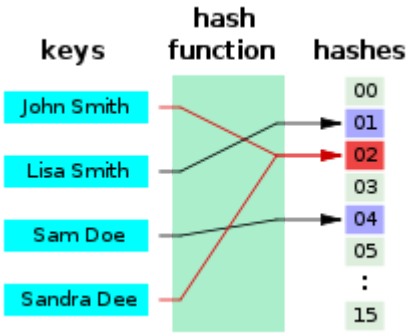
WIKIPEDIA

Hash function

A **hash function** is any function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called **hash values**, **hash codes**, **digests**, or simply **hashes**. One use is a data structure called a hash table, widely used in computer software for rapid data lookup. Hash functions accelerate table or database lookup by detecting duplicated records in a large file. An example is finding similar stretches in DNA sequences. They are also useful in cryptography. A cryptographic hash function allows one to easily verify that some input data maps to a given hash value, but if the input data is unknown, it is deliberately difficult to reconstruct it (or equivalent alternatives) by knowing the stored hash value. This is used for assuring integrity of transmitted data, and is the building block for HMACs, which provide message authentication.



A hash function that maps names to integers from 0 to 15. There is a collision between keys "John Smith" and "Sandra Dee".

Hash functions are related to (and often confused with) checksums, check digits, fingerprints, lossy compression, randomization functions, error-correcting codes, and ciphers. Although these concepts overlap to some extent, each has its own uses and requirements and is designed and optimized differently. The Hash Keeper database maintained by the American National Drug Intelligence Center, for instance, is more aptly described as a catalogue of file fingerprints than of hash values.

Contents

Uses

Hash Tables

Hash functions are used in hash tables,[1] to quickly locate a data record (e.g., a dictionary definition) given its search key (the headword). Specifically, the hash function is used to map the search key to a list; the index gives the place in the hash table where the corresponding record should be stored. Hash tables, also, are used to implement associative arrays and dynamic sets.[2]

Typically, the domain of a hash function (the set of possible keys) is larger than its range (the number of different table indices), and so it will map several different keys to the same index. So then, each slot of a hash table is associated with (implicitly or explicitly) a set of records, rather than a single record. For this reason, each slot of a hash table is often called a *bucket*, and hash values are also called *bucket listing* or a *bucket index*.

Thus, the hash function only hints at the record's location. Still, in a half-full table, a good hash function will typically narrow the search down to only one or two entries.

People who write complete hash table implementations choose a specific hash function—such as a Jenkins hash or Zobrist hashing—and independently choose a hash-table collision resolution scheme—such as coalesced hashing, cuckoo hashing, or hopscotch hashing.

Caches

Hash functions are also used to build caches for large data sets stored in slow media. A cache is generally simpler than a hashed search table, since any collision can be resolved by discarding or writing back the older of the two colliding items. This is also used in file comparison.

Bloom filters

Hash functions are an essential ingredient of the Bloom filter, a space-efficient probabilistic data structure that is used to test whether an element is a member of a set.

Finding duplicate records

When storing records in a large unsorted file, one may use a hash function to map each record to an index into a table T, and to collect in each bucket $T[i]$ a list of the numbers of all records with the same hash value i. Once the table is complete, any two duplicate records will end up in the same bucket. The duplicates can then be found by scanning every bucket $T[i]$ which contains two or more members, fetching those records, and comparing them. With a table of appropriate size, this method is likely to be much faster than any alternative approach (such as sorting the file and comparing all consecutive pairs).

Protecting data

A hash value can be used to uniquely identify secret information. This requires that the hash function is collision-resistant, which means that it is very hard to find data that will generate the same hash value. These functions are categorized into cryptographic hash functions and provably secure hash functions. Functions in the second category are the most secure but also too slow for most practical purposes. Collision resistance is accomplished in part by generating very large hash values. For example, SHA-1, one of the most widely used cryptographic hash functions, generates 160 bit values.

Finding similar records

Hash functions can also be used to locate table records whose key is similar, but not identical, to a given key; or pairs of records in a large file which have similar keys. For that purpose, one needs a hash function that maps similar keys to hash values that differ by at most m, where m is a small integer (say, 1 or 2). If one builds a table T of all record numbers, using such a hash function, then similar records will end up in the same bucket, or in nearby buckets. Then one need only check the records in each bucket $T[i]$ against those in buckets $T[i+k]$ where k ranges between $-m$ and m.

This class includes the so-called acoustic fingerprint algorithms, that are used to locate similar-sounding entries in large collection of audio files. For this application, the hash function must be as insensitive as possible to data capture or transmission errors, and to trivial changes such as timing and volume changes, compression, etc.[3]

Finding similar substrings

The same techniques can be used to find equal or similar stretches in a large collection of strings, such as a document repository or a genomic database. In this case, the input strings are broken into many small pieces, and a hash function is used to detect potentially equal pieces, as above.

The Rabin–Karp algorithm is a relatively fast string searching algorithm that works in $O(n)$ time on average. It is based on the use of hashing to compare strings.

Geometric hashing

This principle is widely used in computer graphics, computational geometry and many other disciplines, to solve many proximity problems in the plane or in three-dimensional space, such as finding closest pairs in a set of points, similar shapes in a list of shapes, similar images in an image database, and so on. In these applications, the set of all inputs is

some sort of metric space, and the hashing function can be interpreted as a partition of that space into a grid of *cells*. The table is often an array with two or more indices (called a *grid file*, *grid index*, *bucket grid*, and similar names), and the hash function returns an index tuple. This special case of hashing is known as geometric hashing or *the grid method*. Geometric hashing is also used in telecommunications (usually under the name vector quantization) to encode and compress multi-dimensional signals.

Standard uses of hashing in cryptography

Some standard applications that employ hash functions include authentication, message integrity (using an HMAC (Hashed MAC)), message fingerprinting, data corruption detection, and digital signature efficiency.

Properties

Good hash functions, in the original sense of the term, are usually required to satisfy certain properties listed below. The exact requirements are dependent on the application, for example a hash function well suited to indexing data will probably be a poor choice for a cryptographic hash function.

Determinism

A hash procedure must be deterministic—meaning that for a given input value it must always generate the same hash value. In other words, it must be a function of the data to be hashed, in the mathematical sense of the term. This requirement excludes hash functions that depend on external variable parameters, such as pseudo-random number generators or the time of day. It also excludes functions that depend on the memory address of the object being hashed in cases that the address may change during execution (as may happen on systems that use certain methods of garbage collection), although sometimes rehashing of the item is possible.

The determinism is in the context of the reuse of the function. For example, Python adds the feature that hash functions make use of a randomized seed that is generated once when the Python process starts in addition to the input to be hashed.[4] The Python hash is still a valid hash function when used in within a single run. But if the values are persisted (for example, written to disk) they can no longer be treated as valid hash values, since in the next run the random value might differ.

Uniformity

A good hash function should map the expected inputs as evenly as possible over its output range. That is, every hash value in the output range should be generated with roughly the same probability. The reason for this last requirement is that the cost of hashing-based methods goes up sharply as the number of *collisions*—pairs of inputs that are mapped to the same hash value—increases. If some hash values are more likely to occur than others, a larger fraction of the lookup operations will have to search through a larger set of colliding table entries.

Note that this criterion only requires the value to be *uniformly distributed*, not *random* in any sense. A good randomizing function is (barring computational efficiency concerns) generally a good choice as a hash function, but the converse need not be true.

Hash tables often contain only a small subset of the valid inputs. For instance, a club membership list may contain only a hundred or so member names, out of the very large set of all possible names. In these cases, the uniformity criterion should hold for almost all typical subsets of entries that may be found in the table, not just for the global set of all possible entries.

In other words, if a typical set of m records is hashed to n table slots, the probability of a bucket receiving many more than m/n records should be vanishingly small. In particular, if m is less than n, very few buckets should have more than one or two records. (In an ideal "perfect hash function", no bucket should have more than one record; but a small number of collisions is virtually inevitable, even if n is much larger than m – see the birthday paradox).

When testing a hash function, the uniformity of the distribution of hash values can be evaluated by the chi-squared test.

Defined range

It is often desirable that the output of a hash function have fixed size (but see below). If, for example, the output is constrained to 32-bit integer values, the hash values can be used to index into an array. Such hashing is commonly used to accelerate data searches.[5] On the other hand, cryptographic hash functions produce much larger hash values, in order to ensure the computational complexity of brute-force inversion.[2] For example, SHA-1, one of the most widely used cryptographic hash functions, produces a 160-bit value.

Producing fixed-length output from variable length input can be accomplished by breaking the input data into chunks of specific size. Hash functions used for data searches use some arithmetic expression which iteratively processes chunks of the input (such as the characters in a string) to produce the hash value.[5] In cryptographic hash functions, these chunks are processed by a one-way compression function, with the last chunk being padded if necessary. In this case, their size, which is called *block size*, is much bigger than the size of the hash value.[2] For example, in SHA-1, the hash value is 160 bits and the block size 512 bits.

Variable range

In many applications, the range of hash values may be different for each run of the program, or may change along the same run (for instance, when a hash table needs to be expanded). In those situations, one needs a hash function which takes two parameters—the input data z, and the number n of allowed hash values.

A common solution is to compute a fixed hash function with a very large range (say, 0 to $2^{32} - 1$), divide the result by n, and use the division's remainder. If n is itself a power of 2, this can be done by bit masking and bit shifting. When this approach is used, the hash function must be chosen so that the result has fairly uniform distribution between 0 and $n - 1$, for any value of n that may occur in the application. Depending on the function, the remainder may be uniform only for certain values of n, e.g. odd or prime numbers.

We can allow the table size n to not be a power of 2 and still not have to perform any remainder or division operation, as these computations are sometimes costly. For example, let n be significantly less than 2^b. Consider a pseudorandom number generator (PRNG) function P(key) that is uniform on the interval $[0, 2^b - 1]$. A hash function uniform on the interval [0, n-1] is $n\,P(\text{key})/2^b$. We can replace the division by a (possibly faster) right bit shift: $nP(\text{key}) >> b$.

Variable range with minimal movement (dynamic hash function)

When the hash function is used to store values in a hash table that outlives the run of the program, and the hash table needs to be expanded or shrunk, the hash table is referred to as a dynamic hash table.

A hash function that will relocate the minimum number of records when the table is – where z is the key being hashed and n is the number of allowed hash values – such that $H(z,n + 1) = H(z,n)$ with probability close to $n/(n + 1)$.

Linear hashing and spiral storage are examples of dynamic hash functions that execute in constant time but relax the property of uniformity to achieve the minimal movement property.

Extendible hashing uses a dynamic hash function that requires space proportional to n to compute the hash function, and it becomes a function of the previous keys that have been inserted.

Several algorithms that preserve the uniformity property but require time proportional to n to compute the value of $H(z,n)$ have been invented.

A hash function with minimal movement is especially useful in distributed hash tables.

Data normalization

In some applications, the input data may contain features that are irrelevant for comparison purposes. For example, when looking up a personal name, it may be desirable to ignore the distinction between upper and lower case letters. For such data, one must use a hash function that is compatible with the data equivalence criterion being used: that is, any two inputs that are considered equivalent must yield the same hash value. This can be accomplished by normalizing the input before hashing it, as by upper-casing all letters.

Continuity

"A hash function that is used to search for similar (as opposed to equivalent) data must be as continuous as possible; two inputs that differ by a little should be mapped to equal or nearly equal hash values."[6]

Note that continuity is usually considered a fatal flaw for checksums, cryptographic hash functions, and other related concepts. Continuity is desirable for hash functions only in some applications, such as hash tables used in Nearest neighbor search.

Non-invertible

In cryptographic applications, hash functions are typically expected to be practically non-invertible, meaning that it is not realistic to reconstruct the input datum x from its hash value $h(x)$ alone without spending great amounts of computing time (see also One-way function).

Hash function algorithms

For most types of hashing functions, the choice of the function depends strongly on the nature of the input data, and their probability distribution in the intended application.

Trivial hash function

If the data to be hashed is small enough, one can use the data itself (reinterpreted as an integer) as the hashed value. The cost of computing this "trivial" (identity) hash function is effectively zero. This hash function is perfect, as it maps each input to a distinct hash value.

The meaning of "small enough" depends on the size of the type that is used as the hashed value. For example, in Java, the hash code is a 32-bit integer. Thus the 32-bit integer `Integer` and 32-bit floating-point `Float` objects can simply use the value directly; whereas the 64-bit integer `Long` and 64-bit floating-point `Double` cannot use this method.

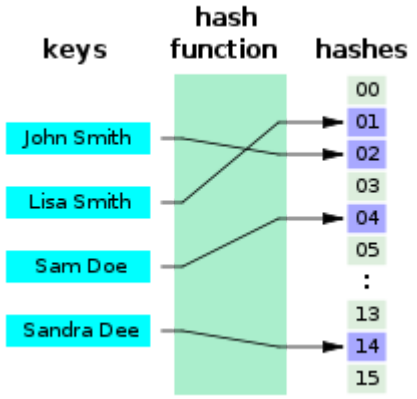
Other types of data can also use this perfect hashing scheme. For example, when mapping character strings between upper and lower case, one can use the binary encoding of each character, interpreted as an integer, to index a table that gives the alternative form of that character ("A" for "a", "8" for "8", etc.). If each character is stored in 8 bits (as in extended ASCII[7]

or ISO Latin 1), the table has only 2^8 = 256 entries; in the case of Unicode characters, the table would have 17×2^{16} = 1 114 112 entries.

The same technique can be used to map two-letter country codes like "us" or "za" to country names (26^2 = 676 table entries), 5-digit zip codes like 13083 to city names (100 000 entries), etc. Invalid data values (such as the country code "xx" or the zip code 00000) may be left undefined in the table or mapped to some appropriate "null" value.

Perfect hashing

A hash function that is injective—that is, maps each valid input to a different hash value—is said to be **perfect**. With such a function one can directly locate the desired entry in a hash table, without any additional searching.



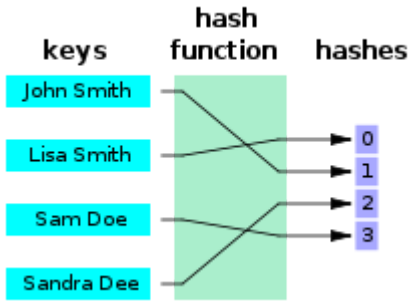
A perfect hash function for the four names shown

Minimal perfect hashing

A perfect hash function for n keys is said to be **minimal** if its range consists of n *consecutive* integers, usually from 0 to $n-1$. Besides providing single-step lookup, a minimal perfect hash function also yields a compact hash table, without any vacant slots. Minimal perfect hash functions are much harder to find than perfect ones with a wider range.

Hashing uniformly distributed data

If the inputs are bounded-length strings and each input may independently occur with uniform probability (such as telephone numbers, car license plates, invoice numbers, etc.), then a hash function needs to map roughly the same number of inputs to each hash value. For instance, suppose that each input is an integer z in the range 0 to $N-1$, and the output must be an integer h in the range 0 to $n-1$, where N is much larger than n. Then the hash function could be $h = z$ **mod** n (the remainder of z divided by n), or $h = (z \times n) \div N$ (the value z scaled down by n/N and truncated to an integer), or many other formulas.



A minimal perfect hash function for the four names shown

Hashing data with other distributions

These simple formulas will not do if the input values are not equally likely, or are not independent. For instance, most patrons of a supermarket will live in the same geographic area, so their telephone numbers are likely to begin with the same 3 to 4 digits. In that case, if m is 10000 or so, the division formula $(z \times m) \div M$, which depends mainly on the leading digits, will generate a lot of collisions; whereas the remainder formula z **mod** m, which is quite sensitive to the trailing digits, may still yield a fairly even distribution.

Hashing variable-length data

When the data values are long (or variable-length) character strings—such as personal names, web page addresses, or mail messages—their distribution is usually very uneven, with complicated dependencies. For example, text in any natural language has highly non-uniform distributions of characters, and character pairs, very characteristic of the language. For

such data, it is prudent to use a hash function that depends on all characters of the string—and depends on each character in a different way.

In cryptographic hash functions, a Merkle–Damgård construction is usually used. In general, the scheme for hashing such data is to break the input into a sequence of small units (bits, bytes, words, etc.) and combine all the units *b*[1], *b*[2], ..., *b*[*m*] sequentially, as follows

```
S ← S0;                         // Initialize the state.
for k in 1, 2, ..., m do        // Scan the input data units:
  S ← F(S, b[k]);               // Combine data unit k into the state.
return G(S, n)                  // Extract the hash value from the state.
```

This schema is also used in many text checksum and fingerprint algorithms. The state variable *S* may be a 32- or 64-bit unsigned integer; in that case, *S*0 can be 0, and *G*(*S*,*n*) can be just *S* **mod** *n*. The best choice of *F* is a complex issue and depends on the nature of the data. If the units *b*[*k*] are single bits, then *F*(*S*,*b*) could be, for instance

```
if highbit(S) = 0 then
  return 2 * S + b
else
  return (2 * S + b) ^ P
```

Here *highbit*(*S*) denotes the most significant bit of *S*; the '*' operator denotes unsigned integer multiplication with lost overflow; '^' is the bitwise exclusive or operation applied to words; and *P* is a suitable fixed word.[8]

Special-purpose hash functions

In many cases, one can design a special-purpose (heuristic) hash function that yields many fewer collisions than a good general-purpose hash function. For example, suppose that the input data are file names such as `FILE0000.CHK`, `FILE0001.CHK`, `FILE0002.CHK`, etc., with mostly sequential numbers. For such data, a function that extracts the numeric part *k* of the file name and returns *k* **mod** *n* would be nearly optimal. Needless to say, a function that is exceptionally good for a specific kind of data may have dismal performance on data with different distribution.

Rolling hash

In some applications, such as substring search, one must compute a hash function *h* for every *k*-character substring of a given *n*-character string *t*; where *k* is a fixed integer, and *n* is greater than *k*. The straightforward solution, which is to extract every such substring *s* of *t* and compute *h*(*s*) separately, requires a number of operations proportional to *k*·*n*. However, with the proper choice of *h*, one can use the technique of rolling hash to compute all those hashes with an effort proportional to *k* + *n*.

Universal hashing

A universal hashing scheme is a randomized algorithm that selects a hashing function *h* among a family of such functions, in such a way that the probability of a collision of any two distinct keys is 1/*n*, where *n* is the number of distinct hash values desired—independently of the two keys. Universal hashing ensures (in a probabilistic sense) that the hash function application will behave as well as if it were using a random function, for any distribution of the input data. It will, however, have more collisions than perfect hashing and may require more operations than a special-purpose hash function. See also unique permutation hashing.[9]

Hashing with checksum functions

One can adapt certain checksum or fingerprinting algorithms for use as hash functions. Some of those algorithms will map arbitrarily long string data *z*, with any typical real-world distribution—no matter how non-uniform and dependent—to a 32-bit or 64-bit string, from which one can extract a hash value in 0 through $n - 1$.

This method may produce a sufficiently uniform distribution of hash values, as long as the hash range size n is small compared to the range of the checksum or fingerprint function. However, some checksums fare poorly in the avalanche test, which may be a concern in some applications. In particular, the popular CRC32 checksum provides only 16 bits (the higher half of the result) that are usable for hashing. Moreover, each bit of the input has a deterministic effect on each bit of the CRC32 output; that is, one can tell without looking at the rest of the input which bits of the output will flip if the input bit is flipped, so care must be taken to use all 32 bits when computing the hash from the checksum.[10]

Multiplicative hashing

Multiplicative hashing is a simple type of hash function often used by teachers introducing students to hash tables.[11] Multiplicative hash functions are simple and fast, but have higher collision rates in hash tables than more sophisticated hash functions.[12]

In many applications, such as hash tables, collisions make the system a little slower but are otherwise harmless. In such systems, it is often better to use hash functions based on multiplication—such as MurmurHash and the SBoxHash—or even simpler hash functions such as CRC32—and tolerate more collisions; rather than use a more complex hash function that avoids many of those collisions but takes longer to compute.[12] Multiplicative hashing is susceptible to a "common mistake" that leads to poor diffusion—higher-value input bits do not affect lower-value output bits.[13]

Hashing with cryptographic hash functions

Some cryptographic hash functions, such as SHA-1, have even stronger uniformity guarantees than checksums or fingerprints, and thus can provide very good general-purpose hashing functions.

In ordinary applications, this advantage may be too small to offset their much higher cost.[14] However, this method can provide uniformly distributed hashes even when the keys are chosen by a malicious agent. This feature may help to protect services against denial of service attacks.

Hashing by nonlinear table lookup

Tables of random numbers (such as 256 random 32-bit integers) can provide high-quality nonlinear functions to be used as hash functions or for other purposes such as cryptography. The key to be hashed is split into 8-bit (one-byte) parts, and each part is used as an index for the nonlinear table. The table values are then added by arithmetic or XOR addition to the hash output value. Because the table is just 1024 bytes in size, it fits into the cache of modern microprocessors and allows very fast execution of the hashing algorithm. As the table value is on average much longer than 8 bits, one bit of input affects nearly all output bits.

This algorithm has proven to be very fast and of high quality for hashing purposes (especially hashing of integer-number keys).

Efficient hashing of strings

Modern microprocessors will allow for much faster processing, if 8-bit character strings are not hashed by processing one character at a time, but by interpreting the string as an array of 32 bit or 64 bit integers and hashing/accumulating these "wide word" integer values by means of arithmetic operations (e.g. multiplication by constant and bit-shifting). The remaining characters of the string which are smaller than the word length of the CPU must be handled differently (e.g. being processed one character at a time).

This approach has proven to speed up hash code generation by a factor of five or more on modern microprocessors of a word size of 64 bit.

Another approach[15] is to convert strings to a 32 or 64 bit numeric value and then apply a hash function. One method that avoids the problem of strings having great similarity ("Aaaaaaaaaa" and "Aaaaaaaaab") is to use a Cyclic redundancy check (CRC) of the string to compute a 32- or 64-bit value. While it is possible that two different strings will have the same CRC, the likelihood is very small and only requires that one check the actual string found to determine whether one has an exact match. CRCs will be different for strings such as "Aaaaaaaaaa" and "Aaaaaaaaab". Although CRC codes can be used as hash values,[16] they are not cryptographically secure, because they are not collision-resistant.[17]

Locality-sensitive hashing

Locality-sensitive hashing (LSH) is a method of performing probabilistic dimension reduction of high-dimensional data. The basic idea is to hash the input items so that similar items are mapped to the same buckets with high probability (the number of buckets being much smaller than the universe of possible input items). This is different from the conventional hash functions, such as those used in cryptography, as in this case the goal is to minimize the probability of "collision" of every item.[18]

One example of LSH is MinHash algorithm used for finding similar documents (such as web-pages):

Let h be a hash function that maps the members of A and B to distinct integers, and for any set S define $h_{\min}(S)$ to be the member x of S with the minimum value of $h(x)$. Then $h_{\min}(A) = h_{\min}(B)$ exactly when the minimum hash value of the union $A \cup B$ lies in the intersection $A \cap B$. Therefore,

$$\Pr[h_{\min}(A) = h_{\min}(B)] = J(A,B).$$ where J is Jaccard index.

In other words, if r is a random variable that is one when $h_{\min}(A) = h_{\min}(B)$ and zero otherwise, then r is an unbiased estimator of $J(A,B)$, although it has too high a variance to be useful on its own. The idea of the MinHash scheme is to reduce the variance by averaging together several variables constructed in the same way.

Origins of the term

The term "hash" offers a natural analogy with its non-technical meaning (to "chop" or "make a mess" out of something), given how hash functions scramble their input data to derive their output.[19] In his research for the precise origin of the term, Donald Knuth notes that, while Hans Peter Luhn of IBM appears to have been the first to use the concept of a hash function in a memo dated January 1953, the term itself would only appear in published literature in the late 1960s, on Herbert Hellerman's *Digital Computer System Principles*, even though it was already widespread jargon by then.[20]

List of hash functions

- NIST hash function competition
- MD5
- Bernstein hash[21]

- Fowler-Noll-Vo hash function (32, 64, 128, 256, 512, or 1024 bits)
- Jenkins hash function (32 bits)
- Pearson hashing (8 or more bits)
- Zobrist hashing

See also

- Comparison of cryptographic hash functions
- Distributed hash table
- Identicon
- Low-discrepancy sequence
- PhotoDNA
- Transposition table

References

1. Konheim, Alan (2010). "7. HASHING FOR STORAGE: DATA MANAGEMENT" (https://books.google.com/books?id=mU6fpT1sXCoC&lpg=PT12&pg=PT174). *Hashing in Computer Science: Fifty Years of Slicing and Dicing*. Wiley-Interscience. ISBN 9780470344736.
2. Menezes, Alfred J.; van Oorschot, Paul C.; Vanstone, Scott A (1996). *Handbook of Applied Cryptography*. CRC Press. ISBN 0849385237.
3. "Robust Audio Hashing for Content Identification by Jaap Haitsma, Ton Kalker and Job Oostveen" (http://citeseer.ist.psu.edu/rd/11787382%2C504088%2C1%2C0.25%2CDownload/http://citeseer.ist.psu.edu/cache/papers/cs/25861/http:zSzzSzwww.extra.research.philips.comzSznatlabzSzdownloadzSzaudiofpzSzcbmi01audiohashv1.0.pdf/haitsma01robust.pdf)
4. "3. Data model — Python 3.6.1 documentation" (https://docs.python.org/3/reference/datamodel.html#object.__hash__). *docs.python.org*. Retrieved 2017-03-24.
5. Sedgewick, Robert (2002). "14. Hashing". *Algorithms in Java* (3 ed.). Addison Wesley. ISBN 978-0201361209.
6. "Fundamental Data Structures – Josiang p.132" (https://www.scribd.com/doc/199819816/Fundamental-Data-Structures). Retrieved May 19, 2014.
7. Plain ASCII is a 7-bit character encoding, although it is often stored in 8-bit bytes with the highest-order bit always clear (zero). Therefore, for plain ASCII, the bytes have only 2^7 = 128 valid values, and the character translation table has only this many entries.
8. Broder, A. Z. (1993). "Some applications of Rabin's fingerprinting method". *Sequences II: Methods in Communications, Security, and Computer Science*. Springer-Verlag. pp. 143–152.
9. Shlomi Dolev, Limor Lahiani, Yinnon Haviv, "Unique permutation hashing", Theoretical Computer Science Volume 475, 4 March 2013, Pages 59–65.
10. Bret Mulvey, *Evaluation of CRC32 for Hash Tables (http://bretmulvey.com/hash/8.html)*, in *Hash Functions (http://bretmulvey.com/hash/)*. Accessed April 10, 2009.
11. Knuth. "The Art of Computer Programming". Volume 3: "Sorting and Searching". Section "6.4. Hashing".
12. Peter Kankowski. "Hash functions: An empirical comparison" (http://www.strchr.com/hash_functions).
13. "CS 3110 Lecture 21: Hash functions" (http://www.cs.cornell.edu/courses/cs3110/2008fa/lectures/lec21.html). Section "Multiplicative hashing".
14. Bret Mulvey, *Evaluation of SHA-1 for Hash Tables (http://bretmulvey.com/hash/9.html)*, in *Hash Functions (http://bretmulvey.com/hash/)*. Accessed April 10, 2009.
15. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.7520 Performance in Practice of String Hashing Functions
16. Peter Kankowski. "Hash functions: An empirical comparison" (http://www.strchr.com/hash_functions).

17. Cam-Winget, Nancy; Housley, Russ; Wagner, David; Walker, Jesse (May 2003). "Security Flaws in 802.11 Data Link Protocols" (http://www.cs.berkeley.edu/~daw/papers/wireless-cacm.pdf) (PDF). *Communications of the ACM*. **46** (5): 35–39. doi:10.1145/769800.769823 (https://doi.org/10.1145%2F769800.769823).
18. A. Rajaraman and J. Ullman (2010). "Mining of Massive Datasets, Ch. 3." (http://infolab.stanford.edu/~ullman/mmds.html).
19. Knuth, Donald E. (2000). *Sorting and searching* (2. ed., 6. printing, newly updated and rev. ed.). Boston [u.a.]: Addison-Wesley. p. 514. ISBN 0-201-89685-0.
20. Knuth, Donald E. (2000). *Sorting and searching* (2. ed., 6. printing, newly updated and rev. ed.). Boston [u.a.]: Addison-Wesley. pp. 547–548. ISBN 0-201-89685-0.
21. "Hash Functions" (http://www.cse.yorku.ca/~oz/hash.html). *cse.yorku.ca*. September 22, 2003. Retrieved November 1, 2012. "the djb2 algorithm (k=33) was first reported by dan bernstein many years ago in comp.lang.c."

External links

- Calculate hash of a given value (http://tools.timodenk.com/?p=hash-function) by Timo Denk
- Hash Functions and Block Ciphers (http://burtleburtle.net/bob/hash/index.html) by Bob Jenkins
- The Goulburn Hashing Function (https://www.webcitation.org/query?url=http://www.geocities.com/drone115b/Goulburn06.pdf&date=2009-10-25+21:06:51) (PDF) by Mayur Patel
- Hash Function Construction for Textual and Geometrical Data Retrieval (http://herakles.zcu.cz/~skala/PUBL/PUBL_2010/2010_WSEAS-Corfu_Hash-final.pdf) Latest Trends on Computers, Vol.2, pp. 483–489, CSCC conference, Corfu, 2010