Optimizing Searches with Hashes

By David Adams

TN 05-44

Overview

4th Dimension のデータベースエンジンは、通常の動作において、大文字と小文字を区別する検索、テキストフィールドの全文検索、BLOB、ピクチャ、ドキュメントの検索をサポートしていません。ハッシュの使用は、こうした検索を特定の状況下で最適化することのできる良く知られたテクニックです。このテクニカルノートでは、ハッシュの概要、検索、ドキュメント比較、チェックサムの検証など、特定の問題に対してハッシュを応用する方法について論じています。このテクニカルノートを読むことによって、下記の事柄についての理解を深めることができます。

- ハッシュが動作する仕組み
- ハッシュを使用できる場合
- ハッシュアルゴリズムの選定方法
- QUERY BY FORMULA の使用を避けるために必要な最適化について

サンプルコードは、文字列、テキスト、BLOB、ピクチャ、ドキュメントのハッシュをサポートしています。加えて、文字列、テキスト、BLOB、ピクチャフィールドに対して使用することのできるハッシュ検索メソッドも提供されています。サンプルデータベースの内容について取り上げた後、ハッシュの理解に必要な基本的背景知識について考察することにしましょう。

Note: 今回、紹介しているハッシュのテクニックは、文字列、テキスト、BLOB、ピクチャ、ドキュメントに対応していますが、物事を簡単にするため、通常は文字列、テキスト、BLOB を対象とした説明をしています。

About the Samples

Packaging

サンプルコードは、HashTools というコンポーネント、サンプルデータベース、ソースコードデータベースという形で提供されています。コンポーネントはテクニカルノート 05-43 The HashTools Component で紹介したのと同一のものです。サンプルデータベースとソースコードデータベースは、機能およびインタフェースの面では同一ですが、HashTools の扱いが異なっており、一方では HashTools がコンポーネントとしてインストールされているのに対し、他方では HashTools のコードをトレースすることができるようになっています。

Supported Hashing Algorithms

HashTools のコードでは、8 種類の良質なハッシュアルゴリズムを採用しています。AP、BKDR、DJB、ELF、JS、PJW、RS および SDBM のオリジナルコードは下記のサイトにあります。

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

加えて、機能的には劣りますが、ハッシュの原理を理解することのできるものとして SumBytes というものも含まれています。以降で取り上げているハッシュおよびハッシュ検索メソッドは、9 種類のハッシュアルゴリズムをすべて使用することができます。

Data Sets

後述するように、最適のハッシュアルゴリズムは、ハッシュされようとしているデータによって決まります。この点を例証するために、サンプルデータベースでは、様々なデータセットを比較できるようになっています。自前のデータをインポートしてテストを実行する場合、以下の点について結果を比較することができます。

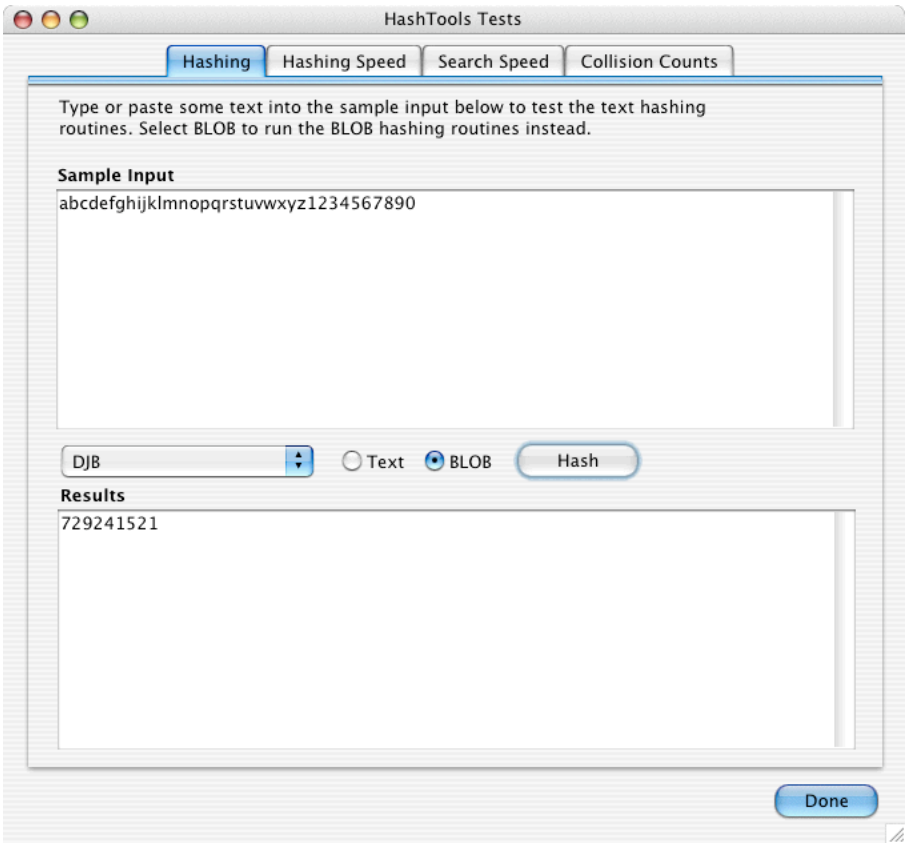
- データをハッシュするために要する時間
- 衝突(異なるデータから同一のハッシュが生まれること)の起こる頻度
- ハッシュを使用してクエリした場合の速度向上

上記の点についてはテクニカルノートの後半で詳しく取り上げます。

The Demonstration Test Screens

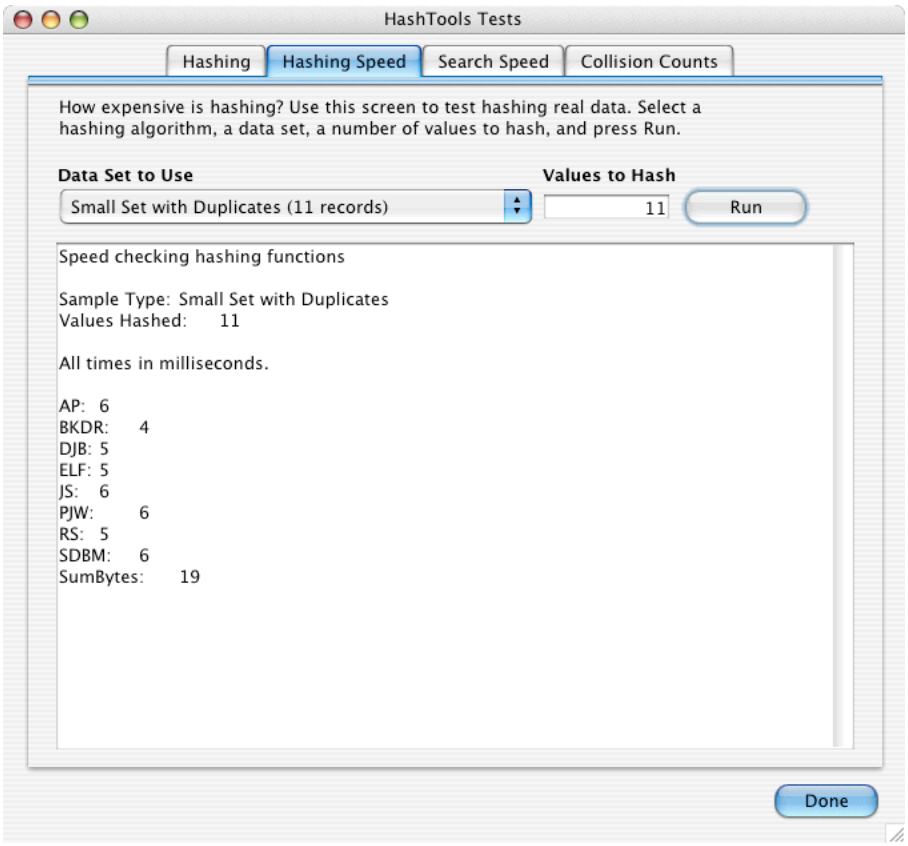
サンプルデータベースは 4 画面で構成されています。それぞれの説明は次のとおりです。

Hashing Tests



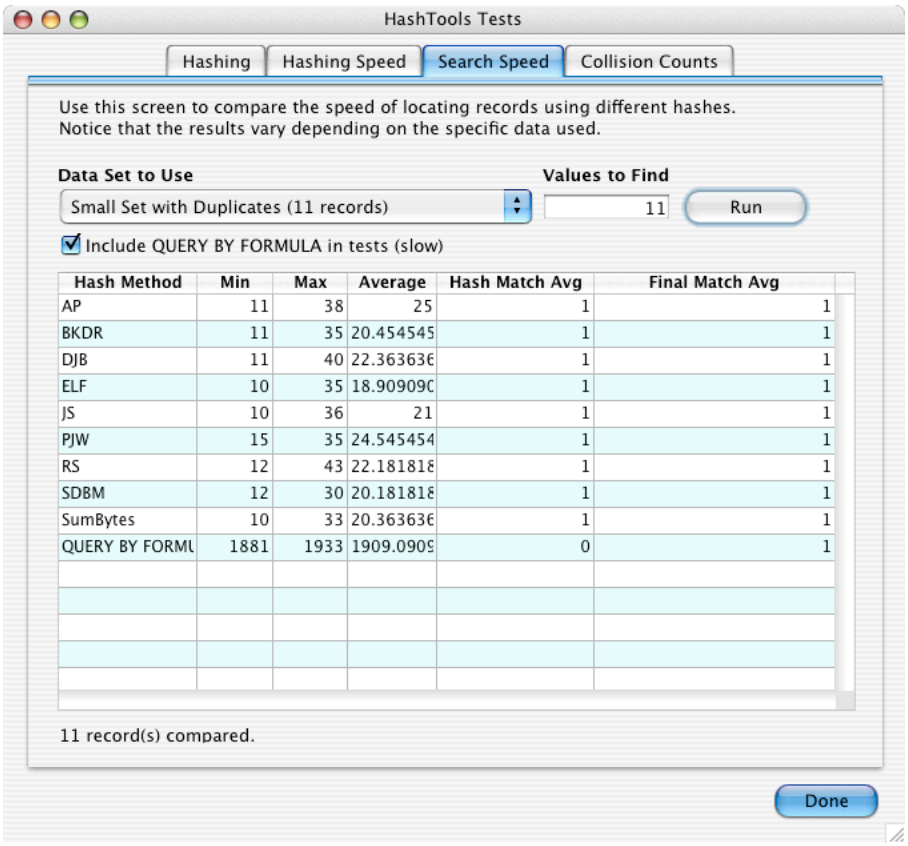
Hashing の画面では、テキストをタイプ入力またはペーストし、選択したアルゴリズムを使用してテキストあるいは BLOB としてハッシュすることができます。この画面は、ハッシュの結果を確認するため、あるいはソースコードデータベースで開いてハッシュの実行方法を知るために利用することができます。

Hashing Speed Tests



Hashing Speed の画面では、異なるハッシュアルゴリズムを使用した際に、データをハッシュまでに要する時間を比較することができます。提供されているサンプルデータセットの中から選択するか、あるいはもっと実際的な比較をしたいのであれば、データをインポートして結果を比較することができます。パフォーマンスの正確な情報を得るため、この画面はコンパイルモードで実行してください。

Hashing Search Speed Tests



Search Speed の画面では、特定のデータセットに対して様々なハッシュ検索を実行した場合の速度の差を比較することができます。使用にあたっては、以下の点に留意してください。

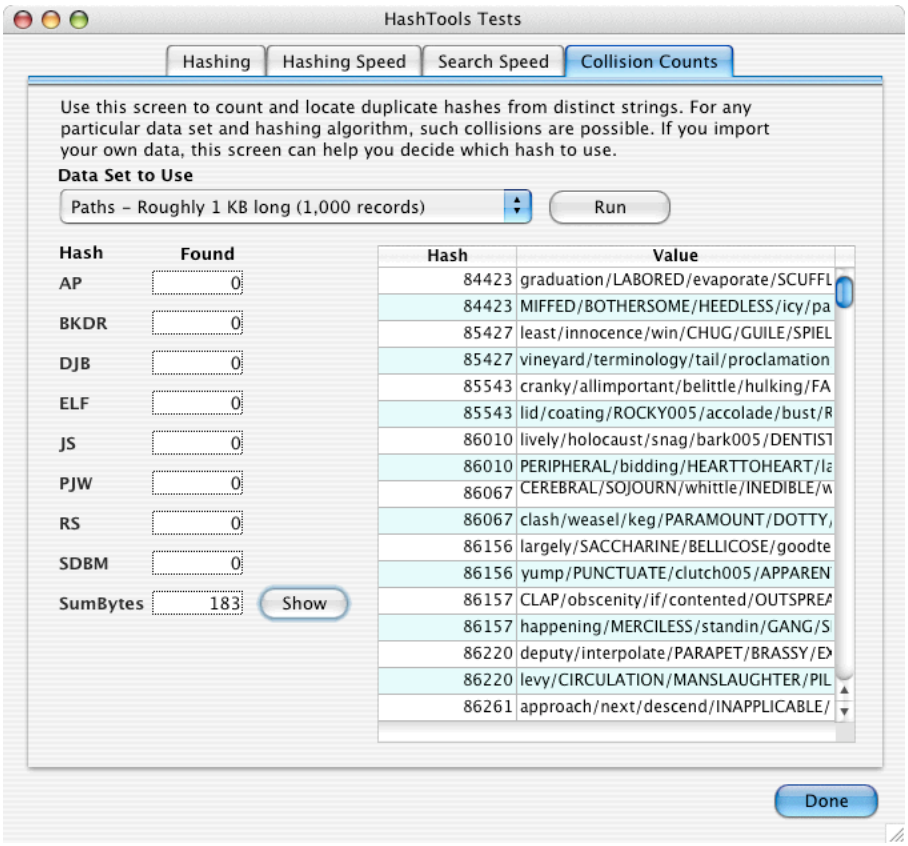
- 4th Dimension と 4D Server では、結果が大きく異なる場合があります。テストは、運用環境で実施してください。

- パフォーマンスの正確な情報を得るためにコンパイルモードで実行してください。

- QUERY BY FORMULA は桁違いに遅く、4D Server では輪をかけて時間がかかります。テストに含めたくない場合は、チェックボックスを外してください。

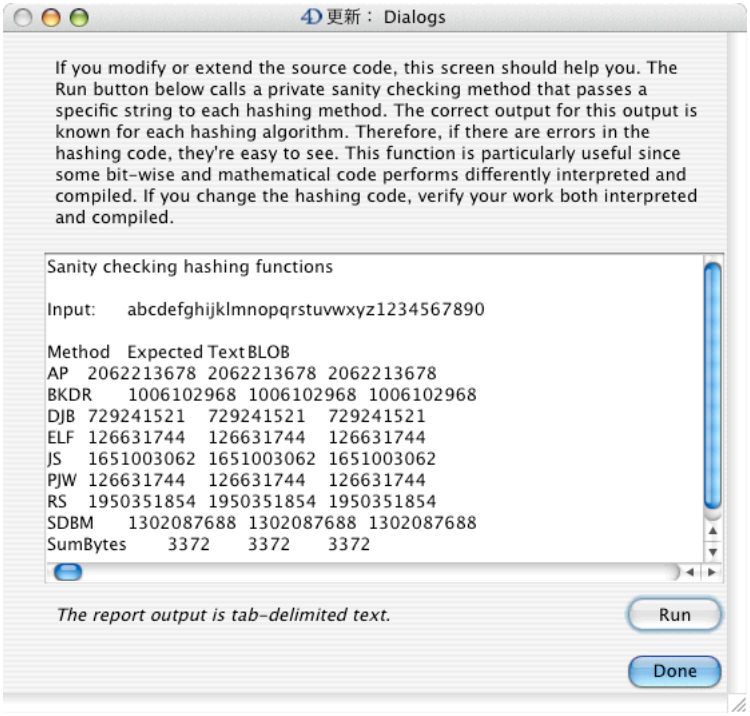
- 実用的なテストを実行するために自前のデータをインポートしてください。

Hashing Collision Tests



Collision Counts の画面では、衝突(異なるデータから同一のハッシュが生まれること)の起こる頻度を比較することができます。衝突のないハッシュはありません。オプションとして発生した衝突の値を表示することもできます。実用的なテストを実行するためには、自前のデータをインポートしてください。

Source Code Only: Sanity Checking Screen



ソースコードデータベースには、[Dialogs];"HashTest_SanityCheck"というフォームがあり、*HashUtility_SanityCheckFunction* という Private 属性のメソッドを実行することができます。このルーチンではテキストと BLOB の両方において正しい結果の分かっているデータを使用し、ハッシュ結果に関するレポートを出力します。ハッシュアルゴリズムに異常があれば、すぐに検出することができます。この画面は、ハッシュのソースコードを書き換える場合を除けば、特に用途はありません。

Related Technical Notes

サンプルデータベースのコードにも活かされているパフォーマンス面の工夫、およびハッシュの応用については、他の関連テクニカルノートで論じられています。

- テクニカルノート 05-43 The HashTools Component では、HashTools コンポーネントの Public および Private 属性メソッドについて解説し、コンポーネントのフルソースコードを紹介しています。
- テクニカルノート 05-42 Scanning Text and BLOBS Efficiently では、4th Dimension において速度と使用メモリの両面でもっとも効果的なテキストおよび BLOB の解析方法について論じています。
- テクニカルノート 05-41 Case-Sensitive Operations in 4th Dimension では、様々な手法を用いて 4th Dimension で大文字と小文字を区別する方法について論じています。その中には HashTools コンポーネントを使用するものも含まれています。

What Is Hashing?

ハッシュとは、あるデータの固まりに対してひとつまたは複数の式を適用し、結果として単一の値を受け取る基本的なプログラミングテクニックのことです。同じデータをハッシュした場合、結果として返される値は必ず決まったものになります。ハッシュは特定の文字列/BLOB に対して一定の値を返すため、サイズの大きな元のデータを表わすものとして利用することができます。しばしばネットワーク経由でドキュメントが正しくコピーされたことを検証するために使用されるチェックサムも、ハッシュの一種です。ハッシュはデータサイズが小さく、インデックスを付けることができるため、元のデータと比較すると非常に速く検索することができます。ハッシュを使用すれば、一度のインデックス検索で、条件に合わないレコードはほとんどすべて除外することができます。まず、簡単なハッシュの実例をみた後に、実際の検索に応用する方法について考慮してみましょう。

SumBytes: A Simple Example Hashing Function

これまでハッシュに接したことがないのであれば、実際の例で考えてみましょう。次のような文字列があったとします。

abc

文字列の各アスキーコードを合算することは、原始的ではありますが、立派なハッシュテクニックのひとつです。上の例では、アスキーコードの値は 97(a)+98(b)+99(c)=294 になります。abc の代わりにハッシュの 294 を検索すれば、abc を含む候補を絞り込むことができます。以下のコードは、合算が 3372 になる例です。

```
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
```

より高度なハッシュアルゴリズムについては後で取り上げますが、上記の例でも、すべてのハッシュアルゴリズムの共通する基本的な特徴が幾つかみられます。

- ハッシュ関数で処理された特定のデータストリームは、必ず一定の数値を返す。
- あらゆるデータがユニークなハッシュを生成するとは限らない。例えば、abc のアスキーコードの合算は、cab や bca の合算と同一である。異なるデータストリームに対して重複するハッシュが返されることを衝突(*collision*)と呼ぶ。本ノートで紹介しているハッシュは、いずれも滅多に衝突を起こさない高度なアルゴリズムばかりである。
- 非常に大きなデータも、ハッシュされると単一の数値になる。1MB の BLOB であろうと、ハッシュすれば倍長整数ひとつになる。
- ハッシュの結果として生成される数値は、文字列/BLOB 全体を代表するものなので、=あるいは≠の比較には適しているが、>、<、>=、<=の比較には適していない。
- ハッシュはテキスト全体にインデックスをつけるものではない。
- ハッシュの結果として生成される数値には何の意味もなく、並び替えには使用できない。
- ハッシュはバイト単位でデータを扱うため、常に大文字と小文字を区別するものである。文字種を区別しないハッシュについては後述する。

ハッシュの使用によって、どのようにテキスト、BLOB、ピクチャ、ドキュメントパスの検索を最適化できるのかを詳しくみてみましょう。

Optimizing Searches with Hashing

テキストの長文や大きな BLOB を検索することは、どのような方法をとっても時間がかかるものです。サイズの大きなデータを検索する際にかかる時間を短縮するために、ハッシュを含め、補足的な情報を用いるデータベースエンジンもあります。残念ながら、4th Dimension には、テキスト/BLOB/ピクチャの検索を高速化するため仕組みが特に組み込まれていません。例えば、ユニークなパスをデータベースのテキストフィールドに記録したいとしましょう。保存されたドキュメント、URL、URI、XML を取り扱うアプリケーションでは、しばしばそのようにパスを管理する必要が生じます。データベースには、すでに 10000 件のユニークなパスが保存されており、新たに追加されようとしているパスも他と重複しないものであることを確認しなくてはなりません。ここで下記のテキストを新しいパスとして追加しようとしましょう。

/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

重複防止策として、もっとも明快なのが、下記のような検索を実行するコードです。

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

上記のコードは短いパスでは正しく動作しますが、長いパスになると誤った結果を返します。4th Dimension のクエリはテキストの全文を比較するわけではなく、大文字と小文字も区別しません。

Note QUERY コマンドのテキストフィールドに対する動作は、明確にドキュメント化されていないようです。比較の対象となる文字数は、一致(=)、前方一致(value@)、含む(@value@)検索では、幾らか異なる可能性があります。問題を避けるため、@を含めて 80 バイトを超えるテキストを検索する場合は、4th Dimension ネイティブコマンドの使用を避けたほうが賢明です。

テキストフィールドの全文検索を 4th Dimension のコマンドで実行する代替案としては、下記のコードのように QUERY BY FORMULA コマンドを使用するものが考えられます。

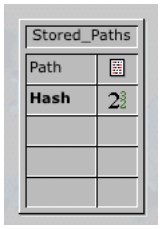
```
C_TEXT($1;$new_path)
$new_path:=$1
QUERY BY FORMULA([Stored_Paths];[Stored_Paths]Path=$new_path)
```

カスタム関数を使用すれば、これに大文字と小文字を区別する処理を付加することができます。

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

Note *HashUtility_AlphasAreEqual* メソッドは、サンプルデータベースに含まれています。4th Dimension で大文字と小文字を区別する方法に関する詳しい考察については、テクニカルノート 05-41　CaseSensitive Operations in 4th Dimension をご覧ください。

大文字と小文字の区別に関しては後でもう一度取り上げます。QUERY BY FORMULA は正確に動作しますが、非常に処理が遅く、特に 4D Server では時間がかかる点に留意してください。フォーミュラによるクエリを実行するため、4th Dimension は毎回、レコードをロードしてテキストフィールドの内容を調べます。4D Server の場合、レコードは解析のためネットワーク越しにクライアントへ転送されるため、パフォーマンスが著しく低下します。このようなシチュエーションにおいて、ハッシュは絶大な効果を発揮します。以下のようなテーブルストラクチャについて考えてみましょう。



Hash フィールドには、レコードに登録された Path のハッシュが保存されています。Hash の値は、トリガを使用することで、簡単に更新することができます。

```
` 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
```

先に挙げたパスの例では、バイトの合算が 3372 になります。このようにすれば、重複レコードの検出は非常に簡単です。QUERY BY FORMULA を使用してレコードをひとつひとつ比較する代わりに、下記のような手順を踏むことで条件に合わないレコードはほとんどあるいはすべて、除外することができるからです。

1. 比較を開始する前に新しいパスをハッシュする。
2. 既存のハッシュレコードに対してインデックス検索を実行する。結果として、条件に合わないレコードはすべて、あるいは実質的にすべて除外される。
3. 残されたレコードがあれば、それらを新しいパスと比較する。

下記のコードは、この方法を具体的に実践したものです。

```
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))
```

上記のコードでは、QUERY の結果、ハッシュの値が新しいパスのものと一致するレコードだけが残されます。データベースに 10000 件のレコードが登録されており、ハッシュ($hash)の値が問題のパスと重複するレコードが 2 件しかないのであれば、QUERY SELECTION BY FORMULA コマンドが調べるレコード数は、ハッシュを使用しない場合の 10000 件ではなく、わずか 2 件のみということになります。

最後に QUERY SELECTION BY FORMULA が必要な理由を説明するために、衝突という問題について論じ、その後、QUERY SELECTION BY FORMULA のより優れた代替案について考慮したいと思います。

Collisions and the Need for Sequential Comparisons

前述のコードを下記のように短縮して書き換えたくなるかもしれません。

```
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
```

ハッシュの値が一致するレコードがあれば、それだけで重複レコードの存在が確認できたのかといえば、必ずしもそうとは限りません。異なる値が、まったく同じハッシュを生み出す可能性があるためです。例えば、abc のバイトデータの合計値は、cab および bca と同じです。上記のコードではハッシュが下記のパスのアスキーデータを合算した値である 3372 と一致するレコードを検索しています。

/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

レコードが見つかれなければ、探しているデータは存在しません。一致するレコードがひとつでも見つかった場合、そのデータは元の値と一致するかもしれず、一致しないかもしれません。ハッシュの値は衝突することがあるので、結果は誤肯定(false positive)の可能性があります。もし、データベースがデータの重複を認めているのであれば、有効な値(legitimate duplicates)と衝突(false positive)の両方の理由から複数のレコードが同じハッシュを持つ場合があります。したがって、ハッシュが一致するレコードは、実際に内容を調べなくてはなりません。

Note ハッシュは誤肯定(false positive)を生むことがありますが、誤拒否(false negatives)を生むことはありません。元のデータに対応するハッシュが検索でみつからない場合、そのデータは存在しません。

Collisions Rates of Different Hashing Algorithms

ハッシュによって全文検索の必要性がなくなるわけではありません。むしろ、全文検索の対象となるレコード数を劇的に減らすことのできる道具として利用価値があります。ハッシュが優れているほど、衝突による誤肯定が少なく、処理時間の短縮が達成できます。すでにみてきたように、*SumBytes* アルゴリズムは衝突を起こす割合が高く、あまり優れたアルゴリズムであるとはいえません。どんなハッシュ機能も衝突の可能性は避けられませんが、それでも高度なハッシュになるとその割合は非常に低くなります。以下の表では、大文字と小文字を区別しなければならない様々なテキストデータに対し、各アルゴリズムがどれほどの頻度で衝突を起こすのかをまとめてみたものです。

Data Set	Values	SDBM	BKDR	RS	AP	DJB	JS	ELF	PJW	Sum-
英単語約 10 万件	118925	2	3	4	5	14	23	327	327	3147
英単語約 3 万件	30405	0	0	0	0	0	3	77	77	1183
蘭語約 4 千件	4754	0	0	0	0	0	0	1	1	791
パス約 1KB 長	1000	0	0	0	0	0	0	0	0	88
パス約 16KB 長	1000	0	0	0	0	0	0	0	0	88
英単語約 9 百件	977	0	0	0	0	0	0	0	0	250
アイルランド語	831	0	0	0	0	0	1	3	3	218
衝突数		2	3	4	5	14	27	408	408	5765

上の表では、総合的にみて衝突数の少ない順に、左からアルゴリズムが並べられており、もっとも衝突が少ないのは SDBM で、もっとも衝突が多いのは SumBytes であることが分かります。表の各行は、ハッシュされるデータの数が多い順に上から並べられています。この表からも読み取れるように、ハッシュについては一般に次のように述べることができます。

- SumBytes のような原始的なハッシュであっても、衝突は案外、起きないものである。たとえば、120000 件程度のユニークな語句をハッシュした場合、2.6%程度の割合でしか同じハッシュは生まれない(3147/118925=2.64%)。検索を実行する場合、対象となるセレクションから 98%のレコード(115778 件)が除外できるのであれば、相当な最適化が実現するわけで、しかもこれはもっとも単純ハッシュを使用した場合の値に過ぎない。
- 特定のデータに対し、もっとも衝突の起きにくいハッシュを探すだけの価値はある。ハッシュの目的は、全文検索の量を軽減することであり、特に 4D Server ではその負荷が大きいためである。
- SumBytes 以外のハッシュは驚くほど優秀である。実際、SDBM、BKDR、RS および AP の結果はどれも甲乙つけがたい。ただし、ここで使用したデータは人工的なものであり、実際のデータでも同じ結果が得られるという意味ではない。

Testing Your Data

念入りなテストを実施する気力もしくは時間がない場合、列挙されているハッシュはいずれも優秀なものであるので、適当に選んで使用しても、かなりの効果を期待することができます。

BKDR と SDBM は一般に無難なデフォルトであるとされています。自前のデータでテストを実施するのであれば、データを書き出してからサンプルデータベースで読み込んでください。ファイルには、列をふたつ用意します。

Data_Set_Name　String 60

Sample_Text　　Text

サンプルコードは、[Sample]Data_Set_Name フィールドの一致するすべてのレコードをひとつのデータセットとみなします。ハッシュは、インポート中、トリガによって自動的に作成されます。インポートが完了後、サンプルの画面でハッシュをテストすることができます。

Note　サンプルデータベースで実行されるコードの多くは、コンパイルモードでのテストおよび運用を想定しており、コンパイルモードで本来の力を発揮します。

More Information About Collisions and Why You Should Test

実際のデータでテストをしなくてもハッシュを選ぶことができるとは述べましたが、本当はテストをしたほうが望ましいのはいうまでもありません。前述のデータセットでは、それぞれのハッシュの優劣関係は常に一定でしたが、すべてのデータにおいてそうであるとは限りません。高度なハッシュは、均等な値、つまり倍長整数の値の範囲内で均等に広がる結果を返し、ある特定の値を他よりも多く返さないものであるとされています。しかしながら、本当に均等な出力は、本当に均等なデータ入力がなければ返されません。現実の世界では、入力されるデータが、真の意味では平均化されていないケースがよくあります。たとえば、英語の t という文字は z よりもはるか多く使用され、外来語を除くイタリア語では j、k、w、x、y はまったく使用されません。そのようなわけで、優秀なアルゴリズムであっても、ハッシュに偏りが生じることがあるのです。

Replacing QUERY SELECTION BY FORMULA

テクニカルノート 05-41 Case-Sensitive Operations in 4th Dimension でも論じられていますが、検索を最適化する方法について、さらに追求してゆきたいと思います。

```
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))
```

前述のハッシュ検索では、QUERY SELECTION BY FORMULA を使用しているため、とりわけ 4D Server では残念ながらパフォーマンス面で問題が残されています。ハッシュ検索の結果として残されるレコードの数が大きければ、QUERY SELECTION BY FORMULA はそれだけ多く

の全文検索を実行しなければなりません。幸い、QUERY SELECTION BY FORMULA コマンドはより高速な代替案で置き換えることができます。以下は最適化されたコードで実行されている処理の流れをまとめたものです。

1. 新しいパスをハッシュする。
2. ハッシュをインデックス検索して可能性のないレコードを排除する。
3. 残ったレコードがあれば、SELECTION TO ARRAY でロードする。
4. 配列の各要素をループで検証し、元のデータと一致するものがあれば最終的なセレクションにそのレコードを追加する。

このようにして構築したコードは、QUERY SELECTION BY FORMULA を使用する場合よりも余分に手間がかかっていますが、飛躍的に実行結果が速くなります。以下は SDBM アルゴリズムを使用し、元のコードに多少の手を入れたものです。

```
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 このコードの汎用的な運用例については、ソースコードベースに含まれている *HashTools_FindTextByHash* メソッドを参照してください。

改良されたコードで最適化の鍵となっているのは、SELECTION TO ARRAY および CREATE SELECTION FROM ARRAY コマンドの使用です。両コマンドは 4D Server 用にネットワーク最適化されています。これらのコマンドを使用すると、レコードデータそのものが転送される

代わりに、要求されたフィールドデータだけが転送されるため、レコードサイズが大きい場合は、かなりの時間短縮につながります。では、実際どれくらいの時間が節約できるのか、スピードテストの結果をみてみましょう。

How Much Time Can Hashing Save?

どんな最適化を施す場合でも、まずは実用テストを行って、改良にかかるコストと益について確かめてみることには価値があります。恣意的なシナリオを作成すれば、最適化によって物事が良くなるようにみせかけることは簡単ですが、現場で使用するデータでは、かえって動作が遅くなることもあるかもしれません。下の図では、4th Dimension で様々なデータセットを使用した検索結果が挙げられています。表の値は、いずれも複数のテストを実行した結果をミリ秒で表わした平均値です。

Source	Values	AP	BKDR	DJB	ELF	JS	PJW	RS	SDBM	Sum-	Q/BY/F
パス 1KB	1000	0.86	0.81	0.81	0.88	0.82	0.83	0.82	0.83	1.67	108.46
パス 16KB	1000	6.67	6.84	6.50	6.32	6.18	6.18	7.13	6.40	23.57	301.41
英語約千件	977	0.43	0.38	0.44	0.43	0.37	0.40	0.45	0.41	1.11	251.89
重複あり	11	0.18	0.27	0.64	0.18	0.18	0.36	0.18	0.36	0.55	268.64
英語約百件	100	0.37	0.27	0.27	0.22	0.30	0.31	0.27	0.33	0.57	267.02
平均		1.70	1.71	1.73	1.60	1.57	1.62	1.77	1.67	5.49	239.48

データセットには、比較的短く、ユニークな値のデータを使用しました。そのようなデータでは、ハッシュを使用するメリットはあまり大きくありません。データが多く、大きくなるほど、ハッシュの効果は目につくようになります。この点についての詳しい考察は、テクニカルノート 05-41 Case-Sensitive Operations in 4th Dimension に掲載されています。

Note QUERY BY FORMULA は毎回レコードをロードするため、何度テストを繰り返しても所要時間がほぼ一定です。その他のテストについては、外的な要素、つまりオペレーティングシステムの動作や、4th Dimension のバッファフラッシュなどによって変動する可能性があります。このノートではそのような影響をなるべく排除するため、テストを複数回実行しました。

AP、BKDR、DJB、ELF、JS、PJW、RS および SDBM によるハッシュはテスト結果が近く、ほぼ同等であるとみなすことができます。次の図はこれらを基準とし、様々なデータセットについて検索した場合、Sumbytes と QUERY BY FORMULA がどれだけ遅いかを表わしているものです。

Source	Values	主なハッシュ	Sumbytes	Q/B/F
パス-約 1KB	1000	1.00	2.01	130.46
パス-約 16KB	1000	1.00	3.61	46.19
約 1000 英単語	977	1.00	2.69	609.54
英単語重複あり	11	1.00	1.85	909.23
約 100 英単語	100	1.00	1.95	912.89
平均		1.00	2.42	521.66

比較すると、Sumbytes は代表的なハッシュよりも約 2.4 倍の時間がかかり、QUERY BY FORMULA は 500 倍以上の時間がかかることが分かります。比率ではなく、実際のクロック数に換算するとどうなるでしょうか。上の例では、Sumbytes 関数を使用した場合でも、データをみつけるのに要した時間はせいぜい数分の 1 秒であり、1 秒以上の時間がかかることがあるのは QUERY BY FORMULA だけです。ハッシュを使用すれば、間違いなく劇的に速度が向上することが分かりましたが、それでは 4D Server を使用した場合、またユニークではない値を検索する場合についてはどうでしょうか。はじめに 4D Server について考えてみましょう。

Source	Values	AP	BKDR	DJB	ELF	JS	PJW	RS	SDBM	Sum-	Q/BY/F
パス 1KB	1000	324	398	401	392	399	400	392	399	399	5064
パス 16KB	1000	338	396	393	395	396	397	395	396	395	8267
英語約千件	977	333	392	400	395	394	398	397	393	398	9490
重複あり	11	328	394	404	400	382	399	405	391	396	9400
英語約百件	100	332	396	398	394	394	396	397	391	397	9510
平均		331	395	399	395	395	398	397	394	397	8346

同じデータによる 4th Dimension と 4D Server のテスト結果を比較してみると、興味深いことが幾つか分かります。

- ハッシュを使用した場合の実測クエリ時間はどちらでも 1 秒以内だが、QUERY BY FORMULA はサーバで 8 秒を超える。
- Sumbytes と他のハッシュアルゴリズムのパフォーマンスの差がサーバではなくなってしまう。これは使用データセットに起因する現象である。
- 他と方法と比較したパフォーマンスの比でいえば、QUERY BY FORMULA はさほど悪くないようにみえるが、実際のクロック数ではかなり劣っている。

こうした点は、先程のように、パフォーマンスを比率で表わすとよく分かります。

Source	Values	主なハッシュ	Sumbytes	Q/B/F
パス-約 1KB	1000	1.00	1.00	12.75
パス-約 16KB	1000	1.00	1.00	20.91
約 1000 英単語	977	1.00	1.01	23.99
英単語重複あり	11	1.00	1.00	23.72
約 100 英単語	100	1.00	1.00	24.07
平均		1.00	1.00	21.09

上の表は、実際の運用環境とクロック数を考慮してテストを実行することの大切さを強調しています。4th Dimension と 4D Server のデータを単純に比較すると、QUERY BY FORMULA は 4D Server 用に最適化されていると思われるかもしれません。同コマンドは、4th Dimension ではハッシュの 521 倍の時間がかかりましたが、4D Server ではその比率が 21 倍に過ぎないからです。言い換えるなら、4D Server では 25 倍速いということもできます。しかしながら、これは他の方法に要する時間が増えたため、相対的に全体のパフォーマンス差が少なくなったことによるものです。クロック数に直せば、ハッシュは 0.3-0.4 秒、QUERY BY FORMULA は 5-9.5 秒となり、後者のほうがかなり遅いということが分かります。

A Few Words About Performance Tests

このテクニカルノートで挙げているテスト結果はどれも、参考情報に過ぎない点に留意してください。テスト結果は多くの要素に左右されるものであり、ここで挙げた結果が実際の運用システムにも当てはまるという保証はないからです。パフォーマンスの相対的な関係についても同じことがいえます。おおまかな傾向は変わらないはずですが、システムによって実際の結果はかなり異なります。すでに述べたように、テストデータが典型的な業務で使用されるものではない点にも、注意する必要があります。

4th Dimension Versus 4D Server

上記のテストで特に目を引くのが、4D Server では SumBytes の結果が優秀なハッシュアルゴリズムとさほど変わらなくなった点です。4th Dimension の場合、SumBytes は、衝突の少ないハッシュよりも平均して 1.8-3.6 倍の時間がかかりました。なぜ 4D Server では結果が異なるのでしょうか。4D Client 用の最適化された検索では、倍長整数のインデックス検索を実行しており、これに最低限、必要な処理時間があることが関係しています。

1.新しいパスをクライアント側でハッシュする。

2.4D Server にハッシュを送信してインデックス検索を実行させる。

3.4D Server から条件に合致するレコード番号とそのテキストを受け取る。

4.ローカルで配列を使用し、テキストを精査して、レコード番号の配列を作る。

5.レコード番号の配列を 4D Server に送信する。

6. CREATE SELECTION FROM ARRAY でセレクションを作る。

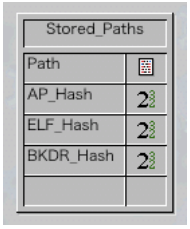
テストに使用したデータは比較的サイズが小さく、また数も少ないので、インデックス検索にかかる負担はそれほどなく、4D Server、4D Cleint は秒速で上記 6 ステップを終了しますが、それでもインデックス検索を実行するためには最低限、必要な処理時間というものがあるように思えます。このことは、10 回のテストに要する時間が 1 回のテストにかかる時間の 10 倍よりも少ないことから分かります。簡単なテストの結果、1000 レコードの中から 500 レコードのセレクションを QUERY SELECTION BY FORMULA で作るのにかかる時間は 16000 ミリ秒、上記の方法では 350 ミリ秒でした。QUERY SELECTION BY FORMULA を使用したほうが速いケースはあるのか、という点が必然的に疑問となりますが、結論からいえば、QUERY SELECTION BY FORMULA はいつでも置き換えたほうが得であるといえます。理論的には、1000 レコードの中から 1000 レコードのセレクションを作るときは QUERY SELECTION BY FORMULA が有利なように思えますが、ハッシュで最適化された方法は、そのような条件でも 400 ミリ秒であり、QUERY SELECTION BY FORMULA の 16000 ミリ秒よりずっと高速です。

Why Is QUERY SELECTION BY FORMULA So Slow?

それにしても、なぜ QUERY BY FORMULA や QUERY SELECTION BY FORMULA はこれほどパフォーマンスが落ちるのでしょうか。ひとつの理由は、QUERY BY FORMULA が SELECTION TO ARRAY のようにフィールド単位ではなくレコード全体をロードしていることにあります。レコードサイズが大きいほど、この違いは顕著です。とはいえ、今回のテストでは比較的サイズの小さなレコードを使用したので、この説明は当てはまりません。ほんとうの理由を知るためには、4th Dimension の内部コードを知る必要があります。いろいろな憶測を立てることは可能ですが、結局のところ、問題なのは再現可能な動作と、それらを回避するための方法です。ここでは、QUERY BY FORMULA や QUERY SELECTION BY FORMULA の使用をどんな場合でも避けるべきだということが明確になりました。この重要な点を念頭に置き、シーケンシャルクエリが関係しそうな処理を最適化するための実際的な方法について考慮してゆきましょう。

Technique:
Refining Query Results with Multiple Hashes

ハッシュによる最適化の中心にあるのは、シーケンシャルクエリを必要とするレコードの絶対数を少なくするという考えです。テーブルにレコードが 1000 件あれば、ネイティブマンドの QUERY BY FORMULA は 1000 件すべてを調べます。ハッシュによってその数が 600 件除外されるならば、シーケンシャルクエリの数は 400 で済みます。4D Server の場合、たとえ 400 件に減ったとしても、まだまだスピードには不満が残ります。ハッシュが衝突を多く生み出すようであれば、複数のハッシュを組み合わせて使用することができます。ハッシュは、テストデータで試してみて、あまり衝突を起こさないものを選びます。次のデータベーストラクチャでは、AP、ELF、BKDR という 3 種類のハッシュを使用して、パスを管理しています。



検索コードでは、次のようにハッシュを 3 種類使用して、セレクションを絞り込みます。

```
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.
```

複数のハッシュを使用する前に、実際のデータでよくテストを実行する必要があります。シーケンシャルクエリを実行するレコード数が少なくなれば、パフォーマンスが向上するのは確かですが、ハッシュを何回も実行するのには時間がかかり、とりわけソースデータが大きければ、それだけハッシュにも時間がかかるということを忘れてはなりません。それでは次に、いろいろなハッシュアルゴリズムによって、どれだけ処理時間に差があるのかをみてゆきましょう。

Speed Cost of Hashing

ハッシュには文字列、テキスト、BLOB、あるいはドキュメントの全文検索が関係しているので、それなりの処理時間が必要です。それでも、コンパイルすれば、ハッシュによってかなりの高速化が期待できるものです。以下の表には、コンパイルされたアプリケーションで様々なハッシュアルゴリズムを使用し、異なるデータタイプをハッシュした場合の所要時間がまとめられています。合計すると、150000 回以上のテストが実行されました。使用したハードウェアは、ごく普通のものです。時間はすべてミリ秒で記載してあります。

Data Set	Values	AP	RS	ELF	BKDR	DJB	SDBM	JS	PJW	Average
英語少	977	1	1	1	2	2	2	1	5	2
愛蘭語少	831	1	1	2	2	1	2	1	5	2
欄語少	4754	9	9	16	34	13	12	10	31	17
英語多	30405	39	41	49	46	56	56	46	185	65
1KB パス	1000	98	129	105	103	129	137	143	156	125
英語超多	118925	270	203	238	215	325	265	251	727	312
16KB パス	1000	1682	1753	1734	1767	2223	2388	2423	2400	2046
sum	157892	2100	2137	2145	2169	2749	2862	2875	3509	2568

列は、総合結果の順に並べられており、もっとも高速なものが左に配置されています。2100(2.1 秒)ミリ秒をたたき出した AP が一番優秀なハッシュで、3509 ミリ秒(3.5 秒)の PJW が一番遅いということが分かりますが、このこ結果からいえるのは、どのハッシュを選ぶかは大した問題ではないということです。普通のマシンを使用し、150000 件以上のデータでテストを実施した場合でも、一番速いハッシュと一番遅いハッシュのスピード差は 1.5 秒以下だからです。現実問題として、ハッシュにかかる時間は、ほとんど無視できるといっても構いません。

行は、総合的な所要時間の順にならべられており、もっとも高速なものが上に配置されています。この順番から、どのデータセットがより長い処理時間を必要としたのかが分かります。所要時間は、データの件数には比例しない点に留意してください。ハッシュは単純なシーケンシャル処理であり、文字列、BLOB、ドキュメントに含まれるデータを最初から最後までバイト単位で評価する必要があるので、全体の処理時間は合計バイト数によって決まります。上のテストの場合、単語はパスよりもずっと短いデータサイズで構成されているので、120000 語の単語は 16KB のパス 1000 件よりも速くハッシュすることができました。要するに、長いデータをハッシュするには長い時間がかかるということです。

Note　ここで紹介したテストでは、データをテキストタイプとして処理しています。同じデータを BLOB で処理すればさらに速くなります。テクニカルノート 05-42 Scanning Text and BLOBs Efficiently では、この点についてサンプルコードとともに詳しく論じています。テキストをポイント経由で読み取る場合の注意点についても書いてあります。

Technique: Building Longer Hash Keys

ひとつのハッシュで複数のレコードにヒットする場合、幾つかの対応策が考えられます。前述したハッシュの複数使用はそのひとつです。ここでは、複数のハッシュを個別のフィールドに保存する代わりに、ハッシュを繋ぎ合わせて単一の文字列として保存する方法を紹介します。その際のコードは次のようなものになります。

```
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
```

基本的な考えは、前述したハッシュの複数使用と同じです。異なっているのは、ハッシュをひとつの文字列フィールドに保存している点です。4th Dimension は、3 個の倍長整数フィールドをインデックス検索するよりも、ひとつの文字列フィールドをインデックス検索したほうが高速に処理できます。

文字列ハッシュをビルドするには、いろいろな方法が考えられます。数値を強制的に文字列変換する方法、16 進数にエンコードする方法、BASE64 にエンコードする方法など、結果が 80 バイト以下であれば、インデックス属性の文字列フィールドとして保存することができます。

複数のハッシュを組み合わせる手法は、HashTools コンポーネントで長いハッシュを作成する唯一の方法ではありません。例えば、32000 バイトのテキストを倍長整数のハッシュにする代わりに、8000 バイトずつのブロックに分けて、4 つのハッシュを合成する方法もあります。そのような処理が頻繁に生じるようであれば、オプションのパラメータをとして開始位置とハッシュの最大サイズを受け取るようにハッシュのルーチンを書き換えることもできます。ハッシュのルーチンを書き換える場合は、Private 属性の *HashUtility_SanityCheckFunction* メソッドを使用して、ハッシュが正しく動作していることを確認してください。もちろん、元のデータを切り分けて標準のルーチンに渡すという方法でも構いません。

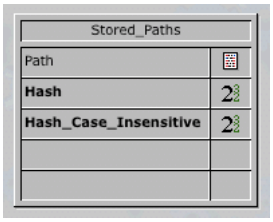
Technique: Hashing Only Part of the Data

BLOB、ドキュメント、ピクチャのような大きなオブジェクトをハッシュする場合、データすべてをハッシュしないという方法についても考慮してみる価値があります。例えば、ドキュメント全体をハッシュするのではなく、最初の 8000 バイトだけをハッシュするようにします。オブジェクトが大きい場合、そのほうが少ない時間でハッシュできます。もうひとつの利点は、ハッシュに要するおおよその時間が推測できるという点です。1000 件のドキュメントがあり、最初の 8000 バイトだけをハッシュするのであれば、所要時間は予測できますが、ドキュメント全体をハッシュする場合、その時間は各ドキュメントのサイズによって左右されます。

一部分だけをハッシュしたとしても、ハッシュの良さは少しも損なわれない点に注意してください。ハッシュが一致することから導きだされるのは、それら元のデータが同じかもしれない、ということだけです。あるいは、異なるデータが、たまたま同じハッシュになっただけなのかもしれません。結局、最終的なバイト単位の精査が行われて、両者がほんとうに一致するのかが確定します。もし、データの一部分だけをハッシュした結果、衝突の回数が増えないのであれば、一方的に処理時間が短縮できたことになります。

Technique: Case-Insensitive Comparisons

ハッシュでは、データをバイト単位で評価するので、4th Dimension の標準動作とは逆で、必然的に大文字と小文字を区別することになります。大文字と小文字を区別しない検索のほうが望ましいのであれば、ハッシュをする前に文字の種類を整えることができます。例えば、次のデータベースでは、ケースセンシティブなハッシュと非ケースセンシティブなハッシュの両方を管理しています。



これに対応するようにトリガも書き換えることができます。

```
` 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
```

ポイントとなるのは以下の部分です。

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

ハッシュをビルドする前に、ソーステキストがすべて大文字に変換されています。Uppercase を便宜上、使用しましたが、別に Lowercase を使用しても構いません。以下はこのハッシュを使用したハッシュ検索の簡単な例題です。

```
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

4th Dimension でハッシュを使用しようと考えた当初の目的は、大文字と小文字を区別し、XML パスを素早く検索するためでした。無論、ハッシュはその目的を充分に果たすことのできるテクニックですが、その他、検索に関連した色々な問題の解決にも役立つものです。コンポーネントの *HashTools_HashDocument* メソッドは、自動的に文字列、テキスト、ピクチャ、ドキュメントのハッシュ検索を実行できます。コンポーネントは、ハッシュで最適化された様々なコードを作成する際の土台とすることができます。ハッシュを応用例としては、次のようなシナリオを挙げることができます。

ある大きな建設会社の保健安全部門では、20 ヶ国以上のオフィスに対して、最新のドキュメントを何百件も配布しなければなりません。同期プロセスの一環として、各ドキュメントのハッシュがチェックサムとして配信されます。ドキュメントは届くと *HashTools_HashDocument* でハッシュされ、チェックサムと照合されます。値が一致しない場合、ドキュメントは再配信されます。

あるタレント事務所では、最近、希望者の写真や履歴書を Web で受け付けるようになりました。中には、ファイル名などをわずかに変えて、自分の書類を何度も送ってくる人がいます。重複を検出するため、システムは写真を *HashTools_HashPicture* でハッシュし、次いで保存され

ているデータから *HashTools_FindByHash* で重複を検索します。

ある商社の法律部門では、契約書のドキュメントをすべて中央のリポジトリで管理しています。4D Server には、保存されたドキュメントのパスとそのハッシュが記録されています。新しいドキュメントを受け取るたびに、データベースはそれが既存のものであるかを確かめなくてはなりません。すべてのドキュメントを全文検索する代わりに、*HashTools_HashDocument* でハッシュ検索が実行され、次いで残った候補に対して個別の比較が行われます。

Tip: Constructing "Not Equals" Searches Correctly

このテクニカルノートで論じてきたように、ハッシュはシーケンシャル検索を実行する対象の数を劇的に減らすことで検索全体を最適化しています。検索の対象は少ないほど良いからです。これまでの例では、すべて一致するレコードを見つけることに焦点が絞られていました。それでは、不一致のレコードを探す場合は、セレクションの作り方を良く考える必要があります。10000 件のレコード中、9999 件の不一致があるのであれば、はじめに一致する 1 件を検索し、セット操作で逆のセレクションを作るようにすれば、シーケンシャル検索の回数は 9999 回ではなく、1 回になります。

A Few Words for People Who Don't Like Math

このテクニカルノートでは、それぞれのハッシュアルゴリズムのコードについては特に触れていませんでした。SumBytes を除くハッシュのコードは、いずれも Java、C、C++、Pascal、Object Pascal のソースコードが下記のサイトから入手できます。

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

4th Dimension のコードは、なるべくオリジナルのコードに似せて記述してあります。例として、次に C で書かれた ELF ハッシュのコードと、それを 4th Dimension のランゲージに訳したものを併記しておきます。

```
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
```

日常的にビットワイズ演算を使用しているのでない限り、上記のコードはまるで訳が分からないものに思えるかもしれませんが、問題はありません。数学が好きな人にとって、ハッシュは非常に興味をそそられる分野ですが、そうではない我々の場合、ハッシュは魔法のコードで動いているということにしておいて、それ以上は追求しないようにしましょう。

A Note About MD5

MD5 ハッシュアルゴリズムについて聞いたことのある人は多いと思います。MD5 は、単なる文字列ハッシュではなく、暗号化をするハッシュです。暗号化されたハッシュは、逆算することが非常に難しく、したがってパスワードなどの重要なデータを処理する際に有用です。しかし、セキュアなハッシュの作成は、単なるハッシュの作成よりも複雑です。HashTools コンポーネントに含まれているその他のハッシュは数行の C 言語であるのに対し、MD5 は数百行に及びます。MD5 はそれだけ処理が重いので、特に暗号化を施す必要がない限り、他のアルゴリズムを使用したほうが賢明です。

More Details About the Source Code

サンプルデータベースには、異なるデータセットに対して様々なハッシュを適用し、スピードや衝突率を計るためのツール群が用意されています。それらユーティリティの心臓部にあたるのが各ハッシュのルーチンです。以下にメインルーチンの一覧を掲載します。太字で強調されているのが HashTools コンポーネントで閲覧することのできるメソッドです。

Method	Scope	Description
Compiler_HashTools	**Protected**	コンパイラ宣言
HashBlob_AP	Private	BLOBのAPハッシュ
HashBlob_BKDR	Private	BLOBのBKDRハッシュ
HashBlob_DJB	Private	BLOBのDJBハッシュ
HashBlob_ELF	Private	BLOBのELFハッシュ
HashBlob_JS	Private	BLOBのJSハッシュ
HashBlob_PJW	Private	BLOBのPJWハッシュ
HashBlob_RS	Private	BLOBのRSハッシュ
HashBlob_SDBM	Private	BLOBのSDBMハッシュ
HashBlob_SumBytes	Private	BLOBのSumBytesハッシュ
HashDocument_AP	Private	ドキュメントのAPハッシュ
HashDocument_BKDR	Private	ドキュメントのBKDRハッシュ
HashDocument _DJB	Private	ドキュメントのDJBハッシュ
HashDocument _ELF	Private	ドキュメントのELFハッシュ
HashDocument _JS	Private	ドキュメントのJSハッシュ
HashDocument _PJW	Private	ドキュメントのPJWハッシュ
HashDocument _RS	Private	ドキュメントのRSハッシュ
HashDocument _SDBM	Private	ドキュメントのSDBMハッシュ
HashDocument _SumBytes	Private	ドキュメントのSumBytesハッシュ

HashDocument_AP	Private	テキストのAPハッシュ
HashDocument_BKDR	Private	テキストのBKDRハッシュ
HashDocument _DJB	Private	テキストのDJBハッシュ
HashDocument _ELF	Private	テキストのELFハッシュ
HashDocument _JS	Private	テキストのJSハッシュ
HashDocument _PJW	Private	テキストのPJWハッシュ
HashDocument _RS	Private	テキストのRSハッシュ
HashDocument _SDBM	Private	テキストのSDBMハッシュ
HashDocument _SumBytes	Private	テキストのSumBytesハッシュ
HashDocument _AP	Private	テキストのAPハッシュ
HashTools Read Me Private	Private	全コードの説明
HashTools Read Me Public	**Public**	Public/Protectedコードの説明
HashTools_FindBlobByHash	Private	BLOBをハッシュ検索
HashTools_FindByHash	**Protected**	ハッシュ検索
HashTools_FindPictureByHash	Private	ピクチャをハッシュ検索
HashTools_FindTextByHash	Private	文字列またはテキストをハッシュ検索
HashTools_GetErrorText	**Protected**	エラーテキスト
HashTools_GetHashTypeNames	**Protected**	サポートされるハッシュタイプリスト
HashTools_GetLastErrorCode	**Protected**	最後のエラーコード
HashTools_GetLastErrorLocation	**Protected**	エラーを記録した最後のメソッド
HashTools_HashBlob	**Protected**	BLOBをハッシュ
HashTools_HashDocument	**Protected**	ドキュメントをハッシュ
HashTools_HashPicture	**Protected**	ピクチャをハッシュ
HashTools_HashText	**Protected**	文字列またはテキストをハッシュ
HashTools_InitializeErrorText	Private	エラーテキストをロード
HashTools_InitializeTypeNames	Private	ハッシュタイプリストをロード
HashTools_InstallErrorSystem	Private	エラーハンドルをインストール
HashTools_RestoreErrorHandler	Private	エエラーハンドルがあれば復元
HashTools_SetError	Private	エラーコードセット
HashUtility_AlphasAreEqual	Private	ケースセンシティブ文字列比較
HashUtility_DocRefIsValid	Private	ドキュメントが開かれているか確認
HashUtility_PicturesAreEqual	Private	ピクチャの比較
HashUtility_SanityCheckFunction	Private	サニティチェック
HashUtility_TypeNameIsValid	Private	ハッシュタイプ名の検証