aws databases

# Introduction to Amazon DynamoDB

## - and some use case examples

Pete Naylor
Sr Database Specialist SA - AWS

# Agenda

- DynamoDB’s place in the history of purpose-built databases
- Key concepts
- How do I use DynamoDB?
  - Basic data models and controls
- Challenges with changing and imbalanced loads
  - And how DynamoDB deals with them
- Integrated DynamoDB architecture and data flows

# Database evolution and DynamoDB

# DynamoDB history

## How did we get here?

Andy Jassy
@ajassy
Following

We migrated 75 petabytes of internal data stored in nearly 7,500 Oracle databases to multiple AWS database services including Amazon DynamoDB, Amazon Aurora, Amazon Relational Database Service (RDS), and Amazon Redshift. The migrations were accomplished with little or no downtime, and covered 100% of our proprietary systems. This includes complex purchasing, catalog management, order fulfillment, accounting, and video streaming workloads. We kept careful track of the costs and the performance, and realized the following results:

- **Cost Reduction** – We reduced our database costs by over 60% on top of the heavily discounted rate we negotiated based on our scale. Customers regularly report cost savings of 90% by switching from Oracle to AWS.
- **Performance Improvements** – Latency of our consumer-facing applications was reduced by 40%.
- **Administrative Overhead** – The switch to managed services reduced database admin overhead by 70%.

868 Retweets 2,281 Likes

76 868 2.3K

# DynamoDB history

## Motivations at Amazon (and elsewhere)

1. Reduce dependence on commercially licensed relational database engines
2. Minimize operational complexity and administrative overhead
3. Provide best possible customer experience across all data indexing needs

"A one size fits all database doesn't fit anyone"

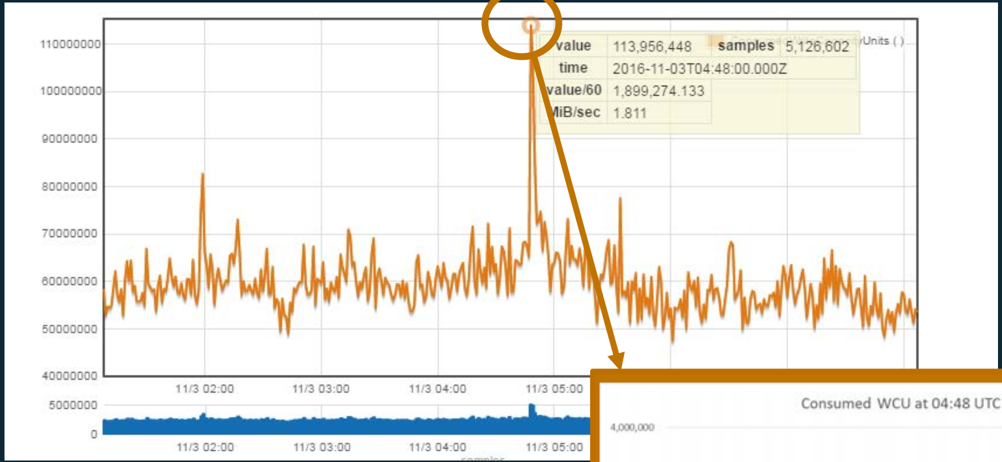
- Werner Vogels

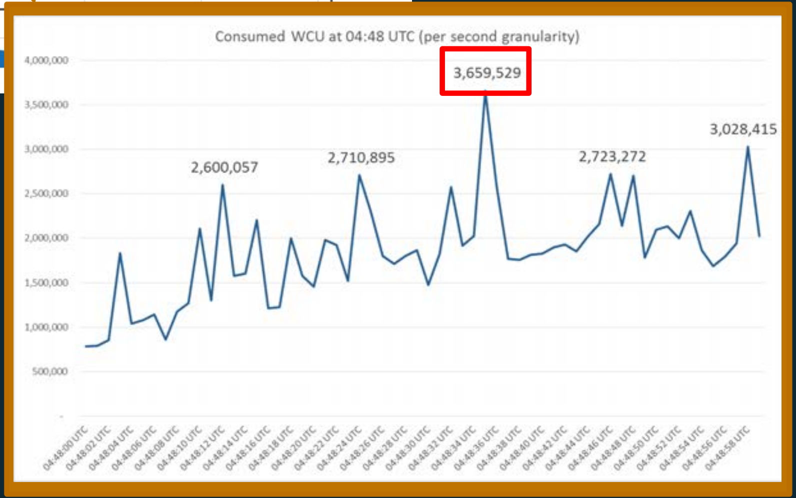
# Internet-scale applications

| | |
|---|---|
| Users | 1M+ |
| Data volume | TB-PB-EB |
| Locality | Global |
| Performance | Milliseconds-microseconds |
| Request rate | Millions |
| Access | Mobile, IoT, devices |
| Scale | Incrementally based on traffic |
| Economics | Pay as you go |
| Developer access | Instant API access |


By: Arturo Pardavila III
https://www.flickr.com/photos/apardavila/albums/72157674624352641

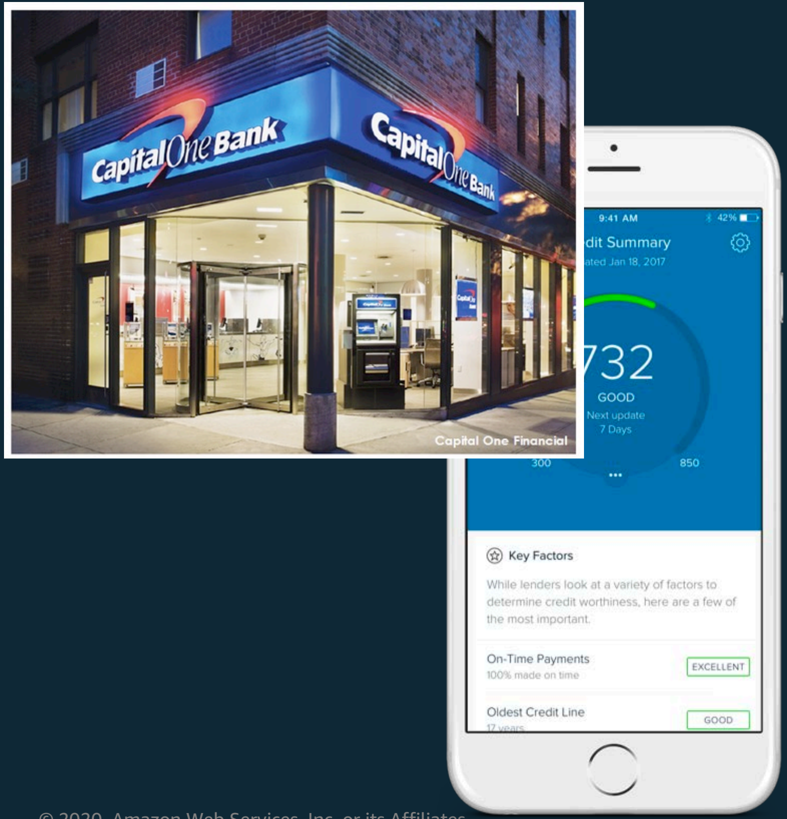
# Snap (Snapchat)




Database writes peak *seconds* after Chicago Cubs win the World Series.

# A migration from mainframe




- Retail business ran on mainframe, which became a bottleneck
- Migrated financial transaction data to DynamoDB
- Unbound scale for customers and app developers

"We built a secure and resilient cloud infrastructure that could solve the scalability and reliability problems with a serverless architecture."

Srini Uppalapati
Capital One



aws

# SQL and DynamoDB side by side

| Traditional SQL | DynamoDB |
| --- | --- |
| **Optimized for storage** | **Optimized for compute** |
| Normalized/relational | Denormalized/hierarchical |
| Ad hoc queries | Instantiated views for known patterns |
| Scale vertically | Scale horizontally |
| Good for OLAP | Built for OLTP at scale |

# Core Strengths of DynamoDB
## Sounds cool, but when should I use it?

**Priorities**

- Security
- Durability
- Scalability
- Availability
- Low Latency
- Easy to Use
- Easy to Manage

Considerations:

- SQL / NoSQL
- Relational / Non-Relational
- Operational / Analytical

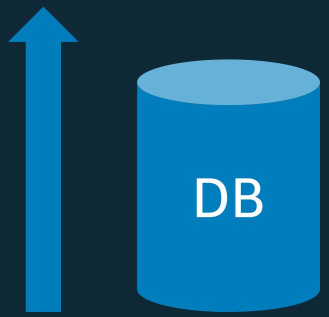
DynamoDB solves for:

- Horizontal scaling
- Decoupled compute/storage
- Asynchronous roll-ups/aggregations
- Availability/Durability/Latency
- Well-known access patterns
- Schema flexibility

# Key concepts

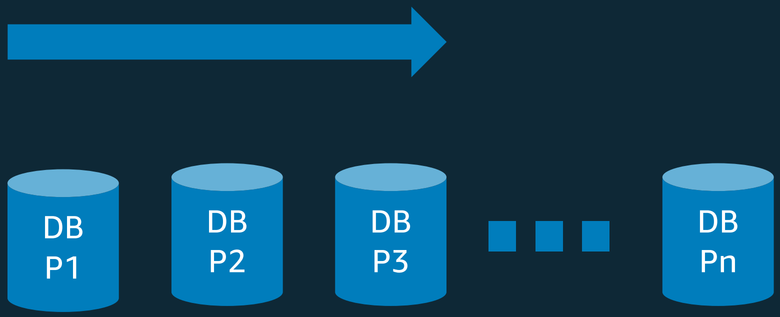
# Scaling databases

## Traditional SQL




Scale up

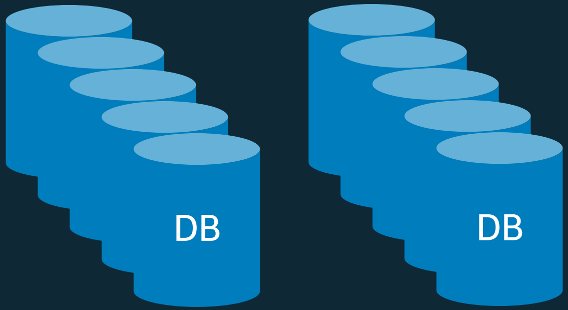
## NoSQL




Scale out to many shards

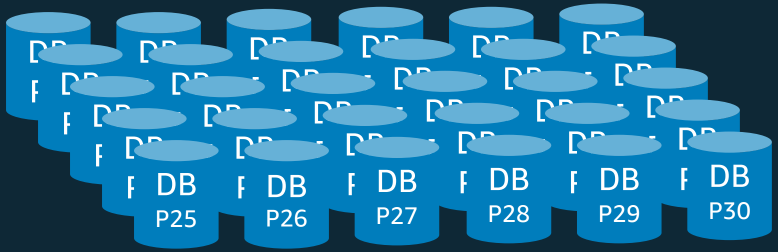
# Scaling NoSQL databases

## Most NoSQL databases




Servers and clusters

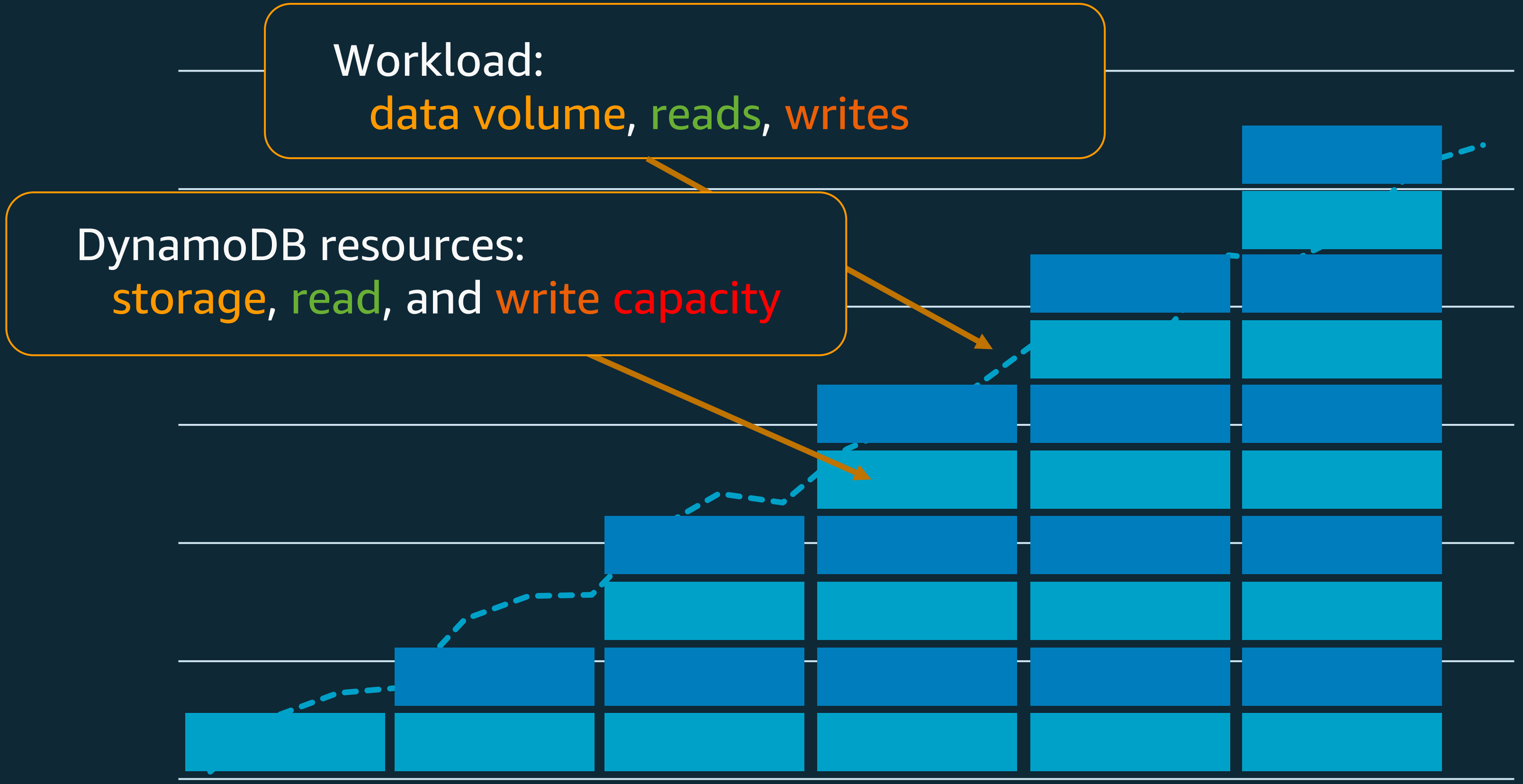
## DynamoDB




DynamoDB: partitions

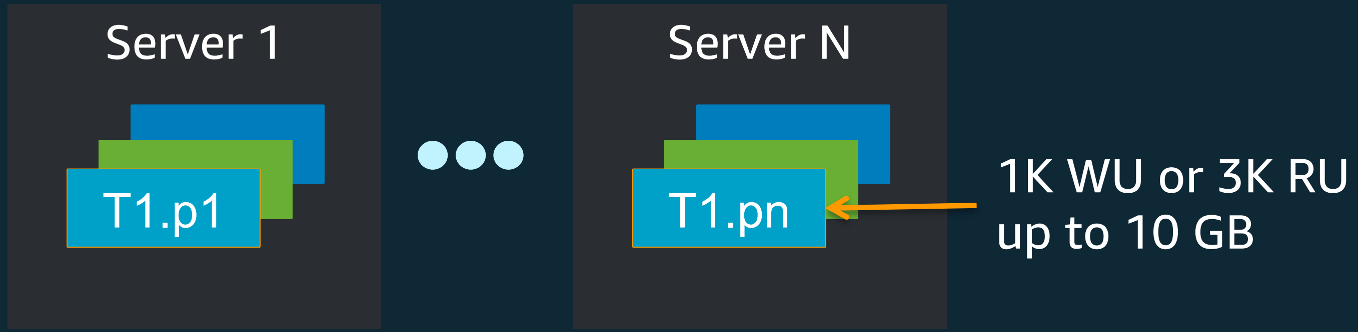
Basic premise: There is a way to shard data that's horizontally scalable.

# Incremental scaling with DynamoDB

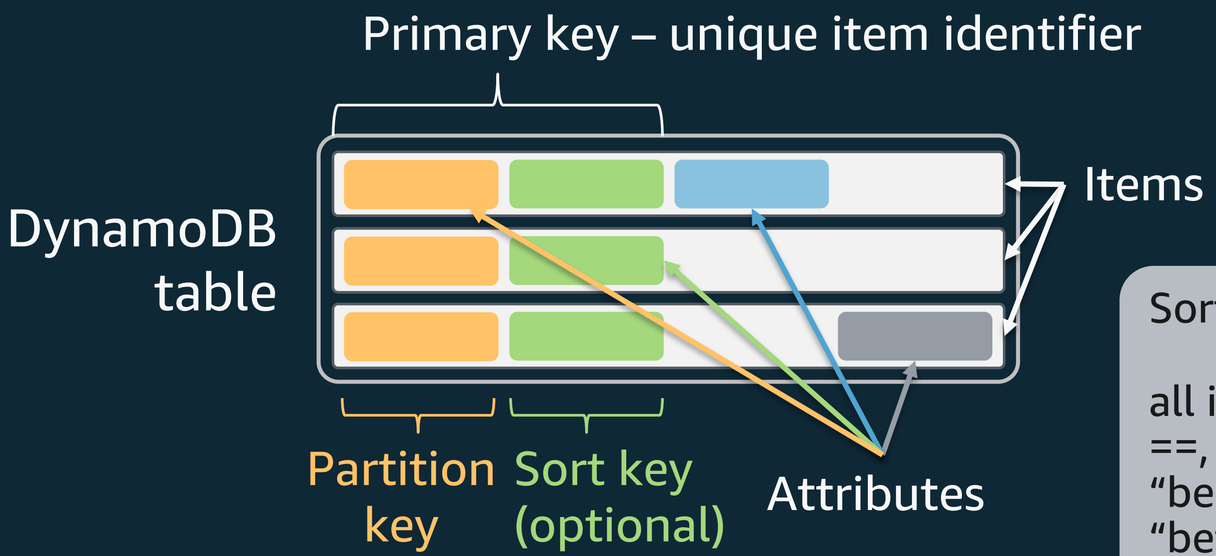


# You work with tables…


# DynamoDB does the rest under the hood…



# Terminology




- 1:1 relationships
- distribute traffic
- collection identity

- 1:many relationships
- **collect** related items
- efficient filtering
- sorting

Sort key conditions:

all items
==, <, >, >=, <=
“begins with”
“between”

sorted results
counts
top/bottom N

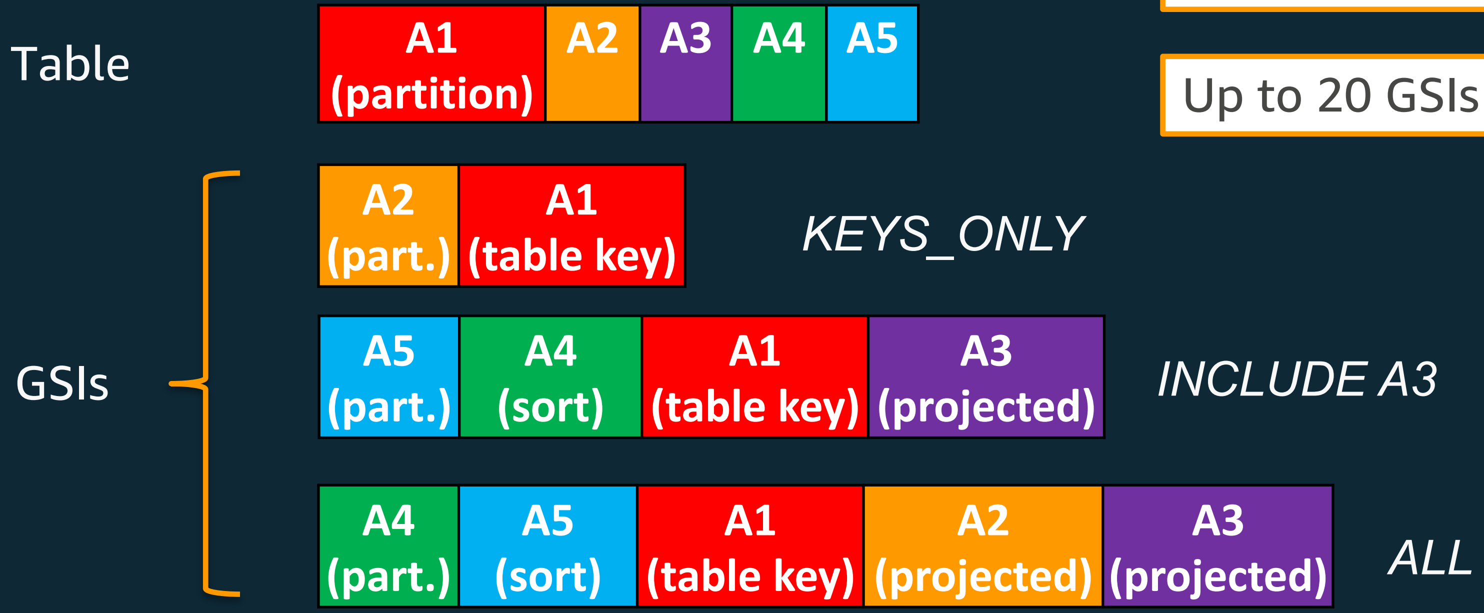
# Global secondary index (GSI)

Alternate partition (+sort) key for alternate materialized view

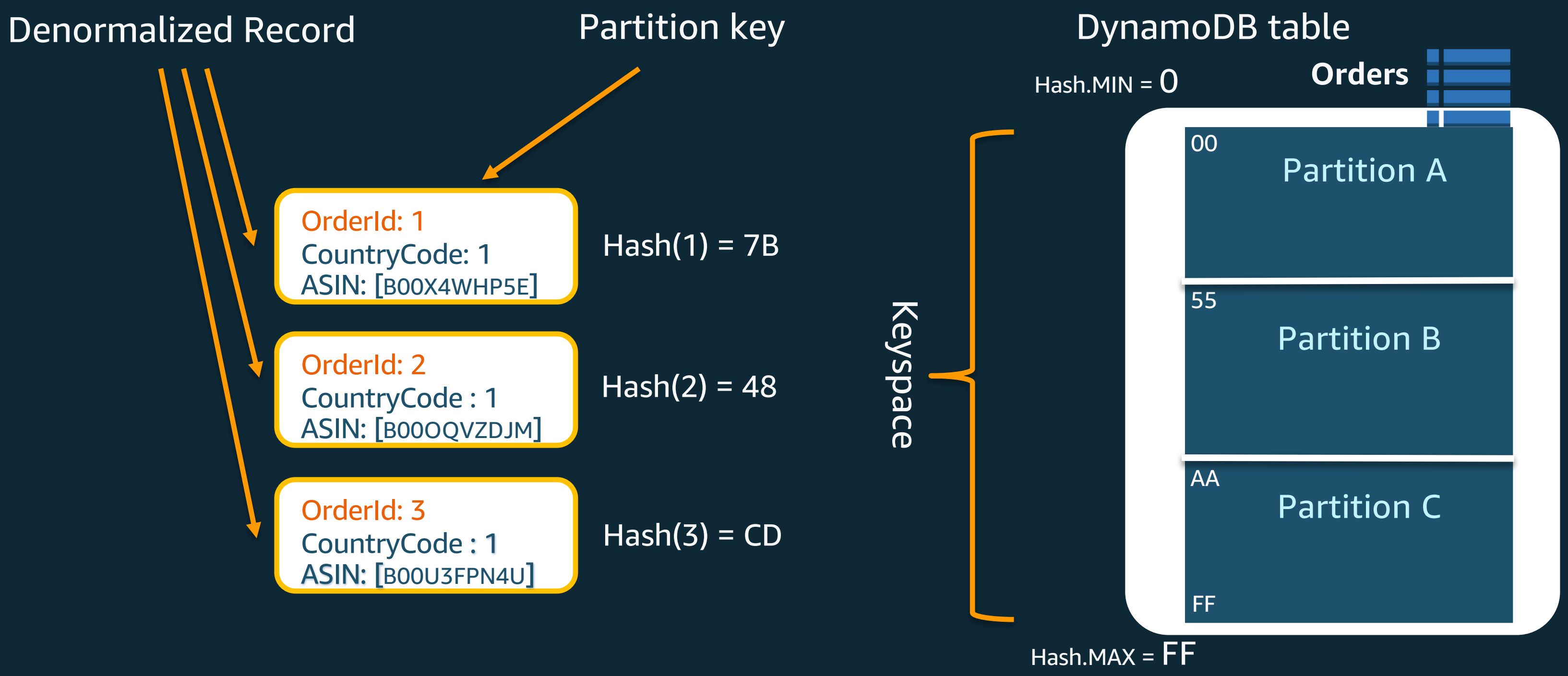
Online indexing

Capacity provisioned separately from table

Up to 20 GSIs per table



# Sharding/partitioning



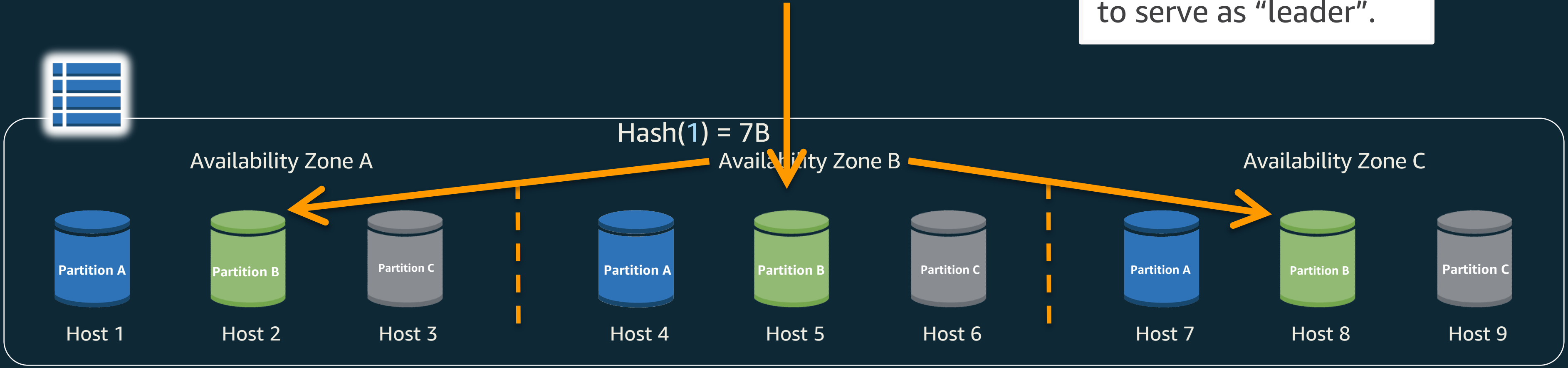

Related data is stored together for efficient access

# A view “from a different angle”

OrderId: 1
CustomerId: 1
ASIN: [B00X4WHP5E]

**Three Replicas**

Across 3 Availability Zones in the Region - one replica is elected to serve as “leader”.




**CustomerOrdersTable**

# Modeling data for DynamoDB

# Partition key

- A good sharding (partitioning) scheme affords even distribution of both data and workload as they grow

- Key concept: partition key as the dimension of scalability
    - Distribute traffic and data across partitions – horizontal scaling

- Ideal scaling conditions:
    - The partition key is from a high cardinality set (that grows)
    - Requests are evenly spread over the key space
    - Requests are evenly spread over time

# Denormalization

## Traditional database

**CART**

CartId: 1,
Version: v3

**CART_ITEM**

ItemId: 2, Qty: 1,
CartId: 1

Normalized schema: multiple items – one table per entity type

## DynamoDB

**CART**

CartId: 1,
Version: v3,
Items: [
{ID: 2, Qty: 1},
{ID: 5, Qty: 2}]

Denormalized: one item with flexible schema

# Ensuring data consistency on updates

## Example: add/remove shopping cart items

```
1. get cart => v_read = Version
2. update cart:
    IF Version = v_read
        add/remove cart items
        ++Version
    ELSE go back to Step 1.
```

Use **`ConditionExpression`** in DynamoDB

Optimistic concurrency control



aws

# Updating cart: data consistency using OCC

## 1. Get the cart: GetItem

```
{   "TableName": "Cart",
    "Key": {"CartId": {"N": "2"}}
}
```

**CART**

CartID:2,
Version: 3,
CartItems: [
 {ID: 2, Qty: 1},
 {ID: 5, Qty: 2}]

## 2. Update the cart: conditional PutItem (or UpdateItem)

```
{   "TableName": "Cart",
    "Item": {
        "CartID": {"N": "2"},
        "Version": {"N":"4"},
        "CartItems": {…}
    },
    "ConditionExpression": "Version = :ver",
    "ExpressionAttributeValues": {":ver": {"N": "3"}}
}
```

- ✓ Use conditions to implement optimistic concurrency control ensuring data consistency
- ✓ Single-item operations are ACID
- ✓ GetItem call can be eventually consistent

# Yes, you can have strongly consistent read-after-write and concurrency control with DynamoDB

# One-to-one relationships or key-values

- Use a table or GSI with a partition key
- Use GetItem or BatchGetItem API

Example: Given a user or email, get attributes

| Users table | |
|---|---|
| Partition key | Attributes |
| UserId = bob | Email = bob@example.com, JoinDate = 2011-11-15 |
| UserId = fred | Email = fred@example.com, JoinDate = 2011-12-01 |

| Users-Email-GSI | |
|---|---|
| Partition key | Attributes |
| Email = bob@example.com | UserId = bob, JoinDate = 2011-11-15 |
| Email = fred@example.com | UserId = fred, JoinDate = 2011-12-01 |

# One-to-many relationships or parent-children

- Use a table or GSI with a partition and sort key
- Use the Query API to get multiple items

Example: Given a device, find all readings between epoch X, Y

| Device-measurements | | |
|---|---|---|
| Part. key | Sort key | Attributes |
| DeviceId = 1 | epoch = 5513A97C | Temperature = 30, pressure = 90 |
| DeviceId = 1 | epoch = 5513A9DB | Temperature = 30, pressure = 90 |

# Many-to-many relationships

- Use a table and GSI with the partition and sort key elements switched
- Use the Query API

Example: Given a user, find all games. Or given a game, find all users.

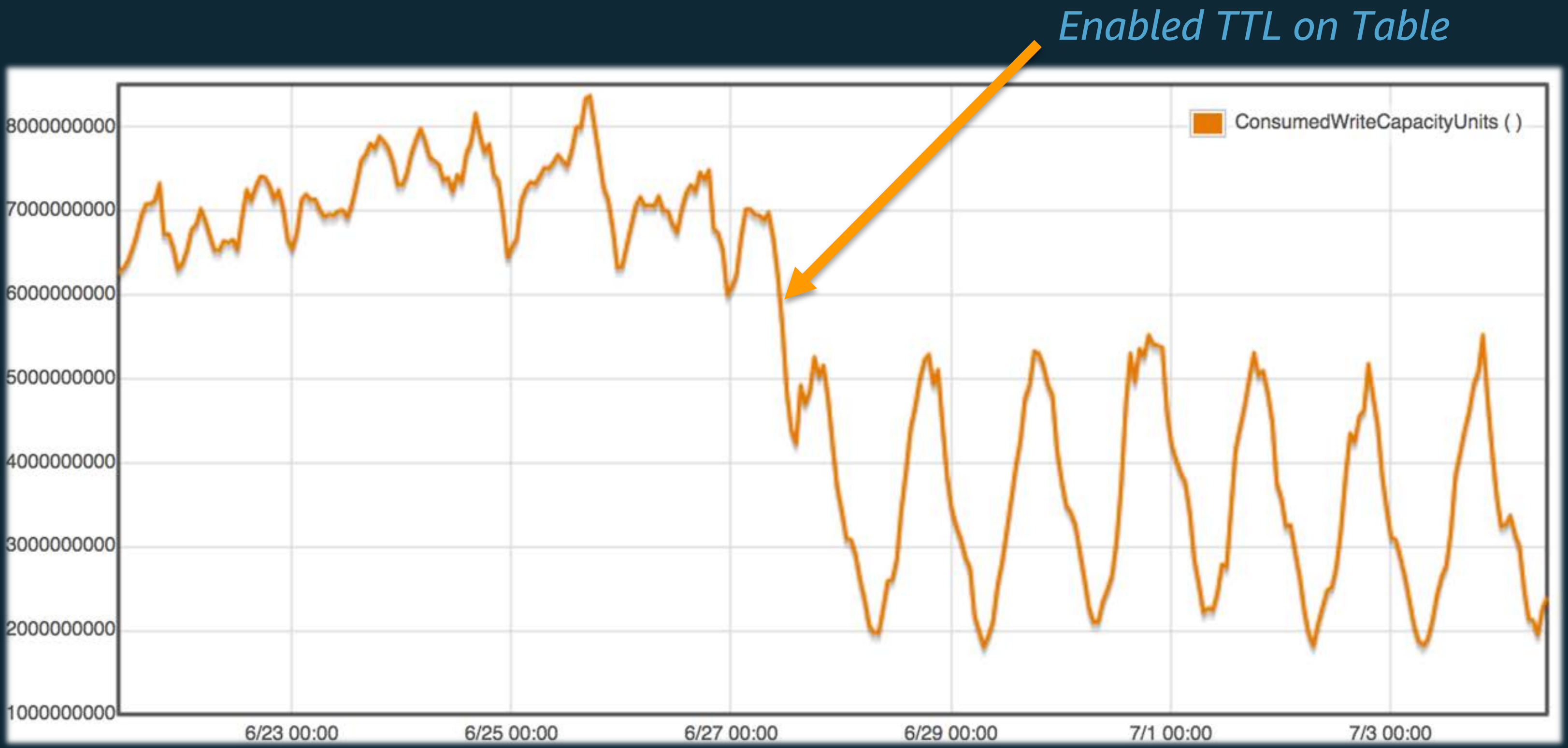
| User-Games-Table | |
|---|---|
| Part. key | Sort key |
| UserId = bob | GameId = Game1 |
| UserId = fred | GameId = Game2 |
| UserId = bob | GameId = Game3 |

| Game-Users-GSI | |
|---|---|
| Part. key | Sort key |
| GameId = Game1 | UserId = bob |
| GameId = Game2 | UserId = fred |
| GameId = Game3 | UserId = bob |

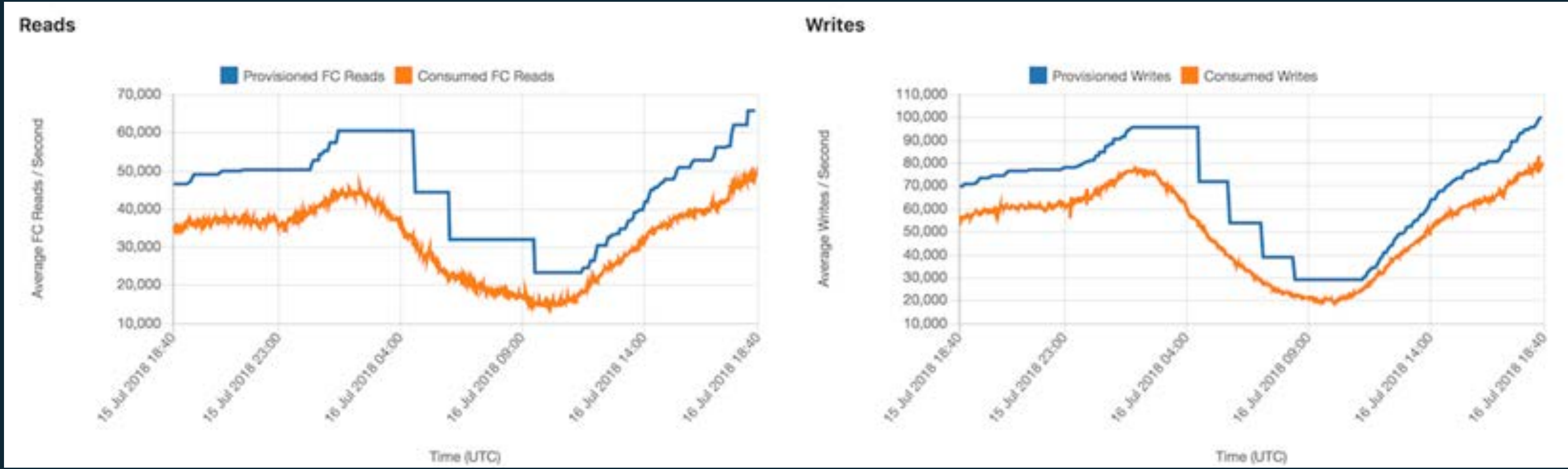
# Yes, you can model complex data relationships with DynamoDB

# Challenges with growing datasets, variable throughput, imbalanced workloads

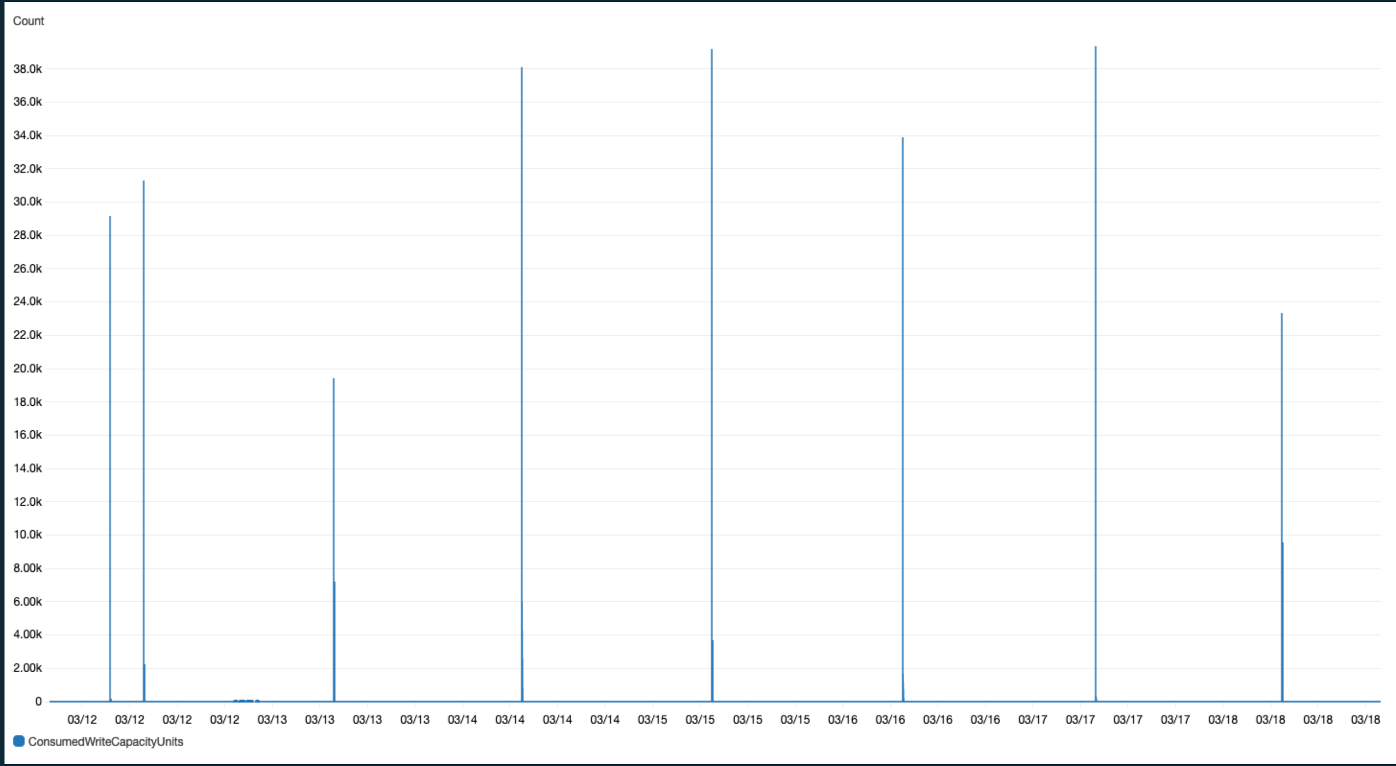
# Time-To-Live (TTL)



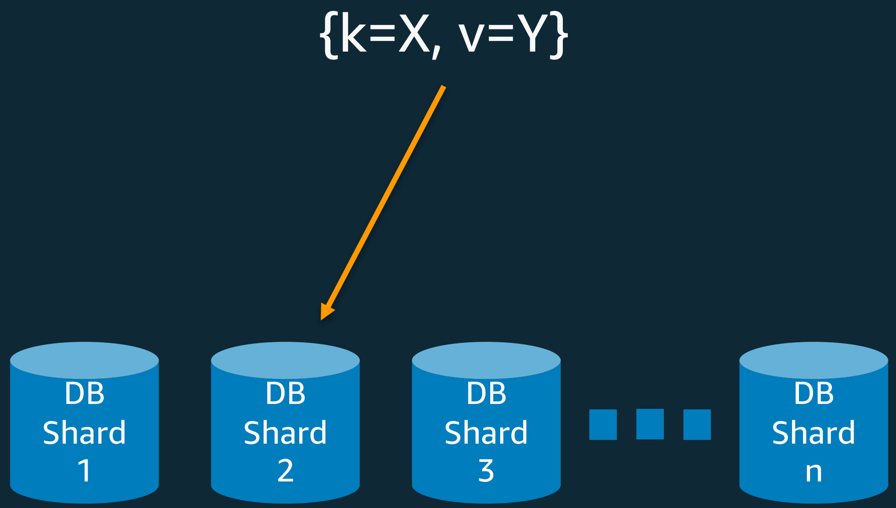
# Provisioned Mode with Auto Scaling



# On-demand Mode: Spiky workloads

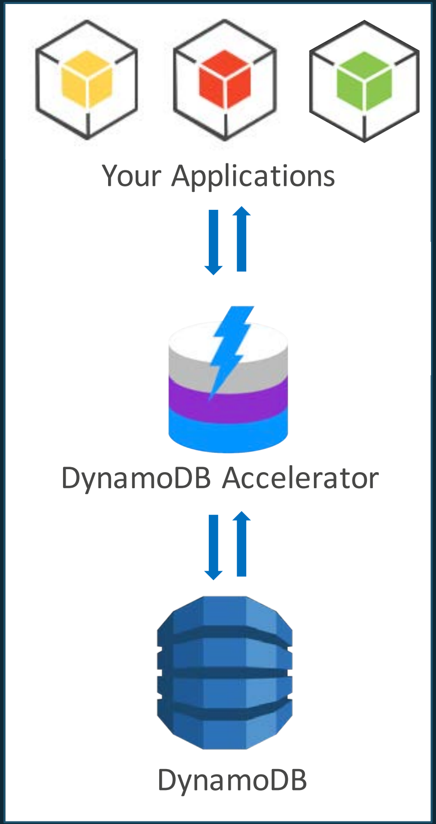


# Imbalanced load and DynamoDB Adaptive Capacity




Poor distribution of traffic across the indexed data

# Dealing with concentrated read throughput




## DynamoDB Accelerator (DAX)

Fully managed, highly available: handles all software management, fault tolerant, replication across multi-AZs within a region

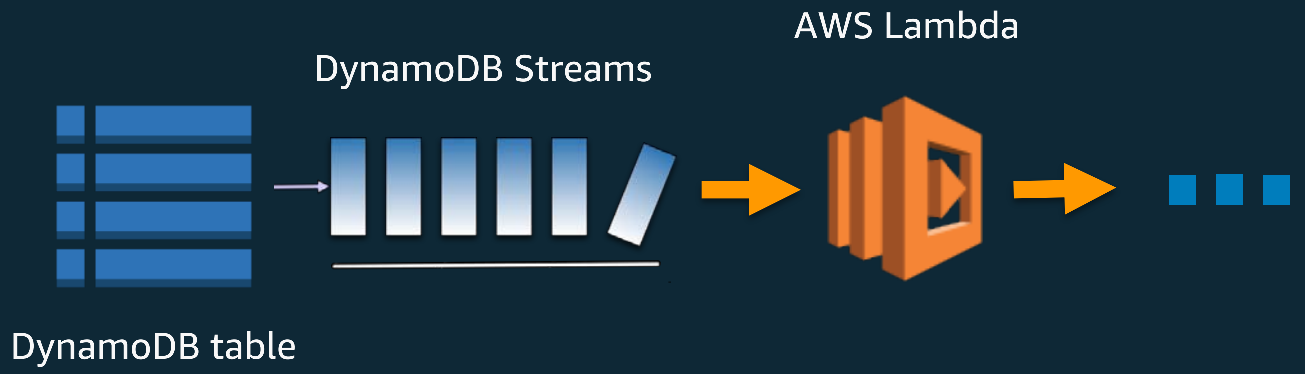
DynamoDB API compatible: seamlessly caches DynamoDB API calls, no application re-writes required

Write-through: DAX handles caching for writes

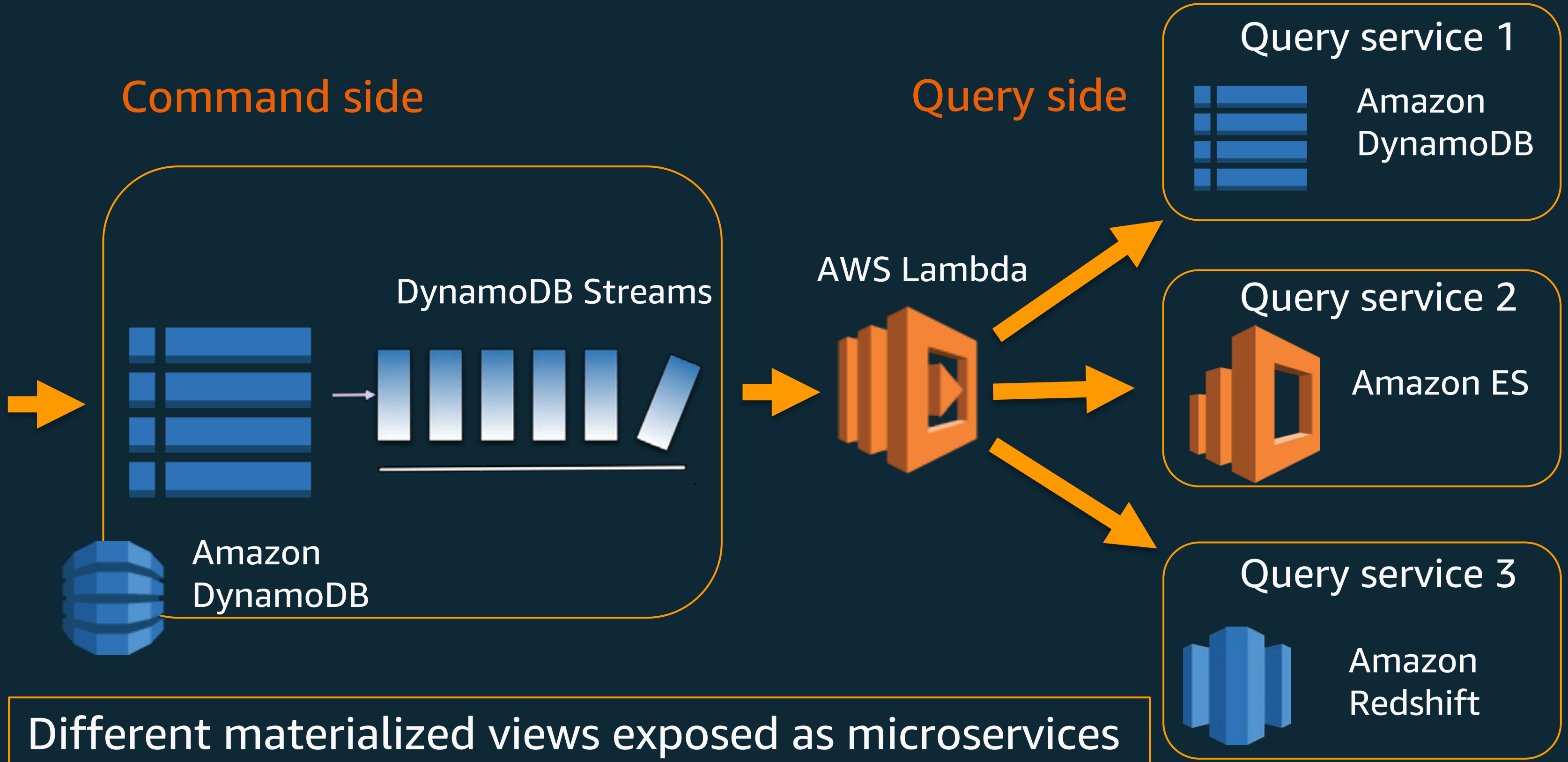
# Integrating DynamoDB into your data flow

# Complex queries and analytics

- Use the service that best meets the requirements
    - Amazon Athena, Amazon Redshift, Amazon Elasticsearch Service…
- Deliver data updates reliably with DynamoDB Streams
    - Integrates with AWS Lambda
    - End-to-end serverless



# Querying in microservices architectures




Different materialized views exposed as microservices

# Closing summary

- Data management at internet scale gave rise to DynamoDB

  - and to polyglot persistence: use the right database for each job

- DynamoDB: highly-automated (serverless) distributed database

  - ideal for mission-critical OLTP use cases

- Remove scaling concerns – distribute your data and traffic
- You *can* have data consistency and integrity in DynamoDB
- You *can* model relationships beyond key-value in DynamoDB
- Distributed databases are hard to operate - use DynamoDB!

# Thank you!