1: List of specifications of all methods for each class

`Class-name` description of class

method-name(parameters) description of method

`BloomFilterDet` deterministic hash function for implementing a Bloom filter

BloomFilterDet(int setSize, int bitsPerElement) Creates a Bloom filter that can store a set S of cardinality `setSize`. The size of the filter is `setSize`$*$`bitsPerElement`. The number of hash functions is the optimal choice, that is, $\frac{\ln 2 \times \texttt{filterSize}}{\texttt{setSize}}$; since it has to be an integer, we might choose a value less than optimal to reduce computational overhead.

add(String s) Adds the string s to the filter. Type of this method is `void`. This method is case-insensitive; more precisely, it first obtains all the characters in the string in uppercase then adding the uppercase string to the filter. For example, "Galaxy" will be turned into "GALAXY".

appears(String s) Returns `true` if s appears in the filter; otherwise returns `false`. This method is also case-insensitive and uses s in the uppercase form in operation.

hash64(String s) Returns the hash value of the string s. The algorithm is based on the deterministic hash function 64-bit FNV. The details are described in the next section.

`BloomFilterRan` random hash function for implementing a Bloom filter

BloomFilterRan(int setSize, int bitsPerElement) Create a Bloom filter that can store a set S of cardinality `setSize`. The size of the filter is `setSize` $*$ `bitsPerElement`, for which number, if not prime, it increments to a prime number. The number of hash functions is the optimal choice, that is, $\frac{\ln 2 \times \texttt{filterSize}}{\texttt{setSize}}$; since it has to be an integer, we might choose a value less than optimal to reduce computational overhead.

hashFunc() Generates `hashFunNum` random hash functions for use with Bloom filter bit array. Type of this method is `void`. The details are described in a later section.

nextPrime(int n) Finds and return the next prime number larger than n.

add(String s) Adds the string s to the filter. Type of this method is `void`. This method is case-insensitive; more precisely, it first obtains all the characters in the string in uppercase then adding the uppercase string to the filter. For example, "Galaxy" will be turned into "GALAXY".

appears(String s) Returns `true` if s appears in the filter; otherwise returns `false`. This method is also case-insensitive and uses s in the uppercase form in operation.

isPrime(int n) Return `true` if n is a prime number; otherwise returns `false`. This is based on sieve of eratosthenes, `http://en.wikipedia.org/wiki/Primality_test`.

`BloomFilter` superclass of `BloomFilterDet` and `BloomFilterRan`

caseInsensitive(String s) Returns a string where all characters are uppercase based on the given string.

totalBitsSet Returns the number of bits that are set to one in the filter (using the class `BitSet`).

totalBitsSetPercentage Returns the percentage number that bits are set to one in the filter, namely, $\frac{\texttt{totalBitsSet()}}{\texttt{filterSize}}\%$.

filterSize() Returns the size of the filter (the size of the table).

dataSize() Returns the number of elements added to the filter.

numHashes() Returns the number of hash function used.

`FalsePositives` false positive rate evaluation experiment

FalsePositives(String file1Path, String file2Path, int[] bitsPerElement) Create experiments to estimate the false positive probability of both the filter, `BloomFilterDet` and `BloomFilterRan`. `file1Path` and `file2Path` are the paths to, respectively, file where all the words are desired to be added into each of the Bloom filters, and file where all the words are needed to be checked in each of the bloom filters. The elements in `bitsPerElement` control the the initial size for the bloom filter. Please see more details in a later section about the design.

countSetSize() Returns the number of strings to be added to the filter in the file for the information of `setSize`. Type of this method is `void`.

readFile1ToBuildFilter(int setSize, int bitsPerElement, int stringNum) Creates both the filters `BloomFilterDet` and `BloomFilterRan` for the given `setSize` and `bitsPerElement`. `stringNum` indicates the number of strings that we would like to add to the filters. It returns the false positive rates of both filters.

readFile2ToTestFilter(BloomFilterDet BFDet, BloomFilterRan BFRan) Checks whether each string in the file is determined as appearance. Here we count each appearance as a false positive as the two files do not share a string. It returns the false positive rates of both filers, `BFDet` and `BFRan`.

estimateFalsePositive() Conducts experiments to obtain false positive probability estimates with defined pairs of `setSize` and `bitsPerElement`. Type of this method is `void`. It returns the resulting array of false positive rates.

`NaiveDifferential` Key first checked in differential file then database file

NaiveDifferential(File databaseFile, File differenFile) Takes the database and differential files and stores as `databaseFile` and `differenFile`, respectively.

retrieveRecord(String key) Gets a key as parameter and returns the record by consulting the files.

tryDiffFile(String key) Consults `differenFile` and returns the record given `key` as query.

tryDataBase(String key) Accesses `databaseFile` and returns the record given `key` as query.

SearchThe(File file, String key) Returns the first found record if `key` appears in `file`; otherwise returns `null`.

`BloomDifferential` Bloom filter of all keys in file containing records that are changed

BloomDifferential(File databaseFile, File differenFile) Takes and stores `databaseFile` and `differenFile`.

createFilter(int setSize, int bitsPerElement, String method) Creates a Bloom filter, given `setSize` and `bitsPerElement`, corresponding to the records in the differential file `differenFile`. The hash function type is specified by `method`, deterministic or random.

retrieveRecord(String key, String method) Returns the record corresponding to the record by consulting the Bloom filter first, given `key` and `method` as key and Bloom filter hash function type, respectively.

tryDiffFile(String key) Consults `differenFile` and returns the record given `key` as query.

tryDataBase(String key) Accesses `databaseFile` and returns the record given `key` as query.

SearchThe(File file, String key) Returns the first found record if `key` appears in `file`; otherwise returns `null`.

`EmpiricalComparison` Performance comparison between `NaiveDifferential` and `BloomDifferential`

EmpiricalComparison(String DBFilePath, String DFFilePath, String testFilePath) Stores the path to the database, differential and test files and stores as `DBFilePath`, `DFFilePath` and `testFilePath`, respectively.

runNaiveDifferntial() Performs queries without a Bloom filter. Type of this method is `void`.

runBloomDifferential(int setSize, int bitsPerElement, String method) Performs queries with a Bloom filter with specified `setSize`, `bitsPerElement` and hash function type. Type of this method is `void`.

`BloomFilterApp` Bloom filter application

main(String[] args) Runs `FalsePositive` and `EmpiricalComparison`.

countSetSize(String diffFileName) Returns the number of strings to be checked in the filter from `differenFile`.

generateTestFiles(String database, double[] sizeFromDB, String diffFile, double[] sizeFromDF, int unit) Generates test files of various sizes. Test files contains keys from the database and differential files with specified numbers. `sizeFromDB`$*$`unit` and `sizeFromDF`$*$`unit` indicate the numbers from each files to be randomly picked into the test files.

2: The process via which k-hash values are generated using the deterministic hash function

To add an element, bloom filters hashes the element, identifying k positions in the bit array, and setting those bits to 1 (A particular bit might be set to 1 multiple times.). We know the probability of a false positive is minimized for $k = \ln 2 \times \frac{m}{n}$, where m is the filter size and n is the set size. In fact k must be an integer and in practice we might chose a value less than optimal to reduce computational overhead. We use the class `BitSet` to implement a vector of m bits, initially all set to 0. To check for an element, we essentially do the same thing and then check to see if all the bits at those positions are set.

Here we implemented the FNV hash (`http://www.isthe.com/chongo/tech/comp/fnv/`). The The hash function takes a sequence of integers and produce a small integer value. The sequence of integers might be a list of integers or it might be an array of characters (a string). When using the algorithm, the input are represented in unsigned integers because signed integers may results in odd behavior. We walk down the sequence of integers, doing operations on the integers one by one; we XOR the next character into the hash value, multiply the current hash value by a carefully chosen constant, and output `hash-value` $=$ `hash-value` $\mod 2^64$.

Then, let h_0 be the hash value obtained as above, for a given input string. To obtain k different hash values, we then use the formula, $g_i(x) = h_1(x) + i * h_2(x), \forall 0 \leq i \leq k-1$, where $g_i(x)$ is the k-th hash value for

the given input string x. For the $h_1(x)$ and $h_2(x)$, we use the first 32 bits of $h_0(x)$ for $h_1(x)$, and second 32 bits for $h_2(x)$. And to set the corresponding bits in the filter, we then do $g_i(x) \mod m$, where m is the size of the hash table. This procedure gets the same hash value for any number of times we apply it onto the same x. There is a paper determines that by this way (simulating), the performance is actually similar to actually running k hash functions (`http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.72.2442`). It starts with a single input and arrive at k different output but consistent every time.

3: The process via which k-hash values are generated using the random hash function

The idea is to choose k independent hash functions, $h_1, h_2, \ldots, h_k$. For each element $s \in S$, the bits at positions $h_1(s), h_2(s), \ldots, h_k(s)$ in the vector constructed by the `BitSet` class are sets to 1. Given a query for q we check the bits at positions $h_1(q), h_2(q), \ldots, h_k(q)$. If any of them is 0, then certainly q is not in the set S. Otherwise, we conjecture that q is in the set. To create a bloom filter with random hash functions, the provided `setSize*bitsPerElement` is a lower bound of the table size; if the value is not prime, the actual length of the bloom filter is set to the next larger prime number, which is the filter size in practice.

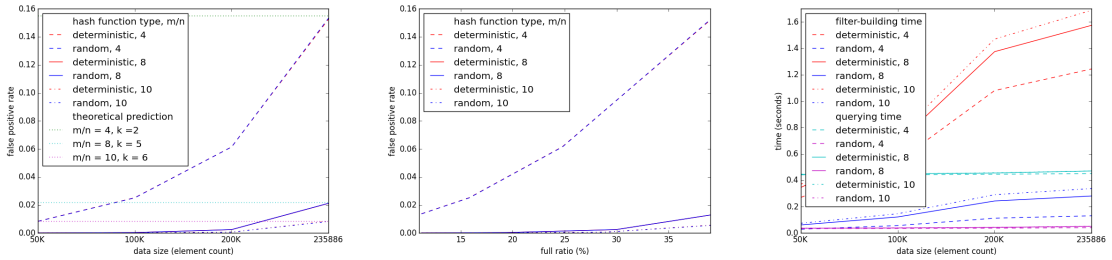
To generate a random hash function for strings, two random coefficients are required, a and b, which are randomly picked from $0, 1, \ldots, p-1$, where p is the prime length of the bloom filter bit array. Then the random hash function is $ax + b \mod m$, where x and m are the input string and hash table size, respectively. This procedure is repeated multiple times to obtain the necessary number of hash functions and make sure there are no identical pairs of coefficients for any two hash functions, meaning $(a_i, b_i) \neq (a_j, b_j), \forall i \neq j, 0 \leq i, j \leq k-1$. To obtain the hash value for a given string x, we walk down the characters of string, doing operations on the characters represented by integers one by one; we XOR the next character into the hash value, multiply the current hash value by a_i and add with b_i, and output `hash-value` $=$ `hash-value` $\mod m$.

4: The experiment design to compute false positives

For the false positive test, we added all the strings in `/usr/share/dict/web2.txt` (on Mac or some Linux systems or `http://code.metager.de/source/xref/freebsd/share/dict/web2`) into each of the bloom filters. To check for false positives, we check for all the strings in `/usr/share/dict/web2a.txt` (on Mac or some Linux systems or `http://code.metager.de/source/xref/freebsd/share/dict/web2a`) in each of the bloom filters. The two files should have completely different sets of strings (as far as I can tell), so for each string that is determined to be contained in the filter for a given string from `web2a.txt` is a false positive. `web2.txt` and `web2a.txt` are roughly 2.37 MB (235,886 strings) and 988 KB (76,205 strings), respectively.

For each bloom filter, we ran tests using the following combinations:

```
for i in elementCount[235886, 200000, 100000, 50000]
    for j in bitsPerElement[4, 8, 10]
        run test with bitsPerElement[j] and elementCcount[i]
    end
end
```



(a) False positive rate v.s. data size (b) False positive rate v.s. full ratio (c) Time versus data size

Figure 1: Comparison between 64-bit FNV and random hash function

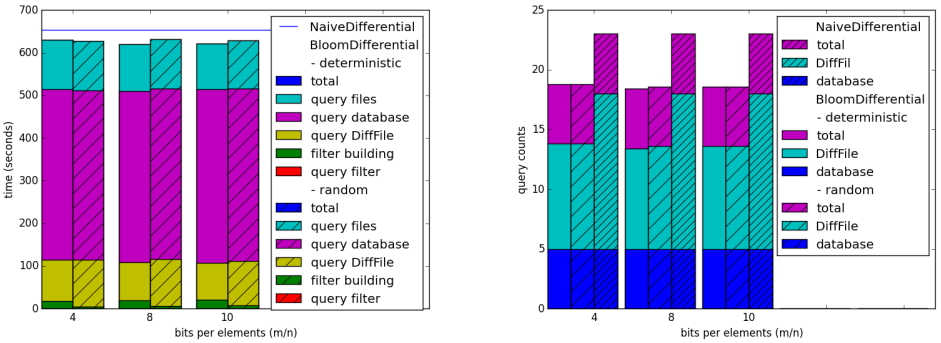
Element count in this is the number of strings randomly picked from `web2a.txt`; this allows us to learn the fill ratio (how full the bit array be). The idea for the element count is to see how the bloom filters will perform when it has a different fill ratio. For the performance test, we added all the words in `web2.txt` into the bloom filters. We then continue to loop through the randomly generated strings from `web2a.txt` until given element count is met to calculate false positives. Some of the words could be picked more than one times, so we ran the experiment multiple times to obtain the average. The test results are presented in the section next.

5: False positives comparisons

Fig. 1. show performance comparison between `BloomFilterDet` and `BloomFilterRan`. We basically wanted to study how the false positive rate varied for different test case and hoe the execution time changes. The results were averaged over more than a thousand runs of randomly generated test scenarios (strings picked from `web2a.txt`). Fig. 1a. reports the false probability estimates when `bitsPerElement` are set to different values, i.e., $\frac{m}{n} = 4$, 8 or 10. The red lines depict FNV's performance, whereas reds record performacne of random hash function. Both approaches have little difference in terms of false positive rate (but `BloomFilterRan` has samller false positives); the lines are almost overlapping for given `bitsPerElement`.

On the other hand, it is observed that when the value of bits-per-element is increased, for the same data size, the false positive rate can be reduced. This is because the filter size is increased and the optimal hash function number is selected to further decrease false positive rate, whereas for fixed `bitsPerElement`, the false positive rate deteriorates as dealing with more data without changing a better number of hash functions, resulting a fuller filter. This is verified in Fig. 1b., where the false positive rate grows with the full ratio of the filter. We also plot the theoretical predictions in Fig. 1a. as reference (from `http://pages.cs.wisc.edu/~cao/papers/summary-cache/node8.html`); for the performed scenarios, the false positive rates are reasonable and satisfactory as below the theoretical probabilities.

However, there seems to be trade-off between computation complexity and false positive rate. Fig. 1c. presents the total time for building and querying the filers during the operations. Although the querying time behaves quite consistent (relatively stable no matter the data size and filter size), the filter-building time depends on a few factors. From the figure, it is evident that using more hash functions causes more computation time while building filters for both type of hash functions. It then is another issue how to decide what `bitsPerElement` value to choose when the time spent on creating a filter matters. Besides, Different hash function methods take different time magnitude for hash value computation, for example, in our experiment, 64-bit FNV hash functions took longer time than random hash functions to construct and



(a) Computation time v.s. bits per element (b) Query counts v.s. bits per element

Figure 2: Performance comparison between naive and Bloom filter based method

query filters.

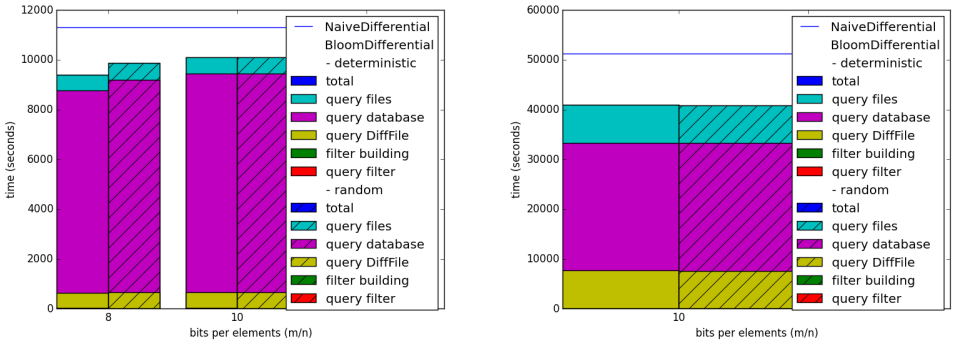
6: Empirical comparisons

Bloom filters can be used in some low-level applications that involve file management. An example of that is *differential files*. Consider a large database of records where each record is of the form $\langle key, value \rangle$, which can both be string. Suppose that such a data is stored in a single file, named `databse.txt`, which has one record per line, and there is another file, named `DiffFile.txt`. Whenever a particular record is changed, the that record is written to `DiffFile.txt`, but not `database.txt` yet. Once `DiffFile.txt` becomes large enough and when the load on the server is low, `database.txt` is updated and all contents of `DiffFile.txt` are removed. Typically, the number of records in `DiffFile.txt` is less than 10 percent of that in `database.txt`.

Now consider the following query mechanism. Users want to retrieve the *value* associated with a key. `NaiveDifferential` first checks whether *key* appears in `DiffFile.txt`; if not, it accesses `database.txt`. Since the number of records in `DiffFile.txt` is much smaller than that in `database.txt`, for most of the queries, `NaiveDifferential` accesses both the files. The efficiency can be improved via Bloom filters. `BloomDifferential` creates a filter of all keys in `DiffFile.txt` and stores the filter in main memory. When a key arrives as query, it first checks if that key appears in the Bloom filter. If positive, it consult `DiffFile.txt`, otherwise access `database.txt` directly. For most keys nt appearing in `DiffFile.txt`, `BloomDifferential` does not consult `DiffFile.txt` unless a false positive.

To compare performances of `NaiveDifferential` and `BloomDifferential`, we are given the following files, `database.txt`, `DiffFile.txt` and `grams.txt`. In `database` (nearly 12 M records) and `DiffFile.txt` (nearly 1.2 M records), each line contains a 4-word phrase (*key*) followed by a sequence of meaningful numbers (*value*). `grams.txt` contains all keys appearing in `database.txt`. From the files, we generate test files consisting of only keys by different numbers. For example, a test file can have 60 keys randomly picked from `grams.txt` and 60 from `DiffFile.txt`. Then for a test file, we first run `NaiveDifferential` to obtain information on number of queries made on files and time spent on queries. The same information is also obtained by `BloomDifferential`, with additional respective information on filters.

We ran experiments on different generated test files consisting of various number of strings. Due to space limits, we only selected representative sets of results for the experiment result discussion (more results are



(a) 120 keys from `database.txt` 60 `DiffFile.txt`

(b) 6000 keys from `database.txt` 1200 `DiffFile.txt`

Figure 3: More comparison between naive and Bloom filter based method

available in the folder). Fig. 2. is from 5 runs of experiment on the test files consisting of 6 keys from `database.txt` and 12 from `DiffFile.txt`, and records the averages. Fig. 2a. first shows the averaged time `NaiveDifferential` spent for querying the 18 keys on the files. It is clear that the naive method is outperformed by the `BloomDifferential` methods, deterministic (64-bit FNV) and random hash function. The total time is the time spent on the whole process, including building filter, and querying filter and files (the total time and file querying time overlap each other).

As shown in the figure, the majority of time was used on querying files, `DiffFile.txt` and `database.txt`, whereas it took relatively small amount of time to build the filters and very little on querying them. Again, we observe the time complexity of 64-bit FNV is higher than that of the random method. For this particular experiment set, looking into the time on querying files, lots of time was used for querying the database file, which is intuitive because about $\frac{2}{3}$ of the keys were not in the differential file. From Fig. 2b. we can verify that the naive approach did try to retrieve values from `database.txt`, resulting in a less inefficient retrieval procedure. It is noteworthy that although a larger hash table was also used, i.e., `bitsPerElement` $= 8$ and 10, the total process time does not necessarily seem to decrease. Similar results can be obtained from Fig. 3.