Lesson 11:
DATA STRUCTURE TOOLBOX FOR BLOCKCHAINS (AND IPFS)
25/3/2021
HASH POINTERS

• Outline of the lesson: introduce data structures for DHT, IPFS, Blockchains
  • Hash pointers
  • Bloom filters
  • Merkle Trees
  • Trie
  • Patricia Trie
  • Patricia Merkle Trie
HASH POINTERS

- an hash pointer is:
  - a pointer to where some info is stored
  - a cryptographic hash of the info
- if we have a hash pointer, we can
  - ask to get the info back
  - verify that it hasn’t changed

tamper-evident data pointer = Hash Pointer
HASH DATA STRUCTURES

- key idea: build data structures with hash pointers
- block chain: list linked with hash pointers
  - to compute a hash pointer to a block, hash the entire block, also its own hash pointer!
  - use case: tamper-evident log, a basic data structure in Bitcoin
**HASH DATA STRUCTURES**

- if someone tampers the k-th block of the chain, the hash of block k+1 is not going to match up

- this is because the **hash is collision resistant**: an adversary cannot tamper the data so that its hash is the same of the data before the tampering
• **use case**
  - tamper-evident log, a basic data structure in Bitcoin and Ethereum
  - in PoW-based blockchain, the block contains also the proof that PoW has been successfully executed
  - if data is changed, PoW has to be re-executed for all the blocks
  - computationally infeasible
MORE GENERALLY...

- can use hash pointers in any pointer-based data structure that has no cycles, for instance, in a DAG.

- several applications applications:
  - **Advanced Intelligent Corruption Handling**, introduced in eMule
    - used to check that the block of a file which has been downloaded from the network has not been tampered
    - exploits Merkle trees: binary trees with hash pointers
  - **Bitcoin**: uses SHA-256 and RIPEMD-160 (double) cryptographic hash functions
    - the block chain of Bitcoin is list of blocks of transactions chained through hash pointers.
  - **Merkle Dag for IPFs**, and many other applications....
BLOOM FILTERS: SET MEMBERSHIP PROBLEM

- consider the set $S = \{s_1, s_2, \ldots, s_n\}$ of $n$ elements chosen from a very large universe $U$.
- define an efficient data structure supporting queries like “$k$ is an element of $S$?”

- the function $f$ returns value true or false according to the presence of $k$ in the given set.
 APPROXIMATE SET MEMBERSHIP PROBLEM

- S may be
  - a set of keywords describing the files shared by a peer, selected from the universe of all the keywords (Gnutella 0.6)
  - the set of pieces of file owned by a peer (BitTorrent)
  - a set of bitcoin addresses:
    - light weight (mobile) mobile nodes build Bloom filters with the address of the transaction they are interested in, not store the whole blockchain
    - send the Bloom filter to the full nodes: bandwidth saving

- Problem: choose a representation of the elements in S such that:
  - the result of the query is computed efficiently
  - the space for the representation of the elements is reduced
  - the results may be approximated to save space
  - possibility of returning false positives
an approximate solution to the set membership problem:

- trade off between:
  - space required
  - probability of false positives
Building Bloom Filters

- $m$ bits (initially set to 0)
- $k$ hash functions

Diagram: A Bloom filter with $m$ bits, initially set to 0, and $k$ hash functions.
Building Bloom Filters

$m$ bits (initially set to 0)
$k$ hash functions

Add

```
S 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

0 1 2 m 1 m
$m$ bits (initially set to 0) $k$ hash functions

Add

if $f(x) = A$, set $S[A] = 1$
BUILDING BLOOM FILTERS

\[ m \text{ bits (initially set to 0)} \]
\[ k \text{ hash functions} \]

If \( f(x) = A \), set \( S[A] = 1 \)

Add

\[ S = 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \]

\[ x \]

\[ f(x) \]
Building Bloom Filters

$m$ bits (initially set to 0)
$k$ hash functions

if $f(x) = A$, set $S[A] = 1$
BUILDING BLOOM FILTERS

$m$ bits (initially set to 0)

$k$ hash functions

if $f(x) = A$, set $S[A] = 1$

Add

$g(x)$

$f(x)$

$h(x)$

$s$

$0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0$

$m-1$ $m$
BUILDING BLOOM FILTERS

$m$ bits (initially set to 0)
$k$ hash functions

if $f(x) = A$, set $S[A] = 1$
BUILDING BLOOM FILTERS

given

- a set $S = \{s_1, s_2, \ldots, s_n\}$ of $n$ elements
- a vector $B$ of $m$ ($n << m$, generally $nk < m$) bits, $b_i \in \{0, 1\}$
- $k$ hash independent functions $h_1, \ldots, h_k$, for each $h_i : S \subseteq U \rightarrow \{1..m\}$, which return a value uniformly distributed in the range $[1..m]$.
- construct a Bloom Filter $B[1..m]$ such that:
  
  for each $x \in S$, $B[h_j(x)] = 1, \forall j = 1, 2, \ldots k$
- a bit in $B$ may be target for more than 1 element
BLOOM FILTERS: LOOK UP

$m$ bits (initially set to 0)
$k$ hash functions

if $f(x) = A$, set $S[A] = 1$

Add

Query
BLOOM FILTERS: LOOK UP

- **m** bits (initially set to 0)
- **k** hash functions

**Add**

- \( f(y) \)
- \( g(x) \)
- \( f(x) \)
- \( h(y) \)
- \( h(x) \)

**Query**

- \( f(z) \)
- \( h(z) \)
- \( g(z) \)

If \( f(x) = A \), set \( S[A] = 1 \)
BLOOM FILTERS: LOOK UP

- **m** bits (initially set to 0)
- **k** hash functions

If \( f(x) = A \), set \( S[A] = 1 \)

Add

Query

One bit set to 0

\( \Rightarrow z \notin S \)
To verify if \( y \) belongs to the set \( S \) mapped on the Bloom Filter, apply the \( k \) hash functions to \( y \)

- \( y \in S \) if \( B[h_i(y)] = 1, \forall i = 1, .. k \)
- if at least a bit = 0, the element does not belong to the set.

**False positives:**

- if all bits are positive
  - the element might be in the set,
  - another element or some combination of other elements could have set the same bits.
BLOOM FILTERS: TRADE OFF

the price paid for this efficiency is that a Bloom filter is a probabilistic data structure: it tells us that the element either definitely is not in the set or may be in the set.
PROBABILITY OF FALSE POSITIVES

- let us consider a set of $n$ elements mapped on a vector of $m$ bits through $k$ hash functions.

- the hash functions used in a Bloom filter should be independent and uniformly distributed and as fast as possible.
  - for instance $h_i(x) = \text{MD5}(x + i)$ or $\text{MD5}(x || i)$, where $i$ is the index of the hashing function, would work

- basic assumption: hash functions random and independent

- for the analysis, you can apply the balls e bins paradigm: like throwing $k \times n$ balls in $m$ buckets

- goal: evaluate the probability of false positives
PROBABILITY OF FALSE POSITIVES

first step: compute the probability that, after all the $n$ elements are mapped to the vector, a specific bit of the filter (of size $m$) has still value 0?

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$ 

The approximation is derived from the definition of $e$

$$\lim_{x \to \infty} \left(1 - \frac{1}{x}\right)^{-x} = e.$$
PROBABILITY OF FALSE POSITIVES

- a percentage of $e^{-kn/m}$ bits are 0, after its construction.

- consider an element not belonging to the set: apply the $k$ functions
  - a false positive is obtained if all hash functions return a value $= 1$

- probability of false positive $\left(1 - e^{-kn/m}\right)^k$

depends on

- $m/n$: number of bits exploited for each element of the set
- $k$: number of hash functions

- if $m/n$ is fixed, it seems two conflicting factors for defining $k$ do exist....
  - decreasing $k$ increases the number of 0 and hence the probability to have a false positive should decrease, but....
  - increasing $k$ increases the precision of the method. Hence the probability of false positive should decrease....
fixed the ratio $m/n$, the probability of false positives first decreases, then increases, when considering increasing values of $k$

- $m/n=2$, a few bits for each element, “too much hash functions” cannot be exploited because they fill the filter of 1.
- $m/n=10$, a larger number of hash functions decreases the probability of false positives
let us now suppose that $k$ is fixed, the probability of false positives exponentially decreases when $m$ increases ($m$ number of bits in the filter).

for low values of $m/n$ (a few bits for each element), the probability is higher for large values of $k$
A Bloom filter becomes effective when \( m = c \times n \), with \( c \) constant value (low value), for instance \( c = 8 \).

In this case with 5-6 hash functions the probability of false positives is low.

Good performances with a limited number of bits.

<table>
<thead>
<tr>
<th>bits/element</th>
<th>( m/n )</th>
<th>2</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of hash functions</td>
<td>( k )</td>
<td>1.39</td>
<td>5.55</td>
<td>11.1</td>
<td>16.6</td>
</tr>
</tbody>
</table>
| false-positive probability | \( f \)  | 0.393| 0.0216| 4.59 \( \times 10^{-4} \)| 9.84 \( \times 10^{-6} \)
PROBABILITY OF FALSE POSITIVES

- trade-off between space/number of hash functions/probability of false positives
- if n and m are fixed (fix the number of bits for each element)
  - determine which is the value of k minimizing the probability of false positives
  - compute the derivative of the function of the previous slide so obtaining the minimum \((\ln 2 \approx 0.7)\)

\[
k = \frac{m}{n} \ln 2,
\]

to this value, corresponds a value of the probability equal \(0.62^{m/n}\)
with the values computed in the previous slides, the probability that one bit is still equal to 0 after the application of the $k$ functions is

$$p = e^{-kn/m} = \frac{1}{2}$$

the optimal values are obtained when the probability that a bit is equal to 0 after the application of the $k$ functions to the $n$ elements is equal to $\frac{1}{2}$

an “optimal” Bloom Filter is a “random bitstring” where half of the bits chosen uniformly at random, is 0.
PROBABILITY OF FALSE POSITIVES

- probability as a function of the number of elements of the set
  - \( k \) optimum, given \( n \) and \( m \)
- logarithmic scale
- if the number of bits for each element is not sufficient the probability exponentially grows

![False positive rate of Bloom filters](image)

- X-axis: Number of inserted elements (\( n \))
- Y-axis: False positive probability (\( p \))

Legend:
- \( m = 64 \)
- \( m = 512 \)
- \( m = 1024 \)
- \( m = 2048 \)
- \( m = 4096 \)
BLOOM FILTERS: OPERATIONS

• **Union** - Given two Bloom Filters, B1 e B2 representing, respectively, the set S1 and S2 through the same number of bits and the same number of hash functions, the Bloom filter representing S1 \( \cup \) S2 is obtained by computing the bitwise OR bit of B1 and B2.

• **Delete**: note that it is not possible to set to 0 all the elements indexed by the output of the hash functions, because of the conflicts.

  • **Counting Bloom Filters**: each entry of the Bloom Filter is a counter, instead of a single bit.
    • exploited to implement the removal of elements from the Bloom filter
    • at insertion time, increment the counter
    • at deletion time, decrement the counter
BLOOM FILTERS: OPERATIONS

• intersection: given two Bloom Filters $B_1$ and $B_2$ representing respectively, the sets $S_1$ and $S_2$ through the same number of bits and the same number of hash functions.

• the intersection of the Bloom filters is obtained by computing the bitwise and of $B_1$ and $B_2$ and approximates $S_1 \cap S_2$

• as a matter of fact, if a bit is set to 1 in both Bloom filters, this may happen because:
  • this bit corresponds to an element $\in S_1 \cap S_2$, therefore it is set to 1 in both filters: in this case no approximation
  • this bit corresponds to an element $\in S_1 - (S_1 \cap S_2)$ and to an element $\in S_2 - (S_1 \cap S_2)$ hence it does not correspond to any element in the intersection
• Google BigTable and Apache Cassandra use Bloom filters to avoid costly disk lookups considerably and to increases the performance of a database query.

• the Google Chrome web browser uses a Bloom filter to identify malicious URLs
  • any URL is first checked against a local Bloom filter and only upon a hit a full check of the URL is performed.

• Bitcoin uses Bloom filters to verify payments without running a full network node.

• Gnutella 0.6 exploits a simplified version of Bloom Filters

but many other ones currently exist....
BLOOM FILTERS: APPLICATIONS

Guard against expensive operations (like disk access)
First line of defence in high performance (distributed) caches
Peer to Peer communication
Routing - Resource Location

Squid Proxy Cache
Google BigTable
Cassandra
HBase
Various RDBMS'
Google Chrome
Cisco Routers

WHAT IS A MERKLE TREE?

- Hash trees or Merkle trees
  - a data structure summarizing information about a big quantity of data
  - with the goal to verifying the correctness of the content
- introduced by Ralph Merkle in 1979
**MERKLE TREE**

- why to use a Merkle tree?
  - to prove that a data is not tampered with

- characteristics
  - simple
  - efficient
  - versatile

- a **complete binary tree** of hashes built starting from an initial set of blocks
  - exploits a hash function H
  - leaves: H is applied to the initial blocks
  - internal nodes: H is applied to the concatenation of the hashes of the sons of a node
MERKLE TREE: A TREE OF HASH VALUES

- data \((m)\) is not considered as a part of the Merkle tree
- hash of the data is part of the Merkle tree
- \(+\) means appending two values (concatenation)
A NAIVE SOLUTION FOR TAMPER FREENESS

- Alice
  - wants to prove to Bob that a data is not tampered with
  - sends the data block $m_6$, together the hash of all other blocks

- Bob
  - gets the root hash of all the data from a trusted source
  - hashes $m_6$, append all hashes to a single string and hash this string to obtain the root hash
  - compares this root hash with the trusted source root
  - if they are equal, $m_6$ is not tampered with
• as before, Bob gets a root hash from a trusted source
• but now Alice needs to send \( m_6 \) and only 4 blue hash values to Bob
  • a more efficient solution
  • 4 hashes for 16 data blocks
A Merkle proof consists of:

- a Merkle proof consists of
- a chunk of data
- the root hash of the tree: the red one
- the "branch" consisting of all of the hashes going up along the path from the data chunk to the root.
Bob

- receives the Merkle proof and a data block
- computes the root hash by composing the hash of the data block with the hashes in the Merkle proof, recursively
- compares the root it has obtained with the root taken from the trusted source
• the more leaves, the less hashed values are needed, in comparison to the number of leaves
  • 6 hashes over 32 leaves
  • validation of tamper ferreness is more efficient
MERKLE TREE: SOME PROPERTIES

- total number of leaves \( L = \frac{N+1}{2} \)
- total number of nodes \( N = 2L - 1 \)
- total number of levels \( LV = \log_2(L) + 1 \)
• remember just the root of the tree in a secure place
• if an adversary tampers some blocks at the bottom of the tree, the hash pointers one level up do not match, and the so on...
TAMPER EVIDENT BINARY TREES: MERKLE TREES

- remember just the root of the tree in a secure place
- if an adversary tampers some blocks at the bottom of the tree, the hash pointers one level up do not match, and the so on...
MERKLE TREE: SUMMARY

- build Merkle tree and store root in a 3rd trusted party

- $\log(n)$ hashes are sufficient for checking each data block
  - “data integrity over untrusted storage with small communication cost”

- Pros:
  - scales logarithmically in the number of objects
  - fine-grained data integrity
  - each write does work proportional to the size of the fragment

- Cons:
  - static: smallest segment size = smallest unit of verification
Merkle trees and peer to peer applications:
  - can help ensuring that data blocks received from other peers are received undamaged and unaltered
  - a trusted third party, for instance the site indexing the .torrent stores
    - the root hash
    - the total size of the file and the piece size
  - a peer:
    - receives a piece and a Merkle proof for it
    - calculates the hash of that piece.
    - request the root hash from the trusted site
    - using this information the client recalculates the root hash of the tree, and compares it to the root hash it received from the trusted source.
MERKLE TREES: APPLICATIONS

- BitTorrent
  - uses Merkle trees to ensure that the files you download from peers haven’t been tampered.
  - was similar for eMule

- IPFS
  - linked authenticated data structures

- Cryptos (Bitcoin, Ethereum)

- Distributed Version Control (Git/Mercurial)

- Copy-On-Write Filesystems (btrfs/ZFS)

- Distributed NoSQL Databases (Cassandra, Riak, Dynamo)
MERKLE TREES: APPLICATION

Peer to Peer communication

gnutella
DC++
Lime Wire

Amazon Dynamo
Google BigTable
Cassandra
HBase
ZFS
Google Wave
• a trie (pronounced “try”), name is from “retrieval”
• a tree representing a collection of strings with one node per common prefix
• a natural way to represent a map where keys are strings

• the smallest tree such that
• each edge is labeled with a character $c \in \Sigma$
• a node has at most one outgoing edge labeled $c$, for $c \in \Sigma$
• each key is “spelled out” along some path starting at the root
map on the left is represented with the trie on the right

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>instant</td>
<td>1</td>
</tr>
<tr>
<td>internal</td>
<td>2</td>
</tr>
<tr>
<td>internet</td>
<td>3</td>
</tr>
</tbody>
</table>
TRIES: IMPLEMENTATION

- let $\Sigma$ be some fixed alphabet.
- each node $x$ of the node implementing the trie corresponds to some string given by the path traced from the root to that node.
- each node of the trie stores
  - a flag (green/red flag in figure) indicating whether the string spelled out to this point is in the set
  - an array of $|\Sigma|$ pointers, one for each character.

three words: cane, car and tear.
TRIE: COMPLEXITIES

• what is the cost of looking up a string w in a trie?
  • follow at most $|w|$ pointers to get to the node for w, if it exists.
  • at each step, look up (LU) of a pointer, depends on the data structure implementing the pointers: $O(1)$, at minimum
  • total time: $O(|w|)$.
  • lookup cost is independent of the number of strings in the trie!
• different costs according to the way the array of pointers associated to characters, are stored
  • an array of child pointers of size $\Sigma$: waist of space, but LU $O(1)$
  • a hash table of child pointers: less waist of space, LU $O(1)$
  • a list of child pointers: compact, LU is $O(\Sigma)$ in the worst-case
  • a binary search tree of child pointers: compact and LU is $O(lg \Sigma)$ in the worst-case
TRIE: INSERTION AND REMOVAL

- insertion of a new string
  - proceed before as if doing a normal lookup, adding in new nodes as needed.
  - set the flag in the final node visited this way

- removal of a string
  - mark the node as no longer containing a word.
  - if the node has no children:
    - remove that node.
    - repeat this process at the node one level higher up in the tree.
COMPLEXITY ANALYSIS: SPACE

- although time-efficient, tries can be extremely space-inefficient.

- a trie with \( N \) nodes and \(|\Sigma|\) alphabet will need space \( \Theta(N \cdot |\Sigma|) \) due to the pointers in each node.

- there are different ways of addressing this:
  - change the data structure for holding the pointers
    - use lists, time/space trade-off
  - eliminate unnecessary trie nodes: Patricia tries
PATRICIA TRIES

- PATRICIA: Practical Algorithm To Retrieve Information Coded In Alphanumeric
  - based on the following observation:
    - having chains of one-child nodes is wasteful
    - make a compressed version of the trie
  - replace a chain of one-child nodes, not including terminal nodes, with an edge labeled with a string of more than one character

But if we were somehow able to associate entire paths with a string, compressing links...

Note: All non-key-containing nodes with just one child node have now vanished, and the paths traversing them have been compressed to a single link!
- the Ethereum Blockchain,
  - store the state of the contracts on the blockchain
- the state is combination of key value pair. For instance
  - key: address of an account
  - value: the account balance
- Merkle Patricia tries used by Ethereum to represent the state
- a combination of Merkle tree and Patricia trie
  - Merkle tree maintains data integrity
  - Patricia tree enable faster search of data
MERKLE PATRICIA TRIES

- build
  - a Patricia trie with keys
  - a Merkle tree of the values stored in the leaves of the Patricia trie
- only the Merkle tree root is meaningful and stored in the blockchain