# **Amazon DynamoDB**

# Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictablperformance with seamless scalability.

* DynamoDB lets you offload the administrative burdens of operating and scaling a distributed database so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling.
* DynamoDB also offers encryption at rest, which eliminates the operational burden and complexity involved in protecting sensitive data.
* With DynamoDB, you can create database tables that can store and retrieve any amount of data and serve any level of request traffic.
* You can scale up or scale down your tables' throughput capacity without downtime or performance degradation. You can use the AWS Management Console to monitor resource utilization and performance metrics.
* DynamoDB provides on-demand backup capability. It allows you to create full backups of your tables for long-term retention and archival for regulatory compliance needs.
* You can create on-demand backups and enable point-in-time recovery for your Amazon DynamoDB tables. Point-in-time recovery helps protect your tables from accidental write or delete operations. With point-in-time recovery, you can restore that table to any point in time during the last 35 days.
* 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.

**High Availability and Durability**

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 is automatically replicated across multiple Availability Zones in an AWS Region, providing built-in high availability and data durability. You can use global tables to keep DynamoDB tables in sync across AWS Regions. For more information, see [Global Tables](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GlobalTables.html).

# **Amazon DynamoDB: How It Works**

# Read Consistency

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. Strongly consistent reads are not supported on global secondary indexes.

**Note**

DynamoDB uses eventually consistent reads, unless you specify otherwise. Read operations (such as GetItem, Query, and Scan) provide a ConsistentRead parameter. If you set this parameter to true, DynamoDB uses strongly consistent reads during the operation.

# Read/Write Capacity Mode

Amazon DynamoDB has two read/write capacity modes for processing reads and writes on your tables:

* On-demand
* Provisioned (default, free-tier eligible)

Global secondary indexes inherit the read/write capacity mode from the base table. For more information, see [Considerations When Changing Read/Write Capacity Mode](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/switching.capacitymode.html).

## On-Demand Mode

Amazon DynamoDB on-demand is a flexible billing option capable of serving thousands of requests per second without capacity planning. DynamoDB on-demand offers pay-per-request pricing for read and write requests so that you pay only for what you use.

When you choose on-demand mode, DynamoDB instantly accommodates your workloads as they ramp up or down to any previously reached traffic level. If a workload’s traffic level hits 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 APIs without changing code.

On-demand mode is a good option if any of the following 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.

The request rate is only limited by the DynamoDB throughput default table limits, but it can be raised upon request.

To get started with on-demand, you can create or update a table to use on-demand mode.

You can switch between read/write capacity modes once every 24 hours.

**Note**

On-demand is currently not supported by AWS Data Pipeline, the DynamoDB import/export tool, and AWS Glue.

## Provisioned Mode

If you choose provisioned mode, you specify the number of reads and writes per second that you require for your application. You can use auto scaling to adjust your table’s provisioned capacity automatically in response to traffic changes. This helps you govern your DynamoDB use to stay at or below a defined request rate in order to obtain cost predictability.

Provisioned mode is a good option if any of the following are true:

* You have predictable application traffic.
* You run applications whose traffic is consistent or ramps gradually.
* You can forecast capacity requirements to control costs.

### DynamoDB Auto Scaling

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.

### 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.

# Partitions and Data Distribution

Amazon 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—you never have to manage partitions yourself.

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

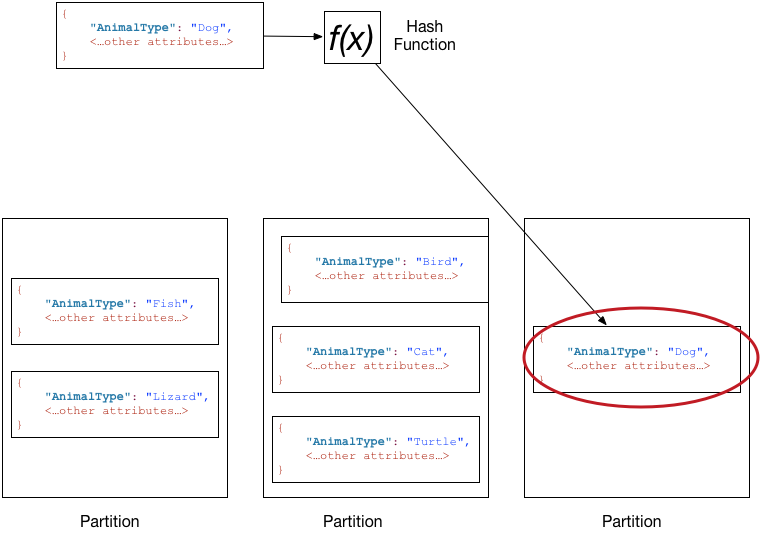
* If you increase the table's provisioned throughput settings beyond what the existing partitions can support.
* If an existing partition fills to capacity and more storage space is required.

Global secondary indexes in DynamoDB are also composed of partitions. The data in a global secondary index is stored separately from the data in its base table, but index partitions behave in much the same way as table partitions.

**Data Distribution: Partition Key**

If your table has a simple primary key (partition key only), DynamoDB stores and retrieves each item based on its partition key value.

.The following diagram shows a table named *Pets*, which spans multiple partitions. The table's primary key is *AnimalType* (only this key attribute is shown). DynamoDB uses its hash function to determine where to store a new item, in this case based on the hash value of the string *Dog*. Note that the items are not stored in sorted order. Each item's location is determined by the hash value of its partition key.



**Note**

DynamoDB is optimized for uniform distribution of items across a table's partitions, no matter how many partitions there may be. We recommend that you choose a partition key that can have a large number of distinct values relative to the number of items in the table.

## Data Distribution: Partition Key and Sort Key

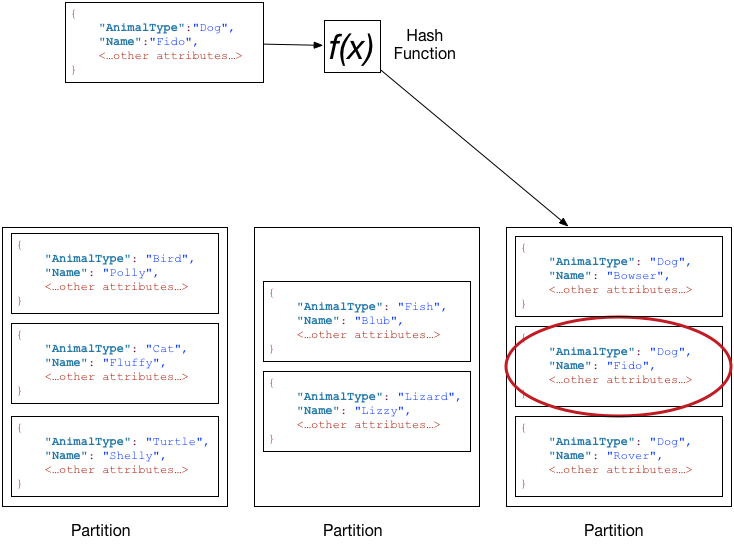
If the table has a composite primary key (partition key and sort key), DynamoDB calculates the hash value of the partition key in the same way as described in [Data Distribution: Partition Key](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.Partitions.html#HowItWorks.Partitions.SimpleKey). However, it stores all the items with the same partition key value physically close together, ordered by sort key value.

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, several items could have 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.

You can read multiple items from the table in a single operation (Query) if the items you want have the same partition key value. DynamoDB returns all of the items with that partition key value. Optionally, you can apply a condition to the sort key so that it returns only the items within a certain range of values.

Suppose that the Pets table has a composite primary key consisting of AnimalType (partition key) and Name (sort key). The following diagram shows DynamoDB writing an item with a partition key value of Dog and a sort key value of Fido.



To read that same item from the Pets table, DynamoDB calculates the hash value of Dog, yielding the partition in which these items are stored. DynamoDB then scans the sort key attribute values until it finds Fido.

To read all of the items with an AnimalType of Dog, you can issue a Query operation without specifying a sort key condition. By default, the items are returned in the order that they are stored (that is, in ascending order by sort key). Optionally, you can request descending order instead.

To query only some of the Dog items, you can apply a condition to the sort key (for example, only the Dog items where Name begins with a letter that is within the range A through K).

**Note**

In a DynamoDB table, there is no upper limit on the number of distinct sort key values per partition key value. If you needed to store many billions of Dog items in the Petstable, DynamoDB would allocate enough storage to handle this requirement automatically.

# **Working with DynamoDB**

# Working with tables in DynamoDB

# Managing Throughput Settings on Provisioned Tables

When you create a new provisioned table in DynamoDB, you must specify its *provisioned throughput capacity*—the amount of read and write activity that the table will be able to support. DynamoDB uses this information to reserve sufficient system resources to meet your throughput requirements.

You can optionally allow DynamoDB auto scaling to manage your table's throughput capacity. However, you still must provide initial settings for read and write capacity when you create the table. DynamoDB auto scaling uses these initial settings as a starting point, and then adjusts them dynamically in response to your application's requirements.

As your application data and access requirements change, you might need to adjust your table's throughput settings. If you're using DynamoDB auto scaling, the throughput settings are automatically adjusted in response to actual workloads. You can also use the UpdateTableoperation to manually adjust your table's throughput capacity. You might decide to do this if you need to bulk-load data from an existing data store into your new DynamoDB table. You could create the table with a large write throughput setting and then reduce this setting after the bulk data load is complete.

You specify throughput requirements in terms of *capacity units*—the amount of data your application needs to read or write per second. You can modify these settings later, if needed, or enable DynamoDB auto scaling to modify them automatically.

## Read Capacity Units

A 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.

For example, suppose that you create a table with 10 provisioned read capacity units. This allows you to perform 10 strongly consistent reads per second, or 20 eventually consistent reads per second, for items up to 4 KB.

Reading an item larger than 4 KB consumes more read capacity units. For example, a strongly consistent read of an item that is 8 KB (4 KB × 2) consumes 2 read capacity units. An eventually consistent read on that same item consumes only 1 read capacity unit.

Item sizes for reads are rounded up to the next 4 KB multiple. For example, reading a 3,500-byte item consumes the same throughput as reading a 4 KB item.

### Capacity Unit Consumption for Reads

The following describes how DynamoDB read operations consume read capacity units:

* GetItem—Reads a single item from a table. To determine the number of capacity units that GetItem will consume, take the item size and round it up to the next 4 KB boundary. If you specified a strongly consistent read, this is the number of capacity units required. For an eventually consistent read (the default), divide this number by two.

For example, if you read an item that is 3.5 KB, DynamoDB rounds the item size to 4 KB. If you read an item of 10 KB, DynamoDB rounds the item size to 12 KB.

* BatchGetItem—Reads up to 100 items, from one or more tables. DynamoDB processes each item in the batch as an individual GetItem request, so DynamoDB first rounds up the size of each item to the next 4 KB boundary, and then calculates the total size. The result is not necessarily the same as the total size of all the items. For example, if BatchGetItem reads a 1.5 KB item and a 6.5 KB item, DynamoDB calculates the size as 12 KB (4 KB + 8 KB), not 8 KB (1.5 KB + 6.5 KB).
* Query—Reads multiple items that have the same partition key value. All items returned are treated as a single read operation, where DynamoDB computes the total size of all items and then rounds up to the next 4 KB boundary. For example, suppose your query returns 10 items whose combined size is 40.8 KB. DynamoDB rounds the item size for the operation to 44 KB. If a query returns 1500 items of 64 bytes each, the cumulative size is 96 KB.
* Scan—Reads all items in a table. DynamoDB considers the size of the items that are evaluated, not the size of the items returned by the scan.

If you perform a read operation on an item that does not exist, DynamoDB still consumes provisioned read throughput: A strongly consistent read request consumes one read capacity unit, while an eventually consistent read request consumes 0.5 of a read capacity unit.

For any operation that returns items, you can request a subset of attributes to retrieve; however, doing so has no impact on the item size calculations. In addition, Query and Scan can return item counts instead of attribute values. Getting the count of items uses the same quantity of read capacity units and is subject to the same item size calculations. This is because DynamoDB has to read each item in order to increment the count.

#### Read Operations and Read Consistency

The preceding calculations assume strongly consistent read requests. For an eventually consistent read request, the operation consumes only half the capacity units. For an eventually consistent read, if the total item size is 80 KB, the operation consumes only 10 capacity units.

## Write Capacity Units

A write capacity unit represents one write per second, for an item up to 1 KB in size.

For example, suppose that you create a table with 10 write capacity units. This allows you to perform 10 writes per second, for items up to 1 KB in size per second.

Item sizes for writes are rounded up to the next 1 KB multiple. For example, writing a 500-byte item consumes the same throughput as writing a 1 KB item.

### Capacity Unit Consumption for Writes

The following describes how DynamoDB write operations consume write capacity units:

* PutItem—Writes a single item to a table. If an item with the same primary key exists in the table, the operation replaces the item. For calculating provisioned throughput consumption, the item size that matters is the larger of the two.
* UpdateItem—Modifies a single item in the table. DynamoDB considers the size of the item as it appears before and after the update. The provisioned throughput consumed reflects the larger of these item sizes. Even if you update just a subset of the item's attributes,UpdateItem will still consume the full amount of provisioned throughput (the larger of the "before" and "after" item sizes).
* DeleteItem—Removes a single item from a table. The provisioned throughput consumption is based on the size of the deleted item.
* BatchWriteItem—Writes up to 25 items to one or more tables. DynamoDB processes each item in the batch as an individual PutItem or DeleteItem request (updates are not supported). So DynamoDB first rounds up the size of each item to the next 1 KB boundary, and then calculates the total size. The result is not necessarily the same as the total size of all the items. For example, if BatchWriteItem writes a 500 byte item and a 3.5 KB item, DynamoDB calculates the size as 5 KB (1 KB + 4 KB), not 4 KB (500 bytes + 3.5 KB).

For PutItem, UpdateItem, and DeleteItem operations, DynamoDB rounds the item size up to the next 1 KB. For example, if you put or delete an item of 1.6 KB, DynamoDB rounds the item size up to 2 KB.

PutItem, UpdateItem, and DeleteItem allow conditional writes, where you specify an expression that must evaluate to true in order for the operation to succeed. If the expression evaluates to false, DynamoDB still consumes write capacity units from the table:

* For an existing item, the number of write capacity units consumed depends on the size of the new item. (For example, a failed conditional write of a 1 KB item would consume one write capacity unit. If the new item were twice that size, the failed conditional write would consume two write capacity units.)
* For a new item, DynamoDB consumes one write capacity unit.

## Request Throttling and Burst Capacity

If your application performs reads or writes at a higher rate than your table can support, DynamoDB begins to throttle those requests. When DynamoDB throttles a read or write, it returns a ProvisionedThroughputExceededException to the caller. The application can then take appropriate action, such as waiting for a short interval before retrying the request.

**Note**

We recommend that you use the AWS SDKs for software development. The AWS SDKs provide built-in support for retrying throttled requests; you do not need to write this logic yourself. For more information, see [Error Retries and Exponential Backoff](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Programming.Errors.html#Programming.Errors.RetryAndBackoff).

The DynamoDB console displays Amazon CloudWatch metrics for your tables, so you can monitor throttled read requests and write requests. If you encounter excessive throttling, you should consider increasing your table's provisioned throughput settings.

In some cases, DynamoDB uses burst capacity to accommodate reads or writes in excess of your table's throughput settings. With burst capacity, unexpected read or write requests can succeed where they otherwise would be throttled.

## Request Throttling and Adaptive Capacity

DynamoDB automatically distributes your data across partitions, which are stored on multiple servers in the AWS Cloud (For more information, see [Partitions and Data Distribution](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.Partitions.html)). It's not always possible to distribute read and write activity evenly all the time. When data access is imbalanced, a "hot" partition can receive such a higher volume of read and write traffic compared to other partitions. Adaptive capacity works by automatically increasing throughput capacity for partitions that receive more traffic.

## Choosing Initial Throughput Settings

Every application has different requirements for reading and writing from a database. When you are determining the initial throughput settings for a DynamoDB table, take the following inputs into consideration:

* **Item sizes.**
* **Expected read and write request rates.** .
* **Read consistency requirements.**

# Managing Throughput Capacity Automatically with DynamoDB Auto Scaling

DynamoDB auto scaling uses the AWS Application Auto Scaling service to dynamically adjust provisioned throughput capacity on your behalf, in response to actual traffic patterns. This enables a table or a global secondary index to increase its provisioned read and write capacity to handle sudden increases in traffic, without throttling. When the workload decreases, Application Auto Scaling decreases 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. You can modify your auto scaling settings at any time.

With Application Auto Scaling, you create a scaling policy for a table or a global secondary index. The scaling policy specifies whether you want to scale read capacity or write capacity (or both), and the minimum and maximum provisioned capacity unit settings for the table or index.

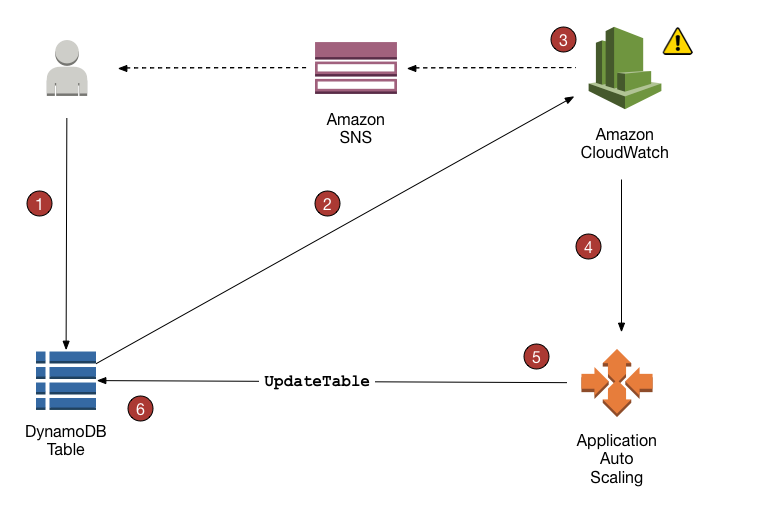
The scaling policy also contains a target utilization—the percentage of consumed provisioned throughput at a point in time. You can set the auto scaling target utilization values between 20% and 90% for your read and write capacity.

**Note**

In addition to tables, DynamoDB auto scaling also supports global secondary indexes. Every global secondary index has its own provisioned throughput capacity, separate from that of its base table.

## How DynamoDB Auto Scaling Works

The following diagram provides a high-level overview of how DynamoDB auto scaling manages throughput capacity for a table:



**Note**

DynamoDB auto scaling modifies provisioned throughput settings only when the actual workload stays elevated (or depressed) for a sustained period of several minutes. The Application Auto Scaling target tracking algorithm seeks to keep the target utilization at or near your chosen value over the long term.Sudden, short-duration spikes of activity are accommodated by the table's built-in burst capacity.

### Usage Notes

Before you begin using DynamoDB auto scaling, you should be aware of the following:

* DynamoDB auto scaling can increase read capacity or write capacity as often as necessary, in accordance with your auto scaling policy. All DynamoDB limits remain in effect, as described in [Limits in DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Limits.html).
* DynamoDB auto scaling doesn't prevent you from manually modifying provisioned throughput settings. These manual adjustments don't affect any existing CloudWatch alarms that are related to DynamoDB auto scaling.
* If you enable DynamoDB auto scaling for a table that has one or more global secondary indexes, we highly recommend that you also apply auto scaling uniformly to those indexes. You can do this by choosing **Apply same settings to global secondary indexes** in the AWS Management Console.

# Working with Items in DynamoDB

In DynamoDB, an item is a collection of attributes. Each attribute has a name and a value. An attribute value can be a scalar, a set, or a document type. DynamoDB provides four operations for basic create/read/update/delete (CRUD) functionality:

* PutItem – create an item.
* GetItem – read an item.
* UpdateItem – update an item.
* DeleteItem – delete an item.

Each of these operations require you to specify the primary key of the item you want to work with. For example, to read an item using GetItem, you must specify the partition key and sort key (if applicable) for that item.

In addition to the four basic CRUD operations, DynamoDB also provides the following:

* BatchGetItem – read up to 100 items from one or more tables.
* BatchWriteItem – create or delete up to 25 items in one or more tables.

These batch operations combine multiple CRUD operations into a single request. In addition, the batch operations read and write items in parallel to minimize response latencies.

# Time To Live

Time To Live (TTL) for DynamoDB allows you to define when items in a table expire so that they can be automatically deleted from the database.

TTL is provided at no extra cost as a way to reduce storage usage and reduce the cost of storing irrelevant data without using provisioned throughput. With TTL enabled on a table, you can set a timestamp for deletion on a per-item basis, allowing you to limit storage usage to only those records that are relevant.

TTL is useful if you have continuously accumulating data that loses relevance after a specific time period. For example: session data, event logs, usage patterns, and other temporary data. If you have sensitive data that must be retained only for a certain amount of time according to contractual or regulatory obligations, TTL helps you ensure that it is removed promptly and as scheduled.

# Working with Queries

The Query operation finds items based on primary key values. You can query any table or secondary index that has a composite primary key (a partition key and a sort key).

You must provide the name of the partition key attribute, and a single value for that attribute. Query will return all of the items with that partition key value. You can optionally provide a sort key attribute, and use a comparison operator to refine the search results.

## Read Consistency for Query

A Query operation performs eventually consistent reads, by default. This means that the Queryresults might not reflect changes due to recently completed PutItem or UpdateItem operations. For more information, see [Read Consistency](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html).

If you require strongly consistent reads, set the ConsistentRead parameter to true in the Queryrequest.

# Working with Scans

A Scan operation reads every item in a table or a secondary index. By default, a Scan operation returns all of the data attributes for every item in the table or index. You can use the ProjectionExpression parameter so that Scan only returns some of the attributes, rather than all of them.

Scan always returns a result set. If no matching items are found, the result set will be empty.

A single Scan request can retrieve a maximum of 1 MB of data; DynamoDB can optionally apply a filter expression to this data, narrowing the results before they are returned to the user.

# Improving Data Access with 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, to allow 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, and issue Query or Scan requests against these indexes.

A secondary index is a data structure that contains a subset of attributes from a table, along with an alternate key to support Query operations. You can retrieve data from the index using a Query, in much the same way as you use Query with a table. A table can have multiple secondary indexes, which gives your applications access to many different query patterns.

**Note**

You can also Scan an index, in much the same way as you would Scan a table.

Every secondary index is associated with exactly one table, from which it obtains its data. This is called the base table for the index. When you create an index, you define an alternate key for the index (partition key and sort key). You also define the attributes that you want to be projected, or copied, from the base table into the index. DynamoDB copies these attributes into the index, along with the primary key attributes from the base table. You can then query or scan the index just as you would query or scan a table.

Every secondary index is automatically maintained by DynamoDB. When you add, modify, or delete items in the base table, any indexes on that table are also updated to reflect these changes.

DynamoDB supports two types of secondary indexes:

* [**Global secondary index**](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.html)**—**an index with a partition key and a sort key that 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.
* [**Local secondary index**](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/LSI.html)**—**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.

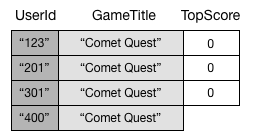
|  |  |  |
| --- | --- | --- |
| **Characteristic** | **Global Secondary Index** | **Local Secondary Index** |
| **Key Schema** | The primary key of a global secondary index can be either simple (partition key) or composite (partition key and sort key). | The primary key of a local secondary index must be composite (partition key and sort key). |
| **Key Attributes** | The index partition key and sort key (if present) can be any base table attributes of type string, number, or binary. | The partition key of the index is the same attribute as the partition key of the base table. The sort key can be any base table attribute of type string, number, or binary. |
| **Size Restrictions Per Partition Key Value** | There are no size restrictions for global secondary indexes. | For each partition key value, the total size of all indexed items must be 10 GB or less. |
| **Online Index Operations** | Global secondary indexes can be created at the same time that you create a table. You can also add a new global secondary index to an existing table, or delete an existing global secondary index. For more information, see [Managing Global Secondary Indexes](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GSI.OnlineOps.html). | Local secondary indexes are created at the same time that you create a table. You cannot add a local secondary index to an existing table, nor can you delete any local secondary indexes that currently exist. |
| **Queries and Partitions** | A global secondary index lets you query over the entire table, across all partitions. | A local secondary index lets you query over a single partition, as specified by the partition key value in the query. |
| **Read Consistency** | Queries on global secondary indexes support eventual consistency only. | When you query a local secondary index, you can choose either eventual consistency or strong consistency. |
| **Provisioned Throughput Consumption** | Every global secondary index has its own provisioned throughput settings for read and write activity. Queries or scans on a global secondary index consume capacity units from the index, not from the base table. The same holds true for global secondary index updates due to table writes. | Queries or scans on a local secondary index consume read capacity units from the base table. When you write to a table, its local secondary indexes are also updated; these updates consume write capacity units from the base table. |
| **Projected Attributes** | With global secondary index queries or scans, you can only request the attributes that are projected into the index. DynamoDB will not fetch any attributes from the table. | If you query or scan a local secondary index, you can request attributes that are not projected in to the index. DynamoDB will automatically fetch those attributes from the table. |

# Global Secondary Indexes

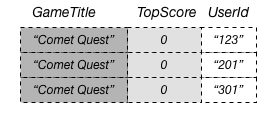
* Every global secondary index must have a partition key, and can have an optional sort key.
* The index key schema can be different from the base table schema; you could have a table with a simple primary key (partition key), and create a global secondary index with a composite primary key (partition key and sort key) — or vice-versa.
* The index key attributes can consist of any top-level String, Number, or Binary attributes from the base table; other scalar types, document types, and set types are not allowed.
* You can project other base table attributes into the index if you want. When you query the index, DynamoDB can retrieve these projected attributes efficiently; however, global secondary index queries cannot fetch attributes from the base table. For example, if you queried *GameTitleIndex*, as shown in the diagram above, the query would not be able to access any non-key attributes other than *TopScore* (though the key attributes *GameTitle* and *UserId* would automatically be projected).
* In a DynamoDB table, each key value must be unique. However, the key values in a global secondary index do not need to be unique.
* A global secondary index only keeps track of data items where its key attribute(s) actually exist. For example, suppose that you added another new item to the GameScores table, but only provided the required primary key attributes:

|  |  |
| --- | --- |
| * **UserId** | **GameTitle** |
| 400 | Comet Quest |

Because you didn't specify the *TopScore* attribute, DynamoDB would not propagate this item to*GameTitleIndex*. Thus, if you queried *GameScores* for all the Comet Quest items, you would get the following four items:



A similar query on *GameTitleIndex* would still return three items, rather than four. This is because the item with the nonexistent *TopScore* is not propagated to the index



* A projection is the set of attributes that is copied from a table into a secondary index. The partition key and sort key of the table are always projected into the index; you can project other attributes to support your application's query requirements. When you query an index, Amazon DynamoDB can access any attribute in the projection as if those attributes were in a table of their own.
* DynamoDB automatically synchronizes each global secondary index with its base table. When an application writes or deletes items in a table, any global secondary indexes on that table are updated asynchronously, using an eventually consistent model. Applications never write directly to an index. However, it is important that you understand the implications of how DynamoDB maintains these indexes.

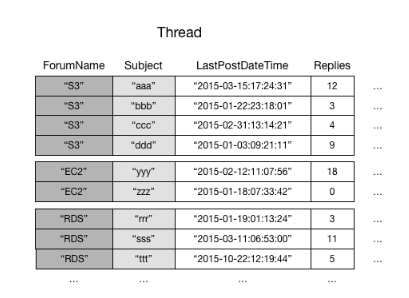
## Provisioned Throughput Considerations for Global Secondary Indexes

When you create a global secondary index on a provisioned mode table, you must specify read and write capacity units for the expected workload on that index. The provisioned throughput settings of a global secondary index are separate from those of its base table. A Query operation on a global secondary index consumes read capacity units from the index, not the base table. When you put, update or delete items in a table, the global secondary indexes on that table are also updated; these index updates consume write capacity units from the index, not from the base table.

# Local Secondary Indexes

Some applications only need to query data using the base table's primary key; however, there may be situations where an alternate sort key would be helpful. To give your application a choice of sort keys, you can create one or more local secondary indexes on a table and issue Query or Scanrequests against these indexes.

For example, consider the Thread table that is defined in [Creating Tables and Loading Sample Data](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/SampleData.html). This table is useful for an application such as the [AWS Discussion Forums](https://forums.aws.amazon.com/). The following diagram shows how the items in the table would be organized. (Not all of the attributes are shown.)



DynamoDB stores all of the items with the same partition key value contiguously. In this example, given a particular ForumName, a Query operation could immediately locate all of the threads for that forum. Within a group of items with the same partition key value, the items are sorted by sort key value. If the sort key (Subject) is also provided in the query, DynamoDB can narrow down the results that are returned—for example, returning all of the threads in the "S3" forum that have a Subject beginning with the letter "a".

Some requests might require more complex data access patterns. For example:

* Which forum threads get the most views and replies?
* Which thread in a particular forum has the largest number of messages?
* How many threads were posted in a particular forum within a particular time period?

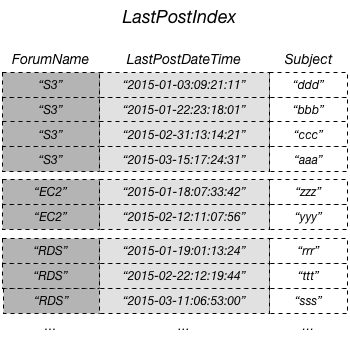
To answer these questions, the Query action would not be sufficient. Instead, you would have to Scan the entire table. For a table with millions of items, this would consume a large amount of provisioned read throughput and take a long time to complete.

However, you can specify one or more local secondary indexes on non-key attributes, such asReplies or LastPostDateTime.

A local secondary index maintains an alternate sort key for a given partition key value. A local secondary index also contains a copy of some or all of the attributes from its base table; you specify which attributes are projected into the local secondary index when you create the table. The data in a local secondary index is organized by the same partition key as the base table, but with a different sort key. This lets you access data items efficiently across this different dimension. For greater query or scan flexibility, you can create up to five local secondary indexes per table.

Suppose that an application needs to find all of the threads that have been posted within the last three months. Without a local secondary index, the application would have to Scan the entire Thread table and discard any posts that were not within the specified time frame. With a local secondary index, a Query operation could use LastPostDateTime as a sort key and find the data quickly.

The following diagram shows a local secondary index named LastPostIndex. Note that the partition key is the same as that of the Thread table, but the sort key is LastPostDateTime.



Every local secondary index must meet the following conditions:

* The partition key is the same as that of its base table.
* The sort key consists of exactly one scalar attribute.
* The sort key of the base table is projected into the index, where it acts as a non-key attribute.

Every local secondary index automatically contains the partition and sort keys from its base table; you can optionally project non-key attributes into the index. When you query the index, DynamoDB can retrieve these projected attributes efficiently. When you query a local secondary index, the query can also retrieve attributes that are not projected into the index. DynamoDB will automatically fetch these attributes from the base table, but at a greater latency and with higher provisioned throughput costs.

For any local secondary index, you can store up to 10 GB of data per distinct partition key value. This figure includes all of the items in the base table, plus all of the items in the indexes, that have the same partition key value.

## Querying a Local Secondary Index

In a DynamoDB table, the combined partition key value and sort key value for each item must be unique. However, in a local secondary index, the sort key value does not need to be unique for a given partition key value. If there are multiple items in the local secondary index that have the same sort key value, a Query operation will return all of the items that have the same partition key value. In the response, the matching items are not returned in any particular order.

You can query a local secondary index using either eventually consistent or strongly consistent reads. To specify which type of consistency you want, use the ConsistentRead parameter of the Query operation. A strongly consistent read from a local secondary index will always return the latest updated values. If the query needs to fetch additional attributes from the base table, then those attributes will be consistent with respect to the index.

## Item Writes and Local Secondary Indexes

DynamoDB automatically keeps all local secondary indexes synchronized with their respective base tables. Applications never write directly to an index.

## Provisioned Throughput Considerations for Local Secondary Indexes

When you create a table in DynamoDB, you provision read and write capacity units for the table's expected workload. That workload includes read and write activity on the table's local secondary indexes.

# Capturing Table Activity with DynamoDB Streams

DynamoDB Streams enables solutions such as these, and many others. DynamoDB Streams captures a time-ordered sequence of item-level modifications in any DynamoDB table, and stores this information in a log for up to 24 hours. Applications can access this log and view the data items as they appeared before and after they were modified, in near real time.

Encryption at rest encrypts the data in DynamoDB streams.

A DynamoDB stream is an ordered flow of information about changes to items in an Amazon DynamoDB table. When you enable a stream on a table, DynamoDB captures information about every modification to data items in the table.

DynamoDB Streams guarantees the following:

* Each stream record appears exactly once in the stream.
* For each item that is modified in a DynamoDB table, the stream records appear in the same sequence as the actual modifications to the item.

AWS maintains separate endpoints for DynamoDB and DynamoDB Streams. To work with database tables and indexes, your application will need to access a DynamoDB endpoint. To read and process DynamoDB Streams records, your application will need to access a DynamoDB Streams endpoint in the same region.

## Reading and Processing a Stream

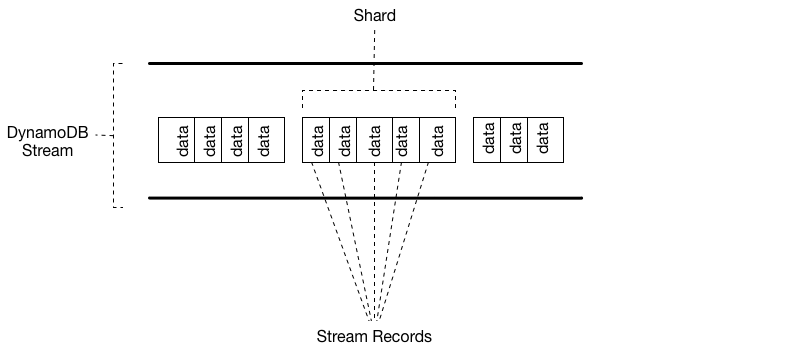
A stream consists of stream records. Each stream record represents a single data modification in the DynamoDB table to which the stream belongs. Each stream record is assigned a sequence number, reflecting the order in which the record was published to the stream.

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

Shards are ephemeral: They are created and deleted automatically, as needed. Any shard can also split into multiple new shards; this also occurs automatically. (Note that it is also possible for a parent shard to have just one child shard.) A shard might split in response to high levels of write activity on its parent table, so that applications can process records from multiple shards in parallel.

Because shards have a lineage (parent and children), an application must always process a parent shard before it processes a child shard. This will ensure that the stream records are also processed in the correct order. (If you use the DynamoDB Streams Kinesis Adapter, this is handled for you: Your application will process the shards and stream records in the correct order, and automatically handle new or expired shards, as well as shards that split while the application is running

The following diagram shows the relationship between a stream, shards in the stream, and stream records in the shards.



**Note**

If you perform a PutItem or UpdateItem operation that does not change any data in an item, then DynamoDB Streams will not write a stream record for that operation.

To access a stream and process the stream records within, you must do the following:

* Determine the unique Amazon Resource Name (ARN) of the stream that you want to access.
* Determine which shard(s) in the stream contain the stream records that you are interested in.
* Access the shard(s) and retrieve the stream records that you want.

**Note**

No more than 2 processes at most should be reading from the same Streams shard at the same time. Having more than 2 readers per shard may result in throttling.

### Data Retention Limit for DynamoDB Streams

All data in DynamoDB Streams is subject to a 24 hour lifetime. You can retrieve and analyze the last 24 hours of activity for any given table; however, data older than 24 hours is susceptible to trimming (removal) at any moment.

If you disable a stream on a table, the data in the stream will continue to be readable for 24 hours. After this time, the data expires and the stream records are automatically deleted. Note that there is no mechanism for manually deleting an existing stream; you just need to wait until the retention limit expires (24 hours), and all the stream records will be deleted.

Using the Kinesis Adapter is the recommended way to consume Streams from DynamoDB.

# Cross-Region Replication

You can create tables that are automatically replicated across two or more AWS Regions, with full support for multi-master writes. This gives you the ability to build fast, massively scaled applications for a global user base without having to manage the replication process. For more information, see[Global Tables](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/GlobalTables.html).

# DynamoDB Streams and AWS Lambda Triggers

Amazon DynamoDB is integrated with AWS Lambda so that you can create triggers—pieces of code that automatically respond to events in DynamoDB Streams. With triggers, you can build applications that react to data modifications in DynamoDB tables.

# **On-Demand Backup and Restore for DynamoDB**

Do it from tutorials dojo

# 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. For more information regarding on-demand backups, see [On-Demand Backup and Restore for DynamoDB](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/BackupRestore.html)

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.

In addition, point-in-time operations don't affect performance or API latencies.

# **Global Tables**

Do it from tutorials dojo

# **Security**

Do it from tutorials dojo

# Resilience and Disaster Recovery in Amazon DynamoDB

In addition to the AWS global infrastructure, Amazon DynamoDB offers several features to help support your data resiliency and backup needs.

**On-demand backup and restore**

DynamoDB provides on-demand backup capability. It allows you to create full backups of your tables for long-term retention and archival.

**Point-in-time recovery**

Point-in-time recovery helps protect your 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.

# Security Best Practices for Amazon DynamoDB

**Preventive Practices -**

Encryption at rest

Use IAM roles to authenticate access to DynamoDB

Use IAM policies for DynamoDB base authorization

Use IAM policy conditions for fine-grained access control

Use a VPC endpoint and policies to access DynamoDB

Consider client-side encryption

**Detective Practices –**

Use AWS CloudTrail to monitor AWS managed KMS key usage

Use CloudTrail to monitor DynamoDB control-plane operations

Consider using DynamoDB Streams to monitor modify/update data-plane operations

Monitor DynamoDB configuration with AWS Config

Monitor DynamoDB compliance with AWS Config rules

Tag your DynamoDB resources for identification and automation

# **In-Memory Acceleration with 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.

DAX supports server-side encryption. With encryption at rest, the data persisted by DAX on disk will be encrypted. DAX writes data to disk as part of propagating changes from the primary node to read replicas. For more information, see [DAX Encryption at Rest](https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DAXEncryptionAtRest.html).

## 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 not ideal for:

* Applications that require strongly consistent reads (or cannot tolerate eventually consistent reads).
* Applications that do not require microsecond response times for reads, or that do not need to offload repeated read activity from underlying tables.
* Applications that are write-intensive, or that do not perform much read activity.
* Applications that are already using a different caching solution with DynamoDB, and are using their own client-side logic for working with that caching solution.

# **Best Practices for DynamoDB**

Do from tutorials dojo