Case-Sensitive Operations in 4th Dimension

By David Adams

TN 05-41

Overview

4th Dimension のプログラム言語およびデータベースエンジンは、基本的にアルファベットの大小文字を区別しないで動作するように設計されています。大文字と小文字、それにアクセント記号の付いた文字は、すべて同格として扱われます。たとえば、次に挙げた文字は、すべて同じものとしてみなされます。

e　é　E　É　Ë

文字の種類を区別しない 4th Dimension の動作は、多くの場合、都合が良いものですが、データを厳密に比較したい場合にはかえって厄介です。アルファベットを区別することが望ましいシチュエーションには次のようなものが挙げられます。

- 品名などのデータを比較し、特定の規則を守らなくてはならない場合。
- 大文字と小文字を区別するボリュームに保存されたドキュメントのパスを扱う場合。
- XML の要素や属性の名前を扱う場合。
-

注記　テキストデータを比較した結果は、4th Dimension のバージョン、オペレーティングシステム、スクリプトシステム、またデータベースプロパティで選択されたスクリプトマネージャの設定に依存しています。いずれにしても、大文字と小文字を区別するネイティブの機能はありません。

このテクニカルノートは、大文字と小文字を比較するための方法を幾つか紹介し、ケースセンシティブなサーチを最適化する方法について検討します。サンプルデータベースには、次のようなタスクを実行するためのコードが含まれています。

- 文字列/テキストタイプの値を比較する。
- 文字列/テキストタイプの配列を比較する。
- 文字列/テキストタイプの配列の中から値を探す。
- 文字列/テキストタイプの配列の中にある値を数える。
- BLOB を正確に比較する。
- QUERY BY FORMULA やストアドハッシュに代わるものとして、最適化されたケースセンシティブサーチを実行する。

Related Technical Notes

このテクニカルノートと付属するサンプルデータベースは、一連のテクニカルノートシリーズの一部です。 関連するトピックには次のようなものが含まれます。

- テクニカルノート 05-42 Scanning Text and BLOBS Efficiently では、もっとも高速で、メモリ使用が効率的なテキストおよび BLOB の解析方法を論じています。
- テクニカルノート 05-43 The HashTools Component では、このテクニカルノートのサンプルデータベースで使用しているサーチ部分で使用しているハッシュルーチンを解説しています。
- テクニカルノート 05-44 Optimizing Text and BLOB Searches with Hashes では、ハッシュツールコンポーネントのソースコードを解説し、テキストおよび BLOB のサーチを最適化する上でハッシュがどのように役立つかを説明しています。異なるハッシュアルゴリズムによるサーチ結果の違いを示すテストデータも提供しています。

4th Dimension における大文字と小文字の処理について取り上げる前に、これが特に重要である理由のひとつを考慮しましょう。それは XML の処理と関係があります。

Pay Special Attention to XML Names

XML がすっかり普及した現在、4th Dimension のプログラマも XML を生成したり、解析したりするように求められることが多くなっています。実際、Web Services は必ず XML で記述する必要があります。そのような状況において、文字種を無視する 4th Dimension の動作は障害となりかねません。XML において次の点はまったく譲歩の余地がない問題です。つまり、XML の要素名を扱う場合は、厳密に大文字と小文字を区別する必要があります。たとえば、次の要素名は 4th Dimension では同じものを指しますが、XML では別のものです。

customer

Customer

CUSTOMER

次のテキストは最初のタグとそれを閉じるタグが違うため、不正な XML となります。

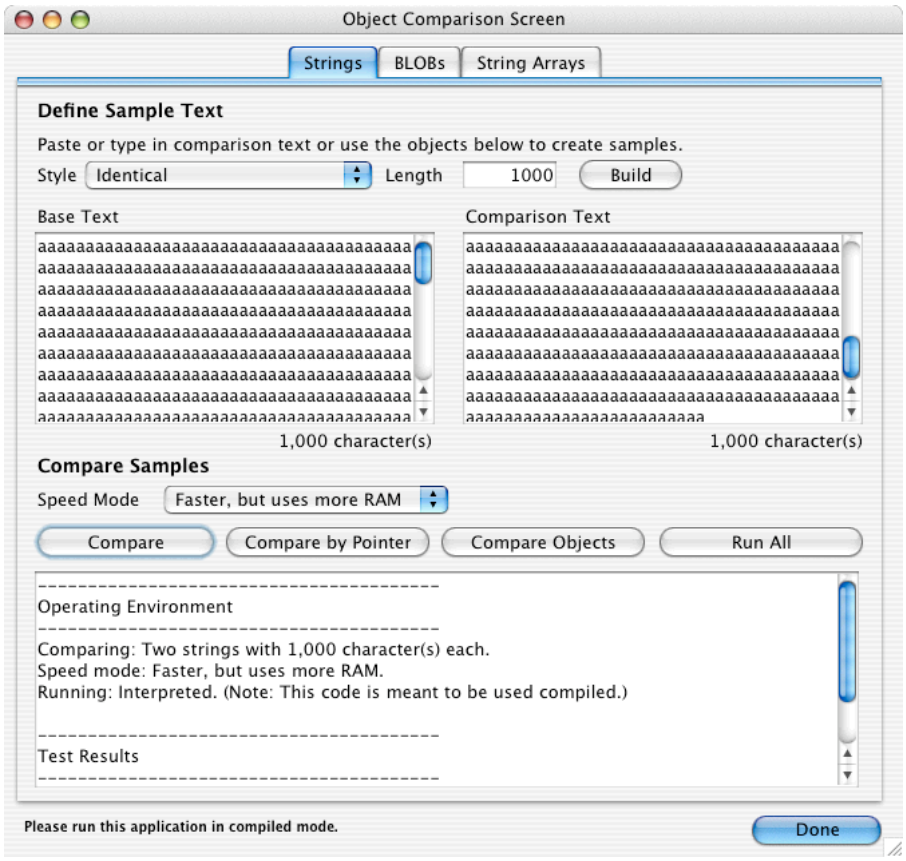
```
<?xml version="1.0" encoding="UTF-8"?>
<customer>
</Customer>
```

実際の場面では、あえて大文字と小文字を混ぜてタグを記述する人は少ないかもしれませんが、そうしてはならない理由はありません。文字の大小を考慮しないコードは、動いているように見えても潜在的なバグを抱えています。物事を運命に任せるのは責任あるプログラマの仕事とはいえません。むしろ、大文字と小文字をきちんと区別するコード処理をするべきです。この

テクニカルノートの解説とサンプルコードは、プロジェクトにケースセンシティビティを付与する上で助けになります。はじめに 4th Dimension のプログラム言語の動作を確認し、次にケースセンシティブサーチの方法について取り上げます。

String, BLOB, and Array Comparison Test Screen

サンプルデータベースには、文字列、BLOB、文字列配列を使用して比較実験をすることのできるインタフェースが用意されています。



デモンストレーションを使用する際には、次の点に留意してください。

- コードに興味があるのであれば、トレースを実行し、それぞれのルーチンの動作を確かめることができます。
- パフォーマンスに興味があるのであれば、コンパイルを実行してください。
- "speed mode"の選択肢では、一方の方法を「高速」と呼んでいますが、サンプルデータのサイズによっては、名前と速度の関係が逆転することもあり得ます。実際、ある程度大きなデータでなければ、高速モードは効果を発揮しません。次は、サンプルデータベースの中で何度も使用されているこのモードについて説明します。

Speed Mode and Performance Considerations

ポインタ渡しである以下のメソッドには、それぞれ"speed mode"というオプションのパラメータを使用することができます。メソッドには必ずポインタを渡しますが、それを内部的に処理する方法は speed mode の設定に依存しています。以下おの表は動作を違いを説明したものです。

コード	モード名	テキスト	BLOB
0	デフォルト	値をコピー	ポインタで読み込み
1	Faster but uses RAM	値をコピー	値をコピー
2	Slower but saves RAM	ポインタでスキャン	ポインタでスキャン

内部的には、値をコピーする場合、ポインタの先にあるテキストまたは BLOB をローカル変数にコピーします。スキャンのコードはポインタの先にある変数ではなく、このローカル変数を使用します。値をコピーする方法は RAM を消費しますが高速です。BLOB を例にあげると、ポインタを介してスキャンする方法はBLOBを直接スキャンするよりも1.67倍時間がかかります。無論、テストの方法によって結果は多少、違いますが、大体そのような比率になるはずです。テキストの場合、結果はテキストのサイズによって変わってきます。4th Dimension のバージョンによっては､ポインタを介してテキストをスキャンする際に要する時間は､テキスト 15000 バイトを超えたあたりで劇的に変化し、場合によっては 100 倍ほどの速度減退が起こります。この点を考慮に入れるなら、テキストはポインタを介してスキャンするよりもローカル変数にコピーしたほうが無難だといえます。4th Dimension のテキストは最大でも 32000 バイトなので、コピーによってメモリが致命的に不足することはないはずです。ここで紹介した情報はテクニカルノート 05-42 Scanning Text and BLOBs Efficiently に含まれています。

注記　4th Dimension のポインタは、C 言語などのポインタとは内部的な仕組みが異なっており、値の参照にかなりのリソースを要します。

Comparing Strings and Text

Compare Character Values, Not Characters

既に論じてきたように、4th Dimension はアルファベット文字の大小やアクセント記号の有無を度外視して文字列の比較をします。たとえば、次の式は True を返します。

"a"="A"

この式の書き換えは比較的簡単で、文字そのものを比較する代わりに、それぞれのキャラクターのアスキーコードを比較するように直せば済みます。たとえば、次の式は False を返します。

Ascii("a")=Ascii("A")

Sample Implementation

基本的にいって、ケースセンシティブな処理に関係するのは、上記のサンプルの原理だけです。文字列やテキストを比較する場合、長さが同じことを確かめ、ループ処理の中で各文字のアスキーコードの値を比較します。サンプルデータベースで使用されている CS_AlphasAreEqual ルーチンの一部である下記のコードは、その点を例証しています。

```
C_LONGINT($base_length)
C_LONGINT($index)
C_BOOLEAN($continue)
$base_length:=Length($baseString_t)
$index:=1  ` Strings are not empty, based on a test performed before this code fragment.
C_LONGINT($base_ascii)
C_LONGINT($comparison_ascii)
$continue:=True
$stringsAreEqual_b:=True
While ($continue)
	$base_ascii:=Ascii($baseString_t[[$index]])
	$comparison_ascii:=Ascii($comparisonString_t[[$index]])
	If ($base_ascii=$comparison_ascii)  ` Equal: continue examination
	 $index:=$index+1  ` Set counter for next character in string
	 If ($index>$base_length)  ` The counter is now larger than the string, therefore we should stop.
	  $continue:=False
	 End if
	Else  ` Not equal: stop examination
	 $stringsAreEqual_b:=False
	 $continue:=False
	End if
End while
```

String Comparison Routines Provided

サンプルデータベースには、利便性を考慮してふたつの文字列比較ルーチンが用意されています。動作結果は同じですが、一方はポインタで文字列/テキストデータを受け取るのに対して、他方は変数を直接パラメータとして受け取ります。両者の違いについては、“Speed Mode and Performance Considerations”の項を参照してください。

CS_AlphasAreEqual

CS_AlphasAreEqual (Text;Text) ->Boolean

CS_AlphasAreEqual (Base text;Comparison text)

このルーチンは、パラメータとしてふたつの文字列/テキストの値を受け取り、厳密に比較します。値が同一であれば、True が返され、そうでなければ False が返されます。次のコードは、トリガで *CS_AlphasAreEqual* を使用して、顧客の名前が変更されているかを厳密に確かめている例です。

```
If (CS_AlphasAreEqual([Customer]Name;Old([Customer]Name))
        ` The name has changed
Else
        ` The name has not changed.
End if
```

CS_AlphasAreEqual_Pointer

CS_AlphasAreEqual_Pointer (Pointer; Pointer;Longint) ->Boolean

CS_AlphasAreEqual_Pointer (->Base text;->Comparison text;{Speed mode})

このルーチンは、パラメータとしてふたつの文字列/テキストの値をポインタで受け取り、厳密に比較します。値が同一であれば、True が返され、そうでなければ False が返されます。次のコードは、トリガで *CS_AlphasAreEqual_Pointer* を使用して、顧客の名前が変更されているかを厳密に確かめている例です。

```
C_TEXT($oldName_text)
$oldName_text:=Old([Customer]Name)
If (CS_AlphasAreEqual_Pointer(->[Customer]Name;->$oldName_text)
        ` The name has changed
Else
        ` The name has not changed.
End if
```

デフォルトのスピードモードは 1 つまり" copy values "です。

Comparing BLOBs

Why Compare BLOBs

4th Dimension のテキストフィールドには最大 32KB のテキストを収めることができますが、実際のアプリケーションでは 32KB 以上のテキストデータを保存する必要が生じることがあります。4th Dimension の BLOB フィールドには巨大なデータブロックを収めることができる一方、テキストほど簡単には検索や比較ができません。4th Dimension の等号(=)演算子を BLOB に対して使用することはできません。

このテクニカルノートとサンプルデータベースは、BLOB を比較したり、値を探す際の助けになるコードが含まれています。前述の文字列を比較するコードも、ループでテキストの各文字のアスキーコードを比較していましたが、同じ原理を BLOB に対しても適用することができます。実際、BLOB の場合は、バイトの値がすでに数値であるため、Ascii 関数を呼び出す必要がなく、テキストのスキャンよりも早く処理が完了します。

Sample Implementation

CS_BlobsAreEqual ルーチンの一部である次のコードは、ふたつの BLOB を比較する方法を例証しています。

```
C_LONGINT($index)
C_BOOLEAN($continue)
$index:=0    ` Bytes in a BLOB are numbered from 0, not 1. (Strings are numbered from 1. )
$continue:=True
$blobsAreEqual_b:=True
While ($continue)
        If ($1{$index}=$2{$index})    ` Compare bytes directly.
         $index:=$index+1    ` Set counter for next character in string
         If ($index=$base_length)    ` The counter is now larger than the string so stop.
           $continue:=False
              ` Notice that the test is = $base_length because BLOBs are numbered from 0.
              ` So, if we had a BLOB of 1 byte, the first time through the While loop, the counter
              ` is set to 0 before the comparison and 1 after the comparison. 1 = the second physical
              ` byte. There isn't a 2nd byte, so the loop should finish.
         End if
        Else    ` Bytes are not equal.
         $blobsAreEqual_b:=False
         $continue:=False
        End if
End while
```

注記 コードの中にあるコメント文でも注意を喚起しているように、BLOB は先頭からのオフセットでバイトの位置を指定します。したがって最初のバイトは 0 番であり、最後のバイトは BLOB size-1 番です。

BLOB Comparison Routines Provided

サンプルデータベースには、利便性を考慮してふたつのBLOB比較ルーチンが用意されています。動作結果は同じですが、一方はポインタでBLOBを受け取るのに対して、他方は変数を直接パラメータとして受け取ります。両者の違いについては、"Speed Mode and Performance Considerations"の項を参照してください。

CS_BlobsAreEqual

CS_BlobsAreEqual (BLOB; BLOB) ->Boolean

CS_BlobsAreEqual (Base BLOB;Comparison BLOB)

このルーチンは、パラメータとしてふたつのBLOBを受け取り、厳密に比較します。値が同一であれば、Trueが返され、そうでなければFalseが返されます。次のコードは受け取ったBLOBの内容が、ロードされたレコードのBLOBと一致するかを調べています。

```
If (CS_BlobsAreEqual(IncomingData_blob;[Sample_Data]BLOB)
    ` The incoming value is a duplicate of the BLOB stored in this record.
Else
    ` The incoming value is different from the BLOB stored in this record.
End if
```

CS_BlobsAreEqual_Pointer

CS_BlobsAreEqual_Pointer (Pointer; Pointer;Longint) ->Boolean

CS_BlobsAreEqual_Pointer (->Base BLOB;->Comparison BLOB;{Speed mode})

このルーチンは、パラメータとしてふたつの BLOB をポインタで受け取り、厳密に比較します。値が同一であれば、True が返され、そうでなければ False が返されます。次のコードは受け取った BLOB の内容が、ロードされたレコードの BLOB フィールドと一致するかを調べています。

```
If (CS_BlobsAreEqual_Pointer(->IncomingData_blob;->[Sample_Data]BLOB)
        ` The incoming value is a duplicate of the BLOB stored in this record.
Else
        ` The incoming value is different from the BLOB stored in this record.
End if
```

デフォルトのスピードモードは 2 つまり"scan through pointer"です。

Comparing String/Text Arrays

サンプルデータベースには*CS_AlphaArraysAreEqual_Pointer*という文字列/テキスト配列を比較するルーチンが含まれています。配列のサイズが同一で、アスキーデータレベルでも一致する場合、trueが返され、そうでなければFalseが返されます。ルーチンのパラメータは次のとおりです。

CS_ArraysAreEqual_Pointer (Pointer; Pointer;Longint) ->Boolean
CS_ArraysAreEqual_Pointer (->Base array;->Comparison array;{Speed mode})
デフォルトのスピードモードは 1 つまり"copy values"です。

General Comparison Routine

サンプルデータベースには*CS_ContentsAreEqual_Pointer*という汎用的な比較ルーチンが含まれています。内部的には渡されたパラメータを判別し、*CS_AlphaArraysAreEqual_Pointer*、*CS_AlphasAreEqual_Pointer*、または*CS_BlobsAreEqual_Pointer*のいずれかに処理を渡すようになっています。ルーチンのパラメータは次のとおりです。

CS_ContentsAreEqual_Pointer (Pointer; Pointer;Longint) ->Boolean
CS_ContentsAreEqual_Pointer (->Base value;->Comparison value;{Speed mode})
デフォルトのスピードモードは渡されたオブジェクトの種類によって決まります。

タイプ	コード	動作
文字列/テキスト配列	1	値をコピー
文字列/テキスト	1	値をコピー
BLOB	2	ポインタでスキャン

Case-Sensitive String/Text Array Utilities

サンプルデータベースには、大文字と小文字を区別するバージョンのFind in arrayおよびCount in array関数が含まれています。

注記 サンプルのデモンストレーションインタフェースには表示されていません。

CS_CountInAlphaArray_Pointer

CS_CountInAlphaArray_Pointer (Pointer; Pointer;Longint) ->Longint

CS_CountInAlphaArray_Pointer (->String/text array;->String/text value;{Starting element})

*CS_CountInAlphaArray_Pointer*メソッドは、4th DimensionのCount in array関数にケースセンシビリティを付加したものです。デフォルトで、配列のすべての要素が評価の対象になります。任意の第3パラメータを渡すことにより、開始する要素の位置を指定できます。

CS_FindInAlphaArray_Pointer

CS_FindInAlphaArray_Pointer (Pointer; Pointer;Longint) ->Longint

CS_FindInAlphaArray_Pointer (->Array;->Text value;{Starting element})

*CS_FindInAlphaArray_Pointer*メソッドは、4th DimensionのFind in array関数ににケースセンシビリティを付加したものです。デフォルトで、配列のすべての要素が評価の対象になります。任意の第3パラメータを渡すことにより、開始する要素の位置を指定できます。

Case-Sensitive Searching

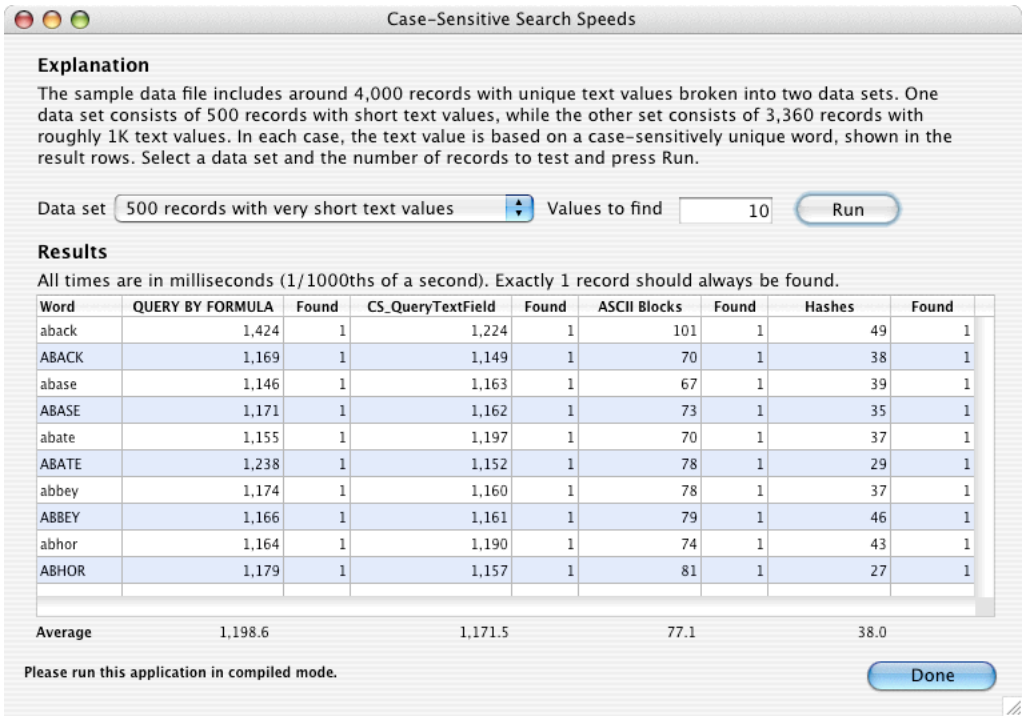
ここまでは値の比較をする方法について論じてきました。多くのデベロッパが鋭い関心を抱いているのは、ケースセンシティブな検索の方法です。たとえば、データベースでXMLを扱ったり、大文字と小文字を区別するボリュームのパスを扱うことがあるかもしれません。そのような場合、4th DimensionのQUERYコマンドは動作に問題があります。標準のQUERYコマンドは、演算子の=を使用しても完全一致の比較をするわけではないからです。さらにドキュメント化されていない動作として、テキストフィールドの場合は、最初の80バイトだけが比較の対象になるようです。したがってこの先の内容は、大文字と小文字の比較だけではなく、長いテキストの扱いに苦慮している人にとっても参考になる情報です、サンプルデータベースのテストインタフェースを概観した後、様々な解決方法について考察します。

- QUERY BY FORMULAを使用する。
- QUERY BY FORMULAを最適化した*CS_QueryTextField*メソッドを使用する。
- アスキーブロックに変換されたテキストを使用して検索する。
- テキストのハッシュを使用して検索する。

パフォーマンスに優れているのは、特に4D Serverを使用する場合、ハッシュを使用する方法ですが、その他のアプローチにもそれぞれ適切な用途があります。

Search Speed Test Screen

サンプルデータベースには、それぞれのアプローチについて、テスト用の画面とデータが用意されています。



テスト画面では、テストデータセットと検索する値の数を指定することができます。テスト結果にはある程度の変動があるので、数回のテストを実行してください。テストを実行すると、検索方法ごとにミリ秒で表わした所要時間とヒットしたレコードの数が表示されます。テスト用のデータでは、レコード数は必ず1件になるはずです。サンプルデータは意図的にそのようなものになっています。パフォーマンスを調べたいのであれば、コンパイルモードで、4D Serverで運用を検討中であれば4D Serverでテストを実行してください。

4種類の検索方法を比較検討した後、4th Dimensionと4D Serverにおける動作の違いについても論じますが、アプリケーションタイプによってかなりの差があることに注目できます。

QUERY BY FORMULA

4th Dimensionで大文字と小文字を比較して検索したい場合、もっとも簡単なのはQUERY BY FORMULAコマンドを使用する方法です。

```
QUERY BY FORMULA([Sample_Data]; CS_AlphasAreEqual ([Sample_Data]Text;$valueToMatch_text))
```

上記のコードは正しく動作しますが、残念なことにQUERY BY FORMULAは非常に遅いという難点があります。4D Serverでは特にそうです。データの種類のよっては、他の方法と比べて数十倍ないし数百倍も時間がかかります。他の代替案を簡単なものから順にみてゆきましょう。

Replacing QUERY BY FORMULA with Custom Code

上記のQUERY BY FORMULAコードは、単純に*CS_AlphasAreEqual*でテーブルの各レコードを比較しているに過ぎません。サンプルデータベースには、これを最適化したメソッドである*QueryTextField*が含まれています。

```
CS_QueryTextField (->[Sample_Data]Text;$valueToMatch_text;)
```

メソッドの内部的な動作は要約すると次のようなものになります。

結果のレコード番号を受け取るために空の数値配列を作成
元のレコード番号を配列に読み込む
比較するテキストの値を配列に読み込む

```
For(テキスト配列の各要素の値につき)
   If(要素の値=検索する値)
      レコード番号を結果の配列に追加
   End if
End for
結果の配列からセレクションを作成
```

QUERY BY FORMULAと同じ処理をしているにも関わらず、実行時間には大きな差があります。データの種類によりますが、4th DimensionではQUERY BY FORMULAよりもかなり速く、4D Serverでは単純なテストで55-81倍の速度差があります。

注記 CS_AlphasAreEqualは、メモリの使用量を抑えるために手を加えられています。すべてのレコードを配列に読み込む代わりに、256レコードずつ読み込んでいます。この値は必要に応じて調整することができるでしょう。

Storing Additional Data to Optimize Searches

QUERY BY FORMULA および CS_QueryTextField の利点は、導入が簡単で、あらゆるテキストフィールドに使用できることです。さらなるパフォーマンスを求めるのであれば、補足的にデータを別の形式で保存するという方法もあります。作成するデータは、インデックス QUERY が実行でき、なおかつ大文字と小文字を区別するようなものであれば、間でも構いません。ここでは、アスキーブロックを使用する方法とハッシュを使用する方法について考察します。

Working with ASCII Blocks

データの形式を変換すれば、文字の種類を区別しない4th Dimensionの動作を回避することができます。アスキーブロックはそのための単寿な方法です。*Demo_ConvertTextToAsciiBlock* メソッドは、テキストの固まりをアスキーブロックに変換するコードです。たとえば、abcというデータは、067098099という数字の文字列に変換します。4th DimensionはabcとABCの違いは判別しませんが、097098099と065066067であれば区別するというわけです。配列であれば、アスキーブロックとFind in arrayを組み合わせることによって、問題なく大文字と小文字を区別することができます。しかし、通常のデータについては幾つかの難点があります。

- アスキーブロックは元のテキストよりも 3 倍長い。この点は、16 進数や BASE64 などを使用することによって幾らか緩和できる。
- 4D Server の場合、CS_QueryTextField 以上のスピードアップは得られない。
- テキストタイプのアスキーブロックはインデックス検索ができない。
- 元のデータが更新されるたびにアスキーブロックも更新する必要がある。
- レコードデータサイズが、事実上、4 倍になる。元のテキストが 10000 バイトを超えるようであれば、BLOB を使用しなくてはならない。

こうした難点があるとはいえ、アスキーブロックによる方法は、形式を変換するというアプローチの分かりやすい例です。たとえば、XML の要素名が 1000 件登録されたデータベースがあり、受信した XML 文書や SOAP リクエストがそのリストによって検証されるとします。スタートアップでアスキーブロック化した配列を用意しておけば、FInd in array を使用して受け付けた XML の要素名を検証することができます。この場合、カスタムサーチコードは不要であり、インタプリタモードであっても非常に速く処理することができます。

Using Stored Hashes

大文字と小文字を区別して検索を実行する方法として、非常に優れているのが、ストアドハッシュによる方法です。ハッシュとは、文字列、BLOB、ピクチャ、あるいはドキュメントなどのブロックデータを加工して単一の倍長整数にするテクニックのことです、ハッシュは、署名、またはルックアップとして使用されます。4th Dimension の場合、検索を最適化するためにこれを応用することができます。テーブルまたはセレクション中のすべてのレコードに対してシーケンシャルな比較を実行する代わりに、ストアドハッシュに対してインデックス検索を実行すれば、条件に適合しないレコードをほとんどあるいはすべて除外することができます。たとえば、1KB のユニークな URL を 100000 件登録したテーブルがある場合、QUERY BY FORMULA または CS_QueryTextField を使用するとすべてのレコードを順番に評価しなくてはならないので、非常に時間がかかります。ストアドハッシュを使用すれば、この行程を次のように最適化することができます。

検索する値をハッシュする。
インデックス属性のストアドハッシュフィールドを検索する。
結果セレクションに対してシーケンシャルな検索を実行する。

最後にはシーケンシャルな検索を実行するとはいえ、この時点でレコード数は相当絞り込まれ

ています。ハッシュの仕組みが優れていれば、レコードはひとつ、あるいは数件程度になります。サンプルデータベースに含まれている HashTools コンポーネントには、ハッシュアルゴリズムが幾つも含まれていますが、その多くは 30000 語のデータセットに対して 30000 のユニークなハッシュを作成することができます。これは 100000 件のテキストレコードがある場合、99999 件近くのレコードを最初に除外できることを意味します。

とはいえストアドハッシュにも次のような制限があります。

- 元の値が更新されるたびに、ストアドハッシュを再構築する必要があります。これには相当な処理時間を要しますが、サンプルのハッシュはチップの設計を考慮に入れており、コンパイルすれば非常に高速です。
- ハッシュを保存するために余分なデータ領域が必要です。ただし、ハッシュは倍長整数なので、ほとんどの場合、大きな問題になりません。
- 検索を実行する前に、その値をハッシュしなくてはなりません。この処理に要する時間は、ハッシュの方法と処理する値のデータサイズによって決まります。
- ハッシュのインデックス検索に続いてシーケンシャルな検索を実行します。この動作は、場合によっては割に合わないものです。この件については後述します。

サンプルデータベースでは、[Sample_Data]のトリガで次のように HashTools_HashText メソッドをコールして[Sample_Data]Text のハッシュを管理しています。

[Sample_Data]AP_Hash_of_Text:=*HashTools_HashText* (->[Sample_Data]Text;"AP")

*HashTools_FindByHash*ルーチンについての詳しい説明は、テクニカルノート05-43 The HashTools Componentで取り上げています。コンポーネントのデザインおよび内部構造については、テクニカルノート05-44 Optimizing Text and BLOB Searches with Hashesを参照してください。

Discussion of Sample Test Results

サンプルデータベースには、2種類のサンプルデータが入っています。

- 3,360 records with roughly 1KB of text
- 500 records with very short text values

2 種類のサンプルデータは、テキストフィールドのデータサイズ(80 バイト/1024 バイト)と、レコード数(500 件/3360 件)が異なります。マシンやネットワーク環境によっては、検索方法ごとの長所と短所については、謝った結論を導きだすような結果がでるので、2 種類のサンプルデータを用意しました。以下はコンパイルモードの 4th Dimension におけるテスト結果です。

検索方法	500 件の短いデータ	3360 件の長いデータ
QUERY BY FORMULA	18.0	146.0
CS_QueryTextField	13.0	159.0
アスキーブロック	8.0	161.0
ハッシュ	0.3	1.1

単位：ミリ秒

小さなデータでは、方法による違いがはっきりしており、ハッシュは QUERY BY FORMULA の 60 倍、CS_QueryTextField の 43 倍速いという結果です。これが多数の大きなデータになると、ハッシュ以外の方法による違いはあまり顕著ではなくなります。次に 4D Server における同じテストの結果をみてみましょう。

検索方法	500 件の短いデータ	3360 件の長いデータ
QUERY BY FORMULA	1495.0	43354.0
CS_QueryTextField	27.0	535.0
アスキーブロック	36.0	1344.0
ハッシュ	110.3	112.0

単位：ミリ秒

4D とはかなり違った結果になることが分かります。小さなデータでは、ハッシュはアスキーブロックや CS_QueryTextField よりも時間がかかっています。アスキーブロックによる方法と CS_QueryTextField では､比較のために 500 件のテキストデータを 4D Client に転送します。ハッシュでは、最後の検索のために 1 件のテキストデータだけを 4D Client に転送します。このテスト結果から分かるのは、4D Server 側でインデックスサーチを実行するには、時間がかかるということです。小さなデータが 500 件程度であれば、その時間のほうがかえって全体のパフォーマンスを低下させますが、多数の大きなデータであれば、割に合うようになります。このテストでは、結果レコードが共に 1 件なので、ハッシュによる検索の結果は多数の長いデータでもほとんど変わりませんが、他の方法では、顕著な差がでています。

注記 ここに挙げたデータは、パフォーマンスの相対関係を示すためのものであり、計測時間そのものは重要ではありません。マシンを変えたり、テストを複数回実行するだけでも、数値は変動するものです。

Summary

このテクニカルノートでは、大文字と小文字の区別について、プログラム言語とデータベースエンジンの両面から問題を論じました。テストの結果から、4th Dimension における大文字と小文字の区別について、次のように要約することができます。

- 4th Dimension と 4D Server では、結果が大きく異なることがある。したがって、テストは運用環境で実施するべきである。
- 単純なテストデータでは表面化しない問題もあり、間違った結論に至りかねない。
- QUERY BY FORMULA は、いつでも遅い。CS_QueryTextField またはストアドハッシュのほうが優れている。
- ストアドハッシュによる方法は、データ量が極端に少ない場合を除けば、最良の結果を得ることができ、データサイズに比例して効率が良くなる。長いテキスト、ピクチャ、BLOB、ドキュメントを扱うのであれば、ハッシュは効果的である。その他の手法も、それなりに効果的である。

このテクニカルノートでは、4th Dimension プログラム言語の範囲内で、文字列/テキストフィールド、文字列/テキスト配列、BLOB を含むデータのケースセンシティブな評価に使用できるユーティリティルーチンを紹介しました。必要に応じて、データベースに組み込むことができます。特に SOAP メッセージや XML ドキュメントを扱っている場合、大文字と小文字を含むテキストの比較は要注意です。XML の要素や属性の名前は、ケースセンシティブなものとして扱う必要がありますが、提供されている資料を問題の解決に役立てることができます。