FOM Hochschule für Oekonomie & Management

university location Bonn

Bachelor Thesis

in the study course Wirtschaftsinformatik

to obtain the degree of

Bachelor of Science (B.Sc.)

on the subject

Development of a Query Language for Full-Text Search in Relational Databases

by

Sebastian Bunge

Advisor:	Prof. Dr. Peter Steininger
Matriculation Number:	539441
Submission:	October 12, 2022

Contents

Index of Figures

Index of Tables

Index of Abbreviations

APT Automatically Programmed Tools

AST abstract syntax tree

DDL Data Definition Language

DSL Domain-Specific Language

EBNF Extended Backus-Naur Form

GPL General-Purpose Language

HTML Hypertext Markup Language

MS Microsoft

PDF Portable Document Format

SQL Structured Query Language

XML Extensible Markup Language

Index of Symbols

p	precision
r	recall
n	number of relevant retrieved documents
d	total number of retrieved documents
v	total number of relevant documents
F_β	weighted harmonic mean
β	nonnegative weight

Index of Formulae

Index of Code Listings

1 Abstract

Abstract

2 Theory

2.1 Full-Text Search

Commercial database management has long focused on structured data and the industry requirements have matched those of structured storage applications quite well. The problem is that only a small part of the data stored is completely structured, while most of it is completely unstructured or only semi-structured, in the form of documents, emails, web pages, etc. (cf. HAMILTON, NAYAK 2001, p. 7) Full-text search describes a search technique in which all words of a document or a full-text database are matched with search criteria, whereby not only exact matches but also word reflections and the like can be searched. A full-text database, as opposed to a regular bibliographic database, contains not only metadata but also the complete textual content of books and similar documents. (cf. TENOPIR, RO 1990, pp. 2-3)

With large amounts of data, matching every word of all entries is time-consuming and non-performant. To improve this process, a full-text search is divided into an indexing and query phase. In the indexing phase, all words found to be irrelevant, e.g. 'and' or 'the', are ignored by matching them against stoplists, words are normalized, e.g. the capitalization of words, and are merged into an index. (cf. COLES, COTTER 2009, p. 11) In the query phase, full-text query predicates are used to execute search queries. These allow not only a search for exact matches but also generational forms. Generational forms can be, for example, words that stem from the same word or alternative search terms using a language-specific thesaurus. A query processor then calculates the most efficient query plan which delivers the required results. The previously created index is searched for documents and text passages that match the search, and the results are returned in a ranked order. (cf. COLES, COTTER 2009, pp. 11-12)

To determine a rank for a search result the quality has to be measured. Two key metrics are used when measuring the quality of search results: precision p and recall r . Precision is defined as the relation of relevant search results to irrelevant search results. If, for example, many results are desired about the Jupiter moon Europa, the search term 'Europa' has low precision, since results for the continent 'Europe', as well as for the mythological figure and the moon are displayed. The search term 'Europa Moon' will again have higher precision. Algebraically, precision can be represented as in Formula 1, where n represents the number of relevant retrieved documents and d represents the total number of retrieved documents.

Formula 1: Precision

$$p = \frac{n}{d} \tag{1}$$

Source: COLES, COTTER 2009, p. 14

The recall is defined as the relation between relevant search results and relevant documents that were not displayed. For example, if five documents in a database deal with the moon Europa and only two are displayed in a search recall is low. Formula 2 shows the mathematical definition, where v represents the total number of relevant documents.

Formula 2: Recall

$$r = \frac{n}{v} \tag{2}$$

Source: COLES, COTTER 2009, p. 14

Although it is nearly impossible to maximize both recall and precision it is still relevant to keep both values as high as possible. Formula 3 offers the possibility to prefer one of the two metrics precision and recall when calculating the quality of a search result. The nonnegative weight β weights both metrics equally for a value of 1.0. A value less than 1.0 prefers recall, while a value above 1.0 prefers precision.

Formula 3: Weigthed harmonic mean

$$F_\beta = \frac{\left(1+\beta^2\right) \cdot (p \cdot r)}{\beta^2 \cdot p + r} \tag{3}$$

Source: COLES, COTTER 2009, p. 15

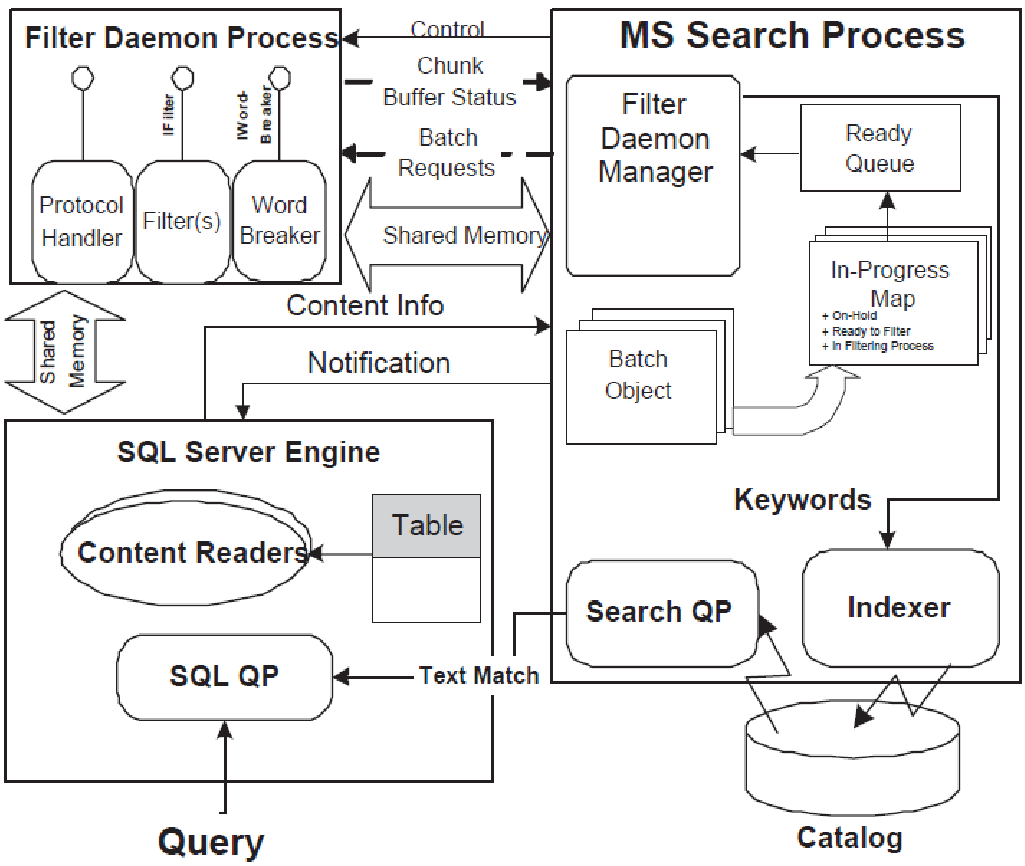
This means F_β represents the desired search quality and should be as high as possible, deciding whether to focus on recall or precision or both. (cf. COLES, COTTER 2009, pp. 13-15)

2.1.1 MS SQL Server Search Architecture

Structured Query Language (SQL) Server uses the same access method and infrastructure for full-text search as other Microsoft (MS) products and the Index Service for file

systems. This decision enables standardized semantics for full-text search of data in relational databases, web-hosted data, and data stored in the file system and mail systems. On SQL servers, not only simple strings can be indexed, but also data structures, such as Hypertext Markup Language (HTML) and Extensible Markup Language (XML), and even complex documents, such as Portable Document Format (PDF), Word, PowerPoint, Excel and other custom document formats. (cf. HAMILTON, NAYAK 2001, p. 7)
The architecture can be divided into five modules, which interact with each other to perform a full-text search. (See Figure 1)
The **content reader** scans indexed data stored in SQL Server tables to assemble data and its associated metadata packets. These packets are then injected into the main search engine, which triggers the search engine filter daemon to consume the data.
Depending on the content, the **filter daemon** calls different filters, which parse the content and output so-called chunks of the processed text. A chunk is a related section with relevant information about this section like the language-id of the text. These chunks are output separately for any properties, which can be elements like the title, an author or other content-specific elements.

Figure 1: Architecture of MS SQL Server Full-Text Search



Source: HAMILTON, NAYAK 2001, p. 8

Word breakers split the chunks into keywords and additionally provide alternative keywords and the corresponding position in the text. Word breakers can recognize human languages and on SQL Server several word breakers for different languages are installed by default. The generated keywords and metadata are passed on to the MS Search process, which processes the data with an indexer.
The **indexer** generates an inverted keyword list with a batch containing all keywords of one or more items. These indexes are compressed to use memory efficiently, this may lead to high costs for updates of these indexes. Therefore a stack of indexes is maintained. New documents first create their small indexes, which are regularly merged into a larger index, which in turn is merged into the base index. This stack can be deeper than three, but the concept remains and allows a strongly compressed index without driving the update costs too high. If a keyword is searched, all indexes are accessed, so the depth should still be kept reasonable.
A **query processor** manages the insertion and merge operations and collects statistics on distribution and frequency for ranking purposes and query execution. (cf. HAMILTON, NAYAK 2001, pp. 8-9)

2.1.2 MS SQL Server Full-Text Query Features

Full-text indexes can be created on SQL Servers with the Data Definition Language (DDL) statement `CREATE INDEX` and can make use of other SQL Server utilities; these include backup and restore and attachment of databases. There are three options to create and manage indexes on SQL Servers. **Full Crawl** always rebuilds the whole full-text index by scanning the entire table. **Incremental Crawl** logs the timestamp of the last re-index and retains changes by storing them in a column. **Change Tracking** enables a near real-time validity between the full-text index and the table by tracking changes to the indexed data using the SQL Server Query Processor. (cf. HAMILTON, NAYAK 2001, p. 9)
Full-text search is represented in SQL with three possible constructs: (cf. HAMILTON, NAYAK 2001, p. 9)

1. Contains Predicate: A contains predicate is true if one of the specified columns contains terms that satisfy the specified search condition. E.g. `Contains(author , ('Ag* or "Marc Miller"'))` will match entries where the column author contains words like 'Ag', 'Agatha', or 'Marc Miller'.

2. Freetext Predicate: Freetext predicates are true if one of the specified columns contains terms that stem from the terms in the specified search condition. E.g. `Freetext(content, 'fishing')` will match entries where content contains words like 'fishing', 'fish', or 'fisher'.

3. ContainsTable and FreetextTable: ContainsTable and FreetextTable are functions that match entries similar to their corresponding function, but additionally return multiple matches including a ranking for each entry and the entire corpus.

The search conditions of these constructs can be of various types to find the intended results: (cf. HAMILTON, NAYAK 2001, p. 9)

1. Keyword, phrase, prefix: E.g. 'fishing', 'Marc Miller', 'Ag*'
2. Inflections and Thesaurus: E.g. `Contains(*, 'FORMSOF(INFLECTIONAL, fishing)AND FORMSOF(THESAURUS, boat)')` will find all entries containing words that stem from 'fishing' and all words sharing the meaning with 'boat' (Thesaurus support).
3. Weighted terms: Keywords and phrases can be assigned a relative weight to impact the rank of entries. E.g. `ContainsTable(*, 'ISABOUT(generator weight (.7), full-text weight (.3))')` will rank entries higher in the result corpus which mention 'generator' over 'full-text'.
4. Proximity: E.g. `Contains(*, 'corn NEAR salad')` contains the proximity term 'NEAR' to match entries where 'corn' appears close to 'salad'.
5. Composition: E.g. `Contains(*, 'full-text AND NOT database')` uses two search query components that are composed using a term like 'AND', 'OR', or 'AND NOT'.

2.2 Domain-Specific Languages

Commonly known programming languages, such as C or Java, are also called a General-Purpose Language (GPL). GPLs are designed to handle any problem with relatively equal levels of efficiency and expressiveness. However, many applications do not require a multifunctional GPL and can describe a problem more naturally using a Domain-Specific Language (DSL). DSLs are languages that have been developed specifically for a particular application or domain, to be able to develop faster and more effectively. (cf. HUDAK 1997, p. 1) By tailoring notations and constructs to the domain in question, DSLs offer significant gains in expressiveness and usability compared to GPLs for the domain in question, with corresponding productivity gains and lower maintenance costs. (cf. MERNIK et al. 2005, p. 317) DSLs are by no means a product of modern software development but have existed since the beginning of programming. One of the first DSLs ever designed

was Automatically Programmed Tools (APT), which was used for the development of numerically controlled machine tools in 1957. (cf. ROSS 1978, pp. 283-284)
DSLs can be found everywhere in the world of IT, for example, this thesis was written with the help of LaTeX to design layout and formatting. Table 1 lists some well-known DSLs and their application/domain to give examples of what is classified as a DSL.

Table 1: Popular DSLs

DSL	Applicaiton
Lex and Yacc	program lexing and parsing
PERL	text/file manipulation/scripting
VDL	hardware description
TeX, LaTeX, troff	document layout
HTML, SGML	document markup
SQL, LDL, QUEL	databases
pic, postscript	2D graphics
Open GL	high-level 3D graphics
Tcl, Tk	GUI scripting
Mathematica, Maple	symbolic computation
AutoLisp/AutoCAD	computer aided design
Csh	OS scripting (Unix)
IDL	component technology (COM/CORBA)
Emacs Lisp	text editing
Prolog	logic
Visual Basic	scripting and more
Excel Macro Language	spreadsheets and many things never intended

Source: HUDAK 1997, p. 3

Programs written in a DSL are considered to be more concise, quicker to write, easier to maintain and easier to reason about and most importantly they can be written by non-programmers. In particular, experts in the domain for which the DSL was developed can use DSLs to program applications without having to acquire programming skills. An expert of a domain already knows the semantics of the domain, all that is needed to start development is the corresponding notation that expresses these semantics. (cf. HUDAK 1997, pp. 2-4)

2.3 Building a language

For a compiler or an interpreter to be able to interpret a DSL, the language must be accurately and precisely defined. Accurately means that the language must be defined consistently down to the smallest detail. Precisely means in this case that all aspects of the language must be laid out. If parts of the language are inconsistent or too vague, authors of compilers are forced to interpret these aspects themselves. This inevitably leads to different authors having different approaches to the same problem. If a DSL is to be created that meets the criteria described above, two components are needed. The first component is a set of rules, also called syntax. The second component is a formal definition of the meaning, also called semantics. (cf. FARRELL 1995, p. 2)

2.3.1 Syntax

The first step when defining syntax is defining an alphabet. This alphabet consists of tokens, which do not necessarily have to be letters. Several tokens, formulated according to a set of rules, make up a sentence or string. The alphabet of the English language is, in the context of syntax, not a list of the permissible characters, which is predominantly called the alphabet or 'ABC', but the permissible tokens. E.g. in the sentence 'the donkey screams' the tokens 'the', 'donkey' and 'screams' are part of the alphabet of the English language. The token 'gHArFk' consists of permissible characters but is not part of the valid alphabet. However, the use of permissible tokens alone does not make a sentence correct. The sentence 'on sleep blue' consists of tokens that are part of the English alphabet, but it is still not a valid sentence. The correct application of the rule set is still missing, in this example a missing object. Only the correct use of the alphabet AND the set of rules make a sentence syntactically correct. (cf. FARRELL 1995, p. 2)
If the alphabet and the set of rules are notated in a normal form, they can be called grammar. Relevant to this thesis is the Extended Backus-Naur Form (EBNF), which will be described in section 2.3.2.

2.3.2 Extended Backus-Naur Form

EBNF, as the name suggests, is based on the Backus-Naur Form, which was proposed by a group of thirteen international representatives in 1960, to serve as a basic reference and guide for building compilers. Backus-Naur Form is a notation for describing computational processes and rules as arithmetic expressions, variables, and functions. (cf. BACKUS et al. 1960, p. 300)
The syntax can be described as a set of metalinguistic formulae best described with an example. The grammar describing a number can be written in Backus-Naur Form as:

$\langle \textit{number} \rangle ::= \langle \textit{positive} \rangle | \text{-} \langle \textit{positive} \rangle | 0$
$\langle \textit{positive} \rangle ::= \langle \textit{digit not zero} \rangle \langle \textit{optional} \rangle$

⟨*optional*⟩ ::= ⟨*digit*⟩⟨*optional*⟩|
⟨*digit*⟩ ::= ⟨*digit not zero*⟩|0
⟨*digit not zero*⟩ ::= 1|2|3|4|5|6|7|8|9

Characters contained in angel brackets '<>' represent a metalinguistic variable. The character '::=' describes a definition of this variable. The character '|' represents the metalinguistic connective 'or'. Other characters in this example have no special meaning but only represent themselves. So the first line of the grammar means that the variable <number> can be defined or replaced as <positive> or -<positive> or as 0. Since the variable <positive> is mentioned in the definition, there must be a definition for this variable in the grammar, otherwise, the grammar would be incomplete. In the third line, we see a metalinguistic connective without content on its right side. This means that the variable <optional> can also be empty and thus without value. Furthermore, in this line, a variable calls itself recursively, which is allowed. (cf. BACKUS et al. 1960, pp. 301-303)
So following this grammar, numbers such as 42 or -3141592 are valid.
In 1977 Wirth proposed a new variant of the Backus-Naur Form to further improve language definition notation. The main goals of this new notation were to (cf. WIRTH 1977, p. 822)

- distinguish clearly between metaterminal and nonterminal symbols
- not exclude metaterminals as possible symbols of the language
- enable iteration without using recursion

This proposal was the basis for the ISO/IEC 14977:1996(E) which now defines the standard for EBNF. The major changes that EBNF brought can be summarized as: (cf. ISO/IEC 14977:1996(E) 1996, p. VI)

- Terminal symbols must be quoted so any symbol can be a terminal symbol of the language
- Added square brackets to indicate optional symbols and avoid the use of a <empty> symbol
- Added curly brackets to indicate repetition
- Every rule must have a final character
- Normal Brackets group items together, similar to their arithmetic use

The number example from above can be rewritten in EBNF as:

⟨*number*⟩ ::= (['-']⟨*digit not zero*⟩{⟨*digit*⟩})|'0';
⟨*digit*⟩ ::= ⟨*digit not zero*⟩|'0';
⟨*digit not zero*⟩ ::= '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';

This version of the grammar produces the same set of numbers but is more concise and arguably more readable for humans.

3 Implementation

When using the full-text search, large parts of the SQL statements needed to describe the search are the same, since the search criteria are defined as either WHERE conditions or JOIN criteria. If you want to define a full-text search, you usually use a combination of the given functions. In MSSQL this would be for example CONTAINS or FORMSOF. Therefore I want to develop a query language where you only have to specify this combination of functions and a few parameters to generate the corresponding SQL.

3.1 Language definition

The first step to defining a language is to define its purpose. In this case, there should be functions that represent full-text functions. Furthermore, one must be able to pass parameters to these functions and one should be able to combine both parameters and functions with logical operators and, or and not. To announce a function, this query language uses an '@', e.g. '@contains'. From programming languages of the C-family one recognizes the use of parentheses '()' to define parameters. To avoid later confusion with parentheses used for logical grouping, this language uses the colon ':' to enclose parameters. For now, a parameter is defined as a simple word or phrase, which is delimited with quotes '"'. These few rules already allow the definition of a query, such as `@contains:apple:` where 'contains' is the name of a function. This first set of rules can be written in EBNF as:

⟨*search*⟩ ::= '@'⟨*function*⟩':'⟨*parameter*⟩':';
⟨*function*⟩ ::= 'contains';
⟨*parameter*⟩ ::= ⟨*word*⟩|' " '{[' ']⟨*word*⟩}' " ';
⟨*word*⟩ ::= {'a'-'z'|'A'-'Z'};

Note that the function variable only includes 'contains'. In future definitions, it should accept the different functions that are going to be defined.
A feature that is also needed is the logical combination and negation of multiple search terms. For example, it should be possible to search for 'apple' or 'tree' and not 'worm'. For and the language accepts the characters '&' and '+', for or it accepts '|' and for negation it accepts '!' and '-'. To cover all possible logical operations, groups are also needed to allow precedence between the different operators. For this parentheses are used. Using groups it is now possible to build a logic like 'apple' AND NOT('tree' OR 'worm'), where the whole statement inside the parentheses is processed negated, and prioritized instead of being processed from left to right.

3.2 Lexer

The first part of a code generator is the lexer. A lexer gets a file or in this case a string as input and divides this input into a series of tokens. So the input **`@contains`**`:apple:` becomes the tokens: '@', 'contains', ':', 'apple' and ':'. These tokens are not interpreted yet but are only being recognized as separate characters. To achieve this in code the crate logos is used, to avoid writing redundant code. To understand the code written in lexer.rs what follows is a short explanation of how this crate is used in the context of this prototype. To define tokens, Logos can be added to the derive statement of an enumeration and a matching rule can be defined using a literal string or a regular expression. For example, in line 73 of code listing 1, a literal string is used to recognize the colon token, and line 33 uses a regular expression to recognize decimals between 0 and 1. It also calls an arbitrary function to_float (code 1, 17-19) to define that in this case the data should be cast into the datatype f64. Logos also requires an error type (code 1, 78-80), which is also used to skip whitespaces (cf. HIRSZ 2022, n.p.).

Code Listing 1: Token defintions

```
16 // helper function to format floats
17 fn to_float(lex: &mut Lexer<Token>) -> Option<f64> {
18     Some(lex.slice().parse().ok()?)
19 }
⋮
26 // List of all tokens that are accepted by the language
27 #[derive(Debug, Clone, Logos, PartialEq)]
28 pub enum Token {
⋮
32     // Regex: any float between 0 and 1
33     #[regex(r"0+(\.[0-9]+)?|1", to_float)]
34     ZeroToOne(f64),
⋮
72     // Colon to surround functions parameters
73     #[token(":")]
74     Colon,
75     // End of File
76     EoF,
77     // Error and skip whitespaces
78     #[error]
```

```
    #[regex(r"[\s\t\n\f]+", logos::skip)]
    Error,
}
```

Source: lexer.rs

These tokens are then compiled in a list and passed over to the parser as the work of the lexer is done.

3.3 Parser

In the parser, a large part of the heavy lifting is done, because here the list of tokens is interpreted and checked for their admissibility in the language. The parser of this custom query language stores a copy of the token list still to be parsed and additionally the current token and the next one in the list. The current token is often used to make comparisons between it and the token that would be expected, while the peek token is often used to see whether the end of the token list has already been reached. When initializing the parser both the current and the peek token are set to `Token::EoF` which represents the edge case end of file (code 2, 66-67).

Code Listing 2: Parser struct

```
54 // Parser saves current and next tokens as attribute
55 struct Parser<'p> {
56     tokens: Iter<'p, Token>,
57     current: Token,
58     peek: Token,
59 }
60
61 impl<'p> Parser<'p> {
62     // Initial parser creation
63     fn new(tokens: Iter<'p, Token>) -> Self {
64         Self {
65             tokens,
66             current: Token::EoF,
67             peek: Token::EoF,
68         }
69     }
```

Source: parser.rs

In the parsing process, two different levels are distinguished: expressions and statements. Statements are the various functions that can be used in the language, such as 'near', which is defined with several expressions as 'parameters' and another expression as a 'proximity' variable (code 3, 25-28). Expressions are the several values that can appear in a search query, for example, words, phrases, or numbers (code 3, 37-38). The special cases of operators are also represented by both statements and expressions. These will be discussed in more detail later.

Code Listing 3: Statements and expressions

```
3  #[derive(Debug, Clone, PartialEq)]
4  pub enum Statement {
⋮
25     Near {
26         parameter: Vec<Expression>,
27         proximity: Expression,
28     },
⋮
35 #[derive(Debug, Clone, PartialEq)]
36 pub enum Expression {
37     WordOrPhrase(String),
38     Number(u64),
39     ZeroToOne(f64),
```

Source: ast.rs

The tokens are normally processed linearly, always watching out not to run past the end of the file (code 4, 74). With each call of read current and peek are updated and the next statement is parsed. Read is called manually twice at the beginning to overwrite the initial `Token::EoF` (code 4, 12-14). Otherwise, the next statements are parsed until the end of the token list is reached (code 4, 16-18).

Code Listing 4: Parser read

```
7  // Main function to start parsing process
8  // Input: vec of tokens
9  // Ouput: abstract syntax tree (vec of statements)
10 pub fn parse(tokens: Vec<Token>) -> Result<Vec<Statement>,
       ParseError> {
11     let mut parser = Parser::new(tokens.iter());
12     // read twice to overwrite intial EoF tokens
```

```
13      parser.read();
14      parser.read();
15      let mut ast: Vec<Statement> = Vec::new();
16      while let Some(statement) = parser.next()? {
17          ast.push(statement);
18      }
19      Ok(ast)
20  }
⋮
71      // Parse next statement if possible
72      // Output: statement or error
73      fn next(&mut self) -> Result<Option<Statement>, ParseError>
          {
74          if self.current == Token::EoF {
75              return Ok(None);
76          }
77          Ok(Some(self.parse_statement(Precedence::Lowest)?))
78      }
79
80      // Set current and peek one step further in the vec of
           tokens
81      fn read(&mut self) {
82          self.current = self.peek.clone();
83          self.peek = if let Some(token) = self.tokens.next() {
84              token.clone()
85          } else {
86              Token::EoF
87          };
88      }
```

Source: parser.rs

When parsing a statement there must be a function token at the beginning of a statement (code 5, 120+140). If this token is found, the procedure is different depending on the function. For example, the function weighted is parsed as follows: (code 5, 325-355)
First, a colon is expected, because according to the language definition the parameters are introduced with one. As parameters, there are expected to be combinations of a search term (word or phrase) and a decimal between 0 and 1. These must be separated by commas. These tuples are expected until a colon appears as a token again. The decimals representing weights must add up to exactly 1. If none of the rules are violated, the list of

tuples is passed back to the function parse_statement and stored in the form of a statement enumeration (code 5, 140-142).

Code Listing 5: Parse weighted

```
116     // Parse statement, can only be a function or combination of functions
117     // Input: precedence
118     // Output: statement or error
119     fn parse_statement(&mut self, precedence: Precedence) -> Result<Statement, ParseError> {
120         let mut statement = match self.current.clone() {
⋮
140             Token::Weighted => Statement::Weighted {
141                 parameter: self.parse_weighted()?,
142             },
⋮
325     // Weighted function expects pairs of words or phrases and a weight between 0 and 1
326     // All weights must add up to exactly 1
327     fn parse_weighted(&mut self) -> Result<Vec<(Expression, Expression)>, ParseError> {
328         self.expect_token_and_read(Token::Weighted)?;
329         self.expect_token_and_read(Token::Colon)?;
330         let mut parameter: Vec<(Expression, Expression)> = Vec::new();
331         let mut sum_weights: f64 = 0.0;
332         while !self.current_is(Token::Colon) {
333             if self.current_is(Token::Comma) {
334                 self.expect_token_and_read(Token::Comma)?;
335             }
336             let expression = match self.parse_expression(Precedence::Lowest)? {
337                 Expression::WordOrPhrase(s) => Expression::WordOrPhrase(s),
338                 _ => return Err(ParseError::UnexpectedToken(self.current.clone())),
339             };
340             self.expect_token_and_read(Token::Comma)?;
```

```
            let weight = match self.parse_expression(Precedence::Lowest)? {
                Expression::ZeroToOne(f) => {
                    sum_weights += f;
                    Expression::ZeroToOne(f)
                }
                _ => return Err(ParseError::UnexpectedToken(self.current.clone())),
            };
            parameter.push((expression, weight));
        }
        if sum_weights != 1.0 {
            return Err(ParseError::WeightError(sum_weights));
        }
        self.expect_token_and_read(Token::Colon)?;
        Ok(parameter)
    }
```

Source: parser.rs

In the parse_statement function, two different functions are called to process tokens. On the one hand, expect_token_and_read (code 6, 110-114) compares the current token with an input variable and reads past it without further logic. This function is mostly used for parsing syntactic tokens, such as the colon, which themselves have no impact on the content of the search.

Code Listing 6: expect_token_and_read

```
90      // See what the current token is
91      // Output: boolean
92      fn current_is(&self, token: Token) -> bool {
93          std::mem::discriminant(&self.current) == std::mem::discriminant(&token)
94      }
95
96      // Current token should match the one given
97      // Input: token
98      // Output: token or error
99      fn expect_token(&mut self, token: Token) -> Result<Token, ParseError> {
100         if self.current_is(token) {
101             Ok(self.current.clone())
```

```
        } else {
            Err(ParseError::UnexpectedToken(self.current.clone()))
        }
    }

    // Current token should match the one given and read to next token
    // Input: token
    // Output: token or error
    fn expect_token_and_read(&mut self, token: Token) -> Result<Token, ParseError> {
        let result = self.expect_token(token)?;
        self.read();
        Ok(result)
    }
```

Source: parser.rs

The second function is parse_expression, which is similar in logic to parse_statement. Here the current token is compared to the possible expressions and returned as an expression enumeration. For example, with the WordOrPhrase token, the content is stored in the variable s and passed when the expression counterpart is generated (code 7, 159-161).

Code Listing 7: Parse WordOrPhrase

```
156     // Parse expression, could be a search term, number, operator or combination of epxressions
157     fn parse_expression(&mut self, precedence: Precedence) -> Result<Expression, ParseError> {
158         let mut expr = match self.current.clone() {
159             Token::WordOrPhrase(s) => {
160                 self.expect_token_and_read(Token::WordOrPhrase("".to_string()))?;
161                 Expression::WordOrPhrase(s.to_string())
162             }
```

Source: parser.rs

With these building blocks, it is already possible to parse a token list, like `Near, Colon, WordOrPhrase("apple"), Comma, WordOrPhrase("tree"), Comma, Number(9), Colon` into the statement

`Near{parameter: (WordOrPhrase("apple"), WordOrPhrase("tree")), proximity: Number(9)}.`

While parsing, attention has been paid to the syntax of the language and the information has been reduced to the minimum necessary in a structured way.

3.4 Parsing operators and groups

In addition to simple search terms and the call of a single function, the query language should also offer the possibility to logically link search terms and use several functions simultaneously. To make this possible, operators such as AND, OR, and NOT, and groups come into play. To interpret these types of operators and groups and store them as part of the abstract syntax tree (AST), there are separate types for them as both statement and expression. The statement enumeration (code 8, 8-12) is used to allow the use of multiple functions in the query language, while the expression enumerations (code 8, 40-41) are used to logically link search terms. The operators themselves are stored as a separate enumeration, with a function to translate tokens into operators (code 8, 52-59).

Code Listing 8: Operator statements and expressions

```
3  #[derive(Debug, Clone, PartialEq)]
4  pub enum Statement {
5      Group {
6          expression: Expression,
7      },
8      Infix {
9          statement: Box<Statement>,
10         operator: Operator,
11         second_statement: Box<Statement>,
12     },
⋮
33 }
34
35 #[derive(Debug, Clone, PartialEq)]
36 pub enum Expression {
⋮
40     Infix(Box<Expression>, Operator, Box<Expression>),
41     Prefix(Operator, Box<Expression>),
```

```
}

#[derive(Debug, Clone, PartialEq)]
pub enum Operator {
    And,
    Or,
    Not,
}

impl Operator {
    pub fn token(token: Token) -> Self {
        match token {
            Token::And | Token::Plus => Self::And,
            Token::Or => Self::Or,
            Token::Minus | Token::Bang => Self::Not,
            _ => unreachable!("{:?}", token),
        }
    }
}
```

Source: ast.rs

The big challenge with operators and groups is that it is no longer sufficient to process the token list linearly because operators are partly written after the affected tokens and hierarchies exist between the operators. For example, the AND operator has a stronger binding power than the OR operator, and groups, or parentheses, have an even higher binding power. This binding power is in text and code further called precedence.
Precedence is implemented as an ordered enumeration, which allows them to be compared and assigned higher or lower precedence. As with operators, there is a function to translate tokens into precedence (code 9, 37-51). The concept of precedence already appears in the code listings 5 and 7 as an input variable for the functions parse_statement and parse_expression.

Code Listing 9: Precedence

```
22 // Precedence to enable priorities between operators
23 // Example: this OR that AND some (AND should have a higher
      priority)
24 #[derive(Debug, Clone, PartialEq, PartialOrd)]
25 enum Precedence {
26     Lowest,
27     Statement,
```

```
    Or,
    And,
    Not,
    Prefix,
    Group,
}

// Match tokens to precedences
impl Precedence {
    fn token(token: Token) -> Self {
        match token {
            Token::Bang | Token::Minus => Self::Not,
            Token::Plus | Token::And | Token::WordOrPhrase(..) => Self::And,
            Token::Or => Self::Or,
            Token::LeftParen => Self::Group,
            Token::Contains
            | Token::Starts
            | Token::Inflection
            | Token::Thesaurus
            | Token::Near
            | Token::Weighted => Self::Statement,
            _ => Self::Lowest,
        }
    }
}
```

Source: parser.rs

Expression operators that are written before the token in question, such as the NOT operator, can be processed similarly to normal expressions. When the parser encounters one of the NOT tokens in parse_expression (code 10, 171-177), an `Expression::Prefix` is returned, where the actual search term is parsed with the parameter `Precedence::Prefix`, which is higher than the default `Precedence::Lowest`.

Code Listing 10: Parse NOT

```
171             t @ Token::Minus | t @ Token::Bang => {
172                 self.expect_token_and_read(t.clone())?;
173                 Expression::Prefix(
174                     Operator::token(t),
```

```
                            Box::new(self.parse_expression(Precedence::Prefix)?),
                        )
                    }
```

Source: parser.rs

More complicated are operators which are written after an affected token. For this, in parse_expression after a token was parsed an attempt is made to parse a postfix or infix operator. At the same time, the precedence of the next token is compared to keep the corresponding hierarchies of the operators (code 11, 189-197). The expression parsed so far is passed to the parse_infix_expression function and it structures it into an `Expression::Infix`, where the next search term is parsed again with the corresponding precedence.

Code Listing 11: Parse infix operator

```
188         // Afer an expression could be an infix operator or directly a new expression (here called postfix operator)
189         while !self.current_is(Token::EoF) && precedence < Precedence::token(self.current.clone()) {
190             if let Some(expression) = self.parse_postfix_expression(expr.clone())? {
191                 expr = expression;
192             } else if let Some(expression) = self.parse_infix_expression(expr.clone())? {
193                 expr = expression
194             } else {
195                 break;
196             }
197         }
```

⋮

```
220     // Infix operators AND and OR expect an expression on either side
221     fn parse_infix_expression(
222         &mut self,
223         expr: Expression,
224     ) -> Result<Option<Expression>, ParseError> {
225         Ok(match self.current {
226             Token::Plus | Token::And | Token::Or => {
227                 let token = self.current.clone();
```

```
                    self.read();
                    let sec_expr = self.parse_expression(Precedence::token(token.clone()))?;
                    Some(Expression::Infix(
                        Box::new(expr),
                        Operator::token(token),
                        Box::new(sec_expr),
                    ))
                }
                _ => None,
            })
        }
```

Source: parser.rs

Postfix operators as such do not exist in the query language; instead, a search term written without an infix operator in between is parsed as a postfix AND (code 12, 203-218). So 'apple tree' is parsed as 'apple AND tree'. The second search term could potentially be negated, so the case 'apple -tree' is also covered and parsed as 'apple AND NOT tree'.

Code Listing 12: Parse postfix operator

```
201     // Postfix operator is called when two expressions are read, automatically inserting an AND inbetween
202     // Second Expression could have an NOT operator before the actual expression
203     fn parse_postfix_expression(
204         &mut self,
205         expr: Expression,
206     ) -> Result<Option<Expression>, ParseError> {
207         Ok(match self.current {
208             Token::Minus | Token::Bang | Token::WordOrPhrase(..) => {
209                 let sec_expr = self.parse_expression(Precedence::And)?;
210                 Some(Expression::Infix(
211                     Box::new(expr),
212                     Operator::And,
213                     Box::new(sec_expr),
214                 ))
215             }
216             _ => None,
```

```
            })
        }
```

Source: parser.rs

Groups are delimited by parentheses and contain the highest precedence of all operators. Inside a group, an expression is expected (code 13, 360), which is then returned as soon as a right parenthesis is read. It should be emphasized that groups are not expressions, but only handle the included expression with higher precedence, so the inner expression itself is returned instead of some kind of group expression (code 13, 184).

Code Listing 13: Parse groups

```
178                 // Start a group which gets higher precedence
179                 Token::LeftParen => {
180                     let group_expression = match self.parse_group()? {
181                         Statement::Group { expression } => expression,
182                         _ => return Err(ParseError::Unreachable),
183                     };
184                     group_expression
185                 }
⋮
357     // Groups must encapsulate an expression with parentheses and have higher precedence then other operators
358     fn parse_group(&mut self) -> Result<Statement, ParseError> {
359         self.expect_token_and_read(Token::LeftParen)?;
360         let expression = self.parse_expression(Precedence::Statement)?;
361         self.expect_token_and_read(Token::RightParen)?;
362         Ok(Statement::Group { expression })
363     }
```

Source: parser.rs

Infix operators were also implemented at the statement level to allow multiple functions to be used in a single query. The logic is the same as that of code listing 11, except that statements are processed instead of expressions.

Code Listing 14: Parse infix operator in statements

```rust
        // After a function could be an infix operator
        while !self.current_is(Token::EoF) && precedence < Precedence::token(self.current.clone()) {
            if let Some(in_statement) = self.parse_infix_statement(statement.clone())? {
                statement = in_statement
            } else {
                break;
            }
        }
```

⋮

```rust
    // Infix operators AND and OR expect a statement on either side
    fn parse_infix_statement(
        &mut self,
        statement: Statement,
    ) -> Result<Option<Statement>, ParseError> {
        Ok(match self.current {
            Token::Plus | Token::And | Token::Or => {
                let token = self.current.clone();
                self.read();
                let second_statement = self.parse_statement(Precedence::token(token.clone()))?;
                Some(Statement::Infix {
                    statement: Box::new(statement),
                    operator: Operator::token(token),
                    second_statement: Box::new(second_statement),
                })
            }
            _ => None,
        })
    }
```

Source: parser.rs

These are all the building blocks needed to parse the entirety of the query language. The result of the parser is a list of statements or AST. This is now passed to the generator to generate SQL code from the logical sequence of operators and search terms.

3.5 Generator

The generator is the last step of the code generator because it does not convert the query language into bytecode or similar, but into another language, in this case, SQL. For this purpose, the generator is similarly structured to the parser, except that it receives an AST as input instead of output and generates a string from it.

In Transact-SQL there are two ways to initiate a full-text search. The use of predicates to specify criteria for the search in the WHERE clause of a query and the use of a CONTAINSTABLE or FREETEXTTABLE, which contains matching results and can be joined to the table to be searched. This generator uses a CONTAINSTABLE to represent all the functions offered by the query language in SQL. This method is preferred to predicates because it allows getting a list of results, sorted by how much they match the search criteria, also called a rank. In addition, CONTAINSTABLE is preferred to FREETEXTTABLE, because FREETEXTTABLE is suitable for more fuzzy searches and less for searches that use defining functions to narrow down the results.

Based on these findings, a part of the SQL command can be preformulated with a few constants containing the metadata of the database (code 15, 7-10). So an INNER JOIN of the CONTAINSTABLE and the table to be searched is made, where the rank must be greater than five to weed out inaccurate matches (code 15, 29). From this, the title and rank are selected and sorted by rank in descending order. The top five results are then finally returned (code 15, 22). The exact specifications of the CONTAINSTABLE, which represent the search criteria, is the part that is filled by the generator (code 15, 26-28).

Code Listing 15: Generate sql_parts

```
6  // Database constants
7  const DB_NAME: &str = "Wikipedia";
8  const TBL_NAME: &str = "[dbo].[Real_Article]";
9  const RETURN_ATTRIBUTE: &str = "Title";
10 const TOP_ROWS: u64 = 5;
```

⋮

```
20     let mut sql_parts: Vec<String> = Vec::new();
21     sql_parts.push(format!(
22         "USE {}; SELECT TOP {} * FROM(SELECT FT_TBL.{}, KEY_TBL
              .RANK FROM {} AS FT_TBL INNER JOIN CONTAINSTABLE({},
               *, '",
23         DB_NAME, TOP_ROWS, RETURN_ATTRIBUTE, TBL_NAME, TBL_NAME
24     ));
25     // generate all functions as JOIN constraints
```

```
    while let Some(sql_part) = generator.next()? {
        sql_parts.push(sql_part);
    }
    sql_parts.push("') AS KEY_TBL ON FT_TBL.[ID] = KEY_TBL.[KEY] WHERE KEY_TBL.RANK > 5) AS FS_RESULT ORDER BY FS_RESULT.RANK DESC;".to_owned());
```

Source: generator.rs

Similar to the parser, the generator stores the list of statements and additionally the current and next statements as attributes, where the peek statement is used to check the end of the list. On initialization current and peek are set to `Statement::EoF` (code 16, 45-46).

Code Listing 16: Generator struct

```
33 // Generator struct with current and next statements as attributes
34 struct Generator<'p> {
35     statements: Iter<'p, Statement>,
36     current: Statement,
37     peek: Statement,
38 }
39
40 impl<'p> Generator<'p> {
41     // Initial generator creation
42     fn new(statements: Iter<'p, Statement>) -> Self {
43         Self {
44             statements,
45             current: Statement::EoF,
46             peek: Statement::EoF,
47         }
48     }
```

Source: generator.rs

The logic to iterate through the list is similar to that of the parser with the use of a next function which checks if the end of the list has already been reached (code 17, 52) and generates the next element if not and a write function which updates the current and peek attributes (code 17, 60-65).

Code Listing 17: Generator write

```
50     // Generate next statement if possible
51     fn next(&mut self) -> Result<Option<String>, GenerateError> {
```

```
        if self.current == Statement::EoF {
            return Ok(None);
        }
        Ok(Some(self.generate_statement(self.current.clone())?))
    }

    // Set current and peek one step further in the ast
    fn write(&mut self) {
        self.current = self.peek.clone();
        self.peek = if let Some(statement) = self.statements.next() {
            statement.clone()
        } else {
            Statement::EoF
        };
    }
```

Source: generator.rs

In the code, three functions are used to generate all elements of an AST. Similar to the parser, a distinction is made between statements and expressions and additionally operators.

The generate_statement function is passed a statement to be generated (code 18, 71). Usually, this statement is one of the implemented functions of the query language, for example, weighted. In the case of the weighted function, the stored attribute 'parameter' is used to translate the statement into SQL as follows. First, the SQL function ISABOUT is called and a parenthesis is opened (code 18, 136). Then, for each tuple of search terms and their weights, the values are written down so that, for example, the values 'apple' and 0.4 give `apple WEIGHT (0.4),` (code 18, 137-142). Finally, the last comma is deleted, the parenthesis of ISABOUT is closed, and all parts are combined into one string and returned(code 18, 143-145+150).

Code Listing 18: Generate weighted

```
68      // Generate statement, always a function or combination of functions
69      // Input: statement to generate
70      // Output: string
71      fn generate_statement(&mut self, statement: Statement) -> Result<String, GenerateError> {
72          let sql: String = match statement {
```

```
⋮
            // Weighted generates tuples of search criteria and their respective weight
            Statement::Weighted { parameter } => {
                let mut sql_parts: Vec<String> = Vec::new();
                sql_parts.push(format!("ISABOUT("));
                for (word_or_phrase_expr, weight_expr) in parameter {
                    let word_or_phrase = self.generate_expression(word_or_phrase_expr)?;
                    let weight = self.generate_expression(weight_expr)?;
                    sql_parts.push(format!("{} WEIGHT({})", word_or_phrase, weight));
                    sql_parts.push(String::from(", "));
                }
                sql_parts.remove(sql_parts.len() - 1);
                sql_parts.push(String::from(")"));
                sql_parts.join("")
            }
⋮
        };
        self.write();
        Ok(sql)
    }
```

Source: generator.rs

There is a special case in generate_statement, infix operators, this case calls generate_statement for the two statements and generate_operator for the operator (code 19, 79-81) and combines the results.

Code Listing 19: Generate infix operator statements

```
73              Statement::Infix {
74                  statement,
75                  operator,
76                  second_statement,
77              } => {
78                  let sql_parts = [
```

```
                    self.generate_statement(*statement)?,
                    self.generate_operator(operator)?,
                    self.generate_statement(*second_statement)
                        ?,
                ];
                sql_parts.join(" ")
            }
```

Source: generator.rs

In the example above for generating weighted, the generate_expression function was called in code listing 18 in lines 138 and 139 to generate the search terms and weights. This function takes as input the expression it should generate and returns the value of the expression as a string if a normal expression was passed (code 20, 158-160).

Code Listing 20: Generate expressions

```
153     // Generate expression, any search criteria or number or
            combination of those
154     // Input: expression to generate
155     // Output: string
156     fn generate_expression(&mut self, expression: Expression)
            -> Result<String, GenerateError> {
157         let sql: String = match expression {
158             Expression::WordOrPhrase(s) => s,
159             Expression::Number(u) => u.to_string(),
160             Expression::ZeroToOne(f) => f.to_string(),
⋮
186         };
187         Ok(sql)
188     }
```

Source: generator.rs

4 Summary

Summary

Appendix

Appendix 1: main.rs

```
use actix_web::{web, App, HttpResponse, HttpServer, Responder};
use regex::Regex;
use serde::{Deserialize, Serialize};
use std::fs::{read_to_string, File};
use std::io::{Error, ErrorKind, Write};
use std::process::Command;
use tera::{Context, Tera};

mod code_gen;

// Path Variables
const PATH_SQL: &str = "files\\fulltext.sql";
const PATH_RESULTS: &str = "files\\results.txt";

// Main function to start website on localhost:8080
// Run using 'cargo watch -x run'
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        let tera = Tera::new("templates/**/*").unwrap();
        App::new()
            .data(tera)
            .route("/", web::get().to(search))
            .route("/", web::post().to(result))
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}

// Code generator to translate an input to SQL
// Input: search string and path to write result to
// Output: SQL statement written to a file
fn run_code_gen(search: String, path: &str) -> std::io::Result
    <()> {
```

```
    // Transform string to list of tokens
    let tokens = code_gen::lexer::lex(search.as_str());
    // Parse tokens to an abstract syntax tree (ast)
    let ast = code_gen::parser::parse(tokens);
    match ast {
        // If parser returns no error, start code generation
        Ok(ast) => {
            let generator = code_gen::generator::generate(ast);
            // If generator returns no error, write SQL statement to file, otherwise throw an error
            match generator {
                Ok(generator) => write!(File::create(path)?, "{}", generator),
                Err(gen_err) => Err(Error::new(ErrorKind::InvalidData, format!("{:?}", gen_err))),
            }
        }
        // If parser returns error, throw an error aswell
        Err(parse_err) => Err(Error::new(
            ErrorKind::InvalidInput,
            format!("{:?}", parse_err),
        )),
    }
}

// Runs a command to execute an sql statement to a local MSSQL Server
// Input: paths to the input file and where to write the result
// Output: txt file interpretation of the MSSQL Server result
fn execute_sql(sql_path: &str, results_path: &str) {
    Command::new("cmd")
        .args(&[
            "/C",
            "sqlcmd",
            "-S",
            "DESKTOP-JKNEH40\\SQLEXPRESS", //Local server name
            "-i",
            sql_path,
            "-o",
            results_path,
```

```
        ])
        .output()
        .expect("failed to execute operation");
}

// Reads the txt file result and extracts the actual results
// Input: path to the txt file
// Output: vec of titles and their search rank
fn read_results(path: &str) -> Option<Vec<(String, u64)>> {
    let contents = read_to_string(path).unwrap();
    let mut contents_vec: Vec<&str> = contents.split("\n").collect();
    // In case of error message, break
    if contents_vec.len() < 6 {
        return None;
    }
    // Remove metadata rows
    // First 3-4 rows and last three rows
    while !contents_vec[0].starts_with("---") {
        contents_vec.remove(0);
    }
    contents_vec.remove(0);
    contents_vec.remove(contents_vec.len() - 1);
    contents_vec.remove(contents_vec.len() - 1);
    contents_vec.remove(contents_vec.len() - 1);
    // Go through each row and extract the titles and their ranks
    let mut results: Vec<(String, u64)> = Vec::new();
    for row in contents_vec {
        // Remove unnecessary whitespaces
        let row = row.replace("\r", "");
        let re = Regex::new(r"\s+").unwrap();
        let row = re.replace_all(&row, " ").to_string();
        // Extract last 'word' as rank and save the rest as the title
        let mut words: Vec<&str> = row.split(" ").collect();
        let rank = words[words.len() - 1].parse::<u64>().unwrap();
        words.remove(words.len() - 1);
        let title = words.join(" ");
```

```rust

            results.push((title, rank));
        }
        Some(results)
}

// Search and Result structs to (de)serialize rust and website
    datatypes
#[derive(Deserialize)]
struct Search {
    search: String,
}
#[derive(Serialize)]
struct Result {
    title: String,
    rank: u64,
    link: String,
}

// Define functional parts of the search page
async fn search(tera: web::Data<Tera>) -> impl Responder {
    let mut data = Context::new();
    data.insert("title", "Search field");
    let rendered = tera.render("search.html", &data).unwrap();
    HttpResponse::Ok().body(rendered)
}

// Define functional parts of the result page
async fn result(tera: web::Data<Tera>, data: web::Form<Search>)
    -> impl Responder {
    let mut page_data = Context::new();
    let mut results: Vec<Result> = Vec::new();
    // Run code generator with the string from the search field
    match run_code_gen(data.search.clone(), PATH_SQL) {
        // If code generator returns no error execute SQL and
            read the results
        Ok(_) => {
            execute_sql(PATH_SQL, PATH_RESULTS);
            let results_vec = read_results(PATH_RESULTS);
```

```
            // Fit search results into Result struct to properly display on the page, otherwise diplay error
            match results_vec {
                Some(results_vec) => {
                    for result in results_vec {
                        results.push(Result {
                            title: result.0.clone(),
                            rank: result.1,
                            // link to the Wikipedia article is also provided, whitespaces need to be replaced
                            link: result.0.replace(" ", "_"),
                        })
                    }
                    page_data.insert("title", "Results");
                    page_data.insert("search", &data.search);
                }
                None => {
                    page_data.insert("title", "Error");
                    page_data.insert(
                        "search",
                        &format!("{} results cannot be read", &data.search),
                    );
                }
            }
        }
        // If code generator returns error, display error instead of search results
        Err(error) => {
            page_data.insert("title", "Error");
            page_data.insert(
                "search",
                &format!("{} threw an error: {}", &data.search, &error.to_string()),
            );
        }
    }
    page_data.insert("results", &results);
```

```
    let rendered = tera.render("result.html", &page_data).
       unwrap();
    HttpResponse::Ok().body(rendered)
}
```

Appendix 2: lexer.rs

```
use logos::{Lexer, Logos};

// Main function to start lexing process
// Input: string
// Output: vec of tokens
pub fn lex(input: &str) -> Vec<Token> {
    Token::lexer(input).collect()
}

// helper function to format strings
fn to_string(lex: &mut Lexer<Token>) -> Option<String> {
    let string = lex.slice().to_string();
    Some(string)
}

// helper function to format floats
fn to_float(lex: &mut Lexer<Token>) -> Option<f64> {
    Some(lex.slice().parse().ok()?)
}

// helper function to format unsigned integer
fn to_u64(lex: &mut Lexer<Token>) -> Option<u64> {
    Some(lex.slice().parse().ok()?)
}

// List of all tokens that are accepted by the language
#[derive(Debug, Clone, Logos, PartialEq)]
pub enum Token {
    // Regex: phrase starting and ending with " and escaped
       character \" or just a word allowing a list of special
       characters
    #[regex(r##""(?:[^"\\]|\\.)*"|[a-zA-Zß?üÜöÖäÄ;\._<>´'#§$
       %/\\=€]+"##, to_string)]
```

```
    WordOrPhrase(String),
    // Regex: any float between 0 and 1
    #[regex(r"0+(\.[0-9]+)?|1", to_float)]
    ZeroToOne(f64),
    // Regex: any postive integer
    #[regex(r"[0-9]+", to_u64)]
    Number(u64),
    // ! and - for NOT
    #[token("!")]
    Bang,
    #[token("-")]
    Minus,
    // & and + for AND
    #[token("&")]
    And,
    #[token("+")]
    Plus,
    // | for OR
    #[token("|")]
    Or,
    // Parentheses for grouping
    #[token("(")]
    LeftParen,
    #[token(")")]
    RightParen,
    // Comma for parameter separation
    #[token(",")]
    Comma,
    // Functions
    #[token("@contains")]
    Contains,
    #[token("@startswith")]
    Starts,
    #[token("@inflection")]
    Inflection,
    #[token("@thesaurus")]
    Thesaurus,
    #[token("@near")]
    Near,
    #[token("@weighted")]
```

```
    Weighted,
    // Colon to surround functions parameters
    #[token(":")]
    Colon,
    // End of File
    EoF,
    // Error and skip whitespaces
    #[error]
    #[regex(r"[\s\t\n\f]+", logos::skip)]
    Error,
}

// Enable tokens to be casted as strings
impl Into<String> for Token {
    fn into(self) -> String {
        match self {
            Token::WordOrPhrase(s) => s,
            _ => unreachable!(),
        }
    }
}
```

Appendix 3: parser.rs

```
use std::slice::Iter;
use thiserror::Error;

use crate::code_gen::ast::*;
use crate::code_gen::lexer::Token;

// Main function to start parsing process
// Input: vec of tokens
// Ouput: abstract syntax tree (vec of statements)
pub fn parse(tokens: Vec<Token>) -> Result<Vec<Statement>,
   ParseError> {
    let mut parser = Parser::new(tokens.iter());
    // read twice to overwrite intial EoF tokens
    parser.read();
    parser.read();
```

```
    let mut ast: Vec<Statement> = Vec::new();
    while let Some(statement) = parser.next()? {
        ast.push(statement);
    }
    Ok(ast)
}

// Precedence to enable priorities between operators
// Example: this OR that AND some (AND should have a higher
   priority)
#[derive(Debug, Clone, PartialEq, PartialOrd)]
enum Precedence {
    Lowest,
    Statement,
    Or,
    And,
    Not,
    Prefix,
    Group,
}

// Match tokens to precedences
impl Precedence {
    fn token(token: Token) -> Self {
        match token {
            Token::Bang | Token::Minus => Self::Not,
            Token::Plus | Token::And | Token::WordOrPhrase(..)
               => Self::And,
            Token::Or => Self::Or,
            Token::LeftParen => Self::Group,
            Token::Contains
            | Token::Starts
            | Token::Inflection
            | Token::Thesaurus
            | Token::Near
            | Token::Weighted => Self::Statement,
            _ => Self::Lowest,
        }
    }
}
```

```
// Parser saves current and next tokens as attribute
struct Parser<'p> {
    tokens: Iter<'p, Token>,
    current: Token,
    peek: Token,
}

impl<'p> Parser<'p> {
    // Initial parser creation
    fn new(tokens: Iter<'p, Token>) -> Self {
        Self {
            tokens,
            current: Token::EoF,
            peek: Token::EoF,
        }
    }

    // Parse next statement if possible
    // Output: statement or error
    fn next(&mut self) -> Result<Option<Statement>, ParseError> {
        if self.current == Token::EoF {
            return Ok(None);
        }
        Ok(Some(self.parse_statement(Precedence::Lowest)?))
    }

    // Set current and peek one step further in the vec of
       tokens
    fn read(&mut self) {
        self.current = self.peek.clone();
        self.peek = if let Some(token) = self.tokens.next() {
            token.clone()
        } else {
            Token::EoF
        };
    }

    // See what the current token is
```

```
    // Output: boolean
    fn current_is(&self, token: Token) -> bool {
        std::mem::discriminant(&self.current) == std::mem::discriminant(&token)
    }

    // Current token should match the one given
    // Input: token
    // Output: token or error
    fn expect_token(&mut self, token: Token) -> Result<Token, ParseError> {
        if self.current_is(token) {
            Ok(self.current.clone())
        } else {
            Err(ParseError::UnexpectedToken(self.current.clone()))
        }
    }

    // Current token should match the one given and read to next token
    // Input: token
    // Output: token or error
    fn expect_token_and_read(&mut self, token: Token) -> Result<Token, ParseError> {
        let result = self.expect_token(token)?;
        self.read();
        Ok(result)
    }

    // Parse statement, can only be a function or combination of functions
    // Input: precedence
    // Output: statement or error
    fn parse_statement(&mut self, precedence: Precedence) -> Result<Statement, ParseError> {
        let mut statement = match self.current.clone() {
            Token::Contains => Statement::Contains {
                expression: self.parse_contains()?,
            },
```

```
            Token::Starts => Statement::Starts {
                expression: self.parse_starts()?,
            },
            Token::Inflection => Statement::Inflection {
                expression: self.parse_inflection()?,
            },
            Token::Thesaurus => Statement::Thesaurus {
                expression: self.parse_thesaurus()?,
            },
            Token::Near => {
                let (parameter, proximity) = self.parse_near()?;
                Statement::Near {
                    parameter,
                    proximity,
                }
            }
            Token::Weighted => Statement::Weighted {
                parameter: self.parse_weighted()?,
            },
            _ => return Err(ParseError::UnexpectedToken(self.current.clone())),
        };
        // After a function could be an infix operator
        while !self.current_is(Token::EoF) && precedence < Precedence::token(self.current.clone()) {
            if let Some(in_statement) = self.parse_infix_statement(statement.clone())? {
                statement = in_statement
            } else {
                break;
            }
        }
        Ok(statement)
    }

    // Parse expression, could be a search term, number, operator or combination of epxressions
    fn parse_expression(&mut self, precedence: Precedence) -> Result<Expression, ParseError> {
```

```
        let mut expr = match self.current.clone() {
            Token::WordOrPhrase(s) => {
                self.expect_token_and_read(Token::WordOrPhrase("".to_string()))?;
                Expression::WordOrPhrase(s.to_string())
            }
            Token::Number(u) => {
                self.expect_token_and_read(Token::Number(0))?;
                Expression::Number(u)
            }
            Token::ZeroToOne(f) => {
                self.expect_token_and_read(Token::ZeroToOne(0.0))?;
                Expression::ZeroToOne(f)
            }
            t @ Token::Minus | t @ Token::Bang => {
                self.expect_token_and_read(t.clone())?;
                Expression::Prefix(
                    Operator::token(t),
                    Box::new(self.parse_expression(Precedence::Prefix)?),
                )
            }
            // Start a group which gets higher precedence
            Token::LeftParen => {
                let group_expression = match self.parse_group()? {
                    Statement::Group { expression } => expression,
                    _ => return Err(ParseError::Unreachable),
                };
                group_expression
            }
            _ => return Err(ParseError::UnexpectedToken(self.current.clone())),
        };
        // Afer an expression could be an infix operator or directly a new expression (here called postfix operator)
```

```
        while !self.current_is(Token::EoF) && precedence < Precedence::token(self.current.clone()) {
            if let Some(expression) = self.parse_postfix_expression(expr.clone())? {
                expr = expression;
            } else if let Some(expression) = self.parse_infix_expression(expr.clone())? {
                expr = expression
            } else {
                break;
            }
        }
        Ok(expr)
    }

    // Postfix operator is called when two expressions are read, automatically inserting an AND inbetween
    // Second Expression could have an NOT operator before the actual expression
    fn parse_postfix_expression(
        &mut self,
        expr: Expression,
    ) -> Result<Option<Expression>, ParseError> {
        Ok(match self.current {
            Token::Minus | Token::Bang | Token::WordOrPhrase(..) => {
                let sec_expr = self.parse_expression(Precedence::And)?;
                Some(Expression::Infix(
                    Box::new(expr),
                    Operator::And,
                    Box::new(sec_expr),
                ))
            }
            _ => None,
        })
    }

    // Infix operators AND and OR expect an expression on either side
```

```
    fn parse_infix_expression(
        &mut self,
        expr: Expression,
    ) -> Result<Option<Expression>, ParseError> {
        Ok(match self.current {
            Token::Plus | Token::And | Token::Or => {
                let token = self.current.clone();
                self.read();
                let sec_expr = self.parse_expression(Precedence
                    ::token(token.clone()))?;
                Some(Expression::Infix(
                    Box::new(expr),
                    Operator::token(token),
                    Box::new(sec_expr),
                ))
            }
            _ => None,
        })
    }

    // Infix operators AND and OR expect a statement on either
        side
    fn parse_infix_statement(
        &mut self,
        statement: Statement,
    ) -> Result<Option<Statement>, ParseError> {
        Ok(match self.current {
            Token::Plus | Token::And | Token::Or => {
                let token = self.current.clone();
                self.read();
                let second_statement = self.parse_statement(
                    Precedence::token(token.clone()))?;
                Some(Statement::Infix {
                    statement: Box::new(statement),
                    operator: Operator::token(token),
                    second_statement: Box::new(second_statement
                        ),
                })
            }
            _ => None,
```

```
        })
    }

    // Functions all have a similar strucure needing colons to surround their parameters

    // Contains function only expects one word or phrase or combination of expressions
    fn parse_contains(&mut self) -> Result<Expression, ParseError> {
        self.expect_token_and_read(Token::Contains)?;
        self.expect_token_and_read(Token::Colon)?;
        let expression: Expression = self.parse_expression(Precedence::Statement)?;
        self.expect_token_and_read(Token::Colon)?;
        Ok(expression)
    }

    // Startswith function only expects one one word or phrase or combination of expressions
    fn parse_starts(&mut self) -> Result<Expression, ParseError> {
        self.expect_token_and_read(Token::Starts)?;
        self.expect_token_and_read(Token::Colon)?;
        let expression: Expression = self.parse_expression(Precedence::Statement)?;
        self.expect_token_and_read(Token::Colon)?;
        Ok(expression)
    }

    // Inflection function only expects one one word or phrase or combination of expressions
    fn parse_inflection(&mut self) -> Result<Expression, ParseError> {
        self.expect_token_and_read(Token::Inflection)?;
        self.expect_token_and_read(Token::Colon)?;
        let expression: Expression = self.parse_expression(Precedence::Statement)?;
        self.expect_token_and_read(Token::Colon)?;
        Ok(expression)
```

```
    }

    // Thesaurus function only expects one one word or phrase or combination of expressions
    fn parse_thesaurus(&mut self) -> Result<Expression, ParseError> {
        self.expect_token_and_read(Token::Thesaurus)?;
        self.expect_token_and_read(Token::Colon)?;
        let expression: Expression = self.parse_expression(Precedence::Statement)?;
        self.expect_token_and_read(Token::Colon)?;
        Ok(expression)
    }

    // Near function expects multiple comma-seperated words or phrases with an optional number as the last parameter
    fn parse_near(&mut self) -> Result<(Vec<Expression>, Expression), ParseError> {
        self.expect_token_and_read(Token::Near)?;
        self.expect_token_and_read(Token::Colon)?;
        let mut parameter: Vec<Expression> = Vec::new();
        // Proximity has a default value of 5 if no number is given
        let mut proximity = Expression::Number(5);
        while !self.current_is(Token::Colon) {
            if self.current_is(Token::Comma) {
                self.expect_token_and_read(Token::Comma)?;
            }
            match self.parse_expression(Precedence::Lowest)? {
                Expression::WordOrPhrase(s) => parameter.push(Expression::WordOrPhrase(s)),
                Expression::Number(u) => {
                    if self.current_is(Token::Colon) {
                        proximity = Expression::Number(u)
                    } else {
                        return Err(ParseError::UnexpectedToken(self.current.clone()));
                    }
                }
```

```
                _ => return Err(ParseError::UnexpectedToken(self.current.clone())),
            }
        }
        self.expect_token_and_read(Token::Colon)?;
        Ok((parameter, proximity))
    }

    // Weighted function expects pairs of words or phrases and a weight between 0 and 1
    // All weights must add up to exactly 1
    fn parse_weighted(&mut self) -> Result<Vec<(Expression, Expression)>, ParseError> {
        self.expect_token_and_read(Token::Weighted)?;
        self.expect_token_and_read(Token::Colon)?;
        let mut parameter: Vec<(Expression, Expression)> = Vec::new();
        let mut sum_weights: f64 = 0.0;
        while !self.current_is(Token::Colon) {
            if self.current_is(Token::Comma) {
                self.expect_token_and_read(Token::Comma)?;
            }
            let expression = match self.parse_expression(Precedence::Lowest)? {
                Expression::WordOrPhrase(s) => Expression::WordOrPhrase(s),
                _ => return Err(ParseError::UnexpectedToken(self.current.clone())),
            };
            self.expect_token_and_read(Token::Comma)?;
            let weight = match self.parse_expression(Precedence::Lowest)? {
                Expression::ZeroToOne(f) => {
                    sum_weights += f;
                    Expression::ZeroToOne(f)
                }
                _ => return Err(ParseError::UnexpectedToken(self.current.clone())),
            };
            parameter.push((expression, weight));
```

```
        }
        if sum_weights != 1.0 {
            return Err(ParseError::WeightError(sum_weights));
        }
        self.expect_token_and_read(Token::Colon)?;
        Ok(parameter)
    }

    // Groups must encapsulate an expression with parentheses and have higher precedence then other operators
    fn parse_group(&mut self) -> Result<Statement, ParseError> {
        self.expect_token_and_read(Token::LeftParen)?;
        let expression = self.parse_expression(Precedence::Statement)?;
        self.expect_token_and_read(Token::RightParen)?;
        Ok(Statement::Group { expression })
    }
}

// Types of errors covered by the parser
#[derive(Debug, Error)]
pub enum ParseError {
    #[error("Unexpected token {0:?}.")]
    UnexpectedToken(Token),
    #[error("Entered unreachable code.")]
    Unreachable,
    #[error("Weights do not add up to 1.0. Sum of all weights: {0}")]
    WeightError(f64),
}
```

Appendix 4: ast.rs

```
use crate::code_gen::lexer::Token;

#[derive(Debug, Clone, PartialEq)]
pub enum Statement {
    Group {
```

```
        expression: Expression,
    },
    Infix {
        statement: Box<Statement>,
        operator: Operator,
        second_statement: Box<Statement>,
    },
    Contains {
        expression: Expression,
    },
    Starts {
        expression: Expression,
    },
    Inflection {
        expression: Expression,
    },
    Thesaurus {
        expression: Expression,
    },
    Near {
        parameter: Vec<Expression>,
        proximity: Expression,
    },
    Weighted {
        parameter: Vec<(Expression, Expression)>,
    },
    EoF,
}

#[derive(Debug, Clone, PartialEq)]
pub enum Expression {
    WordOrPhrase(String),
    Number(u64),
    ZeroToOne(f64),
    Infix(Box<Expression>, Operator, Box<Expression>),
    Prefix(Operator, Box<Expression>),
}

#[derive(Debug, Clone, PartialEq)]
pub enum Operator {
```

```
    And,
    Or,
    Not,
}

impl Operator {
    pub fn token(token: Token) -> Self {
        match token {
            Token::And | Token::Plus => Self::And,
            Token::Or => Self::Or,
            Token::Minus | Token::Bang => Self::Not,
            _ => unreachable!("{:?}", token),
        }
    }
}
```

Appendix 5: generator.rs

```
use std::slice::Iter;
use thiserror::Error;

use crate::code_gen::ast::{Expression, Operator, Statement};

// Database constants
const DB_NAME: &str = "Wikipedia";
const TBL_NAME: &str = "[dbo].[Real_Article]";
const RETURN_ATTRIBUTE: &str = "Title";
const TOP_ROWS: u64 = 5;

// Main function to start the generation process
// Input: vec of statements (ast)
// Output: string (sql statement)
pub fn generate(ast: Vec<Statement>) -> Result<String,
   GenerateError> {
    let mut generator = Generator::new(ast.iter());
    // write twice to overwrite initial EoF statements
    generator.write();
    generator.write();
    let mut sql_parts: Vec<String> = Vec::new();
```

```
    sql_parts.push(format!(
        "USE {}; SELECT TOP {} * FROM(SELECT FT_TBL.{}, KEY_TBL
           .RANK FROM {} AS FT_TBL INNER JOIN CONTAINSTABLE({},
            *, '",
        DB_NAME, TOP_ROWS, RETURN_ATTRIBUTE, TBL_NAME, TBL_NAME
    ));
    // generate all functions as JOIN constraints
    while let Some(sql_part) = generator.next()? {
        sql_parts.push(sql_part);
    }
    sql_parts.push("') AS KEY_TBL ON FT_TBL.[ID] = KEY_TBL.[KEY
       ] WHERE KEY_TBL.RANK > 5) AS FS_RESULT ORDER BY
       FS_RESULT.RANK DESC;".to_owned());
    Ok(sql_parts.join(" "))
}

// Generator struct with current and next statements as
   attributes
struct Generator<'p> {
    statements: Iter<'p, Statement>,
    current: Statement,
    peek: Statement,
}

impl<'p> Generator<'p> {
    // Initial generator creation
    fn new(statements: Iter<'p, Statement>) -> Self {
        Self {
            statements,
            current: Statement::EoF,
            peek: Statement::EoF,
        }
    }

    // Generate next statement if possible
    fn next(&mut self) -> Result<Option<String>, GenerateError>
        {
        if self.current == Statement::EoF {
            return Ok(None);
        }
```

```
        Ok(Some(self.generate_statement(self.current.clone())?))
    }

    // Set current and peek one step further in the ast
    fn write(&mut self) {
        self.current = self.peek.clone();
        self.peek = if let Some(statement) = self.statements.next() {
            statement.clone()
        } else {
            Statement::EoF
        };
    }

    // Generate statement, always a function or combination of functions
    // Input: statement to generate
    // Output: string
    fn generate_statement(&mut self, statement: Statement) -> Result<String, GenerateError> {
        let sql: String = match statement {
            Statement::Infix {
                statement,
                operator,
                second_statement,
            } => {
                let sql_parts = [
                    self.generate_statement(*statement)?,
                    self.generate_operator(operator)?,
                    self.generate_statement(*second_statement)?,
                ];
                sql_parts.join(" ")
            }
            // Contains generates it's search condition without mutation
            Statement::Contains { expression } => {
                format!("{}", self.generate_expression(expression)?)
```

```
            }
            // Startswith adds a * to end of a word or before the last " in a phrase
            Statement::Starts { expression } => {
                let mut word_or_phrase = self.generate_expression(expression)?;
                if word_or_phrase.starts_with('"') && word_or_phrase.ends_with('"') {
                    word_or_phrase.insert(word_or_phrase.len() - 1, '*');
                } else {
                    word_or_phrase.push('*');
                }
                format!("{}", word_or_phrase)
            }
            // Inflection calls the inflection function from MSSQL
            Statement::Inflection { expression } => {
                let mut word_or_phrase = self.generate_expression(expression)?;
                if word_or_phrase.starts_with('"') && word_or_phrase.ends_with('"') {
                    word_or_phrase.remove(0);
                    word_or_phrase.remove(word_or_phrase.len() - 1);
                }
                format!("FORMSOF(INFLECTIONAL,\"{}\")", word_or_phrase)
            }
            // Thesaurus calls the thesaurus function from MSSQL
            Statement::Thesaurus { expression } => {
                let mut word_or_phrase = self.generate_expression(expression)?;
                if word_or_phrase.starts_with('"') && word_or_phrase.ends_with('"') {
                    word_or_phrase.remove(0);
                    word_or_phrase.remove(word_or_phrase.len() - 1);
                }
```

```
                format!("FORMSOF(THESAURUS,\"{}\")",
                    word_or_phrase)
            }
            // Near generates a parameter list of all search
                criteria and proximity in the end
            Statement::Near {
                parameter,
                proximity,
            } => {
                let mut sql_parts: Vec<String> = Vec::new();
                sql_parts.push(format!("NEAR(("));
                for expression in parameter {
                    let string = self.generate_expression(
                        expression)?;
                    sql_parts.push(format!("{}", string));
                    sql_parts.push(String::from(", "));
                }
                sql_parts.remove(sql_parts.len() - 1);
                sql_parts.push(format!("), {})", self.
                    generate_expression(proximity)?));
                sql_parts.join("")
            }
            // Weighted generates tuples of search criteria and
                their respective weight
            Statement::Weighted { parameter } => {
                let mut sql_parts: Vec<String> = Vec::new();
                sql_parts.push(format!("ISABOUT("));
                for (word_or_phrase_expr, weight_expr) in
                    parameter {
                    let word_or_phrase = self.
                        generate_expression(word_or_phrase_expr)
                        ?;
                    let weight = self.generate_expression(
                        weight_expr)?;
                    sql_parts.push(format!("{} WEIGHT({})",
                        word_or_phrase, weight));
                    sql_parts.push(String::from(", "));
                }
                sql_parts.remove(sql_parts.len() - 1);
                sql_parts.push(String::from(")"));
```

```
                sql_parts.join("")
            }
            _ => return Err(GenerateError::UnexpectedStatement(self.current.clone())),
        };
        self.write();
        Ok(sql)
    }

    // Generate expression, any search criteria or number or combination of those
    // Input: expression to generate
    // Output: string
    fn generate_expression(&mut self, expression: Expression) -> Result<String, GenerateError> {
        let sql: String = match expression {
            Expression::WordOrPhrase(s) => s,
            Expression::Number(u) => u.to_string(),
            Expression::ZeroToOne(f) => f.to_string(),
            // Infix operator enclose their expressions with parentheses to ensure precedence
            Expression::Infix(expr1, operator, expr2) => {
                let mut sql_parts = [
                    String::from("("),
                    self.generate_expression(*expr1)?,
                    String::from(")"),
                    self.generate_operator(operator)?,
                    String::from("("),
                    self.generate_expression(*expr2.clone())?,
                    String::from(")"),
                ];
                // If the second expression is a not operator it must write NOT before the parentheses
                match *expr2 {
                    Expression::Prefix(Operator::Not, ..) => sql_parts[4] = String::from("NOT ("),
                    _ => (),
                }
                sql_parts.join(" ")
            }
```

```
            Expression::Prefix(operator, expr) => {
                let sql_parts = [
                    self.generate_operator(operator)?,
                    self.generate_expression(*expr)?,
                ];
                sql_parts.join(" ")
            }
        };
        Ok(sql)
    }

    // Generate operator
    // Input: operator to generate
    // Output: string
    fn generate_operator(&mut self, operator: Operator) -> Result<String, GenerateError> {
        let op = match operator {
            Operator::And => "AND",
            Operator::Or => "OR",
            // has to be set infront of parentheses, see generate_expression for infix
            Operator::Not => "",
        };
        Ok(op.to_owned())
    }
}

// Types of error covered by the generator
#[derive(Debug, Error)]
pub enum GenerateError {
    #[error("Unexpected statement {0:?}.")]
    UnexpectedStatement(Statement),
}
```

Appendix 6: mod.rs

```
mod ast;
pub mod generator;
pub mod lexer;
pub mod parser;
```

Appendix 7: base.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>{{title}}</title>
    </head>
    <body>
        {% block content %}
        {% endblock %}
    </body>
</html>
```

Appendix 8: search.html

```
{% extends "base.html" %}

{% block content %}
<form action="" method="POST">
    <div>
        <label for="search">Search:</label>
        <input type="text" name="search">
    </div>
    <input type="submit" value="Submit">
</form>
{% endblock %}
```

Appendix 9: result.html

```
{% extends "base.html" %}

{% block content %}
<div>
    <p>{{ search }}</p>
</div>
{% for result in results %}
<div>
```

```
    <a href="https://en.wikipedia.org/wiki/{{ result.link }}">
       {{ result.title }}</a>
    <small>{{ result.rank }}</small>
</div>
{% endfor %}
{% endblock %}
```

Appendix 10: convert_wiki_to_csv.py

```
from lxml import etree
from glob import glob
import bz2
import codecs
import csv
import time
import os

PATH_WIKI_XML = "/Users/light/Documents/Wikipedia/"
FILENAME_ARTICLES = "articles.csv"


def hms_string(sec_elapsed):
    h = int(sec_elapsed / (60 * 60))
    m = int((sec_elapsed % (60 * 60)) / 60)
    s = sec_elapsed % 60
    return "{}:{:>02}:{:>05.2f}".format(h, m, s)


def get_parser(filename):
    ns_token = "{http://www.mediawiki.org/xml/export-0.10/}ns"
    title_token = "{http://www.mediawiki.org/xml/export-0.10/}
       title"
    revision_token = "{http://www.mediawiki.org/xml/export
       -0.10/}revision"
    text_token = "{http://www.mediawiki.org/xml/export-0.10/}
       text"

    with bz2.BZ2File(filename, "rb") as bz2_file:
        for event, element in etree.iterparse(bz2_file, events
           =("end",)):
```

```
            if element.tag.endswith("page"):
                namespace_tag = element.find(ns_token)

                if namespace_tag.text == "0":
                    title_tag = element.find(title_token)
                    text_tag = element.find(revision_token).find(text_token)
                    yield title_tag.text, text_tag.text

                element.clear()


def pluck_wikipedia_titles_text(
    out_file, pattern="enwiki-*-pages-articles-multistream*.xml-*.bz2"
):
    totalCount = 0
    longestTitle = 0
    longestText = 0
    with codecs.open(out_file, "a+b", "utf8") as out_file:
        writer = csv.writer(out_file)
        for bz2_filename in sorted(
            glob(pattern),
            key=lambda a: int(a.split("articles-multistream")[1].split(".")[0]),
        ):
            print(bz2_filename)
            parser = get_parser(bz2_filename)
            for title, text in parser:
                if not (text.startswith("#REDIRECT") or text.startswith("#redirect")):
                    totalCount += 1
                    writer.writerow([title, text])
                    longestTitle = (
                        len(title) if len(title) > longestTitle else longestTitle
                    )
                    longestText = len(text) if len(text) > longestText else longestText
```

```
                    if totalCount > 1 and (totalCount % 100000)
                        == 0:
                        print("{:,}".format(totalCount))
                    if totalCount >= 1000:
                        break
    print(
        f"Total Count: {totalCount}\nLongest Title: {
           longestTitle}\nLongest Text: {longestText}"
    )


pathArticles = os.path.join(PATH_WIKI_XML, FILENAME_ARTICLES)

start_time = time.time()

pluck_wikipedia_titles_text(pathArticles)

time_took = time.time() - start_time
print(f"Total runtime: {hms_string(time_took)}")
```

Appendix 11: create_article.sql

```sql
USE Wikipedia;
DROP TABLE Real_Article;
CREATE TABLE [dbo].[Real_Article](
    [Title] [varchar](255) NULL,
    [Text] [varchar](max) NULL
);
INSERT INTO Real_Article(Title, Text)
VALUES ('Example', 'Fulltext');
SELECT *
FROM Real_Article;
DELETE FROM Real_Article
WHERE Title = 'Example';
```

Appendix 12: bulk_insert.sql

```sql
USE Wikipedia;
BULK
```

```
INSERT Real_Article
FROM 'C:\Users\light\Documents\Wikipedia\enwiki-20220901-pages-
    articles-multistream-no-redirect.csv' WITH (
            FIRSTROW = 1,
            FIELDQUOTE = '\'
        , FIELDTERMINATOR = ',
            '
        , ROWTERMINATOR = ' \ n ',
            TABLOCK);
```

Appendix 13: alter_id.sql

```
USE Wikipedia;
ALTER TABLE [dbo].[Real_Article]
ADD ID INT IDENTITY(1, 1) PRIMARY KEY
```

Bibliography

BACKUS, J. W. et al.: Report on the algorithmic language ALGOL 60. en. In: *Communications of the ACM* 3 (May 1960) Nr. 5. Ed. by NAUR, Peter, pp. 299–314. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/367236.367262. URL: https://dl.acm.org/doi/10.1145/367236.367262 (visited on 08/16/2022)

COLES, Michael; COTTER, Hilary: *Pro full-text search in SQL Server 2008*. The expert's voice in SQL server. Berkeley, CA: Apress, 2009. ISBN: 978-1-4302-1594-3

FARRELL, James Alan: Compiler Basics. en. In: (Aug. 1995), p. 7. URL: http://www.cs.man.ac.uk/~pjj/farrell/compmain.html (visited on 08/16/2022)

HAMILTON, James R.; NAYAK, Tapas K.: Microsoft SQL server full-text search. In: *IEEE Data Eng. Bull.* 24 (2001) Nr. 4. Publisher: Citeseer, pp. 7–10

HIRSZ, Maciej: *Logos: Create ridiculously fast Lexers*. June 2022. URL: https://github.com/maciejhirsz/logos (visited on 10/05/2022)

HUDAK, Paul: Domain-specific languages. In: *Handbook of programming languages* 3 (1997) Nr. 39-60, p. 21

ISO/IEC 14977:1996(E): Information Technology - Syntactic Metalanguage - Extended BNF. In: (1996). URL: https://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf (visited on 08/16/2022)

MERNIK, Marjan; HEERING, Jan; SLOANE, Anthony M.: When and how to develop domain-specific languages. en. In: *ACM Computing Surveys* 37 (Dec. 2005) Nr. 4, pp. 316–344. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/1118890.1118892. URL: https://dl.acm.org/doi/10.1145/1118890.1118892 (visited on 08/15/2022)

ROSS, Douglas T.: Origins of the APT language for automatically programmed tools. enin: WEXELBLAT, Richard L. (ed.): *History of programming languages*. New York, NY, USA: ACM, June 1978, pp. 279–338. ISBN: 978-0-12-745040-7. DOI: 10.1145/800025.1198374. URL: http://dl.acm.org/doi/10.1145/800025.1198374 (visited on 08/15/2022)

TENOPIR, Carol; RO, Jung Soon: *Full text databases*. New directions in information management no. 21. New York: Greenwood Press, 1990. ISBN: 978-0-313-26303-3

WIRTH, Niklaus: What can we do about the unnecessary diversity of notation for syntactic definitions?. en. In: *Communications of the ACM* 20 (Nov. 1977) Nr. 11, pp. 822–823. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/359863.359883. URL: https://dl.acm.org/doi/10.1145/359863.359883 (visited on 08/16/2022)

Declaration in lieu of oath

I hereby declare that I produced the submitted paper with no assistance from any other party and without the use of any unauthorized aids and, in particular, that I have marked as quotations all passages which are reproduced verbatim or near-verbatim from publications. Also, I declare that the submitted print version of this thesis is identical with its digital version. Further, I declare that this thesis has never been submitted before to any examination board in either its present form or in any other similar version. I herewith agree that this thesis may be published. I herewith consent that this thesis may be uploaded to the server of external contractors for the purpose of submitting it to the contractors' plagiarism detection systems. Uploading this thesis for the purpose of submitting it to plagiarism detection systems is not a form of publication.

Bonn, 12.10.2022

(Location, Date)

(handwritten signature)