Quotient Filters: Approximate Membership Queries on the GPU

Afton Geil
University of California, Davis

GTC 2016

Outline

- What are approximate membership queries and how are they used?
- Background on quotient filters
- Quotient filter implementation on the GPU
- Performance results
- Conclusions & Future Work

Problem

- You run a web service with user accounts, and you allow users to choose their own unique usernames.
- When someone chooses a username, you need to make sure it is not already being used.
- The data is too large to be stored in memory, so it must be stored on disk, which means slow access times.
- Use a approximate membership query to quickly tell the user whether they need to pick different username.

Approximate Membership Queries (AMQs)

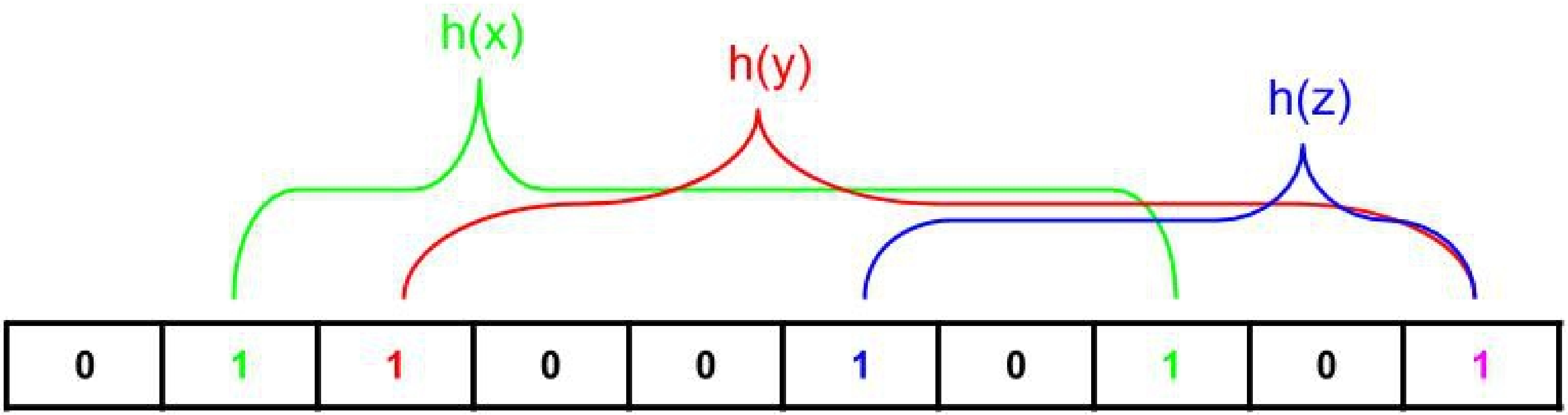
- Fast, small data structures for testing set membership
- Saves space and utilizes memory hierarchy to improve performance
- Want to know if item is in the set without retrieving the data from disk
- Applications in databases, networking, file systems, and more

Approximate Membership Queries (AMQs)

- AMQs return false positives with small, tunable probability
  - *False positive*- AMQ says the item is in the set, but it is not
- No false negatives
  - *False negative*- AMQ says the item is not in the set, but it actually is
- Answer membership queries with “item is probably in the dataset” or “item is not in dataset”

Bloom Filters

- The most well-known AMQ
- Bit array stores items using a set of hash functions
- No deletes
- Simple GPU implementation



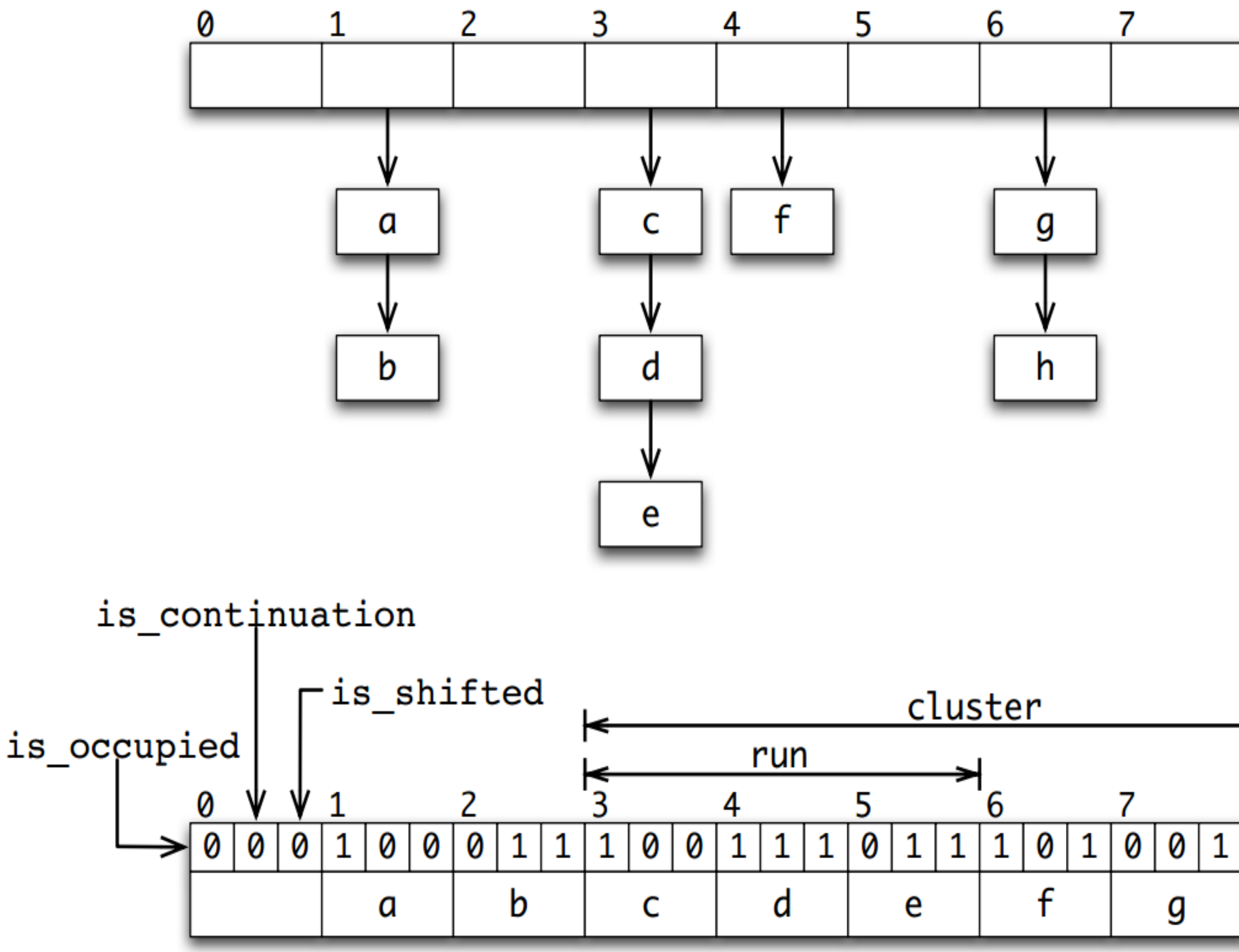
So what is a quotient filter?

- Like a Bloom filter, a quotient filter is a type of hash table.
- Each item is stored in a compressed format in a single slot in the hash table.
- Each slot also contains extra bits to handle collisions.

Quotient Filter Terms

- Quotient / Canonical slot
- Remainder
- Metadata bits
- Run
- Cluster
- How to find items in the quotient filter

Quotient Filter Basics



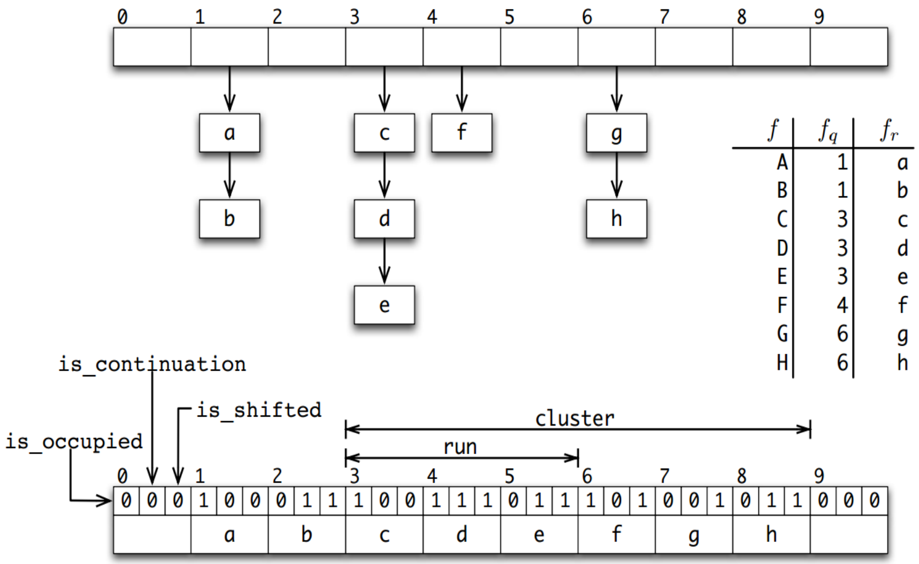
f	f_q	f_r
A	1	a
B	1	b
C	3	c
D	3	d
E	3	e
F	4	f
G	6	g
H	6	h

Image source: Bender, et al., 2012. "Don't thrash: how to cache your hash on flash".

Quotient Filter Basics

- Hash key; divide result into two parts:
  - q most significant bits = *quotient*, f_q
  - r least significant bits = *remainder*, f_r
- Quotient → canonical slot
- Remainder → value stored in QF
- Elements hash to the same slot → shift to the right

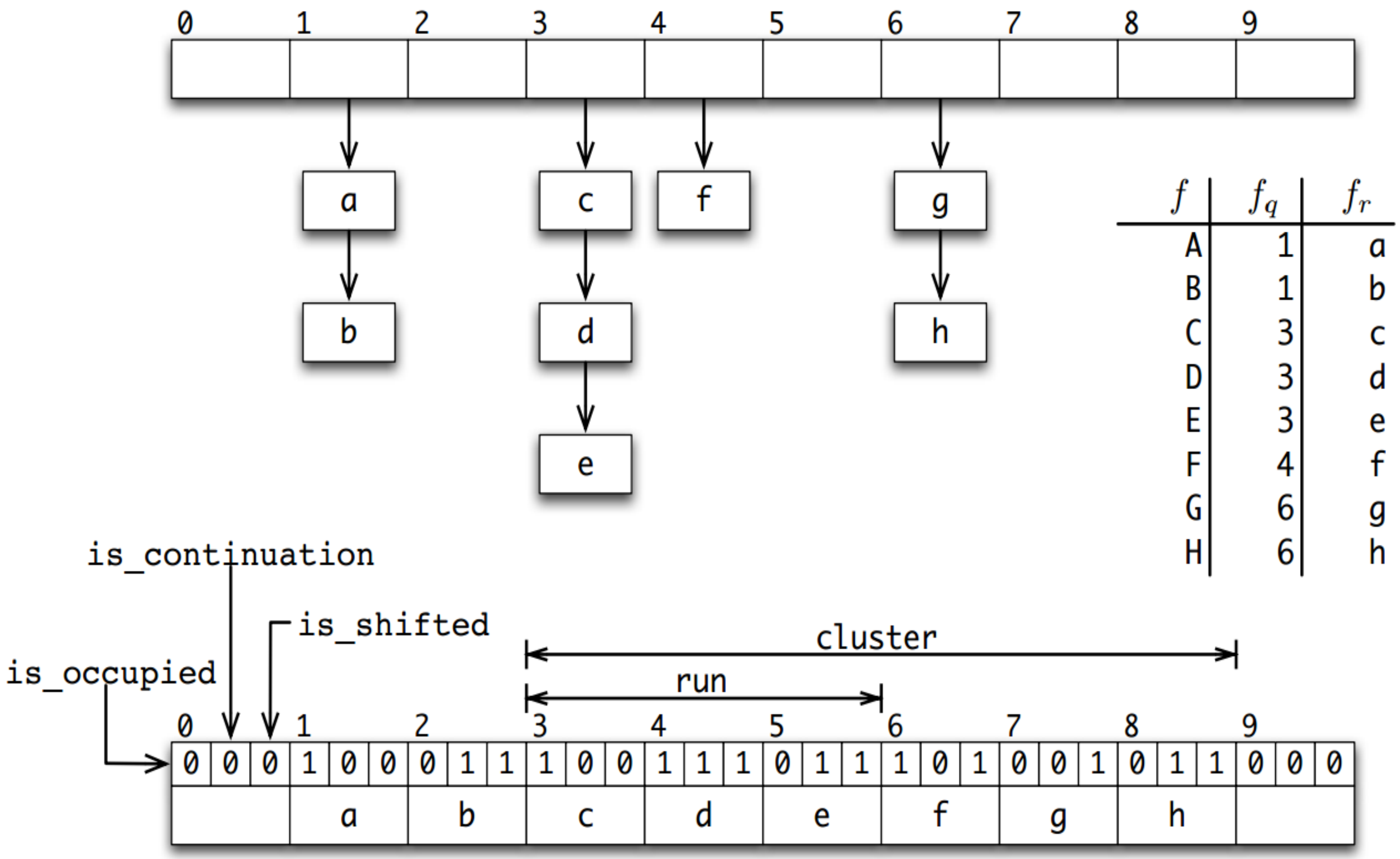
Quotient Filter Basics



- *Run*- group of items with same canonical slot
- *Cluster*- group of runs that have all been shifted

Quotient Filter Basics

- *Metadata*- 3 bits used to resolve collisions



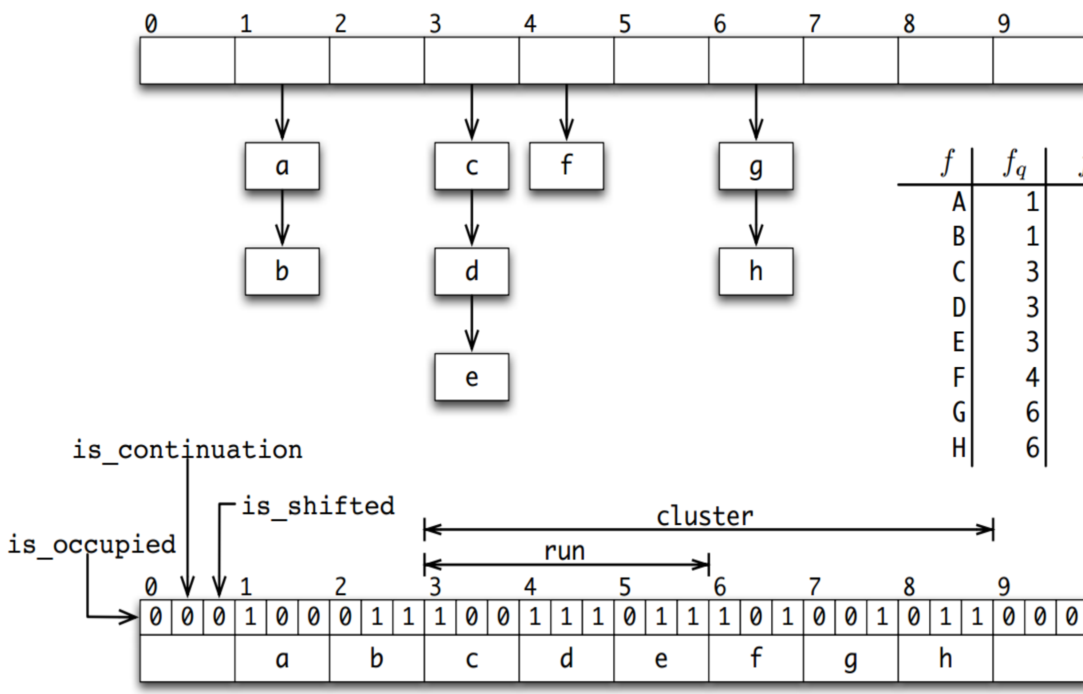
f	f_q	f_r
A	1	a
B	1	b
C	3	c
D	3	d
E	3	e
F	4	f
G	6	g
H	6	h

Metadata Bits: How to Deal with Collisions

- `is_occupied:` set when the slot is the canonical slot for a value stored in the filter (although it may not be stored in this particular slot).
- `is_continuation:` set when the slot holds a remainder that is not the first in a run.
- `is_shifted:` set when the slot holds a remainder that is not in its canonical slot.

Lookup Algorithm

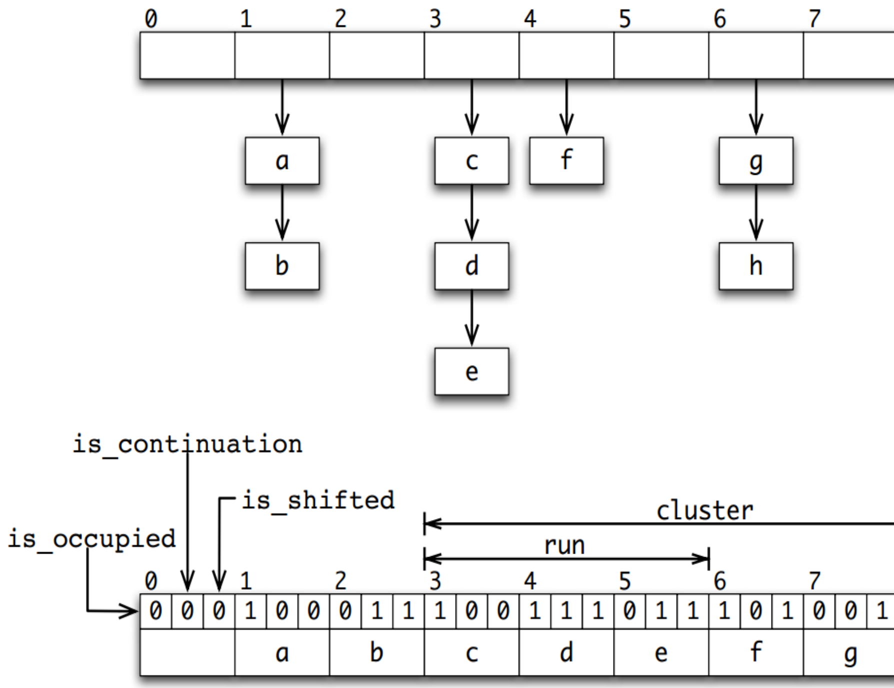
- Check canonical slot, f_q
  - If empty, item is not in filter
  - If occupied, item might be in filter → continue



f	f_q	f_r
A	1	a
B	1	b
C	3	c
D	3	d
E	3	e
F	4	f
G	6	g
H	6	h

Lookup Algorithm

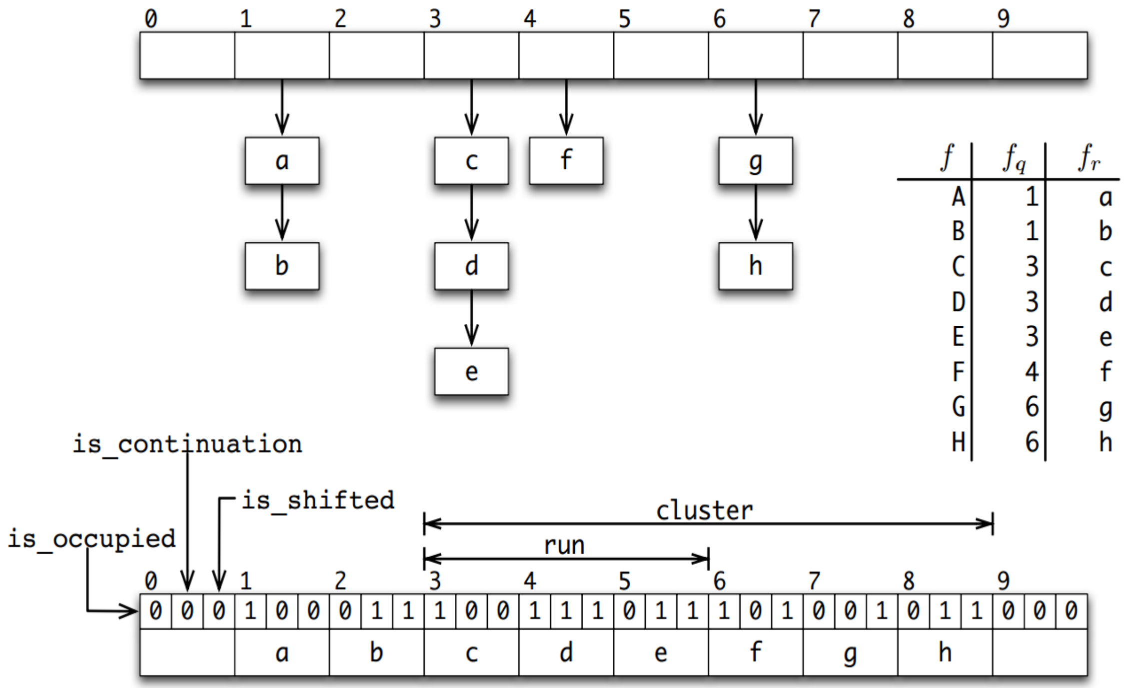
- Search to left, looking for beginning of cluster
  - Look for `is_shifted = false`
  - Count number of runs passed along the way by counting `is_occupied` bits



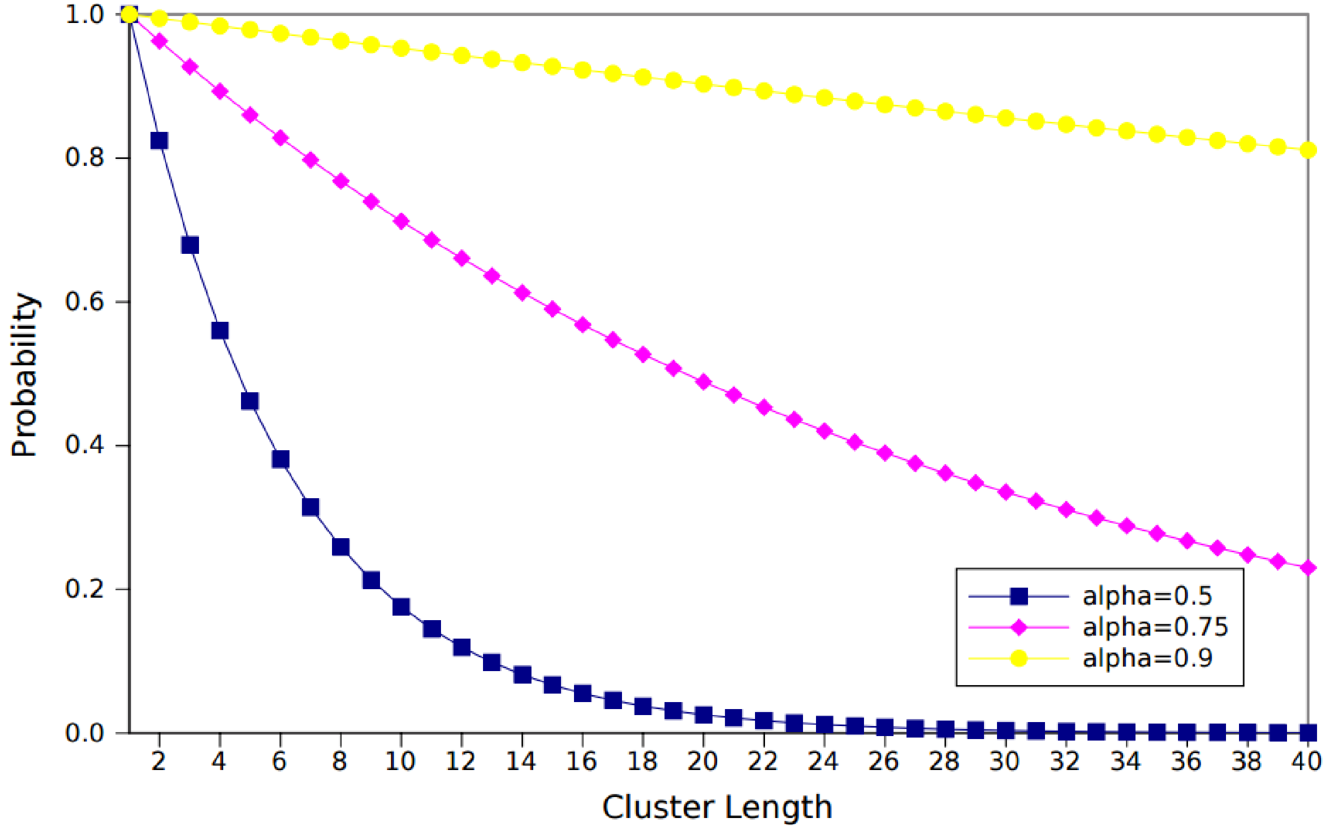
f	f_q	f_r
A	1	a
B	1	b
C	3	c
D	3	d
E	3	e
F	4	f
G	6	g
H	6	h

Lookup Algorithm

- Search right to find desired run
  - Each `is_continuation = 0` marks the start of a run
- Check slots in run for remainder, f_r



Cluster Length



Quotient Filter Advantages

- Much greater memory locality
- Can recover the keys from the data stored in the filter. This allows us to:
  - Delete items
  - Re-size the filter
  - Merge quotient filters

Challenges for Mutable Data Structures on the GPU

- Hard to avoid collisions when making changes in parallel
- Usually easier to just do a complete rebuild
- Can the advantage of better memory locality win out against the restrictions of avoiding collisions?
- Limited memory (< 12 GB)

Quotient Filters on the GPU

- Great memory locality
- Lookups are embarassingly parallel
- Inserts are much more difficult
  - All consecutive items to right of canonical slot may be modified
  - All consecutive items to the left and right of canonical slot may be read

Finding Parallelism in Modifications

- Varying numbers of bits/item → not all stored in the same word
  - Limit ourselves to number of bits/slot divisible by 8 to simplify and maximize available parallelism
- Items will be shifted to the right when new ones are inserted, so we must make sure two inserts do not overlap.
- *Superclusters*- independent regions
  - Separated by empty slots
  - Insert one item per supercluster at a time

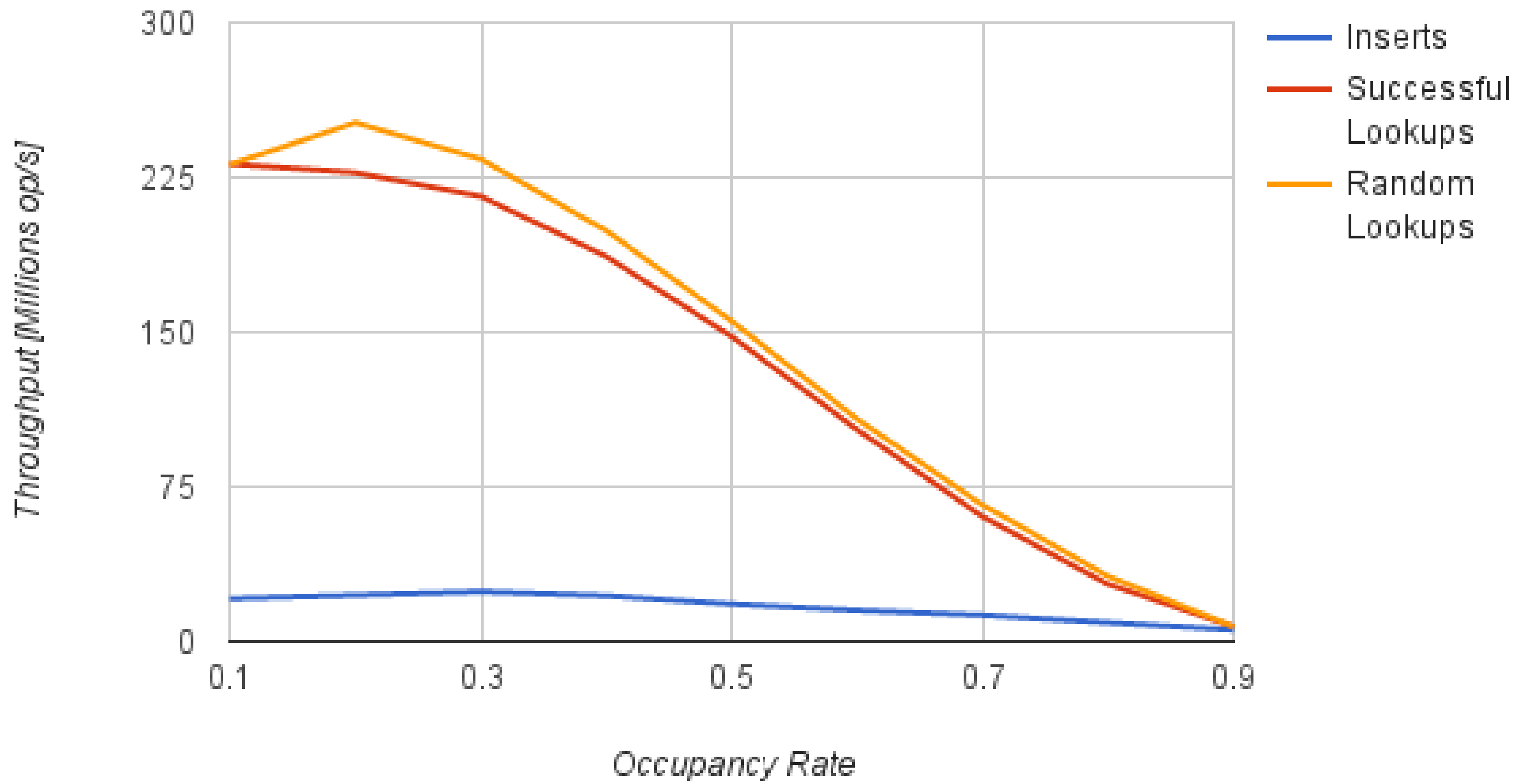
Finding Superclusters

- Let each slot have an indicator bit; initialize to 0.
- Each slot in filter checks its own value and slot to its left. If the slot is occupied and the slot to its left is empty, start of supercluster → set indicator bit to 1.
- Next, use prefix sum over indicator bits to label each slot with its supercluster number.

Supercluster Bidding & Inserts

- Supercluster bidding
  - Array with one item per supercluster
  - Each element in insert queue writes its index to its supercluster
  - Whichever thread wins gets its value sent to insert kernel
- Run insert kernel for winning values
- Remove these items from the queue
- Loop → parallelism reduced as filter gets fuller

Results: Performance Degrades as QF Fills Up



Results: Performance Comparison with Bloom Filter

	BloomGPU	Quotient Filter	Improvement
Inserts [Mops/s]	53.8	15.7	**0.3x**
Lookups [Mops/s]	55.0	163	**3x**

Results: Analysis

- Bloom filter performance is independent of occupancy level
- False positive rate for BF is dependent on fullness, whereas for QF it depends on number of remainder bits
- BloomGPU filters are 5x size of QF for same false positive
- Traditional BF is 10-25% smaller than QF

Which AMQ to use?

Attribute	GPU Quotient Filter	BloomGPU
Size	✔	
Insert Throughput		✔
Lookup Throughput	✔	
Deletes	✔	

Conclusions

- Insert performance limited by parallelism → high filter occupancy hurts twice as much
- BloomGPU beats us at inserts
- Our quotient filter implementation has faster lookups and uses less memory than BloomGPU
- Lookups are usually more frequent and performance-critical than inserts, so QF should be better in many cases

Future Work

- Speeding up inserts
- Merge two quotient filters- see how performance compares to normal batch inserts
- More real world datasets
- Cascade filters

Thanks!

Questions?

angeil@ucdavis.edu