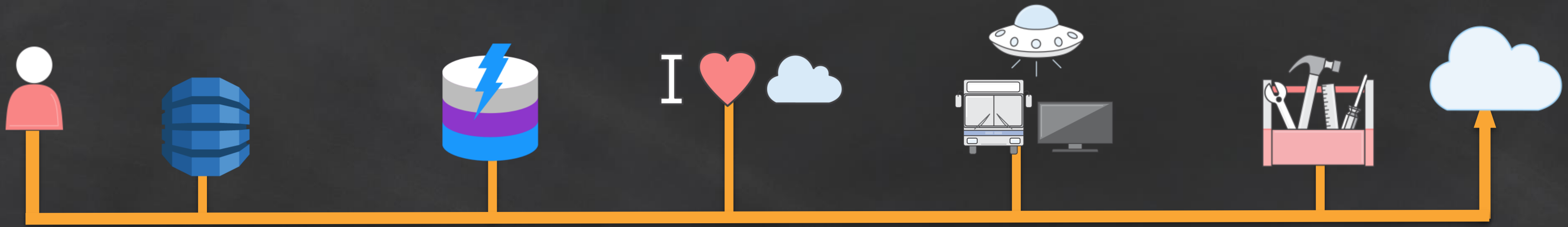
Deep Dive: Amazon DynamoDB

Sean Shriver, AWS

Oct 2017

aws | Pop-up Loft

Plan



Amazon DynamoDB

- Foundations
- Tables
- Indexes
- Partitioning

New Features

- TTL
- VPC Endpoints
- Auto Scaling
- DAX

Dating Website

- DAX
- GSIs

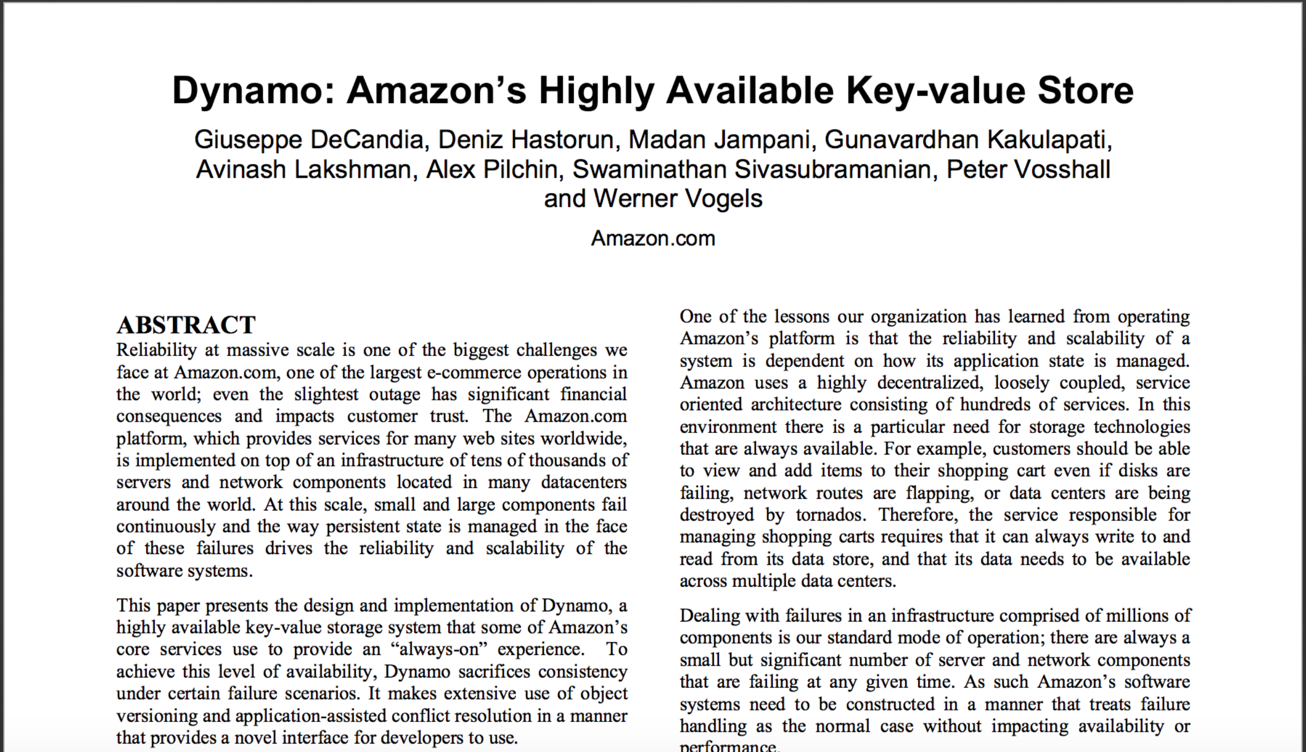
Serverless IoT

- TTL
- Streams
- DAX

Getting Started

- Developer Resources

Dynamo whitepaper



Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

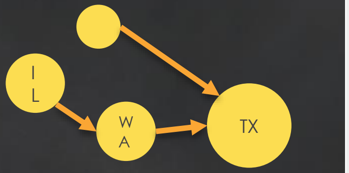
Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

NoSQL foundations

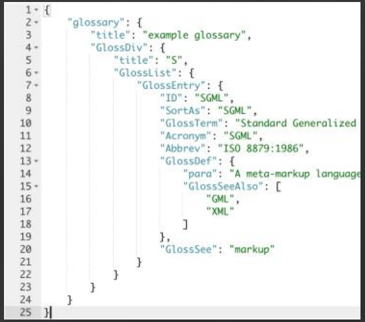
Key Value

0000	{“Texas”}
0001	{“Illinois”}
0002	{“Oregon”}

Graph



Document



```
{
    "glossary": {
        "title": "example glossary",
        "GlossDiv": {
            "title": "S",
            "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                    "SortAs": "SGML",
                    "GlossTerm": "Standard Generalized
                    "Acronym": "SGML",
                    "Abbrev": "ISO 8879:1986",
                    "GlossDef": {
                        "para": "A meta-markup language
                        "GlossSeeAlso": [
                            "GML",
                            "XML"
                        ]
                    },
                    "GlossSee": "markup"
                }
            }
        }
    }
}
```

Column-family

Key	0000-0000-0000-0001	
Column	Game	Heroes
	Version	3.4
	CRC	ADE4

Dynamo: *Amazon’s* *Highly Available* *Key-value* *Store*

Fall 2007

→

Meetup
235 2nd St
San
Francisco

June 2009

→

January 2012

What (some) customers store in NoSQL DBs

Market Orders

Tokenization
(PHI, Credit Cards)

User Profiles
(Mobile)

Chat Messages

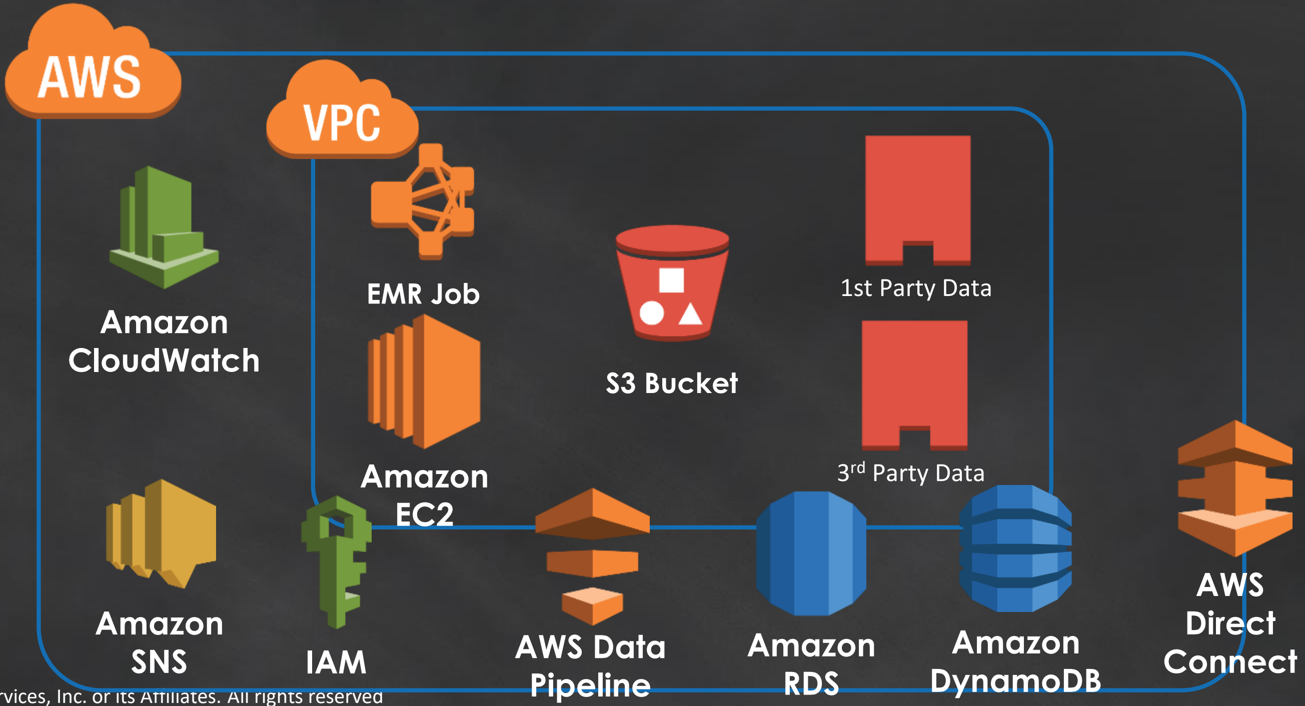
IoT Sensor Data
(& device status!)

Social Media Feeds

File Metadata

DataXu's Attribution Store

"Attribution" is the marketing term of art for allocating full or partial credit to individual advertisements that eventually lead to a purchase or other desired consumer interaction.



Technical challenges

- Amazon EC2 Instances
- EBS Volumes
- CloudWatch Metrics
- Notifications
- Scaling new AZs, new Regions

Amazon DynamoDB

Highly available

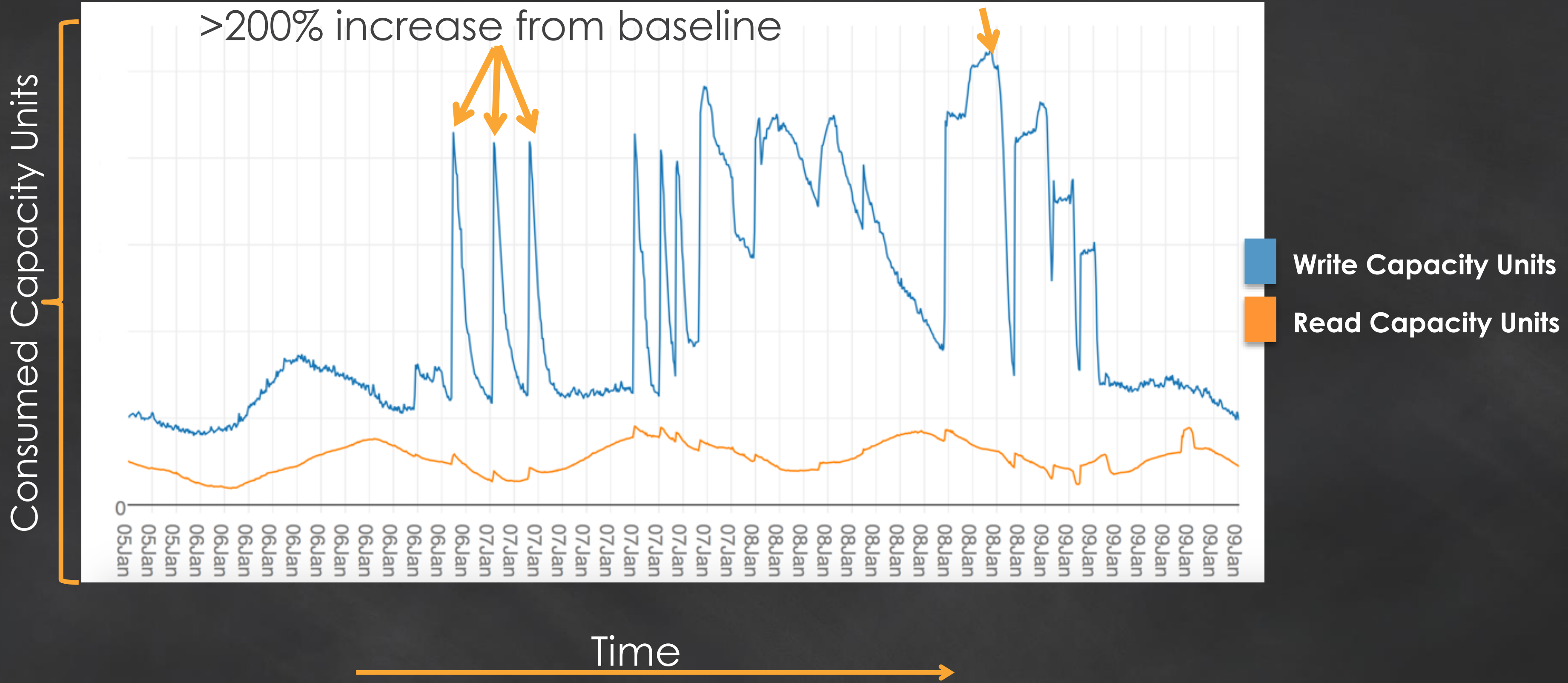
Consistent, single digit millisecond latency at any scale

Fully managed

Secure

Integrates with AWS Lambda, Amazon Redshift, and more.

Elastic is the new normal



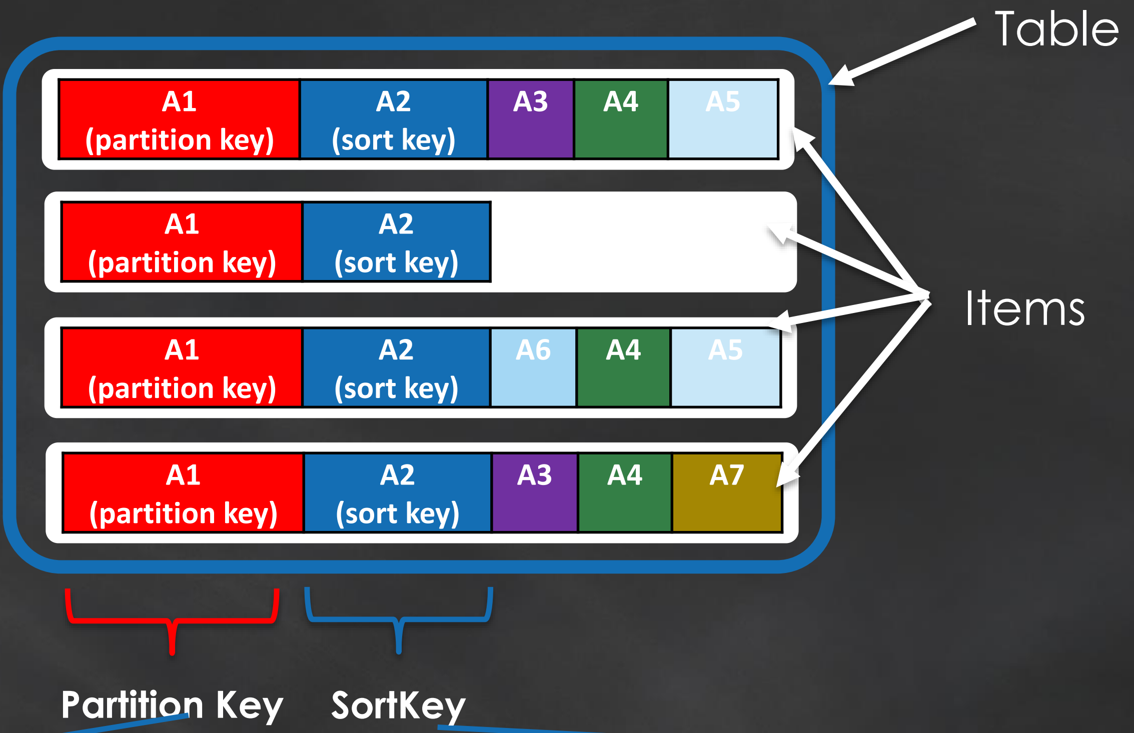
>300% increase from baseline

>200% increase from baseline

Scaling high-velocity use cases with DynamoDB

Ad Tech	Gaming	IoT	Mobile	Web
has offers by TUNE	SUPERCELL	mlbam	duolingo	Expedia
DataXu	zynga	ACTi Connecting Vision	Mapbox	Adobe
ADBRAIN	NEXON	canary	REDFIN	JustGiving
doapp we do cool stuff	FanDuel	dropcam	remind	jobandtalent
jampp	FRONTIER	MEDIATEK	INFRAWARE	amazon marketplace

DynamoDB Table

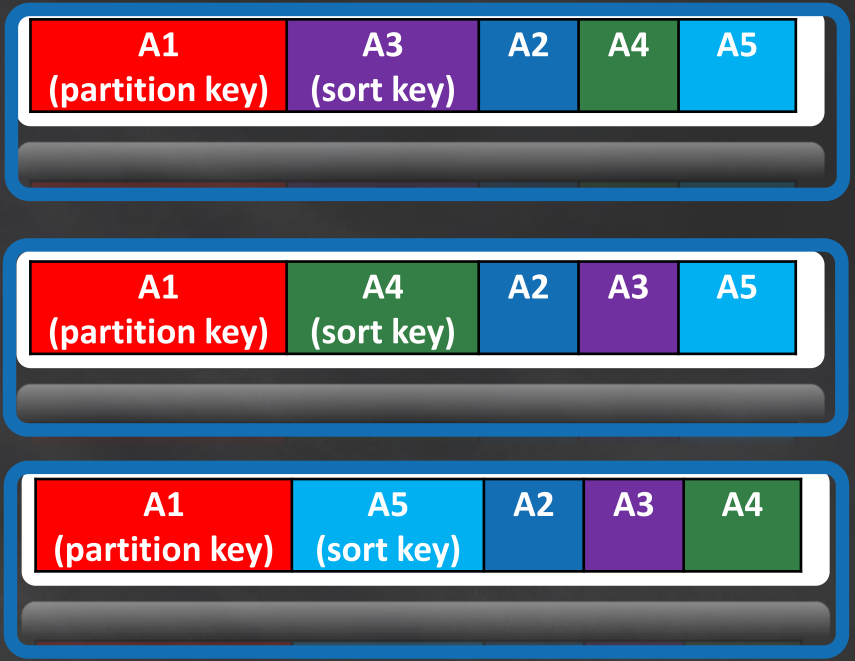


Mandatory
Key-value access pattern
Determines data distribution

Optional
Model 1:N relationships
Enables rich query capabilities

Local Secondary Indexes

- Alternate sort key attribute
- Index is local to a partition key



A1 (partition key)	A3 (sort key)	A2	A4	A5

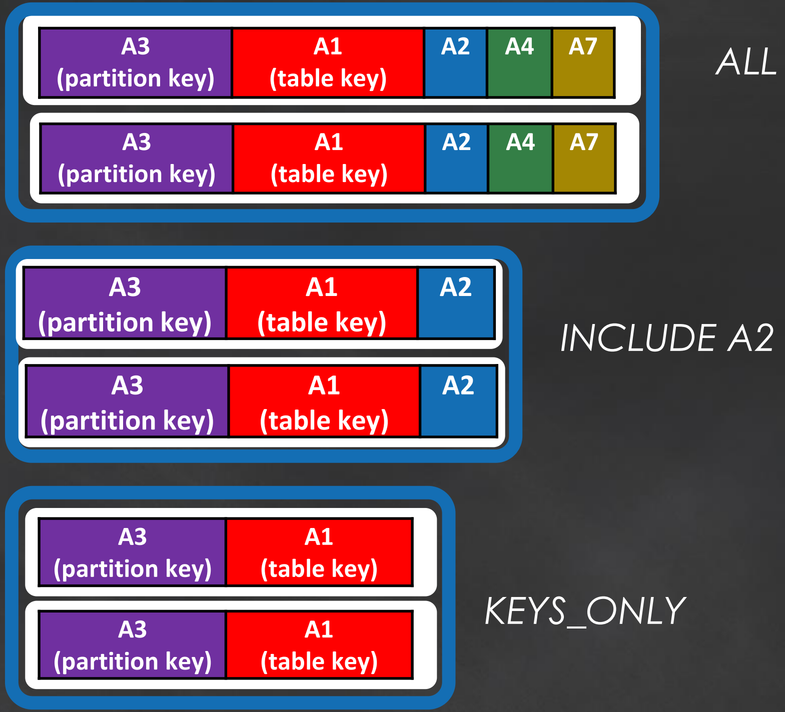
A1 (partition key)	A4 (sort key)	A2	A3	A5

A1 (partition key)	A5 (sort key)	A2	A3	A4

10 GB max per partition key, i.e. LSIs limit the # of sort keys!

Global Secondary Indexes

- Alternate partition (+sort) key
- Index is across all table partition keys
- Can be added or removed anytime

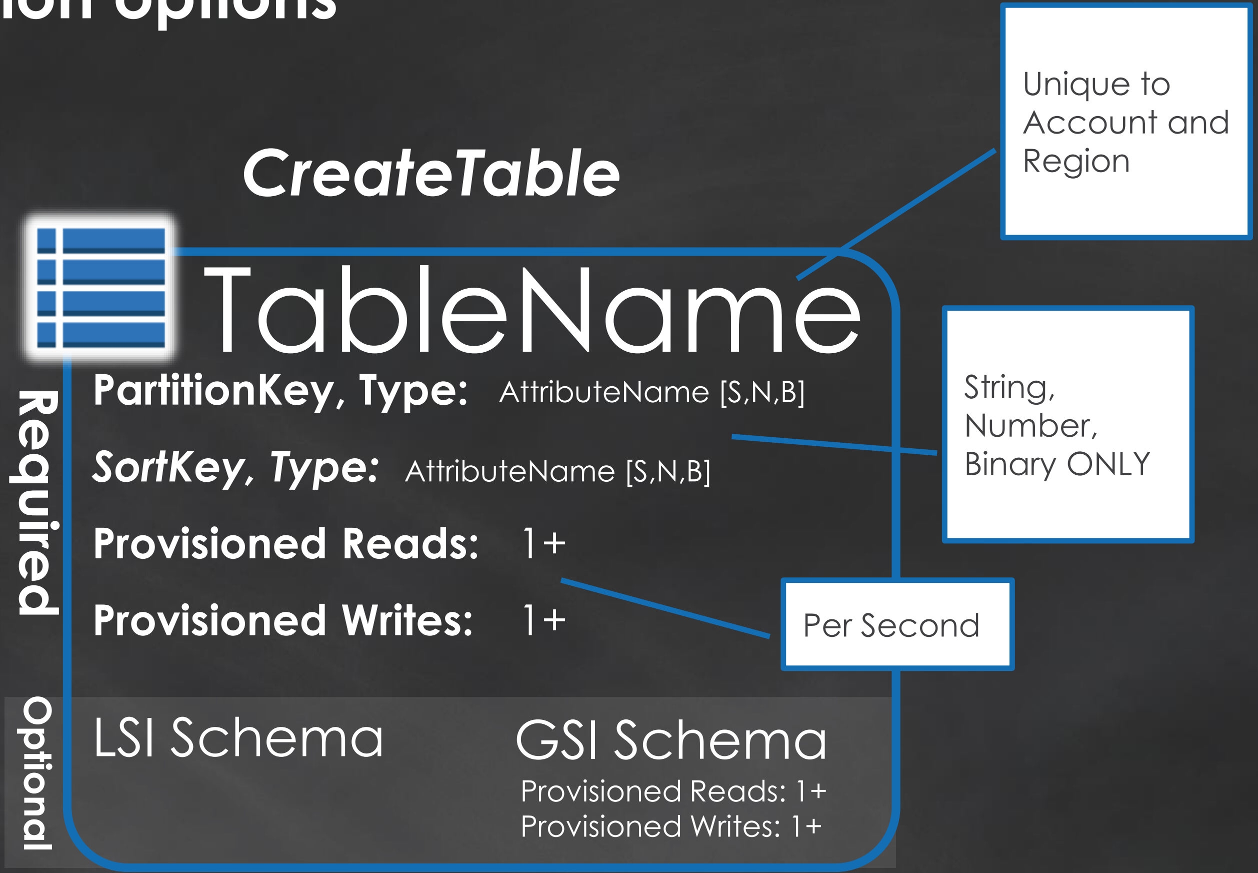


RCUs/WCUs provisioned separately for GSIs

Data types

Type	DynamoDB Type
String	String
Integer, Float	Number
Timestamp	Number or String
Blob	Binary
Boolean	Bool
Null	Null
List	List
Set	Set of String, Number, or Binary
Map	Map

Table creation options



CreateTable

TableName

Unique to Account and Region

Required

PartitionKey, Type: AttributeName [S,N,B]

SortKey, Type: AttributeName [S,N,B]

String, Number, Binary ONLY

Provisioned Reads: 1+

Provisioned Writes: 1+

Per Second

Optional

LSI Schema

GSI Schema

Provisioned Reads: 1+
Provisioned Writes: 1+

Provisioned capacity

Provisioned capacity

Capacity is per second, rounded up to the next whole number

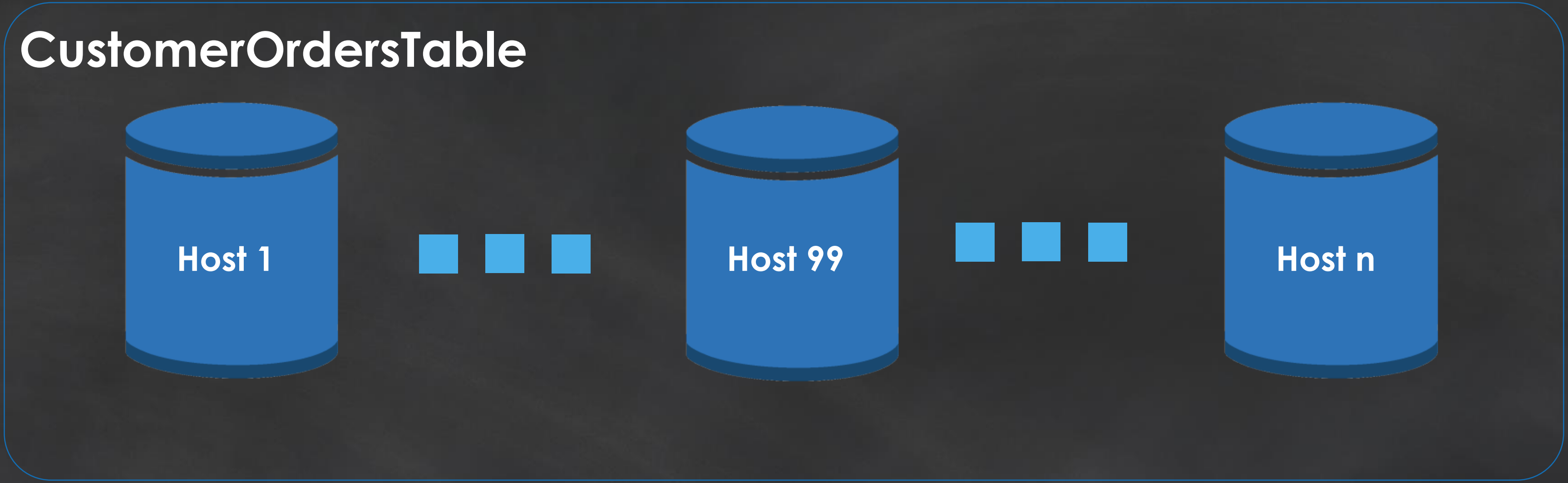
Read Capacity Unit (RCU)

1 RCU returns 4KB of data for strongly consistent reads, or double the data at the same cost for eventually consistent reads

Write Capacity Unit (WCU)

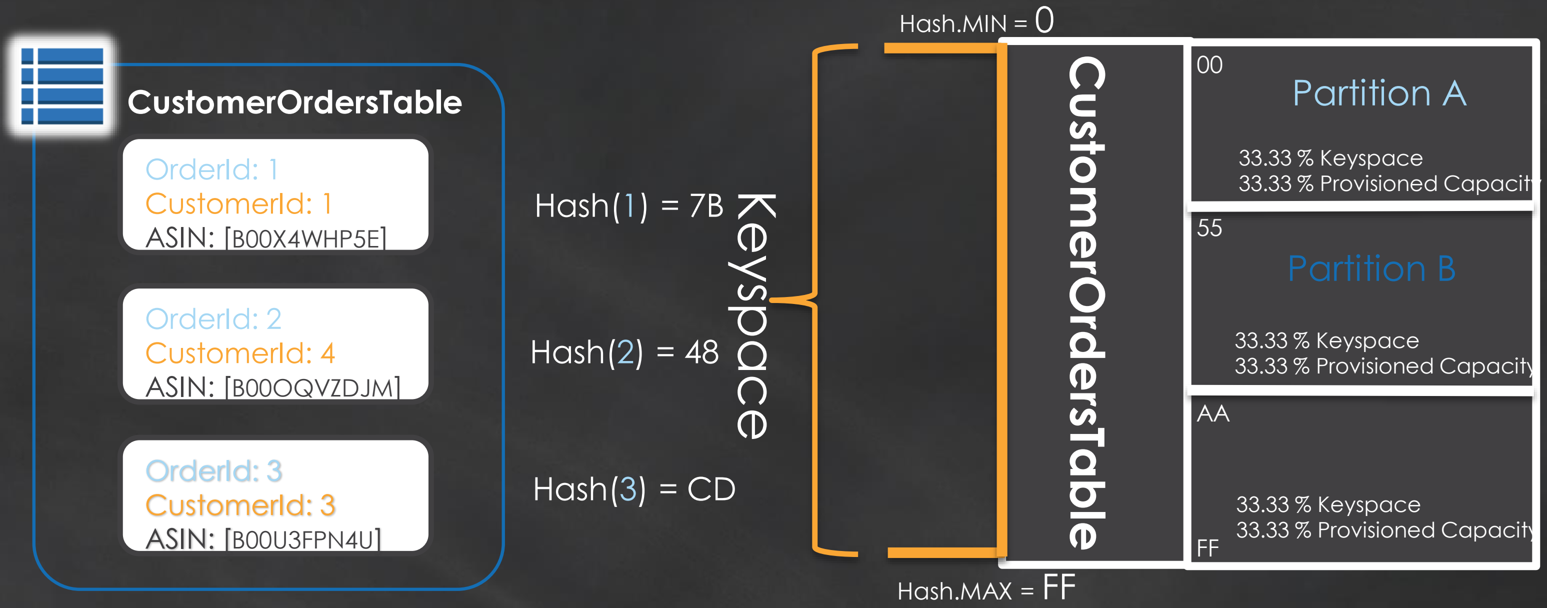
1 WCU writes 1KB of data, and *each item consumes 1 WCU minimum*

Horizontal Sharding

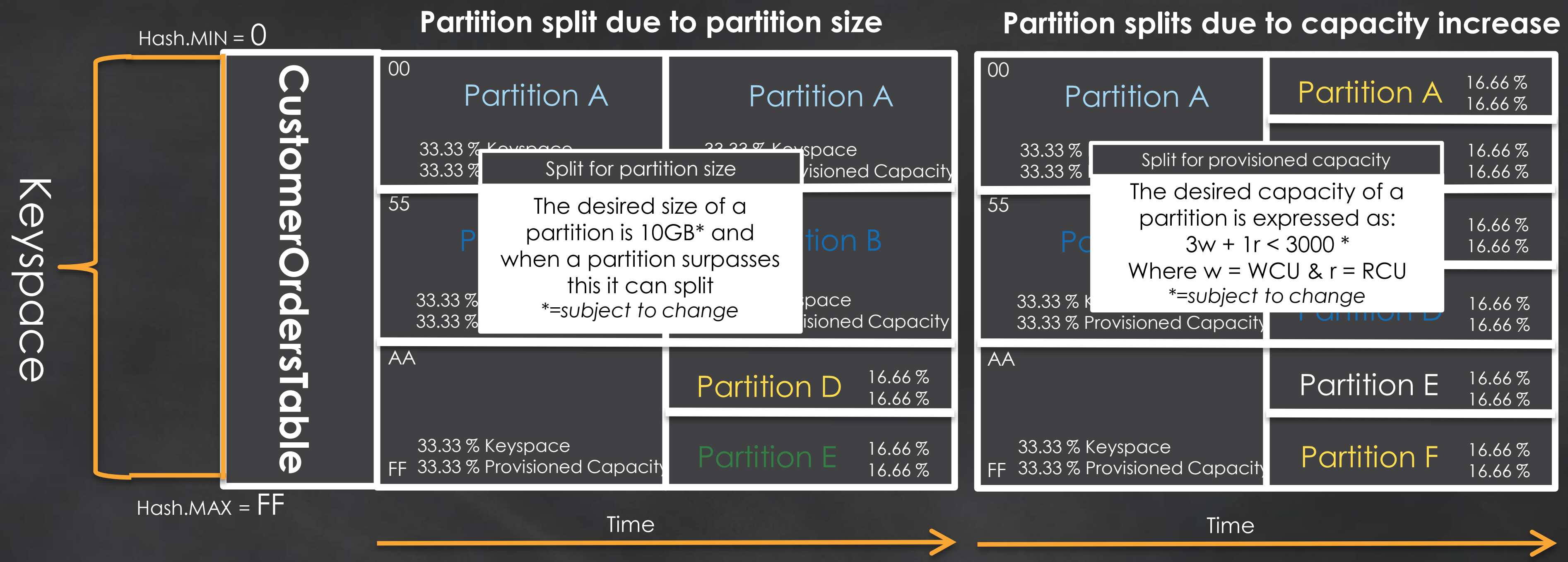


~Each new host brings compute, storage and network bandwidth~

Partitioning



Partitioning



Partition split due to partition size

Split for partition size

The desired size of a partition is 10GB* and when a partition surpasses this it can split
=subject to change

Partition splits due to capacity increase

Split for provisioned capacity

The desired capacity of a partition is expressed as:
3w + 1r < 3000 *
Where w = WCU & r = RCU
=subject to change

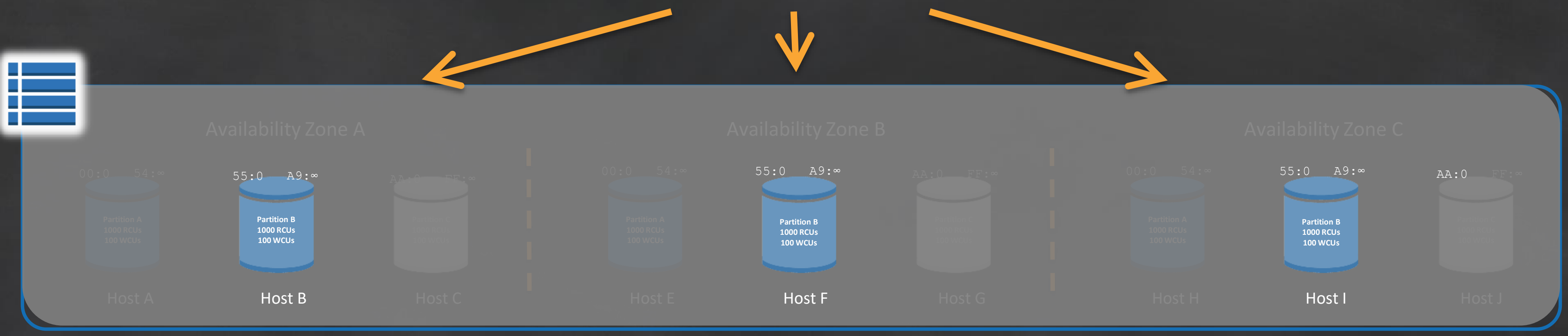
Partitioning

3-way replication

Data is replicated to three Availability Zones by design

OrderId: 1
CustomerId: 1
ASIN: [B00X4WHP5E]

Hash(1) = 7B



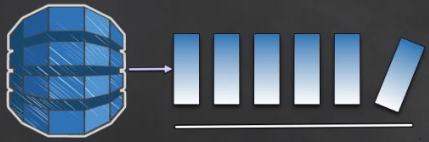
CustomerOrdersTable

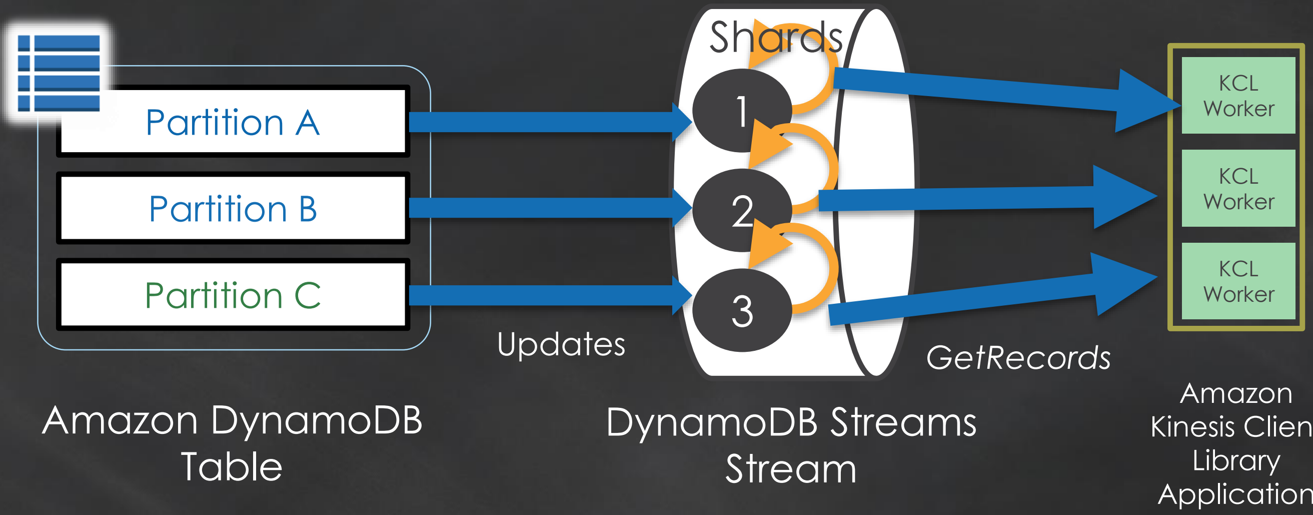
DynamoDB Streams

Shards have a lineage and automatically close after time or when the associated DynamoDB partition splits

DynamoDB Streams

- ✓Ordered stream of item changes
- ✓Exactly once, strictly ordered by key
- ✓Highly durable, scalable
- ✓24 hour retention
- ✓Sub-second latency
- ✓Compatible with Kinesis Client Library



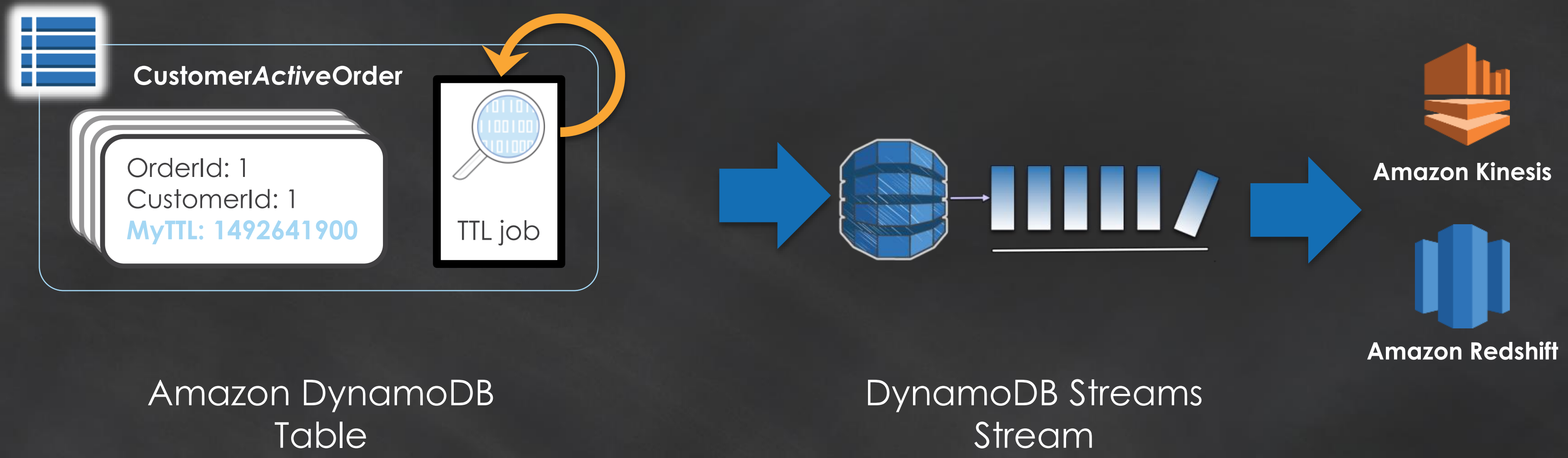


Time-To-Live (TTL)

Removes data that is no longer relevant

Time-To-Live

An epoch timestamp marking when an item can be deleted by a background process, without consuming any provisioned capacity



Time-To-Live (TTL)

✓ TTL items identifiable in DynamoDB Streams

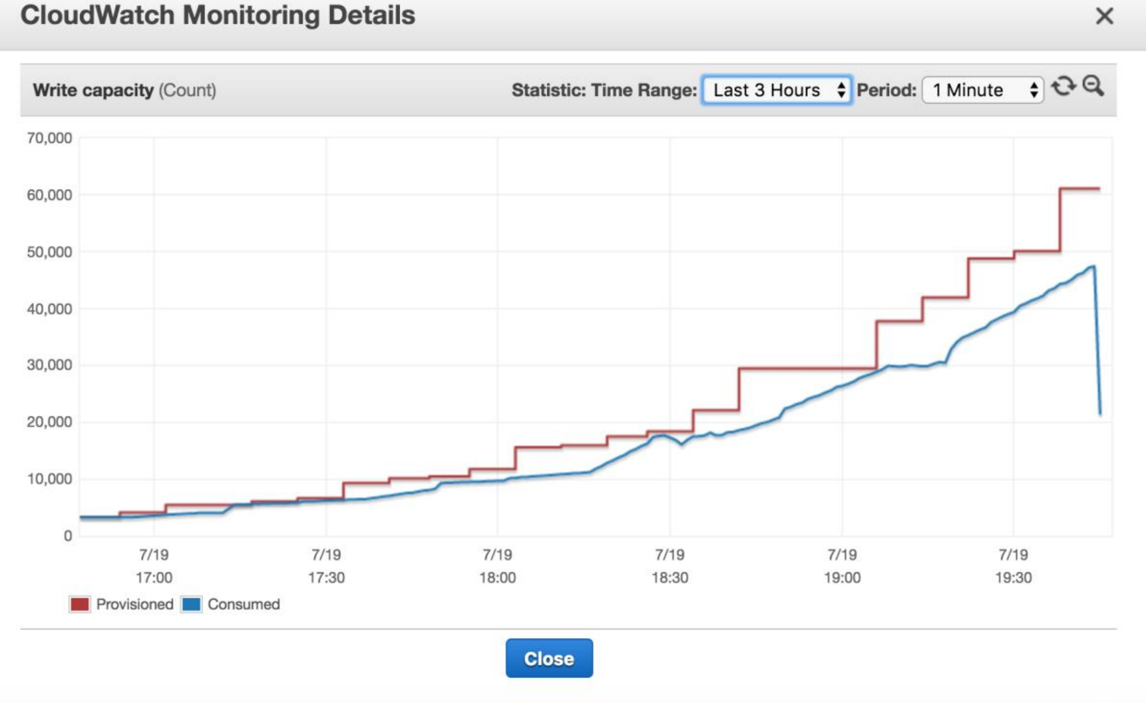
✓ Configuration protected by AWS Identity and Access Management (IAM), auditable with AWS CloudTrail

✓ Eventual deletion, free to use

DynamoDB Auto Scaling

New

Specify: 1) Target capacity in percent 2) Upper and lower bound



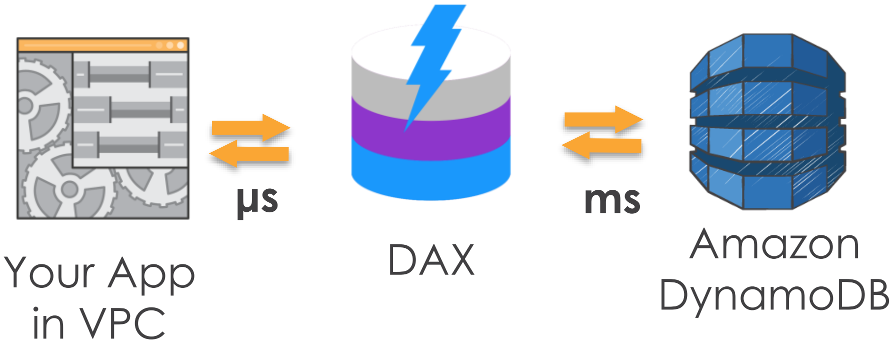
CloudWatch Monitoring Details

Write capacity (Count)

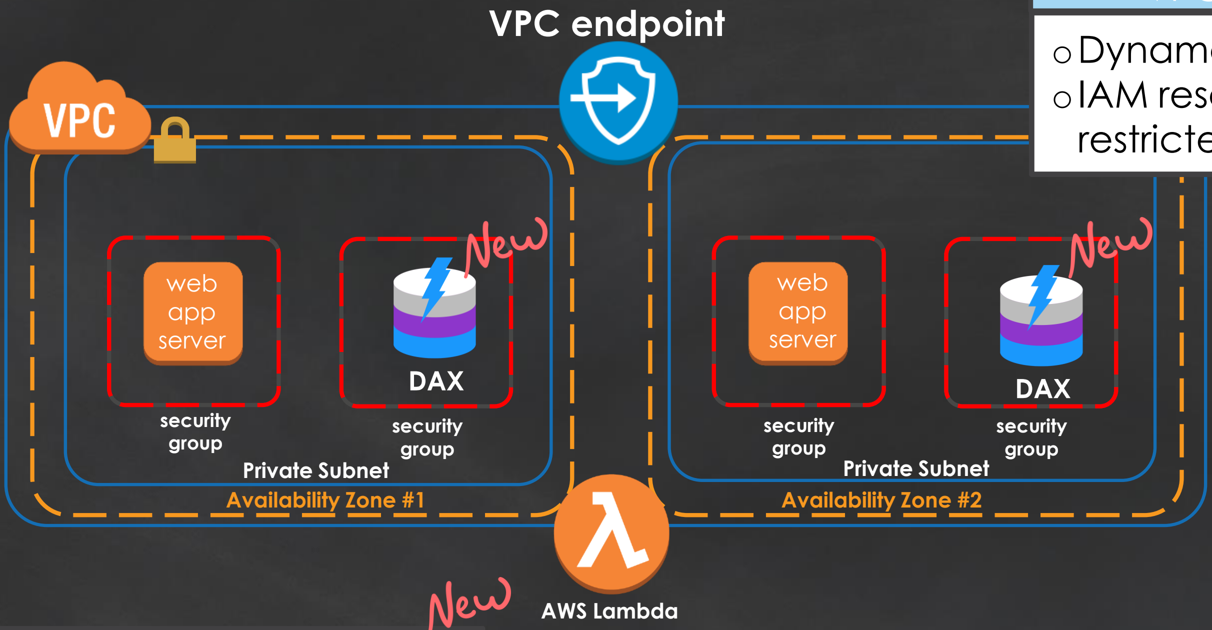
Statistic: Time Range: Last 3 Hours Period: 1 Minute

New

Amazon DynamoDB Accelerator (DAX)



DynamoDB in the VPC



New

VPC Endpoints

- DynamoDB-in-the-VPC
- IAM resource policy restricted

New

DAX

- Microseconds latency in-memory cache
- Millions of requests per second
- Fully managed, highly available
- Role based access control
- No IGW or VPC endpoint required

DynamoDB Accelerator (DAX)

New

Private IP, Client-side Discovery

Supports AWS Java SDK on launch, with more AWS SDKs to come

Cluster based, Multi-AZ

Separate Query and Item cache

Elements of even access in NoSQL

1) Time

2) **SPACE**

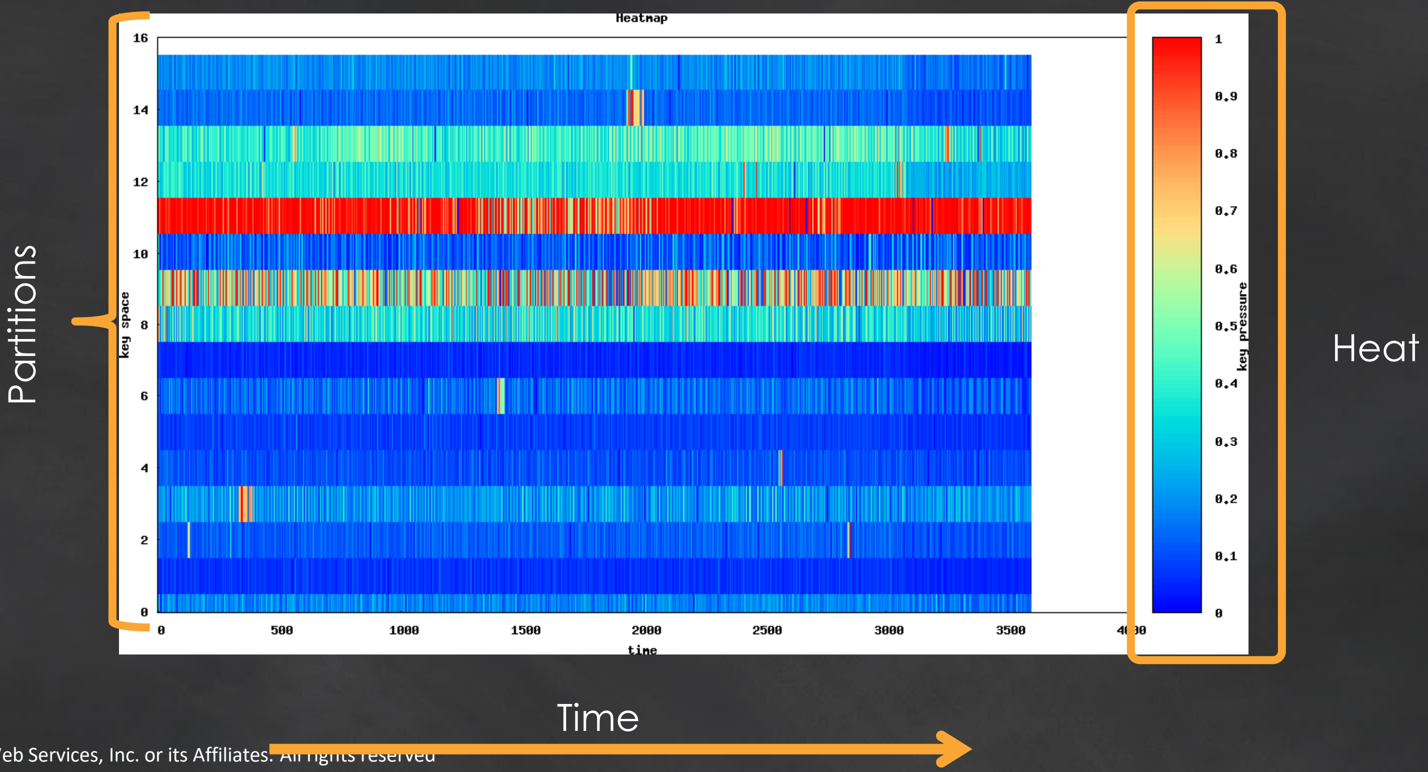
DynamoDB key choice

Amazon DynamoDB Developer Guide

> *To get the most out of DynamoDB throughput, create tables where the partition key has a large number of distinct values, and values are requested fairly uniformly, as randomly as possible.*

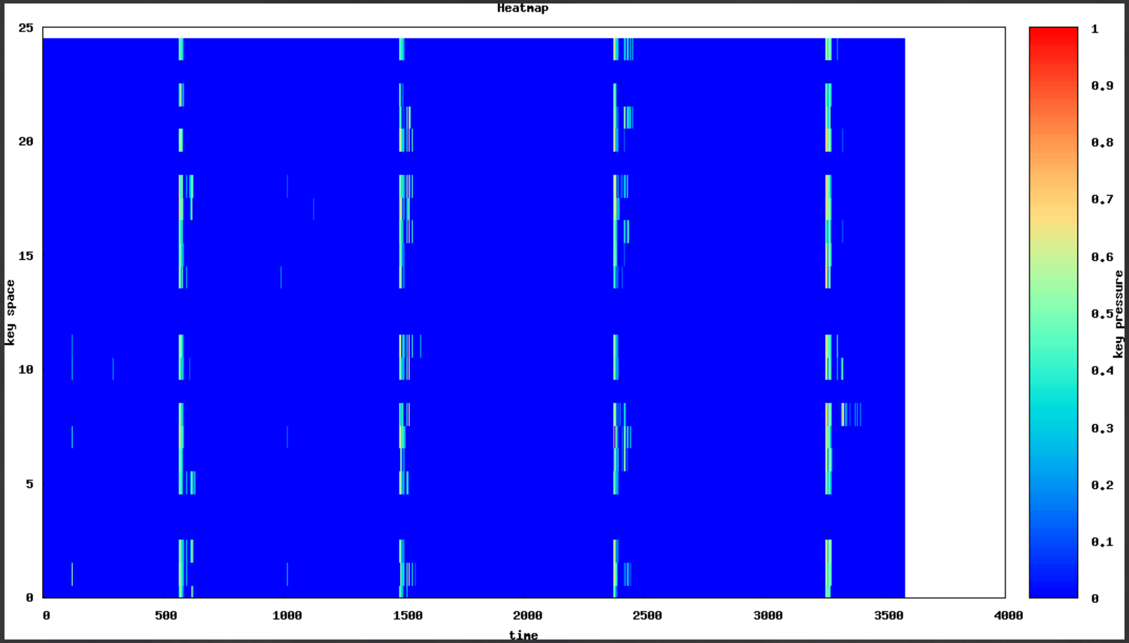
Elements of even access

1. Key choice: high key cardinality
2. Uniform access: access is evenly spread over the key-space



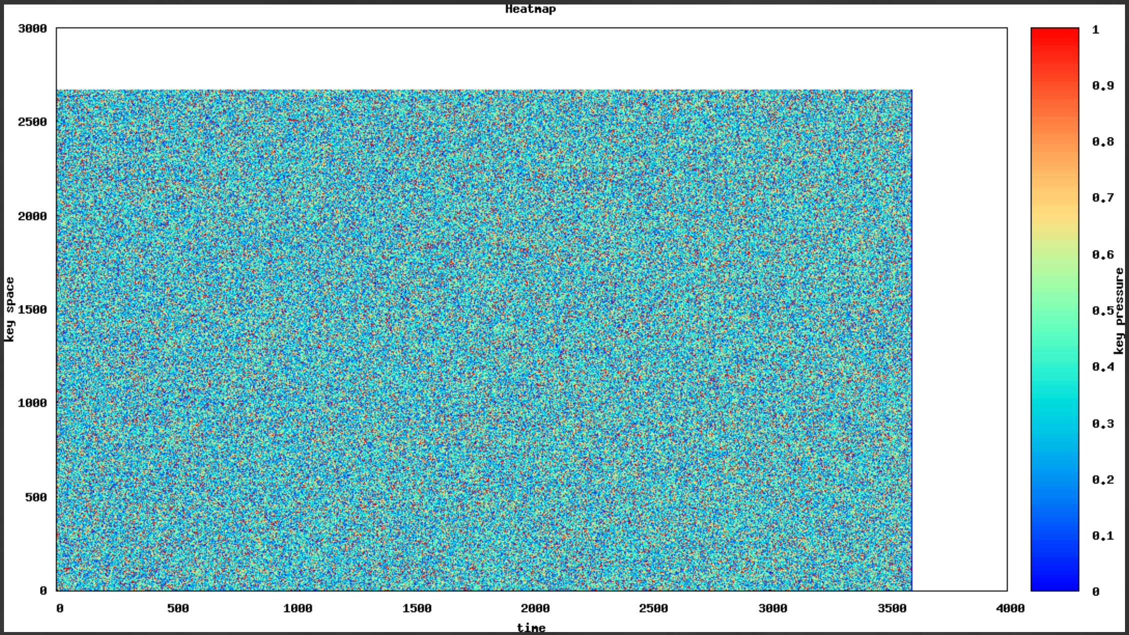
Elements of even access

3.Requests arrive evenly spaced in time



Elements of even access

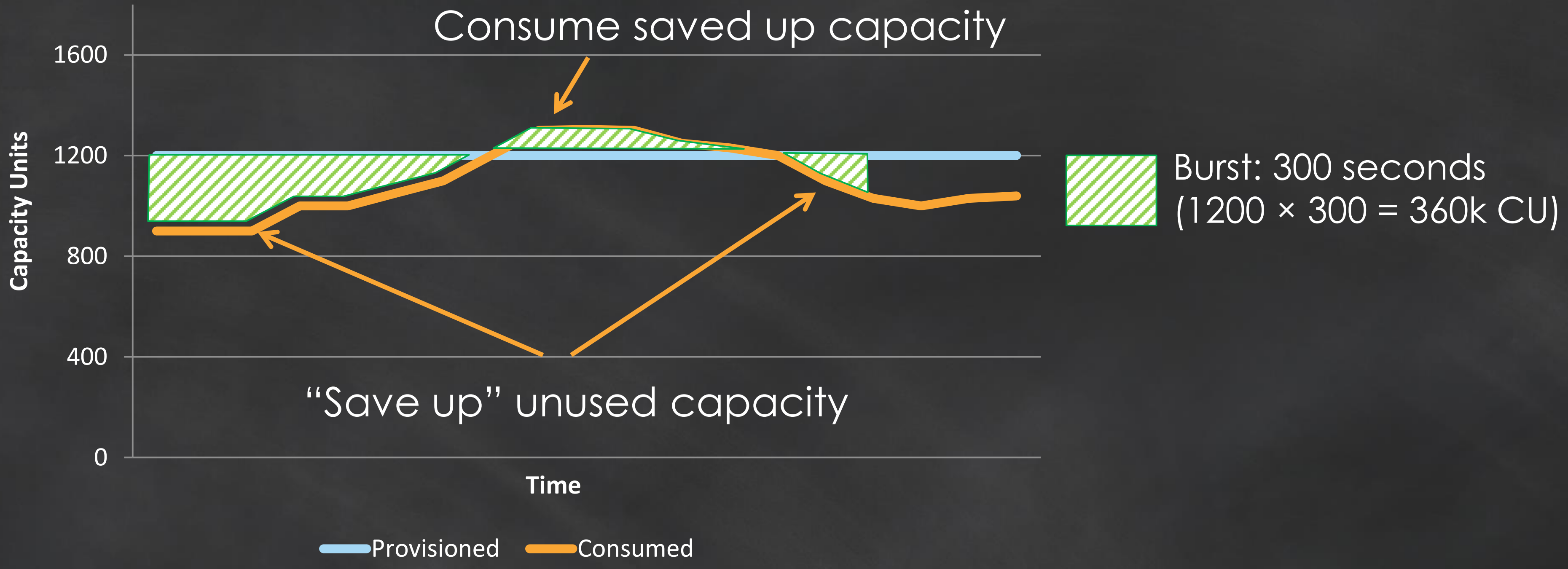
Even access: All three at once



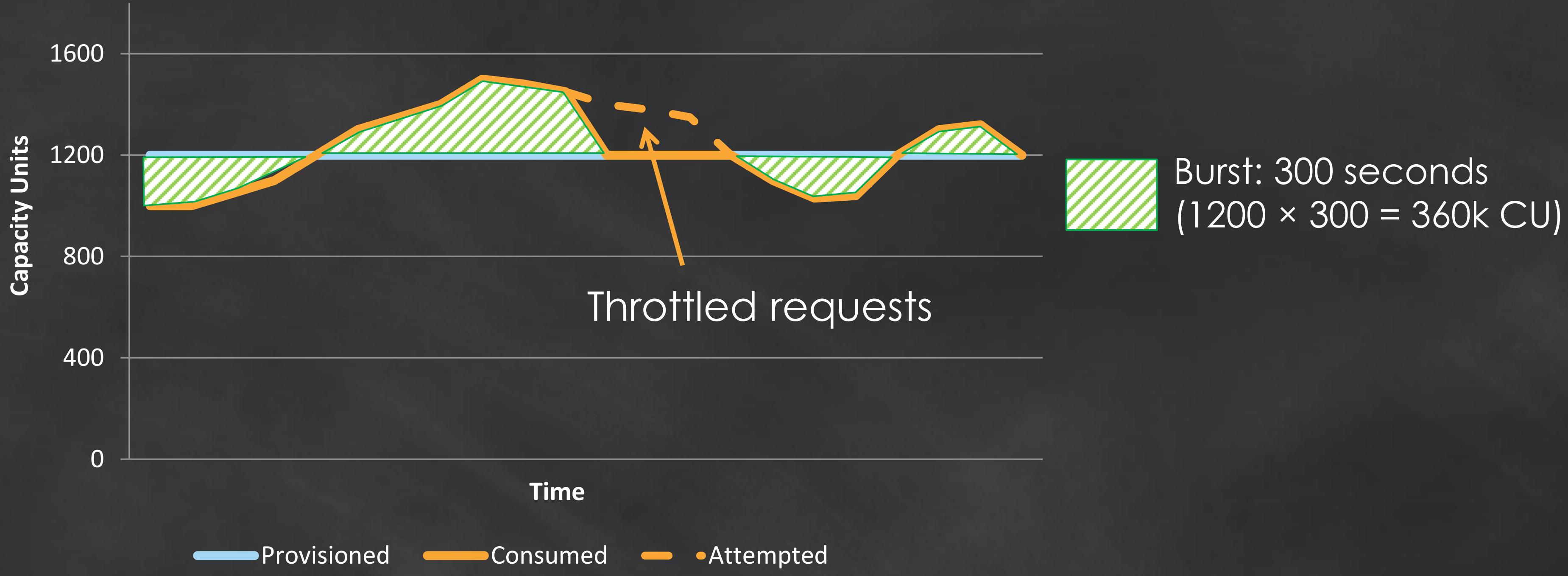
Time

Burst capacity is built-in

DynamoDB "saves" 300 seconds of unused capacity per partition

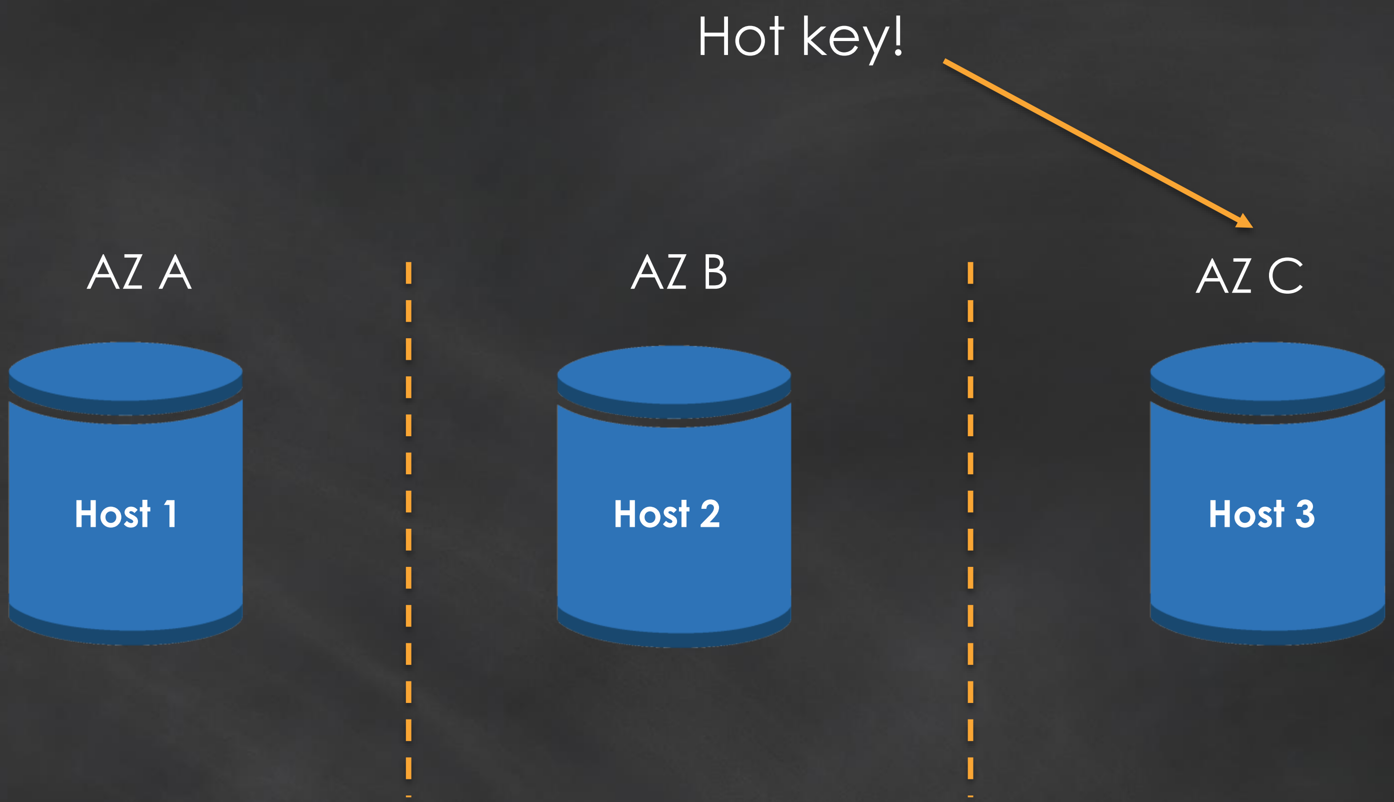


Burst capacity may not be sufficient

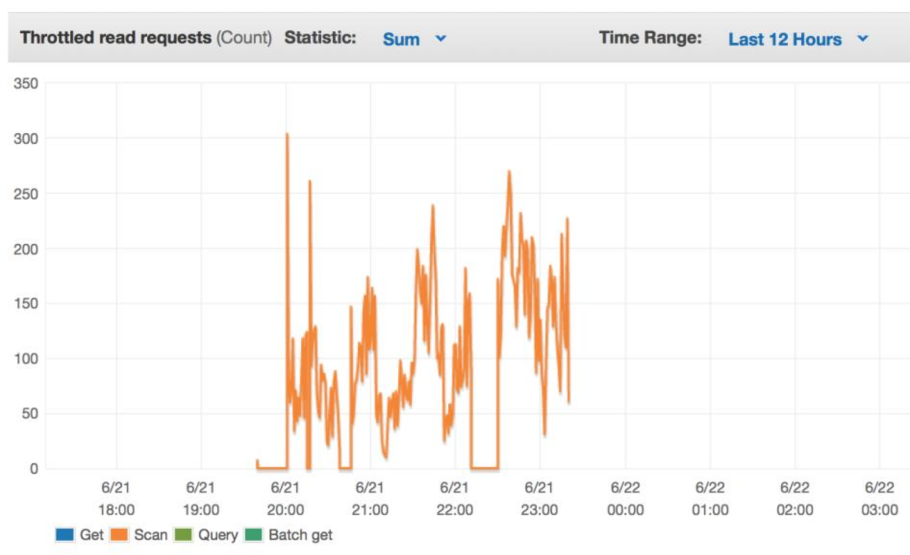


Don’t completely depend on burst capacity… provision sufficient throughput

Hot shards



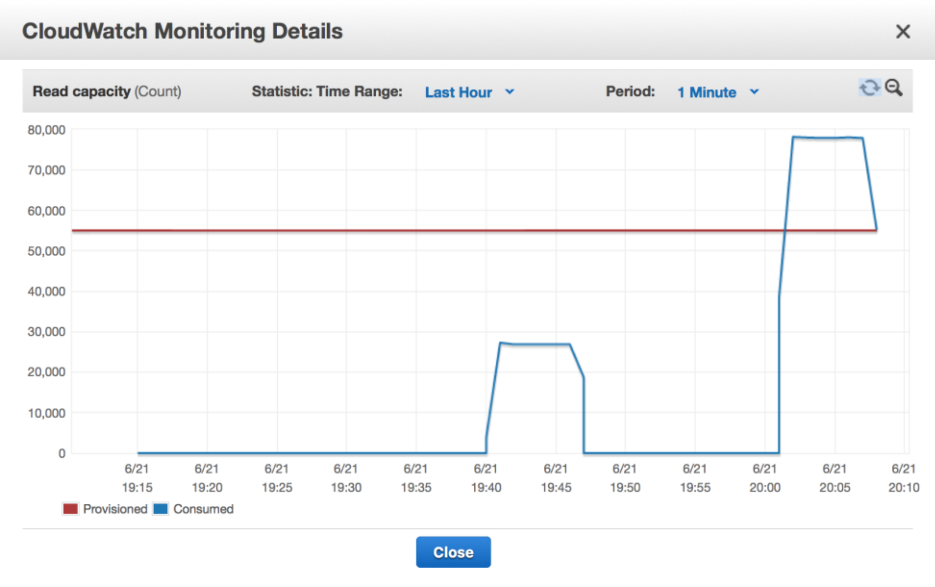
What causes throttling?



A throttle comes from a partition

If **sustained** throughput goes beyond provisioned throughput on a partition

What causes throttling?



In Amazon CloudWatch, if consumed capacity is well under provisioned and throttling occurs, it must be “partition throttling”

If **sustained** throughput goes beyond provisioned throughput on a partition

What causes throttling?

Top Items

- Fire TV Stick
- Echo Dot – Black
- Amazon Fire TV
- Amazon Echo – Black
- Fire HD 8
- Echo Dot – White
- Kindle Paperwhite
- Fire Tablet with Alexa
- Fire HD 8 Tablet with A…
- Fire HD 8 Tablet with A…

Disable retries, writes your own retry code, and log all **throttled or returned keys**

If **sustained** throughput goes beyond provisioned throughput on a partition

Design Patterns

DESIGN PATTERNS:
DynamoDB Accelerator and GSIs

Dating Website

- ✓ Online dating website running on AWS
- ✓ Users have people they like, and conversely people who like them
- ✓ Hourly batch job matches users
- ✓ Data stored in Likes and Matches tables

Schema Design Part 1

Likes

user_id_self (Partition key)	user_id_other (sort key)	MyTTL (TTL attribute)	…	Attribute N

LIKES |

Requirements:

1. Get all people I like
2. Get all people that like me
3. Expire likes after 90 days

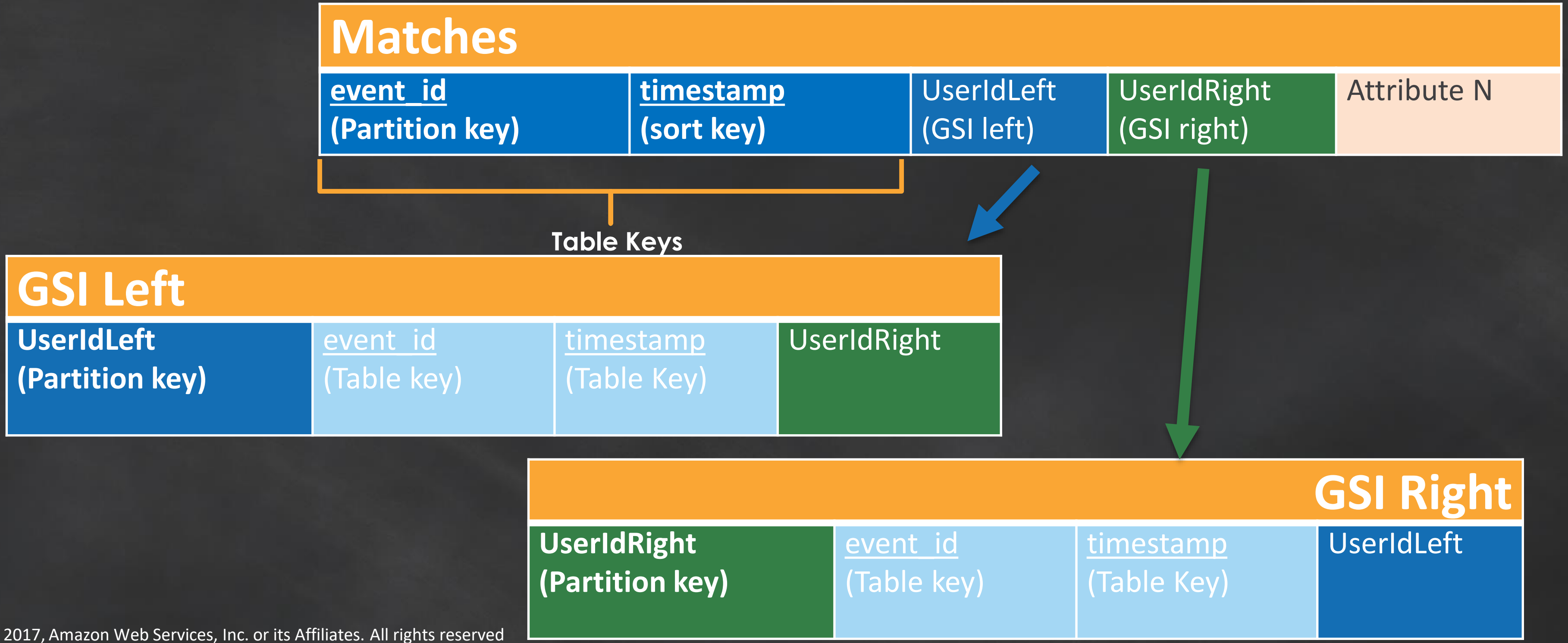
GSI_Other

user_id_other (Partition key)	user_id_self (sort key)

Schema Design Part 2

MATCHES | Requirements:

1.Get my matches



Matches

event_id **(Partition key)**	**timestamp** **(sort key)**	UserIdLeft (GSI left)	UserIdRight (GSI right)	Attribute N

Table Keys

GSI Left

UserIdLeft **(Partition key)**	event_id (Table key)	timestamp (Table Key)	UserIdRight

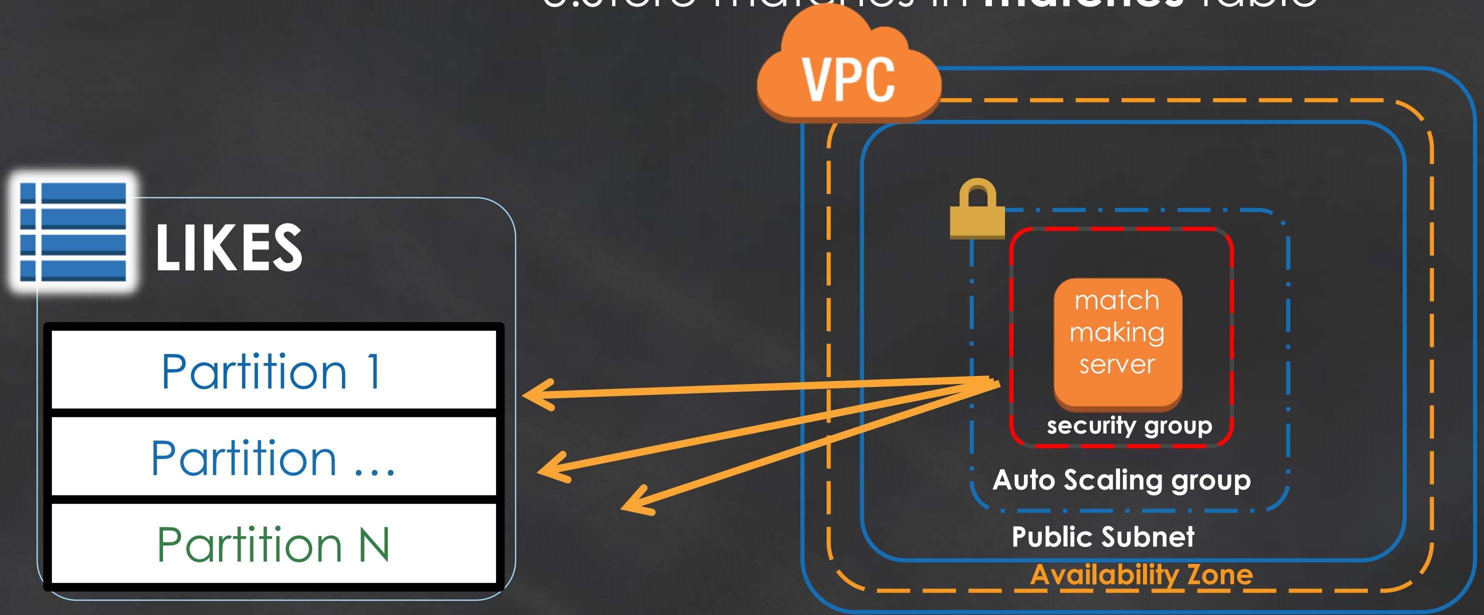
GSI Right

UserIdRight **(Partition key)**	event_id (Table key)	timestamp (Table Key)	UserIdLeft

Matchmaking

Requirements:

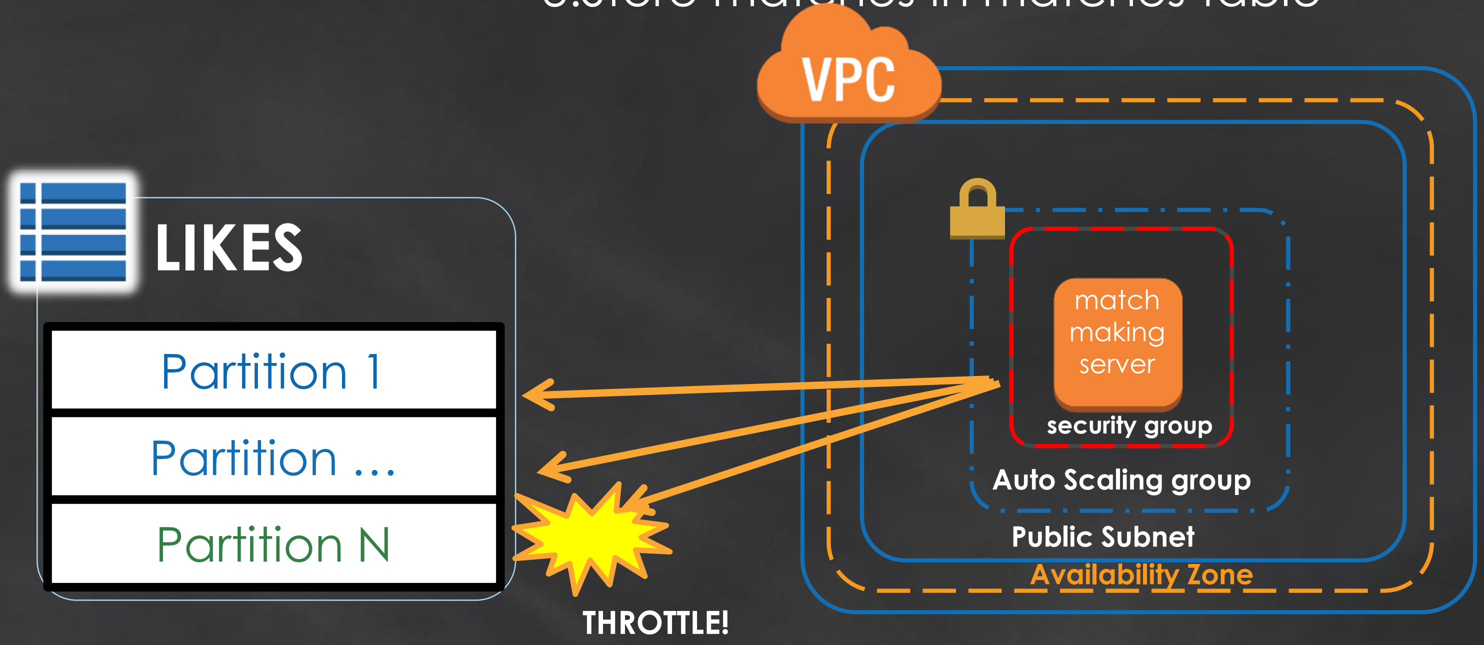
1. Get all new likes every hour
2. For each like, get the other user's likes
3. Store matches in **matches** table



Matchmaking

Requirements:

1. Get all new likes every **hour**
2. For each like, get the other user's likes
3. Store matches in matches table



Matchmaking

Requirements:

1. Get all new likes every **hour**
2. For each like, get the other user's likes
3. Store matches in matches table

Even Access:

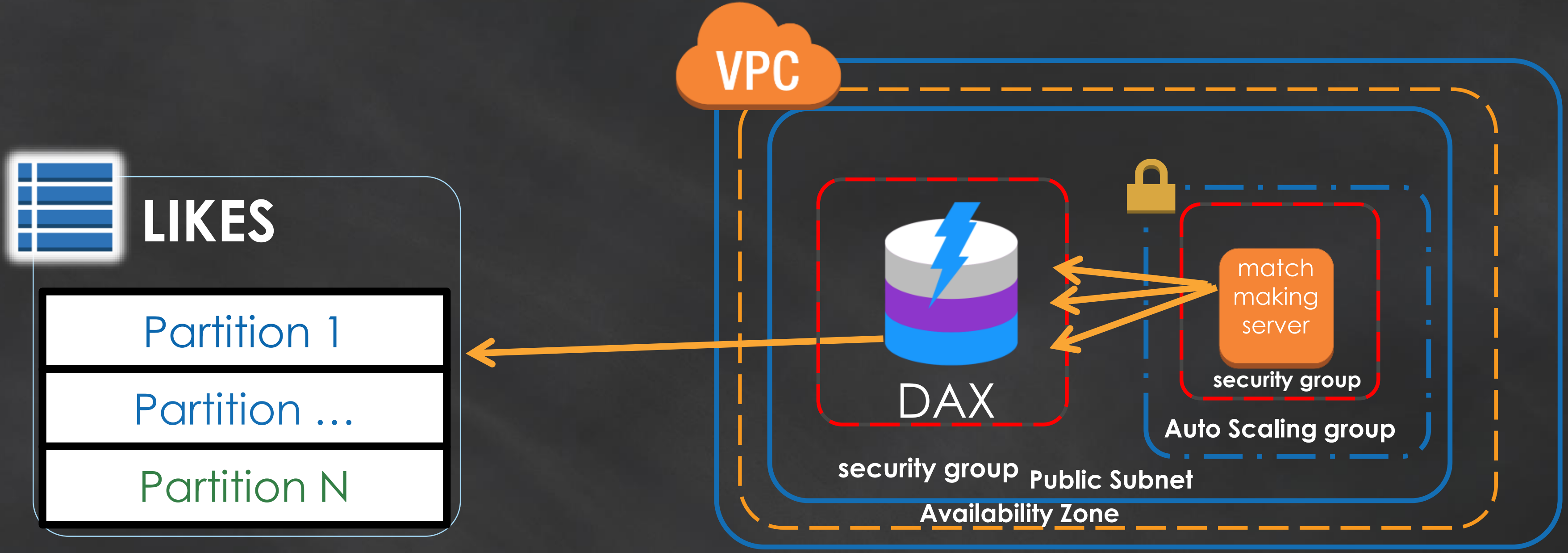
1. Key choice: High key cardinality
2. Uniform access: access is evenly spread over the key-space
3. **Time: requests arrive evenly spaced in time**

Matchmaking

Requirements:

0. Write like to **like** table, then query by user id to *warm* cache, then queue for batch processing

1. Get all new likes every **hour**
2. For each like, get the other user's likes
3. Store matches in matches table



DESIGN PATTERNS:
DynamoDB Accelerator and GSIs

Dating Website

Takeaways:

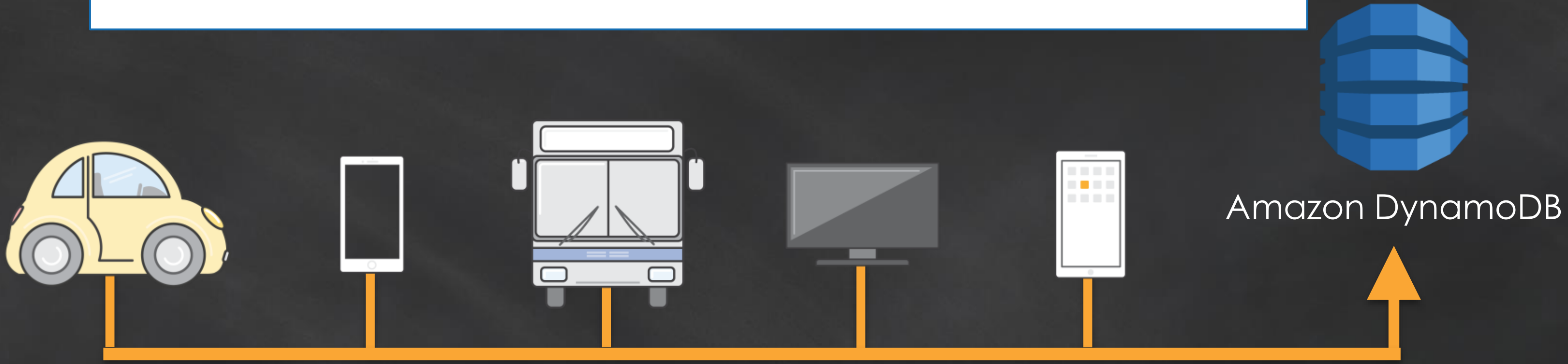
✓Keep DAX warm by querying after writing

✓Use GSIs for many to many relationships

DESIGN PATTERNS:
TTL, DynamoDB Streams, and DAX

Serverless IoT

- ✓Single DynamoDB table for storing sensor data
- ✓Tiered storage to remove archive old events to S3
- ✓Data stored in **data** table

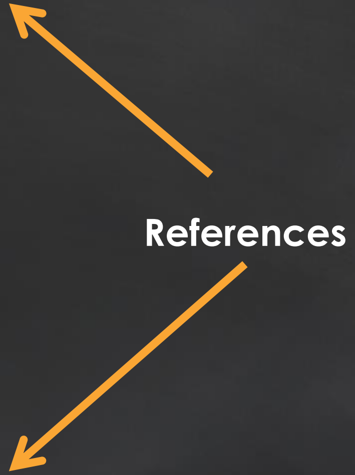


Schema Design

Data				
DeviceId **(Partition key)**	**EventEpoch** **(sort key)**	MyTTL (TTL attribute)	…	Attribute N

DATA | Requirements:

1. Get all events for a device
2. Archive old events after 90 days



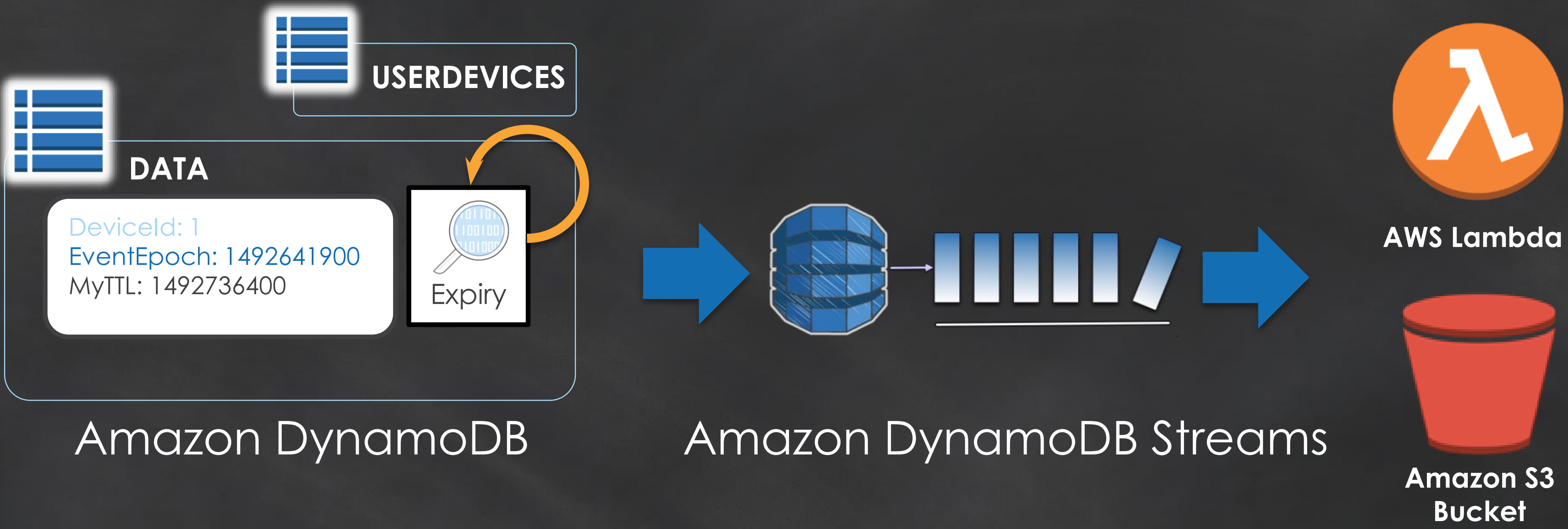
UserDevices				
UserId **(Partition key)**	**DeviceId** **(sort key)**	Attribute 1	…	Attribute N

USERDEVICES | Requirements:

1. Get all devices for a user

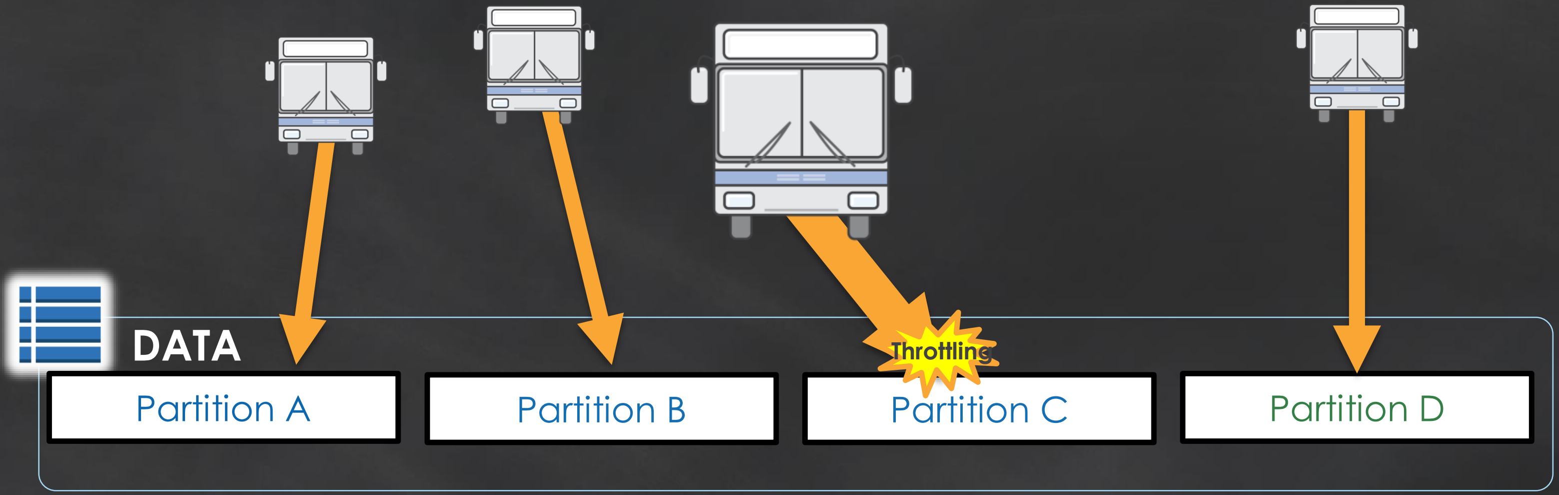
Serverless IoT

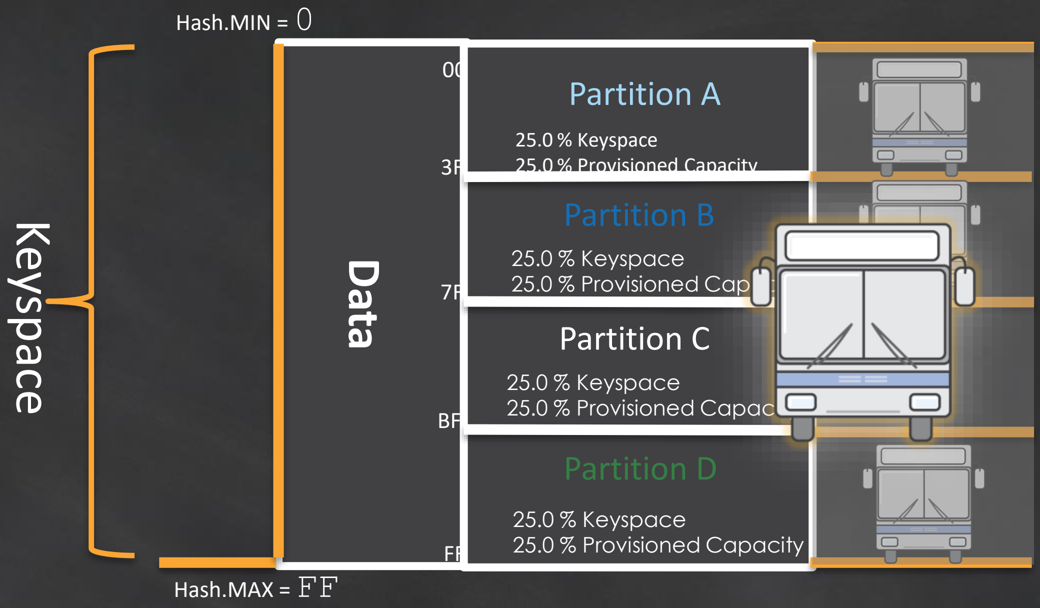
- ✓Single DynamoDB table for storing sensor data
- ✓Tiered storage to remove archive old events to S3
- ✓Data stored in data table



Serverless IoT

Noisy sensor produces data at a rate several times greater than others





Hash.MIN = 0

Keyspace

Data

00

3F

7F

BF

FF

Partition A

25.0 % Keyspace
25.0 % Provisioned Capacity

Partition B

25.0 % Keyspace
25.0 % Provisioned Cap

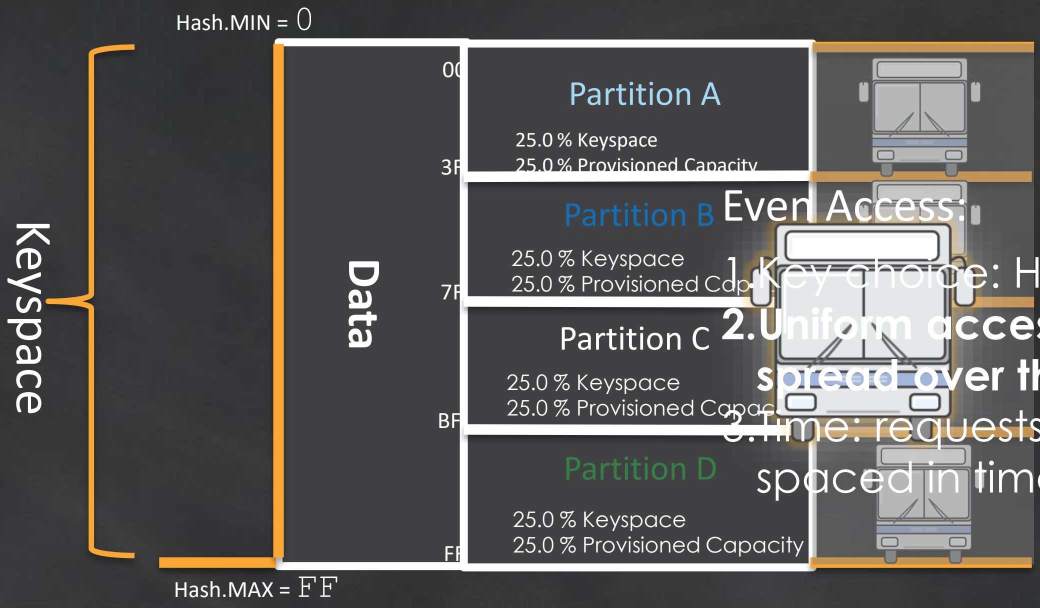
Partition C

25.0 % Keyspace
25.0 % Provisioned Capac

Partition D

25.0 % Keyspace
25.0 % Provisioned Capacity

Hash.MAX = FF



Even Access:

1. Key choice: High key cardinality
2. **Uniform access: access is evenly spread over the key-space**
3. Time: requests arrive evenly spaced in time

Serverless IoT

Requirements:

0. Capable of dynamically sharding to overcome throttling
1. Single DynamoDB table for storing sensor data
2. Tiered storage to remove archive old events to S3
3. Data stored in data table

Schema Design

Naïve Sharding

A sharding scheme where the number of shards is not predefined, and will grow over time but never contract. Contrast with a fixed shard count

Shard

DeviceId (Partition key)	ShardCount Range: 0..1,000

SHARD | Requirements:

1. Get shard count for given device
2. Always grow the count of shards

Data

DeviceId (Partition key)	EventEpoch (sort key)	

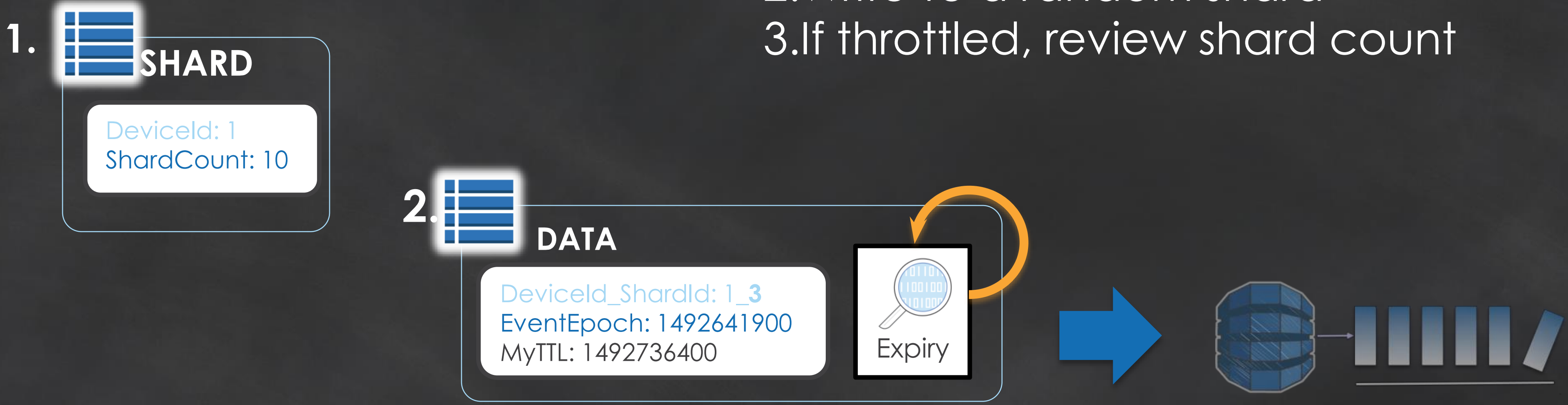
Data | Requirements:

1. Get all events for a device
2. Archive old events after 90 days

Serverless IoT: Naïve Sharding

Request path:

1. Read ShardCount from Shard table
2. Write to a random shard
3. If throttled, review shard count



1. SHARD

DeviceId: 1
ShardCount: 10

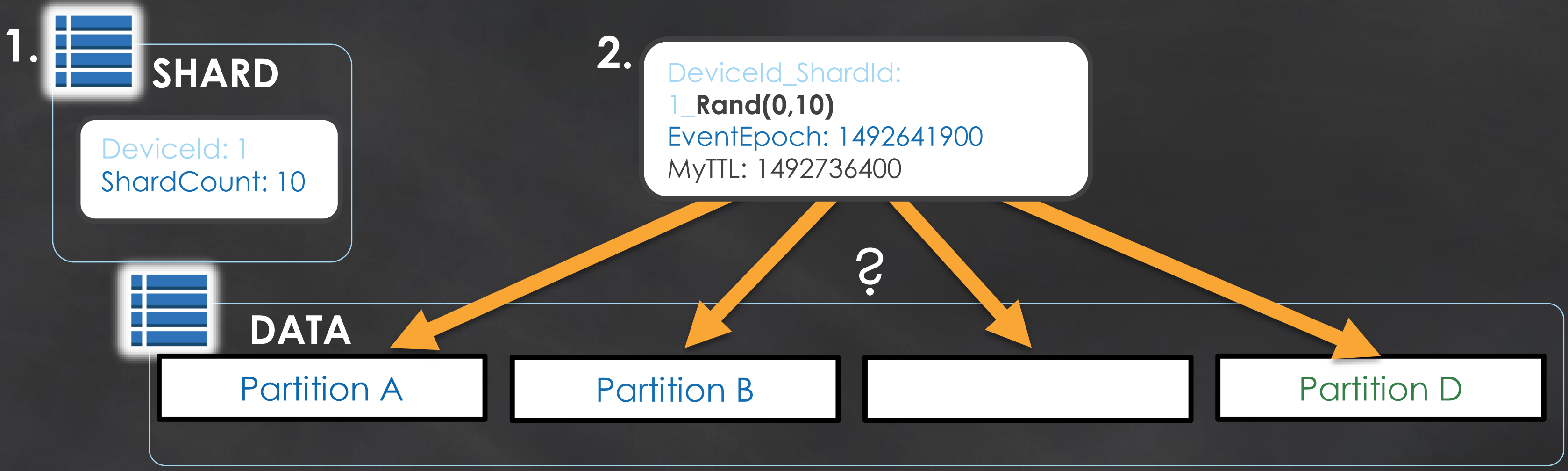
2. DATA

DeviceId_ShardId: 1_**3**
EventEpoch: 1492641900
MyTTL: 1492736400

Expiry

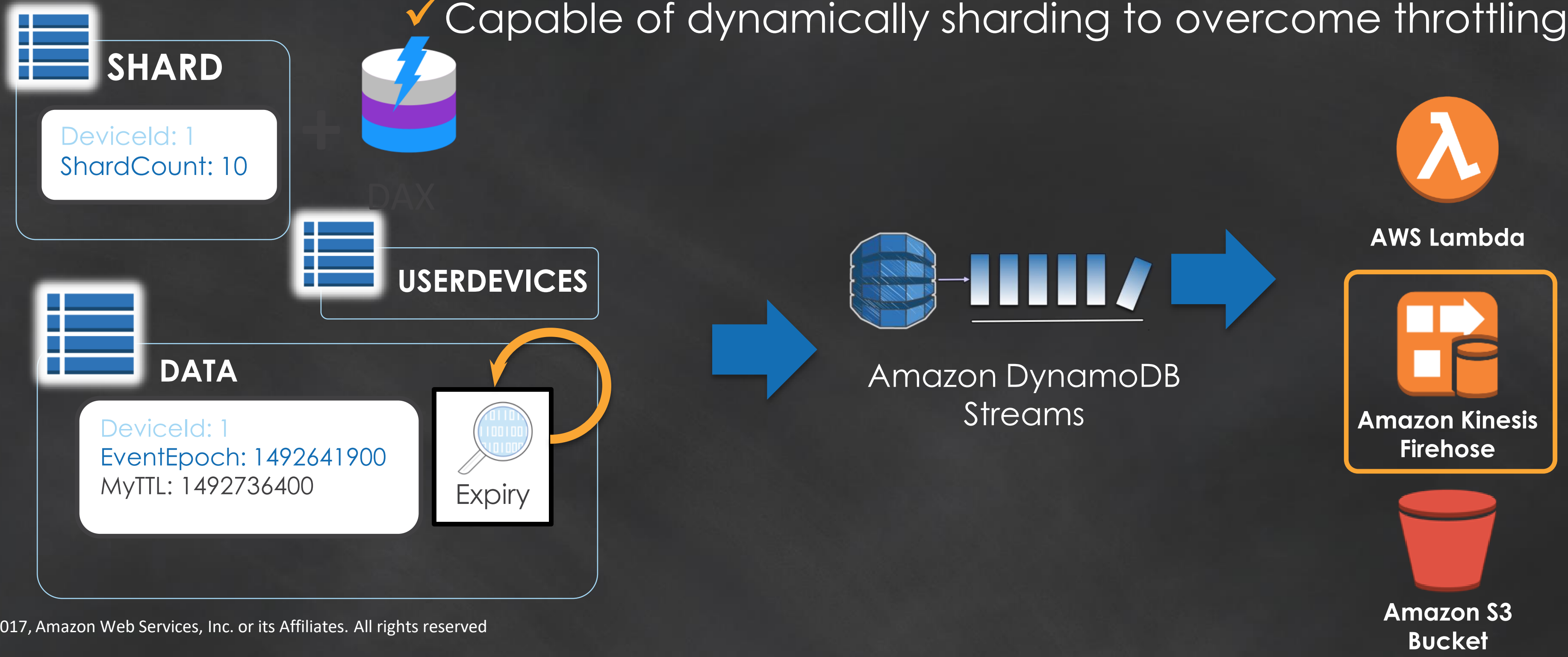
Serverless IoT

Pick a random shard to write data to



Serverless IoT

- ✓Single DynamoDB table for storing sensor data
- ✓Tiered storage to remove archive old events to S3
- ✓Data stored in data table
- ✓Capable of dynamically sharding to overcome throttling



SHARD

DeviceId: 1
ShardCount: 10

DAX

USERDEVICES

DATA

DeviceId: 1
EventEpoch: 1492641900
MyTTL: 1492736400

Expiry

Amazon DynamoDB Streams

AWS Lambda

Amazon Kinesis Firehose

Amazon S3 Bucket

DESIGN PATTERNS:
TTL, DynamoDB Streams, and DAX

Serverless IoT

Takeaways:

✓Use naïve write sharding to dynamically expand shards

✓Use DAX for hot reads, especially from Lambda

✓Use TTL to create tiered storage

Getting started?

DynamoDB Local

Document SDKs

New

DynamoDB as a target for AWS DMS

aws | Pop-up Loft

Thank you!

Remember to fill out your survey