Key Focus Areas (Ballpark)

* RDS (25%)
* Aurora (25%)
* DynamoDB (20%)
* Database Migration (20%)
* Automation (5%)
* Others (5%)

**Purpose-Built Databases**

* Databases used to be a “one size-fits-all” use a relational database and off you go
* New Challenges
  + With the advent of big data, demands on volume, velocity the variety of data has also increased
  + We now store TeraBytes to PentaBytes of data
  + Millions of requests per seconds
  + Sub-millisecond latencies and global users
* Although relational databases are here to stay ( and are evolving), they alone aren’t enough to cater to our data needs today
* In modern applications, it’s common to see a mix of databases
* Finding the using databases fit for purpose – find the right database for the right job

**Choosing the Right Database**

* We Have a lot of managed databases on AWS to choose from
* The question then becomes what is the right database based on your architecture:
  + Read-Heavy, Write-Heavy, or balanced workload?
  + Throughput needs?
  + Will it change, does it need to scale or fluctuate throughout the day?
  + How much data to store and for how long?
  + Will the size of the data grow?
  + Average object size?
  + How are they accessed?
  + Data Durability?
  + Source of truth for the data?
  + Latency Requirements?
  + Concurrent Users?
  + Data model?
  + How does one query the data? (Joins, structured, semi-structured)
  + Strong Schema? More Flexible? Reporting? Search? RDBMS / NoSQL
  + License Costs? Switch to Cloud-Native DB such as Aurora

Purpose-Built Databases from AWS

* Relational
  + Amazon RDS
  + Amazon Aurora
  + Amazon Redshift
* Key-Value
  + DynamoDB
  + DAX
* Document
  + DocumentDB
* In-Memory
  + Elasticache
  + ElastiCache for Redis
  + ElastiCache for Memcached
* Graph
  + Neptune
* Search
  + Elasticsearch Service
* Time-Series
  + Timestream
* Ledger
  + QLDB
* Database Migration
  + AWS DMS

**The Basics (Mastering the basics is the secret key to success)**

* Data: a collection of values or information
* Database: an organized collection of this data
* Thre types of data
  + Structured
  + Semi-Structured
  + Unstructured
* Hence the need for different databases to suit different types of data
* We often end up working with a mix of these types

Structured Data

* Typically stored in tables with predefined structures (aka: Schema)
* Suitable for OLTP ( transactional ) and OLAP (analytical) workloads
  + OLTP: Online Transactional Processing
  + OLAP: Online Analytical Processing
* Typical of relational databases (indexed tables linked with foreign key relationships)
* Suited for complex queries and analytics e.g complex table join operations via SQL ( structured query language)

Semi-Structured

* Organized data, but constrained by a fixed schema.
  + This means if we wanted to add an email column we would have had to change the schema for a structured database, whereas a semi-structured database we can add it as we like
* Typical of non-relational databases
* Well suited for big data and low-latency applications
* An example would be XML and JSON

Unstructured Data

* Unorganized data
* An example files sitting on your computer
* Typical of non-relational databases, file systems, data stores or data lakes like amazon s3

**Relational Databases**

* Have a predefined schema
* Enforce strict ACID compliance and supports “joins”
* Typical use cases - OLTP and OLAP
* Examples: MySQL, PostgreSQL, MariaDB, ORacle, Microsoft SQL Server
* AWS Services:
  + Amazon RDS and Aurora for OLTP
  + Redshift for OLAP
* Characterized by multiple tables interconnected through foreign key relationships

Table Indexes

* Relational databases are written to and queried using SQL
* For efficient query performance, we add indexes to the table
* Tables are indexed by their primary key by default
* Secondary indexed can be added on non-key fields

ACID Compliant

* ACID = Atomicity, Consistency, Isolation, and Durability
  + Atomicity means “ all or nothing “ – a transaction executes completely or not at all
  + Consistency means once a transaction has been committed, the data must conform to the given schema
  + Isolation requires that concurrent transactions execute separately from one another
  + Durability is the ability to recover from an unexpected system failure or outage
* ACID compliance refers only to the compliance enforced by the DB
* ACID behavior can also be enforced by your application
  + Eg your application can maintain strict foreign key relationship between two DynamoDB tables. However, DynamoDB by itself will not enforce this relationship. Another application can always override this behavior.
  + If for example, it was a MySQL database table with relationships defined on the DB, no application would be able to override the behavior

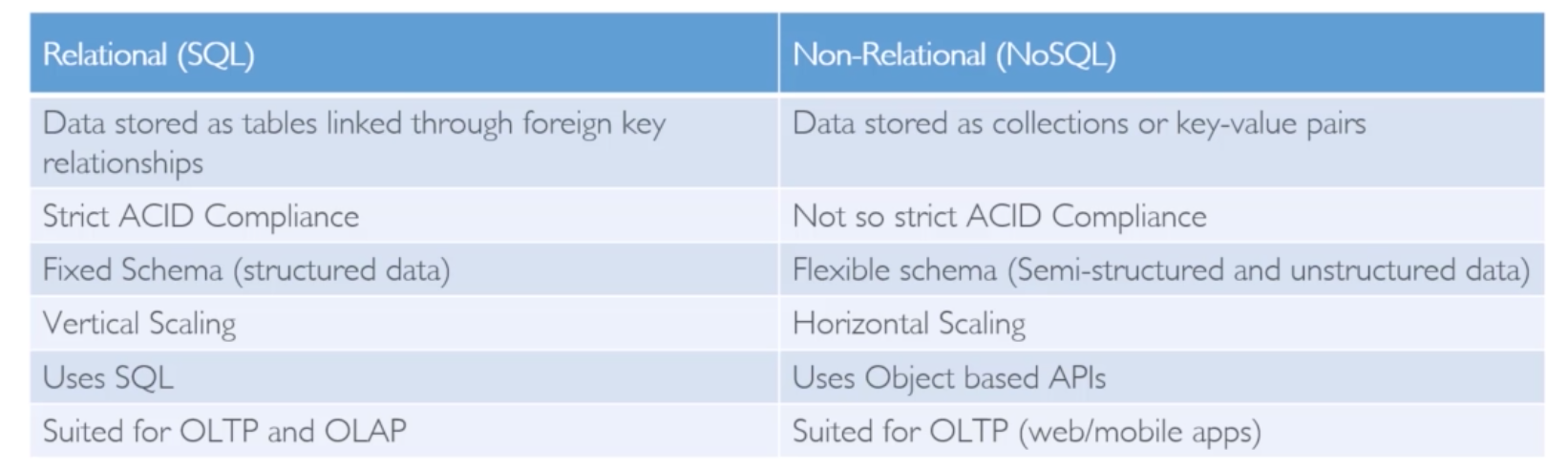
**Non-Relational Databases**

* Sometimes called as NoSQL ( = not only SQL )
* Suitable for semi-structured and unstructured data
* Data stored in denormalized form means that data can be duplicated
* Suited for big data applications
  + Big-data = High Volume, High Velocity, High Variety
  + VVV: Volume, Velocity, Variety
  + Volume: A huge amount of data
  + Velocity: The number of I/O operations per seconds
  + Variety: Lots of different types of data semi-structured or unstructured data
* Also suited for low-latency (high-speed) applications ( why? Simple data formats increase the speed)
* Flexible data model ( ie trades some of the ACID properties for speed and flexibility)
* Not suited for OLAP or analytical workloads
* Types of non-relational databases
  + DynamoDB
  + DocumentDB
  + ElastiCache
  + Neptune

What is BASE compliance

* BASE = Basically Available Soft-State Eventually-consistent
* (BA) Basically available – Basic Read and Write operations are available
* (S) Soft-State – No consistency guarantees and data state may change
* (E) Eventually Consistent – Data is expected to be consistent over time
* Use cases that prefer high performance over strong consistency

Comparison of Relational and Non-Relational Databases



**DynamoDB and DAX**

**High Performance at any scale**

* Non-relational Key-Value store
* Fully Managed, Serverless, NoSQL database in the cloud
* Fast, Flexible, Cost-effective, Fault Tolerant, Secure (Encrypted at rest and in transit)
* Global Tables: Multi-region, multi-master database
* Backup and restore with PITR ( Point-in-time Recovery )
  + Any second within the last 35 days
* Single-digit millisecond performance at any scale
* In-memory caching with DAX (DynamoDB Accelerator, microsecond latency)
* Supports CRUD ( Create / Read / Update / Delete ) operations through APIs
* Supports transactions across multiple tables (ACID support )
* No direct analytical queries ( no joins )
* Access patterns must be known ahead of time for efficient design and performance

Creating a DynamoDB Table Hands On

Important Considerations:

* Primary Key
* Table Name
* Access Patterns

Example

Requirement:

Store player data for different games

We have different players playing different game sessions, and for each player, we went to store the games that have played along with, the time at which the game was played, the outcome of the match ( win or lose ), the score, match duration, and so on

**Access Patterns**

1. Store and retrieve game data for a particular player
   1. We should be able to retrieve the gameplay data of a particular player, as well as the historical data of the player ( all of the games that the player has played in the past )
2. To get the leaderboard for a particular game
   1. The second access pattern would be to create a gaming leaderboard. We should be able to get the top scores of any particular game and display it in real-time

Since DynamoDb is a non-relational database, all anyone item (record) needs is a primary key and the rest of the attributes are optional

A primary key in DynamoDB can be made up of a

* Partition Key
  + How data is partitioned within the table
* Sort Key
  + Designates how the data is sorted for each partition key

The combination of which should fulfill the first access pattern, and create the primary index

A secondary index can also be created

* Partition Key
* Sort Key

There are two types of secondary index, there are global and local secondary indexes

DynamoDB Terminology Compared to SQL

* SQL
  + DynamoDB
* Tables
  + Tables
* Rows
  + Items
* Columns
  + Attributes
* Primary Keys (multi-column and optional )
  + Primary Keys (Mandatory, minimum one, and maximum two attributes)
  + Since Primary Keys are mandatory it helps Dynamo deliver a consistently high performance at any scale
* Indexes
  + Local Secondary Indexes
* Views
  + Global Secondary Indexes

DynamoDB Tables

* Tables are top-level entities
* No strict inter-table relationships ( Independent Entities )
* You control performance at the table level
* Table items stored as JSON ( dynamoDB-specific JSON)
* Primary keys are mandatory, rest of the schema is flexible
* The primary key can be simple or composite
* Simple Key has a single attribute ( = partition key or hash key )
* Composite Key has two attributes ( = partition / hash key + sort / range key )
* Non-key attributes ( including secondary key attributes ) are optional

Data Types In DynamoDB

* Scalar Types
  + One Key => One Value
  + Exactly one value
  + E.g string, number, binary, Boolean, and null
  + Keys or index attributes only support string, number and binary scalar types
* Set Types
  + Multiple scalar values
  + Eg string set, number set, and binary set
* Document Types
  + Complex structure with nested attributes
  + Eg. list or map

AWS Global Infrastructure

* Has multiple AWS Regions across the globe
* Each region has one or more AZs (Availability Zones )
* Each AZ has one or more facilities ( = Data Centers )
* DynamoDB automatically replicates data between multiple facilities within the AWS region
* This is almost Real-time Replication as well
* AZs act as independent failure domains

DynamoDB Consistency

* Read Consistency:
  + Strong Consistency
  + Eventual Consistency
  + Transactional Consistency
* Write Consistency
  + Standard Consistency
  + Transactional Consistency
* Strong Consistency:
  + The most up to date data
  + Must be requested explicitly
* Eventual Consistency
  + May or may not reflect the latest copy of the data
  + Default consistency for all read operations
  + 50% cheaper than strong consistency
* Transactional Reads and Writes
  + For ACID support across one or more tables within a single AWS account and region
  + 2x the cost of strongly consistent reads
  + 2x the cost of standard writes

Strongly Consistent Read vs Eventually Consistent Reads

* Eventually Consistent Reads:
  + If we read just after a write, it’s possible we’ll get an unexpected response (stale copy ) because of the eventual consistency replication of the data. (Because the data is read from any of the servers)
* Strong Consistency Reads:
  + If we read just after a write we will get the correct data ( usually reading from the server that was just written too)
* By Default:
  + DynamoDB uses Eventually consistent Reads, but GetItem, Query & Scan provide a “ConsistentRead” parameter you can set to true.

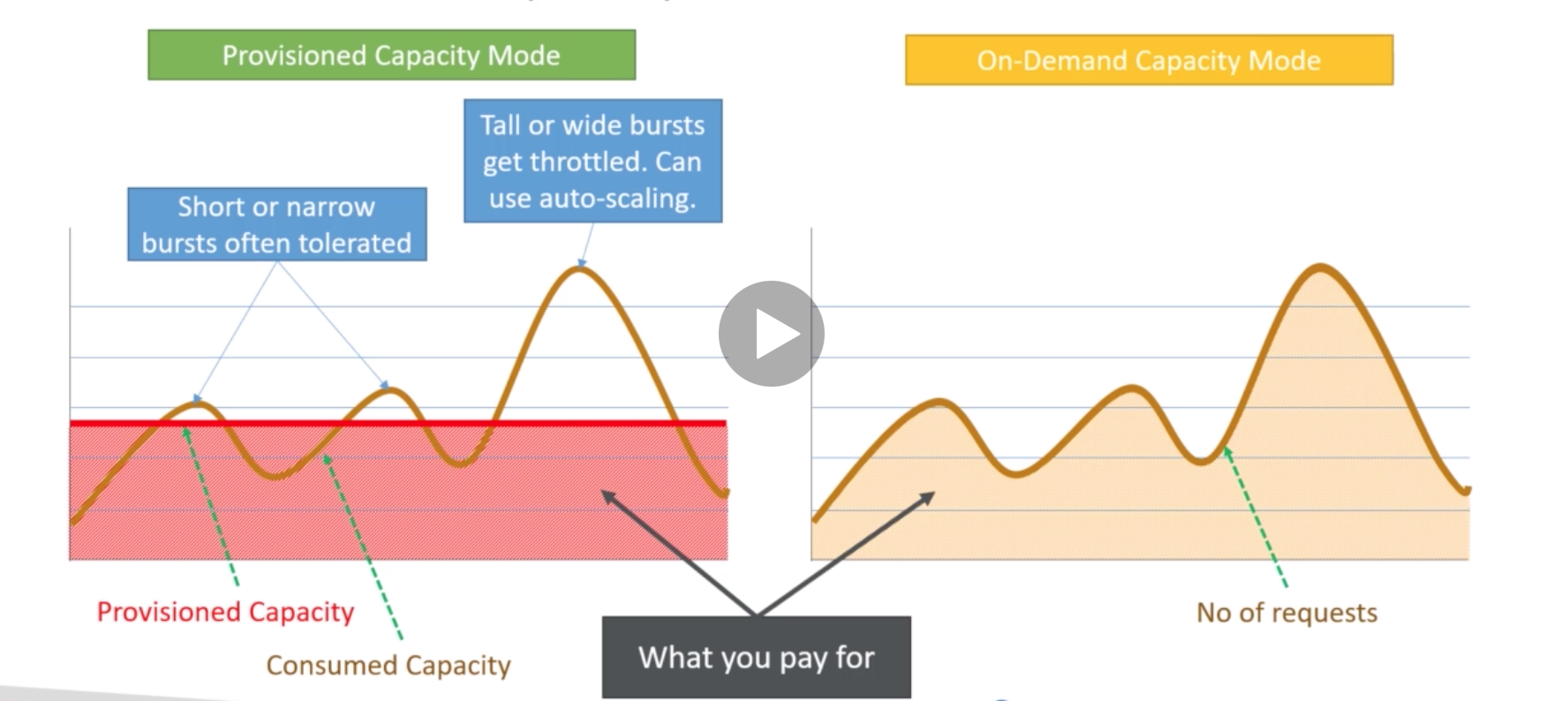
DynamoDB Transactions – Transactional consistency

* It is written to both or none at all

DynamoDB Pricing Model:

* Provisioned Capacity
  + You pay of the capacity you provision (=number of reads and writes per seconds)
  + You can use auto-scaling to adjust the provisioned capacity
  + Uses Capacity Units:
    - Read Capacity Units (RCUs)
    - Write Capacity Units (WCUs)
  + Consumption beyond provisioned capacity may result in throttling
* Reserved capacity
  + USed for discounts over 1 or 3 year term contracts ( you’re charged a one time fee + an hourly fee per 100 RCUs and WCUs )
* On-Demand Capacity
  + You pay per request ( = number of read and write requests your application makes
  + No need to provision capacity units
  + DynamoDB instantly accommodates your workloads as they ramp up or down
  + Uses Request Units:
    - Read Request Units (RRUs)
    - Write Request Units (WRUs)
  + Cannot use reserved capacity with On-Demand mode
* ALL UNITS for throughput calculation purposes are equivalent
* Storage, backup, replication, streams, caching, data transfer out are charged additionally.

Provisioned Capacity vs On-Demand Capacity



DynamoDB Throughput

* Provisioned Capacity Mode
  + 1 capacity unit = 1 request / sec
  + RCUs ( Read Capacity Units )
    - In blocks of 4 KB, last block always round up
    - 1 strongly consistent table read/sec = 1 RCU
    - 2 eventually consistent table reads/sec = 1 RCU
    - 1 transactional read / sec = 2 RCUs
  + WCUs ( Write Capacity Units)
    - In blocks of 1KB, last block always rounded up
    - 1 table write/sec = 1 WCU
    - 1 transactional write / sec = 2 WCUs
* On-Demand Capacity mode
  + Uses Request Units
    - 1 request unit = 1 request / sec
    - Same as Capacity Units for calculation purposes
  + RRU ( Read Request Units )
    - In blocks of 4 KB, last block always round up
    - 1 strongly consistent table read/sec = 1 RRU
    - 2 eventually consistent table reads/sec = 1 RRU
    - 1 transactional read / sec = 2 RRUs
  + WRUs ( Write Request Units)
    - In blocks of 1KB, last block always rounded up
    - 1 table write/sec = 1 WRU
    - 1 transactional write / sec = 2 WRUs

Provisioned Capacity vs On-Demand Mode:

* Provisioned Capacity
  + Typically used in production environment
  + Use this when you have predictable traffic
  + Consider using reserved capacity if you have steady and predictable traffic for cost savings
  + Can result in throttling when consumption shoots up ( use auto-scaling )
  + Tends to be cost-effective as compared to the on-demand capacity mode
* On-Demand Capacity Mode
  + Typically used in dev/test environments or for small applications
  + Use this when you have variable unpredictable traffic
  + Instantly accommodates up to 2x the previous peak traffic on a table
    - HOWEVER: If you exceed that twice the previous maximum limit within 20 minutes, then there could be some throttling

Calculating Capacity Units:

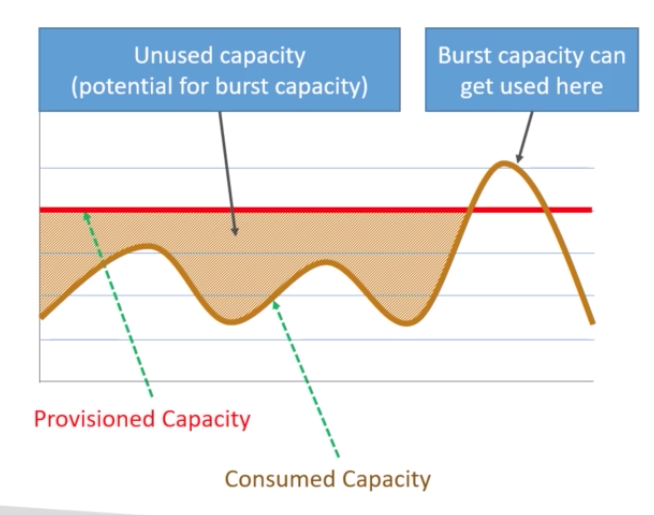
Calculate capacity units to read and write a 15KB item

* RCUs with strong consistency
  + 15KB / 4KB = 3.75 RCUs => Rounded up => 4 RCUs
* RCUs with eventual consistency
  + .5 \* 4 RCUs = 2 RCUs
* RCUs with transactional consistency
  + 2 \* 4 RCUs = 8 RCUs
* WCUs with standard Request
  + 15KB \* 1KB = 15 WCUs
* WCUs with transactional consistency
  + 15 WCUs \* 2 = 30 WCUs

Calculate Throughput

A DynamoDB table has provisioned capacity of 10 RCUs and 10 WCUs. Calculate the total throughput that your application can support:

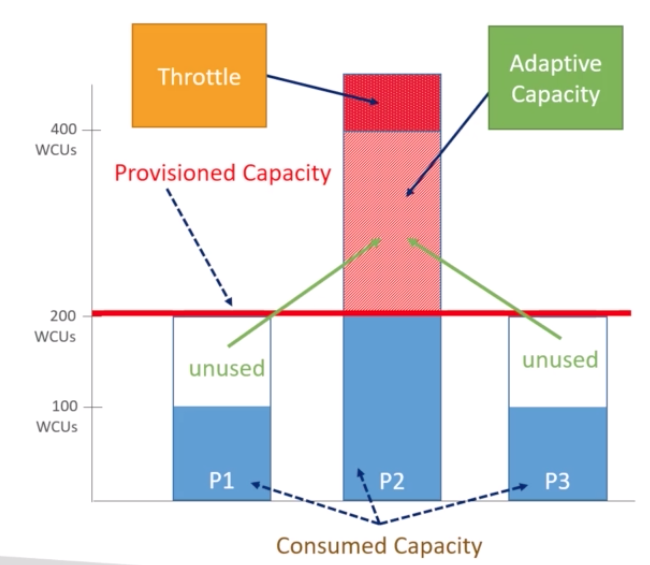
* Read throughput with strong consistency =
  + 4KB \* 10 RCUs = 40 KB/sec
* Read throughput with eventual consistency =
  + 40 KB/sec \* 2 = 80 KB/sec
* Read throughput with transactional consistency =
  + 40 KB/sec / 2 = 20 KB/sec
* Write throughput with standard write =
  + 1 KB \* 10 WCUs = 10 KB/sec
* Write throughput with transactional write =
  + 10 KB/sec / 2 = 5 KB/sec



DynamoDB Burst Capacity:

* To provide for occasional bursts or spikes
* 5 minutes or 300 second of unused read and write capacity is stored for burst capacity
* Can get consumed VERY quickly
* Must not be relied upon

DynamoDB Adaptive Capacity

* Each partition in Dynamo is a 10Gb SSD drive
* Whatever capacity you provision to your table is equally divided between all of your dynamoDB partitions

Example

* Total provisioned capacity = 600 WCUs/sec
* Provisioned capacity per partition = 200 WCUs per sec
* Unused capacity = 200 WCUs per sec
  + Partition 2 would be a hot partition
* So the hot partitions can consume these unused 200 WCUs per sec above its allocated capacity
* Consumption beyond this results in throttling
* For non-uniform workloads
* Works Automatically
* No Guarantees
* 5 - 30 minutes of delay before this kicks in

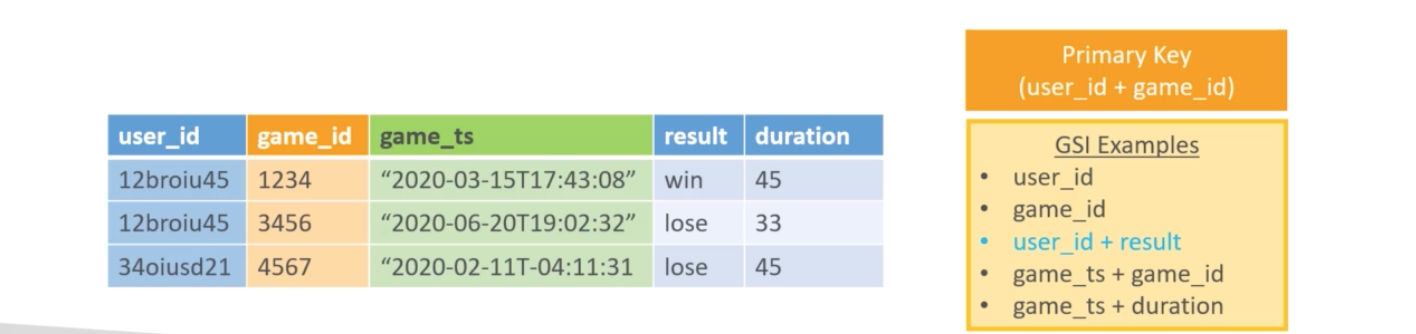
DynamoDB LSI ( Local Secondary Index )

* Can define up to 5 LSIs
* Has the same partition / hash key attribute as the primary index of the table
  + This is basically adding a different sort key to the primary partition / hash key
* Has a different sort / range key than the primary index
* Must have a sort / range key ( = composite key )
* Indexed items must be <= 10 GB in size
* Can only be created at the time of creating the table and cannot be deleted later
* Can only query single partition ( specified by hash key )
* Supports eventual / strong / transactional consistency
* Consumes provisioned throughput of the base table
* Can query any table attributes even if the attributes are not projected on to the index ( can project all or up to 20 individual attributes )



DynamoDB GSI ( Global Secondary Index )

* Can define up to 20 GSIs ( soft limit )
* Can have same or different partition / hash key than the table’s primary index
* Can have same or different sort / range key than the table’s primary index
* Can omit sort / range key ( = simple and composite )
* No size restrictions for indexed items
* Can be created or deleted any time. Can delete only one GSI at a time
* Can query across partitions ( over entire table )
* Supports only eventual consistency
* Has its own provisioned throughput
* Can only query projected attributes ( attributes included in the index )



When to choose which index

* Local Secondary Indexes
  + When the application needs the same partition key as the table
  + When you need to avoid additional costs
  + When the application needs strongly consistent or transactional index reads
* Global Secondary Indexes
  + When the application needs different or same partition key as the table
  + When application needs finer throughput control
  + When application only needs eventually consistent index reads

DynamoDB Indexes and Throttling

* Local Secondary Indexes
  + Uses the WCU and RCU of the main table
  + No special throttling considerations
* Global Secondary Indexes
  + If the writes are throttled on the GSI, then the main table will be throttled! ( even if the WCUs on the main table are fine)
  + Choose your GSI partition key carefully
  + Assign your WCU capacity carefully

Simple design patterns with DynamoDB

* You can model different entity relationships like 1:1, 1:N, N:M ( Since DynamoDB is NoSQL you have to think about the models beyond the schema )
* Store Players’ game states – 1: 1 modeling, 1:N modeling
  + User\_id as PK, game\_id as SK ( 1:N modeling )
  + Primary Index
* Players’ gaming history – 1:N modeling
  + User\_id as PK, games \_ts as SK ( 1:N modeling )
  + Local Secondary Index
* Gaming Leaderboard – N:M modeling
  + GSI with game\_id as PK and score as SK
  + Global Secondary Index

DynamoDB Write Sharding

* Imagine we have a voting application with two candidates A and candidate B.
* Since each partition can only have 10 GB worth of data
* If we use a partition key of candidate\_id we will run into partitions issues, as we only have two partitions
* A solution would be to add a suffix ( usually a random suffix, sometimes calculated suffix)

Error and Exceptions in DynamoDB

* HTTP 5xx ( Server Side Errors )
  + Service Unavailable etc.
* HTTP 4xx ( Client Side Errors )
  + Authentication failure
  + Missing required parameters etc.
* Common Exceptions
  + Access Denied Exception
  + Conditional Check Failed Exception
  + Item Collection Size Limit Exceeded Exception
  + Limit Exceeded Exception
  + Resources In Use Exception
  + Validation Exception
  + Provisioned Throughput Exceeded Exception
    - Solution for throughput exceeded exception
    - Built into AWS SDKs
    - Error Retries
    - Exponential Backoff

DynamoDB Partitions

* Store DynamoDB table data ( physically )
* Each ( physical ) is a partition = 10 GB SSD volume
* Not to be confused with table’s partition / hash key ( which is a logical partition )
* One partition can store items with multiple partition keys
* A table can have multiple partitions
* Number of table partitions depend on its size and provisioned capacity
* Managed internally by DynamoDB
* Provisioned capacity is evenly distributed across table partitions
* Partitions once allocated, cannot be deallocated
  + Once you add a partition you cannot remove a partition

Calculating DynamoDB Partitions

* 1 partition = 1000 WCUS or 3000 RCUs ( Maximum supported throughput per partition )
* 1 partition = 10 GB of data
* No. of Partitions = The number of partitions is based on throughput or size
* Partitions based on Throughput
  + Pt = Round-Up [ ( RCUs / 3000 ) + (WCUs / 1000) ]
* Partitions based on Size
  + Ps = Round-Up [ (Storage Required in GB / 10 GB ) ]
* P = whichever is higher Max(Pt, Ps)

Partition Behavior Example ( Scaling up Capacity )

* Provisioned Capacity: 500 RCUs and 500 WCUs
* Storage requirements < 10 GB
* Number of Partitions:
  + Pt = ( 500 / 3000 + 500 / 1000 )
  + Pt = ( ⅙ + ½)
  + Pt = ( ⅔ )
  + Pt = ( .66 )
  + Pt = 1
* Say, we scale up the provisioned capacity
  + 1000 RCU and WCU
  + Pt = ( 1000 / 3000 + 1000 / 1000 )
  + Pt = ( ⅓ + 1 )
  + Pt = 1.34
  + Pt = 2
  + Each partition would get 500 RCUs and WCUs

Partition Behavior Example ( Scaling up Storage )

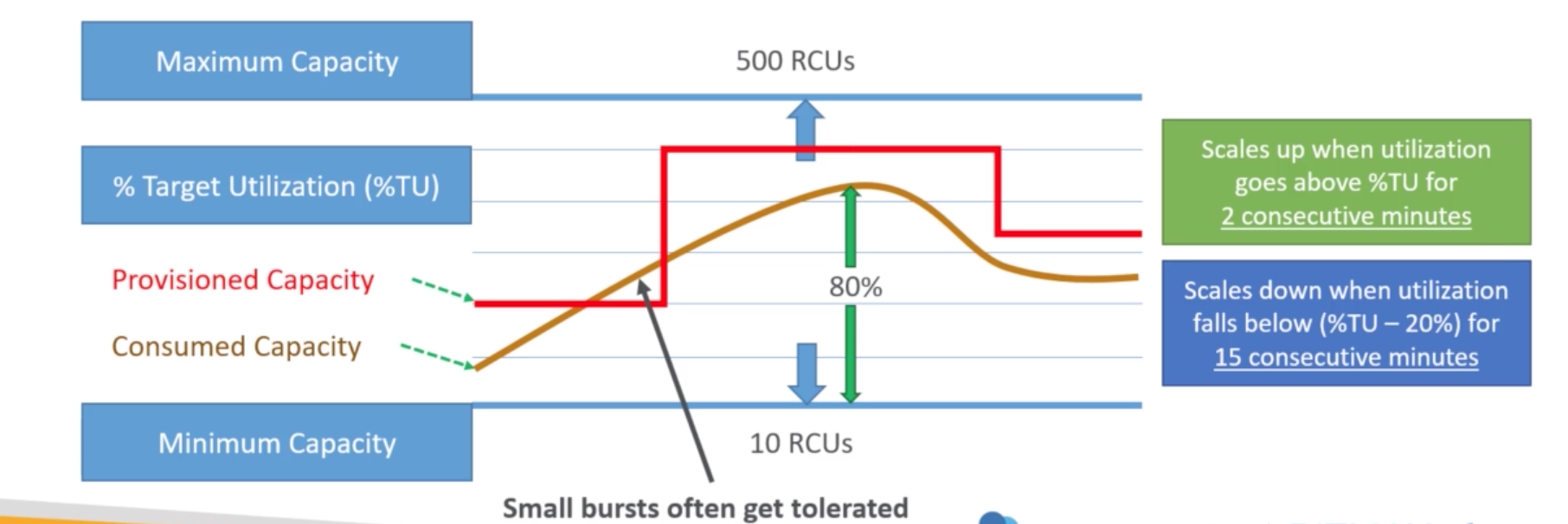
* Provisioned Capacity = 1000 RCUs and 500 WCUs
* Storage Size = 5 Gb
  + Pt = Round Up [ (1000 / 3000 ) + ( 500 / 1000 ) ]
  + = Round [ 0.67 ]
  + Pt = 1
  + Ps = Roundup [ ( 5 GB / 10 Gb) ]
  + Ps = Roundup [ .5 ]
  + Ps = 1
  + Therefore 1 partition will be needed
* Scale up the storage
  + Ps = Roundup ( 50 / 10 )
  + Ps = 5
  + There for 5 partitions will be needed
* Therefore each partition will now only receive ⅕ th of the provoisioned capacity
* If you were to scale to 500 GB the throughput for the 50 Partitions would only be 1/50th of the provisioned capacity

DynamoDB Scaling

* You can manually scale up provisioned capacity as and when needed
* You can only scale down up to 4 times in a day
* Additional one scale down if no scale downs in last 4 hours
* Effectively 9 scale downs per day
* Scaling affects partition behavior
* Any increase in partitions on scale up will not result in decrease on scale down
* Partition once allocated will not get deallocated later

Auto Scaling in DynamoDB

* No additional costs, uses AWS Application Auto Scaling service
* You set the desired target utilization, minimum and maximum provisioned capacity
* Scales up when utilization goes above % Threshold Utilization for 2 Consecutive minutes
* Scales down when the utilization falls below the percentage threshold utilization for 15 consecutive minutes
* Small bursts are often tolerated by the DynamoDB burst and adaptive capacities



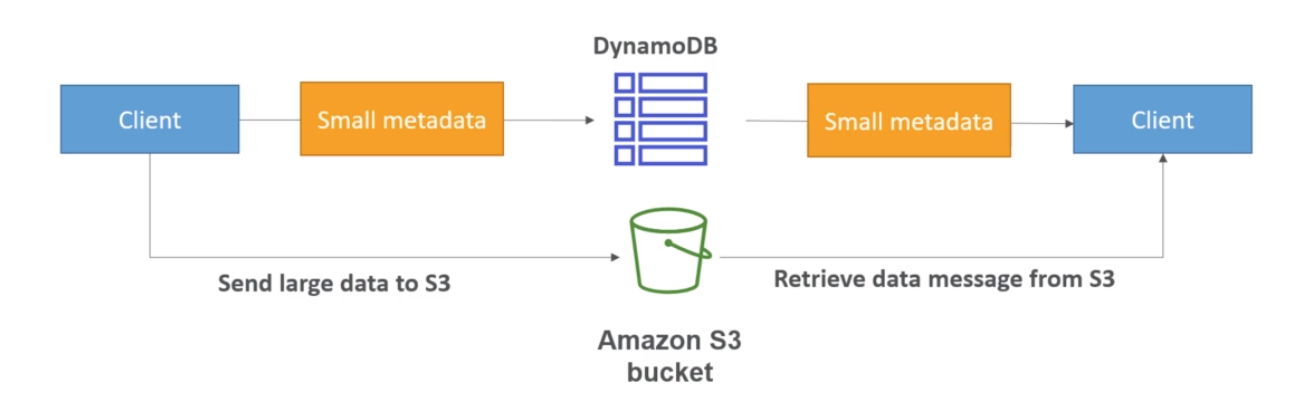
Don’t temporarily increase WCUs or RCUs because you will increase your partitions which post temp increase it will not work effectively

DYnamoDB Best Practices

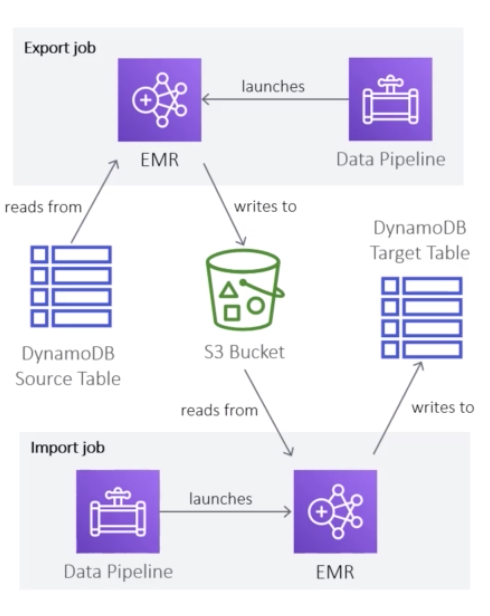
* Efficient Key Design
  + Partition key should have many unique values
  + Distributed reads / writes uniformly across partitions to avoid hot partitions.
  + Store hot and cold data in separate tables
  + Consider all possible query patterns to eliminate use of scans and filters
  + Choose sort key depending on your application’s need
  + Use indexes based on when your application’s query patterns
  + Use primary key or LSIs when strong consistency is desired
  + Use GSIs for finer control over throughput or when your application needs to query using a different partition key
  + Use shorter ( yet intuitive ) attributes names => Save memory and money
* Storing Large Item Attributes
  + USe compression ( GZIP )
  + Use S3 to store large attributes
  + Split large attributes across multiple items
* Reading
  + Avoid scans and filters
  + Use eventual consistency for reads
* LSI
  + Use LSIs Sparingly
  + Project fewer attributes to save space, and decrease duplicated data
  + Watch for expanding item collections ( 10 GB size limit )
* GSI
  + Project fewer attributes to save on storage costs
  + If you do not store the right attributes it will cost more because that data has to be fetched
  + Can only be used for eventually consistent read replicas

DynamoDB – Storing larger items

* DynamoDB supports item sizes up to 400 KB each
* This includes attribute name and attribute value ( = entire JSON object )
* Options for storing large items
  + Compress large attribute values, OR
  + Store large attribute values in S3 and store the corresponding S3 Path in DynamoDB



DynamoDB Operations

* Table Cleanup:
  + Option 1: Scan + Delete => very slow, expensive, consumes RCUs and WCUs
  + Option 2: Drop Table + Recreate Table => faster, cheaper, and more efficient
* Copying a DynamoDB Table:
  + Option 1: Use an AWS DataPipeline ( uses EMR )
  + Option 2: create a backup and restore the backup into a new table name ( can take some time )
  + Option 3: Scan + Write => utilizing your own code

DynamoDB Accelerator ( DAX )

* In-Memory Caching, microsecond latency
* Sits between DynamoDB and Client Application ( acts a proxy )
* Saves costs due to reduced read load on DynamoDB
* Helps prevent hot partitions
* Minimal code changes required to add DAX to your existing DynamoDB app
  + The only thing that you need to is change the ARN that read and write requests use to access the dynamodb to the DAX.
* Supports only eventual consistency ( strong consistency requests pass-through to DynamoDB)
* Not for write heavy application
* Runs inside of the VPC
* Supports multi-az ( 3 nodes minimum recommended for production )
* Secure ( Encryption at res with KMS, VPC, IAM, CloudTrail … )

DAX architecture

* DAX has two types of caches (internally)
  + Item Cache
  + Query Cache
* Item cache stores results of index reads ( =GetItem and BatchGetItem)
  + Default TTL of the cache is 5 min ( specified while creating the DAX cluster )
  + When the cache becomes full, older and less popular items get removed
* Query cache stores results of Query and Scan operations
  + Default TTL of 5 min
  + Updates to the Item cache or to the underlying DynamoDB table does not invalidate the query cache. So, TTL value of the query cache should be chosen accordingly ( it should be low )
  + IF THE VALUE OF THE ITEM CHANGES IN THE ITEM CACHE OR THE TABLE THE NEW DATA WILL NOT BE REFLECTED IN THE QUERY CACHE

DAX Operations

* Only supports item level operations
* Table level operations must be sent directly to DynamoDB
* Write operations use a write-through approach
* Data is first written to DynamoDB and then to DAX, and write operations is considered successful only if both the writes to the table and cache are successful ( much like a transactional write )
* You can use a write-around approach to bypass DAX, this is good for writing large amounts of data, you can write directly to DynamoDB ( however this will make the item cache out of sync with the table )
* For reads, if DAX has the data ( = Cache hits ), the data is simply returned without going through to the table
* IF DAX doesn’t have the data ( = Cache miss ), it’s returned from DynamoDB and updated in DAX on the master node
* THE DAX CLUSTER WILL NOT BE UPDATED WITH STRONGLY CONSISTENT READS

DynamoDB – DAX vs ElastiCache

* In most cases DAX is the best choice with DynamoDB tables
* The exception is that if you need to aggregate data, this is because DAX only works with individual objects or query and scan results.
* This would be the case where ElastiCache would be the better choice.
* Implementing DAX  
  TO implement DAX, we create a DAX cluster
* DAX Cluster consists of one or more nodes ( up to 10 nodes per cluster )
* Each node is an instance of DAX
* One node is the master node or primary node
* Remaining nodes act as read replicas
* DAXinternally handles the load balancing between these nodes
* For production 3 nodes are the recommended minimum

Backup and restore in DynamoDB

* Automatically encrypted, cataloged, and easily discoverable
* Highly Scalable - create or retain as many backups for tables of any size
* Backup operations complete in seconds
* Backups are consistent within seconds across thousands of partitions
* No provisioned capacity consumption
* Backups are preserved regardless of table deletion
* Can backup within the same AWS region as the table
* Restores can be within same region or cross region
* Integrated with AWS Backup Service ( can create periodic backup plans )
* Periodic backups can be scheduled using Lambda and CloudWatch triggers
  + However with the AWS Backup service its a easier
* Cannot overwrite an existing table during restore, restores can be done only to a new table ( = new name )
* To retain the original table name, delete the existing table before running restore
* You can use IAM policies for access control
* Two types of backups
  + Restore table gets the same provisioned RCU/WCU as the source table, as recorded at the time of backup
  + Point in time restore and the number of recovery point objectives ( RPO )
  + Basically what it means is that with a restore a point in time backup you will only use 5 minutes of data loss.
  + PITR RTO (Recovery Time Objective ) can be longer as as restore operations creates a new table

What gets restored with a table restore

* What gets restored:
  + Table data
  + GSI and LSI ( optional you can choose to restore without )
  + Encryption settings ( again this can be changed )
  + Provisioned RCUs / WCUs ( with values at the time when backup was created
  + Billing mode ( with value at the time when backup was created )
* Things that are NOT restored
* What you must manually set up on the restored table
  + Auto Scaling Policies, IAM policies
  + CloudWatch metrics and alarms
  + Stream and Time To Live settings
  + Tags

Continuous Backups with PITR

* Restore table data to any second in the last 35 days
* Priced per GB based on the table size
* If you disable PITR and re-enable it, the 35 day clock gets reset
* Words with unencrypted, encrypted tables as well as global tables
* Can be enabled on each local replica of a global table
* If you restore a table which is part of global tables the restored table will be an independent table ( won’t be a global table anymore )
* Always restores data to a new table
* What cannot be restored
  + Stream Settings
  + TTL options
  + Autoscaling configurations
  + Point in time recovery settings
  + Alarms and tags

DynamoDB Encryption

* Server-side Encryption at Rest
  + Enabled by default
  + Uses KMS
  + 256 bit AES Encryption
  + Can use AWS owned CMK, AWS managed CMK, or customer managed CMK
  + Encrypts primary key, secondary indexes, streams, global tables, backups, and DAX clusters
* Encryption in transit
  + Use VPC endpoints for applications running n a VPC
  + Use TLS endpoints for encrypting data in transit

DynamoDB Encryption Client

* For client-side encryption
* Added protection with encryption in-transit
* Resutlts in end to end encryption
* Doesn’t the entire table
* Encrypts the attribute values, but not the attribute names
* Doesn’t encrypt values of the primary keys attributes
* You can selectively encrypt other attribute values
* You can encrypt selected items in a table, or selected attribute values in some or all items

DynamoDB streams

* 24 hours time-order log of all table-write activity
* React to changes ot the DynamoDB table in real time
* Can be read by AWS Lambda, EC2 , ES, Kinesis
* Use Cases
  + Replication
  + Archival
  + Notifications
  + Log processing
* DynamoDB streams are organized into shards
* Records are not retroactively populated to stream after enabling streams
* Simply enable streams from the DynamoDB console
* Four supported views
  + Keys Only
    - Captures only the key attributes of the changed item
  + New Image
    - Captures the entire item after changes
  + Old Image
    - Captures the entire item before changes
  + New and Old images
    - Captures the entire item before and after changes

Time to Live (TTL)

* Allows you to tell DynamoDB when to delete an item from the table
* Simply designate an item attribute as a TTL
* TTL attribute should contain the expiry timestamp for the item ( EPOCH or Unix timestamp )
* Items get marked for deletion on expiry
* Expired items get removed from the table and indexes automatically within about 48 hours
* Expired items can show up in the API responses until they get deleted
* Applications should use filter operations to exclude items marked for deletion
* Deleted items appear in DynamoDB streams ( if streams are enabled )
* Use Cases
  + Data archival to another table ( using DynamoDB streams )
  + Separating hot and cold data in time-series data ( using DYnamoDB streams )

DynamoDB Global Tables

* Automatic, Multi-Master, Active-Active, Cross-region replication
* Useful for low latency, and data recovery purposes
* Near real-time replication ( < 1 second replication lag )
* Eventual consistency for cross region reads
* Strong consistency for same region reads
* “ Laster Writer Wins “ approach for conflict resolution
* Transactions are ACID-compliant only in the region where write actually occurs
* To enable global tables for a table, the table must be empty across regions
* Only one replica per region
* Must enable DynamoDB Streams with New and Old Images
* Must have the same table name and primary keys across regions
* Recommended to use identical settings for table and indexes across regions

Fine-grained access control in DynamoDB

* Can use IAM to control access to DynamoDB resources
* DynamoDB does not support tag-based conditions
* Can use condition keys in IAM policy for fine-grained access control
  + Can restrict access to certain items / attributes based on user identity (in a table or in a secondary index )
  + Example – allow users to access only the items that belong to them, based on certain primary key values
* ForAllValues:StringEquals:
  + Compares the requested attributes values with those in the table
* dynamodb:LeadingKeys
  + Represents partition key
    - EG access will be allowed only if user’s user\_id matches the primary key value on the table
* dynamodb:Attributes
  + To limit the access to specific attributes
* LINK
  + <https://docs.aws.amazon.com/amazondynamodb/lates/developerguide/specifying-conditions.html>

DynamoDB Web Identity Federation

* Also called as DynamoDB federated identities
* For authentication and authorization of app users
* No need to create individual IAM users
* Login with an identity provider and get a web identity token
* Use Cognito to exchange the web identity token with temporary IAM credentials (STS token )
  + Or callSTS API directly
* Use the temporary credentials to access DynamoDB ( as per the role associated with the credentials )
* Can use fine-grained access control ( with condition keys )

CloudWatch Contributor Insights

* Contributor insights shows you the most accessed and throttled items in DynamoDB
* Also helps you analyze time-series data
* Supported for DynamoDB and CloudWatch Logs
* Identify outliers / contributors impacting system and application performance
* Find the heaviest traffic patterns
* Analyze the top system processes
* Displayed on CloudWatch dashboard
* Integrated with CloudWatch alarms

Amazon RDS and Aurora ( Relational Databases are not going away )

* RDS stands for **Relational DAtabase Service**
* Uses **SQL** ( Structured Query Language )
* **Supported** Engines
  + PostgreSQL
  + MySQL
  + Oracle
  + Microsoft SQL Server
  + Aurora
* **Managed** DB service
* Launched within a VPC, usually in private subnet, control network access using security groups ( important especially when using lambda )
* **Storage** is through Elastic Block Storage EBS ( gp2 or io1), can increase volume size with auto scaling
* **Backups**: automated with point-in-time recovery. Backups expire (take snapshots to retain backups after expiration)
* **Snapshots**: manual, can make copies of snapshots cross region
* **Monitoring** through CloudWatch
* RDS Events: get notified via SNS for events ( operations, outages, etc…)
* Supports multi-az deployments

Why use RDS?

* When you self-host your DB, you manage everything
  + You manage hardware ( physical infrastructure )
  + You manage software ( underlying operating system )
  + You manage the application ( the literal database )
* When you host on AWS Cloud ( on EC2, but not on RDS )
  + AWS manages hardware
  + You manage the software
  + You manage the application
* When you use RDS, AWS manages everything
  + AWS manages hardware
  + AWS manages software
  + AWS manages application
* See the AWS shared scope

Using RDS vs deploying DB on EC2

* RDS is a managed service
  + Automated provisioning, OS patching
  + Continuous backups and restore to specific timestamp (point in time restore)
  + Monitoring dashboards
  + Read replicas for improved read performance
  + Multi AZ setup for DR
  + Maintenance windows for upgrade
  + Scaling capability
  + Storage backed by EBS (gp2 or io 1)
* But you can’t SSH into your underlying DB instance

RDS pricing model

* When creating an RDS database, you choose:
  + Instance type ( on-demand and reserved)
  + Engine type (PostgreSQL, MySQL, MariaDB, Oracle, Microsoft SQL Server, and Aurora)
  + DB instance class ( based on memory, CPU and I/O capacity requirements)
* Uses pay as you go pricing model
  + On-demand ( pay for compute capacity per hour)
  + Reserved (deeply discounted, 1 year - 3 year term contract)
* Storage
  + (GB/month) / Backups / Snapshot export to S3
* I/O ( per million requests )
* Data transfer

Choosing an Instance Class

* Instance class types
  + Standard
  + Memory-optimized ( memory intensive high performance workloads
  + Burstable
* Burstable performance instances
  + Can burst to higher level of CPU performance on demand depending on the workload while providing baseline level performance at other times
  + Managed through CPU credits ( 1 = credit 100% core utilization for one minute)
  + You get CPU credits when you underutilize the CPU

Choosing Storage Type

* General Purpose storage ( = cost-effective SSD storage )
  + You choose the storage size
  + You get a baseline of 3 IOPS / GB
  + Volumes below 1 TiB can burst up to 3,000 IOPS ( uses I/O credits )
  + Used with variable workloads
  + Typically used for small to medium sized DBs and in DEV/TEST environments
* Provisioned IOPS ( = high-performance storage, recommended for production )
  + You choose storage size and required IOPS
  + Fast predictable performance
  + Up to 32,000 IOPS max per DB instance
  + Use when consistent high IOPS are required ( I/O-intensive workloads)
  + Well-suited for write heavy workloads
* If an instance runs out of storage, it may no longer be available until you allocate more storage ( => use storage autoscaling to make sure you do not run into this particular issue )

Storage Auto Scaling in RDS

* Both storage types ( gp2 & io1 ) support storage auto scaling
* Dynamically scales up storage capacity based on workload
* Storage is scaled up automatically when the utilization nears the provisioned capacity
* Zero down time
* You save costs due to over-provisioning and prevent downtime due to under-provisioning
* Use this when the workload is unpredictable
* You must set the max storage limit for auto scaling storage

Parameter Groups

* Define configuration values specific to the selected DB engine
* You can think of RDS parameter group as RDS equivalent of configuring the my.cnf file on an on-premise MySQL DB
* Default parameter group cannot be edited
* To make config changes, you must create a new parameter group
* The New parameter group inherits settings from the default parameter group
* Can be applied to any DB instance within the AWS region
* Examples ( vary by DB engine )
  + Autocommit
  + Time\_zone
  + force \_ssl
  + Default\_storage\_engine
  + Max\_connections

Changes to parameters in parameter groups

* Changes to dynamic parameters always get applied immediately ( irrespective of the Apply Immediately setting )
* Changes to static parameters require a manual reboot
* DB parameter group shows a status of pending-reboot after applying change
* Status of pending-reboot will not trigger automatic reboot during the next maintenance window
* You must manually reboot the DB instance to apply parameter changes
* Status will change from pending reboot to in-sync post manual reboot



RDS Option Groups

* For configuration of optional features offered by DB engines( these are the options that are not covered by the parameter group )
* Default option group is empty and cannot be modified
* To make config changes, you must create a new option group
* New option groups derives its settings from the default option group
* New option groups are empty when created, you can add options to it
* Examples;
  + SQLSERVER\_BACKUP\_RESTORE in SQL Server
  + NATIVE\_NETWORK\_ENCRYPTION in Oracle
  + MARIADB\_AUDIT\_PLUGIN in MariaDB and MySQL
  + MEMCACHED to enable Memcache support in MySQL

RDS Security – Network

* Launch within VPC to restrict internet access to the RDS instance
* You can’t change the VPC after creating the DB
* RDS databases are usually deployed within a private subnet, not in a public one
* RDS security works by leveraging security groups ( the same concept as for EC2 instances ) – it controls which IP / security group can communicate with RDS )
* Use security groups to control access at DB, EC2 and VPC level

RDS Security – Shared Responsibility

* Your Responsibility
  + Check the ports / IP / security group inbound rules in DB’s Security Group
  + In-database user creation and permissions or manage through IAM
  + Creating a database with or without public access
    - For production you would never have a publicly accessible database so it should be sitting a private subnet
  + Ensure parameter groups or DB is configured to only allow SSL connections
* AWS Responsibility
  + No SSH access
  + No manual DB patching
  + No manual OS patching
  + No way to audit the underlying instance

RDS Security – IAM

* Use IAM to secure access to the RDS DB resources
* IAM policies help control who can manage AWS RDS ( through the RDS API )
* Traditional USername and Password can be used to log in to the database
* IAM-based authentication can be used to login into RDS MySQL & PostgreSQL

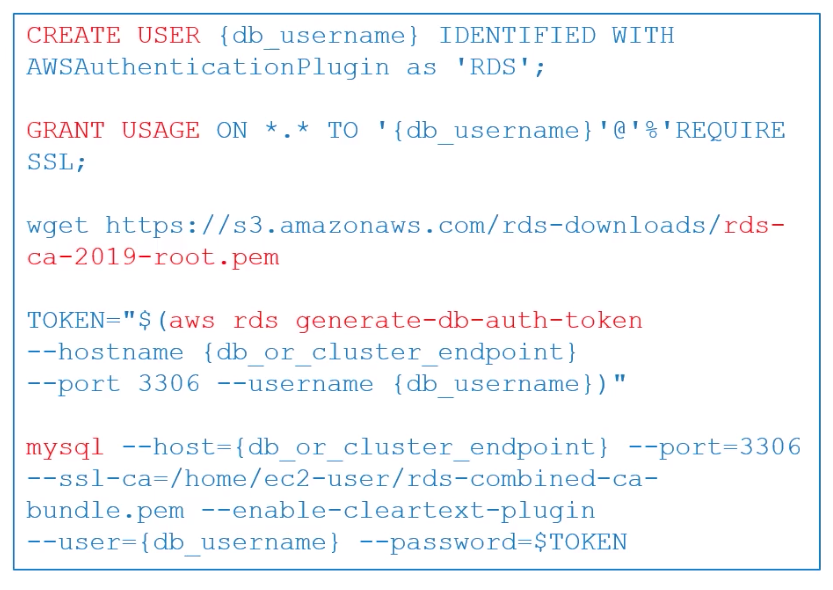
RDS Security – IAM Policy best practices

* IAM policies are used to control who can create, access, or delete DB resources in your account
* Grant least privilege to groups / users / roles ( i.e. grant only the permissions required for the task
* Use MFA for sensitive operations
* Use policy conditions – to restrict access to selected IP addresses, or within a specified date, or to require use of SSL / MFA

RDS - IAM Authentication ( IAM DB AUTH )

* IAM database authentication works with MySQL and PostgreSQL
* You don’t need a password, just an authentication token obtained through IAM & RDS API calls
* Auth token has a lifetime of 15 minutes
* Benefits
  + Network in/out must be encrypted using SSL
  + IAM to centrally manage users instead of DB
  + Can leverage IAM Roles and EC2 Instance profiles for easy integration

Using IAM DB Auth

* MySQL
* Enable IAM DB authentication on the DB cluster
* Create a DB user ( without a password )
* Attach an IAM policy to map the DB user to the IAM role
* Attach the IAM role to an IAM user ( or to an EC2 instance )
* Now you can connect to the DB using IAM token over SSL
* Note: use native GRANT / REVOKE for DB specific privileges

Amazon Aurora

* MySQL and PostgreSQL-compatible relational database in the cloud ( that means your drivers will work as if Aurora was a POstgres or mYSQL DB
* 5x faster than standard MySQL databases
* 3x faster than standard PostgreSQL databases
* 1/10th the cost of commercial-grade RDBMS
* Up to 15 read replicas ( mult-ax, auto scaling read replicas )
* Aurora Serverless ( automatic star/stop, autoscaling, self healing storage )
* Aurora Global DB - support multi-region read replication ( = local reads w/ under one second latency)
* Auto Scaling of storage from 10 GB to 64TB (soft limit)
* Same security / monitoring / maintenance features as RDS
* Only available on RDS ( can’t be hosted on EC2 or elsewhere)
* Maintain 6 copies across 3 AZs ‘
* Backups are stored on S3
* Fast backtracking option for PITR ( point in time restores )
* Automatic and fast failovers
* If desired table indexes can exist on only the replicas