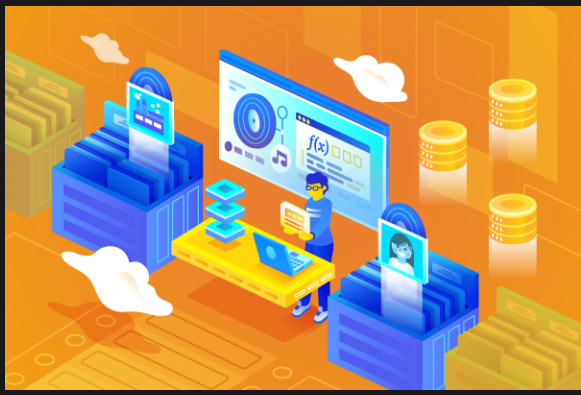
Amazon DynamoDB Deep Dive

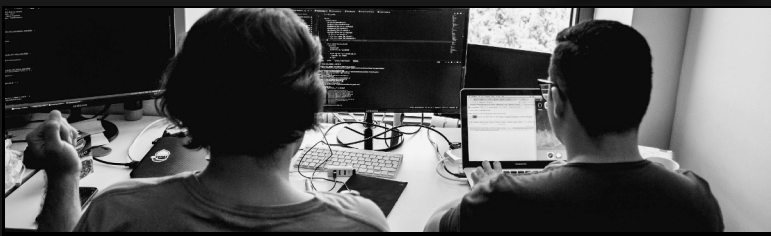
Course Navigation



Linux Academy

Course Introduction

Getting Started



Dear students,

Welcome to the Linux Academy **Amazon DynamoDB Deep Dive** course. This course will help you master DynamoDB!

In this course, you will learn the basics of DynamoDB and how it differs from traditional relational database management systems. A real-world scenario project will help guide you through each of the concepts presented.

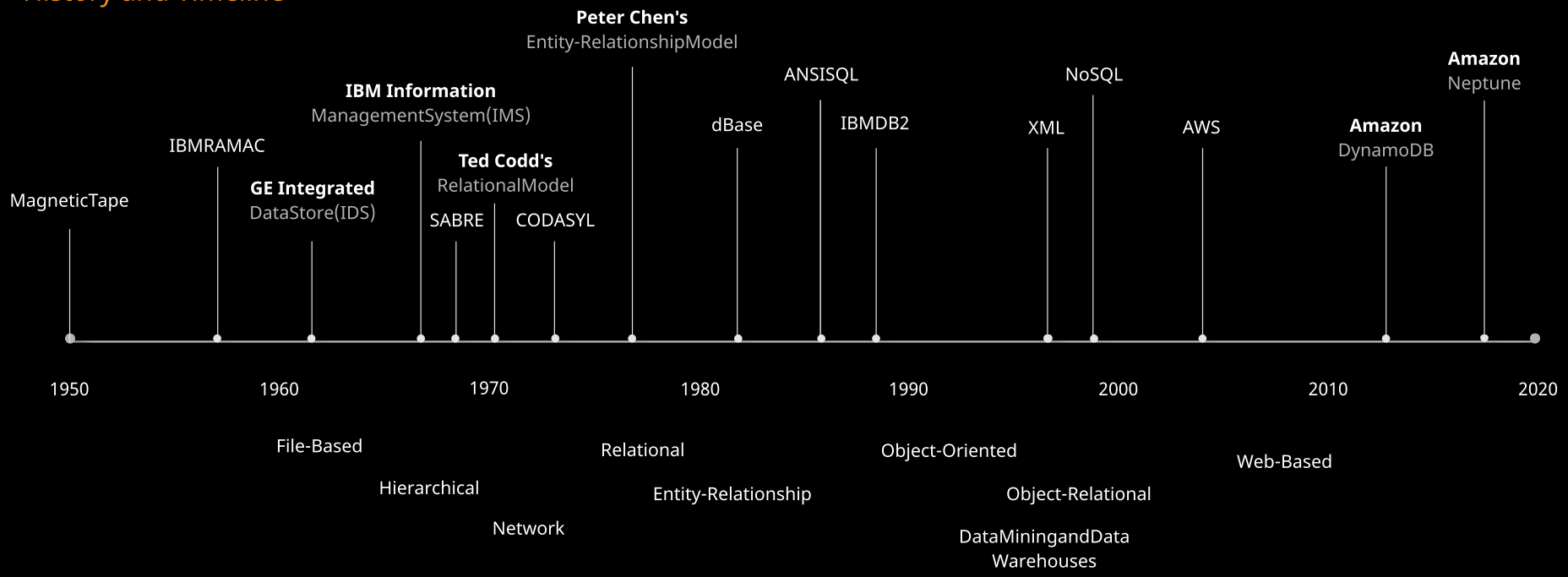
This course is intended for all skill levels, even if you're totally new to database development.

By the end of this course, you'll have the necessary skills and knowledge to effectively build scalable, high performance applications using DynamoDB.

Thank you for taking the course. Let's get started!

Warm Regards,
Mark Richman
AWS Training Architect
Linux Academy

History and Timeline



Next

Microsoft Excel Is Not a Database ... or Is It?

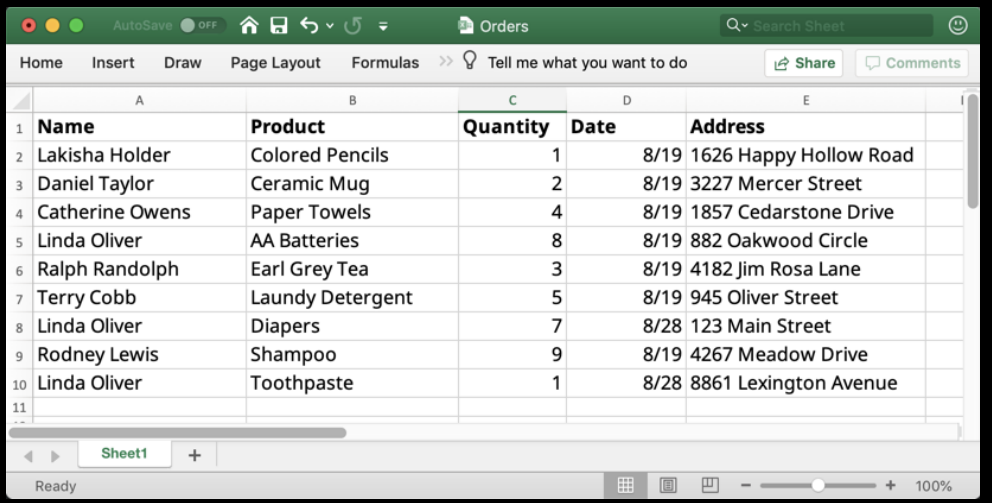
Many people have typically used flat files or **spreadsheet software** as a type of database. In this example, orders are entered into a spreadsheet as they come in.

Over time, redundant data accumulates, along with **potentially contradicting values.**

For example, if Linda Oliver called in about her order, you would see three different entries on different dates, each with a different address.

This situation can easily lead to incorrect shipments, lost revenue, and unhappy customers.

How do we resolve this?



	Name	Product	Quantity	Date	Address
2	Lakisha Holder	Colored Pencils	1	8/19	1626 Happy Hollow Road
3	Daniel Taylor	Ceramic Mug	2	8/19	3227 Mercer Street
4	Catherine Owens	Paper Towels	4	8/19	1857 Cedarstone Drive
5	Linda Oliver	AA Batteries	8	8/19	882 Oakwood Circle
6	Ralph Randolph	Earl Grey Tea	3	8/19	4182 Jim Rosa Lane
7	Terry Cobb	Laundy Detergent	5	8/19	945 Oliver Street
8	Linda Oliver	Diapers	7	8/28	123 Main Street
9	Rodney Lewis	Shampoo	9	8/19	4267 Meadow Drive
10	Linda Oliver	Toothpaste	1	8/28	8861 Lexington Avenue

Spreadsheets are indeed databases in the same way flat text files are, but they lack some fundamental features, which we'll cover shortly.

Relational Data

Instead of having just one massive spread sheet, you could seperate the information in to **different tables.** Each table represents an entity in your system.

Orders

ID	CustomerID	ProductID	Quantity	Date
76354	13579	238	1	$4.99
76355	13580	239	2	$13.98
76356	13581	240	1	$13.95
76357	13582	241	3	$35.97
76358	13583	242	2	$22.77
76359	13584	243	1	$34.95
76360	13585	244	2	$13.90

Customers

ID	Name	Address
13579	LakishaHolder	126HappyHollowRoad
13580	DanielTaylor	3227MercerStreet
13581	CatherineOwens	1857CedarstoneDrive
13582	LindaOliver	882OakwoodCircle
13583	RalphRandolph	4182JimRosaLane
13584	TerryCobb	945OliverStreet
13585	RodneyLewis	4267MeadowDrive

Products

ID	Name	Price
238	Colored Pencils	$4.99
239	Ceramic Mug	$6.99
240	Paper Towels	$13.95
241	AA Batteries	$11.99
242	Early Gray Tea	$7.59
243	Laundry Detergent	$14.89
244	Diapers	$34.95
245	Shampoo	$6.95

Tables are linked, or related, to each other using various IDs for reference. This is what forms a **relational database**.

Linux Academy

Terminology

Table: A collection of related data, organized in rows and columns

Row: A set of related values for all the columns in a given table. Often called a **record, item, or tuple.**

Column: A unit of named data that has a particular data type (e.g., number, text, or date)

Schema: Overall structure of the database, relationships between tables, and the rules that govern them

Key: Columns on which an index is constructed to allow rapid and/or sorted access to a table's row

- **Candidate Key:** Any set of columns that can uniquely identify a row
- **Primary Key:** A candidate key that may uniquely identify any given row
- **Foreign Key:** Uniquely identifies a value (primary key) in a different table

Query: A data retrieval statement that specifies the columns and tables from which data is to be retrieved; and optionally:

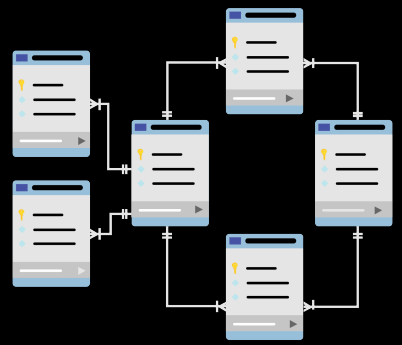
- Conditions the data must satisfy
- Computations to be performed on the retrieved column values
- A desired ordering of the result set
- **Examples:**
  - `SELECT * FROM Customer WHERE ID = 76355`
  - `SELECT ID, Price FROM Product ORDER BY Price DESC`
  - `SELECT * FROM Order WHERE Date = 2019-04-02 AND Total > 100`

Columns

Rows

ID	CustomerID	ProductID	Quantity	Total
76354	13579	238	1	$4.99
76355	13580	239	2	$13.98
76356	13581	240	1	$13.95
76357	13582	241	3	$35.97
76358	13583	242	2	$22.77
76359	13584	243	1	$34.95
76360	13585	244	2	$13.90

Primary Key **Foreign Key** **Foreign Key**



Schema

Linux Academy

Database Components

1. **Hardware** — Physical electronic devices such as computers, I/O devices, storage devices, etc. This provides the **interface** between computers and real-world systems.
2. **Software** — Programs used to **control and manage the overall database**. This includes the DBMS software itself, operating system, network software used to share the data among users, and the application programs used to access data in the DBMS.
3. **Data** — The most important component of the **DBMS**. Data can be either the actual operational data or metadata.
4. **Data Access Method** — Programming language or similar commands to access, query, and **manipulate the data** in the DBMS.
5. **Users** — People who need the information from the database to carry out their primary **business responsibilities**.

Back

Back to Main

ACID Properties in Relational Databases

A sequence of database operations that satisfies these properties is called a **transaction**.
In order to guarantee validity of data after a transaction, operations must satisfy these properties:

A
Atomicity
A sequence of **operations** where either all occur or nothing occurs.

C
Consistency
The state of the **database**, both before and after execution of an operation, remains consistent (i.e., free of any data integrity errors) whether or not the transaction commits or is rolled back.

I
Isolation
Changes made by a transaction are **isolated from the rest** of the system until after the transaction has committed.

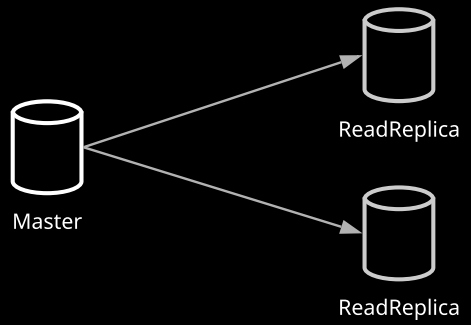
D
Durability
Committed data is saved by the system such that, even in the event of a failure and system restart, the data is available in its correct state.

Next

Back to Main

SQL Databases and Scaling

Replication
Too much load



Pros:
- Offloads master node
- Read replicas can scale horizontally

Cons:
- Does not increase the speed of writes
- Replication lag

Sharding
Too much data

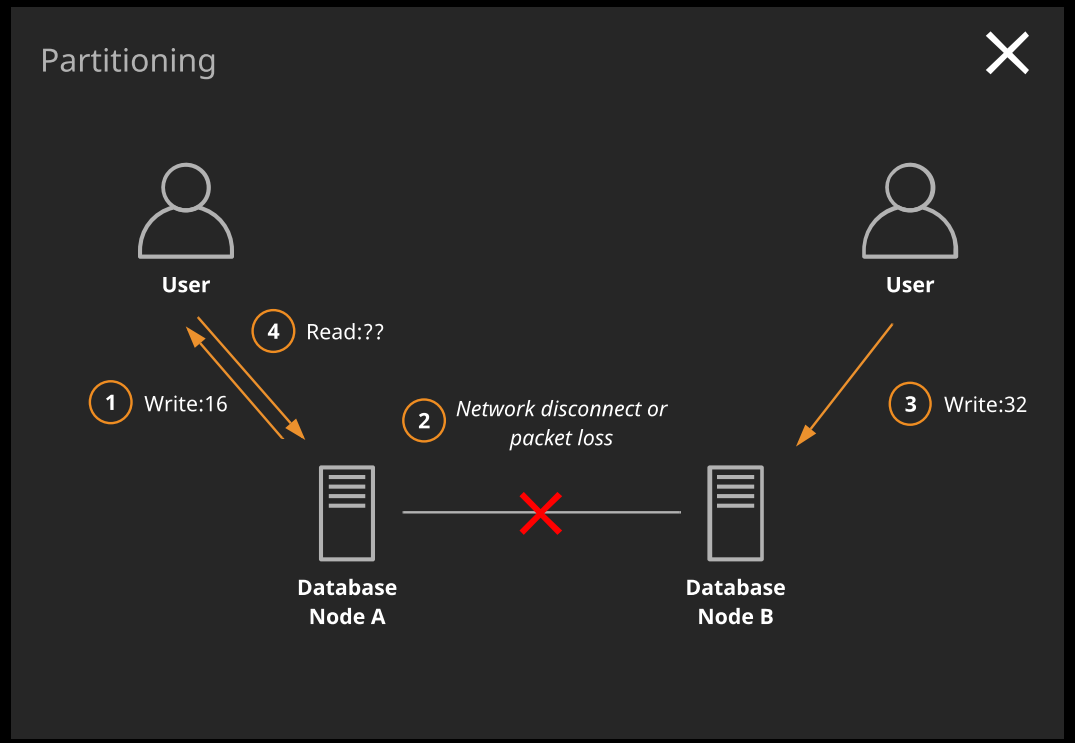
Pros:
- Can store larger data sets
- Can handle more load than a single master node

Cons:
- Queries become more complex
  - Ex: Range queries might hit all nodes
- Joins become difficult — your tables are no longer all on the same node

Back

Next

Back to Main



CAP Theorem

How do we overcome the limitations of **SQL databases**? Scaling horizontally comes with tradeoffs ...

The **CAP theorem** states that it is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:

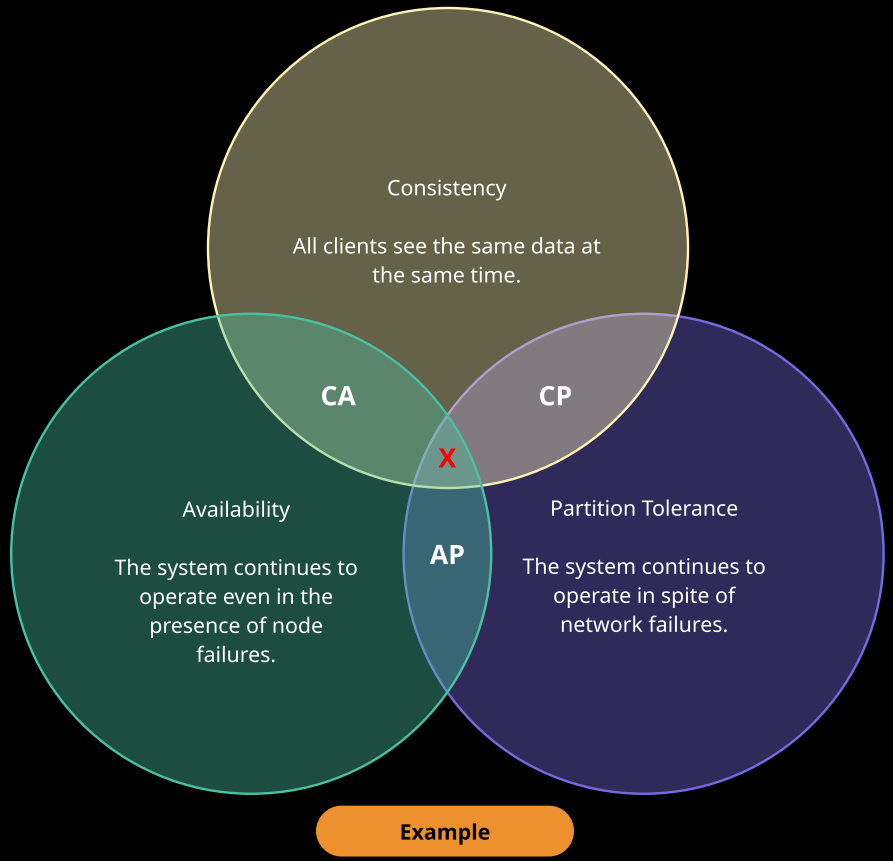
- **CA:** Data is consistent between all nodes — as long as all nodes are online — and you can read/write from any node and be sure the data is the same, but if you ever develop a partition between nodes, the data will be out of sync (and won't re-sync once the partition is resolved).
- **CP:** Data is consistent between all nodes and maintains partition tolerance (preventing data de-sync) by becoming unavailable when a node goes down.
- **AP:** Nodes remain online even if they can't communicate with each other and will re-sync data once the partition is resolved, but you aren't guaranteed that all nodes will have the same data (either during or after the partition).

In a **distributed system**, we always need to select partition tolerance; we need to tolerate packet loss on the network level. So in reality, we only have a choice between consistency and availability (CA).

Given disconnected partitions (packet loss), a database node can either **return an error or stale data**.

SQL databases select both C and A. They suffer from partition intolerance and generally run on only one machine. Replication technologies do not select both C and A.

NoSQL databases select either C or A.



Example

Back

Introduction to NoSQL

NoSQL databases are built for specific data models and offer flexible schemas. These databases are well suited to modern applications such as mobile, web, and gaming that require high scalability, flexibility, and performance.

SQL	NoSQL
Optimized for storage	**Optimized for compute**
Normalized/relational	Denormalized/hierarchical
Ad-hocqueries	Instantiatedviews
Scalevertically	Scalehorizontally
GoodforOLAP	BuiltforOLTPatscale

Next

Back to Main

NoSQL Engine Types

Key-value: Uses a simple key-value method to store data. Stored as a collection of key-value pairs, where the key serves as a unique identifier. Schema is defined per item.

Use cases: Session store, shopping cart

Document-oriented: Designed to store JSON-like data

Use cases: Content management, catalogs

Column-oriented: Stores each column of data in sequential blocks on disk or in memory. Ideal for analytical queries that perform aggregate operations (sum, avg, etc.) over a small number of columns. **Takes advantage of compression**, as similar data is stored together. Slow for single record operations.

Use cases: Analytics, data warehouses

Graph: Designed to store relationships. Data is stored as nodes and edges. Very fast because relationships are not calculated at query time but persisted in the database.

Use cases: Fraud detection, recommendation engines

ProductID	Attributes
1234	Book,Odyssey,Homer
3284	Album,6Partitas,Bach
5283	Movie,Dune,Lynch
9833	Movie,StarWars,Lucas

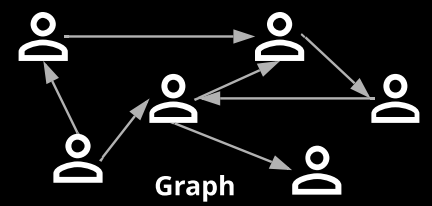
Key-Value

Date	Price	Size
2019-01-20	10.1	10
2019-01-21	10.3	20
2019-01-22	10.5	40
2019-01-23	11.2	3

Column-Oriented

```
[
  {
    year: 2020,
    title: 'Dune',
    info: {
      directors: ['Denis Villeneuve', 'Not David Lynch'],
      release_date: '2020-12-18T00:00:00Z',
      rating: 9.2,
      genres: ['Fantasy', 'Sci-Fi'],
      image_url: 'http://example.com/images/foo.jpg',
      plot: 'Paul Atreides rides worms around Arrakis.',
      actors: ['Timothée Chalamet', 'Zendaya']
    }
  }
]
```

Document-Oriented



Graph

ACID vs. BASE

Basic Availability: The database appears to work most of the time.

Soft state: Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.

Eventual consistency: Stores exhibit consistency at some later point (e.g., lazily at read time).

The main difference between **ACID** and **BASE** compliance is that ACID ensures that at the point in time of the transaction, ACID compliance is respected. BASE compliance allows for that compliance to be violated for a time as long as it eventually gets to a compliant end state.

We can think of ACID and BASE as a **continuum** instead of mutually exclusive choices:

ACID

- Strongconsistency
- Isolation
- Focuson"commit"
- Nestedtransactions
- Availability
- Conservative/pessimistic
- Difficulttoevolveaschema

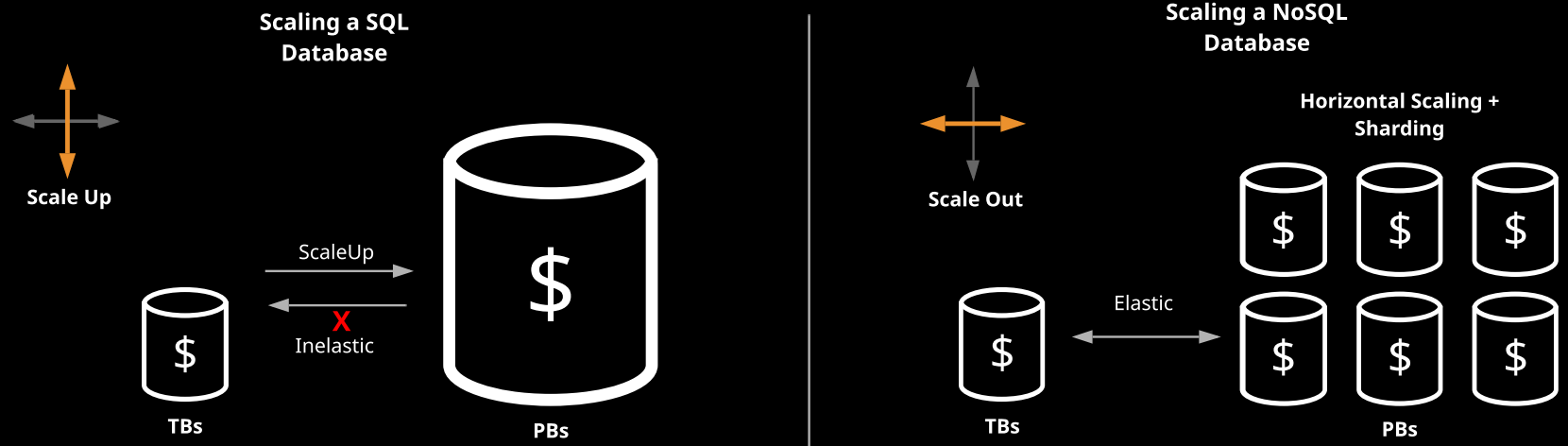
BASE

- Weakconsistency(staledata acceptable)
- Availabilityfirst
- Besteffort
- Approximateanswersare acceptable
- Aggressive/optimistic
- Fasterandeasierschema evolution

Continuum of choices ...

How Scaling in NoSQL Is Different

The biggest advantage of NoSQL databases is horizontal scaling or **sharding.** In a scale-out architecture, a distributed set of nodes known as a **cluster** is used as the basic architecture. It provides highly elastic scaling capability, enabling you to add nodes to handle load on the fly. This is the opposite of a scale-up architecture associated with **SQL databases**, which adds more resources to a single, larger machine.



Back

Linux Academy

A Little History

2004

Oracle configured with clustering and replication fails under load during the holiday season.

Analysis reveals 70% of **DB operations** were key-value and 20% operated on a single table.

Amazon begins development on Dynamo, a purpose-built, highly available key-value data store.

2007

The Amazon engineers behind **Dynamo** publish Dynamo: Amazon's Highly Available Key-value Store (a.k.a., the "Dynamo Paper"), summarizing their learnings.

Dynamo is poorly received inside Amazon; engineers continue to use **S3 and SimpleDB services**.

Amazon redesigns Dynamo for ease of management, performance, high availability, security, and scalability.

2012

Amazon DynamoDB **launches**.

SimpleDB is **effectively deprecated**.

Next

Back to Main

What is DynamoDB?

Here's the marketing pitch...

DynamoDB is a non-relational database for applications that need performance at any scale.

Amazon DynamoDB

- NoSQL managed database service
- Supports both key-value and document data models
- It's really fast
  - Consistent responsiveness
  - Single-digit millisecond
- Unlimited throughput and storage
- Automatic scaling up or down
- Handles **trillions** of requests per day
- ACID transaction support
- On-demand backups and point-in-time recovery
- Encryption at rest
- Data is replicated across multiple Availability Zones
- Service-level agreement (SLA) up to **99.999%**

Back

Next

Back to Main

Where Does DynamoDB Fit In?

Amazon Relational Database Service (RDS)
Support for Amazon Aurora, PostgreSQL, MySQL, MariaDB, Oracle Database, and SQL Server

Amazon DynamoDB
Key-value and document database

Amazon ElastiCache
Managed, Redis- or Memcached-compatible in-memory data store

Amazon Neptune
Graph database for applications that work with highly connected data sets

Amazon Redshift
Petabyte-scale data warehouse service

Amazon QLDB
Ledger database providing a cryptographically verifiable transaction log

Amazon DocumentDB
MongoDB-compatible database service

Back

Back to Main

Example Application Scenario

Pinehead Records is the world's largest retailer of vinyl records. Since establishing an online presence, high traffic and scaling issues have crippled the website on several occasions.

The current platform is Python, Flask, and MySQL.

The website must be migrated off its legacy relational database back end and onto a stable, high-performance platform that can scale with minimal operational overhead.

Application Migration Plan

Version 0: Legacy Relational Model

- Relational model in MySQL
- Limited database optimizations
- Limited application-level caching
- No indexes
- Inefficient queries
- Images (album cover art) stored on local filesystem
- User accounts in database

Version 1: Fundamental DynamoDB

- Naive migration from MySQL to DynamoDB
- Three DynamoDB tables mimicking the relational structure
- Images are moved to S3 with URI in a DynamoDB attribute
- No indexes
- User accounts in DynamoDB

Next

Back to Main

Version 2: Intermediate DynamoDB

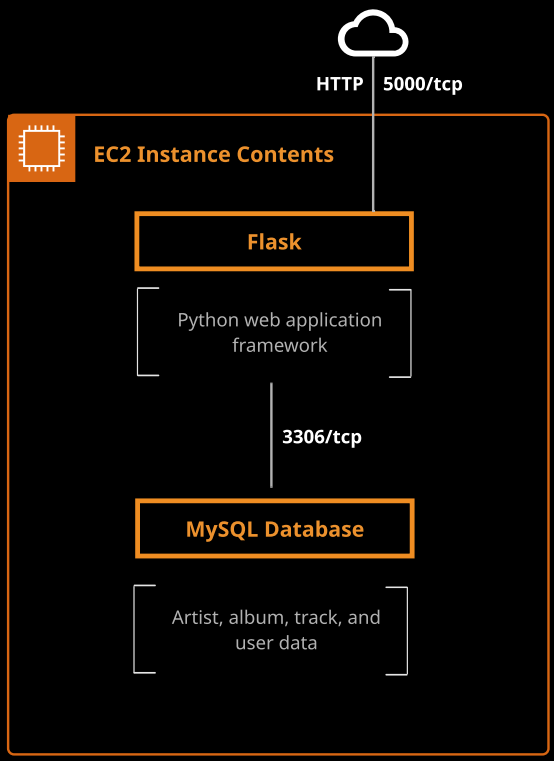
- Some optimizations
- Better table structure (single hierarchical table)
- Indexes
- Transactions
- User accounts in DynamoDB

Version 3: Advanced DynamoDB

- Federated web identity (Cognito)
- Fine-grained policies
- Lambda triggers
- Improved security
- DynamoDB Accelerator (DAX)

Back

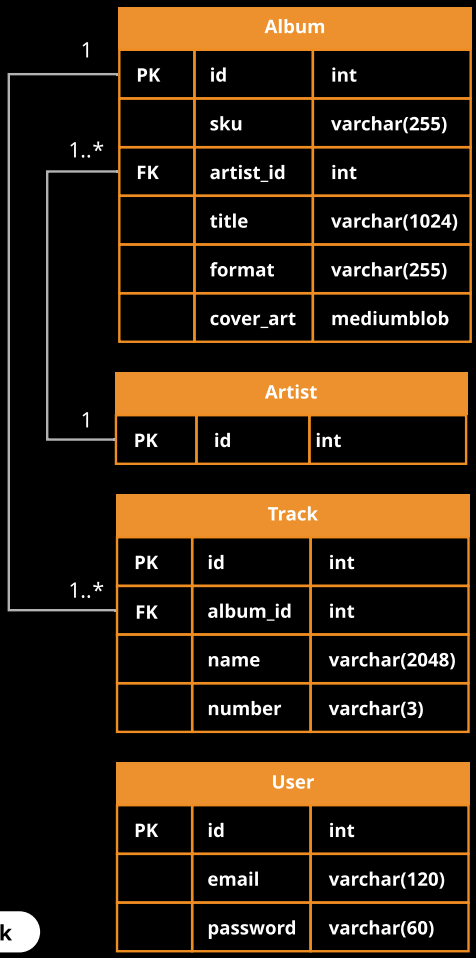
Demo Application Architecture
Version 0



Next

DynamoDB and Scenario Introduction

Scenario Discussion and Architecture Thoughts



Back

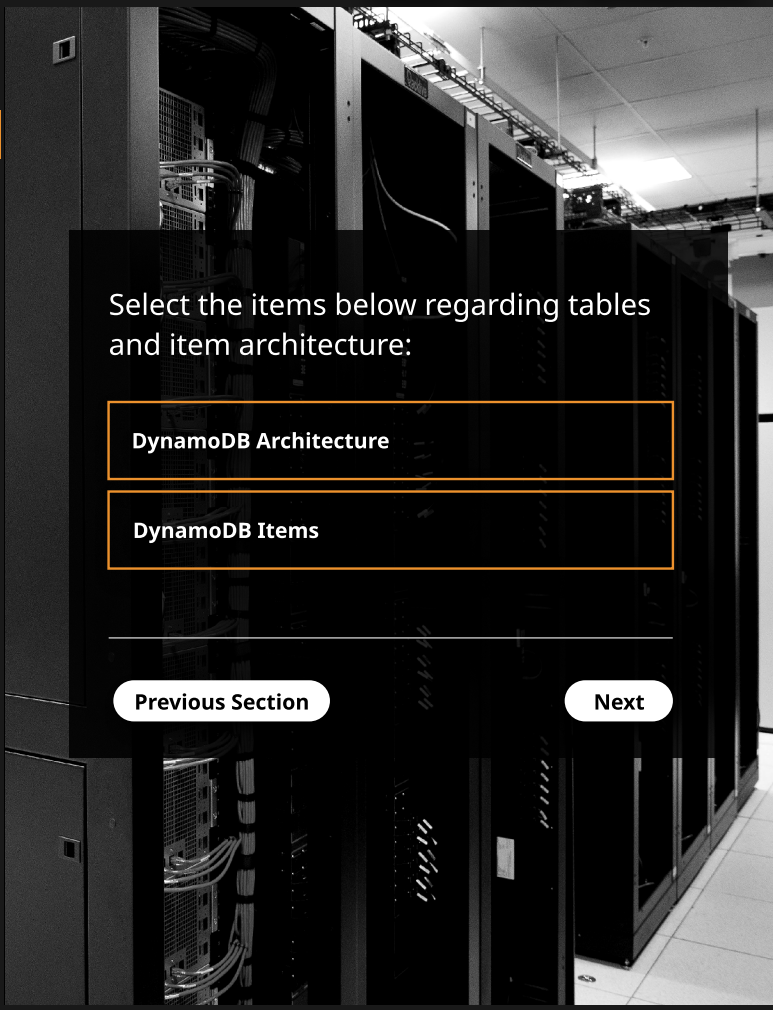
Next

Back to Main

Why DynamoDB?

1. Learning something new is hard. I already know SQL.
2. DynamoDB has no query language.
3. I have to choose between SQL and DynamoDB.
4. We already have a site license for *<commercial database>*.
5. It's just as hard to maintain my code with DynamoDB as it is with SQL, so what's the point?
6. My database isn't that big, isn't DynamoDB for big databases?
7. I enjoy writing SQL.
8. Why do I suddenly need a new database technology? Is this a gimmick?

Back



Select the items below regarding tables and item architecture:

- **DynamoDB Architecture**
- **DynamoDB Items**

Previous Section

Next

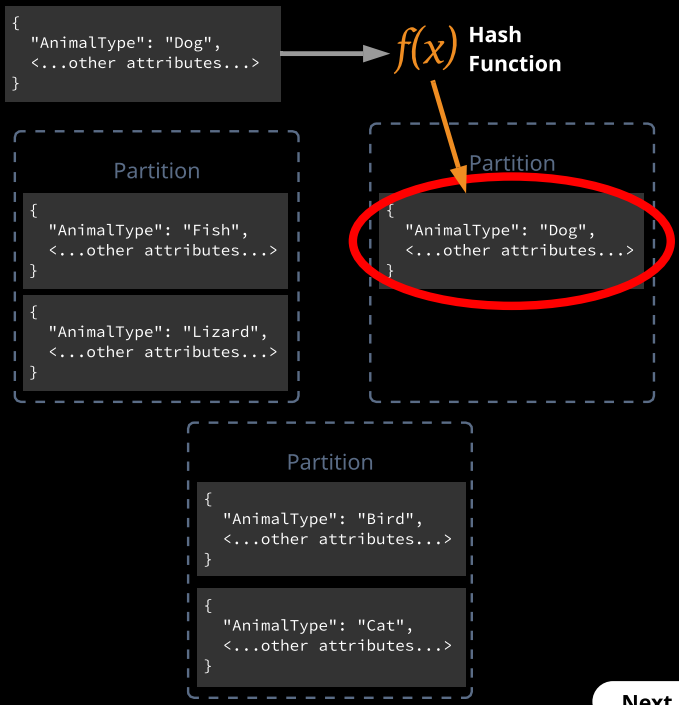
Back to Main

Section 5

Partitions and Data Distribution

DynamoDB stores data in partitions. A partition is an allocation of storage for a table, backed by solid-state drives (SSDs) and automatically replicated across multiple Availability Zones within an AWS Region.

To get the most out of DynamoDB throughput, create tables where the partition key has a large number of distinct values. Applications should request values fairly uniformly and as randomly as possible.



Next

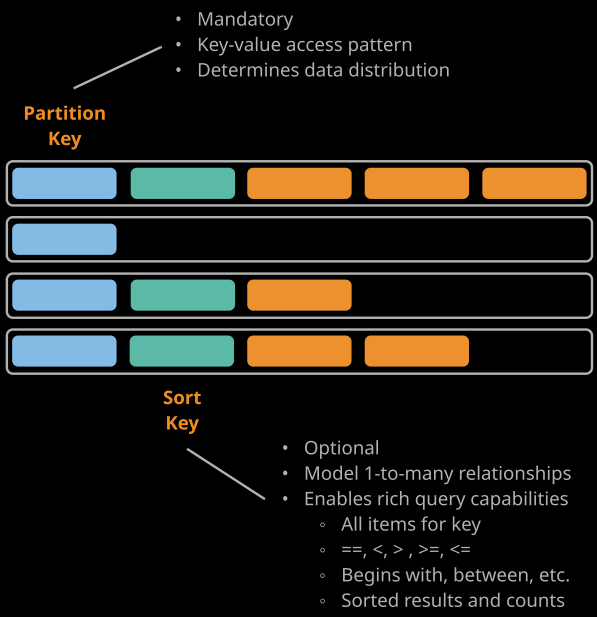
DynamoDB Fundamentals

DynamoDB Architecture

Table: Collection of data. DynamoDB tables must contain a name, primary key, and the required read and write throughput values. **Unlimited** size.

Partition Key: A simple primary key, composed of one attribute known as the **partition key**. This is also called the **hash attribute**.

Partition and Sort Key: Also known as a **composite primary key**, this type of key comprises two attributes. The first attribute is the **partition key**, and the second attribute is the **sort key**. This is also called the **range attribute**.



DynamoDB Performance

On-Demand Capacity

- Database scales according to demand
- Good option for:
  - New tables with unknown workloads
  - Applications with unpredictable traffic
  - Prefer to pay as you go

Provisioned Capacity

- Allows us to have **consistent and predictable** performance
- Specify expected read and write throughput requirements
- Read Capacity Units (**RCU**)
- Write Capacity Units (**WCU**)
- Price is determined by provisioned capacity
- Cheaper per request than On-Demand mode
- **Good option for**:
  - Applications with predictable traffic
  - Applications whose traffic is consistent or ramps gradually
  - Capacity requirements can be forecasted, helping to control costs

Both capacity modes have a limit of **40,000 RCUs and 40,000 WCUs**.

You can switch between modes only once per **24 hours**.

Table Addressing via ARN

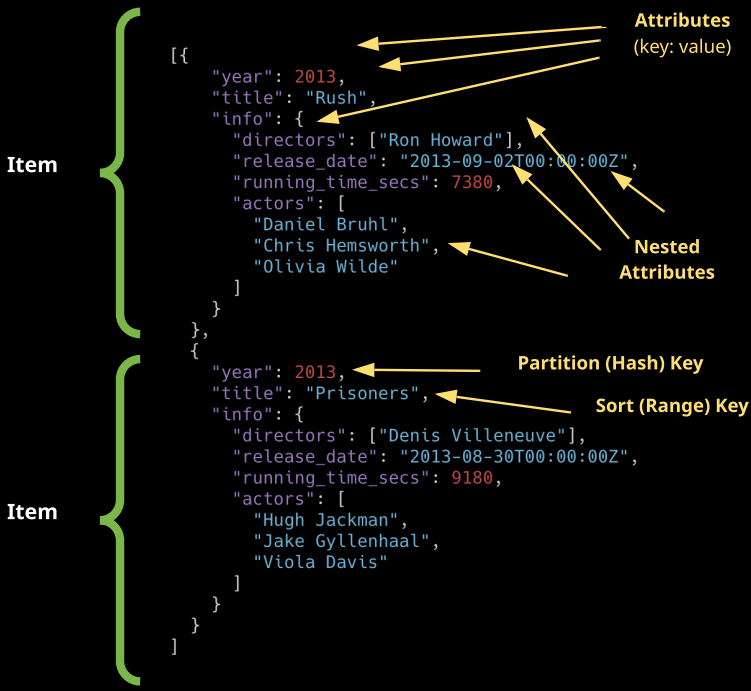
```
arn:aws:dynamodb:us-east-1:123456789012:table/album
```

Region **Account** **Table**

Back

Item: A table may contain multiple items. An item is a unique group of attributes. Items are similar to rows or records in a traditional relational database. Items are limited to 400 KB.

Attribute: Fundamental data element. Similar to fields or columns in an RDBMS.



Next

Data Types

Scalar: Exactly one value — number, string, binary, boolean, and null. Applications must encode binary values in base64-encoded format before sending them to DynamoDB.

Document: Complex structure with nested attributes (e.g.. JSON) — list and map.

Document Types

List: Ordered collection of values

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

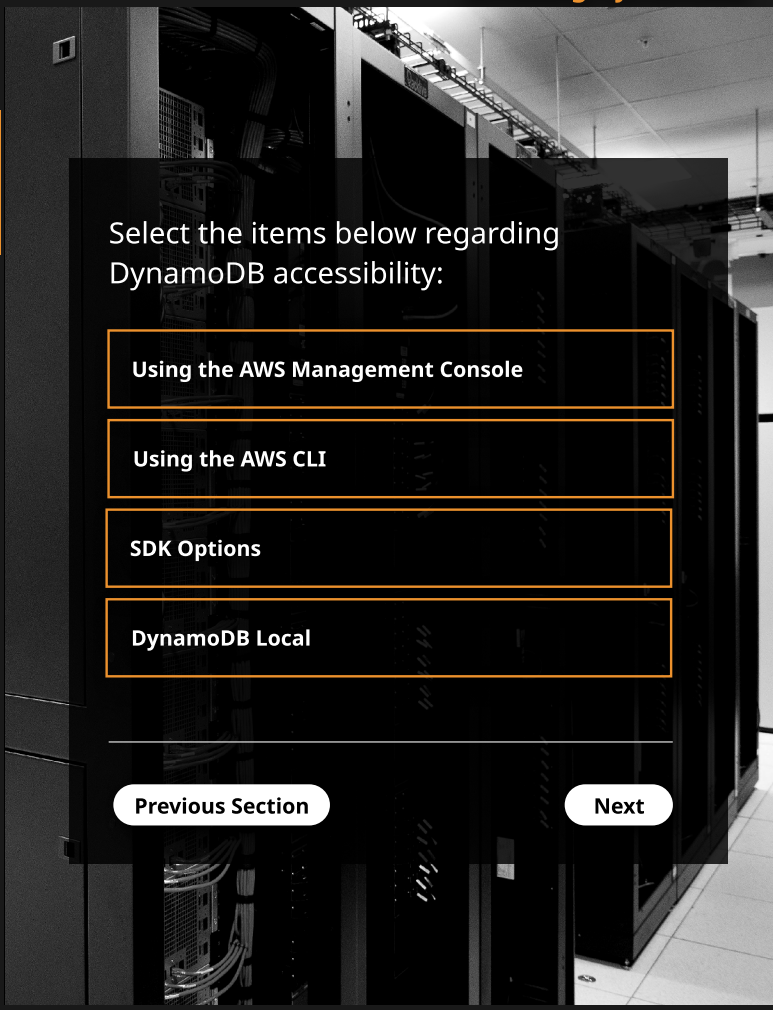
Map: Unordered collection of name-value pairs (similar to JSON)

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

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="]
```

Back



Select the items below regarding DynamoDB accessibility:

- [x] Using the AWS Management Console
- [ ] Using the AWS CLI
- [ ] SDK Options
- [ ] DynamoDB Local

Previous Section

Next

Creating a Table

- **Table names** must be unique per AWS account and region.
  - Between 3 and 255 characters long
  - UTF-8 encoded
  - Case-sensitive
  - Contain a–z, A–Z, 0–9, _ (underscore), - (dash), and . (dot)
- **Primary key** must consist of a partition key or a partition key and sort key.
  - Only string, binary, and number data types are allowed for partition or sort keys
- **Provisioned capacity mode** is the default (free tier).
  - For provisioned capacity mode, read/write throughput settings are required
- **Secondary indexes** creates a local secondary index.
  - Must be created at the time of table creation
  - Same partition key as the table, but a different sort key
- **Provisioned capacity** is set at the table level.
  - Adjust at any time or enable auto scaling to modify them automatically
  - On-demand mode has a default upper limit of 40,000 RCU/WCU — unlike auto scaling, which can be capped manually

Next

Back to Main

Section 5

DynamoDB Console Menu Items

- Dashboard
- Tables
  - Storage size and item count are **not** real-time (updated ~6h)
  - **Items:** Manage items and perform queries and scans.
  - **Metrics:** Monitor CloudWatch metrics.
  - **Alarms:** Manage CloudWatch alarms.
  - **Capacity:** Modify a table's provisioned capacity.
    - Free tier allows 25 RCU, 25 WCU, and 25 GB for 12 months
    - Linux Academy **Cloud Sandbox** within the Cloud Playground
  - **Indexes:** Manage global secondary indexes.
  - **Global Tables:** Multi-region, multi-master replicas
  - **Backups:** On-demand backups and point-in-time recovery
  - **Triggers:** Manage triggers to connect DynamoDB streams to Lambda functions.
  - **Access control:** Set up fine-grained access control with web identity federation.
  - **Tags:** Apply tags to your resources to help organize and identify them.
- Backups
- Reserved capacity
- Preferences
- DynamoDB Accelerator (DAX)

DynamoDB Fundamentals

Using the AWS CLI

Installing the AWS CLI

- Preinstalled on Amazon Linux and Amazon Linux 2
- Linux Academy **Cloud Sandbox** within the Cloud Playground

Obtaining IAM Credentials

- Option 1: Create IAM access keys in your own AWS account.
- Option 2: Use Cloud Sandbox credentials.
- Note the access key ID and secret access key.

Configuring the AWS CLI

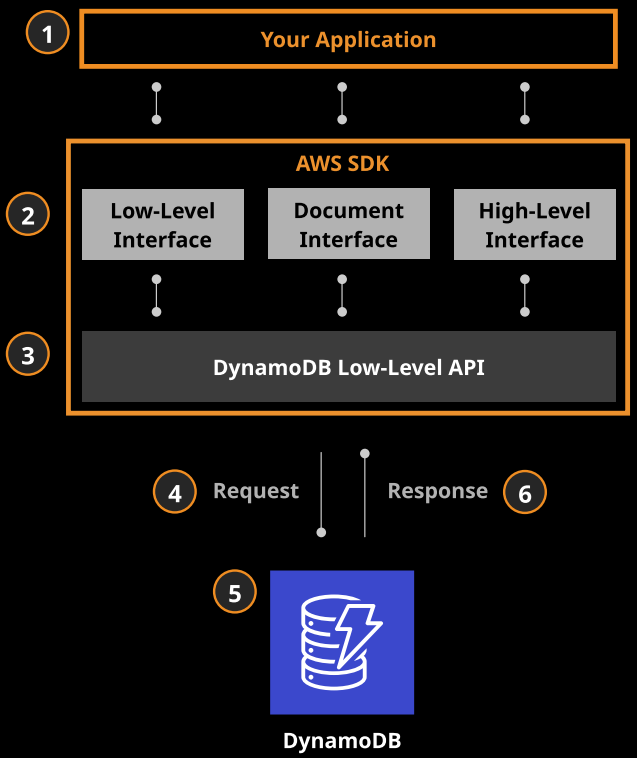
- `aws configure`
- `aws sts get-caller-identity`
- `aws dynamodb help`

Using DynamoDB with the AWS CLI

- `aws dynamodb create-table`
- `aws dynamodb describe-table`
- `aws dynamodb put-item`
- `aws dynamodb scan`

DynamoDB Fundamentals

SDK Options



Next

Data Type Descriptors

S – String	**M** – Map
N – Number	**L** – List
B – Binary	**SS** – String Set
BOOL – Boolean	**NS** – Number Set
NULL – Null	**BS** – Binary Set

Low-Level API

Request

```
POST / HTTP/1.1
Host: dynamodb.us-east-1.amazonaws.com
Accept-Encoding: identity
Content-Length: 158
User-Agent: aws-cli/1.16.225
Content-Type: application/x-amz-json-1.0
Authorization: AWS4-HMAC-SHA256 Credential=...
X-Amz-Date: 20190916T141902Z
X-Amz-Target: DynamoDB_20120810.GetItem

{
    "TableName": "Music",
    "Key": {
        "Artist": {"S": "Queen"},
        "Year": {"N": "1989"}
    }
}
```

Data Type Descriptors

Response

```
HTTP/1.1 200 OK
x-amzn-RequestId: 6B5D43RPJ1PH5CS15RJUV66DFJVV4KQNSO5AEMVJF66Q
x-amz-crc32: 3651145634
Content-Type: application/x-amz-json-1.0
Content-Length: 188
Date: 20190916T141903Z

{
    "Item": {
        "Artist": {"S": "Queen"},
        "Year": {"N": "1989"},
        "AlbumTitle": {"S": "The Miracle"},
        "TrackCount": {"N": "13"},
        "Barcode": {"S", "0077779235728"}
    }
}
```

Document Interface

- Perform data plane operations (create, read, update, delete) on tables and indexes
- No need for data type descriptors
- Available in the AWS SDKs for Java, .NET, Node.js, and JavaScript in the browser

```
var docClient = new AWS.DynamoDB.DocumentClient()
var table = 'Music'
var artist = 'Queen'
var year = 1989

var params = {
  TableName: table,
  Key: {
    artist: artist,
    year: year
  }
}

docClient.get(params, function(err, data) {
  if (err) {
    console.log('Unable to read item: ' + JSON.stringify(err))
  } else {
    console.log('GetItem succeeded: ' + JSON.stringify(data))
  }
})
```

Back
Next

Back to Main
Section 5

Object Persistence Interface

- Do not directly perform data plane operations
- Map complex data types to items in a DynamoDB table
- Create objects that represent tables and indexes
- Define the relationships between objects in your program and the tables that store those objects
- Call simple object methods, such as `save`, `load`, or `delete`
- Available in the AWS SDKs for Java and .NET

Back

- Downloadable version of DynamoDB
- Test applications locally without accessing the DynamoDB web service
- Docker image or Java `.jar` file

Java

Download DynamoDB local from links in the documentation

```
java -Djava.library.path=./DynamoDBLocal_lib \
     -jar DynamoDBLocal.jar
```

Docker

All DynamoDB local dependencies and necessary configuration built in

```
docker run -d -p 8000:8000 amazon/dynamodb-local
```

Using the Local Endpoint

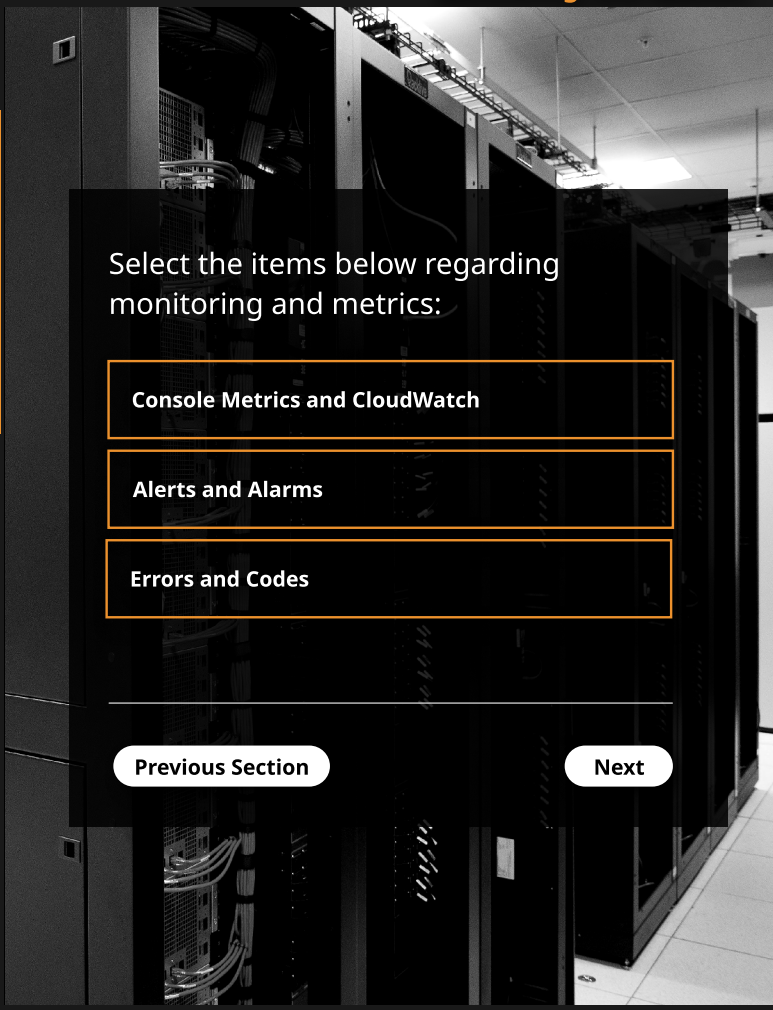
AWS CLI

```
aws dynamodb list-tables \
    --endpoint-url http://localhost:8000
```

Python (Boto3)

```
import boto3

client = boto3.client('dynamodb',
            endpoint_url='http://localhost:8000/')
client.list_tables()
```



Select the items below regarding monitoring and metrics:

- [x] Console Metrics and CloudWatch
- [x] Alerts and Alarms
- [x] Errors and Codes

Previous Section

Next

Back to Main

Section 5

DynamoDB Fundamentals

Console Metrics and CloudWatch

CloudWatch monitors your AWS resources in real time, providing visibility into resource utilization, application performance, and operational health.

- Track metrics (data points over time)
- Create dashboards
- Create alarms
- Create rules for events
- View logs

Helps answer questions like:

- How can I determine how much of my provisioned throughput is being used?
- How can I determine which requests exceed the provisioned throughput limits of a table?
- How can I determine if any system errors occurred?

DynamoDB Metrics

- ConsumedReadCapacityUnits
- ConsumedWriteCapacityUnits
- ProvisionedReadCapacityUnits
- ProvisionedWriteCapacityUnits
- ReadThrottleEvents
- SuccessfulRequestLatency
- SystemErrors
- ThrottledRequests
- UserErrors
- WriteThrottleEvents

Alarms can be created on metrics, taking an action if the alarm is triggered.

Alarms have three states:

- **INSUFFICIENT:** Not enough data to judge the state — alarms often start in this state.
- **ALARM:** The alarm threshold has been breached (e.g., > 90% CPU).
- **OK:** The threshold has not been breached.

Alarms have a number of key components:

- **Metric:** The data points over time being measured
- **Threshold:** Exceeding this is bad (static or anomaly)
- **Period:** How long the threshold should be bad before an alarm is generated
- **Action:** What to do when an alarm triggers
  - SNS
  - Auto Scaling
  - EC2

Back

DynamoDB Fundamentals

Errors and Codes

Successful DynamoDB requests return the HTTP status code for success (`200 OK`), along with results from the requested operation.

If the request fails, DynamoDB returns an error with three components:

- HTTP status code (e.g., `400`)
- Exception name (e.g., `ResourceNotFoundException`)
- Error message (e.g., `Requested resource not found: Table: tablename not found`)

Most AWS SDKs propagate these low-level errors for you. Handle using `try-catch` blocks or `if-then` statements.

Error Retries and Exponential Backoff

AWS SDKs all implement retry logic automatically.

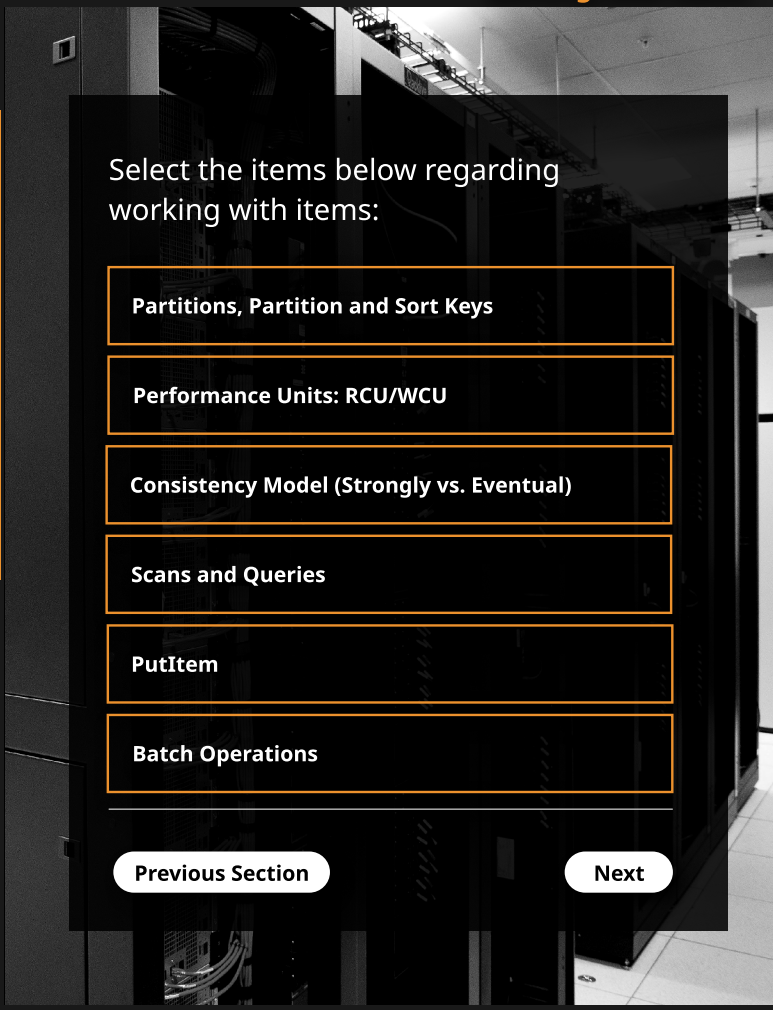
Exponential backoff uses increasing wait times for consecutive error messages. This algorithm helps mitigate request rate errors.

Batch Operations

- Batch operations tolerate the failure of individual requests.
- The most common cause of errors is **throttling.**
- Unprocessed items should be retried using exponential backoff.

AccessDeniedException
ConditionalCheckFailedException
IncompleteSignatureException
ItemCollectionSizeLimitExceededException
LimitExceededException
MissingAuthenticationTokenException
ProvisionedThroughputExceededException
RequestLimitExceeded
ResourceInUseException
ResourceNotFoundException
ThrottlingException
UnrecognizedClientException
ValidationException

InternalServerError
ServiceUnavailable



Select the items below regarding working with items:

- Partitions, Partition and Sort Keys
- Performance Units: RCU/WCU
- Consistency Model (Strongly vs. Eventual)
- Scans and Queries
- PutItem
- Batch Operations

Previous Section
Next

Back to Main
Section 5

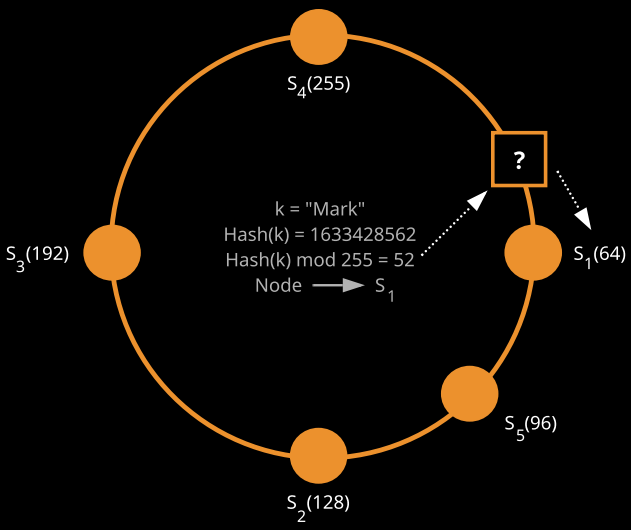
DynamoDB Fundamentals

Partitions, Partition and Sort Keys

- The partition key or partition and sort key are used to uniquely **write** or **read** an item.
- Partition key is used for partition selection via the DynamoDB internal hash function
- One partition holds **10 GB** of data and supports up to **3,000** read capacity units (RCU) or **1,000** write capacity units (WCU).
- Partition key is used to select the item(s) to read/write.
- Sort key is used to range select (e.g., `begins_with`) or to order results.
- Sort keys may not be used on their own.

Consistent Hashing

Key: k
N storage nodes
Selected node: Hash(k) mod N



DynamoDB Fundamentals

Performance Units: RCU/WCU

Provisioned Throughput

Maximum amount of capacity an application can consume from a table or index. Throttled requests: `ProvisionedThroughputExceededException`

Eventually vs. Strongly Consistent Read

Eventually consistent reads might include stale data.

Strongly consistent reads are always up to date but are subject to network delays.

Read Capacity Units (RCUs)

One RCU represents one strongly consistent read request per second, or two eventually consistent read requests, for an item up to 4 KB in size. Transactional read requests require 2 RCUs for items up to 4 KB.

Filtered query or scan results consume full read capacity.

For an **8 KB** item size:

- 2 RCUs for one strongly consistent read
- 1 RCU for an eventually consistent read
- 4 RCUs for a transactional read

Write vs. Transactional Write

Writes are eventually consistent within one second or less.

Write Capacity Units (WCUs)

One WCU represents one write per second for an item up to 1 KB in size. Transactional write requests require 2 WCUs for items up to 1 KB.

For a **3 KB** item size:

- Standard: 3 WCUs
- Transactional: 6 WCUs

Provisioned Throughput Calculations

Provisioned Throughput

Maximum amount of capacity an application can consume from a table or index.

Provisioned Throughput Calculations

A system needs to store 60 patient records of 1.5 KB, each, every minute. What WCU should you allocate on the patient record table?

- 60 records per minute = ~1 per second (and the DDB RCU/WCU buffer can smooth this out if not)
- Each record is 1.5 KB. 1 WCU = 1 KB per second, so each record requires 2 WCU.
- A WCU setting of 2 is required on the table.

A weather application reads data from a DynamoDB table. Each item in the table is 7 KB in size. How many RCUs should be set on the table to allow for 10 reads per second?

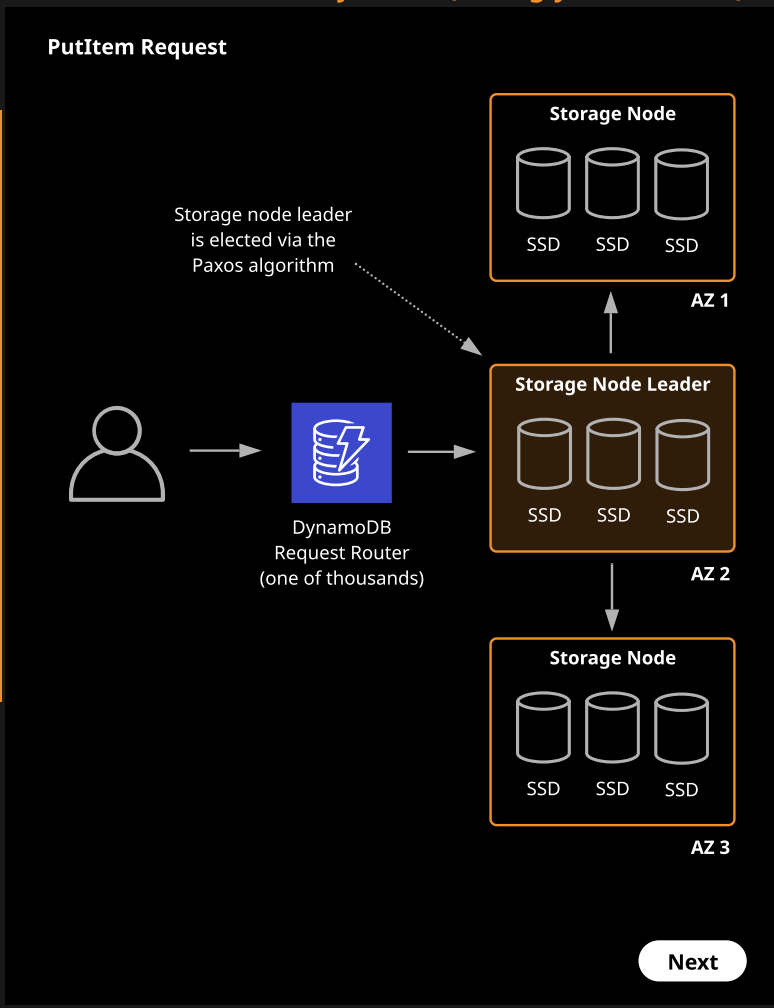
- 1 item is 7 KB, which is 2 RCU (1 RCU is 4 KB).
- 10 reads per second for 7 KB items = 20 RCU
- But the question didn't specify if eventual or strong consistency is required. The default is eventual, which allows for 2 reads of 4 KB per second for 1 RCU.
- Assuming eventually consistent reads, the answer is 10 RCU.

- Standard: 3 WCUs
- Transactional: 6 WCUs

Provisioned Throughput Calculations

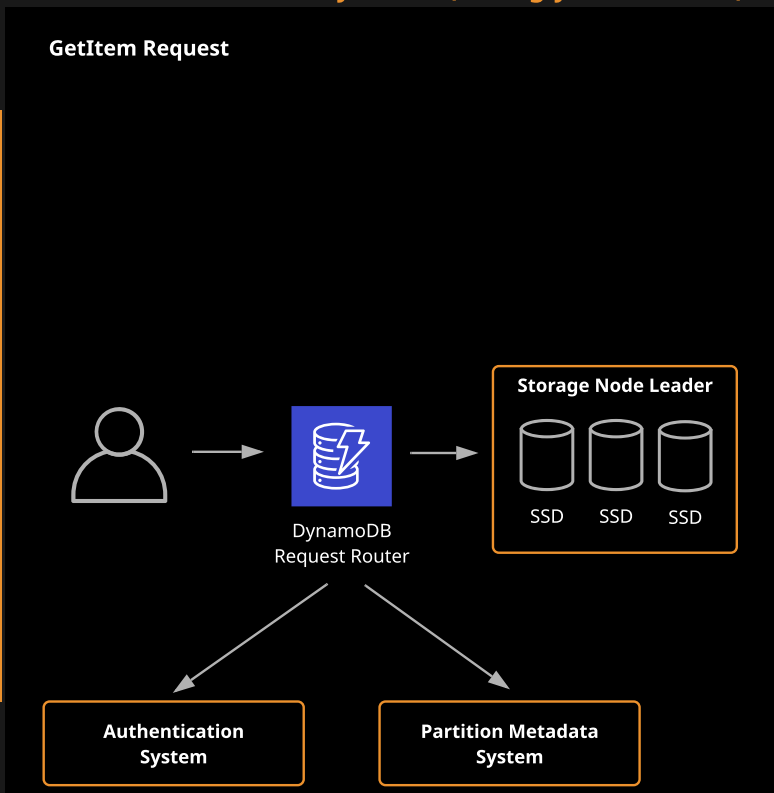
DynamoDB Fundamentals

Consistency Model (Strongly vs. Eventual)



DynamoDB Fundamentals

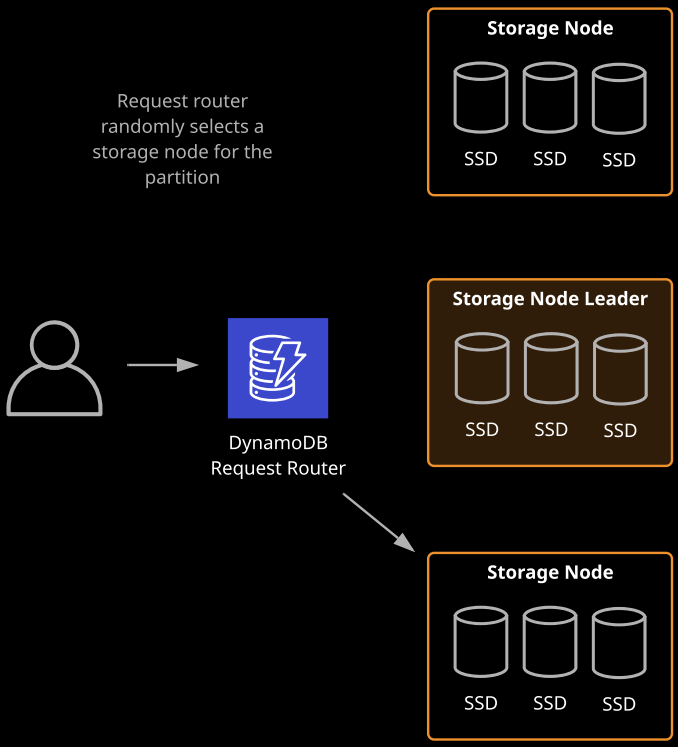
Consistency Model (Strongly vs. Eventual)



Back

Next

Eventual Consistency - GetItem



- 1 in 3 chance that you've requested stale data (worst case).
- Eventually consistent reads are 50% the "cost" of strongly consistent.
- Strongly consistent reads require 2 of 3 matches across replicas.

Back

Scan

- Returns all items and attributes for a given table
- Filtering results do not reduce RCU consumption; they simply discard data
- Eventually consistent by default, but the `ConsistentRead` parameter can enable strongly consistent scans
- Limit the number of items returned
- A single query returns results that fit within 1 MB
- Pagination can be used to retrieve more than 1 MB
- Parallel scans can be used to improve performance
- Prefer query over scan when possible; occasional real-world use is okay
- If you are repeatedly using scans to filter on the same non-PK/SK attribute, consider creating a secondary index

Query

- Find items based on primary key values
- Query limited to PK, PK+SK, or secondary indexes
- Requires PK attribute
- Returns all items with that PK value
- Optional SK attribute and comparison operator to refine results
- Filtering results do not reduce RCU consumption; they simply discard data
- Eventually consistent by default, but the `ConsistentRead` parameter can enable strongly consistent queries
- Querying a partition only scans that one partition
- Limit the number of items returned
- A single query returns results that fit within 1 MB
- Pagination can be used to retrieve more than 1 MB

PutItem

- Creates a new item
- Replaces existing item with the same key

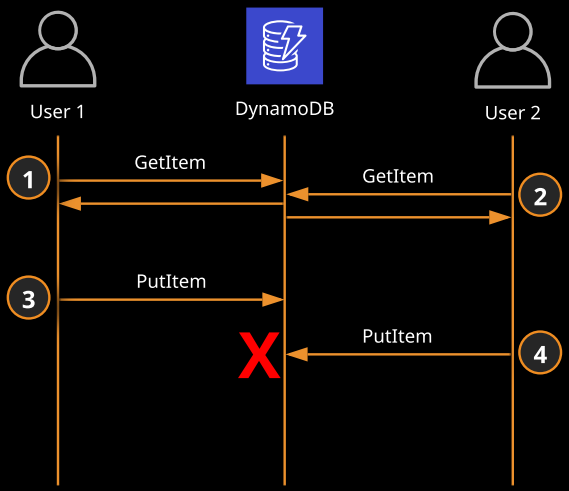
UpdateItem

- Creates a new item if the specified key does not exist; otherwise, it modifies an existing item's attributes

DeleteItem

- Deletes the item with the specified key

Write Conflicts

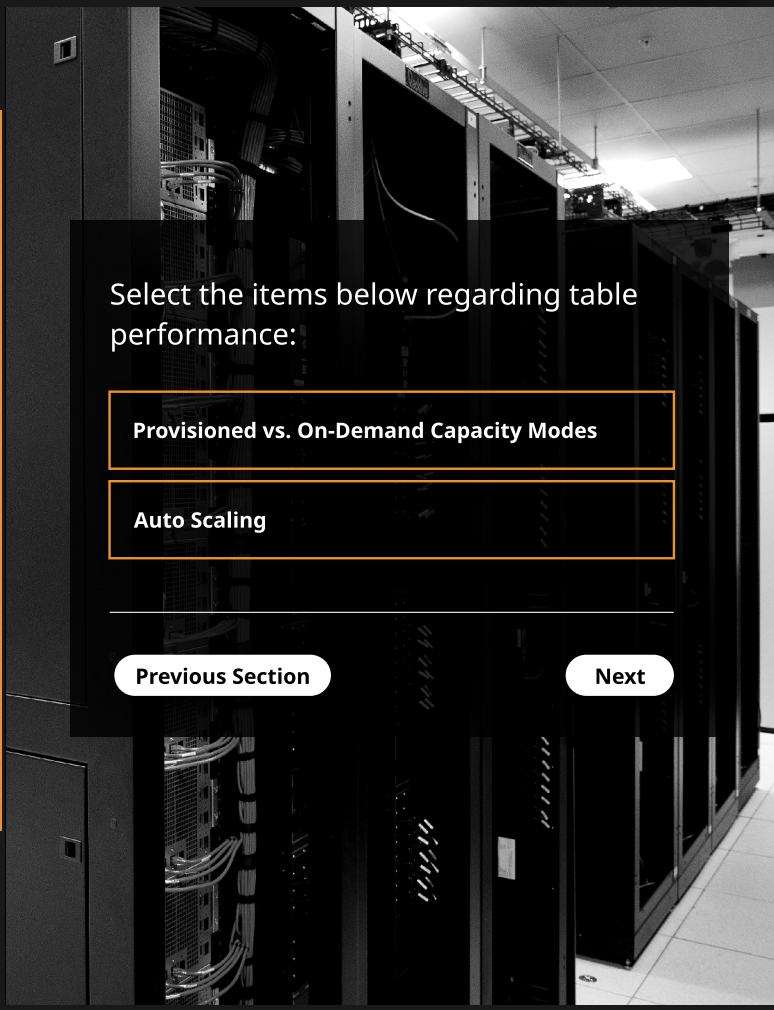


BatchGetItem

- Returns attributes for multiple items from multiple tables
- Request using primary key
- Returns up to 16 MB of data, up to 100 items
- Get unprocessed items exceeding limits via `UnprocessedKeys`
- Eventually consistent by default, but the `ConsistentRead` parameter can enable strongly consistent scans
- Retrieves items in parallel to minimize latency

BatchWriteItem

- Puts or deletes multiple items in multiple tables
- Writes up to 16 MB of data, up to 25 put or delete requests
- Get unprocessed items exceeding limits via `UnprocessedItems`
- Conditions are not supported for performance reasons
- Threading may be used to write items in parallel



Select the items below regarding table performance:

- Provisioned vs. On-Demand Capacity Modes
- Auto Scaling

Previous Section

Next

Back to Main

Section 5

DynamoDB Fundamentals

Provisioned vs. On-Demand Capacity Modes

Provisioned Capacity

- Minimum capacity required
- Able to set a budget (maximum capacity)
- Subject to throttling
- Auto scaling available
- Risk of underprovisioning — 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 (first 25 GB is free)
- Example:
  - 25 WCU, 25 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

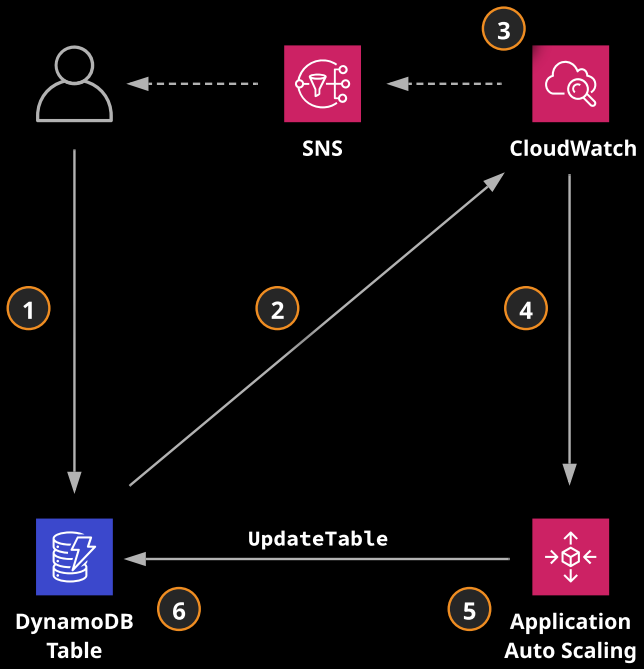
On-Demand Capacity

- 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 Fundamentals

Auto Scaling

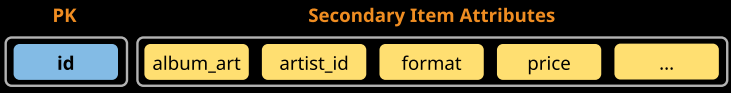
- Auto scaling is best suited for predictable, gradually changing traffic patterns.
- You may still manually change provisioned throughput settings.
- Recommended to apply auto scaling uniformly to tables and global secondary indexes (GSIs).



DynamoDB Fundamentals

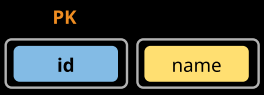
Version 1 Data Model Overview

Table Name: album



Partition Key: id (N)
Secondary Attributes: album_art (S), artist_id (N), format (S), price (N), title (S), year (N)

Table Name: artist

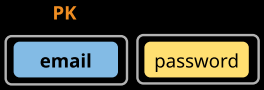


Partition Key: id (N)
Secondary Attributes: name (S)

Table Name: track

Partition Key: id (N)
Secondary Attributes: album_id (N), name (S), number (N), length (N)

Table Name: user



Partition Key: email (S)
Secondary Attributes: password (S)

Course Navigation

Intermediate DynamoDB
Section 5

Table Indexes

Indexes: Part 1
Indexes: Part 2
Importing Tables Using AWS Database Migration Service

Backup and Recovery

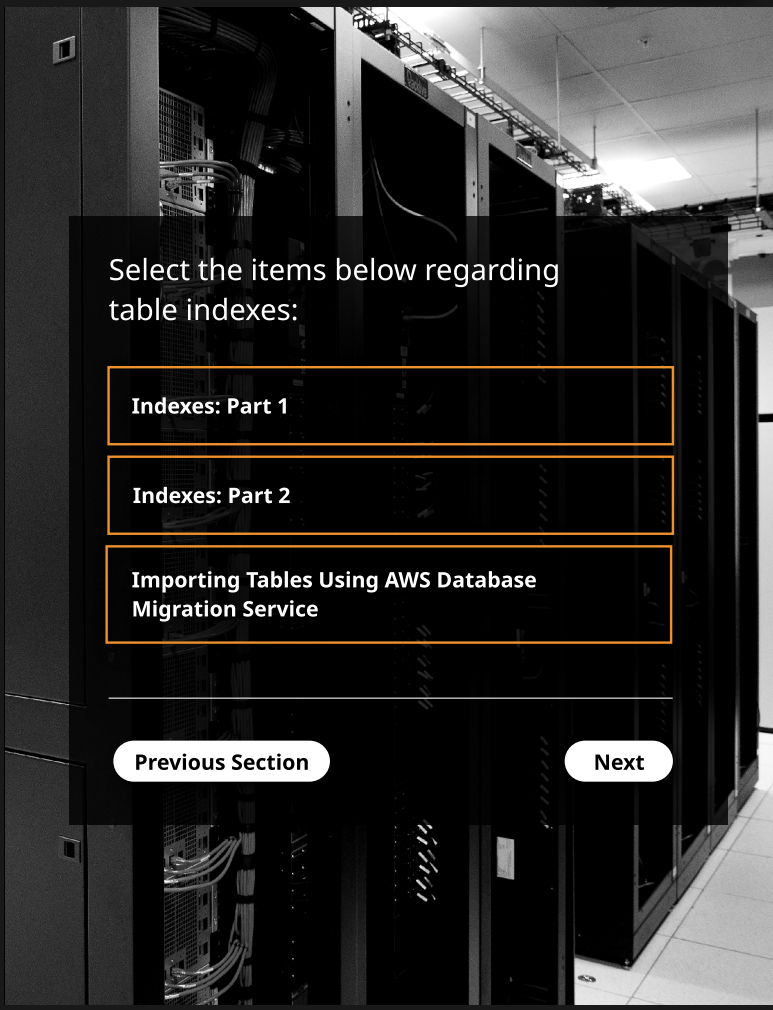
On-Demand vs. Continuous (PITR) Backup

Advanced Performance and Scaling Considerations

Offloading Large Attribute Values to S3
Hot and Cold Partition Imbalance

Data Consistency and Management

Conditional and Update Expressions
Transactions
Time to Live (TTL)



Select the items below regarding table indexes:

- **Indexes: Part 1**
- **Indexes: Part 2**
- **Importing Tables Using AWS Database Migration Service**

Previous Section

Next

Next Topics

Back to Main

Secondary Indexes

Data structure that contains 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: Index with a partition key and a sort key that can be different from those on the base table. The primary key of a GSI can be either simple (partition key) or composite (partition key and sort key). Strong consistency is unsupported.

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

Partition Key — Sort Key

username	email	order_id	total
mark	mark@linuxacademy.com	223	4096.64
terry	tcox@linuxacademy.com	224	1024.16

Order Table

Partition Key — Sort Key

username	email	order_id	total
mark	mark@linuxacademy.com	223	4096.64
terry	tcox@linuxacademy.com	224	1024.16

Global Secondary Index (GSI)

Partition Key — Sort Key

username	email	order_id	total
mark	mark@linuxacademy.com	223	4096.64
terry	tcox@linuxacademy.com	224	1024.16

Local Secondary Index (LSI)

Next Topics
Back to Main

Course Navigation

Intermediate DynamoDB
Section 5

Table Indexes
Indexes: Part 1
Indexes: Part 2
Importing Tables Using AWS Database Migration Service
Backup and Recovery
On-Demand vs. Continuous (PITR) Backup
Advanced Performance and Scaling Considerations
Offloading Large Attribute Values to S3
Hot and Cold Partition Imbalance
Data Consistency and Management
Conditional and Update Expressions
Transactions
Time to Live (TTL)

Index Queries and Projection Expressions

A projection is the set of attributes copied from a table into a secondary index. The partition key and sort key of the table are always projected into the index.

`ALL` — All of the table attributes are projected into the index.

`KEYS_ONLY` — Only the index and primary keys are projected into the index.

`INCLUDE` — Only the specified table attributes are projected into the index.

Using **`GetItem`**, **`Query`**, or **`Scan`** operations return all attributes by default. Use **Projection Expressions** to get a subset of these attributes.

Example: Find all albums by artist_id, returning only title and price:

```
response = table.query(
    IndexName="artist_id-index",
    KeyConditionExpression=Key("artist_id").eq(192),
    ProjectionExpression="title, price",
)
```

Response:

```
{"price": "30.94", "title": "The Game"}
{"price": "7.45", "title": "A Night at the Opera"}
{"price": "7.18", "title": "Crazy Little Thing Called Love"}
{"price": "7.22", "title": "The Invisible Man"}
{"price": "48.97", "title": "Don't Stop Me Now"}
{"price": "3.63", "title": "Keep Yourself Alive"}
{"price": "35.63", "title": "A Day at the Races"}
{"price": "35.31", "title": "Under Pressure"}
{"price": "11.73", "title": "I Want to Break Free"}
...
{"price": "22.6", "title": "Bohemian Rhapsody"}
```

Next

Next Topics

Back to Main

Sparse Indexes

DynamoDB writes a corresponding index item only if the index sort key value is present in the item. If the sort key doesn't appear in every table item, the index is said to be **sparse**.

In other words, the index is "sparse" because it does not include all items of the parent table.

Global secondary indexes are sparse by default. When you create a GSI, you specify a partition key and optionally a sort key. Only items in the parent table that contain those attributes appear in the index.

By creating sparse indexes, you can provision GSIs with lower write throughput than that of the parent table. This can possibly save a lot of money that would otherwise be spent on storage (for no additional benefits).

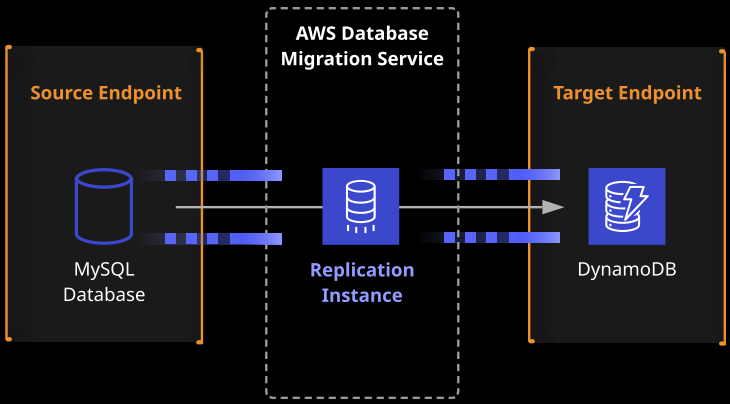
Back

Next Topics

Back to Main

Intermediate DynamoDB

Importing Tables Using AWS Database Migration Service



Configuring DMS

1. Create replication instance
2. Create MySQL source endpoint
3. Create service access role in IAM
4. Create DynamoDB target endpoint
5. Create database migration task

Next Topics

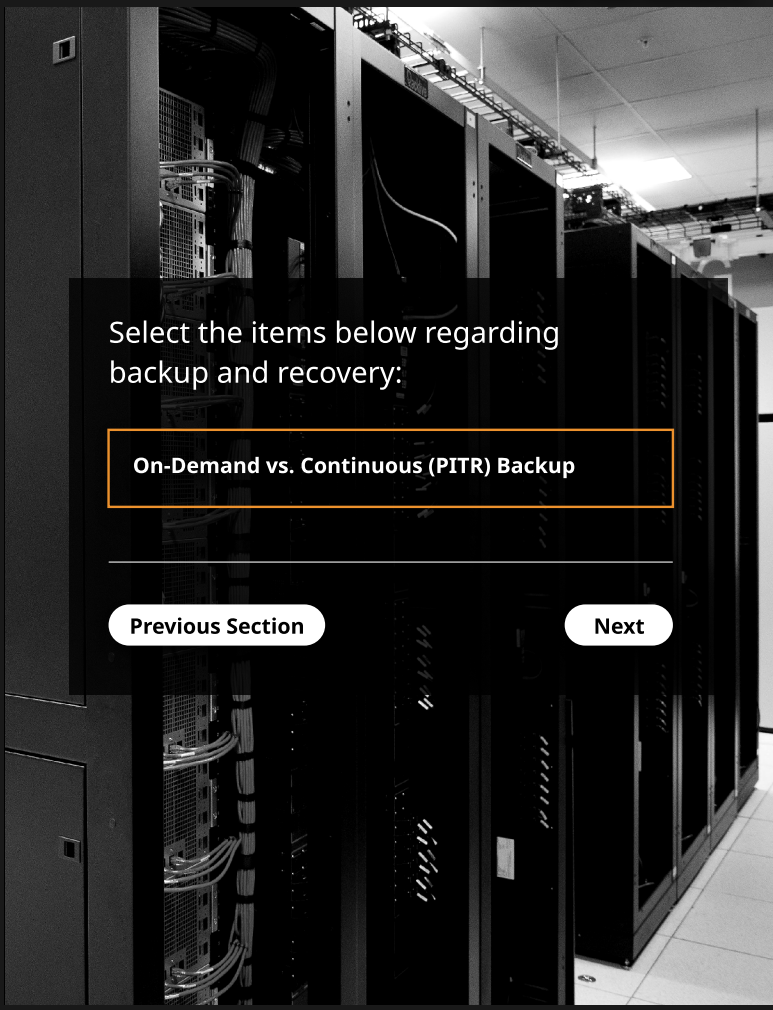
Back to Main

Course Navigation

Intermediate DynamoDB
Section 5

Table Indexes
Indexes: Part 1
Indexes: Part 2
Importing Tables Using AWS Database Migration Service
Backup and Recovery
On-Demand vs. Continuous (PITR) Backup
Advanced Performance and Scaling Considerations
Offloading Large Attribute Values to S3
Hot and Cold Partition Imbalance
Data Consistency and Management
Conditional and Update Expressions
Transactions
Time to Live (TTL)

Next Topics
Back to Main



Select the items below regarding backup and recovery:

On-Demand vs. Continuous (PITR) Backup

Previous Section
Next

On-Demand Backup and Restore

Create full backups of your tables at any time in either the AWS Management Console or with a single API call.

- Backup and restore actions execute with zero impact on table performance or availability.
- Backups are consistent within seconds and retained until deleted.
- Backup and restore operates within the same region as the source table.

Point-in-Time Recovery (PITR)

Helps protect your DynamoDB tables from accidental writes or deletes. You can restore your data to any point in time in the last **35 days**.

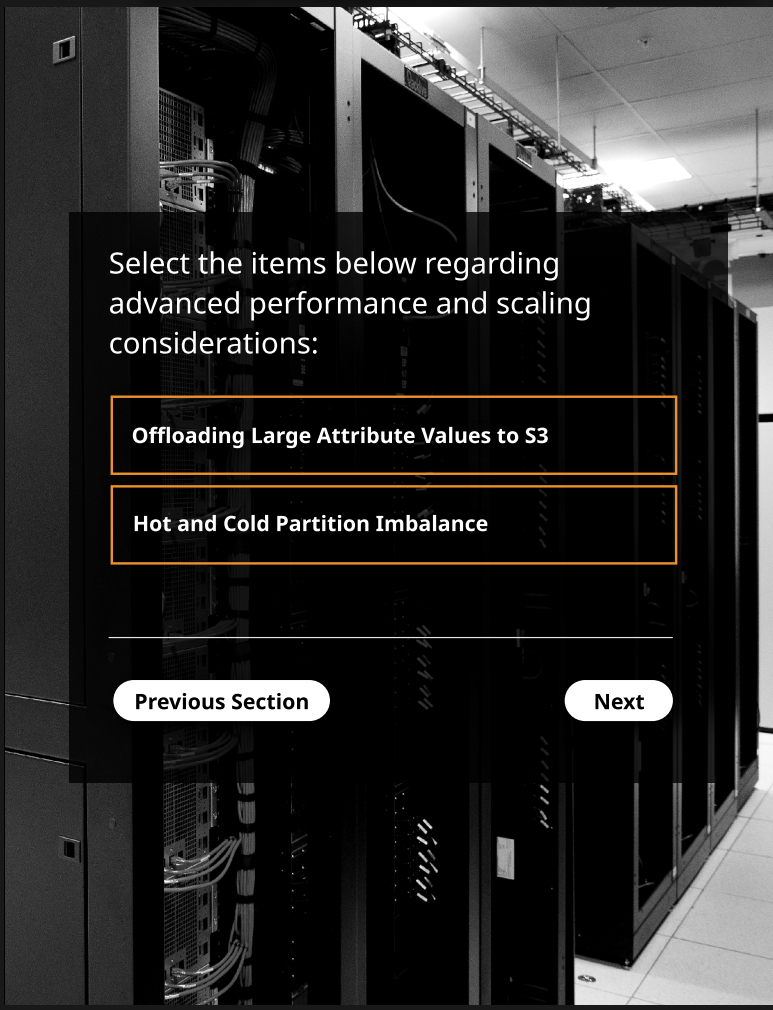
- DynamoDB maintains **incremental** backups of your data.
- Point-in-time recovery is not enabled by default.
- The latest restorable timestamp is typically five minutes in the past.

After restoring a table, you must manually set up the following on the restored table:

- Auto scaling policies
- AWS Identity and Access Management (IAM) policies
- Amazon CloudWatch metrics and alarms
- Tags
- Stream settings
- Time to Live (TTL) settings
- Point-in-time recovery settings

Intermediate DynamoDB

Advanced Performance and Scaling Considerations



Select the items below regarding advanced performance and scaling considerations:

- **Offloading Large Attribute Values to S3**
- **Hot and Cold Partition Imbalance**

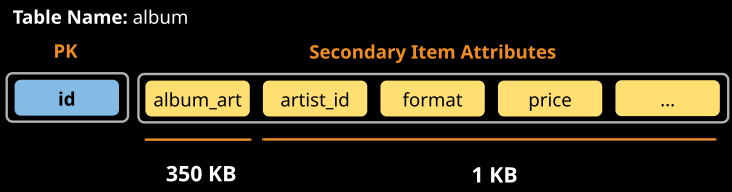
Previous Section

Next

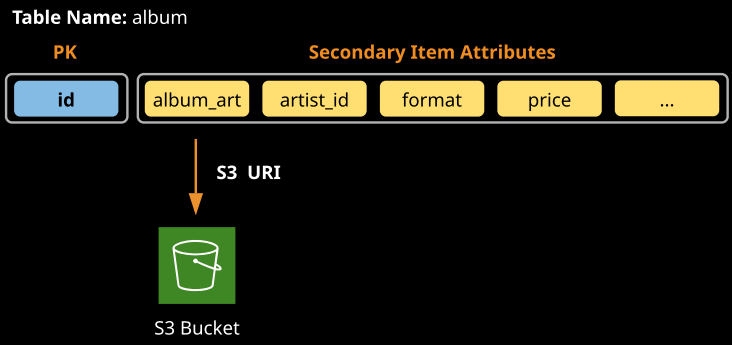
Next Topics

Back to Main

Large Attribute Values in Attributes



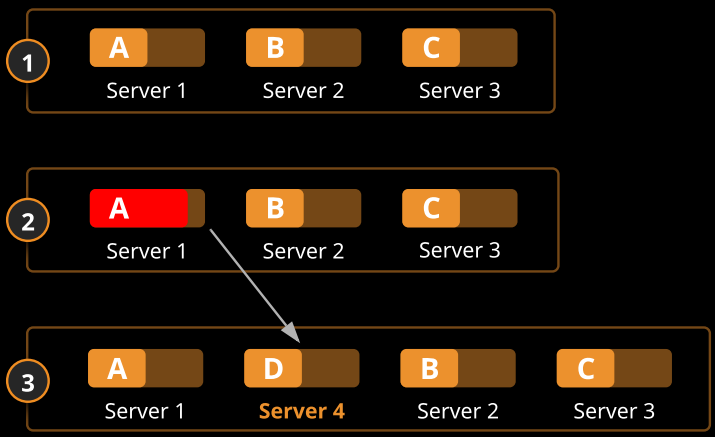
Offloading Large Attribute Values to S3



Intermediate DynamoDB

Hot and Cold Partition Imbalance

Resharding `(RCU/3,000) + (WCU/1,000) = # Partitions`



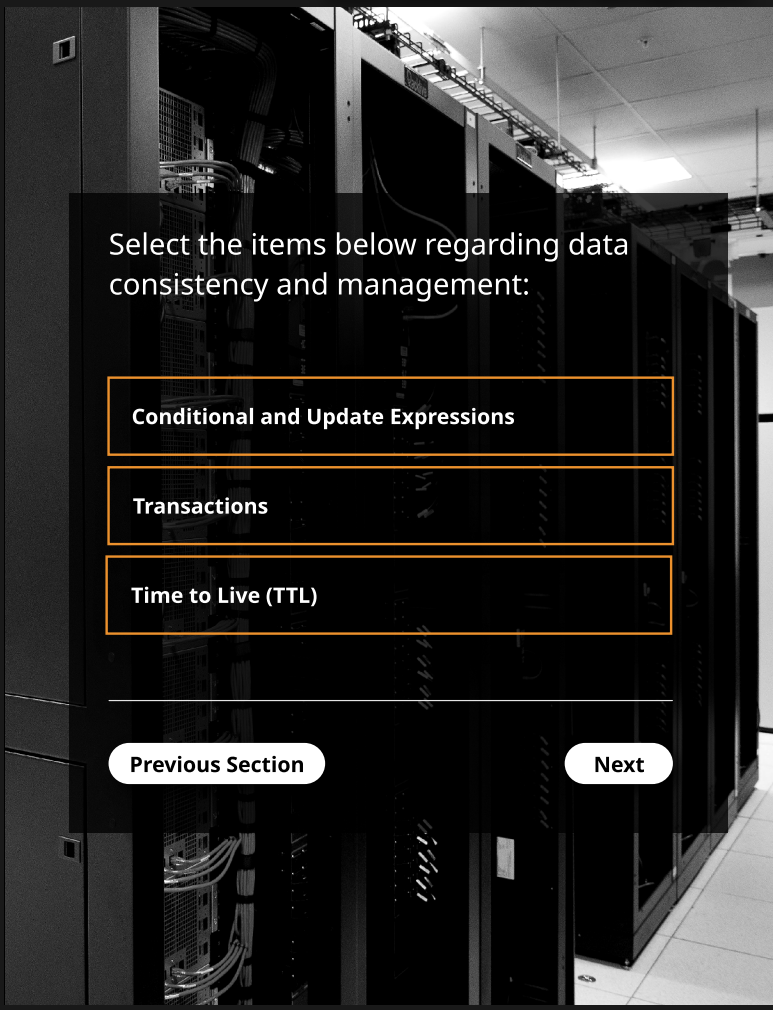
Adaptive Capacity

Example Table
*Total **provisioned** capacity = 400 WCU*
*Total **consumed** capacity = 300 WCU*

Next Topics
Back to Main

Intermediate DynamoDB

Data Consistency and Management



Select the items below regarding data consistency and management:

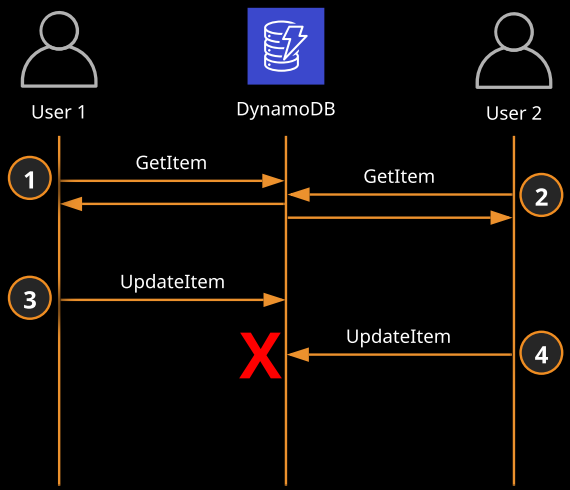
- [ ] Conditional and Update Expressions
- [ ] Transactions
- [ ] Time to Live (TTL)

Previous Section

Next

Next Topics

Back to Main



Conditional Updates

Performs a put/update/delete operation only if a condition is met:

- Check attribute value
- Check if attribute exists

Atomic counter can be used to increment or decrement the value of an existing attribute without interfering with other write requests.

Transactions enable developers to easily perform atomic writes and isolated reads across multiple items and tables.

`TransactWriteItems:`
- Transact up to 25 items or 4 MB
- Evaluate conditions
- If all conditions are simultaneously true, perform write operations

`TransactGetItems:`
- Transact up to 25 items or 4 MB
- Return a consistent, isolated snapshot of all items

DynamoDB performs **two underlying reads or writes** of every item in the transaction, one to prepare the transaction and one to commit the transaction.

Apply transactions to items in:
- Same partition key or across partition keys
- Same table or across tables
- Same region only
- Same account only
- DynamoDB tables only

Transactions scale like the rest of DynamoDB:
- Unlimited concurrent transactions
- Low latency, high availability
- Scales horizontally
- Up to you to design to avoid hot keys

Reasons for transaction failures:
- Precondition failure
- Insufficient capacity
- Transactional conflicts
- Transaction still in progress
- Service error
- Malformed request
- Permissions

Intermediate DynamoDB

Time to Live (TTL)

Time to Live (TTL) lets you define when table items expire so they can be automatically deleted from the database.

- No cost to use TTL
- Reduce storage costs by limiting storage usage to relevant items only
- Does not consume provisioned throughput

Use cases:

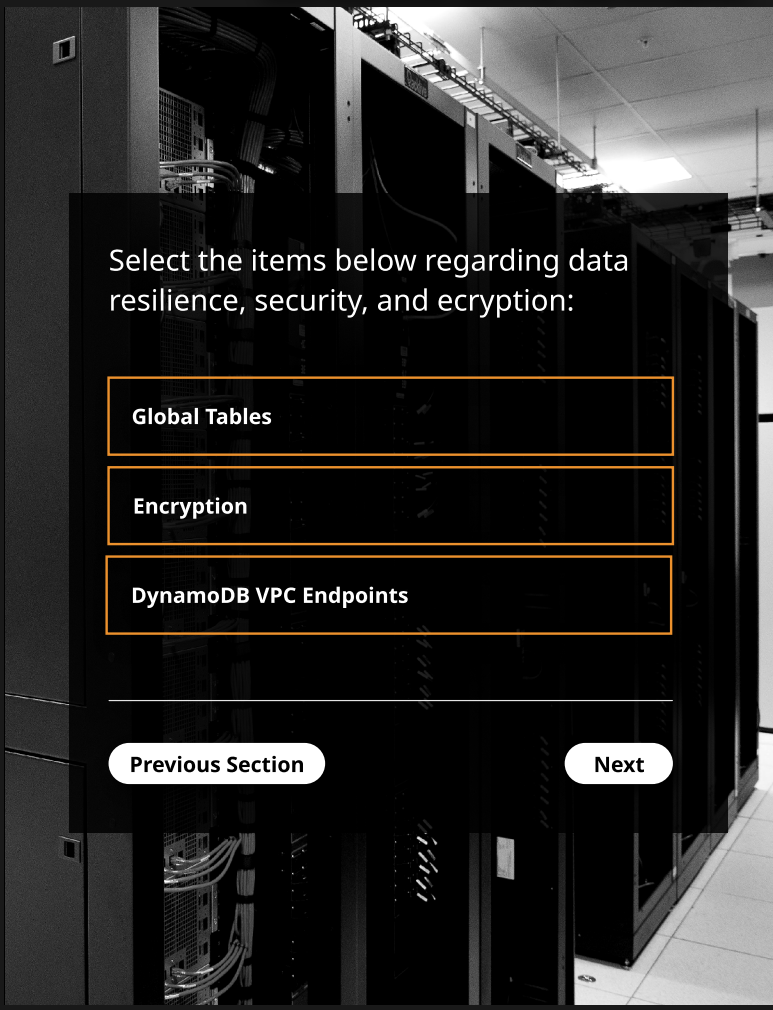
- Session data
- Event logs
- Pattern searching and usage tracking
- Debugging and analytics
- Other temporary sensitive data for contractual or regulatory compliance

How It Works

1. TTL compares the current time to the defined TTL attribute of an item.
2. If current time > item's TTL value, then the item is marked for deletion.
3. DynamoDB typically deletes expired items within **48 hours** of expiration.
4. Items are removed from LSIs and GSIs automatically using an eventually consistent delete operation.

> The **epoch** time format is the number of seconds elapsed since 12:00:00 AM January 1, 1970 UTC.

Due to the potential delay between expiration and deletion time, you might get expired items when you query for items. Use a **filter expression** to return only items where the item's TTL expiration > current time.



Select the items below regarding data resilience, security, and ecryption:

- **Global Tables**
- **Encryption**
- **DynamoDB VPC Endpoints**

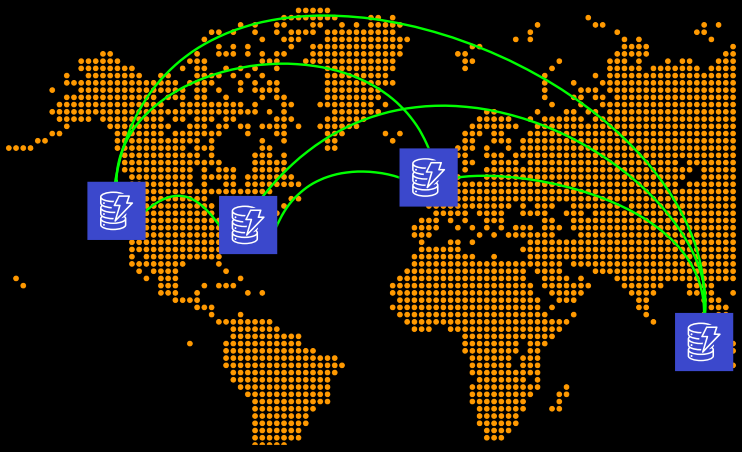
Previous Section **Next**

Previous Topics

Back to Main

Global tables are a fully managed, multi-master, multi-region replication solution.

- Build high performance, globally distributed applications
- Low latency reads and writes to locally available tables
- Multi-region redundancy for disaster recovery (DR) or high availability (HA) applications
- Easy to set up and no application rewrites required
- Replication latency typically under one second



Intermediate DynamoDB

Encryption

DynamoDB protects data stored **at rest** and **in transit** between on-premises clients and DynamoDB, and between DynamoDB and other AWS resources within the same AWS Region.

Encryption in transit uses HTTPS.

Encryption at rest uses AWS Key Management Service (KMS) to encrypt all table data, including:

- Primary key
- LSIs and GSIs
- Streams
- Global tables
- Backups
- DAX clusters

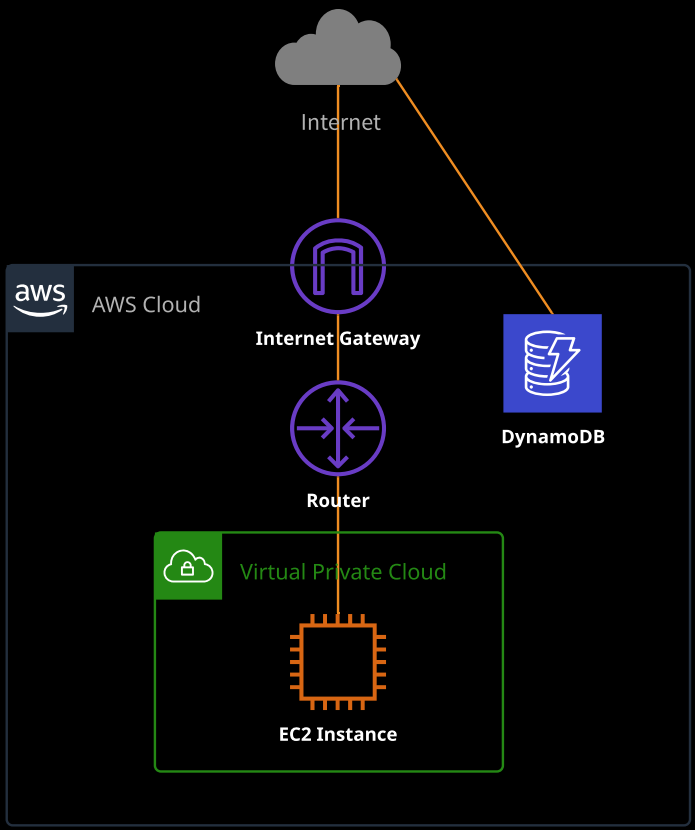
When creating a new table, you may choose:

- AWS owned customer master key (CMK)
  - Default option; key owned by DynamoDB; no charges
- AWS managed CMK
  - Key stored in AWS account; managed by AWS KMS
  - View the CMK and its key policy (policy can't be changed)
  - Audit encryption/decryption of your table using CloudTrail

DynamoDB Encryption Client

- Java and Python libraries that encrypt your data **prior** to sending to DynamoDB
- Use cryptographic materials from any source, including your own cryptography service

Without VPC Endpoints



Next

Previous Topics

Back to Main

Course Navigation

Intermediate DynamoDB
Section 5 (Continued)

Data Resilience, Security, and Encryption

Global Tables

Encryption

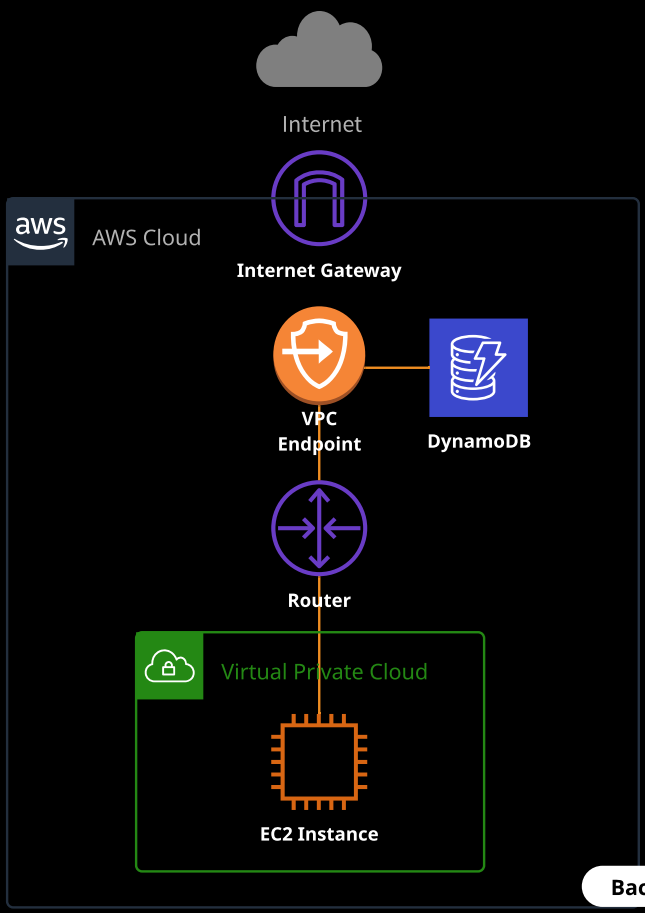
DynamoDB VPC Endpoints

Database Model v2

Data Model v2 Review

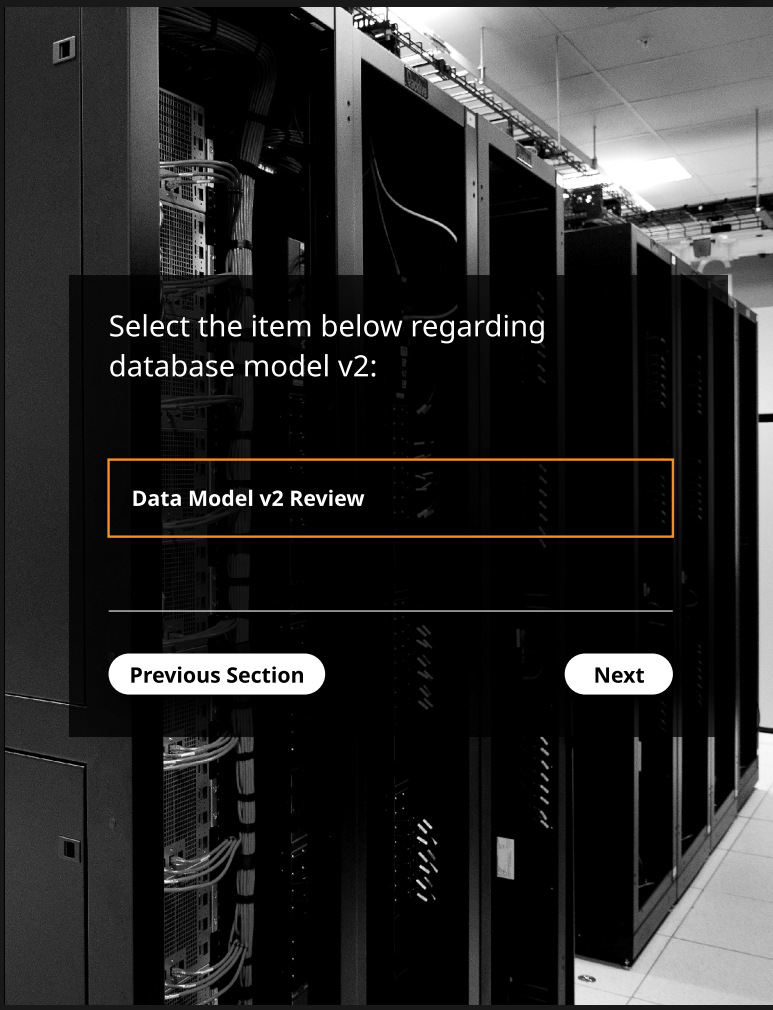
Advanced DynamoDB
Section 6

With VPC Endpoints



Previous Topics

Back to Main



Select the item below regarding database model v2:

Data Model v2 Review

Previous Section

Next

Previous Topics

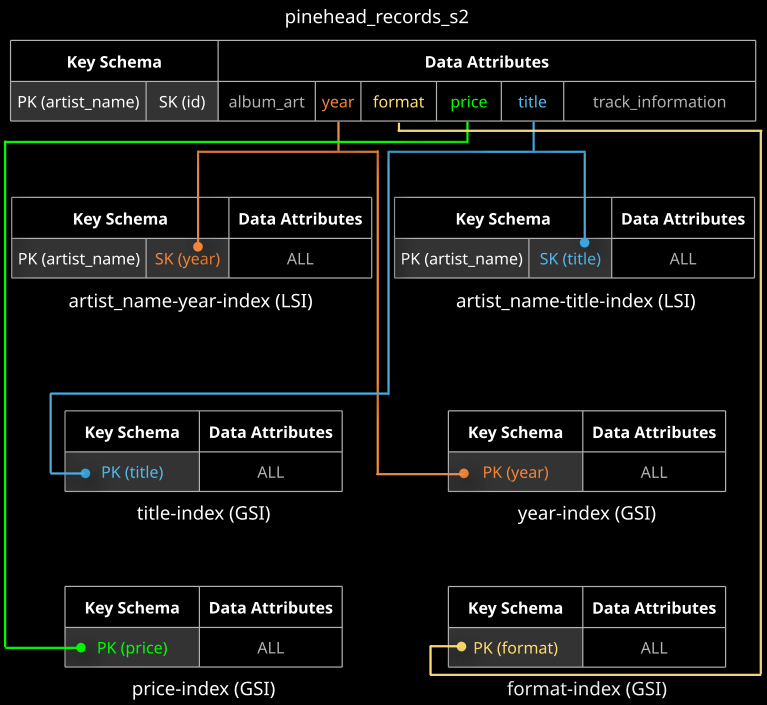
Back to Main

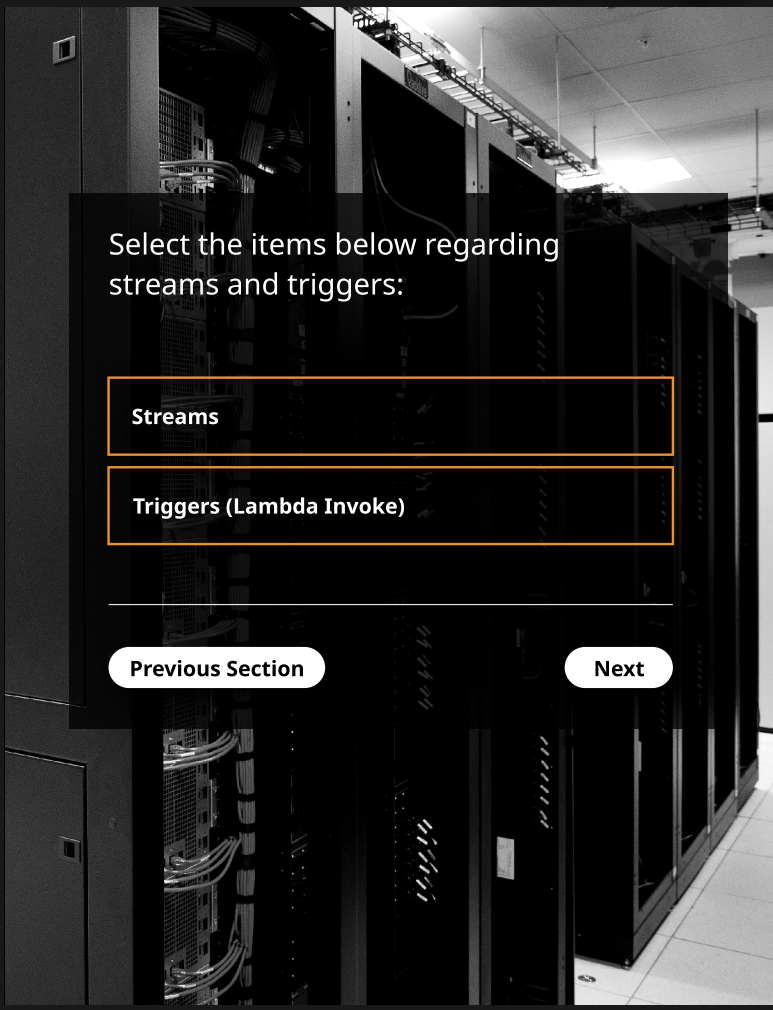
Intermediate DynamoDB

Data Model v2 Review

This version of the Pinehead Records data model features:

- Better table structure (single hierarchical table)
- Local and global secondary indexes
- User accounts in DynamoDB





Select the items below regarding streams and triggers:

- **Streams**
- **Triggers (Lambda Invoke)**

Previous Section **Next**

Back to Main

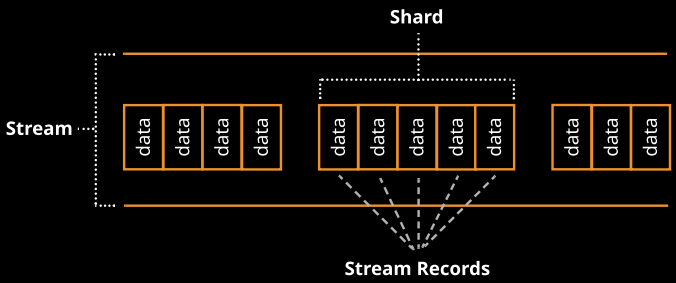
Course Navigation

Advanced DynamoDB
Section 6

Streams and Triggers
Streams
Triggers (Lambda Invoke)
Advanced Index Usage
Selective Write Sharding
Aggregation with Streams
Implementing High-Performance Architectures
Static Data Dumps (streams to static file)
DynamoDB Accelerator (DAX) Architecture
SQS Write Buffer
Advanced Security
Federated and Fine-Grained Access
Auditing Admin Access Using CloudTrail
Database Model v3
Database Model v3 Review
Integration
Amazon EMR
Amazon Elasticsearch Service

What Are DynamoDB Streams?

- Time-ordered sequence of item-level changes in a table
- Partitioned change log (shards) allows scaling
- Information stored for 24 hours
- Provides a stream of inserts, updates, and deletes
- Stream records appear in the same sequence as the item modifications
- Guaranteed to be delivered **only once**
- Use Kinesis Client Library (KCL), Lambda, or the API to query changes
  - `KEYS_ONLY` — Only the **key attributes** of the modified item
  - `NEW_IMAGE` — The entire item, as it appears **after** it was modified
  - `OLD_IMAGE` — The entire item, as it appeared **before** it was modified
  - `NEW_AND_OLD_IMAGES` — Both the **new and the old images** of the item
- Each `GetRecords` API call consumes 1 **streams read request unit** (not RCU) and returns up to 1 MB of data
- The first 2.5 M streams read request units are free; $0.02 per 100k reads thereafter



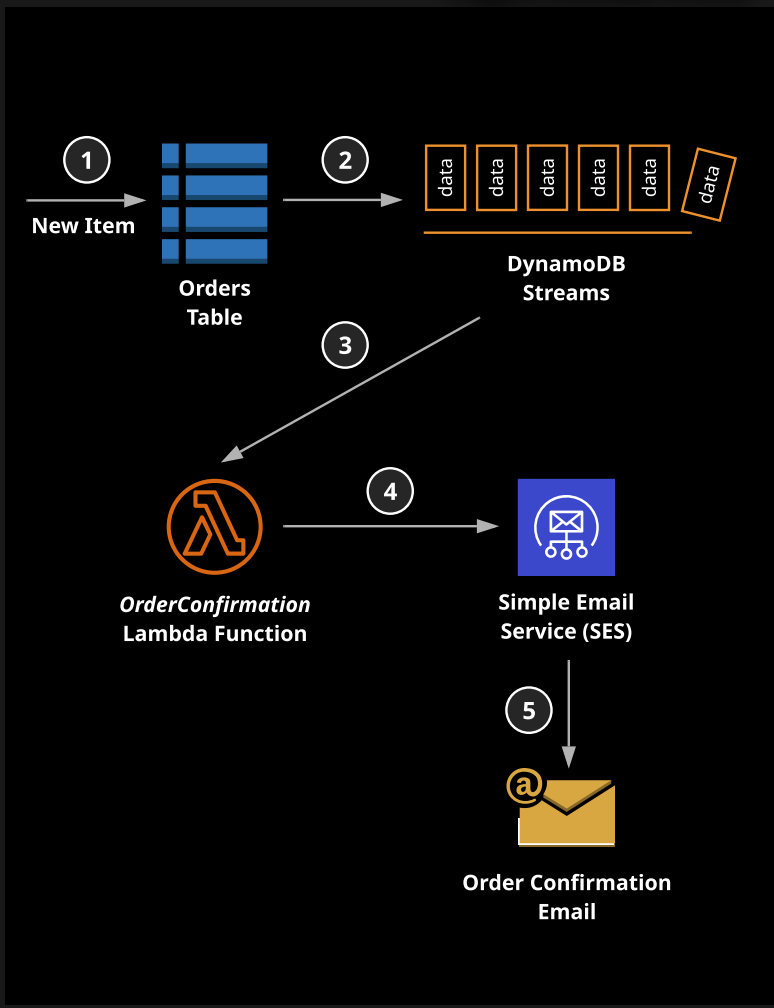
Use Cases

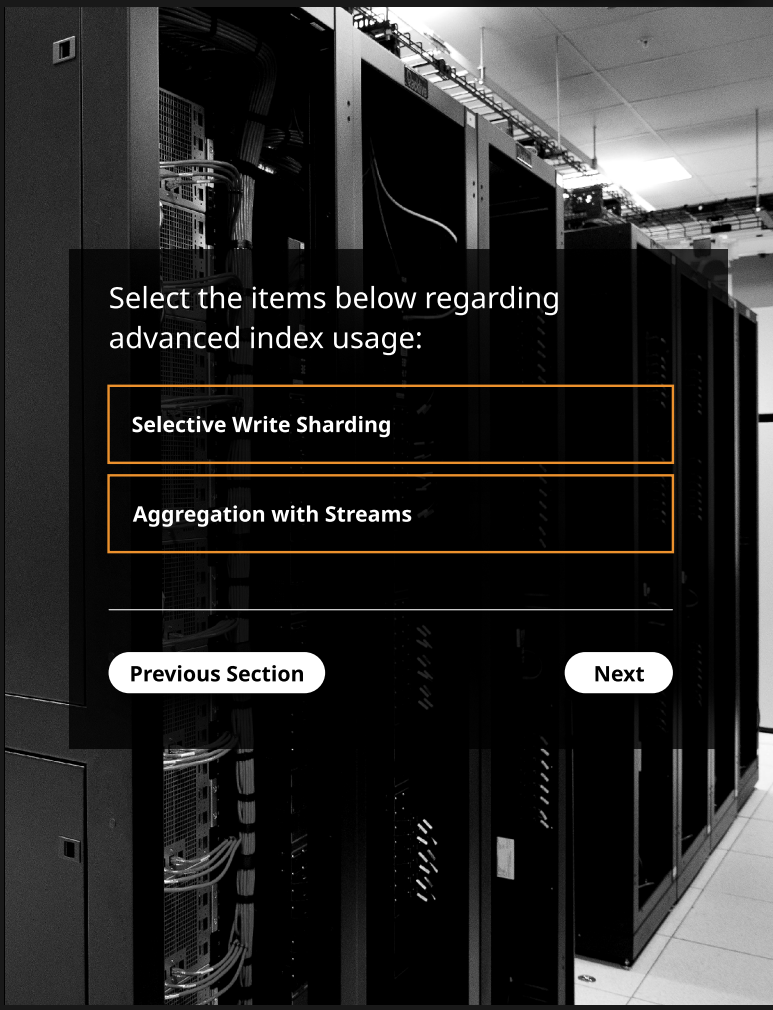
- Cross-region replication (global tables)
- Establish relationships across tables
- Messaging/notifications
- Aggregation/filtering
- Analytical reporting
- Archiving and auditing
- Search

Back to Main

Advanced DynamoDB

Triggers (Lambda Invoke)





Select the items below regarding advanced index usage:

- **Selective Write Sharding**
- **Aggregation with Streams**

Previous Section

Next

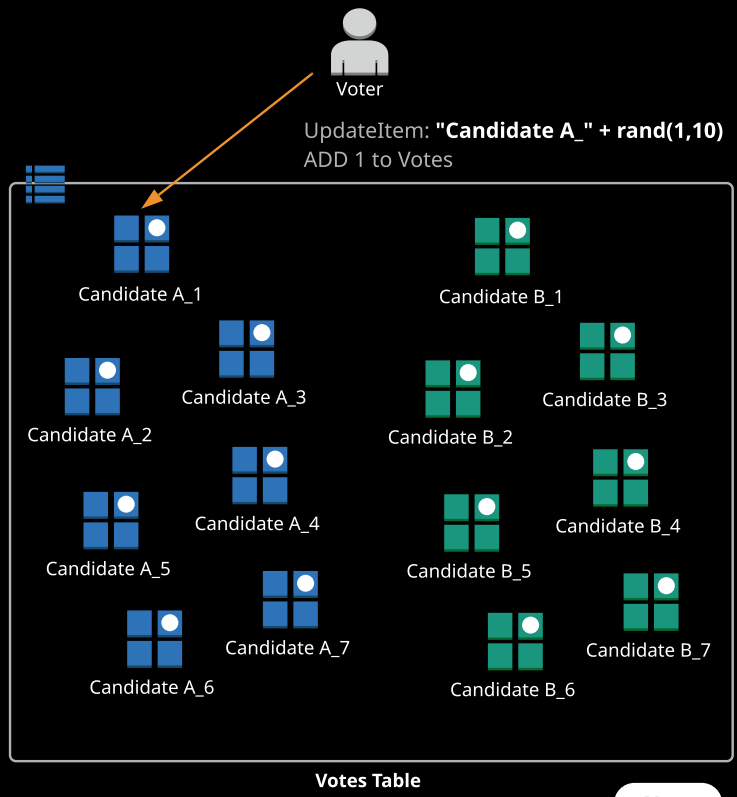
Back to Main

Advanced DynamoDB

Selective Write Sharding

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

Example: Voting Application



Next

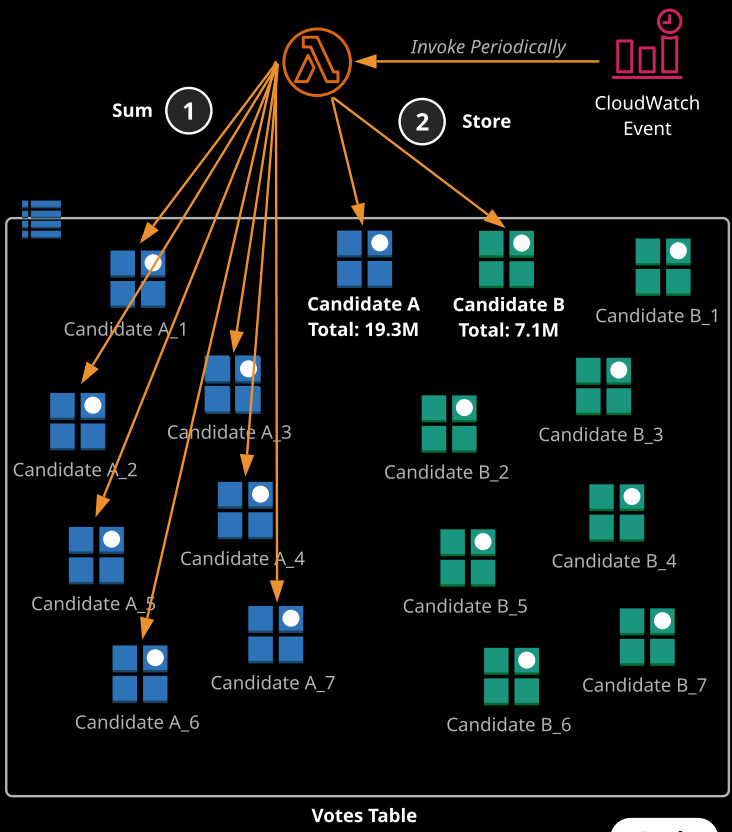
Back to Main

Linux Academy

Advanced DynamoDB

Selective Write Sharding

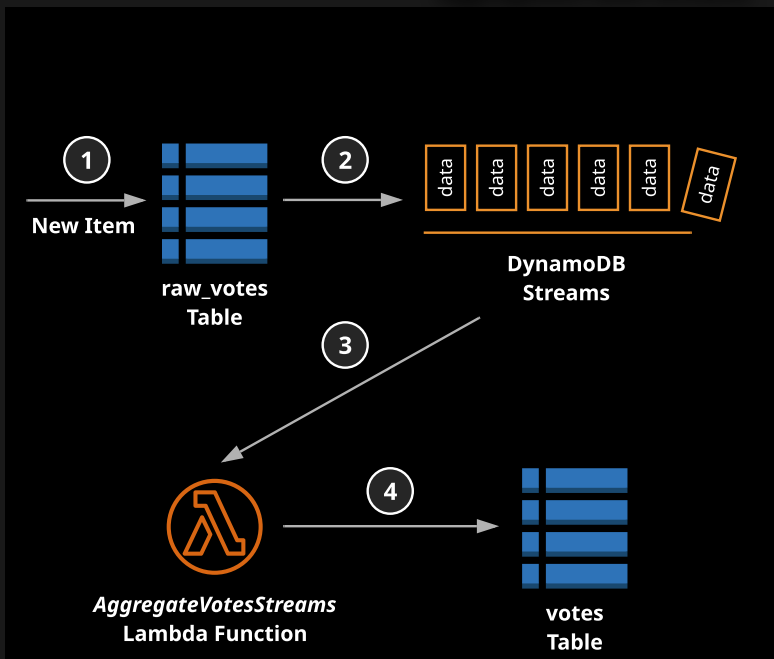
Aggregation: Use a scheduled Lambda function to query the GSI for the sharded totals, sum them, and write an item back to the table with the aggregate total of votes for the given candidate.



Back

Back to Main

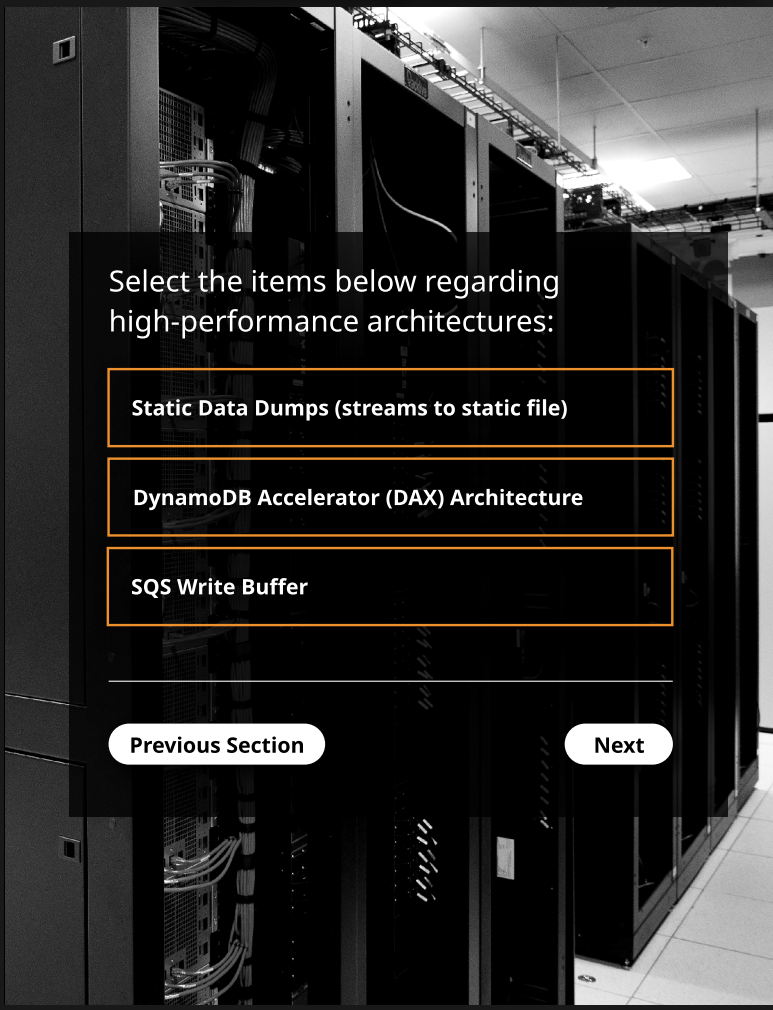
Linux Academy



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.

Advanced DynamoDB

Implementing High-Performance Architectures



Select the items below regarding high-performance architectures:

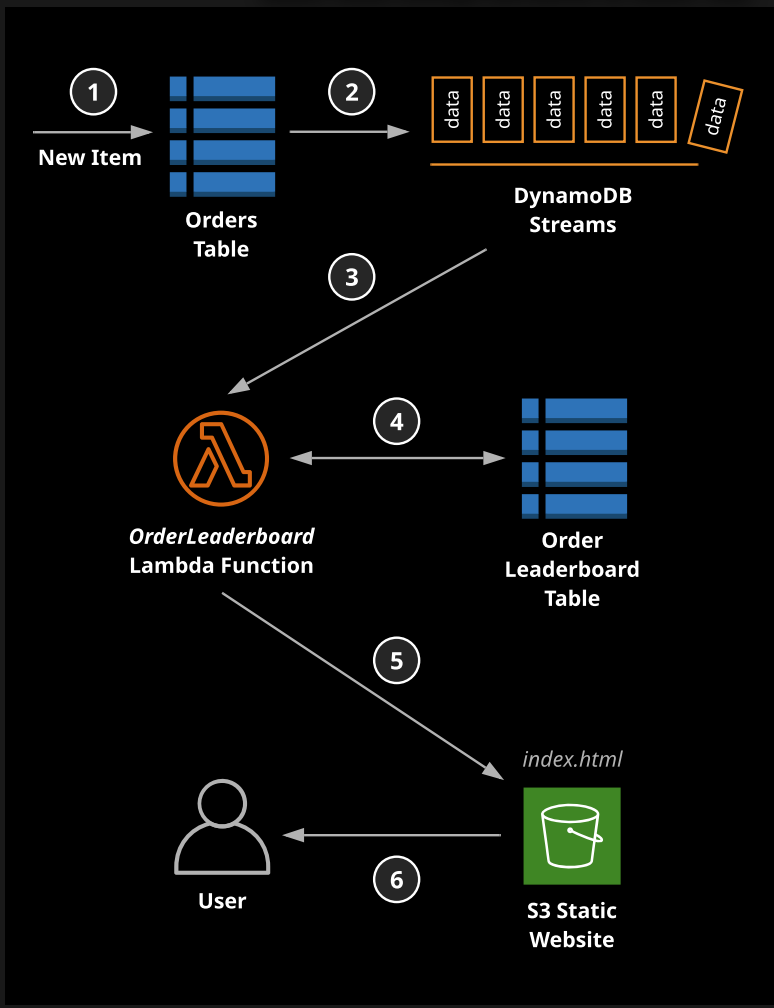
- **Static Data Dumps (streams to static file)**
- **DynamoDB Accelerator (DAX) Architecture**
- **SQS Write Buffer**

Previous Section **Next**

Back to Main

Advanced DynamoDB

Static Data Dumps (streams to static file)



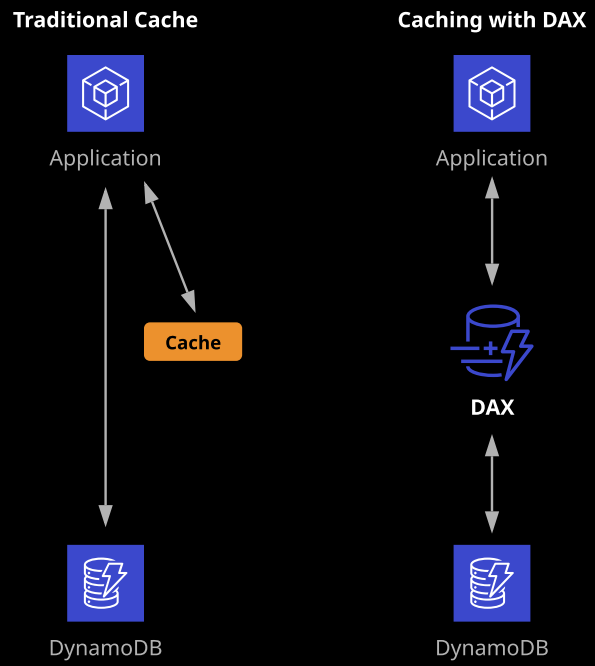
Back to Main

Advanced DynamoDB

DynamoDB Accelerator (DAX) Architecture

What Is DAX?

- Fully managed, highly available, in-memory cache for DynamoDB that delivers up to a 10x performance improvement
- Reduces request times from milliseconds to microseconds, even under load
- No need for developers to manage caching logic
- Compatible with DynamoDB API calls



Next

Back to Main

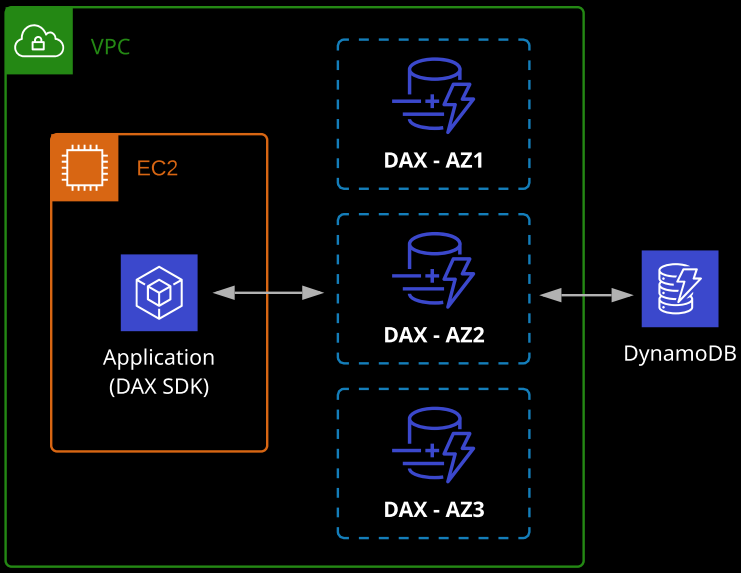
Linux Academy

Advanced DynamoDB

DynamoDB Accelerator (DAX) Architecture

High Availability

DAX is designed to run within a VPC. In the event of a primary node failure, DAX fails over to a read replica and promotes that node as the new primary.



API Compatibility

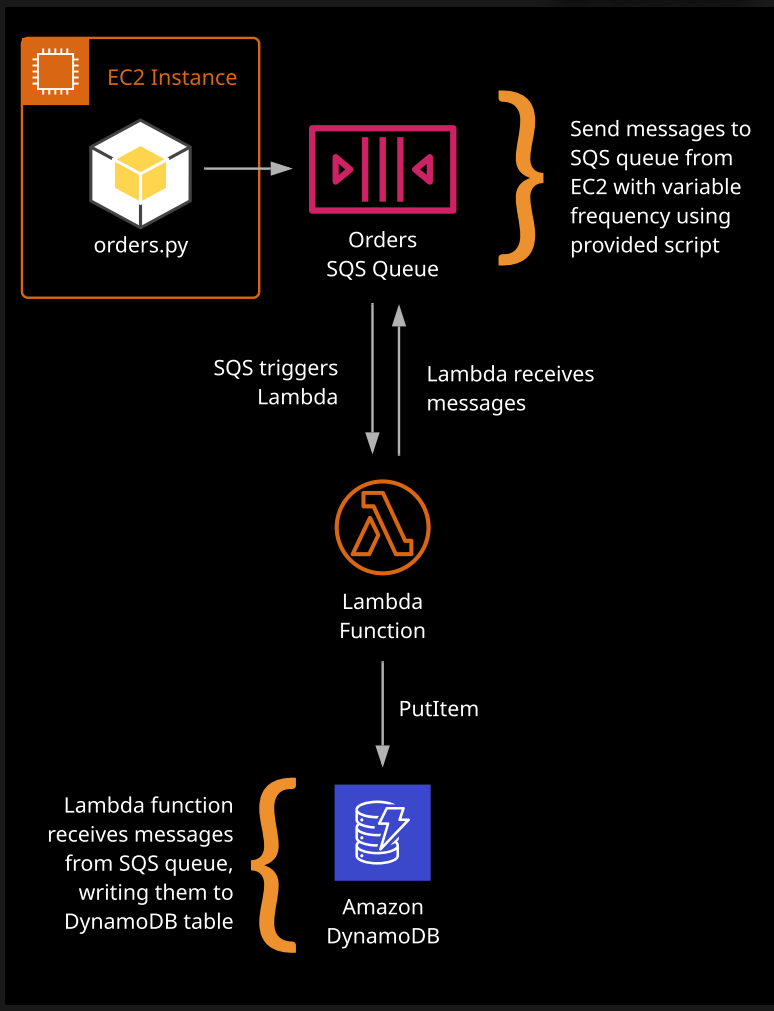
DAX is API compatible with DynamoDB.

- **Read APIs:** GetItem, BatchGetItem, Query, Scan
- **Modify APIs:** PutItem, UpdateItem, DeleteItem, BatchWriteItem
- **Control plane APIs:** Not supported (CreateTable, DeleteTable, etc.)

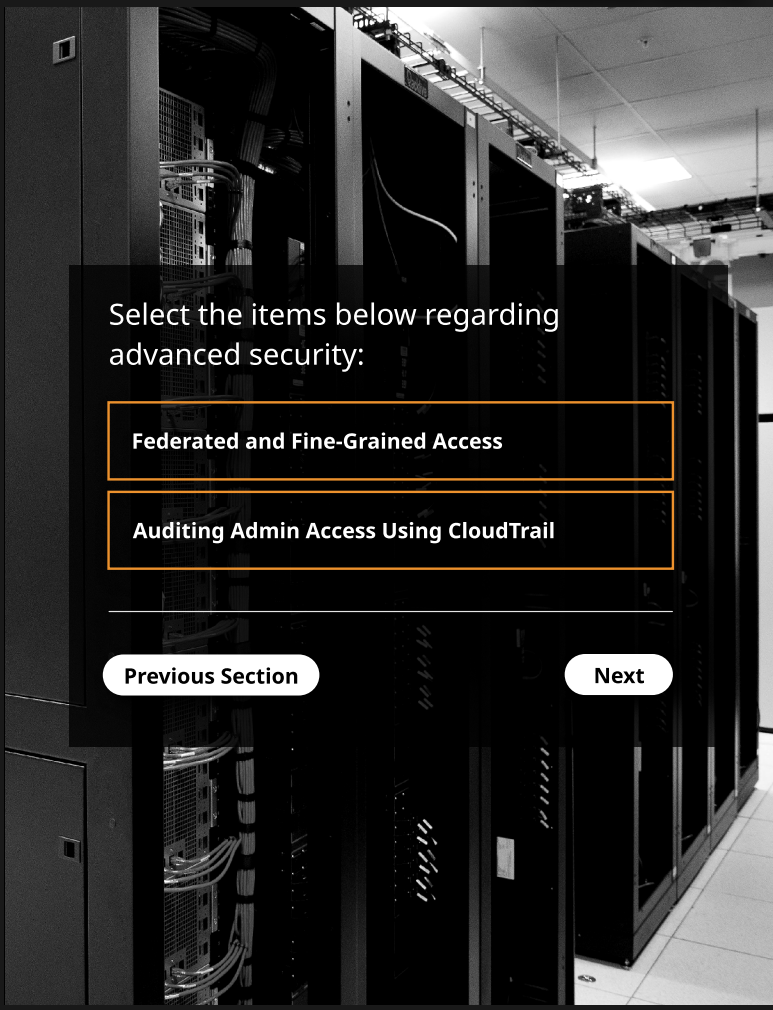
Back

Advanced DynamoDB

SQS Write Buffer



Back to Main



Select the items below regarding advanced security:

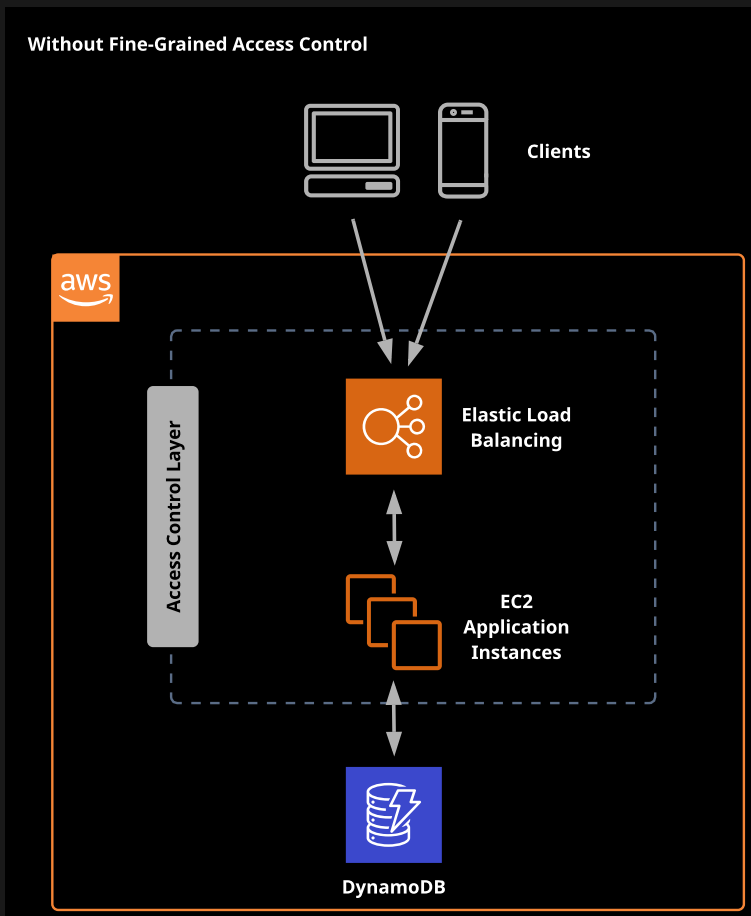
- [ ] Federated and Fine-Grained Access
- [ ] Auditing Admin Access Using CloudTrail

Previous Section
Next

Back to Main

Advanced DynamoDB

Federated and Fine-Grained Access



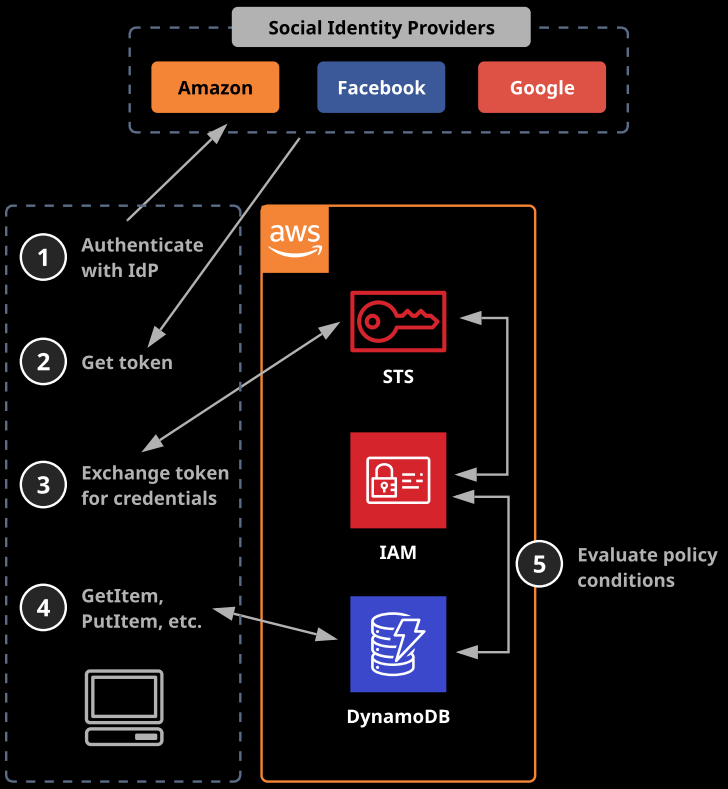
Next

Linux Academy

Advanced DynamoDB

Federated and Fine-Grained Access

With Fine-Grained Access Control



Back

Next

Back to Main

Linux Academy

Advanced DynamoDB

Federated and Fine-Grained Access


```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "FullAccessToUserItems",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:123456789012:table/MyTable"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": [
                        "${www.amazon.com:user_id}"
                    ]
                }
            }
        }
    ]
}
```

Advanced DynamoDB

Federated and Fine-Grained Access

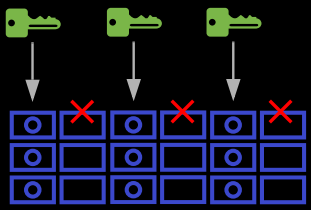

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAccessToOnlyItemsMatchingUserID",
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:BatchGetItem",
                "dynamodb:Query",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:123456789012:table/MyTable"
            ],
            "Condition": {
                "ForAllValues:StringEquals": {
                    "dynamodb:LeadingKeys": [
                        "${www.amazon.com:user_id}"
                    ],
                    "dynamodb:Attributes": [
                        "userid",
                        "artist_name",
                        "album_title",
                        "year",
                        "tracks"
                    ]
                },
                "StringEqualsIfExists": {
                    "dynamodb:Select": "SPECIFIC_ATTRIBUTES"
                }
            }
        }
    ]
}
```

IAM Policy Conditions

Control access to individual data items and attributes using IAM policies.

Grant permissions to a table, but restrict items based on PK values.

Sample Policy



Hide information so that only a subset of attributes is visible to the user.

Sample Policy

Back

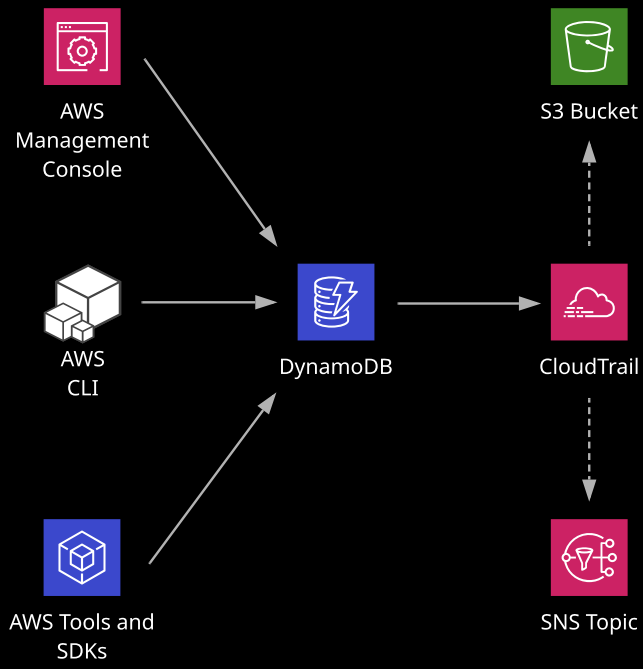
Advanced DynamoDB

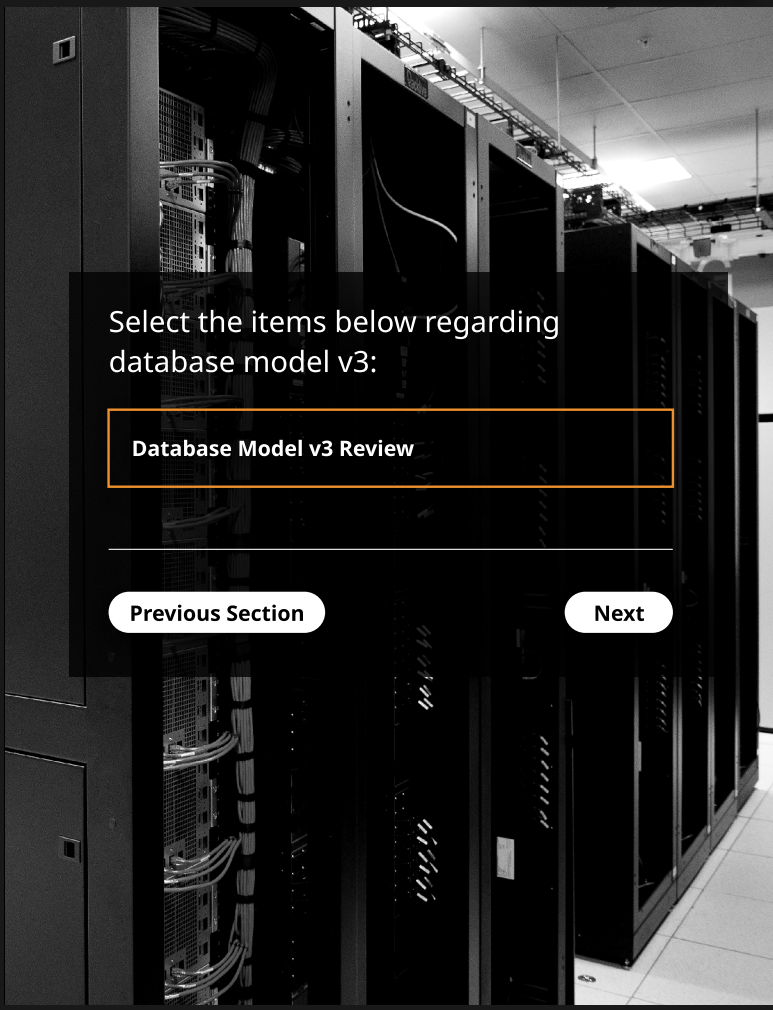
Auditing Admin Access Using CloudTrail

CloudTrail provides a record of actions taken by a user, role, or an AWS service in DynamoDB. CloudTrail captures all API calls for DynamoDB as events.

Most DynamoDB, DynamoDB Streams, and DAX API calls are captured by CloudTrail, and optionally stored in an S3 bucket.

SNS can notify you when CloudTrail publishes new log files to your S3 bucket.





Select the items below regarding database model v3:

Database Model v3 Review

Previous Section

Next

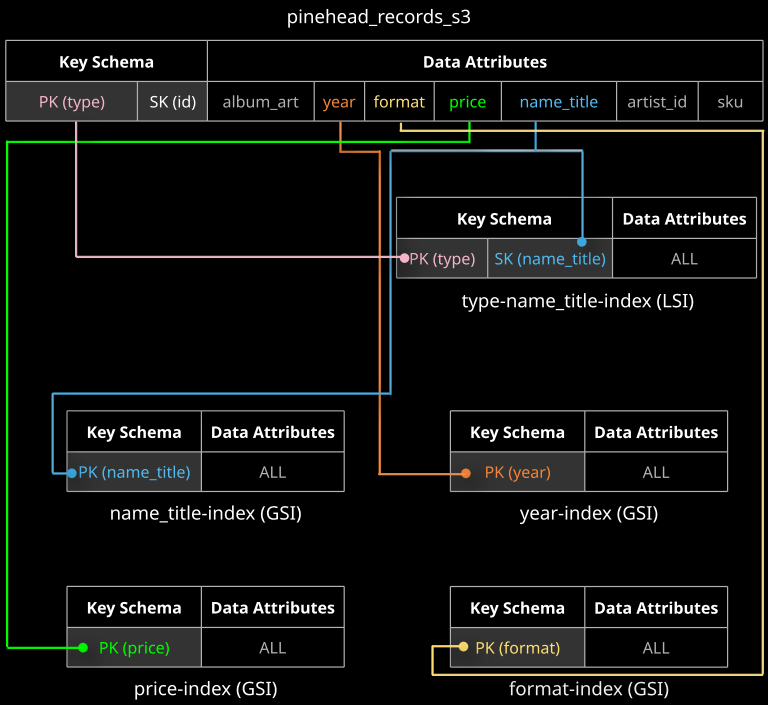
Back to Main

Advanced DynamoDB

Database Model v3 Review

This version of the Pinehead Records data model features:

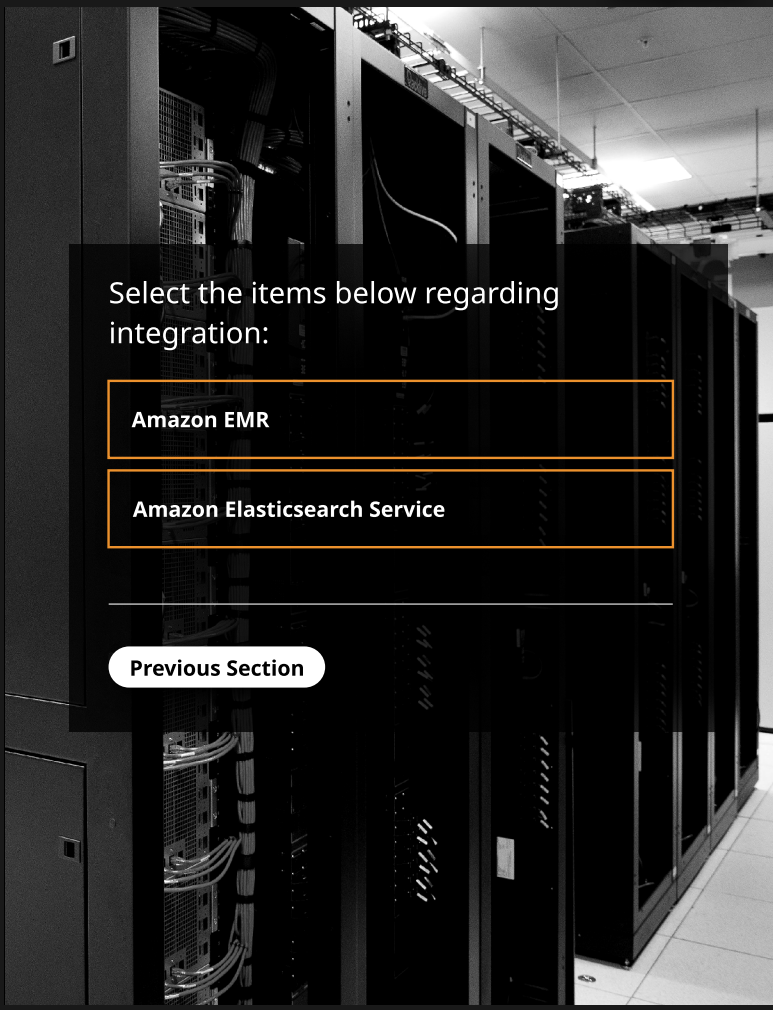
- Entity type partition key schema, where **type** is one of: album, artist, track
- More flexible schema to handle additional entity types



Course Navigation

Advanced DynamoDB
Section 6

Streams and Triggers
Streams
Triggers (Lambda Invoke)
Advanced Index Usage
Selective Write Sharding
Aggregation with Streams
Implementing High-Performance Architectures
Static Data Dumps (streams to static file)
DynamoDB Accelerator (DAX) Architecture
SQS Write Buffer
Advanced Security
Federated and Fine-Grained Access
Auditing Admin Access Using CloudTrail
Database Model v3
Database Model v3 Review
Integration
Amazon EMR
Amazon Elasticsearch Service



Select the items below regarding integration:

- Amazon EMR
- Amazon Elasticsearch Service

Previous Section

Back to Main

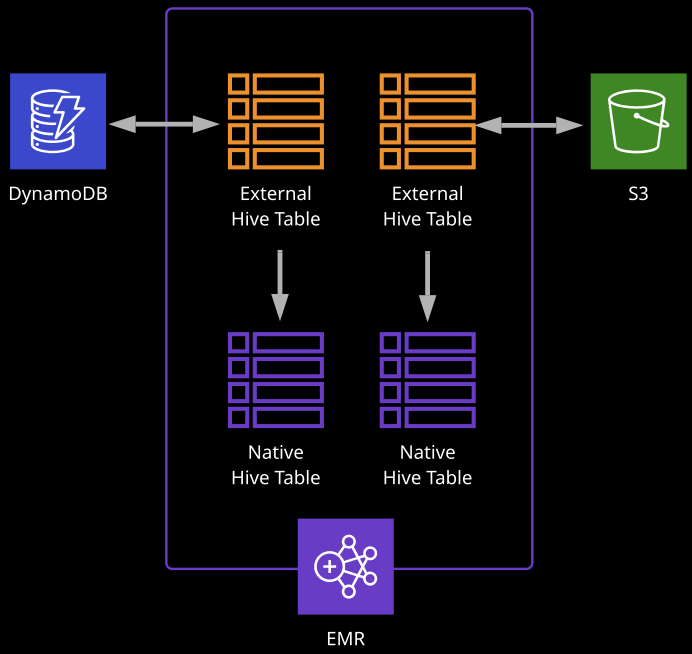
Advanced DynamoDB

Amazon EMR

Using **Amazon EMR** and **Apache Hive**, you can quickly and efficiently process large amounts of data in DynamoDB. Hive allows you to query **MapReduce** clusters using a SQL-like language called **HiveQL**.

EMR comes included with a version of Hive that supports DynamoDB operations such as:

- Loading DynamoDB data into an EMR cluster
- Querying live DynamoDB data using HiveQL
- Exporting data in DynamoDB to S3; importing data in S3 to DynamoDB
- Joining data in DynamoDB with data in S3



Advanced DynamoDB

Amazon Elasticsearch Service

You can use Lambda to send data to your Elasticsearch domain from DynamoDB. New data that arrives in the table triggers an event notification to a Lambda function, which then performs the indexing.