

Technologies for Greener Internet of Things Systems

Nikolaos Doukas

Abstract The Internet of Things processing paradigm arises from the ever increasing tendency for decentralization of the hardware used for processing. Innovative services combine the use of cloud and mobile computational resources in order to provide intuitive and agreeable user experiences. Furthermore, the internet of things paradigm extends to include sensor networks and distributed data collection infrastructures. The essential implication is the necessity to handle big volumes of possibly streaming data. For the case of streaming data, storage in a data warehouse is in general not feasible; a real-time initial processing phase is necessary that will sample, select, organize or summarize the available information. Such systems are categorized in the Big Data Processing paradigm. The two paradigms are correlated, since Internet of Things applications are beneficial when there are “a lot of Things” and hence the amount of data classifies as “Big Data”. The internet of things information processing paradigm is one of the rare occasions where the demand for Green Computing systems does not compete with the need for performance. Indeed, the volumes of data that need to be processed are overwhelming to such an extent that approaches which use unlimited amounts of power, for processing, storage and the associated hardware cooling are simply not feasible. Additionally, remotely operating components that form the distributed processing network, such as smart mobile devices or remote interconnected sensors, need to be Green if they are to be viable. This research focuses on algorithmic developments that make the real-time collection, summarization, analysis and decision making based on streaming data greener.

Keywords Green IT engineering • Internet of Things • Data summarization • Hash functions • Data mining

N. Doukas (✉)

Hellenic Army Academy, Varis—Koropiou Avenue, 116 72 Vari, Greece
e-mail: ndoukas@sse.gr

1 Introduction

The operation of a key based search is considered as a fundamental for a variety of data processing routines, such as real-time collection, summarization, analysis and decision making. In a significant number of widely used practical applications, the relative computational load of search operations may be as large as 80% of the total [1] implying an analogous overhead in energy consumption. The progressive development of information systems and of information integration determines a dynamic increase of the volume of the key indices upon which searches are performed [2].

A particular field of application of key based searching, along with other operations based on keys is the Big Data Processing paradigm, especially in Data Mining applications that attempt to mine in Internet of Things Big Data environments. The term Data Mining in this case, implies a process by which a representative model is sought that adequately describes data sets whose size renders them unmanageable via the classical sequential or batch processing concepts of algorithmic design. The representative requirement that is set for such models inherently dictates that the type of models that are sought are statistical models.

The process of mining into data in streaming form and seeking to develop the corresponding model, may be described as “*learning the data*”, i.e. following the changes in the characteristics of the data and continuously updating critical parameter estimates that lead to decision making. The situations where classical machine learning algorithms, such as support vector machines, decision trees and Hidden Markov Models, may be applied and produce satisfactory results are limited and beyond the scope of this study. There exists however a wide variety of applications where machine learning is not applicable, given the nature of the problem or present no advantages in performance when compared to semi-heuristic, real-time processing algorithms based on statistical measures. The problems of interest that fall in this type of applications may be in general classified in two categories:

- Problems where a summary description of the data is required, e.g. when tracking the progress of a measurement or a statistical measure related to the data, like a mean or a higher order moment, or trying to obtain a representative subset of the data.
- Problems where characteristic features need to be determined that allow the determination of data values that usually appear in groups or data values that are similar, in some sense. Depending on the context, this class of problems may also be described as seeking to determine outlying values in the data stream.

Summarization may be described as the problem of producing a model of the data that groups similar values into the same group. The number of different groups, the limits that separate different groups and the determination of similarity is performed automatically, via the use of purpose designed algorithms. Summarization algorithms may in some cases produce a single measurement that determines the

merit or the rank of a complex data value or may also track the progress of a series of measurements that enable the process of decision making.

Characteristic features correspond to data values that present the extreme variations within a data set and hence may be described as outstanding or as outliers. Applications of being able to determine characteristics features have multiple scopes. It may permit us to describe a complicated interconnections between measurement vectors observed in different situations by only a limited number of values. In the case of collections of objects being observed, the object of the exercise becomes to identify sets of these objects that frequently appear together. The determination of these sets has applications in a variety of applications ranging from marketing to satellite surveillance. Another variation of the same problem is the determination of similarity, based on previous observations. Hence if an object presents a particular behavior and similar objects have exhibited a specific course of evolution after the same behavior, then it may be decided that the original object is likely to exhibit a similar evolution later on. E.g. a customer that has made a certain purchase is likely to proceed to other specific purchases, if similar customers have been observed to perform a similar set of purchases.

A fundamental problem in all the above applications is the requirement for accelerating searches based on a key, as well as the selection of this key, so that it exhibits the required characteristics, given the nature of the data. The necessity for scalability in IoT systems and increase of the volumes of the data that are required to be processed, has as a consequence imposed stricter requirements for the efficiency of search procedure results. A significant justification for requiring ever more efficient systems is the contemporary call for greener IT engineering systems. More specifically, a large proportion of IoT big data processing systems, in which key search is actively used, operate in real time conditions, on hardware that may not thoughtlessly consume energy. Based on these facts, the development of energy saving key based search technologies is not only necessary in order to satisfy environmentalists or to abide by mandatory environmental regulations, but also in order for the system to be scalable and viable.

This chapter is organized as follows. In the following section an analysis of existing search technologies is presented, that illustrates how computational complexity increases exponentially with data size. The importance of suitably designed hash functions for achieving computational and energy efficiency in IoT big data processing is hence explained. Following that, particular references are made to IoT and other big data analysis applications that heavily rely on hash functions for feasibility of their implementation. It is hence shown that well designed hash functions may promote greener IT engineering. A recently proposed class of hash functions is then presented that provides suitable statistical separation properties while maintaining a linear rate of increase of the computational effort involved. Subsequently, the application of this class of functions in particular big data processing problems is explained in detail and benefits in terms of green performance are highlighted. Conclusions are hence drawn for the mathematical specifications required for hash function design algorithms.

2 Analysis of Existing Search Technologies

An important factor for improving the efficiency of key based search, the key element that will determine its greenness in terms of energy consumption, is the incorporation of the multilevel memory organization of modern computational systems into search algorithms. In current conditions, where the volume of indices is constantly increasing and the efficiency requirements upon the search are becoming ever more demanding, the applicability of binary trees and B trees is significantly reduced, given the dependence of the search time on the volume of the key index. Apart from that, such search procedures are anyway less effective in the case of multilevel memory organization.

The fastest current key-based search method available is the associative memory that may be implemented in either hardware (content addressable memory) or in software (hash memory). The basic advantage of associative search is considered to be the fact that the search time is independent of the volume of the key index. For most applications, hash addressing is considered to be the most efficient approach. There exist practical applications of key-based search, where the key index is considered permanent or quasi-permanent (the computational load required for the key based search procedures exceeds by several orders of magnitude the computational load of the procedures for changing the search index). Such applications include principally pattern recognition systems, electronic translation systems, user identification and authentication in systems supporting remote access and a large proportion of database applications.

During the hash search in permanent key indices, it is possible to determine a one-way hash transformation that eliminates collisions. In bibliography, this class of methods is referred to as perfect hash addressing [1]. The fundamental advantage of perfect hash addressing is the absence of collisions, i.e. the key search time is determined by the time required for a single memory access. This permits the search schemes to attain maximum search speed, independent of the volume of the search index [2].

The exercise of determining a hash transformation for perfect hash addressing exhibits exponential complexity. For the completion of this exercise, a series of methods have been proposed in bibliography [1–6]. The disadvantages of these methods are that they do not take into account the multilevel organization of memories and that they do not allow changing of the keys during operation. The purpose of this research is the modification of the organization of hash searches so that it becomes oriented to the quasi-permanent nature of the key index. Additionally, developments are sought in the mathematical model of such hash searches for the optimization of its characteristics.

A hash search organization was recently proposed [7] that uses quasi-permanent keys in conjunction with perfect hash addressing and probing. The particular characteristic of the proposed method is the fact that the model is oriented to a multilevel organization of the memory of modern computational systems.

A mathematical model of the hash search in multilevel memory has been developed that allows the optimization of the hash memory parameters during the design

3 Hash Functions in IoT Big Data Mining Applications

Data mining, as it has already been defined and described in a previous section is a topic that has attracted significant research interest in the context of mining when the amount of data is characterized as “Big”, i.e. despite the evolution of processors, multiprocessor information processing systems and the ample availability of storage facilities, it is still infeasible to plan the processing, in whatever form of all the data. It may be impossible to process the data in real-time, at the rate at which they arrive or it may be economically or physically impractical to store all the data for online processing. An increasingly important inhibiting factor are concerns about the environmental impact of performing all the processing required in order to sufficiently process such amounts of data. This section summarizes common required functionalities of IoT big data mining systems designed according to the green IT paradigm [8–10], where all the above restrictions may be satisfied via the use of appropriate search techniques that are based on suitably designed hash functions.

3.1 Shingle Hashing

A large class of applications need to process observations that can be mapped to a group of objects appearing together. An example of such application is the processing of texts in the form of characters. Typically, the text will be segmented into groups of characters of equal length, called shingles [8]. Typically, since objects are not arithmetic values, an encoding is required for representing those objects that is not based on the frequency of appearance of the corresponding shingles, but rather on the size of the domain from which sample observed objects are drawn. Hence, in the case of character shingles, the representation of independent characters may require 8–16 bits per character, while the information content carried by the corresponding shingle composed of these characters is significantly less. If the shingle size is for example 9 characters, the processing may be performed by using a hash function that will map the minimum size of $8 \times 9 = 72$ bits of each shingle to 32 bits, without loss of performance [8]. This is due to statistical measurements made concerning the English language, showing that there are about 20 frequent characters, resulting to $20^4 \approx 2^{16}$ possible 32-bit (4 byte) shingles. The hashing operation can be hence seen as a classification operation with the number of categories (buckets) being significantly larger than the number of possible different samples. The pre-requisite is that the hash function is collision-free, i.e. it will not classify different shingles into the same bucket, despite of the existence of empty buckets.

In the case of document processing, the use of this type of hashing is an enabling technique for assessing the similarity of documents that are of sizes that are impossible to store into memory in their totality simultaneously.

3.2 Min Hashing

As the numbers of the objects under consideration increases, it becomes increasingly infeasible to store even all the possible the compressed shingle-based representations, corresponding to the pre-processing of all possible objects, in memory for drawing global conclusions.

In order to overcome this problem, an alternative representation of an object needs to be defined. Consider a set of 5 possible shingles {a, b, c, d, e} and objects consisting of collections (sets) of these shingles. It is then possible to represent these objects using a binary matrix, such as the one shown in Table 1.

A value of 1 in a cell of the table shown in Table 1, means that the shingle corresponding to that row is part of the object of the corresponding column. The representation of Table 1 is therefore interpreted that object $S_1 = \{b, e\}$, $S_2 = \{d, e\}$, $S_3 = \{b, d\}$, $S_4 = \{a, b, c\}$ and $S_5 = \{c, d, e\}$. The set of all objects is hence represented, as far as the search for similarity is concerned as a binary matrix of size $N \times M$, where N the number of possible shingles and M the total number of objects being processed. Furthermore, this table is, for large N and M , in general a sparse matrix and the corresponding savings in the required amount of memory and processing effort required may hence be achieved. It is trivial to deduce that two equal objects will exhibit equal columns using this representation.

Consider a permutation of the rows of Table 1. The minhash signature of each object of this permutation is defined as the index of the first row of the permuted Table 1 where a value of 1 appears for this particular object. An example permutation is shown in Table 2.

Table 1 Representation of an object as a binary table

Object		S_1	S_2	S_3	S_4	S_5
Shingle	a	0	0	0	1	0
	b	1	0	1	1	0
	c	0	0	0	1	1
	d	0	1	1	0	1
	e	1	1	0	0	1

Table 2 Permutation of the rows of the binary table

Object		S_1	S_2	S_3	S_4	S_5
Shingle	a	1	1	0	0	1
	b	0	1	1	0	1
	c	0	0	0	1	1
	d	0	0	0	1	0
	e	1	0	1	1	0

For this permutation, the minhash for S_1 is 1, for S_2 is 1, for S_3 is 2 for S_4 is 3 and for S_5 is 1.

Algorithm 1: Minhash Signature Similarity Calculation Based on Hash Functions

1. Generate the binary shingle table T
2. Generate a table S of size $K \times M$, setting all cells to ∞
3. Generate K hash functions $h_1 \dots h_K$ mapping K items $\rightarrow K$ buckets
4. For each value i from 0 to $K - 1$
 - (a) Calculate $h_1(i), h_2(i), \dots, h_K(i)$
5. For each column j from 1 to $M - 1$
 - (a) For each value i from 0 to $K - 1$
 - (i) If $T(i, j) = 0$, do nothing
 - (ii) If $T(i, j) \neq 0$, $S(i, j) = \min\{S(i, j), h_i(r)\}$.

The minhash signature for each object is the set of the minhashes of that object for all possible permutations of the table. It can be shown [8] *that the probability of two objects exhibiting the same minhash signature, is a similarity measure for the two objects that approximates quantitative similarity measures based on vector distances.*

In practice, the number of shingles will be large and it is therefore computationally expensive to examine all the permutations of the rows of the table. Additionally, it becomes computationally expensive, even to generate random permutations of the rows. This problem is overcome by defining a number $K \ll N$, of permutations to be considered. An algorithm for estimating the similarity if the minhash signatures of the objects may then be estimated as shown in Algorithm 1. The success of Algorithm 1 is critically dependent on the hash signatures not presenting collisions.

3.3 Locality Sensitive Hashing

The Technique of Sect. 3.2 becomes infeasible as the table T of Algorithm 1 becomes larger, with increasing numbers of objects. Even though the amount of information, or the number of bits, that need to be processed per object is small, the number of pairs that exist is infeasibly large. As a consequence, the number of comparisons that needs to be performed is correspondingly prohibitive for computation in some cases with reasonable resources, while in other cases the computations cannot be realistically implemented. For example, with 10^6 objects to be

compared, the number of comparisons is of the order of 10^{12} and even with a hash signature of 1 KB, it would take about 6 days to complete in a powerful modern personal computer [8].

The solution taken is a two pass approach; first the binary signature table is split into horizontal bands, i.e. into bands of groups of shingles. Subsequently, smaller hash functions are applied to hash all objects of a band to a small number of bins. The same process is applied to all the bands. Objects that hashed into the same bucket in multiple bands are considered to be candidate pairs and those are compared using the full minhashing procedure described in the previous section.

The hash functions used for the first pass of the Locality Sensitive Hashing, map many objects in the same buckets, i.e. present a large number of collisions. However these collisions need to satisfy similarity conditions. If $d(x, y)$ is the distance between the subset of shingles examined for a specific band and pair of objects then:

1. If $d(x, y) \leq d_1$, the probability of x and y hashing to the same bucket is at least p_1
2. If $d(x, y) \geq d_1$, the probability of x and y hashing to the same bucket is at most p_2

Suitable determination of the values of p_1 and p_2 attain a balance between the number of erroneous proclamations of candidate pairs and the number of misses, i.e. true candidate pairs are classified as irrelevant.

3.4 Stream Data Sampling

When a stream of data is necessary to be processed in real time and the number of unique elements is large, unknown and random, a common requirement is to be able to derive a representative sample of the incoming values. Assuming for example that the incoming tuples are $\{object_identity, variable_value\}$, where the object is one of the unique entities and the variable is some observation, a natural inclination would be to select a portion of the objects, such that storage and processing is economically or environmentally viable and keep only tuples originating from these objects. This could be performed by observing the following procedure:

- For each incoming tuple
 - Draw a random number in the range $1 \dots N$, where N is the number of unique entities
 - If the number drawn is less than M , where M is an integer such that M/N is equal to the portion of unique objects that are to be maintained, appropriately process the tuple

However, this approach requires that the numbers and identities of the unique objects are known. This assumption is severely restricting, since applications of

interest usually deal with very large, unknown and rapidly varying numbers of unique objects (e.g. the unique users visiting an e-shop). Additionally, it may be shown that such an approach would fail when a tuple appear multiple times [8], an occasion that is of particular interest.

This problem may be addressed by use of suitable hash functions. The number of buckets is determined such that it is feasible to select the proportion of objects required. If for example a ratio of M/N of the overall tuples need to be maintained, then the number of buckets may be assigned to N . A suitable hash function is required that will map incoming tuples to the N buckets and only buckets in the range $1 \dots N$ will be taken into consideration. Again, given that the number of unique tuples is extremely large, and the number of buckets is comparatively very small, the hash function is one that presents large numbers of collisions. However it is important that all buckets are selected with equal probability. Given this implementation, it becomes relatively easy to resize or alter the sample size or the proportion by altering the parameters or switching the hash function in real-time and waiting for a significant number of samples to arrive and any transients to die out.

3.5 *Filtering Streams*

Another problem of interest in the case of processing streaming data, is the problem of selecting incoming values that belong to a desired set. A characteristic example of this class of applications is checking e-mails for belonging in a virtuous set, e.g. a set of confirmed non-spam users [8]. Given an average size of 20 bytes per e-mail address and a possible address space of 20M valid e-mail addresses to be allowed through, it is not realistically feasible even to store all valid addresses in memory, while searching through them cannot even be considered.

The problem may be solved [8] by allocating 1 GB (approximately 1 billion bytes) of memory in order to create a bit array. A hash function will be used in order to map valid e-mail addresses to bits in the bit array. When the mapping is complete, bits that according to the hash function correspond to valid addresses have a value of 1, while all other bits have a value of zero. It is trivial to calculate that approximately 1/8th of the bits will be assigned a value of 1. During operation time, each time an incoming e-mail message is received, the address is hashed and the corresponding bit of the bit array is examined. If the bit is 1, the message is accepted, otherwise it is rejected. All need for storage is hence eliminated.

The hash function in this case maps a smaller number of keys to a large range or target space. Collisions will inevitably exist in the complementary set of values. Considering that the number of values to be rejected (in the example, e-mail addresses corresponding to spammers) is much larger than the number of zero values bits, unwanted e-mail addresses will be arranged in a many-to-few mapping. It is also understandable that some unwanted values will collide with acceptable ones. This false acceptance is not catastrophic. The catastrophic case in the context of this

application is the rejection of a valid address. The probability of this happening may be reduced as necessary by increasing the proportion of filled to empty bits.

The probabilities of false positive identifications may be reduced, provided sufficient memory space is available, by maintaining a multiple bit arrays that are populated by multiple hash functions. For a value to get through, different rules may be applied that will allow it if all bit arrays allow it, or the majority allow it or a minimum number of bit arrays allow it.

3.6 *Count Distinct Problem*

Counting the distinct values that appear in a stream of data poses some particular problems. As in the previous case of obtaining a representative data sample, it is realistically impossible either to store or even to know in advance all the possible incoming unique keys or object identities. Attempting to maintain a record of all items seen so far is also a futile endeavor. The record will, for any application of non-trivial nature, become extremely large and hence searching and inserting will rapidly become time-consuming and eventually infeasible.

Despite the unpredictability of the identities, the number of distinct values that will appear is finite and countable. It is hence possible to use a hash function that will map incoming values onto the set N of natural numbers [8]. Define as tail of a number, the number of zero value bits encountered in its binary representation, starting from the least significant bit and ending at the least significant of the bits that has a value of 1. For example, the number 24 has a binary representation of 11000 and its tail size is hence 3.

If the number of distinct values that have appeared so far in the stream is N and given a hash function without collisions that may process a variable input bit length, it may be shown [8] that:

- If $N \gg 2^r$, then the probability of observing a tail length of r approaches 1
- If $N \ll 2^r$, then the probability of observing a tail length of r approaches 0

Hence, if a maximum tail size T has been observed, then 2^T is very likely to be a correct estimate for the number of distinct objects that have appeared up to that point in time within the stream. Outliers can very easily increase the error of such estimates. If computationally feasible, maintaining a running histogram of tail sizes that have been observed may eliminate or significantly reduce the probability of outliers within the data destroying the estimate.

Outliers will still make the calculations diverge even if an attempt is made to use multiple hash functions and take the mean or median of the calculated values. This is because any error has an exponential effect on the formula 2^r . The effect of outliers may however be eliminated [8] by using multiple hash functions in groups, taking the average within each group and the median of all averages.

3.7 *First Order Statistics*

First order statistics, such as the mean and the median, are of particular interest for the majority of applications. Hash function based solutions that maintain a representative value set may be used. This set of representative values will then produce all necessary first order statistics when required.

3.8 *Frequent Object Sets*

When incoming tuples from a stream of data need to be counted, it may be the case that the available main memory is insufficient for maintaining all the counters necessary. Additionally, it is not advisable to use memory paging in order to maintain counters, since this approach will inevitably lead to page swapping that may rapidly deteriorate the performance of the processor due to thrashing. Additionally, searching for the index of the counter corresponding to a tuple value using its contents is both memory and processing power consuming. An elegant solution to this problem is again using hash functions [8]; each incoming value is hashed, and a hash value lookup table is constructed. When a new value comes in, a corresponding entry is not found in this table. It is hence created and contains the hash value and the pointer to the corresponding counter. When a previously seen value is observed, the lookup in the hash table is fruitful and the address of the corresponding counter is recovered and the counter is increased.

This method is not sufficient in all applications, as the number of unique different tuples that appear in a stream may be very large, with only a small proportion of those being frequent. In such a case it becomes unrealizable to hold a counter for each possible tuple. In this case a different approach is used that is based on the observation that for a tuple to be frequent, all the constituting unitary values also need to be frequent. Tuples are hence decomposed into their constituting values and the previously outlined method is used for separating frequent unique values. The space for the associated hash and counter tables will in general be small, since the amount necessary is linear with the number of values and not tuples.

The surplus memory may hence be used for creating a second hash table for measuring bucket counts of all possible tuples. The amount of free memory is measured, and the number of counters K that may be created in that space is determined.

- A hash function is created that maps the space of all possible tuples onto K buckets. This hash function needs to observe similarity criteria.
- All counters are initialized to zero.
- When a new tuple is observed, it is hashed and the counter corresponding to its hash value is incremented.
- Candidate frequent tuples are considered all tuples that contain only frequent values and are hashed in a bucket with a high valued counter.

A second pass is required in order to determine the actual frequent tuples, using the counting method described earlier on.

4 A Hash Search Model for Quasi-permanent Indices

This section analyses design approaches for hash search models and presents a hash function design model [7] that produces quasi-permanent indices.

The purpose of the hash search model that will be presented is to incorporate the analytic form of the dependencies between the characteristics of the hash-memory that determine its organization into the model and enable it to solve problems of optimization of the architecture of hash memory during the design phase. At the basis of the model that will be presented, lies the concept of the determination of a hash transformation $H(X)$ that ensures the mapping of a given set Ω from m keys in s pages of hash memory in such a way that the entire set of keys that are classified in each of the pages do not exceed the value $(\alpha + \delta) \cdot w$, where α is the load factor of the hash memory, δ the allowed variability of the load of a hash memory page load and $\alpha + \delta \leq 1$. The load factor α of the hash memory is defined by the relation between the set of m stored records to the maximum feasible record count $M = s \cdot w$ that is determined by the size of the memory:

$$\alpha = \frac{m}{M} = \frac{m}{s \cdot w} \quad (1)$$

As a record, one may consider the information taken as the key associated to a particular data item. The reference address of the position where the data is stored may be found in the record instead of the data. The determination of the hash transformation $H(X)$ that satisfies the above condition may be done by trial and error. As the test mechanism for the hash transformations, it is proposed that the prototype, block- based cryptographic algorithms (DES or Rijndael) be used, that incorporate the one-way cryptographic encoding using the key K , of the data D in the codeword C : $C = H_K(D)$ [11].

The key for the search data X in this case is used as input data to the cipher block whence the key K of the cipher block assumes the role of synchronization code and actually appears together with the number of the hash transformation. The resulting code C of the cipher block is divided in two parts: an h -bit packet that serves as a hash address $A_K(X)$ of the page and the remaining bits that become the hash Sign $S_K(X)$ of the key X of the search [6, 7]. Consequently the choice of the Hash transformation $H_K(X)$ is attained via the procedure of changing the key K of the cipher block.

Considering that a page may contain w keys, hash address $A_K(X)$, the page has $h = \log_2 s$ -bit codes. The hash function distributes the m keys in s groups that contain $\eta_1, \eta_2, \dots, \eta_s$ keys, since $\sum_{j=1}^s \eta_j = m$. Given that the hash transformation arranged the keys in the hash memory page, it is mandatory that the entire set of

hash addresses of each page does not exceed the maximum allowed number of $u = (\alpha + \delta) \cdot w$ records per page: $\forall j \in \{1, \dots, s\}: \eta_j \leq u$. If this is maintained then in each page of the hash memory, there is enough memory space to store $w \cdot (1 - \alpha - \delta)$ records.

Taking into account the fact that the hash transformation $H_K(X)$ arranges the hash address uniformly, then in each page there exist on average m/s hash addresses. As a theoretical model for the distribution of the Hash addresses, the most accurate model is the Bernoulli probability distribution. According to this model, the distribution of the Hash addresses of the m keys within the limits of a given page may be considered as m experiences. The event of the allocation of a given address to the address space of a given page is then associated with a probability of $1/s$. Hence, according to the properties of the model, one may calculate that the mathematical expectation of the hash addresses that correspond to a given page is equal to m/s with variance $c = m \cdot \frac{1}{s} \cdot (1 - \frac{1}{s})$. Hence, according to the de Moivre-Laplace theorem, all the hash addresses that correspond to the range of a given page are subject to a Gaussian distribution with mean m/s and variance $m \cdot \frac{1}{s} \cdot (1 - \frac{1}{s})$.

The probability P_{os} for a page overflow, i.e. the probability of the number of hash addresses that correspond to a given (constant) page of the hash memory exceeds u , for a Normal distribution and taking into account (1), is defined by the following expression:

$$P_{os} = 0.5 - \Phi\left(\frac{u - m/s}{\sqrt{m/s}}\right) = 0.5 - \Phi\left(\frac{w \cdot (\alpha + \delta) - m/s}{\sqrt{m/s}}\right) = 0.5 - \Phi\left(\delta \cdot \sqrt{\frac{w}{\alpha}}\right) \quad (2)$$

For a permanent key index, i.e. for the case where a page does not need to have excess free memory, for $w \cdot (1 - \alpha - \delta)$ records, where $\delta = 1 - \alpha$, the probability that the number of hash addresses that correspond to a given page does not exceed w , is defined by the following expression:

$$P_{osw} = 0.5 - \Phi\left(\sqrt{w} \cdot \frac{1 - \alpha}{\sqrt{\alpha}}\right) \quad (3)$$

From expression (3) it follows that the probability of overflow of the page at the given order depends on the value of the coefficient δ of the redundant free memory of the page, on the coefficient α of completion as well as on the volume of the page and the memory required for storing these records.

In order to eliminate the possibility of page overflow of all s pages during the population of the constant set Ω of m keys, it is necessary to choose corresponding hash transformations. The probability P_0 of a certain trial of the hash transformation achieves elimination of the possibility of overflow for all the pages of the hash memory, is calculated via the calculation of the probability of each page not overflowing:

$$P_0 = (1 - P_{os})^S = \left[0.5 + \Phi \left(\delta \cdot \sqrt{\frac{w}{\alpha}} \right) \right]^S \quad (4)$$

The mean g of number of trials that are necessary before choosing the hash transformation, so as to eliminate the possibility of overflow of the pages of the hash memory is calculated by the following expression:

$$g = \sum_{j=1}^{\infty} j \cdot P_0 \cdot (1 - P_0)^{j-1} = \frac{1}{P_0} \quad (5)$$

The experiments that were carried out based on the above statistical modeling have shown the sufficiency of the proposed mathematical model of the hash addresses in quasi-permanent key indices in hash memories that are divided in pages. The proposed model may be used for the optimization of the parameters of the hash memory.

From expression (5) it follows that for a given number g_z of trials for the choice of a hash transformation, that guarantees the allocation of records in the pages with completion of less than $(\alpha + \delta) \cdot 100\%$, the values α , δ and w must satisfy the condition:

$$\sqrt{\frac{w}{\alpha}} \cdot \delta = \Phi^{-1} \left(S \sqrt{\frac{1}{q_z}} - 0.5 \right) \quad (6)$$

If it is necessary to ensure the fast selection of a hash transformation, then this may be obtained using the relation $s \cdot P_{os} \approx 1$. In this case for large values of s , the following approximation holds:

$$P_0 = (1 - P_{os})^S = 1 - \sum_{i=1}^S \quad (7)$$

The analysis of the mathematical model demonstrates that, the compromise involved in the hash search, exists in the selection of the number of the pages among which exchanges take place between the main and the cache memory. The analysis leads to the conclusion that the search speed essentially depends on the time for transferring the arranged hash page addresses from the main hash memory to the cache memory, which in turn depends on the size of the pages. Consequently, from the point of view of attaining high speed hash searches, the page size needs to be reduced. At the same time, reducing the time required for selecting the hash transformation requires according to (5) an increase of the page size. A resolution of the above compromise may be found by the defined frequency of the key index reconstruction; the more frequently new keys are assigned, the smaller the required time for selecting the hash transformation and the larger the page size δ may be.

The resolution of these contradicting requirements may be attained either by increasing the size of the pages or by reducing the proportion of the hash memory that is occupied.

5 Organization of Hash Searches

In this section, the principles of the organization of a hash search are outlined [7]. The basic key index exists in the form of Hash Signs and associates the information stored in main memory with the hash addresses. The selection of the hash transformation for nearly constant memories guarantees the assignment of records in the main memory, in accordance with the hash addresses.

The cache memory is set as the active page of the main memory. Additionally, an area is assigned for new records for which there is insufficient space in the main memory pages corresponding to their keys. Apart from that, the storing of the most frequently used codes of the hash transformation may also be organized in the cache memory. The hash transformation $H_K(X)$ is determined by selection for the set Ω of the m keys, either before system boot or at a specially determined time during the system setup. The transformation distributes the user records in the s pages of the hash memory so that each page has enough space for storing more than $w \cdot (1 - \alpha - \delta)$ records.

The transformation $H_K(X)$ is determined by selection in the form of the code K of the key and the hash search to be used is hence defined. During this process the set Ω of the keys is divided into s subsets, each one containing less than $w \cdot (\alpha + \delta)$ elements. This means that each page stores less than $w \cdot (\alpha + \delta)$ records. Beside this, the cipher block forms $\log_2 w$ bit hash addresses $U_K(X)$, that define all the records of the corresponding keys X within the pages. The Hash Sign code $S_K(X)$ together with the Hash address $A_K(X)$ of the page and the hash address $U_K(X)$ within the page, using the one way transformation that implements the cipher block, uniquely defines the key X . Within the page, records are inserted using their own hash address $U_K(X)$. Possible conflicts are resolved using known technologies [11], more frequently by trials.

Each one of the records includes the hash Sign $S_K(X)$ of the keys associated with the data in each case. For the reduction of the size (length) of the records and the subsequent increase of w the reference code of the address may be stored in the place of all codes apart from $S_K(X)$.

For the deletion of a record X , the record is removed from the page by initialization of the memory it occupies and taking into account the displacement of the chain of conflicts to which it formed part.

For the introduction of a new record X , a relocation of the addresses $A_K(X)$ of the pages from main memory to cache memory is generated. If the page is not full, then an attempt is made for the new record X to be inserted at the address $U_K(X)$. If the corresponding page is occupied, then step by step trials are made until a free area is located. If the page that is arranged as $A_K(X)$ appears a completely occupied, then

the record X , that is associated with a subset of the bits of the hash address $A_K(X)$ fits in the special sector of the cache memory that is essentially an overflow area. If the corresponding location in this page is also occupied, then an attempt to resolve the conflict by a step by step trial search is made. In the case where the record is inserted in the overflow area, this does not contain the Hash Sign X , but the entire code. If the predetermined overflow area appears to be completely occupied, then a refresh cycle needs to be initiated.

The process of inserting new records in the hash memory described above may be implemented in two ways, each one giving priority to inserting the record in one of the two available areas. The insertion of a new record between cycles of refreshing the system may occur:

- In the overflow area, which offers a small capacity and is located in the cache memory
- In the main hash memory, where it occupies part of an unoccupied zone in the page that is associated with the hash address key of the new record.

From a theoretical point of view, two cases of record organization may be distinguished:

1. Initially the record is inserted in the free space of the associated page of the main memory and in the absence of that space then it is inserted in the overflow area of the cache memory.
2. The record is initially inserted in the overflow area of the cache memory and when this becomes full, in the free space of the associated page of the main hash memory.

It is apparent that the first case will offer a large number of new record entries, which may be distributed in the hash memory before there is a need to refresh the contents. The advantage of the second case is that the use of the overflow area is more efficient and this implies a significantly higher speed of the hash search.

By using the n -bits key X as a key for code K , the cipher block calculates the h -bit hash address $A_K(X)$ of the page, the hash address $U_K(X)$ internal to the page and the Hash Sign code $S_K(X)$. The hash address $Q_K(X)$ for the record X within the overflow area is also simultaneously formulated. The address $Q_K(X)$ is a subset of the bits $A_K(X)$, $U_K(X)$ and $S_K(X)$. The Hash search of the overflow area is performed for the address $Q_K(X)$. If the search is successful, then the record with code X is retrieved from the overflow area and access to the information associated to X is accessed.

Together with the search in the overflow area of the cache memory, the page that is associated with the code $A_K(X)$ is recovered from the main memory to cache memory. Following that, using the address $U_K(X)$ the corresponding record is read. The hash sign of the record is calculated and compared with the hash sign $S_K(X)$. In the event where these two coincide, the passwords are compared and the process of granting access rights is completed. In the event where the codes of the hash signs do not coincide, step by step trials are performed with the remaining record until

either a matching hash sign or an empty record is found. The empty record case implies that a record with code X is not stored in the memory.

The time t_1 of the search with the distinguishing key X is defined as the sum of:

- t_H the time for calculating the hash transformation using the cipher block
- t_S the time for testing the page with address $A_K(X)$ from main memory to cache memory
- t_X the time for a hash search in the selected page of the cache memory

$$t_1 = t_H + t_S + t_X$$

Considering that $t_S \gg t_H \gg t_X$ the search time is principally defined by the time consumed in testing the page from main memory to cache memory.

6 Preliminary Results

Initial conclusions from this research, that focuses in increasing the efficiency of hash address searching in nearly constant key indices, may be summarized as follows:

1. A hash search model was developed that corresponds to the nearly constant key index case. The model takes into account the multilevel memory organization of modern computational systems.
2. The basis of the model that was developed, proposed the organization of hash searches in nearly constant key indices. It was shown that the search time is defined by access to no more than low level memory pages.

The proposed hash memory organization may be efficiently used for increasing the throughput of databases, of linguistic processors as well as for accelerating authentication of users in computer networks. This increase in throughput is expected to be associated with higher overall computational efficiency, lower energy consumption and therefore greener performance of the overall processing system. The quantification of the environmental benefits is the subject of current investigations. More particularly, Table 3 summarizes the requirements for the design of hash functions as determined for the class of applications relating to big data processing for the purpose of data mining.

- In the case of shingle hashing, the proposed method provides satisfactory results since it can be guaranteed by consideration of the encryption algorithm that each value will produce a unique value, which is the fundamental requirement for this operation.
- In the case of min hashing, it may be similarly guaranteed that the proposed method will permute the rows of the signature table and this permutation will have randomness characteristics.

Table 3 Summary of hash function types required for efficient big data mining

Subsection	Problem	Example application	Hash function type	Collisions allowed	Special Requirements	Satisfied by the proposed method
Section 3.1	Shingle Hashing	Document similarity	Few-to-many	No		Yes
Section 3.2	Min Hashing	Document similarity	N-to-N	No		Yes
Section 3.3	Locality Sensitive Hashing	Document similarity	Many-to-few	Yes	Probability of collision proportional to similarity	Under investigation
Section 3.4	Stream data sampling	Obtain a representative sample of the data	Many-to-few	Yes	Uniform probability of each hash value to appear	Yes
Section 3.5	Filtering streams	Determine membership of a desired set	Few to many	No	Small probability of false acceptance	Yes
Section 3.6	Count distinct problem	Count the number of distinct values appearing in a particular stream	One-to-one	No	Compact bit size of hash value	Under investigation
Section 3.7	First Order statistics	Calculate the mean and median of a stream of values	Many-to-few	Yes		Yes
Section 3.8	Frequent object sets	Determine sets of objects that appear frequently together	Many-to-few	Yes	Maintain similarity	Under investigation

- In the case of locality sensitive hashing, the hash function is required to maintain similarity criteria, i.e. small variations in the input need to leave the output unchanged. This requirement is fundamentally contrary to the operation of the encryption algorithm. The design of a suitable substitute hash function is being investigated.
- In the case of stream data sampling, it is required that hashing different tuple key identities is mapped with uniform probability to a small number of buckets. This is consistent with the operation of the encryption algorithm and is hence satisfied by the proposed method.
- In the case of filtering streams of tuples, it is required that incoming values be mapped to distinct values in a much larger address space. This can be provided by the proposed algorithm via the selection of a suitably large address space and using smaller key values so that only a certain subspace is occupied.
- In the case of the count distinct problem, it is required that the word length occupied by the hash value is compact, i.e. that it in general occupies the minimum number of bits possible. This requirement may be satisfied if multiple hash functions are used, but it is under investigation if the encryption algorithm may be modified so as to satisfy this criterion every time.
- The first order statistics calculation problem is a special case of stream data sampling and is hence also feasibly solvable via the proposed method.
- The frequent object set determination problem is another occasion where similarity needs to be maintained and alternative designs of hash functions need to be pursued.

7 Conclusions

In this chapter, the use of hash functions for the purpose of achieving computational and energy efficiency when extracting information from extremely large sets of data was analyzed. This goal is consistent with current legal, ethical and global requirement for green IT engineering. It was noted that, given the ever increasing use of the IoT model for designing data processing systems capable of optimizing decisions, producing personalized reports that interest particular people or groups, detecting and evading crises and other applications and the ever increasing data sizes, such techniques are a prerequisite for designing implementable solutions. Furthermore, given the scale of such operations, small optimizations in the computational resources required for a particular task are reflected to huge savings in the environmental impact of such types of activities. The case of big data mining, especially as manifested in Internet of Things systems, was therefore seen to be an occasion where technical effectiveness, economic benefit and environmental responsibility are not conflicting goals. In other words, the quest for green IT and system viability and scalability are in this case equivalent and not conflicting. Specific problems in the context of IoT big data management where described,

within which efficiency, in both the economic and environmental sense, is pursued via purpose designed hash functions. It was noted that not all hash functions are suitable for all applications and specialized design techniques are required. Several existing techniques were presented and compared to a technique that was recently proposed in order to achieve quasi-permanent keys. The applicability of this technique on the specific exercises that were listed for the IoT big data management paradigm was investigated. Areas of ongoing research for innovative hash function design algorithms were hence outlined. This research is expected to relate in a quantitative manner computational benefits of the use of such functions along with their green aspect, as well as further energy efficient hash function design algorithms.

References

1. Berman, F., Bock, M.E., Dittert, E., O'Donnel, M.J., Plank, D.: collections of function of perfect hashing. *SIAM J. Comput.* **15**(2), 604–618 (1986)
2. Czech, Z.J., Havas, G., Majeovski, B.S.: An optimal algorithm for generating minimal perfect hash functions. *Inform. Process. Lett.* **43**(5), 257–264 (1997)
3. Jagannathan, R.: Optimal partial-match hashing design. *ORSA J. Comput.* **3**(2), 86–91 (1991)
4. Ramakrishna, M.V., Bannai, Y.: Direct perfect hashing function of external files. *J. Database Admin.* **2**(1), 19–28 (1991)
5. Polymenopoulos, A., Bardis, E.G., Bardis, N.G., Markovskaja, N.A.: Perfect hashing using linear Boolean functions. In: *Problem in Applied Thematics and Computational Intelligence*, pp. 5–11. WSEAS Press, ISBN 960-8052-30-0 (2001)
6. Bardis, E.G., Bardis, N.G., Markovskyy, A.P., Spyropoulos, A.K.: High storage utilization of hash memory by reducing of information redundancy for hashing. In: *Submitting as a Special Issue of IMACS/IEEE CISC'99 International MultiConference, "Software and Hardware Engineering for the 21th Century"*, pp. 272–276 (1999). ISBN 960-8052-06-8
7. Bardis, Nikolaos G., Doukas, Nikolaos, Markovskiy, Oleksandr P.: Hash addressing of the quasi-permanent key arrays in multilevel memory. *J. Appl. Math. Bioinf.* **3**(4), 91–105 (2013)
8. Leskovec, J., Anand R., Ullman, J.D.: *Mining of Massive Datasets*. Cambridge University Press, Cambridge (2014)
9. Kharchenko, V., Illiashenko, O. Concepts of green IT engineering: taxonomy, principles and implementation. In: Kharchenko, V., Kondratenko, Y., Kacprzyk, J. (eds.) *Green IT Engineering: Concepts, Models, Complex Systems Architectures, Studies in Systems, Decision and Control*, Vol. 74, pp. 3–20. Springer, Berlin (2017). doi:[10.1007/978-3-319-44162-7_1](https://doi.org/10.1007/978-3-319-44162-7_1)
10. Kondratenko, Y.P., Korobko, O.V., Kozlov, O.V.: PLC-based systems for data acquisition and supervisory control of environment-friendly energy-saving technologies. In: Kharchenko, V., Kondratenko, Y., Kacprzyk, J. (eds.) *Green IT Engineering: Concepts, Models, Complex Systems Architectures, Studies in Systems, Decision and Control*, Vol. 74, pp. 247–267. Springer, Berlin (2017). doi:[10.1007/978-3-319-44162-7_13](https://doi.org/10.1007/978-3-319-44162-7_13)
11. Sun, N., Nakamura, R., Zhu, N., Tada, A., Sun, W.: An analysis of average search cost of external hashing with separate chain. In: *Processing of 7-th WSEAS International Conference on Circuits, Systems, Communications and Computers (CSCC-2003)*, pp. 315–324 (2003)

Green IT Engineering: Components, Networks and
Systems Implementation

Kharchenko, V.; Kondratenko, Y.; Kacprzyk, J. (Eds.)

2017, XIV, 355 p. 147 illus., 82 illus. in color., Hardcover

ISBN: 978-3-319-55594-2