# AWS Managed Services

# DynamoDB

Adriano Sastre Vieira
adrianosastre@inatel.br
adrianosastre

02

# AWS MANAGED SERVICES
# DYNAMODB

## AGENDA

AWS - The 5 Pillars
Serverless / Microservices
AWS Managed Services
Minimum Stack:
    API Gateway + Lambda + DynamoDB

DynamoDB - Theory
    What is it / What is not
    How to model (NoSql vs RDBMS)
    Single Table Design
    When/how to use
    When/how NOT to use

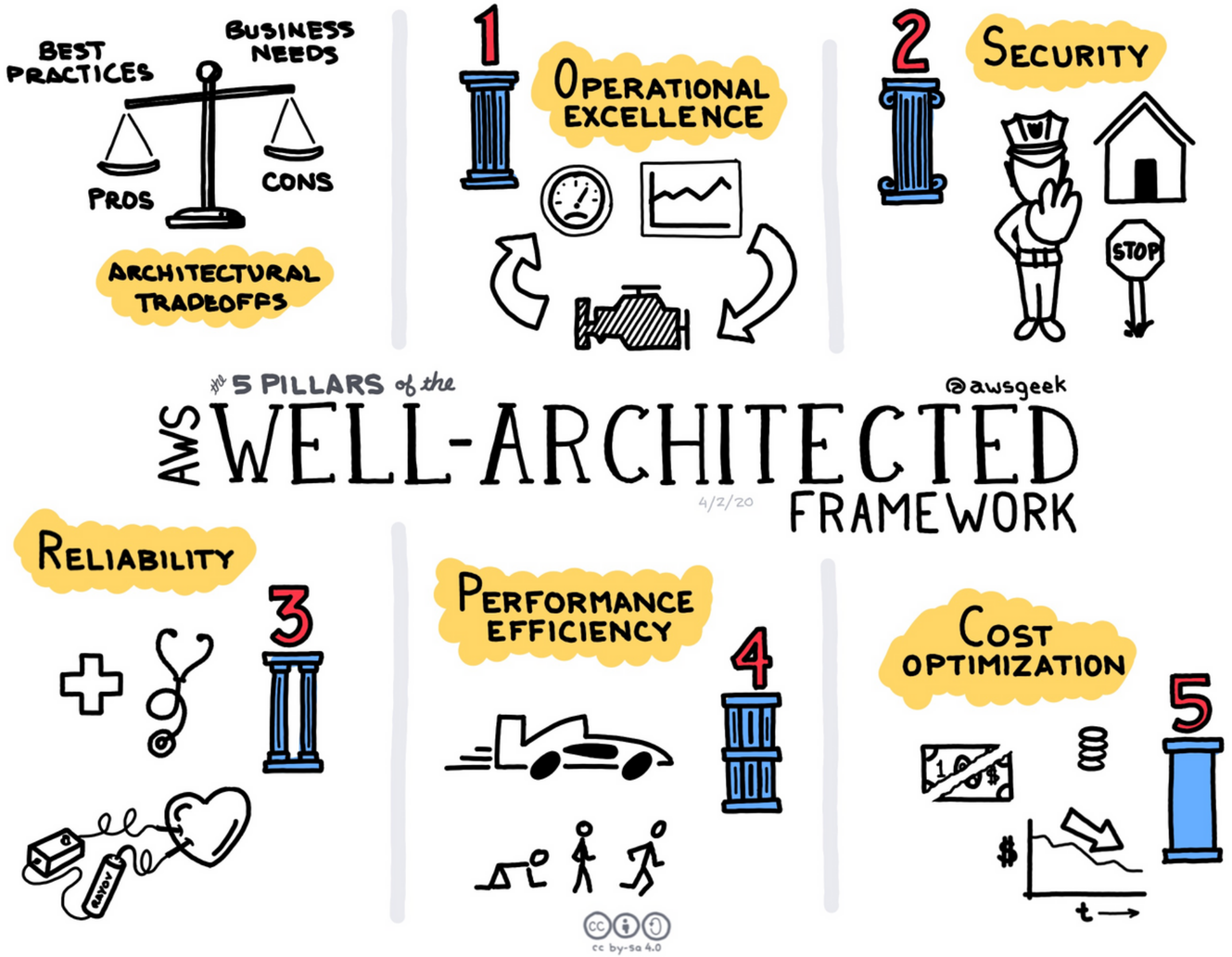
DynamoDB - Hands on
    Cloud Formation + CDK
    The project on Visual Code
    Deploying on AWS
    Testing with Postman

# AWS - THE 5 PILLARS

https://aws.amazon.com/getting-started/fundamentals-core-concepts/

New "mental models":

- Operational excelllence: thinking about operations as automation (CloudFormation, CDK ...)
- Security: zero trust, the principle of least privilege (IAM, data encription ...)
- Reliability: using fault isolation zones to limit blast radius (regions, quotas, throttling ...)
- Performance Efficiency: think of your services as cattle, not pets. (horizontal vs vertical scalling, auto scalling ...)
- Cost Optimization: OpEx (pay-as-you-go) instead of CapEx (one-time purchase)



04

# SOME IMPORTANT CONCEPTS

## Serverless:

- Serverless is the practice of using managed services with event driven compute functions to avoid or minimize infrastructure management, configuration, operations, and idle capacity.
- Wide range of things an application architecture may need. e.g. Compute, Storage, Data, Monitoring, Queue, Notification …
- It's a paradigm shift, potentially even more impactful than the move to t*he "Cloud"* before it

FaaS: Function as a Service

- Serverless approach to compute (i.e.: AWS Lambda)
- But compute (i.e. running code) is not the sole aspect of the application architecture

https://medium.com/serverless-transformation/in-defence-of-serverless-the-term-764514653ea7

https://pauldjohnston.medium.com/serverless-best-practices-b3c97d551535

05

# SOME IMPORTANT CONCEPTS

## Microservices:

- The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.
- These services are built around business capabilities and independently deployable by fully automated deployment machinery.
- Many people are using serverless applications to build a microservice architecture.

https://martinfowler.com/articles/microservices.html

Advantages:

- Independence and decoupling of parts of the system
- You can choose the most appropriate technologies / database per service
- Small teams with the whole knowledge in each service is better for development and maintenance

Main characteristic:

- One database per microservice

Challenges:

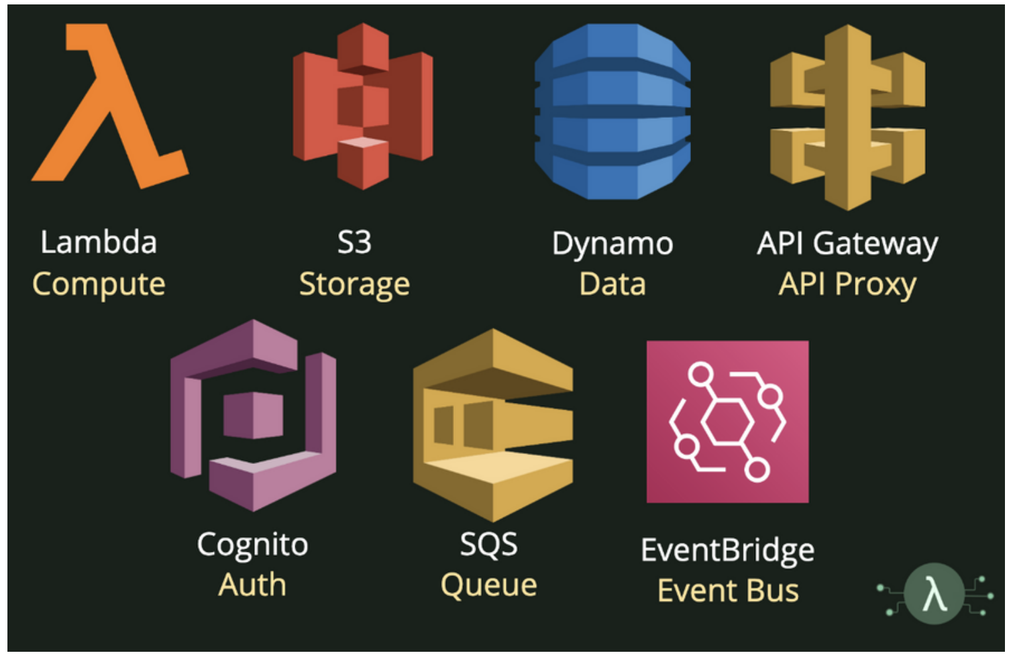
- Distributed transactions between services
- Design for failure
- Interface changes needs to be well coordenated between services

# SOME AWS MANAGED SERVICES

https://aws.amazon.com/serverless/

https://serverlessland.com/

https://aws.amazon.com/free

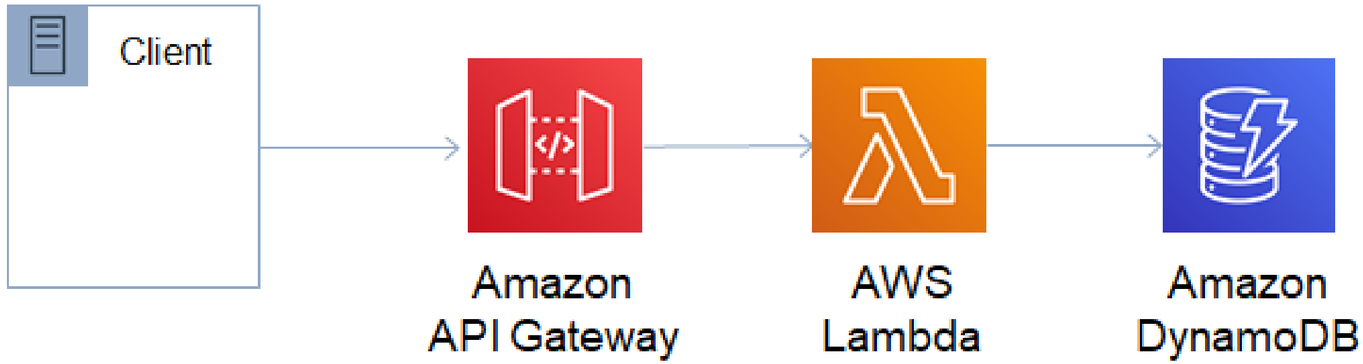



PROS:

- Cost reduction
  - initial cost tend to zero (free tier, pay as you use)
- Security
  - e.g. no need to install security patches on servers
- High performance
- NoOps
  - e.g. no servers / storage to manage or to scale, no need for a infrastructure team
- More scalable
- Greener
- Productivity: developers focus on delivering business value

# SERVERLESS EVENT-DRIVEN DESIGN SAMPLE ARCHITECTURE

## (STARTING WITH THE BASIC)




https://aws.amazon.com/getting-started/deep-dive-serverless/

# API GATEWAY

Fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale.

Handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, CORS support, authorization and access control, throttling, monitoring, and API version management.

REST APIs
WebSocket APIs

Free-tier 12-months applied, up to 1 million requests per month, after that: cost by request and volume of data transfered.

https://aws.amazon.com/api-gateway

https://aws.amazon.com/free/?all-free-tier.q=api-gateway

# LAMBDA

AWS Lambda is an event-driven, serverless computing platform provided by Amazon as a part of Amazon Web Services.

Runs code in response to events and automatically manages the computing resources required by that code.

- Triggered by events (e.g. HTTP Calls via API Gateway, S3 new objects on a bucket, new SQS in a queue, new item in a DynamoDB table …)

Free-tier forever applied, up to 1 million requests / month; after that: cost by allocaded memory and execution time comsumed.

https://aws.amazon.com/lambda

https://aws.amazon.com/free/?all-free-tier.q=lambda

# DYNAMODB

Fast and flexible NoSQL database service for any scale. Key-value and document database that delivers single-digit millisecond performance at any scale.

Fully managed, multi-region, multi-active, durable database with built-in security, backup and restore, and in-memory caching for internet-scale applications.

Free-tier forever applied, up to 25GB storage, 25 read/write provisioned units (about 200M requests/month). After that: cost by storage and requests.

https://www.dynamodbguide.com/the-dynamo-paper

- Created by Amazon (2004-2007), public released by AWS in 2012

https://aws.amazon.com/dynamodb/

https://aws.amazon.com/free/?all-free-tier.q=dynamoDB

# DYNAMODB AS PART OF A SERVERLESS ARCHITECTURE



Just one part of moving to a more serverless architecture often involves adopting DynamoDB in place of the relational databases of the past.

NoSQL isn't new, but the idea of using it for any and all core business requirements is — this is because the whole system now works at a scale and event-driven nature before unknown.

DynamoDB is complex, doing it well even more so.

https://medium.com/serverless-transformation/in-defence-of-serverless-the-term-764514653ea7

# DYNAMODB
# NOSQL DESIGN VS RDBMS



NoSQL design requires a different mindset than RDBMS design.

RDBMS = you can go ahead and create a normalized data model without thinking about access patterns.

By contrast, you shouldn't start designing your schema for DynamoDB until you know the questions it will need to answer. Understanding the business problems and the application use cases up front is essential.

Access Patterns, e.g. :

- Get a user's profile data
- List the user's orders
- Get an order and its items
- List the user's orders by status

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-general-nosql-design.html#bp-general-nosql-design-concepts

# DYNAMODB DESIGN CONSIDERATIONS



https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-general-nosql-design.html#bp-general-nosql-design-concepts

It is important to understand three fundamental properties of your application's access patterns before you begin:

- Data size: Knowing how much data will be stored and requested at one time will help determine the most effective way to partition the data.
- Data velocity: DynamoDB scales by increasing the number of physical partitions that are available to process queries, and by efficiently distributing data across those partitions. Knowing in advance what the peak query loads will be might help determine how to partition data to best use I/O capacity.
- Data shape: Instead of reshaping data when a query is processed (as an RDBMS system does), a NoSQL database organizes data so that its shape in the database corresponds with what will be queried. This is a key factor in increasing speed and scalability.

*"DynamoDB is a Key-value and document database that delivers single-digit millisecond performance at any scale."*

- size / velocity:
  - Is it really needed? (such a scale and performance)
  - Does it make sense to use a NoSQL database? Is it the best option?
  - Is the technical team familiar with it?
- shape
  - Do you know all the questions that needs to be answered in the system?
- NO (for any of the above)
  - Have you considered using a RDS datase, e.g. Amazon Aurora Serverless?

YES we are going to use DynamoDB!

- Then data shape is the most important aspect!

# DYNAMODB DESIGN

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-general-nosql-design.html#bp-general-nosql-design-concepts



After you identify specific query requirements, you can organize data according to general principles that govern performance:

- Keep related data together. Keeping related data in close proximity has a major impact on cost and performance. Instead of distributing related data items across multiple tables, you should keep related items in your NoSQL system as close together as possible. As a general rule, you should maintain as few tables as possible in a DynamoDB application.
- Use sort order. Related items can be grouped together and queried efficiently if their key design causes them to sort together. This is an important NoSQL design strategy.
- Distribute queries. It is also important that a high volume of queries not be focused on one part of the database, where they can exceed I/O capacity. Instead, you should design data keys to distribute traffic evenly across partitions as much as possible, avoiding "hot spots."
- Use global secondary indexes. By creating specific global secondary indexes, you can enable different queries than your main table can support, and that are still fast and relatively inexpensive.

# DYNAMODB DESIGN

https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-general-nosql-design.html#bp-general-nosql-design-concepts



After you identify specific query requirements, you can organize data according to general principles that govern performance:

- Keep related data together. Keeping related data in close proximity has a major impact on cost and performance. Instead of distributing related data items across multiple tables, you should keep related items in your NoSQL system as close together as possible. As a general rule, maintain as few tables as possible in a DynamoDB application.
- Use sort order. Related items can be grouped together and queried efficiently if their key design causes them to sort together. This is an important NoSQL design strategy.
- Distribute queries. It is also important that a high volume of queries not be focused on one part of the database, where they can exceed I/O capacity. Instead, you should design data keys to distribute traffic evenly across partitions as much as possible, avoiding "hot spots."
- Use global secondary indexes. By creating specific global secondary indexes, you can enable different queries than your main table can support, and that are still fast and relatively inexpensive.

**These general principles translate into some common design patterns that you can use to model data efficiently in DynamoDB.**

# DYNAMODB
# PK, SK, ATTRIBUTES

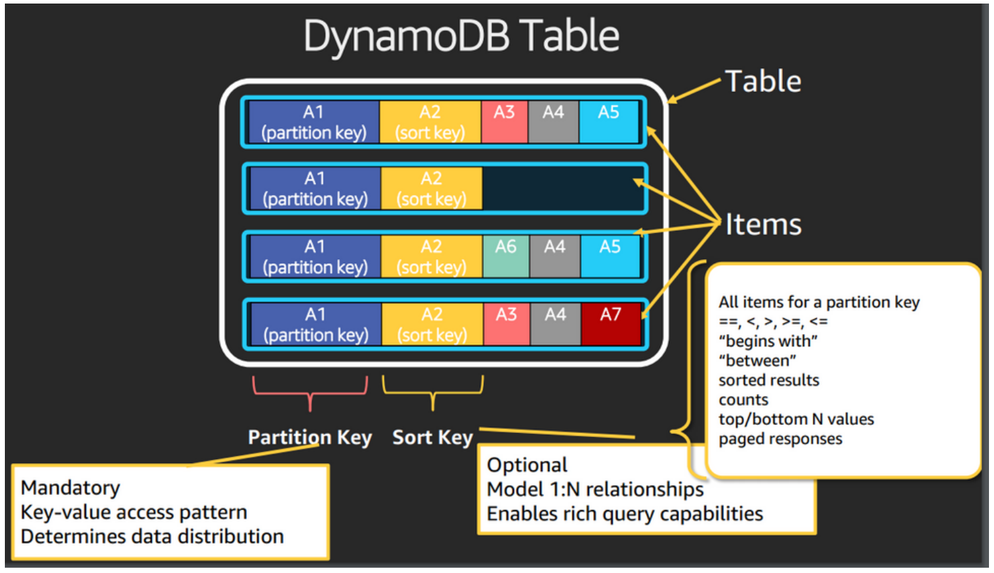
- Primary Key:
  - PK = Partition Key (hash key) (mandarory)
  - SK = Sort Key (range key) (optional)

- Attributes: Binary, Number or String
  - Can be grouped in a JSON-like structure

Each table item may have different attributes!

https://www.youtube.com/watch?v=kSnpuKr3Ajw

| Primary Key | | Attributes | | |
|---|---|---|---|---|
| **Actor (PARTITION)** | **Movie (SORT)** | | | |
| Tom Hanks | Cast Away | **Role** | **Year** | **Genre** |
| | | Chuck Noland | 2000 | Drama |
| | Toy Story | **Role** | **Year** | **Genre** |
| | | Woody | 1995 | Children's |
| Tim Allen | Toy Story | **Role** | **Year** | **Genre** |
| | | Buzz Lightyear | 1995 | Children's |
| Natalie Portman | Black Swan | **Role** | **Year** | **Genre** |
| | | Nina Sayers | 2010 | Drama |



# DYNAMODB
# CAPACITY



Per table configuration.

OnDemand:

- automatic and "infinite" scaling

Provisioned:

- possible to define independent read and write unit capacities
- possible to configure auto-scaling rules, e.g. min/max scaling
- when correctly configured, less cost than onDemand

Possible to change between OnDemand and Provisioned 1x per day!

https://aws.amazon.com/dynamodb/pricing/

# DYNAMODB
# HOW TO QUERY DATA



Get = one specific item, by PK or PK/SK
Query = several items, by PK/SK or indexed attributes

- High performance

Scan = several items, by any table attribute

- Poor performance
- High cost

# DYNAMODB
# INDEXES



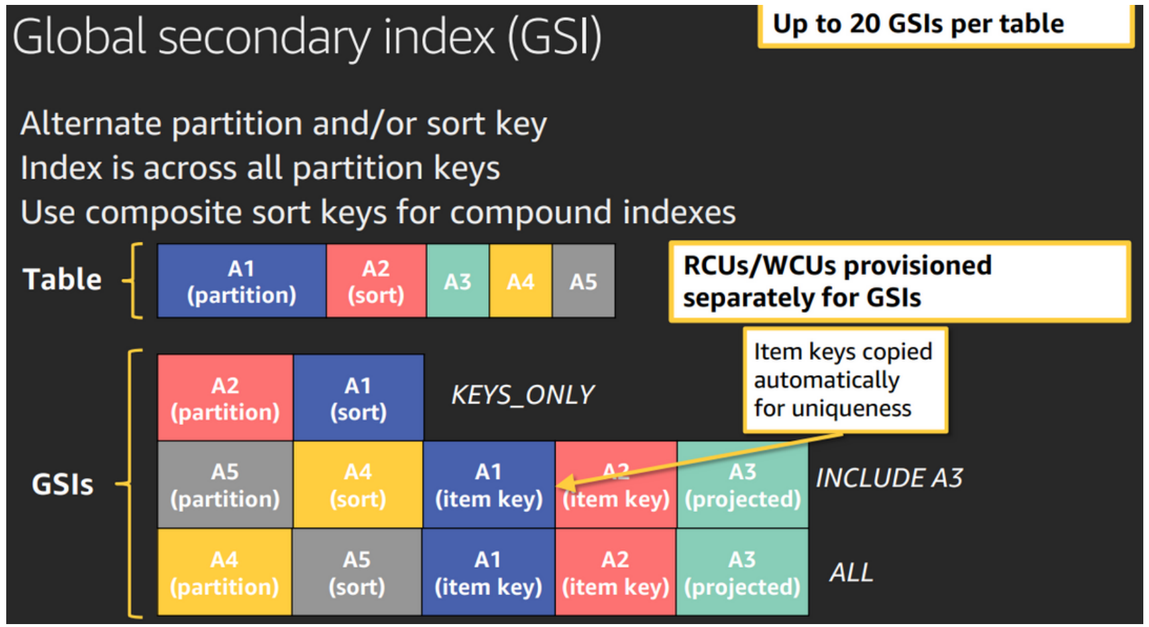
GSI = Global Secondary Index = applied to all table items
LSI = Local Secondary Index = Index inside the table partition (PK)

https://d1.awsstatic.com/events/reinvent/2019/REPEAT1_Advanced_design_patterns_for_Amazon_DynamoDB_DAT334-R1.pdf

Pros: performance: with an index, it is possbile to query (instead of scan) on attributes other than the PK/SK

Cons: behind the scenes, each GSI duplicates the table storage (and costs)

Limit = 20 GSI per table!



# DYNAMODB
## TTL

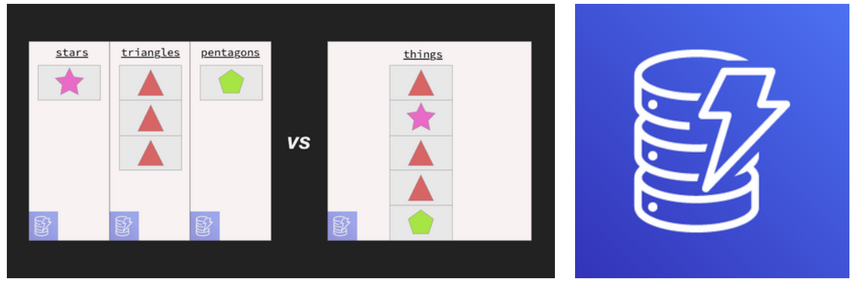


TTL (time to live attribute)

Optional attribute.
Automatically deletes the table item when the TTL is reached.

Each item may have a different time to live!

# DYNAMODB
# SINGLE TABLE DESIGN

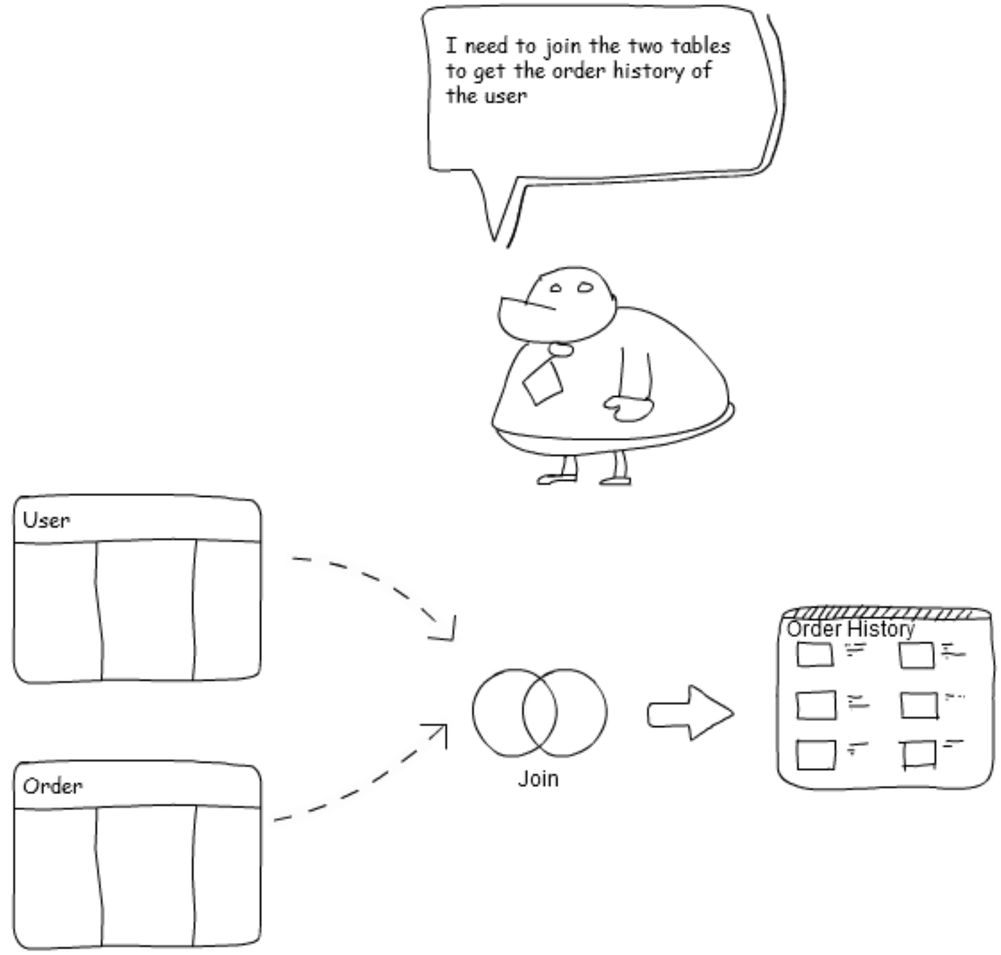



AWS Recommendation: maintain as few tables as possible!

Relational databases = recommend to normalize the data
Pro: data access very flexible
Con: it reduces the scalability (joins)

DynamoDB was built for enormous, high-velocity use cases, such as the Amazon.com shopping cart.

Rather than working to make joins scale better, DynamoDB sidesteps the problem by removing the ability to use joins at all.

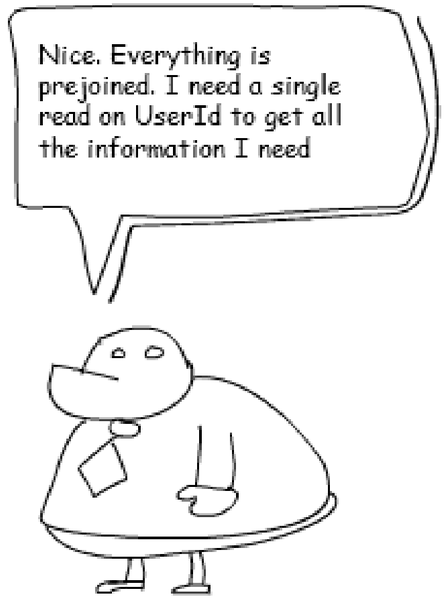


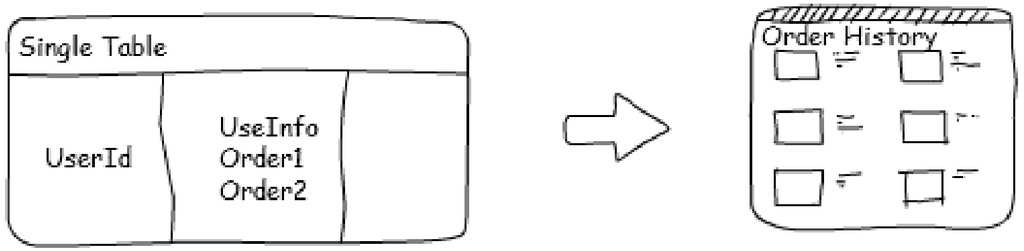
# DYNAMODB
# SINGLE TABLE DESIGN

DynamoDB was build with web scale in mind.
It can grow almost infinitely without degrading performance.
To achieve this DynamoDB removed joins completely.

You have to model the data in such a way that you can read the data in a single request by denormalizing the data.

*The main reason for using a single table in DynamoDB is to retrieve multiple, heterogenous item types using a single request.*



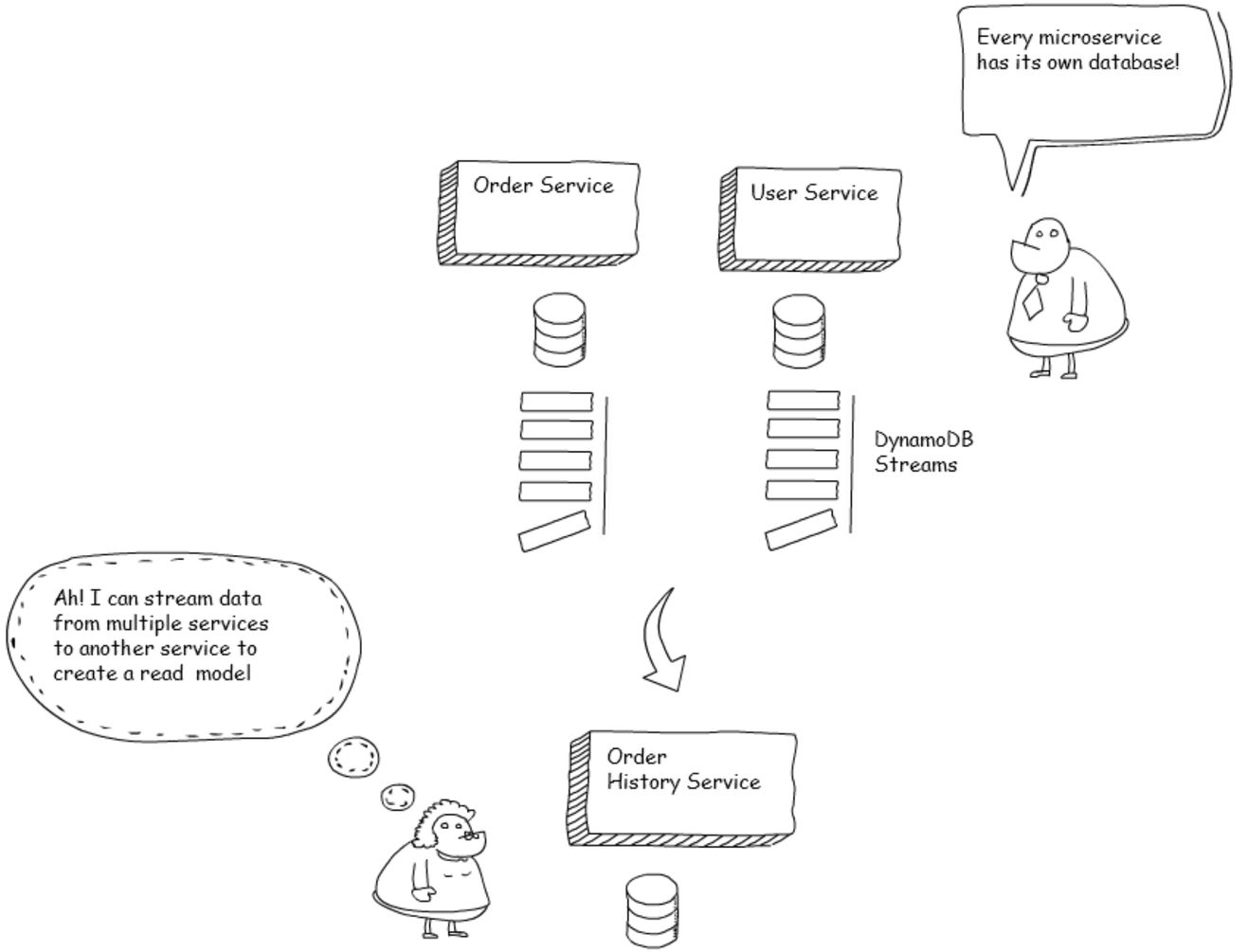



# DYNAMODB
# SINGLE TABLE DESIGN VS MICROSERVICES



One table per service vs One single table for all microservices?

The best practice for microservices is that every microservice owns its own data.



# DYNAMODB
# SINGLE TABLE DESIGN VS MICROSERVICES

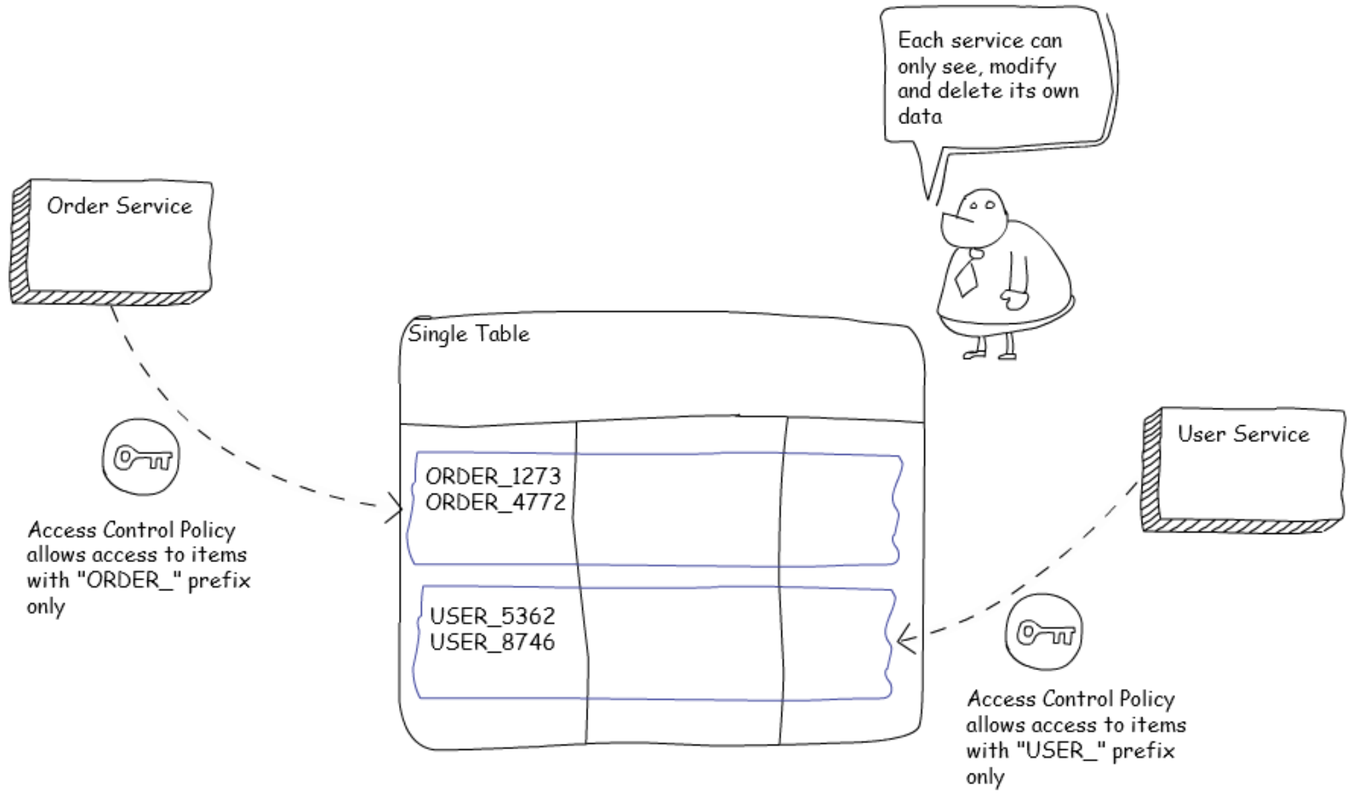


Alternative: use a single table with namespaces

HOW =

1. prefix every primary key (partition or sort key) with the name of the service
2. add a fined grained access control policy to your service that allows access to items with the service's prefix only.

https://cbannes.medium.com/microservices-with-dynamodb-should-you-use-a-single-table-or-use-one-table-per-microservice-25f54cf610d9



# DYNAMODB
# SINGLE TABLE DESIGN
# PROS X CONS



- Reducing the number of requests (Performance!)
- Less operational overhead (alarms, metrics…)
- Save money (capacity, provisioning for read /write …)
- When well designed, less indexes in total (e.g. a "username" GSI can be reused

- Steep learning curve
- Inflexibility of adding new access patterns
- Difficulty of exporting your tables for analytics

- *"NoSQL is not flexible. It's efficient but not flexible." — Rick Houlihan*

https://www.alexdebrie.com/posts/dynamodb-single-table/
https://serverlessfirst.com/dynamodb-modelling-single-vs-multi-table/
https://www.jeremydaly.com/how-to-switch-from-rdbms-to-dynamodb-in-20-easy-steps/

*"A well-optimized single-table DynamoDB layout looks more like machine code than a simple spreadsheet"*

# DYNAMODB
# WHEN AND HOW TO USE



When you have any of the following:

- Huge amount of data
- Performance required
- Well defined data access patterns

*When modeling with DynamoDB, you should be following best practices. This includes denormalization, single-table design, and other proper NoSQL modeling principles.*

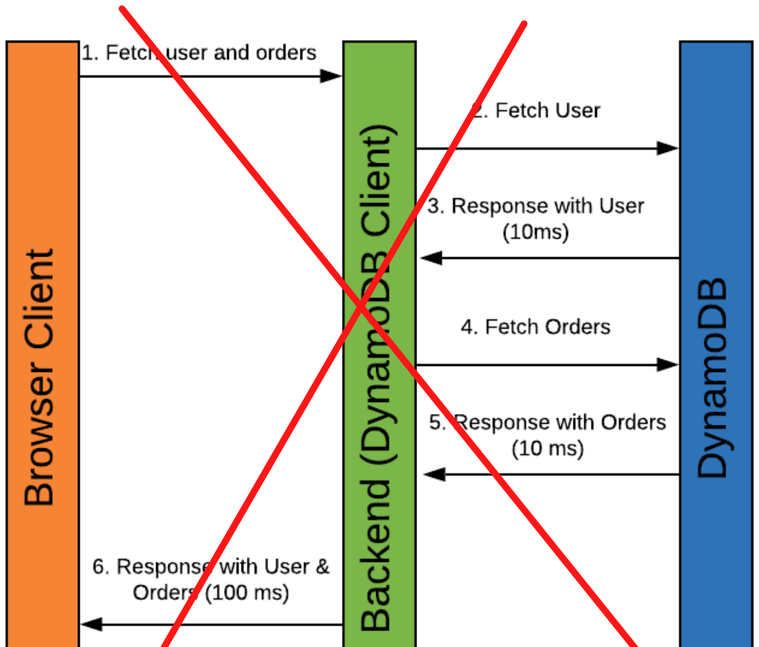
https://www.alexdebrie.com/posts/dynamodb-single-table/

https://www.alexdebrie.com/posts/dynamodb-patterns-serverless/#the-true-microservice

In this pattern, the True Microservice, we will see how to get all of the benefits of DynamoDB with none of the downsides.

# DYNAMODB
# WHEN AND HOW NOT TO USE !

- "Whenever the benefits don't outweigh the costs"
  - In new applications where developer agility is more important than application performance
  - In applications using GraphQL




Faux-SQL design: the practice of choosing parts of both relational and NoSQL concepts, e.g. use multiple DynamoDB tables and normalize their data.

- This pattern is not recommended by the DynamoDB team
- This pattern recommended a lot by the AWS AppSync team (!)

- You are making multiple queries for your calls. You should aim to satisfy your needs with a single query.
- Your application code just became a database management system. Rather than doing a JOIN in SQL, you are doing it in code. This means you are responsible for maintaining referential integrity and for optimizing your queries.

You know what's really good at maintaining referential integrity and optimizing JOINs? A relational database.

https://www.alexdebrie.com/posts/dynamodb-patterns-serverless/#faux-sql

# HANDS ON

# Thank you!

adrianosastre@inatel.br

# HANDS ON

This project on github:

https://github.com/adrianosastre/XPForward_DynamoDB_CDK