All Articles

DynamoDB Transactions Performance Testing

This is part 2 of a two-part post on DynamoDB Transactions. Check out part 1 in this series for a look at how and when to use DynamoDB Transactions.

In this post, we're going to do some performance testing of DynamoDB Transactions as compared to other DynamoDB API calls. As a reminder from the last post, you can use DynamoDB Transactions to make multiple requests in a single call. Your entire request will succeed or fail together — if a single write cannot be satisfied, all other writes will be rolled back as well.

This capability is powerful but obviously comes at a performance cost. I was curious to see how large that performance cost is.

In this post, we'll see the results from a performance test I ran. We'll cover:

- Test setup, methodology, and configuration
- DynamoDB performance test results

All code to reproduce these results are available in the dynamodb-performance-testing GitHub repo.

Let's get started!

Setup and Methodology

When I first considered running performance tests for transactions, I was most interested in two things: (1) how do transactions compare to the batch-based API actions, and (2) how does a failed transaction affect latencies?

As I thought more about it, I realized I had several more performance questions.

How do regular `PutItem` requests compare to `BatchWriteItem` requests?

How does a `BatchWriteItem` request with a few items compare to one with a large number of items?

Ultimately, I decided to go with six different configurations:

- `PutItem` request to write a single item
- `BatchWriteItem` request to write 5 items
- `BatchWriteItem` request to write 25 items
- `TransactWriteItems` request to write 5 items
- `TransactWriteItems` request to write 25 items

- `TransactWriteItems` request to write 5 items, including one item that will fail and thus rollback the transaction.

Test Configuration

To run the test, I wanted to avoid polluting test results from specifics of my machine, such as additional network latency from my perch in Omaha, NE or reduced memory availability due to my hundreds of Chrome tabs. To handle this, I deployed some Lambda functions to execute some queries against DynamoDB.

Each Lambda function is doing a few things:

1. Creating a DynamoDB client that has already created a TCP connection with the DynamoDB servers. Yan Cui has written on the overhead this can create. To do this, I made a `DescribeTable` call to DynamoDB before using it in any of my tests.

2. Preparing a wrapped version of the DynamoDB API call that includes pre-generated arguments for the API call, as I didn't want to include time generating items and UUIDs in my metrics.

3. Executing the API call and tracking the execution time using Python's timeit module. (Note to self: Read the docs on `timeit` before use).

4. Logging the latency of the API call using the CloudWatch Embedded Metric Format. This enables me to run percentile queries using CloudWatch Metrics.

A few other notes on the configuration:

- Each invocation of the function will make the API call ten times using ten different parameters. I did this just to make the testing go a little faster. Each API call is made in serial because I'm using Python and I didn't want to mess with async code. 🤷

- I'm using DynamoDB On-Demand pricing rather than Provisioned Capacity. A few folks on the DynamoDB team assured me there shouldn't be any performance impact from using On-Demand.

- I'm using Python, which could add some serialization/deserialization time to my results. If you want to see results in $YOUR*FAVORITE*FAST_LANGUAGE, feel free to replicate this yourself!

Without further ado, let's look at the results!

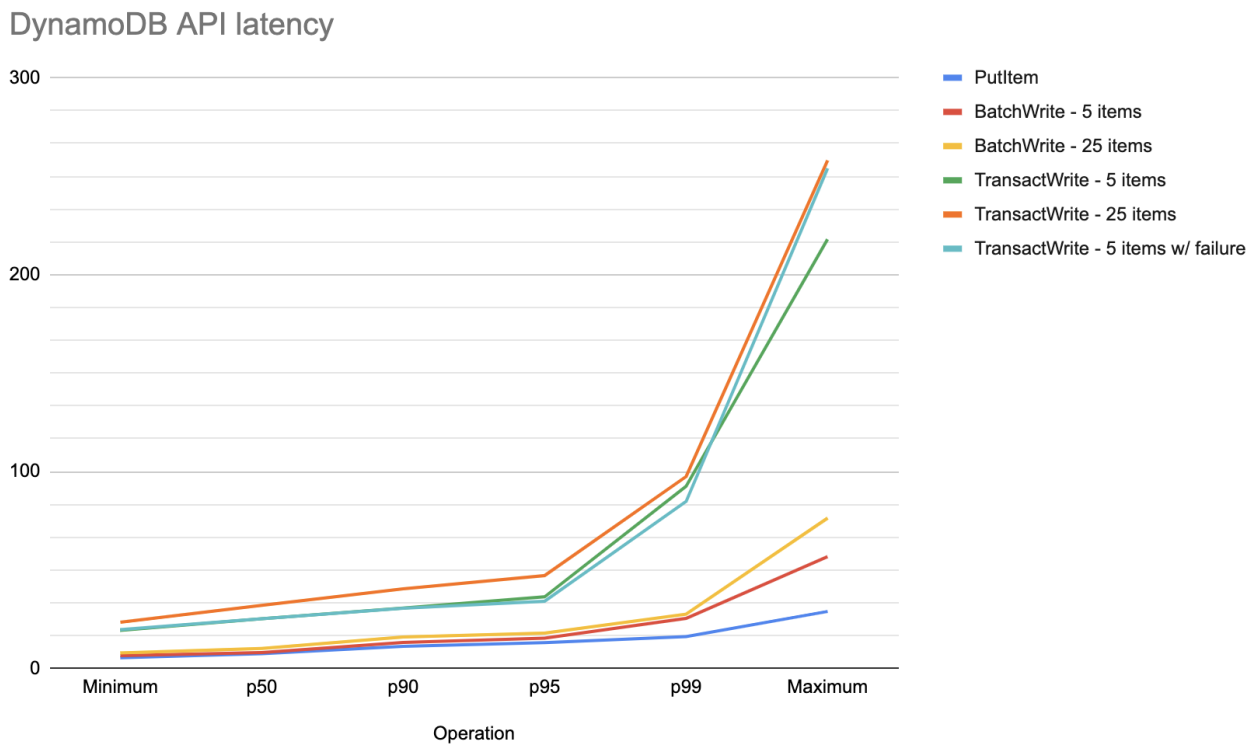
DynamoDB Performance Test Results

For each configuration, I ran 400 Lambda invocations. Each Lambda executed the API call 10 times, so each configuration got 4000 data points total.

Results are shown in the chart below. Each configuration shows minimum and maximum values, as well as p50, p90, p95, and p99 percentiles. Also, note that these times are recorded at the client, so it's showing round-trip time, including the network latency both to and from the DynamoDB server. Times shown are in milliseconds.

Operation	Min	p50	p90	p95	p99	Max
PutItem	5.47	7.55	11.3	13.2	16.2	29
BatchWrite - 5 items	6.59	8.14	13.3	15.4	25.5	56.8
BatchWrite - 25 items	7.96	10.2	16	18	27.6	76.4
TransactWrite - 5 items	19.4	25.3	30.7	36.4	92.5	218
TransactWrite - 25 items	23.5	32.1	40.5	47.2	97.5	258
TransactWrite - 5 items w/ failure	19.8	25.3	30.6	34.2	84.9	254

If you prefer it in graph format, the following graph shows the various configurations:



Key Takeaways

I had the following key takeaways from the results:

1. As one would expect, the simple `PutItem` request was the fastest across the board.

2. The `BatchWriteItem` requests were slower than the single `PutItem` requests but not by much. The performance impact was around 35% at lower percentiles and 70% at higher percentiles. While the percentage seems high, it's still sub-20 milliseconds round-trip to store 25 items in DynamoDB. That's quite a bit faster than doing 25 different `PutItem` requests yourself.

3. There is a performance penalty for `TransactWriteItems` requests — they're between 3x and 4x slower than their batch counterparts.

4. Transaction *failures* don't seem to have much impact on performance. Transaction *size* does have an impact, similar to the impact of batch size with `BatchWriteItem`.

Conclusion

In this post, we did some performance testing with DynamoDB. From these results, we are able to see the relative performance impacts of using operations like `BatchWriteItem` or `TransactWriteItems` in our application.

While the transactional APIs are slower than the non-transactional APIs, the benefits you get from them are often worth the cost. Whether you need idempotency in your operations or confidence to operate on multiple items at once, using transactions will often save you time and headaches over the alternatives.

If you have questions or comments on this piece, feel free to leave a note below or email me directly.

Published 19 Feb 2020

AWS DynamoDB

AWS Data Hero providing training and consulting with expertise in DynamoDB, serverless applications, and cloud-native technology.
Alex DeBrie on Twitter

ALSO ON **ALEXDEBRIE**

How to Migrate an existing DynamoDB …

2 years ago • 2 comments

Take your serverless application global with DynamoDB Global Tables

SQL, NoSQL, and Scale: How …

a year ago • 14 comments

Understand when and why relational databases don't scale, and how …

How and Why to Use CloudFormation …

2 years ago • 2 comments

Macros are a powerful tool for making CloudFormation easier to work with. …

Th Yo

2 ye

Ser hot in 2

What do you think?

5 Responses

 Upvote  Funny  Love  Surprised  Angry  Sad

0 Comments **alexdebrie** **Disqus' Privacy Policy** **Login**

Recommend Tweet Share **Sort by Best**

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS

Name

Be the first to comment.

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Do Not Sell My Data**