Amazon DynamoDB Data Modeling

Linux Academy

Course Introduction

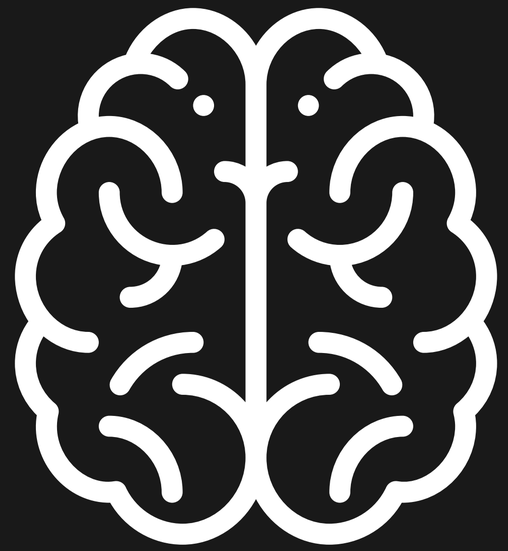
Linux Academy

COURSE INTRODUCTION

Course Introduction

What You'll Learn

- NoSQL concepts and DynamoDB's features
- How to optimize DynamoDB for:
  - Runtime performance
  - Cost
- DynamoDB data modeling patterns
  - 1:N, N:M, hierarchical, GSI overloading, and more
  - Write sharding, sparse indexes, and materialized aggregations
- Strategies to migrate data from RDBMS to DynamoDB
- Real-world examples and hands-on labs!



Prerequisites

- No AWS certification required
- AWS basics
- **Amazon DynamoDB Deep Dive**
- Tools:
  - AWS Management Console
  - AWS CLI
  - Linux/macOS
  - GitHub
  - Python
- Relational databases (SQL)
- **Hands-on** ... follow along!

Linux Academy

COURSE INTRODUCTION

NoSQL Concepts

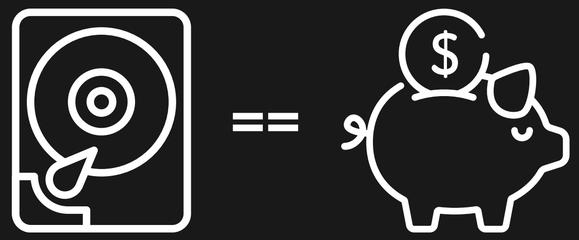
NoSQL Concepts

- Relational databases were created to **decrease** storage pressure.
- Data was **normalized** to save space and cost.

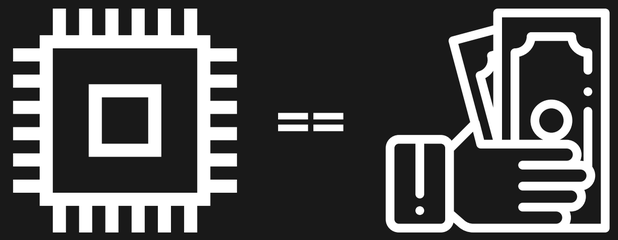
- Optimized for storage
- Normalized
- Relational
- Supports ad hoc queries
- Scales vertically

NoSQL Concepts

- OLAP: Online Analytical Processing
- OLTP: Online Transaction Processing
- Relational database cost grows **geometrically** with storage requirements
- Scalability issues in the **petabyte** range



Storage is cheap



CPU is expensive

Not everything is Excel

NoSQL Concepts

- Modern applications store relationships between data
- Metadata (e.g., social network attributes “known since” or “connected on” dates)

Sparse Data

Contact ID	Email1	Email2	CellPhone	HomePhone	Twitter
123	**mark@example.com**	NULL	NULL	**+1-202-555-0189**	NULL
234	NULL	NULL	NULL	NULL	**@mrichman**
567	**john@example.com**	**tia@example.com**	NULL	NULL	NULL
9876	NULL	NULL	NULL	NULL	NULL

NoSQL Concepts

- Modern applications store relationships between data
- Metadata (e.g., social network attributes “known since” or “connected on” dates)

NoSQL — Schema on Read

Contact ID	Email1	Email2	CellPhone	HomePhone	Twitter
123	mark@example.com			+1-202-555-0189	
234					@mrichman
567	john@example.com	tia@example.com			
9876					

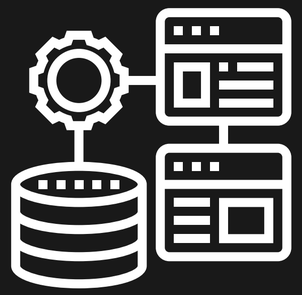
NoSQL Concepts

What exactly **is** data modeling?

- Decide how your data is organized in a DynamoDB table
- Impacts application **performance** and **cost**
- More requests == higher cost
- Properly designed data model is **necessary** for high performance in DynamoDB

NoSQL Concepts

What's the goal of data modeling?



Relational Model

- Allow any query
- Dynamically elicit relationships

DynamoDB Model

- Precompute answers to business questions
- Identify most important queries
- "Access pattern"

Linux Academy

Understanding DynamoDB Data Structures

Linux Academy

UNDERSTANDING DYNAMODB DATA STRUCTURES

DynamoDB and Tables

DynamoDB and Tables

What Is DynamoDB?

- Managed NoSQL database service
- Store and retrieve any amount of data
- Serve any level of request traffic
- Consistent, predictable performance
- Single-digit millisecond latency

DynamoDB and Tables

DynamoDB is both a **wide column** and **key-value** store.

	Email	Gender	Age
Mark	mark@example.com	Male	46
	1579098567	1579098567	1579098567

	Email	Gender
Tia	tia@example.com	Female
	1579098771	1579098771

	Email	Planet	EyeColor
Yoda	yoda@example.com	Dagobah	Green
	1579098812	1579098812	1579098812

Wide Column

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD,2,01/04/2020
K4	Rush,YYZ,2112

Key-Value

DynamoDB and Tables

Tables, Items, and Attributes

DynamoDB and Tables

Tables, Items, and Attributes

DynamoDB and Tables

Tables, Items, and Attributes

DynamoDB and Tables

DynamoDB Capacity Modes

On-Demand

- Pay per request
- Automatically scales your capacity up or down to match the workload
- Good for new tables

Provisioned

- Pay per capacity unit (cheaper than on-demand)
- Specify reads and writes per second
- Use auto scaling

DynamoDB and Tables

Read Capacity Unit (RCU)

- Each API call to read table data is a read request
- Requests can be **strongly** consistent, **eventually** consistent, or **transactional**
- 1 RCU = 1 strongly consistent read/sec up to 4 KB
- 1 RCU = 2 eventually consistent reads/sec up to 4 KB
- 2 RCU = 1 transactional read/sec up to 4 KB
- Example: An eventually consistent read of an 8 KB item would require 1 RCU

Write Capacity Unit (WCU)

- Each API call to write table data is a write request
- 1 WCU = 1 standard write/sec up to 1 KB
- 2 WCU = 1 transactional write/sec up to 1 KB
- Example: A standard write request of a 3 KB item would require 3 WCUs

DynamoDB and Tables

On-Demand Capacity

- No minimum capacity; pay more per request than provisioned capacity
- Idle tables not charged for read/write, but only for storage and backups
- No capacity planning required — just make API calls
- Eliminates the tradeoffs of over- or under-provisioning
- Use on-demand for new product launches
- Switch to provisioned once a steady state is reached
- $1.25 per million WCU (us-east-1)
- $0.25 per million RCU (us-east-1)
- Example:
  - You host a web service where you charge per API call
  - On-demand prevents being throttled because you never want to be down
  - 4,000,000 WCU per month
  - 8,000,000 RCU per month
  - 30 GB storage
  - $1.25 per million WCU x 4 million WCU = $5.00
  - $0.25 per million RCU x 8 million RCU = $2.00
  - 10 GB x $0.25 = $2.50
  - **Total Bill: $9.50**

DynamoDB and Tables

Provisioned Capacity

Provisioned

- Minimum capacity required
- Able to set a budget (maximum capacity)
- Subject to throttling
- Auto scaling available
- Risk of under-provisioning — monitor your metrics
- Lower price per API call
- $0.00065 per WCU-hour (us-east-1)
- $0.00013 per RCU-hour (us-east-1)
- $0.25 per GB-month
- DynamoDB Free Tier:
  - First 25 GB are free
  - First 25 RCUs and WCUs are free
- Example:
  - 50 WCU, 50 RCU, 25 GB
  - 25 WCU x $0.00065 per hour x 24 hours x 30 days = $11.70
  - 25 RCU x $0.00013 per hour x 24 hours x 30 days = $2.34
  - 25 GB = $0.00
  - **Total Bill: $14.04**

Linux Academy

UNDERSTANDING DYNAMODB DATA STRUCTURES

Items and Attributes

Items and Attributes

Items are uniquely identifiable by a **primary key**, set at table creation time.

Table: Contacts

Primary Key	Attributes			
Partition Key: UserID				
johnh	FirstName	LastName	Email	
	John	Hanna	johnh@example.com	
markr	FirstName	LastName	Email	Age
	Mark	Richman	markr@example.com	46
terryc	FirstName	LastName	Email	
	Terry	Cox	terryc@example.com	

Items

Max Item Size: 400 KB

Items and Attributes

PutItem, UpdateItem, DeleteItem

- **PutItem**
  - **Writes** a single item to a table
  - Completely **replaces** existing items and all their attributes
  - If an item with the same primary key exists in the table, it **replaces** that item
- **UpdateItem**
  - **Modifies** a single item in a table
  - Changes or adds attributes without modifying existing ones
- WCU consumed by `PutItem`/`UpdateItem` is calculated using the **larger** of the old and new items
- **DeleteItem**
  - **Removes** a single item from a table
  - Consumes WCU based on the size of the deleted item (1 WCU if the item does not exist)

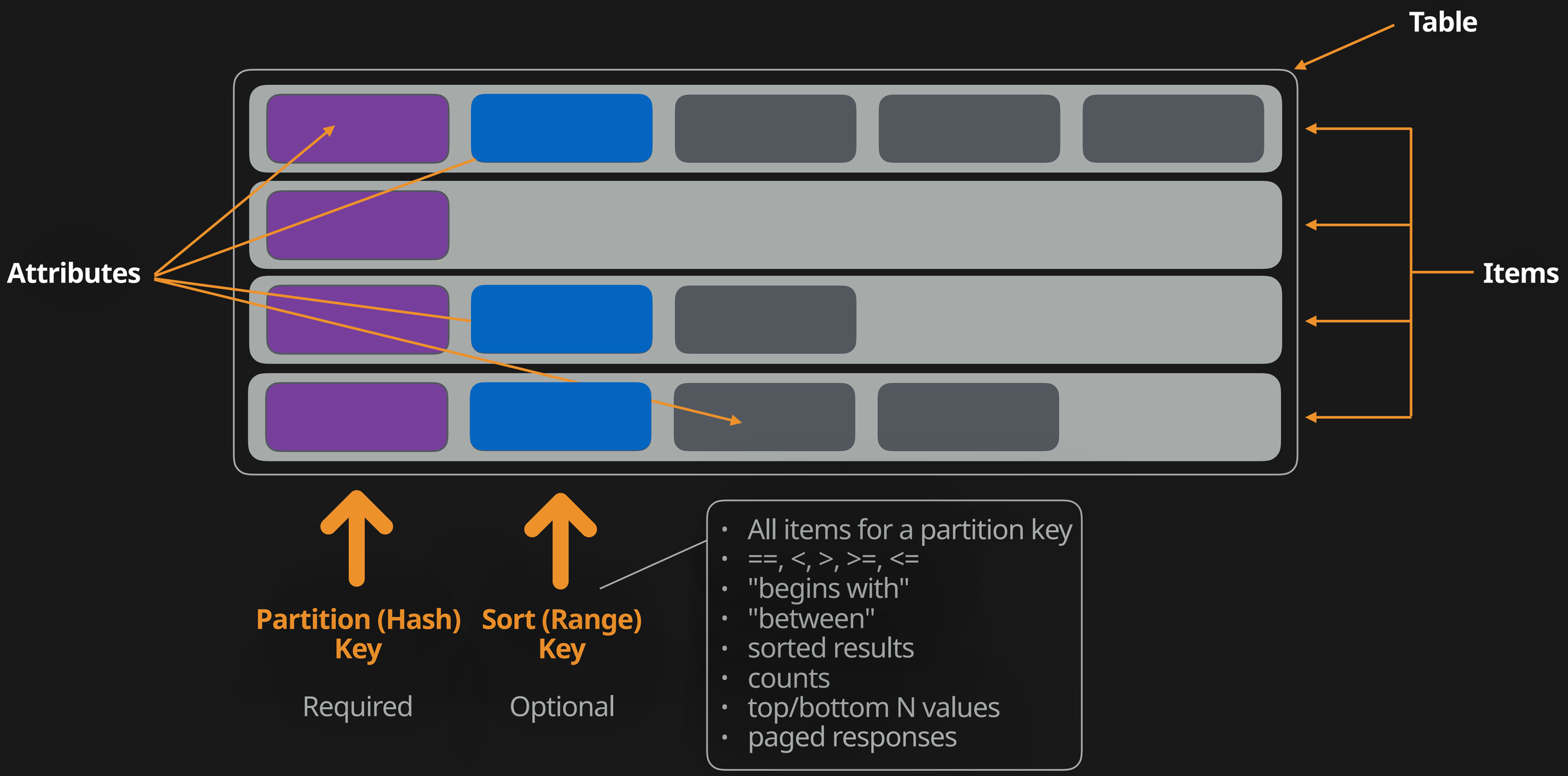
Linux Academy

UNDERSTANDING DYNAMODB DATA STRUCTURES

Partition and Sort Keys

Partition and Sort Keys

Linux Academy



Linux Academy

UNDERSTANDING DYNAMODB DATA STRUCTURES

Data Types

Data Types

Scalar

Exactly one value: **number**, **string**, **binary**, **boolean**, and **null**.

- Applications must encode binary values in base64 (e.g., `dGhpcyB0ZXh0IGlzIGJhc2U2NC1lbmNvZGVk`)
- Number type to represent a date or a timestamp in epoch time (e.g., 1579641111 for 21 Jan 2020 16:11:00)
- String type to represent a date or timestamp in ISO 8601 format (e.g., 2020-01-21T16:11:00Z)

Data Types

Document

Complex structure with nested attributes (e.g., JSON): **list** and **map**

List: Ordered collection of values

```
Favorites: ["Cookies", "Coffee", 3.14159]
```

Map: Unordered collection of name–value pairs (similar to JSON)

```
{
    Day: "Monday",
    UnreadEmails: 42,
    ItemsOnMyDesk: [
        "Coffee Cup",
        "Telephone",
        {
            Pens: { Quantity: 3 },
            Pencils: { Quantity: 2 },
            Erasers: { Quantity: 1 }
        }
    ]
}
```

Data Types

Set

Multiple scalar values of the same type: string set, number set, binary set

```
["Black", "Green", "Red"]
[42.2, -19, 7.5, 3.14]
["U3Vubnk=", "UmFpbnk=", "U25vd3k="]
```

Linux Academy

UNDERSTANDING DYNAMODB DATA STRUCTURES

Demo: Creating a Table

Linux Academy

UNDERSTANDING DYNAMODB DATA STRUCTURES

Indexes

Indexes

Secondary indexes are data structures that contain a **subset of attributes** from a table, along with an **alternate key** to support query operations. You can **retrieve data from the index using a query**, just like with a table. A table can have multiple secondary indexes.

Global Secondary Index (GSI): Index with a **different partition and sort key** from those on the base table. The primary key of a GSI can be either simple (partition key) or composite (partition and sort keys).

Local Secondary Index (LSI): Index that has the **same partition key** as the base table, but a **different sort key**. The primary key of an LSI must be composite (partition and sort key). It must be created at the time of table creation.

Partition Key		Sort Key	
username	**email**	**order_id**	**total**
mark	mark@example.com	223	4096.64
terry	terry@example.com	224	1024.16

Order Table

	Partition Key		Sort Key
username	**email**	**order_id**	**total**
mark	mark@example.com	223	4096.64
terry	terry@example.com	224	1024.16

GSI (PK=email, SK=total)

Partition Key			Sort Key
username	**email**	**order_id**	**total**
mark	mark@example.com	223	4096.64
terry	terry@example.com	224	1024.16

LSI (SK=total)

Linux Academy

UNDERSTANDING DYNAMODB DATA STRUCTURES

Demo: Working with Indexes

Indexes

Features and Limitations

GSI

- Considered **global** because queries on the index span all data in the base table, across all partitions
- No size limitations
- Has its own provisioned throughput settings (RCU/WCU) separate from its base table
- Eventually consistent queries only

LSI

- Considered **local** because each partition is scoped to a base table partition with the same partition key
- Total size per partition key can't exceed 10 GB
- Shares provisioned throughput settings with its base table
- Strongly or eventually consistent queries

In general, use GSIs rather than LSIs, unless you need strong consistency in query results.

Linux Academy

Basics of DynamoDB Data Modeling

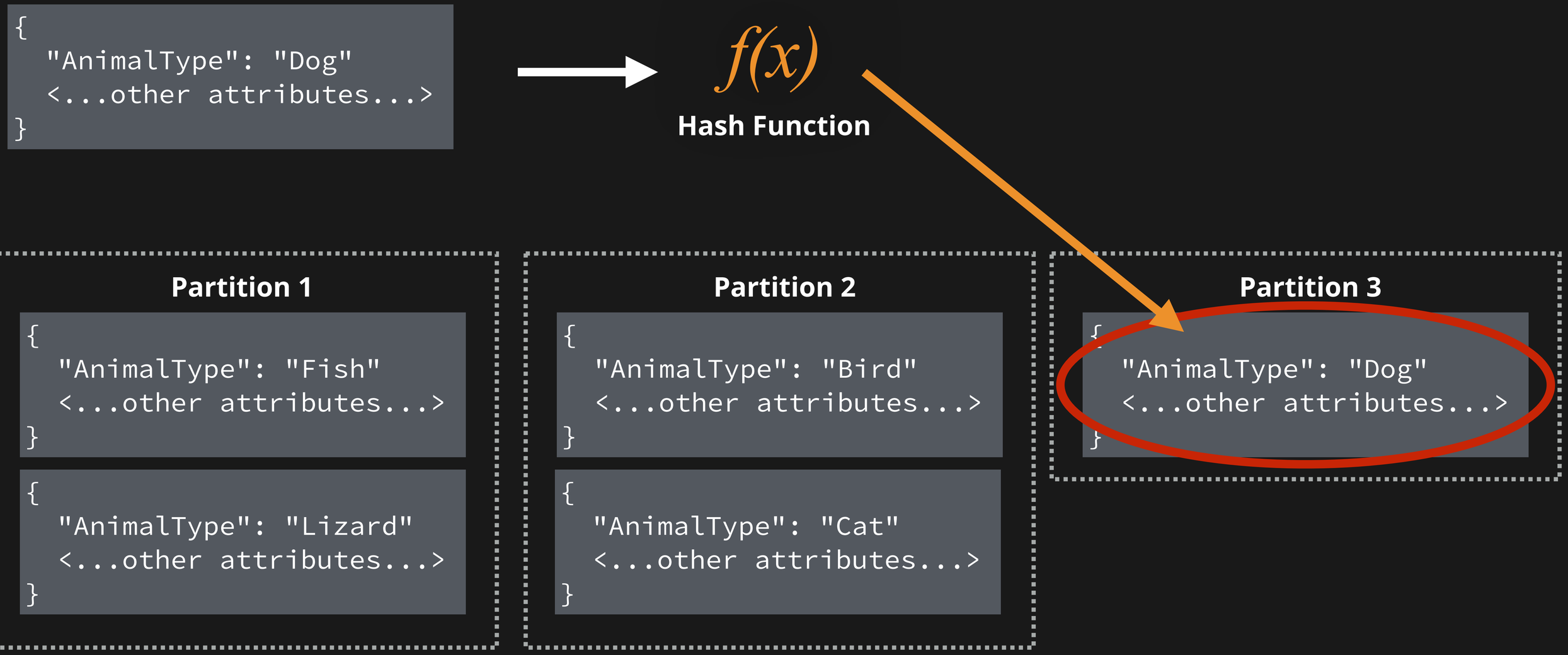
Linux Academy

BASICS OF DYNAMODB DATA MODELING

Partition Keys

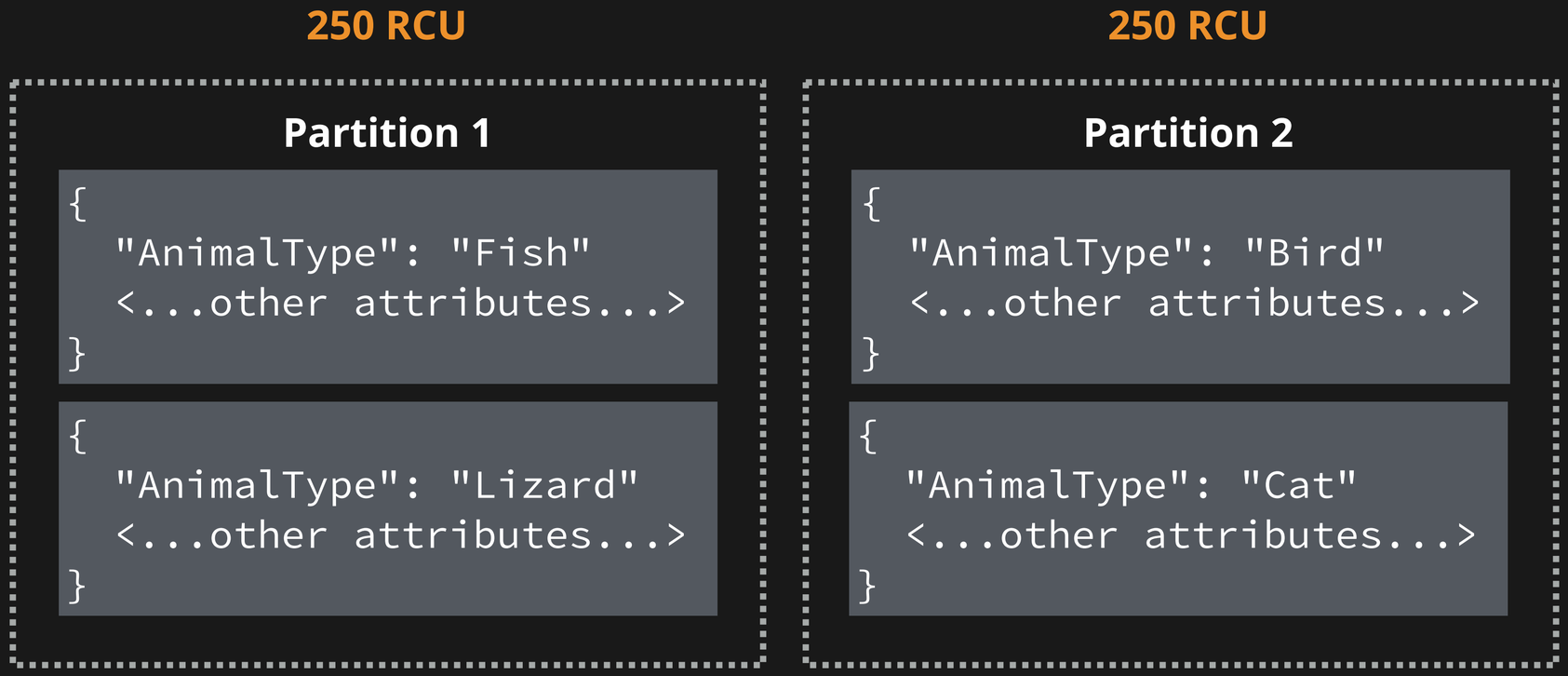
Partition Keys

A partition is an allocation of storage for a table.



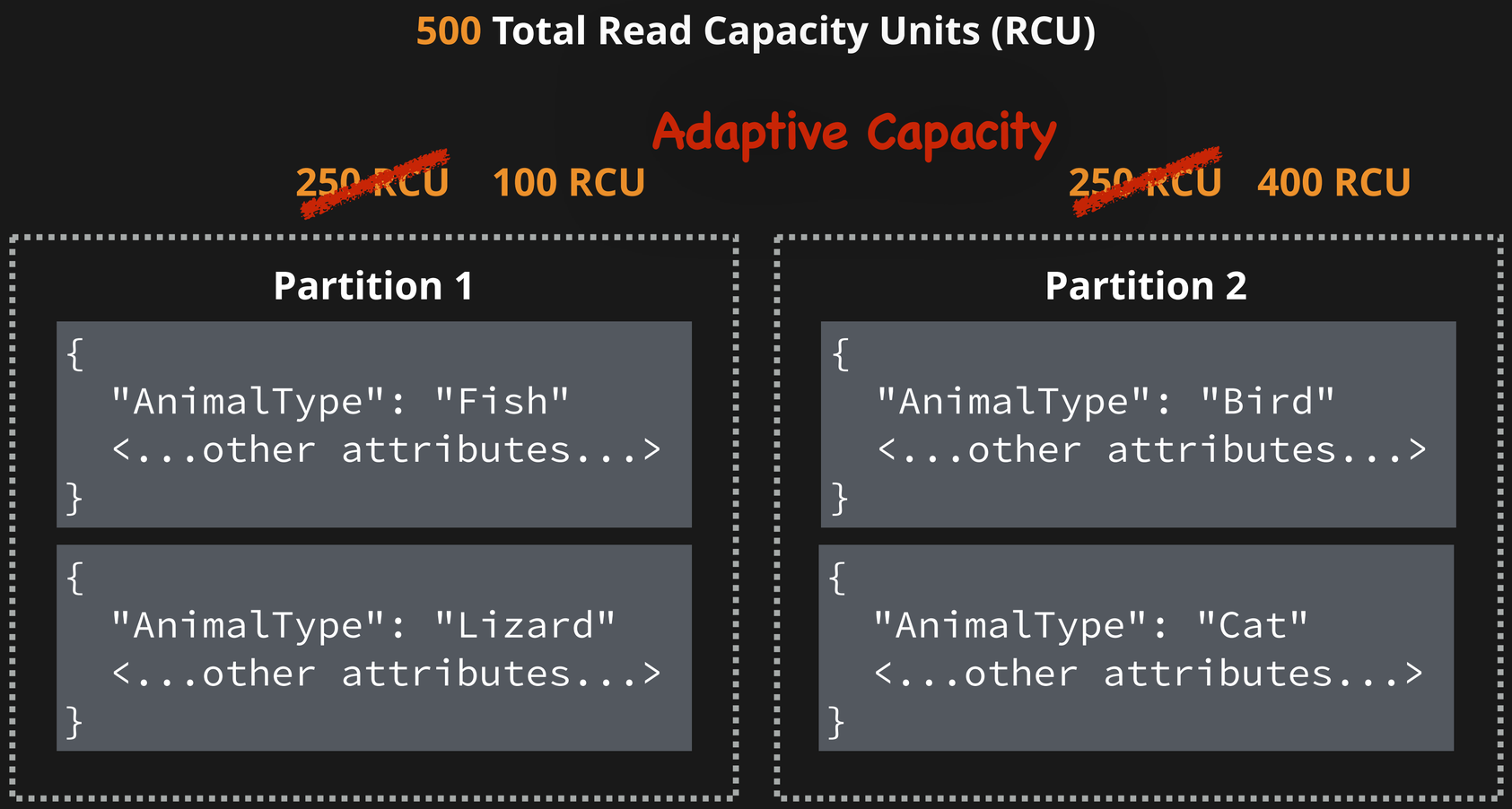
Partition Keys

500 Total Read Capacity Units (RCU)



Maximum **3000 RCU** and **1000 WCU** per partition

Partition Keys



Maximum **3000 RCU** and **1000 WCU** per partition

Partition Keys

Best Practices

- A good partitioning scheme affords **even distribution** of both data and workload, with growth in mind.
- Think of **partition key** as the dimension of scalability, and **distribute aggregates** across partitions (horizontal scaling).
- Ideal scaling conditions:
    1. Partition key is from a high cardinality set (that grows over time)
    2. Requests are evenly spread over the key space
    3. Requests are evenly spread over time

Linux Academy

BASICS OF DYNAMODB DATA MODELING

NoSQL Data Modeling Concepts

NoSQL Data Modeling Concepts

- Design tables to optimize for your application's **access patterns**
- Identify access patterns by:
  - New apps — review user stories and analyze the candidate access patterns
  - Existing apps — analyze logs
  - Example key access patterns:
    - Find employee by ID
    - Get all items for a given order ID
    - Get all employees with a given job title
    - Get total product inventory
    - Get salespeople ranked by total sales per period
    - ...and so on

NoSQL Data Modeling Concepts

- Optimize for the most common queries' speed and cost
  - Each access pattern should require a **single** DynamoDB request
- Use as few tables as possible — preferably a **single table**
- Single-table design helps avoid:
  - Using (or abusing) transactional APIs
  - Multiple DynamoDB API calls per application API call — the **N+1 problem**
  - Higher latency to the end user
  - More complexity per query — "fake joins" across DynamoDB tables
  - Much higher cost — overconsumption of RCU/WCU

Linux Academy

BASICS OF DYNAMODB DATA MODELING

One-To-Many Relationships

One-To-Many Relationships

1 — 1..*

ID	CustomerID	Timestamp	Total
165	234	1580242380	123.45
166	345	1580242381	22.65
167	654	1580242382	189.20
168	756	1580242399	14.85

Orders

ID	OrderID	SKU	Qty
2948	165	B0791TX5P5	1
2949	166	B07PGL2N7J	2
2950	166	1476773092	2
2951	167	B00AZCGF7K	1

OrderItems

One-To-Many Relationships

Primary Key		Attributes				
Partition Key (OrderID)	**Sort Key (SK)**					
1	order:item#1	price	qty	sku		
		19.95	1	B0791TX5P5		
	order:item#2	price	qty	sku		
		5.61	2	B07PGL2N7J		
	shipto	first_name	last_name	address	city	state
		Frank	Rizzo	30 E 60th Street	New York	NY
2	order:item#1	price	qty	sku		
		4.65	1	B00AZCGF7K		
	order:item#2	price	qty	sku		
		6.99	2	B000002JPA		
	shipto	first_name	last_name	address	city	state
		Jack	Tors	123 Main Street	Wilton Manors	FL

Orders

Linux Academy

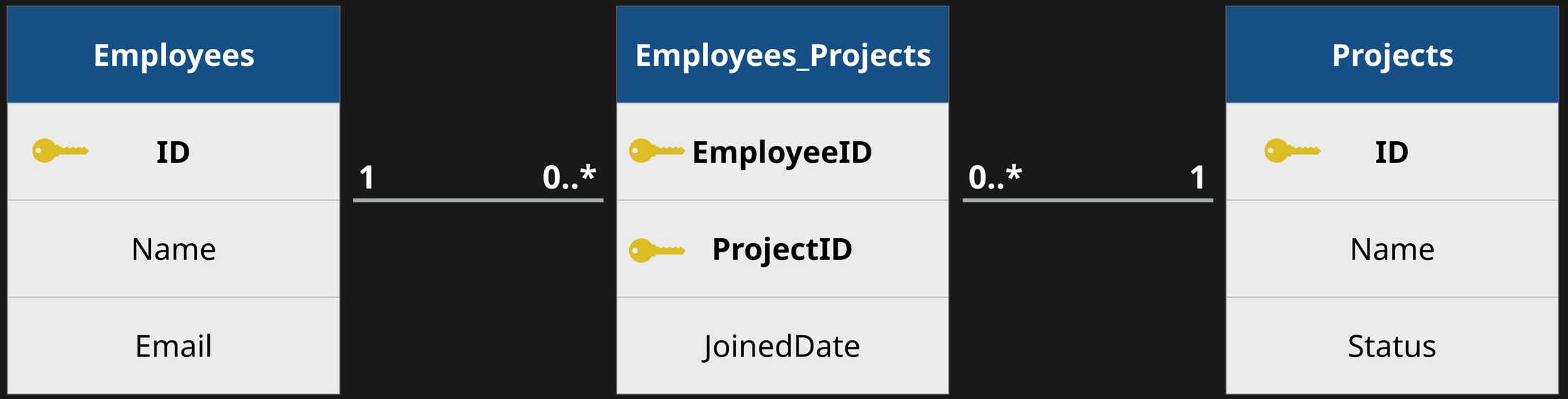
BASICS OF DYNAMODB DATA MODELING

Adjacency Lists and Many-To-Many Relationships

Adjacency Lists and Many-To-Many Relationships

Scenario

- Model a project management system where **employees** and **projects** are in a **many-to-many** relationship



Adjacency Lists and Many-To-Many Relationships

Business Questions

- These questions inform our application's **access pattern**:
  - To which projects does Employee X belong? (Get projects by employee ID)
  - Which employees are assigned to Project Y? (Get employees by project ID)
- Model our data using the **adjacency list** pattern
  - Minimize excessive data duplication
  - Use an **overloaded GSI partition key**
- **Single-table design**
  - Avoid unnecessary and costly fetches
  - Simplify access patterns

Adjacency Lists and Many-To-Many Relationships

Projects Table

Primary Key		Attributes		
Partition Key (ID)	**Sort Key (Item - GSI PK)**			
PROJECT1	PROJECT1	name	date_started	active
		Blue Book	2019-11-10	Y
	EMPLOYEE1	date_joined	active	
		2020-02-06	Y	
	EMPLOYEE2	date_joined	active	
		2019-11-11	N	
EMPLOYEE1	EMPLOYEE1	first_name	last_name	email
		Mike	Portnoy	mp@example.com
EMPLOYEE2	EMPLOYEE2	first_name	last_name	email
		Jim	Matheos	jm@example.com

Adjacency Lists and Many-To-Many Relationships

Global Secondary Index

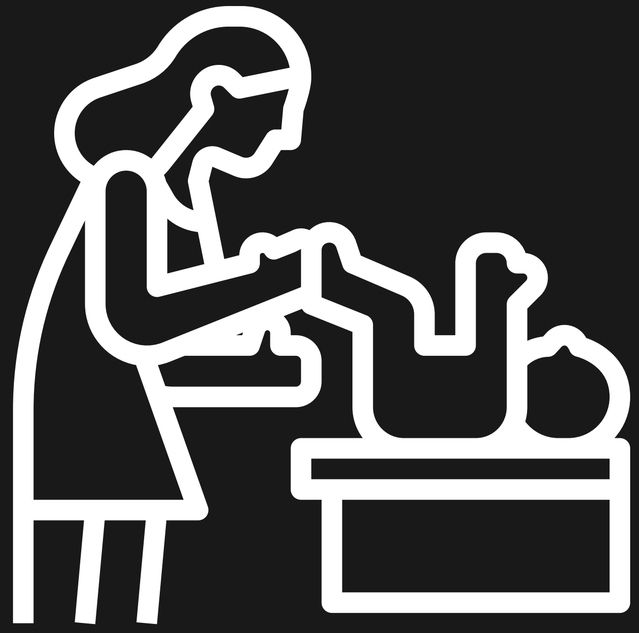
Primary Key	Projected Attributes		
Partition Key (Item)			
PROJECT1	ID	Item	Other Project Attributes
	PROJECT1	PROJECT1	...
EMPLOYEE1	ID	Item	Employee-Project Attributes
	PROJECT1	EMPLOYEE1	...
	ID	Item	Other Employee Attributes
	EMPLOYEE1	EMPLOYEE1	...
EMPLOYEE2	ID	Item	Employee-Project Attributes
	PROJECT1	EMPLOYEE2	...
	ID	Item	Other Employee Attributes
	EMPLOYEE2	EMPLOYEE2	...

Linux Academy

BASICS OF DYNAMODB DATA MODELING

Hierarchical Data

Hierarchical Data



Hierarchical Data

Primary Key	Attributes				
Partition Key (StoreNumber)	GSI PK			GSI SK	
1010	City	State	PostalCode	StateCityPostalCode	...
	Amherst	NY	14228-2704	NY#AMHERST#14228	...
1171	City	State	PostalCode	StateCityPostalCode	...
	Las Vegas	NV	89183-6841	NV#LAS VEGAS#89183	...
1808	City	State	PostalCode	StateCityPostalCode	...
	Spring Valley	NY	10977-5213	NY#SPRING VALLEY#10977	...
1760	City	State	PostalCode	StateCityPostalCode	...
	Orlando	FL	32828-7012	FL#ORLANDO#32828	...

Table: TargetStores

Hierarchical Data

Primary Key		Attributes		
Partition Key (State)	**Sort Key (StateCityPostalCode)**			
NY	NY#AMHERST#14228	City	PostalCode	...
		Amherst	14228-2704	...
	NY#SPRING VALLEY#10977	City	PostalCode	...
		Las Vegas	89183-6841	...
NV	NV#LAS VEGAS#89183	City	PostalCode	...
		Spring Valley	10977-5213	...
FL	FL#ORLANDO#32828	City	PostalCode	...
		Orlando	32828-7012	...

GSI: Location-index

Linux Academy

BASICS OF DYNAMODB DATA MODELING

GSI Overloading

GSI Overloading

- Soft limit of **20** GSIs per table
- Minimizing the number of GSIs **reduces cost**
- Overload an attribute based on the item's **context**

Scenario: Inventory Database

Access Patterns:

- Find all **orders** for a **user** within the last 30 days
- For a given **warehouse**, which **parts** are on backorder?

GSI Overloading

Inventory Table

Primary Key		Attributes (Data - GSI SK)	
Partition Key (ID)	**Sort Key (SK - GSI PK)**		
PART#8829	WH#63	Data	PartName
		Backordered	Tachyon Emitter
WH#24	TX#DALLAS		
PART#8823	WH#63	Data	PartName
		Backordered	Flux Capacitor
PART#8762	WH#24	Data	PartName
		In Stock	Warp Coil
USER#1234	Richman, Mark		
ORDER#9921	USER#1234	Data	OrderDate
		275.49	2020-01-04

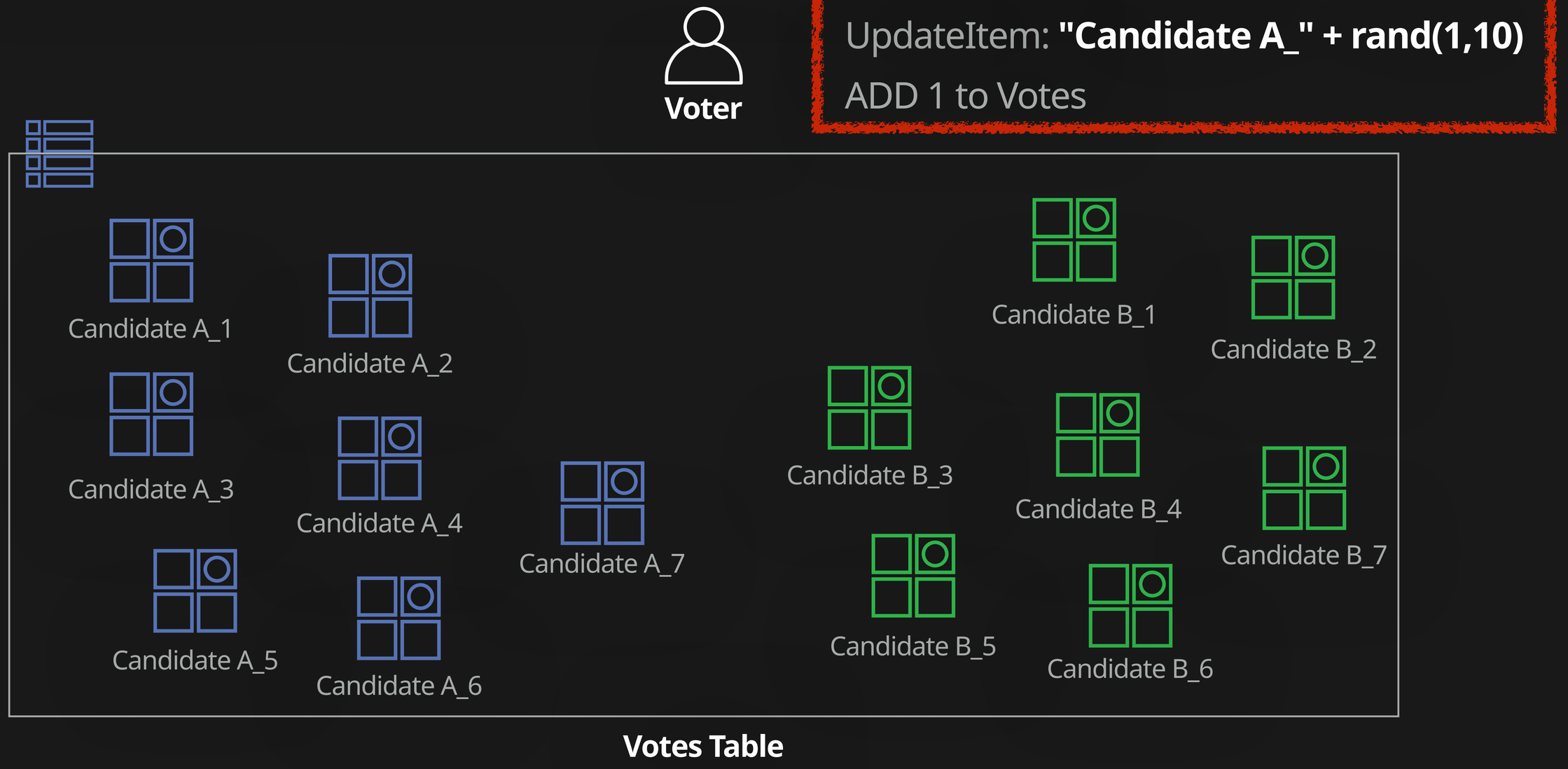
Linux Academy

BASICS OF DYNAMODB DATA MODELING

Write Sharding

Write Sharding

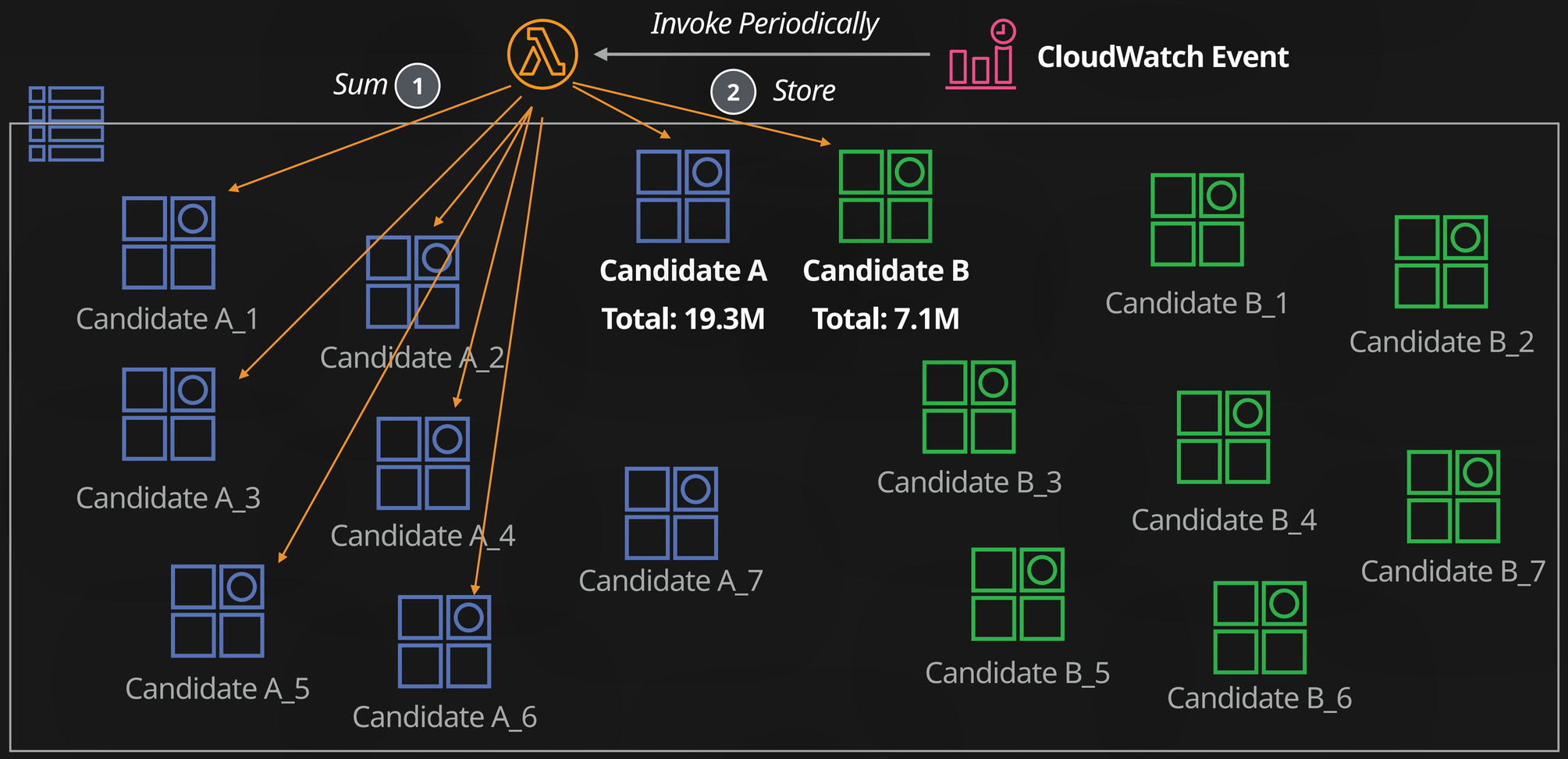
For write-heavy items, avoid hot keys by applying a random GSI suffix to partition keys.



Write Sharding

Linux Academy

Aggregation: Use a scheduled Lambda function to query the GSI for the sharded totals, sum them, and write an item back to the table with the aggregate total of votes for the given candidate.



Linux Academy

BASICS OF DYNAMODB DATA MODELING

Item Versioning

Item Versioning

Versioning Table

Primary Key		Attributes	
Partition Key (ID)	Sort Key (Version)		
1	0		CurrentVersion
			1
	1	CommitID	
		60e3582a139b823e1fe45d4a3ec68ec852e36b04	

Item Versioning

Versioning Table

Primary Key		Attributes	
Partition Key (ID)	Sort Key (Version)		CurrentVersion
1	0		3
	1	CommitID	
		60e3582a139b823e1fe45d4a3ec68ec852e36b04	
	2	CommitID	
		bb698e033d3cd00199df5063bc5cf5d0c22777ab	
	3	CommitID	
		efbeeac57a86de7f830dca9189331281efa3666e	

Linux Academy

BASICS OF DYNAMODB DATA MODELING

Sparse Indexes

Sparse Index

Sparse Index: If a sort key doesn't appear in every table item, the index is said to be **sparse**.

Sparse Index

Orders-SparseIndex Table

Primary Key		Attributes (IsOpen - LSI SK)
Partition Key (CustomerID)	Sort Key (OrderID)	
8675309	902101	IsOpen 1
8675309	906111	
8675309	994931	IsOpen 1
47988	802332	IsOpen 1
47988	901164	
47988	983644	

Sparse Index

Local Secondary Index: IsOpen

Primary Key		Attributes
Partition Key (CustomerID)	**Sort Key (IsOpen)**	
8675309	**1**	OrderID
		902101
8675309	**1**	OrderID
		994931
47988	**1**	OrderID
		802332

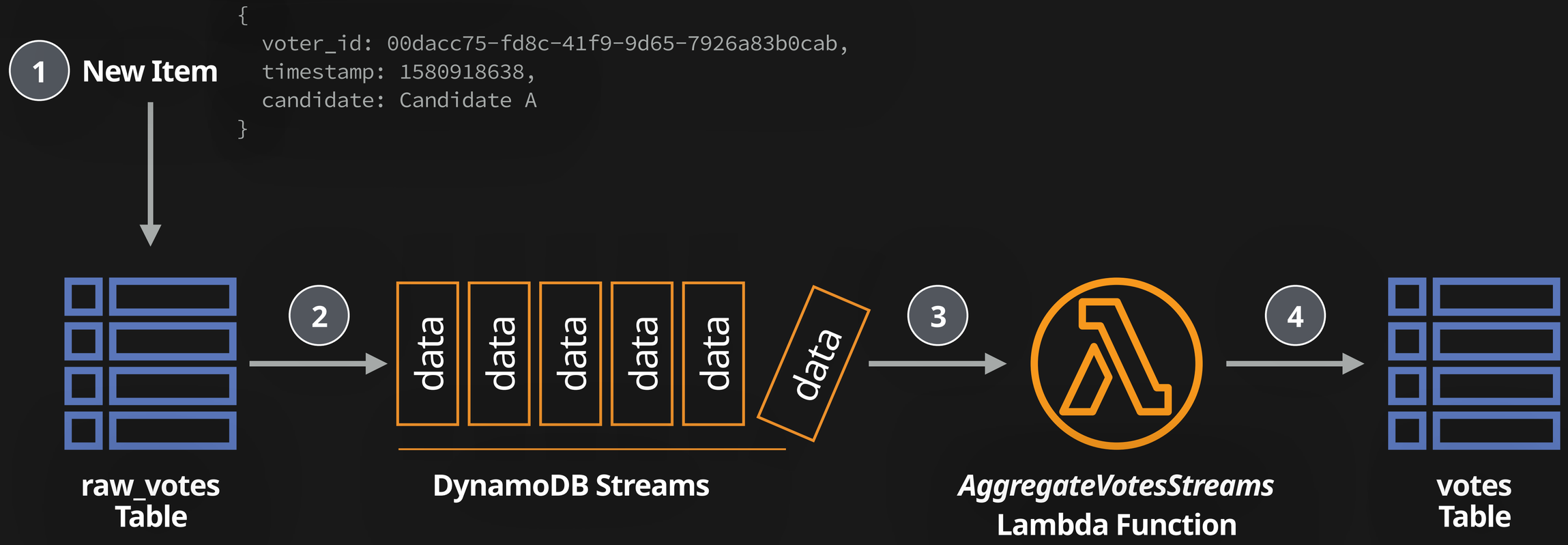
Linux Academy

BASICS OF DYNAMODB DATA MODELING

Materialized Aggregations

Materialized Aggregations

Linux Academy



The Lambda function will increment the vote count for each candidate's **raw_vote** record in the stream, incrementing the sum to the **votes** table.

Linux Academy

Analyzing Data Workloads

Linux Academy

ANALYZING DATA WORKLOADS

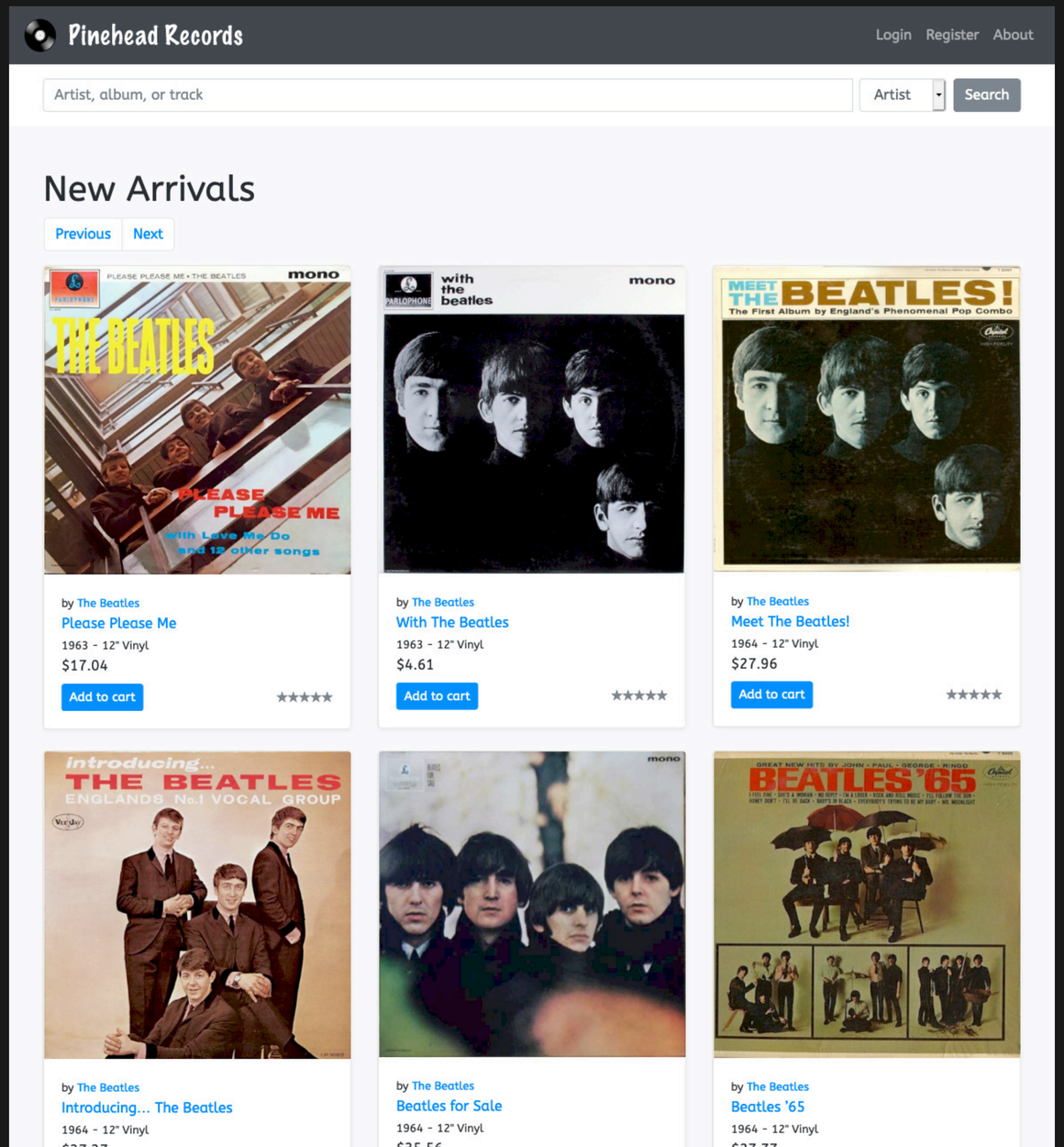
Case Study: Digital Media Service

Case Study: Digital Media Service

Linux Academy

Pinehead Records Digital Store

- **Overview**
- Defining an Entity–Relationship Model
- Documenting Access Patterns
- Data Lifecycle

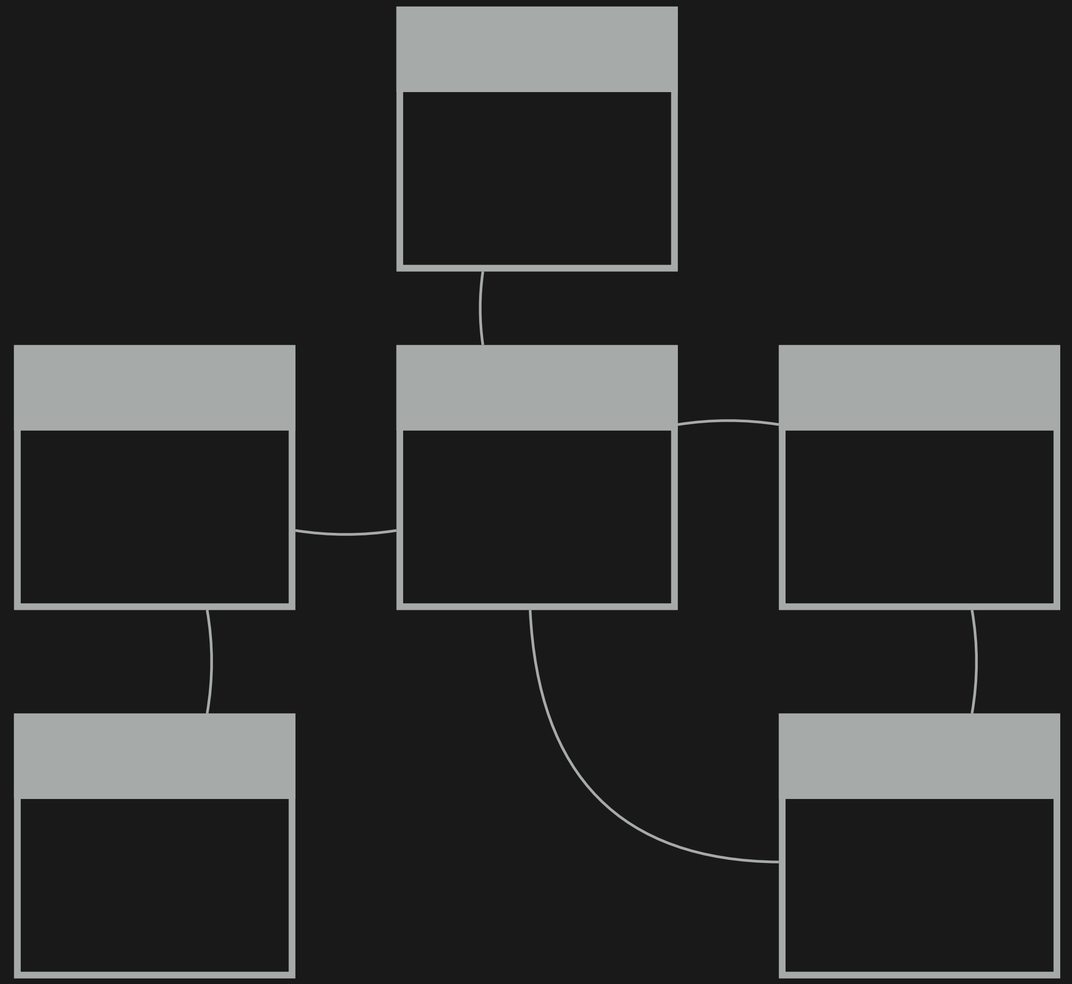


Case Study: Digital Media Service

Linux Academy

Pinehead Records Digital Store

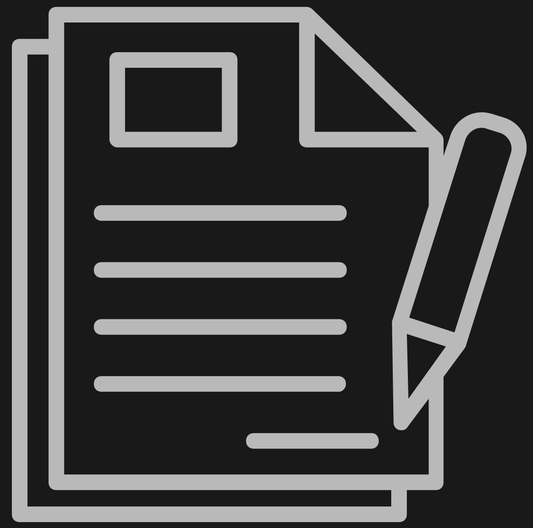
- Overview
- **Defining an Entity–Relationship Model**
- Documenting Access Patterns
- Data Lifecycle



Case Study: Digital Media Service

Pinehead Records Digital Store

- Overview
- Defining an Entity–Relationship Model
- **Documenting Access Patterns**
- Data Lifecycle

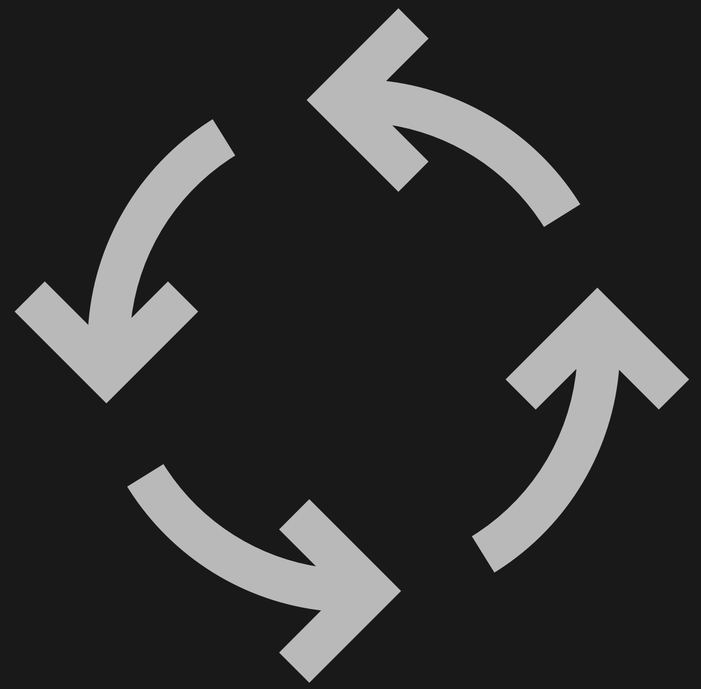


Case Study: Digital Media Service

Linux Academy

Pinehead Records Digital Store

- Overview
- Defining an Entity–Relationship Model
- Documenting Access Patterns
- **Data Lifecycle**



Linux Academy

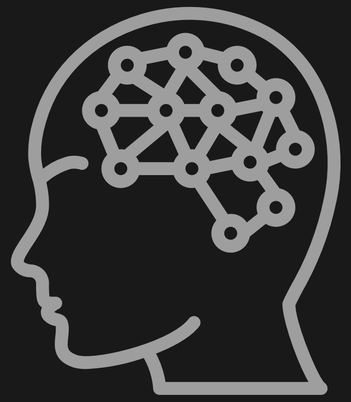
ANALYZING DATA WORKLOADS

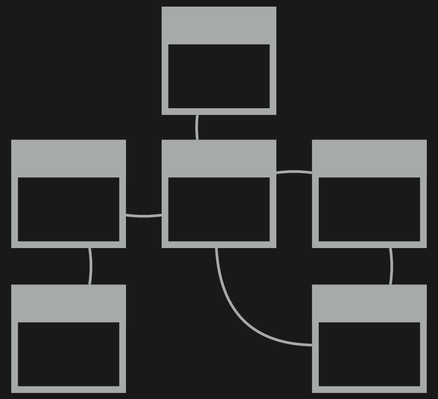
Defining an Entity–Relationship Model

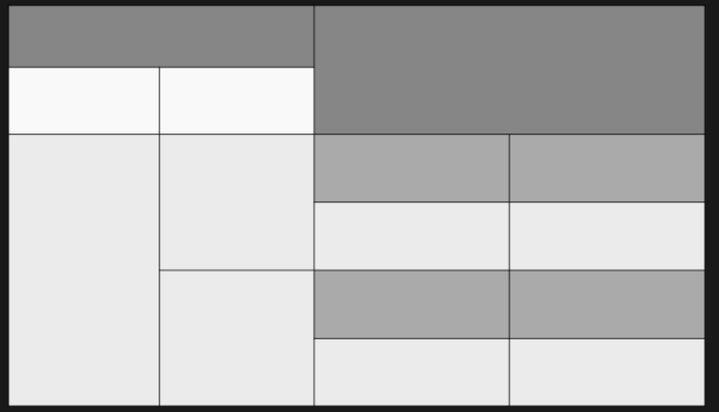
Defining an Entity–Relationship Model

Goal: Develop a system to support the given real-world tasks.

Defining an Entity–Relationship Model







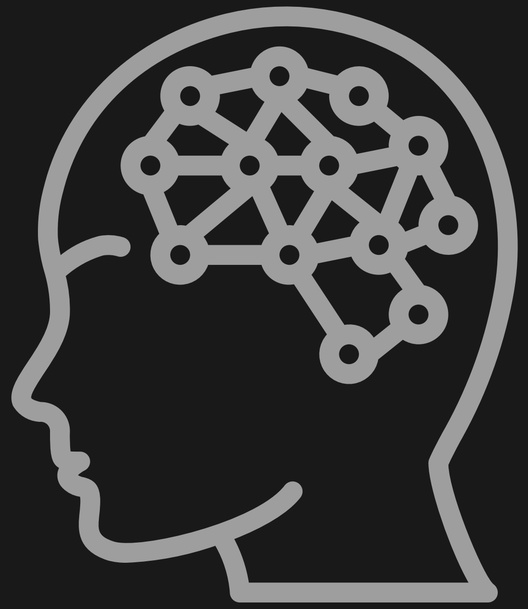
Conceptual **Logical** **Physical**

Abstract → *Concrete*

Defining an Entity–Relationship Model

Conceptual Model

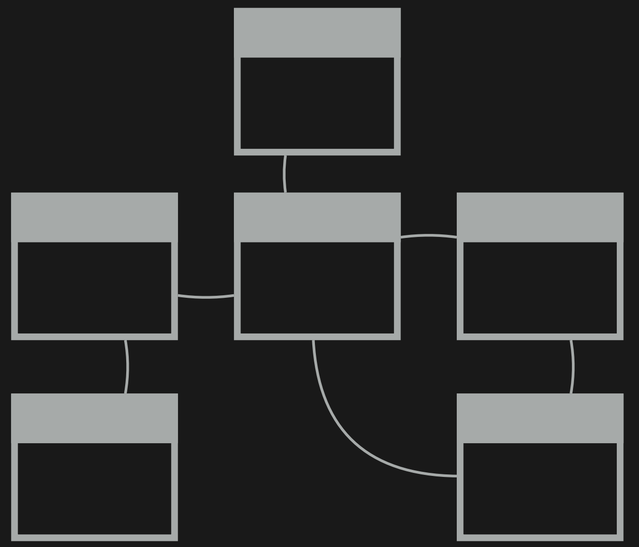
- Representation of a domain's data semantics — "meaning"
- **Entities:** Things you need to store
  - Product, Customer, Order, Order Item, Address
- **Attributes:** Characteristics of each entity
  - Product.Price, Customer.Name, Order.Total
- **Descriptors:** Attributes' requirements
  - Order number is unique
  - Customer email is required
- **Relationships:** Link entities to business rules
  - Customer places order
  - Product appears on order
- **Cardinality:** Defines the relationship degree between entities
  - Customer places 0 or more orders
  - Product appears on 0 or more orders



Defining an Entity–Relationship Model

Logical Model

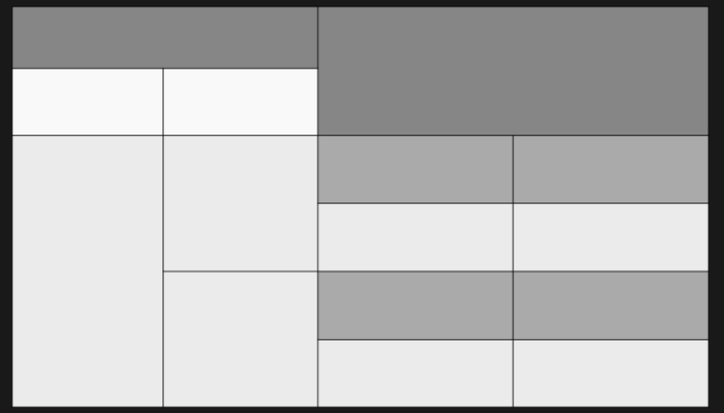
- Expands the detail from the conceptual data model
- Defines the various attributes for each entity
- Key vs. non-key attributes
  - Key defines uniqueness (e.g., ProductID)
- Primary key–foreign key relationships
  - OrderItem.ProductID is a FK to Product.ID
- Human-friendly attribute names
  - "Product Description" vs. "prd_desc"
- Database agnostic
- Data modeling tools can be used to create logical models
  - Export physical models



Defining an Entity–Relationship Model

Physical Model

- Superficially similar to a logical data model
- Implementation is vendor-specific
  - SQL vs. NoSQL
- Single table vs. multiple tables
- Attribute names are not typically human-friendly
  - e.g., GSI_SK_1
- Database-specific data types
  - String, number, binary, list, map
- Indexes and other features
- Difficult for nontechnical users to understand
  - Not typically shared with users
- High effort required to modify vs. logical model
- Tightly coupled with our **access patterns**



Linux Academy

ANALYZING DATA WORKLOADS

Documenting Access Patterns

Documenting Access Patterns

- Purpose: Data layer service for Pinehead Records Online
  - Track customer rights to albums and songs
  - Capture order information
  - Download and search metrics
- Access patterns:
  - Find album or song by title
  - Find album or song by artist
  - Get album details by ID
    - Including songs, artist(s), cover art
  - Get song details by ID
    - Including artist(s)
  - Create order
    - Include album(s) or song(s)
  - Find order by ID
  - Find order(s) by customer
  - Find customer by email

Documenting Access Patterns

- Purpose: Data layer service for Pinehead Records Online
  - Track customer rights to albums and songs
  - Capture order information
  - Download and search metrics
- Access patterns:
  - Find **album** or **song** by title
  - Find album or song by **artist**
  - Get album details by ID
    - Including songs, artist(s), cover art
  - Get song details by ID
    - Including artist(s)
  - Create **order**
    - Include album(s) or song(s)
  - Find order by ID
  - Find order(s) by **customer**
  - Find customer by email

Documenting Access Patterns

Frequency and Cost

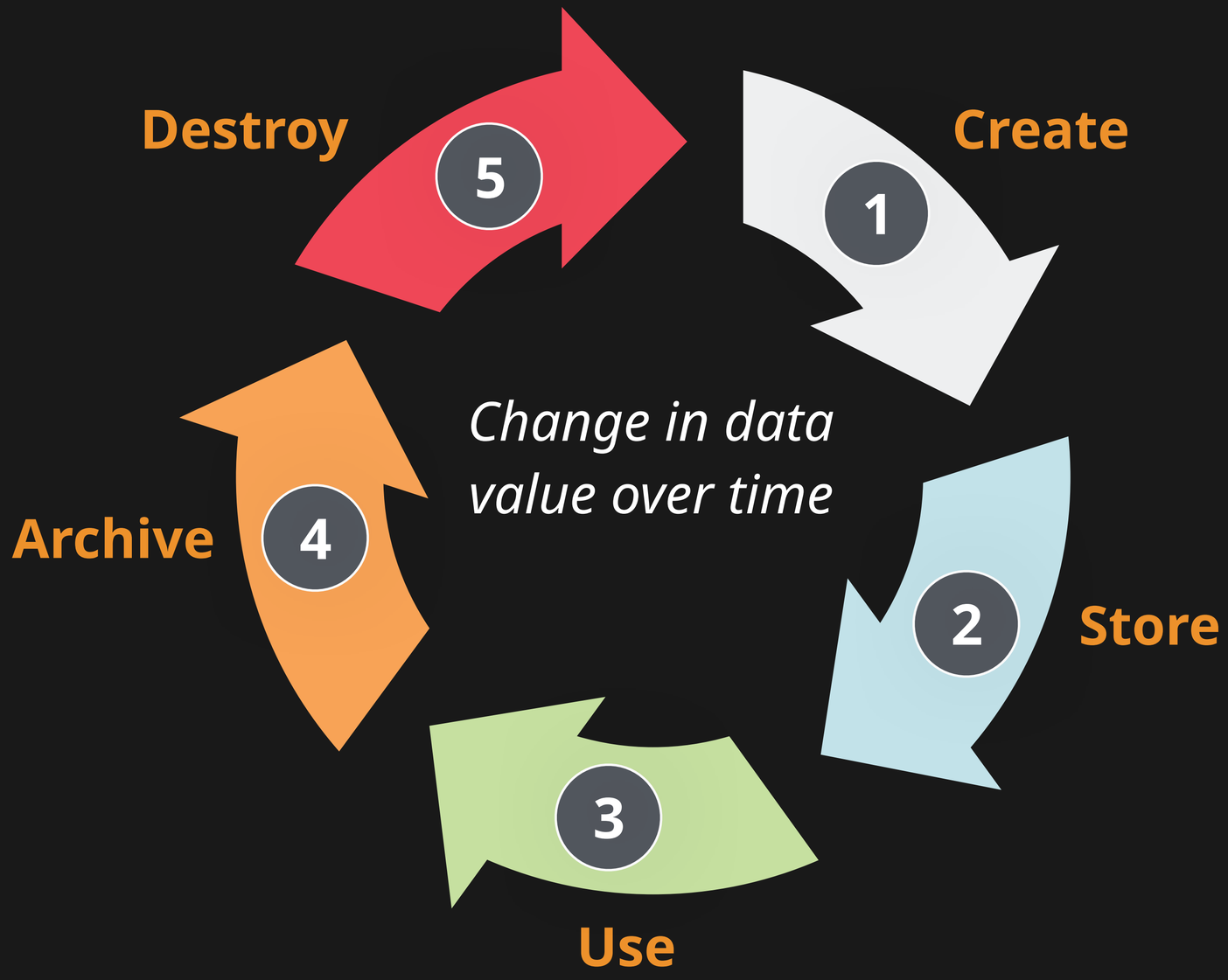
- **How often** do we exercise each access pattern?
- What is the **cost** (at scale) of performing writes and queries?
  - RCU/WCU calculations
  - Provisioned capacity
  - Auto scaling
  - On-demand
- Temporal nature of data — **how long** does it need to live?
- Consider separate tables for:
  - Rights management
  - Product catalog
  - Order history
  - Download and search metrics

Linux Academy

ANALYZING DATA WORKLOADS

Data Lifecycle

Data Lifecycle



Data Lifecycle

1 Create

- Data is first generated and collected
- Various sources:
  - **Transactional or time series**
    - High volume, high velocity
    - Write capacity and potential throttling
    - SQS write buffering + Lambda
    - Kinesis
  - **Unstructured**
    - Free text
    - Large binary objects $\Longrightarrow$ S3
  - **Structured or semi-structured**
    - CSV, JSON, XML
    - Database Migration Service

Data Lifecycle

2 Store

- S3, RDS, Kinesis, or directly to DynamoDB
- Process data before using it
  - Cleansing
  - Enrichment
  - ETL
- Data mutation for DynamoDB
  - Facilitate WCU optimization per partition
  - Create unique PK/SK combinations

Data Lifecycle

3 Use

- Apply the data to business tasks
- Governance considerations
  - Encryption
  - Fine-grained access
- Availability
  - Replication via global tables

Data Lifecycle

4 Archive

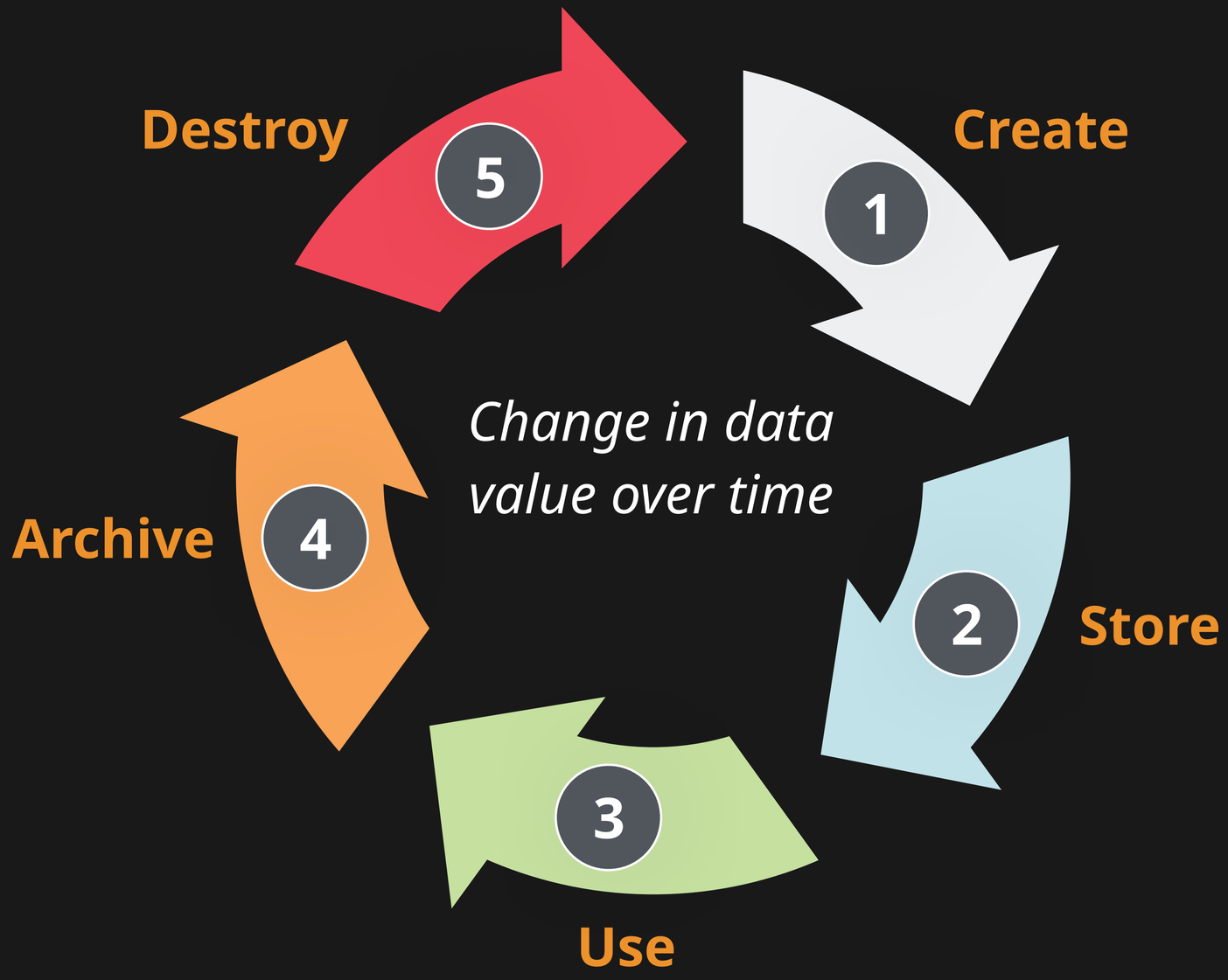
- Copy data to an environment where it's stored until needed again
- Copy to another DynamoDB table
- Copy to S3
  - EMR using Hive
  - Data Pipeline
- DynamoDB backup
- Use TTL and Lambda to archive
- Use an archival attribute
  - Non-temporal business rules

Data Lifecycle

5 Destroy

- End of life for data
- Delete every copy of a data item everywhere
  - Single data item
  - Collection of data items
  - Entire DynamoDB table
  - Archive
- Ideally, we destroy only archives
- Compliance challenge is proving data destruction was done properly
  - CloudTrail

Data Lifecycle



Linux Academy

ANALYZING DATA WORKLOADS

Optimizing for Read/Write Workloads

Optimizing for Read/Write Workloads

Read-Heavy Workloads

- Data models should provide an **efficient path** to the queries supporting the main **access patterns** of your application
- Avoid GSI bloat
  - Favor inefficient (and cheaper) scans for infrequently exercised access patterns
- Avoid unique partition keys
  - Similar to creating an index with cardinality=1
  - Treat partitions like buckets, not rows
- Be careful with legacy identifiers
  - Hot partitions
  - Identify hot keys when throttled by logging the key
- Limited projection into indexes saves money
  - Project only the attributes specific to an access pattern
  - Reduces storage costs
  - Reduces RCU consumption

Optimizing for Read/Write Workloads

Write-Heavy Workloads

- Minimize the number of GSIs
  - More GSIs == more WCUs consumed per write
- Minimize attribute projection into GSIs
  - A 1 KB write might only project a few bytes into a GSI
- Write sharding
  - Trade read cost for write scalability
  - Consider throughput per partition and partition key
- Identify hot keys when throttled by logging the key
  - Aggregate logs and perform analytics
  - CloudWatch Logs Insights

Linux Academy

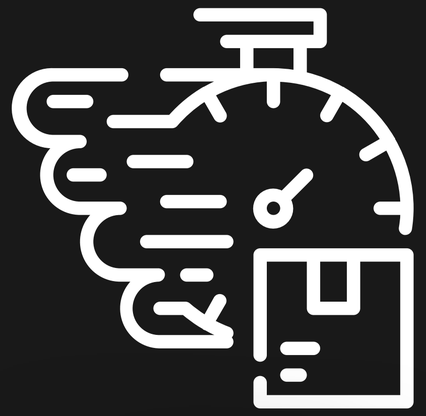
ANALYZING DATA WORKLOADS

Volume, Velocity, and Variety

Volume, Velocity, and Variety

Volume

Amount of data



Velocity

Speed of data

Variety

Diversity of data

Volume, Velocity, and Variety

How does the **amount** of data generated impact query **performance** and storage **costs**?

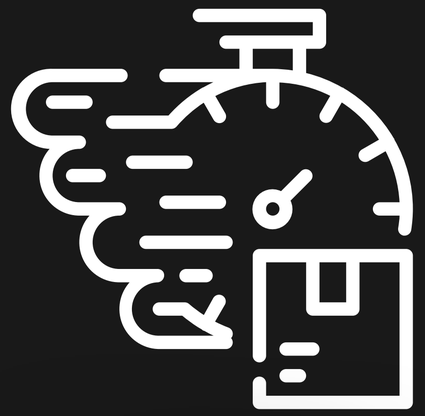
Volume

Amount of data

- DynamoDB storage: $0.25/GB-mo
  - First 25 GB free
  - 1 TB = $250/mo
- Table scans
  - RCU consumption
  - Linear search — O(N)
- Choose **parallel scans** when:
  - Table size >= 20 GB
  - Provisioned throughput (RCU) is underutilized
  - Sequential (default) scan operations are too slow
- Design your tables and indexes to use **`Query`** instead of **`Scan`**

Volume, Velocity, and Variety

How does the **speed** at which data is generated impact query **performance** and storage **costs**?



Velocity

Speed of data

- Throttling
  - WCU
  - On-demand can still throttle
  - Auto scaling is not real time
- SQS + Lambda write buffering
- Kinesis + Lambda for very high volume
  - Risk of data loss
- TTL or otherwise truncate data to reduce costs

Volume, Velocity, and Variety

How does **classifying** incoming data influence design decisions?

Variety

Diversity of data

- **Structured** data
  - CSV, JSON, YAML, XML
  - Map fields to discrete attributes
  - Store raw data in a single attribute
  - Transform data upon ingestion
- **Unstructured** data
  - Store in S3 with references in an attribute
  - Store raw in an attribute
    - 400 KB item size limit
    - Storage costs
    - RCU/WCU consumption

Linux Academy

Scenario 1: Flight Information Microservice

Linux Academy

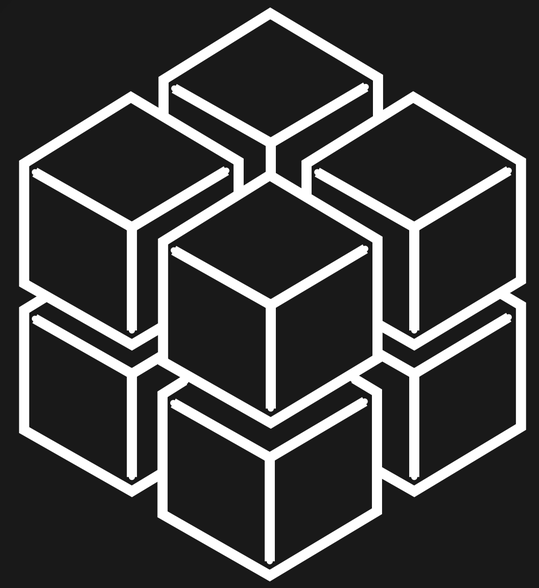
SCENARIO 1: FLIGHT INFORMATION MICROSERVICE

Scenario 1 Overview

Scenario 1 Overview

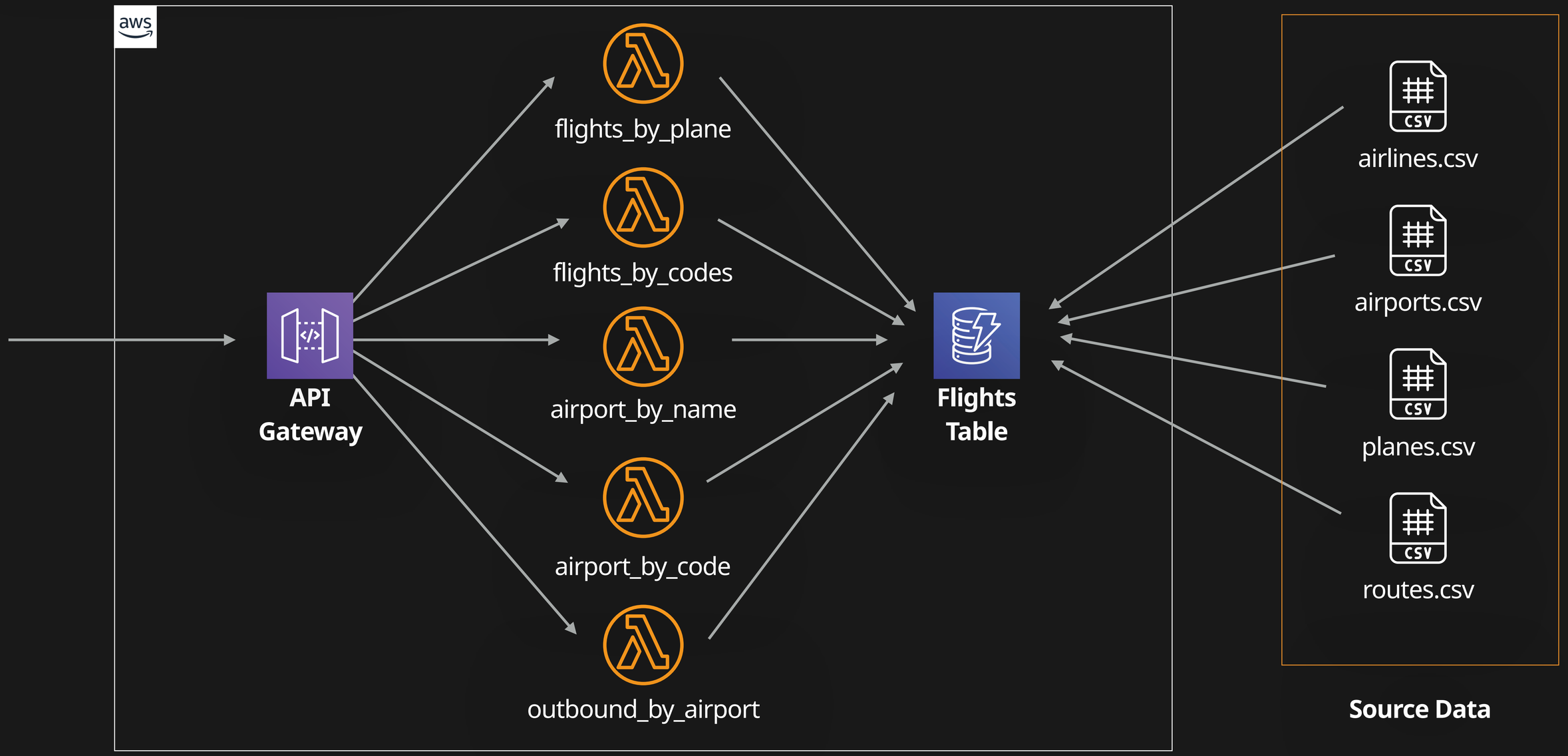
What is a **microservice**?

- Approach to designing applications as independently deployable services
- Small with a **single responsibility**
  - Each application only does one thing
- Organized around business capabilities
- Small enough to fit in your head
- Small enough that you can throw them away — **componentization**
  - Independently replaceable, **deployable**, and upgradeable
  - Prefer to rewrite over maintain
- Typically expose an API via web service request
  - JSON or SOAP over HTTP(S)
  - Remote procedure call (RPC)
- **Decentralized** data store



Scenario 1 Overview

Linux Academy



Linux Academy

SCENARIO 1: FLIGHT INFORMATION MICROSERVICE

Entities and Access Patterns

Entities and Access Patterns

Scenario: Flight Information

Access Patterns:

- Find all **flights** for a given **plane** type (e.g., Boeing 737-800)
- Find the **airport name** for a given **airport code** (e.g., MCO)
- Find the **airport code** for a given **airport name**
- Find all **flights** given ***source*** and ***destination*** **airport codes**
- Find all ***outbound*** **flights** for a given **airport code**

Linux Academy

SCENARIO 1: FLIGHT INFORMATION MICROSERVICE

Data Model

Data Model

Flights Table

Primary Key		Attributes		
Partition Key (PK)	Sort Key (SK)	GSI src_ap-index PK		GSI plane_iata-index PK
FLL	MCO	src_ap	dst_ap	plane_iata
		Fort Lauderdale...	Orlando Internat...	SF3
	LGA	src_ap	dst_ap	plane_iata
		Fort Lauderdale...	La Guardia Airport	320 321
LGA	MCO	src_ap	dst_ap	plane_iata
		La Guardia Airport	Orlando Internat...	319 M88 738
	FLL	src_ap	dst_ap	plane_iata
		La Guardia Airport	Fort Lauderdale...	321 320
SFB	MIA	src_ap	dst_ap	plane_iata
		Orlando Sanford...	Miami International...	763
	IAG	src_ap	dst_ap	plane_iata
		Orlando Sanford...	Niagara Falls Internat...	320

Data Model

Linux Academy

GSI: src_ap-index

Primary Key	Attributes KEYS_ONLY: src_ap, PK, SK	
Partition Key (src_ap)		
Fort Lauderdale–Hollywood International Airport	PK	SK
	FLL	MCO
	PK	SK
	FLL	LGA
Orlando International Airport	PK	SK
	MCO	BNA
	PK	SK
	MCO	DCA
LaGuardia Airport	PK	SK
	LGA	ORD
	PK	SK
	LGA	RSW

Data Model

Linux Academy

GSI: plane_iata-index

Primary Key	Attributes			
Partition Key (plane_iata)				
737	PK	SK	plane	...
	FCO	REG	Boeing 737	...
	PK	SK	plane	
	NAP	TLV	Boeing 737	
	PK	SK	plane	
	TIA	FCO	Boeing 737	
	PK	SK	plane	
	BOM	CJB	Boeing 737	
320	PK	SK	plane	
	CAN	JJN	Airbus A320	
	PK	SK	plane	
	SVO	OSL	Airbus A320	

SCENARIO 1: FLIGHT INFORMATION MICROSERVICE

Demo

Linux Academy

Scenario 2: Relational to DynamoDB Migration

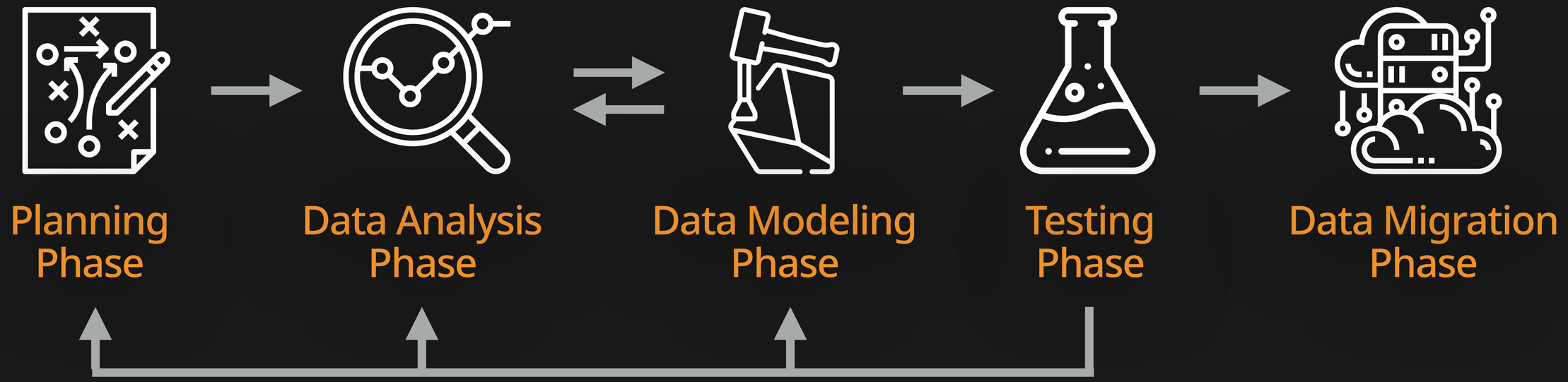
Linux Academy

SCENARIO 2: RELATIONAL TO DYNAMODB MIGRATION

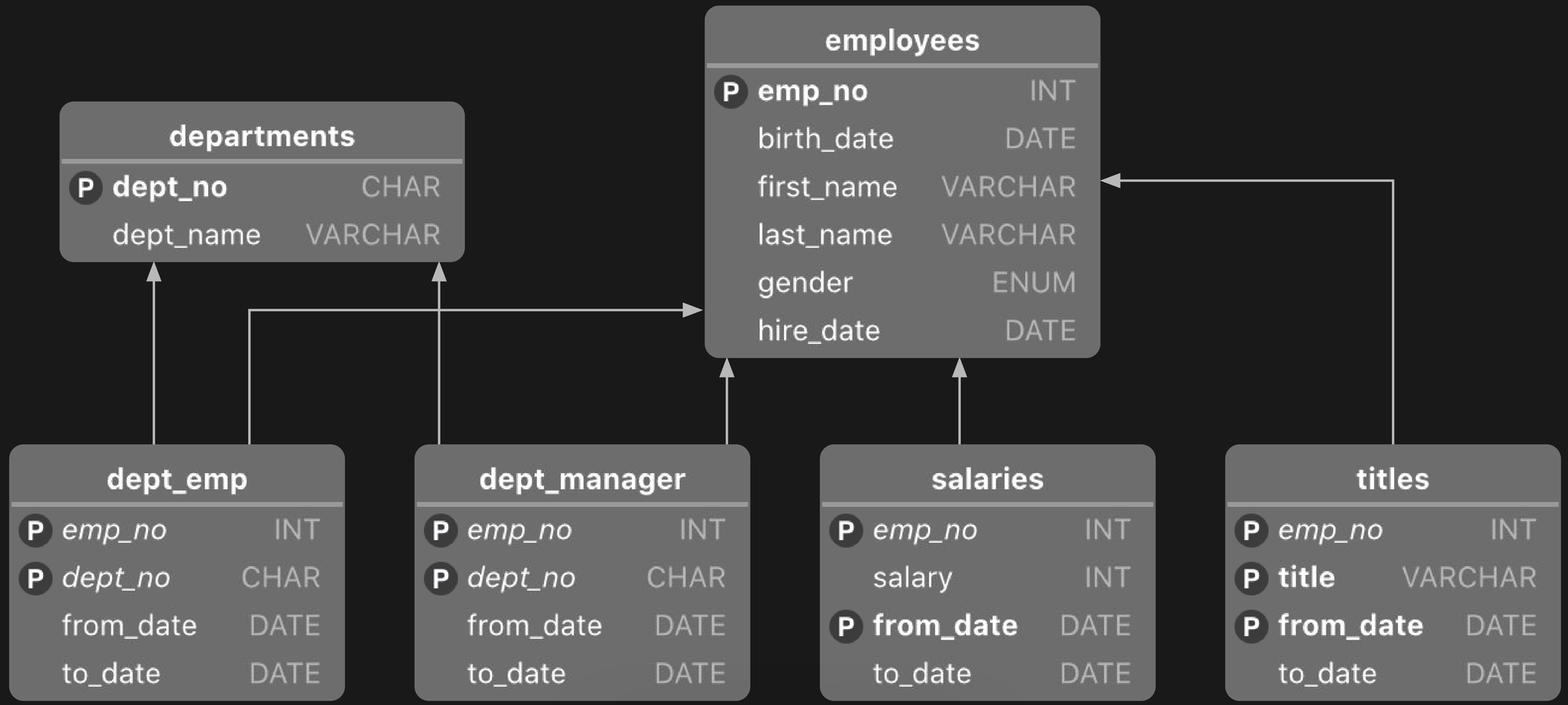
Scenario 2 Overview

Scenario 2 Overview

Five-Phase Migration Strategy



Scenario 2 Overview



Employees Schema

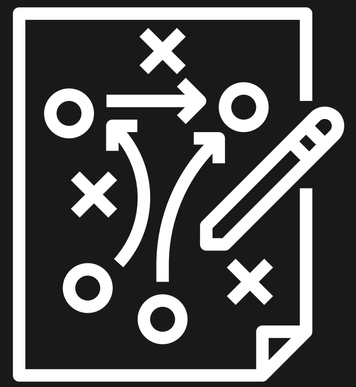
Linux Academy

SCENARIO 2: RELATIONAL TO DYNAMODB MIGRATION

Planning Phase

Planning Phase

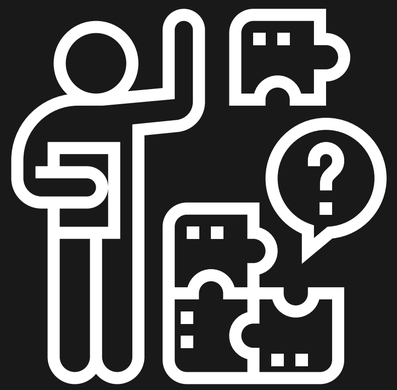
- Identify your data migration goals:
  - Increasing application performance
  - Lowering costs
  - Reducing the load on an existing RDBMS
  - ...or all of the above
- These goals inform the identification of candidate tables to migrate.
- Tables involving non-relational data are ideal migration candidates.
  - Significant improvements in application performance
  - Lower costs
  - Lower load on RDBMS
- Good migration candidates:
  - Entity–Attribute–Value tables
  - Application session state tables
  - User preference tables
  - Logging tables



Planning Phase

Problem: Writes to the RDBMS source table cannot be stopped before or during migration.

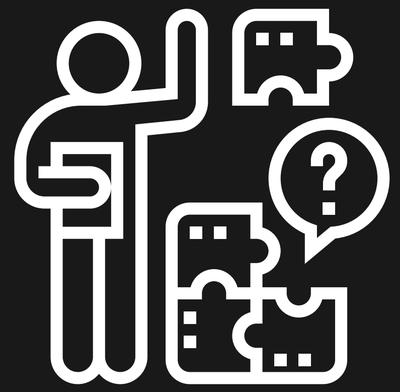
Impact: Synchronizing data with the target DynamoDB table will be difficult. Consider writing data to both the source and target tables in parallel.



Planning Phase

Problem: Volume of source data exceeds reasonable bandwidth limits.

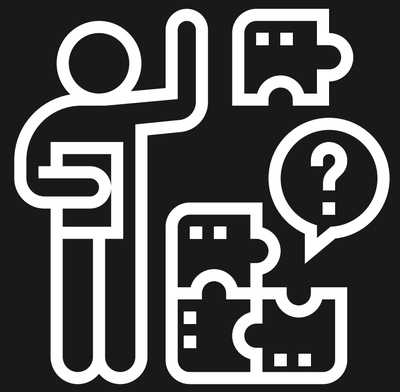
Impact: Export source data to S3 using Snowball. Import data into DynamoDB directly from S3 (e.g., Data Migration Service or custom code).



Planning Phase

Problem: Source data needs to be transformed before it can be imported into DynamoDB.

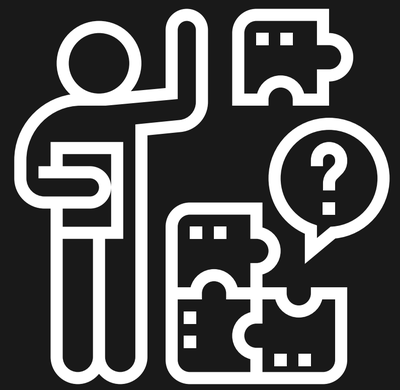
Impact: Export source data to S3. Consider using EMR to perform the transformation, and import the transformed data into DynamoDB.



Planning Phase

Problem: The primary key structure of the source tables are not portable to DynamoDB.

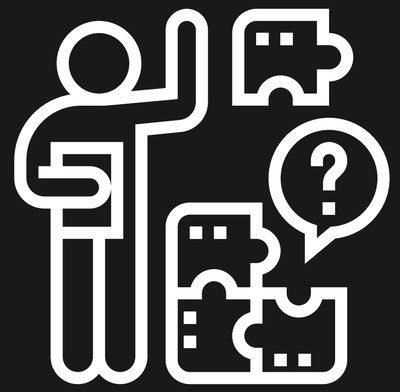
Impact: Identify fields that will make suitable partition and sort keys. Alternatively, consider using a surrogate key (e.g., UUID) to the source table that can act as a suitable partition key.



Planning Phase

Problem: Data in the source table is encrypted.

Impact: Decrypt upon export, and re-encrypt upon import using an application-level encryption scheme. The keys need to be managed outside of DynamoDB.



Ideally, consider KMS customer-managed keys.

Planning Phase

- Define and document **backup and recovery** process.
- For full cutovers from RDBMS to DynamoDB:
  - Define a process for **restoring** RDBMS in case migration fails.
  - Consider running RDBMS and DynamoDB in **parallel.**
  - Disable RDBMS after sufficient production testing.

Linux Academy

SCENARIO 2: RELATIONAL TO DYNAMODB MIGRATION

Data Analysis Phase

Data Analysis Phase

- Analysis of source data should include:
  - Number of items to import into DynamoDB
  - Distribution of item sizes
  - Multiplicity of values to be used as partition and sort keys
  - Pricing concerns:
    - Storage
    - Provisioned capacity
  - Map common SQL data types to DynamoDB types:
    - String, number, or binary
  - Focus on attributes with the largest values when estimating item size
  - Example:
    - If estimated item size = 3.3 KB, then 4 WCU (1 KB) and 1 RCU (4 KB) are required per item
  - Consider choosing an item size representing the 95th percentile for estimating costs

Data Analysis Phase

- Provision at least enough RCU/WCU to read/write 1 item/sec.
- Example:
  - If 4 WCU are required for an item of p95 size, then provision >= 4 WCU
  - Underprovisioning will cause throttling and degrade performance
- Consider applications with concurrent writes when determining provisioned capacity requirements.
- Understand the distribution of RDBMS values that could be partition or sort keys.
  - Values with nonuniform distribution are poor partition or sort keys
  - Hot keys

Data Analysis Phase

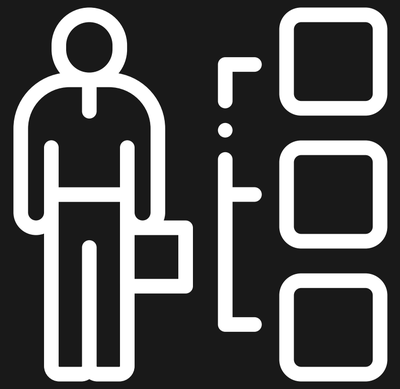
- Categorize data access patterns
- Essential to document the ways in which data will be read/written
- Critical for the data modeling phase
- Common patterns for data access:
  - Write only: Items are written to a table and never read by the application
  - Fetch by distinct value: Items are fetched individually by unique identifier
  - Query over a range: Seen frequently with temporal data

Data Analysis Phase

Scenario: Employee information

Access Patterns:

- Find **employee** by **employee ID**
- Find all **employees** for a given **last name**
- Find all **employees** for a given **title**
- Find all **employees** for a given **department**
- Find all **employees** with **salary** >= 130,000
- Find all current **department managers**



Linux Academy

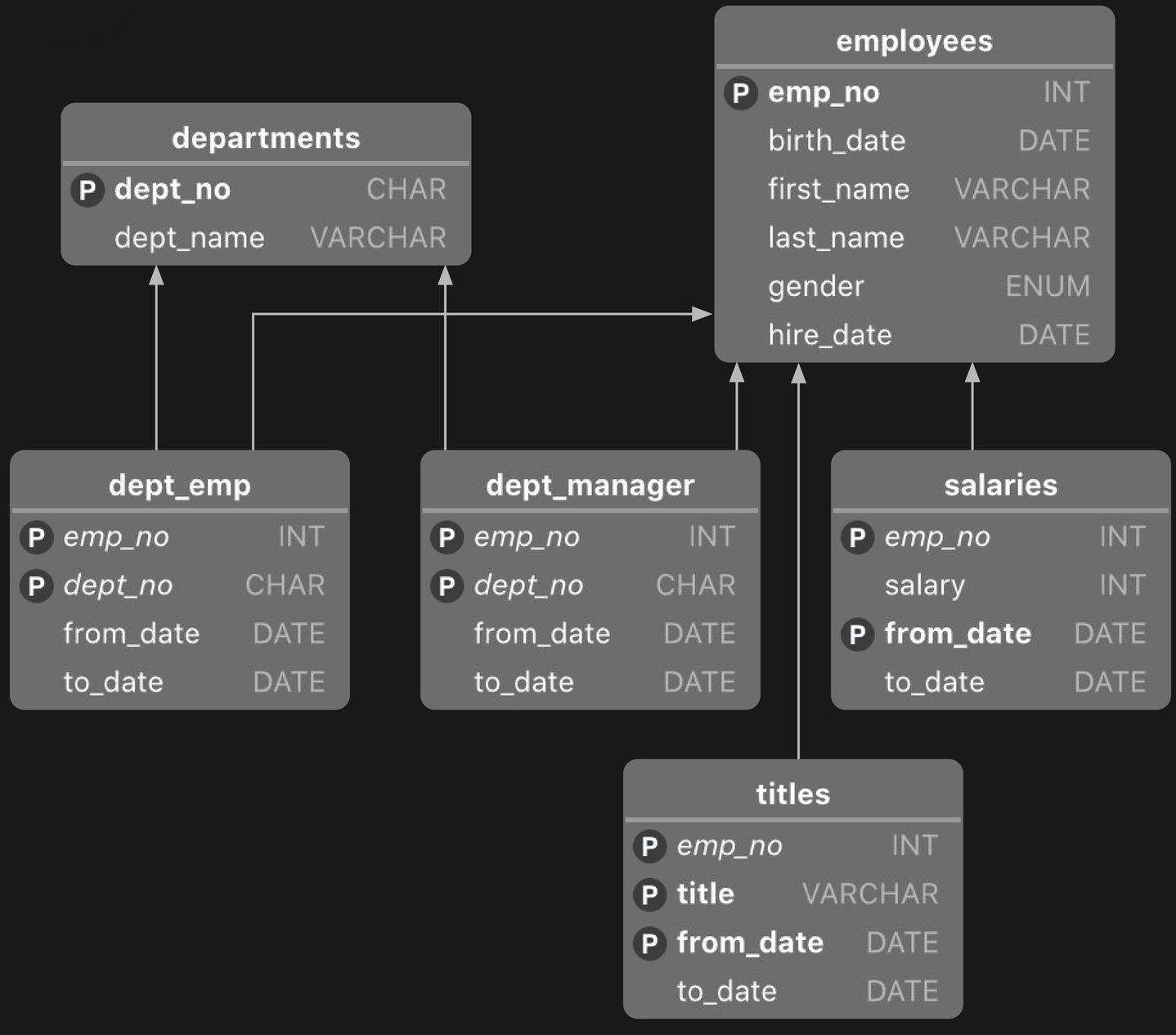
SCENARIO 2: RELATIONAL TO DYNAMODB MIGRATION

Data Modeling Phase

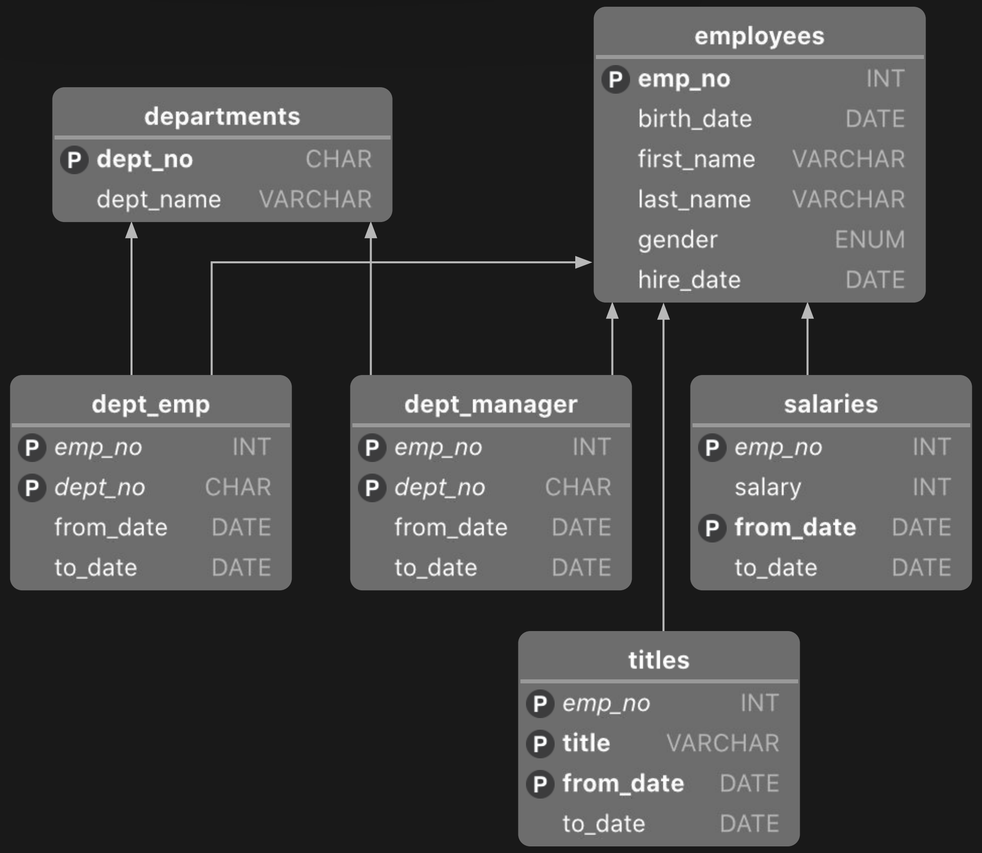
Data Modeling Phase

- Determine partition and sort keys
- Primary key must be **unique**
- Avoid using primary key of the source table as the partition key
- Prefer a "natural key" per the access patterns
- For "write only" access patterns, use a **random** numeric ID for the partition key
- Source tables with an index on **two key values** are good candidates for a primary key using **both** a partition and sort key
- Indexes on **non-key columns** of a source table are also good LSI/GSI candidates

Data Modeling Phase



Data Modeling Phase




```sql
CREATE VIEW dms_source AS
SELECT
  employees.*,
  titles.title,
  titles.from_date AS title_from_date,
  titles.to_date AS title_to_date,
  salaries.salary,
  salaries.from_date AS salary_from_date,
  salaries.to_date AS salary_to_date,
  dept_emp.dept_no AS department_number,
  departments.dept_name AS department_name,
  dept_emp.from_date AS dept_from_date,
  dept_emp.to_date AS dept_to_date,
  CONCAT_WS(' ', manager.first_name, manager.last_name)
    AS manager_name,
  manager.emp_no AS manager_emp_no
FROM employees
  INNER JOIN titles ON employees.emp_no = titles.emp_no
  INNER JOIN salaries ON employees.emp_no = salaries.emp_no
  INNER JOIN dept_emp ON employees.emp_no = dept_emp.emp_no
  INNER JOIN departments ON dept_emp.dept_no = departments.dept_no
  INNER JOIN dept_manager dm ON dm.dept_no = departments.dept_no
  INNER JOIN employees manager ON manager.emp_no = dm.emp_no;

CREATE TABLE dms_source_materialized SELECT * FROM dms_source;
```

Data Modeling Phase

emp_no	birth_date	first_name	last_name	gender	hire_date	title	title_from_date	title_to_date	salary	salary_
10011	1953-11-07	Mary	Sluis	F	1990-01-22	Staff	1990-01-22	1996-11-09	42365	1990-(
10011	1953-11-07	Mary	Sluis	F	1990-01-22	Staff	1990-01-22	1996-11-09	44200	1991-(
10011	1953-11-07	Mary	Sluis	F	1990-01-22	Staff	1990-01-22	1996-11-09	48214	1992-(
10011	1953-11-07	Mary	Sluis	F	1990-01-22	Staff	1990-01-22	1996-11-09	50927	1993-(
10011	1953-11-07	Mary	Sluis	F	1990-01-22	Staff	1990-01-22	1996-11-09	51470	1994-(
10011	1953-11-07	Mary	Sluis	F	1990-01-22	Staff	1990-01-22	1996-11-09	54545	1995-(
10011	1953-11-07	Mary	Sluis	F	1990-01-22	Staff	1990-01-22	1996-11-09	56753	1996-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	40000	1989-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	43527	1990-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	46509	1991-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	49874	1992-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	52969	1993-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	56629	1994-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	58079	1995-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	60322	1996-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	62274	1997-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	62517	1998-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	62458	1999-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	64238	2000-(
10038	1960-07-20	Huan	Lortz	M	1989-09-20	Senior Staff	1996-09-20	9999-01-01	64254	2001-(

Data Modeling Phase

Linux Academy

Employees Table

Primary Key		Attributes				
Partition Key (emp_no)	**Sort Key (salary)**					
16998	40000	first_name	last_name	department_name	dept_to_date	salary_to_date
		Katsuyuki	Weedon	Customer Service	9999-01-01	2002-04-06
	42817	first_name	last_name	department_name	dept_to_date	salary_to_date
		Katsuyuki	Weedon	Customer Service	9999-01-01	9999-01-01
35176	57184	first_name	last_name	department_name	dept_to_date	salary_to_date
		Jiafu	Wilharm	Research	9999-01-01	2002-01-20
	57561	first_name	last_name	department_name	dept_to_date	salary_to_date
		Jiafu	Wilharm	Research	9999-01-01	9999-01-01

Access pattern: Find employee by employee ID

Data Modeling Phase

GSI: **title-index**

Primary Key	Attributes				
Partition Key (title)					
Senior Staff	first_name	last_name	department_name	dept_to_date	salary_to_date
	Katsuyuki	Weedon	Customer Service	9999-01-01	2002-04-06
	first_name	last_name	department_name	dept_to_date	salary_to_date
	Jiafu	Wilharm	Research	9999-01-01	9999-01-01

Access pattern: Find all employees for a given title

Data Modeling Phase

GSI: **last_name-index**

Primary Key	Attributes				
Partition Key (last_name)					
Weedman	first_name	title	department_name	dept_to_date	salary_to_date
	Uri	Senior Engineer	Customer Service	9999-01-01	2002-04-06
	first_name	title	department_name	dept_to_date	salary_to_date
	Aloke	Staff	Research	9999-01-01	9999-01-01

Access pattern: Find all employees for a given last name

Data Modeling Phase

GSI: **salary-index**

Primary Key		Attributes			
Partition Key (salary_to_date)	**Sort Key (salary)**				
9999-01-01	**135,791**	first_name	title	department_name	dept_to_date
		Alice	Senior Engineer	Customer Service	9999-01-01
		first_name	title	department_name	dept_to_date
		Bob	Staff	Research	9999-01-01
9999-01-01	**54,321**	first_name	title	department_name	dept_to_date
		Charles	Senior Engineer	Customer Service	9999-01-01
		first_name	title	department_name	dept_to_date
		David	Staff	Research	9999-01-01

Access pattern: Find all employees with salary >= 130,000

Data Modeling Phase

GSI: **department_name-index**

Primary Key	Attributes			
Partition Key (department_name)				
Customer Service	first_name	last_name	dept_to_date	emp_no
	Alice	Jones	9999-01-01	12345
	first_name	last_name	dept_to_date	emp_no
	Bob	Wilson	9999-01-01	3454
Research	first_name	last_name	dept_to_date	emp_no
	Charles	Parker	9999-01-01	84214
	first_name	last_name	dept_to_date	emp_no
	David	Ellefson	9999-01-01	4554

Access pattern: Find all employees for a given department

Data Modeling Phase

GSI: department_manager-index

Primary Key		Attributes	
Partition Key (department_name)	**Sort Key (manager_name)**		
Customer Service	**Yuchang Weedman**	dept_to_date	...
		9999-01-01	...
		dept_to_date	...
		9999-01-01	...
Research	**Mikhail Undy**	dept_to_date	...
		9999-01-01	...
		dept_to_date	...
		9999-01-01	...

Access pattern: Find all current department managers

Linux Academy

SCENARIO 2: RELATIONAL TO DYNAMODB MIGRATION

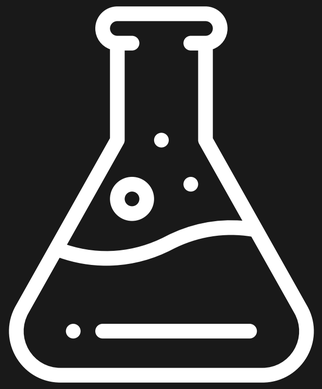
Testing Phase

Testing Phase

Test Category: Basic acceptance tests

Purpose:

- Verify whether data migration was successful
- Automatically executed upon data migration completion
- Common outputs:
  - Total # items processed
  - Total # items imported
  - Total # items skipped
  - Total # warnings
  - Total # errors

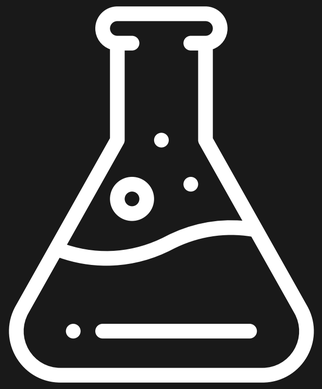


Testing Phase

Test Category: Functional tests

Purpose:

- Identify problems caused by RDBMS → DynamoDB migration
- Exercise the application using DynamoDB
- Combination of automated and manual tests
- Gaps in the data model are revealed

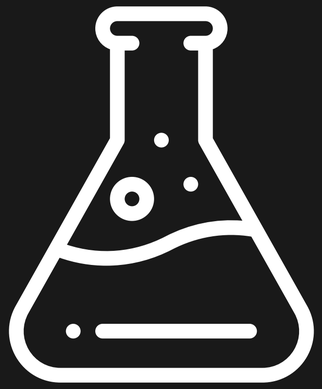


Testing Phase

Test Category: Non-functional tests

Purpose:

- Assess non-functional characteristics:
  - Performance under varying load
  - Resiliency to failure of any part of the app stack
  - May include edge cases
    - e.g., a large number of clients try to update the same item simultaneously
  - Include backup and recovery process

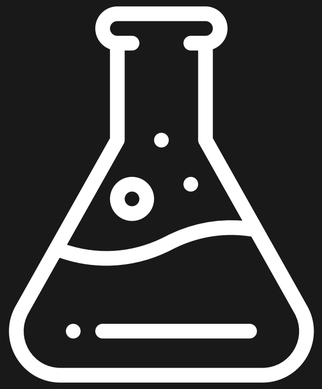


Testing Phase

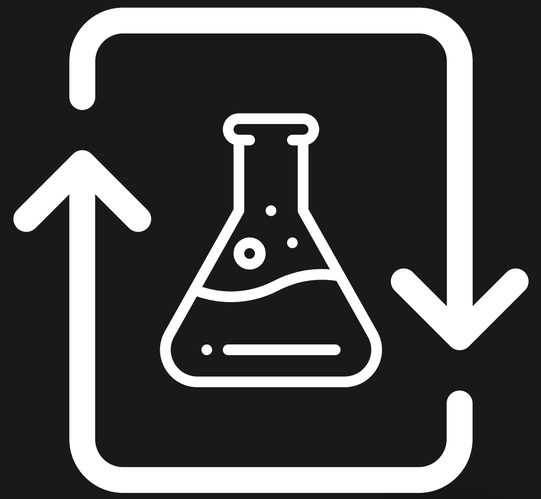
Test Category: User acceptance tests

Purpose:

- End users decide if the application meets requirements
- Executed by end users after final data migration has completed



Testing Phase



Testing activities are **iterative**

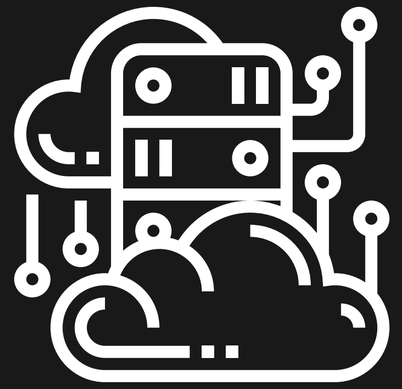
Linux Academy

SCENARIO 2: RELATIONAL TO DYNAMODB MIGRATION

Data Migration Phase

Data Migration Phase

- Full set of production data is migrated to DynamoDB
- End-to-end data migration process already tested
- Full process documentation
- Notify application users of potential downtime
- Final user acceptance testing
- If migration fails, execute backup and recovery procedure
  - Root cause analysis
- Monitor key performance indicators



Data Migration Phase

Linux Academy

Paaaaaartay!

Invite all your friends and have soda and pie

Linux Academy

SCENARIO 2: RELATIONAL TO DYNAMODB MIGRATION

Demo

Linux Academy

SCENARIO 2: RELATIONAL TO DYNAMODB MIGRATION

Evolving a Data Model

Evolving a Data Model

Scenario: Supporting a new access pattern

Access Pattern: Find all employees with tenure > N years

Problem: This could be solved with an LSI on `salary_to_date = 9999-01-01`; however, LSIs can only be added at table creation time.

Solution:

1. Batch job to parallel scan the table for `salary_to_date = 9999-01-01` and add a new attribute: `is_current = 1`
2. Create a GSI on `is_current` with sort key `hire_date`

Considerations: If salary, title, or department change, then we can update this on a per-employee basis — no need to run the batch job again.

Linux Academy

SCENARIO 2: RELATIONAL TO DYNAMODB MIGRATION

Demo

Linux Academy

COURSE CONCLUSION

What's Next?

What's Next?

Further Learning

- Amazon DynamoDB Deep Dive
- Automating AWS with Lambda, Python, and Boto3
- AWS Certified SysOps Administrator - Associate

Contact Me

twitter.com/mrichman

linkedin.com/in/mrichman