TMTO[dot]ORG: Building Integer Based TMTO Tables

Author: Jason R. Davis
Site: TMTO[dot]ORG

Table of Contents

Created Date: December 19th, 2010
Last Edited: December 19th, 2010

Foreword

The process of building TMTO tables has often been a treacherous one. The primary evolution of the concept is to reduce the amount of space the tables consume - as a key space increases so do the memory requirements. We all want larger tables and that is a fact. There are many different ways to build a TMTO table, but this article goes into detail how they are generated on TMTO[dot]ORG. I haven't officially released any code or an application – because that's not really my role as a researcher – I'll leave that to the programmers. However, I am releasing the concept into the public domain for future application of others.

It is important to recognize the time frame in which this method was first developed. Initially, the ideas were conveyed into a concept and design review in October, 2006. The first proof of concept code was written in November, 2006 and finally the first TMTO database that used this method was publicly searchable on January 11th, 2007. The concept of integer based TMTO tables will be turning 3 years old shortly after this paper is published. I had released snippets of code earlier on a few choice sites, but never fully explained it – except through email with a few interested patrons. With that said, I decided it was time to publish a full explanation of what integer based TMTO tables are and how they are built.

It should also be clear that this article is focused on relaying the concept of integer based TMTO tables; not optimization, storage methods or speed. I've had 3 years to optimize and modify this concept, so I'll leave that up to each reader. For readability the code in this article is written in PHP. This allows you to quickly modify and port the concept to faster languages. I've since rewritten the code in C++ and Java. Currently, I use C++ and the MPI library to build tables for TMTO[dot]ORG.

Chapter 1 – Building: *The Math*

After reviewing many of the current methods of storing strings in TMTO tables, it was obvious I would eventually have to switch to an integer based method.

First you must calculate the key space size. In this example I'll use a-z with a length of 1-5.

$$(26^1)+(26^2) \ldots (26^5) = 12,356,630$$

This is a number that you will compare your results to later on. Then, calculate the character ranges for each key space ranges in code (PHP for readability):

Note: Remember that arrays are zero initialized.

```
1.  foreach(range(0,5) as $m) {
2.    foreach(range(0,25) as $n) {
3.      $characters[$m][$n] = chr($n+32);
4.    }
5.  }
```

This builds a multidimensional array that looks like:

```
1.  $characters[0][0]  = "a"
2.  $characters[0][2]  = "b"
3.  $characters[0][3]  = "c"
4.  $characters[0][…]  = "…"
5.  $characters[0][23] = "x"
6.  $characters[0][24] = "y"
7.  $characters[0][25] = "z"
8.  …
9.  $characters[5][0]  = "a"
10. $characters[5][1]  = "b"
11. $characters[5][2]  = "c"
12. $characters[5][…]  = "…"
13. $characters[5][23] = "x"
14. $characters[5][24] = "y"
15. $characters[5][25] = "z"
```

The first dimension is the length and the second dimension is the characters numbered value. Each array element is assigned a value and that is the actual character. This array will lay the foundation for calculating the key space size.

Next we create the two arrays one containing the integer sizes of length and the number of characters in each length.

```
1.  foreach (range(0,5) as $n) {
2.    $length[$n] = $n;
3.    $chrset[$n] = sizeof($characters[$n]);
4.  }
```

Below are the resulting arrays:

```
1.  $length[0] = 0
2.  $length[1] = 1
3.  $length[2] = 2
4.  $length[3] = 3
5.  $length[4] = 4
6.  $length[5] = 5
```

```
1.  $chrset[0] = 26
2.  $chrset[1] = 26
3.  $chrset[2] = 26
4.  $chrset[3] = 26
5.  $chrset[4] = 26
6.  $chrset[5] = 26
```

Then, we use the $length and $chrset arrays to generate our ranges, which in turn will calculate the key space size and beginning point for each length.

```
1.  $nranges[0] = 0;
2.    for ($i=1; $i<=sizeof($length); $i++) {
3.    $nranges[$i] = $nranges[$i-1] + pow($chrset[$i-1],$length[$i-1]);
4.  }
```

This creates the array below:

```
$nranges[0] = 0
$nranges[1] = 1
$nranges[2] = 27
$nranges[3] = 703
$nranges[4] = 18279
$nranges[5] = 475255
$nranges[6] = 12356631
```

Now this array will take a bit of explanation. Index zero of the array has a value of zero – this is intentional and represents a NULL or empty string. This is important to include. Index 1 starts at 1, because that is where the 1 length strings will begin. Index two starts at 27, because there are 26 permutations in the 1 length key space plus the NULL or empty string. Index 3 starts at 703 because there are 676 permutations in the 2 length key space plus 26 in the 1 length key space plus the NULL or empty string. This continues on till the array index of 6, which indicates the last integer in the entire key space and should equal the calculation we made earlier plus one because of the NULL or empty string value of index zero. If you revisit the equation from the beginning the math extrapolates perfectly:

$$(26^1)+(26^2) \ldots (26^5) + 1 = 12{,}356{,}631 == \$nranges[6]$$

Once the starting points for each length are defined we can begin building strings using integers.

First we will create a loop that will run through the entire key space and output each permutation as it goes along:

```
1.  $t = 0;
2.  while ($t<max($nranges)) {
3.    $str  = n2str($t);
4.    $hash = hash('md5',$str);
5.    echo $hash.':'.$str.':'.$t."\n"; // $t is the integer equivalent of $str
6.    $t++;
7.  }
```

There is a function in bold and underlined in the code above on line 3, this is that function:

```
1.  function n2str($num) {
2.    global $nranges, $length, $chrset, $characters;
3.    $i=0;
4.    while ($nranges[$i+1]<=$num) {
5.      $i++;
6.    }
7.    $n = $num-$nranges[$i];
8.    $len = $length[$i];
9.    $chr = $chrset[$i];
10.   $st  = '';
11.   for ($i=0;$i<$len;$i++) {
12.     $k1 = $n % $chr;
13.     $n  = floor($n / $chr);
14.     $st .= $characters[$len][$k1];
15.   }
16.   return $st;
17. }
```

See index 1.0 for a complete walk through of function n2str().

This will output the text below when run:

```
1.  d41d8cd98f00b204e9800998ecf8427e::0
2.  0cc175b9c0f1b6a831c399e269772661:a:1
3.  92eb5ffee6ae2fec3ad71c777531578f:b:2
4.  4a8a08f09d37b73795649038408b5f33:c:3
5.  …
6.  9dd4e461268c8034f5c8564e155c67a6:x:24
7.  415290769594460e2e485922904f345d:y:25
8.  fbade9e36a3f36d3d676c1b808451dd7:z:26
9.  4124bc0a9335c27f086f24ba207a4912:aa:27
10. 07159c47ee1b19ae4fb9c40d480856c4:ba:28
11. 5435c69ed3bcc5b2e4d580e393e373d3:ca:29
12. …
```

On line 1 you see the NULL or empty string value. line 2 begins with the 1 length permutations and increments up. This script will continue to output until the while condition is met, and that is when $t exceeds the max value of $nranges, which in this case is $nranges[6] = 12,356,631. Now we've set a good foundation for generating the hash, the string, and the integer used to generate the string.

Below is all the code discussed so far combined into a functioning script:

```
1.  #!/usr/bin/env php
2.  <?php
3.  foreach (range(0,5) as $m) {
4.    foreach (range(0,25) as $n) {
5.      $characters[$m][$n] = chr($n+97);
6.    }
7.  }
8.  foreach (range(0,5) as $n) {
9.    $length[$n] = $n;
10.   $chrset[$n] = sizeof($characters[$n]);
11. }
12. $nranges[0] = 0;
13. for ($i=1; $i<=sizeof($length); $i++) {
14.   $nranges[$i] = $nranges[$i-1] + pow($chrset[$i-1],$length[$i-1]);
15. }
16. function n2str($num) {
17.   global $nranges, $length, $chrset, $characters;
18.   $i=0;
19.   while ($nranges[$i+1]<=$num) {
20.     $i++;
21.   }
22.   $n = $num-$nranges[$i];
23.   $len = $length[$i];
24.   $chr = $chrset[$i];
25.   $st  = '';
26.   for ($i=0;$i<$len;$i++) {
27.     $k1 = $n % $chr;
28.     $n  = floor($n / $chr);
29.     $st .= $characters[$len][$k1];
30.   }
31.   return $st;
32. }
33. $t = 0;
34. while ($t<max($nranges)) {
35.   $str  = n2str($t);
36.   $hash = hash('md5',$str);
37.   echo $hash.':'.$str.':'.$t."\n"; // $t is the integer equivalent of $str
38.   $t++;
39. }
40. ?>
```

The only reason we generate the string is so we can generate the MD5 hash. Next let's talk about generating the files that will store this information. We are going to create 65,536 files. Each file will contain all the integers that when converted to a string and hashed the first four hex characters match the file name.

Step 1: $int = 1
Step 2: n2str(1) = a
Step 3: hash('md5',a) = 0cc175b9c0f1b6a831c399e269772661
Step 4: $file = hexdec('0cc1') or 3265

This means I would put the integer 1, into file name '3265'. All the hashes that start with '0cc1' will have their equivalent integer put within that file as well. Let's begin with defining where the files will be stored and opening up all the file pointers with the following loop:

```
1.  $scratch = "/storage/scratch/";
2.  foreach (range(0,65535) as $i) {
3.    $fp[$i] = fopen($scratch.$i,'ab');
4.  }
```

A common warning that results from running that loop is “Too many open files…” and is a simple environment limitation that can be fixed by adjusting the limits in /etc/security/limits.conf:

```
username soft nofile 100000
username hard nofile 101024
```

Then log off and log back in. Issuing the “ulimit –n” command should result in the soft limit number.

```
workstation ~ $ ulimit -n
100000
```

This will keep all the file pointers open during the process of building.

Next we must modify the loop to append the integer for each string to the correct file during the build process. Each integer within the file is separated with the “:” character. There are many ways to store the data within a file – binary, etc. – but I’ve decided to keep it very simple to read each file and easy to understand.

```
1.  $t = 0;
2.  while ($t<max($nranges)) {
3.    $str  = n2str($t);
4.    $hash = substr(hash('md5',$str),0,4);
5.    fwrite($fp[hexdec($hash)],':'.$t);
6.    $t++;
7.  }
```

Chapter 1 is continued on the next page.

Once this modification is complete, we can start building the files to complete our table. The modified and final version of the script is below:

```
1.  #!/usr/bin/env php
2.  <?php
3.  foreach (range(0,5) as $m) {
4.    foreach (range(0,25) as $n) {
5.      $characters[$m][$n] = chr($n+97);
6.    }
7.  }
8.  foreach (range(0,5) as $n) {
9.    $length[$n] = $n;
10.   $chrset[$n] = sizeof($characters[$n]);
11. }
12. $nranges[0] = 0;
13. for ($i=1; $i<=sizeof($length); $i++) {
14.   $nranges[$i] = $nranges[$i-1] + pow($chrset[$i-1],$length[$i-1]);
15. }
16. function n2str($num) {
17.   global $nranges, $length, $chrset, $characters;
18.   $i=0;
19.   while ($nranges[$i+1]<=$num) {
20.     $i++;
21.   }
22.   $n = $num-$nranges[$i];
23.   $len = $length[$i];
24.   $chr = $chrset[$i];
25.   $st  = '';
26.   for ($i=0;$i<$len;$i++) {
27.     $k1 = $n % $chr;
28.     $n  = floor($n / $chr);
29.     $st .= $characters[$len][$k1];
30.   }
31.   return $st;
32. }
33. $scratch = "/storage/scratch/";
34. foreach(range(0,65535) as $i) {
35.   $fp[$i] = fopen($scratch.$i,'ab');
36. }
37. $t = 0;
38. while ($t<max($nranges)) {
39.   $str  = n2str($t);
40.   $hash = substr(hash('md5',$str),0,4);
41.   fwrite($fp[hexdec($hash)],':'.$t);
42.   $t++;
43. }
44. ?>
```

Let's run the script and build the files:

```
workstation ~ $ time ./tmto.php
real    4m49.697s
user    1m48.220s
sys     1m12.360s
```

It took about 5 minutes to complete.

Next we read one of the files to make sure they have data in them:

```
workstation ~ $ cat /storage/scratch/0
:17139:42891:61878:179653:308329:333409:374620:399225:406517:517309:723308:916806:988926:1053807:
1093275:1218817:1336310:1426143:1446034:1526186:1593171:1831098:1975512:1995735:1996713:1999306:2
031918:2035682:2072780:2096586:2205244:2302756:2348748:2412345:2414805:2468559:2521387:2532544:26
27792:2645013:2648090:2648785:2701668:2930528:2999013:3047416:3048808:3102979:3461146:3560077:381
8878:3914202:3927968:3959892:3964605:3975065:4025225:4032761:4068698:4199716:4261648:4350758:4529
292:4540701:4643938:4670179:4738511:4919992:4956165:4996781:4999782:5110565:5123544:5318167:55327
80:5538186:5697425:5755319:5778942:5941896:5945722:6106886:6106965:6183879:6236187:6285218:630474
8:6415264:6445963:6451215:6493474:6494560:6574669:6580495:6617161:6713170:6730687:6870096:6910555
:7012424:7052533:7072907:7096011:7158142:7164629:7173807:7346667:7485145:7555513:7608226:7849390:
7906678:7973539:8017743:8067918:8358360:8359647:8449558:8455312:8462555:8678294:8691405:8725156:8
733688:8779321:8829137:8849519:8864831:8918553:8991581:9087059:9098190:9115002:9142953:9192489:93
85049:9449375:9544709:9598982:9852509:9859471:9883066:9931695:9954059:10002514:10213286:10224788:
10363364:10574596:10733415:10887380:11290152:11324482:11378452:11398102:11461153:11521391:1153911
5:11634928:11709823:11730131:11776231:11829844:11894847:11898929:11930288:12000464:12070407:12211
248:12245068:12254208
```

As you can see the file contains only integers with the ":" separator. This is what we want. In the next chapter I go into how a value is retrieved from a file based on a MD5 hash.

Chapter 2 – Retrieving

Now that you've successfully built a table it's time to create a client that will retrieve the values. I'll start by giving the client code and then walking you through how it works. Below is the client code to retrieve a string using a MD5 hash:

```
1.  #!/usr/bin/env php
2.  <?php
3.  foreach (range(0,5) as $m) {
4.    foreach (range(0,25) as $n) {
5.      $characters[$m][$n] = chr($n+97);
6.    }
7.  }
8.  foreach (range(0,5) as $n) {
9.    $length[$n] = $n;
10.   $chrset[$n] = sizeof($characters[$n]);
11. }
12. $nranges[0] = 0;
13. for ($i=1; $i<=sizeof($length); $i++) {
14.   $nranges[$i] = $nranges[$i-1] + pow($chrset[$i-1],$length[$i-1]);
15. }
16. function n2str($num) {
17.   global $nranges, $length, $chrset, $characters;
18.   $i=0;
19.   while ($nranges[$i+1]<=$num) {
20.     $i++;
21.   }
22.   $n = $num-$nranges[$i];
23.   $len = $length[$i];
24.   $chr = $chrset[$i];
25.   $st  = '';
26.   for ($i=0;$i<$len;$i++) {
27.     $k1 = $n % $chr;
28.     $n  = floor($n / $chr);
29.     $st .= $characters[$len][$k1];
30.   }
31.   return $st;
32. }
33.
34. function row_str($n) { return n2str($n); }
35. function row_md5($s) { return hash('md5',$s); }
36.
37. $hash = $argv[1];
38. $file = '/storage/scratch/'.hexdec(substr($hash,0,4));
39. $fc   = file_get_contents($file);
40. $rows_i = explode(':',$fc);
41. $rows_s = array_map("row_str",$rows_i);
42. $rows_h = array_map("row_md5",$rows_s);
43.
44. if(in array($hash,$rows h)) {
45.   echo $hash.':'.$rows_s[array_search($hash,$rows_h)]."\n";
46. }
47. ?>
```

And here are some examples of it running. The first argument is the MD5 that you wish to search for. It'll output the hash and the matching clear text. See below:

```
workstation ~ $ ./ctmto.php e2fc714c4727ee9395f324cd2e7f331f
e2fc714c4727ee9395f324cd2e7f331f:abcd

workstation ~ $ ./ctmto.php 20226ff00e38800e5d0c3e975c646bb7
20226ff00e38800e5d0c3e975c646bb7:vfrd

workstation ~ $ ./ctmto.php 0e7bbf878a447a16ce7ecb55b62399df
0e7bbf878a447a16ce7ecb55b62399df:edcfr
```

Lines 1 through 32 are identical to the script used to build the tables. The same ranges that are used to build the values that are to retrieve them. That is the natural relation of the tables to the ranges and the reason why they must be accurate.

Two callback functions are used in the array_maps() within the script:

```
1.  function row str($n) { return n2str($n); }
2.  function row md5($s) { return hash('md5',$s); }
```

The first changes each integer value in an array to a string. The second changes each string value in an array to a MD5 hash.

```
1.  $hash = $argv[1];
2.  $file = '/storage/scratch/'.hexdec(substr($hash,0,4));
3.  $fc   = file_get_contents($file);
4.  $rows_i = explode(':',$fc);
5.  $rows s = array map("row str",$rows i);
6.  $rows h = array map("row md5",$rows s);
```

Line 1 sets the first script argument to the value of variable $hash. Line 2 uses the hash to determine the file in which to open. Line 3 reads the file into memory and Line 4 separates each integer and assigns it to an array value of the $rows_i array. Line 5 takes each of the values of $rows_i and creates another array ($rows_s) of strings that equal each of the integer values in $rows_i. Line 6 then builds a third array ($rows_h) of all the values of $rows_s and assigns values equal to the MD5 hash.

$rows_i	→	$rows_s	→	$rows_h
1	func(row_str)	a	func(row_md5)	0cc175b9c0f1b6a831c399e269772661
2	func(row_str)	b	func(row_md5)	92eb5ffee6ae2fec3ad71c777531578f

Once all three arrays have been fully built we use in_array() to check if a match exists. If one does, we use array_search to return the index of $rows_s that matches. The match is echo'd to the console.

```
1.  if(in array($hash,$rows h)) {
2.    echo $hash.':'.$rows_s[array_search($hash,$rows_h)]."\n";
3.  }
```

You can now modify the build script and start building tables. Remember that $nranges in the client must match the $nranges use to build the tables. A common mistake I made was forgetting to adjust the retrieval clients $nranges array.

Index

1.0 – function n2str()

This function has a few dependencies. The key space must be pre-calculated and the ranges of each length must be mapped.

This is easily accomplished with the code below, and is explained in Chapter 1.

```
1.  foreach (range(0,5) as $m) {
2.    foreach (range(0,25) as $n) {
3.      $characters[$m][$n] = chr($n+97);
4.    }
5.  }
6.  foreach (range(0,5) as $n) {
7.    $length[$n] = $n;
8.    $chrset[$n] = sizeof($characters[$n]);
9.  }
10. $nranges[0] = 0;
11. for ($i=1; $i<=sizeof($length); $i++) {
12.   $nranges[$i] = $nranges[$i-1] + pow($chrset[$i-1],$length[$i-1]);
13. }
```

Once the $characters, $length, $chrset, and $nranges arrays are built you can pass a number to n2str() and it'll return the text equivalent.

With the following defined :

```
$nranges[0] = 0
$nranges[1] = 1
$nranges[2] = 27
$nranges[3] = 703
$nranges[4] = 18279
$nranges[5] = 475255
$nranges[6] = 12356631
```

I'll now walk through the n2str() function when called with the argument of 1000:

```
1.  function n2str($num) {
2.    global $nranges, $length, $chrset, $characters;
3.    $i=0;
4.    while ($nranges[$i+1]<=$num) {
5.      $i++;
6.    }
7.    $n = $num-$nranges[$i];
8.    $len = $length[$i];
9.    $chr = $chrset[$i];
10.   $st  = '';
11.   for ($i=0;$i<$len;$i++) {
12.     $k1 = $n % $chr;
13.     $n  = floor($n / $chr);
14.     $st .= $characters[$len][$k1];
15.   }
16.   return $st;
17. }
```

At line 2 the arrays are imported into the function.

Lines 3-6 determine which length range the argument of 1000 falls in. In this case $i would increment to 3 because $nranges[3+1] is not less than 1000.

Line 7 then calculates the amount left over when you take 1000 - $nranges[3], which equals 297.

Line 8 and 9 define the $len and $chr values based on $i and retrieves them from the global arrays of $length and $chrset which were pre-computed before the function was called.

At this point:

$i = 0
$n = 297
$len = 3
$chr = 26

At line 11, a for loop is created that increments up while it's less than $len's value. In this case less than three or equal to two.

Line 12 sets the variable $k1 = $n modulus $chr. $k1 = 297 mod 26 = 11. $k1 now equals 11.

Line 13 then modifies $n by dividing it by $chr and rounds it down to a whole number. $n = 297 / 26 = 11.42, when rounded down comes to 11. $n now equals 11.

Line 15, the first character is retrieved by using the $len and $k1 variables as indexes of the multi-dimensional array $characters. In this case $len = 3 and $k1 = 11, so the variable $st now equal $characters[3][11] or the letter 'l' (el).

At this point:
$i = 1
$n = 11
$len = 3
$chr = 26
$st = l

$k1 = $n modulus $chr. $k1 = 11 mod 26 = 11. $k1 equals 11.
$n = 11 / 26 = 0.42, when rounded down comes to 0. $n equals 0.
$st is modified with $characters[3][11], so $st is equal to 'll' (el, el).

We still have one increment left in the for loop.

At this point:
$i = 1
$n = 0;
$len = 3;
$chr = 26;
$st = 'll'

$k1 = $n modulus $chr. $k1 = 0 mod 26 = 0. $k1 equals 0.
$n = 0 / 26 = 0, when rounded down comes to 0. $n equals 0.
$st is modified with $characters[3][0], so $st is equal to 'lla' (el, el, a).

Finally $st is returned from the function.

This means that the ‘lla’ is the string equivalent of 1000.