What is Amazon DynamoDB?

Topics

Welcome to the Amazon DynamoDB Developer Guide. DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. If you are a developer, you can use DynamoDB to create a database table that can store and retrieve any amount of data, and serve any level of request traffic. DynamoDB automatically spreads the data and traffic for the table over a sufficient number of servers to handle the request capacity specified by the customer and the amount of data stored, while maintaining consistent and fast performance. All data items are stored on solid state disks (SSDs) and are automatically replicated across multiple Availability Zones in a Region to provide built-in high availability and data durability.

If you are a database administrator, you can create a new DynamoDB database table, scale up or down your request capacity for the table without downtime or performance degradation, and gain visibility into resource utilization and performance metrics, all through the AWS Management Console. With DynamoDB, you can offload the administrative burdens of operating and scaling distributed databases to AWS, so you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling.

If you are a first-time user of DynamoDB, we recommend that you begin by reading the following sections:

- **What is DynamoDB—**The rest of this section describes the underlying data model, the operations it supports, and the class libraries that you can use to develop applications that use DynamoDB.
- **Getting Started (p. 12)—**The Getting Started section walks you through the process of creating sample tables, uploading data, and performing some basic database operations.

Beyond getting started, you'll probably want to learn more about application development with DynamoDB. The following sections provide additional information.

- **Working with DynamoDB**—The following sections provide in-depth information about the key DynamoDB concepts:
  - Working with Tables (p. 55)

- Working with Items (p. 86)
- Query and Scan Operations (p. 166)
- Improving Data Access with Secondary Indexes (p. 226)

- **Using AWS SDKs**—AWS provides SDKs for you to develop applications using DynamoDB. These SDKs provide low-level API methods that correspond closely to the underlying DynamoDB operations. The .NET SDK also provides a helper class to further simplify your development work. In addition, the AWS SDKs for Java and .NET also provide an object persistence model API that you can use to map your client-side classes to DynamoDB tables. This allows you to call object methods instead of making low-level API calls. For more information, including working samples, see Using the AWS SDKs with DynamoDB (p. 335).

  In addition to .NET, Java, and PHP examples provided in this guide, the other AWS SDKs also support DynamoDB, including JavaScript, Python, Android, iOS, and Ruby. For links to the complete set of AWS SDKs, see Start Developing with Amazon Web Services.

Service Highlights

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. With a few clicks in the AWS Management Console, customers can create a new DynamoDB database table, scale up or down their request capacity for the table without downtime or performance degradation, and gain visibility into resource utilization and performance metrics. DynamoDB enables customers to offload the administrative burdens of operating and scaling distributed databases to AWS, so they don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling.

DynamoDB is designed to address the core problems of database management, performance, scalability, and reliability. Developers can create a database table and grow its request traffic or storage without limit. DynamoDB automatically spreads the data and traffic for the table over a sufficient number of servers to handle the request capacity specified by the customer and the amount of data stored, while maintaining consistent, fast performance. All data items are stored on Solid State Disks (SSDs) and are automatically replicated across multiple Availability Zones in a Region to provide built-in high availability and data durability.

DynamoDB enables customers to offload the administrative burden of operating and scaling a highly available distributed database cluster while only paying a low variable price for the resources they consume.

The following are some of the major DynamoDB features:

- **Scalable —** DynamoDB is designed for seamless throughput and storage scaling.
  - **Provisioned Throughput —** When creating a table, simply specify how much throughput capacity you require. DynamoDB allocates dedicated resources to your table to meet your performance requirements, and automatically partitions data over a sufficient number of servers to meet your request capacity. If your application requirements change, simply update your table throughput capacity using the AWS Management Console or the DynamoDB APIs. You are still able to achieve your prior throughput levels while scaling is underway.
  - **Automated Storage Scaling —** There is no limit to the amount of data you can store in an DynamoDB table, and the service automatically allocates more storage, as you store more data using the DynamoDB write APIs.
  - **Fully Distributed, Shared Nothing Architecture —** DynamoDB scales horizontally and seamlessly scales a single table over hundreds of servers.
- **Fast, Predictable Performance—** Average service-side latencies for DynamoDB are typically single-digit milliseconds. The service runs on solid state disks, and is built to maintain consistent, fast latencies at any scale.

- **Easy Administration—** DynamoDB is a fully managed service – you simply create a database table and let the service handle the rest. You don't need to worry about hardware or software provisioning, setup and configuration, software patching, operating a reliable, distributed database cluster, or partitioning data over multiple instances as you scale.
- **Built-in Fault Tolerance—** DynamoDB has built-in fault tolerance, automatically and synchronously replicating your data across multiple Availability Zones in a Region for high availability and to help protect your data against individual machine, or even facility failures.
- **Flexible —** DynamoDB does not have a fixed schema. Instead, each data item may have a different number of attributes. Multiple data types (strings, numbers, binary, and sets) add richness to the data model.
- **Efficient Indexing —** Every item in an DynamoDB table is identified by a primary key, allowing you to access data items quickly and efficiently. You can also define secondary indexes on non-key attributes, and query your data using an alternate key.
- **Strong Consistency, Atomic Counters—** Unlike many non-relational databases, DynamoDB makes development easier by allowing you to use strong consistency on reads to ensure you are always reading the latest values. DynamoDB supports multiple native data types (numbers, strings, binaries, and multi-valued attributes). The service also natively supports atomic counters, allowing you to atomically increment or decrement numerical attributes with a single API call.
- **Cost Effective—** DynamoDB is designed to be extremely cost-efficient for workloads of any scale. You can get started with a free tier that allows more than 40 million database operations per month, and pay low hourly rates only for the resources you consume above that limit. With easy administration and efficient request pricing, DynamoDB can offer significantly lower total cost of ownership (TCO) for your workload compared to operating a relational or non-relational database on your own.
- **Secure—** DynamoDB is secure and uses proven cryptographic methods to authenticate users and prevent unauthorized data access. It also integrates with AWS Identity and Access Management for fine-grained access control for users within your organization.
- **Integrated Monitoring—** DynamoDB displays key operational metrics for your table in the AWS Management Console. The service also integrates with CloudWatch so you can see your request throughput and latency for each DynamoDB table, and easily track your resource consumption.
- **Amazon Redshift Integration—**You can load data from DynamoDB tables into Amazon Redshift, a fully managed data warehouse service. You can connect to Amazon Redshift with a SQL client or business intelligence tool using standard PostgreSQL JDBC or ODBC drivers, and perform complex SQL queries and business intelligence tasks on your data.
- **Amazon Elastic MapReduce Integration—** DynamoDB also integrates with Amazon Elastic MapReduce (Amazon EMR). Amazon EMR allows businesses to perform complex analytics of their large datasets using a hosted pay-as-you-go Hadoop framework on AWS. With the launch of DynamoDB, it is easy for customers to use Amazon EMR to analyze datasets stored in DynamoDB and archive the results in Amazon Simple Storage Service (Amazon S3), while keeping the original dataset in DynamoDB intact. Businesses can also use Amazon EMR to access data in multiple stores (i.e. DynamoDB and Amazon RDS), perform complex analysis over this combined dataset, and store the results of this work in Amazon S3.

DynamoDB Data Model

Topics

Data Model Concepts - Tables, Items, and Attributes

The DynamoDB data model concepts include tables, items and attributes.

In Amazon DynamoDB, a database is a collection of tables. A table is a collection of items and each item is a collection of attributes.

In a relational database, a table has a predefined schema such as the table name, primary key, list of its column names and their data types. All records stored in the table must have the same set of columns. DynamoDB is a NoSQL database: Except for the required primary key, an DynamoDB table is schema-less. Individual items in an DynamoDB table can have any number of attributes, although there is a limit of 64 KB on the item size. An item size is the sum of lengths of its attribute names and values (binary and UTF-8 lengths).

Each attribute in an item is a name-value pair. An attribute can be single-valued or multi-valued set. For example, a book item can have title and authors attributes. Each book has one title but can have many authors. The multi-valued attribute is a set; duplicate values are not allowed.

For example, consider storing a catalog of products in DynamoDB. You can create a table, *ProductCatalog*, with the Id attribute as its primary key.

```
ProductCatalog ( Id, ... )
```

You can store various kinds of product items in the table. The following table shows sample items.

Example items

```
{
   Id = 101
   ProductName = "Book 101 Title"
   ISBN = "111-1111111111"
   Authors = [ "Author 1", "Author 2" ]
   Price = -2
   Dimensions = "8.5 x 11.0 x 0.5"
   PageCount = 500
   InPublication = 1
   ProductCategory = "Book"
}
```

```
{
   Id = 201
   ProductName = "18-Bicycle 201"
   Description = "201 description"
   BicycleType = "Road"
   Brand = "Brand-Company A"
   Price = 100
   Gender = "M"
   Color = [ "Red", "Black" ]
   ProductCategory = "Bike"
}
```

Example items

```
{
   Id = 202
   ProductName = "21-Bicycle 202"
   Description = "202 description"
   BicycleType = "Road"
   Brand = "Brand-Company A"
   Price = 200
   Gender = "M"
   Color = [ "Green", "Black" ]
   ProductCategory = "Bike"
}
```

In the example, the *ProductCatalog* table has one book item and two bicycle items. Item 101 is a book with many attributes including the Authors multi-valued attribute. Item 201 and 202 are bikes, and these items have a Color multi-valued attribute. The Id is the only required attribute. Note that attribute values are shown using JSON-like syntax for illustration purposes.

DynamoDB does not allow null or empty string attribute values.

Primary Key

When you create a table, in addition to the table name, you must specify the primary key of the table. DynamoDB supports the following two types of primary keys:

- **Hash Type Primary Key—**In this case the primary key is made of one attribute, a hash attribute. DynamoDB builds an unordered hash index on this primary key attribute. In the preceding example, the hash attribute for the ProductCatalog table is *Id*.
- **Hash and Range Type Primary Key—**In this case, the primary key is made of two attributes. The first attribute is the hash attribute and the second one is the range attribute. DynamoDB builds an unordered hash index on the hash primary key attribute and a sorted range index on the range primary key attribute. For example, Amazon Web Services maintains several forums (see Discussion Forums). Each forum has many threads of discussion and each thread has many replies. You can potentially model this by creating the following three tables:

Table Name	Primary Key Type	Hash Attribute Name	Range Attribute Name
Forum (Name, ...)	Hash	Attribute Name: Name	-
Thread (ForumName, Subject, ...)	Hash and Range	Attribute Name: ForumName	Attribute Name: Subject
Reply (Id, ReplyDateTime, ...)	Hash and Range	Attribute Name: Id	Attribute Name: ReplyDateTime

In this example, both the *Thread* and *Reply* tables have primary key of the hash and range type. For the *Thread* table, each forum name can have one or more subjects. In this case, ForumName is the hash attribute and Subject is the range attribute.

The *Reply* table has Id as the hash attribute and ReplyDateTime as the range attribute. The reply Id identifies the thread to which the reply belongs. When designing DynamoDB tables you have to take into account the fact that DynamoDB does not support cross-table joins. For example, the *Reply* table stores both the forum name and subject values in the Id attribute. If you have a thread reply item, you can then parse the Id attribute to find the forum name and subject and use the information to query the *Thread* or the *Forum* tables. This developer guide uses these tables to illustrate DynamoDB functionality. For information about these tables and sample data stored in these tables, see Example Tables and Data (p. 548).

Secondary Indexes

When you create a table with a hash-and-range key, you can optionally define one or more secondary indexes on that table. A secondary index lets you query the data in the table using an alternate key, in addition to queries against the primary key.

With the *Reply* table, you can query data items by Id (hash) or by Id and ReplyDateTime (hash and range). Now suppose you had an attribute in the table—PostedBy—with the user ID of the person who posted each reply. With a secondary index on PostedBy, you could query the data by Id (hash) and PostedBy (range). Such a query would let you retrieve all the replies posted by a particular user in a thread, with maximum efficiency and without having to access any other items.

DynamoDB supports two kinds of secondary indexes:

- Local secondary index — an index that has the same hash key as the table, but a different range key.
- Global secondary index — an index with a hash and range key that can be different from those on the table.

You can define up to five local secondary indexes and five global secondary indexes per table. For more information, see Improving Data Access with Secondary Indexes (p. 226).

DynamoDB Data Types

Amazon DynamoDB supports the following data types:

- **Scalar data types—**Number, String, and Binary.
- **Multi-valued types—**String Set, Number Set, and Binary Set.

For example, in the *ProductCatalog* table, the Id is a number type attribute and Authors is a String Set type attribute. Note that primary key attributes can be any scalar types, but not multi-valued types.

String

Strings are Unicode with UTF8 binary encoding. There is no upper limit to the string size when you assign it to an attribute except when the attribute is part of the primary key. For more information, see Limits in DynamoDB (p. 463). Also, the length of the attribute is constrained by the 64 KB item size limit. Note that the length of the attribute must be greater than zero.

String value comparison is used when returning ordered results in the `Query` and `Scan` APIs. Comparison is based on ASCII character code values. For example, "a" is greater that "A" , and "aa" is greater than "B". For a list of code values, see http://en.wikipedia.org/wiki/ASCII#ASCII_printable_characters.

Number

Numbers are positive or negative exact-value decimals and integers. A number can have up to 38 digits of precision after the decimal point, and can be between 10^-128 to 10^+126. The representation in DynamoDB is of variable length. Leading and trailing zeroes are trimmed.

Serialized numbers are sent to DynamoDB as String types, which maximizes compatibility across languages and libraries, however DynamoDB handles them as the Number type for mathematical operations.

Binary

Binary type attributes can store any binary data, for example, compressed data, encrypted data, or images. DynamoDB treats each byte of the binary data as unsigned when it compares binary values, for example, when evaluating query expressions.

There is no upper limit to the length of the binary value when you assign it to an attribute except when the attribute is part of the primary key. For more information, see Limits in DynamoDB (p. 463). Also, the length of the attribute is constrained by the 64 KB item size limit. Note that the length of the attribute must be greater than zero.

String, Number, and Binary Sets

DynamoDB also supports Number Sets, String Sets and Binary Sets. Multi-valued attributes such as Authors attribute in a book item and Color attribute of a product item are examples of string set type attributes. Because it is a set, the values in the set must be unique. Attribute sets are not ordered; the order of the values returned in a set is not preserved. DynamoDB does not support empty sets.

Supported Operations in DynamoDB

To work with tables and items, Amazon DynamoDB offers the following set of operations:

Table Operations

DynamoDB provides operations to create, update and delete tables. After the table is created, you can use the `UpdateTable` operation to increase or decrease a table's provisioned throughput. DynamoDB also supports an operation to retrieve table information (the `DescribeTable` operation) including the current status of the table, the primary key, and when the table was created. The `ListTables` operation enables you to get a list of tables in your account in the region of the endpoint you are using to communicate with DynamoDB. For more information, see Working with Tables (p. 55).

Item Operations

Item operations enable you to add, update and delete items from a table. The `UpdateItem` operation allows you to update existing attribute values, add new attributes, and delete existing attributes from an item. You can also perform conditional updates. For example, if you are updating a price value, you can set a condition so the update happens only if the current price is $20.

DynamoDB provides an operation to retrieve a single item (`GetItem`) or multiple items (`BatchGetItem`). You can use the `BatchGetItem` operation to retrieve items from multiple tables. For more information, see Working with Items (p. 86).

Query and Scan

The `Query` operation enables you to query a table using the hash attribute and an optional range filter. If the table has a secondary index, you can also `Query` the index using its key. You can query only tables whose primary key is of hash-and-range type; you can also query any secondary index on such tables. `Query` is the most efficient way to retrieve items from a table or a secondary index.

DynamoDB also supports a `Scan` operation, which you can use on a query or a secondary index. The `Scan` operation reads every item in the table or secondary index. For large tables and secondary indexes, a `Scan` can consume a large amount of resources; for this reason, we recommend that you design your applications so that you can use the `Query` operation mostly, and use `Scan` only where appropriate. For more information, see Query and Scan Operations (p. 166).

You can use conditional expressions in both the `Query` and `Scan` operations to control which items are returned.

Data Read and Consistency Considerations

DynamoDB maintains multiple copies of each item to ensure durability. When you receive an "operation successful" response to your write request, DynamoDB ensures that the write is durable on multiple servers. However, it takes time for the update to propagate to all copies. The data is eventually consistent, meaning that a read request immediately after a write operation might not show the latest change. However, DynamoDB offers you the option to request the most up-to-date version of the data. To support varied application requirements, DynamoDB supports both eventually consistent and strongly consistent read options.

Eventually Consistent Reads

When you read data (`GetItem`, `BatchGetItem`, `Query` or `Scan` operations), the response might not reflect the results of a recently completed write operation (`PutItem`, `UpdateItem` or `DeleteItem`). The response might include some stale data. Consistency across all copies of the data is usually reached within a second; so if you repeat your read request after a short time, the response returns the latest data. By default, the `Query` and `GetItem` operations perform eventually consistent reads, but you can optionally request strongly consistent reads. `BatchGetItem` operations are eventually consistent by default, but you can specify strongly consistent on a per-table basis. `Scan` operations are always eventually consistent. For more information about operations in DynamoDB, see Using the DynamoDB API (p. 445).

Strongly Consistent Reads

When you issue a strongly consistent read request, DynamoDB returns a response with the most up-to-date data that reflects updates by all prior related write operations to which DynamoDB returned a successful response. A strongly consistent read might be less available in the case of a network delay or outage. For the query or get item operations, you can request a strongly consistent read result by specifying optional parameters in your request.

Conditional Updates and Concurrency Control

In a multiuser environment, it is important to ensure data updates made by one client don't overwrite updates made by another client. This "lost update" problem is a classic database concurrency issue. Suppose two clients read the same item. Both clients get a copy of that item from DynamoDB. Client 1 then sends a request to update the item. Client 2 is not aware of any update. Later, Client 2 sends its own request to update the item, overwriting the update made by Client 1. Thus, the update made by Client 1 is lost.

DynamoDB supports a "conditional write" feature that lets you specify a condition when updating an item. DynamoDB writes the item only if the specified condition is met; otherwise it returns an error. In the "lost update" example, client 2 can add a condition to verify item values on the server-side are same as the

item copy on the client-side. If the item on the server is updated, client 2 can choose to get an updated copy before applying its own updates.

DynamoDB also supports an "atomic counter" feature where you can send a request to add or subtract from an existing attribute value without interfering with another simultaneous write request. For example, a web application might want to maintain a counter per visitor to its site. In this case, the client only wants to increment a value regardless of what the previous value was. DynamoDB write operations support incrementing or decrementing existing attribute values.

For more information, see Working with Items (p. 86).

Provisioned Throughput in Amazon DynamoDB

When you create or update a table, you specify how much provisioned throughput capacity you want to reserve for reads and writes. DynamoDB will reserve the necessary machine resources to meet your throughput needs while ensuring consistent, low-latency performance.

A unit of *read capacity* represents one strongly consistent read per second (or two eventually consistent reads per second) for items as large as 4 KB. A unit of *write capacity* represents one write per second for items as large as 1 KB.

Items larger than 4 KB will require more than one read operation. The total number of read operations necessary is the item size, rounded up to the next multiple of 4 KB, divided by 4 KB. For example, to calculate the number of read operations for an item of 10 KB, you would round up to the next multiple of 4 KB (12 KB) and then divide by 4 KB, for 3 read operations.

The following table explains how to calculate the provisioned throughput capacity that you need.

Capacity Units Required For	How to Calculate
Reads	Number of item reads per second × 4 KB item size (If you use eventually consistent reads, you'll get twice as many reads per second.)
Writes	Number of item writes per second × 1 KB item size

If your application's read or write requests exceed the provisioned throughput for a table, then those requests might be throttled. You can use the AWS Management Console to monitor your provisioned and actual throughput and to change your provisioned capacity in anticipation of traffic changes.

For more information about specifying the provisioned throughput requirements for a table, see Specifying Read and Write Requirements for Tables (p. 56).

For tables with secondary indexes, DynamoDB consumes additional capacity units. For example, if you wanted to add a single 1 KB item to a table, and that item contained an indexed attribute, then you would need *two* write capacity units—one for writing to the table, and another for writing to the index. For more information, see Provisioned Throughput Considerations (p. 234) and Provisioned Throughput Considerations (p. 281).

Read Capacity Units

If your items are smaller than 4 KB in size, each read capacity unit will give you one strongly consistent read per second, or two eventually consistent reads per second. You cannot group multiple items in a single read operation, even if the items together are 4 KB or smaller. For example, if your items are 3 KB and you want to read 80 items per second from your table, then you need to provision 80 (reads per

second) × 4 KB (rounded up to the next 4 KB boundary) = 80 read capacity units for strong consistency. For eventual consistency, you need to provision only 40 read capacity units.

If your items are larger than 4 KB, you will need to round up the item size to the next 4 KB boundary. For example, if your items are 6 KB and you want to do 100 strongly consistent reads per second, you need to provision 100 (reads per second) × 2 (6 KB / 4 KB = 1.5, rounded up to the next whole number) = 200 read capacity units.

You can use the `Query` and `Scan` operations in DynamoDB to retrieve multiple consecutive items from a table in a single request. With these operations, DynamoDB uses the cumulative size of the processed items to calculate provisioned throughput. For example, if a `Query` operation retrieves 100 items that are 1 KB each, the read capacity calculation is *not* (100 × 4 KB) = 100 read capacity units, as if those items were retrieved individually using `GetItem` or `BatchGetItem`. Instead, the total would be only 25 read capacity units ((100 * 1024 bytes) = 100 KB, which is then divided by 4 KB). For more information see Item Size Calculations (p. 58).

Write Capacity Units

If your items are smaller than 1 KB in size, then each write capacity unit will give you 1 write per second. You cannot group multiple items in a single write operation, even if the items together are 1 KB or smaller. For example, if your items are 512 bytes and you want to write 100 items per second to your table, then you would need to provision 100 write capacity units.

If your items are larger than 1 KB in size, you will need to round the item size up to the next 1 KB boundary. For example, if your items are 1.5 KB and you want to do 10 writes per second, then you would need to provision 10 (writes per second) × 2 (1.5 KB rounded up to the next whole number) = 20 write capacity units.

Accessing DynamoDB

Amazon DynamoDB is a web service that uses HTTP and HTTPS as a transport and JavaScript Object Notation (JSON) as a message serialization format. Your application code can make requests directly to the DynamoDB web service API. Instead of making the requests to the DynamoDB API directly from your application, we recommend that you use the AWS Software Development Kits (SDKs). The easy-to-use libraries in the AWS SDKs make it unnecessary to call the DynamoDB API directly from your application. The libraries take care of request authentication, serialization, and connection management. For more information about using the AWS SDKs, see Using the AWS SDKs with DynamoDB (p. 335).

The AWS SDKs provide low-level APIs that closely match the underlying DynamoDB API. To further simplify application development, the SDKs also provide the following additional APIs:

- The Java and .NET SDKs provide APIs with higher levels of abstraction. These higher-level interfaces 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. This allows you to write object-centric code, rather than database-centric code.
- The .NET SDK provides helper classes that wrap some of the low-level API functionality to further simplify your coding.

For more information, see Using the AWS SDK for Java (p. 335) and Using the AWS SDK for .NET (p. 337).

If you decide not to use the AWS SDKs, then your application will need to construct individual service requests. Each request must contain a valid JSON payload and correct HTTP headers, including a valid AWS signature. For more information on constructing your own service requests, see Using the DynamoDB API (p. 445).

DynamoDB also provides a management console that enables you to work with tables and items. You can create, update, and delete tables without writing any code. You can view all the existing items in a table or use a query to filter the items in the table. You can add new items or delete items. You can also use the management console to monitor the performance of your tables. Using CloudWatch metrics in the console, you can monitor table throughput and other performance metrics. For more information, go to DynamoDB console.

Regions and Endpoints for DynamoDB

By default, the AWS SDKs and console for DynamoDB reference the US-West (Oregon) Region. As DynamoDB expands availability to new regions, new endpoints for these regions are also available to use in your own HTTP requests, the AWS SDKs, and the console. For a current list of supported regions and endpoints, see Regions and Endpoints.