A KOREAN SEARCH PATTERN IN THE LIKE OPERATION

Sung Chul Park[1], Eun Hyang Lo[1]
[1]*Kyungpook National University, Department of EECS 1370, Sangyeok-dong, Buk-gu, Daegu 702-701, Korea*

Jong Chul Park[2], Young Chul Park[1]
[2]*FusionSoft Co., Ltd. Research and Development 3-Division, 969-7 Dongcheon-dong, Buk-gu, Daegu, 702-250, Korea*

Keywords: SQL, LIKE, string pattern, Korean, Unicode.

Abstract: The string pattern search operator LIKE of SQL has been developed based on English such that each search pattern of English of the operator works for each character in the alphabet of English. For finding Korean, search patterns of the operator can be expressed by both the alphabet and syllables of Korean. As a phonetic symbol, each syllable of Korean is composed either of a leading sound and a medial sound or of a leading sound, a medial sound, and a trailing sound. By utilizing that characteristic of Korean syllables, in addition to the traditional complete-syllable based search pattern of Korean, this paper proposes an incomplete-syllable based search pattern of Korean, as a pattern of the operator LIKE, to find Korean syllables having specific leading sounds, specific medial sounds, or both specific leading sounds and medial sounds. Formulating predicates that are equivalent with the incomplete-syllable based search pattern of Korean by way of existing SQL expressions is cumbersome and might cause the portability problem of applications depending on the underlying character set of the DBMS.

1 INTRODUCTION

The operator LIKE of the database language SQL is a string pattern search operator. By providing the string pattern, the operator can identify column values that match with the string pattern. As pattern characters of the string pattern, the standard SQL (American National Standards Institute 1992; Melton & Simson 1993) permits normal characters and reserved characters. The operator LIKE has been developed based on English such that each search pattern of English of the operator works for each character in the alphabet of English. For finding Korean, search patterns of the operator can be expressed by both the alphabet and syllables of Korean. Once a Korean alphabet is used as a search pattern, the Korean alphabet itself is matched with the pattern, and once a Korean syllable is used as a search pattern, the Korean syllable itself is matched with the pattern.

The string pattern of the operator LIKE allows any combination of consonants and vowels of English alphabet. For example, by the string pattern 'M% Ave% %a%', strings like "*M*aple *Ave*nue, Ev*a*nston" and "*M*artin *Ave*. Chic*a*go" can be matched. Traditionally, the string pattern for Korean syllables has been a complete-syllable based one. For example, finding strings that start with Korean syllable '박' ('Park', as it sounds in English) can be done by a string pattern '박%'. However, the problem of finding Korean syllables having specific combinations of Korean alphabet has not been addressed in the literature. We will come back to the detailed specification of that problem in short right after introducing the alphabet and syllables of Korean.

Modern Korean alphabet consists of 30 consonants ('ㄱ', 'ㄲ', 'ㄳ', 'ㄴ', 'ㄵ', 'ㄶ', 'ㄷ', 'ㄸ', 'ㄹ', 'ㄺ', 'ㄻ', 'ㄼ', 'ㄽ', 'ㄾ', 'ㄿ', 'ㅀ', 'ㅁ', 'ㅂ', 'ㅃ', 'ㅄ', 'ㅅ', 'ㅆ', 'ㅇ', 'ㅈ', 'ㅉ', 'ㅊ', 'ㅋ', 'ㅌ', 'ㅍ', and 'ㅎ', in lexicographic order) and 21 vowels ('ㅏ', 'ㅐ', 'ㅑ', 'ㅒ', 'ㅓ', 'ㅔ', 'ㅕ', 'ㅖ', 'ㅗ', 'ㅘ', 'ㅙ', 'ㅚ', 'ㅛ', 'ㅜ', 'ㅝ', 'ㅞ', 'ㅟ', 'ㅠ', 'ㅡ', 'ㅢ', and 'ㅣ', in lexicographic order). As a phonetic symbol, each Korean syllable is composed either of a leading sound and a medial sound or of a leading sound, a medial sound, and a trailing sound. Only consonants can be used for leading sounds and trailing sounds of Korean syllables. In Korean, the consonants that are used for leading sounds and trailing sounds are called leading consonants and trailing consonants, respectively. For medial sounds of Korean syllables, only vowels can be used and all the Korean vowels are used for the medial sounds. Among the 30 modern Korean consonants, 19 consonants (‘ㄱ’, ‘ㄲ’, ‘ㄴ’, ‘ㄷ’, ‘ㄸ’, ‘ㄹ’, ‘ㅁ’, ‘ㅂ’, ‘ㅃ’, ‘ㅅ’, ‘ㅆ’, ‘ㅇ’, ‘ㅈ’, ‘ㅉ’, ‘ㅊ’, ‘ㅋ’, ‘ㅌ’, ‘ㅍ’, and ‘ㅎ’, in lexicographic order) can be used as leading consonants and 27 consonants (‘ㄱ’, ‘ㄲ’, ‘ㄳ’, ‘ㄴ’, ‘ㄵ’, ‘ㄶ’, ‘ㄷ’, ‘ㄹ’, ‘ㄺ’, ‘ㄻ’, ‘ㄼ’, ‘ㄽ’, ‘ㄾ’, ‘ㄿ’, ‘ㅀ’, ‘ㅁ’, ‘ㅂ’, ‘ㅄ’, ‘ㅅ’, ‘ㅆ’, ‘ㅇ’, ‘ㅈ’, ‘ㅊ’, ‘ㅋ’, ‘ㅌ’, ‘ㅍ’, and ‘ㅎ’, in lexicographic order) can be used as trailing consonants.

Figure 1 illustrates two Korean syllables: one syllable ‘배’ (*abdomen, pear,* or *vessel,* in English) that is composed of a leading sound ‘ㅂ’ and a

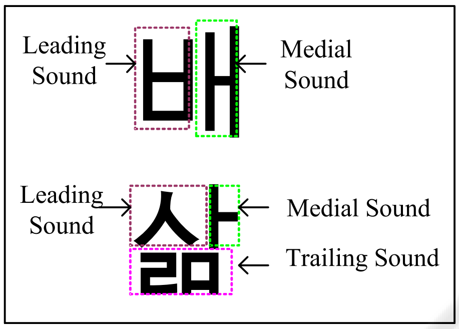
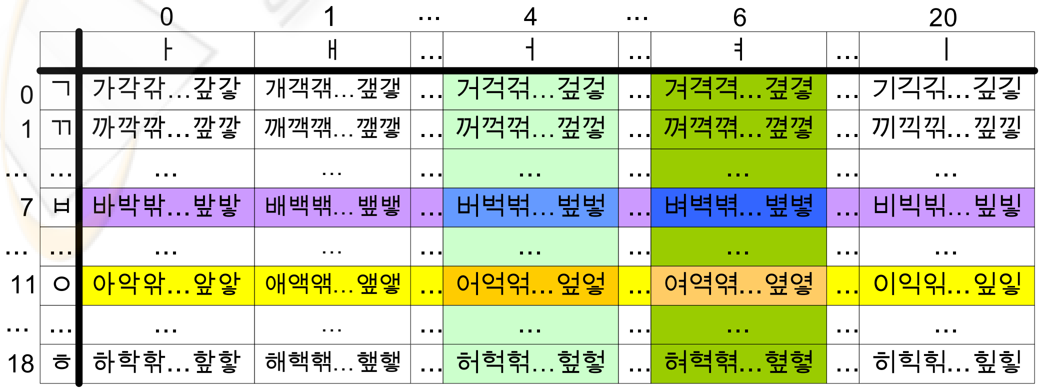


Figure 1: Components of Korean syllables.

medial sound ‘ㅐ’, and another syllable ‘삶’ (*life*, in English) that is composed of a leading sound ‘ㅅ’, a medial sound ‘ㅏ’ and a trailing sound ‘ㄻ’. The lexicographic order among Korean syllables follows the order of <a leading consonant, a vowel, a trailing consonant> that constitute the Korean syllables, which means that it keeps the order of leading consonants; for the same leading consonant, it keeps the order of vowels; and for the same leading consonant and the same vowel, it keeps the order of trailing consonants. In the case of the same leading consonant and the same vowel, a syllable that does not have any trailing consonant precedes syllables that have trailing consonants.

In this paper, we are concerned about finding Korean syllables that have specific leading sounds, specific medial sounds, or both specific leading sounds and medial sounds. Our goal is specifying the combinations of Korean alphabet directly into the string patterns of the operator LIKE without having any notational difficulty. For that purpose, we have devised a two-dimensional table, which we call *the Korean syllable map*. As shown in Figure 2, the Korean syllable map has rows and columns for representing the leading consonants and the vowels of Korean, respectively. Each cell in the map that is formulated by a specific row and a specific column contains all the 28 contiguous syllables that can be constructed with the leading consonant of the row and the vowel of the column. All the rows, columns, and syllables in each cell are arranged according to the lexicographic order of the consonants, vowels, and syllables, respectively. According to that, all the 11,172 modern Korean syllables in Unicode (Unicode, Inc. 2006) can be mapped into the Korean syllable map.

In the Korean syllable map, indexes of rows, which we call *row_indexes*, start from 0 (for the initial consonant ‘ㄱ’) and end with 18 (for the initial consonant ‘ㅎ’), indexes of columns, which we call *column_indexes*, start from 0 (for the vowel ‘ㅏ’) and end with 20 (for the vowel ‘ㅣ’). We call the row of *row_index* i as ROW_i and the column of *column_index* j as COLUMN_j, and the cell of *row_index* i and *column_index* j as $\mathrm{CELL}_{i,j}$. Let the first syllable and the last syllable in $\mathrm{CELL}_{i,j}$ be $\mathrm{FS}_{i,j}$ and $\mathrm{LS}_{i,j}$, respectively. Then the syllables in $\mathrm{CELL}_{i,j}$, ROW_i, and COLUMN_j are in the range of [$\mathrm{FS}_{i,j}$-$\mathrm{LS}_{i,j}$], [$\mathrm{FS}_{i,0}$-$\mathrm{LS}_{i,20}$], and [$\mathrm{FS}_{0,j}$-$\mathrm{LS}_{0,j}$ | $\mathrm{FS}_{1,j}$-$\mathrm{LS}_{1,j}$ | … | $\mathrm{FS}_{18,j}$-$\mathrm{LS}_{18,j}$], respectively. For example, the



		0	1	…	4	…	6	…	20
		ㅏ	ㅐ	…	ㅓ	…	ㅕ	…	ㅣ
0	ㄱ	가각갃…갚갛	개객갟…갶갷	…	거걱걳…겊겋	…	겨격겫…겾곃	…	기긱긳…깊깋
1	ㄲ	까깍깏…깦깧	깨깩깫…꺂꺃	…	꺼꺽꺿…껖껗	…	껴껵껷…꼎꼏	…	끼끽끿…낖낗
…	…	…	…	…	…	…	…	…	…
7	ㅂ	바박밗…밮밯	배백뱏…뱊뱋	…	버벅벇…벞벟	…	벼벽벿…볖볗	…	비빅빇…빞빟
…	…	…	…	…	…	…	…	…	…
11	ㅇ	아악앇…앞앟	애액앣…앺앻	…	어억얷…엎엏	…	여역엯…옆옇	…	이익읷…잎잏
…	…	…	…	…	…	…	…	…	…
18	ㅎ	하학핛…핲핳	해핵햋…햎햏	…	허헉헋…헢헣	…	혀혁혃…혚혛	…	히힉힋…힢힣

Figure 2: The Korean syllable map for Unicode.

syllables in $CELL_{7,4}$, ROW_7, and $COLUMN_4$ are in the range of [$FS_{7,4}$-$LS_{7,4}$] (i.e., [버–벟]), [$FS_{7,0}$-$LS_{7,20}$] (i.e., [바–빟]), and [$FS_{0,4}$-$LS_{0,4}$ | $FS_{1,4}$-$LS_{1,4}$ | … | $FS_{18,4}$-$LS_{18,4}$] (i.e., [거–겋 | 꺼–껗 | … | 허–헣]), respectively.

Based on the Korean syllable map, the problem of finding Korean syllables that have specific leading sounds, specific leading sounds and medial sounds, or specific medial sounds becomes finding syllables that are located in the specific ROWs, CELLs, or COLUMNs of the Korean syllable map, respectively. According to that, string patterns for finding Korean syllables that have specific leading sounds, specific leading sounds and medial sounds, or specific medial sounds are called string patterns of Type_ROW, Type_CELL, or Type_COLUMN, respectively.

A simple solution of specifying those three types of Korean search patterns could be expressing each of them by the use of regular expressions. That solution looks fine and can be executed by any DBMS that supports regular expressions in one or another form. For example, consider finding words having at least two Korean syllables, where the first syllable has 'ㅂ' as its leading sound and the second syllable has 'ㅐ' as its medial sound such as strings like '박애' (*benevolence*, in English) and '비행기' (*airplane*, in English). The regular expression for such strings becomes '[바-빟][개-갷 | 깨-깧 | 내-냏 | 대-댛 | 때-떓 | 래-랳 | 매-맿 | 배-뱋 | 빼-뺗 | 새-샣 | 쌔-쌯 | 애-앻 | 재-잫 | 째-쨓 | 채-챟 | 캐-캫 | 태-탷 | 패-팧 | 해-핳]'. However, that simple solution has the following problems.

First, the simple solution might cause the portability problem of SQL applications. This is because the number of Korean syllables that are supported could be different depending on the underlying character sets. For example, in Unicode, all the 11,172 modern Korean syllables are specified. However, in KS X 1001 (Korean Standards Information Center 2002) that is one of the Korean standards and also the most widely used character set with Unicode in Korea, only 2,350 Korean syllables that are commonly used in Korea today are specified. Because of the discrepancy in the number of supporting Korean syllables, once we build Korean syllable maps for them, many of the first syllables and the last syllables in the cells of the two Korean syllable maps are different. For example, in Unicode, $LS_{7,20}$ is '빟', but, in KS X 1001, it is '빛'. Provided that a character in a regular expression exceeds the range of the underlying character set of the DBMS, the DBMS such as ORACLE 10g (ORACLE 2005b) raises a run time error. Therefore, once an application that runs on the character set of Unicode is moved to the environment that uses the character set of KS X 1001 or the other way, the values of $FS_{i,j}$ and $LS_{i,j}$ must be modified manually. This means that SQL applications adopting that simple solution might have the portability problem.

Second, the simple solution might have the performance problem in executing search patterns of Type_COLUMN. As far as search patterns of Type_ROW and Type_CELL are concerned, they can be executed by checking whether a certain syllable lies in the specified range of syllables. However, the Type_COLUMN search pattern has 19 ranges of syllables such that multiple comparisons should be done to check whether a certain syllable matches with the search pattern.

This paper presents an intuitive, uniform, and simple way of expressing the three types of Korean search patterns that is free from the portability problem of SQL applications. Algorithms for the execution of the Korean search pattern are also presented based on Unicode. Without loss of generality, we assume that 30 consonants and 21 vowels of modern Korean alphabet are arranged on the keyboard systems that take the Korean standard KS X 5002, "Keyboard layout for information processing" (Korean Standards Information Center 1982). Because of that, leading consonants and trailing consonants are not arranged separately on the keyboard and can be discriminated by some appropriate automaton while building Korean syllables. We do not consider archaic characters of Korean. The performance evaluation of the Korean search pattern is not main concern of this paper. Comparing a Korean syllable with the Korean search pattern needs only one range check or a value check. Regular expressions that are equivalent with the Korean search pattern need the same number of comparisons for the search patterns of Type_ROW and Type_CELL. However they need 19 range checks for the search pattern of Type_COLUMN. It is clear that one scheme with a smaller number of comparisons is faster than another with a larger one. This paper does not present performance of the algorithms for such reasons.

The rest of this paper is organized as follows. In Section 2, we introduce the Korean search pattern and its expression. In Section 3, schemes that identify Korean search patterns and matching algorithms for each type of the Korean search pattern are provided. String match algorithms related to the Korean search pattern are presented in Section 4. Section 5 concludes this paper.

2 KOREAN SEARCH PATTERN AND ITS EXPRESSON

The Korean search pattern consists of a predecessor and a searcher. The predecessor of it could be an escape character of the operator LIKE or a newly reserved character (for example, ‘$’ after defining it as a reserved character). The searcher of it could be (1) a leading consonant (i.e., Type_ROW), (2) a syllable that consists only of a leading consonant and a vowel (i.e., Type_CELL), or (3) a vowel (i.e., Type_COLUMN). Each of these searchers matches with (1) Korean syllables that have the specified leading consonant as their leading sounds (i.e., the syllables in a specific ROW), (2) Korean syllables that have the leading consonant and the vowel of the specified Korean syllable as their leading sounds and medial sounds respectively (i.e., the syllables in a specific CELL), or (3) Korean syllables that have the specified vowel as their medial sounds (i.e., the syllables in a specific COLUMN), respectively. In the rest of this paper, an escape character is used as the predecessor, and if not declared, ‘ \ ’ is assumed declared as an escape character.

Trailing consonants that are not used for leading consonants, and syllables that consist of leading sounds, medial sounds and trailing sounds are not included in the searcher. The reason of excluding the trailing consonants that are not used for the leading consonants from the searcher is two-fold. First, we assume that the request of finding Korean syllables that have a specific trailing consonant might be rare. Second, because of the keyboard systems that we take, once consonants that are commonly used for the leading consonants and the trailing consonants are specified in the string pattern, it is impossible to identify whether they are leading consonants or trailing consonants by looking at them only. Because of that, once any of the trailing consonants that are not used for the leading consonants is specified right after the predecessor of the Korean search pattern, we treat the pattern exactly the same way as specifying the consonant only. Once syllables that consist of leading sounds, medial sounds, and trailing sounds are specified right after the predecessor of the Korean search pattern, the request is treated exactly the same way as specifying that syllable only. This is because only that syllable can be matched with that pattern.

Example 1. “Retrieve employees whose addresses start with Korean syllables having ‘ㅂ’ as their leading sounds.”

By using the Korean search pattern, the request can be done by the query “SELECT * FROM employee WHERE address LIKE ‘ \ ㅂ%’ ESCAPE ‘ \ ’;”. Addresses of ‘바다’ (*sea,* in English), ‘보석상자’ (*jewel box,* in English), and ‘뷰티샵’ (*beauty shop,* in English) match with the pattern. In Unicode, 588 Korean syllables having ‘ㅂ’ as their leading consonants are located in the range between ‘바’ and ‘빟’ and they are less than the Korean syllable ‘빠’. Therefore, the above expression is equivalent with “address >= ‘바’ and address < ‘빠’”. The above predicate of Korean search pattern can be represented equivalently either by the use of regular expression “REGEXP_LIKE(name, ‘^[바-빟]’)” of ORACLE 10g or by the extended syntax of operator LIKE “name LIKE ‘[바-빟]%’” of MS SQL Server 2005 (Microsoft 2006).

Example 2. “Retrieve employees whose names consist of exactly three Korean syllables, where the first syllable has ‘ㅂ’ as its leading consonant, the second syllable has ‘ㅇ’ as its leading consonant and ‘ㅕ’ as its vowel, and the third syllable has ‘ㅓ’ as its vowel.”

It can be done by the query “SELECT * FROM employee WHERE name LIKE ‘ \ ㅂ \ 여 \ ㅓ ’ ESCAPE ‘ \ ’;”. The name of ‘박영철’ (‘Park Young Chul’, as it sounds in English) matches with the pattern. Unfortunately, expressing the Korean search pattern by using range predicates can be very cumbersome. For instance, according to the lexicographic order and the arrangement of Korean syllables in Unicode, the above expression is equivalent with “name LIKE ‘___’ and ((name >= ‘바여거’ and name <= ‘바여겋’) or (name >= ‘바여꺼’ and name <= ‘바여껗’) or … or (name >= ‘빟옇허’ and name <= ‘빟옇헣’)”. To formulate that equivalent expression however, since there are 588 Korean syllables having ‘ㅂ’ as their leading consonants, 28 Korean syllables having ‘ㅇ’ as their leading consonants and ‘ㅕ’ as their vowels, and 19 ranges of Korean syllables having ‘ㅓ’ as their vowels, 312,826 (that is, 588 * 28 * 19) range predicates must be enumerated and connected with the OR operator. This large number of predicates means that, apart from the question of troublesome in making that equivalent expression, the query processing time might become tremendous. By the use of regular expression “REGEXP_LIKE(name, ‘^[바-빟][여-옇][거-겋 | 꺼-껗 | 너-넣 | 더-덯 | … | 허-헣]$’)” of ORACLE 10g and the extended syntax of operator LIKE “name LIKE ‘[바-빟][여-옇][거-겋 | 꺼-껗 | 너-넣 | 더-덯 | … | 허-헣]’” of SQL Server 2005, both of them are also equivalent with the above predicate of Korean search pattern. However, formulating both the regular expression and the extended string pattern that have 19 ranges is not easy and is error prone.

Example 3. “Retrieve employees whose names start with at least one arbitrary character that is immediately followed by exactly three Korean syllables, where the first syllable has ‘ㅂ’ as its leading consonant, the second syllable has ‘ㅇ’ as its leading consonant and ‘ㅕ’ as its vowel, and the third syllable has ‘ㅓ’ as its vowel, which in turn are followed by at least one character.”

It can be done by the query “SELECT * FROM employee WHERE name LIKE ‘%_ \ ㅂ \ 여 \ ㅓ_%’ ESCAPE ‘ \ ’;”. Formulating equivalent expressions by using range predicates is not possible in this case. By the use of regular expressions or extended string patterns, both “REGEXP_LIKE(name, ‘(.)+[바-빟][여-옇][거-겋 | 꺼-껳 | 너-넣 | 더-덯 | … | 허-헣](.)+’)” and “name LIKE ‘%_[바-빟][여-옇][거-겋 | 꺼-껳 | 너-넣 | 더-덯 | … | 허-헣]_%’” are also equivalent with the above predicate of Korean search pattern. Note also that these equivalent expressions are not easy to formulate and are also error prone.

3 KOREAN SEARCH PATTERN AND ITS MAPPING WITH KOREAN SYLLABLES

This section presents identifying schemes of Korean syllables that match with a given Korean search pattern. Before illustrating those schemes, a short introduction to the placement of Korean alphabet and Korean syllables in Unicode comes first.

Unicode follows the lexicographic order among consonants, vowels and syllables of Korean in assigning code points to them (Unicode Inc. 2005c, 2005d, 2005e). In Unicode, in the range between 0xAC00 and 0xD7AF, which is called *Hangul Syllables* (Unicode Inc. 2005e), 11,172 modern Korean syllables are encoded and are arranged according to the lexicographic order. Because of that, the smallest Korean syllable ‘가’ is arranged into the code point 0xAC00 (which we call *KS_START*) and the biggest Korean syllable ‘힣’ is arranged into the code point 0xD7A3 (which we call *KS_END*). In Unicode, modern and archaic consonants and vowels of Korean are provided in two different ranges: the range between 0x1100 and 0x11FF, which is called *Hangul Jamo* (Unicode Inc. 2005d) and the range between 0x3130 and 0x318F, which is called *Hangul Compatibility Jamo* (Unicode Inc. 2005c). The characters in Hangul Compatibility Jamo are provided solely for the compatibility with the Korean standard KS X 1001:1998 (Unicode Inc. 2005b). The 30 consonants and 21 vowels of modern Korean characters are declared both in Hangul Jamo and in Hangul Compatibility Jamo. They are arranged according to a predefined lexicographic order. In Hangul Jamo, 19 leading consonants are arranged in the range between 0x1100 (which we call *L_START*) and 0x1112 (which we call *L_END*), 27 trailing consonants are arranged in the range between 0x11A8 and 0x11C2, and the whole vowels are arranged in the range between 0x1161 (which we call *V_START*) and 0x1175 (which we call *V_END*). In Hangul Compatibility Jamo, the whole consonants are arranged in the range between 0x3131 (which we call *CON_START*) and 0x314E (which we call *CON_END*) and the whole vowels are arranged in the range between 0x314F (which we call *VOWEL_START*) and 0x3163 (which we call *VOWEL_END*).

Let *ROW_SIZE* be the number of Korean syllables in a row, *CELL_SIZE* be the number of Korean syllables in a cell, *ROW_COUNT* be the number of rows, and *COLUMN_COUNT* be the number of columns of the Korean syllable map. Actually, *ROW_SIZE* is 588, *CELL_SIZE* is 28, *ROW_COUNT* is 19, and *COLUMN_COUNT* is 21.

Observation 1. According to the placement of Korean syllables in Unicode, we can have the following facts about the Korean syllable map.

(1) $FS_{0,0}$=*KS_START*.

(2) For *row_index* i and *column index* j, where 0≤i<*ROW_COUNT* and 0≤j<*COLUMN_COUNT*, $FS_{i,j}=FS_{0,0}+i*ROW\_SIZE+j*CELL\_SIZE$, and $LS_{i,j}=FS_{i,j}+CELL\_SIZE-1$.

Observation 2. For a Korean syllable of a Unicode code point S, we can have the following facts.

(1) Let *row_index*(S) be the *row_index* of S in the Korean syllable map. Then, *row_index*(S) = $(S - FS_{0,0})/ROW\_SIZE$.

(2) Let *column_index*(S) be the *column_index* of S in the Korean syllable map. Then, *column_index*(S) = $((S–FS_{0,0})/CELL\_SIZE)$ % *COLUMN_COUNT*.

(3) Let IS_FS(S) be a Boolean function that identifies whether a syllable S is the first syllable of a certain cell in the Korean syllable map. In other words, IS_FS(S) becomes TRUE only when the syllable of S is composed of a leading consonant and a vowel only. Then, IS_FS(S) becomes TRUE only when $(S - FS_{0,0})\%CELL\_SIZE$ is equal to 0. Otherwise, that

syllable is composed of a leading consonant, a vowel, and a trailing consonant.

Observation 3. A searcher of a Korean search pattern can be classified into six different groups depending on the code point x of the searcher.

(1) If x is between *L_START* and *L_END*, x is a leading consonant of Hangul Jamo and x – *L_START* becomes *row_index* of the leading consonant.

(2) If x is between *CON_START* and *CON_END*, x is a consonant of Hangul Compatibility Jamo. To identify *row_indexes* of leading consonants in Hangul Compatibility Jamo, we put an array CON_Array. That array has 30 entries and the ith entry contains *row_index* of the ith consonant if that consonant is a leading consonant and contains -1 otherwise. The CON_Array is shown below.

```
static const int CON_Array[] = {
   0, 1, -1, 2, -1, -1, 3, 4, 5, -1,
  -1, -1, -1, -1, -1, -1, 6, 7, 8, -1,
   9, 10, 11, 12, 13, 14, 15, 16, 17, 18
}
```

From that array, for the code point of x between *CON_START* and *CON_END*, if CON_Array[x – *CON_START*] is not –1, x is a leading consonant and that value becomes *row_index* of x. Once that value is –1, x is not a leading consonant such that ‘ \ x’ is handled as ‘x’. For example, for a consonant ‘ㄷ’ of Hangul Compatibility Jamo, which has the Unicode code point 0x3137, CON_Array[0x3137 - *CON_START*], i.e., CON_Array[6] is the entry for the consonant and the value 3 of the entry means *row_index* of the consonant. However, CON_Array[15] is the entry for a consonant ‘ㅀ’ and the value –1 of the entry means that ‘ㅀ’ is not a leading consonant.

(3) If x is between *KS_START* and *KS_END*, x is a Korean syllable. According to Observation 2-(3), if IS_FS(x) is TRUE, x is a code point of a syllable that is composed of a leading consonant and a vowel only. Otherwise, x is a code point of a syllable that is composed of a leading consonant, a vowel and a trailing consonant such that ‘ \ x’ is handled as ‘x’.

(4) If x is between *V_START* and *V_END*, x is a vowel of Hangul Jamo and x – *V_START* becomes *column_index* of the vowel.

(5) If x is between *VOWEL_START* and *VOWEL_END*, x is a vowel of Hangul Compatibility Jamo and x – *VOWEL_START* becomes *column_index* of the vowel.

(6) If x is not in any one of the above five ranges, the pattern is not a Korean search pattern.

For a Type-ROW Korean search pattern of searcher S, there could be two schemes of finding matching Korean syllables. The *row_index* of S, say i, can be found according to Observation 3-(1) or 3-(2) depending on the value of S. Let W be the code point of the syllable to be compared. One scheme is checking whether i = *row_index*(W). The other scheme is setting up the range as [$FS_{i,0}$-$LS_{i,20}$] and then check whether W lies in that range. We take the second scheme.

Once the type of a Korean search pattern is Type-CELL of searcher S, there could be two schemes of finding matching syllables. Let i be *row_index*(S), j be *column_index*(S), and W be the code point of the syllable to be compared. The first scheme is checking whether i = *row_index*(W) and j = *column_index*(W). The second scheme is setting up the range as [$FS_{i,j}$-$LS_{i,j}$] and then check whether W lies in that range. We take the second scheme.

Once the type of a Korean search pattern is Type-COLUMN of searcher S, the following scheme for finding matching syllables is used. The column index of S, say j, can be found according to Observation 3-(4) or 3-(5) depending on the value of S. Let W be the code point of the syllable to be compared. Our scheme is checking whether j = *column_index*(W). In addition to that scheme, for the formulation of the index search range, the following scheme for finding boundaries is also used. Let the code point of an arbitrary syllable whose *column_index* is the same as the searcher of Type_COLUMN of *column_index* j be W, then the range of code points of W be [$FS_{0,j}$-$LS_{18,j}$]. Even though that range encompasses wide space unnecessarily, it could be helpful for restricting the search space of the index search. Note that this range scheme is used only for the search of key values in indexes.

4 SYLLABLE COMPARISON SCHEMES AND PATTERN MATCH ALGORITHMS

Without loss of generality, we take the UTF-8 encoding scheme (Unicode Inc. 2005a) for the representation of characters. There could be two different schemes of comparison between the syllable to be compared and boundary syllables (i.e., $FS_{i,j}$ and $LS_{p,q}$, which we call LB and UB respectively in the rest of this paper) of the Korean search pattern: one is comparing them in Unicode code points and the other is comparing them in the UTF-8 encoding scheme. Korean characters are assigned into the Basic Multilingual Plane (BMP) region of Unicode. Therefore, even though the first scheme has the burden of transforming byte sequences into Unicode code points, since the code points can be stored in variables of unsigned short data type, the comparison itself can be done promptly. The second scheme keeps LB and UB in byte sequences according to the UTF-8 encoding scheme. Therefore, it does not have the burden of transforming Korean syllables into Unicode code points. However, for the comparison of byte sequences within each code unit, (1) in the case of encoding schemes of UTF-8, UTF-16BE, and UTF-32BE, those bytes have to be compared in the forward direction and (2) in the case of encoding schemes of UTF-16LE and UTF-32LE, those bytes have to be compared in the reverse direction.

We take the second scheme for the comparison of Korean syllables and it works as follows. After transforming the searcher of the Korean search pattern of the UTF-8 encoding scheme into a Unicode code point, from that code point, the type of the pattern and *column_index* (if available) are identified, LB and UB of the searcher are decided in Unicode code points, and then these two values are transformed into byte sequences of the UTF-8 encoding scheme. For the matching of Korean syllables in the string to be compared, those syllables are compared directly with LB and UB. For the generation of *column_index* (if necessary), the syllables are transformed into Unicode code points. In the rest of this paper, we assume that all data structures and algorithms take this policy, and all characters in the string pattern and the stored data are either ASCII characters or modern Korean characters.

The string pattern of the operator LIKE should be normalized before performing any matching operation. For that purpose, the string pattern is stored in an array, say StringPattern, and the normalized string pattern is kept in an array, say zPattern. In this paper, we consider normal characters, reserved characters (‘%’ and ‘_’), and escape characters for the string patterns. We do not consider string patterns like ‘[ ]’ and ‘[^]’, which are supported by some commercial DBMSs such as MS SQL Server. Upon including the Korean search pattern, we put an array zPatternFlag to keep types of Korean search patterns, put arrays LBS and UBS to keep LB and UB of each Korean search pattern respectively, and have the following two additional rules for the normalization. Note that each Korean character takes three bytes in the UTF-8 encoding scheme.

(1) Let $zPattern_k$ represent the k^{th} character in the array zPattern. $zPatternFlag_k$ takes the same number of bytes as $zPattern_k$ holds. If $zPattern_k$ is not a Korean search pattern, $zPatternFlag_k$ takes 0. However, if $zPattern_k$ is a Korean search pattern, $zPatternFlag_k$ holds the information of <type, *range_index*, *column_index*>, where *type* means the type (1 for Type_ROW, 2 for Type_CELL, and 3 for Type_COLUMN) of the Korean search pattern, *range_index* identifies the index of arrays LBS and UBS that hold LB and UB of the pattern $zPattern_k$, and *column_index* holds the *column_index* of the search pattern only when the search pattern is one of type Type_COLUMN.
(2) For each Korean search pattern in the string pattern, the following steps have to be done. The predecessor of the pattern is not stored in zPattern and only the searcher of the pattern is stored in zPattern. Let the searcher to be stored in zPattern be the k^{th} character in zPattern. First of all, LB and UB of the searcher are found according to the schemes shown in Section 3 and they are appended into arrays LBS and UBS. Let the index of the values appended in the arrays be *range_index*. If the type of the search pattern is Type_ROW or Type_CELL, <1, *range_index*, 0> or <2, *range_index*, 0> is assigned to $zPatternFlag_k$, respectively. However, if the type of the search pattern is Type_COLUMN, *column_index* of the vowel is calculated and then <3, *range_index*, *column_index*> is assigned to $zPatternFlag_k$.

We have assigned arrays LBS and UBS, and have stored *column_index* in the array zPatternFlag. The reason is simply because a lot of database records should be compared with the given string pattern. If we do not store them, whenever a new database record is met for the string match, the values should be re-calculated. This is not a good idea.

Because of the reserved character ‘%’, the algorithm that executes the matching operation between a string pattern and a string to be compared could be a recursive one. Since discussing the algorithm itself is beyond the scope of this paper, the string match algorithm is simply summarized within the scope of the Korean search pattern. Let the start index of the current pattern in zPattern be k. If zPatternFlag[k] is not 0, the pattern is a Korean search pattern. If zPatternFlag[k] is either 1 (i.e., Type_ROW) or 2(i.e., Type_CELL), zPatternFlag[k+1] has the value of *range_index* of the pattern. Let the value of zPatternFlag[k+1] be r. Then, the range R of Korean syllables that match with that pattern becomes LBS[r..r+2] ≤ R ≤ UBS[r..r+2] and a Korean syllable S to be compared should satisfy the range to be matched with that pattern. If zPatternFlag[k] is 3(i.e., Type_COLUMN), zPatternFlag[k+2] has *column_index* for that pattern. Let the function that finds the Unicode code point of a Korean syllable S in UTF-8 encoding scheme be codepoint(S). Then, for a Korean syllable S, it is declared to be matched with that pattern when (((codepoint(S) - $FS_{0,0}$)/*CELL_SIZE*) % *COLUMN_ COUNT*) is equal to zPatternFlag[k+2]. Otherwise, it is not matched.

5 CONCLUSIONS

This paper proposes three types of Korean search patterns to find Korean syllables having specific leading sounds, specific medial sounds, or both specific leading sounds and medial sounds. The Korean search pattern is expressed in an intuitive, uniform, and simple way such that it can be added into the existing string patterns of the operator LIKE without having any notational difficulty. The expression is free from the portability problem of SQL applications that might be resident in the equivalent regular expressions because of the underlying character sets of the DBMS. Efficient ways of pattern matching for the three types of Korean search patterns are also presented in this paper.

We have implemented the Korean search pattern on two relational DBMSs. One is CellDB that uses Unicode for its character set and takes UTF-8 encoding scheme. The algorithms presented in this paper have been ported directly into the system. The other is BADA-II. The system uses KS X 1001 for its character set such that some modified algorithms of this paper have been implemented into the system. Many commercial DBMSs such as DB2 (Poon & Sud & Chong 2005), Oracle (ORACLE 2005a), and MS SQL Server (Kaplan 2001) support Unicode. Therefore, the Korean search pattern of this paper can be ported into them without having any difficulty.

ACKNOWLEDGEMENTS

This work is financially supported by the Ministry of Education and Human Resources Development (MOE), the Ministry of Commerce, Industry and Energy (MOCIE) and the Ministry of Labor (MOLAB) through the fostering project of the Industrial-Academic Cooperation Centered University.

REFERENCES

American National Standards Institute, 1992. *The Database Language SQL, Standard No. X3.135-1992*, New York.

Kaplan, M., 2001. *International Features in Microsoft SQL Server 2000*, viewed 1 August 2006, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql2k/html/intlfeaturesinsqlserver2000.asp>.

Korean Standards Information Center, 2002. *Code for information interchange (Hangul and Hanja), Standard No. KS X 1001*.

Korean Standards Information Center, 1982. *Keyboard layout for information processing, Standard No. KS X 5002*.

Melton, J., Simon, A. R., 1993. *Understanding the new SQL: A complete guide*, Morgan Kaufmann Publishers, Inc., San Mateo, California.

Microsoft, 2006. *LIKE (Transact-SQL)*, viewed 20 October 2006, <http://msdn2.microsoft.com/en-us/library/ms179859.aspx>.

ORACLE, 2005. *Globalization Support Oracle Unicode database support*, viewed 24 July 2006, < http://www.oracle.com/technology/tech/globalization/pdf/TWP_AppDev_Unicode_10gR2.pdf>.

ORACLE, 2005. *Oracle 10g Downloads*, viewed 20 October 2006, < http://www.oracle.com/technology/software/products/database/oracle10g/index.html>.

Poon, S., Sud, M., Chong, R., 2005. *Understanding DB2 Universal Database character conversion*, viewed 1 August 2006, <http://www-128.ibm.com/developerworks/db2/library/techarticle/dm-0506chong/>.

Unicode, Inc., 2005. *Conformance*, viewed 11 July 2006, <http://www.unicode.org/versions/Unicode4.0.0/ch03.pdf>.

Unicode, Inc., 2005. *East Asian Scripts*, viewed 11 July 2006, <http://www.unicode.org/versions/Unicode4.0.0/ch11.pdf>.

Unicode, Inc., 2005. *Hangul Compatibility Jamo*, viewed 11 July 2006, <http://www.unicode.org/charts/PDF/U3130.pdf>.

Unicode, Inc., 2005. *Hangul Jamo*, viewed 11 July 2006, <http://www.unicode.org/charts/PDF/U1100.pdf>.

Unicode, Inc., 2005. *Hangul Syllables*, viewed 11 July 2006, <http://www.unicode.org/charts/PDF/UAC00.pdf>.

Unicode, Inc., 2006. Unicode Home Page, viewed 5 July 2006, <http://www.unicode.org>.