Exploiting Code Generation for Efficient LIKE Pattern Matching

Adrian Riedl, Philipp Fent, Maximilian Bandle, Thomas Neumann
{riedla,fent,bandle,neumann}@in.tum.de
Technical University of Munich

ABSTRACT

Efficiently evaluating text pattern matching is one of the most common computationally expensive tasks in data processing pipelines. Especially when dealing with text-heavy real-world data, evaluating even simple LIKE predicates is costly. Despite the abundance of text and the frequency of string-handling expressions in real-world queries, processing is an afterthought for most systems. We argue that we must instead properly integrate text processing into the flow of DBMS query execution. In this work, we propose a code generation approach that specifically tailors the generated code to the given pattern and matching algorithm and integrates cleanly into DBMS query compilation. In addition, we introduce a generalized SSE search algorithm that uses a sequence of SSE instructions to compare packed strings in the generated code to efficiently locate longer input patterns. Our approach of generating specialized code for each pattern eliminates the overhead of interpreting the pattern for each tuple. As a result, we improve the performance of LIKE pattern matching by up to 2.5×, demonstrating that code generation can significantly improve the efficiency of LIKE predicate evaluation in DBMSs.

PVLDB Reference Format:
Adrian Riedl, Philipp Fent, Maximilian Bandle, Thomas Neumann. Exploiting Code Generation for Efficient LIKE Pattern Matching. PVLDB, 17(1): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Modern data processing systems offer outstanding performance on simple data, which makes them an essential component for efficient data processing pipelines. However, these systems are still lacking in compute-heavy string processing, which is common in real-world applications. Tableau's research shows that approximately 50% of all attributes use text-based data types, even when there are more suitable data types [26]. Thus, database systems need to focus on efficient text operations such as LIKE expressions.

In current systems, a common technique to process text is to use a third-party library that focuses on matching string patterns, often offering advanced features such as SIMD acceleration [23, 28]. Unfortunately, these do not integrate well with DBMS-specific text representation. For example, DBMS commonly use special string storage formats, e.g., with parts of the string inlined or lightweight compressed [3, 8]. However, to use external text libraries, these systems need expensive string conversions before they can invoke a matching function, while a better integrated approach could allow just-in-time decompression. In addition, string search libraries optimize for finding patterns in a large continuous text corpus, e.g., a text document. In contrast, for database systems, the per-tuple overhead to interpret the string pattern or transition tables leaves significant performance on the table, which a better integrated approach can overcome.

Proceedings of the VLDB Endowment, Vol. 17, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

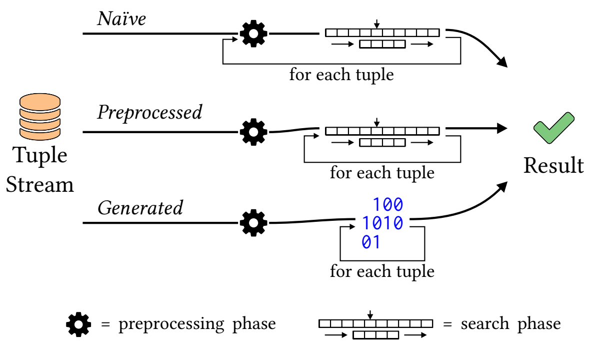


Figure 1: Different *options* to integrate pattern-matching algorithms to evaluate LIKE expressions in DBMS.

Figure 1 illustrates the different *options*, how databases can process the tuples. The two traditional options are:

Naïve: A generic function performs the matching process and is called once per tuple during query execution. For each tuple, it preprocesses the pattern again and executes the search phase.

Preprocessed: The pattern is preprocessed only once before the first search starts, and the result of this preprocessing (e.g., a transition table) is stored. For each tuple, the database engine still calls a generic pattern-matching function, but this function reuses the stored information in the search.

In contrast to these approaches, we need to integrate LIKE pattern matching deeper into the data processing engine. Code generation allows specializing the matching function by inlining the patterns and shift tables. We can also inline the generated code in a larger processing kernel, e.g., using data-centric code generation [19], or in just-in-time compiled vectorized functions. In this work, we integrate common algorithms such as Knuth-Morris-Pratt [13], Boyer-Moore [4, 10, 22], Two-Way [5], and SIMD optimized routines [24]:

Generated: During query compilation, we preprocess the pattern once. Then, we generate code for the entire search process using as much preprocessed results as possible. The entire matching process is performed in the generated code to avoid repeating function calls.

In this paper, we focus on how to generate code for the most common subset of regular expressions in SQL: LIKE expressions.

Section 3 introduces the string-matching algorithms and outlines how our *Generated* approach utilizes a code generation framework to rebuild each matching function and specialize the code for the given pattern. Furthermore, we present a generalized SSE Search algorithm that generates highly efficient code for longer patterns by utilizing SSE instructions. In Section 4, we evaluate these algorithms alongside the mentioned options in our code-generating DBMS Umbra [20]. This provides a fair analysis of the different *options* to evaluate LIKE predicates within a single system. While patterns are typically short, we also analyze the performance with longer patterns, reaching up to nearly 300 characters. Additionally, we compare our *Generated* approach with the publicly available systems Postgres, DuckDB, Hyper, and ClickHouse.

2 RELATED WORK

Kemper and Neumann introduced Hyper, a pure in-memory database for both OLTP and OLAP workloads [11], with a data-centric approach for generating and compiling compact and efficient machine code using LLVM [19].

The idea of code generation has become widely accepted among database researchers and developers [2, 6, 9, 27]. Umbra [20], the research successor of the HyPer system, introduced a type-safe code generation framework with a tailored intermediate representation to compile directly into machine code, which makes Umbra applicable for low latency applications [12]. Together with adaptive execution, this allows us to switch between prioritizing low-latency or high throughput in execution [14].

In general, regular expression matching can benefit from using just-in-time code generation. Thompson introduced the concept in 1968 to produce an IBM 7094 programs for locating characters by regular expressions [25]. Today, many state-of-the-art libraries use code generation to convert the given regular expression pattern into an internal representation of bytecode: Python's `re` module generates bytecode for regular expressions and uses an internal C engine for efficient execution [15]; Google's `re2` represents the regular expression as an automaton in bytecode and passes it for interpretation to an execution engine [29]; Microsoft's `.NET` framework provides both a bytecode interpreter and a just-in-time compiler that converts the expression to native machine code [1].

However, only a few projects aim to generate machine code for regular expression matching at runtime, and no database system actively combines code generation and pattern matching.

3 IMPLEMENTATION

This section outlines how pattern-matching algorithms can use parts of code-generation frameworks to be integrated into compiling database engines. The concept of code generation for queries, as used in HyPer [11, 19] or Umbra [20], presents a novel opportunity for evaluating LIKE expressions in relational database systems by generating code for the matching process instead of interpreting the pattern. We start with *Naïve*, which uses a hand-written function for pattern matching. It is called for every tuple in the generated code by the database system. We can already improve its performance by preprocessing the pattern passed to the function.

However, we aim to entirely replace the function by generating code specifically for the pattern and algorithm. The generated code is then directly embedded in the surrounding generated function code. Our current focus is on constant LIKE patterns without any underscores or collations. Thus, a bytewise comparison between the pattern and input text is feasible, allowing us also to handle non-ASCII characters. Within Umbra, we are using our type-safe code-generation framework, which allows us to pass the generated code in static single assignment (*SSA*) form on to Umbra's different execution backends [12].

Throughout this chapter, let us consider the following query that filters the `uni` relation and counts how many `names` contain the relatively short pattern `'TUM'`:

```
select count(*) from uni where name like '%TUM%';
```

In the upcoming Section 3.1 and Section 3.2, we discuss the Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) algorithms and explain in detail how they integrate into the code generation process. Section 3.3 briefly introduces the Two-Way (TW) search algorithm that combines KMP and BM. Moving away from the well-known pattern matching algorithms, Section 3.4 presents the Hybrid-Search (HS) algorithm which uses an SSE instruction to match patterns up to a certain size. In Section 3.5, we show how blockwise processing can improve performance in finding possible occurrences of the pattern. Finally, Section 3.6 presents the SSE Search algorithm which uses SSE instructions to perform pattern matching expressions for longer patterns. It is important to note that this algorithm can only be implemented effectively in a code-generating database engine. This is due to the diverse nature of input patterns, where the flexibility offered by a code-generating process surpasses that of an interpreting algorithm.

3.1 Knuth-Morris-Pratt Algorithm

In 1977, Knuth, Morris, and Pratt introduced an algorithm for performing exact string pattern matching without the need to backtrack in the input text by preprocessing the pattern [13]. The algorithm builds a table with pattern length + 1 entries that point to where, in the pattern, we continue the search after a mismatch. Thus, the table holds information about the longest proper prefix that is also a proper suffix of the pattern (*lps*). For the prefix of length 0, the table stores the value −1, indicating that if a mismatch occurs, the pattern can be shifted one position to the right since no suffix exists at that point.

```
1  KMP(text, pattern):
2     lpsTable = preprocess(pattern), pPos = 0, tPos = 0;
3     while (tPos - pPos + pattern.size() <= text.size())
4        if (pattern[pPos] == text[tPos])
5           pPos++; tPos++;
6           if (pPos == pattern.size()) return true;
7        else
8           shift = lpsTable[pPos];
9           if (shift < 0) pPos = 0; tPos++;
10          else pPos = shift;
11    return false;
```

Listing 1: Pseudocode for the Knuth-Morris-Pratt algorithm

Listing 1 presents the pseudocode of the KMP algorithm. Line 2 preprocesses the pattern and initializes two position counters for text and pattern. When the characters at these indexes match (lines 4

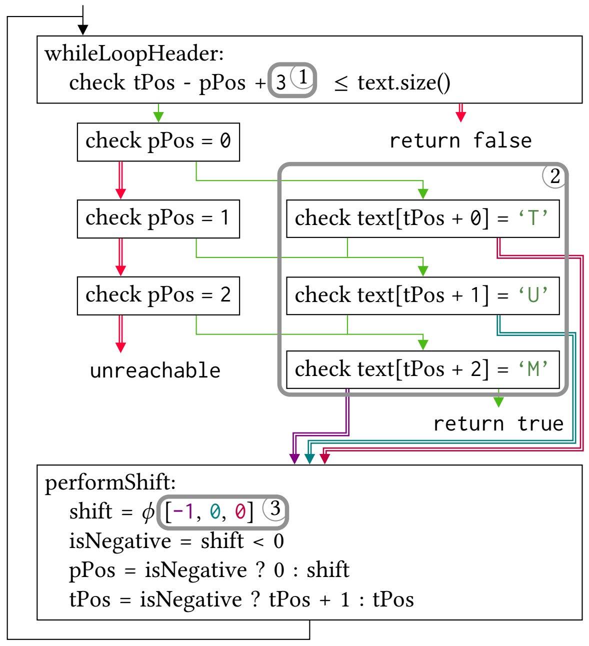

whileLoopHeader:
check tPos - pPos + 3 1 ≤ text.size()
check pPos = 0
return false
check pPos = 1
check pPos = 2
unreachable
2
check text[tPos + 0] = 'T'
check text[tPos + 1] = 'U'
check text[tPos + 2] = 'M'
return true
performShift:
shift = ϕ [-1, 0, 0] 3
isNegative = shift < 0
pPos = isNegative ? 0 : shift
tPos = isNegative ? tPos + 1 : tPos

Figure 2: Control flow of the generated code for the KMP algorithm to search for the pattern 'TUM'. The green arrows (⟶) are taken if the condition evaluated to true, the red or otherwise colored arrows (⟹) if not.

to 6), the function increments both position counters. When reaching the pattern end, a match is found.

If the characters do not match (lines 7 to 10), the function reads the optimal shift value from the lps table based on the pattern position. A negative value indicates that there is no proper suffix in the pattern that is also a prefix of the pattern. Thus, we need to restart the comparison from the following text character. Otherwise, the function updates the pattern position to the lps table value and increments the text index.

In line 3 of Listing 1, we introduce an optimization for the KMP algorithm called the *early return*, which we use in all our variants. This optimization checks every iteration to determine whether the end of the pattern lies within the text. If that is not the case, we will stop the comparison, as it is impossible to find another match.

3.1.1 Preprocessed approach. To prevent the pattern from being processed repeatedly for each tuple, we preprocess the pattern during code generation time to get the lps table and then save this table along with the pattern in the generated program. We do not need to store any additional information for the lps table because its size depends on the pattern size.

When executing the generated code, we still make a function call to perform the search, but instead of recalculating the table every time, we reuse the stored lps table. So, we omit the preprocessing in line 2 of Listing 1.

3.1.2 Generated approach. We replace all calls to the matching function with specifically generated code for the query's pattern. So, we embed the search phase algorithm entirely within the generated code. Figure 2 shows how we can reconstruct the Knuth-Morris-Pratt algorithm using the example we discussed earlier to search for the pattern 'TUM' in the input text.

We begin the algorithm in the `whileLoopHeader` block by checking if the remaining length of the input text is sufficient. In this check, we can inline the size of the pattern into the arithmetic expression (1). Then, we move to the correct pattern position to continue the comparisons from this position. We generate the comparisons out of the pattern from left to right (2). If the characters match, we proceed directly to the next character, but if there is a mismatch, we jump to the `performShift` block. In this block, we choose the shift value from the inlined lps table (3) based on the position of the failed comparison. We then determine how to proceed and continue the search by jumping back to the `whileLoopHeader`.

3.2 Boyer-Moore Algorithm

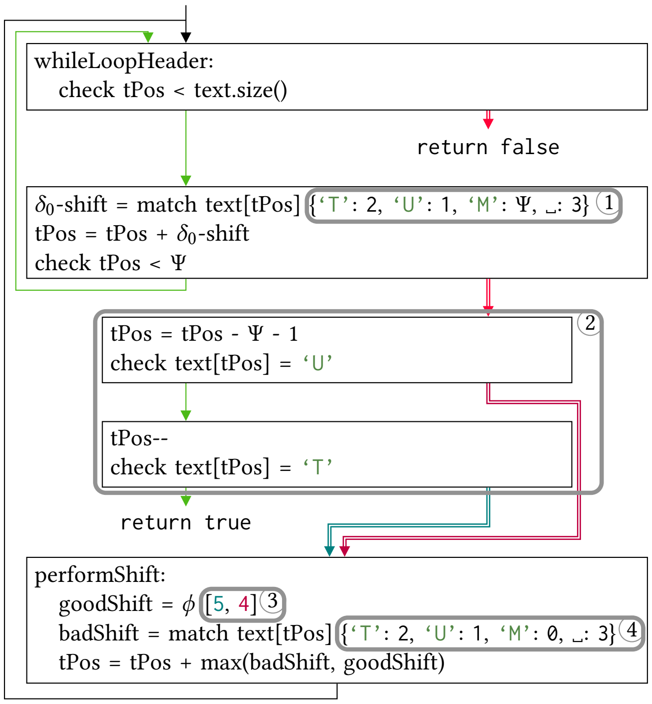
In 1977, Boyer and Moore presented a pattern-matching algorithm that iterates backwards over the pattern to search it in the input text [4]. We focus on their fast implementation shown in Listing 2.

```
1  BM(text, pattern):
2     // $\Psi$ > text.size() + pattern.size() for all inputs
3     $\Psi$ = 1 << 48, pPos = pattern.size() - 1, tPos = pPos;
4     $\delta_1$ = preprocessBadCharacterHeuristic(pattern);
5     $\delta_2$ = preprocessGoodSuffixHeuristic(pattern);
6     $\delta_0$ = $\delta_1$;
7     $\delta_0$[pattern[pattern.size() - 1]] = $\Psi$;
8     while (tPos < text.size())
9        tPos += $\delta_0$[begin[tPos]];
10       if (tPos >= $\Psi$)
11          tPos = tPos - $\Psi$ - 1;
12          if (pattern.size() == 1) return true;
13          else
14             pPos = pattern.size() - 2;
15             while (pPos && begin[tPos] == pattern[pPos])
16                pPos--; tPos--;
17             if (!pPos && begin[tPos] == pattern[pPos])
18                return true;
19             tPos += max($\delta_1$[begin[tPos]], $\delta_2$[pPos]);
20    return false;
```

Listing 2: Pseudocode for the Boyer-Moore algorithm

Before the search phase begins, the algorithm requires two preprocessing steps, namely Bad Character Heuristic (BCH) in line 4 and Good Suffix Heuristic (GSH) in line 5. The BCH ensures that the text's letter at which the mismatch occurred aligns with its rightmost occurrence in the pattern. Alternatively, the GSH shifts the pattern based on the longest suffix of the matched input text. This part is aligned with the rightmost occurrence of that character sequence in the pattern (except for the suffix of the pattern itself). Both heuristics precalculate shift values for the pattern and store them in tables. The original papers contain a more detailed explanation of the heuristics and their effects [4, 22].

The fast implementation requires a third table, δ_0, which is essentially a copy of the result of BCH but holds the value Ψ (called *large* in [4]) for the last character of the pattern. This value needs to be greater than the sum of the lengths of all possible input texts



```
whileLoopHeader:
    check tPos < text.size()

                                            return false

δ0-shift = match text[tPos] {'T': 2, 'U': 1, 'M': Ψ, ␣: 3} (1)
tPos = tPos + δ0-shift
check tPos < Ψ

    tPos = tPos - Ψ - 1                                   (2)
    check text[tPos] = 'U'

    tPos--
    check text[tPos] = 'T'

        return true

performShift:
    goodShift = ϕ [5, 4] (3)
    badShift = match text[tPos] {'T': 2, 'U': 1, 'M': 0, ␣: 3} (4)
    tPos = tPos + max(badShift, goodShift)
```

Figure 3: Control flow of the generated code for the BM algorithm to search for the pattern `'TUM'`. The green arrows ($\longrightarrow$) are taken if the condition evaluated to true, the red or otherwise colored arrows ($\Longrightarrow$) if not.

and patterns. At the beginning of the search phase, the pattern is aligned with the start of the input text and the comparison starts from the rightmost character. The implementation looks up the value in δ_0 and either shifts the pattern to the right or adds Ψ to the current position (line 9 of Listing 2). This allows us to scan through the input text, and once we add Ψ to the current position, we know the last character was found. Thus, the algorithm recalculates the index for the second last character and starts comparing the pattern from right to left with the input text (lines 14 to 18). In case of a mismatch during this comparison, the algorithm applies the maximum of the shifts according to the heuristics (line 19) before the search for the last character of the pattern restarts.

3.2.1 Preprocessed approach. As the BCH table contains 256 values and the size of the GSH table matches the pattern length, repetitive processing is quite expensive. Like the KMP algorithm, we move all preprocessing steps to code generation and store the resulting tables directly along with their corresponding pattern. When storing the tables, we do not require additional information since the size of the tables is either known beforehand or can be derived from the pattern size. We replace the calls to the preprocessing functions (lines 4 to 5) with pointers to the corresponding table. As the only difference between δ_0 and δ_1 is the value for the pattern's last character, we do not copy the table but instead, modify the code in the search phase loop to actively add the correct value, so either the value Ψ or the value from δ_1.

3.2.2 Generated approach. Similar to the KMP algorithm, we rebuild the Boyer-Moore algorithm to get pattern-specific code. Figure 3 presents the conceptual control flow for searching the pattern `'TUM'` in the input text. We check whether enough characters are left in the input string before we start the matching process in the `whileLoopHeader` block. If so, we get the shift value from the inlined δ_0 table (1) and add it to the pattern position. If that value is smaller than Ψ, we continue in this loop by going to the `whileLoopHeader` block. Otherwise, we know the input text contains the last character of the pattern, so further checks are required. Before the checks, we recalculate the character index aligned with the second last character in the pattern. In order to proceed, we check the text based on the reversed pattern (2). In case of a mismatch, we jump to the `performShift` block. In this block, we determine `goodShift` from δ_2 (3) based on the preceding block and the `badShift` from δ_1 (4) based on the mismatching text character. We then add the maximum of both shift values to the text position before returning to the `whileLoopHeader` to continue with the matching process.

When analyzing the instructions in the `performShift` block of Figure 3, one may mistakenly perceive the inclusion of the inlined δ_1 table for determining the shift caused by the BCH as unnecessary. This impression arises from the fact that the minimum shift resulting from the GSH is always greater than the maximum possible shift caused by the BCH. Consequently, the maximum shift is consistently determined by the good suffix heuristics. It is important to note, however, that this observation cannot be universally applied to all patterns. To address this, we have implemented an optimization in the code generation process, which generates code for determining the BCH shift only when it is truly required.

3.3 Two-Way Algorithm

As an alternative to the Knuth-Morris-Pratt and Boyer-Moore algorithms, Crochemore and Perrin presented the Two-Way String-Matching algorithm (TW) which combines both previous algorithms into one [5]. To achieve this, the algorithm first splits the pattern according to the known Critical Factorization Theorem [18]. With that, the pattern is theoretically split into a left and right part. In the search phase, the right half is compared from left to right first; if all characters match, then the left half is compared from right to left. If any mismatches during the comparisons occur, the pattern is shifted by a certain number of positions. After a close analysis of the interpreting Two-Way algorithm, its functionality can be rebuilt using the available code generation framework as for the other algorithms.

Again, we implement a *Naïve*, *Preprocessed*, and *Generated* version for the Two-Way algorithm. In the *Naïve* version, the Critical Factorization preprocessing step is performed repeatedly for each input text. In the *Preprocessed* version, we store the necessary preprocessed result along with the pattern in the data section of the generated code. This value is then loaded along with the pattern when needed and used in an interpretive algorithm. The *Generated* version of the algorithm depends on the output of the preprocessing function. It generates the relevant part of the Two-Way algorithm based on the outcome of the Critical Factorization step and inlines as much of the information as possible into the algorithm.

3.4 Hybrid-Search Algorithm

Sitaridi et al. have introduced an algorithm which uses the SSE 4.2 SIMD instruction set, which comprises instructions that efficiently accelerate string and text processing [24]. According to Intel, these instructions are designed to enhance the performance of databases or complex searching and pattern matching algorithms [21]. However, the presented algorithm is restricted to patterns to fit into a 128-bit SIMD register.

```
1  HS(text, pattern):
2     if (pattern.size() <= 16 && text.size() >= 16)
3        iter = text.begin(), end = text.end();
4        safeMatch = 17 - pattern.size();
5        pattern16 = load16(pattern);
6        while ((iter + 16) < end)
7           match = pcmpistri(pattern16, load16(iter);
8           if (match < safeMatch) return true;
9           iter += safeMatch;
10       if (iter < end)
11          match = pcmpistri(pattern16, load16(end - 16);
12          return match < safeMatch;
13       return false;
14    return TW(text, pattern)
```

Listing 3: Pseudocode for the Hybrid Search algorithm

With our Hybrid Search, we extend this algorithm to handle any size of input text and pattern, as presented in Listing 3: For patterns up to the length of a vector register, we use the `pcmpistri` instruction, given that the input text is at least 16 bytes long. In such cases, we process 16 bytes of the input text at once until less than 16 bytes are left (lines 6 to 9). To check the end of the input text, we load the last 16 bytes of the input text, which is safe since we know that the input text is long enough (lines 10 to 12). However, if either of the input parameters does not meet its length criterion, we resort to a default string search algorithm. In our case, it is the Two-Way algorithm (line 14). Considering that the best-suited algorithm depends on various factors, such as the pattern and workload, it would be beneficial to implement multiple fallback algorithms, allowing the selection of the most appropriate one.

3.4.1 Preprocessed approach. Based on the chosen fallback algorithm, one might consider how to include the corresponding *Preprocessed* function of the chosen algorithm. To match the chosen Two-Way algorithm as default fallback algorithm in the *Naïve* approach, we chose the *Preprocessed* version for this approach.

3.4.2 Generated approach. To generate code for our Hybrid Search, we extended the custom code generation framework of Umbra to support the necessary SSE instruction for comparing packed strings. This enables us to generate the parts of the algorithm needed for the specific pattern. For patterns that are longer than 12 bytes, we only generate the code for the default matching algorithm, as we do not use the SSE instruction for that kind of patterns. For shorter patterns, we generate both the part using the SSE instruction and the default fallback. While executing the code, we determine which part of the algorithm to use based on the length of the input text. The decision to set the limit to 12 bytes is guided by the fact that this still allows performant shifting of the input pattern (cf. `safeMatch`, measured in Figure 9 in Section 4.3.1).

3.5 Blockwise Processing Optimization

Blockwise Processing can improve the initial pattern search. It draws inspiration from SIMD within a register (SWAR) [16] and enhances the efficiency of character search in an input text over the naïve idea. This would involve iterating over the text and examining each character resulting in a wastage of cycles simply searching for the desired character. However, blockwise processing can be implemented to rapidly locate the first character of the pattern and then continue with the chosen pattern-matching algorithm. Listing 4 demonstrates the algorithm to detect the presence of the ASCII character `'T'` in the `block`. We read the next eight bytes from the input text into a register. With another register having the character broadcasted to each byte, we perform various bitwise operations between the registers and specific constants. After these operations, we get a value back which is either 0, so `'T'` could not be found in block, or the highest bit of the byte at which the character appeared is set. This code can also be adjusted for non-ASCII characters which have the highest bit set. While certain SSE instructions may provide similar functionality, our objective is to present a versatile approach that is not limited to any specific hardware support.

```
uint64_t block = loadNext8Bytes(...);
// broadcast 'T' to each byte: 0x5454545454545454ull
uint64_t searchedChar = broadcast('T');
const uint64_t high = 0x8080808080808080ull;
const uint64_t low = ~high;
uint64_t lowChars = (~block) & high;
uint64_t cleared = (block & low) ^ searchedChar;
uint64_t found = ~((cleared + low) & high);
uint64_t matches = found & lowChars;
bool matchFound = matches != 0;
```

Listing 4: Blockwise search for ASCII character `T`

3.6 SSE-Search Algorithm

The Hybrid Search algorithm employs an SSE instruction for pattern matching, limiting the pattern length to at most 16 bytes. For longer patterns, the algorithm provides an alternative approach that does not use the SSE instruction. Expanding the algorithm to handle longer patterns with SSE instructions is feasible but significantly increases its complexity.

However, the emergence of code-generating database engines has opened up new possibilities for generating code optimized with SSE instructions that are specifically tailored to long patterns. In Figure 4, we present the conceptual design of the generated code for searching the pattern `'Technical University of Munich'`. Similar to the KMP algorithm, we initially check if the pattern can fit within the remaining text by directly including the pattern length ①. If there are enough characters remaining, we proceed with the comparison.

The search algorithm aims to locate the starting position of the pattern within the input text. Once the start position is found,

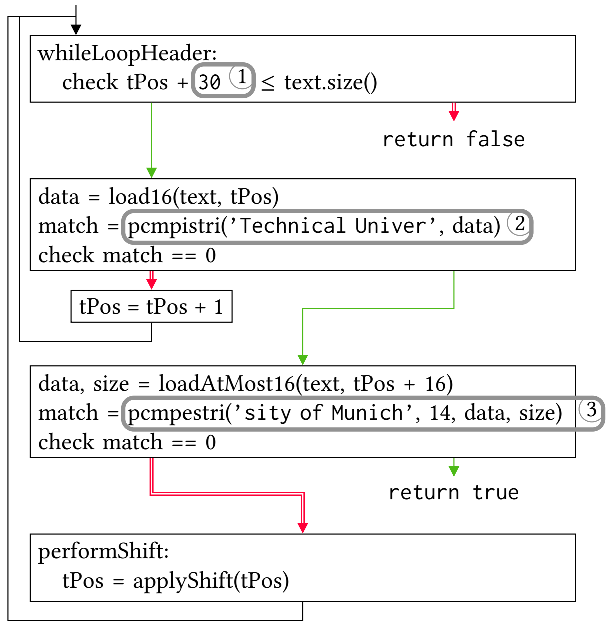

whileLoopHeader:
check tPos + 30 ① ≤ text.size()
return false
data = load16(text, tPos)
match = pcmpistri('Technical Univer', data) ②
check match == 0
tPos = tPos + 1
data, size = loadAtMost16(text, tPos + 16)
match = pcmpestri('sity of Munich', 14, data, size) ③
check match == 0
return true
performShift:
tPos = applyShift(tPos)

Figure 4: Control flow of the generated code for the SSE Search algorithm to search for the long pattern `'Technical University of Munich'`. The green arrows (⟶) are taken if the condition evaluated to true, the red or otherwise colored arrows (⟹) if not.

we continue comparing subsequent parts of the pattern and text sequentially from that position.

To achieve this, we extract the first 16 bytes of the pattern and load the next 16 bytes from the text. Using the SSE instruction `pcmpistri`, we search for the start of the pattern in the input text ②. If no match is found, we shift the text position to the right and restart the overall search for the pattern start in the input text. In the case of a match, we enter the generated code, which loads the next 16 bytes from the input text and compares them to the corresponding part of the pattern. This comparison can be performed using either the SSE instruction `pcmpistri` or another binary comparison function for vector registers. Since we know that the subsequent pattern blocks must follow the previous ones, we can easily generate code to handle this logic. This is repeated until less than 16 bytes of the pattern are left.

Handling the remaining bytes of the pattern requires special handling, as both the pattern and input text may not fully occupy an SSE register. Therefore, we load a maximum of 16 bytes from the input text and also return how many bytes were read. With the loaded block and the number of read bytes, we employ the SSE instruction `pcmpestri` ③. This instruction requires explicit specification of the length of the input data as additional arguments.

If there is a mismatch between one of the blocks of the pattern and the corresponding text block, we stop the comparison and go to the `performShift` block. Within this block, we apply a shift heuristic that moves the pattern as far to the right as feasible. Following the pattern shift, we return to the `whileLoopHeader` to resume the matching process by checking the remaining length of the text.

Shift heuristics. For shift heuristics, we have two options: a simple shift to the right by one position or a more advanced KMP-like heuristic. The latter relies on identifying the longest suffix of the already matched pattern that is also a proper prefix. This operation results in no additional runtime overhead since it can be preprocessed during code generation and is directly written to code.

Size of start block. Figure 4 presents the version of the algorithm, which directly loads the first 16 bytes of the pattern into a vector register. However, when fully utilizing the vector register, one can only shift one position to the right if the start of the pattern does not match the loaded text block. To increase the possible shift in case this part is not found in the loaded input text, we can reduce the number of bytes loaded from the pattern.

4 EVALUATION

In order to check our implementations on a more realistic dataset, we use ClickBench[1]. It includes typical modern workloads and queries used in ad-hoc analytics and real-time dashboards. The data used in the benchmark is collected from a real-world web analytics platform. While it is anonymized, it retains the essential distributions of the data, including non-ASCII characters. For our experiments, we used the queries 20, 21, 22, and 23 from the ClickBench benchmark, which contain the following LIKE predicates:

Q 20, 21, 23:	`url like '%google%'`
Q 22:	`title like '%Google%'`
	`and url not like '%.google.%'`

Query 20 scans the relation `hits` and counts how many tuples fulfill the predicate; the other queries involve more operators like aggregates or sorting, so the overall performance is not entirely dominated by the pattern matching algorithm.

Since the patterns mentioned above are shorter than the length of a vector register, we classify them as short patterns. To evaluate the effects of longer patterns, we increased the pattern length for Q 20 to 31, 160, and 291 characters, categorizing them as long patterns.

We run the microbenchmarks on an Intel i9-7900X CPU (Skylake-X, 3.3-4.5 GHz) with 10 cores and 128 GB 4-channel DDR4-2133 memory, running Ubuntu 22.10 (Kernel 5.19, gcc 12.2), and repeat all measurements five times.

4.1 Full System Comparison

We compared our *Generated* approaches for pattern matching with other popular database systems, namely Postgres, DuckDB, Hyper, and ClickHouse. Figure 5 demonstrates that our approach of generating pattern-specific code performs better than the other databases. While Umbra outperforms the other databases for Query 21 and 22, Hyper is slightly faster than our Boyer-Moore algorithm for Query 20 but slower than the other three algorithms. However, Hyper uses a pattern matching algorithm which is quite similar to our Hybrid-Search algorithm, also using an SSE instruction to search for the given pattern. As our pattern is relatively short, we observe that in nearly all cases we benefit from generating pattern specific code. Due to the shortness of the pattern, one can observe that the Hybrid-Search benefits from the SSE instruction, as it clearly dominates the other algorithms, especially for Query 20.

[1]https://benchmark.clickhouse.com

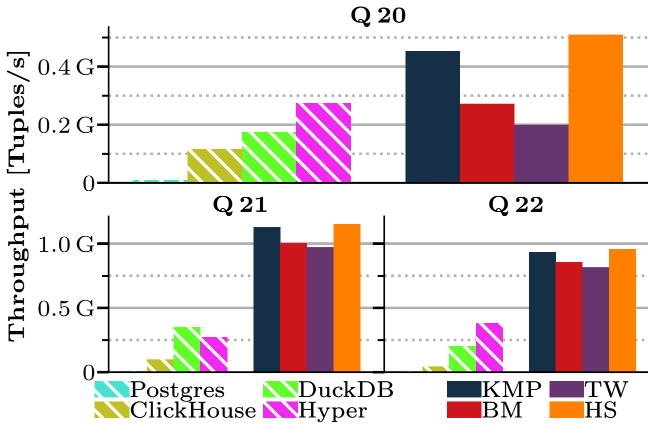


Figure 5: In a system comparison, Umbra's code generation approaches outperform the other database systems using each system's default setting for the parallelism.

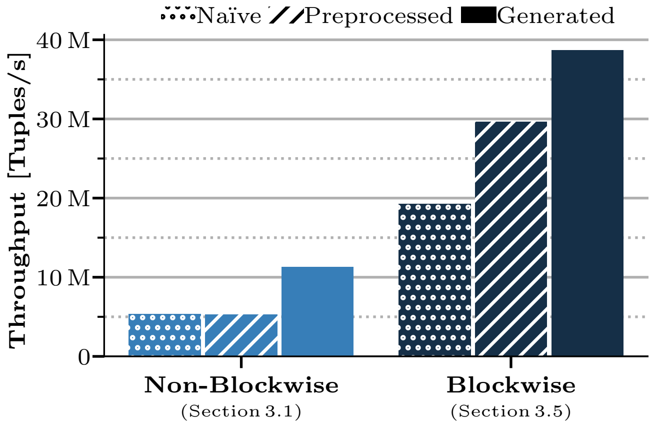


Figure 6: The KMP algorithm with Blockwise Processing outperforms the unoptimized KMP using one thread for Q 20.

4.2 Short Pattern Microbenchmark

We continue our comparison within our database system Umbra to analyze the differences between the algorithms in detail. In this section, we will focus on the more common case of short patterns.

4.2.1 Blockwise Processing. Our initial investigation evaluates the efficacy of using Blockwise Processing in combination with KMP as explained in Sections 3.1 and 3.5. If a mismatch occurs, we check how the KMP algorithm would shift the pattern according to its preprocessing. In case the pattern would be shifted by one character, we switch back to blockwise processing and restart the search for the first character of the pattern. Figure 6 illustrates the advantages of blockwise processing compared to the non-blockwise approach for Query 20 on the ClickBench dataset. By applying this optimization, larger blocks of the input text can be processed at once instead of reading byte by byte.

In the non-blockwise case, both the *Naïve* and *Preprocessed* versions show similar throughputs. After conducting a performance analysis, we identified that loading the lps table value from the data section in the *Preprocessed* approach yields performance similar to repeatedly preprocessing the relatively short pattern in the *Naïve* approach. By using the *Generated* approach, we can completely avoid any indirections, resulting in the highest throughput for the KMP algorithm.

In the blockwise algorithms, larger blocks of the input text can be skipped if the first character of the pattern is not found. Consequently, the *Preprocessed* approach is faster than the *Naïve* one since it doesn't need to access the lps table as often. Still, the *Generated* approach remains superior to the other alternatives. Based on these findings, we directly focus further analysis on the KMP algorithm with blockwise processing.

4.2.2 Algorithm Comparison. Figure 7 illustrates the results of comparing the matching algorithms discussed in Section 3 for Q 20 and Q 21 of the ClickBench benchmark. The *Preprocessed* and *Generated* approaches outperform the interpreting approaches.

This is because the preprocessing phases of the KMP and BM algorithms generate large lookup tables. By employing code generation capabilities for the *Preprocessed* approach, we can avoid redundant preprocessing of the pattern. Storing the tables in the data section of the generated program leads to a substantial improvement in throughput. As the *Generated* approach suggests, generating highly specialized code further enhances performance. However, for the Boyer-Moore algorithm, the performance improves less in Query 20. According to further analysis, this is due to many branches in the generated code, which can result in mispredictions and overall slow performance.

For Query 21, we observe similar behavior as for Query 20. However, the generated code for the Boyer-Moore algorithm appears marginally different, leading to higher performance improvement for the *Generated* approach over the *Preprocessed* version.

The preprocessing function of the Two-Way algorithm only returns a number, so it does not have to generate a table as the other two algorithms. Consequently, the *Generated* approach achieves a higher throughput compared to the *Naïve* approach.

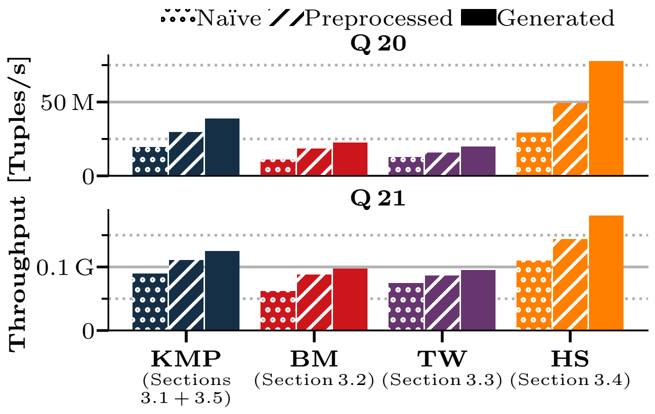


Figure 7: Single threaded throughputs for the different algorithms running Q 20 and Q 21.

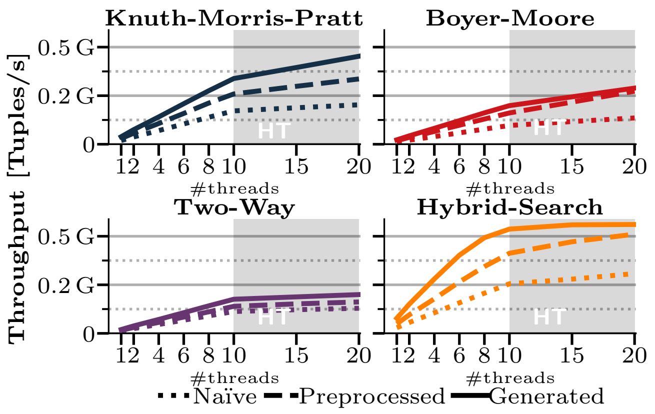


Figure 8: Development of the multi-threaded performance for the different pattern matching implementations for Q 20.

When it comes to the Hybrid-Search algorithm, one can see the benefit of using SSE instructions to search for a pattern in an input text. For each approach, this algorithm dominates all the other algorithms. Additionally, it benefits from avoiding repetitive function calls to the matching function, resulting in the throughput of the *Generated* version being nearly 2.5× the throughput of the *Naïve* approach.

4.2.3 Multithreading. The final interesting aspect of our microbenchmarks is how throughput develops when running queries using multiple threads. We expect the throughput to scale linearly with an increasing number of threads, as Umbra uses morsel-driven parallelism [17]. As shown in Figure 8, this expectation aligns with the observed results. When hyperthreading is reached, the throughput still increases, but at a different rate than before.

Our different degrees of code generation provide the most benefit to the KMP algorithm when comparing the different approaches. In the *Generated* version, we can nearly double the throughput compared to the *Naïve* approach, while the *Preprocessed* version is in the middle.

The Boyer-Moore algorithm also benefits from the code generation approaches. However, the *Preprocessed* and *Generated* versions are much closer together, and when it comes to hyperthreading, both versions are approaching each other. Nevertheless, the main problem with this algorithm is the higher number of branches required to get the correct shift from the BCH table. In the *Preprocessed* approach, this is just a memory lookup. With more branches, more mispredictions happen to cause a generic function to outperform specifically generated code.

Lastly, the *Generated* version of the Two-Way algorithm is also slightly faster than its *Naïve* version. Due to the less complexity of the preprocessing phase, the difference between both versions is small. Still, the *Generated* version has higher throughput.

Analyzing the Hybrid-Search algorithm is a bit more involved, as it depends on both the pattern and input text which specific part of the algorithm is executed. In the experiment, the pattern falls within the length limit and the input texts are on average also large enough. Therefore, the Hybrid-Search algorithm predominantly executes the search with the SSE instruction and rarely falls back to the default algorithm. This method of utilizing SSE instructions for pattern matching shows a significant performance improvement, even with the *Naïve* approach. By applying the *Generated* approach and eliminating the overhead of the repeated function calls, the throughput further increases. However, after using more than eight threads, the throughput levels out because it approaches the memory limit of the machine.

Table 1: Execution (20 threads) and compilation time ([s]) for Q 20.

	Naïve		*Preprocessed*		*Generated*	
	comp.	exec.	comp.	exec.	comp.	exec.
KMP	0.008	0.493	0.008	0.297	0.010	0.221
BM	0.008	0.740	0.008	0.366	0.010	0.346
TW	0.008	0.774	0.008	0.618	0.009	0.501
HS	0.008	0.325	0.008	0.196	0.010	0.178
SSE	-	-	-	-	0.009	0.189

Table 1 presents the execution times of the different approaches for Query 20 using 20 threads. One can observe that, with the *Generated* approach, the SSE Search algorithm is slightly slower than the Hybrid-Search algorithm. This can be attributed to the SSE Search algorithm's need for specialized handling of short patterns and input texts, resulting in more complex code and slower execution.

4.2.4 Compilation Overhead. When dealing with code-generating database engines, it is essential to consider the overhead of compilation. In Table 1, we also show the compilation times for Query 20 using the LLVM backend of Umbra. The data reveals that as we move from the *Naïve* to the *Generated* approach, the compilation times for the algorithms increase marginally. Nevertheless, this increase is balanced out by the reduction in execution time.

Moreover, it is worth mentioning that Umbra could employ the Flying Start technique or the FireARM backend. This means that the compilation overhead could be concealed by using a specific backend until the compiled LLVM function becomes available [7, 12]. However, in our experiments, we did not employ this option, as during compilation, only 0.5% of the tuples could be processed when running the Hybrid-Search algorithm fully multi-threaded.

4.3 Long Pattern Microbenchmark

As the final part of the evaluation, we look at the effect of long patterns. We classify a pattern which exceeds the length of a single vector register (16 bytes) as a long pattern. For our experiments, we use three patterns: pattern **A** with 31 characters, pattern **B** with 160 characters, and pattern **C** is a combination of three long patterns totaling 291 characters.

4.3.1 Size of start block. Figure 9 presents the results of varying the number of characters in the start block, which is employed to locate a potential pattern start. The top plots visualize the performance using only one thread, while the bottom ones show performance with 20 threads.

When executing with a single thread, the algorithm achieves peak performance when three bytes of the pattern are employed in the localization phase. This size allows for sufficient shifting of

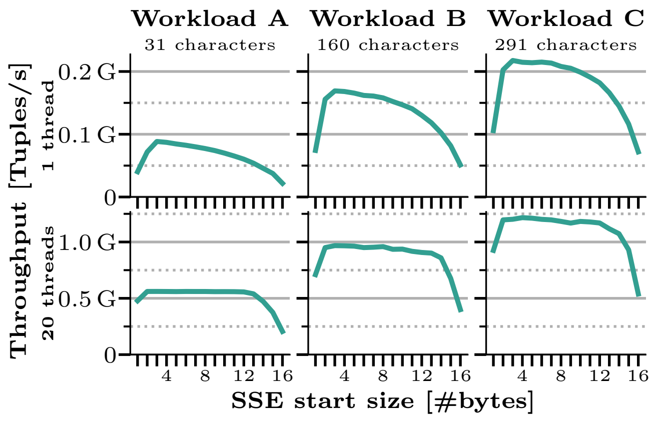


Figure 9: Experimental evaluation for the optimal start block size of the SSE Search. We found using 3 to 5 bytes of the pattern yields the highest performance. In case of mismatches, this range allows shifts of 14 to 12 bytes, respectively.

the pattern to the right while minimizing false positives. When utilizing 20 threads, the performance remains largely unaffected by the size of the start block. The limiting factor in this scenario is the available memory bandwidth, which operates at 68 GB/s and is utilized over 90%.

Moreover, the combination of longer patterns and the early return implementation proves advantageous, leading to increased throughput with larger pattern sizes.

4.3.2 Algorithm overview. Finally, the comparison of different code generating algorithms using 20 threads in terms of long patterns is illustrated in Figure 10. For the SSE Search algorithms, we have chosen the start size with the highest performance. Since the fallback option for the Hybrid-Search algorithm is the Two-Way algorithm for long patterns, both algorithms show similar performance.

For all patterns, the SSE Search algorithm, which generates pattern specific code for the matching process, outperforms the other algorithms. Furthermore, as patterns become longer, the algorithms

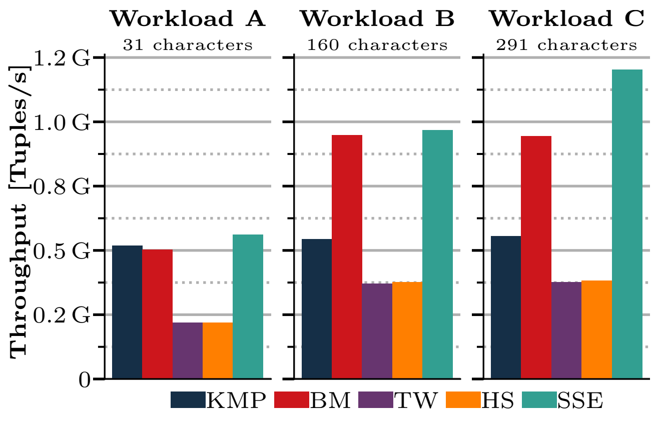


Figure 10: Performance of the code generating pattern matching algorithms for the long patterns using 20 threads.

demonstrate improved performance, as more input texts are too short for the given pattern. The performance for the Boyer-Moore algorithm is quite similar, except for the pattern **C** which is composed of multiple long patterns. In this case, the SSE Search algorithm clearly outperforms the others. Despite the increasing length of the pattern, the performance of the Knuth-Morris-Pratt algorithm only increases marginally compared to the other algorithms.

5 LESSONS LEARNED

Each matching algorithm combined with a code generation approach offers different performance benefits and use cases but also challenges.

Knuth-Morris-Pratt is relatively straightforward to implement in all three approaches. It can be enhanced by adding blockwise optimization with just a few modifications. Our experiments demonstrated that the code-generating approach significantly improved the performance of the KMP algorithm. The blockwise version of KMP is particularly effective when the first character of the pattern has a low frequency of occurrence in the input text. This allows efficient consumption of large portions of the text. Applying the *early return* optimization further enhances the performance. This optimization discards longer patterns faster once they no longer fit in the input text. Additionally, the KMP algorithm iterates over the input text from left to right. Thus, the algorithm can also process texts with Unicode characters based on their codepoints rather than only their bytewise representation.

Boyer-Moore is also easy to realize in the *Naïve* and *Preprocessed* approaches. The *Generated* approach requires more bookkeeping during code generation for correct SSA form. The experiments show that *Generated* is superior, while *Preprocessed* is better in case of hyperthreading. This algorithm is more effective when the last pattern character has a lower distribution than the first character. It also works better with longer patterns due to early rejection once they exceed the input text.

Two-Way is complex to implement in both approaches. In our experiments, *Generated* is slightly faster than *Naïve*, due to its less costly preprocessing function. However, the performance varies depending on pattern factorization. The pattern of the experiment was not optimally factorized, leading to a similar performance as for the KMP algorithm. With a pattern better suited for factorization, performance improves.

Hybrid-Search's complexity is relatively low when using the *Naïve* approach, and it depends solely on the chosen default algorithm. Implementing the SSE search component is a simple task and completely decoupled from the fallback algorithm. However, integrating the *Generated* approach into Umbra and its backends requires more effort. This is because we needed to introduce a new internal instruction for the SSE string comparison function which then maps to the corresponding function for the backend. Nevertheless, this algorithm shows the most promise and consistently outperforms the other algorithms in all three approaches. The *Generated* approach is only limited by the memory speed of the machine. Since the SSE part has a pattern length restriction, this algorithm is particularly suitable for short patterns. For longer patterns, it is necessary to carefully investigate the selected default algorithm.

SSE Search introduces an innovative method for generating specialized code for long patterns by leveraging SSE vector instructions. Incorporating this algorithm into a code-generating database engine is straightforward. The only challenge is adding the necessary SSE instructions to the backends. Furthermore, this approach facilitates the seamless implementation of various shift heuristics and dynamic adjustment of the size of the start block. The performance of the algorithm surpasses that of alternative methods across all three versions, consistently delivering superior results. Moreover, the performance is mostly bound by the available memory bandwidth.

Ultimately, tuning the performance for the pattern-matching algorithms in a code-generating database system is still a trade-off. The *Preprocessed* approach is sufficient to improve performance compared to the classic *Naïve* approach while keeping the complexity of generated code low. However, generating pattern-specific code for the matching process further improves overall query throughput at the cost of increased code complexity. Utilizing specific SSE instructions for pattern matching offers the dual advantage of enhancing performance and reducing code complexity in certain aspects of the matching algorithm. Based on our experimental findings, it can be inferred that when the required SSE instructions are not supported by the hardware, no single matching algorithm exhibits consistently superior performance across all patterns. However, if hardware support is available, we can conclude that for short patterns the Hybrid Search algorithm is superior, while for long patterns, the new SSE Search algorithm is more effective. Integrating both the Hybrid Search algorithms and the SSE Search algorithm into a code-generating database engine can be achieved seamlessly by designating the SSE Search algorithm as the default fallback. Furthermore, employing algorithms that utilize SSE instructions offers an additional advantage. The code required in the database engine for these algorithms is relatively small and straightforward to maintain and the generated code is clear and well-structured, facilitating easy verification and debugging processes.

6 CONCLUSION

This paper demonstrates the effectiveness of code generation for pattern-matching algorithms to evaluate LIKE predicates. The performance increases by up to a factor of two compared to function calls and outperforms state-of-the-art systems like Postgres, DuckDB, Hyper, or ClickHouse on text-heavy datasets like ClickBench. The results indicate that replacing generic function calls with pattern-specific generated code significantly increases the throughput. We have also demonstrated that using SSE instructions to compare packed strings has a positive impact on overall performance, particularly when incorporating them into the generated code. Additionally, we have presented a generalized algorithm which uses multiple SSE instructions to perform efficient pattern matching for long patterns as a favorable alternative to classic algorithms.

REFERENCES

[1] 2021. .NET: Compilation and Reuse in Regular Expressions. https://learn.microsoft.com/en-us/dotnet/standard/base-types/compilation-and-reuse-in-regular-expressions.

[2] Sameer Agarwal, Davies Liu, and Reynold Xin. 2016. *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop.* https://www.databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html

[3] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 11 (2020), 2649–2661.

[4] Robert S. Boyer and J. Strother Moore. 1977. A Fast String Searching Algorithm. *Commun. ACM* 20, 10 (1977), 762–772.

[5] Maxime Crochemore and Dominique Perrin. 1991. Two-Way String Matching. *J. ACM* 38, 3 (1991), 651–675.

[6] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD Conference.* ACM, 1243–1254.

[7] Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, and Jana Giceva. 2023. Bringing Compiling Databases to RISC Architectures. *Proc. VLDB Endow.* 16, 6 (2023), 1222–1234.

[8] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. 2016. Compression-Aware In-Memory Query Processing: Vision, System Design and Beyond. In *ADMS/IMDM@VLDB (Lecture Notes in Computer Science)*, Vol. 10195. Springer, 40–56.

[9] Todd Hoff. 2016. *Code Generation: The Inner Sanctum Of Database Performance.* http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html

[10] R. Nigel Horspool. 1980. Practical Fast Searching in Strings. *Softw. Pract. Exp.* 10, 6 (1980), 501–506.

[11] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE.* IEEE Computer Society, 195–206.

[12] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905.

[13] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350.

[14] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive Execution of Compiled Queries. In *ICDE.* IEEE Computer Society, 197–208.

[15] A.M. Kuchling. 2023. Regular Expression HOWTO. https://docs.python.org/3/howto/regex.html.

[16] Leslie Lamport. 1975. Multiple Byte Processing with Full-Word Instructions. *Commun. ACM* 18, 8 (1975), 471–475.

[17] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference.* ACM, 743–754.

[18] M. Lothaire. 1997. *Combinatorics on words, Second Edition.* Cambridge University Press.

[19] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550.

[20] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR.* www.cidrdb.org.

[21] R.M. Ramanathan. 2006. Extending the World's Most Popular Processor Architecture. https://web.archive.org/web/20090823145906/http://download.intel.com/technology/architecture/new-instructions-paper.pdf

[22] Wojciech Rytter. 1980. A Correct Preprocessing Algorithm for Boyer-Moore String-Searching. *SIAM J. Comput.* 9, 3 (1980), 509–512.

[23] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating Pattern Matching Queries in Hybrid CPU-FPGA Architectures. In *SIGMOD Conference.* ACM, 403–415.

[24] Evangelia A. Sitaridi, Orestis Polychroniou, and Kenneth A. Ross. 2016. SIMD-accelerated regular expression matching. In *DaMoN.* ACM, 8:1–8:7.

[25] Ken Thompson. 1968. Regular Expression Search Algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

[26] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD.* ACM, 1:1–1:6.

[27] Skye Wanderman-Milne and Nong Li. 2014. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.* 37, 1 (2014), 31–37.

[28] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *NSDI.* USENIX Association, 631–648.

[29] Paul Wankadia. 2021. Glossary. https://github.com/google/re2/wiki/Glossary.