ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

“Cooperative caching in proxy-based wireless networks for multi-item queries”

Διπλωματική Εργασία

ΚΟΚΚΑΛΗΣ ΓΙΑΝΝΗΣ

Βόλος, 2015

«Συνεργατικό caching σε ασύρματα δίκτυα με proxies για multi-item ερωτήματα»

Επιβλέπων: Κατσαρός Δημήτριος, Επίκουρος

Συνεπιβλέπων: Αθανάσιος Κοράκης, Λέκτορας

“Cooperative caching in proxy-based wireless networks for multi-item queries”

Supervisor: Katsaros Dimitrios, Assistant Professor

Co-advisor: Korakis Athanasios, Lecturer

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον καθηγητή του τμήματος Μηχανικών Η/Υ Τηλεπικοινωνιών και Δικτύων και βασικό επιβλέποντα της πτυχιακής αυτής εργασίας κ.Δημήτριο Κατσαρό που μου έδωσε την ευκαιρία να πραγματοποιήσω αυτή την μελέτη. Η υποστήριξή του, η αμέριστη συμπαράστασή του, αλλά και οι διαρκείς και εύστοχες υποδείξεις του βοήθησαν στην έγκαιρη ολοκλήρωση αυτής της μελέτης.

Επιπρόσθετα, θα ήθελα να ευχαριστήσω τον έτερο επιβλέποντα καθηγητή κ. Αθανάσιο Κοράκη και τέλος, θα ήθελα να ευχαριστήσω την οικογένειά μου και τους φίλους μου που μου συμπαραστάθηκαν σε όλη την διάρκεια της εκπόνησης αυτής της εργασίας.

Περίληψη

Η επικοινωνία των μνημών cache ανάμεσα στα Web proxies είναι μια σημαντική τεχνική για τη μείωση της κυκλοφορίας στο διαδίκτυο. Αυτή η εργασία δείχνει τα πλεονεκτήματα της διαμερισμένης cache και προτείνει ένα καινούριο πρωτόκολλο που ονομάζεται «sharing caches». Σε αυτό το πρωτόκολλο, κάθε proxy διατηρεί μια εικόνα του καταλόγου cache από κάθε συμμετέχον proxy και ελέγχει αυτές τις εικόνες για πιθανή επιτυχημένη αναζήτηση πριν σταλεί κάποιο ερώτημα στο διαδίκτυο. Η εικόνα του κάθε proxy είναι μία δομή δεδομένων η οποία συνδυάζει Bloom Filters και B-Plus Trees. Οι δύο κύριοι παράγοντες που συνεισφέρουν στη μείωση της κυκλοφορίας στο διαδίκτυο είναι: οι εικόνες ενημερώνονται μόνο περιοδικά και οι κατάλογοι είναι πολύ οικονομικοί. Χρησιμοποιώντας cache-queries προσομοιώσεις δείχνουμε ότι ο ρυθμός επιτυχημένης αναζήτησης υποβαθμίζεται αναλογικά με το πλήθος των στοιχείων που εισάγονται στον κατάλογο του κάθε proxy.

Abstract

The sharing of caches among Web proxies is an important technique to reduce Web traffic. This thesis demonstrates the benefits of cache sharing and proposes a new protocol called “sharing caches”. In this protocol, each proxy keeps an image of the cache directory of each participating proxy and checks these images for potential hits before sending any queries. The image in each proxy is a data structure which combines Bloom Filters and B-Plus Trees. The two main factors that contribute to the reduction of Web traffic are: the images are updated only periodically and the directory representations are very economical. Using cache-queries simulations, we show that the cache hit ratio degrades proportionally to the number of elements added in the cache directory of each proxy.

Contents

1 Introduction & Related Work 9
2 Tools 12
2.1 Bloom Filters 12
2.1.2 Counting Bloom Filters 19
2.2 B+ Trees 21
2.2.1 Implementation 22
2.2.2 Characteristics 24
2.3 Hash Functions 25
2.3.1 Optimal number of hash functions 26
3 Implementation 27
4 Performance Evaluation 29
4.1 Experimental Procedure 29
4.2 Experimental Settings 30
4.3 Experimental Results 31
5 Conclusion & Future Work 34
6 References 35

1 Introduction & Related Work

As the tremendous growth of the World Wide Web continues to strain the Internet, caching has been recognized as one of the most important techniques to reduce bandwidth consumption [1]. In particular, caching within Web proxies has been shown to be very effective [2], [3]. To gain the full benefits of caching, proxy caches behind a common bottleneck link should cooperate and serve each other's misses, thus further reducing the traffic through the bottleneck.

Web cache sharing was first proposed in the context of the Harvest project [4], [5]. The Harvest group designed the internet cache protocol (ICP) [6] that supports discovery and retrieval of documents from neighboring caches. Today, many institutions and many countries have established hierarchies of proxy caches that cooperate via ICP to reduce traffic to the Internet [7], [8], [9], [10], [11].

Nevertheless, the wide deployment of web cache sharing is currently hindered by the overhead of the ICP protocol. ICP discovers cache hits in other proxies by having the proxy multicast a query message to the neighboring caches whenever a cache miss occurs. Suppose that N proxies configured in a cache mesh. The average cache hit ratio is H. The average number of requests received by one cache is R. Each cache needs to handle $(N - 1) * (1 - H) * R$ inquiries from neighboring caches. There are a total $N * (N - 1) * (1 - H) * R$ ICP inquiries. Thus, as the number of proxies increases, both the total communication and the total CPU processing overhead increase quadratically.

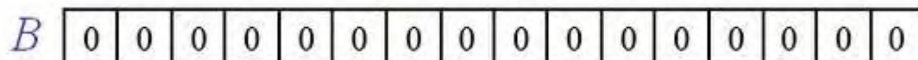
In this paper we propose a new cache sharing protocol called "sharing caches." Under this protocol, each proxy keeps an image of the cache directory of every other proxy. When a cache miss occurs, a proxy first probes all the images to see if the request might be a cache hit in other proxies, and sends a query messages only to those proxies whose images show promising results. The images do not need to be accurate at all times. If a request is not a cache hit when the image indicates so (a false hit), the

penalty is a wasted query message. If the request is a cache hit when the image indicates otherwise (a false miss), the penalty is a higher miss ratio. We combine two instruments to design these images, the first is the Bloom Filter.

Bloom Filters

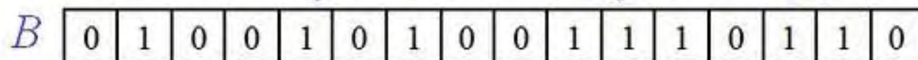
Start with an m bit array, filled with 0s.

B



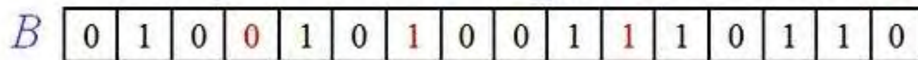
Hash each item x_j in S k times. If $H_i(x_j) = a$, set $B[a] = 1$.

B



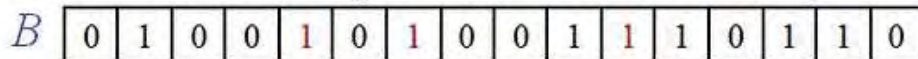
To check if y is in S, check B at $H_i(y)$. All k values must be 1.

B



Possible to have a false positive; all k values are 1, but y is not in S.

B



n items $m = cn$ bits k hash functions

Fig.1: An example of a Bloom Filter

Bloom filters [12] are a powerful technique with many applications. They have been successfully deployed for processing joins in distributed systems, to detect duplicates in data archives, and to speed-up lookups in the Squid cache. Bloom filters have a number of advantages. They are compact and can be implemented efficiently both in space and time. Furthermore, they degrade gracefully; even small Bloom filters are useful and the effectiveness of a Bloom filter increases with its size. One limitation of Bloom filters is that they only work for point queries. In some applications, this limitation is acceptable,

but in many other applications support for range queries is important. Hence the second instrument of our images B-Plus trees.

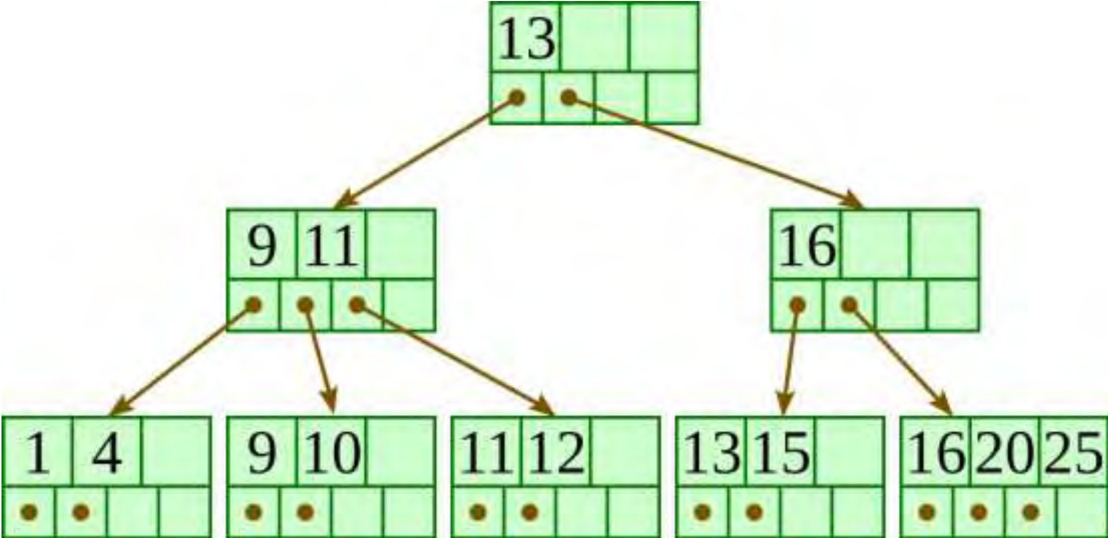


Fig.2: An example of a B+ Tree

A B+ tree is an n-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children.A B+ tree can be viewed as a B-Tree in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves.

2 Tools

2.1 Bloom Filters

A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not, thus a Bloom filter has a 100% recall rate. In other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed (though this can be addressed with a "counting" filter). The more elements that are added to the set, the larger the probability of false positives.

Bloom proposed the technique for applications where the amount of source data would require an impracticably large hash area in memory if "conventional" error-free hashing techniques were applied. He gave the example of a hyphenation algorithm for a dictionary of 500,000 words, out of which 90% follow simple hyphenation rules, but the remaining 10% require expensive disk accesses to retrieve specific hyphenation patterns. With sufficient core memory, an error-free hash could be used to eliminate all unnecessary disk accesses; on the other hand, with limited core memory, Bloom's technique uses a smaller hash area but still eliminates most unnecessary accesses. For example, a hash area only 15% of the size needed by an ideal error-free hash still eliminates 85% of the disk accesses (Bloom (1970)).

More generally, fewer than 10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set (Bonomi et al. (2006)).

2.1.1.1 Algorithm Description

An *empty Bloom filter* is a bit array of *m* bits, all set to 0. There must also be *k* different hash functions defined, each of which maps or hashes some set element to one of the *m* array positions with a uniform random distribution.

To *add* an element, feed it to each of the *k* hash functions to get *k* array positions. Set the bits at all these positions to 1.

To *query* for an element (test whether it is in the set), feed it to each of the *k* hash functions to get *k* array positions. If any of the bits at these positions are 0, the element is definitely not in the set – if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have by chance been set to 1 during the insertion of other elements, resulting in a false positive. In a simple Bloom filter, there is no way to distinguish between the two cases, but more advanced techniques can address this problem.

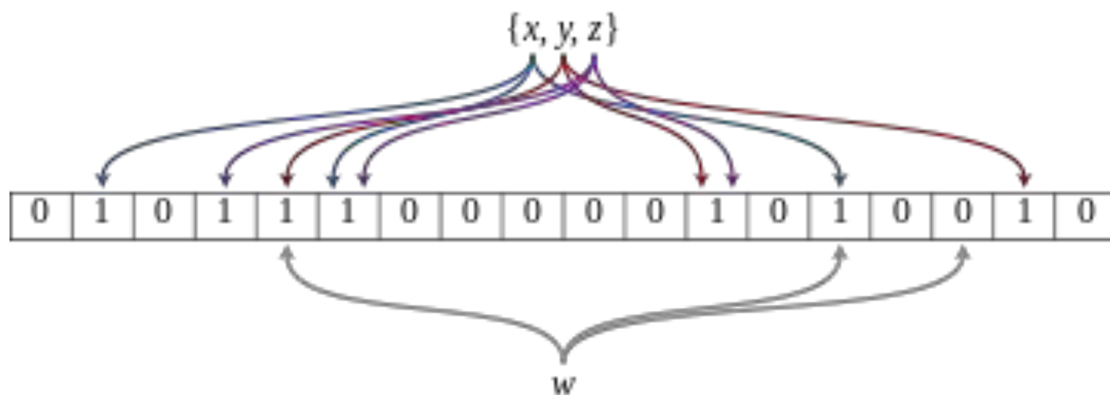


Fig.3: An example of a Bloom filter, representing the set { *x*, *y*, *z* }. The colored arrows show the positions in the bit array that each set element is mapped to. The element *w* is not in the set { *x*, *y*, *z* }, because it hashes to one bit-array position containing 0. For this figure, *m* = 18 and *k* = 3.

The requirement of designing k different independent hash functions can be prohibitive for large k. For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple "different" hash functions by slicing its output into multiple bit fields. Alternatively, one can pass k different initial values (such as 0, 1, ..., $k - 1$) to a hash function that takes an initial value; or add (or append) these values to the key. For larger m and/or k, independence among the hash functions can be relaxed with negligible increase in false positive rate (Dillinger & Manolios (2004a), Kirsch & Mitzenmacher (2006)). Specifically, Dillinger & Manolios (2004b) show the effectiveness of deriving the k indices using enhanced double hashing or triple hashing, variants of double hashing that are effectively simple random number generators seeded with the two or three hash values.

Removing an element from this simple Bloom filter is impossible because false negatives are not permitted. An element maps to k bits, and although setting any one of those k bits to zero suffices to remove the element, it also results in removing any other elements that happen to map onto that bit. Since there is no way of determining whether any other elements have been added that affect the bits for an element to be removed, clearing any of the bits would introduce the possibility for false negatives.

One-time removal of an element from a Bloom filter can be simulated by having a second Bloom filter that contains items that have been removed. However, false positives in the second filter become false negatives in the composite filter, which may be undesirable. In this approach re-adding a previously removed item is not possible, as one would have to remove it from the "removed" filter.

It is often the case that all the keys are available but are expensive to enumerate (for example, requiring many disk reads). When the false positive rate gets too high, the filter can be regenerated; this should be a relatively rare event.

2.1.1.2 Space and time advantages

While risking false positives, Bloom filters have a strong space advantage over other data structures for representing sets, such as self-balancing binary search trees, tries, hash tables, or simple arrays or linked lists of the entries. Most of these require storing at least the data items themselves, which can require anywhere from a small number of bits, for small integers, to an arbitrary number of bits, such as for strings (tries are an exception, since they can share storage between elements with equal prefixes). Linked structures incur an additional linear space overhead for pointers. A Bloom filter with 1% error and an optimal value of *k*, in contrast, requires only about 9.6 bits per element — regardless of the size of the elements. This advantage comes partly from its compactness, inherited from arrays, and partly from its probabilistic nature. The 1% false-positive rate can be reduced by a factor of ten by adding only about 4.8 bits per element.

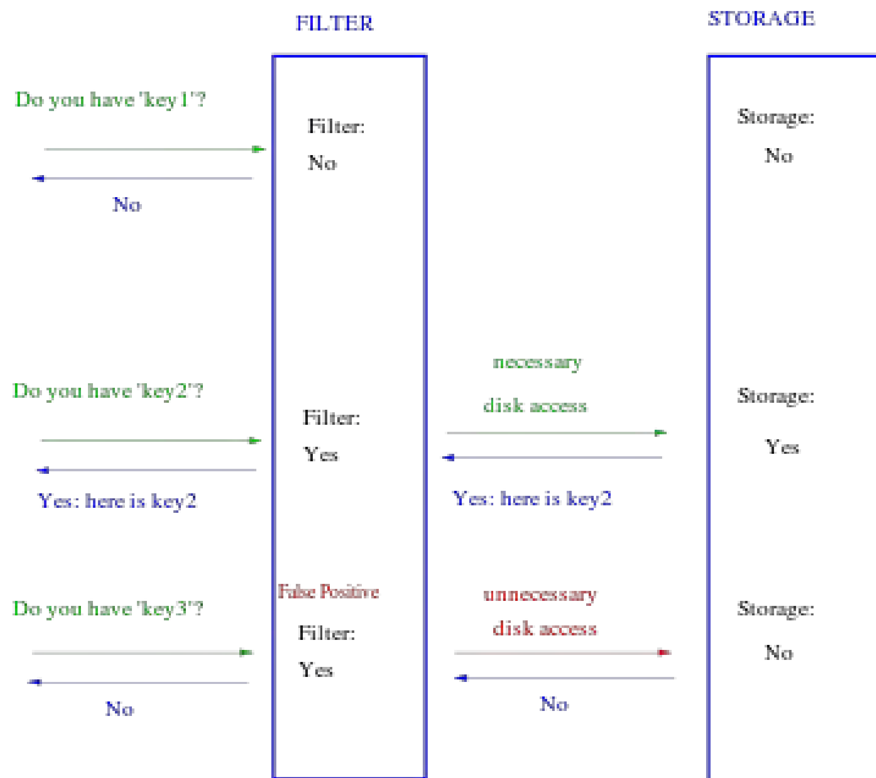


Fig.4: Bloom filter used to speed up answers in a key-value storage system. Values are stored on a disk which has slow access times. Bloom filter decisions are much faster. However some unnecessary disk accesses are made when the filter reports a positive (in order to weed out the false positives). Overall answer speed is better with the Bloom filter than without the Bloom filter. Use of a Bloom filter for this purpose, however, does increase memory usage.

However, if the number of potential values is small and many of them can be in the set, the Bloom filter is easily surpassed by the deterministic bit array, which requires only one bit for each potential element. Note also that hash tables gain a space and time advantage if they begin ignoring collisions and store only whether each bucket contains an entry; in this case, they have effectively become Bloom filters with $k = 1$.

Bloom filters also have the unusual property that the time needed either to add items or to check whether an item is in the set is a fixed constant, $O(k)$, completely independent of the number of items already in the set. No other constant-space set data structure has this property, but the average access time of sparse hash tables can make them faster in practice than some Bloom filters. In a hardware implementation, however, the Bloom filter shines because its k lookups are independent and can be parallelized.

To understand its space efficiency, it is instructive to compare the general Bloom filter with its special case when $k = 1$. If $k = 1$, then in order to keep the false positive rate sufficiently low, a small fraction of bits should be set, which means the array must be very large and contain long runs of zeros. The information content of the array relative to its size is low. The generalized Bloom filter (k greater than 1) allows many more bits to be set while still maintaining a low false positive rate; if the parameters (k and m) are chosen well, about half of the bits will be set, and these will be apparently random, minimizing redundancy and maximizing information content.

2.1.1.3 Probability of false positives

Assume that a hash function selects each array position with equal probability. If *m* is the number of bits in the array, the probability that a certain bit is not set to 1 by a certain hash function during the insertion of an element is $1 - \frac{1}{m}$.

If *k* is the number of hash functions, the probability that the bit is not set to 1 by any of the hash functions is $\left(1 - \frac{1}{m}\right)^k$.

If we have inserted *n* elements, the probability that a certain bit is still 0 is $\left(1 - \frac{1}{m}\right)^{kn}$;

the probability that it is 1 is therefore $1 - \left(1 - \frac{1}{m}\right)^{kn}$.

Now test membership of an element that is not in the set. Each of the *k* array positions computed by the hash functions is 1 with a probability as above. The probability of all of them being 1, which would cause the algorithm to erroneously claim that the element is in the set, is often given as $\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$.

This is not strictly correct as it assumes independence for the probabilities of each bit being set. However, assuming it is a close approximation we have that the probability of false positives decreases as *m* (the number of bits in the array) increases, and increases as *n* (the number of inserted elements) increases.

An alternative analysis arriving at the same approximation without the assumption of independence is given by Mitzenmacher and Upfal. After all *n* items have been added to the Bloom filter, let *q* be the fraction of the *m* bits that are set to 0. (That is, the number of bits still set to 0 is *qm*.) Then, when testing membership of an element not in the set, for the array position given by any of the *k* hash functions, the probability that the bit is found set to 1 is $1 - q$. So the probability that all *k* hash functions find their bit set to 1 is $(1 - q)^k$. Further, the expected value of *q* is the probability that a given array position is left untouched by each of the *k* hash functions for each of the *n* items, which is (as above) $E[q] = \left(1 - \frac{1}{m}\right)^{kn}$.

It is possible to prove, without the independence assumption, that q is very strongly concentrated around its expected value. In particular, from the Azuma–Hoeffding inequality, they prove that $\Pr(|q - E[q]| \geq \frac{\lambda}{m}) \leq 2\exp(-2\lambda^2/m)$

Because of this, we can say that the exact probability of false positives is

$$\sum_t \Pr(q = t)(1-t)^k \approx (1 - E[q])^k = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

as before.

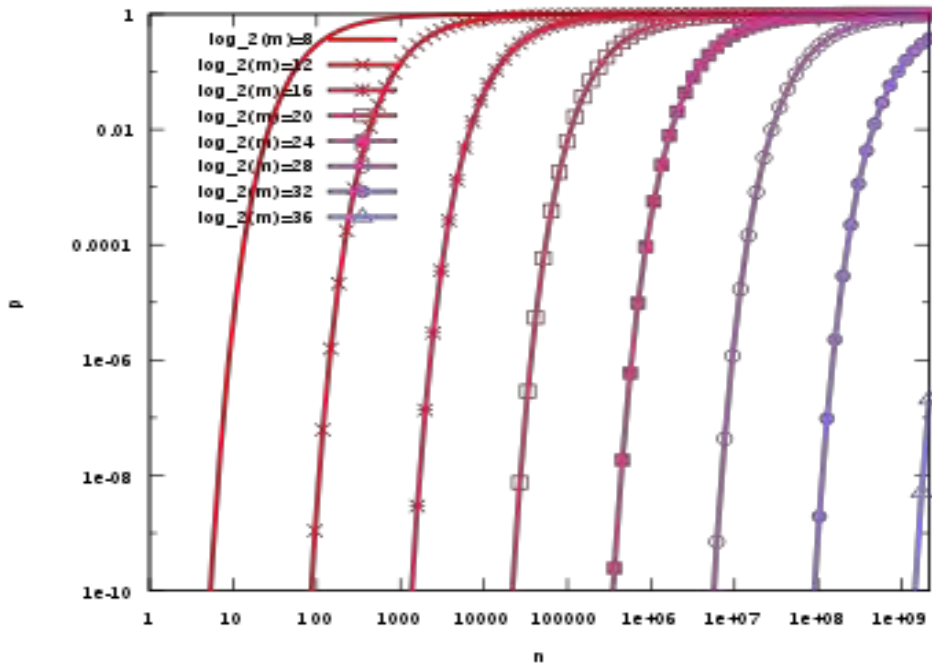


Fig.5: The false positive probability *p* as a function of number of elements *n* in the filter and the filter size *m*. An optimal number of hash functions *k=(m/n)ln2* has been assumed.

2.1.2 Counting Bloom Filters

Counting filters provide a way to implement a *delete* operation on a Bloom filter without recreating the filter afresh. In a counting filter the array positions (buckets) are extended from being a single bit to being an n-bit counter. In fact, regular Bloom filters can be considered as counting filters with a bucket size of one bit. Counting filters were introduced by Fan et al. (1998).

The insert operation is extended to *increment* the value of the buckets and the lookup operation checks that each of the required buckets is non-zero. The delete operation, obviously, then consists of decrementing the value of each of the respective buckets.

Arithmetic overflow of the buckets is a problem and the buckets should be sufficiently large to make this case rare. If it does occur then the increment and decrement operations must leave the bucket set to the maximum possible value in order to retain the properties of a Bloom filter.

The size of counters is usually 3 or 4 bits. Hence counting Bloom filters use 3 to 4 times more space than static Bloom filters. In theory, an optimal data structure equivalent to a counting Bloom filter should not use more space than a static Bloom filter.

Another issue with counting filters is limited scalability. Because the counting Bloom filter table cannot be expanded, the maximal number of keys to be stored simultaneously in the filter must be known in advance. Once the designed capacity of the table is exceeded, the false positive rate will grow rapidly as more keys are inserted.

Bonomi et al. (2006) introduced a data structure based on d-left hashing that is functionally equivalent but uses approximately half as much space as counting Bloom filters. The scalability issue does not occur in this data structure. Once the designed capacity is exceeded, the keys could be reinserted in a new hash table of double size.

The space efficient variant by Putze, Sanders & Singler (2007) could also be used to implement counting filters by supporting insertions and deletions.

Rottenstreich, Kanizo & Keslassy (2012) introduced a new general method based on variable increments that significantly improves the false positive probability of counting Bloom filters and their variants, while still supporting deletions. Unlike counting Bloom filters, at each element insertion, the hashed counters are incremented by a hashed variable increment instead of a unit increment. To query an element, the exact values of the counters are considered and not just their positiveness. If a sum represented by a counter value cannot be composed of the corresponding variable increment for the queried element, a negative answer can be returned to the query.

2.2 B+ Trees

A B+ tree is an n-ary tree with a variable but often large number of children per node. A B+ tree consists of a root, internal nodes and leaves. The root may be either a leaf or a node with two or more children.

A B+ tree can be viewed as a B-tree in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves.

The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, filesystems. This is primarily because unlike binary search trees, B+ trees have very high fanout (number of pointers to child nodes in a node, typically on the order of 100 or more), which reduces the number of I/O operations required to find an element in the tree.

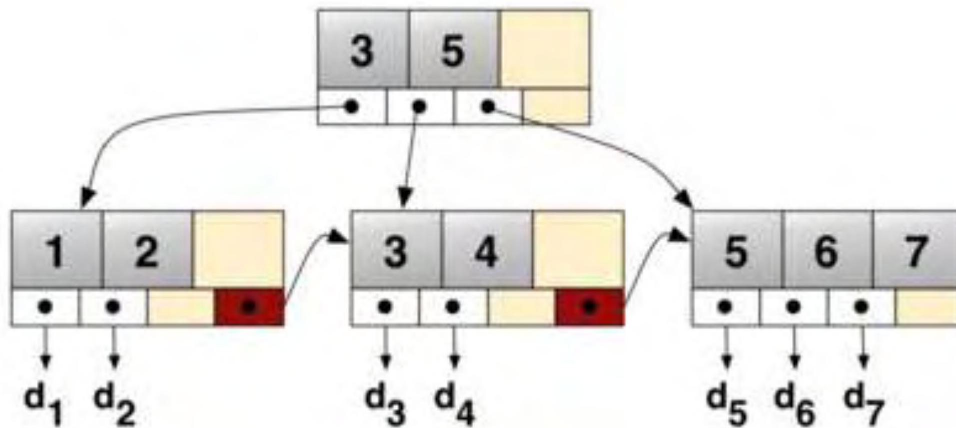


Fig.6: A simple B+ tree example linking the keys 1–7 to data values d_1-d_7. The linked list (red) allows rapid in-order traversal.

2.2.1 Implementation

The leaves (the bottom-most index blocks) of the B+ tree are often linked to one another in a linked list; this makes range queries or an (ordered) iteration through the blocks simpler and more efficient (though the aforementioned upper bound can be achieved even without this addition). This does not substantially increase space consumption or maintenance on the tree. This illustrates one of the significant advantages of a B+tree over a B-tree; in a B-tree, since not all keys are present in the leaves, such an ordered linked list cannot be constructed. A B+tree is thus particularly useful as a database system index, where the data typically resides on disk, as it allows the B+tree to actually provide an efficient structure for housing the data itself.

If a storage system has a block size of B bytes, and the keys to be stored have a size of k, arguably the most efficient B+ tree is one where b=(B/k)-1. Although theoretically the one-off is unnecessary, in practice there is often a little extra space taken up by the index blocks (for example, the linked list references in the leaf blocks). Having an index block which is slightly larger than the storage system's actual block represents a significant performance decrease; therefore erring on the side of caution is preferable.

If nodes of the B+ tree are organized as arrays of elements, then it may take a considerable time to insert or delete an element as half of the array will need to be shifted on average. To overcome this problem, elements inside a node can be organized in a binary tree or a B+ tree instead of an array.

B+ trees can also be used for data stored in RAM. In this case a reasonable choice for block size would be the size of processor's cache line.

Space efficiency of B+ trees can be improved by using some compression techniques. One possibility is to use delta encoding to compress keys stored into each block. For internal blocks, space saving can be achieved by either compressing keys or pointers. For string keys, space can be saved by using the following technique: Normally the ith entry of an internal block contains the first key of block i+1. Instead of storing the full

key, we could store the shortest prefix of the first key of block i+1 that is strictly greater (in lexicographic order) than last key of block i. There is also a simple way to compress pointers: if we suppose that some consecutive blocks i, i+1...i+k are stored contiguously, then it will suffice to store only a pointer to the first block and the count of consecutive blocks.

All the above compression techniques have some drawbacks. First, a full block must be decompressed to extract a single element. One technique to overcome this problem is to divide each block into sub-blocks and compress them separately. In this case searching or inserting an element will only need to decompress or compress a sub-block instead of a full block. Another drawback of compression techniques is that the number of stored elements may vary considerably from a block to another depending on how well the elements are compressed inside each block.

2.2.2 Characteristics

For a *b*-order B+ tree with *h* levels of index:

- The maximum number of records stored is $n_{max} = b^h - b^{h-1}$
- The minimum number of records stored is $n_{min} = 2\left\lceil \frac{b}{2} \right\rceil^{h-1}$
- The minimum number of keys is $n_{kmin} = 2\left\lceil \frac{b}{2} \right\rceil^{h} - 1$
- The maximum number of keys is $n_{kmax} = n^h$
- The space required to store the tree is $O(n)$
- Inserting a record requires $O(\log_b n)$ operations
- Finding a record requires $O(\log_b n)$ operations
- Removing a (previously located) record requires $O(\log_b n)$ operations
- Performing a range query with *k* elements occurring within the range requires $O(\log_b n + k)$ operations
- Performing a pagination query with page size *s* and page number *p* requires $O(p * s)$ operations

2.3 Hash Functions

A hash function is any function that can be used to map digital data of arbitrary size to digital data of fixed size, with slight differences in input data producing very big differences in output data. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. One practical 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 matches a stored hash value, but makes it hard to reconstruct the data from the hash alone. This principle is used by the PGP algorithm for data validation and by many password checking systems.

Hash functions are related to (and often confused with) checksums, check digits, fingerprints, 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 catalog of file fingerprints than of hash values.

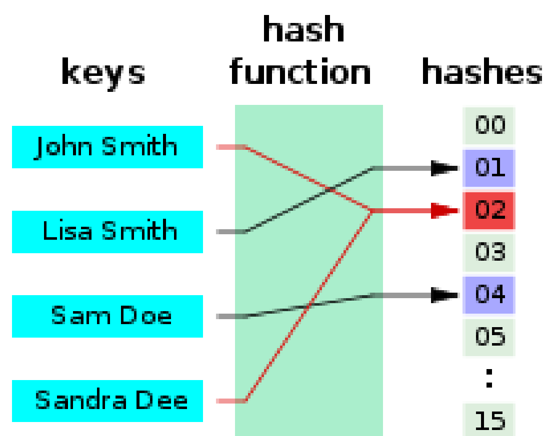


Fig.7: A hash function that maps names to integers from 0 to 15. There is a collision between keys "John Smith" and "Sandra Dee".

2.3.1 Optimal number of hash functions

For a given *m* and *n*, the value of *k* (the number of hash functions) that minimizes the probability is $k = \frac{m}{n} \ln 2,$ which gives $2^{-k} \approx 0.6185^{m/n}$.

The required number of bits *m*, given *n* (the number of inserted elements) and a desired false positive probability *p* (and assuming the optimal value of *k* is used) can be computed by substituting the optimal value of *k* in the probability expression above: $p = \left(1 - e^{-(m/n \ln 2)n/m}\right)^{(m/n \ln 2)}$ which can be simplified to: $\ln p = -\frac{m}{n} \left(\ln 2\right)^2$.

This results in: $m = -\frac{n \ln p}{(\ln 2)^2}$.

This means that for a given false positive probability *p*, the length of a Bloom filter *m* is proportionate to the number of elements being filtered *n*. While the above formula is asymptotic (i.e. applicable as $m,n \to \infty$), the agreement with finite values of *m*, *n* is also quite good; the false positive probability for a finite bloom filter with *m* bits, *n* elements, and *k* hash functions is at most $\left(1 - e^{-k(n+0.5)/(m-1)}\right)^k$.

So we can use the asymptotic formula if we pay a penalty for at most half an extra element and at most one fewer bit.

3 Implementation

This section describes the main ideas of the “sharing caches” technique. It shows how each proxy-cache filter range queries which is the innovation of this thesis. Bloom filter is the best tool for point queries in both, look up time and space efficiency. There has been a development in range-Bloom Filters for range queries but their performance is poor at best. In a range-based Bloom filter, each cell represents a whole range of values, rather than a single value in a classic Bloom filter. Range-based Bloom filters, thus, take an additional parameter, r, which determines the granularity (or size of the range). For r = 1, range-based and classic Bloom filters are the same. It is virtually impossible to set the r parameter correctly. Depending on the query workload and the database, a different setting of r is needed and even minimal changes to the optimum setting can have dramatic impact on the performance. Hence, we implemented the B-Plus Trees.

We begin our implementation by constructing a Counting Bloom Filter (to perform the delete operation of extracted elements) from the data stored in each cache. The bucket size is set to 4 bits which is more than enough to store twice the number of the original keys in the caches. A simple hash function was used to map the elements of the proxy cache to the bloom set because we were more interested in the optimal number of hash functions for best efficiency of the Filter rather than the function itself. When the bit array is created, we implement it in the B-Plus Tree in the following manner. As we mentioned earlier, B-Plus Trees store data only in their leaf nodes, the rest of the nodes store only pointers to lower layers of the tree. Our tree will have the exact number of elements as the Bloom Filter and each leaf will store one bucket of the bloom set, meaning the one significant bit (0 or 1) not all 4 bits of the Counting Bloom Filter for space efficiency. The leaves of the B-Plus Tree are connected with a linked list which makes look up time for range queries significantly faster than any other look up method or tool. But to gain the full advantage of this method we need the leaves of the tree that we are searching to be as consecutive as possible. In order to achieve that, after hashing all the elements of our range query to their respective positions in the tree, we assort them into one list (the

list can have duplicate numbers) and then we begin traversing the tree. The idea is that after finding the left bound of our query, we will use the linked list of the tree to check the rest of the hashed positions of each element. When we reach a gap between our sorted hashed positions, we check whether the gap is bigger than the height of the tree, if so we traverse the tree from the root to that position and we continue checking the list from there, if not we move right in the list until we reach the next leaf of the query. Traversing between the nodes of a B-Plus Tree takes roughly the same time as moving through the linked list because the nodes only store pointers to lower nodes (or leaves in the last layer). If the hashed positions are too far apart from each other (in case we only search for one element) we use directly the bloom set because as mentioned before Bloom Filters are the most efficient method for point queries.

When the whole list is checked for potential hits, a hit rate is generated for each proxy. If a proxy cache has all our elements we probe the query to that, else to the proxies with the highest percentages. Since the collection of cooperating proxies is relatively static, the proxies can just maintain a permanent TCP connection with each other to exchange query and update messages fast. The update messages of the new trees of each proxy will be broadcasted when a fixed percentage of elements are changed in their cache, to keep the false positive percentage to a minimum. The trees of which the update messages are consisted for each proxy are relatively small as we will demonstrate in the following experiments.

4 Performance Evaluation

In this section we present the results and the experimentation analysis of the methods we use.

4.1 Experimental Procedure

We applied the “sharing caches” technique to 4 proxy caches, each with different capacity. The first leg of our experiments was calculating the time (in ms) it took for each proxy to search for different amount of elements(urls). The second was measuring the false positive degradation as elements changed or elements being added in each cache. Below are the tables for the first leg of our results.

2000 url	
%(%)	time(ms)
0	0
5	1192
50	1686
75	2480

5000 url	
%(%)	time(ms)
0	0
25	2926
50	5952
75	9925

7000 url	
%(%)	time(ms)
0	0
25	4471
50	10363
75	13932

10000url	
%(%)	time(ms)
0	0
25	6994
50	18863
75	38465

4.2 Experimental Settings

We begun the first part of our experiments by applying the “sharing caches” method to a proxy cache of 2000 urls, then to a cache of 5000 urls, then to a cache of 7000 urls and finally to one of 10000 urls. For each cache a Counting Bloom Filter was created and 14 hash functions were used to map the urls to 14 different positions in each bloom set in order to achieve a starting false positive probability of 0.0001%. The bloom set of the first cache(2000 urls) has 38.341 bits, the second one(5000 urls) has 95.851 bits, the third(7000 urls) 134.191 bits and the last(10000 urls) 191.702 bits. Then we created the B-Plus Trees for each proxy from their respective bloom set and started measuring look up times for 25%, 50% and 75% of cached elements (urls) for each proxy cache.

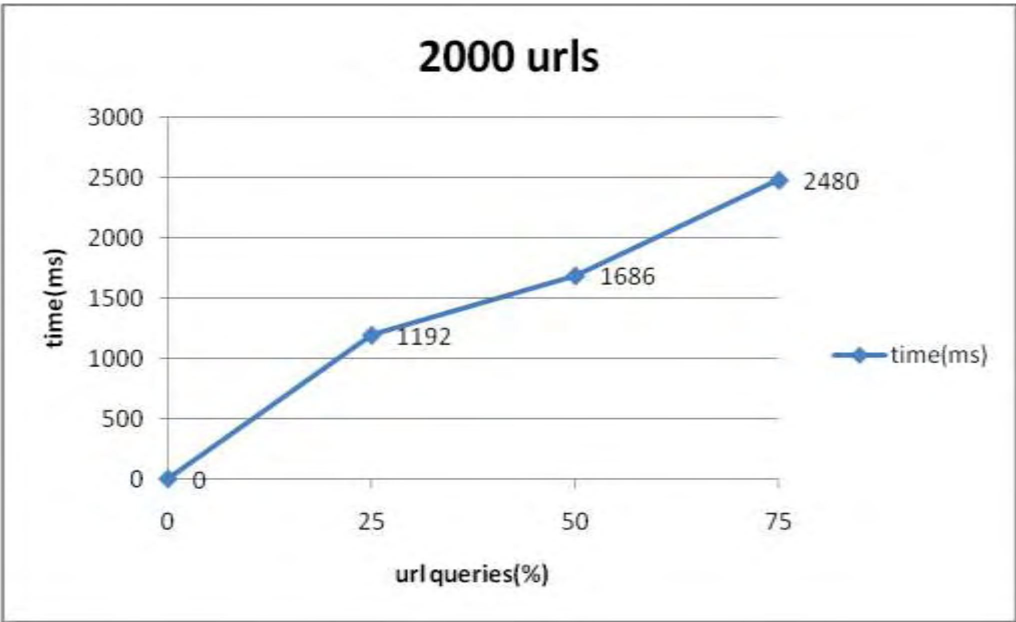
The second part of the experiments has uniform application in all 4 proxy caches, it concerns the false positive degradation. Using trace-driven simulations, we show that the update of the B+Trees can be delayed until a fixed percentage, in our case 5% of cached elements are new, and the hit ratio will degrade proportionally depending on traces, while maintaining a 0% false negative probability. Furthermore we ran simulations for the addition of cached elements 10%, 25% and 50% and we show that the hit ratio degrades proportionally in this situation too. Below are the tables for the second part or our results.

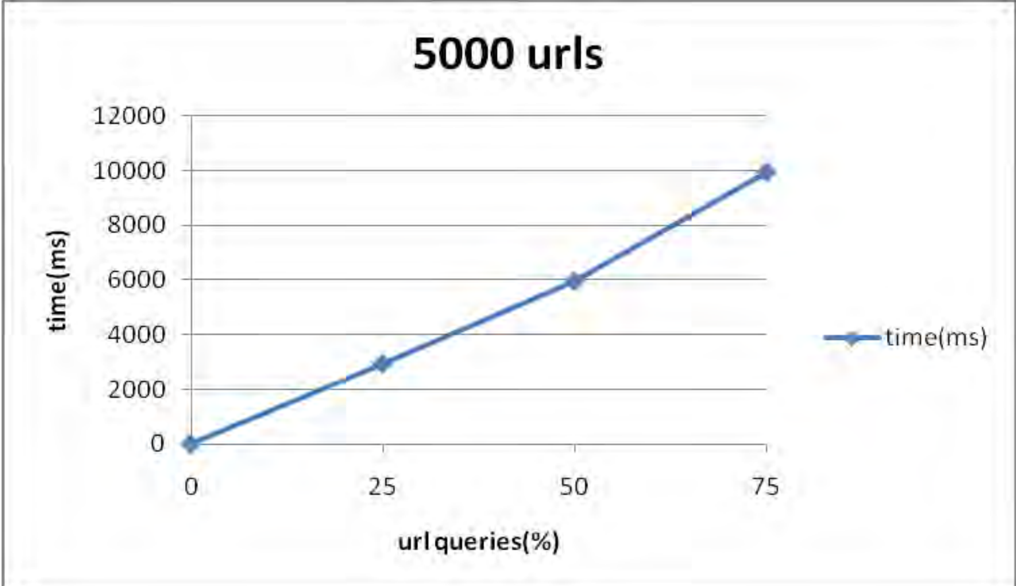
change	false positive
1%	0,02%-1,7%
2,50%	0,08%-5,5%
5%	0,2%-10%

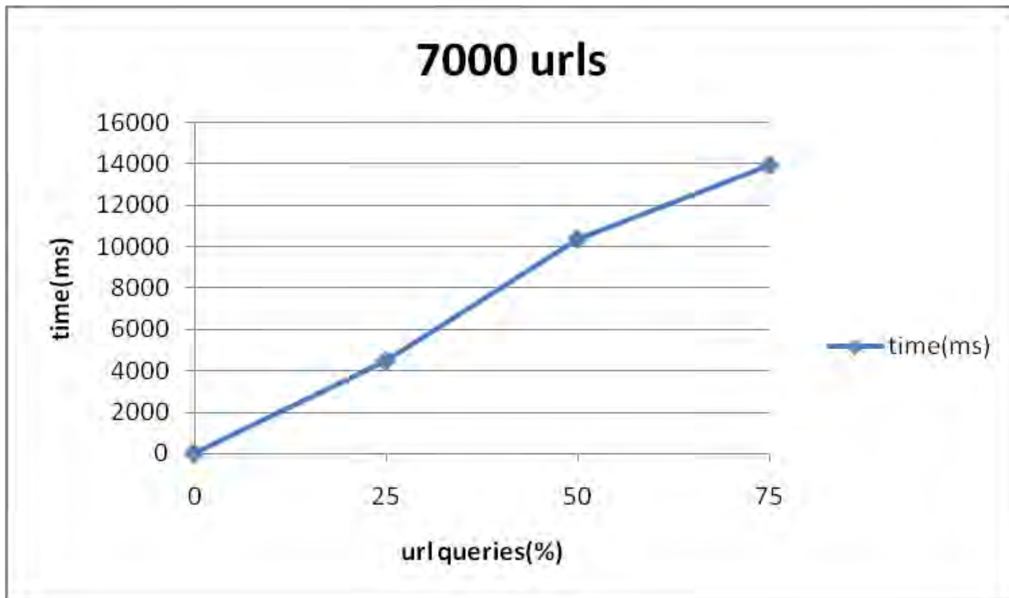
addition	false positive
10%	0,00024
25,00%	0,00075
50%	0,00335

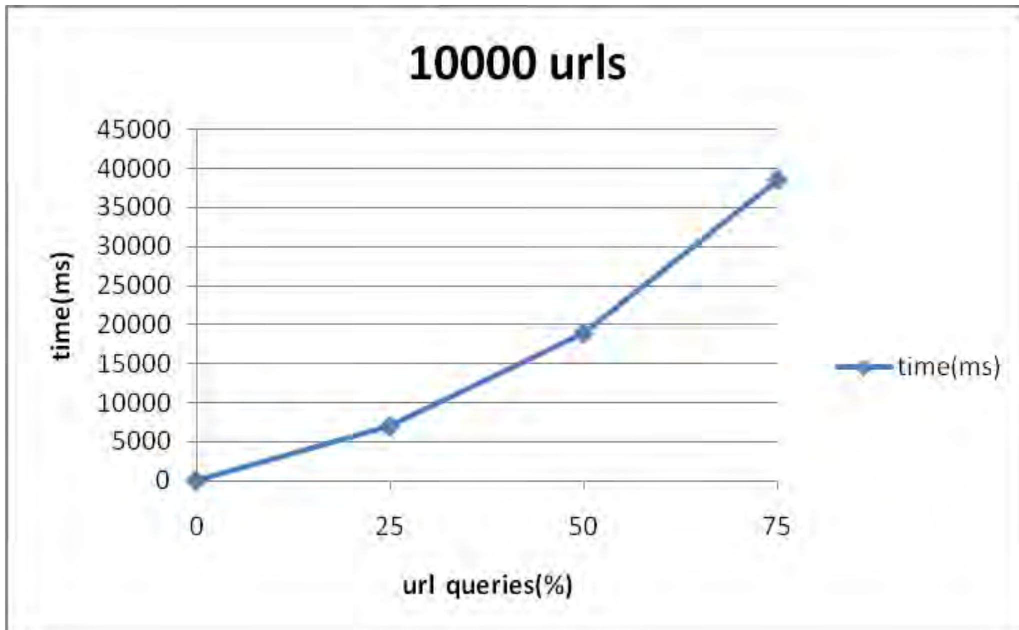
4.3 Experimental Results

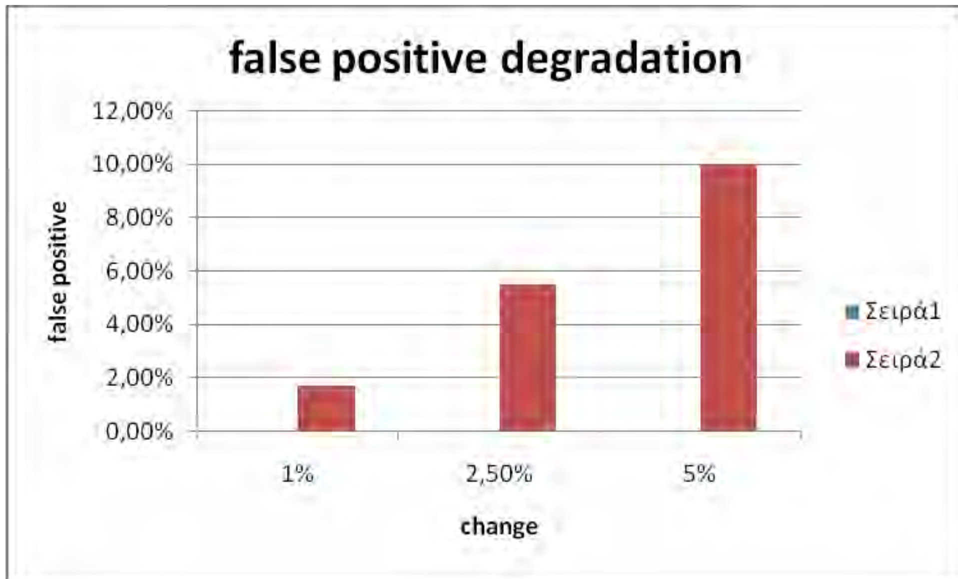
In this section we present the results of our experiments. The first set of results concerns the look up times for elements in each cache and the second one is about the false positive degradation.

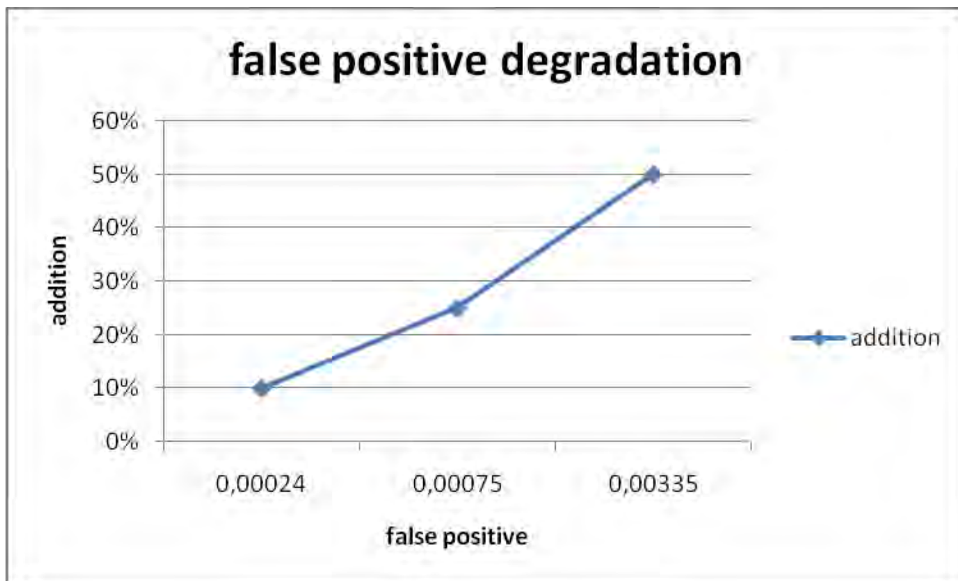





7000 urls
16000
14000
12000
10000
8000
6000
4000
2000
0
time(ms)
0
25
50
75
url queries(%)
time(ms)


10000 urls
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
time(ms)
0
25
50
75
url queries(%)
time(ms)


false positive degradation
12,00%
10,00%
8,00%
6,00%
4,00%
2,00%
0,00%
false positive
1%
2,50%
5%
change
Σειρά1
Σειρά2


false positive degradation
60%
50%
40%
30%
20%
10%
0%
addition
0,00024
0,00075
0,00335
false positive
addition

5 Conclusion & Future Work

It is clear that our technique is beneficial for range queries and most efficient for wide range. The more elements we search for in a cache the more hashed positions we will be seeking for in the B-Plus Tree creating consecutive hits in the list of leaves. Larger number of proxies generates fewer queries to the Web too. Thus, the scalability of our method becomes transparent.

As for future work, we plan on optimizing the B-Plus Trees at the cost of space efficiency though. The nodes in our trees only store pointers to lower nodes(or leaves), we could minimize the search time of the trees by having nodes not only store pointers but also information about the lower levels of the tree. For example, if there is a large amount of leaves that does not store any element of the cache higher nodes could carry that information to avoid search time while traversing down the tree. Since most of our search will be done through the list it will not save a lot of look up time but it will be nonetheless an improvement in our technique. Even though each proxy will need more space, with the decreasing values of cache memory in our era we believe this to be a worthwhile adaptation for our method.

6 References

[1] V. Jacobson. How to kill the internet. presented at SIGCOMM'95 Middleware Workshop, Aug. 1995. [Online]. Available: ftp://ftp.ee.lhl.gov/talks/vj-webflame.ps.Z

[2] B. M. Duska, D. Marwood, and M. J. Feeley, "The measured access characteristics of world-wide-web client proxy caches," in *Proc. USENIX Symp. Internet Technology and Systems*, Dec. 1997.

[3] T. M. Kroeger, D. D. E. Long, and J. C. Mogul, "Exploring the bounds of web latency reduction from caching and prefetching," in *Proc. USENIX Symp. Internet Technology and Systems*, Dec. 1997.

[4] The Harvest Group. (1994) Harvest Information Discovery and Access System. [Online]. Available: http://excalibur.usc.edu/

[5] P. B. Danzig, R. S. Hall, and M. F. Schwartz, "A case for caching file objects inside internetworks," in *Proc. SIGCOMM*, 1993, pp. 239–248.

[6] ICP working group. (1998). National Lab for Applied Network Research. [Online]. Available: http://ircache.nlanr.net/Cache/ICP/

[7] C. Grimm. The dfn cache service in B-WiN. presented at 2nd Web Caching Workshop, Boulder, CO, June 1997. [Online]. Available: http://www-cache.dfn.de/CacheEN/

[8] J. Jung. Nation-wide caching project in korea. presented at 2nd Web Caching Workshop, Boulder, CO, June 1997. [Online]. Available: http://ircache.nlanr.net/Cache/Workshop97/

[9] J. Pietsch. Caching in the Washington State k-20 network. presented at 2nd Web Caching Workshop, Boulder, CO, June 1997. [Online]. Available: http://ircache.nlanr.net/Cache/Workshop97/

[10] K. Beck. Tennessee cache box project. presented at 2nd Web Caching Workshop, Boulder, CO, June 1997. [Online]. Available: http://ircache. nlanr.net/Cache/Workshop97/

[11] B. M. Duska, D. Marwood, and M. J. Feeley, "The measured access characteristics of world-wide-web client proxy caches," in *Proc. USENIX Symp. Internet Technology and Systems*, Dec. 1997.

[12] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13(7):422–426, 1970.

[13] www.vldb.org/pvldb/vol6/p1714-kossmann.pdf

[14] *pages.cs.wisc.edu/~cao/papers/**summarycache**.html*

[15] http://en.wikipedia.org/wiki/Bloom_filter

[16] http://en.wikipedia.org/wiki/B-tree
[17] https://github.com/Baqend/Orestes-Bloomfilter/tree/master/src/main/java/orestes/bloomfilter
[18] http://stackoverflow.com/questions/14204776/b-tree-in-java