Megan Conley / conleyme
CS496 Assignment 3 part 1

I will be making a database of online game high scores using DynamoDB from AWS. DynamoDB tables store *items* which are identified by a key, which are actually attributes used to identify the item. The key is either a singular key called a hash or a combination of two keys, the hash and range keys. The hash key must be unique and if it is the only key, items may be retrieved from the table but no other queries can be made. The addition of the range key allows for queries against the table and to be able to have multiple items with the same hash key.

My database will store 2 entities: users and user high scores. The attributes for users will be date joined, handle, and location. User high scores will have 3 attributes: date, game, and score.

As mentioned before, a table with only a hash key can have items retrieved, but no other queries performed. Because any use case that involves a request to the user table will only need all of the data for that item and no others, this table can safely only have a hash key.

The users table will have the user handle/nickname as the hash key, as these must be unique. Below is a sample item modeling how the data might look in the Users table given in DynamoDB's pseudo-JSON format.

Table name	**Hash key**	**Sample item**

Users	user_id	{
		“user_id = johnnydoe”
		location = “Seattle, WA”
		date_joined = 20150415
		}

The user high score table should be sortable by score; this will allow queries to find, for example, the global top score for a certain game. When a table is queried, it automatically sorts either ascending or descending on the range key; the range key should then be score, with the user id as hash key. Since a single user may play multiple games and thus have a recorded high score, this table needs a global secondary key to allow multiple items with the same hash key. And since this is a high score board, a user will only ever have one entry per game, making the game title a safe secondary key to ensure only unique hash-range-secondary key entries.

Table name	**Hash key**	**Range key**	**Secondary Key**	**Sample items**

HighScores	user_id	score	game	{
				user_id = “johnnydoe”
				score = 40000
				game = “Tetris”
				date = 20150415
				}
				{
				user_id = “johnnydoe”
				score = 3200
				game = “Flappy Bird”
				date = 20150416
				}

Although it seems redundant, the nature of querying DynamoDB requires this sort of completely denormalized structure. Unlike a relational database, there is no way to do cross-table queries. The only way to do so is to query one table, save the result, then use that result in a query on the second table. When I first started to design the database, I was tempted to have the users have a numerical id, and then be represented by this id in other tables, such as I would have done in a MySQL database. However, since this would needlessly require 2 queries in DynamoDB the data ends up being completely unrelated.

With the above structure, any data I would want to request can now be done with fewer transactions with the server than if I had attempted to add relations, increasing response time and decreasing possible costs – most any query I would want to make can be done in a single transaction. This is done at the cost of increased storage size of each entity. Because in this class we will be creating an API to retrieve data, speed of response is more important and was chosen over storage size.

While I went with DynamoDB as it is both native to my cloud platform and is used more widely in the industry, I also looked into MongoDB. MongoDB does not have the same limitations as DynamoDB in that it eschews the entire key concept and simply stores all attributes together in what they refer to as a document. Other than that, the structure of the documents is almost identical to a DynamoDB item – most easily represented in JSON and supports embedded attributes (attributes which are arrays of data). Documents are stored together in a collection, similar to the tables of DynamoDB and relational databases. Ultimately, the way I store the data would not change very much in MongoDB. DynamoDB does not actually use JSON, but something slightly different, and MongoDB uses regular JSON to store the items.

The only real difference in how the items are stored in MongoDB is the lack of keys. Each item is essentially a collection of attributes on which queries can be made. The lack of keys frees up some of the restraints on entering data, making DynamoDB's requirement to have up to 12 keys for a single item (hash, range, and up to 5 each of local or global secondary indexes) to differentiate them from each other seem needlessly complicated. The querying language is much more robust, simplified, and human-readable, supporting regular expressions.

Besides being native to AWS, DynamoDB's big advantage is its automation. MongoDB requires a lot more work into maintaining servers and setting up sharding, while DynamoDB will handle that automatically by scaling up and down as needed, and any server issues taken care of by AWS engineers. MongoDB must have enough capacity at all times to account for any highs, resulting in a higher overall cost over time and initial investment in hardware. This also means that updates, which must be done manually with MongoDB, will be done by AWS engineers with no action required by the users.

While most of that isn't too relevant to these student projects, and the flexibility of MongoDB's querying language was tempting, I decided to go with the platform that would be more useful to know in the long term, which is DynamoDB.