A space-efficient, probabilistic data structure and storage and retrieval method for key-value information

Nicholas Boyd Greenfield

February 5, 2014

Introducing the B-field Data Structure

The B-field is a novel space-efficient, probabilistic data structure for storing key-value pairs. B-fields support insertion (`INSERT`) and lookup (`LOOKUP`) operations, and share a number of mathematical and performance properties with the well-known probabilistic data structure the Bloom filters. In contrast to Bloom filters, which only support set membership queries, B-fields support associating a set of x, y key-value pairs, where S is the set of stored x values, $|S| = n$, D is the set of all possible y value ($f : S \to D$), and θ represents the "value range" of the B-field ($max(D) = \theta$). Put differently, if $f(x)$ is the function that maps all x values to their corresponding y value, such that $f(x) = y$, D is the domain of the function $f(x)$.

B-fields have a number of advantages over existing probabilistic and deterministic data structures for associating keys and values. Specifically, B-fields exhibit a number of key properties that no other single known data structure provides:

- **Space Efficiency:** B-fields probabilistically store key-value pairs in a highly space-efficient manner optimized for in-memory use. For many common use cases or configurations, it is possible to store billions of elements using only tens of gigabytes of storage (or a few bytes per key-value pair). These space requirements scale linearly with the number of elements in the B-field n, or put differently a B-field has $O(n)$ space complexity.

- **Speed:** B-fields are optimized for in-memory storage of key-value pairs, and require both little computational overhead and few memory accesses. A major benefit of the B-field is that all `INSERT` and `LOOKUP` operations have $O(1)$ time complexity.

- **Well-Defined and Bounded Errors:** Three major types of error are possible with probabilistic key-value stores:

  - **False Positives:** A false positive is defined as rate at which a data structure returns a value for y when $x \notin S$ (in this situation, the data structure should return a special value $\perp$, as x is not in the input range of $f(x)$). B-fields do exhibit false positives at an error rate of α. At the cost of lesser space efficiency, it is possible to set α to be arbitrarily small. False positives are a common feature of probabilistic data structures, but a disadvantage of the B-field vis-à-vis deterministic data structures.
  - **Indeterminacy:** An indeterminacy error, denoted β, is the rate at which a data structure returns an indeterminate answer for $x \in S$, either in the form of a subset of D which x might map to or an error. While intermediate components of the B-field do have β errors, for a complete B-field $\beta = 0$ or $\beta \approx 0$ (it is possible to guarantee that $\beta = 0$ but slightly relaxing this requirement simplifies the B-field implementation).
  - **Erroenous Retrieval:** An erroneous retrieval error γ occurs when a data structure is queried for an $x \in S$, and returns the wrong value of y such that $f(x) \neq y$. Erroneous retrievals are not possible with B-fields.

- **Dynamism & Mutability:** While the B-field, like many other probabilistic data structures, only supports `INSERT` and `LOOKUP` operations, it does support dynamic sets where some or all $x \in S$ are unknown at the time the data structure is created. This enables the B-field to be used in a variety of real-time applications impossible with other data structures supporting only static sets (where all $x \in S$ must be known at the time the data structure is created).

- **Generalizability:** The B-field is designed to be a highly generalizable data structure, operating efficiently for both small and large set sizes (n) and small and large value ranges (θ). Current B-field implementations have been used to store billions of keys across both small (e.g., $\theta \approx 10$) and large value ranges (e.g., $\theta \approx 2^{20}$ – the B-field should exhibit good performance properties for a θ as large as 2^{32}). One enabling component of this scalability is the B-fields portability – the implementation is hardware agnostic and can be tuned for specific close-to-the-silicon hardware configurations or eschew platform-specific optimizations for portability across a wide range of computing platforms.

Implementation Details

The B-field works by encoding y into a binary string and then inserting that binary string into a large bit array using k hash functions ($h_1 ... h_k$) for each inserted element x. While some of the implementation details that follow will be familiar to those acquianted with hash-bashed data structures in general and Bloom filters specifically, the B-field is designed for storing key-value pairs rather than simple set membership information. The basic construction of an immutable B-field follows each of the proceeding steps (in the subsequent section the requirements for constructing a mutable B-field are enumerated):

1. **Configuration.** As a first step, the user needs to select several configuration options, which will determine how the initial B-field is constructed. These key configuration variables are:

   (a) The desired maximum false positive rate α.

   (b) The expected number of values to be stored n where $n = |S|$. If more than n elements are inserted into the B-field, the expected false positive rate will exceed the desired maximum.

   (c) The maximum value θ of the set of all possible y values. Note that using the default encoding system outlined here, D must consist of all or a subset of the non-negative integers $\{1, ..., \theta\}$ or $\{0, ..., \theta - 1\}$. This can, however, be easily mapped to any set of up to θ distinct values by simply using y as an index or a pointer to an end-value in a set $D\prime$ where $|D\prime| = \theta$ (in which case y serves as an intermediate value). This use case my make sense where the end-values of y are in a high k-bit space (e.g., images, long strings, etc.).

   (d) A value ν and a value κ such that $\binom{\nu}{\kappa} \geq \theta$, where $\binom{\nu}{\kappa}$ is simply the combination formula, but using ν in place of the more common n (which is already used to denote the cardinality of S or $|S|$) and κ in place of the more common k (which is used to enumerate the required hash functions $h_1...h_k$):

   $$\binom{\nu}{\kappa} = \frac{\nu!}{\kappa!(\nu - \kappa)!}$$

   As the B-field encodes the values y in ν-length bitstrings with exactly κ ones (or a weight of κ in information theoretic terms), it is best to both: a) minimize κ; and b) keep ν within an order of magnitude of κ; while selecting ν and κ. For example, in attempting to select ν and κ such that $\binom{\nu}{\kappa} \geq 1000$ it is preferable to set $\nu = 20$ and $\kappa = 3$ ($\binom{\nu}{\kappa} = 1140$) instead of setting $\nu = 1000$ and $\kappa = 1$. The reason for this should become clear in the proceeding discussion of error rates and the B-field's operation.

2. **Bit Array Sizing and Hash Function Selection.** Next, it is necessary to size the primary bit array used in the B-field and select an appropriate number of hash functions k. As the B-field hashes input values into a bit array in a manner similar to a Bloom filter, the same formulas may be used for selecting the optimal size of the bit array and number of hash functions, with the exception that a B-field requires an $m\kappa$-bit array vs.

the standard m-bit array required by a simple Bloom filter (p represents the probability of a single-bit error, which is defined further below in 5b):

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

$$k = \frac{m}{n} \ln 2$$

With regards to the selection of the hash function itself, any hash function with good pseurandomness and distribution qualities will suffice for the B-field. While the prior equations assumes a perfectly random hash function in theory, the B-field's performs comparably using heuristic off-the-shelf hash functions in practice (such as the fast MurmurHash3 function). Again, while the set of hash functions should in theory be fully independent, in practice it is possible (and preferable for performance reasons) to simply use 2 independent hash functions as seeds and then relax the independence requirements for the remaining hash functions (i.e., two seed hash functions $h_a(x)$ and $h_b(x)$ can be used to create n composite hash functions, with each defined as $h_n(x) = h_a(x) \times ((n-1) \times h_b(x))$).

3. **Bit Array Allocation**. Now having calculated m and selected κ, it is necessary to allocate an $m\kappa$-bit array of memory, setting all bits initially to 0. Note that the B-field may make use of multiple bit arrays, and this initial $m\kappa$-bit array is termed the B-field $Array_0$.

4. **Key-Value Insertions.** Next, the keys of the B-field ($x_1 ... x_n$) and their corresponding values ($y_1 ... y_n$) are inserted. This requires two steps or functions for all x in $x_1 ... x_n$.

   (a) `ENCODE`. The `ENCODE` operation translates a given value y into a binary ν-length string with κ 1s suitable for insertion into the B-field $Array_0$. Using the standard encoding system, this simply involves translating y into the y^{th} ν-length κ-weight bitstring of all $\binom{\nu}{\kappa}$ combinations in lexicographic or reverse lexicographic order (e.g., 1 translates to "0001" if $\nu = 4$ and $\kappa = 1$ using a lexicographically ordered encoding scheme). Other even- and uneven-weighted codes could also be used in this step (e.g., error-correcting codes), but their use is not detailed here. Figure 1 illustrates how the `ENCODE` (and `DECODE`) operation works where $\nu = 5$, $\kappa = 2$.

   (b) `INSERT`. The `INSERT` operation places the encoded value y (associated with the key x) into the B-field. First, the k hash functions are applied to x ($h_i(x) ... h_k(x)$) and the $\texttt{mod} m\kappa$ of each value taken (where $\texttt{mod}\, x$ is remainder of a value divided by x). These k values, where $0 \leq \texttt{mod} m\kappa(h(x)) \leq m\kappa$, serve as indices for the bitwise OR insertion of the encoded value y, where the first bit of the ν-length bit string is

$\texttt{mod}m\kappa(h_1(x))$ and the last bit is $\texttt{mod}m\kappa(h_1(x)) + \nu - 1$. In the case of inserting the first x, y key-value pair into an empty $Array_0$ where $k = 5$, $\nu = 6$, $\kappa = 1$, and $y = 1$ (unrealistically small values), the result would be bitwise ORing the bitstring 000001 into the $m\kappa$-bit array 5 times (and changing a total of 5 bits from 0 to 1). Figure 2 illustrates a sample `INSERT` operation for a key-value pair where the value $y_i = 2$ and the B-field is configured with $\nu = 5$, $\kappa = 2$, and $k = 6$. The shown operation applies equally well to insertion into $Array_0$ of a B-field or any secondary array.

5. **Construction of Subsequent B-field Arrays.** In order to describe the construction of subsequent arrays $Array_1 ... Array_a$ where a is the total number of arrays in the B-field, it is useful to briefly highlight: a) how the `LOOKUP` operation is performed; and b) the different types of error in the B-field as constructed thus far.

   (a) The `LOOKUP` operation. The `LOOKUP` operation mirrors the `INSERT` operation. A given x_i that the user wants to lookup in the B-field is hashed k times using the has functions $h_1 ... h_k$ (all $\texttt{mod}\,\texttt{m}\kappa$). At each index position for $h_1 ... h_x$ an ν-length bit string is taken and and bitwise ANDed with a initial ν-length bit string consisting of κ 1s (i.e., having a weigh of ν). Using the values from the prior example, if the following 5 6-bit strings $\{001101, 111111, 100001, 100011, 110011\}$ are found in the B-field $Array_0$ at $h_1(x_i) ... h_k(x_i)$, the bitwise AND of these returns 000001 which can be `DECODE`-d to yield the value 1 for y_i. Figure 3 illustrates a sample `LOOKUP` operation, where the value $y_i = 2$ associated with a given $x_i \in S$ is retrieved from a B-field where $\nu = 5$, $\kappa = 2$, and $k = 6$.

   Looking across all possible cases, if the resulting bit string has fewer than κ 1s, it is known that $x \notin S$ and the B-field returns the special value $\perp$ (as at least κ 1s result from the `INSERT` operation described in 4b). If the resulting bit string has exactly κ 1s, then the bit string is decoded using the `DECODE` function (simply the inverse of `ENCODE`) and the value y_i mapping to the key x_i is returned. This operation will erroneously return a y_i for an $x \notin S$ at the false positive rate of α (derived below in 5b). And, finally, with a probability of β (also derived below in 5b) an indeterminate result with more than κ 1s will be returned. The reduction of β is one of the primary aims of the construction of subsequent B-field arrays.

   (b) B-field error rates detailed. As noted above, the `LOOKUP` operation in B-field $Array_0$ suffers from both false positive (α) and indeterminacy (β) errors. These error rates can be simply derived based on the above pattern of the `INSERT` operation, which sets up to κ random bits within the bit array to 1 for each of k hash functions.[1]First, it

[1]Note: In practice, the pattern of 0s and 1s will not be random for a B-field at low fill

is necessary to determine the probability that any single bit in the ν-length bit string will be incorrect after the k bit strings are bitwise ANDed together (described here as p, the probability of a single bit error). First, the probability that an individual bit is not set to 1 by a single hash function (each setting κ bits) during the insertion of an x_i is:

$$1 - \frac{\kappa}{m\kappa}$$

Canceling the κ bits set by the hash function out yields:

$$1 - \frac{1}{m}$$

Then, the probability that a single bit is not set for any of the k hash functions is:

$$(1 - \frac{1}{m})^k$$

After n insertions of $x_1 ... x_n$, the probability that a given bit remains 0 is:

$$(1 - \frac{1}{m})^{kn}$$

And the probability that it is a 1 is thus:

$$1 - (1 - \frac{1}{m})^{kn}$$

After each of the k components of the lookup (one per hash function) the probability that an individual bit is wrongly a 1 is:

$$(1 - (1 - \frac{1}{m})^{kn})^k \approx (1 - e^{kn/m})^k$$

Substituting the formulas for optimal m and k values from above yields:

$$p = (1 - e^{-(m/n \ln 2)n/m})^{(m/n \ln 2)}$$

Where p is the probability that a given bit in the ν-bit string encoding a given value y_i is a 1 when it should be a 0. Using p, we can then derive the probability of a false positive for the overall $Array_0$ of the B-field (α). Since a false positive is defined as

rates, as κ adjacent or near-adjacent bits are set at a time. However, as a greater proportoin of the bit array becomes set to 1, the pattern of this spatial clustering should, in practice, sufficiently resemble a fully random $m\kappa$-bit array for the purposes of the above analysis.

> there being $\geq \kappa$ bits incorrectly flipped to 1 in the ν-bit string, α is simple $CumBinom(\nu, \kappa, p)$ where $CumBinom$ is the right tail of the cumulative binomial distribution:
>
> $$\alpha = Pr(X \geq \kappa) = \sum_{i=\kappa}^{\nu} \binom{\nu}{i} p^i (1-p)^{\nu-i}$$
>
> And correspondingly, β is the chance that a given value $x \in S$ returns a ν-length bit string with $> \kappa$ bits flipped to 1 (at least κ bits are guaranteed to be set to 1 by the `INSERT` operation). This corresponds to $CumBinom(\nu - \kappa, 1, p)$. Without correction, β is too high for many applications (though usable for others), and consequently the next step in the B-field's construction is an effort to correct β such that $\beta \approx 0$, which also has the effect of reducing α from the cumulative binomial distribution for all values $\geq \kappa$ to the binomial distribution for κ exactly, which is $\binom{\nu}{\kappa} p^\kappa (1-p)^{\nu-\kappa}$.

Given these error rates α and β, a correction is desirable. In order to eliminate β errors (and as a side effect reduce the magnitude of α false positive errors), all of the input keys $x_1...x_n$ are iterated through a second time. This time, instead of inserting them into $Array_0$, however, a `LOOKUP` is performed and the number of 1 bits in the ν-bit string, or b_i, is noted for each x_i. If $b_i > \kappa$, we have a β-class error as a known $x \in S$ is not returning its corresponding ν-length κ-weight bit string. The subset of $x_1...x_n$ for which $b_i > \kappa$ (which has an expected number of elements βn). This βn subset is then inserted into a second array $Array_1$, which is created following the above procedures except for it is scaled to be a $\beta m \kappa$-bit array as it has only βn elements instead of n.

This secondary array $Array_1$ allows for an indeterminate `LOOKUP` query of x_i against $Array_0$ yielding $> \kappa$ 1 bits to then be checked against $Array_1$ as well. Since the probability of x_i yielding $> \kappa$ 1 bits is β for $Array_0$ and also β for $Array_1$ (which shares identical properties but contains a smaller number of elements), the indeterminacy rate is reduced from β to β^2 with the addition of $Array_1$. Further arrays $Array_2...Array_{a-1}$ can be added until a set of a arrays is formed, choosing a such that $\beta^a n < 1$ (in practical terms, it will often suffice to simply pick a value of a such as 8 or 10 such that very few elements are in the final $Array_{a-1}$ even if n is large). These secondary arrays also have the effect of lowering the false positive rate α. Becase a false positive where $x_i \notin S$ yielding $> \kappa$ 1 bits also leads to a check against $Array_1$ (and $Array_2 \ldots Array_{a-1}$ if the `LOOKUP` operation yields $> \kappa$ 1 bits in $Array_1$ and subsequent secondary arrays), the false positive rate α is also reduced from the cumulative binomial distribution towards the simple binominal $\binom{\nu}{\kappa} p^\kappa (1-p)^{\nu-\kappa}$. Put differently, the secondary arrays reduce the set of circumstances where α errors can occur from anytime $\geq \kappa$ 1 bits are flipped for an $x_i \notin S$ to two cases: 1) when exactly κ 1 bits in $Array_0$ are flipped for an $x_i \notin S$ (the simple binomial); or 2) the exceedingly rare case where $> \kappa$ 1 bits are flipped in $Array_0 \ldots Array_{a-2}$ and exactly κ bits are flipped in the final $Array_{a-1}$. The probability of this second case should ≈ 0 as the chance of $> \kappa$ bits flipping decreases exponentially in subsequent secondary arrays.

To summarize, this cascade of secondary arrays beyond the first array $Array_0$ ($Array_1 ... Array_{a-1}$) reduces the false positive rate α from a cumulative binomial distribution to the binomial for κ individual bit-level errors and eliminates all indeterminacy errors (i.e., $\beta = 0$).[2]And, it achieves this error reduction at a very modest cost to the space efficiency of the B-field. Recall that the original space requirements for $Array_0$ of the B-field is $m\kappa$ bits where $m = -\frac{n \ln \alpha}{(\ln 2)^2}$ and κ is selected is $\binom{\nu}{\kappa} > \theta$ where θ is the maximum value of the domain of stored values $y_1 ... y_n$ or $|D|$ in the case where $y_1 ... y_n$ is $1 ... n$. Formulated differently, $Array_0$ requires $\beta^0 m\kappa$ bits, while $Array_1$ requires $\beta^1 m\kappa$ bits and $Array_{a-1}$ requires $\beta^{a-1} m\kappa$ bits. Dividing everything by $m\kappa$ bits reveals the underlying geometric series:

$$1 + \beta^1 + ... + \beta^{a-1}$$

The sum of which is simply:

$$\frac{1}{1-\beta}$$

Thus, the total area required by a B-field with a primary $Array_0$ and $a-1$ secondary arrays is only $\frac{1}{1-\beta} m\kappa$ bits. In many use cases, an initial indeterminacy rate of $0.05 \leq \beta \leq 0.20$ would not be uncommon. Correcting this for a 5.3% ($\frac{1}{0.95}$) or 25.0% ($\frac{1}{0.80}$) space penalty is generally a good tradeoff, but a B-field can be constructed with only an $Array_0$ at the cost of needing to manage β indeterminacy errors in the key-value lookups.

Mutability

A useful feature of the B-field is its employment of secondary arrays ($Array_1 ... Array_{a-1}$) in order to eleminate indeterminacy errors (β) and reduce false positives (α). This functionality requires additional iterations through the input data, and insertion of elements experiencing β errors into secondary arrays – a straightforward procedure in the case of an immutable data structure (described above) where all $x \in S$ are known in advance. However, because all of the constituent arrays of the B-field are themselves mutable (i.e., it is possible to perform an `INSERT` operation at any time, not just during array construction), it is also

[2] Note, however, that it may be useful in an implementation to allow β to take on a small non-negative value proportional to α so as to constrain the number of required arrays a.

possible to create, update, and maintain the secondary arrays for a mutable B-field variant which supports the dynamic insertion of x, y key-value pairs after creation-time. Two simple steps are required:

1. **Key-Value Insertion Logging.** As every key-value pair x_i, y_i is inserted, they are also logged to disk.[3] Simply appending the keys and values to a log file suffices, as no special access-time or other requirements are needed. In real-world uses cases, logging the inserted key-value pairs is also a best practice as it enables restoring the B-field should the computer system suffer an error or forced reboot.

2. **Regular Key-Value Lookups and Re-insertion.** Iterating through the on-disk log of inserted key-value pairs permits checking that `LOOKUP` operations performed for the set of inserted keys $x_1 ... x_n$ have precisely κ 1s in the returned bit strings. As in 5, x_i keys for which the returned bit string has $> \kappa$ 1 bits are then inserted (with their corresponding value y_i) into $Array_1$, and so on. A separate log of insertions for $Array_1 ... Array_{a-1}$ must also be maintained, although again with geometrically decreasing storage requirements.

With the addition of these two simple layers – on-disk logging of inserted key-value pairs (a best practice to ensure the data's durability should an error arise) and regular key-value lookups and re-insertion – a mutable variant of the B-field can also be constructed that achieves $\beta \approx 0$ and $\alpha = \binom{\nu}{\kappa} p^{\kappa} (1-p)^{\nu-\kappa}$ where p equals the per-bit error rate of the underlying array (defined above in 5b). One simple implementation of this functionality would be to maintain a log file of all previously inserted key-value pairs and to scan through the log in a continuous background job, inserting known keys where a `LOOKUP` yields $> \kappa$ bits into one or more secondary arrays ($Array_1 ... Array_{a-1}$).

Overview of B-field Key Metrics

In this section, several key metrics for the B-field are summarized. As stated earlier, no other known data structures share these unique attributes.

- **B-field Key Metrics**
  - **Creation & Insertions.** $O(n)$ total creation time and $O(1)$ insertions, requiring an average of $\frac{k}{1-\beta}$ hash computations[4] and memory accesses, and ak hash computations and memory accesses in the worst case (for many applications a will be between 4 and 7).
  - **Lookups.** $O(1)$ lookups, requiring the same hash computations and memory accesses as for an insertion.

[3] Note: In situations where the keys are very large (e.g., long strings), it is possible to instead log the outputs of all seed hash functions to reduce the required on-disk storage space.

[4] Note: Because it is possible to use $< k$ seed hash functions, a significant portion of these functions may be computationally simpler and faster than a pseudorandom hash function.

- **Space Complexity.** $O(n)$ space complexity, requiring $\frac{m\kappa}{1-\beta}$ bits where m, κ, and β are defined above ($Array_0$ takes $m\kappa$ bits, while the entire B-field requires an additional factor of $1 - \frac{1}{1-\beta}$).
- **1bn URL Use Case.** To store the association between 1 billion arbitrary-length URL keys and a numeric value in the domain $\{1...1000\}$ requires ≈ 7.1 gigabytes (GB) at an error rate of $\alpha = 2^{-32}$ ($p \approx 2^{-14.1}$) and $\beta = 0$.

Sample Applications and Optional Features/Extensions

The B-field lends itself to a wide variety of computing applications. An initial set of sample applications currently being explored by the authors include:

- Scientific computing. Store larger quantities of data in less space, and with faster access times. The downside associated with error rates should be minimal for any application where there is already underlying (scientific) measurement error, and consequently where case-specific error corrections and management techniques are in play.
- Databases (distributed or otherwise). Minimize disk accesses, network costs, etc. by using a B-field to store the location of database records either locally (i.e., block info) or remotely (i.e., which shard of a database to query). This could also be applied to multi-database or multi-table setups. This application could also enable faster distributed joins and related database operations.
- Local data and applications. Store more data, closer to the client, at lower space or network costs. Example: maintain a richer list of malicious websites within every browser, supporting a more nuanced security environment that details classes of risks and associated precautions (e.g., an ordinal risk scale) vs. a simple binary on/off warning system.

The core feature of the B-field is supporting probabilistic key-value storage in a highly space- and time-efficient manner. Depending on the ultimate use case, this data structure can be extended as needed. Several sample extensions are listed below:

- Cache locality optimizations. (Use the first hash to select a cache-sized block from $m\kappa$-bit array, then hash to locations within it)
- Scalability. (Split $m\kappa$-bit array across multiple machines, either in blocks or in a strided fashion)
- Space efficiency optimizations. (If a large percentage of S maps to y values with a smaller maximum value θ, store that percent of S in a B-field built with lower ν and κ values, while storing the remaining elements in a second B-field. This extends to arbitrary depth, and it's also possible to use combinations of B-fields to further encode y values)

- Alternative encoding schemes. (Use a different encoding scheme, including those with built-in error correction. Then error-correction can be done at both the decoding step and via the use of a set of B-field arrays, $Array_0 - Array_{a-1}$).

```
ENCODE(1)  = “00011”      DECODE(“00011”) = 1
ENCODE(2)  = “00101”      DECODE(“00101”) = 2
ENCODE(3)  = “00110”      DECODE(“00110”) = 3
ENCODE(4)  = “01001”      DECODE(“01001”) = 4
ENCODE(5)  = “01010”      DECODE(“01010”) = 5
ENCODE(6)  = “01100”      DECODE(“01100”) = 6
ENCODE(7)  = “10001”      DECODE(“10001”) = 7
ENCODE(8)  = “10010”      DECODE(“10010”) = 8
ENCODE(9)  = “10100”      DECODE(“10100”) = 9
ENCODE(10) = “11000”      DECODE(“11000”) = 10
```

Figure 1: Sample `ENCODE` and `DECODE` operations for $\nu = 5$ and $\kappa = 2$

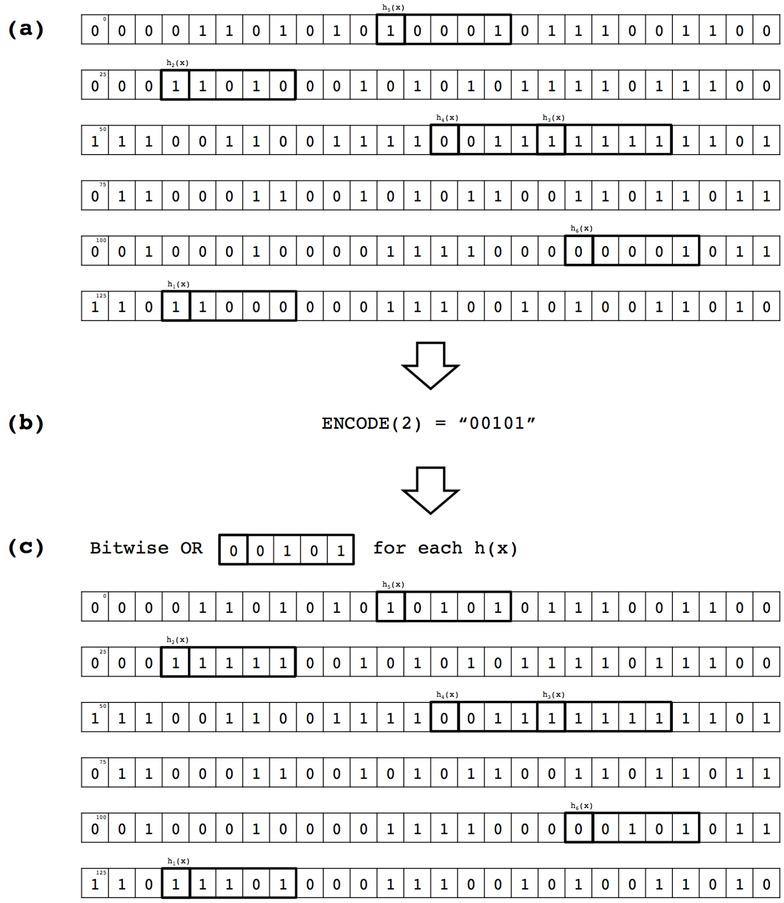

Figure 2: Sample `INSERT` operation

(a): Hash an input key x k times, providing k indexes for positions in a B-field array (illustrated with a small hypothetical sample array here)

(b): Encode the value y associated with the key x into a ν-length, κ-weight bit string (here $y = 2$, $\nu = 5$, and $\kappa = 2$)

(c): Perform a bitwise OR between a ν-length slice of the array and the encoded ν-length bit string representing the value y, for all k hash functions $h_1(x)...h_k(x)$

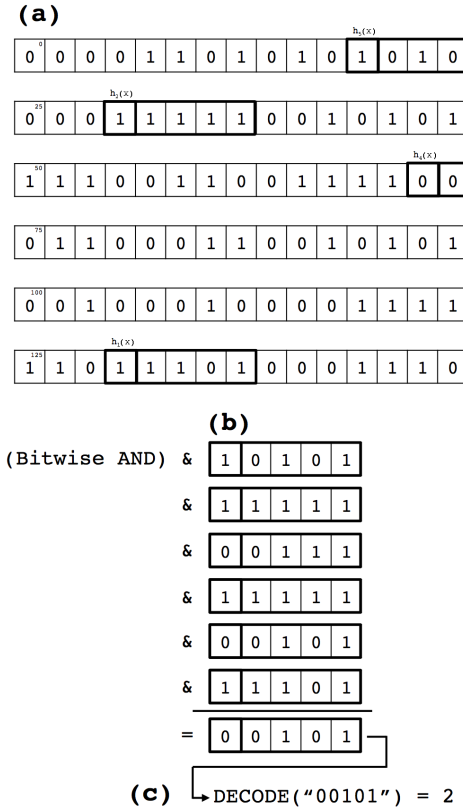

Figure 3: Sample `LOOKUP` operation

(a): A key x is hashed k times, and ν-length, κ-weight bit strings are extracted from the B-field array

(b): The k bit strings are bitwise ANDed together

(c): The resulting bit string is `DECODE`-ed to recover the value y associated with the key x