Optimizing Searches with Hashes

By David Adams

Technical Note 05-44

Overview

4th Dimension's database engine doesn't natively support searching strings case-sensitively, searching text fields completely, or searching BLOBs, picture fields, or documents at all. Hashing is a conceptually simple, well-known technique for optimizing all of these operations, in specific cases. This technical note explains what hashing is and how to apply it to various problems, including searching, document comparison, and validation check sums. By the end of this technical note, you should understand:

- How hashing works.
- When to use hashing.
- How to select a hashing algorithm.
- How to further optimize searches to avoid **QUERY BY FORMULA**.

Code is included that supports hashing alphas, text, BLOBs, pictures, and documents. It also provides a hash-optimized query method for alpha, text, BLOB, and picture fields. After briefly describing the contents of the sample database, we'll review some background information about hashing.

Note *The hashing techniques and tools discussed here can be applied to alphas, text, BLOBs, pictures, and documents. For simplicity, the discussion usually refers to strings, text, or BLOBs.*

About the Samples

Packaging

The sample code is included as a component called HashTools, as a sample database, and as a source code database. The component is documented in Technical Note 05-43 **The HashTools Component**. The sample code and source code databases are nearly identical as each of them has the same example interface, which we'll review briefly. The main difference between the two is how the HashTools code is included. In the sample database, the code is in a component and this is helpful in day-to-day use or when you only want to look at the functionality of the system. The source code database includes the code the HashTools component and examples are based on. If you want or need to look at the internals of the HashTools system, run the source code database, which enables you to trace inside methods that are protected and private once they're installed as a component.

Supported Hashing Algorithms

The HashTools code implements eight high-quality string hashes named *AP*, *BKDR*, *DJB*, *ELF*, *JS*, *PJW*, *RS*, and *SDBM* adapted from code published at:

http://www.partow.net/programming/hashfunctions/index.html

Additionally, a simple-to-understand but lower quality hashing function named SumBytes is included for comparison. The object hashing and hash-optimized query methods each support all nine hashing algorithms listed in this paragraph.

Data Sets

As we'll discuss, the ideal hashing function depends partially on the specific data being hashed. To illustrate and support testing this principle, the database includes several different sample data sets for comparison. You can import values from your own systems to test and compare how each hashing performs in three areas:

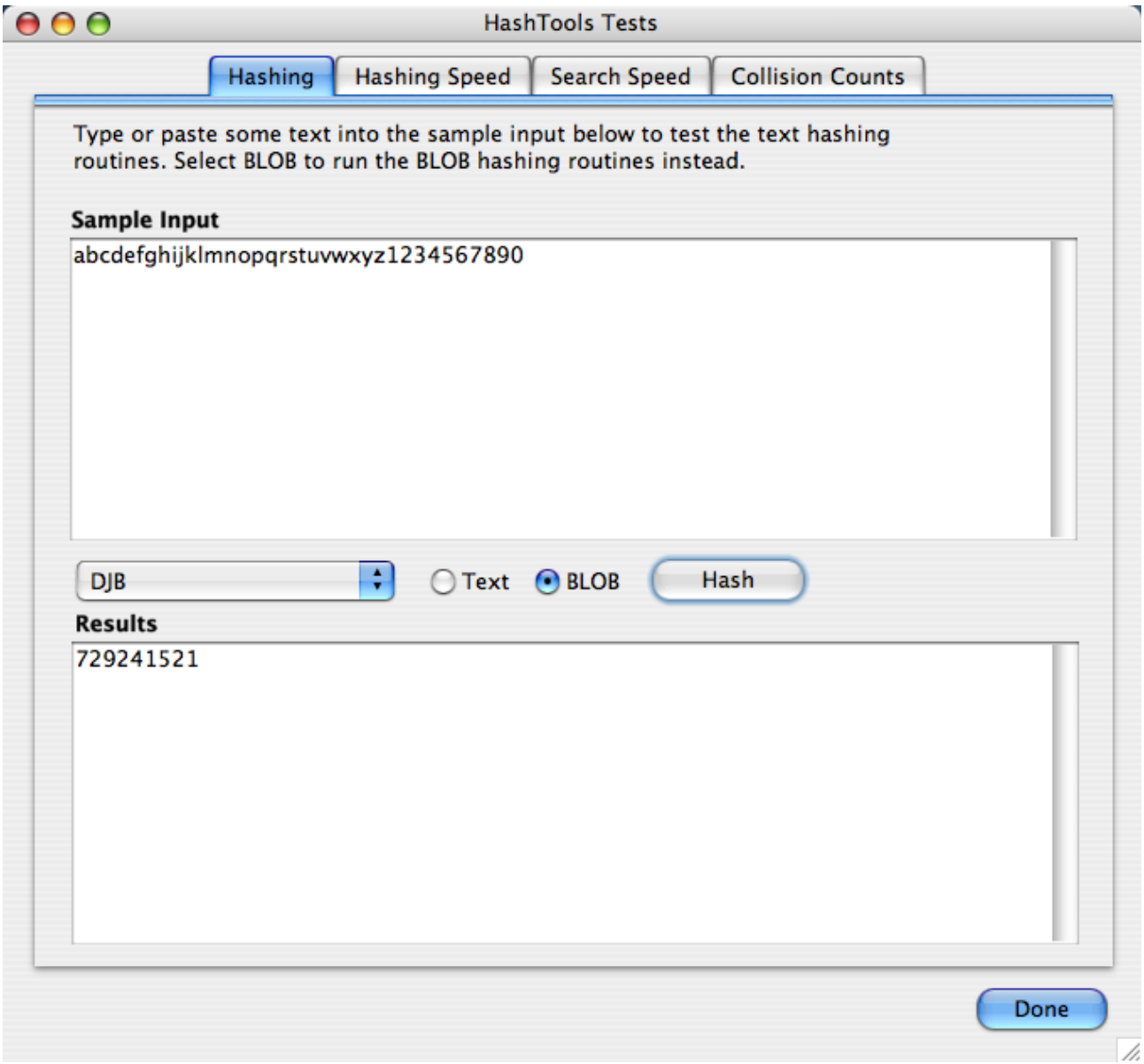
- The time needed to hash the data.
- The number of collisions (duplicate hashes from distinct values) created.
- The speed gains achieved using a stored hash when querying.

We'll address each of these points in more detail throughout the note.

The Demonstration Test Screens

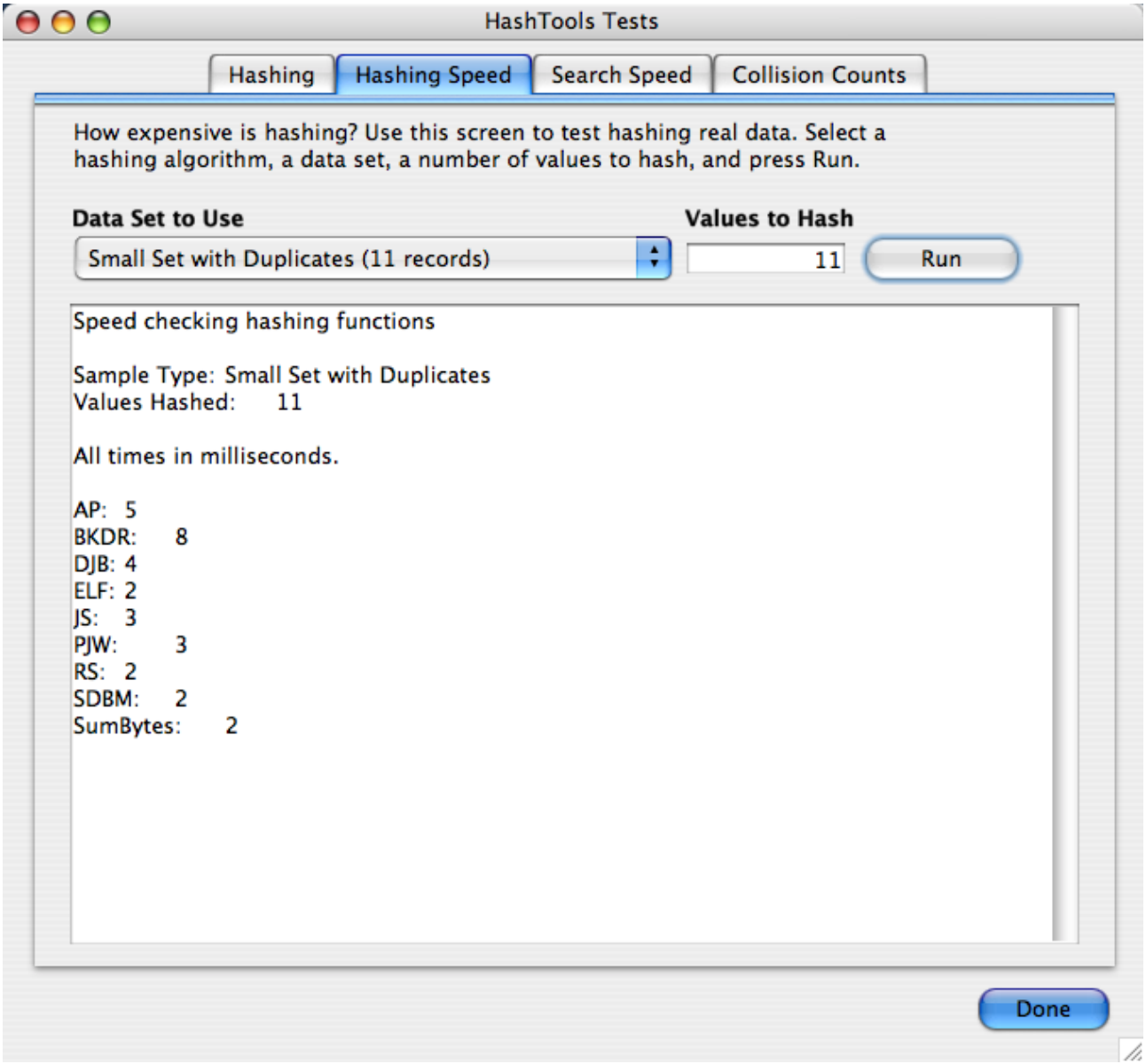
The demonstration window consists of four screens, pictured and briefly described below.

Hashing Tests



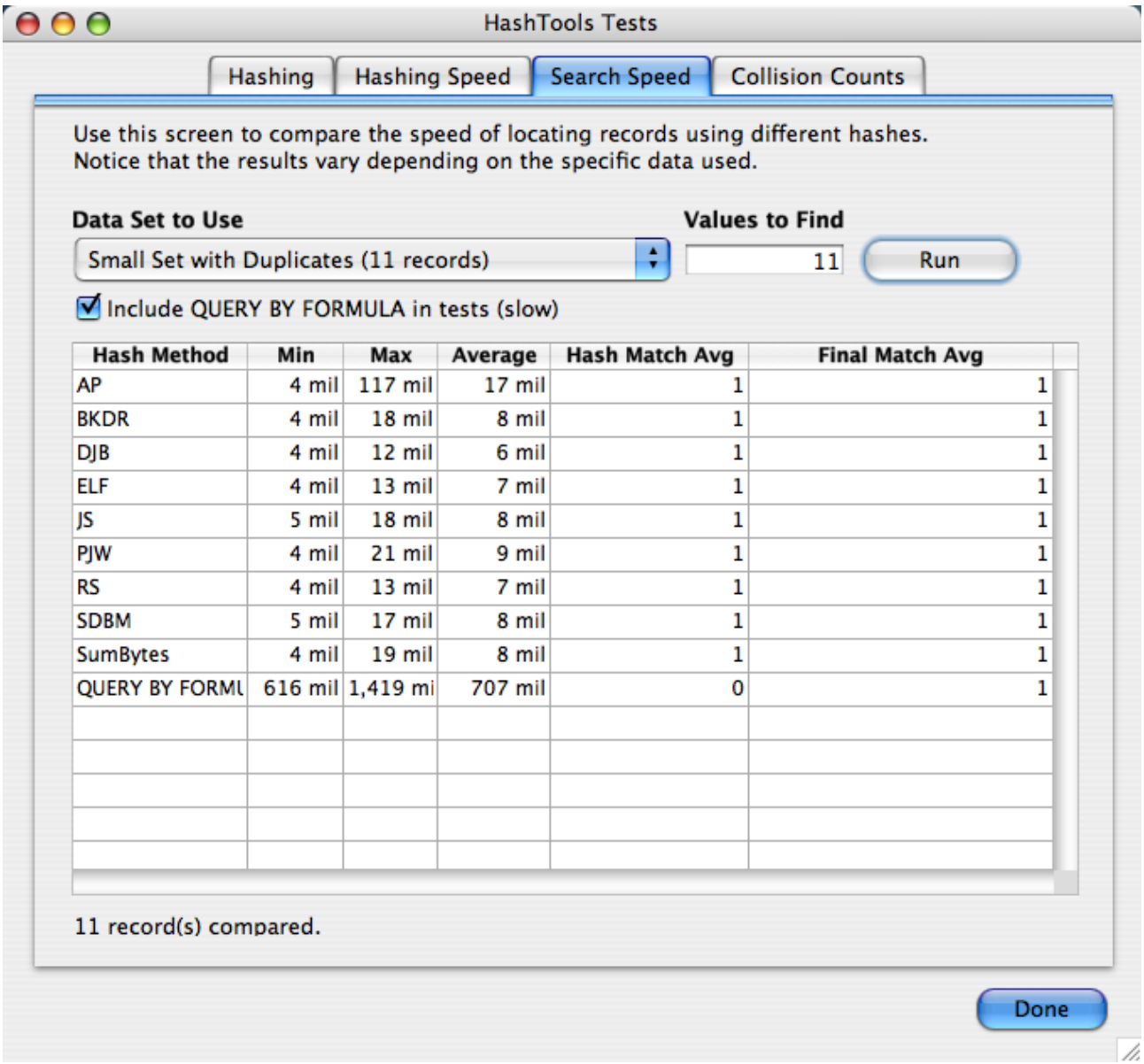
The hashing test screen lets you type or paste in a block of text for hashing. The value can be hashed using any and all of the available algorithms as text or as a BLOB. This screen is useful if you want to double-check a value or, in the source code database, to see the actual functioning of a hashing algorithm.

Hashing Speed Tests



The hashing speed test screen lets you check how long it takes to hash data using the different algorithms. You can use any of the included sample data sets or import your own data for more meaningful results. Run this screen compiled for an accurate sense of performance.

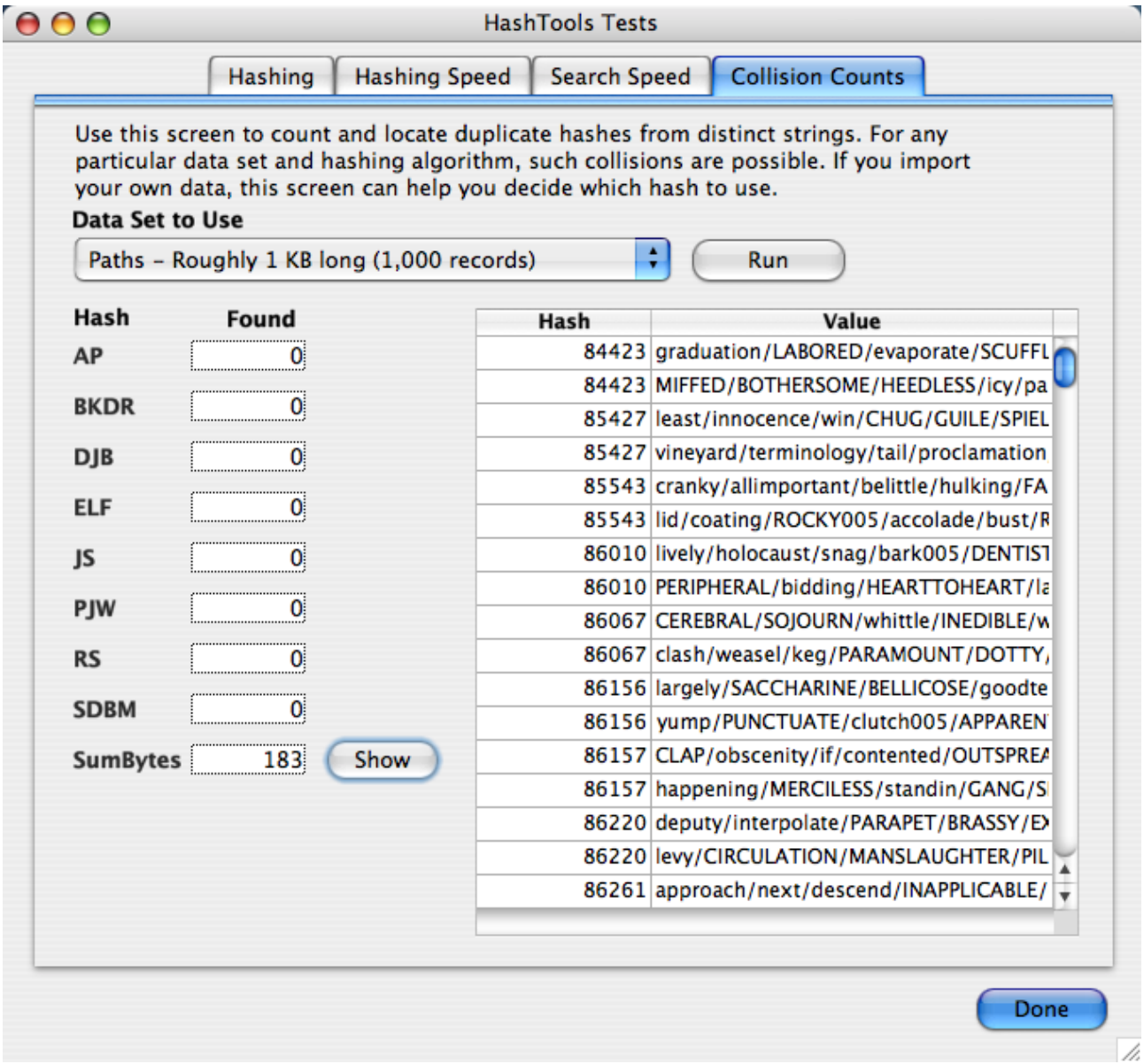
Hashing Search Speed Tests



The search speed test screen lets you check how much different hashing algorithms assist searches for any particular data set. A few notes:

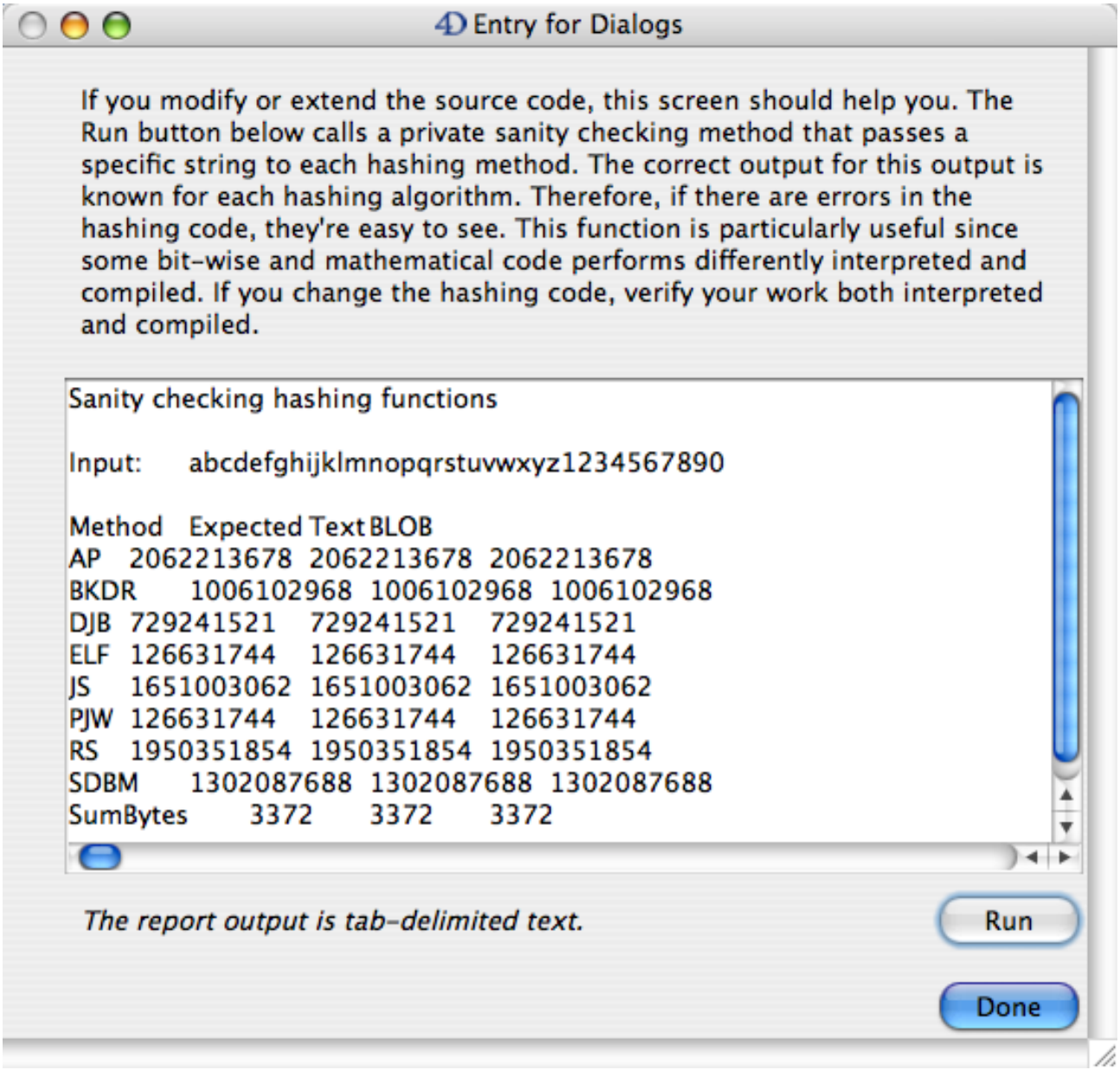
- Performance can be very different under 4th Dimension and 4D Server. Test in the environment you plan to deploy in.
- Test compiled for an accurate sense of performance.
- **QUERY BY FORMULA** can significantly slow the tests, particularly under 4D Server. If it's too slow, don't include it in the tests.
- Import your own data for more meaningful results.

Hashing Collision Tests



Any hashing algorithm can produce collisions, duplicate hashes from distinct values. This screen lets you test how many collisions each algorithm produces for any particular data set. Additionally, you can display the collisions for any particular algorithm. Import your own data to help select the best algorithm for your environment.

Source Code Only: Sanity Checking Screen



The source code database includes a form named [Dialogs];"HashTest_SanityCheck" that calls a private method named *HashUtility_SanityCheckFunction*. This routine passes a specific input with known correct outputs to each of the text and BLOB hashing routines and reports the results. If any of the functions returns an unexpected result, you can spot the problem quickly. This sanity checking screen is useful only if you are reworking the source code of the hashing functions themselves.

Related Technical Notes

A series of complementary notes discusses other applications for hashing and performance considerations contributing to the design of the code found in the sample database.

- Technical Note 05-43 **The HashTools Component** documents the public and protected methods in the HashTools component. This component is built from the full source code accompanying this note.
- 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 05-41 **Case-Sensitive Operations in 4th Dimension** discusses how to perform case-sensitive searches in 4th Dimensions using several different techniques, including using the HashTools component.

What Is Hashing?

Hashing is a fundamental programming technique that applies a formula or set of formulas to a block of data and returns a single number. If the same data is passed to a hashing function more than once, an identical number is always returned. Because the number produced by a hash is consistent for any input string/BLOB, the hash number can be used as a substitute for the original, much larger, data. This is, for example, a common way to produce check sum values used to help verify that documents have been correctly duplicated over a network. Since a hash is a very small and readily indexed number, operations are significantly faster than on the original data. Hashes can dramatically optimize searches by quickly excluding most, if not all, non-matching data with a single indexed search. Let's look at an example hashing function and then return to seeing how it helps with searching.

SumBytes: A Simple Example Hashing Function

If you're new to hashing, it’s easier to understand with an example. Imagine the following string:

```
abc
```

A trivial, but valid, hashing technique is to take the sum of the ASCII values of each character in the string. In the example above, the ASCII values of the three characters are 97 (a), 98 (b), and 99 (c), with a sum of 294. Instead of searching for abc, the system can search for 294 and locate all strings that might be equal to abc. The sample code below shows a slightly longer example string with a sum of 3,372:

```
C_TEXT($text)
C_LONGINT($length;$character;$sum)
```

```
$text:="abcdefghijklmnopqrstuvwxyz1234567890"
$length:=Length($text)

C_LONGINT($sum)
For ($character;1;$length)
    $sum:=$sum+(Ascii($text[[$character]]))
End for
```

We'll look at better hashing algorithms later, but even the simple summing function serves to illustrate several key features about hashing that hold true for the eight high-quality functions implemented in the sample database:

- Any particular stream of data **always** produces the same number when passed through a hashing function.
- There is no guarantee that every unique string will produce a unique hash value. For example, the sum of the bytes `abc` is the same as the sum of the bytes for `cab` or `bca`. When a hashing function creates identical numbers from distinct strings, it's called a *collision*. The high-quality string hashing functions included with this note tend to produce very few, if any, collisions.
- Large chunks of data are reduced to a single number. Even a 1MB BLOB reduces quickly to a single longint.
- The number produced by a hashing function represents the entire string/BLOB. Therefore, hash values are only useful for = and ≠ comparisons, not for >, <, >=, <=, or contains comparisons.
- Hashing is not a full-text indexing system.
- The number produced by a hashing function doesn't mean anything in itself and is not useful for sorting the original data.
- Hashing functions work on raw character/byte values and are, therefore, always case-sensitive. We'll look at a simple technique for creating case-insensitive text hashes later.

Now let's look in more detail at how hashes optimize searching for text, BLOBs, pictures, and linked documents.

Optimizing Searches with Hashing

Searching through large blocks of text or BLOBs is naturally slow in any environment. Database engines may use a variety of supplemental data, including hashes, to optimize queries on large data types. Unfortunately, the current 4th Dimension database engine includes no native text, BLOB, or picture search optimization features. Let’s consider an imaginary example database that stores unique paths in text fields. Paths are typically needed when managing stored documents, URLs, URIs, and in a variety of XML-oriented applications. (If you've never needed to store paths, just imagine any kind of textual or BLOB data.) This imaginary database already has 1,000 unique paths and needs to make sure that new paths are also unique. Take the following text as the new path in the examples below:

```
/Volumes/Echidna/Documents/Projects/Writing/Materials/Tech_Notes_and_Presentatio
ns/1_In_Progress/Hashing/HashingFunctions/4D_2004_2.app/Contents/4D\
Extensions/Spellcheck/CordialSpeller.bundle/Contents/MacOS/CordialSpeller
```

The obvious way to check for this path in the database is to do a search, as in the fragment below:

```
C_TEXT($1;$new_path)
$new_path:=$1
QUERY([Stored_Paths];[Stored_Paths]Path=$new_path)
```

The code above works correctly for short paths and fails for larger paths. The 4th Dimension search engine doesn't compare the full contents of text fields and never compares them case-sensitively.

Note *The exact details of how the* **QUERY** *command treats comparisons on text fields doesn't appear to be documented. The number of characters used in equals (=), starts with (value@) and contains (@value@) searches may differ. For safety's sake, you may not wish to rely on search strings longer than 80 characters, including the @, when using 4th Dimension's native search tools and commands.*

If you need to make a complete comparison on a text field, the native alternative is to use the **QUERY BY FORMULA** command, as in the fragment below:

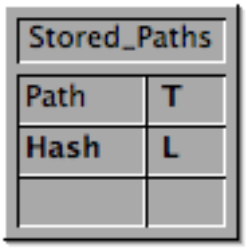
```
C_TEXT($1;$new_path)
$new_path:=$1
QUERY BY FORMULA([Stored_Paths];[Stored_Paths]Path=$new_path)
```

The search can be performed case-sensitively using a custom function, as in the fragment below:

```
QUERY BY FORMULA([Stored_Paths];CS_AlphasAreEqual([Stored_Paths]Path;$new_path))
```

Note *The* HashUtility_AlphasAreEqual *function is found in the sample database. For more details on case-sensitive searches in 4th Dimension, see technical note 05-41* **Case-Sensitive Operations in 4th Dimension**.

We'll consider case-sensitivity in more detail, later. Note that **QUERY BY FORMULA** works correctly, but it's very slow, particularly under 4D Server. In order to perform a formula-based search, 4th Dimension loads each record and examines the complete contents of the text field. Under 4D Server, the records are transferred to the client over the network for analysis, and, as a result, performance can be dragged down to unacceptable levels. This is a situation where hashing can really pay off. Imagine the table structure shown below:



The Hash field stores the hash of the path stored in the record. This hash value is easily maintained using a trigger:

```
` Trigger for [Stored_Paths]
Case of
   : (Database event=On Saving Existing Record Event )
     [Stored_Paths]Hash:=HashTools_HashText(->[Stored_Paths]Path;"SumBytes")

   :(Database event =On Saving New Record Event )
     [Stored_Paths]Hash:=HashTools_HashText(->[Stored_Paths]Path;"SumBytes")
End case
```

The sum of the bytes for the example path shown earlier is 3,372. Now, searching for a duplicate path is far easier. Instead of sequentially comparing every record in the table with **QUERY BY FORMULA**, all or nearly all non-matching records can be excluded from consideration using the hash value alone following the steps below:

1. Hash the new path as a starting point for comparisons.
2. Search the existing data using the indexed hash value. This step should exclude all, or virtually all, records that don't match.
3. If any records are found, test each one to determine if the stored path equals the new path.

The code below implements the steps listed above:

```
C_TEXT($1;$new_path)
$new_path:=$1

C_LONGINT($hash)
$hash:=HashTools_HashText (->$new_path;"SumBytes")

QUERY([Stored_Paths];[Stored_Paths]Hash=$hash)
QUERY SELECTION BY FORMULA([Stored_Paths]; CS_AlphasAreEqual ([Stored_Paths]Path;$new_path))
```

The **QUERY** statement in the method above reduces the selection to only those records that have the same hash value as the path we're testing. If the database has 10,000 records and only two of them share the target hash value ($hash), the **QUERY SELECTION BY FORMULA** command only tests two records, not the full 10,000 that are required without the hash.

Next, we should look at the subject of collisions to explain exactly why the **QUERY SELECTION BY FORMULA** command is needed. Finally, we'll look at how to replace **QUERY SELECTION BY FORMULA** with better-optimized code.

Collisions and the Need for Sequential Comparisons

It's tempting to rewrite the code listed above as follows:

```
C_TEXT($1;$new_path)
$new_path:=$1

C_LONGINT($hash)
$hash:=HashTools_HashText (->$new_path;  "SumBytes")

If (Records in selection[Stored_Paths>1)
    QUERY([Stored_Paths];[Stored_Paths]Hash=$hash)
    QUERY SELECTION BY FORMULA([Stored_Paths];  CS_AlphasAreEqual
([Stored_Paths]Path;$new_path))
End if
```

If there is only one record with a matching hash value, doesn't this indicate a matching record? It might and it might not. Different strings can produce the same hash value. For example, the sum of the bytes abc is the same as the sum of the bytes cab or bca. The **QUERY** statement listed above finds all records with a hash value of 3,372, the sum of the bytes in the string below:

```
/Volumes/Echidna/Documents/Projects/Writing/Materials/Tech_Notes_and_Presentatio
ns/1_In_Progress/Hashing/HashingFunctions/4D_2004_2.app/Contents/4D\
Extensions/Spellcheck/CordialSpeller.bundle/Contents/MacOS/CordialSpeller
```

If no records are found, the string doesn't exist. If one record is found, it may or may not match the original string. Since collisions are possible, any matching values may be false positives. Additionally, many databases need to store non-unique data. In such a case, several records may share the same hashing value through a combination of matching values (legitimate duplicates) and false positives (collisions). Therefore, each record that has a matching hash value needs to be examined exactly.

Note *Hashes can produce false positives but not false negatives. If the hash value for the source string is not found, it means the source string doesn't exist.*

Collisions Rates of Different Hashing Algorithms

Hashing doesn't eliminate the need for sequential comparison; it simply provides a tool for dramatically reducing the number of sequential comparisons required. The better is the hashing function, the fewer are the false positives and the greater are the time savings. As should already be clear, the *SumBytes* hashing algorithm isn't very good as it easily produces false positives. Any hashing function can produce false positives, but high-quality hashing functions produce far fewer. The table below summarizes the collision rates for each function when applied to various sets of case-sensitively unique values.

Data Set	Values	SDBM	BKDR	RS	AP	DJB	JS	ELF	PJW	SumBytes
Huge English Word List	118,925	2	3	4	5	14	23	327	327	3,147
Large English Word List	30,405	0	0	0	0	0	3	77	77	1,183
Large Dutch Word List	4,754	0	0	0	0	0	0	1	1	791
Paths - Roughly 1 KB long	1,000	0	0	0	0	0	0	0	0	88
Paths - Roughly 16 KB long	1,000	0	0	0	0	0	0	0	0	88
Small English Word List	977	0	0	0	0	0	0	0	0	250
Small Irish-Gaelic Word List	831	0	0	0	0	0	1	3	3	218
Sum		2	3	4	5	14	27	408	408	5,765

The columns in the table above are ranked from left-to-right in ascending frequency of collisions. Reading across, it shows that the *SDBM* function created the fewest collisions and the *SumBytes* function created the most collisions. The rows are ranked by the number of distinct values being hashed. There are several points worth making about the data in the table and about hashing functions in general:

- Even a poor hashing function, like *SumBytes*, does surprisingly well at avoiding collisions. For example, it only produces about a 2.6% collision rate when hashing nearly 120,000 distinct words and phrases (3,147 / 118,925 = 2.64 %). During a search, quickly eliminating nearly 98% of the records from consideration (115,778 values) is a dramatic optimization. And, remember, this is the worst of the hashing functions described.
- It pays to find the lowest-collision function for the data. The point of using hashes is to avoid sequential operations, particularly under 4D Server.
- All of the functions, other than *SumBytes*, perform remarkably well. In fact, the results for *SDBM*, *BKDR*, *RS*, and *AP* are so close they may be considered equivalent. Keep in mind that the test data used is artificial and not representative of your data.

Testing Your Data

If you don't have the time or inclination to test out the various hashing algorithms, you can pretty well pick any of them and be done with it as they're all quite good. *BKDR* and SDBM are generally cited as good defaults if you don't want to test your

data. If you do want to test your data, export it and import it into the sample database. For the import, prepare a file with two columns:

Data_Set_Name	Alpha 60
Sample_Text	Text

The demonstration system treats all records with the same [Sample]Data_Set_Name value as a data set. Triggers automatically build the various hashes during the import. Once your data is in the database, you can use the test screens on your data.

Note *Many of the operations in the database are substantially faster compiled. The code in this system is intended for testing and deployment in compiled mode.*

More Information About Collisions and Why You Should Test

Having just said that you don't have to test the hashing functions against your data, it's worth stating why you should consider testing. Using the data sets tested and summarized above, the rankings for the various functions are consistent. This may not be the case with all data sets. High-quality hashing functions produce "evenly distributed" output. In other words, the resulting numbers are evenly spread over the range of possible longints and don't tend to produce any one number more frequently than another. However, a perfect hashing function can only create an evenly distributed output if it's given evenly distributed inputs. In the real world, there is effectively no chance that the inputs will be even. For example, the letter "t" occurs far more often in English than the letter "z" and the letters "j", "k", "w", "x", and "y" are not native to Italian. Given the realities of letter frequencies, even a perfect hashing function is unable to produce a completely collision-free output.

Replacing QUERY SELECTION BY FORMULA

Now let's return to talking about optimizing searches, a subject also addressed in Technical Note 05-41 **Case-Sensitive Operations in 4th Dimension**. Below is the hash-based search code we presented earlier:

```
C_TEXT($1;$new_path)
$new_path:=$1

C_LONGINT($hash)
$hash:=HashTools_HashText (->$new_path;"SumBytes")

QUERY([Stored_Paths];[Stored_Paths]Hash=$hash)
QUERY SELECTION BY FORMULA([Stored_Paths];
    CS_AlphasAreEqual([Stored_Paths]Path;$new_path))
```

Unfortunately, this code may still be too slow, particularly under 4D Server, because of the **QUERY SELECTION BY FORMULA** command. The larger the selection of records returned by the hash query, the more sequential operations performed by **QUERY SELECTION BY FORMULA** will result. The selection will include any false duplicates (collisions) and matching values (valid duplicates). Fortunately, with a bit of

work, it's possible to replace **QUERY SELECTION BY FORMULA** with an optimized replacement. The steps involved in the improved code are listed below:

1. Hash the new path as a starting point for comparisons.
2. Search the existing data using the indexed hash value. This step should exclude all, or virtually all, records that don't match.
3. If any records are found, load the text values using **SELECTION TO ARRAY**.
4. Loop through the array, testing each item. If the item is equal to the target string, add the corresponding array to the final selection.

The code required to implement these steps is more complicated than calling **QUERY SELECTION BY FORMULA** but delivers enormous performance benefits. Below is a simplified version of the new code for a system using the SDBM hashing algorithm.

```
C_TEXT($1;$new_path)
$new_path:=$1

C_LONGINT($hash)
$hash:=HashTools_HashText (->$new_path;"SDBM")

QUERY([Stored_Paths];[Stored_Paths]Hash=$hash)

If (Records in selection([Stored_Paths])>0)
   ARRAY LONGINT($record_numbers_that_might_match;0)
   ARRAY TEXT($text_values_to_check;0)

   SELECTION TO ARRAY([Stored_Paths];$record_numbers_that_might_match)
   SELECTION TO ARRAY([Stored_Paths]Path;$text_values_to_check)

   ARRAY LONGINT($record_numbers_that_do_match;0)

   For ($loop_counter;1;Size of array($text_values_to_check))
     If (HashUtility_AlphasAreEqual ($new_path;$text_values_to_check{$loop_counter}))
        ` The new path matches the current element in the text array.
        ` Add the original record number the array of record numbers
        ` we'll use below to build the resulting selection.
       C_LONGINT($record_number)
       $matching_record_number:=$record_numbers_that_might_match{$loop_counter}

       APPEND TO ARRAY($record_numbers_that_do_match;$matching_record_number)
     End if
   End for

    ` The array below is either empty or has record numbers of records whose
    ` Path field = the new path we're looking for.
   CREATE SELECTION FROM ARRAY([Stored_Paths];$record_numbers_that_do_match)
End if
```

Note *See the* HashTools_FindTextByHash *method in the source code database for a more generalized implementation of the code shown above.*

The key optimization feature of the new code is the use of **SELECTION TO ARRAY** and **CREATE SELECTION FROM ARRAY**. Both of these commands are network-optimized under 4D Server. Instead of transferring entire records, **SELECTION TO ARRAY** only transfers the actual field data requested. For large records, the time savings can be substantial. But just how substantial? Let's look at some speed results.

How Much Time Can Hashing Save?

Before adopting any optimization strategy, it's worth doing a reality check on the costs and benefits of the technique. It's easy to invent artificial scenarios that suggest an optimization is useful when, in real applications, the optimization makes typical operations slower. The table below shows some results when searching various data sets under 4th Dimension. All results shown are the averaged (mean) results of repeated tests reported in milliseconds (1/1000ths of a second) shown to two decimal places:

Source	Values	AP	BKDR	DJB	ELF	JS	PJW	RS	SDBM	SumBytes	QUERY BY FORMULA
Paths - Roughly 1 KB long	1000	0.86	0.81	0.81	0.88	0.82	0.83	0.82	0.83	1.67	108.46
Paths - Roughly 16 KB long	1000	6.67	6.84	6.50	6.32	6.18	6.18	7.13	6.40	23.57	301.41
Small English Word List	977	0.43	0.38	0.44	0.43	0.37	0.40	0.45	0.41	1.11	251.89
Small Set with Duplicates	11	0.18	0.27	0.64	0.18	0.18	0.36	0.18	0.36	0.55	268.64
Tiny English Word List	100	0.37	0.27	0.27	0.22	0.30	0.31	0.27	0.33	0.57	267.02
Avg		1.70	1.71	1.73	1.60	1.57	1.62	1.77	1.67	5.49	239.48

The data sets used have relatively short, unique values. If anything, using such simple data should minimize the benefits of hashing. The larger the data and the larger the sample, the more obvious the speed improvements from hashing become. For more discussion, see the searching section of Technical Note xx-xx **Case-Sensitive Operations in 4th Dimension**.

Note *The* **QUERY BY FORMULA** *results should be nearly constant for any particular data set because the number of records inspected is always exactly the same. Real-world testing shows variation in the test results because of outside factors, such as the behavior of the operating system or 4th Dimension buffer flushing distorting values. To minimize distortion, tests were repeated and averaged for this note.*

The performance of the *AP*, *BKDR*, *DJB*, *ELF*, *JS*, *PJW*, *RS*, and *SDBM* hashes are all similar enough to be considered equivalent. Taking an average of their performances as a basis for comparison, the table below shows how much slower the *SumBytes* and **QUERY BY FORMULA** approaches are for any particular data set:

Source	Values	Main Hashes	SumBytes Hash	QUERY BY FORMULA
Paths - Roughly 1 KB long	1000	1.00	2.01	130.46
Paths - Roughly 16 KB long	1000	1.00	3.61	46.19
Small English Word List	977	1.00	2.69	609.54
Small Set with Duplicates	11	1.00	1.85	909.23
Tiny English Word List	100	1.00	1.95	912.89
Avg		1.00	2.42	521.66

Reading the overall averages shows that the SumBytes hash is about 2.4 times slower than the other hashes and that **QUERY BY FORMULA** is over 500 times slower than the better hashing functions. It's important not to get carried away by factors alone. Also consider actual clock times. In the examples above, even the SumBytes function reduces the time needed to find a unique 1 KB path to a tiny fraction of a second. Only **QUERY BY FORMULA** requires more than a second to locate the same value. After looking at the relative and actual times above, it's clear that hashing does dramatically optimize performance we need to match unique text values. But what about when we work under 4D Server, or when we look for non-unique values? Let's look at performance under 4D Server first. (Times shown in full milliseconds.)

Source	Values	AP	BKDR	DJB	ELF	JS	PJW	RS	SDBM	SumBytes	QUERY BY FORMULA
Paths - Roughly 1 KB long	1000	324	398	401	392	399	400	392	399	399	5,064
Paths - Roughly 16 KB long	1000	338	396	393	395	396	397	395	396	395	8,267
Small English Word List	977	333	392	400	395	394	398	397	393	398	9,490
Small Set with Duplicates	11	328	394	404	400	382	399	405	391	396	9,400
Tiny English Word List	100	332	396	398	394	394	396	397	391	397	9,510
Avg		331	395	399	395	393	398	397	394	397	8,346

There are several interesting differences between these results and the numbers from the same tests run under 4th Dimension using identical data:

- The actual average time needed to find a value remains under one second for the hashing functions and climbs to over eight seconds for **QUERY BY FORMULA**.
- The speed difference between SumBytes and the other functions disappears. This result is an artifact of the data sets used as we'll explain in a moment.
- The performance of **QUERY BY FORMULA** is dramatically worse in actual clock time than under 4th Dimension but produces a less dramatic relative difference.

These points are easier to see by factoring relative performance as we did earlier with the 4th Dimension results:

Source	Values	Main Hashes	SumBytes Hash	QUERY BY FORMULA
Paths - Roughly 1 KB long	1000	1.00	1.00	12.75
Paths - Roughly 16 KB long	1000	1.00	1.00	20.91
Small English Word List	977	1.00	1.01	23.99
Small Set with Duplicates	11	1.00	1.00	23.72
Tiny English Word List	100	1.00	1.00	24.07
Avg		1.00	1.00	21.09

The table above illustrates why it's so important to consider your actual running environment and the clock. Comparing the results for 4th Dimension and 4D Server, it would be easy to come away with the conclusion that **QUERY BY FORMULA** is optimized for 4D Server. Indeed, it's 521 times slower than the main hashing functions under 4th Dimension and only 21 times slower than the main hashing functions under 4D Server. So **QUERY BY FORMULA** is about 25 times better under 4D Server. Of course, this is not the case; the results for the other strategies have all increased, making the relative performance of **QUERY BY FORMULA** less dramatically poor in comparison. Looking at the clock again, we see that the hash-based queries take, on average, about 0.3-0.4 seconds to find a value and that **QUERY BY FORMULA** takes about 5-9.5 seconds to find. In real terms, **QUERY BY FORMULA** is slow.

A Few Words About Performance Tests

Please take all of the speed test values presented in this technical note as suggestive. Many factors influence test results, and none of the numbers presented should assume to hold true for your system. Likewise, the relative results should be treated as suggestive. It's reasonable to take the trends shown by these tests as valid for a wider range of systems, but the actual performance characteristics of your system are bound to differ. As already mentioned, the sample data is not typical of real-world data.

4th Dimension Versus 4D Server

One of the more interesting comparative results above is that, under 4D Server, the performance of SumBytes is no longer distinguishable from the better functions. For example, under 4th Dimension, SumBytes is, on average, 1.8-3.6 times slower than the functions that produce fewer collisions (AP, BKDR, and so on.). Why does this difference disappear under 4D Server? There is a threshold cost to the indexed longint search and an effective minimum time for 4D Client to perform the basic tasks required by the optimized search code discussed already:

1. Hash the new path as a starting point for comparisons on the client.
2. Call 4D Server to perform an indexed query using the hash value.
3. Retrieve the record numbers and text values for any matching records from 4D Server.
4. Examine each text value in the array locally and build an array of matching record numbers.
5. Send 4D Server the finial array of record numbers.
6. Wait for 4D Server to build a selection using **CREATE SELECTION FROM ARRAY**.

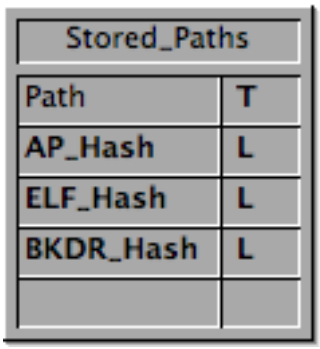
The data sets used in the tests are small and have very short values, both of which increase the visibility of the cost of an indexed search in the results. Even so, 4D Client and 4D Server manage to do all of the steps listed above in a fraction of a second but it appears that there is a minimum time required to perform the work. If this is true, testing should show that processing ten possible matching values should take less than ten times as long as processing one possible matching value. In fact, this is so. Some basic testing on data sets with short strings shows that matching 500 records out of 1,000 only takes about 350 milliseconds compared with over 16,000 when using **QUERY SELECTION BY FORMULA**. A natural and appropriate question is, then: can we find the point where it's better to use **QUERY SELECTION BY FORMULA** instead of the customized replacement? Logically, the worst case scenario where all records in the table match should favor **QUERY SELECTION BY FORMULA**. In reality, it is not so simple. **QUERY SELECTION BY FORMULA** is always worth replacing. For example, finding 1,000 records out of 1,000 using the hash-optimized approach takes 400 milliseconds versus over 16,000 milliseconds using **QUERY SELECTION BY FORMULA**.

Why Is QUERY SELECTION BY FORMULA So Slow?

A reasonable question is why are **QUERY BY FORMULA** and **QUERY SELECTION BY FORMULA** so slow compared to the custom code discussed above? One obvious answer is that **QUERY BY FORMULA** has to load entire records, not just individual fields as **SELECTION TO ARRAY**. The larger the record, the more of a penalty there is. This answer, however, doesn't explain the results cited here. The test records contain nothing but the sample text and nine hash (longint) values. The only way to know for certain why **QUERY BY FORMULA** and **QUERY SELECTION BY FORMULA** are so inexplicably slow is to review the internals of 4th Dimension itself. It's easy enough to come up with theories about why this behavior exists, but, at the end of the day, the theories don't matter. What matters is reproducible behavior and how you work with it. In this case, we need to avoid **QUERY BY FORMULA** and **QUERY SELECTION BY FORMULA** whenever possible. With this important point behind us, let's turn to some other practical ways to optimize otherwise sequential queries and address some subjects I promised earlier to cover in more detail.

Technique: Refining Query Results with Multiple Hashes

The heart of the optimization provided by hashing is the absolute reduction in the number of sequential operations required. If there are 1,000 records in a table, a native **QUERY BY FORMULA** must compare all 1,000 records. If a hash eliminates 600 records, then only 400 records need to be compared sequentially. Under 4D Server, even comparing 400 records sequentially may be slower than you wish. If a particular hash produces collisions in your data, consider maintaining two or more hashes. In this case, be sure to test the hash functions with your data to identify functions that don't produce collisions on the same input values. The imaginary path database structure below stores hashes from three different functions: AP, ELF, and BKDR:



The search code can then use more than one hash to reduce the selection, as in the example below:

```
C_TEXT($1;$new_path)
$new_path:=$1

C_LONGINT($hash)
$hash:=HashTools_HashText (->$new_path;"AP")

QUERY([Stored_Paths];[Stored_Paths]AP_Hash=$hash)

If (Records in selection([Stored_Paths])>1)`    Refine the selection with the second hash.
   $hash:= HashTools_HashText (->$new_path;"ELF")
   QUERY SELECTION([Stored_Paths];[Stored_Paths]ELF_Hash=$hash)

   If (Records in selection([Stored_Paths])>1)`    Refine the selection with the third hash.
      $hash:= HashTools_HashText (->$new_path;"BKDR")
      QUERY SELECTION([Stored_Paths];[Stored_Paths]BKDR_Hash=$hash)
   End if
End if
```

Assume **SELECTION TO ARRAY** *based code is used here.*

Before implementing multiple hashes as discussed above, do some testing in your environment. The speed benefits of avoiding scanning text sequentially for equality has to be measured against the time needed to hash the text more than once and do

additional indexed searches. Keep in mind that the longer the source data, the longer it takes to hash. Let's take a look at how expensive the various hashing algorithms are in more detail. After that, we'll look at how to reduce false-positives again a bit more.

Speed Cost of Hashing

Hashing is not free because it requires sequentially scanning the complete contents of a string, text block, BLOB, or document. Hashes are, however, designed to be as quick as possible compiled. The table below summarizes how long it takes the various functions to hash different data sets in a compiled application. Altogether, each function hashes over 150,000 values of different lengths. The tests were run on modest contemporary hardware. All times are in milliseconds.

Data Set	Values	AP	RS	ELF	BKDR	DJB	SDBM	JS	PJW	Average
Small English Word List	977	1	1	1	2	2	2	1	5	2
Small Irish-Gaelic Word List	831	1	1	2	2	1	2	1	5	2
Large Dutch Word List	4,754	9	9	16	34	13	12	10	31	17
Large English Word List	30,405	39	41	49	46	56	56	46	185	65
Paths - Roughly 1 KB long	1,000	98	129	105	103	129	137	143	156	125
Huge English Word List	118,925	270	203	238	215	325	265	251	727	312
Paths - Roughly 16 KB long	1,000	1,682	1,753	1,734	1,767	2,223	2,388	2,423	2,400	2,046
Sum	157,892	2,100	2,137	2,145	2,169	2,749	2,862	2,875	3,509	2,568

The columns are ordered by overall time per hashing function, with the shortest time being on the left. It's easy to see that the AP hash is the fastest function at 2,100 milliseconds (2.1 seconds) and the PJW function is the slowest at 3,509 milliseconds (3.5 seconds.) Given these results, it hardly matters which hashing function you choose. Even on modest equipment, the overall speed difference between the fastest and slowest functions is less than one and half seconds when applied to over 150,000 values. Hashing is effectively free in many real-world situations.

The rows in the table above are ordered by overall time per data set, with the shortest times being on top and the slowest on the bottom. Reading from top-to-bottom, you can easily see which data sets took longer to process. Notice that the overall speed doesn't depend on the raw number of values. Hashing is a purely sequential function, requiring the code to examine each byte in a string, BLOB, or document from start to finish. Consequently, the overall time is based on the sum of the lengths of all of the values. In the data sets analyzed, the word lists consist of relatively short phrases and the paths are considerably longer. Consequently, nearly 120,000 words and phrases can be hashed more quickly than 1,000 16 KB paths. Restated simply, it takes longer to hash longer data.

Note *The test results shown above are for values stored as text. The results would be quicker for identical data stored as BLOBs. See technical note xx-xx* **Scanning Text and BLOBs Efficiently** *for detailed information and sample code. Be sure to read this note if you plan to access text through pointers.*

Technique: Building Longer Hash Keys

If you are obtaining too many false positives from a single hash, there are a few ways to improve your results. As mentioned earlier, you can create multiple hashes. However, instead of storing these values in distinct fields or records, you can combine them into a single string field. Imagine, for example, code like the following:

```
C_LONGINT($ap_hash)
C_LONGINT   ($elf_hash)
C_LONGINT   ($bkdr_hash)
$ap_hash:=HashTools_HashText (->[Element_Map]XML_Path;"AP")
$elf_hash:=HashTools_HashText (->[Element_Map]XML_Path;"ELF")
$bkdr_hash:=HashTools_HashText (->[Element_Map]XML_Path;"BKDR")

` Convert the three hashes to a single string.
  ` Put a dash between hashes, like so:
  `  24554-89099-34566
C_STRING(20;$longHashKey_s)
$longHashKey_s:=""
$longHashKey_s:=$longHashKey_s+String($ap_hash)
$longHashKey_s:=$longHashKey_s+"-"
$longHashKey_s:=$longHashKey_s+String($elf_hash)
$longHashKey_s:=$longHashKey_s+"-"
$longHashKey_s:=$longHashKey_s+String($bkdr_hash)

[Element_Map]Long_Hash_Key:=$longHashKey_s
```

This idea is very similar to the idea of storing multiple hashes, already discussed. However, using a single string field is well worth considering as it can be quicker for 4th Dimension to search on one string index than an three longint indexes.

If you do decide to build a long string hash key, you can format the string in a variety of ways, depending on your requirements. For example, including forcing the numeric conversion to use a specific number of digits, encode the results as hex, or encode them in base 64. As long as the final string is 80 characters or less, it can be stored and indexed in a alpha field.

Creating multiple hashes is not the only way to create a long hash key with the HashTools component. For example, instead of hashing 32,000 characters of text to get a single longint, hash the text in four blocks of 8,000 characters and then combine the four hashes into a single long key. If you need to generate longer hash-based keys regularly, consider rewriting the internal hashing routines to accept two new parameters: starting position and maximum length of data to hash. If you do go this route, use the private *HashUtility_SanityCheckFunction* to verify the text and BLOB hashing routines are functioning correctly. Alternatively, you can pull chunks of data out of the source value and pass them in turn to the existing code. The later approach, of course, is less runtime time and memory efficient than rewriting the source code. On the other hand, rewriting the source code is less developer time and stress efficient!

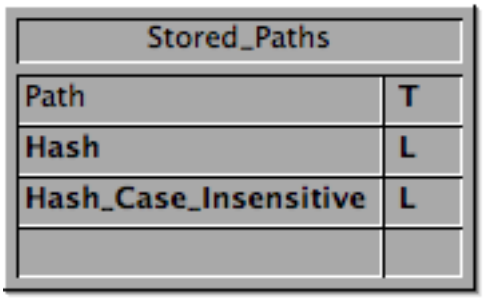
Technique: Hashing Only Part of the Data

If you are using hashes to optimize searching for large objects, such as BLOBs, documents, and pictures, consider not hashing all of the data. For example, instead of hashing a full document, hash only the first 8,000 bytes. The advantage of this approach is that it reduces the time needed to produce each hash, if the objects are large. Also, it makes the time required easier to predict. If you know how long it takes to hash 8,000 bytes and you need to hash 1,000 documents, it's easy to estimate the maximum time required. If you have to hash full documents, the time varies depending on the combined lengths of all of the documents.

It's worth stating explicitly why partial hashing doesn't eliminate any of the benefits of hashing. When you match two blocks of data because of a shared hash, all that means is that the two blocks of data *might* be identical. Then again, these objects may be different but, by chance, produce the same hash. A final, byte-by-byte comparison is still required to confirm if two objects are, in fact, the same. If you can reduce the portion of an object hashed without increasing the number of false positives unacceptably, partial hashing can save processing time.

Technique: Case-Insensitive Comparisons

Hashing functions work on raw byte values and are, therefore, always and necessarily case-sensitive, contrarily to how 4th Dimension's native searches behave. If you need or prefer to search case-insensitively, it's easily done by normalizing the case of the source text before building the hash. Our imaginary path database structure can be extended to hold case-sensitive and case-insensitive hash values:



The trigger code can be updated to maintain both hashes:

```
` Trigger for [Stored_Paths]
Case of
: (Database event=On Saving Existing Record Event )
    [Stored_Paths]Hash:=HashTools_HashText (->[Stored_Paths]Path;"SumBytes")
    C_TEXT($uppercase_text)
    $uppercase_text:= Uppercase([Stored_Paths]Path)
    [Stored_Paths]Hash_Case_Insensitive:= HashTools_HashText (->$uppercase_text;"SumBytes")
```

```
   : (Database event=On Saving New Record Event )
     [Stored_Paths]Hash:=HashTools_HashText (->[Stored_Paths]Path;"SumBytes")
     C_TEXT($uppercase_text)
     $uppercase_text:= Uppercase([Stored_Paths]Path)
     [Stored_Paths]Hash_Case_Insensitive:= HashTools_HashText (->$uppercase_text;"SumBytes")
End case
```

The key lines of code are repeated below:

```
C_TEXT($uppercase_text)
$uppercase_text:= Uppercase([Stored_Paths]Path)
[Stored_Paths]Hash_Case_Insensitive:= HashTools_HashText (->$uppercase_text;"SumBytes")
```

The source text is converted to all uppercase before the hash is built. **Uppercase** was chosen arbitrarily, you could use **Lowercase** just as well. A simple version of search code using this hash is shown below:'

```
C_TEXT($1)
C_TEXT($new_path_uppercase)
$new_path_uppercase:= Uppercase ($1)

C_LONGINT($hash)
$hash:=HashTools_HashText (->$new_path_uppercase;"SumBytes")

QUERY([Stored_Paths];[Stored_Paths]Hash_Case_Insensitive=$hash)
QUERY BY FORMULA([Stored_Paths]; [Stored_Paths]Path=$new_path)
```

Technique: Searching for BLOBs, Pictures, and Documents

The original impetus for implementing hash functions in 4th Dimension was to find a way to quickly search through a large collection of XML paths case-sensitively. Hashing is a great solution for that problem but, as it turns out, hashing is also an excellent solution for a variety of other unrelated search problems. The *HashTools_HashDocument* command can automatically search for strings, text, pictures, and BLOBs based on stored hashes. The *HashTools_HashDocument* command hashes documents and serves as a basis for you to write hash-optimized document query and retrieval code. Below are some example scenarios that illustrate some different uses of hashing:

- The health and safety department for a large international construction company needs to distribute the latest version of hundreds of documents to offices in over 20 countries. As part of the synchronization process, the hash of each document transmitted is also transferred as a check sum. When each document is received, it's hashed using *HashTools_HashDocument* and the value compared with the original check sum. If the values don't agree, the document is retransmitted.
- A talent agency recently started a new Web-based system that allows models and actors to submit their résumé's and photographs. Unfortunately, some hopefuls are submitting their photograph more than once using slightly different names and addresses. To detect these duplicates, the system hashes each photograph

submitted over the Web. When a new photograph is submitted, the system calls *HashTools_HashPicture* to hash the photograph and then calls *HashTools_FindByHash* to find any duplicates.

- The legal department for a major Hong Kong merchant bank maintains a centralized repository of each version of each contract they negotiate. Internally, the 4D Server system stores a path to each document and a hash of its contents. When an associate submits a new document to the system, the database checks if it's a duplicate. Instead of comparing the document directly with tens of thousands of other documents, the system hashes the document with *HashTools_HashDocument*, searches for documents with matching hashes, and then compares these documents individually with the submission.

Tip: Constructing "Not Equals" Searches Correctly

As emphasized throughout this discussion, hashes optimize searches by reducing the number of values that need to be inspected sequentially. The fewer sequential operations, the better. All of the examples have been geared towards finding equal values. If you want to find values that are not equal, think ahead how you'll build the selection. Imagine you have 10,000 records and 9,999 or them are not equal to a particular value and one is equal. It's far quicker to find the 1 that is equal (1 sequential operation) and reverse the selection through set operations than to find all of the not equals values (9,999 sequential operations.)

A Few Words for People Who Don't Like Math

This note has not addressed the code for the various hashing functions implemented in the source code database. Apart from SumBytes, the hashing functions in the database are translations of code available in Java, C, C++, Pascal, and Object Pascal at:

http://www.partow.net/programming/hashfunctions/index.html

The 4th Dimension code is written to look very much like the original code. As an example, below is the source of the ELF hash in C followed by its translation into 4th Dimension code:

```
unsigned int ELFHash(char* str, unsigned int len)
{
   unsigned int hash = 0;
   unsigned int x    = 0;
   unsigned int i    = 0;

   for(i = 0; i < len; str++, i++)
   {
      hash = (hash << 4) + (*str);
      if((x = hash & 0xF0000000L) != 0)
      {
         hash ^= (x >> 24);
         hash &= ~x;
      }
   }

   return (hash & 0x7FFFFFFF);
}
/* End Of ELF Hash Function */
```

```
C_LONGINT($0;$hash)
C_TEXT($1;$str)

$str:=$1

$hash:=0

C_LONGINT($x)
C_LONGINT($len)
$x:=0
$len:=Length($str)

C_LONGINT($i)
For ($i;1;$len) ` Note: 4D numbers strings from 1, not 0.
   $hash:=($hash << 4)+(Ascii($str$i))
   $x:=$hash & 0xF0000000
   If ($x#0)
     $hash:=$hash ^| ($x >> 24)
     $hash:=$hash & ($x ^| -1)
   End if
End for

$0:=$hash
```

Unless you use bit-wise operators regularly and really like math, the code above is almost certainly incomprehensible. If this is the case for you, don't worry about it. For those who are mathematically minded, hashes are an intriguing subject. For the rest of us, we can say they work through magic and leave it at that.

A Note About MD5

Many people are familiar with, or have at least heard of the MD5 hashing algorithm. MD5 is a cryptographic hash, not just a simple string hash. Cryptographic hashing functions produce results that are very difficult to reverse, so they are good candidates for sensitive data such as passwords. Producing secure hashes, however, is more complex than producing simple hashes. The hashing functions implemented in the source code database and packaged into the HashTools component each require only a few lines of code in C while the MD5 algorithm requires several hundred lines in C. Unless you need a cryptographic hash, MD5 is a poor choice as a hashing algorithm as it's slower to run than non-cryptographic hashes.

More Details About the Source Code

The sample database includes a suite of tools for testing speed and collision rates for different hashing functions applied to different data sets. Apart from testing and various support utilities, the heart of the system is found in the hashing functions. Listed below are the main routines in the database. The methods visible in the HashTools component are highlighted in bold.

Method	Scope	Description
Compiler_HashTools	**Protected**	Includes all compiler declarations needed by component methods.
HashBlob_AP	Private	Returns AP hash of a BLOB.
HashBlob_BKDR	Private	Returns BKDR hash of a BLOB.
HashBlob_DJB	Private	Returns DJB hash of a BLOB.
HashBlob_ELF	Private	Returns ELF hash of a BLOB.
HashBlob_JS	Private	Returns JS hash of a BLOB.
HashBlob_PJW	Private	Returns PJW hash of a BLOB.
HashBlob_RS	Private	Returns RS hash of a BLOB.
HashBlob_SDBM	Private	Returns SDBM hash of a BLOB.
HashBlob_SumBytes	Private	Returns SumBytes hash of a BLOB.
HashDocument_AP	Private	Returns AP hash of a document.
HashDocument_BKDR	Private	Returns BKDR hash of a document.
HashDocument_DJB	Private	Returns DJB hash of a document.
HashDocument_ELF	Private	Returns ELF hash of a document.
HashDocument_JS	Private	Returns JS hash of a document.
HashDocument_PJW	Private	Returns PJW hash of a document.
HashDocument_RS	Private	Returns RS hash of a document.
HashDocument_SDBM	Private	Returns SDBM hash of a document.
HashDocument_SumBytes	Private	Returns SumBytes hash of a document.
HashText_AP	Private	Returns AP hash of a text/string.
HashText_BKDR	Private	Returns BKDR hash of a text/string.
HashText_DJB	Private	Returns DJB hash of a text/string.
HashText_ELF	Private	Returns ELF hash of a text/string.
HashText_JS	Private	Returns JS hash of a text/string.
HashText_PJW	Private	Returns PJW hash of a text/string.

HashText_RS	Private	Returns RS hash of a text/string.
HashText_SDBM	Private	Returns SDBM hash of a text/string.
HashText_SumBytes	Private	Returns SumBytes hash of a text/string.
HashTools Read Me Private	Private	Documents all code.
HashTools Read Me Public	**Public**	Documents public and protected code.
HashTools_FindBlobByHash	Private	Finds BLOB by stored hash.
HashTools_FindByHash	**Protected**	Finds alpha, text, BLOB, or picture by stored hash.
HashTools_FindPictureByHash	Private	Finds picture by stored hash.
HashTools_FindTextByHash	Private	Finds alpha or text by stored hash.
HashTools_GetErrorText	**Protected**	Returns error string associated with a HashTools error code.
HashTools_GetHashTypeNames	**Protected**	Returns array of supported hash types.
HashTools_GetLastErrorCode	**Protected**	Returns last recorded error code, if any.
HashTools_GetLastErrorLocation	**Protected**	Returns name of last method to record an error, if any.
HashTools_HashBlob	**Protected**	Hashes a BLOB.
HashTools_HashDocument	**Protected**	Hashes an open document.
HashTools_HashPicture	**Protected**	Hashes a picture.
HashTools_HashText	**Protected**	Hashes a string or block of text.
HashTools_InitializeErrorText	Private	Loads static error strings.
HashTools_InitializeTypeNames	Private	Loads static list of hash types.
HashTools_InstallErrorSystem	Private	Installs HashTools error handler and sets current error location.
HashTools_RestoreErrorHandler	Private	Restores original error handler, if any.
HashTools_SetError	Private	Sets error code.
HashUtility_AlphasAreEqual	Private	Tests if two alphas are identical case-sensitively.
HashUtility_DocRefIsValid	Private	Tests if a document reference refers to an open document.
HashUtility_PicturesAreEqual	Private	Tests if two pictures are identical.
HashUtility_SanityCheckFunction	Private	Sanity checks the low-level hashing routines by passing in a fixed input with known outputs.
HashUtility_TypeNameIsValid	Private	Tests if a hash type name is recognized.

Summary

Hashing is a reliable and efficient technique for optimizing locating strings, text, BLOBs, pictures, and documents. Using the materials presented in this note and the code provided in the accompanying database, you should be able to implement high-speed "equals" and "not equals" searches on strings, text, pictures, BLOBs and documents. Additionally, code and background materials explain why and how to replace **QUERY BY FORMULA** and **QUERY SELECTION BY FORMULA** when performing sequential searches.