A Modeling Approach for AWS DynamoDB

Jim Ladd
Wazee Group, Inc.

December 23, 2018

Wazee Group
Software Construction & Renovation

Contents

Wazee Group
Software Construction & Renovation

Introduction

NoSQL databases represent a fascinating technology and Amazon Web Services DynamoDB, with its performance features and extremely low cost, is a very interesting option. To facilitate the evaluation and potential transition to DynamoDB, this document presents a modeling approach that is straightforward and consistent. A sample problem is presented along with the logical view of the data. The physical view of the table is created by following a set of design rules. The Python code used to create and query the table is also presented.

Overview

DynamoDB is a NoSQL, fully managed database offering by Amazon Web Services (AWS). AWS claims that it provides fast, predictable performance with seamless scalability. Along with other NoSQL databases, DynamoDB provides storage and retrieval capabilities that rely on keys associated with flexible structures.

Besides performance, scalability, and other technological features, DynamoDB has another key factor; extremely low cost. Currently, the free tier being offered by AWS is as follows based on a monthly per-region, per-payer basis

- 25 GB of data storage
- 2.5 million stream read requests from DynamoDB Streams
- 1 GB of data transfer out, aggregated across AWS services

With such economic incentives, many of my clients are evaluating DynamoDB for their storage needs. Even if the technical requirements don't demand DynamoDB, clients are considering incorporating the service for cost considerations alone. One challenge is that the NoSQL paradigm is significantly different than the one of traditional relational databases. The transition is not as quick and easy as some would estimate. Hopefully, this document will assist with the evaluation of DynamoDB and the transition to this exciting paradigm.

Problem Space

To explain the modeling approach, let's assume a problem domain that lends itself to one of the more interesting scenarios with traditional databases, the "many-to-many" relationship. This example is a simple contact information problem where a person's different addresses are maintained over time.

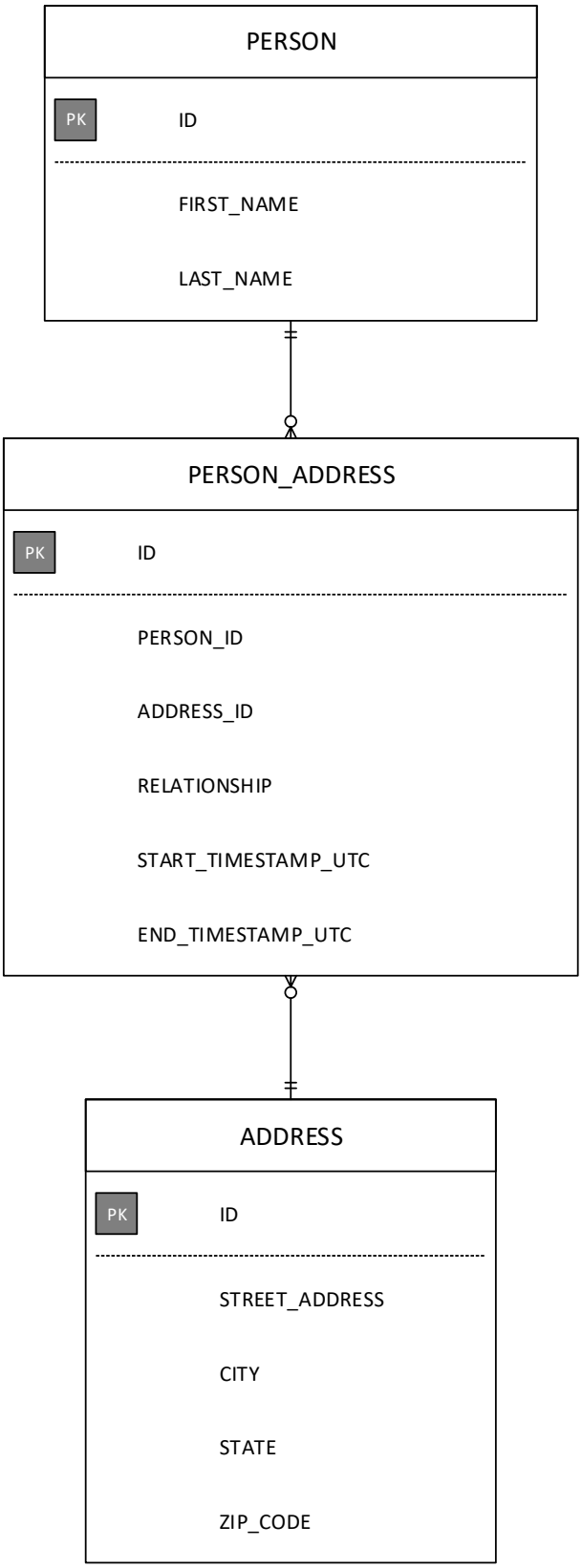
Our information can be represented in a single table. Basically, a person has a relationship with an address for a period of time including present day. The relationship can be either a business or a residential type. The current address is denoted by a lack of end date. The actual data used in the example is shown below.

FirstName	LastName	StreetAddress	City	State	ZipCode	Relationship	StartDate	EndDate
Bob	Smith	1850 Wazee Street	Denver	CO	80202	Business	1/1/2000	12/31/2010
Bob	Smith	1420 Stout Street	Denver	CO	80202	Business	1/1/2011	
Bob	Smith	1600 15th Street	Denver	CO	80202	Residential	1/1/2000	
Sally	Wilson	1850 Wazee Street	Denver	CO	80202	Business	1/1/2005	12/31/2015
Sally	Wilson	4000 Larimer Street	Denver	CO	80202	Business	1/1/2016	
Sally	Wilson	2000 21st Street	Denver	CO	80202	Residential	1/1/2005	
Joe	Penton	1850 Wazee Street	Denver	CO	80202	Business	1/1/2009	12/31/2009
Joe	Penton	4050 Logan Street	Denver	CO	80202	Business	1/1/2010	
Joe	Penton	5000 Blake Street	Denver	CO	80202	Residential	1/1/2009	

Solution Space

Even though we will be using a NoSQL-based solution, one of the first steps I perform in designing a solution is to create a logical view of the data with an Entity Relationship Diagram (ERD). I favor graphical models, especially in the early phases of design, collaborating with colleagues, and presenting to non-technical personnel. The logical ERD for our example is show below:

Wazee Group
Software Construction & Renovation



Wazee Group
Software Construction & Renovation

Once the logical view becomes “stable”, the next step is to create the “physical” view of the database. While there are many different ways to proceed, I wanted a design approach that would 1) cover a wide range of problem domains, business needs, and technical requirements, 2) be straightforward (i.e. few design rules), and 3) consistent (consistency is a MAJOR factor to high levels of productivity).

Here are the rules for this approach:

- There is one table for each domain or service (i.e. Customer, Order, etc.).
- The PrimaryId is called “Id” and is a universally unique identifier (UUID).
- The SortKey is called “Type” and denotes the type of data structure.
- There is a default GlobalSecondaryIndex and has the name of “*<table name>* + TypeIndex”. For example, for a table with the name of Customer will have an index with the name of CustomerTypeIndex.
- The global secondary index has the “Type” as the PrimaryKey and the “Id” as the SortKey.
- There can be several Attributes which compose the remaining data structure.

Applying these rules to our example, the physical view of the database is:

TABLE	*Contact*		
	PrimaryId	**SortKey**	**Attributes**
	Id *(UUID)*	Type *("Person")*	FirstName
			LastName
	Id *(UUID)*	Type *("Address")*	StreetAddress
			City
			State
			ZipCode
	Id *(UUID)*	Type *("PersonAddress”)*	PersonId
			AddressId
			Relationship
			StartTimestampUTC
			EndTimestampUTC
INDEX	*ContactTypeIndex*		

	PrimaryId	**SortKey**	
	Type	Id	

The Python code used to create this table is very straightforward and can be used to construct any table that follows the design rules. The code is:

```
import boto3

dynamodb = boto3.resource( 'dynamodb' )

#########################################################

def createTable( myTableName ) :

    table = dynamodb.create_table(
        TableName = myTableName,
        KeySchema=[
            {
                'AttributeName': 'Id',
                'KeyType': 'HASH'  #Partition key
            },
            {
                'AttributeName': 'Type',
                'KeyType': 'RANGE'  #Sort key
            }
        ],
        AttributeDefinitions=[
            {
                'AttributeName': 'Id',
                'AttributeType': 'S'
            },
            {
                'AttributeName': 'Type',
                'AttributeType': 'S'
            },

        ],
        GlobalSecondaryIndexes=[
            {
                'IndexName': myTableName + 'TypeIndex',
                'KeySchema': [
                                {
                                'AttributeName': 'Type',
                                'KeyType': 'HASH'  #Partition key
                              },
                              {
                                'AttributeName': 'Id',
                                'KeyType': 'RANGE'  #Sort key
                              }
                            ],
                'Projection': {
```

```
                    'ProjectionType': 'ALL'
                    },
                'ProvisionedThroughput': {
                    'ReadCapacityUnits': 1,
                  'WriteCapacityUnits': 1
                }
            },
        ],
        ProvisionedThroughput={
            'ReadCapacityUnits': 1,
            'WriteCapacityUnits': 1
        }
    )

##########################################################

dynamodb = boto3.resource( "dynamodb" )
createTable( 'Contact' )
```

Once the data in our table is inserted in the "Contact" table, there are several different ways to query the table. The remainder of this section of the document provides Python snippets demonstrating some of those queries.

To find all of the persons in the table, the following snippet uses the boto3 API and the global index:

```
dynamodb = boto3.resource( "dynamodb" )
tableName = 'Contact'
indexName = tableName + 'TypeIndex'
table = dynamodb.Table( tableName )

response = table.query(
    IndexName=indexName,
    KeyConditionExpression=Key('Type').eq( 'Person' )
)
```

The result is:

```
{'Items': [
{'FirstName': 'Bob', 'Id': '1302c80a-7c61-4920-93a4-
23c44c931945', 'Type': 'Person', 'LastName': 'Smith'},
{'FirstName': 'Joe', 'Id': '63283cda-028f-4016-852c-
716c4e1b997e', 'Type': 'Person', 'LastName': 'Penton'},
{'FirstName': 'Sally', 'Id': '89e6210d-2dc8-4fb7-824d-
6eb77918f284', 'Type': 'Person', 'LastName': 'Wilson'}],
...
}
```

To retrieve a set of people by their last name:

```
response = table.query(
    IndexName=indexName,
    KeyConditionExpression=Key('Type').eq( 'Person' ),
    FilterExpression=Attr('LastName').eq( 'Wilson' )
)
```

The result is:

```
{'Items': [{'FirstName': 'Sally',
      'Id': '89e6210d-2dc8-4fb7-824d-6eb77918f284',
      'Type': 'Person',
      'LastName': 'Wilson'}],
…
}
```

To find a specific person by their Id (actually this will retrieve any object by the Id value)

```
response = table.query(
    KeyConditionExpression=
         Key('Id').eq('1302c80a-7c61-4920-93a4-23c44c931945' )
)
```

The result is:

```
{'Items': [{'Id': '1302c80a-7c61-4920-93a4-23c44c931945',
'FirstName': 'Bob', 'LastName': 'Smith', 'Type': 'Person'}],
…
}
```

To determine where Bob was working on 1/1/2005:

```
dateFormat = '%m/%d/%Y'

aDate = datetime.strptime( '1/1/2005', dateFormat )

targetDateUTC =
int(aDate.replace(tzinfo=timezone.utc).timestamp())

response = table.query(
    IndexName=indexName,
    KeyConditionExpression=Key('Type').eq( 'PersonAddress' ),
    FilterExpression=
  Attr('PersonId').eq( '1302c80a-7c61-4920-93a4-23c44c931945' ) &
         Attr('Relationship').eq( 'Business' ) &
         Attr('StartTimestampUTC').lte( targetDateUTC ) &
         Attr('EndTimestampUTC').gte( targetDateUTC )
)
```

The PersonAddresses are:

```
{'Items': [{'AddressId': 'ab3161c6-2462-49b3-957a-d1db9478532f',
'PersonId': '1302c80a-7c61-4920-93a4-23c44c931945',
'EndTimestampUTC': Decimal('1293753600'), 'StartTimestampUTC':
Decimal('946684800'), 'Relationship': 'Business', 'Id':
'eca6da1c-0c94-4b3e-8531-f4f9481330ef', 'Type':
'PersonAddress'}],
…
}
```

Now retrieve the Address by its Id:

```
response = table.query(
    KeyConditionExpression=
  Key('Id').eq( 'ab3161c6-2462-49b3-957a-d1db9478532f' )
)
```

This is the address where Bob was working on 1/1/2005:

```
{'Items': [{'StreetAddress': '1850 Wazee Street', 'City':
'Denver', 'Id': 'ab3161c6-2462-49b3-957a-d1db9478532f', 'State':
'CO', 'ZipCode': '80202', 'Type': 'Address'}],
```

To determine where Bob is currently living:

```
response = table.query(
    IndexName=indexName,
    KeyConditionExpression=Key('Type').eq( 'PersonAddress' ),
    FilterExpression=
  Attr('PersonId').eq( '1302c80a-7c61-4920-93a4-23c44c931945' ) &
          Attr('Relationship').eq( 'Residential' ) &
          Attr('StartTimestampUTC').lte( targetDateUTC ) &
          Attr('EndTimestampUTC').not_exists()
)
```

This is the map to the address:

```
{'Items': [{'PersonId': '1302c80a-7c61-4920-93a4-23c44c931945',
'AddressId': '1440e345-99b0-4c4a-941d-67bfbd03ba30',
'StartTimestampUTC': Decimal('946684800'), 'Relationship':
'Residential', 'Id': '713b7bfe-8e80-42f0-bbb0-3c94d98404fd',
'Type': 'PersonAddress'}],
…
}
```

Retrieve the address that was in the map:

```
response = table.query(
    KeyConditionExpression=
```

```
        Key('Id').eq( 'ab3161c6-2462-49b3-957a-d1db9478532f' )
    )
```

This is where Bob is currently residing:

```
{'Items': [{'StreetAddress': '1600 15th Street', 'City':
'Denver', 'Id': '1440e345-99b0-4c4a-941d-67bfbd03ba30', 'State':
'CO', 'ZipCode': '80202', 'Type': 'Address'}],
```

Summary

This document presented a set of rules for designing tables in DynamoDB. The approach is straightforward and consistent. While not the ultimate solution for every case, the approach should cover a wide range of projects and provide a great starting place.