[イチから理解するサーバーレスアプリ開発]

サーバーレスアプリケーション向きの DB 設計ベストプラクティス

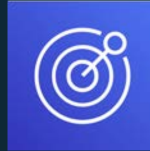
Version 2019-09-05

2019.09.05

Amazon Web Services Japan K.K.
Akihiro Tsukada, Startup Solutions Architect, Manager

何者

- 塚田 朗弘 @akitsukada
- Startup Senior Solutions Architect
- #startup #blockchain
  #dev #mobile **#serverless**
- 好きなサービス：Amazon Pinpoint



アジェンダ

- サーバーレスアプリケーションとデータベースの種類
  （および懸念事項と対策）
- データベース設計プロセス
  ～ RDB と Amazon DynamoDB ～
- かんたんドリル：DynamoDB の設計シミュレーション

※ Amazon DynamoDB の基礎・詳細知識は別資料を参照のこと

- Amazon DynamoDB 開発者ガイド（Link）
  - **DynamoDB のベストプラクティス**（Link）
- Amazon DynamoDB のベストプラクティスに従うという 2019 年の計を立てる - Amazon Web Services ブログ（Link）
- [AWS Black Belt Online Seminar] Amazon DynamoDB Advanced Design Pattern（資料 | 動画）
- DynamoDBデータモデリング虎の巻 - misc.tech.notes （Link）
- AppSyncを使いこなすためのDynamoDB設計パターン（Link）

※ 特にベストプラクティスの詳細についてはこちらを要チェック

Amazon DynamoDB 開発者ガイド（Link）

AWS ドキュメント » Amazon DynamoDB » 開発者ガイド » DynamoDB のベストプラクティス

DynamoDB のベストプラクティス

このセクションでは、Amazon DynamoDB 使用時にパフォーマンスを最大にしてスループットコストを最小にするための推奨事項をすばやく確認することができます。

目次

- DynamoDB に合わせた NoSQL 設計
  - リレーショナルデータ設計と NoSQL の相違点
  - NoSQL 設計の 2 つの重要な概念。
  - NoSQL 設計へのアプローチ
- パーティションキーを効率的に設計し、使用するためのベストプラクティス
  - バーストキャパシティーを効率的に使用する
  - DynamoDB アダプティブキャパシティーを理解する
  - パーティションキーを設計してワークロードを均等に分散する
  - 書き込みシャーディングを使用してワークロードを均等に分散させる
    - ランダムなサフィックスを使用したシャーディング
    - 計算されたサフィックスを使用したシャーディング
  - データアップロード時に書き込みアクティビティを効率的に分散する
- ソートキーを使用してデータを整理するためのベストプラクティス
  - バージョンコントロールにソートキーを使用する
- DynamoDB でセカンダリインデックスを使用するためのベストプラクティス
  - DynamoDB のセカンダリインデックスの一般的なガイドライン
    - インデックスを効率的に使用する
    - 射影を慎重に選択する
    - フェッチを回避するための頻繁なクエリの最適化
    - ローカルセカンダリインデックス作成時に項目コレクションのサイズ制限を確認する
  - スパースなインデックスの利用
    - DynamoDB のスパースインデックスの例
  - マテリアライズされた集計クエリでグローバルセカンダリインデックスを使用する
  - グローバルセカンダリインデックスの多重定義
  - 選択的なテーブルクエリに対してグローバルセカンダリインデックスの書き込みシャーディングを使用する
  - グローバルセカンダリインデックスを使用して結果整合性のあるレプリカを作成する
- 大きな項目と属性を格納するベストプラクティス
  - 大量の属性値を圧縮する
  - 大きな属性値を Amazon S3 に保存する
- DynamoDB で時系列データを処理するベストプラクティス
  - 時系列データの設計パターン
  - 時系列テーブルの例
- 多対多の関係を管理するためのベストプラクティス
  - 隣接関係のリスト設計パターン
  - マテリアライズドグラフパターン
- ハイブリッドなデータベースシステムを実装するためのベストプラクティス
  - すべてを DynamoDB に移行したくない場合は
  - ハイブリッドシステムの実装方法
- DynamoDB でリレーショナルデータをモデル化するためのベストプラクティス
  - DynamoDB でリレーショナルデータをモデル化するための最初のステップ
  - DynamoDB でリレーショナルデータをモデル化する例
- クエリとスキャンデータのベストプラクティス
  - スキャンのパフォーマンスに関する考慮事項
  - 読み込みアクティビティの急激な増大 (スパイク) を回避する
  - 並列スキャンを利用する
    - TotalSegments の選択
- グローバルテーブルを使用するためのベストプラクティス

[AWS Black Belt Online Seminar] Amazon DynamoDB Advanced Design Pattern

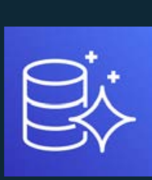
（資料 | 動画）

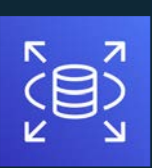
サーバーレスアプリケーションと データベースの種類 （および懸念点と対策）

サーバーレスアプリケーションでの頻出サービス

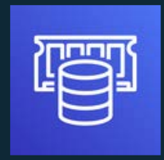
AWS Lambda

AWS Lambda とデータベース





Amazon RDS



Amazon ElastiCache

Amazon Neptune

Amazon DynamoDB

Lambda と RDB を利用する際の考慮事項

1. Lambda のスケールアウトに伴うコネクション数増

サーバーレスサービスである Lambda は負荷に応じてシームレスにスケールするので…（デフォルト同時実行数 1,000 @ 2019.05.09）

Lambda と RDB を利用する際の考慮事項

1. Lambda のスケールアウトに伴うコネクション数増

…

AWS Lambda

Amazon Aurora

Amazon RDS

Lambda 側のコネクションをプールする機構がなく、
同時接続数が膨れてやがて RDBMS の最大同時接続数を超えてしまう

※ それほど同時実行されない、または制御可能なら問題ない

Lambda と RDB を利用する際の考慮事項

~~2. VPC 内リソースアクセス時のコールドスタート~~

VPC

Private subnet

Any Subnet

AWS Lambda

Elastic network interface

Amazon Aurora

Amazon RDS

は、2019/09/04 の アップデートによって 順次改善していきます！

AWS Lambda の VPC 機能改善アップデート

Amazon Web Services ブログ

[発表] Lambda 関数が VPC 環境で改善されます

by AWS Japan Staff | on 04 SEP 2019 | in Amazon VPC, AWS Lambda, Serverless | Permalink | Share

本投稿は AWS サーバーレス アプリケーションのプリンシパルデベロッパーアドボケートであるChris Munnsによる寄稿です。

AWS Lambda 関数が Amazon VPC ネットワークでどのように機能するかが大幅に改善されたことをお知らせします。2019年9月3日（PST）のローンチ機能により、関数の起動パフォーマンスが劇的に改善され、Elastic Network Interface がより効率的に使用されるようになります。 これらの改善は、すべての既存および新しい VPC 接続の関数に追加費用なしで展開されています。 ロールアウトは 2019年9月3日（PST）から始まり、すべてのリージョンで今後数か月にわたって徐々に展開されます。

Lambda は2016年2月に初めて VPC をサポートし、AWS Direct Connect リンクを使用して VPC またはオンプレミスシステムのリソースにアクセスできるようになりました。 それ以来、お客様が VPC 接続を広く使用して、さまざまなサービスにアクセスしているのを見てきました。

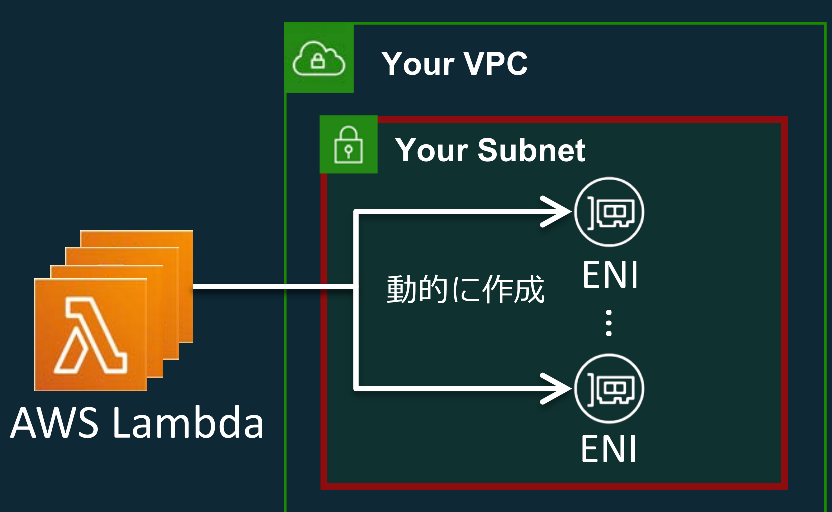
- Amazon RDS などのリレーショナルデータベース
- Redis 用 Amazon ElastiCache または Amazon Elasticsearch Service などのデータストア
- Amazon EC2 または Amazon ECS で実行されているその他のサービス

2019年9月3日（PST）のリリース機能により、LambdaとVPC 間のこの接続性が強化されます。

https://aws.amazon.com/jp/blogs/news/announcing-improved-vpc-networking-for-aws-lambda-functions/

Lambda の VPC 改善について整理

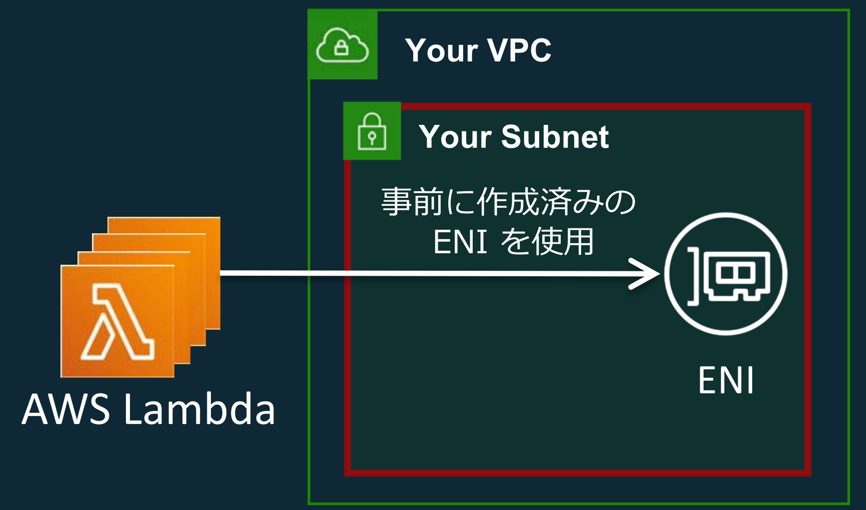
これまで



Lambda 関数の実行時、必要に応じて動的に
ENI が作成される場合があった
→ そのとき最大 10 秒ほどかかっていた

これから

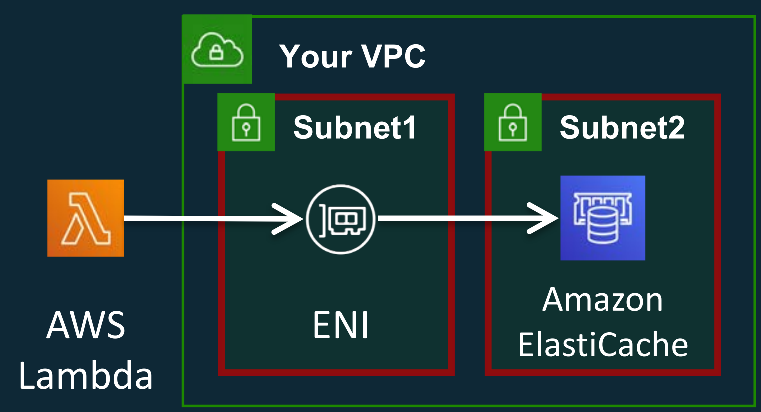
（VPC 改善が適用されてから）

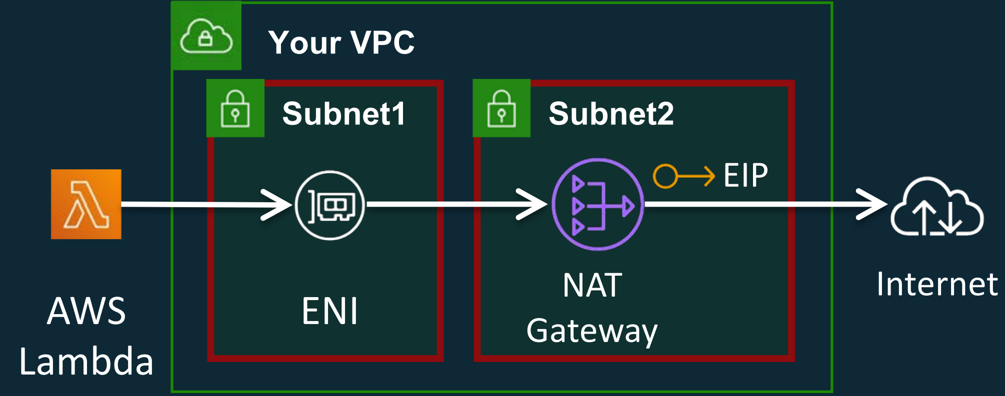


Lambda 関数の作成時または VPC 設定有効化時に
あらかじめ ENI を作成し、以降はそれを使える
（作成には最大90秒かかる場合がある）

aws

たとえばこんな使い方がより気軽にできるように





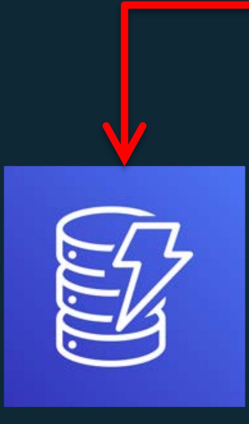
VPC 内の ElastiCache を
Lambda から利用

Elastic IP アドレスを関連付けた
NAT Gateway を通して IP アドレスを固定

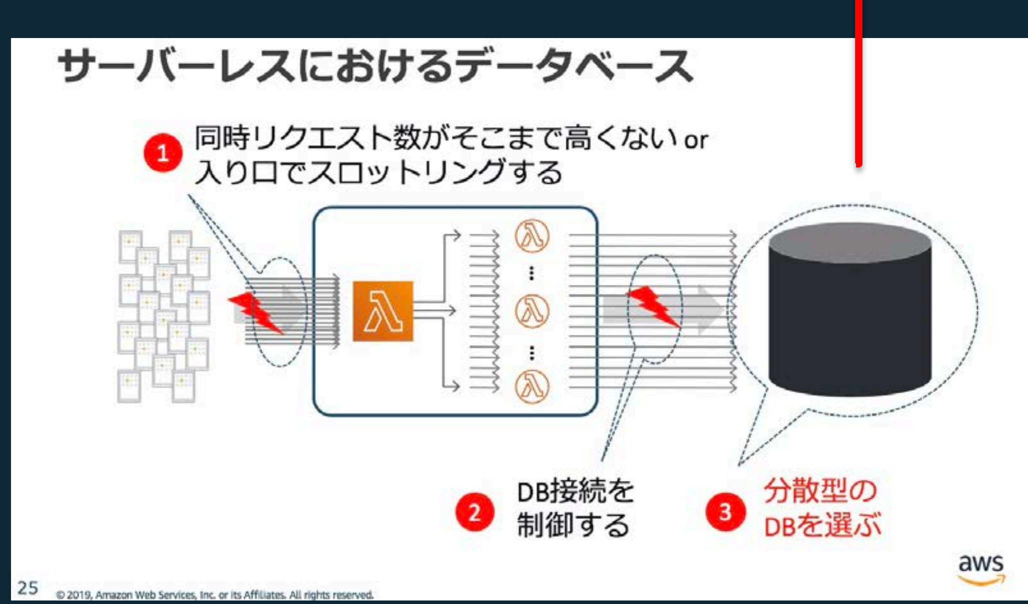
Lambda の VPC 改善について留意事項

- 2019/9/3 (PST) から、今後数ヶ月にわたって適用されていきます
- 「VPC 設定時の ENI 生成に関するコールドスタート」が改善したのであって、**RDB と Lambda を利用する際にコネクション数の問題を考慮すべきであることは変わりません**

というわけで…



Amazon DynamoDB を使う



今日の１つ目の講演「サーバーレスのおさらい」より抜粋

Amazon DynamoDB で懸念事項を解決する

- 分散データベースでありコネクション数の問題から解放
- Capacity Unit のプロビジョニング、Auto Scaling、On-Demand などによりスループットがスケーラブル

データベース設計プロセス
～ RDB と Amazon DynamoDB ～

補足: この先数ページで前提となる DynamoDB の知識とベストプラクティス

- テーブルの数は最小限に留める
  - 一箇所にあるデータに、テーブルやインデックスを通じアクセスすることで望む形のデータが入手しやすいように構成
  - キャパシティユニットの効率向上
  - テーブルを分けるべき例外はある
- DynamoDB は Primary Key（Partition Key(PK)、または PK + Sort Key(SK) の複合）でデータを識別し、アクセスする
  - Scan または Query 等のAPI利用
- グローバルセカンダリインデックスは元テーブルから非同期レプリケーションされる別テーブルのような存在
- 1テーブルや1インデックスに複数種類のアイテムをもたせてもよいし、1属性に複数種類の値を入れてもよい

その他の知識やテクニックは
ドキュメントや前述の参考資料を参照のこと

AWS ドキュメント » Amazon DynamoDB » 開発者ガイド » DynamoDB のベストプラクティス

DynamoDB のベストプラクティス

このセクションでは、Amazon DynamoDB 使用時にパフォーマンスを最大にしてスループットコストを最小にするための推奨事項をすばやく確認することができます。

目次

- DynamoDB に合わせた NoSQL 設計
  - リレーショナルデータ設計と NoSQL の相違点
  - NoSQL 設計の 2 つの重要な概念。
  - NoSQL 設計へのアプローチ
- パーティションキーを効率的に設計し、使用するためのベストプラクティス
  - バーストキャパシティーを効率的に使用する
  - DynamoDB アダプティブキャパシティーを理解する
  - パーティションキーを設計してワークロードを均等に分散する
  - 書き込みシャーディングを使用してワークロードを均等に分散させる
    - ランダムなサフィックスを使用したシャーディング
    - 計算されたサフィックスを使用したシャーディング
  - データアップロード時に書き込みアクティビティを効率的に分散する
- ソートキーを使用してデータを整理するためのベストプラクティス
  - バージョンコントロールにソートキーを使用する
- DynamoDB でセカンダリインデックスを使用するためのベストプラクティス
  - DynamoDB のセカンダリインデックスの一般的なガイドライン
    - インデックスを効率的に使用する
    - 射影を慎重に選択する
    - フェッチを回避するための頻繁なクエリの最適化
    - ローカルセカンダリインデックス作成時に項目コレクションのサイズ制限を確認する
  - スパースなインデックスの利用
    - DynamoDB のスパースインデックスの例
  - マテリアライズされた集計クエリでグローバルセカンダリインデックスを使用する
  - グローバルセカンダリインデックスの多重定義
  - 選択的なテーブルクエリに対してグローバルセカンダリインデックスの書き込みシャーディングを使用する
  - グローバルセカンダリインデックスを使用して結果整合性のあるレプリカを作成する
- 大きな項目と属性を格納するベストプラクティス
  - 大量の属性値を圧縮する
  - 大きな属性値を Amazon S3 に保存する
- DynamoDB で時系列データを処理するベストプラクティス
  - 時系列データの設計パターン
  - 時系列テーブルの例
- 多対多の関係を管理するためのベストプラクティス
  - 隣接関係のリスト設計パターン
  - マテリアライズドグラフパターン
- ハイブリッドなデータベースシステムを実装するためのベストプラクティス
  - すべてを DynamoDB に移行したくない場合は
  - ハイブリッドシステムの実装方法
- DynamoDB でリレーショナルデータをモデル化するためのベストプラクティス
  - DynamoDB でリレーショナルデータをモデル化するための最初のステップ
  - DynamoDB でリレーショナルデータをモデル化する例
- クエリとスキャンデータのベストプラクティス
  - スキャンのパフォーマンスに関する考慮事項
  - 読み込みアクティビティの急激な増大 (スパイク) を回避する
  - 並列スキャンを利用する
    - TotalSegments の選択
- グローバルテーブルを使用するためのベストプラクティス

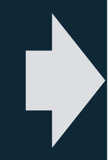
https://docs.aws.amazon.com/ja_jp/amazondynamodb/latest/developerguide/best-practices.html

aws

RDB の設計（モデリング）プロセス

業務分析と論理データモデリング

- 対象ドメイン（業務、サービス）の事実を洗い出し、概念的→論理的にデータモデリング
- One fact in one place に則って正規化を実施



ER 図 等

物理データモデリング

- 論理データモデルを元に、DBMS での実装を意識した情報を追加、定義
- 物理キー設計、データ型、制約、DDL 定義など

テーブル定義書 等

アプリ・SQL 設計、実装

- アプリケーションからは SQL を通じて柔軟にアクセス

実際のクエリ

追加要件が生じたら？

- モデルを拡張する

DynamoDB の設計（モデリング）プロセス

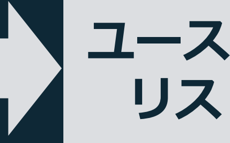
業務分析とデータのモデリング

- 対象ドメインのデータをモデリング
- RDB 設計と同じく ER 図による概念と論理レベルの整理は有効

ER 図 等

アクセスパターン設計

- ビジネス要件からアプリケーションで必要な機能とデータ（アクセスパターン）を整理する
  - 「従業員情報を ID で検索する」など



ユースケースリスト 等

Table と Index 設計

- ユースケースを満たせるテーブルおよびインデックスのスキーマを設計する

スキーマ定義書 等

クエリ条件設計

- クエリの詳細を設計、定義する
- ユースケースごとに利用するパラメータ、インデックス、検索条件などを書き出す

クエリ条件定義書 等

追加要件が生じたら？

- サービスに合わせてテーブル・インデックス設計を変更する
- 他のサービスと連携する etc

DynamoDB の設計（モデリング）プロセス

それぞれのプロセスを
実例を交えてもう少し詳しく
見てみよう

DynamoDB の設計（モデリング）プロセス

業務分析とデータのモデリング

- 対象ドメインのデータをモデリング
- RDB 設計と同じく ER 図による概念と論理レベルの整理は有効

アクセスパターン設計

- ビジネス要件からアプリケーションで必要な機能とデータ（アクセスパターン）を整理する
  - 「従業員情報を ID で検索する」など

ユースケースリスト 等

Table と Index 設計

- ユースケースを満たせるテーブルおよびインデックスのスキーマを設計する

スキーマ定義書 等

クエリ条件設計

- クエリの詳細を設計、定義する
- ユースケースごとに利用するパラメータ、インデックス、検索条件などを書き出す

クエリ条件定義書 等

追加要件が生じたら？

- サービスに合わせてテーブル・インデックス設計を変更する
- 他のサービスと連携する etc

1. 業務分析とデータのモデリング → ER 図

Product

EBOOKACR GUID PK

CompanionMapping

CMID GUID PK

EBOOKACR GUID NULL FK
ABOOKACR GUID NULL FK
InfoID GUID NULL FK

AudioProduct

ABOOKACR GUID PK

ACR_Info

InfoID GUID PK

AusioFile

AudioFileID GUID PK

ABOOKACR GUID FK

Audible eBook Sync Service

- ユーザーが Audible eBooks の セッション情報を保存出来る
- eBook 及び audio product の ユーザー毎のマッピング
- スパイクする負荷に対応する ため多くのキャパシティが重要
- 多数のアクセスパターン対応

(from re:Invent 2018 session)

DynamoDB の設計（モデリング）プロセス

業務分析とデータのモデリング

- 対象ドメインのデータをモデリング
- RDB 設計と同じく ER 図による概念と論理レベルの整理は有効

ER 図 等

アクセスパターン設計

- ビジネス要件からアプリケーションで必要な機能とデータ（アクセスパターン）を整理する
  - 「従業員情報を ID で検索する」など

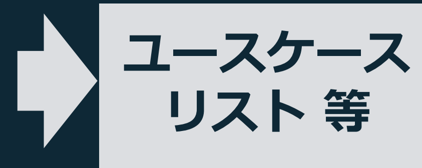


Table と Index 設計

- ユースケースを満たせるテーブルおよびインデックスのスキーマを設計する

スキーマ定義書 等

クエリ条件設計

- クエリの詳細を設計、定義する
- ユースケースごとに利用するパラメータ、インデックス、検索条件などを書き出す

クエリ条件定義書 等

追加要件が生じたら？

- サービスに合わせてテーブル・インデックス設計を変更する
- 他のサービスと連携する etc

2. アクセスパターン設計 → ユースケースリスト

#	Entity	Use Case
1	CompanionMapping	getCompanionMappingsByAsin
2	CompanionMapping	getCompanionMappingsByEbookAndAudiobookContentId
3	CompanionMapping	getCompanionMappingsFromCache
4	CompanionMapping	getCompanionMappings
5	CompanionMapping	getCompanionMappingsAvailable
6	AcrInfo	getACRInfo
7	AcrInfo	getACRs
8	AcrInfo	getACRInfos
9	AcrInfo	getACRInfosbySKU
10	AudioProduct	getAudioProductsForACRs
11	AudioProduct	getAudioProducts
12	AudioProduct	deleteAudioProductsMatchingSkuVersions
13	AudioProduct	getChildAudioProductsForSKU
14	Product	getProductInfoByAsins
15	Product	getParentChildDataByParentAsins
16	AudioFile	getAudioFilesForACR
17	AudioFile	getAudioFilesForChildACR

DynamoDB の設計（モデリング）プロセス

業務分析とデータのモデリング

- 対象ドメインのデータをモデリング
- RDB 設計と同じく ER 図による概念と論理レベルの整理は有効

ER 図 等

アクセスパターン設計

- ビジネス要件からアプリケーションで必要な機能とデータ（アクセスパターン）を整理する
  - 「従業員情報を ID で検索する」など

ユースケースリスト 等

Table と Index 設計

- ユースケースを満たせるテーブルおよびインデックスのスキーマを設計する

スキーマ定義書 等

クエリ条件設計

- クエリの詳細を設計、定義する
- ユースケースごとに利用するパラメータ、インデックス、検索条件などを書き出す

クエリ条件定義書 等

追加要件が生じたら？

- サービスに合わせてテーブル・インデックス設計を変更する
- 他のサービスと連携する etc

3. テーブル・インデックス設計 → テーブル定義書

TABLE

Primary Key		Attributes		
PK	SK, GSI-3-PK	GSI-1-PK	GSI-2-PK	
Acr	TargetACR	AcrInfo1	AcrInfo2	EbookAsin
ABOOKACR1	ABOOKACR1-v0	ABOOK-ASIN1	ABOOK-SKU1	
	MAP-EBOOKACR1	SyncFileAcr	ABOOK-ASIN1	
	ABOOKACR1#TRACK#1	ABOOK-ASIN1	ABOOK-SKU1	
	ABOOKACR1#TRACK#2	ABOOK-ASIN1	ABOOK-SKU1	
EBOOKACR1	EBOOKACR1	EBOOK-SKU1		ASIN

3. テーブル・インデックス設計 → インデックス定義書

	GSI Partition Key	Projected Attributes	
	GSI-1-PK	Acr (Primary Table PK)	TargetACR (Primary Table SK)
GSI 1	ABOOK-ASIN1	ABOOKACR1	ABOOKACR1-v1
			ABOOKACR1#TRACK#1
			ABOOKACR1#TRACK#2
	SyncFileAcr	ABOOKACR1	MAP-EBOOKACR1
	EBOOK-SKU1	ABOOKACR1	EBOOKACR1

	GSI Partition Key	Projected Attributes	
	GSI-2-PK	Acr (Primary Table PK)	TargetACR (Primary Table SK)
GSI 2	ABOOK-ASIN1	ABOOKACR1	MAP-EBOOKACR1
	ABOOK-SKU1	ABOOKACR1	ABOOKACR1-v0
			ABOOKACR1#TRACK#1
			ABOOKACR1#TRACK#2

	GSI Partition Key	Projected Attributes	
	TargetACR (Primary Table SK)	Acr (Primary Table PK)	TargetACR (Primary Table SK)
GSI 3	ABOOKACR1-v0	ABOOKACR1	ABOOKACR1-v0
	MAP-EBOOKACR1	ABOOKACR1	MAP-EBOOKACR1
	ABOOKACR1#TRACK#1	ABOOKACR1	ABOOKACR1#TRACK#1
	ABOOKACR1#TRACK#2	ABOOKACR1	ABOOKACR1#TRACK#2
	EBOOKACR1	EBOOKACR1	EBOOKACR1

aws

DynamoDB の設計（モデリング）プロセス

業務分析とデータのモデリング

- 対象ドメインのデータをモデリング
- RDB 設計と同じく ER 図による概念と論理レベルの整理は有効

ER 図 等

アクセスパターン設計

- ビジネス要件からアプリケーションで必要な機能とデータ（アクセスパターン）を整理する
  - 「従業員情報を ID で検索する」など

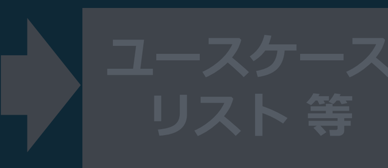


Table と Index 設計

- ユースケースを満たせるテーブルおよびインデックスのスキーマを設計する

スキーマ定義書 等

クエリ条件設計

- クエリの詳細を設計、定義する
- ユースケースごとに利用するパラメータ、インデックス、検索条件などを書き出す

クエリ条件定義書 等

追加要件が生じたら？

- サービスに合わせてテーブル・インデックス設計を変更する
- 他のサービスと連携する etc

4. クエリ条件設計 → クエリ条件定義書

#	Entity	Use Case	Lookup parameters	Table / INDEX	Key Conditions	Filter Conditions
1	CompanionMapping	getCompanionMappingsByAsin	audiobookAsin/ebookSku	GSI2	GSI-2=ABOOK-ASIN1	None
2	CompanionMapping	getCompanionMappingsByEbookAndAudiobookContentId	ebookAcr/sku,version,format or audiobookAcr/asin,version,format	GSI-3 on TargetACR attribute OR PrimaryKey on Table	GSI-3=MAP-EBOOKACR1	version=v and format=f
3	CompanionMapping	getCompanionMappingsFromCache	ebookAcr/sku,version,format or audiobookAcr/asin,version,format	GSI-3 on TargetACR attribute OR PrimaryKey on Table	GSI-3=MAP-EBOOKACR1	version=v and format=f
4	CompanionMapping	getCompanionMappings	syncfileAcr, ebookAcr?, audiobookAcr?	GSI1	GSI-1=SyncFileAcr	None
5	CompanionMapping	getCompanionMappingsAvailable	ebookAcr, audiobookAcr	Primary Key on Table	Acr=ABOOKACR1 and TargetACR beginswith "MAP-"	
6	AcrInfo	getACRInfo	acr	Primary Key on Table	Acr=ABOOKACR1 and TargetACR beginswith "ABOOKACR1-v"	
7	AcrInfo	getACRs	acr / asin,version,format	Primary Key on Table	Acr=ABOOKACR1	version=v and format=f
8	AcrInfo	getACRInfos	acr	Primary Key on table	Acr=ABOOKACR1 and TargetACR beginswith "ABOOKACR1"	
9	AcrInfo	getACRInfosBySKU	sku	GSI2	GSI-2=ABOOK-SKU1	
10	AudioProduct	getAudioProductsForACRs	acr	Primary Key on table	Acr=ABOOKACR1 and TargetACR beginswith "ABOOKACR1"	
11	AudioProduct	getAudioProducts	sku, version, format	GSI2	GSI-2=ABOOK-SKU1	version=v and format=f
12	AudioProduct	deleteAudioProductsMatchingSkuVersions	sku, version	GSI2	GSI-2=ABOOK-SKU1	version=v
13	AudioProduct	getChildAudioProductsForSKU	sku	GSI2	GSI-2=ABOOK-SKU1	
14	Product	getProductInfoByAsins	asin	GSI1	GSI-1=ABOOK-ASIN1	
15	Product	getParentChildDataByParentAsins	asin	GSI1	GSI-1=ABOOK-ASIN1	
16	AudioFile	getAudioFilesForACR	acr	Primary Key on table	Acr=ABOOKACR1 and TargetACR beginswith "ABOOKACR1#"	

aws

再掲：DynamoDB の設計（モデリング）プロセス

業務分析とデータのモデリング

- 対象ドメインのデータをモデリング
- RDB 設計と同じく ER 図による概念と論理レベルの整理は有効

ER 図 等

アクセスパターン設計

- ビジネス要件からアプリケーションで必要な機能とデータ（アクセスパターン）を整理する
  - 「従業員情報を ID で検索する」など

ユースケースリスト 等

Table と Index 設計

- ユースケースを満たせるテーブルおよびインデックスのスキーマを設計する

スキーマ定義書 等

クエリ条件設計

- クエリの詳細を設計、定義する
- ユースケースごとに利用するパラメータ、インデックス、検索条件などを書き出す

クエリ条件定義書 等

追加要件が生じたら？

- サービスに合わせてテーブル・インデックス設計を変更する
- 他のサービスと連携する etc

というわけで DynamoDB でよくある誤解の一つ

「DynamoDB ってスキーマレスだから
事前の設計いらないでしょ？」

というわけで DynamoDB でよくある誤解の一つ

~~「DynamoDB ってスキーマレスだから~~
~~事前の設計いらないでしょ？」~~

→ いいえ。ちゃんとアクセスパターンに
基づいた設計が必要です

かんたんドリル：
DynamoDB の設計シミュレーション

以下の画面を実現するデータモデルを考えよう

イベント検索

イベントID:

イベント名:

会場: AWS Loft Tokyo

開催日:

タグ:

検 索

検索結果

ID	イベント名	会場	開催日	タグ
2	DynamoDB勉強会	AWS Loft Tokyo	yy/3/4	#DynamoDB #Serverless
14	サーバーレス設計勉強会	AWS Loft Tokyo	yy/5/9	#Serverless #Lambda #Design
・・・				
33	API認証認可Night	AWS Loft Tokyo	yy/6/23	#WebAPI #JWT #APIGateway #Auth #OAuth #OIDC #Cognito

※ この例題では、シンプルに各項目を完全一致で検索できるようにするものとします
※ 現実的には Elasticsearch 等の利用も検討してください

再掲：DynamoDB の設計（モデリング）プロセス

業務分析とデータのモデリング

- 対象ドメインのデータをモデリング
- RDB 設計と同じく ER 図による概念と論理レベルの整理は有効

ER 図等

アクセスパターン設計

- ビジネス要件からアプリケーションで必要な機能とデータ（アクセスパターン）を整理する
  - 「従業員情報を ID で検索する」など

ユースケースリスト 等

Table と Index 設計

- ユースケースを満たせるテーブルおよびインデックスのスキーマを設計する

スキーマ定義書 等

クエリ条件設計

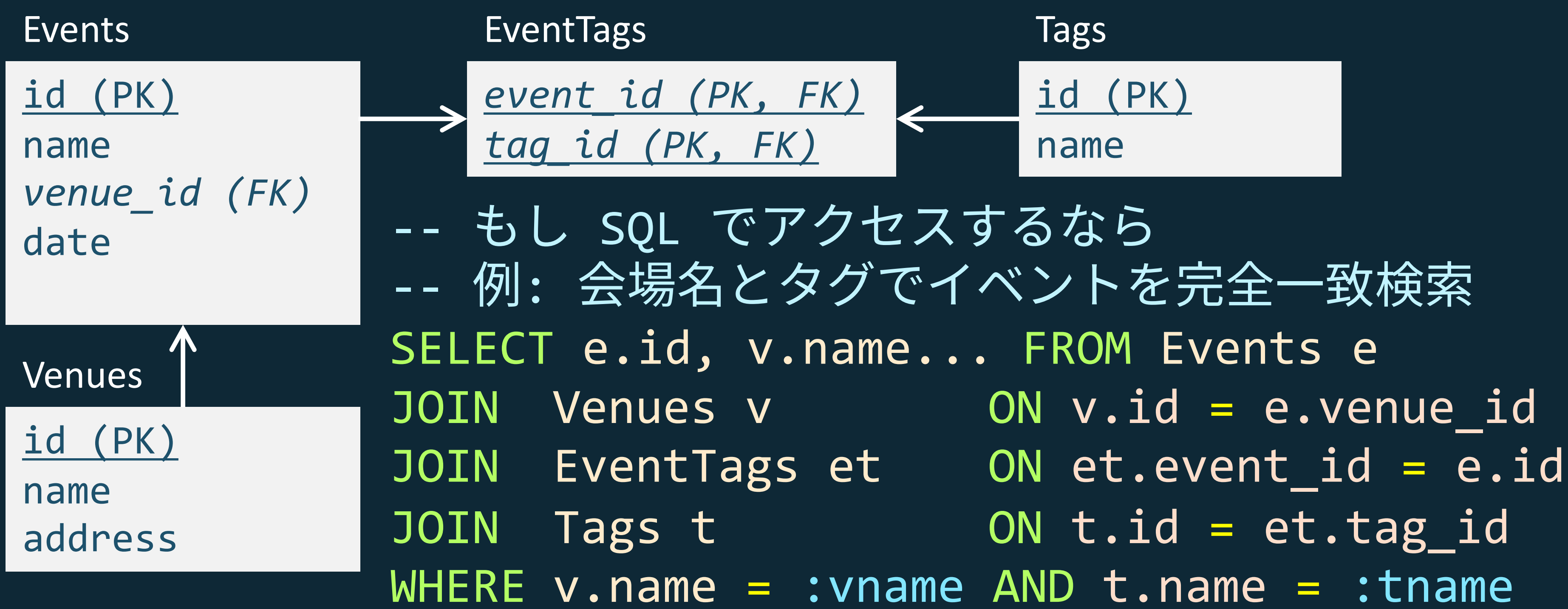
- クエリの詳細を設計、定義する
- ユースケースごとに利用するパラメータ、インデックス、検索条件などを書き出す

クエリ条件定義書 等

追加要件が生じたら？

- サービスに合わせてテーブル・インデックス設計を変更する
- 他のサービスと連携する etc

1. データモデリング



2. アクセスパターン設計 → ユースケースリスト

#	Entity	Use Case
1	Events	getEventByEventID
2	Events	getEventsByEventName
3	Events	getEventsByVenueName
4	Events	getEventsByDate
5	Events	getEventsByTag
6	Tags	getTagsByEventID
7	Venue	getVenueByEventID

まず、何も考えずに
シンプルに DynamoDB に
落とし込んでみると？

3. テーブル・インデックス設計 - シンプルなパターン

TABLE	Partition Key	Attributes			
	EventID	EventName	VenueName	Tags	Date
	{EventID}	*{EventName}*	*{VenueName}*	*{TagName1, ...}*	*{yyyy-mm-dd}*
	...	...	...	...	...

※ *{foo}* は何らかの値が入っていることを示す

この構成の懸念点

- GetItem や Query が EventID にしか使えず、それ以外の属性で検索したいとき Scan + Filter で高コストになってしまう
  - Scan はテーブル・インデックス内を文字通りフルスキャンするため、パフォーマンスおよびコスト面のリスクがある
  - 項目数が少なく Scan で問題ないことが担保できている場合などを除き、一般に Scan でなく Query や GetItem / BatchGetItem の利用を検討すべき

3. テーブル・インデックス設計 - シンプル + GSI パターン

TABLE	Primary Key	Attributes			
	Partition Key	GSI-1-PK	GSI-2-PK	GSI-3-PK	GSI-4-PK
	EventID	EventName	VenueName	Tags	Date
	{EventID}	*{EventName}*	*{VenueName}*	*{TagName1, ...}*	*{yyyy-mm-dd}*
	…	…	…	…	…

この構成で解決した点

- 各属性に Global Secondary Index が貼られたため Query が可能に
  ※ GSI はデフォルトで 20 個/テーブル。それ以上は上限緩和を申請

この構成の懸念点

- GSI が増えることによる金銭（スループット+ストレージ）コスト増、管理コスト増
- 多くの GSI をアプリケーションが意識する必要がある
  ※ このスキーマが絶対にダメというわけではないが他の設計パターンを踏まえて検討すべき

では、DynamoDB らしい テクニックを使ってみると？

- GSI オーバーローディング（GSI の多重定義） 等

3. テーブル・インデックス設計 → テーブル定義書

TABLE	Primary Key		Attributes		
	PK, GSI-1-SK	SK	GSI-1-PK	GSI-2-PK	
	ID	DataType	DataValue	VenueName	VenueAddress
	{EventID}	EventName	*{EventName}*		
	{EventID}	VenueID	*{VenueID}*		
	{EventID}	Date	*{yyyy-mm-dd}*		
	{EventID}	Tag_*{TagName}*	Tag_*{TagName}*		
	{VenueID}	VenueInfo		*{VenueName}*	*{Address}*

- 1 項目内にフラットに属性を並べるのではなく、情報を縦に持つ
- 一つの GSI に複数の検索要件をもたせる（GSI オーバーローディング）

この構成で解決した点

- GSI の数が少なくて済み、コストやクエリを最適化できる

3. テーブル・インデックス設計 → インデックス定義書

<table>
<tr><td rowspan="7">GSI 1</td><td colspan="2">GSI Partition Key</td><td colspan="2">Projected Attributes</td></tr>
<tr><td>GSI-1-PK</td><td>GSI-1-SK</td><td>(Primary Table SK)</td><td></td></tr>
<tr><td>DataValue</td><td>ID</td><td>DataType</td><td>VenueAddress</td></tr>
<tr><td>*{EventName}*</td><td>*{EventID}*</td><td>EventName</td><td></td></tr>
<tr><td>*{VenueID}*</td><td>*{EventID}*</td><td>Venue</td><td></td></tr>
<tr><td>*{yyyy-mm-dd}*</td><td>*{EventID}*</td><td>Date</td><td></td></tr>
<tr><td>Tag_*{TagName}*</td><td>*{EventID}*</td><td>Tag_*{TagName}*</td><td></td></tr>
</table>

<table>
<tr><td rowspan="4">GSI 2</td><td>GSI Partition Key</td><td colspan="2">Projected Attributes</td></tr>
<tr><td>GSI-2-PK</td><td>(Primary Table PK)</td><td></td></tr>
<tr><td>VenueName</td><td>ID</td><td>VenueAddress</td></tr>
<tr><td>*{VenueName}*</td><td>*{VenueID}*</td><td>*{Address}*</td></tr>
</table>

- 検索条件として指定したい属性をキーとする GSI を定義

3.1 格納されるデータイメージ

TABLE	Primary Key		Attributes		
	PK, GSI-1-SK	SK	GSI-1-PK	GSI-2-PK	
	ID	DataType	DataValue	VenueName	VenueAddress
	E123	EventName	DynamoDB勉強会		
	E123	VenueID	V32		
	E123	Date	yy/3/4		
	E123	Tag_#DynamoDB	Tag_#DynamoDB		
	E123	Tag_#Serverless	Tag_#Serverless		
	E145	EventName	サーバーレス設計勉強会		
	E145	VenueID	V32		
	E145	Date	yy/5/9		
	E145	Tag_#Serverless	Tag_#Serverless		
	E145	Tag_#Lambda	Tag_#Lambda		
	E145	Tag_#Design	Tag_#Design		
	V32	VenueInfo		AWS Loft Tokyo	目黒セントラルスクエア

- 実際にはこのような値が入ることを想定
- この例の場合、E123 と E145 の 2 つのイベント情報。
  会場はどちらも V32 としている

3.1 格納されるデータイメージ

TABLE

Primary Key		Attributes		
PK, GSI-1-SK	SK	GSI-1-PK	GSI-2-PK	
ID	DataType	DataValue	VenueName	VenueAddress
E123	EventName	DynamoDB勉強会		
E123	VenueID	V32		
E123	Date	yy/3/4		
E123	Tag_#DynamoDB	Tag_#DynamoDB		
E123	Tag_#Serverless	Tag_#Serverless		
E145	EventName	サーバーレス設計		
E145	VenueID	V32		
E145	Date	yy/5/9		
E145	Tag_#Serverless	Tag_#Serverless		
E145	Tag_#Lambda	Tag_#Lambda		
E145	Tag_#Design	Tag_#Design		
V32	VenueInfo		AWS Loft Tokyo	目黒セントラルスクエア


```
{
    "Items": [
        {"DataType": {"S": "EventName"},"DataValue": {"S": "DynamoDB勉強会"}},
        {"DataType": {"S": "VenueID"},"DataValue": {"S": "V32"}},
        {"DataType": {"S": "Date"},"DataValue": {"S": "yy/3/4"}},
        {"DataType": {"S": "Tag_#DynamoDB"},"DataValue": {"S": "#DynamoDB"}},
        {"DataType": {"S": "Tag_#Serverless"},"DataValue": {"S": "#Serverless"}
    ],
    "Count": 5,
    "ScannedCount": 5,
    "ConsumedCapacity": null
}
```


Query(ID = :eventId) で該当 EventID のアイテムが取得できる

例: Table.Query(ID = "**E123**")

3.1 格納されるデータイメージ

TABLE

Primary Key		Attributes		
PK, GSI-1-SK	SK	GSI-1-PK	GSI-2-PK	
ID	DataType	DataValue	VenueName	VenueAddress
E123	EventName	DynamoDB勉強会		
E123	VenueID	V32		
E123	Date	yy/3/4		
E123	Tag_#DynamoDB	Tag_#DynamoDB		
E123	Tag_#Serverless	**Tag_#Serverless**		
E145	EventName	サーバーレス設計		
E145	VenueID	V32		
E145	Date	yy/5/9		
E145	Tag_#Serverless	**Tag_#Serverless**		
E145	Tag_#Lambda	Tag_#Lambda		
E145	Tag_#Design	Tag_#Design		
V32	VenueInfo		AWS Loft Tokyo	目黒セントラルスクエア


```
{
  "Items": [
    {"ID": {"S": "E123"},"DataValue": {"S": "Tag_#Serverless"
    {"ID": {"S": "E145"},"DataValue": {"S": "Tag_#Serverless"
  ],
  "Count": 2,
  "ScannedCount": 2,
  "ConsumedCapacity": null
}
```


Query(DataValue = :tagName) で該当 タグ を持つ Event の ID を取得できる

例: GSI1.Query(DataValue = "**Tag_#Serverless**")

aws

4. クエリ条件定義

#	Entity	Use Case	Parameters	Table/Index	API & Key Conditions
1	Events	getEventByEventID	*{eventId}*	Primary Table	Query(ID = :eventId)
2	Events	getEventsByEventName	*{eventName}*	GSI-1	Query(DataValue = :eventName)
3	Events	getEventsByVenueName	*{venueName}*	GSI-2, GSI-1	Query(VenueName = :venueName), Query(DataValue = :venueId)
4	Events	getEventsByDate	*{date}*	GSI-1	Query(DataValue = :date)
5	Events	getEventsByTag	*{tagName}*	GSI-1	Query(DataValue = :tagName)
6	Tags	getTagsByEventID	*{eventId}*	Primary Table	Query(ID = :eventId)
7	Venue	getVenueByEventID	*{eventId}*	Primary Table, Primary Table	GetItem(ID = :eventId and DataType = VenueID), GetItem(ID = :venueId and DataType = VenueInfo)

- どのユースケースにはどの Table / Index を使い、どうデータを問い合わせるのか（GetItem? GetBatchItem? Query? Scan?）を列挙
- Filter Condition や Projection Expression などを記述してもよし

まとめ

まとめ

- サーバーレスアプリケーション には
スケーラブルな Amazon DynamoDB が使いやすい
    - 懸念事項を考慮の上で許容できるならその限りではない
- DynamoDB のモデリングプロセスは
    1. 業務分析してデータモデルを整理
    2. ユースケースリスト
    3. テーブル・インデックス設計
    4. クエリ条件設計
- 公式ドキュメントのベストプラクティスを踏まえた上で
設計に臨むべし

ぜひ参考資料を活用してください

- Amazon DynamoDB 開発者ガイド（Link）
  - DynamoDB のベストプラクティス（Link）
- Amazon DynamoDB のベストプラクティスに従うという 2019 年の計を立てる - Amazon Web Services ブログ（Link）
- [AWS Black Belt Online Seminar] Amazon DynamoDB Advanced Design Pattern（資料 | 動画）
- DynamoDBデータモデリング虎の巻 - misc.tech.notes （Link）
- AppSyncを使いこなすためのDynamoDB設計パターン（Link）

Thank you

よいサーバーレスライフを！

Appendix-1

かんたんドリル別解の例

Appendix-1. テーブル・インデックス設計 → テーブル定義書

TABLE	Primary Key		Attributes	
	PK, GSI-1-SK	SK	GSI-1-PK	
	ID	DataType	DataValue1	DataValue2
	{EventID}	EventName	*{EventName}*	
	{EventID}	VenueInfo	*{VenueName}*	*{Address}*
	{EventID}	Date	*{yyyy-mm-dd}*	
	{EventID}	Tag_*{TagName}*	Tag_*{TagName}*	

- GSI を一つにし、会場情報を ID で別持ちせず
各 Event アイテム内に含めたパターン

Appendix-1. 格納されるデータイメージ

TABLE	Primary Key		Attributes	
	PK, GSI-1-SK	SK	GSI-1-PK	
	ID	DataType	DataValue1	DataValue2
	E123	EventName	DynamoDB勉強会	
	E123	VenueInfo	AWS Loft Tokyo	目黒セントラルスクエア
	E123	Date	yy/3/4	
	E123	Tag_#DynamoDB	Tag_#DynamoDB	
	E123	Tag_#Serverless	Tag_#Serverless	
	E145	EventName	サーバーレス設計勉強会	
	E145	VenueID	AWS Loft Tokyo	目黒セントラルスクエア
	E145	Date	yy/5/9	
	E145	Tag_#Serverless	Tag_#Serverless	
	E145	Tag_#Lambda	Tag_#Lambda	
	E145	Tag_#Design	Tag_#Design	

aws

Appendix-1. テーブル・インデックス設計 → インデックス定義書

GSI 1	GSI Partition Key		Projected Attributes	
	GSI-1-PK	GSI-1-SK	(Primary Table SK)	
	DataValue	ID	DataValue1	DataValue2
	{EventName}	*{EventID}*	EventName	
	{VenueID}	*{EventID}*	Venue	
	{yyyy-mm-dd}	*{EventID}*	Date	
	Tag_*{TagName}*	*{EventID}*	Tag_*{TagName}*	
	{VenueName}	*{EventID}*	*{VenueName}*	*{Address}*

aws

Appendix-1. クエリ条件定義

#	Entity	Use Case	Parameters	Table/Index	API & Key Conditions
1	Events	getEventByEventID	*{eventId}*	Primary Table	Query(ID = :eventId)
2	Events	getEventsByEventName	*{eventName}*	GSI-1	Query(DataValue1 = :eventName)
3	Events	getEventsByVenueName	*{venueName}*	GSI-1	Query(DataValue1 = :venueName)
4	Events	getEventsByDate	*{date}*	GSI-1	Query(DataValue1 = :date)
5	Events	getEventsByTag	*{tagName}*	GSI-1	Query(DataValue1 = :tagName)
6	Tags	getTagsByEventID	*{eventId}*	Primary Table	Query (ID = :eventId and DataType begins_with Tag_)
7	Venue	getVenueByEventID	*{eventId}*	Primary Table	GetItem (ID = :event and DataType = VenueInfo)

- この設計でも列挙したユースケースリストの要件は満たせる
- 本文中のパターンと何が違い、Pros/Cons は何なのか？（続きは次ページ）

aws

Appendix-1. と本文でスキーマの違いによってどんな影響がある？

比較事項	本文中のパターン	Appendix-1 のパターン
GSI の数	2	1
getEventsByVenueName を実装する際のアクセス回数	2	1
会場情報 （VenueName, Address等） を修正する際のUpdate回数	1	登録された Event の数 ※ ビジネス要件次第。下記参照

- どちらがよい、悪いとは一概に言えない。要件と制約次第
- 例えば Appendix-1 パターンで会場情報を修正する必要が出た場合、次のような条件や考慮によって処理量やリスクは異なる
  - そもそも過去のイベントの会場情報まで含め修正すべきなのか？
  - 一度に修正すべきなのか？（順次低スピードで修正すれば済むのか？など）
  - 数十万、数百万アイテム程度ならバッチを走らせて更新してしまえばよいのでは？

Appendix-2

途中で要件を満たせないことに気づいた
失敗設計の例と失敗箇所当てエクササイズ
※ フィクションです

Appendix-2. テーブル・インデックス設計 - 失敗パターン

TABLE	Primary Key		Attributes
	PK	**SK, GSI-1-PK**	**GSI-1-SK**
	EventID	**DataType**	**DataValue**
	{EventID}	EventName	*{EventName}*
	{EventID}	VenueName	*{VenueName}*
	{EventID}	VenueAddress	*{VenueAddress}*
	{EventID}	Date	*{yyyy-mm-dd}*
	{EventID}	Tag_*{n}*	*{TagName}*

aws

Appendix-2. 格納されるデータイメージ

TABLE	Primary Key		Attributes
	PK	SK, GSI-1-PK	GSI-1-SK
	EventID	DataType	DataValue
	E123	EventName	DynamoDB勉強会
	E123	VenueName	AWS Loft Tokyo
	E123	VenueAddress	目黒セントラルスクエア
	E123	Date	yy/3/4
	E123	Tag_1	#DynamoDB
	E123	Tag_2	#Serverless
	E145	EventName	サーバーレス設計勉強会
	E145	VenueName	AWS Loft Tokyo
	E145	VenueAddress	目黒セントラルスクエア
	E145	Date	yy/5/9
	E145	Tag_1	#Serverless
	E145	Tag_2	#Lambda
	E145	Tag_3	#Design

Appendix-2. テーブル・インデックス設計

GSI 1	GSI Partition Key		Projected Attributes
	GSI-1-PK	GSI-1-SK	(Primary Table SK)
	DataType	DataValue	EventID
	EventName	*{EventName}*	*{EventID}*
	VenueName	*{VenueName}*	*{EventID}*
	VenueAddress	*{VenueAddress}*	*{EventID}*
	Date	*{yyyy-mm-dd}*	*{EventID}*
	Tag_*{n}*	*{TagName}*	*{EventID}*

aws

Appendix-2. クエリ条件定義

#	Entity	Use Case	Parameters	Table/Index	Key Conditions
1	Events	getEventByEventID	*{eventId}*	Primary Table	EventID = :eventId
2	Events	getEventsByEventName	*{eventName}*	GSI-1	DataType = EventName And EventName = :eventName
3	Events	getEventsByVenueName	*{venueName}*	GSI-1	DataType = VenueName And VenueName = :venueName
4	Events	getEventsByDate	*{date}*	GSI-1	DataType = Date And Date = :date
5	Events	getEventsByTag	**※ここでやっと 実現不可能な ことに気づいた！**		
6	Tags	getTagsByTagName			
7	Tags	getTagsByEventID			
8	Venue	getVenueByEventID			

- この設計だとなぜ getEvenetsByTag が実現できないのか？（答えは次ページ）

Appendix-2. でなぜ getEventsByTag が満たせないのか

GSI 1	GSI Partition Key		Projected Attributes
	GSI-1-PK	**GSI-1-SK**	**(Primary Table SK)**
	DataType	**DataValue**	**EventID**
	EventName	*{EventName}*	*{EventID}*
	VenueName	*{VenueName}*	*{EventID}*
	VenueAddress	*{VenueAddress}*	*{EventID}*
	Date	*{yyyy-mm-dd}*	*{EventID}*
	Tag_*{n}*	*{TagName}*	*{EventID}*

- TagName から Event 情報を引き当てるために
  `Query(DataType = begins_with("Tag_") and DataValue = :tagName)`
  のようなクエリをイメージしていたが、
  begins_with 関数は Sort Key にしか使えない
- `Query(DataType = Tag_{n} and DataValue = :tagName)`
  としようとしても、クエリ発行側からは {n} の値を定めることができず、
  Partition Key が指定できない