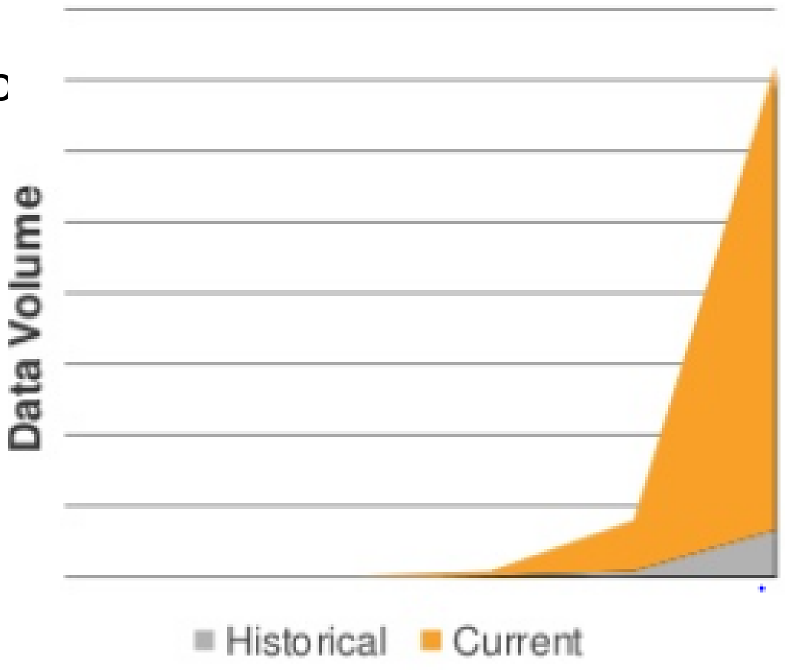
NoSQL Database in AWS
Amazon DynamoDB

Source/Reference:
http://aws.amazon.com/dynamodb

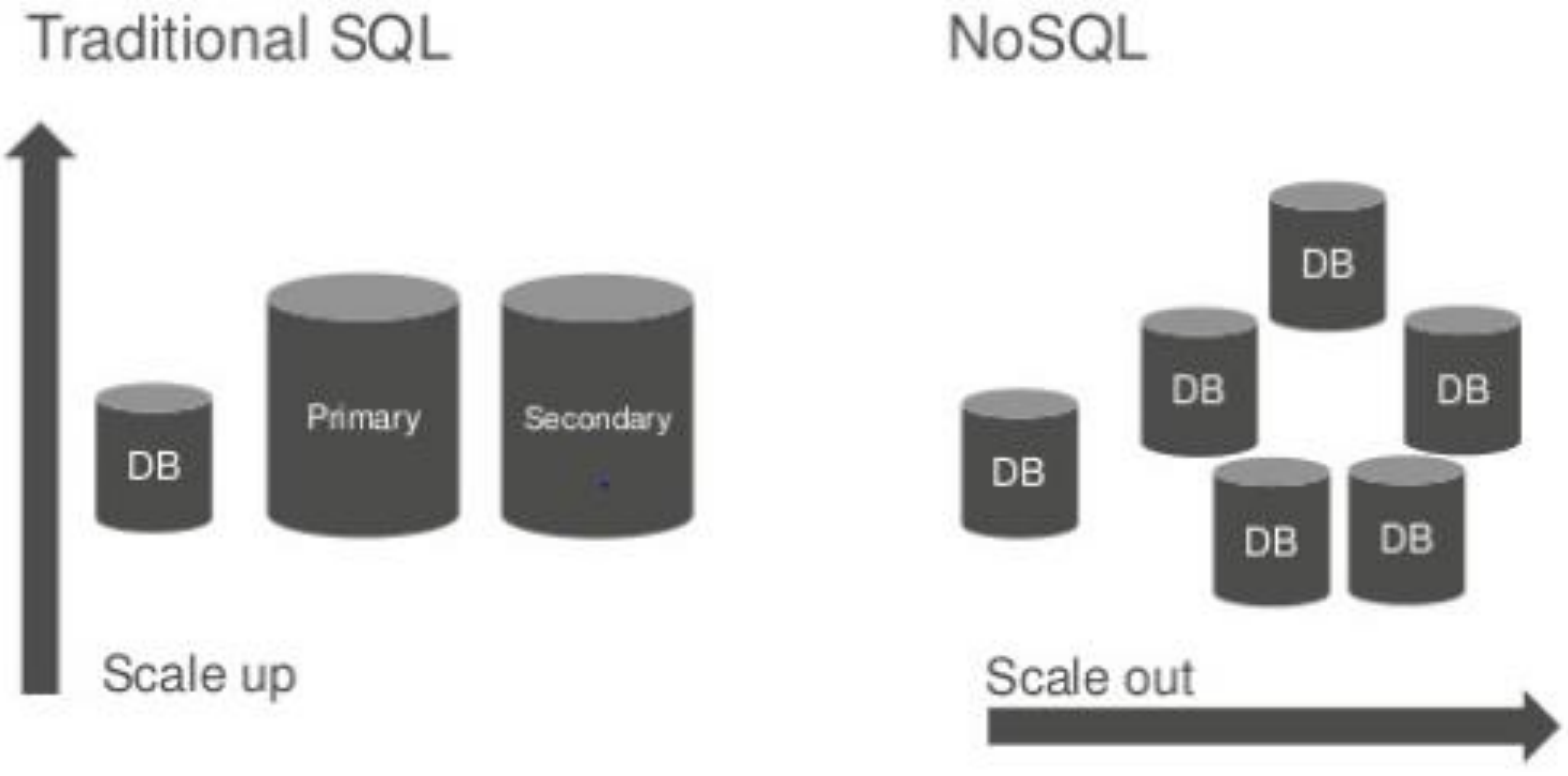
Data Volume since 2011

- 90% of stored data generated in last 2 years
- Petabytes of data is new normal
- No reason these trends will not continue c time
- Need an efficient way to manage this data

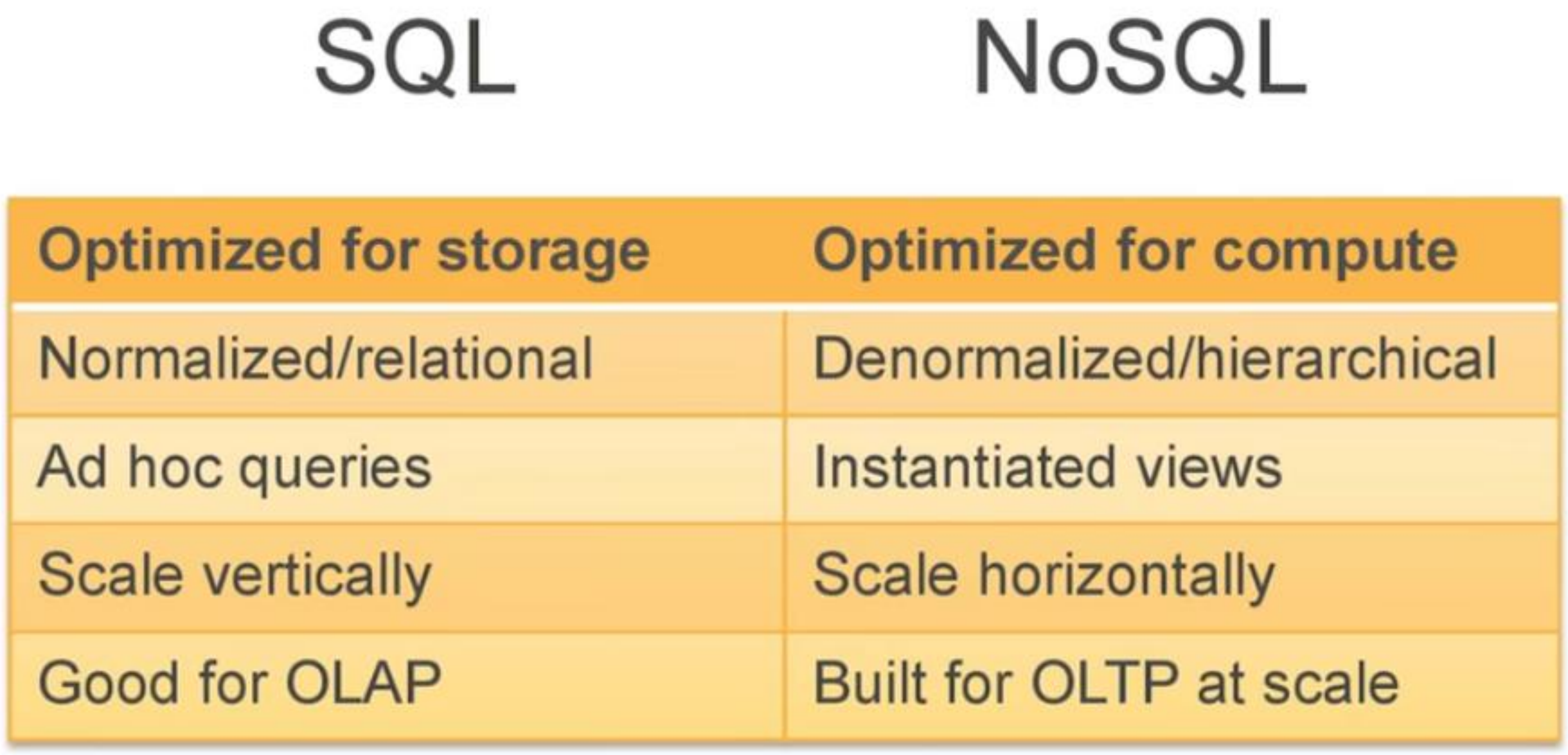


Source: http://aws.amazon.com

Relational (SQL) vs. Non-relational (NoSQL)



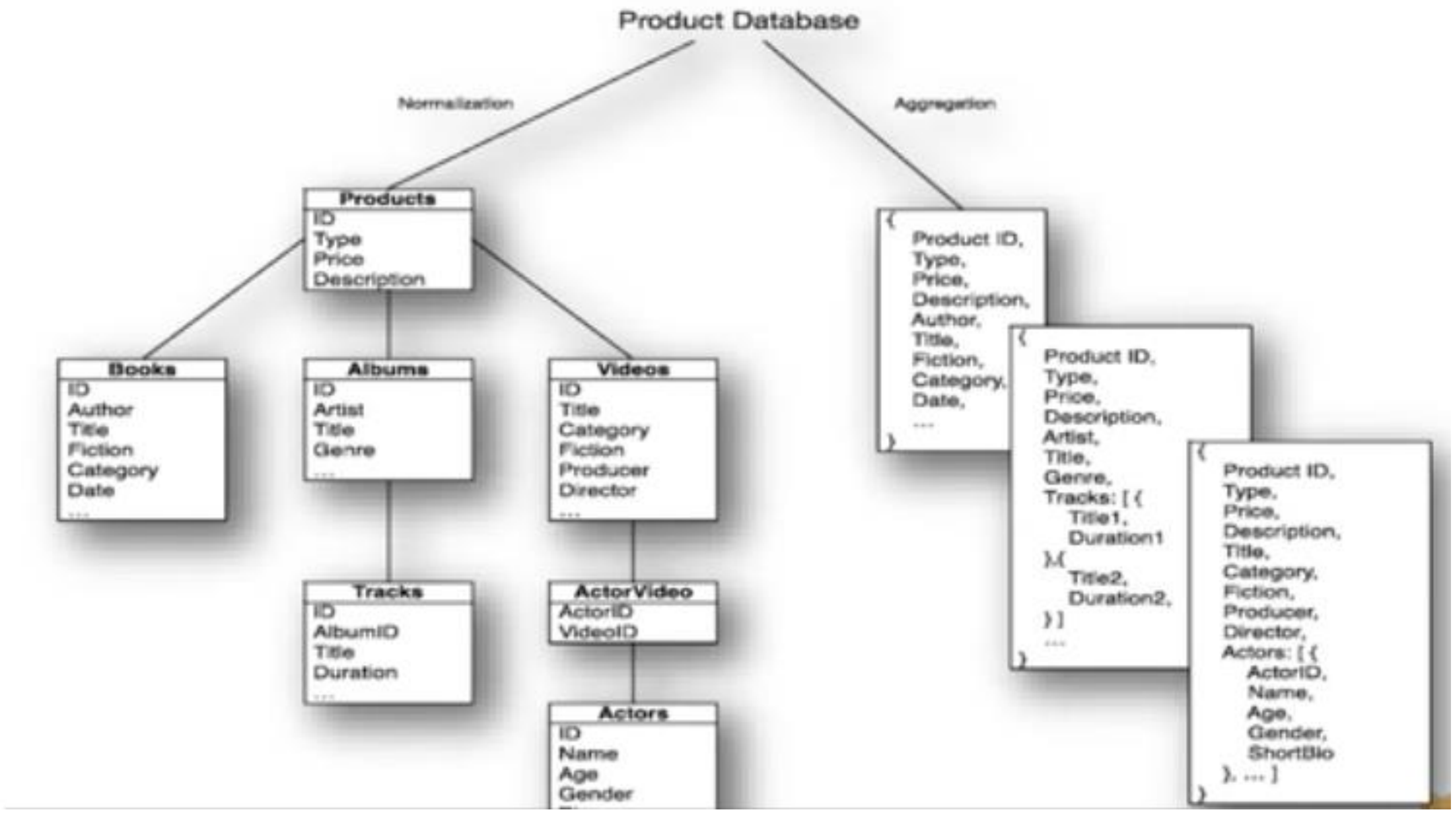
Relational (SQL) vs. Non-relational (NoSQL)



SQL	NoSQL
Optimized for storage	**Optimized for compute**
Normalized/relational	Denormalized/hierarchical
Ad hoc queries	Instantiated views
Scale vertically	Scale horizontally
Good for OLAP	Built for OLTP at scale

Relational (SQL) vs. Non-relational (NoSQL)

SQL vs. NoSQL Access Pattern

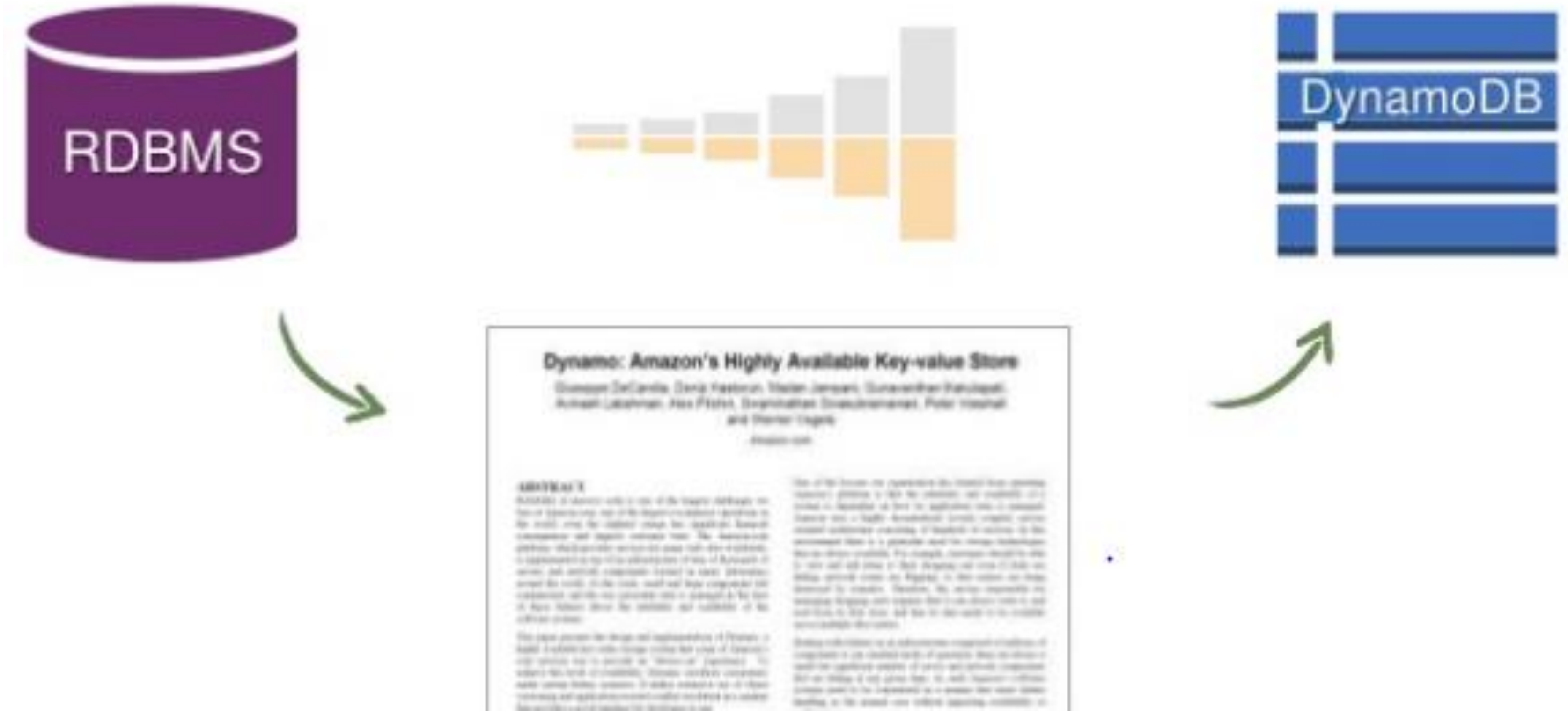

Product Database
Normalization
Aggregation
Products
ID
Type
Price
Description
Books
ID
Author
Title
Fiction
Category
Date
...
Albums
ID
Artist
Title
Genre
...
Videos
ID
Title
Category
Fiction
Producer
Director
...
Tracks
ID
AlbumID
Title
Duration
...
ActorVideo
ActorID
VideoID
Actors
ID
Name
Age
Gender
{ Product ID, Type, Price, Description, Author, Title, Fiction, Category, Date, ... }
{ Product ID, Type, Price, Description, Artist, Title, Genre, Tracks: [{ Title1, Duration1 },{ Title2, Duration2, }] ... }
{ Product ID, Type, Price, Description, Title, Category, Fiction, Producer, Director, Actors: [{ ActorID, Name, Age, Gender, ShortBio }, ...] }

NoSQL Solutions on AWS

- Bring your own NoSQL database, or use DyanamoDB
- Popular NoSQL Options
  - MongoDB
  - Cassandra
  - MarkLogic
  - Couchbase
  - DynamoDB
- Avoid the overhead of provisioning hardware

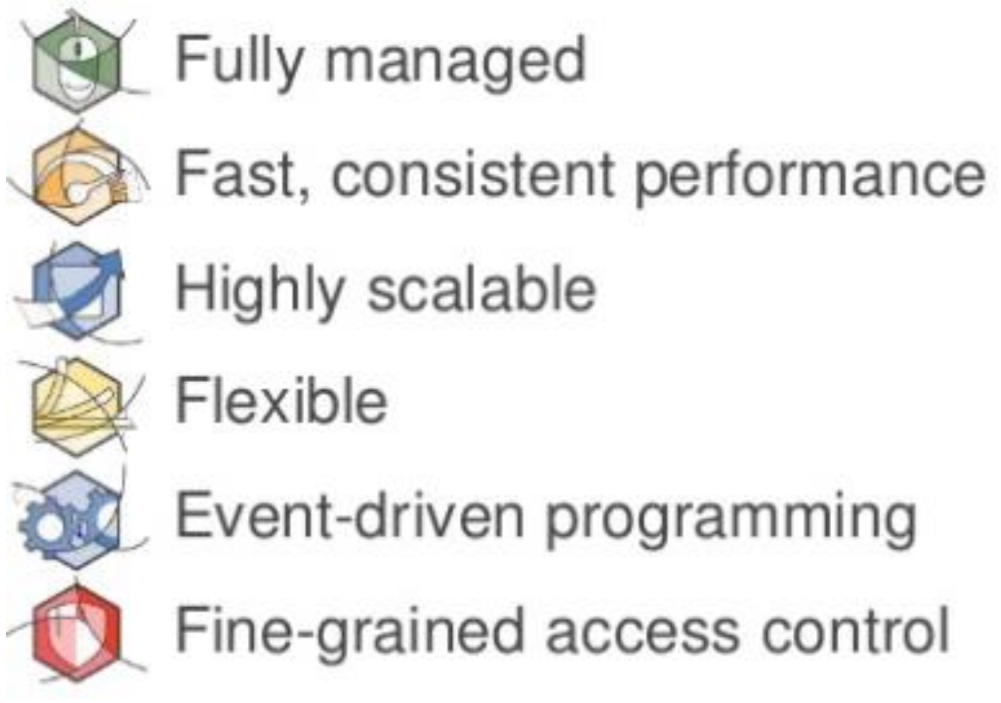
Amazon DynamoDB

- Amazon’s path to DynamoDB



Amazon DynamoDB

- Amazon DynamoDB is a highly scalable, fast, consistent performance and fully managed NoSQL database service
  - Built for applications that need consistent, single-digit millisecond latency at any scale.
  - Supported by auto-scaling to hundreds of terabytes of data, that serve millions of requests per second
- Key Characteristics:



Fully managed

Fast, consistent performance

Highly scalable

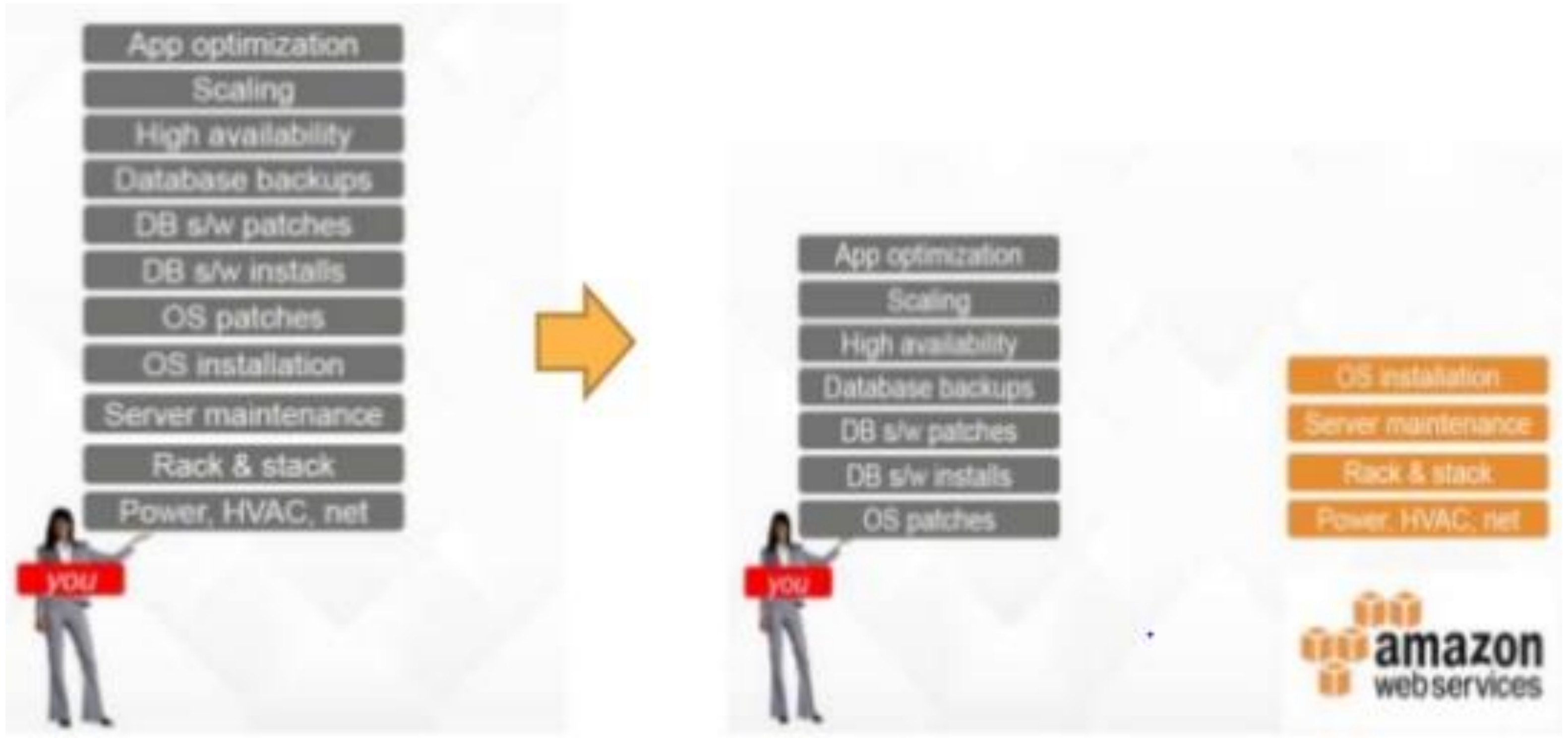
Flexible

Event-driven programming

Fine-grained access control

Amazon DynamoDB

- Fully managed service = automated operations

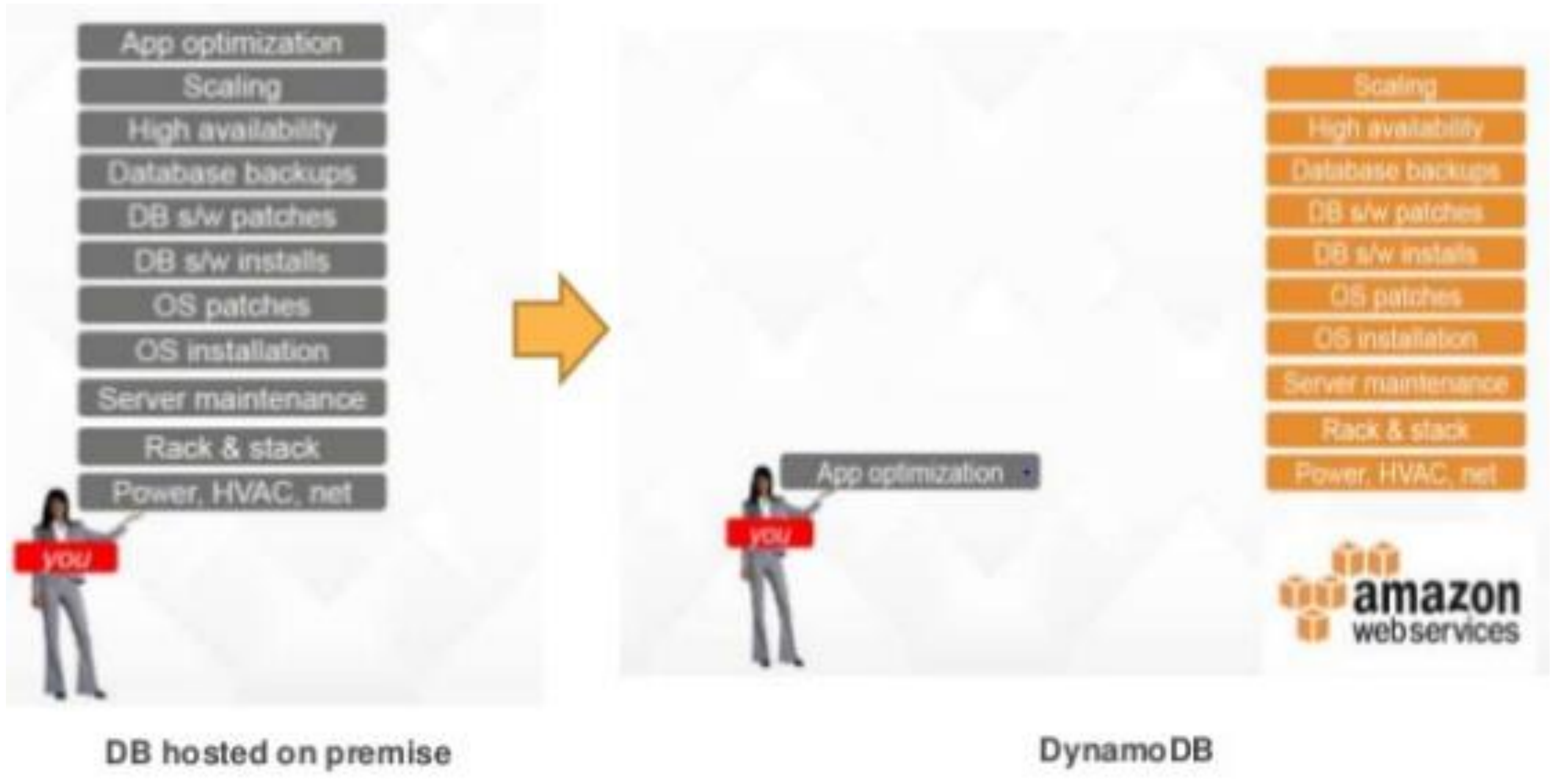


DB hosted on-premises

DB hosted on Amazon EC2

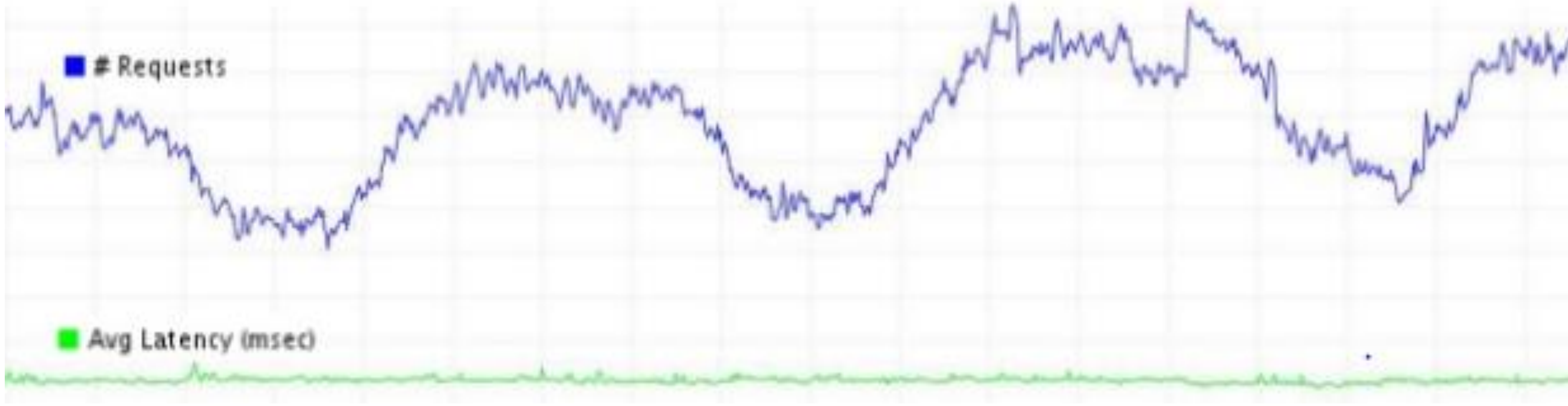
Amazon DynamoDB

- Fully managed service = automated operations



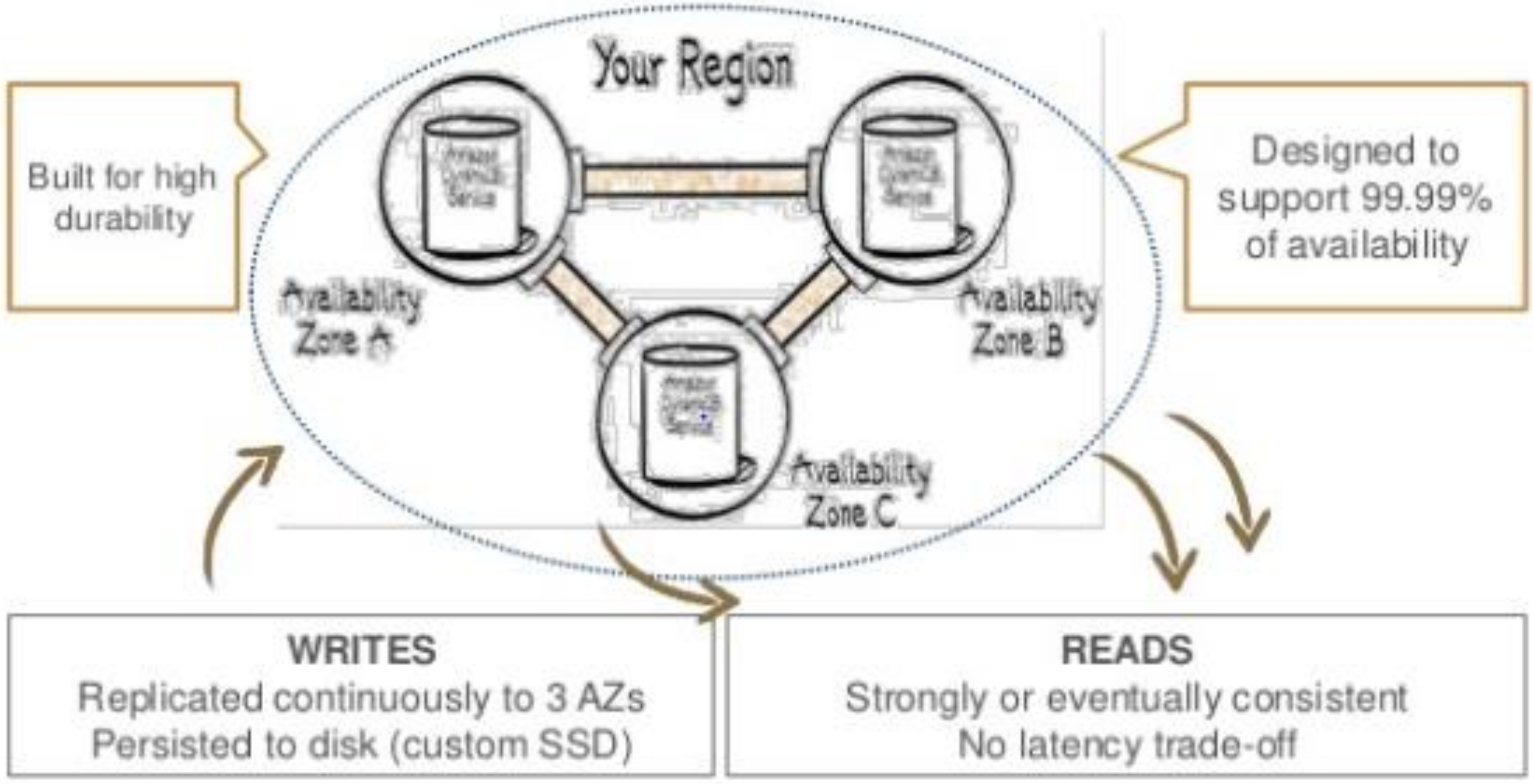
Amazon DynamoDB

- Consistently low latency at scale
- Predictable performance



Amazon DynamoDB

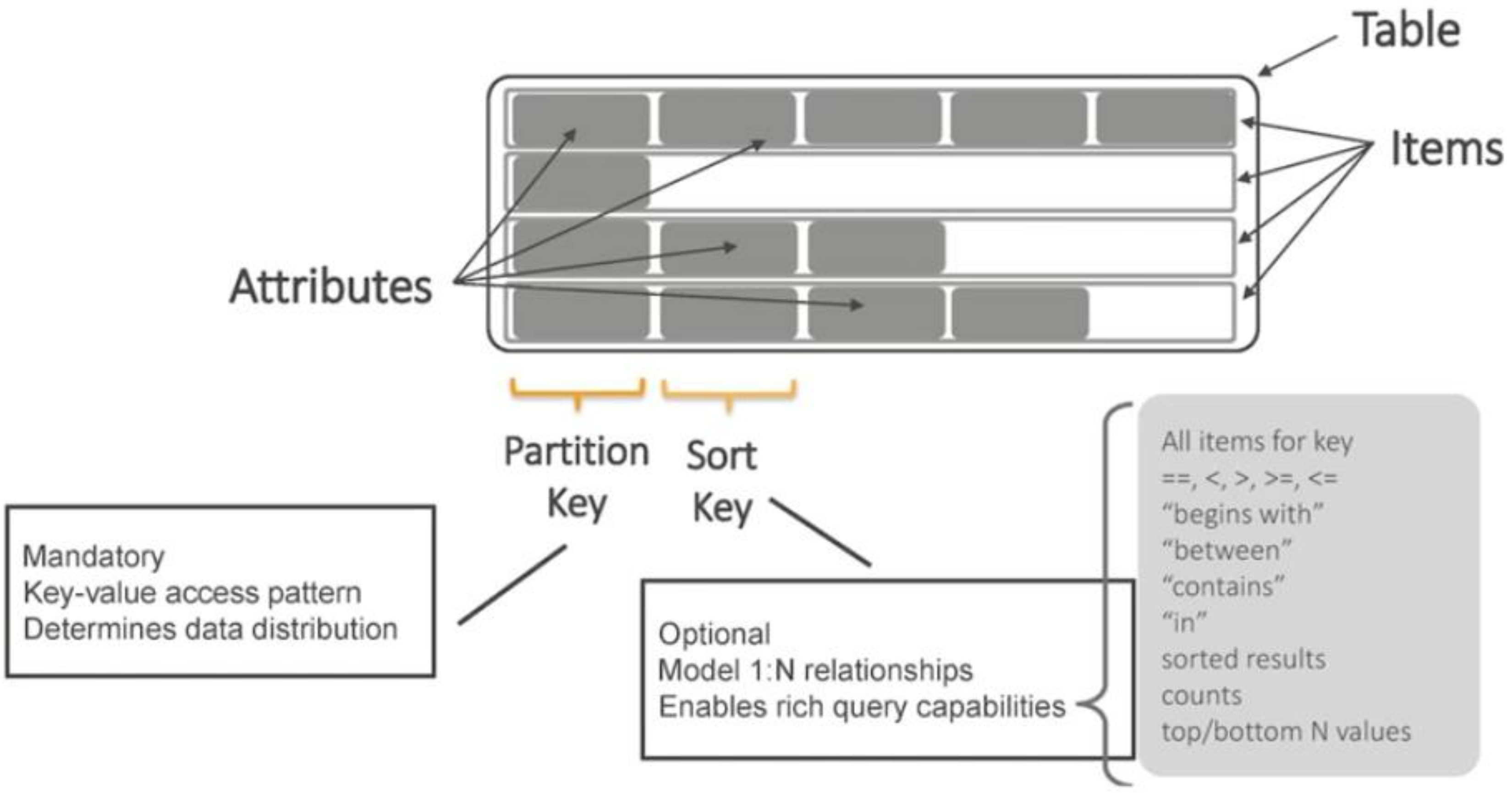
- High availability and durability


Built for high durability
Your Region
Designed to support 99.99% of availability
Availability Zone A
Availability Zone B
Availability Zone C
WRITES
Replicated continuously to 3 AZs
Persisted to disk (custom SSD)
READS
Strongly or eventually consistent
No latency trade-off

Tables and Indexes

Tables and Indexes

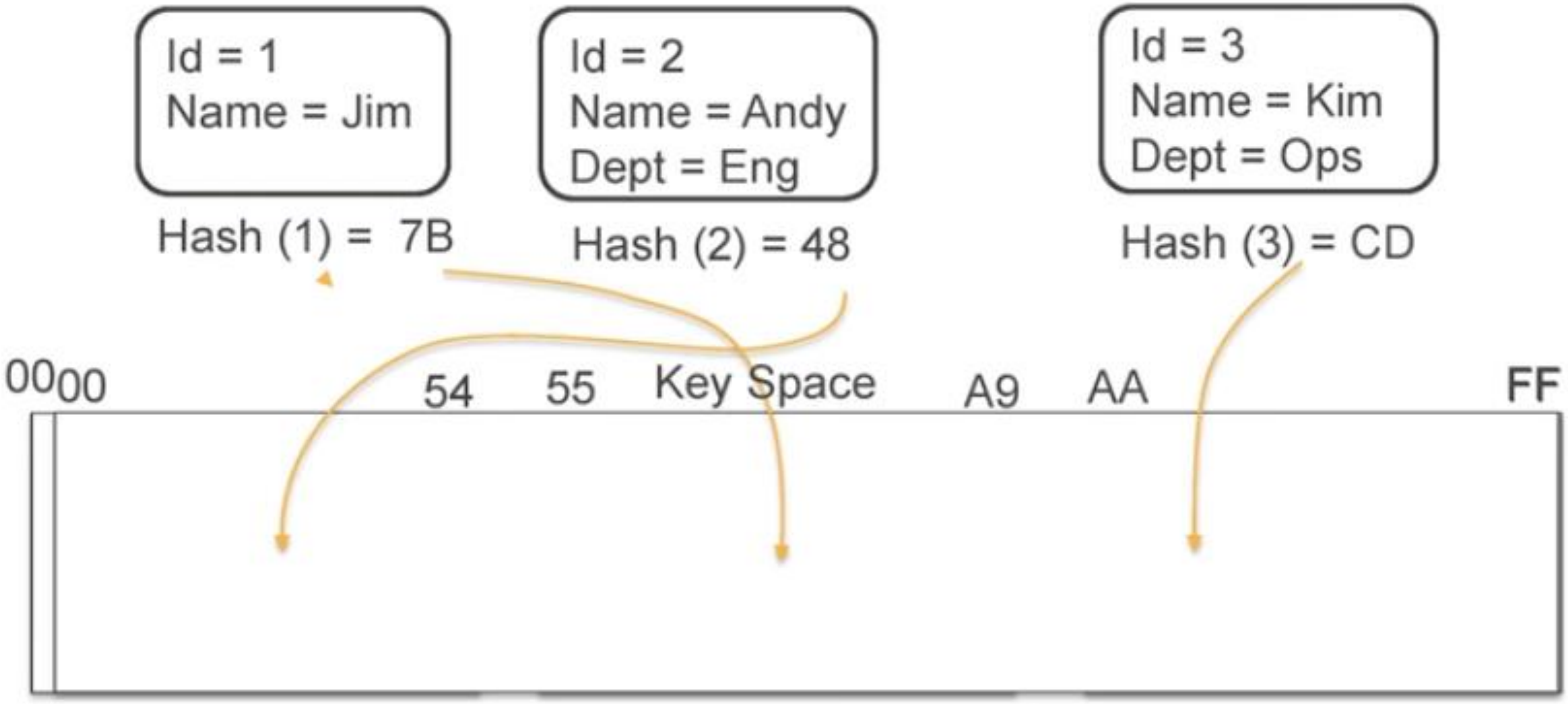
- DynamoDB table structure



Tables and Indexes

Partition Keys

Partition Key uniquely identifies an item
Partition Key is used for building an unordered hash index
Allows table to be partitioned for scale

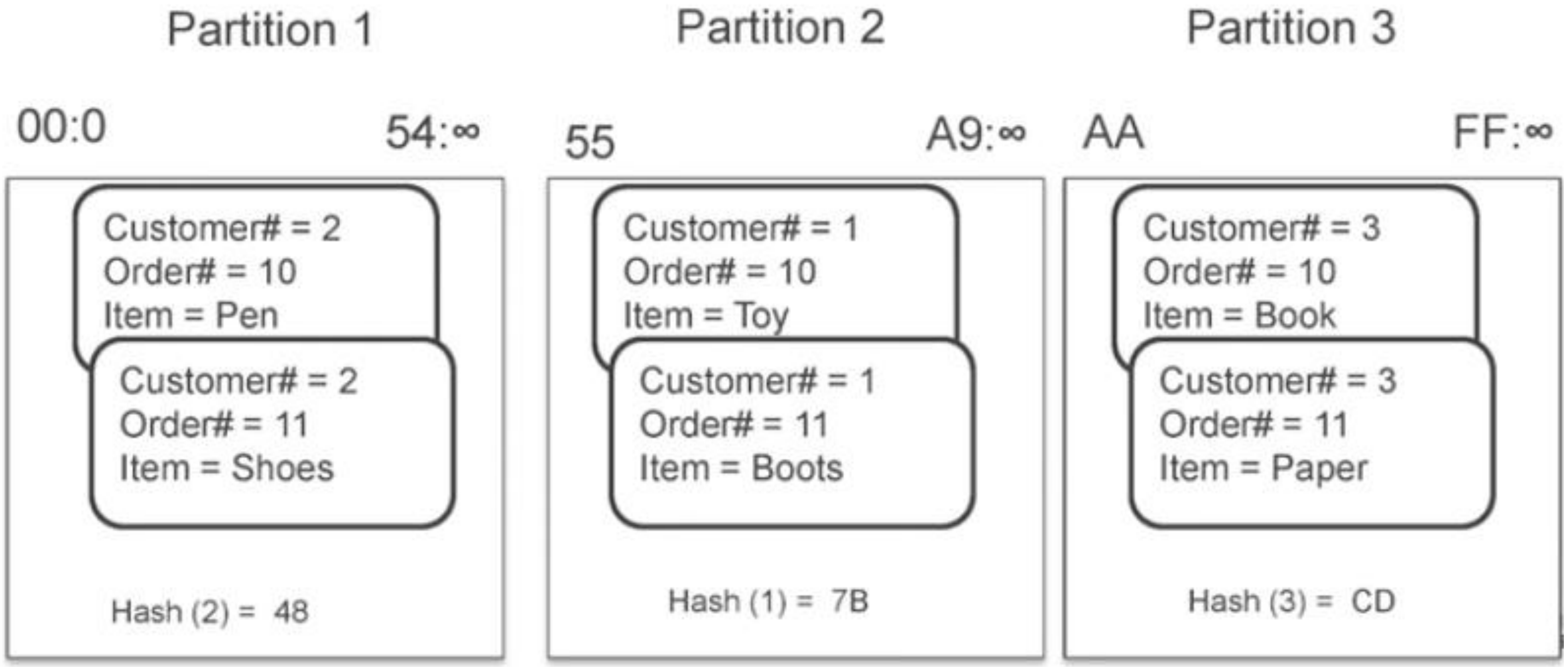


Tables and Indexes

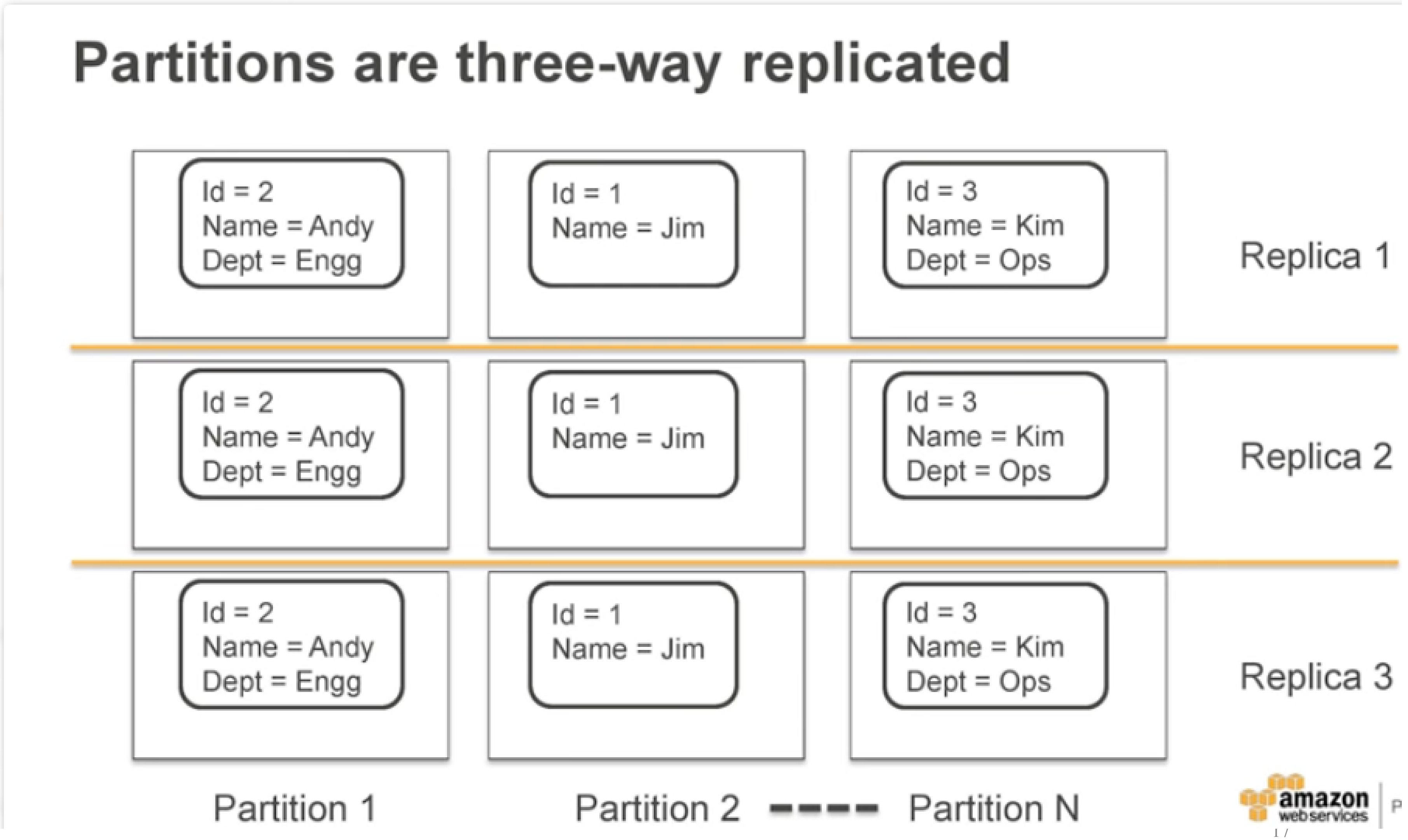
Partition:Sort Key

Partition:Sort Key uses two attributes together to uniquely identify an Item
Within unordered hash index, data is arranged by the sort key
No limit on the number of items (∞) per partition key

- Except if you have local secondary indexes



Tables and Indexes

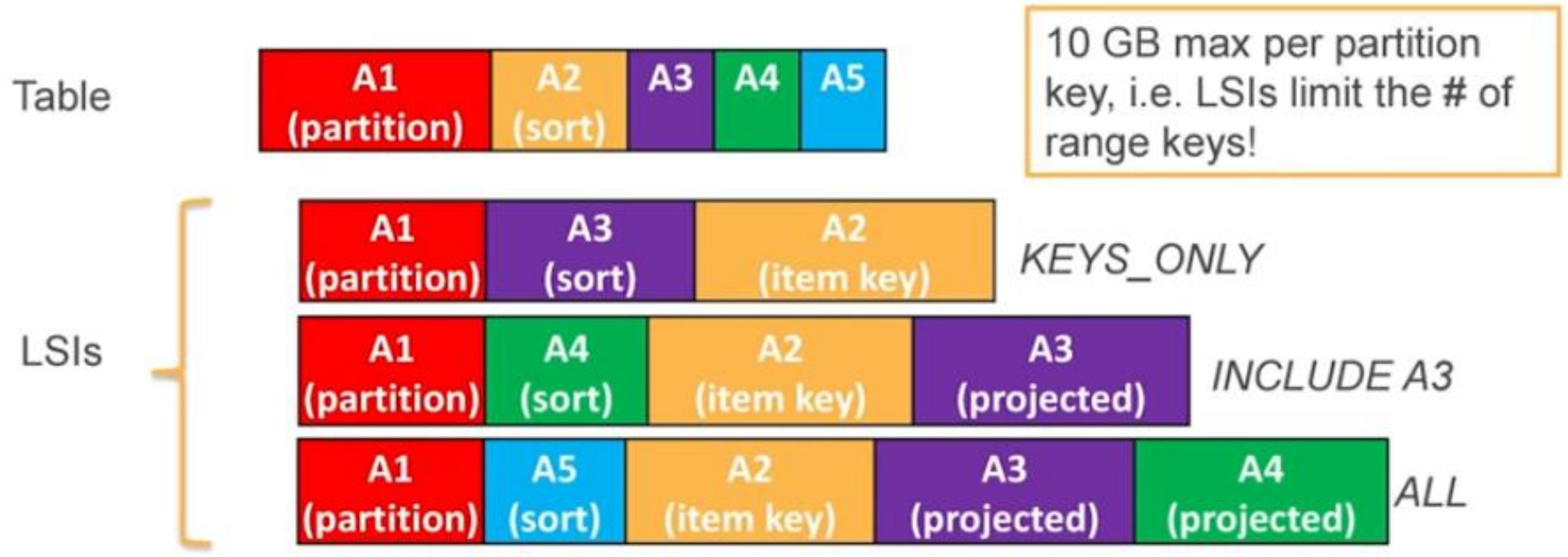


Tables and Indexes

Local secondary index (LSI)

Alternate sort key attribute

Index is local to a partition key



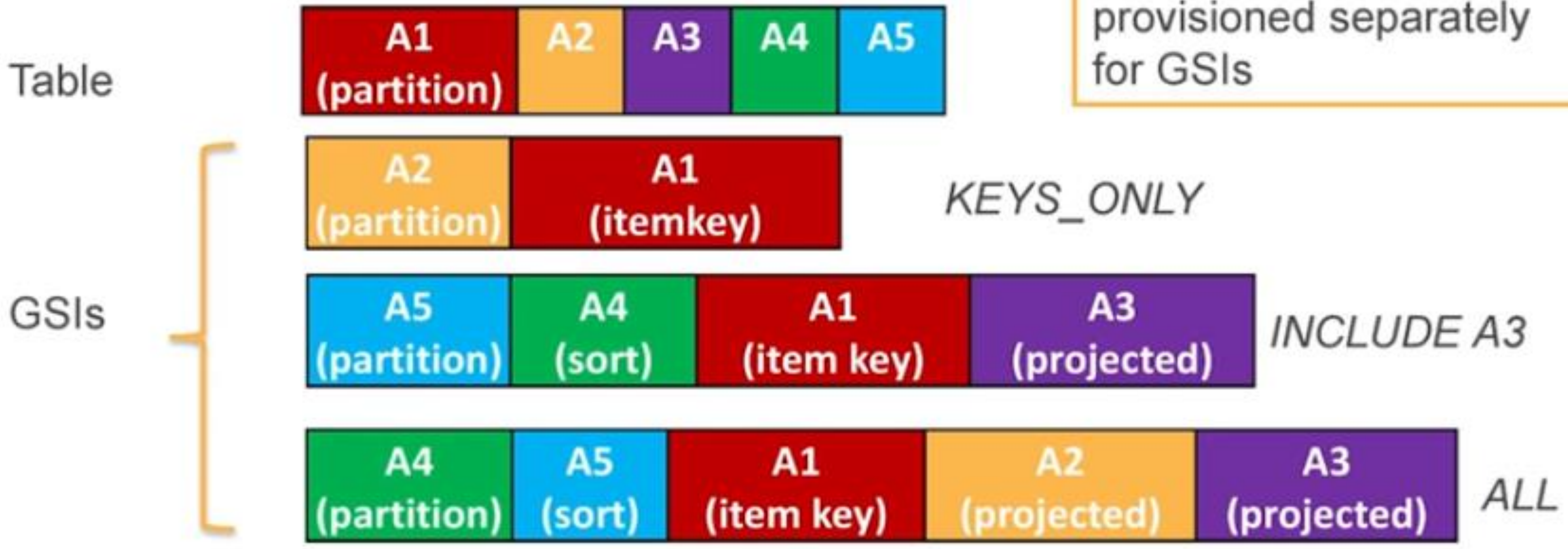
Tables and Indexes

Global secondary index (GSI)

Alternate partition and/or sort key

Index is across all partition keys

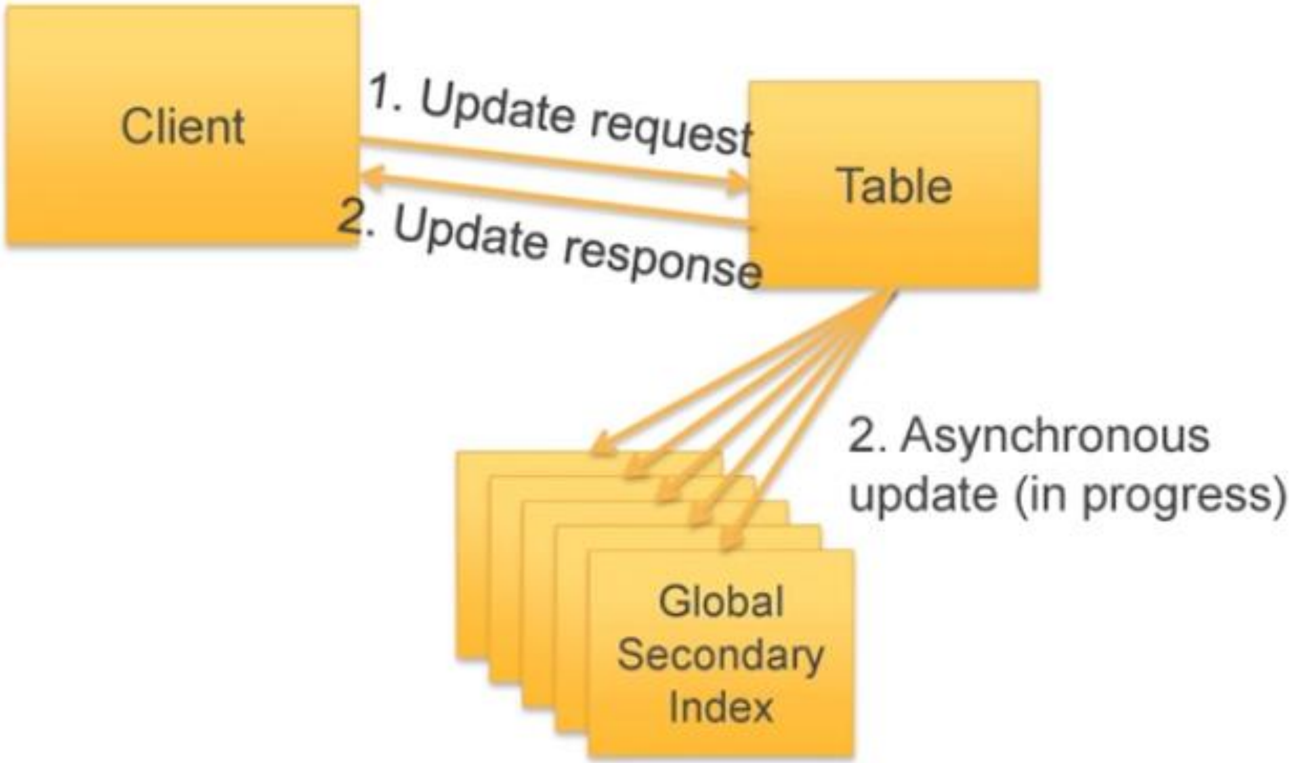
Online indexing



RCUs/WCUs provisioned separately for GSIs

Tables and Indexes

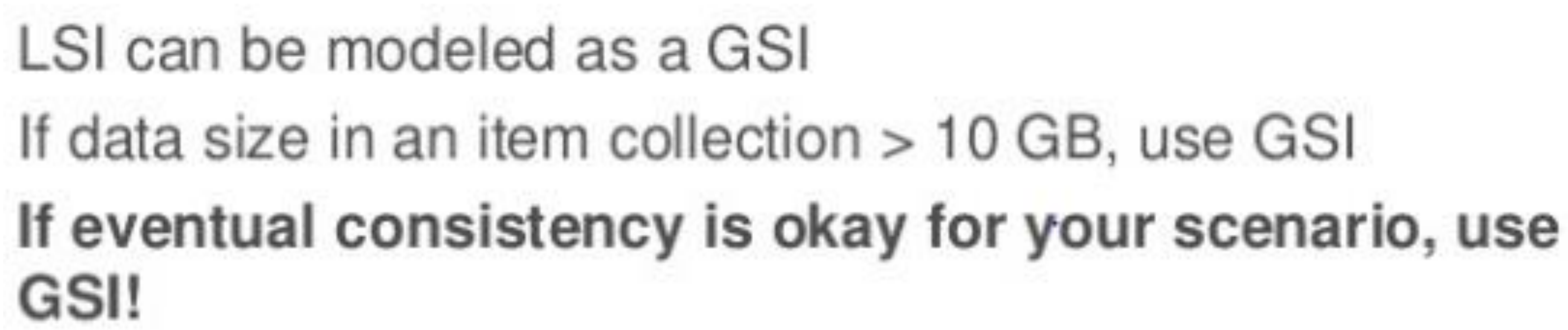
How do GSI updates work?



If GSIs don't have enough write capacity, table writes will be throttled!

Tables and Indexes

- LSI or GSI?



LSI can be modeled as a GSI

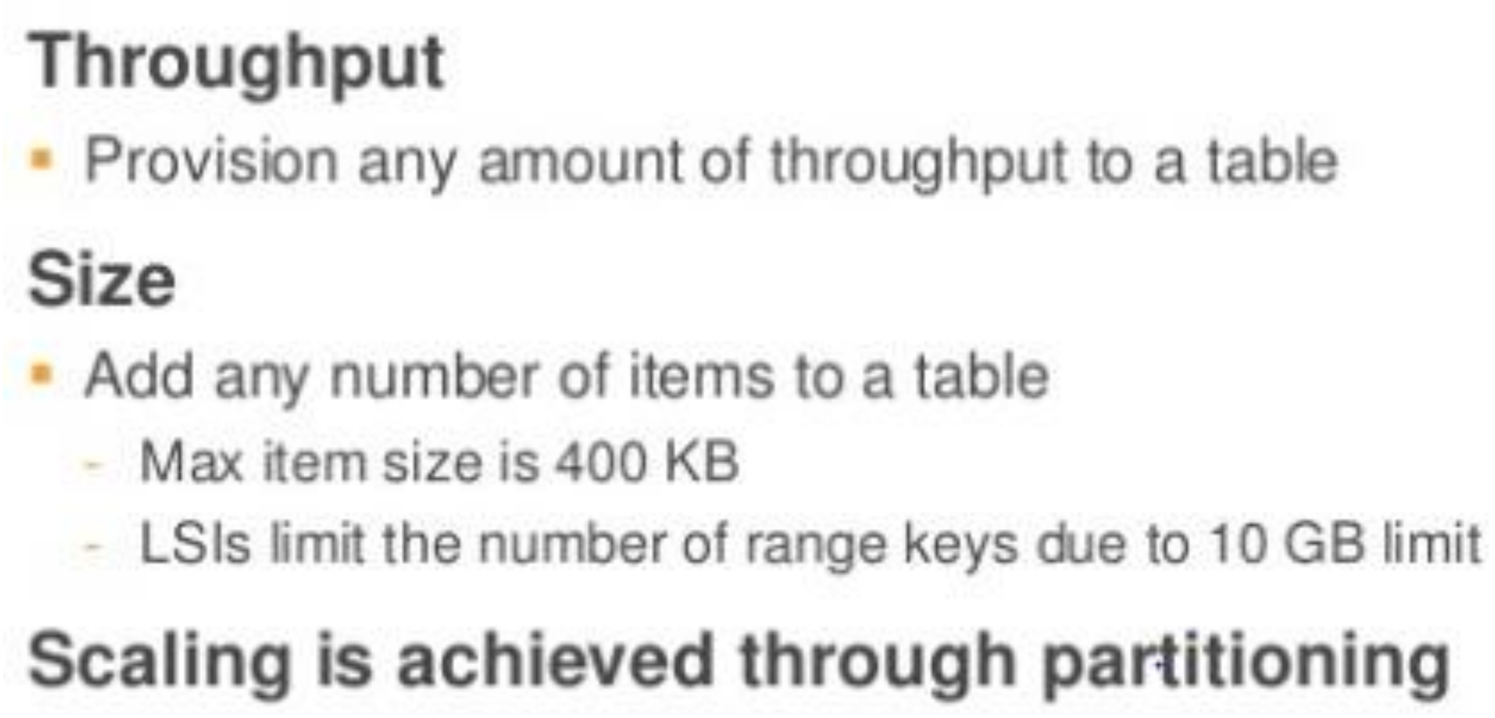
If data size in an item collection > 10 GB, use GSI

If eventual consistency is okay for your scenario, use GSI!

DynamoDB Scaling

Scaling

- Based on Throughput and Size



Throughput

- Provision any amount of throughput to a table

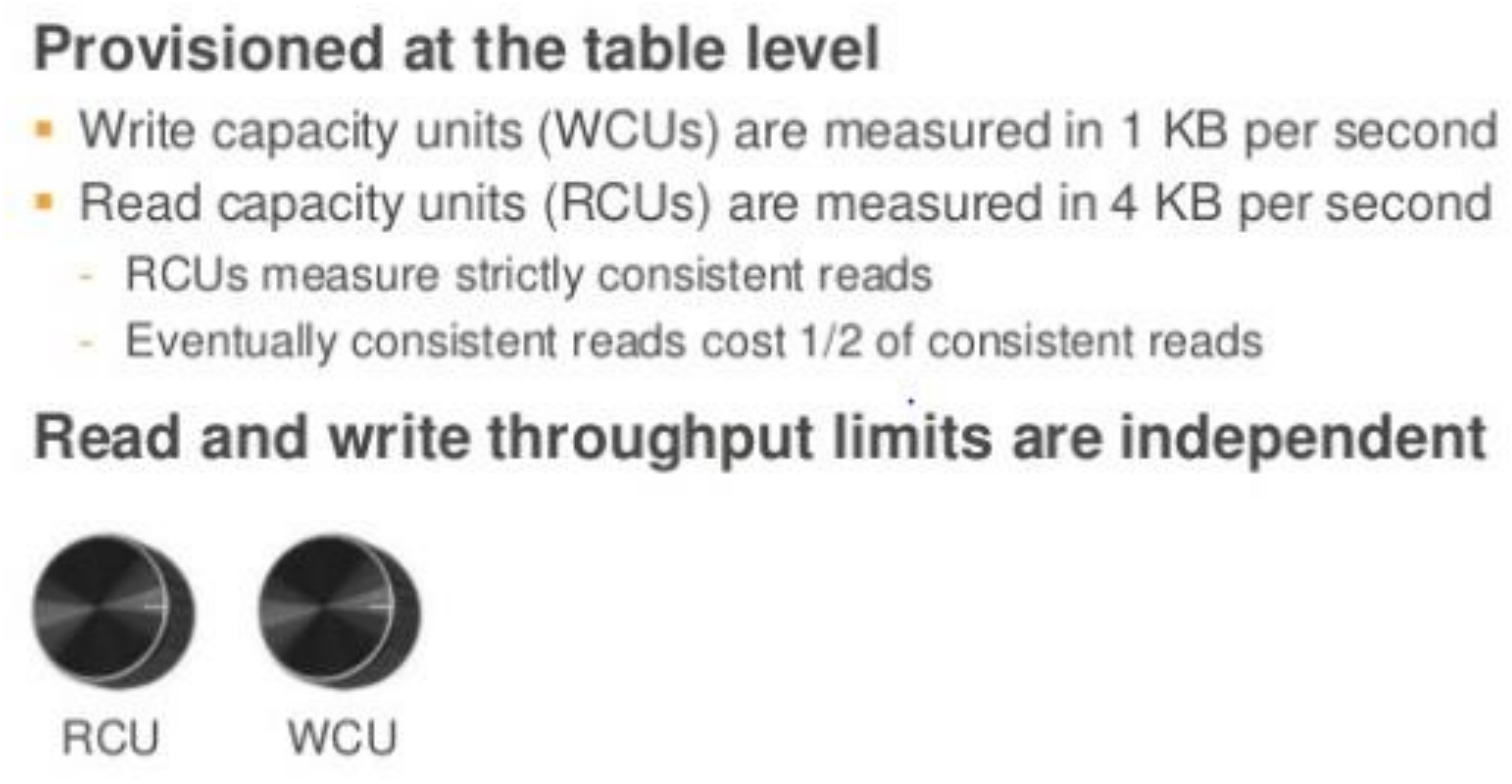
Size

- Add any number of items to a table
  - Max item size is 400 KB
  - LSIs limit the number of range keys due to 10 GB limit

Scaling is achieved through partitioning

Scaling

- Throughput



Provisioned at the table level

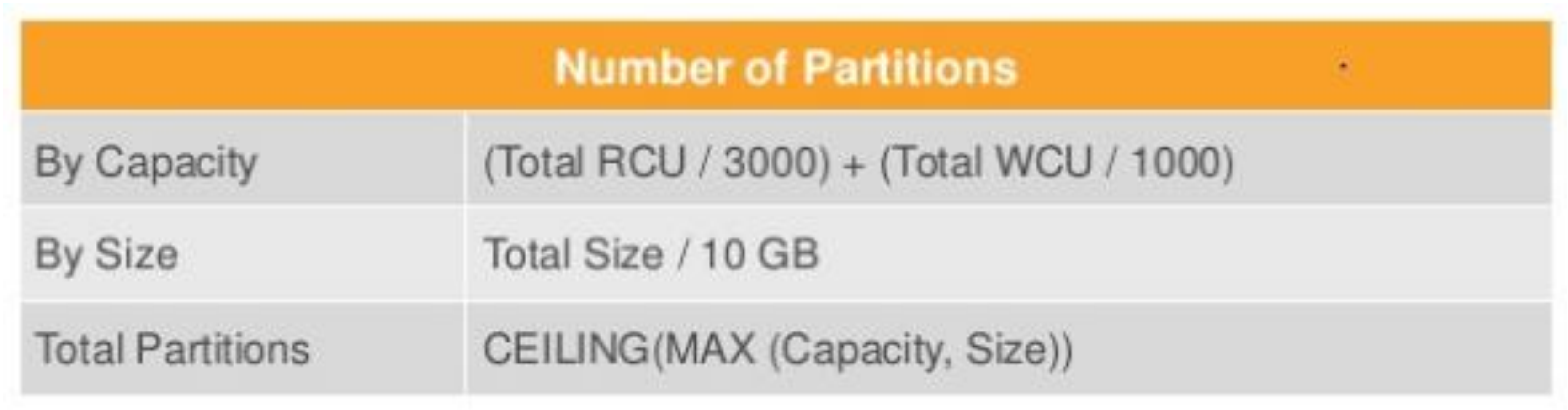
- Write capacity units (WCUs) are measured in 1 KB per second
- Read capacity units (RCUs) are measured in 4 KB per second
  - RCUs measure strictly consistent reads
  - Eventually consistent reads cost 1/2 of consistent reads

Read and write throughput limits are independent

RCU WCU

Scaling

- Partitioning Math



Number of Partitions	
By Capacity	(Total RCU / 3000) + (Total WCU / 1000)
By Size	Total Size / 10 GB
Total Partitions	CEILING(MAX (Capacity, Size))

Scaling

- Partitioning Example

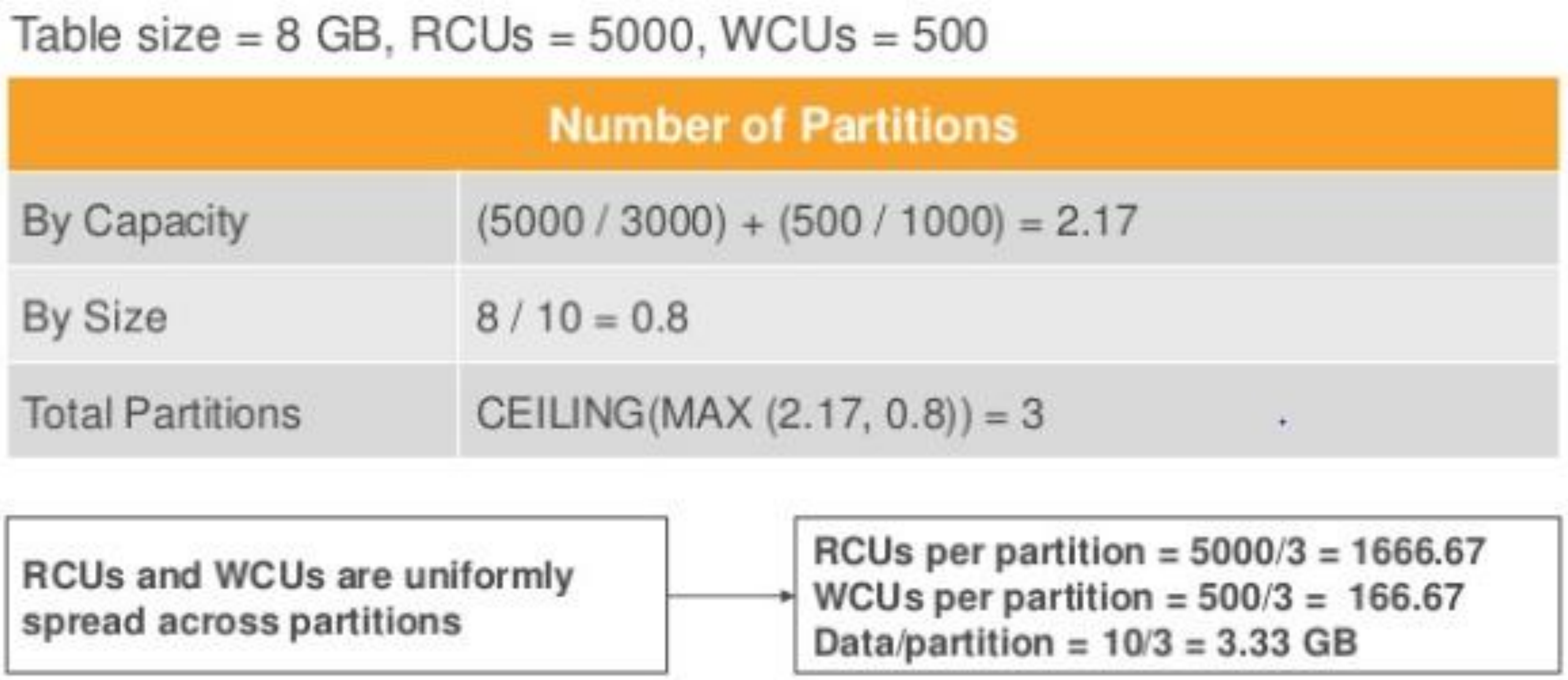


Table size = 8 GB, RCUs = 5000, WCUs = 500

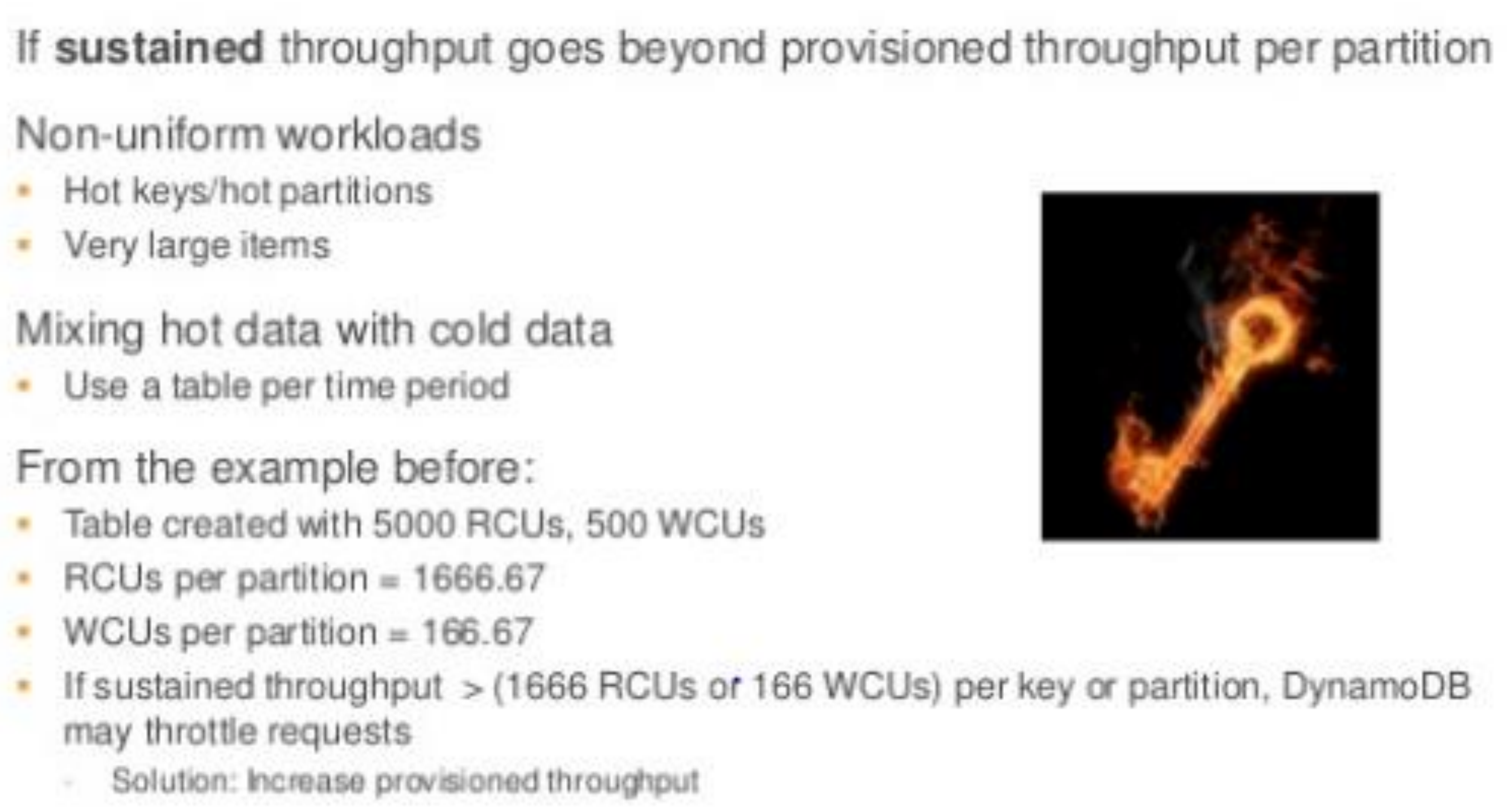
Number of Partitions	
By Capacity	(5000 / 3000) + (500 / 1000) = 2.17
By Size	8 / 10 = 0.8
Total Partitions	CEILING(MAX (2.17, 0.8)) = 3

RCUs and WCUs are uniformly spread across partitions

RCUs per partition = 5000/3 = 1666.67
WCUs per partition = 500/3 = 166.67
Data/partition = 10/3 = 3.33 GB

Scaling

- What causes throttling?



If **sustained** throughput goes beyond provisioned throughput per partition

Non-uniform workloads

- Hot keys/hot partitions
- Very large items

Mixing hot data with cold data

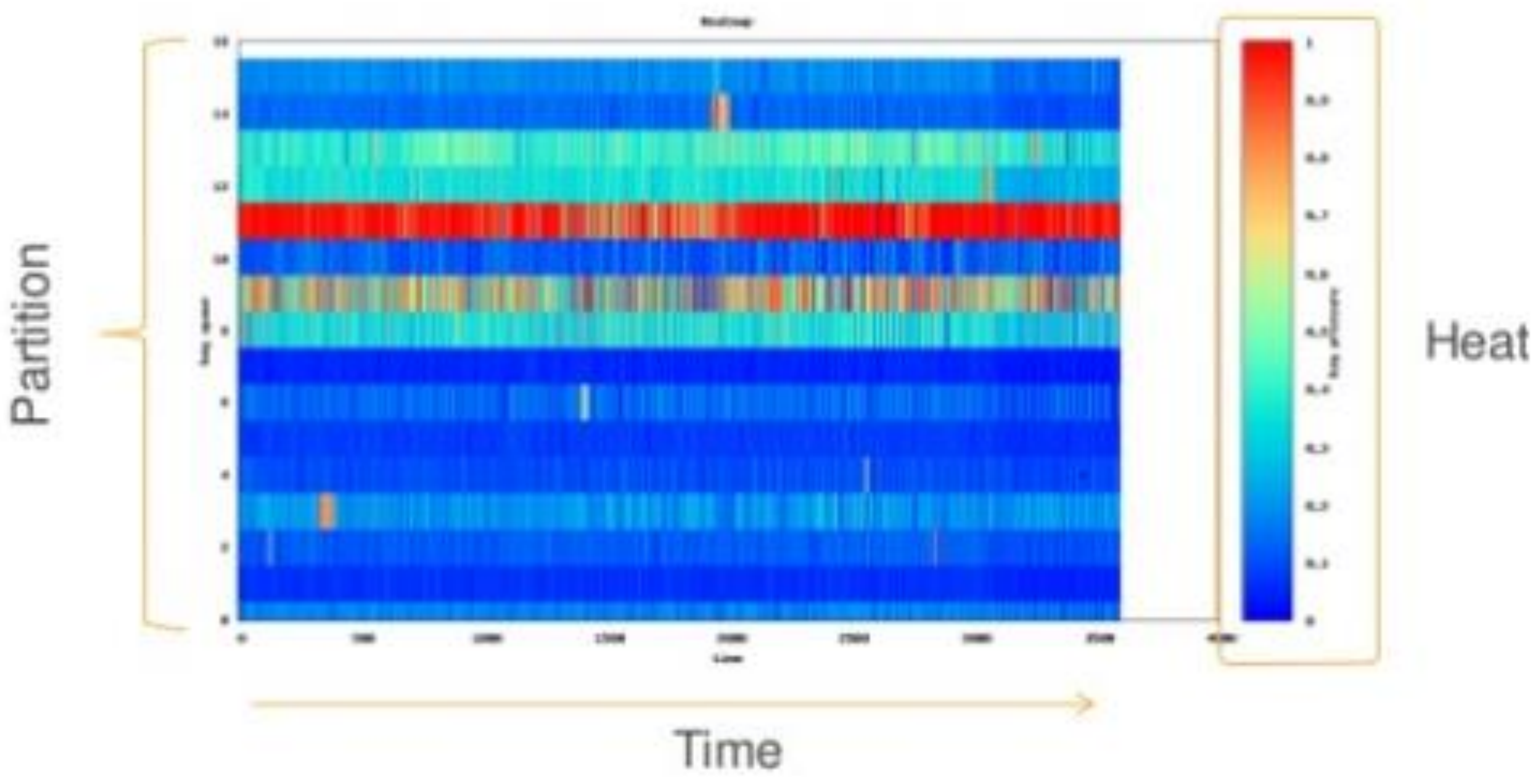
- Use a table per time period

From the example before:

- Table created with 5000 RCUs, 500 WCUs
- RCUs per partition = 1666.67
- WCUs per partition = 166.67
- If sustained throughput > (1666 RCUs or 166 WCUs) per key or partition, DynamoDB may throttle requests
  - Solution: Increase provisioned throughput

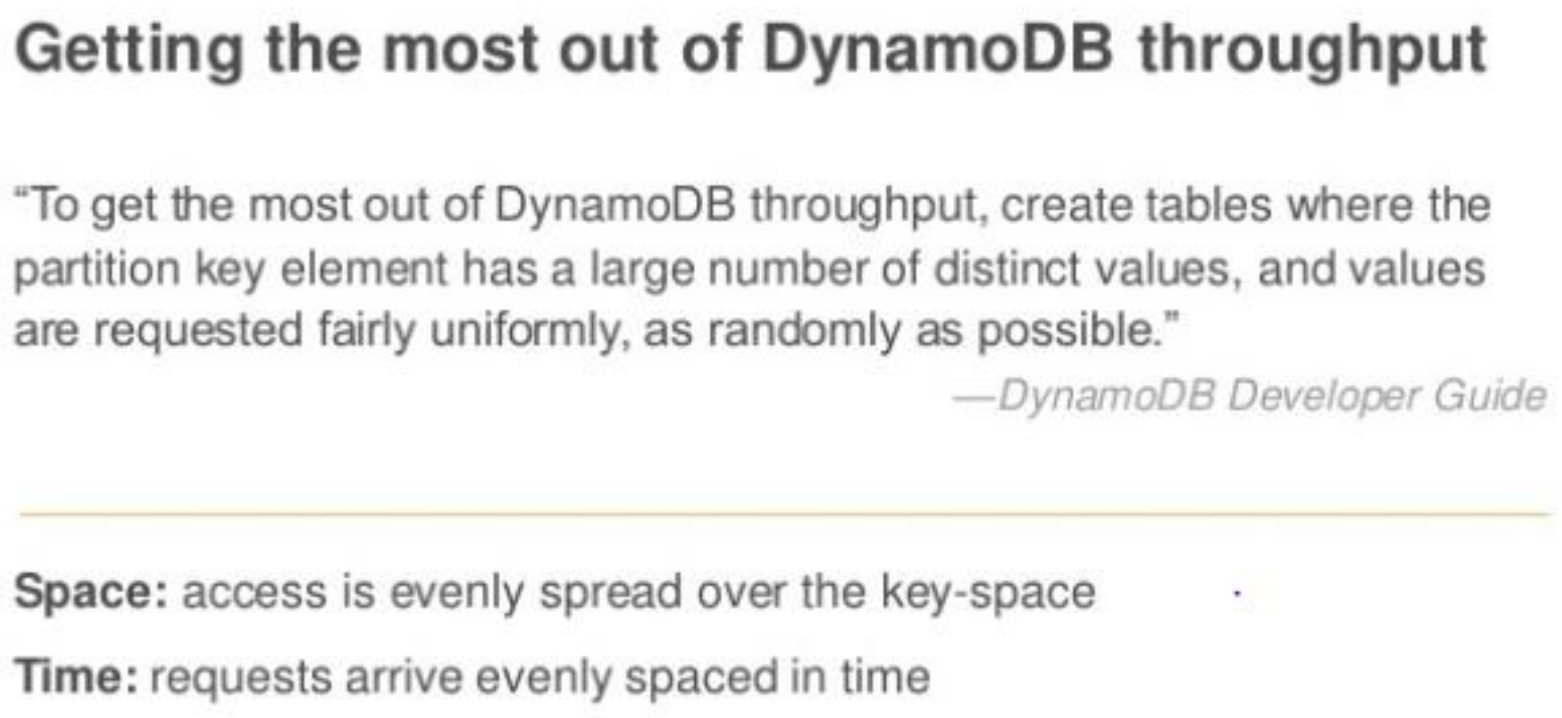
Scaling

- An example a bad NoSQL



Scaling

- Ways to avoid throttling



Getting the most out of DynamoDB throughput

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

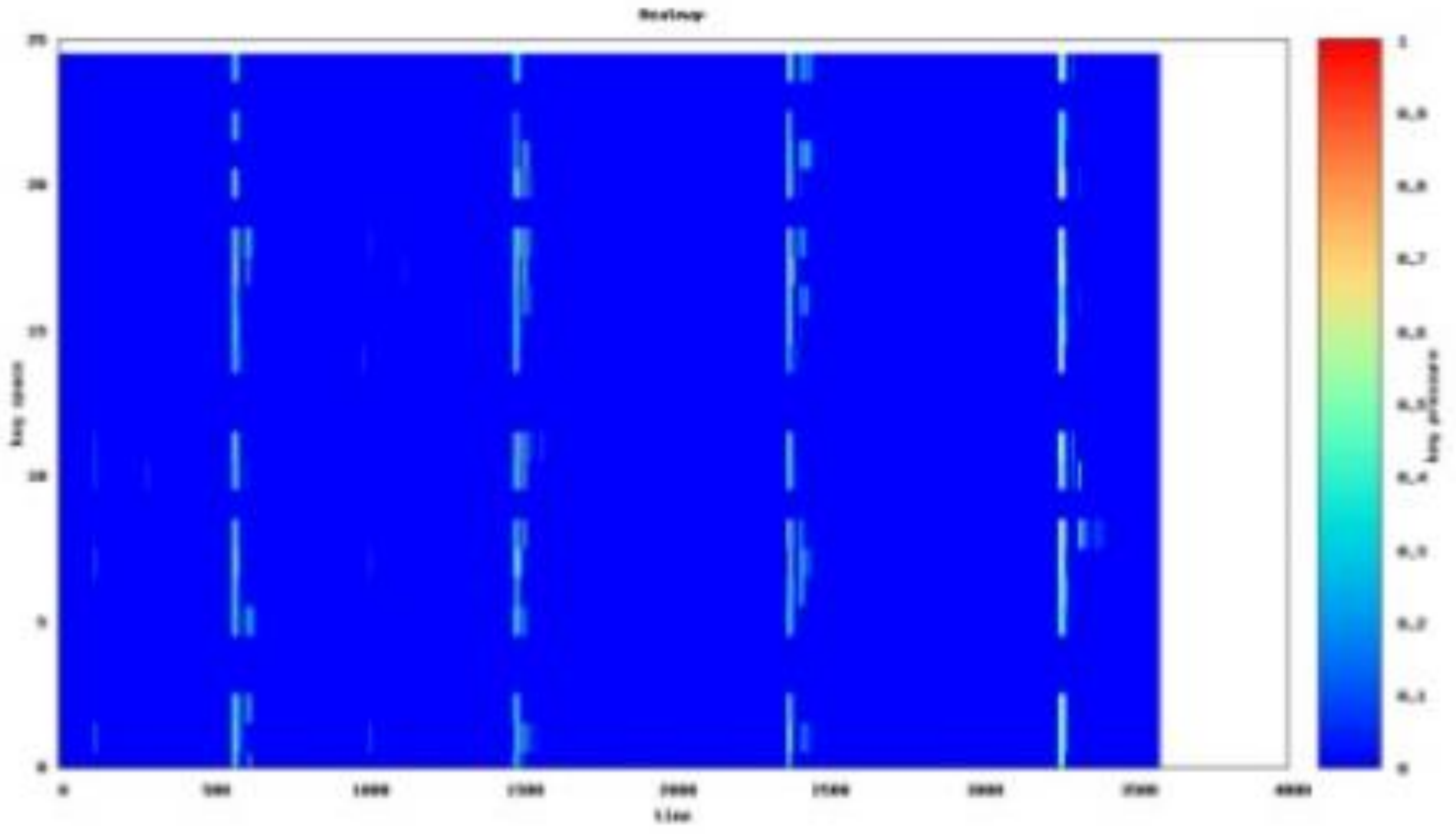
—*DynamoDB Developer Guide*

Space: access is evenly spread over the key-space

Time: requests arrive evenly spaced in time

Scaling

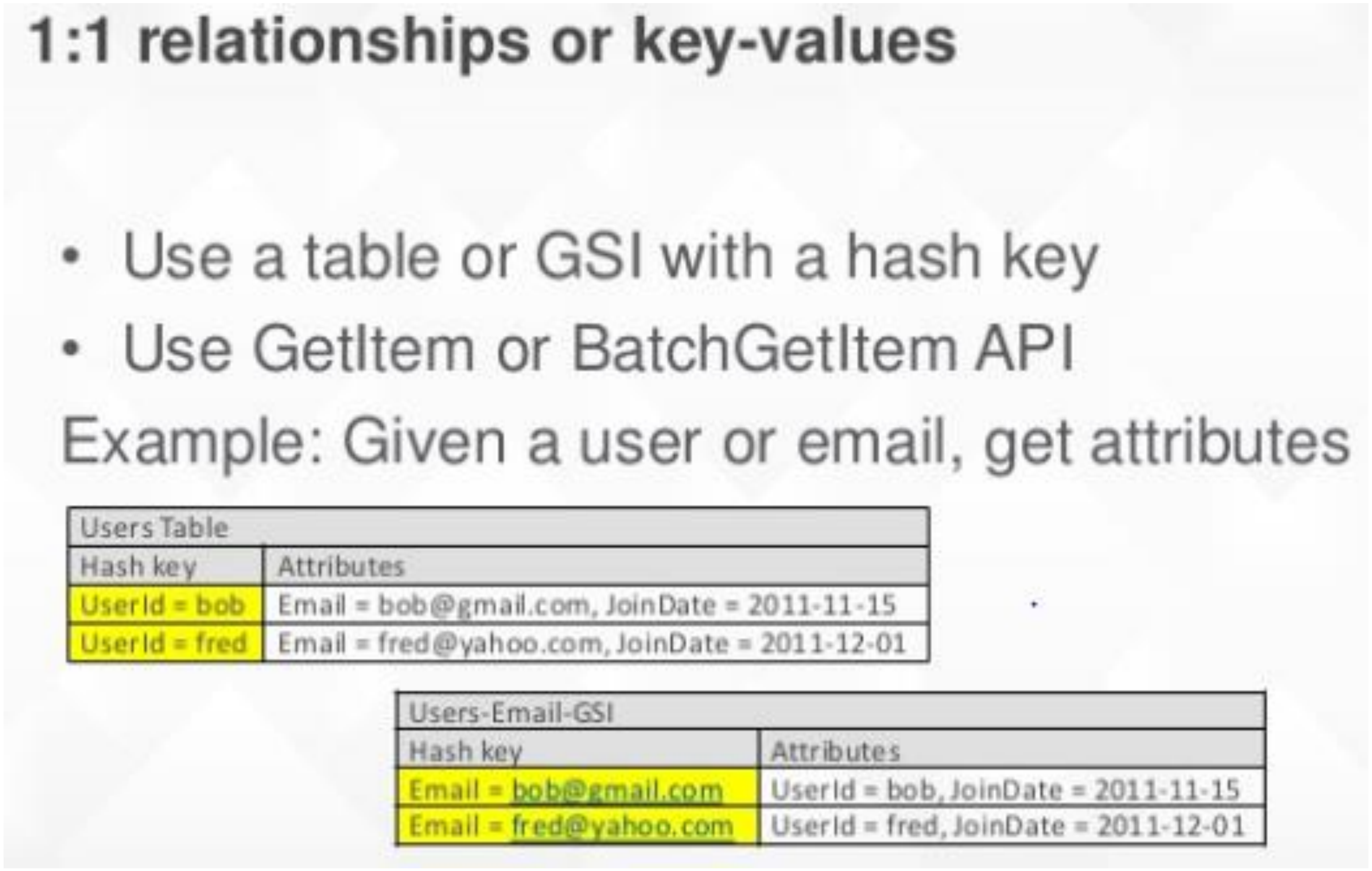
- A much better picture…..



Data Modeling in NoSQL

Data Modeling

- One-to-one relationships



1:1 relationships or key-values

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

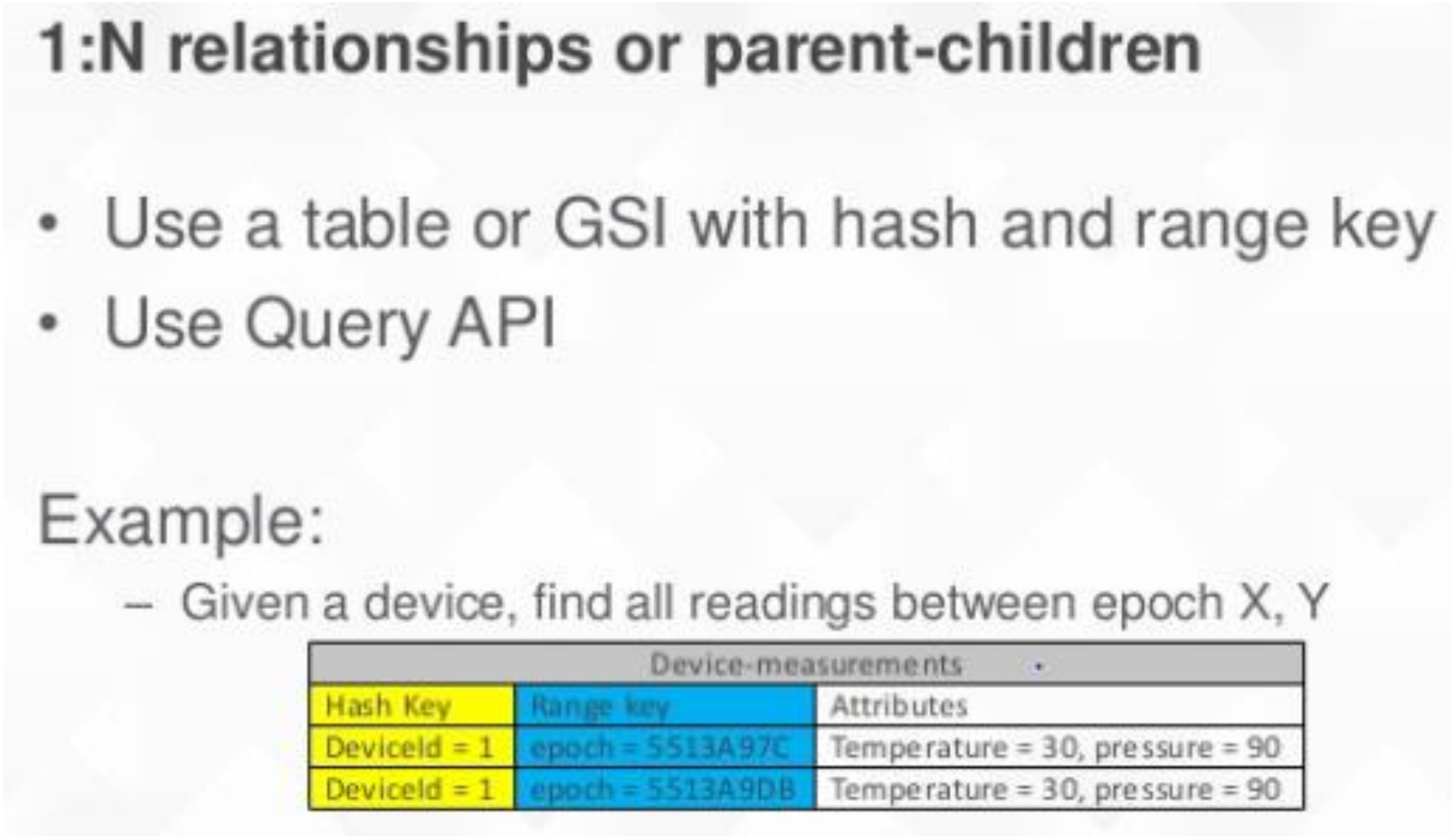
Example: Given a user or email, get attributes

Users Table	
Hash key	Attributes
UserId = bob	Email = bob@gmail.com, JoinDate = 2011-11-15
UserId = fred	Email = fred@yahoo.com, JoinDate = 2011-12-01

Users-Email-GSI	
Hash key	Attributes
Email = bob@gmail.com	UserId = bob, JoinDate = 2011-11-15
Email = fred@yahoo.com	UserId = fred, JoinDate = 2011-12-01

Data Modeling

- One-to-many relationships



1:N relationships or parent-children

- Use a table or GSI with hash and range key
- Use Query API

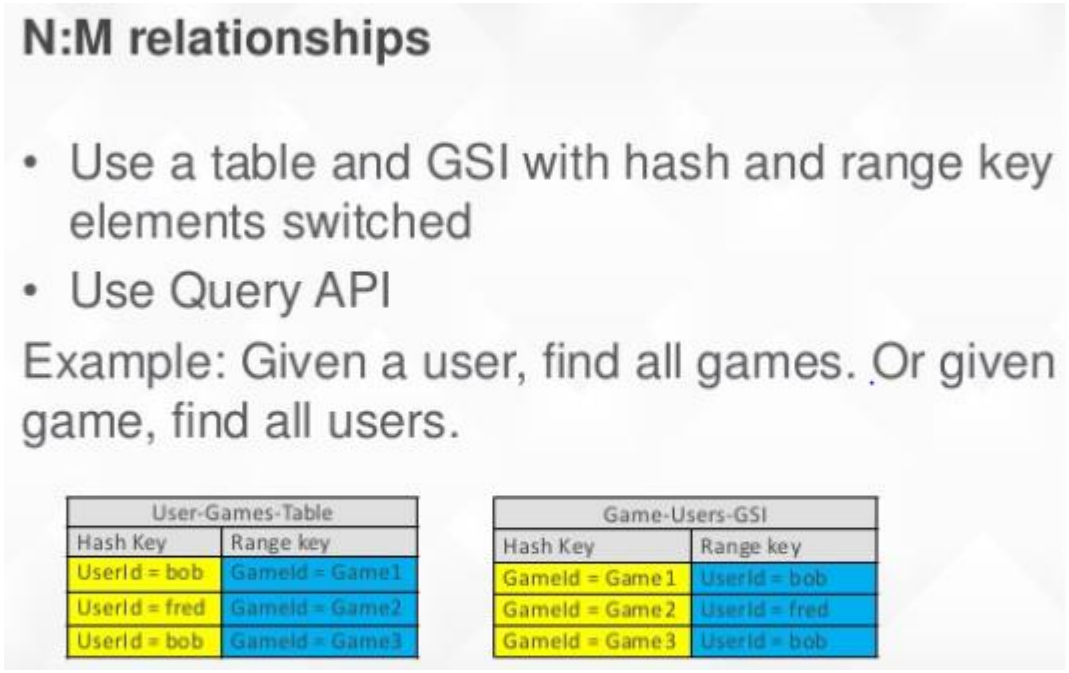
Example:

– Given a device, find all readings between epoch X, Y

Device-measurements		
Hash Key	Range key	Attributes
DeviceId = 1	epoch = 5513A97C	Temperature = 30, pressure = 90
DeviceId = 1	epoch = 5513A9DB	Temperature = 30, pressure = 90

Data Modeling

- Many-to-many relationships



N:M relationships

- Use a table and GSI with hash and range key elements switched
- Use Query API

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

User-Games-Table

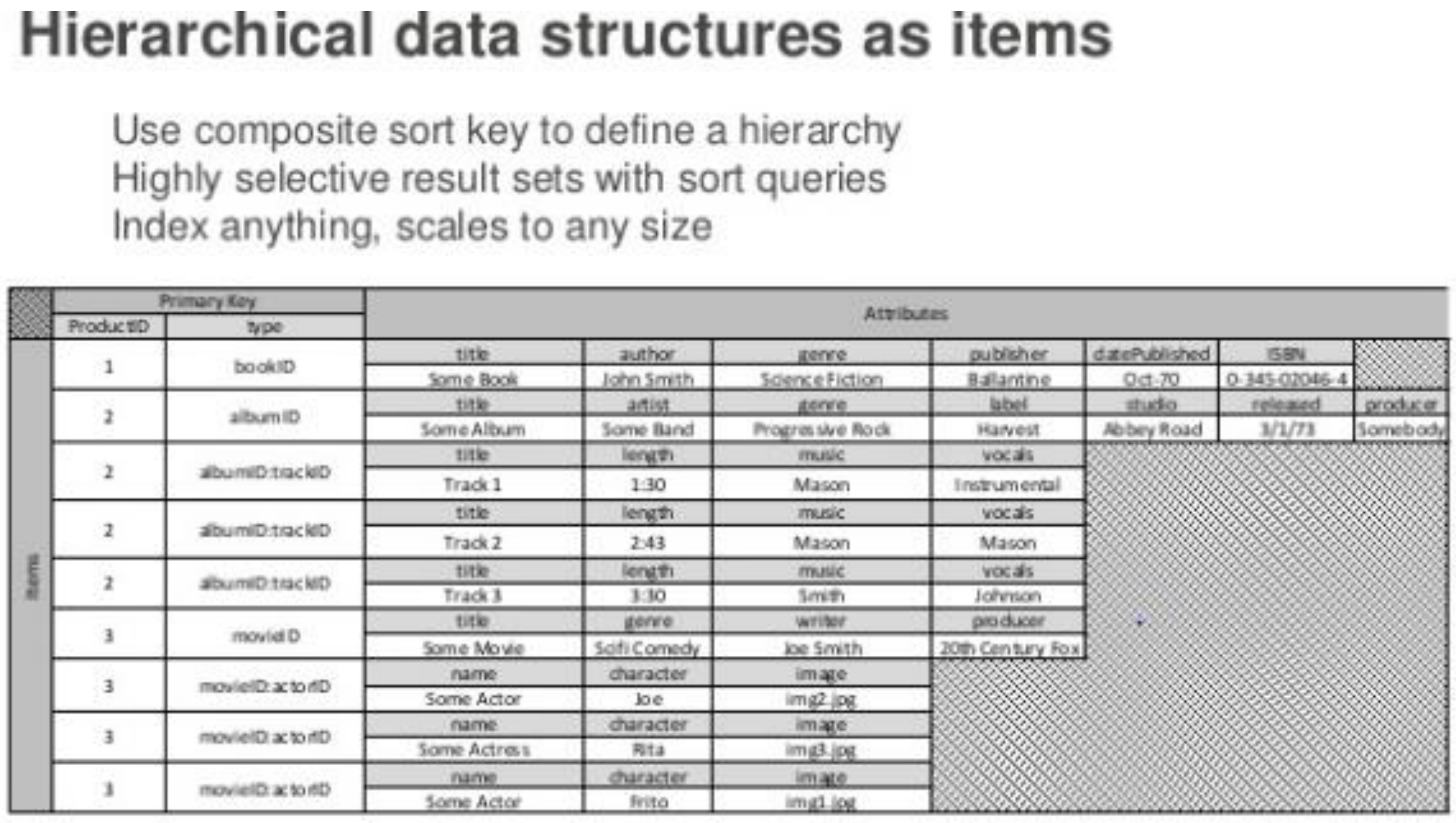
Hash Key	Range key
UserId = bob	GameId = Game1
UserId = fred	GameId = Game2
UserId = bob	GameId = Game3

Game-Users-GSI

Hash Key	Range key
GameId = Game1	UserId = bob
GameId = Game2	UserId = fred
GameId = Game3	UserId = bob

Data Modeling

- Hierarchical Data



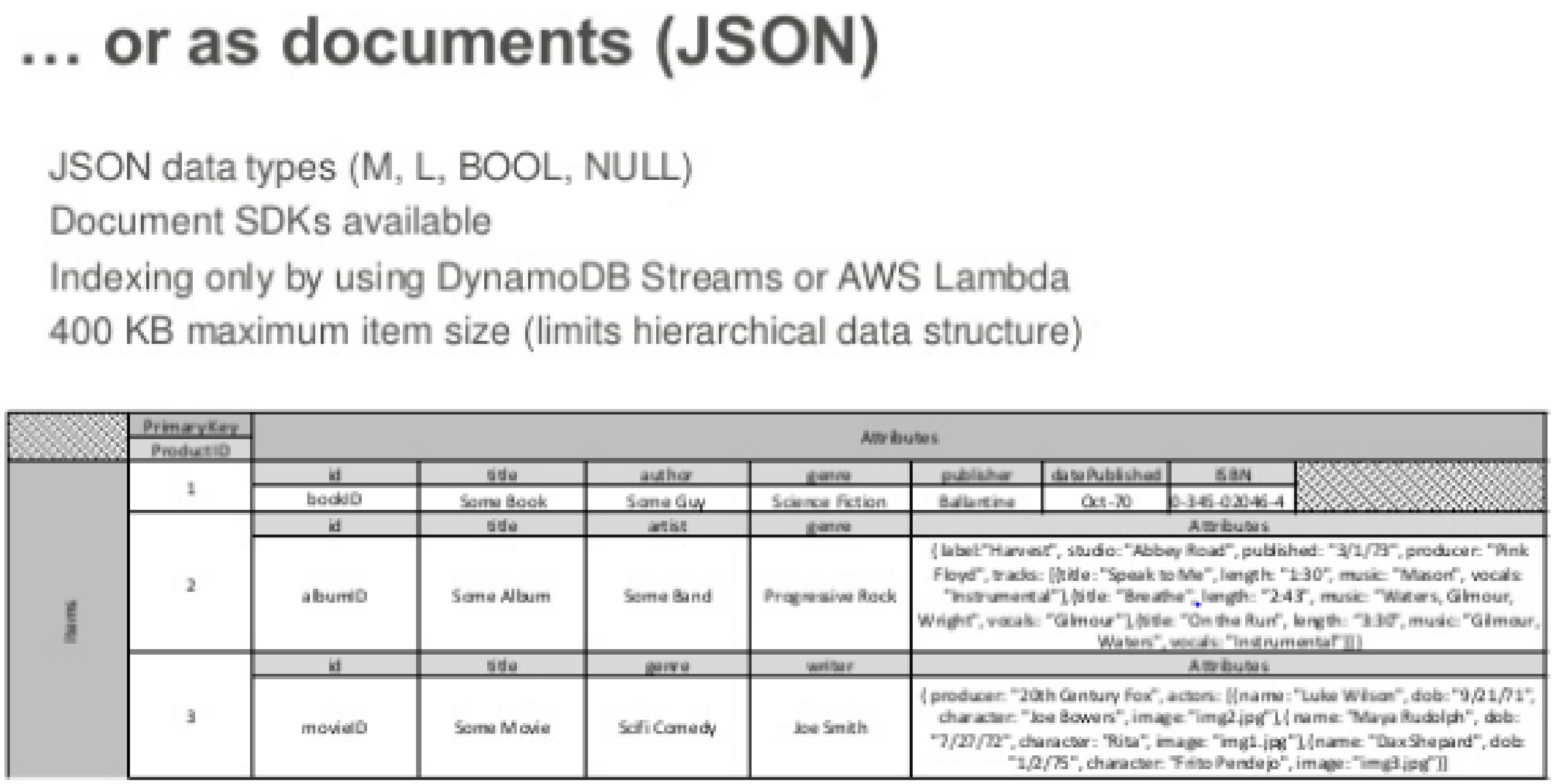
Hierarchical data structures as items

Use composite sort key to define a hierarchy
Highly selective result sets with sort queries
Index anything, scales to any size

	Primary Key		Attributes						
	ProductID	type							
Items	1	bookID	title	author	genre	publisher	datePublished	ISBN	
			Some Book	John Smith	Science Fiction	Ballantine	Oct-70	0-345-02046-4	
	2	albumID	title	artist	genre	label	studio	released	producer
			Some Album	Some Band	Progressive Rock	Harvest	Abbey Road	3/1/73	Somebody
	2	albumID:trackID	title	length	music	vocals			
			Track 1	1:30	Mason	Instrumental			
	2	albumID:trackID	title	length	music	vocals			
			Track 2	2:43	Mason	Mason			
	2	albumID:trackID	title	length	music	vocals			
			Track 3	3:30	Smith	Johnson			
	3	movieID	title	genre	writer	producer			
			Some Movie	Scifi Comedy	Joe Smith	20th Century Fox			
	3	movieID:actorID	name	character	image				
			Some Actor	Joe	img2.jpg				
	3	movieID:actorID	name	character	image				
			Some Actress	Rita	img3.jpg				
	3	movieID:actorID	name	character	image				
			Some Actor	Frito	img1.jpg				

Data Modeling

- Hierarchical Data



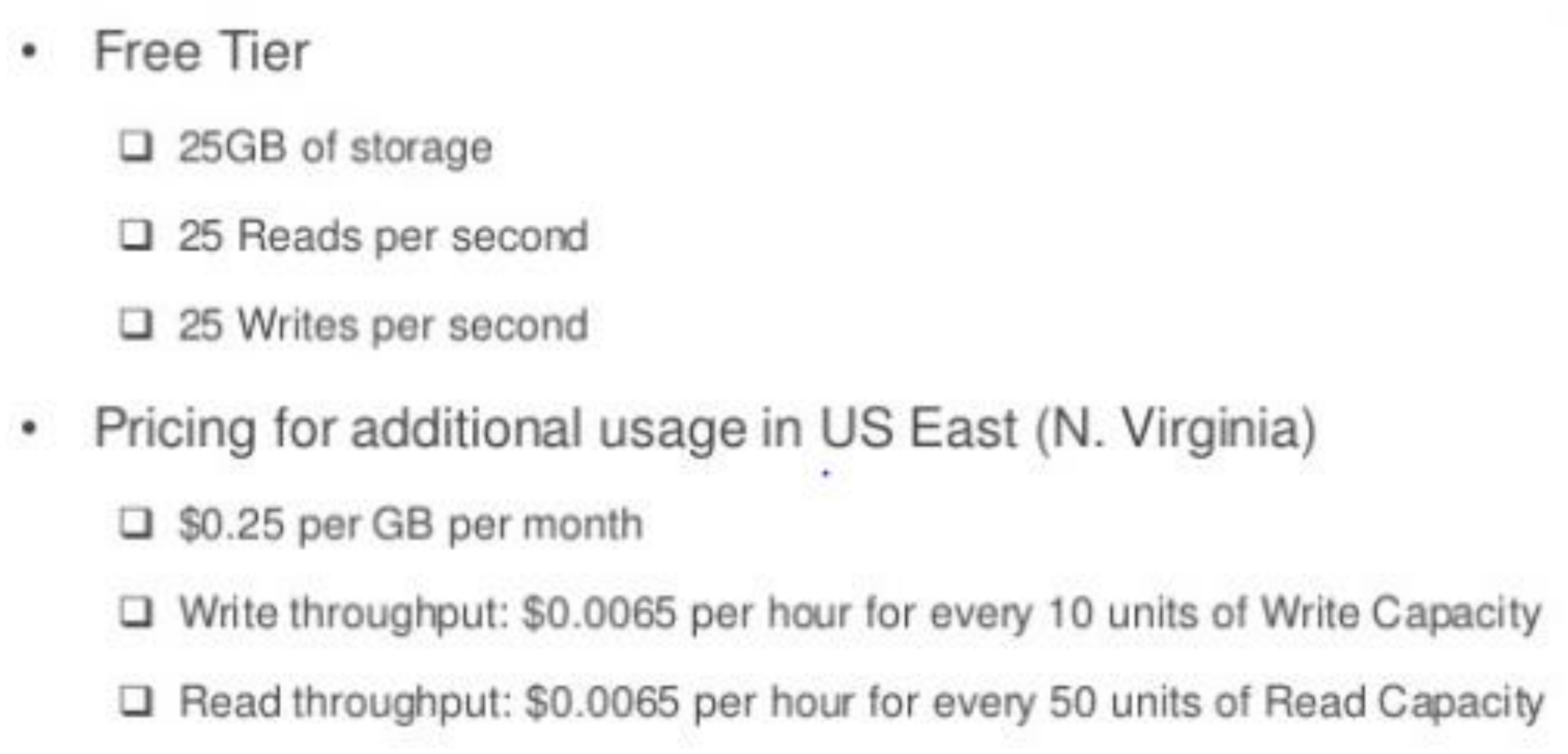
… or as documents (JSON)

JSON data types (M, L, BOOL, NULL)
Document SDKs available
Indexing only by using DynamoDB Streams or AWS Lambda
400 KB maximum item size (limits hierarchical data structure)

▨	PrimaryKey	Attributes							
▨	ProductID								
Items	1	id	title	author	genre	publisher	datePublished	ISBN	▨
		bookID	Some Book	Some Guy	Science Fiction	Ballantine	Oct-70	0-345-02046-4	▨
	2	id	title	artist	genre	Attributes			
		albumID	Some Album	Some Band	Progressive Rock	{ label:"Harvest", studio:"Abbey Road", published: "3/1/73", producer: "Pink Floyd", tracks: [{title: "Speak to Me", length: "1:30", music: "Mason", vocals: "Instrumental"},{title: "Breathe", length: "2:43", music: "Waters, Gilmour, Wright", vocals: "Gilmour"},{title: "On the Run", length: "3:30", music: "Gilmour, Waters", vocals: "Instrumental"}]}			
	3	id	title	genre	writer	Attributes			
		movieID	Some Movie	SciFi Comedy	Joe Smith	{ producer: "20th Century Fox", actors: [{name: "Luke Wilson", dob: "9/21/71", character: "Joe Bowers", image: "img2.jpg"},{ name: "Maya Rudolph", dob: "7/27/72", character: "Rita", image: "img1.jpg"},{name: "Dax Shepard", dob: "1/2/75", character: "Frito Pendejo", image: "img3.jpg"}]			

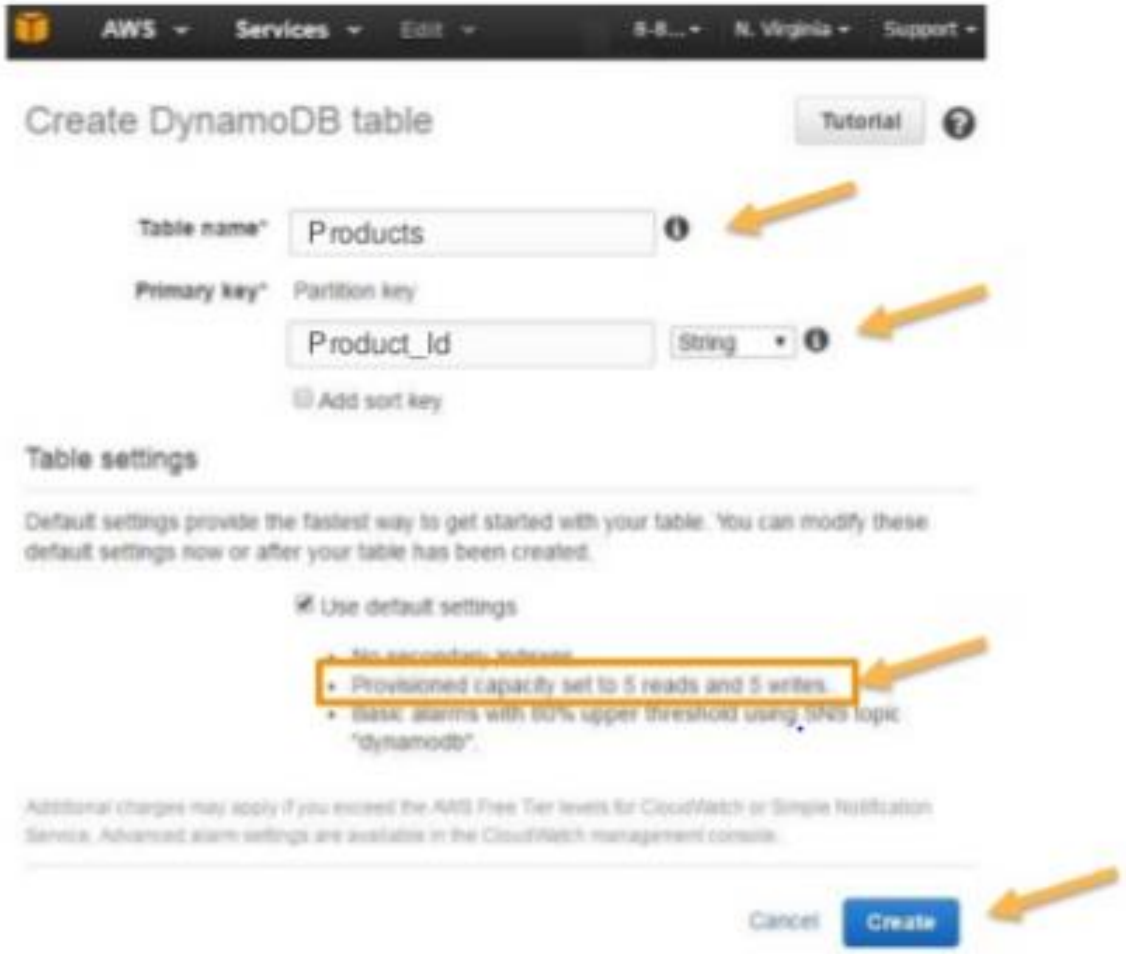
Pricing Model

- DynamoDB Pricing and Free Tier



- Free Tier
  - ❑ 25GB of storage
  - ❑ 25 Reads per second
  - ❑ 25 Writes per second
- Pricing for additional usage in US East (N. Virginia)
  - ❑ $0.25 per GB per month
  - ❑ Write throughput: $0.0065 per hour for every 10 units of Write Capacity
  - ❑ Read throughput: $0.0065 per hour for every 50 units of Read Capacity

- Creating Table in DynamoDB



Create DynamoDB table

Table name* Products

Primary key* Partition key

Product_Id String

☐ Add sort key

Table settings

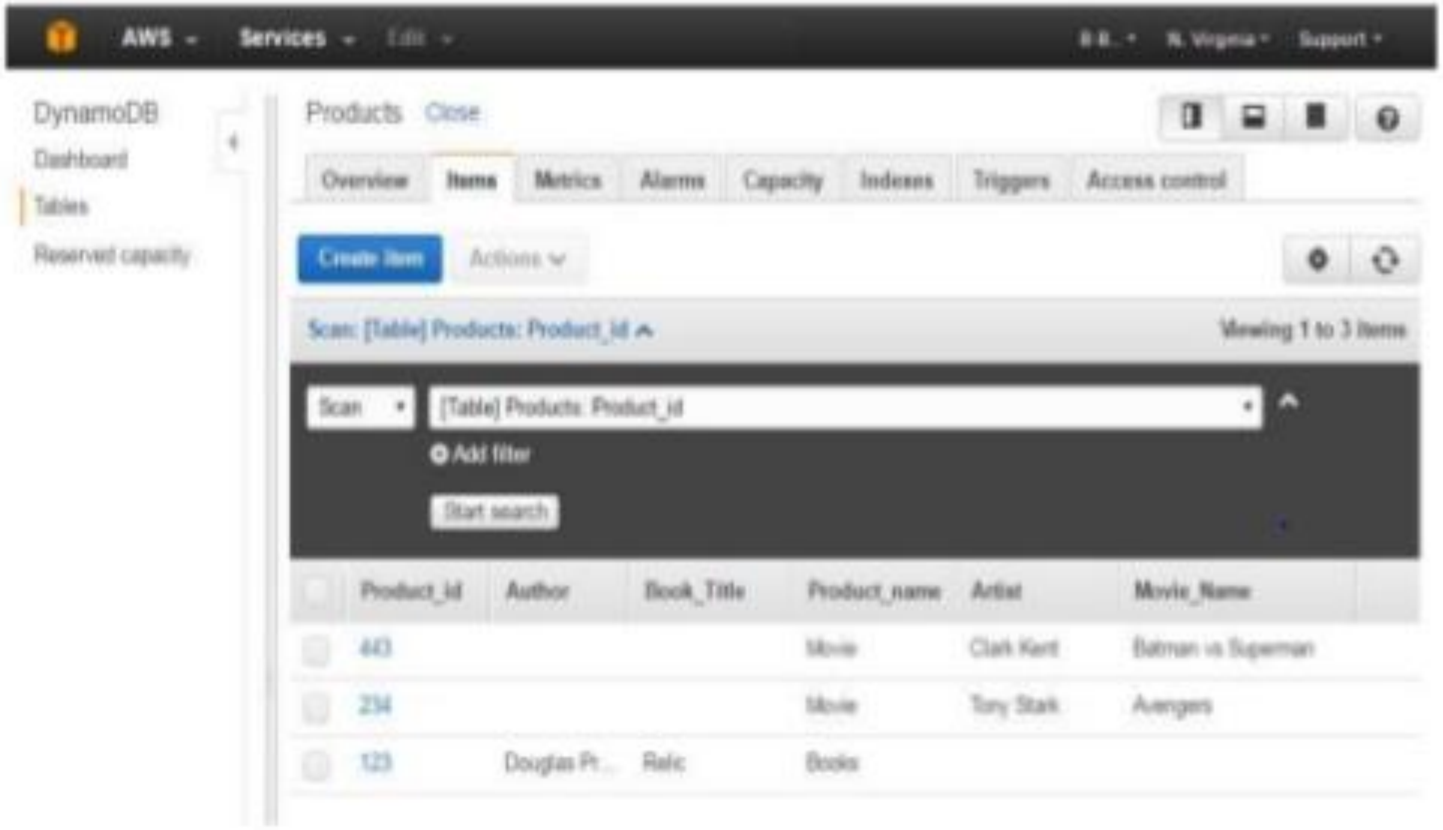
Default settings provide the fastest way to get started with your table. You can modify these default settings now or after your table has been created.

☑ Use default settings

- No secondary indexes.
- Provisioned capacity set to 5 reads and 5 writes.
- Basic alarms with 80% upper threshold using SNS topic "dynamodb".

Additional charges may apply if you exceed the AWS Free Tier levels for CloudWatch or Simple Notification Service. Advanced alarm settings are available in the CloudWatch management console.

- Creating Table in DynamoDB



Scan: [Table] Products: Product_id

Viewing 1 to 3 items

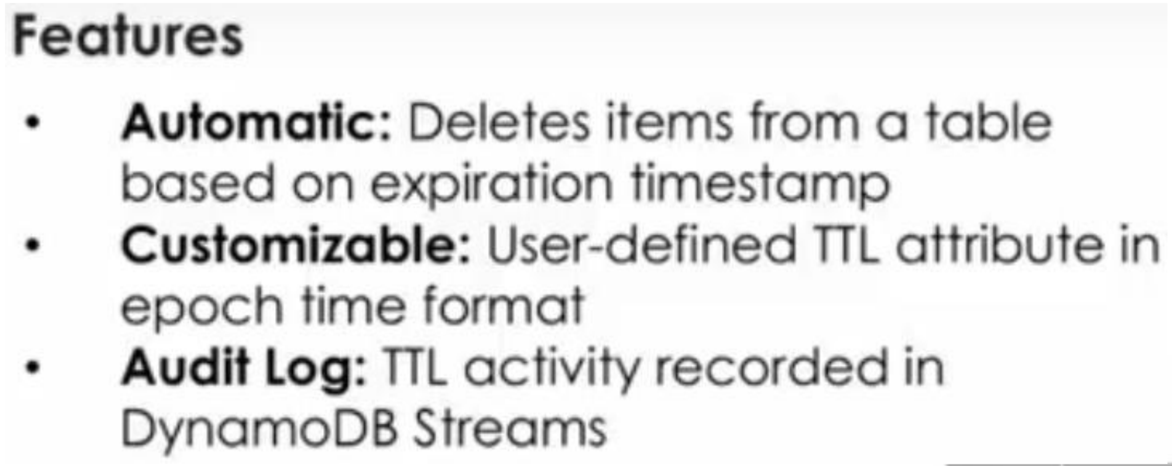
Scan | [Table] Products: Product_id

Product_id	Author	Book_Title	Product_name	Artist	Movie_Name
443			Movie	Clark Kent	Batman vs Superman
234			Movie	Tony Stark	Avengers
123	Douglas Pr...	Relic	Books		

Time-to-Live (TTL)

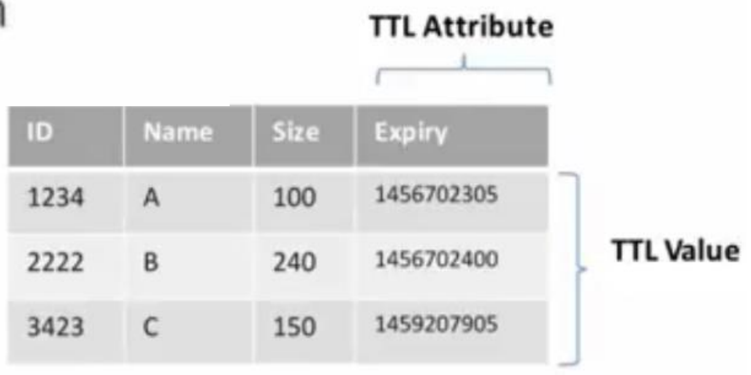
Time-to-Live (TTL)

- TTL is a feature that offers the ability to expire your data when it's not needed in DynamoDB
  - On expiration, DynamoDB will automatically delete expired data



Features

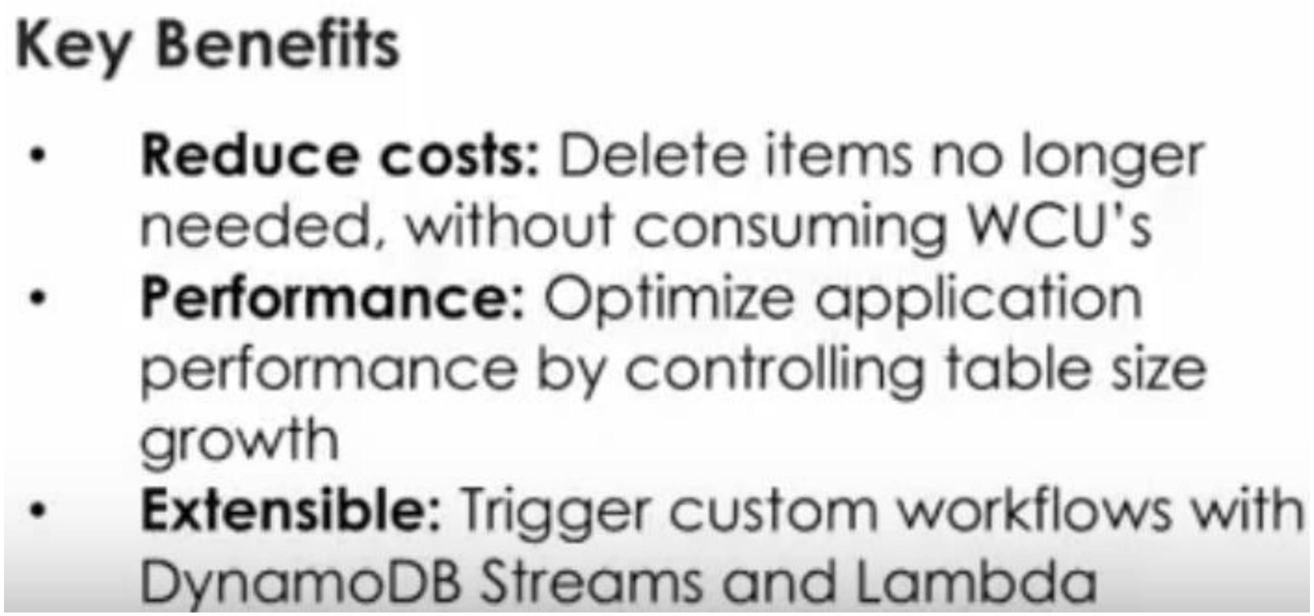
- **Automatic:** Deletes items from a table based on expiration timestamp
- **Customizable:** User-defined TTL attribute in epoch time format
- **Audit Log:** TTL activity recorded in DynamoDB Streams


TTL Attribute
TTL Value

ID	Name	Size	Expiry
1234	A	100	1456702305
2222	B	240	1456702400
3423	C	150	1459207905

Benefits of TTL

- TTL allows you to manage the life cycle of data in DynamoDB
  - Helps reduce costs by deleting items (that are no longer needed) without consuming WCU
  - If you were to delete data using DeleteItem operation, that will consume WCU
    - But deleting based on TTL does not

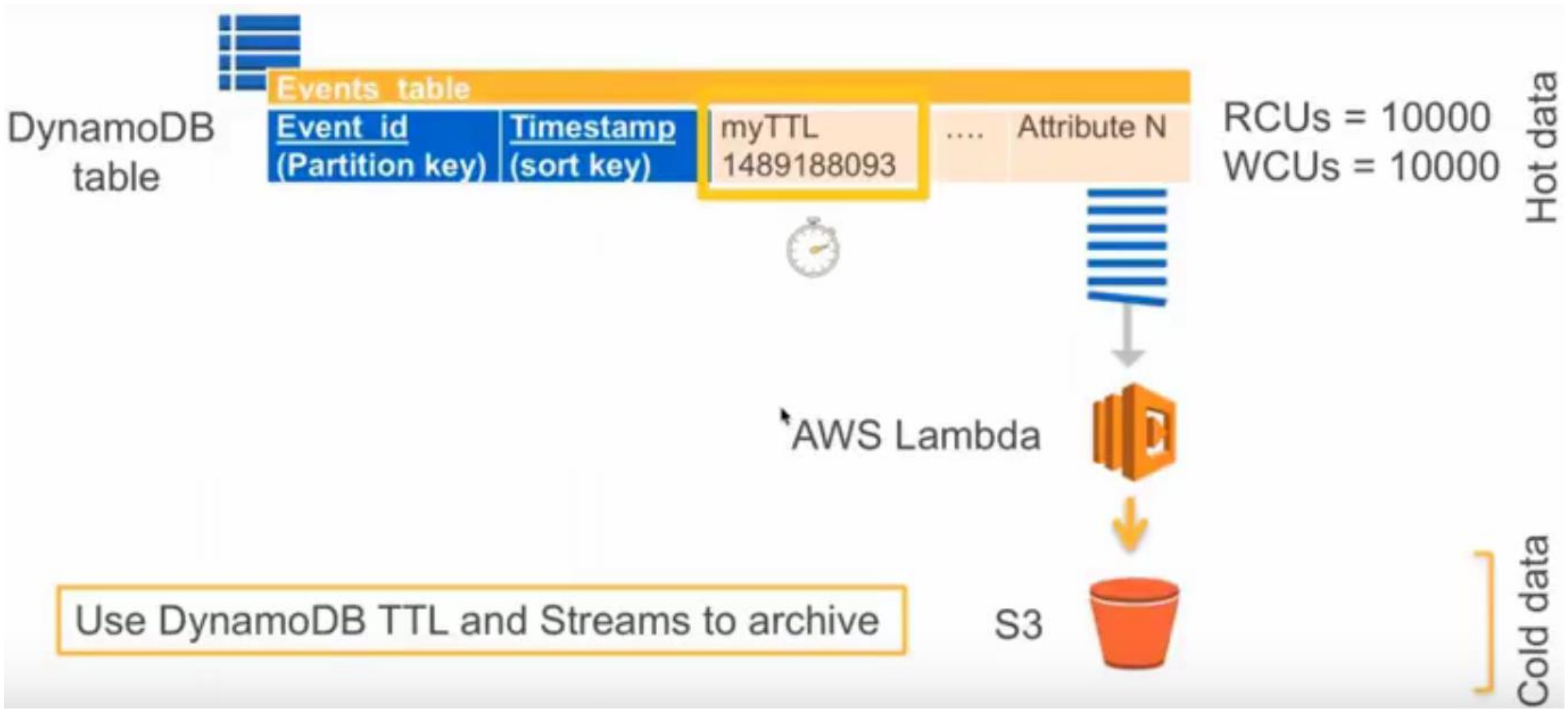


Key Benefits

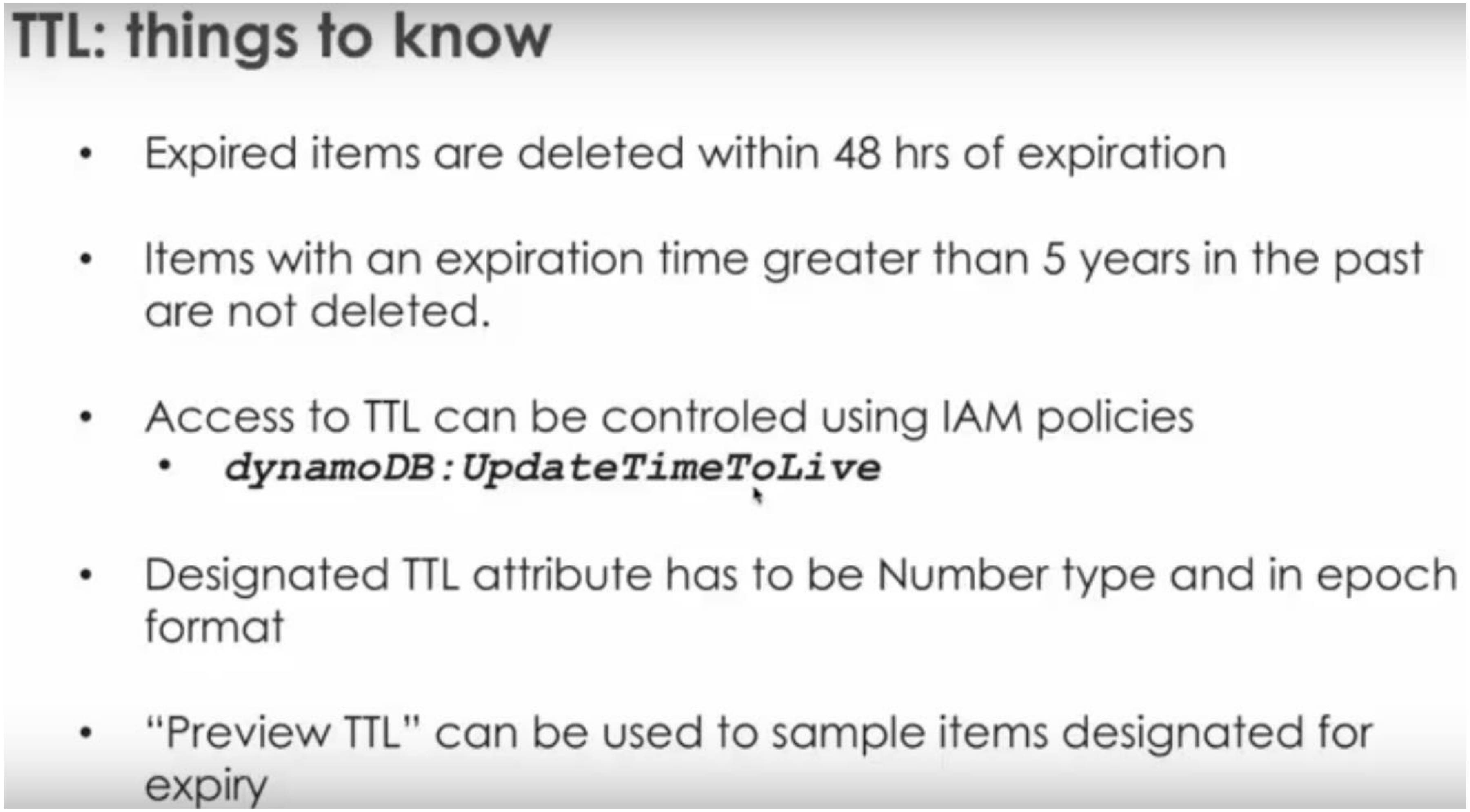
- **Reduce costs:** Delete items no longer needed, without consuming WCU's
- **Performance:** Optimize application performance by controlling table size growth
- **Extensible:** Trigger custom workflows with DynamoDB Streams and Lambda

Using TTL to age out cold data

- Deletion events caused by TTL can be filtered on DynamoDB streams and used to post process TTL deleted data,
  - e.g., to archive the data that were deleted by TTL into S3



Things to know about TTL



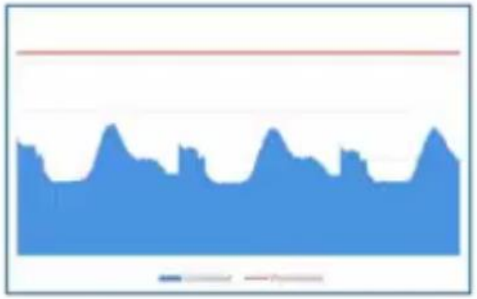
TTL: things to know

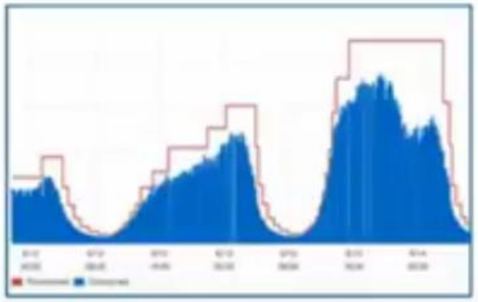
- Expired items are deleted within 48 hrs of expiration
- Items with an expiration time greater than 5 years in the past are not deleted.
- Access to TTL can be controled using IAM policies
  - ***dynamoDB:UpdateTimeToLive***
- Designated TTL attribute has to be Number type and in epoch format
- "Preview TTL" can be used to sample items designated for expiry

Auto Scaling in DyanmoDB

Auto Scaling in DynamoDB

- Auto Scaling makes it easy to ensure that your tables have enough capacity (WCU, RCU) when they need it
  - It reduces the cost by reducing the capacity when it's not needed
- When creating new tables, the auto scaling of WCU and RCU is enabled by default
  - Auto scaling capacity set to 70% target utilization (that is consumed capacity should be at the 70% of provisioned capacity


Without Auto Scaling


With Auto Scaling

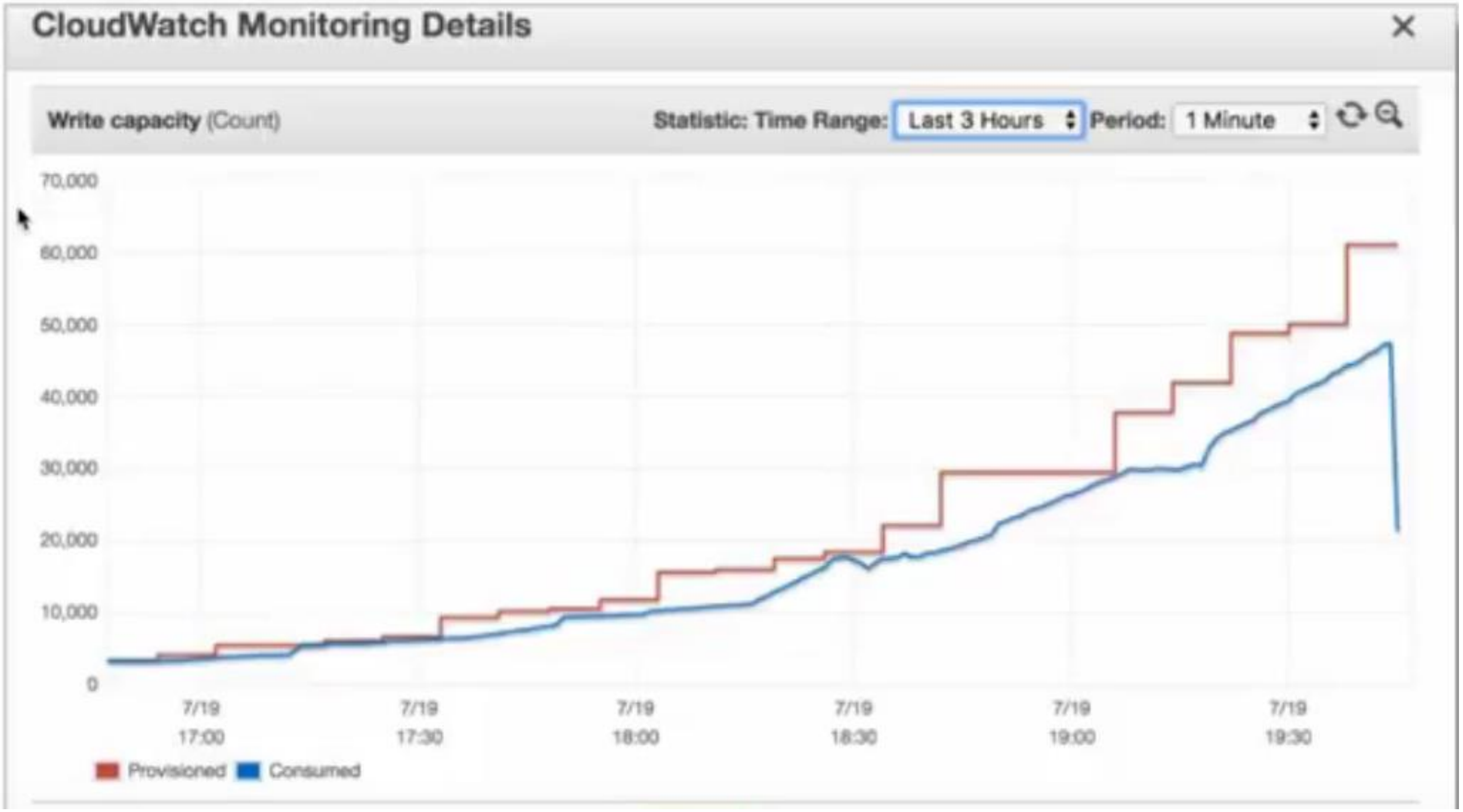
Features

- Fully managed, automatic, independent scaling of read and write capacity of base tables and global secondary indexes
- Set only target utilization % and min/max limits
- Accessed from management console, CLI, and SDK

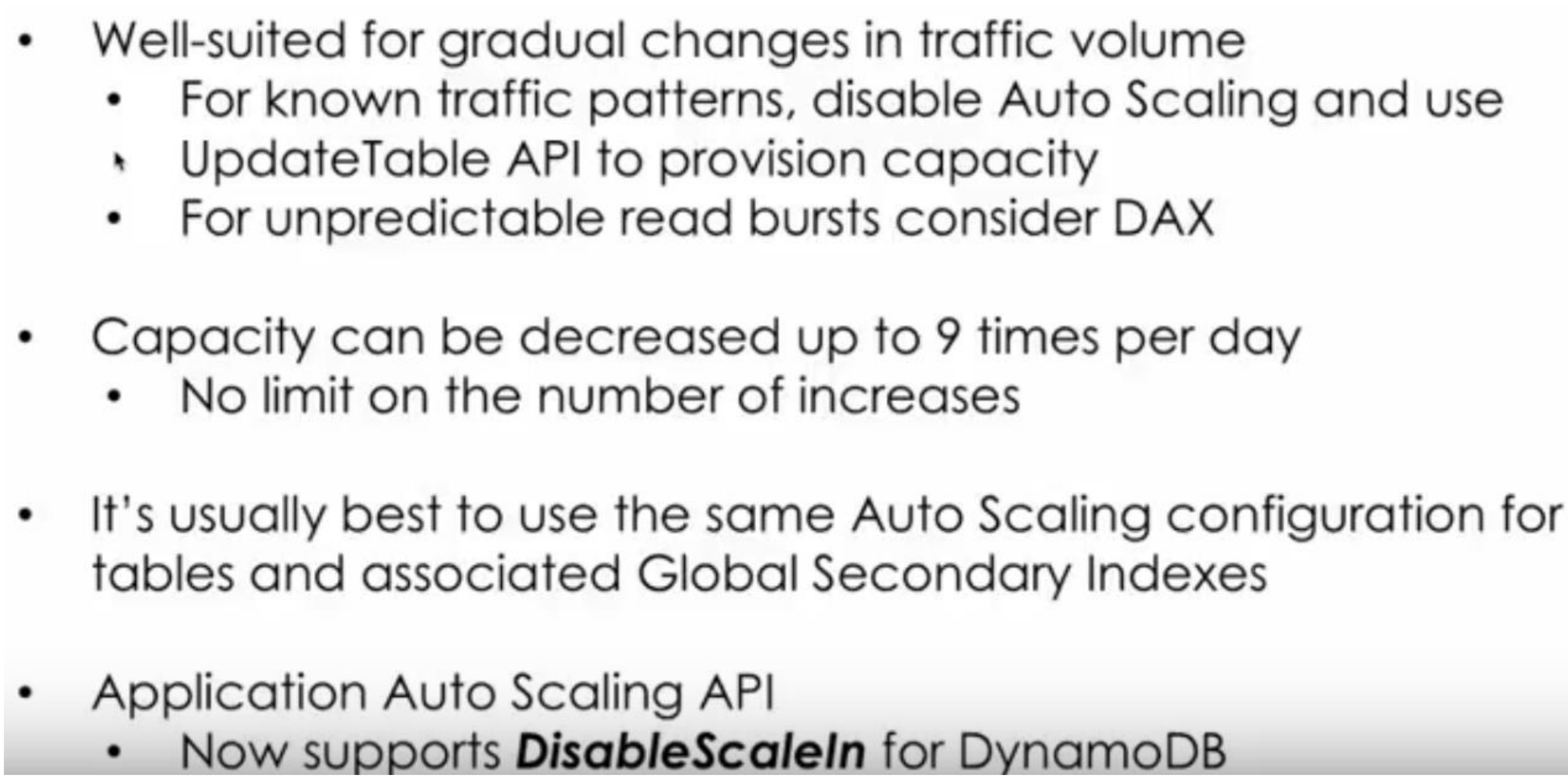
Key Benefits

- Remove the guesswork out of provisioning adequate capacity
- Increases capacity as application requests increase, ensuring performance
- Decreases capacity as application requests reduce, reducing costs
- Full visibility into scaling activities from console

Auto-Scaling Example



Auto Scaling – Things to know



- Well-suited for gradual changes in traffic volume
  - For known traffic patterns, disable Auto Scaling and use UpdateTable API to provision capacity
  - For unpredictable read bursts consider DAX

- Capacity can be decreased up to 9 times per day
  - No limit on the number of increases

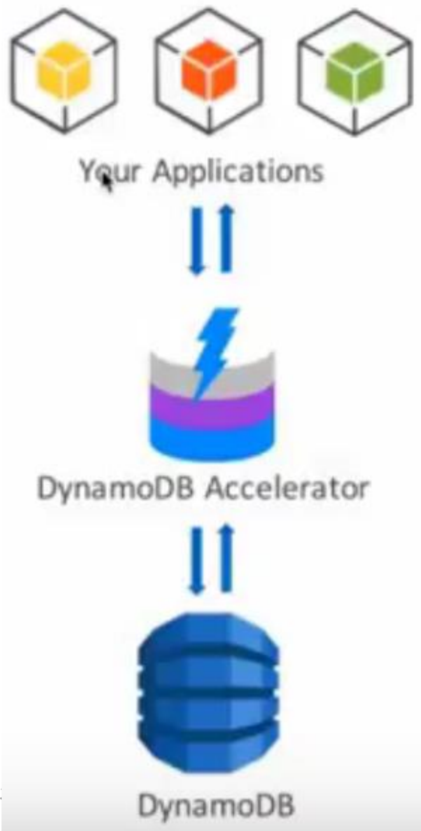
- It's usually best to use the same Auto Scaling configuration for tables and associated Global Secondary Indexes

- Application Auto Scaling API
  - Now supports ***DisableScaleIn*** for DynamoDB

DynamoDB Accelerator (DAX)

DynamoDB Accelerator (DAX)

- DAX is a fully managed front end cache for DynamoDB
  - It targets read use cases for DynamoDB, e.g., read performance
  - Sub milliseconds response time



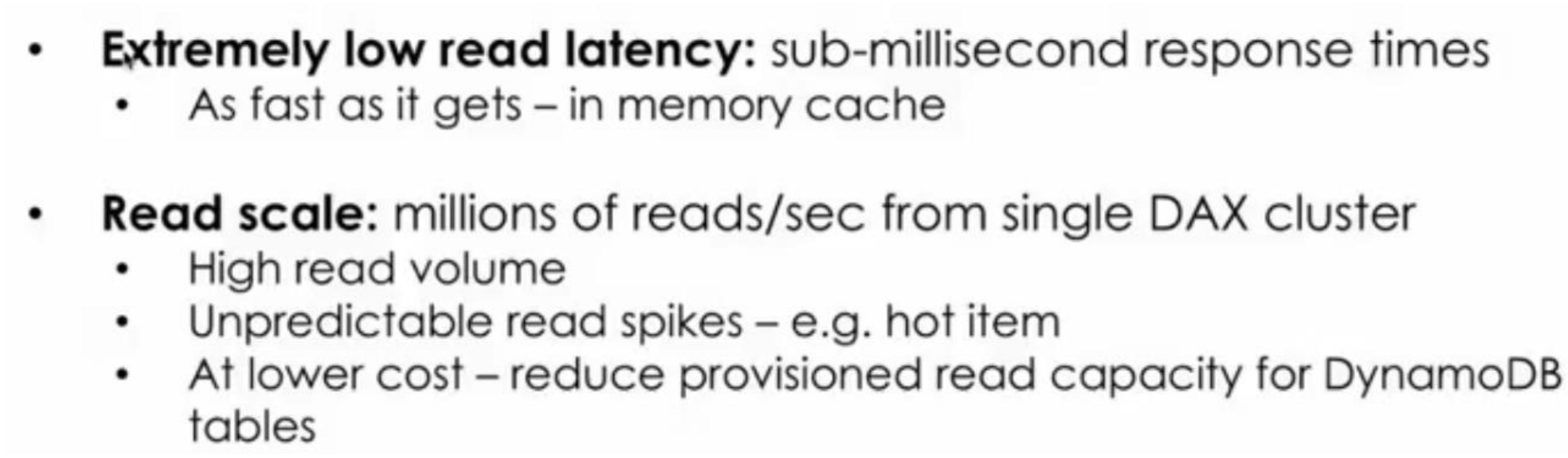
Key Benefits

- **Read performance and scale:** Microseconds response times at millions of reads/sec from single DAX cluster
- **Lower costs:** Reduce provisioned read capacity for DynamoDB tables for tables with hot data

Features

- Fully managed, highly available
- DynamoDB API compatible
- Write-through
- Flexible – use for one or multiple tables
- Scales-out up to 10 read replicas
- Fully integrated AWS service
- Secure

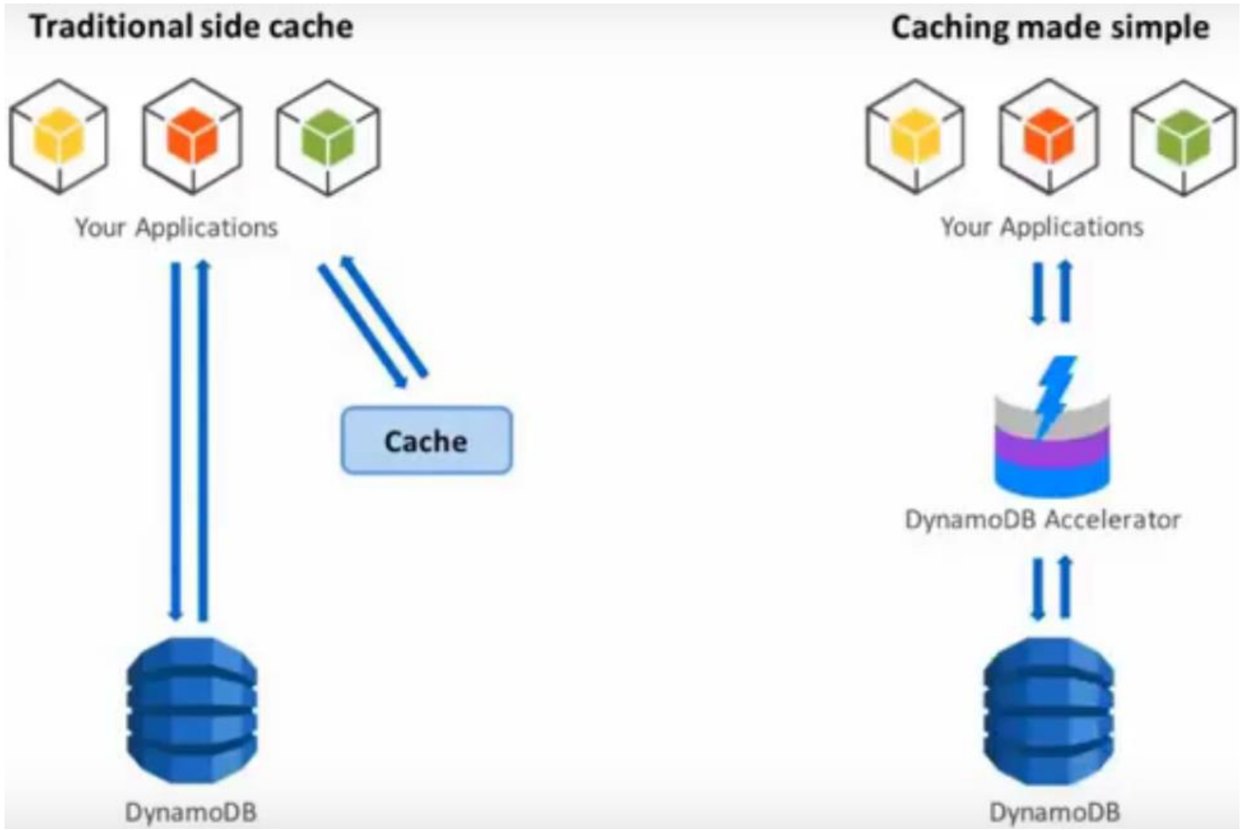
Use DAX if you need……



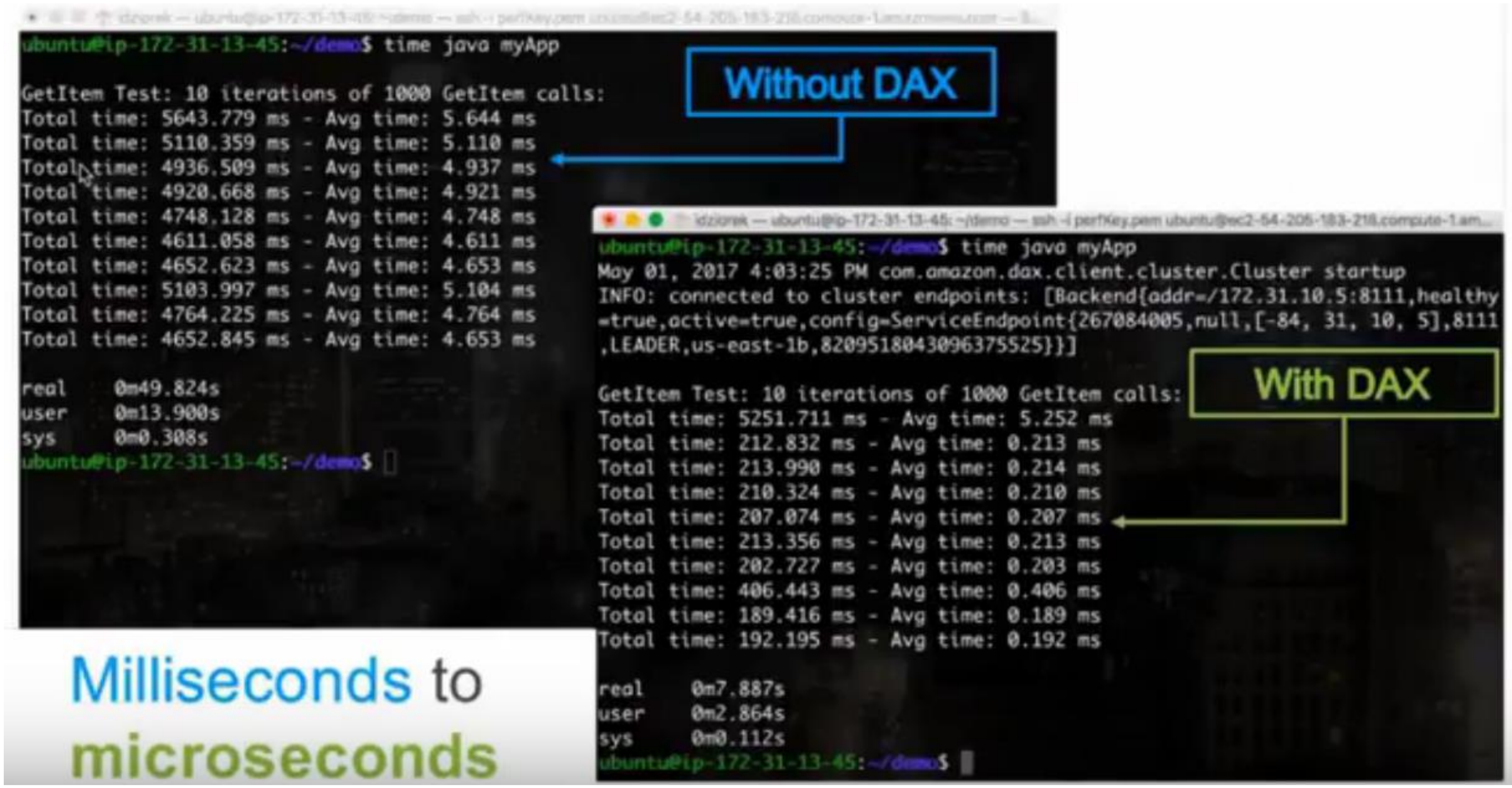
- **Extremely low read latency:** sub-millisecond response times
  - As fast as it gets – in memory cache
- **Read scale:** millions of reads/sec from single DAX cluster
  - High read volume
  - Unpredictable read spikes – e.g. hot item
  - At lower cost – reduce provisioned read capacity for DynamoDB tables

Comparing DAX with traditional Side Cache

- Traditional side cache requires applications to use both Cache and Database APIs
- On the other hand, DAX is Inline Cache and Write Through Cache as well
  - DAX API is the only API you need to use when reading/writing data from DynamoDB through DAX
  - This simplifies application development



Read Performance

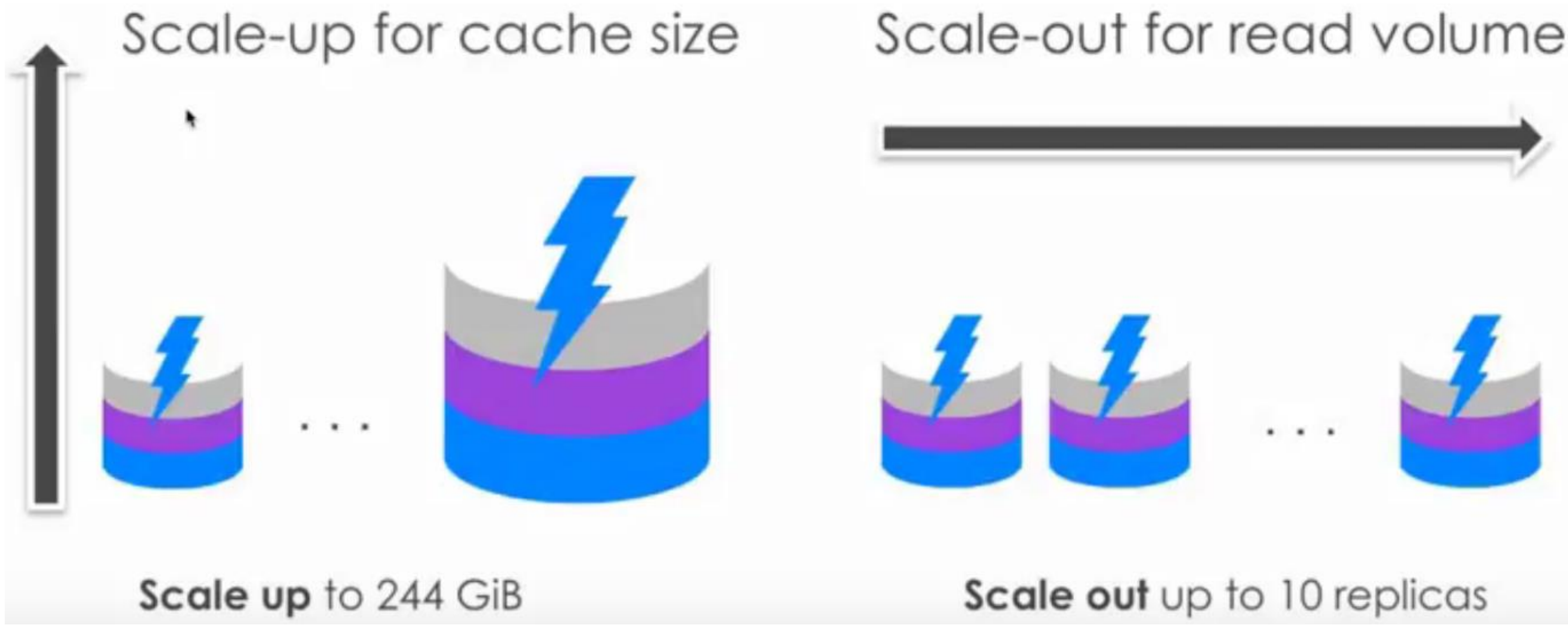


```
ubuntu@ip-172-31-13-45:~/demo$ time java myApp

GetItem Test: 10 iterations of 1000 GetItem calls:
Total time: 5643.779 ms - Avg time: 5.644 ms
Total time: 5110.359 ms - Avg time: 5.110 ms
Total time: 4936.509 ms - Avg time: 4.937 ms
Total time: 4920.668 ms - Avg time: 4.921 ms
Total time: 4748.128 ms - Avg time: 4.748 ms
Total time: 4611.058 ms - Avg time: 4.611 ms
Total time: 4652.623 ms - Avg time: 4.653 ms
Total time: 5103.997 ms - Avg time: 5.104 ms
Total time: 4764.225 ms - Avg time: 4.764 ms
Total time: 4652.845 ms - Avg time: 4.653 ms

real    0m49.824s
user    0m13.900s
sys     0m0.308s
ubuntu@ip-172-31-13-45:~/demo$
```

```
ubuntu@ip-172-31-13-45:~/demo$ time java myApp
May 01, 2017 4:03:25 PM com.amazon.dax.client.cluster.Cluster startup
INFO: connected to cluster endpoints: [Backend{addr=/172.31.10.5:8111,healthy
=true,active=true,config=ServiceEndpoint{267084005,null,[-84, 31, 10, 5],8111
,LEADER,us-east-1b,8209518043096375525}}]

GetItem Test: 10 iterations of 1000 GetItem calls:
Total time: 5251.711 ms - Avg time: 5.252 ms
Total time: 212.832 ms - Avg time: 0.213 ms
Total time: 213.990 ms - Avg time: 0.214 ms
Total time: 210.324 ms - Avg time: 0.210 ms
Total time: 207.074 ms - Avg time: 0.207 ms
Total time: 213.356 ms - Avg time: 0.213 ms
Total time: 202.727 ms - Avg time: 0.203 ms
Total time: 406.443 ms - Avg time: 0.406 ms
Total time: 189.416 ms - Avg time: 0.189 ms
Total time: 192.195 ms - Avg time: 0.192 ms

real    0m7.887s
user    0m2.864s
sys     0m0.112s
ubuntu@ip-172-31-13-45:~/demo$
```

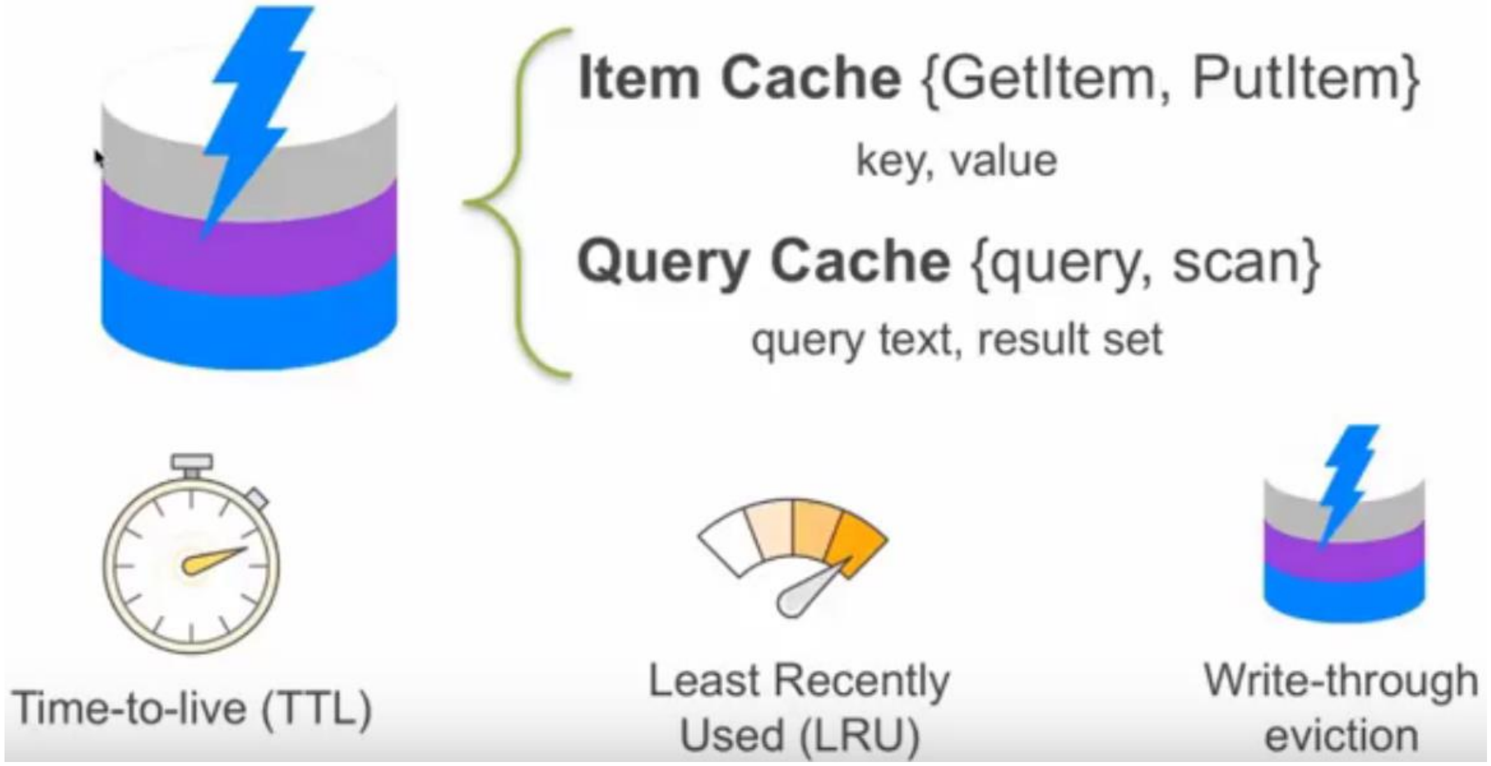
Scalability in DAX

- Scale Up for Cache Size
    - The size of cache is controlled by the selection of instance, which can scale up to 244 GiB (Giga Bytes)
- Scale out for Read Volume
    - Supports scaling out up to ten read replicas



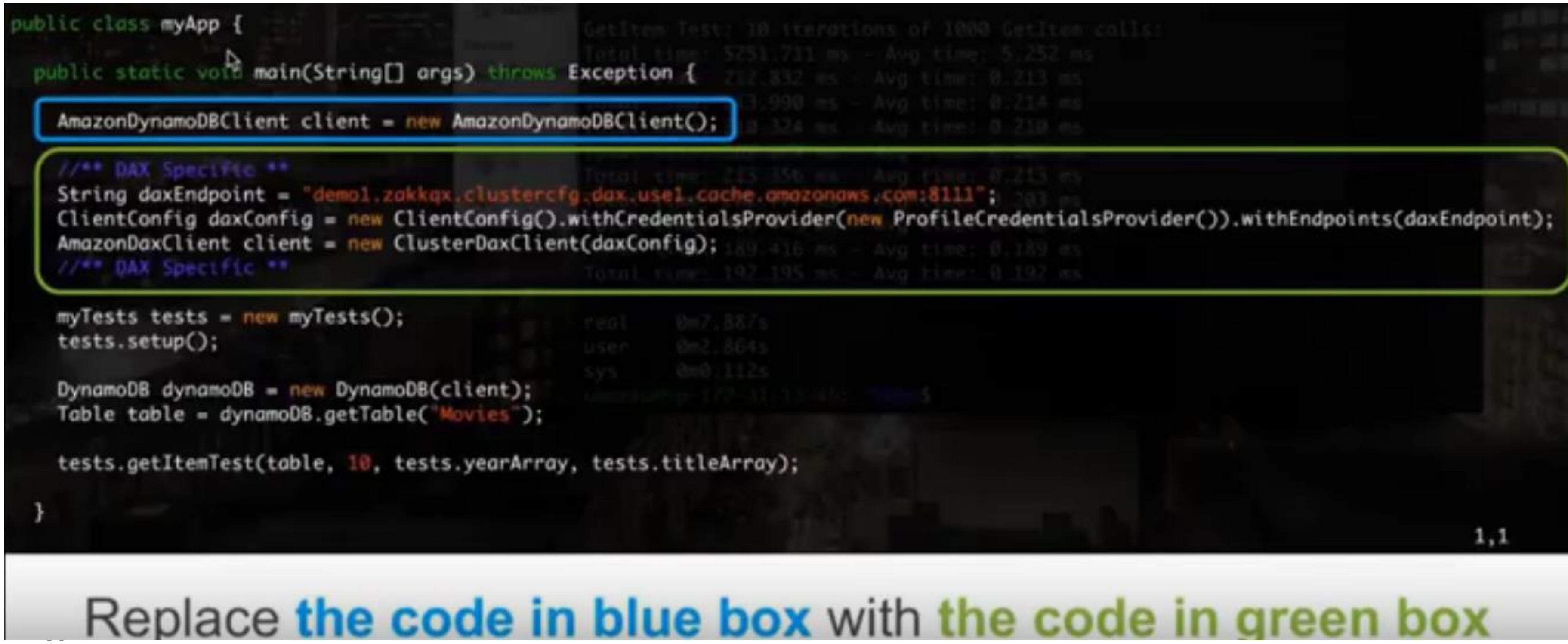
Caches and Eviction in DAX

- DAX Supports Item Cache and Query Cache
  - Item Cache services GetItem and PutItem requests
  - Query Cache serves query and scan calls
- Eviction and lifecycle of data cached in DAX managed by configuring TTL for data cached in DAX
  - DAX uses LRU (Least Recently Used) algorithm and Write-through eviction as well


Item Cache {GetItem, PutItem}
key, value
Query Cache {query, scan}
query text, result set
Time-to-live (TTL)
Least Recently Used (LRU)
Write-through eviction

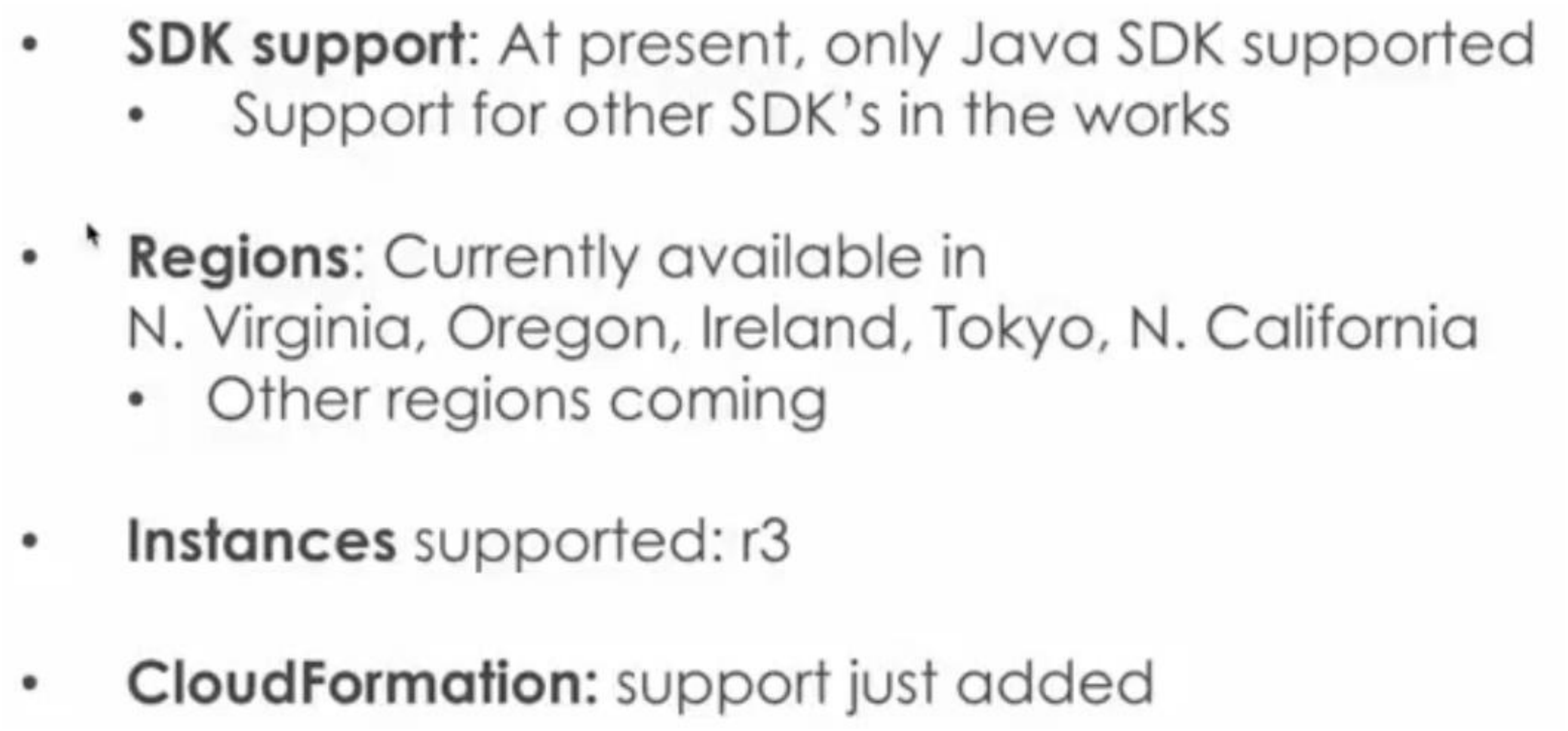
Integrating DAX in your exsting DynamoDB applications

- AmazonDAXClient API is compatible with AmazonDynamoDBClient APIs
- Replace the code in blue box with the code in green box
  - No other changes are required



```
public class myApp {

  public static void main(String[] args) throws Exception {

    AmazonDynamoDBClient client = new AmazonDynamoDBClient();

    //** DAX Specific **
    String daxEndpoint = "demo1.zakkqx.clustercfg.dax.use1.cache.amazonaws.com:8111";
    ClientConfig daxConfig = new ClientConfig().withCredentialsProvider(new ProfileCredentialsProvider()).withEndpoints(daxEndpoint);
    AmazonDaxClient client = new ClusterDaxClient(daxConfig);
    //** DAX Specific **

    myTests tests = new myTests();
    tests.setup();

    DynamoDB dynamoDB = new DynamoDB(client);
    Table table = dynamoDB.getTable("Movies");

    tests.getItemTest(table, 10, tests.yearArray, tests.titleArray);

  }
```

DAX: Things to Know

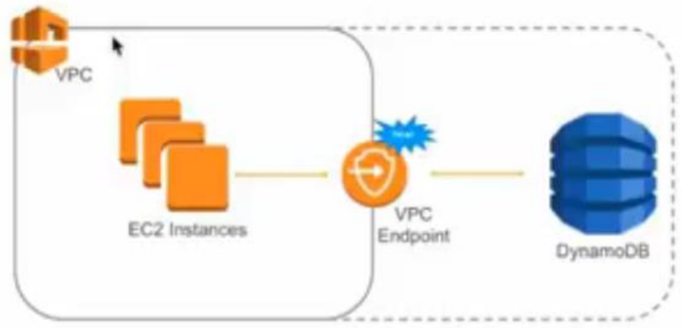


- **SDK support**: At present, only Java SDK supported
  - Support for other SDK's in the works
- **Regions**: Currently available in N. Virginia, Oregon, Ireland, Tokyo, N. California
  - Other regions coming
- **Instances** supported: r3
- **CloudFormation:** support just added

VPC Endpoints for DynamoDB

VPC Endpoints for DynamoDB

- VPC Endpoints enable access to AWS Services (e.g., to DynamoDB in this case) via secure and private Amazon VPC connections
  - It does not leave Amazon network



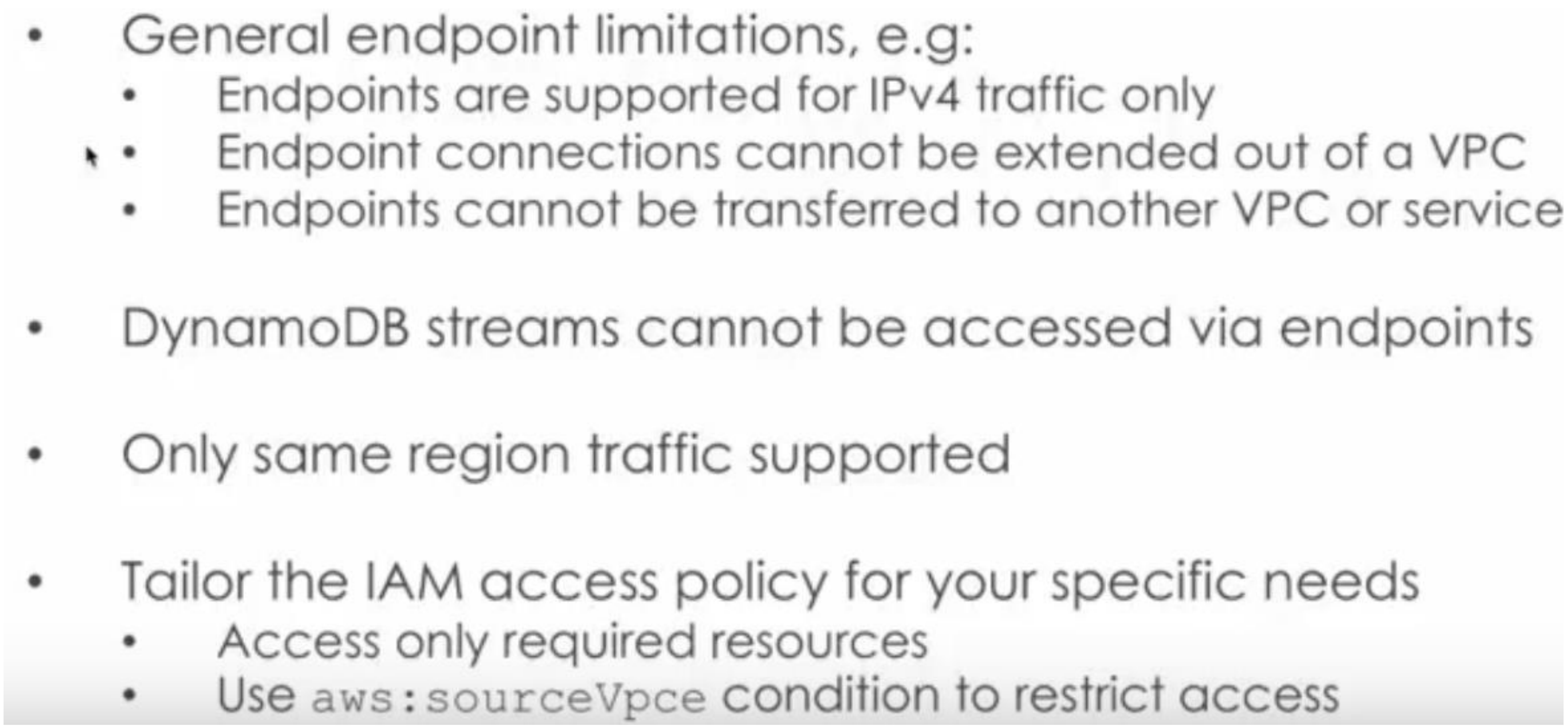
Features

- Access DynamoDB via secure Amazon VPC endpoint
- Customize access for each VPC endpoint with unique IAM role and permissions

Key Benefits

- Turn off access from public Internet gateways enhancing privacy and security
- Secure data transfer between Amazon VPC and DynamoDB without IGW or NATGW
- Simplified network configuration
- Cost savings – no extra charges

VPC E – things to know



- General endpoint limitations, e.g:
  - Endpoints are supported for IPv4 traffic only
  - Endpoint connections cannot be extended out of a VPC
  - Endpoints cannot be transferred to another VPC or service
- DynamoDB streams cannot be accessed via endpoints
- Only same region traffic supported
- Tailor the IAM access policy for your specific needs
  - Access only required resources
  - Use `aws:sourceVpce` condition to restrict access