

SQLSQLLSQ

String eQuality Logic for SQL-Like String Queries



April 1, 2020

Liam George

Loughborough University

SQLSQLLSQ - String eQuality Logic for SQL-like String Queries

Two approaches to evaluating FOL string queries.

Table of Contents

[Project Description 2](#_Toc37708324)

[Theory 3](#_Toc37708325)

[Planning 4](#_Toc37708326)

[Progress/Implementation 5](#_Toc37708327)

[Substring Enumerator 5](#_Toc37708328)

[Parser 5](#_Toc37708329)

[Evaluator 6](#_Toc37708330)

[Term Evaluation 6](#_Toc37708331)

[Top Down Approach 6](#_Toc37708332)

[Bottom Up Approach 7](#_Toc37708333)

[References 8](#_Toc37708334)

[Parsing: 8](#_Toc37708335)

[Data Structures: 8](#_Toc37708336)

TODO:

*Project Description*

Basic Theory

Proofs

Planning

Implementation

Testing

Evaluation

References

# Project Description

*“The project, titled SQLSQLLSQ (String eQuality Logic in SQL-Like String Queries), is an investigation into a method of querying strings that’s more powerful than Regular Expressions. The aim is to emulate some of the more powerful features of SQL querying on large bodied strings, including wildcards and patterns (as in those described in pattern languages - essentially variables, or consistent wildcards) to allow for more effective searching of large documents. This would be done through a variant on First-Order Logic.”*

*-Project Brief submission.*

In all, the project intends to develop a more powerful string querying method than Regular Expressions by modelling powerful features from SQL. This will be done using a purpose crafted language extremely similar to Propositional Logic formulae.

The scope of the project is thus;

-To create and implement a language capable of modelling propositions about the composition of strings  
-To create a parser for said language  
-To create an evaluator for said parser’s result  
-To optimise said evaluator to a reasonable extent.

All of this should be done with resource efficiency in mind, and optimisations being aimed at longer strings.

# Theory

The usage of Propositional (First-Order) Logic as a base for the model is by design, as it is tailored well for computation and has a system in place to model those more powerful functions we intend to replicate from SQL. Usage of existential quantifiers will substitute well for wildcards, giving users the ability to craft more powerful queries with a higher degree of ease.

Propositional Logic formulae may be easily and perfectly represented through parse trees, lending itself to a model well-tailored for computational evaluation.

I have read into some data structures, as well as Parse Trees and Parser Generators in preparation for various parts of the project.

As the number of substrings (or, more formally, ordered sets of *consecutive* characters within a given string, not to be confused with any selection of ordered characters – this clarification has proven necessary in discussion to distinguish between sets of in-order characters from the string and sets of consecutive characters from the string during research) is proportional to the square of the string’s length (which was determined during discussions with my tutor, though a simple proof follows), it occurred to me that usage of a storage method that lends itself to log(n) search times would negate the polynomial growth of expenses. This is because the insertion of n2 items into a data structure whose insertion cost is, on average, belonging to the order of *O*(log(n)) would have a total cost belonging to the order O(log(p(n))), where p(n) is some polynomial function on n. Since no polynomial function on n is not exceeded by some exponent q of n, it follows from the fact that log(nm) ≡ m log(n) that the cost of assembling said data structure would belong to the order O(log(n)). This means that most of the cost of enumerating the substrings of the body string will come from the enumeration itself, not the storing of the results. As said storing is algorithmically similar to (and indeed, often a mere extension of the algorithm for) searching for an input in many data structures, it will be easy to streamline the two together in a way that helps to avoid repeated substrings, by including no substrings of a string already present in the structure.

It remains to prove the completeness of the substring enumeration algorithm mentioned in the Planning chapter, as well as to prove the completeness of both the top-down and bottom-up evaluators.

# Planning

As mentioned in the project description, the scope of the project is thus;

-To create and implement a language capable of modelling propositions about the composition of strings  
-To create a parser for said language  
-To create an evaluator for said parser’s result  
-To optimise said evaluator to a reasonable extent.

Creating such a language should prove easy – simply taking the form of Propositional Formulae and tweaking it for easier computation by inserting tokens to separate different parts of the statement. It follows from a discussion with my tutor that Prenex Normal Form is an appropriate standard to aide in ease of computation, but could lead to reckless expense of computational resources. Despite this drawback, I consider it to be a satisfactory starting point for now.

During the preliminary phases of planning the project I developed an algorithm that efficiently enumerates all sets of consecutive characters within a given string (consecutive substrings), which would be used to define the set of possible strings that any variables within queries could take.

The following is the plan presented in the project brief:

-Build formula representation/model  
-Write text-to-model formula parser  
-Write substring enumerator (with formal proof)  
-Write top-down evaluator (with formal proof)  
-Write bottom-up evaluator (with formal proof)  
-Compare top-down and bottom-up evaluators’ runtime and space efficiencies in different scenarios  
-Have final evaluator run on user-entered queries  
-Final functional testing and code clean-up

Thus far, the first 3 of these 8 items have been completed, albeit with no formal proof for the substring enumerator. Though I would be satisfied if, by the final deadline, I had only succeeded in what has been done thus far as well as having written formal proofs for and implementations of the top-down evaluator and bottom-up evaluator and having them run on user-entered queries, I am optimistic that I can fully complete the project within the allotted timespan.

# Progress/Implementation

## Substring Enumerator

I began by coding the substring enumeration algorithm, since it seemed to be an easy way to make a dent in the project. This progressed quickly and easily, with only minor errors which were easily and quickly fixed.

Said enumerator works by considering the string as a whole, before dismissing either the leftmost or the rightmost character. If it dismisses the leftmost character, it does so repeatedly until no characters remain. If it dismisses the rightmost character, it repeats the algorithm recursively, treating the new truncated string as it did the original. Finally, it considers the empty string.

Currently, these results are stored in a list, since these are effective for the purpose of testing the algorithm. I intend to change this so results are stored in a tree, since it would be possible to make the enumeration algorithm give a maximally succinct result faster than using a list and searching it for repetitions, wherever any part of the string to be searched is repeated (such as in the string “abcdefbcdgh”, in which the phrase “bcd” is repeated despite the fact that it and its substrings need only be considered once).

## Parser

I have decided upon a good starting model for a language for queries – essentially the form of Propositional Formulae with some minor tweaks for ease of parsing, including the limitation that all formulae should be in Prenex normal form, which separates existential quantifiers from the rest of the statement. This restriction should be easy enough to lift retroactively by widening the recursion of the parsing algorithm to include the section which parses quantifiers, but is thus far a satisfactory start.

Though I attempted to utilise a parser generator to ease writing a parser and modifying its grammar, I found this to be unfortunately difficult since most parser generators I found met my needs well were difficult to utilise within c# - the language I’m writing the project in. After some deliberation, I erred on the side of caution with the January deadline in mind, and changed plans to write a parser manually instead. Though this proved time consuming, in this case clearer and more definitive progress proved a harbinger of success.

With control of the language used to specify the queries, I could require useful marker characters to denote the edges of various parts of the code, most useful in parsing terms. Due to the Prenex normal form, I found I was able to split the queries into three parts – the signature, the quantifiers and the proposition – which helped chunk the problem, and I decided that the representation of the queries would reflect this until such a time that the Prenex normal form restriction was removed.

Later, I would come back to have the parser decompose the

## Evaluator

### Term Evaluation

### Top Down Approach

I proceeded to work on the top-down evaluator, beginning without regard to free variables, with an intent to build them in later. The evaluator would take a quantified expression and consider the quantifier used and the variable quantified. It would loop through all possible values for the variable, as determined by the subword generator, stopping when it found one that gives an answer to the quantifier (that is, a value that satisfies the quantified expression if the quantifier is “There Exists”, but is a value that does not satisfy the quantified expression if the quantifier is “For All”), and returns the appropriate truth value. If none are found, then the opposite truth value is returned.

Propositional logic operators were restricted to \* (representing logical AND) and + (representing logical OR) whose symbols were decided on with regards to softly typed languages’ approach to mathematical operations applied to Booleans and Boolean evaluation of integers. The evaluator simply considered the left operand and the right operand separately and returned the logical operator’s result when applied to the two.

Terms were evaluated using the Term Evaluator, substituting all the variables with their values as provided by the top down evaluator’s enumeration through the subwords of the body of text to be searched, both for quantified and free variables in their own manners. A variety of functions were included for a term to provide, including reference to a text’s length using mathematical comparators, and reference to its content using a double equals sign (==). To aide with the ability for the user to build a fixed string into the query and the use of special characters in them, the terms were later divided into atoms, which involved developing a method of escaping special characters and finding the first “unquoted” instance of particular characters in strings with escaped quotes being taken into consideration.

### Bottom Up Approach

After some initial testing to make sure the top-down quantifier was working in at least the basic use cases, I worked on the bottom-up quantifier. This would work by exploring the parse tree with a depth-first approach and evaluating terms as though their variables were all free (with the exception of the variable representing the body of text to be searched, denoted S), calling the top-down evaluator’s method to do so, and returning the resulting set of solutions.

*Such is the current state of the project – universe defined and query internalised, with operator precedence considered - what stands left to implement is an algorithm by which both free and quantified variables have their possible values considered, such that the proposition may be evaluated and the free variables’ tuples of proposition-validating values returned, should they be requested in the query. Once such an algorithm is in place, and is shown to be realistically efficient, the project shall be complete.*

# References

## Parsing:

[HasaniH](https://stackoverflow.com/users/7141/hasanih), [JasonMArcher](https://stackoverflow.com/users/64046/jasonmarcher), 2015, [What is a good C# compiler-compiler/parser generator? [closed]](https://stackoverflow.com/questions/1194584/what-is-a-good-c-sharp-compiler-compiler-parser-generator)**,** Stack Overflow, **<**<https://stackoverflow.com/questions/1194584/what-is-a-good-c-sharp-compiler-compiler-parser-generator>>  
7.6. Parse Tree, Runestone Academy, <<https://runestone.academy/runestone/books/published/pythonds/Trees/ParseTree.html>>  
Federico Tomassetti, 2017, The ANTLR Mega Tutorial, Amazon, <<https://convertkit.s3.amazonaws.com/landing_pages/incentives/000/375/965/original/ANTLR_Mega_Tutorial.pdf?1566483883>>  
Federico Tomassetti, 2017, The ANTLR Mega Tutorial, tomassetti.me, <https://tomassetti.me/antlr-mega-tutorial/>  
**Terence Parr**, 2014, Antlr, Antlr, <<https://www.antlr.org/>>

## Data Structures:

Tomek Tarczynski, 2017, Heap class in .NET [duplicate], Stack Overflow, <<https://stackoverflow.com/questions/2231796/heap-class-in-net>>  
[Matthew Cochran](https://www.c-sharpcorner.com/members/matthew-cochran), 2019, C# Heap(ing) Vs Stack(ing) In .NET - Part One, C# Corner, <<https://www.c-sharpcorner.com/article/C-Sharp-heaping-vs-stacking-in-net-part-i/>>  
2019, Binary heap, Wikipedia <<https://en.wikipedia.org/wiki/Binary_heap>>  
2019, Binary search tree, Wikipedia, <<https://en.wikipedia.org/wiki/Binary_search_tree>>  
2019, Heap (data structure) , Wikipedia, <<https://en.wikipedia.org/wiki/Heap_(data_structure)>>  
2019, Hash table, Wikipedia, <<https://en.wikipedia.org/wiki/Hash_table>>  
2019, B-tree, Wikipedia, <<https://en.wikipedia.org/wiki/B-tree>>