Case-Sensitive Operations in 4th Dimension

By David Adams

Technical Note 05-41

Overview

The 4th Dimension language and database engine don't automatically compare values case-sensitively. Instead, upper- and lower-case versions of a letter, and many of their diacritically marked variations, are considered equal. For example, the **QUERY** command treats the following characters as identical:

e é E É Ë

It is often helpful that 4th Dimension doesn't distinguish between case and diacritically distinct characters. This behavior is an obstacle, however, when you need to compare values case-sensitively. There are several situations when case-sensitivity is desirable or necessary:

- Testing values, such as names, where specific case rules must be observed.
- Searching for stored paths that refer to documents on a case-sensitive volume.
- Comparing and testing XML element and attribute names.

Note *The exact behavior when comparing text values depends on the version of 4th Dimension, the operating system in use, the script system in use, and the Script Manager database properties selected. Regardless, case-sensitive comparisons are not a native feature.*

This note explains how to compare text case-sensitively and examines several strategies for optimizing case-sensitive searches. The sample database includes code to perform the following tasks:

- Comparing alpha/text values case-sensitively.
- Comparing alpha/text arrays case-sensitively.
- Finding values in string/text arrays case-sensitively.
- Counting values in string/text arrays case-sensitively.
- Comparing BLOBs exactly.
- Accelerating case-sensitive text searches with an optimized replacement for **QUERY BY FORMULA** or stored hashes.

Related Technical Notes

This technical note and the accompanying sample database are part of a collection of related notes. The following may also be of interest:

- Technical Note 05-42 **Scanning Text and BLOBS Efficiently** explores in detail how to design 4th Dimension code to analyze text and BLOB values most efficiently for best speed and best memory conservation.
- Technical Note 05-43 **The HashTools Component** documents the hashing routines employed by the searching portions of the sample database included with the note you are reading.
- Technical Note 05-44 **Optimizing Text and BLOB Searches with Hashes** describes the source code of the HashTools component and how hashing can optimize text and BLOB queries. Detailed test results are provided to compare the performance characteristics of different hashing algorithms.

Before examining case-sensitivity in 4th Dimension in detail, let's revisit one of the reasons why it is important: XML.

Recommendation: Pay Special Attention to XML Names

Now that XML is ubiquitous, 4th Dimension programmers are increasingly called upon to create and consume XML. Web Service messages, for example, are always written in XML. In these cases, 4th Dimension's native "case-blind" behavior is a problem. The XML standards are 100% clear on this point: **XML element and attribute names are always case-sensitive**. The following element names, for example, are distinct in XML but identical in 4th Dimension:

```
    customer
    Customer
    CUSTOMER
```

As an example, the following text is not legal XML because the opening and close tags don't match case-sensitively:

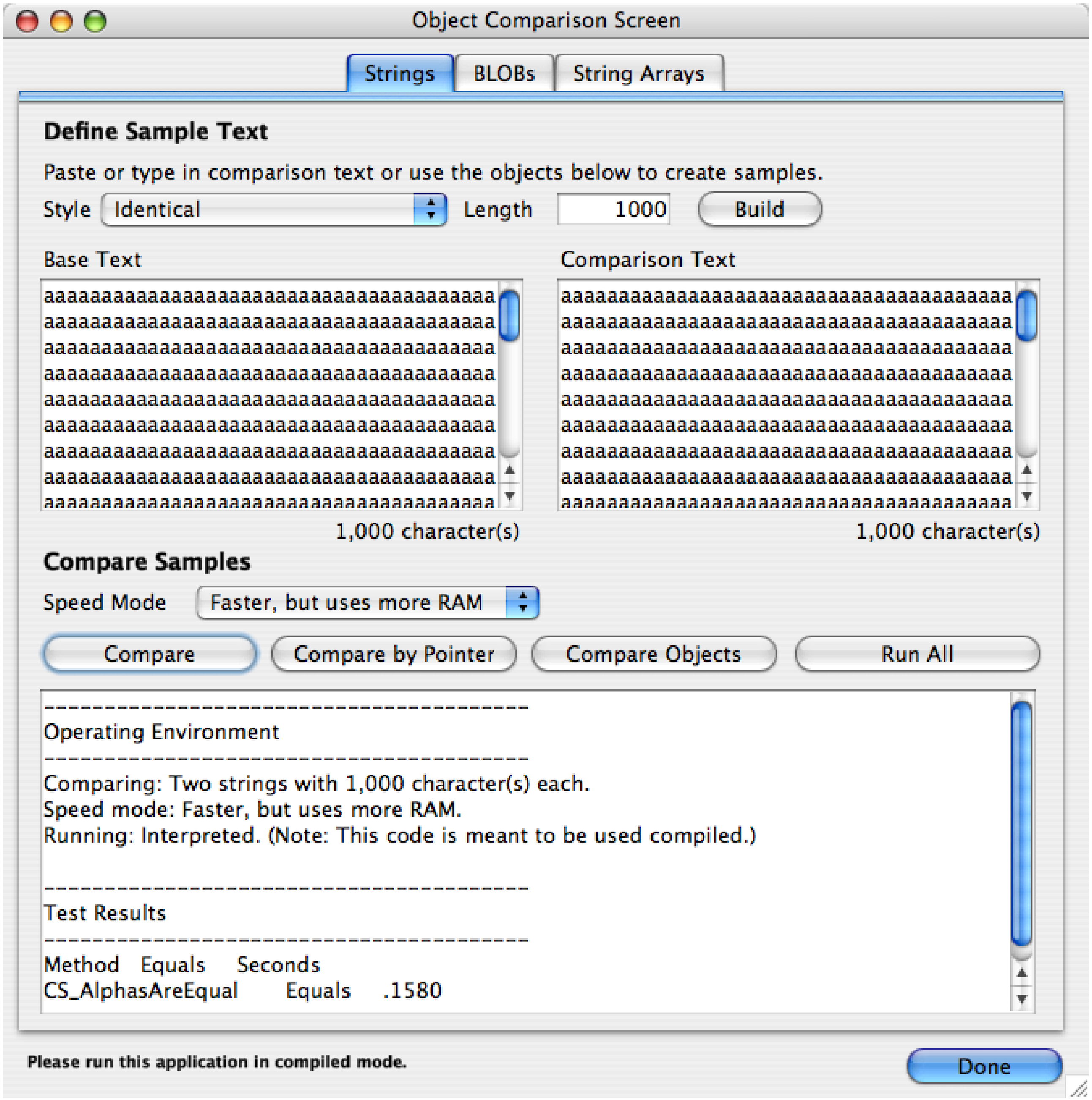
```
<?xml version="1.0" encoding="UTF-8"?>
<customer>
</Customer>
```

In practice, although most people writing XML don't depend on case to distinguish elements, some do. If your 4th Dimension code is case-blind, it may work. Then again, it may not. Worse, it may work today and then behave improperly if the underlying XML changes. It goes without saying that leaving the ultimate success of your code to luck is a poor practice. Rather, it is far better to make the effort to write case-sensitive code for case-sensitive operations. The explanations and sample code provided with this note should facilitate adding case-sensitivity to your project.

Now let's look at string comparisons in the 4th Dimension language. After that, we'll look at case-sensitive searching. First, let's take a glimpse at the demonstration interface.

String, BLOB, and Array Comparison Test Screen

The sample database includes a demonstration screen that lets you experiment with the string, BLOB, and alpha array comparison routines, pictured below:



Paste or type in comparison text or use the objects below to create samples.

Style: Identical

Length: 1000

Base Text

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

1,000 character(s)

Comparison Text

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

1,000 character(s)

Speed Mode: Faster, but uses more RAM

```
------------------------------------------
Operating Environment
------------------------------------------
Comparing: Two strings with 1,000 character(s) each.
Speed mode: Faster, but uses more RAM.
Running: Interpreted. (Note: This code is meant to be used compiled.)


------------------------------------------
Test Results
------------------------------------------
Method   Equals   Seconds
CS_AlphasAreEqual     Equals    .1580
```

Please run this application in compiled mode.

As you use this demonstration, keep a few points in mind:

- If you're interested in the code, you can trace it to see exactly how the various routines work.
- If you're testing performance, run the system compiled.
- The "speed mode" describes one option as faster and the other slower. Depending on the size of your samples, the faster option may be slower. In fact, the time/speed trade-off controlled by the speed mode setting only becomes meaningful with larger samples. We'll look into the speed mode next as it applies to many of the routines in the sample database.

Speed Mode and Performance Considerations

The pointer-based comparison routines documented below support an optional parameter called "speed mode". While these routines always take pointers to the values, the speed mode setting controls what is done with the pointer internally. The table below documents the behavior:

Code	Mode Name	Text	Blob
0	Default	Copy value	Read value through pointer
1	Faster but uses RAM	Copy value	Copy value
2	Slower but saves RAM	Scan through pointer	Scan through pointer

Internally, when a routine is told to copy a value, it takes the pointer and copies the text or BLOB value into a local variable. The scanning code then uses the local copy instead of reading the original through the pointer. Duplicating the value in this manner requires more RAM but saves time. In the case of BLOBs, scanning through a pointer is roughly 1.67 times slower than scanning a BLOB directly. You will certainly find different results if you test this yourself, but the figure cited here is a reasonable rule-of-thumb. For text, the behavior depends on the size of the text pointed to. In some versions of 4th Dimension, the rate of scanning a text value through a pointer changes dramatically when the text reaches around 15,000 characters. The scanning rate can slow down overall by as much as 100 times. Given this behavior, it is safer to copy the text than to scan text through a parameter. Since text values in 4th Dimension are limited to 32,000 characters, duplicating a single text value is unlikely to produce memory-related problems. The research and test code behind the information in this paragraph is included in Technical Note 05-42 **Scanning Text and BLOBs Efficiently**.

Note *Internally, 4th Dimension pointers are unlike pointers in languages like C and are considerably more expensive to dereference.*

Comparing Strings and Text

Compare Character Values, Not Characters

As noted, 4th Dimension ignores case and other character variations when making comparisons. For example, the following expression returns **True** in 4th Dimension:

```
"a"="A"
```

Reworking this behavior for text is easy: compare the ASCII codes of each character in a string rather than the characters themselves. The following expression returns **False** in 4th Dimension:

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

Sample Implementation

The expression shown above is all there is to case-sensitive string comparisons. To compare two string/text values case-sensitively, confirm that they have the same length and then loop through the values comparing each character's ASCII value. The following code fragment from the *CS_AlphasAreEqual* routine in the sample database fleshes out this idea:

```
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

The sample database includes two string-comparison routines for your convenience. Their behavior is identical but one takes pointers to the string/text values, and the other takes the string/text values directly as a parameter. See “Speed Mode and Performance Considerations” above for more discussion on these two approaches to passing values. The string-comparison routines are documented below.

CS_AlphasAreEqual

CS_AlphasAreEqual (Text;Text) Boolean
CS_AlphasAreEqual (Base text;Comparison text) Strings are equal?

This routine takes two string/text values as parameters and compares them case-sensitively. If the strings are identical, the function returns **True**; otherwise, it returns **False**. The example fragment below shows how a trigger can use *CS_AlphasAreEqual* to test if a customer’s name has changed case-sensitively.

```
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}) : Strings are equal?

This routine takes two string/text values by pointer and compares them case-sensitively. If the strings are identical, the function returns **True**; otherwise, it returns **False**. The example fragment below shows how a trigger can use *CS_AlphasAreEqual* to test if a customer’s name has changed case-sensitively.

```
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
```

The default speed mode for this routine is 1, “copy values”.

Comparing BLOBs

Why Compare BLOBs

4th Dimension's text fields can hold less than 32KB of text. In modern applications, it's often necessary to store text values far longer than 32KB. 4th Dimension's BLOB fields can hold enormous blocks of data but, unfortunately, are not as easily searched or compared as text values. For example, 4th Dimension's equals (=) operator doesn't work on BLOBs.

This note and the sample database include some code to help compare and find BLOB values. The string comparison code shown earlier is nothing more than a loop that steps through each character in the two strings comparing their ASCII values. The same code can be reworked slightly to operate on BLOBs. In fact, scanning BLOBs in this manner is slightly faster than scanning text because the **Ascii** function doesn't need to be called on each byte since BLOB bytes are already treated as numeric values.

Sample Implementation

The code fragment below from the *CS_BlobsAreEqual* routine shows how to compare two BLOBs:

```
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
```

Note *As the comments in the code fragment indicate, BLOBs are numbered as offsets from the start of the BLOB. Therefore, the first byte is numbered 0 and the last byte is numbered* **Size of blob**-1*.*

BLOB Comparison Routines Provided

The sample database includes two BLOB-comparison routines for your convenience. Their behavior is identical, but one takes pointers to the BLOBs to compare, and the other takes the BLOB value directly as parameters. See "Speed Mode and Performance Considerations" above for more discussion on these two approaches to passing values. The BLOB-comparison routines are documented below.

CS_BlobsAreEqual

CS_BlobsAreEqual (BLOB; BLOB) Boolean
CS_BlobsAreEqual (Base BLOB;Comparison BLOB) BLOBs are equal?

This routine takes two BLOBs as parameters and compares them case-sensitively. If the BLOBs are identical, the function returns **True**; otherwise, it returns **False**. The example fragment below tests if an incoming BLOB matches the value stored in a record that is already loaded:

```
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
```

Tip *You can test if two pictures are identical by converting them to BLOBs and then comparing the BLOBs.*

CS_BlobsAreEqual_Pointer

CS_BlobsAreEqual_Pointer (Pointer; Pointer;Longint) : Boolean
CS_BlobsAreEqual_Pointer (->Base BLOB;->Comparison BLOB;{Speed mode}) : BLOBs are equal?

This routine takes two BLOB values by pointer and compares them case-sensitively. If the BLOBs are identical, the function returns **True**; otherwise, it returns **False**. The example fragment below tests if an incoming BLOB matches the value stored in a record that is already loaded:

```
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
```

The default speed mode for this routine is 2, "scan through pointer".

Tip *You can test if two pictures are identical by converting them to BLOBs and then comparing the BLOBs.*

Comparing String/Text Arrays

As a convenience, the sample database includes a routine named *CS_AlphaArraysAreEqual_Pointer* that compares two string/text arrays. If the arrays are equal in size and each element has case-sensitively identical contents, the function returns **True**; otherwise it returns **False**. The parameters for this routine are listed below:

CS_ArraysAreEqual_Pointer (Pointer; Pointer;Longint) : Boolean
CS_ArraysAreEqual_Pointer (->Base array;->Comparison array;{Speed mode}) : Arrays are equal?

The default speed mode for this routine is 1, “copy values”.

General Comparison Routine

The sample database includes a general comparison routine named *CS_ContentsAreEqual_Pointer*. Internally, this routine tests what sorts of values are pointed to and then dispatches the call to the appropriate comparison routine: *CS_AlphaArraysAreEqual_Pointer*, *CS_AlphasAreEqual_Pointer*, or *CS_BlobsAreEqual_Pointer*. The parameters for this routine are listed below:

CS_ContentsAreEqual_Pointer (Pointer; Pointer;Longint) : Boolean
CS_ContentsAreEqual_Pointer (->Base value;->Comparison value;{Speed mode}) : Values are equal?

The items pointed to in $1 and $2 must agree. Therefore, you may pass pointers to two string/text arrays, two string/text values, or two BLOBs.

The default speed mode for this routine is based on the objects pointed to, as listed below:

Type	Code	Behavior
Alpha/text arrays	1	Copy value
Alpha/text	1	Copy value
BLOBs	2	Scan through pointer

Case-Sensitive String/Text Array Utilities

The sample database also includes case-sensitive versions of 4th Dimension's **Find in array** and **Count in array** routines, as documented below.

Note *These routines are not shown in the sample database demonstration interface.*

CS_CountInAlphaArray_Pointer

CS_CountInAlphaArray_Pointer (Pointer; Pointer;Longint) : Longint
CS_CountInAlphaArray_Pointer (->String/text array;->String/text value;{Starting element}) : Count

CS_CountInAlphaArray_Pointer is a case-sensitive replacement for 4th Dimension's **Count in array** function. By default, the routine tests all items in the array. Alternatively, you may pass a starting element position in $3. The code fragment below shows the command in use:

```
C_LONGINT($count)
$count:=CS_CountInAlphaArray_Pointer (->LegalElementNames_at;->$currentElementName_t)
```

CS_FindInAlphaArray_Pointer

CS_FindInAlphaArray_Pointer (Pointer; Pointer;Longint) : Longint
CS_FindInAlphaArray_Pointer (->Array;->Text value;{Starting element}) : Position

CS_FindInAlphaArray_Pointer is a case-sensitive replacement for 4th Dimension's **Find in array** function. By default, the routine looks starting at element 1 of the array. Alternatively, you may pass a starting element position in $3. The code fragment below shows the command in use:

```
C_LONGINT   ($element)
$element:=CS_FindInAlphaArray_Pointer (->CustomerXMLElementNames_at;->$currentElementName_t)
```

Case-Sensitive Searching

So far this note has considered code for comparing pairs of values case-sensitively. For many developers, a more pressing concern is searching records case-sensitively. This can be a critical requirement if, for example, you're working on a database that defines XML elements or case-sensitive paths within records. In such cases, 4th Dimension's **QUERY** command doesn't work satisfactorily. Additionally, if you are searching on text fields using 4th Dimension's standard **QUERY** commands, the = operator doesn't actually do a full equals comparison, even case-insensitively. As an undocumented behavior, the **QUERY** command appears to only scan the first 80 characters of text fields. Therefore, even if you are unconcerned about case-sensitive searches, the materials in this section may be of use to you. After taking a quick look at the demonstration database's test interface, we'll discuss four approaches to case-sensitive text searches:

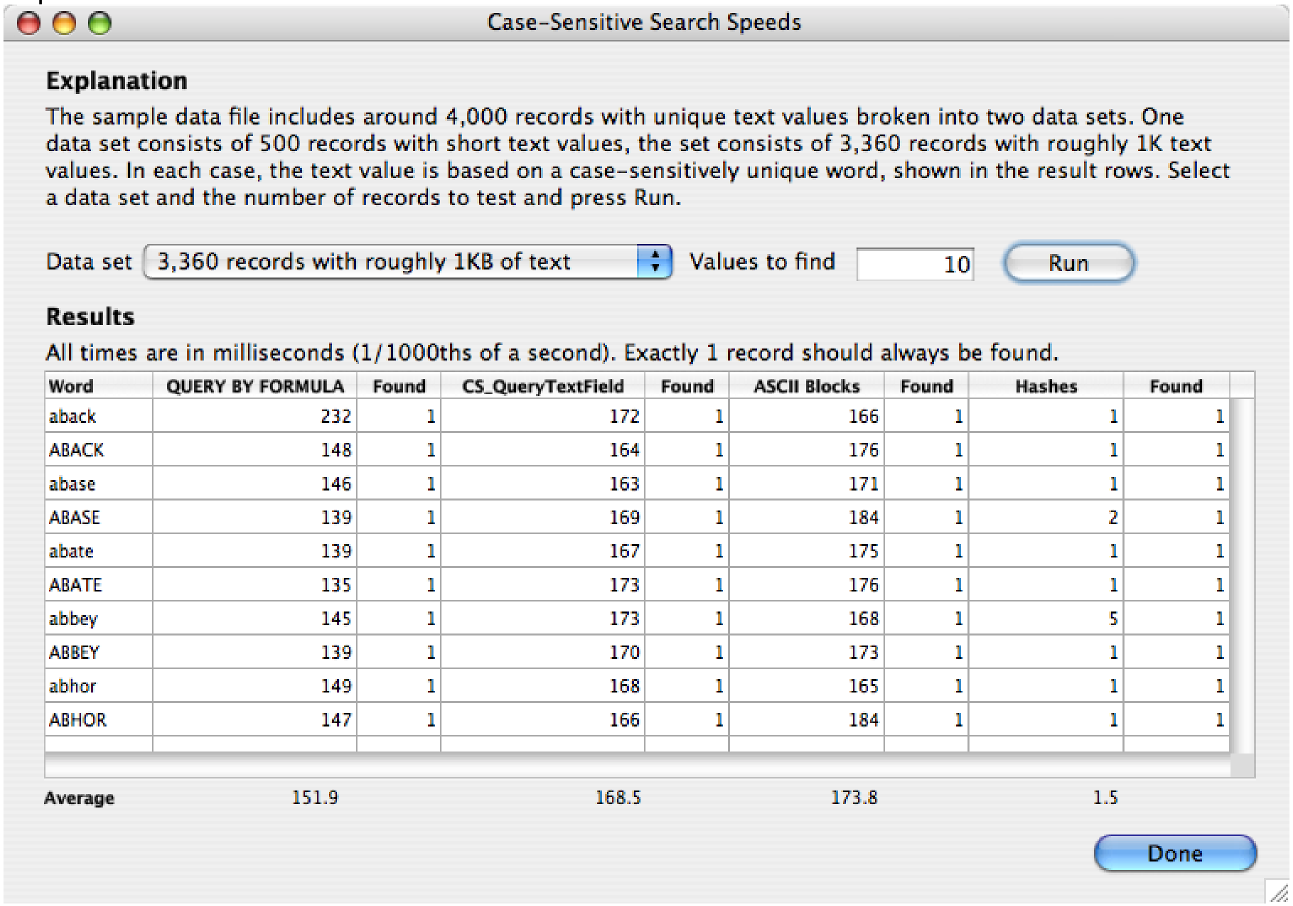
- Searching with **QUERY BY FORMULA**.
- Replacing **QUERY BY FORMULA** with a better-optimized alternative called *CS_QueryTextField*.
- Searching on stored ASCII block translations of stored text.
- Searching on hashes of stored text.

As a general rule, hashing is the best-performing and most scalable of these alternatives, particular under 4D Server. As we'll see, however, other techniques are worth considering in special cases.

Note *Any comments made about* **QUERY** *also apply to* **QUERY SELECTION** *and, likewise, any comments made about* **QUERY BY FORMULA** *also apply to* **QUERY SELECTION BY FORMULA**.

Search Speed Test Screen

The demonstration database includes pre-configured sample data and test screen for comparing the four text searching strategies listed above. A sample of the test screen is pictured below:



Case-Sensitive Search Speeds

Explanation

The sample data file includes around 4,000 records with unique text values broken into two data sets. One data set consists of 500 records with short text values, the set consists of 3,360 records with roughly 1K text values. In each case, the text value is based on a case-sensitively unique word, shown in the result rows. Select a data set and the number of records to test and press Run.

Data set: 3,360 records with roughly 1KB of text
Values to find: 10

Results

All times are in milliseconds (1/1000ths of a second). Exactly 1 record should always be found.

Word	QUERY BY FORMULA	Found	CS_QueryTextField	Found	ASCII Blocks	Found	Hashes	Found
aback	232	1	172	1	166	1	1	1
ABACK	148	1	164	1	176	1	1	1
abase	146	1	163	1	171	1	1	1
ABASE	139	1	169	1	184	1	2	1
abate	139	1	167	1	175	1	1	1
ABATE	135	1	173	1	176	1	1	1
abbey	145	1	173	1	168	1	5	1
ABBEY	139	1	170	1	173	1	1	1
abhor	149	1	168	1	165	1	1	1
ABHOR	147	1	166	1	184	1	1	1
Average	151.9		168.5		173.8		1.5	

The test screen lets you select a data set to search and a number of searches to perform. Perform at least a few searches as you'll find that there is some variation in results. For each test, you'll see a result time for each search strategy in milliseconds and the number of records that were found in each case. The number of records found should always be one, given the contents of the sample data. (The sample data was constructed to deliberately produce unique results to help verify that the search code is working correctly.) If you're interested in performance, be sure to run the database compiled. If you plan to deploy code under 4D Server, make sure to test under 4D Server.

After reviewing the four search strategies, we'll compare some search results under 4th Dimension and 4D Server. You'll see that the two environments can produce very different results.

QUERY BY FORMULA

The most obvious way to perform case-sensitive text searches in 4th Dimension is to use the **QUERY BY FORMULA** command, as in the example below:

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

The line of code shown above works correctly. Unfortunately, the **QUERY BY FORMULA** command is inexplicably slow, particularly under 4D Server. Depending on the data being searched, the other techniques described here can offer performance dozens and sometimes hundreds of times faster than **QUERY BY FORMULA**. First, we'll look at the most straightforward alternative.

Replacing QUERY BY FORMULA with Custom Code

The **QUERY BY FORMULA** code shown above does nothing but apply the *CS_AlphasAreEqual* function to each record in the table to see if the stored field matches a test value. The demonstration database includes a generalized routine to perform this task called *CS_QueryTextField*, illustrated in the line below:

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

The internal behavior of this command is outlined in pseudo-code below:

```
Build an empty numeric array to hold the record numbers of matching values

Load the record numbers of the records to test into an array
Load the text values from the records to test into an array

For (each element in the array of text values)
      If (the current element's value = the value we're looking for)
           Add the current record number to the array of matching values
      End if
End for
```

```
Build a selection from the array of matching values (may be empty)
```

The pseudo-code outline above is functionally equivalent to what **QUERY BY FORMULA** does: the value in each text field is compared case-sensitively with the value in a text variable, and a selection of matching records, if any, is built. Internally, both approaches use *CS_AlphasAreEqual* to perform the text comparison. Despite performing similar work, the two strategies often operate at vastly different rates. Depending on the data being tested, **QUERY BY FORMULA** is sometimes faster but generally slower than the replacement code under 4th Dimension. Under 4D Server, **QUERY BY FORMULA** is 55-81 times slower in some simple tests.

Note *Internally, the* CS_AlphasAreEqual *function implements the code outlined above with some refinements to avoid using up too much memory. Instead of loading all records into arrays, it only loads 256 values at a time. You can adjust this value, as you like, if you include* CS_AlphasAreEqual *in your own work.*

Storing Additional Data to Optimize Searches

A nice feature of **QUERY BY FORMULA** and the *CS_QueryTextField* approaches to searching is that they don't require any special preparation and can be used with any text field. You can, however, achieve greater performance gains by storing additional data. The idea is to store data that is small enough to be used in an indexed **QUERY**, or that is smaller than the original text but still unique, or that is easier to compare than the original text. Anything that reduces the number of records that have to be tested sequentially and/or that simplifies the necessary sequential operations can improve performance. We'll look at two approaches to stored data: ASCII blocks and hashes.

Working with ASCII Blocks

One way to work around 4th Dimension's inability to compare values case-sensitively is to convert the values into case-insensitive forms. ASCII blocks are one simple, if less than ideal, approach. The demonstration database includes a simple routine named *Demo_ConvertTextToAsciiBlock* that takes a block of text and converts it into an "ASCII block". The idea is straightforward: scan through the text block and convert each character into a new text block based on the ASCII codes of the original characters. For example, the string `abc` consists of three characters a, b, and c. The ASCII codes for these characters are 97, 98, and 99 respectively. Converted to an ASCII block, abc becomes `097098099`, the combination of each ASCII value. This numeric string is a complete and case-sensitive representation of the original string abc. While 4th Dimension can't see the difference between `abc` and `ABC`, it can easily see the difference between their ASCII block representations `097098099` and `065066067`. In practice, ASCII blocks can work well for values stored in arrays as they enable you to use 4th Dimension's **Find in array** command. For searches, however, ASCII blocks are less than ideal for several reasons:

- The ASCII block representation of a text is three times longer than the original text. Ideally, a rendered representation should be smaller, not larger. You could improve the situation by converting to hex, base-64, or another case-insensitive base.
- Under 4D Server, the ASCII block system doesn't deliver better results than *CS_QueryTextField*.
- The ASCII block value is a text field so, like the original, it can't be indexed for searching.
- The ASCII block representation of the text value must be updated each time the original text is modified, so the processing time is increased.
- The ASCII block increases the size of each record, effectively quadrupling the size of each text field. The ASCII block will, ultimately, need to be moved into a BLOB field if it is to completely represent text values longer than about 10,000 characters.

Despite these many disadvantages, the ASCII block system is a simple-to-understand example of how one value can be transformed into another to simplify some operation. For example, imagine a database that stores the names of 1,000 legal XML elements. Incoming XML documents and SOAP requests are validated against this catalog of 1,000 legal names. Instead of storing the ASCII blocks, they can be generated at startup in an array parallel to an array of legal names. In this situation, the ASCII block system and **Find in array** are all that is needed to validate each incoming XML element case-sensitively. Because **Find in array** is used instead of custom case-comparison codes, these comparisons will work very quickly, even in interpreted mode.

Using Stored Hashes

An excellent way to optimize searching for text case-sensitively is through stored hashes. Hashing is a technique for rendering a block of data, such as a string, BLOB, picture, or document, as a single longint. Hashes can be used as signatures or to simplify lookups. A powerful application for hashing in 4th Dimension is to optimize searches. Instead of performing sequential comparisons on all of the records in a table or selection, an indexed search can be run on the stored hashes of large data types to exclude most, or all, of the non-matching values from consideration. Imagine a table that stores 100,000 unique 1KB URLs. Searching through these values accurately is a lengthy operation using **QUERY BY FORMULA** or *CS_QueryTextField*. Why? Because all 100,000 records need to be compared sequentially. That's a lot of work. With a stored hash, the process is greatly improved, as outlined in the pseudo-code below:

```
Hash the test value
Search on an indexed stored hash field
Compare the resulting selection for matching values sequentially
```

The final comparison step is still a sequential comparison, but the number of records being tested should be significantly reduced. With a good hashing function, the chances are that the number of records to be compared will have been reduced to a single value or just a handful of values. As an example, many of the hashing algorithms from the HashTools component used in the sample database for this note produce 30,000 unique hash values for a data set of 30,000 unique words. If you're searching 100,000 text fields, it's a lot faster to first eliminate 99,999 or so from consideration.

Of course, nothing comes for free. The costs of stored hashes are listed below:

- The hash value needs to be rebuilt each time the source value changes. This task adds processing time. Fortunately, the hashing functions used here were refined with the chip in mind. These hashing functions are very, very fast once compiled.
- The hash value needs to be stored, so the size of each record increases. Since these values are longints, each hash only adds a few bytes per record and should not be a worry for most databases.
- During any particular search, the test value needs to be hashed, a procedure which takes some time. The exact time required depends on the hash function and the size of the search value.
- During any particular search, an indexed search is performed before any direct comparisons are made. Under certain situations, the time required for the search doesn't pay for itself. We'll discuss this point in more detail later when we review some test results.

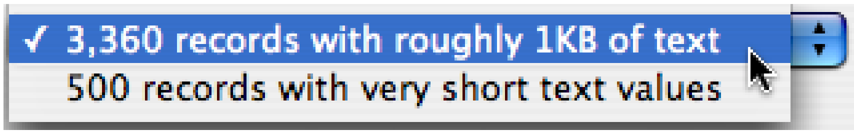
Within the sample database, the *HashTools_HashText* routine is used in the [Sample_Data] trigger to maintain a hash of [Sample_Data]Text, as shown in the code fragment below:

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

Drawing on this stored hash value, the demonstration database can use the *HashTools_FindByHash* routine to locate records. For more information on these routines, see Technical Note 05-43 **The HashTools Component**. For more information on the design and inner-workings of the component, see Technical Note 05-44 **Optimizing Text and BLOB Searches with Hashes**.

Discussion of Sample Test Results

The sample database includes two different sets of sample data, as indicated in the search test screen interface:


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

The main differences between the two data sets are the lengths of the text field's contents (about 80 characters version about 1,024 characters) and the number of records (500 versus 3,360). The system includes these different data sets because, depending on your equipment and network environment, they can lead to very different conclusions about how to perform case-sensitive searching. Consider some sample results with these two data sets running compiled under 4th Dimension:

Method	500 Short	3,360 Long
QUERY BY FORMULA	18.0	146.0
CS_QueryTextField	13.0	159.0
ASCII Blocks	8.0	161.0
Hashes	0.3	1.1

All times in milliseconds.

With the small data set, the differences between these techniques is relatively pronounced, with hashes performing 60 times better than **QUERY BY FORMULA** and 43 times better than *CS_QueryTextField*. With the larger data set of longer text values, the differences amongst the techniques change. Now, all of the techniques apart from hashing perform roughly the same. Next, let's look at the same tests run under 4D Server:

Method	500 Short	3,360 Long
QUERY BY FORMULA	1,495.0	43,354.0
CS_QueryTextField	27.0	535.0
ASCII Blocks	36.0	1,344.0
Hashes	110.0	112.0

All times in milliseconds.

There are some surprises in these results. For the smaller data set, the hashing system is slower than the ASCII block and *CS_QueryTextField* techniques. On the face of it, these results makes no sense at all. Both the ASCII block and *CS_QueryTextField* approaches require the code on 4D Client to download 500 text values for comparison. The hashing system, however, only requires one text value to be downloaded for final comparison. What these results show, however, is that there is a cost to performing an indexed longint search under 4D Server. In the case of these 500 records with very short text values (about 80 characters), the threshold where the indexed search pays for itself has not been crossed. The test results with 3,360 1KB text values produces more expected results. Here, the speed of using a hash doesn't change meaningfully (in both cases it returns one record in our sample data). The other techniques, however, suffer markedly.

Note *The exact numbers shown here aren't important as only the overall performance trends are relevant. You will certainly see different values if you run the tests on your own equipment or even if you run the tests more than once.*

Summary

This note discusses two different areas of case-sensitivity: comparisons within the language and comparisons by the database engine. Given the search results shown above, a few guidelines and conclusions can be offered about case-sensitivity and the 4th Dimension database engine:

- Performance under 4th Dimension and 4D Server can be very different. Test in the environment you plan to use.
- Artificial test data, such as the 500 record data set discussed above, can lead to bad conclusions if considered in isolation of other results.
- **QUERY BY FORMULA** is always slow. Use *CS_QueryTextField* or stored hashes.
- Stored hashes pay for themselves rapidly if you are working with anything but trivial amounts of data. Additionally, they scale beautifully. If you are dealing with large blocks of text, pictures, BLOBs, or linked documents, hashes work well. All of the other techniques mentioned here are as flexible, extensible, or scalable as hashing.

For case-sensitive operations through the language, the note and its sample database provide a suite of utility routines to compare all sorts of data, including alpha/text fields, alpha/text arrays, and BLOBs. Given that this code is already written for you, it is worth the effort to add case-sensitivity to your databases, when needed. Pay particular note to handling case-sensitive comparisons correctly if you are doing any work with SOAP messages or XML documents. **XML element and attribute names should always be considered case-sensitively**. With the materials available to you here, you should be able to support XML name comparisons easily.