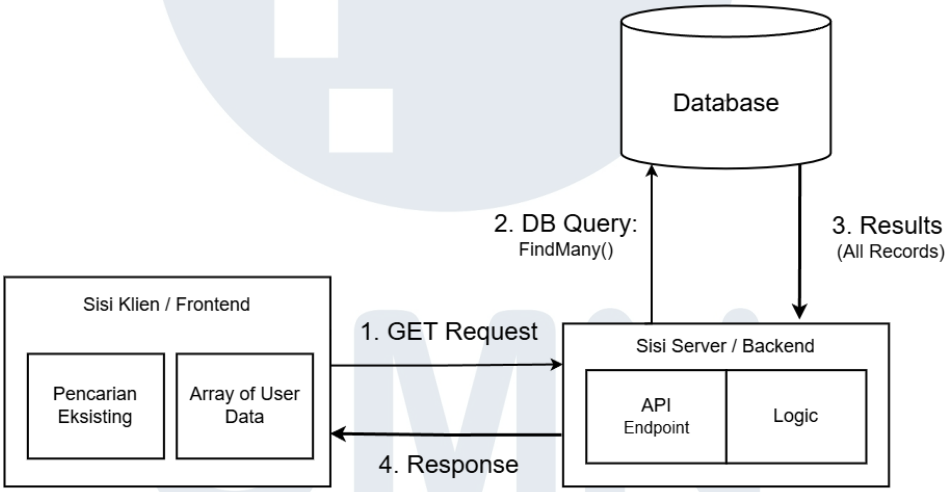
BAB III

METODE PENELITIAN

3.1 Perancangan Solusi

Sebelum memasuki tahapan perancangan solusi, perlu dilakukan analisis terhadap kinerja sistem eksisting, khususnya pada bagian halaman daftar pengguna. Analisis ini penting untuk mengidentifikasi penyebab utama meningkatnya waktu pemuatan data (*load time)* serta memahami bagaimana mekanisme pencarian *substring matching* yang diterapkan saat ini bekerja.

Analisis pertama, analisis difokuskan pada proses penyajian data pada arsitektur eksisting. Alur proses penyajian data pada arsitektur eksisting dapat dilihat pada gambar 3.1.

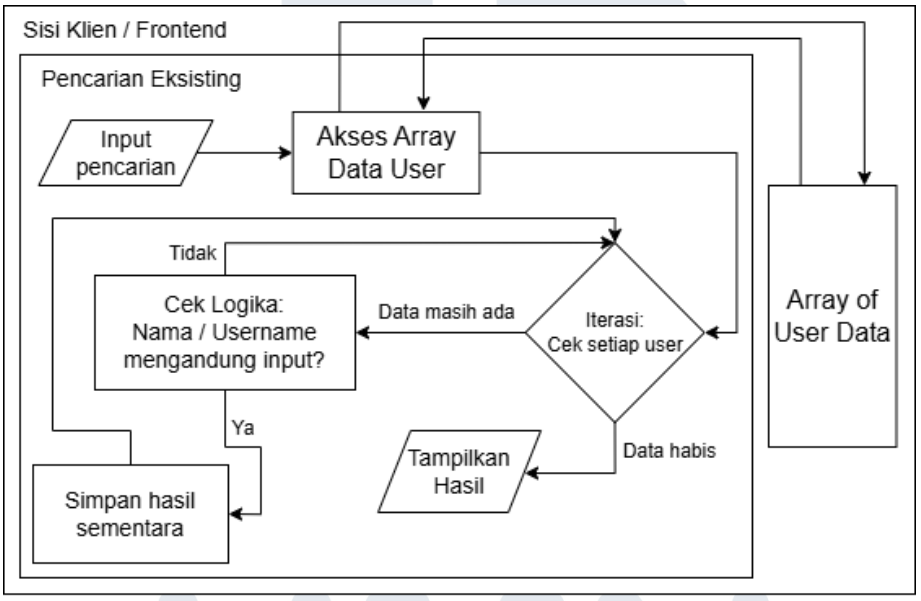


Gambar 3.1 Alur penyajian data pada arsitektur eksisting

Alur proses dimulai dari permintaan (GET *request*) yang dikirimkan oleh sisi klien ketika halaman dibuka, kemudian diteruskan ke server atau *backend*. Pada sisi *backend*, permintaan tersebut diproses dengan menjalankan fungsi findMany() untuk mengambil seluruh data pengguna dari database. Selanjutnya, database mengembalikan seluruh hasil tersebut kepada *backend* untuk kemudian diteruskan kembali dan disimpan dalam bentuk array pada sisi klien yang nantinya data tersebut dipakai untuk ditampilkan. Implementasi ini mengirim seluruh data pengguna dikirim sekaligus tanpa adanya

pembatasan, sehingga semakin banyak data yang tersimpan pada database, semakin lama waktu yang dibutuhkan untuk memproses dan mengirimkannya.

Setelah seluruh data pengguna dimuat dan disimpan di sisi klien melalui mekanisme penyajian data sebelumnya, proses pencarian dilakukan secara langsung di sisi klien. Analisis kedua pada penelitian ini difokuskan pada mekanisme pencarian *substring matching* yang diterapkan pada arsitektur eksisting. Alur proses pencarian *substring matching* pada arsitektur eksisting ditunjukkan pada gambar 3.2.



Gambar 3.2 Alur pencarian *substring matching* pada arsitektur eksisting

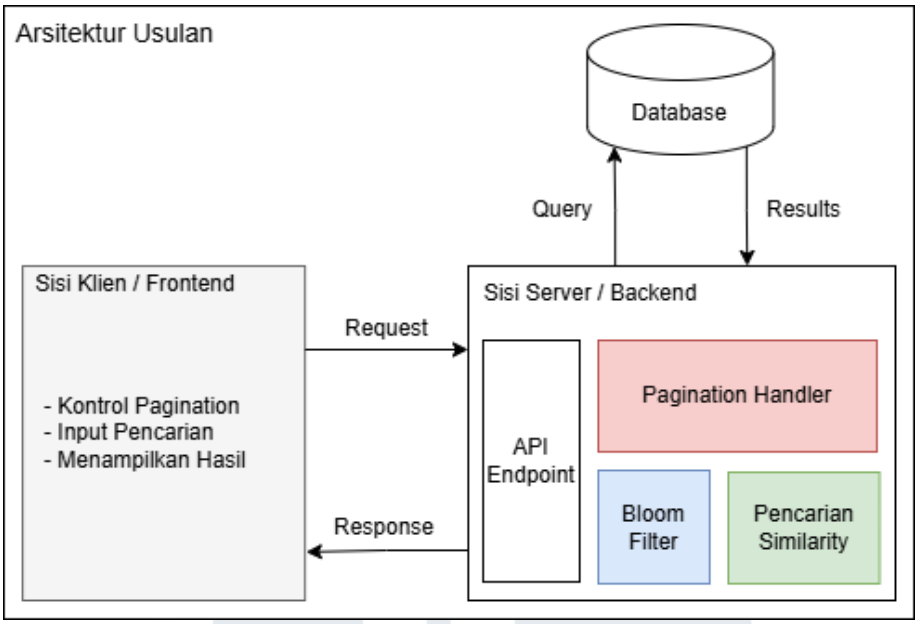
Alur proses pencarian pada sistem eksisting dimulai dari input yang dimasukkan ke dalam pencarian. Setelah itu, di akses array data user yang sudah disiapkan sebelumnya. Selanjutnya dilakukan sebuah iterasi, sistem akan memeriksa setiap data pengguna untuk mencari nama atau username yang "mengandung" kata yang sama dengan input yang diberikan. Jika terdapat kecocokan, data pengguna tersebut akan ditampilkan sebagai hasil pencarian. Namun, jika tidak ada kecocokan, sistem akan menampilkan keluaran kosong atau *no result found*.

Implementasi pencarian ini sangat bergantung pada ketersediaan seluruh data pengguna di sisi klien, yang pada analisis sebelumnya diidentifikasi

sebagai penyebab meningkatnya waktu muat halaman seiring bertambahnya jumlah data. Selain itu, metode *substring matching* memiliki keterbatasan dalam toleransi kesalahan (*fault tolerance*), karena hanya mampu menemukan data yang memiliki kesesuaian karakter. Akibatnya, variasi penulisan nama maupun kesalahan pengetikan (*typo*) sekecil apapun dapat menyebabkan hasil pencarian tidak ditemukan. Di samping itu, proses pemeriksaan kecocokan dilakukan secara satu per satu terhadap seluruh data pengguna, sehingga menimbulkan beban komputasi yang tidak efisien dan semakin tidak optimal ketika jumlah data pengguna meningkat.

Hasil analisis terhadap sistem eksisting menunjukkan dua temuan utama. Pertama, mekanisme penyajian data yang digunakan saat ini mengirim seluruh data pengguna sekaligus tanpa adanya pembatasan, sehingga waktu pemuatan halaman (*load time*) meningkat secara signifikan seiring dengan bertambahnya jumlah pengguna. Kedua, mekanisme pencarian sangat bergantung pada mekanisme penyajian data yang diterapkan. Perubahan pada mekanisme penyajian data akan menimbulkan konsekuensi arsitektural yang mengharuskan mekanisme pencarian turut disesuaikan. Selain itu, mekanisme pencarian eksisting mengandalkan metode *substring matching* yang memiliki keterbatasan dalam toleransi kesalahan input (*fault tolerance*). Variasi penulisan nama maupun kesalahan ejaan (*typo*) berpotensi menghasilkan hasil pencarian yang tidak sesuai.

Berdasarkan hasil analisis terhadap sistem eksisting, penelitian ini merancang sebuah solusi berupa arsitektur baru, sebagaimana ditunjukkan pada gambar 3.3.



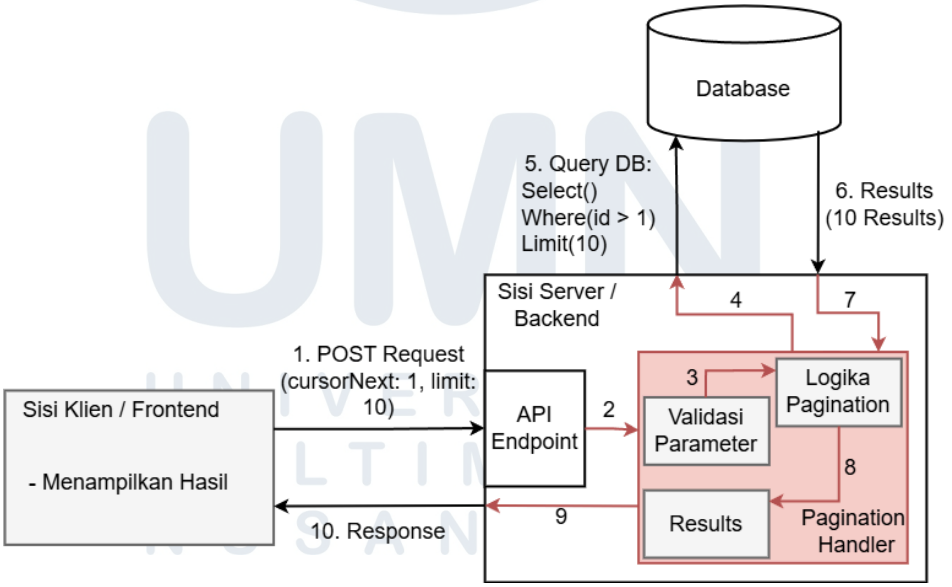
Gambar 3.3 Bentuk arsitektur usulan

Arsitektur yang diusulkan dirancang secara menyeluruh untuk mengatasi permasalahan performa waktu muat halaman (*load time*) serta keterbatasan mekanisme pencarian pada sistem eksisting (*fault tolerance*), sehingga solusi yang dihasilkan merupakan satu kesatuan arsitektur yang baru.

Arsitektur sistem yang diusulkan terdiri dari tiga komponen utama yang berada pada *backend*. Komponen pertama adalah mekanisme penyajian data berbasis *pagination* yang berfokus pada perbaikan performa waktu muat halaman (*load time*) dengan membatasi jumlah data yang dikirim ke sisi klien. Komponen kedua adalah mekanisme pencarian berbasis *similarity detection* yang dirancang untuk menangani variasi input dan kesalahan pengetikan (*fault tolerance*) sebagai konsekuensi dari pemindahan proses pencarian ke sisi *backend*. Komponen ketiga adalah penerapan struktur data Bloom Filter sebagai lapisan optimasi untuk melakukan *early rejection* terhadap kueri pencarian yang tidak ada dalam database (*non-existing elements*), sehingga proses pencarian *similarity* tidak perlu dilakukan untuk menghemat waktu dan komputasi.

3.1.1 Perancangan Komponen Penyajian Data

Perancangan komponen penyajian data pada penelitian ini akan berfokus untuk mengatasi masalah pemuatan data yang terjadi pada arsitektur eksisting. Berdasarkan hasil tinjauan pustaka, salah satu hal yang esensial dalam pengembangan REST API adalah penggunaan metode *pagination*, seperti dijelaskan oleh Viktor Bogutskii (2025) dalam penelitiannya mengenai *high-load systems*. Sejalan dengan temuan tersebut, komponen penyajian data usulan pada penelitian ini menerapkan metode *Cursor-based pagination*, metode ini bekerja dengan membatasi jumlah data yang dikirimkan melalui parameter limit serta menentukan posisi data melalui cursorNext atau cursorPrev. CursorNext memberikan data lebih dari *cursor* sebanyak limit, sedangkan cursorPrev memberikan data kurang dari *cursor* sebanyak limit. Dengan demikian, sistem hanya menyajikan data dalam jumlah yang diperlukan pada setiap permintaan, bukan seluruh data pengguna sekaligus. Metode *pagination* dipilih karena mampu mengurangi beban komputasi dan mempercepat waktu pemuatan halaman (*load time*). Alur komponen penyajian data usulan pada penelitian ini dapat dilihat pada gambar 3.4.

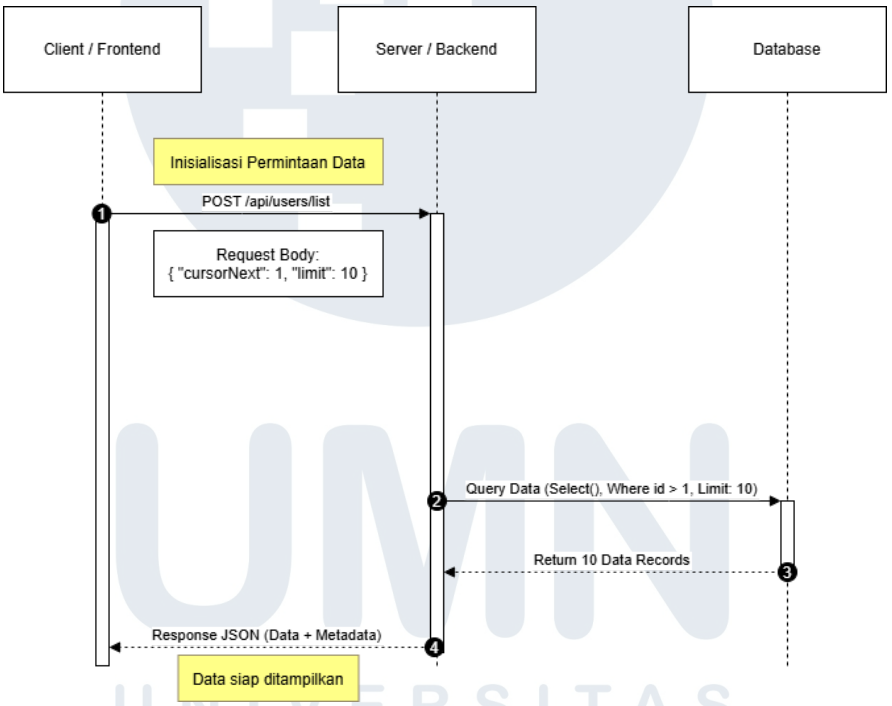


Gambar 3.4 Alur komponen penyajian data usulan

Alur komponen penyajian data usulan dimulai dari sisi klien yang mengirimkan permintaan ke *backend* melalui *endpoint* /listing/users/pagination-cursor menggunakan metode POST. Permintaan

tersebut membawa parameter limit serta cursorNext atau cursorPrev yang digunakan untuk menentukan jumlah data yang dimuat dan posisi data yang akan ditampilkan. Sebelum diproses oleh logika *pagination*, parameter yang diterima terlebih dahulu divalidasi untuk memastikan kesesuaian tipe data. Setelah proses validasi berhasil, sistem menjalankan logika *pagination* dengan melakukan kueri ke database berdasarkan parameter yang diberikan. Hasil kueri kemudian dikembalikan ke sisi klien dalam format JSON untuk ditampilkan pada antarmuka pengguna.

Alur interaksi antara sisi klien dan *backend* pada implementasi penyajian data usulan ini digambarkan dalam bentuk *sequence diagram* yang ditunjukkan pada gambar 3.5.



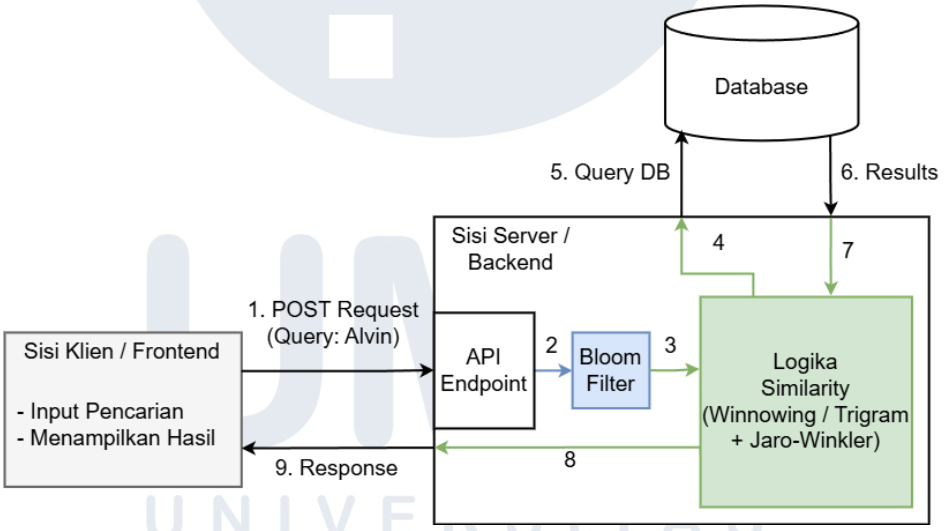
Gambar 3.5 *Sequence diagram* implementasi penyajian data usulan

Dalam penerapannya, metode POST digunakan karena sisi klien perlu mengirimkan parameter limit dan cursor sebagai bagian dari *request body*, yang kemudian diproses oleh server untuk mengembalikan data sesuai dengan kriteria yang diminta. Pendekatan ini memungkinkan proses penyajian data berjalan lebih terkontrol dan efisien, serta lebih siap dalam mendukung peningkatan jumlah pengguna di masa mendatang.

3.1.2 Perancangan Komponen Pencarian *Similarity*

Karena komponen penyajian data pada arsitektur usulan tidak lagi memuat seluruh data pengguna ke sisi klien, diperlukan penyesuaian arsitektural pada fitur pencarian. Pada tahap ini, perancangan komponen pencarian *similarity* difokuskan pada pemindahan proses pencarian dari sisi klien ke *backend*. Selain itu, pendekatan pencarian ini dirancang untuk mengatasi keterbatasan toleransi kesalahan (*fault tolerance*) pada mekanisme pencarian eksisting berbasis *substring matching*, khususnya dalam menangani variasi penulisan nama dan kesalahan ejaan (*typo*).

Secara keseluruhan, komponen pencarian *similarity* pada arsitektur usulan dilengkapi dengan mekanisme *pre-filtering* menggunakan Bloom Filter sebagai lapisan awal untuk menyaring kueri yang tidak memiliki kemungkinan kecocokan sebelum proses pencarian lanjutan dijalankan. Alur komponen pencarian *similarity* ditunjukkan pada gambar 3.6.



Gambar 3.6 Alur komponen pencarian *similarity*

Penelitian ini merancang dua pendekatan algoritmik untuk mendukung pencarian yang lebih fleksibel, yaitu algoritma *Winnowing* serta kombinasi *Trigram* dan Jaro-Winkler. Kedua pendekatan tersebut akan dievaluasi dan dibandingkan dengan mekanisme pencarian eksisting (*substring matching*) melalui berbagai skenario pengujian guna

menentukan metode yang paling baik dalam menangani kasus pencarian dengan input yang bervariasi atau mengandung kesalahan pengetikan.

Pemilihan *Winnowing* serta kombinasi *Trigram* dan Jaro-Winkler dilakukan dengan mempertimbangkan karakteristik pencarian nama pengguna pada sistem Talent Discovery. Walaupun *Winnowing* lazim digunakan pada dokumen panjang, algoritma ini tetap relevan untuk nama manusia yang relatif pendek karena mampu menghasilkan *fingerprint* representatif dengan parameter *k* dan *w* yang tepat, *robust* terhadap *typo*, serta sangat efisien ketika *fingerprint* telah diindeks.

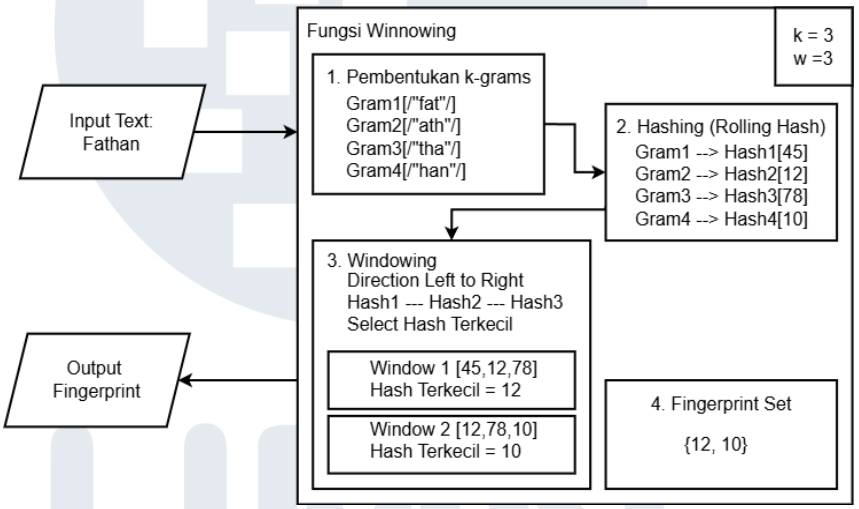
Sebaliknya, pendekatan *Trigram* dan Jaro-Winkler dipilih karena keduanya secara empiris terbukti unggul dalam menangani *fuzzy name matching*, *Trigram* efektif sebagai *candidate generator* pada teks pendek, sementara Jaro-Winkler memberikan skor kemiripan yang stabil dan *prefix-sensitive*, yang sangat penting dalam pencarian nama. Penelitian ini tidak menggunakan Levenshtein Distance karena kompleksitas komputasinya yang lebih tinggi, kurangnya sensitivitas terhadap kesamaan *prefix*, serta performa yang kurang stabil pada string pendek. Selain itu, Levenshtein tidak menyediakan mekanisme indexing yang efisien untuk skala dataset besar, sehingga tidak sesuai dengan kebutuhan sistem yang menuntut pencarian cepat dengan latensi rendah.

3.1.2.1 Perancangan Komponen Pencarian *Similarity Winnowing*

Perancangan komponen pencarian *similarity* yang pertama menggunakan algoritma *Winnowing*. Berdasarkan hasil tinjauan pustaka, algoritma *Winnowing* terbukti efektif dalam mendeteksi tingkat kesamaan teks secara efisien, seperti yang diimplementasikan oleh Puspaningrum et al. (2020), dalam penelitian berjudul "*Detection of Text Similarity for Indication Plagiarism Using Winnowing Algorithm Based K-gram and Jaccard Coefficient.*". *Winnowing* bekerja dengan memecah teks menjadi rangkaian karakter (*k-gram*), kemudian mengubah setiap

k-gram menjadi nilai hash. Selanjutnya, nilai-nilai hash tersebut diproses melalui mekanisme *windowing* untuk memilih sebagian kecil nilai yang dianggap paling representatif, yang kemudian membentuk "sidik jari" (*fingerprint*) dari sebuah dokumen. *Fingerprint* inilah yang digunakan untuk membandingkan tingkat kemiripan antara dua teks.

Perancangan komponen dimulai dari pembentukan fungsi *Winnowing* sebagai komponen utama. Fungsi ini bertujuan menghasilkan *fingerprint* yang merepresentasikan pola karakteristik dari sebuah input teks. Alur fungsi *Winnowing* dapat dilihat pada gambar 3.7.



Gambar 3.7 Alur fungsi *Winnowing*

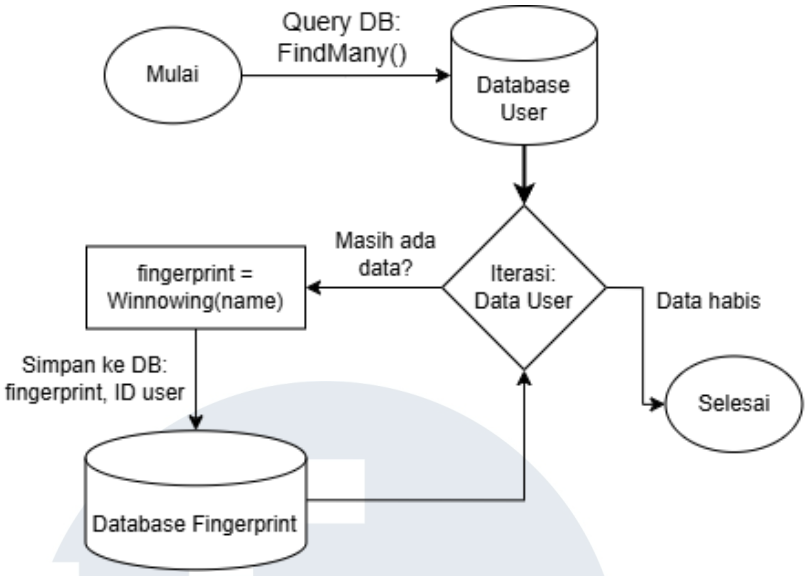
Fungsi Winnowing diawali dengan menerima input berupa teks yang kemudian di normalisasi dengan tujuan menghilangkan karakter yang tidak diperlukan, seperti tanda baca dan simbol non-alfanumerik, serta mengubah seluruh huruf menjadi huruf kecil. Normalisasi ditujukan agar proses pembentukan *k-grams* konsisten pada seluruh input. Pembentukan *k-grams* bertujuan untuk merepresentasikan struktur lokal dari teks dan memungkinkan pendeteksian kemiripan meskipun terjadi pergeseran atau variasi kecil pada susunan karakter.

Setelah itu, setiap *k-gram* kemudian dikonversi menjadi nilai numerik menggunakan metode *rolling hash*. Pada penelitian ini digunakan konstanta *base* = 101 dan *mod* = $10^9 + 7$ untuk menghasilkan nilai hash yang stabil serta meminimalkan potensi terjadinya tabrakan hash (*collision*). *Rolling hash* memungkinkan perhitungan nilai hash *k-gram* dilakukan secara efisien saat jendela bergeser, tanpa menghitung ulang seluruh substring.

Tahap selanjutnya adalah proses *windowing*, di mana deretan nilai hash dari *k-grams* dikelompokkan ke dalam jendela (*window*) dengan ukuran tertentu dan digeser secara satu arah dari kiri ke kanan (*left-to-right*). Pada setiap *window*, dipilih satu nilai hash terkecil sebagai representasi jendela tersebut. Nilai hash terpilih dari setiap window kemudian dikumpulkan sebagai *fingerprint* dokumen. Kumpulan *fingerprint* inilah yang digunakan sebagai representasi ringkas dari teks, sehingga perbandingan kemiripan antar teks dapat dilakukan secara efisien tanpa harus membandingkan keseluruhan isi teks secara langsung.

Fungsi *Winnowing* pada penelitian ini menggunakan parameter $k = 3$ dan $w = 3$. Pemilihan nilai tersebut dilakukan setelah serangkaian percobaan awal yang menunjukkan bahwa $k = 3$ mampu menghasilkan tingkat toleransi kesalahan yang baik tanpa meningkatkan risiko *false positive* secara signifikan. Sementara itu, nilai $w = 3$ memberikan keseimbangan antara jumlah *fingerprint* yang dihasilkan dan efisiensi proses pencocokan, sehingga tetap mempertahankan performa pencarian pada skala data yang besar.

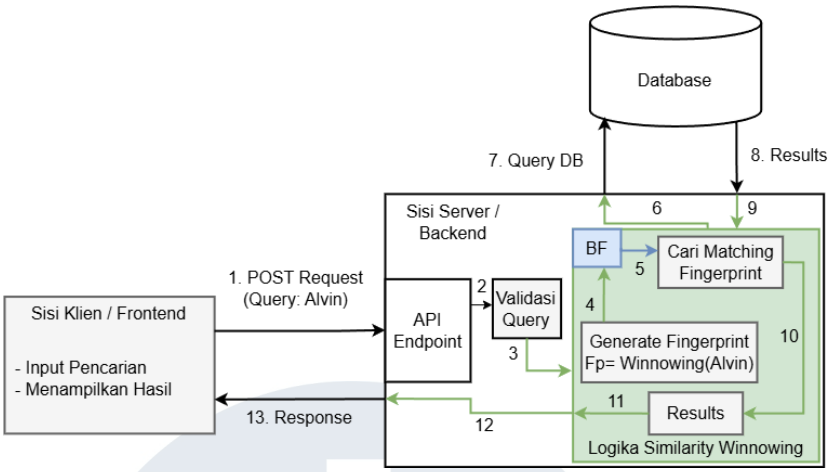
Selanjutnya, dilakukan proses indeksasi atau *seeding data fingerprint*, yang bertujuan menghasilkan *fingerprint* untuk seluruh data nama pengguna yang telah tersimpan di dalam database. Alur skrip *seeding data fingerprint* dapat dilihat pada gambar 3.8.



Gambar 3.8 Alur skrip *seeding data fingerprint*

Proses skrip *seeding data fingerprint* dimulai dengan mengambil seluruh data pengguna dari database. Selanjutnya, sistem melakukan iterasi pada array data pengguna tersebut. Untuk setiap entri pengguna, fungsi *Winnowing* dipanggil dengan input berupa nama pengguna, lalu hasil *fingerprint* yang dihasilkan disimpan ke dalam tabel database baru yang berisi pasangan *fingerprint* dan ID pengguna. Penggunaan tabel terpisah ini bertujuan untuk memisahkan proses indeksasi *fingerprint* dari tabel pengguna utama sehingga pencarian dapat dilakukan secara lebih efisien tanpa mempengaruhi struktur data yang sudah ada. Selain itu, penyimpanan *fingerprint* dalam tabel khusus memungkinkan proses pencarian dijalankan secara cepat dan terstruktur.

Tahap terakhir adalah pembuatan *endpoint* untuk komponen pencarian *similarity* menggunakan algoritma *Winnowing*, di mana *fingerprint* dari input pencarian dibandingkan dengan *fingerprint* data yang telah diindeks untuk menentukan tingkat kemiripan. Alur komponen pencarian *similarity winnowing* ditunjukkan pada gambar 3.9.



Gambar 3.9 Alur komponen pencarian *similarity winnowing*

Alur komponen pencarian *similarity winnowing* dimulai dari sisi klien yang mengirimkan permintaan ke *backend* melalui *endpoint* /search/users/search-winnow menggunakan metode POST. Dengan parameter berisi kueri atau kata kunci yang kemudian divalidasi lalu diproses oleh fungsi *Winnowing* untuk menghasilkan *fingerprint* yang menjadi representasi dari karakteristik kata kunci. Selanjutnya, *fingerprint* hasil pemrosesan digunakan untuk mencari kecocokan pada tabel Fingerprint di database, yaitu dengan melakukan pencarian terhadap pengguna yang memiliki nilai *fingerprint* yang sama atau serupa. Apabila ditemukan kecocokan, sistem akan mengembalikan daftar pengguna yang relevan beserta nilai kesesuaian (*match count*) sebagai bagian dari response. Sebaliknya, jika tidak ditemukan kecocokan, server akan mengembalikan response *not found* yang menandakan bahwa tidak ada pengguna yang sesuai dengan kriteria pencarian.

3.1.2.2 Perancangan Komponen Pencarian *Similarity* Kombinasi *Trigram* dan Jaro-Winkler

Perancangan komponen pencarian *similarity* yang kedua menggunakan kombinasi algoritma *Trigram* dan Jaro-Winkler. Berbeda dengan *Winnowing*, pendekatan *Trigram* dan

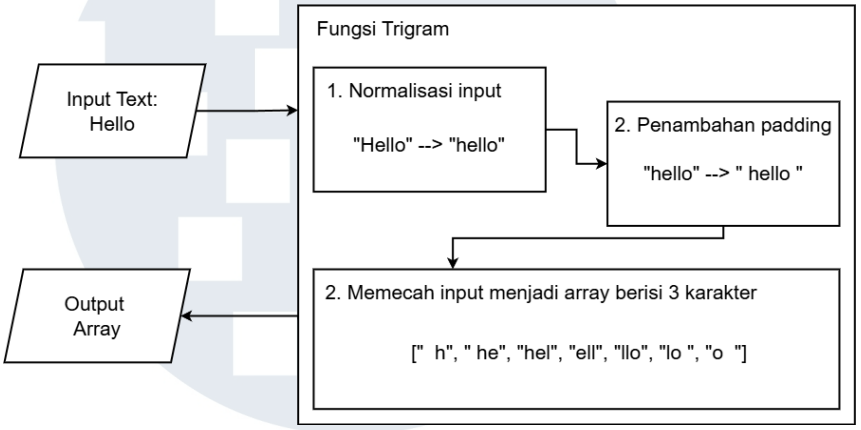
Jaro-Winkler memerlukan dua tahap yang saling melengkapi. Pada perancangan ini, *Trigram* berperan sebagai *candidate generator* yang menghasilkan daftar kandidat nama dengan tingkat kemiripan leksikal berdasarkan pemecahan teks menjadi tiga karakter berurutan. Selanjutnya, algoritma Jaro-Winkler digunakan untuk menghitung skor *similarity* yang lebih presisi dari kandidat tersebut, sehingga diperoleh nilai *similarity* akhir yang mencerminkan kemiripan sebenarnya antara kata kunci pencarian dan nama pengguna.

Berdasarkan hasil tinjauan pustaka, menurut studi komparatif yang dilakukan oleh Akinwale dan Niewiadomski (2015) dalam penelitian berjudul "*Efficient Similarity Measures for Texts Matching*", *Trigram* merupakan salah satu algoritma *n-gram* yang memiliki performa sangat baik dalam menangani kesalahan ejaan (*typo*). *Trigram* bekerja dengan cara memecah sebuah kata menjadi unit-unit tiga karakter berurutan, sehingga pola karakter yang terbentuk dapat digunakan untuk mengidentifikasi kesamaan leksikal meskipun terdapat perbedaan kecil dalam penulisan. Pendekatan ini memungkinkan sistem menghasilkan kandidat nama yang secara struktur memiliki kemiripan dengan kata kunci pencarian, sehingga dapat menjadi dasar yang kuat sebelum dilakukan penghitungan skor *similarity* yang lebih presisi menggunakan algoritma Jaro-Winkler.

Begitu pula dengan algoritma Jaro-Winkler. Berdasarkan hasil tinjauan pustaka, penelitian berjudul "*A Comparison of Techniques for Name Matching*" yang dilakukan oleh Peng et al. (2012) menunjukkan bahwa Jaro-Winkler memiliki kinerja paling unggul dibandingkan lima algoritma *string matching* lainnya yang dievaluasi. Keunggulan utama Jaro-Winkler terletak pada mekanisme yang memberikan bobot lebih tinggi pada kesamaan awalan (*prefix*). Selain itu, penelitian tersebut juga menyebutkan

bahwa Jaro-Winkler membutuhkan waktu pemrosesan yang paling rendah dibandingkan algoritma lainnya, sehingga menjadikannya salah satu metode yang paling efisien secara komputasi untuk *string matching*.

Perancangan komponen dimulai dari pembentukan fungsi *Trigram* dan Jaro-Winkler sebagai dua komponen utama. Pertama, fungsi *Trigram* bertujuan menghasilkan daftar kandidat nama yang memiliki kemiripan leksikal dengan input. Alur fungsi *Trigram* dapat dilihat pada gambar 3.10.



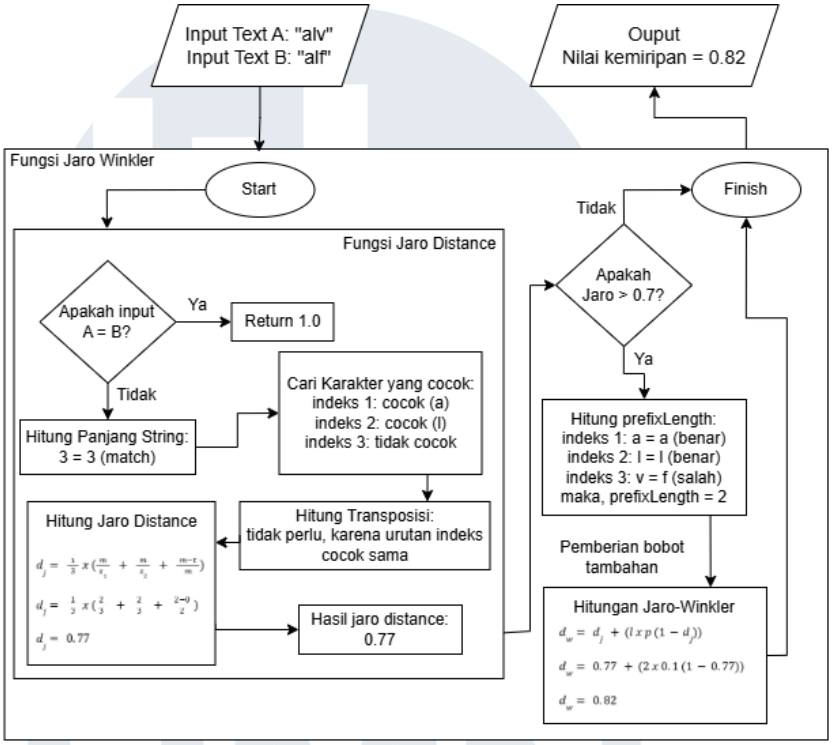
Gambar 3.10 Alur fungsi *Trigram*

Fungsi *Trigram* diawali dengan menerima input berupa nama atau teks yang kemudian dinormalisasi. Proses normalisasi dilakukan untuk menghapus karakter yang tidak diperlukan, seperti tanda baca dan simbol non-alfanumerik, serta mengubah seluruh huruf menjadi huruf kecil. Langkah ini bertujuan menjaga konsistensi pemrosesan sehingga perbandingan hanya dilakukan terhadap karakter yang relevan.

Setelah itu, sistem menambahkan *padding* pada bagian awal dan akhir teks. *Padding* digunakan untuk mempertahankan informasi posisi awal dan akhir kata, sehingga proses perhitungan kemiripan tetap sensitif terhadap perubahan pada *prefix* maupun *suffix*. Tahap terakhir adalah pemecahan teks yang telah dinormalisasi ke dalam unit tiga karakter berurutan. Hasil dari

proses ini berupa array *Trigram* yang akan digunakan sebagai dasar dalam menghasilkan kandidat nama yang memiliki kemiripan leksikal dengan input.

Kedua, fungsi Jaro-Winkler bertujuan untuk menghitung skor *similarity* dari kandidat yang sudah dibuat oleh *Trigram*. Alur fungsi Jaro-Winkler dapat dilihat pada gambar 3.11.

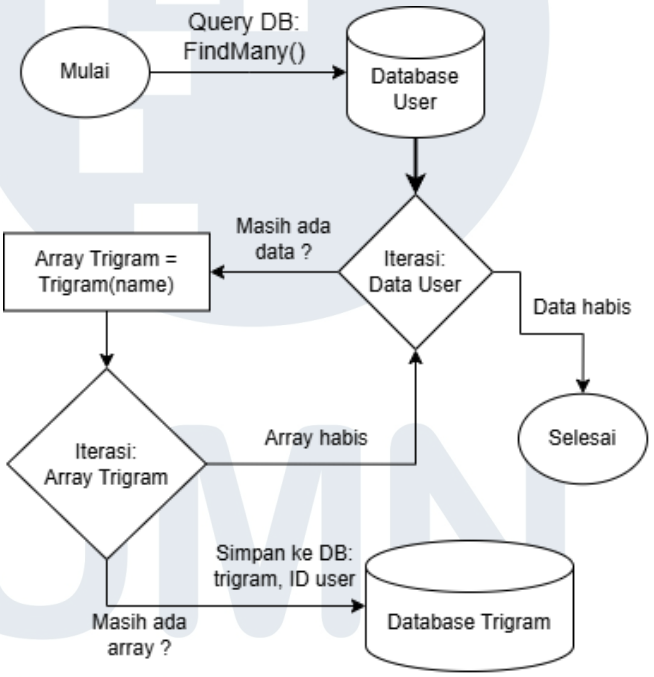


Gambar 3.11 Alur fungsi Jaro-Winkler

Fungsi Jaro-Winkler menerima dua buah input string yang akan dihitung tingkat kemiripannya berdasarkan pendekatan Jaro Distance sebagai dasar perhitungan. Jaro-Winkler, sebagai modifikasi dari Jaro Distance, memberikan bobot tambahan (*weight boosting*) apabila nilai Jaro Distance melebihi ambang batas tertentu. Melalui beberapa pengetesan nilai 0.7 adalah ambang batas umum untuk menunjukkan kecocokan yang signifikan. Setelah nilai Jaro Distance diperoleh, sistem menghitung prefixLength, yaitu panjang awalan yang sama dari kedua string hingga maksimum empat karakter. Nilai prefixLength

ini kemudian digunakan sebagai faktor penambah bobot dalam rumus Jaro-Winkler. Hasil akhir dari fungsi ini berupa nilai kemiripan antara dua string yang telah mempertimbangkan baik proporsi kemiripan keseluruhan maupun kesamaan pada bagian awal kata.

Sama seperti implementasi pencarian *Winnowing*, proses pencarian menggunakan kombinasi *Trigram* dan Jaro-Winkler juga memerlukan proses indeksasi atau *seeding data Trigram*, yang bertujuan untuk menghasilkan representasi *trigram* dari seluruh data nama pengguna yang tersimpan dalam database. Alur dari skrip *seeding data Trigram* dapat dilihat pada gambar 3.12.

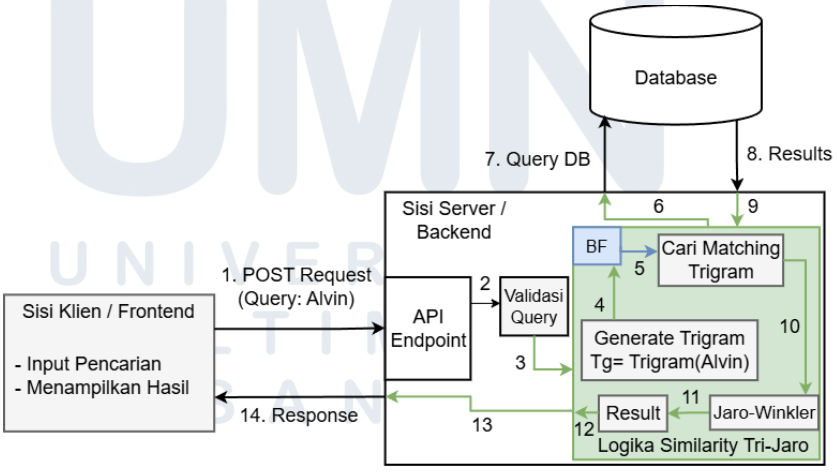


Gambar 3.12 Alur fungsi skrip *seeding data Trigram*

Proses *seeding data Trigram* dimulai dengan pengambilan seluruh data pengguna dari database. Setelah data diperoleh, sistem melakukan iterasi terhadap setiap entri pengguna. Untuk setiap nama pengguna, fungsi *Trigram* dipanggil untuk menghasilkan array *trigram* yang merepresentasikan struktur leksikal nama tersebut. Setiap *trigram* kemudian disimpan ke dalam tabel khusus yang berisi pasangan antara *trigram* dan ID

pengguna. Penggunaan tabel terpisah ini bertujuan memisahkan hasil indeksasi *trigram* dari tabel utama pengguna, sehingga proses pencarian dapat dilakukan secara lebih efisien tanpa membebani struktur data inti. Selain itu, penyimpanan *trigram* ke dalam tabel indeks memungkinkan proses pencarian kandidat berjalan lebih cepat, terstruktur, dan mudah dioptimalkan pada tahap pemrosesan berikutnya.

Tahap terakhir adalah pembuatan *endpoint* untuk komponen pencarian *similarity* menggunakan kombinasi *Trigram* dan Jaro-Winkler, di mana *trigram* yang dihasilkan dari input pencarian dibandingkan dengan *trigram* yang telah diindeks sebelumnya untuk memperoleh daftar kandidat nama. Setelah kandidat diperoleh, setiap kandidat dihitung nilai kemiripannya menggunakan fungsi Jaro-Winkler untuk menghasilkan skor *similarity* yang lebih presisi. Skor ini digunakan untuk menentukan urutan prioritas hasil pencarian, mulai dari kandidat dengan tingkat kemiripan tertinggi hingga yang terendah. Alur komponen pencarian *similarity* kombinasi *Trigram* dan jaro-winkler dapat dilihat pada gambar 3.13.



Gambar 3.13 Alur komponen pencarian *similarity* kombinasi *Trigram* dan Jaro-Winkler

Alur komponen pencarian *similarity* kombinasi *Trigram* dan Jaro-Winkler dimulai dari sisi klien yang mengirimkan

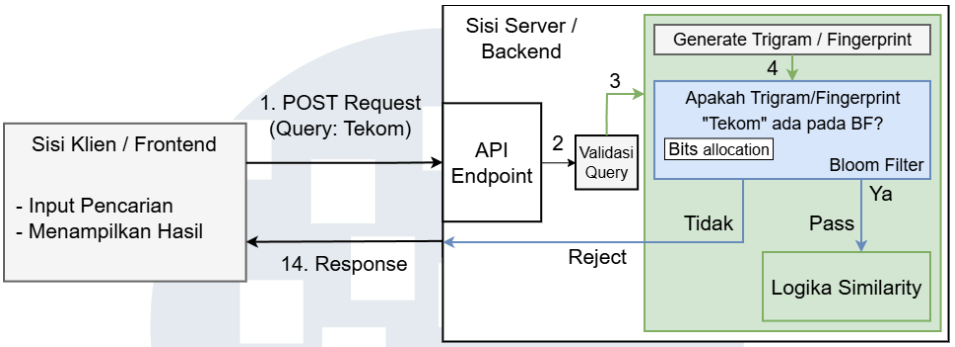
permintaan ke *backend* melalui *endpoint* /search/users/search-tri-jaro menggunakan metode POST, dengan parameter berisi kueri atau kata kunci yang kemudian divalidasi. Selanjutnya, kata kunci tersebut diproses oleh fungsi *Trigram* untuk menghasilkan array *trigram* sebagai representasi leksikal dari input. *Trigram* hasil pemrosesan kemudian digunakan untuk melakukan pencarian pada tabel indeks *Trigram* di database untuk menemukan pengguna yang memiliki *trigram* serupa.

Apabila ditemukan kandidat pengguna berdasarkan kesesuaian *trigram*, tahap berikutnya adalah menghitung nilai kemiripan menggunakan algoritma Jaro-Winkler. Perhitungan diawali dengan Jaro Distance untuk memperoleh nilai kemiripan dasar, kemudian kandidat dengan nilai kemiripan di bawah 0,7 dieliminasi karena dianggap tidak cukup relevan. Setelah itu, algoritma Jaro-Winkler digunakan untuk memberikan bobot tambahan pada kandidat yang memiliki kesamaan awalan, dan hasil skor *similarity* ini digunakan untuk mengurutkan pengguna dari tingkat kemiripan tertinggi hingga terendah sebelum dikembalikan kepada klien. Jika tidak ditemukan kecocokan trigram pada tahap awal, server akan mengembalikan response *not found* yang menandakan bahwa tidak ada pengguna yang sesuai dengan kata kunci pencarian.

3.1.3 Perancangan Komponen Bloom Filter

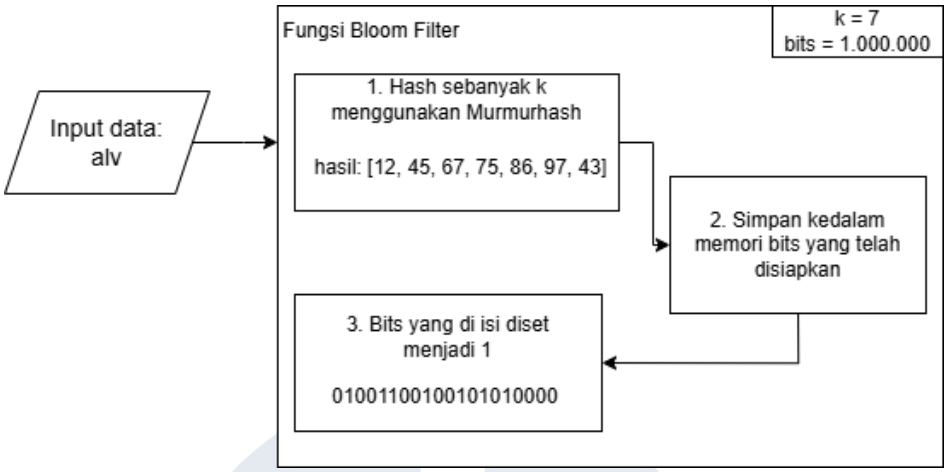
Komponen Bloom Filter berfungsi sebagai lapisan *pre-filter* sebelum proses pencarian *similarity*, yang bertujuan memotong komputasi yang tidak diperlukan terhadap input yang tidak terdapat dalam database (*non-existing elements*). Berdasarkan hasil tinjauan pustaka, penelitian berjudul "*Building Verified Neural Networks for Computer Systems with Ouroboros*" oleh Wei et al. (2023) menunjukkan bahwa Bloom Filter merupakan metode yang sangat efektif dalam mendeteksi elemen yang tidak ada dalam himpunan dibandingkan pendekatan

lainnya. Bloom Filter mampu melakukan deteksi dengan latensi yang sangat rendah karena hanya memerlukan operasi hash sederhana tanpa perlu mengakses database secara langsung. Kemampuan ini menjadikan Bloom Filter pilihan ideal sebagai mekanisme *early rejection*. Alur Bloom Filter pada *endpoint* pencarian *similarity* dapat dilihat pada gambar 3.13.



Gambar 3.13 Alur Bloom Filter pada *endpoint* pencarian *similarity*

Bloom Filter ditempatkan sebagai tahap awal sebelum proses logika *similarity*, baik pada pencarian menggunakan *Winnowing* maupun pada kombinasi *Trigram* dan Jaro-Winkler. Dengan pendekatan ini, arsitektur usulan dapat langsung menolak (*early rejection*) kata kunci yang dipastikan tidak mungkin cocok dengan data mana pun di dalam database, sehingga komputasi berat seperti pencarian *fingerprint* atau perhitungan *similarity* tidak perlu dijalankan. Penempatan Bloom Filter sebagai lapisan awal ini secara signifikan meningkatkan efisiensi, terutama saat menangani kueri yang salah eja parah atau benar-benar tidak ada dalam database. Cara kerja Bloom Filter dapat dilihat pada gambar 3.14.

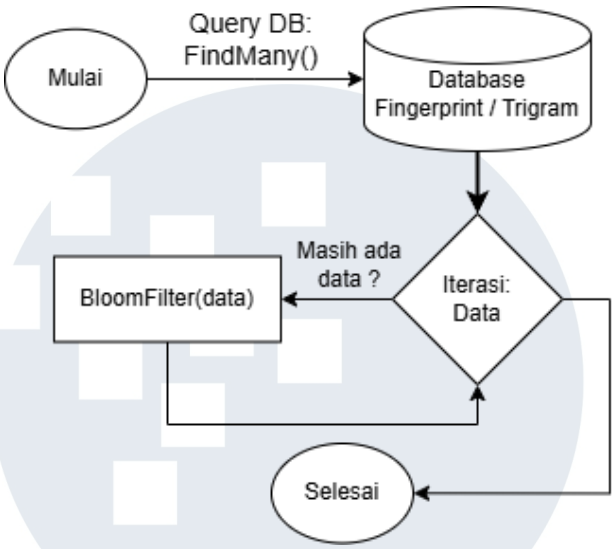


Gambar 3.14 Cara kerja Bloom Filter

Bloom Filter bekerja dengan cara mengalokasikan memori berupa bit array yang digunakan untuk menyimpan hasil proses hashing. Ketika sebuah data dimasukkan ke dalam Bloom Filter, nilai tersebut diubah menggunakan fungsi hash *MurmurHash*, yaitu algoritma hash non-kriptografis yang dirancang untuk memberikan distribusi nilai hash yang baik dengan kecepatan tinggi. Hasil hash tersebut kemudian digunakan untuk menghasilkan sejumlah k indeks, dan setiap indeks yang diperoleh diset menjadi 1 pada bit array. Bit yang belum pernah terisi tetap bernilai 0.

Pada saat Bloom Filter digunakan untuk melakukan pengecekan, sistem akan menjalankan kembali proses hashing yang sama terhadap nilai input untuk mendapatkan k indeks yang relevan. Apabila seluruh bit pada indeks tersebut bernilai 1, maka elemen tersebut mungkin ada di dalam himpunan. Sebaliknya, apabila terdapat satu saja bit bernilai 0, maka elemen tersebut dipastikan tidak ada. Dengan mekanisme ini, Bloom Filter mampu melakukan *early rejection* tanpa perlu melakukan lookup secara langsung ke database, sehingga proses pengecekan dapat dilakukan secara jauh lebih cepat dan efisien. Namun, efisiensi ini datang dengan kompromi berupa potensi *false positive*, yaitu kondisi ketika Bloom Filter menyatakan suatu elemen mungkin ada padahal sebenarnya tidak terdapat dalam himpunan.

Agar Bloom Filter dapat mendeteksi *non-existing elements*, diperlukan proses inisialisasi data. Proses ini bertujuan untuk memuat seluruh *fingerprint* atau *trigram* yang telah tersimpan di dalam database ke dalam struktur Bloom Filter. Alur inisialisasi Bloom Filter ditunjukkan pada gambar 3.15.



Gambar 3.15 Alur insialisasi Bloom Filter

Proses inisialisasi Bloom Filter dimulai dengan mengambil seluruh data yang relevan dari database, yaitu data *fingerprint* untuk pencarian menggunakan *Winnowing* atau data *trigram* untuk pencarian menggunakan kombinasi *Trigram* dan Jaro-Winkler. Setelah itu, data tersebut diproses satu per satu dan dimasukkan ke dalam Bloom Filter melalui fungsi hashing yang telah ditentukan. Dengan memasukkan seluruh *fingerprint* atau *trigram* ke dalam Bloom Filter pada tahap awal ini, Bloom Filter mampu melakukan *early rejection* terhadap input yang tidak terdapat di dalam database (*non-existing elements*).

3.2 Metode Pengujian

Metode pengujian dirancang untuk memvalidasi efektivitas arsitektur baru yang diusulkan serta membandingkannya dengan implementasi eksisting secara kuantitatif. Berdasarkan tujuan penelitian, prosedur pengujian diklasifikasikan ke dalam dua kategori utama. Pertama adalah pengujian performa yang mencakup dua skenario spesifik, pengukuran kecepatan waktu

muat (*load time*) pada mekanisme penyajian data, serta pengukuran waktu komputasi (*computation time*) pencarian dengan mekanisme Bloom Filter dalam melakukan *early rejection* terhadap data yang tidak tersedia (*non-existing elements)*. Kedua adalah pengujian *fault tolerance*, yang difokuskan untuk mengevaluasi dan membandingkan batasan toleransi kesalahan input pencarian eksisting (*substring matching*), pencarian *similarity Winnowing* serta kombinasi *Trigram* dan Jaro-Winkler dalam menangani variasi ejaan dan kesalahan ejaan (*typo*). Seluruh rangkaian pengujian dilakukan pada lingkungan pengujian *end-to-end* dengan pengendalian pada sisi konfigurasi sistem dan skenario uji, tanpa mengeliminasi variabel jaringan yang merepresentasikan kondisi penggunaan nyata.

3.2.1 Lingkungan Pengujian

Seluruh rangkaian pengujian, baik pengujian performa maupun *fault tolerance*, menggunakan arsitektur *client–server* yang merepresentasikan kondisi penggunaan sistem secara nyata (*end-to-end environment*). Lingkungan pengujian terdiri atas dua komponen utama, yaitu mesin penguji (*testing machine*) dan server aplikasi (*host machine*). Mesin penguji digunakan untuk menjalankan alat pengujian, spesifikasi mesin penguji (*testing machine*) dapat dilihat pada tabel 3.1.

Tabel 3.1 Spesifikasi *testing machine*

Komponen	Spesifikasi
Perangkat Keras	
Processor	AMD Ryzen 5 5600H
RAM	16 GB DDR4 3200 MHz
Penyimpanan	512 GB NVMe SSD
Perangkat Lunak	
Sistem Operasi	Windows 11 Home 64-bit
Tools Pengujian	
Load Testing	Grafana k6 (dijalankan melalui

	Docker Container)
Development IDE	Visual Studio Code

Sedangkan aplikasi *backend* yang menjadi objek pengujian di-*host* pada server milik MNP. Dengan pendekatan ini, pengujian dapat merepresentasikan interaksi sistem melalui jaringan sebagaimana terjadi pada skenario penggunaan aktual. Spesifikasi yang digunakan sebagai *host machine* selama pengujian dapat dilihat pada tabel 3.2.

Tabel 3.2 Spesifikasi *host machine*

Komponen	Spesifikasi
CPU	4 vCPU
Memori	4 GiB RAM
Penyimpanan	10 GB
Perangkat Lunak	
Sistem Operasi	Linux Ubuntu
Hosting Control Panel	Plesk Web Pro Edition
Runtime Environment	Node v23.11.1
Database	MariaDB 10.3.39

Pada pengujian performa, Grafana k6 dijalankan di dalam Docker container pada mesin penguji untuk mensimulasikan lalu lintas permintaan (*HTTP traffic*) dari sisi klien. Kontainer k6 ini dikonfigurasi untuk mengirimkan permintaan ke aplikasi *backend* yang berjalan pada server MNP. Pendekatan kontainerisasi digunakan untuk menjaga konsistensi lingkungan pengujian, mempermudah replikasi skenario uji, serta meminimalkan pengaruh konfigurasi sistem operasi utama pada mesin penguji. Dengan pemisahan ini, pengujian performa merepresentasikan kondisi *end-to-end*, sehingga hasil pengujian mencakup waktu respons sistem secara keseluruhan, termasuk latensi jaringan.

3.2.2 Persiapan Data Uji

Data uji yang digunakan dalam penelitian ini merupakan data dummy yang dibuat secara sintetis menggunakan *library* Faker.js. Data ini mencakup atribut pengguna lengkap seperti ID, nama, dan username yang kemudian diekspor ke dalam format CSV dan diinjeksikan ke dalam database sebelum pengujian dimulai. Pendekatan berbasis CSV ini dipilih untuk menjamin konsistensi lingkungan pengujian serta memudahkan pengaturan volume data secara presisi, seperti simulasi 1.000 ,3000, dan 5.000 pengguna, agar proses pengujian dapat dilakukan secara terkontrol tanpa bergantung pada data operasional sistem yang asli.

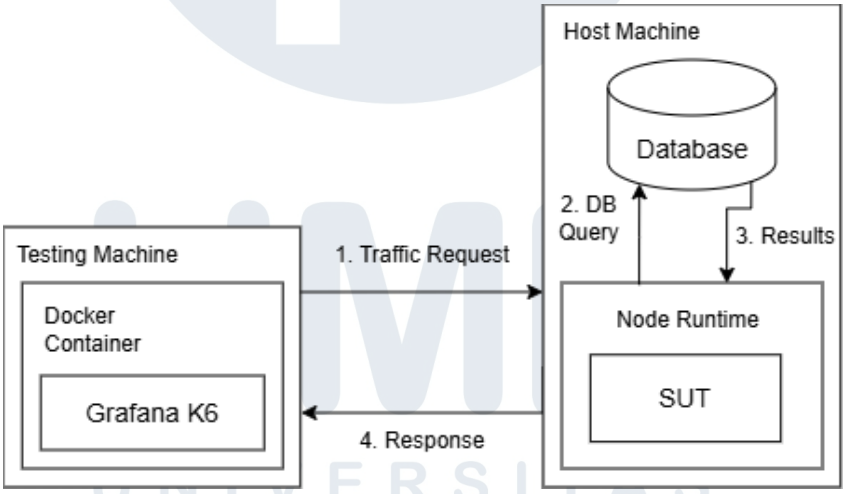
Khusus untuk skenario pengujian *fault tolerance*, disusun sebuah dataset pengujian yang terdiri dari berbagai kueri dan entitas target sebagai referensi evaluasi. Dataset ini mencakup sepuluh kesalahan pengetikan (*typo*) tunggal dan campuran yang dirancang secara sistematis seperti yang sudah dijelaskan oleh studi klasik Damerau. Setiap kueri *typo* dipetakan secara manual ke entitas pengguna target yang diharapkan masih dapat ditemukan oleh sistem. Pemetaan ini digunakan sebagai referensi untuk mengevaluasi apakah entitas target masih muncul dalam daftar hasil pencarian pada batas jumlah kandidat tertentu (*Top-10*).

3.2.3 Pengujian Performa

Pengujian performa memiliki dua tujuan spesifik mengenai *load time*. Tujuan pertama untuk mengukur dan membandingkan waktu muat antara arsitektur baru yang diusulkan dengan implementasi eksisting secara kuantitatif. Melalui pengujian ini, diperoleh data mengenai waktu pemrosesan penyajian data pada kedua arsitektur saat menangani berbagai tingkat beban. Data yang terkumpul akan digunakan untuk menghitung rata-rata waktu pemrosesan (*average processing time*) serta melakukan analisis performa. Hasil analisis tersebut kemudian dimanfaatkan untuk memvalidasi efektivitas arsitektur baru dalam meningkatkan kecepatan penyajian data dibandingkan arsitektur eksisting.

Tujuan kedua difokuskan untuk mengukur dan membandingkan waktu komputasi (*computation time*) pencarian terhadap data yang tidak tersedia (*non-existing elements*) antara sistem yang menggunakan Bloom Filter dan tanpa Bloom Filter. Melalui pengujian ini, diperoleh data selisih waktu proses yang menunjukkan seberapa cepat sistem mampu menolak kueri kosong. Hasil analisis tersebut digunakan untuk membuktikan efektivitas mekanisme *early rejection* pada Bloom Filter dalam memangkas jalur eksekusi program, yang berdampak langsung pada percepatan dan penghematan sumber daya komputasi.

Untuk mensimulasikan interaksi *client-server* secara realistis, penelitian ini mengadopsi pendekatan kontainerisasi berbasis Docker. Dalam skema ini, *testing machine* menjalankan Grafana k6 yang berlaku sebagai klien dieksekusi di dalam lingkungan kontainer yang terisolasi untuk membuat lalu lintas permintaan (*traffic request*) menuju SUT (*System Under Testing*) yang berjalan di *host machine*.



Gambar 3.15 Skema pengujian performa

3.2.3.1 Skenario Pengujian Performa

Skenario pengujian performa diklasifikasikan menjadi dua skenario pengujian. Pertama, skenario pengujian performa penyajian data dirancang untuk mengukur dampak pertumbuhan volume data terhadap waktu respons sistem dalam proses penyajian data pengguna. Variabel yang dimanipulasi pada

skenario ini adalah jumlah data pengguna yang disimpan di dalam database. Tujuan dari pengujian ini adalah untuk membuktikan hipotesis bahwa mekanisme penyajian data pada arsitektur eksisting mengalami peningkatan waktu pemrosesan secara signifikan seiring bertambahnya jumlah data, serta memvalidasi peningkatan kecepatan penyajian data melalui penerapan mekanisme *pagination* pada arsitektur baru yang diusulkan.

Skenario pengujian dijalankan menggunakan konfigurasi 1 *Virtual User* (VU) untuk menjaga konsistensi baseline dan mengisolasi pengaruh pertumbuhan data tanpa interferensi beban *concurrency*. Permintaan (*request*) dieksekusi selama durasi 10 detik untuk memperoleh hasil rata-rata.

- Tahap 1: 1000 data pengguna
- Tahap 2: 3000 data pengguna
- Tahap 3: 5000 data pengguna

Pembagian tahap ini memungkinkan analisis bertingkat terhadap performa sistem, sehingga peningkatan *load time* dapat diamati secara gradual dari volume kecil hingga besar.

Kedua, skenario pengujian performa Bloom Filter dirancang untuk mengevaluasi efektivitas *mekanisme early rejection* dalam mempercepat proses pencarian ketika kueri tidak terdapat di dalam database (*non-existing elements*). Pada skenario ini, pengujian dilakukan dengan mengirimkan kueri yang dipastikan tidak ada dalam database, kemudian mencatat waktu komputasi yang dihasilkan oleh sistem tanpa Bloom Filter dibandingkan dengan sistem yang menggunakan Bloom Filter. Indikator keberhasilan skenario ini adalah jika penggunaan Bloom Filter mampu menurunkan waktu komputasi secara signifikan dan konsisten, ketika menangani kueri yang tidak memiliki kecocokan pada database.

3.2.4 Pengujian Fault Tolerance

Pengujian *fault tolerance* bertujuan untuk mengevaluasi dan membandingkan kemampuan pencarian eksisting (*substring matching*), pencarian *similarity Winnowing* serta kombinasi *Trigram* dan Jaro-Winkler dalam menangani variasi ejaan dan kesalahan ejaan (*typo*). Pengujian dilakukan menggunakan dataset kueri yang disusun secara sistematis dengan beberapa tingkat kesalahan input. Setiap kueri dieksekusi pada sistem pencarian untuk memperoleh daftar kandidat hasil pencarian. Evaluasi dilakukan dengan mengamati apakah entitas target masih muncul pada hasil pencarian dalam batas jumlah kandidat tertentu (*Top-10*). Tingkat toleransi (*tolerance level*) kesalahan ditentukan berdasarkan level kesalahan maksimum yang masih menghasilkan kandidat yang relevan.

3.2.4.1 Skenario Pengujian Fault tolerance

Skenario pengujian *fault tolerance* dilakukan dengan mengeksekusi seluruh input pencarian yang telah disusun dalam dataset pengujian pada masing-masing pendekatan pencarian, yaitu pencarian *substring* pada sistem eksisting, metode *Winnowing*, serta kombinasi *Trigram* Jaro-Winkler. Setiap kueri dieksekusi untuk menghasilkan daftar kandidat hasil pencarian dengan batas jumlah tertentu (*Top-10*).

Evaluasi dilakukan dengan mengamati apakah entitas target yang telah ditetapkan sebagai referensi (*ground truth*) masih muncul dalam daftar hasil pencarian (*Top-10*) pada setiap variasi kesalahan input. Fokus evaluasi meliputi kemampuan sistem dalam mempertahankan kemunculan entitas target serta perubahan posisi peringkat entitas tersebut di dalam daftar hasil pencarian, meskipun input pencarian mengalami variasi ejaan atau kesalahan pengetikan (*typo*).

Tingkat *fault tolerance* kemudian ditentukan berdasarkan *tolerance level*, yaitu tingkat kesalahan input maksimum yang masih dapat ditoleransi oleh masing-masing pendekatan pencarian. Hasil pengujian dianalisis secara komparatif dan disajikan dalam bentuk grafik untuk menggambarkan perbedaan kemampuan toleransi kesalahan antar pendekatan pencarian.

3.2.5 Metrik Pengujian dan Evaluasi

Metrik pengujian digunakan untuk mengevaluasi performa arsitektur baru yang diusulkan dibandingkan dengan implementasi eksisting, serta untuk menilai tingkat toleransi pencarian eksisting dan pencarian *similarity*, yaitu *Winnowing* serta kombinasi *Trigram* dan Jaro-Winkler. Metrik dibagi ke dalam dua kategori utama, yaitu Metrik pengujian performa dan metrik pengujian *fault tolerance*.

3.2.5.1 Metrik Pengujian Performa

Kategori ini berfokus pada pengukuran efisiensi waktu dan penggunaan sumber daya. Metrik ini digunakan untuk menjawab pertanyaan penelitian terkait optimasi penyajian data dan kecepatan mekanisme *early rejection*.

1. *Load Time*

   Metrik ini mengukur total waktu yang dibutuhkan sistem untuk memproses satu siklus permintaan, mulai dari permintaan dikirim oleh klien hingga seluruh data diterima. *Load Time* diukur dalam satuan milidetik atau *millisecond* (ms), dan digunakan untuk mengevaluasi dampak penerapan *pagination*. Penurunan nilai *load time* yang signifikan pada arsitektur usulan dibandingkan sistem eksisting menjadi indikator utama keberhasilan implementasi arsitektur baru.

2. *Computation Time*

Metrik ini mengukur total waktu komputasi yang dibutuhkan sistem untuk memproses pencarian, dari diterima oleh server sampai server ingin mengembalikan response. *Computation time* diukur dalam satuan milidetik atau *millisecond* (ms), dan digunakan untuk mengevaluasi dampak penerapan Bloom Filter. Penurunan nilai *Computation time* yang signifikan pada arsitektur usulan dibandingkan sistem eksisting menjadi indikator utama keberhasilan implementasi mekanisme *early rejection* Bloom Filter.

3.2.5.2 Metrik Pengujian Fault Tolerance

Kategori ini difokuskan untuk mengevaluasi kinerja pencarian eksisting, pencarian *similarity Winnowing* serta kombinasi *Trigram* dan Jaro-Winkler dalam menangani variasi input dan kesalahan ejaan (*typo*). Evaluasi dilakukan dengan membandingkan hasil pencarian terhadap data referensi (*ground truth*) yang telah ditetapkan sebelumnya. Berbeda dengan pengujian berbasis akurasi klasifikasi, pengujian *fault tolerance* tidak menilai kebenaran hasil pencarian secara biner (0 atau 1), melainkan mengamati sejauh mana sistem masih mampu menoleransi kesalahan input dengan tetap menampilkan entitas target pada daftar hasil pencarian.

Hasil pencarian dikategorikan berdasarkan kemunculan dan posisi entitas target dalam daftar kandidat hasil pencarian (*Top-10*), dengan definisi sebagai berikut::

- *Target Found* (TF): Entitas target muncul dalam daftar hasil pencarian *Top-10*.

- *Target Not Found* (TNF): Entitas target tidak muncul dalam daftar hasil pencarian *Top-10*.

Selain itu, posisi peringkat (*ranking position*) dari entitas target juga dicatat untuk mengevaluasi stabilitas peringkat akibat variasi kesalahan input.

Berdasarkan kategori tersebut, evaluasi *fault tolerance* dilakukan menggunakan metrik berikut:

1. *Tolerance Level*

   Mengukur tingkat kemampuan sistem pencarian dalam menoleransi variasi ejaan dan kesalahan pengetikan (*typo*) dengan menghitung persentase kueri di mana entitas target masih berhasil muncul dalam daftar *Top-10* hasil pencarian.

$$Tolerance\ Level\ = \frac{Jumlah\ Kueri\ dengan\ TF}{Total\ Kueri\ Pengujian} x100\%$$