**Amazon DynamoDB**

is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability.

Amazon DynamoDB stores three geographically distributed replicas of each table to enable high availability and data durability.

Data is synchronously replicated across 3 facilities (AZs) in a region.

DynamoDB is schema-less.

It is a non-relational, key-value type of database.

DynamoDB is a serverless service – there are no instances to provision or manage.

Push button scaling means that you can scale the DB at any time without incurring downtime.

DynamoDB can be used for storing session state data.

Provides very low latency.

Data is stored on SSD storage.

Multi-AZ redundancy and Cross-Region Replication option.

The underlying hardware storing data is spread across 3 geographically distinct data centers.

DynamoDB is made up of:

* Tables.
* Items.
* Attributes.

Tables are a collection of items and items are made up of attributes (columns).

Attributes consists of a name and a value or set of values.

Attributes in DynamoDB are like fields or columns in other database systems.

The aggregate size of an item cannot exceed 400KB including keys and all attributes.

Can store pointers to objects in S3, including items over 400KB.

Supports key value and document structures.

A key-value database stores data as a collection of key-value pairs in which a key serves as a unique identifier.

Key = the name of the data; Value = the data itself.

Documents can be written in JSON, HTML, or XML.

Some of the features and benefits of Amazon DynamoDB are summarized in the following table:

|  |  |
| --- | --- |
| **DynamoDB Feature** | **Benefit** |
| Serverless | Fully managed, fault tolerant service |
| Highly available | 99.99% Availability SLA – 99.999% for Global tables |
| NoSQL type of database with Name / Value structure | Flexible Schema, good for when data is not well structured or unpredictable |
| Horizontal scaling | Seamless scalability to any scale with push button scaling or Auto scaling |
| DynamoDB Streams | Captures a time-ordered sequence of item-level modifications in a DynamoDB table and durably stores the information for 24 hours. Often used with Lambda and the Kinesis Client Library (KCL). |
| DynamoDB Accelerator (DAX) | Fully managed in-memory cache for DynamoDB that increases performance (microsecond latency) |
| Transaction options | Strongly consistent or eventually consistent reads, support for ACID transactions |
| Backup | Point-in-time recovery down to the second in last 35 days; On-demand backup and restore |
| Global Tables | Fully managed multi-region, multi-master solution |

**Anti-Patterns**

Amazon DynamoDB is not ideal for the following situations:

* Traditional RDS apps.
* Joins and/or complex transactions.
* BLOB data.
* Large data with low I/O rate.

**Access control**

All authentication and access control is managed using IAM.

DynamoDB supports identity-based policies:

* Attach a permissions policy to a user or a group in your account.
* Attach a permissions policy to a role (grant cross-account permissions).

DynamoDB doesn’t support resource-based policies.

You can use a special IAM condition to restrict user access to only their own records.

In DynamoDB, the primary resources are tables.

DynamoDB also supports additional resource types, indexes, and streams.

You can create indexes and streams only in the context of an existing DynamoDB table (subresources).

These resources and subresources have unique Amazon Resource Names (ARNs) associated with them, as shown in the following table.

|  |  |
| --- | --- |
| **Resource Type** | **ARN Format** |
| Table | arn:aws:dynamodb:region:account-id:table:table/name-name |
| Index | arn:aws:dynamodb:region:account-id:table:table/name-name/index/index-name |
| Stream | arn:aws:dynamodb:region:account-id:table:table/name-name/stream/stream-label |

**Partitions**

Amazon DynamoDB stores data in partitions.

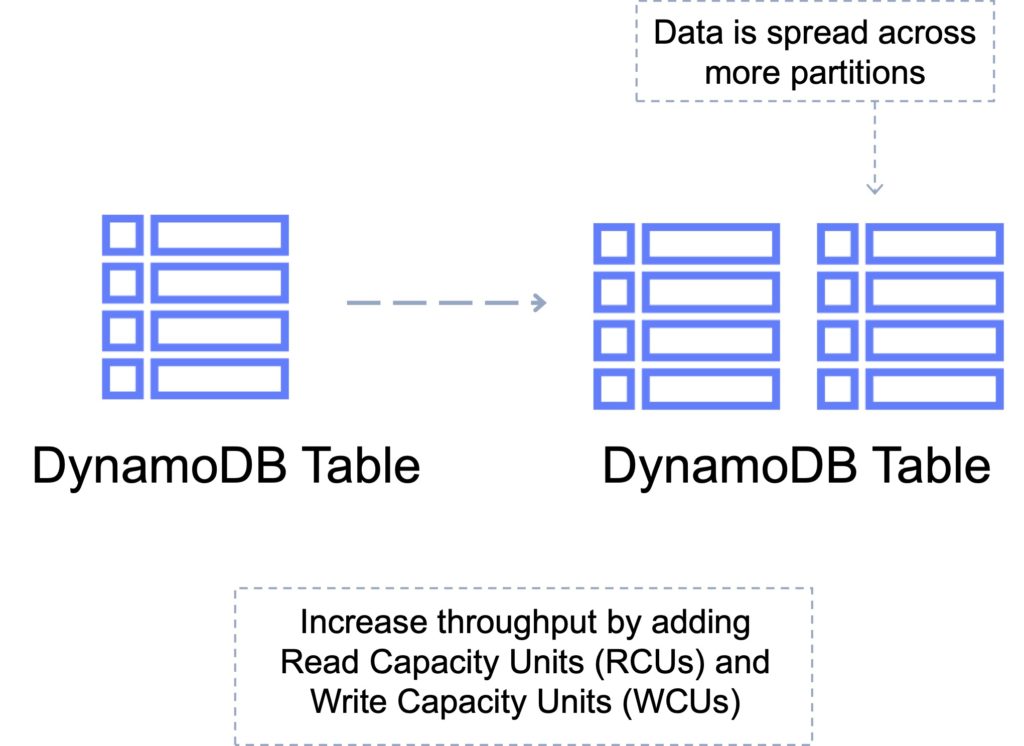
A partition is an allocation of storage for a table that is automatically replicated across multiple AZs within an AWS Region.

Partition management is handled entirely by DynamoDB—you never have to manage partitions yourself.

DynamoDB allocates sufficient partitions to your table so that it can handle your provisioned throughput requirements.

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.



**Primary Keys**

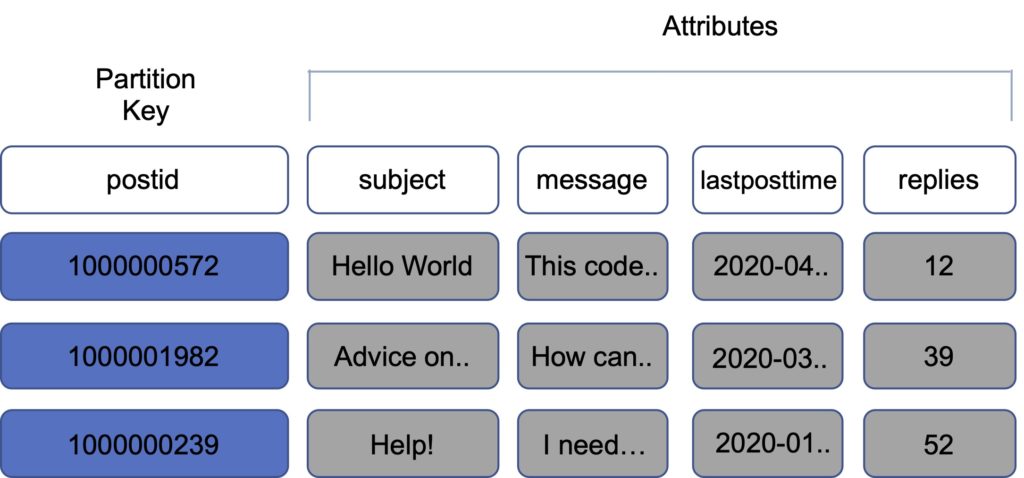
DynamoDB stores and retrieves data based on a Primary key.

There are two types of Primary key:

Partition key – unique attribute (e.g. user ID).

* Value of the Partition key is input to an internal hash function which determines the partition or physical location on which the data is stored.
* If you are using the Partition key as your Primary key, then no two items can have the same partition key.

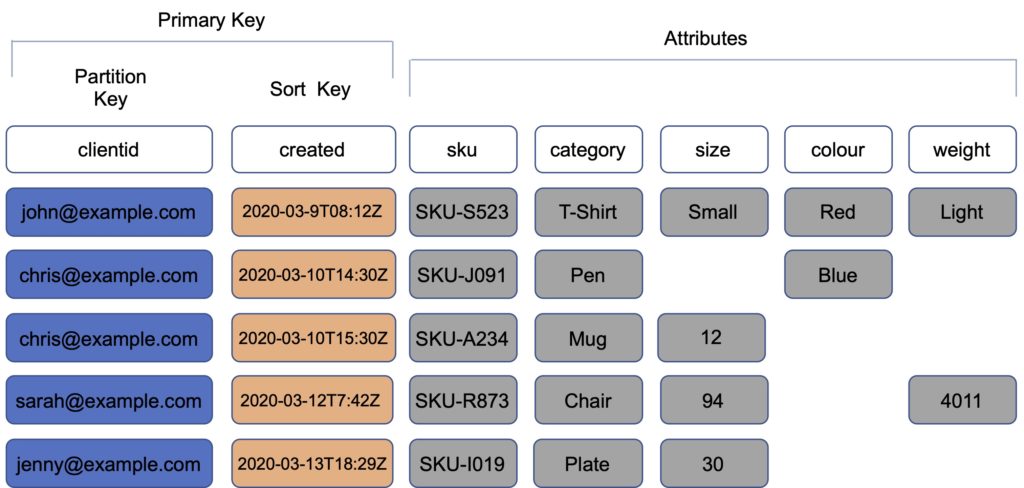
The image below depicts a table with a partition key:



Composite key – Partition key + Sort key in combination.

* Example is user posting to a forum. Partition key would be the user ID, Sort key would be the timestamp of the post.
* 2 items may have the same Partition key, but they must have a different Sort key.
* All items with the same Partition key are stored together, then sorted according to the Sort key value.
* Allows you to store multiple items with the same partition key.

The image below depicts a table with a composite key:



**Partitions and Performance**

DynamoDB evenly distributes provisioned throughput—read capacity units (RCUs) and write capacity units (WCUs) among partitions

If your access pattern exceeds 3000 RCU or 1000 WCU for a single partition key value, your requests might be throttled.

Reading or writing above the limit can be caused by these issues:

* Uneven distribution of data due to the wrong choice of partition key.
* Frequent access of the same key in a partition (the most popular item, also known as a hot key).
* A request rate greater than the provisioned throughput.

Best practices for partition keys:

* Use high-cardinality attributes – e.g. e-mailid, employee\_no, customerid, sessionid, orderid, and so on.
* Use composite attributes – e.g. customerid+productid+countrycode as the partition key and order\_date as the sort key.
* Cache popular items – use DynamoDB accelerator (DAX) for caching reads.
* Add random numbers or digits from a predetermined range for write-heavy use cases – e.g. add a random suffix to an invoice number such as INV00023-04593

**Consistency Models**

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. In this case, DynamoDB may return a server error (HTTP 500).
* Strongly consistent reads may have higher latency than eventually consistent reads.
* Strongly consistent reads are not supported on global secondary indexes.
* Strongly consistent reads use more throughput capacity than eventually consistent reads.

DynamoDB uses eventually consistent reads by default.

You can configure strongly consistent reads with the GetItem, Query and Scan APIs by setting the –consistent-read (or ConsistentRead) parameter to “true”.

**DynamoDB Transactions**

Amazon DynamoDB transactions simplify the developer experience of making coordinated, all-or-nothing changes to multiple items both within and across tables.

Transactions provide atomicity, consistency, isolation, and durability (ACID) in DynamoDB.

Enables reading and writing of multiple items across multiple tables as an all or nothing operation.

Checks for a pre-requisite condition before writing to a table.

With the transaction write API, you can group multiple Put, Update, Delete, and ConditionCheck actions.

You can then submit the actions as a single TransactWriteItems operation that either succeeds or fails as a unit.

The same is true for multiple Get actions, which you can group and submit as a single TransactGetItems operation.

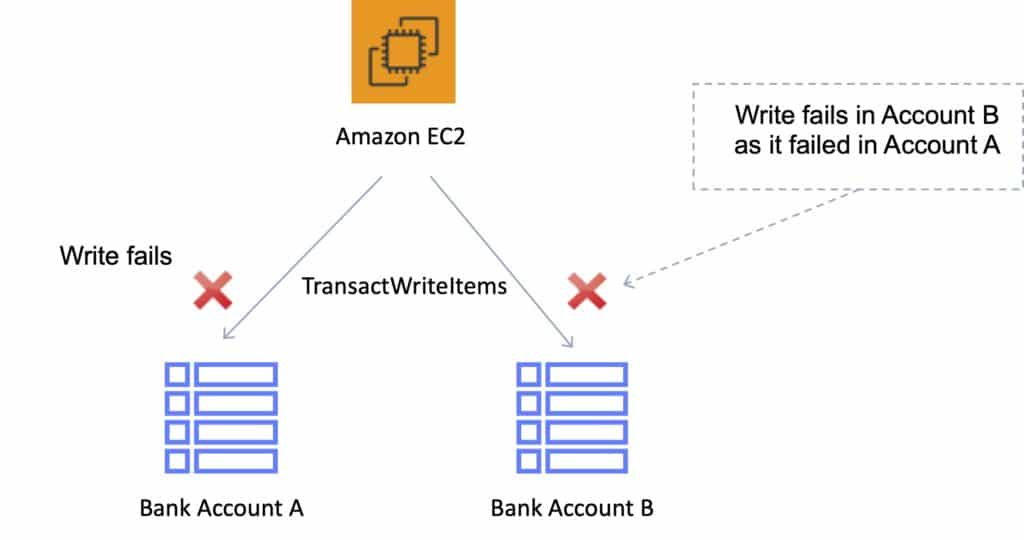
There is no additional cost to enable transactions for DynamoDB tables.

You pay only for the reads or writes that are part of your transaction.

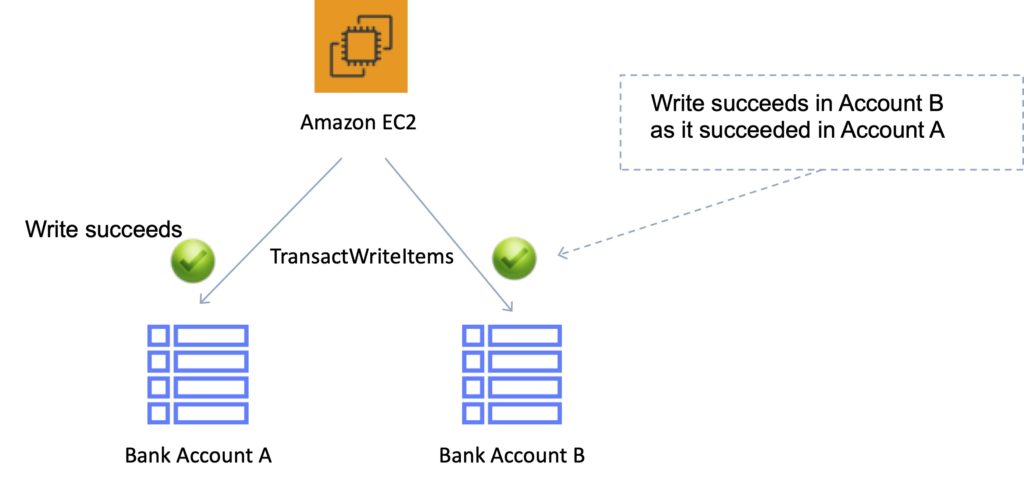
DynamoDB performs two underlying reads or writes of every item in the transaction: one to prepare the transaction and one to commit the transaction.

These two underlying read/write operations are visible in your Amazon CloudWatch metrics.

The following diagram depicts a failed write using DynamoDB Transactions:



The following diagram depicts a successful write using DynamoDB Transactions:



**Scan and Query API calls**

**Scan**

The Scan operation returns one or more items and item attributes by accessing every item in a table or a secondary index.

A single Scan operation reads up to the maximum number of items set (if using the Limit parameter) or a maximum of 1 MB.

Scan API calls can use a lot of RCUs as they access every item in the table.

You can use the ProjectionExpression parameter so that Scan only returns some of the attributes, rather than all of them.

If you need to further refine the Scan results, you can optionally provide a filter expression.

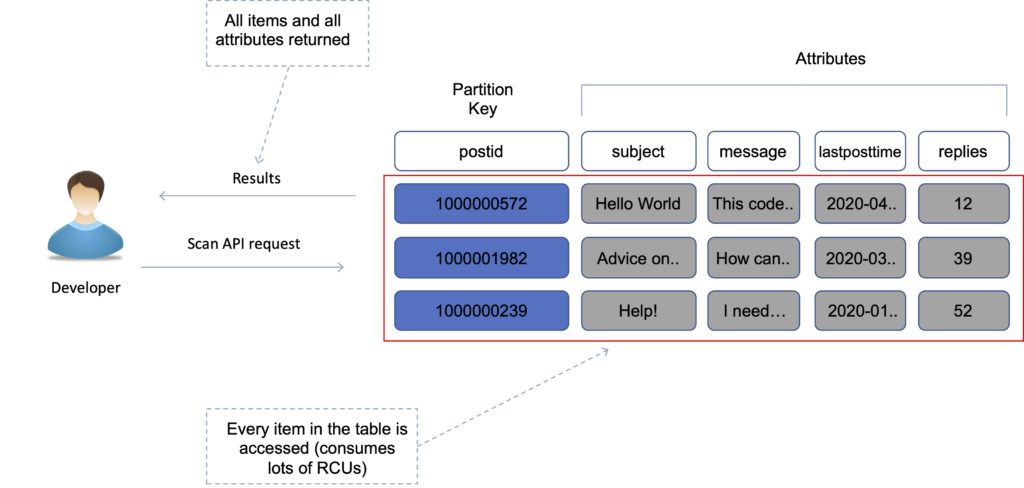
A filter expression is applied after a Scan finishes but before the results are returned.

Scan operations proceed sequentially.

For faster performance on a large table or secondary index, applications can request a parallel Scan operation by providing the Segment and TotalSegments parameters.

Scan uses eventually consistent reads when accessing the data in a table.

If you need a consistent copy of the data, as of the time that the Scan begins, you can set the ConsistentRead parameter to true.



**Query**

A query operation finds items in your table based on the primary key attribute and a distinct value to search for.

For example, you might search for a user ID value and all attributes related to that item would be returned.

You can use an optional sort key name and value to refine the results.

For example, if your sort key is a timestamp, you can refine the query to only select items with a timestamp of the last 7 days.

By default, a query returns all the attributes for the items, but you can use the ProjectionExpression parameter if you want the query to only return the attributes you want to see.

Results are always sorted by the sort key.

Numeric order is used – by default in ascending order (e.g. 1,2,3,4).

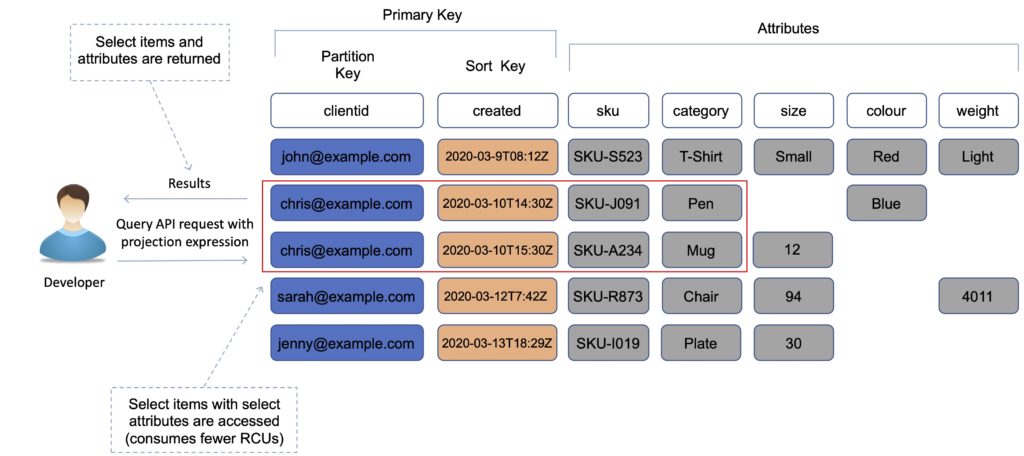
ASCII character code values are used.

You can reverse the order by setting the ScanIndexForward (yes, it’s a query, not a scan) parameter to false.

By default, queries are eventually consistent.

To use strongly consistent you need to explicitly set this in the query.

The following diagram shows a query API call with a projection expression limiting the attributes that are returned:



**Scan vs Query**

Query is more efficient than Scan.

Scan dumps the entire table, then filters out the values that provide the desired result (removing unwanted data).

This adds an extra step of removing the data you don’t want.

As the table grows, the scan operation takes longer.

A Scan operation on a large table can use up the provisioned throughput for a large table in just a single operation.

**Performance optimization**

You can reduce the impact of a query or scan by setting a smaller page size which uses fewer read operations.

A larger number of smaller operations will allow other requests to succeed without throttling.

Avoid using scan operations if you can: design tables in a way that you can use the Query, Get, or BatchGetItem APIs.

Scan performance optimization:

* By default, a scan operation processes data sequentially and returns data in 1MB increments before moving on to retrieve the next 1MB of data. It can only scan 1 partition at a time.
* You can configure DynamoDB to use Parallel scans instead by logically dividing a table or index into segments and scanning each segment in parallel.

**Note:** best to avoid parallel scans if your table or index is already incurring heavy read / write activity from other applications.

**Indexes**

An index is a data structure which allows you to perform fast queries on specific columns in a table.

You select columns that you want included in the index and run your searches on the index instead of the entire dataset.

There are 2 types of index supported for speeding up queries in DynamoDB:

* Local Secondary Index.
* Global Secondary Index.

**Local Secondary Index (LSI)**

An LSI provides an alternative sort key to use for scans and queries.

It provides an alternative range key for your table, local to the hash key.

You can have up to five LSIs per table.

The sort key consists of exactly one scalar attribute.

The attribute that you choose must be a scalar String, Number, or Binary.

An LSI must be created at table creation time.

It can only be created when you are creating your table.

You cannot add, remove, or modify it later.

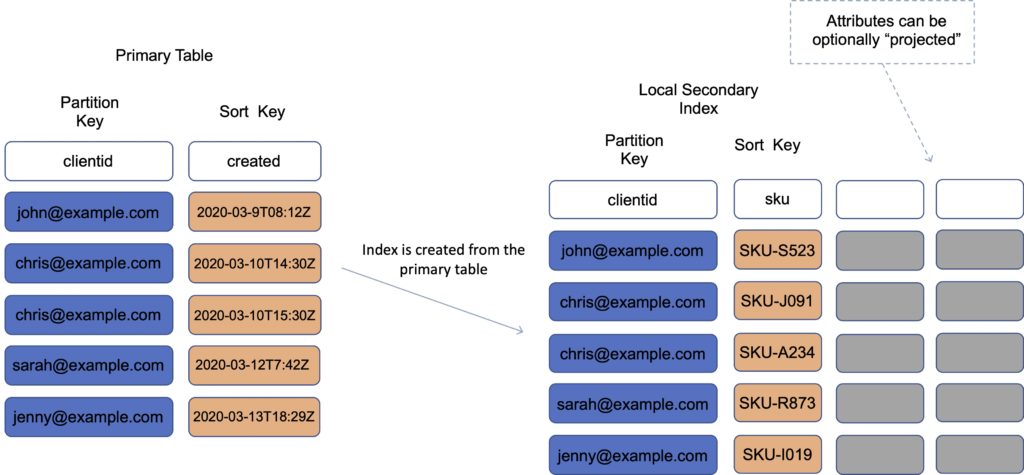
It has the same partition key as your original table (different sort key).

It gives you a different view of your data, organized by an alternative sort key.

Any queries based on this sort key are much faster using the index than the main table.

An example might be having a user ID as a partition key and account creation date as the sort key.

The key benefit of an LSI is that you can query on additional values in the table other than the partition key / sort key.



**Global Secondary Index (GSI)**

A GSI is used to speed up queries on non-key attributes use a GSI

You can create when you create your table or at any time later.

A GSI has a different partition key as well as a different sort key.

It gives a completely different view of the data.

It speeds up any queries relating to this alternative partition and sort key.

An example might be an email address as the partition key, and last login date as the sort key.

With a GSI the index is a new “table”, and you can project attributes on it.

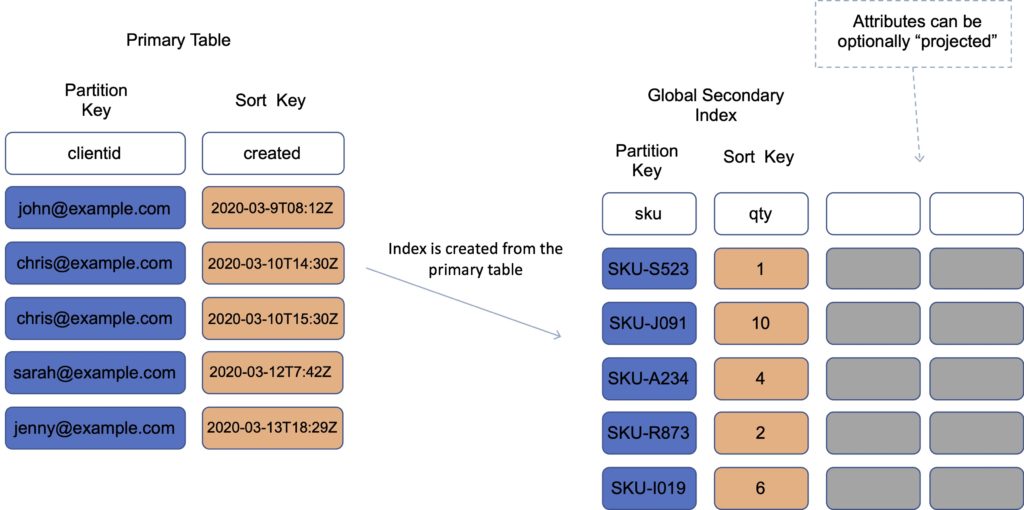
* The partition key and sort key of the original table are always projected (KEYS\_ONLY).
* Can specify extra attributes to project (INCLUDE).
* Can use all attributes from main table (ALL).

You must define RCU / WCU for the index

It is possible to add / modify GSI at any time.

If writes are throttled on the GSI, the main table will be throttled (even if there’s enough WCUs on the main table). LSIs do not cause any special throttling considerations.

***Exam tip:*** *You typically need to ensure that you have at least the same, or more, RCU/WCU specified in your GSI as in your main table to avoid throttling on your main table.*



**Performance and Optimization**

**DynamoDB Provisioned Capacity**

With provisioned capacity mode you specify the number of data reads and writes per second that you require for your application.

You can use auto scaling to automatically adjust your table’s capacity based on the specified utilization rate to ensure application performance while reducing costs.

When you create your table you specify your requirements using Read Capacity Units (RCUs) and Write Capacity Units (WCUs).

**Note:** WCUs and RCUs are spread between partitions evenly.

You can also use Auto Scaling with provisioned capacity.

DynamoDB auto scaling uses the AWS Application Auto Scaling service to dynamically adjust provisioned throughput capacity on your behalf, in response to 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.

Read capacity unit (RCU):

* Each API call to read data from your table is a read request.
* Read requests can be strongly consistent, eventually consistent, or transactional.
* For items up to 4 KB in size, one RCU can perform one *strongly consistent* read request per second.
* Items larger than 4 KB require additional RCUs.
* For items up to 4 KB in size, one RCU can perform two *eventually consistent* read requests per second.
* *Transactional* read requests require two RCUs to perform one read per second for items up to 4 KB.
* For example, a strongly consistent read of an 8 KB item would require two RCUs, an eventually consistent read of an 8 KB item would require one RCU, and a transactional read of an 8 KB item would require four RCUs.

Write capacity unit (WCU):

* Each API call to write data to your table is a write request.
* For items up to 1 KB in size, one WCU can perform one *standard* write request per second.
* Items larger than 1 KB require additional WCUs.
* *Transactional* write requests require two WCUs to perform one write per second for items up to 1 KB.
* For example, a standard write request of a 1 KB item would require one WCU, a standard write request of a 3 KB item would require three WCUs, and a transactional write request of a 3 KB item would require six WCUs.

Replicated write capacity unit (rWCU):

* When using DynamoDB global tables, your data is written automatically to multiple AWS Regions of your choice.
* Each write occurs in the local Region as well as the replicated Regions.

Streams read request unit:

* Each GetRecords API call to DynamoDB Streams is a streams read request unit.
* Each streams read request unit can return up to 1 MB of data.

Transactional read/write requests:

* In DynamoDB, a transactional read or write differs from a standard read or write because it guarantees that all operations contained in a single transaction set succeed or fail as a set.

**DynamoDB On-Demand Capacity**

With on-demand, you don’t need to specify your requirements.

DynamoDB instantly scales up and down based on the activity of your application.

Great for unpredictable / spikey workloads or new workloads that aren’t well understood.

You pay for what you use (pay per request).

You can switch between the provisioned capacity and on-demand pricing models once per day.

**Performance and Throttling**

Throttling occurs when the configured RCU or WCU are exceeded.

May receive the ProvisionedThroughputExceededException error.

This error indicates that your request rate is too high for the read / write capacity provisioned for the table.

The AWS SDKs for DynamoDB automatically retry requests that receive this exception.

Your request is eventually successful, unless your retry queue is too large to finish.

Possible causes of performance issues:

* Hot keys – one partition key is being read too often.
* Hot partitions – when data access is imbalanced, a “hot” partition can receive a higher volume of read and write traffic compared to other partitions.
* Large items – large items consume more RCUs and WCUs.

Resolution:

* Reduce the frequency of requests and use exponential backoff.
* Try to design your application for uniform activity across all logical partition keys in the table and its secondary indexes.
* Use burst capacity effectively – DynamoDB currently retains up to 5 minutes (300 seconds) of unused read and write capacity which can be consumed quickly.

**DynamoDB Accelerator (DAX)**

Amazon DynamoDB Accelerator (DAX) is a fully managed, highly available, in-memory cache for DynamoDB that delivers up to a 10x performance improvement.

Improves performance from milliseconds to microseconds, even at millions of requests per second.

DAX is a managed service that provides in-memory acceleration for DynamoDB tables.

Provides managed cache invalidation, data population, and cluster management.

DAX is used to improve READ performance (not writes).

You do not need to modify application logic, since DAX is compatible with existing DynamoDB API calls.

Ideal for read-heavy and bursty workloads such as auction applications, gaming, and retail sites when running special sales / promotions.

You can enable DAX with just a few clicks in the AWS Management Console or using the AWS SDK.

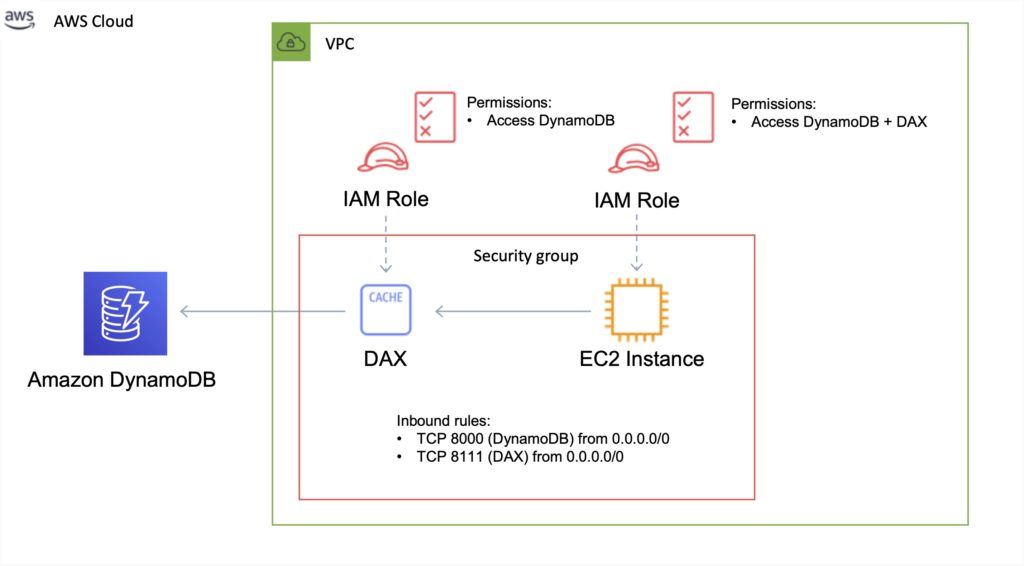
Just as with DynamoDB, you only pay for the capacity you provision.

Provisioned through clusters and charged by the node (runs on EC2 instances).

Pricing is per node-hour consumed and is dependent on the instance type you select.

How it works:

* DAX is a write-through caching service – this means the data is written to the cache as well as the back-end store at the same time.
* Allows you to point your DynamoDB API calls at the DAX cluster and if the item is in the cache (cache hit), DAX returns the result to the application.
* If the item requested is not in the cache (cache miss) then DAX performs an Eventually Consistent GetItem operation against DynamoDB
* Retrieval of data from DAX reduces the read load on DynamoDB tables.
* This may result in being able to reduce the provisioned read capacity on the table.



**DAX vs ElastiCache**

DAX is optimized for DynamoDB.

DAX does not support lazy loading (uses write-through caching).

With ElastiCache you have more management overhead (e.g. invalidation).

With ElastiCache you need to modify application code to point to cache.

ElastiCache supports more datastores.

**DynamoDB Time To Live (TTL)**

Automatically deletes an item after an expiry date / time.

Expired items are marked for deletion.

Great for removing irrelevant or old data such as:

* Session data.
* Event logs.
* Temporary data.

No extra cost and does not use WCU / RCU.

TTL is a background task operated by DynamoDB.

A TTL helps reduce storage and manage the table size over time.

The TTL is enabled per row (you define a TTL column and add the expiry date / time there).

DynamoDB typically deletes expired items within 48 hours of expiration.

Deleted items are also deleted from the LSI / GSI.

DynamoDB streams can help recover expired items.



**Exponential backoff**

Many components in a network can generate errors when overloaded.

In addition to simple retries all AWS SDKs use Exponential Backoff.

Progressively longer waits will occur between retries for improved flow control.

If after 1 minute this does not work, your request size may be exceeding the throughput for your read/write capacity.

If your workload is mainly reads consider offloading using DAX or ElastiCache.

If your workload is mainly writes consider increasing the WCUs for the table.

**DynamoDB Streams**

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.

You can also use the CreateTable or UpdateTable API operations to enable or modify a stream.

Logs are encrypted at rest and stored for 24 hours.

Accessed using a dedicated endpoint.

By default, just the Primary key is recorded.

Before and after images can be captured.

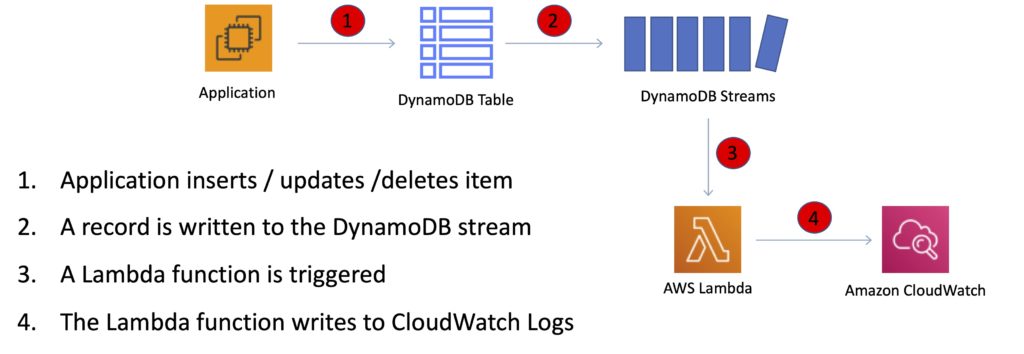
Events are recorded in near real-time.

Applications can take actions based on contents.

A stream can be an event source for Lambda.

Lambda polls the DynamoDB stream and executes code based on a DynamoDB streams event.

Data is stored in stream for 24 hours only.



The StreamSpecification parameter determines how the stream is configured:

StreamEnabled — Specifies whether a stream is enabled (true) or disabled (false) for the table.

StreamViewType — Specifies the information that will be written to the stream whenever data in the table is modified:

* KEYS\_ONLY — Only the key attributes of the modified item.
* NEW\_IMAGE — The entire item, as it appears after it was modified.
* OLD\_IMAGE — The entire item, as it appeared before it was modified.
* NEW\_AND\_OLD\_IMAGES — Both the new and the old images of the item.

**API calls**

**Writing data**

* PutItem – create data or full replacement (consumes WCU).
* UpdateItem – update data, partial update of attributes (can use atomic counters).
* Conditional writes – accept a write / update only if conditions are met.
* DeleteItem – delete an individual row (can perform conditional delete).
* DeleteTable – delete a whole table (quicker than using DeleteItem on all items).
* BatchWriteItem – can put or delete up to 25 items in one call (max 16MB write / 400KB per item).

Batching allows you to save in latency by reducing the number of API calls.

Operations are done in parallel for better efficiency.

**Reading data**

* GetItem – read based on primary key (eventually consistent by default, can request strongly consistent read). Projection expression can be specified to include only certain attributes.
* BatchGetItem – up to 100 items, up to 16MB per item. Items are retrieved in parallel to minimize latency.
* Query – return items based on PartitionKey value and optionally a sort key. FilterExpression can be used for filtering. Returns up to 1MB of data or number of items specified in Limit. Can do pagination on results. Can query table, local secondary index, or a global secondary index.
* Scan – scans the entire table (inefficient). Returns up to 1MB of data – use pagination to view more results. Consumes a lot of RCU. Can use a ProjectionExpression + FilterExpression.

**Optimistic Locking**

Optimistic locking is a strategy to ensure that the client-side item that you are updating (or deleting) is the same as the item in Amazon DynamoDB.

Protects database writes from being overwritten by the writes of others, and vice versa.

**Conditional Updates**

To manipulate data in an Amazon DynamoDB table, you use the PutItem, UpdateItem, and DeleteItem operations.

You can optionally specify a condition expression to determine which items should be modified.

If the condition expression evaluates to true, the operation succeeds; otherwise, the operation fails.

**Security**

VPC endpoints are available for DynamoDB.

Encryption at rest can be enabled using AWS KMS.

Encryption in transit uses SSL / TLS.

**Best practices**

Keep item sizes small.

If you are storing serial data in DynamoDB that will require actions based on date/time use separate tables for days, weeks, months.

Store more frequently and less frequently accessed data in separate tables.

If possible compress larger attribute values.

Store objects larger than 400KB in S3 and use pointers (S3 Object ID) in DynamoDB.

**Integrations**

ElastiCache can be used in front of DynamoDB for performance of reads on infrequently changed data.

Triggers integrate with AWS Lambda to respond to triggers.

Integration with RedShift:

* RedShift complements DynamoDB with advanced business intelligence.
* When copying data from a DynamoDB table into RedShift you can perform complex data analysis queries including joins with other tables.
* A copy operation from a DynamoDB table counts against the table’s read capacity.
* After data is copied, SQL queries do not affect the data in DynamoDB.

DynamoDB is integrated with Apache Hive on EMR. Hive can allow you to:

* Read and write data in DynamoDB tables allowing you to query DynamoDB data using a SQL-like language (HiveQL).
* Copy data from a DynamoDB table to an S3 bucket and vice versa.
* Copy data from a DynamoDB table into HDFS and vice versa.
* Perform join operations on DynamoDB tables.

**Scalability**

Push button scaling without downtime.

You can scale down only 4 times per calendar day.

AWS places default limits on the throughput you can provision.

DynamoDB can throttle requests that exceed the provisioned throughput for a table.

DynamoDB can also throttle read requests for an Index to prevent your application from consuming too many capacity units.

When a request is throttled it fails with an HTTP 400 code (Bad Request) and a ProvisionedThroughputExceeded exception.

**Cross Region Replication with Global Tables**

Amazon DynamoDB global tables provide a fully managed solution for deploying a multi-region, multi-master database.

When you create a global table, you specify the AWS regions where you want the table to be available.

DynamoDB performs all the necessary tasks to create identical tables in these regions and propagate ongoing data changes to all of them.

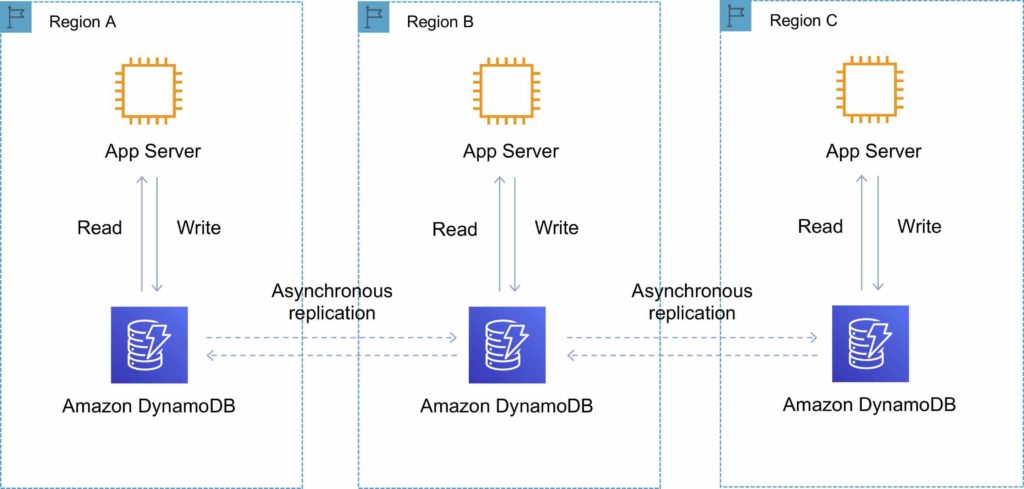
DynamoDB global tables are ideal for massively scaled applications, with globally dispersed users.

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.

Definitions:

* **A *global table*** is a collection of one or more replica tables, all owned by a single AWS account.
* **A *replica table*** (or *replica*, for short) is a single DynamoDB table that functions as a part of a global table. Each replica stores the same set of data items. Any given global table can only have one replica table per region.

The following diagram depicts the Amazon DynamoDB Global Tables topology:



You can add replica tables to the global table, so that it can be available in additional AWS regions.

With a global table, each replica table stores the same set of data items. DynamoDB does not support partial replication of only some of the items.

An application can read and write data to any replica table. If your application only uses eventually consistent reads, and only issues reads against one AWS region, then it will work without any modification.

However, if your application requires strongly consistent reads, then it must perform all its strongly consistent reads and writes in the same region. DynamoDB does not support strongly consistent reads across AWS regions.

It is important that each replica table and secondary index in your global table has identical write capacity settings to ensure proper replication of data.

**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.

How Application Auto Scaling works:

* 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.
* Uses a *target tracking* algorithm to adjust the provisioned throughput of the table (or index) upward or downward in response to actual workloads, so that the actual capacity utilization remains at or near your target utilization.

Currently, Auto Scaling does not scale down your provisioned capacity if your table’s consumed capacity becomes zero.

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

**High Availability Approaches for Databases**

If possible, choose DynamoDB over RDS because of inherent fault tolerance.

If DynamoDB can’t be used, choose Aurora because of redundancy and automatic recovery features.

If Aurora can’t be used, choose Multi-AZ RDS.

Frequent RDS snapshots can protect against data corruption or failure, and they won’t impact performance of Multi-AZ deployment.

Regional replication is also an option but will not be strongly consistent.

If the database runs on EC2, you must design the HA yourself.

<https://digitalcloud.training/aws-content-delivery-and-dns-services/>

<https://digitalcloud.training/amazon-redshift/>

<https://digitalcloud.training/aws-cloud-management-services/>