On the Costs of Multilingualism in Database Systems

A. Kumaran Jayant R. Haritsa

Department of Computer Science and Automation
Indian Institute of Science, Bangalore 560012, INDIA
{kumaran,haritsa}@csa.iisc.ernet.in

Abstract

Database engines are well-designed for storing and processing text data based on Latin scripts. But in today's global village, databases should ideally support multilingual text data equally efficiently. While current database systems do support management of multilingual data, we are not aware of any prior studies that compare and quantify their performance in this regard. In this paper, we first compare the multilingual functionality provided by a suite of popular database systems. We find that while the systems support most SQL-defined multilingual functionality, some needed features are not yet implemented. We then profile their performance in handling text data in ISO:8859, the standard database character set, and in Unicode, the multilingual character set. Our experimental results indicate significant performance degradation while handling multilingual data in these database systems. Worse, we find that the query optimizer's accuracy is different between standard and multilingual data types. As a first step towards alleviating the above problems, we propose Cuniform, a compressed format that is trivially convertible to Unicode. Our initial experimental results with Cuniform indicate that it largely eliminates the performance degradation for multilingual scripts with small repertoires. Further, the Cuniform format can elegantly support extensions to SQL for multilexical text processing.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

1 Introduction

Database engines have been designed and fine tuned for storing and processing text data in Latin-based scripts, encompassing languages such as English, French and German. But, in today's global village, the engines should ideally be equally efficient in supporting alternative scripts such as Arabic, Indic and Chinese-Japanese-Korean (CJK), as the Internet demographics are turning steadily multilingual [7]. In fact, a recent survey indicates that users are *likely to stay twice as long and four times more likely to buy if the information is presented in their native language* [1], making multilingualism a critical factor in global *e-Commerce*. Similarly, the importance of multilingual support in *e-Governance* solutions has been well documented [2].

While today's commercial database systems do support storage and manipulation of multilingual data, we are not aware of any in-depth study to compare their support, or to quantify their performance, with multilingual data as compared to the standard ISO:8859 (popularly known as *ASCII*) data. In this paper, we take a first step towards addressing this lacuna. The basic question we address here is *what does one pay, in functionality or performance, for choosing to store data in native languages of non-Latin scripts?*

With regard to functionality, we compared a suite of popular database systems with respect to storage formats and facilities for user interface, access and manipulation. We found that while these systems support most SQL-defined multilingual functionality, some needed features, such as *user-defined collations* are yet to be implemented.

With regard to performance, the database systems were subjected to a common testing framework to evaluate the performance impact on the basic database operators (e.g. *scan*, *join*, etc.) of storing multilingual data in the popular multilingual character set, Unicode as opposed to the standard ASCII. An obvious performance hit is that Unicode takes more storage space than ASCII, therefore requiring more disk accesses to retrieve the same semantic content. However, the novel insight from our experiments is that there are significant *in-memory* performance degradations as well,

leading to an overall situation wherein Unicode performance is seriously compromised with regard to *both* storage and processing. Worse, we find that the query optimizer's accuracy is different between standard and multilingual text data, and in some cases the optimizer is impervious to the differences between them. Such inequity could result in choice of sub-optimal plans for query execution.

The above results raise concerns about the suitability of current database systems for multilingual deployment. An obvious solution would be to translate all multilingual data into ASCII and to do a reverse translation when providing the results to the user. But translation, apart from being itself slow, can be a semantically lossy process [3] and would therefore not serve as a viable general remedy.

We outline here an alternative solution wherein multilingual data is stored in a new compressed internal storage format, called **Cuniform**, which is trivially convertible to Unicode format. Our implementation of Cuniform is also tested under the same performance framework and the results indicate that the database performance can be made *almost as good as that of ASCII* for scripts with repertoires less than 256, covering a majority of the Unicode-specified scripts. Further, the script of the underlying data is identified explicitly in Cuniform, which can be extended to support some of the multilingual requirements specified in SQL:1999. Additionally, such identification helps in implementing multilexical operators that can extend the current capabilities of SQL.

1.1 Organization

The remainder of this paper is organized as follows: Section 2 outlines a sample *e-Commerce* application, underscoring the multilingual support needed from database systems. Section 3 reviews the required background and related research. Section 4 provides an overview of the multilingual support specified in the SQL standards and that provided by the popular database systems. Section 5 describes a framework for measuring the multilingual performance and Section 6 presents the experimental results. Section 7 describes and evaluates our new Cuniform storage format. Finally, Section 8 summarizes the paper and outlines future research avenues.

2 Example Multilingual e-Business

In this section we highlight the need for efficient support from databases for multilingual processing with a sample global *e-Business*. Similar need from a large scale *e-Governance* application is detailed in [14].

Consider a hypothetical *Books.com* that sells books around the globe, presumably in different languages. The product catalog of such an *e-Business* may have records in different languages, as shown in Figure 1. It should be noted that such a need is different from the SQL:1999 recommended option of restricting a column to be of a specified language type. Further, a given attribute value itself may store a string that contains characters from different languages, as it is customary to represent common abbreviations such as *ISO*, in English. Overall, the requirement for multilingual storage may be at the attribute, record, table or schema level. In special cases the requirement may even be at the catalog level, if the database objects are to be identified with multilingual names.

Author_LastName	Author_FirstName	Title	Price
Descartes	René	Meditations	€ 9.00
நாராயண்	R.K.	சுவாமியும் சினேகிதர்களும்	INR 250
നാരായൺ	R.K.	മാല്ഗുഡി ഡേസ്	INR 155
د ، بهنسي	عفيف	العمارة عبر التاريخ	SAR 75
Nehru	Jawaharlal	Discovery of India	$ 9.95
寺井正博	著	秋の風 普及版	¥ 7500
नेहरु	जवहरलाल	भारत एक खोज	INR 175

Figure 1: **Catalog of Multilingual *Books.com***

We expect that in due course querying across languages would be a feature desired by Internet service providers like *Books.com*. As will be shown in subsequent sections, the performance of such multilingual applications is expected to be noticeably worse with current database systems, raising serious concerns about the quality of service rendered to global customers. Hence, it is imperative that solutions be found to improve multilingual database performance.

2.1 Multilingual Application Requirements

In a nutshell, global multilingual applications impose the following two major requirements for database systems:

Standard Multilingual Interchange: The multilingual data must be interchanged in a standard format that is recognizable by other systems. Hence it is preferable to store data in an internal format that is closer to the interchange format, in order to minimize conversion costs at query time.

Language Independent Query Processing: The database system must be equally efficient in any of the languages chosen by the user; that is, the database performance for two languages must be identical, if their repertoire sizes are similar.

We explore, in the remainder of this paper, how well current database systems handle the above requirements, and address their limitations in this regard.

3 Background

In this section, we review basic concepts in encoding multilingual data and survey prior research in multilingual database processing.

3.1 Character Set and Encoding

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* [10] (based on ASCII), *UCS-2* [9] and *Unicode* [25], that form the basis for storage and interchange of text data among computer systems. Regional encodings, such as *ISCII* [5] 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.

Language	Encoding	String	Representation (Hexadecimal)
English	ASCII	Narayan	E4.16.27.16.97.16.E6
English	Unicode (UTF-16)	Narayan	00.E4.00.16.00.27.00.16.00.97.00.16.00.E6
English	Unicode (UTF-8)	Narayan	E4.16.27.16.97.16.E6
Tamil	ISCII	நாராயண்	A8.BE.B0.BE.AF.A9.CD
Tamil	Unicode (UTF-16)	நாராயண்	0B.A8.0B.BE.0B.B0.0B.BE.0B.AF.0B.A9.0B.CD
Tamil	Unicode (UTF-8)	நாராயண்	E0.AE.A8.E0.AE.BE.E0.AE.B0.E0.AE.BE.E0.AE.AF.E0.AE.A3.E0.AF.8D
Kanji	Unicode (UTF-16)	寺井正博	5B.FA.4E.95.6B.63.53.5A
Kanji	Unicode (UTF-8)	寺井正博	E5.BA.AF.BA.E4.E6.95.A3.A5.8D.E5.9A

Figure 2: **Sample Encoding in Various Formats**

Unicode [25] 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 [9] standard, called *Universal Character Set* or *UCS-2*. The Unicode codes are arranged in *Character Blocks*, which encode contiguously the characters of a given script, typically characters in a single repertoire. Unicode has specified 3 different byte encodings (*UTF-8*, *UTF-16* and *UTF-32*) to store the same character codes in a byte, word or double-word formats. UTF-16 specifies the basic 2-byte representation for each and every character, 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. Understandably, such preference for ISO:8859 is 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 [24].

Figure 2 shows some sample strings in three different scripts (Latin script – English, Indic script – Tamil, and CJK script – 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. The bytes per character for Indic and CJK scripts are 2 in UTF-16, but 3 in UTF-8. In particular, the storage for Indic strings doubles in UTF-16 and triples in UTF-8, from their proprietary ISCII encoding. 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.

3.2 Related Research

While a rich body of literature on multilingual data exists from the Natural Language Processing and Information Retrieval communities, there is comparatively very little in the database context. The only papers we are aware of are the implementations of multilingual database systems for Arabic and CJK languages that are detailed in [12] and [15]. These two papers focus on implementations of features in databases to support specific requirements of the respective languages, such as storage of composite characters, and features for complex sub-string searches, but do not extend to general purpose solutions.

The performance of various algorithms in compressing large Unicode files are discussed in [4], [8] and [26]. However, these algorithms are not suited for compressing attribute data. BOCU [22] and Multicode [19] discuss encodings that reduce the size of even small Unicode strings, but neither of them support random access of sub-strings, which is necessary for most database operations. A framework for storing shared lexical resources for databases is discussed in [27], but the focus is on improving the efficiency of administration and not query performance.

4 Support for Multilingual Data

In this section, we outline briefly the multilingual support specified by the SQL standards and the implementation approaches taken by different database vendors.

4.1 SQL Standards

SQL-92 [16] was the first standard that specified SQL features for multilingual support, and the current SQL:1999 Standard [11] [17] has largely left it unmodified. Currently, SQL specifies a new data type – *NATIONAL CHAR* (referred to as *NChar*) – large enough to store characters from any national character set. However, the NChar data type is not a core requirement in SQL:1999. The NChar data type may be

defined and manipulated similar to the normal character data type and may be used in all character predicates. The storage format of NChar is left unspecified, though Unicode is considered to be the primary candidate for future SQL specification. SQL also specifies that new repertoires may be defined by the users and that a column may be restricted to hold only characters from a specific repertoire, whether system-defined or user-defined. Finally, the standard specifies that comparison and sorting of strings are meaningful only *within* a repertoire.

4.2 Database Systems

Table 1 provides a comparison of the multilingual features supported by a suite of popular database systems including Oracle 9*i* (9.0.1), IBM DB2 Universal Server (7.1.0), Microsoft SQL Server (8.00.194), MySQL (4.0.3 *Beta*). The information provided in this comparison is gathered from white papers, product literature and other information published in their respective web-sites [13] [18] [20] [21]. Due to legal restrictions, the databases are randomly identified only as α, β, γ and δ.

All these database systems, except for δ, support multilingual storage using either Unicode or UCS-2. In δ, multilingual data may be stored only using *binary* data type. All systems support multilingual specification at all levels – schema, table, record and attribute, while β 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 as no explicit language identification is available at the attribute level. All of them pre-define *collations*, the lexical resource required to sort the data for user presentation and for internal indexes. Though the SQL standard specifies user-defined collations, most database systems haven't implemented this feature as yet. User defined collations may be added to the δ system by source changes. The multilingual query processing is supported along the same lines as that for standard database character sets, using SQL predicates. Though minor differences exist in the different database systems, each vendor has a road map to support the multilingual requirements specified in the SQL standards.

Only binary comparison is used to compare strings in different scripts in all database systems. Currently, no database system supports cross-language querying of data – that is, searching across different languages for a given query string. Support for linguistic querying of text data is not uniform among the databases due to the fact that the SQL standard has not yet specified guidelines for such content-based retrieval. A variety of statistical and natural language processing techniques are employed, though such capabilities are currently restricted to a handful of languages.

5 Multilingual Performance Study

In this section, we describe a test framework for measuring the query performance of the aforementioned database systems with respect to multilingual data.

5.1 Experimental Setup

5.1.1 Hardware Environment

A stand-alone Intel Pentium 4 (1.7 GHz) system with 512MB memory, 40 GB disk, and 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.

5.1.2 Database Environment

Three of the aforementioned database systems were evaluated in our performance study. Due to legal restrictions, we identify them randomly as *A*, *B* and *C* in the remainder of this discussion. The database systems were installed with default configurations as suggested by the vendor-provided installation scripts. All three 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 data type, we found no other database system parameters that are specifically designated for multilingual character sets.

The **TPC-H benchmark** [23] data generator was used to generate a large database for the the study. A specific table (*partsupp*) that stores the part-supplier relationship was modified further, as shown in Figure 3, for our experiments. Specifically, two different tables – *partsuppChar* and *partsuppNChar*, with attributes in Char and NChar (in *UTF-16* format) data types, 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. After population, each of the tables held the same information as the original *partsupp* table, but with keys that are in Char or NChar data types, 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. Thus,

Database	System α	System β	System γ	System δ
Storage Format	Supports Unicode 3.01 UTF-8 / 16	Supports UCS-2 Supports UTF-8 for XML	Supports Unicode 3.01 UTF-8 / 16	No Unicode support
Support Level	At schema, table record and attr levels	At schema, table, record and attr levels; Supports catalog	At schema, table record and attr levels	At Schema, table record and attr levels
Collation Sequence	Pre-defined;	Pre-defined Also uses OS Collations.	Pre-defined	Pre-defined; User definable via source changes
Indexing	Using pre-defined Collations; Uses Unicode Algorithms	Using pre-defined Collations; Uses Unicode Algorithms	Using pre-defined Collations; Uses Unicode Algorithms	Using pre-defined & user-defined Collations
Query Processing	NChar can use all Char predicates	NChar can use all Char predicates	NChar can use all Char predicates	Limited to *Binary* predicates
Locale	About 50 Locales pre-specified	Uses all Locale specified in OS	About 40 Locale pre-specified	About 23 Locale pre-specified

Table 1: **Comparison of Database Systems vis-a-vis Multilingual Support**

the performance of a given query on each of these tables is indicative of performance of the operators on each of the data types.

Finally, a common table, *partsuppCom*, was created by adjoining all the attributes of the above individual tables. While the queries on the Char and NChar tables provide differential performance between the data types including the I/O costs, queries on the common table isolate the differential performance solely due to *in-memory processing*, since the queries need to access the same database blocks irrespective of the data type on which the query was issued. Hence, queries on the common table provide a lower bound on differential performance between the data types.

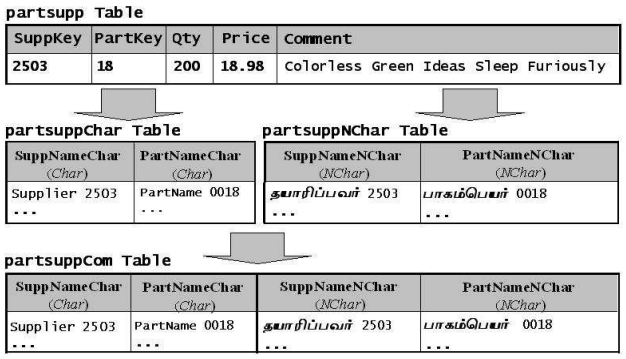

Figure 3: **Data Setup for Performance Study**

The tables were populated with four 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.

5.1.3 Queries for Performance Measurement

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 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:

```
select count(*) from partsuppCom
where { suppNameChar  } <= {    'Part 200000'    }
      { suppNameNChar }    { '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 }       { suppNameNChar }
and P1.{ partNameChar  } <> P2.{ partNameChar  }
       { partNameNChar }       { 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. The query run time was measured as the wall-clock time, using database time-stamps. The average of the run times from several runs was taken as the final run time of a query. 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.

5.2 Performance Metrics

We measured three different performance metrics, to quantify the differential performance of the database operators and the optimizer, as outlined below.

5.2.1 Operator Performance

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

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

where T_{Char} and T_{NChar} are the run times for the Char and NChar data types, respectively. This metric measures the performance overhead of operators on multilingual data in Unicode with respect to default character data in ASCII. A figure close to *zero* indicates equitable performance between Char and NChar data types.

5.2.2 Multilingual Efficiency

We also define an aggregate metric for a database system, *Multilingual Efficiency* (ME_{DBMS}), as:

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

where G_{NChar} is the geometric mean of the run times of operators on NChar data and G_{Char} is the geometric mean[1] of the run times of operators on Char data. While the run times from a complete set of operators will model this metric accurately, we use the run time figures for the following 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.

[1]Similar to other database benchmarks (e.g. Bucky [6]), we use the geometric mean to ensure that all queries are represented in the final metric, independent of the scales of their run times.

5.2.3 Optimizer Prediction Accuracy

In addition to measuring operator run times, we also recorded the optimizer estimate of the cost, to assess the relative accuracy of the optimizer between Char and NChar data types. We define the optimizer metric for an operator, *Multilingual Prediction Equity* (MPE_{Oper}), as:

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

where O_{Char} and T_{Char} are the optimizer estimate and the actual run time of the operator for $Char$, while O_{NChar} and T_{NChar} are the corresponding numbers for $NChar$.

The *MPE* metric measures how equitable the optimizer is between the two character data types, by comparing the ratio of optimizer prediction to the ratio of actual performance. A *MPE* value close to 1 indicates equitable prediction accuracy between Char and NChar data types, while numbers significantly different from 1 indicate non-uniform prediction accuracies.

6 Performance Results

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

6.1 Space Overheads

We found, as expected, 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 (UTF-16) format. Curiously, however, 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 indicated by the minimal difference in the storage size between the two formats ($< 1\%$) and a very slight query performance degradation ($\approx 4\%$).

6.2 Separate Table Processing

When the Char and NChar data types 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, 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 we run all queries 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.

6.3 Common Table Processing

In Table 2, we present the performance of the various operators when the queries were run on the *partsupp-Com* common table, forcing the same database blocks to be accessed, irrespective of the data type on which the query was issued. This means that the performance differentials are solely due to in-memory processing.

Database System	Char Query Time (*Sec*)	NChar Query Time (*Sec*)	MRO_{Oper} (%)	MPE_{Oper}
Table Scan Operator				
A	50	136	172	0.37
B	116	154	32.8	0.75
C	232	246	5.90	0.94
Sort Operator				
A	78	142	80.7	1.30
B	159	235	47.8	0.68
C	352	431	22.4	1.01
Index Create Operator				
A	214	259	21.1	*NA*
B	457	591	24.9	*NA*
C	388	538	38.7	*NA*
Index Scan Operator				
A	2.73	4.78	75.1	0.38
B	8.51	11.4	35.3	1.55
C	3.33	6.54	96.7	0.31
Join (Sort-Merge) Operator				
A	1156	2198	91.5	0.89
B	841	1304	55.0	1.20
C	852	1143	34.2	0.95
Join (Hash) Operator				
A	4558	11848	159.9	1.26
B	576	778	35.1	0.75
C	754	971	28.8	1.22
Join (Nested-Loop) Operator				
A	799	823	3.1	0.97
B	323	334	3.4	0.97
C	144	230	59.3	1.16

Table 2: **Performance of Basic Operators**

Table Scan Operator: For the *Table-Scan* operator, very similar performance for Char and NChar should be expected, since the same database blocks are accessed for both the queries and the selection is done on the fly. While we observe that systems B and C do exhibit this behavior, for system A, however, there is a very substantial difference.

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

Index Create Operator: All three database systems were slower in building index on 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 all three systems have significant deterioration in NChar performance, with systems A and C being especially slow. 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, we evaluated three standard implementation techniques: *Sort-Merge*, *Hash* and *Nested-Loop*. 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, time-wise.

In Table 2, we see that for all these various join implementations, there are substantial performance differences between NChar and Char. Specifically, the join queries are 35% to 90% slower for *Sort-Merge*, 25% to 160% for *Hash*, and up to 60% for *Nested-Loop* across the database systems.

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 3.

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

Table 3: **Multilingual Efficiency**

We see here that all the database systems are inequitable with a wide variation in relative performance, indicated by the ME values ranging from 0.57 to 0.76, implying that the systems are 33% to nearly 100% slower in handling multilingual data.

6.4 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 also provides the optimizer metric, *Multilingual Prediction Equity*, for each of the database operators (except Index-Create, which is a DDL statement).

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 data types; 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. 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.

6.5 Overall Performance Analysis

Given that all the database systems were slow in processing multilingual data, we conducted a series of experiments to understand the trend of and reasons for the slowdown, in order to pinpoint the sources of inefficiency. We selected the database system that exhibited the most inequitous performance, that is, system A, for this study.

6.5.1 Slowdown vs. 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. Specifically, we ran the table scan and join (*Sort-Merge* and *Hash*) queries 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 results for this experiment are shown in Figure 4, 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 multilingual data over ASCII data. As the string length increases, the variable cost of string comparison becomes significant, dominating the function overheads, and hence the differential performance reduces.

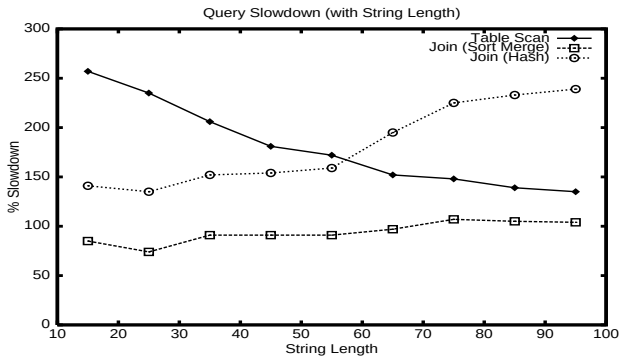

Figure 4: **Query Slowdown with String Size**

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.

6.5.2 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 we assumed that the same code path will be taken for each of the above queries irrespective of the data types on which the queries were issued, as long as the plans are the same. Under the above assumption, the slowdown between Char and NChar data types 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 data type (Char vs. NChar) 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 data type, 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. We ran the scan and join queries on each data type as before to find any variation in performance which can be attributed to data type specific processing. The NChar queries are slower by about 10% indicating that ΔT_{Type} is small, but not insignificant.

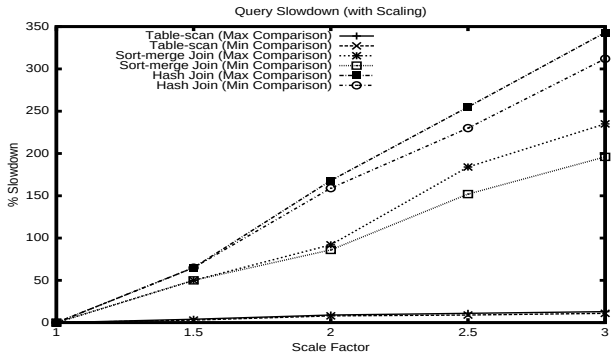

Figure 5: **Query Slowdown with Scaling**

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 Char attribute, are shown in Figure 5, 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 5, match closely with those reported earlier, in Table 2, for NChar that takes twice the space as Char.

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 5 as *Min Comparison*. The difference between the operator performance for maximal and minimal comparison provide the differential cost due to byte comparison itself. We found such cost to be negligible for *Table-scan* operator, confirming our initial observation that disk I/O time is dominating the string processing time. For *Join* operators, such costs are not negligible, and becomes 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 data types in system A as the following: primarily, the differential costs associated with the size of the data ($> 80\%$) and, secondarily, that due to the data type. While 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 we find methods to reduce the storage required, which is explored in the next section.

7 The Cuniform Storage Format

In this section, we propose a simple, compressed representation for Unicode to reduce the multilingual storage size, identified as the prime reason for the slowdown. Our proposal 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: We expect that even in a highly 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 compressed internal representation of Unicode called **Cuniform** (**C**ompressed **Uni**code **Form**at), which splits each Unicode string into two pieces. The first piece stores the information about the common character block from where the characters of the strings occur. This information may be the starting code of that character block or a *Script Identifier* that may be translated to the starting code of the character block. The second piece stores the offsets of

each character in the original Unicode string, in the common character block. 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 only 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.

Original Multilingual Unicode Strings ...

Data	*Unicode* Strings
Narayan	00.E4.00.16.00.27.00.16.00.97.00.16.00.E6
நாராயணன்	0B.A8.0B.BE.0B.B0.0B.BE.0B.AF.0B.A9.0B.CD
RK ನಾರಾಯಣ್	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
நாராயணன்	0B	A8.BE.B0.BE.AF.A9.CD
RK ನಾರಾಯಣ್	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 6: **Skinning of Unicode Strings**

Examples of Unicode to Cuniform transformation are shown in Figure 6. The first two strings (in English and Tamil) have only characters from a single character block each 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 we expect only a small fraction of strings 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 character block, it may not provide any saving over the storage needed for the Unicode format itself. Hence for such languages that have a large repertoire, we store the Unicode strings *as-is*.

7.1 Changes to Database Architecture

The specific database architecture components that are to be modified to handle the Cuniform format are highlighted in Figure 7.

The Data and Record Management module of the server must be enhanced with the functions to handle the *Cuniform* representation and efficient bit-wise operations to convert strings between the Unicode and Cuniform formats. The parser/code generator

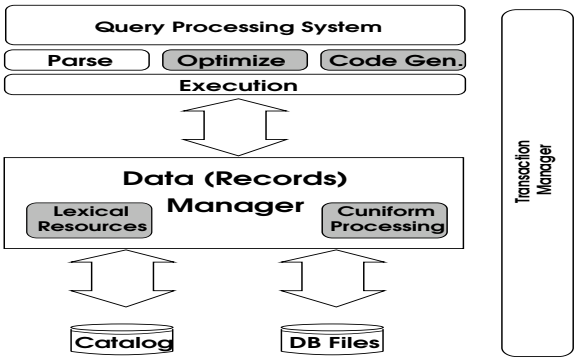

Figure 7: **Changes to Database Architecture**

modules of the query processing code must map the user queries transparently to those that take into account the split image of Cuniform. Most importantly, changes must be made in the optimizer module to accurately model the costs associated with the Cuniform representation. The semantics of conversion of the Cuniform format to other standard types must be specified, though we expect them to be similar to those of NChar data type. Finally, lexical resources, such as Unicode character block information, must be stored in main memory for efficient processing.

7.2 An *Outside the Server* Implementation

Since we lacked access to the source code of the database system A, we implemented a prototype of the Cuniform representation 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 into the common character block of each character in the original 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 *null* was inserted into $Cuni_{sid}$. For output, the $Cuni_{sid}$ was *byte-by-byte* merged with $Cuni_{string}$, thereby reconstructing the original Unicode string.

All SQL queries were 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 Cuniform format, the predicates involving NChar attributes in `WHERE` clause were 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. The details are omitted due to space limitations. Overall, for most operations the Cuniform attributes were used *as-is*, with no conversion to Unicode.

7.2.1 Performance of Cuniform Format

The common *partsuppCom* table that was used for the experiments detailed in Section 5 was augmented with *Part* and *Supplier* names in Cuniform format. We assumed that all the NChar values are from a distinct multilingual script and hence each value was skinnable into Cuniform. All the previous queries were run on this new table and the performance of operators on each of the attributes are provided in Table 4. It should be noted that the figures are different from the previous results, since the new table has two additional attributes (in Cuniform format).

DBMS Opera-tor	Char Time (*Sec*)	NChar Time (*Sec*)	Cuni Time (*Sec*)	*MRO* (*NChar*) (%)	*MRO* (*Cuni*) (%)
T-Scan	52.9	135	55.5	156	5.10
I-Scan	2.89	5.46	5.60	88.3	99.3
SM Join	1188	2371	1370	99.6	15.3
H Join	4575	12534	5591	174	22.2
NL Join	805	834	827	3.60	2.74

Table 4: **Performance of Cuniform Data type**

As can be seen from Table 4, 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 ASCII and the performance of join operators are only moderately slower than that on ASCII. However, the performance of *Index-Scan* on Cuniform attribute is slower than the corresponding Unicode data type, 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 *0.81*. Compared to the ME figure of *0.57* presented in Table 3, the NChar stored using Cuniform improves the multilingual performance of A substantially, bringing it to within 20% of the performance on ASCII. Note further that this is a *conservative* improvement since it has been achieved with an *outside-the-server* implementation. An internal engine implemenation could be expected to narrow the gap even further.

In summary, Cuniform shows that multilingual data may be stored and manipulated almost as efficiently as the default character data in ASCII by using an appropriate internal storage format. Further it retains random access into strings that is needed for database processing and is trivially convertible to Unicode format for data interchange.

7.2.2 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 attributes. This additional piece of information may be used for partitioning the multilingual data, and further used as a *query predicate* to improve the selectivity of the query. Such partitioning of data would make all the operators more efficient, as they need to work only on a subset of the tuples. In such an environment, it may be advantageous to store information in multilingual scripts, rather than in a single script.

7.3 Limitations of Cuniform Format

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

Firstly, 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.

Secondly, languages with character block size more than 256 may be able to store the offsets in a string of bits that is logarithmic in the size of the character block, and hence may still benefit from Cuniform format. However, due to the non-byte aligned nature of the offsets, the performance improvements may not be easily realizable.

8 Conclusions and Future Research

In this paper, we outlined the features required for supporting multilingual text in relational databases and compared a suite of popular database systems in this regard. While all the systems support equivalent multilingual functionality, we found that some features that are needed, such as user-defined collations and multilexical string comparisons, are yet to be supported.

An experimental framework to measure the storage and query processing efficiency of basic database operators on multilingual data was described and the performance of the database systems in this framework were presented. Our experimental results indicate that multilingual data stored in the popular Unicode encod-

ing suffers from a serious space and query processing overhead in all the database systems.

We proposed Cuniform, a compressed storage format that is trivially convertible to Unicode, to overcome such performance overheads. Multilingual data in Cuniform format exhibited marginal space overhead and correspondingly small query overheads, improving significantly the corresponding performance in Unicode format. While our *outside the server* implementation yielded such performance benefits, we expect to see further improvements with an *inside the server* implementation, leading to efficiency that is almost as good as that of ASCII. Further, performance of operators on Cuniform could further be improved in highly multilingual environments by partitioning of data using the explicit script handle available in Cuniform.

The proposed Cuniform representation encodes the script of the multilingual data explicitly, which may be extended further to implement the character set restriction specification of SQL:1999 and to richly extend SQL with multilexical operators.

8.1 Further Research Areas

The SQL:1999 [11] requirement that a column of a table may be restricted to contain data only from a single character set, has not been implemented in any of the databases due to the lack of explicit language handles for stored multilingual strings. We are working in extending Cuniform format for this purpose, using the *Script Identifier* that is available explicitly in Cuniform.

While the SQL:1999 specifies that comparison of multilingual strings across scripts to be meaningless, consider the user requirement in *Books.com* to retrieve all works of a specific author, irrespective of the language of publication. Currently, a query with an exhaustive list of the author's name in different languages would be needed. We therefore see a need for an operator that can compare multilingual attributes across languages. Such an operator may also be used for sorting of multilingual attributes, to create a common index tree, for accessing strings in different languages. We are currently implementing such an operator to solve the real-life *e-Governance* application outlined in [14] for retrieval between English and Indic languages.

Since database servers are the backbones of *e-Commerce* and *e-Governance* applications, multilingual text is becoming a major component of the storage today. But we are not aware of any benchmarks for comparing different database systems with respect to multilingual functionality and performance, similar to the TPC [23] set of benchmarks. We suggest that standard benchmark suites be developed along the lines of experiments outlined in this paper.

References

[1] The Aberdeen Group Ltd., Boston, Massachusetts. http://www.aberdeen.com/.

[2] C. Alexander and L. A. Pal. *Digital Democracy: Policy and Politics in the Wired World.* Oxford University Press, Oxford, United Kingdom, 1998.

[3] J. Allen. *Natural Language Understanding (*2nd Edition*).* Addison-Wesley, 1995.

[4] S. Atkin and R. Stansifer. Unicode Text Compression. *Proc. of 22nd Intl. Unicode Conf., San Jose, California*, 2002.

[5] Bureau of Indian Standards. IS 13194:1991 8–bit Coded Char Set for Information Interchange. 1991.

[6] M. J. Carey *et al.* The BUCKY Object-Relational Benchmark. *Proc. of ACM SIGMOD Intl. Conf. on Management of Data, Tucson, Arizona*, 1997.

[7] The Computer Scope Ltd., Dublin, Ireland. http://www.NUA.ie/Surveys.

[8] P. Fenwick and S. Brierley. Compression of Unicode Files. *Proc. of Data Compression Conf., Snowbird, Utah*, 1998.

[9] ISO. ISO/IEC 10646-1:1993, Universal Multiple-Octet Coded Character Set (UCS). 1993.

[10] ISO. ISO/IEC 8859 Information Processing – 8-bit Single-byte Graphic Coded Character Sets. 1999.

[11] ISO. ISO/IEC 9075-1-5:1999, Information Technology – Database Languages – SQL (Parts 1 – 5). 1999.

[12] R. King and A. Morfeq. Bayan: An Arabic Text Database Management System. *Proc. of ACM SIGMOD Intl. Conf. on Management of Data, Atlantic City, New Jersey*, 1990.

[13] IBM Corporation, Armonk, New York. *http://www.ibm.com.*

[14] A. Kumaran and J. R. Haritsa. On Database Support for Multilingual Environments. *IEEE RIDE Workshop on Multilingual Information Management, Hyderabad, India*, 2003.

[15] C. Lu and K. Lee. A Multilingual Database Management System for Ideographic Languages. *Chinese University of Hong Kong Tech. Rep.*, 1992.

[16] J. Melton and A. R. Simon. *Understanding the New SQL: A Complete Guide.* Morgan Kaufmann, 1993.

[17] J. Melton and A. R. Simon. *SQL 1999: Understanding Relational Language Components.* Morgan Kaufmann, San Francisco, California, 2001.

[18] Microsoft Corporation, Redmond, Washington. *http://www.microsoft.com.*

[19] M. Mudawwar. Multicode: A Multilingual Text Encoding. *IEEE Computer Magazine*, April 1997.

[20] MySQL AB. *http://www.mysql.com.*

[21] Oracle Corporation, Redwood Shores, California. *http://www.oracle.com.*

[22] M. Scherer and M. Davis. BOCU-1: Mime Compatible Unicode Compression. *Unicode Notes #6*, 2002.

[23] The Transaction Processing Council, San Francisco, California. *http://www.tpc.org.*

[24] The Unicode Consortium, Mountain View, California. *http://www.unicode.org.*

[25] The Unicode Consortium. *The Unicode Standard.* Addison-Wesley, 2000.

[26] M. Wolf. Standard Compression Scheme for Unicode. *Technical Standard #6*, Unicode Consortium, 2002.

[27] C. Yip. A Framework for the Support of Multilingual Computing Environment. *Tech. Rep. TR-97-02, University of Hong Kong*, 1997.