

Multilingual Information Processing on Relational Database Architectures

A THESIS
SUBMITTED FOR THE DEGREE OF
Doctor of Philosophy
IN THE FACULTY OF ENGINEERING

by

A. KUMARAN



DEPARTMENT OF COMPUTER SCIENCE AND AUTOMATION
INDIAN INSTITUTE OF SCIENCE
BANGALORE 560 012 INDIA

December 2005

©A. Kumaran
December 2005
All rights reserved

॥ श्री ॥

गुरुर्ब्रह्मा गुरुर्विष्णुः गुरुर्देवो महेश्वरः ।
गुरुः साक्षात्परब्रह्म तस्मै श्री गुरवे नमः ॥ †

DEDICATED
TO
ALL MY TEACHERS

அறிவாய் அறியாமை நீங்கியவனே
பொறிவாய் ஒழிந்தெங்குந் தானுன போதன்
அறிவா யவற்றினுட்டானு யறிவன்
செறிவாகி நின்ற அச்சேவனுமாகுமே. ‡

† The forces of creation, sustenance and destruction – the material causes of the universe – are the teachers. Through them, I offer my salutations to the ultimate teacher – the Supreme Being. *Upanishat.*

‡ He is Knowledge – complete and pure; He is the Knower – omniscient and self-illuminating. Yet, due to *Māya*, He manifests Himself as the Teacher and as the Student. He is all-pervading. *Thirumandiram.*

Acknowledgements

First and foremost, I thank my doctoral thesis advisor, Prof. Jayant R Haritsa, who taught me not just database management systems, but also fundamental values, such as, committment, sincerity and hardwork. The latter, I believe, is as beneficial, if not more, as the former, in professional and personal life. He continues to demonstrate that by demanding quality, one can achieve it not only within oneself, but also in those around.

I wish to express my deep gratitude to Prof. Y N Srikant, Prof. M Narasimhamurty, Prof. Priti Shankar, Prof. Matthew Jacobs and Prof. Y Narahari of Computer Science Department and Prof. N Balakrishnan, Associate Director of IISc, all of whom have offered regular guidance and assured their faith in me in various ways. I thank them all.

I thank my colleagues, Srikanta, Maya, Suresha, Vikram, Amit, Bharat and Aditya, for endless hours in the Coffee Board and the Tea Board, discussing academic and other issues. I also thank other students that contributed to technical content of this thesis – Nithya, Sivaramakrishnan, Manjunath, Sandhya, Girish, Rupesh, Mitesh and Pavan.

The Institute provided a wonderful environment not just for academics and research, but also for philosophical musings (long walks & NIAS lectures). I would always miss it.

In my personal life, I am indepted to my parents, Sri. Arumugam and Smt. Chandra, who had inculcated the love of and discipline for learning, when I was young.

Most important of all, I thank my wife, Vijayalakshmi, for her whole-hearted and graceful support through the years of my doctoral studies, while managing our family with two young daughters. My doctoral studies and this thesis were possible, solely due to her encouragement and complete support.

Publications based on this Thesis

REFEREED PAPERS

1. On Database Support for Multilingual Environments

Proceedings of the 13th IEEE Research Issues in Data Engineering (RIDE/ICDE) Workshop, held in conjunction with *19th IEEE International Conference on Data Engineering* [pgs. 23-30], Bangalore/Hyderabad, India, *March 2003*.

2. On the Costs of Multilingualism in Database Systems

Proceedings of the 29th International Conference on Very Large Data Bases (VLDB) [pgs. 105-116], Berlin, Germany, *September 2003*.

3. LexEQUAL: Supporting Multilexical Queries in SQL (poster)

Proceedings of the 20th IEEE International Conference on Data Engineering (ICDE) [pg. 845], Boston, United States, *March 2004*.

4. LexEQUAL: Supporting Multiscript Matching in Database Systems

Proceedings of the 9th International Conference on Extending Database Technology (EDBT), published as *Advances in Database Technology - EDBT 2004*, Springer, *Lecture Notes in Computer Science (LNCS) Vol. 2992*, eds. *E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Bohm and E. Ferrari* [pgs. 292-309], Heraklion-Crete, Greece, *March 2004*.

5. LexEQUAL: Multilexical Matching Operator in SQL (software demo)

Proceedings of the 23rd ACM SIGMOD International Conference on Management of Data (SIGMOD) [pgs. 949-950], Paris, France, *June 2004*.

6. MIRA: Multilingual Information-processing on Relational Architectures

Proceedings of the EDBT/ICDE 2004 Workshops, published as *Current Trends in Database Technology: Revised Selected Papers*, Springer, *Lecture Notes in Computer Science (LNCS) Vol. 3268*, eds. W. Lindner, M. Mesiti, C. Türker, Y. Tzitzikas, A. Vakali, Vol-3268 [pgs. 12-23], November 2004.

7. On Semantic Multilingual Matching in Relational Systems (poster)

Proceedings of the 13th ACM International Conference on Information and Knowledge Management (CIKM) [pgs. 230-231], Washington DC, United States, November 2004.

8. SemEQUAL: Multilingual Semantic Matching in Relational Systems

Proceedings of the 10th International Conference on Database Systems and Advanced Applications (DASFAA) [pgs. 214-225], Beijing, China, April 2005.

9. On Pushing Multilingual Query Operators inside Relational Engines

(To appear) Proceedings of the 22nd IEEE International Conference on Data Engineering (ICDE), Atlanta, United States, April 2006.

TECHNICAL REPORTS

1. Supporting Multilexical Matching in Database Systems

Technical Report TR-2004-01, DSL/SERC, Indian Institute of Science, 2004.

2. Multilingual Semantic Matching Operator in SQL

Technical Report TR-2004-03, DSL/SERC, Indian Institute of Science, 2004.

3. On Pushing Multilingual Query Operators inside Relational Engines

Technical Report TR-2005-01, DSL/SERC, Indian Institute of Science, 2005.

Abstract

Efficient storage and query processing of data spanning multiple natural languages are of crucial importance in today’s globalized world. A primary prerequisite to achieve this goal is that the principal data repositories, relational database systems, should efficiently and seamlessly support multilingual data. Our survey of current relational systems indicates that while they do support storage and management of multilingual data, querying is restricted to be within a given language, with no crosslingual query support. Further, quantitative performance study of the systems working on different character sets has not been published so far and therefore is an open issue.

In this thesis, we first profile the multilingual performance of a set of current relational database systems, using an environment based on the TPC benchmark suites. The results indicate a significant performance degradation while handling multilingual data. While the differential performance is huge when disk traffic is a factor, it is substantial even when only *in-memory* processing is considered. To address this inequity, we propose a split representation format that reduces the multilingual storage space and largely eliminates the differential performance for most languages except those with unusually large repertoires.

Next, we propose functionality enhancements that complement the standard lexicographic matching, specifically in the multilingual text space. Two new multilingual join operators – one for joining names across languages and the second for joining multilingual categories based on their meanings – are proposed and formally defined. These operators are implemented in an *outside-the-server* approach using existing SQL features of relational systems, and using standard linguistic resources. While the performance

of these basic implementations is too slow for real-world deployments, a host of optimization techniques that tune the schema and index choices to match typical linguistic features are employed and shown to improve the performance to a level sufficient for practical use.

Finally, for a full integration of multilingual functionality with the database engine, we specify a query algebra with a new multilingual storage datatype and the above join operators. The operators are implemented *natively* as first-class features in an open-source database system, along with all components that are required to leverage the relational query optimizer, specifically, the operator cost models and their selectivities. The performance experiments indicate that this native implementation of the multilingual operators improves the performance significantly over the outside-the-server implementation. Further, the power of the algebra is demonstrated through selection of better execution plans for queries using the multilingual operators.

In summary, this thesis presents a multilingual query processing architecture, with a set of functionalities, algorithms, implementation and optimization techniques, all geared towards the goal of developing *natural-language-neutral* database engines.

Keywords

Multiscript Text Database Systems

Multilingual Names / Semantic Matching

Homophonic / Homosemic Query Processing

Database Performance

Multilingual Query Algebra

Multilingual Query Processing Architecture

Multilingual Information Retrieval

Contents

Acknowledgements	ii
Publications based on this Thesis	iii
Abstract	v
Keywords	vii
Notation and Abbreviations	xv
1 Introduction	1
1.1 Motivation for Multilingualism	1
1.2 Existing Support for Multilingual Data	2
1.2.1 Multilingual Specifications in SQL Standard	2
1.2.2 Multilingual Support in Database Systems	2
1.2.3 Multilingualism in Database Research	5
1.3 A Sample Multilingual Application: <i>Books.com</i>	6
1.4 Research Issues Explored	7
1.4.1 Multilingual Performance	7
1.4.2 Multilingual Names Matching	8
1.4.3 Multilingual Semantic Matching	10
1.4.4 Complex Multilingual Operations	12
1.5 Proposed Solution Strategies	14
1.5.1 Design Goals for Our Research Strategy	14
1.5.2 Multilingual Performance	15
1.5.3 Multilingual Matching Functionalities	16
1.5.4 Multilingual Query Processing Architecture	22
1.5.5 Applicability in Other Domains	23
1.6 Real-life Multilingual Systems	23
1.7 Organization of this Thesis	27
2 Multilingual Performance of Current Systems	30
2.1 Overview of the Chapter	30
2.2 Setup for Multilingual Performance Study	30
2.2.1 System and Database Environment	31

2.2.2	Dataset	31
2.2.3	Query Workload	33
2.2.4	Performance Metrics	34
2.3	Performance Results	36
2.3.1	Space Overheads	36
2.3.2	Separate Table Processing	37
2.3.3	Common Table Processing	37
2.3.4	Optimizer Prediction Accuracy	41
2.4	Performance Analysis	42
2.4.1	Slowdown <i>vis-a-vis</i> String Length	42
2.4.2	Components of the Slowdown	43
2.5	The Cuniform Storage Format	46
2.5.1	Sample Unicode and Cuniform Strings	47
2.5.2	Limitations of Cuniform Format	48
2.6	Cuniform Performance	49
2.6.1	Performance of Cuniform Storage	50
2.6.2	Potential for Further Performance Improvement	52
2.7	Related Research	52
2.8	Conclusions on Multilingual Performance Study	53
3	Multilingual Names Matching	55
3.1	Overview of the Chapter	55
3.2	Background Information	55
3.2.1	Pseudo-Phonetic Matching Function	55
3.2.2	Approximate Matching	57
3.2.3	Q-Grams	57
3.3	Multilingual Names Matching Implementation	58
3.3.1	MLNameJoin Implementation Details	59
3.3.2	Linguistic Issues	60
3.3.3	Existing Database Support for Implementation	61
3.4	MLNameJoin Matching Algorithm	63
3.5	Access Structures for MLNameJoin	66
3.5.1	B+ Tree Index	66
3.5.2	Metric Distance Index	66
3.5.3	Approximate Index Structures	70
3.6	Multilingual Names Matching Quality	72
3.6.1	Dataset	72
3.6.2	Performance Metrics	74
3.6.3	Multilingual Names Matching Quality	75
3.7	MLNameJoin Performance	77
3.7.1	System and Database Environment	77
3.7.2	Dataset	78
3.7.3	Baseline MLNameJoin Performance	79
3.7.4	Optimization #1: Q-Gram Index	81

3.7.5	Optimization #2: Phonemic Index	83
3.8	Related Research	85
3.9	Conclusions on Multilingual Names Matching	87
4	Multilingual Semantic Matching	89
4.1	Overview of the Chapter	89
4.2	Background Information	89
4.2.1	WordNet: A Linguistic Resource	89
4.3	Multilingual Semantic Matching Implementation	92
4.3.1	MLSemJoin Implementation Details	92
4.4	MLSemJoin Matching Algorithm	96
4.4.1	Derived Operator Approach	97
4.4.2	Following Through with an Example	97
4.5	MLSemJoin Performance	99
4.5.1	System and Database Environment	99
4.5.2	Dataset	99
4.5.3	Query Workload	100
4.5.4	Performance Metrics	101
4.6	Performance Results and Analysis	102
4.6.1	Baseline MLSemJoin Performance	102
4.6.2	Optimization #1: Precomputed Closure	104
4.6.3	Optimization #2: Reversed Traversal	105
4.6.4	Optimization #3: Reorganizing Schema	106
4.6.5	MLSemJoin Performance with Scaling of Languages	107
4.7	Related Research	109
4.8	Conclusions on Multilingual Semantic Matching	111
5	A Multilingual Operator Algebra	113
5.1	Overview of the Chapter	113
5.2	Mural: Multilingual Relational Algebra	113
5.2.1	Uniform: A Multilingual Text Datatype	114
5.2.2	Uniform Equality (Ξ) Operator	116
5.2.3	Uniform Names Matching (Ψ) Operator	116
5.2.4	Uniform Semantic Matching (Φ) Operator	118
5.3	Interaction Between Mural Operators	120
5.4	Relational Completeness of Mural	121
5.5	Mural Query Optimization Strategies	123
5.5.1	Cost-based Optimization Strategies	123
5.5.2	Rule-based Optimization Strategies	126
5.6	Related Research	127

6	A Native Implementation Experience	128
6.1	Overview of the Chapter	128
6.2	Implementation Methodologies	128
6.3	A <i>Native</i> Implementation Experience	130
6.3.1	System Environment	130
6.3.2	Native Ψ Operator Implementation	130
6.3.3	Native $\Phi_{\mathcal{H}}$ Operator Implementation	131
6.4	Performance of <i>Native</i> Implementation	132
6.4.1	Performance of Ψ Implementation	133
6.4.2	Performance of $\Phi_{\mathcal{H}}$ Implementation	134
6.5	Optimizer Prediction Performance	135
6.5.1	A Motivating Optimization Example	136
6.6	A Prototype Demonstration	138
6.7	Conclusions on <i>Native</i> Implementation	139
7	Conclusions and Future Research Avenues	140
7.1	Conclusions	140
7.1.1	Practical Solutions from the Thesis	142
7.2	Future Research Avenues	144
A	Character Encoding Standards	146
A.1	Unicode	146
B	Phonology and Phonemes Encoding Standards	148
B.1	Phonology and Phonemes	148
B.2	International Phonetic Alphabet	149
	Bibliography	150

List of Tables

1.1	Database Systems <i>vis-a-vis</i> Multilingual Support	3
2.1	Multilingual Performance of Operators	39
2.2	Multilingual Efficiency	40
2.3	Multilingual Performance of Operators on Cuniform	51
3.1	MLNameJoin Operator Performance	79
3.2	MLNameJoin Operator Baseline Performance	80
3.3	MLNameJoin Performance with Q-Gram Index	82
3.4	MLNameJoin Performance with Phonemic Index	84
4.1	Statistical Profile of WordNets	100
4.2	Closures for English Word Forms	101
5.1	Mural Operator Composition Rules	120
5.2	Symbols used in Mural Operator Cost Models	124
5.3	Mural Operator Cost Models	125
6.1	Performance of Ψ Operator	133

List of Figures

1.1	Hypothetical Multilingual <i>Books.com</i> Catalog	7
1.2	SQL:1999 Compliant Multilingual Names Query and Result Set	8
1.3	A Multilingual Names Query	9
1.4	SQL:1999 Compliant Multilingual Semantic Query and Result Set	10
1.5	A Multilingual Equivalent Semantic Query and Result Set	11
1.6	A Multilingual Generalized Semantic Query and Result Set	12
1.7	Multilingual Publisher Table	13
1.8	Sample Multilingual Complex Query – 1	13
1.9	Sample Multilingual Complex Query – 2	14
1.10	Sample Multilingual Complex Query – 3	14
1.11	Ontology for Text Data	17
1.12	Ontology for Names Matching of Text Data	19
1.13	Ontology for Semantic Matching of Text Data	21
1.14	aAqua: An Indic Multilingual Agricultural Portal	26
1.15	Euroseek: A Pan-European Multilingual Search Engine	28
2.1	Data Setup for Performance Study	33
2.2	Query Slowdown with String Size	43
2.3	Query Slowdown with Scaling	45
2.4	Skimming of Unicode Strings	48
3.1	<i>Soundex</i> Algorithm	56
3.2	The MLNameJoin Matching Algorithm	64
3.3	Search Efficiency of Approximate Indexes	71
3.4	Phonemic Representation of Test Data	73
3.5	Distribution of Multiscript Dataset	74
3.6	MLNameJoin Operator Recall and Precision	75
3.7	MLNameJoin Combined Precision-Recall Graphs	77
3.8	Distribution of Generated Multiscript Data Set	79
3.9	MLNameJoin SQL Script with Q-Gram Index	82
3.10	MLNameJoin SQL Script with Phonemic Indexes	84
4.1	Sample Inter-linked <i>WordNet</i> Noun Taxonomic Hierarchy	91
4.2	The MLSemJoin Matching Algorithm	95
4.3	Baseline Performance of Computing Closure	103

4.4	Closure Performance with Precomputed Closures	104
4.5	Closure Performance with Reversed Traversal	106
4.6	Fan-out Histogram and Plot	107
4.7	Closure Performance with Re-Organized Schema	108
4.8	Closure Performance with Number of Languages	109
5.1	The Ξ Operator	116
5.2	The Ψ Operator	117
5.3	The Φ Operator	119
6.1	Postgres Closure Performance	134
6.2	Optimizer Prediction Performance	135
6.3	Alternate Query Plans for Example 6.1	137
6.4	A Prototype Implementation	138
A.1	Sample Encoding in Various Formats	147

Notation and Abbreviations

Abbreviation	Stands for
ASCII	<i>American Standard Code for Information Interchange</i>
Char/NChar	<i>Character / National Character</i>
CJK	<i>Chinese, Japanese and Korean</i>
Cuniform	<i>Compressed UNICode FORMat</i>
GiST	<i>Generalized Index Structure Tree</i>
ILI	<i>Inter-Lingual Index</i>
IPA	<i>International Phonetic Association</i>
IR	<i>Information Retrieval</i>
ISO	<i>International Standards Organization</i>
LHS/RHS	<i>Left Hand Side / Right Hand Side</i>
MIRA	<i>Multilingual Information-processing on Relational Architecture</i>
MURAL	<i>MUltilingual Relational ALgebra</i>
NLP	<i>Natural Language Processing</i>
OLTP	<i>On-Line Transaction Processing</i>
SQL	<i>Structured Query Language</i>
TTS/TTP	<i>Text-to-Speech / Text-to-Phoneme Systems</i>
UCS	<i>Universal Character Set</i>
UDF	<i>User Defined Function</i>
Uniform	<i>UNICode FORMat</i>
UNL	<i>Universal Networking Language</i>
UTF	<i>Unicode Transfer Format</i>
WN	<i>WordNet (lexical resource)</i>

Chapter 1

Introduction

1.1 Motivation for Multilingualism

Efficient storage and query processing of data spanning multiple natural languages are of crucial importance in today's globalized world. A case in point is the changing user and data demographics of the highly popular Internet, which has become the primary medium for information access and commerce¹. Surveys indicate that the demographics of the Internet are steadily turning multilingual: the fraction of Internet users that are non-native English speakers has grown from about *half* in mid-90's, to about *two-thirds* now [24] and it is predicted that the majority of information available in the Internet will be multilingual by 2010 [132]. The changing demographics will affect the way in which the Internet-based *e-Commerce* or *e-Governance* systems are to be deployed: It has been found that a user is likely to stay *twice* as long at a site and *four-times* more likely to buy a product or consume a service, if the information is presented in their native language [2]. Hence, it is imperative that the information systems support storage and management of multilingual data, efficiently and effectively. A primary prerequisite to achieve this goal is that the principal data repositories – relational database management systems – should natively support multilingual information.

¹The size of Internet user population is ≈ 1 B and generates an economic activity of ≈ 1 T US\$ [48].

1.2 Existing Support for Multilingual Data

All commercial and open-source relational database systems profess support for multilingualism. To baseline the multilingual support currently available, we first present a survey of the multilingual support specification in the SQL standard and that offered by a suite of relational database systems².

1.2.1 Multilingual Specifications in SQL Standard

SQL-92 [83] was the first standard that specified SQL features for multilingual support, and the current SQL:1999 Standard [59, 84] has largely left it unmodified. SQL Standard specifies a new datatype – National Char (referred to as NChar) – large enough to store characters from any language or script. However, the NChar datatype is not a core requirement in SQL:1999 and hence is not supported uniformly even by SQL:1999 compliant database systems. The NChar datatype may be defined and manipulated similar to the normal character datatype and may be used in all character predicates. Further, the storage format of NChar is left unspecified in the SQL standard, leading to different support formats between different database vendors. For the support of a language, specification of its *collation* – the sort order of the characters in that language – is needed; SQL standard allows specification of collations dynamically. SQL standard also specifies that new repertoires may be defined and that a column may be restricted to hold only characters from such a specific repertoire. Finally, the standard specifies that comparison and sorting of strings to be meaningful only *within* a repertoire.

1.2.2 Multilingual Support in Database Systems

Table 1.1 provides a comparison of the multilingual features supported by a suite of commercial database systems, namely Oracle 9i Database Server (Version 9.0.1), IBM DB2 Universal Server (Version 7.1.0) and Microsoft SQL Server (Version 8.00.194) and

²Knowledge of multilingual character encodings standards is assumed as a background for this thesis. However, a brief overview is provided in Appendix A.

the popular open-source database systems, namely, My SQL (Version *4.0.3 Beta*) and PostgreSQL (Version *8.0.1-Beta*) database servers³. The information provided in this comparison is gathered from white papers, product literature and other information published in their respective web-sites [56, 86, 94, 100, 103].

Database	Oracle 9i Server	Microsoft SQL Server	IBM DB2 Univ. Server	MySQL	PostgreSQL
Storage Format	Unicode UTF-8 / 16	UCS-2 UTF-8	Unicode UTF-8 / 16	Binary	Unicode UTF-8
Support Level	At Attribute level	At Attribute level and also Schema Objects	At Attribute level	At Attribute level	At Attribute level
Collation Sequence	Pre-defined	Pre-defined OS Collations	Pre-defined	Pre-defined; User-definable (source-level)	Pre-defined; User-definable (source-level)
Indexing	Using only predefined Collations	Using only predefined Collations	Using only predefined Collations	Using predefined and User-defined Collations	Using predefined and User-defined Collations
Locale	≈ 50 Locales pre-specified	Uses all Locales specified in OS	≈ 40 Locales pre-specified	≈ 23 Locales pre-specified	≈ 30 Locales pre-specified
Query Predicate	All Char predicates	All Char predicates	All Char predicates	Binary predicates	All Char predicates
Cross-Lingual Queries	No Support	No Support	No Support	No Support	No Support

Table 1.1: **Database Systems *vis-a-vis* Multilingual Support**

While all these database systems except for MySQL Server, support multilingual storage using either Unicode or UCS-2, the MySQL Server supports storage only using *binary* datatype. However, Unicode is in the road-map of all vendors, as the prime candidate for multilingual storage. All systems support multilingual specification at all levels

³Locale is the subset of user's computing environment that defines the language for user input/output, the cultural and national conventions for formatting time, date, numeric and monetary data. Collation sequence specifies a sort order for the strings in a given language environment. This sort order may be different from the standard binary sort order of the character codes; in addition, the same character codes may sort differently in different language environments.

– schema, table, record and attribute, while Microsoft SQL Server provides multilingual support for database catalogs as well. No system has support for restricting the data in a column to be from a single repertoire, though specified in SQL:1999. All the systems pre-define *collations* that are needed for sorting the data for output and for building internal indexes. Though the SQL standard specifies user-defined collations, none of the systems have implemented this feature at this time. User defined collations may be added only to the MySQL and PostgreSQL systems, with source changes. Support for linguistic querying of text data is available in commercial database systems, but the techniques used are not uniform among the systems, due to the lack of specified guidelines in the SQL standard. A variety of statistical and natural language processing techniques are employed, resulting in, for a given query on a given data-set, a non-uniform result sets among different systems. Further, such querying is done only within a single language and the capability is available only in a handful of languages.

Multilingual query processing is supported along the same lines as that for standard database character sets, using *Char* predicates, in all database systems. However, the use of *Char* predicates implies the use of lexicographic comparisons for comparing multilingual text strings – even for strings that are from different languages. Such a methodology is bound to fail when the multilingual text strings do not share the same script; the equality will always fail and the sort order between two multilingual text strings in different scripts would depend only on the placement of the script in the *Unicode* codespace. As a result of such sorting methodology, the indexes that are built on multilingual data will arrange the strings from different languages in the same order as that of respective languages in the *Unicode* codespace. Finally, no database system currently supports *cross-language* querying of data – that is, searching across different languages for a given query string.

In summary, our survey indicates that all database systems do provide for the storage and management of multilingual data, primarily by supporting the *Unicode* character set and by specifying collations for supported languages. In query processing, each system offers the same SQL querying power in each of the supported languages; that is, the same querying capability (lexicographic, regular-expression matching, comparison, etc.)

is offered within *each* of the languages. However, no specialized operators or enhanced query semantics are offered to support queries *across* languages – that is, for combining information across database columns that may be in different languages.

1.2.3 Multilingualism in Database Research

While a rich body of literature on multilingual information processing exists from the Natural Language Processing [5] and Information Retrieval [116] communities, there is comparatively very little in the database context. In database literature, the multilingual data management issues may be classified as one of, solutions for specific languages, data integration solutions or proprietary solutions.

An implementation of a database system for Arabic data is presented in [71], where the authors present specific issues and solutions for storing, indexing, querying and presenting Arabic language data, in an object-oriented paradigm. A database for storing and query processing ideographic Chinese, Japanese and Korean (CJK) character data is detailed in [81], where the focus was primarily on definition of resources needed for handling ideographic scripts in database systems. While both these papers address issues specific to the languages concerned (Arabic and CJK languages respectively), neither of them propose general purpose solutions for multilingual data management.

In the second category, the FEMUS [4] system, though referred to as a multilingual database system, outlines a federated system that can integrate data from different *data-models* and the associated *query languages*; it does not address issues in integrating data from different *natural languages*. Similarly, a multilingual query processing framework for sharing lexical resources is discussed in [139], but the focus of this work is on improving the efficiency of administration of multilingual resources in a database environment, and not on multilingual query processing or performance.

Finally, proprietary solutions exist in integrating multilingual data in specific applications: The Look-Alike-Sound-Alike [78] (LASA) system is employed by the pharmaceutical industry, to identify strings that look or sound similar to each other, to prevent trademark violations and potentially dangerous medical situations. However, this system

works only in Latin-based scripts. The EROS [33] system for art conservation, matches multilingual records (that refer to diverse objects of art by masters) using specialized paired multilingual thesaurii that are specific to the art domain. However, such systems do not address general purpose multilingual data management issues.

To the best of our knowledge, there is not much research literature that deals with issues that are specific to supporting multilingualism in database systems or those that may be extended to general purpose solutions in multilingual data management.

1.3 A Sample Multilingual Application: *Books.com*

In order to highlight the multilingual data management issues, we first outline a hypothetical *e-Commerce* portal – *Books.com* – that requires multilingual data storage and query processing functionality. The same multilingual query paradigm can be used in other information systems that need to integrate multilingual data, such as, digital libraries, search engines, etc. This multilingual portal is used as a running example throughout the remainder of this thesis. Consider *Books.com* that sells books across the globe, with the database table – **Book** – that stores book information in multiple languages, as shown in Figure 1.1.

The **Book** table may be considered as a logical view assembled from data from a set of distributed unilingual databases, each of which stores book data in a local language, aligned with the local needs. A common view might have been defined, as shown in Figure 1.1, to support searches in a unified manner for multilingual users or for corporate reports. Clearly, the storage format of such text must be in an encoding scheme that is capable of storing all language data unambiguously; Without loss of generality, the data is assumed to be stored in **Unicode** [125] format, as **Unicode** is supported by all database management systems as the default format for storing multilingual data.

The table in Figure 1.1 stores the author’s first and last names, book title, price and the subject category of the book. The category of the book is assumed to be from a well-defined classification scheme that spans across languages. All the attributes of

Author	Author_FN	Title	Price	Category	Language
Durant	Will/Ariel	History of Civilization	\$ 149.95	History	English
Descartes	René	Les Méditations Métaphysiques	€ 49,00	Philosophie	French
राम	ब्रह्मचारी	भारत एक शक्ति	INR 175	ग्रन्थ	Hindi
Franklin	Benjamin	Ein Amerikanischer Autobiography	\$ 19.95	Autobiography	German
கார்த்தி	செனகேசுவரன்	சத்திய சேதனம்	INR 950	கவிதை	Tamil
Gilderhus	Mark	History and Historians	\$ 19.95	Historiography	English
Nero	Bicci	The Coronation of the Virgin	€ 99,00	Art/Film	Italian
Nehru	Jawaharlal	Letters to My Daughter	£ 15.00	Journal	English
無門	慧開	無門關	¥ 7500	禪	Japanese
Σοφία	Κατερίνα	Παρθένος στο Πνάο	€ 12,00	Μουσική	Greek
Lebrun	François	L'Histoire De La France	€ 75,00	Histoire	French
சென	செனகேசுவரன்	சத்திய சேதனம்	INR 250	சாத்திரம்	Tamil
Franklin	Benjamin	Un Américain Autobiographie	\$ 19.95	Autobiographie	French
بهنسی ، د	عقیف	العمارة عبر التاريخ	SAR 95	مضاري	Arabic

Figure 1.1: Hypothetical Multilingual *Books.com* Catalog

the records are assumed to be in the language of the publication. The column titled *Language* is presented here in English in order to enhance the readability of the table; it may be assumed to be an identifier corresponding to the language of publication, stored explicitly or derived implicitly based on the source of the record.

1.4 Research Issues Explored

Given the need for supporting multilingualism in an increasingly global economy and the state of the commercial art as given in Section 1.2, in this thesis we explore means of enhancing the support for multilingualism in relational database systems – both in terms of performance and functionality – as follows:

1.4.1 Multilingual Performance

While the survey given in Section 1.2 indicates the near-uniform multilingual support of database systems using Unicode, to the best of our knowledge, no quantitative data has been published on the performance of the systems working on Unicode data; that is, their multilingual performance, specifically their relative performance with respect to

that on default character set, is largely unknown.

Hence, in the first part of our research, we set out to quantify the multilingual performance of a suite of current commercial and open-source relational database systems, and to alleviate any differential performance in handling multilingual data.

1.4.2 Multilingual Names Matching

As discussed earlier, the current relational systems do not offer any alternate matching semantics for integrating data across languages. To rectify this state of affairs, we define two functionalities for matching multilingual data: First, the functionality of **Multilingual Names Matching** is defined, as the retrieval of records that store the same proper name, perhaps in different natural languages, including names with minor spelling variations in each of the languages. Though restricted to proper names, such matching represents a significant part of the user query strings in text databases and search engines, as proper and generic names constitute a *fifth* of normal corpora [67].

For example, consider a query by a multilingual user to retrieve the works of an author, say **Nehru**, in English, Greek, Hindi and Tamil, in the *Books.com* catalog shown in Figure 1.1. A SQL:1999 compliant query and the result set for this retrieval is as given in Figure 1.2. Note that the output would be in the language of the respective records.

<pre>SELECT Author, Title FROM Book WHERE Author = 'Nehru' OR Author = 'Nηρρυ' OR Author = 'नेहरु' OR Author = 'நேரு'</pre>	
Author	Title
Nehru	Letters to My Daughter
நேரு	சூசிய கண்காணி
नेहरु	भारत एक खोज

Figure 1.2: SQL:1999 Compliant Multilingual Names Query and Result Set

Such a query specification that requires the author's name in several languages is undesirable, due to requirement of linguistic expertise of the user and the availability

of special lexical resources in several languages for the query input. Further, given that the error rate for English query input is approximately 3% [67], the error in multilingual query input could be expected to be much worse. Also, similar to the case of monolingual matching, any differences in the spellings between the stored name and the query string will result in false-dismissals.

Formally, a more intuitive **Multilingual Name Join** operator (referred simply hereafter as **MLNameJoin**) is defined as follows: **MLNameJoin** takes an input name in one language, and returns all records that have the same name in all or in a user-specified set of languages. The input query name may be specified either in the most comfortable language for the user or the one for which the lexical resources for input are available. The multilingual query given in Figure 1.2, is shown in Figure 1.3 specified with the **MLNameJoin**, producing an identical result set. The **Threshold** parameter specified in the query determines the quality of matches, as described later in this thesis.

<pre>SELECT Author, Title FROM Book WHERE Author MLNameJoin 'Nehru' Threshold 0.25 InLang { English, Greek, Hindi, Tamil }</pre>	
Author	Title
Nehru	Letters to My Daughter
நேரு	சூனிய கௌரவி
नेहरु	भारत एक खोज

Figure 1.3: A Multilingual Names Query

The **MLNameJoin** operator, in addition to having a simpler input syntax, has two powerful features that extend the power of standard **SQL**: First, it can express retrieval of all records matching a name *irrespective of the language*. The specification of **ALL** for the list of languages retrieves all records containing the same author name in any of the languages. Second, it can also express a join functionality (as given in Section 1.4.4) that is not possible with standard **SQL** syntax. In the second part of our research, we explore a strategy for implementing multilingual names matching on unmodified relational database systems and optimize the performance of such an implementation.

1.4.3 Multilingual Semantic Matching

A second Multilingual Semantic Join operator (referred simply hereafter as `MLSemJoin`) is defined as follows: `MLSemJoin` takes an input query classification and outputs all records that have classifications that are equivalent to (or, optionally, subclass of) the input classification, irrespective of the language of the record. This matching is restricted to attributes that store the categorical value of a record, possibly in different languages.

Consider in the *Books.com* example, an SQL:1999 compliant query to retrieve all `History` books in a set of user specified languages. Clearly, a query with a selection condition as `Category = 'History'` would return only those books that have `Category` as `History`, in English. A multilingual query to retrieve the required answer set needs specification of categorical value strings that are equivalent to 'History' in all the languages in which output is desired. Figure 1.4 shows such a multilingual query in which the query categorical value, 'History', is specified in all the target languages, namely, English, French and Tamil. The output records have categorical values that are semantically equivalent to 'History'⁴.

<pre>SELECT Author, Title, Category FROM Book WHERE Category = 'History' OR Category = 'Histoire' OR Category = 'சரித்திரம்'</pre>		
Author	Title	Category
Durant	History of Civilization	History
செரு	ஆசிய ஜோதி	சரித்திரம்
Lebrun	L'Histoire De La France	Histoire

Figure 1.4: SQL:1999 Compliant Multilingual Semantic Query and Result Set

Just as in the multilingual names matching, specification of the categorical value in different languages may be undesirable due to the need for linguistic expertise and specialized lexical resources. Further, the synonyms of the query terms will not be retrieved,

⁴The second record has as *category*, the value சரித்திரம் (transliterated as, *Charitram*) in Tamil, meaning History.

even within the same language (such as, **Annals**, **Chronicle**, etc., that are synonyms of **History** in English). Finally, even if all the synonyms are specified explicitly, classification that are specializations of a query term will not be retrieved (such as, **Biography**, **Autobiography**, **Genealogy**, etc., which are specializations of **History** in English).

We define formally the multilingual semantic matching as **Multilingual Semantic Join** operator that takes input categorical value in one language and returns all multilingual records that semantically match the input categorical value. Figure 1.5 shows the same example as in Figure 1.4, but using the **MLSemJoin** operator, producing an identical result set. The output contains all books that have categorical values semantically equivalent to **History** in the respective languages.

<pre>SELECT Author, Title, Category FROM Book WHERE Category MLSemJoin 'History' InLang {English, French, Tamil }</pre>		
Author	Title	Category
Durant	History of Civilization	History
செரு	ஆசிரியர் ஜோதி	சரித்திரம்
Lebrun	L'Histoire De La France	Histoire

Figure 1.5: A Multilingual Equivalent Semantic Query and Result Set

The definition of the **MLSemJoin** operator may be made more general, to match not just on categories that are *equivalent* to the query categorical value, but also to those that may be *generalized* to the query category. In Figure 1.6, the **MLSemJoin** operator with the optional **ALL** clause takes an input category and returns all books that have multilingual categories that may be generalized to **History**. Note that the first three records in the output are the same as in Figure 1.5 and have the *Category* value equivalent to **History**, in English, French and Tamil, respectively. The last four records have categorical values that are *subsumed* by **History**⁵, in the languages specified in the query for output.

⁵**Historiography** (Oxford English Dictionary – OED – definition: *the study of of history writing and written histories*), **Autobiography** (OED definition: *writing ones own life history*) and **Journal** (OED definition: *a personal record*) are considered as specialized branches of **History** itself. The fifth record

<pre>SELECT Author,Title,Category FROM Book WHERE Category MLSemJoin ALL 'History' InLang {English, French, Tamil }</pre>		
Author	Title	Category
Durant	History of Civilization	History
டூரன்	சுமியர் டூரன்	சரித்திரம்
Lebrun	L'Histoire De La France	Histoire
Gilderhus	History and Historians	Historography
கர்ட்டி	சுமியர் டூரன்	சுமியர்
Franklin	Un Américain Autobiographie	Autobiographie
Nehru	Letters to My Daughter	Journal

Figure 1.6: A Multilingual Generalized Semantic Query and Result Set

The MLSemJoin operator has features that are not expressible in standard SQL: First, MLSemJoin may be used for retrieving *all* records, irrespective of language, that match a specific categorical value. Second, matching on categories that are *equivalent* and *subsumed* by the query categorical value is possible with the proposed operator. Finally, MLSemJoin may be used for defining a join between two multilingual columns, a feature not possible in standard SQL.

In the third part of our research, we explore a strategy for implementing multilingual semantic matching using standard linguistic resources and present its implementation, along with optimization techniques to make the operator efficient for practical use.

1.4.4 Complex Multilingual Operations

Similar to normal SQL operators, the multilingual query operators, namely MLNameJoin and MLSemJoin, may be combined with other SQL operators, as well as among themselves, to express more complex queries, depending on the user needs in an application domain. The declarative specification of such queries makes them intuitive to understand and express, in addition to being amenable for optimization.

has as *category* the value சுயசரிதம் (transliterated as *suyacharitam*) in Tamil, meaning Autobiography.

Consider the e-Commerce portal *Books.com*, whose catalog of books (the **Book** Table) is as given in Figure 1.1. Let the publisher information is stored in a **Publisher** table, as shown in Figure 1.7. Assume that the **Book** record includes a foreign key (**Book.PubID**) to the publisher of the book.

Publishers	PubID	Address	Specialization	Language
McGraw-Hill	P001	New York, United States	Technical	English
சிறீலக்ஷ்மி	P002	Chennai, India	Quincy	Tamil
Rutledge	P003	London, United Kingdom	Philosophy	English
Addison-Wesley	P004	Reading, MA, United States	General	English

Figure 1.7: Multilingual Publisher Table

Suppose the user wants to retrieve *the books whose author's name is similar to that of the book's publisher*. This query requires a join between the **Book** and **Publisher** tables, as shown in Figure 1.8. While the first equijoin join is necessary to establish the relationship between the books and the publishers, the second **MLNameJoin** operator verifies similarity of names.

```
SELECT B.Author
FROM Book B, Publisher P
WHERE B.PubID = P.PubID
      AND A.Author MLNameJoin P.Publisher threshold 0.25
```

Figure 1.8: Sample Multilingual Complex Query – 1

Consider another query *to retrieve the books that are published in subjects that are outside the publisher's area of specialization*. This query requires an explicit join of **Book** and **Publisher** tables to establish the publication relationship and a negated **MLSemJoin** operator, as specified in Figure 1.9:

An interesting self-join variation of a complex query *to retrieve those authors who have published books in at least two distinct areas (possibly in different languages)*, is specified as shown in Figure 1.10. This query requires a self join of **Book** table using **MLNameJoin** operator, to ensure similarity of names in different languages, and a pair

```
SELECT B.Title
FROM Books B, Publisher P
WHERE B.PubID = P.PubID
AND NOT (B.Category MLSemJoin ALL P.Specialization)
```

Figure 1.9: **Sample Multilingual Complex Query – 2**

of negated `MLSemJoin` operators between the categories. Note that this answer set is a superset of the real results, and requires filtering out those distinct authors who have similar names.

```
SELECT B1.Author, B2.Author
FROM Book B1, Book B2
WHERE B1.Author MLNameJoin B2.Author threshold 0.25
AND NOT ((B1.Category MLSemJoin ALL B2.Category)
          OR (B2.Category MLSemJoin ALL B1.Category))
```

Figure 1.10: **Sample Multilingual Complex Query – 3**

1.5 Proposed Solution Strategies

Next, we outline our research strategy and our solution methodologies for implementing the proposed multilingual functionalities in relational database systems.

1.5.1 Design Goals for Our Research Strategy

The following design goals were pursued, for the addition of multilingual features to the relational database systems in a useful, usable and scalable manner.

Relational Systems Oriented: Our focus is on adding multilingual support to relational database systems – the backbone for most current information systems.

Attribute Data Oriented: The focus of multilingual query processing is only on attribute-level data, in order to support high-volume Internet-based applications. To have a fast, light-weight query processing of attribute-level data, the usage of linguistic resources, rather than Natural Language Processing (NLP) techniques was pursued. Note that our approach is well suited for search strategies on documents, since the inverted index used in the text processing systems contains primarily stemmed text keywords that are similar to attribute data.

Standards Based: Standard linguistic resources must be preferred in order to ensure uniformity and consistency in multilingual query processing, across different database systems. As a side-effect, the techniques would yield the same answer set for a given query on a given data-set, irrespective of the systems on which they are implemented.

Customizable Matching: The matching must be customizable by users, depending on the requirements of specific domains and applications.

Few Database Kernel Changes: Database software has been developed and fine-tuned over a period of decades, representing substantial resources spent by academic and industrial research communities. Hence, our aim is to leverage the capabilities of the system with minimal changes. Also, such an approach makes it easier for adoption of our methodology among the existing systems.

1.5.2 Multilingual Performance

Our survey on the multilingual support offered by a suite of popular commercial and open-source relational database management systems indicates that almost all the systems do provide for the storage and management of multilingual data, by their support of Unicode as the storage mechanism. However, there is no published research literature on performance implications of Unicode data in database systems.

Hence, the first part of this thesis, Chapter 2, focuses on quantifying the performance of systems handling multilingual data using the standard TPC benchmark suites, modified appropriately for multilingual environments. The objective was to quantify the differential performance of popular commercial and open-source relational database systems in storing and processing multilingual data in the Unicode character set, as compared with their performance on the default ISO:8859 character set and to explore ways of making the performance *natural language neutral*.

1.5.3 Multilingual Matching Functionalities

In the second part of the thesis, we pursue solution strategies for supporting the cross-lingual query processing functionalities, specifically for implementing the multilingual names matching and multilingual semantic matching functionalities (as in Sections 1.4.2 and 1.4.3).

Our view of storage and semantics of textual information in database systems is shown in Figure 1.11. Note that our discussion here pertains exclusively to the text data types, which are relevant for multilingual information processing. Non-text datatypes, such as `number`, `date` etc., are not relevant and hence are not addressed in this thesis. The top half of the figure sketches the ontology of text data stored in database systems, and the bottom part of the figure sketches the storage mechanisms that are used to store textual data. Simply, the top half refers to *what* types of textual data is stored in database systems and the bottom half refers to *how* they are stored. The dotted and dashed lines represent how the matching semantics of specific attribute types are implemented.

The **Text Data** in database systems may represent a wide variety of information: **Text String** that stores singular string representing a proper name (tagged as **Proper Noun** in Figure 1.11), a categorical information (tagged as **Category** in Figure 1.11) or a compound string (tagged as **Other Text Data** in Figure 1.11). The **Other Text Data** may represent complex information, such as, an address, a description etc. Another important type of text information in databases is a document, which may be a very long string

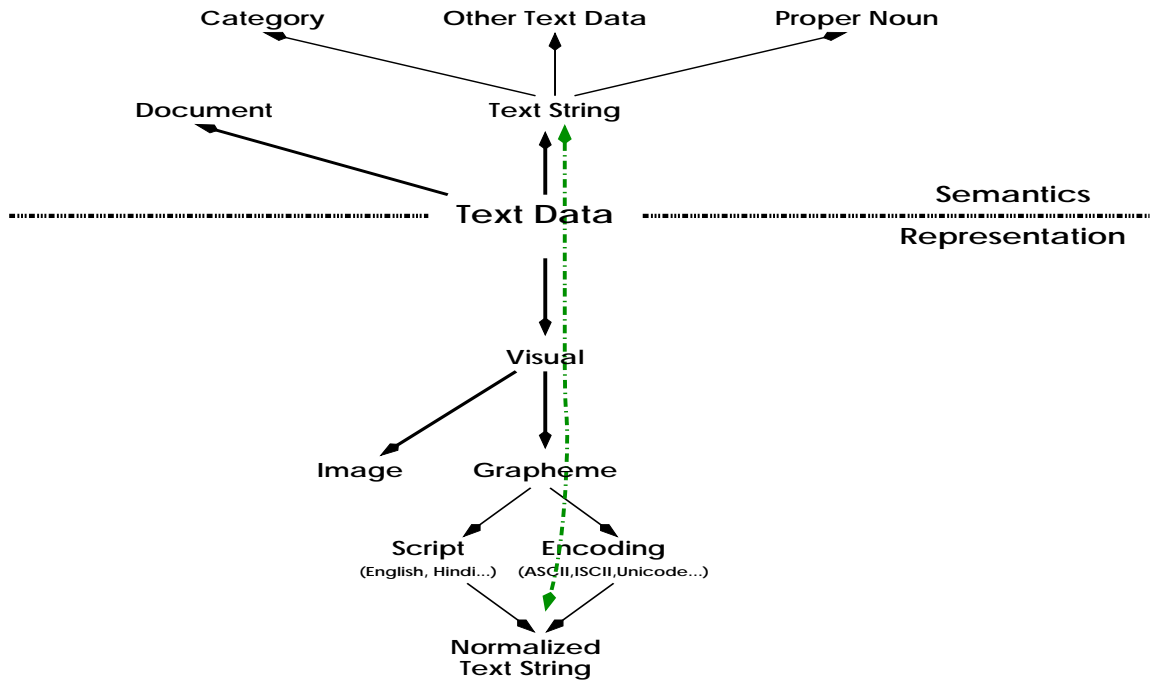


Figure 1.11: Ontology for Text Data

representing a full document (tagged as Document in Figure 1.11). The document may be stored either as a long text string in the table itself or as a binary object in a local or a foreign table or as a logical pointer to the database location where the actual document is stored; each database system may implement the document storage differently and hence it is not classified under Text String. As mentioned in Sections 1.4.2 and 1.4.3, this thesis focuses on multilingual query processing on the Proper Noun and Category types of textual attributes only. Query processing on Other Text Data and Document attributes requires Natural Language Processing (NLP) algorithms and are beyond the scope of this thesis.

How the information is represented in the database systems, is sketched by the lower half of Figure 1.11. Though multimedia systems may store the visual representation, namely, *Glyphs*⁶, as images of the text (tagged as Images in Figure 1.11), normal text databases store them only as graphemes internally. The Grapheme representation in a database depends on two orthogonal specifications: the Script (such as, *English, Hindi*,

⁶The characters are composed into *Glyphs* by a rendering engine, based on the composition rules and visual representations that are specific to a language.

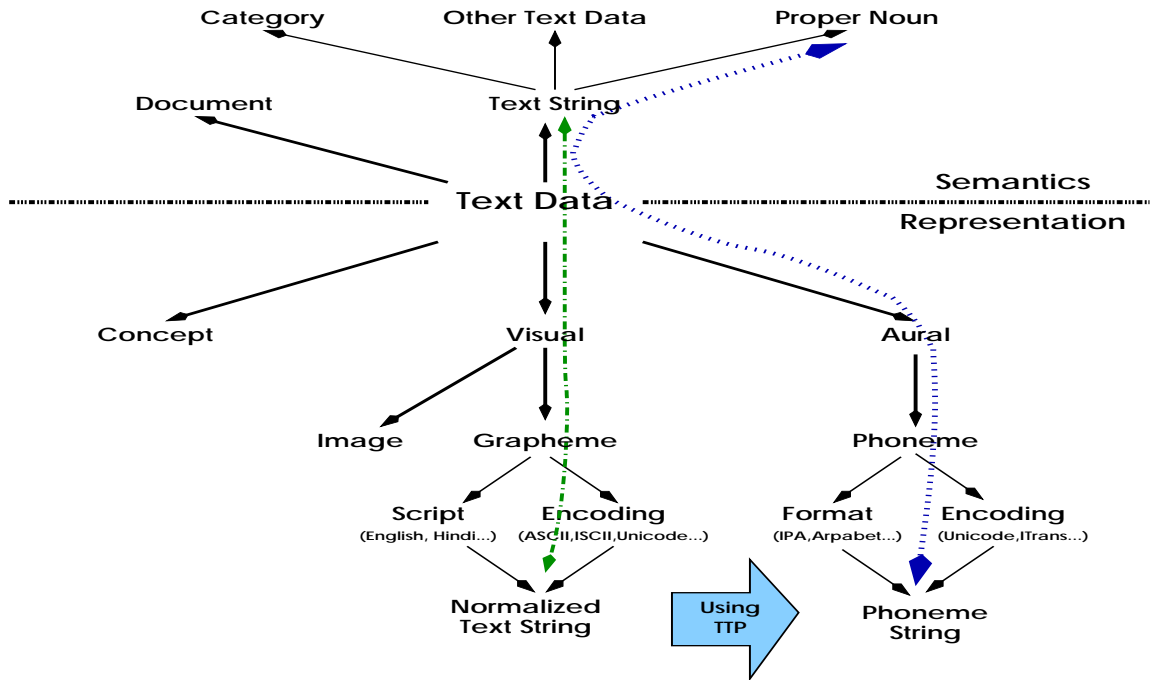
Arabic, Chinese, etc.) and the **Encoding** (such as, ASCII[57], ISCII[122], Unicode[125], etc.). Given the two specifications, a multilingual text string is represented by a **Normalized Text String**. While, in general, the **Encoding** is specified at the database creation time (as the *database character set*), the **Script** is not usually specified explicitly.

Databases employ lexicographic comparison (represented by the *dash-dotted* line in Figure 1.11) for all the text operations (such as, matching, sorting, search, etc.), but this facility fails in multilingual environments – for example, the matching always fails since the strings are represented in different scripts and the sorting depends solely on the placement of the scripts in the **Unicode** codespace. Hence, in multilingual environments, the semantics of the matching operator itself must be defined and their implementations explored. The solution strategies for the two alternative **MLNameJoin** and **MLSemJoin** matching semantics are presented in the following sections.

Multilingual Names Matching Strategy

In multilingual environments, for a specific class of attributes, such as those that store the names of individuals, corporations and places, we start with the premise that the value of the string is primarily *aural*; that is, when a name is explicitly queried for, the user may be interested in retrieving all names that *aurally* match the query string irrespective of the language (at least in the set of languages in which the user is interested). For example, in **MLNameJoin** matching semantics, comparing “Nehru” and “नेहरु” should succeed, as both the strings encode the same name in English and Hindi, respectively. We propose a framework to capture this intention, by transforming the multilingual name match in the *text space* to a corresponding match in the *phoneme space*.

The database text data ontology is augmented to store the phonemic equivalents of the multilingual text strings. Note that though the ontology is augmented, the phoneme strings need not be materialized and stored explicitly; they may be generated at the query processing time. However, materialization may be employed, in order to improve the query processing efficiency. The storage of phoneme strings parallels the storage of grapheme strings in the database systems: a phonemic string may be represented by

Figure 1.12: **Ontology for Names Matching of Text Data**

specifying a Format and an Encoding. The Format is one among several competing *phonetic repertoires* that specify different phoneme alphabets. While a variety of phoneme formats are available, the phoneme alphabet specified by the International Phonetic Association [60] (IPA) was chosen in our strategy, as it covers the set of phonemes from all the languages and its repertoire is explicitly specified and supported in Unicode. The Encoding specification for the phoneme strings as Unicode allows the storage and manipulation of phoneme strings in the NChar datatype, in all the database systems. The phonetic representations of given multilingual strings may be derived from the standard *Text-to-Phoneme* (TTP) systems of the respective languages. However, the transformed phoneme strings of the same name in different languages may not match exactly, since phoneme sets of two languages are seldom identical; that is, the same name in two different languages may transliterate only to a pair of *close* phoneme strings. Hence, we propose the use of approximate matching techniques to compare phoneme strings. In summary, we propose to implement the `MLNameJoin` operator by transforming multilingual text strings to their equivalent phonemic strings in a common alphabet (IPA)

and employing approximate matching techniques (shown in *dotted* line in Figure 1.12), instead of the standard lexicographic matching (shown in *dash-dotted* line in Figure 1.12).

Storage of aural representation of multilingual names as an audio file (in a specific format) or as a mathematical representation (such as the Fourier Transformation of the waveforms) is possible; however, these choices result in heavy overheads in storage and/or query processing. Further, our methodology leverages the ready availability of the multilingual names as text strings in the database systems.

While the use of pre-defined mappings between names in different languages is possible, this approach suffers from several drawbacks: First, even in the monolingual domain, proper names suffer from large expansions of the term being matched⁷; in multilingual searching, the expansion may be worse. Second, a new name (for which the corresponding variations may not be readily available) given as an input cannot be handled effectively. Third, such an approach requires creation and constant maintenance of meta-data information that could result in large manual overheads. In contrast, our approach is very generic, and depends only on integration of language-specific TTPs and approximate matching operators with the database system.

Chapter 3 details our strategy of implementing multilingual names matching, by converting matches in *text space* to *phoneme space*. Further, an *outside-the-server* approach to implement the functionality on existing database systems is presented, along with optimization techniques to make the performance sufficient for practical use.

Multilingual Semantic Matching Strategy

Consider the class of attributes that store the classification information of a record (such as, the *Category* attribute in *Books.com* table). The values of this attribute are usually specific Concepts from a well defined set. For example, the *Category* attribute in *Books.com* may be from the *Dewey Decimal Classification* [29] that classifies all subject categories. Hence, we propose that the categorical value strings may be compared after

⁷For example, a popular Internet search engine [50] employs about 600 variations of a popular query name – *Britney Spears*, to correct a mis-spelt form of the name.

transforming them to canonical semantic atoms using appropriate linguistic resources⁸. Further, if the semantic atoms are arranged in a semantic network for the domain (such as, Dewey Decimal Classification that arrange the subject categories in a taxonomical hierarchy), more expressive power may be added to the matching, leveraging on the richer semantics available.

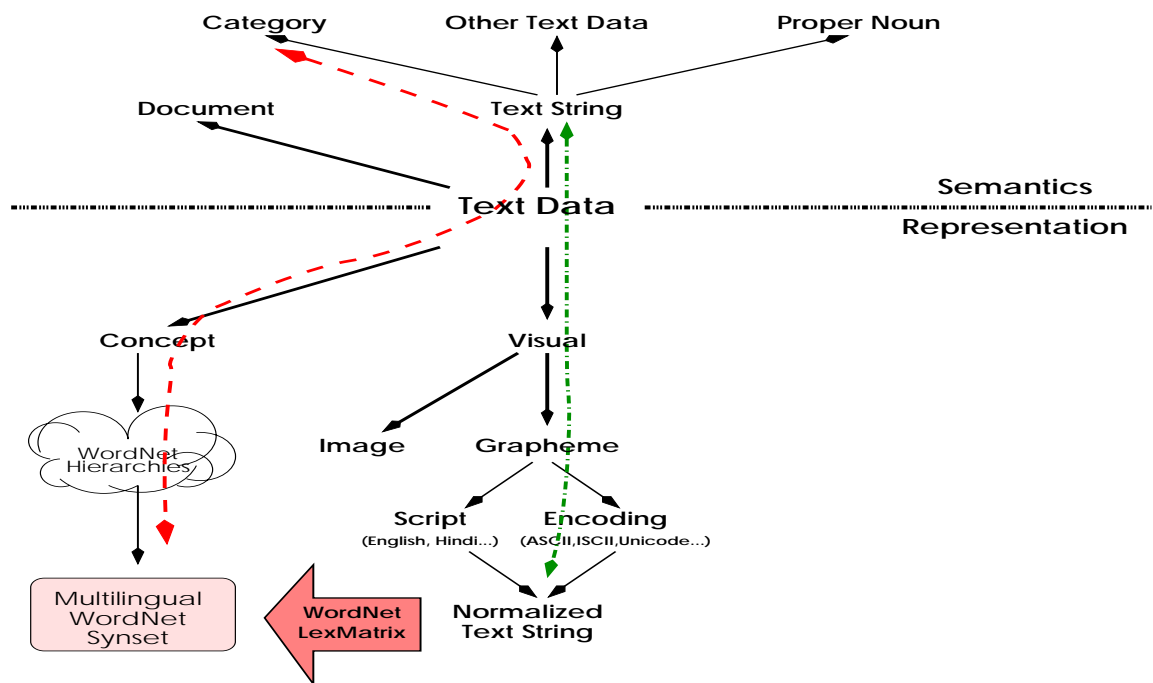


Figure 1.13: Ontology for Semantic Matching of Text Data

For MLSemJoin matching functionality, a specific linguistic resource – the *WordNet* [135, 39], a *Computational Linguistics* resource that arranges the concepts of a language using psycho-linguistic principles, is used. WordNet contains two important features that may be leveraged for semantic query processing in the linguistic domain: First, a *Lexical Matrix* that converts a *word form* (lexicographic representation) to a *word sense* (the semantic atom of the language, called *Synset*). Second, the *Taxonomic Hierarchy* that arranges all the synsets of a language in an inheritance hierarchy based on

⁸Here we distinguish from *Words* and *Semantic Atoms*, though the semantic atoms may be expressible only in terms of words. The words in a natural language, in general, have low resolution power and refer to multiple concepts. For example, in the two contexts “bow and arrow” and “ship’s bow”, the same word – bow – instantiates to distinctive *semantic atoms*.

their meanings. Using the above two resources word forms may be mapped to synsets of that specific language and compared. Further, with the available taxonomical hierarchy of noun forms, matching may also be defined on *specializations* and *generalizations* of the synset corresponding to the query term. For multilingual domains, more importantly, efforts are under way to develop WordNets in many languages with semantic links between their individual synsets [37, 65, 19]. The multilingual categorical attribute values may be matched leveraging the interlinked WordNets in multiple languages (shown as *dashed* line in Figure 1.13), instead of the standard lexicographic matching (shown in *dash-dotted* line in Figure 1.13).

Chapter 4 presents details of our implementation strategy for multilingual semantic matching, by leveraging the WordNet linguistic resources that are available in multiple languages. The implementation methodology uses standard SQL:1999 features on unmodified relational database systems. Further, optimization strategies for improving the performance, by tuning storage and access structures to match the characteristics of the linguistic resources, are presented.

1.5.4 Multilingual Query Processing Architecture

Traditionally, a new functionality is added to the database systems as a user-defined function (UDF), due to its simplicity. However, such an approach suffers from performance overheads due to remote execution and due to the fact that a query using UDFs cannot be optimized by the query optimizer. Further, the query specification tends to be unintuitive, since the functional UDF calls get interspersed with declarative SQL constructs. Hence, for an intuitive and efficient functionality enhancement in database systems, the functionality must be made available as first-class engine operators.

Chapter 5 presents a unified multilingual query processing architecture that *natively* integrates the proposed multilingual functionality to the database system. A multilingual query algebra – Mural – that specifies a uniform framework for expressing complex queries declaratively and intuitively, is presented. Subsequently, Chapter 6 presents a *native* implementation of the query processing architecture with Mural algebra on the

PostgreSQL open-source relational database system. The performance of such a native implementation is presented and the power of optimization opportunities that it affords, is demonstrated.

1.5.5 Applicability in Other Domains

It should be specially noted here that though our solution methodology is primarily designed for matching multilingual strings, it is equally applicable for matching of monolingual strings (for example, for matching text strings that are all in English).

Monolingual Names / Semantic Matching

The `MLNameJoin` operator may be used for matching the English name `Catherine` and all its variations, such as `Kathrin` and `Katrina`, with the names being matched *phonetically* using English TTP and using approximate matching techniques. Similarly, using only the English WordNet, the semantic matching methodology presented here may be used for matching `Disk Drive` with `Computer Storage Devices`.

Domain-specific Semantic Matching

The semantic matching methodology outlined here may be applied, in general, to any domain, where a well-defined ontological hierarchy of concepts is available. For example, while the WordNet linguistic ontological hierarchies were used in our methodology, the same query processing methodology may be applied in Bio-informatics domain by using *Gene-Ontology* [45] ontology or in Library domain by using *Dewey Decimal Classification* [29] system. Specialized domain hierarchies, such as *Yahoo!Directory* [137] may also be used for specific applications.

1.6 Real-life Multilingual Systems

In this section, a few multilingual initiatives from around the globe are outlined. While all the initiatives given below cater to a multilingual population, none of them support cross-lingual searches, as defined in this thesis. The functionality we proposed could enhance the multilingual support provided by each of them, in a meaningful way.

Global e-Governance Portals⁹

In the European Union, one of the most linguistically diverse regions of the world, a pan-European Union portal – Europe [34] – disseminates information on its legislative, judicial, economic and social programmes to the member citizens, in all of the EU languages. The goal of this portal is *to provide the public with the information that they are looking for in their own language or in a language that they can understand*.

Similarly, the portals of United Nations [126] or United Nations Educational, Scientific and Cultural Organization [127] present data in the six official languages of the organizations – namely, English, French, Russian, Spanish, Arabic and Chinese. Among the stated objectives of the organizations is *the dissemination of information to all member organizations in a timely and transparent manner, to foster better governance*.

Indic e-Governance Portals

The Government of India has introduced several e-Governance initiatives to serve the citizens of India, efficiently and effectively. Use of local language content is encouraged, since it provides better access for the vast majority of the population that are functionally literate only in the local language. For example, Bhoomi [9] is a *e-Governance* system designed for computerizing land records of about 20 Million agricultural properties in the local language, *Kannada*, serving 7 Million farmers in the state of Karnataka in India. The system is being modified for adoption in different states of the Union of India, in the respective local languages.

As a result of such multilingual e-Governance systems, other system that need to integrate data from different sources need to scale multilingually: consider the Income Tax department of Government of India, which requires a citizen of India to file an income tax return, if he/she satisfies any two of the following six criteria [51]:

- Owns a landed property
- Possesses a passport and travels abroad

⁹Though this portal deals with documents, as indicated earlier, the proposed multilingual matching operator may be useful here, as the keywords used for inverted-indexes and document searches are similar to the attribute data in relational database systems.

- Owns a motor vehicle
- Subscribes to a telephone or a mobile phone connection
- Possesses a credit card
- Is a member of any exclusive clubs.

An automated system to identify potential tax payers must work with information that is in different languages; for example, as mentioned earlier, the land records are maintained in a local language for a large portion of rural properties, whereas the telephone and banking records are maintained in English. Hence it introduces a problem of matching of names in Indic character set from land records, with names in English from telephone records, presenting a potential application for the `MLNameJoin` operator. Similarly, there are other demographic data (such as, profession, religion, etc.) that require the `MLSemJoin` operator. The matching problem is also compounded by the fact that the income tax returns themselves may be filed in either English or Hindi. Hence, matching potential and existing income tax assesseees require merging of data in potentially three or more languages.

aAQUA: A Rural Multilingual Agricultural Portal

aAQUA [1] (almost All QUestions Answered) is a project of Indian Institute of Technology-Bombay (IIT-Bombay), and a part of Development Gateway India Research Center, funded by Government of India. It is currently operational in English and two Indic languages, to enable Indian farmers to get in touch with and get advice from by the agricultural experts. The portal, as shown in Figure 1.14, provides a multilingual interactive forum for farmers – primarily from the state of Maharashtra in India – to interact with experts and among themselves. This portal also provides constantly updated prices on agricultural commodities in the markets.

The query address system hosted in this portal allows farmers to post queries that are answered by experts about any agricultural issues of interest, in any of the supported languages; in addition, it provides a wealth of information on common problems faced by the farmers, and data about appropriate crop cycles, pesticides, soil and weather



Figure 1.14: aAqua: An Indic Multilingual Agricultural Portal

conditions. Potential features that may be supported on the top of aAQUA are the multilingual names search on entities (say, pesticide names), and the multilingual semantic search on concepts (say, “vegetable prices”). The pesticide name may be searched in all the supported languages as the names are likely to be transliterated in different languages; the resulting documents may be presented to the user in his/her own language as the translated pages are also readily available in aAQUA. The semantic search may return potential answer set for a given search; for example, “price of vegetable” search may yield prices for different vegetables, and “crop failure” may retrieve articles ranging from *insects in rose buds* to *flooded rice fields*.

Vidyanidhi: E-Scholarship Portal

Vidyanidhi [130] is India’s premier digital library initiative to facilitate the creation, archiving and dissemination of doctoral theses that are produced in a host of Indian universities, in English and a set of Indic languages. Vidyanidhi is envisioned to evolve as a national repository of research publications in India, encouraging dissemination and sharing of knowledge. Such a portal may support searching of potential plagiarism, by semantically matching the multilingual keywords associated with the scholarly work in different languages.

Multilingual Search Engines

There are several initiatives that attempt to search multilingual web-sites and documents. EuroSeek [35] – shown in Figure 1.15 is a search initiative that works currently with most major European languages and has a stated goal of *creating a pan-European search engine that is transparent to national and linguistic boundaries*.

However, EuroSeek, based on the popular Google [50], supports searches based on patterns (that is, lexicographic) only. As can be noticed in Figure 1.15, the query word “Mira” is searched only lexicographically even on a collection of documents that are not in Latin script. Similarly, even in the multilingual UN and UNESCO portals, search is compartmentalized to each of the official languages; integration happens, at best, in matching between those languages that share the same script – such as English and French. In contrast, some of the local initiatives, such as, **Agro Explorer** [105, 106], search based on the meaning that is represented in UNL [129], and hence are language independent. All search engines may leverage on the alternative matching methodologies highlighted in this thesis, and enhance support in multilingual domains.

1.7 Organization of this Thesis

The remainder of this thesis is organized in the following manner:

Chapter 2 profiles the performance of a set of popular database systems in handling multilingual data, using the standard TPC performance test suites, modified appropriately for

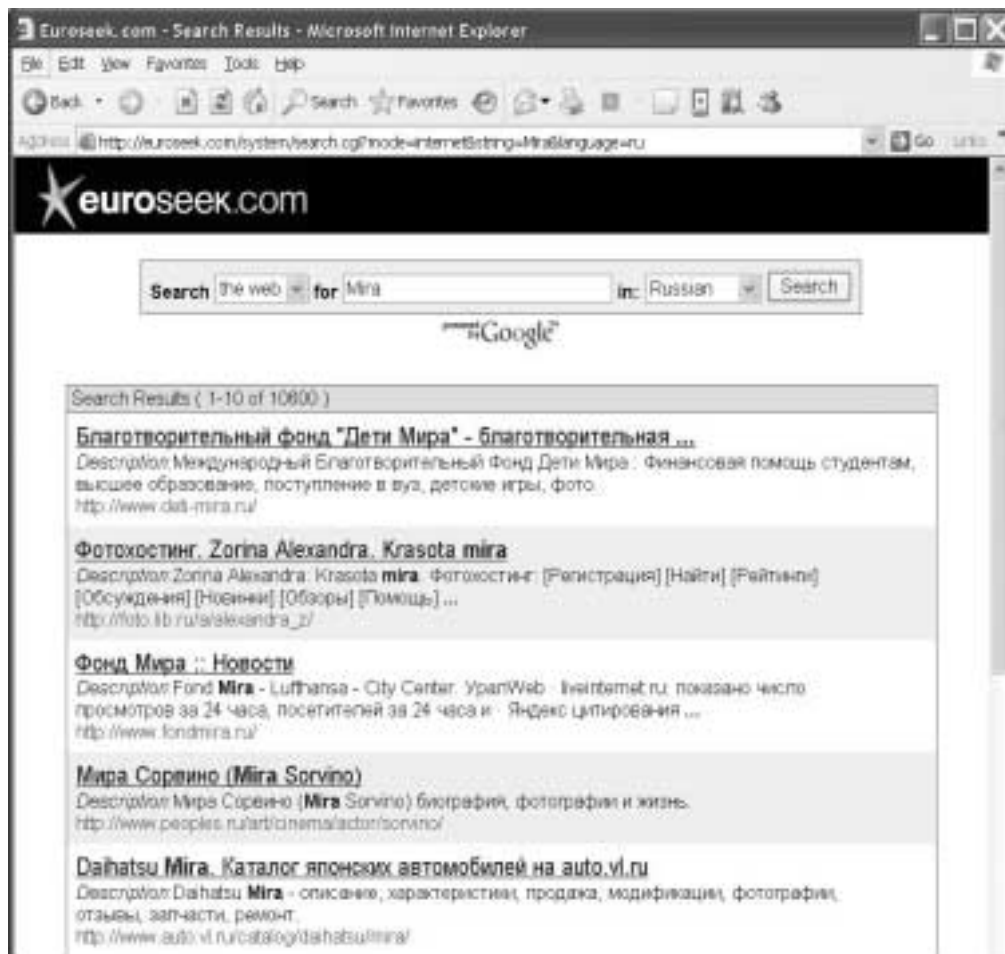


Figure 1.15: Euroseek: A Pan-European Multilingual Search Engine

multilingual environments. While the results highlight the differential multilingual performance of the database systems, we explore efficient character representation formats to make the performance equitable across languages.

Chapter 3 outlines our implementation strategy for multilingual names matching, by transforming matches in *text space* to *phoneme space*. While the basic implementation using UDF's is too slow for practical use, we show that with specialized indexing techniques, the performance may be improved substantially.

Chapter 4 outlines our implementation strategy for multilingual semantic matching, by leveraging WordNet linguistic resources. We present our implementation using standard SQL features on unmodified database systems and a set of optimization techniques that demonstrate acceptable performance for practical use.

In Chapter 5, we formalize the previous functionality proposals as operators and define a query algebra that ties them together in a unifying multilingual query processing architecture, along with all the components needed for a native implementation of the multilingual functionality in a relational database system.

In Chapter 6, we outline a *native* implementation of the multilingual operators along with the query algebra, in the PostgreSQL open-source database system. Subsequently, the performance of such a native implementation is presented, along with the optimization opportunities afforded by such an implementation.

Finally, Chapter 7 concludes the thesis, with avenues open for further research, in extending the problem or solution methodologies.

Traditionally, the database management systems have become transparent to physical storage formats (by automatic transcriptions) and to logical data models (by programming-language-neutral access methods). In this thesis, we take the next step of making the query processing transparent to the natural languages, with a set of functionalities, algorithms, implementation techniques and an architecture, all geared towards the goal of developing *natural-language-neutral* database engines.

Chapter 2

Multilingual Performance of Current Systems

2.1 Overview of the Chapter

In this chapter, we profile the performance of a set of popular database systems in handling multilingual data, using the standard TPC [123] performance test suites modified appropriately for the multilingual environment. The results highlight the inequitous performance of the database systems while working on multilingual data, compared with their monolingual performance. To alleviate the magnitude of such inequity, we propose a split storage format that largely eliminates the differential performance for most languages, except those with unusually large repertoires (specifically, those with repertoire size > 256).

2.2 Setup for Multilingual Performance Study

In this section, we first describe a testing framework for measuring the differential performance of database systems with respect to multilingual data. Next, we define the metrics measured to quantify the differential performance and subsequently present the performance of a suite of popular commercial and open-source database systems.

2.2.1 System and Database Environment

A standard Intel Pentium IV (1.7 GHz) workstation with 256MB memory running Windows 2000 Professional operating system was used as the test machine for the performance study. All the database systems were installed and tested on this machine to normalize the effects of the hardware environment. Before each experiment, the machine was quiesced and only the database system being tested and allied processes were allowed to run in order to have measurement parity between the systems.

Four of the popular database systems – specifically, Oracle *9i Database Server* (Version 9.0.1), IBM *DB2 Universal Server* (Version 7.1.0), Microsoft *SQL Server* (Version 8.0.194) and PostgreSQL *Database Server* (Version 8.0.1) – were evaluated in our performance study. In the performance section, they are identified randomly, as *A*, *B*, *C* and *D*, to protect their identities. The database systems were installed with default configurations, with the vendor-provided installation scripts. All the systems were configured to use only 64 MB for the database buffer pool, a popular choice among the systems. No optimization of the parameter settings was attempted, as the focus of our study was to report the performance of the database systems under default conditions and not to optimize individual performance. It is worth noting here that apart from the format specification of NChar datatype, we found no other database system parameters that are specifically designated for multilingual character sets.

2.2.2 Dataset

The TPC-H benchmark [123] data generator was used to generate a large database for the performance study. A specific table (`partsupp`) that stores the part-supplier relationship¹ was modified further to hold equivalent data in the default Char character set and the multilingual NChar character set, as shown in Figure 2.1, for our experiments. Specifically, two different tables – `partsuppChar` and `partsuppNChar`, with attributes

¹The *Part-Supplier* relationship in TPC-H benchmarks captures the relationship between *Parts* and *Suppliers*. In addition to foreign keys identifying the part and a supplier, it stores three additional attributes – *Price* quoted for the part by the supplier, *Quantity* available with the supplier and *Comments*, a general field for any remarks.

in Char and NChar (in Unicode format) datatypes, respectively, were created. The Char attributes are in English, while Tamil, a prominent Indian language, was used for the NChar attributes.

These tables were populated with a modified TPC-H generator that embeds integer keys in the part and supplier name attributes, resulting in $\{SuppName, PartName\}$ becoming a candidate key, in the respective tables. After being populated with data, each of the tables held the same information as the original `partsupp` table, but with keys that are in Char or NChar datatypes, respectively. It should be noted that both the tables contain data of the same logical length, but the NChar attributes need more *physical* storage than the Char attributes, due to Unicode format of storage in the NChar attribute. Thus, the performance of a given query on each of these tables is indicative of performance of the operators on each of the datatypes.

Finally, a common table, `partsuppCom`, was created by adjoining all the attributes of the above two individual tables. While the queries on the `partsuppChar` and `partsuppNChar` tables provide differential performance between the datatypes including the I/O costs, queries on the common `partsuppCom` table isolate the differential performance solely due to *in-memory processing*, since the queries need to access the same database blocks irrespective of the datatype on which the query was issued, assuming horizontal partitioning of the table attributes in datablocks. Hence, queries on the common table provide a lower bound on differential performance between the datatypes.

The tables were populated with 4 million records, taking up to 1.2 GB in the common table. Appropriate commands were issued to ensure that the systems computed the table statistics necessary for the optimizer to make more precise estimates of operator costs. Lastly, indexes were created as and when necessary on Char and NChar fields to measure index performance.

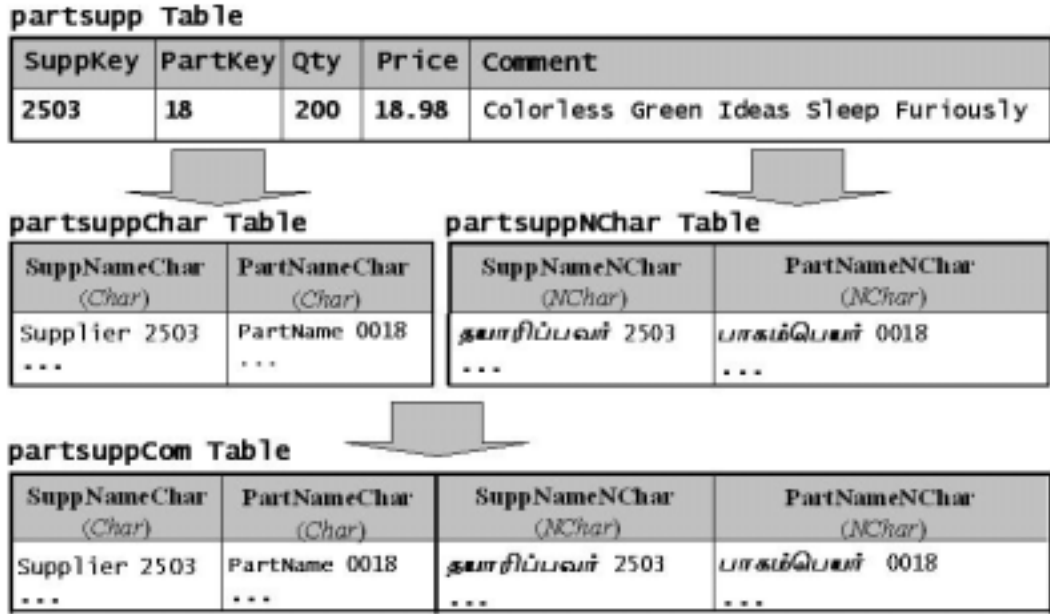


Figure 2.1: Data Setup for Performance Study

2.2.3 Query Workload

The prime objective of our performance study was to measure the performance of basic database operators; hence, simple queries as described below were used.

To model the *Table-scan* operator, a query that scans the appropriate table for retrieving *all the parts supplied by a given manufacturer* was used. To model the performance on Char and NChar data types, the select condition was specified on the appropriate attribute. For example, the table scan query on `partsuppCom` table is as follows:

```
select count(*) from partsuppCom
where {
  suppNameChar } = { 'Supplier 2503' }
   {
  suppNameNChar } = { 'தயாரிப்பவர் 2503' }
```

The *Index-scan* operator performance was measured by running a index-scan query, which returns 20% of the tuples in the table (i.e. 800,000 rows), making the run time large enough to nullify any measurement errors. For example, the index scan query on `partsuppCom` table is as follows (after the indexes were created on appropriate attributes):

```

select count(*) from partsuppCom
where { partNameChar } <= { 'Part 200000' }
      { partNameNChar } <= { 'பாகம்பெயர் 200000' }

```

The *Join* query finds *those suppliers who supply at least two distinct parts*, modeling a *multi-scan* operation. An example join query that self-joins the `partsuppCom` table is given below. The join query was used for measuring performance of the join operator, using one of three different join techniques : *Sort-Merge*, *Hash* or *Nested-Loop*.

```

select count(*) from partsuppCom P1, partsuppCom P2
where P1. { suppNameChar } = P2. { suppNameChar }
         { suppNameNChar } = P2. { suppNameNChar }
and P1. { partNameChar } <> P2. { partNameChar }
        { partNameNChar } <> P2. { partNameNChar }

```

All queries were further simplified by eliminating the post-processing of output data. As the queries return a large number of records (up to 12M records), an aggregate function, `count(*)`, is used to nullify the output time. The query plans obtained from the optimizers confirmed that most of the work done for the queries was executed in the targeted basic relational operators. While the individual query run times were measured as the wall-clock times using database time-stamps, the average runtime from several identical runs was taken as the runtime of a specific query. The number of runs were chosen such that the error in the computed runtime is less than 5% at the 90% confidence interval. Before each query was executed, a large unrelated table was scanned to flush the database buffers and a large unrelated file was read to flush the OS buffers, thereby ensuring a cold start.

2.2.4 Performance Metrics

We measured three different performance metrics, to quantify the differential performance of the database operators and the optimizer while working on multilingual data, as outlined below:

Operator Performance

The operator performance is measured by the run times for the above simple queries that approximate the database operators under default conditions. First, a metric *Multilingual Runtime Overhead* (MRO_{Oper}), is defined as follows:

$$MRO_{Oper} = \frac{T_{NChar}}{T_{Char}}$$

where T_{Char} and T_{NChar} are the run times for the operator on Char and NChar datatypes, respectively. This metric measures the performance overhead of operators working on multilingual data in Unicode with respect to the corresponding performance on default character data in ASCII. A figure close to 1 indicates equitable performance between Char and NChar data types and the magnitude of the metric indicates the relative inefficiency of the database systems in handling multilingual data. We expect no values less than 1 for this metric, since NChar performance cannot be better than that of Char data.

Multilingual Efficiency

Next, an aggregate metric for capturing the relative performance of a given database system, *Multilingual Efficiency* (ME_{DBMS}), is defined as follows:

$$ME_{DBMS} = \frac{G_{Char}}{G_{NChar}}$$

where G_{NChar} is the geometric mean of the run times of operators on NChar data and G_{Char} is the geometric mean of the run times of operators on Char data. We use the ratio of the geometric means² to measure the overall efficiency, in order to ensure that all queries are represented in the final metric, independent of the scales of their run times. While the run times from a complete set of operators will model this metric accurately, we used the run time figures for the following seven operators measured in the study – *Table-Scan*, *Sort*, *Index-Create*, *Index-Scan* and the variations of *Join* operator, to provide an estimate of this efficiency. The ME_{DBMS} measure indicates how well the database handles multilingual character sets with respect to the basic database character set, with a value close to 1 indicating equitable performance across the sets.

²Similar to other database benchmarks, such as Bucky [14].

Optimizer Prediction Accuracy

In addition to the operator run times, we also recorded the optimizer estimates of the cost of each query, to assess the relative accuracy of the optimizer between Char and NChar datatypes. In all database systems, the optimizer estimate for completing an SQL query is either output explicitly in the plan diagram or recorded in the plan table along with the execution plan corresponding to the query. These estimates were retrieved and recorded for each query run. An optimizer metric, *Multilingual Prediction Equity* (MPE_{Oper}) is defined as follows:

$$MPE_{Oper} = \frac{\left(\frac{O_{NChar}}{O_{Char}}\right)}{\left(\frac{T_{NChar}}{T_{Char}}\right)}$$

where O_{Char} and T_{Char} are the optimizer estimate and the actual run time of the query to run on Char datatype, and O_{NChar} and T_{NChar} are the corresponding numbers for the query on NChar datatype.

The MPE metric measures how equitable the optimizer is between the two character datatypes, by comparing the ratio of optimizer prediction to the ratio of actual performance. An MPE value close to 1 indicates equitable prediction accuracy between the data types, while significantly deviations from 1 indicate non-uniform prediction accuracies.

2.3 Performance Results

In this section, we present the results of the experiments that we conducted in the above framework for a set of popular database systems.

2.3.1 Space Overheads

As expected, there was a space overhead of 100% for multilingual data, since each ASCII character that is coded in 1-byte in Char attribute, needs 2-bytes in Unicode format. Curiously, the database systems seem to store even NChar data specified in the UTF-8 format *internally as* UTF-16 (and convert it to UTF-8 format at the interface layer).

This was confirmed by a set of experiments in which the same multilingual data was stored in UTF-16 and UTF-8 formats and a set of queries were run on each; there was no significant difference in the storage size between the two formats and a very slight query performance degradation ($\approx 4\%$) in UTF-8 format.

2.3.2 Separate Table Processing

When the Char and NChar datatypes were created and queried in separate tables, namely, `partsuppChar` and `partsuppNChar`, the *Table-scan* operator was slower on the NChar table by up to 475% from the corresponding Char performance (for 55 characters long Char and NChar attributes), and the *join* operators were slower by up to 275% (for 55 characters long Char and NChar attributes). At first glance, it might be thought that these effects are solely due to the increased storage required by NChar. However, as we will show next, even if all queries are run on a common table, thereby ensuring that the total disk I/O is *identical* for both query sets, there still remain computational factors that come into play resulting in differential performance.

2.3.3 Common Table Processing

Table 2.1 presents the performance of the various operators when the queries were run on the `partsuppCom` common table, forcing the same database blocks to be accessed, irrespective of the datatype on which the queries were issued. This implies that the performance differentials are solely due to *in-memory* processing.

We wish to emphasize that the performance figures in Table 2.1, are not meant to compare the *absolute* performance of database systems in handling multilingual data, but only to highlight their *relative* performance in handling Char and NChar data. Hence we draw attention only to the figures in the MRO_{Oper} and MPE_{Oper} columns of the Table 2.1. It should be noted here that the performance of System *D* was measured on the same test machine, but with a 512 MB main memory, while all other systems were measured with a 256 MB main memory. Hence, the absolute runtimes for System *D*

may be faster, though we expect the relative performance (of NChar *vs.* Char datatypes) of system *D* to be similar, in a machine with less memory.

Table Scan Operator: For the *Table-Scan* operator, very similar performance for Char and NChar should be expected in a database that partitions the table data horizontally, since the same database blocks are accessed for both the queries. While we observe that systems B, C and D do exhibit this behavior, for system A, however, there is a very substantial difference. Such a differential may be due to vertical partitioning of the table data and/or storage of NChar attributes in foreign tables, or due to a very high overhead in multilingual data processing functions.

Sort Operator: The cost of this operator includes the cost for the required initial table scan. The differential sort cost is between 20% and 40% in systems B, C and D, but it is a high 80% in system A.

Index Create Operator: All the database systems were slower in building an index on the NChar attribute by about 20 to 40 percent. Though the slowdown in index creation may not be a source of concern as it is typically an off-line activity, index maintenance, especially in a 24 x 7 operation may well be affected adversely by this slowdown.

Index Scan Operator: The *Index-Scan* performance figures indicate that two of the four systems (specifically, systems A and C) have significant deterioration in NChar performance, and one system has a moderate deterioration of NChar performance (system B). System D has near equal performance in *index-scan*. Since the query is answered by accessing a small number of index blocks, thus incurring only a small I/O cost, the index scan performance is a good indicator of the *absolute* main memory processing efficiency of the databases with respect to multilingual data.

Join Operator: For the join operator, the three standard join implementation techniques were evaluated: *Sort-Merge*, *Hash* and *Nested-Loops*. In different relational

Database System	Query Runtime on Char data (Sec)	Query Runtime on NChar data (Sec)	MRO_{Oper}	MPE_{Oper}
Table Scan Operator				
A	50	136	2.72	0.37
B	116	154	1.33	0.75
C	232	246	1.06	0.94
D	31.2	35.1	1.13	0.89
Sort Operator				
A	78	142	1.81	1.30
B	159	235	1.48	0.68
C	352	431	1.22	1.01
D	221	276	1.24	0.80
Index Create Operator				
A	214	259	1.21	NA
B	457	591	1.25	NA
C	388	538	1.39	NA
D	156	206	1.32	NA
Index Scan Operator				
A	2.73	4.78	1.75	0.38
B	8.51	11.4	1.35	1.55
C	3.33	6.54	1.97	0.31
D	0.78	0.79	1.02	0.99
Join (Sort-Merge) Operator				
A	1156	2198	1.92	0.89
B	841	1304	1.55	1.20
C	852	1143	1.34	0.95
D	909	2459	2.70	0.37
Join (Hash) Operator				
A	4558	11848	2.60	1.26
B	576	778	1.35	0.75
C	754	971	1.29	1.22
D	3068	5521	1.80	0.55
Join (Nested-Loop) Operator				
A	799	823	1.03	0.97
B	323	334	1.03	0.97
C	144	230	1.60	1.16
D	584	791	1.35	0.74

Table 2.1: Multilingual Performance of Operators

database systems, the join operators were invoked by specification of appropriate system parameters or optimization parameters. In Oracle *9i Database Server*, optimizer hints were added to the SQL statements that make the optimizer prefer the desired join operator. In IBM *DB2 Universal Server* the optimization level may be set to different values, to ensure that the desired join operator is chosen by the optimizer. In SQL Server, a desired join operator is chosen by explicit specification in the corresponding SQL statement. In all these cases, the selection of appropriate join operator is verified by examining the final execution plan for the given SQL query. In PostgreSQL *Database Server* the different join operators are forced to be invoked, after explicitly disabling the other join operators by appropriate setting of system parameters. Only a small portion of the original table was used for the *Nested-Loop* implementation, since joining the full table proved to be prohibitively expensive (in the order of days), time-wise.

Table 2.1 shows that there are substantial performance differences between NChar and Char, for all the join implementations. Specifically, the join queries are 35% to 170% slower for *Sort-Merge*, 25% to 160% for *Hash*; the *Nested-Loop* join is most equitable, though it could be as much as 60% slower in System C.

To summarize the above results, we computed the *Multilingual Efficiency* of each of the database systems using the run time figures for the seven database operators – the results are presented in Table 2.2.

Database System	ME_{DBMS}
System A	0.57
System B	0.76
System C	0.70
System D	0.69

Table 2.2: **Multilingual Efficiency**

As can be clearly seen in Tables 2.1 and 2.2, all the database systems are inequitable

with respect to multilingual data, and no single system (or a set of systems) has performed badly consistently in all the operators. Also, there is a wide variation in relative performance, indicated by the ME values ranging from 0.57 to 0.76: for example, System A is slower by nearly 75% in handling multilingual data.

Some database architectures organize table data in a *vertically* partitioned manner, to make the query processing cache-friendly. For example, the *Sybase*[120] and the *MonetDB* [90] database systems partition the table data vertically. In such architectures, we expect absolute performance of our tested queries on both **Char** and **NChar** datatypes to be better, when compared to those in the horizontally partitioned database systems. Also, we expect the relative performance of database operators on multilingual character set over the default character set, to be similar to that of storing the **Char** and **NChar** data in separate tables. Conversely, should the output of the query span multiple attributes or if the selection condition involves multiple attributes, then we expect the horizontally partitioned database systems to perform better than the vertically partitioned ones. However, such hypothesis needs to be verified with experiments, which we hope to take up in our future work.

2.3.4 Optimizer Prediction Accuracy

The accuracy of the optimizer is an important factor in database system performance, since errors in estimation could lead to a huge performance degradation as grossly inefficient plans could be chosen. Table 2.1 also provides the optimizer metric, MPE_{Oper} , for each of the database operators. No MPE_{Oper} figure is calculated for *Index-Create* operator, since it is a DDL statement that requires no optimization.

For most of the operators, the optimizer predictions were inequitable (indicated by the MPE figures much different from 1). The accuracies of the *Table-Scan*, *Sort*, *Index-Scan*, *Sort-Merge join* and *Hash join* estimates on **NChar** are different by up to 60%, 30%, 60%, 20% and 25%, with respect to the corresponding **Char** estimates. In addition, we find that in some cases, the optimizers are impervious to the differences between the datatypes; they estimate the operators to perform equally, though the actual run times vary by

more than 100%. Such inequities in prediction may indicate a non-uniform cost model between Char and NChar datatypes. In conjunction with the large slowdowns in query performance, such mis-estimation may have serious impact on database performance, due to selections of inefficient plans for complex queries.

2.4 Performance Analysis

The results from the previous section indicate that all the database systems were slow in processing data in multilingual character set, compared with their performance in handling default character set. In this section, we conduct a series of experiments to understand the trend of, and the reasons for, this multilingual query processing overhead and to pinpoint the sources of inefficiency. The database system that exhibited the most iniquitous performance, namely, system *A*, was chosen for this study.

2.4.1 Slowdown *vis-a-vis* String Length

As a first step towards calibrating the performance with respect to multilingual data, we studied the effect of the string length on the differential performance, MRO_{Oper} . Specifically, the table scan and a set of join operator queries were run on the common table with Char and NChar attributes of equal logical length, varying from 15 to 95 characters long. Note that, as mentioned before, though the strings lengths are equal, the NChar strings need twice as many bytes as Char strings for storage. The experiments were conducted on the common table to nullify the effects of the disk traffic.

The results for this experiment are shown in Figure 2.2, which captures how the NChar performance slowdown with respect to Char varies with the length of a text string. The table scan slowdown is very high at small string lengths but decreases with increasing length and asymptotically settles at about 125%. At small string lengths, the large differential performance in NChar data indicates very high fixed cost (such as function call overheads) in NChar data over Char data. As the string length increases, the variable

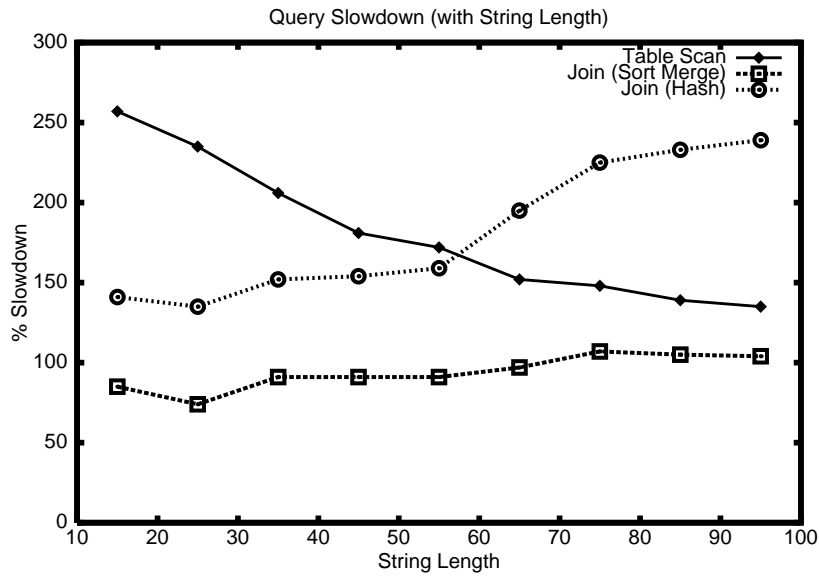


Figure 2.2: Query Slowdown with String Size

cost of string comparison becomes significant, dominating the function overheads, and hence the differential performance reduces.

The *Hash* join technique exhibits a fairly steady trend of increasing differential performance with string length, indicating that the operator is affected more by the string processing overheads in *NChar* than those in *Char*. *Sort-Merge*, on the other hand, exhibits a fairly constant slowdown, indicating that the slowdown is balanced between string processing and disk access.

Overall, one can observe that the slowdowns exist for all operators and at all string lengths, though it is more serious for short strings for scan operators and for long strings for join operators. It should be noted that the observed slowdowns are significant for the chosen queries, given that the runtimes are in the order of tens of seconds for table scan, and in the order of hundreds of seconds for join operators.

2.4.2 Components of the Slowdown

We took the default size of character attribute in TPC-H database (55 characters) and conducted a second set of experiments to determine the specific reasons for the slowdown. In database systems, typically the operators are implemented as common functions, but

invoked with different type parameters. Hence it is reasonable to assume that the same code path will be taken for each of the above queries irrespective of the datatypes on which the queries were issued, as long as the plans are the same.

Under the above assumption, the slowdown between Char and NChar datatypes may be attributable to the following three components:

$$\Delta T = \Delta T_{I/O} + \Delta T_{Type} + \Delta T_{StringProcessing}$$

where $\Delta T_{I/O}$ is the differential cost due to the increased disk access for NChar storage over Char storage, ΔT_{Type} is the differential cost in handling different datatype (NChar vs. Char) and $\Delta T_{StringProcessing}$ is the difference in the cost due to processing of the string – due to both the function call overheads invoked with different byte lengths and the actual comparison of different byte strings. Of the three, the first factor corresponds to slowdown due to increased disk access and the next two correspond to that due to in-memory computation.

The slowdown due to the increased disk access, namely $\Delta T_{I/O}$, is *zero*, as all the performances were observed by running the queries on `partsuppCom` table, thereby forcing the same disk blocks to be accessed.

Next, to isolate the cost due to the datatype, namely ΔT_{Type} , we created the `partsuppCom` table with Char attribute of size 110 and NChar attribute of size 55, forcing each attribute to store the attribute values in equal number of bytes. The scan and join queries were run on each datatype as before to find any variation in performance which can be attributed to datatype specific processing. The NChar queries are slower by about 10% indicating that ΔT_{Type} is small, but not insignificant.

Finally, to isolate the cost due to the size of the data, namely $\Delta T_{StringProcessing}$, we created a set of tables with NChar attributes replaced by Char attributes, but with different sizes ranging from 55 characters to 165 characters, corresponding to scale factors between 1 and 3. The keys were embedded at the end of the character strings forcing each comparison to scan the entire length of the string to determine [in]equality. The slowdowns of *Table-scan* and *Join* queries on scaled up Char attribute, relative to normal

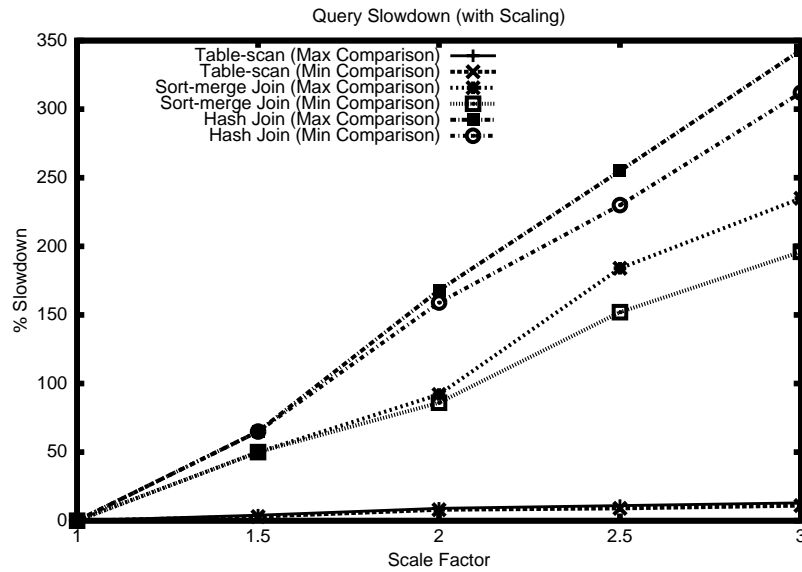


Figure 2.3: Query Slowdown with Scaling

Char attribute, are shown in Figure 2.3, as lines marked *Max Comparison*. The relative slowdown in a single-scan *Table-Scan* operator is very low with a maximum slowdown of 10% for a scale up factor of 3. Such negligible relative slowdown indicates that the overall cost of the operator is dominated by disk I/O, which is equal for the two attributes due to the common table design. The performance of the multi-scan *Sort-Merge* and *Hash* join operators, however, show that the relative slowdowns increase substantially with scale-up, indicating that the join and string processing costs dominate disk I/O cost for long strings. Also, the slowdowns for the *Sort-Merge* and *Hash* joins, for a scale up factor of 2 in Figure 2.3, match closely with those reported earlier, in Table 2.1, for NChar that takes twice as much space as Char.

The I/O dominated Table-Scan operator exhibits no differential performance in the table with both character attributes, but a substantial difference in the table with Char and NChar attributes. This behaviour indicates that the NChar attribute may be stored in a foreign table, in database system A. Also, the slowdowns for the *Sort-Merge* and the *Hash* joins in Figure 2.3, for a scale up factor of 2, match closely with those reported earlier, in Table 2.1, for NChar that takes twice the space as Char. The behaviour of the processing-dominated joins confirm the above observation of a foreign table for NChar

attributes.

We also isolated the specific costs due to actual string comparison itself, by re-running the experiments with `Char` data strings that have integer keys embedded in the *beginning* of the string, thus causing nearly 95% of the comparisons to fail in the first few bytes of data itself. The associated performance graphs are marked in Figure 2.3 as *Min Comparison*. The difference between the operator performance for maximal and minimal comparison indicates the differential cost due to byte comparison itself. We found this cost to be negligible for the *Table-scan* operator, confirming our initial observation that disk I/O time dominates the string processing time. For *Join* operators, these costs are not negligible, and become significant for long strings (up to 15%).

As a result of the above experiments we could effectively isolate the main reasons for the differential performance between `Char` and `NChar` datatypes in system *A* as the following: primarily, the differential costs associated with the size of the data (> 80%) and, secondarily, that due to the datatype. Though we have established that the comparison of strings itself plays a role in the slowdown, we ignore this data-dependent slowdown for efficiency improvements. Hence, to improve the performance of `NChar` it is imperative that methods to reduce the storage space required by the multilingual character sets be found.

2.5 The Cuniform Storage Format

In the previous section, the storage size of the multilingual character sets was identified as the prime reason for the inequitable multilingual performance of the database systems. In this section, we propose a simple, split storage representation for storing Unicode data, to reduce the multilingual storage space required. Subsequently, we present the performance profiles of the database systems working on this storage format.

Our proposal for the split-representation of Unicode strings stems from the following two observations:

Character Block Information: Unicode characters are organized in Character Blocks

(variable in length, corresponding to the size of the script used in that language). Character block information forms a part of the character code in Unicode characters. Since most scripts in Unicode have less than 256 characters, for these scripts about half of the Unicode code is used for representing the character block information.

Language of an Attribute Value: It is reasonable to assume that in a multilingual environment, a data item stored in NChar field is likely to have all the characters from the same script. Hence, storing the character block information for each character would be wasteful of resources in the database context.

Based on the above two observations, we propose a new internal split representation of Unicode called Cuniform (Compressed UNicode FORMat), which splits each Unicode string into two pieces. The first piece stores the information about the character block corresponding to the script from which the characters of the string occur. This information may be the starting code of that character block corresponding to the script or a *Script Identifier* that may be translated to the previous one. The second piece stores the offsets of each character in the original Unicode string, in the character block corresponding to the specific script. We term such splitting of a Unicode string into a pair of Cuniform strings as “skinning”. When the string contains characters from multiple code blocks, skinning is not possible, and hence the original string is stored without any modification. Skinning allows the code block information to be stored as a meta-data once for the entire string, effectively reducing the storage of Unicode strings, yet ensuring that the original string is reproducible by assembling the two pieces. The proposed format is trivially convertible to the Unicode format, since our primary design goal is to find a solution within, and not outside, the framework of Unicode.

2.5.1 Sample Unicode and Cuniform Strings

Examples of Unicode to Cuniform transformation are shown in Figure 2.4. The first two strings (in English and Tamil) have only characters from a single character block each

Original/Multilingual/Unicode Strings ...		
Data	Unicode Strings	
Narayan	00.E4.00.16.00.27.00.16.00.97.00.16.00.E6	
நாராயணர்	0B.A8.0B.BE.0B.B0.0B.BE.0B.AF.0B.A9.0B.CD	
RK ಸಾರಾಂಯನ್	00.27.00.EF.0C.A8.0C.BE.0C.B0.0C.BE.0C.AF.0C.A3.0C.CD	
寺井正博	5B.FA.4E.95.6B.63.53.5A	
...after Skinning into Cuniform Strings...		
Data	SID	Offsets
Narayan	00	E4.16.27.16.97.16.E6
நாராயணர்	0B	A8.BE.B0.BE.AF.A9.CD
RK ಸಾರಾಂಯನ್	NULL	00.27.00.EF.0C.A8.0C.BE.0C.B0.0C.BE.0C.AF.0C.A3.0C.CD
寺井正博	NULL	5B.FA.4E.95.6B.63.53.5A

Figure 2.4: **Skinning of Unicode Strings**

and hence may be skinned, setting the script identifier (*SID*) as the respective code block identifier, and the skinned string as the string of offsets into the code block. The sizes of the Cuniform strings thus reduce to about half that of the corresponding Unicode strings. The third string which has mixed scripts is not skinnable, but only a small fraction of strings is expected to have such a characteristic. The fourth string in Kanji may be skinned, but since each of the offsets would need about 2 bytes due to the large size of the language repertoire, it may not provide any saving over the storage needed for the Unicode format itself. Hence for such languages that have a large repertoire, the Unicode strings are stored *as-is*.

2.5.2 Limitations of Cuniform Format

While there are advantages to the Cuniform representation as discussed above, there are some limitations as well.

Firstly, if the database stores primarily English (or, Latin based) data, then usage of Cuniform may add a slight storage and query processing overhead, due to the split nature of the storage. In such cases, the usage of ISO:8859 format will be more efficient.

Secondly, if each of the data items stored in the Cuniform attribute is a *mix* of characters from different code blocks, the space compression and the associated performance

improvements may not materialize. As Unicode has allocated special blocks for common characters (e.g., Math symbols), mix of characters from different blocks may occur frequently in some domains.

Finally, languages with character block size more than 256 may not be able to gain any performance benefits by the Cuniform format. The storage space required to store strings from such character blocks may be reduced by storing the offsets; typically, each offset may require more than one byte³. However, due to the non-byte aligned nature of the offsets, the performance of any substring operation will be more expensive and the performance benefits of Cuniform storage may not materialize.

2.6 Cuniform Performance

Since we lacked access to the source code of the database system *A*, a prototype of the Cuniform representation was implemented using an *outside the server* approach: Each NChar attribute was converted into a pair of attributes – *Cuni_{sid}* and *Cuni_{string}*, where *Cuni_{sid}* is the *Script Identifier* that stores the starting code of the character block, and *Cuni_{string}* stores the offsets of each character in the original Unicode string into the character block corresponding to the script of the Unicode string. During data input, the common character block of the Unicode string was identified and stored in *Cuni_{sid}* and the offsets of each character in the input string was stored in *Cuni_{string}*. If a mix of code blocks existed in the input string, then the input string was stored with no modification in *Cuni_{string}* and a Mixed (or Null) was inserted into *Cuni_{sid}*. For output, the character block information from *Cuni_{sid}* was merged *byte-by-byte* with *Cuni_{string}*, reconstructing the original Unicode string. Fortunately, all the database operations may be executed directly on the Cuniform strings without any explicit conversion to Unicode strings.

In the processing side, all SQL queries need to be recast to handle the split image of Cuniform attributes. While explicit representation of Unicode strings in NChar attributes in SELECT, INSERT and UPDATE statements are handled easily by skinning them into

³They need a bit-string of size that is logarithmic in size of the repertoire of the language.

Cuniform format, the predicates involving NChar attributes in WHERE clause need to be recast into more complex predicates. An equality predicate between NChar attributes was replaced with a conjunction of equalities on both *Cuni_{sid}* and *Cuni_{string}* components of the respective attributes. Similarly, an inequality predicate was replaced by a disjunction of inequalities on *Cuni_{sid}* and *Cuni_{string}* components of the respective attributes. Correlated sub-queries were replaced with the conjunction or disjunction of the pair of Cuniform attributes, as appropriate. In summary, all operations on the Cuniform attributes were executed on Cuniform pair of attributes, with no conversion to Unicode, except for the output.

2.6.1 Performance of Cuniform Storage

To measure the performance of the operators with the multilingual strings stored in the Cuniform format, we used the following procedure: The common `partsuppCom` table that was used for the experiments detailed in Section 2.2 was augmented with *Part* and *Supplier* names in Cuniform format. All the NChar values are assumed to be from a distinct multilingual script and hence each value was skinned into Cuniform format. All the previous queries, appropriately modified for Cuniform datatype, were run on this new table and the performance of operators measured. The MRO_{Oper} for the operators, running on each of the datatypes in the `partsuppCom` table are provided in Table 2.3⁴. Also, no MPE_{Oper} figures were reported, as the optimizer prediction for operators working on Cuniform datatype is meaningless in an *outside-the-server* implementation.

As can be seen from Table 2.3, the performance of the operators on multilingual data in the Cuniform format is vastly better than the corresponding performance in the Unicode format, except for *Index-Scan*. The performance of *Table-Scan* on Cuniform is almost identical to Char datatype and the performance of join operators are only marginally slower than that on Char datatype. However, the performance of *Index-Scan*

⁴It should be noted that the figures are slightly different from those presented in Table 2.1, since the new table has two additional Cuniform attributes and hence incurs additional disk I/O. However, the MRO_{Oper} is found to be almost the same as in Table 2.1.

DBMS Operator	Query Runtime on Char (Sec)	Query Runtime on NChar (Sec)	Query Runtime on Cuniform (Sec)	MRO_{Oper} on NChar (%)	MRO_{Oper} on Cuniform (%)
Table Scan	52.9	135	55.5	1.56	1.05
Sort	81.1	143.5	86.1	1.77	1.06
Index Scan	2.89	5.46	5.60	1.88	1.99
Join (Sort-Merge)	1188	2371	1370	1.99	1.15
Join (Hash)	4575	12534	5591	2.74	1.22
Join (Nested-Loops)	805	834	827	1.04	1.03

Table 2.3: Multilingual Performance of Operators on Cuniform

on Cuniform attribute is substantially slower than the corresponding Unicode datatype, primarily due to the additional overheads of the composite index on a pair of Cuniform attributes. Significantly, the Cuniform representation incurred only a negligible space overhead (approximately 2%), a tremendous improvement over NChar’s 100%.

Finally, we computed a new *Multilingual Efficiency* for system *A* using the Cuniform performance numbers, which evaluated to 0.83. Compared to the *ME* figure of 0.57 presented in Table 2.2, the NChar stored using Cuniform improves the multilingual performance of *A* substantially, bringing it to within 20% of the performance on the default ISO:8859 character set.

In summary, Cuniform shows that multilingual data may be stored and manipulated almost as efficiently as the default character data in ISO:8859 by using an appropriate internal storage format. Further, the Cuniform datatype supports substring operations efficiently. Specifically, when a substring of a Cuniform string is needed, a normal substring function call may be invoked on the string that stores the offsets and the resulting substring may be converted into a Unicode string efficiently, by appending appropriate code block information corresponding to the script identifier to every character in the result. If the script identifier is *NULL*, then the substring requires no modification as no skinning was done to the original Unicode string. Thus, Cuniform retains random access of its substrings, aiding efficient database query processing.

2.6.2 Potential for Further Performance Improvement

An important by-product of skinning Unicode strings into Cuniform strings is the explicit availability of character block information of the multilingual attributes⁵. This additional piece of information may be used for partitioning the multilingual data: either as a *query predicate* to improve the selectivity of the query or in partitioning the table data into subtables. Such partitioning of data would make the operators proportionally more efficient, as they need to process only on a subset of the tuples. Suppose the multilingual table has data in n different languages, then the table may be horizontally split into n tables, each storing data from a single language. For selection operation, only that subtable that contains the records in the language of the query string needs to be examined, and for join operation, only that subtable that contains the records in the language of the outer loop value needs to be accessed. Hence, a substantial improvement in the runtime may be achieved by appropriate partitioning of the table data over the languages. In our experiments, partitioning the records in the *partsupp* table assuming a uniform distribution of data over 5 different languages, we observed the performance of the operators on NChar in Cuniform format to be faster than even Unicode by about 70% for *Table-scan* operator and upto 40% for *Join* operators.

In such an environment, it may be advantageous to store information in multilingual scripts, rather than in a single script.

2.7 Related Research

To the best of our knowledge, performance evaluation of relational database systems with respect to multilingual data or their differential performance had not been published in the database research literature.

The following studies address, partially, the compression and efficient storage of Unicode data: The *Standard Compression Scheme for Unicode* (SCSU) for Unicode data is reported to have compression characteristics similar to that of ISO:8859 data, in [133].

⁵This language identifier is used in Chapter 5 for designing a new multilingual datatype, Uniform.

SCSU uses a dynamically positioned window covering 128 consecutive characters for compression. This scheme is intended primarily for medium and large text strings, and is not well suited for attribute level strings. A study of different compression techniques (such as *bzip*, *gzip*, *pksip*, *Huffman*, etc.) on Unicode data is presented in [6] and [40], where the authors indicate that all these techniques produce similar compression ratios for basic Unicode files, for a *given* Unicode encoding format. An interesting proposal that they offer is that it may be advantageous, under certain conditions, to transcode the base Unicode document (to among one of UTF-8, UTF-16 etc.), and then use a compression technique, to achieve better performance. However, all their experimentation is done on large files and hence, not directly relevant for efficient storage of attribute-level data.

Multicode [92] overcomes some of the storage space related issues in Unicode, by allowing efficient representation for characters of a language and by allowing special switch characters to mix characters from a different language in the same multilingual string. While this scheme optimizes space, the substring searches become more expensive due to the need for decompression of the entire string; essentially, no random access of substrings is possible. Similarly, the *Binary Ordered Compression for Unicode* (BOCU) [108] adapts well for compressing small Unicode strings, but is not suited in environments where random access of substrings is necessary. In database environments random access of substrings is essential for supporting normal text operations. Finally, the efficient multilingual framework described in [139] makes the administration of multilingual resources in a multilingual database environment more efficient, but not the query performance of multilingual data.

2.8 Conclusions on Multilingual Performance Study

In this chapter, we focused on identifying the differential performance of a suite of popular database management systems while handling multilingual data, and provided our solutions to overcome this differential performance.

First, an experimental framework to measure the storage and query processing efficiency of basic database operators on multilingual data was described and the performance of a suite of database systems in this framework was presented. Our experimental results indicate that multilingual data stored in the popular Unicode encoding suffers from a serious space overhead and a corresponding query processing overhead, in all database systems. The query performance overhead was significant, even when only the *in-memory* processing is considered. The primary factor for the inefficiency was identified as the storage size of multilingual data.

Cuniform, a split internal storage format that is trivially convertible to Unicode, was proposed to overcome such performance overheads. Multilingual data in the Cuniform format exhibited marginal space overhead and correspondingly small query overhead, improving significantly the query performance over that in the Unicode format. In addition, performance of operators on Cuniform could further be improved in highly multilingual environments by partitioning of data using the explicit script handle available in it.

Chapter 3

Multilingual Names Matching

3.1 Overview of the Chapter

In this chapter, we outline the implementation of the multilingual names join operator that was proposed in Chapter 1. First, some background needed for this chapter is provided; next, the `MLNameJoin` functionality is defined formally and our strategy for the implementation of the functionality, by transforming the matches from the *textual space* to the *phonetic space*, is detailed. Finally, while a basic implementation using UDFs on database systems was too slow for practical use, we show that with specialized indexing techniques, the performance may be improved substantially to a level that appears commensurate with requirements for practical deployment.

3.2 Background Information

In this section, some background needed to implement our multilingual names matching methodology, is presented.

3.2.1 Pseudo-Phonetic Matching Function

The currently popular algorithm for pseudo-phonetic matching of English text strings in database systems is the *Soundex* [73] algorithm. This simple algorithm defines groups of

similar sounding vowels and consonants and converts a given text string into a string of alpha-numeric characters (the first being an alphabet, referred to as *Soundex-key*, and the remaining being a numeric between 0 and 6 corresponding to unique consonants in the string). The transformed string is truncated after 4 characters and the resulting string is used as a key for the original English name. The details of the algorithm are shown in Figure 3.1.

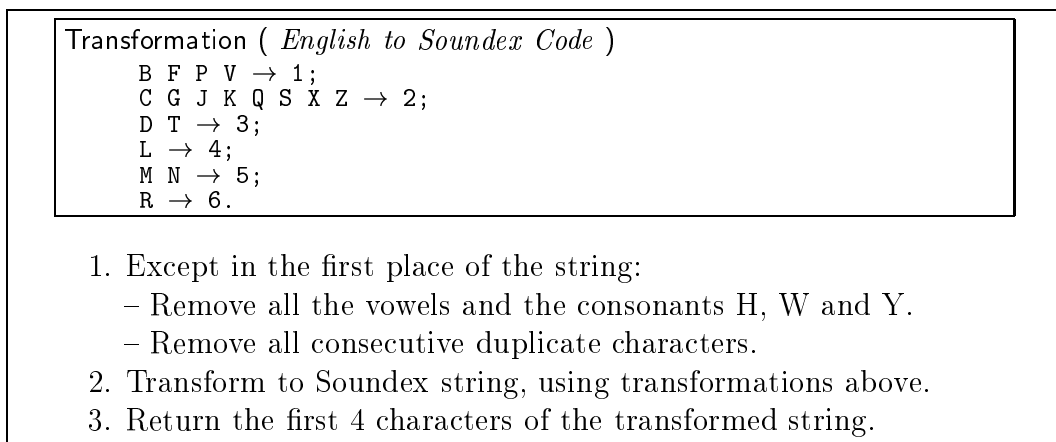


Figure 3.1: *Soundex* **Algorithm**

For example, the Soundex-key for both the English strings, **Interpid** and **International**, is **I536**. The English word **India** has a soundex-key of **I53** and the word **Enterprise** has a soundex-key of **E536**; both these keys are at an edit-distance of 1 from the original key, namely **I536**. Hence one can see that the value of such keys as a similarity measure is limited. However, they may be used as a filter for efficiently narrowing down *possible* proximity, which we explore later.

Other algorithms, such as *Phonix* [44] and *Metaphone* [77] are similar in principle to *Soundex*, but they employ, in addition, English spelling and pronunciation rules. It is clear that the *Soundex* algorithm and its variations were devised for English, and do not scale across languages.

3.2.2 Approximate Matching

Approximate matching techniques are used for matching strings that are *close to each other* in a common alphabet, but which are not exactly equal. A common use for approximate matching techniques is in Bioinformatics for genomic comparisons and in Information Retrieval for compensating typographic errors. Several frameworks exist to capture the notion of *closeness* of strings. A popular example is the *Edit Distance* metric [54], which is used in *Approximate String Matching*, as given in the following definitions:

Definition 3.1 [Edit Distance]: The *edit distance* between two strings in a common alphabet Σ , is the minimum number of edit operations (i.e., insertions, deletions and substitutions) that are needed to transform one string to the other.

Definition 3.2 [Approximate String Matching]: Two strings are considered to *match approximately*, if the *edit distance* between them is less than a user specified threshold¹.

To compute the edit distance between two given strings S_i and S_j , a standard *Dynamic Programming* algorithm [112] may be used. This algorithm sets up a matrix of size $(|S_i| \times |S_j|)$ and computes the transformation of S_i to S_j . Though this algorithm is not the most efficient, it is preferred for its flexibility and adaptability in modeling a wide variety of other distance measures[96].

3.2.3 Q-Grams

In situations where approximate matching of strings is applicable, *Q-Grams* have been successfully employed to narrow down the search space effectively. In this section, we briefly sketch the concepts of q-grams, and refer to [54] for details.

Let σ be a string of size n in a given alphabet Σ . $\sigma[i, j]$, $1 \leq i \leq j \leq n$, denotes a substring starting at position i and ending at position j of σ .

¹Usually, the threshold is specified as a symmetric function of the input strings, in order to make the approximate matching symmetric.

Definition 3.3 [Q-Gram]: Given a string σ and q , a substring of σ of length q , that is, $\sigma[i, i + q - 1]$, is called a q -gram of σ .

The q -grams of σ consists of all q -length substrings of σ , and is obtained by sliding a window of size q over the string.

Definition 3.4 [Positional Q-Gram]: The pair $(i, \sigma[i, i + q - 1])$ is called the *positional q -gram*, where i is the starting position of the q -gram in σ .

Usually, the q -gram matching techniques use an augmented string σ_{aug} , where $(q - 1)$ start symbols (say, \triangleleft) are pre-pended to σ and $(q - 1)$ end symbols (say, \triangleright) are appended to σ , where \triangleleft and \triangleright are not part of the original alphabet, Σ . Note that for a given string σ , there are $(|\sigma| + q - 1)$ q -grams. For example, a string **LEXEQUAL** will have the following positional q -grams: $\{(1, \triangleleft\text{L}), (2, \triangleleft\text{LE}), (3, \text{LEX}), (4, \text{EXE}), (5, \text{XEQ}), (6, \text{EQU}), (7, \text{QUA}), (8, \text{UAL}), (9, \text{AL}\triangleright), (10, \text{L}\triangleright\triangleright)\}$. The q -gram can be implemented as an auxiliary table, in the relational databases, in $(n * \sigma_{ave} * (q + C))$ space, where n is the number of multilingual strings in the original table, σ_{ave} is the average length of the strings and C is the overhead of storing each of the q -grams.

The intuition behind using q -grams is that strings that match approximately will share a large number of q -grams [52]; hence, an approximate match may be replaced by more efficient exact matching of the q -grams; further, the database matching functionality may be used efficiently, since any query using the standard query features may leverage on the well-developed optimizer of relational systems.

3.3 Multilingual Names Matching Implementation

Multilingual names matching was defined as matching of the same names across multiple languages, as shown in Figure 1.3. Such multiscrypt matching functionality is applicable to many user domains, especially with regard to *e-Commerce* and *e-Governance* applications, web search engines, digital libraries and multilingual data warehouses. As expounded in Section 1.5.3, we assume that when a name is queried for, the primary intention of the user is in retrieving all names that match *aurally*, in the specified set

of target languages. Hence, the matching is restricted to attributes that contain *proper names* (such as attributes containing names of *individuals, corporations, cities, etc.*), which are assumed not to have any semantic value other than their vocalizations.

3.3.1 MLNameJoin Implementation Details

In this section, the details of our strategy to implement the multilingual names matching operator – MLNameJoin – are provided.

Let L_i be a natural language with an alphabet Σ_i . Let s_i in language L_i be a string composed of characters from Σ_i , and let $\mathcal{S}_{\mathcal{I}}$ be set of all such s_i . Then, $\mathcal{S} = \cup_{\mathcal{I}} \mathcal{S}_{\mathcal{I}}$, represent the set of all name strings in a given set of languages. Similar to the multilingual name strings, the phoneme strings are assumed to be encoded in the IPA [60] alphabet, namely, Σ_{IPA} . Further, it is assumed that every natural language string can be transformed to a phonetic string in the IPA alphabet (in line with the phonetic conventions of the language). A transformation, $\mathcal{T}_{\mathcal{I}}$, between a given language string s_i and a corresponding phonemic string p_i , is represented by $\mathcal{T}_{\mathcal{I}} : \mathcal{S}_{\mathcal{I}} \rightarrow \mathcal{S}_{IPA}^2$, where $s_i \in \mathcal{S}_{\mathcal{I}}$ and $p_i \in \mathcal{S}_{IPA}$. The union of such transformation functions \mathcal{T} ($= \cup_i \mathcal{T}_{\mathcal{I}}$) in a set of desired languages, represented by $\mathcal{T} : \mathcal{S} \rightarrow \mathcal{S}_{IPA}$, is assumed to be given as an input to the query processing engine. Given the above, *phonetic equality* is defined as follows:

Definition 3.5 [Phonetic Equality]: Two strings $s_i \in \mathcal{S}_{\mathcal{I}}$ and $s_j \in \mathcal{S}_{\mathcal{J}}$ are *phonetically equal*, if $p_i = p_j$, where $p_i = \mathcal{T}(s_i)$ and $p_j = \mathcal{T}(s_j)$.

Example 3.1: Given that {“Nehru” in English, “நெரு” in Tamil and “नेहरु” in Hindi} have corresponding phonemic representations {“næhru”, “næru” and “næhru”}, only the English “Nehru” and the Hindi “नेहरु” are phonetically equal. \diamond

Though all the names in the above set refer to the same name written in different languages, it is almost impossible to exactly match their respective phoneme strings, since the sets of phonemes used by different languages are seldom identical and the rules for conversion of a textual string to a phoneme string may differ, due to linguistic and

²Such transformation $\mathcal{T}_{\mathcal{I}}$ is coded as linguistic rules in the language specific TTP engines.

cultural differences. Hence in the phonetic domain, *phonetic closeness*, a weaker notion of equality, is defined as follows:

Definition 3.6 [Phonetic Closeness]: Two strings s_i and s_j are *phonetically close* if $\{d(p_i, p_j) \leq t\}$, where p_x is the phonemic representation of s_x ($= \mathcal{T}(s_x)$), $d(x, y)$ is the edit distance function as per Definition 3.1, and t is a user defined parameter for match.

Example 3.2: With the sample strings as in Example 3.1 and assuming the string “*Nero*” has a corresponding phonemic string “nerou”, “ $\mathbb{C}\mathbb{N}\mathbb{U}$ ” is at a phonetic distance of 1 and “*Nero*” is at a distance of 2, from the English “*Nehru*”. They may be *phonetically close* depending on the user specified value of t . \diamond

We propose to implement the MLNameJoin functionality, using *phonetic closeness*, as follows:

Definition 3.7 [MLNameJoin Matching]: $\{s_i \text{ MLNameJoin } s_j\} \iff \{d(p_i, p_j) \leq t\}$, where t is a parameter that is specified for a specific domain or application.

Definition 3.7 provides the basis for multilingual names matching operator for comparing phoneme strings corresponding to the multilingual text strings. The quality of match is determined by the user-specified threshold parameter, t , which is usually defined symmetrically, as a function of the two phonemic strings p_i and p_j . Traditionally, this is specified as a fraction (in the range $[0, 1]$) of the length of the smaller of the two strings being compared.

We wish to emphasize that while our implementation methodology works well (as will be shown in subsequent sections), it may introduce significant ($\approx 15\%$ in our experiments) *false-positives* in the result set. Depending on the setting for the threshold parameter, the matching may also have *false-negatives*; however, at the expense of precision, the false-negatives may be nullified, by specifying a higher value for the threshold parameter.

3.3.2 Linguistic Issues

We hasten to add that phonetic matching of multilingual names is, not surprisingly given the diversity of natural languages, fraught with a variety of linguistic pitfalls, accentuated

by the attribute level processing in the database context. While simple lexicographic variations in names are handled in our methodology, issues such as language-dependent vocalizations and context-dependent vocalizations, discussed below, appear harder to resolve, and are left as future extensions to the current work.

Language-dependent Vocalizations A single text string (say, *Jesus*) could be different phonetically in different languages (“*Jesus*” in English and “*Hesus*” in Spanish). So, it is not clear when a match is being looked for, which vocalization(s) should be used. One plausible solution is to take the vocalization that is appropriate to the language in which the base data is present. But, automatic language identification is not a straightforward issue, as many languages are not uniquely identified by their associated Unicode character-blocks. With a large corpus of data, IR and NLP techniques may perhaps be employed to make this identification.

Context-dependent Vocalizations In some languages (especially, Indic), the vocalization of a set of characters is dependent on the surrounding context. For example, consider the Hindi name *Rama*. It may have different vocalizations depending on the gender of the person (pronounced as *Rāmā* for males and *Ramā* for females). While it is possible to make the appropriate associations in a running text based on the context, it is nearly impossible while processing the database attributes, which are stored at an atomic value level. Specifically, a noun occurring in isolation may have no cues to its pronunciation, as it does not carry the contextual information needed for proper vocalization.

3.3.3 Existing Database Support for Implementation

While a survey of general multilingual support by current database systems was outlined in Section 1.2, in this section, the support provided by database systems, specifically for implementing multilingual names matching functionality, is provided.

Unicode, the default multilingual storage standard supported in all database systems, specifies the semantics of comparison of a pair of multilingual strings at three different

levels [26]: using *base characters* (plain vanilla lexicographic matching of the strings), *case* (where the case of a character is ignored), or *diacritical marks* (where the diacritical marks are ignored). For example, `Miller` and `miller` are matched successfully in *Level 2* and `Müller` and `Muller` are matched successfully in *Level 3*, but both the matchings fail in *Level 1*. More importantly, such matching levels are applicable only between strings in languages that share a common script. In Unicode, the comparison of multilingual strings across scripts is considered only as a *binary* comparison. Hence, no meaningful comparison is possible across scripts. Also, the SQL:1999 standard [59, 84] specifies that any comparison across collations is *binary*.

To the best of our knowledge, none of the commercial and open-source database systems currently support multilingual string matching. Further, their support of even other techniques to implement our proposal of multilingual names matching is limited, as given below:

Phonetic Matching Most database systems allow matching text strings using pseudo-phonetic *Soundex* algorithm [73], primarily for English text strings.

Regular Expression (LIKE) Matching The regular expression matching feature – the LIKE predicate that is available in all database systems – are designed for regular expressions, but cannot be used for approximate matching in *metric* space.

Multiscript Comparison and Indexing All systems have pre-defined collation sequences for the supported languages. While comparison within a collation has normal semantics, comparison across collations is binary; that is, the sort order is same as that of the binary strings corresponding to the text strings. Consequently, any index built on multiscript strings is based on the binary sort order of the multilingual text strings.

Approximate Matching Approximate matching is not supported by any of the commercial or open-source databases. However, most database systems support *User-defined Functions* (UDF) that may be used to add new functionality to the server. The major drawbacks with UDF implementations are the overheads in making UDF

calls and the inability of queries using UDFs to leverage on the well-tuned relational optimizer, as the UDFs are not costed.

In summary, while current databases are effective and efficient for processing monolingual data (that is, within a collation sequence), they do not support processing multilingual strings across languages in an integrated manner.

3.4 MLNameJoin Matching Algorithm

The `MLNameJoin` algorithm for matching multilingual names strings is provided in this section, following the strategy outlined in Section 1.5.3. In essence, the multilingual name strings are converted into equivalent phonemic strings using calls to the TTP engines and compared using approximate matching techniques.

The algorithm is as shown in Figure 3.2. The `MLNameJoin` operator accepts two multilingual text strings and a match threshold value as input. In addition, the language identifiers were also input, explicitly³. The strings are first transformed to their equivalent phonemic strings using the `PhoneticTransform` function that takes a multilingual string in a given language and returns its phonemic representation in IPA alphabet (Lines 3 and 4), by calls to standard TTP systems of the appropriate language. For efficient query processing, we used the materialized phonemic string corresponding to a multilingual text string, instead of an on-line call to TTP system. The edit distance between them is then computed, using the `editdistance` function [54] that takes two strings and returns the edit distance between them; by changing the input parameters for the matching, the function may be made to compute the standard *Levenshtein* edit distance, or a weighted edit distance using a special substitution cost matrix, as required. A *dynamic programming* algorithm is used for this computation, due to the flexibility that it offers in experimenting with different cost functions. If the edit distance is less than the

³As explained earlier, automatic identification of the language of the input string is possible only for a very limited set of languages. In Chapter 5 we propose a new datatype, which stores explicitly the language identifiers, which may be used in this function.

MLNameJoin ($S_l, L_l, S_r, L_r, e, \mathcal{S}_O$)

Input: *Input Strings* S_l, S_r , *Input String Languages* L_l, L_r , *Threshold* e
Set of Languages for output \mathcal{S}_O
Set of Languages with IPA transformations \mathcal{S}_L (as global resource)

Output: TRUE, FALSE or NORESOURCE

1. **if** $L_l \notin \mathcal{S}_L$ **or** $L_r \notin \mathcal{S}_L$ **then return** NORESOURCE;
2. **if** $L_l \in \mathcal{S}_O$ **then**
3. $T_l \leftarrow \text{PhoneticTransform}(S_l, L_l)$;
4. $T_r \leftarrow \text{PhoneticTransform}(S_r, L_r)$;
5. **if** $|T_l| \leq |T_r|$ **then** $Smaller \leftarrow |T_l|$
 else $Smaller \leftarrow |T_r|$;
6. **if** $\text{editdistance}(T_l, T_r) \leq (e * Smaller)$ **then**
 return TRUE **else return** FALSE;

editdistance(S_L, S_R)

Input: String S_L , String S_R

Output: Edit-distance k

1. $L_l \leftarrow |S_L|$; $L_r \leftarrow |S_R|$;
2. Create $DistMatrix[L_l, L_r]$ and initialize to Zero;
3. **for** i **from** 0 **to** L_l **do** $DistMatrix[i, 0] \leftarrow i$;
4. **for** j **from** 0 **to** L_r **do** $DistMatrix[0, j] \leftarrow j$;
5. **for** i **from** 1 **to** L_l **do**
6. **for** j **from** 1 **to** L_r **do**
7. $DistMatrix[i, j] \leftarrow \text{Min} \left\{ \begin{array}{l} DistMatrix[i-1, j] + InsCost(S_{L_i}) \\ DistMatrix[i-1, j-1] + SubCost(S_{R_j}, S_{L_i}) \\ DistMatrix[i, j-1] + DelCost(S_{R_j}) \end{array} \right\}$
8. **return** $DistMatrix[L_l, L_r]$;

Figure 3.2: The MLNameJoin Matching Algorithm

user-specified threshold value (specified as a fraction of the length of the smaller of the equivalent phoneme strings), a positive match is flagged (Line 6).

Match Threshold Parameter

A user-settable parameter, *Threshold* (a fraction between 0 and 1) is an input parameter for the *MLNameJoin* matching. This parameter specifies the user tolerance for approximate matching: 0 signifies that only perfect matches are accepted, whereas a positive threshold specifies the allowable error (that is, edit distance) as the fraction of the size of smaller of the two phonemic strings being compared. The appropriate value for the threshold parameter is determined by the requirements of the application domain, and may be set globally by the administrators for the environment.

Intra-Cluster Substitution Cost Parameter

The three cost functions in Figure 3.2 (Line 7), namely *InsCost*, *DelCost* and *SubsCost*, provide the costs for inserting, deleting and substituting characters in matching the phonemic strings. With different cost functions, different flavors of edit distances may be implemented easily in the above algorithm. For example, all values set to 1 will simulate *Levenshtein* edit distance function.

In addition, *MLNameJoin* supports a *Clustered Edit Distance* parameterization, by extending the *Soundex* [73] algorithm to the phonetic domain, under the assumptions that clusters of like phonemes exist and a substitution of a phoneme from within a cluster is more acceptable as a match than a substitution from across clusters. For example, a substitution of *like-phonemes*, such as, *j* (pronounced as *sh*), is a more acceptable match for the standard *s*, than a substitution of *distinct-phonemes*, such as, *k*. Hence, near-equal phonemes are clustered, based on the similarity measure as outlined in [82], and the substitution cost within a cluster is made a tunable parameter, the *Intra-Cluster Substitution Cost*. This parameter may be varied between 0 and 1, with 1 simulating the standard *Levenshtein* cost function and lower values modeling the phonetic proximity of the like-phonemes.

3.5 Access Structures for MLNameJoin

The approximate matching algorithm used for implementing MLNameJoin is an expensive $O(n^2)$ algorithm; hence, in this section, we explore different index structures and their utility in improving the performance of the multilingual names query, by narrowing the candidate result set, to be checked using explicit calls to MLNameJoin UDF. The proximity measure (*i.e.*, *the edit distance*) used for approximate matching of phoneme strings, is a *metric* and hence requires specialized index structure. The standard B+Tree index is also discussed, due to its availability as the standard index structure in all database systems. It should be noted here that indexes are considered only on the materialized phoneme strings corresponding to the multilingual names attribute.

3.5.1 B+ Tree Index

The MLNameJoin operator may leverage only marginally on the availability of B+ Tree index structures on the materialized phoneme strings corresponding to the multilingual names attribute. The B+ Tree index *cannot* be used for retrieving those phonetic strings that are within an edit-distance of k , as the B+ tree uses the lexicographic ordering of the string values. A proper retrieval based on edit-distance proximity requires the *edit-distance metric* measure to be stored explicitly in the index structure or calculated easily from the contents of the index structure. However, the lexicographic ordering available in the B+Tree index may be leveraged on, to improve the performance of the MLNameJoin operator, as follows: first, by accessing only the index pages (thus reducing the disk I/O's), and, second, by reducing the number of invocations of the EditDistance function to unique values of the attribute (thus reducing in-memory computation). Hence, the improvements depend directly on the number of replicated values in the dataset.

3.5.2 Metric Distance Index

In this section, an alternate index structure, based on pre-computed and indexed metric distances from a *Key String*, is introduced. First, some standard properties of the

edit-distance metric are provided and subsequently used for designing this index structure. The performance of the multilingual names matching operator using this index is discussed later.

Properties of Metric Distances

The following standard properties of edit distances based on their definition are used for designing metric distance index for narrowing the search for a given query string:

Property 3.1: Given two strings a and b at a distance of d_{ab} from each other, for any string s to exist within a distance of d_{as} from a and distance of d_{bs} from b , the condition $(d_{as} + d_{bs}) \geq d_{ab}$ must hold. •

Property 3.2: Given two strings a and b at a distance of d_{ab} from each other, a query to return strings within a distance of d_a and d_b from a and b respectively, and a candidate string s at a distance of d_{sa} ($< d_a$) from a , it may be in the result set if and only if $d_{sa} + d_{ab} \leq d_b$. •

Property 3.3: Given two strings a and b at a distance of d_{ab} from each other, and a query to return strings within a distance of d_a and d_b from a and b respectively, there could be no satisfying strings, if $d_a + d_b < d_{ab}$. •

These properties follows directly from the *triangular property of metric distances* that states that given the edit distance between the strings a and s is d_{as} and between the strings s and b is d_{sb} , then the edit distance between the strings a and b is $\leq (d_{as} + d_{sb})$.

Example 3.3: Consider a query to find the Authors with names close to **Silversmith** (*match threshold 2*) and **Aerosmith** (*match threshold 2*). This query could return no result set as per Property 3.3, since the distance between **Silversmith** and **Aerosmith** is 5. Hence, the query could return with empty result set (correctly), without accessing any table data. ◊

Example 3.4: Consider the query to find the Authors, *phonetically* close to **Silversmith** (*match threshold 2*) and to **Aerosmith** (*match threshold 3*). Suppose the candidate string under consideration is **Silbersmith**; As soon as the edit-distance of the candidate string from **Silversmith** is computed as 1, it may be ruled out immediately from the result

set, as the distance between `Silversmith` and `Aerosmith` is 5, and by Property 3.2, it cannot be within a distance of 4 from `Aerosmith`. \diamond

Properties 3.1 through 3.3 are useful in designing the operator implementation, as they provide means of reducing the edit distance computation, based on the currently evaluated results with no extra edit distance computations. More importantly, if edit-distances of all data strings from a known string is stored, these distances could be exploited intelligently, to reduce the computation required for a given query evaluation, as given in subsequent sections.

Metric Distance Index Structures

Properties 3.1 through 3.3 suggest an alternative index structure that may be used for searching *close* strings: indexing phonemes strings, along with the pre-computed *edit-distances* from a known *key string*, S_{key} . We generated a candidate string for S_{key} , of length equal to the average length of the phonemic string values that are being indexed. It is not necessary or desirable for the S_{key} be chosen from among the values of the attribute, since it may force re-evaluation of all distances when the tuple containing the key is deleted from the table. The edit-distance of each of the phonemic strings in the database from S_{key} is computed and stored along with attribute. A B+tree index is built on the pair, $\langle distance, string \rangle$, called the *Metric Distance Index* (M).

Using Metric Distance Index for Pruning Search

Given M , a **scan** query to retrieve the strings that are at an edit-distance less than d_q from the query string, S_q , may be computed as follows: First, compute distance d_{kq} of S_q from S_{key} . Second, access the index M , and output all those strings with an indexed distance d_k , such that $d_{kq} + d_k \leq d_q$. Third, of the remaining strings, all strings that are at an indexed distance of d_k , such that $|d_{kq} - d_k| > d_q$ are clearly not potential candidates for the answer set, hence pruned. The correctness of Steps 2 and 3 is guaranteed by Properties 3.1 and 3.2, respectively, and they require no explicit distance computation. Finally, for each of the remaining strings (say, s , with a known distances d_s from the key

string S_{key}), such that $d_{kq} + d_s \geq d_q$, compute edit distance to verify if the edit-distance $d_{sq} \leq d_q$, where d_{sq} is the edit distance of s from query string S_q . The final step examines all the remaining strings as candidate strings, and invokes edit distance computation for all of them. The following examples illustrate the power of using metric distance index to prune search:

Example 3.5: Consider a query to find the Authors, phonetically close to **Silversmith**, within a threshold distance of 3. Assume that the metric index structure had been built with S_{key} string as **Silbersmith**. The distance between the query string and S_{key} is 1. First, all records that have a *pre-computed* distance of 2 (that is, $3 - 1$, where 3 is the query distance and 1 is the distance between query string and S_{key}) or less are in the result set. They may be added to the result set, with no distance computation. Second, we could eliminate all records that have a *pre-computed* distance of above 4 (that is, $3 + 1$, as before), as they cannot be in the result set. They are eliminated again with no distance computation. The remaining strings are examined explicitly (with an invocation of `EditDistance` function) to verify if they are part of the result set. Thus, having a metric distance index structure may eliminate a large number of edit-distance invocations, making the query performance better. \diamond

A **join** operation merely repeats the above procedure for every unique string of LHS attribute. The algorithm for using M , on the worst case, examines every string in the index, and hence a worst case complexity of matching with no index.

Using Weighted Metric Distance Index for Pruning Search

Though standard *Levenshtein* edit-distance is used in above discussion, specific characteristics of phonemes – *phonetic closeness* – may be exploited for a more intuitive matching in linguistic domains. The phonemes were clustered based on *like-ness* [82] and a weighted substitution cost matrix is devised as follows: all phoneme substitution within a cluster is costed at a tunable *Intra-cluster substitution cost* parameter (as discussed in Section 3.4), with values from $[0, 1]$, and all substitutions across clusters are

costed at 1. This weighted substitution matrix is used for computing phonetic closeness, which proved to be more intuitive in multilingual names matching. The following Theorem 3.1⁴ ensures that Properties 3.1 through 3.3 hold for weighted edit-distances as well.

Theorem 3.1: A distance measure based on weighted substitution matrix remains metric provided the following conditions hold:

1. The weighted substitution matrix itself is metric;
2. The distance between strings is defined as the minimum of the weighted sums based on substitution matrix. •

The weighted substitution matrix for the clustered phoneme matching, as described in Section 3.4 has the following characteristics (assuming $i, j, p, q \in \Sigma$, and D_x , I_x and $S_{x,y}$ are deletion, insertion and substitution costs, respectively):

1. $\forall_i D_i = I_i = 1$;
2. $\forall_{i,j,i=j} S_{i,j} = 0$;
3. Σ is partitioned into n clusters ($C_i, i = 1 \dots n$) such that, $\forall_{i,j,p,q \in C_k} S_{i,j} = S_{p,q} (\in (0, 1])$ and $\forall_{i \in C_u, j \in C_v, u \neq v} S_{i,j} = 1$.

The clustered phoneme edit distance, based on the above substitution matrix satisfies the conditions of Theorem 3.1 (in fact, these conditions are stronger than necessary). Hence, the pruning of search strings for such weighted edit-distances may leverage on Properties 3.1 through 3.3.

3.5.3 Approximate Index Structures

Approximate Index Structures are designed to identify the candidate records (that is, those that are within an edit distance of k), without having to examine the entire data set. While several approximate index structures, such as, BK tree [13], VP tree [138], M Tree [20] and Bisector tree [69], etc., may be used to get a candidate answer set, an explicit check is necessary for weeding out *false-positives*. A representative sample of approximate index structures were experimented with, to establish their effectiveness of

⁴Here we provide a succinct version of the theorem that was presented and proved in [111].

the index in narrowing down the candidate answers while searching *approximately* for a query string in the database. The experiments were conducted on a real data set of $\approx 100,000$ words from an English dictionary. Figure 3.3 plots the real answer set (the bottom-most curve) and the fraction of database returned as candidate set by different index structures, for a variety of threshold values in the range [0 to 1].

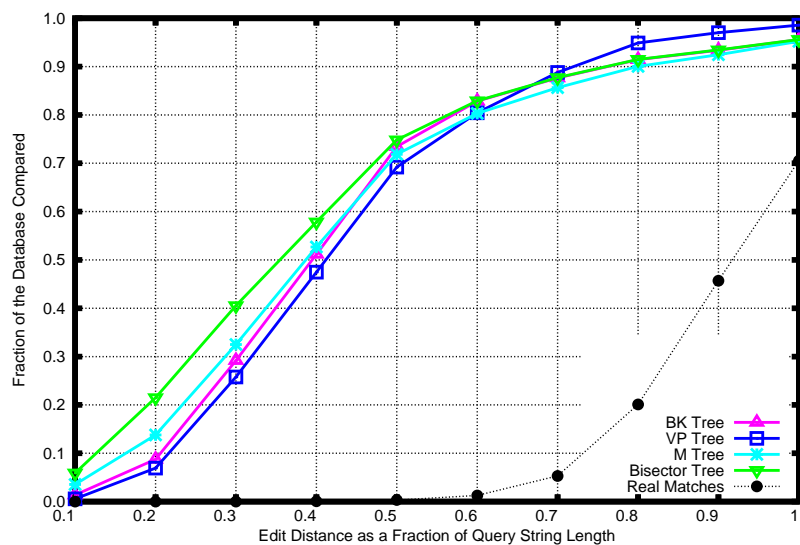


Figure 3.3: Search Efficiency of Approximate Indexes

The *Search Efficiency*⁵ of the approximate indexes may be obtained by dividing the fraction of database that is the real answer by the fraction of the database that was returned as a candidate set by the index structure. It is apparent that the efficiencies of all the approximate structures are abysmally low ($< 1\%$) for threshold values below 0.5.

It should be noted that in the absence of an index structure, a full scan of the database must be made, represented by a horizontal line at the y-axis value of 1.0. At typical threshold values of around 0.3, the cost of using an index structure may well be higher than a full table scan, due to the random access pattern generated by the index-structure. However, we pursue the the implementation of approximate indexes, for the sake of completion and to quantify their search performance for practical query processing. Since approximate indexes cannot be built and used in an *outside-the-server*

⁵*Search Efficiency* is defined as the ratio of the number of real answers to the size of candidate set returned by an index structure.

implementation, the operator performance with only the normal B+Tree and Metric Distance indexes, are presented here. In our *native* implementation of the MLNameJoin operator presented in Chapter 6, the performance of the operator using a height-balanced version of the metric tree index – namely M-Tree [20] – is presented.

3.6 Multilingual Names Matching Quality

So far, the details of the implementation of MLNameJoin operator, such as, its algorithm, parameterization, index structures etc., were discussed; In this section, an experimental setup to measure the quality (in terms of *precision* and *recall*) of the implementation in matching multilingual names is discussed. Subsequently, the results from a set of matching experiments executed on this setup is presented, along with a methodology for tuning the parameters for a high quality match.

3.6.1 Dataset

With regard to the datasets to be used in the experiments for establishing the quality of multilingual names matching, we had two choices: experiment with multilingual lexicons and verify the match quality by *manual relevance judgment*, or alternatively, experiment with tagged multilingual lexicons (that is, those in which the expected matches are marked beforehand) and verify the quality mechanically. We chose to take the second approach, but because no tagged lexicons of multiscript names were readily available⁶, we created our own lexicon from existing monolingual ones, as described below.

Proper names from three different sources were selected so as to cover common names in English and Indic domains. The first set consists of randomly picked Indic names from the *Bangalore Telephone Directory*, covering most frequently used Indian names. These names were transcribed into two markedly different Indic languages that share no common characters and are distinct phonetically – Hindi and Tamil. The second set consists of randomly picked English names from the *San Francisco Physicians Directory*, covering

⁶Bi-lingual dictionaries mark *semantically* equivalent words, and not *phonetically*, similar nouns.

most common American first and last names. The third set consisting of generic English names representing Places, Objects and Chemicals, was picked from the *Oxford English Dictionary*. Together the set yielded about 400 names, covering three distinct name domains. Most of the names were converted from the original script to the other two in the set $\{English, Tamil, Hindi\}$, thus yielding about a thousand names, in three different scripts. All phonetically equivalent names (but in different scripts) were manually tagged with a common *tag-number*. The tag-number is used subsequently in determining quality of a match as follows: – any match of two multilingual strings is considered to be correct if their tag-numbers are the same, and considered to be a *false-positive* otherwise. Further, the fraction of *false-dismissals* can be readily computed since the expected set of correct matches is known, based on the tag-numbers in a given set of multilingual names.

To convert English names into corresponding phonetic representations, standard linguistic resources, such as the *Oxford English Dictionary* [101] and TTP converters available on-line in the multilingual portal *www.ForeignWord.com* [42], were used. For Indic strings, *Dhvani* TTP converter [30] was used. Further those symbols specific to speech generation, such as the supra-segmentals, diacritics, tones and accents were removed. Sample phoneme strings for some multiscript strings are shown in Figure 3.4.

Lexicographic String	Language	Phonetic Representation (in IPA)
University	English	junəvɜrsɪti
நெரு	Tamil	neɪru
École	French	eikøl
இந்திரியா	Tamil	ɪndɪɾɪjɑ
हार्द्रेडजन	Hindi	hɑrdɾɛdʒən
Espanöl	Spanish	ɛspanjøl

Figure 3.4: **Phonemic Representation of Test Data**

The frequency distribution of the data set with respect to string length is shown in Figure 3.5, for both lexicographic and materialized phonetic representations. The set had an average lexicographic length of 7.35 and an average phonemic length of 7.16. Note that though the Indic strings are typically visually much shorter than the corresponding

English strings, their character lengths are similar owing to the fact that most Indic characters are composite glyphs and are represented by multiple Unicode characters. Further, it can be observed that their phonemic string length profiles are nearly identical, confirming our hypothesis that their aural representations are similar.

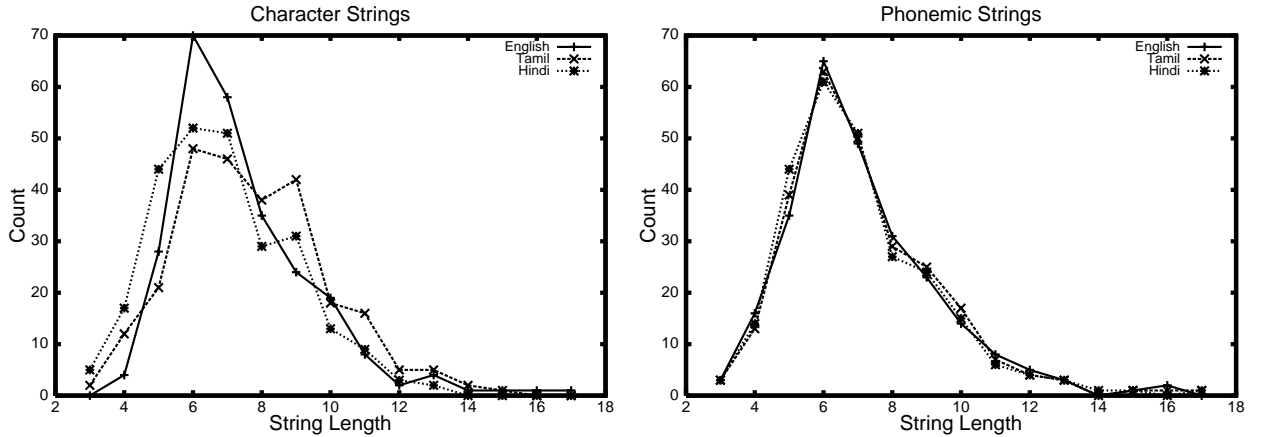


Figure 3.5: Distribution of Multiscript Dataset

3.6.2 Performance Metrics

Multilingual name matching queries (as shown in Figure 1.3) were run on the dataset described above. For each query, two metrics – *Recall* and *Precision* – were measured. The recall and the precision figures were computed using the following methodology: each phonemic string in the data set was matched with every other phonemic string, counting the number of matches (m_1) that were correctly reported (that is, the tag-numbers of multiscript strings being matched are the same), along with the total number of matches that are reported as the result (m_2). If there are n equivalent groups (that is, those with a distinct tag-number) with n_i of multiscript strings each (note that both n and n_i are known for a given data set), the *precision* and *recall* metrics are calculated as follows:

$$Recall^7 = m_1 / \sum_{i=1}^n \binom{n_i}{2}$$

$$Precision^8 = m_1 / m_2.$$

⁷ *Recall*, in plain English, is the fraction of correct matches that appear in the result.

⁸ *Precision*, in plain English, is the fraction of the delivered results that are correct.

The expression in the denominator of *recall* metric is the ideal number of matches, as every pair of strings (*i.e.*, ${}^n C_2$) with the same tag-number must match. Further, for a perfect answer set, both the metrics must be 1. Any deviation indicates the inherent fuzziness in the query processing, due to the differences in the phoneme sets of the languages and the losses in the transformation to phonemic strings. Further, the two query input parameters – *user match threshold* and *intracluster substitution cost* (explained in Section 3.4) were varied over a range of values in the interval $[0,1]$, to measure their effect on the quality of the output.

3.6.3 Multilingual Names Matching Quality

The plots of the *recall* and *precision* metrics, on the matching experiments outlined in the previous section are provided in Figure 3.6. The results plot the measured metrics against various combinations of *user match threshold* and *intracluster substitution costs* parameters.

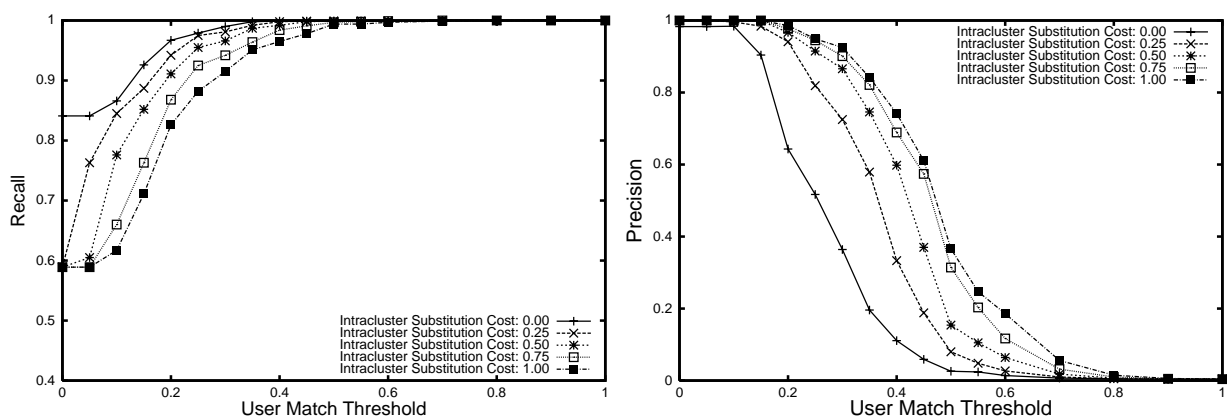


Figure 3.6: MLNameJoin Operator Recall and Precision

The curves in the recall plot of Figure 3.6 indicate, not surprisingly, that the recall metric improves with increasing user match threshold and asymptotically reaches perfect recall, after a value of about 0.5, for all intra-cluster substitution costs. An interesting point to note is that the recall gets better with reducing intracluster substitution costs, validating the assumption of the *Soundex* algorithm [73].

In contrast, the curves in the precision plot of Figure 3.6 indicate, as expected, that the precision metric drops with increasing threshold; the drop is negligible for threshold values upto about 0.3, but is steep beyond. It is interesting to note that with an intracluster substitution cost of 0, the precision drops very rapidly at a user match threshold of 0.1 itself. That is, the *Soundex* method, which is good in recall, is very ineffective with respect to precision, as it introduces a large number of false-positives even at low thresholds.

For an optimal match, the recall and precision values must be as close to 1 as possible. However, it is apparent from Figure 3.6 that a high recall requires a high threshold value and a low intracluster substitution cost, and that a high precision requires a low threshold value and a high intrasubstitution cost. Hence, to obtain the best quality match, it is important to select optimal parameters that maximize both recall and precision.

Selection of Ideal Parameters for Phonetic Matching

Figure 3.7 shows the combined *precision-recall* curves, with respect to each of the query parameters, namely, *intracluster substitution cost* and *user match threshold*. For the sake of clarity, only the plots corresponding to the costs of 0, 0.5 and 1, and plots corresponding to thresholds of 0.2, 0.3 and 0.4, are shown. The top-right corner of the precision-recall space corresponds to a perfect match and the closest points on the precision-recall graphs to the top-right corner correspond to the query parameters that result in the best match quality.

As can be seen from Figure 3.7, the best possible matching for our dataset is achieved by a substitution cost between 0.25 and 0.5, and for thresholds between 0.25 and 0.35, corresponding to the knee regions of the respective curves. With such parameters, the *recall* is $\approx 95\%$, and *precision* is $\approx 85\%$. That is, $\approx 5\%$ of the real matches would be *false-dismissals*, and about $\approx 15\%$ of the results are *false-positives*, which must be discarded by post-processing, using non-phonetic methods.

We also would like to emphasize that the quality of approximate matching depends on the phoneme sets of languages, the accuracy of the phonetic transformations, and

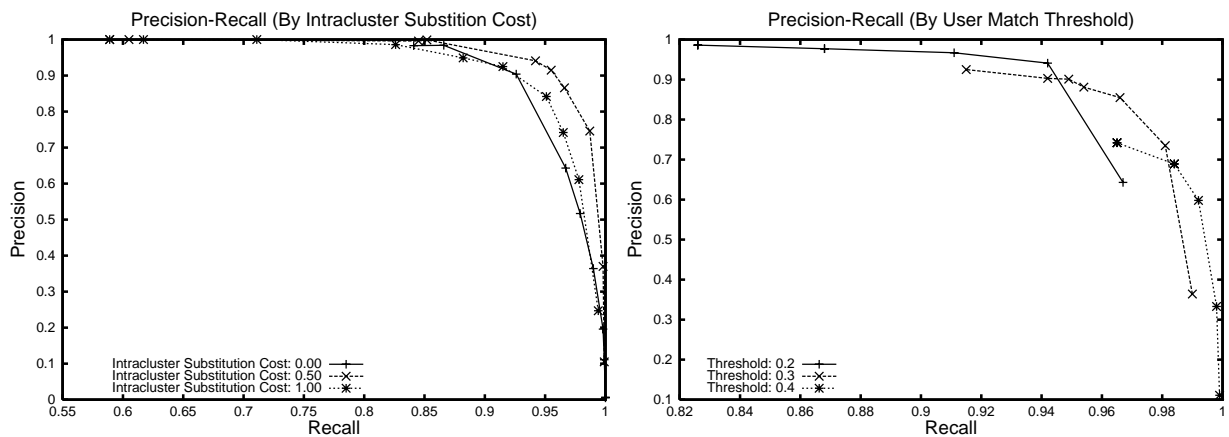


Figure 3.7: MLNameJoin Combined Precision-Recall Graphs

more importantly, on the data sets themselves. Hence the optimal matching parameters need to be tuned, for specific datasets and domains. While automatic generation of the ideal matching parameters for a given data set is possible with a given hand-verified training set using machine learning techniques, such work is left as future extensions to the current research.

3.7 MLNameJoin Performance

In this section, the performance of query processing using the MLNameJoin operator, implemented using UDF methodology and a set of different index structures, on a set of commercial database management systems, is presented. We show that the performance is primarily affected by the high overheads involved in the UDF calls and subsequently, explore avenues to improve the query performance to a level sufficient for practical use.

3.7.1 System and Database Environment

A standard Intel Pentium IV (1.7 GHz) workstation with 256MB memory running Windows 2000 Professional operating system was used as the test machine for the performance study. Three popular database management systems – Microsoft SQL Server (Version 8.0.194), IBM DB2 Universal Server (Version 7.1.0) and Oracle 9i Database

Server (Version *9.0.1*) – were used to measure the baseline performance of the multilingual names matching operator, implemented as a UDF function. The systems are identified randomly, as *A*, *B* and *C*, to protect their identities. Before each experiment, the machine was quiesced and only the database system being tested and allied processes were allowed to run in order to have measurement parity between the systems.

The `MLNameJoin` operator was implemented on top of the relational database system in an environment that is appropriate for each of the systems – specifically, using the PL/SQL and Java procedures in Oracle and DB2 database systems respectively, and using SQL scripts in SQL Server database system. Both the multilingual name string and their phonetic representations materialized in the IPA alphabet, were stored in Unicode format. The SQL queries using multilingual names matching `MLNameJoin` operator invokes the UDF at the runtime.

3.7.2 Dataset

Since the real multiscript lexicon used in the previous section was not large enough for performance experiments, a large dataset was synthetically generated using the multiscript lexicon used in the previous section, as a seed. Specifically, each of the multilingual name strings were concatenated, with all remaining name strings *within a given language*, generating a set of about 200,000 names. The corresponding materialized phoneme strings were also concatenated similarly. Each of the generated records is tagged with a generated tag number, specifically $(t_1 * 1000 + t_2)$, where t_1 and t_2 are, respectively, the tag numbers of the first and the second strings being concatenated. The above scheme ensures that all strings that match will have the same generated tag number, since the original tag numbers are much less than 1000. Figure 3.8 shows the frequency distribution of the string lengths of the generated data set – in both character and (generated) phonetic representations with respect to string lengths. The average character and phonemic lengths of the generated set are, 14.71 and 14.31, respectively.

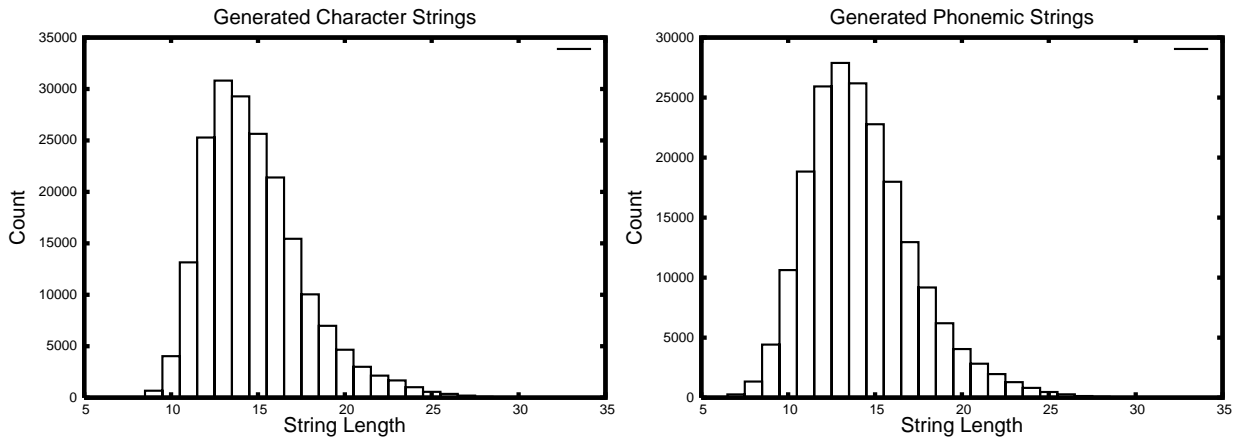


Figure 3.8: Distribution of Generated Multiscript Data Set

3.7.3 Baseline MLNameJoin Performance

To create a baseline for performance, the scan and the join queries using the MLNameJoin operator (along the lines of the samples shown in Figures 1.3 and 1.8) were run on the large generated data set, in each of the database systems. The baseline performance of the UDF on the three popular database systems (as the runtime of respective queries, in *Seconds*) is given in Table 3.1. The join experiment was done on a 0.2% subset of the original table, since the full table join using UDF takes order of days to complete.

Query	System A	System B	System C
Scan	2410	1564	1418
Join	2474	2220	4004

Table 3.1: MLNameJoin Operator Performance

As can be seen clearly, the performance of the multilingual names matching operator, implemented as a UDF is slow in all database systems. The primary reason for the performance seem to be the *UDF*, since the equi-join operator (using the standard lexicographic equality) executes the queries under a second, in each of the database systems. Hence, we chose a system with the lowest performance, to experiment with some optimization strategies for making the execution more efficient. Though the average

performance was worst in System A, we chose System C for the subsequent analysis and performance improvement due to the fact that its implementation is more tightly integrated with database server, allowing us to observe the query plans by varying query parameters and available index structures. Further, the average performance difference between the systems A and C (computed as, 2442 and 2382) is not very significant.

As an immediate measure for improving the performance of the operator, different index structures were created: First, a B+ Tree index was created on the phonemic attribute and the `MLNameJoin` algorithm was modified to invoke the UDF, only on unique data values. Second, a Metric Distance Index structure (as discussed in Section 3.5.2) was created and the names matching algorithm modified to invoke the UDF on only those records which could potentially match with the query string. When there are no duplicates in the database, the B+Tree index offers no performance improvements over the case with no index, since the UDF is invoked on all records. Hence, for the B+Tree case, all the attributes are replicated between 1 and 7 uniformly, resulting in an average multiplicity of 4. For the metric distance index, a random phonemic string of length equal to the average length of the dataset was used as the *Key String*. The results of the performance of System C, with index structures is given in Table 3.2.

Query	Matching Methodology	Time
Scan	<code>MLNameJoin UDF</code>	1418 Sec
Scan	<code>MLNameJoin UDF (with B+ Tree Index)</code>	374 Sec
Scan	<code>MLNameJoin UDF (with Metric Distance Index)</code>	356 Sec
Join	<code>MLNameJoin UDF</code>	4004 Sec
Join	<code>MLNameJoin UDF (with B+ Tree Index)</code>	1824 Sec
Join	<code>MLNameJoin UDF (with Metric Distance Index)</code>	1728 Sec

Table 3.2: `MLNameJoin` Operator Baseline Performance

It is clear that even with the index structures, the `MLNameJoin` operator has unacceptably low performance. To improve the efficiency of matching with `MLNameJoin` operator, we explore two alternative optimization techniques – *Q-Grams* and *Phonemic Indexes* – that cheaply provide a candidate answer set, members of which are verified

by the accurate but expensive `MLNameJoin` UDF. These two techniques exhibit different quality and performance characteristics, and may be chosen depending on application requirements.

3.7.4 Optimization #1: Q-Gram Index

In this section, we show here that the popular *Q-Gram* technique for approximate matching of normal text strings [52], may be adapted successfully for phonetic matching as well. The database was first augmented with a table of *positional q-grams* of the original phonemic strings. The size of q-grams was set at 3, (that is, *trigrams*), as it was empirically shown to perform effectively for approximate matching tasks [53]. Subsequently, the three filters that are employed and shown to be successful in the monolingual world for approximate matching were used to filter out a majority of the non-matching strings, using standard database operators only. These filters weed out most non-matches cheaply, leaving the accurate, but expensive `MLNameJoin` UDF to be invoked (to weed out *false-positives*) on a vastly reduced candidate set. Of the three filters that are employed here, the *length* filter ensures that the length of the candidate strings are *close enough*, the *count* filter ensures a minimum number of common q-grams between matching strings and the *position* filter ensures that the matching q-grams are in approximately in the same positions in the respective strings. A brief descriptions of the filters are provided here, and interested readers are referred to [52] for details.

Length Filter leverages the fact that strings that are within an edit distance of k cannot differ in length by more than k . This filter does not depend on the q-grams themselves, but only on their counts.

Count Filter ensures that the number of matching q-grams between two strings σ_1 and σ_2 of lengths $|\sigma_1|$ and $|\sigma_2|$ respectively, must be at least $(\max(|\sigma_1|, |\sigma_2|) - 1 - (k - 1) * q)$, a necessary condition for two strings to be within an *edit-distance* of k .

Position Filter ensures that a positional q-gram of one string does not get matched to a positional q-gram of the second that differs from it by more than k positions.

```

SELECT N.ID, N.Name
FROM Names N, AuxNames AN, Query Q, AuxQuery AQ
WHERE N.ID = AN.ID
      AND Q.ID = AQ.ID
      AND AN.Qgram = AQ.Qgram
      AND |len(N.PName) - len(Q.str)| ≤ e * length(Q.str)
      AND |AN.Pos - AQ.Pos| ≤ (e * length(Q.str))
GROUP BY N.ID, N.PName
HAVING count(*) ≥ (len(N.PName) - 1 - ((e * len(Q.str) - 1) * q))
      AND MLNameJoin(N.PName, Q.str, e)

```

Figure 3.9: MLNameJoin SQL Script with Q-Gram Index

A sample SQL query using q-grams is shown in Figure 3.9, assuming that the query string is transformed into a record in table Q, and the auxiliary q-gram table of Q is created in AQ. The *Length Filter* is implemented in the fourth condition of the SQL statement, the *Position Filter* by the fifth condition, and the *Count Filter* by the GROUP BY/HAVING clauses. As can be noted in the above SQL expression, the MLNameJoin UDF function is called at the end, *after* all three filters have been utilized, implying that the UDF is invoked only on those records that has passed through all the filters.

Query	Matching Methodology	Time
Scan	MLNameJoin UDF + q-gram index	13.5 Sec
Join	MLNameJoin UDF + q-gram index	856 Sec

Table 3.3: MLNameJoin Performance with Q-Gram Index

The performance of the selection and join queries, after including the q-gram optimization, are given in Table 3.3. Compared to the values in Table 3.2, the use of this optimization improves the *scan* query performance by an order of magnitude and the *join* query performance by two-fold. The improvement in join performance is not as dramatic as in the case of scans, due to the additional joins that are required on the large q-gram tables. The performance improvements here are not as high as those reported in previous literature [52], perhaps due to the use of a standard database system and the implementation of MLNameJoin as a UDF in an interpreted environment.

3.7.5 Optimization #2: Phonemic Index

In this section, we outline a *phonemic indexing* technique that may be used for accessing the *near-equal* phonemic strings, using a standard database index. We exploit the following two facts to build a compact database index: First, the substitutions of *like* phonemes keeps the recall high (as evidenced in Figure 3.6), and second, phonemic strings may be transformed into smaller numeric strings for indexing as a number. However, the downside of this method is that it suffers from a drop in recall with the definition of our closeness measure based on edit-distance.

To implement the above strategy, the phoneme strings need to be transformed to a number, such that phoneme strings that are *close* to each other map to the same number. For this, a modified version of the *Soundex* algorithm [73] was used, customized to the phoneme space: first the phonemes were grouped into equivalent clusters along the lines outlined in [82], and a unique number was assigned to each of the clusters. Each phoneme string was transformed to a unique numeric string, by concatenating the cluster identifiers of each phoneme in the string. The numeric string thus obtained was converted into an integer – *Grouped Phoneme String Identifier* – which is stored along with the phoneme string. A standard database B+ Tree index was built on the grouped phoneme string identifier attribute, thus creating a compact index structure using only integer datatype.

For an `MLNameJoin` query using phonemic index, the operand multiscrypt string is transformed to its phonetic representation and subsequently to its grouped phoneme string identifier, using the same methodology. The index on the grouped phoneme string identifier is used to retrieve all the candidate phoneme strings, which are then tested for a match invoking the `MLNameJoin` UDF explicitly with the user specified match tolerance. The query with `MLNameJoin` operator is transparently mapped to an internal query that uses the phonemic index, as shown in Figure 3.10. Note that any two strings that match in the above scheme are *close phonetically*, as the differences between individual phonemes are from only within the pre-defined cluster of phonemes. However, the downside of this methodology is that those strings that are within the classical definition

of *edit-distance*, but with substitutions across groups, will not be reported, resulting in *false-dismissals*. While some of these false-dismissals may be corrected by a more robust design of phoneme clusters and cost functions, not all *false-dismissals* can be corrected in this method.

```
SELECT N.ID, N.Name
FROM Names N, Query Q
WHERE N.GroupedPhonStringID = Q.GroupedPhonStringID
      AND MLNameJoin(N.PName, Q.PName, e)
```

Figure 3.10: MLNameJoin SQL Script with Phonemic Indexes

A B+Tree index was created on the grouped phoneme string identifier attribute and the same selection and join queries on the large synthetic multiscrypt dataset were rerun. The MLNameJoin operator was modified to use this index, as shown in the SQL expression in Figure 3.10. The scan and join query performance with the phonemic index, is given in Table 3.4.

Query	Matching Methodology	Time
Scan	MLNameJoin <i>UDF</i> + <i>phonemic index</i>	0.71 <i>Sec</i>
Join	MLNameJoin <i>UDF</i> + <i>phonemic index</i>	15.2 <i>Sec</i>

Table 3.4: MLNameJoin Performance with Phonemic Index

While the performance of the queries with phonemic index is an order of magnitude better than that achieved with q-gram technique, the phonemic index introduces a small, but significant 5 – 6% *false-dismissals*, with respect to the classical edit-distance metric. A more robust grouping of like phonemes may reduce this drop in quality, but may not eliminate it. Hence, the phonemic index approach may be suitable for applications which can tolerate false-dismissals, but require a very fast response time.

3.8 Related Research

To the best of our knowledge, the problem of matching multilingual names strings across languages has not been addressed previously in the database research literature. Though the Information Retrieval (IR) and Speech Processing research communities have pursued multilingual matching, both these communities have primarily focused on the quality of matching in the respective domains. While we use some of the techniques proposed by the IR community, our focus is on the implementation and performance of these techniques on database systems. Our use of a phonetic matching scheme for multiscrypt strings is inspired by the successful use of this technique in the monolingual context by the database and IR research communities.

There are vast amounts of research literature in the IR community on cross-lingual search issues, but nearly all of them focus on Natural Language Processing techniques that may be quite unsuitable for database query processing of attribute level data. We refer to [116] for a complete list of research in this area. Specific phonetic matching approaches were addressed in [102] and [141], where the authors present their experience in phonetic matching of uniscript text strings, and provide measures on correctness and performance of matches with a suite of techniques (such as *Soundex*, *Phonix* etc.). Such algorithms work on English strings only. In [32], the performance of the above algorithms were evaluated on Swedish names, with similar results reported. Our work extends some of these ideas for multilingual names matching. Proprietary matching techniques, based on similar techniques are employed in the pharmaceutical industry [78] to find *look-alike sound-alike (LASA)* drug names, which may lead to trademark violations or potentially dangerous medical conditions. A machine learning algorithm for learning cross-lingual phonetic similarity between English and Chinese strings given a training set, was explored in [80]. Though this work focuses on learning phonetic similarity between the two languages, the authors conclude that once the similarity is learned, the phoneme based approaches for matching multilingual data perform comparably to grapheme based approaches for monolingual data, supporting our solution strategy for multilingual names matching using phonemic matching. This study did not evaluate the

runtime performance of such matching.

The approximate matching of strings has been a problem of prime interest to the data integration community. For matching phonemic strings, the basic string *edit-distance* metric, for its generality and its flexibility for modeling variations that are specific for phonetic domains (such as, phoneme clusters and specialized substitution cost matrix), were resorted to. A comprehensive comparison of performance of different similarity measures for names and records matching is presented in [22]. Among the edit-distance based approaches, the authors concluded that on an average, the Monge-Elken [91] method (a variation of standard edit-distance function with allowance for differential costing of gaps) resulted in the best match quality, though they also show that the standard edit-distance measure performed better in nearly half of their datasets. We decided to implement our multilingual names operator using standard edit-distance method for the flexibility it provided in experimenting with different variations. Further, the performance may be expected to be similar to that of Monge-Elkin, as both the approaches use the same basic dynamic-programming algorithm. In [66], the authors present a similarity join technique for matching string attributes, after mapping them to an Euclidean space using fast *string map* algorithm. While the filtering techniques used in our approach ensured accuracy of the results, the approach provided in [66] promises efficient performance, at the expense of a small drop in accuracy. We hope to pursue this technique in the future, and to verify its utility in database environments with frequent changes in data values.

In addition, the standard edit-distance measure is used successfully for approximate matching in database research community [52]. The techniques for efficient implementation of edit-distance measure is an active research topic, and we refer to [96] for a comprehensive survey. The basic techniques from such research were used here, with the algorithms modified appropriately to suit the requirements of phonetic domain.

Apart from being multiscript, another novel feature of our work is the quantification of the *run-time efficiency* of the multilingual names matching in the context of a popular state-of-the-art database systems. This is essential for establishing the viability of multilingual matching in online *eCommerce* and *eGovernance* applications. To improve

the efficiency of `MLNameJoin`, the Q-Gram filters that were employed successfully in [52] for approximate matches in monolingual databases were used and shown to be efficient in phonemic space too. We also investigate the phonemic indexes to speed up the match process – such indexes have been previously considered in [140] where the phonetic closeness of English lexicon strings is utilized to build simpler indexes for text searches. Their evaluation is done with regard to in-memory indexes, whereas our work investigates the performance for persistent on-disk indexes. Further, these techniques are extended to multilingual domains.

In [93], a search engine for Indic languages has been described, that searches Devanagari documents for a given search strings. Though this system is designed to work with Unicode character set, the search is primarily restricted to Hindi language. An interesting search option provided in this system is the phonetic tolerance, in which the search query is expanded with similar sounding words that are generated with substitution of specific characters from an equivalent set. However, this system does not search cross-lingually. A cross-lingual search feature for Indic languages is proposed in [117, 118], where the authors present a phonetic distance based measure for similarity based on representation of a phoneme as a vector in a multi-dimensional space. The distance between two sequences of phonemes was computed using a dynamic time warping algorithm [70], weighted appropriately using a multivalued feature vector [74]. This work focuses on the linguistic issues in phonetic matching, while our work focuses on database performance on such matching methodology, and hence are complementary to each other. The `MLNameJoin` algorithm could be easily altered to adopt the similarity measure presented in their work.

3.9 Conclusions on Multilingual Names Matching

In this chapter, we detailed the implementation of multilingual names matching functionality, as defined in Chapter 1. Currently, such functionality is not supported by any of the current commercial or open-source database systems.

Our implementation methodology depends on transforming matching in the *lexicographic space* to the equivalent *phonetic space*. The multilingual text strings were converted into equivalent phoneme strings using standard TTP linguistic resources and due to the inherently fuzzy nature of the phonetic space, approximate matching techniques were employed for matching the transformed phonemic strings.

The multilingual names matching operator was implemented as a UDF in commercial systems to confirm the feasibility of our strategy on unmodified relational database systems. A suite of experiments to measure the match quality, namely *Recall* and *Precision*, in a real multilingual data set, showed good recall ($\approx 95\%$) and precision ($\approx 85\%$), indicating the potential of such an approach for practical query processing. Further, we showed that the poor performance associated with the UDF implementation of approximate matching may be improved significantly, by employing one of the two alternate methods: the *Q-Gram* technique, and a *Phonemic Indexing* technique. These two techniques exhibit different quality and performance characteristics, and may be chosen depending on the requirements of an application. However, both the techniques are capable of improving the multilingual name matching performance by orders of magnitude, to a level sufficient for practical adoption in deployed systems.

Chapter 4

Multilingual Semantic Matching

4.1 Overview of the Chapter

In this chapter, after providing some background information, we define formally the multilingual semantic matching operator – `MLSemJoin` – that matches multilingual text strings that store categorical information. Subsequently, we detail our implementation of the functionality as a derived-operator using existing `SQL` features of database systems. Finally, the matching performance is profiled on a set of commercial database systems; while the basic implementation may be too slow for practical deployments, we show that by tuning the storage and indexes to match the characteristics of the linguistic resources, the performance may be improved to a level sufficient for practical use.

4.2 Background Information

In this section, some background information on the linguistic resources that are needed for our semantic matching methodology is provided.

4.2.1 WordNet: A Linguistic Resource

A brief overview of `WordNet` [135], a standard linguistic resource that organizes words and their meanings of a language in a form that may be mechanically interpreted, is

presented here; further details may be found in [39]. The availability of WordNet in multiple languages with rich semantic interlinking between them, has made possible our proposed implementation of multilingual semantic matching operator.

Word Representations: Form *vs.* Meaning

A word may be thought of as a lexicalized concept; simply, it is the written form of a mental concept that may be an object, action, description, relationship, etc. Formally, it is referred to as a *Word-form*. The concept that it stands for is referred to as *Word-sense*, or in WordNet parlance, *Synset*. The defining philosophy in the design of WordNet is that a synset is sufficient to identify a concept uniquely. A short description, similar to the dictionary meaning, called the *Gloss* is provided with every synset, for human understanding. Two words are said to be synonymous, *or semantically the same*, if they have the same synset and hence map to the same mental concept. Synonymy satisfies the *Leibniz's* principle that says that two words are synonymous if the substitution of one of the words for the other does not change the truth value of the sentence. WordNet organizes all relationships between the concepts of a language as a semantic network between synsets.

The WordNet has a *Lexical Matrix* function that maps word forms to their meanings, which constitutes the basis for mapping words to synsets. For example, the word-form `bird` corresponds to several different synsets, two of which are {*a vertebrate animal that can typically fly*} and {*an aircraft*}; each of these two synsets is denoted differently with subscripts, in the English Noun Hierarchy shown in Figure 4.1. The synsets of a language are divided into five distinct categories (specifically, *nouns, verbs, adjectives, adverbs and relationships*), and in this thesis we only consider the *Nouns* category of words for multilingual categorical matching; Nouns represent the most significant category of a language for query processing, since *nearly a fifth of the normal text corpora and the majority of query strings* [79] are from this category.

Interlinked Noun Taxonomical Hierarchy

The nouns in English WordNet are grouped under approximately twenty-five distinct *Semantic Primes* [39], covering distinct conceptual domains, such as *Animal*, *Artifact*, etc. Under each of the semantic primes, the nouns are organized in a taxonomic hierarchy, as shown in Figure 4.1, with *Hyponyms* links signifying the *is-a* relationships (shown in solid arrows). Several efforts are underway – such as the *European WordNet* (EWN) [37] and the *Indo-WordNet* (IWN) [15] – to develop WordNet linguistic resources in different languages along the lines of English WordNet, including the taxonomic hierarchies of the respective noun forms. A Chinese WordNet (CWN) initiative, along the lines of English WordNet, is outlined in [18].

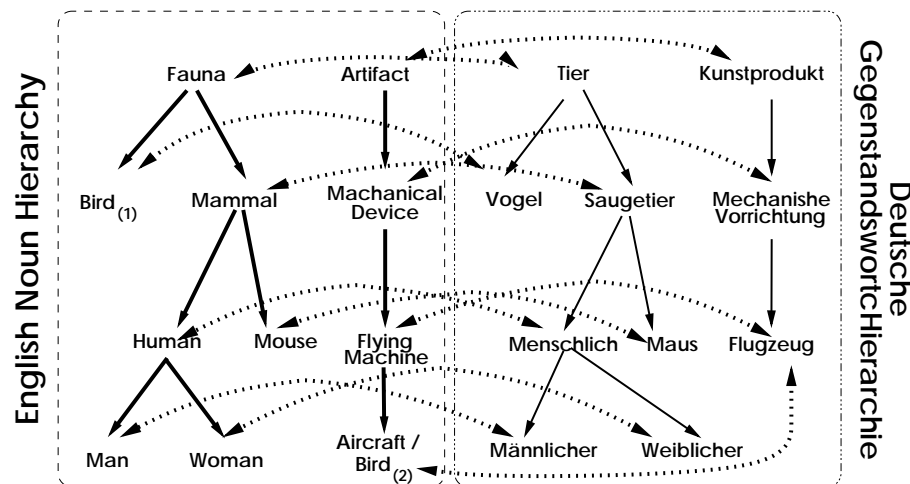


Figure 4.1: Sample Inter-linked *WordNet* Noun Taxonomic Hierarchy

A common feature among such initiatives is that they keep the basic taxonomic hierarchies nearly the same as that of English and provide mappings from their synsets to that of English. Further, semantically equivalent synsets between WordNets of different languages are interlinked using *Inter-Lingual-Index* (ILI) links (shown as dotted arrows) and are available partly in some western European languages currently [37], and is planned for in Indic languages [65, 95]. Figure 4.1 shows a simplified interlinked hierarchy in English and German. Such interlinked hierarchies are used for defining semantic matching in the following section.

4.3 Multilingual Semantic Matching Implementation

In this section, the crosslingual matching of multilingual text attributes that store categorical values, as shown in Figure 1.5 and Figure 1.6, is detailed. The intuition behind the matching strategy is outlined in Section 1.5.3. In this section, first the multilingual semantic matching functionality is described formally and subsequently its implementation using WordNet linguistic resources is presented.

4.3.1 MLSemJoin Implementation Details

The definitions in this section assume only that the values of an attribute are from a specified domain, \mathcal{D} , with a set of distinct atomic semantic values. Within a domain, the values are assumed to be arranged in a taxonomic hierarchy \mathcal{H} that define *is-a* relationships among them¹. Note that this hierarchy may be a collection of directed acyclic graphs. Given an atom x and a domain hierarchy \mathcal{H} , the transitive closure of x in \mathcal{H} is unique, and is denoted by $\mathcal{T}_{\mathcal{H}}(x)$. Similarly, the transitive closure of a set (X) of values from \mathcal{D} , is denoted by $\mathcal{T}_{\mathcal{H}}(X)$, and is defined as $\cup_i \mathcal{T}_{\mathcal{D}}(x_i)$, where $x_i \in X$. Assuming the above notation, we provide the following definitions for *semantic matching* using \mathcal{H} as:

Definition 4.1 [Is-A]: Given a taxonomic hierarchy \mathcal{H} in domain \mathcal{D} and two nodes x and y in \mathcal{D} , x *is-a* y , $\iff x \in \mathcal{T}_{\mathcal{H}}(y)$.

Example 4.1: The predicate (Man *is-a* Human) is true, in the hierarchy of Figure 4.1, since the transitive closure of Human in the English noun taxonomic hierarchy in Figure 4.1, is {Human, Man, Woman}. \diamond

Definition 4.2: Given \mathcal{H} in domain \mathcal{D} and two sets of nodes X and Y in \mathcal{D} , X *is-a* Y , $\iff X \subseteq \mathcal{T}_{\mathcal{H}}(Y)$.

Example 4.2: The predicate (Bird *is-a* Fauna) will evaluate to false, as the set of synsets by the lexical matrix function for Bird, namely the set {Bird₁, Bird₂}, is not a subset of the

¹The hierarchy determines the domain in which semantic matching is done: while WordNet is used in *linguistic* domain, appropriate domain-specific hierarchies may be used for matching in specific domains.

closure (in English noun taxonomic hierarchy) of Fauna, which is {Fauna, Bird₁, Mammal, Human, Mouse, Man, Woman}. \diamond

Since linguistic domain ontologies have low resolution power (that is, words usually have multiple meanings), a weaker version of the semantic equality is provided, as follows:

Definition 4.3 [Is-Possibly-A]: Given a taxonomic hierarchy \mathcal{H} in domain \mathcal{D} and two sets of nodes X and Y in a domain \mathcal{D} , X *is-possibly-a* Y , iff $X \cap \mathcal{T}_{\mathcal{H}}(Y) \neq \phi$.

Example 4.3: The predicate (Bird *is-possibly-a* Fauna) evaluates to true, as the instantiation of Bird, namely, {Bird₁, Bird₂} has a *non-empty* intersection with the closure of Fauna in the English noun taxonomic hierarchy. On the same logic, the predicate (Bird *is-possibly-a* Artifact) also evaluates to true. \diamond

We use this weaker notion of semantic equality – namely, *is-possibly-a* – to implement the MLSemJoin operator. The *direct* matching of multilingual categorical values is done by examining the Inter-Lingual-Indexes (ILI) links between the WordNet taxonomical hierarchies, and the *possible* matching (which are necessary when the LHS operand value is a subclass of the RHS operand value) requires an evaluation of the transitive closure of the RHS operand in the interlinked WordNet hierarchy. Such matching methodology is *semantic* as it makes use of WordNet semantic classifications from the respective languages. The usage of WordNet linguistic taxonomic hierarchy, in order to implement the MLSemJoin operator, is as follows: Let $\mathcal{W}_{\mathcal{I}}$ be the WordNet of language L_i . Let $\mathcal{P}_{\mathcal{I}} = \cup_i p_i$, where p_i is the semantic primitives (that is, *synsets*) of L_i . By definition, $\mathcal{W}_{\mathcal{I}}$ contains semantic primitives p_i , of L_i . The noun taxonomic hierarchy defines a set of DAG's, $\mathcal{H}_{\mathcal{I}}$ between the elements of $\mathcal{P}_{\mathcal{I}}$. A WordNet defines a mapping $\mathcal{M}_{\mathcal{I}}$, between a wordform (w_i) and its meanings (P_w), as $\mathcal{M}_{\mathcal{I}}:w_i \rightarrow P_w$, where P_w is a set of semantic primitives of L_i , that is, $P_w \subseteq \mathcal{P}_{\mathcal{I}}$. Consider the union of all semantic primitives of a set of languages of interest, $\mathcal{P}_{\mathcal{ML}} (= \cup_i \mathcal{P}_{\mathcal{I}})$ and the union of interrelationships between them $\mathcal{H} (= \cup_i \mathcal{H}_{\mathcal{I}})$. Clearly, \mathcal{H} is a set of DAG's, among the elements of $\mathcal{P}_{\mathcal{ML}}$. Augmenting \mathcal{H} with the ILI links, a taxonomic network, $\mathcal{H}_{\mathcal{ML}}$, is created. This $\mathcal{H}_{\mathcal{ML}}$ is used for the implementation of MLSemJoin, as follows:

Definition 4.4 [MLSemJoin Matching]: Given the multilingual taxonomic hierarchy

$\mathcal{H}_{\mathcal{ML}}$, $\{w_i \text{MLSemJoin } w_j\}$ is true, if $\{P_I \text{ is-possibly-a } P_J\}$ under $\mathcal{H}_{\mathcal{ML}}$. In this definition, $P_I = \mathcal{M}_{\mathcal{I}}(w_i)$ and $P_J = \mathcal{M}_{\mathcal{J}}(w_j)$, where \mathcal{M}_X is the lexical matrix function of the language X .

Note that Definition 4.4 guarantees not to produce *false-dismissals* (within the context of WordNet), though it may introduce *false-positives* by matching on unintended word-senses.

Example 4.4: The predicate (Bird MLSemJoin Fauna) evaluates to true, as it is the same as (Bird *is-possibly-a* Fauna). Similarly, the predicate (Bird MLSemJoin Artifact) also evaluates to true. \diamond

Example 4.5: Consider the predicate (English “Bird” MLSemJoin German “Kunstprodukt”). The answer is evaluated as, $Is \{Bird_1, Bird_2\} \cap \{Kunstprodukt, Machanische Vorrichtung, Flugzeug, Bird_2, Artifact, Mechanical Device, Flying Machine\} \neq \phi$, which evaluates to true. \diamond

Example 4.6: Under Definition 4.4, consider the following canonical MLSemJoin query predicate:

$$\{Attr\} \text{MLSemJoin } \{\text{Const } c\} \text{InLanguages } L_1, L_2, \dots, L_N$$

Let L_c denote the language in which the constant **Const** is specified, and let $L_{out} = \{L_1, L_2, \dots, L_N\}$. Then, $\mathcal{M}_{L_c}(c)$ denotes the set of semantic primitives corresponding to the constant c in language L_c . Then, $\mathcal{T}_{\mathcal{H}_{\mathcal{ML}}}(\mathcal{M}_{L_c}(c))$ denotes the transitive closure of the semantic primitives corresponding to c in L_c , under the taxonomic network $\mathcal{H}_{\mathcal{ML}}$. Let, $\Pi_{L_{out}}(\mathcal{T}_{\mathcal{H}_{\mathcal{ML}}}(\mathcal{M}_{L_c}(c)))$ be the set of semantic primitives in the languages in which the output is desired ($= \{s | s \in \cup_i S_I, L_I \in L_{out}\}$). Further, let the value of the attribute, in the database tuple currently under consideration, be denoted by d , its language by L_d , and the set of semantic primitives of d with respect to L_d , by $\mathcal{M}_{L_d}(d)$. With this notation, the basic MLSemJoin returns true iff $\mathcal{M}_{L_d}(d) \cap \Pi_{L_{out}}(\mathcal{T}_{\mathcal{H}_{\mathcal{ML}}}(\mathcal{M}_{L_c}(c))) \neq \phi$. \diamond

MLSemJoin (*StringData*, *StringQuery*, \mathcal{L}_D , \mathcal{L}_Q , *match*, \mathcal{T}_L)

Input: *StringData* and *StringQuery* in languages \mathcal{L}_D and \mathcal{L}_Q
Flag *match*, Target Languages \mathcal{T}_L

Output: TRUE or FALSE, [Optionally] Gloss of Matched Synset

1. $\mathcal{S}_D \leftarrow \mathcal{M}_{\mathcal{L}_D}(\text{StringData});$
2. $\mathcal{S}_Q \leftarrow \mathcal{M}_{\mathcal{L}_Q}(\text{StringQuery});$
3. **if** *Match* **is** EQUIVALENT **then**
4. **if** $(\mathcal{S}_D \cap \mathcal{S}_Q) \neq \phi$ **return** true **else** **return** false;
5. **else if** *Match* **is** GENERALIZED **then**
6. $\mathcal{T}\mathcal{C}_Q \leftarrow \text{TransitiveClosure}(\mathcal{S}_Q, \mathcal{T}_L);$
7. **if** $(\mathcal{S}_D \cap \mathcal{T}\mathcal{C}_Q) \neq \phi$ **return** true **else** **return** false;
8. [Optionally] **return** Gloss of the Matched Synset in a Parameter;

TransitiveClosure (*S*, \mathcal{T}_L)

Input: Set of String *S*, Target Languages \mathcal{T}_L

Output: The transitive closure of *S*

1. $\mathcal{S}_C \leftarrow \mathcal{S};$
 $\mathcal{S}_N \leftarrow \phi;$
2. **repeat until no change in** *S*:
3. **for every element** *s* **in** \mathcal{S}_C
4. $\mathcal{S}_N \leftarrow \mathcal{S}_N \cup \text{Synsets that are subclasses of } s \text{ within the Language } L_s$
 $\cup \text{Synsets linked to } s \text{ through } \|\cdot\| \text{ to a new } L \in \mathcal{T}_L ;$
5. $\mathcal{S} \leftarrow \mathcal{S} \cup \mathcal{S}_N;$
 $\mathcal{S}_C \leftarrow \mathcal{S}_N;$
 $\mathcal{S}_N \leftarrow \phi;$
6. **return** *S*

Figure 4.2: The MLSemJoin Matching Algorithm

4.4 MLSemJoin Matching Algorithm

The skeleton of the MLSemJoin algorithm is shown in Figure 4.2. The MLSemJoin takes two multilingual strings to be matched, along with their language identifiers². A user-specified flag, *match*, and a set of target languages for matching, are also input. The MLSemJoin functionality needs two significant steps (distributed among lines 3 through 7): First, the computation of the transitive closure of the synsets of the RHS operand in the interlinked WordNet taxonomic hierarchy, and second, the testing of non-empty intersections between the sets of synsets corresponding to the LHS operand values and the computed transitive closure of the RHS operand. While the second step may be implemented efficiently using hash-table based approaches, the first step is recognized to be inefficient in relational systems [3, 55, 63]. The TransitiveClosure algorithm computes the transitive closure of a set S in the interlinked WordNet taxonomic hierarchy $\mathcal{H}_{\mathcal{ML}}$. Note that in line 4 of the function, the augmentation is done only for $||L$ links going into a target language that has not been visited yet and not to all languages, for the following reason: The computation of the transitive closure in the traditional sense reduces precision of the result set with every traversal across languages (due to the multiple senses for a given word). Due to our modification in computing the closure of the RHS operand value, the closure computed by the algorithm in Figure 4.2 may be slightly different from the traditional one. However, we expect the result set to be more precise, as it takes into consideration the alternate meanings of a query term, only for the first semantic mapping into the target language set. As a useful side effect, it keeps the growth of $\mathcal{S}_{\mathcal{N}}$ linear in the number of languages in the $\mathcal{T}_{\mathcal{L}}$ list, hence the computation is more efficient.

²The language identifiers are explicitly provided since identification of a language is not possible at the attribute level. In Chapter 5, we propose a new datatype which stores the language identifier explicitly, along with the value string.

4.4.1 Derived Operator Approach

While the above algorithm may be implemented as a UDF, we chose to use standard SQL:1999 features for implementing `MLSemJoin` as a derived operator in commercial database management systems. The reasons for this approach are as follows: First, the UDF call overheads affect the performance adversely, as previously observed in the implementation of `MLNameJoin` operator; significantly, computation of transitive closure itself is an expensive operation in relational systems which may be further affected by UDF implementation. Second, the availability of SQL features offers an efficient solution that may also leverage on the optimizer for selection of better execution plans.

In our derived operator approach, the transitive closure of the *StringQuery* on WordNet taxonomic hierarchy is computed using the recursive SQL constructs; once computed, the standard `IN` clause is used for testing if the LHS operand is a member of the computed transitive closure. An `MLSemJoin` query is transparently re-written to a standard SQL query using the above constructs and executed in the relational system. The results from the SQL query are output to the user, with no modification.

The use of standard SQL constructs helps in leveraging the efficient implementations and optimization opportunities afforded by the well-tuned relational optimizer. Further, this approach indicates that the `MLSemJoin` operator may be implemented on an existing relational database system. However, in this derived operator approach, some obvious performance optimization measures, such as, generating the closure only up to the point to determine set membership of the LHS operand, may not be possible. Further, relevance ranking of the results is also not possible, as the results from the standard SQL query are output to the user with no modification.

4.4.2 Following Through with an Example

We present an example to illustrate the *derived-operator* implementation of the `MLSemJoin` function. The WordNet resource is stored in the \mathcal{W}_L table. The user query,

```

SELECT Author, Title FROM Books
WHERE Category MLSemJoin ALL 'History'
INLANGUAGES {English, French, Tamil}

```

is mapped to the following query, where the transitive closure on $\mathcal{W}_{\mathcal{L}}$ is computed using the recursive SQL constructs and the set membership is tested by the IN predicate:

```

WITH Descendants (child, lang)
  (SELECT  $\mathcal{W}_{\mathcal{L}}.sub$ ,  $\mathcal{W}_{\mathcal{L}}.lang$  FROM WordNet  $\mathcal{W}_{\mathcal{L}}$ 
   WHERE  $\mathcal{W}_{\mathcal{L}}.super$  = 'History'
   AND  $\mathcal{W}_{\mathcal{L}}.lang$  IN ('ENGLISH', 'FRENCH', 'TAMIL'))
UNION ALL
  (SELECT  $\mathcal{W}_{\mathcal{L}}.sub$ ,  $\mathcal{W}_{\mathcal{L}}.lang$  FROM WordNet  $\mathcal{W}_{\mathcal{L}}$ , Descendants Dec
   WHERE  $\mathcal{W}_{\mathcal{L}}.parent$  = Dec.child AND  $\mathcal{W}_{\mathcal{L}}.lang$  = Dec.lang)
SELECT Author, Title FROM Books
WHERE Category IN (SELECT child FROM Descendants)

```

Thus, the user query effectively translates to the following SQL query:

```

SELECT Author, Title FROM Books
WHERE Category IN {'History', 'Memoir', 'Autobiography', ...
  'Histoire', 'Mémoire', 'Autobiographie'...
  'சரித்திரம்', 'நினைவுகள்', 'சுயசரிதம்'...}

```

Here, the values in the IN clause are the subclasses of **History**, in English WordNet, and their equivalents in French and Tamil WordNets. Note that any conjunction (disjunction, respectively) of MLSemJoin predicates can be handled by computing the intersection (union, respectively) of closures for the IN predicate.

4.5 MLSemJoin Performance

In this section, an experimental setup to measure the performance of the `MLSemJoin` derived operator is presented, along with the performance of the operator on a set of popular database systems.

4.5.1 System and Database Environment

A standard Intel Pentium IV (1.7 GHz) workstation with 256MB memory running Windows 2000 Professional operating system, was used as the experimental platform. Three popular database systems – Oracle 9i Database Server (Version 9.0.1), IBM DB2 Universal Server (Version 7.1.0) and Microsoft SQL Server (Version 8.0.194) – were installed with default configurations and their performance on `MLSemJoin` operator was studied. The database systems are identified in the performance section randomly, as *A*, *B* and *C*, to protect their individual identities. The `MLSemJoin` operator itself was implemented in recursive SQL in the IBM DB2 Universal Server (using `WITH` clause) and in Oracle 9i Database Server (using `CONNECT BY` clause). In Microsoft SQL Server, the recursive functionality is not supported natively, hence we implemented the closure computation using SQL scripts.

4.5.2 Dataset

The entire set of noun taxonomic hierarchies of WordNet (Version 1.5), totaling about 110,000 word forms, 80,000 synsets and about 140,000 relationships between them, was loaded into each of the database systems, in a simple hierarchy table (as `Parent-Child` relationships). All the dependent information (such as, *Gloss*, the textual description) was stored separately, in order to keep the hierarchy table compact. The needed storage space for storing English WordNet in the default ISO:8859 character set is approximately 4 MB (including index storage).

Since at this point of time the English WordNet is the most developed, and various

WordNets are at different stages of development, we used the following strategy to simulate the interlinked multilingual WordNet resource for our performance experiments: We first compared the structural characteristics of the current versions of Euro and Indo WordNets with English WordNet [15, 37], and the results are given in Table 4.1.

Characteristic	English	French	German	Spanish	Hindi
<i>Word Forms (Words)</i>	114,648	32,809	20,453	50,526	22,522
<i>Word Sense (Synsets)</i>	80,000	22,745	15,132	23,378	7,868
<i>Average Synsets per Word Form</i>	2.24	2.18	2.30	2.36	3.89
<i>Average Synset Span-out</i>	1.99	1.44	1.35	2.16	2.29
<i>Equivalence Relations per Synset (to English Synsets)</i>	1.00	0.99	1.08	0.91	N/A

Table 4.1: Statistical Profile of WordNets

The statistics indicate a very close match between the structural characteristics (such as, average span-out) of different WordNets. In addition, since both Euro and Indo WordNets have conformance to English WordNet as their stated goal, it is reasonable to expect that their structures will be similar to that of English WordNet, when fully developed. Assuming that the WordNet of each language will be similar to that of English when fully developed, English WordNet was replicated in Unicode format and 1:1 links between every English synset and its corresponding synset in the replica were created, with a probability of 0.95, closely matching the equivalence relations of existing WordNets to English WordNet. The resulting taxonomic hierarchy is used in the performance experiments. Note that the replicated WordNets were stored in Unicode format, for an accurate modeling of the multilingual resources; each of these WordNets requires about 8 MB of storage space (including index storage).

4.5.3 Query Workload

For profiling the performance of the `MLSemJoin` operator, a set of queries that compute closures of varying sizes in the WordNet taxonomic hierarchy, using recursive SQL constructs (as shown in Section 4.4.2) were used. The input strings for the queries were

chosen such that their closure cardinalities are from a few tens to a few thousands of synsets. Such selection of query terms provides a sufficiently wide range for calibrating the performance of the `MLSemJoin` query, when closures of different sizes are computed. Table 4.2 shows some of the query terms used for calibration, and the corresponding closure sizes in English WordNet.

Semantic Primes	Size of Closure
<i>Hotel, Restaurant</i>	67
<i>Sex</i>	78
<i>Baby, Children, Kids</i>	107
<i>Profession, Job, Career</i>	298
<i>Business, Company, Organization</i>	488
<i>Music, Song</i>	548
<i>Artist, Creator, Performer</i>	862
<i>Education, Training</i>	969
<i>Food, Drink</i>	2,570
<i>Fauna, Animal</i>	4126
<i>Flora</i>	4955
<i>Knowledge, Subjects</i>	5340
<i>Human, Person</i>	11551

Table 4.2: Closures for English Word Forms

To establish the *likely* closure size (*i.e.*, the average closure size for likely query strings), we selected the top-hundred most used nouns in English [11] and the top-fifty nouns that are used in popular web-search engines [134], and computed the average of *their* closure-sizes in English WordNet, which resulted in an average closure size of around 625. Hence, assuming that a multilingual user would typically want answers in about three languages, it is realistic to use a figure of around 2,000 for a representative closure size for a typical multilingual query.

4.5.4 Performance Metrics

In all the experiments, the wall-clock runtime of a given query with a specific query term from the list given in Table 4.2 was measured. The queries were run in an SQL or a

programming language environment, as appropriate. Before each experiment, the test machine was quiesced and only the database system being tested and allied processes were allowed to run in order to have measurement parity between the systems. The average runtime from several identical runs was taken as the runtime of a specific query (the graphs show mean values with relative half-widths about the mean of less than 5% at the 90% confidence interval).

It should be noted here that the *quality* of the retrieval is determined solely by the coverage (for *recall*) and the resolution power (for *precision*) of the taxonomic hierarchy used for semantic equivalence. While precise taxonomic hierarchies such as *Gene-Ontology* [45] are expected to have perfect recall and precision, the WordNet taxonomic hierarchy has low resolution power, due to the polysemic words that lead to low precision of the result set. In our experiments, the quality of the match was not measured, since most WordNets are not fully developed yet and hence not available for experimentation. Further, measurement of such quality in the linguistic domain is beyond the scope of our research, which focuses solely on optimizing the database performance, given the linguistic hierarchies.

4.6 Performance Results and Analysis

In this section, the performance of a suite of popular database systems in executing the MLSemJoin functionality, as a derived-operator, is presented, and subsequently, optimized.

4.6.1 Baseline MLSemJoin Performance

For the baseline performance experiments, the interlinked WordNet taxonomic hierarchy as discussed in Section 4.5.2, was stored and queried. The queries that compute different closure cardinalities, as given in Section 4.5.3, are run. The *SQL-Baseline* performance (in seconds) for the basic closure computation in the three commercial database systems, is given in Figure 4.3 (the graph is shown in *log-log* scale). For each system, the

performance with and without the B+ tree index on the elements of the hierarchy \mathcal{H} is provided. As can be observed, the closure computations are very slow in all the systems under study, taking up to hundreds of seconds without an index and up to a few seconds even with an index.

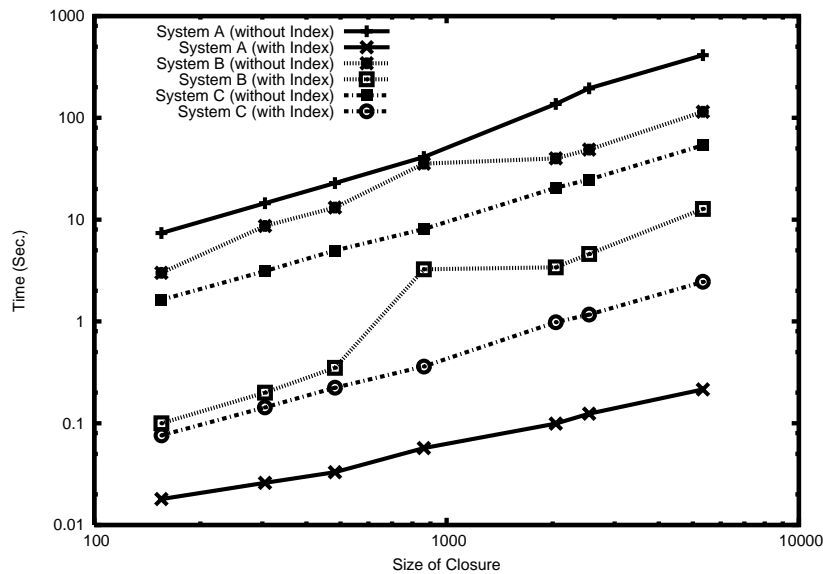


Figure 4.3: Baseline Performance of Computing Closure

The run-times in different database systems are influenced by their implementation methodology for computing transitive closures. For example, two systems used *breadth-first-search* for expanding the result set, while the third used *depth-first-search*. One system expanded hierarchies a level at a time, thus reducing the overall number of joins, resulting in smallest run times of the three. Further, one of the systems detected cycles during traversal and exited gracefully, indicating that it maintains an internal hash-table (or a sorted list) to make the *in-progress* closure unique. The other two systems ran indefinitely, indicating the absence of such a check. This observation is confirmed when the systems report different closure sizes, in the presence of nodes that have multiple paths from an ancestor (in which case, some of the systems multiply count the descendant nodes). The query execution plans indicated that indexes on the taxonomic table were made use of whenever available, in all the systems. However, no optimizations (such as, sorting, hash-indexes or maintaining incremental views) were used for efficient scanning

of the temporary *in-progress* closure table.

In summary, irrespective of wide variation in the performance of the different systems, none of them exhibit a performance suitable for practical use, if the size of the closure exceeds a few hundred items. Hence, in the following sections, three different optimization techniques to improve the performance of the closure computation, a necessary component for implementing the *MLSemJoin* operator, is presented. We choose *System B*, which exhibits the worst performance among the three studied, as the candidate for subsequent optimizations.

4.6.2 Optimization #1: Precomputed Closure

First, a standard optimization technique of *pre-computing* the closures of every element in the WordNet hierarchy and *storing* them explicitly as the immediate children of the corresponding element, was used. Thus, the closures could be found with a simple scan of the enhanced table. To further reduce the cost of computation, a clustered index was built on the parent attribute of the pre-computed table. The transitive closure queries were run on the resulting data set, and the performance, with and without the index, is presented in Figure 4.4 (where the graph is shown in *log-log* scale).

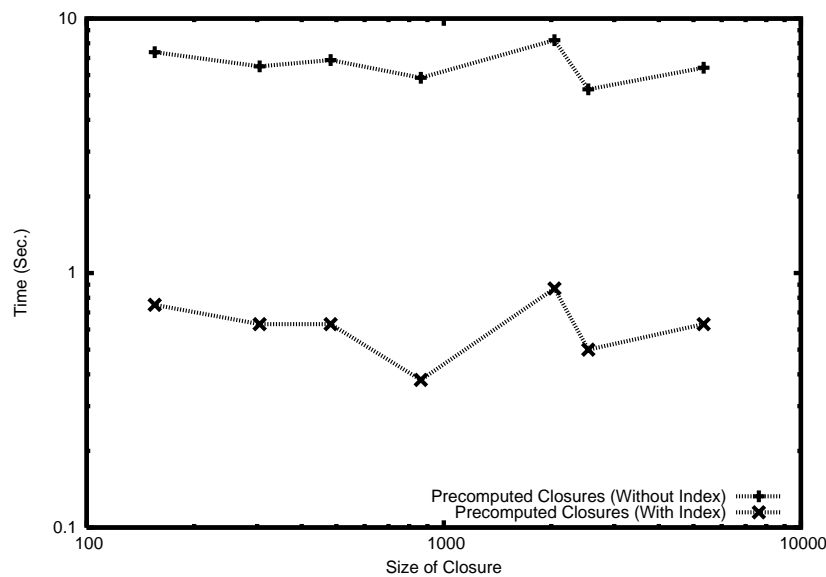


Figure 4.4: Closure Performance with Precomputed Closures

We observe here an improvement in performance, to approximately 7 seconds (without index) for computing any closure cardinality. Understandably, the closure computation takes approximately the same time, since it can be computed with a single scan of the table storing the precomputed closures. With the clustered index, as expected, the runtime is much better, improved by an order of magnitude from the baseline index performance, to under one second. However, this gain comes with the penalty of enormous storage costs: the space required for the taxonomic tables with precomputed closures increased by about 30 times, to roughly 120 MB.

4.6.3 Optimization #2: Reversed Traversal

Next, an alternate strategy that improves the run time with no space overheads, was pursued. First, the implementation of the `MLSemJoin` functionality was modified to traverse the taxonomic hierarchy in the *reverse* order; that is, the transitive closure of LHS operand traversing the ancestor links was computed, and a check was made to verify if the RHS is a member of the ancestral closure of the LHS operand. Given that the average *in degree* in the WordNet taxonomic hierarchy is 1.124, which is significantly smaller than the average *out degree*, a better performance, due to the smaller size of the ancestral closure, may be expected. A closure computation is still needed as the WordNet taxonomic hierarchies are not strict trees. The performance of the modified `MLSemJoin` is given in Figure 4.5 (where the graph is shown in *log-log* scale).

As expected, the computation of the closure was improved by 2 orders of magnitude from the baseline, to a few seconds, without index and to under a second, with index. It is interesting to note that the graph in Figure 4.5, merely corresponds to the lower end of Figure 4.3, since the computation is done on the same hierarchy table, with the primary difference being the smaller sized closures that are being computed. However, this optimization methodology suffers from a major drawback: while the normal closure corresponding to the constant RHS needs to be computed only *once* in the forward direction, this optimization requires that the reversed closure be computed *once per LHS value*, resulting in numerous closure computations. Hence for a canonical `MLSemJoin`

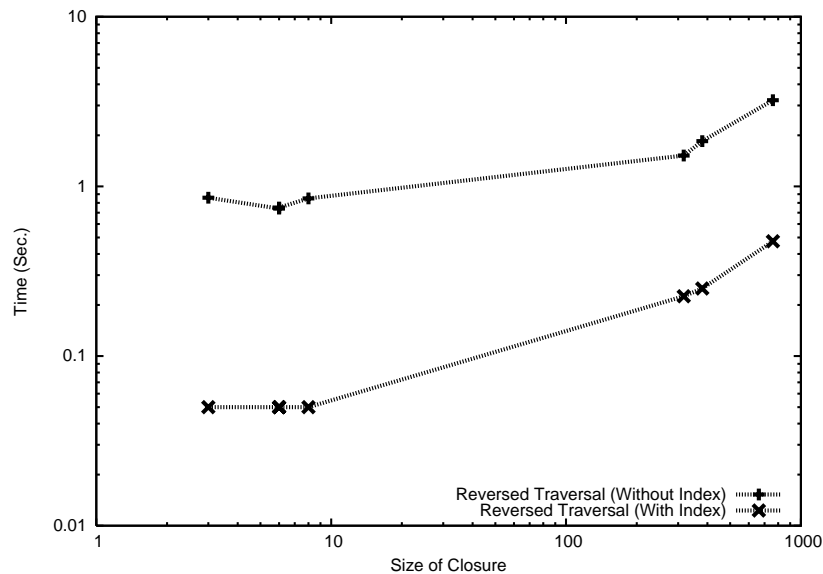


Figure 4.5: Closure Performance with Reversed Traversal

operator for which the LHS values are from a table attribute and the RHS value is a constant, this optimization may prove to be expensive.

4.6.4 Optimization #3: Reorganizing Schema

Finally, a third performance optimization strategy that is based on leveraging the *distribution* of synsets in the WordNet hierarchy to reduce the calls to the expensive recursive SQL statement, is pursued. First, the span-out of every node in English and Hindi WordNets is computed and plotted. The plot of the span-outs, shown in Figure 4.6, exhibit a characteristic *power-law* distribution (*with an exponent of -2.75*). More interestingly, the Hindi and English WordNets exhibit a very similar span-out profile differing only in scale, suggesting the applicability of power-laws in linguistic domains.

Further analysis indicated that only a small number of synsets (*less than 10%*) have a large number of children (*more than 16*), with the large majority having only a few children. This distribution suggests a new, more efficient organization of WordNet hierarchy, where a majority of the sub-classes may be *in-lined*, along the lines of optimization that is done in XML arena [28]. We chose to in-line those synsets with up to 16 subclasses in an in-lined taxonomy table, reducing the number of records in the new in-lined taxonomy

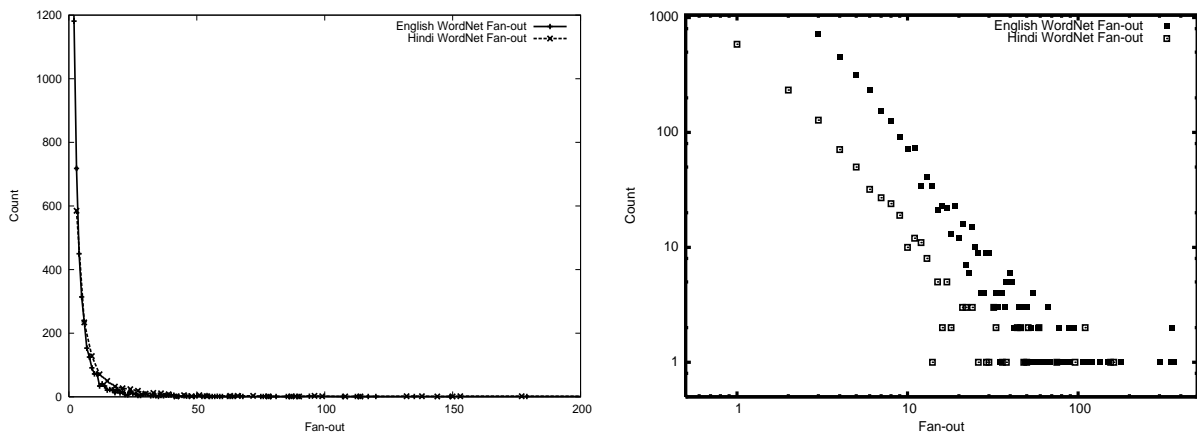


Figure 4.6: Fan-out Histogram and Plot

table to about a tenth of that of the original table. All synsets with greater than 16 subclasses remained in the original table. The storage size of the combined original and the in-lined tables was about 8 MB (when stored in Unicode format), approximately the same as that in the baseline experiments. The closure computation algorithm is modified to take care of the schema changes: the access is made to the in-lined table for all synsets with up to 16 children, or to the original table, for those synsets with more than 16 children. For the above schema, the performance of the closure queries – with and without indexes – are shown in Figure 4.7 (where the graph is shown in *log-log* scale).

As can be observed from Figure 4.7, the performance with reorganized schema is improved by about 2 orders of magnitude on the plain table, and by about 3 orders of magnitude on the indexed table, with *no perceptible increase in storage requirements from the baseline*. Further, for a closure computation of about 2,000, the runtime is approximately 25 milliseconds (with index on the taxonomic hierarchy), which is commensurate with the requirements for user on-line interaction.

4.6.5 MLSemJoin Performance with Scaling of Languages

Finally, we explore how the performance scales with increase in the number of languages that are considered for query processing. Simulated experiments, varying the number

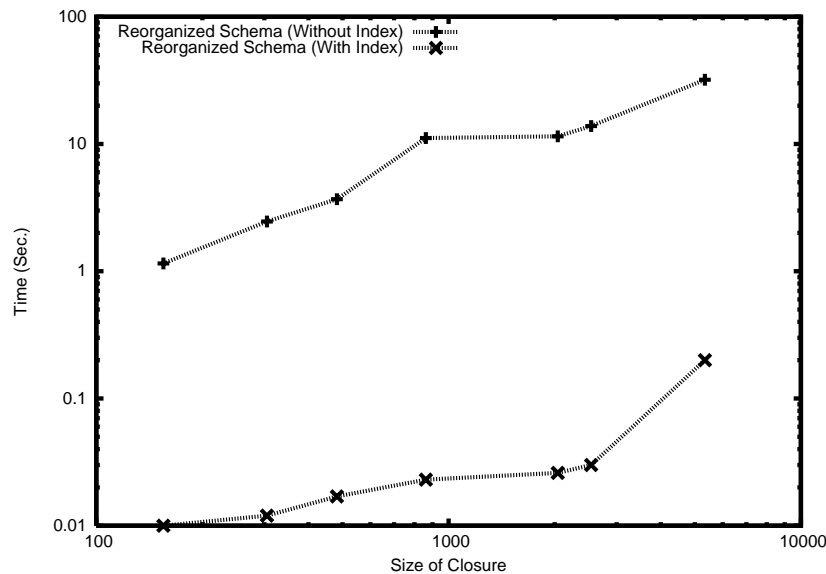


Figure 4.7: Closure Performance with Re-Organized Schema

of languages from 1 to 8, were conducted. For each experiment an interlinked WordNet hierarchy with required number of languages (say, n), was created as follows: the English WordNet was replicated n times, and the lll links between the replicas created using a similar methodology as outlined in the performance section. Since the typical query term requires a closure size of 625 *per language*, in these experiments runtime to compute closure sizes of approximately $(625 * n)$ is measured. Further, for each experiment, both the *Precomputed Closure* and *Reorganized Schema* optimizations were implemented, and their respective performances measured. The results are shown in Figure 4.8 (where the graph is shown in *log-linear* scale).

In Figure 4.8, we observe that the query processing time does not grow exponentially with the number of languages. A linear approximation holds well for the tested performance range, for both *pre-computed closure* and *re-organized tables* methodologies. The increase in run times is attributed to two factors: the increasing sizes of the hierarchy tables proportional to the number of languages, and the increasing cardinality of the closures being computed, which are proportional to the number of languages. Though there is an exponential increase in the lll links in the hierarchy table, their traversals are bounded by restricting the transitive closure expansion to be within the target languages

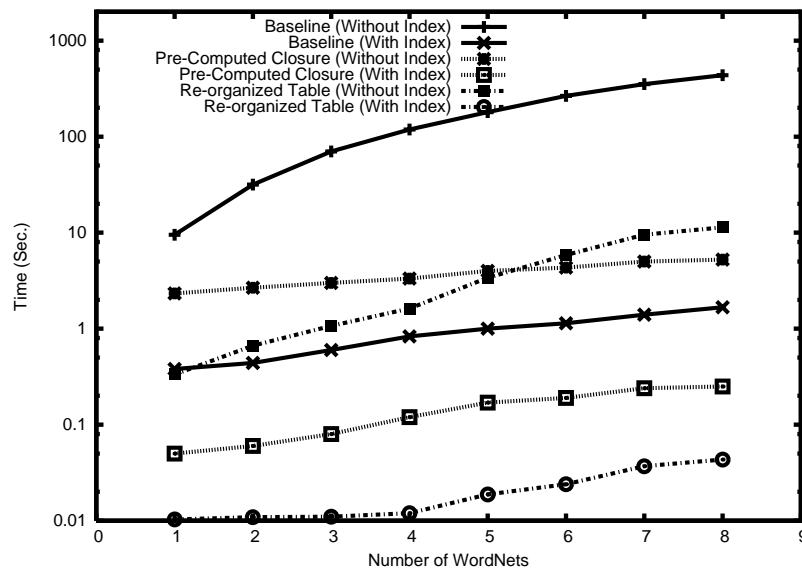


Figure 4.8: Closure Performance with Number of Languages

only. In summary, the index-based run times for the typical query remained within a few tens of milliseconds, even with about 8 languages, which appears sufficiently small to support online user interaction for a multilingual user.

Thus, we show that the `MLSemJoin` functionality may be implemented on unmodified relational database systems by integrating standard linguistic resources, and leveraging only on existing SQL features. Further, we show that the performance of this matching may be sufficiently optimized to support online-user interactions for multilingual e-commerce applications.

4.7 Related Research

To the best of our knowledge, multilingual semantic matching of attribute data – by integrating standard linguistic resources with the database engine, has not been discussed previously in the database literature.

Currently, all commercial database systems support some flavour of semantic query processing, based on NLP techniques. However, since no standards have been specified in SQL [59] for semantic query processing, each vendor [100, 86, 56] has taken an unique approach for such querying. As a result, the same query on the same document collection

could yield different answer sets in each of the systems. Further, such NLP techniques are applied to document data, and not to attribute-level data. Finally, even considering only queries on documents, such semantic querying is applied on a *per-language* basis, with no cross-lingual matches possible. Our approach could be used for cross-lingual semantic retrieval of attributes (as proposed), or of documents (by appropriately modifying our technique to query on the elements of the inverted index, which is a collection of multilingual key-words pointing to documents in multiple languages).

Recently a major database vendor proposed an ontological matching operator [25] to support query processing based on user defined ontologies. While their operator is not yet available in released versions, the operator – when available – may be adopted for multilingual semantic matching, by using interlinked WordNet taxonomic hierarchies. However, some of the optimizations proposed in this thesis may be suitable candidates for improving the performance of their operator, as the reported performance in [25] may not be practical for query processing with large ontologies (such as, WordNet). The reorganized schema optimization proposed in this thesis may provide a practical space-time trade-off for such ontological query processing. Nevertheless, we are heartened by this parallel effort by a commercial database system vendor, as it validates our intuition about the need for such novel matching methodologies in relational systems. More recently, in [110], Hopi, a connection index to compute the descendants of a node efficiently in an XML collection has been presented. This work could be extended for closure computation in any taxonomical hierarchy and hence may present an interesting optimization possibility for MLSemJoin implementation; we hope to explore this possibility in the future.

International WordNet initiatives [65, 18, 19, 37], with a stated objective of following near-identical taxonomical structures, are enabling resources for realizing our implementation. The WordNet based approach was used for semantic information retrieval in [107], where the emphasis was on the *quality* of the results and not on performance; our work on performance of such retrievals is complementary to this research. Similar approaches have been attempted in other languages as well: in [114] encouraging results

(up to $\approx 70\%$ accuracy) in noun sense disambiguation using Indo WordNet on Hindi corpora was reported. A practical system for crosslingual semantic search was reported in [68], where the Universal Networking Language (UNL) [129] based search strategy of matching expression graphs, was pursued. In our work, we did not pursue UNL based crosslingual semantic matching as the MLSemJoin deals with attribute level information and hence is closer to key-word matching than expression graph matching.

There are vast amounts of literature in the Information Retrieval (IR) Research community in the areas of Knowledge-based and Natural-language based retrieval. The techniques employed are diverse, ranging from syntactic and morphological analysis [41] to Machine Translation [36], statistical techniques [46], and Latent Semantic Indexing [27] for semantic querying in a single language, and to paired dictionaries [115] techniques for handling cross-language querying. We refer to the Multilingual Information Retrieval Track of the ACM SIGIR conference [116] for a survey of current techniques. Such techniques are more suited for document level processing and not suited for attribute level processing in relational systems. Also, in-line with our design goals for consistent query processing across relational systems, only on standard linguistic resources for query processing, were relied upon.

4.8 Conclusions on Multilingual Semantic Matching

In this chapter, we proposed a new multilingual functionality – MLSemJoin – to support seamless multilingual matching of attributes that store categorical data based on semantics. The proposal outlines an implementation of this feature, by adopting and integrating the WordNet linguistic resources in the database system. Multilingual text attribute data are matched after transforming them to a canonical semantic form (that is, *synset* of the corresponding WordNet), and leveraging the rich cross-linked taxonomic hierarchies of different language-specific WordNets.

We outlined a *derived-operator* approach for implementing the MLSemJoin operator, using standard SQL:1999 constructs. Our performance experiments with real WordNet

data on three popular database systems, underscored the inefficiencies in computing transitive closure, an essential component for semantic matching. The run times are in the order of a few seconds, unsuitable for practical deployments. We showed that by tuning the storage and access structures to match the characteristics of linguistic resources, the closure computation may be improved up by nearly 3 orders of magnitude – to *a few milliseconds* – to make the operator efficient enough for supporting online user query processing.

The following summary may provide a proper perspective on the operator performance with various optimization strategies: the baseline performance of `MLSemJoin` operator with *Education* (which includes the subject taxonomic hierarchy shown in Section 4.4.2) as RHS operand is approximately 40 seconds (without index), and approximately 2 seconds (with index). With pre-computed closure optimization, the performance figures are 8 seconds (without index) and 0.4 seconds (with index); however the storage required for WordNet hierarchy went up from the baseline figure of 4 MB to approximately 120 MB. The reorganized schema optimization produced the best performance figures of 9 seconds (without index), and 0.02 seconds (with index), with a less than 2% increase in storage size from the baseline figure of 4 MB.

These results underscore the viability of the `MLSemJoin` operator for immediate practical use, using unmodified commercial relational database systems. As a side effect, such a methodology provides repeatable and consistent results for a given data set across different database systems. Finally, we expect that for specific applications, semantic matching using domain-specific ontological hierarchies may also benefit from a similar approach to that outlined here.

Chapter 5

A Multilingual Operator Algebra

5.1 Overview of the Chapter

In the previous chapters, two multilingual join functionalities were proposed and their implementation methodologies on commercial database systems were presented. In this chapter, we put these proposals together in a holistic query processing architecture for multilingual attribute data, by formalizing the multilingual functionalities as operators and presenting an operator algebra that ties them together in a unified framework. The required composition rules, cost models and selectivity estimations, for a native implementation of the architecture in relational database systems, are also presented.

5.2 Mural: Multilingual Relational Algebra

In this section, an operator algebra – Multilingual Relational ALgebra (Mural), along with a new datatype `Uniform` to store the multilingual data and a few `Uniform`-specific operators along the lines of the multilingual join functionality presented in the previous chapters, is presented. For the sake of analysis, simplified versions of `MLNameJoin` and `MLSemJoin` functionalities are assumed.

5.2.1 Uniform: A Multilingual Text Datatype

All the basic types of relational systems are preserved in the Mural algebra except the Text datatype, which is replaced by Uniform (Unicode FORMat) datatype, specifically to hold the multilingual text strings. The Uniform datatype implementation parallels the Cuniform representation presented in Chapter 2. The Uniform datatype is a 2-tuple, where the first part is a text string in a standardized encoding¹ (referred to as Text) and the second part is an identifier² for the language of the string (referred to as LangID). A unique language identifier is possible only if the text string is in a single language. Unique values for LangID are allowed for unclassified strings (as, Unknown) or not uniquely classifiable strings (as, Mixed).

An explicit identifier is necessary, as several languages share a common script and a string may have different pronunciations or meanings, depending on its language. It should be noted here that automatic language detection is not, in general, possible with attribute level datum, though it is possible for languages that have a one-to-one mapping to a script.

Example 5.1: While ‘Sample String’, ‘Une Corde Témoïn’ and ‘உவமான சரம்’ are examples of Unicode strings, <‘The SQL Standard’, English>, <‘El Estándar del SQL’, Spanish> and <‘SQL தரம்’, Mixed> are examples for the Uniform datatype. ◊

Decomposing and Composing Uniform Datatype

It is apparent that with Uniform datatype as above, standard database operations need to be redefined to work on the new datatype. First, two simple operators on Uniform datatype are introduced; specifically, a *Composing Operator* (denoted as, \succ) that can compose a Uniform datatype out of a given Unicode string and a language identifier and a corresponding inverse *Decomposing Operator* (denoted as, \prec) that decomposes a Uniform data to a Unicode string and a language identifier. These two operators – \succ and \prec –

¹For the current implementation, we used Unicode as the encoding format for Text part of Uniform. The skinned Unicode representation may facilitate performance improvements, as outlined in Chapter 2.

²To enhance readability, all the examples present the identifier as an English string, such as English, French, etc.; in actual implementation, it is assumed to be an identifier.

may be implemented in a fairly straight forward manner.

Example 5.2: Given a piece of data, (*Name* Uniform), it may be flattened into a 2-tuple of basic datatypes, ($\{Name_{Text} \text{ Text}, Name_{LangID} \text{ ID}\}$). Similarly, given two appropriate pieces of data, a datum of Uniform datatype may be constructed, by nesting them. For example, consider a relation (TextID Integer, TextString Uniform). Instances of this relation may be $\{\langle 1, \langle \text{'A Sample String'}, \text{English} \rangle \rangle, \langle 2, \langle \text{'Une Corde Témoin'}, \text{French} \rangle \rangle\}$ in Mural, and may become $\{\langle 1, \text{'A Sample String'}, \text{English} \rangle, \langle 2, \text{'Une Corde Témoin'}, \text{French} \rangle\}$ in normal database schema. In essence, the MURAL datatype may be thought of as a composite datatype obtained by the nesting of the Unicode value of the multilingual data and the explicit identifier for the language. \diamond

Normal Text Operators on Uniform

All simple text comparison operations (such as, $=, \neq, <, >$, etc.) applied to Uniform datatype operate on the text component of the decomposed Uniform datatype. Specifically, the expression ($a R b$), where R is one of the normal text operators applied on a pair of Uniform datatypes a and b , is equivalent to $((\Pi_{a_{Text}}(\prec(a))) R (\Pi_{b_{Text}}(\prec(b))))$.

Example 5.3: The $(\langle \text{'Gift'}, \text{English} \rangle = \langle \text{'Gift'}, \text{German} \rangle)$ predicate evaluates to true (the strings are lexicographically the same, but semantically are different). The predicate $(\langle \text{'Nehru'}, \text{English} \rangle = \langle \text{'नेहरु'}, \text{Hindi} \rangle)$ evaluates to false (the strings are lexicographically different, but represent the same name). \diamond

It should be noted that while the $=, \neq, <, >$ operations are legal with Uniform strings, the results are meaningless when the strings are from different scripts; the equality always evaluates to false (as it should) and the sorting results depend only on where the scripts corresponding to the languages occur in the Unicode codespace. Such operational semantics are preserved in order for Uniform datatype to be consistent with well-known Text datatype that has this behaviour.

5.2.2 Uniform Equality (Ξ) Operator

A simple lexicographic matching operator (Ξ), on **Uniform** datatype is defined as follows:

$$\Xi : U_1 \times U_2 \rightarrow \langle U_1, U_2, \{\text{true}, \text{false}\} \rangle$$

This operator compares two **Uniform** datatypes and results in a tuple composed of the inputs, tagged with a **true** or **false**. The match is tagged with a **true** if both the components of the 2-tuples match, or tagged with a **false** otherwise. Ξ is a simple algorithm that returns **false** if the languages of the two strings are different. If the languages are the same, then the strings are compared using normal lexicographic semantics. The outline of the Ξ algorithm is as given in Figure 5.1.

```

 $\Xi (U_l, U_r)$ 
Input: Uniform  $U_l, U_r$ 
Output: boolean flag
1. if  $\Pi_{LangID}(\prec (U_l)) \neq \Pi_{LangID}(\prec (U_r))$  then return false
   else if  $\Pi_{Text}(\prec (U_l)) \neq \Pi_{Text}(\prec (U_r))$  then return false
   else return true;

```

Figure 5.1: The Ξ Operator

Example 5.4: The predicate ($\langle \text{'Jean'}, \text{English} \rangle \Xi \langle \text{'Jean'}, \text{English} \rangle$) is **true**, and the predicates ($\langle \text{'Jean'}, \text{English} \rangle \Xi \langle \text{'Jean'}, \text{French} \rangle$) and ($\langle \text{'Gift'}, \text{English} \rangle \Xi \langle \text{'Gift'}, \text{German} \rangle$) are **false**. \diamond

5.2.3 Uniform Names Matching (Ψ) Operator

The multilingual name matching Ψ operator is specified as follows:

$$\Psi : U_1 \times U_2 \rightarrow \langle U_1, U_2, \text{dist} \rangle$$

The input to Ψ is two **Uniform** datatypes, and the output is a tuple composed of the inputs, tagged with the *edit-distance* between the phonemic representations corresponding to the text parts of the input **Uniform** datatypes. Removal of either of the input attributes is by a subsequent projection operation, which is left to the user. The materialization of the phoneme strings corresponding to the multilingual strings is left unspecified³. The

³The algorithmic complexity of Ψ operator implementation is dominated by the **EditDistance** function

algorithm for Ψ is given in Figure 5.2, where a simplified version of the implementation from Chapter 3, is assumed. The `PhoneticTransform` function converts the input multilingual string into its corresponding phonemic string, by using an appropriate TTP resource.

```

 $\Psi$  ( $U_l, U_r$ )
Input: Uniform  $U_l, U_r$ 
Output: integer  $dist$ 
1.    $T_l \leftarrow \text{PhoneticTransform}_{\Pi_{LangID}(\prec(S_l))}(\Pi_{Text}(\prec(U_l)))$ ;
2.    $T_r \leftarrow \text{PhoneticTransform}_{\Pi_{LangID}(\prec(S_r))}(\Pi_{Text}(\prec(U_r)))$ ;
3.   return  $\text{EditDistance}(T_l, T_r)$ ;

```

Figure 5.2: The Ψ Operator

Example 5.5: To select all Authors from table Authors (A) that have names close to Nehru (*match threshold 2*), the query expression is as follows:

$$\Pi_{A.Author}(\sigma_{dist < 2}(A.Author \Psi \{ 'Nehru' \})) \quad \diamond$$

Example 5.6: The expression to join two tables – Authors (A) and Books (B) – based on the author’s name (*match threshold 2*), is as follows:

$$\Pi_{A.Author, B.Author}(\sigma_{dist < 2}(A.Author \Psi B.Author)) \quad \diamond$$

Example 5.7: To select all Authors from table Authors (A) that have names close to Silversmith (*match threshold 2*) and to (*match threshold 3*), the query expression is as follows (Ψ_1 and Ψ_2 are assumed to be the invocations of the Ψ operator in the first and second predicates of the query shown, and $dist_1$ and $dist_2$ are the distance results from the respective invocations.):

$$\Pi_{A.Author}(\sigma_{dist_1 < 2}(A.Author \Psi_1 \{ 'Silversmith' \}) \wedge (\sigma_{dist_2 < 3}(A.Author \Psi_2 \{ 'Aerosmith' \}))) \quad \diamond$$

that has $O(n^2)$ complexity. However, materialization of equivalent phoneme strings at runtime could be very expensive, especially in multi-scan operators, such as, *Nested-Loops* join. Hence, it is advantageous to materialize the corresponding phoneme string and store it explicitly with the multilingual attribute.

The Ψ Operator Properties

The Ψ operator defined as above is functionally analogous to the database equality operator, but with a closeness measure based on the edit-distance between their phonemic equivalents. The following are the properties of this operator.

Property 5.1: The Ψ operator is commutative.

Property 5.2: The Ψ operator commutes with selection, sort and join operators.

Property 5.3: The Ψ operator commutes with projection, provided the attributes used in Ψ are preserved by the projection operator.

Property 5.4: The Ψ operator commutes with aggregate operators, as long as the aggregation preserves the attribute that is used in Ψ operator.

The first property follows immediately based on the definition, and assuming normal semantics for threshold measure. The second through fourth properties follow, since the values in the result set are not altered by the operator.

5.2.4 Uniform Semantic Matching (Φ) Operator

The multilingual semantic matching operator (Φ) is specified as follows:

$$\Phi : U_1 \times U_2 \rightarrow \langle U_1, U_2, \{\text{true}, \text{false}\} \rangle$$

The input to the Φ operator is a pair of Uniform strings, and the output is a tuple composed of the input strings, tagged with *flag*. *Flag* is a boolean value, set to **true** if $u_1 \in \mathcal{T}_{\mathcal{H}}(u_2)$, where \mathcal{H} is the WordNet interlinked taxonomic hierarchy and u_1, u_2 are the semantic atoms corresponding to the Text components of U_1 and U_2 , respectively. Removal of either of the input attributes is by a subsequent projection operation, which is left to the user. The skeleton of the algorithm is as shown in Figure 5.3, which assumes a simplified version of the **MLSemJoin** operator implementation as given in Chapter 4. It should be noted here that the $\Phi_{\mathcal{H}}$ operator is, by definition, asymmetrical and hence is non-commutative; in this simplified version of $\Phi_{\mathcal{H}}$, the operator is used only to verify if the LHS operand is a subclass of the RHS operand, as this variety is the more expensive alternative for the $\Phi_{\mathcal{H}}$ operator. The operator properties are discussed subsequently and

their effect on alternative plan generation is discussed in Section 5.3.

```

 $\Phi_{\mathcal{H}}(U_{Data}, U_{Query})$ 
Input: Uniform  $U_{Data}, U_{Query}$ 
Output: boolean flag
1.  $\mathcal{TC}_{\mathcal{Q}} \leftarrow \text{TransitiveClosure}(\Pi_{Text}(\prec(U_{Query})), \mathcal{H});$ 
2. if  $\Pi_{Text}(\prec(U_{Data})) \in \mathcal{TC}_{\mathcal{Q}} \neq \phi$  then return true else return false;

```

Figure 5.3: The Φ Operator

Example 5.8: The query to retrieve all Books that are categorized under History, may be retrieved as follows:

$$\Pi_{B.BookID}(\sigma_{flag=true}(B.Category \Phi_{\mathcal{H}} \{ \text{'History'} \})) \quad \diamond$$

The Φ Operator Properties

The following are the properties of $\Phi_{\mathcal{H}}$:

Property 5.5: The $\Phi_{\mathcal{H}}$ operator is not commutative.

Property 5.6: The $\Phi_{\mathcal{H}}$ operator commutes with selection, sort or join operators.

Property 5.7: The $\Phi_{\mathcal{H}}$ operator commutes with projection, provided the attributes used in $\Phi_{\mathcal{H}}$ are preserved by the projection operator.

Property 5.8: The $\Phi_{\mathcal{H}}$ operator commutes with aggregate operators, as long as the aggregation preserves the attribute that is used in $\Phi_{\mathcal{H}}$.

Property 5.9: The definition implies that $A \Phi_{\mathcal{H}} B$ is true, iff A is a descendant of B , in \mathcal{H} . Or, equivalently, $A \Phi_{\mathcal{H}} B$ is true, iff B is an ancestor of A , in \mathcal{H} .

The Property 5.5 follows directly from the definition of the $\Phi_{\mathcal{H}}$ operator, based on the fact that the transitive closure is computed only for the right-hand-side operand of the predicate. Hence, in general, $(A \Phi_{\mathcal{H}} B) \neq (B \Phi_{\mathcal{H}} A)$. This property can be used for reducing the plan search space. Properties 5.6 through 5.8 follow from the algebra of the standard relational operators, and provide means of rearranging the $\Phi_{\mathcal{H}}$ operator with other relational operators, in order to generate candidate execution plans without affecting the result set. Property 5.9 follows directly from graph theory (as \mathcal{H} is a set of

DAG's), and suggests an alternative method for implementing $\Phi_{\mathcal{H}}$ operator, by traversing \mathcal{H} in the reverse direction.

5.3 Interaction Between Mural Operators

In this section, an overview of the composition of the new operators, namely, Ξ , Ψ and $\Phi_{\mathcal{H}}$, with each other and with the traditional relational algebra operators, namely \times , σ and π and the aggregation operator Δ , is presented, based on Properties 5.1 through 5.9. The highlights are given in Table 5.1, along with some illustrative examples.

Oper	Commutativity	Associativity	Distributes Over
Ξ	Yes	Yes	$\times, \sigma, \pi, \Psi, \Phi_{\mathcal{H}}, \Delta$
Ψ	Yes	Yes	$\times, \sigma, \pi, \Xi, \Phi_{\mathcal{H}}, \Delta$
$\Phi_{\mathcal{H}}$	No	Yes	$\times, \sigma, \Xi, \Delta$

Table 5.1: Mural Operator Composition Rules

These composition rules are essential for the optimizer, to enumerate and cost alternative plans for a given query, by reordering the multilingual operators Ξ , Ψ and $\Phi_{\mathcal{H}}$ among themselves and/or with other relational operators. Such enumeration of alternative plans is used to choose the most efficient plan, based on estimated execution time.

Example 5.10: The query *to get the addresses of Authors whose names are close to Silversmith (match threshold 3) and whose total book sales exceeded 1M*, may be executed in the following two ways:

Plan 1: $\Pi_{C.Author}(\sigma_{dist \leq 3}(C.Author \Psi \{ 'Silversmith' \}))$

$\rho \rightarrow C(\sigma_{\Sigma(sales) > 1,000,000}(\Delta_{A.Author, \Sigma(sales)}))$

Plan 2: $\Pi_{A.Author}(\sigma_{\Sigma(sales) > 1,000,000}(\Delta_{A.Author, \Sigma(sales)}$

$(\sigma_{dist \leq 3}(A.Author \Psi \{ 'Silversmith' \}))))$

The first plan aggregates every author's book sales and checks if any of those authors with sales of $> 1M$ have a name that is close to *Silversmith*. The second one retrieves

authors whose name is close to **Silversmith** and checks if their book sales exceed 1M. \diamond

Example 5.11: The query to find *all Authors whose names are close to Silversmith (match threshold 3) and who has authored History books*, may be evaluated as follows:

$$\begin{aligned} \text{Plan 1: } \quad & \Pi_{D.Author}(\sigma_{dist \leq 3}(D.Author \Psi \{ \text{'Silversmith'} \})) \\ & \rho_{\rightarrow D}(\sigma_{flag=true}(C.Category \Phi_{\mathcal{H}} \{ \text{'History'} \})) \rho_{\rightarrow C}(A \bowtie_{AuthorID} B) \end{aligned}$$

The above expression can also be transformed to the following equivalent expression, in which the Ψ and the $\Phi_{\mathcal{H}}$ operators are evaluated independently and joined using the standard \bowtie operator. Each plan may have a very different runtime, depending on the size and profile of data in the tables **Author** and **Book**:

$$\begin{aligned} \text{Plan 2: } \quad & \Pi_{A.Author}((\sigma_{dist \leq 3}(A.Author \Psi \{ \text{'Silversmith'} \})) \\ & \bowtie_{AuthorID} (\sigma_{flag=true}(B.Category \Phi_{\mathcal{H}} \{ \text{'History'} \}))) \quad \diamond \end{aligned}$$

5.4 Relational Completeness of Mural

A formal system is said to be *Relationally Complete* [21] if it is at least as powerful as relational algebra, or equivalently, if all relational algebra queries may be expressed in the proposed system. In this section we show that **Mural** is relationally complete.

Lemma 5.1 [Mapping Lemma]: There exists a mapping scheme Ω_{Sch} between a **Mural** schema and a standard relational schema.

Proof: **Mural** has all the datatypes of standard relational algebra, excepting the **Text** datatype (which is replaced by the **Uniform** datatype). Hence, for all schema objects, other than **Text**, Ω_{Sch} is identity. Ω_{Sch} between **Uniform** and **text** datatypes is defined as follows: Given a n -tuple relation $R_{Mural} (= \{r_1, r_2, \dots, r_n\})$ in **Mural** specification and that r_i is of **Uniform** datatype, R_{Mural} can be mapped onto an equivalent relation R_{Rel} in standard relational algebra, where R_{Rel} is, $((R_{Mural} - r_i) \cup (\prec(r_i)))$ (that is, $\{r_1, r_2, \dots, r_{i-1}, r_{i_{Text}}, r_{i_{LangID}}, \dots, r_n\}$). The resulting R'_{Rel} is a relation composed of $(n + 1)$ -tuple of standard relational datatypes. Similarly, a given a n -tuple relation S_{Rel} composed of standard relational datatypes, may be converted into $(n - 1)$ -tuple relation S_{Mural} in **Mural** algebra, by composing **Uniform** datatype from two *appropriate* standard relational

datatypes (say, s_i of type `Text` and s_j of type `ID`) as, $((S_{Rel} - s_i - s_j) \cup (\succ(s_i, s_j)))$. In the absence of the language identifier, a new identifier attribute may be created (with a value of `Null`), yielding an n -tuple in `Mural` algebra. Thus, a relation in normal relational algebra may be transformed with no loss of information into a relation in `Mural`, and *vice-versa*. •

Theorem 5.2 [Relational Completeness Theorem]: There is a mapping scheme Ω that maps a relational algebra database D to a `Mural` database $\Omega(D)$, such that, for every query Q on D , there is a corresponding expression \hat{Q} such that $\hat{Q}(\Omega(D)) = \Omega(Q(D))$.

Proof: For proving relational completeness, it is only necessary to show the existence of mappings for all possible queries from *standard* database to the *transformed* database, and *not vice-versa*. Mapping lemma defines and ensures that a mapping exists between a `Mural` schema and a schema in standard algebra.

For normal datatype attributes and normal relational algebra operators, the Ω is identity, trivially. The new operators $\Xi, \Psi, \Phi_{\mathcal{H}}$ can be applied only on `Uniform` datatype; hence, there is no need for defining Ω for these operators. We only need to show that a Ω for normal `Text` manipulating operators applied on D has an equivalence in $\Omega(D)$. Suppose Q is an expression (in conjunctive normal form) in standard relational algebra ($= q_1 \wedge q_2 \wedge \dots \wedge q_n$). Each q_i is a disjunction of the form $q_{i1} \vee q_{i2} \vee \dots \vee q_{id_i}$, where q_{ij} is a predicate of the form $(a R b)$, where R is one of the standard operators on standard relational attributes, a and b . As discussed in Section 5.2.1, such operations, depending on whether the text part or the ID part was used in Q , may be mapped to an expression \hat{Q} as, $((\Pi_{a_{uni_{Text}}} (\prec(a_{uni}))) R (\Pi_{b_{uni_{Text}}} (\prec(b_{uni}))))$ or $((\Pi_{a_{uni_{ID}}} (\prec(a_{uni}))) R (\Pi_{b_{uni_{ID}}} (\prec(b_{uni}))))$, where a_{uni} is $\succ(a_{uni_{Text}}, a_{uni_{ID}})$ and b_{uni} is $\succ(b_{uni_{Text}}, b_{uni_{ID}})$.

Thus, the `Mural` algebra is *relationally complete*. •

A critically important outcome of the above result is that the existing systems, which are relationally complete, may be extended relatively easily to handle multilingual data. Only new multilingual datatype and operator functionality needs to be added. In Chapter 6, our methodology to extend relational database management systems and the standard database query language `SQL` to encompass multilingual operators, is presented.

5.5 Mural Query Optimization Strategies

In this section, some optimization opportunities afforded by the Mural algebraic definitions of multilingual operators, are presented.

5.5.1 Cost-based Optimization Strategies

For **Cost-based optimization** strategies, the following specific inputs are needed:

- **Composition rules between operators**
- **Operator cost models**
- **Estimated output size of operators**

The **composition rules** of the operators were discussed in Section 5.3. The **cost models** and the **estimated output sizes** operators, are discussed in this section.

Operator Cost Analysis

Two variations of each of the operators are analysed in this section: Specifically, they are, **scan** type, which is of the form $\langle Attr \rangle \text{ Oper } \langle Const \rangle$, and **join** type, which is of the form $\langle Attr \rangle \text{ Oper } \langle Attr \rangle$. For a **scan** version of the operator, the RHS operand is a constant and hence only a scan of the LHS table is costed. For a **join** version of the operator, a combination of the basic *nested-loops* and *hash* type join procedure is costed. Specifically, a nested-loops type join procedure is done, but with the LHS and RHS values partitioned, so that the operators are invoked only on unique values pairs of the operands⁴. For Φ operator, the phoneme strings are assumed to be materialized and the indexes are assumed to be created on the materialized phoneme strings. All *edit-distance* computations are assumed to be implemented using *diagonal transition* [96] algorithm for its better complexity.

The notation to be used for defining the cost models are given in Table 5.2 and the costs of operations – both the disk I/O operations and the algorithmic complexity in

⁴It should be noted here that the standard versions of *sort-merge* or *hash* join types cannot be used, since the operators need to be invoked for *every* pair of unique LHS and RHS operand values.

Symbol	Represents
LHS (L) and RHS (R) Operands	
R_L, R_R	No. of Records in L, R
U_L, U_R	No. of Unique Values of L, R
l_L, l_R	Avg. length of Records in L, R
P_L, P_R	No. of Pages in L, R
E_L, E_R	No. of Pages for Exact Index in L, R
A_L, A_R	No. of Pages for Approximate Index in L, R
I_L, I_R	No. of keys per Index page
k	Ψ Error Tolerance (as a fraction in $(0, 1]$)
σ	Ψ Size of the Alphabet ($= \Sigma $)
$R_{\mathcal{H}}$	No. of Records storing \mathcal{H} ($= \mathcal{H} $)
$P_{\mathcal{H}}$	No. of Pages storing \mathcal{H}
$E_{\mathcal{H}}$	No. of Pages storing Index of \mathcal{H}
f, h	Average <i>fan-out</i> and <i>height</i> of \mathcal{H}

Table 5.2: Symbols used in Mural Operator Cost Models

big-O notation – are given in Table 5.3.

Estimation of Output Size of Operators

In this subsection, heuristics to estimate the output size of the multilingual matching operators, based on the the input sizes and other meta-information stored in the database, are presented.

Estimation of Size of Ξ Output: The output size of the Ξ operator is similar to that of a normal equality operator for Text datatype. Accurate estimations may be obtained by maintaining histograms; An end-biased histogram [62] is employed for estimation, as it is shown to be practical and near-optimal, for database estimation.

Estimation of Size of Ψ Output: Estimation of matching in metric domain is a known open problem. In our work, we employed a simple estimation technique based on the practical and near-optimal end-biased histograms [62], as follows: The 10 most-frequent values of the phonemic string attribute are stored, along with their frequencies, explicitly in the histogram associated with that attribute. The selectivities based on the approximate matching (with the user specified threshold for the specific query) from

Operator	Remarks	Algorithm Complexity	Disk I/O
scan-type Operations			
Ξ	No Index	$R_L l_L$	P_L
Ξ	Index	$\log E_L I_L l_L$	$\log E_L$
Ψ	No Index	$R_L l_L k / \sqrt{\sigma}$	P_L
Ψ	Approximate Index	$R_L l_L k^2 / \sqrt{\sigma}$	A_L
$\Phi_{\mathcal{H}}$	No Index	$R_L + R_{\mathcal{H}}(h+1)$	$P_{\mathcal{H}}(h+1)$
$\Phi_{\mathcal{H}}$	Index on \mathcal{H}	$R_L + \log E_{\mathcal{H}}(h+1)$	$\log E_{\mathcal{H}}(h+1)$
join-type Operations			
Ξ	No Index	$R_L R_R l_L$	$3(P_L + P_R)$
Ξ	Index	$U_L \log E_R I_R l_R$	$E_L + E_R$
Ψ	No Index	$R_L R_R l_L k / \sqrt{\sigma}$	$3(P_L + P_R)$
Ψ	Approximate Index	$R_L R_R l_L k^2 / \sqrt{\sigma}$	$A_L + A_R$
$\Phi_{\mathcal{H}}$	No Index	$R_L + R_R + R_R R_{\mathcal{H}}(h+1)$	$3(P_L + P_R) + P_{\mathcal{H}}$
$\Phi_{\mathcal{H}}$	Index on \mathcal{H}	$R_L + R_R + R_R \log E_{\mathcal{H}}(h+1)$	$3(P_L + P_R) + E_{\mathcal{H}}$

Table 5.3: Mural Operator Cost Models

among these most frequent values are used as an approximation for the selectivity of the entire query. For a **scan** query, the query string is compared (using the approximate matching algorithm with the user specified threshold value) with the most frequent values of the attribute, and for a **join** query, the most frequent values of each of the LHS and the RHS operands are compared with each other; the resulting selectivity is used as the selectivity of the Ψ operator.

Estimation of Size of $\Phi_{\mathcal{H}}$ Output: The output size of $\Phi_{\mathcal{H}}$ operator is estimated using the notation in Table 5.2, as follows: Given the average height of \mathcal{H} is h , the selectivity of **scan** predicate is given by $(h+1)/|\mathcal{H}|$, and the selectivity of **join** predicate is given by $R_L(h+1)/|\mathcal{H}|$. In the case where closures are pre-computed and stored, the estimation accuracy may be improved further by using the exact values as, $|\mathcal{T}_{\mathcal{H}}(v)|/|\mathcal{H}|$ and $R_L|\mathcal{T}_{\mathcal{H}}(v)|/|\mathcal{H}|$, respectively, where $|\mathcal{T}_{\mathcal{H}}(v)|$ is the size of the closure of v in \mathcal{H} .

5.5.2 Rule-based Optimization Strategies

In addition to the cost-based optimization given in the previous section, several rule-based optimization heuristics may be applied to effectively reduce the cost of the query. Some of the significant heuristics are described below:

Reducing Input Size to Ψ and $\Phi_{\mathcal{H}}$: The selection and grouping operators are pushed below Ψ and $\Phi_{\mathcal{H}}$, to reduce the size of the input to these expensive operators.

Sort Input to Ψ and $\Phi_{\mathcal{H}}$: Sorting the input values, though it incurs additional cost, saves invocations of expensive Ψ and $\Phi_{\mathcal{H}}$ operators. However, explicit sorting may be avoided, by pushing Ψ and $\Phi_{\mathcal{H}}$ above those operators that produce sorted output, such as *index-scan*.

Partitioned Input Values : Partitioning of the input values could be used in restricting the invocation of Ψ and $\Phi_{\mathcal{H}}$ operators to only unique values. Hence all structures that sort or partition data, such as the B+tree indexes, must be used.

Order Ψ and $\Phi_{\mathcal{H}}$, by decreasing selectivity : Highly selective operators, including those among Ψ and $\Phi_{\mathcal{H}}$, must be pushed to the bottom of the execution tree, in order to reduce the input size to the Ψ and $\Phi_{\mathcal{H}}$ operators. In case of $\Phi_{\mathcal{H}}$, the highly selective operators must be pushed beyond the *RHS* branch of $\Phi_{\mathcal{H}}$ operator; for example, Ψ with small *threshold* parameter or $\Phi_{\mathcal{H}}$ with a *RHS* value deep in the taxonomic tree, must be pushed down.

Use of Approximate Index Structures: The availability of approximate index structures may reduce the *in-memory* computations for the Ψ operator, by only invoking Ψ on a fraction of the table; however, they could very well increase the disk I/O cost, due to the random access pattern that they direct. Hence, it should be used only for small match thresholds (say, ≤ 0.2). For higher match thresholds, a full scan of the table may reduce the disk I/O cost.

5.6 Related Research

To the best of our knowledge, an algebra for adding multilingual functionality, has not been discussed earlier in database literature. While our implementation proposals in Chapters 3 and 4 were motivated by the need to add multilingual functionality on unmodified relational database systems as they exist today, in this Chapter we proposed a holistic approach to multilingual query processing, with a query algebra.

Our motivation and approach for a query algebra for multilingual query processing parallels the domain-specific algebras that are available (such as PiQA[121] in Bioinformatics and TAX[64] in XML query processing). The Mural algebra would make the query processing declarative and amenable for optimization by the relational optimizer. For approximate matching algorithm techniques and cost models for implementing the Ψ and $\Phi_{\mathcal{H}}$ operators, we leverage on the fertile research for string matching, presented in [17, 63, 96, 97, 98]. Estimation of the output size for the Ξ operator is based on the end-serial histograms[61]. However, the estimation of the output for approximate matching is a known open problem. An estimation technique presented in [16] is based on an assumption that the selectivity of a short substring approximates the selectivity of the whole string; such assumption, while applicable for text data in specific domains, is not a valid assumption in the phonetic domains. In our implementation, we resorted to a simple estimation technique paralleling the estimations in text space using end-serial histograms, but with approximate matching of the most frequent values. A good estimation technique in metric space, is still an open research problem.

Chapter 6

A Native Implementation Experience

6.1 Overview of the Chapter

Different integration strategies exist for adding new functionalities to the kernel of relational database systems. In previous chapters, UDF-based and SQL-based approaches were pursued for implementing `MLNameJoin` and `MLSemJoin`, primarily due to the fact that the implementations were on unmodified commercial database systems to demonstrate the addition of these functionalities to existing systems. In this chapter, we present our experience in *native* implementation of the functionalities on an open-source database system, and subsequently demonstrate the optimization opportunities that such an implementation affords.

6.2 Implementation Methodologies

The methodology to add a functionality to a database system may be classified, as either *Outside-the-Server* or *Native*, based on how tightly the functionality is integrated with the server.

Outside-the-Server Implementation

An *outside-the-server* approach implements a new functionality, without any source modification to the existing servers. Usually, a functionality is added using UDFs or using stored procedures in an appropriate programming environment supported by the system. While such a methodology adds a functionality relatively easily to the existing systems, it suffers from significant overheads in processing the UDF calls. Further, the queries may not be optimized by the relational query optimizer, as no costing could be done on the UDF calls. A relatively efficient variation in *outside-the-server* implementation is to execute the UDF in an unfenced mode; that is, the UDF executes in the same server address space. While the query execution is more efficient since the call overheads are largely eliminated, selection of better execution plans may still not be possible due to the lack of optimizer support. In addition, most commercial systems do not allow such unfenced execution.

Native Implementation

The *Native* implementation makes a new functionality fully integrated with the database system by making it a first class operator in the system. The functionality is at the same level as other relational operators (such as, *scan*, *sort* etc.). While the execution is efficient as it executes as a part of the server process, more importantly, the feature is part of the operator algebra, enabling the query optimizer to generate alternate equivalent plans and choose the best plan for execution. Costing of the plans require specific cost models for the operator depending on the algorithm and the input operand sizes, and an estimation model to predict the expected size of the output of this operator. Further, only at this level of integration, any specialized index structures (specific to the new operator) may be integrated with the query processing engine. As a result, this level of integration is the hardest to achieve, but once achieved, provides the best results, *performance-wise*, due to the effective and efficient execution of queries using the new operator.

6.3 A *Native* Implementation Experience

In this section, we outline implementation of a multilingual query processing architecture – MIRA [76] – that integrates the multilingual functionalities natively to the PostgreSQL open-source database system, along with the Mural algebra, as specified in Chapter 5. Performance experiments were conducted on this native implementation, and a baseline *outside-the-server* implementation, to demonstrate and quantify the performance improvement. Subsequently, the optimization opportunities that such a native implementation afforded, are highlighted.

6.3.1 System Environment

The native implementation of the functionality was done on the PostgreSQL open-source database system [103] (Version 7.3.4)¹, on RedHat Linux (Version 2.4) operating system. The implementation was installed and the performance of queries measured on a stand-alone standard Intel Pentium IV workstation (2.3 GHz) with 1 GB main Memory. The operator algorithms themselves were implemented in the C language and built into the database server. In order to quantify the performance improvement specifically due to the *native* implementation, baseline *outside-the-server* implementations of the two operators were also done on PL/SQL environment. The PL/SQL environment was chosen, even though more efficient PL/C was available in the PostgreSQL database system, primarily in order to have a meaningful comparison parity with the commercial systems, where only an interpreted PL/SQL or Java environment was available for adding UDFs.

6.3.2 Native Ψ Operator Implementation

The Ψ operator was implemented as a binary join operator, using the facility provided by the PostgreSQL system to define new operators. Since there is no facility to add a tertiary operator in PostgreSQL system, the third parameter – match threshold – was

¹We opted to implement MIRA on Version 7.3.4 of PostgreSQL database system, as the newer Version 8.0.1 was available only as a beta version at the time of our implementation.

implemented as a user settable session parameter. As a side effect, the value of this parameter may be set globally by the administrators of the system depending on the requirements of the domain. Further, default value for this parameter may be set and, optionally, such defaults may even be made unmodifiable by the users. The Ψ algorithm in Figure 5.2 is modified to take two operands from the operator specification and the threshold parameter from the session parameter setting. The algorithm was implemented in C language and built into database system. This approach allowed us to implement a few optimization measures such as pre-allocation of space on the heap for the dynamic programming algorithm for the entire query, instead of the default allocation on stack for each invocation of the matching function, thus making the execution more efficient.

An open-source text-to-phoneme engine – *Dhvani* [30], was integrated with the system, after modifying it to output the phonemic strings in the standard IPA alphabet. From an efficiency point of view, the phonemic strings corresponding to the multilingual strings were materialized and stored persistently to avoid repeated conversions (as happens during a join processing). A specialized index structure for metric spaces – *M-Tree* [20] – was added using the GiST feature available in PostgreSQL system. The materialized phoneme strings were indexed using this M-Tree index structure. The cost models and the selectivity estimations of the Ψ operator, as outlined in Table 5.3 and Section 5.5.1, were added to the optimizer code. Finally, the Ψ operator was added to the command repertoire of the PostgreSQL database system, so user SQL queries may use the Ψ operator, natively.

6.3.3 Native $\Phi_{\mathcal{H}}$ Operator Implementation

The $\Phi_{\mathcal{H}}$ operator was added to PostgreSQL system, also as a binary join operator, using the operator addition facility in the system. The multilingual semantic matching functionality, as given in Figure 5.3, was implemented in C and built into the database system. WordNet taxonomic hierarchies were stored in the database tables, but due to the high cost involved in computing closures over the database tables, the hierarchy records were read once and pinned in the main memory. We benefited by this optimization strategy

since the size of WordNet hierarchy is in the order of a few MB, and can fit easily in the main memory, though this strategy may not work for very large \mathcal{H} that may not fit in the main memory. The *in-progress* closure set is materialized as a hash-table in the main memory and used to prevent insertion of duplicate values during closure computation. Further, once the closure is computed, the same hash-table is used for checking if the LHS operand values are members of the closure (line 7 of the Algorithm in Figure 4.2).

To reduce potentially multiple closure computations on the same RHS operand value, the hash-table is persistently maintained in the main memory for possible reuse. When a closure is needed for an RHS operand value, the materialized hash table in the main-memory is first checked to see if the closure is already available for the same RHS value. Thus, a class of operators that need to process several LHS operand values for a given RHS operand value, may amortize the cost of computing and materializing the closures. An example for such effective reuse is the nested-loops join query using $\Phi_{\mathcal{H}}$ operator with RHS operand table as the outer loop; this loop may reuse the closure computed for an RHS operand value in the outer loop, for all of the LHS operand values in the inner loop. Further optimization was achieved by sorting the RHS operand values and computing the closure only for unique values. The cost model as given in Table 5.3 and the selectivity estimation as given in Section 5.5.1, were added to the optimizer code. Finally, the new $\Phi_{\mathcal{H}}$ operator was added to the command repertoire of the PostgreSQL database system, so user SQL queries may use the $\Phi_{\mathcal{H}}$ operator, natively.

6.4 Performance of *Native* Implementation

In this section, we explore the performance of the *native* implementation of the multi-lingual functionality on the PostgreSQL database system, and compare it with the corresponding baseline (i.e., *outside-the-server*) performance in the PL/SQL environment.

6.4.1 Performance of Ψ Implementation

First, the baseline performance of the PL/SQL based implementation of the Ψ operator in the PostgreSQL database system was established, by running the Ψ queries (**scan** and **join** types) on the same dataset that was used for `MLNameJoin` experiments on the commercial systems (as detailed in Section 2.2.2). In the baseline experiments, the standard B+Tree index is used, with duplicated data as given in Section 3.7.3. After implementing the Ψ operator natively as detailed in Section 6.3.2, the same performance experiments were repeated. For native implementation, an M-Tree index on the materialized phoneme strings was used.

Table 6.1 provides the baseline and native performance of the Ψ operator (with and without appropriate indexes), in the PostgreSQL database system:

	Query Type	Scan-type (<i>Sec.</i>)	Join-type (<i>Sec.</i>)
Baseline Implementation	<i>No Index</i>	3618	453
	<i>With B+Tree Index</i>	498	169
Native Implementation	<i>No Index</i>	5.20	1.97
	<i>With M-Tree Index</i>	4.24	1.92

Table 6.1: **Performance of Ψ Operator**

As can be seen, the baseline performance (first two lines in Table 6.1) is approximately similar to that of the commercial systems. The main impediment to the performance is the expensive UDF invocations. The native performance of the operator (last two lines in Table 6.1) is about two orders of magnitude better, even over the indexed baseline performance.

Surprisingly, the metric *M*-index, was not found to be very effective in improving the performance. The scan operator was faster by about 20%, but the join operator was hardly made any more efficient by the index. Such ineffective performance is the result of the metric index's low *Search Efficiency* (as presented in Section 3.5.3); metric index structures with poor search efficiency pose a major handicap for database processing,

and hence present a viable area for further research.

6.4.2 Performance of $\Phi_{\mathcal{H}}$ Implementation

Similarly, the basic $\Phi_{\mathcal{H}}$ operator was implemented as a UDF in the PL/SQL environment, and a baseline performance is established, by running queries as specified in Section 4.5.3. After a native implementation of the $\Phi_{\mathcal{H}}$ operator, the same queries were run to establish any improvements in performance. In both the cases the queries were repeated with a B+Tree index on the parent attribute of the taxonomy table, to quantify the effect of the index.

Figure 6.1 presents the performance of the baseline and the native implementation of $\Phi_{\mathcal{H}}$ operator, on a suite of queries computing closure cardinalities of various sizes in the WordNet taxonomic hierarchy. Note that the graph is shown in *log-log* scale.

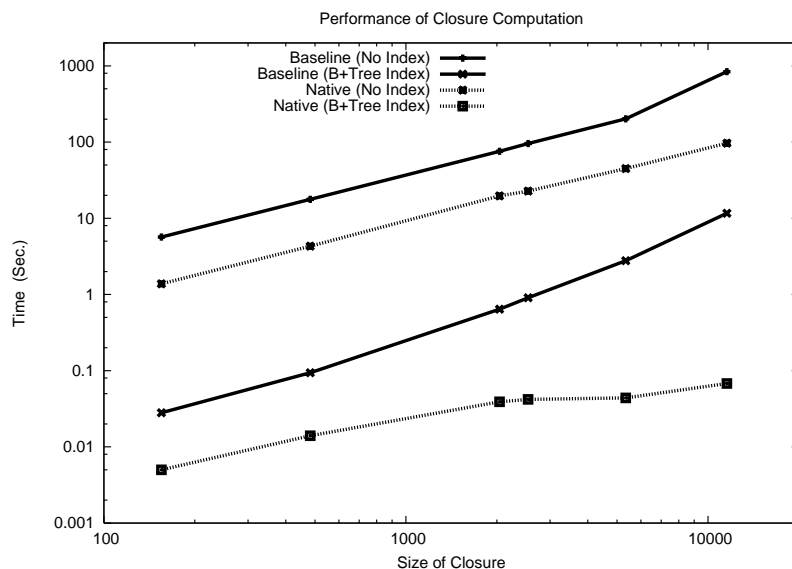


Figure 6.1: Postgres Closure Performance

It is observed that the baseline *no-index* and *index* performance are similar to that of System B (in Chapter 4), which was chosen for subsequent optimizations. Further, in this baseline implementation, even the indexed performance is of the order of a few seconds. Compared with the baseline performance of the $\Phi_{\mathcal{H}}$ operator, the performance of the native implementation (without an index), was found to be about an order of

magnitude better. With index, the performance was more than two orders of magnitude better, confirming the expected efficiency of the native implementation. In addition, such performance with a few tens of milliseconds for a closure size of around 2,000, is sufficient for practical deployments.

6.5 Optimizer Prediction Performance

In order to ascertain the the accuracy of the optimizer in predicting the query costs, using the cost models and selectivity estimations that were presented in Sections 5.5.1 and 5.5.1, we used the following methodology: A set of 16 tables, with varying data characteristics (such as, tuple count, attribute size, number of database blocks, etc.) were created. A set of scan and join queries using our multilingual operators were run on several combinations of the tables created as above. For each query, the optimizer predicted cost and the actual runtime (in milliseconds) of the query were recorded, to ascertain the correlation between the two measured metrics. Figure 6.2 plots the predicted optimizer costs and the actual runtimes of the queries, which indicates a fairly linear correlation between the metrics.

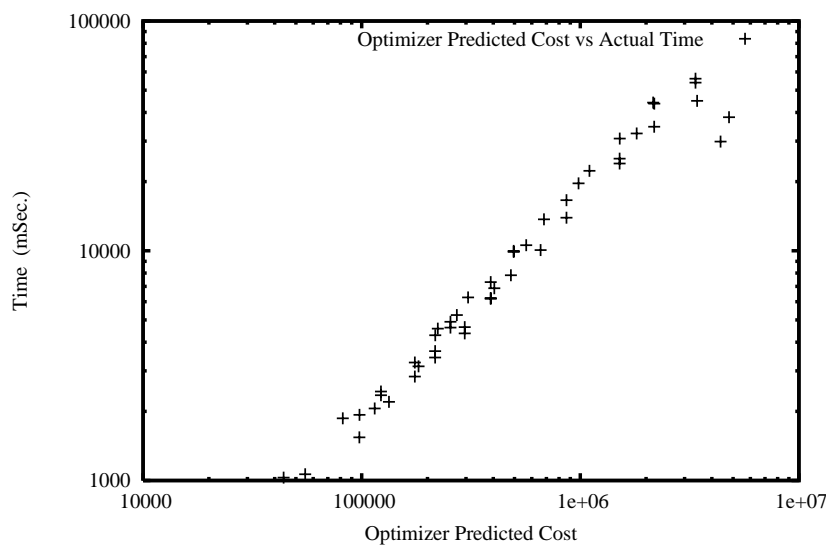


Figure 6.2: Optimizer Prediction Performance

The computed correlation coefficient² on the plot is about 0.92, indicating a well correlated data set, implying reasonably accurate cost models and selectivity estimations. Though there are some errors in computing large queries, these errors were found to be in the same range as those for the queries with standard relational operators.

6.5.1 A Motivating Optimization Example

Next, we highlight the power of the Mural algebra and optimization strategies to distinguish between efficient and inefficient execution plans, with the following example:

Example 6.1: Assume a relational schema that has Author (A) table with AuthorID and AName, Publisher (P) table with PublisherID and PName, and Book (B) table with BookID and foreign keys to Author and Publisher. Now consider the query – *Find the books whose author’s name sounds like that of a publisher’s name (match threshold of 3)*. For this query, both of the following expressions (also shown pictorially in Figure 6.3) capture the query semantics:

Plan 1: $\Pi_{B.BookID}$

$$(\sigma_{(Threshold \leq 3)}(\Psi_{A.AName, P.PName}(\sigma_{(Threshold \leq 3)}(\Psi_{A.AName, P.PName}(P, (A \bowtie_{BookID} B))))))$$

Plan 2: $\Pi_{B.BookID}(B \bowtie_{BookID}$

$$(\sigma_{(Threshold \leq 3)}(\Psi_{A.AName, P.PName}(P, A))))$$

The tables Author, Book and Publisher were created along the lines of our examples in the previous sections, with 100,000, 1 Million and 1,000 tuples, respectively. To compare different plans, we forced the optimizer to evaluate and run different execution plans for the same query on the same tables, by enabling or disabling different optimizer

²The correlation coefficient (r) quantifies the quality of the least squares fitting to a given pair of metrics. The correlation coefficient between two metrics x and y is computed as, $\sqrt{S_{xy}^2 / S_{xx}S_{yy}}$, where $S_{xy} = \Sigma xy - n\bar{x}\bar{y}$, $S_{xx} = \Sigma x^2 - n\bar{x}^2$ and $S_{yy} = \Sigma y^2 - n\bar{y}^2$. A correlation coefficient value of 1 indicates perfect correlation and a value of 0 indicates absence of any correlation between the metrics.

options. The optimizer predicted cost and the runtime of the execution of the query (with different execution plans) were recorded.

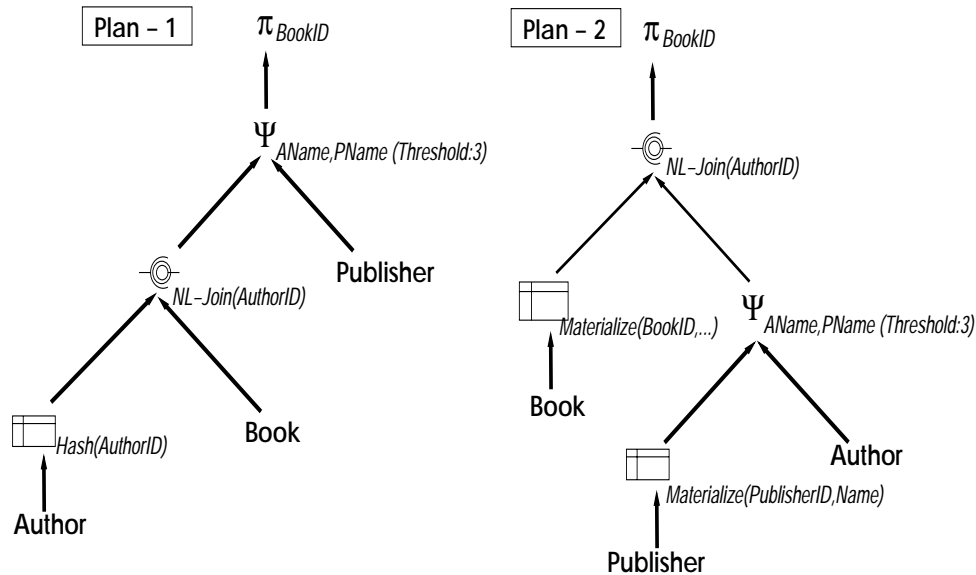


Figure 6.3: Alternate Query Plans for Example 6.1

In the above example, the optimizer predicted cost and the runtime for Plan(1) were 2,439,370 and 82 seconds, respectively. The corresponding figures for Plan(2) were 7,513,852 and 2338 seconds, respectively. Clearly, Plan(1) is superior (in terms of runtime, an observation after execution) and was chosen (due to its lower a priori predicted cost by the optimizer) for execution by the optimizer.

This example demonstrates the effectiveness of the query optimizer in selecting effective execution plans, as the runtime for Plan(2) is nearly 30 times that of Plan(1). Further, the fact that different query execution plans could be chosen, by changing the profile of the data (such as, size of attributes, number of records, amount of duplication in the table, etc.) in the underlying table, confirms the ability of the optimizer to cost and choose alternate plans in our *native* implementation of the multilingual functionalities in the PostgreSQL database system. \diamond

6.6 A Prototype Demonstration

A prototype implementation of the multilingual query processing architecture presented in this thesis – MIRA – was demonstrated in the ACM SIGMOD International Conference on Management of Data, in Paris, France, in 2004 [75].

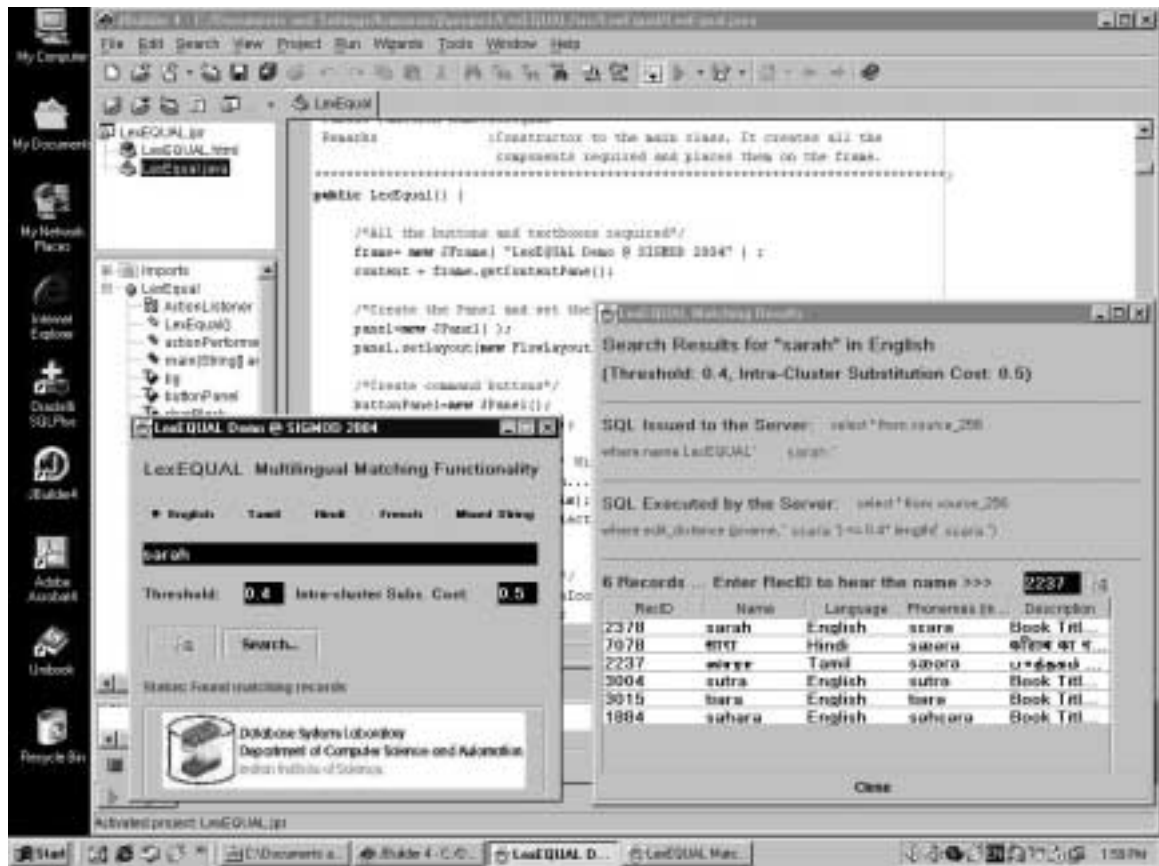


Figure 6.4: A Prototype Implementation

The prototype was implemented following the *outside-the-server* methodology, to demonstrate the multilingual querying capabilities outlined in this thesis. The functionalities were implemented on Oracle 9i[100] Database Server, using PL/SQL environment. The user-interface was developed in Java [119] environment to facilitate multilingual input and output, on Microsoft Windows 2000 Professional [86] platform. The FreeTTS [43] English text-to-speech engine was integrated with the user-interface to vocalize specific input and output strings.

A screen shot from the system is shown in Figure 6.4, that demonstrates the multilingual names retrieval capabilities of the Ψ operator, for an English input name “Sarah” (*Match Threshold* 0.4 and *Intracluster Substitution Cost* 0.5), with the answers retrieved in English, Tamil and Hindi.

6.7 Conclusions on *Native Implementation*

Our earlier implementations presented in Chapters 3 and 4 were meant to demonstrate that the functionality proposed in this thesis may be implemented using existing features of relational database systems. While such approaches have performance overheads, they provide a viable solution to implement the functionality on commercial systems *as they exist today*. In this chapter, we presented a native implementation of the multilingual functionality as first class operators, on PostgreSQL open-source relational database system. This implementation demonstrated nearly two orders of magnitude better performance, over a baseline implementation that paralleled the approach taken in the earlier chapters. Such improvement in performance suggests that it is worthwhile moving the functionality natively in the commercial systems too, by the respective vendors.

The real significance of the native implementation is its ability to leverage the relational query optimizer to generate alternate query execution plans and choose the most effective plan for a given query. The ability of the optimizer to choose an effective plan, based on the Mural operator algebra, cost-models and operator selectivities that are integrated with the PostgreSQL database system, was highlighted in this native implementation.

Chapter 7

Conclusions and Future Research Avenues

7.1 Conclusions

In this thesis, we motivated the need for seamless multilingual processing of text data in relational database systems – the backbone for most global *e-Commerce* and *e-Governance* portals that need to manage text data in multiple languages, simultaneously. A survey of popular relational systems indicates that the multilingual support of the systems is limited and that the query performance is highly *inequitable* between languages based on the Latin script and those using alternative scripts. Further, *crosslingual* queries that combine information across database columns in different languages are not supported.

First, we calibrated the performance of a suite of relational database systems in handling multilingual data, using a multilingual environment based on popular TPC benchmarks. The results indicated a significant performance degradation while handling multilingual data. While the differential performance was huge when disk traffic was a factor, it was substantial even when only *in-memory* processing was considered. To alleviate these problems, we proposed a split representation format, *Cuniform*, which reduced the space needed for storing multilingual data. The experimental results with *Cuniform*

showed that it largely eliminated the differential performance for most languages, except those with unusually large repertoires.

Next, we proposed extending the standard database lexicographic matching semantics for processing of multilingual text data, with motivating examples. Two specific multilingual join operators were proposed, based on commonly available linguistic resources. We proposed the `MLNameJoin` operator for matching multilingual names that may be in different languages and scripts, adding a join functionality that was not possible in lexicographic space. We proposed implementation of `MLNameJoin` operator by transforming the matches in the multilingual *text space* into matches in an equivalent *phoneme space*, after converting the multilingual strings to their equivalent phoneme strings in a canonical format. Approximate matching techniques were employed for matching these phoneme strings, due to the inherently fuzzy nature of the phoneme space. We demonstrated that we could simultaneously achieve good *recall* and *precision* by appropriate settings of the tunable match parameters. Further, while the basic performance of `MLNameJoin` on a commercial database system using user-defined functions was very inefficient, we showed that the performance could be improved substantially, to a level acceptable for practical implementations, by using *q-gram* or *phonetic* indexing techniques.

We also proposed the `MLSemJoin` operator to match multilingual categorical attributes across languages, as well as to match categories to their subclasses in a taxonomical hierarchy. To implement the `MLSemJoin` operator, we leveraged the WordNet linguistic resources that define rich semantic relationships between the semantic primitives of a language. A performance evaluation of `MLSemJoin`, implemented using standard SQL on a commercial database system, indicated unacceptably slow response times, due to the computation of transitive closure, a necessary feature for implementing the `MLSemJoin` operator. However, by tuning the schema and index choices to match typical features of linguistic taxonomies, we demonstrated that the performance can be improved to a level commensurate with on-line user interaction.

For a full integration of multilingual functionality with the database engine, we specified a query algebra, *Mural*, with a new multilingual storage datatype and the above join operators. The operators were implemented as first-class features in the PostgreSQL open-source database system, along with all components that were required to leverage the relational query optimizer, specifically, the operator cost models and their selectivities. The experiments demonstrated that this native implementation of the multilingual operators improved the performance significantly over the outside-the-server implementation. Further, the power of the algebra was demonstrated through selection of better execution plans for queries using the multilingual operators.

In summary, this thesis presented a multilingual query processing architecture with a comprehensive set of functionalities, algorithms, implementation and optimization techniques, all geared towards achieving the goal of developing *natural-language-neutral* database engines.

7.1.1 Practical Solutions from the Thesis

This thesis proposes novel multilingual query processing semantics, along with the implementation and optimization strategies to realize them in current database management systems. While some of the advanced features require a native implementation, most of the query processing semantics may be realized on off-the-shelf commercial and open-source database management systems, and used for multilingual query processing, even today. In this section, we outline those functionalities that may be realized and their implications on query processing functionality and efficiency.

MLNameJoin Operator This operator may be added as an *outside-the-server* SQL callable function in all existing database management systems. Though the functionality may thus be added easily, it may suffer from large function call overheads. The query runtime may be improved by using the standard B+ index on the *Q-Grams* of the materialized phoneme strings corresponding to the multilingual data. In addition, in open-source database systems, the overheads may be reduced significantly

by making the function execute within the server space. However, a native implementation is required to leverage the relational query optimizer, in order to ensure effective and efficient execution of complex queries that employ the `MLNameJoin` operator.

MLSemJoin Operator This operator may be implemented by storing the taxonomical hierarchy in a table and computing the transitive closure of its nodes in a subquery. As detailed in Chapter 4, the closure computation is inefficient in relational systems and results in heavy runtime overheads. However, such an approach is still practical, if the taxonomic hierarchy is small; in such cases, the closures of all elements may be pre-computed and stored explicitly to improve the runtime efficiency. For large taxonomic hierarchies, re-organization of storage is necessary to make the query performance acceptable for practical use. In all these cases, the `MLNameJoin` operator needs to be implemented as a callable function, and may therefore not leverage the optimizer for effective execution. A native implementation is necessary for leveraging the query optimizer.

Cuniform Datatype The `Cuniform` datatype may be implemented on existing database management systems as an object that has a pair of attributes as its private members. Query processing based on these two attributes will result in a performance profile similar to that presented in Chapter 2. However, since the input and output of the system require strings to be in the `Unicode` format, and the internal processing will be in the `Cuniform` format, efficient transcoding between the two formats must be added to the input/output layer of the query processing system.

MURAL Algebra and Optimizer Support Of the strategies proposed in this thesis, adding the `Mural` query algebra is the critical element in making the database system natively multilingual. While adding the functionalities is possible and may even be made efficient, there can be no assurance that the runtime is the best possible for a given query and for a given state of the database. Addition of the `Mural` algebra with its necessary components requires substantial modifications to the

optimizer code, and hence can be done only by the respective commercial database system vendors. While we have demonstrated the power of the Mural algebra in the open-source PostgreSQL database system, we hope that other database system vendors may add similar native multilingual support to their systems, in the future.

7.2 Future Research Avenues

The following are some of the future research areas, extending the ideas presented in this thesis, which could be pursued in the future.

Approximate Indexes for Efficient Searches

Though several approximate index structures offer search capability on a data set, all of them suffer from low search efficiency. The search efficiency is an indicator of the effectiveness of the index in narrowing down the search, as explained in Section 3.5.3. For example, a query with a user match threshold of 0.5 returned about 75% of the strings in the database as candidate matches, while less than 1% of the database are real matches. Hence, better techniques to improve the effectiveness of an index in narrowing the search must be found. Better partitioning and clustering techniques in the Metric space may be explored, for building approximate indexes with better search efficiencies.

Automatic Tuning of Phonetic Match Quality

In `MLNameJoin` operator, the parameter settings for the best match quality clearly depend on the application domain. From the research point of view, the match quality is strongly influenced by the phoneme set of the languages being considered and the contents of the database. The determination of optimal match parameters as a function of the above two may be automated based on user-supplied, pre-tagged training dataset. Further, domain-specific datasets may be used for greatly improving the accuracy, and thereby the usability, of the matching operator.

Domain-Specific Ontologies

While our proposed MLSemJoin operator used WordNet taxonomic hierarchies for semantic matching, the same methodology may be applied to specific domains with well-defined ontological hierarchies. The domain-specific ontologies may be expected to be faster and more precise, due to their compact nature and due to their high resolution power. Experiments with smaller and more precise domain specific ontologies and participation on evaluation of result quality by the domain experts, may underscore the utility of the MLSemJoin operator.

Multilingual Glyphic Matching

With the advent of Internet naming convention that makes use of Unicode characters (rather than the traditional ASCII characters), it is now possible that two URL strings may *look* the same, though *spelt* with characters from different character sets. For example, an exact look-alike URL string of the popular search engine, <http://www.google.com> may be constructed with a combination of characters from English and other languages (by replacing the LATIN CHARACTER SMALL O in *google* with MALAYALAM LETTER TTHA or TIBETAN DIGIT ZERO), as <http://www.goo gle.com>, misleading users who are more prone to click a hyperlink, than to type out the URL. The need for finding such look-alike and sound-alike character strings [78] (though in the monolingual domains) is important in the pharmaceutical industry, for detecting trademark violations and for preventing potentially dangerous medical situations.

Multilingual Performance Suites

The multilingual performance tests presented in this thesis were based on the standard TPC benchmark suites that are used for calibrating the performance of database systems for OLAP applications. A *real-life* multilingual application, with a well defined data set, query set and hand-tagged or hand-verified answer set, may provide a more intuitive framework for comparing the quality and performance of multilingual systems. It would be useful for the research community to identify such a comprehensive application and design a comprehensive and scalable performance suites, for comparing different database management systems, for multilingual deployments.

Appendix A

Character Encoding Standards

In this appendix, we review basic concepts and standards for representing and encoding multilingual data.

A *Character* is the smallest component of a written language that has a semantic value. The set of all the characters in a language is called a *Repertoire*. A *Character Encoding* assigns a unique value to each of the characters in a repertoire. There are several well-known encodings, such as ISO:8859 [57] (based on ASCII), UCS-2 [58] and Unicode [125], that form the basis for storage and interchange of text data among information processing systems. Regional encodings, such as ISCII [12] for Indic languages, also exist, catering to specific regional requirements. While ISO:8859 based character sets are the most widely used currently, Unicode is becoming a de-facto standard for global interchange of information.

A.1 Unicode

Unicode [125] is a uniform 2-byte encoding standard that allows storage of characters from any known alphabet or ideographic system irrespective of platform or programming environments. Unicode is closely aligned to the ISO:10646 [58] standard, called *Universal Character Set* (UCS). The Unicode codes are arranged in *Character Blocks*, which encode contiguously the characters of a given *Script*, typically but not always, characters in a

single repertoire. Unicode has specified 3 different byte encodings, called Unicode Transfer Format (UTF), specifically as UTF-8, UTF-16 and UTF-32, to store the same character codes in a byte, word or double-word formats. While UTF-16 specifies the basic 2-byte representation similar to UCS-2, UTF-8 provides a variable length encoding that preserves the encoding of the ISO:8859 based character sets (1-byte per character), while using 2, 3 or 4 bytes for other character sets. Such preference for ISO:8859 is primarily due to the existence of large legacy data. Each of these encodings are equivalent and can be transformed into the others by simple, fast bit-wise operations. A vendor is free to choose from any of the above three encodings to be fully compliant with Unicode [124].

Language	Encoding	String	Representation (Hexadecimal)
English	ASCII	Narayan	E4.16.27.16.37.16.E6
English	Unicode (UTF-16)	Narayan	00.E4.08.16.0D.27.00.16.00.97.80.16.00.E6
English	Unicode (UTF-8)	Narayan	E4.16.27.16.37.16.E6
Tamil	ISCII	நாராயணர்	AB.0E.80.0E.AF.A9.CD
Tamil	Unicode (UTF-16)	நாராயணர்	0B.AB.0B.0E.0B.80.0B.0E.0B.AF.0B.A9.0B.CD
Tamil	Unicode (UTF-8)	நாராயணர்	E0.AE.AB.E0.AE.8E.E0.AE.80.E0.AE.8E.E0.AE.AF.E0.AE.A3.E0.AF.8D
Kanji	Unicode (UTF-16)	寺井正博	58.FA.4E.95.E8.E3.53.5A
Kanji	Unicode (UTF-8)	寺井正博	E5.BA.AF.BA.E4.E6.95.A3.A5.8D.E5.9A

Figure A.1: Sample Encoding in Various Formats

Figure A.1 shows some sample strings in three different scripts (Latin script for English, Indic script for Tamil, and CJK – standing for *Chinese, Japanese and Korean* – script for Kanji). Each string is shown in UTF-16 and UTF-8 encodings. The English string that needs 1 byte/character in ASCII needs double the space in UTF-16 but preserves the ASCII encoding in UTF-8. Indic and CJK scripts are coded in 2 bytes in the UTF-16 encoding, but need 3 bytes in the UTF-8 encoding. Specifically, the storage for Indic characters doubles in UTF-16 and triples in UTF-8, from their proprietary ISCII encoding, in which each Indic character is encoded in one byte. It should be noted here that due to the large repertoire size of CJK languages, *any* proprietary encoding of these languages would need a minimum of 2 bytes per character, equal to the storage needed under Unicode.

Appendix B

Phonology and Phonemes Encoding Standards

In this appendix, we review basics on phonology and phonemes for representing the vocalization of text, that are needed to implement our multilingual names matching methodology.

B.1 Phonology and Phonemes

Phonology is the study of sound structure related to speech, conforming to the grammar of a language. Each human language usually has between *20* to *40* abstract linguistic units, called *Phonemes*, that provide an alphabet of sounds to describe the articulation of the words in that language. A *Phone* is the physical sound produced conforming to a phoneme. Since the phone is produced by the vocal tracks of individuals, there are infinite variations of phones (called *Allophones*) that are possible based on speaker's individual, cultural and environmental factors. However, they are identified with a specific phoneme using common aural signatures. Fundamentally, phonemes are to speech, what characters are to written text.

Phonemes are grouped together in *syllables*, which are in turn grouped together in *words* of a language. Not all possible orders of phonemes are allowed, much as not

all possible sequences of characters are allowed in the written text. However, there is no simple, one-to-one mapping between characters of a language to phonemes, as the vocalization of the characters depend on context of the character within the word, or even words around it. Such rules of the mapping of a group of characters to a group of phonemes are extensively researched in Linguistics and Speech Processing communities. While such transformation rules are outside the scope of this thesis, they are coded in standard implementations of *Text-to-Speech/Text-to-Phoneme* (TTS/TTP) systems of a language.

B.2 International Phonetic Alphabet

International Phonetic Association (IPA) [60] is one of the popular standards for describing phonemes of any given language¹. The phonetic alphabet of IPA is capable of representing the full range of vocalizations primitives, irrespective of languages. Popular linguistic resources, such as *Oxford English Dictionary* [101], define and publish the phonemic equivalent of nouns in IPA alphabets and standard TTP systems can generate a phonetically equivalent IPA string for a given character string of that language. Some TTP engines generate phonemes in one of the other phoneme standards and not in IPA; in all cases, such phoneme strings may be mapped IPA strings, using linguistic rules.

The IPA alphabets are described by a combination of the basic *Latin* character set and characters from a specific IPA block in the Unicode[125] encoding scheme. Hence, phonetic representation of character strings in *any* language may be stored and manipulated as Unicode strings in IPA character set. Further, since Unicode is supported in all database systems as the default encoding scheme for multilingual text, it provides a transparent mechanism for storing phoneme strings as well. That is, IPA strings may be stored and manipulated as NChar data, in the standard database systems.

¹There are other standards available for coding the phonemes but they are usually restricted in their coverage of phonemes, since they are designed either for a specific language or a group of related languages. Examples for such standards are, *Arpabet* [67] that is designed specifically for American English and *ITrans* [122] designed for a subset of Indic languages.

Bibliography

- [1] aAQUA Project. <http://aaqua.persistent.co.in/mvnforum/mvnforum/index>
- [2] The Aberdeen Group Ltd. <http://www.aberdeen.com>
- [3] R. Agrawal and H. V. Jagadish. Direct algorithms for computing Transitive Closure of DB Relations. *Proc. of the 13th Intl. Conf. on Very Large Data Bases (VLDB)*, 1987.
- [4] M. Andersson, Y. Dupont, S. Spaccapietra, K. Yetongnon, M. Tresch and H. Ye. *FEMUS: A Federated Multilingual Database System*. Advanced Database Systems, Springer-Verlag, 1993.
- [5] The Association for Computational Linguistics. <http://www.aclweb.org>
- [6] S. Atkin and R. Stansifer. Unicode Text Compression. *Proc. of the 22nd Intl. Unicode Conf.*, 2002.
- [7] R. Baeza-Yates and G. Navarro. Faster Approximate String Matching. *Algorithmica*, 23(2), 1999.
- [8] C. Bell. Customer Experience @ Amazon.com. *Keynote Address at the SIGMOD Intl. Conf. on Mgmt. of Data*, 2002.
- [9] Bhoomi: Computerized Land Records System. <http://www.revdept-01.kar.nic.in>
- [10] P. A. Boncz, A. N. Wilschut and M. L. Kersten. Flattening an Object Algebra to Provide Performance. *Proc. of the 14th IEEE Intl. Conf. on Data Engg.*, 1998.

- [11] The British National Corpus. <http://www.comp.lancs.ac.uk>
Oxford University Press, Oxford, UK, 2001.
- [12] Bureau of Indian Standards. IS 13194:1991 8-bit Coded Character Set for Information Interchange. 1991.
- [13] W. A. Burkhard and R. M. Keller. Some Approaches to Best-match File Searching. *Comm. of the ACM*, 16(4), 1973.
- [14] M. J. Carey, D. J. DeWitt, J. F. Naughton, M. Asgarian, P. Brown, J. E. Gehrke and D. N. Shah. The BUCKY Object-Relational Benchmark. *Proc. of the ACM SIGMOD Intl. Conf. on Mgmt. of Data*, 1997.
- [15] Centre for Indian Language Technology, IIT-Bombay. <http://www.cfilt.iitb.ac.in>
- [16] S. Chaudhuri, V. Ganti and L. Gravano. Selectivity Estimation for String Predicates: Overcoming the Underestimation Problem. *Proc. of the 20th IEEE Intl. Conf. on Data Engg.*, 2004.
- [17] E. Chavez, G. Navarro, R. Baeza-Yates and J. Marroquin. Searching in Metric Space. *ACM Computing Surveys*, 33(3), 2001.
- [18] H. Chen, C. Lin and W. Lin. Building a Chinese-English WordNet for Translingual Applications. *ACM Trans. on Asian Languages Information Processing*, 1(2), 2002.
- [19] K. S. Choi and H. S. Bae. Procedures and Problems in Building a Korean-Chinese-Japanese WordNet with Shared Semantic Hierarchy. *Proc. of the 2nd Global WordNet Conf.* 2004.
- [20] P. Ciaccia, M. Patella and P. Zezula. M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces. *Proc. of the 23rd Intl. Conf. on Very Large Data Bases (VLDB)*, 1997.
- [21] E. Codd. Relational Completeness of Data Base Sublanguages. in *Database Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

- [22] W. Cohen, P. Ravikumar and S. E. Fienberg. A Comparison of String Distance Metrics for Name-Matching Tasks. *Proc. of the IJCAI-2003 Workshop on Information Integration on the Workshop (IIWeb-03)*, 2003.
- [23] W. Cohen, P. Ravikumar and S. E. Fienberg. A Comparison of String Metrics for Matching Names and Records. *Proc. of the Workshop on Data Cleaning in Conjunction with SIGKDD*, 2003.
- [24] The Computer Scope Ltd., Dublin, Ireland. <http://www.NUA.ie/Surveys>
- [25] S. Das, E. I. Chong, G. Eadon, J. Srinivasan. Supporting Ontology-based Semantic Matching in RDBMS. *Proc. of the 30th Intl. Conf. on Very Large Data Bases (VLDB)*, 2004.
- [26] M. Davis. Unicode collation algorithm. *Unicode Consortium Technical Report*, 2001.
- [27] S. Deerwester, S. T. Dumais and W. C. Ogden. Indexing by Latent Semantic Analysis. *Jour. of American Soc. of Information Sciences*, 41(6), 1990.
- [28] A. Deutsch, M. Fernandez and D. Siciu. Storing Semistructured Data with STORED. *Proc. of the ACM SIGMOD Intl. Conf. on Mgmt. of Data*, 1999.
- [29] Dewey Decimal Classification System. <http://www.oclc.org>
- [30] Dhvani - A Text-to-Speech System for Indian Languages. <http://dhvani.sourceforge.net>
- [31] The Dublin Core Metadata Initiative. <http://www.dublincore.org>
- [32] K. Erikson. Approximate Swedish Name Matching - Survey and Test of Different Algorithms. *Technical Report TRITA-NA-E9721*, Royal Institute of Technology, Stockholm, Sweden, 1997.
- [33] The EROS System. <http://merovingio.c2rmf.cnrs.fr/eros/eros.xhtml>
- [34] The Europe Portal. <http://europe.eu.int>

- [35] The EuroSeek Corporation. <http://www.euroseek.com>
- [36] The Euro-Spider. <http://www.euospider.ch>
- [37] The Euro-WordNet. <http://www.illc.uva.nl/EuroWordNet>
- [38] The Euro-WordNet – Final Results Report.
<http://www.illc.uva.nl/EWN/finalresults-ewn.html>
- [39] C. Fellbaum and G. A. Miller. WordNet: An electronic lexical database (language, speech and communication). *MIT Press*, Cambridge, Massachusetts, 1998.
- [40] P. Fenwick and S. Brierley. Compression of Unicode Files. *Proc. of the Data Compression Conf.*, 1998.
- [41] C. Fluhr, D. Schmit, F. Elkateb and K. Gurtner. Multilingual Database and Crosslingual Interrogation in a Real Internet Application. *Proc. of the AAAI Sym. on Crosslanguage Text and Speech Retrieval*, 1997.
- [42] The Foreign Word – The Language Site, <http://www.ForeignWord.com>
- [43] FreeTTS Speech Synthesis System. <http://freetts.sourceforge.net> and <http://research.sun.com>
- [44] T. N. Gadd. PHONIX: The Algorithm. *Program*, 24(4), October 1990.
- [45] The Gene Ontology. <http://www.geneontology.org>
- [46] F. Gey, A. Chen, M. Buckland and R. Larson. Translingual Vocabulary Mapping for Multilingual Information Access. *Proc. of the 25th ACM SIGIR Conf. on Research and Development in Information Retrieval*, 2002.
- [47] J. Gilarranz, J. Gonzalo and F. Verdejo. An Approach to Conceptual Text Retrieval using the Euro-WordNet Multilingual Semantic Database. *Proc. of the AAAI Conf. on Crosslanguage Text and Speech Retrieval*, 1997.
- [48] The Global Reach. <http://www.globalreach.biz>

- [49] The Global WordNet Association. <http://www.globalwordnet.org>
- [50] The Google Corporation. <http://www.google.com>
- [51] Income Tax Department, Ministry of Finance, Government of India.
<http://www.incometaxindia.gov.in>
- [52] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan and D. Srivastava. Approximate String Joins in a Database (almost) for Free. *Proc. of the 27th Intl. Conf. on Very Large Data Bases (VLDB)*, 2001.
- [53] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text Joins in an RDBMS for Web Data Integration. *Proc. of the World-Wide Web (WWW) Conf.*, 2003.
- [54] D. Gusfield. Algorithms on Strings, Trees and Sequences. *Cambridge University Press*, Cambridge, United Kingdom, 2001.
- [55] J. Han, H. Lu. Some Performance Results on Recursive Query Processing in Relational Database Systems. *Proc. of the 2nd IEEE Intl. Conf. on Data Engg.*, 1986.
- [56] IBM Corporation, Armonk, New York. <http://www.ibm.com>
- [57] International Organization for Standardization. ISO/IEC 8859 Information Processing – 8-bit Single-byte Graphic Coded Character Sets. 1999.
- [58] International Organization for Standardization. ISO/IEC 10646-1:1993, Universal Multiple-Octet Coded Character Set (UCS). 1993.
- [59] International Organization for Standardization. ISO/IEC 9075-1-5:1999, Information Technology – Database Languages – SQL (parts 1 through 5). 1999.
- [60] International Phonetic Association. Univ. of Glasgow, Glasgow, UK.
<http://www.arts.gla.ac.uk/IPA/ipa.html>
- [61] Y. Ioannidis. Universality of Serial Histograms. *Proc. of the 19th Intl. Conf. on Very Large Data Bases (VLDB)*, 1993.

- [62] Y. Ioannidis and V. Poosala. Histogram-based Solutions to Diverse Database Estimation Problems. *IEEE Data Engineering*, 18(3), 1995.
- [63] Y. Ioannidis. On the Computation of Transitive Closure of Relational Operators. *Proc. of the 12th Intl. Conf. on Very Large Data Bases (VLDB)*, 1986.
- [64] H. V. Jagadish, L. Lakshmanan, D. Srivastava and K. Thompson. TAX: A Tree Algebra for XML. *Proc. of the DBPL Conf.*, 2001.
- [65] B. D. Jayaram and P. Bhattacharyya. Report on Indo-WordNet Workshop. *Central Institute of Indian Languages*, January 1999.
- [66] L. Jin, C. Li and S. Mehrotra. Efficient Record Linkage in Large Data Sets. *Proc. of the 8th Intl. Conf. on Database Systems for Advanced Applications (DASFAA)*, 2003.
- [67] D. Jurafsky and J. Martin. Speech and Language Processing. *Pearson Education*, New Delhi, India, 2000.
- [68] S. Kagathara, M. Deodalkar and P. Bhattacharyya. A Multistage Fall-back Search Strategy for Cross Lingual Information Retrieval. *Proc. of the Sym. on Indian Morphology, Phonology and Language Engineering*, 2005.
- [69] I. Kalantari and G. McDonald. A Data Structure and Algorithm for the Nearest Point Problem. *IEEE Trans. on Software Engineering*, 9(5), 1983.
- [70] E. Keogh. Exact Indexing of Dynamic Time Warping. *Proc. of the 28th Intl. Conf. on Very Large Data Bases (VLDB)*, 2002.
- [71] R. King and A. Morfeq. Bayan: An Arabic Text Database Management System. *Proc. of the ACM SIGMOD Intl. Conf. on Mgmt. of Data*, 1990.
- [72] D. Knuth. The Art of Computer Programming, Volume 1: Fundamental Algorithms. *Addison-Wesley*, Reading, Massachusetts, 1968.

- [73] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.
- [74] G. Kondrak. A New Algorithm for the Alignment of Phonetic Sequences. *Proc. of the 1st Meeting of North American Chapter of Association of Computational Linguistics*, 2000.
- [75] A. Kumaran and J. R. Haritsa. LexEQUAL: Multilexical Matching Operator in SQL. *Proc. of the ACM SIGMOD Intl. Conf. on Mgmt. of Data*, 2004.
- [76] A. Kumaran. MIRA: Multilingual Information-processing on Relational Architecture. *Spinger's Lecture Notes in Computer Science (Volume: 3268)*, November 2004.
- [77] A. J. Lait and B. Randell. An Assessment of Name Matching Algorithms. *Technical Report, Department of Computing Sciences, University of Newcastle upon Tyne*, 1993.
- [78] B. Lambert, K. Chang and S. Lin. Descriptive analysis of the drug name lexicon. *Drug Information Journal*, 35(1), 2001.
- [79] M. Liberman and K. Church. Text Analysis and Word Pronunciation in TTS Synthesis. *Advances in Speech Processing*, 1992.
- [80] W. Lin and H. Chen. Backward Machine Transliteration by Learning Phonetic Similarity. *Proc. of the 6th Conf. on Natural Language Learning*, 2002.
- [81] C. Lu and K. Lee. A Multilingual Database Management System for Ideographic Languages. *Chinese University of Hong Kong Technical Report*, 1992.
- [82] P. Mareuil, C. Corredor-Ardoy and M. Adda-Decker. Multilingual Automatic Phoneme Clustering. *Proc. of the 14th Intl. Congress of Phonetic Sciences*, 1999.
- [83] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, San Francisco, California, 1993.

- [84] J. Melton and A. R. Simon. *SQL 1999: Understanding Relational Language Components*. Morgan Kaufmann, San Francisco, California, 2001.
- [85] E. Mena, V. Kashyap, A. Illarramendi and A. Sheth. *Domain Specific Ontologies for Semantic Information Brokering on Global Information Infrastructure*, 1998.
- [86] Microsoft Corporation, Redmond, Washington. <http://www.microsoft.com>
- [87] G. A. Miller. WordNet: A Lexical Database. *Comm. of the ACM*, 38(11), 1995.
- [88] G. A. Miller. Nouns in WordNet: A Lexical Inheritance System. *Princeton University*, Princeton, New Jersey, 1993.
- [89] The MIRA Project. <http://dsl.serc.iisc.ernet.in/projects/MIRA>
- [90] The Monet DB. <http://monetdb.cwi.nl>
- [91] A. Monge and C. Elkan. The Field-matching problem: Algorithms and Applications, *Proc. of the 2nd Intl. Conf. on Knowledge Discovery and Data Mining*, 1996.
- [92] M. Mudawwar. Multicode: A Multilingual Text Encoding. *IEEE Computer Magazine*, April 1997.
- [93] A. Mujoo, M. Kumar. Malviya, R. Moona, T. V. Prabhakar. A Search Engine for Indian Languages. *Proc. of the 1st Intl. Conf. on Electronic Commerce and Web Technologies (EC-Web 2000)*, 2000.
- [94] MySQL AB, Uppsala, Sweden. <http://www.mysql.com>
- [95] D. Narayan, D. Chakrabarty, P. Pande and P. Bhattacharyya. Experience in Building the Indo WordNet - A WordNet for Hindi. *Intl. Conf. on Global WordNet (GWC)*, 2002.
- [96] G. Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 33(1), 2001.

- [97] G. Navarro, E. Sutinen, J. Tanninen, J. Tarhio. Indexing Text with Approximate q -grams. *Proc. of the 11th Combinatorial Pattern Matching Conf.*, 2000.
- [98] G. Navarro, R. Baeza-Yates, E. Sutinen and J. Tarhio. Indexing Methods for Approximate String Matching. *IEEE Data Engg. Bulletin*, 24(4), 2001.
- [99] The OntoWeb. <http://ontoweb.aifb.uni-karlsruhe.de>
- [100] Oracle Corporation, Redwood Shores, California. <http://www.oracle.com>
- [101] The Oxford English Dictionary. *Oxford University Press*, Oxford, UK, 1999.
- [102] U. Pfeifer, T. Poersch and N. Fuhr. Searching Proper Names in Databases. *Proc. of the Conf. Hypertext-Information Retrieval-Multimedia*, 1995.
- [103] PostgreSQL Database Systems, Berkeley, California. <http://www.postgresql.com>
- [104] L. Rabiner and B. Juang. *Fundamentals of Speech Processing*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.
- [105] Agro Explorer System. <http://agro.mlasia.iitb.ac.in>
- [106] M. Surve, S. Singh, S. Kagathara, K. Venkatasivaramasastry, S. Dubey, G. Rane, J. Saraswati, S. Badodekar, A. Iyer, A. Almeida, R. Nikam, C. G. Pere, P. Bhattacharyya. AgroExplorer: A Meaning Based Multilingual Search Engine. Intl. Conf. on Digital Libraries, 2004.
- [107] R. Richardson and A. F. Smeaton. Using WordNet in a Knowledge-based Approach to Information Retrieval. *Working Paper CA-0395, Dublin City University*, 1999.
- [108] M. Scherer and M. Davis. BOCU-1: Mime Compatible Unicode Compression. *Unicode Notes #6 of Unicode Consortium*, 2002.
- [109] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price. Access Path Selection in a Relational Database Management System. *Proc. of the ACM SIGMOD Intl. Conf. on Mgmt. of Data*, 1979.

- [110] R. Schenkel, A. Theobald and G. Weikum. HOPI: An Efficient Connection Index for Complex XML Document Collection. *Proc. of 9th Intl. Conf. on Extending Database Technology*, , 2004.
- [111] P. H. Sellers. On the Theory and Computation of Evolutionary Distances. *SIAM Jour. of Applied Math.*, 26(4), 1974.
- [112] P. H. Sellers. The Theory and Computation of Evolutionary Distances: Pattern Recognition. *Jour. of Algorithms*, 1(4), 1980.
- [113] Semantic Web. <http://www.w3.org/2001/sw>
- [114] M. Sinha, M. Kumar, P. Pande, L. Kashyap and P. Bhattacharyya. Hindi Word Sense Disambiguation. *Proc. of the Intl. Sym. on Machine Translation, Natural Language Processing and Translation Support Systems*, 2004.
- [115] D. Soergel. Multilingual Thesauri in Cross-language Text and Speech Retrieval. *Proc. of the AAAI Sym. on Crosslanguage Text and Speech Retrieval*, 1997.
- [116] Special Interest Group in Information Retrieval (ACM SIGIR).
<http://www.acm.org/sigir>
- [117] S. Sriram. A Report on Transliteration and Crosslingual Search. *Birla Institute of Technology and Science*, 2004.
- [118] S. Sriram, P. P. Talukdar, S. Badaskar, K. Bali and A. G. Ramakrishnan. Phonetic Distance Based Crosslingual Search. *Proc. of the Intl. Conf. on Natural Language Processing*, 2004.
- [119] Sun Microsystems Corporation. <http://www.sun.com>
- [120] Sybase Corporation. <http://www.sybase.com>
- [121] S. Tata and J. M. Patel. PiQA: An Algebra for Querying Protein Data Sets. *Proc. of the 15th Intl. Conf. on Scientific and Statistical Database Management*, 2003.

- [122] Technology Development for Indian Languages. <http://tdil.mit.gov.in>
- [123] The Transaction Processing Council, San Francisco, California. <http://www.tpc.org>
- [124] The Unicode Consortium, Mountain View, California. <http://www.unicode.org>
- [125] The Unicode Consortium. *The Unicode Standard*. Addison-Wesley, Reading, Massachusetts, 2000.
- [126] The United Nations. <http://www.un.org>
- [127] The United Nations Educational, Scientific and Cultural Organization. <http://www.unesco.org>
- [128] The Unisyn Project. The Center for Speech Technology Research, Univ. of Edinburgh, United Kingdom. <http://www.cstr.ed.ac.uk/projects/unisyn>
- [129] The Universal Networking Language (UNL) System. *The UNDL Foundation*. <http://www.undl.org>
- [130] Vidyanidhi: The Digital Library and E-Scholarship Portal. <http://www.vidyanidhi.org.in>
- [131] P. Vossen. EuroWordNet: Final Report. *University of Amsterdam*, 1999.
- [132] The WebFountain. <http://www.almaden.ibm.com/WebFountain>
- [133] M. Wolf. Standard Compression Scheme for Unicode. *Technical Standard #6 of Unicode Consortium*, 2002.
- [134] Word Discover. <http://www.worddiscover.com>
- [135] The WordNet. <http://www.cogsci.princeton.edu/~wn>
- [136] The World Wide Web Consortium. <http://www.w3c.org>
- [137] Yahoo!Dictionary. <http://www.yahoo.com>

-
- [138] P. N. Yianilos. Datastructures and Algorithms for Nearest Neighbor Search in General Metric Spaces. *Proc. of the 4th ACM-SIAM Sym. on Discrete Algorithms*, 1993.
- [139] C. Yip. A Framework for the Support of Multilingual Computing Environment. *Tech. Rep. TR-97-02, University of Hong Kong*, 1997.
- [140] J. Zobel and P. Dart. Finding Approximate Matches in Large Lexicons. *Software – Practice and Experience*, 25(3), 1995.
- [141] J. Zobel and P. Dart. Phonetic String Matching: Lessons from Information Retrieval. *Proc. of the 19th ACM SIGIR Conf.*, 1996.