

https://dspace.org/

A generic framework for the matching of similiar names.

Schay, Warren Marc.

2012-01-30
http://hdl.handle.net/10539/11201

Downloaded from a DSpace repository. https://dspace.org/

A GENERIC FRAMEWORK FOR THE MATCHING OF SIMILAR NAMES

Warren Marc Schay
Supervisor: Prof. B. Dwolatzky

A dissertation submitted to the Faculty of Engineering and the Built Environment, University of the Witwatersrand, in fulfilment of the requirements for the degree of Master of Science in Engineering

Johannesburg, 2011

Contents

DECLARATION **IV**

ABSTRACT **V**

ACKNOWLEDGEMENTS **VI**

TEST DATA PRIVACY ISSUES **VII**

LIST OF FIGURES **VIII**

LIST OF TABLES **IX**

GLOSSARY **X**

1 INTRODUCTION **1**

1.1 OVERVIEW 3

2 LITERATURE REVIEW **4**

2.1 CAUSES OF VARIATION IN WORDS 4
2.1.1 Causes of Variation within Names *6*
2.2 FUZZY MATCHING ALGORITHMS 8
2.2.1 Orthographic Models *10*
2.2.2 Phonetic Models *23*
2.2.3 Hybrid Techniques *32*
2.2.4 Probabilistic Models *34*
2.2.5 Commonality between Fuzzy Matching Algorithms *36*
2.3 PRE-SEARCH DATA PARTITIONING 37
2.3.1 First Letter *38*
2.3.2 Word Length *39*
2.3.3 Word Halves *39*
2.3.4 N-Grams *40*
2.3.5 Bloom Filter *40*
2.3.6 External Criteria *42*
2.4 FRAMEWORKS 42
2.4.1 Framework Fundamentals *42*
2.4.2 Advantages of a Framework *48*
2.4.3 Disadvantages of a Framework *48*
2.4.4 Fuzzy Matching Frameworks *50*

3 RESEARCH QUESTION AND METHODOLOGY EMPLOYED **51**

3.1 RESEARCH QUESTION 51
3.2 METHODOLOGY EMPLOYED 51

4 DESIGN REQUIREMENTS **53**

4.1 MULTIPLE RELATED SEARCHES 53
4.2 MULTIPLE FUZZY MATCHING ALGORITHMS 54
4.3 ABSTRACT THE FUZZY MATCHING PROCESS 55
4.4 USER DEFINED FUZZY MATCHING PROCESS 55
4.4.1 Exclusion List *55*
4.4.2 Substitution List *56*
4.5 EXTERNAL APPLICATION ABSTRACTION FROM THE FUZZY MATCHING PROCESS 56
4.6 PERFORMANCE REQUIREMENTS 56
4.6.1 High Speed *57*
4.6.2 Non-Excessive Memory and Processor Utilisation *57*

5 SOLUTION OVERVIEW 59

5.1 DESIGN DECISIONS 59
5.1.1 Performance Trade-off for Generic Behaviour 59
5.1.2 Greybox Framework 60
5.1.3 Interfaces 61
5.1.4 C# Language and .Net Framework 64
5.1.5 Dynamically Loaded DLL's 65
5.1.6 Use of a Relational Database 66
5.1.7 Maintain All Search Names within an Internal Database 67
5.1.8 Abstraction of the Database away from the Third Party Developer 68
5.2 DESIGN OVERVIEW 68
5.2.1 Name Pre-Processing 70
5.2.2 Database Searching 71
5.2.3 Match Scoring 72
5.2.4 Database Storage 75
5.3 INITIAL DESIGN (VERSION 1) 76
5.3.1 Database 76
5.3.2 Initialisation 81
5.3.3 Database Maintenance 81
5.3.4 Search Input 82
5.3.5 Search Initialisation 83
5.3.6 Pre-processing 85
5.3.7 Storage 87
5.3.8 Search 88
5.3.9 Word Set Component Scoring 92
5.3.10 Search Set Score Aggregation 94
5.3.11 Match Evaluation 94
5.3.12 Search Output 98
5.4 DESIGN STRENGTHS 99
5.4.1 Support of multiple Fuzzy Matching Paradigms 99
5.4.2 All logic is supplied by the Third Party Developer 99
5.4.3 Abstraction of the Fuzzy Matching Process 99
5.4.4 Runtime Implementation of Third Party Logic 100
5.4.5 Generic 101
5.4.6 Capable of Maintaining Relationships 101
5.4.7 Capable of Serving Multiple Applications 102
5.5 DESIGN SHORTCOMINGS 103
5.5.1 Expensive Database Search 103
5.5.2 Unable to Pre-Filter based upon Non-Name Related Requirements 103
5.5.3 Main Reliance on Triggers to update the BaseWord table 104
5.5.4 Caching of Rules Table 104
5.5.5 Unintuitive 104
5.5.6 Inflexible Interfaces 105
5.5.7 Latency due to the Loading of DLL's at Runtime 105
5.6 IMPROVEMENTS TO THE ORIGINAL DESIGN (VERSION 2) 106
5.6.1 Pre-Search Filter 106
5.6.2 Separation of the single Pre-Processing Component into Storage Pre-Processing and Search Pre-Processing Sub-Components 111
5.6.3 In Memory Database Caching 112
5.6.4 Delegate to Access Contents of the Rules Table 116
5.6.5 Pre-Search Filter In-Memory Database Caching 119

6 SOLUTION TESTING 123

6.1 TEST OBJECTIVES 123
6.2 TEST DATA 125
6.2.1 Test Data Set Properties. *127*
6.3 TEST CASES 127
6.3.1 Test Case 1 - Basic Framework Search *128*
6.3.2 Test Case 2 - Related Searches *128*
6.3.3 Test Case 3 - Flexibility *129*
6.3.4 Test Case 4 - Speed *129*
6.3.5 Test Case 5 - High-Load Testing *130*
6.4 TEST RESULTS AND DISCUSSION 131
6.4.1 Test Case 1 - Basic Framework Search *131*
6.4.2 Test Case 2 - Related Searches *132*
6.4.3 Test Case 3 – Flexibility *133*
6.4.4 Test Case 4 – Speed *135*
6.4.5 Test Case 5 - High-Load Testing *142*
6.5 SOLUTION DEPLOYMENT 144

7 ANALYSIS AND CONCLUSION 147

7.1 CRITICAL ANALYSIS 147
7.2 RECOMMENDATIONS AND FUTURE DEVELOPMENTS 149
7.3 CONCLUSION 152

REFERENCES 156

APPENDIX A EDIT DISTANCE VARIANTS 161

1 WEIGHTED EDIT DISTANCE 161
2 EDIT DISTANCE WITH UPPERBOUND AND CUT-OFF CRITERION 161
3 NORMALISED EDIT DISTANCE (NED) 162
4 DISCRETISED EDIT DISTANCE (DED) 163
5 EXPONENTIAL EDIT DISTANCE (EED) 163
6 EDIT DISTANCE WITH A TRIE 163
REFERENCES 166

APPENDIX B DATABASE MODEL 167

APPENDIX C SEARCH INPUT XML SCHEMA 168

APPENDIX D FRAMEWORK DEFINED INTERFACES FOR THIRD PARTY CODE 171

APPENDIX E SEARCH OUTPUT XML SCHEMA 175

APPENDIX F OUTWARD FACING PRE-FILTER FUNCTIONALITY CHANGES 178

APPENDIX G DELEGATE ACCESS TO THE CONTENTS OF THE RULES TABLE 180

APPENDIX H ANALYSIS OF TEST DATA SET 182

1 COMMON FIRST NAME PREFIXES 182
2 COMMON SURNAME PREFIXES 191
3 PHONETIC PROPERTIES OF TEST DATA SET 200

Declaration

I declare that this dissertation is my own unaided work. It is being submitted to the Degree of Master of Science to the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination to any other University. This work has been performed while employed by Synthesis Software Technologies (Pty) Ltd, who provided the necessary facilities.

_____

(Warren Marc Schay – 0303537F)

_____ day of _____ year _____

Abstract

Name matching is a common requirement in modern business systems, wherein fuzzy matching techniques are employed to overcome variations between names. The purpose of this dissertation was the development of a framework, which is capable of implementing various fuzzy matching algorithms, while abstracting the name matching process away from external business systems. Through a study of existing fuzzy matching algorithms and frameworks, several design requirements were identified; the maintaining of name relationships, non-algorithm specific logic, abstraction of the matching process, user configured matching logic, consistent external interface and performance considerations. The deployment to a production environment and a series of tests, demonstrated that the framework fulfilled all but one of its design requirements, as certain algorithm implementations yielded excessive search times. The cause and remedy of this shortcoming were identified. Finally, based on an evaluation of the design's strengths and weaknesses, recommendations for future developments were suggested.

Acknowledgements

The author would like to thank his employer Synthesis Software Technologies (especially Dr. Jack Cohen) for the financial, academic and technical support, without which the dissertation would not have been possible. The author would also like to acknowledge his supervisor Prof. Barry Dwolatzky from the School of Electrical and Information Engineering at the University of the Witwatersrand for the invaluable insight and guidance that allowed the submission of this dissertation to come to fruition. Additional thanks are extended to the author's parents and wife for all their support and encouragement throughout the course of this dissertation, despite at times it appearing to be unending.

Test Data Privacy Issues

The data used for the test cases was collected from three sources, namely:

- Who's Who of Southern Africa (Media24, 2009).
- 1990 US Census (U.S. Census Bureau, 2009).
- A scramble list of employee first names and surnames from a company.

Who's Who of Southern Africa consists of person information that is already contained within the public domain and / or information which people have agreed to be placed within the public domain. The terms of conditions of the website allows the website to distribute content without limitation (Media24, 2009). The US Census Bureau states that it is committed to the making of all documents on its internet server accessible to all (U.S. Census Bureau, 2009). Furthermore, since the site provides the frequency of occurrence of various first names and surnames and not the actual first name – surname pairs, it is assumed that no privacy issues were violated. The names utilised from the above mentioned company were provided to the author on condition that the first names and surname combinations were scrambled.

List of Figures

Figure 2.1: Fuzzy Matching Algorithms investigated within Literature Review 9
Figure 2.2: Guth Algorithm Comparison Steps (Lait & Randell, 1993) 10
Figure 2.3: Example of Guth Algorithm Name Matching (Lait & Randell, 1993) 11
Figure 2.4: Calculating the minimum edit distance between Filips and Phillips (Du, 2005) 15
Figure 2.5: Soundex Code Look-Up Table (Du, 2005) 24
Figure 2.6: Phonix Code Look Up Table (Du, 2005) 28
Figure 2.7: Editex Letter Groups (Zobel & Dart, 1996) 33
Figure 2.8: A search performed against a partition of the search database 38
Figure 2.9: A Fuzzy Matching Framework 50
Figure 4.1: Searching multiple related names 54
Figure 5.1: High Level Diagram depicting the Fuzzy Matching Process 69
Figure 5.2: High Level Diagram depicting the Upkeep of the Fuzzy Matching Database Process 70
Figure 5.3: Example of the Assembly configuration section in the Framework's Configuration file 85
Figure 5.4: Search Input for name John Mark Smith 90
Figure 5.5: Generated SQL query for search name John Mark Smith 91
Figure 6.1: Results of the profiling of the Framework Soundex Implementation 138
Figure 6.2: Results of the profiling of the Framework Levenshtein Distance Implementation 140
Figure 6.3: Results of the profiling of the Framework N-Gram Similarity Implementation 141
Figure A.1: A trie built using the names filips, fillips, phan, phillip and phillips (Du, 2005) 164
Figure A.2: Construction of the Dynamic Programming Matrices using the Trie 165
Figure B.1: Framework Database Model 167
Figure H.1: The hamming distance between nine on the most common 5 character first name prefixes and their corresponding names 190
Figure H.2: The hamming distance between nine on the most common 5 character surname prefixes and their corresponding names 199
Figure H.3: Three of the most commonly occurring Soundex codes, the corresponding first names and the number of occurrences of each first name 201
Figure H.4: Two of the most commonly occurring Soundex codes, the corresponding surnames and the number of occurrences of each first name 202

List of Tables

Table 5.1: Example Inputs and Output for Wildcard search SetDatabaseSearchCondition method 89
Table 5.2: List of Search Results input in Match Evaluation Component 96
Table 5.3: Ordered List of Search Results 96
Table 5.4: List of Match Scores input into the Third Party Developer method 97
Table 5.5: Index list of results returned by the third party developer method 97
Table 5.6: Search Result List return by the Match Evaluation Components 97
Table 6.1: Sub-Objective to high level objective mapping 125
Table 6.2: Test Data Set Properties 127
Table 6.3: Test Case 1 Results 131
Table 6.4: Test Case 2 Results 132
Table 6.5: First Name Search Results 132
Table 6.6: Surname Search Results 132
Table 6.7: Soundex Search Results 134
Table 6.8: Levenshtein Search Results 134
Table 6.9: N-Gram Similarity Search Results 135
Table 6.10: Soundex Search Times 136
Table 6.11: Levenshtein Search Times 136
Table 6.12: N-Gram Similarity Search Times 136
Table 6.13: Batch Process Details 142
Table 6.14: Results of Simultaneous Search Requests Test Case 144
Table H.1: Prefix length of 1 182
Table H.2: Prefix length of 2 182
Table H.3: Prefix length of 3 184
Table H.4: Prefix length of 4 184
Table H.5: Prefix length of 5 186
Table H.6: Prefix length of 6 186
Table H.7: Prefix length of 7 188
Table H.8: Prefix length of 8 188
Table H.9: Prefix length of 1 191
Table H.10: Prefix length of 2 191
Table H.11: Prefix length of 3 193
Table H.12: Prefix length of 4 193
Table H.13: Prefix length of 5 195
Table H.14: Prefix length of 6 195
Table H.15: Prefix length of 7 197
Table H.16: Prefix length of 8 197

Glossary

.Net	See .Net Framework.
.Net Framework	The .Net Framework is the Microsoft platform for the building of applications. It consists of a Common Language Runtime (CLR) that provides an abstraction layer over the operating system, base class libraries that provide pre-built code for common low-level programming tasks and various development frameworks and technologies (Microsoft, 2009).
API	An Application Programming Interface (API) "is a boundary across which application software uses facilities of programming languages to invoke services. These facilities may include procedures or operations, shared data objects and resolution of identifiers" (ISO/IEC JTC 1, 2007).
C	C is a general-purpose computer programming language developed between 1969 and 1973 (Giannini, 2004).
C++	C++ is a general-purpose programming language, which supports data abstraction and object-orientated and generic programming. It was developed by Bjarne Stroustrup as an enhancement to the C language and was originally named "C with Classes" (Stroustrup, 1997).
C#	C# is a modern, general-purpose, object-oriented programming language developed by Microsoft, which uses similar syntax to C and C++. The language has been designed with support for software engineering principles such as strong type checking,

	array bounds checking, detection of attempts to use uninitialized variables and automatic garbage collection (Ecma International, 2006).
CLR	The Common Language Runtime (CLR) is the runtime environment, in which all languages within the .Net framework run their code. This runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used (thereby providing the garbage collection capabilities of the .Net framework). The use of the common CLR allows the interoperability of all .Net languages (MSDN Library, 2011a).
DLL	"A dynamic-link library (DLL) is an executable file that acts as a shared library of functions... Dynamic linking differs from static linking in that it allows an executable module (either a .dll or .exe file) to include only the information needed at run time to locate the executable code for a DLL function. (MSDN Library, 2011b)"
Garbage Collection	Garbage Collection is the process in which an application's memory space is scanned for unused objects and their space reclaimed, thereby freeing the memory to be used by other objects. Furthermore, this process aids in the prevention of memory corruption (Bacon, 2007).
Java	The Java programming language is a general-purpose, concurrent, class-based, object-oriented language. Though related to C and C++, it is organised differently, with a number of aspects of C and C++ omitted and aspects from other

	languages included. The language is strongly typed and includes automatic garbage collection (Sun Microsystems, Inc, 2000).
JVM	The Java Virtual Machine (JVM) is an abstract computing machine, which executes instruction sets and manipulates various memory areas at run time. The particular instruction set executed by the JVM is called bytecode and is generated by the Java compiler. The JVM is the component of the Java technology responsible for the hardware- and operating system- independence of Java (Sun Microsystems, Inc, 1999).
MS SQL	Microsoft SQL Server (MS SQL) is a database management and analysis system produced by Microsoft (Microsoft, 2011).
Oracle	Oracle Database (Oracle) is an object-relational database management system produced by the Oracle Corporation (Oracle Corporation, 2010).
SQL	The Structured Query Language (SQL) is a standardised language for defining and manipulating data in a relational database (IBM, 2006).
Trigger	Database triggers are programs that implicitly start (are “fired” off) when an INSERT, UPDATE, or DELETE statement is executed on a table. A trigger can contain several SQL statements (SAP Library, 2009).
Python	Python is an interpreted, object-oriented, high-level programming language, which is often used for Rapid

	Application Development. In addition it is often used as a scripting or glue language to connect existing components together (Python Software Foundation, 2007).
Python Script	A Python script is a series of Python commands that are saved in a file and can be retrieved and executed at a later stage. (Python Software Foundation, 2004)
XML	The Extensible Markup Language (XML) is a set of rules for the encoding of documents for distribution (across the internet). The design goals of XML include simplicity, generality, usability and human readability (W3C, 2008).
XSD	The XML Schema Definition Language (XSD) is used to define the structure of XML documents (W3C, 2004).

1 Introduction

The need to perform name matching has become a common requirement in today's business systems. At the heart of name matching is the simple objective of determining whether two or more names are the same. Examples of such business processes are client / customer (entity) searches (i.e. whether the searched entity exists within a particular system), comparisons of two or more entities (i.e. to determine whether the two compared entities are the same person / company), the lookup of entities against external lists (i.e. to determine whether a particular person / company exists on a list of entities external to the searching system), etc.

Names unlike ordinary words present unique challenges in their comparison as two names which are not identical may both represent the same person or company. This is due to the particular characteristics inherent within names which allow a name to have several valid variations. Two examples of these characteristics are:

- alternate spellings
- nicknames or abbreviations

To this end, as the title of this dissertation suggests, name matching involves the process of matching similar names.

Computers are easily able to determine whether or not two compared names are an exact match to one another, i.e. verify whether or not both names are identical. The problem is, as highlighted previously, that this approach within a name matching context is inadequate. Through a combination of experience and knowledge of language constructs, humans are capable of overcoming the variations in names to determine whether or not two non-identical names match to one another. Computers are able to mimic this ability through the implementation of fuzzy matching algorithms. Although fuzzy matching algorithms have been developed to match non-

identical words, names are a subset of general words and therefore these algorithms can be utilised to overcome the variations between two similar names.

In performing fuzzy matching one must be aware that there is no single fuzzy matching algorithm that can wholly duplicate the human ability to match similar names. Rather there is a host of algorithms, each of which has its own strengths and weaknesses and therefore has a particular scenario where it performs best. Therefore, when implementing name matching within a business context it is imperative that an algorithm be chosen that best suits the business's needs and provides a level of fuzziness with which the business is most comfortable. One must be aware that one of the inherent drawbacks of fuzzy matching is the concept of false positives, where an algorithm determines that two words / names are a match where in reality they are not. Again, when choosing a fuzzy matching algorithm, the tolerable number of generated false positives must be considered.

As mentioned previously, since there is no "perfect" fuzzy matching algorithm, it might occur that over time as business rules change, the fuzzy matching algorithm used to perform a particular name matching task is found to be no longer adequate and is required to be replaced. This process can occur regularly within a system's lifecycle, especially within the development of a new system. The above mentioned scenario demonstrates the need for a platform upon which fuzzy matching algorithms can be run but also allows for these algorithms to be easily replaced when there are changing requirements. This platform can be achieved through the use of a generic framework, which abstracts the end user away from the fuzzy matching algorithms and in turn abstracts the algorithms away from the underlying database. Furthermore this framework can be designed to cater for the specific characteristic inherent within names.

The objective of this dissertation was therefore the design of a generic framework for the matching of similar names. Beyond the requirements of this dissertation, the

development of the framework was necessitated by the name matching requirements of a real world business application. The framework was successfully deployed and has been operational for over a year.

1.1 Overview

In Section 2 an in-depth literature review is performed in order to clarify the nature of fuzzy name matching. This includes an analysis of the causes of variation in both general words and in names. Thereafter, an analysis of existing fuzzy matching algorithms and search methodologies is performed. Finally, both general and fuzzy matching frameworks are investigated. Having explored the subject background, Section 3 defines the problem at hand and thereafter proposes a methodology through which a solution is to be developed. Section 4 outlines the requirements that the proposed solution must fulfil. The solution design is documented in Section 5 and the remainder of the section evaluates the design and discusses the improvements made to the design to overcome several of the design's shortcomings. Section 6 discusses the testing performed against the design implementation to verify whether it fulfils the requirements defined in Sections 3 and 4. This section also includes an analysis of the test case results. Section 7 critically analyses the solution, proposes avenues for future developments and improvements and finally presents a conclusion.

2 Literature Review

A review of the literature relevant to the topic, the Design of a Generic Framework for the Matching of Similar Names, was necessary in order to understand the subject and to define the research question. In particular, the literature review investigates the following sub-topics:

- The causes of variation within general words and names
- Fuzzy matching algorithms and data-partitioning techniques
- Frameworks in general and fuzzy matching frameworks

2.1 Causes of Variation in Words

The need for fuzzy matching arises from the fact that words that should be identical to one another could have a variation in their spellings. Through investigations of the causes of these variations, several fuzzy matching algorithms have been developed in an attempt to overcome these discrepancies.

One of the major causes of variation in words is spelling error. Spelling errors in words can be divided into three categories, namely (Christen, 2006; Du, 2005):

- Typographical Errors – This type of error occurs when the data capturer knows the correct spelling of the word but makes a typing mistake (i.e. erroneously pushes the wrong button). An example of this would be the typing of the word “teh” when the intention was to write “the”.
- Cognitive Errors – This type of error is caused when the person writing/typing the word either has a misconception or a lack of knowledge regarding the correct spelling of the word. An example of such a case would be that the person spells the word “receive” as “recieve".
- Phonetic Errors – This type of error occurs when a person substitutes a phonetically equivalent sequence of letters as opposed to spelling the intended word correctly, i.e. the person replaces several letters within a word with other letters that sound the same. An instance of this would be the word “naturally”

being spelled as “nacherly”. It has been found the phonetic errors tend to distort spellings more so than typographic and cognitive errors.

Another source of error that cannot be ignored is the medium by which the words are transmitted and submitted. “A human ear might misinterpret similar sounding letters (e.g. d–t, m–n)” (Du, 2005). This could be further exacerbated if the word has been repeated by a series of people, some of whom do not communicate the word correctly (Zobel & Dart, 1996). The use of automated technology for the input of a word can also cause errors, for example, an optical character recogniser (OCR) might confuse letters that look similar. An example of such letters is the confusing of the letter ‘u’ with the letter ‘v’ (Du, 2005).

Several studies, investigating spelling errors in general words, have found that single character errors account for the majority[1] of errors. A single character error is caused by the insertion, deletion or substitution of a single character into the correct spelling of a word.

A study investigating patient names within various hospital databases revealed that personal names presented different types and distributions of errors as compared to general words. The insertion of an additional name, initial or title was found to be a common error within this study as it accounted for 36% of errors. Of even greater interest, is that single character errors (which normally cause a large majority of errors in general words), only accounted for 39% of errors (Christen, 2006). Thus, the characteristics of personal names (as compared to general words), have resulted in their having unique causes of variation in addition to those that affect general words.

[1] One study in particular found that up to 80% of errors are single character errors (Christen, 2006).

2.1.1 Causes of Variation within Names

Names[2] have several characteristics that cause additional complexity, when matching two or more names. The following is an explanation of these characteristics

- Personal names can have multiple valid spelling and character variations.
  Unlike normal words, which generally only have one correct spelling, personal names can have multiple valid spellings. For example the name 'Gail', 'Gale' and / or 'Gayle' (Christen, 2006).

- Names can have multiple variations in the characters that constitute the name. Several types of variations are presented below (Snae, 2007):
  - Capitalisation, e.g. brown and Brown; SMITH and Smith
  - Punctuation, e.g. WILL SMITH and WILL-SMITH; SMIT and S.M.I.T; B.Z. Smith and BZ Smith
  - Spacing, e.g. YOUNGSMITH and YOUNG SMITH

- Nicknames, Abbreviations, Middle Names and Suffixes
  Often, people do not necessarily go by their full first names, rather they make use of nicknames or abbreviations of their names. A nickname could have originated from variations of a person's first name or surname (like 'Vesty' for 'Vest') or might relate to some life event, character sketch or physical characteristics of that person. An abbreviation is merely a shortened form of that person's name for example "Bob" for 'Robert', or "Liz" for 'Elizabeth' (Christen, 2006).

  Similarly, people may use their first name in some situations, whereas in other situations, they might go by the first and middle names. People might also on

[2] Though personal names are generally the first type of names that comes to mind when discussing name matching, one may not ignore corporate names as they possess characteristics that make them more complex to match than ordinary strings.

occasion specify their name with a suffix, for example Jr or II (Bell & Sethi, 2001).

- Married Names.

  It is common for Western women to change their surnames to that of their husbands' when they get married. Thus a person can potentially have two separate unrelated surnames, to which they are associated; depending on when their name was captured into a particular database (i.e. prior to or post their wedding) (Christen, 2006).

- Transcription

  When transcribing a name that was originally written in one alphabet into another (e.g. from the Arabic alphabet to the Latin alphabet), one approximates the phonemes of the source alphabet to those in the destination alphabet. Due to their often being several letters / letter sequences that sound the same (e.g "c" and "k", f and "ph", etc), there can be a variety of ways in that the original name can be transcribed. Therefore, there may be several legitimate spelling or phonetic variations of the original name (Du, 2005).

- Cultural Differences

  Various cultures have different ways in which names are presented. This introduces further complexity when comparing names as it may not be assumed that standard naming conventions apply to other cultures. For example, in "Asian names, the surname traditionally appears before the given name, and frequently a Western given name is added. Hispanic names can contain two surnames, while Arabic names are often made of several components and contain various affixes (Christen, 2006)". Similarly, in several European countries it is common for people to have compound names for example "Hans-Peter" or "Jean-Pierre" (Christen, 2006).

2.2 Fuzzy Matching Algorithms

Fuzzy matching algorithms can be divided into two high level categories, namely Orthographic and Phonetic matching. Within these two broad categories, there are two types of metrics, which are used to determine the degree of a match between two strings. One can either determine the distance between the two strings (how different the two strings are) or one can determine the similarity between the two strings. Thus the lower the distance between two strings, the higher the similarity and therefore the more alike they are (Kolatch et al., 2004).

The majority of string matching algorithms have been developed for general word matching and not specifically for name matching.

This literature review focuses only on well-known, well documented algorithms. Many of the newer algorithms are built upon (or are adaptations of) the classical algorithms, whereas a few use completely new ideas (for example syllable alignment). The reason only well documented algorithms have been reviewed is due to the fact that there is very little literature and analysis available on the newer algorithms. In addition, since the majority of algorithms are built upon older research, an analysis of classical fuzzy matching algorithms is sufficient to show the varying requirements and computation strategies required to be supported by a framework.

Figure 2.1 below provides a roadmap of the various Orthographic, Phonetic and Hybrid (see §2.2.3) fuzzy matching algorithms that are investigated within this literature review. The figure demonstrates the relationships between the various algorithms and in particular highlights how one or more algorithms are extended to a form a new algorithm.

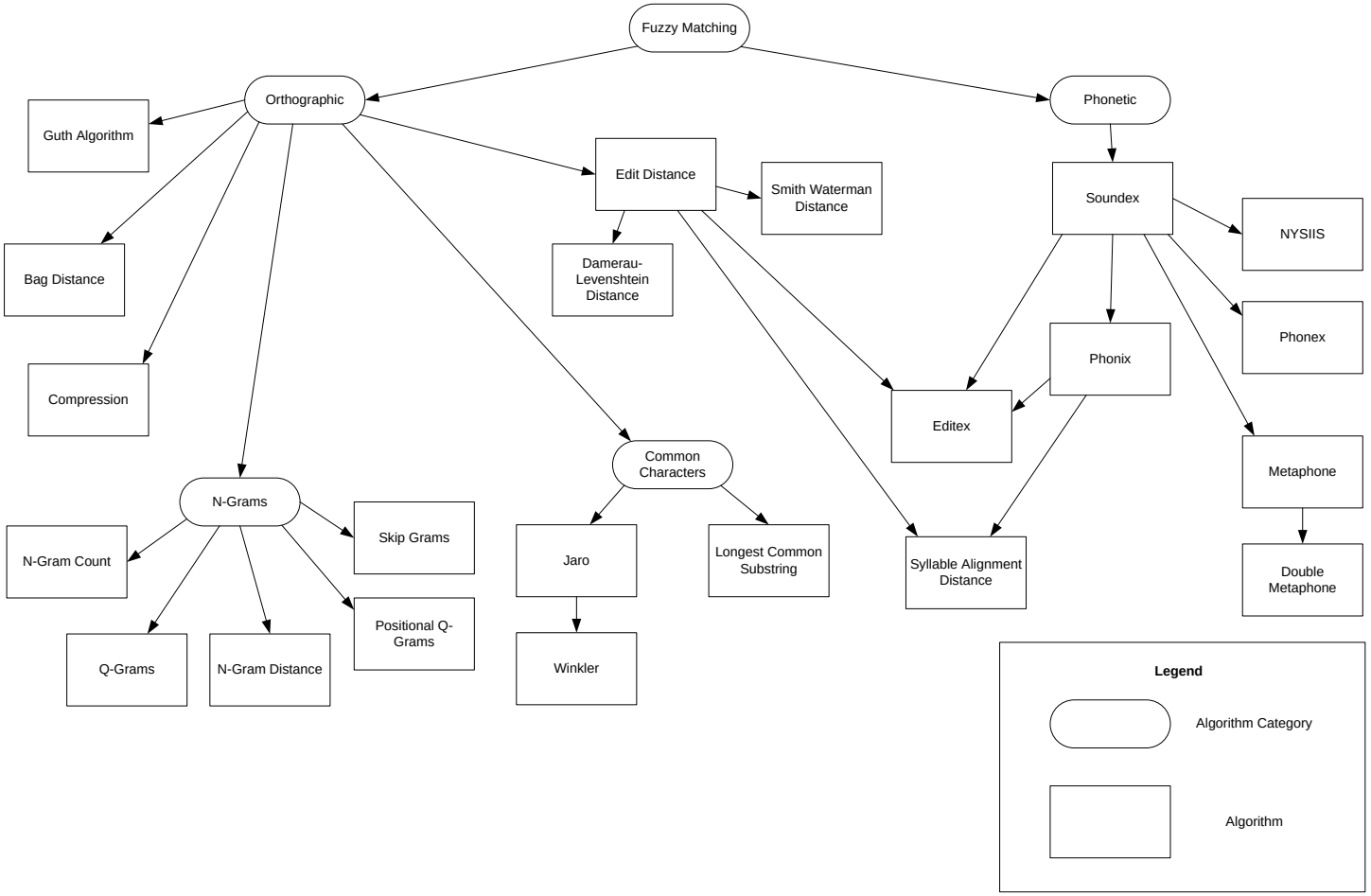


Figure 2.1: Fuzzy Matching Algorithms investigated within Literature Review

2.2.1 Orthographic Models

Orthographic models calculate the similarity or distance between two strings based on the number of steps required to transform one into the other or the number of characters they have in common (Kolatch et al., 2004).

Guth Algorithm

The Guth Algorithm is an alphabetic method, which is independent of language as it performs letter by letter comparisons (Lait & Randell, 1993; Snae, 2007).

Algorithm Implementation

The initial step in the algorithm implementation is a check to determine whether the two compared names are identical (Lait & Randell, 1993). Upon this step failing, the algorithm proceeds to compare the words letter-by-letter. When the algorithm "encounters different letters in the same position it then searches for matching letters in other positions (Lait & Randell, 1993)". The algorithm implementation is explained in Figure 2.2 below:

	Position in Name 1	Position in Name 2
Test 1	X	X
Test 2	X	X + 1
Test 3	X	X + 2
Test 4	X	X – 1
Test 5	X – 1	X
Test 6	X + 1	X
Test 7	X + 2	X
Test 8	X + 1	X + 1
Test 9	X + 2	X + 2

Figure 2.2: Guth Algorithm Comparison Steps (Lait & Randell, 1993)

If none of the above mentioned “tests” pass, the two words that are being compared are declared to be different.

Using the “tests” shown in Figure 2.2, Figure 2.3 explains how the Guth Algorithm would match the names Glavin and Glawyn.

I	II	III
G L A V I N	G L A ***V*** I N	G L A ***V*** I N
G L A W Y N	G L A W ***Y*** N	G L A W Y ***N***
IV	V	VI
G L A ***V*** I N	G L ***A*** V I N	G L A V ***I*** N
G L ***A*** W Y N	G L A ***W*** Y N	G L A ***W*** Y N
VII	VIII	IX
G L A V I ***N***	G L A V ***I*** N	G L A V I ***N***
G L A ***W*** Y N	G L A W ***Y*** N	G L A W Y ***N***

Figure 2.3: Example of Guth Algorithm Name Matching (Lait & Randell, 1993)

Algorithm Discussion

The algorithm is claimed to have four main advantages over other Fuzzy Matching algorithms (Lait & Randell, 1993):

1. It is not dependent on prior generation of a key, which is required by certain phonetic algorithms (See §2.2.2).
2. Since it requires no knowledge of the phonetics or the context in which the name is used, it is data independent and could be adapted easily to different types of data or linkage requirements.
3. The algorithm does not alter the input records in any way.
4. Alternative spellings of the same word are identified by the position of the letters in the word and not by phonetic equivalency.

A further strength of the algorithm is due to its use of character references (character comparisons) as opposed to a generated key and therefore the matching of two names requires little additional computing. (Lait & Randell, 1993)

Since the algorithm only compares the letters that are present in the two compared names, it would not be capable of matching a name to the same name when it is in the form of double name. (Lait & Randell, 1993) A further weakness of the algorithm is due to it "hunting" for letters in the compared name, which can cause incorrect matches between names that bear little visual resemblance to one another (causing to match names "liberally"). (Lait & Randell, 1993) This problem is exacerbated when comparing short names where one or two common vowels can produce a mismatch (Lait & Randell, 1993; Snae, 2007).

Edit Distance (Levenshtein Distance)

One of the most common Orthographic measures is that of the string edit distance. The Edit Distance provides a measure of the minimum number of edit operations (insertions, deletions and substitutions) required to convert one string (name) into another (Christen, 2006; Hsiung et al., 2005). The edit distance utilises an algorithm paradigm called Dynamic Programming to perform the calculation (Christen, 2006).

Dynamic Programming

Dynamic Programming is an algorithm design paradigm used to solve optimisation problems. This paradigm makes use of the following three components to solve a problem (Atallah, 1999):

- Principle of Optimality
- Sub-problem solutions
- Caching

Atalah defines the Principle of Optimality as “the observation, in some optimization problems, that components of a globally optimum solution must themselves be globally optimum (Atallah, 1999)”. Therefore, one is able to find the optimum solution to a problem though breaking it up into smaller sub-problems and finding the optimal solution to each. Using the solution of the smaller components, one is capable of finding the solution of the problem as a whole.

Following from the “Principle of Optimality”, it would become necessary to break the problem up into multiple sub-problems and find their solutions.

Often the solution of one sub-problem is dependent on the solutions of previous sub-problems. Instead of re-performing all the calculations for each of the sub-problems, as a solution for a sub-problem is calculated it is cached (stored) to be utilised by subsequent sub-problems. Through caching, the dynamic programming saves both time and additional computation.

Implementation of the Edit Distance

The edit distance (edit) is calculated by iterating through and then comparing each of the letters in the two words that are being compared. Each set of letters that are compared are compared using the function *d(i,j)*. The minimum edit distance for the entire word is calculated by determining $d(|x|,|y|)$ (where $|x|$ is the length of word x and $|y|$ is the length of word y). See Equation 1 (Zobel & Dart, 1996).

$$d(i,j) = \min\begin{bmatrix} d(i-1,j)+1, \\ d(i,j-1)+1, \\ d(i-1,j-1)+c(x_i,y_j) \end{bmatrix} \quad (1)$$

where:

$$d(0,0) = 0 \quad (2)$$

$$d(i,0) = i \quad (3)$$

$$d(0,j) = j \quad (4)$$

$$c(x_i,y_i) = \begin{cases} 0 & if\ x_i = y_i \\ 1 & if\ x_i \neq y_i \end{cases} \quad (5)$$

x, y are the two words being compared

x_i is the i^{th} letter of the word x

y_j is the j^{th} letter of the word y

The need for the *min* function in Equation 1 is in order to determine which edit operation has been performed between the two letters and how much it cost. The first expression takes into account *insertion*, the second expression takes into account *deletion* whilst the third takes into account *substitution* (Du, 2005).

In order to calculate the edit distance for the entire word (finding the edit distance between the final letters of the two words), the algorithm is required to recursively determine the edit distance for each of the previous letters until it has reached the beginning of both of the two words. This would cause the algorithm to evaluate the distance between a set of two letters several times and thus make the algorithm highly computationally expensive. The computational complexity is reduced through the use of Dynamic Programming, where the edit distance between each of the letter sets is calculated and stored within a zero-indexed matrix of dimensions $(|x|+1)(|y|+1)$. In this manner, the algorithm is able to start in the upper left-hand corner of the matrix, and determine the distance of each cell, having used the distances that are stored in the adjacent three cells (the cell above, the cell diagonally above and to the left and the cell to the left). The value contained in the bottom right-hand corner cell is the

minimum edit distance for the two words. Figure 2.4 displays the evaluation of the Edit Distance through the use of a Dynamic Programming matrix (Du, 2005).

	ϵ	p	h	i	l	l	i	p	s
ϵ	0	1	2	3	4	5	6	7	8
f	1	1	2	3	4	5	6	7	8
i	2	2	2	2	3	4	5	6	7
l	3	3	3	3	2	3	4	5	6
i	4	4	4	3	3	3	3	4	5
p	5	4	5	4	4	4	4	3	4
s	6	5	5	5	5	5	5	4	**3**

Figure 2.4: Calculating the minimum edit distance between Filips and Phillips (Du, 2005)

Algorithm Discussion

Using the Dynamic Programming approach, the complexity of the edit distance algorithm is $O(|x||y|)$. The algorithm can only evaluate the edit distance between two words at a single time and if one were required to compare a search name to an entire dictionary[1], the whole dictionary would have to be processed, as it would be required that every name in the dictionary be compared against the search name. This effectively makes the edit distance algorithm very slow when comparing against large dictionaries (Du, 2005).

Damerau-Levenshtein Distance

The Damerau-Levenshtein Distance differs from the basic Edit Distance, in that the transposition of two letters (characters) is considered to be a single operation as opposed to being two operations in the basic algorithm (Christen, 2006). The implementation of the Damerau-Levenshtein algorithm extends the basic Edit Distance algorithm (as is shown in Figure 2.1) through the addition of an expression to the basic Edit-Distance function. The expression takes into account the

[1] A dictionary is a list of words (names), against which one compares a search word. A Lexicon is a synonym for the word dictionary.

transposition of characters. The Damerau-Levenshtein Distance is shown in Equation 6 (Du, 2005).

$$d(i,j) = \min \begin{bmatrix} d(i-1,j)+1, \\ d(i,j-1)+1, \\ d(i-1,j-1)+c(x_i,y_j), \\ d(i-2,j-2)+c(x_{i-1},y_j)+c(x_i,y_{j-1})+1 \end{bmatrix} \quad (6)$$

The internal expressions of Equation 6 are described in Equations 2 – 5 above.

Further derivatives of the Edit Distance algorithm can be found in Appendix A.

Bag Distance

A bag is defined as a multi-set of letters within a string (i.e. a substring). The Bag Distance is considered to be a cheap approximation of the edit distance and is always smaller and equal to the corresponding edit distance (Christen, 2006). The reason it is considered a cheap approximation of the edit distance is due to it not requiring further processing of the strings (that are being compared) and is very fast to compute (Bartolini et al., 2002).

Implementation

The Bag Distance is implemented through the following equation (Bartolini et al., 2002):

$$d_{bag}(x,y) = \max\left\{|X-Y|,|Y-X|\right\} \quad (7)$$

where:

x, y are the two words being compared

$$X = multiset(x) = ms(x) \quad (8)$$

$$Y = multiset(y) = ms(y) \quad (9)$$

The following example explains how the bag distance between “spire” and “fare” is found (Bartolini et al., 2002).

$$d_{bag}(\text{"}spire\text{","}fare\text{"}) = \max\{|X-Y|, |Y-X|\}$$
$$= \max\{3,2\} = 3$$

where:

$$X = ms(\text{"}spire\text{"}) = \{\{i, e, p, r, s\}\}$$
$$Y = ms(\text{"}fare\text{"}) = \{\{a, e, f, r\}\}$$
$$X - Y = \{\{i, e, p, r, s\}\} - \{\{a, e, f, r\}\} = \{\{i, p, s\}\}$$
$$Y - X = \{\{a, e, f, r\}\} - \{\{i, e, p, r, s\}\} = \{a, f\}$$
$$|X - Y| = |\{\{i, p, s\}\}| = 3$$
$$|Y - X| = |\{\{a, f\}\}| = 2$$

The difference between two multi-sets is found by taking all the elements in first set and removing all the elements that are common to both the first and second set (Bartolini et al., 2002). For example: *X*-*Y* = all elements of *X*, which are not elements of *Y*.

Agrep

Agrep is a utility which allows for the rapid identification of strings that contain substrings, which match either the query string or variations thereof. These variations of the query string can contain up to a user specified number of insertions, deletions or substitutions. Unlike other algorithms, Agrep does not rank any of the results. It must be noted that Agrep was not designed to be used as a name matching tool but rather for the fast searching of large files (Zobel & Dart, 1996).

Smith Waterman Distance

The Smith-Waterman Algorithm was developed to compute the edit distance between two sequences, typically that of either DNA or protein, using a dynamic programming approach (Christen, 2006; Pang, 2007). Since this algorithm relates to a very specific field within string matching it will not be investigated further in this literary review. The relationship between this algorithm and the edit distance algorithm is demonstrated in Figure 2.1.

Common Characters

Longest Common Substring (LCS)

The LCS determines the longest substring that is common to both the words that are being compared (Christen, 2006)

Jaro

The Jaro algorithm calculates a similarity measure between two strings by determining the number of common characters shared between the two strings and also the number of transpositions that would be required to convert the one string into the other (Christen, 2006; Yancey, 2005).

Before discussing the formula used within the algorithm calculation it is necessary to define several terms; especially the context in which they are used within this particular algorithm.

- A common character is defined as a character that is common to both of the compared strings and is found within half the length of the longer string (Yancey, 2005). i.e. if the length of the longest string is x then for a character to be considered common between the two strings the same character must be

found within a distance of *x/2* from the position in which it was found in the first string.

- "The definition of transposition is that the character from one string is out of order with the corresponding common character from the other string" (Yancey, 2005).

The Jaro similarity measure is described in Equation 10 below (Yancey, 2005).

$$\Phi_j(x,y) = \frac{1}{3}\left(\frac{N_C}{|x|} + \frac{N_C}{|y|} + \frac{N_T}{N_C}\right) \qquad (10)$$

where

$\Phi_j(x,y)$ is the Jaro similarity function between two strings x and y

N_C is the number of common characters between strings x and y as per the above mentioned definition

N_T is calculated by subtracting N_C from the number of transpositions. The number of transpositions is calculated by dividing the number of out-of-order common character pairs by 2 and thereafter rounding down the result to the nearest whole integer (Yancey, 2005).

Winkler (Jaro-Winkler)

The Winkler algorithm utilises the results of a study that found that there are typically fewer errors at the beginning of names and increases the Jaro Similarity measure when there are up to four characters that match at the beginning of two names (Christen, 2006)

The Winkler algorithm is calculated through the use of equation 11 (Yancey, 2005).

$$\Phi_w(x,y) = \Phi_j(x,y) + \frac{p(1-\Phi_j(x,y))}{10} \qquad (11)$$

where:

p is the length of the common prefix (shared by both strings x and y). p can have a maximum value of 4

N-Grams

An N-gram is a substring contained within a word of length n (Navarro, 2001). The advantage of using n-gram analysis is that it is language independent as it only involves the comparison of letters; furthermore since n-gram analysis utilises unique combinations of letters (as opposed to individual letters) the effective "alphabet" used to compare two or more words is expanded, allowing algorithms to be more sensitive to the similarity between the compared word (Du, 2005; Salmela et al., 2007).

The subsequent sections deal with various variations of N-Grams. Please refer to Figure 2.1 for an overview of the various N-Gram algorithms.

N-gram Count

This is the simplest of the n-gram algorithms and is simply a count of the number of n-grams that the two words (that are being compared) have in common. The formula for this calculation is displayed in equation 12 (Du, 2005).

$$gram-count = |N_1 \cap N_2| \qquad (12)$$

where: N_1 and N_2 are the sets of n-grams in the two words respectively

The main weakness of the N-gram count algorithm is that it does not take into account the lengths of the two compared words and could give inaccurate results for two words that have very different lengths (Zobel & Dart, 1996).

N-Grams (Q-Grams)

This algorithm calculates a similarity measure between two strings. This is achieved through the determining of the number of substrings of length n (or length q) that are common to both the compared words and then dividing that total by the aggregated number of n-grams between both the words. There are several variations to this algorithm, where the aggregated number is the minimum number[2], the average number[3] or the maximum number[4] of n-grams between the two words (Christen, 2006).

Another variation is the gram-coefficient, where the denominator consists of the total number of n-grams in both the words and the numerator is the N-Gram count between both words (Du, 2005). The gram-coefficient is also referred to as the Jaccard similarity (Veronica & Li, 2009). This is shown in Equation 13 (Du, 2005).

$$gram-coefficien\,t = \frac{|N_1 \cap N_2|}{|N_1 \cup N_2|} \qquad (13)$$

N-gram Distance

As the name states the n-gram distance measure calculates the distance between two words as opposed to the similarity between them (as has been calculated in the previous n-gram methods). The formula for the n-gram distance is specified in Equation 14 (Du, 2005).

$$gram-dist = |N_1| + |N_2| - 2|N_1 \cap N_2| \qquad (14)$$

[2] The minimum number of n-grams is the minimum number of n-grams that are contained in either one of the two words that are being compared.
[3] This is the average number of n-grams contained in both the words that are being compared.
[4] This is the maximum number of n-grams contained in either of the words that are being compared.

Variations to Traditional N-Gram Analysis

Positional Q-Grams

Positional Q-Grams are an extension of the basic q-gram (n-gram). The positional q-gram differs from basic n-gram in that it also contains positional information about the various n-grams (i.e. the position of the n-gram within the word). A positional q-gram is defined by the following equation (Yang et al., 2008):

$$PQ(i,s) = (i, s[i, i+q-1]) \quad (15)$$

where:

PQ(i,s) is the positional q-gram at position i in string s, with q-gram length of q

s[i,j] is the substring of string s from position i to position j

The set of positional q-grams in a word is defined as follows:

$$G(s,q) = \{PQ(1,s), PQ(2,s), \dots, PQ(|s|-1,s)\} \quad (16)$$

The following example demonstrates the set of positional q-grams associated with the word “bitingin”, when using a q-gram length 2 (Yang et al., 2008):

$$\text{G("bitingin",2)} = \{(1,bi),(2,it),(3,ti),(4,in),(5,ng),(6,gi),(7,in)\}$$

Unlike traditional q-grams, two positional q-grams are only considered to be a match, when they are within a specified distance from one another (Christen, 2006; Piskorski et al., 2007). This aids in an algorithm’s accuracy as it does not blindly match common q-grams but rather uses their positions to verify that the two q-grams are applicable to one another.

Skip Grams

Skip grams extend basic n-grams by not only utilising bigrams[5] or trigrams[6], which consist of adjacent characters (from the original strings) but rather utilise n-grams which are composed of non-adjacent letters (Christen, 2006; Piskorski et al., 2007).

This type of analysis can assist in the matching of words that have transposition errors caused by typos.

Compression

It has been proposed that data compression be used as a similarity measure, in that one can derive a measure of similarity between two compressed strings (that have been compressed using the same compression algorithm) by comparing their lengths (Christen, 2006).

2.2.2 Phonetic Models

Phonetic models utilise rules on how words / names are pronounced to perform name matching and are therefore often referred to as "sound alike" algorithms (Du, 2005). One class of phonetic models is Phonetic Encoding. "Phonetic encoding techniques convert a name string into a code" (Christen, 2006) according to how it is pronounced. Once the code has been produced one is able to compare the codes of two strings. One can either perform an exact match or use some other form of approximate string matching to determine whether there is a match between the two names. Due to the nature of the pronunciation of the various letters and syllables within a name, phonetic encoding algorithms are generally language dependent (Christen, 2006).

[5] A bigram is an n-gram which is two characters long.
[6] A trigram is an n-gram which is three characters long.

Soundex (Russel Soundex)

Soundex is one of the oldest (having been patented in 1918) and most well known phonetic algorithm (Zobel & Dart, 1996). Due to this reason many modern phonetic algorithms extend from Soundex (refer to Figure 2.1) and use it as a performance benchmark.

Soundex converts each name into a four-character code, based on the sound of each letter. The code is generated by maintaining the first letter of the name and then converting the rest of the letters to numeric codes (Christen, 2006; Du, 2005). The conversion of the various letters to numbers is achieved through the use of a look-up table, which is shown in Figure 2.5.

Letters	Code
a e h i o u w y	0
b f p v	1
c g j k q s x z	2
d t	3
l	4
m n	5
r	6

Figure 2.5: Soundex Code Look-Up Table (Du, 2005)

Implementation

As stated previously, in converting the name to a code, the algorithm begins by copying the first letter of the name. Thereafter, each letter in the name is converted to its Soundex equivalent code and added to the name's full Soundex code. However, any letter whose mapping code is identical to the one of the preceding letter is ignored. Having coded the entire name, the algorithm thereafter removes all zeroes

from the code. Finally, the code is either truncated or padded with zeroes to ensure that it is four characters long (Du, 2005).

The following is an example of how the word Pfister is converted to a Soundex Code: "Pfister → P102306 → P02306 (since f has the same Soundex digit as its preceding letter p) → P236" (Du, 2005).

Discussion

Due to the fact that Soundex maintains the first letter of the original word, two names with the same pronunciation but different initial letters will have different Soundex codes, for example, the name Karlsson and Carlson result in the following codes, K642 amd C642 (Du, 2005). Furthermore, the truncating to four characters, though improving indexing, causes the algorithm to lose some of the original word's phonetic data (Zobel & Dart, 1996).

In addition Soundex is only capable of determining whether two strings are or are not similar and is not able to perform ranking of various strings for a particular search string (Zobel & Dart, 1996).

It must be however noted that compared to other algorithms, Soundex-like algorithms are relatively computationally inexpensive (Christen, 2006).

Phonex Algorithm

In an attempt to improve on the Soundex algorithm, the Phonex algorithm pre-processes the name according to its English pronunciation (Christen, 2006). Though the Phonex algorithm also converts the name into a four-character code (namely a letter followed by three digits), the produced code is not compatible or comparable to an equivalent code that would be produced for the same word by the Soundex algorithm (Lait & Randell, 1993). Thereafter, as performed in the Soundex algorithm

the codes of the two compared names are compared. If the codes are the same then the two names are considered to be similar.

Implementation

The Phonex algorithm consists of two sets of rules. The first set of rules deals with the pre-processing of the name, thereafter a second set of rules is implemented on the already processed name (Lait & Randell, 1993).

The following rules are applied to pre-process the name (Lait & Randell, 1993):

1. All trailing 'S' characters are removed from the end of the name.
2. Certain leading letter-pairs are converted to an equivalent single letter, as follows:
    a. KN → N
    b. WR → R
    c. PH → F
3. Certain leading single letters are converted to an equivalent letter, as follows:
    a. E, I, O, U, Y → A
    b. K, Q → C
    c. P → B
    d. J → G
    e. V → F
    f. Z → S
4. The leading letter of the name is removed if it is an 'H'

Once pre-processed, the following rules are applied to the name to generate the equivalent Phonex code (Lait & Randell, 1993):

1. The first letter of the name is retained, while any subsequent occurrences of A, E, H, I, O, U, W, Y are dropped.

2. All letters, barring the first one, of the name are converted to a numeric code. The mappings are as follows:
    a. B, F, P, V → 1
    b. C, G, J, K, Q, S, X, Z → 2
    c. D, T → 3 If followed by C, ignore the current letter
    d. L → 4 If the letter is followed by a vowel or is at the end of name
    e. M, N → 5 If the next letter is either a D or G, ignore it.
    f. R → 6 If the letter is followed by a vowel or is at the end of name
3. If the generated code contains any repeated consecutive numeric codes, the repeated number is dropped.
4. The produced code is either truncated or padded with zeroes to ensure that it is four characters long.

Discussion

As in the case of Soundex, since the code is truncated to four characters, the algorithm will lose some of the phonetic information contained in longer names and could therefore cause false-positive matches. Furthermore, the algorithm's initial pre-processing step adds further complexity to the algorithm, causing it to be more computationally expensive than Soundex.

Phonix

Phonix attempts to improve on both Soundex and Phonex, through the application of transformation rules on groups of letters (in total there are 160 transformations (Du, 2005)). The rules vary depending on the position of the letter / group of letters within

the word (Christen, 2006). Once the transformation rules have been applied, the transformed name is mapped to an equivalent code (Du, 2005).

Implemenation

Phonix can be broken into six distinct steps as listed below (Du, 2005):

1. Phonetic transformations are performed on the name where certain letter groups are replaced with other letter groups. An example of such is that **gn**, **ghn** and **gne** are replaced with the letter **n**.
2. If the initial letter is either a vowel or the letter y, it must be replaced with the letter **v**.
3. The ending sound (suffix) is separated from the name. (This is roughly the substring from last vowel or last y to the end of the word).
4. All the remaining vowels, the consonants **h**, **w** and **y** and all repeated characters are removed.
5. Using the Phonix table (shown in Figure 2.6) all the letters in the name (without the suffex) barring the initial letter, are replaced with their equivalent digits to form the Phonix code.
6. Step 5 is repeated on the suffix that was removed from the word in step 3.

Letters	Code
a e h i o u w y	0
b p	1
c g j k q	2
d t	3
l	4
m n	5
r	6
f v	7
s x z	8

Figure 2.6: Phonix Code Look Up Table (Du, 2005)

As in the case of Soundex, Phonix codes also have maximum length, however in the case of Phonix, the length is eight characters (Du, 2005).

Discussion

Phonix is much more complex than Soundex and is therefore slower. However, since Phonix breaks the word into two codes, it allows for greater versatility when attempting to match names as both the main name code and the suffix code can be utilised to determine the degree of similarity between two names. The use of the suffix code allows the algorithm to maintain the phonetic information that is lost in other algorithms.s

New York State Identification Intelligence System (NYSIIS)

NYSIIS is another Soundex-like algorithm, which uses a series of transformation rules to convert letters / groups of letters to a phonetically similar letter or group of letters. Therefore, unlike Soundex and the other Soundex-like equvalents, NYSIIS converts the words (names) into alphabetic codes and not numeric / alphanumeric codes (Christen, 2006). Originally the NYSIIS algorithm converted words / names into a six character code, however modern variations of the algorithm produce codes of differing lengths (Rajković & Janković, 2007).

Implementation

The NYSIIS algorithm consists of seven discreet steps (Taft, 1970):

1. If certain letter combinations are found at the beginning of a name, they are replaced with algorithm defined phonetically similar letter combinations. The mappings are as follows:
    a. MAC → MCC
    b. KN → N
    c. K → C
    d. PH → FF
    e. PF → FF
    f. SCH → SSS
2. If certain letter combinations are found at the end of a name, they are replaced with algorithm defined phonetically similar letter combinations. The mappings are as follows:
    a. EE → Y
    b. IE → Y
    c. DT, RT, RD, NT, ND → D
3. The first character of the name is copied, to form the first character of the algorithm produced key.
4. The remaining characters (after the first character) are mapped to algorithm defined letter combinations as per the following mapping rules:
    a. EV → AF else A, E, I, O, U → A
    b. Q → G
    c. Z → S
    d. M → N
    e. KN → N else K → C
    f. SCH → SSS
    g. PH → FF
    h. H → If the previous or next character is a consonant, the previous character is used.

i. W → If the previous character is a vowel, the previous character is used.

j. If the current character is not same as the last character added to the key, the current character is added to the key.

5 If the last the character is an S, it is removed.

6 If the last characters are AY, they are replaced with the letter Y.

7 If last character is A, it is removed.

Discussion

It has been found that the NYSIIS algorithm offers a 2.7% improvement on the Soundex algorithm (Rajković & Janković, 2007).

Metaphone / Double Metaphone

Metaphone and Double Metaphone have been designed in order to accommodate non-English words. Being Soundex-like algorithms, they use a series of rules to transform letters and groups of letters to a phonetically equivalent letter, ultimately transforming the entire name / word into an equivalent code (Christen, 2006). Metaphone reduces the alphabet to sixteen consonant sounds and retains vowels only when they occur as the initial letter of the name (Du, 2005). Metaphone and Double Metaphone attempt to better Soundex by taking into account the following (Christen, 2006):

- The position of the particular letter/s in question
- The letters that both proceed and follow the letter/s in question.

Double Metaphone was designed as an enhancement on the original Metaphone algorithm as it improves on some of the letter encodings (mappings). Furthermore, Double Metaphone attempts to account for the different pronunciations of the input

name / word through the output of multiple encodings (namely a primary and secondary encoding) for each input string (Elmagarmid et al., 2007).

Discussion

When compared to the Soundex algorithm, the Metaphone and Double Metaphone algorithms are found to be more computationally expensive as more processing is required. It has been found that Double Metaphone's generation of additional encodings greatly improves the algorithm's matching performance, while adding little overhead to the algorithm (Elmagarmid et al., 2007).

2.2.3 Hybrid Techniques

These techniques combine both alphabetic (orthographic) and phonetic approaches in order to perform the fuzzy matching (Snae, 2007). Figure 2.1 demonstrates how both the Editex and Syllable Allignment algorithms (discussed below) are extended from existing orthographic and phonetic algorithms.

Editex

The Editex algorithm combines the letter grouping techniques of the Soundex and Phonix algorithms with Edit Distance methods. Editex works by assigning a cost of 0 if two letters (being compared between the two words) are the same, 1 if they are in the same letter group (see Soundex and Phonix letter transformation tables) and a cost of 2 if the two letters are neither the same nor in the same group (Christen, 2006). See Equations 17 to 22 and Figure 2.7 respectively (Zobel & Dart, 1996).

$$edit(i,j) = \min\begin{bmatrix} edit(i-1,j) + d(s_{i-1}, s_i), \\ edit(i,j-1) + d(t_{j-1}, t_j), \\ edit(i-1,j-1) + r(s_i, t_j) \end{bmatrix} \quad (17)$$

where:

$$edit(0,0) = 0 \quad (18)$$

$$edit(i,0) = edit(i-1,0) + d(s_{i-1}, s_i) \quad (19)$$

$$edit(0,j) = edit(0,j-1) + d(t_{j-1}, t_j) \quad (20)$$

$$r(a,b) = \begin{cases} 0 & a = b \\ 1 & group_a = group_b \\ 2 & a \neq b \quad and \quad group_a \neq group_b \end{cases} \quad (21)$$

$$d(a,b) = \begin{cases} 0 & a = b \\ 1 & group_a = group_b \\ 1 & (a = 'h' \; or \; a = 'w') \; and \; a \neq b \\ 2 & a \neq b \quad and \quad group_a \neq group_b \end{cases} \quad (22)$$

0	1	2	3	4	5	6	7	8	9
aeiouy	bp	ckq	dt	lr	mn	gj	fpv	sxz	csz

Figure 2.7: Editex Letter Groups (Zobel & Dart, 1996)

Syllable Alignment Distance

The Syllable Alignment Distance matches two names on the basis of comparing their syllables as opposed to individual letters. Initial pre-processing is achieved through the use of the Phonix transformations. Thereafter, the distance between two strings of syllables is calculated for each set of syllables. This distance is calculated by using Edit Distance methods. The beginning of each syllable is found through the implementation of a series of rules (Christen, 2006).

2.2.4 Probabilistic Models

Probabilistic models use other information (beyond that of the superficial analysis) known about the two names to determine if there is a match. This type of information is applied through the use of probabilities as one is able to determine the likelyhood of a specific feature occurring based on previous experience.

Dot Product (DP)

The DP calculates a measure of the degree of relation between two names / words. The DP does not make use of any orthographic nor phonetic features of the names but rather utilises the occurrences of both of the two names in various sources. It is calculated by summing together the product of the number of occurrences of each string in each particular source as is shown in Equation 23 (Hsiung et al., 2005).

$$DP(x,y) = \sum_{i=1}^{N} \left(\left| O(x)_i \right| \cdot \left| O(y)_i \right| \right) \qquad (23)$$

where: $O(k)_i$ is the occurrence of string k in source *i*

N is the number of sources in which the two strings are mentioned

Normalised Dot Product (NDP)

The NDP is implemented in a similar manner to the DP, however the number of occurrences of each string within a single source is normalised before the DP is calculated. This is achieved by dividing the number of occurrences of the string in a single source by dividing by the total number of occurrences of the string throughout all the sources (Hsiung et al., 2005). See Equation 24 for the implementation thereof.

$$NDP(x,y) = \sum_{i=1}^{N} \left(\frac{|O(x)_i|}{|O(x)_T|} \cdot \frac{|O(y)_i|}{|O(y)_T|} \right) \qquad (24)$$

where: $O(k)_T$ is the occurrence of string k throughout all the sources

Common Friends (CF)

CF is simply a count of the number of sources (friends) that contain both the two names that are being compared (Hsiung et al., 2005).

KL Distance (KL)

The KL is calculated through the use of the normalised number of occurrences of each string per source. It must be noted that one must use "add-one smoothing"[7] (Hsiung et al., 2005) to allow for the inclusion of sources that have no occurrences of either one / both of the strings (Hsiung et al., 2005). The formula for the calculation of the KL is given in equation 25 (Hsiung et al., 2005).

[7] Add-one smoothing is achieved by adding one to the number of occurrences (of the string) in a single source before one divides by the total number of occurrences (of the string) found throughout all the sources (Hsiung et al., 2005).

$$KL(x,y)=\sum_{i=1}^{N}\left(O'(x)_i \log\left(\frac{O'(x)_i}{O'(y)_i}\right)+O'(y)_i \log\left(\frac{O'(y)_i}{O'(x)_i}\right)\right) \quad (25)$$

where: $$O'(k)_i=\frac{|O(k)_i|}{|O(k)_T|} \quad (26)$$

k is a string that is being compared

2.2.5 Commonality between Fuzzy Matching Algorithms

Having completed a thorough analysis of various fuzzy matching algorithms, it has become apparent that Orthographic and Phonetic algorithms contain common elements, which can be implemented through a series of generic processes. The nature and implementation of Probabilistic algorithms prevent them from being incorporated within generic fuzzy matching processes.

Unlike the other types of fuzzy matching algorithms, Probabilistic algorithms attempt to determine whether two words are related, as opposed to determining whether they are similar. This is further elaborated by stating that these algorithms do not perform inexact matching as they do not generate a metric of how similar or different the two compared words are. Secondly the main problem that hinders the incorporation of these algorithms into a generic process is their implementation requirements; these algorithms require that multiple sources of information be parsed to determine the number of occurrences of a particular search word or name. This attribute is not shared with any of the other discussed algorithm types.

Commonality between Orthographic and Phonetic Algorithms

The common aspects between Orthographic and Phonetic algorithms can generally be broken into three high level processes:

1. Input processing – this step is performed prior to the search of the input word to ensure that both the input word and the words against which it is

to be searched, conform to one another. This step could either be an inherent aspect of the algorithm, for example where both the input word and the words against which the input word is to be searched, are to be converted into a code (e.g. the Soundex algorithm) and/or is used to ensure that all the words are in an optimum form for a search, i.e. the words are all capitalised and punctuation has been removed.

2. Match searching – this step is core to any matching algorithm (even beyond fuzzy matching), in which potential matches to the input word are retrieved. This could be achieved through a database search, an in-memory search (the word list could be stored in several different structures) or simply the potential match is a comparison word that was input with the original search word.
3. Match evaluation – the final step requires the degree of the match between the input word and each of the words returned from the previous step to be determined. This step is intuitive for algorithms that calculate a metric (either a distance or a similarity) through the direct comparison between the search word and the potential matches. In the case of algorithms, which have already evaluated the fuzzy matches to the search word through the match searching step, this step can be used to quantify an exact degree of the match.

2.3 Pre-Search Data Partitioning

"When matching personal names, the size of the name list can be extensive. It is therefore important to avoid exhaustive searches of the list every time a new name needs to be matched. A partitioning method can be used to retrieve the part of the list which is most likely to be interesting. Then an approximate name matching algorithm is used on this partition to find the relevant matches (Du, 2005)". As Figure 2.8 demonstrates, the advantage of searching against a partition of the search database (as opposed to the whole database), is that one is able to search against a smaller subset.

This in turn improves search times as there is less data to search against and also reduces the number of false-positive matches as the data against which one is searching has been optimised for that particular search.

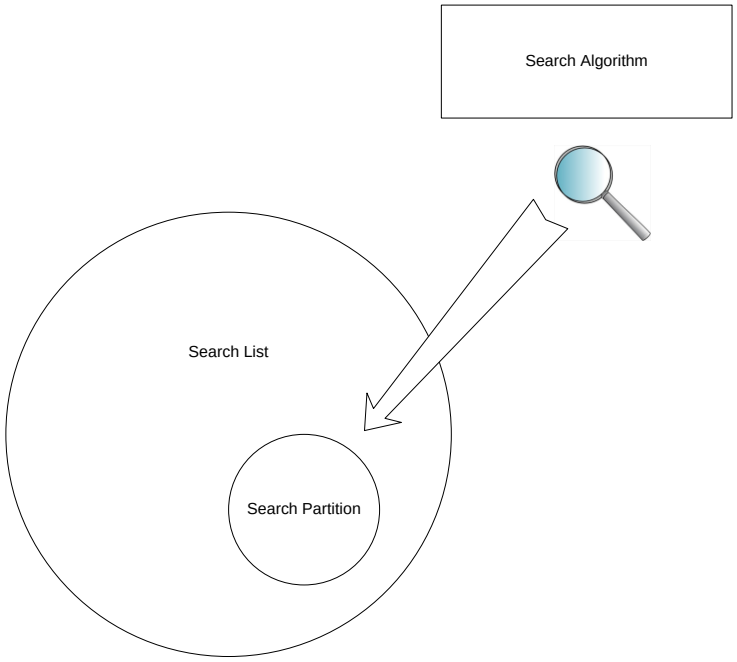


Figure 2.8: A search performed against a partition of the search database

The remainder of this section discusses the various partitioning methods.

2.3.1 First Letter

A simple means of partitioning the list of names is to partition the list by the first letter of each of the names (Du, 2005).

This method, however, is problematic when one attempts to match names that do not have the same initial letter (Du, 2005).

2.3.2 Word Length

Another means of partitioning a list of names, is to partition the list based on the length of the various names. When attempting to match a name, one searches through the portioned list for all names that have a word length within the specified tolerance (Du, 2005).

Du (2005) suggests that this type of partitioning technique may not significantly decrease search time because the length of the majority of names within a list does not vary much.

2.3.3 Word Halves

A list of words can be partitioned by indexing both the first and the second half of each word. The reason for splitting up the words into two halves is that if two words differ by only a single error then there must be an exact match on either the first or second half of the word (Du, 2005).

The indices for the two halves can be implemented through the use of two trees, namely a prefix tree and a suffix tree (See Appendix A for the implementation of such) (Du, 2005). It is assumed that the prefix tree is produced in the same manner as in Appendix A, starting at the beginning of the word and building up the tree letter by letter until the end of the word is reached. It is assumed that the suffix tree is built by starting at the end of the word and working one's way to the beginning of the word.

Matches to the search word are found by dividing the search word into two halves. The prefix of the word is searched against the prefix tree and the suffix of the word is searched against the suffix tree. Therefore, all words that require further searching are the words that have further branches below the matched prefix (in the prefix tree) and the matched suffix (in the suffix tree) (Du, 2005).

2.3.4 N-Grams

N-Grams can be used as another means to partition a list of names. This is achieved through the use of an "inverted index of n-grams" (Du, 2005). Each name in the list is given a unique number. Thereafter, for every possible n-gram, a list of numbers corresponding to the numbers of the words that contain that specific n-gram is generated. All potential matches to a search word are found by retrieving all the words that are associated with each n-gram contained within the search word. This is achieved through the use of the constructed word number lists associated with each n-gram (Du, 2005).

2.3.5 Bloom Filter

A Bloom Filter can be used as a further tool to pre-partition a list of names.

Explanation of Bloom Filter

A Bloom Filter is a particular type of hash table, in which all entries are either 1 or 0. The Bloom Filter hash table is constructed such that every single word in a list contains a corresponding entry in the table. In order to determine if a word is within the list, it is hashed repeatedly with multiple hash functions. If all the hash entries are equal to one, then the word is contained within the list. However, if any of the entries are equal to zero, the word is not contained within the list (Du, 2005).

The problem with using hash functions is that there is a chance that a word that is not part of the originally hashed list, could appear to be in the list (a false-positive). This could occur if the word has the same hash signature as one of the words contained within the list. Due to this possibility of false-positives there is an error probability associated with the use of a Bloom Filter (Du, 2005). See Equations 27 and 28 for the calculation of the error probability (Du, 2005).

$$k = -\frac{\ln 2}{N \cdot \ln(1 - \frac{1}{M})} \approx \ln 2 \cdot \frac{M}{N} \approx 0.69 \cdot \frac{M}{N} \quad (27)$$

$$f(k) = 2^{-k} \quad (28)$$

where:

k is the number of hash functions

N is the size of the list of words

M is the chosen size of the hash table

f(k) is the error probability

Use of a Bloom Filter to Pre-Partition Search Data

One of the ways that a Bloom Filter can be used to partition a list is through the use of "Damerau's reverse edit distance algorithm (Du, 2005)." This algorithm generates all the possible words which are a single error apart from the original search word and then uses the Bloom Filter to determine if any of these words are contained within the list of names (Du, 2005).

The maximum number of words checked (when using the Damerau reverse edit distance algorithm) is always $l(2n + 1) + n - 1$ where l is the size of the alphabet and n is the length of the original word. This number is independent on the size of the word list (Du, 2005). Furthermore, one can easily insert new words to a list but if a word is deleted, the entire Bloom Filter is required to be rebuilt (Du, 2005).

2.3.6 External Criteria

Another method by which one can partition the list of words prior to a fuzzy search, is through filtering the list using external, non-fuzzy matching related criteria. An example of this would be the pre-filtering of a search list based on a particular birth date. This ensures that one only searches for names that are relevant, instead of returning a larger list of names where most of the names will be later rejected as they do not conform to the external requirements.

2.4 Frameworks

Markiewicz and de Lucena (2001) state that "frameworks are application generators that are directly related to a specific domain, i.e., a family of related problems." "A framework is a model of a particular domain or an important aspect thereof. A framework may model any domain, be it a technical domain like distribution or garbage collection, or an application domain like banking or insurance (Riele, 2000)."

2.4.1 Framework Fundamentals

Object Oriented Software Architecture

The object oriented paradigm was born and developed throughout the 1970's and 1980's. However, only in the 1990's was the paradigm absorbed and accepted by the software development community at large (Capretz, 2003).

Karne (1995) states that the fundamental difference between object-oriented programming (OOP) versus conventional programming is as follows: "In conventional programming, the data and control are separated and there is a no easy way to associate and derive data from control. In OOP, the data and control are merged together to form an object, the interface to the object is clearly specified, and in addition, the access to the object can also be controlled by the creator of the object". The object's data and behaviour is defined by means of a class (the object

itself is the instantiation of the class) (Korson & McGregor, 1990). The above is further elaborated through the key concepts of object-oriented programming, namely (Cohen, 1984; Karne, 1995; Korson & McGregor, 1990):

Encapsulation: Encapsulation allows OOP to integrate control and data into an object and thus hides all the details of the object within the object itself. While the data is encapsulated in the object, access to the data is governed by the object's methods or control mechanisms. The benefits of encapsulation are both a reduction in complexity and an additional level of security. The reason for such is that the manipulation of the object's data can only be performed internally or through specified methods.

Inheritance: Inheritance is the ability to derive new classes from other classes. The newly derived classes inherit the properties of the parent classes but can have additional properties beyond those of the parent class. In addition to a reduction in code (as different classes can share a common code base), inheritance allows one to create a general parent class and derive more specialised child classes from the parent.

Abstraction: Abstraction follows on directly from inheritance. Abstraction allows one to define a particular pattern for classes. Thereafter, objects can be declared to be of that particular pattern and inherit all the pattern's attributes. This is achieved through the objects being derived from the abstract class. The abstract class, itself may only provide structure for the pattern but does not implement any of the functionality.

Polymorphism: Polymorphism allows a single method to have many forms and also allows the method invocation to be postponed to Runtime. Thus parent objects are able to invoke derived child object's methods selectively at Runtime. This is due to the fact that all child objects are derived from the parent objects and thus contain (all) the parent's defined methods. Though, the child objects have the same defined functionality as the parent, the implementation of this functionality can differ between various child classes. The ability to bind the appropriate child methods (even though, only the parent object was defined at compile time) at Runtime is integral to polymorphism and is called Dynamic Binding. Dynamic Binding provides polymorphism with its flexibility.

Extensibility: Through the use of abstraction, object oriented programming allows one to extend an existing application through the creation of new classes which are derived from an abstract parent class. This allows one to provide new functionality but still ensures that the newly defined classes conform to the existing framework.

Modularity: Since a class contains both a collection of data and a set of allowable operations (that can be performed on the data), if designed properly, classes should be self-sufficient. The inherent properties of a modular system are weak coupling and strong cohesion. Classes should have weakly coupling, in that they have limited dependencies on each other but should have strong cohesion, in that each class is designed to perform a single function (without multiple classes have overlapping functionality). Through modularity, object oriented

programming allows one to easily change one's code when there is a change in functionality, without impacting on other parts of the application's code.

Framework Design

An Object Orientated Framework is built upon a class model, which describes how several classes interact in order to represent the domain that is being modelled. The inherent nature of a framework, dictates that it is outward facing. It is not intended to be used as a standalone application but rather be a stepping stone on which other applications can be used or be built upon. It is therefore imperative to discuss the framework's classes that interact with the external "world". Before continuing further, it is of the utmost importance to state that the internal framework classes cannot be neglected and must be designed thoroughly as these classes provide the core functionality of the framework. Failing to properly design these classes would mean that the designed framework would be useless.

One can group the sets of classes used to interact with the outside world into three class set types, namely (Riele, 2000):

- The *free role type set* of a framework, are the framework classes that are instantiated by the client classes (these are classes of an external application that is utilising the framework).
- The *built-on class set* of a framework are external classes (from other class models, frameworks, or framework extensions) upon which the framework itself is built. In essence the built-on-class set is the framework's dependencies.
- The *extension-point class set* of a framework are classes that must be inherited in order to implement application specific functionality. These classes provide "hooks" to provide specific functionality in the abstract domain describing framework.

Framework Use

Following from the previous section, the various types of interfacing classes define how frameworks are to be utilised. There are two types of framework clients, namely *use-clients* and *extension clients* (Riele, 2000).

A *use-client* instantiates one or more of the framework's classes and uses these objects for its own purposes. This is achieved through the use of the framework's free role type set of objects. This type of framework is called a black-box framework as the framework can be used as is. The external application developer need not know nor understand the workings of the utilised framework (Riele, 2000).

The advantage of using a framework in such way is that through the instantiation of the framework's own classes (free role type set classes); the framework is able to ensure that the external application's objects behave in a specified manner and thus prevents framework misuse (Riele, 2000).

An *extension client* creates subclasses which are inherited from the framework's extension-point classes. These subclasses allow the extension client to adapt the framework classes according to its specific application needs (Riele, 2000). "A framework that can be extended using sub-classing is called a white-box framework". (Riele, 2000) A framework that utilises this type of client requires that the client's developer be intimately aware and understand the framework's inner workings.

It is necessary to further elaborate on the manner in which a framework utilises its extension-point classes. There are two ways in which a framework can make use of (is coupled to) extension-point classes (Riele, 2000):

- *Coupling using concept specialization*: In this situation, when the framework is required to make use of a particular extension point class, it attempts to

invoke application specific subclasses of the required extension-point class. However, if no such subclass exists, the framework invokes the original extension-point class to implement the required task. The program flow is not affected by the use of either the original extension point class or the inherited subclass.

- *Coupling using callback interface*: When implementing this type of coupling, the framework defines a callback interface through the use of an interface or an abstract class. When the particular extension point class is required to be instantiated (during the course of normal program flow), the framework hands over control to the application specific subclass (which is an implementation of the callback interface) to perform the necessary task. When using this type of coupling, the framework is completely dependent on the client to have instantiated a subclass of the callback interface as the framework itself does not possess the required functionality. Unlike in the case of coupling using concept specialization, program flow is halted if there are no subclasses implemented.

It must be noted that often frameworks cannot be considered as either blackbox or a whitebox only framework but rather these frameworks consist of a combination of both blackbox and whitebox components. These types of frameworks are called greybox frameworks (Riele, 2000).

2.4.2 Advantages of a Framework

The solving of a domain problem is the key advantage of a framework as it allows for both design and code reuse. There are several reasons why design and code reuse is beneficial (Riele, 2000):

- It allows for developers to be more productive and also for a shorter time to market for a new application as core functionality has already been developed. All that is required is for the custom application logic to be implemented.
- Applications that are built upon frameworks tend to have fewer bugs. The reason for such is that the underlying framework is generally a mature technology. Since a framework can be utilised by multiple applications, it is more thoroughly tested as it is exposed to very different scenarios. This exposure leads to a more robust system.
- The use of a single underlying framework allows for a more homogenous end-user experience when using a suite of related applications.
- Furthermore, a framework localises all domain knowledge into a single location and therefore allows applications to be more maintainable.

2.4.3 Disadvantages of a Framework

Though the domain centric (as opposed to problem-centric) approach of a framework is one of its major advantages, it is also one of its greatest disadvantages. There are several reasons for this:

- The development time for a framework is generally far greater and more costly than that of a custom made application. The reason for this is that more analysis is required to cover an entire domain. Additionally, it is far more difficult to design and implement the generic aspects of the framework. Furthermore, the framework must be designed to be extensible so that it is able to cater for future domain requirements (Markiewicz & de Lucena, 2001).

- Frameworks are built for flexibility and generality, at the expense of performance and efficiency (Markiewicz & de Lucena, 2001).
- There is a steep learning curve for third-party developers before they are able to utilise a framework. Time is required to understand the framework, its implementation and its utilisation (Markiewicz & de Lucena, 2001).
- Since a framework is intended to be used by third-party developers (people who were not involved in the framework design and development), there is strong chance that it can be misused. Incorrect usage of the framework could (in the worst case) cause the framework to be utterly useless as it is incapable of performing its specified functionality (Markiewicz & de Lucena, 2001).
- Framework maintenance and upkeep is much more difficult than a custom-built application as (Markiewicz & de Lucena, 2001):
    - A framework is inherently complex (more complex than a stand-alone application) due to its generic nature.
    - Several applications are dependent on the framework. A change in the framework may affect the dependant applications and they too may be required to have some development to cater for framework changes.

Though not a disadvantage in itself, framework documentation is crucial as it informs third party users how to utilise the framework (it is generally not viable for one to be able to contact one of the framework's developers directly). Without the documentation a framework is very difficult to understand and use, if not completely useless. This dependency on up-to-date, highly detailed documentation is the disadvantage of a framework. Further it is of utmost importance that both current and intended framework changes are documented and communicated (Markiewicz & de Lucena, 2001).

2.4.4 Fuzzy Matching Frameworks

The literature review has demonstrated that there are multiple algorithms to address fuzzy matching problems. However, each of these has its own strengths and weaknesses, aimed at solving a particular sub-problem within the greater fuzzy matching domain. As with most concepts within software development, there is no "silver bullet" catch-all algorithm that would be able to duplicate the human ability of fuzzy matching. A more achievable solution is the use of a fuzzy matching framework, in which one is able to swap and change fuzzy matching algorithms as needs arise and change (as is shown in Figure 2.9 below).

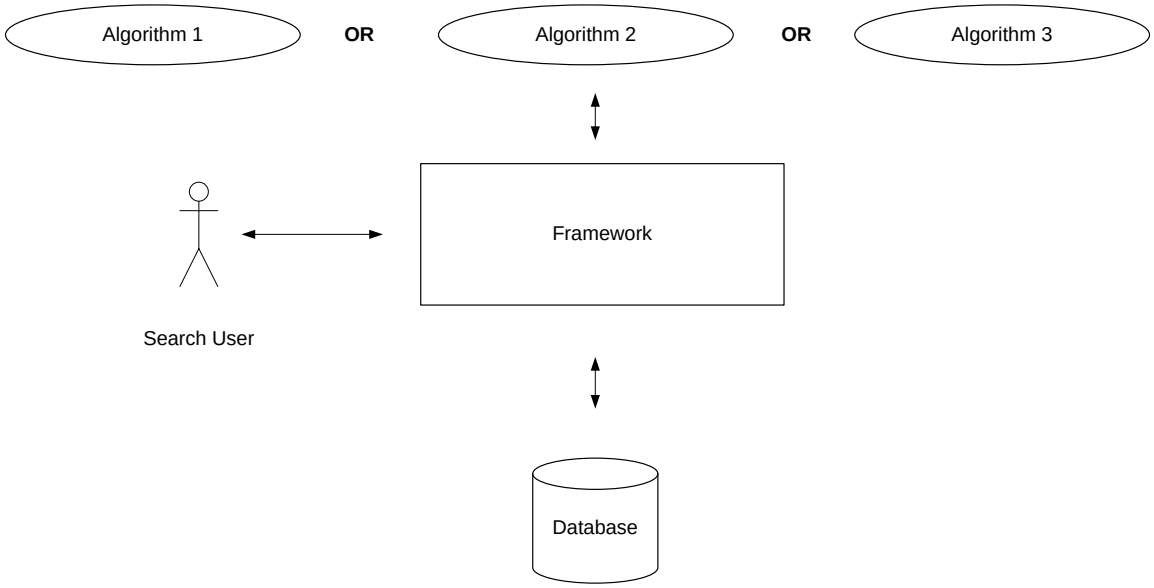


Figure 2.9: A Fuzzy Matching Framework

Furthermore, it has become apparent (through the literature review) that a framework that is capable of implementing the above mentioned functionality does not exist. The remainder of the dissertation discusses the design, implementation and testing of such a framework.

3 Research Question and Methodology Employed

Having completed the literature review it is now possible to contextualise the objective of this dissertation within a research question. Thereafter the methodology employed in the achievement of this objective is described.

3.1 Research Question

The nature of the research topic necessitates that the objectives be defined within a series of research questions:

1. Do the majority of Fuzzy Matching algorithms contain one or more common high level processes that can be integrated into a single generic framework?
2. Is it possible for this generic framework to cater for the requirements of the matching of various combinations of names?
3. Can this framework provide a mechanism for the implementation of custom logic for the common processes?

3.2 Methodology Employed

The following methodology was followed in order to achieve the objectives defined by the research question:

- A literature a review is performed to define the research requirements and to investigate any existing solutions. The following areas are included within the research:
    - Causes of Error in strings and names
    - Fuzzy Matching Algorithms
    - Frameworks
    - Fuzzy Matching Frameworks
- Based on the requirements defined by the literature review, an initial version of the framework is designed and implemented.
- The developed framework is tested through the use of two test methodologies:

    - The framework is subjected to specifically designed test cases. This test methodology verifies whether the framework conforms to the objectives defined the literature review.
    - The framework is deployed as a production system. The deploying of the framework to a real-world business environment moves it out of the realm a proof of concept and forces it to be un-objectively tested, where shortcomings cannot be ignored. Furthermore, this testing methodology exposes the framework to high load, time critical situations.
- The test results are analysed and the reasons for shortcomings are investigated.
- A revised version of the framework that overcomes the weaknesses exposed by the test analysis is designed and implemented.
- The research, design and testing are documented.

4 Design Requirements

Following from the literature review, it has been identified that a generic framework for the matching of similar names has the following high level design requirements:

4.1 Multiple Related Searches

The main aspect that differentiates the fuzzy matching of names from the fuzzy matching of general words is the fact that several names can be related (despite that, they may be each searched against a different set of names) and therefore a search that takes multiple names as an input must be cognisant of this. For example, when searching for a person, one may input the person's first name, middle name and surname. The search would thereafter be required to perform three separate sub-searches where the first name, middle name and surname are searched independently. However, before a result is returned to the user, the search is required to ensure that the results returned by all of the sub searches correspond to the same person and therefore, any results from the three sub searches that do not correspond to the same person may not be returned as a valid result. This example is illustrated in Figure 4.1, where the returned search matches are represented by the area (highlighted in yellow) at the intersection of the three sub searches. It must be noted that the search methodology explained in the example is not necessarily the optimal manner to implement this requirement but is rather used to provide a conceptual understanding of the requirement.

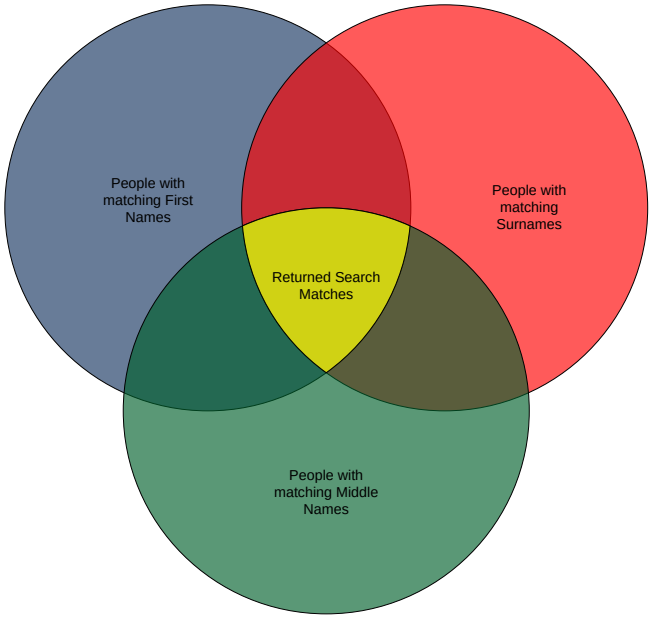


Figure 4.1: Searching multiple related names

4.2 Multiple Fuzzy Matching Algorithms

As discussed previously, many algorithms have been designed in order to cater for different aspects within the Fuzzy Matching domain. The framework must be capable of accommodating the majority of the well-known, well documented algorithms. Ultimately, the framework is required to be generic – not subscribing to a single algorithm and thus independent of the application domain.

In catering for the various types of algorithms, the framework must be capable of accommodating the various requirements of each of the algorithms. This includes the ability to pre-process search names, access a database, post-process a set of matching words and aggregate multiple matches. In doing this, the framework should be capable of implementing future algorithms (assuming that they are implemented in similar manner to the investigated algorithms).

4.3 Abstract the Fuzzy Matching Process

The process of matching two names often requires more than just "plugging" in the two names and then receiving a degree of similarity. Often both pre- and post-processing is required before the degree of the match can be found. (This is discussed in later sections) The framework must ensure that the user need only concentrate and concern himself with key aspects of the name matching process (e.g. algorithm implementation, scoring, aggregation, etc), whereas the framework manages the fuzzy matching process, calling the user implemented aspects and maintaining the data flow.

4.4 User Defined Fuzzy Matching Process

Following from the previous requirement, though the user is only required to implement key aspects of the fuzzy matching process, the user must be able to configure exactly each step in the fuzzy matching process.

Similarly, although the framework is required to be generic, the framework must be capable of implementing domain-specific rules and/or algorithms. For example a user may wish to utilise an exclusion list and/or a substitution list.

4.4.1 Exclusion List

An exclusion list is a list of words, phrases and characters that are to be excluded from a search. The reason for the use of such a list is to remove common words which may not add to the effectiveness of a search but would rather yield false positive matches. For example, words like "and", "company" and "limited" could cause false positive matches as the only common part of the two names are the above mentioned words. Ignoring words like those mentioned above would yield more accurate results.

4.4.2 Substitution List

The substitution list is a list of words that, when found in a name, are to be substituted with the equivalent word from the list. An example of a substitution is: If a name contains the word Ltd, the word Ltd is to be substituted with the word Limited. A substitution list enables the framework to remove the variations that can be found in different names and ensure that all names have a consistent formatting, despite the manner in which they were initially input into the system. Furthermore, the substitution list allows the framework to cater for abbreviations and acronyms.

4.5 External Application Abstraction from the Fuzzy Matching Process

The framework must ensure that any application that utilises it must be completely removed from the fuzzy matching process. To achieve this, the framework must maintain a consistent interface to the outside application regardless of the underlying fuzzy matching process. Furthermore, the fuzzy matching algorithm (and process) should be capable of being changed within the framework without any impact on the external application. This requirement is from a coding / method call perspective as the changing of an algorithm / process could cause the number of returned matches for the same set of search names to vary.

4.6 Performance Requirements

Unlike exact matching, fuzzy matching can potentially be computationally expensive as often one cannot use the search name as is; one may be required to pre-process the search name and the names against which the search name is to be compared, before the search is performed. Furthermore, when performing a fuzzy search, depending on how fuzzy the search is, the returned result set could potentially be very large. As fuzzy matching allows for varying degrees of a match (unlike in exact matching where the name either does or does not match the search name), the framework would

be required to calculate the degree of the match. This process of scoring is one of most computationally expensive aspects of the search as the framework must:

1. Calculate the match score, which is not required for exact matching
2. Repeat this process for a potentially very large result set

Thus, in addition to the functional requirements, the framework has the following performance requirements:

4.6.1 High Speed

Though the framework is to be generic, in order for it to be viable, the time required to perform a fuzzy match using the framework must be comparable to that required for native[8] implementation of the same algorithm. One must bear in mind that the framework may be used to do bulk matches and therefore, even if the time required to perform a single match is acceptable by human measures, the cumulative effect of the batch process could lead to the process being prohibitively long. An acceptable time is deemed to be within one order of magnitude above the time required to perform the corresponding native algorithm match.

4.6.2 Non-Excessive Memory and Processor Utilisation

It would be naive to think that the framework would have low memory and processor utilisation as the fuzzy matching process is inherently memory and processor intensive. However, while the framework is in use, the framework must not monopolise the system resources. Therefore, the framework must contain mechanisms to ensure that only the crucial processes are performed at match time (at the time when search word is being matched against the database), while the majority

[8] The native implementation of an algorithm is considered to be an application that only implements a single fuzzy matching algorithm, unlike the framework which is designed to cater for multiple algorithms.

of the processing is done before hand. In essence the search word should be matched against a pre-processed fuzzy database.

In summary, from the expanded research question (defined in §3.1), six high level requirements for the design of a fuzzy matching framework have been identified, namely:

1. The ability to perform searches on multiple related names.
2. The ability to implement a variety of fuzzy matching algorithms.
3. The abstraction of the fuzzy matching process "plumbing" from the third party user (i.e. the abstraction of common tasks that are required within a fuzzy search but do not add to the matching logic).
4. The ability for the definition and implementation of a custom fuzzy matching process.
5. Abstraction of the fuzzy matching process from external applications.
6. Performance considerations.

With these requirements in mind, the following section describes the design of the fuzzy matching framework.

5 Solution Overview

This section outlines the details of the design of the solution to the research question. Prior to the proposal of a solution, the various decisions that are inherent to the design are discussed, leading to a discussion of the resultant solution. Following from the in-depth description of the solution design, the strengths and shortcomings of the designed fuzzy matching framework are discussed. Finally, a revised version of the design that improves on the identified shortcomings is detailed.

5.1 Design Decisions

It must be stressed that this dissertation does not intend to provide a new fuzzy matching algorithm but rather to provide a platform upon which new (or old) algorithms can be tested and implemented. Thus the core focus of the framework is its usability and accessibility.

5.1.1 Performance Trade-off for Generic Behaviour

As discussed within §2.4, the inherent nature of the framework is that performance is sacrificed in order to provide flexibility and generality. Rather than being optimised for a particular fuzzy matching algorithm, the framework was designed such that it would support a variety of algorithms.

The process of name matching is often not as simple as matching a single name against a collection of single names but rather there can be several permutations within the input set of search names. Generally, when searching for a person's name both the person's first name **and** surname are input as the search names and it is required that the returned result matches to both the first name and surname. However, when searching a juristic entity there is only a single name. Furthermore, in some situations, it may be required that a series of names are input and the results are ORed together; i.e. the union of all of the matches relating to all the search words is returned. Ultimately, the framework is to be capable of enforcing the relationships

between the search words (e.g. the first name search name and the surname search name are related to one another) and therefore, the returned matches that correspond to each of the search words, themselves must be related. However, since there can be any variety of related words, the framework's database cannot enforce a particular structure but rather have a generic structure with the ability to map relationships.

When performing a fuzzy search on a particular name, it may be required that that name be searched against multiple datasets as opposed to just one. For example, one may wish to search against a set containing people's names and a set which contains their aliases. It has been decided that the framework is to allow a user to specify a dataset, against which each input name is to be searched.

All things considered the framework is required to be as flexible as possible (as opposed to supplying a single matching mechanism, to which the user must subscribe) in order that a user can utilise the framework according to an application's needs. As is shown in later sections, the resulting framework design utilises more general data structures and a simple, segmented workflow to achieve this design decision.

5.1.2 Greybox Framework

It has become apparent that the crux of the fuzzy matching framework is that it is flexible enough to be able to implement application specific code but still be capable of abstracting the fuzzy matching process away from the user. Thus it has been decided that the framework is to be implemented as a Greybox framework as described in §2.4.1.

5.1.3 Interfaces

Following from the previous sections, it is important that the user is able to utilise the framework such that it fulfils his / her particular needs. This is achieved through the user implementing the application requirements himself (as it is impossible for the domain specific framework to cater for this). The problem however, is that the framework must be capable of "plugging in" the user code despite the fact the framework is completely unaware of the content of the user code nor was the user code included at the time that the framework was originally built. It is therefore apparent that there must be a generic contract between the framework and the custom user code such that:

1. The custom code is aware of what parameters will be supplied by the framework.
2. The framework is aware of what will be returned by the custom code.

The above mentioned conditions can be achieved through the use of an interface. An interface is used to define a set of behaviours that can be implemented by any class. However, unlike a class, an interface cannot have instance data members nor does it implement any method. Only through being implemented by a class can an interface's methods be implemented (Hu, 2006).

A well designed interface provides a contract (as mentioned above) that has identified the various sets of requirements of the problem domain (large family of abstractions) but is still restrictive enough to ensure concrete realisations of the various applications specific implementations (Hu, 2006). Furthermore, both the method calling the interface and the method implementing the interface are written against the interface and not against each other (in fact neither of the two methods is aware of the other). Thus the method calling the interface is unaffected by any changes that are made to the method implementing the interface (Hu, 2006; Schmolitzky, 2004).

At first glance it appears that the same functionality can be achieved through the use of an abstract class. An abstract class defines but does not necessarily implement a set of methods. Rather, it is the prerogative of the subclasses (that inherit from the abstract class) to implement the functionality defined by the abstract class. Furthermore, any members that are defined by the abstract class will also be inherited by the subclasses (Hu, 2006). The framework and user implemented code can therefore utilise an abstract class in the following manner:

- The framework defines an abstract class.
- The user implemented code inherits from the framework defined abstract class.
- The framework, itself, only knows about the abstract class but through the use of polymorphism, the framework is able to instantiate the user implemented class at Runtime. Thereafter it is able to call unique implementations of the methods that were specified by the abstract class but are implemented in the user implemented subclass.

Although either an abstract class or an interface can be used to enable the framework to implement custom user code, due to the fundamental differences between the two, it has been decided that interfaces are to be used. The rationale is as follows:

The commonality between an abstract class and its subclasses is due to their being related, in that the subclasses are a refinement / specialisation of the original abstract class. For example the abstract class might be an *Employee* class whereas the subclasses might be an *HourlyEmployee* class and a *SalariedEmployee* class. Both the two subclasses are related to the abstract class as they are inherently a type of employee (Hu, 2006).

An interface, however, is used to capture "similarities among unrelated classes without artificially forcing a class relationship (Hu, 2006)". Unlike an abstract class (and its subclasses), classes that implement the same interface need not be similar in

role nor in overall functionality; an interface allows them to provide their own implementation to a well defined common set of functionality (behaviour). One can further elaborate this point, in that interfaces allow polymorphic implementations of a specified behaviour between disparate classes. For example the classes *Lawyer*, *Doctor* and *Student* all inherit from the interface *IWorkable*. Despite that the functionality of the classes are completely different, they all have a common task – in that they *work*. The manner, however, in which they work, is completely different. Though, a lawyer and a doctor work to earn money, the way in which they work is fundamentally different. Furthermore, unlike a lawyer and a doctor, a student does not even work for money but rather to pass a course.

Since the various fuzzy matching algorithms have completely different mechanisms in which they achieve a fuzzy match it would be highly difficult (and impractical) to force them to all inherit from a single abstract class in order that the framework could utilise the polymorphic behaviour. A more pragmatic approach is rather to specify the interface to which the framework subscribes and allow the various fuzzy matching classes to implement that interface. Thus these classes will provide the framework's specified functionality but are not hindered in any other way in which they implement their desired functionality.

In this way, the fuzzy matching framework is capable of implementing a large variety of fuzzy matching algorithms despite their internal matching mechanisms being completely different.

Another option that was investigated but later decided against is the concept of a delegate.

A delegate (found in the .Net framework) performs a similar task to a function pointer (which is found in C and C++), in that it enables one to pass methods as if they were parameters. A delegate defines a method signature by specifying both input

parameters and a return type (Naugler, 2004). A delegate was considered as a potential mechanism to allow the user to implement custom code as the framework could define the delegate and thereafter utilise the delegate when the user code is required to be run. Whereas, the user code would be required to implement the method signature defined by the delegate.

The problem with a delegate however, is that it is very difficult to enforce that the custom user code has implemented the delegate. If a class does not implement a defined delegate, it will still be compiled perfectly and the lack of delegate implementation will only be determined at Runtime when the delegate cannot be found. Furthermore, since the delegate defines a method signature and not the actual method name, it could be quite difficult to determine which method actually conforms to the delegate (at Runtime). However, if a class implements an interface and it does not contain the methods defined by the interface, there will be an error at compile time – alerting the user that the custom code is inadequate.

5.1.4 C# Language and .Net Framework

Before discussing the choice of programming language, it is important to mention that the focus of this dissertation is the design of a fuzzy matching framework and not the implementation thereof, i.e. this dissertation serves as a proof of concept. With the previous statement in mind, the choice of language is not overtly critical and is largely a matter of personal taste. However, there are several reasons for the choice of using the C# language and hence the .Net framework upon which C# runs.

From the previous discussion, it is imperative that the framework give the user as much flexibility as possible when implementing the custom fuzzy matching logic. The .Net platform supports several languages (namely, C#, C++, VB.Net, J#, etc) which all share the same API (application programming interface) and are all compiled to the same Intermediate Language (IL), which in turn is run through the

Common Language Runtime (CLR) (Chappell, 2002; Kachru & Gehringer, 2004). Compared to the .Net framework, the Java Virtual Machine (JVM) is only capable of running the byte-code instructions generated by the Java compiler i.e. only the Java language can be utilised (Kachru & Gehringer, 2004).

The JVM is platform independent as it can run on Windows, Unix, MacOS and Linux, whereas the .Net framework has been designed to run solely on the Windows operating system. However, it must be mentioned that there have been several attempts to allow .Net to have cross-platform implementations; Mono is an open-source implementation of the .Net Framework for the Unix operating system and Microsoft (with Intel and HP) have submitted C# and a subset of the CLR to be standardised under Ecma International (Chappell, 2002; Kachru & Gehringer, 2004). Ecma International is an industry association that is dedicated to the standardisation of Information and Communication Technology and Consumer Electronics (Ecma International, 2009).

Since all .Net languages are compiled to the same IL, they all provide the same functionality and thus, the main difference between them is their syntax. Due to the author's C++ background, the framework itself is written in C# as it has similar syntax to C++ (Chappell, 2002).

5.1.5 Dynamically Loaded DLL's

Though it has been decided that the framework is to be completely flexible to the user's needs and that the user is to implement application specific code through the implementation of the framework's specified interfaces, there is a need to abstract the framework's core logic away from the user. The reason for this decision is that it is not necessary for the user to be able to alter the internal framework code as it controls the greater fuzzy matching process. However, since the core of the framework has

essentially been “locked” away from the user, there is a requirement for some type of mechanism which can implement the user’s application specific code base.

This is achieved by dynamically loading the user’s application specific DLL’s at runtime. In this way the user’s code can be implemented without the core framework needing to be rebuilt / compiled.

Also, through the dynamic loading of DLL’s the framework’s flexibility is increased as the implemented fuzzy matching algorithm can be changed at runtime in order to accommodate changing fuzzy matching requirements.

5.1.6 Use of a Relational Database

Relational databases have become the de facto standard for the storage and retrieval of data for almost every custom business application over the past three decades (Haigh, 2006; Seltzer, 2005).

Compared to previous database models, relational databases have shifted the responsibility of specifying the relationships between data from design time to Runtime; this is achieved through the use of database queries. Furthermore, they are well suited for systems where the content is well defined. An example of such is a payroll administrative system, where every record consists of the same fields, namely: years of service, hourly rate, overtime status, etc. Conversely, relational databases do not fare well for the searching and storing of less rigidly formatted data, such as a full-text record (Haigh, 2006). A full-text query performs linguistic searches against a body of text by operating on words and/or phrases based on the rules of a particular language, for example English or Japanese (MSDN Library, 2009b).

It was concluded that a relational database would be more than adequate for use within the framework, as the content of the framework is well defined; the framework

is to store names and is not to be used to store large bodies of undefined text. Though the framework's database is a relational database, it is assumed that the calling application's database is also a relational database.

5.1.7 Maintain All Search Names within an Internal Database

It was initially considered that the framework would utilise the database of the external application, for which the framework is providing the fuzzy matching functionality. This is advantageous in that the application's database already maintains the relationship between the related fields. For example, a person's first names and surname are two different columns within a single table entry (row).

This type of database configuration poses a problem in that different applications have different database structures and thus it would prove difficult to enable the framework to satisfy the search requirements of various databases; i.e. different fields (within different tables) are required to be searched within different databases.

It was ultimately decided that the framework must maintain its own database, which contains all the search names. The advantage of this type of database configuration is that the framework is able to maintain all the search names in a consistent format, despite the differing search fields and requirements of each application. This in turn allows the framework further flexibility as the names are stored in a generic database and are thus not limited to the relationships imposed by the calling application. This design allows multiple applications to make use of a single instance of the framework as opposed to the implementation of multiple instances of the framework, where each framework instance is dedicated to a single application.

Furthermore, by the framework maintaining its own copy of the search names, there is no threat of the original entries (in the calling database) being modified when the framework processes the names for the fuzzy matching. This ensures data integrity.

5.1.8 Abstraction of the Database away from the Third Party Developer

The core focus of the third party developer is the development/implementation of the fuzzy matching algorithm. The developer need not be concerned with the inner workings of the framework as the framework provides a platform upon which the algorithm is to be run. Therefore, there is no need whatsoever for the developer to have access to the framework's database as this is the lowest level of the framework. Any aspects that the developer may require from the database must be provided through framework supplied interfaces.

5.2 Design Overview

As mentioned previously, it has been deduced that the majority of fuzzy matching algorithms consist of at least one or more of the following three common high level processes - Pre-processing, Database Searching and Match Scoring. With this in mind, the fuzzy matching framework consists of four main components. Combinations of these components are used, in order to perform the framework's two main tasks, namely: fuzzy matching and upkeep of the fuzzy matching database. The components are as follows:

- Name Pre-processing
- Database Storage
- Database Searching
- Match Scoring.

Though the various framework components are themselves distinct, through the use of its own embedded code, the framework is able to seamlessly connect the various combinations of the components in order to achieve the above-mentioned two tasks. The key to the framework's design is its ability (and hence its flexibility) to integrate

third party developer specific implementations to form the core logic of each component. This is achieved through each component providing an interface against which the developer can develop his own custom logic.

Through the use of the Name Pre-processing, Database Searching and Match Scoring components, the framework performs a fuzzy match. Similarly, through the use of the Name Pre-processing and Database Storage components, the framework is able to add new names into the fuzzy matching database. The two figures below display the above mentioned processes.

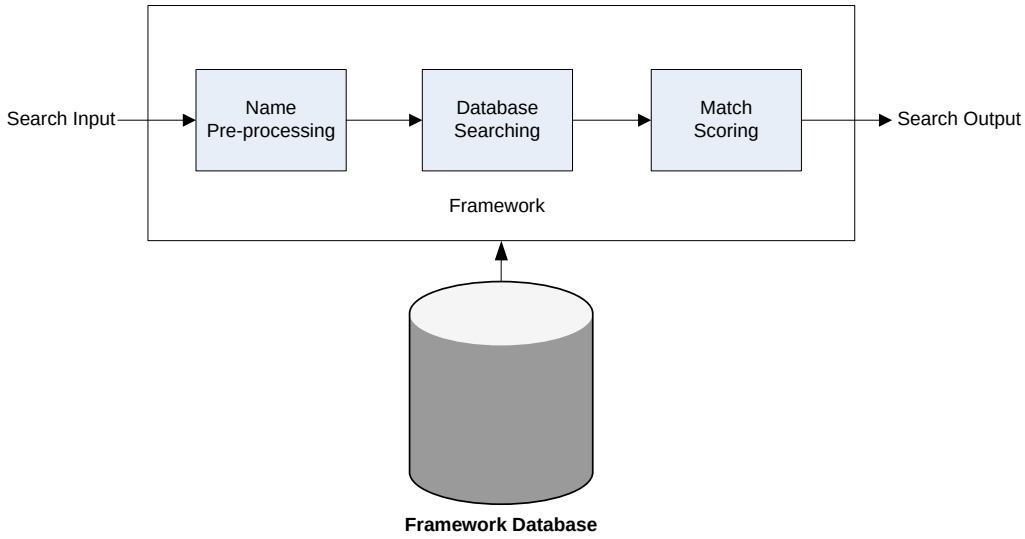


Figure 5.1: High Level Diagram depicting the Fuzzy Matching Process

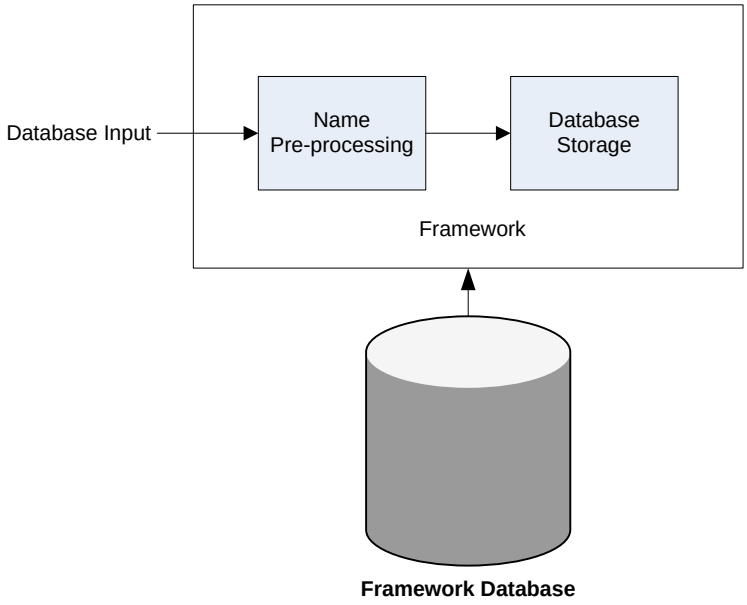


Figure 5.2: High Level Diagram depicting the Upkeep of the Fuzzy Matching Database Process

5.2.1 Name Pre-Processing

The Name Pre-Processing component allows for the input name to be processed in order that it can be converted into the optimum form for the subsequent database search or for database insertion. As discussed previously, the manner in which the input name is processed is dependent on the logic contained within the third party developer code as the framework itself is incapable of performing any processing operations on the input name.

The framework itself manages the pre-processing process by doing the following tasks:

- Locating and loading the specified custom pre-processing operations. The framework has been designed to accommodate multiple custom operations for a particular component as it may be required that several operations are to be performed on the name before the flow of control can be passed on to the next framework component. Each loaded operation must conform to the framework's defined pre-processing interface.

- Running each of the operations in the user specified order and thus ensuring that the output of a former operation is used as the input for the subsequent operation.

Examples of operations that could be performed within the pre-processing component are:

- The division of a multi-worded name into the various components that form the name.
- Capitalisation of a search name.
- Exclusion of various components from a search name.
- Substitution of certain words within a name to other words that are defined within a substitution list.
- The conversion of a name into the equivalent phonetic code (if one were to be using a phonetic algorithm).

5.2.2 Database Searching

Throughout this dissertation it has been discussed that at the heart of the framework's design is the abstraction of the third party developer away from the general aspects of fuzzy matching and allowing him/her to focus on the core logic required for a particular type of fuzzy matching algorithm / process. With this in mind, the actual database search is completely encapsulated within the framework and thus there is no need for the third party developer to be aware or to understand the underlying framework database.

Though the framework manages the database search, the framework still provides an interface through which the third party developer is able to specify the database search condition. Due to the nature of this component (database orientated) the interface itself is designed to be closely aligned to SQL syntax. Through the

implementation of this component's interface, the third party developer is able to specify the following:

- An Exact Match (SQL syntax: =)
- A wild card search (SQL syntax: `LIKE %`)
- Return all name (SQL syntax: `IS NOT NULL`)
- Return no names (SQL syntax: `IS NULL`)

The rationale behind allowing the third party developer code to define the SQL "`WHERE`" clause, is to allow the framework to cater for the requirements of the various fuzzy matching algorithms. Phonetic and exact match algorithms generally pre-process the name in advance and thereafter attempt to find an exact match to the processed word in the database.

Algorithms such as the edit distance and n-grams require the direct comparison of two words (in order to score the degree of match) and therefore the crux of the algorithm implementation is only post the database search. In this situation, the database search component is only used as a mechanism to collate the database names against which the search word will be later scored. Depending on the requirements, the database search could be used as a mechanism to pull out all names from the database or it could be used as a filter. For example one could use the database search to return all names that start with the same letter as the search name (through the use of the SQL wildcard search).

5.2.3 Match Scoring

Though the previous component returns a list of names that match the search criteria, some names are a better match to the search name than others. The Match Scoring component provides a mechanism by which the third party developer is able to implement scoring logic to enable the framework to evaluate the degree of the match

between the matched name/s and the original search name. Through the implementation of this component the third party developer provides the framework with the quasi human intelligence whereby a person is able to identify the best match between multiple non-identical words.

In order to achieve the above high-level functionality, the Match Scoring component is divided into three sub-components, namely:

- Individual Name Match Scoring
- DataSet Aggregation
- Evaluation of Returned Matches

Before discussing the various sub-components, it is necessary to further elaborate on the role of the match scoring component. In §5.2.2, it was discussed that for certain fuzzy matching algorithms, database searching is inadequate for the returning of potential matches but is rather used as a filter. Due to the nature of these algorithms, in that they return a score to describe the degree of the match, the match scoring component provides the ideal place for their implementation.

Individual Name Match Scoring

The Name Match Scoring sub-component provides a mechanism to quantify the degree of the match of a potentially matched name to the original search name. This component is only capable of comparing two names in each operation and hence this operation is required to be called repeatedly for each of the names returned from the database search.

As with previous components, the framework has defined an interface against which a third party developer is able to implement the custom scoring logic.

Search Set Aggregation

A dataset provides a means by which the framework is able to differentiate all the names in the database into logical search sets. For example, through the use of datasets, the framework is able to identify which names (in the database) are first names and which names are surname. (Please refer to §5.3.1 as it discusses datasets in further detail).

Often a fuzzy search is not limited to a single search word as it may be required that one would search on two or more related words within a single search. For example one would search both a first name and a surname in a single search. Furthermore, due to the relationship between the multiple search names from different datasets, it would be inaccurate for the framework to only evaluate each match to each search name in isolation, rather the framework is to evaluate the degree of all potential matches to the search set as a whole. Whereas the previous sub-component scored individual search names and their corresponding potential matches, this sub-component is required to aggregate all the individual search name scores into a single score that represents the entire search set.

It has been found that often some datasets may be considered more important than others. For example, people have several variations on their first names (multiple names, nicknames), whereas there is little room for variation on their surname, therefore it may be required that the surname score be considered more important than the first names score. The framework caters for this through the use of dataset weightings (see §5.3.4).

As with previous components, the framework provides an interface against which the third party developer is able to develop custom aggregation logic.

Evaluation of Returned Matches

Once all the match sets have been scored, the framework by default orders the list of matches in descending order. However, depending on both the search name and the names in the database, the number of matches that are returned by the framework could vary from no matches to thousands of matches (and potentially even more). Generally, it is not viable that a search can return an almost unlimited number of matches.

The framework provides an interface against which a third party developer can implement logic to evaluate the number of matches that are returned by the framework as the results of the search. This for example could be all the matches whose score is above a specified threshold or could be the top (specified) number of matches.

5.2.4 Database Storage

The database storage component stores both new names and updates existing ones. Like the Database Searching component this component is only run after the input names have been pre-processed by the Name Pre-Processing Component. Unlike, the other framework components, this component does not implement any custom logic and therefore does not define any interface. The reason for this, as discussed in §5.2.2, is to abstract the third party developer away from the database.

Since the focus of the framework is to provide a platform upon which various fuzzy matching algorithms are to be implemented, it is irrelevant (to the third party developer, who is actually implementing the algorithms) the manner in which the names are stored, so long that the framework provides a mechanism to accessed them when they are required. This is provided by the Database Search component.

5.3 Initial Design (Version 1)

Following from the above mentioned design decisions, an initial framework design (version 1) is discussed in the following sections.

5.3.1 Database

Please refer to Appendix B for the database model.

High Level Explanation

At the core of the database is the `BaseWord` table. This table maintains a local version (for the framework) of all the names that are to be searched against. Furthermore, as described §5.1.7, all names contained within this table are stored in a single column in order that the database (and therefore the framework) be as generic as possible. Due to the framework being required to interact with an external application and be capable of returning meaningful results; the `BaseWord` table stores, in addition to the name itself, references to the table, column and row from which the name originated.

Since many search words could have originated from the same table (in an external database) and more particularly the same column, the names of the external database table and column are grouped together to form a dataset. All datasets are stored in the `Dataset` table. Returning to the discussion of the fields contained in the `BaseWord` table, the table stores the search name, a reference to the search name's corresponding dataset and the row number of the search name in the originating table within the external application database.

Datasets (as introduced in §5.2.3) fulfil three important roles within the framework:

- They provide a generic means to differentiate names into different search sets. This is achieved through each dataset containing a table name and column name relating to a particular search set within the external application's

database. It must be noted, however, that a dataset cannot store the row number from which the name originated as this is specific to that word whereas the table name and column name are common to the entire search set.

- They provide a generic means to enforce relationships between two or more fields that originated from the same table in the external application's database. This is achieved through all the related datasets, which are referenced by the `BaseWord` entries, referencing the same table name. Therefore by matching on both the table name and the row number (contained within the `BaseWord` entry), the framework is able to determine whether two search names are related entries that originated from the same record in the originating database. An example of such fields would be person's first name and surname. Both entries would correspond to different datasets, as the fields from which they originated are different, but their specified table name and row number would be the same as they originated from the same record.
- They provide a means to maintain relationships between different tables in the external application's database. This enables one to search for information that is related to a root name, without searching the name itself and still be able to return the root name. For example one may want to search a person's alias; however, this alias is contained in a different table to that in which the person's original name (root name) is stored. The relationship between the alias and the person's name is therefore maintained by a reference in the alias table to the person's original name. Through the use of the dataset, the framework is able to return the reference to the original person's name, when a match has been found on the person's alias. This is achieved by the `BaseWord` table containing a field called the "`CoreTableRowNumber`" and the `DataSet` table containing a reference to a table group. The table group is the core search table, in which one is interested, and the `CoreTableRowNumber` is the corresponding row in the core search table. As in the case of the example, the `CoreTableRowNumber` field of alias

name entry in the `BaseWord` table would store the row, in which the corresponding originating name is contained. Furthermore, the dataset referenced by the alias name entry in the `BaseWord` table would contain a reference to the name of the table in which the person's original name is to be found.

As has been discussed previously, it is often required that names be pre-processed before they can be searched against (i.e. provide a meaningful result from a fuzzy search). To this end the database contains a second table (called the `EditedWord` table) that stores the processed names and is the table against which the framework performs its database searches. Each entry in the table consists of a processed name and a reference to its original entry in the `BaseWord` table. It must be noted that multiple entries in the `EditedWord` table can reference the same `BaseWord`. There could be several reasons for this:

- The original name consists of multiple components and has been separated out in order that the search algorithm can easily search on each of these components. For example, the entry in the `BaseWord` table is the "The Bank of England", whereas the `EditedWord` table consists of four entries that reference the original name, namely: "The", "Bank", "of" and "England".
- The pre-processing component has been configured to return multiple variations of a name in order to provide more potential matches to a search name. For example, the pre-processing component may break up a name into its various components, capitalise each of the components and then store all the capitalised components. Thereafter, each of the components are converted into their phonetic equivalent code (e.g. the Soundex algorithm is being implemented) and these codes are in turn stored. When a search is performed against the database, the framework is able to perform both an exact match search and phonetic search.

In order to cater for substitution and exclusion lists (and any other types of similar lists), the database contains a `Rules` table. Each entry in this table contains the following entries:

- A reference to the action (operation) that is to be performed i.e. is a word substitution or exclusion to be performed.
- The search word. If this word is found in a name, the above mentioned action is to be performed.
- The replacement word - If the entry belongs to a substitution list, the replacement word is the word that is to replace the found instance of the search word.
- A penalty. Since the use of an exclusion or substitution list, alters the original name (i.e. the name could be inadvertently changed to an unrelated but similar name), the user may wish to penalise the search for the performing of such operations.

Triggers

Several mechanisms for the input of search names from the external application into the framework's database were investigated. One of the mechanisms that was tried and implemented is an exposed framework method that can be called by the external application. Through the calling of this method the application is able to pass a name from its own database to the framework. In turn, the framework processes this name and places it into its own database, thereby including the name in future searches. However, this method was found to be cumbersome when the calling application contains thousands of names that are updated daily.

It was realised that it would be more efficient for the insertion, updating and deletion of names in the framework database to be managed at a database level; therefore when the external application's searched name table is modified (i.e. an insertion, update or deletion), the framework's `BaseWord` is correspondingly modified. In the

prototype, this was achieved through the use of triggers as both the external application and the framework shared the same database. The maintenance of the search names in prototype framework's database is achieved in the following manner:

- Upon insertion of a name into a "search" table in the external application's database, the associated insert trigger for that table, then inserts that name and its corresponding details (namely, `DataSet`, "`RowNumber`" and "`CoreTableRowNumber`") into the framework's `BaseWord` table. However, an additional field in the `BaseWord` table is also populated – the `WordChangeType` reference. This reference specifies what operation has been performed on the name entry (i.e. the word has been newly inserted into the `BaseWord` table), enabling the framework to perform the necessary maintenance (see §5.3.3). Therefore, in the case of an insertion into the `BaseWord` table, the `WordChangeType` references an insert operation.
- Upon the updating of a name in a "search" table in the external application's database, the associated update trigger for the table, searches for an entry in the `BaseWord` table that has the same row number and dataset as the name that has just been updated. If the name is found in the `BaseWord` table, the actual name itself (i.e. the name field) is updated to the new Name value and the `WordChangeType` reference is set to update.
- Upon the deletion of a name in a "search" table in the external application's database, the associated delete trigger for the table, searches for an entry in the `BaseWord` table that has the same row number and dataset as the name that has just been deleted. If the name is found in the `BaseWord` table the `WordChangeType` reference for the name is set to delete.

5.3.2 Initialisation

Upon the framework start-up, the `Rules` table (i.e. the Exclusion and Substitution lists) is imported into memory. The reason these values are kept in memory as opposed to their being retrieved from the database when they are required is twofold:

- One of the design decisions is that the third party developer may not have access to the database; therefore the framework must provide these lists in case they are required in a particular fuzzy matching solution.
- Once configured, these lists generally remain quite stagnant, therefore, it would be a waste of time and system resources if each fuzzy search performed by the framework, would re-retrieve the same lists from the database.

These in-memory lists can be distributed for use by any third party developer's code.

5.3.3 Database Maintenance

§5.3.1 discussed how triggers are used in the prototype to import names into the framework's database, however it is not adequate for names to be merely imported into the `BaseWord` table as this is not the table used for fuzzy searches. These names are required to be placed into the `EditedWord` table. It is however required that all the names in the `EditedWord` table are processed in order that they can be searched against. The framework maintains both the `BaseWord` and `EditedWord` tables through a process that is repeatedly run every minute or so[9]. The following tasks are performed as part of the database maintenance process:

- Any entries in the `BaseWord` table that have a `WordChangeType` associated with them are retrieved.
- The process then iterates through each name retrieved in the previous step.
    - If the `WordChangeType` reference is an insert, the name contained within the `BaseWord` entry is input into the Pre-Processing

[9] This time period can be pre-configured by the framework administrator

framework component (§5.3.6) and the output of this component is then stored in the `EditedWord` table. The `WordChangeType` reference associated with the `BaseWord` is now removed.

- If the `WordChangeType` reference is an update, all the entries in the `EditedWord` table, associated with the `BaseWord` entry are removed. Thereafter, the same steps are performed as those performed when a new name is inserted into the database.
- If the `WordChangeType` reference is a delete, all the entries in the `EditedWord` table, associated with the `BaseWord` entry are removed. Thereafter, the entry in the `BaseWord` table is also removed.

5.3.4 Search Input

The search input has been designed in order that one can perform multiple unrelated searches by making a single call to the framework. Furthermore, the search input also allows one to group related names into a single search, for example one would logically group together a person's first name and surname. These names can be grouped together despite that each name would be searched against a different datasets.

The rest of this section describes and explains the input XML packet (the xml schema for the input packet can be found in Appendix C).

At the highest level, the search input consists of one or more Search Sets. Each search set represents a distinct search and the results of each are logically ORed together. As discussed previously, one may require the searching of two or more names that are related but are to be searched against different search sets. This is accommodated by each search sets consisting of one or more Word Set Components. Each Word Set Component consists of a single name that is to be searched within the framework.

The results of all the Word Set Components within a single Search Set are logically ANDed together as they are related to one another.

A Word Set Component provides the framework with both the search name and the corresponding search instructions for that particular name. A Word Set Component consists of the following:

- The search name
- The dataset (in the database) against which the name is to be searched.
- A weighting for that name. This is used when calculating the overall score for a particular Search Set[10], as the weighting provides a mechanism for the user to assign levels of importance to the various names within a search. For example one may feel that the results returned for a person's surname are to be considered more important than those returned for that person's first name.

5.3.5 Search Initialisation

Prior to a search being performed, the framework reads in the list of user defined assemblies (DLL's) and (if specified) particular classes within these assemblies from the framework's configuration file.

The framework's configuration file contains multiple sections in which the custom user code for each of the framework's components is specified. At the very least a user must supply the path of an assembly file that contains at least one implementation of the interface defined for that framework component. In turn, when the framework runs the user code, it searches for all the classes that implement the interface that is defined for the particular framework component. If the user however,

[10] Since a Search Set corresponds to a set of related names, it is inaccurate for one to return several match scores for each of the Word Set Components within the Search Set, as this does not take into account the relationships between the various Word Set Components (one of the word set components could return a high scoring match that does not relate to any of the results returned by the other word set components). Rather, only a single score may be returned for each search set, which takes into account all the scores of the constituting Word Set Components.

wants to run a particular class within that assembly, that class must be specified in the configuration file. If the user requires multiple classes from an assembly to be implemented, the user must specify a comma delimited list of the class names. It must be noted that if the user specifies exactly which classes are to be run, the framework will implement (and hence instantiate) the classes in the order that they are specified, however if the user just defines an assembly and no classes within that assembly, one cannot be sure in which order the classes contained within that assembly will be run.

In Figure 5.3 an example of the assembly configuration section within the framework's configuration file is provided. If one looks at the "`DataTransformConfig`" section (the section that relates the configuration information for the framework's Pre-Processing component), one is able to see an example of how both an assembly and list of specified classes would be configured. The "`SearchParameterConfig`" section, however, displays how one may supply the assembly name, without specifying particular classes.

```
<AssemblyConfig>
  <DataTransformConfig>
    <AssemblyFile>Z:\FuzzyMatching.dll</AssemblyFile>
    <ClassName>StripPunctuation;SeparateNames;Capitalise;StripSingleLetters;
       ExclusionListImplementation</ClassName>
  </DataTransformConfig>
  <SearchParameterConfig>
    <AssemblyFile>Z:\FuzzyMatching.dll</AssemblyFile>
    <ClassName></ClassName>
  </SearchParameterConfig>
  <SearchResultScoringConfig>
    <SearchResultScoringSteps>
      <AssemblyFile>Z:\FuzzyMatching.dll</AssemblyFile>
      <ClassName>SingleScoreCalculationMethodNoPhonetic</ClassName>
    </SearchResultScoringSteps>
  </SearchResultScoringConfig>
  <DataSetScoreAggregationConfig>
    <AssemblyFile>Z:\FuzzyMatching.dll</AssemblyFile>
    <ClassName></ClassName>
  </DataSetScoreAggregationConfig>
  <MatchEvaluationConfig>
    <AssemblyFile>Z:\FuzzyMatching.dll</AssemblyFile>
    <ClassName></ClassName>
  </MatchEvaluationConfig>
</AssemblyConfig>
```

Figure 5.3: Example of the Assembly configuration section in the Framework's Configuration file

5.3.6 Pre-processing

This component deals with the processing and manipulation of a name before it is either queried against the database or is stored within it. The nature of pre-processing is to ensure that the name is in its optimum form before any further work is performed on it. Some of the reasons why this is performed is in order to ensure that the name is converted to the format required by the fuzzy matching algorithm (e.g. conversion to codes for phonetic matching), that the name is altered in order that it can return more matches to the search phase (e.g. the use of exclusion and substitution lists) and that all names conform to a uniform format (e.g. substitution list).

As has been discussed at length, the framework itself contains no fuzzy matching specific logic; rather it provides a platform upon which the custom third party code is implemented. The pre-processing component applies each of the pre-processing operations that are supplied in the `DataTransformConfig` section of the framework's configuration file to each of the input search names (these operations were read into the framework within the search initialisation step - §5.3.5). These operations are run sequentially[11] in order that the output of the first specified pre-processing operation becomes the input for the following specified operation. This enables a previous operation to affect a subsequent operation. For example, a user may have defined two operations; one operation that removes all excluded words from a name and second one that capitalises all entries in the name[12].

If no tasks have been configured for the pre-processing component, the framework will utilise the input names "as is" in the subsequent component.

IDataTransformationInterface Interface

All the third party developer classes that are specified within the `DataTransformConfig` section of the configuration file must implement the `IDataTransformationInterface` interface (the interface definitions have been supplied in Appendix D). This interface defines the `TransformedInputData` method, which takes in a `SearchWordClass` packet and a list of `DataSetID's` as input parameters and returns a `SearchWordClass` packet. The Search Word Class packet contains both the original input search name

[11] No need could be found to provide a means to allow two or more pre-processing operations to be run in parallel.

[12] The intention of this design is for the third party developer to construct a library of common functions that can be used in multiple fuzzy matching algorithms. This would be implemented through the "chopping and changing" of these various library components to form the various configurations required for each of the various fuzzy matching algorithms. This type of development model enforces code reuse and hence greater flexibility as the developer has many tools at his/her disposal and is able to adjust quickly to changes in requirements, without having to redevelop any code.

and a list of processed names. The items in this list serve as the search names for the Database Search component.

The reason the Search Word Class contains a list of names as opposed to just a single database search name, is to provide the user with the ability to convert the search name into multiple forms. An example explaining this requirement is as follows: the search name is the "The Bank of England" and the required fuzzy matching algorithm logic is that the framework is to search on each of the components of the name. Therefore, after the implementation of the custom third party pre-processing code, the input search name would be transformed into the following list of processed names – "The", "Bank", "of" and "England". Thereafter, each name in the list can be searched independently. Another example would be that the algorithm implementer requires that the framework search on the search name in both its original form and also in its phonetic equivalent (its phonetic code). Therefore, the output of the pre-processing component is a list containing both the original word and the phonetic equivalent (for each of the input search names).

The list of `DataSetID's` that is passed to this method is the list that was supplied with the originally input Word Set Component. The inclusion of the `DataSetID's` into the `TransformedInputData` method enables the third party developer to perform dataset specific processing.

5.3.7 Storage

This component is utilised when new names are submitted to the framework to be inserted into the framework's database or when existing names in the framework's database are required to be updated. The previous component is utilised in order to transform the input name into the optimum form to be searched against. Each item that was return in the list from the pre-processing component is stored by the

Framework as a new entry in the "`EditedWord`" table. In addition to the processed name, a reference to the original unedited word in the `BaseWord` table is also stored.

This component has no associated interfaces as the manner in which the framework persists the search names is immaterial to the third party user and hence there is no need for user involvement.

5.3.8 Search

The search component is similar to the storage component in that it takes in the output list from the pre-processing component and uses it to interact with the framework's database. Unlike the Storage component, the Search component attempts to find matches in the database that correspond to each of the words in the edited name list.

Through the use of the `ISearchParameterInterface` Interface, the third party developer is able to specify the database search criteria (please refer to §5.2.2) for each of the members in the edited word list. The `ISearchParameterInterface` defines the `SetDatabaseSearchCondition` method; as the name describes, this method is used to specify the SQL-like search conditions to be used in the database search. Both the Search Word and the list of Data Sets against which the word is to be searched are input into this method, while a `DatabaseSearchParameters` object is returned. The `DatabaseSearchParameters` consists of the search term, a SQL clause enumerated type and the list of Data Sets against which the word is to be searched. An example of the use of this method is as follows, the user wishes to perform a wildcard search, which searches for all names that start with the letter "B". In the pre-processing component, the search name would have processed in order that all that is to be search is the letter "B". However, it is the responsibility of the

`SetDatabaseSearchCondition` method to enforce the wildcard search. Example inputs and output for this method are demonstrated in Table 5.1.

Table 5.1: Example Inputs and Output for Wildcard search SetDatabaseSearchCondition method

Inputs	**Output (DatabaseSearchParameters object)**
SearchWord: "B"	Search Term: "B%"
DataSetID: 1	SQL Clause: like
	DataSet: 1

The main complexity of this component is the actual database search itself. The framework is required to maintain the relationships specified by each of the various Search Sets within the search input while it searches against a generic database. As described in §5.3.1, in order to ensure that the database is generic, all the search names (the processed names against which the framework searches) are contained in a single column in the `EditedWord` table and the relationships are persisted through the use of the associated datasets and core-table row numbers. Furthermore, the framework allows for the input of an unlimited number of word set components to be included within each search set and therefore must be capable of dynamically enforcing all of these "and" relationships within the database search.

Since, at design time, one does not know how many "and" relationships will be specified within each of the framework's search requests, the framework's search query is designed that it is constructed at run-time by taking account of all of the word set components in each of the search sets. In order to achieve this functionality, the framework builds multiple sub-queries corresponding to each of the word set components within the input search set. These sub-queries define the database search specific to the corresponding word set component. Thereafter, the database search query is composed, through the concatenation of all of the previously built up sub-

queries. The joins between these sub-queries are enforced by ensuring that both the table group and core table search row are the same for each of the sub-queries.

The example below demonstrates the query that would be generated for an input search name of “John Mark Smith”, where the pre-processed name components are capitalised and converted into their equivalent Soundex Codes.

```
<SearchSets>
        <WordSetComponents>
                <Word>John</Word>
                <DataSetID IdString = "1"/>
                <WordWeighting>100</WordWeighting>
        </WordSetComponents>
        <WordSetComponents>
                <Word>Mark</Word>
                <DataSetID IdString = "2"/>
                <WordWeighting>100</WordWeighting>
        </WordSetComponents>
        <WordSetComponents>
                <Word>Smith</Word>
                <DataSetID IdString = "3"/>
                <WordWeighting>100</WordWeighting>
        </WordSetComponents>
</SearchSets>
```

Figure 5.4: Search Input for name John Mark Smith

```
select q1.word as Word1, q1.datasetid as DataSetID1,
       q2.word as Word2, q2.datasetid as DataSetID2,
       q3.word as Word3, q3.datasetid as DataSetID3,
       q1.coresearchtablename, q1.coretablerownumber
from
(
       select b.word, b.datasetid, b.coretablerownumber, d.tablegroupid,
       g.coresearchtablename
       from fmf_editedword e
       join fmf_baseword b on e.basewordid = b.basewordid
       join fmf_dataset d on d.datasetid = b.datasetid
       join fmf_tablegroup g on g.tablegroupid = d.tablegroupid
       where 1 = 1
       and b.DataSetID = 1 and (e.text = 'JOHN' or e.text = 'J500')
) q1,
(
       select b.word, b.datasetid, b.coretablerownumber, d.tablegroupid,
       g.coresearchtablename
       from fmf_editedword e
       join fmf_baseword b on e.basewordid = b.basewordid
       join fmf_dataset d on d.datasetid = b.datasetid
       join fmf_tablegroup g on g.tablegroupid = d.tablegroupid
       where 1 = 1
       and b.DataSetID = 2 and (e.text = 'MARK' or e.text = 'M620')
) q2,
(
       select b.word, b.datasetid, b.coretablerownumber, d.tablegroupid,
       g.coresearchtablename
       from fmf_editedword e
       join fmf_baseword b on e.basewordid = b.basewordid
       join fmf_dataset d on d.datasetid = b.datasetid
       join fmf_tablegroup g on g.tablegroupid = d.tablegroupid
       where 1 = 1
       and b.DataSetID = 3 and (e.text = 'SMITH' or e.text = 'S530')
) q3
where 1=1
and q1.tablegroupid = q2.tablegroupid and q1.coretablerownumber =
q2.coretablerownumber
and q2.tablegroupid = q3.tablegroupid and q3.coretablerownumber =
q3.coretablerownumber
```

Figure 5.5: Generated SQL query for search name John Mark Smith

The results of the search are thereafter organised, in order that each Search Set is associated with a list of corresponding matches. The matches themselves are organised to ensure that each match consists of a list of the matches to the individual Word Set Components, i.e. the first entry in the match list corresponds to the Search Set's first Word Set Component, the second entry in the match list corresponds to the Search Set's second Word Set Component, etc.

5.3.9 Word Set Component Scoring

This component, like some of the other framework components, contains no inherent logic within the framework but rather depends on its execution of the third party developer code to perform the match scoring. As discussed in §5.2.3, this component handles the scoring on a Word Set Component level. In order to achieve this, the framework iterates through each of the matches (returned from the Search Component). Furthermore, within each of the matches the framework iterates through each of the Word Set Components. At this point, the framework calls the third party developer code to evaluate the degree of the match between the original search Word Set Component and the corresponding match returned from the database search. The degree of the match is quantified by a score.

The list of third party assemblies and classes called by this component is stored within the `SearchResultScoringConfig` section of the framework's configuration file. This component is designed in order that if more than one scoring method is defined in the configuration file, it will call each of the scoring methods sequentially.

The `ISearchResultScoringInterface`, which the third party developer code implements, contains the `ScoreMatchResult` method. The method is defined in that it is both passed and returns a `MatchScoring` packet (the `MatchScoring` packet consists of the search name, the database match name, the dataset from which

the match name originates and the match score). The reason that the method does not define just the returning of a score but rather the whole packet is to allow both the processing performed and the calculated score from a previous scoring method to affect the subsequent scoring methods[13]. With this in mind the framework passes the `MatchScoring` packet that was returned from the previous scoring method as an input parameter to the subsequent scoring method. The `MatchScoring` component that is passed into the first scoring method consists of the original unedited search name, the database match name, the match's dataset and a score of zero. As is in the case of previous components, the reason the dataset is provided to the `ScoreMatchResult` method is to enable it to perform, if required, dataset specific processing.

An example explaining the design of the scoring component is as follows:

The user may wish to evaluate the degree of the match by calculating the percentage of the number of common words between the original search name and the database match, however, it is also required that all words within the exclusion be stripped out of the name and the score be penalised for each word that is removed. This can be achieved through the implementation of two scoring methods, one method that removes the excluded words from both the search name and the match name, and then applies a penalty to the score for each word that is removed. The second method then determines the percentage of common words between the edited search and match names (both these names have been stripped of the words that are contained within the exclusion list) and adds its calculated score to the already penalised score.

[13] As in the case of the Pre-Processing component, through the manner in which the framework implements the third party developer code, it is hoped that the developer constructs a library of common functions that can be "chopped and changed" depending on the fuzzy matching algorithm's requirements.

5.3.10 Search Set Score Aggregation

As has been discussed at length, the function of the search set score aggregation component, is to provide a single score that is representative of the entire search set by taking into account each of the individual scores of the search set's Word Set Components. The framework itself does not contain any of this logic and is again reliant on the implementation of the third party developer code.

This component is only capable of implementing one third party developer method. The reason for this design decision is that there is no need for multiple successively-called methods – a single method should be more than adequate to collate the results of the individual Word Set Components. This method is specified in the `DataSetScoreAggregationConfig` section of the framework's configuration file.

The component's interface (the `IDataSetScoreAggregationInterface`) defines the `AggregateScores` method which takes in a list of match scoring information packets and returns a double (the score). The list of match scoring information packets contains the scoring information for each of the word set components contained within a search set. The scoring information contained in a match scoring information packet is as follows:

- The score (for that Word Set Component)
- The dataset of the Word Set Component (in order to implement dataset specific logic)
- The weighting of that word set component (refer to §5.3.2)

5.3.11 Match Evaluation

The Match Evaluation component dictates which of the search results are actually returned by the framework to the external application. As the framework leaves all of the fuzzy matching logic to be implemented by the third party developer code, the

framework contains no inherent logic within this component. It is however assumed that since it is often not feasible to return all the results of the search, one would only return the top number of matches and therefore, prior to framework's instantiation of the third party developer code, the framework orders the search results in descending according to their scores.

Like the Search Set Score Aggregation component, the framework has been designed to only instantiate a single third party developer method, as there is no need for a cumulative effect on the final result (as is required in the PreProcessing and Word Set Component Scoring components).

The `IMatchEvaluationInterface` interface (the interface used by the Match Evaluation component) defines the `EvaluateReturnList` method. This method's input parameter is a list of all the search results' scores. The method returns a list of the indices of the matches that are to be returned by the framework. The input list of search results is not the entire match list but rather a separate list of the match's scores where each entry in the score list corresponds to actual match in the match list.

Once the third party developer method has been executed, the framework selects only the matches whose indices were specified by the third party developer's method and adds them to the framework's matches return list.

Below is an example of how the Match Evaluation component would process the list of search results, when the third party developer method is designed to only return the matches that have a score of 70 or more:

The Match Evaluation component is input the following list search results list.

Table 5.2: List of Search Results input in Match Evaluation Component

Match
Match 1 (Score 65)
Match 2 (Score 70)
Match 3 (Score 69)
Match 4 (Score 80)
Match 5 (Score 90)

Prior to the calling of the third party developer method, the search results are ordered in descending order of score.

Table 5.3: Ordered List of Search Results

Match
Match 5 (Score 90)
Match 4 (Score 80)
Match 2 (Score 70)
Match 3 (Score 69)
Match 1 (Score 65)

The scoring list that is input into the third party developer method is:

Table 5.4: List of Match Scores input into the Third Party Developer method

Score
90
80
70
69
65

The third party developer method returns the following index list.

Table 5.5: Index list of results returned by the third party developer method

Index
0
1
2

The final output of the Match Evaluation component is the following:

Table 5.6: Search Result List return by the Match Evaluation Components

Match
Match 5 (Score 90)
Match 4 (Score 80)
Match 2 (Score 70)

5.3.12 Search Output

After the evaluation of the search results, the fuzzy search is complete and the framework returns the results.

The response of the framework's search method is an array of `ResultSets` (please refer to Appendix E regarding the output XML schema). The output Result Sets correspond directly to the input Search Sets, where each Result Set contains the search result for the original Search Set. It is assumed that there will always be at least one input Search Set and therefore there will always be a minimum of one Result Set returned by the framework.

The remainder of this section discusses the structure of the Result Sets.

Each Result Set contains a DataSet Number and the list of Data Set Search Results. A DataSet Number must not be confused with a DataSet, rather it is merely a number, used to define the Search Set to which the Result Set corresponds. Each Data Set Search Result contains information relating to a match that was found by the framework when performing the fuzzy search for a particular Search Set.

The following is contained in each Data Set Search Result:

- The Word Set Components that compromise the matched name, which consist of the combination of matches to each of the input Word Set Components Each output Word Set Component consists of the following:
    - The Name
    - The dataset to which it is associated
- The match's score (the degree of match to the input Search Set).
- The following two items are used to locate the match within the external database (from which the name originated):
    - The Table Name

- The Row Number. This is the primary key of the row within the previously mentioned table, where the match entry is to be found.

5.4 Design Strengths

5.4.1 Support of multiple Fuzzy Matching Paradigms

Due to the compartmentalised framework design, each of the framework's components performs one of the identified common tasks required by fuzzy matching algorithms. Furthermore, as discussed in the subsequent section, the framework contains no logic itself; rather it relies on the third party developer code to supply all algorithmic logic. In this way, if a particular fuzzy matching algorithm requires no functionality to be performed within a specific component, the framework inherently does not enforce any logic. In this situation, all that is required by the third party developer is to implement placeholder code that does not implement any logic but rather conforms to the framework's interfaces. To this end, the framework is capable of implementing almost any algorithm.

5.4.2 All logic is supplied by the Third Party Developer

The framework does not dictate an algorithm's logic and therefore any logic that is contained within a fuzzy matching search has been implemented by a third party developer. Through this characteristic, the framework is capable of implementing multiple different algorithms (as discussed in the previous section).

5.4.3 Abstraction of the Fuzzy Matching Process

Through the framework's design, the onus of maintaining the fuzzy matching database and the management of both the search name/s and match name/s during the fuzzy matching process is removed away from the external third party developer. As has been discussed constantly throughout this dissertation, the abstraction of the nuts

and bolts of the framework enables the third party developer to focus solely on the fuzzy matching algorithm logic, without needing to focus on the aspects of the fuzzy matching process. In addition, abstraction limits the amount of "meddling" that can be done by external parties and enables the framework to be more robust as it entry points are well defined.

5.4.4 Runtime Implementation of Third Party Logic

The framework's ability to incorporate the user logic at Runtime, allows the third party developer to "chop and change" fuzzy matching algorithms at will without having to recompile the framework. This enables the framework to be highly flexible because it can easily adapt to changing requirements. It can facilitate the testing of various fuzzy matching algorithms or the determination of the optimum algorithm for a particular requirement set.

There are two advantages to the framework not needing to be recompiled when the third party fuzzy matching logic is changed:

1. Development lead time is minimised, especially when the fuzzy matching algorithm requirements are frequently changing as one needs only to concentrate on the development of the required fuzzy matching logic.
2. Distribution and deployment of the framework is easier. Since the framework, does not need to be recompiled, one can simply deploy the framework's binary, which in turn shields the third party user from having to struggle with the building and compilation of the framework (please see the previous section).

5.4.5 Generic

The framework's flexibility is due to its generic nature. There are several aspects to the framework's design that enables to it to be generic:

- The framework does not dictate the number of names that are to be searched simultaneously within a single search; rather it is the third party developer's prerogative to define what names are to be input simultaneously.
- The framework also does not dictate the relationship between the input search words, i.e. the framework does not specify that one must input both a first name and surname in every search. Rather (as explained in §5.3.4) the framework utilises the relationships specified (by the calling the application) in the search input. The framework's generic outlook on relationships is further reiterated in the design of its database. As been explained previously, all names are stored in a single database field and relationships are defined by common table names and table row numbers. The upkeep of datasets (where the various table names are defined) is maintained by the framework administrator.
- Though already discussed in depth, the framework's lack of subscription to a particular fuzzy matching algorithm further testifies to its generic nature.

5.4.6 Capable of Maintaining Relationships

Despite the generic nature of the framework, it is able to maintain the relationships between the various names as defined by the external applications from which the names originate. This is achieved through both the design of the framework's search input (as all names that compose a search set are ANDed together) and the use of datasets.

5.4.7 Capable of Serving Multiple Applications

Through the use of the different datasets, the framework is capable of serving multiple applications simultaneously. Datasets enable the framework to logically separate the data from the various external applications and furthermore ensure that the different search data does not get "mixed up" (despite the database's storing of the names in a single field). Another advantage in using datasets is that the framework is capable of maintaining the relationships between the different types of names, which is inherent to the source application. The framework's design enables it to "support" new applications without the need to be rebuilt; rather the newly supported application must be configured in the framework's database, through the following:

- Set up of the application's Data Sets.
- Import and processing of the application's search data.
- Set up an update mechanism (programmatically or through database triggers) to ensure that the framework's version of the search data is in line with application's search data.

Another aspect of the framework's design that has not yet been discussed as it is not relevant within the context of the specified design requirements (§4), is that the framework has been implemented as a web service. A Web service is a self-contained, modular business application that has a published interface that can be invoked across the Internet. It interacts and integrates with other loosely coupled, distributed applications through the exchange of XML-based messages exchanged via Internet-based protocols (Alonso et al., 2004). This enables the framework to serve multiple remote applications simultaneously.

5.5 Design Shortcomings

5.5.1 Expensive Database Search

Despite the framework's generic nature being one of its core strengths, it is also one of its main shortcomings. Unfortunately, in achieving a generic solution, optimisation and speed is lost as the framework must be able to cater for the various different types of fuzzy searches. This trade-off between performance and flexibility is particularly highlighted within the database search. Since the database stores all names in a single column (as discussed in §5.3.1), the database search (see §5.3.8) creates and performs a sub-query for each input search name and only once the results of all the sub-queries are returned are the relationships between the various datasets enforced. This search mechanism is highly inefficient as generally the majority of the returned sub-query results will be discarded as they do not have corresponding entries across all of the sub-queries. Ultimately this results in the retrieval of a large number of irrelevant names for each fuzzy search.

5.5.2 Unable to Pre-Filter based upon Non-Name Related Requirements

Due to the very nature of Fuzzy Matching (the searches are "Fuzzy"); the database search can return potentially thousands of results. The effects of this are twofold: each of the potential matches thereafter is required to be scored, which greatly increases the fuzzy matches' processing time and the framework could potentially return a large number of matches. Though the probability of false positives remains the same despite the size of the returned list, the increased size of a large result set causes a larger number of false positive matches. It is therefore a requirement that the framework be able to pre-filter the potential matches in order that the framework search is performed against a smaller search set. Furthermore, it may be required that the potential matches be pre-filtered based on a non-name related requirement, for example the date when the name was submitted into the database.

5.5.3 Main Reliance on Triggers to update the BaseWord table

Though the framework does supply methods (as discussed §5.3.1), which the application can call to insert / update/ delete entries in the `BaseWord` table, triggers are the preferred mechanism to edit the `BaseWord` table. The use of triggers has several complications:

1. An external user is required to understand the framework's database – at the very least the `BaseWord` table and all other tables that it references must be understood.
2. It is complex to trigger an action in one database from an event in another database. It is even more difficult to utilise triggers between databases that utilise different database management systems – for example, triggering an action in a MS SQL database based on an event in an Oracle database.

5.5.4 Caching of Rules Table

Upon the framework's initialisation, the contents of the Rules Table are loaded into the domain cache. The rationale behind this decision is to provide the framework with the ability to supply the third party developer code the contents of the `Rules` Table but still abstracting the framework's database away from the code. The problem, however, is that over time the contents of domain cache are overwritten with other content, effectively removing the `Rules` Table from memory.

5.5.5 Unintuitive

Due to requirement of supporting multiple fuzzy matching algorithms, the framework has been divided up into multiple components. Though, this may make the framework more flexible, it may be unintuitive for the third party developer to break

up his / her fuzzy matching algorithm to fit into the constraints of the framework's components.

5.5.6 Inflexible Interfaces

The intention of the various interface designs is that they provide the external code sufficient information to perform the required task but also ensure that unnecessary information is abstracted away from the external code. The problem with defined interfaces is that when one is developing the external code, it may become apparent that the information being supplied to the interface is inadequate. In this situation, it would be highly difficult (if not impossible) for the third party developer to acquire the desired information. The only way that the framework would be able to cater for such requirements is through the redesign of the interfaces.

5.5.7 Latency due to the Loading of DLL's at Runtime

Though the loading of the third party developer code at Runtime may add more flexibility, it also adds an additional delay to the Fuzzy Matching search, as each of the specified DLL's have to be loaded into memory when one of their contained methods are required to be run. In addition since the assembly was not known at compile time, the framework can not explicitly call the required method but rather the method is required to be invoked, which in itself can cause additional overhead. Furthermore, a DLL may contain multiple methods that are required within a particular fuzzy matching algorithm configuration. In this situation, the same DLL is loaded and unloaded each time a method within it is required. Though, the time required to load to DLL's into memory at runtime may be negligible, when performing a batch search (of thousands of names) this delay could cause a significant increase in the overall time.

5.6 Improvements to the Original Design (Version 2)

Having completed a thorough analysis of the framework design, a second version of the framework was designed and implemented in order to overcome some of the initial version's shortcomings.

5.6.1 Pre-Search Filter

Due to the shortcoming expressed in §5.5.2, it was necessary to build pre-filter functionality into the framework to enable it to pre-filter the search data through the use of fields in the external application's database (non-framework related search criteria). Thereafter, the framework makes use of the filtered search data subset (contained within the framework's database) to perform the fuzzy search.

The pre-search filter functionality is achieved through the inclusion of an additional interface (`ISearchFilterInterface`) to the existing list of framework defined interfaces and also through the addition of an element into the search input XML request packet (please see Appendix F for the changes). Since both the criteria and filter methodology (for non-Framework related filtering) can differ between different applications, it would be inadequate for the framework itself to provide the pre-search filter logic and therefore, an interface is defined to enable custom pre-search filter implementations.

Implementation of Pre-Search Filter

As with all the other of the framework's interfaces, the custom third party classes are defined in the `PreSearchFilterConfig` section of the framework's configuration file. Prior to the fuzzy search, the framework calls each of the `PreFilterSearch` methods that are contained within each of the defined pre-search filter classes. It is intended that the `PreFilterSearch` method be used to supply the framework with the filter query and not to perform the query itself.

The `PreFilterSearch` method is input a set of arguments that are later used as search parameters; these arguments are input into the framework through the `PreSearchFilterArguments` input element (within the `InputSetDef` structure). The method returns a `SearchFilter` object, which contains the following fields:

- `CoreSearchTableName` – the name of the table (in the external application's database) which the results of the Pre-Search Filter will filter. Through the specifying of this table, the framework is able to verify that the user specified pre-search filter corresponds to the search dataset as both should reference the same external table.
- Search Query – the query that will be queried against the external application's database. The results of this query form the sub-set against which the fuzzy search is performed. This query must return a list of primary keys (row numbers) in the core search table, which fulfil the filter's criteria.
- Argument List – Though the search arguments are provided to the `PreFilterSearch` method, the method itself is also required to return a list of search parameters. The reasons for this are twofold:
    a. To enable the method to perform any pre-processing on the search arguments.
    b. To enable the framework to pass parameters itself when the actual query is performed. This is intended to prevent a SQL injection attack.

If the method returns a null object, the framework assumes that no pre-filtering is to be performed.

Once all the defined `PreFilterSearch` methods have been run, the framework continues with fuzzy search through the construction of the search query. While constructing each of `WordSetComponent` sub-queries (refer to §5.3.8), the

framework iterates through each of the `SearchFilter` objects (returned from the various user defined `PreFilterSearch` methods), appending the object's Search Query to the end of the sub-query's `WHERE` clause. The `SearchFilter` object's Search Query is appended to the `WordSetComponent's` sub-query, through the addition of the following statement: `and b.coretablerownumber in (SearchFilter defined Search Query)`. At this point the framework adds the Search Filter object's argument list to the query's list of parameters.

The following example demonstrates how the pre-search filter is incorporated into the main fuzzy search query. The example extends from the example in §5.3.8, by filtering the search on employees who are over 45 years of age.

```
<SearchSets>
        <WordSetComponents>
                <Word>John</Word>
                <DataSetID IdString = "1"/>
                <WordWeighting>100</WordWeighting>
        </WordSetComponents>
        <WordSetComponents>
                <Word>Mark</Word>
                <DataSetID IdString = "2"/>
                <WordWeighting>100</WordWeighting>
        </WordSetComponents>
        <WordSetComponents>
                <Word>Smith</Word>
                <DataSetID IdString = "3"/>
                <WordWeighting>100</WordWeighting>
        </WordSetComponents>
        <PreSearchFilterArguments>45</PreSearchFilterArguments>
</SearchSets>
```

```
select q1.word as Word1, q1.datasetid as DataSetID1,
       q2.word as Word2, q2.datasetid as DataSetID2,
       q3.word as Word3, q3.datasetid as DataSetID3,
       q1.coresearchtablename, q1.coretablerownumber
from
(
       select b.word, b.datasetid, b.coretablerownumber, d.tablegroupid,
       g.coresearchtablename
       from fmf_editedword e
       join fmf_baseword b on e.basewordid = b.basewordid
       join fmf_dataset d on d.datasetid = b.datasetid
       join fmf_tablegroup g on g.tablegroupid = d.tablegroupid
       where 1 = 1
       and b.DataSetID = 1
       and b.coretablerownumber in
              (select EmployeeID
               from Employee
               where age > 45)
       and (e.text = 'JOHN' or e.text = 'J500')
) q1,
(
       select b.word, b.datasetid, b.coretablerownumber, d.tablegroupid,
       g.coresearchtablename
       from fmf_editedword e
       join fmf_baseword b on e.basewordid = b.basewordid
       join fmf_dataset d on d.datasetid = b.datasetid
       join fmf_tablegroup g on g.tablegroupid = d.tablegroupid
       where 1 = 1
       and b.DataSetID = 2
       and b.coretablerownumber in
              (select EmployeeID
               from Employee
               where age > 45)
       and (e.text = 'MARK' or e.text = 'M620')
) q2,
(
       select b.word, b.datasetid, b.coretablerownumber, d.tablegroupid,
       g.coresearchtablename
       from fmf_editedword e
       join fmf_baseword b on e.basewordid = b.basewordid
       join fmf_dataset d on d.datasetid = b.datasetid
       join fmf_tablegroup g on g.tablegroupid = d.tablegroupid
       where 1 = 1
       and b.DataSetID = 3
```

```
        and b.coretablerownumber in
                (select EmployeeID
                 from Employee
                 where age > 45)
        and (e.text = 'SMITH' or e.text = 'S530')
) q3
where 1=1
and q1.tablegroupid = q2.tablegroupid and q1.coretablerownumber =
q2.coretablerownumber
and q2.tablegroupid = q3.tablegroupid and q3.coretablerownumber =
q3.coretablerownumber
```

Weaknesses in Pre-Search Filter Design

There are two inherent weaknesses in the Pre-Search Filter design.

The main weakness is the assumption that both the framework and the external application run on the same database. This assumption is required in order to perform a single query that queries both framework and non-framework tables and fields. Furthermore, this design requires that the database user, under which the framework runs, is allowed to access non-framework tables. This design flaw inhibits the framework's independence from external applications and also limits the number of applications that the framework can serve as it may not be possible for all the applications to run on a single database.

The second weakness is that the pre-search filter query is repeated for every `WordSetComponent`. This is highly inefficient as the same results are returned multiple times, which in turn adds additional load to an already expensive query.

5.6.2 Separation of the single Pre-Processing Component into Storage Pre-Processing and Search Pre-Processing Sub-Components

It was found that in certain situations, the processing required to prepare a name for insertion into the database is different to that required to prepare a name for a search. Since the framework is designed to be as flexible as possible, in that the search algorithm can be quickly and easily changed (which includes the search pre-processing), it is preferable that the database be able to cater for these changes. This is achieved through the insertion of various permutations of an input name[14] into the database, whereas not all of the permutations are necessarily used in the processing of a search name. Through the insertion of the different permutations, one need not re-process all the names in the database, when the search algorithm changes.

The separating of the PreProcessing component into a Storage PreProcessing Component and a Search PreProcessing Component resulted in very few changes to the framework's logic. The following changes were made:

- The `DataTransformConfig` section in the framework's configuration files was replaced with two new sections, namely the `PreStorageDataTransformConfig` section and the `PreSearchDataTransformConfig` section.
- Both the two new pre-processing components utilise the `IDataTransformationInterface` in the same manner as was done previously, however, the Storage PreProcessing Component is only used prior to the insertion of names into the database and the Search PreProcessing Component is only used prior to the searching of a name against the database. The two pre-processing components cannot be used interchangeably (as was done previously with the single pre-processing component)

[14] One subjects the name to multiple processing techniques.

5.6.3 In Memory Database Caching

When the framework was deployed to a real-world business environment, the shortcoming expressed in §5.5.1 became obvious. The framework was incorporated within a nightly batch process that performs in excess of 300 000 searches per night and due to the inefficient (fuzzy) database search the process exceeded the allocated processing time and ran into business hours. This was obviously not acceptable. The devised solution was to move the fuzzy search from a database search to an in-memory search. This was achieved through the use of hash tables.

A hash table stores a list of key–value pairs, where the hash value of the key (the result of placing the key into a hashing function) denotes the address in memory where the associated record (value) is stored (Maurer & Lewis, 1975). There are several reasons why a hash table was chosen:

- A record can be found almost immediately, "without any repeated comparison with other items (Maurer & Lewis, 1975)"
- "With a hash table, we can search a file of n records in a time which is, for all practical purposes, independent of n. Thus, when n is large, a hash table method is faster than a linear search method, for which the search time is proportional to n, or a binary search method, whose timing is proportional to $\log_2 n$. (Maurer & Lewis, 1975)"
- Since a hash table uses the hash of the key to find the address of the associated data, the hash table is easily able to determine if a potential key belongs to the hash table – a key that does not exist in the hash table, would hash to an empty address.

Building Up of the In-Memory Database Structure

At start up the framework loads the entire contents of the `EditedWord` table (including the associated `BaseWord` and `DataSet`) from the database into memory. The structure, in which the names are stored in memory, has been designed

to aid in the easy retrieval of the base word (and therefore the core search row number) from a search name.

The structure of the database entries in memory is structured as:

```
Dictionary<int, Dictionary<string, Dictionary<string, List<BaseWordObject>>>>
```

The remainder of this section is devoted to the explanation of this structure:

At the highest level the data is organised as dictionary with the various dataset ID's forming the keys and the fuzzy search data (associated with each of the datasets) forming the dictionary values. A dictionary is a .Net implementation of hash table that has strongly typed keys and values (MSDN Library, 2009a).

The fuzzy search data is organised as another dictionary, where the various unique edited names (ie the contents of the `EditedWord` table) form the keys. The values associated with the various edited names are the sets of Base Word data[15]. Through this structure, the system is quickly able to determine whether the processed search name matches any of the edited names in the database. If it does, the system is quickly able to access the associated base word data.

The Base Word data is stored by means of a further dictionary, where the unique base words form the dictionary keys and a list of Base Word objects form the associated dictionary values. A Base Word Object consists of the following fields:

- `BaseWordID`
- `TableRowNumber`
- `CoreTableRowNumber`

[15] A dictionary does not store duplicate keys, therefore, if there are multiple entries in the edited word table that have the same text but reference different Base Words, they are all stored (in memory) through the same key. The unique Base Words that the different edited words reference are stored within the Base Word data structure that is associated with the edited word (key).

Entries in the Base Word table that share the same word but different `BaseWordID's`, `TableRowNumbers` and/or `CoreTableRowNumber` are all referenced by the same key in the above mentioned dictionary but each have a unique entry in the associated `BaseWordObject` list.

Searching of the In-Memory Database Structure

The framework "searches" for fuzzy matches to a search name through the following steps:

1. Using the search name's dataset as the key, the framework accesses the appropriate edited word dictionary (associated with the search dataset).
2. Having retrieved the appropriate list of edited name entries, the framework is able to determine:
    a. Whether the edited search name[16] exists in the database (i.e. does the edited search name exist as a key in the edited word dictionary)
    b. If the name does exist, the framework is then able to retrieve the list of Base Words with which the edited search name is associated. This is achieved through the lookup of the list of Base Words that are associated with the edited search name key.

In-Memory Database Maintenance

By moving the database search out of the database and into memory, the framework is required to keep both the database and in-memory versions of the data in sync at all times. This is achieved through the modification of the database maintenance functionality (see §5.3.3). An additional step has been added to the database maintenance process, in that once the necessary additions/modifications/deletions (of both the `BaseWord` and `EditedWord` tables) have been performed on the database, these changes are applied to the in-memory structure.

[16] One of the entries in the list of edited names returned by Search Pre-Processing component

Multiple edited words can be associated with a single base, therefore by taking advantage of C#'s referencing of objects through pointers, all edited words that reference the same `BaseWord` (in the database) point to the same list of Base Word Objects. In addition, all the entries in the base word table that have the same base word, all point to the same in-memory list of Base Word Objects. This ability to have several different objects referencing the same single object aids in the maintenance of the in-memory database structure as one is quickly able to locate and update the in-memory structure.

Ternary Search Tree

As discussed previously in this chapter, the use of dictionaries provide considerable improvements in the framework's search time, however, the limitation of using dictionaries is their reliance on keys and more importantly strict requirement of "exact" matching of keys. Through the implementation of dictionaries as search structures, the framework's designed search flexibility is undermined as dictionaries are incapable of performing a wildcard search that is possible through the use of SQL's "`LIKE`" clause. It was found that through the modification of the Ternary Search Tree (TST) algorithm (Bentley & Sedgewick, 1997), a "dictionary" with wildcard search functionality could be achieved. The implementation of the modified TST algorithm provided the framework with the above-mention functionality. An example of this functionality is that the framework is now capable of searching for all `BaseWords` that begin with the letter "B".

It was decided to make use of TST structure for the following reasons:

- TST's provide an implementation of symbol tables[17] with Partial Matching functionality (Bentley & Sedgewick, 1997; Siegel, 1999). As discussed

[17] A symbol table is another name for a dictionary (Siegel, 1999).

previously this was the primary reason for the change from a dictionary to TST.

- TST's are a derivative of Radix Search Trees. Unlike traditional trees, where each node represent the entire key, each node in a Radix Search Tree corresponds to a part of the whole key, where the complete key is specified by a path through the tree (Siegel, 1999). This sub-key structure lends itself particularly well to the use of strings as keys, which enable TST's to provide an efficient implementation of string symbol tables (Bentley & Sedgewick, 1997).

5.6.4 Delegate to Access Contents of the Rules Table

In order to address the problem discussed in §5.5.4, a new mechanism to supply the third party application code with the contents of the `Rules` table was developed. It was identified that rather than caching the contents of the `Rules` Table, it would be best to supply the user with a mechanism for direct access to the database (i.e. the Rules Table) while still providing a level of abstraction away from the database itself. A delegate provides the ideal mechanism to achieve the above mentioned functionality as a delegate (in C#) "encapsulates both an object instance and a method (Jagger et al., 2007)".

Through the use of a delegate one is able to pass an internal Framework method to the external code (i.e. the Third Party developer's code) as if it were a parameter. Furthermore, since the delegate points to an instance of a Framework object (within which the method belongs), the delegate is able to use the object's private data members to connect to the database and retrieve the necessary information.

This enables the framework to internally instantiate an object with the database connection. The class, from which the above mentioned object is instantiated, contains a method that retrieves the contents of the `Rules` Table and returns the data

structured into a dictionary. This method has the same method signature as that defined by a delegate. By defining the delegate in the same library that defines the framework's interfaces, the external third party code is also aware of the delegates. When the framework calls a particular implementation of an interface, it also passes through the name of the method (that implements the defined delegate) and through the use of this passed method, the external code is able to access items from the Rules Table, without any knowledge of neither the database nor its connection strings.

Implementation of Rules Table Access

Though the previous section discussed how a delegate could be used within the framework's interfaces, the manner in which it was actually implemented is slightly different as it is intended to make the access to the `Rules` Table more intuitive.

Since delegates are not well known, it would be more intuitive to wrap the delegate within a defined class (called the `FMFRules` class) and then pass the instantiated object to the third party code. The `FMFRules` class is defined as follows (Refer to Appendix G for the implementation of the `FMFRules` class):

- The class defines a delegate called `LoadRuleTableDelegate`. The `LoadRuleTableDelegate` has no input parameters but returns a dictionary.
    - The returned structure is composed of two dictionaries; one within the other.
        - Each of the Action Types (e.g. exclude, substitute, etc), defined within the `FMF_ActionType` table, forms a unique key within the outer dictionary. The associated values (with each of the keys) correspond to the entries in the Rules Table that implement the various action types. Each of these values themselves is stored as a dictionary structure.

        - The Inner Dictionary's keys consist of the unique Search Phrases in the `FMF_Rules` table that correspond to the action type defined by the outer dictionary key. The values corresponding to each of these keys consist of the list of entries in the Rules Table that reference both the Search Phrase, defined by the inner dictionary key, and the action type defined by the outer dictionary key.
        - The reason each key (in the inner dictionary) maps to a list of entries, rather than mapping to a single entry, is to cater for the possibility that there might be more than one entry in the Rules Table that has both the same search phrase and action type.
        - Each entry in the above mentioned list is composed of a `RulesTable` object. The `RulesTable` class contains the properties stored in each entry in the `Rules` Table, i.e. `ActionTypeID`, `ActionType`, `SearchPhrase`, `ReplacePhrase` and `Penalty`
    - A private method called `LoadRulesTable` calls the `LoadRuleTableDelegate`, thereby populating the internal `_RulesTable` dictionary.
    - The constructor is input the `LoadRuleTableDelegate` and then calls the `LoadRulesTable` method.
    - A public method called `getRulesList` which is passed an `ActionType` (as an input parameter) and returns the entries in the `Rules` Table that implement the specified action type.

This design appears to be counter-intuitive as due to the way in which the `FMFRules` class is implemented, the same functionality can be achieved more simply without the use of a delegate. This design, however, must be viewed rather as a proof of concept.

Currently, the contents of the `Rules` Table are loaded into an internal dictionary, when the object is instantiated. Since the object is instantiated within the framework's code, the use of a delegate is actually redundant as the third party code receives the `FMFRules` object with an already populated dictionary and has no ability to call the delegate (that is wrapped within the class).

However, through the implementation of this class, it has been demonstrated that delegates can achieve the specified objectives. In the future, the `FMFRules` class `LoadRuleTable` method can be changed from a private to a public method. This will enable the third party code to refresh the `FMFRules` dictionary, enabling the latest contents of the `FMFRules` table to be retrieved from the database.

5.6.5 Pre-Search Filter In-Memory Database Caching

When the database search was moved from the database to memory, it was initially assumed that there was no need to re-implement the pre-search filter functionality as due to the speed of the memory search, the larger search set would make a negligible difference. This assumption was proved to be wrong due to the following two factors:

- The need to be able to filter the search set with non-framework related information still remained.
- Though there was a negligible difference in the time required to perform the in-memory search (on the non-pre-filtered search set), it was found that the time required to process the validity of the returned entries was excessive. The underlying reason that caused the bottleneck is the search process requirement that the viability of the returned search results must be computed one entry at a time and therefore, the more entries returned by the "database", the more post-processing time is required.

Due to the above mentioned reasons, the pre-search filter was re-implemented in order that it could be utilised with the in-memory database search.

Implementation of Revised Search Pre-Filter

Prior to the database search, the framework pre-filters the Base Word ID's, against which the database search will later be performed. This is achieved by the framework sequentially calling each of the user defined `PreFilterSearch` methods (please see 0, regarding the `ISearchFilterInterface` and the implementation thereof). After running each `PreFilterSearch` method, the framework queries the database to determine which `BaseWordID's` conform to the query described in the method. The results of the query are then added to the list[18] of pre-filtered `BaseWordID's`.

Having completed the processing of the defined `PreFilterSearch` methods, the framework continues with the existing search process; the only difference being the inclusion of an additional step to the database search. Once the framework has determined that a particular edited search name exists in the database, the framework then checks whether the `BaseWordID`, corresponding to the database match, is contained within the list of pre-filtered `BaseWordID's`. Only if the `BaseWordID` is contained within the filtered list, will the matched database entry be added to the search result set. If no pre-filter arguments were supplied to the search, the framework will omit this step and will immediately add the matching entry to the result list.

[18] This list contains only unique Base Word ID's.

Caching of Previous Pre-Search Filter Results In-Memory

Through the implementing of the pre-search filter, the framework is again required to query the database when performing a search. The moving of the database search into memory, however, was intended to do away with the requirement of performing actual database queries. In addition, the queries that run as part of the pre-search filter may be unoptimised, potentially causing searches to be slower.

When performing single (non-batch) searches, the user will generally tolerate a slightly longer search time. Though the duration of individual fuzzy matching searches may be acceptable, when performed as part of a batch process, the cumulative effect of the multiple database queries (for each of the pre-search filter queries) results in the batch process taking a prohibitively long amount of time. Often batch processes involve the lumping together of similar entries. For example, one would search for people whose entries have been updated after a certain date. Another example is the searching for all people born in the same month. This results in the same pre-search filter argument often being passed to the framework across many different searches.

One is able to monopolise on this characteristic to speed up batch searches, through the caching of the pre-search filter results after each search. In caching the pre-search filter results, the framework also stores the corresponding pre-search filter arguments. The pre-search filter process is modified in that prior to the database search being performed, the framework first checks whether the same pre-filter arguments, as those that were passed for the previous search, have again been specified. If so, the framework will not run the database query but rather use the previous pre-search filter results.

Weakness in Pre-Search Filter Caching Design

The biggest weakness in this design is that the cached results remain stagnant and can become stale over time as the results of the original pre-search filter query may have changed. One way to overcome this is to attach a timestamp to the cached results and after a configurable time period has elapsed the query is rerun and the cached results are refreshed. Another option is for the framework to provide a mechanism by which the user can force the refreshing of the pre-search filter results; effectively clearing out the cached results and rerunning the query.

The design, implementation, review and revision process of a generic fuzzy name matching framework has been documented through the course of this section. As has been discussed, the framework has been designed to address the requirements specified in §4. The subsequent section (§6) defines and describes the testing process used to verify whether the designed framework conforms to outlined requirements.

6 Solution Testing

Having completed several iterations of the framework, it was necessary to evaluate its design and implementation through a series of test cases. This section details the testing process and outcomes through the defining of the test objectives and corresponding test cases and the analysis of the results thereof.

6.1 Test Objectives

Prior to the outlying of the test objectives, it is necessary to review the dissertation's research questions to ensure that the various test objectives and therefore test cases are compliant with research objectives of the dissertation.

In reviewing the first research question[19], it is apparent that by virtue of the framework's design, this question has been answered. If no common processes could be found between the various fuzzy matching algorithms, no framework could be designed and therefore, it would be impossible to continue this dissertation beyond the literature review.

The subsequent two research questions[20] have been refined in the form of the following two high level test objectives:

1. To verify whether the framework is a viable alternative to algorithm specific implementations of various fuzzy matching algorithms within a name matching context. This high level objective relates to research question 2.
2. To verify whether the framework can be utilised as a black-box search tool, in which an external application supplies search parameters to the

[19] Do the majority of Fuzzy Matching algorithms contain one or more common high level processes that can be integrated into a single generic framework? (found in §3.1)

[20] Is it possible for this generic framework to cater for the requirements of the matching of various combinations of names? And can this framework provide a mechanism for the implementation of custom logic for the common processes? (both found in §3.1)

framework and the framework then returns a result. This high level objective relates to research question 3.

The following is a detailed list of the test case objectives:

1. To verify that the framework is capable of loading third party code and integrating into its own search, in order to implement a particular fuzzy matching algorithm
2. To verify that the framework provides a high level abstracted search process
    a. In particular, the aim of this objective is to determine whether the framework is capable of managing both the internal search flow and the database maintenance with limited input from the third party developer. The only third party developer input, should be the actual implementation of the fuzzy matching algorithm logic.
    b. This objective also aims to verify that the framework presents a single start and end point to an external application
3. To verify whether the framework is able to search on two or more related names (e.g. first name and surname) and ensure that the individual search results are all resolved to single entities whose names each match each of the input search names
4. To verify whether the framework is generic and does not subscribe to a single fuzzy matching algorithm
    a. Furthermore, to verify whether the framework can implement different types of algorithms
5. To verify whether the framework's search times are comparable to algorithm specific implementations of the various fuzzy matching algorithms
6. To verify that the framework is capable of serving batch processes
7. To verify that the framework is capable of simultaneously serving multiple applications

Table 6.1 illustrates under which high level objectives, the sub-objectives fall.

Table 6.1: Sub-Objective to high level objective mapping

Objective	High Level Objective
1	1,2
2	2
3	1
4	1
5	2
6	1,2
7	2

Using the various test objectives (defined above), a series of test cases was developed.

6.2 Test Data

When performing the literature review, it was found that there is no standard database of names, against which all fuzzy matching algorithms are tested and hence there is no standard metric. Each paper used a different a source for its test data; the following is a list of several of these sources:

- A U.S. District Court database containing records for cases assigned to various judges (Branting, 2003)
- Customer and employee lists of travel agencies accessed through Amadeus' systems (Du, 2005)
- Online onomastikon (dictionary of names) (Du, 2005)
- A dictionary distributed with the *ispell* interactive spelling checker (Zobel & Dart, 1995)
- A set of distinct personal names compiled from student records and a bibliographic database (Zobel & Dart, 1995)
- Confidential datasets through commercial work (Zobel & Dart, 1996)

- A Dictionary of English Surnames (Snae, 2007)
- Given- and surnames extracted from a midwives database (Christen, 2006)

It must be noted that the lack of standard fuzzy matching dataset will not hinder the above mentioned test cases. A standard database of test data is important when one has developed a new fuzzy matching algorithm and is required to compare its performance to existing algorithms. The testing required for this dissertation, however, aims to compare the performance of different implementations of a particular fuzzy matching algorithm (i.e. comparing the performance of an algorithm specific implementation to the performance of the same algorithm having been implemented within the framework) and therefore a standard dataset is less important.

The test data set was constructed from three sources:

1. A list of high profile Southern African individuals (5847 entries) (Media24, 2009).
2. A generated[21] list of surnames and first names using the most common surnames and first names found in the 1990 US Census (20 163 entries) (U.S. Census Bureau, 2009)
3. A scrambled list of employee first names and surnames (491 entries) from a company. This company did not wish to be named.

[21] The source provided lists of the frequency of occurrence of the most common surnames, male first names and female first names. Each list of names was downloaded individually. Since the framework testing requires first name – surname pairs, first name - surname pairs were generated through the combining of the separate lists. This was achieved by combining the most common surnames with the most common male and female first names.

6.2.1 Test Data Set Properties.

The main properties of the test data set are described in Table 6.2.

Table 6.2: Test Data Set Properties

Property	Metric
Total Number of Entries	26481
Total Number of Unique Entries	26431
Total Number of Unique Surnames	5656
Total Number of Unique First Names	3335
Average Surname Length	6.820 characters
Average First Name Length	7.194 characters

Please refer to Appendix H for a more detailed analysis of the test data set.

6.3 Test Cases

The test cases have been developed to verify whether the framework conforms to original design requirements.

Generally most tests involve the comparison of the framework's performance to an algorithm specific implementation of a particular fuzzy matching algorithm. In an algorithm specific implementation the fuzzy matching algorithm is "hard coded" within the compiled code and therefore the application is not capable of implementing a different type of algorithm without being rewritten and recompiled.

Unless otherwise specified, all tests are performed against the same database. This database is composed of the test data set that has been described above.

6.3.1 Test Case 1 - Basic Framework Search

This test demonstrates objectives 1 and 2.

The test is performed through the implementation of the same fuzzy matching algorithm in both an algorithm specific implementation and within the framework. The same search name is input into both algorithm implementations and the results are compared.

Success is defined by both implementations of the algorithm returning the same results.

6.3.2 Test Case 2 - Related Searches

This test demonstrates objective 3.

The test is performed through the implementation of a fuzzy matching algorithm within the framework (The fuzzy matching algorithm that is implemented for this test is immaterial). A person's first name and a surname are both input into the framework and the results are then evaluated.

Three criteria are required to be fulfilled for the test to be considered a success:

1. Each first name-surname pair returned by the framework corresponds to the same `TestCaseID` (the primary key of the test data set table) in the database
2. When investigating the `TestCaseID` within the external database, one must ensure that the first names and surname corresponding to the `TestCaseID`, must be an identical match to the ones returned by the framework's search
3. Both the returned first names and surname must each be a fuzzy match to the input first names and surname

6.3.3 Test Case 3 - Flexibility

This test demonstrates objective 4.

The test methodology is the same as that of Test Case 1 (see §6.3.1). This test, however, is repeated three times. The first time the algorithm used is a Phonetic Encoding algorithm, the second time an Edit Distance algorithm is used and finally the third time an N-Gram algorithm is used.

One is able to consider the test a success if the results of both the framework implementation and the specific algorithm implementation of each of the three different fuzzy matching algorithm types are the same.

6.3.4 Test Case 4 - Speed

This test demonstrates objective 5.

The test methodology is the same as that of Test Case 3 (see §6.3.3). However, instead of comparing the search results, the search times are compared.

Unlike previous tests, where success was defined on the return of matching results, this test is considered to be successful if the time required to perform a search within the framework's implementation of an algorithm is within 5% of the time required to perform the same search within the algorithm specific implementation. This applies to the implementations of all three algorithm types.

6.3.5 Test Case 5 - High-Load Testing

Batch Processes

This test demonstrates objectives 6.

The test is performed by making a large number of search requests to the framework over an extended amount of time. The overall duration and the number of searches performed is then determined. One must note the metrics used within this test:

- Number of Searches ≈ 120 000
- Search Database Size ≈ 200 000 entries
- Process Duration ≈ 12 hours

Success is defined as whether the framework is capable of being responsive throughout the entire time period and thus servicing all search requests.

Simultaneous Search Requests

This test demonstrates objective 7.

A small test application, which is capable of spinning multiple threads, is to be developed. Each of the application's individual threads is to request multiple searches from the framework. The intention of this test is to simulate the load of multiple applications simultaneously accessing the framework.

This test is considered to be successful if the framework is capable of successfully responding to each of the search requests.

6.4 Test Results and Discussion

The subsequent sections describe and discuss the results of the various test cases. In situations where test case failed an explanation is provided.

6.4.1 Test Case 1 - Basic Framework Search

The Soundex algorithm was implemented as both a Soundex specific implementation (Birkby, 2002) and within the framework. Thereafter, the search name "Frank" was input into both the Soundex implementations and the outputted results were recorded. The results of the two searches are displayed in Table 6.3. The results were identical between the two implementations.

Table 6.3: Test Case 1 Results

Search Name	Soundex Specific Implementation	Framework Implementation
Frank	FARANAAZ (1 Entries)	FARANAAZ (1 Entries)
	FRANCES (24 Entries)	FRANCES (24 Entries)
	FRANCESCO (1 Entries)	FRANCESCO (1 Entries)
	FRANCINE-ANNE (1 Entries)	FRANCINE-ANNE (1 Entries)
	FRANCIS (45 Entries)	FRANCIS (45 Entries)
	FRANCIS KWAME (1 Entries)	FRANCIS KWAME (1 Entries)
	FRANCISCO (20 Entries)	FRANCISCO (20 Entries)
	FRANCISCUS (1 Entries)	FRANCISCUS (1 Entries)
	FRANCKI (1 Entries)	FRANCKI (1 Entries)
	FRANCO (2 Entries)	FRANCO (2 Entries)
	FRANCOIS (14 Entries)	FRANCOIS (14 Entries)
	FRANK (35 Entries)	FRANK (35 Entries)
	FRANK REGINALD (1 Entries)	FRANK REGINALD (1 Entries)
	FRANKIE (20 Entries)	FRANKIE (20 Entries)
	FRANKLIN (22 Entries)	FRANKLIN (22 Entries)
	FRANKLYN (1 Entries)	FRANKLYN (1 Entries)
	FRANS (4 Entries)	FRANS (4 Entries)
	FRANZ (3 Entries)	FRANZ (3 Entries)

As can be seen in the above table, both implementations returned exactly the same results and therefore test case 1 is considered to be successful.

6.4.2 Test Case 2 - Related Searches

In order to perform this test case the framework was configured to implement the Soundex algorithm. The first name "Lora" and the surname "Schroeder" were then input as the search criteria. Though the input first name was configured to search against a dataset of first names and the input surname was configured to search against a dataset of surnames, both datasets corresponded to the same table group. This informed the framework that the results of the first name search and the surname search must resolve to the same row in the search table (i.e. the table containing the test case names). Table 6.4 below displays the results of the search, while Table 6.5 and Table 6.6 verify the validity of each of the individual searches.

Table 6.4: Test Case 2 Results

Search First Name	Search Surname	Match First Name	Match Surname	Match Row Number	Does Correspond to DB	Are Both First Name and Surname viable fuzzy matches
Lora	Schroeder	LORA	SCHROEDER	458	Yes	Yes
		LEROY	SCHNEIDER	17756	Yes	Yes
		LORI	STARR	23817	Yes	Yes

Table 6.5: First Name Search Results

Search First Name	Unique First Name Matches
Lora	LARRY
	LARRY
	LAURA
	LAURIE
	LEROY
	LERUO
	LORA
	LORI

Table 6.6: Surname Search Results

Search Surname	Unique Surname Matches
Schroeder	SCHNEIDER
	SCHNEIDER
	SCHREUDER
	SCHROEDER
	STARR
	STEAR
	STORE
	STOREY
	STORY
	STUHLER

Table 6.4 displays that the search names "Lora" and "Schroeder" returned three matches, however, it was required that these results be verified before the test case could be considered successful. The following was performed to verify the validity of the search results:

1. The original test case name table was queried to verify that the Match Row Number that was returned by the framework does actually reference both the first name and surname that were returned by the framework. The results of the query are displayed in the "Does Correspond to DB" column in Table 6.4.

2. Both the first name and surname were independently searched to ensure that results returned by the test case search are actually fuzzy matches to both the input first name and surname. The results of the individual searches are displayed in Table 6.5 and Table 6.6 respectively.

Through the verification of the search results, it has been shown that the framework is capable of performing searches on related names and therefore Test Case 2 can be considered as a success.

6.4.3 Test Case 3 – Flexibility

In order to test the framework's flexibility, three different fuzzy matching algorithms were implemented as algorithm specific implementations and as framework implementations. These three implemented algorithms were the Soundex, Levenshtein Distance (Foidl, 2009) and N-Gram Similarity (Dao, 2005) algorithms. It was necessary that the same search name, "Carl", be used across all the searches to ensure standard test conditions; these searches included the three algorithm specific implementations and the three framework implementations.

The three above mentioned algorithms were chosen as it has been identified that the most commonly used algorithms can be divided into three main groups, namely Phonetic, Edit Distance and N-Gram's. Each of the chosen algorithms is representative of a particular fuzzy matching algorithm group, which enables the test case to provide coverage over a large array of algorithms. Furthermore, the chosen algorithms demonstrate two considerably different search approaches:

- The Soundex algorithm pre-processes the name prior to the search, converting it into an appropriate code. Thereafter, the algorithm searches for all other names that have the same Soundex code, i.e. since all the entries in the database have already been processed and the corresponding Soundex codes stored, the algorithm is required to find all Soundex entries in the database

that have an exact match to the search Soundex code. As a result, the algorithm only retrieves entries from the database that are matches.

- Both the Levenshtein Distance and N-Gram Similarity algorithms require a direct comparison between the search name and the database entry to compute a score. Hence, the algorithm blindly pulls out entries from the database, unaware whether the entry will or will not be a match (the degree of match is only determined after a score has been computed) and therefore potentially very few of the database returned entries will be a match.

The results of the various searches are displayed in Table 6.7, Table 6.8 and Table 6.9 below. The results of all three of the algorithm specific implementations and their framework equivalents were identical.

Table 6.7: Soundex Search Results

Soundex	
Algorithm Specific Implementation	**Framework Implementation**
CAREL (7 Entries)	CAREL (7 Entries)
CARL (29 Entries)	CARL (29 Entries)
CARLA (21 Entries)	CARLA (21 Entries)
CARLO (2 Entries)	CARLO (2 Entries)
CAROL (28 Entries)	CAROL (28 Entries)
CAROLE (23 Entries)	CAROLE (23 Entries)
CARROL (1 Entries)	CARROL (1 Entries)
CARROLL (20 Entries)	CARROLL (20 Entries)
CHARL (6 Entries)	CHARL (6 Entries)
CHARLIE (20 Entries)	CHARLIE (20 Entries)
CHERYL (26 Entries)	CHERYL (26 Entries)
CYRIL (4 Entries)	CYRIL (4 Entries)
CYRILLE (1 Entries)	CYRILLE (1 Entries)

Table 6.8: Levenshtein Search Results

Levenshtein	
Algorithm Specific Implementation	**Framework Implementation**
CAREL (7 Entries)	CAREL (7 Entries)
CARL (29 Entries)	CARL (29 Entries)
CARLA (21 Entries)	CARLA (21 Entries)
CARLO (2 Entries)	CARLO (2 Entries)
CAROL (28 Entries)	CAROL (28 Entries)
CARY (20 Entries)	CARY (20 Entries)
CHARL (6 Entries)	CHARL (6 Entries)
EARL (22 Entries)	EARL (22 Entries)
KARL (25 Entries)	KARL (25 Entries)

Table 6.9: N-Gram Similarity Search Results

N-Gram Similarity	
Algorithm Specific Implementation	**Framework Implmementation**
CAREL (7 Entries)	CAREL (7 Entries)
CARL (29 Entries)	CARL (29 Entries)
CARLA (21 Entries)	CARLA (21 Entries)
CARLO (2 Entries)	CARLO (2 Entries)
CARLOS (22 Entries)	CARLOS (22 Entries)
CARLTON (20 Entries)	CARLTON (20 Entries)
CAROL (28 Entries)	CAROL (28 Entries)
CARROL (1 Entries)	CARROL (1 Entries)
CARROLL (20 Entries)	CARROLL (20 Entries)
CARY (20 Entries)	CARY (20 Entries)
CHARL (6 Entries)	CHARL (6 Entries)
EARL (21 Entries)	EARL (21 Entries)

All three of the above tables verify that the framework is capable of implementing multiple different fuzzy matching algorithms as the framework returned the same results as the algorithm specific implementation for each of the algorithms. It can be concluded that Test Case 3 fulfilled its success criteria.

6.4.4 Test Case 4 – Speed

The same test conditions that were implemented for Test Case 3 were also used for this test case and the same search name, “Carl”, was also used. Since this test case involves the measurement of the search times, it was not adequate to simply perform the search once for each of the different algorithm implementations; rather the same search was repeated ten times for each of the six different algorithm implementations (i.e. three algorithm specific implementations and three framework implementations). The average search time was thereafter calculated. The results are shown in Table 6.10, Table 6.11 and Table 6.12.

Table 6.10: Soundex Search Times

	Soundex	
	Algorithm Specific Implementation	**Framework Implementation**
Individual Search Times	< 0.00001 ms	62.4984 ms
	< 0.00001 ms	46.8738 ms
	< 0.00001 ms	46.8738 ms
	< 0.00001 ms	46.8738 ms
	< 0.00001 ms	46.8738 ms
	< 0.00001 ms	46.8738 ms
	< 0.00001 ms	46.8738 ms
	< 0.00001 ms	62.4984 ms
	< 0.00001 ms	46.8738 ms
	< 0.00001 ms	46.8738 ms
Average Search Time	< 0.00001 ms	49.99872 ms

Table 6.11: Levenshtein Search Times

	Levenshtein	
	Algorithm Specific Implementation	**Framework Implementation**
Individual Search Times	171.897 ms	6859.1994 ms
	156.27 ms	6874.824 ms
	125.016 ms	6749.8272 ms
	125.016 ms	6765.4518 ms
	125.016 ms	6796.701 ms
	140.643 ms	6765.4518 ms
	125.016 ms	6765.4518 ms
	125.016 ms	6921.6978 ms
	125.016 ms	6968.5716 ms
	125.016 ms	6781.0764 ms
Average Search Time	134.3922 ms	6824.82528 ms

Table 6.12: N-Gram Similarity Search Times

	N-Gram Similarity	
	Algorithm Specific Implementation	**Framework Implmementation**
Individual Search Times	296.883 ms	6874.824 ms
	218.7598 ms	6827.9502 ms
	218.7598 ms	6827.9502 ms
	203.1341 ms	6937.3224 ms
	218.7598 ms	6890.4486 ms
	218.7598 ms	6874.824 ms
	203.1341 ms	6843.5748 ms
	218.7598 ms	6827.9502 ms
	234.3855 ms	6874.824 ms
	218.7598 ms	7062.3192 ms
Average Search Time	225.00955 ms	6884.19876 ms

Upon reviewing the above tables, it is evident that the framework's search times are not comparable to those of the algorithm specific implementations. Before an in depth explanation of the search times of each of the algorithm implementations is provided, it must be considered that although ultimately one is only concerned with the results returned from each of the algorithm implementations, the functionality provided by

the algorithm specific implementations is not equivalent to that provided by the framework and therefore a direct comparison cannot be considered.

Explanation of the Test Case Results

The algorithm specific implementations have the following high level process:

1. A text file of names is read into memory
    a. In the case of the Soundex implementation, each of the names is processed and the equivalent code is generated.
2. A search name is input into the application
3. Matches to the search name are found and added to the result set
    a. In the case of the Levenshtein and N-Gram implementations, only matches that have a score above the specified threshold are added to the result set
4. The results are displayed

The following aspects within the framework's functionality cause additional overhead:

1. The framework has been developed as a web service and therefore the time required to do the necessary routing must be considered.
2. The framework validates the search inputs; in particular the framework verifies that the specified datasets are valid and in the case of a set of related names the framework evaluates whether all the specified datasets belong to the same table group.
3. The framework is generic and measures are required to determine which logic is to be implemented to which datasets.
4. The framework manages it own database of search names against which all input names are searched.

The remainder of the section discusses the individual algorithm implementations.

Soundex

Two aspects within the framework's implementation of the Soundex algorithm caused the excessive search times, namely:

- A check within the user defined code to determine whether the Soundex processed datasets have been specified, which accounted for more than 96% of the search time. This check was necessary as the framework had been configured that the Soundex, Levenshtein or N-Gram search could be performed depending on which dataset was input to the search and therefore some logic was required to select the correct fuzzy matching process. If only a Soundex implementation had been configured, there would be no need for this check.
- The framework's validation of the input dataset accounted for more than 3% of the search time.

Figure 6.1 below shows the results of performance profiling performed on the framework, during its Soundex search. The times displayed in the performance profile should not be considered as the profiling process retards the performance of the profiled code. Rather one should consider the percentage duration of each of the methods.

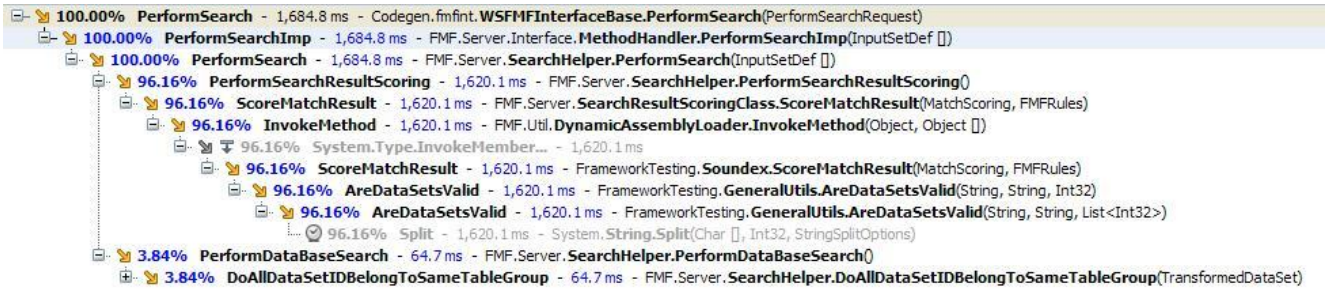


Figure 6.1: Results of the profiling of the Framework Soundex Implementation

As can be seen from the figure above, if one were not to consider the above mentioned issues which themselves are not implemented in the Soundex algorithm

specific implementation, the framework's search time would be almost identical to that of the algorithm specific implementation.

Levenshtein Distance and N-Gram Similarity

Both the Levenshtein Distance and the N-Gram Similarity algorithms are adversely affected by the same framework implementation issues. As discussed previously, both these algorithms are only able to determine the degree of the match of a name to a search name subsequent to the database search and therefore a large number (if not all) of the database entries is generally retrieved. This in turn, results in the framework performing numerous scoring operations.

The reason for the excessive search times is due to the manner in which the scoring has been implemented. Every time the framework is required to compute the score, which is done for every entry retrieved from the database, the framework invokes a user defined method[22]. After an investigation it was found that the manner in which the framework invokes this method is particularly slow. In order to simplify the accessing of the run-time loaded classes, the framework wraps each of the user defined classes within its own classes. These wrapper classes then handle the invocation of the underlying class' methods. This is achieved through the wrapping of the method invocation within a method that shares the same name as the invoked method. Prior to the invocation of the method, a stack trace is performed to determine the invoked method's name and therefore through the retrieval of the wrapping method's name, the framework is able to invoke the correct method. It was discovered that this performing of a stack trace is inefficient. In the case of the test cases, every search resulted in a stack trace being performed over 28 000 times, which accounted for between 55% and 90% of the overall search time.

[22] As discussed previously, this method is only loaded into memory at runtime.

This problem can be easily addressed by explicitly specifying the name of the method to be invoked instead of the framework attempting to determine the name from the wrapping method.

Further reasons for the slower search times between the algorithm specific implementations and the framework implementations are as follows:

- The framework processes all the entries that are retrieved from the database regardless of whether their scores are above the defined threshold, whereas in the algorithm specific implementations, the algorithm immediately assesses if the entry's score is above the defined threshold and only processes conforming entries.
- As discussed as an issue for the Soundex implementation, the framework had been configured to use the input datasets to determine which fuzzy matching algorithm should be used (the AreDataSetsValid method).
- The framework sorts all the results according to their scores.

Figure 6.2 and Figure 6.3 below show the results of performance profiling performed on the framework, during the Levenshtein Distance and N-Gram Similarity searches. Again one should only consider the percentage duration and not the times specified.

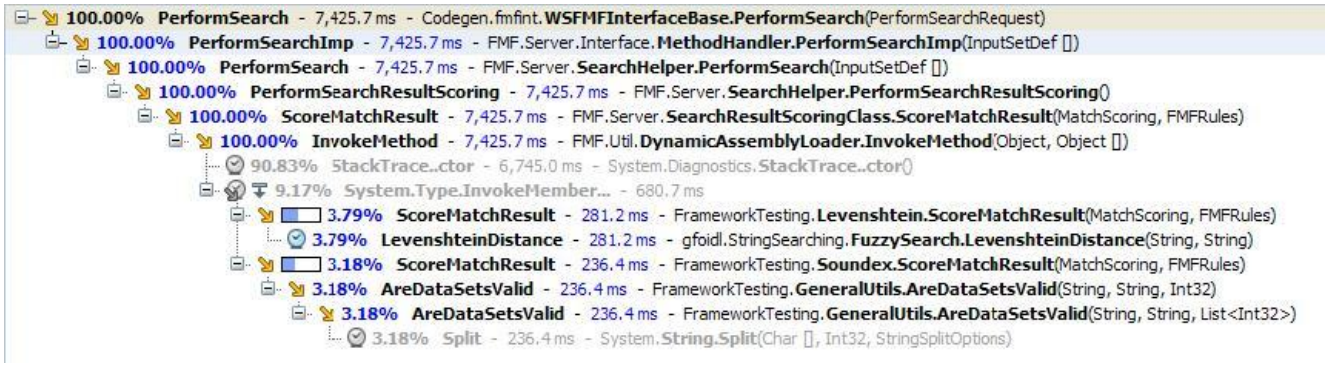


Figure 6.2: Results of the profiling of the Framework Levenshtein Distance Implementation

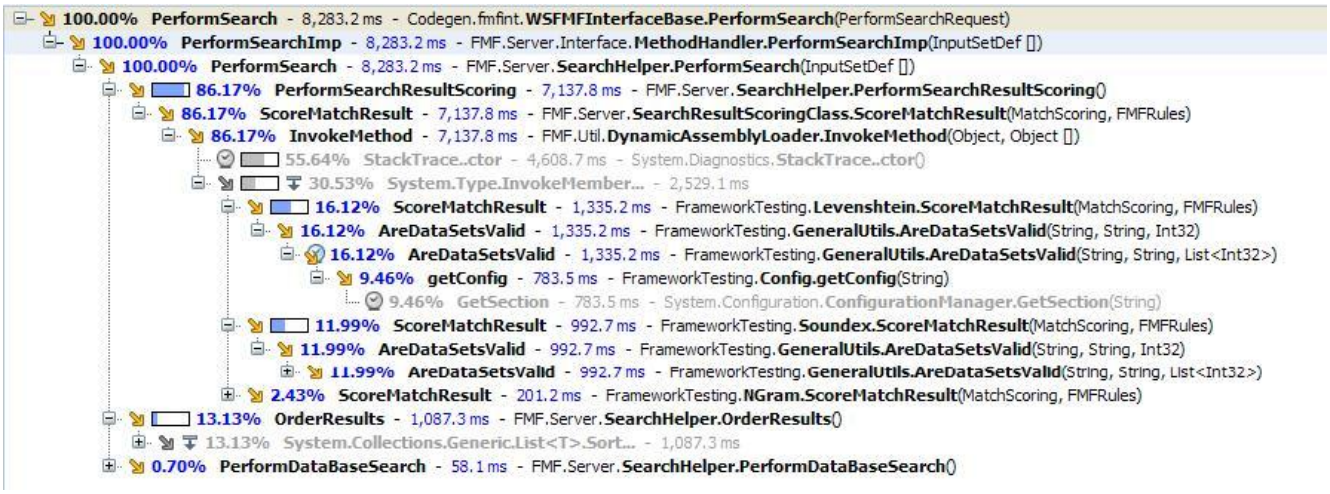


Figure 6.3: Results of the profiling of the Framework N-Gram Similarity Implementation

Through the analysis of this test case it has been identified that the framework's inferior performance is due to the following factors:

1. Some aspects of the framework implementation are inefficient and cause unnecessary delay. These are issues that can be easily addressed and resolved in future versions of the framework.
2. The generic nature of the framework requires that it makes provision for the different algorithm requirements. Generally, this will always be less efficient than the algorithm specific implementation.
3. The framework does not only provide search functionality but is rather an entire fuzzy matching solution, which has been designed to respond to XML requests, return meaningful results[23] to calling applications and maintain its own fuzzy matching database. All of these aspects provide additional overhead, which would is not found in a simple algorithm specific implementation.

Following from the above discussions it can be concluded the framework is capable of achieving equivalent search times for algorithms that are able to pin-point fuzzy

[23] These results have context within the calling application and are not just a list of matching names

matches from the database, without the need to evaluate the fuzzy matches subsequent to the database search (e.g. the Soundex algorithm). In summary the framework is comparable for algorithms that retrieve a minimum amount of entries from the database and therefore do not require large numbers of scoring operations to be performed. To conclude, it can be considered that the objective of Test Case 4 has been partially achieved.

6.4.5 Test Case 5 - High-Load Testing

Batch Processes

Unlike the previous test cases, which were performed against the Test Data set on an ad-hoc basis, this test case involved the use of an instance of the framework that has been deployed within industry. This deployed framework instance serves an external application that runs a batch search process every night. Please refer to §6.5 regarding the details of the framework deployment.

Since this is an ongoing process, no special changes were made to implement this particular test case. Rather, the existing batch process was monitored and the results of which were analysed. One of the main tools used to analyse the batch process is the framework's logs, which store the request and response of every search call made to the framework. Through the use of a Python script, the logs were parsed and the high level details of the batch process extracted. These details are shown in Table 6.13.

Table 6.13: Batch Process Details

Process Time Period	12:05:27	Hours
Total Number of Searches	118223	
Total Search Time	59.97	Minutes
Average Search Time	30.44	ms
Longest Search Time	14.063	s
Shortest Search Time	0	s

As is displayed in the above table the framework performed over a hundred thousand searches over a twelve hour period, furthermore through the analysis of the logs it was found that searches were performed at regular intervals throughout the process time. Following from the previous test case, an average search time of 30ms is considered to be acceptable. The most interesting result of the test case is that of the total duration of the process (12 hours) the framework searches only account for an hour, which corresponds to about 3% of the total time. This clearly indicates that the framework did not cause any bottleneck within the process.

It is therefore evident that this test case achieved its objectives as the framework was capable of responding to search requests throughout an extended period of high load. Regarding the high level objective of this test case, it was proved that the framework can be integrated with another system and not inhibit its operations.

Simultaneous Search Requests

This test case was performed through the implementing of the Soundex algorithm within the framework and the developing of a test application, which creates twenty threads. Each of the threads, in the application, requests the framework to search an arbitrary name at arbitrary intervals over a five minute period. The results of each the threads' searches are tabulated below.

Table 6.14: Results of Simultaneous Search Requests Test Case

Thread	Number Passes	Number Fails	Average Search Time (ms)	Longest Search Time (ms)	Shortest Search Time (ms)
1	107	0	234.375	1265.625	109.375
2	109	0	221.9036697	1250	93.75
3	113	0	238.9380531	1578.125	109.375
4	106	0	229.8054245	1265.625	93.75
5	114	0	230.2631579	1328.125	93.75
6	107	0	249.853972	1546.875	93.75
7	110	0	269.1761364	2484.375	93.75
8	106	0	233.490566	1562.5	93.75
9	106	0	215.5070755	921.875	109.375
10	112	0	248.4654018	1453.125	93.75
11	100	0	242.03125	1000	93.75
12	113	0	223.5896018	1359.375	93.75
13	109	0	222.6204128	968.75	109.375
14	108	0	255.931713	1421.875	93.75
15	101	0	245.8230198	1421.875	109.375
16	107	0	235.5432243	1421.875	93.75
17	108	0	222.2222222	921.875	109.375
18	106	0	244.2511792	1609.375	93.75
19	103	0	237.1055825	1343.75	109.375
20	115	0	237.2282609	1687.5	93.75
Total:	2160	0	236.9062462	2484.375	93.75

Table 6.14 clearly indicates that the framework was able to respond to every search as the total number of failed search requests is zero. It can therefore be concluded that the test case objectives were achieved and the framework is capable of serving multiple applications simultaneously.

6.5 Solution Deployment

The framework has been deployed to a well-known bank to fulfil the client search requirements of an internal application and has been operational since mid July 2008.

In addition to performing ad-hoc real time searches (as per a user initiated search), the framework is utilised within nightly batch searches. As was described in Test Case 5 (§6.3.5 and §6.4.5), the framework is requested to perform on average 120 000 searches against a database of about 200 000 entries every night.

Due to the framework's design, it has been possible to make rapid changes to the search methodology to fall in line with changing business requirements. On average, changes to the fuzzy matching logic can be completed and implemented within three hours. The reason that changes can be implemented so quickly is mainly due to the framework's requirement for the loading of its fuzzy matching logic at Runtime. This

requirement forces the fuzzy matching logic to be contained within separate DLL's (to both the application and the framework). Generally, the sole focus of these DLL's is the implementation of fuzzy matching logic and therefore changes can be made quickly and easily. Furthermore, since the changes are not made to the framework itself, the framework does not need to be recompiled; only the custom logic DLL needs to be. Finally, since the framework runs as a separate application, the risk, to the application, caused by a bug being introduced into the fuzzy matching logic when it is being altered, is negligible as it will cause the framework's search to fail and this will have no further impact on the calling application as it is a separate application. This aids in the business's change control process as only the framework can be affected by errors in the fuzzy matching logic.

In conclusion, five test cases were developed in order to verify whether the following objectives within the framework's design had been achieved, namely:

- The ability to integrate third party code in order to implement a particular fuzzy matching algorithm.
- The abstraction of the search process.
- The resolution of the results of individual searches of several names in a series of related names into a single common result set.
- The capability of implementing multiple fuzzy matching algorithms.
- The provision of comparable search times to those of equivalent algorithm specific implementations of various fuzzy matching algorithms.
- The support of batch processes.
- The simultaneous serving of multiple applications.

In all but one of the test cases it was found that the framework achieved the defined objectives. As documented within this section, it was demonstrated that the framework did not provide equivalent search times to the algorithm specific implementations for all fuzzy matching algorithms. An investigation into the underlying issue identified that this shortcoming is limited to fuzzy matching

algorithms that only evaluate the fuzzy match subsequent to the database search and that the inefficient code can be easily corrected.

The following section provides the analysis of the framework and a conclusion to the dissertation.

7 Analysis and Conclusion

The final section critically analyses the framework design, provides future recommendations for the framework design based on the analysis and finally concludes this dissertation.

7.1 Critical Analysis

Throughout this dissertation various aspects of the framework's strengths and weaknesses have been discussed; however these discussions are found sporadically throughout the course of the dissertation. This section collates the various analyses and provides a summary thereof.

The following strengths have been identified within the framework's design:

- The framework is capable of implementing a variety of Fuzzy Matching algorithms (§5.4.1)
- The framework itself contains no Fuzzy Matching logic and rather relies on a third party to supply the logic. This aids in the framework's flexibility (§5.4.2)
- The framework has been constructed as a complete fuzzy matching solution and therefore the entire fuzzy matching process is abstracted away from the third party developer. This allows the developer to focus on fuzzy matching algorithm logic rather than the various housekeeping required to maintain a fuzzy matching system (§5.4.3)
- The framework incorporates the custom fuzzy matching logic at runtime, which enables it to adapt quickly to changing fuzzy matching logic requirements without the need to be recompiled (§5.4.4)
- The framework's generic design goes beyond its support for a variety of fuzzy matching algorithms, in that it does not limit the number of related names that are input into a single search nor does it limit the number of simultaneous searches that can be specified within a single search request (§5.4.5)

- Following from the previous strength the framework is capable of maintaining the relationships between two or more search names as is defined by the external application and hence is able to return meaningful results (§5.4.6)
- Through the use of the concept of data sets and the implementation of the framework as a web service, the framework is capable of serving multiple applications simultaneously, whilst still catering for each application's individual requirements (§5.4.7)

The following weaknesses have been identified within the framework's design:

- In order to allow the framework to be generic, the various aspects of fuzzy matching algorithms had to be broken down into functional components. Within each of these functional components, the framework has supplied an interface through which the corresponding custom logic can be implemented. The process of breaking a fuzzy matching algorithm into these components can be unintuitive and complex (§5.5.5)
- The framework's interfaces have been designed in order that they supply the developer with the necessary data. In the situation where the developer may require data that is not supplied by the interface, it would be very difficult for him / her to acquire this information, if at all possible (§5.5.6)
- As Test Case 4 demonstrated, the framework's requirement of runtime loading of DLL's and the invocation of previously unknown methods cause additional overhead to the search performance (§5.5.6)
- The implementation of the framework's pre-search filter requires that the framework run queries directly against the external application's database, which practically forces both the framework and the calling application to run on the same database. This inhibits the framework's ability to serve remote applications (§5.6.1)
- The framework's caching of the Pre-Search Filter results provides no mechanism to update the results and therefore over time these results could become stale and hence in invalid (§5.6.5)

- During the course of the framework's testing, a previously un-discussed weakness was found. The framework makes no provision for different conditional logic i.e. different data sets may require different fuzzy matching logic; rather at each interface point, the framework blindly invokes the specified custom methods without regard to the current data set. If the third party developer requires that different data sets are to have different logic applied to them, he/she needs to create validations within the custom code to ensure that it is only applied to the particular data sets. The implementation of the custom code in this manner forces the framework search to be less efficient as the framework still invokes every single configured method; despite it being that some of the methods will not be applicable for the current data set and will exit early. Furthermore, there is no workaround to this issue when one requires applying different logic to the same data set in different situations.

7.2 Recommendations and Future Developments

Having completed a thorough analysis of the framework design and implementation, several recommendations aimed at overcoming some the specified weaknesses are presented.

1. Instead of the framework invoking every single method specified in the configuration file, the configuration file should consist of multiple search configurations. These search configurations are used by the calling application to specify the search logic that is to be utilised by the framework in a particular search. This would be achieved through specifying the search configuration within the search request. Each search configuration is to specify the assemblies and methods that are to be used by the framework within each of its components; in addition they are to contain provision for parameters that are specific to a particular

configuration. Through the use of the search configurations the framework will only invoke methods that are necessary for a particular search. In addition, the use of search configurations allows one to apply different logic to the same data set in different situations.

2. Both the framework's implementation of the pre-search filter and the corresponding defined interface can be altered in order that the framework no longer performs the pre-search filter query itself but rather forces the user defined code to perform the query and then return the subset list back to the framework. This allows the framework to be more loosely coupled to the application as it need not connect to its database; allowing the framework and the external application to be run on separate databases.
3. The mechanism for the invocation of methods within external assemblies should be altered (see §6.4.4); instead of framework traversing the frame stack to determine the method's name (as currently both the wrapping method and invoked the external assembly method share the same name), the invoked method name must be explicitly supplied. This explicit specifying of the method name should not cause any issues as the methods that are being called are defined by the framework's interfaces and are already known at compile time.
4. An assembly factory could be incorporated into the framework, into which all configured assemblies are loaded at start-up. This would reduce additional delay to the framework's search as there would be no lag due the loading of assemblies into memory as they would already be there. Alternatively, the framework could maintain the most recently used assemblies in memory and only after a configured amount of time has elapsed, are they unloaded.
5. The FMFRules class's LoadRulesTable method (which calls the delegate that was passed to class) can be changed from being a private method to being public one. This would provide the third party custom code with the

ability to update the rules table itself to ensure that the in-memory rules table does not become stale over time.

6. Currently the pre-search filter only caches the results for the most recent pre-search filter argument. It would be useful if the framework would cache the results of last few pre-search filter arguments. For example the framework could store the results of the last ten unique arguments. This could be stored in a First In First Out system.

7. Currently the framework maintains and processes all entries that have been retrieved from the database, which in the case of Edit Distance and N-Gram like algorithms, results in the framework processing an excessively large, mostly irrelevant number of entries throughout the majority of the search. It would be advantageous to apply the threshold cut-off immediately after the match scoring has been performed, where entries whose match scores fall below the threshold are immediately removed from the list of potential matches[24]. This prevents the framework from having to process a large list of entries where the majority of entries are not relevant. In addition, this would reduce framework overheads, as the sorting of entries (according to their respective score), which is an already expensive operation, would be optimised as there are fewer entries to sort.

8. The scoring of the returned database entries is one of the most computationally expensive and longest operations performed by the framework. This is due to the potentially large number of entries that are required to be processed and the fact they are processed sequentially. There is no need to process these entries sequentially as they are independent of one another. It would be advantageous to perform this process in parallel, where the framework can spin off multiple processing threads and therefore evaluate the scores of multiple matches simultaneously.

[24] Currently the threshold evaluation is only performed at the end of the search process.

9. An additional feature that could be added to the framework is the ability to perform a comparison between two input names (as opposed to performing a database search against a single input name), in which the framework computes and returns the degree of the match between the two names.

7.3 Conclusion

Through the analysis of the characteristics of personal and corporate names and those of multiple fuzzy matching algorithms, a generic framework was developed that is capable of performing custom name searches. The framework does not subscribe to any fuzzy matching algorithm but rather, through the use of interfaces, provides a platform through which custom fuzzy matching logic can be implemented. Furthermore, the framework has been designed with the intention that fuzzy matching logic can be quickly changed in order to conform to non-technical (business) requirements. This has been achieved through the abstraction of the search and maintenance processes away from the person developing the fuzzy matching logic. An inherent aspect within name matching is that names do not exist in isolation but rather a relationship can exist between multiple names. For example, a person's first name, middle name and surname are all related as they all belong to the same person. The framework has been designed to cater for this unique aspect of name matching through the independent search of each of the individual names and thereafter the collation of the search results into a single unique result set to ensure that the returned result set maintains the relationships defined by the input search names.

Through a series of test cases and the deployment of an instance of the framework into a corporate environment, the framework's design and implementation have been thoroughly tested. The framework has been proven to be highly flexible in implementing a variety of fuzzy matching algorithms that can be altered and developed within a minimum amount of time. In addition, it has been shown that the

framework is successfully capable of maintaining the relationships between the input search names. The framework's main shortcoming, which was demonstrated by the test cases, is its search times. When implementing algorithms that are able to search the database for potential matches, the framework is comparable to the algorithm specific implementation; however the framework has performed poorly when implementing algorithms that are unable to perform pin-point database searches and require that a large number of entries be retrieved from the database and their degree of match is evaluated thereafter. It has been identified that there are three factors that cause this undesirable performance, namely: limitations within the framework's design (the requirement to perform a stack trace when invoking an external method), the generic nature of the framework and the fact that the framework is an entire fuzzy matching solution and not just a fuzzy matching implementation. Of the identified factors, the limitations in the framework's design cause the majority of the latency; however, the underlying problems and their resolutions have been discussed and the necessary remedies can be easily implemented.

Through the course of this dissertation the three research questions have been answered. It has been found that the majority of fuzzy matching algorithms can be broken into the following series of high level logical components; namely input processing, match searching and match evaluation. Based on these commonalities a generic framework could be built to cater for a variety of algorithms, thereby answering the first research question[25]. Furthermore, the characteristics of names have been successfully incorporated within the framework in that the framework enables a variety of different fuzzy matching algorithms to perform searches on various combinations of names, whilst maintaining the relationships defined by the input search names within the returned results. This functionality clearly demonstrates that the framework can cater for the requirements of various

[25] Do the majority of Fuzzy Matching algorithms contain one or more common high level processes that can be integrated into a single generic framework? (found in §3.1)

combinations of names and thus answers the second research question[26]. The third research question[27] is answered through the framework's use of interfaces. Through the use of its defined interfaces, the framework is capable of incorporating custom fuzzy matching logic.

It is important to mention that the scope of the dissertation has grown during the course of its write-up. This can be attributed to two factors:

1. As further research was performed, it became evident that scope of the original thesis would not be particularly useful.
2. The deployment of the framework within a real-world context caused the framework to grow to fit with changing business requirements.

The framework was originally scoped to be used as a test harness to aid in the rapid development of fuzzy matching algorithms. It was intended that the framework be used as a mechanism where one could both implement and test changes to an algorithm as the framework had been designed to load external assemblies at Runtime. Through the course of this dissertation, the framework has grown into a complete fuzzy matching solution, which is intended to provide custom fuzzy matching functionality to external applications whilst abstracting the fuzzy matching logic developer away from both the database maintenance and search process. To summarise, the framework provides the following functionality:

- A fuzzy search, in which the returned results are application specific
- A self maintained system, in which the internal database is constantly being maintained to ensure that it's up to date with external databases and all the necessary processing has been performed
- The external developer has minimal interaction with the actual mechanics of the framework's search. The developer need only

[26] Is it possible for this generic framework to cater for the requirements of the matching of various combinations of names? (found in §3.1)
[27] Can this framework provide a mechanism for the implementation of custom logic for the common processes? (found in §3.1)

concern himself / herself with the fuzzy matching algorithm's specific issues, which is incorporated into the framework's search by means of the defined interfaces

- The framework is geared towards the implementation of rapid changes in business logic. This allows changes to be made quickly to the fuzzy matching logic with minimal risk to the external application (requesting the fuzzy search). As discussed at length, these changes can be made without the need for either the external application or the framework to be rebuilt

References

Alonso, G., Casati, F., Kuno, H. & Machiraju, V. (2004) Web Services. In Alonso, G., Casati, F., Kuno, H. & Machiraju, V. *Web services: concepts, architectures and applications*. Berlin, Heidelberg, Germany: Springer-Verlag. Ch. 5. pp.124-32.

Atallah, M.J., ed. (1999) *Algorithms and Theory of Computation Handbook*. New York, USA: CRC Press.

Bacon, D.F. (2007) Realtime Garbage Collection. *ACM Queue*, February. pp.40-49.

Bartolini, I., Ciaccia, P. & Patella, M. (2002) String Matching with Metric Trees using an Approximate Distance. In *SPIRE 2002: Proceedings of the 9th Symposium on String Processing and Information Retrieval*. Lisbon, Portugal, 2002. Springer-Verlag.

Bell, G.B. & Sethi, A. (2001) Matching records in a national medical patient index. *Communications of the ACM*, vol. 44, no 9, pp.83-88.

Bentley, J.L. & Sedgewick, R. (1997) Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. New Orleans, Louisiana, United States, 1997. ACM.

Birkby, R. (2002) *A SoundEx implementation in.NET*. [Internet] http://www.codeproject.com/KB/recipes/soundex.aspx Accessed 8 November 2009.

Branting, L.K. (2003) A comparative evaluation of name-matching algorithms. In *Proceedings of the 9th international conference on Artificial intelligence and law*. Scotland, United Kingdom, 2003. ACM Press.

Capretz, L.F. (2003) A brief history of the object-oriented approach. *ACM SIGSOFT Software Engineering Notes*, vol. 28, no 2, p.6.

Chappell, D. (2002).NET Languages. In Chappell, D. *Understanding.NET: A Tutorial and Analysis*. Boston, USA: Pearson Education. Ch. 4. pp.119-66.

Christen, P. (2006) A Comparison of Personal Name Matching: Techniques and Practical Issues. In *ICDMW '06: Proceedings of the Sixth IEEE International Conference on Data Mining - Workshops*. Washington, DC, 2006. IEEE Computer Society.

Cohen, A.T. (1984) Data abstraction, data encapsulation and object-oriented programming. *ACM SIGPLAN Notices*, vol. 19, no 1, pp.31-35.

Dao, T. (2005) *Term frequency/Inverse document frequency implementation in C#*. [Internet] http://www.codeproject.com/KB/cs/tfidf.aspx Accessed 9 November 2009.

Du, M. (2005) *Approximate Name Matching: Finding Similar Personal Names in Large International Lists*. Masters Thesis. Stockholm: Royal Institute of Technology.

Ecma International (2006) *C# Language Specification*. [Internet] Ecma International (4) http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf Accessed 6 July 2011.

Ecma International (2009) *What is Ecma International*. [Internet] http://www.ecma-international.org/memento/index.html Accessed 7 February 2010.

Elmagarmid, A.K., Ipeirotis, P.G. & Verykios, V.S. (2007) Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, January. pp.1-16.

Foidl, G. (2009) *Fuzzy Search*. [Internet] http://www.codeproject.com/KB/recipes/fuzzysearch.aspx Accessed 8 November 2009.

Giannini, M. (2004) C/C++. In H. Bidgoli, ed. *The Internet encyclopedia*. Hoboken, New Jersey: John Wiley & Sons. p.164.

Haigh, T. (2006) "A veritable bucket of facts" origins of the data base management system. *ACM SIGMOD Record*, vol. 35, no 2, pp.33-49.

Hsiung, P., Moore, A., Neill, D. & Schneider, J. (2005) Alias Detection in Link Data Sets. In *Proceedings of the International Conference on Intelligence Analysis*. McLean, VA, 2005.

Hu, C. (2006) When to use an interface? *ACM SIGSE Bulletin*, vol. 38, no 2, pp.86-90.

IBM (2006) *Structured Query Language (SQL)*. [Internet] http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0004100.htm Accessed 17 July 2011.

ISO/IEC JTC 1 (2007) Annex J: Guidelines for API standardization. In *ISO/IEC JTC 1 Directives*. 3rd ed. New York, NY. p.146.

Jagger, J., Perry, N. & Sestoft, P. (2007) Delegates. In Jagger, J., Perry, N. & Sestoft, P. *Annotated C# Standard*. Burlington, MA, USA: Morgan Kaufmann Publishers. Ch. 22. pp.604-11.

Kachru, S. & Gehringer, E.F. (2004) On the relative advantages of teaching Web Services in J2EE vs..NET. In *Educator's Symposium, OOPSLA 2004: object-oriented languages*. Vancouver, Canada, 2004.

Karne, R.K. (1995) Object-oriented computer architectures for new generation of applications. *ACM SIGARCH Computer Architecture News*, vol. 23, no 5, pp.8-19.

Kolatch, E., Toye, J. & Dorr, B. (2004) *Look Alike / Sound Alike Algorithms for Assessing Drug Name Similarities*. McLean, Virginia: Project Performance Corporation.

Korson, T. & McGregor, J.D. (1990) Understanding object-oriented: a unifying paradigm. *Communications of the ACM*, vol. 33, no 9, pp.40-60.

Lait, A.J. & Randell, B. (1993) *An assessment of name matching algorithms*. Technical Report. Newcastle upon Tyne: University of Newcastle upon Tyne.

Markiewicz, M.E. & de Lucena, C.J.P. (2001) Object oriented framework development. *Crossroads*, Summer. pp.3-9.

Maurer, W.D. & Lewis, T.G. (1975) Hash Table Methods. *ACM Computing Surveys (CSUR)*, vol. 7, no 1, pp.5-19.

Media24 (2009) *Who's Who of Southern Africa*. [Internet] http://www.whoswhosa.co.za Accessed 19 October 2009.

Microsoft (2009).*NET Framework Overview*. [Internet] http://www.microsoft.com/net/overview.aspx Accessed 12 July 2011.

Microsoft (2011) *Microsoft SQL Server*. [Internet] http://technet.microsoft.com/en-us/library/bb545450.aspx Accessed 20 July 2011.

MSDN Library (2009a) *Dictionary <(Of <(TKey, TValue>)>) Class*. [Internet] http://msdn.microsoft.com/en-us/library/xfhwa508.aspx Accessed 6 September 2009.

MSDN Library (2009b) *Full-Text Search Overview*. [Internet] http://msdn.microsoft.com/en-us/library/ms142547.aspx Accessed 17 December 2009.

MSDN Library (2011a) *Common Language Runtime (CLR)*. [Internet] http://msdn.microsoft.com/en-us/library/8bs2ecf4.aspx Accessed 17 July 2011.

MSDN Library (2011b) *DLLs*. [Internet] http://msdn.microsoft.com/en-us/library/1ez7dh12.aspx Accessed 17 July 2011.

Naugler, D.R. (2004) Delegates and functional programming. In *Proceedings of the 2nd annual conference on Mid-South College computing*. Little Rock, Arkansas, 2004. Mid-South College Computing Conference.

Navarro, G. (2001) A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)*, vol. 33, no 1, pp.31-88.

Oracle Corporation (2010) *Introduction to Oracle Database*. [Internet] http://download.oracle.com/docs/cd/E11882_01/server.112/e16508/intro.htm#autoId1 Accessed 20 July 2011.

Pang, A.Y. (2007) *Implementation of the Smith-Waterman Algorithm on the FLEET simulator*. Master's Thesis. Vancouver, Canada: The University of British Columbia.

Piskorski, J., Sydow, M. & Kupść, A. (2007) Lemmatization of Polish person names. In *Proceedings of the Workshop on Balto-Slavonic Natural Language Processing: Information Extraction and Enabling Technologies*. Prague, Czech Republic, 2007. Association for Computational Linguistics.

Python Software Foundation (2004) *Writing a Python Script*. [Internet] http://docs.python.org/release/2.4/mac/IDEwrite.html Accessed 17 July 2011.

Python Software Foundation (2007) *What is Python? Executive Summary*. [Internet] http://www.python.org/doc/essays/blurb/ Accessed 17 July 2011.

Rajković, P. & Janković, D. (2007) Adaptation and application of Daitch–Mokotoff SoundEx algorithm on Serbian names. In *XVII Conference on Applied Mathematics*. Novi Sad, Serbia, 2007.

Riele, D. (2000) No. 13509 *Framework Design: A Role Modeling Approach*. PhD Thesis. Zurich: Swiss Federal Institute of Technology.

Salmela, L., Tarhio, J. & Kytöjoki, J. (2007) Multipattern string matching with q-grams. *Journal of Experimental Algorithmics (JEA)*, vol. 11, pp.1-19.

SAP Library (2009) *Database Triggers*. [Internet] http://help.sap.com/saphelp_nw70/helpdata/en/08/db4940f0030272e10000000a155106/content.htm Accessed 20 July 2011.

Schmolitzky, A. (2004) "Objects first, interfaces next" or interfaces before inheritance. In *Conference on Object Oriented Programming Systems Languages and Applications, Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. Vancouver, BC, Canada, 2004. ACM.

Seltzer, M. (2005) Beyond Relational Databases. *Queue*, vol. 3, no 3, pp.50-58.

Siegel, D.E. (1999) All searches are divided into three parts: string searches using ternary trees. *ACM SIGAPL APL Quote Quad*, vol. 29, no 3, pp.57-68.

Snae, C. (2007) A Comparison and Analysis of Name Matching. *Transactions on Engineering, Computing and Technology*, vol. 19, pp.252-57.

Stroustrup, B. (1997) *The C++ Programming Language*. 3rd ed. Reading, Massachusetts: Addison-Wesley.

Sun Microsystems, Inc (1999) *The Java Virtual Machine Specification*. [Internet] http://java.sun.com/docs/books/jvms/second_edition/html/Introduction.doc.html Accessed 17 June 2011.

Sun Microsystems, Inc (2000) *Java Language Specification*. [Internet] http://java.sun.com/docs/books/jls/second_edition/html/intro.doc.html#237601 Accessed 17 July 2011.

Taft, R.L. (1970) *Name search techniques*. Special Report No. 1. Albany, New York: New York State Identification and Intelligence System.

U.S. Census Bureau (2009) *Frequently Occurring First Names and Surnames From the 1990 Census*. [Internet] http://www.census.gov/genealogy/names/names_files.html Accessed 19 October 2009.

Veronica, R. & Li, C. (2009) Efficient top-k algorithms for fuzzy search in string collection. In *Proceedings of the First international Workshop on Keyword Search on Structured Data*. Providence, Rhode Island, 2009. ACM.

W3C (2004) *XML Schema Part 0: Primer Second Edition*. [Internet] http://www.w3.org/TR/xmlschema-0/ Accessed 20 July 2011.

W3C (2008) *Extensible Markup Language (XML) 1.0*. [Internet] http://www.w3.org/TR/REC-xml/ Accessed 20 Juy 2011.

Yancey, W.E. (2005) Statistics: 2005-05 *Evaluating String Comparator Performance for Record Linkage*. Technical Report. Washington, DC: U.S. Census Bureau.

Yang, X., Wang, B. & Li, C. (2008) Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. Vancouver, Canada, 2008. ACM.

Zobel, J. & Dart, P.W. (1995) Finding Approximate Matches in Large. *Software - Practice & Experience*, vol. 25, no 3, pp.331-45.

Zobel, J. & Dart, P.W. (1996) Phonetic string matching: lessons from information retrieval. In *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*. Zurich, 1996. ACM Press.

Appendix A Edit Distance Variants

Several variants of the basic Edit Distance algorithm have been developed in order to overcome particular obstacles that occur when matching strings.

1 Weighted Edit Distance

Some letters are more easily confused than others (due to "keyboard layout, similar shapes or phonetic similarity" (Du, 2005)) and therefore it is obvious that common errors should cost less than unusual errors. Bearing this in mind, the Weighted Edit Distance, assigns different weightings to the different letter pairs, instead of the standard unit cost (for unlike letters) as specified in the basic Edit Distance algorithm (Du, 2005).

It must be noted that the various weightings for the various letter pairs are highly statistical and depend on the means of input, the language used and the nature of the text involved. The algorithm therefore requires that an additional table of size $(l \times l)$[1], which maps the weightings of the various letter pairs. Furthermore, a one dimensional matrix is required (of length l) in order to map the weightings for when a letter is being compared to no letter (in the case of insertions and substitutions) (Du, 2005).

2 Edit Distance with Upperbound and Cut-Off Criterion

As discussed previously, one of the shortcomings of the Edit-Distance algorithm is that every single entry within a dictionary needs to be processed in order to determine if there are any matching words to the search word. The Edit Distance with Upperbound and Cut-Off Criterion attempts to improve the search time by placing an

[1] l is the size of the alphabet.

error threshold within which matches can be found (Du, 2005). Equation 1 shows that for two words to be a match they must have a maximum *t* edit distance (Du, 2005).

$$|x| - t \leq |y| \leq |x| + t \qquad (1)$$

where: *t* is the threshold error distance

In this manner, if during the calculation of the edit distance, the threshold is exceeded, the calculation (for that particular word within the dictionary) is aborted and the search continues with the next word (Du, 2005).

3 Normalised Edit Distance (NED)

The NED is calculated in exactly the same manner as the Edit Distance; however, after the Edit Distance is computed, the computed value is divided by the length of the longer of the two compared strings (Hsiung et al., 2005). See Equation 2 (Hsiung et al., 2005).

$$NED(x, y) = \frac{edit(x, y)}{\max(|x|, |y|)} \qquad (2)$$

where: x is the first string being compared

y is the second string being compared

Normalising the edit distances allows one to be able to compare the edit distance of various string pairs regardless of their different word lengths as all the results are normalised and will therefore all have values between zero and one.

4 Discretised Edit Distance (DED)

The DED forces the result of all string comparisons to be placed within the discrete binary set of either one or zero. This is achieved by setting the result of a NED to one if the value is above the specified threshold of 0.7 otherwise, the value is set to zero. The threshold of 0.7 was determined purely by empirical observation (Hsiung et al., 2005).

5 Exponential Edit Distance (EED)

The EED is calculated by passing the result of the basic Edit Distance through an exponential function. See Equation 3 (Hsiung et al., 2005).

$$EED(x, y) = \exp(edit(x, y)) \quad (3)$$

6 Edit Distance with a Trie

The Edit Distance with a Trie algorithm attempts to improve on the Edit Distance algorithm search speed when a large dictionary of names is required to be searched. Using a trie, the algorithm is able to pre-store the results of the edit distance calculations for a particular search word.

The trie used in this algorithm is a tree with labelled edges where every node corresponds to a unique prefix that belongs to one or more words. The method in which the trie is built is now discussed. The trie's root node corresponds to an empty string, ε. When adding a new word to the trie, the algorithm iterates through each of the letters contained within the word. For each letter, the algorithm checks if there are any nodes at the same level as the letter that correspond to it, e.g. if it is the first letter of the word, the algorithm looks at all nodes at the 1st level (the first level of nodes after the root node) of the trie. If a matching node is found, the algorithm traverses the branch to the matching node. If no matches are found, a new edge is formed by

branching away from the previous node. This process is repeated for every word the dictionary. Due to the manner in which the trie is constructed, each leaf[2] in the trie corresponds to a unique word. It must be noted that non-leaf nodes can also correspond to complete words (Du, 2005). Please refer to Figure A.1 as an example of how a trie is built.

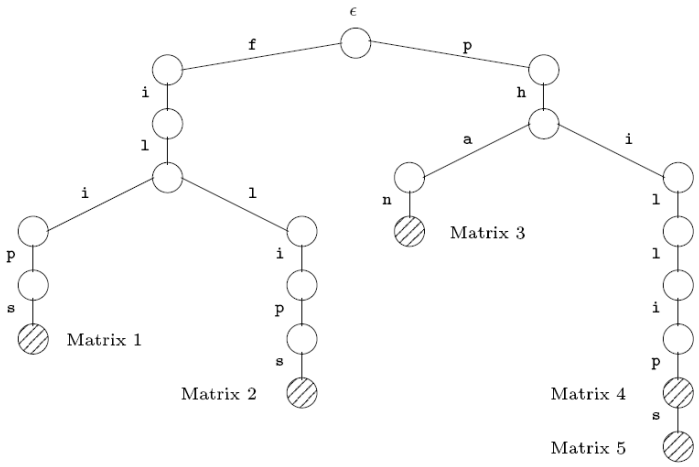


Figure A.1: A trie built using the names filips, fillips, phan, phillip and phillips (Du, 2005)

The names filips and fillips (in the above mentioned example) both have the same prefix of "fil" and therefore share the same first four nodes. The trie thereafter splits at the "i" in filips and the second "l" in fillips. The trie can now be searched to locate the word in the dictionary that has the minimum edit distance to the search word.

The algorithm attempts to match the search word by traversing the trie depth first[3]. For every node that is traversed, a new column of the Edit Distance dynamic programming matrix is calculated. The first column that is populated corresponds to the root node of the trie (the root node corresponds to an empty string, ε) and therefore is the common prefix to all the strings in the dictionary (Du, 2005). "The

[2] A leaf is the end node of a tree, to which there are no further nodes attached.
[3] A depth first search is performed by starting at the root of the trie and then explores as far down as possible along each branch before backtracking. Backtracking is achieved by the algorithm returning to the most recent node that it has not fully explored (Wikipedia, 2007).

branches are then visited recursively. Children nodes generate their column from the columns of their predecessors. As a result, two words having a common prefix will also share the matrix columns up until the column corresponding to that prefix" (Du, 2005). When a node marked as the end of a specific word is reached, the last cell of the newly computed column contains the minimum edit distance between that word and the search word (Du, 2005). Through the searching of the trie in this manner, repeated calculations of the edit distances for words that have common prefixes are avoided. Figure A.2 (in conjunction with Figure A.1) describes the previously explained logic.

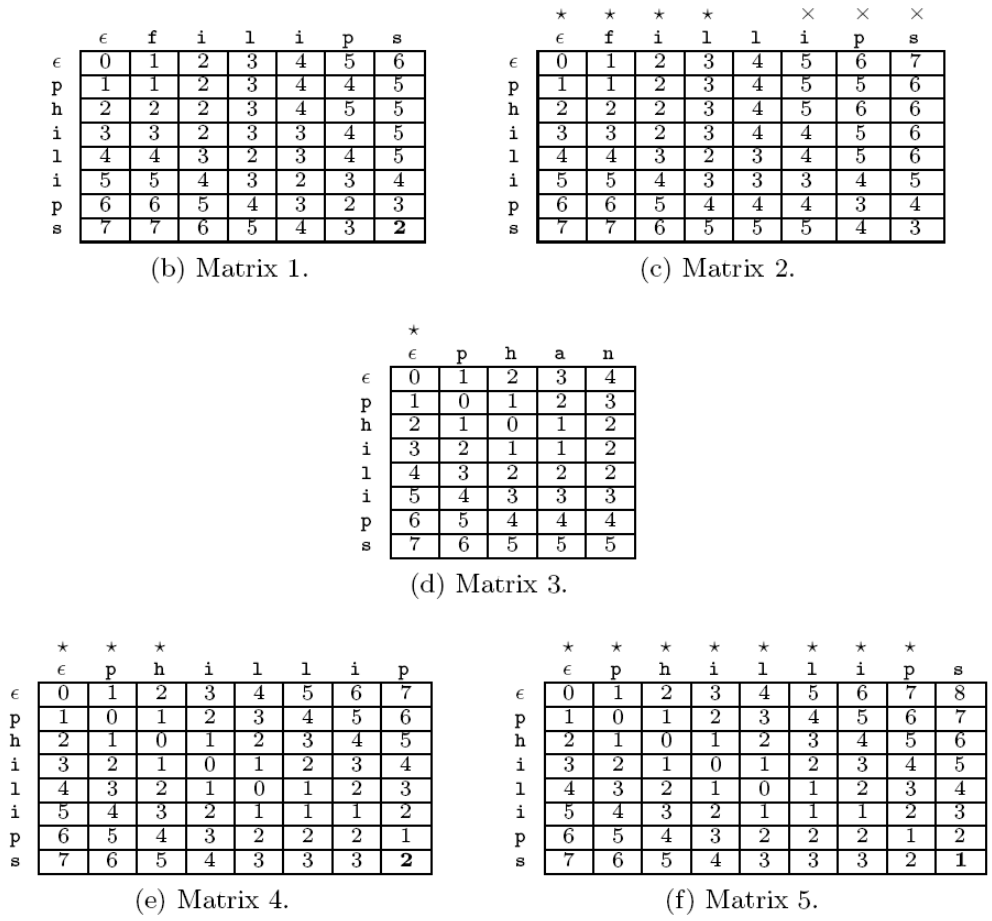


Figure A.2: Construction of the Dynamic Programming Matrices using the Trie

The figure above depicts the columns that have been pre-calculated from previous matrices, through the use of a star (★). A further advantage of the use of a trie, is that if one has specified an error threshold, which the minimum edit distance must not exceed, one is often able to determine if a branch has exceeded the threshold without need to calculate the edit distance for the leaf node. This is due to the fact that once the threshold has been reached; no further nodes along the branch will have a lower edit distance. The matrices in Figure A.2 have an error threshold of 2. Once there is a column in the matrix with all the cells above threshold, there is no further need to calculate the remaining columns of the matrix. This can be seen by columns marked with a cross (×) in Matrix 2 (Du, 2005).

References

Du, M. (2005) *Approximate Name Matching: Finding Similar Personal Names in Large International Lists*. Masters Thesis. Stockholm: Royal Institute of Technology.

Hsiung, P., Moore, A., Neill, D. & Schneider, J. (2005) Alias Detection in Link Data Sets. In *Proceedings of the International Conference on Intelligence Analysis*. McLean, VA, 2005.

Wikipedia (2007) *Depth-First Search*. [Internet] Available at: http://en.wikipedia.org/Depth_first_search Accessed 4 July 2007.

Appendix B Database Model

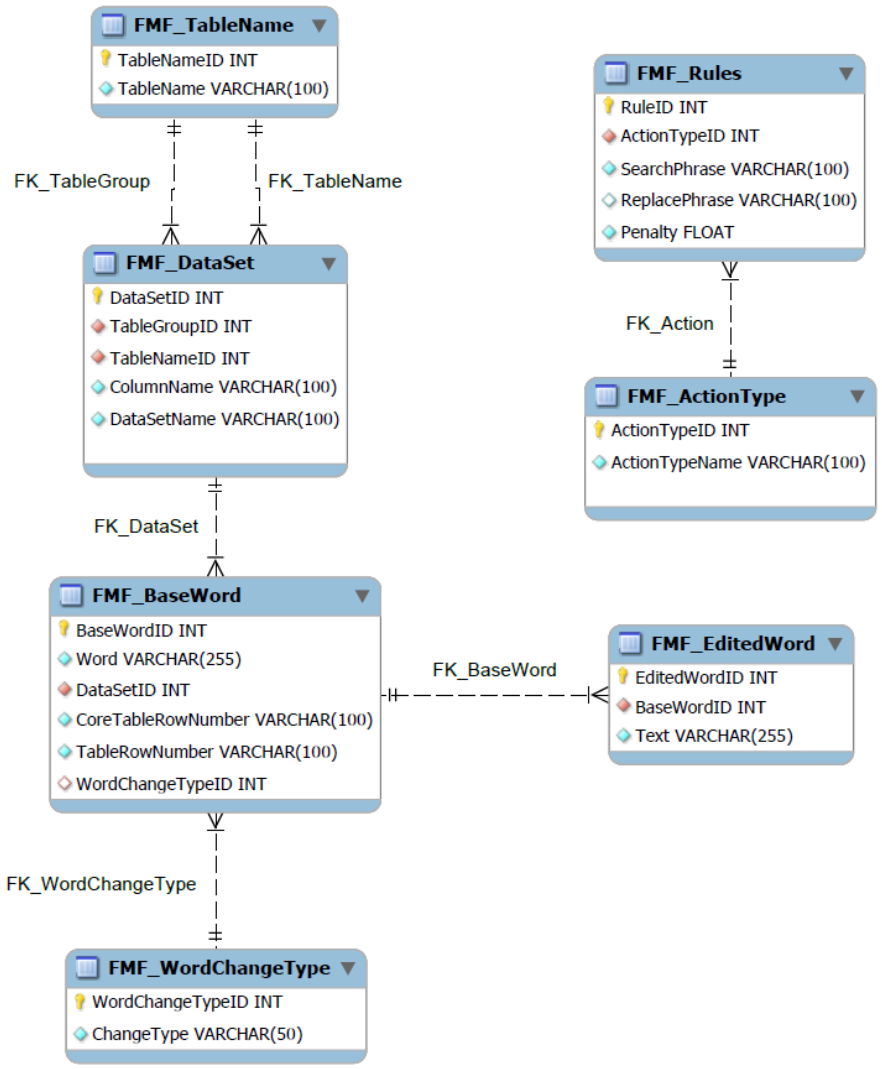


Figure B.1: Framework Database Model

Appendix C Search Input XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:fmf="urn:synthesis-co-za:fmf"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="urn:synthesis-co-za:fmf"
elementFormDefault="unqualified">

	<xsd:complexType name="BaseIDDef" mixed="false">
		<xsd:attribute name="IdString" type="xsd:string">
			<xsd:annotation>
				<xsd:appinfo>
					Generic ID String, usually refers to the primary
					key of the object
				</xsd:appinfo>
			</xsd:annotation>
		</xsd:attribute>
		<xsd:attribute name="IdType" type="xsd:string" use="optional">
			<xsd:annotation>
				<xsd:appinfo>
					Generic ID Type, refers to the type of field the
					IdString is
				</xsd:appinfo>
			</xsd:annotation>
		</xsd:attribute>
		<xsd:attribute name="IdSource" type="xsd:string" use="optional">
			<xsd:annotation>
				<xsd:appinfo>
					Generic ID Source, refers to the source of the
					IdString
				</xsd:appinfo>
			</xsd:annotation>
		</xsd:attribute>
	</xsd:complexType>

	<xsd:complexType name="NamedIDDef" mixed="false">
		<xsd:complexContent mixed="false">
			<xsd:extension base="fmf:BaseIDDef">
				<xsd:sequence>
					<xsd:element name="Name" type="xsd:string"
					minOccurs="0">
						<xsd:annotation>
							<xsd:appinfo>
								The name of this object.
```

```
                                        </xsd:appinfo>
                                    </xsd:annotation>
                                </xsd:element>
                            </xsd:sequence>
                        </xsd:extension>
                    </xsd:complexContent>
                </xsd:complexType>

                <xsd:complexType name="InputWordDef" mixed="false">
                    <xsd:sequence>
                        <xsd:element name="Word" type="xsd:string">
                            <xsd:annotation>
                                <xsd:appinfo>
                                    The word against which a search is to be
                                    performed / The resultant word retrieved
                                    from a search
                                </xsd:appinfo>
                            </xsd:annotation>
                        </xsd:element>
                        <xsd:element name="DataSetID" type="fmf:NamedIDDef"
                        maxOccurs="unbounded">
                            <xsd:annotation>
                                <xsd:appinfo>
                                    The dataset against which the search word
                                    is to be searched / The dataset to which
                                    the retrieved word belongs
                                </xsd:appinfo>
                            </xsd:annotation>
                        </xsd:element>
                    </xsd:sequence>
                </xsd:complexType>

                <xsd:complexType name="InputDataSetWordDef" mixed="false">
                    <xsd:complexContent mixed="false">
                        <xsd:extension base="fmf:InputWordDef">
                            <xsd:sequence>
                                <xsd:element name="WordWeighting"
                                type="xsd:double">
                                    <xsd:annotation>
                                        <xsd:appinfo>
                                            The weighting of the word
                                            when a match is found (for
                                            scoring puposes). e.g. the
                                            weighting associated with
                                            finding a match on a
                                            surname would be greater
```

```
                                                        than finding a match on
                                                        the firstname
                                                </xsd:appinfo>
                                        </xsd:annotation>
                                </xsd:element>
                        </xsd:sequence>
                </xsd:extension>
        </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="InputSetDef" mixed="false">
        <xsd:sequence>
                <xsd:element name="WordSetComponents"
                type="fmf:InputDataSetWordDef" maxOccurs="unbounded">
                        <xsd:annotation>
                                <xsd:appinfo>
                                        This is a set of search words that are to
                                        be 'anded' together in a search. Eg.
                                        Firstname and Surname
                                </xsd:appinfo>
                        </xsd:annotation>
                </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="dummy" type="xsd:string" />
</xsd:complexType>

<xsd:complexType name="PerformSearchRequest" mixed="false">
        <xsd:sequence>
                <xsd:element name="SearchSets" type="InputSetDef"
                maxOccurs="unbounded">
                        <xsd:annotation>
                                <xsd:appinfo>
                                        Each set represents a distinct search.
                                        The sets are to be 'ored' together
                                </xsd:appinfo>
                        </xsd:annotation>
                </xsd:element>
        </xsd:sequence>
</xsd:complexType>

</xsd:schema>
```

Appendix D Framework Defined Interfaces for Third Party Code

```
using System;
using System.Collections.Generic;
using System.Text;

namespace FMFBaseComponents
{

    public enum SQLClauseEnum
    {
        like, equals, notEqual, notLike, isNull, isNotNull
    };

    public enum SQLOperatorEnum
        {
          wildcard, upper, lower
        };

    public class SearchWordClass
    {
        private string _word;
        private List<string> _searchWords;

        public string Word
        {
            get { return _word; }
            set { _word = value; }
        }

        public List<string> SearchWords
        {
            get { return _searchWords; }
            set { _searchWords = value; }
        }
    }
```

```
public class DatabaseSearchParameters
{
    private string _searchWord;
    SQLClauseEnum _SQLClause;
    List<int> _dataSet;

    public string SearchWord
    {
        get { return _searchWord; }
        set { _searchWord = value; }
    }

    public SQLClauseEnum SQLClause
    {
        get { return _SQLClause; }
        set { _SQLClause = value; }
    }

    public List<int> DataSet
    {
        get { return _dataSet; }
        set { _dataSet = value; }
    }

    public DatabaseSearchParameters()
    {
        _SQLClause = SQLClauseEnum.equals;
    }

    public DatabaseSearchParameters(string SearchWord, List<int> Dataset)
    {
        _SQLClause = SQLClauseEnum.equals;
        _searchWord = SearchWord;
        _dataSet = Dataset;
    }
}

public class MatchScoring
{
    private string _orginalWord;
    private string _matchWord;
    private int _dataSetID;
    private double _score;

    public string OriginalWord
```

```
        {
            get { return _orginalWord; }
            set { _orginalWord = value; }
        }

        public string MatchWord
        {
            get { return _matchWord; }
            set { _matchWord = value; }
        }

        public int DataSetID
        {
            get { return _dataSetID; }
            set { _dataSetID = value; }
        }


        public double Score
        {
            get { return _score; }
            set { _score = value; }
        }

        public MatchScoring(string OriginalWord, string MatchWord, int DataSetID,
double Score)
        {
            _orginalWord = String.Copy(OriginalWord);
            _matchWord = String.Copy(MatchWord);
            _dataSetID = DataSetID;
            _score = Score;
        }
    }

    public class MatchScoringInformation
    {
        private double _score;
        private int _dataSetID;
        private double _wordWeighting;

        public double Score
        {
            get { return _score; }
            set { _score = value; }
        }
```

```
        public int DataSetID
        {
            get {return _dataSetID;}
            set {_dataSetID = value;}
        }

        public double WordWeighting
        {
            get {return _wordWeighting;}
            set {_wordWeighting = value;}
        }

    }

    public interface IDataTransformationInterface
    {
        SearchWordClass TransformedInputData(SearchWordClass InputData, List<int>
DataSetID);
    }

    public interface ISearchParameterInterface
    {
        DatabaseSearchParameters SetDatabaseSearchCondition(String SearchWord,
List<int> DataSetID);
    }

    public interface ISearchResultScoringInterface
    {
        MatchScoring ScoreMatchResult(MatchScoring InputData);
    }

    public interface IDataSetScoreAggregationInterface
    {
        double AggregateScores(List<MatchScoringInformation> DataSetScores);
    }

    public interface IMatchEvaluationInterface
    {
        List<int> EvaluateReturnList(List<double> DataSetScores);
    }
}
```

Appendix E Search Output XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:fmf="urn:synthesis-co-za:fmf"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" targetNamespace="urn:synthesis-co-za:fmf"
elementFormDefault="unqualified">

	<xsd:complexType name="BaseIDDef" mixed="false">
		<xsd:attribute name="IdString" type="xsd:string">
			<xsd:annotation>
				<xsd:appinfo>Generic ID String, usually refers to the
				primary key of the object</xsd:appinfo>
			</xsd:annotation>
		</xsd:attribute>
		<xsd:attribute name="IdType" type="xsd:string" use="optional">
			<xsd:annotation>
				<xsd:appinfo>Generic ID Type, refers to the type of
				field the IdString is</xsd:appinfo>
			</xsd:annotation>
		</xsd:attribute>
		<xsd:attribute name="IdSource" type="xsd:string" use="optional">
			<xsd:annotation>
				<xsd:appinfo>Generic ID Source, refers to the source of
				the IdString</xsd:appinfo>
			</xsd:annotation>
		</xsd:attribute>
	</xsd:complexType>

	<xsd:complexType name="NamedIDDef" mixed="false">
		<xsd:complexContent mixed="false">
			<xsd:extension base="fmf:BaseIDDef">
				<xsd:sequence>
					<xsd:element name="Name" type="xsd:string"
					minOccurs="0">
						<xsd:annotation>
							<xsd:appinfo>The name of this
							object.</xsd:appinfo>
						</xsd:annotation>
					</xsd:element>
				</xsd:sequence>
			</xsd:extension>
		</xsd:complexContent>
	</xsd:complexType>
```

```
<xsd:complexType name="OutputWordDef" mixed="false">
	<xsd:sequence>
		<xsd:element name="Word" type="xsd:string">
			<xsd:annotation>
				<xsd:appinfo>The resultant word retrieved from a
				search</xsd:appinfo>
			</xsd:annotation>
		</xsd:element>
		<xsd:element name="DataSetID" type="fmf:NamedIDDef">
			<xsd:annotation>
				<xsd:appinfo>The dataset to which the retrieved
				word belongs</xsd:appinfo>
			</xsd:annotation>
		</xsd:element>
	</xsd:sequence>
</xsd:complexType>

<xsd:complexType name="SearchResultDef" mixed="false">
	<xsd:sequence>
		<xsd:element name="WordSetComponents" type="fmf:OutputWordDef"
		maxOccurs="unbounded">
			<xsd:annotation>
				<xsd:appinfo>The returned words</xsd:appinfo>
			</xsd:annotation>
		</xsd:element>
		<xsd:element name="Score" type="xsd:double">
			<xsd:annotation>
				<xsd:appinfo>How good a match the result set
				is</xsd:appinfo>
			</xsd:annotation>
		</xsd:element>
		<xsd:element name="TableName" type="xsd:string">
			<xsd:annotation>
				<xsd:appinfo>The table to which the result data
				set originates</xsd:appinfo>
			</xsd:annotation>
		</xsd:element>
		<xsd:element name="RowNumber" type="xsd:string">
			<xsd:annotation>
				<xsd:appinfo>The row in which the data set can
				be found (please note: that this value is not
				numeric as the table's primary key may not be
				numeric</xsd:appinfo>
			</xsd:annotation>
		</xsd:element>
	</xsd:sequence>
```

```
        </xsd:complexType>

        <xsd:complexType name="ResultSetDef" mixed="false">
                <xsd:sequence>
                        <xsd:element name="DataSetNumber" type="xsd:integer">
                                <xsd:annotation>
                                        <xsd:appinfo>The orginal dataset to which the
                                        search results match</xsd:appinfo>
                                </xsd:annotation>
                        </xsd:element>
                        <xsd:element name="DataSetSearchResults"
                        type="fmf:SearchResultDef" minOccurs="0" maxOccurs="unbounded">
                                <xsd:annotation>
                                        <xsd:appinfo>All the results corresponding to
                                        the input dataset</xsd:appinfo>
                                </xsd:annotation>
                        </xsd:element>
                </xsd:sequence>
        </xsd:complexType>

        <xsd:complexType name="PerformSearchResponse" mixed="false">
                <xsd:sequence>
                        <xsd:element name="ResultSets" type="fmf:ResultSetDef"
                        maxOccurs="unbounded">
                                <xsd:annotation>
                                        <xsd:appinfo>Each set represents a set of
                                        results corresponding to the input
                                        set.</xsd:appinfo>
                                </xsd:annotation>
                        </xsd:element>
                </xsd:sequence>
        </xsd:complexType>

</xsd:schema>
```

Appendix F Outward Facing Pre-Filter Functionality Changes

Below is the definition of the interface used to integrate the custom third party pre-search filter logic with the framework.

```
    public class SearchFilter
    {
        protected string _coreSearchTableName;
        protected string _searchQuery;
        protected object[] _args;

        public string CoreSearchTableName
        {
            get { return _coreSearchTableName; }
            set { _coreSearchTableName = value; }
        }

        public string SearchQuery
        {
            get { return _searchQuery; }
            set { _searchQuery = value; }
        }

        public object[] Args
        {
            get { return _args; }
            set { _args = value; }
        }
    }

    public interface ISearchFilterInterface
    // Would prefer the parameters to be of type object but there is a problem with
the xsd:anyType type as it is not supported by JAX-RPC.
    // Therefore all arguments are to be parsed as strings
    {
        SearchFilter PreFilterSearch(params string[] Parameters);
    }
```

The InputSetDef structure (used as part of the search input, see Appendix C) has been altered to accept search parameters for the pre-search filter. Please note the inclusion of the PreSearchFilterArguments element.

```
<xsd:complexType name="InputSetDef" mixed="false">
	<xsd:sequence>
		<xsd:element name="WordSetComponents" type="fmf:InputDataSetWordDef"
		maxOccurs="unbounded">
			<xsd:annotation>
				<xsd:appinfo>
					This is a set of search words that are to be
					'anded' together in a search. Eg. Firstname and
					Surname
				</xsd:appinfo>
			</xsd:annotation>
		</xsd:element>
		<xsd:element name="PreSearchFilterArguments" type="xsd:string"
		minOccurs="0" maxOccurs="unbounded">
			<xsd:annotation>
				<xsd:appinfo>
					These are any arguments that need to be passed
					to the SearchFilter Method. Should be of type
					xsd:anyType instead of xsd:string but
					xsd:anyType is not supported by JAX-RPC
				</xsd:appinfo>
			</xsd:annotation>
		</xsd:element>
	</xsd:sequence>
	<xsd:attribute name="dummy" type="xsd:string" />
</xsd:complexType>
```

Appendix G Delegate Access to the Contents of the Rules Table

```
public class RulesTable
{
    int _ActionTypeID;
    string _ActionType;
    string _SearchPhrase;
    string _ReplacePhrase;
    double? _Penalty;

    public int ActionTypeID
    {
            get { return _ActionTypeID;}
            set { _ActionTypeID = value;}
    }

    public string ActionType
    {
            get { return _ActionType;}
            set { _ActionType = value;}
    }

    public string SearchPhrase
    {
            get { return _SearchPhrase;}
            set { _SearchPhrase = value;}
    }

    public string ReplacePhrase
    {
            get { return _ReplacePhrase;}
            set { _ReplacePhrase = value;}
    }

    public double? Penalty
    {
            get { return _Penalty;}
            set { _Penalty = value;}
    }

}
```

```
    public class FMFRules
    {
        private Dictionary<string, Dictionary<string,  List<RulesTable>>> _RulesTable;
        private LoadRuleTableDelegate _LoadRulesTableMethod;

        public delegate
       Dictionary<string, Dictionary<string, List<RulesTable>>>
LoadRuleTableDelegate();

        public const string RulesTable = "RulesTable";

        private void LoadRulesTable()
        {
            _RulesTable = _LoadRulesTableMethod();
        }

        public FMFRules(LoadRuleTableDelegate LoadRulesTableMethod)
        {
            _LoadRulesTableMethod = LoadRulesTableMethod;
            _RulesTable = new Dictionary<string, Dictionary<string,
List<RulesTable>>>();
            LoadRulesTable();
        }

        public Dictionary<string, List<RulesTable>> getRulesList(string ActionType)
        {
            if (String.IsNullOrEmpty(ActionType))
                throw new Exception("An ActionType must be supplied");

            if (!_RulesTable.ContainsKey(ActionType.ToUpper()))
               throw new Exception(String.Format(@"The ActionType ({0}) does not exist
               in                              the database", ActionType));

            return _RulesTable[ActionType.ToUpper()];
        }
    }
}
```

Appendix H Analysis of Test Data Set

1 Common First Name Prefixes

The following tables describe the top 50 unique first name entries that share the same prefix (i.e. share the same n number of initial characters). The length of the prefix increases with each subsequent table.

Table H.1: Prefix length of 1

Entry Count	Prefix
422	M
282	S
208	A
193	J
188	B
185	D
184	N
177	T
170	R
168	L
161	C
127	E
127	G
120	P
118	K
106	H
78	V
77	F
58	W
58	Z
51	I
32	O
19	Y

Table H.2: Prefix length of 2

Entry Count	Prefix
191	MA
72	TH
70	MO
65	JO
63	AN
60	SI
54	DA
54	JA
53	BE
52	RO
52	SA
51	RA
51	NO
49	LE
49	SH
48	AL
48	DE
45	CA
40	CH
39	HE
37	NA
36	EL
36	JE

13	U
6	Q
6	X

34	BR
34	HA
33	SE
33	KA
31	PH
31	PA
31	MI
31	LI
30	KE
30	CO
30	BA
29	FR
28	VI
28	DO
27	BO
27	LO
27	NI
27	LU
26	DI
26	ME
26	JU
26	ST
25	WI
25	TE
23	GE
23	RE
22	LA

Table H.3: Prefix length of 3

Entry Count	Prefix
55	MAR
32	THE
25	MAN
24	SHA
22	CAR
20	THA
20	MAT
19	SHE
19	NOM
18	FRA
18	JOS
18	WIL
17	CHR
17	BER
17	PHI
16	ANN
16	JAN
16	BON
15	DAR
15	AND
15	STE
14	DAN
14	NIC
14	JOH
14	HEN
13	JAC
13	GER
13	MAK
13	SAN
13	CHA
13	TER
12	LOR
12	MON

Table H.4: Prefix length of 4

Entry Count	Prefix
18	FRAN
17	MARI
17	CHRI
13	ANDR
13	THEM
11	BHEK
11	PHIL
10	JOSE
10	MAKH
10	KRIS
10	BONG
10	MAND
9	ANNE
9	WILL
9	JEAN
9	BERN
9	LIND
9	SHER
8	JULI
8	THEO
8	MICH
8	MART
8	CHAR
7	MOHA
7	THAN
7	TERR
7	MARC
7	NKOS
7	THAB
7	DUMI
7	DANI
7	JOHN
7	JOHA

11	DEL
11	LIN
11	PAT
11	BHE
11	ANT
11	CLA
11	MIC
11	THO
10	HAR
10	ROS
10	MAL
10	KRI
10	BEN
10	HER
10	ELI
10	SAL
9	ALE

7	CARO
7	ANTO
6	FRED
6	ANGE
6	PHUM
6	MATH
6	PATR
6	BERT
6	NICO
6	LAUR
6	THUL
6	DAVI
6	HEND
6	MATT
6	SIPH
5	CLAR
5	ALBE

Table H.5: Prefix length of 5

Entry Count	Prefix
17	CHRIS
13	THEMB
10	ANDRE
10	FRANC
8	KRIST
7	JOSEP
7	JOHAN
7	CAROL
7	DUMIS
6	NKOSI
6	PATRI
6	ANTON
6	THAND
6	DAVID
6	CHARL
6	ANGEL
6	MARIA
6	HENDR
5	BONGI
5	NICOL
5	FRANK
5	GEORG
5	JACQU
5	MAKHO
5	CECIL
5	WILLI
5	NICHO
5	ANNEL
5	ALBER
4	ROBER
4	MANDI
4	PHILL
4	THULA

Table H.6: Prefix length of 6

Entry Count	Prefix
16	CHRIST
7	JOSEPH
6	DUMISA
6	PATRIC
5	JOHANN
5	ANNELI
5	THEMBA
5	FRANCI
5	NICHOL
5	HENDRI
5	THEMBI
5	ALBERT
5	DAVID
4	ROBERT
4	THANDI
4	CONSTA
4	STEFAN
4	ERNEST
4	MICHAE
4	JACQUE
4	KRISTI
4	RICHAR
4	ANTONI
4	THEODO
3	GEORGE
3	THERES
3	PHILLI
3	MANDIS
3	VICTOR
3	MOHAME
3	MARIA
3	SIPHIW
3	JONATH

4	BERNA
4	MANDL
4	STEFA
4	DANIE
4	MICHA
4	ERNES
4	MATTH
4	STEPH
4	BHEKI
4	THABA
4	BRAND
4	THOKO
4	CONST
4	LOREN
4	THEOD
4	JUSTI
4	RICHA

3	ZACHAR
3	SIDNEY
3	SIBUSI
3	TRACEY
3	STEPHA
3	REGINA
3	BONGIN
3	MARGAR
3	NKOSIN
3	DANIEL
3	JEROME
3	ANDREW
3	COLLIN
3	HERMAN
3	SIBONG
3	ANGELI
3	MANDLA

Table H.7: Prefix length of 7

Entry Count	Prefix
7	CHRISTO
5	DUMISAN
5	CHRISTI
4	CONSTAN
4	MICHAEL
4	FRANCIS
4	THEODOR
4	RICHARD
3	JOHANNE
3	WILLIAM
3	MOHAMED
3	JOSEPH
3	BONGINK
3	NKOSINA
3	CORNELI
3	KRISTIN
3	SIPHIWE
3	STEPHAN
3	NICHOLA
3	SIBONGI
3	SIBUSIS
3	THULANI
3	PATRICK
3	THEMBA
3	JABULAN
3	THEMBIN
3	PHILLIP
2	MATTHEW
2	THAMSAN
2	COENRAA
2	JONATHA
2	ROSEMAR
2	VANESSA

Table H.8: Prefix length of 8

Entry Count	Prefix
4	DUMISANI
3	THEMBINK
3	BONGINKO
3	MICHAEL
3	SIBUSISO
3	CHRISTOP
3	RICHARD
2	LAWRENCE
2	JOHANNES
2	HERSCHEL
2	THAMSANQ
2	EVANGELI
2	JONATHAN
2	THEODORA
2	CHARMAIN
2	MOHAMED
2	MBUKENI
2	BHEKOKWA
2	COENRAAD
2	CHRISTOF
2	CONSTANT
2	MBONGENI
2	CHRISTIA
2	CHRISTIN
2	PHUMELEL
2	LEBOHANG
2	SIBONGIL
2	BHEKITHE
2	NKOSINAT
2	SIKHUMBU
2	WILLIAM
2	STHEMBIS
2	SIMPHIWE

2	EVANGEL
2	WELCOME
2	ABRAHAM
2	KENNETH
2	GILBERT
2	ANTHONY
2	SINDISW
2	MAXWELL
2	LEBOHAN
2	MLUNGIS
2	BHEKOKW
2	CHRISTA
2	CAROLYN
2	MAKHOSA
2	TIMOTHY
2	FRANCES
2	ANDREW

2	MKHANYIS
2	POOBALAN
2	ELIZABET
2	BERTRAND
2	PATRICK
2	JABULANI
2	SAMANTHA
2	SIPHIWE
2	FRANCISC
2	NICHOLAS
2	THULANI
2	ZACHARIA
2	MARGARET
2	JOSEPHUS
2	MARIMUTH
2	ANNA-MAR
2	ARUNAJAL

In addition, the variation between the first name prefixes and their corresponding names was determined. This was achieved through the calculation of the hamming distance between the two. Figure H.1 below displays nine of the most common 5 character prefixes and the corresponding first names to each of these prefixes.

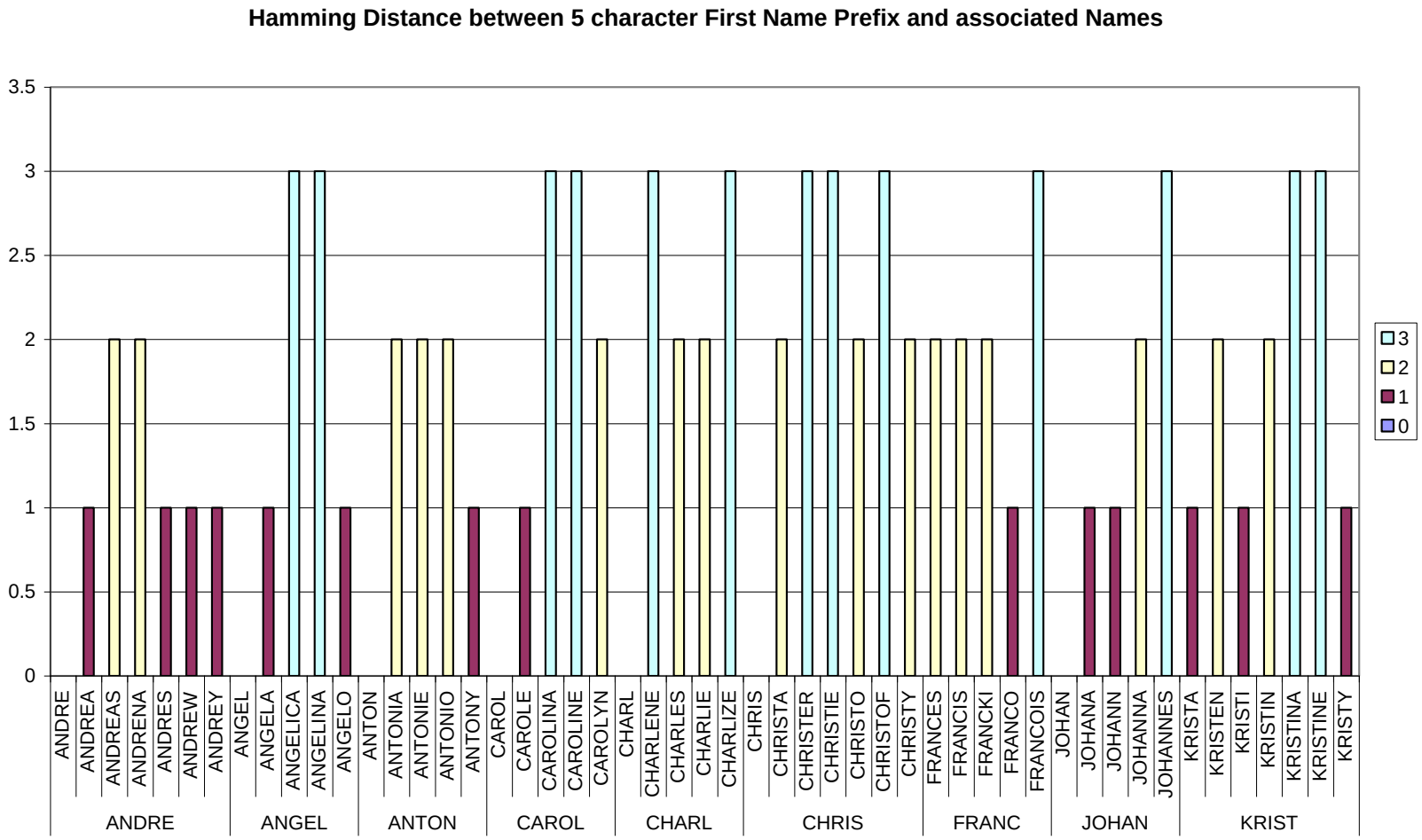


Figure H.1: The hamming distance between nine on the most common 5 character first name prefixes and their corresponding names

2 Common Surname Prefixes

The following tables describe the top 50 unique surname entries that share the same prefix (i.e. share the same n number of initial characters). The length of the prefix increases with each subsequent table.

Table H.9: Prefix length of 1

Entry Count	Prefix
904	M
532	S
435	B
382	C
338	H
301	D
284	R
281	G
261	L
247	K
243	P
230	N
206	W
177	T
165	F
159	V
153	A
95	J
90	E
68	O
29	Z
23	Y
19	I
12	U
12	Q

Table H.10: Prefix length of 2

Entry Count	Prefix
357	MA
189	MO
118	HA
111	CO
109	RA
92	BA
91	ST
89	CH
88	MC
85	VA
83	CA
80	BR
79	BO
79	HO
74	SH
74	DE
73	BE
73	LA
71	RO
70	HE
69	GO
68	SA
65	SE
65	PA
63	LE

9	X

60	DA
59	GR
57	GA
55	ME
53	WI
53	SC
53	WA
51	LO
48	SI
48	DO
48	KA
47	DU
47	MU
46	BU
44	KE
41	CR
41	JA
40	PE
39	DI
38	WE
38	RE
38	TH
37	LU
37	KO
36	FA

Table H.11: Prefix length of 3

Entry Count	Prefix
63	VAN
49	SCH
44	MAT
42	MAR
39	RAM
37	CHA
37	DE
37	MAS
34	CAR
31	HAR
29	MOR
28	MCC
26	STA
26	MAK
26	MAL
25	MAN
25	BAR
24	HER
23	MOK
23	MOT
22	BRA
22	MAD
22	SHA
22	WIL
22	SHE
21	COR
20	LAN
20	CHI
19	CON
19	BER
19	CRO
19	STE
19	MOL

Table H.12: Prefix length of 4

Entry Count	Prefix
58	VAN
15	MATH
14	WHIT
14	MASH
13	GOLD
12	RAMA
12	WOOD
11	SCHO
11	GREE
10	ABRA
10	MATS
10	VON
10	MOLO
10	WILL
10	LANG
10	MADI
9	GOOD
9	MOKG
9	DAVI
9	CONN
8	MAGA
8	MCCA
8	MABU
8	MAKH
8	HOLL
8	FRAN
8	MOTS
7	MCCO
7	SCHA
7	CHAN
7	MONT
7	MILL
7	MCCL

19	GRE
19	HAN
19	STO
19	BUR
18	MAC
18	HOL
18	MAG
18	MAB
18	BEN
17	GAR
17	SAN
17	MAH
16	BLA
16	MOS
16	PAR
16	LIN
16	STR

7	SCHU
7	CHRI
7	GILL
7	CHAM
7	DICK
7	LAND
7	HERR
7	HEND
7	RICH
7	JACO
7	MAHL
6	VILL
6	RAMO
6	MASE
6	BARR
6	SHER
6	MAYE

Table H.13: Prefix length of 5

Entry Count	Prefix
31	VAN D
8	GREEN
7	CHRIS
7	MASHI
7	JACOB
6	HENDR
6	MATHE
6	BLACK
5	MARTI
5	MAKHA
5	MAHLA
5	MOKGA
5	ABRAH
5	WILLI
5	DAVID
5	VALEN
4	MANDE
4	VILLA
4	PICKE
4	SCHLE
4	LANGE
4	CONNO
4	MOTLA
4	JOHNS
4	NICHO
4	ABRAM
4	THORN
4	WILKE
4	VAN W
4	JANSE
4	LEVIN
4	MOKGO
4	FRANK

Table H.14: Prefix length of 6

Entry Count	Prefix
26	VAN DE
7	CHRIST
5	JACOBS
5	HENDRI
5	MARTIN
5	ABRAHA
4	SCHWAR
4	MANDEL
4	NICHOL
4	WILLIA
3	HUTCHI
3	ALBERT
3	PETERS
3	CONNEL
3	EDMOND
3	GOLDST
3	RICHAR
3	JANUAR
3	VALENT
3	LAWSON
3	PARSON
3	CUMMIN
3	MADIKI
3	VAN WY
3	GRIFFI
3	DLAMIN
3	DANIEL
3	PIETER
3	MADLAL
3	MOKGAT
3	JANSEN
3	WASSER
3	SCHOON

4	SCHWA
4	GOLDS
4	HOLLI
4	WHITE
4	CONNE
3	STEIN
3	ROBER
3	WHITT
3	WOODS
3	BRICK
3	MATSE
3	RICHA
3	VAN B
3	SCHOL
3	MODIS
3	HARRI
3	ALBER

2	MORGAN
2	BLACKM
2	JOSEPH
2	ROBERT
2	WATERS
2	BUTLER
2	DENNIS
2	GOUNDE
2	SPENCE
2	FRIEDM
2	MUELLE
2	HIRSCH
2	SCHREU
2	MASHEG
2	MOHLAL
2	MCCULL
2	MCCLEL

Table H.15: Prefix length of 7

Entry Count	Prefix
19	VAN DER
5	ABRAHAM
4	WILLIAM
4	CHRISTI
4	VAN DEN
3	MADLALA
3	HENDRIC
3	PIETERS
3	DLAMINI
3	CONNELL
3	RICHARD
3	SCHWART
2	BURNETT
2	MUELLER
2	MAYENDE
2	HAVENGA
2	HANKINS
2	LAMBERT
2	SCHREUD
2	VAN DE
2	MCCLELL
2	HOFFMAN
2	METCALF
2	VORSTER
2	SCHULTZ
2	GILBERT
2	CUMMING
2	MOTAUNG
2	HOLLAND
2	PARSONS
2	SANGWEN
2	KNOTT-C
2	MALULEK

Table H.16: Prefix length of 8

Entry Count	Prefix
19	VAN DER
4	VAN DEN
3	WILLIAMS
3	PIETERSE
3	CHRISTIA
3	SCHWARTZ
3	HENDRICK
3	ABRAHAMS
2	NTOMBELA
2	PADAYACH
2	HUTCHINS
2	GRIFFITH
2	TSHABALA
2	RICHARDS
2	JANSEN V
2	GELDENHU
2	RODRIGUE
2	STEPHENS
2	FERNANDE
2	ESTERHUY
2	KNOTT-CR
2	JANUARY-
2	DA SILVA
2	VAN WYNG
2	VALENTIN
2	MADLALA-
2	PALMBOOM
2	DRUMMOND
2	SCHREUDE
2	JOHNSTON
2	CILLIERS
2	SANGWENI
2	NKABINDE

2	MCCLAIN
2	JANSEN
2	SUBRAMO
2	MOKGATL
2	STEVENS
2	NTOMBEL
2	WASSERM
2	PRESTON
2	PADAYAC
2	HUTCHIN
2	FRIEDMA
2	COCHRAN
2	STEPHEN
2	FREDERI
2	GONZALE
2	EDMONDS
2	HERRING

2	FREDERIC
2	FRIEDMAN
2	SUBRAMON
2	MCCLELLA

In addition, the variation between the surname prefixes and their corresponding names was determined. This was achieved through the calculation of the hamming distance between the two. Figure H.2 below displays six of the most common 5 character prefixes and the corresponding surnames to each of these prefixes.

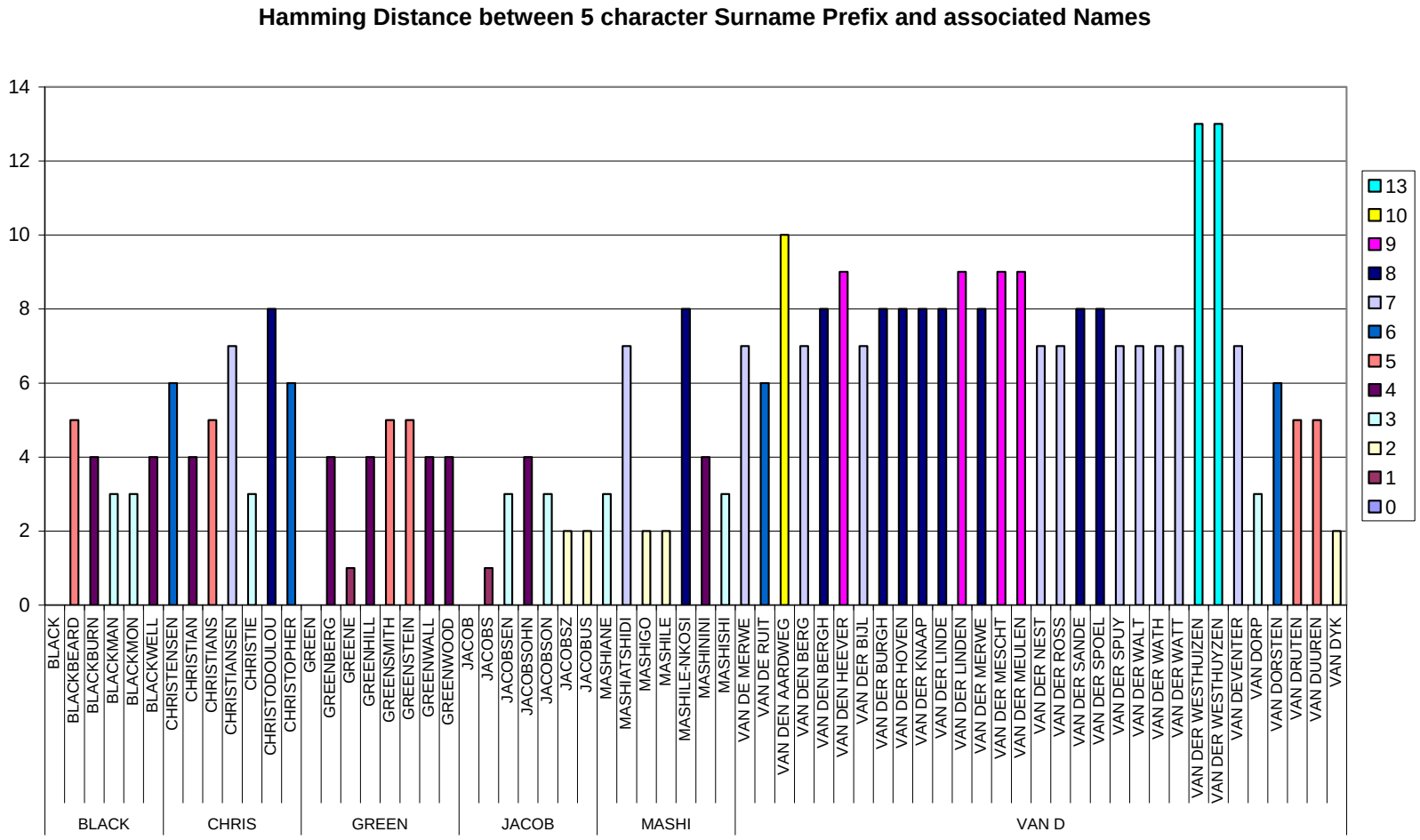


Figure H.2: The hamming distance between nine on the most common 5 character surname prefixes and their corresponding names

3 Phonetic Properties of Test Data Set

In order to develop a better high level understanding of the phonetic properties of the test data set, the Soundex code for each of the entries was determined. It must be noted that this is a crude test as Soundex is one of the simpler phonetic algorithms. As it is not viable to display all the results (owing to size of the data set), several of the most commonly occurring Soundex codes, within the test data set, were selected. These codes, their corresponding entries and the number of occurrences of these entries are displayed in Figure H.3 and Figure H.4.

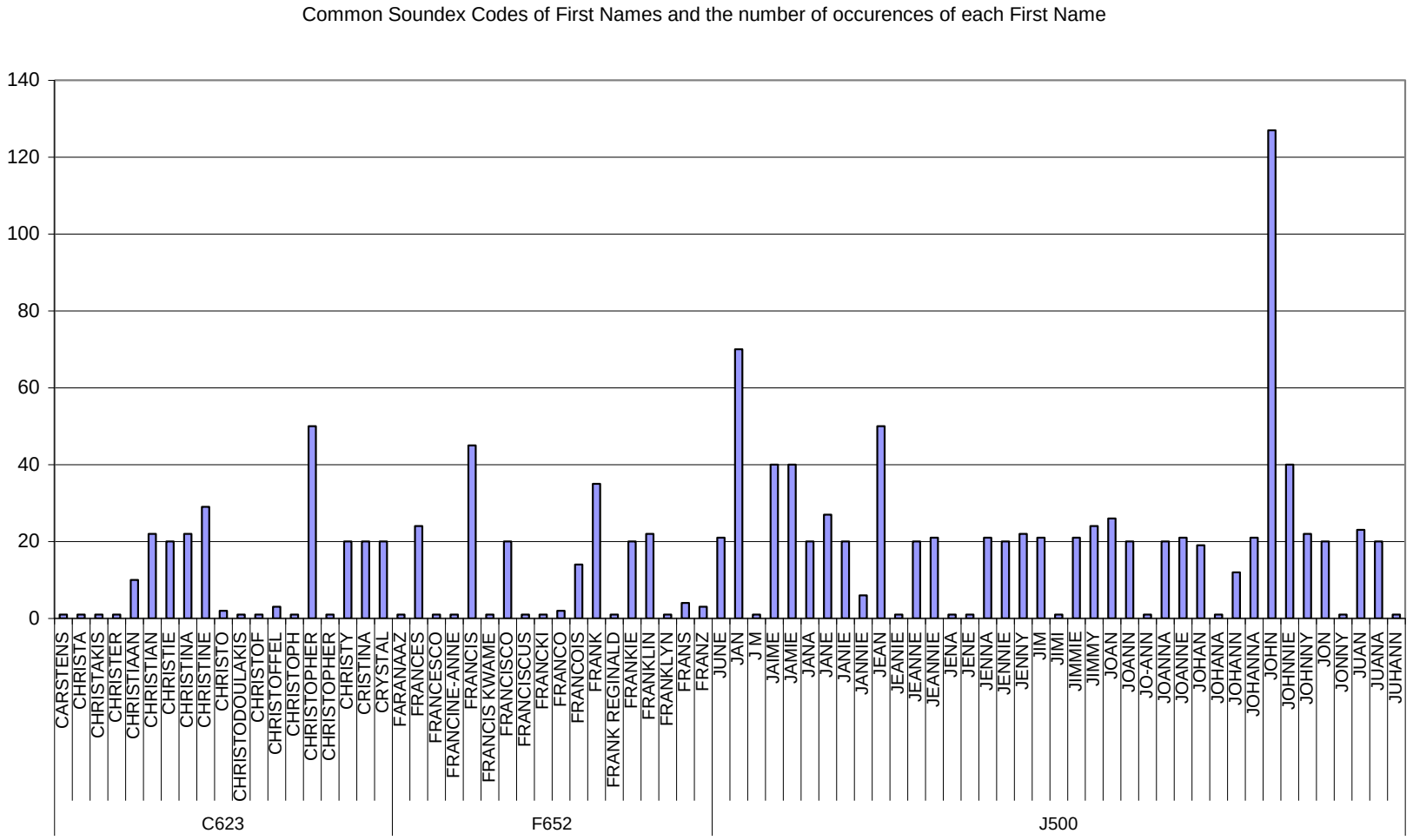


Figure H.3: Three of the most commonly occurring Soundex codes, the corresponding first names and the number of occurrences of each first name

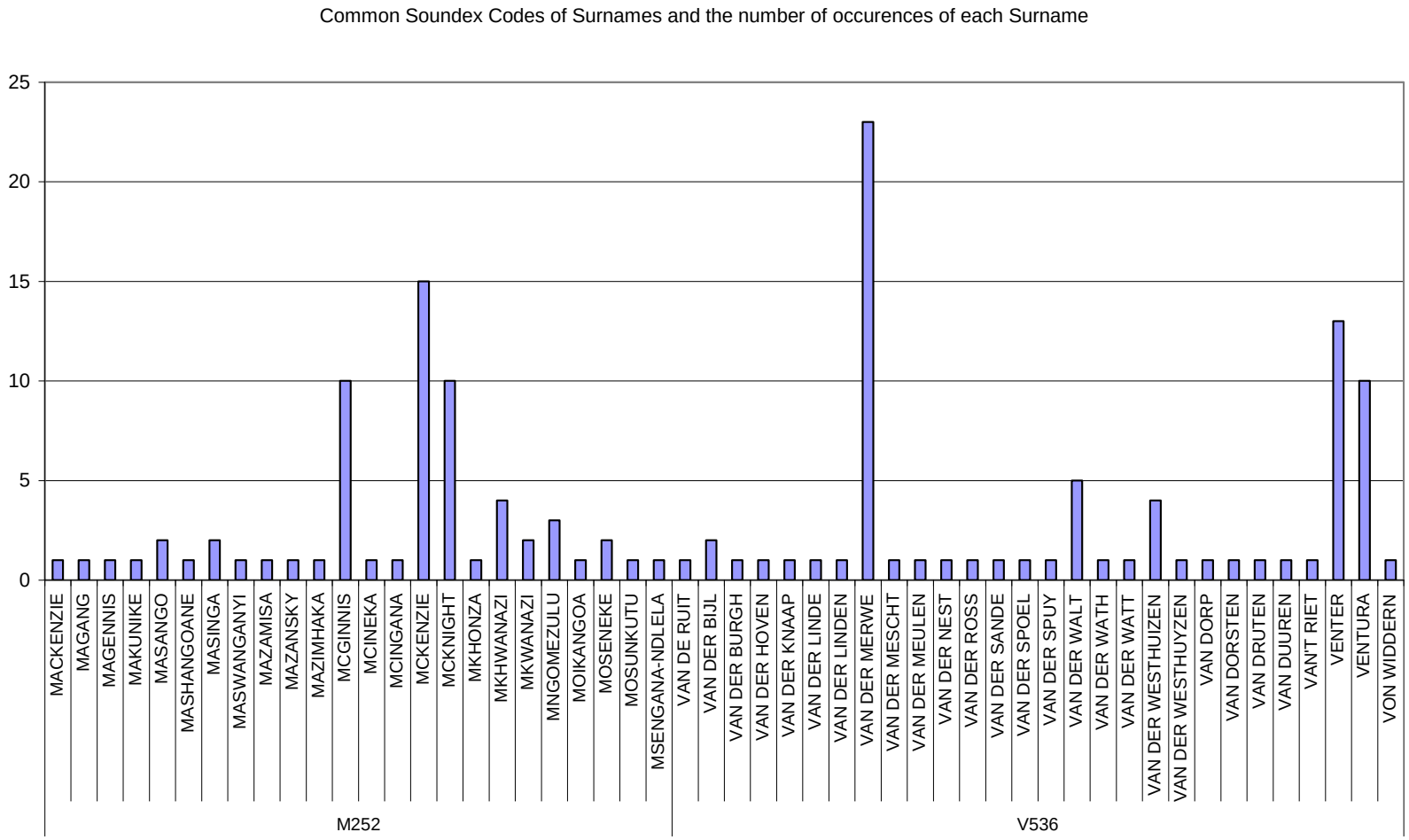


Figure H.4: Two of the most commonly occurring Soundex codes, the corresponding surnames and the number of occurrences of each first name