



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 6, 2022

Contents

Index of Figures	III
Index of Tables	IV
Index of Abbreviations	V
Index of Symbols	VI
Index of Formulae	VII
Index of Code Listings	VIII
1 Abstract	1
2 Theory	2
2.1 Full-Text Search	2
2.1.1 MS SQL Server Search Architecture	3
2.1.2 MS SQL Server Full-Text Query Features	5
2.2 Domain-Specific Languages	6
2.3 Building a language	7
2.3.1 Syntax	8
2.3.2 Extended Backus-Naur Form	8
3 Implementation	11
3.1 Language definition	11
3.2 Lexer	12
3.3 Parser	13
4 Summary	18
Appendix	19
Bibliography	51

Index of Figures

Figure 1: Architecture of MS SQL Server Full-Text Search	4
--	---

Index of Tables

Table 1: Popular DSLs	7
---------------------------------	---

Index of Abbreviations

APT	Automatically Programmed Tools
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

Formula 1: Precision	3
Formula 2: Recall	3
Formula 3: Weighed harmonic mean	3

Index of Code Listings

Code Listing 1: Token definitions	12
Code Listing 2: Parser struct	13
Code Listing 3: Statements and expressions	14
Code Listing 4: Parser read	14
Code Listing 5: Parse weighted	16

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

$$p = \frac{n}{d} \quad (1)$$

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

$$r = \frac{n}{v} \quad (2)$$

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: Weighed harmonic mean

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

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:

$$\begin{aligned}\langle number \rangle &::= \langle positive \rangle | -\langle positive \rangle | 0 \\ \langle positive \rangle &::= \langle digit \text{ not zero} \rangle \langle optional \rangle\end{aligned}$$

$\langle optional \rangle ::= \langle digit \rangle \langle optional \rangle |$

$\langle digit \rangle ::= \langle digit not zero \rangle | 0$

$\langle digit not zero \rangle ::= 1|2|3|4|5|6|7|8|9$

Characters contained in angel brackets ' $\langle \rangle$ ' 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 $\langle number \rangle$ can be defined or replaced as $\langle positive \rangle$ or $-\langle positive \rangle$ or as 0. Since the variable $\langle positive \rangle$ 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 $\langle optional \rangle$ 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 $\langle empty \rangle$ 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:

$\langle number \rangle ::= (['-'] \langle digit not zero \rangle \{ \langle digit \rangle \}) ['0'] ;$

$\langle digit \rangle ::= \langle digit not zero \rangle ['0'] ;$

$\langle digit not zero \rangle ::= '1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' ;$

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 definitions

```

16 // helper function to format floats
17 fn to_float(lex: &mut Lexer<Token>) -> Option<f64> {
18     Some(lex.slice().parse().ok()?)
19 }
20
21 :
22
26 // List of all tokens that are accepted by the language
27 #[derive(Debug, Clone, Logos, PartialEq)]
28 pub enum Token {
29
30     :
31
32     // Regex: any float between 0 and 1
33     #[regex(r"0+(\.[0-9]+)?|1", to_float)]
34     ZeroToOne(f64),
35
36     :
37
72     // Colon to surround functions parameters
73     #[token(":")]
74     Colon,
75     // End of File
76     EoF,
77     // Error and skip whitespaces
78     #[error]

```

```

79     #[regex(r"[\s\t\n\f]+", logos::skip)]
80     Error,
81 }

```

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, 61-62).

Code Listing 2: Parser struct

```

49 // Parser saves current and next tokens as attribute
50 struct Parser<'p> {
51     tokens: Iter<'p, Token>,
52     current: Token,
53     peek: Token,
54 }
55
56 impl<'p> Parser<'p> {
57     // Initial parser creation
58     fn new(tokens: Iter<'p, Token>) -> Self {
59         Self {
60             tokens,
61             current: Token::EoF,
62             peek: Token::EoF,
63         }
64     }

```

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 {
5
6      :
7
8      :
9
10     :
11
12     :
13
14     :
15
16     :
17
18     :
19
20     :
21
22     :
23
24     :
25     Near {
26         parameter: Vec<Expression>,
27         proximity: Expression,
28     },
29
30     :
31     :
32     :
33     :
34     :
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, 69). 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  // Output: abstract syntax tree (vec of statements)
10 pub fn parse(tokens: Vec<Token>) -> Result<Vec<Statement>,
11     ParseError> {
12     let mut parser = Parser::new(tokens.iter());
13     // read twice to overwrite initial 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 }

:

66     // Parse next statement if possible
67     // Output: statement or error
68     fn next(&mut self) -> Result<Option<Statement>, ParseError>
        {
69         if self.current == Token::EoF {
70             return Ok(None);
71         }
72         Ok(Some(self.parse_statement(Precedence::Lowest)?))
73     }

74
75     // Set current and peek one step further in the vec of
       tokens
76     fn read(&mut self) {
77         self.current = self.peek.clone();
78         self.peek = if let Some(token) = self.tokens.next() {
79             token.clone()
80         } else {
81             Token::EoF
82         };
83     }

```

Source: parser.rs

When parsing a statement there must be a function token at the beginning of a statement (code 5, 115+135). If this token is found, the procedure is different depending on the function. For example, the function `weighted` is parsed as follows: (code 5, 320-350)

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, 135-137).

Code Listing 5: Parse weighted

```

111 // Parse statement, can only be a function or combination
    of functions
112 // Input: precedence
113 // Output: statement or error
114 fn parse_statement(&mut self, precedence: Precedence) ->
    Result<Statement, ParseError> {
115     let mut statement = match self.current.clone() {
    :
    :
135         Token::Weighted => Statement::Weighted {
136             parameter: self.parse_weighted()?,
137         },
    :
    :
320 // Weighted function expects pairs of words or phrases and
    a weight between 0 and 1
321 // All weights must add up to exactly 1
322 fn parse_weighted(&mut self) -> Result<Vec<(Expression,
    Expression)>, ParseError> {
323     self.expect_token_and_read(Token::Weighted)?;
324     self.expect_token_and_read(Token::Colon)?;
325     let mut parameter: Vec<(Expression, Expression)> = Vec
        ::new();
326     let mut sum_weights: f64 = 0.0;
327     while !self.current_is(Token::Colon) {
328         if self.current_is(Token::Comma) {
329             self.expect_token_and_read(Token::Comma)?;
330         }
331         let expression = match self.parse_expression(
            Precedence::Lowest)? {
332             Expression::WordOrPhrase(s) => Expression::
                WordOrPhrase(s),
333             _ => return Err(ParseError::UnexpectedToken(
                self.current.clone())),
334         };
335         self.expect_token_and_read(Token::Comma)?;

```



```
336         let weight = match self.parse_expression(Precedence
337             ::Lowest)? {
338             Expression::ZeroToOne(f) => {
339                 sum_weights += f;
340                 Expression::ZeroToOne(f)
341             }
342             _ => return Err(ParseError::UnexpectedToken(
343                 self.current.clone())),
344         };
345         parameter.push((expression, weight));
346     }
347     if sum_weights != 1.0 {
348         return Err(ParseError::WeightError(sum_weights));
349     }
350     self.expect_token_and_read(Token::Colon)?;
351     Ok(parameter)
352 }
```

Source: parser.rs

4 Summary

Summary

Appendix

Appendix 1: main.rs

```

1 use actix_web::{web, App, HttpResponse, HttpServer, Responder};
2 use regex::Regex;
3 use serde::{Deserialize, Serialize};
4 use std::fs::{read_to_string, File};
5 use std::io::{Error, ErrorKind, Write};
6 use std::process::Command;
7 use tera::{Context, Tera};
8
9 mod code_gen;
10
11 // Path Variables
12 const PATH_SQL: &str = "files\\fulltext.sql";
13 const PATH_RESULTS: &str = "files\\results.txt";
14
15 // Main function to start website on localhost:8080
16 // Run using 'cargo watch -x run'
17 #[actix_web::main]
18 async fn main() -> std::io::Result<()> {
19     HttpServer::new(|| {
20         let tera = Tera::new("templates/**/*.html").unwrap();
21         App::new()
22             .data(tera)
23             .route("/", web::get().to(search))
24             .route("/", web::post().to(result))
25     })
26     .bind("127.0.0.1:8080")?
27     .run()
28     .await
29 }
30
31 // Code generator to translate an input to SQL
32 // Input: search string and path to write result to
33 // Output: SQL statement written to a file
34 fn run_code_gen(search: String, path: &str) -> std::io::Result
    <()> {

```

```

35 // Transform string to list of tokens
36 let tokens = code_gen::lexer::lex(search.as_str());
37 // Parse tokens to an abstract syntax tree (ast)
38 let ast = code_gen::parser::parse(tokens);
39 match ast {
40     // If parser returns no error, start code generation
41     Ok(ast) => {
42         let generator = code_gen::generator::generate(ast);
43         // If generator returns no error, write SQL
44         // statement to file, otherwise throw an error
45         match generator {
46             Ok(generator) => write!(File::create(path)?, "
47                 {}", generator),
48             Err(gen_err) => Err(Error::new(ErrorKind::
49                 InvalidData, format!("{:?}", gen_err))),
50         }
51     }
52     // If parser returns error, throw an error aswell
53     Err(parse_err) => Err(Error::new(
54         ErrorKind::InvalidInput,
55         format!("{:?}", parse_err),
56     )),
57 }
58
59 // Runs a command to execute an sql statement to a local MSSQL
60 // Server
61 // Input: paths to the input file and where to write the result
62 // Output: txt file interpretation of the MSSQL Server result
63 fn execute_sql(sql_path: &str, results_path: &str) {
64     Command::new("cmd")
65         .args(&[
66             "/C",
67             "sqlcmd",
68             "-S",
69             "DESKTOP-JKNEH40\\SQLEXPRESS", //Local server name
70             "-i",
71             sql_path,
72             "-o",
73             results_path,

```

```

71         ])
72         .output()
73         .expect("failed to execute operation");
74     }
75
76     // Reads the txt file result and extracts the actual results
77     // Input: path to the txt file
78     // Output: vec of titles and their search rank
79     fn read_results(path: &str) -> Option<Vec<(String, u64)>> {
80         let contents = read_to_string(path).unwrap();
81         let mut contents_vec: Vec<&str> = contents.split("\n").
            collect();
82         // In case of error message, break
83         if contents_vec.len() < 6 {
84             return None;
85         }
86         // Remove metadata rows
87         // First 3-4 rows and last three rows
88         while !contents_vec[0].starts_with("---") {
89             contents_vec.remove(0);
90         }
91         contents_vec.remove(0);
92         contents_vec.remove(contents_vec.len() - 1);
93         contents_vec.remove(contents_vec.len() - 1);
94         contents_vec.remove(contents_vec.len() - 1);
95         // Go through each row and extract the titles and their
            ranks
96         let mut results: Vec<(String, u64)> = Vec::new();
97         for row in contents_vec {
98             // Remove unnecessary whitespaces
99             let row = row.replace("\r", "");
100             let re = Regex::new(r"\s+").unwrap();
101             let row = re.replace_all(&row, " ").to_string();
102             // Extract last 'word' as rank and save the rest as the
                title
103             let mut words: Vec<&str> = row.split(" ").collect();
104             let rank = words[words.len() - 1].parse::<u64>().unwrap
                ();
105             words.remove(words.len() - 1);
106             let title = words.join(" ");

```

```

107
108         results.push((title, rank));
109     }
110     Some(results)
111 }
112
113 // Search and Result structs to (de)serialize rust and website
114 // datatypes
115 #[derive(Deserialize)]
116 struct Search {
117     search: String,
118 }
119 #[derive(Serialize)]
120 struct Result {
121     title: String,
122     rank: u64,
123     link: String,
124 }
125
126 // Define functional parts of the search page
127 async fn search(tera: web::Data<Tera>) -> impl Responder {
128     let mut data = Context::new();
129     data.insert("title", "Search field");
130     let rendered = tera.render("search.html", &data).unwrap();
131     HttpResponse::Ok().body(rendered)
132 }
133
134 // Define functional parts of the result page
135 async fn result(tera: web::Data<Tera>, data: web::Form<Search>)
136     -> impl Responder {
137     let mut page_data = Context::new();
138     let mut results: Vec<Result> = Vec::new();
139     // Run code generator with the string from the search field
140     match run_code_gen(data.search.clone(), PATH_SQL) {
141         // If code generator returns no error execute SQL and
142         // read the results
143         Ok(_) => {
144             execute_sql(PATH_SQL, PATH_RESULTS);
145             let results_vec = read_results(PATH_RESULTS);

```

```

143     // Fit search results into Result struct to
144     // properly display on the page, otherwise display
145     // error
146 match results_vec {
147     Some(results_vec) => {
148         for result in results_vec {
149             results.push(Result {
150                 title: result.0.clone(),
151                 rank: result.1,
152                 // link to the Wikipedia article is
153                 // also provided, whitespaces need
154                 // to be replaced
155                 link: result.0.replace(" ", "_"),
156             })
157         }
158         page_data.insert("title", "Results");
159         page_data.insert("search", &data.search);
160     }
161     None => {
162         page_data.insert("title", "Error");
163         page_data.insert(
164             "search",
165             &format!("{}", results cannot be read", &
166                 data.search),
167         );
168     }
169 }
170 // If code generator returns error, display error
171 // instead of search results
172 Err(error) => {
173     page_data.insert("title", "Error");
174     page_data.insert(
175         "search",
176         &format!("{}", threw an error: {}", &data.search,
177             &error.to_string()),
178     );
179 }
180 }
181 page_data.insert("results", &results);

```

```

176     let rendered = tera.render("result.html", &page_data).
        unwrap();
177     HttpResponse::Ok().body(rendered)
178 }

```

Appendix 2: lexer.rs

```

1 use logos::{Lexer, Logos};
2
3 // Main function to start lexing process
4 // Input: string
5 // Output: vec of tokens
6 pub fn lex(input: &str) -> Vec<Token> {
7     Token::lexer(input).collect()
8 }
9
10 // helper function to format strings
11 fn to_string(lex: &mut Lexer<Token>) -> Option<String> {
12     let string = lex.slice().to_string();
13     Some(string)
14 }
15
16 // helper function to format floats
17 fn to_float(lex: &mut Lexer<Token>) -> Option<f64> {
18     Some(lex.slice().parse().ok())
19 }
20
21 // helper function to format unsigned integer
22 fn to_u64(lex: &mut Lexer<Token>) -> Option<u64> {
23     Some(lex.slice().parse().ok())
24 }
25
26 // List of all tokens that are accepted by the language
27 #[derive(Debug, Clone, Logos, PartialEq)]
28 pub enum Token {
29     // Regex: phrase starting and ending with " and escaped
        character \" or just a word allowing a list of special
        characters
30     #[regex(r#"\"(?:[^\\"\\\\]|\\.)*\"| [a-zA-Zß?üÜöÖäÄ; \_<>'`#$%
        %/\ \=€]+\"##, to_string)]

```



```
31 WordOrPhrase(String),
32 // Regex: any float between 0 and 1
33 #[regex(r"0+(\.[0-9]+)?|1", to_float)]
34 ZeroToOne(f64),
35 // Regex: any postive integer
36 #[regex(r"[0-9]+", to_u64)]
37 Number(u64),
38 // ! and - for NOT
39 #[token("!")]
40 Bang,
41 #[token("-")]
42 Minus,
43 // & and + for AND
44 #[token("&")]
45 And,
46 #[token("+")]
47 Plus,
48 // | for OR
49 #[token("|")]
50 Or,
51 // Parentheses for grouping
52 #[token("(")]
53 LeftParen,
54 #[token(")")]
55 RightParen,
56 // Comma for parameter separation
57 #[token(",")]
58 Comma,
59 // Functions
60 #[token("@contains")]
61 Contains,
62 #[token("@startswith")]
63 Starts,
64 #[token("@inflection")]
65 Inflection,
66 #[token("@thesaurus")]
67 Thesaurus,
68 #[token("@near")]
69 Near,
70 #[token("@weighted")]
```

```

71     Weighted,
72     // Colon to surround functions parameters
73     #[token(":")]
74     Colon,
75     // End of File
76     EoF,
77     // Error and skip whitespaces
78     #[error]
79     #[regex(r"[\s\t\n\f]+", logos::skip)]
80     Error,
81 }
82
83 // Enable tokens to be casted as strings
84 impl Into<String> for Token {
85     fn into(self) -> String {
86         match self {
87             Token::WordOrPhrase(s) => s,
88             _ => unreachable!(),
89         }
90     }
91 }

```

Appendix 3: parser.rs

```

1 use std::slice::Iter;
2 use thiserror::Error;
3
4 use crate::code_gen::ast::*;
5 use crate::code_gen::lexer::Token;
6
7 // Main function to start parsing process
8 // Input: vec of tokens
9 // Output: 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 initial 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 }
21
22 // Precedence to enable priorities between expressions
23 // Example: this OR that AND some (AND should have a higher
24 // priority)
25 #[derive(Debug, Clone, PartialEq, PartialOrd)]
26 enum Precedence {
27     Lowest,
28     Statement,
29     Or,
30     And,
31     Not,
32     Prefix,
33     Group,
34 }
35
36 // Match tokens to precedences
37 impl Precedence {
38     fn token(token: Token) -> Self {
39         match token {
40             Token::Bang | Token::Minus => Self::Not,
41             Token::Plus | Token::And | Token::WordOrPhrase(..)
42                 => Self::And,
43             Token::Or => Self::Or,
44             Token::LeftParen => Self::Group,
45             Token::Contains | Token::Starts | Token::Inflection
46                 => Self::Statement,
47             _ => Self::Lowest,
48         }
49     }
50 }
51
52 // Parser saves current and next tokens as attribute
53 struct Parser<'p> {
54     tokens: Iter<'p, Token>,

```

```

52     current: Token,
53     peek: Token,
54 }
55
56 impl<'p> Parser<'p> {
57     // Initial parser creation
58     fn new(tokens: Iter<'p, Token>) -> Self {
59         Self {
60             tokens,
61             current: Token::EoF,
62             peek: Token::EoF,
63         }
64     }
65
66     // Parse next statement if possible
67     // Output: statement or error
68     fn next(&mut self) -> Result<Option<Statement>, ParseError>
69     {
70         if self.current == Token::EoF {
71             return Ok(None);
72         }
73         Ok(Some(self.parse_statement(Precedence::Lowest)?))
74     }
75
76     // Set current and peek one step further in the vec of
77     // tokens
78     fn read(&mut self) {
79         self.current = self.peek.clone();
80         self.peek = if let Some(token) = self.tokens.next() {
81             token.clone()
82         } else {
83             Token::EoF
84         };
85     }
86
87     // See what the current token is
88     // Output: boolean
89     fn current_is(&self, token: Token) -> bool {
90         std::mem::discriminant(&self.current) == std::mem::discriminant(&token)

```

```

89     }
90
91     // Current token should match the one given
92     // Input: token
93     // Output: token or error
94     fn expect_token(&mut self, token: Token) -> Result<Token,
95         ParseError> {
96         if self.current_is(token) {
97             Ok(self.current.clone())
98         } else {
99             Err(ParseError::UnexpectedToken(self.current.clone
100             ()))
101         }
102     }
103
104     // Current token should match the one given and read to
105     // next token
106     // Input: token
107     // Output: token or error
108     fn expect_token_and_read(&mut self, token: Token) -> Result
109     <Token, ParseError> {
110         let result = self.expect_token(token)?;
111         self.read();
112         Ok(result)
113     }
114
115     // Parse statement, can only be a function or combination
116     // of functions
117     // Input: precedence
118     // Output: statement or error
119     fn parse_statement(&mut self, precedence: Precedence) ->
120     Result<Statement, ParseError> {
121         let mut statement = match self.current.clone() {
122             Token::Contains => Statement::Contains {
123                 expression: self.parse_contains()?,
124             },
125             Token::Starts => Statement::Starts {
126                 expression: self.parse_starts()?,
127             },
128             Token::Inflection => Statement::Inflection {

```

```

123         expression: self.parse_inflection()?,
124     },
125     Token::Thesaurus => Statement::Thesaurus {
126         expression: self.parse_thesaurus()?,
127     },
128     Token::Near => {
129         let (parameter, proximity) = self.parse_near()
130             ?;
131         Statement::Near {
132             parameter,
133             proximity,
134         }
135     },
136     Token::Weighted => Statement::Weighted {
137         parameter: self.parse_weighted()?,
138     },
139     _ => return Err(ParseError::UnexpectedToken(self.
140         current.clone())),
141 };
142 // After a function could be an infix operator
143 while !self.current_is(Token::Eof) && precedence <
144     Precedence::token(self.current.clone()) {
145     if let Some(in_statement) = self.
146         parse_infix_statement(statement.clone())? {
147         statement = in_statement
148     } else {
149         break;
150     }
151 }
152 Ok(statement)
153 }
154
155 // Parse expression, could be a search term, number,
156 // operator or combination of expressions
157 fn parse_expression(&mut self, precedence: Precedence) ->
158     Result<Expression, ParseError> {
159     let mut expr = match self.current.clone() {
160         Token::WordOrPhrase(s) => {
161             self.expect_token_and_read(Token::WordOrPhrase(
162                 s.to_string()))?;

```

```

156         Expression::WordOrPhrase(s.to_string())
157     }
158     Token::Number(u) => {
159         self.expect_token_and_read(Token::Number(0))?;
160         Expression::Number(u)
161     }
162     Token::ZeroToOne(f) => {
163         self.expect_token_and_read(Token::ZeroToOne
164             (0.0))?;
165         Expression::ZeroToOne(f)
166     }
167     t @ Token::Minus | t @ Token::Bang => {
168         self.expect_token_and_read(t.clone())?;
169         Expression::Prefix(
170             Operator::token(t),
171             Box::new(self.parse_expression(Precedence::
172                 Prefix)?),
173         )
174     }
175     // Start a group which gets higher precedence
176     Token::LeftParen => {
177         let group_expression = match self.parse_group()
178             ? {
179                 Statement::Group { expression } =>
180                     expression,
181                 _ => return Err(ParseError::Unreachable),
182             };
183         group_expression
184     }
185     _ => return Err(ParseError::UnexpectedToken(self.
186         current.clone())),
187 };
188 // After an expression could be an infix operator or
189 // directly a new expression (here called postfix
190 // operator)
191 while !self.current_is(Token::EoF) && precedence <
192     Precedence::token(self.current.clone()) {
193     if let Some(expression) = self.
194         parse_postfix_expression(expr.clone())? {
195         expr = expression;
196     }

```

```

187         } else if let Some(expression) = self.
188             parse_infix_expression(expr.clone())? {
189             expr = expression
190         } else {
191             break;
192         }
193     Ok(expr)
194 }
195
196 // Postfix operator is called when two expressions are read
197 // , automatically inserting an AND inbetween
198 // Second Expression could have an NOT operator before the
199 // actual expression
200 fn parse_postfix_expression(
201     &mut self,
202     expr: Expression,
203 ) -> Result<Option<Expression>, ParseError> {
204     Ok(match self.current {
205         Token::Minus | Token::Bang | Token::WordOrPhrase
206         (..) => {
207             let sec_expr = self.parse_expression(Precedence
208                 ::And)?;
209             Some(Expression::Infix(
210                 Box::new(expr),
211                 Operator::And,
212                 Box::new(sec_expr),
213             ))
214         }
215         _ => None,
216     })
217 }
218
219 // Infix operators AND and OR expect an expression on
220 // either side
221 fn parse_infix_expression(
222     &mut self,
223     expr: Expression,
224 ) -> Result<Option<Expression>, ParseError> {
225     Ok(match self.current {

```



```

221         Token::Plus | Token::And | Token::Or => {
222             let token = self.current.clone();
223             self.read();
224             let sec_expr = self.parse_expression(Precedence
225                 ::token(token.clone()))?;
226             Some(Expression::Infix(
227                 Box::new(expr),
228                 Operator::token(token),
229                 Box::new(sec_expr),
230             ))
231         }
232     }
233 }
234
235 // Infix operators AND and OR expect a statement on either
236 // side
237 fn parse_infix_statement(
238     &mut self,
239     statement: Statement,
240 ) -> Result<Option<Statement>, ParseError> {
241     Ok(match self.current {
242         Token::Plus | Token::And | Token::Or => {
243             let token = self.current.clone();
244             self.read();
245             let second_statement = self.parse_statement(
246                 Precedence::token(token.clone()))?;
247             Some(Statement::Infix {
248                 statement: Box::new(statement),
249                 operator: Operator::token(token),
250                 second_statement: Box::new(second_statement),
251             })
252         }
253         _ => None,
254     })
255 }
256
257 // Functions all have a similar strucure needing colons to
258 // surround their parameters

```

```
256
257 // Contains function only expects one word or phrase or
    combination of expressions
258 fn parse_contains(&mut self) -> Result<Expression,
    ParseError> {
259     self.expect_token_and_read(Token::Contains)?;
260     self.expect_token_and_read(Token::Colon)?;
261     let expression: Expression = self.parse_expression(
        Precedence::Statement)?;
262     self.expect_token_and_read(Token::Colon)?;
263     Ok(expression)
264 }
265
266 // Startswith function only expects one one word or phrase
    or combination of expressions
267 fn parse_starts(&mut self) -> Result<Expression, ParseError
    > {
268     self.expect_token_and_read(Token::Starts)?;
269     self.expect_token_and_read(Token::Colon)?;
270     let expression: Expression = self.parse_expression(
        Precedence::Statement)?;
271     self.expect_token_and_read(Token::Colon)?;
272     Ok(expression)
273 }
274
275 // Inflection function only expects one one word or phrase
    or combination of expressions
276 fn parse_inflection(&mut self) -> Result<Expression,
    ParseError> {
277     self.expect_token_and_read(Token::Inflection)?;
278     self.expect_token_and_read(Token::Colon)?;
279     let expression: Expression = self.parse_expression(
        Precedence::Statement)?;
280     self.expect_token_and_read(Token::Colon)?;
281     Ok(expression)
282 }
283
284 // Thesaurus function only expects one one word or phrase
    or combination of expressions
```

```

285 fn parse_thesaurus(&mut self) -> Result<Expression,
    ParseError> {
286     self.expect_token_and_read(Token::Thesaurus)?;
287     self.expect_token_and_read(Token::Colon)?;
288     let expression: Expression = self.parse_expression(
        Precedence::Statement)?;
289     self.expect_token_and_read(Token::Colon)?;
290     Ok(expression)
291 }
292
293 // Near function expects multiple comma-separated words or
    phrases with an optional number as the last parameter
294 fn parse_near(&mut self) -> Result<(Vec<Expression>,
    Expression), ParseError> {
295     self.expect_token_and_read(Token::Near)?;
296     self.expect_token_and_read(Token::Colon)?;
297     let mut parameter: Vec<Expression> = Vec::new();
298     // Proximity has a default value of 5 if no number is
        given
299     let mut proximity = Expression::Number(5);
300     while !self.current_is(Token::Colon) {
301         if self.current_is(Token::Comma) {
302             self.expect_token_and_read(Token::Comma)?;
303         }
304         match self.parse_expression(Precedence::Lowest)? {
305             Expression::WordOrPhrase(s) => parameter.push(
                Expression::WordOrPhrase(s)),
306             Expression::Number(u) => {
307                 if self.current_is(Token::Colon) {
308                     proximity = Expression::Number(u)
309                 } else {
310                     return Err(ParseError::UnexpectedToken(
                        self.current.clone()));
311                 }
312             }
313             _ => return Err(ParseError::UnexpectedToken(
                self.current.clone())),
314         }
315     }
316     self.expect_token_and_read(Token::Colon)?;

```

```

317         Ok((parameter, proximity))
318     }
319
320     // Weighted function expects pairs of words or phrases and
321     // a weight between 0 and 1
322     // All weights must add up to exactly 1
323     fn parse_weighted(&mut self) -> Result<Vec<(Expression,
324         Expression)>, ParseError> {
325         self.expect_token_and_read(Token::Weighted)?;
326         self.expect_token_and_read(Token::Colon)?;
327         let mut parameter: Vec<(Expression, Expression)> = Vec
328             ::new();
329         let mut sum_weights: f64 = 0.0;
330         while !self.current_is(Token::Colon) {
331             if self.current_is(Token::Comma) {
332                 self.expect_token_and_read(Token::Comma)?;
333             }
334             let expression = match self.parse_expression(
335                 Precedence::Lowest)? {
336                 Expression::WordOrPhrase(s) => Expression::
337                     WordOrPhrase(s),
338                 _ => return Err(ParseError::UnexpectedToken(
339                     self.current.clone())),
340             };
341             self.expect_token_and_read(Token::Comma)?;
342             let weight = match self.parse_expression(Precedence
343                 ::Lowest)? {
344                 Expression::ZeroToOne(f) => {
345                     sum_weights += f;
346                     Expression::ZeroToOne(f)
347                 }
348                 _ => return Err(ParseError::UnexpectedToken(
349                     self.current.clone())),
350             };
351             parameter.push((expression, weight));
352         }
353         if sum_weights != 1.0 {
354             return Err(ParseError::WeightError(sum_weights));
355         }
356         self.expect_token_and_read(Token::Colon)?;

```

```

349         Ok(parameter)
350     }
351
352     // Groups must encapsulate an expression with parentheses
353     // and have higher precedence than other operators
354     fn parse_group(&mut self) -> Result<Statement, ParseError>
355     {
356         self.expect_token_and_read(Token::LeftParen)?;
357         let expression = self.parse_expression(Precedence::
358             Statement)?;
359         self.expect_token_and_read(Token::RightParen)?;
360         Ok(Statement::Group { expression })
361     }
362 }
363
364 // Types of errors covered by the parser
365 #[derive(Debug, Error)]
366 pub enum ParseError {
367     #[error("Unexpected token {0:?}.")]
368     UnexpectedToken(Token),
369     #[error("Entered unreachable code.")]
370     Unreachable,
371     #[error("Weights do not add up to 1.0. Sum of all weights: {0}")]
372     WeightError(f64),
373 }

```

Appendix 4: ast.rs

```

1 use crate::code_gen::lexer::Token;
2
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     },
13     Contains {
14         expression: Expression,
15     },
16     Starts {
17         expression: Expression,
18     },
19     Inflection {
20         expression: Expression,
21     },
22     Thesaurus {
23         expression: Expression,
24     },
25     Near {
26         parameter: Vec<Expression>,
27         proximity: Expression,
28     },
29     Weighted {
30         parameter: Vec<(Expression, Expression)>,
31     },
32     EoF,
33 }
34
35 #[derive(Debug, Clone, PartialEq)]
36 pub enum Expression {
37     WordOrPhrase(String),
38     Number(u64),
39     ZeroToOne(f64),
40     Infix(Box<Expression>, Operator, Box<Expression>),
41     Prefix(Operator, Box<Expression>),
42 }
43
44 #[derive(Debug, Clone, PartialEq)]
45 pub enum Operator {
46     And,
47     Or,
48     Not,
49 }
50

```

```

51 impl Operator {
52     pub fn token(token: Token) -> Self {
53         match token {
54             Token::And | Token::Plus => Self::And,
55             Token::Or => Self::Or,
56             Token::Minus | Token::Bang => Self::Not,
57             _ => unreachable!("{:?}", token),
58         }
59     }
60 }

```

Appendix 5: generator.rs

```

1 use std::slice::Iter;
2 use thiserror::Error;
3
4 use crate::code_gen::ast::{Expression, Operator, Statement};
5
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;
11
12 // Main function to start the generation process
13 // Input: vec of statements (ast)
14 // Output: string (sql statement)
15 pub fn generate(ast: Vec<Statement>) -> Result<String,
16     GenerateError> {
17     let mut generator = Generator::new(ast.iter());
18     // write twice to overwrite initial EOF tokens
19     generator.write();
20     generator.write();
21     let mut sql_parts: Vec<String> = Vec::new();
22     sql_parts.push(format!(
23         "USE {}; SELECT TOP {} * FROM(SELECT FT_TBL.{{}}, KEY_TBL
24             .RANK FROM {} AS FT_TBL INNER JOIN CONTAINSTABLE({{},
25                 *, ' ",
26         DB_NAME, TOP_ROWS, RETURN_ATTRIBUTE, TBL_NAME, TBL_NAME

```

```

24     ));
25     // generate all functions as JOIN constraints
26     while let Some(sql_part) = generator.next()? {
27         sql_parts.push(sql_part);
28     }
29     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());
30     Ok(sql_parts.join(" "))
31 }
32
33 // Generator struct with current and next token 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     }
49
50     // Generate next statement if possible
51     fn next(&mut self) -> Result<Option<String>, GenerateError>
52     {
53         if self.current == Statement::EoF {
54             return Ok(None);
55         }
56         Ok(Some(self.generate_statement(self.current.clone())?))
57     }
58
59     // Set current and peek one step further in the ast
60     fn write(&mut self) {

```



```

60     self.current = self.peek.clone();
61     self.peek = if let Some(statement) = self.statements.
        next() {
62         statement.clone()
63     } else {
64         Statement::EoF
65     };
66 }
67
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 {
73         Statement::Infix {
74             statement,
75             operator,
76             second_statement,
77         } => {
78             let sql_parts = [
79                 self.generate_statement(*statement)?,
80                 self.generate_operator(operator)?,
81                 self.generate_statement(*second_statement)
82                     ?,
83             ];
84             sql_parts.join(" ")
85         }
86         // Contains generates it's search condition without
            mutation
87         Statement::Contains { expression } => {
88             format!("{}", self.generate_expression(
89                 expression)?)
90         }
91         // Startswith adds a * to end of a word or before
            the last " in a phrase
92         Statement::Starts { expression } => {
93             let mut word_or_phrase = self.
94                 generate_expression(expression)?;

```

```

92         if word_or_phrase.starts_with('"') &&
           word_or_phrase.ends_with('"') {
93             word_or_phrase.insert(word_or_phrase.len()
                                   - 1, '*');
94         } else {
95             word_or_phrase.push('*');
96         }
97         format!("{}", word_or_phrase)
98     }
99     // Inflection calls the inflection function from
      MSSQL
100     Statement::Inflection { expression } => {
101         let mut word_or_phrase = self.
            generate_expression(expression)?;
102         if word_or_phrase.starts_with('"') &&
           word_or_phrase.ends_with('"') {
103             word_or_phrase.remove(0);
104             word_or_phrase.remove(word_or_phrase.len()
                                   - 1);
105         }
106         format!("FORMSOF (INFLECTIONAL, \"{}\\\" )",
            word_or_phrase)
107     }
108     // Thesaurus calls the thesaurus function from
      MSSQL
109     Statement::Thesaurus { expression } => {
110         let mut word_or_phrase = self.
            generate_expression(expression)?;
111         if word_or_phrase.starts_with('"') &&
           word_or_phrase.ends_with('"') {
112             word_or_phrase.remove(0);
113             word_or_phrase.remove(word_or_phrase.len()
                                   - 1);
114         }
115         format!("FORMSOF (THESAURUS, \"{}\\\" )",
            word_or_phrase)
116     }
117     // Near generates a parameter list of all search
      criteria and proximity in the end
118     Statement::Near {

```

```

119         parameter,
120         proximity,
121     } => {
122         let mut sql_parts: Vec<String> = Vec::new();
123         sql_parts.push(format!("NEAR("));
124         for expression in parameter {
125             let string = self.generate_expression(
126                 expression)?;
127             sql_parts.push(format!("{}", string));
128             sql_parts.push(String::from(", "));
129         }
130         sql_parts.remove(sql_parts.len() - 1);
131         sql_parts.push(format("), {})", self.
132             generate_expression(proximity)?);
133         sql_parts.join("")
134     }
135     // Weighted generates tuples of search criteria and
136     // their respective weight
137     Statement::Weighted { parameter } => {
138         let mut sql_parts: Vec<String> = Vec::new();
139         sql_parts.push(format!("ISABOUT("));
140         for (word_or_phrase_expr, weight_expr) in
141             parameter {
142             let word_or_phrase = self.
143                 generate_expression(word_or_phrase_expr)
144                 ?;
145             let weight = self.generate_expression(
146                 weight_expr)?;
147             sql_parts.push(format!("{}", WEIGHT({}),",
148                 word_or_phrase, weight));
149             sql_parts.push(String::from(", "));
150         }
151         sql_parts.remove(sql_parts.len() - 1);
152         sql_parts.push(String::from(")"));
153         sql_parts.join("")
154     }
155     _ => return Err(GenerateError::UnexpectedStatement(
156         self.current.clone())),
157 };
158 self.write();

```

```

150         Ok(sql)
151     }
152
153     // Generate expression, any search criteria or number or
154     // combination of those
155     // Input: expression to generate
156     // Output: string
157     fn generate_expression(&mut self, expression: Expression)
158     -> Result<String, GenerateError> {
159         let sql: String = match expression {
160             Expression::WordOrPhrase(s) => s,
161             Expression::Number(u) => u.to_string(),
162             Expression::ZeroToOne(f) => f.to_string(),
163             // Infix operator enclose their expressions with
164             // parentheses to ensure precedence
165             Expression::Infix(expr1, operator, expr2) => {
166                 let mut sql_parts = [
167                     String::from("("),
168                     self.generate_expression(*expr1)?,
169                     String::from(")"),
170                     self.generate_operator(operator)?,
171                     String::from("("),
172                     self.generate_expression(*expr2.clone())?,
173                     String::from(")"),
174                 ];
175                 // If the second expression is a not operator
176                 // it must write NOT before the parentheses
177                 match *expr2 {
178                     Expression::Prefix(Operator::Not, ..) =>
179                     {
180                         sql_parts[4] = String::from("NOT ");
181                     }
182                     _ => {}
183                 }
184                 sql_parts.join(" ")
185             }
186             Expression::Prefix(operator, expr) => {
187                 let sql_parts = [
188                     self.generate_operator(operator)?,
189                     self.generate_expression(*expr)?,
190                 ];
191                 sql_parts.join(" ")
192             }
193         }
194     }

```

```

185         }
186     };
187     Ok(sql)
188 }
189
190 // Generate operator
191 // Input: operator to generate
192 // Output: string
193 fn generate_operator(&mut self, operator: Operator) ->
    Result<String, GenerateError> {
194     let op = match operator {
195         Operator::And => "AND",
196         Operator::Or => "OR",
197         // has to be set infront of parentheses, see
198         // generate_expression for infix
199         Operator::Not => "",
200     };
201     Ok(op.to_owned())
202 }
203
204 // Types of error covered by the generator
205 #[derive(Debug, Error)]
206 pub enum GenerateError {
207     #[error("Unexpected statement {0:?}.")]
208     UnexpectedStatement(Statement),
209 }

```

Appendix 6: mod.rs

```

1 mod ast;
2 pub mod generator;
3 pub mod lexer;
4 pub mod parser;

```

Appendix 7: base.html

```

1 <!DOCTYPE html>
2 <html lang="en">

```

```
3     <head>
4         <meta charset="utf-8">
5         <title>{{title}}</title>
6     </head>
7     <body>
8         {% block content %}
9         {% endblock %}
10    </body>
11 </html>
```

Appendix 8: search.html

```
1 {% extends "base.html" %}
2
3 {% block content %}
4 <form action="" method="POST">
5     <div>
6         <label for="search">Search:</label>
7         <input type="text" name="search">
8     </div>
9     <input type="submit" value="Submit">
10 </form>
11 {% endblock %}
```

Appendix 9: result.html

```
1 {% extends "base.html" %}
2
3 {% block content %}
4 <div>
5     <p>{{ search }}</p>
6 </div>
7 {% for result in results %}
8 <div>
9     <a href="https://en.wikipedia.org/wiki/{{ result.link }}">
10         {{ result.title }}</a>
11     <small>{{ result.rank }}</small>
12 </div>
13 {% endfor %}
```

```
13 | {% endblock %}
```

Appendix 10: convert_wiki_to_csv.py

```

1 from lxml import etree
2 from glob import glob
3 import bz2
4 import codecs
5 import csv
6 import time
7 import os
8
9 PATH_WIKI_XML = "/Users/light/Documents/Wikipedia/"
10 FILENAME_ARTICLES = "articles.csv"
11
12
13 def hms_string(sec_elapsed):
14     h = int(sec_elapsed / (60 * 60))
15     m = int((sec_elapsed % (60 * 60)) / 60)
16     s = sec_elapsed % 60
17     return "{}:{:>02}:{:>05.2f}".format(h, m, s)
18
19
20 def get_parser(filename):
21     ns_token = "{http://www.mediawiki.org/xml/export-0.10/}ns"
22     title_token = "{http://www.mediawiki.org/xml/export-0.10/}title"
23     revision_token = "{http://www.mediawiki.org/xml/export-0.10/}revision"
24     text_token = "{http://www.mediawiki.org/xml/export-0.10/}text"
25
26     with bz2.BZ2File(filename, "rb") as bz2_file:
27         for event, element in etree.iterparse(bz2_file, events
28             = ("end",)):
29             if element.tag.endswith("page"):
30                 namespace_tag = element.find(ns_token)
31
32                 if namespace_tag.text == "0":

```

```

32         title_tag = element.find(title_token)
33         text_tag = element.find(revision_token).
           find(text_token)
34         yield title_tag.text, text_tag.text
35
36         element.clear()
37
38
39 def pluck_wikipedia_titles_text(
40     out_file, pattern="enwiki-*--pages-articles-multistream*.xml
           -*.bz2"
41 ):
42     totalCount = 0
43     longestTitle = 0
44     longestText = 0
45     with codecs.open(out_file, "a+b", "utf8") as out_file:
46         writer = csv.writer(out_file)
47         for bz2_filename in sorted(
48             glob(pattern),
49             key=lambda a: int(a.split("articles-multistream")
                               [1].split(".")[0]),
50         ):
51             print(bz2_filename)
52             parser = get_parser(bz2_filename)
53             for title, text in parser:
54                 if not (text.startswith("#REDIRECT") or text.
                           startswith("#redirect")):
55                     totalCount += 1
56                     writer.writerow([title, text])
57                     longestTitle = (
58                         len(title) if len(title) > longestTitle
59                         else longestTitle
60                     )
61                     longestText = len(text) if len(text) >
62                         longestText else longestText
63                     if totalCount > 1 and (totalCount % 100000)
64                         == 0:
65                         print("{:,}".format(totalCount))
66                     if totalCount >= 1000:
67                         break

```



```
65     print (
66         f"Total Count: {totalCount}\nLongest Title: {
           longestTitle}\nLongest Text: {longestText}"
67     )
68
69
70 pathArticles = os.path.join(PATH_WIKI_XML, FILENAME_ARTICLES)
71
72 start_time = time.time()
73
74 pluck_wikipedia_titles_text(pathArticles)
75
76 time_took = time.time() - start_time
77 print(f"Total runtime: {hms_string(time_took)}")
```

Appendix 11: create_article.sql

```
1 USE Wikipedia;
2 DROP TABLE Real_Article;
3 CREATE TABLE [dbo].[Real_Article] (
4     [Title] [varchar] (255) NULL,
5     [Text] [varchar] (max) NULL
6 );
7 INSERT INTO Real_Article (Title, Text)
8 VALUES ('Example', 'Fulltext');
9 SELECT *
10 FROM Real_Article;
11 DELETE FROM Real_Article
12 WHERE Title = 'Example';
```

Appendix 12: bulk_insert.sql

```
1 USE Wikipedia;
2 BULK
3 INSERT Real_Article
4 FROM 'C:\Users\light\Documents\Wikipedia\enwiki-20220901-pages-
   articles-multistream-no-redirect.csv' WITH (
5     FIRSTROW = 1,
6     FIELDQUOTE = '\'
```

```
7      , FIELDTERMINATOR = ' , '
8      ,
9      , ROWTERMINATOR = ' \ n ' ,
10     TABLOCK) ;
```

Appendix 13: alter_id.sql

```
1 USE Wikipedia;
2 ALTER TABLE [dbo].[Real_Article]
3 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, 6.10.2022

(Location, Date)

A handwritten signature in blue ink, appearing to read 'S. Bunge', is written above a horizontal line.

(handwritten signature)