String Matching Strategies in Database-Oriented Architectures

Yonatan Edward Njoto - 12325036
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: yonatan.njoto@gmail.com , 13523036@std.stei.itb.ac.id

***Abstract*—This project demystifies database performance by demonstrating the power of indexing. Through a real-world PostgreSQL benchmark and a foundational C++ Generalized Inverted Index implementation, showing how to achieve search speedups. This incredible gain comes at the minor cost of overhead when adding new data. Designed as a hands-on guide, this project is the perfect resource for developers and students to learn both the practical impact and the core mechanics of database optimization.**

Keywords—Computer Science, String, Indexing, B-Tree, Database

I. INTRODUCTION

In modern software development, the efficiency of data retrieval is a crucial part of application performance. As datasets grow and user loads increase, databases can quickly be a significant bottleneck. To address this, architects employ various scaling strategies, such as partitioning (horizontally sharding tables across multiple servers or vertically splitting them by columns) help manage massive datasets. The use of read-only replicas allows for the distribution of query loads, separating read-heavy traffic from the primary write database. Furthermore, caching layers and load balancers are often implemented to enhance responsiveness and ensure high availability.

While these architectural patterns are mandatory to scale a system that can serve huge number of users, performance optimization also relies heavily on foundational techniques and algorithms. Among the most critical of these is the strategic implementation of database indexing, a method for optimizing query performance directly within the database engine. However, the benefits of indexing are often accompanied by a trade-off in write performance, a nuance that is critical for developers and database administrators to understand.

This project undertakes a comprehensive, hands-on analysis of the performance impact of a specific and powerful indexing strategy: GIN (Generalized Inverted Index) for string operations. While scaling techniques like partitioning and read replicas address the "where" and "how many" of data processing, this study focuses on the "how fast" at the query level. Aiming to bridge the gap between abstract theory and concrete results through a dual-pronged approach. First, a benchmark will be made on a live PostgreSQL database to measure real-world performance on tables with and without indexes. Second, to demystify the underlying mechanics, an implementation of a GIN-like data structure in C++ is provided, comparing its performance to a simple linear search through text.

The following sections will detail the methodology used for both the PostgreSQL benchmark and the C++ implementation. Then a comparative analysis of the performance data collected, highlighting the emergent patterns in reading and writing speeds. Finally, this report will discuss the broader implications of these findings, offering insights into the practical trade-offs of indexing and providing guidance on making informed decisions in database design and system architecture.

II. THEORITICAL BACKGORUND

A. Brute Force

The foundational challenge in data retrieval, especially for complex data like text, is finding specific information within a large collection. The most basic approach is the brute-force algorithm. To find documents containing a specific term, this method requires a Linear Scan. The system must read every single record in a table, scan its entire text content for the desired word, and repeat this for every record. This approach has a time complexity of O(n*m), where n is the number of records and m is the size of the content to be scanned, making it profoundly inefficient and unscalable.

B. Decrease and Conquer

To overcome the severe limitations of the brute-force method, databases employ sophisticated index structures. One of the most well-known is the B-Tree, which excels at rapidly finding individual, ordered values (like a specific username or an ID number). However, a B-Tree is not designed for the unique challenge of searching for multiple, independent terms within large blocks of text. For this purpose, a specialized solution like the GIN (Generalized Inverted Index) is required, which operates on a Divide and Conquer strategy. Instead of treating a search for multiple terms as one large problem, this paradigm breaks it down into smaller, independent sub-problems [1]. For instance, a query for "database AND performance" is divided into two distinct tasks: 1) find all

records containing "database," and 2) find all records containing "performance." The system then "conquers" each sub-problem individually and "combines" the results, allowing it to efficiently pinpoint relevant records.

C. *String Matching*

In the context of this database index, string matching refers to the fundamental operation of using strings as keys for comparison which can be done with various algorithms such as Knuth–Morris–Pratt algorithm or Boyer–Moore [4]. The B-tree efficiently navigates its nodes by performing basic lexical comparisons (<, =, >) on these string keys. This allows the database to quickly locate a specific record without resorting to a slow, full scan that would compare against every string in the table.

III. Implementation

To analyze the performance of DBMS (database management systems), a connection using a Python client to PostgreSQL is established. The initial setup involves provisioning a test table with an id SERIAL PRIMARY KEY column and a data TEXT column to hold string-based content. The one being used to search is the string-based content because we are comparing index toward strings.

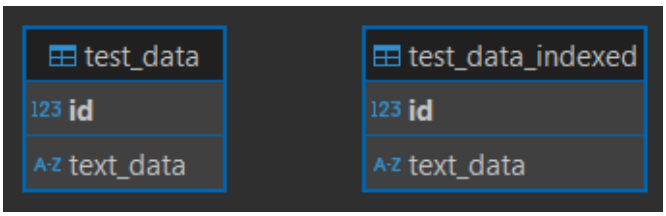

Figure 1. Database Schema
Source: https://github.com/yonatan-nyo/string-dbms-handling

To ensure the integrity and repeatability of benchmarks, the benchmark code must account for PostgreSQL's internal data handling. For read-side consistency, executing DISCARD ALL to reset the session's state, clearing any cached query plans. On the write-side, it is important to note how PostgreSQL ensures data is safely stored on disk. As the official documentation notes:

One aspect of reliable operation is that all data from a committed transaction must be stored in a nonvolatile area safe from power loss, OS failure, or hardware failure. While this seems straightforward, it is complicated by the fact that "disk drives are dramatically slower than main memory and CPUs", which has led to several layers of caching between the main memory and the physical disk. These layers include the operating system's buffer cache, a potential cache in the disk drive controller (common on RAID cards), and finally, caches within most disk drives themselves [2].

These caches, particularly "write-back" caches that delay sending data to the drive, can present a reliability hazard as their contents are often volatile and can be lost when there is power failure. To handle this, PostgreSQL utilizes operating system features to force writes from the buffer cache to the disk. However, the responsibility falls on the administrator to use reliable hardware, such as disk controllers with "battery-backed caches", and to correctly configure all components. For consumer-grade drives, this may involve disabling the drive's write-back cache if it cannot guarantee data will be written before a shutdown [2].

Another risk to data integrity is the possibility of partial page writes, where a power failure occurs after some, but not all, sectors of a page have been written to the disk platter. To guard against this, "PostgreSQL periodically writes full page images to permanent storage before modifying the actual page on disk". This strategy ensures that during a crash recovery, PostgreSQL can restore any partially written pages from these full-page images, preserving data consistency. [2]

```python
def clear_postgres_cache(conn):
    """Clears PostgreSQL cache to ensure consistent measurements."""
    # Commit any pending transaction
    if not conn.autocommit:
        conn.commit()

    # Execute cache-clearing commands
    with conn.cursor() as cur:
        try:
            # These commands must run outside transaction blocks
            conn.set_session(autocommit=True)
            cur.execute("DISCARD ALL;")
            # pg_stat_reset() is useful for resetting statistics,
            # DISCARD ALL for cache
            cur.execute("SELECT pg_stat_reset();")
        finally:
            # Restore original autocommit setting
            conn.set_session(autocommit=False)
```

Figure 2. Database Clear Cache
Source: https://github.com/yonatan-nyo/string-dbms-handling

This experiment is designed to benchmark database performance under controlled data growth, observing how internal index structures evolve and impact operation times. The methodology involves populating two tables, one with a index (GIN) and the other one without any index.

To capture a good performance curve, the database tables are populated in stages. The total number of records grows multiplicatively by a factor of 1.2 at each stage, starting from a small base and scaling up to a maximum of 250,000 records. This multiplicative approach provides information at smaller record counts and generates a smooth distribution of data points across the entire test range, making it easier to visualize performance trends as the dataset scales.

At each stage, a chunk of data is inserted into the indexed and non-indexed tables, to measure the overhead associated with maintaining a GIN index during write operations by comparing the time consumed on operations.

Batch Insertion: To ensure efficient data loading and simulate a realistic bulk-insert scenario, the program will use the psycopg2.extras.execute_batch function which optimizes the insertion process by minimizing network roundtrips between the application and the database server.

The comparison between the insertion time for the indexed table (test_data_indexed) and the un-indexed table (test_data) will be the write penalty. This overhead is expected, as an indexed insert requires the database to perform additional work: parsing the text, breaking it into tokens, and updating the GIN index's complex structure of token dictionaries and posting lists to be able to address data table as it grows.

```python
def benchmark_select(conn, table_name, search_term, repeats):
    """Benchmarks SELECT performance with proper setup."""
    total_time = 0
    for i in range(repeats):
        # Clear cache before each select for consistency
        clear_postgres_cache(conn)
        with conn.cursor() as cur:
            start_time = time.perf_counter()
            cur.execute(
                SQL("SELECT * FROM {} WHERE text_data = %s;").format(
                    Identifier(table_name)),
                (search_term,)
            )
            # Fetching is part of the operation
            if cur.fetchone() is None:
                print(
                    f"Warning: Search term '{search_term}' not found in
'{table_name}'.")

            total_time += time.perf_counter() - start_time

    return total_time / repeats
```

Figure 3. Benchmark Select
Source: https://github.com/yonatan-nyo/string-dbms-handling

In accordance with official PostgreSQL documentation, the GIN (Generalized Inverted Index) has been selected [3]. To better understand the mechanics behind GIN, Generalized Inverted Index will be modeled and implemented in C++ as part of this investigation.

```cpp
//imports

// GIN (Generalized Inverted Index) implementation
// Similar to PostgreSQL's GIN index for text search
class GINIndex {
private:
    // Inverted index: token -> set of record pointers that contain this
token
    std::unordered_map<std::string, std::set<const Record *>> tokenIndex;

    // Store all records for reference
    std::vector<const Record *> allRecords;

    // Tokenize a string into individual tokens/terms
    std::vector<std::string> tokenize(const std::string &text) const;

    // Normalize token (lowercase, remove special chars, etc.)
    std::string normalizeToken(const std::string &token) const;

public:
    GINIndex();
    ~GINIndex();

    // Insert a record into the index
    void insert(const std::string &textToIndex, const Record &record);

    // Search for records containing the given term
    std::vector<const Record *> search(const std::string &searchTerm) const;

    // Search for records containing all the given terms (AND operation)
    std::vector<const Record *> searchAND(const std::vector<std::string>
&searchTerms) const;

    // Search for records containing any of the given terms (OR operation)
    std::vector<const Record *> searchOR(const std::vector<std::string>
&searchTerms) const;

    // Get statistics about the index
    void printIndexStats() const;
};

#endif // GININDEX_H
```

Figure 4. Generalized Inverted Index
Source: https://github.com/yonatan-nyo/string-dbms-handling

To facilitate this, a C++ program demonstrates core principles of a Generalized Inverted Index, as detailed in the GINIndex.h header file. This in-memory model serves as a simulation, to observe the logical operations or algorithm that provides better performance of production systems like PostgreSQL.

The fundamental concept is an "inversion" of the typical data-to-record relationship. Instead of mapping a record to its content, GINIndex maps individual components of the content specifically words or "tokens" back to the records that contain them. This is achieved through the primary data structure std::unordered_map<std::string, std::set<const Record *>>, which acts as the heart of inverted index. In this map, each unique token serves as a key, while the value is a set of pointers to every record containing that token.

The "write performance" measured in benchmark directly corresponds to the execution time of the GINIndex::insert method. When a new record is introduced, its text content undergoes a multi-stage process.

```cpp
std::vector<std::string> GINIndex::tokenize(const std::string &text) const {
    std::vector<std::string> tokens;
    std::stringstream ss(text);
    std::string token;

    // Simple tokenization by whitespace and common delimiters
    while (std::getline(ss, token, ' ')) {
        if (!token.empty()) {
            // Further split by common delimiters
            std::string currentToken;
            for (char c : token) {
                if (std::isalnum(c)) {
                    currentToken += c;
                } else {
                    if (!currentToken.empty()) {
                        tokens.push_back(normalizeToken(currentToken));
                        currentToken.clear();
                    }
                }
            }
            if (!currentToken.empty()) {
                tokens.push_back(normalizeToken(currentToken));
            }
        }
    }

    return tokens;
}

std::string GINIndex::normalizeToken(const std::string &token) const {
    std::string normalized = token;

    // Convert to lowercase
    std::transform(normalized.begin(), normalized.end(), normalized.begin(),
::tolower);

    return normalized;
}
```

Figure 5. Normalize Token
Source: https://github.com/yonatan-nyo/string-dbms-handling

First, the tokenize function splits the text into a vector of individual terms. Subsequently, each term is passed to a normalizeToken function to ensure consistency by converting it to lowercase and removing punctuation. This prevents semantic duplicates, like "Apple" and "apple," from being treated as distinct entries.

Finally, for each normalized token, the system updates the main tokenIndex, adding a pointer to the new record into the corresponding set. This process highlights why GIN index updates can be resource-intensive; a single record rich with unique words can trigger dozens of distinct updates within the index structure.

```cpp
std::vector<const Record *> GINIndex::search(const std::string &searchTerm)
const {
    std::vector<const Record *> results;

    std::string normalizedSearchTerm = normalizeToken(searchTerm);

    // Check if the normalized search term exists in our index
    auto it = tokenIndex.find(normalizedSearchTerm);
    if (it != tokenIndex.end()) {
        // Convert set to vector
        results.assign(it->second.begin(), it->second.end());
    }

    return results;
}

std::vector<const Record *> GINIndex::searchAND(const
std::vector<std::string> &searchTerms) const {
    if (searchTerms.empty()) {
        return {};
    }

    // Start with records containing the first term
    std::string firstTerm = normalizeToken(searchTerms[0]);
    auto it = tokenIndex.find(firstTerm);
    if (it == tokenIndex.end()) {
        return {}; // No records contain the first term
    }

    std::set<const Record *> resultSet = it->second;

    // Intersect with records containing each subsequent term
    for (size_t i = 1; i < searchTerms.size(); ++i) {
        std::string term = normalizeToken(searchTerms[i]);
        auto termIt = tokenIndex.find(term);
        if (termIt == tokenIndex.end()) {
            return {}; // No records contain this term
        }

        std::set<const Record *> intersection;
        std::set_intersection(
            resultSet.begin(), resultSet.end(),
            termIt->second.begin(), termIt->second.end(),
            std::inserter(intersection, intersection.begin()));

        resultSet = intersection;

        if (resultSet.empty()) {
            break; // No records contain all terms
        }
    }

    std::vector<const Record *> results(resultSet.begin(), resultSet.end());
    return results;
}

std::vector<const Record *> GINIndex::searchOR(const std::vector<std::string>
&searchTerms) const {
    std::set<const Record *> resultSet;

    // Union records containing each term
    for (const std::string &searchTerm : searchTerms) {
        std::string term = normalizeToken(searchTerm);
        auto it = tokenIndex.find(term);
        if (it != tokenIndex.end()) {
            resultSet.insert(it->second.begin(), it->second.end());
        }
    }

    std::vector<const Record *> results(resultSet.begin(), resultSet.end());
    return results;
}
```

Figure 6. Generalized Inverted Index Search
Source: https://github.com/yonatan-nyo/string-dbms-handling

Similarly, read performance is assessed by timing the searchAND and searchOR operations, which simulate common text-based queries. To find records containing any of the specified terms (an OR search), the system retrieves the set of record pointers for each term and then computes the union of these sets.

Conversely, to find records containing all specified terms (an AND search), it computes the intersection of the pointer sets. This intersection is a particularly efficient operation that rapidly narrows the result set to only the records that satisfy the strict criteria. By meticulously measuring the performance of these fundamental insertion and search operations at each stage of data growth, this benchmark will provide clear, empirical data on how a GIN index behaves as its internal structures become larger and more complex.

To quantify the true performance benefit of the GIN index, the benchmark includes a contrasting unindexed search methodology, brute-force linear scan. When a search is initiated, the function iterates through every single record in the collection, one by one from the beginning. For each record, it performs a direct string comparison on the target field. This process only concludes when a match is found or, in the worst-case scenario, after the entire dataset has been examined.

IV. Usage

The result of the indexed and unindexed field on insert action from PostgreSQL is as follows:

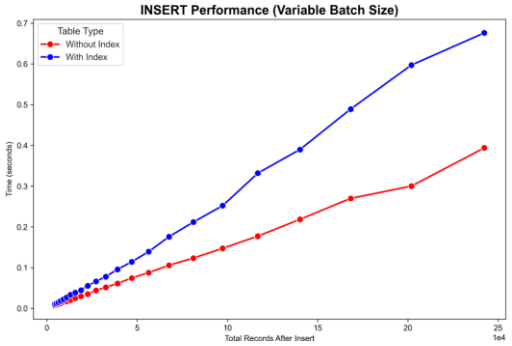

Figure 7. Insert Performance
Source: https://github.com/yonatan-nyo/string-dbms-handling

On insert the performance of unindexed is faster around twice as fast, because of the absence of index maintenance overhead, clearly showing that the red line ("Without Index") remains consistently below the blue line ("With Index"), and this gap widens as the dataset grows. This performance advantage is due to the simplicity of an unindexed insertion.

When a record is inserted into an unindexed table, the database performs a single, straightforward operation: it writes the new row data to the end of the table's data file (the "heap"). This is a relatively cheap append operation.

In contrast, when inserting into an indexed table, the database must perform a two-step process (writing the data and updating the index), Updating the index is the critical overhead. The database must also update the GIN index to make the new record searchable. This involves tokenizing the text, and for every unique token, it must find its corresponding entry in the index and add a new pointer to the new row.

This "index maintenance" is a significant additional workload. As the chart shows, the cost of this workload is not constant; it increases as the index itself grows larger and more complex, which is why the blue line becomes progressively steeper. The unindexed insert avoids this entire second step, making it fundamentally faster.

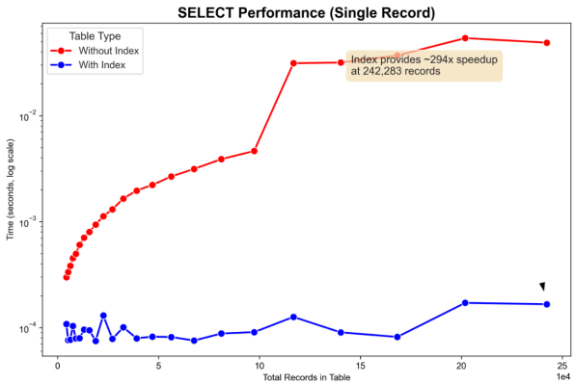

Figure 8. Select Performance
Source: https://github.com/yonatan-nyo/string-dbms-handling

On select the performance of unindexed is far worse than indexed because of the difference in data retrieval strategy, which is dramatically illustrated in the right chart, "SELECT Performance." The performance of the indexed select is so superior (over 294x faster at ~242,000 records) that the y-axis must be on a logarithmic scale just to visualize both lines.

The reason for this stark contrast is how the database finds the requested data, without an index, the database has no guide to where a specific record might be. To satisfy the SELECT query, it has no choice but to perform a Full Table Scan (or Sequential Scan). It must read every single row in the table from the beginning and compare its value to the search term. As the table grows, the amount of work grows in direct proportion. This is why the red line trends sharply upwards more records mean more work and more time. This is known as O(n) complexity.

With a GIN index, the database uses a highly efficient, multi-step process. Instead of scanning the table, it first consults the index. It performs a very fast lookup (akin to looking up a word in a dictionary's index) to find the search term(s). This lookup instantly provides it with a list of direct pointers (TIDs) to the exact physical locations of all the matching rows in the table. The database then uses these pointers to fetch only the required rows, completely ignoring the rest of the table.

This "Index Seek" operation is incredibly efficient. Its performance depends on the structure of the index, not the size of the table. This is why the blue line remains virtually flat and close to zero, regardless of whether the table has 10,000 or 250,000 records. This “nearly constant” time is the primary benefit of indexing.

Therefore, It is important to acknowledge that in production environments handling massive datasets, indexing is just one component of a much larger performance and scalability strategy. To manage extreme, write loads and read traffic, architects often employ advanced techniques such as database partitioning, which splits large tables into smaller, more manageable pieces, and master-slave (or source-replica) architectures. This replication strategy allows read queries to be distributed across multiple slave databases, drastically reducing the load on the master database which is dedicated to handling writes. These architectural patterns are critical for ensuring high availability and throughput in large-scale systems. Nevertheless, as the focus here is to fundamentally understand the index-based approach at a granular level, C++ program will further investigate the mechanics of the GIN algorithm through the implementation.

```cpp
int main() {
    const int recordCount = 500000;
    // GIN Index (Generalized Inverted Index) - similar to PostgreSQL's
approach

    std::cout << "Generating " << recordCount << " records..." << std::endl;

    std::vector<Record> records;
    records.reserve(recordCount);
    for (int i = 0; i < recordCount; ++i) {
        records.push_back({i, "user" + std::to_string(i), "user" +
std::to_string(i) + "@example.com"});
    }

    // Shuffle the data to simulate random insertions, which is more
realistic.
    std::random_device rd;
    std::mt19937 g(rd());
    std::shuffle(records.begin(), records.end(), g); // 1. Unindexed
Structure: A simple vector for linear scan.
    std::vector<Record> unindexedData = records;

    // 2. Indexed Structure: GIN (Generalized Inverted Index).
    GINIndex indexedData;
    std::cout << "Populating GIN index..." << std::endl;
    for (const auto &rec : records) {
        // Index the username field for text search
        indexedData.insert(rec.username, rec);
    }
    std::cout << "Data generation and indexing complete." << std::endl;

    //perform a search

    return 0;
}
```

Figure 9. C++ Data Preparation
Source: https://github.com/yonatan-nyo/string-dbms-handling

To assess search performance, the C++ program was populated with a dataset of 500,000 records, each consisting of an ID, username, and password. A performance test was then conducted to measure search retrieval times, with the results presented below in **Figure 10.**

```
------------------------------------------------
Performing UNINDEXED search for username 'user499950'...
Found record with email: 'user499950@example.com'
Time taken (Unindexed Linear Scan): 1168.35 microseconds
------------------------------------------------
Performing INDEXED search for username 'user499950'...
Found 1 record(s), first match email: 'user499950@example.com'
Time taken (Indexed GIN Search):      2.902 microseconds
------------------------------------------------
Result: The GIN indexed search was approximately 402 times faster.
------------------------------------------------
```

Figure 10. C++ Select Performance
Source: https://github.com/yonatan-nyo/string-dbms-handling

The vast performance gap between the unindexed and indexed search methods, as demonstrated by the 402 times speedup shown in the output image, can be formally explained through a deep analysis of their time and space complexity. The unindexed approach relies on a brute-force linear scan. To find the record for user499950, it must start at the very first record and iterate through the entire collection, comparing each username until it reaches the 499,950th entry. This means its search time grows linearly with the number of records, n, resulting in a time complexity of O(n). Doubling the records

would double the average search time. Its space requirement is also O(n), as it only needs to store the data itself with no significant structural overhead. This direct relationship between data size and search time is why the unindexed method becomes untenably slow as the dataset grows.

In contrast, the GIN index achieves its remarkable search speed by leveraging a more sophisticated "inverted index" data structure. As detailed in the GINIndex.cpp implementation, the index is built around an std::unordered_map named tokenIndex. Think of this like the index at the back of a large textbook: instead of reading the whole book to find a term, you go directly to the index page for that term. The C++ unordered_map (a hash map) allows for direct lookups of any search term with an average time complexity of O(1), or constant time. When the benchmark performs an indexed search for 'user499950', the search function in GINIndex.cpp does not scan the records. Instead, it hashes the term 'user499950' and goes directly to the corresponding entry in the tokenIndex map. This immediately provides it with a set of pointers to the exact memory locations of the matching records. This O(1) complexity, which is critically independent of the total number of records, is the core reason for the dramatic performance improvement from over 1100 microseconds to under 3 microseconds.

However, this search efficiency comes at a calculated cost, which is evident in the GIN index's insertion time and space usage. The insert function in GINIndex.cpp reveals this "write penalty". When a new record is added, its text is broken into w words (tokens). For each of these w tokens, the program must perform a lookup in the tokenIndex and then insert a pointer into a std::set, an operation with a time complexity of roughly O(log p), where p is the number of records already associated with that token. This multi-step process is significantly more work than the simple O(1) unindexed insertion. Furthermore, the space complexity is substantially higher at O(n + U + P), where U is the number of unique tokens and P is the total number of pointer instances stored in the index. For example, a single common word like "the" might appear in thousands of records, meaning its entry in the tokenIndex would hold thousands of pointers, consuming significant memory. This additional space is required to build and maintain the entire inverted index structure that makes the O(1) search possible. This trade-off slower writes and higher memory usage in exchange for nearly instantaneous reads is fundamental to high-performance database indexing.

V. Conclusion

This project demonstrates the critical trade-offs inherent in database indexing for string-based searches. Through a hands-on PostgreSQL benchmark and a foundational C++ implementation of a Generalized Inverted Index (GIN), a clear performance dichotomy emerges.

Unindexed operations exhibit superior write performance, with insertions being approximately twice as fast as their indexed counterparts. This is because they involve a simple append operation without the overhead of index maintenance. However, this advantage is completely overshadowed by their profoundly inefficient read performance, which degrades linearly as the dataset grows, following an O(n) complexity.

Conversely, indexing provides a dramatic acceleration in data retrieval. The PostgreSQL benchmark recorded indexed SELECT queries performing over 294 times faster than unindexed scans on a dataset of around 242,000 records. This was further corroborated by the C++ model, which showed a 402-fold speedup on a dataset of 500,000 records. This remarkable efficiency stems from the GIN index's ability to perform lookups in nearly constant time, or O(1), by using a hash map to directly locate data pointers.

The trade-off for this near-instantaneous read capability is the "write penalty" the additional computational work required to tokenize text and update the index structure upon data insertion. This analysis confirms that while indexing is an indispensable tool for optimizing read-heavy applications, its impact on write speeds and memory usage must be a key consideration in database design. For large-scale systems, while advanced strategies like replication and partitioning are essential, a fundamental understanding of indexing mechanics remains the cornerstone of building efficient and scalable data architectures.

Video Link at Youtube

https://youtube.com/shorts/TIy2hmAbqOg

Acknowledgment

The author would like to express sincere gratitude to God Almighty for the guidance and ease in writing this paper. Special thanks are also extended to Dr. Nur Ulfa Maulidevi, S.T, M.Sc. and Dr. Ir. Rinaldi Munir, M.T. for their role as the lecturer in the IF2211 Strategy and Algorithm course and for publishing the lecture materials on the website, which were instrumental in the research process. The author is deeply appreciative of the unwavering support from family and friends throughout the completion of this paper

References

[1] Munir, Rinaldi. 2025. “Algoritma Decrease and Conquer (Bagian 1)”. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/11-Algoritma-Decrease-and-Conquer-2025-Bagian1.pdf (Diakses pada 22 Juni 2025)

[2] The PostgreSQL Global Development Group. 2025. “28. Write-Ahead Logging (WAL)”. https://www.postgresql.org/docs/8.1/wal.html (Diakses pada 22 Juni 2025)

[3] The PostgreSQL Global Development Group. 2025. “12.9. Preferred Index Types for Text Search”. https://www.postgresql.org/docs/current/textsearch-indexes.html (Diakses pada 22 Juni 2025)

[4] Munir, Rinaldi. 2025. “Pencocokan String”. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-(2025).pdf (Diakses pada 23 Juni 2025)

STATEMENT

Hereby, I declare that this paper I have written is my own work, not a reproduction or translation of someone else's paper, and not plagiarized.

Bandung, 24 Juni 2025

Yonatan Edward Njoto 13523036