Surviving Client/Server: Phonetic Matching

by Steve Troxell

Many database systems contain an element that keeps track of people's names: customers, registered owners, donators, voters, etc. Most of the time it is a good idea to avoiding introducing the same individual into the database more than once for related transactions. For instance, an individual might fill out a warranty registration card for a radio and some months later fill out another one for a CD player from the same company. When the second registration comes in, we would like to recognize that the person was already in our database and associate them with two products rather than have two separate customer records each associated with a single product.

This is all fine and obvious, except that for the situation I just described, we have nothing but the person's name and address to match with an existing record. There is no concrete identifier such as an employee number, social security number, driver's license number, etc.

Matching on free-form data becomes problematic when you consider typographical errors, letter transpositions or omissions and spelling substitutions that cannot be completely avoided in manual data entry. I can't tell you how many variations of my name I've seen come through the mailbox over the years (see Figure 1), yet all these distinct "name values" refer to the same individual.

You can see how a database gathering data from multiple sources, or multiple independent transactions may easily generate duplicate records. Frequently lists of this nature are used for mass postal mailings and duplication of names and addresses in the mailing list results in wasted printing costs and postage for the redundant material, not to mention the less than positive impression given to the mailing recipient. Duplicate detection and elimination is a significant selling point for mailing houses that process lists for outside clients.

This month, we'll look at two general algorithms that will help us reduce the degree of duplication within free-form text. These algorithms were specifically intended for name matching, but they really apply to words in general and can be useful in matching addresses, cities, states, and provinces.

Soundex

Soundex is a very old and very familiar phonetic encoding algorithm, invented in 1918 and attributed to Robert C Russell and Margaret O'Dell. Soundex was developed far in advance of computers to help in the manual sorting of census records. The algorithm ignores vowels and groups the remaining consonants phonetically into six groups based on how the sounds are made: bilabial, labiodental, dental, alveolar, velar and glottal.

We produce a Soundex code as follows:

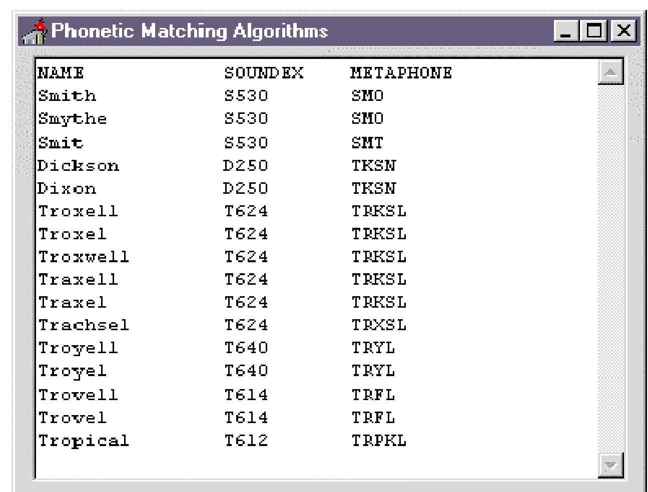
- Convert all letters to uppercase and retain the first letter.
- Drop any occurrence of the letters A, E, I, O, U, H, W, or Y, except in the first letter.
- Convert remaining letters to numbers as follows:

B, F, P, V	1
C, G, J, K, Q, S, X, Z	2
D, T	3
L	4
M, N	5
R	6

- Drop any digit that matches the immediately previous digit.
- Stop once a four character code is produced, or pad with zeroes if necessary to produce a four character code (a letter followed by three digits).

Listing 1 shows a Delphi 3 implementation for Soundex. Figure 1 shows some sample names and their corresponding Soundex values. For this data I took a mixture of names that are phonetically similar, and a sampling of typographical and transcription errors I've seen in my mail over the years.

Soundex is widely used in many name-matching applications. In fact, Soundex functions are built into several RDBM systems such as Microsoft and Sybase SQL Servers, Oracle, and SQL Anywhere. However, some of these vendor implementations contain quirks that in some cases result in different



NAME	SOUNDEX	METAPHONE
Smith	S530	SM0
Smythe	S530	SM0
Smit	S530	SMT
Dickson	D250	TKSN
Dixon	D250	TKSN
Troxell	T624	TRKSL
Troxel	T624	TRKSL
Troxwell	T624	TRKSL
Traxell	T624	TRKSL
Traxel	T624	TRKSL
Trachsel	T624	TRXSL
Troyell	T640	TRYL
Troyel	T640	TRYL
Trovell	T614	TRFL
Trovel	T614	TRFL
Tropical	T612	TRPKL

➤ *Figure 1*

```
function Soundex(aKey: string): string;
const          {ABCDEFGHIJKLMNOPQRSTUVWXYZ}
  LetterCodes = '01230120022455012623010202';
  MaxCodeLength = 4;
var
  I: Integer;
  Ch: Char;
  LastCh: Char;
begin
  Result := '';
  LastCh := #0;
  I := 1;
  while Length(Result) <> MaxCodeLength do begin
    if I > Length(aKey) then
      Result := Result + '0'
    else begin
      Ch := UpCase(aKey[I]);
      if Ch in ['A'..'Z'] then
        if Length(Result) = 0 then
          Result := Ch
        else begin
          Ch := LetterCodes[Ord(Ch) - 64];
          if (Ch <> '0') and (Ch <> LastCh) then begin
            Result := Result + Ch;
            LastCh := Ch;
        end;
      end;
      Inc(I);
    end;
  end;
end;
```

➤ *Listing 1*

codes being generated than the algorithm shown here (see this month's *Surviving SQL*).

One drawback to Soundex is that it is somewhat biased for English language words and may not perform as well for other languages. There are Soundex-like alternatives for the French (Henry Name-Matching) and Slavic/German (Daitch-Mokotoff Coding Method) languages.

One of Soundex's notable shortcomings is that it is limited to producing at most 6,708 unique codes. Even this assumes an even distribution of first letters of names across the alphabet, so in practice the actual number of unique codes will be somewhat less. As the number of varying names to encode increases, the greater the propensity to group truly unrelated names into the same code value. Note that the sheer number of names is not the issue; it's the number of *different* names. Even so, with a database of one million names, a Soundex search for matches to any given name will return 150 matches on average, including duplicates.

Metaphone

A notable attempt to improve the basic Soundex concept came in 1990 from Lawrence Philips. This method is called Metaphone and goes beyond single consonant sounds to consider groups of letters, or dipthongs. Metaphone tends to isolate truly different names a bit better than Soundex because it applies more detailed transformations and has a wider range of possible code values. Like Soundex, Metaphone is inherently tied to English pronunciation.

Metaphone encoding is accomplished as follows:

- ➢ Convert all letters to uppercase and retain the first letter.
- ➢ Drop all vowels, except if the first letter is a vowel.
- ➢ Collapse duplicating letters except for C down to a single letter.
- ➢ Convert remaining letter groups as shown in Table 1.

Listing 2 shows my implementation of the Metaphone algorithm and Figure 1 shows some representative values.

In general, Metaphone produces fewer false positives than Soundex. While the sample data shown in Figure 1 seems to indicate otherwise, consider that Soundex concludes that Sante, Simondi and Sindhi are phonetic matches for Smith. Metaphone assigns unique codes in each of these cases. Notice also from Figure 1 how Metaphone distinguishes the "th" sound in Smith and Smythe from the hard "t" sound in Smit. This may or may not be advantageous depending on whether you'd consider Smit to be a misspelling of Smith.

The standard Metaphone algorithm has a theoretical upper limit of approximately 49 million unique codes. The degree of granularity of the Metaphone algorithm can be influenced by increasing or decreasing the maximum size of the resulting code.

Obviously, Metaphone will be slower to execute since it's more detailed transformations result in bulkier coding. Metaphone typically operates about half as fast as a Soundex algorithm on the same platform.

Conclusion

Both Soundex and Metaphone provide useful methods by which words that are phonetically similar but textually different may be matched with some degree of accuracy. Obivously, there will still be cases of true matches being missed and false matches being picked up, but that's the nature of fuzzy logic. Both of these algorithms are biased towards English language words, but there are two language independent methods you might care to investigate on your own: K-Approximate Matching and Guth Name-Matching.

Next month we'll look at a technique for indexing words within free-form text for building keyword lists for document retrieval systems.

Steve Troxell is a software engineer with Ultimate Software Group in the USA. He can be contacted via email at Steve_Troxell@USGroup.com

References

Practical Algorithms For Programmers, Andrew Binstock and John Rex, 1995, Addison-Wesley Publishing, ISBN 0-201-63208-X, pp157-169.

SQL For Smarties: Advanced SQL Programming, Joe Celko, 1995, Morgan Kaufman Publishers, ISBN 1-55860-323-9, pp83-90.

An Assessment of Name Matching Algorithms, A J Lait and B Randall, University of Newcastle upon Tyne, UK.

Letter	Conversion
F, J, L, M, N, R	Do not change
GN-, KN-, PN-	N
AE-	E
WH-	H
WR-	R
X-	S
B	Dropped if in -MB, otherwise B
C	X if in -CIA- or -CH-, S if in -CI-, -CE- or -CY-, dropped if in -SCI-, SCE- or -SCY-, otherwise K
D	J if in -DGE-, -DGI- or -DGY-, otherwise T
G	Dropped if in -GH-, and not at end or before a vowel, dropped if in -GNED, -GN, -DGE-, -DGI- or -DGY-, J if in -GE-, -GI, -GY and not GG, otherwise K
H	H if before a vowel and not after C, G, P, S, T
K	Dropped if after C,otherwise K
P	F if before H, otherwise P
Q	K
S	X in -SIO- or -SIA-, otherwise S
T	X in -TIA- or -TIO-, O if before H, drop if in -TCH-, otherwise T
V	F
W	W if after a vowel, otherwise dropped
X	KS
Y	Y if after a vowel, otherwise dropped
Z	S

➤ *Table 1: Metaphone letter group conversion*

➤ *Listing 2*

```
function Metaphone(aKey: string): string;
const
  MaxCodeLength = 6;
  VowelSet = ['A', 'E', 'I', 'O', 'U'];
  NonTransformSet = ['F', 'J', 'L', 'M', 'N', 'R'];
  EIYSet = ['E', 'I', 'Y'];
var
  Ch: Char;
  I: Integer;
  KeyBuffer: array[0..256] of Char;
  KeyBufLen: Integer;    { Number of chars in buffer }
  Key: PChar;            { Pointer to start of string }
  LastCharPos: Integer;  { Position of last char in buffer }
begin
  Result := '';
  { Retain uppercase alpha characters in buffer; buffer will
    always have at least one #0 placeholder before and after
    keyword. This avoids need to check length bounds when
    comparing previous or next letters.  }
  FillChar(KeyBuffer, SizeOf(KeyBuffer), #0);
  Key := @KeyBuffer[1];
  KeyBufLen := 0;
  for I := 1 to Length(aKey) do begin
    Ch := UpCase(aKey[I]);
    if Ch in ['A'..'Z'] then begin
      Key[KeyBufLen] := Ch;
      Inc(KeyBufLen);
    end;
  end;
  LastCharPos := KeyBufLen - 1;
  { Transform prefixes }
  if CompareMem(Key, PChar('GN'), 2) or
    CompareMem(Key, PChar('KN'), 2) or
    CompareMem(Key, PChar('PN'), 2) or
    CompareMem(Key, PChar('AE'), 2) or
    CompareMem(Key, PChar('WH'), 2) or
    CompareMem(Key, PChar('WR'), 2) then
    Inc(Key)
  else if Key[0] = 'X' then
    Key[0] := 'S';
  for I := 0 to LastCharPos do begin
    { Skip duplicating letters except for C }
    if (Key[I - 1] = Key[I]) and (Key[I] <> 'C') then
      Continue;
    { Retain nontransform letters }
    if (Key[I] in NonTransformSet) or
      ((I = 0) and (Key[I] in VowelSet)) then begin
      Result := Result + Key[I];
      Continue;
    end;
    { Apply transforms }
    case Key[I] of
      'B': { retain unless within -MB }
           if not ((I = LastCharPos) and
             (Key[I - 1] = 'M')) then
             Result := Result + 'B';
      'C': { drop if in -SCI-, -SCE- or -SCY- }
           if not ((Key[I - 1] = 'S') and
             (Key[I + 1] in EIYSet)) then
             { map to X if in -CIA- or -CH-}
             if ((Key[I + 1] = 'I') and
               (Key[I+2] = 'A')) or (Key[I+1] = 'H') then
               Result := Result + 'X'
             else
               { map to S if in -CE-, -CI- or -CY- }
               if Key[I + 1] in EIYSet then
                 Result := Result + 'S'
               else  { otherwise K }
                 Result := Result + 'K';
      'D': { map to J if in -DGE-, -DGI- or -DGY- }
           if (Key[I + 1] = 'G') and
             (Key[I + 2] in EIYSet) then
             Result := Result + 'J'
           else  { otherwise T }
             Result := Result + 'T';
      'G': { map to J if in -GE-, -GI, -GY and not GG }
           if (Key[I + 1] in EIYSet) and
           (Key[I - 1] <> 'G') then Result := Result + 'J'
           else
             { drop if in -GH- but not at end or before a vowel }
             if not ((Key[I + 1] = 'H') and
               (I <> LastCharPos - 1) and
               not (Key[I + 2] in VowelSet)) or
               { drop if in -GNED }
               ((I = LastCharPos - 3) and
               CompareMem(@Key[I + 1], PChar('NED'), 3)) or
               { drop if in -GN }
               ((I = LastCHarPos - 1) and
               (Key[I + 1] = 'N')) or
               { drop if in -DGE-, -DGI- or -DGY- }
               ((Key[I - 1] = 'D') and
               (Key[I + 1] in EIYSet)) then
               { otherwise K }
               Result := Result + 'K';
      'H': { retain if before a vowel and not after C, G, P, S or T }
           if (Key[I + 1] in VowelSet) and
             not (Key[I-1] in ['C','G','P','S','T']) then
             Result := Result + 'H';
      'K': { retain unless after C }
           if Key[I - 1] <> 'C' then
             Result := Result + 'K';
      'P': { map to F if before H }
           if Key[I + 1] = 'H' then
             Result := Result + 'F'
           else  { otherwise P }
             Result := Result + 'P';
      'Q': { map to K }
           Result := Result + 'K';
      'S': { map to X if in -SH-, -SIO- or -SIA- }
           if (Key[I + 1] = 'H') or
              ((Key[I + 1] = 'I') and
             (Key[I + 2] in ['O', 'A'])) then
             Result := Result + 'X'
           else  { otherwise S }
             Result := Result + 'S';
      'T': { map to X if in -TIA- or -TIO- }
           if (Key[I + 1] = 'I') and
             (Key[I + 2] in ['O', 'A']) then
             Result := Result + 'X'
           else
             { map to 0 (zero) if before H }
             if Key[I + 1] = 'H' then
               Result := Result + '0'
             else
               { drop if in -TCH- }
               if not ((Key[I + 1] = 'C') and
                 (Key[I + 2] = 'H')) then
                 { otherwise T }
                 Result := Result + 'T';
      'V': Result := Result + 'F'; { map to F }
   'W','Y': { retain if after a vowel }
           if Key[I - 1] in VowelSet then
             Result := Result + Key[I];
      'X': Result := Result + 'KS'; { map to KS }
      'Z': Result := Result + 'S'; { map to S }
    end;
    { terminate if max code length is reached }
    if Length(Result) = MaxCodeLength then Break;
  end;
end;
```

TDataSet Update

In the September and October issues, we developed a customize `TDataSet` descendant component to allow us to access a proprietary database format. I was fortunate enough to have Borland's Mark Edington, the engineer responsible for the `TDataSet` abstraction in Delphi 3, take a look at those articles. Mark pointed out a couple of flaws in my implementation.

First, the `CompareBookmarks` method should return `-1` if Bookmark1 refers to a location before Bookmark2 in the table (contrary to what the online help says). This behavior is needed by `TDBGrid` when multiselect is enabled. Listing 3 shows Mark's corrected method.

Second, in the `GetRecord` method, a problem exists when the last physical record of the file has been deleted. In my original code, an infinite loop would occur as the dataset tried to call for the prior record from the end of the file. Listing 4 shows the corrected code for the `gmPrior` case within the `GetRecord` method.

```
function TMyDataSet.CompareBookmarks(
  Bookmark1, Bookmark2: TBookmark): Integer;
const
  RetCodes: array[Boolean, Boolean]
    of Integer = ((2, -1), (1, 0));
begin
  Result := RetCodes[Bookmark1 = nil, Bookmark2 = nil];
  if Result = 2 then begin
    if TBookmarkInfo(Bookmark1^) =
      TBookmarkInfo(Bookmark2^) then
      Result := 0
    else if TBookmarkInfo(Bookmark1^) >
      TBookmarkInfo(Bookmark2^) then
      Result := 1
    else
      Result := -1;
  end;
end;
```

➤ *Listing 3*

```
gmPrior:
  begin
    AtEof := Eof;
    repeat
      FilePosition := FilePos(FInternalFile);
      if FilePosition < (2 * FRecSize) then
        Result := grBOF
      else begin
        if AtEof then
          Seek(FInternalFile,
            FileSize(FInternalFile) - FRecSize)
        else
          Seek(FInternalFile,
            FilePosition - (2 * FRecSize));
        BlockRead(FInternalFile, Buffer^, FRecSize);
        AtEof := False;
      end;
    until (Result <> grOk) or (Byte(Buffer^) = 0);
  end;
```

➤ *Listing 4*