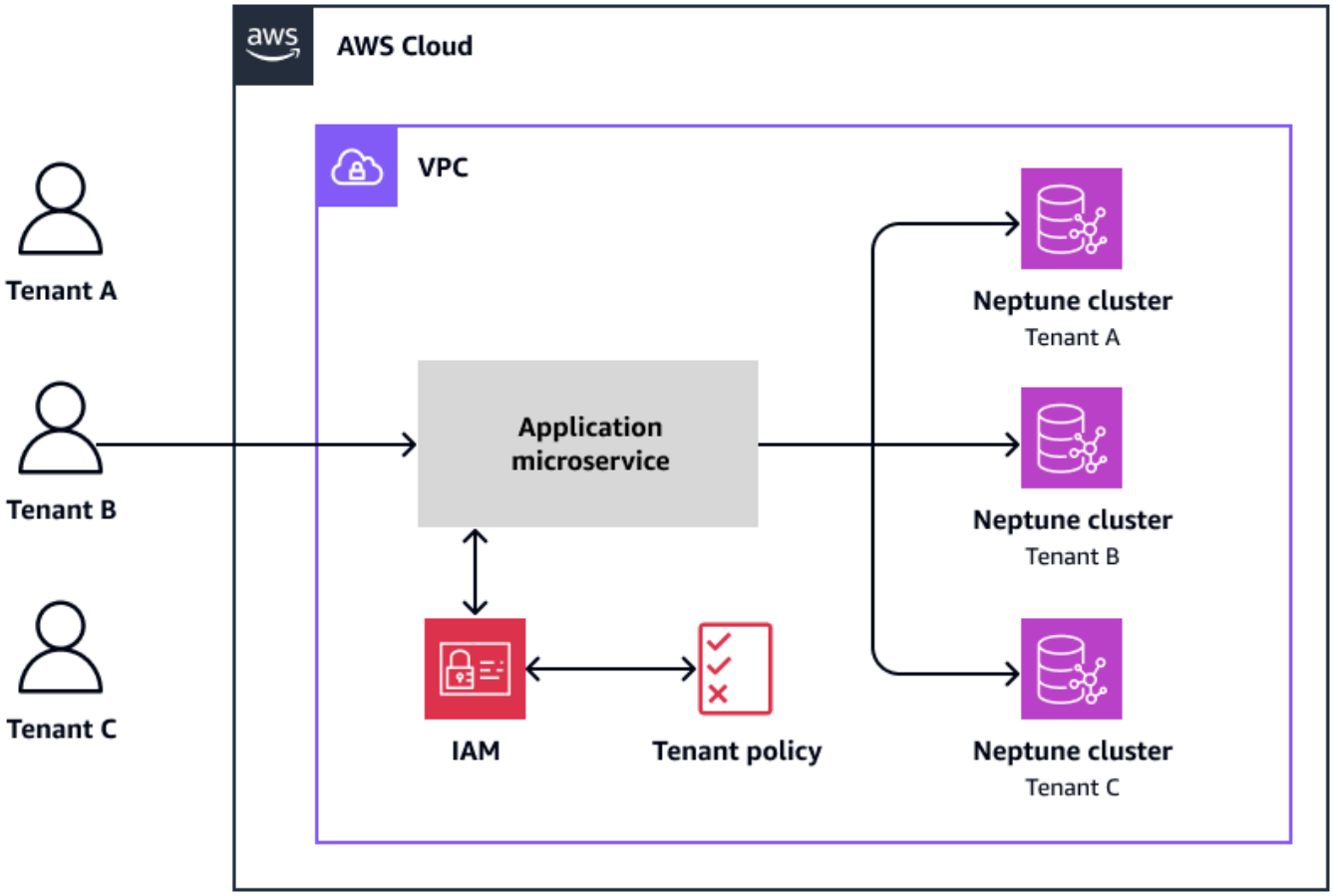
サイロモデルマルチテナンシー

一部のマルチテナント SaaS 環境では、コンプライアンスと規制要件により、テナントのデータを完全に分離されたリソースにデプロイする必要がある場合があります。場合によっては、ノイズの多い近隣への影響を軽減するために、大規模なお客様専用のクラスターが必要です。このような状況では、サイロモデルを適用できます。

サイロモデルでは、テナントデータのストレージは他のテナントデータから完全に分離されます。テナントのデータを表すために使用されるすべてのコンストラクトは、そのクライアントに物理的に一意であると見なされます。つまり、各テナントには通常、個別のストレージ、モニタリング、管理があります。各テナントには、暗号化用の個別の AWS Key Management Service (AWS KMS) キーもあります。Amazon Neptune では、サイロはテナントごとに 1 つのクラスターです。

テナントあたりのクラスター

クラスターごとに 1 つのテナントを持つことで、Neptune でサイロモデルを実装できます。次の図は、仮想プライベートクラウド (VPC) 内のアプリケーションマイクロサービスにアクセスする 3 つのテナントと、テナントごとに個別のクラスターを示しています。



各クラスターには、効率的なデータインタラクションと管理のために個別のアクセスポイントを確保するのに役立つ個別のエンドポイントがあります。各テナントを独自のクラスターに配置することで、テナント間に明確に定義された境界を作成し、データが他のテナントのデータから正常に分離されるようにします。この分離は、厳格な規制とセキュリティの制約がある SaaS ソリューションにも魅力的です。さらに、各テナントに独自のクラスターがある場合、ノイズの多い近隣について心配する必要はありません。1 つのテナントが負荷を課し、他のテナントのエクスペリエンスに悪影響を及ぼす可能性があります。

cluster-per-tenantサイロモデルには利点がありますが、管理と俊敏性の課題も生じます。このモデルの分散性により、テナントアクティビティとすべてのテナントの運用状態を集約して評価することが困難になります。新しいテナントをセットアップするには別のクラスターのプロビジョニングが必要になるようになったため、デプロイも難しくなります。クライアントのアップグレードとバージョンがデータベースのアップグレードと緊密に結合されている場合、クライアントレイヤーが共有されている環境ではアップグレードがより困難になります。

Neptune は、サーバーレスクラスターとプロビジョニングされたクラスターの両方をサポートしています。アプリケーションワークロードがサーバーレスインスタンスまたはプロビジョニングされ

たインスタンスによってより適切に処理されているかどうかを評価します。一般的に、ワークロードの需要が一定である場合、プロビジョニングされたインスタンスはコスト効率が高くなります。サーバーレスは、要求の厳しい可変性の高いワークロードに最適化されており、データベースの使用量が短時間多くなると、軽いアクティビティが長時間続くか、アクティビティがない状態になります。

テナントごとに Neptune でプロビジョニングされたクラスターを使用する場合は、テナントの需要の最大負荷に近似するインスタンスサイズを選択する必要があります。このサーバーへの依存は、SaaS 環境のスケーリング効率とコストにもカスケード的に影響します。SaaS の目標は実際のテナント負荷に基づいて動的にサイズ設定することですが、Neptune プロビジョニングクラスターでは、使用量の多さと負荷の急増を考慮して過剰プロビジョニングする必要があります。オーバープロビジョニングにより、テナントあたりのコストが増加します。さらに、テナントの使用状況が時間の経過とともに変化するにつれて、クラスターのスケールアップまたはスケールダウンをテナントごとに個別に適用する必要があります。

Neptune チームは通常、アイドル状態のリソースによって発生するコストが高くなり、運用が複雑になるため、サイロモデルに対してアドバイスします。ただし、規制の厳しいワークロードや機密性の高いワークロードでは、この追加の分離が必要になるため、お客様は追加料金を支払うことができます。

サイロモデルの実装ガイダンス

テナントcluster-per-tenantサイロ分離モデルを実装するには、 AWS Identity and Access Management (IAM) データアクセスポリシーを作成します。これらのポリシーは、テナントが独自のデータを含む Neptune クラスターにのみアクセスできるようにすることで、テナントの Neptune クラスターへのアクセスを制御します。各テナントの IAM ポリシーを IAM ロールにアタッチします。次に、アプリケーションマイクロサービスは IAM ロールを使用して、 AWS Security Token Service () の `AssumeRole`メソッドを使用してきめ細かな一時的な認証情報を生成しますAWS STS。これらの認証情報は、そのテナントの Neptune クラスターにのみアクセスでき、テナントの Neptune クラスターへの接続に使用されます。

次のコードスニペットは、データベースの IAM ポリシーのサンプルを示しています。

```
{
  "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"
        }
      }
    }
  ]
}
```

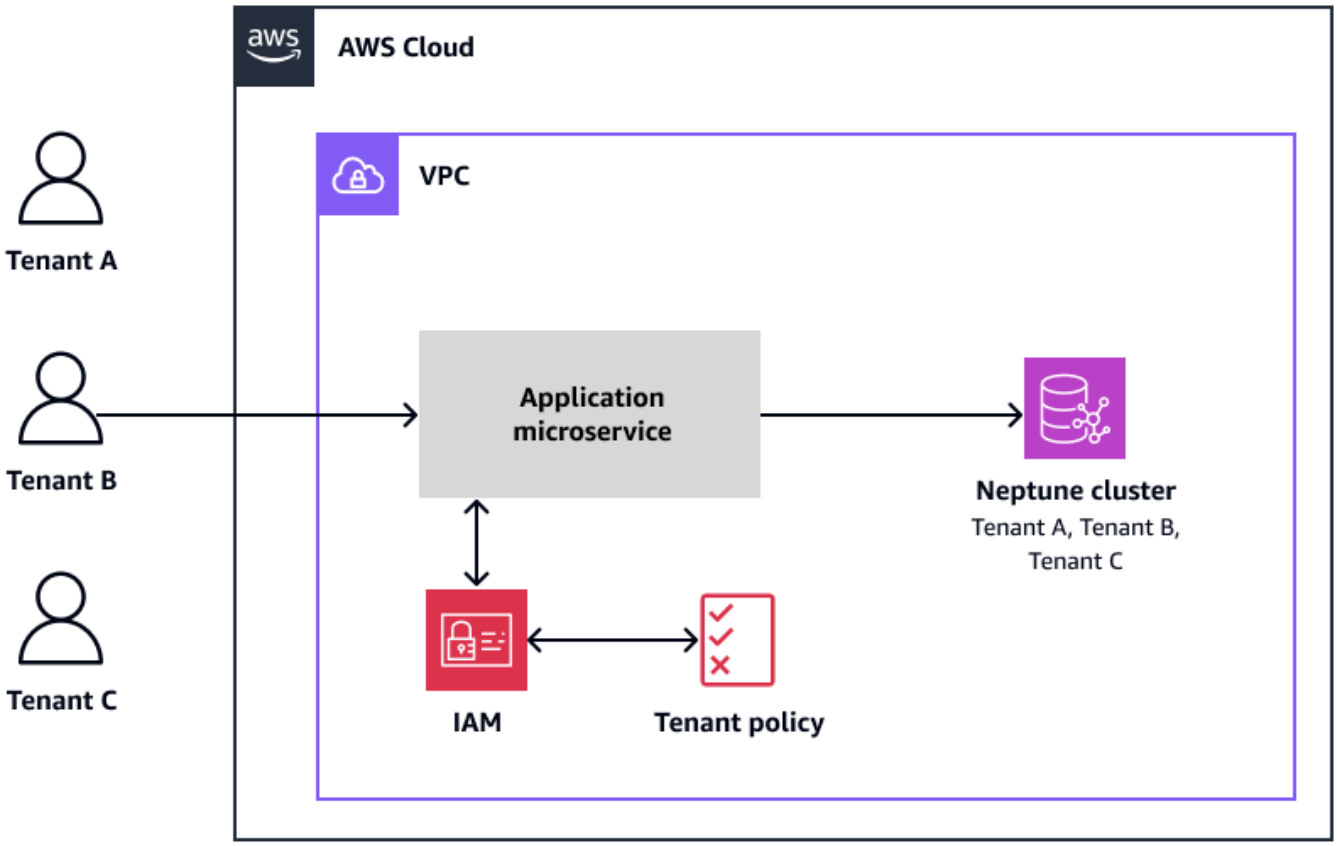
このコードは、サンプルテナント にtenant-1、それぞれの Neptune クラスターへの読み取りおよび書き込みクエリアクセスを提供します。Condition 要素により、IAM ロール () tentant-1 を引き受けた呼び出し元のエンティティ (プリンシパルtenant-role-1) のみが tenant-1の Neptune クラスターにアクセスできるようになります。

プールモデルマルチテナンシー

コストや運用上のオーバーヘッドにより、サイロモデルを実装する必要や実現不可能な場合があります。

- テナントごとに個々のクラスターを維持するリソースがない可能性があります。
- 各テナントのデータを物理的に分離する必要はなく、論理的に分離するだけでニーズとコンプライアンス要件を満たすことができます。

次の図は、テナントデータが単一の Amazon Neptune クラスターに配置され、すべてのテナントが共通のデータベースを共有するプールモデルを示しています。



このプール分離モデルは、管理するクラスターが少ないため、管理オーバーヘッドを削減し、運用効率を向上させることができます。また、コンピューティングリソースは、顧客の非アクティブ期間中もアイドル状態のままではなく、複数の顧客間で共有できます。

プールモデルを使用する場合、データをモデル化する方法は 2 つあります。アプローチは、ラベル付きプロパティグラフ (LPG) を構築するか、リソース記述フレームワーク (RDF) を使用してグラフを構築するかによって異なります。

ラベル付きプロパティグラフのプールモデル

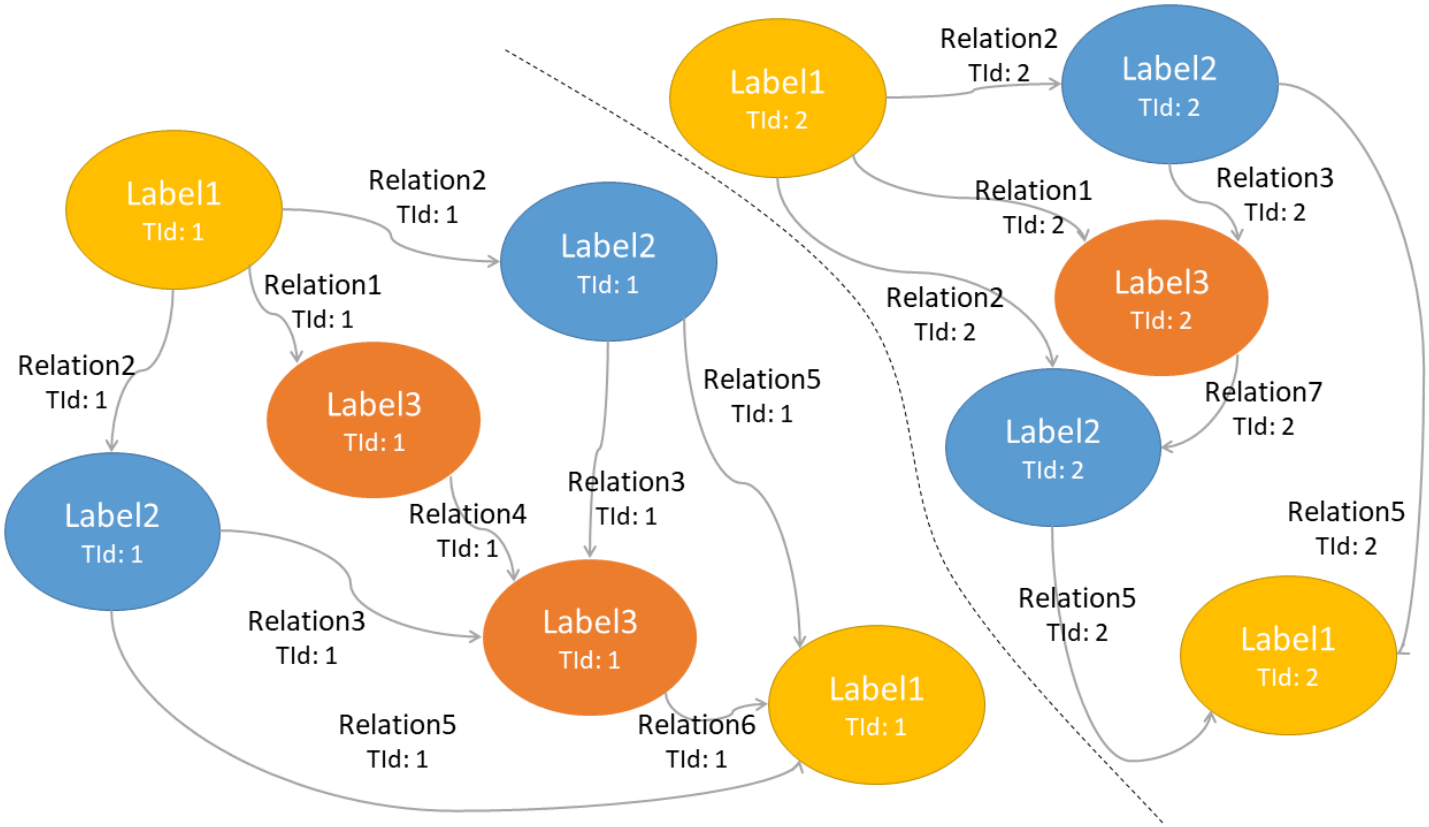
Amazon Neptune の LPGsには 3 つの異なるアプローチがあります。

- プロパティ戦略 - Apache TinkerPop Gremlin 言語の PartitionStrategy など、確立されたライブラリコンストラクトの使用をパフォーマンスよりも優先する必要がある場合は、プロパティ戦略を選択します。
- プレフィックスラベル戦略 – パフォーマンスとノイズの多い近隣効果の制限に基づいて、ほとんどのシナリオでプレフィックスラベル戦略をお勧めします。
- マルチラベル戦略 – マルチラベル戦略では、プレフィックスラベル戦略のパフォーマンスが向上しています。また、クラスター上のすべてのテナントにまたがるクエリの実行もサポートされています (たとえば、すべてのテナントでレポートまたはモニタリングするための ISV クエリ)。

プロパティ戦略

LPGs、ユーザーはキーと値のペアのプロパティをノード、頂点、エッジに追加できます。論理的な分離を実現するために、ほとんどのお客様は、これをすべてのノードと エッジで共通のテナントプロパティキーを持つ一意のプロパティとして直感的にモデル化します。テナントプロパティキーは、ノードを所有するすべてのテナントを表します。テナント識別子は、個々のテナントを識別する一意の値です。

次の図は、このモデルを示しています。切断された 2 つのサブグラフには、さまざまなラベル付きノードとエッジがあり、テナントプロパティキーは で表されますTId。1 つのサブグラフのすべてのノードとエッジのTId値は です1。もう 1 つのサブグラフでは、すべてのノードとエッジのTId値はです2。



ラベル付きプロパティグラフ内では、これを管理する方法は 2 つあります。Gremlin クエリ言語は、 データのデータパーティショニングの管理に役立つ PartitionStrategy トラバーサルライブラリを提供します。次の例のコードでは、すべてのノードとエッジに というプロパティがあることを想定していますTId。

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

新しいノードまたはエッジが書き込まれると、 プロパティ"TId"は、 "1" または のどちらが選択されたかに応じて"2"、 strategy1または の値で追加strategy2されます。が "TId"のお客様は"1"、 を使用しますstrategy1。次の例は、その顧客のデータの書き込みを示しています。

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

読み取りクエリの場合、 "TId == '1'" または のフィルタ"TId == '2'"はstrategy2、それぞれ strategy1または を使用してすべてのノードまたはエッジトラバーサルに追加されます。これら

のパーティション戦略はコードを簡素化しますが、必須ではありません。戦略を使用する利点は、認可レベルで挿入し、クエリを形成する下位レベルのコードに渡すことができることです。これにより、顧客識別子 (TId) を決定するコードとクエリのロジックが分離されます。

次のコード例は、データを読み取るための Gremlin クエリを示しています。

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

上記のコードは、次の例と同等です。

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

同様に、Gremlin を使用してデータを書き込む場合は、次のクエリを使用できます。

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

上記のコードは、パーティション戦略を使用しないため、 "TId"プロパティを明示的に記述する必要がある次の例と同等です。

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

openCypher では、これらのライブラリは存在しません。テナント識別子をノードとエッジのプロパティとして追加するには、クエリを記述して変更する必要があります。例えば、次のようになります。

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

パーティション戦略を使用しない Gremlin コード間の類似性に注意してください。その後、次のコードを使用して、最初のCREATEステートメントから書き込まれたノードを読み取ることができます。

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

PartitionStrategy などのネイティブ TinkerPop Gremlin コンストラクトを使用する場合は、プロパティ戦略を選択できます。ただし、このモデルには、プレフィックスラベル戦略と比較して、Amazon Neptune のパフォーマンス上の欠点があります。これらのパフォーマンスの欠点については、「LPG モデルのパフォーマンスへの影響」セクションを参照してください。

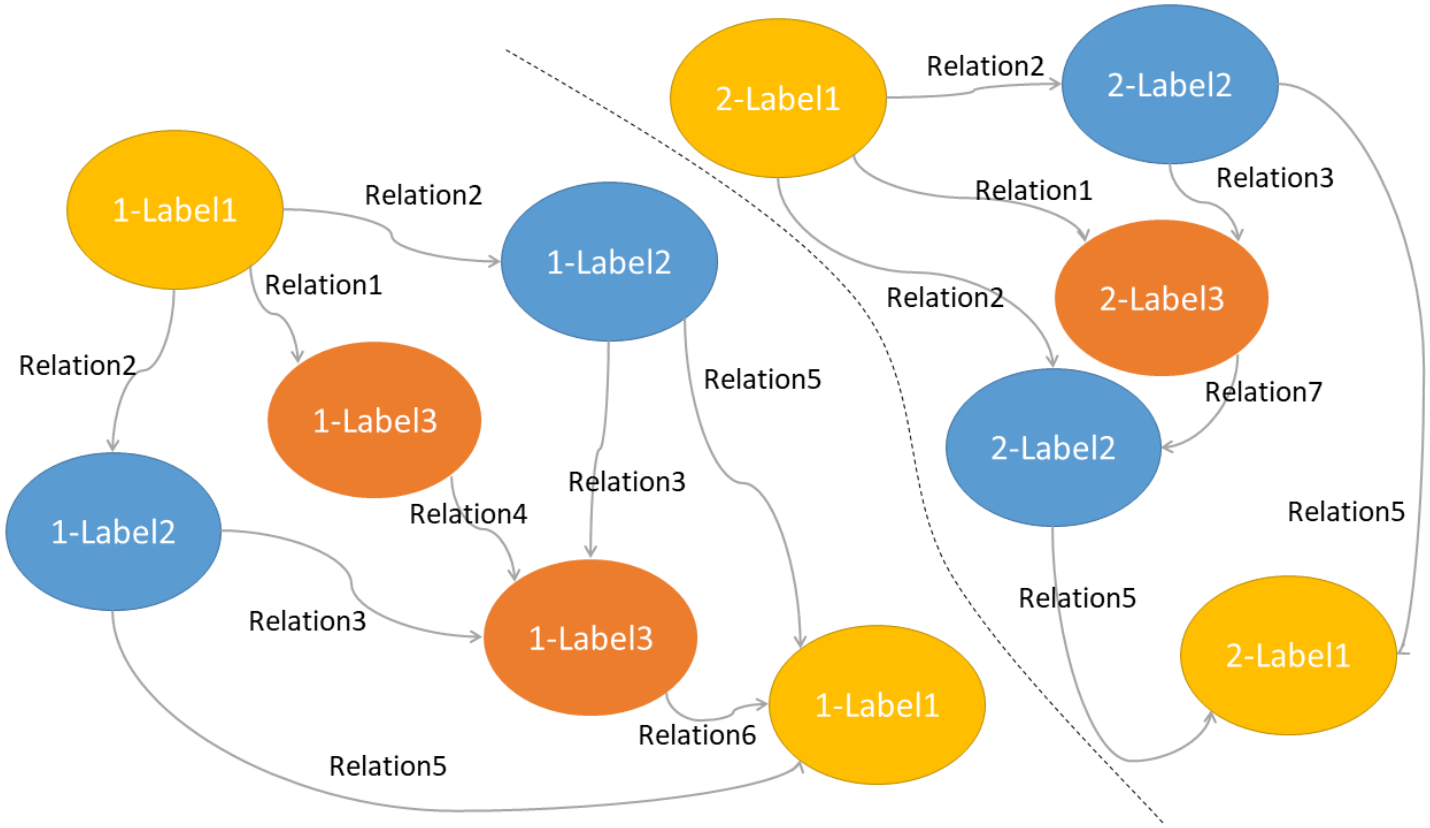
以下の条件が適用される場合は、プロパティ戦略をエッジではなくノードでのみモデリングすることを検討してください。

- グラフのエッジはラベルよりも大幅に多くなります。
- 各テナントは切断されたグラフです。
- グラフにアクセスするには、ラベルではなくノードを開始点として使用します。

プレフィックスラベル戦略

パフォーマンスが最大の懸念事項である場合は、プロパティ戦略よりもプレフィックスラベル戦略を検討することを強くお勧めします。

プレフィックスラベル戦略では、各ノードにテナント識別子とノードラベルの組み合わせでラベル付けします。たとえば、テナントの識別子が "1"で、ノードラベルが の場合"Label1"、ノードラベルを として指定します"1-Label1"。次の図は、このモデルを使用する 2 つの切断されたサブグラフを示しています。



Gremlin でデータを書き込むときは、任意のノードのラベルに識別番号を追加できます。

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

このグラフをクエリするときは、ノードにこのプレフィックスが存在するかどうかを確認できます。

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

openCypher では、 CREATEステートメントを使用してデータを書き込むことができます。

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

openCypher で書き込んだデータをクエリするには、次のコードを使用します。

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

プレフィックスラベル戦略では、すべてのノードが 1 つ以上のテナントに割り当てられ、アクセス許可がエッジスコープに割り当てられていないことを前提としています。多数の述語を引き起こ

し、Neptune のパフォーマンスに悪影響を及ぼすため、エッジラベルにこの戦略を使用しないでください。

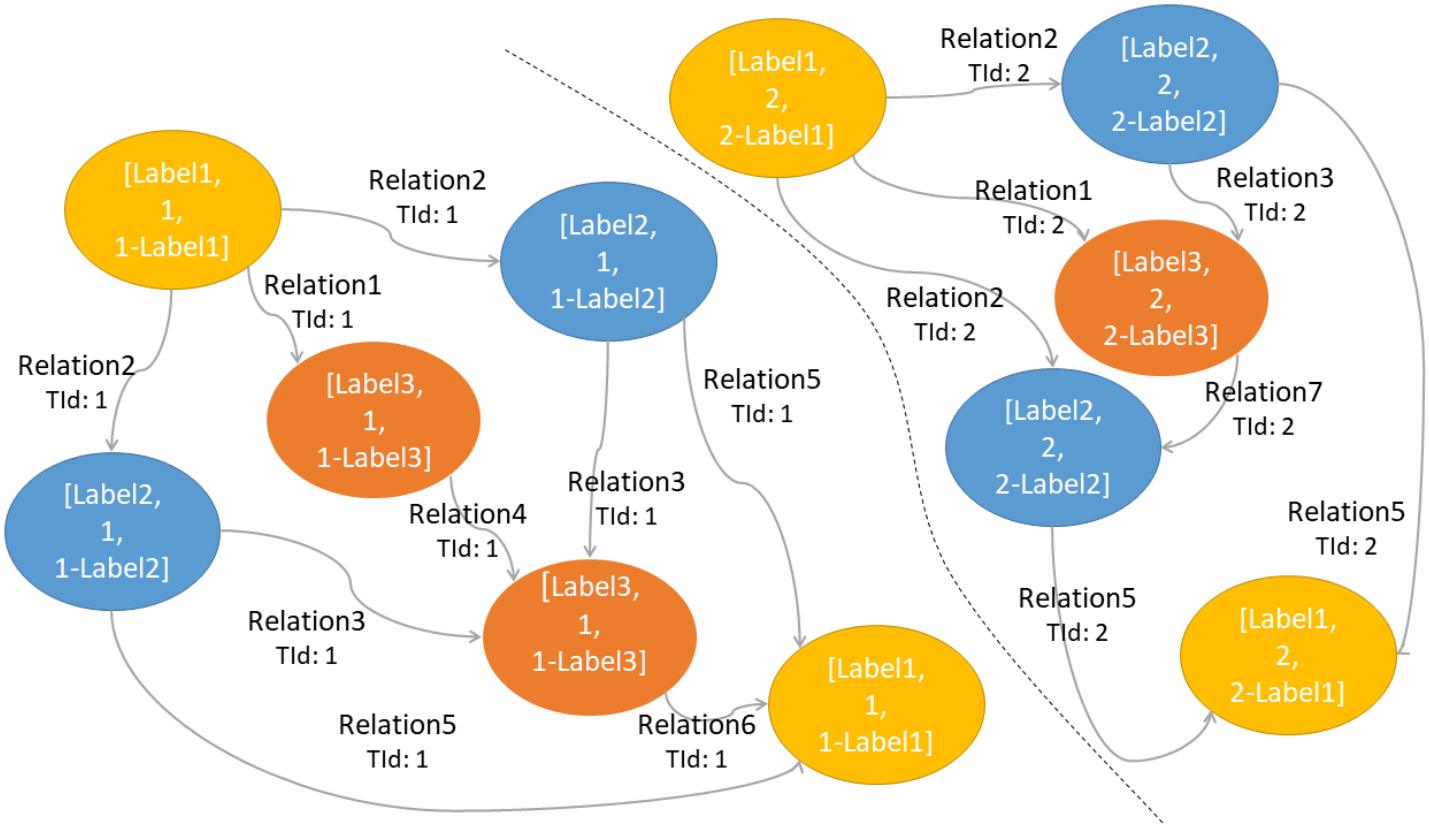
プレフィックスラベルアプローチには 2 つの主な欠点があります。まず、テナント間でクエリを実行することは困難です。一例は、レポートまたはモニタリングのために特定のラベルのすべてのノードをカウントするクエリです。これがユースケースである場合は、この戦略を複数ラベル戦略と組み合わせることを検討してください。戦略の組み合わせの詳細については、「ハイブリッドモデル」セクションを参照してください。

次に、プレフィックスラベル戦略では、データ漏洩を防ぐために、すべてのクエリに適切なプレフィックスを適切に適用するコントロールが必要です。ただし、この戦略は低レイテンシーのクエリを必要とするワークロードにとって最も効率的なオプションであり、強くお勧めします。LPG モデルのパフォーマンスへの影響セクションでは、これが最も効率的な戦略である理由の例を示します。

マルチラベル戦略

3 番目のオプションは、複数ラベル戦略を使用することです。このアプローチでは、グラフ上のすべてのノードにラベルを追加します。たとえば、特定のテナントのすべてのデータをフィルタリングする必要がある場合は、テナント ID ラベルを追加します。テナントに関係なく、特定のラベルのすべてのデータをフィルタリングする必要がある場合は、そのラベルを追加します。次の図は、ノードごとに 3 つのラベルを使用して適用される複数ラベル戦略を示しています。

次の 3 つの異なるパターンを使用してグラフにアクセスできるようになりました。



- をフィルタリングLabel1して、すべてのテナントLabel1で を持つすべてのノードを返します。
- でフィルタリング1して、テナント 1 のすべてのノードを返します。
- をフィルタリング1-Label1して、ラベル を持つテナント 1 のみのすべてのノードを返しますLabel1。

LPGs、これを実装する方法は 2 つあります。

Gremlin では、SubgraphStrategy というトラバーサル戦略を使用して、すべてのクエリの範囲を などの特定のラベルを持つ頂点のみに制限できます"Label1"。

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

PartitionStrategy とは異なり、SubgraphStrategy はデータの読み取りにのみ影響し、データの書き込みには影響しません。データを書き込むには、各クエリにラベルを手動で割り当てます。

```
g.addV("Label1").property("Value","XYZ123456")
.addV("Label2").property("Value","XYZ123456")
```

データを読み取るときは、SubgraphStrategy を使用して ですべてのノードをクエリできます"Label1"。

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

Neptune は、 "Label1"と の値を持つ最初のレコードのみを返します"XYZ123456"。これは、SubgraphStrategy を使用しない次のクエリに相当します。

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

この基本的なクエリでは、SubgraphStrategy の使用がより複雑であるように見えます。ライブラリは、既に定義された戦略gで のインスタンスを提供できることに注意してください。開発者は、適切なフィルターが適用されていることを確認する必要はありません。

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

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

openCypher ライブラリにはこれらのコンストラクトがないため、ノードごとに複数のラベルを作成する必要があります。

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

これらのラベルを使用してサブグラフをフィルタリングする場合、探している顧客ラベルを持つノード、またはそのラベルを持つ別のノードとの関係を共有するノードを返すことができます。

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

マルチラベル戦略を使用すると、ノードをタイプ (Label1) またはテナント () でクエリしたり、パフォーマンスが最も重要である場合により効率的なプレフィックスラベル戦略 (1) を使用したりできます1-Label1。

この戦略の主な欠点は、各ラベルがグラフに保存されている追加のオブジェクトであることです。オブジェクトは、LPGs。取り込み速度は 1 秒あたりのオブジェクトによって測定され、バインドされます。ストレージコストは消費されたギガバイト数によって異なります。つまり、追加のオブジェクトは大規模な測定可能な影響を与える可能性があります。

LPG モデルのパフォーマンスへの影響

Amazon Neptune AWS のスキルビルダーコースのデータモデリングでは、Neptune データモデルの内部とモデリングの影響について詳しく説明しますが、これらの設計に関する重要な考慮事項をここで要約します。 Amazon Neptune 1 つの Neptune クラスターに 3 つのテナント (T1、T2、T3) を持つことを検討してください。これらのテナントには次の属性があります。

- テナント 1 (T1) には合計 1 億のノードがあり、1,000 万のノードは Item 型です。
- テナント 2 (T2) には合計 1,000 万ノードがあり、100 万ノードは Item 型です。
- Tenant 3 (T3) には合計 1 億のノードがあり、100 万のノードは Item 型です。

プロパティ戦略を使用して、テナント 3 の項目を取得するクエリを実行します。Neptune は、2 つのインデックス呼び出しの統計を検査します。

- `tenant property key=T3` に 1 億件の結果がある
- `label = Item` に 1,200 万件の結果がある (T1 から T1,000 万件 + T2 から 100 万件 + T3 から 100 万件)

Neptune クエリオプティマイザは、後者のクエリが最初に適用されるのが最適であると判断し (1,200 万の結果) 、各項目で を検査しますtenant property key=T3。1,200 万件の項目を取得して、100 万件の結果を見つけます。

このクエリのノイズの多い近隣の影響に注目してください。テナントあたり 1 億のアイテムノードがある場合、最初のクエリの結果は 1,200 万ではなく 3 億になります (これは説明のために過度に単純化されています。 Neptune オプティマイザは、異なる順序のオペレーションを適用した可能性があります) 。

次に、プレフィックスラベル戦略を検討します。100 万件の結果を 返す で label=T3-Item1 回のインデックス呼び出しを行います。これにより、プロパティ戦略と同じ結果が得られますが、取得す

るレコードは 1,100 万件少なくなります。さらに、ラベルがインデックス内で重複しないため、ノイズの多い近隣の懸念がなくなりました。

マルチラベル戦略では、プロパティ戦略よりもクエリパフォーマンスが直接向上することはありません。プロパティ値によるフィルタリングは、検索スペースが比較可能な場合、ラベル値によるフィルタリングと同等です。代わりに、マルチラベル戦略はより柔軟性をサポートします。 マルチラベル戦略は、 `label=T3`または ラベル のプレフィックスラベル戦略と同等のパフォーマンスを提供しますT3-Item。マルチラベル戦略は、 のプロパティ戦略と同等のパフォーマンスを提供しますlabel=Item。利点は、さまざまなアクセスパターンをサポートすることです。

RDF のプールモデル

Resource Description Framework (RDF) には、名前付きグラフの概念があり、データを論理的に分離する方法を提供します。Amazon Neptune には、デフォルトの名前付きグラフとユーザー定義の名前付きグラフがあります。名前付きグラフは必要な数だけ作成できます。総称して、RDF データセットと呼ばれます。デフォルトまたはユーザー定義のすべての名前付きグラフは、RDF データセット内の国際化リソース識別子 (IRI) によって定義されます。Neptune では、ユーザーがデータの書き込み時に名前付きグラフを宣言しない限り、すべてのトリプルはデフォルトの名前付きグラフの一部と見なされます。

名前付きグラフには複数のユースケースがあります。

- データパーティショニングとデータ分離
- データ出典
- バージョニング
- 推測

このガイドでは、データパーティショニングのユースケースに焦点を当てています。テナントごとに 1 つのユーザー定義の名前付きグラフを作成することをお勧めします。

Graph Store HTTP Protocol を使用した SPARQL クエリオプション

次のクエリ例では、SPARQL プロトコルと RDF クエリ言語 (SPARQL) と Graph Store HTTP プロトコルを使用して、テナントの名前付きグラフをクエリまたは作成します。

- `HTTP GET` – テナントの特定のグラフを取得するには:

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

- HTTP PUT – 特定の名前付きグラフを作成またはリクエストで指定されたペイロードに置き換えるには:

```
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'
```

  RDF では、オブジェクトはトリプルです。

- HTTP POST – 名前付きグラフが存在しない場合は新しいグラフを作成するか、既存のグラフとマージするには:

```
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'
```

RDF のテナント分離

アプリケーションレイヤーに必要なガードレールがあるデータを論理的に分離するには、テナントとユーザー定義の名前付きグラフ間のマッピングを作成します。RDF データセットのマルチテナンシーを設計する場合は、RDF と SPARQL の次の側面に注意してください。

- Neptune では、名前付きグラフを指定せずにクエリを実行すると、データベース内のすべての名前付きグラフでパターンに一致するすべてのトリプルを取得します。
- RDF では、異なる名前付きグラフのノード間の接続に制約はありません。たとえば、前の図では、 のノードを エッジを介して G2のノードに接続:G1できます。

たとえば、特定のテナントのエンドユーザーが API にクエリを送信する場合、API は Neptune データベースにクエリを送信する前に次の要件を検証する必要があります。

- 単一のテナントを対象とするクエリでは、名前付きグラフを指定する必要があります。そうしないと、テナント間でデータが漏洩するリスクがあります。

- 更新または削除クエリでは、常に名前付きグラフを指定する必要があります。
- エッジまたはリレーションシップの両側にあるノードは、常に正しい名前付きグラフに属している必要があります。

ベストプラクティスの詳細については、Neptune ドキュメントを参照してください。

成長に備える

プールモデルを正常に使用すると、最終的に単一の Neptune クラスターのサイズが大きくなります。テナントの増加、またはテナントの数の増加、すべての顧客に必要なデータの取り込み率がクラスターの機能を超えています。この場合、顧客を複数のクラスターに分割する必要があります。後で改良を試みるのではなく、この設定を事前に設計してください。初期スケールが 1 つのクラスターのみを使用する場合でも、そのスケールに達したときに将来複数のクラスターにテナントをルーティングする必要があるコンポーネントをモックアップします。

テナントのサイズに基づいてソリューションにより多くのリソースが必要な場合は、その成長に備えてください。1 つのクラスターの複数のお客様が大幅に増加した場合、そのクラスターは要件をサポートしなくなる可能性があります。Amazon Neptune DB クローン作成機能を使用して、テナントを別のクラスターに移動するか、既存のクラスターを 2 つに分割する戦略を設計します。

DB クローニングを実装するときにコストを削減できる Neptune Copy-on-Write Protocol に精通してください。取り込みのボトルネックのためにクラスターを分割すると、ポリシーで許可されている限り、クラスターからデータを削除しない方が効率的です。2 つのクラスターは、変更されていない場合はデータページを共有しますが、データページが変更されている場合は共有しません (データページの一部が削除されたためです)。

> **Note**
>
> このガイダンスは、この執筆時の最新の Neptune バージョンである Neptune バージョン 1.3.1 に適用されます。このガイダンスは、Neptune ストレージレイヤーが進化するにつれて、将来のバージョンで変更される可能性があります。

マルチテナンシーシナリオの制限事項

一部の Neptune 機能は、マルチテナンシーシナリオ用に構築されていないことに注意してください。これらのマルチテナンシー戦略はデータベースレベルで強制されないため、テナントにはプール

モデル内の Neptune エンドポイントへの直接アクセスを許可しないでください。このドキュメントで説明されている設計を適用する Neptune エンドポイントと顧客の間には、常に何らかのプロキシを保持してください。このようなプロキシの例は次のとおりです。

- クライアントレイヤーにラベルフィルターを追加する
- 認証トークンをテナント ID にマッピングし、このフィルターをクエリに挿入する API がある

このガイダンスは、Neptune グラフノートブック、Neptune グラフエクスプローラー、Neptune ストリームなどの機能に直接アクセスできるようにする場合にも適用されます。

ハイブリッドモデルマルチテナンシー

SaaS ソリューションは、多くの場合、サイロモデルとプールモデルの組み合わせを使用します。さまざまな要因が、同じ環境内でサイロモデルとプールモデルの両方をいつどのように採用するかの決定に影響を与えます。

このような要因の 1 つは階層化です。SaaS SaaS ソリューションは、テナントの各階層に独自のエクスペリエンスを提供します。例えば、階層が無料、スタンダード、プレミアムの場合、無料階層のテナントデータはプールモデルを使用して共有 Neptune クラスターに保存できます。スタンダードおよびプレミアム階層テナントでは、 cluster-per-tenant サイロモデルを使用できます。

さらに、一部の SaaS プロバイダーには、基盤として共有 Amazon Neptune クラスターにプールソリューションを構築できる機能があります。その後、多くの場合、コンプライアンスと規制の義務により、サイロ化されたストレージを必要とするテナント用に個別の Neptune クラスターを作成できます。

これにより、データアクセスレイヤーと管理プロファイルにある程度の複雑さが加わる可能性がありますが、お客様の要件を満たすために、提供サービスを階層化する方法をビジネスに提供することもできます。

ISVsの運用上のベストプラクティス

このセクションのガイドラインの多くは、すべてのお客様向けのベストプラクティスですが、ISVsには重要性が付加されています。

Neptune クラスターを最新バージョンで更新する

Amazon Neptune リリースノートでは、すべてのバージョンが多数のバグ修正、パフォーマンスの向上、新機能をもたらしていることがわかります。Neptune クラスターをできるだけ最新バージョンに保ちます。

以前に検出されなかったバグがワークロードにあり、クラスターが最新バージョンである場合、Neptune エンジニアはクラスターのプライベートパッチを作成できます (必要に応じて) 。パッチは、その修正が一般公開される次のリリースまでブリッジできます。クラスターを最新バージョンに更新するには、Neptune Blue/Green ソリューションを使用します。

データインジェストに削除と置換の代わりに差分を使用する

Neptune にデータを取り込んだり書き込んだりするには、いくつかの手法を使用できます。多くのお客様は、フィードで変更が受信されるたびにグラフを削除して再挿入することで、データの取り込みを簡素化しようとしています。各ノードに`last-modified`プロパティを追加し、指定した日付以降に変更されていないノードを定期的にスキャンして削除する場合があります。これらの手法はデータ取り込みプロセスを簡素化しますが、Neptune クラスターには長期的なヘルスとスケーラビリティの影響があります。

まず、Neptune は文字列のディクショナリエンコーディングを使用します。ノードとエッジの IDs を明示的に指定しない限り、Neptune は ID の文字列として表される GUID を生成し、その文字列をディクショナリに保存します。オブジェクトを常に削除して追加している場合、自動的に生成された IDs によってディクショナリが肥大化します。

次に、Neptune は最大 1 秒あたり約 120 K オブジェクトを取り込むようにスケールアップします。オブジェクトを継続的に削除して追加する場合、基本的に変更されていないオブジェクトでその帯域幅の多くを消費します。これにより、クラスターでホストできるテナントの数が制限され、クラスター内により大きなライターインスタンスが必要になり、より多くの I/O オペレーションが必要になります。これらの要因はすべてコストを増加させます。

削除および追加メソッドを使用する代わりに、変更内容の真の差分を計算する方法を開発することを強くお勧めします。ただし、一部のデータソースはこれには適していません (たとえば、現在の状

態を返す API コールや、変更された内容を正確に追跡しないイベントなど) 。raw データソースが変更の特定に適していない場合は、抽出、変換、ロード (ETL) プロセスを使用して差分を計算します。例えば、以前の各データキャプチャのスナップショットを Parquet 形式で維持 AWS Glue し、を使用してそれらのスナップショットの違いを計算し、その違いのみを Neptune にプッシュできます。

Neptune のコストがテナントとともにどのように進化するかをモデル化する

サイロモデル、プールモデル、ハイブリッドモデルのいずれを使用する場合でも、クラウドコストはテナントのサイズに応じてスケールします。同時接続数が多いテナントでは、同時接続数が少ないテナントよりも大きなインスタンスまたはリードレプリカが必要です。より迅速なデータ取り込みを必要とするテナントにも同じことが当てはまります。

Neptune クラスターコストの 3 つのコンポーネントは、インスタンスサイズ (および数) 、データサイズ (GB/月) 、I/O オペレーション (100 万あたり) です。これらのコストは一般的にワークロード固有ですが、サイズとデータ量に応じてスケールしますが、 AWS ツールを使用して測定できます。時間の経過とともにサイズがどのように変化するかなど、テナントのサイズの主要な指標に照らしてスケールの経済を追跡し、理解します。I/O 料金の予測不能性がマージンに影響する場合は、より予測可能なコストで Neptune I/O 最適化ストレージを選択することを検討してください。

お客様の需要に合わせてクラスターをスケールする

Neptune インスタンスサイズを適切にサイズ設定するための試行式や true 式はありません。Neptune ドキュメントにはガイダンスが記載されていますが、変数が多すぎて直接マッピングを推奨できません。これらの変数には以下が含まれますが、これらに限定されません。

- データモデル
- データシェイプ
- クエリの同時実行数
- クエリの複雑さ。

ワークロードとテナントプロファイルに最適なサイズを決定するためのテストを計画します。一般的に、コスト効率と予測可能性のために、プロビジョニングされたインスタンスを使用することをお勧めします。カスタマーエクスペリエンスの目標がコストよりも最適なスケーリングを優先する場合

は、Neptune Serverless インスタンスを使用して、ワークロードの変動に関係なく、より一貫したエクスペリエンスを確保することを検討してください。

テナントの読み取りワークロードのピークとトラフに大きなばらつきがある場合は、Neptune Serverless インスタンスを Neptune 自動スケーリングと組み合わせてください。 通常、新しいリードレプリカが初期化されてからオンラインになるまでに 10～15 分かかります。つまり、自動スケーリングだけではトラフィックの長期的な変化を処理できますが、アクティビティの急激な急増には不十分です。Neptune サーバーレスと Neptune 自動スケーリングを組み合わせることで、インスタンスをスケールアップまたはスケールダウンしたり、リードレプリカの数をスケールインおよびスケールアウトしたりできます。

テナントのワークロードプロファイルまたはサービスレベルアグリーメント (SLAs) が大幅に異なる場合は、カスタムエンドポイントと専用リードレプリカを使用して、そのトラフィックに最適化されたインスタンスにトラフィックを誘導することを検討してください。最適化には、インスタンスの異なるサイズ設定、特定のクエリパターン、バッファキャッシュの事前ウォーミングなどがあります。

次のステップ

マルチテナンシーISVアプリケーションに Amazon Neptune を実装するジャーニーを開始したばかりの場合は、必要なモデルについてさらに考慮してください。モデルを変更すると、ジャーニーの後半でコストが高くなります。

ジャーニーの早い段階では、ニーズに最適なモデルを使用し、そのモデルのガイダンスに従っていることを確認します。

常に余裕を持って計画するようにしてください。ジャーニーの早い段階では、頂点や エッジを削除して再追加するのではなく、クラスター間での顧客のシャーディングやETLプロセスの最適化作業を延期して変更のデルタを提供するのが魅力的です。スケーリングすると、これらの決定がパフォーマンスとコストに悪影響を及ぼす可能性があります。

最後に、すでにジャーニーに慣れている場合、このガイダンスにより、アーキテクチャが最適であることが再確認されたり、アーキテクチャを改善するための変更が加えられたりする可能性があります。

このガイダンスについて質問がある場合や、さらにサポートが必要な場合は、 AWS アカウントチームに連絡して、Neptune スペシャリストとのセッションを依頼してください。

リソース

- Amazon Neptune ドキュメント
- Amazon Neptune のデータモデリング (コース)
- Amazon Neptune 用の AWS Well-Architected フレームワークの適用
- SaaS レンズ Well-Architected フレームワーク
- でのマルチテナントアーキテクチャのガイダンス AWS
- SaaS テナント分離戦略: マルチテナント環境でのリソースの分離
- Apache TinkerPop ドキュメント
- SPARQL

寄稿者

このガイドの寄稿者には以下が含まれます。

- Brian O'Keefe、プリンシパル WW SSA Neptune、 AWS
- Veeresham Gande、シニアテクニカルアカウントマネージャー、 AWS
- Dana Owens、スタートアップソリューションアーキテクト、 AWS
- Nima Seifi、スタートアップソリューションアーキテクト、 AWS

ドキュメント履歴

以下の表は、本ガイドの重要な変更点について説明したものです。今後の更新について通知を受け取る場合は、RSSフィード をサブスクライブできます。

変更	説明	日付
初版発行	—	2024 年 9 月 3 日

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。