

COS80011

Web Application Architectures

Lecture 4 Database

includes material from
ACF Module 2.4 – Databases
ACA Module 7 – Designing Web-scale media
ACD Module 5 – Developing Flexible
NoSQL Solutions with DynamoDB



Last Week



■ Access Control

- ☐ IAM: Users, Groups, Roles (ACD Module 3)
- ☐ Authentication with
 - ☐ IAM (ACD Module 3)
 - ☐ AWS Security Token Service (ACD Module 12)
 - ☐ AWS Cognito (ACD Module 12)
- ☐ Authorisation with Policies (ACD Module 3)
- ☐ Securing your application (ACD Module 12)
 - ☐ Secure Connections
 - ☐ Managing Secrets

Quizzes:
ACD Mod 3 IAM
ACD Mod 4 S3
ACD Mod 12 Developing Secure Apps

■ Storage services (ACD Module 4)

- ☐ File Storage – AWS EBS, EFS
- ☐ Object storage – AWS S3
- ☐ Managing access to S3

This week



■ Relational vs non-relational 'NoSQL' databases

- ☐ High-level Comparison
- ☐ NoSQL data models

■ RDS (*in brief*)

■ Dynamo DB

- ☐ Key Concepts
- ☐ Partitions and data distribution
- ☐ Secondary indexes
- ☐ Read/write throughput
- ☐ Streams and global tables
- ☐ Basic operation on tables

Quizzes:

ACF 2.0.4 Database

ACA Mod 7 Web Scale media

ACD Mod 5 NoSQL

SQL and NoSQL Databases

	SQL	NoSQL												
Data Storage	Rows and Columns	Key-Value, Document, Graph based, ...												
Schemas	Predefined, Fixed	Dynamic												
Querying	Using SQL, complex joins possible	Diverse (can have SQL-like QLs)												
Scalability	Vertical	Horizontal												
	<table><tr><th>ISBN</th><th>Title</th><th>Author</th><th>Format</th></tr><tr><td>9182932465265</td><td>Cloud Computing Concepts</td><td>Wilson, Joe</td><td>Paperback</td></tr><tr><td>3142536475869</td><td>The Database Guru</td><td>Gomez, Maria</td><td>eBook</td></tr></table>	ISBN	Title	Author	Format	9182932465265	Cloud Computing Concepts	Wilson, Joe	Paperback	3142536475869	The Database Guru	Gomez, Maria	eBook	<pre>{ ISBN: 9182932465265, Title: "Cloud Computing Concepts", Author: "Wilson, Joe", Format: "Paperback" }</pre>
ISBN	Title	Author	Format											
9182932465265	Cloud Computing Concepts	Wilson, Joe	Paperback											
3142536475869	The Database Guru	Gomez, Maria	eBook											

A SQL database stores data in rows and columns. Rows contain all the information about one entry, and columns are the attributes that separate the data points. A SQL database schema is fixed: columns must be locked before data entry. Schemas can be amended if the database is altered entirely and taken offline. Data in SQL databases is queried using SQL (Structure Query Language), which can allow for complex queries. SQL databases scale vertically by increasing hardware power.

NoSQL databases store data using one of many storage models including key-value pairs, documents, and graphs. NoSQL schemas are dynamic, and information can be added rapidly. Each 'row' doesn't have to contain data for each 'column'. Data in NoSQL databases is queried by focusing on collections of documents. NoSQL databases scale horizontally, by increasing servers.

ACID properties vs Eventual consistency



- Traditional databases typically offer ACID (Atomicity, Consistency, Isolation, Durability) guarantees
- Hard to guarantee without performance cost in distributed databases
- Eventually consistent - Optimistic Concurrency Control (OCC) is used.
- Achieves high availability that informally guarantees eventually all accesses to an item will return the last updated value.
- OCC assumes that multiple transactions can frequently be completed without interfering with each other.
- While they are running, transactions use data resources without acquiring locks on those resources. If check reveals conflicting modifications, the committing transaction rolls back and can be restarted.

ACID Properties - (source: <https://en.wikipedia.org/wiki/ACID>)

Atomicity

Transactions are often composed of multiple statements. Atomicity guarantees that each transaction is treated as a single "unit", which either succeeds completely, or fails completely: if any of the statements constituting a transaction fails to complete, the entire transaction fails and the database is left unchanged. An atomic system must guarantee atomicity in each and every situation, including power failures, errors and crashes.

Consistency

Consistency ensures that a transaction can only bring the database from one valid state to another, maintaining database invariants: any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This prevents database corruption by an illegal transaction, but does not guarantee that a transaction is correct. Referential integrity guarantees the primary key - foreign key relationship.

Isolation

Transactions are often executed concurrently (e.g., multiple transactions reading and writing to a table at the same time). Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially. Isolation is the main goal of concurrency control; depending on the method used, the effects of an incomplete transaction might not even be visible to other transactions.

Durability

Durability guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that

completed transactions (or their effects) are recorded in non-volatile memory.

NoSQL Databases

- Can be an alternative to relational databases for some types of applications
- Can process large amounts of data with high availability (depending on the NoSQL solution, configuration, and architecture)
- Form a broad category with different implementations and data models
- Have the common feature of distributed fault tolerance

NoSQL Data models and examples



- Wide Column:
 - AWS DynamoDB, Cassandra, Hbase, ...
- Document:
 - Apache CouchDB,, IBM Domino, MongoDB, ...
- Key-value:
 - Apache Ignite, MemcacheDB, Oracle NoSQL Database, Redis, ...
- Graph:
 - Apache Giraph, Neo4J, ...

NoSQL Databases

Unmanaged NoSQL (e.g. run on Amazon EC2):

- Cassandra, HBase, Redis, MongoDB, Couchbase, and Riak

Managed NoSQL:

- Amazon DynamoDB
- ElastiCache with Redis
- Amazon Elastic Map Reduce supports HBase

NoSQL Databases

Does your app need transaction support, ACID compliance, joins, SQL?

- Can it do without these for all, some, or part of its data model?
Refactor database hotspots to NoSQL solutions
- NoSQL databases can offer increases in flexibility, availability, scalability, and performance

ACID refers to atomicity, consistency, isolation, and durability, which are the four primary attributes that guarantee that database transactions are processed reliably. For more on the ACID model, see:

<http://databases.about.com/od/specificproducts/a/acid.htm>

This week



■ Relational vs non-relational 'NoSQL' databases

- ☐ High-level Comparison
- ☐ NoSQL data models

■ **RDS (in brief)**

■ Dynamo DB

- ☐ Key Concepts
- ☐ Partitions and data distribution
- ☐ Secondary indexes
- ☐ Read/write throughput
- ☐ Streams and global tables
- ☐ Basic operation on tables

Amazon RDS Relational Database Service

© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Welcome to an introduction to the database services available on Amazon Web Services.

Unmanaged vs. Managed Services



Unmanaged:

Scaling, fault tolerance, and availability are managed by you.



Managed:

Scaling, fault tolerance, and availability are typically built in to the service.



© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

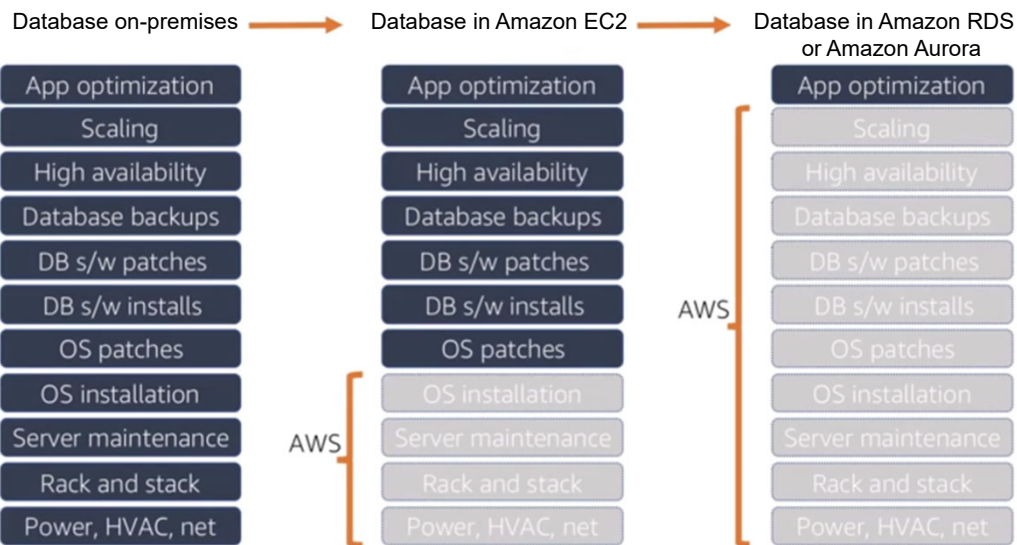
AWS solutions typically fall into one of two categories: unmanaged or managed.

Unmanaged services are typically provisioned in discrete portions as specified by you. Unmanaged services require the user to manage how the service responds to changes in load, errors, and situations where resources become unavailable. For instance, if you launch a web server on an Amazon EC2 instance, that web server will not scale to handle increased traffic load or replace unhealthy instances with healthy ones unless you specify it to use a scaling solution such as AWS Auto Scaling, because Amazon EC2 is an "unmanaged" solution. The benefit to using an unmanaged service is that you have more fine-tuned control over how your solution handles changes in load, errors, and situations where resources become unavailable.

Managed services require the user to configure them (for example, creating an Amazon S3 bucket and setting permissions for it); however, managed services typically require far less configuration. For example, if you have a static website that you're hosting in a cloud-based storage solution such as Amazon S3 without a web server, those features (scaling, fault-tolerance, and availability) would be automatically handled internally by Amazon S3, because it is a managed solution.

Now, let's look at the challenges of running an unmanaged, standalone relational database. Then we will see how Amazon RDS addresses these challenges.

From On-Premises to Amazon RDS



© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.



What do we mean by *managed services*? Let's take a look.

When your database is on-premises, the database administrator is responsible for everything from app and query optimization to standing up the hardware, patching the hardware, and setting up networking, power and HVAC.

If you move to a database running on an Amazon EC2 instance, you no longer have to manage the underlying hardware or handle data center operations. However, you're still responsible for patching the operating system and handling all software and backup operations.

If you set up your database on Amazon RDS or Amazon Aurora, you free yourself from the administrative responsibilities. By moving to the cloud, you can automatically scale your database, enable high availability, manage backups and perform patching so that you can focus on what really matters most – optimizing your application.

Amazon RDS DB Instances

Amazon
RDS



RDS DB
master
instance

DB Instance Class

- CPU
- Memory
- Network Performance

DB Instance Storage

- Magnetic
- General Purpose (SSD)
- Provisioned IOPS

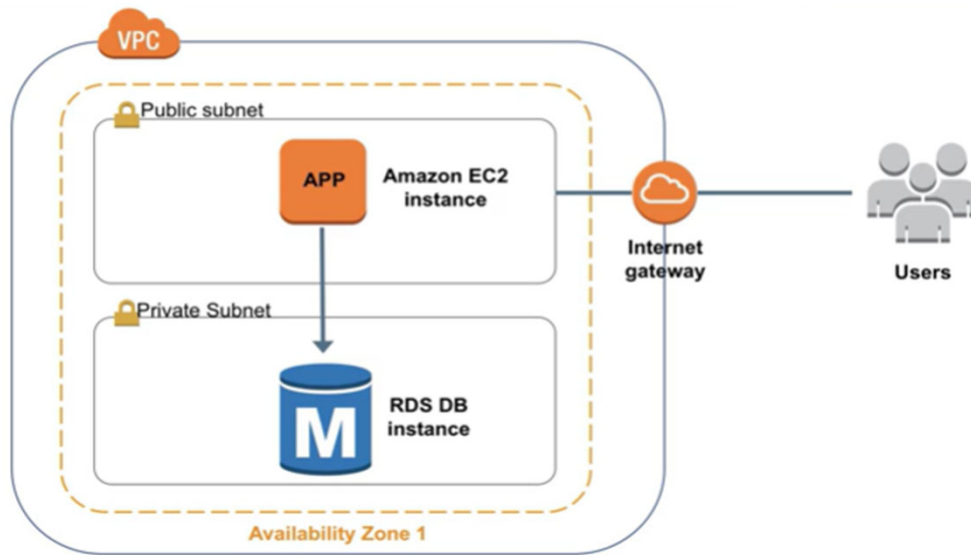


DB Engines

© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

The basic building block of Amazon RDS is the database instance. A *database instance* is an isolated database environment that can contain multiple user-created databases and can be accessed by using the same tools and applications that you use with a standalone database instance. The resources found in a database instance are determined by its database instance class, and the type of storage is dictated by the type of disks. Database instances and storage differ in performance characteristics and price, allowing you to tailor your performance and cost to the needs of your database. When you choose to create a database instance, you first have to specify which database engine to run. Amazon RDS currently supports six databases: MySQL, Amazon Aurora, Microsoft Sequel Server, PostgreSQL, MariaDB, and Oracle.

Amazon RDS In a Virtual Private Cloud

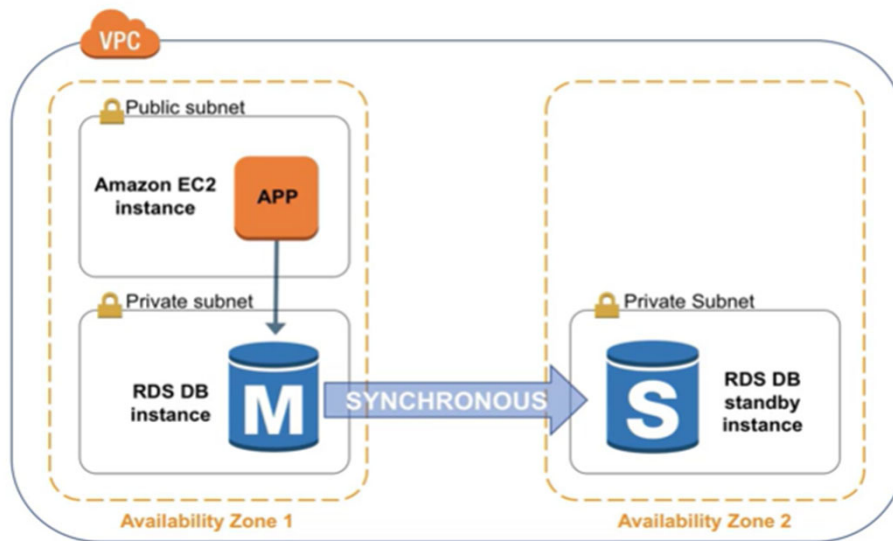


© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

You can run an instance using Amazon Virtual Private Cloud (Amazon VPC). When you use an Amazon VPC, you have control over your virtual networking environment.

You can select your own IP address range, create subnets, and configure routing and access control lists. The basic functionality of Amazon RDS is the same whether or not it is running in an Amazon VPC. Usually the database instance is isolated in a private subnet and is only made directly accessible to indicated application instances. Subnets in an Amazon VPC are associated with a single Availability Zone, so when you select the subnet, you're also choosing the Availability Zone or physical location for your database instance.

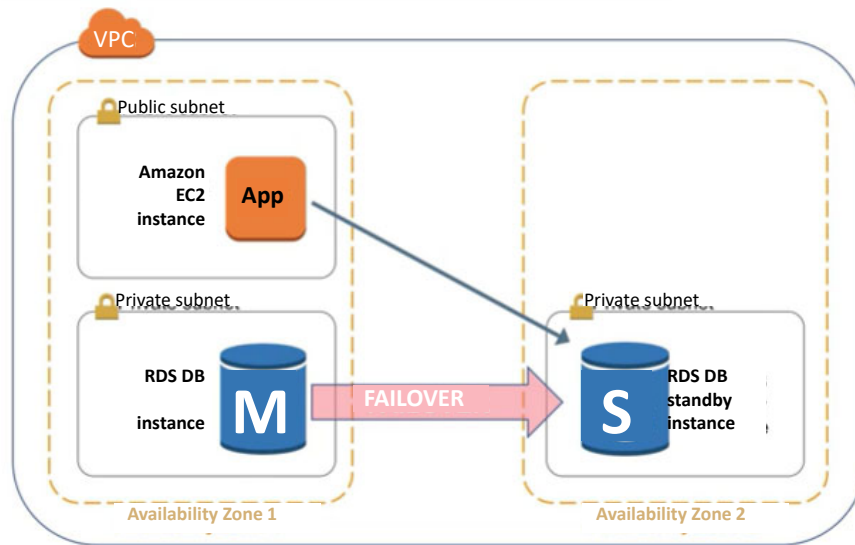
High Availability with Multiple Availability Zones



© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

One of the most powerful features of Amazon RDS is the ability to configure your database instance for high availability with a multi-AZ deployment. Once configured, Amazon RDS automatically generates a standby copy of the database instance in another Availability Zone within the same Amazon VPC. After seeding the database copy, transactions are synchronously replicated to the standby copy. Running a database instance with multiple Availability Zones can enhance availability during planned system maintenance and help protect your databases against database instance failure and Availability Zone disruption.

High Availability with Multiple Availability Zones



© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

If the master database instance fails, Amazon RDS automatically brings the standby database instance online as the new master. Because of the synchronous replication, there should be no data loss. Because your applications reference the database by name using RDS DNS endpoint, you don't need to change anything in your application code to use the standby copy for failover.

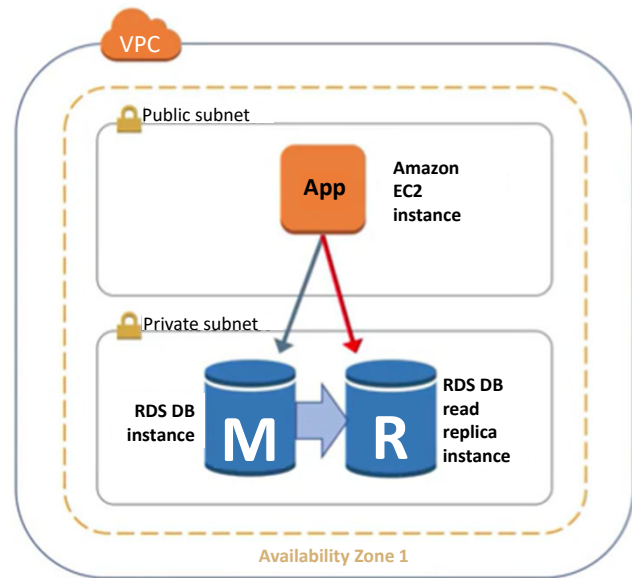
Amazon RDS Read Replicas

Features

- Asynchronous replication
- Promote to master if needed

Functionality

- Read-heavy database workloads
- Offload read queries



© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Amazon RDS also supports the creation of read replicas for MySQL, MariaDB, PostgreSQL, and Amazon Aurora. Updates made to the source database instance are asynchronously copied to the read replica instance. You can reduce the load on your source database instance by routing read queries from your applications to the read replica. Using read replicas, you can also scale out beyond the capacity constraints of a single database instance for read-heavy database workloads. Read replicas can also be promoted to become the master database instance, but due to the asynchronous replication, this requires manual action.

Read replicas can be created in a different region than the master database. This feature can help satisfy disaster recovery requirements or cut down on latency by directing reads to a read replica closer to the user.

*Set up, operate, and scale **relational databases** in the cloud. Features:*

- 📦 Managed service
- 📦 Accessible via the console, AWS RDS CLI, or simple API calls
- 📦 Scalable (compute and storage)
- 📦 Automated redundancy and backup available
- 📦 Supported database engines:
 - 📦 Amazon Aurora, PostgreSQL, MySQL, MariaDB, ORACLE, Microsoft SQL Server



Amazon RDS is a web service that makes it easy to set up, operate, and scale a relational database in the cloud. It provides cost-efficient and resizable capacity while managing time-consuming database administration tasks, so you can focus on your applications and business.

Amazon RDS supports very demanding database applications. You can choose between two SSD-backed storage options: one optimized for high-performance OLTP applications, and the other for cost-effective general-purpose use. With Amazon RDS, you can scale your database's compute and storage resources with no downtime and use the console, the Amazon RDS CLI, or simple API calls to manage the service. Amazon RDS runs on the same highly reliable infrastructure used by other Amazon web services. It also lets you run your database instances and Amazon VPC, which provides you with control and security.

ACF Lab 4

(recommended if you haven't done COS80001)

Build your DB Server and Interact with
your DB Using an App

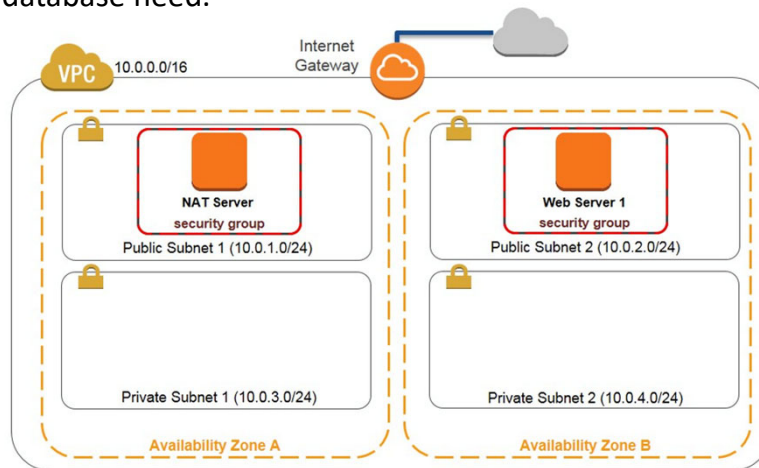


~ 45 minutes

Lab 4 Scenario



This lab is designed to show you how to leverage an AWS-managed database instance for solving relational database need.



© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Amazon RDS makes it easy to set up, operate, and scale a relational database in the cloud. It provides cost-efficient and resizable capacity while managing time-consuming database administration tasks, which allows you to focus on your applications and business. Amazon RDS provides you with six familiar database engines to choose from: Amazon Aurora, Oracle, Microsoft SQL Server, PostgreSQL, MySQL, and MariaDB.

Amazon RDS multi-AZ deployments provide enhanced availability and durability for DB instances, making them a natural fit for production database workloads. When you provision a multi-AZ DB instance, Amazon RDS automatically creates a primary DB instance and synchronously replicates the data to a standby instance in a different Availability Zone.

After completing this lab, you will be able to:

- Launch an Amazon RDS DB instance with high availability.
- Configure the DB instance to permit connections from your web server.
- Open a web application and interact with your database.

Lab 4: Tasks



Security Group

Create a **VPC Security Group**.



Subnet Group

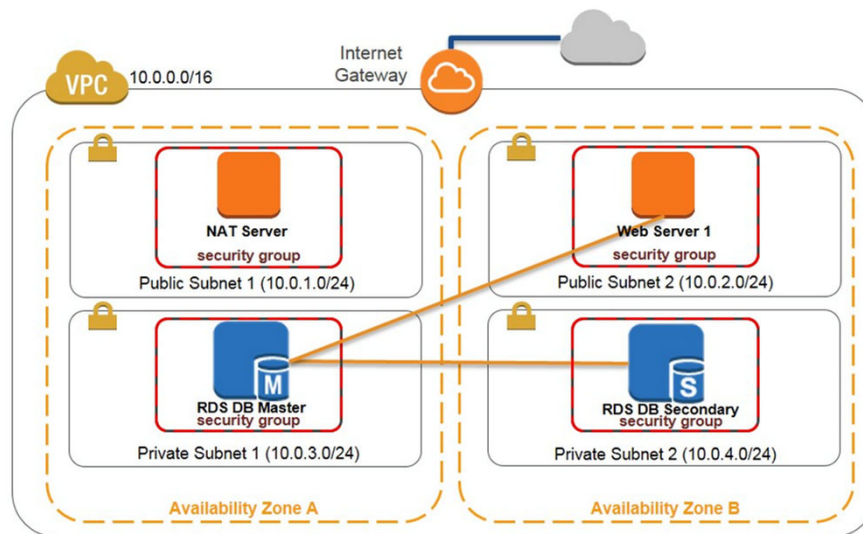
Create a **DB Subnet Group**.



Amazon
RDS

Create an **Amazon RDS DB** instance and interact with your database.

Lab 4: Final Product




~ 45 minutes

© 2018, Amazon Web Services, Inc. or its Affiliates. All rights reserved.

In this lab, you:

- Launched an Amazon RDS DB instance with high availability.
- Configured the DB instance to permit connections from your web server.
- Opened a web application and interacted with your database.

This week



■ Relational vs non-relational 'NoSQL' databases

- ☐ High-level Comparison
- ☐ NoSQL data models

■ *RDS (in brief)*

■ Dynamo DB

- ☐ Key Concepts
- ☐ Partitions and data distribution
- ☐ Secondary indexes
- ☐ Read/write throughput
- ☐ Streams and global tables
- ☐ Basic operation on tables

Developing Flexible NoSQL Solutions with Amazon DynamoDB



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Welcome to Module 5: Developing Flexible NoSQL Solutions with Amazon DynamoDB

- Part 1: Introduction to Amazon DynamoDB
- Part 2: Amazon DynamoDB key concepts
- Part 3: Partitions and data distribution
- Part 4: Secondary indexes
- Part 5: Read/write throughput
- Part 6: Streams and global tables
- Part 7: Backup and restore
- Part 8: Basic operations for Amazon DynamoDB tables

Lab

- Developing with Amazon DynamoDB using the AWS SDK

This module covers:

- Introduction to Amazon DynamoDB
- Amazon DynamoDB key concepts
- Partitions and data distribution
- Secondary indexes
- Read/write throughput
- Streams and global tables
- Backup and restore
- Basic operations for Amazon DynamoDB tables

At the end of this module, you will complete the lab *Developing with Amazon DynamoDB using the AWS SDK*.

Part 1. Introduction to Amazon DynamoDB

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Part 1. Introduction to Amazon DynamoDB

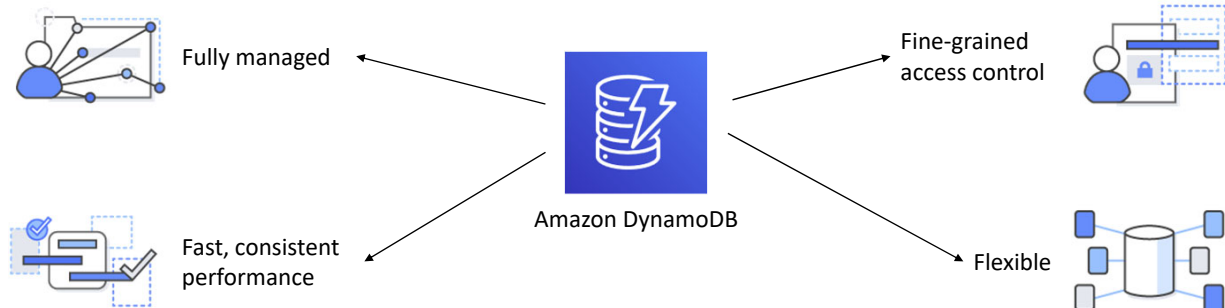


“In 2017, we surveyed 200 of our customers and others and they told us that their **unstructured data is growing between 40-60% per year**, and this is faster growth than any other business data they have. They identified top growing data types as being **rich media – audio, video, images, and research data** – all of which are non-text data types. They also identified **IoT data** as being a growing part of their unstructured data. This is leading to an explosion of unstructured data due [to] the increased size and volume of these data types.”

- Stefaan Vervaet, Sr. Director Solutions Marketing, Data Center Systems

Traditionally, digital data was structured data—that is, data that could be structured into columns and rows, and stored in a relational database. However, according to a 2017 survey conducted by 451 Research and sponsored by Western Digital, unstructured data is growing between 40–60 percent per year. This growth is due to the increased size and volume of unstructured data types, such as audio, video, images, research data, and Internet of Things (IoT) data. (Source: <https://blog.westerndigital.com/2018-data-trends-today-tomorrow-change/>)

Additional reading: <https://datascience.berkeley.edu/structured-unstructured-data/>



Unstructured data can be stored in a non-relational NoSQL database, such as Amazon DynamoDB. Amazon DynamoDB is a fast and flexible non-relational database service for all applications that need consistent, single-digit millisecond latency at any scale. It is a fully managed cloud database and supports both document and key-value store models.

Amazon DynamoDB offers the following benefits:

- **Fully managed** – DynamoDB is a fully managed, non-relational database service—you simply create a database table, set your target utilization for automatic scaling, and let the service handle the rest. You no longer need to worry about database management tasks, such as hardware or software provisioning, setup and configuration, software patching, operating a distributed database cluster, or partitioning data over multiple instances as you scale. DynamoDB also provides point-in-time recovery, backup, and restore for all your tables, helping you meet your corporate and regulatory archival requirements.
- **Fast, consistent performance** – Average service-side latencies are typically single-digit milliseconds. As your data volumes grow and application performance demands increase, DynamoDB uses automatic partitioning and solid state drive (SSD) technologies to meet your throughput requirements and deliver low latencies at any scale.
- **Fine-grained access control** – DynamoDB integrates with AWS Identity and Access Management (IAM) for fine-grained access control of users in your organization. You can assign unique security credentials to each user and control each user's access to services and resources.
- **Flexible** – DynamoDB supports storing, querying, and updating documents. By using the AWS software development kit (SDK), you can write applications that store JavaScript Object Notation (JSON) documents directly into Amazon DynamoDB tables. This capability reduces the amount of new code that must be written to insert, update, and retrieve JSON documents. You can also perform powerful database operations, such as nested JSON queries, by using a few lines of code.

Serverless
web
applications

Microservices
data store

Mobile
backends

Adtech

Gaming

Internet of
Things

Duolingo case study

Duolingo uses Amazon DynamoDB to store 31 billion items in support of an online learning site that delivers lessons for 80 languages. The US startup reaches more than 18 million monthly users around the world who perform more than six billion exercises using the free Duolingo lessons. The company relies heavily on Amazon DynamoDB for its highly scalable database and also for its high performance, which reaches 24,000 read units per second and 3,300 write units per second.

AWS customers in a wide variety of industries use Amazon DynamoDB to store mission-critical data. Financial services, commerce, adtech, IoT, and gaming applications (to name a few) make millions of requests per second to individual tables that contain hundreds of terabytes of data and trillions of items, and they count on DynamoDB to return results in single-digit milliseconds.

There are several use cases for Amazon DynamoDB :

- *Serverless web applications* – Build powerful web applications that automatically scale up and down. You don't need to maintain servers, and your applications have automated high availability.
- *Microservices data store* – Build flexible and reusable microservices using DynamoDB as a serverless data store for consistent and fast performance.
- *Mobile backends* – Build personalized mobile apps with smooth experiences for your users. DynamoDB takes care of operational tasks so that you can focus on your applications.
- *Adtech* – Create real-time bidding platforms and recommendation engines with the scalability, throughput, and availability of DynamoDB.
- *Gaming* – Create responsive games for mobile, console, and desktop with DynamoDB. Store and query game data such as player state, high scores, or world dynamic content.
- *Internet of Things (IoT)* – Analyze your devices by connecting your high-velocity, high-volume IoT data in DynamoDB to Amazon Redshift and Amazon QuickSight.

Duolingo case study

Duolingo is a free language-learning platform that includes a language-learning website and app. Duolingo uses Amazon DynamoDB to store 31 billion items in support of an online learning site that delivers lessons for 80 languages. The US startup reaches more than 18 million monthly users around the world, who perform more than six billion exercises using

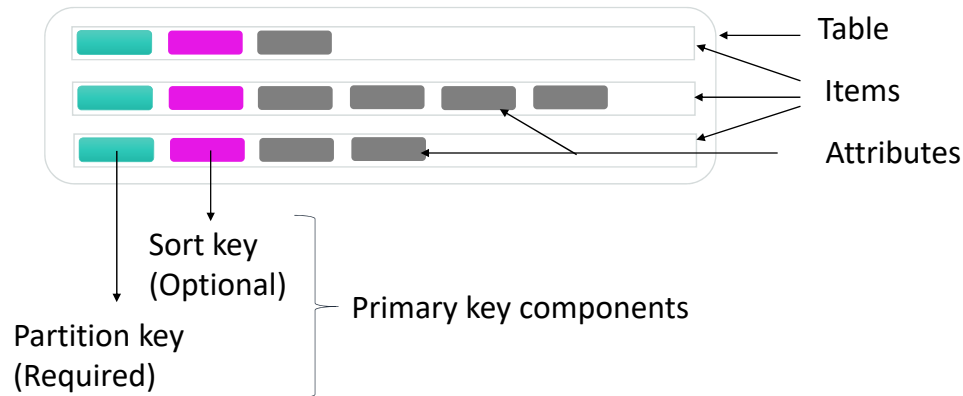
the free Duolingo lessons. The company relies heavily on Amazon DynamoDB for its highly scalable database and also for its high performance, which reaches 24,000 read units per second and 3,300 write units per second. Read the case study: <https://aws.amazon.com/solutions/case-studies/duolingo-case-study-dynamodb/>

Part 2. Amazon DynamoDB key concepts

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Part 2. Amazon DynamoDB key concepts



The basic components of DynamoDB are:

- **Tables** – DynamoDB stores data in tables. A table contains items with attributes.
- **Items** – Each table contains zero or more items. An item is a group of attributes that is uniquely identifiable among all of the other items.
- **Attributes** – Each item is composed of one or more attributes. An attribute is a fundamental data element, something that does not need to be broken down any further.
- **Primary key** – A table has a primary key that uniquely identifies each item in the table. No two items can have the same key.

When compared to the components of a relational database table, items are analogous to rows and attributes are analogous to columns.

Types of primary keys

Simple Primary Key (partition key only)

SensorId (Partition Key)	Latitude	Longitude
SensorA	40.712784	-74.005941
SensorB	35.689488	139.691706

SensorLocation Table

Composite Primary Key (partition key and sort key)

SensorId (Partition Key)	Time (Sort Key)	Value
SensorA	2018-01-03T10:15:30	30
SensorA	2018-01-04T10:19:30	35
SensorB	2018-03-04T11:21:20	28

SensorReadings Table

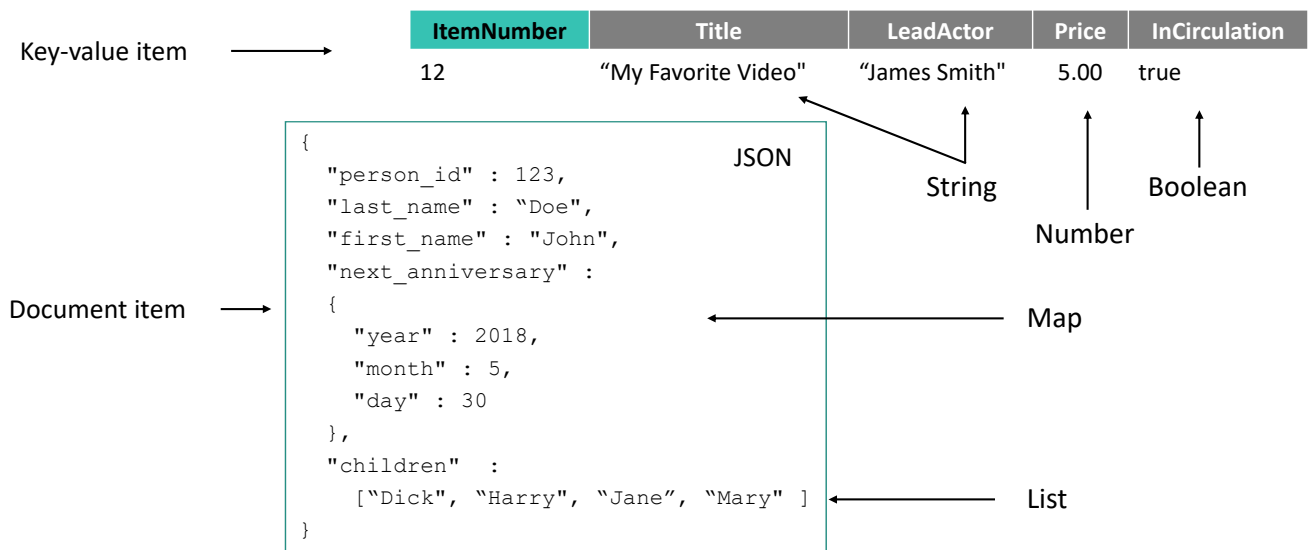
DynamoDB supports two types of primary keys:

- **Simple primary key** – The simple primary key is composed of one attribute known as the *partition key*. DynamoDB builds an unordered index on this primary key attribute. Each item in the table is uniquely identified by its partition key value.
 - In the SensorLocation table, the primary key is a *partition key* because it consists of a single attribute *SensorId*, which is the partition key. Each item in the table is uniquely identified by its partition key value (*SensorA* and *SensorB*). Also, each sensor has exactly one location, which is expressed in terms of latitude and longitude.
- **Composite primary key** – The composite primary key is composed of two attributes: the *partition key* and the *sort key*. DynamoDB builds an unordered index on the partition key attribute and a sorted index on the sort key attribute. In a table that has a partition key and a sort key, it's possible for two items to have the same partition key value. However, those two items must have different sort key values.
 - In the SensorReadings table, the primary key is a *partition and sort primary key* because it is composed of the *SensorId* attribute (the partition key) and the *Time* attribute (the sort key). For each *SensorId*, there might be multiple items corresponding to sensor readings at different times. The combination of *SensorId* and *Time* uniquely identifies items in the table. This design enables you to query the table for all readings related to a

particular sensor.

Remember that DynamoDB can be used as a key-value store and document store. In the examples shown, the primary key value (partition key and sort key if present) is the key, and the remaining attributes constitute the value that corresponds to the key.

Items and attribute types



Unlike a relational database, DynamoDB is not constrained by a pre-defined schema. An item can have any number of attributes with different value types. Each attribute has a name and a value. An attribute value can be one of the following types:

- Scalar types – Number, String, Binary, Boolean, and Null
- Multi-valued types – String Set, Number Set, and Binary Set
- Document types – List and Map

The size of an item is the sum of the lengths of its attribute names and values. An item can be a maximum of 400 KB in size.

Recorded demo: Amazon DynamoDB

35



Set up demo Amazon DynamoDB

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Now, take a moment to watch the [DynamoDB demo](#). The recording runs a little over 2 minutes, and it reinforces many of the concepts that were discussed in this section of the module.

The demonstration shows how to create a table running in Amazon DynamoDB by using the AWS Management Console. It also demonstrates how to interact with the table using the AWS Command Line Interface. The demonstration shows how you can query the table, and add data to the table.

.

Part 3: Partitions and data distribution

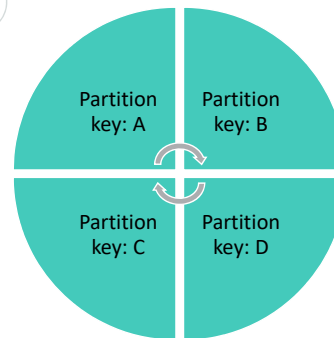
© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Part 3: Partitions and data distribution



Table data is stored in partitions
based on the **partition key**.

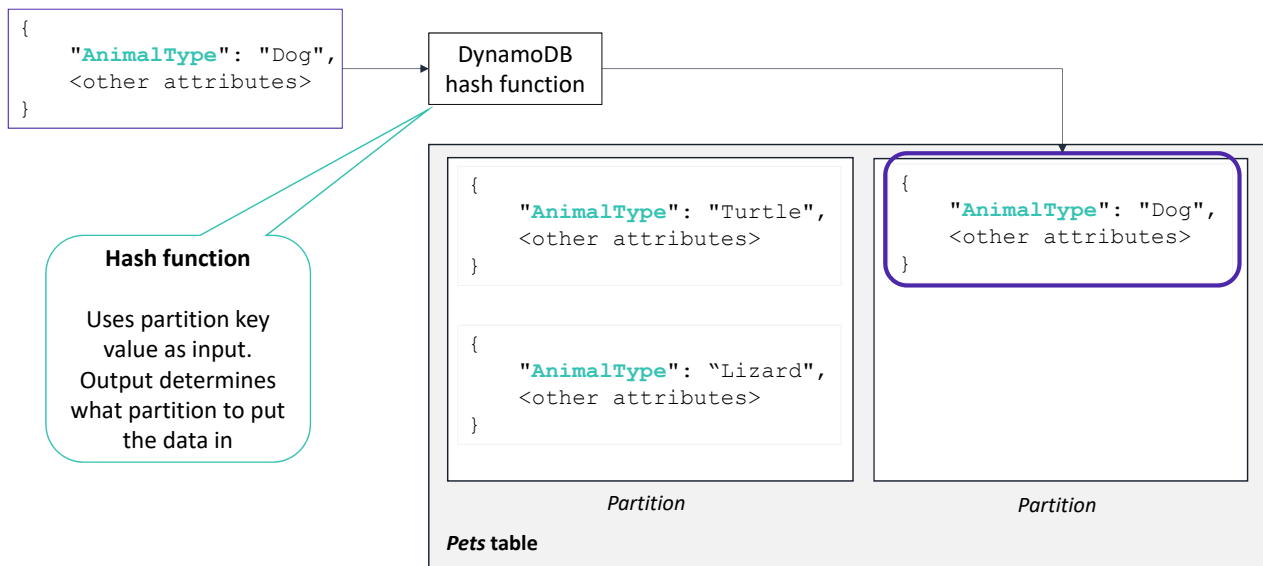


DynamoDB stores data in partitions. A *partition* is an allocation of storage for a table, backed by solid state drives (SSDs), and automatically replicated across multiple Availability Zones within an AWS Region. Partition management is handled entirely by DynamoDB.

If your table has a simple primary key (partition key only), DynamoDB stores and retrieves each item based on its partition key value. The partition key of an item is also known as its *hash attribute*. If a table has a composite primary key (partition key and sort key), then DynamoDB will store all of the items with the same partition key value physically close together and order them by sort key value in the partition. The sort key of an item is also known as its *range attribute*.

For information about how partitioning works, see Partitions and Data Distribution: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.Partitions.html>

Partitioning with simple primary key: Example



In this example, the *Pets* table has a simple key (partition key only) and spans multiple partitions. DynamoDB calculates the hash value of the partition key to determine which partition should contain the item, which in this case, is based on the hash value of the string *Dog*. Note that the items are not stored in sorted order.

Partitioning with composite primary key: Example

```
{
  "AnimalType": "Dog",
  "Name": "Rover",
  <other attributes>
}
```

DynamoDB
hash function

DynamoDB stores the new
item among the others with
the same partition key in
ascending order by sort key.

```
{
  "AnimalType": "Turtle",
  "Name": "Ernest",
  <other attributes>
}
```

```
{
  "AnimalType": "Lizard",
  "Name": "Quincy",
  <other attributes>
}
```

Partition

Pets table

```
{
  "AnimalType": "Dog",
  "Name": "Fido",
  <other attributes>
}
```

```
{
  "AnimalType": "Dog",
  "Name": "Rover",
  <other attributes>
}
```

```
{
  "AnimalType": "Dog",
  "Name": "Spot",
  <other attributes>
}
```

Partition

Suppose that the *Pets* table has a composite primary key consisting of *AnimalType* (partition key) and *Name* (sort key). DynamoDB again calculates the hash value of the partition key to determine which partition should contain the item. In that partition, there are several items with the same partition key value, so DynamoDB stores the new item among the others with the same partition key, in ascending order by sort key. In this example, DynamoDB writes an item with a partition key value of *Dog* and a sort key value of *Fido* in ascending order.

Part 4: Secondary indexes



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Part 4: Secondary indexes

Amazon DynamoDB provides fast access to items in a table by specifying primary key values. However, many applications might benefit from having one or more secondary (or alternate) keys available, which allows for efficient access to data with attributes other than the primary key. To address this, you can create one or more secondary indexes on a table.

- Enable you to query data based on non-primary key attributes.
 - A secondary index defines an alternate key.
- Contain:
 - Alternate key attributes.
 - Primary key attributes.
 - Optional subset of other attributes from the base table (**projected** attributes).
- Can be one of two types:
 - Global secondary index (GSI).
 - Local secondary index (LSI).

A *secondary index* enables you to perform queries on attributes that are not part of the table's primary key. A secondary index lets you query the data in the table by using an alternate key, in addition to queries against the primary key.

In addition to the alternate key attributes and primary key attributes (partition key and sort key), a secondary index contains a subset of the other table attributes. When you create an index, you specify which attributes will be copied, or projected, from the base table to the index. At a minimum, DynamoDB projects the key attributes from the base table into the index.

DynamoDB supports two types of secondary indexes:

- *Global secondary index* – An index with a partition key and a sort key, both of which can be different from those on the base table. A global secondary index is considered *global* because queries on the index can span all of the data in the base table, across all partitions. A global secondary index has no size limitations and has its own provisioned throughput settings for read and write activity that are separate from those of the table.
- *Local secondary index* – An index that has the same partition key as the base table, but a different sort key. A local secondary index is *local* in the sense that every partition of a local secondary index is scoped to a base table partition that has the same partition key value. As a result, the total size of indexed items for any one partition key value can't exceed 10 GB. Also, a local secondary index shares provisioned throughput settings for read and write activity with the table it is indexing.

Each table in DynamoDB has a limit of 20 global secondary indexes (default limit) and 5 local secondary indexes per table.

For more information on secondary indexes, see

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SecondaryIndexes.html>.

Global secondary index example

Music Table

```
{
  "Artist": "No One You Know",
  "SongTitle": "My Dog Spot",
  "AlbumTitle": "Hey Now",
  "Price": 1.98,
  "Genre": "Country",
  "CriticRating": 8.4
}

{
  "Artist": "No One You Know",
  "SongTitle": "Somewhere Down the Road",
  "AlbumTitle": "Somewhat Famous",
  "Genre": "Country",
  "CriticRating": 8.4
  "Year": 1984
}

{
  "Artist": "The Acme Band",
  "SongTitle": "Look Out, World",
  "AlbumTitle": "The Buck Starts Here",
  "Price": 0.99
  "Genre": "Rock"
}
```

GenreAlbumTitle Secondary Index

```
{
  "Genre": "Country",
  "AlbumTitle": "Hey Now",
  "Artist": "No One You Know",
  "SongTitle": "My Dog Spot",
}

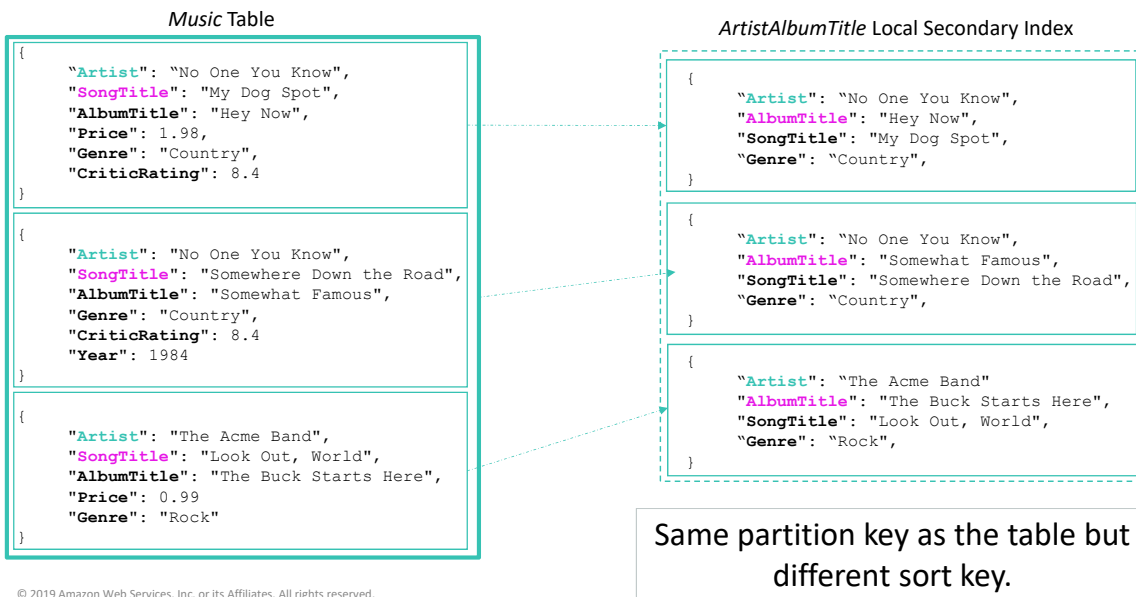
{
  "Genre": "Country",
  "AlbumTitle": "Somewhat Famous",
  "Artist": "No One You Know",
  "SongTitle": "Somewhere Down the Road",
}

{
  "Genre": "Rock",
  "AlbumTitle": "The Buck Starts Here",
  "Artist": "The Acme Band",
  "SongTitle": "Look Out, World",
}
```

Different partition key and sort key than the table.

Here is an example of when you would use a global secondary index. Say you have a *Music* table. You can query data items by *Artist* (partition key) or by *Artist* and *SongTitle* (partition key and sort key). What if you also wanted to query the data by *Genre* and *AlbumTitle*? You cannot do this query with the *Music* table. To do this, you create a global secondary index on *Genre* and *AlbumTitle* from the base *Music* table, and then query the index in much the same way that you would query the *Music* table. In the index, *Genre* is the partition key and *AlbumTitle* is the sort key. Note that the combination of *Genre* and *Album Title* might not be unique. Multiple albums can belong to one genre.

Local secondary index example



Now, say that you want to query by *Artist* and by *AlbumTitle*. In this case, you would create a local secondary index (called *ArtistAlbumTitle*) from the base *Music* table. The local secondary index has the same partition key as the base table (*Artist*) but a different sort key (*AlbumTitle*).

For a local secondary index, the partition key is the same as the table's partition key. The sort key can be any scalar attribute.

Part 5: Read/write throughput

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Part 5: Read/write throughput

- DynamoDB maintains multiple copies of data for durability.
 - All copies of your data are usually consistent within a second after a write operation.
- **Eventually consistent** read might return slightly stale data if a read operation is performed immediately after a write operation.
 - Use case: blog posts
- **Strongly consistent** read returns most up-to-date data.
 - Use cases: score boards, booking systems, financial algorithms
- You can specify the desired consistency level when reading data.

DynamoDB automatically replicates your data across multiple Availability Zones in an AWS Region, which provides built-in high availability and data durability. All copies of your data are usually consistent within a second after a write operation.

DynamoDB supports *eventually consistent* and *strongly consistent* reads.

- **Eventually Consistent Reads** – When you read data from a DynamoDB table, the response might not reflect the results of a recently completed write operation. The response might include some stale data. If you repeat your read request after a short time, the response should return the latest data.
- **Strongly Consistent Reads** – When you request a strongly consistent read, DynamoDB returns a response with the most up-to-date data, which reflects the updates from all prior write operations that were successful. A strongly consistent read might not be available if there is a network delay or outage. Consistent reads are not supported on global secondary indexes.

You can specify the desired consistency level when reading data.

For more information on read consistency, see the AWS Documentation:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html>

- Simplify the developer experience of making **coordinated, all-or-nothing changes** (inserts, deletes, or updates) to multiple items both within and across tables.
- Provide **atomicity, consistency, isolation, and durability (ACID)**, enabling you to maintain data correctness in your applications easily.
- Use cases
 - Processing financial transactions
 - Fulfilling and managing orders
 - Building multiplayer game engines
 - Coordinating actions across distributed components and services

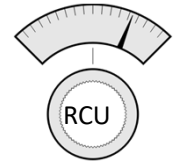
Amazon DynamoDB transactions simplify the developer experience of making coordinated, all-or-nothing changes to multiple items both within and across tables. Transactions provide atomicity, consistency, isolation, and durability (ACID) in DynamoDB, and they enable you to maintain data correctness in your applications.

Many use cases are easier and faster to implement using transactions, for example:

- Processing financial transactions
- Fulfilling and managing orders
- Building multiplayer game engines
- Coordinating actions across distributed components and services

For more information about Amazon DynamoDB Transactions, see the Developer Guide:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/transactions.html>

- **Throughput** = maximum amount of capacity that an application can consume from a table or index
- **Divided evenly** among partitions
- **Read capacity unit (RCU)**
 - Number of strongly consistent reads per second of items that are up to 4 KB in size
 - Eventually consistent reads use half of the provisioned read capacity
- **Write capacity unit (WCU)**
 - Number of 1 KB writes per second



Amazon DynamoDB supports both provisioned and on-demand throughput. *Provisioned throughput* is the maximum amount of capacity that an application can consume from a table or index. If your application exceeds your provisioned throughput capacity on a table or index, it is subject to request throttling.

DynamoDB divides throughput evenly among partitions. The throughput per partition is the total provisioned throughput divided by the number of partitions.

With provisioned throughput, you specify the throughput capacity in terms of *read capacity units (RCU)* and *write capacity units (WCU)*:

- A *read capacity unit* is the number of strongly consistent reads per second of items that are up to 4 KB in size. If you perform eventually consistent reads, you use half of the read capacity units that are provisioned. In other words, for eventually consistent reads, one read capacity unit is *two* reads per second for items that are up to 4 KB.
- A *write capacity unit* is the number of 1 KB writes per second.

- Pay-per-request pricing for read and write requests
- DynamoDB adapts rapidly to accommodate the workload
- Request rate limited by throughput default table limits
- Create or update a table to use on-demand mode
- Use cases
 - You create new tables with unknown workloads.
 - You have unpredictable application traffic.
 - You prefer the ease of paying for only what you use.

When you choose *Amazon DynamoDB on-demand*, DynamoDB instantly accommodates your workloads as they ramp up or down to any previously reached traffic level. If a workload's traffic level reaches a new peak, DynamoDB adapts rapidly to accommodate the workload. Tables that use on-demand mode deliver the same single-digit millisecond latency, service-level agreement (SLA) commitment, and security that DynamoDB already offers. You can choose on-demand for both new and existing tables, and you can continue using the existing DynamoDB application programming interfaces (APIs) without changing code. DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you pay only for what you use.

On-demand mode is a good option if any of the following situations are true:

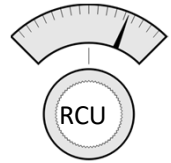
- You create new tables with unknown workloads.
- You have unpredictable application traffic.
- You prefer the ease of paying for only what you use.

For more information about Amazon DynamoDB on-demand and provisioned modes, see *Read/Write Capacity Mode*:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html>

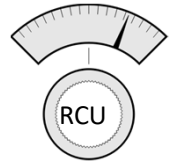
Activity: Calculating RCU

- Say that you need to read 20 items that are 11 KB in size every second, with eventual consistency.
- How many RCUs must you provision?
- 1 RCU = 1 strongly consistent , or 2 eventually consistent, reads per second for items up to 4 KB in size



Say that you need to read 20 items that are 11 KB in size every second, with eventual consistency. How many RCUs do you need?

- Answer: **30 RCU**
- Round 11 KB up to the next multiple of 4: *12*
- Divide by 4 KB per RCU: *3*
- Multiply by items read per second: *60*
- Divide by 2 for eventual consistency: *30*



Answer: You need 30 RCU.

Compute as follows:

- Round 11 KB up to the next multiple of 4: *12*
- Divide by 4 KB per RCU: *3*
- Multiply by items read per second: *60*
- Divide by 2 for eventual consistency: *30*

Activity: Calculating WCU

- Say that you need to write 120 items that are 7 KB in size every minute.
- How many WCUs must you provision?
- 1 WCU = 1 write per second, for items up to 1 KB in size



Say that you need to write 120 items that are 7 KB in size every minute. How many WCUs do you need?

Calculating WCU: Answer

Answer: 14 WCU



- WCU required to write each 7 KB item: 7
- Number of items per second: 2
- Multiply WCU per item by items per second: 14

Answer: You need 14 WCU.

Compute as follows:

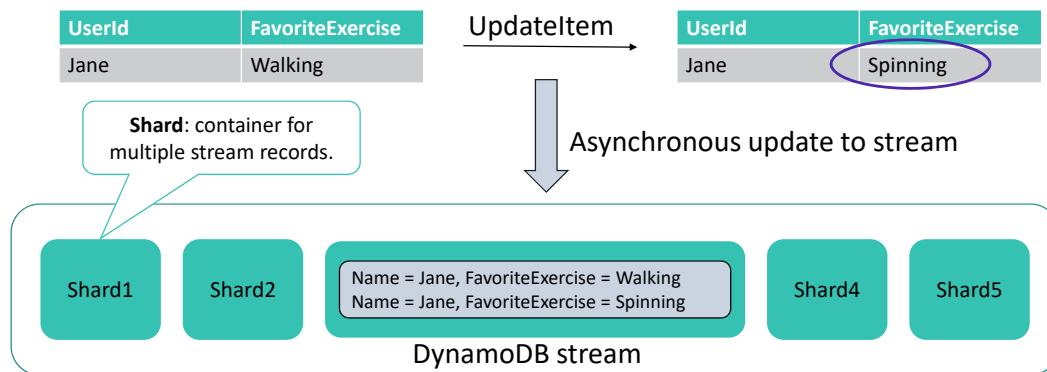
- WCU required to write each 7 KB item: 7
- Number of items per second: 2
- Multiply WCU per item by items per second: 14

Part 6: Streams and global tables

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Part 6: Streams and global tables



Many applications can benefit from the ability to capture changes to items that are stored in a DynamoDB table at the point in time when these changes occur. For example, consider an application that makes changes to user preferences that are stored in a DynamoDB table. Another application, such as an ad server, needs to respond to the new preferences and present different advertisements.

You can enable *DynamoDB Streams* as a solution to this type of use case. This is an optional feature that captures a time-ordered sequence of item-level modifications in any DynamoDB table (called a *stream*), and stores this information for up to 24 hours. Applications can view the data items as they appeared both before and after they were modified, in near real time.

A stream consists of *stream records*. Each stream record represents a single data modification in the DynamoDB table to which the stream belongs. Each stream record is assigned a sequence number, which reflects the order in which the record was published to the stream. Whenever an application creates, updates, or deletes items in the table, DynamoDB Streams writes a stream record with the primary key attributes of the items that were modified. You can configure the stream so that the stream records capture

additional information, such as the *before* and *after* images of modified items.

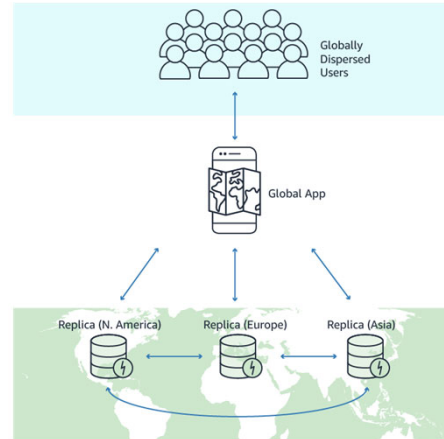
Stream records are organized into groups, or *shards*. Each shard acts as a container for multiple stream records, and it contains information that is required for accessing and iterating through these records. The stream records in a shard are automatically removed after 24 hours.

For information about working with streams, see Capturing Table Activity with DynamoDB streams:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html>

Global tables

- Solution for deploying a multi-Region, multi-master database without having to build and maintain your own replication solution.
- Specify the AWS Regions where you want the table to be available.



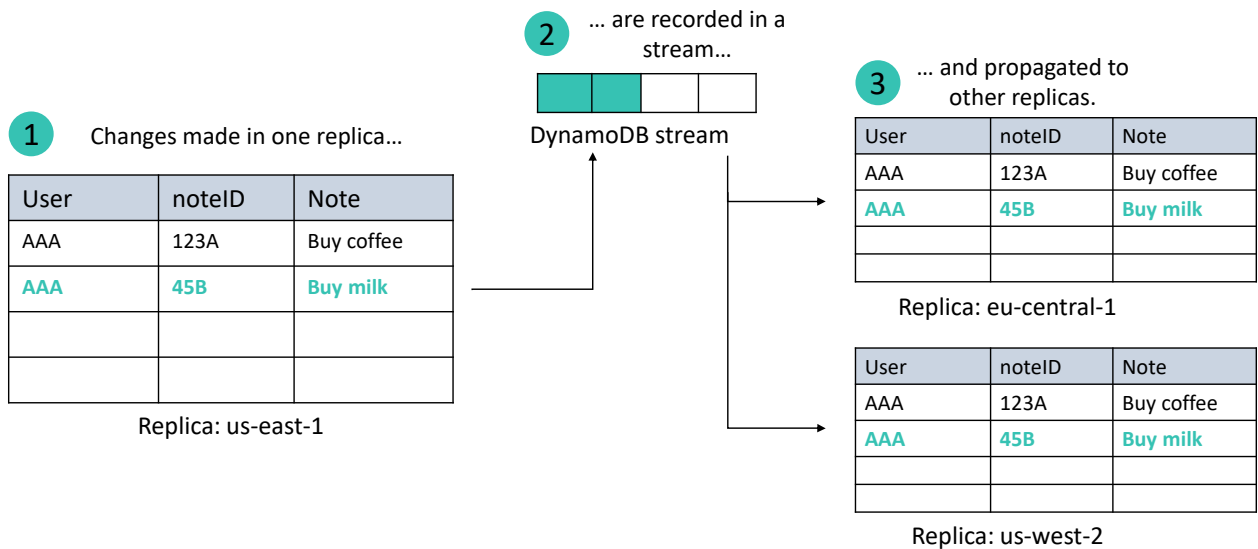
DynamoDB automatically spreads the data and traffic for your tables over a sufficient number of servers to handle your throughput and storage requirements, while maintaining consistent and fast performance. All of your data is stored on solid state disks (SSDs) and automatically replicated across multiple Availability Zones in an AWS Region, which provides built-in high availability and data durability. You can use *global tables* to keep DynamoDB tables in sync across AWS Regions.

Global tables provide a fully managed solution for deploying a multi-Region, multi-master database, without having to build and maintain your own replication solution. When you create a global table, you specify the AWS Regions where you want the table to be available. DynamoDB performs all of the necessary tasks to create identical tables in these Regions, and to propagate ongoing data changes to all of them.

To illustrate one use case for a global table, suppose that you have a large customer base spread across three geographic areas—North America, Europe, and Asia. Customers need to update their profile information while using your application. To address these requirements, you could create three identical DynamoDB tables named *CustomerProfiles* in three different AWS Regions. These three tables would be entirely separate from each other, and changes to the data in one table would not be reflected in the other tables. Without a managed replication solution, you could write code to replicate data changes among these tables; however, this would be a time-consuming and labor-intensive effort.

Instead of writing your own code, you could create a global table consisting of your three Region-specific *CustomerProfiles* tables. DynamoDB would then automatically replicate data changes among those tables, so that changes to *CustomerProfiles* data in one Region would be seamlessly propagated to the other Regions. In addition, if one of the AWS Regions were to become temporarily unavailable, your customers could still access the same *CustomerProfiles* data in the other Regions.

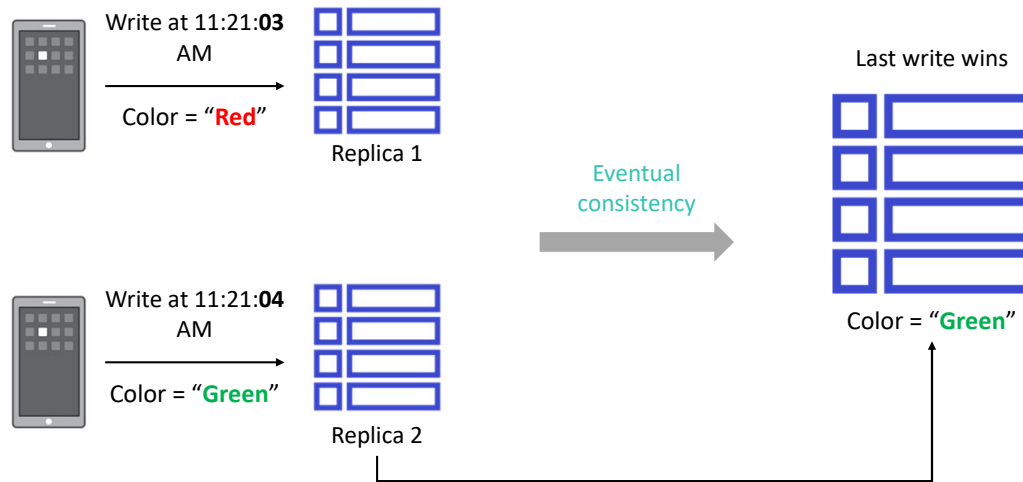
Global tables: Data replication



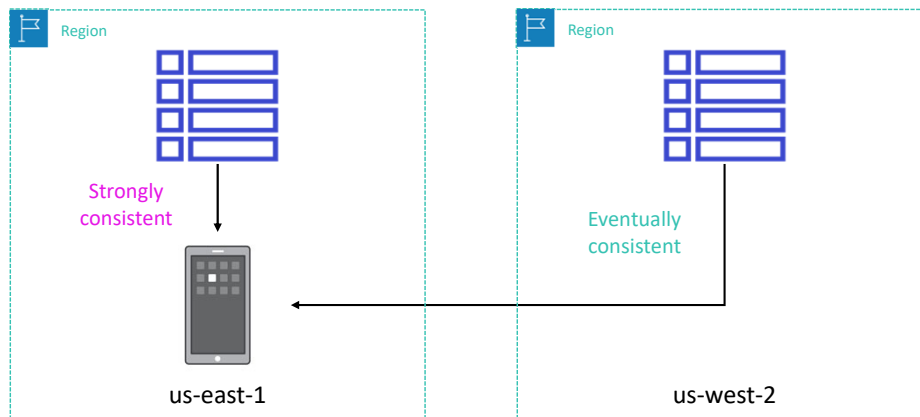
A global table is a collection of one or more DynamoDB tables called *replica tables*. A replica table (or replica, for short) is a single DynamoDB table that functions as a part of a global table. Each replica stores the same set of data items. Any given global table can only have one replica table per Region, and every replica has the same table name and the same primary key schema.

Global tables use DynamoDB Streams to propagate changes between replicas. Any changes that are made to any item in any replica table will be recorded in a stream and replicated to all of the other replicas within the same global table. In a global table, a newly written item is usually propagated to all replica tables within seconds. With a global table, each replica table stores the same set of data items. DynamoDB does not support partial replication of only some of the items.

Global tables: Concurrent updates



Conflicts can arise if applications update the same item in different Regions at about the same time. To achieve eventual consistency, DynamoDB global tables use a "last writer wins" reconciliation between concurrent updates, where DynamoDB makes a best effort to determine the last writer. With this conflict resolution mechanism, all of the replicas will agree on the latest update, and converge toward a state in which they all have identical data.



An application can read and write data to any replica table. If your application only uses eventually consistent reads, and only issues reads against one AWS Region, then it will work without any modification. However, if your application requires strongly consistent reads, then it must perform all of its strongly consistent reads and writes in the same Region. DynamoDB does not support strongly consistent reads across AWS Regions. Therefore, if you write to one Region and read from another Region, the read response might include stale data that doesn't reflect the results of recently completed writes in the other Region. *Strongly consistent reads require using a replica in the same Region where the client is running.*

Transactions are enabled for all single-Region DynamoDB tables and are disabled on global tables by default. You can choose to enable transactions on global tables by request, but replication across Regions is asynchronous and eventually consistent. You might observe partially completed transactions during replication to other Regions. Additionally, simultaneous writes to the same item in different Regions are not enabled to be serially isolated.

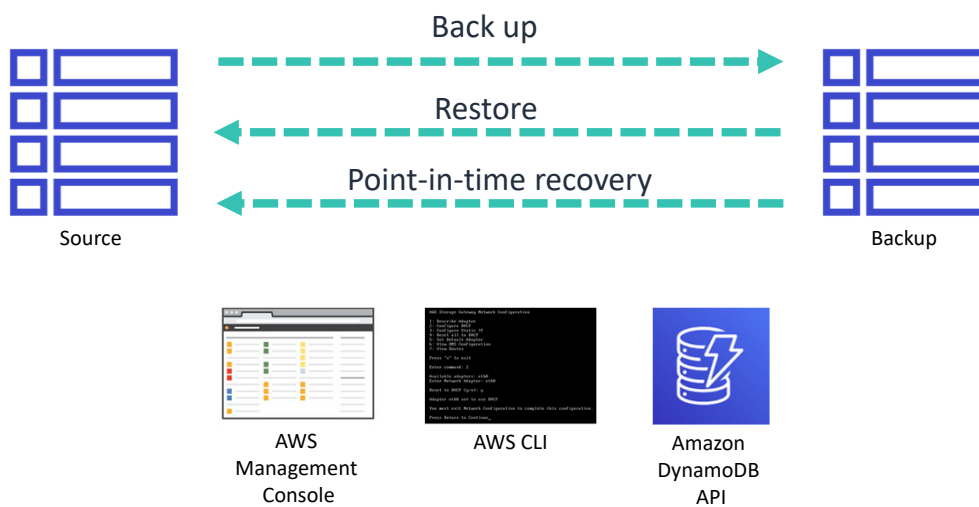
Part 7: Backup and restore

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.



Part 7: Backup and restore

On-demand backups and point-in-time recovery



On-demand backups

Amazon DynamoDB provides on-demand backup and restore capabilities. You can use DynamoDB to create full backups of your tables so that you can meet your regulatory requirements for long-term retention and archiving. You can back up and restore your DynamoDB table data any time with a single click in the AWS Management Console or with a single application programming interface (API) call. Backup and restore actions execute with zero impact on table performance or availability.

When you create an on-demand backup, a time marker of the request is cataloged. The backup is created asynchronously by applying all changes until the time of the request to the last full table snapshot. Backup requests are processed instantaneously and become available for restore within minutes.

Each time you create an on-demand backup, the entire table data is backed up. There is no limit to the number of on-demand backups that you can take. All backups in DynamoDB work without consuming any provisioned throughput on the table.

Point-in-time recovery

Point-in-time recovery helps protect your Amazon DynamoDB tables from accidental write or delete operations. With point-in-time recovery, you don't have to worry

about creating, maintaining, or scheduling on-demand backups. For example, suppose that a test script writes accidentally to a production DynamoDB table. With point-in-time recovery, you can restore that table to any point in time during the last 35 days. DynamoDB maintains incremental backups of your table.

Part 8: Basic operations for Amazon DynamoDB tables



© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Part 8: Basic operations for Amazon DynamoDB tables

The Amazon DynamoDB API allows you to invoke operations on DynamoDB tables from an application. In this section, you will learn about operations that you can perform on tables and items that are supported by the DynamoDB API.

- **Control operations** – Create and manage DynamoDB tables.
 - E.g., CreateTable
- **Data operations** – Create, read, update, and delete (*CRUD*) actions on data in a table.
 - E.g., Write operations: PutItem, UpdateItem, DeleteItem,
 - E.g., Read operations: GetItem, Query, Scan
- **Batch operations** – Read and write batches of items in a DynamoDB table
- **Transaction operations** – Make coordinated, all-or-nothing changes to multiple items both within and across tables

You can use the Amazon DynamoDB API to invoke the following types of operations from an application:

- *Control operations* – Let you create and manage DynamoDB tables. These operations also let you work with indexes, streams, and other objects that depend on tables.
- *Data operations* – Let you perform create, read, update, and delete (CRUD) actions on data in a table. CRUD refers to all of the major functions that are implemented in relational database applications. Some of the data operations also let you read data from a secondary index.
- *Batch operations* – Let you enable or disable a stream on a table, and allow access to the data modification records that are contained in a stream.
- *Transaction operations* – Simplify the developer experience of making coordinated, all-or-nothing changes to multiple items both within and across tables.

For more information on basic table operations, see the AWS Documentation:

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithTables.Basics.html>

Creating a table

TableName is the name of the table to create.

KeySchema specifies the attributes that make up the primary key for the table.

AttributeDefinitions is an array of attributes that describes the key schema for the table.

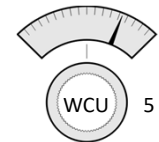
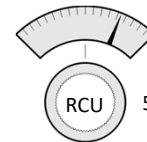
ProvisionedThroughput represents the provisioned throughput settings for the table.

```
table = dynamodb.create_table(  
    TableName = 'users',  
    KeySchema = [  
        {  
            'AttributeName': 'username',  
            'KeyType': 'HASH'  
        },  
        {  
            'AttributeName': 'last_name',  
            'KeyType': 'RANGE'  
        }  
    ],  
    AttributeDefinitions = [  
        {  
            'AttributeName': 'username',  
            'AttributeType': 'S'  
        },  
        {  
            'AttributeName': 'last_name',  
            'AttributeType': 'S'  
        }  
    ],  
    ProvisionedThroughput = {  
        'ReadCapacityUnits': 5,  
        'WriteCapacityUnits': 5  
    }  
)
```

Create_table:
Creates a new table.

users Table

username (Partition Key)	last_name (Sort Key)
-----------------------------	-------------------------



CreateTable is an asynchronous operation that creates a new table. On receiving a *CreateTable* request, DynamoDB immediately returns a response with a *TableStatus* of *CREATING*. After the table is created, DynamoDB sets the *TableStatus* to *ACTIVE*. You can perform read and write operations only on an *ACTIVE* table. You can optionally define secondary indexes on the new table as part of the *CreateTable* operation.

This example shows how to create a table using Boto 3 (the AWS SDK for Python). The invocation of the `create_table` method creates a table named **users** with a partition-and-sort primary key that is composed of the attributes named **username** and **last_name** (which have hash and range key types, respectively). The provisioned throughput for the table is 5 RCU and 5 WCU.

For more information on the *CreateTable* operation and the parameters it takes, see the AWS Documentation:

https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_CreateTable.html

Creating an item

Put_item: Creates a new item or replaces (overwrites) an existing item.

Item is a map of attribute name-value pairs, one for each attribute. Only the primary key attributes are required. You can optionally provide other attribute name-value pairs for the item.

users Table Before

username (Partition key)	last_name (Sort key)
-----------------------------	-------------------------

```
table.put_item(  
    Item = {  
        'username': 'janedoe',  
        'first_name': 'Jane',  
        'last_name': 'Doe',  
        'age': 25,  
        'account_type': 'standard_user'  
    }  
)
```

users Table
After

username (Partition key)	last_name (Sort key)	first_name	Age	account_type
janedoe	Doe	Jane	25	standard_user

The *PutItem* operation creates a new item or replaces an old item with a new item. If an existing item in the specified table has the same primary key as the new item, the new item completely replaces the existing item. You can perform a conditional PUT operation (add a new item if an item with the specified primary key doesn't exist), or you can replace an existing item if it has certain attribute values. You can return the item's attribute values in the same operation by using the *ReturnValues* parameter.

Continuing with the Python example, after the **users** table has been created and assigned to the **table** variable, the `put_item` method is invoked to add a new item to the table.

For more information on the *PutItem* operation and the parameters it takes, see the AWS Documentation:

https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_PutItem.html

Updating an item

Update_Item:

Edits an existing item's attributes, or adds a new item to the table if it does not already exist.

users Table
Before

username (Partition key)	last_name (Sort key)	first_name	Age	account_type
janedoe	Doe	Jane	25	standard_user

Key is the primary key of the item to be updated.

UpdateExpression defines one or more attributes to be updated.

ExpressionAttributeValues is one or more values that can be substituted in an expression.

```
table.update_item(  
    Key = {  
        'username': 'janedoe',  
        'last_name': 'Doe'  
    },  
    UpdateExpression = 'SET age = :vall',  
    ExpressionAttributeValues = {  
        ':vall': 26  
    }  
)
```

users Table
After

username (Partition Key)	last_name (Sort key)	first_name	Age	account_type
janedoe	Doe	Jane	26	standard_user

The *UpdateItem* operation allows you to update an existing item's attributes, or add a new item to the table if it does not already exist. You add, set, or remove attribute values. You can also perform a conditional update on an existing item (for example, replace an existing name-value pair if it has certain expected attribute values). Additionally, you can return the item's attribute values, from either before or after the update, by using the *ReturnValues* parameter.

In this Python example, the `update_item` method is used to modify the age attribute of an item in the **users** table.

For more information on the *UpdateItem* operation and the parameters it takes, see the AWS Documentation:

https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_UpdateItem.html

Deleting an item

Delete_Item:
Deletes a single item in a table by primary key.

users Table
Before

username (Partition key)	last_name (Sort key)	first_name	Age	account_type
janedoe	Doe	Jane	26	standard_user

Key represents the primary key of the item to delete.

```
table.delete_item(  
    Key = {  
        'username': 'janedoe',  
        'last_name': 'Doe'  
    }  
)
```

users Table
After

username (Partition key)	last_name (Sort key)
-----------------------------	-------------------------

The *DeleteItem* operation allows you to delete an item in a table by using its primary key. You can perform a conditional delete operation that deletes the item if it has an expected attribute value. In addition to deleting an item, you can also return the item's attribute values by using the *ReturnValues* parameter.

In this Python example, the `delete_item` method is used to delete an item from the **users** table.

For more information on the *DeleteItem* operation and the parameters it takes, see the AWS Documentation:

https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_DeleteItem.html

AccountStatus Table

userId (primary key)	lastFailedLoginTime	accountLocked
1	2018-06-06T19:20+01:00	Y

UpdateItem Operation

Primary key: `userId` = 1

Set: `accountLocked` = N

ConditionExpression:

`currentLoginTime > lastFailedLoginTime + 24 hours`

For the PutItem, UpdateItem and DeleteItem operations, you can specify a *condition expression* to determine which items should be modified. If the condition expression evaluates to true, the operation succeeds; otherwise, the operation fails.

In the example shown, the `accountLocked` attribute in the `AccountStatus` table can be set to N only if the last failed login attempt was more than 24 hours earlier.

For more information about performing conditional writes, see

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Expressions.ConditionExpressions.html>.

Reading an item

Get_Item:
Reads an item
from a table.

users
Table

username (Partition key)	last_name (Sort key)	first_name	Age	account_type
janedoe	Doe	Jane	25	standard_user

Key represents the
primary key of the
item to retrieve.

```
response = table.get_item (  
    Key = {  
        'username': 'janedoe',  
        'last_name': 'Doe'  
    }  
)  
item = response['Item']  
print (item)
```

Expected
Output

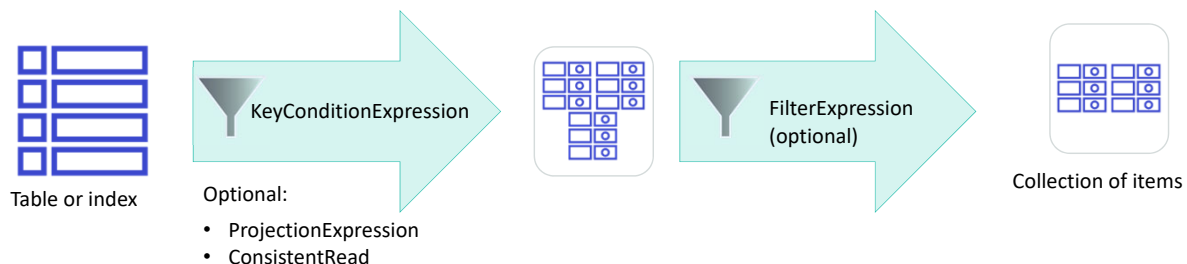
```
{u'username': u'janedoe',  
u'first_name': u'Jane',  
u'last_name': u'Doe',  
u'account_type': u'standard_user',  
u'age': Decimal('25')  
}
```

The *GetItem* operation allows you to retrieve a specific item from a DynamoDB table. You must specify the table name and the full primary key (partition key and sort key, if any) to retrieve a single item from a table. You can optionally specify a projection expression to retrieve only certain attributes instead of retrieving the entire item. By default, all attributes of the item are returned. You can also request that the operation use a strongly consistent read instead of the default eventually consistent read.

In this Python example, the `get_item` method is used to retrieve an item in the **users** table, which is then printed to standard output. The expected result is a JSON object that contains all of the attributes of the item.

See the AWS Documentation for more information on:

- GetItem operation and the parameters it takes:
https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_GetItem.html
- Reading items from a DynamoDB table:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html#WorkingWithItems.ReadingData>



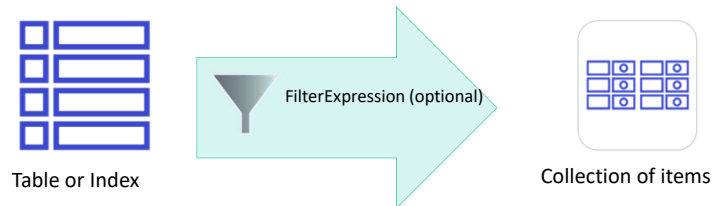
```
response = table.query (
    KeyConditionExpression = Key('username').eq('johndoe')
)
```

The *Query* operation allows you to read only the items that match the primary key from a table or secondary index. The primary key is specified in the key condition expression. If a filter expression is specified, then the *Query* operation further refines the result set based on the filter. You must explicitly specify the name of the table or secondary index that you want to query. The *Query* operation returns a result set with the items that match the conditions that were specified. If none of the items satisfy the given criteria, then the *Query* operation returns an empty result set.

In this Python example, the `query` method is used to retrieve all the items in the **users** table that have a primary key value that is equal to *johndoe*.

See the AWS Documentation for more information on:

- Query operation and the parameters it takes:
https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Query.html
- Working with queries:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.html>



A **scan operation** reads **all items** from the table or index.
A query operation is more efficient than a scan operation.

```
response = table.scan (
    FilterExpression = Attr('age').lt(27)
)
```

The *Scan* operation is similar to a *Query* operation, but the *Scan* operation reads all items from the table or index. The result set can be refined by using a filter expression.

In this Python example, the `scan` method is used to retrieve all the **users** whose age is less than 27.

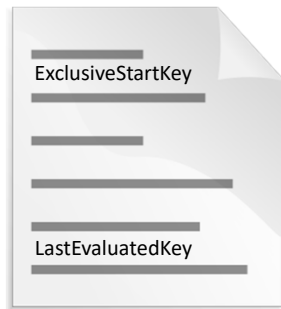
Because the *Scan* operation reads all items from the table or index, it is an expensive operation. It is more efficient to perform a *Query* operation that has the appropriate key condition expression to return only the data that your application needs. If you must do a *Scan* operation, perform a parallel scan when possible.

See the AWS Documentation for more information on:

- Scan operation and the parameters it takes:
https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_Scan.html
- Working with scans:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Scan.html>
- Parallel scans:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Scan.html#Scan.ParallelScan>

Limiting the amount of data returned to improve performance and optimize cost

- **Pagination** limit of 1 MB of data limits the amount of data that Query or Scan operations return.
- **Limit** parameter specifies the maximum number of items that a Query or Scan operation returns.



To improve performance and optimize cost, you can limit the amount of data returned by Query and Scan operations. The number of items returned by a `Query` or `Scan` operation is affected by two factors:

- **Pagination limit** – By default, DynamoDB divides the results of Query and Scan operations into *pages* of data that are 1 MB in size (or less). An application can process the first page of results, then the second page, and so on. A single `Query` or `Scan` will only return a result set that fits within the 1 MB size limit.
- **Limit parameter value** – The `Query` and `Scan` operations also allow you to limit the number of items that are returned in the result. To do this, set the `Limit` parameter to the maximum number of items that you want.

See the AWS Documentation for more information on:

- **Paginating results:**
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.html#Query.Pagination>
- **Limiting the number of items in the result set:**
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Query.html#Query.Limit>
- **Best practices for querying and scanning data:**
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp->

query-scan.html

- **BatchGetItem**: Read up to 16 MB of data that consists of up to 100 items from multiple tables.
- **BatchWriteItem**: Write up to 16 MB of data that consists of up to 25 PUT or DELETE requests to multiple tables.
- If one request in a batch fails, the entire operation does not fail.
 - Retry with failed keys and data returned by operation.

You can use batch operations to achieve higher throughput by writing, deleting, or replacing multiple items in a single request. Batch operations also let you take advantage of parallelism without having to manage multiple threads on your own. You can achieve lower average latencies, compared to using single PutItem or DeleteItem operations, when multiple items are written, deleted, or replaced.

You can use the following operations to retrieve or write data to DynamoDB in batches:

- **BatchGetItem** – Returns the attributes of one or more items from one or more tables. You identify requested items by primary key. A single operation can retrieve up to 16 MB of data, which can contain as many as 100 items.
- **BatchWriteItem** – Puts or deletes multiple items in one or more tables. A single call to BatchWriteItem can write up to 16 MB of data, which can comprise as many as 25 PUT or DELETE requests. Individual items to be written can be as large as 400 KB.

With batch operations, if one request in a batch fails, the entire operation does not fail.

See the AWS Documentation for more information on:

- Batch operations:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html#WorkingWithItems.BatchOperations>
- BatchGetItem:
https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchGetItem.html
- BatchWriteItem:
https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchWriteItem.html

- **TransactWriteItems**
 - Contains a write set
 - Includes one or more PutItem, UpdateItem, and DeleteItem operations across multiple tables
- **TransactGetItems**
 - Contains a read set
 - Includes one or more GetItem operations across multiple tables
- If one operation fails, the entire transaction fails.

You can use the DynamoDB transactional read and write APIs to manage complex business workflows that require adding, updating, or deleting multiple items as a single, all-or-nothing operation. For example, a video game developer can ensure that players' profiles are updated correctly when they exchange items in a game or make in-game purchases.

New DynamoDB operations have been introduced for handling transactions:

- *TransactWriteItems* – A batch operation that contains a write set, with one or more PutItem, UpdateItem, and DeleteItem operations. TransactWriteItems can optionally check for prerequisite conditions that must be satisfied before updates are made. These conditions could involve the same or different items than the items in the write set. If any condition is not met, the transaction is rejected.
- *TransactGetItems* – A batch operation that contains a read set, with one or more GetItem operations. If a TransactGetItems request is issued on an item that is part of an active write transaction, the read transaction is canceled. To get the previously committed value, you can use a standard read.

With transactional operations, if one operation fails, the entire transaction fails.

For more information on transactional operations, see this blog post:

<https://aws.amazon.com/blogs/aws/new-amazon-dynamodb-transactions/>

Recorded demo: Amazon DynamoDB

74



Set up demo Amazon DynamoDB

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

Now, take a moment to watch the [DynamoDB demo](#). The recording runs a little over 2 minutes, and it reinforces many of the concepts that were discussed in this section of the module.

The demonstration shows how to create a table running in Amazon DynamoDB by using the AWS Management Console. It also demonstrates how to interact with the table using the AWS Command Line Interface. The demonstration shows how you can query the table, and add data to the table.

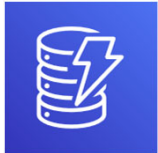
.

Lab: Developing with Amazon DynamoDB using the AWS SDKs





AWS Cloud9



Amazon
DynamoDB

In the AWS Cloud9 environment, you will use the AWS SDK to do the following:

- Task 1: Create a DynamoDB table
- Task 2: Add items to the table
- Task 3: Edit a table item
- Task 4: Query the table for an item
- Task 5: Create a global secondary index



lostcats table



breed_index
global secondary index

In this lab, you will use the AWS SDKs to create a DynamoDB table to store data about missing cats. You will add items to the table and edit them. You will also query the table, as well as create and query a global secondary index to search on the breed attribute.

After completing this lab, you will be able to use the AWS SDKs to:

- Create a DynamoDB table.
- Add data to the table.
- Edit an entry in the table.
- Query the table.
- Create a global secondary index.

Module review

- Introduction to Amazon DynamoDB
- Amazon DynamoDB key concepts
- Partitions and data distribution
- Secondary indexes
- Read/write throughput
- Streams and global tables
- Backup and restore
- Basic operations on Amazon DynamoDB tables
- Experience developing with Amazon DynamoDB using the AWS SDK
- To finish this module, complete the **knowledge check**.

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

53

In this module, we addressed the following topics:

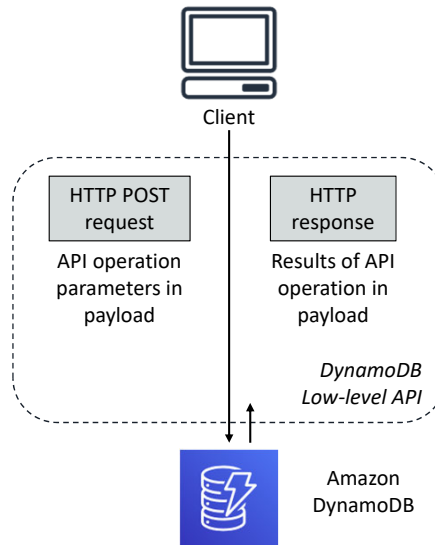
- Introduction to Amazon DynamoDB
- Amazon DynamoDB key concepts
- Partitions and data distribution
- Secondary indexes
- Read/write throughput
- Streams and global tables
- Backup and restore
- Basic operations on Amazon DynamoDB tables

You also gained experience developing with Amazon DynamoDB using the SDK.

To finish this module, please complete the corresponding knowledge check.

Additional resources





The Amazon DynamoDB API is a low-level HTTP-based API. It is the protocol-level interface for Amazon DynamoDB. The API accepts an HTTP(S) POST request as input, processes it, and returns an HTTP response. Every HTTP(S) request must be correctly formatted and carry a valid digital signature. The API uses JavaScript Object Notation (JSON) as a wire protocol format.

To help you use the API, the AWS SDKs construct low-level DynamoDB API requests on your behalf, and they process the responses from DynamoDB. This process lets you focus on your application logic, instead of low-level details. Use an AWS SDK to access the API at a higher level.

To familiarize yourself with the DynamoDB API, see the SDK for your preferred language:

Java

- Index page: <http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/>
- AmazonDynamoDBClient: <http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/dynamodbv2/AmazonDynamoDBClient.html>
- DynamoDB class: <http://docs.aws.amazon.com/AWSJavaSDK/latest/javadoc/com/amazonaws/services/dynamodbv2/document/DynamoDB.html>

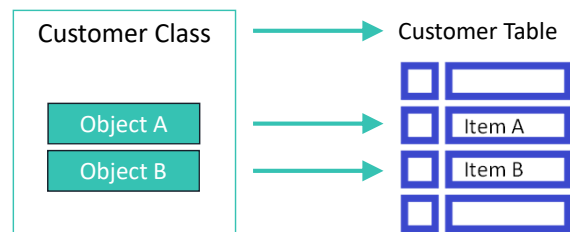
.NET

- Index page: <http://docs.aws.amazon.com/sdkfornet/v3/apidocs/Index.html>
- AmazonDynamoDBClient: <https://docs.aws.amazon.com/sdkfornet/v3/apidocs/items/DynamoDBv2/TDynamoDBClient.html>

Python

- Index page: <http://boto3.readthedocs.org/en/latest/reference/core/index.html>
- DynamoDB resource: <http://boto3.readthedocs.org/en/latest/reference/services/dynamodb.html#service-resource>

- Allows you to persist client-side objects in DynamoDB.
 - Supports the mapping of objects (class instances) to tables.
- Provides higher-level programming interfaces to:
 - Connect to DynamoDB.
 - Perform CRUD operations.
 - Execute queries.
- Available in Java and .NET SDKs



The AWS SDKs provide applications with low-level interfaces for working with Amazon DynamoDB. These client-side classes and methods correspond directly to the low-level DynamoDB API. However, many developers experience a sense of disconnect, or "impedance mismatch", when they need to map complex data types to items in a database table. With a low-level database interface, developers must write methods for reading or writing object data to database tables, and vice-versa. The amount of extra code required for each combination of object type and database table can seem overwhelming.

To simplify development, the AWS SDKs for Java and .NET provide additional interfaces with higher levels of abstraction. The higher-level interfaces for DynamoDB let you define the relationships between objects in your program and the database tables that store those objects' data. After you define this mapping, you call simple object methods such as save, load, or delete, and the underlying low-level DynamoDB operations are automatically invoked on your behalf. This allows you to write object-centric code, rather than database-centric code.

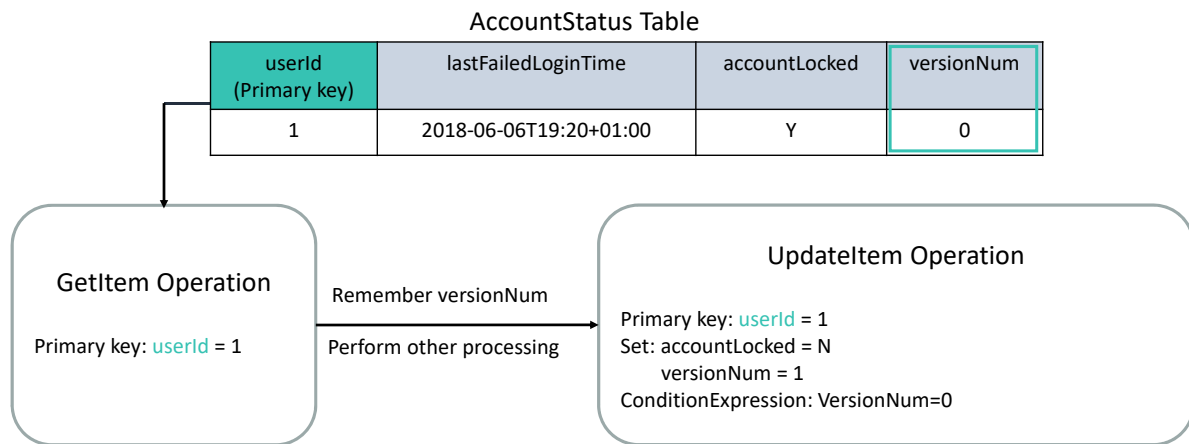
The DynamoDB *object persistence model* enables you to map client-side classes to DynamoDB tables. The instances of these classes (objects) map to items in a DynamoDB table. You can use the object persistence programming interface to connect to DynamoDB, perform CRUD operations, execute queries, and implement optimistic locking with a version number.

Optimistic locking is a strategy to ensure that the client-side item that you are updating (or deleting) is the same as the item in DynamoDB. If you use this strategy, then your database writes are protected from being overwritten by the writes of others — and vice-versa.

Support for the object persistence model is available in the Java and .Net SDKs.

See the AWS Documentation for more information on:

- Higher-level programming interfaces for DynamoDB:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HigherLevelInterfaces.html>
- Optimistic Locking with Version Number (Java):
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBMapper.OptimisticLocking.html>
- Optimistic Locking with Version Number (.NET):
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBContext.VersionSupport.html>



The optimistic locking support in the object persistence model ensures that the item version for your application is the same as the item version on the server-side before updating or deleting the item.

Consider a scenario where you read an item from the AccountStatus table and do some processing. Then, you set the `accountLocked` attribute to N because you determined that the last failed login was over 24 hours earlier. It is possible that the user might have made another failed login attempt between the time that you read the data and when you made the update. The update is incorrect because it is based on stale information.

Optimistic locking solves this problem. Use optimistic locking with a version number as described in the following steps to make sure that an item has not changed since the last time you read it. Maintain a version number to check that the item has not been updated between the last read and update:

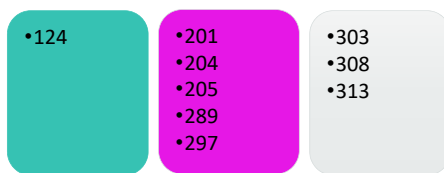
1. Read the item and remember the version number (`versionNum = 0`).
2. Make the state transition in memory after validating information (`accountLocked = N` if `currentLoginTime > lastFailedLoginTime + 24 hours`).
3. Increment the version number (`versionNum = 1`).
4. Write the item with updated attributes (`accountLocked = N` and `versionNum = 1`). Use a conditional expression to perform a write only if the item has not changed since it was last read.
5. If the condition fails, repeat from step 1.

This approach is also known as the *read-modify-write* design pattern, or *optimistic concurrency control*.

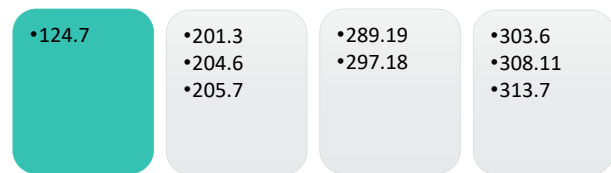
Best practices

This section covers some best practices for working with Amazon DynamoDB.

- Distribute read and write operations evenly across partition keys to maximize throughput.
 - Read or write operations are limited to a small number of partitions. The range of partition keys is limited.
 - Read or write operations are distributed over more partitions. Partition keys contain a calculated suffix.



Hot Partition



The partition key determines the distribution of data across the partitions where data is stored. The total throughput that is provisioned for a table is divided equally across partitions.

When you make a large number of consecutive reads or consecutive writes to a narrow range of partition keys, the same partitions are accessed repeatedly (hot partitions). The throughput allocated to remaining partitions remains unused.

To achieve maximum read/write throughput, implement your read/write operations as follows:

- Choose the partition key carefully to avoid hot spots.
- Consider concatenating a random number or a calculated value to the partition key when data is written so that you can ensure the distribution of partition keys. For example, you might concatenate the sum of the ASCII values of each character in the partition key.
- Distribute reads and writes across multiple partitions.

- Choose a key that will provide uniform workloads.
- Take advantage of sparse indexes.
- Create a global secondary index with a subset of table's attributes for quick lookups.
- Use as an eventually consistent read replica.

The importance of uniform workloads in the context of DynamoDB tables was discussed. The same considerations apply to global secondary indexes. Choose a partition key and sort key that will distribute reads across multiple partitions.

DynamoDB will only write a corresponding entry to a global secondary index if the index key value is present in the item. Therefore, the index might be sparse because it does not contain all items that are in the parent table. You can use these sparse indexes for efficient queries.

You can create a global secondary index that has the same key schema as the table, but with a subset of the table's attributes. You can then provision minimal throughput for this index, and use it for lookups of data.

Consider the following scenarios:

- You want to restrict read access to a table that contains sensitive data.
- You have a high priority application that should read strongly consistent data with high throughput. You have a low-priority application that does not need the latest data.

To handle these scenarios, you can create a global secondary index that has lower provisioned throughput and that also has a subset of the table's attributes. The high priority application can perform strongly consistent reads with high throughput from the table. The low-priority application can perform eventually consistent reads from the global secondary index with lower throughput. This approach ensures that the required throughput is always available for the high-priority application. The global secondary index functions as an eventually consistent read replica.

For more information, see

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-indexes->

[general.html.](#)

- Use indexes sparingly.
- Choose projections carefully.
 - Project only those attributes that you request frequently.
- Take advantage of **sparse indexes**.

Customer Table

CustomerId	Customer Name	SportsNewsInterest
1	John	
2	Mary	Y
3	Tom	Y

SportsInterest Index

CustomerId	SportsNewsInterest
2	Y
3	Y

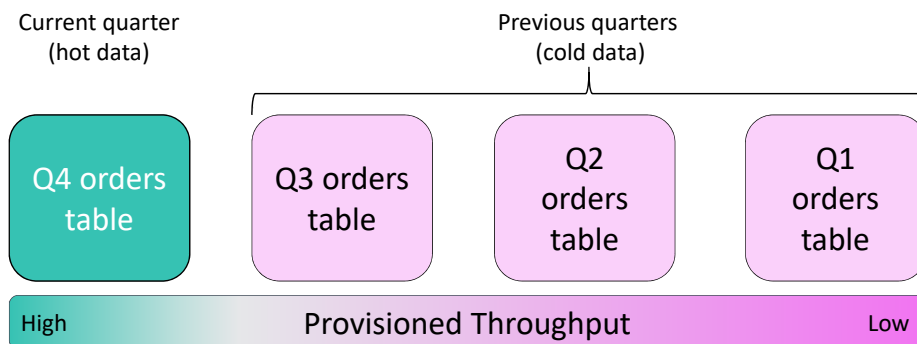
Local secondary indexes consume storage and the provisioned throughput for the table. Keep the size of the index as small as possible.

Create local secondary indexes only on attributes that you query often.

For any item in a table, DynamoDB will only write a corresponding entry in the index when the sort key attribute for the index is present in the item. For example, consider the Customer table (the partition key is CustomerId) and SportsInterest index (the partition key is CustomerId, and the sort key is SportsNewsInterest). DynamoDB will add an entry to the SportsNewsInterest index only if the original item contains a value for the SportsNewsInterest attribute. Therefore, the index is said to be sparse because it does not contain an entry for every item in the table. You can use the sparse index to easily retrieve information about customers who are interested in sports news.

For more information about local secondary indexes, see the AWS Documentation: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-indexes-general.html>.

- Separate data that is accessed frequently (hot) from data that is not accessed frequently (cold).



Consider access patterns for your data. For example, you might have an Orders table with a partition key of **customer id** and sort key of **timestamp**. Your application probably accesses the latest order data most of the time. It might rarely access data about old orders.


For situations like this, consider breaking time series data into separate tables. Store the frequently accessed *hot* data in a separate table with higher throughput. Store rarely accessed *cold* data in tables with lower throughput.

You can even move the old data to other storage options, such as an Amazon S3 bucket, and delete the table that contains the old data.

- Use one-to-many tables instead of a large number of attributes.


Complex Forum Thread Table

```
{
  "thread_id" : 123,
  "subject" : "How do I cook potatoes?",
  "replies" :
    ["Boil...", "Bake...", "Roast...", "Fry...", "Mash..."]
}
```




Simple Forum Thread Table

```
{
  "thread_id" : 123,
  "subject" : "How do I cook potatoes?"
}
```



ReplyTable

```
{
  "thread_id" : 123,
  "reply_id" : "abc"
  "reply" : "Boil for 10 minutes"
}
{
  "thread_id" : 123,
  "reply_id" : "def"
  "reply" : "Bake for 45 minutes"
}
```



If your table has items that store a large number of values in an attribute that is of a *set* type, such as string set or number set, consider removing the set attribute from the table and splitting it into separate items in another table.

For example, consider a table that stores threads in a forum. If the table stores replies for each thread as a string set in each thread item, an individual thread item can become large. The item size is likely to exceed the maximum item size in Dynamo DB. Throughput will be reduced because you will unnecessarily fetch large amounts of data, even if you only need minimum information such as the thread subject.

Instead, you can remove the replies from the forum thread table. Create a separate table to store the replies as individual items.

- Store frequently accessed small attributes in a separate table.

Company Table



```
{
  "companyName" : "Example",
  "stockPrice": 285,
  "aboutCompany": "Example company builds ships",
  "missionStatement": "Our mission is to build the best ships.",
  "logoHighResolution": "example.png"
}
```

Company Stock Table



```
{
  "companyName" : "Example",
  "stockPrice": 285
}
```

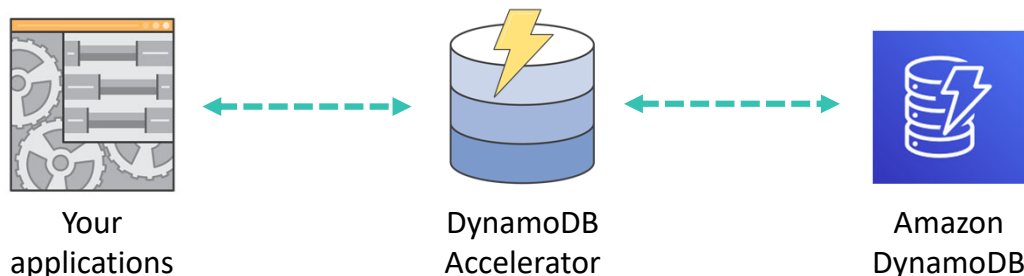
Company Table



```
{
  "companyName " : "Example",
  "aboutCompany": "Example company builds ships",
  "missionStatement": "Our mission is to build the best ships.",
  "logoHighResolution": "example.png"
}
```

If you frequently access large items in a table but do not use the large attribute values, consider storing frequently accessed smaller attributes in a separate table.

For example, consider the Company table. It has fairly large attributes, such as company information, mission statement, and logo. These attributes are fairly static and rarely accessed. To improve throughput, consider splitting the frequently changing and frequently read stock price attribute to a separate table.



You can add in-memory acceleration to DynamoDB tables. Amazon DynamoDB is designed for scale and performance. In most cases, the DynamoDB response times can be measured in single-digit milliseconds. However, certain use cases require response times in microseconds. For these use cases, *DynamoDB Accelerator (DAX)* delivers fast response times for accessing eventually consistent data. DAX is a caching service that is compatible with DynamoDB, and it enables you to benefit from fast in-memory performance for demanding applications.

DAX addresses three core scenarios:

1. As an in-memory cache, DAX reduces the response times of eventually consistent read workloads by an order of magnitude, from single-digit milliseconds to microseconds.
2. DAX reduces operational and application complexity by providing a managed service that is API-compatible with Amazon DynamoDB, and thus requires only minimal functional changes to use with an existing application.
3. For read-heavy or bursty workloads, DAX provides increased throughput and potential operational cost savings by reducing the need to overprovision read capacity units. This is especially beneficial for applications that require repeated reads for individual keys.

DAX is a read-through/write-through cache. When a read is issued to DAX, it first checks to see whether that item is in cache. If it is, DAX returns the value with response times in microseconds. If the item is not in cache, DAX automatically fetches the item from DynamoDB, caches the result for subsequent reads, and returns the value to the application. For writes, DAX first writes the value to DynamoDB, caches the value in DAX, and then returns success to the application.

Troubleshooting

- Check the AWS error code returned from operations.
- Log DynamoDB API calls by using AWS CloudTrail.
 - Will only show you the CreateTable and UpdateTable.
- Set up Amazon CloudWatch alarms and monitor DynamoDB performance metrics.



See the AWS Documentation for more information about:

- DynamoDB error handling:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Programming.Errors.html#APIError>
- Monitoring DynamoDB with CloudWatch:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/MonitoringDynamoDB.html>
- Logging DynamoDB API operations using AWS CloudTrail:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/logging-using-cloudtrail.html>

- 400 error codes must be resolved by the user.



- For some 400 errors, fix the issue before re-submitting the request.
 - There was a problem with the request.
 - Some required parameters were missing.



- For others, retry until the request succeeds.
 - The provisioned throughput was exceeded.

Handle error codes in exceptions gracefully to ensure a smooth customer experience.

For more information about how to handle DynamoDB errors, see the AWS Documentation:
<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Programming.Errors.html>



- 500 and 503 error codes must be resolved by Amazon.
- You can retry until the request succeeds.
 - An internal server error occurred.
 - The service was unavailable.

5xx-series error codes must be resolved by Amazon.

- Use **batch operations** to work with multiple items in a table:
 - BatchGetItem
 - BatchWriteItem
- Use an exponential backoff algorithm and retry failed tables and items in a batch operation.
- Use information in following parameters:
 - **UnprocessedKeys** – Contains information about individual requests that failed in a BatchGetItem operation.
 - **UnprocessedItems** – Contains information about individual requests that failed in a BatchWriteItem operation.

© 2019 Amazon Web Services, Inc. or its Affiliates. All rights reserved.

95

Batch operations can read or write items from one or more tables. Individual requests in a batch operation might fail. The most likely reason for failure is that the table in question does not have enough provisioned read or write capacity.

Use the information about failed tables and items to retry the batch operation with exponential backoff.

See the AWS Documentation for more information about:

- BatchGetItem operation:
https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchGetItem.html
- BatchWriteItem operation:
https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_BatchWriteItem.html

- `ProvisionedThroughputExceededException`
 - Verify throughput calculation, and provision adequate throughput.
 - Consider creating or refining indexes.
 - Consider rewriting queries.
 - Set up Amazon CloudWatch alarms to notify you when request rates exceed a certain threshold of provisioned throughput.

As discussed in previous sections, ensure that you have provisioned adequate throughput for the application's workload. Follow the best practices discussed to create appropriate indexes and write efficient queries.

You can set up CloudWatch alarms to notify you when request rates for a table exceed a certain threshold of the provisioned throughput. For example, you might set up a CloudWatch alarm to notify you when request rates exceed a level that is 80 percent of the table's provisioned throughput. If your application has a sudden surge in requests, you can respond to the CloudWatch alarm by provisioning additional throughput. You can also set up an AWS Lambda function that will be triggered based on a CloudWatch alarm. The Lambda function can update the table to provision additional throughput.

For more information about the metrics that are available, see the AWS Documentation: <https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/aws-services-cloudwatch-metrics.html>