Jurnal Teknik Informatika (JUTIF)
P-ISSN: 2723-3863
E-ISSN: 2723-3871
Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

Benchmarking Relational and Array-Based Models for Genealogical Data Storage in PostgreSQL

Suwanto Raharjo*[1], Ema Utami[2]

[1]Faculty of Science and Information Technology, Universitas AKPRIND Yogyakarta, Indonesia
[2]Faculty of Computer Science, Universitas Amikom Yogyakarta, Indonesia

Email: [1]wa2n@akprind.ac.id

Received : Dec 24, 2025; Revised : Feb 9, 2026; Accepted : Feb 13, 2026; Published : Feb 15, 2026

Abstract

Genealogical information systems manage inherently hierarchical data structures that represent family relationships across multiple generations. Traditional implementations predominantly rely on normalized relational database designs using junction tables to model parent–child relationships. While this approach ensures strong referential integrity, it often incurs substantial performance overhead due to complex join operations during deep hierarchical traversal. Recent versions of PostgreSQL provide native support for array data types This study compares two genealogical database models implemented in PostgreSQL: a normalized relational model using a junction table and a denormalized model that stores child identifiers directly as UUID arrays. To evaluate their performance, we conducted controlled benchmarking experiments using synthetically generated genealogical datasets with varying generational depth and branching patterns. The comparison focuses on storage efficiency, recursive traversal performance, and write operation costs under realistic hierarchical workloads. Results obtained from a large-scale dataset containing more than 7 million individual records show that the UUID array–based model reduces disk space usage by 31%. During deep recursive traversal involving over 12 million nodes at the tenth generation, the array-based model demonstrates improved data locality, leading to a 5.2% reduction in execution latency and 7% fewer shared buffer accesses compared to the relational model. Interestingly, contrary to common expectations in normalized database design, the array-based model achieves 22% faster single-insert performance because it avoids foreign key validation and multiple index updates. This improvement comes with slightly higher write amplification, reflected in a 6.6% increase in buffer usage due to PostgreSQL's multi-version concurrency control mechanism. These findings contribute to the field of Informatics by providing empirical evidence on how database internal mechanisms influence performance trade-offs in hierarchical data management, offering guidance for designing scalable and read-efficient information systems beyond genealogical applications.

Keywords : *genealogical database, postgresql, hierarchical data, uuid array, tree traversal*

1. INTRODUCTION

Genealogical information systems are designed to represent and manage kinship relationships among individuals, such as parent–child ties, marriages, and family structures that span multiple generations. By nature, this type of data forms a hierarchy that tends to grow more complex over time. As family records span across generations, the database must store information accurately while also supporting efficient traversal and enabling the reconstruction of kinship relationships. In practice, this complexity creates challenges in database design, particularly when a system must strike a balance between data integrity and query performance.

Over the past decade, genealogical tracking systems have undergone significant changes. These shifts have been driven largely by large-scale efforts to digitize family archives, along with the growing popularity of at-home DNA testing services. Shan and Luther [1] point out that modern genealogical platforms no longer function simply as repositories for names and dates; instead, they are expected to

Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

support complex collaboration and large-scale data integration. Such developments introduce new technical demands, especially for database architectures that must store data efficiently while maintaining fast and reliable access to continuously expanding datasets.

Relational databases commonly represent family trees using adjacency lists and junction tables. While this approach provides strong data integrity, its efficiency tends to decline when dealing with deep hierarchical queries [2]. A key reason is the reliance on Recursive Common Table Expressions (CTEs), which involve repeated self-join operations. As the depth of the hierarchy increases, these operations place greater strain on CPU and I/O resources, potentially degrading overall query performance [3]. This study addresses these performance challenges by exploring alternatives beyond traditional relational structures.

These challenges become even more pronounced in modern genealogical systems that extend beyond basic kinship records to include temporal and geographic information. For instance, the China Family Tree Geographic Information System demonstrates that integrating family data across historical periods and geographic regions introduces substantial performance and interoperability issues [4]. This observation suggests that performance limitations in genealogical systems are not solely the result of application-layer design, but are also strongly influenced by the structure and efficiency of the underlying database schema.

Although relational databases remain the primary backend technology for many genealogical systems, much of the existing research places greater emphasis on application features and user experience than on database-level performance. As noted by Kleppmann [5], there is a persistent structural mismatch between the flat nature of relational schemas and the inherently hierarchical characteristics of genealogical data. When nested family relationships are forced into fully normalized formats, the recursive queries required for traversal become increasingly demanding, particularly under read-heavy workloads that are common in genealogical applications.

Although genealogical data modeling has received increasing research attention, empirical evaluations at the database-engine level remain relatively limited. Most prior studies tend to emphasize application-layer improvements or visualization techniques rather than conducting systematic analyses of internal database performance. Existing literature provides only a small number of controlled benchmarks comparing normalized relational schemas with denormalized array-based approaches in PostgreSQL, particularly for large-scale datasets that resemble real-world genealogical structures. Without sufficient empirical evidence, database designers often face uncertainty when selecting modeling strategies that can sustain performance under demanding hierarchical workloads.

Modern database systems have evolved beyond purely relational structures by adopting hybrid storage capabilities. PostgreSQL, as an object-relational database management system (ORDBMS), reflects this shift by offering native support for array data types as well as advanced indexing mechanisms such as Generalized Inverted Indexes (GIN) [6]. Comparative research by Čerešňák and Kvet [7] shows that reducing reliance on strictly relational representations can improve performance in certain data manipulation tasks by minimizing relational overhead. These PostgreSQL features make it possible to incorporate non-relational design advantages without sacrificing the reliability and consistency typically associated with relational models.

For example, when combined with GIN indexes, inclusion operators such as `@>` can significantly improve the efficiency of array-based element searches by reducing the need for repeated recursive joins. This approach represents a form of controlled denormalization aimed at improving read performance while keeping structural complexity manageable. In genealogical database design, developers can balance the strict consistency offered by normalization with the performance benefits gained from denormalized array representations. Given that genealogical archives are generally read-

Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

intensive, the performance gains of array-based structures can, in many cases, justify a relaxation of strict normalization requirements.

Another challenge in genealogical databases arises from the persistent and distributed nature of individual identification across systems. Commonly used auto-increment identifiers become impractical due to synchronization limitations between systems. To address this issue, this study adopts Universally Unique Identifiers (UUIDs) as primary keys. In the proposed model, UUIDs serve a dual role: ensuring global uniqueness across distributed datasets and acting as the fundamental elements within the evaluated array-based structures (UUID[]).

To this day, GEDCOM remains the dominant standard for genealogical data exchange, although its continued use is driven more by historical compatibility than by technical efficiency [8]. Originally designed for serial data transfer, GEDCOM is not inherently optimized for internal database storage. This study clearly distinguishes between data exchange formats and internal storage design, adopting an approach more closely aligned with modern frameworks such as GEDCOM X, which emphasize the use of persistent identifiers to improve interoperability [9].

Theoretical perspectives, such as those presented in the study *"Nested Parquet Is Flat"* [10], highlight how choices in data representation can significantly affect system performance. This consideration is particularly relevant in the Indonesian context, where distinctive family structures—such as extended kinship networks and polygamous relationships—can result in genealogical graphs of greater complexity than those typically assumed in conventional genealogical modeling approaches.

This study approaches genealogical data modeling from a database-engine perspective, focusing on how PostgreSQL's internal storage architecture and indexing strategies influence performance in deeply hierarchical structures. This perspective emerged from practical observations during large-scale testing, where query efficiency was often shaped more by storage design decisions than by application-layer optimizations.

From an Informatics perspective, selecting appropriate data modeling strategies is essential for designing systems that must operate efficiently under large-scale hierarchical workloads. Applications such as social network analysis and knowledge representation similarly depend on database architectures that balance scalability and query performance. Therefore, this study presents a controlled comparison between two PostgreSQL modeling approaches: a normalized Adjacency List structure and a denormalized array-based design. By evaluating both models in terms of storage efficiency and traversal performance, this research seeks to provide both theoretical insight and practical guidance for managing hierarchical data in modern information systems.

2. RELATED WORK

2.1. Genealogical Data as Hierarchical Structures

Genealogical data can be viewed as a directed graph, $G = (V, E)$ where V represents the set of individuals (vertices) and denotes kinship relationships (edges). In most situations, this structure takes the form of a Directed Acyclic Graph (DAG), since each individual is linked to two biological parents, while lineage itself is typically organized in a hierarchical manner.

The efficiency of navigating such structures is strongly influenced by the depth of the tree (d) and the average branching factor (b). In conventional relational database implementations, traversing ancestors or descendants often exhibits time complexity proportional to $O(d)$. This behavior largely stems from the reliance on recursive or iterative indexing mechanisms commonly used in relational systems.

Recent studies, including graph-oriented approaches such as ProgenyAI proposed by Aceto et al. [11], have demonstrated the effectiveness of non-relational models for pedigree analysis. Even so,

Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

hybrid relational implementations continue to be appealing in many practical scenarios. PostgreSQL, for instance, offers a mature ecosystem, proven stability, and strong integration capabilities, making it a pragmatic choice for large-scale enterprise environments.

2.2. Relational Modeling of Hierarchical Data

Genealogical information systems have long been a subject of interest within information systems research. Traditional implementations typically rely on normalized relational database designs, where individual entities and kinship relationships are stored in separate tables. Parent–child relationships are explicitly defined through foreign key constraints, allowing referential integrity to be enforced in a structured and consistent way.

While this design simplifies the management of data consistency, separating entities across tables can introduce significant join overhead during query execution. Min et al. [12], in their study of large-scale genealogical systems, observed that performance limitations occur in visualization and also in the reconstruction of hierarchical structures from relational tables.These challenges become increasingly pronounced as the number of nodes grows into the hundreds of thousands or even millions, where repeated join operations can substantially reduce query efficiency.

2.3. Hierarchical Data Modeling Strategies

Celko [3] provides a comprehensive overview of hierarchical data modeling techniques in relational databases, identifying three classical approaches that continue to be widely referenced. The first is the Adjacency List model, in which each record stores a reference to its parent. This design is relatively flexible for insert and update operations, but it often encounters difficulties during subtree traversal due to its reliance on Recursive Common Table Expressions (CTEs).

To improve read efficiency, the Nested Sets approach is used to represent hierarchical relationships. Although this model enables faster subtree retrieval, write operations can become expensive, as inserting a single node may require updating all boundary values. Another alternative is Path Enumeration, where each node stores its complete path from the root (for example, “1.5.12”). This method strikes a balance between read and write efficiency, but it introduces maintenance complexity when structural changes occur within the hierarchy.

In contrast to these traditional methods, this study examines a hybrid approach based on a Materialized List of Children implemented using native array features. This design is expected to introduce a different set of performance trade-offs compared to classical hierarchical modeling strategies.

2.4. Normalization versus Denormalization

There is an inherent tension between the goals of normalization, which emphasizes data integrity, and denormalization, which prioritizes query performance. Normalization seeks to reduce redundancy and preserve consistency, whereas denormalization focuses on improving read efficiency by minimizing join operations. Taipalus [13] notes that such logical design choices influence query performance and also contribute to system complexity and energy consumption. In genealogical information systems, where read operations typically far outweigh write operations, denormalization can therefore become an appealing strategy.

Min et al. [14] show that queries involving multiple joins over large hierarchical datasets can suffer significant performance degradation. These findings point to the need for more compact data representations, such as structured or composite data types. At the same time, contemporary genealogical research increasingly integrates diverse data sources, ranging from digitized archives to

Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

genomic information [1], which places additional demands on storage solutions to support both scalability and flexibility.

Within the PostgreSQL ecosystem, this discussion often extends to comparisons between strictly relational schemas and more flexible data formats such as JSONB. However, Turutin and Puzevich [15] report that JSONB indexing can introduce considerable performance overhead when compared to typed scalar representations. For this reason, the present study focuses on PostgreSQL's native array data type (UUID[]), which offers a practical balance between schema flexibility and efficient binary storage.

2.5. PostgreSQL Arrays and Internal Storage

PostgreSQL provides advanced support for array data types along with appropriate indexing mechanisms, making it a suitable platform for evaluating array-based hierarchical modeling. To enable efficient element searches, PostgreSQL employs the Generalized Inverted Index (GIN). As described by Toktomusheva [16], GIN uses an inverted index structure in which each array element is associated with a posting list of row identifiers. This design allows element lookup operations to approach logarithmic complexity, offering clear performance advantages over sequential scans, particularly for large datasets.

One important technical consideration is PostgreSQL's Oversized-Attribute Storage Technique (TOAST). TOAST manages large column values by compressing them and storing them externally once their size exceeds the page threshold, typically between 2 and 8 KB [17]. While external storage can introduce additional retrieval latency, this mechanism remains well suited to genealogical data. A single UUID occupies 16 bytes, which means that approximately 128 child identifiers can still be stored within a 2 KB inline storage threshold. Given the biological and demographic constraints common in genealogical contexts, the number of children per individual is unlikely to exceed this limit. As a result, most parent–child relationships represented as UUID arrays are expected to remain inline and accessible with minimal overhead. This behavior contrasts with domains such as time-series data, where array sizes may grow unpredictably and significantly increase retrieval costs.

2.6. Research Gap

Based on the literature review conducted, it becomes evident that there are still meaningful research opportunities in the area of genealogical database systems. While extensive work has been done on visualizing family trees and optimizing algorithms, our understanding of how database storage behaves at the internal level remains limited. In particular, there is a lack of empirical evidence examining the behavior of GIN indexing, as well as PostgreSQL's TOAST storage mechanism, when they are used to manage complex and hierarchical UUID arrays. As family trees continue to grow deeper and develop more branches, a clearer understanding of how these internal mechanisms scale becomes increasingly important. This study aims to address this gap by comparing the traditional junction-table model with a UUID array–based approach, thereby providing the empirical data needed to inform more effective database design at the storage level.

3. METHOD

This section describes the methodology used to compare two data modeling strategies in PostgreSQL: a normalized relational schema and a denormalized model based on UUID arrays (UUID[]). The experimental setup was carefully designed so that any observed differences in performance could be attributed to the database modeling approach itself, rather than to external factors. To support this objective, both schemas were populated with identical synthetic datasets. The data were validated to ensure referential integrity and were indexed appropriately to preserve semantic equivalence between the two models.

Jurnal Teknik Informatika (JUTIF)
P-ISSN: 2723-3863
E-ISSN: 2723-3871
Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

The complete experimental workflow is summarized in Figure 1. The process begins with schema design and dataset generation, followed by a systematic phase of validation, workload execution, and precise performance measurement. The workflow concludes with a comparative analysis to evaluate how each model performs when subjected to high workload conditions.

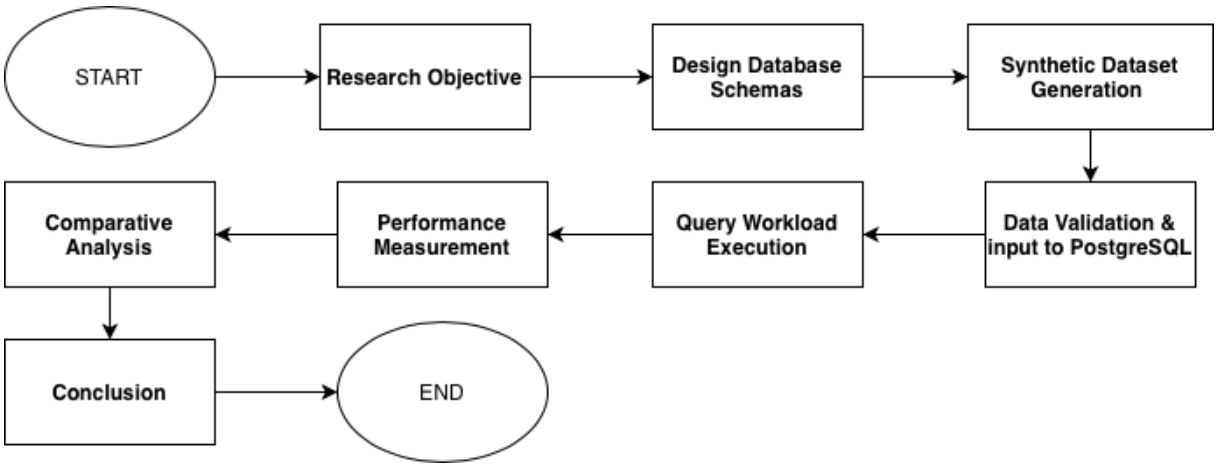

Figure 1. Experimental workflow for benchmarking relational and array-based genealogical database models in PostgreSQL.

3.1. Research Design

A quantitative experimental approach was employed to obtain accurate measurements of latency and resource utilization, aspects that are not always fully captured through theoretical estimation alone. Initial testing showed that uncontrolled workload variability could lead to inconsistencies in performance evaluation. For this reason, the experimental design follows the principles of systematic performance analysis outlined by Jain [18], treating the database schema as the primary variable while keeping all other environmental factors constant. To reduce potential bias in the benchmarking process [19], both the Adjacency List model and the UUID array–based model were evaluated using identical datasets and hardware configurations. This design ensures that any observed performance differences can be attributed directly to the modeling strategies being compared, rather than to external influences.

3.2. Database Models Under Evaluation

In this study, two different approaches are evaluated. The first is a **Normalized Relational Model**, which follows the classical Adjacency List pattern using three interconnected tables. Individual information is stored in the *person* table, while family units are defined in the *family* table. To represent the one-to-many relationship between parents and children, a *family_child* junction table is used. Although this structure is effective at enforcing strict referential integrity through foreign keys, it requires the database to perform explicit join operations whenever parent–child relationships need to be traversed.

The second approach is a **Denormalized Array-Based Model**. This design simplifies the schema by adopting the concept of a *Materialized List of Children*. The *person* table remains identical to that of the first model, but the *family* table is extended with a *children_ids* column of type UUID[]. By storing child identifiers directly within the family row, this approach takes advantage of data locality. In practical terms, child data is stored physically alongside parent data within the same storage tuple, which can significantly reduce the need for costly join operations.

Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

3.3. Synthetic Dataset Generation

Although the datasets used in this study are synthetic, their generation follows the strict integrity principles defined in the SyntheRela framework [20]. The parameters were intentionally grounded in real-world statistics to ensure that the benchmark remains representative. Given that Indonesia's fertility rate has shifted to approximately 2.18 [21], two scenarios based on Poisson distributions were employed to evaluate different structural pressures.

Scenario A (λ = 3) represents a "modern family" structure and is used to observe array performance under relatively low branching conditions. In contrast, **Scenario B (λ = 6)** simulates "historical families," where larger branching factors are used to test the scalability of UUID arrays before they are potentially moved to external TOAST storage. The difference in scale between these two scenarios is substantial; as shown in Table 1, the λ = 6 configuration generates more than seven million records, providing a suitable environment for evaluating large-scale genealogical workloads..

Table 1. Dataset Size Specifications (Row Count)

Scenario	Parameter	Person Table (Individuals)	Family Table (Families)
A (Small)	λ=3	11,906	3,927
B (Large)	λ=6	7,107,539	1,184,286

For entity identification, Universally Unique Identifiers version 4 (UUIDv4) were used. Although the PostgreSQL version employed in this study already supports time-ordered UUIDv7 [22], UUIDv4 was deliberately chosen to simulate random memory access patterns. This choice places additional pressure on B-tree index structures by removing the locality advantages typically associated with time-ordered identifiers. By disabling time-based ordering optimizations, the experiment seeks to observe architectural performance differences between the relational and array-based models under more demanding, worst-case indexing conditions.

3.4. Experimental Environment

To ensure consistent performance measurements and to reduce variability caused by network-related factors, all experiments were carried out within a single hardware environment. The system specifications are summarized in Table 2. Benchmark testing was conducted on an ASUS ZenBook equipped with an AMD Ryzen 7 5700U processor and 16 GB of memory. Of this total memory, 4 GB was allocated to PostgreSQL's *shared_buffers* configuration to support efficient processing of large-scale database operations.

Table 2. Experimental Environment Specifications

Component	Specification
Device	ASUS ZenBook UX325UA
Processor	AMD Ryzen 7 5700U (8 cores, 16 threads) @ 1.8 GHz
Memory	16 GB LPDDR4x (4 GB allocated to PostgreSQL *shared_buffers*)
Storage	NVMe M.2 SSD
Operating System	Windows 11 Home 64-bit (Build 26100)
DBMS	PostgreSQL 18.1 (Nov 2025)
DBMS Features	Asynchronous I/O enabled, JIT compilation enabled

Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

Several steps were taken to stabilize the testing environment. The system was run in *Best Performance* mode, and non-essential background processes were minimized to reduce potential variability in CPU usage. PostgreSQL configuration parameters were also tuned to better match modern hardware characteristics: *work_mem* was set to 16 MB, and *random_page_cost* was configured at 1.1 to reflect the high-speed access behavior of NVMe SSD storage. PostgreSQL 18.1 was chosen as the experimental platform based on recent comparative studies [23], which report superior memory management performance for read-heavy analytical workloads when compared with alternative database systems.

3.5. Query Workloads and Metrics

To assess the performance of each model under realistic usage conditions, a query workload was designed to reflect the read and write operations commonly encountered in genealogical data management [24]. The evaluation focused on three primary types of queries:

a. **Child Lookup (Read):** This operation retrieves the immediate list of children associated with a specific family node. It is primarily used to assess indexing efficiency and data retrieval performance for direct parent–child relationships.
b. **Descendant Traversal (Read):** This workload retrieves all descendant nodes across multiple generations using Recursive Common Table Expressions (CTEs). It evaluates how effectively each model handles deep hierarchical traversal and multi-level query execution.
c. **Insert New Child (Write):** This operation measures the cost of recording the addition of a new child. In the relational model (Model R), it involves inserting a new row into the *family_child* junction table. In contrast, in the array-based model (Model A), the operation updates an existing row in the *family* table by appending a UUID value to the corresponding array column.

4. RESULTS

In this section, we present the results of the performance benchmarks that were conducted. The primary objective of these experiments was to examine how the normalized relational model (Model R) and the array-based model (Model A) perform when confronted with deep hierarchical structures and large-scale datasets. The evaluation focuses on three key aspects: storage overhead, the speed of reading and traversing complex genealogical lineages, and the computational cost associated with write operations.

4.1. Validation of Hierarchical Data Depth

Before running the main benchmarks, a validation step was carried out to ensure that the synthetic dataset accurately reflected a deeply hierarchical structure. Using the larger dataset scenario ($\lambda = 6$), the distribution of nodes across generations was analyzed. The results show clear exponential growth, with the tenth generation containing 12,112,060 nodes. A full recursive traversal spanning all ten generations required processing a cumulative total of 14,537,140 nodes, as summarized in Table 3. These results provide a solid foundation for evaluating model performance under complex hierarchical conditions.

4.2. Comparison of Storage Efficiency

The total disk space consumption for both models at the $\lambda = 6$ scale is shown in Figure 2. Model R required 2,115 MB of storage, while Model A used only 1,460 MB, corresponding to a 31% reduction in disk usage. This difference in storage requirements reflects structural variations between the two modeling approaches, primarily due to the absence of a junction table and its associated indexing structures in Model A. This more compact representation has the potential to improve data retrieval efficiency when working with large-scale genealogical datasets.

Jurnal Teknik Informatika (JUTIF)
P-ISSN: 2723-3863
E-ISSN: 2723-3871
Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

Table 3: Node Distribution per Generation (Validation Sample λ=6)

Generation	Node Count	Cumulative Count
1	1	1
2	7	8
3	42	50
4	262	312
5	1,590	1,902
6	9,415	11,317
7	56,117	67,434
8	336,819	404,253
9	2,020,827	2,425,080
10	12,112,060	14,537,140
Total	**14,537,140**	

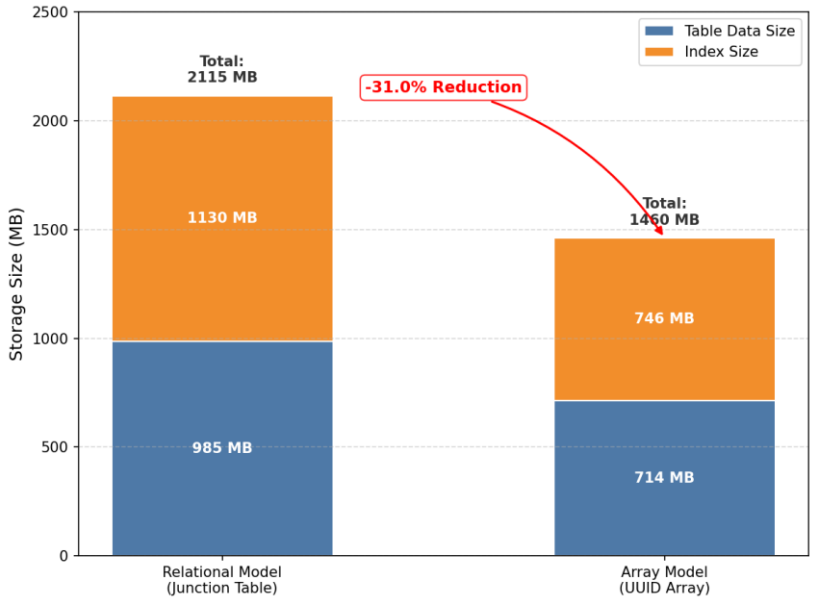

Figure 2. Storage Efficiency Comparison (λ=6)

4.3. Query Performance Analysis

The performance benchmark results are summarized across three main operational scenarios. In the **Child Lookup** tests shown in Table 4, both models exhibited broadly comparable performance, with Model R recording a slightly lower average execution time of 0.058 ms compared to Model A. During Recursive Traversal under the $\lambda = 6$ scenario, differences in buffer utilization became more apparent. Model A required approximately 93 million shared buffer accesses, whereas Model R involved approximately 100 million accesses, as illustrated in Figure 3.

The most differences were observed in **Write Operations**. As presented in Table 5 and Figure 4, Model A achieved an average execution time of 0.124 ms, whereas Model R recorded 0.159 ms, corresponding to a 22% reduction in write latency. However, buffer usage during write operations was

Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

slightly higher for Model A (25.8 buffer hits) than for Model R (24.2 buffer hits), indicating differing memory access patterns between the two approaches.

Table 4. Average Performance for Scenario 2 (Child Lookup)

Model Data	Time (ms)	Buffer Hits
Model R (Relational)	0.058	53
Model A (Array)	0.068	55

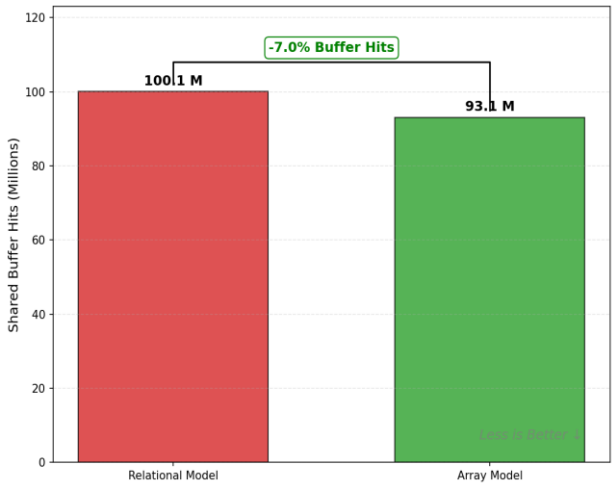

Figure 3: Comparison of total Shared Buffer Hits during deep recursive traversal (λ = 6).

Table 5. Write Performance (Insert Child))

Model Data	Time (ms)	Buffer Hits
Model R (Relational)	0.159	24.2
Model A (Array)	0.124	25.8

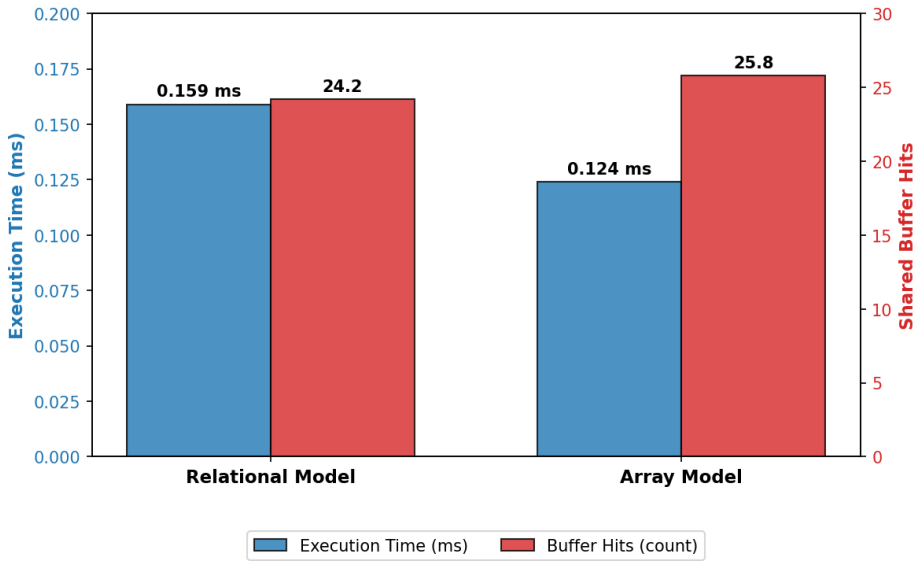

Figure 4: Write operation performance metrics.

5. DISCUSSION

The experimental results reveal several performance patterns that become more pronounced as the dataset grows in scale. While some findings align with the initial expectations, others—particularly those related to write performance—provide additional insight into PostgreSQL's internal mechanisms.

5.1. Interpretation of Storage and Indexing

The 31% reduction in storage usage observed in Model A can be explained by the structural differences between the two models. In Model R, each parent–child relationship is stored as a separate tuple, which includes a row header overhead of approximately 23 bytes. In contrast, Model A represents these relationships by appending 16-byte UUID elements to an existing array column. This approach reduces metadata overhead and increases storage density. The findings indicate that PostgreSQL's object-relational features can effectively minimize redundancy in hierarchical data representations.

5.2. Locality of Reference in Recursive Queries

These findings are consistent with earlier studies showing that normalized schemas can introduce performance overhead during hierarchical reconstruction due to repeated join operations [12]. Turutin and Puzevich [15] similarly observed that denormalized structures are able to reduce query latency by lowering relational complexity. The results of this study reinforce these observations by demonstrating how data locality influences database behavior at the internal engine level.

The performance differences observed during recursive traversal (Figure 2) are closely tied to data locality. Model R requires multiple index scans across separate tables, which increases shared buffer usage. In contrast, Model A stores child identifiers within the same physical tuple as the parent data, thereby reducing page transitions during traversal. This observation aligns with Celko's [3] discussion of the limitations of the Adjacency List model when handling deep hierarchical queries.

5.3. The MVCC Trade-off in Write Operations

The write performance results reveal a trade-off between execution speed and buffer utilization. Model A delivers faster write performance, largely because it avoids foreign key validation and the multiple index updates required by Model R. At the same time, Model A exhibits slightly higher buffer activity. This behavior can be explained by PostgreSQL's Multi-Version Concurrency Control (MVCC) mechanism. Updating an array column requires rewriting the entire row version [25, 26], which leads to increased memory I/O. As a result, while the array-based model reduces index maintenance overhead, it also introduces greater write amplification during update operations.

5.4. Practical Implications

The results of this study indicate that array-based representations can offer significant advantages for genealogical workloads that are dominated by read operations. These findings are consistent with design principles commonly associated with NoSQL systems [27], where denormalization is used to improve read performance. For large-scale genealogical platforms that require efficient data traversal and low-latency queries [1], array-based storage provides a practical solution that balances performance with structural complexity.

6. CONCLUSION AND FUTURE WORK

This study examined whether alternative data modeling strategies can improve the efficiency of genealogical databases. By comparing a normalized relational structure with an array-based representation in PostgreSQL, the research provides empirical evidence at the database engine level regarding performance trade-offs in hierarchical data management.

Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

Experimental results obtained from a dataset containing more than seven million records show that the array-based model offers clear advantages for read-dominant workloads. A 31% reduction in disk space usage was observed, reflecting a more compact representation of parent–child relationships without the need for junction tables and their associated indexes. In addition, the array-based model exhibited lower shared buffer usage during deep recursive traversal, indicating improved data locality when related entities are stored within the same physical tuple.

The analysis of write operations reveals a trade-off between execution speed and memory utilization. While the array-based model achieves faster insert performance by avoiding foreign key validation and multiple index updates, it also leads to a modest increase in buffer activity. This behavior is consistent with PostgreSQL's Multi-Version Concurrency Control (MVCC) mechanism, in which updates to array columns require rewriting the entire row, resulting in additional memory overhead.

These findings highlight important architectural considerations when selecting a database model for hierarchical data. Although normalized schemas remain well suited for systems that prioritize strict database-level integrity, array-based representations can deliver performance benefits for read-oriented genealogical applications, where efficient traversal is a critical requirement.

Future research may explore the scalability limits of array indexing mechanisms, particularly with respect to potential GIN index saturation. Comparative evaluations involving other PostgreSQL data types, such as JSONB, as well as native graph database systems, could further clarify the performance boundaries of hybrid relational approaches. In addition, investigating the impact of time-ordered identifiers, such as UUIDv7, on indexing behavior and write efficiency represents a promising direction for further study.

CONFLICT OF INTEREST

The authors declare that there is no conflict of interest between the authors or with research objects in this paper.

REFERENCES

[1] F. Shan and K. Luther, "Reexamining Technological Support for Genealogy Research, Collaboration, and Education," *Proceedings of the ACM on Human-Computer Interaction*, vol. 9, no. 2, pp. 1–33, May 2025, doi: 10.1145/3711053.

[2] B. Karwin, SQL *Antipatterns: Avoiding the Pitfalls of Database Programming*. Raleigh, NC, USA: Pragmatic Bookshelf, 2010.

[3] J. Celko, *Joe Celko's Trees and Hierarchies in SQL for Smarties*. 2012. doi: 10.1016/c2010-0-69241-8.

[4] D. Hu, X. Cheng, G. Lü, Y. Wen, and M. Chen, "The China Family Tree Geographic Information System," in *Human dynamics in smart cities*, 2020, pp. 13–37. doi: 10.1007/978-3-030-52734-1_3.

[5] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. Sebastopol, CA, USA: O'Reilly Media, 2017.

[6] PostgreSQL Global Development Group, "PostgreSQL documentation: GIN indexes," 2025. [Online]. Available: https://www.postgresql.org/docs/current/gin.html. Describes the usage of Generalized Inverted Indexes (GIN) and support for indexing array and other composite data types.

[7] R. Čerešňák and M. Kvet, "Comparison of query performance in relational a non-relation databases," *Transportation Research Procedia*, vol. 40, pp. 170–177, Jan. 2019, doi: 10.1016/j.trpro.2019.07.027.

[8] J. T. Harviainen and B.-C. Björk, "Genealogy, GEDCOM, and popularity implications," *Informaatiotutkimus*, vol. 37, no. 3, Oct. 2018, doi: 10.23978/inf.76066.

[9] FamilySearch, "*GEDCOM X conceptual model*," 2014. [Online]. Available:

Vol. 7, No. 1, February 2026, Page. 662-674
https://jutif.if.unsoed.ac.id
DOI: https://doi.org/10.52436/1.jutif.2026.7.2.5629

https://github.com/FamilySearch/gedcomx. Modern genealogical data model for web based systems

[10] A. Rey, T. Neumann, and M. Rieger, "Nested Parquet Is Flat, Why Not Use It? How To Scan Nested Data With On-the-Fly Key Generation and Joins," *Proceedings of the ACM on Management of Data*, vol. 3, no. 3, pp. 1–24, Jun. 2025, doi: 10.1145/3725329.

[11] M. A. Aceto *et al.*, "#3418 Genealogical analysis in the Era of big data: implications for biomedical research," *Nephrology Dialysis Transplantation*, vol. 40, no. Suppl 3, Oct. 2025, doi: 10.1093/ndt/gfaf116.004.

[12] K. Min, M. Jung, H. Lee, and J. Cho, "Optimization for Large-Scale n-ary Family Tree Visualization," *Journal of information and communication convergence engineering*, vol. 21, no. 1, pp. 54–61, Mar. 2023, doi: 10.56977/jicce.2023.21.1.54.

[13] T. Taipalus, "On the effects of logical database design on database size, query complexity, query performance, and energy consumption." Jan. 13, 2025. doi: 10.48550/arxiv.2501.07449.

[14] K. Min, M. Jung, H. Lee, and J. Cho, "Analysis of Impact Between Data Analysis Performance and Database," *Journal of information and communication convergence engineering*, vol. 21, no. 3, pp. 244–251, Sep. 2023, doi: 10.56977/jicce.2023.21.3.244.

[15] Gennadii Turutin, Mikita Puzevich, "PostgreSQL JSONB-based vs. Typed-column Indexing: A Benchmark for Read Queries", *IEEE Dataport*, October 29, 2025, doi:10.21227/fxws-3a11

[16] G. Toktomusheva, "Indexing in PostgreSQL: Performance Evaluation and Use Cases." MDPI Preprints, Nov. 27, 2025. doi: 10.20944/preprints202511.2170.v1.

[17] R. O. Obe and L. Hsu, *PostgreSQL: Up and Running—A Practical Guide to the Advanced Open Source Database*, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2017.

[18] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* New York, NY, USA: John Wiley & Sons, 1991.

[19] M. Raasveldt et al., "Fair benchmarking considered difficult: Common pitfalls in database performance comparisons," in *Proceedings of the Workshop on Testing Database Systems*, ACM, 2018. doi: 10.1145/3209950.3209955.

[20] M. Jurkovic, V. Hudovernik, and E. Štrumbelj, "SyntheRela: A benchmark for synthetic relational database generation," in *Will Synthetic Data Finally Solve the Data Access Problem?*, 2025. https://iclr.cc/virtual/2025/32181

[21] Badan Pusat Statistik, "Indikator fertilitas long form Sensus Penduduk 2020," 2022.

[22] K. Davis, P. Leach, and B. Peabody, "Universally Unique IDentifiers (UUIDs)," rfc editor, May 2024. doi: 10.17487/rfc9562.

[23] S. V. Salunke and A. Ouda, "A Performance Benchmark for the PostgreSQL and MySQL Databases," *Future Internet*, vol. 16, no. 10, p. 382, Oct. 2024, doi: 10.3390/fi16100382.

[24] T. Taipalus, "Database management system performance comparisons: A systematic literature review," *Journal of Systems and Software*, vol. 208, p. 111872, Oct. 2023, doi: 10.1016/j.jss.2023.111872.

[25] H.-J. Schönig, *Mastering PostgreSQL 17: Advanced Techniques to Build and Administer Scalable Databases*, Packt Publishing Ltd, 5th ed., November 2024. Chapter 10: Understanding Transaction Management and Concurrency.

[26] J. Han and Y. Choi, "Analyzing Performance Characteristics of PostgreSQL and MariaDB on NVMeVirt." Nov. 15, 2024. doi: 10.48550/arxiv.2411.10005.

[27] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th ed. Boston, MA, USA: Pearson, 2016. Chapter 24: NOSQL Databases and Big Data Storage Systems

[28] PostgreSQL Global Development Group, "PostgreSQL 18.1 documentation: UUID functions," 2025.