EC504 Data Cache Sync Project Report

Tarana Chowdhury and Aparna Rolfe

December 2, 2015

Project Overview

The Data Cache Sync project consisted of two main tasks: synchronizing two data sets on different computers with as little communication as possible, and implementing an autocomplete function to list the most popular completions of a query from the dataset. We created a Java App with a Swing GUI to implement this functionality. As a data file is uploaded the contents are stored into specialized data structures that support the efficient completion of the project tasks.

Data Structures

We implemented two specialized data structures for the project: a Bloom Filter to allow sychronization with minimum exchange of data and a Trie to support fast prefix-completion queries.

Bloom Filter

We chose a Bloom filter to minimize the communication traffic between the two instances of the app trying to synchronize data. Focusing on the requirement for minimum data exchange, we were looking for a data structure that can store a lot of information in a small space, and so a Bloom filter was a natural choice.

A Bloom Filter is a probabilistic data structure that can be used to check elements for membership in a set. It saves space at the cost of adding uncertainty to the result of a membership check. A Bloom Filter must be initialized by adding all the members of a set to it. It can then be queried whether an element is present in the set; if it returns a negative answer the element is definitely not in the set, but if it returns a positive answer there is a small chance that this is a false positive. Implementing the Bloom Filter required deciding what error rate is acceptable in order to utilize less space.

The design variables for a Bloom Filter are:

n The number of strings to be inserted in the Filter

m The number of bits in the Bloom Filter

k The number of hash functions (number of bits set per string inserted); ideally $\lfloor \frac{m}{n} \log 2 \rfloor$

The false positive rate is equal to $0.619^{m/n}$, which is dependent solely on the ratio of m to n.

Complexity Analysis

The Bloom Filter supports two key operations: Adding a string, and searching for a string. The length of the input string is m and the number of input strings is n.

Adding a string : $\Theta(m)$ The input is hashed to generate k hash values and one bit is set to 1 each time. The runtime of each hash operation is dependent on m, so the overall time for hashing is $\Theta(mk)$. Since k is a small constant ($k = 13$ in our implementation), this is equivalent to $\Theta(m)$ runtime.

Searching for a string : $\Theta(m)$ The input is hashed to generate k hash values and one bit is checked each time. The runtime of each hash operation is dependent on m, so the overall time for hashing is $\Theta(mk)$. Since k is a small constant ($k = 13$ in our implementation), this is equivalent to $\Theta(m)$ runtime.

Initializing the data structure : $\Theta(n.m_{avg})$ Initializing the Bloom Filter is done by adding every string in the dataset to it. We have seen that each add operation takes $\Theta(m)$ time. So the runtime for adding n strings is $\Theta(n.m_{avg})$.

Updating the data structure : $\Theta(n_{update}.m_{avg})$ Updating the Bloom Filter is done by adding each new string to it. We have seen that each add operation takes $\Theta(m)$ time. So the runtime for adding n_{update} strings is $\Theta(n_{update}.m_{avg})$.

Space complexity : $\Theta(n)$ By design, our Bloom Filter uses 16 bits per input string, so it needs $\Theta(n)$ space where n is the expected number of strings that will be added.

Implementation Analysis

When a dataset is uploaded to our app, all the queries of the local dataset are added to a Bloom Filter. When a sync request is sent, this Bloom Filter is sent to the remote app instance. The remote app instance checks all its queries for membership in the Bloom Filter, and sends back the queries that are reported to not be present. This means that the false positives will result in some queries that should have been sent not being sent.

Our Bloom Filter implementation assigns 16 bits per expected entry, which sets the theoretical false positive rate at 0.046%. On the test data sets provided, each .txt file was 1.7MB and the corresponding Bloom Filter data exchanged was 0.3MB, which is 18% of the original file size. The two test datasets had approximately 138,000 queries each, and a total of 12 queries that needed to be synced were missed due to false positives, so the observed false positive rate was 0.004%, which is even better than the theoretical 0.046%.

Trie

We chose a Trie to minimize the search time for all the possible completions of a string the user types in the search box.

A Trie is a tree where all the descendants of a node share the same prefix. Each node of the Trie contains one character and a flag indicates whether the node is the end of a word (which is constructed by starting at the root and following the characters down to the given node). Each internal node in the Trie can have as many children as there are symbols in the alphabet, and the height of the Trie is the length of the longest input string.

The performance of a Trie depends on:

d The number of symbols in the alphabet of the input strings

m The number of characters in a given input string

M The sum of the number of characters in all strings added to the Trie

Complexity Analysis

The Trie supports four key operations: Adding a string and its associated popularity to the Trie, returning the popularity of a given string, searching for the four most popular completions of a given prefix, and returning a list of all the strings stored in the Trie. The length of the input string is m and the number of strings added to the data structure is n.

Adding a string : $\Theta(m)$ A new string is inserted by traversing or adding m nodes, so the time taken is a function of the length of the string and the length of the alphabet. Since the length of the alphabet is a small constant (approximately 26), time complexity of inserting a string into the Trie is $O(m)$.

Returning popularity of a given string : $\Theta(m)$ This operation first searches for the string in the Trie, which is a function of the length of the string and the length of the alphabet, and then does a constant-time look-up to report the popularity of the string. Since the length of the alphabet is a known constant (approximately 26), time complexity of inserting a string into the Trie is $O(m)$.

Searching for popular prefix completions : $\Theta(n)$ This operation traverses the subTrie rooted at the end node of the input, storing the four most popular queries to return when it is done. The time to search is proportional to the number of strings that start with the given prefix, which is roughly proportional to the total number of strings in the Trie.

Returning list of all strings : $\Theta(M)$ This operation visits every node in the Trie, adding words to a list as it goes and returning the list when the traversal is complete. Since the number of nodes in the Trie is proportional to the sum of the number of characters in all strings added to the Trie, the time complexity is $\Theta(M)$.

Initializing the data structure : $\Theta(n.m_{avg})$ Initializing the Trie is done by adding every entry in the dataset to it. We have seen that each add operation takes $\Theta(m)$ time. So the runtime for adding n strings is $\Theta(n.m_{avg})$.

Updating the data structure : $\Theta(n_{update}.m_{avg})$ Updating the Trie is done by adding each new entry to it. We have seen that each add operation takes $\Theta(m)$ time. So the runtime for adding n_{update} strings is $\Theta(n_{update}.m_{avg})$.

Space complexity : $\Theta(M)$ Space required depends on the number of nodes, which is $\Theta(M)$.

Implementation Analysis

When a user starts typing a query in the Search box, every keystroke triggers a search function in the Trie which takes the typed string as input and returns the four most popular queries.

The number of characters in each test data set was approximately 1,250,000, which got compressed into approximately 555,000 nodes in the Trie. For the test data set, the average length of a word was 9 and the effective alphabet size was 26 (46 different characters were present in the input strings but only 26 appeared more than 10 times) so the average number of nodes searched was 234; compared to looking through 138,000 entries, the search is approximately 50,000 times faster.

Team Member Contributions

Tarana Major part of Trie and GUI implementation, milestone video editing, final presentation.

Aparna Major part of bloom filter and data transfer implementation, final project report.