Data Structures for IPv6 Network Traffic Analysis Using Sets and Bags

John McHugh, Ulfar Erlingsson

The nature of the problem

- IPv4 has 2^{32} possible addresses, IPv6 has 2^{128}.
- IPv4 sets can be realized as bit arrays. (0.5GB)
- IPv4 bags can be realized as sparse arrays if
  - reasonably good locality is present in the data
  - an efficient lookup structure is used
- Current SiLK implementations work reasonably well, but can be improved using
  - block allocation strategies,
  - adaptive counter sizes, but
    - that is a story for another day

IPv6 is not business as usual

- Bit arrays and pointer based sparse arrays are infeasible for IPv6
  - Too many pointer levels to reach the real data
  - Don’t know anything about possible locality leverage
- This talk will look at several alternatives.
  - Tree representations
  - Hash based representations
    - Bloom filters
    - Perfect Hashes
    - Cukcoo Hashes
  - Column oriented databases

Requirements

- We impose stronger requirements that those imposed by the current SiLK implementation.
  - Constant time access and insertion
  - Indexing by composite quantities (connections, services, etc.)
- The first comes from a desire to be able to build sets and bags in real time and to use sets for filtering in real time (outside SiLK).
- The second is motivated by some of our visualization needs for connection bundles and the like.

More Requirements

- We want to be able to do the usual set and bag operations with reasonable efficiency.
  - We do not anticipate real time requirements for set union, bag add and inversion, etc.
- We would like to be open ended in the type of data stored in bags.
  - Within the current structure, we have implemented "time bags" in which the payload is first and last seen epoch times.
  - We also have index bags using powerset bitmaps
  - User defined payloads at the library level would be useful.

General considerations

- Index representation
  - Current sets / bags use implicit representations for the index sets
  - All the replacements will require explicit index set representation
  - Keys require space in the tables or in auxiliary storage.
- Operations
  - Set and bag operations generally require sorted index lists imposing a potential O(N log(N)) operation on the N actual keys of each list.
  - With sorted keys, the operations are O(N) in the total number of operand keys in general.

Trees

- SiLK uses red / black trees for a number of purposes.
  - Lookup and insert operations are O(log(N))
  - About 2 pointers / entry so 50% space usage if pointer and entries are same size
  - Inherently sorted, so operations are O(N)
  - Marginally useful for real time.
  - Could be adapted to arbitrary keys, index sets
- Existing implementation will be used as a base for comparison

Bloom Filters

- Capture sets of arbitrary keys
  - Bit array, indexed by multiple, independent, hash functions.
    - Entry: N functions will set up to N bits.
    - Lookup: N bits set -> hit (or fp); <N bits set -> miss
  - No false negatives, bounded false positives
    - P(collision) function of %bits set (parallel formulation of birthday paradox)
  - Optimum size can be calculated.
    - Typically a few bits per entry
  - Non invertible.
    - Separate key list must be maintained

Bloom Filter Operations

- Union (of similar filters) is bitwise AND
- Intersection can be approximated by bit operations
  - Risk of higher FP rate on subsequent insertions / lookups
- Other operations done on key lists with preliminary sort or by construction of new filter from combined key list.
- We have made a number of Bloom filter additions to SiLK, primarily for filtering and extracting exemplars of connectione, etc.
- Probably not suited for general set operations.

(Minimal) Perfect Hashes

- A perfect has takes a key set into indexes with no collisions.
  - A minimal perfect hash takes a key set of size N into 0...N-1 with no collisions.
  - Fast O(C) lookup, no insertion
- This can be useful when the key set is static and known, e.g. the IP addresses for a past month.
- MPH function generators exist for up to several billion keys
  - We have experimented with using MPHs for MAC addresses in the Dartmouth wireless data to store them in the input / output index fields in SiLK.

MPH Operations

- Any operation that increases the key set size requires recomputing the hash function. But order preserving MPHs can be extended.
- Operations on the index sets can be done on the sorted index lists.
- Bag operations would require creation of a new MPH and combination of the existing bags into the new one.
- The best usage of MPHs with network data would be for activity indices for historical data
    - Hourly powerset index for month (750 bits/IP/month)
    - 5 year monthly index (50 bits/IP)
    - etc.

Cuckoo Hashes

- Like Bloom filters. Uses multiple hashes, but resolve collisions by evicting and rehashing.
  - O(C) lookup, insertion.
  - Multiple hashes can be parallelized on multicore
    - Can guarantee 50% space utilization w 2 functions
    - above 90% with 4. Usually better.
- Must reallocate to a larger size table and rehash if a collision cycle occurs.
  - Estimating index set size will help.
  - Not an issue for non-real time as cost amortizes

Cuckoo Hash Operations

- In general, operations on sets and bags realized as Cuckoo hashes require construction of the resulting table from scratch.
  - Depending on whether an ordered index list is needed, a sort may be involved, but operations such as union, addition, etc. can be done in O(N) time where N is the total number of entries in all operands.
- The high space utilization and constant time behavior appear to make cuckoo hashes a viable candidate for general purpose set and bag implementations at IPv6 and for composite keys such as connections.

Experimental Results

- Preliminary cuckoo hash results
- Implemented using hash functions from Arash Partow - http://www.partow.net modified for counted strings. Arbitrary choice of 4 of 10
- 4 functions, table into 4 disjoint parts. 7 level BFS for collision resolution
- Instrumentation includes eviction count, hash coverage, percent utilization
  - some backtrace when entry fails.
- Table regenerates (size doubles) when not possible to add an entry. Must copy and rehash contents.

Trial cases

- table with 2^{24} entries.
- hashed 2^{24} random() keys
  - length 4, 8, 12, 16 bytes
- looked at a number of statistics
  - table utilization, evictions, regenerations
  - hash function coverage

Statistics at regeneration 1

- Key (data) 16 (8), size 16,777,216, - 2,071,506 evictions
  4 hash functions, 13,001,138 entries, 77.49% full
- Key (data) 12 (8), size 16,777,216 - 2,032,597 evictions
  4 hash functions, 12,982,761 entries, 77.38% full
- Key (data) 8 (8), size 16,777,216 - 2,300,169 evictions
  4 hash functions, 13,089,848 entries, 78.02% full
- Key (data) 4 (8), size 16,777,216, - 1,503,281 evictions
  4 hash functions, 9,312,033 entries, 55.50% full
-

Coverage at regeneration 1

- 16 byte keys: 16,282,508 of 16,777,216 97.05%
  - Hash[0] 4,070,540 of 4,194,304 97.05%
  - Hash[1] 4,070,635 of 4,194,304 97.05%
  - Hash[2] 4,070,965 of 4,194,304 97.06%
  - Hash[3] 4,070,368 of 4,194,304 97.05%

- 12 and 8 byte keys in 95%+ range
- 4 byte keys: 12,121,248 of 16,777,216 72.25%
  - Hash[0] 3,860,727 of 4,194,304 92.05%
  - Hash[1] 583,344 of 4,194,304 13.91%
  - Hash[2] 3,816,732 of 4,194,304 91.00%
  - Hash[3] 3,860,445 of 4,194,304 92.04%

What does it mean?

- Theory says that we should expect table loads in the 95% range, not upper 70s
  - May have a problem in the free space finder as it seems to terminate on graph cycles.
- The hash coverage indicates that the low bits of, at least, hash(2) are not what we want. Don't yet know if distributions are uniform.
- Nonetheless, the results are encouraging.

Coda: Column Oriented Databases

- Google uses a distributed database technology (Bigtable) in which entries are stored in columns, rather than in rows. It is claimed to offer high performance for datasets with billions of rows and thousands of columns.
  - The system can be distributed over thousands of servers, allowing wide distribution of data and processing.
  - Sparse columns are efficiently handled using Bloom filters to identify non empty rows.
- This type of organization would appear to be suited for storing massive amounts of NetFlow and similar data.

Acknowledgements

- Department of Homeland Security under Contract No. N66001-08-C-2032.
- Ron McLeod of TARA
- CA Labs
- NSERC