LET'S BECOME THE MASTER OF DYNAMODB DATA MODELING TO BECOME THE MASTER OF SERVERLESS

MASASHI TERUI @ AWS DEV DAY TOKYO 2018

MASASHI TERUI
ARCHITECT / DEVELOPER

/SERVERWORKS CO.,LTD.
+ FREELANCER

- **Serverless Oji-san**
- Serverless Framework Plugin Developer
- **Serverlessconf Tokyo 2016,2017,2018 speaker**
- Remote worker (in Sapporo-shi, Hokkaido)
- **http://marcy.hatenablog.com/entry/2018/08/04/231241**

⚠⚠⚠⚠⚠⚠⚠⚠⚠⚠⚠⚠

このセッションは
SERVERLESS要素が
ほとんどありません！！

⚠⚠⚠⚠⚠⚠⚠⚠⚠⚠⚠⚠

/SERVERLESSなんて飾りです

偉い人にはそれがわからんのですよ

ここは一足先に(勝手に)Databaseトラックです！！

DynamoDB使ってますか？

/SERVERLESSやってますか？

DynamoDBは好きですか？

- 好き 😁
- 普通 😌
- 嫌い 😰
- LambdaとRDSの相性が悪いから仕方なく使っている

LambdaとRDSの相性が悪い？

- コネクションモデル
  - 永続的
- セキュリティ
  - 組み込みの弱い認証
  - **VPC内へのアクセスコスト（ENI生成処理）**

とてもOLTPでは使えない(数秒〜十数秒)

本当はRDBを使いたいけど

/仕方がないからDynamoDB……

そんなマインドで大丈夫？

/DynamoDBを積極的に使おう

/DynamoDB Best Practices

- 端的に言ってバイブル
- これを読んで全て理解できた人は
  たぶんこの先を聞く必要はありません！！
- https://docs.aws.amazon.com/ja_jp/amazondynamodb/latest/developerguide/best-practices.html

DynamoDBのよくある不安

- テーブル設計とかインデックス設計ってどうやるの？
- ACIDトランザクション無いのに大丈夫なの？
- 結果整合性って大丈夫なの？
- 上手く性能を引き出すにはどうすれば良い？

/DynamoDBのよくある不安

- テーブル設計とかインデックス設計ってどうやるの？
- ACIDトランザクション無いのに大丈夫なの？
- 結果整合性って大丈夫なの？
- 上手く性能を引き出すにはどうすれば良い？

分からないから不安になる

DynamoDBの基本

- あえて分類するなら分散KVSというジャンルのNoSQL
- Keyによる完全一致とIndex探索が可能
- フルマネージド
- 無制限の自動拡張ストレージ（使った分だけ）
- インスタンスの管理不要、Write/Readの性能だけを指定する

ではなく

/DynamoDBはどんな ~~“AWSサービス”~~ か

知ってますか？

/DynamoDBはどんな“データベース”か

/Amazon's Dynamo

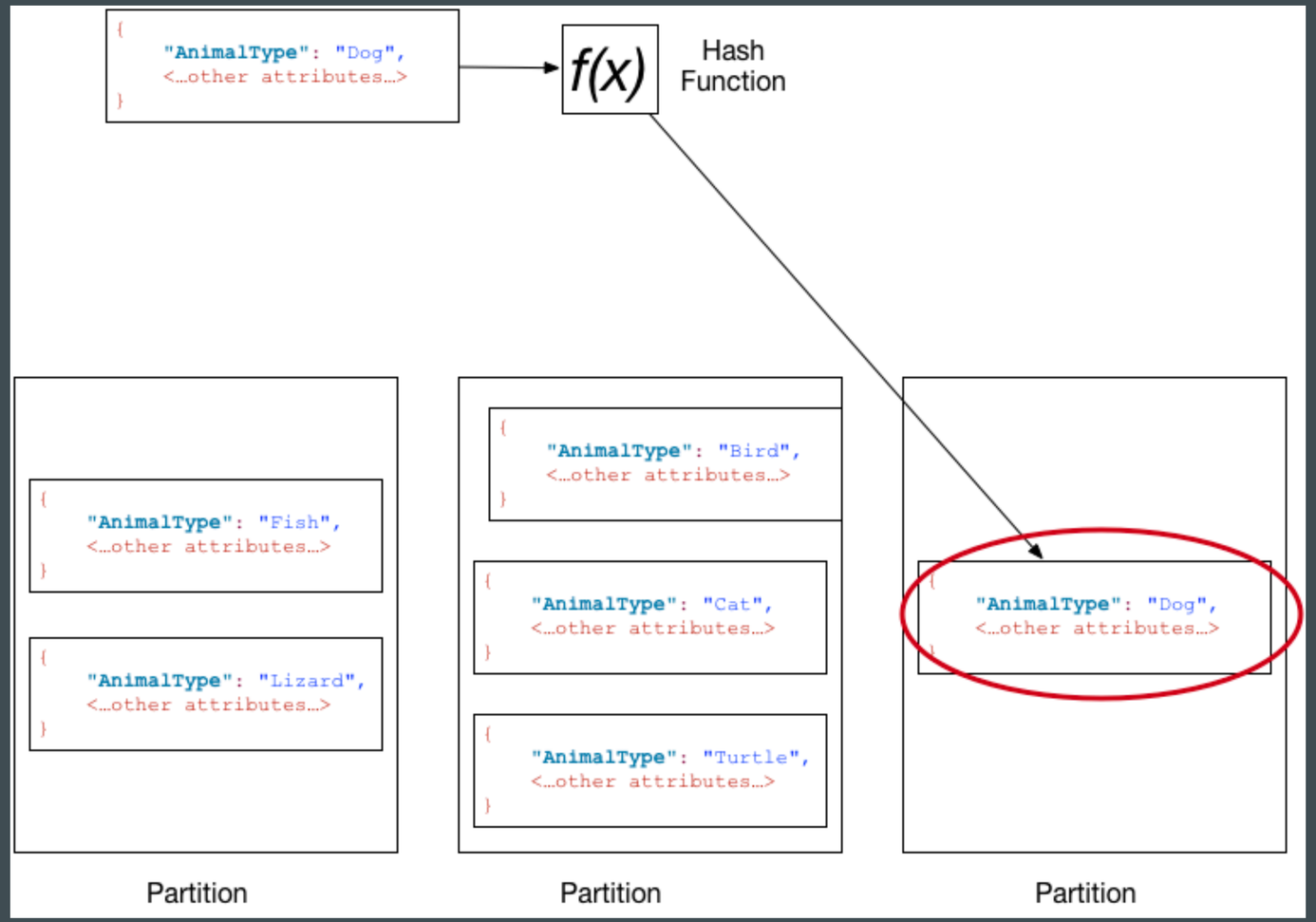
- Consistent Hashing
- Vector clocks with reconciliation during reads
- Sloppy Quorum and hinted handoff
- Anti-entropy using Merkle trees
- Gossip-based membership protocol and failure detection

詳しく知りたいなら（BetterであってMustではない）

https://www.allthingsdistributed.com/2007/10/amazons_dynamo.html

抑えるべきポイント

- キーのハッシュ値で分散
- バックグランドで更新を後勝ちで同期
  - 結果整合性ベース
- 書き込み・読み込みのQuorum数で整合性が決まる
  - キーを指定した読み込みは強整合性が選択可能



/Partition Keyのハッシュ値で分散

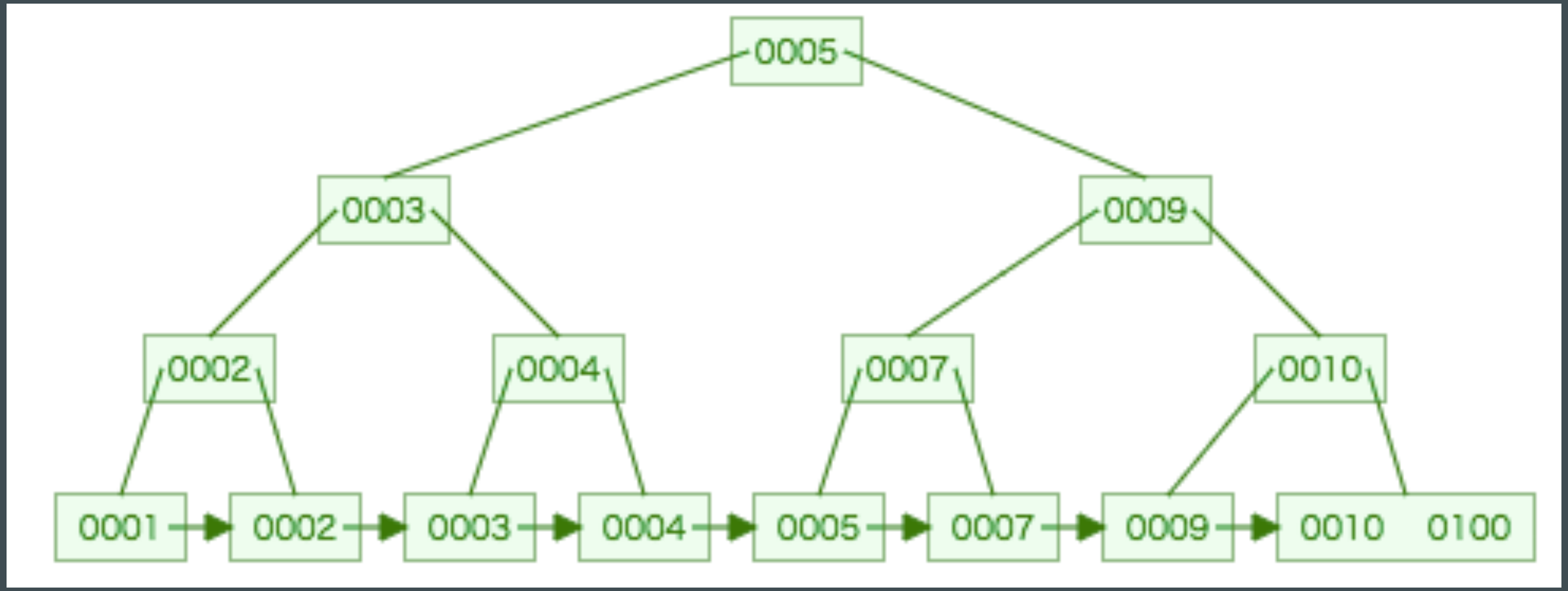
Like Hash Table(Index)

この前提を踏まえると

- Partition Keyに連番を使うのは何の意味もないことが分かる
  - 連番カウンターを作るとPartitionが偏る
  - Max + 1なんて以ての外
- 論理的に一意となる属性があるならそれを使う
- なければUUIDのような衝突可能性が極小なランダム値で良い
  - SortもできないのでSnowflakeのような順序性も要らない
- User IDはCognitoで認証しているならCognitoが発行するIDを使う
  - アクセス制御にも使える

Partitionごとのキャパシティ

- 3000RCU, 1000WCU, Index Size 10GB (2018-10-31現在)
- どれか一つでも超過するとPartitionが分割される
- 分割されると確保したユニットが均等に割り振られる
- スロットリング時の一時的なバーストがある
  - 限定的なバースト
  - 余剰分を回す (Adaptive Capacity)



/B+TREE INDEX

B-treeなのかB+treeなのかは公表されていないけど
Range Keyとか呼ぶくらいだから...ね？

Customer(Partition Key)	Date(Sort Key)	Status	...
Alice	2017-01-01	IN_PROGRESS	...
Bob	2017-01-02	IN_PROGRESS	...
Bob	2017-02-11	PENDING	...
Bob	2017-11-01	END	...

Customer(Partition Key)	StatusDate(Sort Key)	...
Alice	IN_PROGRESS#2017-01-01	...
Bob	IN_PROGRESS#2017-01-02	...
Bob	PENDING#2017-02-11	...
Bob	END#2017-11-01	...

/TARGETING QUERY

B+treeの構造を知っていれば

組み合わせキーのインデックスが無くても

こうやれば良いことが分かる

このイメージを持つことがとても重要

/Partition Keyで分散した中でSort Keyで整列

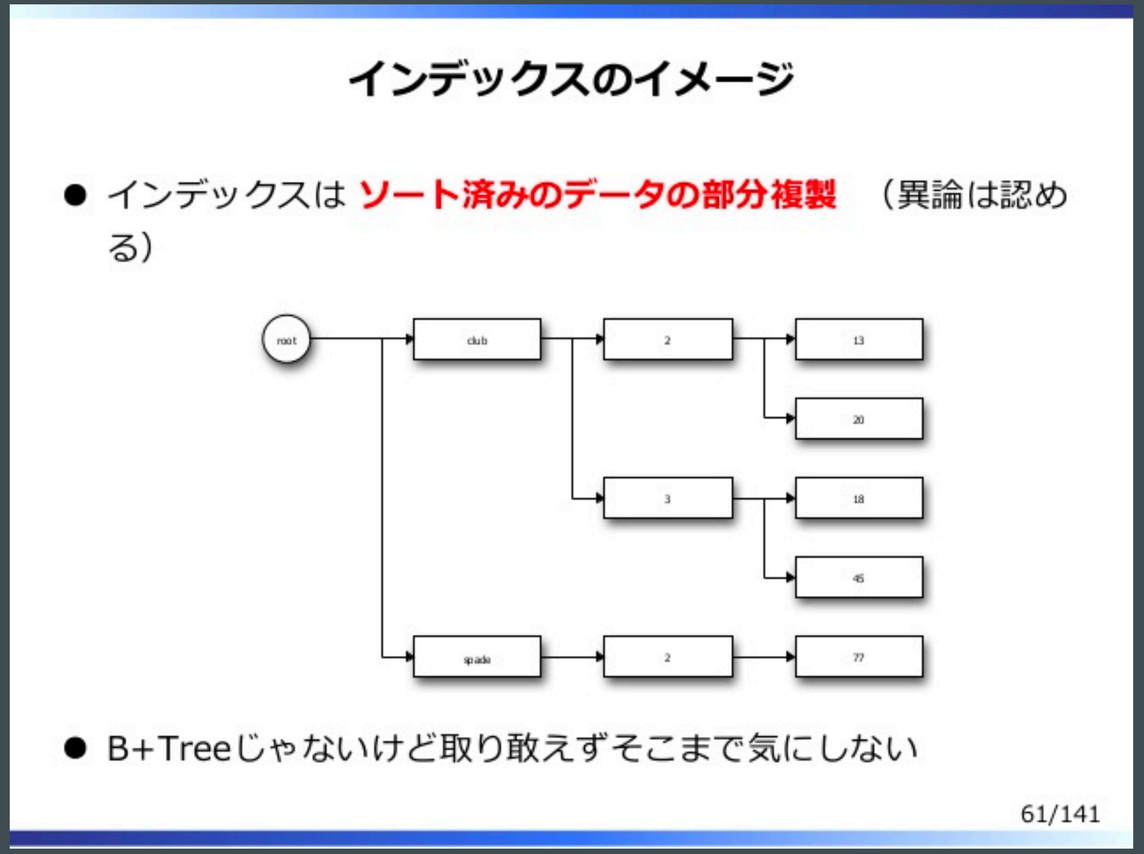
Primary Sort Key, LSI

- Partition Keyで分散したインスタンスにIndexが構築される
- 一つのインスタンス内では同期的にIndexも更新される
  - 強整合性が選択可能
- Primary Sort Keyのリーフノードに実データのアドレスが格納されているイメージ（と思われる）
- LSIは実データを持たないので重複可能

/GSIは実テーブルの射影

/射影

- 集合論における射影
- GSIのPartition Keyを元に分散
  （更新は非同期で適用）
- リーフノードに属性とPrimary Key
  （詳しくは後述）
- ソート済みのデータの部分複製
  （異論は認める）



雑なMySQLパフォーマンスチューニング from yoku0825
https://www.slideshare.net/yoku0825/mysql-57449062/62

GSIが射影する属性のオプション

- KEYS_ONLY: Primary Keyの値のみ
- INCLUDE: Primary Keyの値 + 選択した属性
- ALL: 全ての属性
- GSIに含めた属性はPrimary Keyを引き直す必要がなくなる
  - 読み込みの効率化
  - 更新コストとのトレードオフ
  - MySQLやPostgreSQLで言うCovering Index / Index Only Scanのイメージ

GSIに対するクエリは結果整合性

- メインテーブルとは別のPartitionで分散される
- 更新は非同期で反映
- 結果整合性って大丈夫なの？
- RDSだってRead Replica読んだら結果整合性ですよ？？
  - GSIへのクエリはだいたいRead Replicaに投げるようなクエリ

全部RDBで既にあった仕組みですよ？？

仕組みを知れば実にシンプル

✕ DynamoDBは癖がある

○ RDBの癖がありすぎる

→シンプルすぎるDynamoDBとのギャップが強くてそう感じるだけ

RDBに洗脳されているんじゃないですか？？

つまるところ

DynamoDBのデータモデリングとは

異論は認める！！

データの配置を決めること

配置を決めるための

データモデリングの考え方のポイント

テーブルの分け方

- スキーマレスの意味を考える
  - List, Mapが持てること？
  - 後から属性をシームレスに追加できること？
  - A. なんでも突っ込めること
- Partition Keyで分散
  - テーブルを分けなくても負荷は分散する
  - 極論すると分ける意味がない、分けない方がキャパシティが管理しやすい
  - “設計が優れたアプリケーションでは、必要なテーブルは 1 つのみです”
  - GSIの個数制限(5個)などもあるので業務ドメイン毎に一つくらいの気持ち

RDBとはアプローチが全く異なる

- RDB
  - まずスキーマを考える
  - 正規化
  - それに対してどうアクセスするか(SQL)考える
- DynamoDB
  - スキーマレス
  - 非正規化
  - まずアクセスを考えて、それに合わせたデータを作る
  - アプリケーションとデータモデルは同時に設計する

とはいえ

/ACIDトランザクションが無いのはどうするの？

そもそもACIDってなんだっけ？

- 原子性 (Atomicity)
- 一貫(整合)性 (Consistency)
- 独立性 (Isolation)
- 永続性 (Durability)

これらを担保することが目的であって
トランザクションシステムである必要は無い

ACIDの担保

- RDB
  - 正規化していくとテーブルの数は基本的に増える
  - 論理的に分かれたテーブル(レコード)間の整合性を取る仕組みがある
    - リレーションシップ
    - 外部キー制約
    - ACIDトランザクション
- DynamoDB
  - 非正規化して1つの実体について1つのアイテムに収める
  - 1アイテムの更新はアトミック

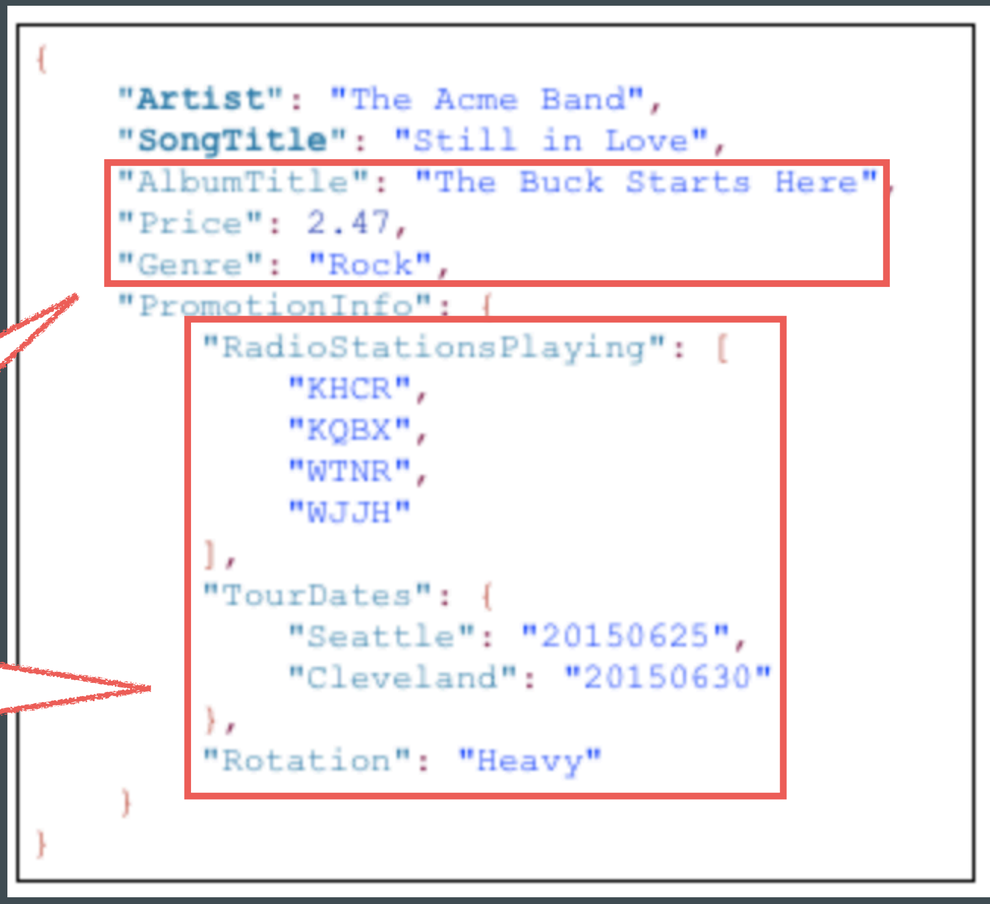
DynamoDBの更新操作

- PutItem
  - アイテムをまるごと更新する、無かったら作る
  - Get -> 書き換え -> Putだと独立性が崩れる
- UpdateItem
  - アイテムを部分更新、無かったら作る
  - List, Mapの部分追加、削除なども可能
  - 属性の値を利用した更新も可能（ある属性に+1する等)
  - 属性の値を利用した条件付き書き込み

非正規化すると読み込みがつらい

全部GSI貼るの？

フルスキャンするしかない

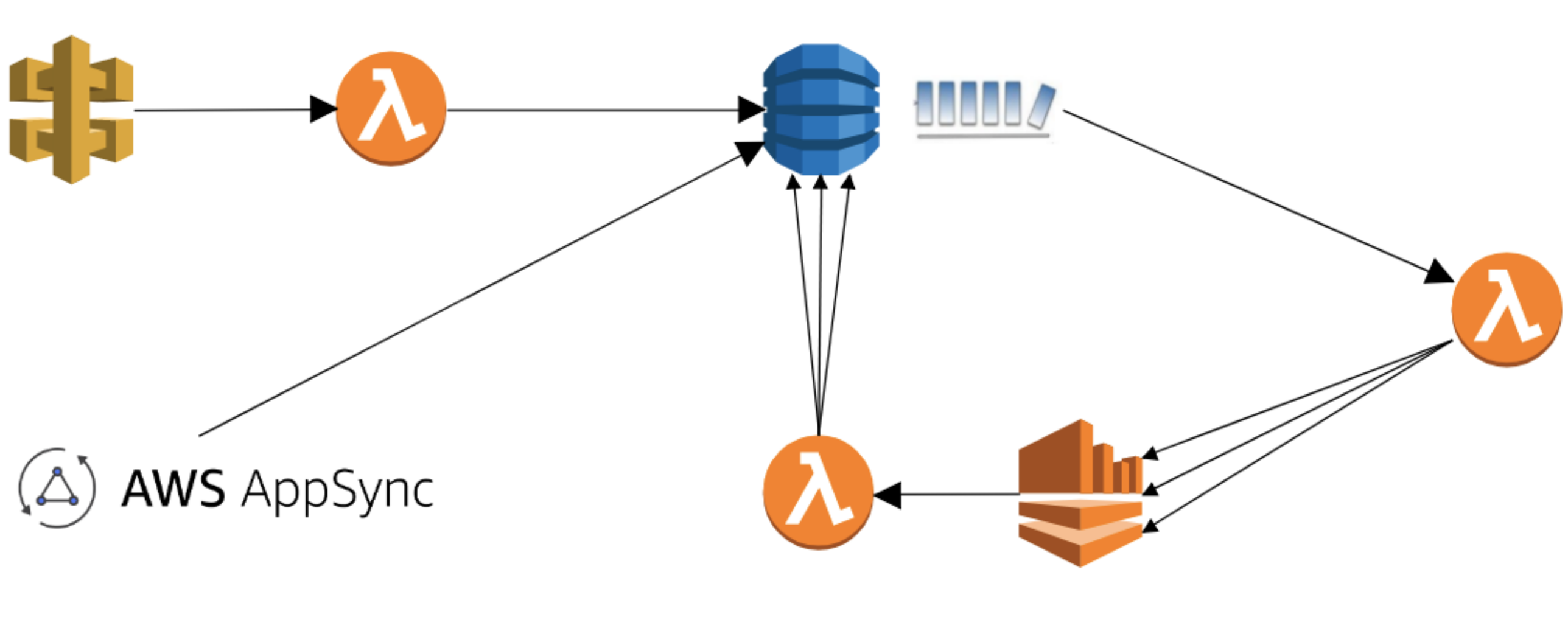


そこで

/CQRSですよ

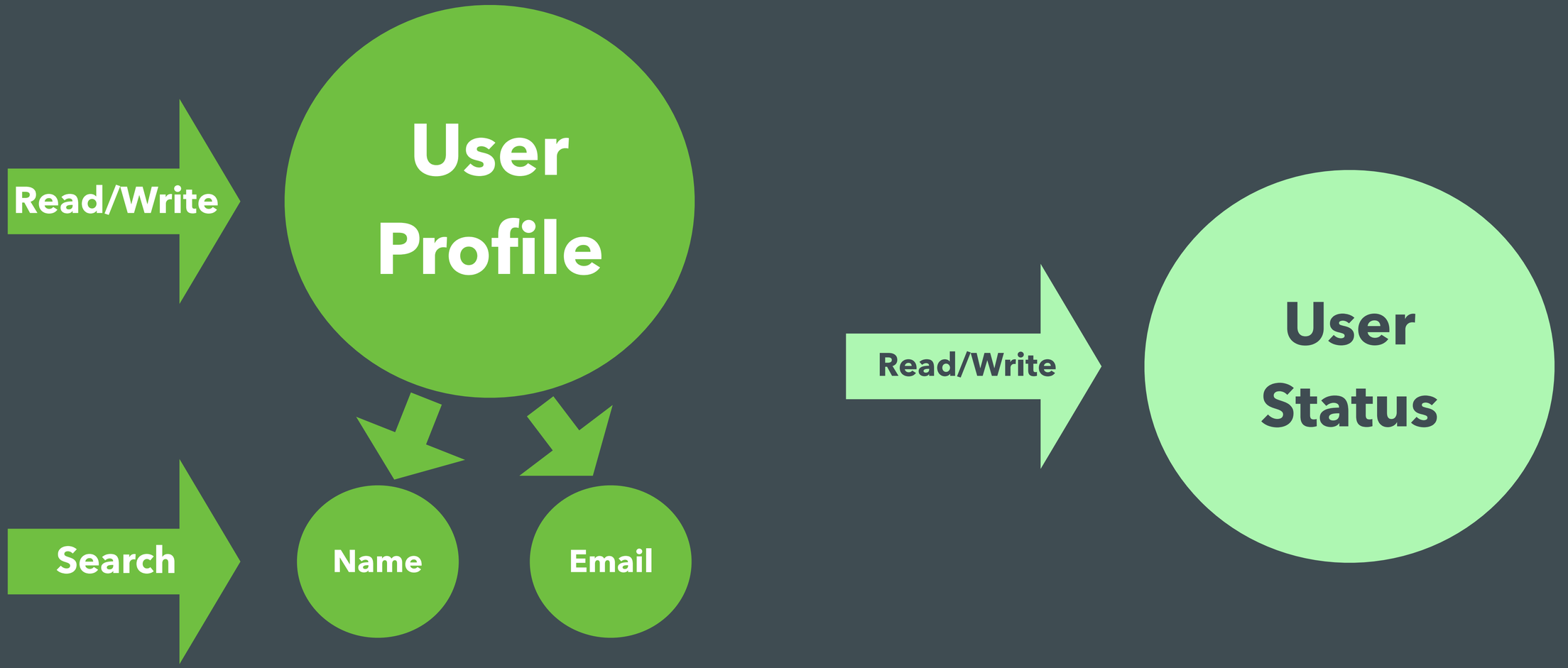
/CQRS (Command Query Responsibility Segregation)

- 書き込むデータと読み込むデータは同じである必要は無い
  - 結果整合性を受け入れて非同期で読み込みデータを作れる
- Commandの完了をもってQuery用データを作れる
  - DynamoDB Streamsがマッチする
  - CQRSを素直に実装するとまずKinesis Streamsだけど
  - Materialized View



/ARCHITECTURE PATTERN

AppSyncがこれから重要な位置を占めるはず



/DATA ATOMICITY

Microservies的なイメージではこの原単位が重要
データストア・テーブルの分け方ではない
跨がせたいならMaterialized Viewを作る

	GSI-PK	GSI-SK		
id (PK)	key (SK)	value	name	email
48178de7-884b-4349-a6be-bb23efe83dd7	profile		Terui	terui@willy.works
48178de7-884b-4349-a6be-bb23efe83dd7	status	active		
48178de7-884b-4349-a6be-bb23efe83dd7	name	Terui		
14c7eb12-9a19-4f17-9df3-4b243ff78452	profile		Takagi	takagi@serverworks.co.jp
14c7eb12-9a19-4f17-9df3-4b243ff78452	status	inactive		
14c7eb12-9a19-4f17-9df3-4b243ff78452	name	Takagi		

↓ ↓

key (PK)	value(SK)
status	active
status	inactive
name	Takagi
name	Terui

/こんなイメージ

1つのGSIで様々な検索ができる

/とはいえ限界はある

- 複数の実体に跨るトランザクション
  - DynamoDB単体だとここはどうしてもつらい
- トランザクション分離レベルの最高はSelializable
  - つまり直列化
  - Kinesis Streamsで直列化
  - コケるかもしれないので冪等に実装する
  - 条件付書き込みで既に行った処理をSkipするのも冪等
- というか、Kinesis Streamsで同時接続数をコントロールすればRDS使ったって良いわけで...

いくつかご紹介

/ADVANCED TIPS

GSIのシャーディング

- Partition Keyのパターンが少ない場合
  - 分散しなくなってしまう
- 1Partitionのキャパシティを越えそうならシャーディングする
- Key以外の自明な属性から計算可能なSuffixを付与する
  - 集計や範囲検索の必要がなければHash
  - 必要があればMod等

	GSI-PK	GSI-SK		
id (PK)	key (SK)	value	name	email
48178de7-884b-4349-a6be-bb23efe83dd7	profile		Terui	terui@willy.works
48178de7-884b-4349-a6be-bb23efe83dd7	status	active		
48178de7-884b-4349-a6be-bb23efe83dd7	name-161	Terui		
14c7eb12-9a19-4f17-9df3-4b243ff78452	profile		Takagi	takagi@serverworks.co.jp
14c7eb12-9a19-4f17-9df3-4b243ff78452	status	inactive		
14c7eb12-9a19-4f17-9df3-4b243ff78452	name-181	Takagi		

↓ ↓

key (PK)	value(SK)
status	active
status	inactive
name-161	Terui
name-181	Takagi

/こんなイメージ

Unicode Pointを合計して200で

割った余りを使用している

Sparse Index

- DynamoDBのIndexはSparse
- Index属性が存在しない場合、Indexにそのデータは含まない
- 特定のアイテムのみに存在する属性を指定することで
  Index容量が大きく圧縮できる

	GSI-PK		GSI-SK
user_id (PK)	event_id (SK)	score	champ
1111-1111-1111	aaaa	80	1
2222-2222-2222	aaaa	100	
2222-2222-2222	bbbb	50	
3333-3333-3333	aaaa	70	
3333-3333-3333	bbbb	90	1
3333-3333-3333	cccc	100	1
4444-4444-4444	bbbb	30	
4444-4444-4444	cccc	90	
5555-5555-5555	cccc	70	

→

event_id	champ	user_id (PK)	event_id (SK)
aaaa	1	1111-1111-1111	aaaa
bbbb	1	3333-3333-3333	aaaa
cccc	1	3333-3333-3333	aaaa

/Sortも要らない

GSIには暗黙でPrimary Keyが含まれる

Hierarchical data

- よくある階層構造を持つデータ
  - 組織構造
  - ネストしたカテゴライズ
  - などなど
- 検索を考えなければ階層構造をまるっと1アイテムに押し込む
- **Sort Keyで階層を表現**する

genre (PK)	category (SK)	name
food	japanese#seafood#crub	かに飯
food	japanese#seafood#fish	焼き魚
food	japanese#seafood#fish	刺し身
food	western#seafood#shrimp	エビフライ
electronics	computer#parts#disk	HDD
electronics	computer#parts#disk	SSD
electronics	computer#parts#memory	DDR-3
electronics	computer#parts#memory	DDR-4

/Sort Keyで前方一致検索

あ、これSQLアンチパターンで（ｒｙ

商品設計が雑なのはご愛嬌

一応未完なのでまだ書くかも

“DynamoDB 虎の巻” [検索]

http://marcy.hatenablog.com/entry/2018/07/31/213705

“LET’S BECOME THE MASTER OF DYNAMODB”

“TO BECOME THE MASTER OF SERVERLESS”

/ THANKS!!