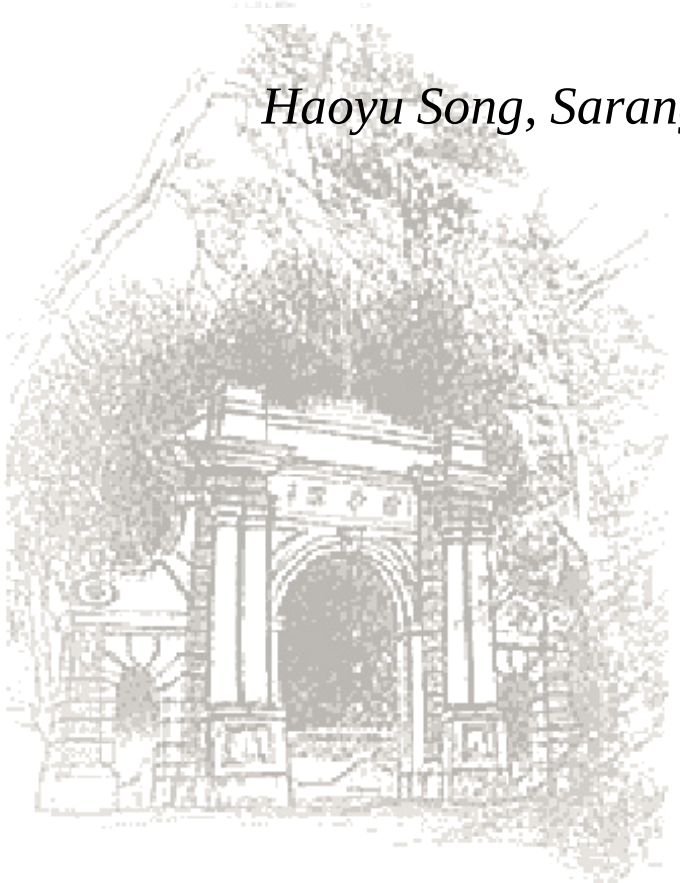
Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing

Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, John Lockwood
Washington University in Saint Louis



Presenter: Kyle WANG

Nslab Seminar, Jun. 5th, 2013

Outline

- Introduction
  - Hash Tables for Packet Processing
  - Related Work
  - Scope for Improvement
- Data Structures and Algorithm
  - Basic Fast Hash Table (BFHT)
  - Pruned Fast Hash Table (PFHT)
  - List-balancing Optimization
  - Shared-node Fast Hash Table (SFHT)
- Analysis
  - Expected Linked List Length
  - Effect of the Number of Hash Functions
  - Average Access Time
  - Memory Usage
- Conclusion

Introduction: Hash Tables

- A hash table is one of the most attractive choices for quick lookups which requires O(1) average memory accesses per lookup.

- Hash tables are prevalent in network processing applications
  - Per-flow state management
    - E.g. Direct Hash (DH), Packet Handoff (PH), Last Flow Bundle (LFB)
  - IP lookup
    - E.g. Balanced Routing Table (BART)
  - Packet classification
    - E.g. Hashing Round-down Prefixes (HaRP)
  - Pattern matching
    - E.g. BART-based Finite State Machine (B-FSM)
  - …

Introduction: Related Work

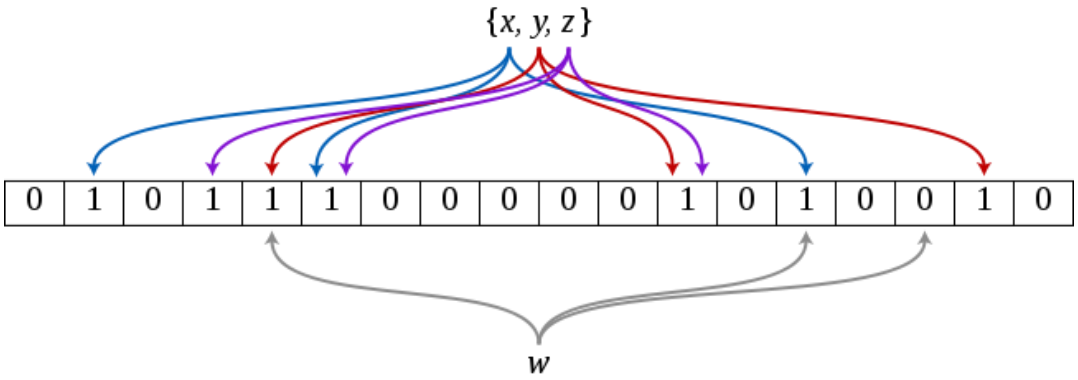
- A hash table lookup involves hash computation followed by memory accesses
  - Using sophisticated cryptographic hash functions such as MD5 or SHA-1
    - Reducing memory accesses raised by collisions moderately
    - Difficult to compute quickly
  - Devising a perfect hash function based on the items to be hashed
    - Searching for a suitable hash function can be a slow process and needs to be repeated whenever the set of items undergoes changes
    - When a new hash function is computed, all the existing entries in the table need to be re-hashed for correct search
  - Multiple hash functions
    - *d* hash functions and *d* hash tables, all hash functions can be computed in parallel
    - *d* hash functions but only one hash table, the item is stored into the least loaded bucket
    - Partitioning buckets into *d* sections, the item is inserted into the least loaded bucket (left-most in case of a tie)

Introduction: Scope for Improvement

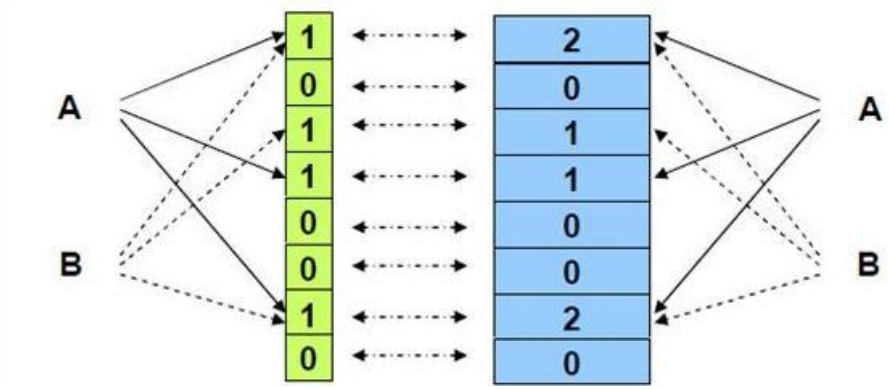
- The proposed hash table
  - Avoids looking up the items in all the buckets pointed to by the multiple hash functions, and always lookups the item in just one bucket.
  - Uses the multiple hash functions to lookup a on-chip counting Bloom filter (due to the small size) instead of multiple buckets in the off-chip memory.
  - Is capable to exploit the high lookup capacity offered by modern multi-port on-chip memory to design an efficient hash table.

Introduction: Bloom Filter

- Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set.



- Counting Bloom filter provides a way to implement a delete operation on a Bloom filter without recreating the filter afresh.



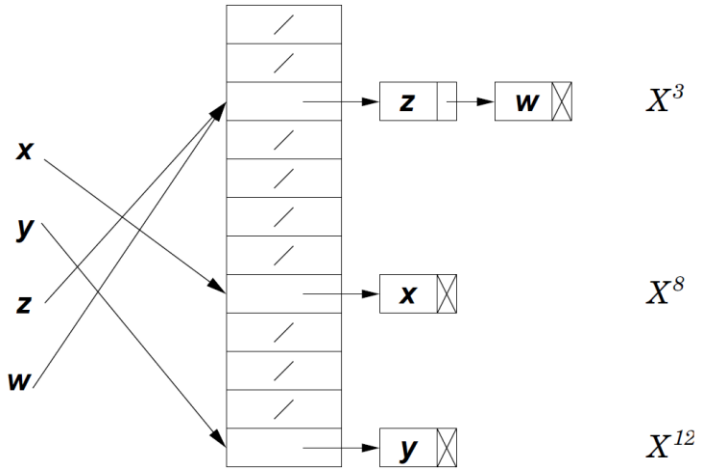
Introduction: A Naïve Hash Table (NHT)

- An NHT consists of an array of m buckets with each bucket pointing to the list of items hashed into it. We denote by X the set of items to be inserted in the table. Further, let X^i be the list of items hashed to bucket i, and X^i_j the jth item in this list. Thus,

$$X^i = \{X^i_1, X^i_2, X^i_3, ..., X^i_{a_i}\}$$

$$X = \bigcup_{i=1}^{L} X^i$$

- Where a_i is the total number of items in the bucket i, and L is the total number of lists present in the table. E.g., X^3_1=z, X^3_2=w, a_3=2, L=3.

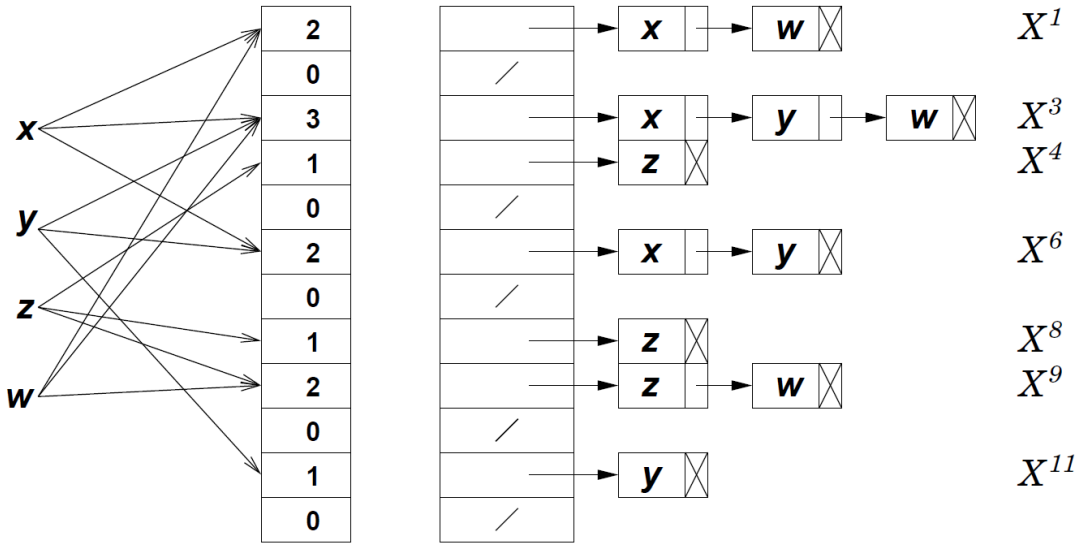


InsertItem$_{NHT}(x)$
1. $X^{h(x)} = X^{h(x)} \cup x$

SearchItem$_{NHT}(x)$
1. if $(x \in X^{h(x)})$ return true
2. else return false

DeleteItem$_{NHT}(x)$
1. $X^{h(x)} = X^{h(x)} - x$

Algorithm: Basic Fast Hash Table (BFHT)



- We maintain an array C of m counters where each counter C_i is associated with bucket i of the hash table. We compute k hash functions $h_1()$, ..., $h_k()$ over an input item and increment the corresponding k counters indexed by these hash values. Then, we store the item in the lists associated with each of the k buckets.

InsertItem$_{BFHT}(x)$

1. for (i = 1 to k)
2. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
3. $C_{h_i(x)}++$
4. $X^{h_i(x)} = X^{h_i(x)} \cup x$

SearchItem$_{BFHT}(x)$

1. $C_{min} = min\{C_{h_1(x)}, ..., C_{h_k(x)}\}$
2. if ($C_{min} == 0$)
3. return false
4. else
5. i = SmallestIndexOf(C_{min})
6. if ($x \in X^i$) return true
7. else return false

DeleteItem$_{BFHT}(x)$

1. for (i = 1 to k)
2. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
3. $C_{h_i(x)}--$
4. $X^{h_i(x)} = X^{h_i(x)} - x$

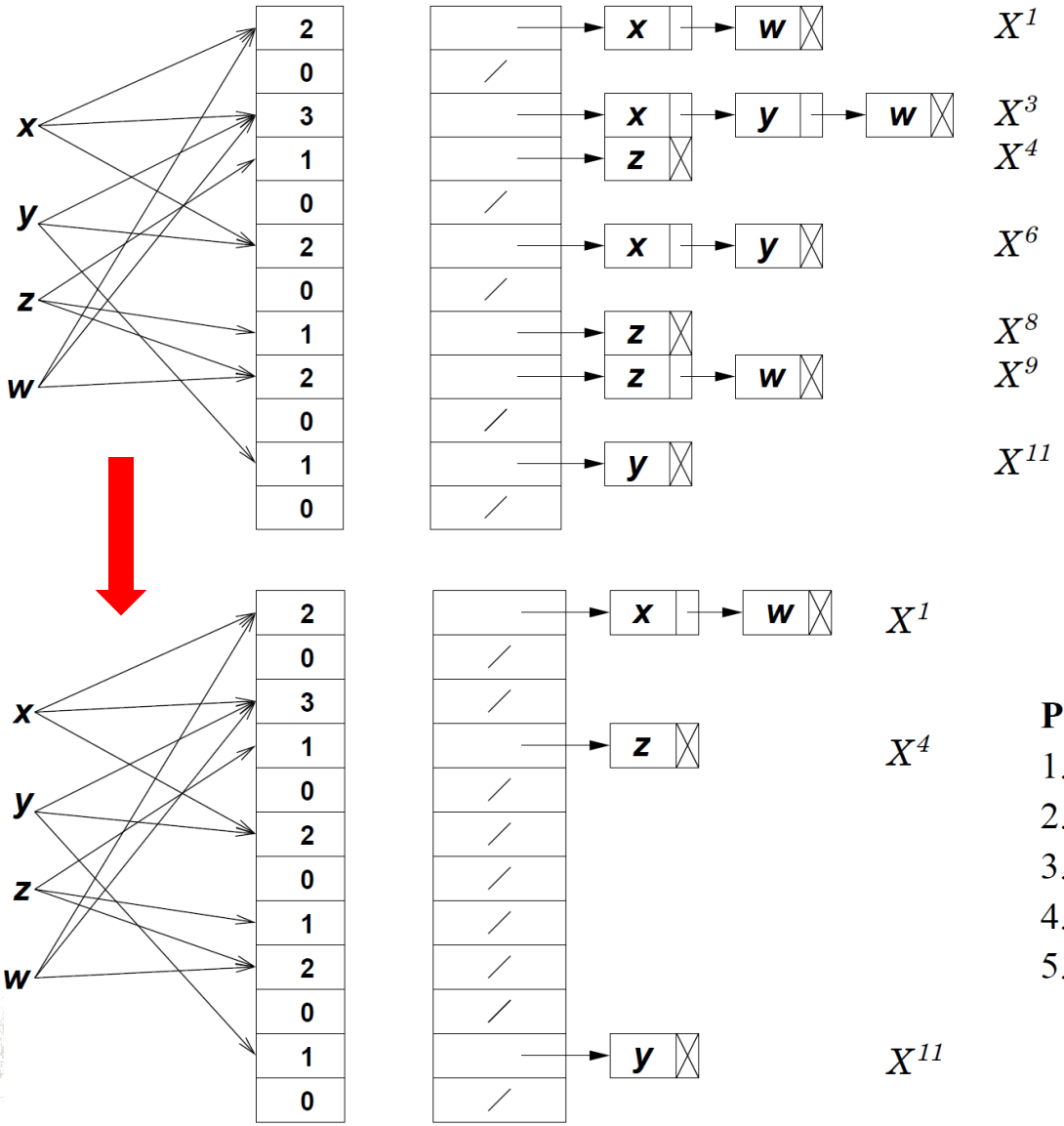
Algorithm: Basic Fast Hash Table (BFHT)

- The speedup of BFHT comes from the fact that it can choose the smallest list to search where as an NHT does not have any choice but to trace only one list which can potentially have several items in it.

- We need to maintain up to *k* copies of each item in BFHT which requires *k* times more memory compared to NHT. However, it can be observed that in a BFHT only one copy of each item — the copy associated with the minimum counter value — is accessed when the table is probed. The remaining (*k*-1) copies of the item are never accessed.

- So ...

Algorithm: Pruned Fast Hash Table (PFHT)

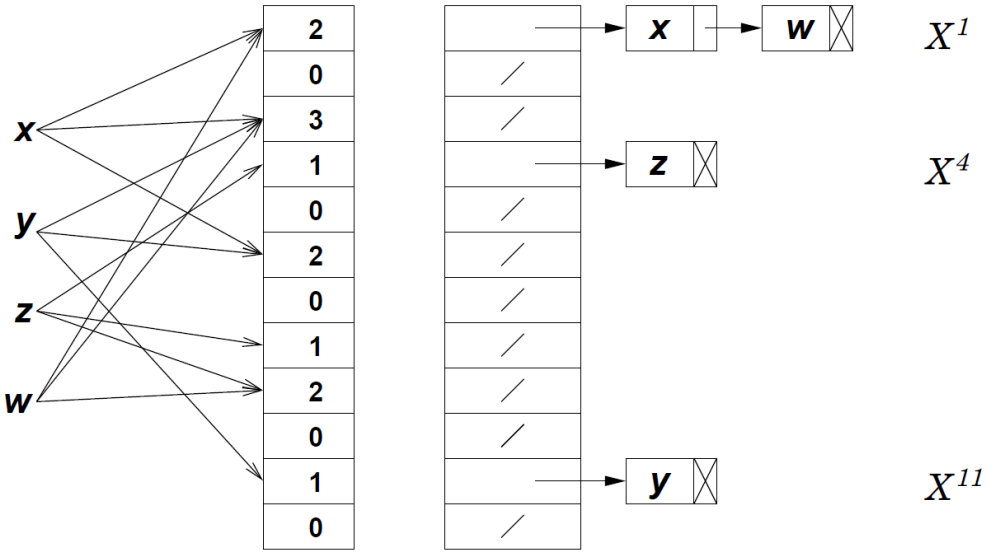


- It is important to note that during the Pruning procedure, the counter values are not changed.
- A limitation of the pruning procedure is that now the incremental updates to the table are hard to perform.

PruneSet(X)

1. for (each $x \in X$)
2. $C_{min} = min\{C_{h_1(x)}, ..., C_{h_k(x)}\}$
3. i = SmallestIndexOf(C_{min})
4. for ($l = 1$ to k)
5. if ($h_l(x) \neq i$) $X^{h_l(x)} = X^{h_l(x)} - x$

Algorithm: Pruned Fast Hash Table (PFHT)



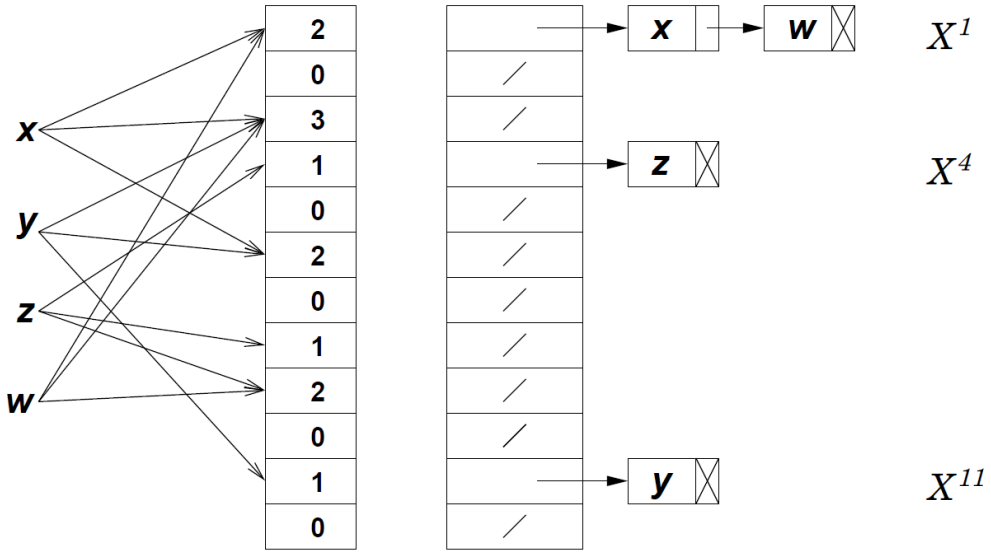
- The basic idea used for insertion is to maintain the invariant that out of the k buckets indexed by an item, it should always be placed in a bucket with smallest counter value.
- Hence, we need to re-insert all and only the items in those *k* buckets.

- *n* items in *m* buckets, average number of items per bucket is *n*/*m*, and total number of items read from buckets is *nk*/*m*, thus 1+*nk*/*m* items are inserted in the table.
- The insertion complexity is O(1+2*nk*/*m*). For an optimal Bloom filter configuration, *k*=*m*ln2/*n*. Hence, the overall memory accesses required for insertion are 1+2ln2=2.44.

InsertItem$_{PFHT}(x)$

1. $Y = x$
2. for (i = 1 to k)
3. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
4. $Y = Y \bigcup X^{h_i(x)}$
5. $X^{h_i(x)} = \phi$
6. $C_{h_i(x)}++$
7. for (each $y \in Y$)
8. $C_{min} = min\{C_{h_1(y)}, ..., C_{h_k(y)}\}$
9. i= SmallestIndexOf(C_{min})
10. $X^i = X^i \cup y$

Algorithm: Pruned Fast Hash Table (PFHT)



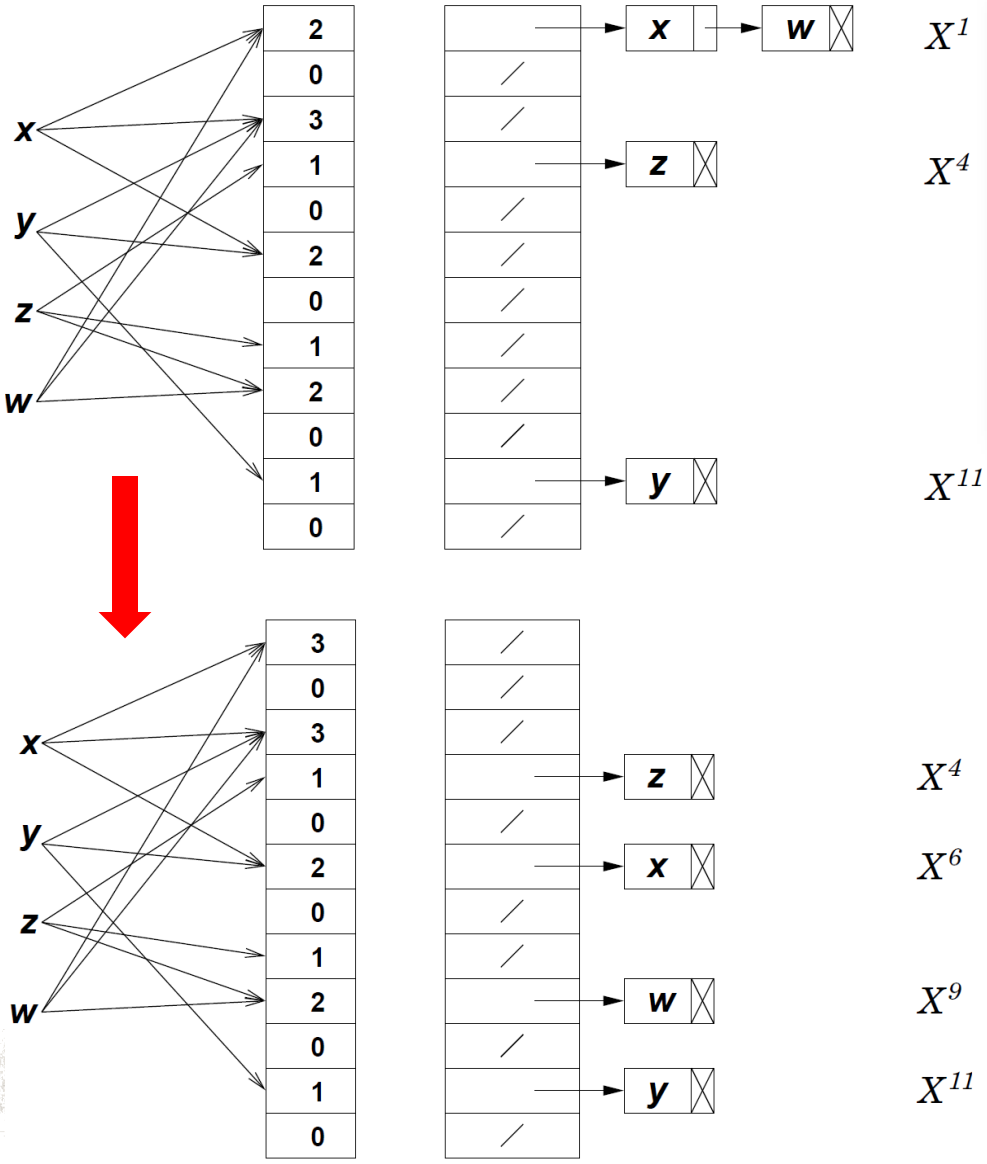
- We denote the off-line lists by χ and the corresponding counter by ζ. Thus, χ^i denotes the list of items associated with bucket i, χ^i_j the jth item in χ^i and ζ_i the corresponding counter.
- Number of items per non-empty bucket in BFHT is $2nk/m$ (optimal Bloom filter), for k buckets, $2nk^2/m$ items are re-adjusted.
- The deletion complexity is $O(4nk^2/m)$ (read + write). For optimal Bloom filter $k=m\ln 2/n$, it boils down to $4k\ln 2=2.8k$.

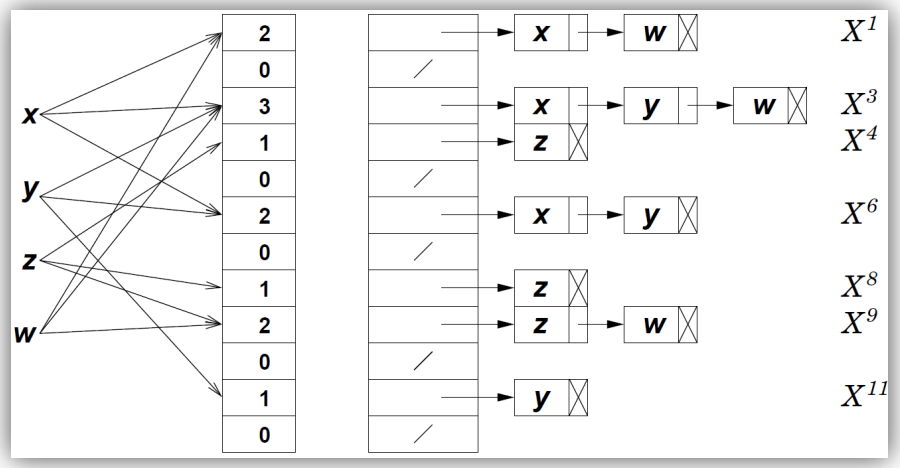
- Due to just one copy of each item, we can not tell which items hash to a given bucket if the item is not in that bucket.
- Hence, we must maintain an off-line pre-pruning BFHT for deletion.

DeleteItem$_{PFHT}(x)$

1. $Y = \phi$
2. for (i = 1 to k)
3. if ($h_i(x) \neq h_j(x)\ \forall j < i$)
4. $\zeta_{h_i(x)}$ − −
5. $\chi^{h_i(x)} = \chi^{h_i(x)} - x$
6. $Y = Y \bigcup \chi^{h_i(x)}$
7. $C_{h_i(x)}$ − −
8. $X^{h_i(x)} = \phi$
9. for (each $y \in Y$)
10. $C_{min} = min\{C_{h_1(y)}, ..., C_{h_k(y)}\}$
11. i= SmallestIndexOf(C_{min})
12. $X^i = X^i \cup y$

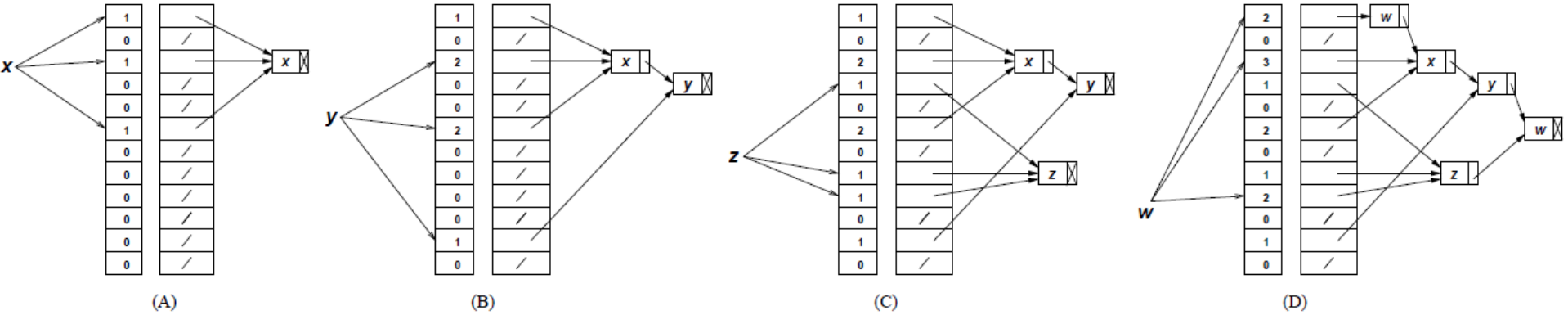
Algorithm: List-balancing Optimization





- The reason that a bucket contains more than one items is because this bucket is the first least loaded bucket indicated by the counter values for the involved items that are also stored in this bucket.
- If we artificially increment this counter, all the involved items will be forced to reconsider their destination buckets to maintain the correctness of the algorithm.
- We perform this scheme only if this action does not result in any other collision.

Algorithm: Shared-node Fast Hash Table (SFHT)



InsertItem$_{SFHT}(x)$

```
1.  for (i = 1 to k)
2.    if (h_i(x) ≠ h_j(x) ∀j < i)
3.      if (C_{h_i(x)} == 0)
4.        Append(x, X^{h_i(x)})
5.      else
6.        l ← 0
7.        while (l ≠ C_{h_i(x)})
8.          l++
9.          read X_l^{h_i(x)}
10.         if (X_{l+1}^{h_i(x)} ≠ NULL) Prepend(x, X^{h_i(x)})
11.         else Append(x, X^{h_i(x)})
12.       C_{h_i(x)}++
```

DeleteItem$_{SFHT}(x)$

```
1.  for (i = 1 to k)
2.    if (h_i(x) ≠ h_j(x) ∀j < i)
3.      l ← 1
4.      while (l ≠ C_{h_i(x)} AND X_l^{h_i(x)} ≠ NULL)
5.        if (X_l^{h_i(x)} == x)
6.          X^{h_i(x)} = X^{h_i(x)} − x
7.          break
8.        l++
9.      C_{h_i(x)} − −
```

For easy incremental updates

Analysis: Expected Linked List Length

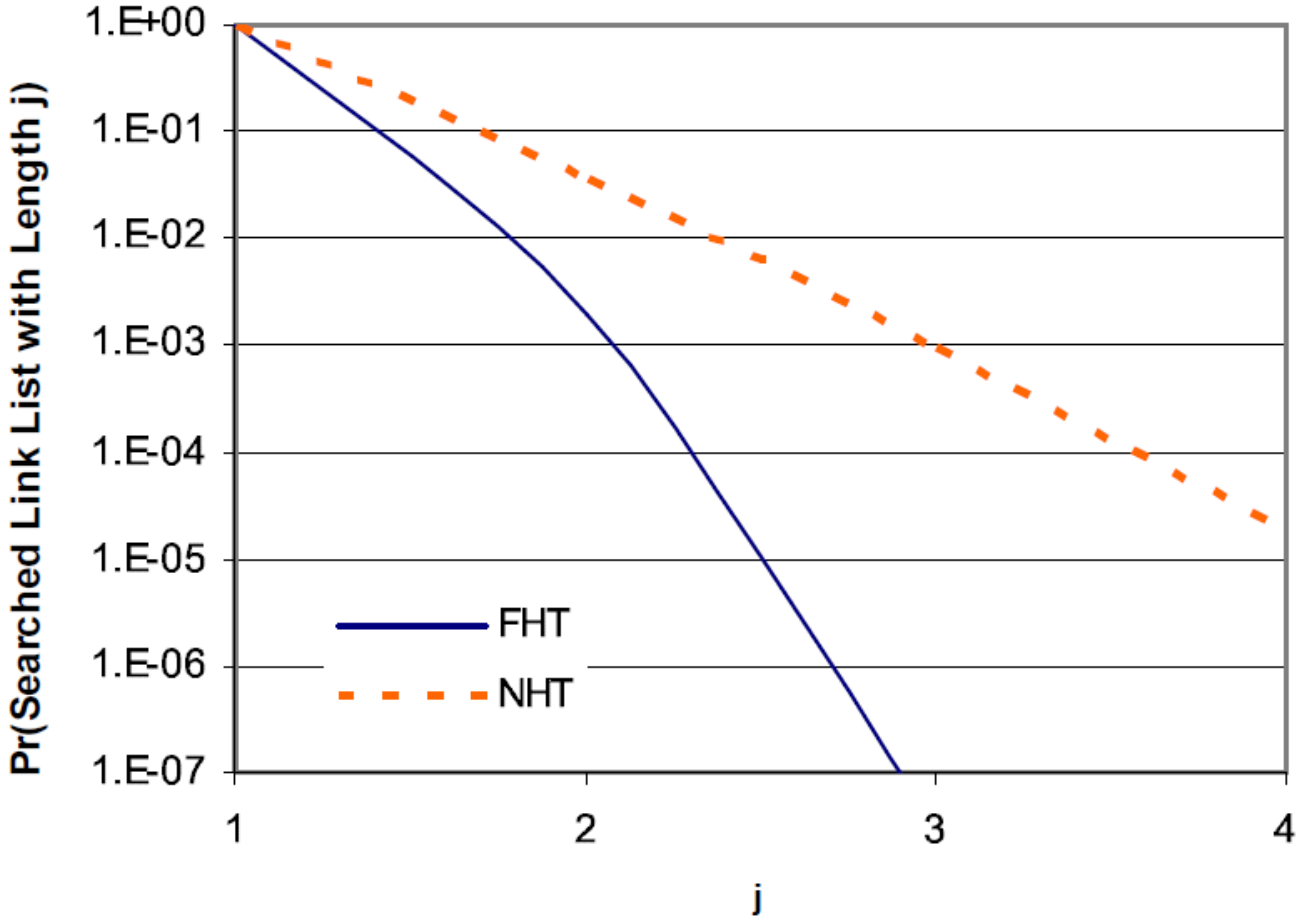


Figure 5: Probability distribution of searched linked list length: $n = 10,000$, $m = 128K$. $k = 10$ **for FHT.**

Analysis: Expected Linked List Length

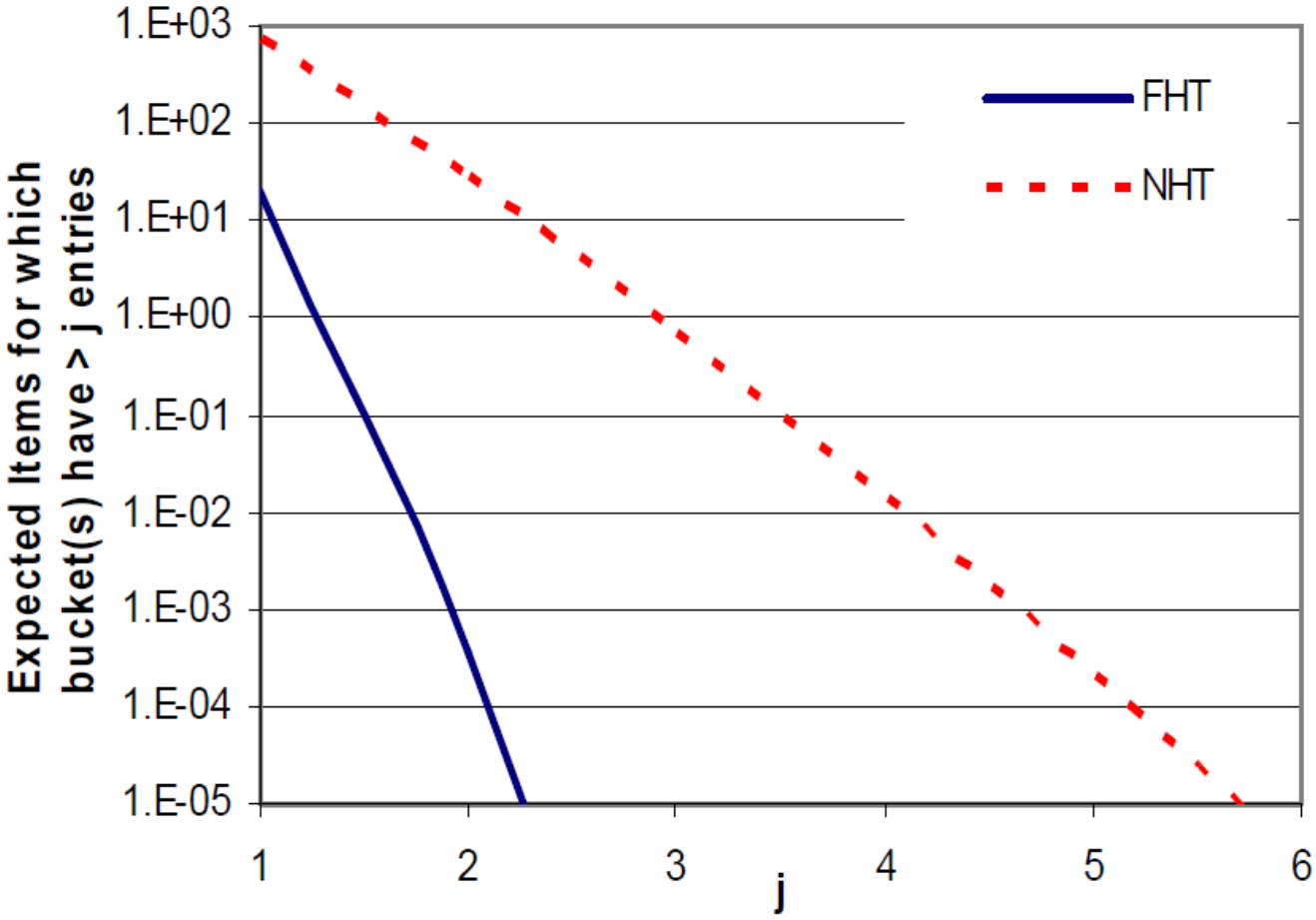


Figure 6: Expected number of items for which the searched bucket contains $> j$ items. $n = 10,000$, $m = 128K$. $k = 10$ for FHT.

Analysis: Effect of the Number of Hash Functions

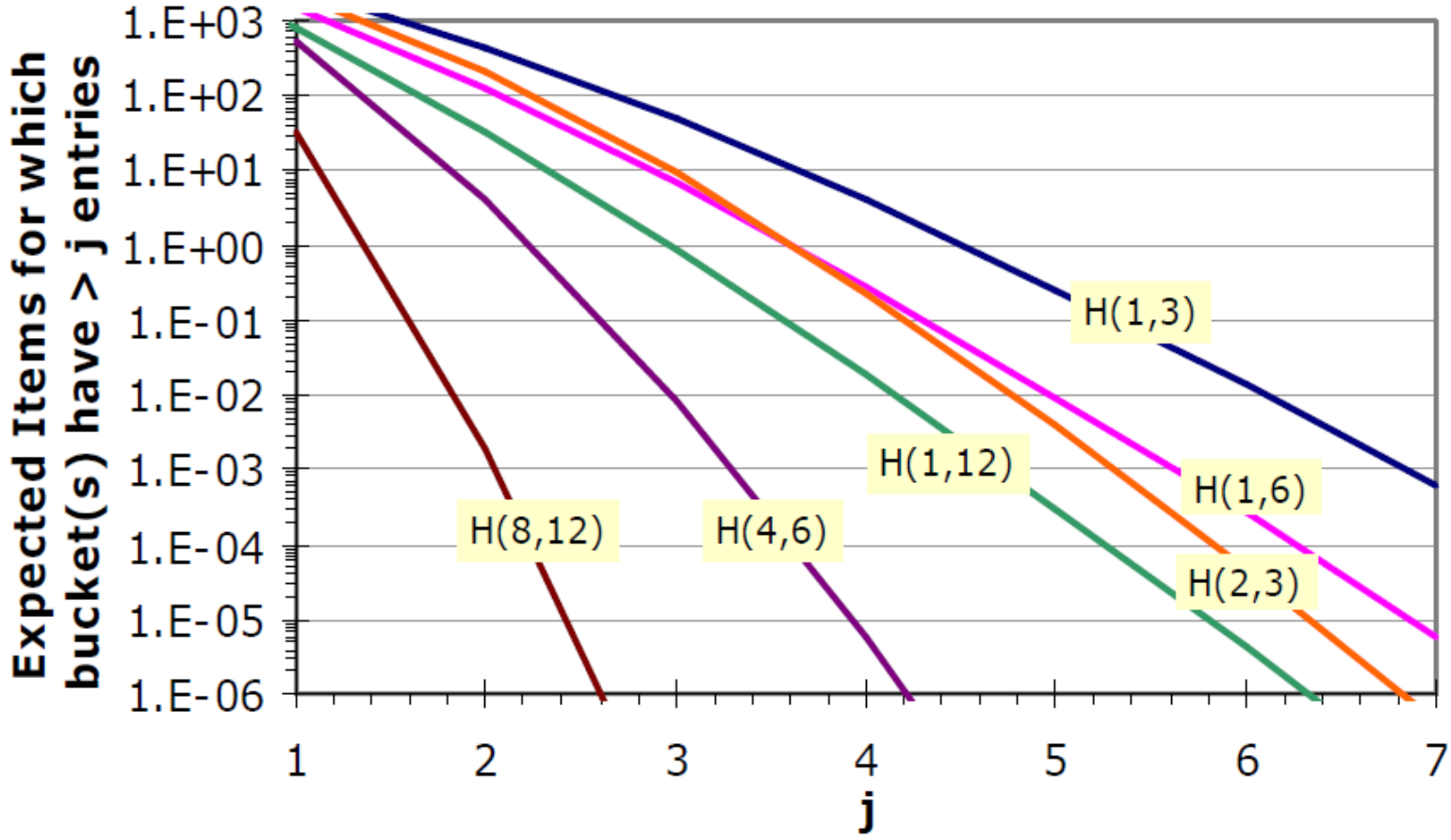


Figure 7: The effect of optimal configuration of hash table. $H(i,j)$ indicates $i = k$ and $j = m/n$. When $i = 1$, it implies an NHT.

Analysis: Effect of the Number of Hash Functions

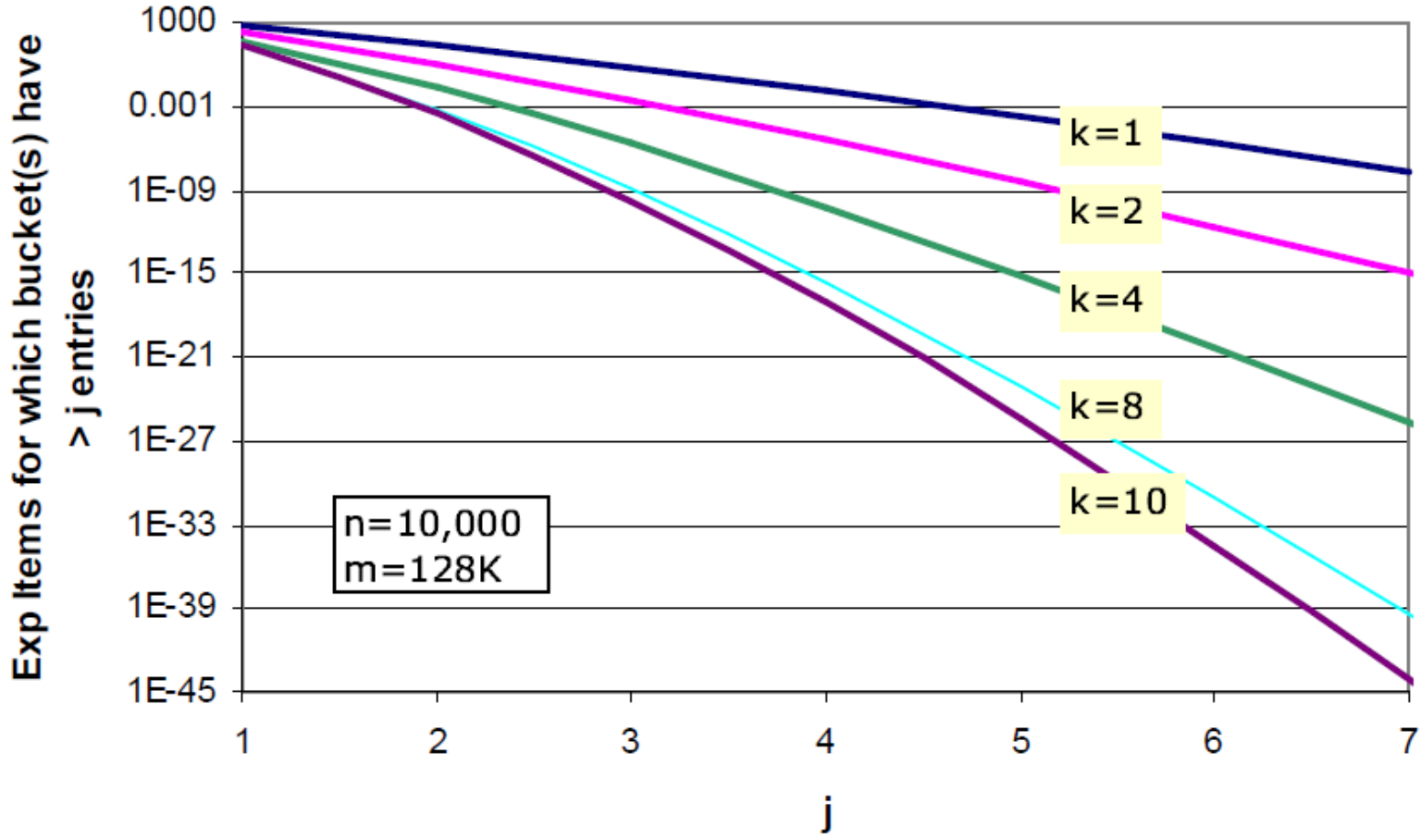


Figure 8: The effect of non-optimal configuration of FHT. $k = 1$ corresponds to the NHT.

Analysis: Average Access Time

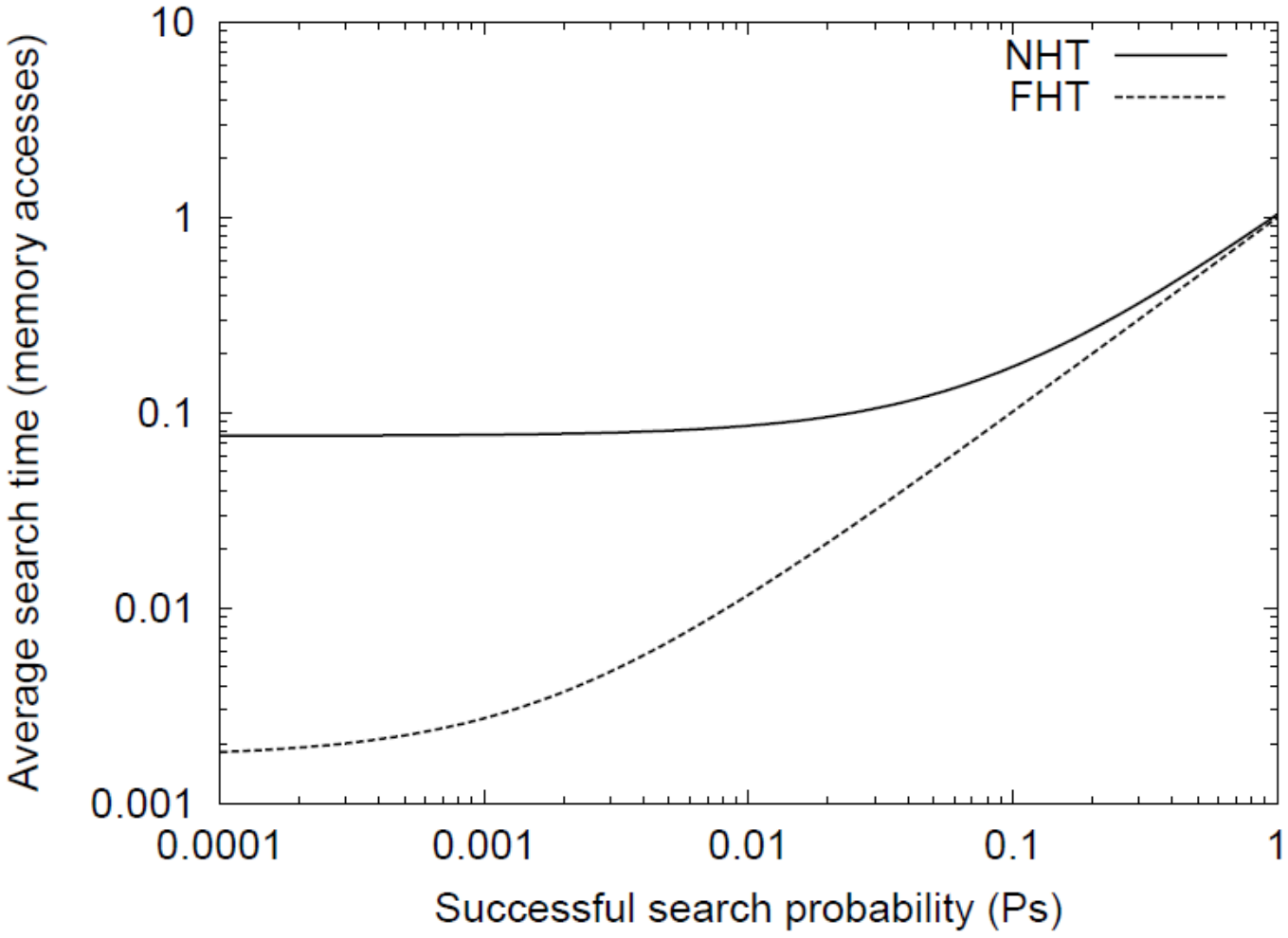


Figure 9: Expected search time for the NHT and FHT as a function of successful-search rate. $m = 128K$, $n = 10,000$. $k = 10$ **for FHT**

Analysis: Memory Usage

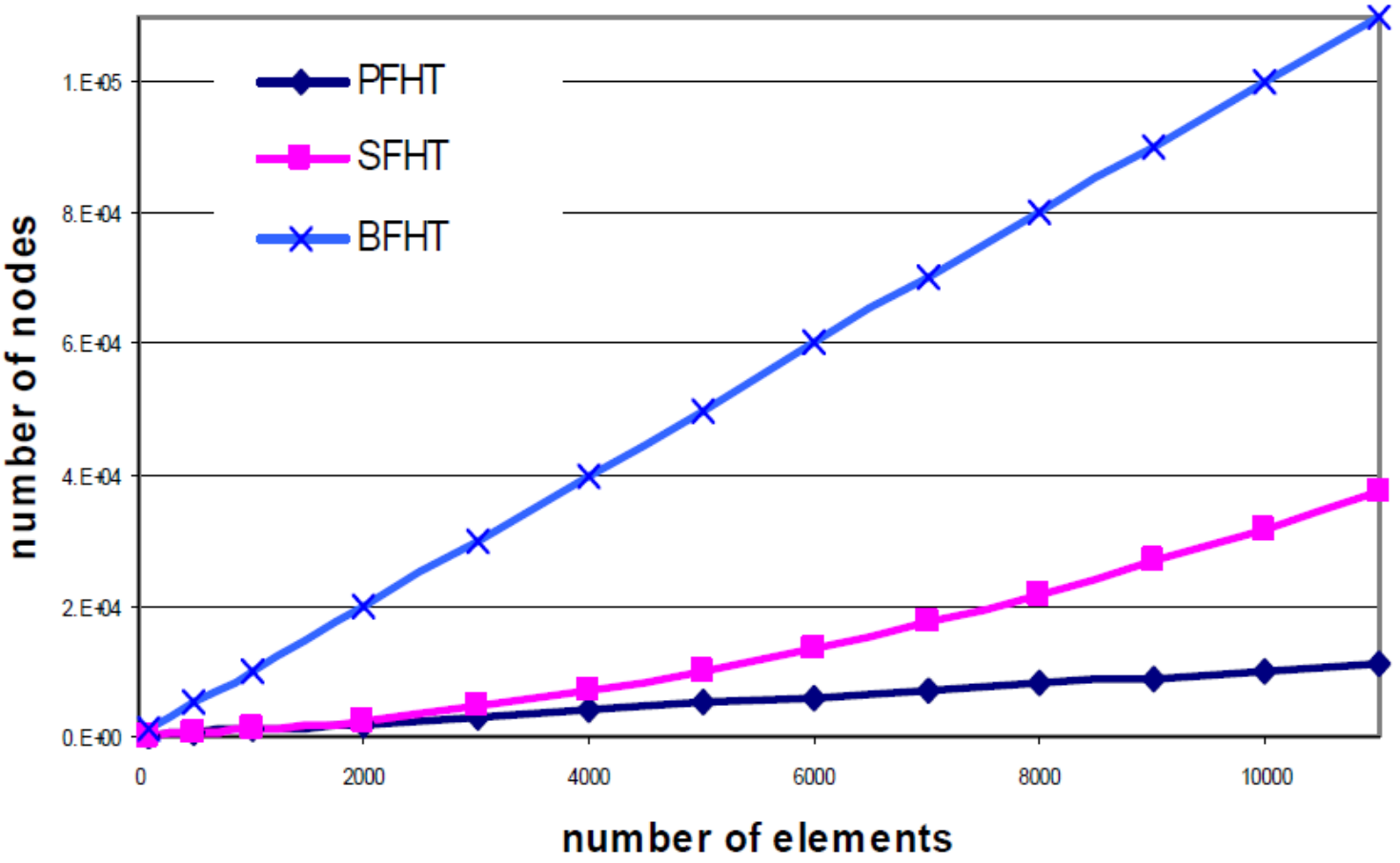


Figure 10: Item memory usage of different schemes. $m = 128K$, $n = 10,000$ **and** $k = 10$.

Analysis: Simulation

j	*Fast Hash Table*				*Naive Hash Table*	
	Analysis	*Simulation*			*Analysis*	*Simulation*
		basic	*pruning*	*balancing*		
1	19.8	18.8	5.60×10^{-2}	0	740.32	734.45
2	3.60×10^{-4}	4.30×10^{-4}	0	0	28.10	27.66
3	2.21×10^{-10}	0	0	0	0.72	0.70
4	1.00×10^{-17}	0	0	0	1.37×10^{-2}	1.31×10^{-2}
5	5.64×10^{-26}	0	0	0	2.10×10^{-4}	1.63×10^{-4}
6	5.55×10^{-35}	0	0	0	2.29×10^{-6}	7×10^{-6}

Table 1: Expected # of items for which all buckets have $> j$ entries. In the table, $n = 10,000$ and $m = 128K$. $k = 10$ for FHT

Conclusion

- Extends the multi-hashing technique, Bloom filter, to support exact match
- Provides better bounds on hash collisions and the memory access per lookup
  - it requires only one external memory for lookup

Thanks & Questions