TINFO210: DynamoDB

Design Goal

My requirements for this project are to create a database that can accept an ORDER with related ORDER ITEMS and to measure insert and read times for test data. I took two approaches to this – a multi-table approach and a single-table approach. The multi-table approach was easier to wrap my head around as it was similar to a traditional relational database. However, Amazon recommends using a single table for an application unless some of the data has access patterns that are very different from the rest of the data. I abandoned the multi-table approach and focused my efforts on a single-table.

I can support three entities with my design – the required ORDER and ORDER ITEMS, as well as an associated CUSTOMER. I attempted to follow something similar to the adjacency list pattern. Unfortunately, I don't think I got it quite right.

Schema Example

	Primary Key						
Entity	**pk**	**sk (gsi1 pk, gsi2 sk)**	**gsi1-sk (gsi2 pk)**	**Data**			
Customer	**CUSTOMER ID** cust-d28@uw.edu	**DATE JOINED** 2018-06-04		**FIRST** Derek	**LAST** Miller	**ADDRESS** USA|WA|Tacoma|1933 Dock St Unit 221	**ZIP** 98402
Order	**ORDER ID** order-6938c54e-1aaa-425b-8a7c-5ba1354b5183	**CUSTOMER ID** cust-d28@uw.edu	**ORDER STATUS** UNFILLED	**STATUS DATE** 2018-06-03			
Order Item	**ORDER ID** order-6938c54e-1aaa-425b-8a7c-5ba1354b5183	**PRODUCT ID** prod-gvoHn	**CUSTOMER ID** cust-d28@uw.edu	**PRICE** 103	**QTY** 4		

Supported Use Cases

	Query Support		
Use Case	**Index**	**Partition Key: Data**	**Sort Key: Data**
createOrder	table	pk: orderID	sk: custId
getOrder	table	pk: orderID	sk: custId
getAllOrdersByCust	sk-date-placed	sk: custID	date-placed: isoDateTime
getAllOrderItemsByCust	gsi2	gsi1-sk: custId	sk: prodId
updateOrderStatus	table	pk: orderId	sk: custId
getOrdersByPriority	order-priority	gsi1-sk: status	date-placed: isoDateTime

Test Data

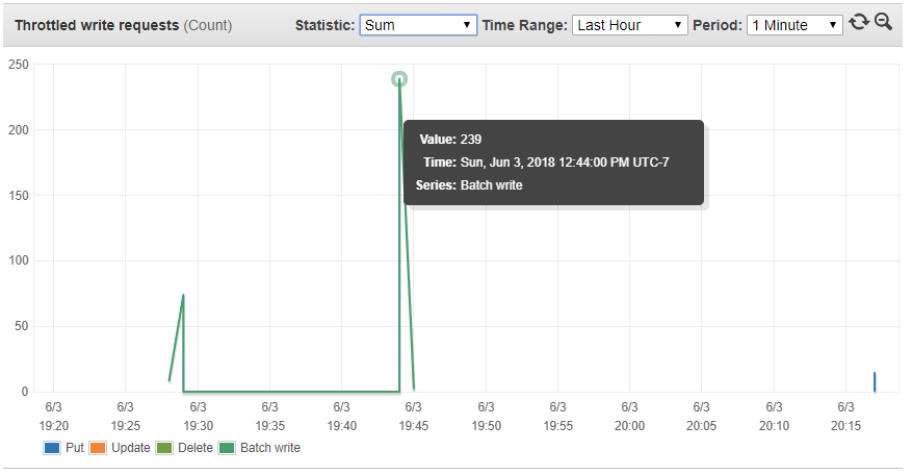
I was required to utilize test data. For a small database, creating my own seed data would probably not be difficult. However, one of the requirements was to generate 1000 order items. I decided to generate random data that was still somewhat consistent For example, I used the NPM package 'randomstring' to generate random product IDs. While this did not provide perfect randomization, it fit my needs. To generate a price, I used the UTF code of the first character of the product ID. This allowed for variation in prices, but was still consistent for each product ID. Since DynamoDB is not capable of generating unique keys, I used another NPM package to generate UUIDs, virtually guaranteeing that no two orders would have the same ID.

First Attempt

One of the requirements for this project was to run a query that creates orders with 1, 10, 100, and 1000 lines. I wrote a script that runs these queries asynchronously (which in turn runs the batch jobs asynchronously). I got the following error:

```
1_Items: 385.191ms
10_Items: 382.370ms
{
    "message": "The level of configured provisioned throughput for the table was
    exceeded. Consider increasing your provisioning level with the UpdateTable API.",
    "code": "ProvisionedThroughputExceededException",
    "time": "2018-06-03T19:44:59.004Z",
    "requestId": "9KC71UH2CS34DL7L4VTBTV70BBVV4KQNSO5AEMVJF66Q9ASUAAJG",
    "statusCode": 400,
    "retryable": true
}
Size of data: 0.3 KB
100_Items: 40526.318ms
```

You can see that the 1 and 10 item requirements completed fairly normally. Note that these aren't DynamoDB insert latency times, but rather the time it took to complete a Promise (roughly). This includes network latency which is obviously unpredictable. The 100_Items script completed very slowly since it was throttled. The 1000_Items didn't even complete. You can see significant throttling at 1 WCU:



Orders were in fact inserted into the database:

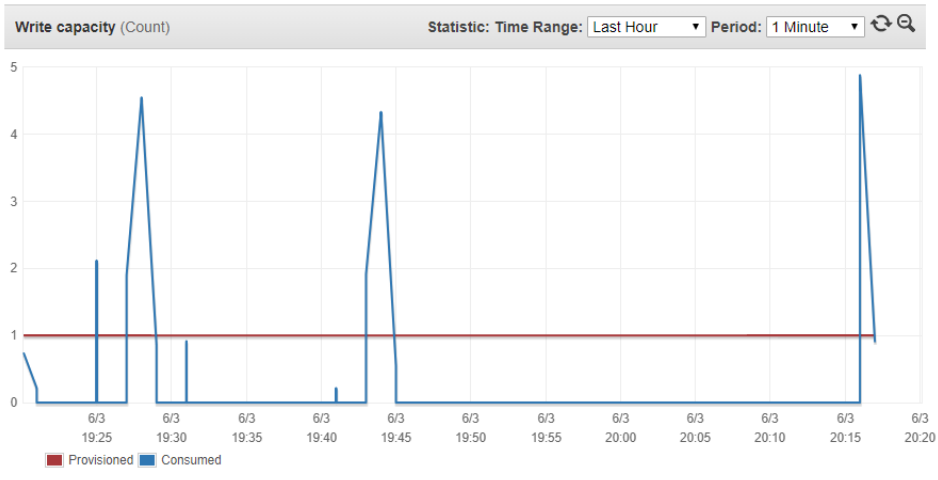
pk	sk	gsi1-sk	price	qty
order-d39bd390	prod-eZpcQ	cust-d28@uw...	101	12
order-d39bd390	prod-eZufq	cust-d28@uw...	101	3
order-d39bd390	prod-egqtj	cust-d28@uw...	101	10
order-d39bd390	prod-ehOvi	cust-d28@uw...	101	8
order-d39bd390	prod-ejUHN	cust-d28@uw...	101	7
order-d39bd390	prod-eqGiz	cust-d28@uw...	101	1
order-d39bd390	prod-fHSoI	cust-d28@uw...	102	5
order-d39bd390	prod-fUYvC	cust-d28@uw...	102	12
order-d39bd390	prod-fkCdC	cust-d28@uw...	102	6

However, only 293 out of 1115 items were inserted. I tried again, but without the 1000_Item script:

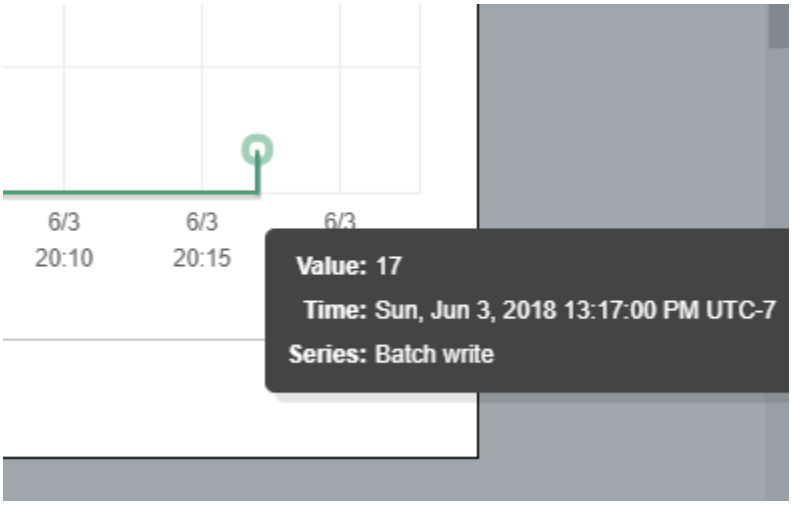
```
1_Items: 360.497ms
10_Items: 2357.932ms
100_Items: 5499.700ms
ALL_QUERIES: 5553.613ms
```

All the queries completed after about 5.5 seconds. This is odd because earlier I was getting sub-400ms finish times.

You can see that I am exceeding my Write Capacity of 1 WCU:



And in fact, I was being throttled just a little:



I think the reason I wasn't being throttled in previous queries was because Amazon provides a small amount of burst capacity for short periods of time. I must have exceeded this burst capacity when I attempted to write 1000 items in the database. After waiting a couple of hours, I attempted the query again, with much better results, albeit still not great:

```
1_Items: 263.487ms
10_Items: 271.555ms
100_Items: 276.822ms
ALL_QUERIES: 329.608ms
```

My major mistake was attempting to run all four tests asynchronously at only 1RCU/1WCU. I changed that for the next test.

Second Attempt

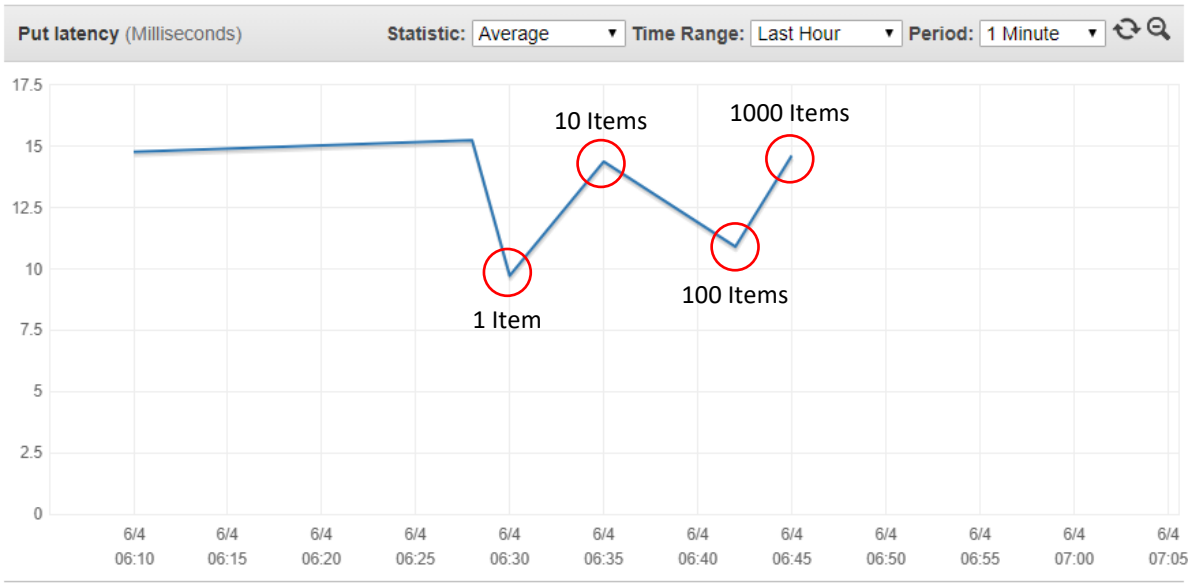
To get latency times that I could correlate with the actual queries, I needed to run each query individually, one minute apart. Otherwise, CloudWatch aggregates these values together. The 1000 item requirement was exceeding my 1WCU capacity so I upped my WCU's and RCU's to 25 for the table and indexes. Evidently this is the maximum allowed for the free tier. At 25 RCUs/WCUs I got the following results:

Number of Items	Avg Put Latency (ms)	Avg Get Latency (ms)	Put Client Op Time (ms)	Get Client Op Time (ms)
1 Item	9.7	8.7	285.7	285.9
10 Items	14.4	16.9	227.8	308.3
100 Items	10.9	10.5	269.7	381.0
1000 Items	14.6	**39.7**	3293.7	511.7

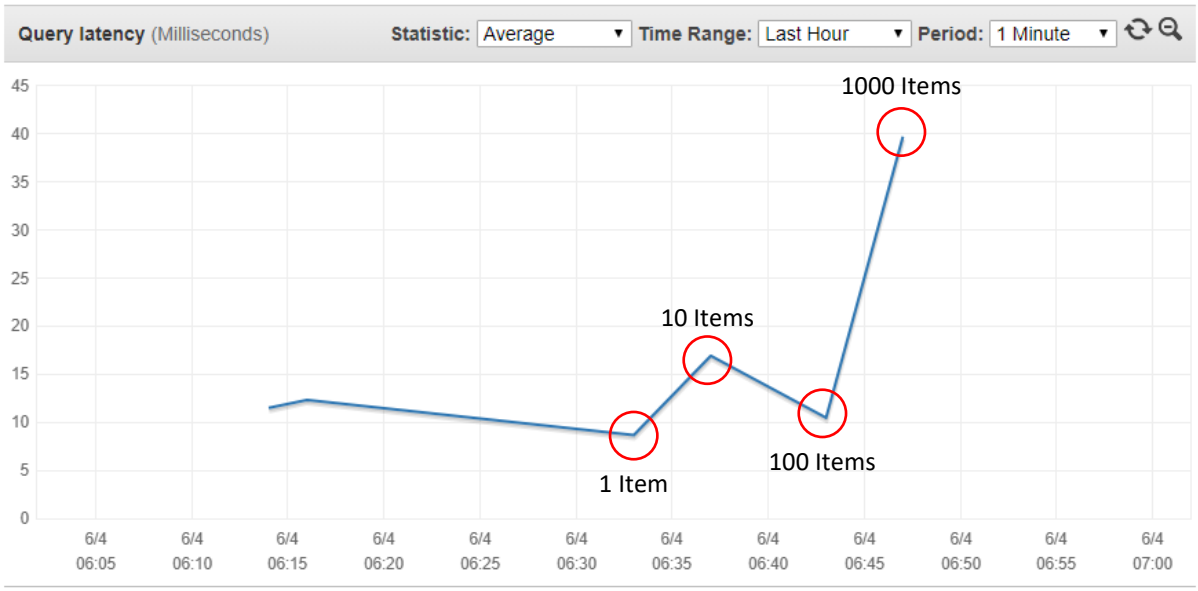
I am more interested in the Put and Get latency values because I think that these are considered important metrics. I believe sub-10ms latency is considered good. My results never really achieved that, but they were fairly consistent, except for the 39.7ms latency on the 1000 item get request. I'm not

certain what was going on there – I did hit the 25 RCU capacity. However, I did not go over and no throttling occurred. Here are the CloudWatch metrics:

Puts (start at 0630):



Gets (start at 06:33):



Conclusion / Reflection

Overall, I feel my database was functional but not well designed. I ended up with four GSI's (five is the maximum allowed). Had I properly followed the adjacency list pattern, I could have more efficiently overloaded my indexes. Also, you can see my index names don't follow a consistent pattern. This is because I was adding and deleting them as I was working. Obviously not a good way to design a system.

This database can perform the queries it was designed to perform. However, I don't think it would respond well to change and offers only mediocre performance.

This experience has been enlightening and frustrating. DynamoDB is much more open ended than relational databases which means the developer must make more decisions about how to interact with the data. Much of the work is programming, not queries. This increases the flexibility of working with data but is more work on the part of the developer. It is also frustrating that there really aren't a lot of publicly available resources to learn technical aspects in depth. Even StackOverflow was a bit light on material. But perhaps the most important lesson I learned is that in NoSQL, every feature also has a trade-off. Nothing comes free.