Cache-Oblivious Dictionaries and Multimaps with Negligible Failure Probability

Justin Thaler, Harvard University
Joint work with:
Michael Mitzenmacher (Harvard)
Michael T. Goodrich (UC Irvine)
Daniel S. Hirschberg (UC Irvine)

Dynamic Dictionaries: Statement of Results

Dynamic Dictionaries

- Goal: Maintain a set of *n* (key, value) pairs.
  - Assume each key is associated with a **unique** value.
  - E.g. Employees and salaries, symbol table within a compiler.

- Must support the following operations efficiently (ideally constant worst-case time per operation):
  - Insert(k, v)
  - Delete(k, v)
  - Lookup(k)

- Goal: Use close to minimum amount of space: $(1+\varepsilon)n$ words of memory for some small constant $\varepsilon > 0$.

New Goal: Negligible Failure Probability

- We aim for structures with **sub-polynomial** failure probability.
  - That is, all operations succeed in worst-case constant time with probability say $1-1/n^{\log n}$.

- Motivations:
  - Use in cryptographic applications like oblivious RAM simulation, prevention of timing attacks, and clocked adversaries.
  - Handling super-polynomially long sequences of updates.

Our Dictionary Results

- **Assuming "sufficiently random hash functions" that can be evaluated in constant time:**
  - For any ε, k > 0, we use (1+ε)n words of memory and:
  - With probability $1-1/n^{\log^k n}$, all inserts, deletes, and lookups will run in time O(1).

- Previous work based on cuckoo hashing [Arbitman, Naor, Segev 2010] achieved this but with polynomial failure probability.

Our Dictionary Results

- **Assuming "sufficiently random hash functions" that can be evaluated in constant time:**
  - For any ε, k > 0, we use $(1+\varepsilon)$n words of memory and:
  - With probability $1-1/n^{\log^k n}$, all inserts, deletes, and lookups will run in time O(1).

- Previous work based on cuckoo hashing [Arbitman, Naor, Segev 2010] achieved this but with polynomial failure probability.

- "Sufficiently random hash functions" = (almost) n^{α}-wise independent hash family.
  - It is open how to construct these with O(1) evaluation time and $1/n^{\omega(1)}$ failure probability.
  - We give partial results toward making the failure probability subpolynomial, building on [Siegel 2004].

Dynamic Multimaps: Statement of Results

Dynamic Dictionaries

- Goal: Maintain a set of n (key, value) pairs.
  - Assume each key is associated with a **unique** value.
  - E.g. Employees and salaries, symbol table within a compiler.

- Must support the following operations efficiently (ideally constant worst-case time per operation):
  - Insert(k, v)
  - Delete(k, v)
  - Lookup(k)

- Goal: Use close to minimum amount of space: $(1+\varepsilon)n$ words of memory for some small $\varepsilon > 0$.

Dynamic Multimaps

- Goal: Maintain a set of *n* (key, value) pairs.
  - Assume each key is associated with a **unique** value.
  - E.g. Employees and salaries, symbol table within a compiler.

- Must support the following operations efficiently (ideally constant worst-case time per operation):
  - Insert(k, v)
  - Delete(k, v)
  - Lookup(k)

- Goal: Use close to minimum amount of space: (1+ε)n words of memory for some small ε > 0.

Dynamic Multimaps

- Goal: Maintain a set of n (key, value) pairs.
  - Each key may be associated with many values.

Dynamic Multimaps

- Goal: Maintain a set of n (key, value) pairs.
  - Each key may be associated with many values.
- Applications:
  - Inverted indices.
    - Keys are words, and each value is a document that contains the word.
    - Used in web search.
  - Graphical data.
    - Efficient adjacency list representation.
    - Keys are vertices, and the values are its neighbors.

Dynamic Multimaps

- Goal: Maintain a set of n (key, value) pairs.
  - Each key may be associated with many values.
- Applications:
  - Inverted indices.
    - Keys are words, and each value is a document that contains the word.
    - Used in web search.
  - Graphical data.
    - Efficient adjacency list representation.
    - Keys are vertices, and the values are its neighbors.

- Must support the following operations efficiently:
  - Insert(k, v)
  - Delete(k, v)
  - Lookup(k, v)
  - **FindAll(k)**
  - **RemoveAll(k)**

Dynamic Multimaps

- Multimaps are now a standard abstraction.
  - Multimaps appear in the C++ Standard Template Library, Google Java Collections Library, and Apache Common Collections API.
  - Somebody thinks they're useful!

Dynamic Multimaps

- Multimaps are now a standard abstraction.
  - Multimaps appear in the C++ Standard Template Library, Google Java Collections Library, and Apache Common Collections API.
  - Somebody thinks they're useful!

- Possible Approaches:
  - C++ Standard Template library uses red-black trees.
  - O(log n) worst-case operations.
  - What about hashing?

Our Work: External Memory Multimaps

- For big data sets, number of memory accesses is paramount.
  - Each memory block can store B items (B may be $\omega(1)$).

- Goal: minimize the number of memory blocks that must be touched (I/Os) especially for findAll(k) operations.
  - Requires keeping all values associated with a particular key in contiguous memory.

Our Work: External Memory Multimaps

- For big data sets, number of memory accesses is paramount.
  - Each memory block can store B items (B may be $\omega(1)$).
- Goal: minimize the number of memory blocks that must be touched (I/Os) especially for findAll(k) operations.
  - Requires keeping all values associated with a particular key in contiguous memory.
- Additional goal: be **cache oblivious**.
  - Algorithm shouldn't be tuned for parameters of the memory hierarchy, like the block size B.
  - Prior work [Angelino et al. 2011] gave a cache-aware dynamic multimap implementation.

Dynamic Dictionaries in the Standard RAM Model

Background: Q-Heaps and Q*-Heaps

- Q-heaps [Fredman and Willard, 1993] support worst-case O(1)-time inserts, deletes, lookups, and predecessor queries into subsets of size $O(\log^{1/5} n)$ from a 'master set' of size n.
  - Require o(n) space and preprocessing time for pre-computed lookup tables shared among all the subsets.
  - Can be made to work in the AC^0 RAM model.

Background: Q-Heaps and Q*-Heaps

- Q-heaps [Fredman and Willard, 1993] support worst-case O(1)-time inserts, deletes, lookups, and predecessor queries into subsets of size $O(\log^{1/5} n)$ from a 'master set' of size n.
  - Require o(n) space and preprocessing time for pre-computed lookup tables shared among all the subsets.
  - Can be made to work in the AC^0 RAM model.

- **Q*-heap is a constant-depth B-tree with internal nodes implemented as Q-heaps.**
  - **Worst-case constant-time inserts, deletes, and lookups for subsets of size $O(\log^c n)$ for an arbitrary constant c > 0.**

First Idea for a Dynamic Dictionary

Q*-heap of capacity $6\log^3 n$

Q*-heap of capacity $6\log^3 n$

Q*-heap of capacity $6\log^3 n$

Q*-heap of capacity $6\log^3 n$

Each time a (k, v) pair is inserted, hash it to a random bucket

Use $n/\log^3 n$ buckets, each of capacity $6\log^3 n$.

Analysis

- Expected number of items mapped to any bucket is $\log^3 n$.
- Each bucket has capacity $6\log^3 n$.
- By Chernoff bounds, any bucket overflows with probability $1/n^{\log^3 n}$.
- By union bound over buckets, *no* bucket overflows with probability $n/n^{\log^3 n}$.
- Problem: space usage is > 6n words of memory. How can remove the 6?

Second idea

- Use "Front Yard" of [Arbitman, Naor, Segev 2010] to create a two-level hashing scheme.
- The top level keeps $m = (1 + \varepsilon/2)n/d$ "bins" of size d, where d is a suitably chosen constant that depends on ε.
- Lookups, inserts, and deletes to each top-level bin can trivially be done in time $O(d)=O(1)$.

Second idea

- Use "Front Yard" of [Arbitman, Naor, Segev 2010] to create a two-level hashing scheme.
- The top level keeps $m = (1 + \varepsilon/2)n/d$ "bins" of size d, where d is a suitably chosen constant that depends on ε.
- Lookups, inserts, and deletes to each top-level bin can trivially be done in time O(d)=O(1).
- With $1/n^{\omega(1)}$ probability, at most $(\varepsilon/16)n$ items will "overflow" from the top level.
  - Use our array of Q*-heaps to handle the overflow.
  - Holds as long as hash functions are n^{α}-wise independent for some $\alpha > 0$.

Dynamic Multimaps in the External Memory Model

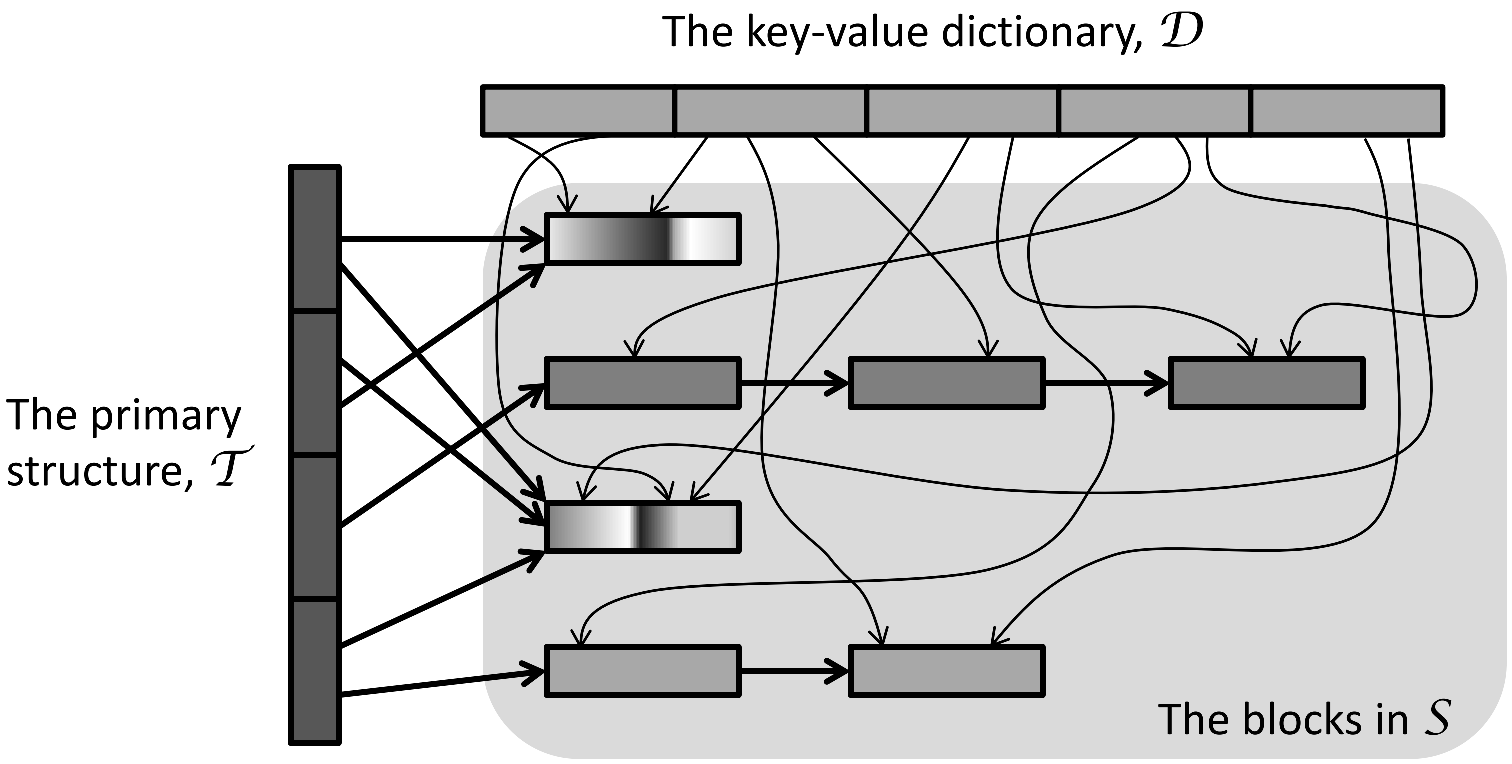
Recall: Dynamic Multimaps

- Goal: Maintain a set of n (key, value) pairs.
  - Each key may be associated with many values.
- Must support the following operations efficiently:
  - Insert(k, v)
  - Delete(k, v)
  - Lookup(k, v)
  - **FindAll(k)**
  - **RemoveAll(k)**
- Want to use O(n) words of memory.

Methodology

- Utilize two data structures.
- A fast dictionary data structure. Supports fast Insert(k, v), Delete(k, v), and Lookup(k, v) operations.
- External-memory multiqueues (for fast FindAlls and RemoveAlls).
  - Keep values associated with each key in a queue.
  - Need to keep entire queue in *contiguous* memory while using O(n) space.

Pictorial Representation



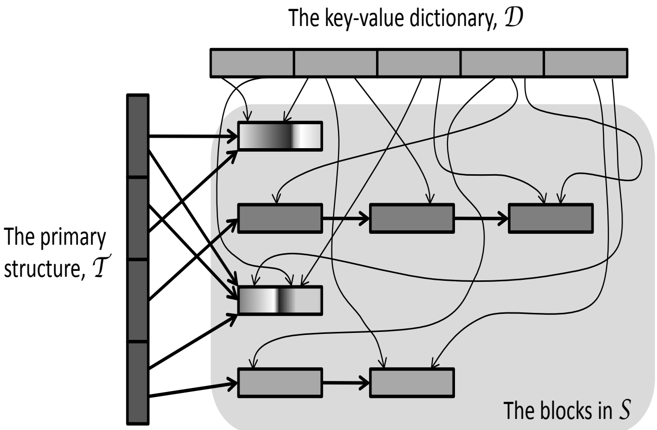
Our Solution, Step-by-Step

Method
Lookup(k, v)
Insert(k, v)
Delete(k, v)
FindAll(k)
RemoveAll(k)

Our Solution, Step-by-Step

Method
Lookup(k, v)
Insert(k, v)
Delete(k, v)

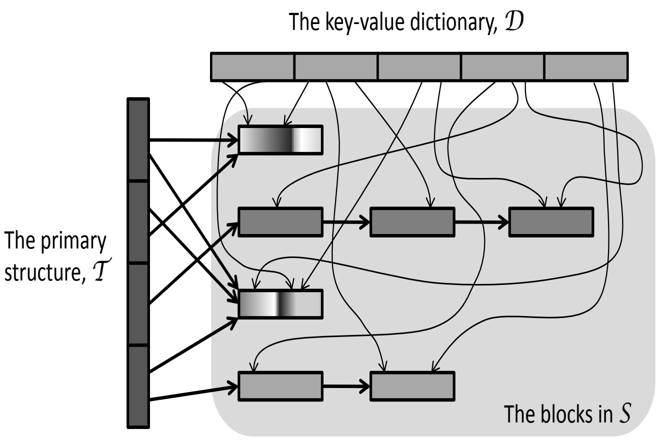
- Keep (k, v) pairs in a dynamic dictionary D.
- Supports lookups, inserts, and deletes worst case O(1) time.



Our Solution, Step-by-Step

Method
FindAll(k)
RemoveAll(k)

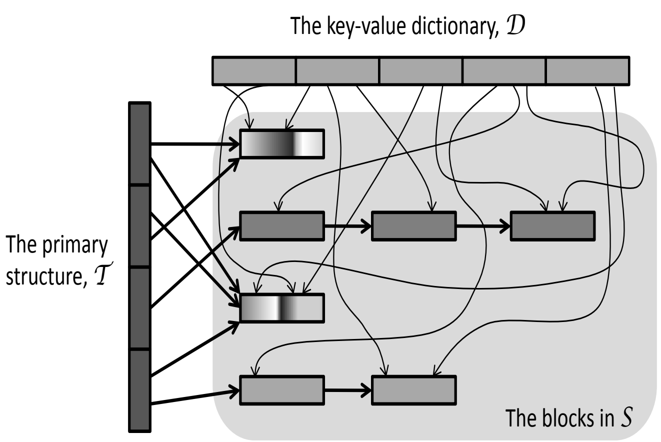
- For each key k, keep array A_k of all values associated with k.
- To find k's array quickly, keep *second* dynamic dictionary *T* of (k, ptr(A_k)) pairs, where ptr(A_k) points to k's array.



Our Solution, Step-by-Step

Method
FindAll(k)
RemoveAll(k)

- For each key k, keep array A_k of all values associated with k.
- To find k's array quickly, keep *second* dynamic dictionary *T* of (k, ptr(A_k)) pairs, where ptr(A_k) points to k's array.

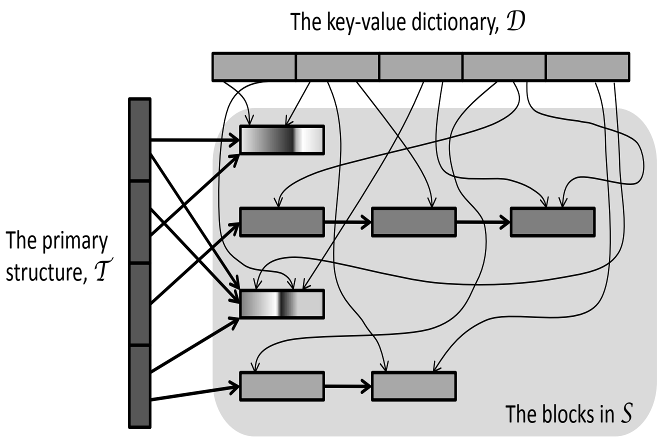


- If a value is deleted from A_k, keep A_k contiguous by moving last item in the array into the vacated position.

Our Solution, Step-by-Step

Method
FindAll(k)
RemoveAll(k)

- For each key k, keep array A_k of all values associated with k.
- To find k's array quickly, keep *second* dynamic dictionary *T* of (k, ptr(A_k)) pairs, where ptr(A_k) points to k's array.

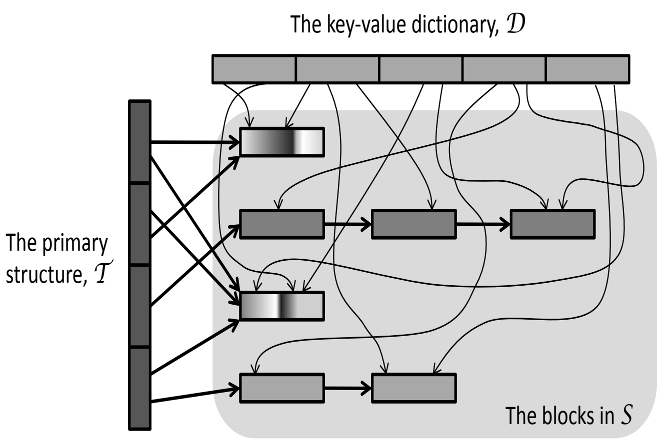


- If a value is deleted from A_k, keep A_k contiguous by moving last item in the array into the vacated position.
- Need to dynamically expand and shrink arrays. Can do in constant time using "Improved Buddy System" of [Brodal et al.]

Our Solution, Step-by-Step

Method
FindAll(k)
RemoveAll(k)

- During a RemoveAll(k), just delete k's entry from *T* and free A_k. Cannot afford to remove all (k, v) entries from *D* at this time.
- This creates "spurious" (k, v) entries in *D* that must be dealt with.



Conclusions

- We give new dictionary and multimap data structures that support constant-time worst-case operations with subpolynomial failure probabilities.
  - Our dictionary is for the standard RAM model and use $(1+\varepsilon)n$ words of memory.
  - Our multimap is for the external memory model. It uses $O(n)$ words of memory and is **cache oblivious.**

- Open questions about hash functions remain.
  - Instantiate Siegel's hash functions with subpolynomial failure probability and polynomial preprocessing time?
  - Or avoid using n^{α}-wise independent hash functions entirely?

Thank you!

Hash Functions with O(1)-Evaluation Time and Negligible Failure Probability

Siegel's Construction

- Given: a universe U.
- Store a fixed bipartite graph G of degree d=O(1), where there is a left vertex for each universe item x.
- Populate each right vertex v with a random value R[v].
- Define $h(x) = \bigoplus_{v \text{ in } N(x)} R[v]$.

Siegel's Construction

- Define $h(x) = \bigoplus_{v \text{ in } N(x)} R[v]$.
- If G is a good vertex expander for sets S of size $< k$, then this defines a k-wise independent hash family.
  - That is: for any distinct $x_1, x_2, \ldots, x_k$:
    $(h(x_1), h(x_2), \ldots, h(x_k))$ is uniformly distributed.

Siegel's Construction

- Define $h(x) = \bigoplus_{v \text{ in } N(x)} R[v]$.
- If G is a good vertex expander for sets S of size $< k$, then this defines a k-wise independent hash family.
  - That is: for any distinct $x_1, x_2, \ldots, x_k$: $(h(x_1), h(x_2), \ldots, h(x_k))$ is uniformly distributed.
  - Idea: For any set S of size $< k$, the expansion property of G guarantees $\exists\, v \in N(S)$ with exactly one neighbor $x_1 \in S$.
  - Then $h(x_1)$ will be independent of $h(x_i)$ for all $i \neq 1$.
  - Intuitively, we can then "ignore" x and iterate the argument on the set $S \setminus \{x_1\}$, which also expands well.

Wait a minute

- G is huge – it has a vertex for each universe item.
- Instead, store a succinct representation of G.
  - Store a "small" expander G' of size $O(n^{\beta})$ for some $\beta < 1$.
  - When evaluating the hash function, blow G' up into a large expander G "on-the-fly" using graph products.
  - Only works for polynomial-sized universes.

Two Sources of "Failure"

- There are two sources of failure in Siegel's construction.
  1. There are no known explicit constructions of the expanders Siegel needs.
     - So he generates a graph at random and hopes it is an expander.
  2. If the universe has superpolynomial size, it must first be hashed down to a poly-sized universe before applying Siegel's construction.
     - This introduces "collisions" with probability 1/poly(n).

Two Sources of "Failure"

- There are two sources of failure in Siegel's construction.
  1. There are no known explicit constructions of the expanders Siegel needs.
     - So he generates a graph at random and hopes it is an expander.

  2. If the universe has superpolynomial size, it must first be hashed down to a poly-sized universe before applying Siegel's construction.
     - This introduces "collisions" with probability 1/poly(n).

Addressing Failure Source 1

- There are no known explicit constructions of the expanders Siegel needs.
  - So he generates G at random and hopes it is an expander.

- Observation: the probability G is not an expander is dominated by the probability that small sets of vertices fail to satisfy the condition.

Addressing Failure Source 1

- There are no known explicit constructions of the expanders Siegel needs.
  - So he generates G at random and hopes it is an expander.

- Observation: the probability G is not an expander is dominated by the probability that small sets of vertices fail to satisfy the condition.
  - To get failure probability $1/n^{\log^k n}$, randomly generate G and exhaustively check the expansion of all sets of size $\leq \log^k n$.
  - Requires quasi-polynomial preprocessing time, but constant evaluation time "online."

Two Sources of "Failure"

- There are two sources of failure in Siegel's construction.
  1. There are no known explicit constructions of the expanders Siegel needs.
     - So he generates a graph at random and hopes it is an expander.
  2. If the universe has superpolynomial size, it must first be hashed down to a poly-sized universe before applying Siegel's construction.
     - This introduces "collisions" with probability 1/poly(n).

Two Sources of "Failure"

- If the universe has superpolynomial size, it must first be hashed down to a poly-sized universe before applying Siegel's construction.
  - This introduces "collisions" with probability 1/poly(n).

- Idea: run say loglog n independent copies of Siegel's construction, and define h(x) as the XOR of the results.
  - For any set S, if even one copy of Siegel's construction is fully random on S, then the XOR will also be fully random on S.
  - Requires evaluation time O(log log n) and has failure probability $1/n^{\text{loglog } n}$.