**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: ***[1]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, select, 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, performa, ULID, UUID, kunci utama*

***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, select, update, and join. The results show that ULID generally outperforms UUID as the primary key in databases.*

***Keywords—*** *database, performance, ULID, UUID, primary key*

## 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. Numerous studies have attempted to compare the performance of SQL and NoSQL databases [6]–[8], but the results are inconclusive as the choice of database technology largely depends on the specific use case and data storage requirements, including indexing and data storing strategies, which may differ significantly between the two types of databases.

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 [9]. 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 [10]. 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.

PostgreSQL is an open source object-relational database system (ORDBMS), that supports features such as secondary indexing, sorting, and range queries [11]. In addition to those features, PostgreSQL also supports transactions, concurrency control, and referential integrity, making it a powerful tool for managing complex data sets. It also supports a variety of programming languages and interfaces, making it highly extensible and flexible. In addition to those features, PostgreSQL also supports transactions, concurrency control, and referential integrity, making it a powerful tool for managing complex data sets. It also supports a variety of programming languages and interfaces, making it highly extensible and flexible. In PostgreSQL, there are various types of indexes, including B-tree, Hash, GiST, SP-GiST, and GIN [12]. However, for the purposes of this paper, we will adopt the default B-tree as our chosen index strategy.

We evaluate UUID and ULID performance in four operations: insert, select, 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*

```yaml
version: '3.7'

services:
  postgres:
    container_name: postgres
    image: postgres
    environment:
      POSTGRES_USER: ${POSTGRES_USER:-postgres}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD:-postgres}
      PGDATA: /data/postgres
    volumes:
       - postgres:/data/postgres
    ports:
      - "5432:5432"
    networks:
      - postgres
    deploy:
      resources:
        limits:
          cpus: 1
          memory: 4G
    restart: unless-stopped

networks:
  postgres:
    driver: bridge

volumes:
    postgres:
```

Figure 1 Docker configuration for PostgreSQL

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 configuration as shown in Figure 1. The CPU resource limit was chosen because the experiment will only be run in a single thread, thus only using 1 CPU. While the memory resource limit was chosen because after testing for all test cases we found out that the memory never exceed 4 GB.

*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 2.

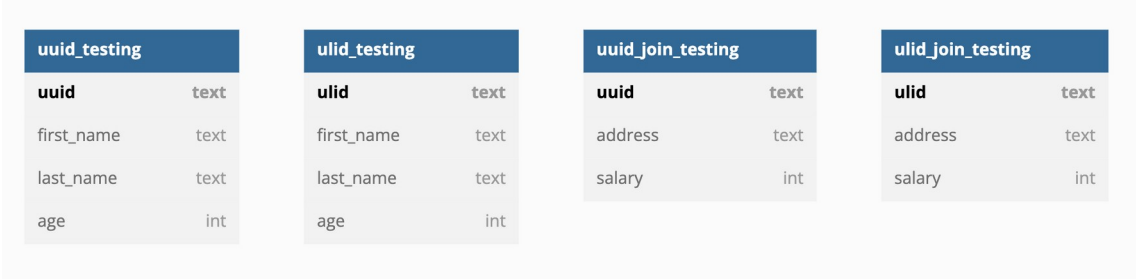


Figure 2 Testing database schema

*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, select, 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.
- Select: 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*

The results of our performance comparison study are presented in Table 1, which details the time taken to incrementally insert one million records using UUID and ULID as primary keys. Our findings indicate that, initially, UUID had faster insertion times than ULID, up until approximately three million records were added. However, after the addition of four million records or more, ULID's insertion time remained relatively constant, while UUID became increasingly slower, as depicted in Figure 3. The average insertion time for UUID was 49,106 ms, whereas ULID had an average insertion time of 38,286 ms.

The superior performance of ULID in terms of faster and more consistent insertion times can be attributed to its lexicographically ordered nature. As a result, when new data is inserted, there is less B-Tree balancing required, leading to a more efficient insertion process. Conversely, UUID has a random nature, which means that each new data ID is not lexicographically ordered, resulting in a more significant amount of B-Tree balancing. This can cause a slower insertion process, especially as the number of records grows.

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 |

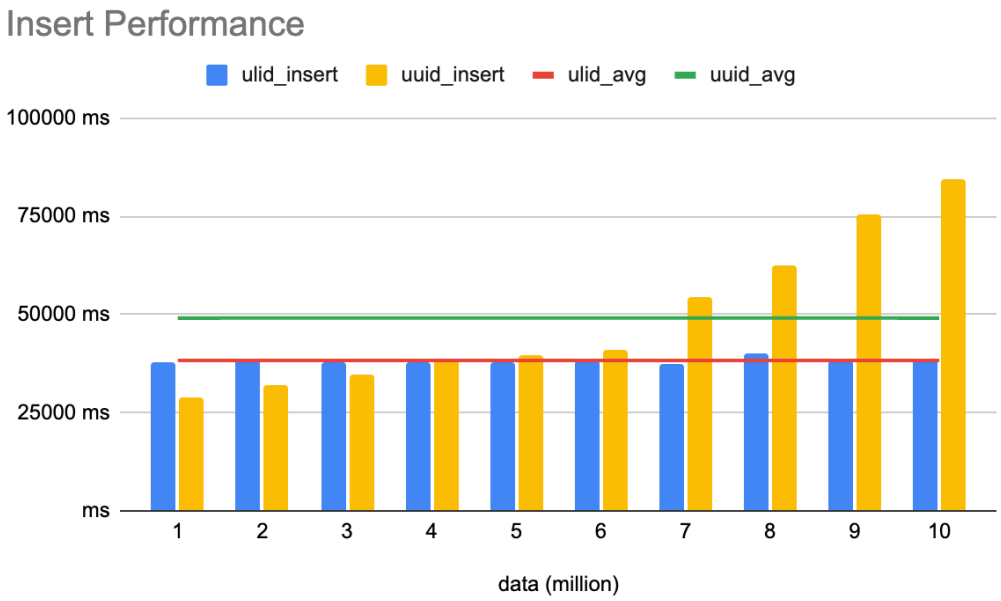


Figure 3 Insert performance

*3.2 Select Performance*

In our investigation of the performance of using UUID and ULID as primary keys for locating records within progressively increasing data counts, Table 2 illustrates the time required to select one million records using either primary key type. The data suggests that the time difference between using UUID and ULID for data retrieval is minimal. In fact, the results indicate that there is not much difference between the two methods, and any difference that is present is insignificant. Figure 4 provides a more detailed visualization of this finding, showing that the difference in the time required for selecting data using UUID and ULID as primary keys is negligible.

We found that the average time required to select data using UUID as a primary key was 628 ms, while it was 630 ms for ULID as the primary key. These findings suggest that both primary key types perform similarly in terms of data retrieval. This similarity in performance

can be attributed to the fact that both UUID and ULID are designed to be globally unique identifiers, optimized for fast lookup operations. Although they differ in their structure and method of generating unique identifiers, they are both highly efficient when used for locating records.

Table 2 Select performance

| Data Count (million) | ULID Select Time | ULID Select 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 |

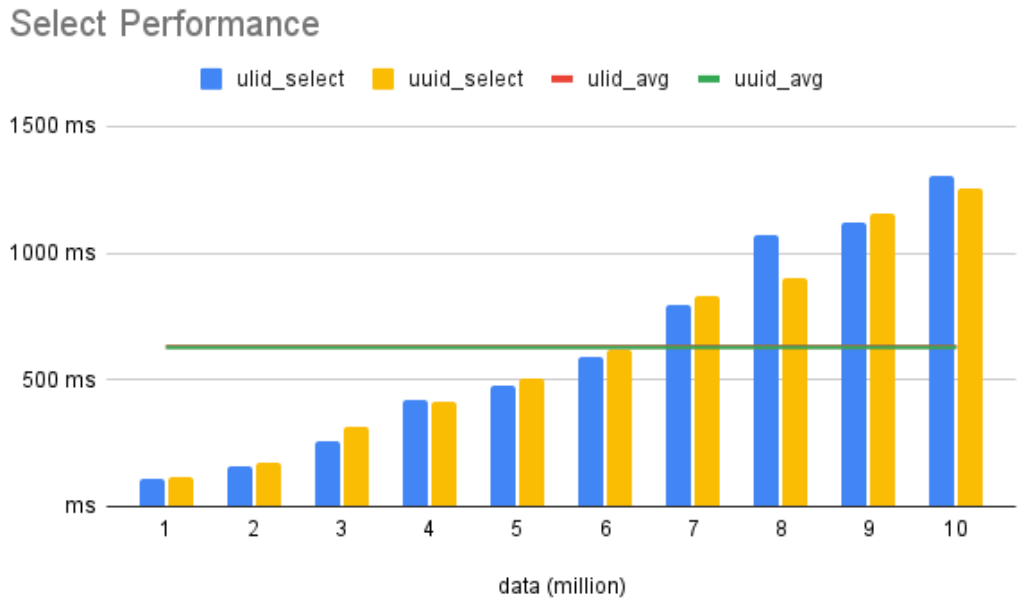


Figure 4 Select performance

*3.3 Update Performance*

We examined the performance of using UUID and ULID as primary keys for updating records within progressively increasing data counts, Table 3 presents the time required to update one million records using each primary key type. Our findings suggest that the difference in update performance between using UUID and ULID as primary keys is minimal. While ULID performs slightly better than UUID on average, the difference is negligible, as demonstrated by Figure 5.

Specifically, the average time required to update records using UUID as a primary key was 577 ms, while using ULID it was 556 ms. These results indicate that both UUID and ULID are highly efficient for updating records, and the difference in performance between them is insignificant.

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 |

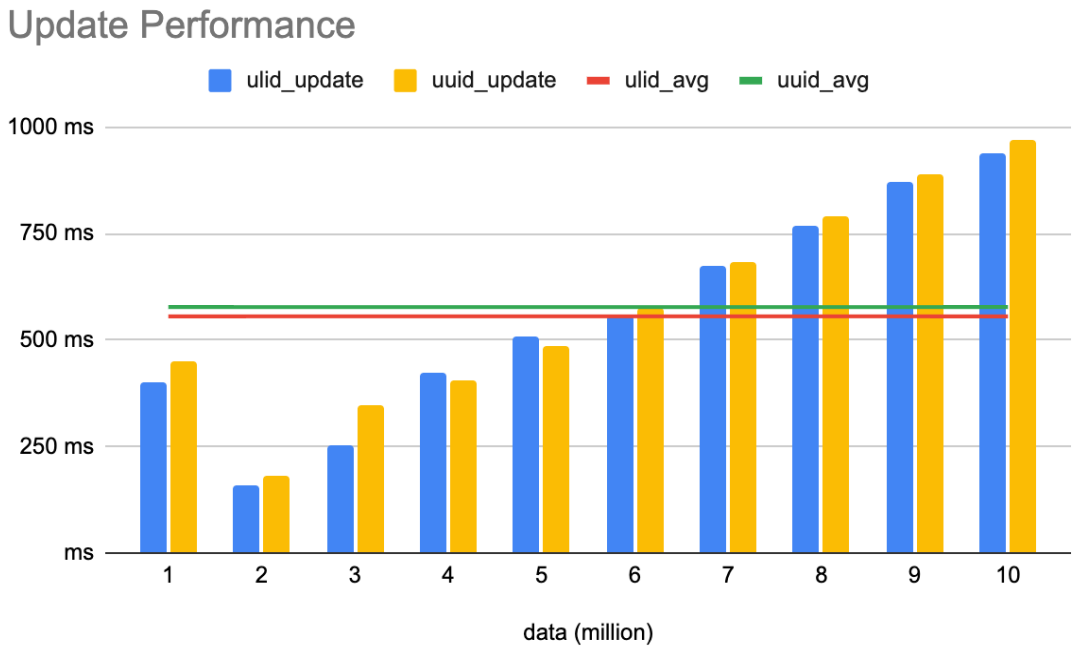


Figure 5 Update performance

*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 6. 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 |

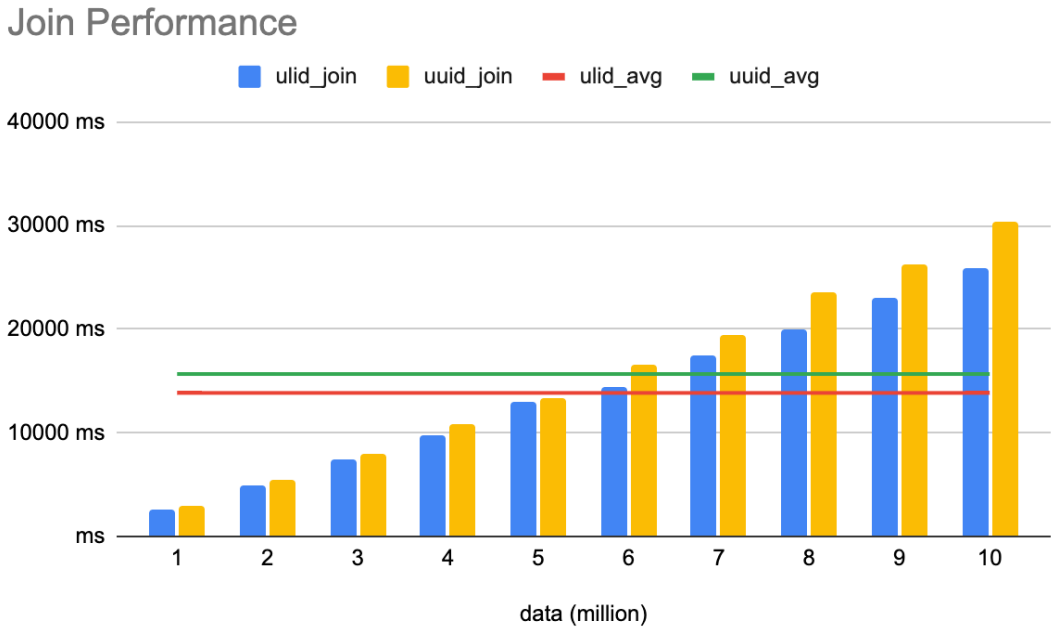


Figure 6 Join performance

## 4. CONCLUSIONS

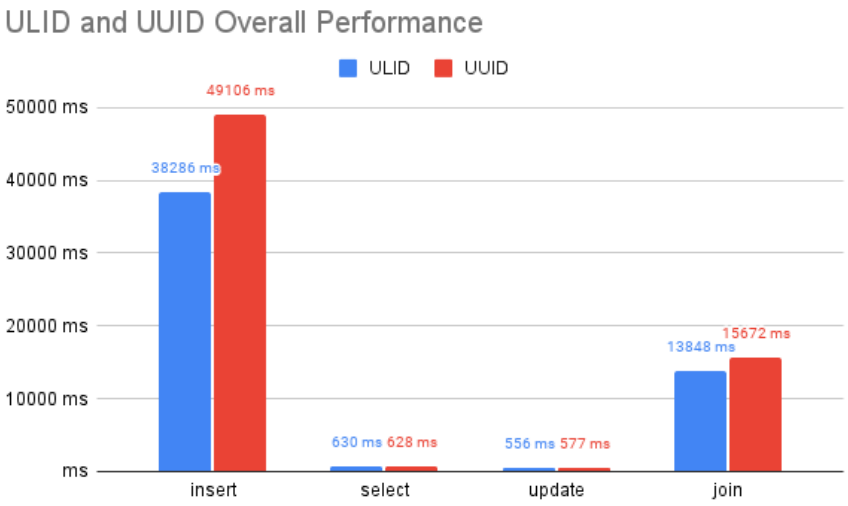


Figure 7 ULID and UUID overall performance

The overall experiment results in Figure 7 shows that 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.

ULID's shorter representation also contributes to better performance in join and update operations. The shorter representation of ULID results in faster string comparison, leading to faster operations overall.

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] W. Khan, T. Kumar, Z. Cheng, K. Raj, A. M. Roy, and B. Luo, "SQL and NoSQL Databases Software architectures performance analysis and assessments -- A Systematic Literature review." arXiv, Sep. 14, 2022. Accessed: May 04, 2023. [Online]. Available: http://arxiv.org/abs/2209.06977

[7] Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," in *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, Victoria, BC, Canada: IEEE, Aug. 2013, pp. 15–19. doi: 10.1109/PACRIM.2013.6625441.

[8] Z. Parker, S. Poe, and S. V. Vrbsky, "Comparing NoSQL MongoDB to an SQL DB," in *Proceedings of the 51st ACM Southeast Conference*, Savannah Georgia: ACM, Apr. 2013, pp. 1–6. doi: 10.1145/2498328.2500047.

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

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

[11] Makris *et al.*, "Performance Evaluation of MongoDB and PostgreSQL for Spatio-temporal Data," *EDBTICDT Workshop*, 2019.

[12] A. Borodin, S. Mirvoda, I. Kulikov, and S. Porshnev, "Optimization of Memory Operations in Generalized Search Trees of PostgreSQL," in *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation*, S. Kozielski, D. Mrozek, P. Kasprowski, B. Małysiak-Mrozek, and D. Kostrzewa, Eds., in Communications in Computer and Information Science, vol. 716. Cham: Springer International Publishing, 2017, pp. 224–232. doi: 10.1007/978-3-319-58274-0_19.