Pricing example:

1 million reads & Writes per day – storing 3GB of data.

Write throughput - $0.0065 per hour for every 10 units

Read throughput - $0.0065 per hour for every 50 units

**Cost per day for read and write ??**

1,000,000/24/60/60 = 11.6 reads/second and writes/second

Write - 0.0065/10 X 12 X 24 = $0.1872 / day

Read – 0.0065/50 \* 12 \* 24 = $0.0374 / day

**From:** Vernekar, Shantaram [CCC-OT NE]   
**Sent:** Thursday, August 2, 2018 5:17 PM  
**To:** Shantaram Vernekar ([shantaram\_vernekar@persistent.com](mailto:shantaram_vernekar@persistent.com)) <[shantaram\_vernekar@persistent.com](mailto:shantaram_vernekar@persistent.com)>; Vernekar, Shantaram [CCC-OT NE] <[sv16217@imcnam.ssmb.com](mailto:sv16217@imcnam.ssmb.com)>  
**Subject:** RE: AWS - Dynamo DB

Amazon DynamoDB is a nonrelational database that delivers reliable performance at any scale. It's a fully managed, multi-region, multi-master database

* provides consistent  single-digit millisecond latency, and offers
* built-in security,
* backup and restore
* in-memory caching.

AWS customers have chosen DynamoDB for

* mobile,
* web – Serverless web applications
* gaming,
* ad tech,
* IoT,

DynamoDB is a serverless database that automatically scales throughput up or down continuously backs up your data for protection.

DynamoDB gives your globally distributed applications fast access to local data by replicating tables across multiple AWS Regions

DynamoDB allows you to delete expired items from tables automatically to help you reduce storage usage and the cost of storing data that is no longer relevant. For more information, see [Time To Live](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/TTL.html).

data is stored on solid state disks (SSDs) and automatically replicated across multiple Availability Zones in an AWS region, providing built-in high availability and data durability

use global tables to keep DynamoDB tables in sync across AWS Regions

Collection

Item

Arrributes

**Primary Key**

When you create a table, in addition to the table name, you must specify the primary key of the table. The primary key uniquely identifies each item in the table, so that no two items can have the same key.

DynamoDB supports two different kinds of primary keys:

**Partition key**

A simple primary key, composed of one attribute known as the partition key.

DynamoDB uses the partition key's value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored.

In a table that has only a partition key, no two items can have the same partition key value.

**Partition key and sort key** – Referred to as a *composite primary key*, this type of key is composed of two attributes. The first attribute is the *partition key*, and the second attribute is the *sort key*.

DynamoDB uses the partition key value as input to an internal hash function. The output from the hash function determines the partition (physical storage internal to DynamoDB) in which the item will be stored.

All items with the same partition key are stored together, in sorted order by sort key value.

it's possible for two items to have the same partition key value. However, those two items must have different sort key values.

The partition key of an item is also known as its *hash attribute*. The term *hash attribute* derives from the use of an internal hash function in DynamoDB that evenly distributes data items across partitions, based on their partition key values.

The sort key of an item is also known as its *range attribute*. The term *range attribute* derives from the way DynamoDB stores items with the same partition key physically close together, in sorted order by the sort key value.

## Secondary Indexes

## You can create one or more secondary indexes on a table. A *secondary index* lets you query the data in the table using an alternate key, in addition to queries against the primary key

DynamoDB supports two kinds of indexes:

* Global secondary index – An index with a partition key and sort key that can be different from those on the table.
* Local secondary index – An index that has the same partition key as the table, but a different sort key.

## You can define up to 5 global secondary indexes and 5 local secondary indexes per table.

## DynamoDB Streams

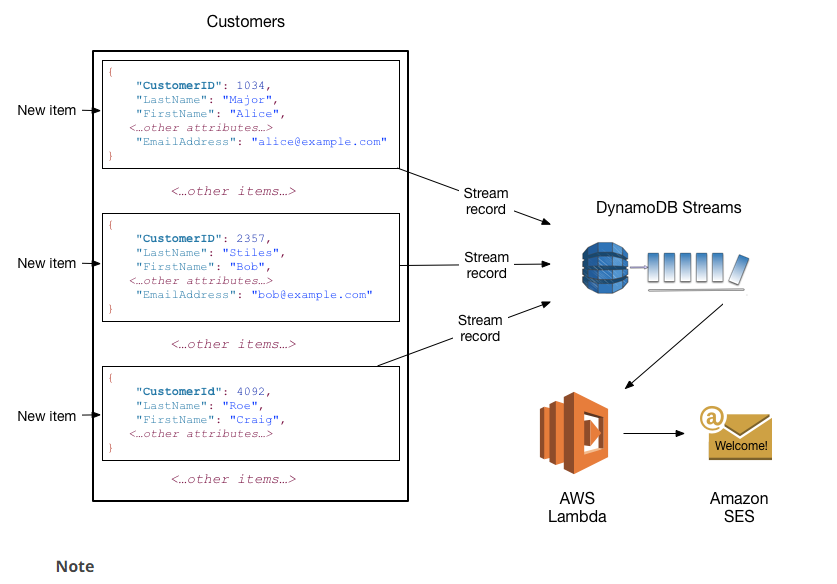
DynamoDB Streams is an optional feature that captures data modification events in DynamoDB tables. The data about these events appear in the stream in near real time, and in the order that the events occurred.

Each event is represented by a ***stream record*.** If you enable a stream on a table, DynamoDB Streams writes a stream record whenever one of the following events occurs:

* A new item is added to the table: The stream captures an image of the entire item, including all of its attributes.
* An item is updated: The stream captures the "before" and "after" image of any attributes that were modified in the item.
* An item is deleted from the table: The stream captures an image of the entire item before it was deleted.

Each stream record also contains the name of the table, the event timestamp, and other metadata. Stream records have a lifetime **of 24 hours**; after that, they are automatically removed from the stream.

You can use DynamoDB Streams together with AWS Lambda to create a trigger—code that executes automatically whenever an event of interest appears in a stream



# Read Consistency

Each region is independent and isolated from other AWS regions. For example, if you have a table called People in the us-east-2 region and another table named People in the us-west-2 region these are considered two entirely separate tables

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, reflecting 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.

**Note**

DynamoDB uses eventually consistent reads, unless you specify otherwise.

# Throughput Capacity for Reads and Writes

When you create a table or index in Amazon DynamoDB, you must specify your capacity requirements for read and write activity. By defining your throughput capacity in advance, DynamoDB can reserve the necessary resources to meet the read and write activity your application requires, while ensuring consistent, low-latency performance.

You specify throughput capacity in terms of read capacity units and write capacity units:

* One *read capacity unit* represents one strongly consistent read per second, or two eventually consistent reads per second, for an item up to 4 KB in size. If you need to read an item that is larger than 4 KB, DynamoDB will need to consume additional read capacity units.

The total number of read capacity units required depends on the item size, and whether you want an eventually consistent or strongly consistent read.

* One *write capacity unit* represents one write per second for an item up to 1 KB in size. If you need to write an item that is larger than 1 KB, DynamoDB will need to consume additional write capacity units.

The total number of write capacity units required depends on the item size.

Throughput calculation

For example, suppose that you create a table with 5 read capacity units and 5 write capacity units. With these settings, your application could:

* Perform strongly consistent reads of up to 20 KB per second (4 KB × 5 read capacity units). = 5 \* 4
* Perform eventually consistent reads of up to 40 KB per second (twice as much read throughput). = 5\*2\*4
* Write up to 5 KB per second (1 KB × 5 write capacity units). = 5\*1

up to the DynamoDB maximum item size of **400 KB**

**What if read/write request is more than configured capacity ?**

Requests are throttled and request fails

SDK provides retry option

If your read or write requests exceed the throughput settings for a table, DynamoDB can throttle that request. DynamoDB can also throttle read requests exceeds for an index. Throttling prevents your application from consuming too many capacity units. When a request is throttled, it fails with an HTTP 400 code (Bad Request) and aProvisionedThroughputExceededException. The AWS SDKs have built-in support for retrying throttled requests

## DynamoDB Auto Scaling – to prevent request from being denied..

DynamoDB auto scaling actively manages throughput capacity for tables and global secondary indexes. With auto scaling, you define a range (upper and lower limits) for read and write capacity units. You also define a target utilization percentage within that range.

DynamoDB auto scaling seeks to maintain your target utilization, even as your application workload increases or decreases.

With DynamoDB auto scaling, a table or a global secondary index can increase its provisioned read and write capacity to handle sudden increases in traffic, without request throttling. When the workload decreases, DynamoDB auto scaling can decrease the throughput so that you don't pay for unused provisioned capacity.

**Note**

If you use the AWS Management Console to create a table or a global secondary index, DynamoDB auto scaling is enabled by default.

## Provisioned Throughput

If you aren't using DynamoDB auto scaling, you have to manually define your throughput requirements. *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 settings, it is subject to request throttling.

For example, suppose that you want to read 80 items per second from a table. The items are 3 KB in size, and you want strongly consistent reads. For this scenario, each read requires one provisioned read capacity unit. To determine this, you divide the item size of the operation by 4 KB, and then round up to the nearest whole number, as in this example:

* 3 KB / 4 KB = 0.75, or **1** read capacity unit

For this scenario, you have to set the table's provisioned read throughput to 80 read capacity units:

* 1 read capacity unit per item × 80 reads per second = **80** read capacity units

Now suppose that you want to write 100 items per second to your table, and that the items are 512 bytes in size. For this scenario, each write requires one provisioned write capacity unit. To determine this, you divide the item size of the operation by 1 KB, and then round up to the nearest whole number:

* 512 bytes / 1 KB = 0.5, or **1**

For this scenario, you would want to set the table's provisioned write throughput to 100 write capacity units:

* 1 write capacity unit per item × 100 writes per second = **100** write capacity units

## Reserved Capacity

As a DynamoDB customer, you can purchase reserved capacity in advance, as described at [Amazon DynamoDB Pricing](https://aws.amazon.com/dynamodb/pricing).

With reserved capacity, you pay a one-time upfront fee and commit to a minimum usage level over a period of time. By reserving your read and write capacity units ahead of time, you realize significant cost savings compared to on-demand provisioned throughput settings.

# The DynamoDB API

Control Plane – APIS at table level – Create/Delete/Update/List/Describe table

Data Plane – APIS at data level – Put/get/Batch get/write – efficient way.. single network roundtrip

DynamoDB Steams –

# Partitions and Data Distribution

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.

When you create a table, the initial status of the table is CREATING. During this phase, DynamoDB allocates sufficient partitions to the table so that it can handle your provisioned throughput requirements. You can begin writing and reading table data after the table status changes to ACTIVE.

DynamoDB allocates additional partitions to a table in the following situations:

* If you increase the table's provisioned throughput settings beyond what the existing partitions can support.
* If an existing partition fills to capacity and more storage space is required.

## Data Distribution: Partition Key

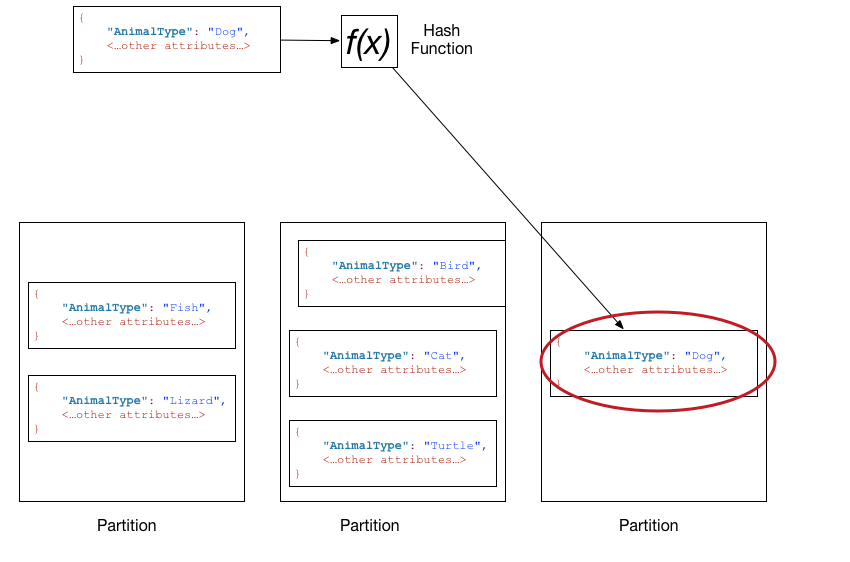
To write an item to the table, DynamoDB uses the value of the partition key as input to an internal hash function. The output value from the hash function determines the partition in which the item will be stored.

To read an item from the table, you must specify the partition key value for the item. DynamoDB uses this value as input to its hash function, yielding the partition in which the item can be found.

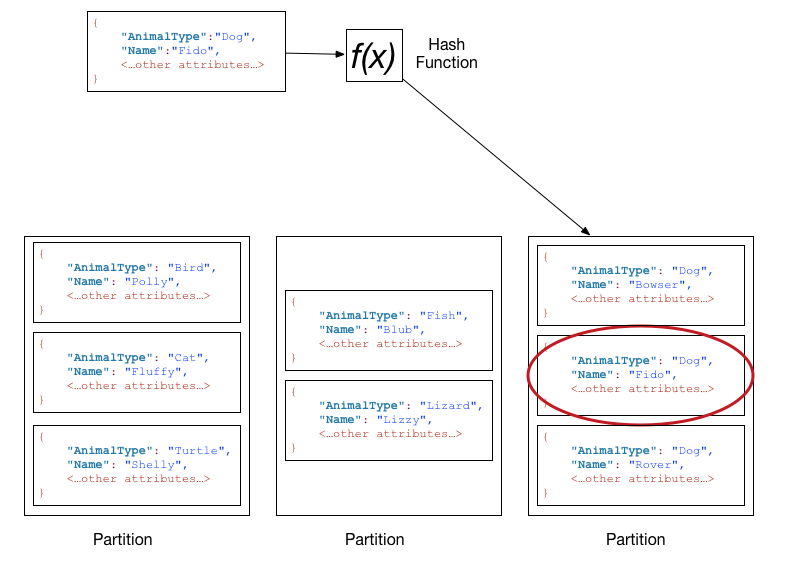
## DynamoDB is optimized for uniform distribution of items across a table's partitions

To write an item to the table, DynamoDB calculates the hash value of the partition key to determine which partition should contain the item. In that partition, there could be several items with the same partition key value, so DynamoDB stores the item among the others with the same partition key, in ascending order by sort key.

To read an item from the table, you must specify its partition key value and sort key value. DynamoDB calculates the partition key's hash value, yielding the partition in which the item can be found.



## Data Distribution: Partition Key and Sort Key



# On-Demand Backup and Restore for DynamoDB

Backup and restore actions execute with zero impact on table performance or availability.

Currently, the backup and restore functionality works in the same Region as the source table

On-demand backup and restore scales without degrading the performance or availability of your applications. It uses a new and unique distributed technology that allows you to complete backups in seconds regardless of table size. You can create backups that are consistent within seconds across thousands of partitions without worrying about schedules or long-running backup processes.

# Point-in-Time Recovery for DynamoDB

You can enable point-in-time recovery as well as create on-demand backups for your Amazon DynamoDB tables

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.

# Global Tables

Amazon DynamoDB 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 propagate ongoing data changes to all of them.

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.

DynamoDB global tables are ideal for massively scaled applications, with globally dispersed users. In such an environment, users expect very fast application performance.

Global tables provide automatic multi-master replication to AWS regions world-wide, so you can deliver low-latency data access to your users no matter where they are located.

DAX

Amazon DynamoDB is designed for scale and performance. In most cases, the DynamoDB response times can be measured in single-digit milliseconds. However, there are certain use cases that require response times in microseconds. For these use cases, *DynamoDB Accelerator (DAX)* delivers fast response times for accessing eventually consistent data.

DAX is a DynamoDB-compatible caching service that 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 over-provision read capacity units. This is especially beneficial for applications that require repeated reads for individual keys.

## Use Cases for DAX

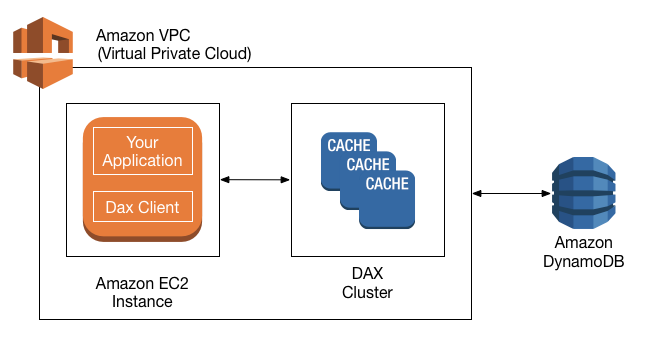
DAX provides access to eventually consistent data from DynamoDB tables, with microsecond latency. A multi-AZ DAX cluster can serve millions of requests per second.

DAX is ideal for:

* Applications that require the fastest possible response time for reads. Some examples include real-time bidding, social gaming, and trading applications. DAX delivers fast, in-memory read performance for these use cases.
* Applications that read a small number of items more frequently than others. For example, consider an e-commerce system that has a one-day sale on a popular product. During the sale, demand for that product (and its data in DynamoDB) would sharply increase, compared to all of the other products. To mitigate the impacts of a "hot" key and a non-uniform data distribution, you could offload the read activity to a DAX cache until the one-day sale is over.
* Applications that are read-intensive, but are also cost-sensitive. With DynamoDB, you provision the number of reads per second that your application requires. If read activity increases, you can increase your tables' provisioned read throughput (at an additional cost). Alternatively, you can offload the activity from your application to a DAX cluster, and reduce the amount of read capacity units you'd need to purchase otherwise.
* Applications that require repeated reads against a large set of data. Such an application could potentially divert database resources from other applications. For example, a long-running analysis of regional weather data could temporarily consume all of the read capacity in a DynamoDB table, which would negatively impact other applications that need to access the same data. With DAX, the weather analysis could be performed against cached data instead.

DAX is designed to run within an Amazon Virtual Private Cloud environment (Amazon VPC).

To run your application, you launch an Amazon EC2 instance into your Amazon VPC, and then deploy your application (with the DAX client) on the EC2 instance. At runtime, the DAX client directs all of your application's DynamoDB API requests to the DAX cluster. If DAX can process one of these API requests directly, it does so; otherwise, it passes the request through to DynamoDB. Finally, the DAX cluster returns the results to your application.



## How DAX Processes Requests

A DAX cluster consists of one or more nodes. Each node runs its own instance of the DAX caching software. One of the nodes serves as the primary node for the cluster. Additional nodes (if present) serve as read replicas

Your application can access DAX by specifying the endpoint for the DAX cluster. The DAX client software works with the cluster endpoint to perform intelligent load-balancing and routing, so that incoming requests are evenly distributed across all of the nodes in the cluster.

### Read Operations

If the request specifies *eventually consistent reads* (the default behavior), it attempts to read the item from DAX:

* If DAX has the item available (a *cache hit*), DAX returns the item to the application without accessing DynamoDB.
* If DAX does not have the item available (a *cache miss*), DAX passes the request through to DynamoDB. When it receives the response from DynamoDB, DAX returns the results to the application—but it also writes the results to the cache on the primary node.

If there are any read replicas in the cluster, DAX automatically keeps the replicas in sync with the primary node

If the request specifies strongly consistent reads, DAX passes the request through to DynamoDB. The results from DynamoDB are not cached in DAX; instead, they are simply returned to the application.

### Write Operations

With these operations, data is first written to the DynamoDB table, and then to the DAX cluster. The operation is successful only if the data is successfully written to boththe table and to DAX.

### Other Operations

DAX does not recognize any DynamoDB operations for managing tables (such as CreateTable, UpdateTable, and so on). If your application needs to perform these operations, it will need to access DynamoDB directly rather than using DAX.

<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DAX.access-control.html>