Scanning Text and BLOBs Efficiently Or Testing Performance Meaningfully

By David Adams

Technical Note 05-42

Overview

Many programming tasks require scanning the contents of a string, text block, or BLOB character-by-character or byte-by-byte. For example, implementing case-sensitive string comparisons, many custom parsers, and comparing pictures, BLOBs, or documents may require sequentially scanning large blocks of data. The optimal method for sequentially scanning text and BLOBs is a subject programmers can debate passionately, given their various biases, experiences, goals and values. Such debates remain exchanges of opinion unless real and repeatable test results can be brought into the conversation. This technical note includes a sample database that generates tests results for several different approaches to scanning text and blobs. While finding optimal approaches to scanning is the purpose of the test system, it is only one of the aims of this technical note. The other objective is to illustrate how test results can be either helpful or dangerously misleading, depending on their design, execution, and application. After reading this note, you should have:

- A clearer understanding of the trade-offs to be made between speed and memory during text/BLOB scanning.
- A workable set of strategies for designing and using scanning code.
- An awareness of a very specific, and very dramatic, performance issue related to scanning text through a pointer.
- A practical appreciation of some of the main points to consider when designing or interpreting test results.

Apart from the sample testing database included in this note, also see Technical Notes ##-## **Case-Sensitive Operations in 4th Dimension**.

Scanning: Small Differences Accumulate

As an introduction to why it is important to occasionally look closely at the performance characteristics of code, consider the test results summarized below for operations scanning 32,000 characters of data:

Type of Data	Comparison	Speed
Text	Scan original text directly.	2
Pointer to text	Duplicate original text and scan copy directly.	2
Pointer to text	Scan original text through a pointer.	710
BLOB	Scan original BLOB directly.	1
Pointer to BLOB	Duplicate original BLOB and scan copy directly.	1
Pointer to BLOB	Scan original BLOB through a pointer.	4

Surprisingly, scanning a block of text through a pointer takes over 700 times longer than scanning the same data directly in a BLOB. (Don't stop reading now as later we'll see why these results don't tell the whole story.) This sort of finding can't easily be predicted or explained. Before looking at these and some other test results more closely, we'll review the basic test setup and factors contributing to the speed differences between the methods.

About the Test Database

The original test database used to generate the results discussed in this note is included for your use. The best way to generate meaningful results is to run tests on equipment typical of your environment. Additionally, you may find bugs or develop additional tests to improve the value of the results in your systems.

About the Test Results

All of the results listed in this note are based on actual tests in a database running under 4th Dimension 2004.2. Unless otherwise stated, the results don't show clock times. Instead, the results are normalized against the best result. Therefore, the quickest result appears as 1 and all other results are a factor of the quickest result. For example, in the results shown above, scanning a BLOB through a pointer is 4 times slower than scanning a BLOB directly. (Over a wide range of tests, the ratio is more typically 1:1.67.) For actual times in milliseconds, run the test database on your own system.

Note *All results are based on a compiled database. Since operations on text and BLOBs benefit dramatically from compilation, this note doesn't consider interpreted behavior. Compiling with range checking on or off makes little difference to the results*

Review: Syntax for Scanning Text and BLOBs

Reading an individual character from a string or block of text is conceptually identical to reading an individual byte from a BLOB. However, the syntax and numbering rules for strings/text and BLOBs differ in 4th Dimension. For review, the table below illustrates the main rules applied to a block of text and a BLOB, each with 32,000 bytes of data.

	Syntax	First	Last	Length
Text	text[[1]] text≤1≥ (Mac OS)	1	32,000	32,000
BLOB	BLOB{0}	0	29,999	32,000

The main points in the table are restated below descriptively:

- Text is read from character 1 while BLOBs are read as offsets from byte 0. Therefore, the first byte in a string is at position 1 and the first byte in a BLOB is at position 0.
- The length of a string or block of text, read with **Length**, equals the actual number of characters stored. The size of a string or block of text, read with **Blob size**, equals the actual number of bytes stored. Therefore, the length of a block of data stored as text equals the length stored as a BLOB. However, since BLOBs are numbered from 0, the last byte in a BLOB is address at length-1.
- The syntax for reading characters from a string or block of text can be either [[character_number]] or ≤character_number≥ on Mac OS and must be [[character_number]] on Windows.
- The syntax for reading bytes from a BLOB is {byte_number}. The curly braces are also used as the syntax for reading elements in an array. Conceptually, you can think of BLOBs as an array of bytes.

Review: Passing Parameters by Value or Pointer

4th Dimension supports passing parameters by value or by pointer. The table below illustrates the relevant syntax for a subroutine reading text or BLOB copied into a parameter or passing indirectly as a pointer:

Type of Data	Pass By	Scan	Example
Text	Value	Copy in parameter	$1[[$character_index]]
Text pointer	Pointer	Original through pointer	$1->[[$character_index]]
BLOB	Value	Copy in parameter	$1{$byte_index}
BLOB pointer	Pointer	Original through pointer	$1->{$byte_index}

Passing by value or by pointer each have their respective advantages and disadvantages. When passing a parameter by value, 4th Dimension copies the original value into the parameter. Therefore, the contents are duplicated and additional memory is consumed. Below is an example of passing a BLOB by value:

ScanTest_BlobParameter (ScanTest_Sample_blob)

Within the *ScanTest_BlobParameter* subroutine, the $1 parameter contains a copy of the contents of ScanTest_Sample_blob variable. If code clears the contents of the

ScanTest_Sample_blob, the contents of $1 within *ScanTest_BlobParameter* don't change. The advantage of passing parameters as values is that the subroutine can work directly on the data, as in the code fragment below:

```
$1{$byte_index}
```

The disadvantage of passing parameters by value is that extra memory is consumed. If a 2MB BLOB is passed as a parameter, at least 4MB of memory are consumed, 2MB for the original and 2MB for the copy. If memory is a concern, the alternative is to pass a pointer to the original data. When passing by pointer, the original data is not copied. Below is an example of passing a BLOB by pointer:

```
ScanTest_BlobPointer (->ScanTest_Sample_blob)
```

Within the *ScanTest_BlobPointer* subroutine, the $1 parameter contains a pointer to the ScanTest_Sample_blob variable. If the routine clears the contents of the ScanTest_Sample_blob, the original contents $1 points to no longer exists. The advantage of passing pointers is that the system spends less memory. The disadvantage is that reading the contents of the original value requires eliminating one level of indirection by dereferencing a pointer, as in the code fragment below:

```
$1->{$byte_index}
```

There is a cost to dereferencing a pointer, which, per operation, is quite small but it can add up to something measurable when iterated tens of thousands or millions of times.

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

Results Revisited

Keeping in mind the points about memory and performance just made, let's consider some test results. The tests were conducted by scanning text and BLOB variables with 32,000 characters. Each test was performed over 50 times and then averaged. The range of minimum and maximum values for any particular operation was slight. All tests were performed in a compiled database. Remember that the times are factors of the quickest result, not absolute clock measurements. The results or ordered from fastest to slowest:

Type of Data	Comparison	Speed
Pointer to BLOB	Duplicate original BLOB and scan copy directly.	1
Pointer to text	Duplicate original text and scan copy directly.	2
Pointer to BLOB	Scan original BLOB through a pointer.	4
Pointer to text	Scan original text through a pointer.	710

A few obvious implications and conclusions can be drawn from the results shown above:

- Operations on pointers are expensive. A pointer makes the scanning operation 355 times slower than a direct read for text and 4 times slower for BLOBs. Therefore, you shouldn’t use pointers.
- Operations on text are expensive. A direct text scan is twice as slow as a direct BLOB scan and a pointer-based text scan is nearly 180 times slower than a pointer-based BLOB scan. Therefore, you should use BLOBs instead of text.

The conclusions just offered are based on reproducible test results and sound logic. They're also sometimes bad advice. While the results shown are correct, they are also incomplete. So, we'll look at some more results and then consider how to apply test results.

Tests Results Are Easy to Overapply

Benchmarks and test results are very compelling pieces of information in a discussion. Unfortunately, it is all to easy to make any or all of the following errors through the incomplete or erroneous testing, or the misapplication of valid results:

- Relying on inaccurate or meaningless results.
- Believing results show a causal relationship that doesn’t exist, thereby reinforcing prior prejudices.
- Overgeneralizing conclusions, such as applying valid results from unusual boundary cases that may not hold true for more typical situations.
- Overlooking incomplete test results, or only true of unusual cases.

Despite these hazards, testing is an extremely worthwhile practice. Ideally, tests and test results should be reviewed by more than one person. It is impossible for a single programmer to see the biases and prejudices one brings to test design and interpretation.

This note includes the original test database to give you a chance to confirm or challenge both the results and methodologies presented here. If you find problems or gain new insights, please write me at dpadams@island-data.com

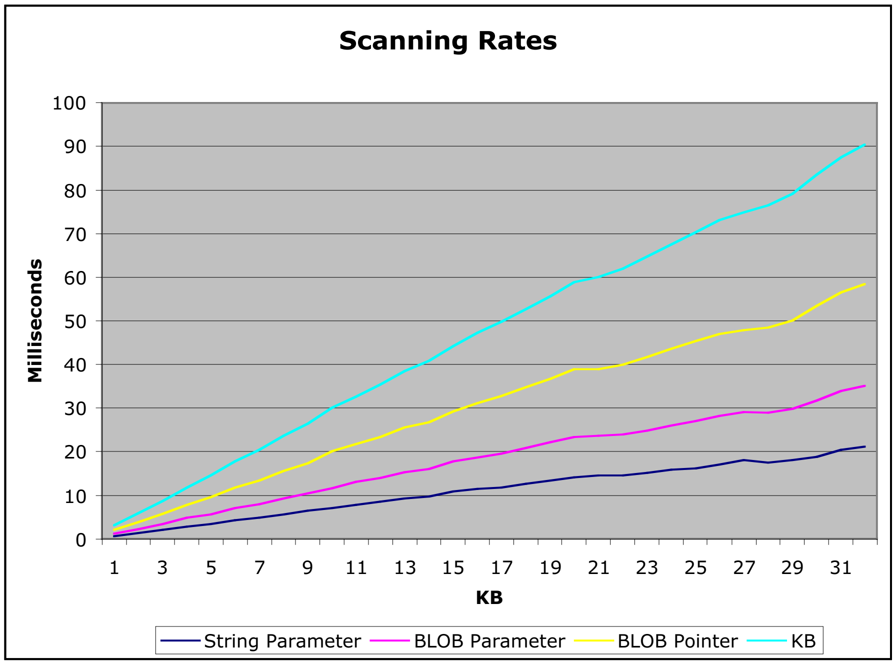
Now, with all of those warnings about testing in mind, let's look at test results starting from different initial conditions.

How Slow Is It to Read Text Through a Pointer?

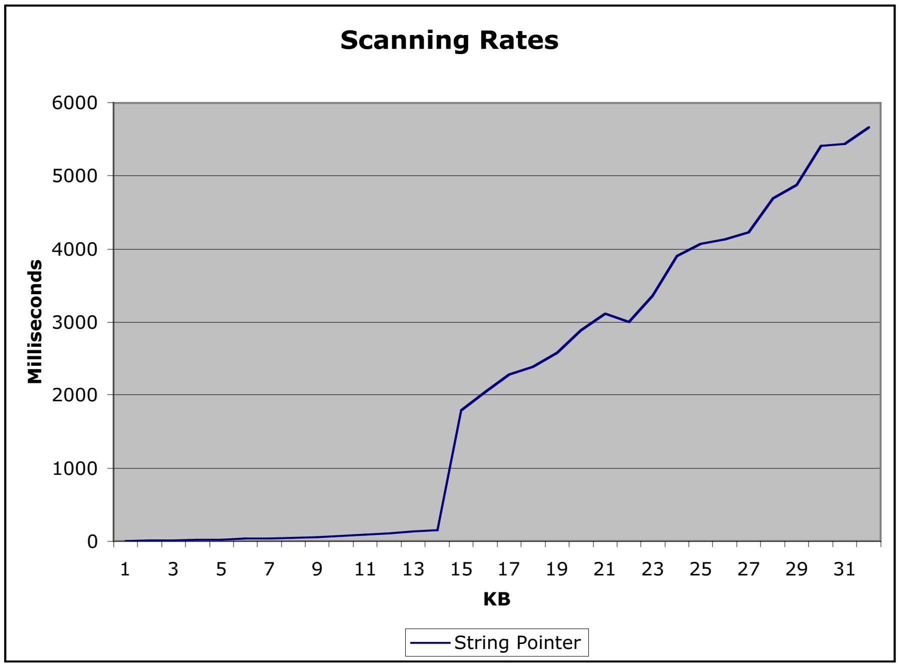
The results below are based on identical tests to those already listed but with sample text/BLOBs of 1,000 characters instead of 32,000 characters. (The values are shown to two decimal places as the range of values is much smaller than in the previous case.) Again, the results are normalized to the quickest operation and ordered from fastest to slowest.

Type of Data	**Comparison**	**Speed**
Pointer to BLOB	Duplicate original BLOB and scan copy directly.	1.00
Pointer to BLOB	Scan original BLOB through a pointer.	1.45
Pointer to text	Duplicate original text and scan copy directly.	5.69
Pointer to text	Scan original text through a pointer.	7.25

Since these results are all adjusted to show the relative performance of each technique within a test setting, the ratios should remain nearly the same across tests. As the two result tables show, such is not the case. The most dramatic difference shown above is that on a 1,000 character test, using a text pointer is roughly 7 times slower, while with a 32,000 character test, using a text pointer is roughly **700** times slower. Any time you get unexpected results that differ by two orders of magnitude, you should be highly suspicious of the tests. If the tests are sound, you may need to design additional tests to figure out what is really happening. In this case, you would expect the time taken to scan 32,000 characters by any technique to be roughly 32 times longer than the time for scanning 1,000 characters. The existing test code suits itself to testing this assumption. The chart below summarizes the results of a series of tests that start with 1,000 characters, 2,000 characters, and so on. As the chart below shows, the rate follows the idealized curve represented by KB. As the number of characters increases, the time required to do a complete sequential scan increases proportionally, exactly as we should expect.



The chart shown above leaves out reading text through a pointer as the results are so skewed that the chart becomes unreadable. Below is a chart showing the curve for scanning text through a pointer:



In this test setting, the rate progresses evenly until around 14,000-15,000 characters when it suddenly defines a nearly perpendicular slope. The performance of scanning text through a pointer differs radically, depending on the number of characters in the text value.

Note *Note that on different equipment or using a different version of 4th Dimension, the position of the sudden change in scanning rate is likely to move or even disappear.*

The Danger of Incomplete Testing

The two sets of benchmarks shown so far illustrate the dangers of incomplete testing. Two developers using identical test code could reach very different conclusions about the cost of using pointers, depending on the size of their test strings. A developer using a 10,000 character string could legitimately claim that pointers aren't too expensive while a developer using a 20,000 character string would find pointers hundreds of times more expensive. This example shows how incomplete testing can lead to mistakes. When designing tests, try values from a wide range. Make this an automatic part of your test design because it is essentially impossible to notice that a result is "missing" from a test design. Rows and columns of numbers look reassuringly complete but can very easily be telling a misleading story.

Why Does the Text Scanning Rate Vary?

An obvious question is why does the performance of text scanned through a pointer fall off a cliff? The only way to answer to this question is to inspect the internals of 4th Dimension itself. We can't determine a reason conclusively through external testing.

While it is easy enough to come up with theories about why this behavior exists (or even why it can't possibly exist), at the end of the day, the theories don't matter. What matters is reproducible behavior and how you manage unusual performance characteristics.

Numbers Are Meaningless without Context

It’s all too easy to look at test results and immediately jump to hard-and-fast conclusions. More often, the best approach depends on the larger context. Consider again the conclusions the initial test results lead to:

- You should not use pointers.
- You should use BLOB instead of text.

Despite what the original numbers say, there are any number of reasons to reject these conclusions as absolutes:

- Pointers avoid duplicating values, thus saving memory. In the case of large BLOBs, this can be an important consideration. This benefit is worth spending some time for. How much it is worth paying depends on the operating environment and customer requirements for the overall system. We'll look at this subject again in a moment.
- Pointers are one of the best ways to make code more reusable in 4th Dimension thus reducing code redundancy and inconsistencies. Avoiding duplicate work and lowering maintenance costs is a substantial benefit.
- Operations on BLOBs may be faster than operations on text, but many developers find BLOBs a pain to work with. Many of 4th Dimension's commands, such as **Position**, don’t work on BLOBs. Perhaps it’s worth paying a speed penalty to get the convenience of working with native commands and easy-to-read textual data.

As the previous points illustrate, choosing between alternatives in programming is rarely simple and clear-cut. The best approach is almost never free. Figuring out how to balance the pros and cons of various approaches is a judgment call. Ideally, the decisions are based on real user requirements, such as specific performance benchmarks or memory-use restrictions.

How Fast Is Fast?

Given that, under typical conditions, pointers are slightly slower than direct access, let's look a bit more at what "fast" and "slow" mean. The tables of test results used in this note show comparative times, not the milliseconds generated by the test code. This approach highlights the relative performance of each technique, but be careful not to consider speed factors in isolation. If a solution is "1,000 times faster" but the original takes only 1 millisecond and doesn't run in a loop, it is completely meaningless that an alternative is 1,000 times faster. **You can never save more time than the original operation requires**. If a task takes 1 second you can save, at most, 1 second. In the case of examining test and BLOBs, consider a finding from the first table: scanning a BLOB through a pointer takes four times longer than scanning a BLOB copied directly into a parameter. "Four times longer" sounds dramatic and compelling. (As noted above, testing with various BLOB sizes shows a more typical ratio of 1:1.67.) However, measured against the clock, the difference amounts to less than a 2 second difference when scanning a 1MB BLOB on modest contemporary hardware. Is it worth spending 1MB of memory to buy less than a second? It may or may not be, depending on what else the application needs to do and if users are present. For an unattended process, 2 seconds is meaningless while for a user waiting for a screen to redraw it is a very long time indeed. Be sure to calibrate test results with real needs.

Available Memory and Operations on BLOBs

The test results consistently show that working through a pointer is slightly to somewhat slower than working on a BLOB directly. Therefore, for best speed, it seems obvious that values should be passed instead of pointers. In the case of BLOBs, however, this idea deserves careful scrutiny. Since passing by value creates a duplicate, any BLOB passed by value forces 4th Dimension to find enough contiguous memory to hold the copy. In the case of a large BLOB, it can be expensive. While 4th Dimension 2004 only runs on modern operating systems with decent virtual memory schemes, moving memory is never entirely free. In fact, if memory is tight and contents need to be shifted to create room for a BLOB, overall performance will go down. Accordingly, it is possible that, under certain memory conditions, accessing a BLOB through a pointer will be faster than reading the BLOB directly as a parameter. You could construct a test system to confirm this behavior but, as a more immediate consideration, 4th Dimension can fail on BLOB operations when enough memory is not available.

Personal Recommendation

As a personal recommendation, I use the following as default rules for scanning operations, in the absence of any specific guidance from user requirements:

- Pass text by value (directly) to avoid falling off the performance cliff illustrated in the charts above.
- Pass BLOBs by pointer (indirectly) to avoid consuming extra RAM.

These rules-of-thumb are one programmer's starting point and are not absolute rules. You can use different defaults, as you see fit and break the rules when a particular situation obviously favors saving memory or saving speed.

Summary

This technical note examines the speed and memory differences between scanning text and BLOBs passed by value or by pointer. A testing database is included that generates speed comparisons used in this note to develop rules of thumb for designing text and BLOB scanning code. The test system and test results are a point of departure for looking at some of the hazards and principles of performance testing.