|  |  |
| --- | --- |
| 1924down voteaccepted  +300 | I tested some different algorithms, measuring speed and number of collisions.  I used three different key sets:   * [A list of 216,553 English words](http://www.sitopreferito.it/html/all_english_words.html) (in lowercase) * The numbers "1" to "216553" (think ZIP codes, and [how a poor hash took down msn.com](http://blogs.msdn.com/b/ericlippert/archive/2003/09/19/arrrrr-cap-n-eric-be-learnin-about-threadin-the-harrrrd-way.aspx)) * 216,553 "random" (i.e. [type 4 uuid](http://www.ietf.org/rfc/rfc4122.txt)) GUIDs   For each corpus, the number of collisions and the average time spent hashing was recorded.  I tested:   * [DJB2](http://www.cse.yorku.ca/~oz/hash.html) * [DJB2a](http://www.cse.yorku.ca/~oz/hash.html) (variant using xor rather than +) * [FNV-1](http://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function) (32-bit) * [FNV-1a](http://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function) (32-bit) * [SDBM](http://www.cse.yorku.ca/~oz/hash.html) * CRC32 * [Murmur2](http://en.wikipedia.org/wiki/MurmurHash) (32-bit) * [SuperFastHash](http://landman-code.blogspot.ca/2008/06/superfasthash-from-paul-hsieh.html)   Results  Each result contains the average hash time, and the number of collisions  Hash Lowercase Random UUID Numbers  ============= ============= =========== ==============  Murmur 145 ns 259 ns 92 ns  6 collis 5 collis 0 collis  FNV-1a 152 ns 504 ns 86 ns  4 collis 4 collis 0 collis  FNV-1 184 ns 730 ns 92 ns  1 collis 5 collis 0 collis▪  DBJ2a 158 ns 443 ns 91 ns  5 collis 6 collis 0 collis▪▪▪  DJB2 156 ns 437 ns 93 ns  7 collis 6 collis 0 collis▪▪▪  SDBM 148 ns 484 ns 90 ns  4 collis 6 collis 0 collis\*\*  SuperFastHash 164 ns 344 ns 118 ns  85 collis 4 collis 18742 collis  CRC32 250 ns 946 ns 130 ns  2 collis 0 collis 0 collis  LoseLose 338 ns - -  215178 collis  **Notes**:   * The [LoseLose algorithm](http://www.cse.yorku.ca/~oz/hash.html) (where hash = hash+character) is truly ***awful***. Everything collides into the same 1,375 buckets * SuperFastHash is fast, with things looking pretty scattered; by my goodness the *number*collisions. I'm hoping [the guy who ported it got something wrong; it's pretty bad](http://landman-code.blogspot.ca/2008/06/superfasthash-from-paul-hsieh.html) * CRC32 is *pretty good*. Slower, and a 1k lookup table   Do collisions actually happen?  Yes. I started writing my test program to see if hash collisions *actually* happen - and are not just a theoretical construct. They do indeed happen:  **FNV-1 collisions**   * creamwove collides with quists   **FNV-1a collisions**   * costarring collides with liquid * declinate collides with macallums * altarage collides with zinke * altarages collides with zinkes   **Murmur2 collisions**   * cataract collides with periti * roquette collides with skivie * shawl collides with stormbound * dowlases collides with tramontane * cricketings collides with twanger * longans collides with whigs   **DJB2 collisions**   * hetairas collides with mentioner * heliotropes collides with neurospora * depravement collides with serafins * stylist collides with subgenera * joyful collides with synaphea * redescribed collides with urites * dram collides with vivency   **DJB2a collisions**   * haggadot collides with loathsomenesses * adorablenesses collides with rentability * playwright collides with snush * playwrighting collides with snushing * treponematoses collides with waterbeds   **CRC32 collisions**   * codding collides with gnu * exhibiters collides with schlager   **SuperFastHash collisions**   * dahabiah collides with drapability * encharm collides with enclave * grahams collides with gramary * ...snip 79 collisions... * night collides with vigil * nights collides with vigils * finks collides with vinic   Randomnessification  The other subjective measure is how randomly distributed the hashes are. Mapping the resulting HashTables shows how evenly the data is distributed. All the hash functions show good distribution when mapping the table linearly:  Enter image description here  Or as a [*Hilbert Map*](http://en.wikipedia.org/wiki/Hilbert_curve) ([XKCD is always relevant](http://xkcd.com/195/)):  Enter image description here  Except when hashing number strings ("1", "2", ..., "216553") (for example, [zip codes](http://blogs.msdn.com/b/ericlippert/archive/2003/09/19/arrrrr-cap-n-eric-be-learnin-about-threadin-the-harrrrd-way.aspx)), where patterns begin to emerge in most of the hashing algorithms:  **SDBM**:  Enter image description here  **DJB2a**:  Enter image description here  **FNV-1**:  Enter image description here  All except **FNV-1a**, which still look pretty random to me:  Enter image description here  In fact, **Murmur2** seems to have even better randomness with Numbers than FNV-1a:  Enter image description here  *When I look at the FNV-1a "number" map, I****think****I see subtle vertical patterns. With Murmur I see no patterns at all. What do you think?*  The extra **\*** in the above table denotes how bad the randomness is. With FNV-1a being the best, and **DJB2x** being the worst:  Murmur2: .  FNV-1a: .  FNV-1: ▪  DJB2: ▪▪  DJB2a: ▪▪  SDBM: ▪▪▪  SuperFastHash: .  CRC: ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪  Loselose: ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪  ▪  ▪▪▪▪▪▪▪▪▪▪▪▪▪  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪  ▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪  I originally wrote this program to decide if I even had to *worry* about collisions: I do.  And then it turned into making sure that the hash functions were sufficiently random.  FNV-1a algorithm  The FNV1 hash comes in variants that return 32, 64, 128, 256, 512 and 1024 bit hashes.  The [FNV-1a algorithm](http://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function) is:  hash = FNV\_offset\_basis  for each octetOfData to be hashed  hash = hash xor octetOfData  hash = hash \* FNV\_prime  return hash  Where the constants FNV\_offset\_basis and FNV\_prime depend on the return hash size you want:  Hash Size Prime Offset  =========== =========================== =================================  32-bit 16777619 2166136261  64-bit 1099511628211 14695981039346656037  128-bit 309485009821345068724781371 144066263297769815596495629667062367629  256-bit  prime: 2^168 + 2^8 + 0x63 = 374144419156711147060143317175368453031918731002211  offset: 100029257958052580907070968620625704837092796014241193945225284501741471925557  512-bit  prime: 2^344 + 2^8 + 0x57 = 35835915874844867368919076489095108449946327955754392558399825615420669938882575126094039892345713852759  offset: 9659303129496669498009435400716310466090418745672637896108374329434462657994582932197716438449813051892206539805784495328239340083876191928701583869517785  1024-bit  prime: 2^680 + 2^8 + 0x8d = 5016456510113118655434598811035278955030765345404790744303017523831112055108147451509157692220295382716162651878526895249385292291816524375083746691371804094271873160484737966720260389217684476157468082573  offset: 1419779506494762106872207064140321832088062279544193396087847491461758272325229673230371772250864096521202355549365628174669108571814760471015076148029755969804077320157692458563003215304957150157403644460363550505412711285966361610267868082893823963790439336411086884584107735010676915  See [the main FNV page](http://www.isthe.com/chongo/tech/comp/fnv/index.html#FNV-param) for details.  As a practical matter:   * 32-bit UInt32, * 64-bit UInt64, and * 128-bit Guid can be useful   All my results are with the 32-bit variant.  FNV-1 better than FNV-1a?  No. FNV-1a is all around better. There was more collisions with FNV-1a when using the English word corpus:  Hash Word Collisions  ====== ===============  FNV-1 1  FNV-1a 4  Now compare lowercase and uppercase:  Hash lowercase word Collisions UPPERCASE word collisions  ====== ========================= =========================  FNV-1 1 9  FNV-1a 4 11  In this case FNV-1a isn't *"400%"* worse than FN-1, only 20% worse.  I think the more important takeaway is that there are two classes of algorithms when it comes to collisions:   * **collisions rare**: FNV-1, FNV-1a, DJB2, DJB2a, SDBM * **collisions common**: SuperFastHash, Loselose   And then there's the how evenly distributed the hashes are:   * **outstanding distribution:** Murmur2, FNV-1a, SuperFastHas * **excellent distribution:** FNV-1 * **good distribution:** SDBM, DJB2, DJB2a * **horrible distribution:** Loselose   **Update**  Murmur? *Sure, why not*  **Update**  @whatshisname wondered how a **CRC32** would perform, added numbers to the table.  CRC32 is *pretty good*. Few collisions, but slower, and the overhead of a 1k lookup table.  *Snip all erroneous stuff about CRC distribution - my bad*  Up until today I was going to use FNV-1a as my *de facto* hash-table hashing algorithm. But now I'm switching to Murmur2:   * Faster * Better *randomnessification* of all classes of input   And I really, *really* hope there's something wrong with the [SuperFastHash algorithm I found](http://landman-code.blogspot.ca/2008/06/superfasthash-from-paul-hsieh.html); it's too bad to be as popular as it is.  **Update:** From [the MurmurHash3 homepage on Google](http://code.google.com/p/smhasher/wiki/MurmurHash3):  (1) - SuperFastHash has very poor collision properties, which have been documented elsewhere.  So I guess it's not just me.  **Update:** I realized why Murmur is faster than the others. MurmurHash2 operates on four bytes at a time. Most algorithms are *byte by byte*:  for each octet in Key  AddTheOctetToTheHash  This means that as keys get longer Murmur gets its chance to shine.  **Update**  [GUIDs are designed to be unique, not random](http://blogs.msdn.com/b/oldnewthing/archive/2012/05/23/10309199.aspx)  A timely post by Raymond Chen reiterates the fact that *"random"* GUIDs are not meant to be used for their randomness. They, or a subset of them, are unsuitable as a hash key:  Even the Version 4 GUID algorithm is not guaranteed to be unpredictable, because the algorithm does not specify the quality of the random number generator. [The Wikipedia article for GUID contains primary research which suggests](http://en.wikipedia.org/wiki/Globally_unique_identifier) that future and previous GUIDs can be predicted based on knowledge of the random number generator state, since the generator is not cryptographically strong.  Randomess is not the same as collision avoidance; which is why it would be a mistake to try to invent your own "hashing" algorithm by taking some subset of a "random" guid:  int HashKeyFromGuid(Guid type4uuid)  {  //A "4" is put somewhere in the GUID.  //I can't remember exactly where, but it doesn't matter for  //the illustrative purposes of this pseudocode  int guidVersion = ((type4uuid.D3 & 0x0f00) >> 8);  Assert(guidVersion == 4);  return (int)GetFirstFourBytesOfGuid(type4uuid);  }  **Note**: Again, I put *"random GUID"* in quotes, because it's the "random" variant of GUIDs. A more accurate description would be Type 4 UUID. But nobody knows what type 4, or types 1, 3 and 5 are. So it's just easier to call them "random" GUIDs.  **All English Words mirrors**   * [https://web.archive.org/web/20070221060514/http://www.sitopreferito.it/html/all\_english\_words.html](https://web.archive.org/web/20070221060514/http:/www.sitopreferito.it/html/all_english_words.html) * <https://drive.google.com/file/d/0B3BLwu7Vb2U-dEw1VkUxc3U4SG8/view?usp=sharing> |