Compression-Aware LIKE: Matching Patterns in the FSST Domain

Călin-George Pop
calin.george.pop@tum.de
Technical University of Munich
Munich, Bavaria, Germany

Adrian Riedl
adrian.riedl@in.tum.de
Technical University of Munich
Munich, Bavaria, Germany

Thomas Neumann
thomas.neumann@in.tum.de
Technical University of Munich
Munich, Bavaria, Germany

Abstract

Database users frequently utilise string data types for general-purpose storage, even when more specialised alternatives are available [18]. Thus, modern analytical database systems rely on lightweight string compression schemes, such as Fast Static Symbol Table (FSST) [2] or OnPair [6], to reduce memory footprint. However, evaluating SQL LIKE predicates on compressed strings typically requires full decompression, negating the benefits of compression.

We present an approach that reverses this paradigm: instead of decompressing data to match patterns, we *compress the pattern* by constructing finite automata that parse FSST-encoded symbol sequences directly. Our technique partitions LIKE patterns into prefixes, suffixes, and substrings, building specialised automata for each, while considering the properties of FSST. We provide interpreted and compiled implementations along with optimisations, including early rejection and SIMD-accelerated self-loop traversal.

Evaluation on TPC-H, StackOverflow, and IMDB datasets demonstrates speedups of 2.5–17× over decompression-based pattern matching, suggesting that co-designing compression schemes and query operators, already successful for integer data, represents a promising direction for string-dominated analytical workloads.

CCS Concepts

• **Theory of computation** → *Automata extensions*; **Data compression**; • **Information systems** → **Database query processing**.

Keywords

Text Compression, Pattern Matching, SIMD, JIT Compilation

ACM Reference Format:
Călin-George Pop, Adrian Riedl, and Thomas Neumann. 2026. Compression-Aware LIKE: Matching Patterns in the FSST Domain. In *22nd International Workshop on Data Management on New Hardware (DaMoN '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3789237.3809128

1 Introduction

Modern database systems increasingly rely on lightweight compression to mitigate memory bandwidth bottlenecks. Data movement has become one of the most expensive operations in data centres [10, 12], particularly for analytical workloads, where memory bandwidth frequently limits query throughput. Lightweight compression schemes address this challenge by reducing the volume

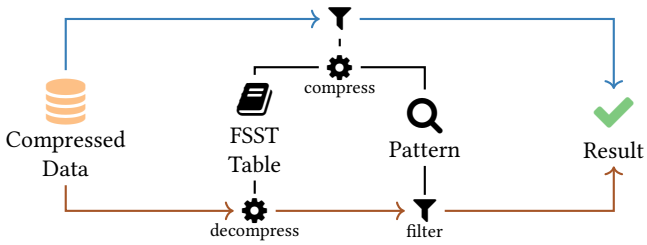


Figure 1: Comparison of pattern matching strategies on FSST-compressed data. The conventional approach (bottom) decompresses data before filtering, whereas our method (top) builds an automaton that directly operates on compressed representations, eliminating materialisation overhead.

of data transferred between memory hierarchies while enabling direct execution of operations on compressed representations.

For integer data, extensive research has established effective compression schemes and query processing techniques that operate natively on compressed encodings. Frame-of-reference encoding and delta encoding enable substantial reductions in memory traffic while supporting direct predicate evaluation and aggregation [5, 17]. The key insight enabling these gains is that lightweight integer compression preserves sufficient structural information to permit filtering values without uncompressing them.

In practice, however, roughly half of the columns stored in database systems are string datatypes [18]. Database systems therefore employ specialised string compression techniques, such as dictionary compression, OnPair [6] and FSST (Fast Static Symbol Table) [2]. Dictionary compression does not decompress strings when filtering, as patterns are checked within the dictionary of values. However, dictionary compression is effective only when the number of unique values is small. To overcome this, FSST constructs a compact symbol table by identifying frequent byte sequences and mapping them to single-byte codes, achieving high compression ratios while enabling fast random access; that is, individual values may be decompressed without accessing entire blocks.

Despite the multiple string-compression algorithms used by database systems, filter predicates on compressed strings typically follow a decompress-then-operate paradigm. As illustrated in Figure 1, the conventional approach for evaluating SQL `LIKE` patterns requires full string decompression before filtering, materialising uncompressed data in memory and negating the memory bandwidth benefits provided by compression.

This paper introduces our approach that reverses this paradigm for FSST-compressed data: instead of decompressing data to match patterns, we construct a finite automaton and parse it directly using the compressed representations. By avoiding the decompression

DaMoN '26, Bengaluru, India

ACM ISBN 979-8-4007-2455-8/2026/05
https://doi.org/10.1145/3789237.3809128

step and leveraging JIT compilation, we achieve up to 17× improvement in query throughput.

2 Background

Fast Static Symbol Table [2] is a lightweight compression algorithm that replaces frequently occurring substrings of length at most eight with single-byte codes, organised in a symbol table. We denote a sequence of FSST symbols with $\langle s_0, \ldots, s_{m-1} \rangle$, where s_i represents the s_i^{th} symbol. We reserve the byte 255 for escaping a single byte, so $\langle 255, c \rangle$ represents the literal byte c. In the following, we present the important properties guaranteed by FSST.

Property (Greedy Encoding). *Given a symbol table, encoding a string "$\sigma_0 \ldots \sigma_{k-1}$" involves greedily picking the longest symbol $s =$ "$\sigma_0 \ldots \sigma_{l-1}$" completely contained within the string ($l \leq k$), then re-applying this algorithm for the rest of the uncompressed string "$\sigma_l \ldots \sigma_{k-1}$".*

Definition (Valid Sequence). *Given a symbol table, a sequence $\mathbf{s} = \langle s_0, \ldots, s_{m-1} \rangle$ is valid if there is a string σ that compresses to $\mathbf{s}$ using FSST.*

Example. *For the symbol table $sym_0 =$ "AB", $sym_1 =$ "C" and $sym_2 =$ "ABC", the sequence $\mathbf{s} = \langle 0, 1 \rangle$ decompresses to $\sigma =$ "ABC". However, σ compresses to $\mathbf{s}^{(0)} = \langle 2 \rangle$ due to the greedy encoding property of FSST. As a consequence, $\mathbf{s}$ is not a valid sequence.*

To ensure the correctness of the algorithms presented later, we state two important properties. The latter property, while not explicitly stated in the original paper, follows naturally through swapping symbols within the symbol table.

Property (Unique Prefix). *No two symbols have the same three-byte prefix.*

Property (Index-Suffix Divergence). *The last byte of the i^{th} symbol is different from i.*

For a `LIKE` pattern $\mathbf{p}$, there may be multiple different FSST sequences matching $\mathbf{p}$. Therefore, to accept them simultaneously, we construct a deterministic finite automaton, which we subsequently parse using the compressed string representation. We provide its formal definition as follows:

Definition (Deterministic Finite Automaton). *A deterministic finite automaton is a quintuple $(Q, \Sigma, \delta, S, \mathcal{F})$, where Q is the finite set of states, Σ is the finite alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, S is the start state and $\mathcal{F}$ is the set of accept states. To simplify parsing, we define $\varepsilon \in Q$ as the error state and, for each $q \in Q$, we define $\delta_{def}(q) \in Q$ as the state to which q transitions when a transition is not explicitly defined; unless specified otherwise, $\delta_{def}(q) = \varepsilon$. We call $\delta_{def}(q)$ the default transition of the state q.*

3 Automaton Construction

In this section, we present algorithms for constructing finite automata that recognise valid symbol sequences matching individual subpatterns of a `LIKE` expression, and then for connecting them together. Considering only the '%' wildcard character that matches any number of characters, we categorise subpatterns $\mathbf{p}$ into three distinct classes based on their wildcard boundaries: (1) *start patterns* of the form "$p_0 \ldots p_{n-1}$%", which match prefixes; (2) *middle patterns* of the form "%$p_0 \ldots p_{n-1}$%", which match arbitrary substrings; and (3) *end patterns* of the form "%$p_0 \ldots p_{n-1}$", which match suffixes. For each pattern category, we define $\mathcal{S}$ as the set of all minimal valid symbol sequences $\mathbf{s} = \langle s_0, \ldots, s_{m-1} \rangle$ such that the decompressed representation of $\mathbf{s}$ matches the subpattern $\mathbf{p}$. Our algorithm constructs an automaton that accepts precisely the sequences in $\mathcal{S}$.

3.1 End Patterns: "%$p_0 \ldots p_{n-1}$"

End patterns perform suffix matching by processing the compressed string backwards from its end. The corresponding automaton contains a single start state S and transitions to an end state E_i upon accepting a sequence $s \in \mathcal{S}$, where $i \in \{0, \ldots, 8\}$ denotes the character position within symbol s_0 ($\langle 255, s_1 \rangle$) at which the first pattern character p_0 aligns. The index i ranges from 0 to 8 because FSST symbols encode at most eight characters.

For $\mathbf{s} = \langle s_0, \ldots, s_{m-1} \rangle \in \mathcal{S}$ with $s_0 \neq 255$, decoding s_0 ends with the character p_k of the pattern $\mathbf{p}$. However, the index-suffix divergence property of FSST yields $s_0 \neq p_k$; therefore, decoding $\langle 255, s_0, \ldots, s_{m-1} \rangle$ ends with $\mathbf{p}[k+1:]$, but does not end with $\mathbf{p}[k:]$. As a consequence, there is no sequence $s' \in \mathcal{S}$ ending with $\langle 255, s_0, \ldots, s_{m-1} \rangle$. This ensures that the automaton built greedily is correct. We partition $\mathcal{S}$ into disjoint subsets $\mathcal{S}_k$ based on the alignment position k of pattern $\mathbf{p}$ relative to the first symbol s_0 ($\langle 255, s_1 \rangle$).

For $k = 0$, the first character of s_0 ($\langle 255, s_1 \rangle$) aligns with p_0. Using the greedy encoding property (Section 2), $\mathcal{S}_0$ contains exactly one element: the compressed representation of $\mathbf{p}$. For $k > 0$, symbol s_0 ends with the prefix $\mathbf{p}[:k]$, while the remaining substring $\mathbf{p}[k:]$ is compressed independently. Consequently, we define $\mathcal{S}_k$ as $\mathcal{S}_k = \left\{ i \mid i^{th} \text{ symbol ends with, but is not equal to, } \mathbf{p}[:k] \right\} \times \{\mathbf{c}^{(k)}\}$,

$$\text{where } \mathbf{c}^{(k)} = \begin{cases} \text{compress}(\mathbf{p}[k:]) & \text{if } k < n \\ \langle\rangle & \text{if } k = n \\ \emptyset & \text{if } k > n \end{cases} \quad \text{and } j_k = |\mathbf{c}^{(k)}|.$$

We construct the automaton as a suffix tree. For $k = 0$, the tree builds a single path corresponding to the unique sequence $\mathbf{c}^{(0)} \in \mathcal{S}_0$. For $k > 0$, all $\mathbf{s} \in \mathcal{S}_k$ satisfy $\mathbf{s} = \langle i, c_0^{(k)}, \ldots, c_{j_k-1}^{(k)} \rangle$, where sym_i ends with, but is not equal to, $\mathbf{p}[:k]$. The construction algorithm follows the initial trie-building phase of Aho-Corasick [1], with the distinction that sequences $\mathbf{s} \in \mathcal{S}$ transition to an end state E_j through i rather than to a regular state within the automaton. Since symbols encode at most eight characters, we apply the algorithm for all $k \in [0, \min(n, 7)]$.

Due to FSST using 255 as an escape character, indicating that the subsequent byte should be interpreted literally, if we finish parsing a sequence backwards in 255 and 255 remains unparsed, we introduce false positives as we misinterpret the k^{th} byte for the k^{th} symbol. To solve this, we introduce the concept of a *pseudo-end state*: a state q is called a *pseudo-end state* if it accepts when the input is exhausted, but possesses both a default transition to an accept state and a non-empty subset of transitions leading to non-accept states.

For this, we construct nine pseudo-end states E_i' for $0 \leq i \leq 8$, where the transition function satisfies $\delta(E_i', x) = E_i$ for all $x \neq 255$ and $\delta(E_i', 255) = \varepsilon$. Sequences $s \in \mathcal{S}$ now transition to the

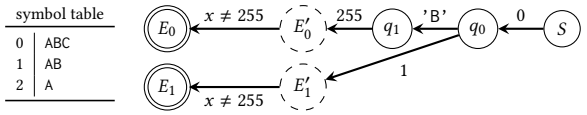


Figure 2: Automaton for the end pattern '%BABC' with E_i representing end states and E'_i denoting pseudo-end states.

mirrored pseudo-end states E'_i rather than to the accept states E_i themselves. During backwards automaton traversal, upon reaching E'_i, we perform a one-byte lookahead when possible to verify that the preceding byte is not an escape character.

Figure 2 illustrates a backwards-matching automaton for the suffix '%BABC'. For $k = 0$, "BABC" encodes greedily as $\mathbf{c}^{(0)} = \langle 255, \texttt{'B'}, 0\rangle$. To reach the pseudo-end state E'_0 via $\mathbf{c}^{(0)}$, we create states q_0 and q_1, matching the suffixes $\langle 0\rangle$ and $\langle \texttt{'B'}, 0\rangle$ respectively, and set $\delta(S, 0) \leftarrow q_0$, $\delta(q_0, \texttt{'B'}) \leftarrow q_1$, and $\delta(q_1, 255) \leftarrow E'_0$.

For $k = 1$, sym_0 is suffixed by "B", and "ABC" encodes as $\mathbf{c}^{(1)} = \langle 1\rangle$, yielding $\mathcal{S}_1 = \{\langle 1, 0\rangle\}$. Since $q_0 = \delta(S, 0)$ is already defined, reaching E'_1 requires only the transition $\delta(q_0, 1) \leftarrow E'_1$. For $k \geq 2$, no symbol is suffixed by $\mathbf{p}[:k]$, so no further transitions are needed.

3.2 Start Patterns: "$p_0 \dots p_{n-1}$%"

We parse a sequence $\mathbf{s}$ from the unique start state S_0 and, if $\mathbf{s} \in \mathcal{S}$, we accept it by transitioning to an end state $E_i, 0 \leq i \leq 8$ based on the final symbol s_{m-1}: if s_{m-1} is escaped (i.e., $m \geq 2$ and $s_{m-2} = 255$), then $i = 0$; otherwise, i denotes the index of the first unconsumed character within s_{m-1}.

Let $S(\mathbf{p})$ be the set of all valid, minimal sequences that, when decompressed, are prefixed by $\mathbf{p}$. For simplicity, the set of accepted sequences is a superset of $S(\mathbf{p})$ rather than precisely $S(\mathbf{p})$. This approach guarantees correctness by ensuring no false positives are added during matching. The construction algorithm depends on the pattern length n and is divided into three cases: $n = 1$, $n = 2$ and $n \geq 3$.

We first define the set $\mathcal{P}ref = \{\langle i\rangle | i^{th}$ symbol starts with $\mathbf{p}\}$, which contains all mono-symbolic sequences whose decompression is prefixed by $\mathbf{p}$. Regardless of n, any sequence $\mathbf{c} \in \mathcal{P}ref$ is valid and minimal; we integrate these sequences in the automaton by setting $\delta(S_0, c_0) \leftarrow E_n$ for all $\langle c_0\rangle \in \mathcal{P}ref$. We remain to integrate the sequences $\mathbf{c} \in S(\mathbf{p}) \setminus \mathcal{P}ref$ in the automaton.

Case $n = 1$. Consider a sequence $\mathbf{c} \in S(\mathbf{p})$. By definition, $\mathbf{c}$ is minimal and prefixed by $\mathbf{p}$. If "p_0" is a standalone symbol, $|\mathbf{c}| = 1$ by the greedy encoding property of FSST and the fact that symbols are not empty, directly implying $\mathbf{c} \in \mathcal{P}ref$. Otherwise, $\mathbf{c} = \langle 255, p_0\rangle$ is a valid sequence prefixed by "p_0" minimally; to add this sequence, we introduce the transitions $\delta(S_0, 255) \leftarrow q$ and $\delta(q, p_0) \leftarrow E_0$, where q is a newly created state.

Case $n = 2$. If "p_0p_1" exists as a standalone symbol, the greedy encoding property of FSST implies that any $\mathbf{c} \in S(\mathbf{p})$ satisfies $|\mathbf{c}| = 1$, so $\mathbf{c} \in \mathcal{P}ref$ and no additional transitions are required. When "p_0p_1" is not a standalone symbol, then the sequences $\mathbf{c} \in S(\mathbf{p}) \setminus \mathcal{P}ref$ (i.e., the sequences that we are left to integrate in the automaton) satisfy that c_0 ($\langle 255, c_1\rangle$) decompresses as "p_0". Therefore, we first build an automaton having the start state S'_0 that matches the prefix "p_1", and then connect S_0 to S'_0 using the encoding of "p_0".

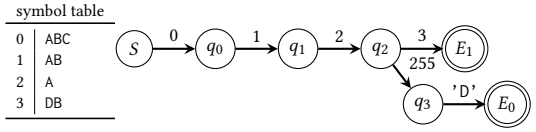


Figure 3: Automaton for the start pattern 'ABCABAD%' with E_i representing the end states.

If "p_0" is the i^{th} symbol for some i, then any $\mathbf{c} \in S(\mathbf{p}) \setminus \mathcal{P}ref$ is of the form $\mathbf{c} = \langle i\rangle \times \mathbf{c}'$, where $\mathbf{c}'$ is prefixed minimally by "p_1", implying $\mathbf{c}' \in S(\mathbf{p}[1:])$. The reverse is not necessarily true; $\mathbf{c}' \in S(\mathbf{p}[1:])$ does not imply $\langle i\rangle \times \mathbf{c}' \in S(\mathbf{p})$, as $\langle i\rangle \times \mathbf{c}'$ may be an invalid sequence. We complete the construction of the automaton by simply adding the transition $\delta(S_0, i) \leftarrow S'_0$.

If neither "p_0p_1" nor "p_0" are standalone symbols, p_0 may be escaped, and any $\mathbf{c} \in S(\mathbf{p}) \setminus \mathcal{P}ref$ is of the form $\mathbf{c} = \langle 255, p_0\rangle \times \mathbf{c}'$, with $\mathbf{c}' \in S(\mathbf{p}[1:])$. Similarly, $\mathbf{c}' \in S(\mathbf{p}[1:])$ does not imply $\langle 255, p_0\rangle \times \mathbf{c}' \in S(\mathbf{p})$. The automaton is then completed by adding the transitions $\delta(S_0, 255) \leftarrow q$ and $\delta(q, p_0) \leftarrow S'_0$, where q is a newly created state. While in the last two subcases the automaton may accept invalid sequences, such sequences never occur in real compressed data; therefore, no false positives arise during matching.

Case $n \geq 3$. By the unique prefix property of FSST, at most one symbol has the prefix "$p_0p_1p_2$". If there exists a (unique) symbol $sym_i =$ "$p_0 \dots p_{l-1}$", $3 \leq l \leq n$, then every $\mathbf{c} \in S(\mathbf{p})$ is of the form $\mathbf{c} = \langle i\rangle \times \mathbf{c}'$ by the greedy encoding property, where $\mathbf{c}' \in S(\mathbf{p}[l:])$. Therefore, we first build the sub-automaton for the prefix $\mathbf{p}[l:]$ having the start state S'_0, and then add the transition $\delta(S_0, i) \leftarrow S'_0$.

If no such symbol exists, then, for $\mathbf{c} \in S(\mathbf{p}) \setminus \mathcal{P}ref$, c_0 ($\langle 255, c_1\rangle$) decompresses to a non-empty prefix of "p_0p_1". If "p_0p_1" is the j^{th} symbol for some j, the greedy encoding property then yields $c_0 = j$, and $\mathbf{c}[1:]$ is prefixed by $\mathbf{p}[2:]$ minimally. As a consequence, $\mathbf{c} = \langle j\rangle \times \mathbf{c}'$, where $\mathbf{c}' \in S(\mathbf{p}[2:])$. We construct the automaton by initially building the sub-automaton for the prefix $\mathbf{p}[2:]$ having the start state S'_0, and then adding the transition $\delta(S_0, j) \leftarrow S'_0$.

If neither of the above symbols exists, then c_0 ($\langle 255, c_1\rangle$) decompresses to "p_0" and the rest of the sequence $\mathbf{c}$ is prefixed by $\mathbf{p}[1:]$ minimally. Therefore, we first build the sub-automaton for the prefix $\mathbf{p}[1:]$ having the start state S'_0 and consider two final subcases. If "p_0" is the k^{th} symbol for some k, the greedy encoding property yields $c_0 = k$ and, as a consequence, $\mathbf{c} = \langle k\rangle \times \mathbf{c}'$, where $\mathbf{c}' \in S(\mathbf{p}[1:])$; we finish the process of building the automaton by adding the transition $\delta(S_0, k) \leftarrow S'_0$. If "$p_0$" is not a standalone symbol, then p_0 may be escaped and $\mathbf{c} = \langle 255, p_0\rangle \times \mathbf{c}'$ with $\mathbf{c}' \in S(\mathbf{p}[1:])$, and we add the transitions $\delta(S_0, 255) \leftarrow q$ and $\delta(q, p_0) \leftarrow S'_0$, where q is a newly created state.

Figure 3 illustrates the automaton constructed for the prefix 'ABCABAD%'. Since $sym_0 =$ "ABC", we create the state q_0 and set $\delta(S, 0) \leftarrow q_0$. We then construct the automaton for the remaining pattern "ABAD" with start state q_0. Neither "ABA" nor "ABAD" is a symbol, but $sym_1 =$ "AB"; accordingly, we create the state q_1 and set $\delta(q_0, 1) \leftarrow q_1$.

The sub-automaton matching the prefix "AD" is rooted at q_1. Since "AD" is not a symbol but $sym_2 =$ "A", we create the state q_2 and set $\delta(q_1, 2) \leftarrow q_2$. The sub-automaton matching the prefix "D" is rooted at q_2. Because sym_3 begins with "D", we have $\langle 3\rangle \in \mathcal{P}ref$ and set $\delta(q_2, 3) \leftarrow E_1$. The character "D" may also appear

escaped; we therefore create the state q_3 and add the transitions $\delta(q_2, 255) \leftarrow q_3$ and $\delta(q_3, \texttt{'D'}) \leftarrow E_0$.

3.3 Middle Patterns: "$\%p_0 \dots p_{n-1}\%$"

A sequence **s** begins parsing the automaton built for a middle pattern at start state $S_i, 0 \le i \le 8$, where i denotes the position within symbol s_0 ($\langle 255, s_1 \rangle$) at which character p_0 aligns. If $\mathbf{s} \in \mathcal{S}$, we accept the sequence by transitioning to an end state $E_j, 0 \le j \le 8$, where $j = 0$ if the final symbol s_{m-1} is escaped (i.e., $m \ge 2$ and $s_{m-2} = 255$), and otherwise j denotes the index of the first unconsumed character within s_{m-1}.

A key distinction from prefix and suffix automata is that middle-pattern matching must handle partial-match failures during traversal. Consequently, failed transitions default to the start state S_0 rather than the reject state ε. We decompose the automaton construction into three phases to simplify the building process.

3.3.1 Initial Phase. The first phase constructs the sub-automata rooted at each start state S_i. We further partition the construction into three cases based on pattern alignment and sequence length.

Case 1: Alignment at sequence start ($i = 0$). If the first character of s_0 ($\langle 255, s_1 \rangle$) aligns with p_0, then decompressing **s** yields a string prefixed by the pattern **p**. We therefore construct the prefix automaton for **p** having the start state S_0 using the algorithm from the previous section.

Case 2: Single-symbol match ($|\mathbf{s}| = 1$). If the sequence consists of a single symbol containing **p** as a substring, we establish direct transitions to the corresponding end states: for each symbol sym_c and index $i > 0$ satisfying $\text{sym}_c[i : i + n] = \mathbf{p}$, set $\delta(S_i, c) \leftarrow E_{i+n}$.

Case 3: Multi-symbol match ($|\mathbf{s}| \ge 2$). For sequences spanning multiple symbols, symbol s_0 ends with, but is not equal to, a prefix $\mathbf{p}[: l]$ for some $1 \le l < 8$, while the rest of the sequence $\mathbf{s}[1 :]$ decompresses to a string beginning with $\mathbf{p}[l :]$. We define the suffix set $\mathcal{S}uff^{(l)} = \{i \mid i^{th}$ symbol ends with, but is not equal to, $\mathbf{p}[: l]\}$. For each non-empty $\mathcal{S}uff^{(l)}$, we build the prefix automaton for $\mathbf{p}[l :]$ having the start state S'. For each $i \in \mathcal{S}uff^{(l)}$, we add the transition $\delta(S_m, i) \leftarrow S'$, where $m = |\text{sym}_i| - l$ denotes the index within sym_i aligned with p_0.

Escape character handling. If S_0 does not transition through the escape byte 255, $\delta(S_0, 255)$ defaults to S_0. To prevent the wrong interpretation of escaped sequences $\langle 255, c \rangle$ as the symbol sym_c, we introduce a sink state q_{esc} with no outgoing transitions, and we set $\delta(S_0, 255) \leftarrow q_{esc}$. The sequence $\langle 255, c \rangle$ now triggers $\delta(S_0, 255) = q_{esc}$, and then $\delta(q_{esc}, c)$ defaults back to S_0.

Figure 4a shows the initial automaton built for the middle pattern `'%AAAC%'`. We first construct the sub-automaton for the prefix `'AAAC%'` having the start state S_0. Afterwards, we compute $\mathcal{S}uff^{(1)} = \{0, 1\}$, build the sub-automaton for the prefix `'AAC%'` having the start state q_1, and add the transitions $\delta(S_1, 0) \leftarrow q_1$ and $\delta(S_1, 1) \leftarrow q_1$. Finally, because S_0 does not currently transition through 255, we create the sink state q_{esc} and we add the transition $\delta(S_0, 255) \leftarrow q_{esc}$.

3.3.2 Link/Split Phase. The second phase establishes consistency between start states through linking and splitting operations. Consider the strings σ_i and σ_j obtained by decompressing a sequence **s** starting from the i^{th} and j^{th} characters of s_0, respectively, where $i > j$. By construction, σ_i is a substring of σ_j. Consequently, if σ_i contains some pattern **p**, then σ_j must also contain **p**. This implies that any sequence that reaches an end state from S_i must also reach an end state from S_j, to ensure the correctness of the algorithm.

We enforce this invariant through two complementary operations. Consider two start states S_i, S_j, indexed such that $i > j$, and a symbol sym_c satisfying $\delta(S_i, c) = q$.

Linking. If S_j currently has no transition through c (i.e., $\delta(S_j, c)$ is undefined), we set $\delta(S_j, c) \leftarrow q$. This operation, called *linking*, copies all accepting paths from S_i through c to the start state S_j.

Splitting. If $\delta(S_j, c) = q_0$ is already defined, we must copy all paths starting from q into the sub-automaton rooted at q_0. We implement this *splitting* operation via breadth-first search initialised with queue entry (q_0, q), where the first component denotes the destination state and the second the source.

At each step, for the pair $(q_{\text{dest}}, q_{\text{src}})$, we first enqueue the pair (q'_{dest}, q'_{src}) for all c such that $\delta(q_{\text{dest}}, c) = q'_{\text{dest}}$, $\delta(q_{\text{src}}, c) = q'_{\text{src}}$ and q'_{dest}, q'_{src} are not end states. Thereafter, for all c for which $\delta(q_{\text{src}}, c) = q'$ but $\delta(q_{\text{dest}}, c)$ is undefined, set $\delta(q_{\text{dest}}, c) \leftarrow q'$.

Efficient implementation. We perform all linking and splitting operations in a single pass by iterating through the start states in reverse order from S_8 to S_0. We maintain a mapping $\mathcal{M}$ from each c to the state $\delta(S_j, c)$, where j is minimal among all previously processed start states. When processing S_i, we first copy all the paths from $\mathcal{M}$ by performing linking and splitting operations, and then update $\mathcal{M}$ with the transitions of S_i.

In Figure 4b and 4c, we present the splitting and linking processes of the start states of the previously built automaton. The mapping $\mathcal{M}$ is empty when S_1 is processed; therefore, no splitting and linking operations are performed for S_1. Afterwards, $\mathcal{M}$ is updated with the transitions of S_1: $\mathcal{M}[0] \leftarrow \delta(S_1, 0) = q_1$ and $\mathcal{M}[1] \leftarrow \delta(S_1, 1) = q_1$. The splitting phase of S_0 involves copying the paths starting from $q_1 = \mathcal{M}[0]$ into the sub-automaton rooted in $q_0 = \delta(S_0, 0)$ by adding $\delta(q_0, 0) \leftarrow \delta(q_1, 0) = q_2$. For the linking phase of S_0, we simply add the transition $\delta(S_0, 1) \leftarrow \mathcal{M}[1] = q_1$, as $\delta(S_0, 1)$ is initially undefined.

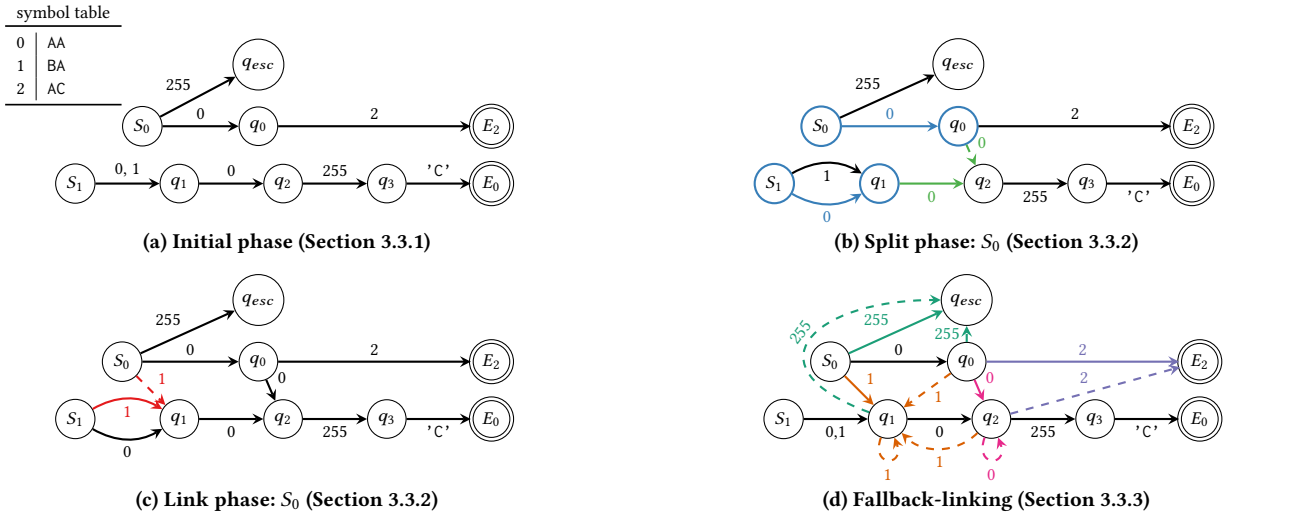
3.3.3 Fallback-Linking Phase. The third phase introduces failure transitions to ensure the constructed automaton is deterministic. Currently, parsing failures lead only to S_0; this is not always the intended behaviour, as we might have already matched a prefix of another state even though the current symbol is a mismatch.

Each state q in the automaton corresponds to a unique symbol sequence prefix of the form $\langle \langle s_0^{(0)} \mid \cdots \rangle, s_1, s_2, \dots, s_{k-1} \rangle$, where the first position may accept multiple alternative symbols, while subsequent positions are deterministic. For example, Table 1 displays the prefix sequences corresponding to states in Figure 4c.

Table 1: Prefixes and Trailing States for Figure 4c

State	q_0	q_1	q_2	q_3	q_{esc}
Prefix	$\langle 0 \rangle$	$\langle \langle 0 \mid 1 \rangle \rangle$	$\langle \langle 0 \mid 1 \rangle, 0 \rangle$	$\langle \langle 0 \mid 1 \rangle, 0, 255 \rangle$	$\langle 255 \rangle$
Trailing State	S_0	S_0	q_0	q_{esc}	NULL

We create failure transitions through a *fallback mechanism*. When a state q consumes the character c for which no transition has been



(a) Initial phase (Section 3.3.1)

(b) Split phase: S_0 (Section 3.3.2)

(c) Link phase: S_0 (Section 3.3.2)

(d) Fallback-linking (Section 3.3.3)

Figure 4: Three-phase construction of the automaton for the middle pattern '%AAAC%'. (a) The initial phase builds the initial sub-automata starting in S_i. (b) The split phase ensures that all the paths starting at S_1 (green) are copied into S_0 along BFS paths (blue) when a sub-path already exists. (c) The link phase ensures that all paths starting at S_1 (red) are copied into S_0 when no sub-path already exists. (d) Fallback-linking phase adds fallback transitions (dashed) to reset to the correct state on mismatches.

defined yet, rather than resetting to S_0, we identify the longest suffix of the prefix of q such that there exists a state q_0 with this prefix and $\delta(q_0, c)$ is defined; we then transition to $\delta(q_0, c)$. Formally, we find the minimal $k_1 > 0$ such that there is a state q' with prefix $\langle s_{k_1}, \ldots, s_{k-1}, c\rangle$, and we add the transition $\delta(q, c) \leftarrow q'$. The state q' is called the *fallback state* of q through c. This construction adapts the failure function in the Aho-Corasick algorithm to the problem at hand.

The construction of fallback transitions is reduced to a transition-copying operation. Following the approach of Aho-Corasick [1], we observe that the fallback transition of state q through c is equivalent to $\delta(q_{trail}, c)$ for a *trailing state* q_{trail} independent of c, provided that all fallback transitions of q_{trail} have already been computed.

We implement this using breadth-first search, which guarantees that each state is visited exactly once and processes states in order of their prefix length. The algorithm maintains a queue of pairs of states (q, q_{trail}), where q_{trail} is the trailing state of q.

Initialisation. For each state $q \neq q_{esc}$ satisfying $\delta(S_i, c) = q$ for some start state S_i and some c, we enqueue the pair (q, S_0) once. If we had enqueued (q_{esc}, S_0), q_{esc} would have copied the transitions of S_0, so $\langle 255, c\rangle$ would have transitioned to $\delta(S_0, c)$ when defined, treating the byte c as sym_c. To avoid this, we treat this state differently and enqueue $(q_{\text{esc}}, \texttt{NULL})$.

Processing. At each iteration, for the pair (q, q_{trail}), we initially enqueue the pair $(q', \delta(q_{\text{trail}}, c))$ for each c satisfying $\delta(q, c) = q'$, with q' not previously visited; in this case, $\delta(q_{trail}, c)$ may be a default or a fallback transition. Afterwards, for each c for which $\delta(q, c)$ is undefined but $\delta(q_{\text{trail}}, c) = q''$, set $\delta(q, c) \leftarrow q''$.

Correctness. The BFS traversal order ensures that, when processing a state q, its trailing state q_{trail} has already been processed, since q_{trail} has a strictly shorter prefix than q. This guarantees that all fallback transitions of q_{trail} are available when traversing q.

Figure 4d presents the complete middle pattern automaton with all fallback transitions created, and we list the trailing states of all states in Table 1. Initially, q_{esc} has the trailing state NULL and the states $q_0 = \delta(S_0, 0)$ and $q_1 = \delta(S_0, 1)$ have the trailing state S_0. Therefore, q_0 and q_1 copy the transitions of S_0 as follows: $\delta(q_0, 255) \leftarrow \delta(S_0, 255) = q_{esc}$ and $\delta(q_0, 1) \leftarrow \delta(S_0, 1) = q_1$, respectively $\delta(q_1, 255) \leftarrow \delta(S_0, 255) = q_{esc}$ and $\delta(q_1, 1) \leftarrow \delta(S_0, 1) = q_1$. The trailing state S_0 satisfies $\delta(S_0, 0) = q_0$; as a consequence, $q_2 = \delta(q_0, 0)$ has the trailing state q_0 and it copies its transitions: $\delta(q_2, 2) \leftarrow \delta(q_0, 2) = E_2, \delta(q_2, 0) \leftarrow \delta(q_0, 0) = q_2$ and $\delta(q_2, 1) \leftarrow \delta(q_0, 1) = \delta(S_0, 1) = q_1$. Thereafter, we enqueue $q_3 = \delta(q_2, 255)$ with the trailing state $q_{esc} = \delta(q_0, 255)$. Finally, q_3 has no transitions to copy from q_{esc}.

3.4 Connecting the Automata

We previously presented the algorithms for building the automaton for each subpattern. As such, we assume we have built a vector of automata $\mathcal{F} = [\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_{n-1}, \mathcal{F}_n]$, where $\mathcal{F}_0$ is the automaton for the optional prefix $\mathbf{p}_0$, $\mathcal{F}_i, 1 \leq i \leq n-1$ are the automata for the middle subpatterns $\mathbf{p}_i$ and $\mathcal{F}_n$ is the automaton for the optional suffix $\mathbf{p}_n$. Our goal is to connect all the forward automata $\mathcal{F}_i, i \leq n-1$, into a single forward automaton $\mathcal{F}_f$ to avoid splitting symbols when parsing.

We connect the forward automata on-the-fly by constructing them in reverse order. When building $\mathcal{F}_m$, $m < n-1$, the end states $E_i^{(m)}$ are the start states $S_i^{(m+1)}$ of $\mathcal{F}_{m+1}$. Adding a transition $\delta(q^{(m)}, c) \leftarrow E_k^{(m)}, k \neq 0$, implies using the first k characters of sym_c for matching $\mathbf{p}_m$; if the other characters of sym_c may be used to start matching $\mathbf{p}_{m+1}$, then, $\delta(S_k^{(m+1)}, c) = q^{(m+1)}$, so we redirect $\delta(q^{(m)}, c) \leftarrow q^{(m+1)}$. If either $k = 0$ or $S_k^{(m+1)}$ does not transition through c, we cannot use the remaining characters of the symbol to match $\mathbf{p}_m$, so we consume the entire symbol by setting $\delta(q^{(m)}, c) \leftarrow S_0^{(m+1)}$. As a result, we have two automata: a forward automaton $\mathcal{F}_f$ and a backwards automaton $\mathcal{F}_b = \mathcal{F}_n$.

For an arbitrary automaton F, we implement a parsing function $\mathcal{P}(F)$ that takes the compressed string as a parameter and returns false, in case we reject the string, and a pair of indices (i, s) denoting the final position in the string and the resulting end-state index, respectively, otherwise. For a pair $(\mathcal{F}_f, \mathcal{F}_b)$, we reject a string if if either $\mathcal{P}(\mathcal{F}_f)$ or $\mathcal{P}(\mathcal{F}_b)$ return false. We accept the string if the pairs of indices (i_f, s_f) and (i_b, s_b) satisfy $i_f < i_b$ or $i_f = i_b$ and $s_f \leq s_b$, to ensure $\mathcal{F}_f$ and $\mathcal{F}_b$ do not consume the same character.

4 Implementation and Optimisations

We present five different optimisation techniques as follows. The first optimisation targets automaton construction. The second one replaces the matching function with a JIT-compiled automaton-specific one. The last three focus on execution performance.

4.1 Sub-automaton Caching

For a middle pattern $\mathbf{p}$, when constructing the sub-automata matching prefixes $\mathbf{p}[l :]$ in the initial phase, we recursively build sub-automata for prefixes of the form $\mathbf{p}[j :]$. Consequently, a naive implementation may build an automaton for the same prefix multiple times. We eliminate this redundancy through *sub-automaton caching*, maintaining a lookup table $\mathcal{T} : \mathbb{N} \to Q$ that maps each index j to the start of the sub-automaton built for the prefix $\mathbf{p}[j :]$.

The construction algorithm proceeds as follows: when building an automaton for the prefix $\mathbf{p}[j :]$, we first query $\mathcal{T}[j]$. If $\mathcal{T}[j] = q$, under certain assumptions, we return q directly. If the assumptions do not hold, we perform a recursive deep copy of the sub-automaton starting in q to obtain a new state q', which is then returned. If $\mathcal{T}[j]$ is undefined, we construct the automaton for the prefix $\mathbf{p}[j :]$ having the start state q, set $\mathcal{T}[j] \leftarrow q$, and return q.

4.2 Compiling the Automaton

In addition to the *interpreted* approach, we implement two JIT-compiled variants. We generate automaton-specific parsing code compiled via *C++* or *LLVM*. Compilation cost can be hidden [7, 8] by running the interpreted approach until the compiled function is ready, then switching if beneficial. If we estimate that decompressing and matching the decoded string is faster, we fall back to decompression rather than parsing the automaton while waiting for the compilation process to complete.

4.3 Level Assignment

We accelerate string rejection by assigning to each state q a *level* $\mathcal{L}(q)$, defined as the minimum distance to an accepting state, allowing for early rejection when the remaining input cannot reach an accepting state.

For prefixes, we use the right-to-left automaton construction to efficiently compute levels. When adding a transition $\delta(q, c) \leftarrow q_0$, $\mathcal{L}(q_0)$ has already been computed and is immutable. If $\mathcal{L}(q)$ is undefined or $\mathcal{L}(q) > \mathcal{L}(q_0) + 1$, we update $\mathcal{L}(q) \leftarrow \mathcal{L}(q_0) + 1$.

For middle patterns, we apply a reverse breadth-first search on the automaton $\mathcal{F}$ using a priority queue storing pairs (ℓ, q) to ensure the levels we assign are minimal. To avoid assigning the same state a level multiple times, we employ a set of visited states. Initially, for all states $q' \notin \mathcal{F}$ and $q \in \mathcal{F}$ such that $\delta(q, c) = q'$, we enqueue $(\mathcal{L}(q') + 1, q)$. At each step, for the pair (ℓ, q), if q has not been previously visited, we first assign $\mathcal{L}(q) \leftarrow \ell$ and add q to the visited set. Afterwards, for all states q'' such that $\delta(q'', c) = q$, we enqueue $(\ell + 1, q'')$. By employing a priority queue, we guarantee that the assigned levels are minimal. Finally, for all sink states q (i.e., without any transitions), we set $\mathcal{L}(q) \leftarrow \mathcal{L}(\delta_{def}(q)) + 1$ as $\delta(q, c) = \delta_{def}(q)$ for all c. Table 2 presents the levels for the states in Figure 4d.

Table 2: Levels of states from Figure 4d

State	S_0	S_1	q_0	q_1	q_2	q_3	q_{esc}	E_0	E_2
Level	2	3	1	2	1	1	3	0	0

For suffixes, we construct a suffix tree matching sequences of the form $\langle i, c_0^{(k)}, \ldots, c_{j_k-1}^{(k)} \rangle$ for $k > 0$ (or $\langle c_0^{(0)}, \ldots, c_{j_0-1}^{(0)} \rangle$ for $k = 0$), ensuring that a state $q_i^{(k)}$ exists for each suffix $\langle c_i^{(k)}, \ldots, c_{j_k-1}^{(k)} \rangle$. For the base case $k = 0$, we set $\mathcal{L}(q_i^{(0)}) \leftarrow i$ for each state $q_i^{(0)}$ in the initial suffix tree. For $k > 0$, if $\mathcal{L}(q_i^{(k)})$ is undefined or $\mathcal{L}(q_i^{(k)}) > i + 1$, we set $\mathcal{L}(q_i^{(k)}) \leftarrow i + 1$.

Given the start states S_f and S_b of $\mathcal{F}_f$ and $\mathcal{F}_b$ respectively, we reject any string of length $\ell < \mathcal{L}(S_f) + \mathcal{L}(S_b) - 1$ immediately. During execution, we verify in each step that a match may still occur, given the current level and position.

4.4 Prefix and Suffix Verification

Automata for start and end patterns often exhibit a unique transition path from the initial state to the first point of divergence; we call this the *prefix* (for start patterns) or *suffix* (for end patterns). While end patterns need not have a common suffix, start patterns of length at least eight are guaranteed a prefix of length at least one, since each symbol encodes at most eight characters and we use greedy encoding. Inductively, for any prefix of length $n \geq 8$, only the last seven characters may cause divergence.

We exploit this by first verifying the prefix (or suffix) against the input before continuing from the diverging state. In Figure 3, the prefix is $\langle 0, 1, 2 \rangle$, and successful matching resumes execution from state q_2. In compiled implementations, these sequences are encoded as compile-time constants and verified using integer comparisons.

4.5 Self-Loop Optimisation

States q with $\delta_{def}(q) = q$ frequently consume many symbols without advancing the automaton. A naive switch-based implementation incurs $\Omega(\log_2 N)$ comparisons per symbol $c \notin \delta(q, \cdot)$ when the compiler does not generate a jump table for the $N = |\delta(q, \cdot)|$ transitions. This is common in practice due to small transition sets.

We apply two optimisations to such states. First, we precompute a compile-time boolean array mapping each symbol c to whether $\delta(q, c)$ is defined, reducing the self-loop check to a constant number of instructions per character. Second, for $N \leq 16$, we apply SIMD acceleration: we load 16 input bytes via unaligned loads and compute a 16-bit bitmask in which bit i is set if and only if the i-th byte triggers a transition out of q.

For small transition sets ($N < 6$), we broadcast each transition symbol $c \in \delta(q, \cdot)$ to a 128-bit register and compare against the

input using `_mm_cmpeq_epi8`, producing one 16-bit bitmask per symbol. The N resulting bitmasks are combined via bitwise `OR` to obtain the final mask.

For $6 \leq N \leq 16$, we store all transition symbols in a zero-padded 16-byte array and invoke the `_mm_cmpestrm` intrinsic with the `_SIDD_CMP_EQUAL_ANY` flag, computing the mask in a single operation and avoiding the overhead of multiple `OR` operations.

5 Evaluation

As discussed in Section 4.2, we consider three execution strategies for our automaton, yielding three variants: *Interpreted*, which parses the automaton at runtime; *C++*, which lowers it to C++, invokes the `clang++` compiler to produce a shared library, and dynamically loads it; and *LLVM*, which emits LLVM IR and JIT-compiles it.

Experimental Setup. All experiments are run on an Intel Core i9-7900X clocked at 3.30 GHz, supporting SSE4.2 and AVX-512. The CPU provides 10 physical cores and 20 hardware threads via hyperthreading. The system runs Ubuntu 25.10 with kernel 6.17.0-20. Our implementation[1] extends a fork of the FSST repository [2], ported to C++20 and compiled with GCC 15.2.0 using the `-O3` and `-march=native` flags.

Datasets. We evaluate on four datasets, each compressed with FSST and serialised to disk together with its symbol table:

- *TPC-H*, generated at scale factors 1, 10, and 100, from which we extract all string columns targeted by `LIKE` predicates across the 22 benchmark queries [16].
- *StackOverflow*[2], from which we extract the three longest string columns: `Body` of `Posts`, `Comment` of `PostHistory`, and `Text` of `Comments`.
- *IMDB*[3,4], targeting the `plot`, `quotes`, `films`, and `actors` tables.
- *Public BI*[5], imported into DuckDB, from which we extract the longest columns where FSST is actively used for compression.

These columns are chosen for their high cardinality: the large number of unique values renders dictionary-based compression highly inefficient. For StackOverflow and IMDB, we sample the data to emulate scale factors and assess scalability.

Pattern Generation. For each dataset, we generate three categories of `LIKE` patterns: *prefixes*, which are start-anchored patterns beginning with a non-wildcard character and followed by optional middle subpatterns (e.g., `'ABC%'` or `'ABC%DEF%'`); *suffixes*, defined as end-anchored patterns terminating with a non-wildcard character and preceded by optional middle subpatterns (e.g., `'%DEF'` or `'%ABC%DEF'`); and *substrings*, representing unanchored patterns composed exclusively of middle subpatterns surrounded by the wildcard character `%` (e.g., `'%ABC%'`). The distribution is approximately 20% prefixes, 20% suffixes, and 60% substrings. All patterns are data-aware and typically contain multiple short subpatterns. We follow a hybrid generation strategy: following SQLStorm [14], we use a large language model to derive patterns from a subsample of each dataset, and complement these with manually curated patterns. For TPC-H, we additionally include the original `LIKE` patterns from the 22 benchmark queries to align with standard performance evaluations.

Competitors. Both competitors lack native support for compressed matching and must first decompress the strings into a transient buffer before evaluating the predicate. The first, Hybrid String Search (*HSS*) [13], combines SIMDSearch [15] with the Two-Way Search algorithm [4]. For patterns of at most 16 characters matched against texts longer than 16 characters, HSS uses SIMDSearch with SSE4.2 instructions to locate the first occurrence of the `needle` within the `haystack`; otherwise, it falls back to Two-Way Search, which itself combines Knuth–Morris–Pratt [9] and Boyer–Moore [3]. Following prior work [13], we restrict SIMDSearch to `needles` of at most 12 characters, as SSE4.2 performance degrades beyond this length. The second competitor, Vectorscan[6] (*VS*), extends Hyperscan [19]: we translate the `LIKE` pattern into a regular expression, which the library compiles into optimised bytecode. This bytecode is then run on the decompressed string using the execution engine best suited to the available hardware, such as exploiting SIMD where available.

5.1 Throughput

Single-threaded Throughput. Figure 5 reports the median single-threaded throughput of all benchmarked algorithms, grouped by pattern type and dataset. Each measurement covers only the optional decoding of the input string and the execution of the matching algorithm. The one-time cost of constructing and, where applicable, compiling the automaton is reported separately in Section 5.2.

Because our automaton-based variants operate directly on compressed data, they can reject non-matching strings for *prefixes* and *suffixes* within a few CPU instructions, giving them a substantial throughput advantage over the decoding-based alternatives. The magnitude of this advantage exhibits a strong positive correlation with the average string length in the dataset. On the Public BI dataset, where strings average between 5 and 20 characters, we observe a 3.5× median improvement. On StackOverflow, whose columns span several hundred characters, the improvement grows to 17×.

For non-selective *prefixes* and *suffixes* that contain middle subpatterns, however, the matching phase is dominated by the substring search. This explains the anomalous performance of automata approaches on the lower tail of the throughput distribution for StackOverflow *suffixes*; one benchmarked pattern features an end pattern matched by 50% of the column entries. In this case, the lack of an early exit forces the runtime to be dominated by checking for the middle patterns.

For *substrings*, the compiled variants maintain a consistent 2.5–3.5× throughput improvement across all datasets, with gains driven primarily by pattern complexity rather than string length. The interpreted approach lacks the self-loop optimisation from Section 4.5

[1] https://github.com/calin2110/FSST-LIKE-Matching/
[2] Downloaded using a script from [14] (accessed: July 28, 2025).
[3] `plot` and `quotes` from https://ftp.fu-berlin.de/pub/misc/movies/database/frozendata/ (accessed: December 11, 2025).
[4] `films` and `actors` from https://datasets.imdbws.com/ (accessed: December 11, 2025).
[5] Downloaded from https://github.com/cwida/public_bi_benchmark (accessed: April 1, 2026).
[6] https://github.com/VectorCamp/vectorscan

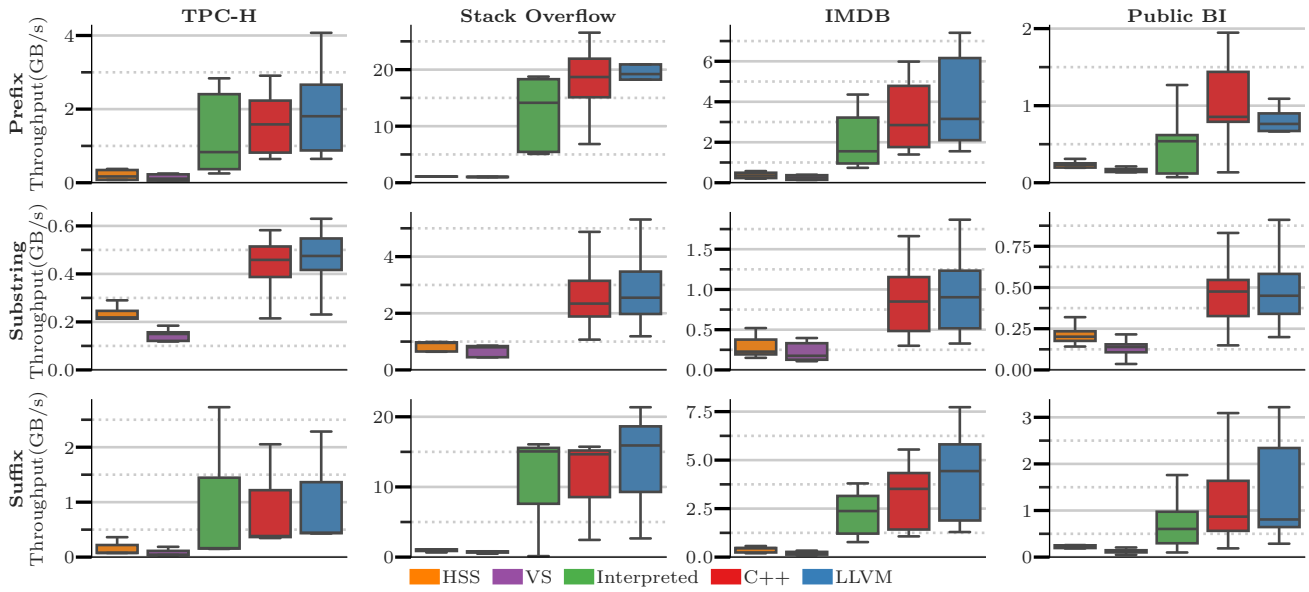


Figure 5: Median single-threaded throughput (GB/s) for the presented algorithms, grouped by dataset and pattern type.

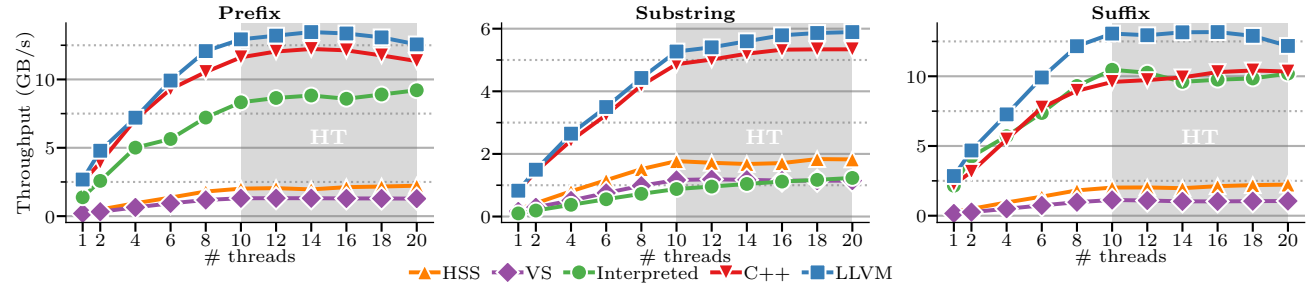


Figure 6: Median multithreaded throughput (GB/s) for the presented algorithms on the IMDB datasets.

due to implementation complexity, which causes its performance to degrade on *substrings*; we therefore omit its results for this category.

Multithreaded Throughput. Figure 6 shows the median matching throughput of the approaches presented above on the IMDB dataset across varying thread counts. We implement a multithreaded version that partitions the input into equal-sized chunks and processes them in parallel via a thread pool, following the concepts of morsel-driven parallelism [11]. For the decoding-based competitors, we ensure correctness by maintaining a thread-local FSST decoder instance per worker. Additionally, *Vectorscan* must re-compile the pattern independently for each worker to avoid concurrency issues in the matching phase.

For all pattern types, we observe near-linear scaling up to 10 threads, matching the number of physical cores. Beyond this threshold, the throughput of *prefixes* and *suffixes* flattens out or declines slightly as hyper-threading takes effect, leading to resource contention. In contrast, *substrings* continue to show modest gains up to 20 threads. Notably, for this category of patterns, the *Interpreted* approach keeps up with *Vectorscan* across the entire thread range; we attribute this to the relatively short lengths of the strings in the IMDB dataset.

5.2 Runtime Breakdown

As our approach requires constructing and compiling a specialised automaton for each pattern, we decompose the runtime into three execution phases: the *preparation* phase captures pattern preprocessing, including automaton construction and, where applicable, JIT compilation; the *decoding* phase measures FSST decompression into a transient buffer for algorithms that cannot match on compressed data; and the *matching* phase covers predicate evaluation on either the compressed or decompressed representation. Section 5.1 presents only the numbers of the decoding and matching phase.

Figure 7 shows the cumulative runtime across the three execution phases for all algorithms, measured on three sample sizes of the IMDB `quotes` table. As before, the interpreted variant is excluded from the substring results; without the self-loop optimisation (Section 4.5), it cannot match the efficiency of the other approaches for middle subpatterns.

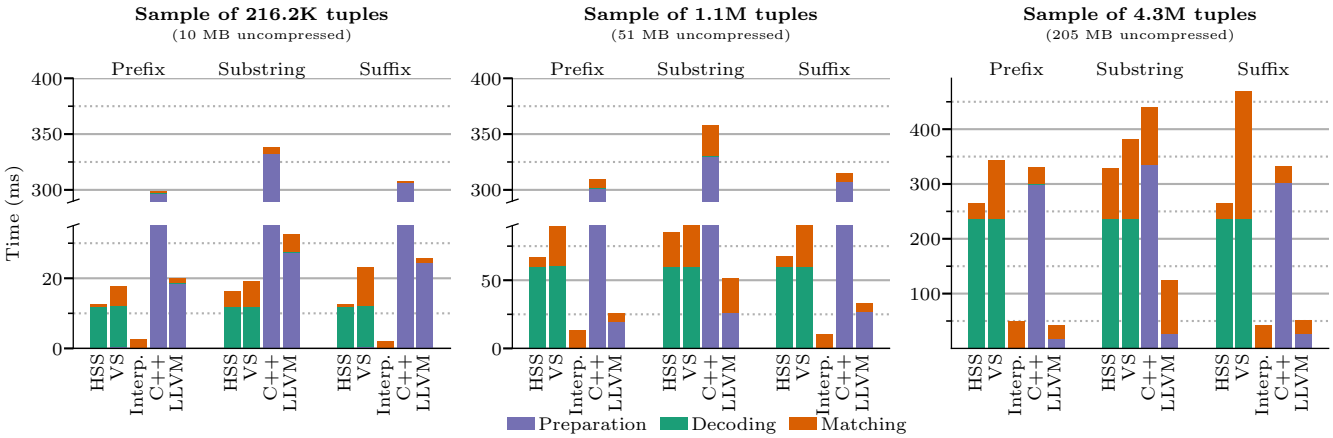


Figure 7: Runtime breakdown of all algorithms across three sample sizes for the IMDB `quotes` dataset. Runtimes are decomposed into preparation, decoding, and matching phases.

At small sample sizes, compilation overhead dominates the total runtime. Automaton construction alone remains cheap, however, which lets the interpreted variant outperform both decoding-based competitors on *prefixes* and *suffixes*. As the sample size grows, the fixed compilation cost amortises over more data, and the LLVM variant becomes the fastest approach across all pattern categories.

The C++ and LLVM variants yield nearly identical matching-phase runtimes, yet C++ compilation is consistently significantly slower. We attribute this to the fact that `clang++` internally lowers C++ to LLVM IR before generating machine code, so emitting LLVM IR directly avoids the overhead of the C++ frontend while retaining the benefits of compiled execution.

The compilation overhead observed at low sample sizes has direct implications for deployment in real database systems. In a typical DBMS, strings are compressed at the granularity of storage units such as data blocks of fixed size or a fixed number of tuples, each with its own FSST symbol table. A naive integration would therefore require constructing and compiling a new automaton for every data block scanned, amortising the compilation cost over a comparatively small amount of data. One way to mitigate this overhead is to share symbol tables across data blocks, constructing a new symbol table only when the compression ratio of the current one drops below a given threshold. Under such a scheme, the same automaton can be reused across many blocks, reducing the per-block cost of automaton construction and compilation to a negligible fraction of the total runtime.

6 Conclusion

This paper demonstrates the performance gains achieved by filtering FSST-compressed strings without decompression. By operating directly on compressed data, we reduce the best-case computational complexity for prefix and suffix checking from $O(m)$ to $O(1)$, where m represents the length of the compressed string. Furthermore, we improve the worst-case complexity from $O(m)$ to $O(\min(L+1, m))$, where L is the level of the start state of the corresponding automaton. This theoretical improvement is supported by benchmarks, which show 2–14× improvements for highly selective patterns using the interpreted approach and 3.5–17× improvements when generating and compiling LLVM-IR. For substrings, although the theoretical computational complexity is hard to quantify, we observe a 2.5–3.5× improvement across all benchmarked datasets when using the LLVM approach.

References

[1] Alfred V. Aho and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM* 18, 6 (1975), 333–340.

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

[3] Robert S. Boyer and J. Strother Moore. 1977. A fast string searching algorithm. *Commun. ACM* 20, 10 (Oct. 1977), 762–772. doi:10.1145/359842.359859

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

[5] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *EDBT*. OpenProceedings.org, 72–83.

[6] Francesco Gargiulo and Rossano Venturini. 2025. OnPair: Short Strings Compression for Fast Random Access. *CoRR* abs/2508.02280 (2025).

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

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

[9] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350. arXiv:https://doi.org/10.1137/0206024 doi:10.1137/0206024

[10] Peter M. Kogge and John Shalf. 2013. Exascale Computing Trends: Adjusting to the "New Normal"' for Computer Architecture. *Comput. Sci. Eng.* 15, 6 (2013), 16–26.

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

[12] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Muhammad Mukarram Bin Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve D. Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. 2022. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *SIGCOMM*. ACM, 66–85.

[13] Adrian Riedl, Philipp Fent, Maximilian Bandle, and Thomas Neumann. 2023. Exploiting Code Generation for Efficient LIKE Pattern Matching. In *VLDB Workshops (CEUR Workshop Proceedings, Vol. 3462)*. CEUR-WS.org.

[14] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proc. VLDB Endow.* 18, 11

(2025), 4144–4157.
[15] Evangelia A. Sitaridi, Orestis Polychroniou, and Kenneth A. Ross. 2016. SIMD-accelerated regular expression matching. In *DaMoN*. ACM, 8:1–8:7.
[16] Standard Specification and Transaction Processing Performance Council TPC. 1993. Tpc benchmark tm h. (1993).
[17] Julia Spindler, Philipp Fent, Adrian Riedl, and Thomas Neumann. 2024. Can Delta Compete with Frame-of-Reference for Lightweight Integer Compression?. In *VLDB Workshops*. VLDB.org.
[18] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD*. ACM, 1:1–1:6.
[19] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang