AWS Database Blog

Choosing the Right DynamoDB Partition Key

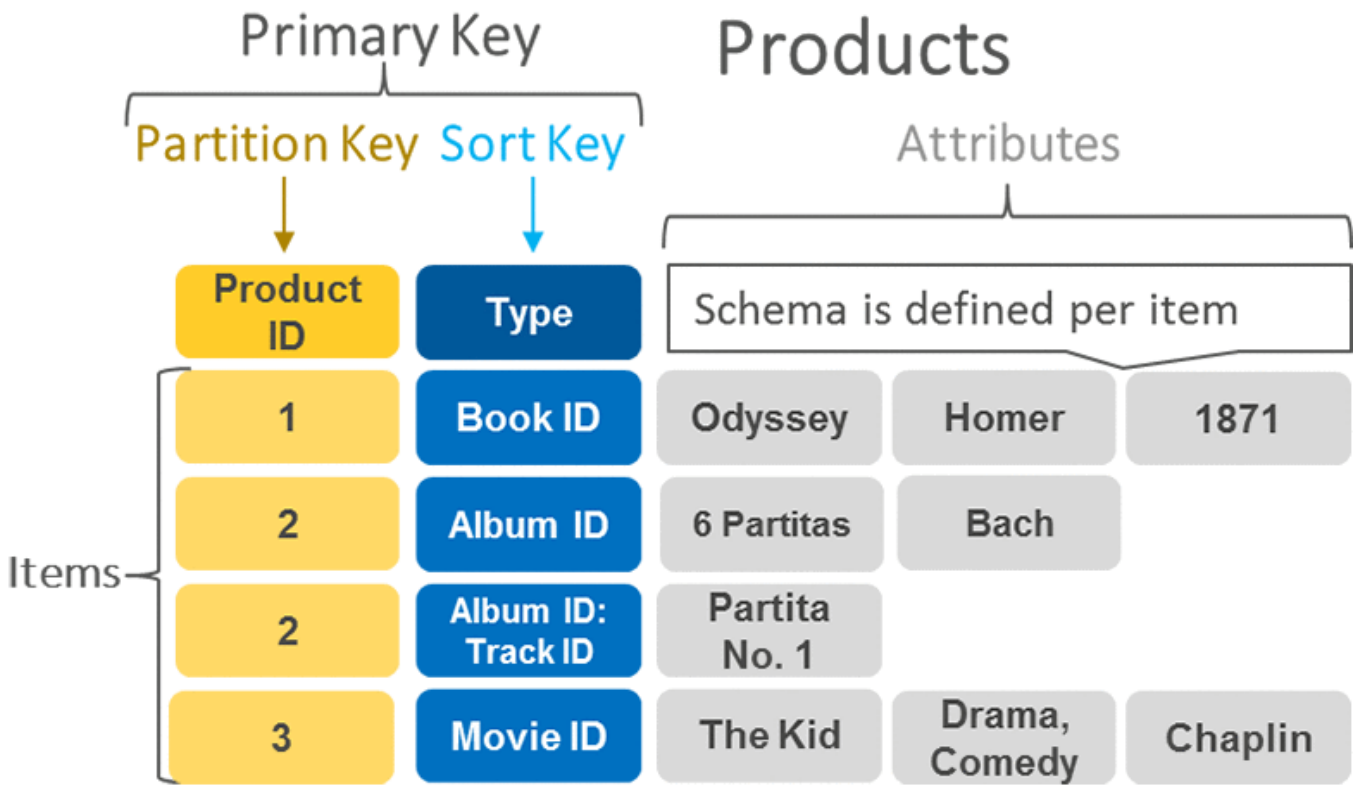
by Gowri Balasubramanian | on 20 FEB 2017 | in Amazon DynamoDB, Database | Permalink | Comments | Share

This blog post covers important considerations and strategies for choosing the right partition key for designing a schema that uses Amazon DynamoDB. Choosing the right partition key is an important step in the design and building of scalable and reliable applications on top of DynamoDB.

What is a partition key?

DynamoDB supports two types of primary keys:

- Partition key: A simple primary key, composed of one attribute known as the *partition key*. Attributes in DynamoDB are similar in many ways to fields or columns in other database systems.
- Partition key and sort key: Referred to as a *composite primary key*, this type of key is composed of two attributes. The first attribute is the *partition key*, and the second attribute is the *sort key*. Following is an example.

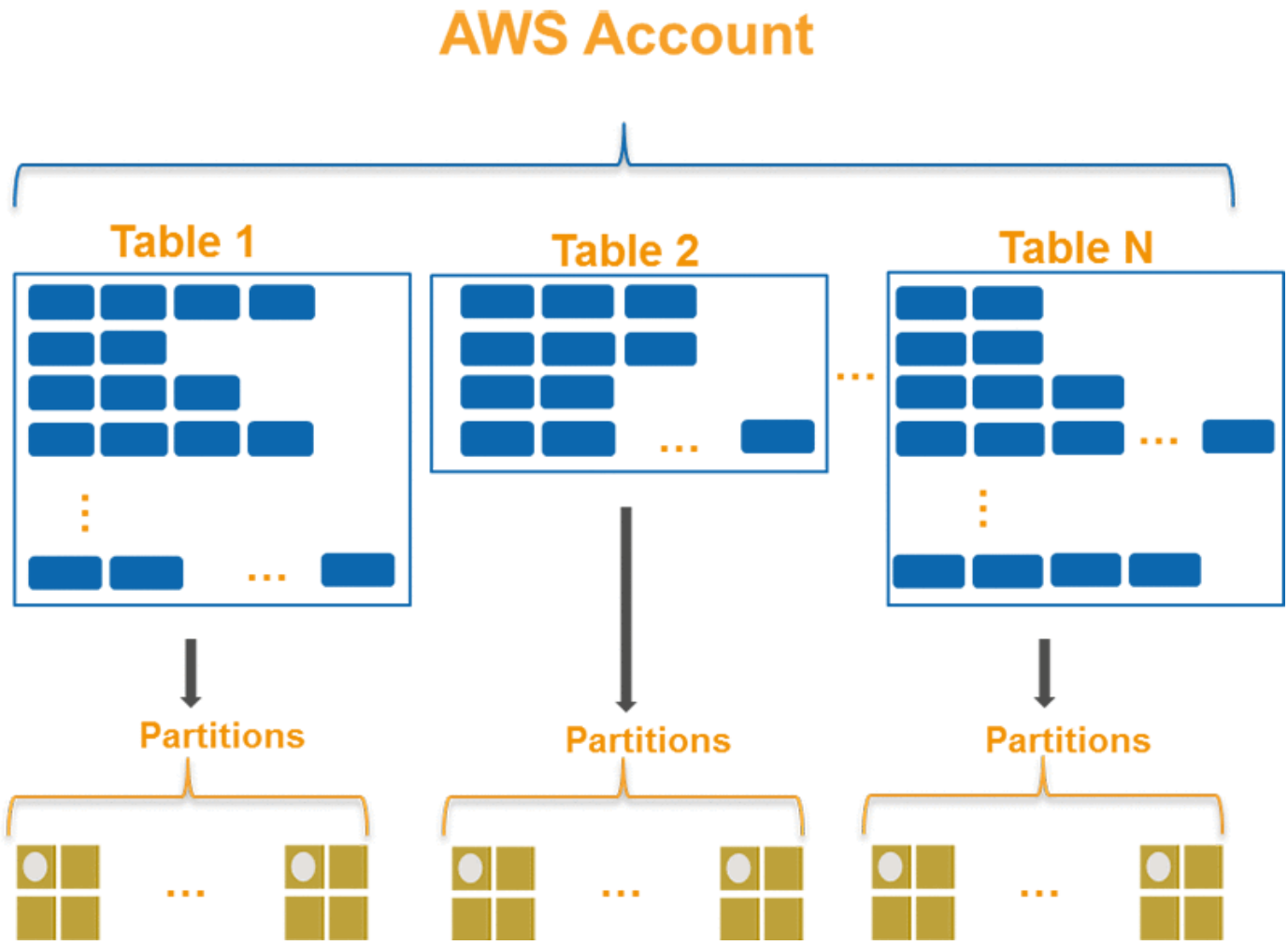


Why do I need a partition key?

DynamoDB stores data as groups of attributes, known as *items.* Items are similar to rows or records in other database systems. DynamoDB stores and retrieves each item based on the primary key value, which must be unique. Items are distributed across 10-GB storage units, called partitions (physical storage internal to DynamoDB). Each table has one or more partitions, as shown in the following illustration. For more information, see Partitions and Data Distribution in the *DynamoDB Developer Guide.*

DynamoDB uses the partition key's value as an input to an internal hash function. The output from the hash function determines the partition in which the item is stored. Each item's location is determined by the hash value of its partition key.

All items with the same partition key are stored together, and for composite partition keys, are ordered by the sort key value. DynamoDB splits partitions by sort key if the collection size grows bigger than 10 GB.



Partition keys and request throttling

DynamoDB evenly distributes provisioned throughput—read capacity units (RCUs) and write capacity units (WCUs)—among partitions and automatically supports your access patterns using the throughput you have provisioned. However, if your access pattern exceeds 3000 RCU or 1000 WCU for a single partition key value, your requests might be throttled with a `ProvisionedThroughputExceededException` error.

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

To avoid request throttling, design your DynamoDB table with the right partition key to meet your access requirements and provide even distribution of data.

Recommendations for partition keys

Use high-cardinality attributes. These are attributes that have distinct values for each item, like `e-mailid`, `employee_no`, `customerid`, `sessionid`, `orderid`, and so on.

Use composite attributes. Try to combine more than one attribute to form a unique key, if that meets your access pattern. For example, consider an orders table with `customerid+productid+countrycode` as the partition key and `order_date` as the sort key.

Cache the popular items when there is a high volume of read traffic using Amazon DynamoDB Accelerator (DAX). The cache acts as a low-pass filter, preventing reads of unusually popular items from swamping partitions. For example, consider a table that has deals information for products. Some deals are expected to be more popular than others during major sale events like Black Friday or Cyber Monday. DAX is a fully managed, in-memory cache for DynamoDB that doesn't require developers to manage cache invalidation, data population, or cluster management. DAX also is compatible with DynamoDB API calls, so developers can incorporate it more easily into existing applications.

Add random numbers or digits from a predetermined range for write-heavy use cases. Suppose that you expect a large volume of writes for a partition key (for example, greater than 1000 1 K writes per second). In this case, use an additional prefix or suffix (a fixed number from predetermined range, say 1–10) and add it to the partition key.

For example, consider a table of invoice transactions. A single invoice can contain thousands of transactions per client. How do we enforce uniqueness and ability to query and update the invoice details for high-volumetric clients?

Following is the recommended table layout for this scenario:

- Partition key: Add a random suffix (1–10 or 1–100) with the `InvoiceNumber`, depending on the number of transactions per `InvoiceNumber`. For example, assume that a single `InvoiceNumber` contains up to 50,000 1K items and that you expect 5000 writes per second. In this case, you can use the following formula to estimate the suffix range: (Number of writes per second * (roundup (item size in KB),0)* 1KB) /1000). Using this formula requires a minimum of five partitions to distribute writes, and hence you might want to set the range as 1-5.
- Sort key: `ClientTransactionid`

Partition Key	**Sort Key**	**Attribute1**
InvoiceNumber+Randomsuffix	ClientTransactionid	Invoice_Date
121212-1	Client1_trans1	2016-05-17 01.36.45
121212-1	Client1-trans2	2016-05-18 01.36.30

121212-2	Client2_trans1	2016-06-15 01.36.20
121212-2	Client2_trans2	2016-07-1 01.36.15

- This combination gives us a good spread through the partitions. You can use the sort key to filter for a specific client (for example, where `InvoiceNumber=121212-1` and `ClientTransactionid` begins with `Client1`).
- Because we have a random number appended to our partition key (1–5), we need to query the table five times for a given `InvoiceNumber`. Our partition key could be 121212-[1-5], so we need to query where partition key is 121212-1 and `ClientTransactionid` begins_with Client1. We need to repeat this for 121212-2, on up to 121212-5 and then merge the results.

Note:

After the suffix range is decided, there is no easy way to further spread the data because suffix modifications also require application-level changes. Therefore, consider how hot each partition key might get and add enough of a random suffix (with buffer) to accommodate the anticipated future growth.

This option induces additional latency for reads due to X number of read requests per query.

As mentioned in the DynamoDB documentation, a randomizing strategy can greatly improve write throughput. But it's difficult to read a specific item because you don't know which suffix value was used when writing the item.

To make it easier to read individual items, consider sharding by using calculated suffixes, as explained in Using Write Sharding to Distribute Workloads Evenly in the *DynamoDB Developer Guide*. For example, suppose that a large number of invoice transactions are being processed but the read pattern is to retrieve small number of items for a particular `sourceid` by date range. In this case, it's more effective to distribute the items across a range of partitions using a particular attribute, in this case `sourceid`. You can hash the `sourceId` to annotate the partition key rather than using random number strategy. This way, you know which partition to query and retrieve the results from.

As with tables, we recommend that you consider a sharding approach for global secondary indexes if you are anticipating a hot key scenario with a global secondary index partition_key.

For example, consider the following schema layout of an `InvoiceTransaction` table. It has a header row for each invoice and contains attributes such as total amount due and transaction_country, which are unique for each invoice. Assuming we need to find the list of invoices issued for each transaction country, we can create a global secondary index with `partition_key` as `trans_country`. However, this approach leads to a hot key write scenario, because the number of invoices per country are unevenly distributed.

The following table shows the recommended layout with a sharding approach.

Table **Partition Key**	**Table** **Sort Key**	**Attribute1**	Attribute2 **GSI** **Partition_Key**	Attribute3 GSI **Sort Key**	Attribute4	Attribute5

InvoiceNumber	Sort_key attribute	Invoice_Date	Random prefix range	Trans_country	Amount_Due	Currency
121212	head	2018-05-17 T1	Random (1-N)	USA	10000	USD
121213	head	2018-04-1 T2	Random (1-N)	USA	500000	USD
121214	head	2018-04-1 T2	Random (1-N)	FRA	500000	EUR

Following is the global secondary index (GSI) for the preceding scenario.

GSI **Partition Key**	**GSI** **Sort Key** **Trans_country**	Projected Attributes	
(Random range)	Trans_country	Invoice_Number	Other Data attributes
1-N	USA	121212	
1-N	USA	121213	
1-N	FRA	121214	

In the preceding example, you might want to identify the list of invoice numbers associated with the USA. In this case, you can issue a query to the global secondary index with `partition_key = (1-N)` and `trans_country = USA`.

Antipatterns for partition keys

Use sequences or unique IDs generated by the DB engine as the partition key, especially when you are migrating from relational databases. It's common to use sequences (`schema.sequence.NEXTVAL`) as the primary key to enforce uniqueness in Oracle tables. Sequences are not usually used for accessing the data.

The following is an example schema layout for an order table that has been migrated from Oracle to DynamoDB. The main table partition key (`TransactionID`) is populated by a UID. A GSI is created on `OrderID` and `Order_Date` for query purposes.

Partition key	Attribute1	Attribute2
TransactionID	OrderID	Order_Date
1111111	Customer1-1	2016-05-17 01.36.45
1111112	Customer1-2	2016-05-18 01.36.30
1111113	Customer2-1	2016-05-18 01.36.30

Following are the potential issues with this approach:

- You can't use `TransactionID` for any query purposes, so you lose the ability to use the partition key to perform a fast lookup of data.
- GSIs support eventual consistency only, with additional costs for reads and writes.

Note: You can use the conditional writes feature instead of sequences to enforce uniqueness and prevent the overwriting of an item.

Using low-cardinality attributes like product_code as the partition key and order_date as the sort key greatly increases the likelihood of hot partition issues. For example, if one product is more popular, then the reads and writes for that key is high, resulting in throttling issues.

Except for `scan`, DynamoDB API operations require an equal operator (EQ) on the partition key for tables and GSIs. As a result, the partition key must be something that is easily queried by your application with a simple lookup. An example is using `key=value`, which returns either a unique item or fewer items. There is a 1-MB limit on items that you can fetch through a single query operation, which means that you need to paginate using `LastEvaluatedKey`, which is not optimal.

In short: Do not lift and shift primary keys from the source database without analyzing the data model and access patterns of the target DynamoDB table.

Conclusion

When it comes to DynamoDB partition key strategies, no single solution fits all use cases. You should evaluate various approaches based on your data ingestion and access pattern, then choose the most appropriate key with the least probability of hitting throttling issues. Along with the best partition key design, DynamoDB adaptive capacity can protect your application from throttling issues against an uneven data access pattern.

For further guidance on schema design for various scenarios, see NoSQL Design for DynamoDB in the *DynamoDB Developer Guide.*

About the Author

Gowri Balasubramanian is a senior solutions architect at Amazon Web Services. He works with AWS customers to provide guidance and technical assistance on both relational as well as NoSQL database services, helping them improve the value of their solutions when using AWS.