

Eberhard Karls Universität

Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Database Systems Research Group

Bachelorthesis Computer Science

Bringing Row Pattern Matching to DuckDB: NFA construction

Samuel Heid

06.05.2025

Examiner

Prof. Dr. Torsten Grust

Supervisor

Louisa Lambrecht

Samuel Heid:

Bringing Row Pattern Matching to DuckDB: NFA construction

Bachelorthesis Computer Science

Eberhard Karls Universität

From 06.01.2025 to 06.05.2025

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorthesis selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorthesis wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Rotenburg, 05.09.25
Ort, Datum

S. Heid
Samuel Heid

Abstract

`MATCH_RECOGNIZE` is a relatively recent addition to SQL, designed for performing row pattern matching using regular expressions across rows in a database table. However, due to limited support across many database management systems (DBMSs), this functionality is not widely available. To address this gap, this thesis presents a transpiler that translates `MATCH_RECOGNIZE` statements into semantically equivalent SQL code using only `WITH RECURSIVE` and `WINDOW` functions.

A core component of this approach is the construction of a transition table from the regular expression defined in the `PATTERN` clause. This process leverages Glushkov's construction algorithm to generate a non-deterministic finite automaton (NFA) directly from the syntax tree of the expression. To ensure deterministic evaluation suitable for SQL queries, a transition penalty system is introduced. This mechanism assigns a penalty value to each transition, enabling the resolution of ambiguities and the transformation of the NFA into a deterministic finite automaton (DFA).

The resulting transition table serves as the foundation for the recursive SQL query, enabling portable pattern matching even in systems without native support for `MATCH_RECOGNIZE`.

Contents

Abstract	v
Acronyms	ix
1 Introduction	1
1.1 Goal of the thesis	1
2 Match Recognize	5
2.0.1 PATTERN	5
2.0.2 DEFINE	6
2.0.3 Example	7
3 NFA Algorithm	9
3.1 Glushkov's construction algorithm	9
3.2 Algorithm	10
3.2.1 Elimination of the quantifier	11
3.2.2 Linearisation	12
3.2.3 NFA construction	12
3.2.4 Properties	13
3.3 Example	14
4 Implementation	15
4.1 Regular expression syntax tree	15
4.2 AST as an input	17
4.2.1 AST-visitor	17
4.3 AST-Visitor in Python	19
4.4 NFA algorithm in Python	23
4.5 Visualisation	28
5 Results	29
5.1 Limitations	29
5.1.1 Recursion limit	29
5.1.2 Runtime	29
5.2 Conclusion	30
5.3 Future Work	30
Bibliography	33

Acronyms

AST abstract syntax tree
CTE common table expression
DBMSs database management systems
DFA deterministic finite automaton
NFA non-deterministic finite automaton
RST regular expression syntax tree

Introduction

In an increasingly digitalized world, vast amounts of data are collected daily across a wide range of topics. Much of this data is stored in relational databases and accessed through SQL queries. Consequently, SQL is continually evolving, with new features being introduced from time to time. SQL is not a single, standardized language but rather a collection of dialects developed by various companies and open-source communities. Prominent examples include Oracle [1], DuckDB [2], MySQL [3], and Trino [4]. Although these dialects share a common foundation, each introduces its own extensions and syntax variations. For example, DuckDB supports an extensive set of aggregate functions, while Transact-SQL [5] uses brackets to define identifiers, as in `SELECT * FROM [table]`. All statements are defined as part of the ANSI SQL standard [6] to ensure consistent syntax across different systems. However, each DBMS implements only a subset of these statements. Therefore, even if a statement is included in the standard, it does not guarantee support by a specific DBMS. Our goal is to enable the use of a statement on DBMSs that do not natively support it. To achieve this, we use a `WITH RECURSIVE` query, which is widely supported among modern DBMSs, including DuckDB.

A variety of tools are available to analyze the vast amounts of data being generated. Analyzing time series data—such as financial or sensor data—can be particularly valuable when specific patterns can be identified, or potential fraud can be detected [7]. In databases, such pattern detection can be performed using the `MATCH_RECOGNIZE` statement. The introduction of a transpiler addresses the issue of the lack of compatibility. Instead of writing separate queries for different database systems, a single statement can be written and translated as needed, significantly reducing development time, implementation errors, and maintenance efforts. Moreover, executing pattern matching directly within the existing database infrastructure, rather than exporting data to external analysis tools or other DBMS, offers several advantages. These include minimizing security risks, reducing data movement overhead, and saving a substantial amount of time.

1.1 Goal of the thesis

This project aims to develop a transpiler that can translate the `MATCH_RECOGNIZE` query into a semantically equivalent SQL representation that is comprehensible to any other DBMS. The prerequisite for the DBMS is its support for the `WITH RECURSIVE` and `WINDOW` function. The project is divided into three distinct theses each which have their own part to contribute to the final result.

- The first thesis posits the creation of a parser capable of analyzing the specified `MATCH_RECOGNIZE` query and generating an abstract syntax tree (AST) of the query. The Antler library [8] and a specific grammar for the syntax are utilized to translate the given query into our data structure, which serves as the representation of the query. This data structure is the starting point for the other two algorithms.
- The second thesis posits the creation of a transpiler that is capable of translating the AST into a semantically equivalent SQL representation. The AST from the first thesis is taken, and a `pglast` [9] representation of the new query is built with specific rules.
- The third is to construct an automaton from the regular expression, which the transpiler can thereafter use.

In my thesis, I will talk about the construction of the automaton. If you are interested, you can find further information on the other components in the thesis of Marcel Knüdel [\[10\]](#) or Tim Findling.

The automaton is required in the form of a transition table to support the `WITH RECURSIVE` query in identifying all relevant rows. DBMSs without the support for match recognize are not capable of processing raw regular expressions directly, this intermediate representation is essential. Moreover, the automaton must be deterministic to guarantee the correctness and uniqueness of the result at any given point in time. To address this challenge, a penalty mechanism will be introduced. This ensures that, in cases where multiple transitions are theoretically possible, only one is selected based on its assigned penalty. In this way, deterministic behavior is maintained throughout the evaluation process.

The pattern defined in the `PATTERN` clause must be evaluated systematically. Since the aim is to formulate the logic using a recursive common table expression (CTE), it is not feasible to apply a regular expression directly across multiple rows within a database table. Therefore, the regular expression string needs to be understood and interpreted by a computer. A widely used approach for this purpose is the use of automata theory. Specifically, NFA are capable of representing any regular expression. However, the syntax of the expression must first be translated into an equivalent, yet structurally different form. For instance, `A?` can also be represented as `(A| ϵ)`, which I will explain further in following sections.

An NFA can also be represented as a transition table: a structured collection of rows that describe the state transitions. These rows can be stored in a database table, and then utilized by the `WITH RECURSIVE` query for evaluating the final result.

The current thesis will concentrate on the creation of the transition table for the NFA. For more in-depth information regarding the parsing and transformation from `MATCH_RECOGNIZE` syntax to the recursive CTE. I recommend having a glance at the other theses about this topic.

The complete grammar of `MATCH_RECOGNIZE` includes a wide range of options. I am going to talk only about the main structure.

The basic structure of a `MATCH_RECOGNIZE` query is as follows:

```
1 SELECT *
2 FROM t
3 MATCH_RECOGNIZE (
4     PARTITION BY ...
5     ORDER BY ...
6     MEASURES ...
7     PATTERN ...
8     DEFINE ...
9 );
```

Similar to the use of window functions in SQL, the `PARTITION BY` clause allows the data to be divided into logical partitions, after which the `ORDER BY` clause can be applied to sort the rows within each partition. The ordering of rows plays a critical role in this context. As with window functions, the ordering determines the outcome, since the pattern-matching process operates on the ordered data—altering the order would consequently change the result.

To utilize the computed values within the `SELECT` clause, the `MEASURES` clause must be used to export the relevant attributes. This clause enables the specification of variables, expressions involving correlation variables, ordering attributes, singleton variables, and aggregate functions applied to the attributes of the input stream defined by the recursive CTE. Furthermore, the `CLASSIFIER()` function can be used to tag matches, providing information about which rows correspond to which variables in the `PATTERN` clause.

In this thesis, I am going to concentrate on the `PATTERN` and `DEFINE` clauses because they are absolutely relevant to this topic. For more detailed information about the settings and specific options, you can look at the documentation from Oracle [1].

To define the condition which rows should be matched during the evaluation, there are three possible clauses we can use.

Match Recognize

Since 2018 `MATCH_RECOGNIZE` was put into the SQL dialect from Oracle, it is possible to match specific patterns throughout many rows in a database table more easily [1]. Everything can be programmed in SQL to get the desired result from the data, but in the same cases, working with `WITH RECURSIVE` and attempting to get the correct result can be tedious. There is `MATCH_RECOGNIZE` to make your life easier. It performs exceptionally well on time series data. After ordering the data, it is possible to discover special patterns. For example, you have a dataset on the USD/EUR chart. It is possible to see the EUR trend with your own eyes, but it is impossible to see the specific movements and the exact dates. Suppose you want to find a specific behavior of the chart. The price is falling 3 or more days followed by some days without movement and then rising again. For this, you can formulate a regular expression `D{3,} E+ U+`. Consequently, you will obtain every instance of this movement. The aggregate function could be applied to every matching group to facilitate a more profound analysis.

2.0.1 PATTERN

The pattern is a regular expression used to match the rows inside the partitions. The regular expression contains correlation variables defined in the `DEFINE` clause. These variables are valid if the expression inside the define clause is true. If a variable is not specified, it is assumed to be always true. The regexp grammar is shown in Figure 2.1. The `correlation_name` is a string or variable defined in the `DEFINE` clause. After a variable, a `pattern_quantifier` is allowed. The quantifiers are common in the regular expression syntax. Oracle supports these six symbol

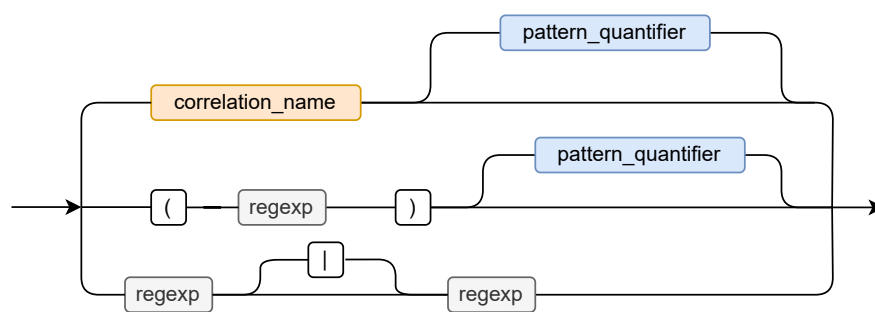


Figure 2.1: Regexp grammar for the regular expression inside the pattern of the statement.

```

1 PATTERN (A (B | C*)+ D)
2 SUBSET S1 = (A,B)
3 DEFINE
4   A AS t.x < 1
5   B AS t.y > 1
6   C AS t.x < prev(t.x)
7   D AS t.y > prev(t.y)

```

Figure 2.2: Pattern definition example illustrating the use of PATTERN, SUBSET, and DEFINE clauses within a MATCH_RECOGNIZE statement.

quantifiers: `*?`, `+?`, `??`, `*`, `+`, `?`. The plus and star operators are greedy. They are trying to match as many rows as possible. The `?` The quantifier marks if a variable is lazy, and, for example, `A+?` has to match at least one row, but after one match, it tries to end this match and get as few rows as possible. Same applies to `A*?` but it is also possible to match nothing so that A is not present in the final result. The range quantifier `A{n,m}` restricts how often the condition A is matched. It has to be matched at least n-times but at most m-times. There are short forms like `A{n}` where A has to be matched exactly *n* times or `A{n,}` corresponding to at least n occurrences.

The second option is to group a regular expression with parenthesis, and it is, for example, possible to use a quantifier on the whole group: `A (B | C*)+ D`. Regular expressions can be separated with the Or-quantifier (`|`), and the match is true if either the right or the left expression evaluates to true.

2.0.2 DEFINE

The boolean conditions for the correlation variables are defined in the **DEFINE** clause. Any column which is present in the schema of the table can be utilized to define variables. A simple condition might be a column comparison with a constant variable like `t.x < 1`; comparison between different columns is allowed `t.x < t.y` or the usage of a typical function to get, for example, the value of the previous row `t.x < prev(t.x)`. With this function, it is possible to find a pattern of a rising value, such as the rising EUR.

An optional **SUBSET** clause is possible if multiple variables should be grouped together. A new variable can be introduced, which can hold multiple correlation variables from the **DEFINE** clause. You can use the new variable inside the **MEASURES** and the **PATTERN** clause.

The three clauses look like this in Figure 2.2. Four variables are defined: A, B, C and D. A and B are just restrictions on the related column, either greater or less than 1. C is true if the current value is smaller than the last value in the column x. In other words, x is declining. D is just the opposite, and y is rising.

A subset S1 is defined as the combination of A and B. It can be used to make some computations in the measure clause over a subset of the matches, which are true for A or B. The pattern clause with the regular expression `A (B | C*)+ D` is responsible for the actual rows that are part of the result.

2.0.3 Example

I will present a short example of the `MATCH_RECOGNIZE` statement. The goal of the query is to find a pattern inside the USD/EUR price since 2003 where the price falls 3 days or more, followed by at least one day of no movement and then rises again.

```
1 SELECT *
2 FROM exchange_rate
3 MATCH_RECOGNIZE (
4     ORDER BY price_date
5     MEASURES
6         MATCH_NUMBER()      AS match_no,
7         CLASSIFIER()        AS tag,
8         FIRST(price_date)   AS first_date,
9         LAST(price_date)    AS last_date,
10        MAX(last_price)     AS max_price,
11        COUNT(last_price)   AS cnt
12     ALL ROWS PER MATCH
13     PATTERN (D{3,} E+ U+)
14     SUBSET S1 = (D, E)
15     DEFINE
16         D AS last_price < PREV(last_price),
17         U AS last_price > PREV(last_price),
18         E AS last_price = PREV(last_price)
19 ) AS T;
```

Listing 2.1: Match_Recognize example for evaluation the matches for the given expression on a dataset of the USD/EUR data.

The data is ordered along the `price_date` to be in the chronically correct order. The following essential clauses are the `PATTERN` and `DEFINE` clauses. As previously described, this section provides the logic of which rows should be selected. Here, you can find the definition of U, D and the price being even (E). The key word, `ALL ROWS PER MATCH`, is a change to the output. It returns all rows that are part of the matching groups, not only the first and last rows. The `MEASURES` clause is used to export the result of the matching process. The `MATCH_NUMBER()` function gives the number of the match. The `CLASSIFIER()` function returns the name of the variable which was matched. The `FIRST()` and `LAST()` functions give the first and last dates of the matches. The `MAX()` and `COUNT()` functions are used to get the maximum and the count of the price.

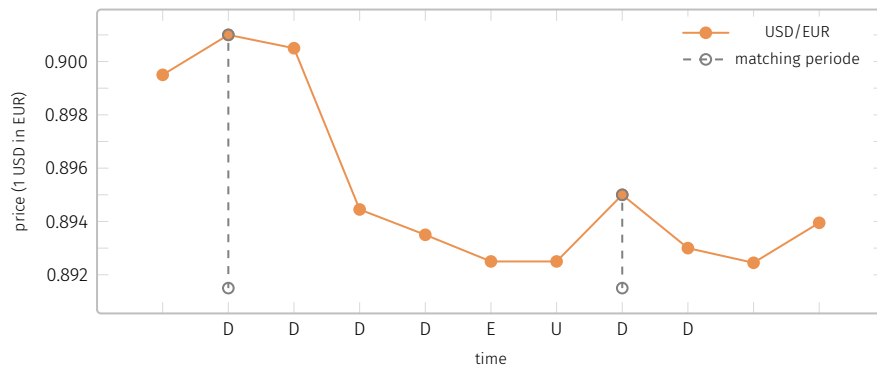


Figure 2.3: The time series *orange* show the price development of EUR price at every working date between 2019-12-25 and 2020-01-06 [11]. Every dot marks the current ending price at that day. The row tags are given. The dashed *gray* line denotes the matching periode with the given regular expression.

The result of the query is a table with all the matches. For visualization in Figure 2.3, only the plot of the longest match is shown. With this query the exact time periode for this match can be seen and the movement of the price. For every day the tag of the match is shown under the plot.

NFA Algorithm

An automata is an abstract model containing states, transitions, and actions. It is in a specific state and can take a string of letters as input. With different input letters, the automaton can change its state. How the state reacts to the various input letters is defined inside the transitions. A five-tuple with the following entries represents an automaton. [12]

1. Q : a set of states
2. Σ : the alphabet
3. q_0 : the initial state $q_0 \in \Sigma$
4. F : the accept states $F \subset Q$
5. δ : a function $Q \times \Sigma \rightarrow Q$ a transition function

There are two types of machines: deterministic and non-deterministic automaton. A deterministic finite automaton (DFA), or short DFA, is an automaton where the transition is clearly defined for every input letter. A non-deterministic finite automaton (NFA) can have one state and more than one following state. If this happens, an arbitrary state will be selected.

3.1 Glushkov's construction algorithm

Viktor M. Glushkovs designed an algorithm to translate a regular expression into a non-deterministic finite automaton without any ϵ -transitions. An epsilon transition is a transition without any input, epsilon is also equal to the empty word. He described the process in his paper [13] "THE ABSTRACT THEORY OF AUTOMATA." An automaton can be constructed for every regular expression with the alphabet Σ . The number of states in this automaton is limited by $2^n + 1$ states where n is the number of letters in the alphabet. Lets define the alphabet $\Sigma = \{A, B\}$ and the regular expression $R = (A B^* A)$. The algorithm is divided into three steps.

- **Step 1:** This is the linearisation of the regular expression. Each letter inside the expression R is given a unique unique index. After that, every letter occurs only once inside the expression. The order in which the indexes are given does not matter. For Example:
 $(A B^* A) \rightarrow (A1 B2^* A3)$
- **Step 2:** Three sets are evaluated. The first one is to determine the starting letters. This set contains every letter with which the expression can start. The second set is the opposite; every letter with which the expression can be ended is evaluated. The third set contains the possible following letters for every letter.

I will call the set of starting letters S, terminating letters T, and following letters M:

$$S = \{A1\}$$

$$T = \{A3\}$$

$$M: A1 = \{B2\}, B2 = \{B2, A3\}, A3 = \{\}$$

- **Step 3: Construction** We add a specific starting state called a_0 and a transfer function $\delta(a, x) = ax$ where x is an arbitrary letter of the alphabet Σ and x is inside the starting symbols of the regular expression.

For every set in M, another transfer function $\delta(M, x) = Mx$ with x being a member of the following letters in a set in M.

Now we are able to define for the regular expression $R = (A B^* A)$ as a 5-tuple.

1. $Q = \{a_0, A1, B2, A3\}$

2. $\Sigma = \{A, B\}$

3. $q_0 = a_0$

4. $F = \{A3\}$

5. δ : given as a transition table

δ	A	B
a_0	A1	
A1	A3	B2
B2	A3	B2
A3		

3.2 Algorithm

The **MATCH_RECOGNIZE** syntax for the pattern clause is more versatile than the construction algorithm, which is only capable of handling the star quantifier (*) and the or quantifier (|). In my approach, the plus-quantifier is also a valid symbol even though the star quantifier could replace it: $A+ \rightarrow A A^*$. All other quantifiers can be and have to be replaced with only the plus, star and or quantifier.

The resulting automaton should be deterministic, but the algorithm from Glushkov has, as a result, an NFA. Therefore, I am going to introduce a penalty for each transition. The automaton can be non-deterministic by its symbols, but by adding the penalty, it becomes deterministic. Transitions with lower penalties are preferred. The leftmost symbols in the expression should have a lower penalty because they are more likely to be taken. A (B | C) we have two outgoing transitions from A: $A \rightarrow B$ and $A \rightarrow C$ in this case the automaton would be nondeterministic. At first it might seem this are two different transitions but that's not always the case. Because A and B are defined inside the **DEFINE** they could be true for the same input. In this case a automaton would not know which transition to take. But because B is more on the left of the expression, it gets evaluated first. Therefore, the transition from A to B has a zero penalty, and A to C gets a penalty of one.

3.2.1 Elimination of the quantifier

To display the transformation rules for the different quantifiers I want to eliminate, let A be an arbitrary regular expression and ϵ the empty word.

Optional

The $?$ matches its preceding element zero or one time. $?$ is greedy and its lazy equivalent is $??$.

- **Greedy Optional:** $A? \rightarrow (A \mid \epsilon)$ The greedy optional quantifier can be replaced with a simple Or-clause. Because A is inside the expression more on the left. It gets evaluated first, which means A is preferred (lower penalty) over ϵ .
- **Lazy Optional:** $A?? \rightarrow (\epsilon \mid A)$ through changing the order inside the Or-clause, the empty word ϵ is preferred.

Range quantifier

Let $n, m \in \mathbb{N}$ be two arbitrary but fixed natural numbers for the range quantifier. To specify the variables, they must be $n \geq 0$ and $m > 0$. The dot (\cdot) is the concatenation of two expressions ($A \cdot B$)

The range quantifier matches its preceding element in a range given by n and m .

1. **Exactly n-times:** n has to be greater than zero.

$$A\{n\} \rightarrow A_1 \cdot A_2 \cdot \dots \cdot A_{n-1} \cdot A_n \quad (3.1)$$

You can transform this only by concatenating the expression n-times.

2. **Match between n and m times:** $\{n, m\}$ The precondition is that $n \leq m$. For n equals m ($n = m$) the result is equal to the expression defined in Equation (3.1).

$$A\{n, m\} \rightarrow ((A\{m\} \mid A\{m-1\} \mid \dots \mid A\{n\})) \quad (3.2)$$

the expression is greedy, which means it tries to match as much A 's as possible. To calculate the penalty correctly in the following steps, the longest concatenations of A 's have to be more on the left of the expression. This results in case that the Or-clause with the most A 's is preferred. In contrast, the lazy equivalent is the reversed expression: $A\{n, m\}? \rightarrow ((A\{n\} \mid A\{n+1\} \mid \dots \mid A\{m\}))$

3. **At most m-times:**

$$A\{, m\} \rightarrow (A\{m\} \mid A\{m-1\} \mid \dots \mid A\{1\} \mid A\{0\}) \text{ with } A\{0\} = \epsilon \quad (3.3)$$

The expression is only syntactical sugar for $A\{, m\} = A\{0, m\}$, which is defined in Equation (3.2).

4. **At least n-times:**

$$A\{n, \} = A\{n\} \cdot A^* = A_1 \cdot A_2 \cdot \dots \cdot A_{n-1} \cdot A_n \cdot A^* \quad (3.4)$$

The expression has to be at least n letters long. Because A^* is greedy, it tries to match as much as possible. For the lazy equivalent $A\{n, \}?$ only $A^*?$ has to be lazy.

PERMUTE

The permute statement permutes all expressions inside the parenthesis. It can hold from 1 to $n \in \mathbb{N}$ different or equal expressions. Let A be an expression where $A_1, A_2 \dots$ doesn't need to be the same expression.

$PERMUTE(A_1, A_2 \dots A_n)$ the result of the permutations are exactly $n!$ different expression. The first element can be placed at n different positions, second only at $n-1$, etc. One expression is in the same order as the element inside the permute list. $A_1 A_2 \dots A_n$. Every $n!$ permutation is then concatenated with an OR-quantifier.

For example, let's look at the expression $PERMUTE(A, B, C)$. First, we have to get all different permutations, which should be six because we have three elements $3! = 6$.

$$permutations = ((A B C), (A C B), (B A C), (B C A), (C A B), (C B A)) \quad (3.5)$$

After concatenating the permutations with the OR-quantifier, we have the result:

$$result = ((A B C) | (A C B) | (B A C) | (B C A) | (C A B) | (C B A))$$

Since the algorithm is sensible for the position of each variable, it does matter in which order the permutations are created. The preferred option is the expression without any change ($A B C$). The permutation is like a position-anchored pattern sequence, where certain elements maintain their relative positions for longer periods until changing their original position.

3.2.2 Linearisation

Every letter inside the regular expression should be unique. To do this, give each of the letters from the left to the right a number from 0 to n . A letter can be duplicated but by being unique with its index.

3.2.3 NFA construction

The input for the NFA construction is the linearised pattern with the quantifier $*$, $+$, $|$.

I also want to introduce the attribute of an expression called nullable. An expression is nullable if and only if the regular expression has ϵ in its language. This means that the empty word is a valid match for this expression. For example A^* is nullable but $A+$ would not.

1. **starting symbols:** Giving the pattern the leftmost child of the expression is the starting symbol. If the leftmost expression is nullable, there could be more than one starting symbol. If the whole expression is nullable, the starting symbol is also terminating. In the case of multiple starting symbols, a penalty for each possible symbol has to be calculated. The first condition is its position. The leftmost child gets a penalty of one, and the second leftmost child a penalty of two, and so on. If the leftmost child is lazy, the penalty swaps and the next symbol gets the lowest penalty if it's not lazy. If all starting symbols are lazy, the rightmost starting symbol has the lowest penalty.

2. **terminating symbols:** In contrast to the starting symbols in 1, the terminating symbol is the rightmost child inside the expression. If this is nullable, there might also be more than one terminating symbol. The penalty is evaluated using the same rules described in 1.
3. **Connections:** A connection consists of a source symbol, a target symbol, and a penalty. For every symbol, find every successor (the target symbol). The connection with the first greedy successor gets a penalty of 0. Every other penalty is based on the position and its greediness. The source symbol and the penalty can uniquely identify every connection. The penalty is the key to the deterministic characteristics of these automata.
4. **state creation:** For every linearised symbol, a state is created. Because our automaton should only contain one starting and one terminating state, these two states are added as extra states.
5. **start transitions:** For every starting symbol create a transition from the start state to the according symbol with its calculated penalty.
6. **transitions:** For each connection, create a transition between the respective states. For every triple (s, t, p) s =source, t =target, p =penalty, and t in the terminal symbols, create an additional transition to the terminal state.
7. **remove transitions:** All states with no outgoing transitions that are not in the terminate state can be removed. This process should be done until nothing changes.

3.2.4 Properties

The resulting NFA has all the required properties for the usage inside the **WITH RECURSIVE** query.

- exactly one start state
- exactly one terminal state
- not minimal
- non-deterministic in symbols but deterministic with its penalty

3.3 Example

Let's look at the already linearized example with the regular expression $R = A1 B2^* A3$.

Starting symbols = {A1}

Terminating symbols = {A3}

Connections = {A1 → B2, A1 → A3, B2 → B2, B2 → A3}

We are creating a state for every letter inside the expression and adding the start and terminate state.

states = {1, 2, 3 start, terminate}

For the steps 5, 6 and 7 the connections are shown in the corresponding tables.

The resulting automaton is illustrated in Figure 3.1. Each edge is labeled with the corresponding

source	target	penalty
start	1	0

Table 3.1: Step 5 (Item 5)

source	target	penalty
start	1	0
A1	2	0
A1	3	1
A1	termi.	2
B2	2	0
B2	3	1
B2	termi.	2

Table 3.2: Step 6 (Item 6)

source	target	penalty
start	1	0
A1	2	0
A1	termi.	2
B2	2	0
B2	termi.	2
A3	termi.	0

Table 3.3: Step 7 (Item 7)

symbol and the penalty assigned to that transition. It is worth noting that no transition is assigned a penalty value of 1, despite a penalty of 2 being present for the state 2. This omission does not affect the correctness of the automaton, as the relative order of penalties remains valid. Such gaps in penalty values can occur when certain transitions are removed during step 7 of the algorithm, typically because they are no longer required for the final structure of the automaton.

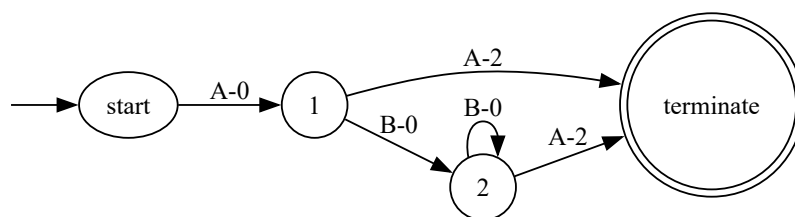


Figure 3.1: The Automaton generated by our algorithm. This is also the visual representation on demo the website for a match recognize based Compiler Explorer, which is part of Knüdelers thesis.

Implementation

After the theoretical background, the focus shifts to the implementation. The transpiler is written in Python and receives a `MATCH_RECOGNIZE` query as input. Since an SQL query is simply plain text, Python cannot directly interpret its syntax. To address this, and to enable more efficient processing of the query, the input is first parsed into an AST. This is accomplished using a parser generator called Antlr [8]. The result is an AST representing the SQL structure of the `MATCH_RECOGNIZE` query. For simplicity and easier maintenance, the generated AST is then translated into an internal representation of the query. Details on the reasons and methods behind this process can be found in Knüdelers [10] bachelor's thesis. In this work, the focus is primarily on the representation of the pattern clause, as it plays a crucial role in the construction of the automaton.

4.1 Regular expression syntax tree

As mentioned previously, a representation of the regular expression within the pattern clause is required. This representation must uniquely identify every possible expression. The algorithm supports only the basic quantifiers: Star, Plus, Cat, Or, and subexpressions. Figure 4.1 shows the class diagram for the regular expression syntax tree (RST). The RST is a recursive data structure that can theoretically grow to infinite depth, allowing an expression to contain as many subexpressions as needed. In this structure, everything is treated as an expression, and each expression must implement the methods `isNullable` and `isReluctant`.

- `isNullable`: An expression is nullable if the empty word exists inside the language.
- `isReluctant`: An expression is reluctant if it is not trying to match as much as possible. For example, `(A B)*` is not reluctant because it is greedy and tries to match the subexpression `(A B)` as often as possible. In contrast, `(A B)*?` is reluctant and prefers fewer matches.

An expression exists either of another subexpression or is itself a leaf. A leaf is the end of the data structure and contains the information about the variables or the matching string. Therefore, the leaf holds information about what should be matched, the nodes about how it should be matched, and which options exist. A leaf can have two different options.

- `Epsilon` is the empty word.
- `Variable` is a string that should be matched. The exact definition of the variable is defined inside the `DEFINE` clause, but for the automaton, the definition is not relevant. The variable holds the information about the `name` of the variable and the `index`, which is given at the linearisation of the algorithm. To tell the algorithm if the variable should be preferred

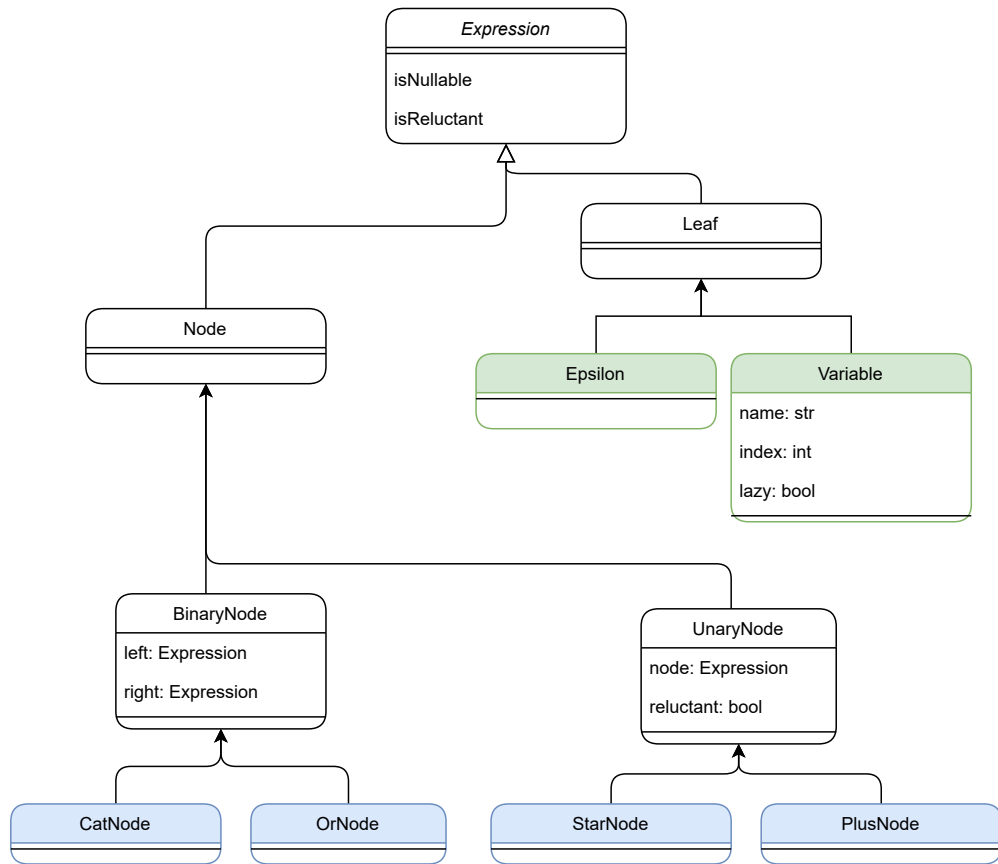


Figure 4.1: RST-Class diagram for the internal representation of the regular expression in Python.

if more than one successor is a possible match. The boolean attribute `lazy` is needed, however the exact usage of the attribute is presented later during the construction of the automaton.

A node is a more specific type of expression. It is responsible for the structure and quantifier inside the regular expression. Every quantifier or subexpression (parenthesis) is either directly or indirectly encoded inside these nodes. The nodes are categorized into two different sub-nodes with different attributes.

- **BinaryNode**: is a node that contains two different expressions: one left expression and one right expression.
- **UnaryNode**: contains only one expression, and the attribute `reluctant` is needed to tell if it should be reluctant.

The four different quantifiers `Cat`, `Or`, `Star`, and `PlusNode` are represented by four different classes. `Cat` and `Or` Nodes have two different expressions: one left and one right expression.

```
rst = CatNode(left=Variable("A"), right=StarNode(node=Variable("B")))
```

Figure 4.2: (A B*) as a RST-representation

Suppose we want to concatenate more than two variables or expressions. A list of cat-Nodes does not represent this, but because of the recursive design, a cat-Node can hold another cat-Node inside and this cat-Node also.

star and plus-Node only have one expression, which is surrounded by the quantifier. If the expression is a variable, we can represent the expression A*. However, a subexpression is also possible, in that case the expression is surrounded by parenthesis, and then the quantifier is applied (expr.)*.

Let us look at a small example Figure 4.2. The regular expression A B* should be represented by the RST. First, we can see that we have one quantifier and one indirect quantifier. B* has the Star quantifier, and the other variable is concatenated with B.

The result is a CatNode with a variable A as the left argument, and the right is a StarNode, which holds the other variable B.

4.2 AST as an input

After presenting the data structure, the focus shifts to the generation of the RST. In SQL, the regular expression is represented as plain text. With the help of the parser generator ANTLR and the work of Knüdel [10], the regular expression is first transformed into an abstract syntax tree (AST). However, this AST has two main drawbacks: it is verbose and contains unnecessary information that is not needed for the generation of the automaton. In contrast, the data structure presented earlier is compact and captures all the necessary information required to represent a regular expression.

The second problem is that the AST still contains some quantifiers that serve merely as syntactic sugar for other expressions, such as PERMUTE or the lazy quantifier "?". Therefore, in the next step, all unsupported quantifiers must be eliminated, and the AST must be transformed into the RST. To illustrate the difference in complexity, Figure 4.4 shows the same regular expression as in Figure 4.2. Without further explanation, it is difficult to determine which regex is represented. Moreover, the additional information in nested RowPattern and Term nodes does not contribute meaningful value to the actual regex representation.

4.2.1 AST-visitor

What do we need to generate the RST? A typical and widely used design Pattern is the visitor principle. Performing operations on a data structure without changing the classes is possible. Every "node" or class visits every object of the structure and runs a specific operation to change the class or generate something new. The structure of the operation and the data structure are

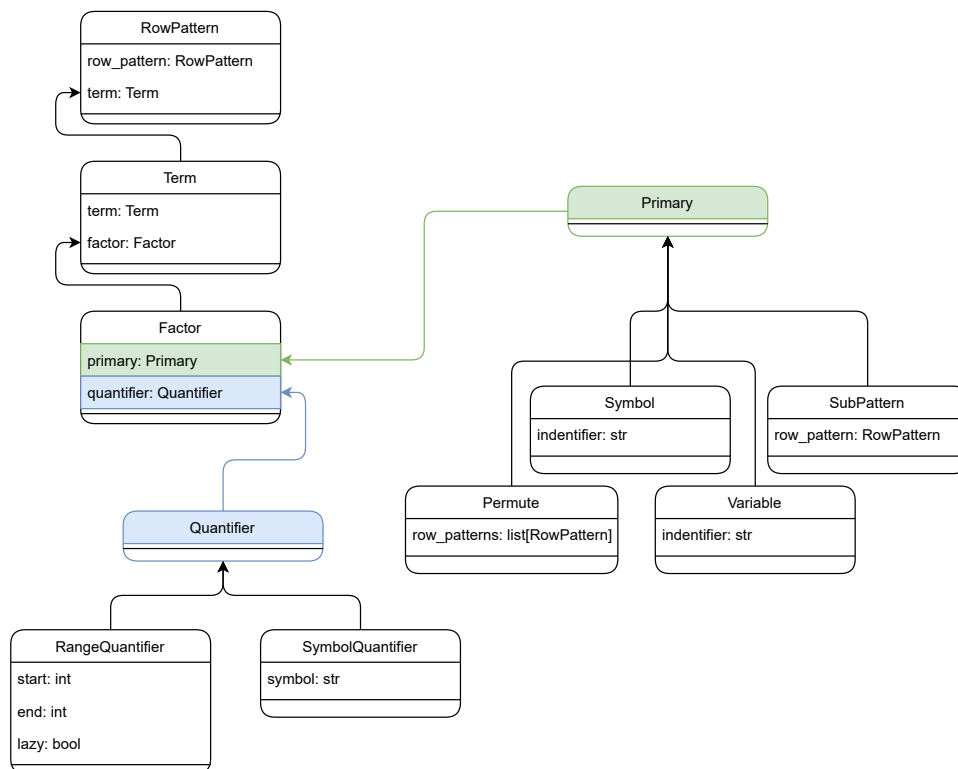


Figure 4.3: A subset of the AST class diagram representing the match recognize syntax. In the graphic only the structure of the pattern is shown, which is the relevant input for my implementation.

```

1 RowPattern(
2     row_pattern=None,
3     term=Term(term=Term(term=None,
4                     factor=Factor(primary=Variable(identifier='A'),
5                                 quantifier=None)),
6     factor=Factor(primary=Variable(identifier='B'),
7                   quantifier=SymbolQuantifier(symbol='*'))))
  
```

Figure 4.4: (A B*) as an AST representation

separated so that a part of the code can be modified more easily without affecting something else.

A `Visitor` class is needed to implement different methods for the object types inside the data structure. Inside these methods, the logic for this object is implemented. The visitor accepts the data structure and calls recursively for every object the corresponding method. The visitor is primarily used in the traversal and processing of tree-like structures, like abstract syntax trees, in compiler development or complex data analysis.

4.3 AST-Visitor in Python

In Python, I defined a class named `AstVisitor`. The attribute `self.index` is used for the linearisation of the variables. The smallest index be can configured using the constant `START_IDX`. For every node in the AST Figure 4.3, a method is defined with the following naming convention: `visit_<object name>`. The class entry method is the `visit` method.

```
1 class AstVisitor:
2     def __init__(self):
3         self.index = START_IDX
4
5     def visit(self, node):
6         if node is None:
7             return None
8         method_name = 'visit_' + type(node).__name__
9         visitor = getattr(self, method_name)
10        return visitor(node)
```

Listing 4.1: Visit method of the `AstVisitor` class.

The `visit` method accepts any object of the AST. The method name is evaluated with the prefix `visit_` and the object name. `getattr` returns the method with the given name, which is, in our case, the corresponding method for the passed object. As a result, the result is the result of the visitor method. After the whole tree is traversed, this method gives us the final RST. Inside the AST the information about which type of concatenation is present, is encoded in the class usage.

RowPattern:

If the `RowPattern` object is used, an OR concatenation is present, and the `Term` represents a regular concatenation. Therefore, from the `RowPattern` object, a `RST-OR-Node` should be built. This logic has to be present in the function for the `RowPattern` (`visit_RowPattern(object)`).

```
1 def visit_RowPattern(self, node: ast.RowPattern):
2     if node.row_pattern is None:
3         return self.visit(node.term)
4     return rst.OrNode(self.visit(node.row_pattern), self.visit(node.term)
5                       )
```

The `RowPattern` has two attributes, `row_pattern` and `term`. If the row pattern is none, the relevant information is inside the term attribute, and the visitor is called for this attribute. If both attributes are present, an Or-Node is built. An Or exists with the left and right side of the Or quantifier. Because the evaluation goes from the leaf to the root node, the object inside the attributes has not been evaluated yet. But we promise that the visitor will return the correct `rst` for this subexpression. As a result, we will get the correct `rst`. Therefore, we can return an Or-Node and, as an argument, the result from the `self.visit` method is used.

Term:

In contrast to the `RowPattern`, `Term` builds a `Cat-Node`, which works with the same principle. If the term attribute is `None`, we call the visit on the factor attribute; otherwise, a `cat-Node` is built.

```
return rst.CatNode(self.visit(node.term), self.visit(node.factor))
```

Factor:

The `Factor` object represents an expression with a quantifier. As mentioned earlier, we must transform some quantifiers because the RST does not support them. This logic has to be implemented inside this method. A quantifier has two subclasses `SymbolQuantifier` and `RangeQuantifier`.

The `Symbol Quantifier` can contain four different types of symbols.

1. *** or *?:** The Star quantifier maps directly to a `Star-Node`, and to tell if the quantifier is lazy, a boolean named `reluctant` is passed to the `Star-Node`.

```
return rst.StarNode(self.visit(node.primary), reluctant=reluctant)
```

2. **+ or +?:** As the `star-Node`, the plus quantifier is building precisely the same as with the `plus-Node`

```
return rst.PlusNode(self.visit(node.primary), reluctant=reluctant)
```

3. **??:** Translating the lazy optional quantifier is impossible, but it can be reformulated as an `or-Node` as in Section 3.2.1 described.

```
return rst.OrNode(rst.Epsilon(), self.visit(node.primary))
```

The `Epsilon` class represents the empty quantifier, and because it is at the first position of the `or-Node`, it is preferred, and the automaton tries at first not to match the expression `node.primary`.

4. **?:** The only difference for the optional quantifier is the order inside the `or-Node`. The expression `node.primary` comes first, and as a second argument, the `epsilon`.

The **RangeQuantifier** is entirely not supported by the RST and must be translated. The quantifier repeats the expression the given number of times. A repetition of an expression is just an n -times concatenation of the given expression. But the concatenation has to be built into the depth. We start by creating a cat-Node, and another cat-Node must be built for the right argument. We continue this process until we have n cat-Nodes nested inside each other.

```

1 def build_cut_range(self, primary, times):
2     if times == 1:
3         return self.visit(primary)
4     elif times == 0:
5         return rst.Epsilon()
6     build_cut = self.build_cut_range(primary, times - 1)
7     return rst.CatNode(build_cut, self.visit(primary))

```

For range quantifiers, possible arguments can be passed on how often the repetition continues.

exactly n iterations

We build a cat-Node with exactly n repetitions.

n or more iterations

For the n repetitions, we can build a cat-Node again with n nodes and then concatenate a star-Node with the same expression. Because a star Node allows from 0 to infinite matches, they depict the same regular expression

```

    build_cut = self.build_cut_range(primary, quantifier.start)
    return rst.CatNode(build_cut, rst.StarNode(self.visit(primary),
                                              reluctant=quantifier.lazy))

```

Between n and m iterations or between 0 and m (inclusive) iterations

In this case, we cannot just concat the expression n times. Because every different number of matches should be possible, we have to concatenate every possible length of the repetition of the expression and use the or-Node to connect every built nested cat-Node. The denotation of the range quantifier being lazy changes the order of the OR-Nodes. With a greedy range quantifier, the first concatenation inside the OR-Node is with a length of m and ends with a small cat-Node with a length of n . The order gets reversed with the lazy quantifier (?), and we start with the smallest one.

To build this expression, I created a function called `build_or_range` (Listing 4.2), which takes three parameters. `primary`, which is the expression to build the nodes for, `times` how many or-Nodes we need, `concatTimes` the number of concatenations, and finally, a boolean `reverse` to tell in which order the expression should be built. The function `build_or_range` is recursively called the amount of times specified in the `times` argument. For every recursive call, a concatenation of the expression is created. How often the expression has to be concatenated is calculated with `concatTimes + times - 1`. Another way to formulate this would be to add the lower bound (n) plus the current iteration of the or-Node. Therefore, a different concatenation length starting from n until m is built. For example the regular expression $A\{2,4\} = ((AAA))(((AAA))(AA))$

```

1 def build_or_range(self, primary, times, concatTimes, reverse=False):
2     if times == 1: # no or-Node is needed for one concatenation
3         return self.build_cut_range(primary, concatTimes)
4
5     if reverse: # order changes inside the or-Node
6         build_cut = self.build_cut_range(primary, concatTimes + times - 1)
7
8         return rst.OrNode(build_cut,
9                             self.build_or_range(
10                                primary,
11                                times - 1,
12                                concatTimes, W
13                                reverse=True)
14                            )
15     build_cut = self.build_cut_range(primary, concatTimes + times - 1)
16     return rst.OrNode(self.build_or_range(primary, times - 1, concatTimes),
17                       build_cut)

```

Listing 4.2: build_or_range function for building a given number of or nodes with the concatenation of a expression inside.

SubPattern

The SubPattern does not have any relevant information, and the visitor is directly called for the row_patten attribute.

Permute

The permute class gives us a list of expressions. Every permutation of the list should be matchable from the automaton.

```

1 permutations = itertools.permutations(permute.row_patterns)
2 and_nodes = []
3 for perm in permutations:
4     and_nodes.append(self.build_expression_from_ast_list(list(perm),
5                                                             rst.CatNode))
6
7     return self.build_expression_form_rst_list(and_nodes, rst.OrNode)

```

The in-built Python function `permutations` from `itertools` return a list with every permutation of the passed list. Then, for every permutation, which is a list of AST-Nodes, the expressions are concatenated inside a cat-Node. After this step, a list of cat-Nodes with every permutation is present, and now we have to connect those Nodes with or-Nodes. **Variable**

When a **Variable** node is visited, the leaf of the tree is reached, and no further recursive calls are necessary for this object. A **Variable** in the RST representation can then be returned. An index is assigned to the variable and incremented by one to achieve the linearization of the pattern. In the end, each variable has a unique index.

```

self.index += 1
return rst.Variable(node.identifier, index=self.index)

```

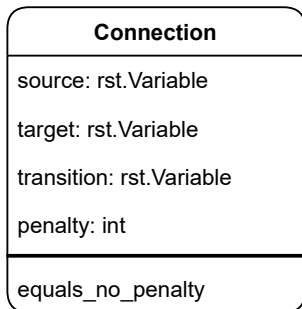


Figure 4.5: Dataclass Connection

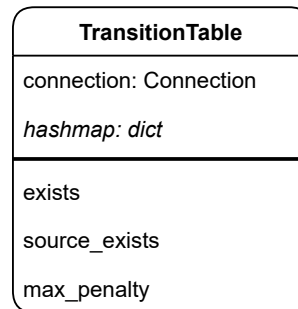


Figure 4.6: Class TransitionTable

After the whole tree is visited, the final RST is created, and it does not lose any information. This tree is used as the input for the NFA algorithm and the generation of the transition table.

4.4 NFA algorithm in Python

To create the transition table, I have to introduce two new classes representing the transition table and the connections inside. A Connection (Figure 4.5) is a transition between two states. As the algorithm stated, for every variable a state is created. However, I am not going to define every state explicitly. The states are indirectly defined inside the connections. If a variable exists inside a connection there is also a corresponding state. A connection has a source, target state and the transition with which this connection is taken.

A Transition table (Figure 4.6) is a list of connections with some useful helper functions to help build the transition table. The hashmap is used for faster lookup inside the connections.

Like the algorithm, I split the code into different steps, and the logic is inside those functions. As an input the algorithm takes a linearised pattern in the RST-representation and will return a transition table.

1. starting_symbols = find_symbols(lin)
2. terminate_symbols = find_symbols(lin, terminate=True)
3. transTable = gen_transition_table(lin)
4. with_start_end = add_start_end_state(starting_symbols, terminate_symbols)
5. without_end = remove_end_states(with_start_end)
6. calculate_the_correct_indices(without_end)

starting and terminating symbols

I defined a helper function for the algorithm to find the starting or terminating symbols in a given expression. As an input, it takes an expression and a boolean to tell if the starting or terminating symbols should be found. The result is a list of Variables that are either starting or terminating symbols inside the expression. The function recursively visits every node, and each node is given different operations. The logic is defined with the `match` expression for the different classes. If a variable is visited, it will be added to the found nodes and returned. The unary nodes will go one step deeper inside the recursion.

CatNode

If the starting symbols are searched, they first have to be looked up on the left side of the cat-Node, and the terminating symbols are on the right side. But if this side is nullable, the first symbols of the other side must be searched for. For example, if we look at the expression $A B$, the first starting symbol is found on the left side and is the variable A because A is not nullable, and B can not be a starting symbol. Unlike the expression $(A*B)$, A is nullable; therefore, A and B are potential starting symbols, but A should have a lower penalty because it comes first inside the expression.

OrNode

Both sides have to be explored for the starting symbols inside the or-Node, but the order matters if the left or right side is visited first. For example, in the cat-Node, the left side is visited first for the starting symbols and the right side for the terminating symbols.

```
1 left = find_symbols_recursive(children[0], starting_nodes, terminate)
2 right = find_symbols_recursive(children[1], None, terminate)
3 if len(left) == len(starting_nodes) and not terminate:
4     set_lazy(right)
5 return left + right
```

I calculate the starting symbols for both sides and if no new symbol is added to the left side, all variables on the right side must be marked as lazy. If no new symbol is added, it means the left side is an epsilon expression, and for this or-Node, no match is preferred. For example the expression $A? B \rightarrow (A | \epsilon) B$ both A and B are starting symbols and A is preferred over B , but then we make A reluctant $A?? B \rightarrow ((\epsilon | A) B)$, A comes first in the expression but if we look at the logic of the expression the epsilon is matched first. Therefore A has to be marked as lazy.

After all the starting symbols are found they have to be ordered correctly.

```
copy = result.copy()
result.sort(key=lambda x: (int(x.lazy), copy.index(x)))
```

They are sorted first after the criteria of being lazy and then after their position. We get a list in the order in which the variables are matched. The first variable is more likely to be matched than the second one, etc. For correctness, I have to mention if the terminating symbols want to be found the resulting list has to be reversed because not the first symbols are interesting but the last ones.

With the help of this function for the automaton, the starting and terminating nodes are determined.

```

1 connections = []
2 for var in left:
3     for pen, varR in enumerate(right):
4
5         if trans_table.exists(Connection(var, varR, varR),
6             exclude_penalty=True):
7             continue
8
9         exists = trans_table.source_exists(source=var)
10        isPenaltyUnsure = trans_table.max_penalty(var) is None
11
12        pen_offset = 0
13        # the penalty has to be greater than the existing ones
14        if exists and not isPenaltyUnsure:
15            pen_offset = trans_table.max_penalty(var) + 1
16
17        connections.append(Connection(source=var.copy(),
18            target=varR.copy(),
19            transition=varR.copy(),
20            penalty=pen_offset + pen))

```

Listing 4.3: build_connection function definition

gen_transitionTable

The function takes an rst-tree as an input and returns a TransitionTable; like the other functions, every node is visited, and during the transversal, the transition table is built. With the match statement for the different nodes, different code is executed

Leaf

An empty transition table is returned.

CatNode

The transition table is created for the left and right sides of the cat-Node. However, the connections between these transition tables have to be created. For this, we can determine the possible terminating symbols for the left side of the expression and the starting symbols for the right.

```

left = find_symbols(expression.left, terminate=True)
right = find_symbols(expression.right)

```

Now, we have two lists. For every variable in the list called left, a connection to every variable in the list right has to be created. For this process, I created the function build_connection (Listing 4.3), as an input it takes to lists, left and right, and a transition table to calculate the penalty. After the calculation, the function returns the new connections. With the two nested for loops, every combination from every variable on the left to every variable on the right is visited. First, it is checked if this connection already exists; if so, it is not added. For better

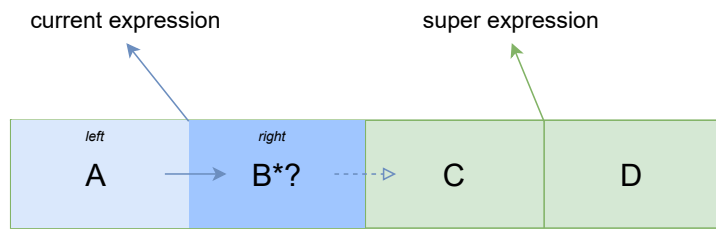


Figure 4.7: Special case there the penalty cannot be determined during the visit of the sub expression.

readability, I used two boolean variables, one to check if a connection in the given transition table exists with the current left variable as a source node. If this state exists in the transition table, the `pen_offset` is the maximal penalty for this state. Then, I can append the connection to the list with the arguments source, target, transition, and penalty. The transition is the variable with which this transition is taken, and the penalty is the existing maximal penalty for this state (`pen_offset`) plus the current index of the variable. Because the variables are all correctly ordered inside the list. The states with a lower index also get a lower penalty because the index of the variable in the right list is added to the offset, resulting in the penalty.

Let's return to the generation of the transition table; new connections are created for the found variables.

```
connections = build_connection(left, right, transitions)
```

But it would be too simple if this was already the result. There is a special case if the right side of the expression is reluctant. I want to explain this as an example with the regular expression $A B^* C D$.

At one stage in the generation, we will have as a current expression a `CatNode` with A and B (marked in blue in Figure 4.7). With the `find_symbols` function, the terminating symbols from A and the starting symbols from B are evaluated, and the connections are created. In this case, there is only one connection from A to B . Because of the nested tree structure of the syntax tree at the moment, the algorithm doesn't know what comes "after" the B^* , this part is in the super expression. But if the variable B is not the end, there will be a connection between B and C (marked in green). Because B is nullable, a connection from A to C must also be created. This connection would get a penalty of 1 because the connection $A \rightarrow B$ has a penalty of 0. In the case of B being reluctant, this is wrong, and the connection $A \rightarrow C$ should have a penalty of 0.

To fix this problem, I have to check if the right expression is reluctant, and if so, the penalty from every new connection is set to -1 and will be evaluated later.

OrNode

No extra logic is needed here. The function for right and left expressions is called, and the two resulting transition tables are combined and returned.

unaryNode

The logic for the Plus and Star nodes can be combined within the unary node. As always, the transition table for the node inside the unary node is evaluated. The starting and terminating symbols are determined for this node, and a connection is built from every last node to every first node. This is essential because the expression inside the unary node can be repeated multiple times. As standard, this node is greedy at looping on itself and has a higher penalty, but with the expression being reluctant, there are two special cases.

1. Like the cat-Node in Figure 4.7, the new connections cannot yet have the right penalty because the transition away from the unary node will be preferred over the loop. Therefore, every penalty from the new connections has to be set to -1.
2. The first problem solves the issue from the connection that loops over the expression. However, the connections inside the expression which are created during the recursive call are not all valid too. Every connection with the target being inside the expression's terminating symbols must be set to -1. This is for the same reason as before: We want to leave the loop, and I can calculate the connections away from the loop later.

```
1 transitions = gen_transition_table(expression.node)
2 if expression.isReluctant():
3     for con in transitions:
4         if con.target in last:
5             con.penalty = -1
```

Let's look at the regular expression: $(A B)^*$?. With the recursive call, the transition $A \rightarrow B$ is created with a penalty of 0. But we want to end the match as fast as possible, and the connection $A \rightarrow \text{terminate}$, which is added later, should be preferred. The solution for this is to set every connection inside the create transition table to -1, and this penalty will be calculated later on.

add_start_end_state

After creating the transition table, the start and end state have to be added. The resulting therefrom connections have to be added to the transition table. For the start state, a connection from the start state to all the starting variables is created, and this is done with the help of the `build_connection` function.

For the connection to the terminate state, a connection from every state with an outgoing connection to a state inside the terminating symbols must be created. For example, let's say A is our only terminating symbol, and we have the connection $B \rightarrow A$. We would build a new connection $B \rightarrow \text{terminate}$.

Now, we can deal with the unset penalties, which are -1. Because every connection, including start and end connections, is created, we can set all other penalties that are not determinable because a potential super expression was not yet created. For every connection with the penalty of -1, one gets the subsequent greater penalty.

remove_end_states

This step aims to remove all states without any outgoing transitions that are not in the terminal state. For every connection inside the transition table, if the target state has outgoing connections. This is done with the help of the hashmap of the transition, which I mentioned earlier. The hashmap is built to reduce time for lookup. A state with outgoing edges is present in the hashmap with a list of the target states. Therefore, we can check if the target of the current connection is inside the hashmap. If not, the connection is removed from the transition table. This process is repeated until nothing changes in the transition table.

calculate_the_correct_indices

In the final recursive CTE, working with the string representation of the variables is no longer necessary. Therefore, at the end of the transition table creation, each state and transition is assigned a numeric representation. The states already have almost the correct indices, as they are derived from the linearization process. The starting state is assigned index 0, and the subsequent states are assigned indices from 1 to n . The terminal state is assigned the index $n + 1$.

For the transitions—corresponding to the variables defined in the **DEFINE** clause—each distinct transition is assigned an index corresponding to 2^x . To support Tim Findling in working with the new indices, and for performance reasons, a mapping from variable names to their assigned indices is provided. This mapping can be retrieved from the transition class.

Another useful function is **max_penalty_bits**, which returns the number of bits required to represent the maximum penalty in the transition table.

With this, the transition table is complete and can be used for further processing.

4.5 Visualisation

With the generation the core step of the thesis is done, but there are some steps which are in between our work. Marcel Knüdeler builds a front end as a demo for the transpiler [14]. To visualize the regular expression, a picture of the automaton is shown. This helps understanding what is going on under the hood. For the visual representation of the automaton, I used the Python library automathon [15], which is able to create a PNG of the given automaton. However, a PNG is unsuitable for Websites because an image should be scalable. Therefore, an SVG (Scalable Vector Graphics) might be more sophisticated, but this was not supported. I was able to add this feature to this repository on Github, and it got accepted by the author. However, there is not a new release from his side, and therefore, I had to find another way to produce this SVG. With the creation of the PNG, there is also a ".gv" file, which represents the graphic that can be understood by the library Graphviz [16]. With that, I could create, with the help of Graphviz, this image as an SVG for the Website.

Results

This thesis successfully developed and implemented an algorithm based on Glushkov's construction method to convert regular expressions, as defined in the `MATCH_RECOGNIZE` PATTERN clause, into a corresponding deterministic finite automaton (DFA). The implementation begins with an Abstract Syntax Tree (AST) representation of the input pattern, which is generated from the input string. A visitor pattern, as described in Chapter 4, is then used to traverse this AST.

The algorithm supports key functionalities including the handling of concatenation, alternation, basic quantifiers, as well as the expansion of range quantifiers (e.g., `{m,n}`) and `PERMUTE` clauses. The final output is a representation of the automaton's transition function in the form of a transition table, making it suitable for further processing or for translation into a recursive SQL query.

5.1 Limitations

5.1.1 Recursion limit

The recursion limit in Python is limited by a fixed number, which depends on the system [17]. To determine the current limit, it can be done by `sys.getrecursionlimit()`, which is on a Windows pc at 1000. This limits our algorithm because the depth of the abstract syntax tree can not be greater than 1000. If so, the program will raise an `RecursionError: maximum recursion depth exceeded`. Increasing the recursion limit with the function `sys.setrecursionlimit(limit)` is possible. But it has to be done carefully because if it is set too high, the program could run out of memory and crash.

The issue will most likely appear with the range quantifier and the permute statement because these are translated in many or and cat-Notes; therefore, a relatively short expression will throw an error. For example `A B{0,1000}` or `A PERMUTE(A,B,C,D,E,F)` both are not possible with a limit of 1000. However, another question is whether there is a use case for a permute statement with that many combinations.

5.1.2 Runtime

For "normal" regular expression, there are no problems running the algorithm and getting a result quickly. However, by investigating the limits, the runtime increases with very complex or deep expressions. Like in the recursion limit section, the runtime might increase when creating

a very deep syntax tree. A statement closely exceeding the recursion depth might likely not finish in a qualified time, and the algorithm might run seconds to finish.

5.2 Conclusion

In summary, this thesis demonstrates that it is feasible to construct a transition table from a regular expression given as a string. By analyzing the positions and combinations of symbols within the expression, each transition is assigned a penalty value to explicitly determine the path taken during evaluation. While the resulting automaton is non-deterministic in structure, the use of penalties ensures deterministic behavior during execution.

All non-essential quantifiers are systematically replaced with equivalent constructs to simplify the automaton. The resulting transition table is then incorporated into a `WITH RECURSIVE` query, where it functions as an inline table used for identifying matching rows. To facilitate query processing without relying on string representations, all variables are mapped to corresponding integer identifiers.

Aside from the previously mentioned limitations, the algorithm produces a correct deterministic finite automaton (DFA) that faithfully represents the original regular expression.

The generated automaton forms the core logic for translating the `PATTERN` clause of a `MATCH_RECOGNIZE` statement. This approach enables sequence pattern matching using regular expressions via a `WITH RECURSIVE` query, providing a practical alternative for database systems lacking native support for the full `MATCH_RECOGNIZE` specification, and serving as a foundation for further comparative analysis or extensions.

5.3 Future Work

To further improve the system, several enhancements could be made—particularly with regard to performance and scalability. One potential improvement involves optimizing the traversal of the syntax tree. Currently, the tree is traversed multiple times during the construction process, which impacts runtime efficiency. Reducing the number of traversals would not only enhance performance for larger expressions but would also allow the system to operate more effectively on machines with limited computational resources.

Another important consideration is the recursion limit encountered during the generation of large pattern constructs, such as `B{,1000}`. These cases currently rely on a recursive structure that can exceed Python's default recursion depth. A promising alternative would be to implement an iterative approach, which avoids stack overflow and improves robustness. Since such constructs are essentially repeated concatenations of the same element, their structure is predictable and lends itself to systematic iteration. Implementing a more efficient connection mechanism for such cases would significantly improve the handling of large-scale patterns.

To compare the effectiveness of the proposed approach, performance benchmarks across various Database Management Systems (DBMSs) could be conducted. These benchmarks would as-

sess how well the translated `WITH RECURSIVE` queries perform relative to native `MATCH_RECOGNIZE` implementations. Such performance testing would provide valuable insights into the scalability and efficiency of the approach across different systems and configurations. Additionally, an important optimization for future development would be the minimization of the automaton. By applying minimization techniques, the size of the resulting automaton could be reduced, thereby decreasing the number of rows produced in the transition table. This would not only improve memory efficiency but also enhance query performance, especially when working with large datasets or complex patterns.

Bibliography

- [1] Oracle. *Pattern Recognition With MATCH_RECOGNIZE*. 2025. URL: https://docs.oracle.com/en/middleware/fusion-middleware/osa/19.1/cqlreference/pattern-recognition-match_recognize.html#GUID-EC0F2E34-5621-4F58-A6FD-C02B28893D00 (visited on 02/23/2025).
- [2] DuckDB. *DuckDB Documentation*. 2025. URL: <https://duckdb.org/docs/stable/> (visited on 03/23/2025).
- [3] MySQL. *MySQL*. 2025. URL: <https://www.mysql.com/de/> (visited on 03/23/2025).
- [4] Trino. *Trino Documentation*. 2025. URL: <https://trino.io/docs/current/> (visited on 03/23/2025).
- [5] Microsoft. *Transact-SQL Reference*. 2025. URL: <https://learn.microsoft.com/en-us/sql/t-sql/language-reference?view=sql-server-ver16> (visited on 03/23/2025).
- [6] Ansi. *The SQL Standard*. 2025. URL: <https://blog.ansi.org/sql-standard-iso-iec-9075-2023-ansi-x3-135/> (visited on 04/23/2025).
- [7] Dušan Petković. "Identifying Possible Financial Frauds using SQL Row Pattern Recognition". In: *International Journal of Computer Applications* 975 (), p. 8887.
- [8] Antlr4. 2025. URL: <https://github.com/antlr/antlr4/blob/master/doc/index.md> (visited on 03/23/2025).
- [9] Lele Gaifax. *Pglast*. 2025. URL: <https://pglast.readthedocs.io/en/latest/> (visited on 04/23/2025).
- [10] Marcel Knuedeler. "Bringing Row Pattern Matching to DuckDB: Parsing and Frontend". Department of Computer Science, Database Systems Research Group. PhD thesis. Tübingen, Germany: University of Tübingen, Jan. 2025. URL: <https://db.cs.uni-tuebingen.de/theses/2025/marcel-knuedeler/>.
- [11] Investing.com. *USD EURO Historical Data*. 2025. URL: <https://www.investing.com/currencies/usd-eur-historical-data> (visited on 04/12/2025).
- [12] Uwe Schöning. *Theoretische Informatik - kurz gefasst*. London: Spektrum Akademischer Verlag, 2008. ISBN: 3827418240.
- [13] V M Glushkov. "THE ABSTRACT THEORY OF AUTOMATA". In: *Russian Mathematical Surveys* 16.5 (Oct. 1961), p. 1. doi: [10.1070/RM1961v016n05ABEH004112](https://doi.org/10.1070/RM1961v016n05ABEH004112). URL: <https://dx.doi.org/10.1070/RM1961v016n05ABEH004112>.
- [14] Uni Tübingen. *Compiler Explorer*. 2025. URL: <https://db.cs.uni-tuebingen.de/matchrecognize/> (visited on 04/23/2025).
- [15] Rohaquinlop. *Automathon: Finite Automata and Regular Expressions*. Accessed: 2025-04-03. 2025. URL: <https://github.com/rohaquinlop/automathon>.

- [16] Graphviz. *Graphviz: Graph visualization*. Accessed: 2025-04-03. 2025. URL: <https://graphviz.org>.
- [17] *System-specific parameters and functions*. 2025. URL: <https://docs.python.org/3.11/library/sys.html#sys.setrecursionlimit> (visited on 04/04/2025).