

Distributed Systems Advanced & Blockchain

Fully Distributed Systems: Algorithms

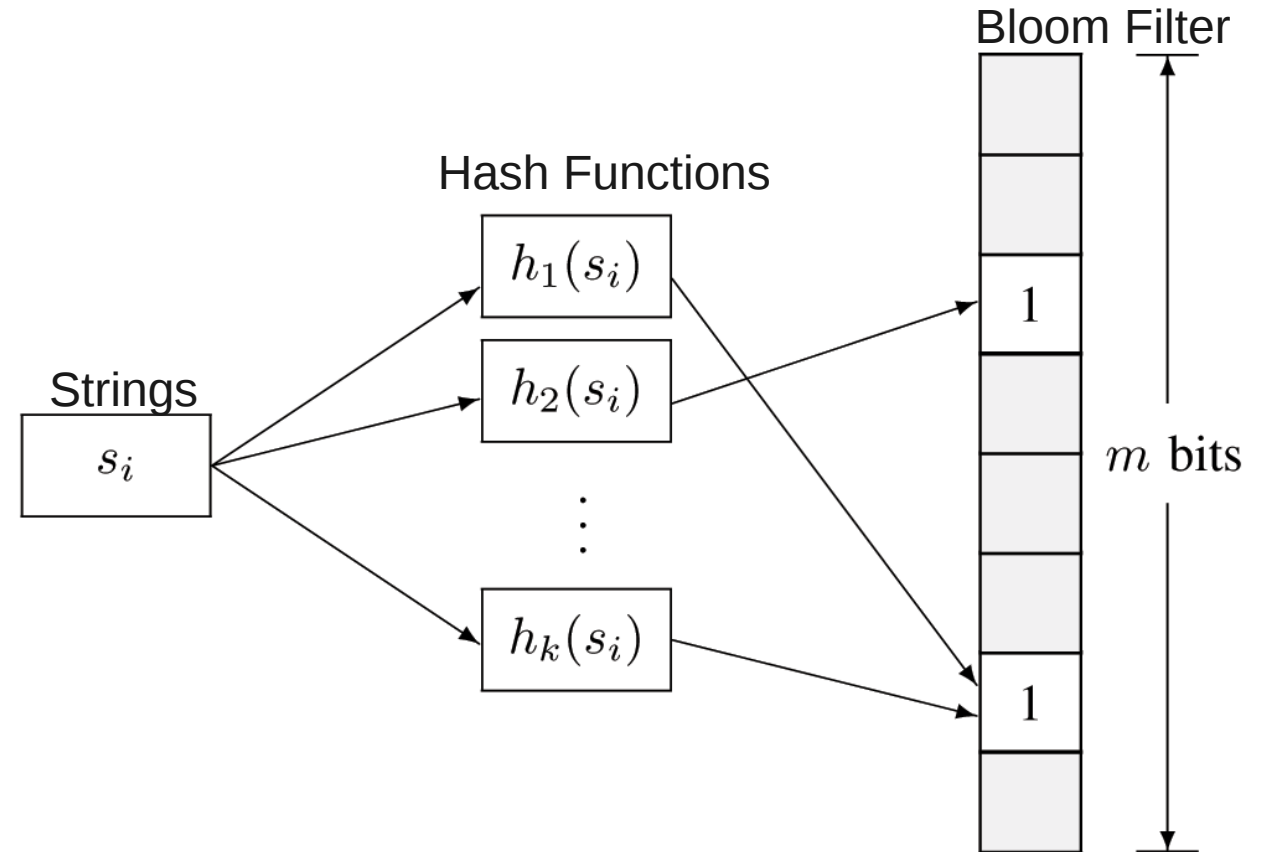
Thomas Bocek

1 November 2020

Algorithms for distributed Systems: Bloom Filter

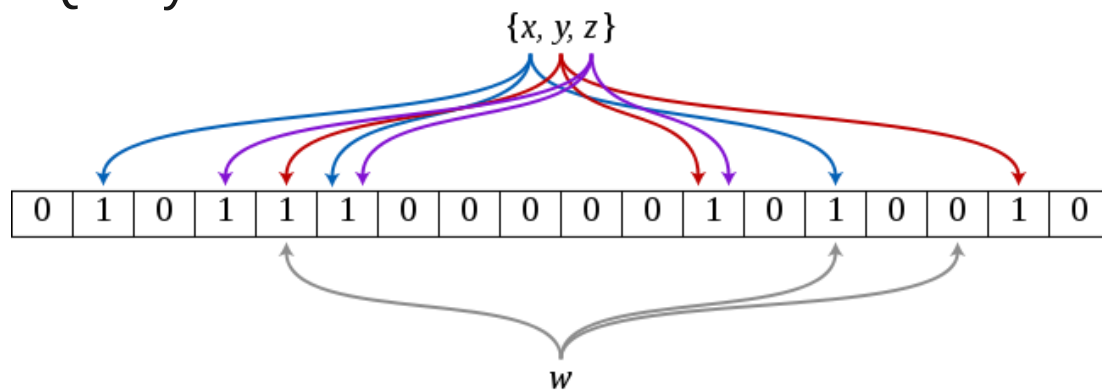
Traditional Bloom Filter

- An array of m bits, initially all bits set to 0
- A bloom filter uses k independent hash functions
 - $h_1, h_2, \dots, h_k$ with range $\{1, \dots, m\}$
- Each input is hashed with every hash function
 - Set the corresponding bits in the vector
- Operations
 - Insertion
 - The bit $A[h_i(x)]$ for $1 \leq i \leq k$ are set to 1
 - Query
 - Yes if all of the bits $A[h_i(x)]$ are 1, no otherwise
 - Deletion
 - Removing an element from this simple Bloom filter is impossible



Query of an Element, $m=18$, $k=3$

- Insert x, y, z
- Query w



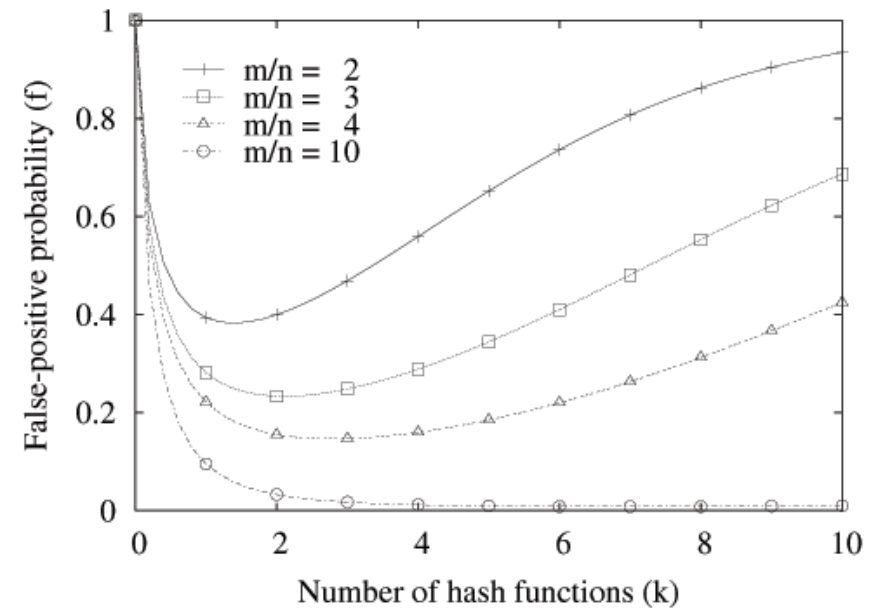
http://en.wikipedia.org/wiki/Bloom_filter

- Example for False-positives
 - Insertions
 - Hash („color printer“) $\Rightarrow (1,4,6)$
 - Hash („digital camera“) $\Rightarrow (3,4,5)$
 - Bloom filter (1,3,4,5,6)
 - Query
 - Hash („heat sensor“) $\Rightarrow (3,4,6)$
 - Matches since bits 3,4,6 are all set to 1
 - Online
- False-negative
 - Query
 - Hash („color printer“) $\Rightarrow (1,4,6)$, matches (1,3,4,5,6)
 $\rightarrow$ no false-negative

Properties

- Space Efficiency
 - Any Bloom filter can represent the entire universe of elements
 - In this case, all bits are 1
- No Space Constraints
 - Add never fails
 - But false positive rate increases steadily as elements are added
- Simple Operations
 - Union of Bloom filters: bitwise OR
 - Intersection of Bloom filters: bitwise AND
- No false negative, but false positive
- False-positive probability:
 - n number of strings: k hash functions; m -bit vector

$$f = \left(1 - e^{-\frac{nk}{m}}\right)^k$$



=> Given m/n , there is an optimal number of hash functions (opt. $k = m/n \ln 2$) (when 50% of the bits are set)

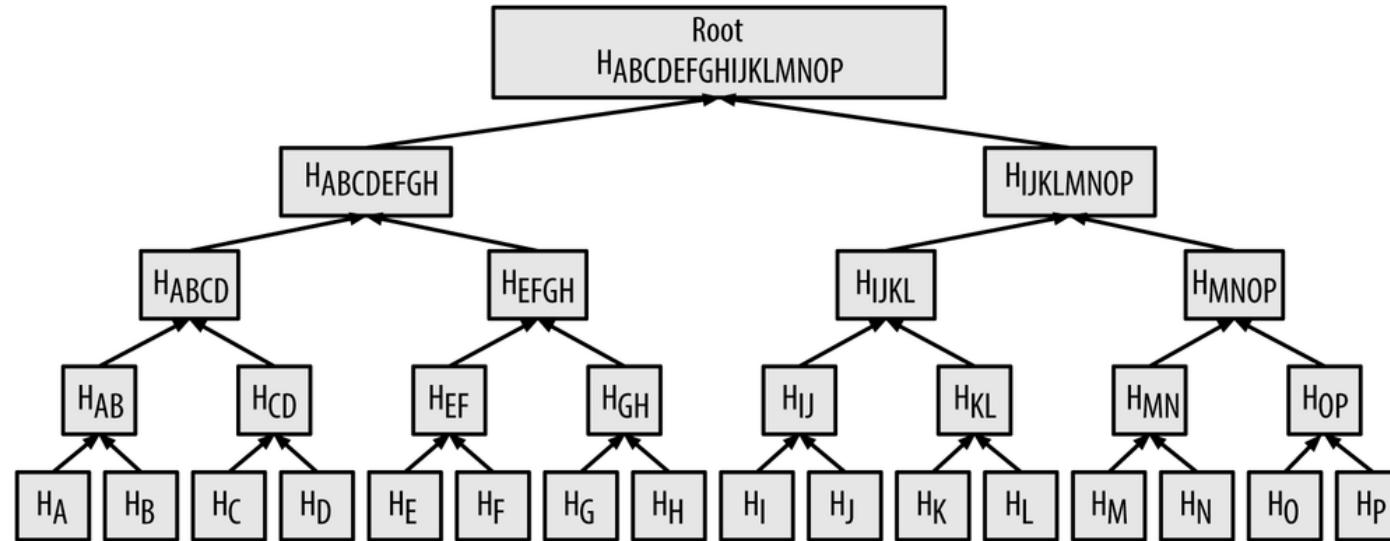
Bloom Filter Variants

- Compressed Bloom Filters
 - When the filter is intended to be passed as a message
 - False-positive rate is optimized for the compressed bloom filter (uncompressed bit vector m will be larger but sparser)
 - However, compression/decompression, more memory
- Generalized Bloom Filter
 - Two type of hash functions g_i (reset bits to 0) and h_j (set bits to 1)
 - Start with an arbitrary vector (bits can be either 0 or 1)
 - In case of collisions between g_i and h_j , bit is reset to 0
 - Store more info with low false positive
 - Produces either false positives or false negatives
- Counting Bloom Filters
 - Entry in the filter not be a single bit but a counter
 - Delete operation possible (decrementing counter)
 - Variable-Increment Counting Bloom Filter
- Scalable Bloom Filter
 - Adapt dynamically to number of elements, consist of regular Bloom filters
 - “A SBF is made up of a series of one or more (plain) Bloom Filters; when filters get full due to the limit on the fill ratio, a new one is added; querying is made by testing for the presence in each filter”
- Others, e.g., Cuckoo filter
- Usage: e.g., fast search

Algorithms for distributed Systems:

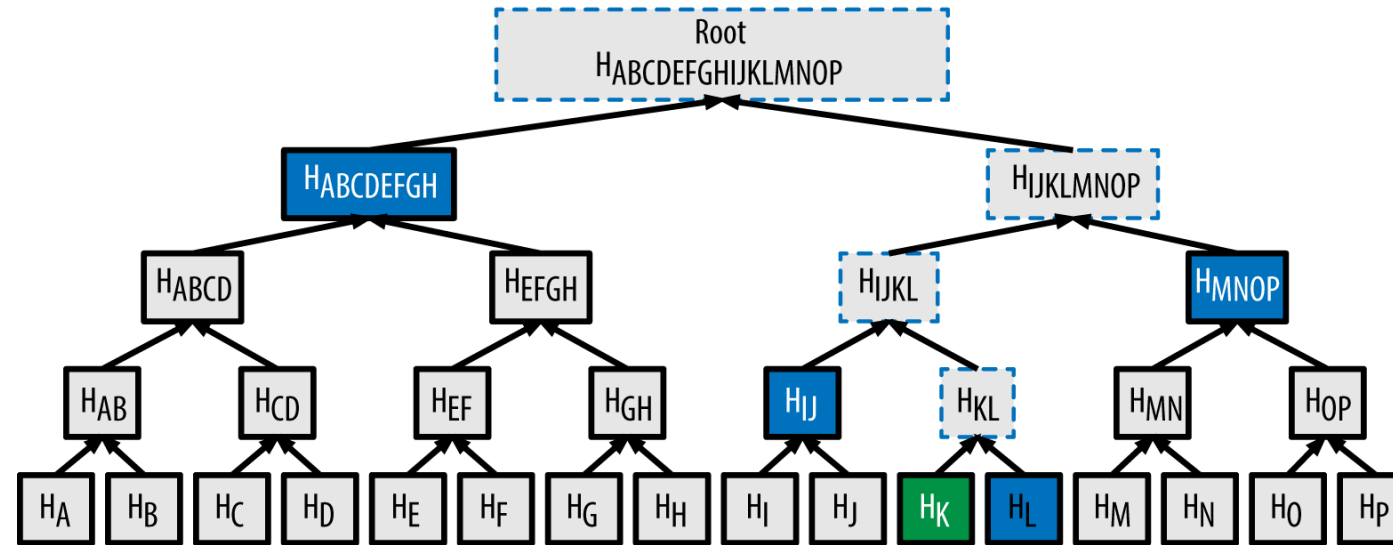
Merkle Tree

Merkle Trees



- A Merkle tree is a *binary hash tree* containing N leaf nodes
- Constructed bottom-up, i.e., $H_{AB} = \text{Hash}(HA + HB)$
- Used to summarize all transactions in a block
- To prove that a specific transaction is included in a block, a node only needs to produce $\log_2(N)$ hashes, constituting a merkle path connecting the specific transaction to the root of the tree.

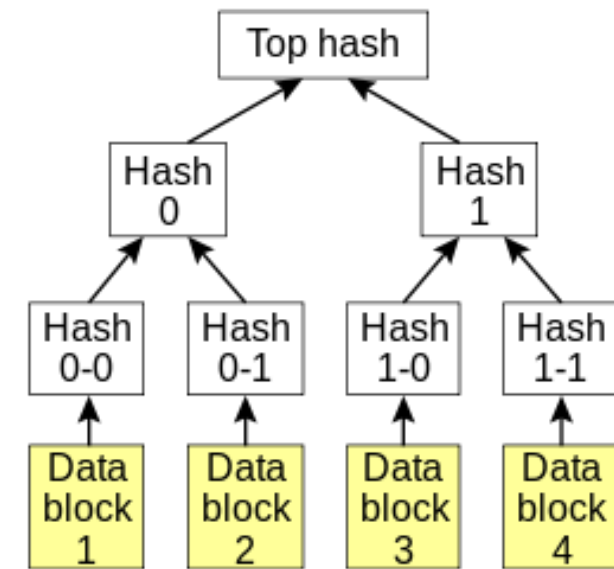
Merkle Proofs



- A node can prove that transaction K is included in the block by producing a merkle path
 - $\log_2(16) = 4$ *hashes* long

BitTorrent: Mechanisms

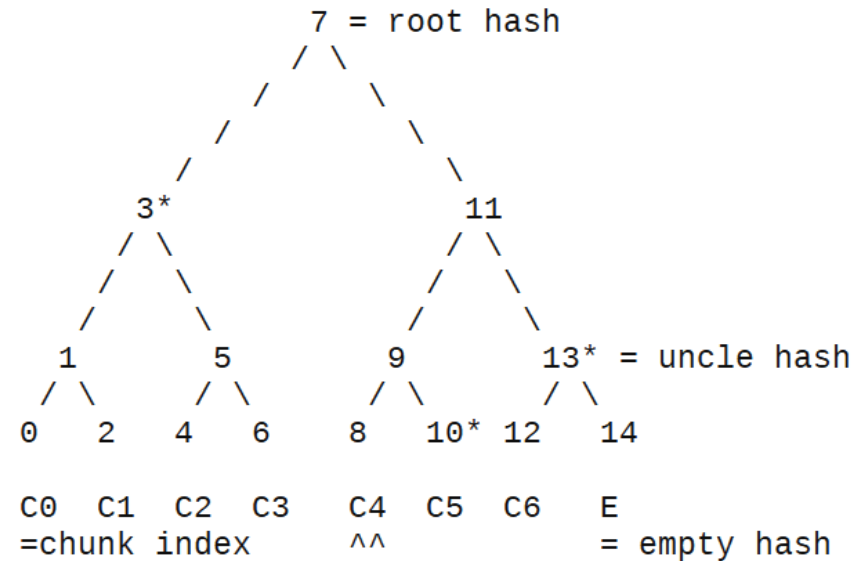
- Magnet links
 - Magnet is URI scheme, does not point to a centralized tracker
 - No centralized tracker: pointer to DHT
 - General purpose, not only for BT
 - `magnet:?xl=1000&dn=song1.mp3&xt=urn:tree:tiger:2A3B...`
- `tree:tiger` → Hash Tree
 - Tree of hashes (`||` → concatenation)
 - `hash 0 = hash( hash 0-0 || hash 0-1 )`
 - `hash 1 = hash( hash 1-0 || hash 1-1 )`
 - `Top hash = hash( hash 0 || hash 1 )`
- Merkle hash / hash tree also seen in Bitcoin blocks (transactions)



http://en.wikipedia.org/wiki/Hash_tree

BitTorrent: Mechanisms

- Verification
 - Peer A has top hash (root hash)
 - Peer downloads C4 from peer B
 - create hash 8
 - Need hash 10, 13, 3 (uncle hash)
 - Can be from peer B
 - With 8,10,13,3 can create root hash
 - → verify this root hash
- Usage: Blockchain, P2P filesharing, git, Amazons Dynamo, ZFS



The Merkle hash tree of an interval of width $w=8$

<http://datatracker.ietf.org/doc/draft-ietf-ppsp-peer-protocol/> Section 5.2

Algorithms on top of distributed Systems: And / or searching Range queries

Mechanisms based on Hashing in KV storage

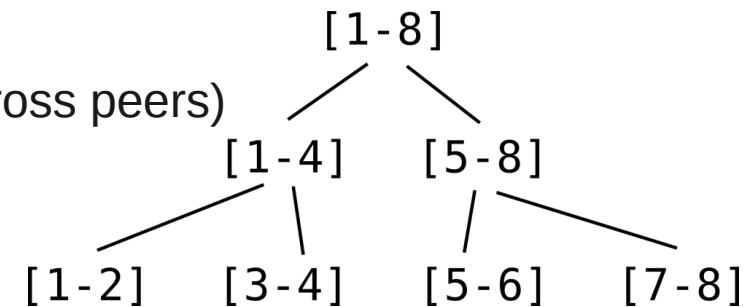
- Search in DHT
 - `DHT.get(h(«Institut für Software»))`
 - In order to find it: `DHT.put(h(«Institut für Software»), value)`
- Keywords
 - `DHT.get(h(«Institut»))`
 - Find it: `DHT.put(h(«Institut»), value), DHT.put(h(«für»), value), DHT.put(h(«Software»), value)`
 - value points to `h(«Institut für Software»)`
- Keywords drawbacks
 - Find good keywords → “the”, “a” are not good keywords
 - Exact matches only

Mechanisms based on Hashing in KV storage

- Find “Institut” or “Software” - OR Systems
 - `DHT.get(h(«Institut»))` and `DHT.get(h(«Software»))`, combine results
- Find “Institut” and “Software” - AND Systems
 1. `DHT.get(h(«Institut»))` and `DHT.get(h(«Software»))`, intersect results
 2. `DHT.get(h(«Institut») xor h(«Software»))`
 - In order to find it:
`DHT.put(h(«Institut») xor h(«Software»), value),`
`DHT.put(h(«Institut») xor h(«für»), value)`
`DHT.put(h(«für») xor h(«Software»), value)`
 - Combination needs to be known in advance
 3. Use Bloom Filters
 - `bf = DHT.getBF(h(«Institut»))` and `DHT.get(h(«Software»), bf)`
 - Sequential (less network, slower) vs. parallel (more network, faster)

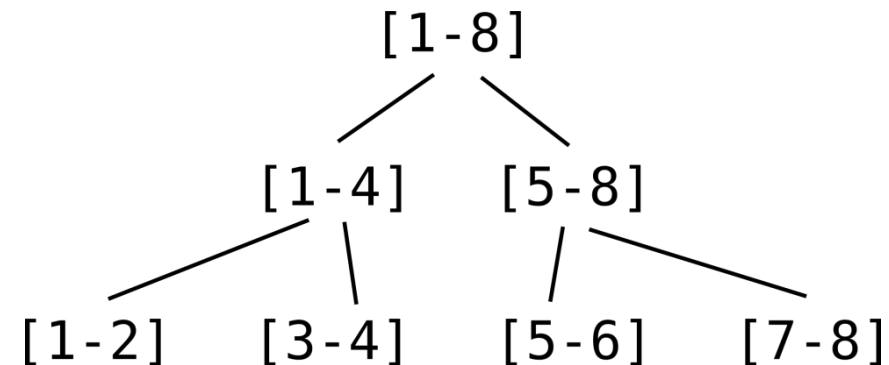
Mechanisms based on Hashing in KV storage

- Range Queries
 - Problem: random insert vs. sequence insert
 - Sequence $\rightarrow [0..n-1] [n..2n-1] [2n..3n-1] [\dots] \rightarrow$ peer responsible for range, hash it, store it, done.
 - Insert 10 items: $N = 5 \rightarrow [0, 1, 2, 3, 4], [5, 6, 7, 8, 9]$ – sequential, 2 peers
 - Insert 10 items: $N = 5 \rightarrow [0], [5], [10], [15], [20], [25], [30], [35], [40], [45]$ – random, 10 peers
 - But random: worst case: 1 peers has 1 data item, range query for range $[0..x]$ contacts x/n peers.
- Over-DHT
 - **PHT**: trie (prefix tree); **DST**: segment $\rightarrow$ tree on top of DHT
 - Main idea: hash of tree-node (resp. for range) $\rightarrow$ DHT
 - PHT: Peer stores n data items, if n reached, splits data (moves data across peers)
 - DST: stores data on each level (redundancy) up to a threshold
 - No data splitting



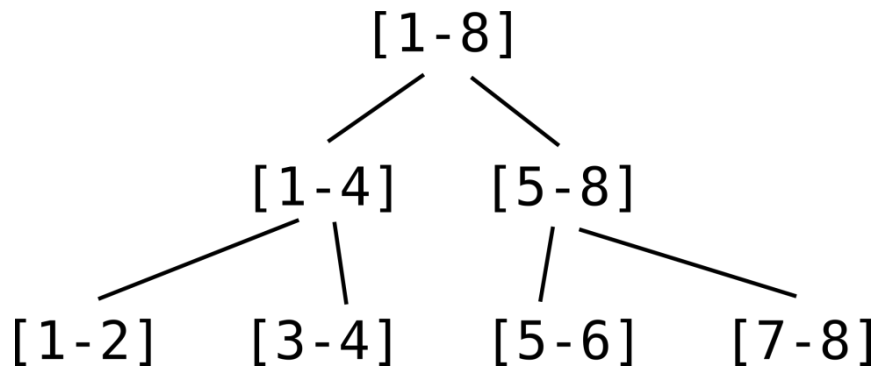
Mechanisms based on Hashing in KV storage

- Example:
 - Set $n = 2, m = 8$
 - 1, "test"; 2, "hallo"; 3, "world"; 5, "sys"; 6, "ost"; 7, "ifs"
- Tree: store value
 - Translate `putDST(1, "test")` to
 - `put(hash([1-8]), "test")`
→ may be stored (only if threshold not reached)
 - `put(hash([1-4]), "test")` → may be stored
 - `put(hash([1-2]), "test")` → will be stored
 - Store `put(2, "hallo"), put(3, "world"), put(5, "sys"), ...`
- Query `getDST(1..5)` translates to
 - `get(hash[1-8])` → returns "1,test; 2,hallo"
 - `get(hash[1-4])` → returns "1,test; 2,hallo"
 - `get(hash[1-2])` → returns "1,test; 2,hallo"
 - `get(hash[3-4])` → returns "3,world"
 - `get(hash[5-8])` → returns "5,sys; 6,ost"
 - `get(hash[5-6])` → returns "5,sys; 6,ost"



Mechanisms based on Hashing in KV storage

- Example:
 - Set $n = 2, m=8$
 - $1, \text{"test"}; 7, \text{"ifs"}$
- Tree: store value
 - Translate `putDST(1, "test")` to
 - `put(hash([1-8]),"test")`
→ may be stored (only if threshold not reached)
 - `put(hash([1-4]),"test")` → may be stored
 - `put(hash([1-2]),"test")` → will be stored
 - Store `put(7, "ifs")`
 - Query `getDST(1..5)` translates to
 - `get(hash[1-8])` → returns "1,test; 7,ifs"
 - `get(hash[1-4])` → returns "1,test;"
 - `get(hash[5-8])` → returns "7,ifs"
 - Range query as series of `put()` and `get()`



Algorithms on top of distributed Systems: Similarity Search

Mechanisms based on Hashing in KV storage

- Similarity Search in DHT
 - <https://fastss.csg.uzh.ch>
- Project that brings similarity search to HT / DHT
 - Problem: Search for “netwrk” fails for DHTs
- Similarity: Edit distance / Levenshtein distance
 - Min operations to transform one string into another, operations: insert, delete, replace
 - Calculated in matrix size $O(m \times n)$

FastSS

$$\begin{aligned}d[i, 0] &= i, \quad d[0, j] = j, \\d[i, j] &= \min (d[i-1, j] + 1, \quad d[i, j-1] + 1, \\&\quad d[i-1, j-1] + (if \ s1[i] = s2[j] \ then \ 0 \ else \ 1))\end{aligned}$$

Mechanisms based on Hashing in KV storage

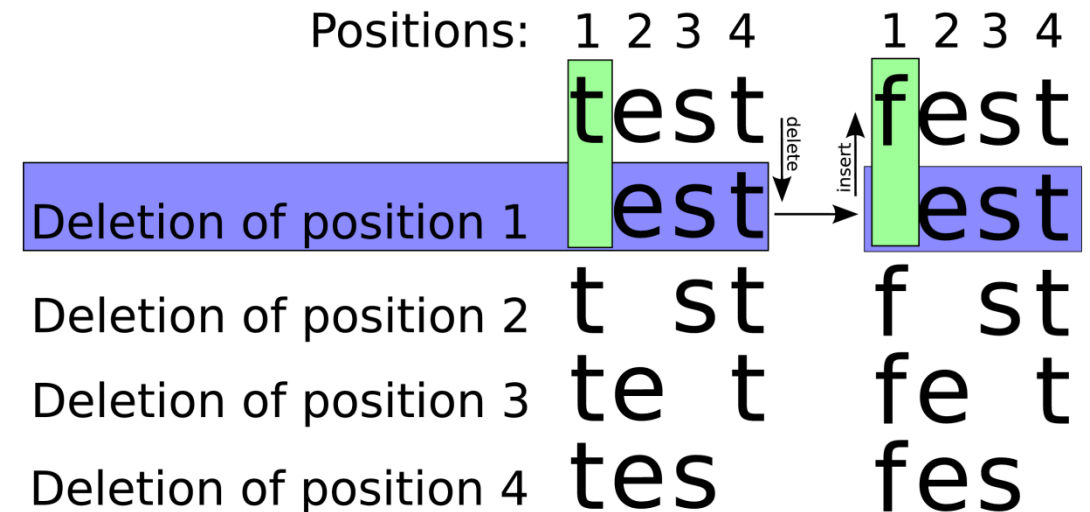
- Example $d(\text{test}, \text{east}) = 2$ (remove a, insert t)
- Expensive operation if all words need testing
- Main idea: pre-calculate errors
 - All possible errors? Neighbors for test with ed 2: test, test^a, test^{aa}, test^{ab}, ... , te^a, te^b, te^c, ..., te^{aa}, te^{ab}, ... → 23883 more of those!

		T	E	S	T
	0	1	2	3	4
E	1	1	1	2	3
A	2	2	2	2	3
S	3	3	3	2	3
T	4	3	4	3	2

$$\begin{aligned}d[i, 0] &= i, \quad d[0, j] = j, \\d[i, j] &= \min(d[i-1, j] + 1, d[i, j-1] + 1, \\&\quad d[i-1, j-1] + (if \ s1[i] = s2[j] \ then \ 0 \ else \ 1))\end{aligned}$$

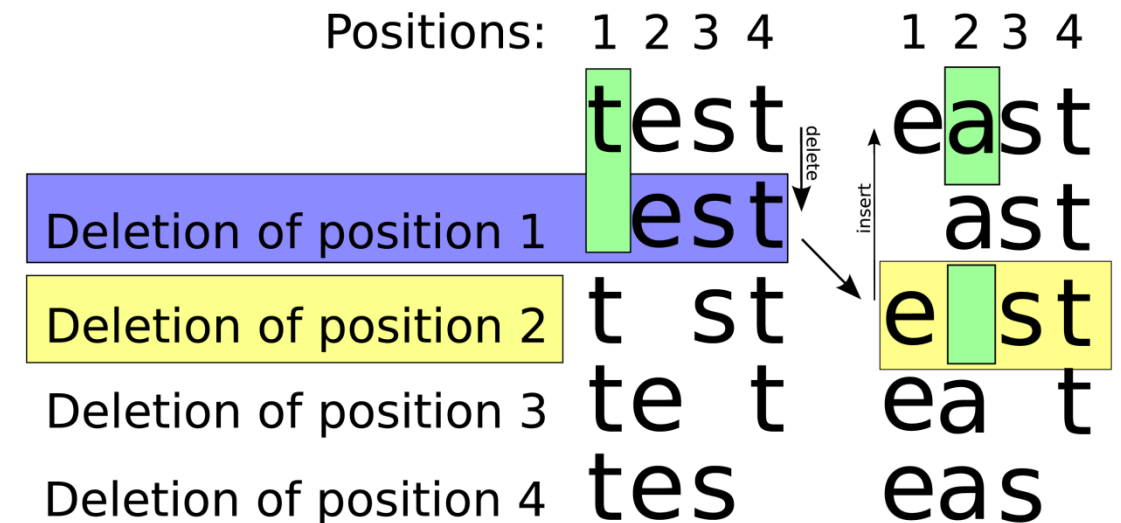
Mechanisms based on Hashing in KV storage

- FastSS pre-calculates with deletions only
 - Neighbors for *test* with ed 2: test, est, st, et, es, tst, tt, ts, tet, te, tes
 - Pre-calculation on query **and** index
 - 11 neighbors → 11 more queries, indexed enlarged by 11 entries
- Example $d(\text{test}, \text{fest})=1$
 - test: indexed
 - fest: query



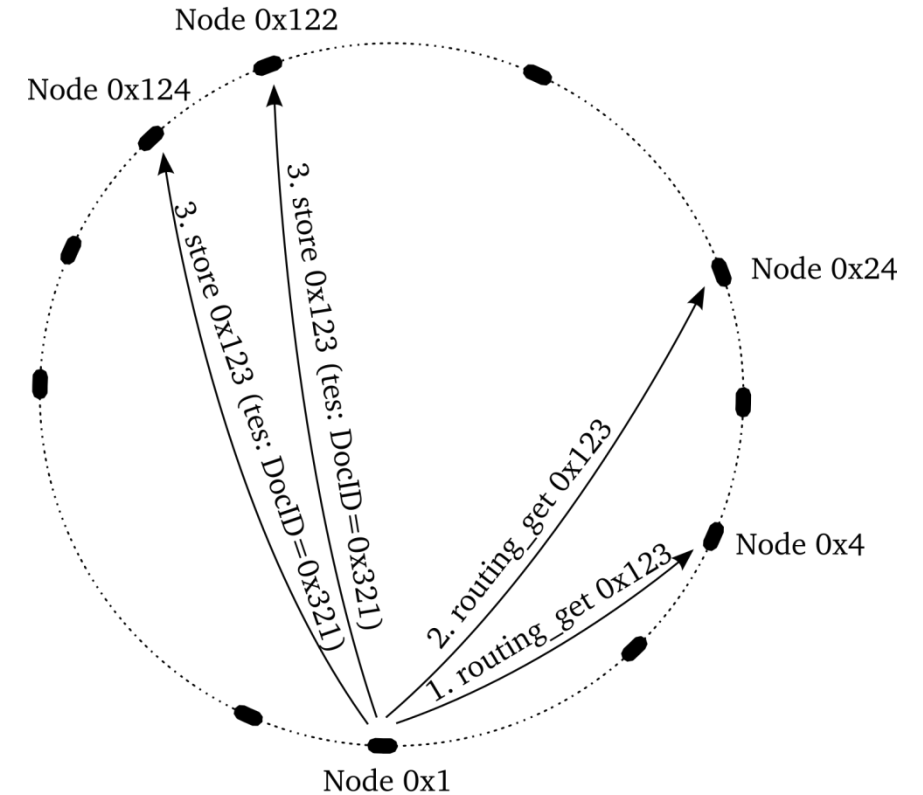
Mechanisms based on Hashing in KV storage

- Example $d(\text{test}, \text{east})=2$
 - test: indexed
 - east: query
- P2PFastSS implemented on top of TomP2P (early version) – tests with indexing Wikipedia abstracts



Mechanisms based on Hashing in KV storage

- Index documents using `put(hash(document), document)`
 - Document (0x321) contains word test
- Index all neighbors (test, tes, tst, tet, est) using `put(hash(neighbor), point to document)`
 - `hash("tes") = 0x123`



Mechanisms based on Hashing in KV storage

- User searches for “tesx”
- Neighbors are generated (tesx, esx, tsx, tex, **tes**)
 - $\text{get}(\text{hash}(\text{neighbor})) \rightarrow 0x123$
 - Find pointer to document (0x321)
 - $\text{document} = \text{get}(0x321)$
- Tests with edit distance 1, partially 2, ignoring delete pos.
 - Overhead (n choose k) for query and index
- Similarity search as series of `put()` and `get()`

