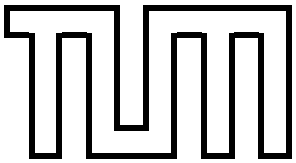


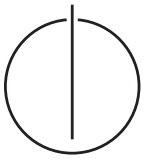
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

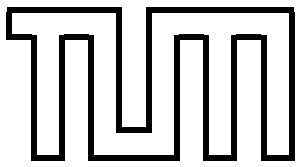
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Code Generation for Pattern Matching in DBMS

Adrian Riedl





SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY - INFORMATICS

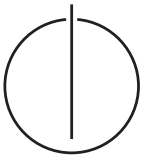
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Code Generation for Pattern Matching in DBMS

Effiziente Code-Generierung für die musterbasierte Suche in DBMS

Author:	Adrian Riedl
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Philipp Fent, M.Sc.
Submission Date:	January 18, 2023



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, January 17, 2023 Adrian Riedl

Acknowledgments

I would like to express my gratitude to my advisor Philipp Fent and my supervisor Prof. Dr. Thomas Neumann for the opportunity to work on this interesting topic. Furthermore, I would like to thank Philipp for his support and time he spent with discussing the different approaches, giving hints, checking the results and their interpretation, and everything he did for me. Many thanks to Johannes for proof-reading this thesis, the years of friendship, and the funny times inside and outside of the lecture halls. I would like to thank many people, who supported me in my spare time – without you, this thesis would have turned out to be more difficult.

Lastly, thanks to my wonderful parents, who supported me over all the years and without whom I would not be, where I am right now.

Abstract

In this thesis, we examine the use of code generation for improving the performance of pattern matching in database systems. In databases, a text datatype is often used to store information when there is no appropriate datatype, and pattern matching on these texts is a critical aspect in many database systems. However, this matching process is often the most costly part of a query. By using code generation for the matching process, we show that query throughput can be improved by a factor of up to 2.

We focus on the LIKE expressions, which are offered by SQL to match a given pattern against input texts. We investigate the advantages of generating code for the exact pattern matching algorithms, as opposed to the traditional method of interpreting the pattern using hand-written C++ functions. For the commonly known matching algorithms Knuth-Morris-Pratt and Boyer-Moore, we explore different strategies to improve their performance. Additionally, we present a strategy to build an automaton for the pattern, which is used to process the input text and determine if a pattern is present.

For the discussed algorithms, we explain how these and their optimizations can be implemented in our database system Umbra using its custom code generation framework. The analysis evaluates the performance differences we could achieve when running several queries with both the interpreting and the code generating approaches. We also inspect the development of the compile time, as well as the effect when running the queries with multiple threads on a large dataset.

Our results demonstrate that there are multiple factors influencing the performance of pattern matching algorithms, but that code generation is worth the effort for most workloads. Our thesis provides insight and guidance for database developers looking to improve the performance of pattern matching in their systems.

Kurzfassung

In dieser Arbeit untersuchen wir die Verwendung der Codegenerierung zur Verbesserung der Leistung der musterbasierten Suche in Datenbanksystemen. In Datenbanken wird oft ein Textdatentyp verwendet, um Informationen zu speichern, wenn es keinen geeigneten Datentyp gibt, und die musterbasierte Suche für diese Texte ist ein wichtiger Aspekt in vielen Datenbanksystemen. Dieser Abgleichsprozess ist jedoch oft der kostspieligste Teil einer Abfrage. Wir zeigen, dass der Abfragedurchsatz durch die Verwendung von Codegenerierung für den Abgleichsprozess mitunter verdoppelt werden kann.

Wir konzentrieren uns auf die LIKE-Ausdrücke, die von SQL angeboten werden, um ein bestimmtes Muster mit Eingabetexten abzugleichen. Wir untersuchen die Vorteile der Codegenerierung für die exakten Algorithmen zur musterbasierten Suche im Gegensatz zur traditionellen Methode der Interpretation des Musters mit handgeschriebenen C++-Funktionen. Für die allgemein bekannten Matching-Algorithmen Knuth-Morris-Pratt und Boyer-Moore untersuchen wir verschiedene Strategien zur Verbesserung ihrer Leistung. Darüber hinaus stellen wir eine Idee zur Erstellung eines Automaten für das Muster vor, mit dem der Eingabetext verarbeitet und das Vorhandensein eines Musters entschieden werden kann.

Für die besprochenen Algorithmen erläutern wir, wie diese und ihre Optimierungen in unserem Datenbanksystem Umbra unter Verwendung seines eigenen Codegenerierungsgerüstes implementiert werden können. Die Analyse bewertet die Leistungsunterschiede, die wir bei der Ausführung mehrerer Abfragen sowohl mit dem interpretierenden als auch mit dem codegenerierenden Ansatz erzielen konnten. Wir untersuchen auch die Entwicklung der Kompilierzeit sowie die Auswirkungen, wenn die Abfragen mit mehreren Threads auf einem großen Datensatz ausgeführt werden.

Unsere Ergebnisse zeigen, dass es mehrere Faktoren gibt, die die Leistung von Pattern-Matching-Algorithmen beeinflussen, dass aber die Codegenerierung für die meisten Arbeitslasten den Aufwand wert ist. Unsere Arbeit bietet Einblicke und Anleitungen für Datenbankentwickler, die die Leistung von musterbasierten Suchen in ihren Systemen verbessern möchten.

Contents

Acknowledgments **v**

Abstract **vii**

Kurzfassung **ix**

1. Introduction **1**
1.1. Motivation . . . 1
1.2. State of the Art . . . 2
1.3. Structure . . . 3

2. Exact String Pattern Matching **5**
2.1. LIKE Patterns in SQL . . . 5
2.2. Knuth-Morris-Pratt Algorithm . . . 6
2.2.1. Preprocessing . . . 6
2.2.2. Original KMP . . . 8
2.2.3. KMP with One Loop . . . 8
2.2.4. Optimizations . . . 9
2.2.4.1. Early Return . . . 10
2.2.4.2. Compression of the LPS Table . . . 10
2.2.4.3. Blockwise Processing . . . 11
2.3. Boyer-Moore Algorithm . . . 12
2.3.1. Preprocessing . . . 12
2.3.1.1. Bad Character Heuristics . . . 12
2.3.1.2. Good Suffix Heuristics . . . 13
2.3.2. Original BM . . . 14
2.3.3. Fast BM . . . 15
2.3.4. Blockwise BM . . . 16
2.4. Automaton Approach . . . 17
2.4.1. Non-Deterministic Finite Automaton for LIKE patterns . . . 17
2.4.2. Deterministic Finite Automaton for LIKE patterns . . . 18

3. Code Generation for Exact String Pattern Matching **21**
3.1. Code Generation Framework in Umbra . . . 21
3.2. Knuth-Morris-Pratt Algorithm . . . 22
3.2.1. Original KMP . . . 22
3.2.2. KMP with One Loop . . . 23
3.2.3. Adding Optimizations to Code Generation Process . . . 25
3.3. Boyer-Moore Algorithm . . . 27
3.3.1. Original BM . . . 27
3.3.2. Fast BM . . . 28
3.3.3. Blockwise BM . . . 28

3.4. Automaton Approach . . . 30
3.4.1. Direct Translation . . . 30
3.4.2. Blockwise Translation . . . 31
3.5. Concatenating Multiple Subpatterns . . . 32

4. Evaluation for Exact String Pattern Matching **33**
4.1. Experimental Setup . . . 33
4.1.1. Hardware Specification . . . 33
4.1.2. Data and Queries . . . 33
4.1.2.1. TPC-H Data . . . 33
4.1.2.2. ClickBench . . . 34
4.1.3. Query Settings . . . 34
4.2. Results . . . 35
4.2.1. Knuth-Morris-Pratt Algorithm . . . 35
4.2.1.1. Regular LPS Table . . . 35
4.2.1.2. Compressed LPS Table . . . 38
4.2.2. Boyer-Moore Algorithm . . . 39
4.2.3. Automaton Approach . . . 41
4.2.4. Comparison of the Code Generating Algorithms . . . 42
4.2.5. Compilation Time . . . 44
4.2.6. ClickBench Results . . . 45

5. Non-Exact Pattern Matching **49**
5.1. Extended Automaton Approach . . . 49
5.1.1. Extended Like-NFA . . . 49
5.1.2. Extended Like-DFA . . . 50
5.2. Reduced Automaton Approach . . . 52

6. Conclusions and Outlook **55**
6.1. Code Generation Independent Insight . . . 55
6.2. Code Generation Dependent Insight . . . 56
6.3. Outlook . . . 56

A. Blockwise Processing **59**
A.1. Blockwise Search for ASCII character - Example . . . 59
A.2. Blockwise Search for Non-ASCII character . . . 60

B. Determinization of Like-NFA **61**

C. Distribution of ClickBench Dataset **63**

List of Figures **65**

List of Tables **67**

Listings **69**

Bibliography **71**

1. Introduction

Pattern matching is a prevalent problem in various fields of computer science, with a wide range of applications: the Unix command *grep* to search for certain patterns in files; in internet browsers to scan through massive amounts of text available online for specific keywords; online newspapers to filter the articles based on the field of interests; online journals allowing to scan through the content for specific catchwords; and several specialized databases, especially in the field of molecular biology to process patterns on the stored DNA or RNA strings.

Given the numerous applications, exact pattern matching has been an extensively researched topic in the past. With the advancement in technology and the widespread availability of search functions in various software, it is now considered a granted functionality. However, libraries such as Google's `re2`[1], the `Regex` class from Microsoft's .NET framework[2], and modules of programming languages like `std::regex` from the C++ standard library[3] or the `re` module of python[4] are still actively developed to improve the speed and efficiency of pattern matching. All of those deal with matching given regular expressions against an input text. For those libraries, it is not only about searching the pattern in the text, but also about doing this as fast as possible.

Relational databases also need to be able to evaluate pattern matching expressions in their queries, and thus, it is essential to process the data quickly while maintaining precision. SQL syntax allows for the use of LIKE expressions, which are a somewhat weakened version of regular expressions in terms of the matching process.

1.1. Motivation

Standardized benchmarks like TPC-H or TPC-C are commonly used to evaluate the performance of both academic and industrial database systems. These benchmarks primarily test the functionalities of the database system, such as the ability of the optimizer to find the best join order or the correct implementation of the individual operators. However, as Vogelsgesang et al. discuss in their paper [Vog+18], these benchmarks may not accurately reflect the real-world challenges that a database system needs to manage. Through their analysis of over 60k data repositories and the generated query workload, they identify several insights.

One of the key findings of their study is that almost 50% of the attributes of the tuples in the analyzed datasets are stored as text. This highlights the importance of efficient pattern matching, as it is a critical component of query performance on these datasets. Due to this, LIKE pattern matching in database systems is not yet fully solved.

[1] `https://github.com/google/re2` (accessed: 03.12.2022)
[2] `https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions` (accessed: 03.12.2022)
[3] `https://en.cppreference.com/w/cpp/regex` (accessed: 03.12.2022)
[4] `https://docs.python.org/3/library/re.html` (accessed: 03.12.2022)

In 2011, Alfons Kemper and Thomas Neumann introduced the research database HyPer, a pure main memory database for both OLTP and OLAP workloads [KN11]. Later, Thomas Neumann proposed a data-centric approach for generating and compiling compact and efficient machine code using the LLVM compiler framework for queries in relational database systems [Neu11]. This approach offers a new way to evaluate LIKE expressions in relational database systems by generating code for the matching process instead of interpreting the pattern. The traditional approach is to use a hand-written function in the database system to perform the pattern matching on the input text. By using the code generating approach, the matching process is integrated into the generated code for the entire query, replacing the function call in the interpreting approach.

This thesis presents a detailed examination of how to integrate pattern matching for LIKE expressions into the compiling database engine Umbra, an evolution of the pure main memory database system HyPer towards a SSD-based system [NF20]. It explores different exact pattern matching algorithms in both the interpreting and code generating versions, as well as various modifications and optimizations. Through this examination, it aims to answer the question of whether to interpret or generate code for pattern matching expressions in queries.

1.2. State of the Art

Regular expression matching can be improved by using code generation and just-in-time techniques. There are several libraries such as Google's `re2` library, Microsoft's `.NET` framework, and the `re` module of the programming language python that use code generation to convert the given regular expression pattern into an internal representation of bytecode. In the case of the python's `re` module, the internal representation is executed by a matching engine written in C. Google's `re2` represents the regular expression as an automaton in bytecode, which is then interpreted by the execution engine. Microsoft's `.NET` framework is an exception, as it provides an option to compile the regular expression to machine code when creating a regular expression object. This option uses a just-in-time compiler to convert the expression to native machine code. However, this option needs to be set explicitly, otherwise the internal representation is interpreted [Mic].

The concept of generating code to match regular expressions dates back to 1968, when Ken Thompson presented a method to produce an IBM 7094 program as object language from a regular expression to locate a specific character sequence in a character text [Tho68]. Today, many regular expression matchers use internal representations to perform matching. However, there is only a limited number of projects that aim to generate and execute machine code to support regular expression matching at runtime.

As code generation has become widely accepted in the database system community [ALX16; Dia+13; Hof16; WL14], it is interesting to investigate the benefits of generating code for the matching process instead of interpreting a given pattern for a query. To examine this, we will focus on a simplified version of regular expressions, specifically LIKE expressions, which are offered in SQL. Through this investigation, we aim to make a contribution to both the database and regular expression communities by reevaluating the capabilities of code generation for the purpose of pattern matching.

1.3. Structure

The outline of this thesis is as follows. In Chapter 2, we provide the necessary foundations for our research. Sect. 2.1 introduces the LIKE pattern in SQL and the available wildcards. Sections 2.2, 2.3, and 2.4 present the algorithms that we focus on in this thesis for pattern matching and their adaptions to improve the search process.

In Chapter 3, we summarize the code generation process of Umbra in Sect. 3.1. Sections 3.2, 3.3, and 3.4 explain the code generation process for the Knuth-Morris-Pratt, Boyer-Moore, and Automaton algorithms respectively, highlighting the optimizations included in each process.

Chapter 4 presents the results of the experiments to compare the performance of code generation with an interpreting version. In Sect. 4.1, we describe the hardware and the experimental setup used. Sect. 4.2 presents the results of our experiments and explains the observed behavior. We evaluate each algorithm individually before comparing them in one graph. Additionally, we include the effect of compilation time in Sect. 4.2.5 and the results when running queries on a larger dataset with multiple threads in Sect. 4.2.6.

In Chapter 5, we present two approaches to manage non-exact pattern matching for LIKE expressions in Umbra. Both ideas are based on the idea of the Automaton Approach.

At the end of this thesis, the outcome of the experiments is presented in Chapter 6. Therefore, we summarize what we could figure out and present the facts split up in two parts; the first is in Sect. 6.1 and describes everything that applies for both interpreting and code generating cases, while Sect. 6.2 highlights the outcome of our comparison. Finally, Sect. 6.3 presents some ideas of future work to be done on the basis of this project.

2. Exact String Pattern Matching

Focusing on exact pattern matching in relational database systems, let us assume we want to count all students who contain the pattern `'en'` in their name. To accomplish this, we must iterate through all student records, process their names, and check for an exact match with the desired pattern. Efficient and reliable pattern matching algorithms are crucial to avoid performance issues in such queries.

To perform such a matching process, SQL offers the LIKE expression which will be discussed in Sect. 2.1. Focusing on exact pattern matching in this chapter, we present the Knuth-Morris-Pratt algorithm in Sect. 2.2 and introduce some optimizations which can be applied to speed up the search. Sect. 2.3 covers the Boyer-Moore algorithm and Sect. 2.4 covers the Automaton Approach, where the pattern is converted into an automaton and used to evaluate the input text.

2.1. LIKE Patterns in SQL

According to the SQL-92 standard [Iso] and the PostgreSQL documentation [Pos], the LIKE expression in SQL is used to match a text with a specific pattern. The syntax for this expression is as follows: `string LIKE pattern [ESCAPE escape-character]`. The pattern can contain two wildcards: the underscore _ represents a match with any single character and the percent sign % represents a match with any sequence of zero or more characters. If the pattern does not contain any wildcard, the pattern represents the string itself and the operator acts like an equals operator. For exact pattern matching, the pattern must only contain percent wildcards.

To match a literal underscore or percent sign without interpreting the character as a wildcard, the escape character must precede the corresponding character. The default escape character is the backslash \, but a custom one can be defined by using the ESCAPE clause. To match the escape character itself, it must be written twice.

Example. Let us consider the following pattern: `red%green%blue`. For a given input text to be considered a match, it must start with the character sequence **red**, contain **green** somewhere in the middle, and end with **blue**. In Table 2.1, we present examples of texts and whether they are accepted or not based on this pattern.

From this example, it can be seen that the pattern can be broken down into several parts: the prefix, the suffix, and the patterns in between. In exact pattern matching, the prefix and suffix can be checked at the start and end of the input text directly. However, for patterns surrounded by two percent wildcards, a pattern matching algorithm must be applied to search for those within the input text. Unless stated otherwise, we will refer to this process of finding the subpatterns surrounded by two % wildcards.

text	accepted	explanation
red orange **green** purple **blue**	✓	
red orange purple **green blue**	✓	
red orange **green** blue *purple*	✗	suffix mismatches
red orange purple **blue**	✗	pattern **green** is not found

Table 2.1.: Example texts and results for the pattern `red%green%blue`. **Bold** part of the text means an expected match with the pattern; *italics* means a mismatch.

2.2. Knuth-Morris-Pratt Algorithm

In 1977, Knuth, Morris, and Pratt introduced an algorithm for performing exact string pattern matching to find all occurrences of a given pattern in a text [KMP77]. The objective of the algorithm is to find the pattern in the text without the need to backtrack in the input text and to only use the information from the pattern itself to determine where the search continues. This means that, once a character from the input text at index i is read, no character from an index smaller than i will be read again. Before executing the algorithm, the pattern needs to be preprocessed as discussed in Sect. 2.2.1. Then, we will consider two different implementations of the KMP algorithm:

1. Original KMP in Sect. 2.2.2
2. KMP with One Loop in Sect. 2.2.3.

Finally, we discuss three optimizations in Sect. 2.2.4 and explain how to apply them to the KMP algorithms to speed up the search.

2.2.1. Preprocessing

In the preprocessing phase of the KMP algorithm, the longest proper prefix which is also proper suffix (lps) is searched for each substring of the pattern from its start up to the given position i. Therefore, let us introduce a formal definition of the terminology [Lan01b]:

Definition: Let A be an alphabet and $x = x_0 \dots x_{k-1}, k \in \mathbb{N}$ be a string of length k over A.

Prefix: A prefix of x is a substring u with $u = \begin{cases} \epsilon & b = 0 \\ x_0 \dots x_{b-1} & b \in \{1, \dots, k\} \end{cases}$ i.e. x starts with u.

Suffix: A suffix of x is a substring u with $u = \begin{cases} \epsilon & b = 0 \\ x_{k-b} \dots x_{k-1} & b \in \{1, \dots, k\} \end{cases}$ i.e. x ends on u.

Proper Prefix/Suffix: A prefix u of x / suffix u of x is called proper prefix/suffix if $u \neq x$, i.e. its length b is less than k.

Border: A border of x is a substring $r = \begin{cases} \epsilon & b = 0 \\ x_0 \dots x_{b-1} = x_{k-b} \dots x_{k-1} & b \in \{1, \dots, k\} \end{cases}$.A border of x is a substring that is both proper prefix and proper suffix of x. b denotes the width of the border.

In the preprocessing phase of a pattern p, we compute a lps array with the length $|p|+1$. Each entry lps$[i]$ contains the width of the widest border of the prefix of length i of the pattern p with $i \in \{0, \ldots, |p|\}$. For the prefix ϵ of length $i = 0$, we set lps$[0] = -1$.

Listing 2.1 shows the algorithm to compute the lps array for a given pattern.

```
1  preprocess(pattern)
2    vector<int> lps(pattern.size() + 1, 0);
3    i = 0, j = -1;
4    lps[i] = j;
5
6    while (i < pattern.size())
7      while (j >= 0 && pattern[j] != pattern[i])
8        j = lps[j];
9      i++; j++;
10     lps[i] = j;
11   return lps;
```

Listing 2.1: Preprocessing of KMP

As presented in Fig. 2.1, the lps table is used to determine the shift after a mismatch occurred when comparing the pattern and text. If at position i in the text there is a mismatch with the j^{th} character of the pattern, then the entry in the lps table holds the length of the suffix α which is also the longest proper prefix of the compared part of the pattern. Using this value, we can directly determine how to shift the pattern to correctly align its prefix α and can continue the search at position lps$[j]$.

Example. Table 2.2 presents the calculation of the lps table for the pattern `abaa` step by step. Therefore, we extract the prefix of the pattern with the length of i and analyze this prefix to find the longest proper prefix which is also suffix. Its length is then stored at the index i in the lps table. In the visualization, the stored numbers are printed bold to show which have already been calculated.

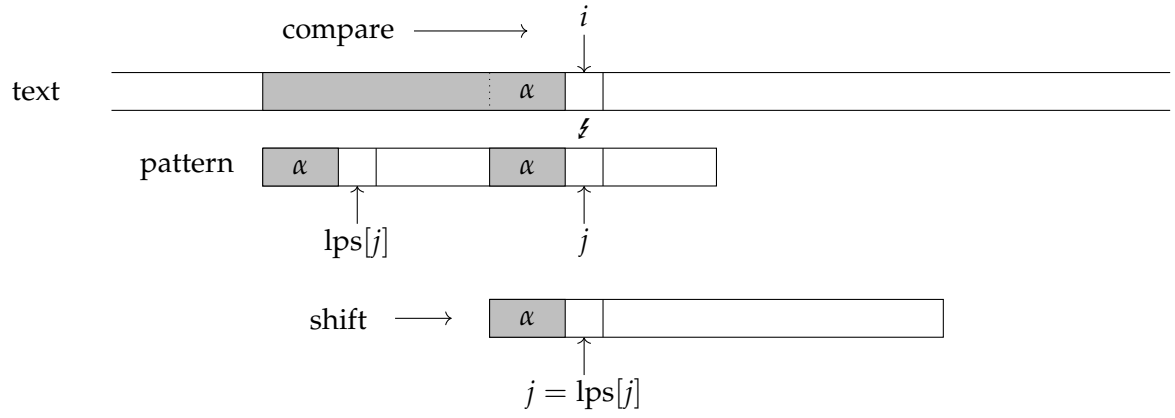


Figure 2.1.: Visualization of the shift of the pattern when mismatch at j occurs

i	a	b	a	a		determine lps[i]
0	0	0	0	0	0	prefix = $\epsilon \rightarrow$ lps[0] $= -1$
1	**-1**	0	0	0	0	prefix = a $\rightarrow$ lps[1] $= 0$
2	**-1**	**0**	0	0	0	prefix = ab $\rightarrow$ lps[2] $= 0$
3	**-1**	**0**	**0**	0	0	prefix = aba $\rightarrow$ lps[3] $= 1$
4	**-1**	**0**	**0**	**1**	0	prefix = abaa $\rightarrow$ lps[4] $= 1$
	-1	**0**	**0**	**1**	**1**	

Table 2.2.: Process to build the lps table for the pattern abaa. The bold numbers were already calculated in the previous iterations.

2.2.2. Original KMP

The pseudocode for the Original KMP algorithm, as proposed by Knuth, Morris, and Pratt in [KMP77], is presented in Listing 2.2: The first step of the algorithm is to preprocess the pattern to calculate the corresponding lps table (line 2). In line 5, we start iterating over the text from left to right. In the lines 6 - 10, the current character of the text is compared with the corresponding character from the pattern. In case of mismatching characters, the position in the pattern is updated by reading the shift from the lps table; in case of matching characters, based on the position in the pattern, either a match is reported or the position in the pattern is incremented (line 10). Finally, as the iteration over the input text progresses, the position in the text is incremented by one (line 11).

```
1  O-KMP(text, pattern)
2    lpsTable = preprocess(pattern);
3    pPos = 0, tPos = 0;
4
5    while (tPos < text.size())
6      while (pPos && pattern[pPos] != text[tPos])
7        pPos = lpsTable[pPos];
8      if (pattern[pPos] == text[tPos])
9        if (pPos == pattern.size() - 1) return true;  // pattern found
10       else pPos++;
11     tPos++;
12   return false;  // pattern not found
```

Listing 2.2: Original KMP (O-KMP)

Example. We search for the pattern abaa in the text abababacabaa. As in Listing 2.2, tPos refers to the position in the text and pPos to the position in the pattern. Table 2.2 shows the lps table, the result of the preprocessing of the given pattern; in Table 2.3, the previously discussed algorithm is performed and explained step by step.

2.2.3. KMP with One Loop

The Original KMP algorithm can also be implemented with only one loop. The corresponding pseudocode can be found in Listing 2.3. The first step in line 2 is to preprocess the pattern by generating the lps table. The algorithm also iterates from left to right over the input text and

tPos	pPos	a	b	a	b	a	c	a	b	a	a	
0	0	a	b	a	a							characters match
1	1	a	b	a	a							characters match
2	2	a	b	a	a							characters match
3	3	a	b	a	a							mismatch, pPos ← lps[3] = 1
3	1			a	b	a	a					characters match
4	2			a	b	a	a					characters match
5	3			a	b	a	a					mismatch, pPos ← lps[3] = 1
5	1					a	b	a	a			mismatch, pPos ← lps[1] = 0
5	0						a	b	a	a		mismatch, pPos = 0, increment tPos
6	0							a	b	a	a	characters match
7	1							a	b	a	a	characters match
8	2							a	b	a	a	characters match
9	3							a	b	a	a	pattern found

Table 2.3.: KMP search for pattern `abaa` in text `ababacabaa`

checks if the characters at the text and pattern position match (line 6). In case of a match, the positions are incremented and in line 8, we check if the pattern end is reached, in which case, we return that a match was found. If not, the algorithm continues. In case of no match, the shift is looked up in the lps table to find out where the search needs to continue (line 10). Due to the construction of the lps table, we need to add a check whether the shift is negative. If it is, the first character of the pattern mismatched, so the pattern can be moved one position to the right to restart the search from the pattern start, as no more information is available at this point (line 11); otherwise, the position in the pattern is updated to the shift and the search continues (line 12).

```
1  OL-KMP(text, pattern)
2    lpsTable = preprocess(pattern);
3    pPos = 0, tPos = 0;
4
5    while (tPos < text.size())
6      if (pattern[pPos] == text[tPos])
7        pPos++; tPos++;
8        if (pPos == pattern.size()) return true;  // match found
9      else
10       shift = lpsTable[pPos];
11       if (shift < 0) pPos = 0; tPos++;
12       else pPos = shift;
13   return false;  // no match found
```

Listing 2.3: KMP with One Loop (OL-KMP)

2.2.4. Optimizations

In this section, we present three optimization ideas that can be used to improve the performance of the KMP algorithm when searching for a specific pattern in an input text. These techniques are assumed to be effective for sufficiently large alphabets. The first optimization, presented in Sect. 2.2.4.1, allows for early termination of the algorithm if the remaining text

length is insufficient for a match. The second optimization, presented in Sect. 2.2.4.2, is specific to the KMP algorithm. Lastly, the third optimization discussed in Sect. 2.2.4.3 is more general and can be applied to other algorithms as well.

2.2.4.1. Early Return

The first optimization we implement in all KMP algorithms is the early return. This means we do not iterate over the entire input text, but instead check in each iteration if the end of the pattern is still within the text. By doing this, we can stop the comparison once the pattern end goes beyond the text end, as no match can be found in this scenario. In future visualizations of the control flow for the generated code, we will directly use this improved loop condition.

2.2.4.2. Compression of the LPS Table

The compression of the lps table can help avoid the "bouncing ball" effect observed in the Original KMP algorithm as discussed by Dan Gusfield in [Gus97, p. 48-52]. This refers to the scenario where the same letter in the pattern is seen consecutively at different positions multiple times. When looking at the regular lps table of the pattern `abaabab` presented in Table 2.4, let us assume the character `a` at index 5 has a mismatch with the character from the input text. Then, the position in the pattern is updated to index 2 at which again the letter `a` is found. As this letter also mismatches, the position in the pattern is updated to the position 0. The result of these jumps is that the pattern needs to be shifted one to the right of the current position in the text and begin the search from the pattern start.

To avoid this jumping, the preprocess function needs to be modified as presented in Listing 2.4. Compared with the original preprocessing algorithm, the only change is in line 10. In this line, we now do not directly set the entry to the lps table, but check whether the characters at the positions are equal and choose the corresponding value for the entry.

Table 2.4 also shows the result of the compressed preprocessing function for the pattern `abaabab`. By using the compressed version of the preprocessing, the number of comparisions between the character from the pattern and from the input text and also the jumps backwards in the pattern can be reduced. However, the Original KMP algorithm needs to be adapted to work with the compressed lps table properly.

index	0	1	2	3	4	5	6	
pattern	`a`	`b`	`a`	`a`	`b`	`a`	`b`	
regular lps table	-1	0	0	1	1	2	3	2
compressed lps table	-1	0	-1	1	0	-1	3	2

Table 2.4.: Regular and compressed LPS table for the pattern `abaabab`. The arrows visualize the "bouncing ball" if a mismatch at index 5 occurred.

```
preprocessCompressed(pattern)
  vector<int> lps(pattern.size() + 1, 0);
  i = 0, j = -1;
  lps[i] = j;

  while (i < pattern.size())
    while (j >= 0 && pattern[j] != pattern[i])
      j = lps[j];
    i++; j++;
    lps[i] = (pattern[i] == pattern[j]) ? lps[j] : j;
  return lps;
```

Listing 2.4: Compressed preprocessing of KMP

2.2.4.3. Blockwise Processing

An alternative way to improve the performance of the search is to optimize the process of finding the first character of the pattern. In real-world text inputs, it is common for the first character of the pattern to mismatch frequently. This results in the KMP algorithm repeatedly moving one character forward. To avoid this, blockwise processing can be used to quickly find the first character of the pattern.

Listing 2.5 demonstrates how to use bitwise operations to check if the ASCII character `'c'` is present in `'block'`. You can find an example of this function in Appendix A.1. Additionally, we have provided a modified version of the function to search for a character whose highest bit is set to 1, along with a corresponding example in Appendix A.2.

```
uint64_t block = loadNext8Bytes(...);
uint64_t searchedChar = broadcast(c); // broadcast c to each byte
const uint64_t high = 0x8080808080808080ull;
const uint64_t low = ~high;
uint64_t lowChars = (~block) & high;
uint64_t found = ~((((block & low) ^ searchedChar) + low) & high);
uint64_t matches = found & lowChars;
bool matchFound = matches != 0;
```

Listing 2.5: Blockwise search for ASCII character `c`

2.3. Boyer-Moore Algorithm

The Boyer-Moore algorithm, published in 1977 by Boyer and Moore, is a widely-used method for searching a given pattern in an input text [BM77]. The original paper presented the algorithm but not the necessary preprocessing of the pattern. A follow-up paper was published with the preprocessing information, but it contained errors that were corrected by Rytter in 1980 [Ryt80].

The basic idea of the algorithm is to scan the pattern from right to left and on mismatches, one of the shift rules are applied, which are the result of the preprocessing. In Sect.2.3.1, we will discuss the preprocessing of the algorithm. We will also explain the different versions of the algorithm presented in the original paper in Sect. 2.3.2 and 2.3.3. Finally, we will introduce an idea on how to add blockwise processing to the Boyer-Moore algorithm in Sect. 2.3.4.

2.3.1. Preprocessing

The Boyer-Moore algorithm has a preprocessing phase which includes two parts: the Bad Character Heuristics and the Good Suffix Heuristics. When the algorithm is executed, both rules are applied and the maximum possible shift is chosen. However, it is also possible to use only one of the rules and set the shift of the other rule to 1.

2.3.1.1. Bad Character Heuristics

The Bad Character Heuristics (BCH) is a preprocessing technique used in the Boyer-Moore algorithm to improve the performance of pattern matching. The basic idea behind BCH is to shift the pattern when a mismatch occurs between the input text and the pattern. The preprocessing step for BCH involves creating a table for all possible characters in the input alphabet. For each character in the pattern, the table contains the distance of its right-most position in the pattern to the end of the pattern. For all other characters, the shift distance is set to the length of the pattern.

When a mismatch occurs between the input text and the pattern, the algorithm checks the BCH table for the mismatched character. If the character is present in the pattern, the pattern is shifted so that the right-most occurrence of that character in the pattern aligns with the mismatched character in the text. However, if the character is not present in the pattern, the pattern is shifted to the right of the mismatched character. In Table 2.5, we show how the BCH can be applied to the search, once if the text character is present in the pattern and once if not.

While BCH can greatly improve the performance of the Boyer-Moore algorithm, it can also cause negative shifts. A negative shift occurs when the right-most occurrence of the mismatched character of the text is to the right of the current position in the pattern. This can lead to the pattern being shifted back and forth, making it impossible to find the matching part in the text. To resolve this shifting issue, the pattern needs to be moved to the right by 1 and restart the search. Table 2.6 presents a compact example of a negative shift which must not be performed. In this example, the pattern would always be shifted back and forth and the matching part in the text would not be found.

```
preprocessBad(pattern)
  vector<int> δ1(256, pattern.size());
  for (i = 0; i < pattern.size(); i++)
    δ1[pattern[i]] = pattern.size() - 1 - i;
  return δ1;
```

Listing 2.6: Preprocessing for the Bad Character Heuristics

a	b	a	b	a	c	a	b	a	a	
a	b	a	a							mismatch, character b on index 1 → shift by 2
		a	b	a	a					mismatch, character c not in pattern → shift by 4
						a	b	a	a	pattern found

Table 2.5.: Search for pattern abaa in text ababacabaa using Bad Character Heuristics

a	a	a	a	b	a	b	
a	b	a	b				mismatch, character a on index 2
	a	b	a	b			mismatch, character a on index 2
a	b	a	b				**must not** shift to the left

Table 2.6.: Negative shift with Bad Character Heuristics

2.3.1.2. Good Suffix Heuristics

The Good Suffix Heuristics (GSH) is a technique used to optimize the Boyer-Moore algorithm by determining the maximum possible shift for a given mismatch. To explain the Good Suffix Heuristics, let us consider Fig. 2.2, adapted from [Lan01a]:

Case A: For a given alignment of the pattern with the text, assume that the suffix α of the pattern matches a substring of the text, but a mismatch $x \neq y$ occurs at the next comparison to the left of the suffix. Let α' be the right-most copy of α in the pattern with the preceding character $z \neq y$. We can now shift the pattern to align the occurrence of α in the text with the occurrence of α' in the pattern.

Case B: If no such substring α' in the pattern exists, we try to align a prefix of the pattern with a matching suffix of α. If this is not possible, then the left side of the pattern is aligned with the right side of α in the input text, so the pattern is shifted completely to the right of the occurrence α.

The preprocessing phase for GSH involves creating a lookup table that stores the maximum shift for each position in the pattern. Listing 2.7 presents the pseudocode for generating the lookup table. The algorithm starts by iterating through the pattern from right to left and checking for any prefixes that match the suffix of the pattern. If a match is found, the index is stored for further calculations. Then, for each position in the pattern, the algorithm calculates the maximum shift by considering the distance of the current position to the right and the last known position of the prefix in the pattern. This allows for efficient alignment of the pattern in the event of a mismatch.

In the second loop, Case A is processed by iterating over the pattern from left to right, identifying the size of the longest suffix of the current substring that is also a suffix of the larger pattern. With this size, we update the entry in the table corresponding to the index of the longest possible suffix of the larger pattern. This value is the shift required to align the characters currently matching the suffix of the pattern with the suffix characters of the subpattern plus the number of characters to set the text position to the end of the aligned pattern.

```
1  preprocessGood(pattern)
2    δ2 = vector<int>(pattern.size());
3    lastPrefixPosition = pattern.size();
4
5    for (i = pattern.size(); i > 0; --i)
6      // isPrefix(pat, pos) returns
7      // pat[0 ... pat.size - pos[ == pat[pos ... pat.size[
8      if (isPrefix(pattern, i)) lastPrefixPosition = i;
9      δ2[i - 1] = lastPrefixPosition + pattern.size() - i;
10   for (i = 0; i < pattern.size() - 1; ++i)
11     // suffixSize(pat, pos) returns the maximal length of the suffix
12     // of pat[0 ... pos] which is also suffix of pat
13     δ2[pattern.size() - 1 - size] = pattern.size() - 1 - i + suffixSize(pattern, i);
14   return δ2;
```

Listing 2.7: Preprocessing for the Good Suffix Heuristics

2.3.2. Original BM

Boyer and Moore present two versions for their algorithm in [BM77]. The pseudocode for the original version is given in Listing 2.8: As the first two steps, the pattern is preprocessed to get the BCH table δ_1 and the GSH table δ_2. Next, the pattern is aligned with the text and the position in the text is set to point to the character aligned with the last character of the pattern. In the loop, we iterate over the input text as long as the input is not exhausted. As we always compare the pattern from right to left, we set `pPos`, the position in the pattern, to point to the

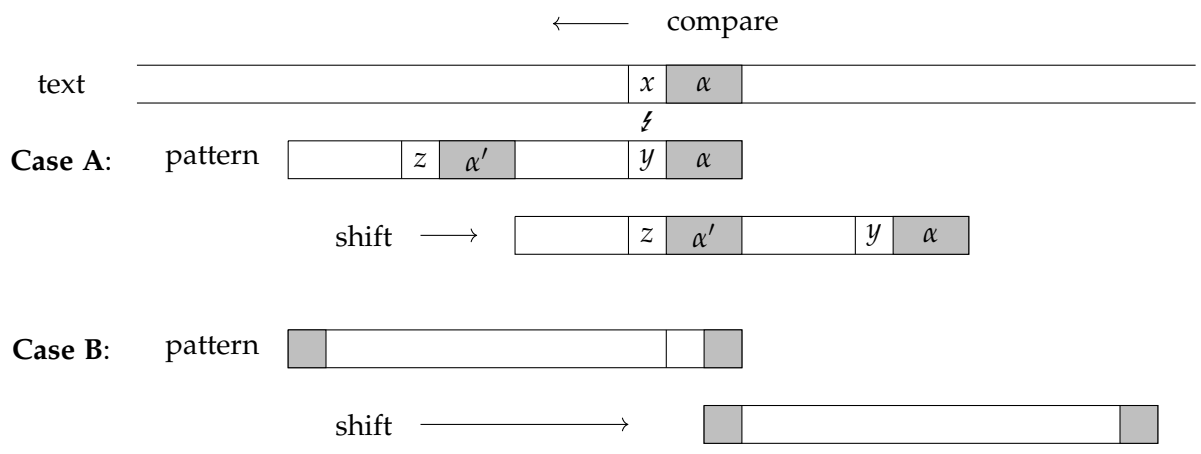


Figure 2.2.: Visualization of the shift by the Good Suffix Heuristics

last character of the pattern. In the inner loop, we compare the character from the pattern and from the text and move one character to the left if both coincide. Once a mismatch occurred or the index 0 of the pattern is reached, the inner loop stops; if `pPos` is 0, we know that nearly the complete pattern matches and we only have to check whether the first character and its aligned character in the text are the same. If that holds, then we can return successfully as the pattern was found in the text. If there was a mismatch before or the first character did not match, then we get the shift values from the tables δ_1 and δ_2 and shift the pattern by adding the maximum of the two values to the position in the text. With that, we move the alignment of the right end of the pattern to the right and the next matching phase can start.

```
1  O-BM(text, pattern)
2    δ1 = preprocessBad(pattern);
3    δ2 = preprocessGood(pattern);
4
5    pPos = pattern.size() - 1;
6    tPos = pPos;
7
8    while (tPos < text.size())
9      pPos = pattern.size() - 1;
10     while (pPos && begin[tPos] == pattern[pPos])
11       pPos--; tPos--;
12     if (!pPos && begin[tPos] == pattern[pPos]) return true;
13     tPos += max(δ1[begin[tPos]], δ2[pPos]);
14   return false;
```

Listing 2.8: Original BM (O-BM)

2.3.3. Fast BM

In [BM77], the authors also discuss an improved version of the algorithm which we call Fast BM (F-BM). The idea behind this modification is that most of the time, the algorithm is in the situation that the character of the text with which the last pattern character is aligned mismatches and leads to a shift of the pattern, so basically the idea of the Bad Character Heuristics from Sect. 2.3.1.1.

Listing 2.9 shows how the fast version can be implemented. When preprocessing, we again generate the tables δ_1 and δ_2, but also a table δ_0. This table is a copy of δ_1, replacing the entry for the last character of the pattern with the value *large*. The value *large* must be greater than the sum of the text length and pattern length. The algorithm now adds the entry of δ_0 to the position in the text. With that, the pattern is either shifted to the right according to the BCH or the position is greater than or equal to *large*. In the case of shifting, we know that the shift is always to the right, as we are comparing the last character of the pattern. In the other case, the value of the position in the text indicates that we have found the last character of the pattern. So, we need to enter the matching phase to check the remaining pattern. We calculate the original position which is aligned with the second last character of the pattern and the corresponding position in the pattern. If necessary, we then perform the matching and shifting phase as in the original version. The only exception is to add a special handling of patterns which are only one character long because then, we can directly return if the last character of the pattern was found in the text.

```
F-BM(text, pattern)
  // large > text.size() + pattern.size() must hold for all inputs
  large = 1 << 48;
  pPos = pattern.size() - 1;
  tPos = pPos;

  δ1 = preprocessBad(pattern);
  δ2 = preprocessGood(pattern);
  δ0 = δ1;
  δ0[pattern[pattern.size() - 1]] = large;

  while (tPos < text.size())
    tPos += δ0[begin[tPos]];
    if (tPos >= large)
      tPos = tPos - large - 1;
      if (pattern.size() == 1) return true;
      else
        pPos = pattern.size() - 2;
        while (pPos && begin[tPos] == pattern[pPos])
          pPos--; tPos--;
        if (!pPos && begin[tPos] == pattern[pPos]) return true;
        tPos += max(δ1[begin[tPos]], δ2[pPos]);
  return false;
```

Listing 2.9: Fast BM (F-BM)

2.3.4. Blockwise BM

Considering the idea of the Fast BM, we can also apply the blockwise processing to search for the last character of the pattern. The Fast BM scans through the text character by character and implicitly matches the characters by adding the value *large* to the position in the text. To speed up that search, we can replace this byte-by-byte search with a blockwise search for the last character.

In this proposed algorithm, we search for the last character of the pattern using blockwise processing. Once an occurrence was found, we perform the comparison of the remaining pattern from right to left. In case of a mismatch, the pattern is shifted according to the precalculated tables. After the shift, we again search for the last character of the pattern; once another occurrence is found, we restart the comparison.

2.4. Automaton Approach

In this section, we present a new method for string pattern matching using automata. Unlike a similar approach for DNA sequence matching in molecular biology, as presented in [Per+09], our method is more versatile, as it is working on an arbitrarily large alphabet, allowing to match both ASCII and Non-ASCII characters.

On the other hand, when dealing with the more general type of patterns, the regular expressions, the Thompson algorithm can be used to translate regular expressions into a non-deterministic finite automaton (NFA) [Tho68], which can then be converted into a deterministic finite automaton (DFA) using the powerset construction. However, this can result in a large number of states in the DFA. To minimize the number of states while preserving matching capabilities, the Hopcroft algorithm can be applied [Hop71]. However, this adds additional computational complexity.

Our approach addresses the challenges of the limitations of the fixed alphabet size and the computational complexity. We present a new method for building a Like-DFA from an pattern for exact pattern matching, with this method having no limitations and being able to handle matching both ASCII and Non-ASCII characters. This approach is more versatile and can be used for pattern matching.

DFA: A deterministic finite automaton consists of

- a finite number of states Q,
- a (finite) input alphabet Σ,
- a transition function $\delta : Q \times \Sigma \to Q$,
- a start state $q_0 \in Q$, and
- a set of end states $F \subseteq Q$

Due to the transition function, each state has only one transition for each character of the alphabet.

NFA: A non-deterministic finite automaton (NFA) is a generalization of a DFA, as each state of an NFA can have 0, 1, or more transitions with the same letter from the alphabet.

- Q, Σ, q_0, and F same to DFA
- a transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ with $\mathcal{P}(Q) = 2^Q$

Upon examination of the potential patterns, their structure, and the available wildcards for LIKE expressions, it becomes apparent that the concept of LIKE expressions is a simplified version of regular expressions. With this understanding, we can utilize the idea of translating the pattern into a non-deterministic finite automaton and subsequently determinizing the automaton. This automaton can then be utilized to generate code for the given pattern and determine whether the pattern is accepted or not.

2.4.1. Non-Deterministic Finite Automaton for LIKE patterns

Given the information discussed above and the definition of NFAs, we propose a method to generate a specialized NFA specifically for LIKE patterns, referred to as the Like-NFA. To construct the automaton, we employ a straightforward approach: Initially, we create a start

state. Next, we iterate through the pattern from left to right and make decisions based on the current character:

% : add a loop to the last state accepting Σ

ESC : go to the escaped character α and add edge accepting α to the last state going to a newly added state at the end

$\alpha \in \Sigma$: add edge accepting α to the last state going to a newly added state at the end

By implementing the algorithm, we can easily convert a pattern into a Like-NFA. When storing the transitions, we distinguish between two types of transitions: the forward transition, which takes us from one state to the next one (all states have exactly one, only the last state has none), and the normal transitions, which are up to this point only the loops accepting Σ. By introducing Σ transitions, we accept all characters, even if the state has an explicit transition for this character.

Fig. 2.3 depicts the resulting Like-NFA for the pattern `%abaa%`. Similar to the percent wildcards at the begin and end of the pattern, the automaton has two loops at the start and end state. In between, the forward transitions of the states represent the pattern by connecting each other.

2.4.2. Deterministic Finite Automaton for LIKE patterns

With the current Like-NFA, we cannot immediately generate code to verify if the pattern is present in an input text. To accomplish this, we need to determinize the automaton, so, the classic approach to use the powerset construction would be straightforward. However, as the name suggests, this may result in exponential growth of states in the automaton, thus it is not an ideal solution.

Additionally, it is not necessary to process the entire automaton at once. For the prefix and suffix, we do not need to determinize the automaton, as we know the start and end of the input text and can examine them in sequence. The remaining pattern can be divided into subpatterns which begin at a state with a loop and end at the next state with a loop (essentially, a subpattern is the part of the pattern between two percent wildcards). Furthermore, it is sufficient to process the subpatterns individually, as we know that once one subpattern has been found, the following ones must appear completely after the found one, as the subpatterns must not overlap.

Determinizing Like-NFA. In our approach, we employ the concept of the longest prefix which is also a proper suffix, drawing inspiration from the KMP algorithm (as outlined in Sect.2.2), while determinizing the automaton. During this process, we traverse our states

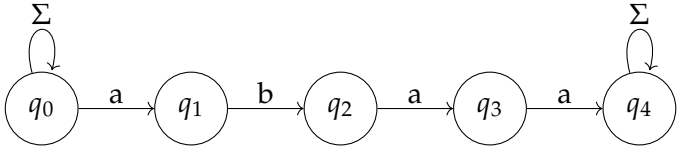


Figure 2.3.: Like-NFA for pattern `%abaa%`

sequentially, add transitions of the lps state that are not present in the current state, and update the lps when progressing to the next state. This is achieved by distinguishing between the forward transition and other transitions. In Listing 2.10, we demonstrate the concept of converting a Like-NFA into a Like-DFA for a subpattern surrounded by two percent wildcards.

The determinization needs to be done for each subpattern in the input Like-NFA. At initialization, we get the required pointers (line 2-4): the start and end state of the subpattern which is next to be processed. The pointer to the lps state is set to the start state of the subpattern and the pointer for the current state is the state following the lps state. With this initialization, we start iterating through the subpattern until the current pointer reaches the subpattern end. For each state, we add the transitions and the forward transition of the lps state to the transitions of the current state if the character of the corresponding transition is not the character of the forward transition of the current state (line 7-12). So, we achieve that if we are at a state and read a certain character from the input text, we have two cases for this character: the first case is that the character matches the forward transition character of the current state and we then take this transition to the next state; the other case is that the character does not match the forward transition character. In this situation, the lps pointer indicates in which state the longest proper prefix ends which is also proper suffix for the part of the pattern up to the current state. Based on this lps state, we can now find out which state we need to go to based on the character from the input text. For this purpose, we add non-present transitions of the lps state to the current state.

```
1  determinize(nfa)
2    [subPatStart, subPatEnd] = getNextSubpattern(nfa);
3    lps = subPatStart;
4    current = lps.next;
5    while (current != subPatEnd)
6      // handle all transitions except forwardTransition of lps
7      for (transition : lps.transitions)
8        if (transition.character == current.forwardTransition.character) continue;
9        current.transitions.insert(transition);
10     // handle forwardTransition of lps explicitly
11     if (lps.forwardTransition.character != current.forwardTransition.character)
12       current.transitions.insert(lps.forwardTransition);
13     // update lps and move to the next state
14     lps = determineNextLps(lps, current.forwardTransition.character);
15     current = current.next;
```

Listing 2.10: Algorithm to convert a Like-NFA into a Like-DFA

Determining corresponding LPS. When determinizing, we need to update the lps pointer accordingly, every time we move to the next state. Moreover, we know that we are processing sequentially by following the forward transitions, which means that all states preceding a certain state have already been processed. In Listing 2.11, we present the algorithm for updating the lps pointer. To do this, we need the lps pointer and the character of the forward transition of the current state. Based on that character, we can determine the new state for the lps pointer: Initially, we check the forward transition of the lps state if the character matches; if it does, we can return the target of this forward transition as our new lps state. Otherwise, we check the transitions of our current lps state and if it is present in there, we return the

associated state to which the transition leads. At this point, we have checked all transitions with a character, but none of them coincided with our character. As we also have Σ transitions which, unlike in the Like-NFA, consume only those characters that are not explicitly stored in transitions, we also consider that one. By construction of our algorithm, we know that our lps state has already been fully processed and that it contains a Σ transition. So, when no transition with a matching character was found previously, we return the state which is reached by the Σ transition.

```
1 determineNextLps(lps, character)
2   if (lps.forwardTransition.character == character) return lps.next;
3   // find returns either the associated pointer or NULL
4   iter = lps.transitions.find(character);
5   if (iter) return iter;
6   // get Σ transition
7   return lps.transitions.find(Σ);
```

Listing 2.11: Determine the next lps state

A step-by-step explanation for the determinization of the Like-NFA for the pattern %abaa% is shown in Fig. B.1 in the Appendix. In Fig. 2.4, we present the result of determinizing the Like-NFA from Fig. 2.3.

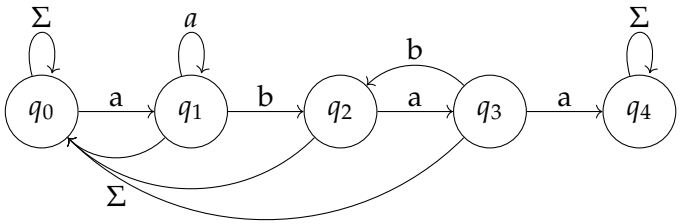


Figure 2.4.: Like-DFA for pattern %abaa%. A Σ transition consumes all characters which are not explicitly a character of any transition of the state.

Example. Let us discuss the transitions for state q_3 in the resulting Like-DFA: Once state q_3 is reached and the next character from the text to be consumed is α, then we know that before α the text has the sequence *aba*. If α is a, then we would go to state q_4 via the forward transition and accept the subpattern. If α is b, then we need to go back to state q_2, as the longest proper prefix which is also proper suffix for *aba* is a which ends in q_1. As q_1 has a forward transition with the letter b to q_2, q_3 needs to go back to q_2 if $\alpha = b$ holds. So in other words, for the sequence *abab*, the second *ab* would qualify for the first two characters of the pattern abaa and the search continues from state q_2. Lastly, if α is any other character, then we need to go back to the begin of the automaton, state q_0. In this case, the lps still ends in state q_1, but as the Σ transition leads back to q_0, we also need to go back to q_0 from state q_3 with the Σ transition. So, if we have seen the sequence $aba\alpha$ with $\alpha \neq a \wedge \alpha \neq b$, we can directly tell that the longest proper prefix which is also proper suffix is the empty string meaning that we need to start the search from state q_0 again.

With the requirements and the limitations of our Like-NFA and the determinization algorithm explained above, we can build a Like-DFA without growing in size.

3. Code Generation for Exact String Pattern Matching

Code generating database engines allow to think of a new approach of performing exact LIKE pattern matching. The concept behind the code generating pattern matching approach is to get a compact representation of the pattern and any necessary tables, such as transition tables or tables from preprocessing functions. By doing this, we aim to eliminate any additional overhead from recomputing or accessing information that is known at query compilation time.

This chapter discusses the process of generating code for the algorithms outlined in Chapter 2 within the database system Umbra [NF20; KLN21]. Umbra is an evolution of the pure main memory database system HyPer [KN11] towards an SSD-based system and translates queries into low-level code [Neu11]. The current method for pattern matching in Umbra involves storing the pattern in the data section of the generated program and calling a matching function written in C++ handing over a pointer to the stored pattern and the input text. This function then performs the matching in an interpreting way and returns whether the pattern was found or not. In the following sections, we will discuss how to generate code to search for a pattern surrounded by two % wildcards.

In Sect. 3.1, we give a short summary of the code generation framework in Umra and its specifics. Sect. 3.2 covers the code generation for the different implementations of the KMP algorithm, Sect. 3.3 explains the BM algorithm, and Sect. 3.4 the automaton approach. To cover multiple subpatterns, Sect. 3.5 explains how a pattern composed of multiple subpatterns is translated into one big block performing the pattern matching.

3.1. Code Generation Framework in Umbra

In [KLN21], Kersten et al. describe the steps for building a compiling query execution engine, which is also implemented in the database system Umbra: first, they introduce a code generation framework which simplifies core concepts like control flow instructions to reduce complexity and generates code in one pass; second, they present a program representation whose data structures are designed to generate and compile code rapidly; third, they propose a new compiler backend that is designed for minimal compile time and has exceptional performance compared to other methods such as the Volcano-style or bytecode interpretation.

For our implementations of the pattern matching algorithms, we mainly access the **Tuples**, **SQL Values**, and **Codegen API** layer. Umbra generates code in static single assignment (SSA) form; with that form, an additional compiler pass can be saved when compiling the query.

In Umbra, the generated code is organized in basic blocks; during code generation, there is always one current block to which new instructions append. While writing the pattern specific code, we handle the basic blocks manually, to which the instruction needs to be appended .

Moreover, as the code is in SSA form, we need to use the PHI nodes as a substitute for

multiple variable assignment. A PHI node is an instruction at the start of a basic block with multiple arguments. Based on which basic block was executed before the current block with the PHI node, it chooses one of its associated values from the PHI node arguments.

In the following sections, we present the conceptual control flow of the algorithms without going into the specifics of code generation in Umbra in detail.

3.2. Knuth-Morris-Pratt Algorithm

In this section, we will examine the algorithms discussed in Sect. 2.2 and explain how to include optimizations to improve performance. The first optimization, the early return, is already included in every implementation of the algorithm.

3.2.1. Original KMP

As presented in Listing 2.2, there are multiple accesses to the pattern and, in case of a shift, also to the lps table of the pattern. When translating the algorithm into code, we first analyze which parts of the algorithm can be inlined: the pattern itself, the lps table of the pattern, and the length of the pattern. The structure of the generated code can mostly be transferred from the original algorithm by only adding mechanisms to deal with the SSA format of the code generation in Umbra. In Fig. 3.1, we present the overall structure of the generated code for the Original KMP algorithm; the darkgreen comments represent the places at which the pattern, lps table, and pattern length need to be inlined, the green arrows are taken in case the condition evaluates to true, the red arrows otherwise. During the following explanation of the control flow, we sometimes refer to the original code from Listing 2.2 by putting the corresponding line in brackets.

The entry point for the search is the block with the label `whileLoopHeader`. This block represents the loop over the input text length and checks whether the remaining length of the input text is still sufficient (similar to line 5), otherwise directly returns false. As in further steps the character at the position `pPos` is required, we look up the current character in the `determinePatternChar` block. Next, we check the condition of the inner loop in the original algorithm (see line 6): If the condition holds, we need to update `pPos`. For this update, we enter the `determineLPSTableEntry` block to look up the lps entry for the given position in the pattern. When inlining the lps table here, we can leave out the entry at index 0, as, due to the check before, `pPos` cannot be 0 at this point. After reading and setting the new `pPos`, we go back to determine the character at the current position in the pattern and check and optionally execute the inner loop again. If the condition of the inner loop fails, we check if the current pattern character and text character match (see line 8). This can be done as we have looked up the character from the pattern in the `determinePatternChar` block before. Following a successful character check, we check if the pattern end is already reached and, if yes, return true (see line 9), otherwise increment the position in the pattern. After all, we branch to the block `whileLoopIteration` to increment the position in the text and then go back to the `whileLoopHeader` block to continue with the search.

By using this approach to generate code for the Original KMP algorithm, we see that no additional access to the pattern or the lps table of the pattern is necessary as any relevant information is put into the generated code.

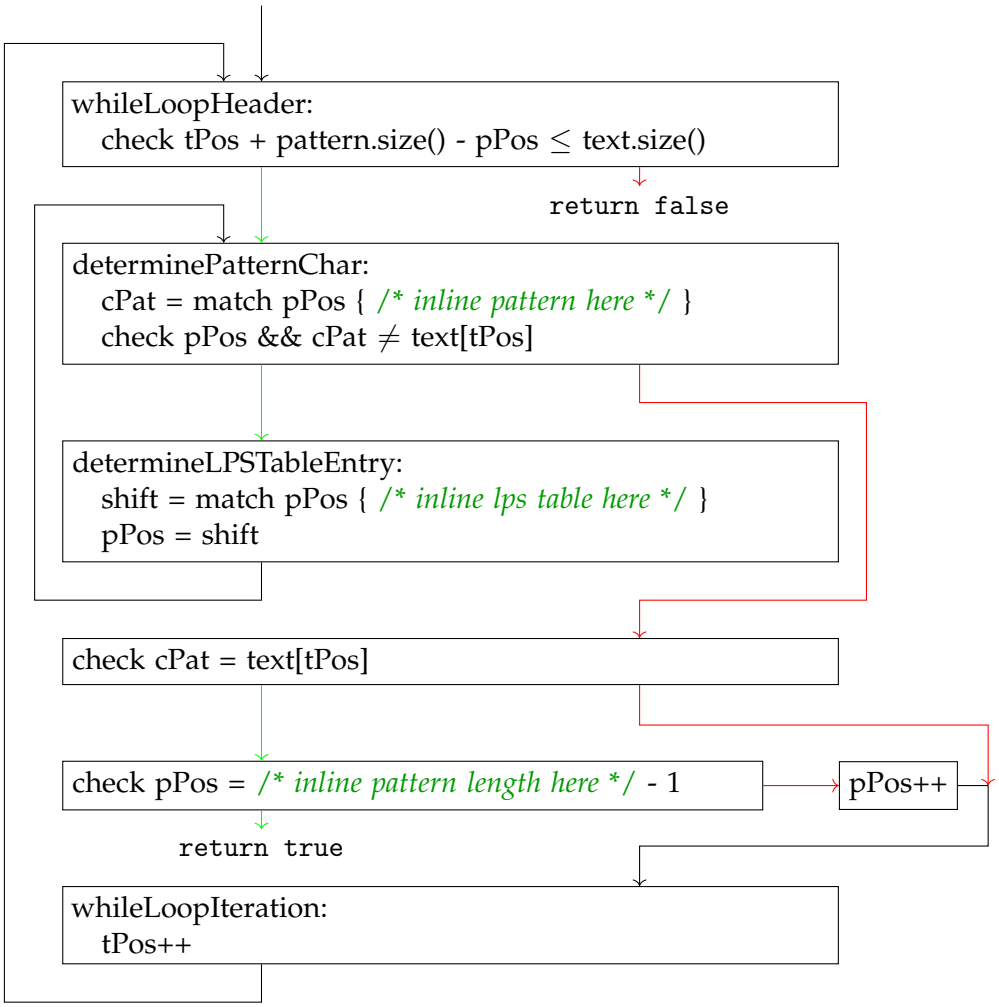


Figure 3.1.: General control flow of the generated code for the Original KMP algorithm. A green arrow is taken if the previous check evaluates to true, the red arrow otherwise.

3.2.2. KMP with One Loop

As in the Original KMP algorithm, the modified implementation also requires the pattern and the corresponding lps table to be inlined into the generated code. In this approach, we follow the structure of the code from Listing 2.3. However, we change the control flow in such a way that we do not have to iterate multiple times to match the pattern against the input text, but write out the comparisons between pattern and input text sequentially.

To start, we enter the block `whileLoopHeader` to check if there are still characters from the input text to read. In that case, we need the condition of the early return, as we have to make sure that the input text is not exceeded by the pattern; otherwise, we would access parts of the memory not belonging to the input text. If there are not enough characters left, the algorithm returns false; otherwise, the control flow starts to perform the comparison of the characters starting at the current position in the pattern. To do that, the generated code lists the possible indexes of the pattern covering the interval $[0, \ldots, \text{len}(\text{pattern})[$ and checks if the current

position matches this index. By design of the algorithm, we know that the current position in the pattern and one of the listed indexes needs to match. Once this index is found, the control flow goes to the corresponding check of the pattern character at this index which can also be inlined into the generated code and verifies if the character of the pattern and the input text coincide. If the characters are the same, the control flow moves on to the next character in both pattern and text and checks these again; once all characters have been checked, i.e. no mismatching characters could be found, the function can return that the pattern was found in the text. If two characters are not the same, the matching process stops and the control flow goes to the `performShift` block. In this block, we determine based on the position in the pattern which shift needs to be performed by looking up in the inlined lps table. This shift is then used to set the position in both pattern and text for the next search and the control flow

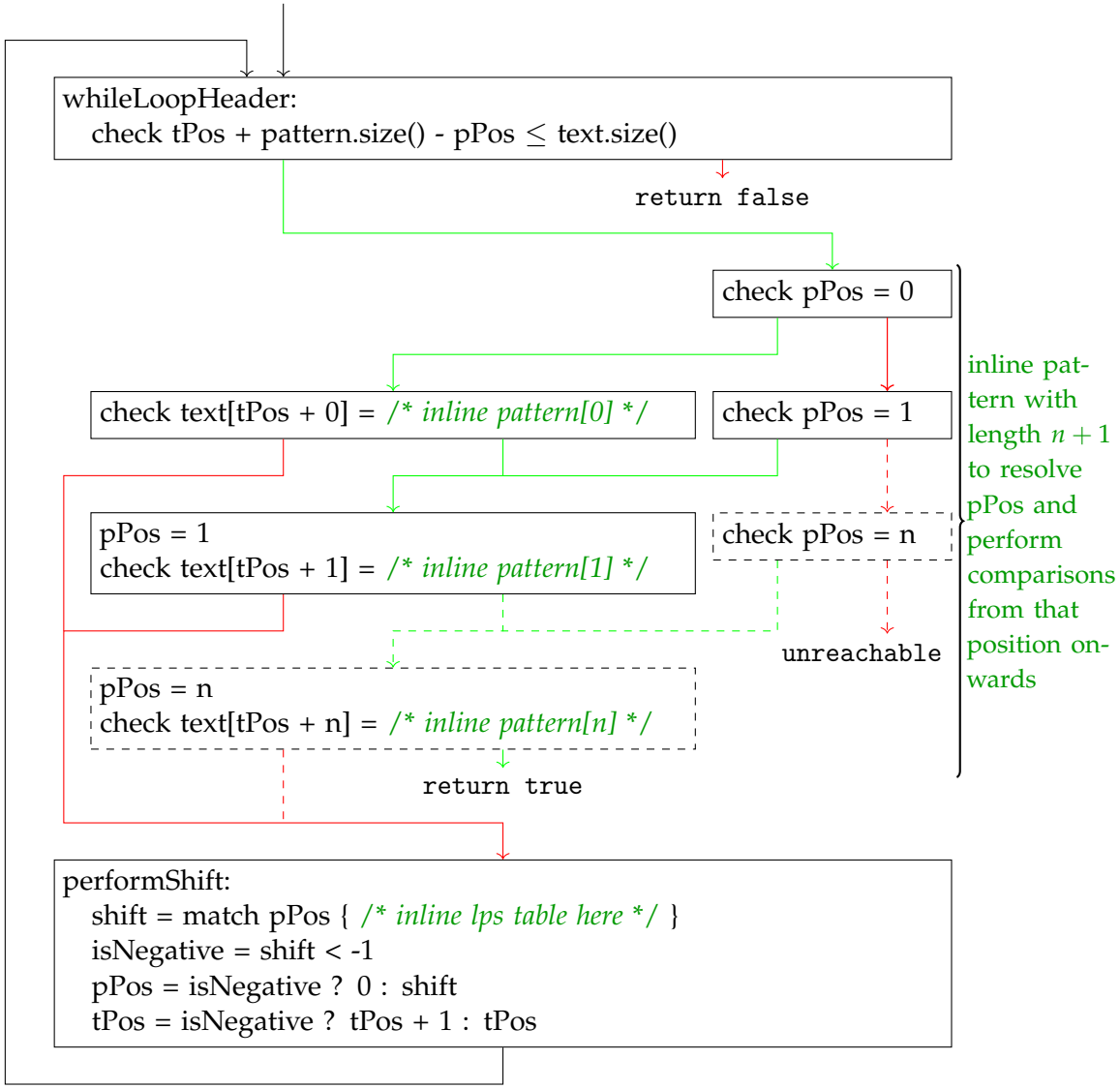


Figure 3.2.: General control flow of the generated code for the KMP algorithm with one loop. A green arrow is taken if the previous check evaluates to true, the red arrow otherwise.

goes to the `whileLoopHeader` block to restart the search.

3.2.3. Adding Optimizations to Code Generation Process

As discussed in Sect. 2.2.4, we have three different optimizations for the KMP algorithms. The first optimization, the early return, is already applied to all implementations.

Another optimization, the compression of the lps table, primarily affects the preprocessing function. The Original KMP algorithm needs to be modified slightly, as now also negative shifts can be reached at indexes other than 0; however, the KMP algorithm with one loop does not need any changes, as it can already handle negative shifts. A negative shift indicates that the pattern can be reset to its start and the search can begin at the character to the right of the current character in the input text.

The third optimization is blockwise processing. The idea is to find a given character in a larger block using bitwise operations, instead of iterating through the characters in the block. This optimization can be applied when searching for the first character of the pattern: As we assume that the alphabet is reasonably big and the occurrence ratio of the character is low, it is likely that the first character will not match. In this case, the KMP algorithm would only shift by 1, resulting in searching for the character one by one through the text. To avoid this, blockwise processing can be applied to search for the first character of the pattern when the algorithm would just iterate one by one. Once found, the matching for the rest of the pattern can start from the found position onwards.

The original control flow from Fig. 3.2 is just extended by the block `blockwiseProcessing`, which performs the blockwise search for the first character of the pattern. Fig. 3.3 visualizes this new control flow, leaving out the part of inlining the pattern which is the same as in the control flow of Fig. 3.2.

The basic idea behind blockwise processing is to shift to a position in the text where the first character coincides with its aligned text character. This allows the matching to continue from the second character. The process then proceeds as discussed in the unmodified version. The only exception is when a shift occurs due to two mismatching characters: In this case, a negative shift would indicate that the pattern should be moved by one to restart the search from the beginning. Instead, blockwise processing is applied again to search for the next occurrence of the first character of the pattern in the text and start the matching process from this position. If the shift is not negative, the positions are updated and the algorithm proceeds in the `whileLoopHeader` block with performing a normal matching phase.

In the blockwise processing, we insert a check if it is safe to read the next 8 bytes from the input text. If it is, we read these bytes and perform the blockwise processing on them; if not, we skip the phase and leave the step-by-step processing to the original algorithm.

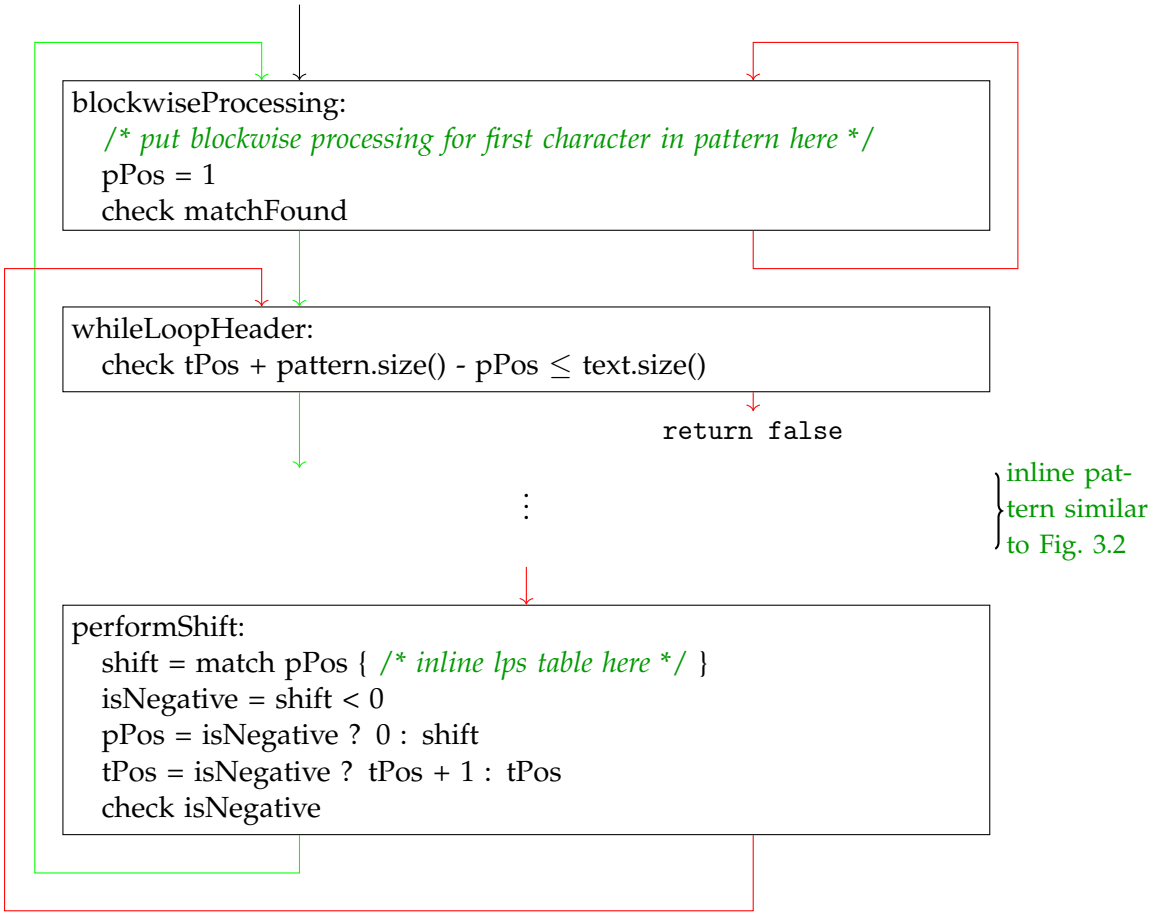


Figure 3.3.: Modified control flow of the generated code for the KMP algorithm with one loop to include optimizations. A green arrow is taken if the previous check evaluates to true, the red arrow otherwise.

3.3. Boyer-Moore Algorithm

This section covers the code generation for the three versions of the Boyer-Moore algorithm discussed in Sect. 2.3.

3.3.1. Original BM

Similar to the KMP algorithm, the original version of the Boyer-Moore algorithm has accesses to the pattern and the precomputated tables, which are required to determine the correct shift in case of a mismatch. To generate the code for this pattern matching algorithm, we again keep its structure, but write out the comparisons of the pattern characters with the input characters from right to left sequentially into the code. Fig. 3.4 shows the control flow for the Original BM algorithm.

The block `whileLoopHeader` represents the loop over the input text, checking if the input text

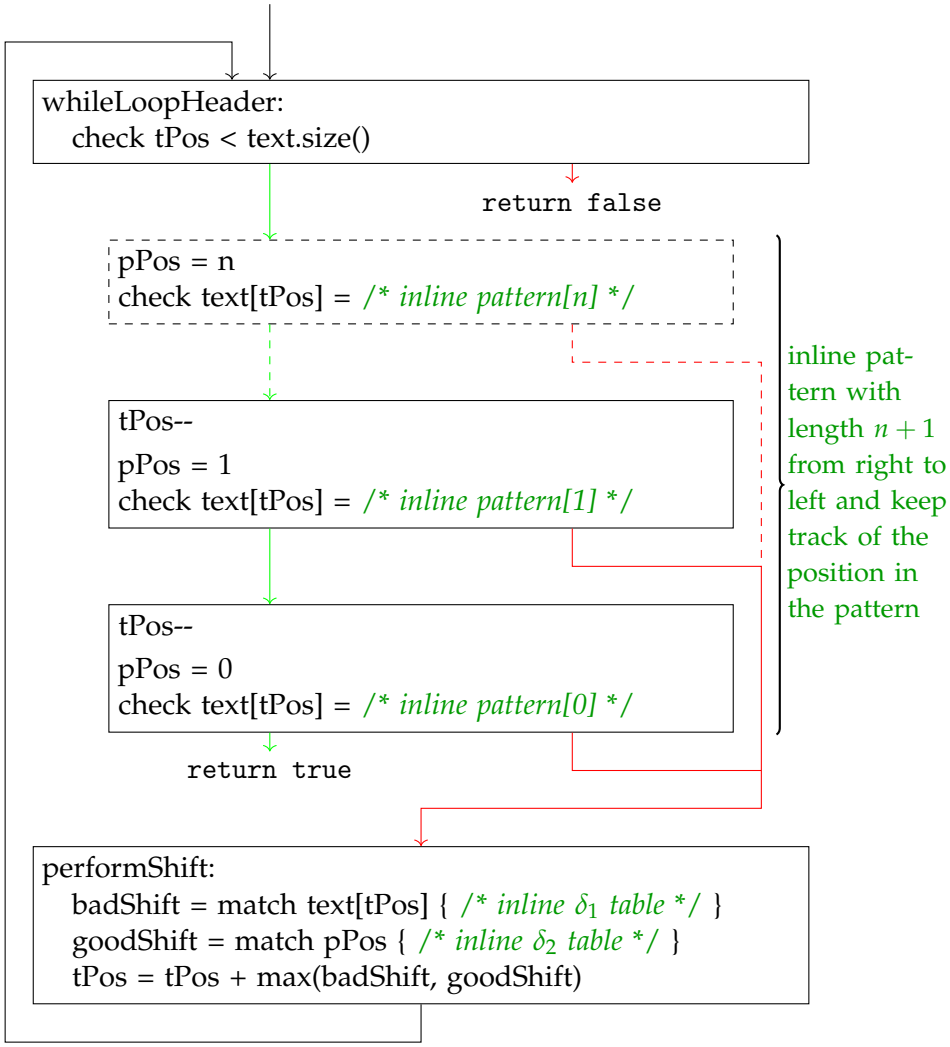


Figure 3.4.: General control flow of the generated code for the Original BM. A green arrow is taken if the previous check evaluates to true, the red arrow otherwise.

is exhausted. In this case, we can return false; otherwise, we start comparing the pattern from right to left. These comparisons are listed in the generated code by inlining the corresponding character at this position. After each successful character check, we go to the block for the next character. In order to read the correct character in the text, we also need to decrement the position in the text in this block. If all characters match, the pattern was found in the text and we can return successfully. In case of mismatching characters, we need to determine the shift for that position. For this, we have the block `performShift` in which the tables δ_1 and δ_2 are inlined. Based on those tables and the information about the position at which the mismatch occurred, the relevant shift can be found. Finally, we add the maximum of the two shifts to the position in the text and go back to the `whileLoopHeader` to continue the search.

When inlining the table δ_1, we do not inline the whole table, but only the entries whose value differs from the pattern length and use that value as a default value for all other characters. Looking up the value in the table can lead to two situations: if the character is in the table, then the corresponding value is used; if not, the default value, the pattern length, is used.

3.3.2. Fast BM

As presented in Fig. 3.5, the control flow of the Original BM algorithm can easily be extended to the fast version presented in [BM77].

The key idea of this algorithm is the table δ_0. This table can also be inlined completely into code by only storing the values for the characters that appear in the pattern and putting the pattern length as a default value for all characters not in the pattern. By reading the value from that table, it is used to detect if the current text character matches the last character of the pattern. If not, the pattern is automatically shifted to the right by adding the value of the δ_0 table and the search can continue; in case of a match, the comparison itself is started from the second last character of the pattern towards the left. For this, we only need to inline the pattern starting from the second last character to the begin of the pattern; the last character of the pattern was already matched indirectly by adding the value from δ_0. However, before starting the comparison from the second last character, we need to recalculate the position in the text to be correctly aligned with the second last character in the pattern. To do that, we only subtract the value *large*, which was added to the position when getting the shift from δ_0. With that, we get the position in the text with which the last character of the pattern is aligned. The comparison of the pattern then starts at the second last character of the pattern, as the last character was already matched implicitly. Once all characters were compared successfully, the algorithm can return true, otherwise, the `performShift` block is entered in which we determine the shift based on δ_1 and δ_2, similar to the original algorithm. After applying this shift, we go back to the `whileLoopHeader` to continue with the search.

3.3.3. Blockwise BM

In Sect. 2.3.4, we present an adaption of the Fast BM algorithm to replace the byte-by-byte search for the last character of the pattern with blockwise processing. With that idea, we want to discuss now how to modify the Fast BM algorithm to include the blockwise processing: When generating code for a pattern, we introduce a `blockwiseProcessing` block which handles the functionality to perform blockwise search for a character, as discussed in Sect. 2.2.4.3. In this basic block, we process a complete block as long as possible before

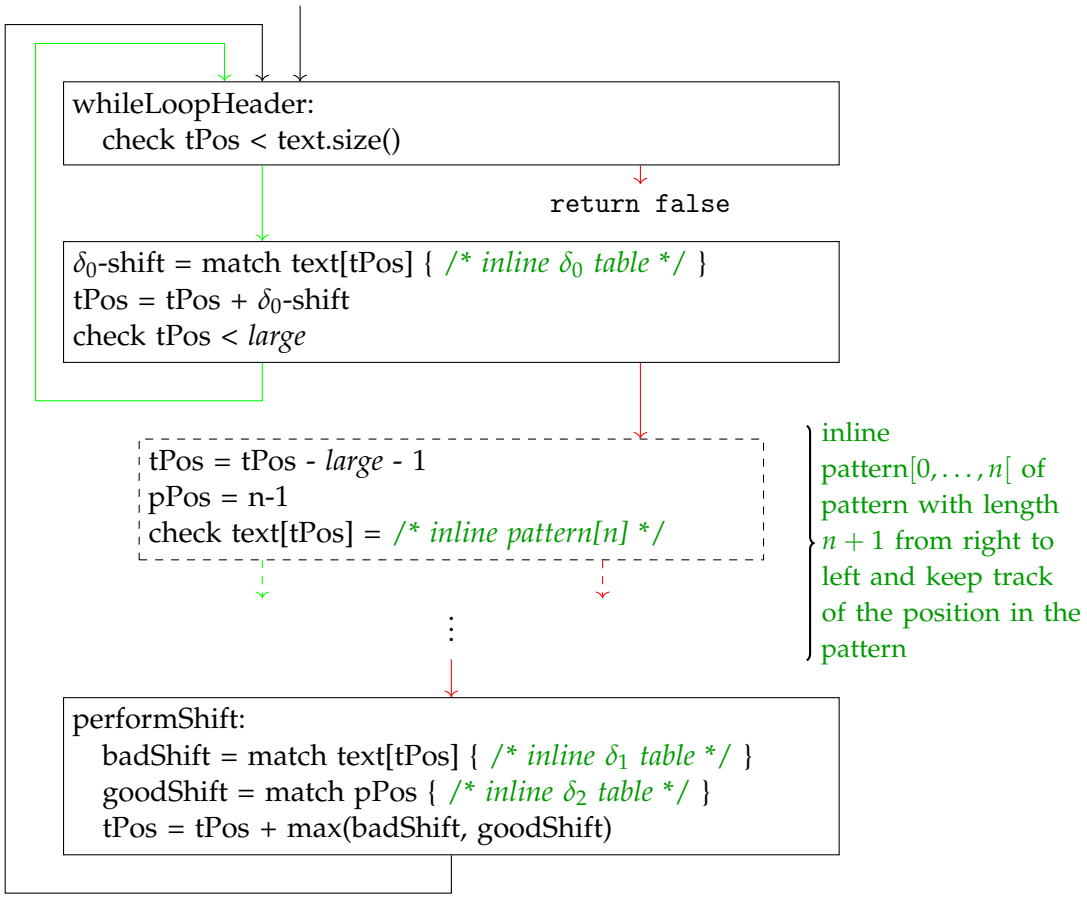


Figure 3.5.: General control flow of the generated code for the Fast BM. A green arrow is taken if the previous check evaluates to true, the red arrow otherwise.

switching to byte-by-byte processing until the input text is exhausted. By applying these restrictions, we make sure that once the `blockwiseProcessing` block is left, the pattern is either aligned with a character matching the last one or the input text is exceeded and the search can terminate. Afterwards, we can directly start the remaining comparison as presented in Fig. 3.5 for the Fast BM implementation. In case of a shift, we get the shift values from the δ_1 and δ_2 table, take the maximum, and then go to the `blockwiseProcessing` block to search for the last character from the new position in the input text on.

By using this modified approach, we can get rid of the inlining of the δ_0 table, as well as the checks whether the position in the text is greater than or equal to the value *large*.

3.4. Automaton Approach

In Sect. 2.4, we have discussed our approach to generate an automaton from our pattern and to determinize it to get a Like-DFA. Moreover, we have already presented how to handle the prefix and the suffix of the pattern without the requirement to build a full automaton for those parts. In this section, we present two ideas how to translate the Like-DFA into generated code, the direct approach in Sect. 3.4.1 and the approach to apply the blockwise search functionality in Sect. 3.4.2.

3.4.1. Direct Translation

Our direct approach is to translate the automaton directly into code. This means that we iterate through the states of our automaton and generate code for each one of them. Inside every generated code block, we first check if the text is exhausted or not. After that, we generate code to get the next character of the input text. For this character, it is then checked if it fits any transition going out from the state. For this, we generate a check for each transition of the current state. However, due to the design of our transitions, we first only write out the transitions including the forward transition which do consume a specific character and place the Σ transition as the default handler at the end. When the character was found or the default transition is taken, we increment the position in the text and jump to the code that belongs to the state of this transition.

In Listing 3.1, we present the generated code for the automaton in Fig. 2.4 for the pattern `%abaa%`.

```
1  q0: if (tPos >= text.size()) { return false; }
2      c = text[tPos];
3      if (c == 'a') { tPos++; goto q1; }
4      else { tPos += Utf8::length(c); goto q0; }
5  q1: if (tPos >= text.size()) { return false; }
6      c = text[tPos];
7      if (c == 'a') { tPos++; goto q1; }
8      else if (c == 'b') { tPos++; goto q2; }
9      else { tPos += Utf8::length(c); goto q0; }
10 q2: if (tPos >= text.size()) { return false; }
11     c = text[tPos];
12     if (c == 'a') { tPos++; goto q3; }
13     else { tPos += Utf8::length(c); goto q0; }
14 q3: if (tPos >= text.size()) { return false; }
15     c = text[tPos];
16     if (c == 'a') { tPos++; goto q4; }
17     else if (c == 'b') { tPos++; goto q2; }
18     else { tPos += Utf8::length(c); goto q0; }
19 q4: return true;
```

Listing 3.1: Generated code for automaton in Fig. 2.4 for the pattern `%abaa%`

When translating the Σ transition, we need to consider that our input text might contain UTF8-encoded characters, so when taking this transition, we need to skip the number of bytes belonging to that character. This is abstracted by the function call `Utf8::length(...)`. When generating the code itself, we replace these function calls with generated code to determine

the number of characters to be skipped: for ASCII characters, we just return the length 1; for Non-ASCII characters, we return the number of leading 1's by using an assembly instruction which performs that counting.

3.4.2. Blockwise Translation

Considering the build algorithm and construction of the Like-NFA and Like-DFA, we see that for each start state of a subpattern, i.e. a state with a Σ transition to itself, we scan through the input text until reaching a character that matches the character of the forward transition. As we have already discussed in Sect. 2.2.4.3, this byte-by-byte search can be replaced with the blockwise functionality to directly search for a specific character in a block. While generating code for our Like-DFA, we now only have to replace the part when handling the start state of a subpattern; the process for the remaining states of the subpattern can be kept the same. For the start state of a subpattern, we generate code which searches for the character of the forward transition in a blockwise manner as long as possible and only changes to byte-by-byte processing when the remaining text length is too short. Once the character of the forward transition was found, the control flow goes to the second block, where the processing continues as in the direct approach.

Listing 3.2 shows the generated code for the automaton from Fig. 2.4 with the blockwise processing at the section start. This extension is highlighted in the code with a gray background. The function `blockwiseSearch(...)` takes the input text, the position in the text where the search should start, and the searched character. Moreover, it also encapsulates the functionality to process the input text blockwise as long as possible and to change to the byte-by-byte search, once the length is not sufficient anymore; it also stops if the input is exhausted. It returns the index of the searched character in the input text.

```
1  q0: tPos = blockwiseSearch(text, tPos, 'a');
2      if (tPos >= text.size()) { return false; }
3      tPos++; goto q1;
4  q1: if (tPos >= text.size()) { return false; }
5      c = text[tPos];
6      if (c == 'a') { tPos++; goto q1; }
7      else if (c == 'b') { tPos++; goto q2; }
8      else { tPos += Utf8::length(c); goto q0; }
9  q2: if (tPos >= text.size()) { return false; }
10     c = text[tPos];
11     if (c == 'a') { tPos++; goto q3; }
12     else { tPos += Utf8::length(c); goto q0; }
13 q3: if (tPos >= text.size()) { return false; }
14     c = text[tPos];
15     if (c == 'a') { tPos++; goto q4; }
16     else if (c == 'b') { tPos++; goto q2; }
17     else { tPos += Utf8::length(c); goto q0; }
18 q4: return true;
```

Listing 3.2: Generated blockwise code for automaton in Fig. 2.4 for the pattern `%abaa%`

3.5. Concatenating Multiple Subpatterns

Until now, we have only discussed the concept of generating code for one pattern surrounded by two % wildcard. In this section, we want to explain how to conceptually generate code for the pattern $\alpha\%\beta\%\gamma\%\delta$ with α, β, γ, and δ representing any sequence of characters without a _ wildcard.

In Fig. 3.6, we show a visualization how the generated code per subpattern, grouped into logical blocks, is connected. For the pattern above, we start with writing out a linear comparison for the subpattern α from left to right, as we directly can compare these characters with the same number of the first ones from the input text. Then, we continue with generating code for the subpatterns β and γ as presented in the sections before. However, we do not generate a return from each of the matching phases like in the visualized control flows. Instead, once we accept the subpattern β, we go to the entry block for matching γ and pass on the index at which the previous pattern ended. With that, we can start the search for γ. Once only the end δ is left, we write out a direct comparison for this sequence from right to left with the end of the input text. If all character match, we only need to check the indexes at which γ ends and δ starts to avoid that the patterns overlap. Once all blocks were passed successfully, we can accept the input text; otherwise, we need to reject it.

With this approach, we can deal with any kind of pattern, regardless of which parts of the pattern are present.

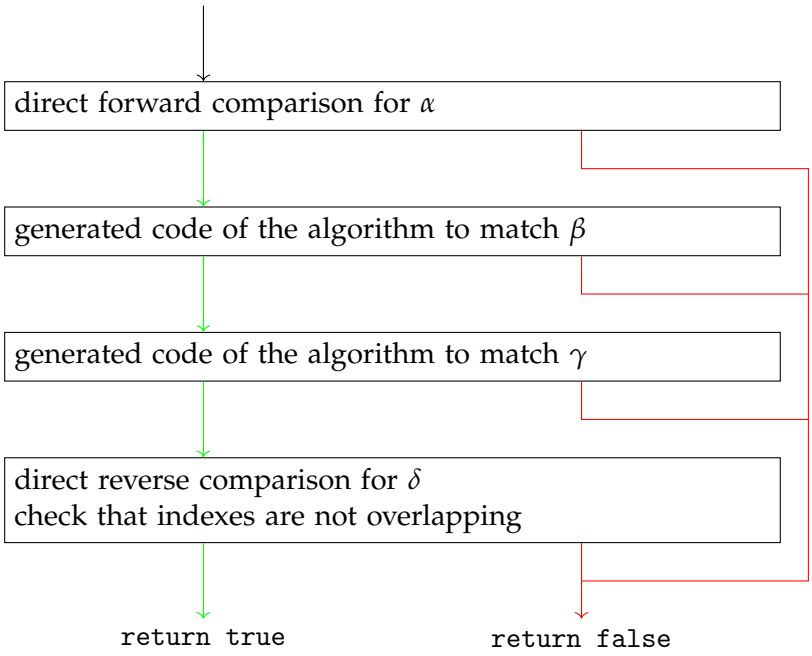


Figure 3.6.: Conceptual control flow of the generated code for the pattern $\alpha\%\beta\%\gamma\%\delta$. Each node encapsulates the whole functionality noted down in it. A green arrow is taken if the previous check or algorithm evaluates to true, the red arrow otherwise.

4. Evaluation for Exact String Pattern Matching

In this chapter, we present the results of the experiments evaluating the performance of code generation for pattern matching. Our findings indicate that the use of code generation leads to a significant increase in throughput compared to an interpreting approach. In particular, we observed that for certain queries, the throughput of code generation was almost twice that of the interpreting method.

4.1. Experimental Setup

The section outlines the environment and settings of our experiments. In Sect. 4.1.1, we list the hardware information of the server which we used to run the experiments. Sect. 4.1.2 gives an overview over the used datasets and queries; Sect. 4.1.3 explains the settings of Umbra to execute the queries.

4.1.1. Hardware Specification

For the experiments, a single core (one NUMA region) of a dual-socket machine running two Intel Xeon E5-2680 processors at 2.40 GHz has been used. The machine has in total 256 GB of DDR4-RAM running at 2400 MHz. The OS is a 64-bit Ubuntu 22.04.1 LTS with a Linux 5.15.0-56-generic kernel.

Umbra is implemented in C++ and compiled with g++-12.1.0. The Makefile is written and maintained manually and compiles Umbra with optimization level `O3`. Optionally, one can also use CMake (at least cmake-3.21 required).

4.1.2. Data and Queries

To evaluate the performance of code generation for pattern matching, we conducted experiments on two datasets: the TPC-H benchmark dataset and the ClickBench dataset. The next two sections provide an overview of the datasets and present the queries that were used for our measurements.

4.1.2.1. TPC-H Data

To measure the single-threaded performance of our implementation, we use the TPC-H schema[1] and its data with the scale factor 1. In Fig. 4.1, we show the distribution of all characters which occur in the input texts in the TPC-H scheme. Based on the TPC-H scheme and its example query 9, we choose to use the following queries as workloads for our measurements:

[1] `https://www.tpc.org/tpch/`

short: The pattern length is less than 25% of the average input text length:
`select count(*) from part where p_name LIKE '%spring%';`
10825 tuples fulfill the condition.

medium: The pattern length is between 25% and 50% of the average input text length:
`select count(*) from part where p_name LIKE '%medium spring%';`
96 tuples fulfill the condition.

long: The pattern length is more than 50% of the average input text length:
`select count(*) from part where p_name LIKE '%midnight medium spring%';`
2 tuples fulfill the condition.

multiple: The pattern is composed of multiple subpatterns:
`select count(*) from part where p_name LIKE '%midnight%medium%spring%';`
4 tuples fulfill the condition.

As the names of the queries suggest, we intend to cover four different types of patterns which might occur frequently in real-world queries. The table `part` of the TPC-H scheme has 200k tuples, the length of text input of the attribute `p_name` is on average 32.75 characters.

4.1.2.2. ClickBench

In order to check our implementations on a more realistic dataset, we decided to use ClickBench[2], a benchmark which represents typical workload in areas such as clickstream and traffic analysis, web analytics, machine-generated data, structured logs, and events data. It includes typical 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, and while it is anonymized, it retains the essential distributions of the data. The queries are designed to reflect realistic workloads.

For our experiments, we used queries 20, 21, 22, and 23 from the ClickBench benchmark. The table used for the benchmark contains 99,997,497 records, and the average length of the input text is 89 characters. The dataset contains both ASCII and Non-ASCII characters, the distribution of the occurring bytes in the input text can be seen in the Tables C.1 and C.2 in the Appendix

4.1.3. Query Settings

For our implementation, we extended the functionality in Umbra to match LIKE expressions, allowing for dynamic configuration changes at runtime. To measure performance, we utilized the built-in measurement functionality provided by Umbra when running a query. The TPC-H schema queries were executed using a single thread and the process of compilation and execution was repeated 1000 times. For the ClickBench queries, we ran them using one thread, eight threads, and 20 threads, and repeated each query 20 times. Additionally, we compiled the queries using both the **JITBackend**, which employs LLVM for code generation, and the **DDCGBackend**, which generates machine code directly.

[2] `https://benchmark.clickhouse.com`

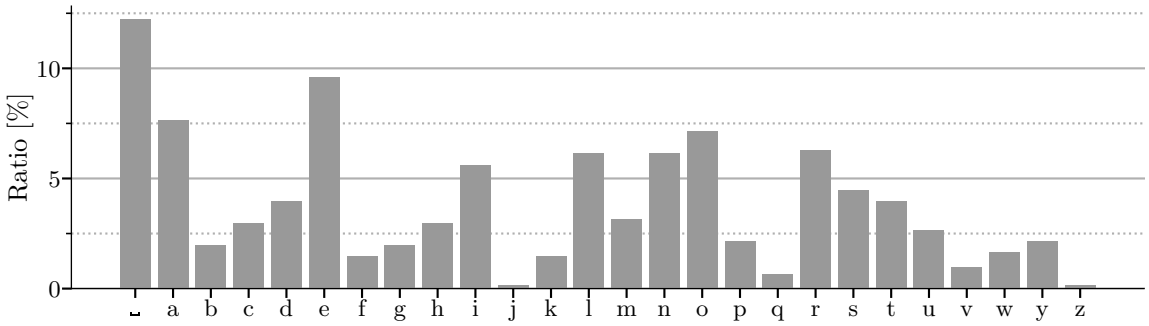


Figure 4.1.: Distribution of the ASCII characters in the input texts of the TPC-H scheme

4.2. Results

In this section, we present the results of our experiments. For each algorithm that has both an interpreting and code generating version, we present the results side-by-side, with the left bar (hatched) representing the interpreting version, which involves calling the corresponding C++ function for the matching process, and the right bar showing the throughput for the query when generating code for the matching process.

4.2.1. Knuth-Morris-Pratt Algorithm

As already discussed in Sect. 2.2 and 3.2, we have presented two different versions to implement the KMP algorithm and three additional optimization ideas. The idea of the early return is directly applied to all algorithms, so, they use the revisited condition when looping over the input text.

4.2.1.1. Regular LPS Table

Fig. 4.2 presents the throughputs, we could achieve on the dataset with the queries discussed above. On the left side of the dashed vertical line of each graph, there is the Original KMP algorithm; on the right side, the KMP algorithm with One Loop and its blockwise modification are shown.

Short pattern. When using the **JITBackend**, we see that the code generating version of each algorithm has a higher performance than the interpreting implementation.

Looking at the KMP with One Loop (OL-KMP) and its variants, we see the benefit of the code generating approach over the interpreting one. Although the interpreting approach is slower than the O-KMP, the code generating version benefits from the easy approach without the need to handle two interleaved loops as it is necessary in the O-KMP. The code generating OL-KMP also benefits from writing out the pattern completely into the code, unlike the O-KMP which includes the pattern by writing it in a PHI node.

We also notice that the algorithm using the blockwise processing optimization (BOL-KMP) has a much higher throughput than the ones without. This optimization allows us to reduce the number of cycles needed by processing eight bytes at a time instead of iterating through the input text one byte at a time. Additionally, we experience fewer cache accesses to the

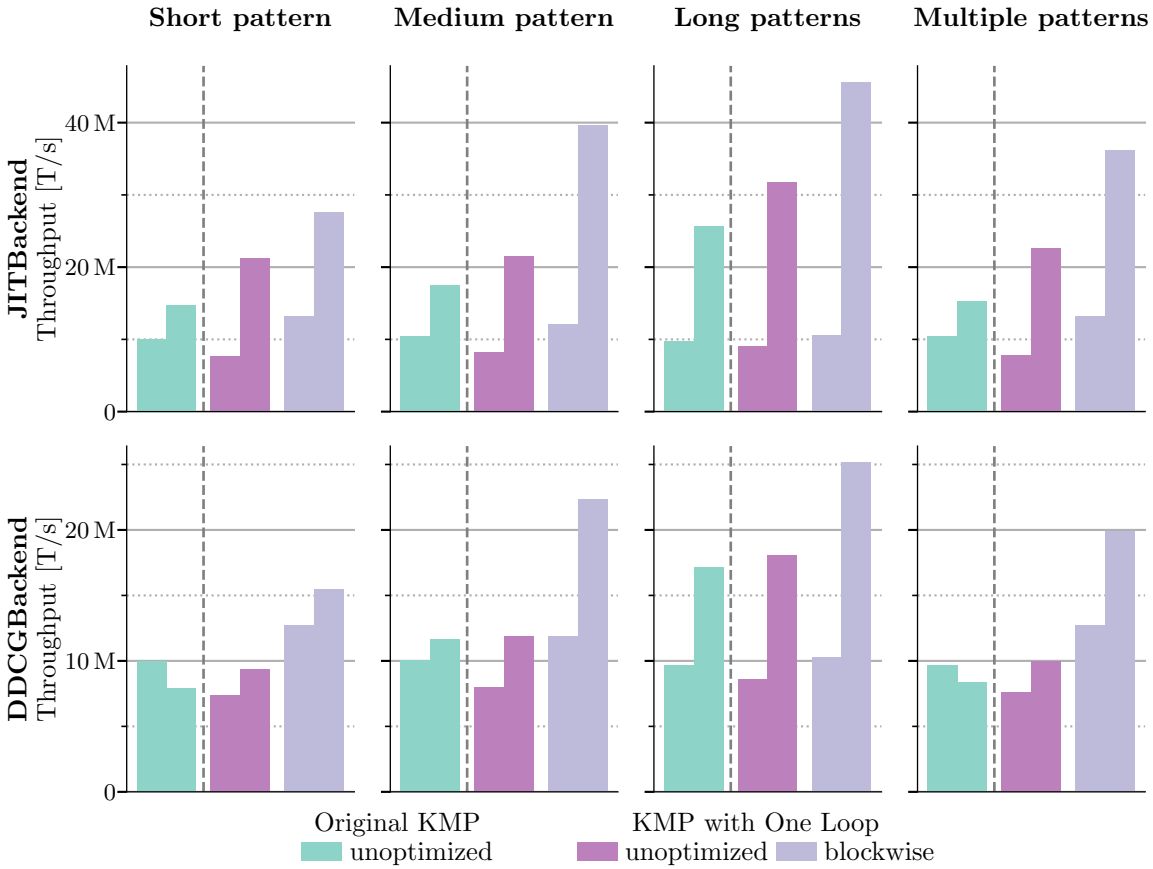


Figure 4.2.: Throughput for the different workloads of the KMP algorithms. For each algorithm, the left (hatched) bar presents the throughput of the interpreting version, the right one is the code generating version. At the left of the dashed vertical line, the Original KMP (Sect. 2.2.2 and 3.2.1) is shown with its optimization; on the right, the KMP with One Loop (Sect. 2.2.3 and 3.2.2) and its optimized algorithms are shown. (Higher is better.)

input text compared to the unoptimized versions. The blockwise optimization improves the performance of both the interpreting and code generating versions.

When using the **DDCGBackend**, we observed similar performance in terms of throughput for both the interpreting and code generating versions. However, for the O-KMP algorithm, the code generating version had a lower throughput than the interpreted version. Like the other backend, this is due to the lack of instruction reordering when compiling the generated code as it happens for the interpreting code when Umbra is compiled. Additionally, the backend that produces the generated code and converts it to machine code handles PHI nodes, which are necessary for loops in SSA form, by storing values in memory, resulting in increased memory accesses. In the O-KMP implementation, there is an outer loop and an inner loop, which requires more PHI nodes and results in more memory loads and stores.

Similar to the JITBackend, the blockwise processing results in an increase in throughput for both the interpreting and code generating algorithms. The throughput of the generated code is again higher than its corresponding interpreting version.

Medium pattern. For the medium pattern, we observe that the throughput of the code generating implementations begins to improve compared to the short pattern. The throughput of the interpreting approaches only vary slightly in comparison.

In the **JITBackend**, the performance of the BOL-KMP improves significantly when compared to the other algorithm. However, it is important to note that the blockwise search used the character `'m'` which has a lower ratio than the `'s'` searched for in the short pattern workload. Additionally, we can see a performance improvement for the 0-KMP and OL-KMP algorithms, although it is less significant.

In the **DDCGBackend**, we also see that the throughput of the code generating version of the O-KMP is now higher than its interpreting version. This is because the medium pattern is longer and thus, the early return can be applied sooner than for the short pattern. This allows for more input texts to be rejected, resulting in an overall performance improvement.

Long pattern. When it comes to the long pattern, the benefit of code generation over interpreting algorithms is clear for both backends. The main advantage of code generating algorithms is that the lps table is calculated only once at code generation time, and is then inlined into the code.

In contrast, interpreting algorithms require the lps table to be generated every time a tuple is processed, which reduces the throughput of the interpreting algorithms as the pattern length increases. However, the additional preprocessing time for long patterns is partially offset by the early return optimization: longer patterns can be rejected faster if there is no match, so more tuples can be processed in the same time interval.

The performance gained by the early return optimization can be seen when comparing the throughput of the code generating versions of the short and long pattern: the long pattern leads to more input texts to be rejected due to the early return, resulting in a significantly higher throughput for the long pattern than the short one. In the code generating version, both the pattern and the lps table are completely inlined into the code, so we do not have any additional memory accesses to the pattern or the lps table. The only issue could be the limit of the code generation framework and the resulting size of the generated program if the pattern is too long.

The blockwise optimization, applied in the BOL-KMP, again results in a significant performance gain compared to the unoptimized versions. For the **JITBackend** and **DDCGBackend**, we almost quadruple and double the throughput, respectively, compared to the interpreting versions.

Multiple patterns. For the query with the multiple pattern workload, we achieve a similar throughput as for the short pattern. Our pattern is composed of three rather small subpatterns.

When processing such patterns, we process one subpattern at a time. Once one subpattern is found, we start searching for the next one starting after the occurrence of the first pattern.

In the interpreting versions, we need to preprocess each subpattern before starting the search in the input text, which results in a loss of performance due to the preprocessing.

In the code generating versions, the subpatterns are preprocessed at code generation time and the lps tables are written into the code. The achieved throughput is between the throughputs of the short and medium patterns. When searching for multiple patterns, it depends on the individual subpatterns: In our case, the first subpattern is in the category of

a medium pattern. Due to that, the search for this first subpattern already leads to several input texts to be rejected as it cannot be found. However, for those texts that contain the first subpattern, the search for the next pattern takes some additional processing time; the same holds for the last subpattern. With this, we achieve a throughput between the small and medium queries.

4.2.1.2. Compressed LPS Table

One of our optimizations for the Knuth-Morris-Pratt algorithm is the use of the compressed lps table. This optimization aims to reduce the number of jumps in the pattern when a mismatch occurs between the character of the pattern and the text, and is particularly useful when searching for patterns with several repetitive parts.

However, during our evaluation, we found that this optimization did not result in a noticable increase in performance compared to the unoptimized version for real-world patterns. In some cases, such as with the O-KMP algorithm, it even led to a decrease in throughput. This is because the algorithm required modification in order to work correctly with the new table. This introduced additional branches and checks, which ultimately slowed down performance by causing mispredictions during execution. Although also the interpreting algorithm needs to be adapted, the compiler and optimizer rearrange the instructions to get more efficient code when Umbra is built.

For completeness, we present the results for the compressed OL-KMP in Fig. 4.3. The graph shows that, for the OL-KMP, this optimization did not lead to a higher throughput.

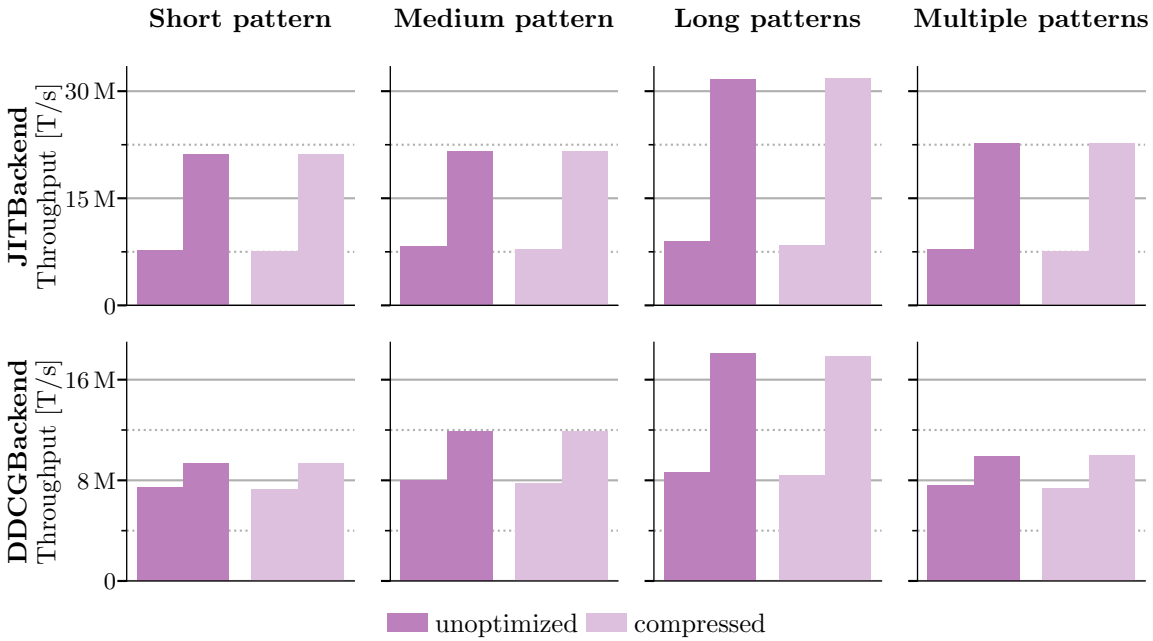


Figure 4.3.: Throughput for the different workloads of the unoptimized OL-KMP and of the compressed OL-KMP. The left (hatched) bar presents the throughput of the interpreting version, the right one is the code generating version. (Higher is better.)

4.2.2. Boyer-Moore Algorithm

In Sect. 2.3 and 3.3, we have demonstrated three different implementations for the Boyer-Moore algorithm. The first two are the proposed ones by Boyer and Moore, the Original BM (O-BM) and the Fast BM (F-BM). The third version is our proposed modification, the Blockwise BM (B-BM), that includes blockwise processing to speed up the F-BM implementation.

Fig. 4.4 illustrates the performance of the algorithms for the previously discussed queries. Similar to the KMP algorithms, the interpreting versions of the Boyer-Moore algorithm require preprocessing before the search can start for each tuple. The BM preprocessing generates two tables for the O-BM and B-BM and theoretically three tables for the F-BM. The preprocessing is explained in Sect. 2.3.1. Overall, we can see that the code generation version of each algorithm has a higher throughput than its corresponding interpreting implementation.

Short pattern. The performance of the interpreting versions of all implementations is quite similar, with the F-BM and B-BM only slightly faster than the O-BM. Based on performance

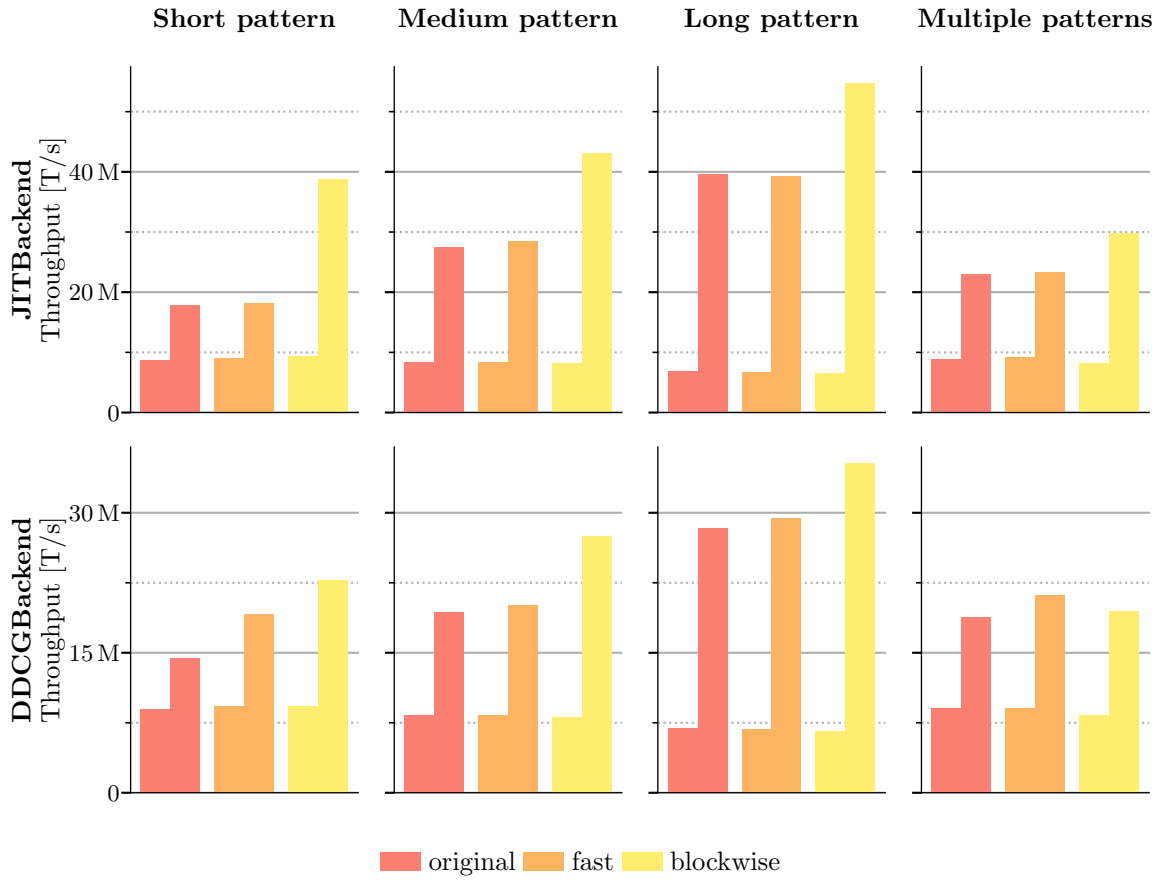


Figure 4.4.: Throughput for the different workloads of the BM algorithms. For each algorithm, the left (hatched) bar presents the throughput of the interpreting version, the right one is the code generating version. We present the Original BM (Sect. 2.3.2 and 3.3.1), the Fast BM (Sect. 2.3.3 and 3.3.2), and the Blockwise BM (Sect. 2.3.4 and 3.3.3). (Higher is better.)

measurements throughout the implementation phase of the F-BM, we decide not to explicitly generate the table δ_0, but instead to perform a check in order to add the value *large* correctly. Our suggested B-BM has similar performance to the F-BM in its interpreting version. By profiling the individual implementations and analysing their performance, we found that repeatedly preprocessing the pattern takes a significant amount of the computation time, so the F-BM and B-BM implementations cannot benefit from the applied optimization methods.

However, the code generating versions using the **JITBackend** clearly benefit from preprocessing the pattern only once and writing out everything to code. As shown in the graph, we nearly double the interpreting throughputs for the code generating O-BM and F-BM. The throughput for the F-BM is also a slightly higher than for the O-BM, indicating the advantage of the Fast BM implementation. Additionally, the B-BM benefits greatly from the blockwise optimization, with its throughput about four times that of the interpreting version.

For the **DDCGBackend**, we also observe the advantage of generating code for the pattern matching process. With this backend, the performance improvement of the F-BM over the O-BM is more obvious. However, due to the internal handling in the backend, the throughput for the O-BM and B-BM is lower than with the **JITBackend**; only for the F-BM, the throughput can keep up with the corresponding equivalent of the **JITBackend**.

Medium pattern. The throughputs for the interpreting algorithms with the medium pattern workload are similar to the short pattern but lower. This is because, even though a longer pattern allows for faster search termination when the input text is exceeded, the time for repeating preprocessing accumulates and reduces the throughput.

When it comes to the code generating versions, we see that for both the **JITBackend** and the **DDCGBackend**, the throughput of each of the algorithms starts to increase compared to the short pattern. This is because preprocessing once and writing everything to code as well as the higher number of characters in the medium pattern allows for processing more input texts in the same time interval.

Long pattern. As the pattern length increases from short over medium to long, the performance of interpreting versions also decreases due to more costly preprocessing. This extra time spent on preprocessing cannot be compensated by the advantage of long patterns that allow for faster search termination when the input text is exceeded.

Similar to the observations before, the throughput of the code generating implementations even increase further and are at least five times the throughput of the corresponding interpreting versions. However, as the pattern length increases, the difference between O-BM and F-BM becomes less pronounced, with the B-BM still being the dominant algorithm among the three.

Multiple patterns. For our pattern composed of multiple subpatterns, we process each subpattern individually. The first subpattern is a medium one, the remaining two are short ones. The throughput of each interpreting version for the composed pattern falls between the one for the short and medium patterns.

In the **JITBackend**, the throughput increases from the O-BM to the F-BM and then to the B-BM. This suggests that incorporating the blockwise optimization improves performance when searching for a pattern composed of multiple subpatterns. Although the throughputs

for the O-BM and F-BM are between the throughputs of the short and medium patterns, the throughput of the B-BM drops below the medium one. Analysis of the performance of the generated code reveals that the backend resolves some PHI nodes by writing to and reading from memory. Thus, adding the blockwise optimization to the algorithm does not allow the backend to perform an optimal register assignment as before causing a little performance drop.

Similar issues arise with the **DDCGBackend**. For the O-BM and the F-BM, the throughput remains comparable to that of the medium pattern. However, the B-BM is outperformed by the F-BM. This is due to the added effort required by blockwise processing in addition to the memory access used to resolve PHI nodes, which results in an overall drop in performance compared to the unoptimized versions.

4.2.3. Automaton Approach

In Sect. 2.4 and 3.4, we detailed the process of converting a given pattern into a Like-NFA, determinizing it to create a Like-DFA, and utilizing it for code generation.

The results of the generated code can be seen in Fig. 4.5, which illustrates the performance of the queries using the automaton for the string pattern matching process. Overall, the Blockwise Translation yields a higher throughput than the Direct Translation method.

However, there is a significant difference between the two backends, as the throughput of the **DDCGBackend** is mostly around half of that of the **JITBackend**. This is primarily due to the handling of PHI nodes: When generating the code for the automaton, during execution, each state needs to determine the current position in the text; to do that in SSA form, we need to put a PHI node at the start of each state. This PHI node needs to have exactly the same number of entries as the state has incoming edges. As the **DDCGBackend** resolves these PHI nodes by memory accesses, this results in a drop in performance compared to the **JITBackend**.

Another observation is that for both backends, the throughput for the medium pattern is the highest. This is because the automaton does not have an early return option, as the implementations of the other algorithms do. Each input text is processed until the last character is read if the pattern is not in the input text.

However, the throughputs of the medium and long pattern are slightly above the one for the short pattern. For those queries, the execution benefits from the starting character of the pattern. The short pattern starts with an `s`, whereas the other two patterns start with an `m`. As the distribution of the letters in our dataset shows, the letter `s` is more likely to appear than the `m`. This means that the automaton for the short pattern is entered more often than for the medium and long patterns. This highlights that the characters of the pattern and their distribution also play a role in performance.

For the pattern composed of multiple ones, the throughput is again between that of the short and medium patterns. However, the automaton consumes the entire input text as no early return optimization is included. The subpatterns themselves start with an `m` twice and an `s` once. Similarly to the explanation above, the `m` is found less frequently, so the throughput is closer to the medium one than the short one.

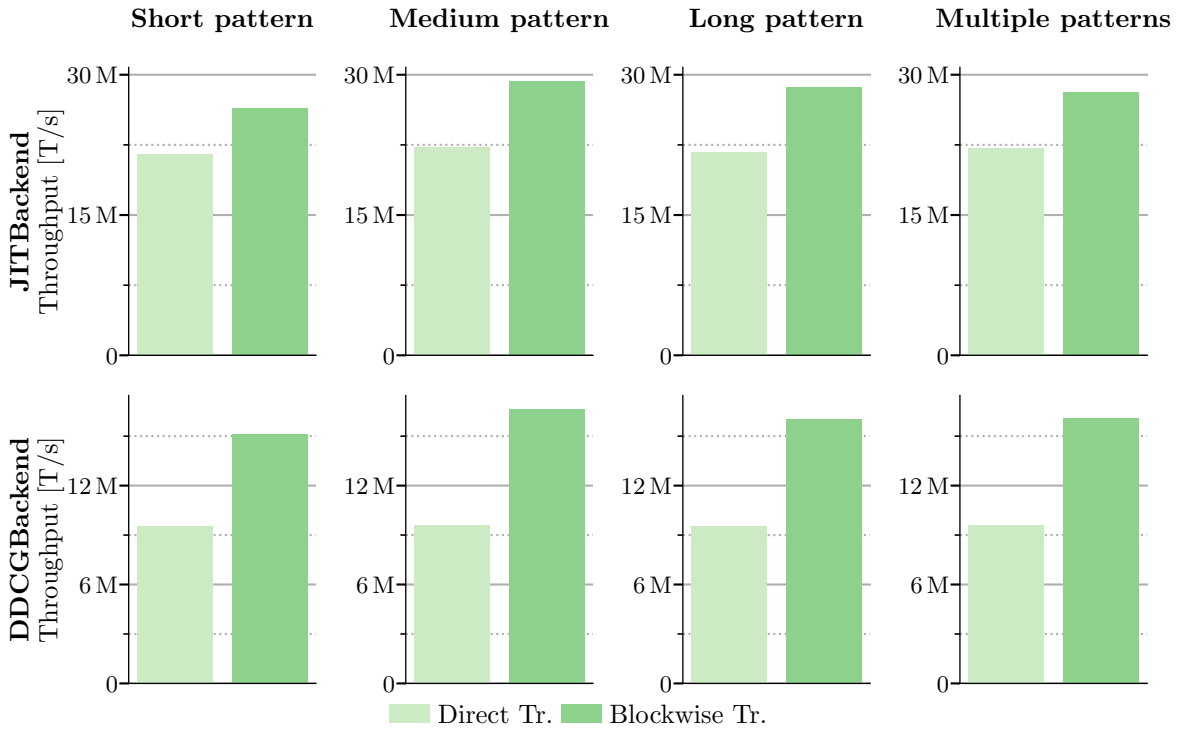


Figure 4.5.: Throughput of the different workloads with the Automaton Approach for the DFA. We present the Direct Translation (Sect. 3.4.1) and the Blockwise Translation (Sect. 3.4.2). (Higher is better.)

4.2.4. Comparison of the Code Generating Algorithms

Based on the results of the previous sections, we have found the best implementations and optimizations for each of the discussed string pattern matching algorithms.

Fig. 4.6 compares the best-performing algorithms for the given workloads. However, it should be noted that the algorithms cannot be compared that easily with another, as other patterns will lead to other results. For short, medium, and long patterns, we can see that the Blockwise Boyer-Moore algorithm yields the highest throughput for both the **JITBackend** and the **DDCGBackend**. Additionally, for both the KMP and BM algorithm, the performance of the code generating versions improve with increasing pattern length. This is because the KMP algorithms incorporate the early return optimization which allows for faster rejection of input texts when the pattern length exceeds the number of remaining characters in the input text. In the BM algorithms, we do not have to add such an optimization, as the algorithm itself compares from right to left and always checks if the end of the pattern is still inside the input text boundaries. Our Automaton Approach do not implement the early return optimization as including this step in the code generation process would require additional processing of the automaton itself. Due to this, the throughput does not increase with a longer pattern.

When it comes to a pattern composed of multiple subpatterns, the best-performing algorithm changes: In the **JITBackend**, the Blockwise KMP algorithm outperforms the Blockwise BM. However, when looking at the individual subpatterns, we notice that the last character of the subpattern is a `t`. This character has a higher ratio in the input text and there is a higer

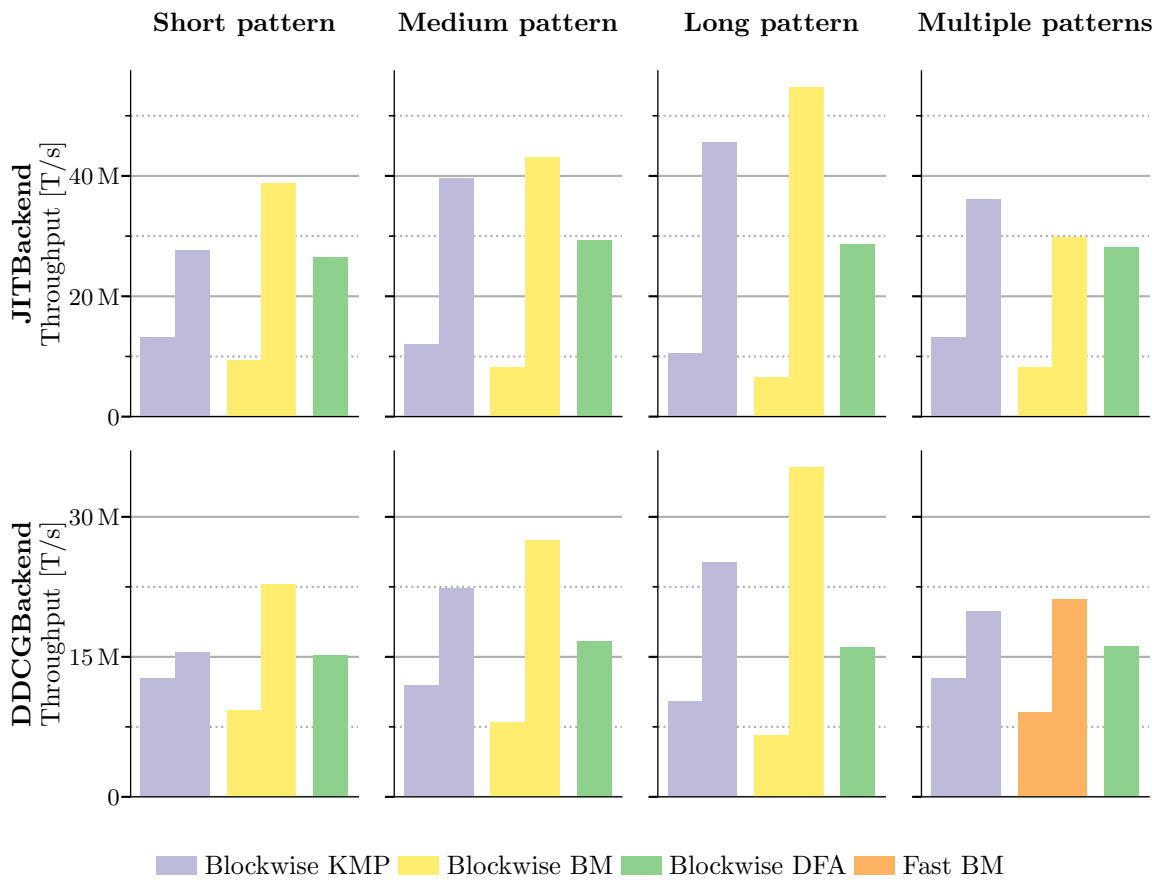


Figure 4.6.: Comparison of the best throughputs of the different code generating algorithms for the workloads. We present the best-performing algorithms for the corresponding workload. If available, the hatched bar presents the corresponding interpreting version of the algorithm. (Higher is better.)

probability that it is matched than the starting character of the pattern, an `m`, for the KMP algorithm. Based on that insight, we can conclude that for this pattern, it is better to use the Blockwise KMP than the Blockwise BM algorithm.

For the **DDCGBackend**, we can see a similar behaviour: For the short workload, the Automaton Approach and the Blockwise KMP perform similarly. However, as the pattern length increases, the throughput of the Automaton Approach does not improve due to the lack of the early return optimization, while the throughput of the Blockwise KMP continues to improve. The same holds for the Blockwise BM, whose performance increases from the short over the medium to the long patterns. For the composed pattern (multiple patterns), the Blockwise BM is outperformed by the alternative Fast BM implementation as the internal resolving for the B-BM becomes too complex and the optimization starts to slow down the execution.

4.2.5. Compilation Time

Another important consideration when using the code generation approach is the time required to compile the generated code. In the interpreting approaches, the generated code only contains calls the function performing the matching process. When replacing that with the code generating approaches, we add several instructions and basic blocks to the generated code. The increasing size of the code leads to a higher compilation time.

In Fig. 4.7, we present the runtime of the best-performing algorithms including the compilation time. The graph displays two pieces of information: the lower portion (in a lighter color) represents the compilation time, while the upper portion (in a darker color) represents the execution time. contain two information: the lower one (brighter color) visualizes the compilation time, the upper one (normal color) the time for the execution of the code. It is worth noting that lower runtime is better.

When analyzing the performance of the **JITBackend**, we can see that the compilation time for the code of the interpreting versions is relatively similar across all patterns. However, for the code generating versions, we observe that the compilation time increases as the patterns become longer and more complex. Specifically, the pattern composed of multiple subpatterns leads to a higher compilation time. The Automaton Approach has a much higher latency due to compilation than the other versions, as it introduces multiple basic blocks per state in the automaton: The longer the pattern, the more basic blocks are generated, and the longer the compilation takes.

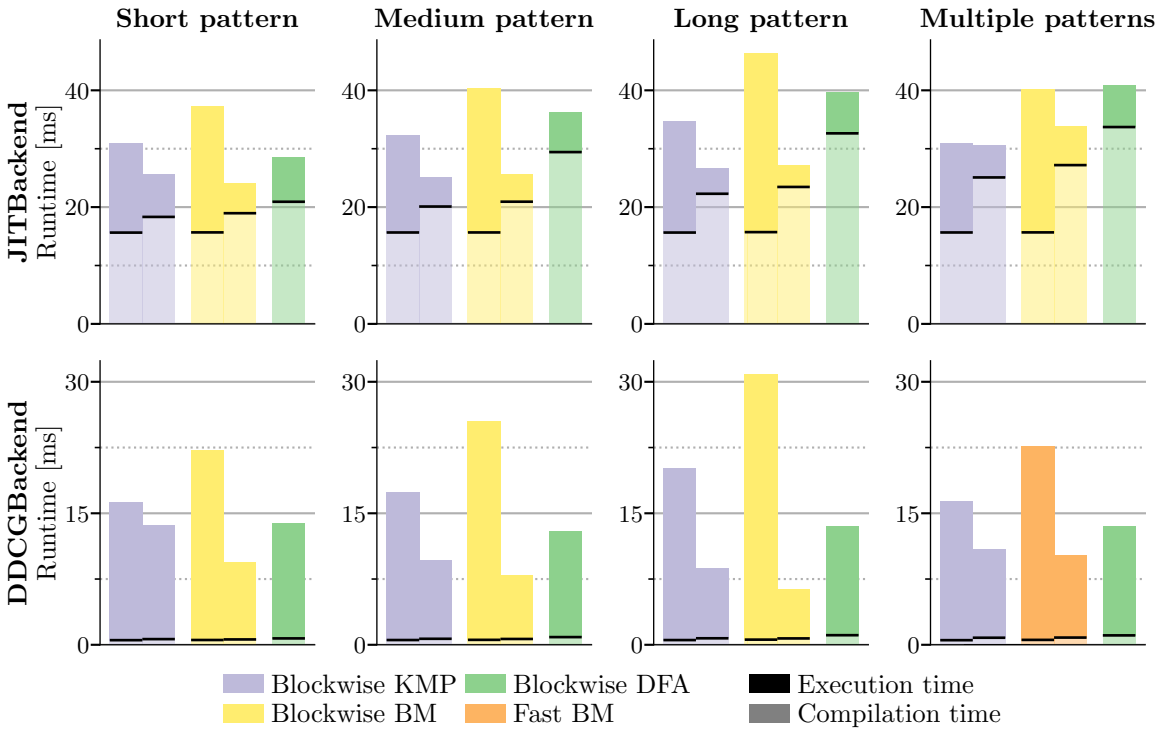


Figure 4.7.: Compilation and execution times for the best-performing algorithms on the TPC-H dataset with one thread. If available, the hatched bar is the interpreting version. The stacked bar is composed of the compile and execution time. (Lower is better.)

In the **DDCGBackend**, the use of the code generation framework and compiler backend helps to minimize compilation time and resulting latency, as presented in [KLN21]. Similarly to the **JITBackend**, the compile time for the interpreting approaches remains consistent across different patterns. However, for the code generating approaches, we notice a slight increase in compilation time with longer patterns, though the increase is hardly noticeable. Again, the Automaton Approach has the highest increase in compilation time with longer patterns.

When taking into account both the compilation and execution time, we can see that in nearly all combinations of backend, pattern size, and applied algorithm, the code generation performs better than the interpreting ones. The only exceptions are when using the **JITBackend** for the long and multiple pattern workloads: in the former case, the Automaton Approach is outperformed by all other implementations and is slower than the interpreting versions. In the latter case, the Automaton Approach is slower than the interpreting versions due to the high latency caused by the compilation overhead, and the code generating Blockwise KMP is also slower than its interpreting version. However, the main reason is the compile time difference between the two versions.

4.2.6. ClickBench Results

The main problem of code generation is latency caused by compiling the code. In this section, we will examine the impact of running queries on a larger dataset and using multiple threads to run a single query. When executing the generated code with a single thread on a small dataset, a significant portion of the total runtime is spent on compiling the code, as shown in Figure 4.7.

When running a query on a larger dataset like ClickBench, we expect the effect of compilation on the total runtime to be smaller. However, using multiple threads to run one query should result in a decrease in overall execution time. Fig. 4.8 shows the overall single-threaded runtimes for the queries on the ClickBench dataset, Fig. 4.9 with eight, and Fig. 4.10 with 20 threads (in all figures, we present the overall runtimes, so lower is better).

For all queries, we notice that the Blockwise BM is outperformed by the Fast BM, especially in the code generating approaches; so, we replace the results for the Blockwise BM with the Fast BM. When looking at the performance records for the implementations in both interpreting and code generating approaches, we can see that the compiler rearranges the instructions for the Fast BM and generates a tight loop in which the shift value from δ_0 is added to find the last character of the pattern. The hotspots for the Fast BM code are when reading the shift value, reading the next character of the input text, and checking if we had a match or not. In the Blockwise BM, the code is not that tight due to the increased number of instructions and we also require an unaligned load from memory as we start reading at random positions of the input text. Other hotspots are throughout the bitwise and arithmetic operations to determine if there is an occurrence or not and, in the case there is one, finding the index at which it is also takes several cycles.

One thread. As presented in Fig. 4.8, when using one thread for the workload, the compile time makes up only a small percentage of the overall runtime. For the execution of the generated code, we see a similar behaviour as for the TPC-H scheme. For both backends, almost all code generating versions have a better runtime than their interpreting versions.

Additionally, we also notice that the Automaton Approach performs similarly to the other algorithms. By using the bigger dataset, the huge compilation overhead for the generated code of this approach can be amortized and achieve almost the same runtime as the Blockwise KMP.

For the **JITBackend**, we clearly benefit from the code generation for the patterns. The execution times for the interpreting algorithms are all slower than for the code generating approaches and the compile time does not have a huge effect for the overall performance.

In the **DDCGBackend**, we notice that the two Blockwise KMP versions do not differ that much. Especially, for Query 22, the interpreting version is slightly faster than the code generating implementation. However, for the other approaches, we again see better performance of the code generating versions than the interpreting ones.

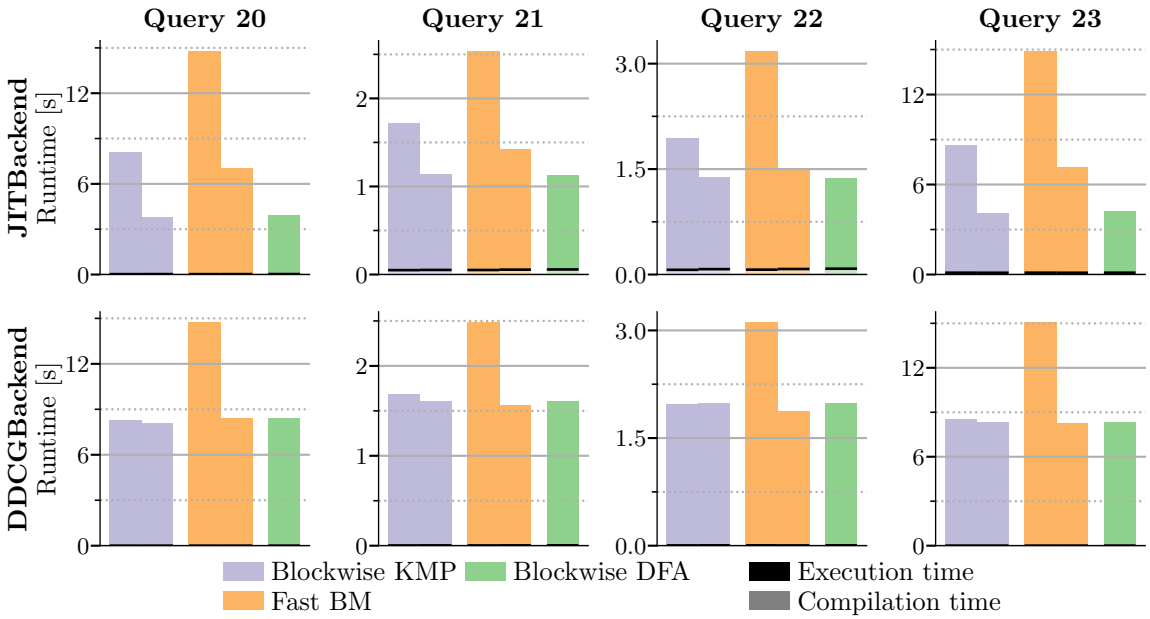


Figure 4.8.: Compilation and execution times for the best-performing algorithms on the ClickBench dataset using 1 thread. If available, the hatched bar is the interpreting version. The stacked bar is composed of the compile and execution time. (Lower is better.)

Eight threads. The compilation and execution times for the queries using 8 threads can be seen in Fig. 4.9. Compared to the single-threaded execution, we achieve a huge speedup of the queries by using multiple threads for the execution.

In the **JITBackend**, we observe that the compile time for the queries start to make up a noticalbe portion of the runtime for Query 21 and 22. As before, the code generating approaches perform better than the interpreting approaches. For all queries and algorithms, code generation is faster, in some cases even by a factor of two. Similarly to the single-threaded execution, the compilation overhead for the Automaton Approach is balanced out by the longer runtime on the larger dataset, so it performs similarly to the Blockwise KMP algorithm.

For the **DDCGBackend**, the code generating versions are also better, but not as much of a

factor as for the other backend. Looking at the Blockwise KMP versions, the code generation only leads to a small performance improvement compared to the interpretation. For Query 22, the interpretation and the code generation are almost equal. For the BM, we can see a performance gain of up to a factor of 1.5. Similarly to before, the Automaton Approach is in the same range of runtime as the KMP and BM algorithm.

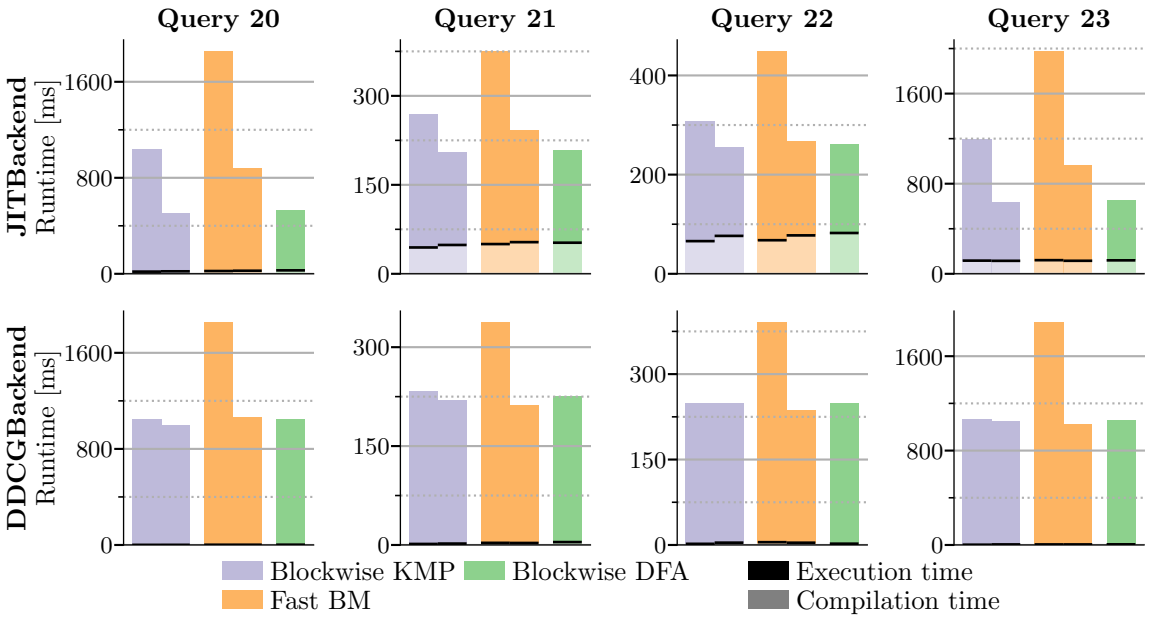


Figure 4.9.: Compilation and execution times for the best-performing algorithms on the ClickBench dataset using 8 threads. If available, the hatched bar is the interpreting version. The stacked bar is composed of the compile and execution time. (Lower is better.)

20 threads. In our final experiment, we executed the queries using 20 threads. Once again, we see that for all workloads and algorithms, the code generating ones are more efficient. Fig. 4.10 shows the compile and execution time for the workload. For the execution times, we see a similar pattern as in the single-threaded case of when using 8 threads.

In the **JITBackend**, the compile time makes up almost half of the overall runtime of Query 21 and 22. However, the advantages of code generation still outweigh the interpretation, altough the effects are not as drastic anymore. For the queries 20 and 23, we still see a performance improvement in the execution time by nearly a factor of 2. When the Automaton Approach is used, we start to notice the drawback of the higher compilation time. Although the execution time is about the same as for the KMP, the compile time is higher which results in a slightly worse overall runtime. However, the generated code is still faster than both the interpreting KMP and BM.

For the **DDCGBackend**, the compile time does not take such a large ratio of the runtime. However, due to the limitations of the backend, the performance benefits of the code generating algorithms are not as significant. Only for the BM algorithm, the code generation runtime is around 50% faster than the interpreting one. For the KMP and Automaton Approach, the

benefit is not that much as for the BM, but the code generation is faster than the interpretation.

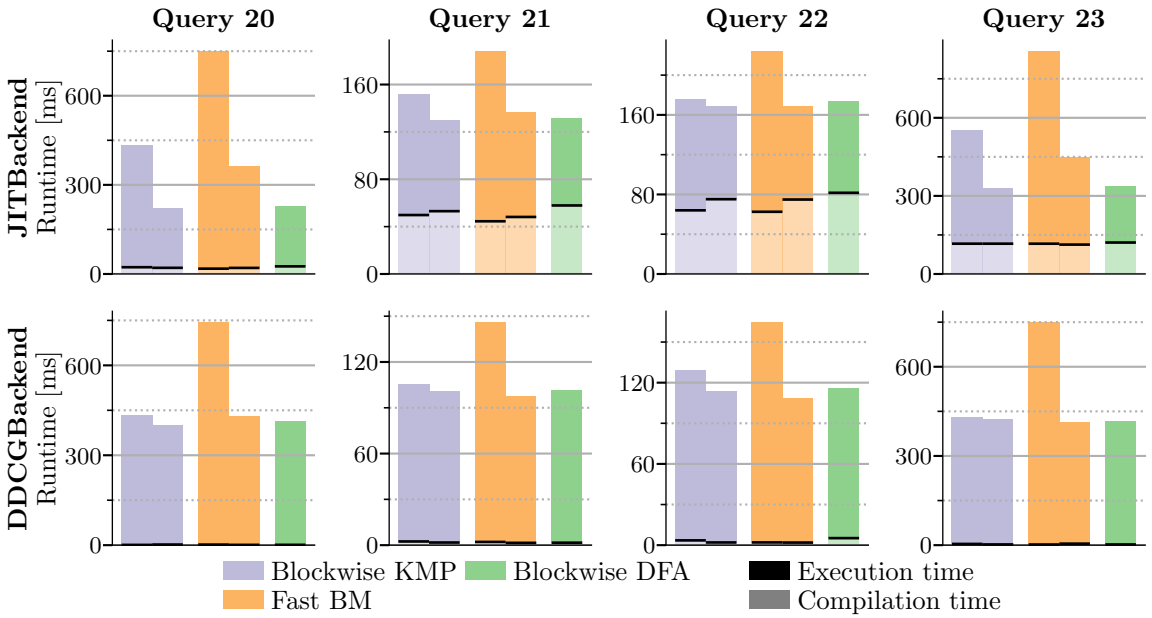


Figure 4.10.: Compilation and execution times for the best-performing algorithms on the ClickBench dataset using 20 threads. If available, the hatched bar is the interpreting version. The stacked bar is composed of the compile and execution time. (Lower is better.)

5. Non-Exact Pattern Matching

In the preceding chapters, we have extensively explored exact pattern matching using LIKE expressions, with a specific focus on patterns that only include the % wildcard. However, as the LIKE expression also allows the use of the _ wildcard, it is necessary to also consider non-exact pattern matching. Unlike exact pattern matching, there are fewer algorithms available for handling patterns with 'Don't Care' characters. One such algorithm, presented by Gusfield in [Gus97], involves splitting the subpattern at the Don't Care symbols and searching for each of these character sequences individually. Indices for these sequences are stored, and later, we check whether we can find an index for each of the subsequences that fulfill the requirements of the number of Don't Care symbols between them. However, the algorithm described above is complex, as it requires storing various information about the indices, making it difficult to implement a code generation approach.

In this chapter, we present two alternative approaches for dealing with these types of patterns and generating code to perform the matching process. For simplification, we assume that the patterns do not start with a Don't Care symbol. If a Don't Care symbol is at the start of a pattern, we would scan through the pattern from the start until the first character that is not a Don't Care wildcard is found and handle the previous _ wildcards with the pattern before.

5.1. Extended Automaton Approach

In Sect. 2.4, we have discussed a new approach how to generate an automaton from a LIKE expression by explaining the steps how to build a Like-NFA from the given pattern. However, we only included the handling of % wildcards as only those are allowed for exact pattern matching.

5.1.1. Extended Like-NFA

To address the issue of non-exact pattern matching, we must implement a new case to handle the _ wildcard character. With this addition, we have a total of four rules for constructing a Like-NFA from the pattern:

_ : add an edge accepting Σ to the last state going to a newly added state at the end

% : add a loop to the last state accepting Σ

ESC : go to the escaped character α and add edge accepting α to the last state going to a newly added state at the end

$\alpha \in \Sigma$: add edge accepting α to the last state going to a newly added state at the end

By applying the rules above, we see that an _ wildcard is handled like a normal character by leading from one state to the next one, just that the transition accepts any character.

Example. In Fig. 5.1, we see the automaton for the pattern %a_ba%. Like with exact pattern matching, the transitions between the states represent the pattern, with the exception of the Σ transition between states q_1 and q_2, which represents the _ wildcard.

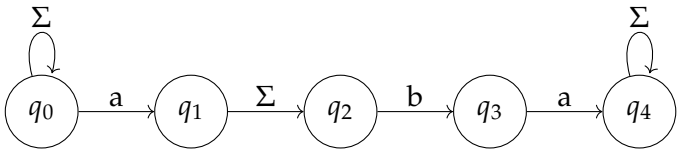


Figure 5.1.: Extended Like-NFA for pattern %a_ba%

5.1.2. Extended Like-DFA

With the addition of a new rule for building a Like-NFA from a pattern, we must also revise our approach for constructing the Like-DFA from the Like-NFA. We continue to use the KMP approach to determine the lps state, but now require a special case to handle the Σ transitions between states.

When processing a pattern with a _ wildcard character, it is necessary to keep track of the character that was read for the corresponding wildcard. The code in Listing 5.1 demonstrates how the determinization process works for the Extended Like-NFA.

```
1  determinizeExtended(nfa)
2    [subPatStart, subPatEnd] = getNextSubpattern(nfa);
3    // initialize the fifoList with current and its lps
4    fifoList = initializeFIFO(subPatStart.next, subPatStart);
5    while (fifoList.hasElement())
6      [current, lps] = fifoList.popFirstElement();
7      fTChar = current.forwardTransition.character;
8      if (fTChar != Σ)
9        // handle case A
10       for (transition : lps.transitions)
11         if (transition.character == fTChar) continue;
12         current.transitions.insert(transition);
13       if (lps.forwardTransition.character != fTChar)
14         current.transitions.insert(lps.forwardTransition);
15       fifoList.append(current.next, determineNextLps(lps, fTChar));
16     else
17       // handle case B
18       copiedStates = copyStates(current.next, subPatEnd);
19       relinkedState = relink(current, lps, copiedStates);
20       fifoList.append(relinkedState, lps.next);
21       fifoList.append(current.next, determineNextLps(lps, Σ));
```

Listing 5.1: Algorithm to convert an Extended Like-NFA into an Extended Like-DFA

When converting an Extended Like-NFA to an Extended Like-DFA, it is necessary to keep track of the longest proper prefix (lps) state for each state. To achieve this, the Like-NFA is traversed in a breadth-first order using a First-In-First-Out (FIFO) queue.

During each iteration, the first element of the queue is taken and processed (line 6). If the forward transition of the current state is not a Σ transition, then the process is similar to the

one described in Sect. 2.4.2. The lps transitions are added to the current state; the next state and its corresponding lps state are added to the FIFO queue for the next iteration (line 15).

However, when we encounter a Σ transition as a forward transition, additional steps must be taken to keep track of which character is read at that position. As the current lps state has exactly one forward transition, we know that it matters whether the character read for the Σ transition is the same as the character of the forward transition of the lps or not. To resolve this in the automaton, we need to copy all of the states following the one reached by the Σ transition up to the state whose forward transition leads to the end state of the subpattern (line 18). This is done by the function `copyStates(...)` which returns a pointer to the head of the newly copied states. This head will now be linked to the current state with a new transition (line 19). This transition must accept the same character as the forward transition of the lps state, which means that, in case of a Non-ASCII character, we additionally need to copy the states of the character of the lps' forward transition. All of that functionality is handled in the function `relink(...)` which returns a pointer to the first state of the newly added states.

After the relinking took place, we need to insert two new pairs into the FIFO queue. The first pair is the relinked state and its new lps pointer. The second pair is the state that is reached by the Σ transition, but we need to call the `determineNextLps(...)` function to determine the corresponding lps state.

Example. In Fig. 5.2, we present the Extended Like-DFA for the pattern `%a_ba%`. State q_1 now has two outgoing transitions: one Σ transition to q_2 and one with character `a` to q_5. This allows us to implicitly store which character was read for the Don't Care wildcard.

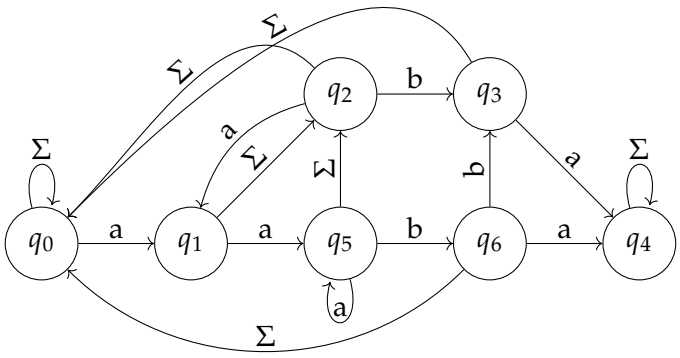


Figure 5.2.: Extended Like-DFA for pattern `%a_ba%`

Let us examine an example for this automaton: Imagine we have reached state q_6, indicating that the last three characters of the input text must be `aab`. From there on, we need to consider the following three cases:

- If the next character is an `a`, we have found the substring `aaba` which matches the pattern. Therefore, we proceed to state q_4 to accept the input text as a match.
- If the next character is a `b`, we have found the substring `aabb` which does not match the pattern. However, the last three characters of the input text match the first three characters of the pattern. Therefore, we need to end in state q_3.

- If the next character is neither an `a` nor a `b`, we must return to the start state q_0, as no suffix of the input matches a prefix of the pattern.

5.2. Reduced Automaton Approach

In the previous section, we discussed the option of building a full automaton from the given pattern. However, during implementation, we encountered a limitation of some of the backends of Umbra, which prevents the code generation framework from generating and executing code that contains irreducible loops [Hav97]. This limitation can cause problems during execution and, in the worst case, even cause the database system to crash when certain patterns are used.

To address this challenge, we present an alternative approach for translating patterns with Don't Care wildcards to code. This approach requires backtracking in the input text if a mismatch occurs.

Preparation. The concept of this approach is as follows:

1. First, we convert the pattern into an Extended Like-NFA using the rules outlined in Sect. 5.1.1.
2. During the determinization process to a Reduced Like-DFA, we do not construct the full automaton as we did in the approach outlined in Sect. 5.1.2.
   Instead, we divide the Extended Like-NFA into logical parts:
   - Part A includes the states from the start state to the first state whose forward transition is a Σ transition.
   - Part B begins at the state where Part A ends and goes to that state after which only Σ transitions follow.
   - Part C includes the remaining states from the previous one until the last one.
3. For Part A, we can apply the standard algorithm for determinization of the Like-DFA. The other parts remain unchanged.

With that, we receive a Reduced Like-DFA which is used for the following code generation.

Code Generation. During the code generation of the Reduced Like-DFA, we utilize the different parts defined earlier. At the start, we include a backtracking block, which sets the position in the input text and saves the next index in case of backtracking.

For Part A, we have built the full automaton, for which we can generate code directly. For Part B, we write out the different types of forward transitions that can be found: For Σ transitions, we generate code to skip the corresponding number of bytes in the input text and to go to the next state. For transitions with a specific character, we generate code to compare this character with the character from the input text. If the characters match, we go to the next state; if not, we need to start backtracking. To do this, we jump back to the previously generated backtracking block in which we stored the index at which the last scan of the pattern started, move one position to the right, store the new index again, and perform

the next matching phase. For Part C, we generate code to consume the corresponding number of bytes in the input text and move on.

In Fig. 5.3, we present the conceptual idea of the Reduced Like-DFA for the pattern `%ab_a__%`. The generated code can be found in Listing 5.2.

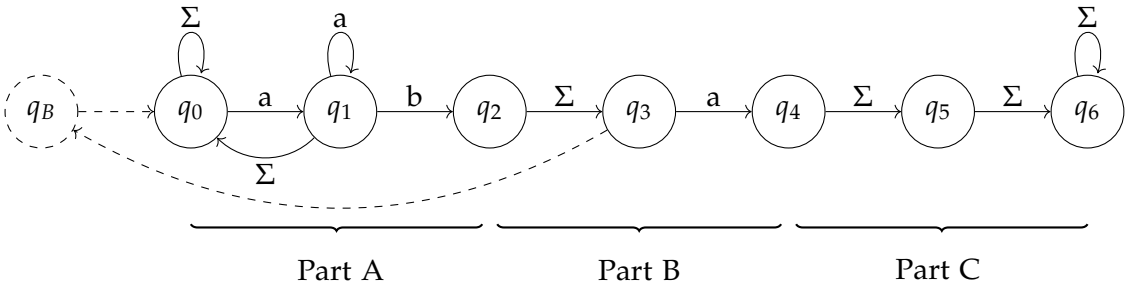


Figure 5.3.: Extended Like-NFA for pattern `%ab_a__%`. The state q_B and dashed edges are inserted during the code generation process to handle backtracking.

```
// backtracking block
qB: tPos = tPosBefore + 1; goto q0;
// handling Part A
q0: if (tPos >= text.size()) { return false; }
    c = text[tPos];
    if (c == 'a') { tPos++; goto q1; }
    else { tPos += Utf8::length(c); goto q0; }
q1: if (tPos >= text.size()) { return false; }
    c = text[tPos];
    if (c == 'a') { tPos++; goto q1; }
    else if (c == 'b') { tPos++; goto q2; }
    else { tPos += Utf8::length(c); goto q0; }
// handling Part B
q2: if (tPos >= text.size()) { return false; }
    c = text[tPos];
    tPos += Utf8::length(c); goto q3;
q3: if (tPos >= text.size()) { return false; }
    c = text[tPos];
    if (c == 'a') { tPos++; goto q4; }
    else { goto qB; }
// handling Part C
q4: if (tPos >= text.size()) { return false; }
    c = text[tPos];
    tPos += Utf8::length(c); goto q5;
q5: if (tPos >= text.size()) { return false; }
    c = text[tPos];
    tPos += Utf8::length(c); goto q6;
q6: return true;
```

Listing 5.2: Generated code for the automaton in Fig. 5.3 for the pattern `%ab_a__%`

6. Conclusions and Outlook

In this chapter, we want to discuss the results we got from the experiments, summarize them, and answer the question whether to generate code for the pattern or to interpret it.

With our experiments, we are confident to show the benefits of code generation for pattern matching; however, no experiment can be complete enough to represent each application and algorithm in every situation. Moreover, we hope that the database developer recalls that choosing the wrong algorithm, whether interpreting or code generating, for the wrong dataset and application can have significant effects on performance and runtime.

6.1. Code Generation Independent Insight

All of the points listed here apply for both the interpreting and the code generating algorithms we looked at in our experiments.

Distribution of characters. As one can see from the used datasets introduced in Sect. 4.1.2, we intend to perform our experiments on real-world like text datasets. This means that our alphabet is sufficiently large and the input texts are not just a repetition of a few characters, as it would be for e.g. gene sequences. In our cases, the TPC-H dataset covers all lowercase ASCII-letters and the whitespace character; the ClickBench dataset includes several different characters like lowercase and uppercase letters, special characters, and also Non-ASCII characters.

The first observation is that the distribution of the characters in the input text and pattern matters. Although we only present the results for the given patterns, we noticed that changing the pattern leads to varying throughputs. The reason is that the algorithms like the Boyer-Moore ones or the ones with the blockwise optimization search for a certain character in the input text. If the probability of the searched character in the input text is low, more of the input text can be processed before an occurrence is found; if it is high, then the character is found more often and a matching process must start. For the algorithms with blockwise processing or the ones searching for a specific character to start the matching process, that could even lead to a slowdown of the execution if the character is found too often.

When looking at the ClickBench Query 20 and the single-threaded results for it in Fig. 4.8, we can see the effect we have discussed: the pattern in that query is just `%google%`. Combining it with the distribution of the letters in Table C.1, the letter `g` makes up around 1% of all letters, whereas the letter `e` over 5%. This effect is also noticeable in the results, as the BM implementation which searches for the `e` in the input text takes nearly twice the time the KMP algorithm takes which searches for the `g`. The BM needs to stop and start the pattern matching process more often than the KMP does and thus requires more time to search for the pattern.

With that in mind, we cannot say exactly which algorithm is best for which pattern. The performance depends on several factors like the size of the alphabet, the distribution of the

characters in the input text, and the characters in the pattern.

Pattern length. Another quite important part of the overall performance is the pattern length and the optionally applied early return optimizations. With longer patterns, one can detect earlier if the pattern exceeds the input text and reject that input text. As we have seen on the TPC-H dataset, longer patterns achieved a higher throughput for all algorithms as both the optimized KMP and the regular BM algorithms stop the execution once the input text is exceeded. For the Automaton Approach, this does not hold, as we do not generate code for the early return optimization. Due to this, the throughput does not increase as it does for the other implementations for longer patterns.

6.2. Code Generation Dependent Insight

In this section, we deal with points that have a direct effect on the code generation for the algorithms.

Choosing the matching algorithm. As discussed before, the choice of the algorithm to perform the pattern matching process itself is not straightforward, as it depends on several factors like the character distribution in the input text or the number of different characters in the pattern. Based on the results, we cannot directly say which of the KMP or the BM algorithms is better for pattern matching, as it depends on the pattern. However, once the algorithm was chosen, our results clearly show that it makes sense to generate code for the matching process instead of using the interpreting approach. The main drawback of the interpreting algorithms is both the preprocessing of the patterns, which can take a significant part of the overall runtime, and the repeating access to the pattern and the results of the preprocessing. In the code generating approaches, this only needs to be done once at generation time and then no further accesses to pattern or preprocessed tables is required.

Compilation backend. After generating the code, we need to compile and then execute it. As we have seen before, the performance depends on the chosen backend: in the **JITBackend**, the compilation latency is quite high, but can be evened out most of the time when executing the query; for the **DDCGBackend**, the compilation time is very low, but the generated code is not much faster due to some internal restrictions. Additionally, we notice that in the **DDCGBackend**, the performance also depends on the structure of the generated code, as interleaved loops with many PHI nodes in the code slow down the execution, as it is the case for the Original KMP.

However, for both backends, it is obvious that a suitable code generation approach can be found which then leads to a higher throughput in most of the cases compared to the interpreting alternative.

6.3. Outlook

With the previous algorithms, we have covered exact pattern matching algorithms which are already known for quite some time. Umbra itself already includes an interpreting version

of a hybrid string search algorithm. For some patterns fulfilling certain length conditions, the hybrid search uses the SSE4.2 vector instruction `_mm_cmpistri`, which compares packed strings; for all other patterns, the default Two Way Search algorithm is applied. The Two Way Search combines both the KMP and BM algorithm to perform the matching process [CP91]. Until now, we have already implemented a code generating version for the Two Way algorithm. One future project could be to add support for the required SSE4.2 instructions in Umbra. With these instructions, we can implement the hybrid string search algorithm as a code generating version to generate specific code for a given pattern and to analyze the performance.

In this thesis, we have primarily covered and analyzed exact matching algorithms, which only involve % wildcards. However, to fully support the SQL LIKE expression, we also need to consider the use of the _ wildcard. This topic was briefly discussed in Chapter 5. We have developed two methods to extend the Automaton Approach to handle patterns with any combination of wildcards. The two options presented, the Extended and Reduced Automaton, can be used for code generation. A potential next step would be to evaluate the performance of these approaches in comparison to a manually implemented matching function. Additionally, incorporating the previously discussed pattern matching algorithms with the concept of pattern splitting could be explored as a means of applying code generation for pattern matching.

A. Blockwise Processing

A.1. Blockwise Search for ASCII character - Example

Table A.1 lists the example for the blockwise search of the ASCII character `0x44` in the block `0x4142434445464748`. The code for this search was presented in Listing 2.5.

`block`	`0x4142434445464748`
`searchedChar = broadcast(0x44)`	`0x4444444444444444`
`high`	`0x8080808080808080`
`low`	`0x7F7F7F7F7F7F7F7F`
`lowChars = (~block) & high`	`0x8080808080808080`
`t1 = block & low`	`0x4142434445464748`
`t2 = t1 ^ searchedChar`	`0x050607000102030C`
`t3 = t2 + low`	`0x8485867F8081828B`
`t4 = t3 & high`	`0x8080800080808080`
`found = ~t4`	`0x7F7F7FFF7F7F7F7F`
`matches = found & lowChars`	`0x0000008000000000`

Table A.1.: Blockwise processing to search `0x44` in `0x4142434445464748`

A.2. Blockwise Search for Non-ASCII character

Listing A.1 shows the modified code to search for the start byte of a Non-ASCII character. Table A.2 lists the example for the blockwise search of the Non-ASCII character `0xA4` in the block `0x414243A4B5C64724`.

```
uint64_t block = loadNext8Bytes(ptrToText);
uint64_t searchedChar = broadcast((c & 0x7F)); // broadcast c to each byte
constexpr uint64_t high = 0x8080808080808080ull;
constexpr uint64_t low = ~high;
uint64_t highChars = block & high;
uint64_t found = ~((((block & low) ^ pattern) + low) & high);
uint64_t matches = found & highChars;
bool matchFound = matches != 0;
```

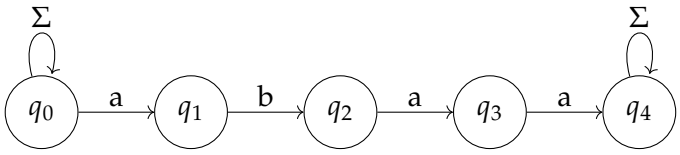
Listing A.1: Blockwise Character Search for Non-ASCII characters

block	0x414243A4B5C64724
searchedChar = broadcast((0xA4 & 0x7F))	0x2424242424242424
high	0x8080808080808080
low	0x7F7F7F7F7F7F7F7F
highChars = block & high	0x0000008080800000
t1 = block & low	0x4142432435464724
t2 = t1 ^ searchedChar	0x6566670011626300
t3 = t2 + low	0xE4E5E67F90E1E27F
t4 = t3 & high	0x8080800080808000
found = ~t4	0x7F7F7FFF7F7F7FFF
matches = found & highChars	0x0000008000000000

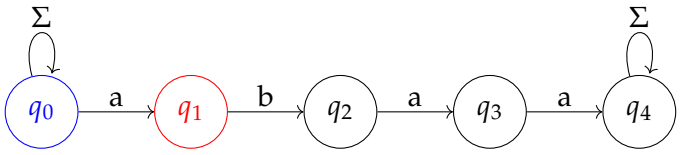
Table A.2.: Blockwise processing to search `0xA4` in `0x414243A4B5C64724`

B. Determinization of Like-NFA

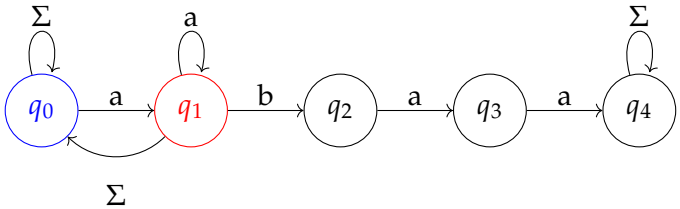
In Fig. B.1, we present the process to determinize the Like-NFA from Fig. 2.3. Every step is annotated with an explanation what changes are made and why by referring to Listing 2.10.



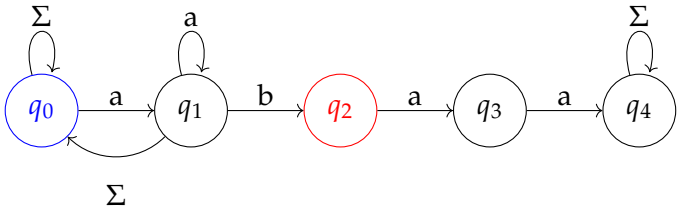
(a) Like-NFA to pattern `%abaa%`



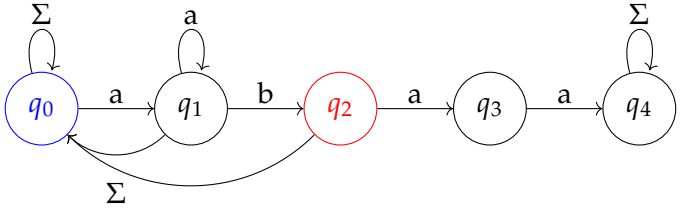
(b) Initialization: lps $= q_0$, current $= q_1$



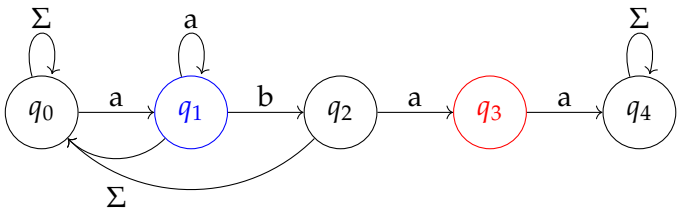
(c) Adding transitions of lps to current



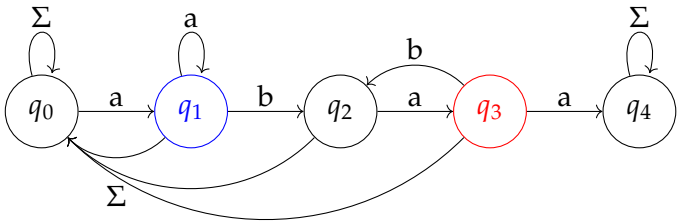
(d) Advancing current with character b to q_2, lps stays in q_0 due to Σ transition.



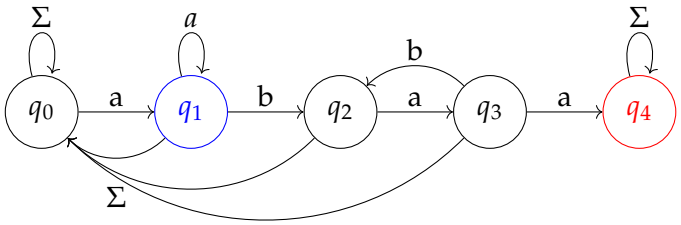
(e) Adding transitions of lps to current. As the forward transition of current has the letter a, we only add the Σ transition of q_0.



(f) Advancing current with character a to q_3, lps moves to q_1 due to forward transition.



(g) Adding transitions of lps to current. As the forward transition of current has the letter a, we only add the Σ transition of q_1.



(h) Advancing current with character a to q_4, lps stays in q_1 due loop with character a. Termination of the algorithm as end of section reached.

Figure B.1.: Step-by-step determinization of the Like-NFA of pattern `%abaa%`

C. Distribution of ClickBench Dataset

character	ratio [%]	character	ratio [%]	character	ratio [%]
15	$5.5128 \cdot 10^{-08}$	;	0.1030	^	$3.7377 \cdot 10^{-06}$
23	$3.3077 \cdot 10^{-08}$	<	0.0011	_	1.3734
24	$1.3230 \cdot 10^{-07}$	=	2.2382	‘	$9.3387 \cdot 10^{-06}$
25	$1.1025 \cdot 10^{-08}$	>	0.0001	a	4.6978
26	$1.4333 \cdot 10^{-07}$	?	0.4663	b	0.8836
27	$6.6154 \cdot 10^{-08}$	@	0.0004	c	1.9633
28	$1.4333 \cdot 10^{-07}$	A	0.1510	d	2.3606
29	$6.6154 \cdot 10^{-08}$	B	0.0569	e	5.2432
␣	0.7085	C	0.1427	f	0.6899
!	0.0053	D	0.3235	g	1.0464
"	0.3393	E	0.1108	h	2.8257
#	0.1551	F	0.4601	i	4.1304
$	0.0006	G	0.0789	j	0.1557
%	0.8889	H	0.0492	k	1.1877
&	1.9633	I	0.2033	l	2.3201
’	0.0005	J	0.0189	m	1.8866
(	0.0634	K	0.0674	n	2.3531
)	0.0581	L	0.1122	o	4.0155
*	0.0006	M	0.1432	p	3.4544
+	0.0015	N	0.1159	q	0.0365
,	0.2187	O	0.0743	r	4.9654
-	1.1360	P	0.0727	s	3.2526
.	2.5085	Q	0.0157	t	5.5865
/	5.2571	R	0.1122	u	2.5190
0	1.9912	S	0.1107	v	0.8533
1	1.9697	T	0.1896	w	0.6793
2	1.9619	U	0.0823	x	0.6762
3	1.5160	V	0.0947	y	1.0674
4	0.8768	W	0.0907	z	0.2436
5	1.1263	X	0.0461	{	0.0011
6	1.2414	Y	0.0329	|	0.0028
7	1.0524	Z	0.0263	}	0.0019
8	0.9003	[	0.0921	~	0.0094
9	0.9841	\	0.0012		
:	1.1528	]	0.0949		

Table C.1.: Distribution of the ASCII characters in the ClickBench dataset

character	ratio [%]	character	ratio [%]	character	ratio [%]
128	0.2006	162	0.0054	198	$7.7179 \cdot 10^{-08}$
129	0.1103	163	0.0010	200	$1.1026 \cdot 10^{-08}$
130	0.1338	164	0.0036	201	$5.9539 \cdot 10^{-07}$
131	0.0507	165	0.0005	204	$1.2018 \cdot 10^{-06}$
132	0.0128	166	0.0010	206	$7.3872 \cdot 10^{-07}$
133	0.0141	167	0.0006	207	$8.1479 \cdot 10^{-06}$
134	0.0078	168	0.0007	208	1.9439
135	0.0243	169	0.0001	209	0.7829
136	0.0199	170	$9.2285 \cdot 10^{-06}$	210	$3.9263 \cdot 10^{-05}$
137	0.0089	171	0.0007	211	$5.2262 \cdot 10^{-06}$
138	0.0003	172	0.0003	213	$1.2459 \cdot 10^{-06}$
139	0.0823	173	0.0013	214	$3.0872 \cdot 10^{-07}$
140	0.0597	174	0.0011	215	$2.4862 \cdot 10^{-05}$
141	0.0024	175	0.0007	216	$2.5458 \cdot 10^{-05}$
142	0.0103	176	0.1682	217	$1.5799 \cdot 10^{-05}$
143	0.0439	177	0.0746	219	$2.0949 \cdot 10^{-07}$
144	0.0037	178	0.0778	224	$2.8667 \cdot 10^{-07}$
145	0.0084	179	0.0274	225	$1.0254 \cdot 10^{-06}$
146	0.0026	180	0.0928	226	0.0004
147	0.0021	181	0.1934	227	$6.6154 \cdot 10^{-08}$
148	0.0123	182	0.0210	228	$6.2846 \cdot 10^{-07}$
149	0.0016	183	0.0256	229	$2.3816 \cdot 10^{-06}$
150	0.0014	184	0.2415	230	$2.4587 \cdot 10^{-06}$
151	0.0012	185	0.0410	231	$1.5105 \cdot 10^{-06}$
152	0.0022	186	0.1429	232	$1.0254 \cdot 10^{-06}$
153	0.0003	187	0.1119	233	$7.8282 \cdot 10^{-07}$
154	0.0093	188	0.0698	234	$3.3077 \cdot 10^{-08}$
155	0.0032	189	0.1517	235	$1.4333 \cdot 10^{-07}$
156	0.0062	190	0.2729	236	$1.2128 \cdot 10^{-07}$
157	0.0043	191	0.1245	237	$3.3077 \cdot 10^{-08}$
158	0.0043	194	0.0011	239	$3.7410 \cdot 10^{-05}$
159	0.0148	195	0.0005	240	$5.5128 \cdot 10^{-08}$
160	0.0053	196	$1.4003 \cdot 10^{-05}$		
161	0.0099	197	$6.2405 \cdot 10^{-06}$		

Table C.2.: Distribution of the bytes of the Non-ASCII characters in the ClickBench dataset

List of Figures

2.1. Visualization of the shift of the pattern when mismatch at j occurs 7
2.2. Visualization of the shift by the Good Suffix Heuristics 14
2.3. Like-NFA for pattern `%abaa%` 18
2.4. Like-DFA for pattern `%abaa%`. A Σ transition consumes all characters which are not explicitly a character of any transition of the state. 20

3.1. General control flow of the generated code for the Original KMP algorithm. A green arrow is taken if the previous check evaluates to true, the red arrow otherwise. 23
3.2. General control flow of the generated code for the KMP algorithm with one loop. A green arrow is taken if the previous check evaluates to true, the red arrow otherwise. 24
3.3. Modified control flow of the generated code for the KMP algorithm with one loop to include optimizations. A green arrow is taken if the previous check evaluates to true, the red arrow otherwise. 26
3.4. General control flow of the generated code for the Original BM. A green arrow is taken if the previous check evaluates to true, the red arrow otherwise. . . . 27
3.5. General control flow of the generated code for the Fast BM. A green arrow is taken if the previous check evaluates to true, the red arrow otherwise. 29
3.6. Conceptual control flow of the generated code for the pattern $\alpha\%\beta\%\gamma\%\delta$. Each node encapsulates the whole functionality noted down in it. A green arrow is taken if the previous check or algorithm evaluates to true, the red arrow otherwise. 32

4.1. Distribution of the ASCII characters in the input texts of the TPC-H scheme . 35
4.2. Throughput for the different workloads of the KMP algorithms. For each algorithm, the left (hatched) bar presents the throughput of the interpreting version, the right one is the code generating version. At the left of the dashed vertical line, the Original KMP (Sect. 2.2.2 and 3.2.1) is shown with its optimization; on the right, the KMP with One Loop (Sect. 2.2.3 and 3.2.2) and its optimized algorithms are shown. (Higher is better.) 36
4.3. Throughput for the different workloads of the unoptimized OL-KMP and of the compressed OL-KMP. The left (hatched) bar presents the throughput of the interpreting version, the right one is the code generating version. (Higher is better.) 38

4.4. Throughput for the different workloads of the BM algorithms. For each algorithm, the left (hatched) bar presents the throughput of the interpreting version, the right one is the code generating version. We present the Original BM (Sect. 2.3.2 and 3.3.1), the Fast BM (Sect. 2.3.3 and 3.3.2), and the Blockwise BM (Sect. 2.3.4 and 3.3.3). (Higher is better.) 39
4.5. Throughput of the different workloads with the Automaton Approach for the DFA. We present the Direct Translation (Sect. 3.4.1) and the Blockwise Translation (Sect. 3.4.2). (Higher is better.) 42
4.6. Comparison of the best throughputs of the different code generating algorithms for the workloads. We present the best-performing algorithms for the corresponding workload. If available, the hatched bar presents the corresponding interpreting version of the algorithm. (Higher is better.) 43
4.7. Compilation and execution times for the best-performing algorithms on the TPC-H dataset with one thread. If available, the hatched bar is the interpreting version. The stacked bar is composed of the compile and execution time. (Lower is better.) 44
4.8. Compilation and execution times for the best-performing algorithms on the ClickBench dataset using 1 thread. If available, the hatched bar is the interpreting version. The stacked bar is composed of the compile and execution time. (Lower is better.) 46
4.9. Compilation and execution times for the best-performing algorithms on the ClickBench dataset using 8 threads. If available, the hatched bar is the interpreting version. The stacked bar is composed of the compile and execution time. (Lower is better.) 47
4.10. Compilation and execution times for the best-performing algorithms on the ClickBench dataset using 20 threads. If available, the hatched bar is the interpreting version. The stacked bar is composed of the compile and execution time. (Lower is better.) 48

5.1. Extended Like-NFA for pattern `%a_ba%` 50
5.2. Extended Like-DFA for pattern `%a_ba%` 51
5.3. Extended Like-NFA for pattern `%ab_a__%`. The state q_B and dashed edges are inserted during the code generation process to handle backtracking. 53

B.1. Step-by-step determinization of the Like-NFA of pattern `%abaa%` 62

List of Tables

2.1. Example texts and results for the pattern `red%green%blue`. **Bold** part of the text means an expected match with the pattern; *italics* means a mismatch. 6
2.2. Process to build the lps table for the pattern `abaa`. The bold numbers were already calculated in the previous iterations. 8
2.3. KMP search for pattern `abaa` in text `ababacabaa` 9
2.4. Regular and compressed LPS table for the pattern `abaabab`. The arrows visualize the "bouncing ball" if a mismatch at index 5 occurred. 10
2.5. Search for pattern `abaa` in text `ababacabaa` using Bad Character Heuristics . . 13
2.6. Negative shift with Bad Character Heuristics 13

A.1. Blockwise processing to search `0x44` in `0x4142434445464748` 59
A.2. Blockwise processing to search `0xA4` in `0x414243A4B5C64724` 60

C.1. Distribution of the ASCII characters in the ClickBench dataset 63
C.2. Distribution of the bytes of the Non-ASCII characters in the ClickBench dataset 64

Listings

2.1. Preprocessing of KMP 7
2.2. Original KMP (O-KMP) 8
2.3. KMP with One Loop (OL-KMP) 9
2.4. Compressed preprocessing of KMP 11
2.5. Blockwise search for ASCII character `c` 11
2.6. Preprocessing for the Bad Character Heuristics 13
2.7. Preprocessing for the Good Suffix Heuristics 14
2.8. Original BM (O-BM) 15
2.9. Fast BM (F-BM) 16
2.10. Algorithm to convert a Like-NFA into a Like-DFA 19
2.11. Determine the next lps state 20

3.1. Generated code for automaton in Fig. 2.4 for the pattern `%abaa%` 30
3.2. Generated blockwise code for automaton in Fig. 2.4 for the pattern `%abaa%` . . 31

5.1. Algorithm to convert an Extended Like-NFA into an Extended Like-DFA . . . 50
5.2. Generated code for the automaton in Fig. 5.3 for the pattern `%ab_a__%` 53

A.1. Blockwise Character Search for Non-ASCII characters 60

Bibliography

[ALX16] S. Agarwal, D. Liu, and R. Xin. *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop*. 2016. URL: https://www.databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html. (accessed: 20.12.2022).

[BM77] R. S. Boyer and J. S. Moore. "A Fast String Searching Algorithm." In: *Commun. ACM* 20.10 (1977), pp. 762–772. DOI: 10.1145/359842.359859.

[CP91] M. Crochemore and D. Perrin. "Two-Way String Matching." In: *J. ACM* 38.3 (1991), pp. 651–675. DOI: 10.1145/116825.116845.

[Dia+13] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. "Hekaton: SQL server's memory-optimized OLTP engine." In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. Ed. by K. A. Ross, D. Srivastava, and D. Papadias. ACM, 2013, pp. 1243–1254. DOI: 10.1145/2463676.2463710.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997. ISBN: 0-521-58519-8. DOI: 10.1017/cbo9780511574931.

[Hav97] P. Havlak. "Nesting of Reducible and Irreducible Loops." In: *ACM Trans. Program. Lang. Syst.* 19.4 (1997), 557–567. ISSN: 0164-0925. DOI: 10.1145/262004.262005.

[Hof16] T. Hoff. *Code Generation: The Inner Sanctum Of Database Performance*. 2016. URL: http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html. (accessed: 20.12.2022).

[Hop71] J. E. Hopcroft. *An n Log n Algorithm for Minimizing States in a Finite Automaton*. Tech. rep. Stanford, CA, USA, 1971.

[Iso] *Information Technology - Database Language SQL*. Standard SQL-92. Massachusetts, USA: International Organization for Standardization, 1992. https://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt (accessed: 03.12.2022).

[KLN21] T. Kersten, V. Leis, and T. Neumann. "Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra." In: *VLDB J.* 30.5 (2021), pp. 883–905. DOI: 10.1007/s00778-020-00643-4.

[KMP77] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. "Fast Pattern Matching in Strings." In: *SIAM J. Comput.* 6.2 (1977), pp. 323–350. DOI: 10.1137/0206024.

[KN11] A. Kemper and T. Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots." In: *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*. Ed. by S. Abiteboul, K. Böhm, C. Koch, and K. Tan. IEEE Computer Society, 2011, pp. 195–206. DOI: 10.1109/ICDE.2011.5767867.

[Lan01a] H. W. Lang. *Boyer-Moore Algorithm algorithm*. 2001. URL: https://www.inf.hs-flensburg.de/lang/algorithmen/pattern/bmen.htm. (updated by author: 04.06.2018, accessed: 06.12.2022).

[Lan01b] H. W. Lang. *Knuth-Morris-Pratt Algorithm*. 2001. URL: https://www.inf.hs-flensburg.de/lang/algorithmen/pattern/kmpen.htm. (updated by author: 04.06.2018, accessed: 18.11.2022).

[Mic] *.NET: Compilation and Reuse in Regular Expressions*. 2021. URL: https://learn.microsoft.com/en-us/dotnet/standard/base-types/compilation-and-reuse-in-regular-expressions. (accessed: 06.01.2023).

[Neu11] T. Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware." In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550. DOI: 10.14778/2002938.2002940.

[NF20] T. Neumann and M. J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance." In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[Per+09] G. Perez, Y. P. Mejia, I. Olmos, J. A. Gonzalez, P. Sánchez, and C. Vázquez. "An Automaton for Motifs Recognition in DNA Sequences." In: *MICAI 2009: Advances in Artificial Intelligence, 8th Mexican International Conference on Artificial Intelligence, Guanajuato, Mexico, November 9-13, 2009. Proceedings*. Ed. by A. H. Aguirre, R. M. Borja, and C. A. R. García. Vol. 5845. Lecture Notes in Computer Science. Springer, 2009, pp. 556–565. DOI: 10.1007/978-3-642-05258-3_49.

[Pos] PostgreSQL Development Team. *PostgreSQL Documentation - Pattern Matching*. URL: https://www.postgresql.org/docs/current/functions-matching.html. (accessed: 18.11.2022).

[Ryt80] W. Rytter. "A Correct Preprocessing Algorithm for Boyer-Moore String-Searching." In: *SIAM J. Comput.* 9.3 (1980), pp. 509–512. DOI: 10.1137/0209037.

[Tho68] K. Thompson. "Regular Expression Search Algorithm." In: *Commun. ACM* 11.6 (1968), pp. 419–422. DOI: 10.1145/363347.363387.

[Vog+18] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then. "Get Real: How Benchmarks Fail to Represent the Real World." In: *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*. Ed. by A. Böhm and T. Rabl. ACM, 2018, 1:1–1:6. DOI: 10.1145/3209950.3209952.

[WL14] S. Wanderman-Milne and N. Li. "Runtime Code Generation in Cloudera Impala." In: *IEEE Data Eng. Bull.* 37.1 (2014), pp. 31–37.