Jurnal Teknologi Informasi dan Pendidikan
Volume 19, No. 2, September 2026
https://doi.org/10.24036/jtip.v19i2.1098

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

Comparative Performance of Indonesian Stemming: PL/SQL Implementation of Nazief and Adriani Algorithm versus Sastrawi Library

I Putu Gede Panji Badra Mahayana[1*]✉, Komang Oka Saputra[1], Made Sudarma[1]
[1] Electrical Engineering, Universitas Udayana, Badung, Indonesia
**Corresponding Author: imahayana@unud.ac.id*

Article Information

Article history:

No. 1098
Rec. January 08, 2026
Rev. February 23, 2026
Acc. May 05, 2026
Pub. May 05, 2026
Page. 1545 – 1564

Keywords:

- *Accuracy*
- *Computing Time*
- *Nazief and Adriani Algorithm*
- *PL/SQL*
- *Sastrawi Library*

ABSTRACT

Text preprocessing in Indonesian applications commonly relied on external libraries such as Sastrawi. However, performing this task outside the database layer often introduced significant latency due to data communication overhead between the application and the server. This study proposed and evaluated a native stemming mechanism utilizing the Nazief and Adriani algorithm implemented directly within an PL/SQL environment. The primary objective was to determine whether in-database processing could offer better performance than the standard application-layer approach. The assessment compared the PL/SQL implementation against the Python-based Sastrawi library using a comprehensive dataset of 54,715 words sourced from the Kamus Besar Bahasa Indonesia (KBBI). Performance metrics focused on stemming accuracy and total execution time. The empirical results revealed that the proposed PL/SQL method achieved an accuracy of 96.82%, which proved slightly superior to the 96.58% accuracy obtained by Sastrawi. Furthermore, the stored procedure implementation demonstrated significant efficiency, completing the process in 602.22 seconds, whereas the baseline method required 1,259.28 seconds. It was concluded that migrating the stemming logic into the database layer effectively reduced execution time by approximately 52.18% while maintaining high precision. These findings suggested that native database implementation provided a more robust solution for systems requiring high-performance text processing.

How to Cite:

Mahayana, I. P. G. P. B., & et al. (2026). Comparative Performance of Indonesian Stemming: PL/SQL Implementation of Nazief and Adriani Algorithm versus Sastrawi Library. Jurnal Teknologi Informasi Dan Pendidikan, 19(2), 1545-1564. https://doi.org/10.24036/jtip.v19i2.1098

Jurnal Teknologi Informasi dan Pendidikan
Volume 19, No. 2, September 2026
https://doi.org/10.24036/jtip.v19i2.1098

1. INTRODUCTION

Indonesian is characterized as an agglutinative language with complex morphology, where word formation relies heavily on the attachment of affixes prefixes, suffixes, infixes, or confixes to a root word [1]. Consequently, a single root word can generate various derivative forms, creating challenges in text computation. To address this, stemming is applied to map these variations back to their base form by removing the attached affixes [2]. This process plays a vital role in text preprocessing to enhance efficiency and accuracy in applications such as search engines, sentiment analysis, and automatic document classification [1], [3].

The history of stemming algorithms for the Indonesian language is extensive. The Nazief and Adriani algorithm, introduced in 1996, is regarded as one of the most comprehensive approaches due to its dictionary-based method and complete morphological rules [1]. Currently, the Sastrawi library, developed in Python, serves as the *de facto* standard implementation among developers [4]. Previous studies indicated that Sastrawi provided more accurate stemming results compared to the Porter algorithm, specifically in reducing overstemming errors in document classification tasks [5].

While application-layer processing (e.g., using Python) is common, the paradigm of in-database processing remains less explored. Procedural languages such as PL/SQL in databases offer robust computational capabilities integrated directly with storage. Integrating procedural logic into the database can eliminate massive network communication overhead often found in traditional architectures when processing large text volumes [6]. Modern SQL has even been proven capable of implementing complex algorithms, such as machine learning, with competitive performance against external libraries [7]. Locally, PL/SQL has been utilized for Full-Text Search in Balinese historical chatbots [8].

However, a review of the literature reveals a significant research gap. The majority of current Indonesian stemming research focuses on: (1) comparing accuracy between algorithms [3], [9]; (2) modifying algorithms for local languages like Javanese or Balinese [10], [11]; or (3) optimizing speed using caching methods [12]. There is no research specifically comparing computation time and accuracy between a native PL/SQL implementation of the Nazief and Adriani algorithm versus the standard Sastrawi library. This comparison is crucial for determining the most efficient architecture for large-scale systems.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

Based on this background, this study aims to implement the Nazief and Adriani stemming algorithm within a PL/SQL environment and evaluate its performance against Sastrawi. The evaluation focuses on word cutting accuracy and computational efficiency (execution time) in processing large-scale datasets. This research is expected to provide empirical evidence regarding the technical feasibility and performance advantages of native text processing within a relational database environment.

2. RESEARCH METHOD

This study used a quantitative experimental method to assess the comparative performance of the Nazief and Adriani stemming algorithm implemented through two distinct approaches. The proposed method involved developing a native stemming mechanism directly within the database layer using PL/SQL, while the Sastrawi library served as the baseline for the application-layer approach. To ensure objective results, the evaluation was conducted using a standardized Indonesian dataset, focusing specifically on two performance metrics: accuracy and execution time.

2.1. Research Flow

The research framework was structured by adapting a systematic software development cycle to ensure the validity of the comparative results. The system development approach referred to a simplified Extreme Programming (XP) methodology, which prioritized technical efficiency through iterative phases of planning, designing, coding, and testing [13], [14]. Broadly, the research process was divided into four main phases, as illustrated in Figure 1 below.

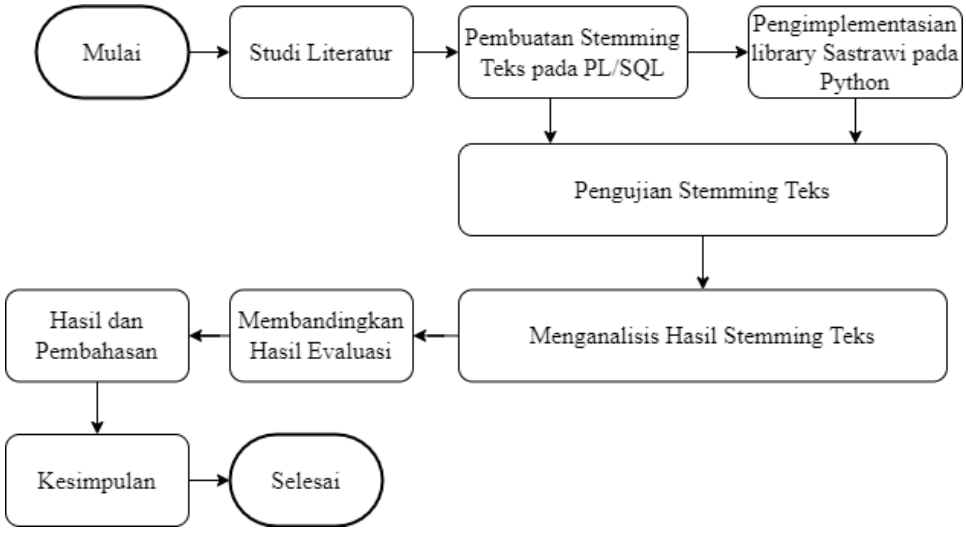

Figure 1. Research workflow diagram detailing the stages from literature review to comparative evaluation

The first phase was Literature Study and Planning, which involved a deep analysis of the Nazief and Adriani morphological rules and the mapping of technical constraints within the PL/SQL environment. This phase aimed to design a procedural algorithm construction that was equivalent to the logic used in the Sastrawi library.

The second phase was System Implementation, carried out in parallel across two distinct environments. In the database environment, stemming rules were converted into a series of Stored Functions and Stored Procedures using PL/SQL. Concurrently, in the application environment, the Python-based Sastrawi library was configured to establish an automated testing interface.

The third phase was Testing and Data Collection. At this stage, the dataset of test words was processed by both systems. This process not only recorded the final root word results but also logged the success status of the stemming for each data entry to facilitate error analysis auditing.

The final phase was Comparative Analysis and Evaluation. The stemming results from both methods were juxtaposed to evaluate performance based on accuracy and execution time parameters. The analysis extended beyond quantitative figures to review specific cases where discrepancies occurred between the PL/SQL and Sastrawi implementations.

2.2. Dataset and Database Design

The data utilized in this research were classified as secondary qualitative data [15]. The primary source consisted of a collection of Indonesian words obtained through a crawling process from the official online *Kamus Besar Bahasa Indonesia* (KBBI). A total of 54,715 lemmas were successfully gathered, encompassing both root words and various affixed derivations. The selection of KBBI as the data source aimed to ensure that the tested morphological variations strictly adhered to standard and comprehensive Indonesian linguistic rules.

To support native text processing within the PL/SQL environment, data management was implemented using a relational database system. The database schema design was constructed by prioritizing normalization principles and data integrity to ensure storage efficiency as well as access speed during dictionary lookup operations [16]. The resulting database structure comprised four main interrelated tables, as visualized in Figure 2 below.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

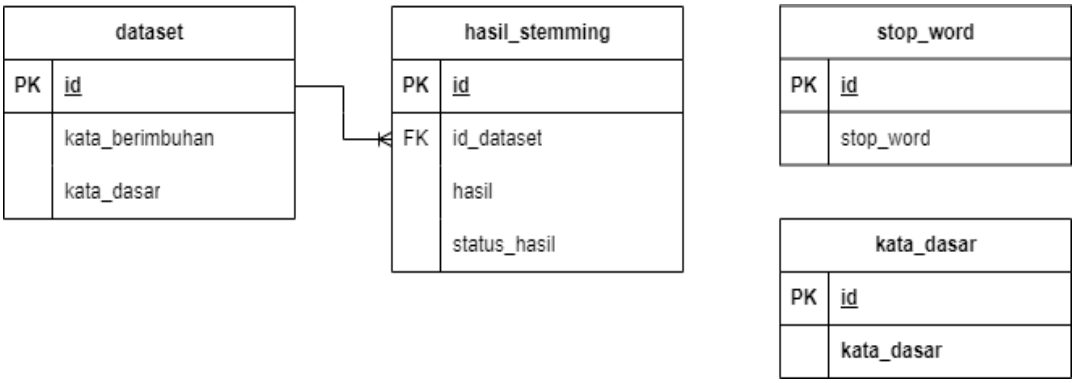

Figure 2. Relational database schema for stemming process

The dataset table functioned as the primary repository for all test words to be processed. The kata_dasar table acted as the reference dictionary containing valid root words used to verify the results of affix removal. The stop_word table contained a list of common words lacking significant lexical meaning, which were utilized for the initial filtering stage. Finally, the hasil_stemming table was designed to store the output of the stemming process, linked via a foreign key to the dataset table to facilitate the comparative analysis between the algorithmic results and the actual root words.

2.3. Proposed Method: PL/SQL Implementation

The implementation of the algorithm within the database environment was constructed using a server-side programming approach. The system structure consisted of two primary object components: Stored Procedures acting as logic flow controllers and Stored Functions serving as morphological processing units. This modular strategy was selected to ensure intensive data processing efficiency, encapsulating business logic within Stored Procedures to minimize redundancy and enhance overall system execution performance [12].

The transformation process from an input word to a root word followed a deterministic logic flow adapted from the standard rules of Nazief and Adriani [1]. Specifically, the implementation stages in PL/SQL began with normalization. Before entering the core algorithm, every input word passed through the clean_kata function, which removed non-alphabetic characters and converted text to lowercase to ensure search consistency. Subsequently, the system performed a rapid lookup in the kata_dasar and stop_word tables. The EXISTS query was utilized for this operation, as it is computationally more efficient than COUNT(*) for detecting data existence in large table indexes [6].

If the word was not found in the dictionary during the initial check, the system executed the stemming procedure. This procedure implemented the complex branching and looping logic required for affix removal, as depicted in Figure 3 below.

Jurnal Teknologi Informasi dan Pendidikan
Volume 19, No. 2, September 2026
https://doi.org/10.24036/jtip.v19i2.1098

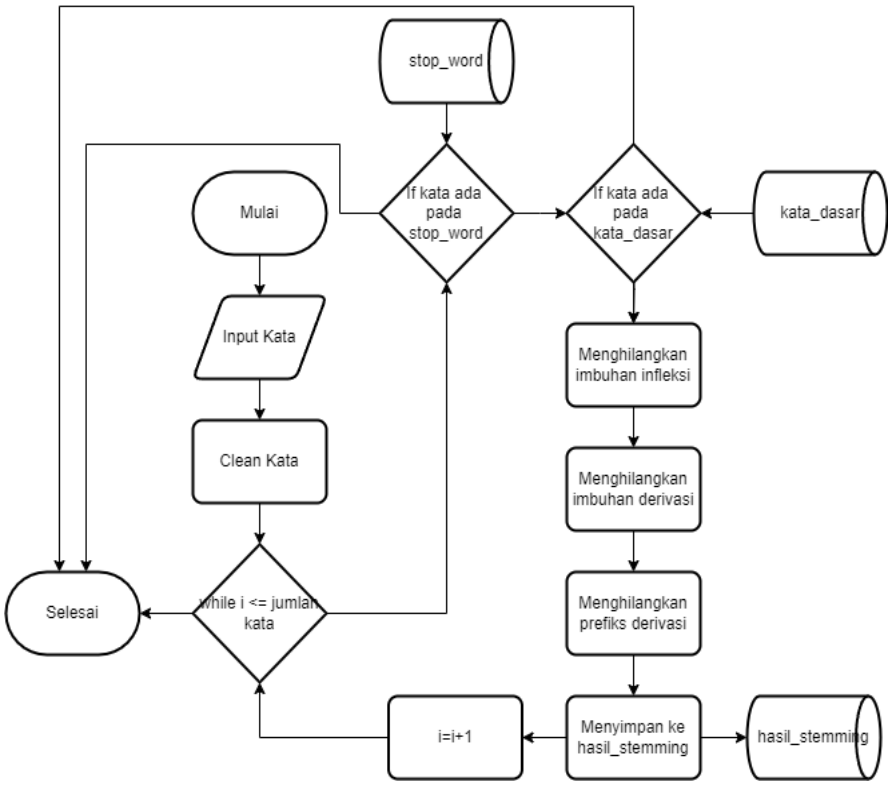

Figure 3. Flowchart of the stemming process in PL/SQL

The cutting logic was implemented in a specific priority order: removal of particles and possessive pronouns (inflection suffixes), followed by removal of derivation suffixes. This logic was equipped with a simple backtracking mechanism; if removing a suffix resulted in a word not found in the dictionary, the system attempted to restore the suffix and proceeded to prefix removal.

The primary technical challenge in this implementation was handling words with ambiguous derivation prefixes, such as "mem-", which could either melt into 'p' or remain 'm'. To resolve this, the menghapus_awalan_ambigu function was designed with limited recursive logic. This function intelligently attempted various root word possibilities and performed cross-verification against the kata_dasar table at each logic branch. This procedural logic for handling ambiguity is visualized in the pseudocode in Figure 4 below.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

```
/* Logic for Handling 'mem-' Prefix Ambiguity in PL/SQL */
IF kata LIKE 'mem%' THEN
    /* Attempt to remove 'mem-' (assume no assimilation) */
    SET kandidat1 = RIGHT(kata, LENGTH(kata)-3);

    IF EXISTS(SELECT 1 FROM kata_dasar WHERE kata = kandidat1) THEN
        RETURN kandidat1;
    ELSE
        /* Attempt to assume 'p' assimilation */
        SET kandidat2 = CONCAT('p', RIGHT(kata, LENGTH(kata)-3));

        IF EXISTS(SELECT 1 FROM kata_dasar WHERE kata = kandidat2) THEN
            RETURN kandidat2;
        END IF;
    END IF;
END IF;
```

Figure 4. Pseudocode for handling ambiguous prefixes

By encapsulating this procedural logic directly within the database engine, the system could perform morphological inference without requiring external application layers, thereby significantly reducing data round-trip latency.

2.4. Baseline Method: Sastrawi Implementation

As a performance benchmark to validate the effectiveness of the proposed method, this study utilized the Sastrawi library. Sastrawi was selected due to its status as a mature, widely adopted industry standard for the Nazief and Adriani algorithm within the modern software development ecosystem [4]. While originally developed for PHP, the variant employed in this research was the Sastrawi library version 1.2.0, running on the Python 3.10.11 environment.

To ensure an objective comparison, the testing scenario was designed to treat data sources identically, adhering to the algorithmic comparison methodology applied by Pamungkas et al. for measuring processing time discrepancies [17]. Consequently, the Python script was not designed to process local text files; instead, it was integrated directly with the MySQL database using the mysql.connector module.

The workflow for this baseline method proceeded as follows: first, the script executed a SELECT query to extract test data from the dataset table. Second, each word was processed using the StemmerFactory class from the Sastrawi library without parameter modification. Finally, the stemming results were sent back to the database using an UPDATE command and stored in a separate column. This integration mechanism ensured that the recorded computation time included data communication latency, which served as the primary differentiating variable against the PL/SQL method.

2.5. Experimental Environment and Specifications

To ensure reproducibility and fairness in comparing the stemming algorithms, both the Sastrawi and the Stored Procedure approaches were executed on the same database engine. The experiments were conducted using the following hardware, software, and database configurations:

2.5.1. Hardware Specifications

- Device: Macbook M3 Pro
- Processor & Graphic: Apple M3 Pro Chip
- Memory (RAM): 18 GB
- Storage: 512 GB SSD

2.5.2. Software Specifications

- Operating System: macOS Tahoe 26.0.1
- Programming Language: Python 3.10.11
- Stemming Library: Sastrawi version 1.2.0

2.5.3. Database and Execution Settings

- Database Management System: MySQL Server 9.5.0 (MySQL Community Server-GPL)
- Database Tool: MySQL Workbench 8.0.44
- Execution Time Measurement: The execution times recorded purely reflect the algorithmic processing and exclude database I/O, commit, and update costs. The detailed mechanism for isolating this execution time is demonstrated in Section 3.1. System Implementation Results.

2.6. Performance Evaluation Metrics

To assess the system's efficiency comprehensively, this study focused on two critical performance metrics: Accuracy, which validated the quality of the morphological transformations, and Computation Time, which measured the speed of data processing.

2.6.1 Accuracy Measurement Evaluation

Accuracy served as the primary indicator for validating the output quality of the stemming algorithm. In this context, accuracy was defined as the percentage of conformity between the system's cutting results and the standard root words stored in the reference dictionary. The validation process employed an exact matching method, where a stemming result was deemed "Correct" solely if it was character-identically consistent with the verified lemma in the kata_dasar table [15]. This dictionary-based comparison approach aligns with

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

the methodology used by Mustikasari et al. to determine algorithmic precision against standard Indonesian root words [18]. The accuracy value was calculated using Equation (1):

$$A=\frac{N_{correct}}{N_{total}} \times 100\% \quad (1)$$

Where A is the accuracy percentage, $N_{correct}$ represents the number of correctly stemmed words, and N_{total} is the total population of the test dataset (54,715 words).

2.6.2 Computation Time Measurement

The measurement of computational time aimed to determine the system's throughput in processing the entire dataset. The recording mechanism involved capturing timestamps immediately prior to the commencement of the stemming process T_{start} and instantaneously upon its completion T_{end}. The average processing time per word was calculated as formulated in Equation (2):

$$T_{avg}=\frac{T_{total}}{N_{total}} \quad (2)$$

Where T_{avg} denotes the average time per word, N_{total} is the total test data, and T_{total} is the total processing duration calculated from the difference between the end time and start time T_{end} - T_{start}.

3. RESULTS AND DISCUSSION

The empirical findings derived from the experimental implementation are detailed in this section. The presentation begins with the technical realization of the Nazief and Adriani algorithm within the database environment, contrasting it with the Sastrawi configuration. Subsequently, the discussion evaluates the performance disparities between the two approaches, specifically analyzing the achieved accuracy rates and the total time required to process the large-scale dataset.

3.1. System Implementation Results

The implementation phase successfully translated the morphological rules into a set of executable database objects. The architecture was modularized into two distinct categories: Stored Procedures, which functioned as the primary control logic, and Stored Functions, which handled specific affix removal tasks. This modular configuration, as displayed in Figure 5 and Figure 6 below, was chosen to enhance code maintainability while optimizing the execution of repetitive string manipulation tasks [12].

Jurnal Teknologi Informasi dan Pendidikan
Volume 19, No. 2, September 2026
https://doi.org/10.24036/jtip.v19i2.1098

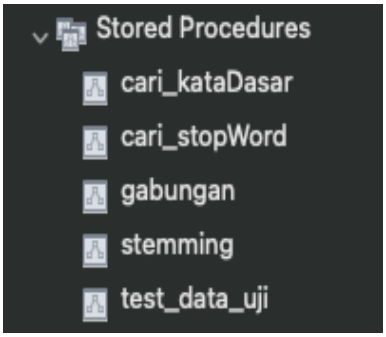

Figure 5. List of compiled Stored Procedures

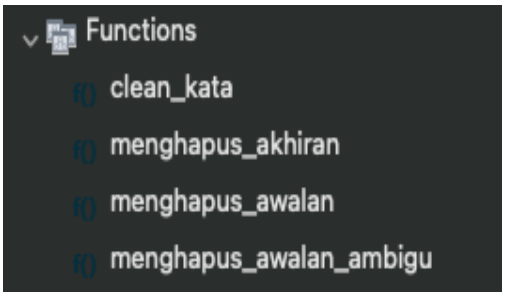

Figure 6. List of compiled Stored Functions

The central orchestration was handled by the stemming procedure. This procedure enforced a deterministic flow, calling specific functions to remove suffixes and prefixes based on the standard morphological rules [1]. A significant technical feature of this implementation was the integration of a backtracking mechanism within the SQL logic to resolve ambiguous prefixes (such as "mem-"). Unlike standard rule-based approaches that might fail on non-standard variations, the stored procedure was designed to verify intermediate results against the kata_dasar table in real-time before finalizing the cut. The snippet of this core logic is captured in Figure 7 below.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

https://doi.org/10.24036/jtip.v19i2.1098

```
CREATE PROCEDURE stemming(IN kata_asli VARCHAR(255), IN kata
VARCHAR(255))
BEGIN
    /* 1. Initial Validation: Check Dictionary & Stopwords */
    IF EXISTS(SELECT 1 FROM kata_dasar WHERE kataDasar=kata) THEN
        CALL cari_kataDasar(kata_asli, kata); RETURN;
    ELSEIF EXISTS(SELECT 1 FROM stop_word WHERE stopWord=kata) THEN
        CALL cari_stopWord(kata_asli, kata); RETURN;
    END IF;

    /* 2. Non-Ambiguous Path (Words without complex prefixes) */
    IF kata NOT REGEXP '^(me|be|pe|te)' THEN
        SET hasil_akhir = menghapus_awalan(kata);
        SET hasil_akhir1 = menghapus_akhiran(kata_asli, hasil_akhir);

        /* Check validity of standard cutting results */
        IF EXISTS(SELECT 1 FROM kata_dasar WHERE kataDasar=hasil_akhir1)
THEN
            CALL cari_kataDasar(kata_asli, hasil_akhir1);
        ELSEIF EXISTS(SELECT 1 FROM kata_dasar WHERE
kataDasar=hasil_akhir) THEN
            CALL cari_kataDasar(kata_asli, hasil_akhir);
        END IF;

    /* 3. Ambiguous Path & Backtracking (Core Logic) */
    ELSE
        /* Step A: Attempt level-1 ambiguity cutting */
        SET hasil_1 = menghapus_awalan_ambigu(kata_asli, kata);

        IF EXISTS(SELECT 1 FROM kata_dasar WHERE kataDasar=hasil_1) THEN
            CALL cari_kataDasar(kata_asli, hasil_1);
        ELSE
            /* Step B: Backtracking (Recursive)
               If level-1 fails, use result for level-2 cutting */
            SET hasil_2 = menghapus_awalan_ambigu(kata_asli, hasil_1);

            IF EXISTS(SELECT 1 FROM kata_dasar WHERE kataDasar=hasil_2)
THEN
                CALL cari_kataDasar(kata_asli, hasil_2);
            ELSE
                /* Step C: Attempt suffix removal combination from
level-2 result */
                SET hasil_3 = menghapus_akhiran(kata_asli, hasil_2);
                /* ... (Logic continues to deep inspection) ... */
            END IF;
        END IF;
    END IF;
END
```

Figure 7. Code snippet of the core stemming logic in PL/SQL

The implementation of this procedural logic directly within the database engine allowed for morphological inference without the need to transfer data to an external application layer, effectively eliminating the round-trip latency usually associated with such operations.

In parallel, the baseline environment was established using the Sastrawi library version 1.2.0 running on Python 3.10.11. To ensure the performance comparison reflected real-world application architectures, the testing script was not designed to process static text files. Instead, it was integrated with the database using the mysql.connector module, adhering to the comparative testing standards applied in similar algorithmic studies [17]. This configuration was essential to capture the total processing time, including the data retrieval and update phases that typically burden application-layer stemming. The specific code structure used to bridge the Python environment with the database is shown in Figure 8 below.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

```
from Sastrawi.Stemmer.StemmerFactory import StemmerFactory
import mysql.connector
from mysql.connector import Error
import time

try:
    connection = mysql.connector.connect(
        host='localhost',
        database='db_steaming',
        user='root',
        password=''
    )

    query = 'select kata_berimbuhan from data_uji'
    update = 'update data_uji SET hasil_sastrawi=%s WHERE
kata_berimbuhan=%s'

    mycursor = connection.cursor(buffered=True)
    myupdate = connection.cursor()

    factory = StemmerFactory()
    stemmer = factory.create_stemmer()

    print("Sedang mengambil data dari database...")
    mycursor.execute(query)
    myresult = [item[0] for item in mycursor.fetchall()]

    total_data = len(myresult)
    print(f"Total data yang akan diproses: {total_data}")

    # Variabel khusus untuk menampung AKUMULASI WAKTU MURNI algoritma
    total_stemming_time = 0.0
    count = 0

    for i in myresult:
        # 1. NYALAKAN STOPWATCH (Hanya mengapit fungsi Sastrawi)
        stem_start = time.time()

        output = stemmer.stem(i) # <-- Proses inti komputasi algoritma

        # 2. MATIKAN STOPWATCH
        stem_end = time.time()

        # 3. Akumulasikan waktunya
        total_stemming_time += (stem_end - stem_start)

        # 4. Eksekusi query Update ke DB (Proses ini TIDAK MASUK
hitungan waktu)
        myupdate.execute(update, (output, i,))
        count += 1
```

```
        # Update progress setiap 100 data
        if count % 100 == 0:
            print(f"Memproses {count} dari {total_data}...")

    # Melakukan commit massal di akhir (TIDAK MASUK hitungan waktu)
    connection.commit()
    print("Commit ke database berhasil.")

except Error as e:
    print("Error while connecting to MySQL", e)

finally:
    if 'connection' in locals() and connection.is_connected():
        mycursor.close()
        myupdate.close()
        connection.close()
        print("MySQL connection is closed")

# 5. Hitung dan tampilkan durasi murni algoritma
menit = int(total_stemming_time // 60)
detik = int(total_stemming_time % 60)

print("--------------------------------------------------")
print(f"Proses Selesai!")
print(f"Total Waktu Murni Stemming: {total_stemming_time:.4f} detik
({menit} menit {detik} detik)")
print("--------------------------------------------------")
```

Figure 8. Python script configuration for Sastrawi database integration

As demonstrated in the code snippet above, the execution time measurement for the Sastrawi algorithm is specifically designed to isolate the computational load of the stemming process. To prevent architectural bias, the initial data extraction from the database (`fetchall()`) is executed prior to the stemming phase, loading the entire dataset into memory. Furthermore, the time measurement function (`time.time()`) strictly encapsulates only the `stemmer.stem()` operation. The execution time for each individual word is then cumulatively added to the `total_stemming_time` variable. The execution of the `UPDATE` queries and the final permanent storage `COMMIT` to the database are intentionally excluded from this timing window. This methodology guarantees that the recorded duration accurately reflects the pure algorithmic performance of the Sastrawi library, completely independent of Input/Output (I/O) latency or database transaction overhead.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

3.2. Comparative Accuracy Analysis

The accuracy assessment was conducted on the entire population of 54,715 entries to validate the correctness of the morphological transformations. The statistical summary of the testing results is presented in Table 1.

Table 1. Comparison of stemming accuracy

Method	Correct Data Amount	Wrong Data Amount	Accuracy (%)
PL/SQL (Proposed)	52,977	1,738	96.82%
Sastrawi (Python)	52,843	1,872	96.58%

The empirical data revealed that the native PL/SQL implementation achieved a correct stemming rate of 52,977 words, corresponding to an accuracy of 96.82%. In comparison, the Sastrawi library successfully processed 52,843 words, resulting in an accuracy of 96.58%. Consequently, the proposed method demonstrated a positive margin of 0.24% (134 words) over the baseline. These findings, where both methods exceeded the 96% threshold, are consistent with Jumadi et al., who observed that the Nazief and Adriani algorithm generally offers superior precision for standard text compared to other algorithms like Paice-Husk [9].

To investigate the root cause of the 0.24% discrepancy, a drill-down analysis was performed on the log data. The inspection identified 1,102 specific instances where the PL/SQL method successfully returned the correct root word while Sastrawi failed (understemming or overstemming). As exemplified in Figure 9 below, these cases predominantly involved words with complex ambiguous prefixes. The success of the PL/SQL system in this segment confirmed the effectiveness of the backtracking logic within the menghapus_awalan_ambigu function, which allowed the system to re-verify the dictionary when a standard cutting pattern failed.

Jurnal Teknologi Informasi dan Pendidikan
Volume 19, No. 2, September 2026
https://doi.org/10.24036/jtip.v19i2.1098

id	kata_berimbuhan	hasil_benar	hasil_stemming	status_stemming_saya	hasil_sastrawi	status_sastrawi
179	pengacakan	acak	acak	benar	kaca	salah
244	mengada-adakan	ada	ada	benar	mengada-ada...	salah
332	beradik-berkakak	adik	adik	benar	beradik-berkak...	salah
424	aduk-adukan	aduk	aduk	benar	aduk-adukan	salah
425	adukan	aduk	aduk	benar	adu	salah
432	pengadukan	aduk	aduk	benar	adu	salah
533	mengagak-agihkan	agak-agih	agak-agih	benar	mengagak-agi...	salah
693	ajakan	ajak	ajak	benar	aja	salah
743	keajukan	ajuk	ajuk	benar	aju	salah
776	mengakan	akan	akan	benar	kak	salah
824	berakikah	akikah	akikah	benar	berak	salah
948	beraku	aku	aku	benar	bera	salah
949	beraku-akuan	aku	aku	benar	beraku-akuan	salah
1064	memperalati	alat	alat	benar	ralat	salah
1298	amai-amai	amai-amai	amai-amai	benar	ama	salah
1315	pengaman	aman	aman	benar	kam	salah
1318	teraman	aman	aman	benar	ram	salah
1363	ambah-ambah	ambah-a...	ambah-ambah	benar	ambah	salah
1369	ambai-ambai	ambai-am...	ambai-ambai	benar	ambai	salah
1447	mengambung-am...	ambung	ambung	benar	ambung	salah
1520	berampai	ampai	ampai	benar	rampa	salah

Figure 9. Sample cases where PL/SQL produced correct results while Sastrawi failed

The test results reveals three primary failure patterns in the Sastrawi library when compared to the Stored Procedure method. First, Sastrawi frequently struggles with complex multi-affix combinations, resulting in understemming (e.g., failing to fully reduce the word '*mengada-adakan*' to its root '*ada*'). Second, Sastrawi exhibits baseline root interpretations that occasionally lack contextual accuracy, such as extracting '*adu*' instead of the more appropriate '*aduk*' from the word '*pengadukan*'. Lastly, Sastrawi is susceptible to severe overstemming anomalies caused by rigid rule limitations, incorrectly reducing words like '*teraman*' to '*ram*'. In contrast, the customized Stored Procedure approach consistently navigates these morphological edge cases, yielding more accurate and contextually appropriate root words for this dataset.

Conversely, the analysis also recorded 972 instances where the Sastrawi library outperformed the PL/SQL implementation, as shown in Figure 10 below. This suggests that the library contains specific exception handling for certain non-standard morphological variations that were not fully accommodated in the procedural logic.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

id	kata_berimbuhan	hasil_benar	hasil_stemming	status_stemming_saya	hasil_sastrawi	status_sastrawi
160	pengabuan	abu	abuan	salah	abu	benar
161	perabuan	abu	abuan	salah	abu	benar
245	mengadakan	ada	adakan	salah	ada	benar
417	teradukan	adu	aduk	salah	adu	benar
955	pengakuan	aku	akuan	salah	aku	benar
3329	pembabatan	babat	babatan	salah	babat	benar
4045	pembantaran	bantar	bantaran	salah	bantar	benar
4171	pembaringan	baring	baringan	salah	baring	benar
4597	bedolan	bedol	dolan	salah	bedol	benar
4605	bedungan	bedung	dung	salah	bedung	benar
4735	belaian	belai	lai	salah	belai	benar
4794	belanjaan	belanja	lanja	salah	belanja	benar
4888	membelikan	beli	belikan	salah	beli	benar
4890	pembelian	beli	belian	salah	beli	benar
4948	belokan	belok	lokan	salah	belok	benar
5241	membentangkan	bentang	bentangkan	salah	bentang	benar
5571	besaran	besar	saran	salah	besar	benar
5599	beslahan	beslah	slah	salah	beslah	benar
5621	besutan	besut	sutan	salah	besut	benar
6497	pemboyongan	boyong	boyongan	salah	boyong	benar
6643	pembubuhan	bubuh	bubuhan	salah	bubuh	benar

Figure 10. Sample cases where Sastrawi produced correct results while PL/SQL failed

In contrast to the previous findings, an analysis of the instances where the Stored Procedure failed while Sastrawi succeeded reveals specific algorithmic limitations within the database-side approach. The most prominent error pattern is understemming during confix processing, where the procedure frequently removes the prefix but fails to execute the subsequent suffix removal; for instance, reducing '*pembabatan*' to '*babatan*' instead of the correct root '*babat*'. Additionally, the Stored Procedure is prone to overstemming due to the misidentification of pseudo-prefixes. It aggressively strips the characters 'be-' even when they are part of the original root word, erroneously reducing '*belanjaan*' to '*lanja*' and '*besaran*' to '*saran*'. Furthermore, the customized algorithm shows a weakness in morphophonemic reconstruction, as it often fails to restore omitted initial consonants after removing nasalized prefixes, such as failing to derive '*kapur*' from '*pengapuran*', resulting in '*apuran*'.

Despite the slight superiority of the proposed method, both systems shared a residual error rate of approximately 3%. These remaining errors were largely attributed to the limitations of the reference dictionary (KBBI) rather than algorithmic flaws. This phenomenon aligns with the observations of Hatta [19] and Rifai and Winarko [20], who emphasized that in rule-based systems, the upper bound of accuracy is heavily dependent on the completeness of the root word corpus used for validation.

3.3. Computational Efficiency Analysis

Beyond accuracy, execution speed is a critical parameter in determining the feasibility of an algorithm for real-time systems [21]. The measurement results for the end-to-end processing duration are summarized in Table 2.

Table 2. Comparison of computational execution time

Method	Total Time (Seconds)	Average Time per Word (ms)
PL/SQL (Proposed)	602.22	11 ms
Sastrawi (Python)	1,259.28	23 ms

Based on the data, the proposed PL/SQL implementation completed the stemming task for the entire dataset (54,715 words) in 602.22 seconds. In contrast, the Sastrawi library required 1,259.28 seconds to complete the same task. This indicates that the in-database approach is faster by 302.69 seconds, representing a performance improvement of 52.18% over the standard Python-based library.

This significant disparity provides empirical evidence supporting the theory of *data locality*, where processing performed adjacent to the data source offers inherent speed advantages over application-layer processing. In conventional architectures, such as those used by Sastrawi or standard web-based systems [23], there is a hidden computational cost in the form of network communication overhead. Every word must be fetched from the database, transferred to the application memory for processing, and then sent back for updates. This round-trip cycle creates an I/O bottleneck that accumulates massively as the data volume increases.

Conversely, the stored procedure implementation eliminates this distribution channel by executing the stemming logic directly within the database engine. Consequently, computational resources can be focused entirely on morphological manipulation rather than data transport. This finding complements the study by Yusliani et al., who proved that stemming time could be reduced by 59% through parallel multiprocessing [22]. While Yusliani et al. achieved efficiency by splitting the workload across processor cores, this research demonstrates that significant efficiency can also be attained by structurally migrating business logic to the database layer. This confirms that for large-scale text systems, the in-database processing approach offers a more efficient and scalable solution than traditional architectures.

4. CONCLUSION

This research successfully demonstrated that transforming the Nazief and Adriani stemming logic into a native database environment using PL/SQL yields measurable performance enhancements compared to the traditional application-layer approach. Empirically, the proposed PL/SQL method recorded an execution time that was 52.18% faster than the Sastrawi library, completing the task in 602.22 seconds opposed to 1,259.28 seconds for processing 54,715 words. This confirms that eliminating the communication overhead between the application and the database server is a decisive factor in optimizing large-scale text processing systems.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id

Beyond computational speed, the native implementation also exhibited a slightly superior precision rate of 96.82%, outperforming the baseline by 0.24%. This accuracy advantage was primarily driven by the integration of a backtracking mechanism for ambiguous prefixes, which effectively resolved morphological cases where standard rules often fail. However, a residual error margin of approximately 3% persists in both systems, largely attributed to the static nature of the reference dictionary. Consequently, future work should prioritize the development of dynamic dictionary update mechanisms or hybrid statistical approaches to better handle non-standard vocabulary.

REFERENCES

[1] M. Adriani, J. Asian, B. Nazief, H. E. Williams, and S. M. Tahaghoghi, "Stemming Indonesian: A confix-stripping approach," *ACM Transactions on Asian Language Information Processing*, vol. 6, no. 4, pp. 1–33, 2007, doi: 10.1145/1316457.1316459.

[2] P. G. S. C. Nugraha and N. W. Wardani, "Stemming dokumen teks bahasa Bali dengan metode rule base approach," *Jurnal Teknik Informatika dan Sistem Informasi (JATISI)*, vol. 7, no. 3, pp. 510-521, 2020, doi: 10.35957/jatisi.v7i3.538.

[3] A. S. Rizki, A. Tjahyanto, and R. Trialih, "Comparison of stemming algorithms on Indonesian text processing," *TELKOMNIKA*, vol. 17, no. 1, pp. 95-102, 2019, doi: 10.12928/telkomnika.v17i1.10183.

[4] A. Librian, "Sastrawi," GitHub, 2017. [Online]. Available: https://github.com/sastrawi/sastrawi.

[5] M. A. Rosid, A. S. Fitrani, I. R. I. Astutik, N. I. Mulloh, and H. A. Keller, "Improving text preprocessing for student complaint document classification using Sastrawi," in *IOP Conference Series: Materials Science and Engineering*, vol. 874, no. 1, p. 012017, 2020, doi: 10.1088/1757-899X/874/1/012017.

[6] S. Feuerstein and B. Pribyl, *Oracle PL/SQL Programming*. Sebastopol: O'Reilly Media, Inc., 2005.

[7] M. Blacher, J. Giesen, S. Laue, J. Klaus, and V. Leis, "Machine learning, linear algebra, and more: Is SQL all you need?," in *CIDR*, 2022. [Online]. Available: https://www.cidrdb.org/cidr2022/papers/p17-blacher.pdf.

[8] K. T. Wirawan, I. M. Sukarsa, and I. P. A. Bayupati, "Balinese historian chatbot using full-text search and artificial intelligence markup language method," *International Journal of Intelligent Systems and Applications*, vol. 11, no. 8, pp. 21–34, 2019, doi: 10.5815/ijisa.2019.08.03.

[9] Jumadi, D. Sartika, and M. I. Musrifin, "Comparison of Nazief-Adriani and Paice-Husk algorithm for Indonesian text stemming process," *IOP Conference Series: Materials Science and Engineering*, vol. 1098, no. 3, p. 032044, 2021, doi: 10.1088/1757-899X/1098/3/032044.

[10] D. E. Cahyani, L. M. T. Utami, and H. Setiadi, "Clustering of Javanese news in krama alus level with Javanese stemming," in *International Conference on Information and Communications Technology (ICOIACT)*, 2019, pp. 462-467, doi: 10.1109/ICOIACT46704.2019.8938438.

[11] N. W. Wardani and P. G. S. C. Nugraha, "Stemming teks bahasa Bali dengan algoritma enhanced confix stripping," *International Journal of Natural Science and Engineering*, vol. 4, no. 3, pp. 103–113, 2020, doi: 10.23887/ijnse.v4i3.30309.

[12] M. F. Tanjung, "Boosting stemmer performance using cache method," *Jurnal Matematika Dan Ilmu Pengetahuan Alam LLDikti Wilayah 1 (JUMPA)*, vol. 1, no. 1, pp. 6-9, 2021, doi:

10.54076/jumpa.v1i1.34.
[13] I. Ahmad, P. M. R. I. B., and S. Samsugi, "Software development dengan extreme programming (XP) pada aplikasi deteksi kemiripan judul skripsi berbasis Android," *Jurnal Inovtek Polbeng Seri Informatika*, vol. 5, no. 2, pp. 297-307, 2020, doi: 10.35314/isi.v5i2.1654.
[14] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Boston: Addison-Wesley Professional, 2004.
[15] Sugiyono, *Metode Penelitian Kuantitatif, Kualitatif dan R&D*. Bandung: Alfabeta, 2013.
[16] T. Connolly and C. Begg, *Database Systems: A Practical Approach to Design, Implementation, and Management*, 6th ed. London: Pearson Education, 2015.
[17] N. Pamungkas, A. P. Kharisma, and L. Fanani, "Comparison of stemming test results of Tala algorithms with Nazief Adriani in abstract documents and national news," *Jurnal Ilmiah Bidang Teknologi Informasi Dan Komunikasi (Inform)*, vol. 8, no. 1, pp. 33-41, 2023, doi: 10.25139/inform.v8i1.5569.
[18] D. Mustikasari, D. S. Aga, S. M. Arifin, and N. Y. Setiawan, "Comparison of effectiveness of stemming algorithms in Indonesian documents," in *Proceedings of the 2nd Borobudur International Symposium on Science and Technology (BIS-STE 2020)*, 2021, pp. 154-158, doi: 10.2991/aer.k.210810.025.
[19] M. Hatta, "Stemmer bahasa Indonesia dengan pendekatan aturan," *Jurnal Teknologi Pintar (JUTP)*, vol. 2, no. 7, pp. 1-11, 2022. [Online]. Available: http://teknologipintar.org/index.php/teknologipintar/article/view/206.
[20] W. A. Rifai and E. Winarko, "Modification of stemming algorithm using a non deterministic approach to Indonesian text," *Indonesian Journal of Computing and Cybernetics Systems (IJCCS)*, vol. 13, no. 4, pp. 379-388, 2019, doi: 10.22146/ijccs.49072.
[21] A. T. Ni'mah, S. Rochimah, and S. Y. J. Prasetyo, "Autonomy stemmer algorithm for legal and illegal affix detection use finite-state automata method," *EPI International Journal of Engineering*, vol. 2, no. 1, pp. 46-55, 2019, doi: 10.25042/epi-ije.022019.09.
[22] N. Yusliani, R. Primartha, and M. D. Marieska, "Multiprocessing stemming: A case study of Indonesian stemming," *International Journal of Computer Applications*, vol. 182, pp. 15-19, 2019, doi: 10.5120/ijca2019918476.
[23] F. Riza, D. A. Ostheider, and M. A. Wibowo, "Information retrieval technique for Indonesian PDF document with modified stemming Porter method using PHP," *Journal of Physics: Conference Series*, vol. 1477, no. 2, p. 032016, 2020, doi: 10.1088/1742-6596/1477/3/032016.

P.ISSN: 2086 – 4981
E.ISSN: 2620 – 6390
tip.ppj.unp.ac.id