IJCCS (Indonesian Journal of Computing and Cybernetics Systems)
Vol.x, No.x, July xxxx, pp. 1~5
ISSN (print): 1978-1520, ISSN (online): 2460-7258
DOI: 10.22146/ijccs.xxxx

UUID vs ULID as Primary Keys in PostgreSQL: Performance Comparison

Muhammad Bintang Bahy*[1], Umi Laili Yuhana[2]
[1,2]Department of Informatics, Information Technology Faculty, Institut Teknologi Sepuluh Nopember Surabaya, Indonesia
e-mail: ***6025222003@mhs.its.ac.id**, [2]yuhana@if.its.ac.id

Abstrak

Dalam beberapa tahun terakhir, semakin banyak digunakan Universally Unique Identifiers (UUID) sebagai kunci utama dalam basis data. Namun, untuk memenuhi kebutuhan skalabilitas dan performa, muncul sebuah jenis pengenal unik baru bernama Unique Lexicographically Sortable Identifier (ULID) yang menggabungkan keunikan UUID dengan keuntungan pengenal berurutan. Dalam penelitian ini dibahas perbedaan performa UUID dan ULID dalam basis data berbasis PostgreSQL. Pengukuran kinerja UUID dan ULID dilakukan pada empat jenis operasi: insert, find, update, dan join. Hasil menunjukkan bahwa secara umum ULID lebih baik performanya dibandingkan dengan UUID sebagai kunci utama pada basis data.

Kata kunci*—Basis data, UUID, ULID, Performa*

Abstract

In recent years, Universally Unique Identifiers (UUID) have been increasingly used as primary keys in databases. However, to meet the scalability and performance requirements, a new type of unique identifier called Unique Lexicographically Sortable Identifier (ULID) has emerged, which combines the uniqueness of UUID with the benefits of sequentially ordered identifiers. This study discusses the performance differences between UUID and ULID in PostgreSQL-based databases. Performance measurements of UUID and ULID were conducted on four types of operations: insert, find, update, and join. The results show that ULID generally outperforms UUID as the primary key in databases.

Keywords—*3-5 Database, UUID, ULID, Performance*

1. INTRODUCTION

Databases are an essential component of modern applications, allowing for the efficient storage and management of large amounts of data. The selection of a primary key, which uniquely identifies each record in a table, is an important aspect of database design [1]. The primary key is critical for data integrity, consistency, and performance because it ensures that each record is uniquely identified and quickly retrieved.

Databases have traditionally used integer-based primary keys, which are sequentially ordered and simple to generate. However, as databases grew in size and complexity, the limitations of integer-based primary keys became clear. Integer-based primary keys, for

Received June 1st,2012; Revised June 25th, 2012; Accepted July 10th, 2012

example, can cause performance issues when scaling horizontally because they may require coordination among multiple servers to ensure uniqueness. Furthermore, in distributed systems, integer-based primary keys may not provide enough entropy to ensure uniqueness.

In recent years, Universally Unique Identifiers (UUIDs) have been a popular choice as primary keys in databases to address these limitations. UUIDs are 128-bit globally unique identifiers that are generated using a combination of time and node-specific information [2]. They provide a high level of uniqueness because the likelihood of producing the same UUID twice is extremely low [3]. They are also simple to generate, do not require coordination among multiple servers, and can be used across multiple systems and networks. UUID represented in 32 characters hexadecimals value.

There have been a few research that compares the performance of integer ID vs UUID. Such as [4] that benchmarked the performance of UUID vs integer ID on write operation. The research shows that integer ID have a better performance compared to UUID partially due to ordered nature. Another research [5] also shows that in insert operation integer ID have a better performance compared to UUID, while in other operation the performance difference is negligible.

As databases grow in size and complexity, the need for scalability and performance grows. To address this, the Unique Lexicographically Sortable Identifier (ULID) has emerged as a new type of unique identifier. ULID is intended to strike a balance between distinctiveness, scalability, and performance. It's a 128-bit identifier made up of a 48-bit timestamp and an 80-bit random number [6]. The timestamp provides the identifier's sequential ordering, which is important for efficient database indexing and querying, while the random value ensures the identifier's uniqueness. ULID is represented in 26 characters Crockford's base32 [7].

While ULID is a relatively new identifier, its unique properties have helped it gain popularity in recent years. However, research on its performance in comparison to UUIDs has been limited. As a result, the purpose of this research is to compare the performance of UUID and ULID as primary keys in PostgreSQL-based databases.

We evaluate UUID and ULID performance in four operations: insert, find, update, and join. The findings of this study will provide valuable insights into the performance of these identifiers in various database operations, which will help guide the selection of the best primary key for specific database requirements. Furthermore, this research advances our understanding of the properties of ULID and its potential advantages over other identifiers.

2. METHODS

This section describes the methodology used to compare the performance of UUID and ULID in PostgreSQL-based databases. We first describe the hardware and software environment used in the experiments. Then, we present the database schema used in the experiments and the indexing strategy. Finally, we describe the experimental setup and the metrics used to measure the performance of UUID and ULID in different database operations.

2.1 Environment

The experiments were run on a machine with an Intel Core i7-8750H CPU, 16 GB RAM, and a 512 GB NVMe SSD. The machine runs MacOS 13.1 Ventura. Additionally the test were run on Docker based container using PostgreSQL 15.2 image with 1 CPU and 4 GB memory resource limit.

2.2 Database Schema

We used four tables in this research, two table for each identifier type UUID/ULID. The first table for each identifier have a text primary key, two text column, and one integer column. The second table we used for join operation consists of text primary key, one text column, and one integer column. For generating UUID we used uuid-oosp extension, and for generating ULID we used pgulid library. The database overall schema can be seen in Figure 1.

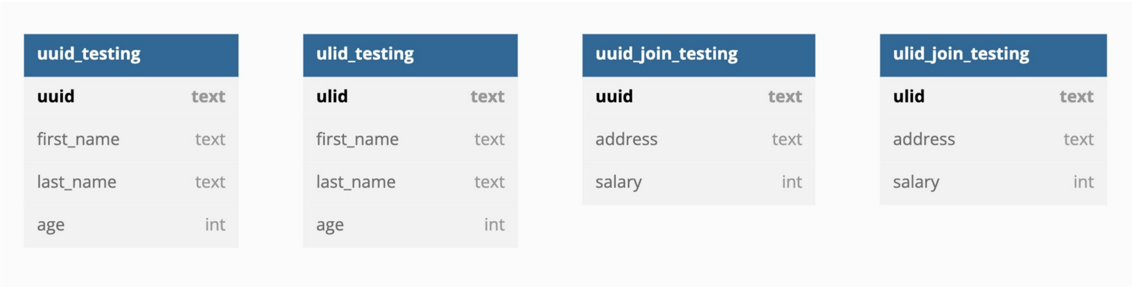

Figure 1 Testing database schema

2.3 Indexing Strategy

To ensure efficient querying of the database, we used B-tree indexes on the primary keys of each table. B-tree indexes are a common type of index used in PostgreSQL, which are efficient for range queries and can handle large datasets.

2.4 Experimental Setup

To evaluate the performance of UUID and ULID, we conducted tests involving four different operations: insert, find, update, and join. For each of these operations, we ran 1 million queries and recorded the total amount of time required to complete them using PostgreSQL "timing" command. We repeated each operation 10 times and then calculated the average results.

- Insert: We inserted 1 million records into each table using both UUID and ULID as primary keys.
- Find: We executed queries to retrieve records from each table based on the primary key. We randomly selected 1 million primary keys and executed queries to retrieve the corresponding records.
- Update: We updated 1 million records in each table based on the primary key. We randomly selected 1 million primary keys and executed queries to update the corresponding records.
- Join: We executed queries to join the "uuid_testing" table with the "uuid_join_testing" and "ulid_testing" table with the "ulid_join_testing" based on the primary key.

3. RESULTS AND DISCUSSION

This section presents the results of the experiments conducted to compare the performance of UUID and ULID in PostgreSQL-based databases.

3.1 Insert Performance

Table 1 shows the time taken to insert 1 million records incrementally using UUID and ULID as primary keys. The findings indicate that, in the beginning, UUID was quicker in the insert operation until roughly 3 million records were added. However, after 4 million records or more were added, ULID's insertion time remained relatively constant, whereas UUID became increasingly slower as shown in Figure 2. The average insertion time for UUID was 49106 ms and UUID was 38286 ms.

The insert performance of UUID is generally faster and constant because its lexicographically ordered nature. The ordered nature makes that when inserting new data, there will not be a lot of B-Tree balancing occured. While on the other hand UUID have a random nature, so every new data ID is not ordered lexicographically and this results in a lot of B-Tree balancing to occur.

Table 1 Insert performance

Data Count (million)	ULID Insert Time	UUID Insert Time
1	37880 ms	28790 ms
2	38939 ms	31934 ms
3	37880 ms	34663 ms
4	37818 ms	38347 ms
5	37804 ms	39532 ms
6	38067 ms	40795 ms
7	37372 ms	54567 ms
8	40082 ms	62473 ms
9	38487 ms	75378 ms
10	38533 ms	84585 ms

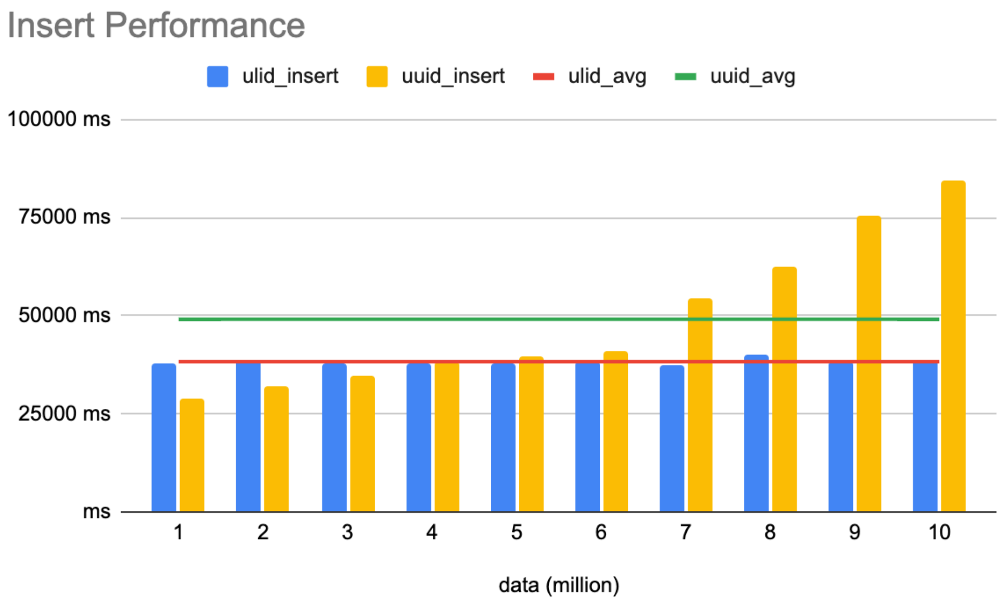

Figure 2 Insert performance

3.2 Find Performance

Table 2 shows the amount of time it took to locate 1 million records in progressively increasing data counts using UUID and ULID as primary keys. The data indicates that there is not much difference between the time required to find data using UUID and ULID. If there is a difference, it is insignificant. Figure 3 provides a better representation of this, showing that the time difference between using UUID and ULID as primary keys for finding data is minimal.

Average time taken for finding data using UUID primary keys was 628 ms while using ULID it was 630 ms.

Table 2 Find performance

Data Count (million)	ULID Find Time	ULID Find Time
1	107 ms	114 ms
2	160 ms	176 ms
3	256 ms	315 ms
4	422 ms	416 ms
5	477 ms	506 ms
6	589 ms	615 ms
7	791 ms	830 ms
8	1073 ms	900 ms
9	1122 ms	1154 ms
10	1304 ms	1257 ms

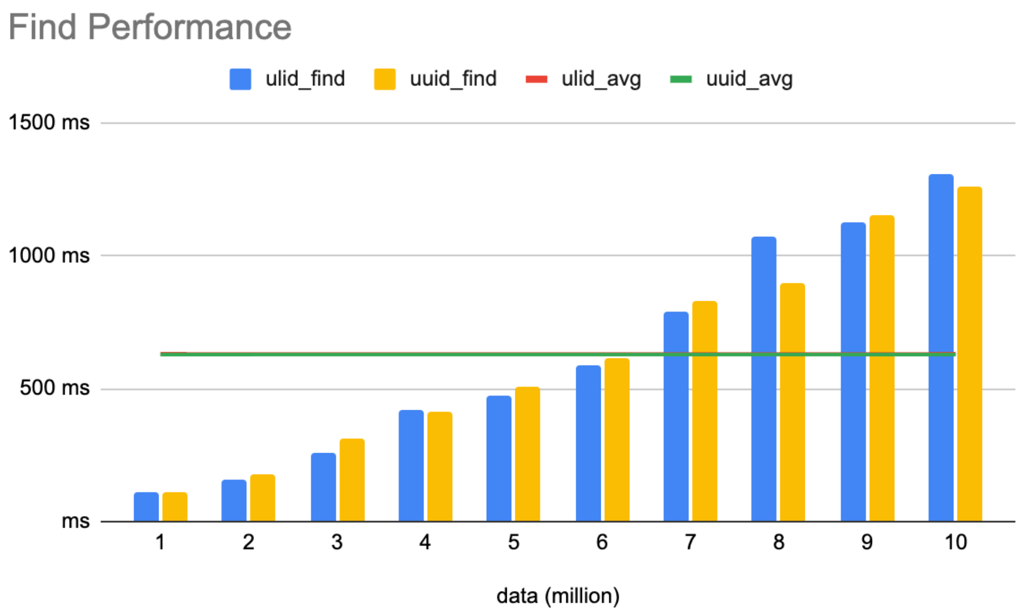

Figure 3 Find performance

3.3 Update Performance

Table 3 shows the amount of time needed to update 1 million records while incrementally increasing data counts using UUID and ULID as primary keys. Similar with find data performance, update data performance using UUID and ULID as primary keys only have a slight difference. While in average ULID performs a slightly better than UUID, the difference is negligible as shown on Figure 4. The average time taken for update date using UUID as primary keys was 577 ms and using ULID it was 556 ms.

Table 3 Update performance

Data Count (million)	ULID Update Time	UUID Update Time
1	402 ms	450 ms
2	159 ms	181 ms
3	253 ms	346 ms
4	422 ms	403 ms
5	507 ms	488 ms
6	559 ms	575 ms
7	675 ms	682 ms
8	770 ms	790 ms
9	872 ms	888 ms
10	938 ms	971 ms

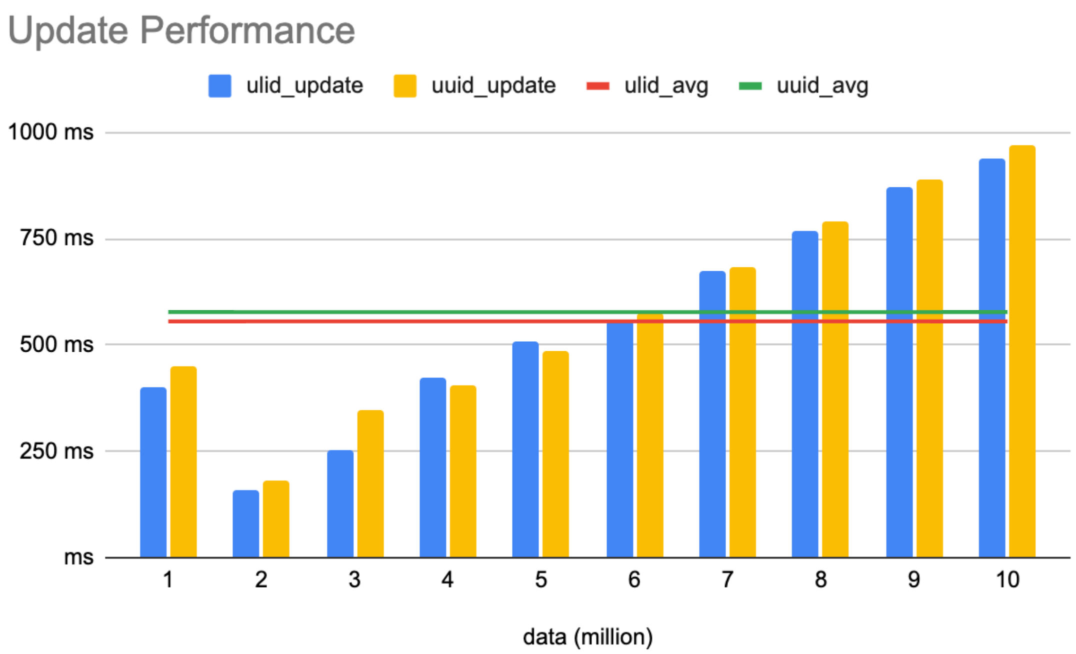

Figure 4 Update performance

3.4 Join Performance

Table 4 shows the time taken to perform a join operation on two tables while progressively increasing the data count using UUID and ULID as primary keys. As the data count increases, ULID performs progressively better than UUID, as displayed in Figure 5. The average time taken to perform a join operation using UUID as primary keys was 15672 ms, while using ULID, it was 13848 ms. The better performance of ULID primary keys is possibly due to the shorter representation of ULID compared UUID which causes faster string matching that happen a lot in join operation.

Table 4 Join performance

Data Count (million)	ULID Join Time	UUID Join Time
1	2588 ms	2864 ms
2	4903 ms	5417 ms
3	7404 ms	7983 ms
4	9769 ms	10791 ms
5	12897 ms	13424 ms
6	14435 ms	16517 ms
7	17494 ms	19482 ms
8	20007 ms	23526 ms
9	23105 ms	26318 ms
10	25880 ms	30402 ms

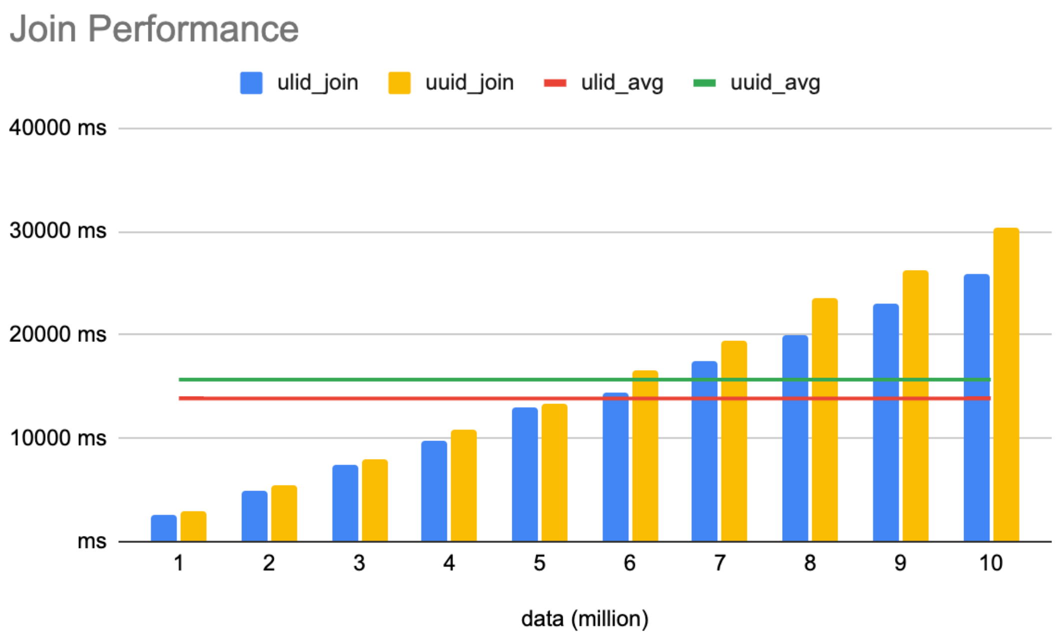

Figure 5 Join performance

4. CONCLUSIONS

The results of the experiment show sthat using ULID as primary keys generally results in better performance compared to UUID. This difference in performance is especially apparent in insert and join operations, while find and update operations have similar results.

The performance difference in the insert operation is due to the ordered nature of ULID. ULID is designed to be lexicographically sortable, so not a lot of B-Tree balancing operation is done when performing an insert. On the other hand, UUID generates random IDs, resulting in a lot of B-Tree balancing operation being done behind the scenes.

As for the better performance in join operation and slightly better performance in update operation, it is due to the shorter representation of ULID compared to UUID. The shorter representation of ULID leads to string comparation, which leads to faster join and update operations.

In conclusion, the study shows that using ULID as primary keys in a PostgreSQL database can result in better performance compared to using UUID. The ordered nature and shorter representation of ULID lead to a reduction in B-Tree balancing operation and faster string matching, resulting in faster operations overall.

REFERENCES

[1] L. Jiang and F. Naumann, “Holistic primary key and foreign key detection,” *J. Intell. Inf. Syst.*, vol. 54, no. 3, pp. 439–461, Jun. 2020, doi: 10.1007/s10844-019-00562-z.

[2] P. Leach, M. Mealling, and R. Salz, “A universally unique identifier (uuid) urn namespace,” 2005.

[3] H. Aftab, K. Gilani, J. Lee, L. Nkenyereye, S. Jeong, and J. Song, “Analysis of identifiers in IoT platforms,” *Digit. Commun. Netw.*, vol. 6, no. 3, pp. 333–340, Aug. 2020, doi: 10.1016/j.dcan.2019.05.003.

[4] Politechnika Rzeszowska and M. Penar, “Performance analysis of write operations in identity and UUID ordered tables,” *Sci. J. Rzesz. Univ. Technol. Ser. Electrotech.*, pp. 81–96, 2020, doi: 10.7862/re.2020.6.

[5] J. Pebrianto, “Perbandingan Kecepatan Baca Dan Tulis Data Pada Mysql Menggunakan Primary Key Auto Increment Dengan Universally Unique Identifier (UUID),” *Media J. Inform.*, vol. 14, no. 2, p. 86, Dec. 2022, doi: 10.35194/mji.v14i2.2681.

[6] Suyash, I. Ilin Alexander, and R. Pereira, “ulid/spec.” ULID, Mar. 25, 2023. Accessed: Mar. 26, 2023. [Online]. Available: https://github.com/ulid/spec

[7] D. Crockford, “Base 32,” *Douglas Crockford*, Mar. 04, 2019. https://www.crockford.com/base32.html (accessed Mar. 26, 2023).