FOM Hochschule für Oekonomie & Management

university location Bonn

Bachelor Thesis

in the study course Wirtschaftsinformatik - Business Information Systems

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.-Ing. Peter Steininger
Matriculation Number:	539441
Submission:	October 28, 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
MS Microsoft
SQL Structured Query Language
TSQL Transact-SQL

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 Introduction

Current relational databases offer much more functionality than just transactional processing of data. One of these functionalities is the full-text search in documents, which allows possibilities like word and phrase-based searches and inflectional searches using specialized functions and combinations of query terms. In today's IT world, code generators, in conjunction with graphical user interfaces or Domain-Specific Language (DSL)s, make powerful tools such as full-text search more accessible, enabling individuals and companies with little IT knowledge to use them.

The development of a code generator including its DSL can be allocated in the research fields of compiler construction and programming language design. Knowledge in these areas allows a more abstract view of many other areas of development and how superficially independent topics can find an application in IT. The question to be answered within the scope of the thesis is 'To what extent can a software component be developed using a custom query language and code generator to enable full-text search on Structured Query Language (SQL) servers?'

The objective is a code generator that uses a custom query language to define search criteria that enable full-text search. The code generator takes text as input and generates SQL code which is executed on an SQL server. The respective search results should be displayed to the user, without them having to interact with SQL code or servers.

To achieve this objective, a prototype is developed that practically implements the concepts of compiler construction and represents the functionality more effectively than textual descriptions or purely static models. The implementation of a prototype is an iterative process: Implementation is followed by evaluating and adapting to new problems and specifications. In each case, as much functionality is implemented as is necessary to verify the targeted phase result (cf. POMBERGER et al. 1992, p. 3).

2 Theory

The query language to be developed can be classified as a DSL since it is designed specifically for full-text search to simplify the search process. Therefore, in chapter 2.1 DSLs are illustrated in greater detail.
The development of a query language also entails the design of the language itself. How this is done is described in chapter 2.2. The notation used for writing down the design choices is also discussed.
The technical part of the development, a code generator, is addressed in chapter 2.3 to specify common components and procedures.
Since the goal is to simplify the use of full-text search on SQL servers, chapter 2.4 explains how such a server is structured, and what the full-text search capabilities entail.

2.1 Domain-Specific Languages

Commonly known programming languages, such as C or Java, are also called 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 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 the 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	Application
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 this semantics (cf. HUDAK 1997, pp. 2-4).

2.2 Language Design

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, file 2).

2.2.1 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, file 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 2.2.2.

2.2.2 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).
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:

⟨*number*⟩ ::= ⟨*positive*⟩|-⟨*positive*⟩|0
⟨*positive*⟩ ::= ⟨*digit not zero*⟩⟨*optional*⟩
⟨*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';

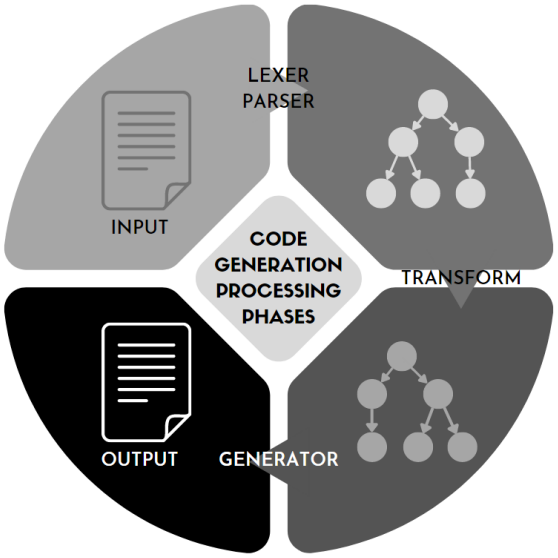
This version of the grammar produces the same set of numbers but is more concise and arguably more readable for humans.

2.3 Code Generator

As soon as a language is defined, for example by writing it down in EBNF, the development of a code generator can be started. Generators can be implemented using a

template-based approach (see CLEAVELAND 2001), the template metaprogramming capabilities of a language (e.g., in C++), or extendable programming systems (e.g., OpenC++) (cf. CZARNECKI, EISENECKER 2000, p. 16).
Code generators are divided into two phases, a front end, and a back end. The front end primarily contains a lexical and syntactical analysis of the input up to the conversion of this input into an intermediate representation, here Abstract Syntax Tree (AST). The back end then takes this representation and deals with the code generation (cf. WIRTH 1996, p. 7). With this modular decoupling, many different kinds of code generators can be built, which can implement the front end independent of the target language.
Some further split these into more phases or components, as shown in the following figure.

Figure 1: Code Generation Processing Phases



Source: Based on SARKAR, CLEAVELAND 2001, p. 5

A program written by a developer in either a predefined DSL or an executable model is stored in a file for development. The code generator is given the grammar of the source language, which it uses to split it into tokens and then relate them to each other by parsing them. Such dependencies can also exist between multiple input files, by importing or referencing each other. The parser then arranges them in an AST, which can be optimized and transformed for the target language. The generator or writer then generates a file in the target language from this AST (cf. SARKAR, CLEAVELAND 2001, pp. 5-6).
The code generator built in this thesis makes use of a lexer that takes care of the first initial step of scanning. Scanning is a good starting point and the first step in any compiler or interpreter. It takes the input file and groups a series of characters into so-called tokens. These words or punctuation make up the grammar of the query language (cf. NYSTROM 2021, p. 39).

Keywords and many other special kinds of tokens are almost always part of a grammar of a language and have to be handled separately from other tokens, to achieve that tokens must be assigned a kind to differentiate them. Each keyword, operator, punctuation, and literal type are some kind of token (cf. NYSTROM 2021, p. 43). Tokens can be recognized either by simply matching them to a character like '(' which would be then recognized as a left parenthesis token or if a token is a more complex logic like if statements or regex can be used to further distinguish tokens (see NYSTROM 2021, pp. 47-54).
The next goal is a transformation of the series of tokens into a more complex in-memory representation that can be used by an interpreter or, in the case of the thesis, a generator. This representation should be simple to generate for the parser and easy to understand for the next component (cf. NYSTROM 2021, p. 59).
There exist two different types of parsers, namely, the event-driven parser and the tree-constructing parser (cf. SARKAR, CLEAVELAND 2001, p. 5). Event-driven parsers search the input file for predefined syntax structures and notify the user when one is found. The user can then decide which events to ignore and which to include in their data structure. Tree constructors document related tokens as an AST. Thus, this AST contains an in-memory representation of the input files on which all further processing algorithms are based. If several languages and grammars are parsed several ASTs can develop (cf. SARKAR, CLEAVELAND 2001, pp. 5-6).
In the context of the thesis, a tree-constructing parser is implemented which will be based on a previously defined grammar. The use of a notation such as EBNF, described in chapter 2.2.2, helps to silhouette the informal syntax design and can serve as a guide when implementing the parser (cf. NYSTROM 2021, p. 64).
In the implementation of the parser, many types of so-called expressions are needed to define parts of the language. For example, there are literal expressions, which represent elementary values, such as numbers, booleans, or strings, and binary expressions, which logically connect two other expressions with an operator. These different types of expressions are to be recognized by the parser and sorted accordingly into the tree structure (see NYSTROM 2021, pp. 64-68).
With expressions alone, it is possible to correctly process the syntax for inputs like $1 * (2 - 3) == true$ according to their respective priority (cf. NYSTROM 2021, p. 64).
Since the language to be developed is only a query language, it does not have to be able to store variables or similar, but it is still necessary to implement statements in addition to expressions. Statements do not evaluate to a value themselves but allow the use of functions and bindings that can be passed expressions to process (cf. NYSTROM 2021, pp. 111-112). To be able to parse statements, certain tokens trigger the start of a statement, such as `print` or `for` in C. Each kind of statement has its syntax, which defines the

structure of the statement and expected tokens/expressions. If a token is not a trigger to a statement it is automatically assumed that an expression must be processed. If a statement is parsed correctly it is placed in the corresponding form in the AST (cf. NYSTROM 2021, pp. 113-116).
Depending on the type of code generator that is to be implemented, different paths can be taken after the transfer to an intermediate representation. If the goal is to generate machine code or bytecode for a virtual machine, optimization and selection techniques would be applied to make the code as fast as possible (cf. NYSTROM 2021, p. 15).
This would go beyond the scope of the thesis since the goal is only a translation into Transact-SQL (TSQL) to enable a full-text search on a SQL server, so an alternative route is taken here: a transpiler.
A transpiler hijacks the backend of a different GPL by producing a string of valid source code of that GPL, in this case, TSQL (cf. NYSTROM 2021, p. 17). This means the only missing part of the query language code generator is a writer, further referred to as just generator, which translates the AST into valid TSQL code.

2.4 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 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: Weighted harmonic mean

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

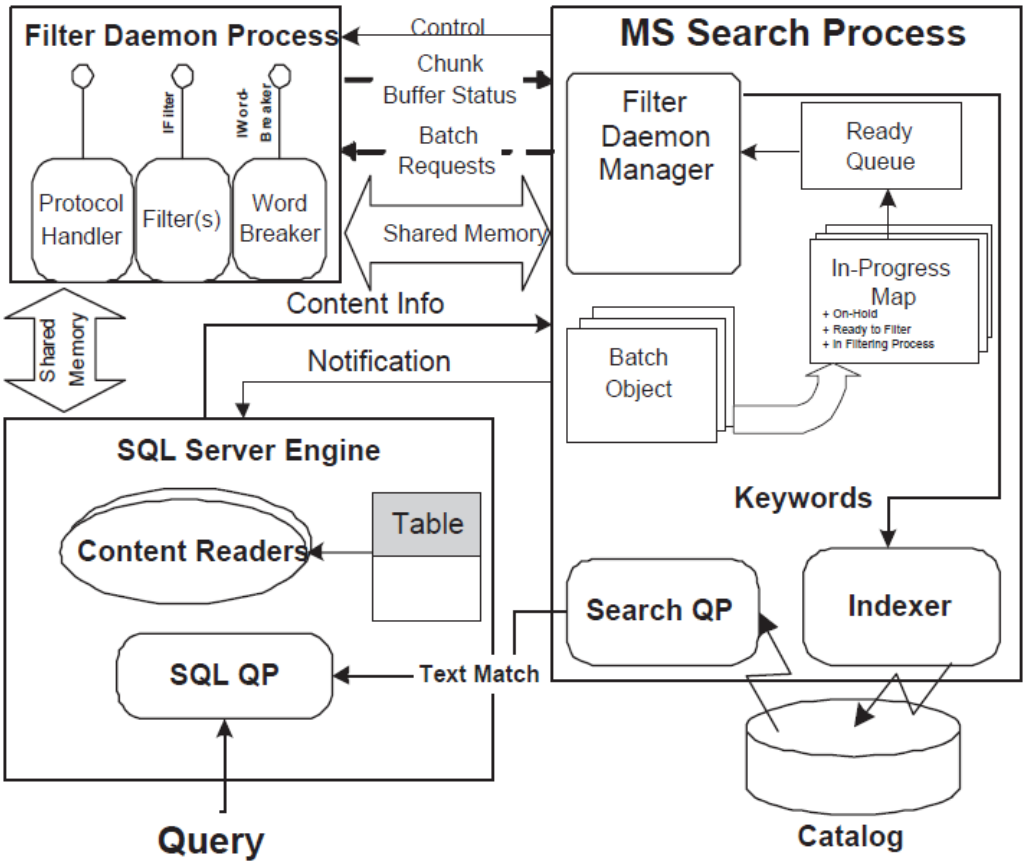
Source: COLES, COTTER 2009, p. 15

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).

2.4.1 SQL Server Search Architecture

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 HTML and XML, and even complex documents, such as 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 (cf. HAMILTON, NAYAK 2001, pp. 8-9). (See figure 2)
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 2: Architecture of 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, which 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.

2.4.2 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'.

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. To define a full-text search, usually use a combination of the given functions is used. In TSQL this would be for example CONTAINS or FORMSOF. Therefore the goal is to develop a query language where only a combination of functions and a few parameters is necessary to generate the corresponding SQL.

3.1 Language Design

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'. To represent 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 simply being processed from left to right.

To cover a large part of the possible full-text search queries, six functions were finally selected which are to be implemented in the query language:
'Contains' should be a simple search for a search term or a combination of search terms.
'Startswith' searches for terms, which start with the given search term.
'Inflection' takes the given search terms and searches for words with the same root and variations of it.
'Thesaurus' uses a thesaurus to search for entries with the same meaning as the search terms.
'Near' can search for documents where two or more search terms must occur within a certain distance of each other. Distance in this case means how many words separate the search terms
'Weighted' enables the search for multiple search terms and assigns each a weight to allow certain terms to be prioritized.
Some of the functions require the definition of more possible parameters than just words and phrases. For the function Near a positive integer is needed to specify the distance and for Weighted positive decimal numbers between zero and one are needed to assign a weight to the search parameters. In addition, functions should also be combinable with operators.
These specifications and rules can be defined in EBNF as follows:

⟨*search*⟩ ::= ⟨*function*⟩{[⟨*infix*⟩⟨*function*⟩]};
⟨*infix*⟩ ::= '+'|'&'|'|';
⟨*function*⟩ ::= '@'(⟨*contains*⟩|⟨*startswith*⟩|⟨*inflection*⟩|⟨*thesaurus*⟩|⟨*near*⟩|⟨*weighted*⟩);

⟨*contains*⟩ ::= 'contains:'⟨*expression*⟩':';
⟨*startswith*⟩ ::= 'startswith:'⟨*expression*⟩':';
⟨*inflection*⟩ ::= 'inflection:'⟨*wordorphrase*⟩':';
⟨*thesaurus*⟩ ::= 'thesaurus:'⟨*wordorphrase*⟩':';
⟨*near*⟩ ::= 'near:'⟨*wordorphrase*⟩{','⟨*wordorphrase*⟩}[','⟨*posinteger*⟩]':';
⟨*weighted*⟩ ::= 'weighted:'{⟨*wordorphrase*⟩','⟨*zerotoone*⟩}':';

⟨*expression*⟩ ::= ⟨*wordorphrase*⟩[' '⟨*wordorphrase*⟩]|'('⟨*expression*⟩')'|('-'|'!')⟨*expression*⟩
|⟨*expression*⟩⟨*infix*⟩⟨*expression*⟩;
⟨*wordorphrase*⟩ ::= ⟨*word*⟩|' " '{[' ']⟨*word*⟩}' " ';
⟨*word*⟩ ::= {'a'-'z'|'A'-'Z'};
⟨*posinteger*⟩ ::= {'0'-'9'};
⟨*zerotoone*⟩ ::= {0}['.'{'0'-'9'}]|'1';

3.2 Code Generator

3.2.1 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 }
   ⋮
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]
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.

3.2.2 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)?;
341             let weight = match self.parse_expression(Precedence
                    ::Lowest)? {
342                 Expression::ZeroToOne(f) => {
343                     sum_weights += f;
344                     Expression::ZeroToOne(f)
345                 }
346                 _ => return Err(ParseError::UnexpectedToken(
                        self.current.clone())),
347             };
348             parameter.push((expression, weight));
349         }
350         if sum_weights != 1.0 {
351             return Err(ParseError::WeightError(sum_weights));
352         }
353         self.expect_token_and_read(Token::Colon)?;
354         Ok(parameter)
355     }
```

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())
102         } else {
103             Err(ParseError::UnexpectedToken(self.current.clone
                    ()))
104         }
105     }
106
107     // Current token should match the one given and read to
            next token
108     // Input: token
109     // Output: token or error
110     fn expect_token_and_read(&mut self, token: Token) -> Result
            <Token, ParseError> {
111         let result = self.expect_token(token)?;
112         self.read();
113         Ok(result)
114     }
```

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.

3.2.3 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 AST, there are separate types for them as both statements and expressions. 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>),
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  }
```

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,
28      Or,
29      And,
30      Not,
31      Prefix,
32      Group,
33  }
34
35  // Match tokens to precedences
36  impl Precedence {
37      fn token(token: Token) -> Self {
38          match token {
39              Token::Bang | Token::Minus => Self::Not,
40              Token::Plus | Token::And | Token::WordOrPhrase(..)
                    => Self::And,
41              Token::Or => Self::Or,
42              Token::LeftParen => Self::Group,
43              Token::Contains
44              | Token::Starts
45              | Token::Inflection
46              | Token::Thesaurus
47              | Token::Near
48              | Token::Weighted => Self::Statement,
49              _ => Self::Lowest,
50          }
51      }
52  }
```

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),
```

```
175                     Box::new(self.parse_expression(Precedence::
                            Prefix)?),
176                 )
177             }
```

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();
```

```
228                 self.read();
229                 let sec_expr = self.parse_expression(Precedence
                        ::token(token.clone()))?;
230                 Some(Expression::Infix(
231                     Box::new(expr),
232                     Operator::token(token),
233                     Box::new(sec_expr),
234                 ))
235             }
236             _ => None,
237         })
238     }
```

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,
```

```
217            })
218        }
```

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

```
145         // After a function could be an infix operator
146         while !self.current_is(Token::EoF) && precedence <
               Precedence::token(self.current.clone()) {
147             if let Some(in_statement) = self.
                   parse_infix_statement(statement.clone())? {
148                 statement = in_statement
149             } else {
150                 break;
151             }
152         }
⋮
240     // Infix operators AND and OR expect a statement on either
           side
241     fn parse_infix_statement(
242         &mut self,
243         statement: Statement,
244     ) -> Result<Option<Statement>, ParseError> {
245         Ok(match self.current {
246             Token::Plus | Token::And | Token::Or => {
247                 let token = self.current.clone();
248                 self.read();
249                 let second_statement = self.parse_statement(
                       Precedence::token(token.clone()))?;
250                 Some(Statement::Infix {
251                     statement: Box::new(statement),
252                     operator: Operator::token(token),
253                     second_statement: Box::new(second_statement
                           ),
254                 })
255             }
256             _ => None,
257         })
258     }
```

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.

3.2.4 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 TSQL 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].[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
```

```
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());
```

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>
            {
```

```
52          if self.current == Statement::EoF {
53              return Ok(None);
54          }
55          Ok(Some(self.generate_statement(self.current.clone())?)
                )
56      }
57
58      // Set current and peek one step further in the ast
59      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      }
```

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, expressions, and 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 {
```

```
⋮
133             // Weighted generates tuples of search criteria and
                    their respective weight
134             Statement::Weighted { parameter } => {
135                 let mut sql_parts: Vec<String> = Vec::new();
136                 sql_parts.push(format!("ISABOUT("));
137                 for (word_or_phrase_expr, weight_expr) in
                        parameter {
138                     let word_or_phrase = self.
                            generate_expression(word_or_phrase_expr)
                            ?;
139                     let weight = self.generate_expression(
                            weight_expr)?;
140                     sql_parts.push(format!("{} WEIGHT({})",
                            word_or_phrase, weight));
141                     sql_parts.push(String::from(", "));
142                 }
143                 sql_parts.remove(sql_parts.len() - 1);
144                 sql_parts.push(String::from(")"));
145                 sql_parts.join("")
146             }
⋮
148         };
149         self.write();
150         Ok(sql)
151     }
```

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 = [
```

```
79                      self.generate_statement(*statement)?,
80                      self.generate_operator(operator)?,
81                      self.generate_statement(*second_statement)
                            ?,
82                  ];
83                  sql_parts.join(" ")
84              }
```

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

If a form of an operator is passed as an expression, the corresponding generate function is called for the operators (code 21, 167+181) and the respective expressions are passed to the generate_expression function. In the case of infix operators, parentheses are also set to safely represent the precedence as recognized by the parser (code 21, 164-170). Something to pay attention to in SQL is that if a NOT operator follows an infix operator, it must be written before the parenthesis, otherwise, SQL throws a syntax error. To cover this case, infix operator generation checks if the second expression is a negation and rewrites the string in that case (code 21, 173-176).

Code Listing 21: Generate operator expressions

```
161             // Infix operator enclose their expressions with
                   parentheses to ensure precedence
162             Expression::Infix(expr1, operator, expr2) => {
163                 let mut sql_parts = [
164                     String::from("("),
165                     self.generate_expression(*expr1)?,
166                     String::from(")"),
167                     self.generate_operator(operator)?,
168                     String::from("("),
169                     self.generate_expression(*expr2.clone())?,
170                     String::from(")"),
171                 ];
172                 // If the second expression is a not operator
                       it must write NOT before the parenthesis
173                 match *expr2 {
174                     Expression::Prefix(Operator::Not, ..) =>
                           sql_parts[4] = String::from("NOT ("),
175                     _ => (),
176                 }
177                 sql_parts.join(" ")
178             }
179             Expression::Prefix(operator, expr) => {
180                 let sql_parts = [
181                     self.generate_operator(operator)?,
182                     self.generate_expression(*expr)?,
183                 ];
184                 sql_parts.join(" ")
185             }
```

Source: generator.rs

The third and simplest generate function is generate_operator, which translates operators into a string (code 22, 195-196). Noticeable here is the NOT operator, which is already treated separately in code listing 21 lines 173-176.

Code Listing 22: Generate operators

```
190     // Generate operator
191     // Input: operator to generate
192     // Output: string
```

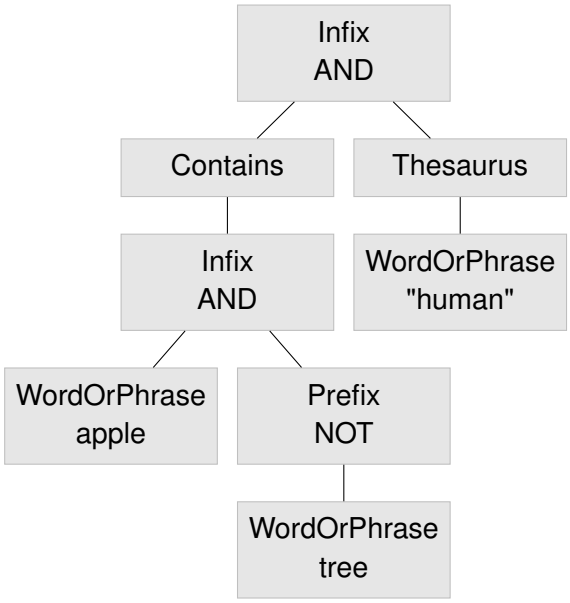
```rust
193     fn generate_operator(&mut self, operator: Operator) ->
            Result<String, GenerateError> {
194         let op = match operator {
195             Operator::And => "AND",
196             Operator::Or => "OR",
197             // has to be set infront of parentheses, see
                    generate_expression for infix
198             Operator::Not => "",
199         };
200         Ok(op.to_owned())
201     }
```

Source: generator.rs

3.2.5 Example Generation

All parts up to the generator combined already make up a working code generator for a custom query language. For example the input `@contains:apple -tree: + @thesaurus:"human":`, the lexer converts into a list of tokens. The parser then interprets these tokens into an AST, one exemplary representation can be seen in figure 3.

Figure 3: Parsed example AST



Source: Own representation

The generator then takes the AST and generates from it the SQL in code listing 23. Lines 1 to 9 and 11 to 14 are the more or less pre-formulated part using the constants of the test database. Line 10 is the intriguing part, which is generated from the passed functions and search criteria.

Code Listing 23: Generated example SQL

```
1  USE Wikipedia;
2  SELECT TOP 5 *
3  FROM(
4          SELECT FT_TBL.Title,
5              KEY_TBL.RANK
6          FROM [dbo].[Article] AS FT_TBL
7              INNER JOIN CONTAINSTABLE(
8                  [dbo].[Article],
9                  *,
10                 ' ( apple ) AND NOT ( tree ) AND FORMSOF(
                       THESAURUS,"human") '
11             ) AS KEY_TBL ON FT_TBL.[ID] = KEY_TBL.[KEY]
12         WHERE KEY_TBL.RANK > 5
13     ) AS FS_RESULT
14 ORDER BY FS_RESULT.RANK DESC;
```

Source: Own code generator

3.3 Interface SQL Server

To interface with the SQL server hosting the full-text index and test data a connection must be established to run any queries. Since the server is hosted on the same machine as the Rust script, the local command line can be used. The command is passed the SQL as a path to the respective file and the result is then written to a path by the server as well (code 24, 67-70).

Code Listing 24: Execute SQL

```
57 // Runs a command to execute an sql statement to a local MSSQL
       Server
58 // Input: paths to the input file and where to write the result
59 // Output: txt file interpretation of the MSSQL Server result
60 fn execute_sql(sql_path: &str, results_path: &str) {
61     Command::new("cmd")
62         .args(&[
```

```
            "/C",
            "sqlcmd",
            "-S",
            "DESKTOP-JKNEH40\\SQLEXPRESS", //Local server name
            "-i",
            sql_path,
            "-o",
            results_path,
        ])
        .output()
        .expect("failed to execute operation");
}
```

Source: main.rs

The result of the server is stored in a txt file. The content is formatted to be easier for a human to read, but less suitable for machine processing. The first and last lines of the file contain query metadata, such as column names and numbers, and characters to visually delimit them, so they are removed (code 25, 86-94). The results themselves are formatted with unnecessary whitespace, and the last entry in a row is interpreted as the rank value (code 25, 99-106).

Code Listing 25: SQL server results

```
// 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);
```

```
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 }
```

Source: main.rs

These two functions enable the automatic execution and evaluation of queries on the SQL server, where any kind of test data can be stored.

3.4 Website as User Interface

To interact with the code generator a website is used as an interface. To create a simple website the Rust crate actix_web is used, which provides a web framework for Rust, and the crate tera, which enables the use of templates to quickly create a working frontend.
To host a website using actix_web, an HTTP server must be started in an asynchronous main function (cf. EDE 2022, n.p.). With the help of tera, HTML templates can now be used (cf. PROUILLET 2022, n.p.) to pass data to the server (code 26, 20+22). The HTML code of the website can be found in the appendix. The HTTP server hosts two subpages (code 26, 23-24). These are the start page, where a search query can be submitted, and the result page, where the search results will be displayed.

Code Listing 26: HTTP Server

```
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/**/*").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 }
```

Source: main.rs

The interface between the website and Rust code serves two structs, which can exchange data between the two parties. For this purpose, the crate serde is used, with which structs can be easily serialized (cf. TOLNAY, TRYZELAAR 2017, n.p.). A struct for the search query including a string is needed (code 27, 114-117), and a struct for the search results, which contain a title and link as a string and a rank as an unsigned integer each (code 27, 118-123).

Each of the subpages is assigned a function, which defines for the respective page, what functionally happens on it. The search page is straightforward, with a text field that can be submitted (code 27, 125-131).

Code Listing 27: Website search

```
113 // Search and Result structs to (de)serialize rust and website
    datatypes
114 #[derive(Deserialize)]
115 struct Search {
116     search: String,
117 }
118 #[derive(Serialize)]
119 struct Result {
120     title: String,
121     rank: u64,
122     link: String,
```

```
123 }
124
125 // Define functional parts of the search page
126 async fn search(tera: web::Data<Tera>) -> impl Responder {
127     let mut data = Context::new();
128     data.insert("title", "Search field");
129     let rendered = tera.render("search.html", &data).unwrap();
130     HttpResponse::Ok().body(rendered)
131 }
```

Source: main.rs

The function of the result page is a bit more complex because here the code generator is run, the SQL is executed and the results are read. Because the result page resembles a POST request the function is passed the data of the search page. From this, the search query is extracted and the code generator is executed (code 28, 138). With the help of two functions, the generated SQL is executed and the results are read (code 28, 141-142). The results of the query are in the form of a string-integer tuple and are now fitted into the form of the result struct (code 28, 146-153). The link attribute is very similar to the title, but all blanks are replaced with _ so that this attribute can be used to link directly to the associated Wikipedia article, a small quirk of the chosen test data.

Code Listing 28: Website results

```
133 // Define functional parts of the result page
134 async fn result(tera: web::Data<Tera>, data: web::Form<Search>)
        -> impl Responder {
135     let mut page_data = Context::new();
136     let mut results: Vec<Result> = Vec::new();
137     // Run code generator with the string from the search field
138     match run_code_gen(data.search.clone(), PATH_SQL) {
139         // If code generator returns no error execute SQL and
              read the results
140         Ok(_) => {
141             execute_sql(PATH_SQL, PATH_RESULTS);
142             let results_vec = read_results(PATH_RESULTS);
143             // Fit search results into Result struct to
                  properly display on the page, otherwise diplay
                  error
144             match results_vec {
145                 Some(results_vec) => {
146                     for result in results_vec {
```

```
147                             results.push(Result {
148                                 title: result.0.clone(),
149                                 rank: result.1,
150                                 // link to the Wikipedia article is
                                        also provided, whitespaces need
                                        to be replaced
151                                 link: result.0.replace(" ", "_"),
152                             })
153                         }
154                         page_data.insert("title", "Results");
155                         page_data.insert("search", &data.search);
156                     }
⋮
175     page_data.insert("results", &results);
176     let rendered = tera.render("result.html", &page_data).
           unwrap();
177     HttpResponse::Ok().body(rendered)
178 }
```

Source: main.rs

More specifications about the build and display of the website can be found in the appendix in the HTML files.

3.5 Error Handling

To prevent a simple input error or faulty communication between the SQL server, script, or website from crashing the software immediately, instead of displaying clarifying error messages, proper error handling must be implemented. This is not only relevant for the finished prototype and thus a reasonable user experience, but also helpful during development to detect and fix errors early on.

To handle errors caused by incorrect user input in the search field, the error token has already been introduced in the lexer (code 1, 78-80). In the parser, grammar errors must be handled. For this purpose, several custom error types are defined with the help of the crate thiserror. To do this, the derive statement must be passed Error (cf. TOLNAY 2019, n.p.), as in the case of the parser in code listing 29 line 367.

These errors are then used in all parse functions, such as parse_statement and parse_weighted, by returning a datatype called Result (code 29, 119+327). Every time

an error case is encountered in the code, the corresponding error message is returned (code 29, 143+351). This procedure using results as datatype allows the use of a '?' operator when calling the function to forward an error in case one is returned (code 29, 141).

Code Listing 29: Error handling in parser

```
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             },
143             _ => return Err(ParseError::UnexpectedToken(self.
                    current.clone())),
144         };
⋮
327     fn parse_weighted(&mut self) -> Result<Vec<(Expression,
            Expression)>, ParseError> {
⋮
350         if sum_weights != 1.0 {
351             return Err(ParseError::WeightError(sum_weights));
352         }
⋮
366 // Types of errors covered by the parser
367 #[derive(Debug, Error)]
368 pub enum ParseError {
369     #[error("Unexpected token {0:?}.")]
370     UnexpectedToken(Token),
371     #[error("Entered unreachable code.")]
372     Unreachable,
373     #[error("Weights do not add up to 1.0. Sum of all weights:
            {0}")]
374     WeightError(f64),
375 }
```

Source: parser.rs

A similar approach can be found in the generator.
When the code generator is executed, it checks after each step to see if an error was thrown. If so, an error is also thrown and the error message is passed. In the result function of the website, where the code generator is executed, a check is made whether an error was thrown (code 30, 167-173). In this case, the website is not passed a list of results, but an error message to display.

Code Listing 30: Error display

```
134 async fn result(tera: web::Data<Tera>, data: web::Form<Search>)
        -> impl Responder {
135     let mut page_data = Context::new();
136     let mut results: Vec<Result> = Vec::new();
137     // Run code generator with the string from the search field
138     match run_code_gen(data.search.clone(), PATH_SQL) {
```

⋮

```
166         // If code generator returns error, display error
               instead of search results
167         Err(error) => {
168             page_data.insert("title", "Error");
169             page_data.insert(
170                 "search",
171                 &format!("{} threw an error: {}", &data.search,
                        &error.to_string()),
172             );
173         }
```

Source: main.rs

There are many other small prompts to catch errors, but the above examples are the most important elements of the code generator itself.

4 Demonstration

In this chapter, the prototype is shown demonstratively to highlight the purpose and workings of the code generator. By adjusting the constants in the generator.rs file, it is possible to apply the generator to several similarly constructed databases. The test data in this demonstration is the same database as in the development and testing of the prototype. This is a formatted version of an official Wikipedia dump from September 20th, 2022 (see WIKIMEDIA 2022, n.p.).

4.1 Database Preparation

The Wikipedia dump is provided in one or more compressed .bz2 files. With the help of an etree of lxml a parser can be built, with which the title and content of each entry can be extracted. Many of the entries are so-called redirects, which are irrelevant for a full-text search and are filtered out (code 31, 54). Otherwise, the title and content are written to a CSV file, and further, the longest title and text entries as well as the total amount of entries are being kept track of.

Code Listing 31: Wikipedia as CSV

```
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")):
```

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

Source: convert_wiki_to_csv.py

When running the script a total of 6,703,714 entries were written in a total runtime of 58 minutes and 14 seconds.

Based on the values of the longest title and text a table with a unique id is created. Since the CSV does not contain an id, a temporary view is created into which a bulk insert is executed. The path to the file has been redacted afterward.

Code Listing 32: Insert into SQL server

```
CREATE TABLE [dbo].[Article](
   [id] [int] IDENTITY(1, 1) NOT NULL,
   [Title] [varchar](255) NULL,
   [Text] [varchar](max) NULL
);
```

```
CREATE VIEW [dbo].[Article_insert] AS
SELECT [Title],
    [Text]
FROM [dbo].[Article];
```

```
BULK
INSERT [dbo].[Article_insert]
FROM 'path redacted' WITH (
             FIRSTROW = 1,
             FIELDQUOTE = '\'
      , FIELDTERMINATOR = ','
      , ROWTERMINATOR = '\n',
```

```
8                TABLOCK);
```

Source: create_article, create_view, bulk_insert

Using this SQL, all entries were successfully loaded within 41 minutes and 57 seconds. The graphical interface of the Microsoft SQL Server Management Tool then enabled an effortless full-text index creation.

4.2 Prototype Utilization

To use the prototype the SQL server must be started and the website including the code generator is started by running the command `cargo run` in the project folder. On localhost:8080 a minimal website with a search field is now visible, where you can enter any request in the custom query language.

Figure 4: Search field

Source: Own prototype

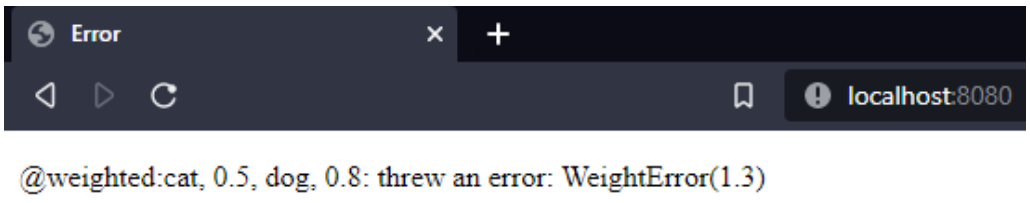
For example, if you enter the command `@near:space, dog, 10:` you will be redirected to a result page where either the search results or an error message will be displayed. In this case, three matching results are displayed sorted by rank. For each result, the title and rank are displayed and the title can be clicked on to be redirected to the relevant Wikipedia article.

Figure 5: Results for space dogs

Source: Own prototype

The actual generated SQL can be looked up in the prototype files if needed. If an incorrect command is entered, the error message thrown in the generator is displayed. For example, the command `@weighted:cat, 0.5, dog, 0.8:` should not be processed as the weights do not add up to exactly 1.

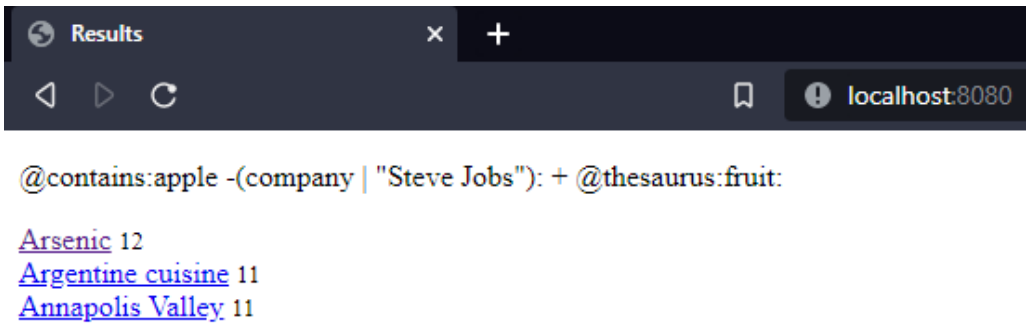
Figure 6: Weight error



Source: Own prototype

For demonstration purposes, a complicated search query with multiple functions and nested search terms can also be submitted. A search for apple but without the presence of the terms company or Steve Jobs in combination with a term that has the same meaning as fruit can be formulated in the query language as `@contains:apple -(company | "Steve Jobs"): + @thesaurus:fruit:`.

Figure 7: Results for fruit apples



Source: Own prototype

The first result 'Arsenic' describes a chemical element that is listed because of several sentences in the article mentioning its occurrence in apple juice and other fruit juices. Note that it may not be the very best result expected as the test data does not seem to contain every single Wikipedia article, only a large amount.

5 Conclusion

5.1 Summary

The implementation can be summed up in a technical summary, highlighting milestones and their placement in the overall project.
First, the language was defined and the rough structure of the language was discussed. The query language should consist of functions starting with '@', which contain parameters. Afterward, it was defined which functions should be implemented, and which parameters are needed or allowed. The finished syntax definition was then written down in EBNF.
The core of the prototype, the code generator itself, was divided into three modules: lexer, parser, and generator. The lexer analyzes the input string and cuts one or more characters into accepted tokens of the query language. The definition of these acceptable tokens is done by literal strings and regular expressions, whereas whitespaces are ignored by the lexer.
The list of tokens is passed to the parser, which goes linearly through the entire list and tries to verify the logical order and convert it into an ordered structure, the AST. Statements are interpreted as functions, which contain expressions as search parameters. Special cases are operators, which are more complex and therefore treated in more detail in chapter 3.2.3. The use of operators requires the introduction of precedence, to build hierarchical structures, despite the linear processing of tokens. Thus also the final elements of the parser were implemented.
The final TSQL is then created from the AST using the generator. Parts of the TSQL are preformulated by constants and the built-in function CONTAINSTABLE is used. The specifications of this function are filled with the contents of the AST. Each of the implemented functions is translated to TSQL in its way. In addition, the correct notation of operators is taken care of as well. All parts are combined into one long string and output as the result of the generator, and thus the whole code generator.
As part of the prototype, a website was also built as a user interface, where the user enters the query language request and results are displayed. In addition, other implemented functions execute the generated TSQL and parse the results of the SQL server.

5.2 Discussion

- What research question or problem was addressed?
- What type of research was done?

- How were data collected and analyzed?
- What were the most important findings?
- What is the overall answer to the research question?
- What are the implications of the results?
- Are there any important limitations?
- Are there any key recommendations?

Appendix

Appendix 1: Cargo.toml

```
[package]
name = "fulltext_search_code_gen"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.
   org/cargo/reference/manifest.html

[dependencies]
logos = "0.12.1"
thiserror = "1"
regex = "1"
actix-web = "3"
tera = "1.17.0"
serde = "1"
```

Appendix 2: 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'
```

```rust
#[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);
    }
```

```
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
        datatypes
114 #[derive(Deserialize)]
115 struct Search {
116     search: String,
117 }
118 #[derive(Serialize)]
119 struct Result {
120     title: String,
121     rank: u64,
122     link: String,
123 }
124
125 // Define functional parts of the search page
126 async fn search(tera: web::Data<Tera>) -> impl Responder {
```

```
127     let mut data = Context::new();
128     data.insert("title", "Search field");
129     let rendered = tera.render("search.html", &data).unwrap();
130     HttpResponse::Ok().body(rendered)
131 }
132
133 // Define functional parts of the result page
134 async fn result(tera: web::Data<Tera>, data: web::Form<Search>)
        -> impl Responder {
135     let mut page_data = Context::new();
136     let mut results: Vec<Result> = Vec::new();
137     // Run code generator with the string from the search field
138     match run_code_gen(data.search.clone(), PATH_SQL) {
139         // If code generator returns no error execute SQL and
              read the results
140         Ok(_) => {
141             execute_sql(PATH_SQL, PATH_RESULTS);
142             let results_vec = read_results(PATH_RESULTS);
143             // Fit search results into Result struct to
                  properly display on the page, otherwise diplay
                  error
144             match results_vec {
145                 Some(results_vec) => {
146                     for result in results_vec {
147                         results.push(Result {
148                             title: result.0.clone(),
149                             rank: result.1,
150                             // link to the Wikipedia article is
                                  also provided, whitespaces need
                                  to be replaced
151                             link: result.0.replace(" ", "_"),
152                         })
153                     }
154                     page_data.insert("title", "Results");
155                     page_data.insert("search", &data.search);
156                 }
157                 None => {
158                     page_data.insert("title", "Error");
159                     page_data.insert(
160                         "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 3: 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)
}
```

```
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-ZÃ?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 4: parser.rs

```rust
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 {
```

```
37      fn token(token: Token) -> Self {
38          match token {
39              Token::Bang | Token::Minus => Self::Not,
40              Token::Plus | Token::And | Token::WordOrPhrase(..)
                    => Self::And,
41              Token::Or => Self::Or,
42              Token::LeftParen => Self::Group,
43              Token::Contains
44              | Token::Starts
45              | Token::Inflection
46              | Token::Thesaurus
47              | Token::Near
48              | Token::Weighted => Self::Statement,
49              _ => Self::Lowest,
50          }
51      }
52  }
53
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      }
70
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      }
89
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())
102         } else {
103             Err(ParseError::UnexpectedToken(self.current.clone
                    ()))
104         }
105     }
106
107     // Current token should match the one given and read to
            next token
108     // Input: token
109     // Output: token or error
```

```
110     fn expect_token_and_read(&mut self, token: Token) -> Result
            <Token, ParseError> {
111         let result = self.expect_token(token)?;
112         self.read();
113         Ok(result)
114     }
115
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() {
121             Token::Contains => Statement::Contains {
122                 expression: self.parse_contains()?,
123             },
124             Token::Starts => Statement::Starts {
125                 expression: self.parse_starts()?,
126             },
127             Token::Inflection => Statement::Inflection {
128                 expression: self.parse_inflection()?,
129             },
130             Token::Thesaurus => Statement::Thesaurus {
131                 expression: self.parse_thesaurus()?,
132             },
133             Token::Near => {
134                 let (parameter, proximity) = self.parse_near()
                        ?;
135                 Statement::Near {
136                     parameter,
137                     proximity,
138                 }
139             }
140             Token::Weighted => Statement::Weighted {
141                 parameter: self.parse_weighted()?,
142             },
143             _ => return Err(ParseError::UnexpectedToken(self.
                    current.clone())),
144         };
```

```
145         // After a function could be an infix operator
146         while !self.current_is(Token::EoF) && precedence <
               Precedence::token(self.current.clone()) {
147             if let Some(in_statement) = self.
                   parse_infix_statement(statement.clone())? {
148                 statement = in_statement
149             } else {
150                 break;
151             }
152         }
153         Ok(statement)
154     }
155 
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             }
163             Token::Number(u) => {
164                 self.expect_token_and_read(Token::Number(0))?;
165                 Expression::Number(u)
166             }
167             Token::ZeroToOne(f) => {
168                 self.expect_token_and_read(Token::ZeroToOne
                       (0.0))?;
169                 Expression::ZeroToOne(f)
170             }
171             t @ Token::Minus | t @ Token::Bang => {
172                 self.expect_token_and_read(t.clone())?;
173                 Expression::Prefix(
174                     Operator::token(t),
175                     Box::new(self.parse_expression(Precedence::
                           Prefix)?),
176                 )
177             }
```

```
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             }
186             _ => return Err(ParseError::UnexpectedToken(self.
                    current.clone())),
187         };
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         }
198         Ok(expr)
199     }
200
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,
217         })
218     }
219
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();
228                 self.read();
229                 let sec_expr = self.parse_expression(Precedence
                        ::token(token.clone()))?;
230                 Some(Expression::Infix(
231                     Box::new(expr),
232                     Operator::token(token),
233                     Box::new(sec_expr),
234                 ))
235             }
236             _ => None,
237         })
238     }
239
240     // Infix operators AND and OR expect a statement on either
        side
241     fn parse_infix_statement(
242         &mut self,
```

```
243             statement: Statement,
244         ) -> Result<Option<Statement>, ParseError> {
245             Ok(match self.current {
246                 Token::Plus | Token::And | Token::Or => {
247                     let token = self.current.clone();
248                     self.read();
249                     let second_statement = self.parse_statement(
                            Precedence::token(token.clone()))?;
250                     Some(Statement::Infix {
251                         statement: Box::new(statement),
252                         operator: Operator::token(token),
253                         second_statement: Box::new(second_statement
                                ),
254                     })
255                 }
256                 _ => None,
257             })
258         }
259
260         // Functions all have a similar strucure needing colons to
            surround their parameters
261
262         // Contains function only expects one word or phrase or
            combination of expressions
263         fn parse_contains(&mut self) -> Result<Expression,
            ParseError> {
264             self.expect_token_and_read(Token::Contains)?;
265             self.expect_token_and_read(Token::Colon)?;
266             let expression: Expression = self.parse_expression(
                    Precedence::Statement)?;
267             self.expect_token_and_read(Token::Colon)?;
268             Ok(expression)
269         }
270
271         // Startswith function only expects one one word or phrase
            or combination of expressions
272         fn parse_starts(&mut self) -> Result<Expression, ParseError
            > {
273             self.expect_token_and_read(Token::Starts)?;
274             self.expect_token_and_read(Token::Colon)?;
```

```
275             let expression: Expression = self.parse_expression(
                    Precedence::Statement)?;
276             self.expect_token_and_read(Token::Colon)?;
277             Ok(expression)
278         }
279
280         // Inflection function only expects one one word or phrase
                or combination of expressions
281         fn parse_inflection(&mut self) -> Result<Expression,
                ParseError> {
282             self.expect_token_and_read(Token::Inflection)?;
283             self.expect_token_and_read(Token::Colon)?;
284             let expression: Expression = self.parse_expression(
                    Precedence::Statement)?;
285             self.expect_token_and_read(Token::Colon)?;
286             Ok(expression)
287         }
288
289         // Thesaurus function only expects one one word or phrase
                or combination of expressions
290         fn parse_thesaurus(&mut self) -> Result<Expression,
                ParseError> {
291             self.expect_token_and_read(Token::Thesaurus)?;
292             self.expect_token_and_read(Token::Colon)?;
293             let expression: Expression = self.parse_expression(
                    Precedence::Statement)?;
294             self.expect_token_and_read(Token::Colon)?;
295             Ok(expression)
296         }
297
298         // Near function expects multiple comma-seperated words or
                phrases with an optional number as the last parameter
299         fn parse_near(&mut self) -> Result<(Vec<Expression>,
                Expression), ParseError> {
300             self.expect_token_and_read(Token::Near)?;
301             self.expect_token_and_read(Token::Colon)?;
302             let mut parameter: Vec<Expression> = Vec::new();
303             // Proximity has a default value of 5 if no number is
                    given
304             let mut proximity = Expression::Number(5);
```

```
305         while !self.current_is(Token::Colon) {
306             if self.current_is(Token::Comma) {
307                 self.expect_token_and_read(Token::Comma)?;
308             }
309             match self.parse_expression(Precedence::Lowest)? {
310                 Expression::WordOrPhrase(s) => parameter.push(
                       Expression::WordOrPhrase(s)),
311                 Expression::Number(u) => {
312                     if self.current_is(Token::Colon) {
313                         proximity = Expression::Number(u)
314                     } else {
315                         return Err(ParseError::UnexpectedToken(
                               self.current.clone()));
316                     }
317                 }
318                 _ => return Err(ParseError::UnexpectedToken(
                       self.current.clone())),
319             }
320         }
321         self.expect_token_and_read(Token::Colon)?;
322         Ok((parameter, proximity))
323     }
324
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)?;
341             let weight = match self.parse_expression(Precedence
                    ::Lowest)? {
342                 Expression::ZeroToOne(f) => {
343                     sum_weights += f;
344                     Expression::ZeroToOne(f)
345                 }
346                 _ => return Err(ParseError::UnexpectedToken(
                        self.current.clone())),
347             };
348             parameter.push((expression, weight));
349         }
350         if sum_weights != 1.0 {
351             return Err(ParseError::WeightError(sum_weights));
352         }
353         self.expect_token_and_read(Token::Colon)?;
354         Ok(parameter)
355     }
356
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     }
364 }
365
366 // Types of errors covered by the parser
367 #[derive(Debug, Error)]
368 pub enum ParseError {
369     #[error("Unexpected token {0:?}.")]
```

```
370     UnexpectedToken(Token),
371     #[error("Entered unreachable code.")]
372     Unreachable,
373     #[error("Weights do not add up to 1.0. Sum of all weights:
            {0}")]
374     WeightError(f64),
375 }
```

Appendix 5: 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 {
```

```
        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 6: generator.rs

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

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

```

```rust
// Database constants
const DB_NAME: &str = "Wikipedia";
const TBL_NAME: &str = "[dbo].[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,
}

```

```
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         if self.current == Statement::EoF {
53             return Ok(None);
54         }
55         Ok(Some(self.generate_statement(self.current.clone())?)
               )
56     }
57 
58     // Set current and peek one step further in the ast
59     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                  sql_parts.join(" ")
84              }
85              // Contains generates it's search condition without
                    mutation
86              Statement::Contains { expression } => {
87                  format!("{}", self.generate_expression(
                        expression)?)
88              }
89              // Startswith adds a * to end of a word or before
                    the last " in a phrase
90              Statement::Starts { expression } => {
91                  let mut word_or_phrase = self.
                        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(
                            expression)?;
126                     sql_parts.push(format!("{}", string));
127                     sql_parts.push(String::from(", "));
128                 }
129                 sql_parts.remove(sql_parts.len() - 1);
130                 sql_parts.push(format!("), {})", self.
                        generate_expression(proximity)?));
131                 sql_parts.join("")
132             }
```

```
133             // Weighted generates tuples of search criteria and
                    their respective weight
134             Statement::Weighted { parameter } => {
135                 let mut sql_parts: Vec<String> = Vec::new();
136                 sql_parts.push(format!("ISABOUT("));
137                 for (word_or_phrase_expr, weight_expr) in
                        parameter {
138                     let word_or_phrase = self.
                            generate_expression(word_or_phrase_expr)
                            ?;
139                     let weight = self.generate_expression(
                            weight_expr)?;
140                     sql_parts.push(format!("{} WEIGHT({})",
                            word_or_phrase, weight));
141                     sql_parts.push(String::from(", "));
142                 }
143                 sql_parts.remove(sql_parts.len() - 1);
144                 sql_parts.push(String::from(")"));
145                 sql_parts.join("")
146             }
147             _ => return Err(GenerateError::UnexpectedStatement(
                    self.current.clone())),
148         };
149         self.write();
150         Ok(sql)
151     }
152
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(),
161             // Infix operator enclose their expressions with
                    parentheses to ensure precedence
162             Expression::Infix(expr1, operator, expr2) => {
```

```
163                 let mut sql_parts = [
164                     String::from("("),
165                     self.generate_expression(*expr1)?,
166                     String::from(")"),
167                     self.generate_operator(operator)?,
168                     String::from("("),
169                     self.generate_expression(*expr2.clone())?,
170                     String::from(")"),
171                 ];
172                 // If the second expression is a not operator
                       it must write NOT before the parenthesis
173                 match *expr2 {
174                     Expression::Prefix(Operator::Not, ..) =>
                           sql_parts[4] = String::from("NOT ("),
175                     _ => (),
176                 }
177                 sql_parts.join(" ")
178             }
179             Expression::Prefix(operator, expr) => {
180                 let sql_parts = [
181                     self.generate_operator(operator)?,
182                     self.generate_expression(*expr)?,
183                 ];
184                 sql_parts.join(" ")
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
                   generate_expression for infix
198             Operator::Not => "",
```

```
199             };
200             Ok(op.to_owned())
201         }
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 7: mod.rs

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

Appendix 8: 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 9: search.html

```
1 {% extends "base.html" %}
2
3 {% 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 10: 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 11: 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 = "path redacted"
FILENAME_ARTICLES = "articles.csv"

```

```
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
              =("end",)):
28             if element.tag.endswith("page"):
29                 namespace_tag = element.find(ns_token)
30
31                 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:
```

```
        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 % 100000 == 0:
                        print("{:,}".format(totalCount))
    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 12: create_article.sql

```
CREATE TABLE [dbo].[Article](
```

```
    [id] [int] IDENTITY(1, 1) NOT NULL,
    [Title] [varchar](255) NULL,
    [Text] [varchar](max) NULL
);
```

Appendix 13: create_view.sql

```
CREATE VIEW [dbo].[Article_insert] AS
SELECT [Title],
    [Text]
FROM [dbo].[Article];
```

Appendix 14: bulk_insert.sql

```
BULK
INSERT [dbo].[Article_insert]
FROM 'path redacted' WITH (
             FIRSTROW = 1,
             FIELDQUOTE = '\'
      , FIELDTERMINATOR = ','
      , ROWTERMINATOR = '\n',
          TABLOCK);
```

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. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/367236.367262

CLEAVELAND, Craig: *Program generators with XML and Java*. The Charles F. Goldfarb series on open information management. Upper Saddle River, NJ: Prentice Hall PTR, 2001. ISBN: 978-0-13-025878-6

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

CZARNECKI, Krzysztof; EISENECKER, Ulrich: *Generative programming: methods, tools, and applications*. Boston: Addison Wesley, 2000. ISBN: 978-0-201-30977-5

EDE, Rob: *Actix Web: A powerful, pragmatic, and extremely fast web framework for Rust*. Jan. 2022. URL: https://github.com/actix/actix-web (visited on 09/14/2022)

FARRELL, James Alan: Compiler Basics. en. In: (Aug. 1995). 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

HIRSZ, Maciej: *Logos: Create ridiculously fast Lexers*. June 2022. URL: https://github.com/maciejhirsz/logos (visited on 09/05/2022)

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

ISO/IEC 14977:1996(E): Information Technology - Syntactic Metalanguage - Extended BNF. In: (1996)

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. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/1118890.1118892

NYSTROM, Robert: *Crafting interpreters*. eng. Daryaganj Delhi: Genever Benning, 2021. ISBN: 978-0-9905829-3-9

POMBERGER, Gustav; PREE, Wolfgang; STRITZINGER, Alois: Methoden und Werkzeuge für das Prototyping und ihre Integration. In: *Inform., Forsch. Entwickl.* 7 (1992) Nr. 2

PROUILLET, Vincent: *Tera: A template engine for Rust based on Jinja2/Django*. Aug. 2022. URL: https://github.com/Keats/tera (visited on 09/14/2022)

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. ISBN: 978-0-12-745040-7. DOI: 10.1145/800025.1198374

SARKAR, Soumen; CLEAVELAND, Craig: Code generation using XML based document transformation. In: *Published on The Server Side–Your J2EE Community* (2001)

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

TOLNAY, David: *Thiserror: For struct and enum error types*. Oct. 2019. URL: https://github.com/dtolnay/thiserror (visited on 09/17/2022)

TOLNAY, David; TRYZELAAR, Erick: *Serde: Serialization framework for Rust*. Apr. 2017. URL: https://github.com/serde-rs/serde (visited on 09/15/2022)

WIKIMEDIA: *enwiki dump progress on 20220920*. Sept. 2022. URL: https : / / dumps . wikimedia.org/enwiki/20220920/ (visited on 10/19/2022)

WIRTH, Niklaus: *Compiler construction*. eng. International computer science series. Harlow, England ; Reading, Mass: Addison-Wesley Pub. Co, 1996. ISBN: 978-0-201-40353-4

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. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/359863.359883

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, 28.10.2022

(Location, Date)

(handwritten signature)