DevAx connect

今からはじめる、開発者のための DynamoDB 再入門

2021/10/29

Tsutsumi, Hayato
Senior Solutions Architect

自己紹介

堤 勇人 (Tsutsumi, Hayato)

Senior Solutions Architect

前職はパッケージベンダーの開発者
10年程Cassandraでサービスを作っており、
設計から開発、運用、サポート、コスト管理まで一通り経験しました。
好きなAWSのサービス：Amazon DynamoDB、Amazon Keyspaces

本セッションの対象

- RDBを使ったアプリケーションに関わったことがある開発者の方
- DynamoDBで実際開発するのはまだちょっと、という方
- Amazon DynamoDB(以下DynamoDB)の上手い使い方が知りたい方

本セッションの概要

DynamoDBで

- 出来ること
- 必要なこと
- 考えるべきところ

等のベストプラクティスを紹介します。

本セッションのゴール

1. DynamoDBの名前を知る
2. DynamoDBの使い所を知る
3. 新規プロジェクトでDynamoDBを提案できる

100点！

1000点！

1000000000000000000
点！

Purpose Builtのおさらい

- キーワード：適材適所
- DynamoDBの選択ポイントは
  - サーバーレスで運用負荷が少ない
  - 一貫した低レイテンシ
  - 高いスケーラビリティ
- 本セッションではRDBとDynamoDBを対比して話す箇所があるが、RDBを全てDynamoDBで置き換えられるという話ではない
- DynamoDBの使いどころを見極め、活用する一助として欲しい

アジェンダ

- DynamoDBに向いていない？
  - 検討中のシステムはDynamoDBに向いてない、と考えている人はどんなことを思っているのか
- DynamoDBの特長と使い方
  - スケーラビリティ
  - 整合性とトランザクション
  - データモデリングとアクセスパターン
  - OLTPとOLAP
- ラーニングパス
- まとめ

DynamoDBに向いてない？

DynamoDBに向いてない？

RDBのスケール限界になる
量のデータはないね

→ スケーラビリティの問題

aws DevAx connect

DynamoDBに向いてない？

結果整合性って古いデータが
返ってくるんでしょ？それは
困る

→ 整合性とトランザクションの問題

aws DevAx connect

DynamoDBに向いてない？

ウチはビジネス向けなので、
トランザクションが無いと
ダメ

→ 整合性とトランザクションの問題

aws DevAx connect

DynamoDBに向いてない？

joinしているところ全部非正規化したら大変なことになる、ウチは向いてない

→ データモデリングと
アクセスパターンの問題

aws DevAx connect

DynamoDBに向いてない？

アクセスパターンからデータモデリングをしろと言うが、ユーザーに依るのでわからない

→ データモデリングと
アクセスパターンの問題

aws DevAx connect

DynamoDBに向いてない？

ウチは分析が主なので
当てはまらないですね

→ OLTPとOLAPの問題

aws DevAx connect

DynamoDBに向いてない？

気にはなっているが、どう勉強すればいいか分からないし、独学だと不安

→ ラーニングパス

aws DevAx connect

DynamoDBに向いてない？

スケーラビリティ

OLTPとOLAP

整合性とトランザクション

ラーニングパス

データモデリングと
アクセスパターン

aws DevAx connect

スケーラビリティ

スケーラビリティとは？

- スケーラビリティってどういう意味だと思いますか？
  - スケールできる？（直訳）
  - すごく大きいデータも入る？
  - 性能がいい？

スケーラビリティとは？

- 幅広い意味で使われるが、この文脈では
  RDBがスケール限界に来た時に対応できる能力、では△

コストと性能がパフォーマンス良く
比例するのがスケーラビリティ

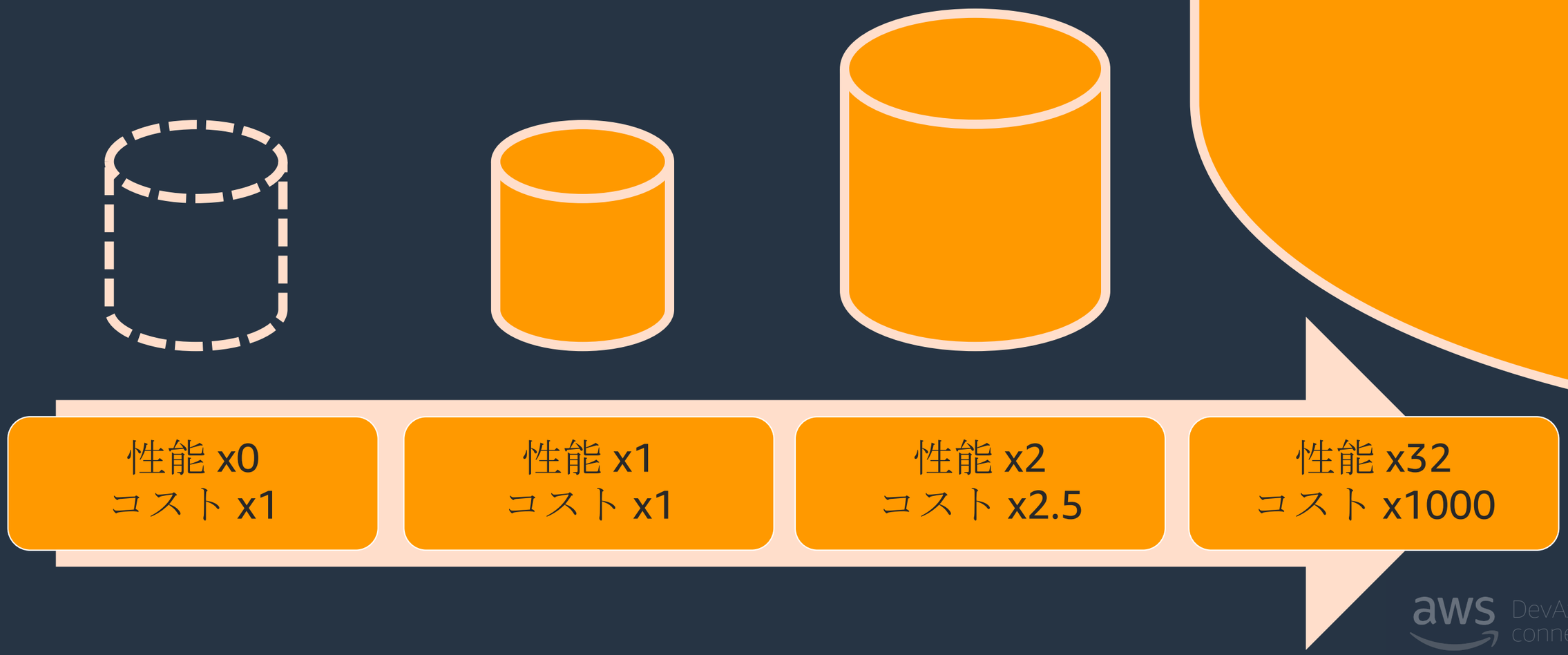
aws DevAx connect

サーバーの性能とコスト

性能 x1 コスト x1	性能 x2 コスト x2.5	性能 x32 コスト x1000

aws DevAx connect

サーバーの性能とコスト



aws DevAx connect

DynamoDBのキャパシティーユニット

- Capacity Unitという単位でスループットを管理する

- 1 書き込みキャパシティーユニット (WCU) = 1 KBの書き込み
- 1 読み込みキャパシティーユニット (RCU) = 4 KBの読み込み

- Read/Writeは別々に管理される

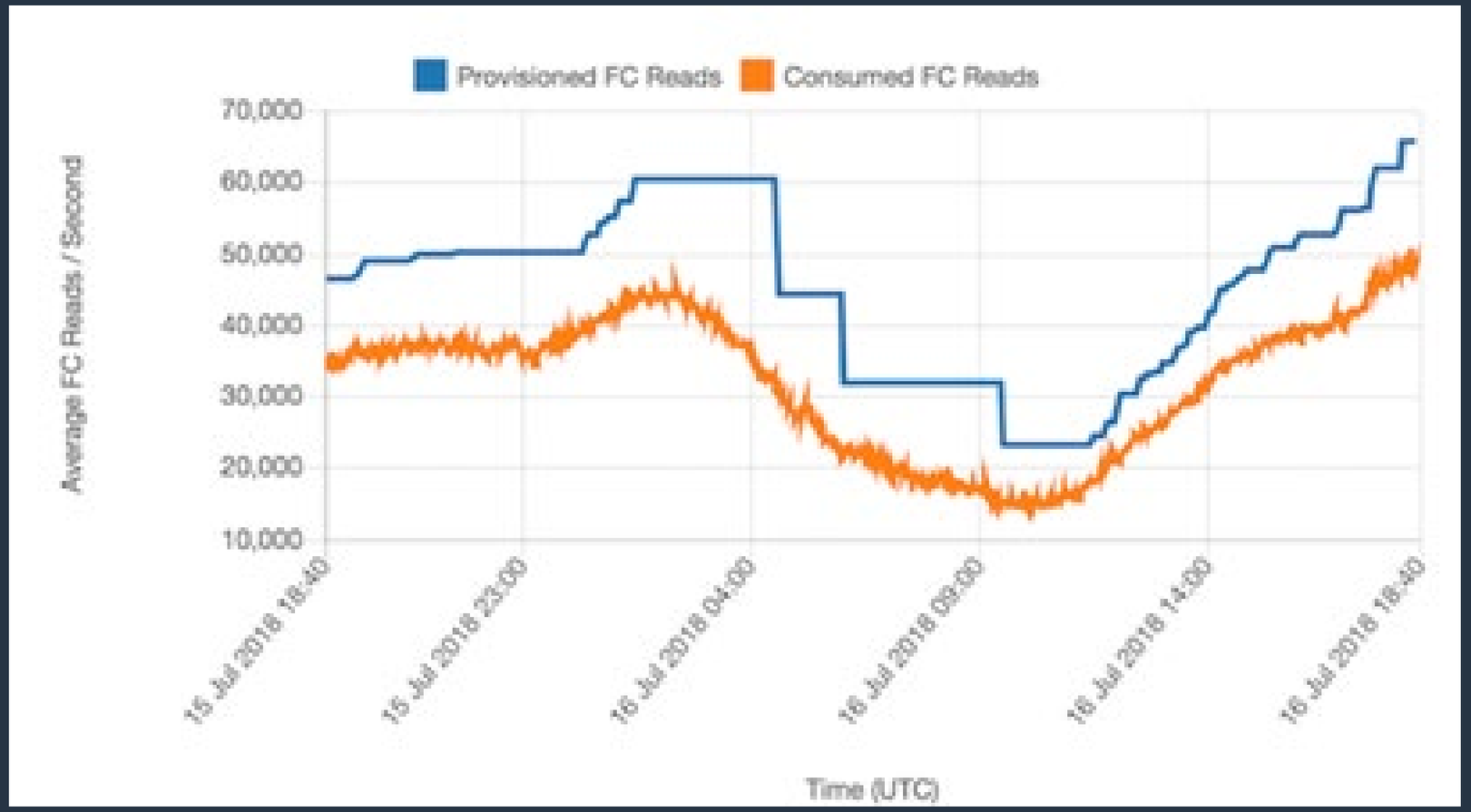
DynamoDBのキャパシティーモード

- 1KBからPBクラスまで変わらないパフォーマンスとコスト
- 2種類ありどちらかを選ぶ。切り替えることも出来る

- プロビジョンドキャパシティーモード
  - 時間あたりで使う分を設定するキャパシティーモード
  - Auto Scalingできる

- オンデマンドキャパシティーモード
  - 使った分だけ支払うキャパシティーモード
  - 最低0にまで落とせる

プロビジョンドキャパシティーモード

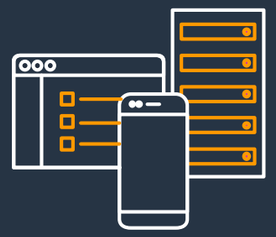


秒単位のキャパシティーユニットを指定

トラフィックが既知のワークロードに最適

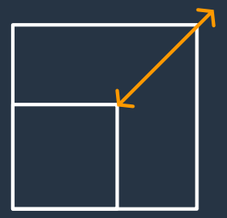
スケーリングポリシーによる自動化

オンデマンドキャパシティーモード



キャパシティ管理なし

使用すると予想される読み取り/書き込みスループットを設定する必要無し



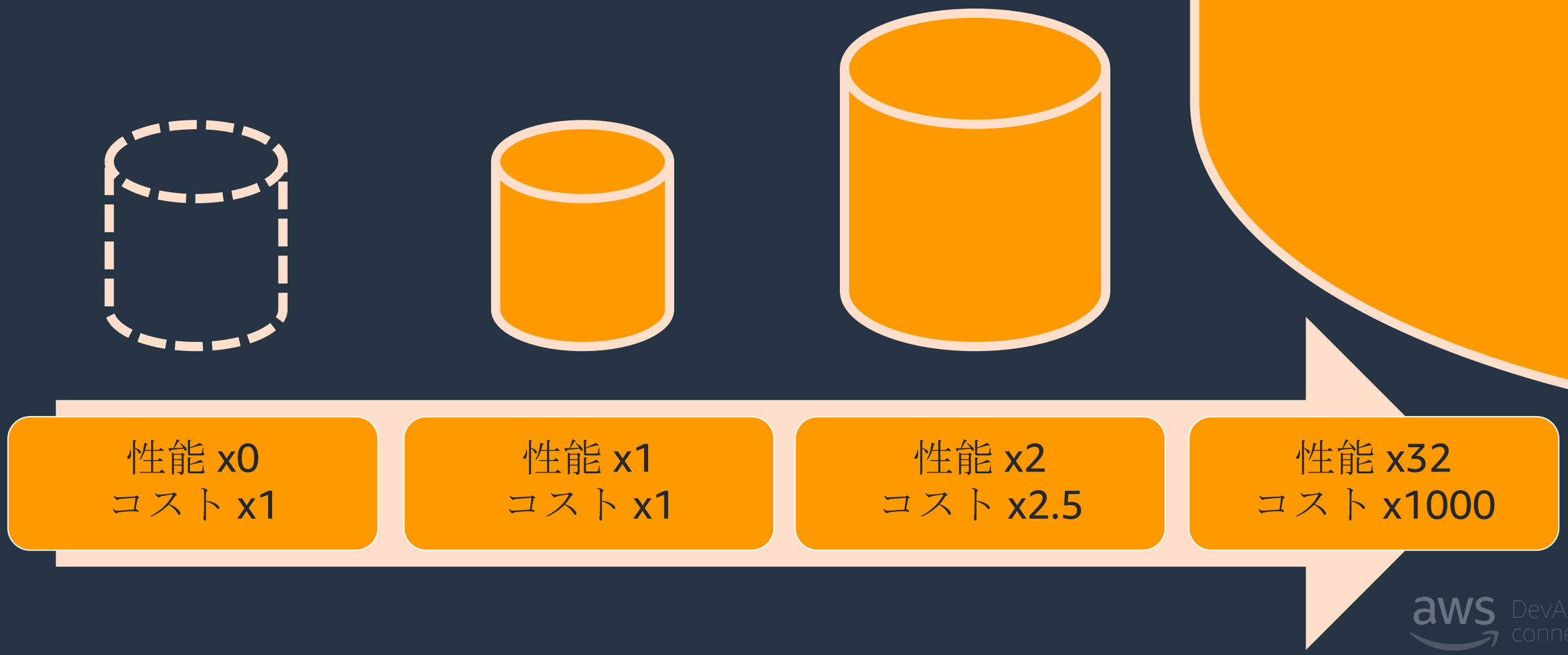
予測不可能なワークロードに最適

要求に応じて、1 秒あたりゼロから数万のリクエストに対応

使用した金額のみ支払う

リクエストごとの料金

サーバーの性能とコスト

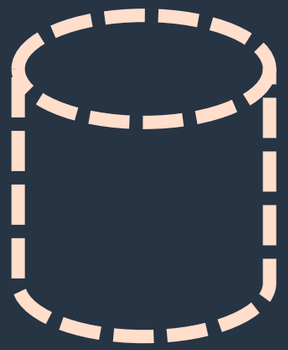


aws DevAx connect

サーバーの性能とコスト

DynamoDBの特長

- 1KBからPBスケールまで
  性能とコストが比例
- 最小キャパシティが極めて小さい



性能 x0
コスト x1

性能 x1
コスト x1

性能 x2
コスト x2.5

性能 x32
コスト x1000

aws DevAx connect

スケーラビリティまとめ

- スケーラビリティとは、コストと性能がパフォーマンス良く比例すること
- DynamoDBは、Internet Scaleのアプリケーションのみならずスモールスタートでもスケーラビリティを発揮できる

整合性とトランザクション

整合性の定義

- 読み書きしたデータに矛盾が無い状態
  - 単純なケースでは、書いたものが読める
- 複雑なケースはトランザクションによって管理される

トランザクションの定義

- レベルが色々あるが、目指すところは
  - 他のプロセスの影響を無視できる
  - 書いたものが読める、一度読んだものが読める
- かなり高級・高コストな機能

※参考

トランザクション分離レベル
SERIALIZABLE
REPEATABLE READ
READ COMMITTED
READ UNCOMMITTED

完璧なトランザクション？

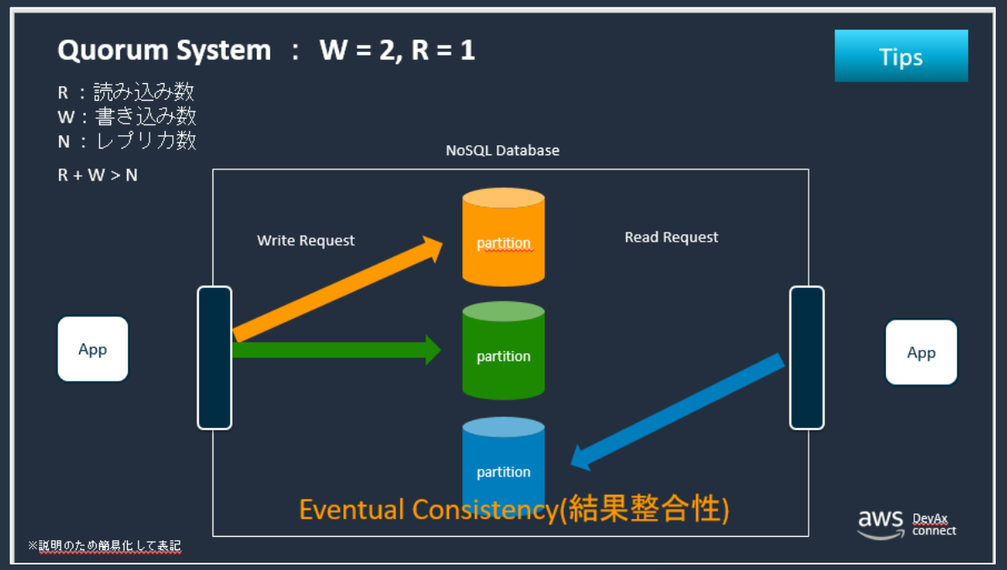
- 実はRDBでも完璧には出来ていない
  - デフォルト：READ COMMITTED、REPEATABLE READ
  - トランザクションレベルを意識してプログラムを書くケースってどれくらいありますか？
- RDB側でもトランザクションは色々進化していますが、ここでは触れません。

トランザクションの範囲

- トランザクションはデータベースの機能
- アプリケーションで出来ないとは言ってない
  - アプリケーションがちょっとだけ気にすれば解決する問題もある
- 複数のデータベースやマイクロサービスアーキテクチャ、非同期処理を採用すると難しくなってくる
  - Read replicaでも非同期反映になるものがある。その時トランザクションは？

DynamoDBの整合性

- 2つから選択できる
- 書いた後には必ず読める
  - 強い整合性(Strong Consistency)
- 書いてすぐには、古いものが返ってくる可能性がある
  - 結果整合性(Eventual Consistency)
  - しばらく後には普通に読めるようになる
  - 確率は3分の1、通常1秒以内
  - 必要なRead Capacity Unit半分

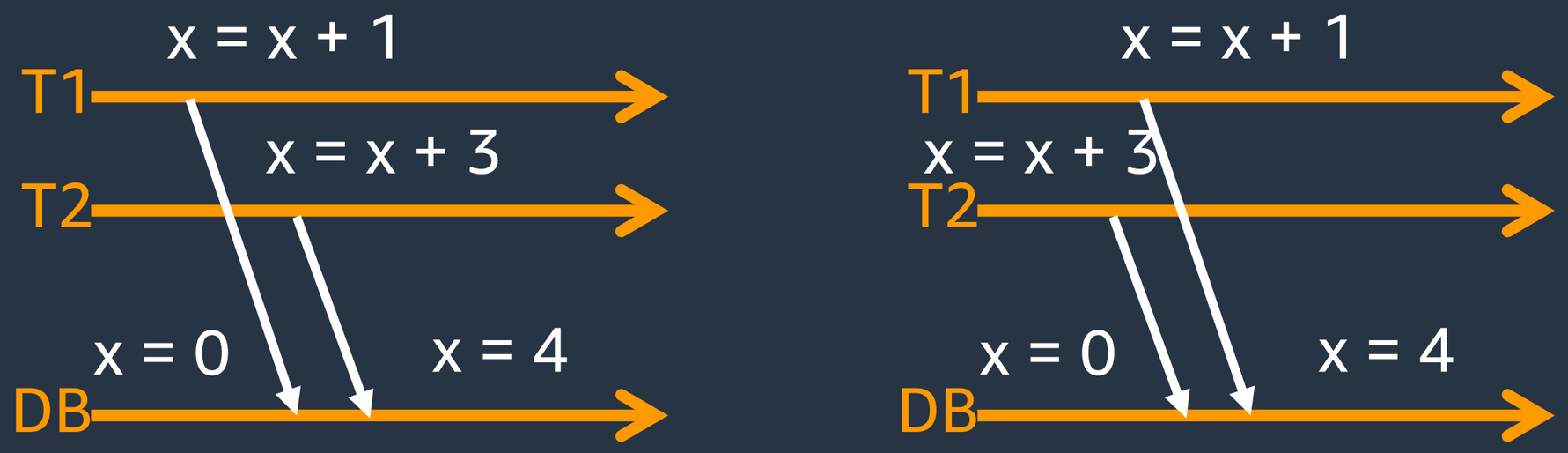


DynamoDBのトランザクション

- START TRANSACTIONやCOMMITが無いので、全ての操作がinstant
- そのような前提に立てば基本的なレベルはREAD COMMITTED
- トランザクション周りで使える機能がいくつかある

Atomic Counter

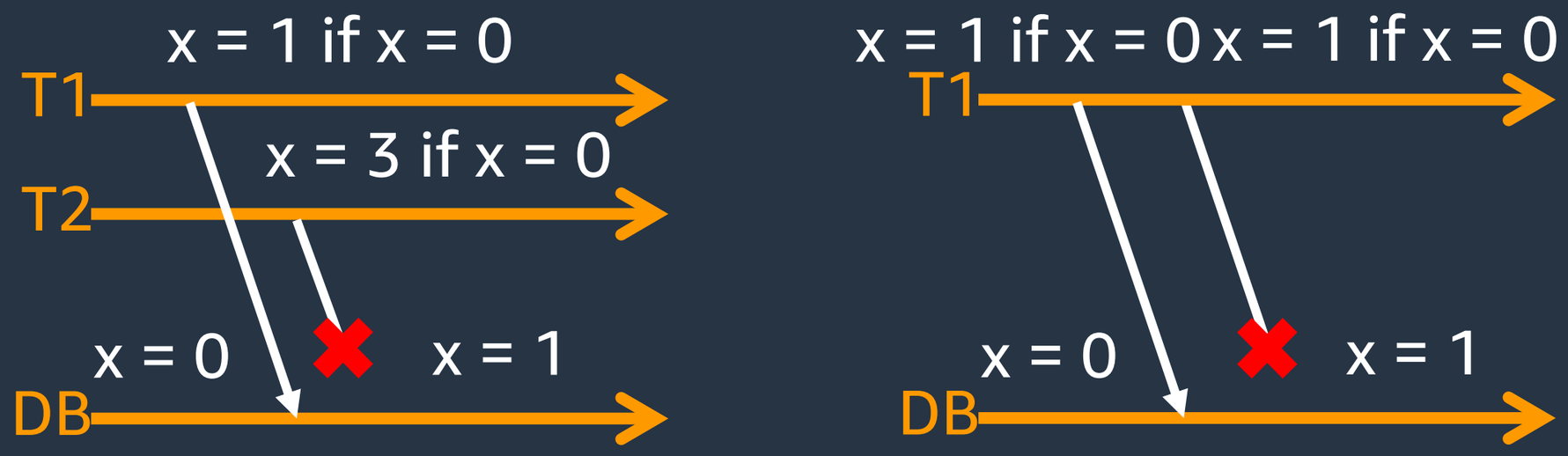
- Update WatchList SET watchCount = watchCount + 1 WHERE watchId = ‘xxx’;
- 仮にUpdateが競合しても更新内容が失われない



aws DevAx connect

Conditional Update

- Update WatchList SET watchCount = 4 WHERE watchId = 'xxx' AND watchCount = 3;
- conditionに失敗したらエラーとなる



aws DevAx connect

TransactWriteItems

- Conditional Updateを含んだ25までのリクエストをまとめて実行
- Conditional Updateのどれかに失敗したら全てabort。all or nothing
- SERIALIZABLE = この25個の間に割り込まれない

整合性が必要、トランザクションが必要、どこに必要？

- 複数テーブルの関係性を保ちたい
  - 例：Userテーブルには登録されているのに、Scoreテーブルには登録されてない！
  - 正規化ならではの悩み

→　読み込まれる形で非正規化しておけば、画面やサービスの単位では反映の前後があるかもしれないが、データの関係性が矛盾することはない

整合性が必要、トランザクションが必要、どこに必要？

- rollbackしたい
- rollbackということはuncommittedな期間に書き込んだものが存在する

→　READ COMMITTEDでエラー時ロールバックするなら、アプリケーション側に貯め込んでも似た動きが実現できる

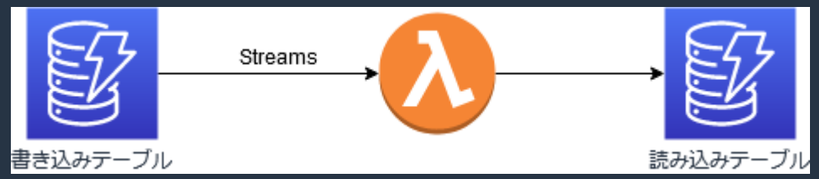
整合性が必要、トランザクションが必要、どこに必要？

- 更新が競合する可能性がある
- 本当にカラムレベルまで競合するのか？
  - 非正規化まで考えた時に個別の操作が同一カラムを更新しなければならないケースは少ない
  - あるのは大抵Aggregation、複数のデータを集計・集約するプロセス

→ Aggregationの競合は回避しなければならない

Aggregationの競合回避

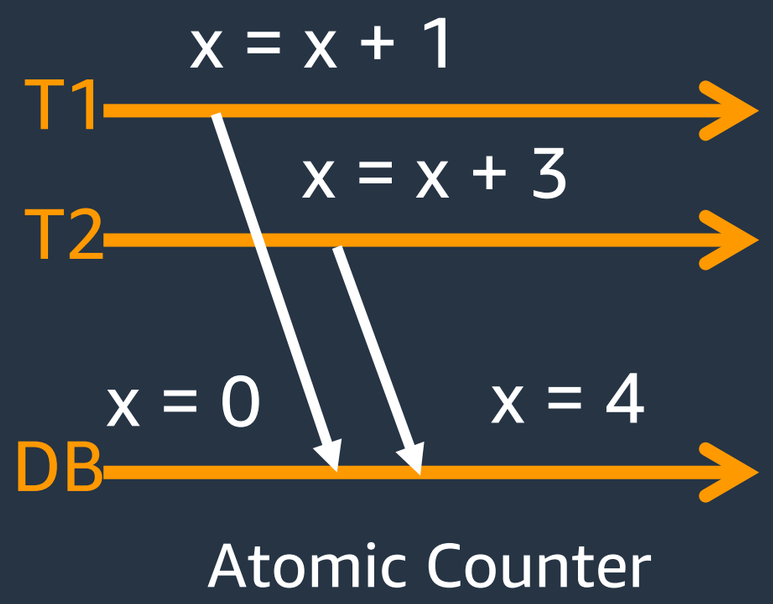
- 書き込みと重要な計算を同時に行わないように制御する
  - 例：日中書き込み、夜間バッチ処理
- 集約プロセスを直列化する
  - 例：書き込み用テーブルの変更をDynamoDB Streamsで受け、集約テーブルを更新。パーティション単位で直列化する



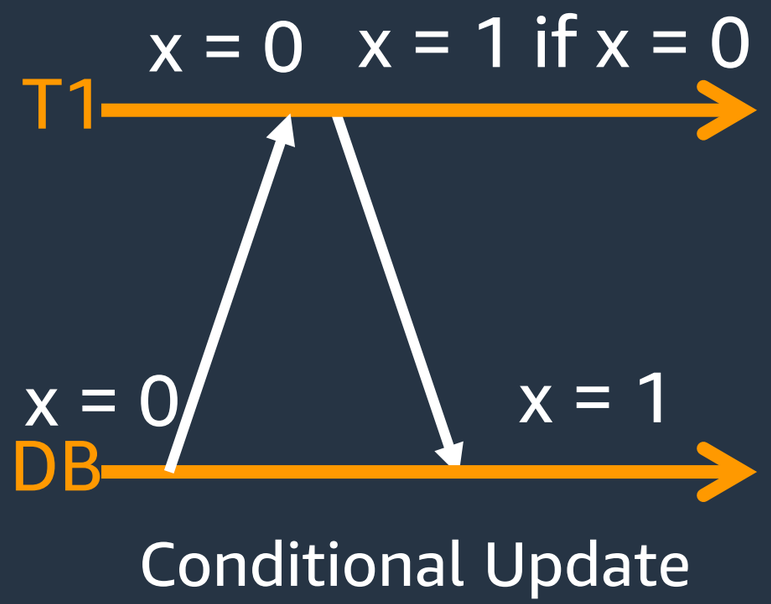
aws DevAx connect

Aggregationの競合回避

- Atomic Counterで競合しても適切に処理されるようにする
- Conditional UpdateやTransactWriteItemsを使い疑似的なSERIALIZABLEを確保する



Atomic Counter



Conditional Update

aws DevAx connect

整合性とトランザクションまとめ

- トランザクションが必要、で止まらず何に必要なのかを認識することはDynamoDBに限らず必要
- DynamoDBにもトランザクションに一部対応する機能が用意されている
  - トランザクションが必要なケースが多いAggregationにも対応できる

データモデリングと
アクセスパターン

aws DevAx connect

データモデリングの教科書的ベストプラクティス

- "教科書的"というのは例外もあり得るということ
- JOIN
  - アクセスパターンから非正規化を行う
  - Application JOINはしない
- Filtering, Sort
  - キー設計に組み込む
- 一度のリクエストで必要なデータを全て取得することを目指す

データモデリングの手順

- アプリケーションを理解する
- ER図を描く (optional)
- アクセスパターンを書き出す
- キーを設計する

アクセスパターンを書き出す

- 他のレイヤとのアクセス境界にはアクセスパターンが示されている
  - API-centric approach
  - UI-centric approach
  - Function-centric approach
- アクセスパターンを決めるのが何故難しく感じるか
  - サービス間連携、UI/UXの問題にまで関わってくる
- 難しく考えすぎずに良い
  - 今拡張性高く作るより、必要になってから変更した方がコストが低いことが多い

アクセスパターンが多すぎる！

- ここが一番陥りやすい
- そのアクセスパターンは本当に必要ですか？
  - 使われないアクセスパターンはないか？
  - レイテンシが増しても他のアクセスパターンで代用できるケースはないか？

アクセスパターンを定義する

- 何でもできるアクセスパターンを定義しない
- データモデリングの話だけではなく、UI/UXの向上にも繋がる
- 例：何でもソートできて何でもフィルタできるグリッド
  - RDBであっても何でもするにはコストがかかる
  - ユーザーが使うのはせいぜい数通りだったりする
  - よく使うアクセスパターンが分かれば作り込める
- 何でも出来るツールはExcelを超えられない

キー設計に関わるDynamoDBの基礎機能

- パーティションキー
  - 完全一致でしかデータを取得できない
- ソートキー
  - 前方一致を行うことができる
  - この順にデータが並んでいる
- フィルタ式
  - キー出ない項目についても絞り込めるが、しまう
- Global Secondary Index(GSI)
  - 同じテーブルに対して異なるアクセスパターンを実現する
  - 非同期で更新

Partition Key	Sort Key	Schema
DeviceID	DateTime	State
d#12345	2021-10-29T14:55:00	Normal
d#12345	2021-10-29T15:00:00	Normal
d#12345	2021-10-29T15:05:00	Warning1

aws DevAx connect

例：デバイスログテーブル

5分毎にデバイスの状態を記録するテーブル

状態を記録
稀にWarning

デバイス毎のID

日付時刻で
ソート・絞込み

Partition Key	Sort Key	Schema
DeviceID	DateTime	State
d#12345	2021-10-29T14:55:00	Normal
d#12345	2021-10-29T15:00:00	Normal
d#12345	2021-10-29T15:05:00	Warning1
d#23456	2021-10-29T14:55:00	Normal
d#23456	2021-10-29T15:00:00	Normal
d#23456	2021-10-29T15:05:00	Normal
d#54321	2021-10-29T14:55:00	Normal
d#54321	2021-10-29T15:00:00	Normal
d#54321	2021-10-29T15:05:00	Normal
d#65432	2021-10-29T14:55:00	Normal
d#65432	2021-10-29T15:00:00	Normal
d#65432	2021-10-29T15:05:00	Normal

アクセスパターン

特定のデバイスの状態を日付降順に取得

```
SELECT * FROM DeviceLog
WHERE DeviceID='d#12345'
ORDER BY DateTime DESC
```

Partition Key	Sort Key	Schema
DeviceID	DateTime	State
d#12345	2021-10-29T14:55:00	Normal
d#12345	2021-10-29T15:00:00	Normal
d#12345	2021-10-29T15:05:00	Warning1
d#23456	2021-10-29T14:55:00	Normal
d#23456	2021-10-29T15:00:00	Normal
d#23456	2021-10-29T15:05:00	Normal
d#54321	2021-10-29T14:55:00	Normal
d#54321	2021-10-29T15:00:00	Normal
d#54321	2021-10-29T15:05:00	Normal
d#65432	2021-10-29T14:55:00	Normal
d#65432	2021-10-29T15:00:00	Normal
d#65432	2021-10-29T15:05:00	Normal

アクセスパターン

特定のデバイスの警告ログを日付降順に取得

```
SELECT * FROM DeviceLog
WHERE DeviceID='d#12345'
AND State='Warning1'
ORDER BY DateTime DESC
```

Partition Key	Sort Key	Schema
DeviceID	DateTime	State
d#12345	2021-10-29T14:55:00	Normal
d#12345	2021-10-29T15:00:00	Normal
d#12345	2021-10-29T15:05:00	Warning1
d#23456	2021-10-29T14:55:00	Normal
d#23456	2021-10-29T15:00:00	Normal
d#23456	2021-10-29T15:05:00	Normal
d#54321	2021-10-29T14:55:00	Normal
d#54321	2021-10-29T15:00:00	Normal
d#54321	2021-10-29T15:05:00	Normal
d#65432	2021-10-29T14:55:00	Normal
d#65432	2021-10-29T15:00:00	Normal
d#65432	2021-10-29T15:05:00	Normal

SortKey条件を有効に使うための複合文字列key

特定のデバイスの警告ログを日付降順に取得

StateとDateTimeを
#で繋げてしまう

```
SELECT * FROM DeviceLog
WHERE DeviceID='d#12345'
AND BEGINS_WITH(
  State#DateTime, ‘Warning1#’)
ORDER BY State#DateTime DESC
```

Partition Key	Sort Key
DeviceID	State#DateTime
d#12345	Normal#2021-10-29T14:55:00
d#12345	Normal#2021-10-29T15:00:00
d#12345	Warning1#2021-10-29T15:05:00
d#23456	Normal#2021-10-29T14:55:00
d#23456	Normal#2021-10-29T15:00:00
d#23456	Normal#2021-10-29T15:05:00
d#54321	Normal#2021-10-29T14:55:00
d#54321	Normal#2021-10-29T15:00:00
d#54321	Normal#2021-10-29T15:05:00
d#65432	Normal#2021-10-29T14:55:00
d#65432	Normal#2021-10-29T15:00:00
d#65432	Normal#2021-10-29T15:05:00

アクセスパターン

特定のデバイスの状態を日付降順に取得

```
SELECT * FROM DeviceLog
WHERE DeviceID='d#12345'
AND BEGINS_WITH(
  State#DateTime, 'Warning1#')
ORDER BY State#DateTime DESC
```

```
SELECT * FROM DeviceLog
WHERE DeviceID='d#12345'
AND BEGINS_WITH(
  State#DateTime, 'Normal#')
ORDER BY State#DateTime DESC
```

Partition Key	Sort Key
DeviceID	State#DateTime
d#12345	Normal#2021-10-29T14:55:00
d#12345	Normal#2021-10-29T15:00:00
d#12345	Warning1#2021-10-29T15:05:00
d#23456	Normal#2021-10-29T14:55:00
d#23456	Normal#2021-10-29T15:00:00
d#23456	Normal#2021-10-29T15:05:00
d#54321	Normal#2021-10-29T14:55:00
d#54321	Normal#2021-10-29T15:00:00
d#54321	Normal#2021-10-29T15:05:00
d#65432	Normal#2021-10-29T14:55:00
d#65432	Normal#2021-10-29T15:00:00
d#65432	Normal#2021-10-29T15:05:00

新しいアクセスパターン

すべてのデバイスから特定のオペレーターのある期間のログ

ベーステーブル

Partition Key	Sort Key	Schema	
DeviceID	State#DateTime	Operator	DateTime
d#12345	Normal#2021-10-29T14:55:00	Liz	2021-10-29T14:55:00
d#12345	Normal#2021-10-29T15:00:00	Liz	2021-10-29T15:00:00
d#12345	Warning1#2021-10-29T15:05:00	Sue	2021-10-29T15:05:00
d#23456	Normal#2021-10-29T14:55:00	Liz	2021-10-29T14:55:00
d#23456	Normal#2021-10-29T15:00:00	Sue	2021-10-29T15:00:00
d#23456	Normal#2021-10-29T15:05:00	Liz	2021-10-29T15:05:00
d#54321	Normal#2021-10-29T14:55:00	Sue	2021-10-29T14:55:00

デバイスのOperator

日時

aws DevAx connect

新しいアクセスパターン

日時

デバイスのOperator

ベーステーブル

Partition Key	Sort Key
DeviceID	State#DateT
d#12345	Normal#202 29T14:55:0
d#12345	Normal#202 29T15:00:0
d#12345	Warning1#2 29T15:05:0
d#23456	Normal#202 29T14:55:0
d#23456	Normal#202 29T15:00:0
d#23456	Normal#2021-10-29T15:05:00
d#54321	Normal#2021-10-29T14:55:00

GSI-Operator

Partition Key	Sort Key	Schema	
Operator	DateTime	DeviceID	State#DateTime
Liz	2021-10-29T14:55:00	d#12345	Normal#2021-10-29T14:55:00
Liz	2021-10-29T14:55:00	d#23456	Normal#2021-10-29T14:55:00
Liz	2021-10-29T15:00:00	d#12345	Normal#2021-10-29T15:00:00
Liz	2021-10-29T15:00:00	d#65432	Normal#2021-10-29T15:00:00
Liz	2021-10-29T15:05:00	d#23456	Normal#2021-10-29T15:05:00
Liz	2021-10-29T15:05:00	d#65432	Normal#2021-10-29T15:05:00
Sue	2021-10-29T14:55:00	d#54321	Normal#2021-10-29T14:55:00
Sue	2021-10-29T14:55:00	d#65432	Normal#2021-10-29T14:55:00
Sue	2021-10-29T15:00:00	d#23456	Normal#2021-10-29T15:00:00
Sue	2021-10-29T15:00:00	d#54321	Normal#2021-10-29T15:00:00
Sue	2021-10-29T15:05:00	d#12345	Warning1#2021-10-29T15:05:00
Sue	2021-10-29T15:05:00	d#54321	Normal#2021-10-29T15:05:00

aws DevAx connect

アクセスパターン

すべてのデバイスから特定のオペレーターのある期間のログ

SELECT * FROM GSI–Operator

WHERE Operator = ‘Liz‘ AND

BETWEEN DateTime= ‘2021-10-29T15:00’ AND ‘2021-10-29T15:05’

ORDER BY DateTime DESC

Partition Key	Sort Key	Schema	
Operator	DateTime	DeviceID	State#DateTime
Liz	2021-10-29T14:55:00	d#12345	Normal#2021-10-29T14:55:00
Liz	2021-10-29T14:55:00	d#23456	Normal#2021-10-29T14:55:00
Liz	2021-10-29T15:00:00	d#12345	Normal#2021-10-29T15:00:00
Liz	2021-10-29T15:00:00	d#65432	Normal#2021-10-29T15:00:00
Liz	2021-10-29T15:05:00	d#23456	Normal#2021-10-29T15:05:00
Liz	2021-10-29T15:05:00	d#65432	Normal#2021-10-29T15:05:00
Sue	2021-10-29T14:55:00	d#54321	Normal#2021-10-29T14:55:00
Sue	2021-10-29T14:55:00	d#65432	Normal#2021-10-29T14:55:00
Sue	2021-10-29T15:00:00	d#23456	Normal#2021-10-29T15:00:00
Sue	2021-10-29T15:00:00	d#54321	Normal#2021-10-29T15:00:00
Sue	2021-10-29T15:05:00	d#12345	Warning1#2021-10-29T15:05:00
Sue	2021-10-29T15:05:00	d#54321	Normal#2021-10-29T15:05:00

GSI-Operator

新しいアクセスパターン

特定のスーパーバイザ のエスカレートされたすべてのログを取得

ベーステーブル

スーパーバイザ

Partition Key	Sort Key	Schema		
DeviceID	State#DateTime	Operator	DateTime	EscalatedTo
d#12345	Normal#2021-10-29T14:55:00	Liz	2021-10-29T14:55:00	
d#12345	Normal#2021-10-29T15:00:00	Liz	2021-10-29T15:00:00	
d#12345	Warning1#2021-10-29T15:05:00	Sue	2021-10-29T15:05:00	
d#23456	Normal#2021-10-29T14:55:00	Liz	2021-10-29T14:55:00	
d#23456	Normal#2021-10-29T15:00:00	Sue	2021-10-29T15:00:00	
d#23456	Normal#2021-10-29T15:05:00	Liz	2021-10-29T15:05:00	
d#54321	Normal#2021-10-29T14:55:00	Sue	2021-10-29T14:55:00	

aws DevAx connect

新しいアクセスパターン

特定のスーパーバイザ のエスカレートされたすべてのログを取得

ベーステーブル

Partition Key	Sort Key	Schema		
DeviceID	State#D			
d#12345	Normal			
d#12345	Norma			
d#12345	Warning 29T15:05:00		29T15:05:00	
d#23456	Normal#2021-10-29T14:55:00	Liz	2021-10-29T14:55:00	
d#23456	Normal#2021-10-29T15:00:00	Sue	2021-10-29T15:00:00	
d#23456	Normal#2021-10-29T15:05:00	Liz	2021-10-29T15:05:00	
d#54321	Normal#2021-10-29T14:55:00	Sue	2021-10-29T14:55:00	

スーパーバイザ

GSI-Supervisor

Partition Key	Sort Key	Schema		
Escalated To	State#DateTime	Operator	DateTime	DeviceID
Sara	Normal#2021-10-29T15:05:00	Sue	2021-10-29T15:05:00	d#54321

aws DevAx connect

データモデリングとアクセスパターンまとめ

- アクセスパターンを想定するのは、恐らくRDBでの開発作業と最も違う部分であり、迷う部分
- どの道、マイクロサービスアーキテクチャの導入やサービス分割を行うと、アクセス境界にアクセスパターンの定義が必須となる
- キー設計はトレードオフ。そのためにアクセスパターン等のアプリケーションの知識が必要となる
- UI/UXにも関わっている部分であり、DynamoDBがなくともやっておいて損はない

OLTPとOLAP

定義

- 正確な定義については触れない
- OLTP
  - 並列性を重視し、小さいデータ要求に迅速に応答する
- OLAP
  - 大量データを様々なアスペクトから分析する
  - DynamoDBには向かない分野

aws DevAx connect

分析用途におけるDynamoDB

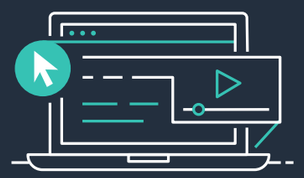
- OLAPの前後でDynamoDBを使うケースはある
- 例
  - DynamoDBからAmazon Redshiftにロード
  - Amazon Athenaからフェデレーテッドクエリで DynamoDBにアクセス
  - DynamoDBからS3にエクスポートしたデータにAthenaからアクセス
  - DynamoDBのデータをAWS Glue Elastic Views (Preview)を通してAmazon OpenSearch Serviceに投入し、Amazon QuickSightでBIツールとして使う

ラーニングパス

aws DevAx connect

AWS Training と認定でデータベースを学ぶ

AWS のエキスパートが作成した教材でデータベーススキルの習得とチェック



25 以上ある無料のデジタルトレーニングでは、リレーショナルデータベースとノンリレーショナルデータベースサービスをカバー

クラスルームでは、AWS 上のデータベースのプランニングや設計を、AWS のエキスパートによる機能紹介やハンズオン演習を通して実施

AWS 認定 データベース – 専門知識　でスキルを検証

データベースの学習パスはこちら　aws.amazon.com/training/path-databases

amazon-dynamodb-labs.com

aws

Search...

Bank Payments References

Workshop: Hands-on Labs for Amazon DynamoDB

Workshop: Modeling Game Player Data with Amazon DynamoDB

Workshop: Advanced Design Patterns for Amazon DynamoDB

Scenarios: Design Challenges

- Retail Cart Scenario
- Bank Payments Scenario
  - Bank Payments References

Links: NoSQL Design: Reference Materials

Contributing (GitHub)

Authors

Clear History

Amazon DynamoDB Labs > Design Challenges > Bank Payments Scenario

Edit this page

BANK PAYMENTS SCENARIO

NoSQL Design: Bank Payments Scenario

A bank has asked you to develop a new backend system to handle their scheduled payments.

This is primarily an OLTP workload with daily batch processes. Items in the table(s) represent payments that are scheduled between accounts. As items are inserted, they are scheduled on a specific date to have the payment processed. Each day, items are regularly sent to a transactional system for processing, at which time their status changes to `PENDING`. Upon a successful transaction, an item's status is set to `PROCESSED` and updated with a new transaction ID.

Workload dimensions

- Accounts can have multiple payments scheduled for any day in the future.
- Payments have the following data fields: *AccountID*, *ScheduledDate*, *Status* (`SCHEDULED`, `PENDING` or `PROCESSED`), *DataBlob* (total item size is <= 8 KB)
- One million automated scheduled payments are added every day at 1:00 AM *for that day*, which need to complete in 30 minutes.
- One million payments are added every day with the `SCHEDULED` state, mostly in the hours of 6 AM to 6 PM.
- During the day, a batch job runs regularly to query for today's `SCHEDULED` payments. This service sends the `SCHEDULED` items to the transaction service. Upon sending the items to the transaction service, the payment status is changed to `PENDING`.
- When the transaction service completes, the status of an item is changed to `PROCESSED` and a new transaction ID is

DevAx connect

まとめ

まとめ

- 本セッションではDynamoDB導入時によくある懸念について解説した
  - こういった懸念はデータモデリングやアーキテクチャで大体クリアできる
- DynamoDBのメリットがビジネスやシステムに合っているかを確認する方が、より重要
- DynamoDBの選択ポイントは
  - サーバーレスで運用負荷が少ない
  - 一貫した低レイテンシ
  - 高いスケーラビリティ

今回話さなかった部分

- DynamoDBの詳細なアーキテクチャ
- 各種機能
- パーティショニング
- キャパシティープランニング
- 他データベースからの移行
- DynamoDBテーブル間の移行
- バージョンアップ戦略
- 他サービスとの連携
- セキュリティベストプラクティス
- マルチテナンシー
- etc...

Thank you!

Tsutsumi, Hayato
Senior Solutions Architect

aws DevAx connect