ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 3 March 2024

Disparities Between Windows and Linux Architectures in Containerized Dynamodb

[1] Saket Aryan, [2] Adnan Shahid, [3] Subrata Goswami, [4] Rohan Das, [5] Sabya Sachi Pal, [6] Sagar Roy Prodhan

[1] [2] [4] [5] Siksha 'O' Anusandhan
[3] [6] Samsung Research Institute, Bangalore
Corresponding Author Email: [1] saketaryan2002@gmail.com, [2] adnansh2804@gmail.com, [3] subrata.g@samsung.com, [4] rohandas0205@gmail.com, [5] sabyasachipal335@gmail.com, [6] sagar.roy@samsung.com

Abstract*— DynamoDB, a NoSQL database, offers scalability and integration with other AWS services, making it a popular choice for large-scale data storage. However, its performance can suffer with increasing table size and suboptimal schema design. This study investigates the impact of increasing table size on DynamoDB performance across diverse platforms, hardware configurations, and operating systems. We explore the role of optimal schema design in achieving peak performance at minimal cost and provide insights for guiding schema decisions and platform selection. By analysing performance variations with table size growth, we aim to derive recommendations for effective DynamoDB implementation, taking into account both performance and cost considerations.

***Index Terms*— AUFS, Docker Daemon, Hyper-V Containers, Native Windows Containers.**

I. INTRODUCTION

Cloud computing is a model for delivering IT services over the Internet with pay-as-you-go pricing. It has revolutionized the way businesses operate, providing access to scalable and affordable computing resources on demand.

One of the key drivers of the rise in cloud computing is the exponential growth of data. Businesses of all sizes are now collecting and storing more data than ever before. This data is essential for businesses to operate efficiently and make informed decisions. However, storing and managing large amounts of data can be complex and expensive.

Cloud computing offers a scalable and cost-effective solution for data storage. Cloud providers such as Microsoft Azure, Google Cloud, Amazon Web Services, etc, offer a wide range of data storage services that can be tailored to meet the specific needs of each business. AWS currently being the most popular cloud computing platform because it offers a wide range of services, is highly scalable and reliable, and is cost-effective.

II. CONTEXT

A. Structured Query Language (SQL):

It is a relational database management system (RDBMS) that is used to store and manage data in a tabular format. SQL databases are typically used for storing and managing structured data that needs to be accessed and analysed in a relational manner, such as customer information, product information, and financial data.

B. Non-SQL (NoSQL):

It is a type of database that does not use the traditional SQL relational model. NoSQL databases are designed to be more scalable and flexible than SQL databases, making them a good choice for storing and managing large amounts of unstructured data that does not need to be accessed and analysed in a relational manner, such as images, videos, and social media posts.

C. AWS DynamoDB:

AWS DynamoDB is a fully managed, multi-region, multi-master, durable NoSQL database with built-in security, backup and restore, and in-memory caching for internet-scale applications. DynamoDB offers consistent single-digit millisecond latency at any scale. It is a popular choice for storing and managing large amounts of data due to its scalability, reliability, and performance.

D. Windows Subsystem for Linux (WSL):

Windows Subsystem for Linux (WSL) is a Windows feature that allows users to operate a Linux environment on their Windows devices without needing a separate virtual machine or dual boot configuration. The design aims to provide a seamless and effective experience for developers who want to use both Windows and Linux at the same time.

III. METHODOLOGY

In this section, we discuss how we determined the benchmarks and queries to be run, along with the dataset. Before benchmarking, the main aspects to note and address include:

1. Determining the correct table schema.
2. Identifying the appropriate Global Secondary Index (GSI).
3. Collecting/Generating the database data.
4. Defining the queries to be performed.

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 3 March 2024

A. Dataset

So, the dataset was sourced from Kaggle, specifically the "List of People Names by Countries." It comprised a total of 42,399 rows or entries, with the following column names:

1. Country Code
2. Country
3. Name of Athlete
4. Sport

Out of this, the Name, Country, and Sport fields were selected for the final dataset. Additional fields, namely Class, Grade, and ID, were introduced to emulate a database for students, including information about their grades, class, and favourite sports. It's important to note that the Class and Grade fields were randomly generated, and no methods were employed to ensure the homogeneity of the data. Furthermore, the ID field was generated sequentially to mimic the concept of a Registration Number or Roll Number. We planned to execute queries and scans on multiple table sizes to assess the performance differences as the table size scaled up or down. Consequently, we opted to consider five table sizes: 10,000, 42,399 (Original Size), 100,000, 500,000, and 1 million.

Throughout the remainder of our paper, we refer to the 10,000 table as the 10K table, the original table as the 42K table, the 100,000 table as the 100K table, the 500,000 table as the 500K table, and the 1 million table as the 1M table. Now, since the data was only available for the size of 42K, in order to scale down 10000 random rows were picked and put into the final 10K dataset with new IDs. For the scaling up the same method was followed and only the ID was provided in sequential order again.

B. Setting up the database

The chosen database for our study was DynamoDB. To set up DynamoDB locally, we opted for the official Docker image provided by AWS. Using Docker Desktop as the client, we pulled the image from the source.After setting up the local DynamoDB, the next task was to create the table using a specified table schema. The chosen schema was influenced both by the planned queries and the consideration that there is a fixed number of countries, making it practical to use one as a Global Secondary Index (GSI) for artificial data partitioning.

The final schema includes the following attributes: Student_class, country, sport, grade, and ID. For the primary key, we decided to designate ID as the partition key, as it is the only viable option and is relevant to the real-world scenario. Additionally, student_class was selected as the sort key. Regarding GSIs, we opted for a single GSI, with the country column as the GSI partition key. This choice facilitates the artificial partitioning of the table, especially given the fixed number of countries. The sort key assigned to this GSI is student_class. Furthermore, sport and grade were assigned as Local Secondary Indexes (LSIs). Sport and Grade were assigned to be the sort key as they had only a few fixed values in it and they can be used to optimize the queries even further.

C. Adding data to the database

With the database schema in mind, the next step was to connect to the database and create it, along with adding data to it. Python's Boto3 Library was used for database connection, and Jupyter Notebook was used to further execute these tasks.

To follow the DynamoDB format, a JSON file was created with the required schema. This file was then used to create the database inside the shared database file. Then, all the data was read from the CSV using Pandas and added to their respective database tables. There were no logs kept regarding the duration of the DynamoDB data addition operation. However, for reference, it took about 5 minutes to add data to the 100K table, 35 minutes for the 500K table, and 1 hour and 40 minutes for the 1M table.

D. Queries

In order to test the data, and the difference in scan and query times, we conducted tests on three different queries with different complexities. For each of these queries, different sizes of results were expected and delivered.

The Queries were as follows:

1. Students from India
2. Students from India in class 5
3. Students from India with grade A in class 4, and whose name starts with ‘s’ or ‘S’

The first query is a direct look-up, and since Country is a GSI, it should, theoretically, take O(1) time to return the query results.

The second query is slightly more complex due to the additional parameter. However, since the column Class is an LSI, on paper, it should not negatively impact the look-up time. In fact, it may theoretically reduce the search area, resulting in improved performance.

The third query is more intricate than the previous two. It introduces two additional parameters, Grade and Name starting with 'S' or 's'. As Grade is an LSI, it should enhance performance. The last added parameter, "Name starts with 'S' or 's'", should not significantly impact the results. The query can efficiently retrieve the mentioned data in O(1) time and then filter the data based on the specified criteria. Hence, including "Name starts with 'S' or 's'" in the query should not substantially affect the performance on paper.

E. Machines

A total of four machines were used during the testing, each with very different specifications.

The Machines

1. Machine 1 - Apple MacBook Air M1 (macOS Ventura 13.5.1)
2. Machine 2 - HP Pavilion g6 (Ubuntu 22.04)
3. Machine 3 - MSI Bravo 15 B5DD (Windows 11)
4. Machine 4 - HP Pavilion Gaming 15-ec2008AX (Windows 11)

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 3 March 2024

Table 1. Specification of all machines used

Specifications	Machine 1	Machine 2 (HP g6)	Machine 3 (MSI)	Machine 4 (HP)
CPU	Apple M1	Intel Core i5-3210M	5th Gen AMD Ryzen 5 5600H	5th Gen AMD Ryzen 5 5600H
Memory	16 GB	8 GB	8 GB	8 GB
Memory Type	LPDDR4X-4266 MHz SDRAM	DDR3 / 1600 MHz	DDR4 / 3200 MHz	DDR4 / 3200 MHz
Memory Bandwidth	66.67GB/s	25.6 GB/s	25.6 GB/s	25.6 GB/s
Storage	256GB SSD	240GB SSD	512 GB SSD	512 GB SSD
Storage Free	159.9 GB of 245.1 GB	77.7 GB of 218 GB	320.1 GB of 459 GB	212.1 GB of 477 GB

F. Storage Benchmarking

For Machine 1 (MACBOOK Air M1):

All (Type)	Read [MB/s]	Write [MB/s]
Sequential	2752.15	2399.2
Random	582.065	63.865

For Machine 2 (HP Pavilion g6):

All (Type)	Read [MB/s]	Write [MB/s]
Sequential	523.11	273.55
Random	61.675	96.925

For Machine 3 (MSI):

All (Type)	Read [MB/s]	Write [MB/s]
Sequential	2606.3	1459.115
Random	257.895	171.625

For Machine 4 (HP):

All (Type)	Read [MB/s]	Write [MB/s]
Sequential	1671.875	1605.37
Random	212.915	185.415

IV. EVALUATION

A. Queries

➢ ***Query 1*:** People from India

Results: The findings of the study reveal the following results for different query counts:

- For a query on 10K, the count query will be 378.
- For a query on 42, the count query will be 1696.
- For a query on 100K, the count query will be 3987.
- For a query on 500K, the count query will be 20,007.
- For a query on 1M, the count query will be 39,940.

➢ ***Query 2*:** People from India in class 5

Results: The study yields the following results for different query counts:

- For a query on 10K, the count query will be 74.
- For a query on 42K, the count query will be 350.
- For a query on 100K, the count query will be 812.
- For a query on 500K, the count query will be 3,867.
- For a query on 1M, the count query will be 7,604**.**

➢ **Query 3:** People from India with grade A in class 4, and whose name starts with ‘s’ or ‘S’

***Results*:** The study yields the following results for different query counts:

- For a query on 10K, the count query will be 2.
- For a query on 42K, the count query will be 7.
- For a query on 100K, the count query will be 17.
- For a query on 500K, the count query will be 37.
- For a query on 1M, the count query will be 63.

Scan*:* Scan time is the same for all runs(query) since the entire table must be iterated over to perform a scan. Scan time may vary depending on the environment (machine) on which it is executed.

1) Machine 1 (MacOS)

Query 1:

Table Size (Entries)	AvgRun for Query (seconds)	AvgRun for Scan (seconds)
10K	0.038366	0.119098
42K	0.103153	0.635979
100K	0.162459	2.70425

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 3 March 2024

500K	0.564999	80.556607
1M	1.242491	361.0251

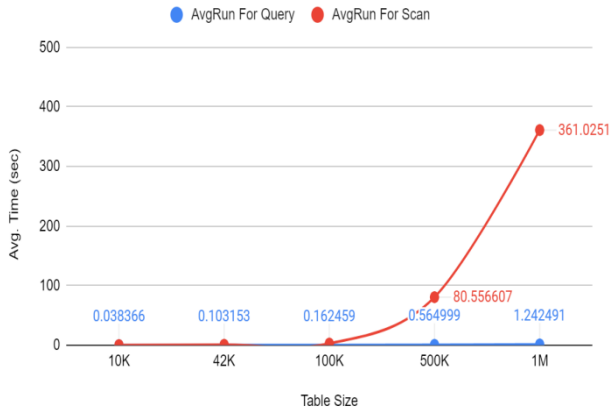

Figure 1. AvgRun for Query and Scan for Machine 1

Query 2:

Table Size (Entries)	**AvgRun for Query (seconds)**	**AvgRun for Scan (seconds)**
10K	0.020791	0.099849
42K	0.047972	0.589463
100K	0.096505	2.687555
500K	0.249246	86.205821
1M	0.422039	375.26977

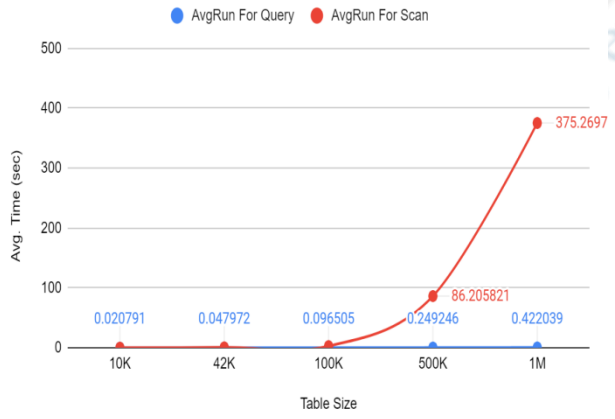

Figure 2. AvgRun for Query and Scan for Machine 1

Query 3:

Table Size (Entries)	**AvgRun for Query (seconds)**	**AvgRun for Scan (seconds)**
10K	0.015785	0.100204
42K	0.018136	0.578728
100K	0.022566	2.565082
500K	0.054499	85.265347
1M	0.081196	379.95171

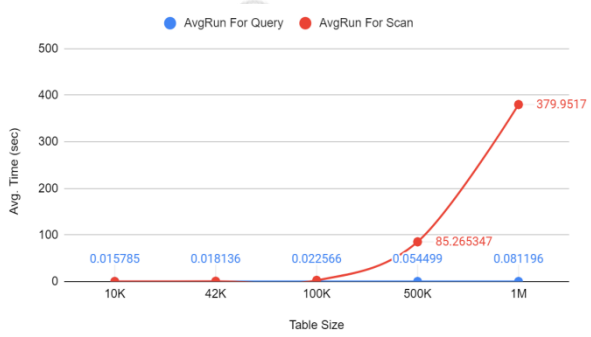

Figure 3. AvgRun for Query and Scan for Machine 1

2) Machine 2 (Ubuntu)

Query 1:

Table Size (Entries)	**AvgRun for Query (seconds)**	**AvgRun for Scan (seconds)**
10K	0.078668	0.202456
42K	0.151059	1.571861
100K	0.319824	7.194271
500K	1.329084	185.51873
1M	2.87935	779.46558

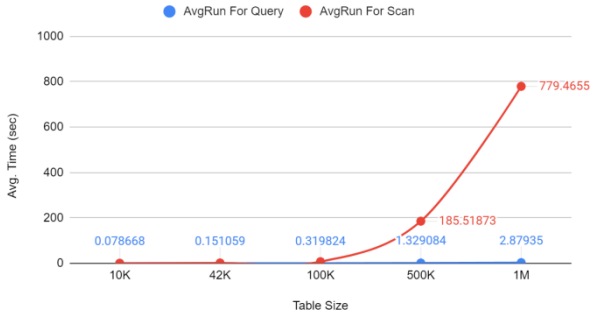

Figure 4. AvgRun for Query and Scan for Machine 2

Query 2:

Table Size (Entries)	**AvgRun for Query (seconds)**	**AvgRun for Scan (seconds)**
10K	0.03044	0.198689
42K	0.087938	1.523186

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 3 March 2024

100K	0.153156	7.005081
500K	0.498439	186.41436
1M	0.893432	870.96605

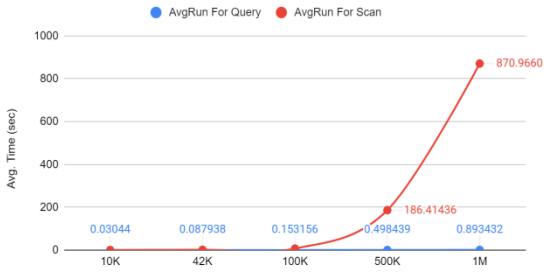

Figure 5. AvgRun for Query and Scan for Machine 2

Query 3**:**

Table Size(Entries)	**AvgRun for Query(seconds)**	**AvgRun for Scan(seconds)**
10K	0.015763	0.191276
42K	0.019278	1.527247
100K	0.023669	6.950054
500K	0.070197	210.99766
1M	0.113171	875.43011

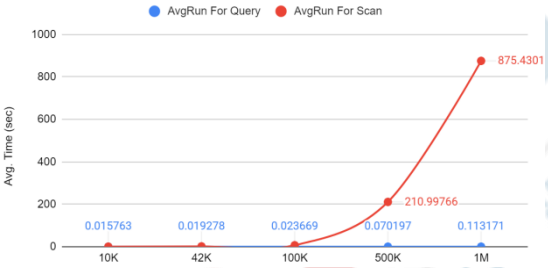

Figure 6. AvgRun for Query and Scan for Machine 2

3) Machine 3 (MSI - Windows)

Query 1:

Table Size(Entries)	**AvgRun for Query(seconds)**	**AvgRun for Scan(seconds)**
10K	0.029117	1.592261
42K	0.067816	36.660815
100K	1.117506	185.348
500K	10.866278	Time Out
1M	34.746781	Time Out

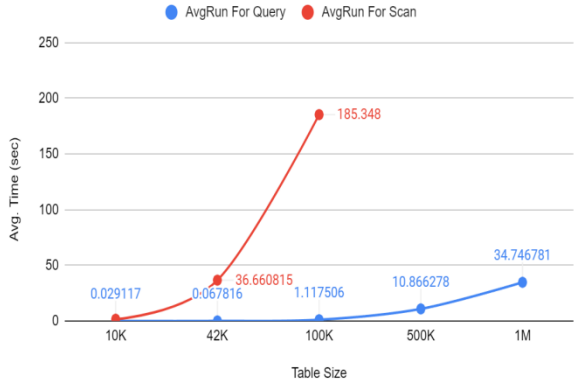

Figure 7. AvgRun for Query and Scan for Machine 3

Here is a summary of the percentage increase in time with respect to increase in data sizes:

Table size (Entries)	**Increase in result count (%)**	**Increase in query time (%)**	**Increase in scan time (%)**	**Increase in time for scan with respect to query (%)**
10K - 42K	348.6772	132.9086	2202.438	53959.24
42K - 100K	135.0825	1547.85	405.5752	16485.86
100K - 500K	401.8059	872.3686	Time Out	Time Out
500K - 1M	99.63013	219.7671	Time Out	Time Out

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 3 March 2024

Query 2:

Table Size(Entries)	AvgRun Query(seconds)	Avg Run Scan(seconds)
10K	0.015893	1.592261
42K	0.029828	36.660815
100K	1.031821	185.348
500K	7.470234	Time Out
1M	16.592536	Time Out

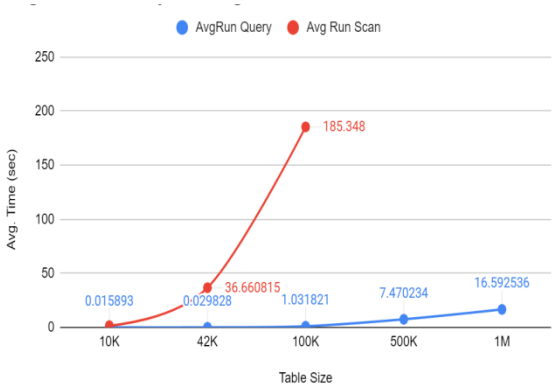

Figure 8. AvgRun for Query and Scan for Machine 3

Here is a summary of the percentage increase in time with respect to increase in data sizes:

Table size (Entries)	Increase in result count (%)	Increase in query time (%)	Increase in scan time (%)	Increase in time for scan with respect to query (%)
10K - 42K	372.973	87.68011	2202.438	122807.4
42K - 100K	132	3359.236	405.5752	17863.19
100K - 500K	376.2315	623.9855	Time Out	Time Out
500K - 1M	96.63822	122.1153	Time Out	Time Out

Query 3:

Table Size (Entries)	AvgRun Query (seconds)	Avg Run Scan (seconds)
10K	0.011382	1.592261
42K	0.012839	36.660815
100K	0.014303	185.348
500K	0.824029	Time Out
1M	1.683862	Time Out

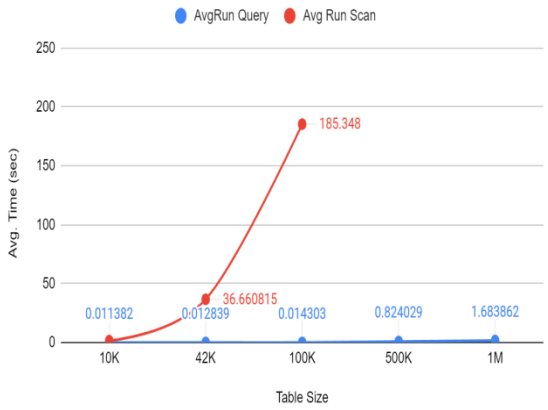

Figure 9. AvgRun for Query and Scan for Machine 3

Here is a summary of the percentage increase in time with respect to increase in data sizes:

Table size (Entries)	Increase in result count (%)	Increase in query time (%)	Increase in scan time (%)	Increase in time for scan with respect to query (%)
10K - 42K	250	12.80091	2202.438	285442.6
42K - 100K	142.8571	11.40276	405.5752	1295768
100K - 500K	117.6471	5661.232	Time Out	Time Out
500K - 1M	70.27027	104.345	Time Out	Time Out

4) Machine 4 (HP-Windows)

Machines 3 and 4 were both based on Windows. Despite

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 3 March 2024

machine 3 having faster SSD read and write speeds than machine 4, both of them performed identically, and no noticeable difference was observed.

B. MongoDB Setup

Moreover, we decided to include scan times in another popular NoSQL Database, MongoDB, aiming to measure and analyse the performance on both Native Windows and Docker environments.

For Native Windows, we used MongoDB CLI, 'mongod', to run the database directly on the Windows environment. Again, for the Docker environment, we established a Docker container using the latest 'mongo' image. The process of adding data to the MongoDB database was straightforward. Using the CSV data we generated earlier, we used the "csv-parser" module with Nodejs to convert the CSV file into the necessary JSON format.

Further, the next step involved was adding the collection or model to the database. To do this, we used the "mongoose" module to create the schema design for the User/Student model within the database.

The next step involved was populating the database, and we opted for an easy approach. Using the JSON file created earlier, we fed it into the Mongo Compass program to add data to the database. This process was exceptionally fast and took mere seconds for all table sizes in comparison to the hours required in DynamoDB.

Further, all other steps were the same as with the DynamoDB part, we wrote the same queries and performed the scans with the same timing function used before, the only change was that instead of boto3 we used "pymongo" to run the queries.

C. Docker Architecture in Windows and Linux

Between the Windows and Linux Docker architectures, both are almost identical but Windows used the Computer Service layer instead of the "containerd" process, possibly causing the Windows machine to perform poorly. However, if all these differences are due to the Docker architecture variance between Windows and Linux, it could most likely be verified with the test we prepared, which it did, as shown in the next section.

D. WSL Optimization

After some digging, we discovered that Docker performance on Windows can be enhanced through WSL2 optimization. Following the official guidelines provided by the Docker documentation, we implemented the best practices for using and setting up Docker Desktop on Windows.

1. Mac and Linux computers demonstrated superior performance over Windows, even excluding the faster Mac system.
2. Linux outperformed Windows significantly, despite Windows having advantages in memory speed and CPU performance.
3. Windows struggled with scan executions, while Linux, though slower than MacOS, exhibited markedly better performance than Windows.
4. Windows struggled with scan executions, while Linux, though slower than MacOS, exhibited markedly better performance than Windows.
5. WSL optimization involved installing WSL, adding Ubuntu, copying shared database data to the Ubuntu directory, reconfiguring Docker to use WSL2, and running the container inside Ubuntu on top of Window

Machine 3 (WSL - MSI)

Query 1:

Table Size(Entries)	AvgRun for Query(seconds)	AvgRun for Scan(seconds)
10K	0.037514	0.099463
42K	0.0988	0.734957
100K	0.157216	3.087912
500K	0.820549	90.891437
1M	2.656253	Time Out

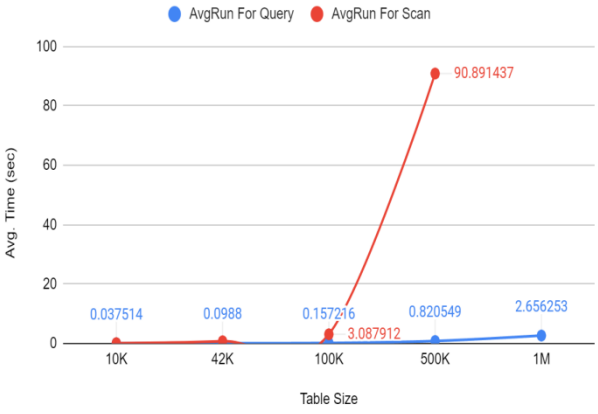

Figure 10. AvgRun for Query and Scan for Machine 3 with WSL optimization

Here is a summary of the percentage increase in time with respect to increase in data sizes:

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 3 March 2024

Table size (Entries)	Increase in result count (%)	Increase in query time (%)	Increase in scan time (%)	Increase in time for scan with respect to query (%)
10K - 42K	348.6772	163.3683	638.925	643.8836
42K - 100K	135.0825	59.12551	320.1487	1864.121
100K - 500K	401.8059	421.9246	2843.459	10976.91
500K - 1M	99.63013	223.7166	Time Out	Time Out

Query 2:

Table Size(Entries)	AvgRun for Query(seconds)	AvgRun for Scan(seconds)
10K	0.009732	0.071067
42K	0.027881	0.651922
100K	0.060008	3.425509
500K	0.251538	85.61952
1M	0.690358	Time Out

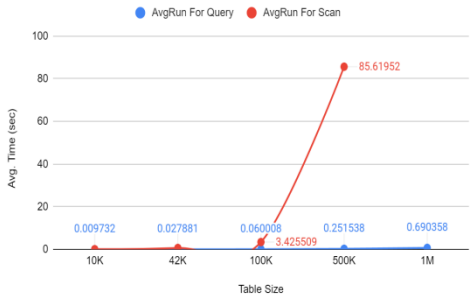

Figure 11. AvgRun for Query and Scan for Machine 3 with WSL optimization

Here is a summary of the percentage increase in time with respect to increase in data sizes:

Table size (Entries)	Increase in result count (%)	Increase in query time (%)	Increase in scan time (%)	Increase in time for scan with respect to query (%)
10K - 42K	**372.973**	**186.4879**	**817.3343**	**2238.23**
42K - 100K	**132**	**115.229**	**425.4477**	**5608.421**
100K - 500K	**376.2315**	**319.1741**	**2399.469**	**33938.4**
500K - 1M	**96.63822**	**174.4548**	**Time Out**	**Time Out**

Query 3:

Table Size(Entries)	AvgRun for Query(seconds)	AvgRun for Scan(seconds)
10K	0.007057	0.074459
42K	0.006276	0.685363
100K	0.008715	3.032363
500K	0.023514	86.564659
1M	0.036525	Time Out

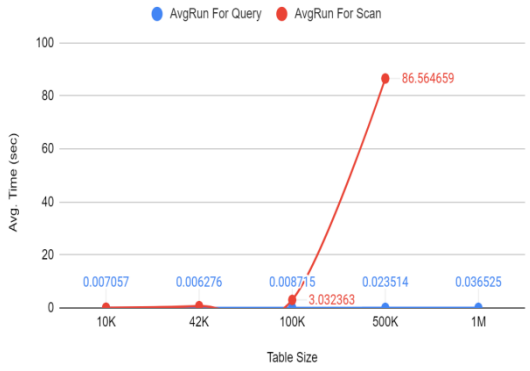

Figure 12. AvgRun for Query and Scan for Machine 3 with WSL optimization

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 3 March 2024

Here is a summary of the percentage increase in time with respect to increase in data sizes:

Table size (Entries)	Increase in result count (%)	Increase in query time (%)	Increase in scan time (%)	Increase in time for scan with respect to query (%)
10K - 42K	250	-11.067	820.4569	10820.38
42K - 100K	142.8571	38.86233	342.4463	34694.76
100K - 500K	117.6471	169.8107	2754.693	368040.9
500K - 1M	70.27027	55.33299	Time Out	Time Out

Discussion: To validate our hypothesis, we executed the operation within the Windows Subsystem for Linux (WSL) environment and observed a noticeable improvement in performance. The execution time achieved in WSL closely resembled that observed on UNIX-based operating systems. This finding strongly suggests that the substantial performance disparity between Windows and UNIX-based systems is indeed attributable to the architectural differences between Docker implementations on these platforms.

V. CONCLUSION

Through our tests, it becomes evident that there is a significant performance gap when using DynamoDB on Windows compared to Unix or Linux-based systems. Notably, the slower machine initially outperformed the faster Windows machines. However, after implementing WSL2 optimization, the performance of Windows machines is on par with that of Mac device. This conclusion is reinforced when observing the use of WSL2 optimization on the Windows machine, indicating that the issue lies not with Windows itself, but with Docker's optimization for DynamoDB specifically on Windows. It's worth noting that MongoDB exhibited similar performance in both native and Docker modes, highlighting that the discrepancy is specific to Docker optimization for DynamoDB on Windows.

REFERENCES

[1] David E Lares S, “Understanding Windows and Linux differences in Docker Architecture,” https://medium.com/analytics-vidhya/understanding-windows-and-linux-differences-in-docker-architecture-6c224b6c9285

[2] Suyash Singh, “Increase WSL2 and Docker Performance on Windows By 20x,” https://medium.com/@suyashsingh.stem/increase-docker-performance-on-windows-by-20x-6d2318256b9a