DynamoDB Design Patterns for Single Table Design

Designing DynamoDB data models with single table design patterns can unlock its potential of unlimited scalability and performance for a very low price.

Saturday, November 7, 2020

DynamoDB is the beast that combined with the right design patterns unlocks the potential of unlimited scalability, performance, and low price. However, designing DynamoDB data models can be tricky. Its feature set is limited, but with the correct design patterns, mainly based on prevailed single table design, you can cover almost all use cases that are expected from an OLTP system.

Do you need temporary or permanent help on your project?

There are multiple resources about designing DynamoDB that did not exist a few years ago. However, I still could not find a short and concise list of design patterns, hence the need for this article. The intention here is not to explain in detail how DynamoDB works, but please see the resources section for more information if needed.

RDBMS databases (SQL databases) were designed at a time when data storage was expensive but processing power was relatively cheap. You save costs with data normalization (removing duplicated data, etc.). But when reading the data, you need to denormalize. This normalization and denormalization require computing power. The only way to scale up the RDBMS database, except with sharding and replication, is vertical, putting it in a larger machine.

Nowadays, we store incomparably more data. Storage is cheap. Correlating to data-size processing power is expensive. With a lot of data, vertical scaling is no longer an option. You have to split data and load it onto multiple computers. That is why we needed a paradigm shift from RDBMS databases. That came with the key-value store and document databases as a variant of NoSQL databases. DynamoDB is one of the most established solutions in that space.

SQL	NoSQL
Optimize for storage	Optimize for compute
Normalized/relational	Denormalized/hierarchical
Ad hoc queries	Scale horizontally
Good for OLAP	Build for OLTP at scale

Designing #DynamoDB data models with single table design patterns can unlock its potential of unlimited scalability and performance for a very low price.

CLICK TO TWEET

DynamoDB is a hyper scalable, performant, and afordable managed NoSQL database. It should be the first choice for serverless and all solutions that demand scalability. However, because of scalability requirements, like all NoSQL databases, it lacks features common in RDBMS databases. Designing data models is quite difficult and goes against established principles of designing RDBMS databases. That is why it is even more difficult for beginners.

Designing #DynamoDB data models goes against established principles of designing RDBMS databases.

CLICK TO TWEET

DynamoDB also has fewer features than some other NoSQL databases. That is for a reason. It provides only features that are scalable, in contrast to other databases like MongoDB, which is feature richer.

Because of limited functionality, you should think twice if DynamoDB is the right choice for you if:

- your database is not enormous or can be sharded,
- it requires a complex data mode,
- it requires future reach ad hoc query language (SQL)

You should think twice when choosing DynamoDB if: 1) your database is not enormous or can be sharded 2) it requires a complex data mode 3 it requires future reach ad hoc query language (SQL)

CLICK TO TWEET

Refresher on How DynamoDB Works

- **Partition (hash) key (PK)**
  Defines in which portion the data is stored. Use as a distinct value as possible. You can only query by the exact value. UUID is commonly used.
- **Sort (range) key (SK)**
  Defines the sorting of items in the partition. It is optional. Combined with the partition key, it defines the primary key of the item. You can query it by condition expression (`=, >, <, between, begins_with, contains, in`) combined with PK. You can never leave out PK when querying.
- **Local secondary index (LSI)**
  Allows defining another sort key for the same partition key. It can only be created when you create the table. Compared to the global secondary index, it offers consistent reads. You can have up to five LSI on a table.
- **Global secondary index (GSI)**
  Allows defining a new partition and optional sort key. It should be preferred compared to the LSI, except when you need consistent reads. You can have up to 20 GSI on a table, so you would try to reuse them within the same table. GSI is the central part of most design patterns in a single table design. Both LSI and GSI are implemented as copying all the data to a new table-like structure. You have projections that enable you to copy only the data that you need.
- **Attributes can be scalar (single value) or complex (list, maps, sets of unique values).**

Global secondary index is the central part of most design patterns in a #DynamoDB single table design.

CLICK TO TWEET

DynamoDB query can return of maximum of 1 MB results. You can retrieve the additional records with pagination by specifying the last read item from the previous one (`LastEvaluatedKey` property). You can also set a limit on how many records you want to retrieve (`Limit` property).

DynamoDB has supported transactions since late 2018. This is an essential feature of many patterns. However, transactions are not what you are used to in RDBMS databases. You cannot start the transaction from the code, do some interactions with the database, and commit or rollback. Each transaction can include up to 25 unique items or up to 4 MB of data, including conditions. You send data to the database the same way as you use batch. Operations are exceeded according to ACID principals.

Although transactions are quite limiting, you need it just to form some patterns. They are not commonly used like in SQL databases because you do not need them that much. The principal in RDBMS databases is to split every complex item into multiple tables. To do that safely, you need transactions. But, in DynamoDB, you usually do not split items, and do not need transactions for that purpose.

DynamoDB has many limits that prevent you from making unscalable solutions. Here are the most important ones:

- Item size: max. 400 KB item
- Size of the result set for query and scan: max. 1 MB. The limit is applied before filter and projection. You can retrieve all records with subsequent requests.
- Records with the same partition key: max. 10 GB. That includes LSI (but not GSI).
- Partition throughput: 1000 WCU, 3000 RSU
- Transactions: up to 25 unique items or up to 4 MB of data, including conditions.
- Batch write: max. 25 items, max. 16 MB
- Batch get item: max. 100 items, max. 16 MB

Principals of Data Modeling

- **Draw the ER diagram as you usually do.**
- **Get all access patterns before starting data modeling.**
  This is really hard with a new project, but extremely important. The data model is tied to service and its access patterns. You must also assess the volume of the data because that also influences the patterns that you use.
- **Denormalization.**
  In contrary to RDBMS databases, you are storing data in a way that you can read it as quickly as possible. Duplicating data is standard practice. Taking care of consistency is your job unless you are using secondary indexes. Secondary indexes are not indexes in the standard sense. They are actually copy data, but you do not have to manage that.
- **Avoid hot partition.**
  Hot partition occurs when you have a lot of requests that are targeted to only one partition. In 2018, AWS introduced adaptive capacity, which reduced the problem, but it still very much exists. To avoid hot partition, you should not use the same partition key for a lot of data and access the same key too many times. The partition key defines partition, and if you request data mainly from one partition, you do not distribute the load. The partition key should be as distinct as possible.
- **Do not use scan on large datasets.**
  Scan means reading all the data from the table. It can be used when migrating a data model or when you actually have to process all the records. You can limit the data with a sparse index pattern, where scans are also commonly used.
- **Do not use a filter on large datasets.**
  A filter is applied after the data is read from the database, so you are throwing away all the work you just did. Another reason to not use a filter is the general limitation of DynamoDB that you can only read up to 1 MB of data with each request. From this dataset, DynamoDB removes the date that does not fit the filter. That means your query can return zero records, although there would be records you searched for in the database. You can retrieve them with paging, but you have to be aware of that behavior.
- **Prefers eventually consistent reads.**
  That is the default and the only option for reading from GSI. To avoid the need for strong consistency reads, do not read data immediately after you wrote it. You already have it, so you should not do that anyway, and now you have another reason not to do it.
- **The data model is hard to change.**
  This is contrary to intuition because DynamoDB does not enforce data schema. It is simple to add some attributes but very hard to change relations to adapt to new access patterns. In most cases, you have to scan the whole database and restructure each record.
- **Prepare for data model change.**
  Because you cannot avoid all the data model changes, prepare a system to reformat the data structure to adopt a new date model. That way, you are not constrained too much when the new requirements come.

- **Avoid transactions.**
  Transactions perform more reads and writes than normal operations. They are more limiting than in RDBMS databases, but you do not need them that much. Do not use transactions to maintain a normalized data model.

Principals of #DynamoDB data modeling: draw ER, GET ALL ACCESS PATTERNS, denormalize, avoid scans, filters, transactions, prefers eventually consistent reads, learn #singletabledesign

CLICK TO TWEET

There are many options on how to design a data model for DynamoDB. Different developers will end up with a different result for the same task. That is contrary to modeling for RDBMS databases, where in most cases, they will end up with very similar results.

RDBMS data modeling	DynamoDB
Access patterns can be defined later	Access patterns must be defined before data modeling
Normalization	Denormalization
Powerful ad hoc queries (SQL)	Limiting query capabilities
Powerful transactions	Limiting transactions
Reasonably flexible to change the data model	Hard to change the data model

Do you need temporary or permanent help on your project?

Migrating Data Model

There are a lot of approaches on how to change the data model. The most common is to scan all elements and write them in a new structure. To speed up transformation, you can use DynamoDB parallel scans, which enables you to transform with multiple workers.

There is another more straightforward solution. You can use scalable serverless infrastructure to do that. You configure streams on the old table that writes to a new one. You read only the keys of items in the old table with dedicated GSI and then write them back with the migration flag. The stream will take the item, spin an appropriate number of Lambdas, and write restructured data. You can find a detailed description here. The solution moves data to a new table. If you want to keep them in the same table when storing the restructured item, you will trigger the stream again, which is not resource-wise. Make sure you process only unprocessed data, so you do not spin an infinitive loop.

When restructuring the data, make sure you configure enough capacity units on DynamoDB tables. If you use streams and Lambda, pay attention to the maximum concurrent executions of Lambdas in your account.

Principals of Single Table Design

When we are designing a data model for DynamoDB, it is recommended to use a single table design because:
- It offers a rich set of flexible and efficient design patterns,
- Better utilization of resources,
- Easer configuration, capacity planning, and monitoring.

The downside of a single table design is that data looks confusing and hard to read without proper documentation.

Why #DynamoDB #singletabledesign: rich set of flexible and efficient design patterns, better utilization of resources, easer configuration, capacity planning, and monitoring.

CLICK TO TWEET

Principals:

- **Put all data in one table or as few as possible.**
- **Reuse/overload keys and secondary indexes for different purposes.**
- **Name keys and secondary indexes with uniform names**, like PK (partition key), SK (sort key), GS1PK (first global secondary index partition key), GS1SK (first global secondary index sort key).
- **Identify the type of the item by prefixing keys with type**, like PS: USER#123 (USER = type, 123 = id).
- **Duplicate and separate values for keys, indexes from actual application attributes.** That means that you can have the same values in two, four, or even more places.
- **Add a special attribute to define the type**, so the data is more readable.
- **Do not store too much data in one record because of the hard limit of 400 KB**, and you also cannot read just part of it (projections are applied after reading).
- **Use the partition key that is actually used in access patterns.** For example, for the user, use username or email, not some random ID. This way, you take advantage of the primary index, which otherwise would be unused.

Principals of #DynamoDB #singletabledesign: one table, reuse keys and secondary indexes, prefixing keys with type, duplicate and separate values for keys&indexes from application attributes

CLICK TO TWEET

Denormalization Patterns

Normalization is a standard technique in designing the RDBMS database model for organizing data that prevents duplication and enforces consistency. This is achieved by splitting data into multiple tables. Reading this requires joins, which you do not have in DynamoDB.

DynamoDB is designed to hold a large amount of data. That is why data is partitioned. When you request the data, you do not want to spend time and compute power to gather data from various tables. That is why DynamoDB does not have joins. The solution is to store data in a form that is already prepared for our access patterns. This is usually done by denormalization and duplication. Even secondary indexes are by design data duplication. They do not just point to the data; they contain whole or part of the data, depending on the projection that you have configured.

Denormalization with a Complex Attribute Pattern

The most simple denormalization is to contain all the data in one item.

Example:
The partition key contains the type of entity (USER) and user ID, which in this case, is email. Why email, not UUID, or similar? Because the user will log in with their email, and we will access data by email. The address is stored as a complex attribute and is not separated into another table or record. There is an additional attribute TYPE that defines the type. In the same table, we will store multiple types. The attribute type does not have a special purpose but can be useful when processing the records.

PK	TYPE	Name	Address	
USER#jake@gmail.com	USER	Jake Doug	**{ "street": "4283 Hinkle Deegan Lake Road", "city": "Syracuse", "state": "NY", "zip": "13202" }**	...
USER#milo@gmail.com	USER	Milo Nick	**{ "street": "3305 Broadway Avenue", "city": "Dayton", "state": "TN", "zip": "37321" }**	...

If you usually do not need all the data, it is better to share it in another item and keep only the summary if needed.

Denormalization by Duplicating Data Pattern

The solution is the same as in the previous case. You have an item that also contains related data. But in this case, this data is related to other items, so it needs to be duplicated.

If the data is immutable (does not change), there is no problem. But if the data can change, you have to take care of consistency by updating each record when data changes.

Example:

PK	TYPE	Name	Author	
BOOK#HAMLET	BOOK	Hamlet	**William Shakespeare**	...
BOOK#ROMEO_AND_JULIET	BOOK	Romeo and Juliet	**William Shakespeare**	...
BOOK#WAR_AND_PEACE	BOOK	War and Peace	**Leo Tolstoy**	...

The partition key contains type and book ID. For book ID, we choose the book's URL-friendly name. The author's name is duplicated, but that is not a problem because it will never change.

Relationship Patterns

Most patterns for modeling a relationship take advantage of the following principals:

- Collocate related data by having **the same partition key and different sort key** to separate it.
- **Sort key allows searching so you can limit which related data you want to read** and how much of them (e.g., first 10 records)
- **Swap partition and sort key in GSI so you can query the opposite direction of the relationship.**

Knowing these patterns is an eye-opener, and it becomes clear to you that DynamoDB is not just a simple featureless key-value store and can be used in almost any use case.

Patterns that take advantage of the sharing partition key enable you to read related data with just one query, and because it is collocated on the same partition, reads are very fast. But this approach has an obvious downside. If you are using one partition too much, you are not distributing the load. The same problem occurs in GSI if too much data shares a partition key. You can also hit a 10 GB limit on one partition key, but that does not apply to GSI. But to hit that problem, your data size and load should be significant, so you should not over-engineer the data model. To avoid the problem, see Optimizing Partition Management—Avoiding Hot Partitions

The sort key enables a lot of the following patterns. With the sort key, we can filter the data. For example, filtering by date is very common. You usually store the date in ISO format like 2020-07-16T19:20:30. This format is sortable, and data can also be filtered by the period you need (2020-07=filter by July). A common mistake is ignoring the fact that sorting strings is not strictly lexicographical. Uppercase letters come before lowercase, and numbers and symbols have their own positions.

#DynamoDB sort key does not sort lexicographically 😲🤯👹

CLICK TO TWEET

If you need to put ID in a sort key, consider using KSUID (K-Sortable Unique Identifier) instead of the commonly used UUID. KSUID is a 27 characters long string that combines a timestamp and an additional random key. The timestamp part allows sorting. An additional key is just to make sure the same key is deduplicated in some rare scenario.

Use KSUID to have sortable unique ID as replacment of UUID in #DynamoDB #singletabledesign

CLICK TO TWEET

Take note that data in the sort key of the primary index cannot be changed. If you have data that can change (e.g., customer last visit) and want to take advantage of features of the sort key, you can use them on the secondary index.

One to Many Relationship Pattern

All the related records have the ID of the parent item in the partition key. The ID of each item is stored in the sort key. For the parent item, it is the same as the partition key.

Example:

PK	SK	TYPE	Customer ID	Order ID	Name	
CUSTOMER#XYQ	CUSTOMER#XYQ	CUSTOMER	XYQ		Tom	...
CUSTOMER#XYQ	**ORDER#00001**	**ORDER**		**00001**		**...**
CUSTOMER#XYQ	**ORDER#00002**	**ORDER**		**00002**		**...**
CUSTOMER#VLD	CUSTOMER#VLD	CUSTOMER	VLD		Linda	...
CUSTOMER#VLD	**ORDER#00003**	**ORDER**		**00003**		**...**
CUSTOMER#VLD	**ORDER#00004**	**ORDER**		**00004**		**...**

This allows you to:
- Read customer by ID (`PK = CUSTOMER#XYQ, SK = CUSTOMER#XYQ`)
- Read customer and last 10 orders together (`PK = CUSTOMER#XYQ, Limit = 11`)
- Read only orders of the customer (`PK = CUSTOMER#XYQ, SK = begins_with("ORDER")`)
- Read only one order (`PK = CUSTOMER#XYQ, SK= ORDER#00001`)

Processing each order results in multiple logs. Here is how we can extend the previous example with the one to many relationship between order and logs with help of GSI:

PK	SK	TYPE	Name	Customer ID	Order ID	Log ID	GSI1PK
CUSTOMER#XYQ	CUSTOMER#XYQ	CUSTOMER	Tom	XYQ			
CUSTOMER#XYQ	ORDER#00001	ORDER			00001		CUSTOMER#XYQ#ORDER#00001
LOG#00001	**LOG#00001**	**LOG**				**00001**	**CUSTOMER#XYQ#ORDER#00001**
LOG#00002	**LOG#00002**	**LOG**				**00002**	**CUSTOMER#XYQ#ORDER#00001**
CUSTOMER#XYQ	ORDER#00002	ORDER			00002		CUSTOMER#XYQ#ORDER#00002
CUSTOMER#VLD	CUSTOMER#VLD	CUSTOMER	Linda	VLD			
CUSTOMER#VLD	ORDER#00003	ORDER			00003		CUSTOMER#VLD#ORDER#00003
CUSTOMER#VLD	ORDER#00004	ORDER			00004		CUSTOMER#VLD#ORDER#00004

Let's view the same data from the perspective of GSI:

GSI1PK	GSI1SK	TYPE	Order ID	Log ID	
CUSTOMER#XYQ#ORDER#00001	ORDER#00001	ORDER	00001		...
CUSTOMER#XYQ#ORDER#00001	**LOG#00001**	**LOG**		**00001**	**...**
CUSTOMER#XYQ#ORDER#00001	**LOG#00002**	**LOG**		**00002**	**...**
CUSTOMER#XYQ#ORDER#00002	ORDER#00002	ORDER	00002		...
CUSTOMER#VLD#ORDER#00003	ORDER#00003	ORDER	00003		...
CUSTOMER#VLD#ORDER#00004	ORDER#00004	ORDER	00004		...

This enables us to read the order with all their logs.

A customer can send many questions to the support team. Customers and questions are again a one to many relationship. What if you want to query both by sort order. That could be a problem. For that reason, we subtract Question ID from a very large number.

PK	SK	TYPE	Customer ID	Question ID	Order ID	Name	
CUSTOMER#XYQ	**#QUESTION#99998**	**QUESTION**		**00002**			**...**
CUSTOMER#XYQ	**#QUESTION#99999**	**QUESTION**		**00001**			**...**
CUSTOMER#XYQ	CUSTOMER#XYQ	CUSTOMER	XYQ			Tom	...
CUSTOMER#XYQ	ORDER#00001	ORDER			00001		...
CUSTOMER#XYQ	ORDER#00002	ORDER			00002		...

We used another trick to make this work. We added # before QUESTION, so the questions come before the customer.

This allows following queries:

- Read customer by ID (`PK = CUSTOMER#XYQ, SK = CUSTOMER#XYQ`)
- Read customer and last 10 orders together (`PK = CUSTOMER#XYQ, SK >= CUSTOMER#XYQ, Limit = 11`)
- Read customer and last 10 question together (`PK = CUSTOMER#XYQ, SK <= CUSTOMER#XYQ, Limit = 11, ScanIndexForward: false`)
- Read last 10 questions (`PK = CUSTOMER#XYQ, SK = begins_with("#QUESTION"), Limit = 10, ScanIndexForward: false`)
- Read customer, orders, and questions with one request (`PK = CUSTOMER#XYQ`)

Many to Many Relationship Patterns

Many to Many Relationship with Adjacency List (Duplicities for Relationship)

Each entity is stored independently with its own partition key. Relationships are stored as an additional item with the partition key's value as one part of the relationship and the sort key as the other part. In those additional items, you also duplicate essential data from related items. You invert PK and SK in GSI to access the other side of the relationship.

Example:

PK	SK	TYPE	StudentName	SportName	Coach	GSIPK1
STUDENT#XYQ	STUDENT#LKJ	STUDENT	Tom			STUDENT#LKJ
STUDENT#XYQ	**SPORT#BASKETBALL**	**STUDENT_SPORT**	**Tom**	**Basketball**		**SPORT#BASKET**
STUDENT#XYQ	**SPORT#FOOTBALL**	**STUDENT_SPORT**	**Tom**	**Football**		**SPORT#FOOTBA**
STUDENT#VLD	STUDENT#VLD	STUDENT	Linda			STUDENT#VLD
STUDENT#VLD	**SPORT#BASKETBALL**	**STUDENT_SPORT**	**Linda**	**Basketball**		**SPORT#BASKET**
STUDENT#VLD	**SPORT#TENNIS**	**STUDENT_SPORT**	**Linda**	**Tennis**		**SPORT#TENNIS**
SPORT#BASKETBALL	*SPORT#BASKETBALL*	*SPORT*		*Basketball*	*Simon*	*SPORT#BASKETE*
SPORT#FOOTBALL	*SPORT#FOOTBALL*	*SPORT*		*Football*	*James*	*SPORT#FOOTBAL*
SPORT#TENNIS	*SPORT#TENNIS*	*SPORT*		*Tennis*	*Susan*	*SPORT#TENNIS*

This allows you to:

- Read student by ID (`PK = STUDENT#XYQ, SK = STUDENT#XYQ`)
- Read student and their sports (`PK = STUDENT#XYQ`)
- Read only student sports (`PK = STUDENT#XYQ, SK = begins_with("SPORT")`)

We have GSI that flips PK and SK of the primary index. This enables us to see the other side of the relationship.

GSIPK1	GSISK1	TYPE	StudentName	SportName	Coach	
STUDENT#LKJ	STUDENT#XYQ	STUDENT	Tom			...
SPORT#BASKETBALL	**STUDENT#XYQ**	**STUDENT_SPORT**	**Tom**	**Basketball**		**...**
SPORT#FOOTBALL	**STUDENT#XYQ**	**STUDENT_SPORT**	**Tom**	**Football**		**...**
STUDENT#VLD	STUDENT#VLD	STUDENT	Linda			...
SPORT#BASKETBALL	**STUDENT#VLD**	**STUDENT_SPORT**	**Linda**	**Basketball**		**...**
SPORT#TENNIS	**STUDENT#VLD**	**STUDENT_SPORT**	**Linda**	**Tennis**		**...**
SPORT#BASKETBALL	*SPORT#BASKETBALL*	*SPORT*		*Basketball*	*Simon*	*...*
SPORT#FOOTBALL	*SPORT#FOOTBALL*	*SPORT*		*Football*	*James*	*...*
SPORT#TENNIS	*SPORT#TENNIS*	*SPORT*		*Tennis*	*Susan*	*...*

This allows you to:

- Read all the students that play a particular sport and details of that sport (`GSIPK1 = SPORT#BASKETBALL`)
- Read all the students that play a particular sport (`GSIPK1 = SPORT#BASKETBALL, SK = begins_with("STUDENT")`)

This pattern works best when data is immutable. In case data is not immutable, and there is a lot of related data that changes frequently, the pattern might not be sustainable because, with every update, you have to update too many records. In that case, try to include only immutable data (key, username, email) in records that store relationship. If you need more data after reading relationship records, you have to execute additional queries to retrieve detailed data. You can use `WithBatchGet` to read more of them in a single request.

Many to Many Relationship with Materialized Graph Pattern

Materialized graph pattern is not so commonly used. It is the evolution of the adjacency list to a more complex pattern. You do not store just relationships but also define the type of relationship and hierarchy of the data. It is useful when you want to store a graph structure, for example, for a modern social networking app. It is a combination of all the tricks described at the beginning of the relationship chapter. You store connections and type of connection between nodes. Then you play with sort keys and GSI to see the other part of the relationship and achieve other access patterns.

Example:

PK	SK	TYPE	Subject	Name	Address	GSI1PK
STUDENT#TOM	STUDENT#TOM	STUDENT		Tom		STUDENT#TOM
STUDENT#TOM	**TEACHER#SIMON#CLASS#MATH**	**CLASS**	**Math**	**Simon**		**TEACHER#SIMO**
STUDENT#TOM	**TEACHER#MICHAEL#CLASS#PHYSICS**	**CLASS**	**Physics**	**Michael**		**TEACHER#MICH**
STUDENT#TOM	**HOME#USA#CA#LOS_ANGELES**	**HOME**			**...**	**COUNTRY#USA**
TEACHER#SIMON	TEACHER#SIMON	TEACHER		Simon		TEACHER#SIMON
TEACHER#MICHAEL	TEACHER#MICHAEL	TEACHER		Michael		TEACHER#MICHA

Hierarchical Data Pattern (Composite Sort Key)

A sort key enables range queries. If you combine multiple attributes in the sort key, you can filter by some or all of them, but only in the order that values were combined. It works great if values have a limited set of potential values. The last value can be used to filter the range of values.

This is most useful in hierarchical data. See storing sales data per country/city/store:

PK	SK	TYPE	COUNTRY	CITY	STORE	DATE	Sales
SALE#USA	SAN_FRANCISCO#00235#2020-09-22	SALE	USA	San Francisco	00235	2020-09-22	...
SALE#USA	LOS_ANGELES#00316#2020-10-12	SALE	USA	Los Angeles	00316	2020-10-12	...
SALE#USA	SEATTLE#00110#2020-08-04	SALE	USA	Seattle	00110	2020-08-04	...
SALE#FRANCE	PARIS#00512#2020-15-15	SALE	FRANCE	Paris	00512	2020-15-15	...

You can get data for:

- country (`PK = SALE#USA`)
- city (`PK = SALE#USA, SK = begins_with("SAN_FRANCISCO")`)
- store (`PK = SALE#USA, SK = begins_with("SAN_FRANCISCO#00235")`)
- by year / month / day for store (`PK = SALE#USA, SK = begins_with("SAN_FRANCISCO#00235#2020-09")`)

If that particular combination does not fit your case, you can easily add a new hierarchy/composite sort key in the new GSI. Pay attention so that too many items do not share the same partition key.

Unique Field

The partition key is unique, which is enforced by the database. But you cannot have another attribute that the database would also impose to be unique. To have another unique field, create another record that has this filed as a partition key. When you create, update, or delete a primary record, you must maintain this additional record in a transaction. The additional record will not store any data. It will be used just to ensure the transaction will fail if two records with the same value exist.

A typical example is the user's username and email (TypeScript):

```
const docClient = new DocumentClient({ apiVersion: '2012-08-10' });
const result = docClient.transactWrite({
  TransactItems: [
    {
      Put: {
        TableName: 'data',
        Item: {
          PK: 'USER#johndoe',
          SK: 'USER#johndoe',
          Username: 'johndoe',
          FirstName: 'John'
        },
        ConditionExpression: 'attribute_not_exists(PK)'
      }
    },
    {
      Put: {
        TableName: 'data',
        Item: {
          PK: 'USEREMAIL#johndoe@gmail.com',
          SK: 'USEREMAIL#johndoe@gmail.com',
        },
        ConditionExpression: 'attribute_not_exists(PK)'
      }
    }]
});
result.on('extractError', (response: any) => {
  try {
    let cancellationReasons =
JSON.parse(response.httpResponse.body.toString()).CancellationReasons;
    console.error("Transaction fail reasons:", cancellationReasons);
  } catch (err) {
    console.error('Error extracting cancellation error', err);
  }
});
await result.promise();
```

Schedule Delete

With Time to Live (TTL), DynamoDB enables you to schedule the deletion of items. You select an attribute on the table as TTL and set the value of that attribute to the time you want the item to be deleted. This timestamp must be set in the Unix epoch time format. The downside of this feature is that it guarantees deletion within 48 hours. If you want to be sure that you only get data that has not passed TTL, you can combine with filter expression that will remove items that have passed the TTL timestamp.

Example (TypeScript):

```
const docClient = new DocumentClient({ apiVersion: '2012-08-10' });
const nowEpoch = Math.floor(Date.now() / 1000);
const result = await docClient.query({
  TableName: 'data',
  KeyConditionExpression: "#PK = :PK",
  FilterExpression: "#ttl <= :now",
  ExpressionAttributeNames: {
    "#PK": "PK",
    "#ttl": "ttl"
  },
  ExpressionAttributeValues: {
    ":PK": "USER#johndoe",
    ":now": nowEpoch
  }
}).promise();
```

The Auto-Incrementing Number for ID or Counting

For item identification (ID), most commonly, UUID is used. That is a random string that is long and subsequently unique enough that there is a guarantee it will not be repeated. Compared to the auto-incrementing number that is more commonly used in RDBMS databases, UUID can be generated on the client, so you do not depend on the database to retrieve it. Nevertheless, sometimes there is a requirement to have an auto-incrementing number.

This pattern requires having a special item (only one) in the database with counting as its whole purpose. You execute the request to increment the number in the update expression. You must set the return value `UPDATED_NEW` to retrieve the new number.

Example (TypeScript):

```
const docClient = new DocumentClient({ apiVersion: '2012-08-10' });
const result = await docClient.update({
  TableName: 'data',
  Key: {
    PK: 'AUTOINCREMENT',
    SK: 'AUTOINCREMENT'
  },
  UpdateExpression: "SET #number = #number + :incr",
  ExpressionAttributeNames: {
    "#number": "number",
  },
  ExpressionAttributeValues: {
    ":incr": 1
  },
  ReturnValues: 'UPDATED_NEW'
}).promise();
console.log(`Auto-increment ID: ${(result.Attributes as any).number}`);
```

If you need this number for the sort key, you would typically pad the number with zeroes so that they can be used for sorting.

Just counting is a pattern on its own. You cannot get the number of items in the table without reading the whole table. With this pattern, you count things when you insert them, but do not forget to decrease the number when you delete the data.

Example (TypeScript):

```
const docClient = new DocumentClient({ apiVersion: '2012-08-10' });
const result = docClient.transactWrite({
  TransactItems: [
    {
      Put: {
        Item: {
          PK: "POST#ABC",
          SK: "LIKE#john-doe"
        },
        TableName: "data",
        ConditionExpression: "attribute_not_exists(PK)"
      }
    },
    {
      Update: {
        Key: {
          PK: "POST#ABC",
          SK: "POST#ABC"
        },
        TableName: "data",
        ConditionExpression: "attribute_exists(PK)",
        UpdateExpression: "SET #likeCount = #likeCount + :incr",
        ExpressionAttributeNames: {
          "#likeCount": "likeCount"
        },
        ExpressionAttributeValues: {
          ":incr": 1
        }
      }
    }
  ]
});
result.on('extractError', (response: any) => {
  try {
    let cancellationReasons =
JSON.parse(response.httpResponse.body.toString()).CancellationReasons;
    console.error("Transaction fail reason:", cancellationReasons);
  } catch (err) {
    console.error('Error extracting cancellation error', err);
  }
});
await result.promise();
```

Sparse Index

In the single table design, you try to reuse indexes. However, the sparse index pattern is an exception. The point is to have values in this index just to separate/filter them from the rest of the data. You set the value only to a portion of the data.

A typical example is to separate by entity type or other characteristics of the entity. For those entities, you would set the value of this GSI and leave it empty for all the others. But why would we need a separate index for that? Why not reuse the index as in other patterns and set type as the partition key? Because the partition key should be as distinct as possible. All the items would end up in the same partition.

For the partition key of the sparse index, you would use entity ID or any other value that is distinct enough. You can use a scan to read all the data, which is acceptable because the sparse index is meant for filtering and usually contains only a small subset of the data. You can also combine with the sort key and use any other pattern described in this article.

The sparse index also works if you set the partition key of GSI and not the sort key. Items without a sort key are not stored in the index if the sort key is defined. You can then use the sort key for additional filtering.

Example:

PK	STATUS	CUSTOMER	SHIP_DATE	SPARSE_SHIPPED_PK	SPARSE_SHIPPED_SK
ORDER#00001	AWAITING_PAYMENT	CUSTOMER#KHJ			

PK	STATUS	CUSTOMER	SHIP_DATE	SPARSE_SHIPPED_PK	SPARSE_SHIPPED_SK
ORDER#00002	AWAITING_SHIPMENT	CUSTOMER#JHD			
ORDER#00003	SHIPPED	CUSTOMER#JHD	2020-10-26T09:39:14	CUSTOMER#JHD	2020-10-26T09:39:14

We created a sparse index named SPARSE_SHIPPED. We put values of the primary key and sort key only for shipped orders. We put customer ID in PK and ship date in SK, which enables us additional uses of this index.

This allows you to:

- Read all shipped orders (`IndexName = SPARSE_SHIPPED`)
- Read last 10 shipped orders for a customer (`IndexName = SPARSE_SHIPPED, SPARSE_SHIPPED_PK = CUSTOMER#JHD, Limit = 11`)
- Read all shipped orders for a customer shipped after X date (`IndexName = SPARSE_SHIPPED, SPARSE_SHIPPED_PK = CUSTOMER#JHD, SPARSE_SHIPPED_SK >= 2020-10-01`)

Vertical Partition Patterns

DynamoDB can store items that can be up to 400 KB. You should avoid having such large documents if, in most access patterns, do not need the whole item. DynamoDB always reads whole items and, after that, applies projections and filtering, so having large items brings a huge waste of resources. The common practice is to split into multiple items.

Split into Multiple Documents with the Same Partition Key

You can easily split the document into multiple documents with the same partition key and separate parts with a different sort key. You can read everything at once if you specify only the portion key or part of it if you also specify the sort key or part of it. So the principle is the same as in the One to Many Relationship Pattern.

Separate Content into Another Record and Leave Summary

The trick is to move the large portion of data into separate items and leave the only summary in the source table. That works well with most applications where you only want to see the data's extract, and when you click the item, you get all of the data.

Optimizing Partition Management—Avoiding Hot Partitions

The key principle of DynamoDB is to distribute data and load it to as many partitions as possible. But that does not work if a lot of items have the same partition key or your reads or writes go to the same partition key again and again. Doing so, you got hot partition, and if you want to avoid throttling, you must set high Read/Write Request Units and overpay. However, you can still hit the maximum quotas set by AWS. You should structure the data to avoid that. There is an escape hatch—caching data. You can cache data on the application level with, for example, Radis, or you can use DAX, the DynamoDB build-in solution for caching. DAX also ensures that you always get current data.

Caching is suitable solution for same cases. Properly structuring data should be the first choice.

Do not forget that the following pattern, like most patterns in this article, can also be used in GSI. The principals are the same.

Duplicating Records

If you are reading the same item repeatedly, you can make more copies and read from one of them. You add a sequential number of the copy to the partition key. When you access the data, just add a random number and read from one of the copies.

This pattern optimizes reads, not writes, because you have to copy data upon each change.

Example:

PK	Settings
GLOBAL_SETTINGS_01	...
GLOBAL_SETTINGS_02	... (same data as GLOBAL_SETTINGS_01)
GLOBAL_SETTINGS_03	... (same data as GLOBAL_SETTINGS_01)

Write Sharding

When you cannot avoid having the same partition key, you can shard data by adding a random number to the portion key to distribute items among partitions. You add a number, typically as a suffix, from 1 to N. When you read the data, you must iterate through 1 to N to find the records that you are searching for and then manually combine results.

This pattern optimizes writes. Reads are slower, but the main point is to avoid hot partition. You can do additional optimization that will, in some cases, enable you to avoid this iteration to read the data. This number for suffix does not need to be random. Maybe there is some other data you already have before reading that this number can be based on. It can be a short hash or modulus (the remainder in division) that gives you the number from 1 to N.

For example, you want to store all the events that occur while processing the order. For the logs partition key, you chose the date. That would mean too much data for one partition key, so you add a suffix from 1 to N. You can calculate that suffix from order data. So when you read logs for just one order, you know the suffix. But if you need to read all the data for that day, you have to iterate from 1 to N, read all data, combine it, and probably order it.

Example:

PK	SK	TYPE	Data
2020-10-19_1	2020-10-19T09:19:32	LOG	...
2020-10-19_2	2020-10-19T09:22:52	LOG	...
2020-10-19_3	2020-10-19T09:24:42	LOG	...
2020-10-19_4	2020-10-19T09:30:44	LOG	...
2020-10-20_1	2020-10-20T12:58:14	LOG	...
2020-10-20_2	2020-10-20T12:59:23	LOG	...
2020-10-20_3	2020-10-20T13:02:53	LOG	...
2020-10-20_4	2020-10-20T13:04:34	LOG	...

In this example, we sharded logs for each day to 4 shards. If we want to read all logs for a particular date, we need to execute 4 reads. We put a timestamp into the sort key so that we can limit the results to time range.

Replica with GSI

One of the easiest ways to resolve hot partition is to replicate data with GSI. You can create GSI with the same partition and sort key as primary data. You can use projection only to copy a part of the data. You can use this GSI for some read patterns without affecting the primary table. By doing so, you can optimize reads. In some cases, you would also reduce costs.

Singleton Pattern

There is nothing special about this pattern. If you need just one record, for example, holding system states or settings, you can store everything in just one record. The important part is that if the item is frequently read, you should consider making multiple copies. See Duplicating Records.

Times Series Data Cost Optimization Pattern

Times series data typically has a characteristic that only the recent data is accessed most frequently. The older the data it is, the less frequently it is needed. You can optimize this for cost saving by having a new table for each period. You set higher read/write capacity for tables with more recent data and decrease it when it gets older.

Patterns Based on Modification Constraints

For the following patterns, two DynamoDB features are essential:

- **Condition expressions**
  When manipulating data in an Amazon DynamoDB table, you can specify a condition expression to determine which items should be modified. This is an extremely powerful feature.
- **Set collection type**
  DynamoDB can, amongst many attribute types, hold lists or sets. They are similar, but sets values are unique. The useful feature of the set is also when removing the item, you can specify the value that you want to remove. In lists, you have to specify the index of the element, which you might not know. Using sets in DocumentClient, the most commonly used JavaScript library, is tricky. Arrays are by default converted to lists.

Limiting the Number of Items in a Set (or List)

When writing to a list or set, you can limit the number of items. This can be useful for some business requirements or just simple technical ones. For example, you want to maintain a list of up to 10 data processing jobs, and you do not want to exceed that number.

Example (TypeScript):

```
const docClient = new DocumentClient({ apiVersion: '2012-08-10' });
//Creating a record to store the job queue.
await docClient.put({
  TableName: 'data',
  Item: {
    PK: "JOBQUEUE",
    SK: "JOBQUEUE",
  },
}).promise();
const jobId = 'JOB#6412';
//Adding a job in case it is the first job.
//DynamoDB does not support an empty set.
//If the set already exists, the following action will do nothing.
await docClient.update(
  {
    TableName: 'data',
    Key: {
      PK: "JOBQUEUE",
      SK: "JOBQUEUE",
    },
    UpdateExpression: "SET #inProgress = if_not_exists(#inProgress, :jobId)",
    ExpressionAttributeNames: {
      "#inProgress": "inProgress"
    },
    ExpressionAttributeValues: {
      ":jobId": docClient.createSet([jobId])
    }
  }).promise();
try {
  //Adding a job to the existing set.
  const result = await docClient.update(
    {
      TableName: 'data',
      Key: {
        PK: "JOBQUEUE",
        SK: "JOBQUEUE",
      },
      ConditionExpression: "size(#inProgress) < :maxItems",
      UpdateExpression: "ADD #inProgress :jobId",
      ExpressionAttributeNames: {
        "#inProgress": "inProgress"
      },
      ExpressionAttributeValues: {
        ":jobId": docClient.createSet([jobId]),
        ":maxItems": 10
      },
      ReturnValues: "ALL_NEW"
    }).promise();
  const jobs = (result.Attributes!["inProgress"] as DocumentClient.DynamoDbSet).values;
  console.log("Processing job: ", jobs);
}
catch (err) {
  if (err.code === "ConditionalCheckFailedException") {
    console.log("Job queue is full");
  }
  else {
    throw err;
  }
}
```

Condition Expression on the Same Item

In this example, we are limiting changes of the document only to users that are editors. The list of editors is stored as a list in the attribute.

PK	TYPE	Document	Editors
DOCUMENT#JKK	DOCUMENT	...	["John", "Michael"]
DOCUMENT#MKG	DOCUMENT	...	["Susan", "Tom"]

Example (TypeScript):

```
//Creating the document
const docClient = new DocumentClient({ apiVersion: '2012-08-10' });
await docClient.put({
  TableName: 'data',
  Item: {
    PK: "DOCUMENT#JKK",
    SK: "DOCUMENT#JKK",
    TYPE: "DOCUMENT",
    editors: ["John", "Michael"],
    content: "Some content"
  },
}).promise();

try {
  const result = await docClient.update(
    {
      TableName: 'data',
      Key: {
        PK: "DOCUMENT#JKK",
        SK: "DOCUMENT#JKK",
      },
      ConditionExpression: "contains(#editors, :user)",
      UpdateExpression: "SET #content = :newContent",
      ExpressionAttributeNames: {
        "#content": "content",
        "#editors" : "editors"
      },
      ExpressionAttributeValues: {
        ":newContent": "New content"           ,
        ":user" : "John"
      },
      ReturnValues: "ALL_NEW"
    }).promise();
  const content = result.Attributes!["content"];
  console.log("Content after update: ", content);
}
catch (err) {
  if (err.code === "ConditionalCheckFailedException") {
    console.log("The user is not an editor");
  }
  else {
    throw err;
  }
}
```

Condition Expression across Multiple Items

The list of editors can be written on another item. The principal is the same, but you have to use a transaction.

PK	TYPE	Document	Editors
DOCUMENT#JKK	DOCUMENT	...	
DOCUMENT#MKG	DOCUMENT	...	
EDITORS	EDITORS		["John", "Michael"]

```
//Creating the document
await docClient.put({
  TableName: 'data',
  Item: {
    PK: "DOCUMENT#JKK",
    SK: "DOCUMENT#JKK",
    TYPE: "DOCUMENT",
    content: "Some content"
  },
}).promise();

//Creating record to store editors
await docClient.put({
  TableName: 'data',
  Item: {
    PK: "EDITORS",
    SK: "EDITORS",
    TYPE: "EDITORS",
    editors: ["John", "Michael"],
  },
}).promise();

const result = await docClient.transactWrite({
  TransactItems: [
    {
      ConditionCheck: {
        Key: {
          PK: "EDITORS",
          SK: "EDITORS"
        },
        TableName: "data",
        ConditionExpression: "contains(#editors, :user)",
        ExpressionAttributeNames: {
          "#editors": "editors"
        },
        ExpressionAttributeValues: {
          ":user": "John"
        }
      }
    },
    {
      Update: {
        TableName: 'data',
        Key: {
          PK: "DOCUMENT#JKK",
          SK: "DOCUMENT#JKK",
        },
        UpdateExpression: "SET #content = :newContent",
        ExpressionAttributeNames: {
          "#content": "content",
        },
        ExpressionAttributeValues: {
          ":newContent": "New content",
        }
      }
    }
  ]
});
result.on('extractError', (response: any) => {
  try {
    let cancellationReasons =
JSON.parse(response.httpResponse.body.toString()).CancellationReasons;
    console.error("Transaction fail reason:", cancellationReasons);
  } catch (err) {
    console.error('Error extracting cancellation error', err);
  }
});
await result.promise();
```

Idempotent Set Manipulation

Because sets contain unique values, they enable you to add the same item many times, and you still end up with only one item. The operation is idempotent, which means it can be executed many times and it still provide the same result. This is extremely useful in the serverless system because many services provide at least once invocation, meaning you should expect that Lambda could be triggered multiple times for the same event. Operation is also idempotent when removing the item from the set.

You can see example in chapter Limiting the Number of Items in a Set. Here is also extended blog on the subject.

Differentiation of Insert and Update

Put method does not differentiate between insert and replacing the item. Sometimes you want to restrict to inserts only, which you can do with the `attribute_not_exists()` condition.

Example (TypeScript):

```
try {
  const result = await docClient.put({
    TableName: 'data',
    Item: {
      PK: "ACTION#2341",
      SK: "ACTION#2341",
      ExecutedAt: (new Date()).toISOString()
    },
    ConditionExpression: "attribute_not_exists(#PK)",
    ExpressionAttributeNames: {
      "#PK": "PK"
    }
  }).promise();
}
catch (err) {
  if (err.code === "ConditionalCheckFailedException") {
    console.log("Action has already been logged");
  }
  else {
    throw err;
  }
}
```

Optimistic Locking

The principal of optimistic locking is that you update the item only if it was not changed since you read it from the database. This is simplify achieved with a special attribute that holds the version number, which we increase when we execute an update. With the condition, we guarantee that the item was not updated.

```
try {
  var currentVersion = 3;
  const result = await docClient.update({
    TableName: 'data',
    Key: {
      PK: 'ITEM#2345',
      SK: 'ITEM#2345'
    },
    UpdateExpression: 'set #data = :newData, #version = :newVersion',
    ConditionExpression: '#version = :expectedVersion',
    ExpressionAttributeNames: {
      '#data': 'data',
      '#version': 'version'
    },
    ExpressionAttributeValues: {
      ':newData': 'New data',
      ':newVersion': currentVersion + 1,
      ':expectedVersion': currentVersion
    },
    ReturnValues: 'ALL_NEW'
  }).promise();
}
catch (err) {
  if (err.code === "ConditionalCheckFailedException") {
    console.log("Concurrency error");
  }
  else {
    throw err;
  }
}
```

Version History Pattern with Manual Transactions

This pattern fits two purposes:

- **Maintaining multiple versions of the documents**
- **Manually handling a transaction for cases when built-in transactions are too limiting**

Stored multiple versions of a document can be decorated with the metadata who and when changed it. This way, you have an audit trail of all the changes.

In RDBMS databases, while executing transactions, you can access the database or other systems multiple times. The transaction can last for some, preferably short, time. This is not possible in DynamoDB. DynamoDB transactions are executed as a batch of up to 25 items. No starting and then committing or rollbacking the transaction and doing something while it is open.

With Version History pattern, you can do multiple access to the database, prepare data for the new state, then mark data as current and in a sort of committing the transaction.

Example:

PK	SK	CurVer	Timestamp	Author	Attributes
1	V0	2			
	V1		2020-10-19T09:19:32	Tom	...
	V2		2020-10-19T09:22:52	Simon	...
	V3		2020-10-19T09:24:42	John	...

All versions are stored with the same partition key. Sort key identifies the version, except for V0 that only holds the number of the current version. In this sample, while preparing a third version of the data, the second version is current. Creating a new version can consist of changing multiple records or even accessing other systems. When everything is completed, we change the first record to point to the new version 3.

The downside of this pattern is that each read demands two access to the database. First, to get the appropriate version and then for the actual read.

Attribute Versioning Pattern

Storing multiple versions of the document is sometimes needed. The problem is if the document is big and you pay a lot of WCU for writing it. The solution is to split large items into smaller items, and when some part changes, you store and version only those parts. This pattern optimizes writes. Reads are more complex because you have to assemble the item from different parts and pick only the newest or appropriate version for that part.

PK	SK	Attributes	Timestamp	Author
DOCUMENT#KBH	INTRO_V1	...	2020-10-19T09:19:32	Tom
DOCUMENT#KBH	MAIN_CONTENT_V1	...	2020-10-19T09:22:52	Simon
DOCUMENT#KBH	SUMMARY_V1	...	2020-10-19T09:24:42	John
DOCUMENT#KBH	MAIN_CONTENT_V2	...	2020-10-19T10:19:32	Tom

DynamoDB Streams

DynamoDB streams provide you with an option to process data after it has been inserted, updated, or deleted in the database. They are great for modern event-driven systems. The data can be processed with Lambda or a standalone application that uses the Kinesis Client Library (KCL) with the DynamoDB Streams Kinesis Adapter.

Most common use cases:

- **Aggregation**
  DynamoDB does not have a powerful ad hoc query language like SQL, and you cannot extract data for analytics in an easy way. With streams, you can precalculate that data.
- **Real-time analytics**
  It is really common today to analyze a large amount of data just coming to the system. You can process the data yourself or forward from Lambda to Kinesis Data Streams and then to Kinesis Data Analytics. Kinesis Data Analytics has a special SQL-like query language for real-time processing. Unfortunately, there is no direct connection to the Kinesis Data Analytics jet.
- **Hybrid database solutions**
  DynamoDB has its drawbacks. You can transfer data to the SQL database or ElasticSearch and use the more suitable database for a particular use case.
- **Fan-out processing**
  Streams will scale up with the amount of data and invoke as many Lambdas as needed.
- **Notification**
  You can notify the system of some important events. Lambda can trigger SNS or send messages to EventBreach, etc.
- **Event streaming/Event stream processing**
  This is the pattern that is commonly used with Kafka or AWS Kinesis. The concept is not to view data as a whole set but to deal with a continuously created data flow. You can attach multiple listeners to the stream so each microservice can process the data for their purpose in a way already listed above. Data is asynchronously shared across multiple parts of the system.

DynamoDB streams guarantee at least once delivery, which means it could receive the same event twice. You should make your processing idempotent, taking into account possible multiple triggering of the same event. For example, if you are calculating the summary, you should mark the already processed data. You can use DynamoDB transactions for that. The other option, because this repetition of the call is rare, is to ignore the problem. This is acceptable if the data is more informative and does not have to be absolutely correct. An example would be counting page visits.

Tools

NoSQL Workbench for Amazon DynamoDB

Documenting a single table data model is challenging. You cannot visualize it like an ER diagram. That is why NoSQL Workbench for Amazon DynamoDB from AWS is an extremely useful tool. You enter some sample data, define indexes, and see how data looks like when accessing through secondary indexes. You can even generate code.

CloudWatch Contributor Insights for DynamoDB

CloudWatch Contributor Insights for DynamoDB enables you to see the most accessed and throttled items in a table or GSI. This will help you resolve hot keys.

Node.js Tricks

AWS Node.js SDK offers two APIs for accessing DynamoDB: AWS.DynamoDB and the AWS.DynamoDB.DocumentClient. DocumentClient is a wrapper around the first one to provide simplified API. Both of them are awful, and it is hard to work with them. That's why there are alternatives:

- **DynamoDB Toolbox** is a library from Jeremy Daly that offers you better API, compared to the one from AWS SDK. It is designed with single tables in mind. Unfortunately, it does not have a TypeScript support jet, but it is in the making.
- **Dynoexpr** is another Node.js library for accessing DynamoDB. It supports TypeScript. Its goal is to improve features offered by AWS DocumentClient by simplifying building expressions.

Enable HTTP Keep-Alive

AWS SDK, by default, does not enable HTTP keep-alive. For every request, a new TCP connection has to be established with a three-way handshake.

To enable it, set environment variable `AWS_NODEJS_CONNECTION_REUSE_ENABLED` to `1` or set the property of a HTTP or HTTPS agent set to true in code.

Resources

The DynamoDB Book

As the title says, this is the go-to book for DynamoDB. Author Alex DeBrie did an amazing job describing all the patterns cleanly and consistently.

The DynamoDB Book by @alexbdebrie is go-to book for #DynamoDB https://www.dynamodbbook.com/

CLICK TO TWEET

He also blogs about DynamoDB:

- https://www.alexdebrie.com,
- https://www.dynamodbguide.com

And maintains a great list of DynamoDB resources:

- https://github.com/alexdebrie/awesome-dynamodb

His session at re:Invent was also exceptional:

- AWS re:Invent 2019: Data modeling with Amazon DynamoDB

Rick Houlihan Lectures

Rick Houlihan is the master wizard of DynamoDB at AWS. You cannot miss his AWS re:Invent sessions:

- AWS re:Invent 2019: Advanced Design Patterns
- AWS re:Invent 2018: Advanced Design Patterns
- AWS re:Invent 2017: Advanced Design Patterns

DynamoDB Key Concept

Kisan Tamang just recently posted an amazing introduction to DynamoDB.

Design Patterns Intro

This is a nice short video tutorial for the most important DynamoDB patterns.

Do you need temporary or permanent help on your project?

8 Challenges Teams Face When Doing Serverless

Serverless, as new technology, brings new challenges to the team. New technology is not the only thing to learn.

JavaScript - From Obscure Language to Mainstream Language for Serverless

JavaScript was for decade considered a failed language. But now it is a mainstream multipurpose language that also fits perfectly with serverless solutions.

AWS Serverless Common Mistakes – Architecture (2/7)

Mistakes you do while architecting serverless solutions. This is part of a multipart blog post about serverless common mistakes.