TECHNISCHE HOCHSCHULE DEGGENDORF

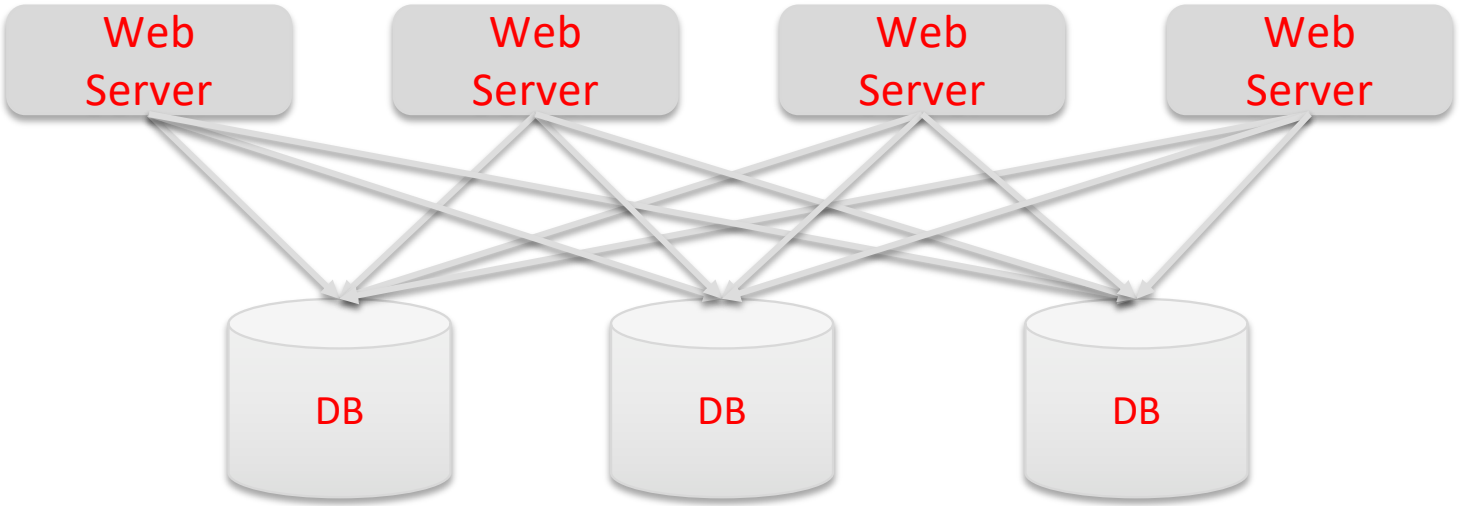
Datacenter Network Programming

In-Network Caching (NetCache)

Requirements for many cloud providers

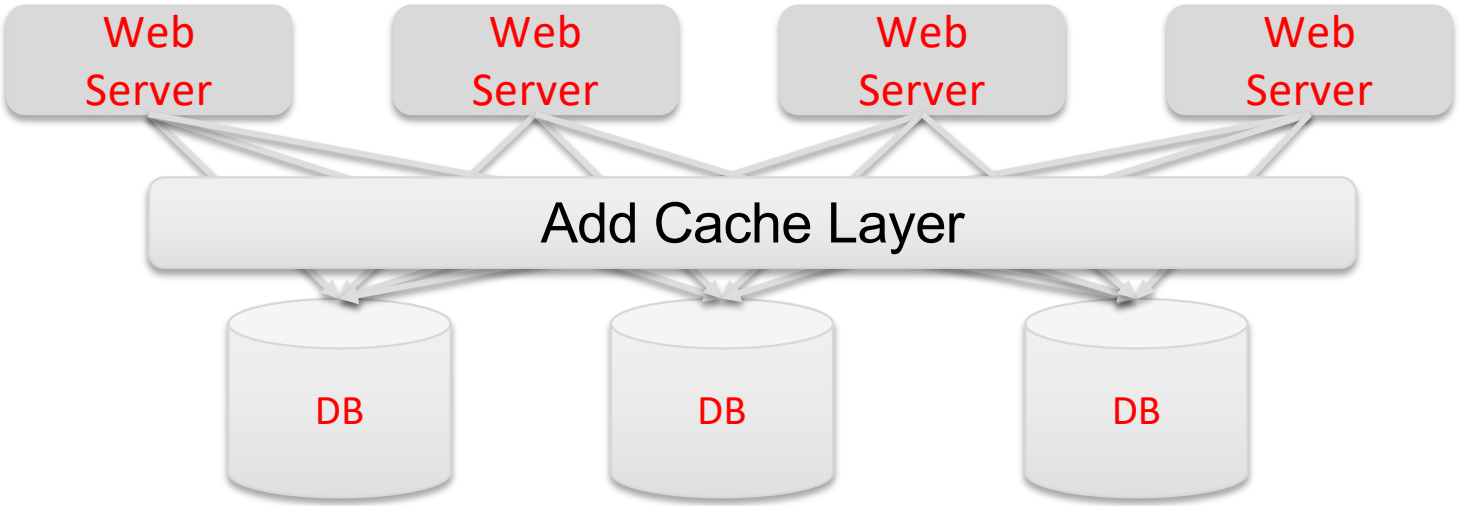
- **Low latency responses**
- **Multiple source content aggregation on the fly**
- **Some content is very popular, access and update this very fast**
- **Scalability for deterministic response time for millions of requests per second**
  - Billion reads very common
  - Should not hurt backend services
- **Geographic diversity**
  - Many datacenters

Memcache



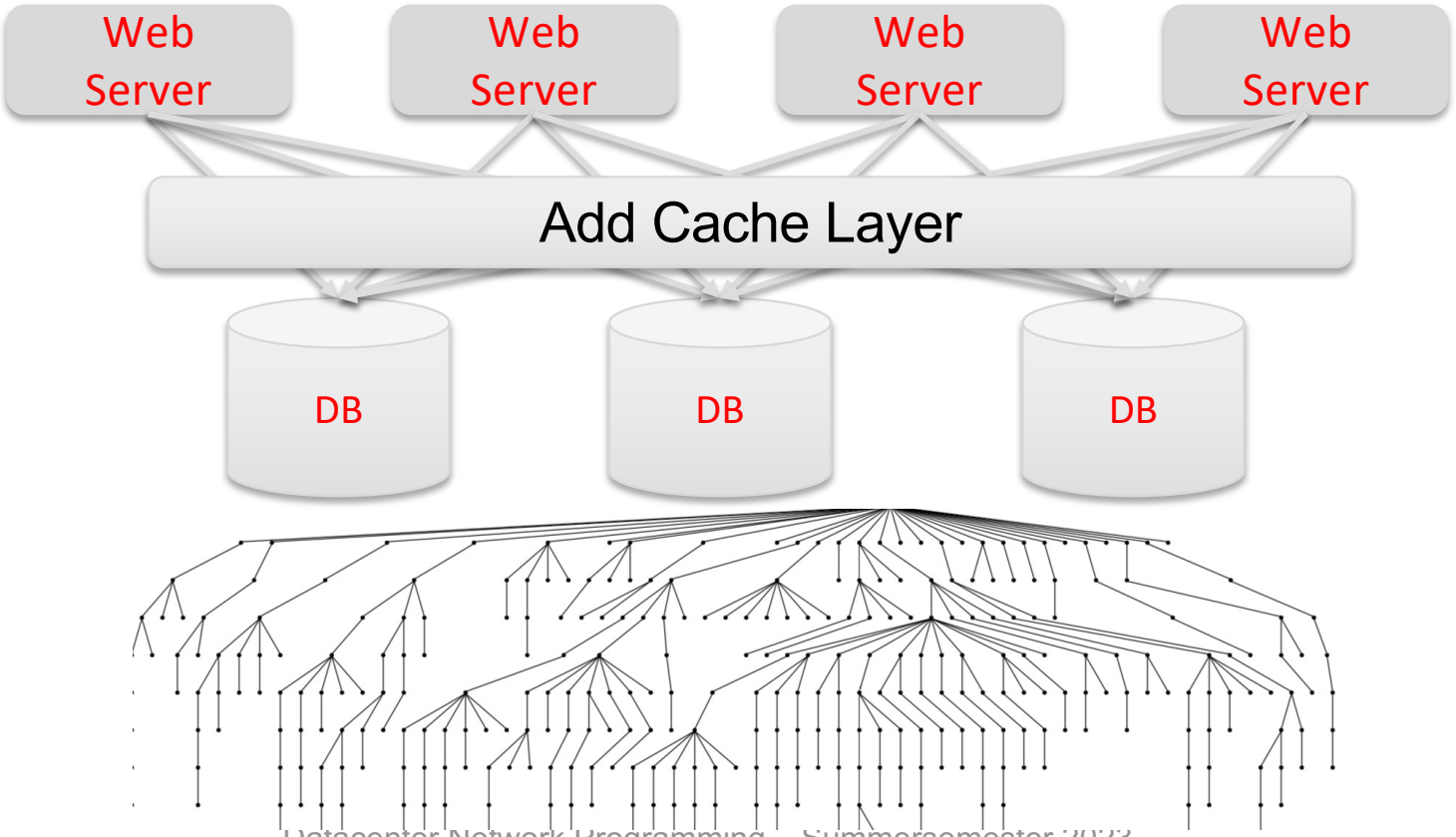
Data needs to be sharded (partitioned) across DBs for efficient access. Once access pattern changes that may be an issue

Memcache



Data needs to be sharded (partitioned) across DBs for efficient access. Once access pattern changes that may be an issue

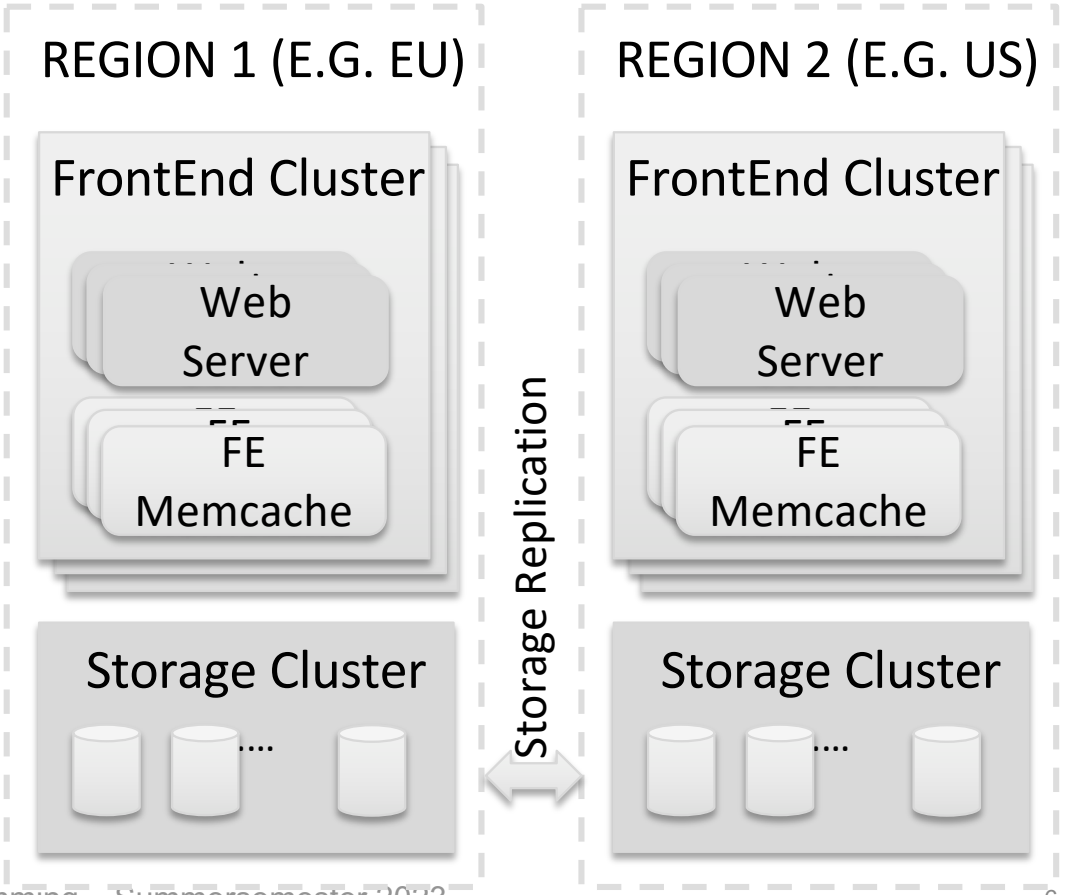
Memcache



Partition/ aggregate computing model results in high fanout and multiple fetching rounds

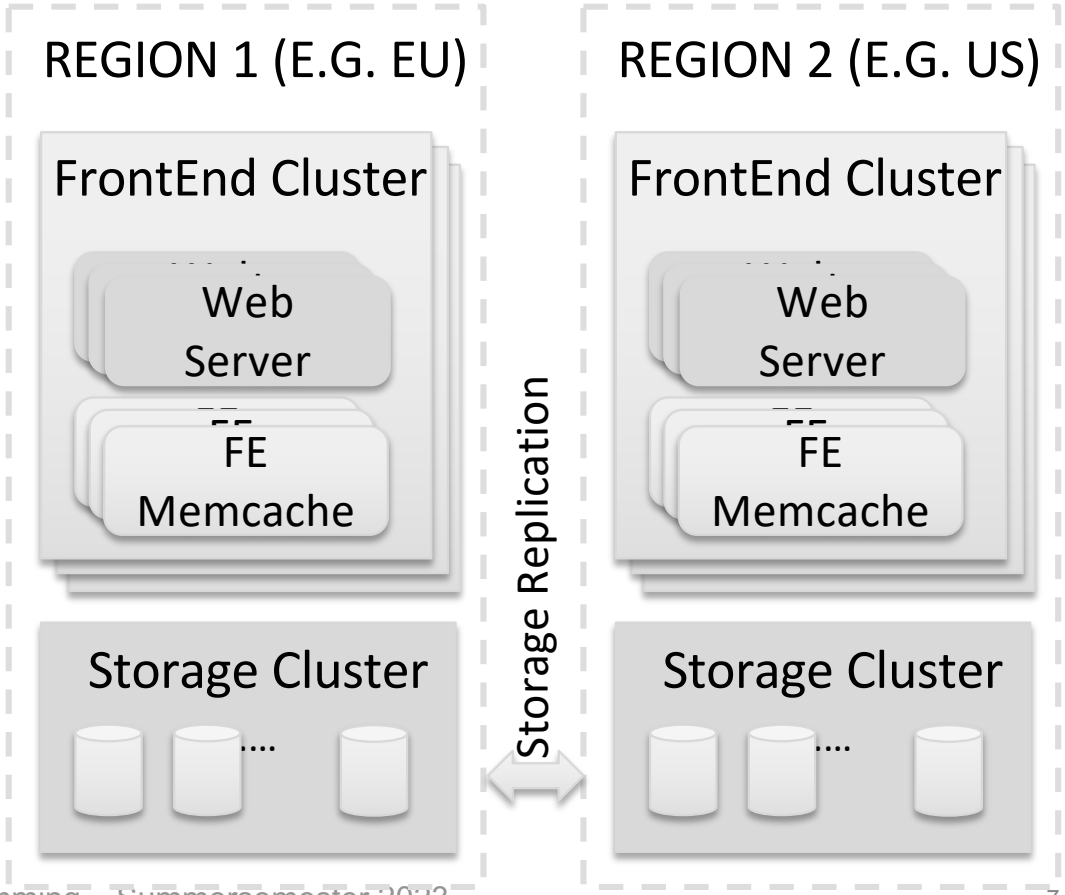
Memcached

- **Distributed Key Value Store**
  - E.g. Facebook:
    - Trillions of items
    - Billions requests per sec
  - But also for Google, or Cloud Native NFVs
  - In-Memory Hash Table
    - Attached to the network
    - LRU based eviction



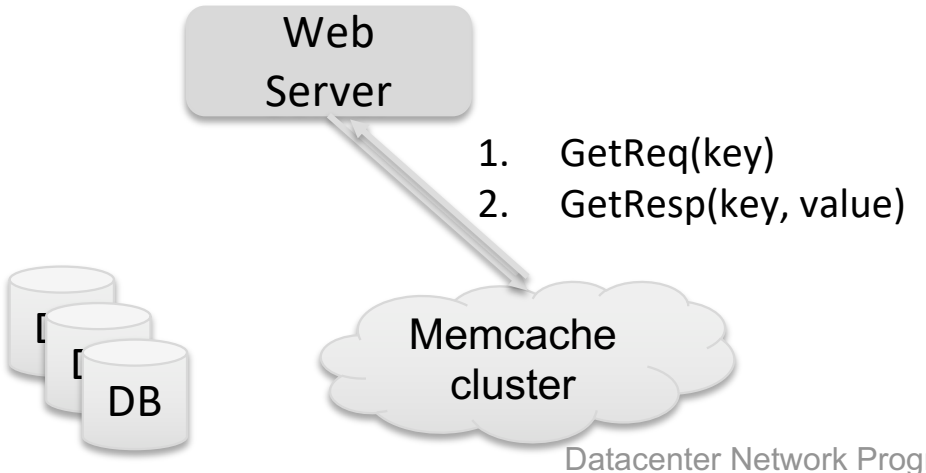
Memcached

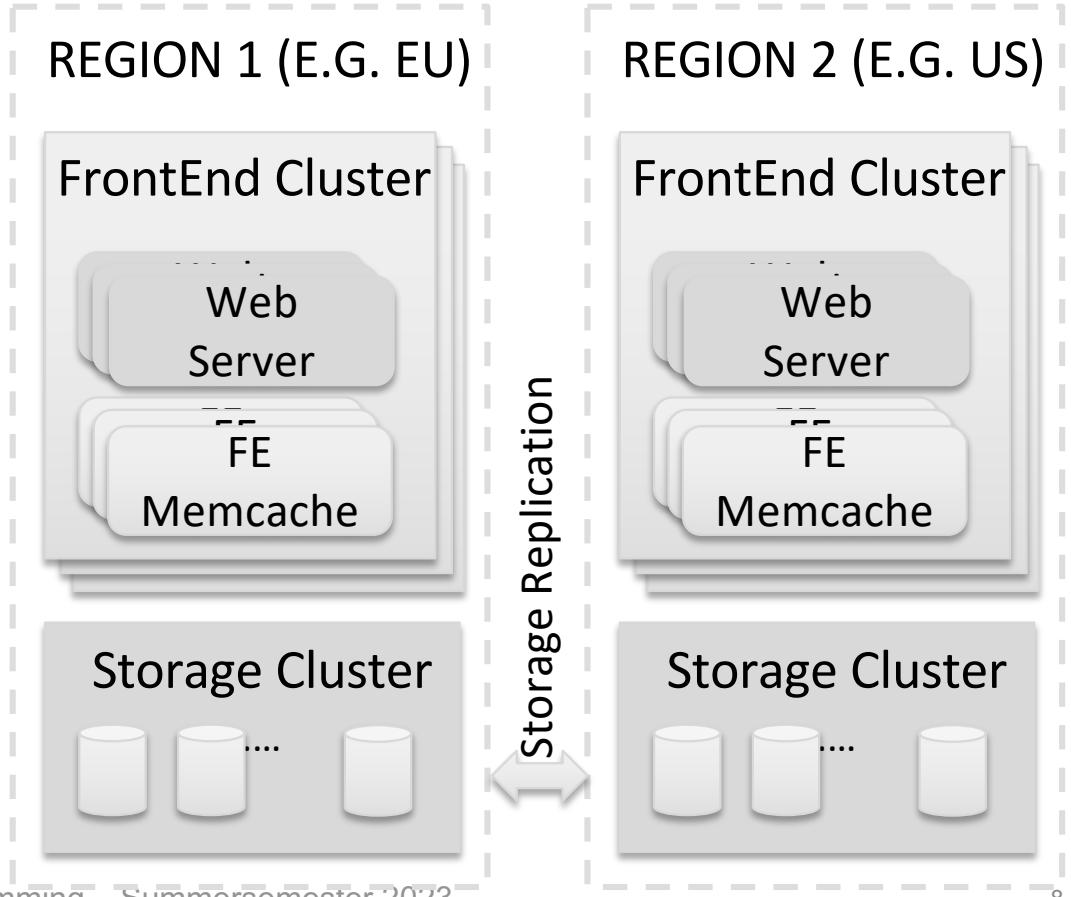
- **Single Front End Cluster**
  - Workload is read-heavy
  - Single Point of Failure
  - Wide Fanout
- **Multiple FE Clusters**
  - Data replication
  - Data consistency
- **Multiple GEO Regions**
  - Replication and consistency



How many memcached servers?

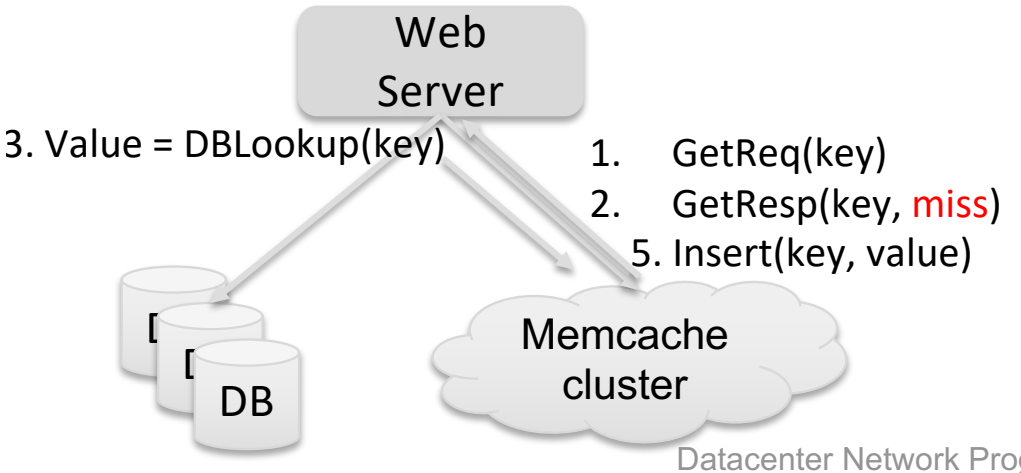
- **A few memcache servers**
- **Many servers in single cluster**
- **Many servers in multiple clusters**
- **Geographically distributed clusters**

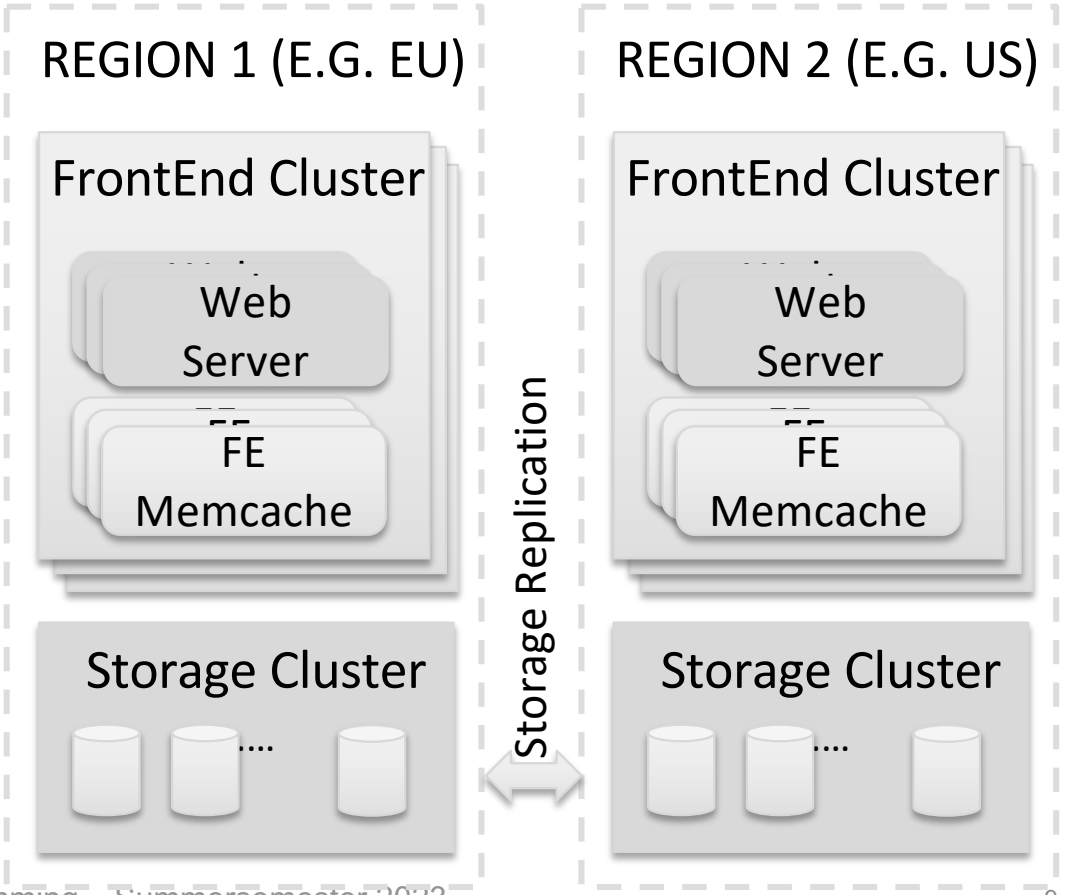




How many memcached servers?

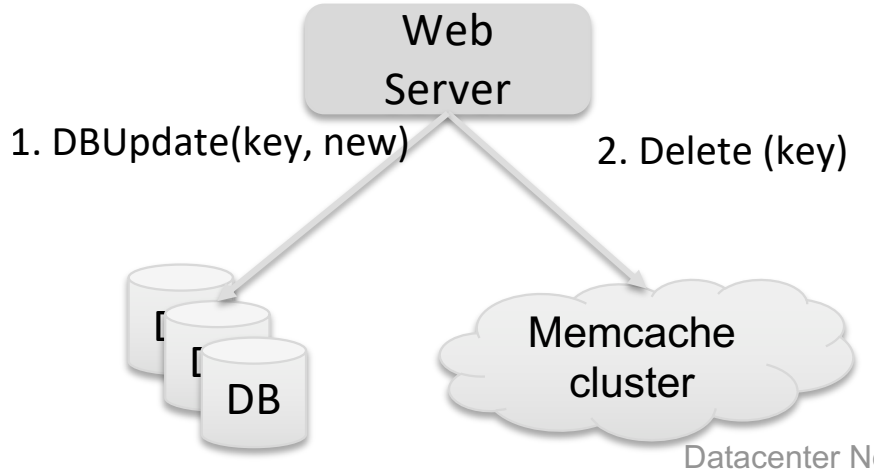
- **A few memcache servers**
- **Many servers in single cluster**
- **Many servers in multiple clusters**
- **Geographically distributed clusters**





Problems

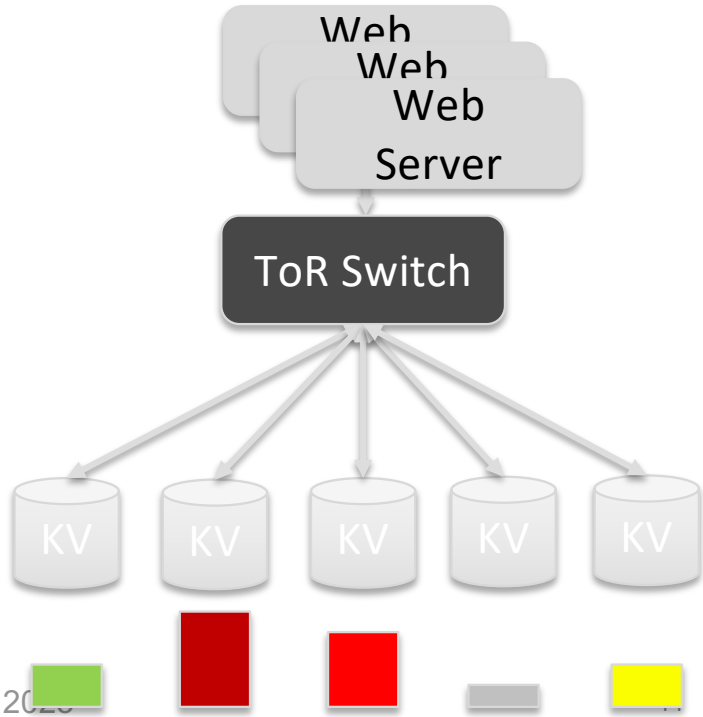
- **Two orders of magnitudes more reads than writes**
- **Updates need to invalidate cached entry**
- **For scalability: items need to be distributed across many memcache servers by using consistent hashing on key**
- **All Webservers talk to all memcached servers → Accessing 100s memcached servers for serving single request is common → Incast Problem when responses come back**



NetCache: Can we cache items in the dataplane to eliminate the need for 1000s memcache servers? In P4 enabled switches?

Netcache

- **What is NetCache**
  - Leveraging P4 and programmable data planes for in-network data plane caching
  - Rack-scale key-value store
- **Characteristics**
  - Supports highly skewed and rapidly changing workload
  - Billions queries per sec at 10s microsec latency
- **Target Workload:**
  - Small objects
  - More reads than writes
  - Dynamic key popularity

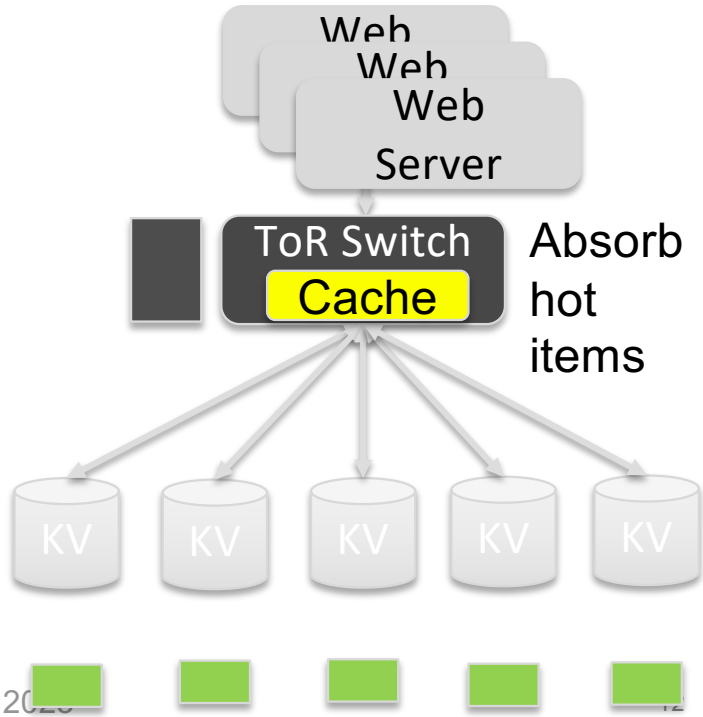


Skewed workload
→low throughput, high tail latency

Netcache

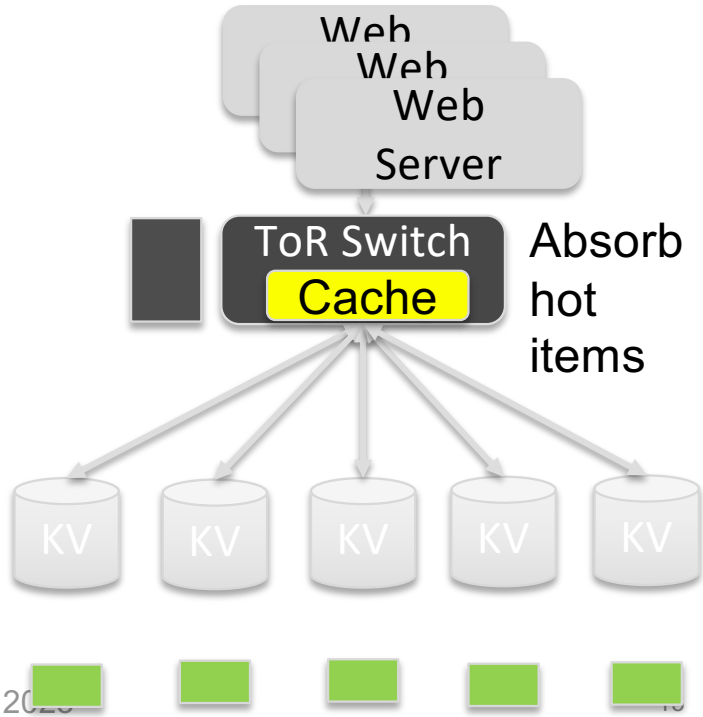
- **What is NetCache**
  - Leveraging P4 and programmable data planes for in-network data plane caching
  - Rack-scale key-value store
- **Characteristics**
  - Supports highly skewed and rapidly changing workload
  - Billions queries per sec at 10 microsec latency
- **Target Workload:**
  - Small objects
  - More reads than writes
  - Dynamic key popularity

even workload → High throughput, low tail latency



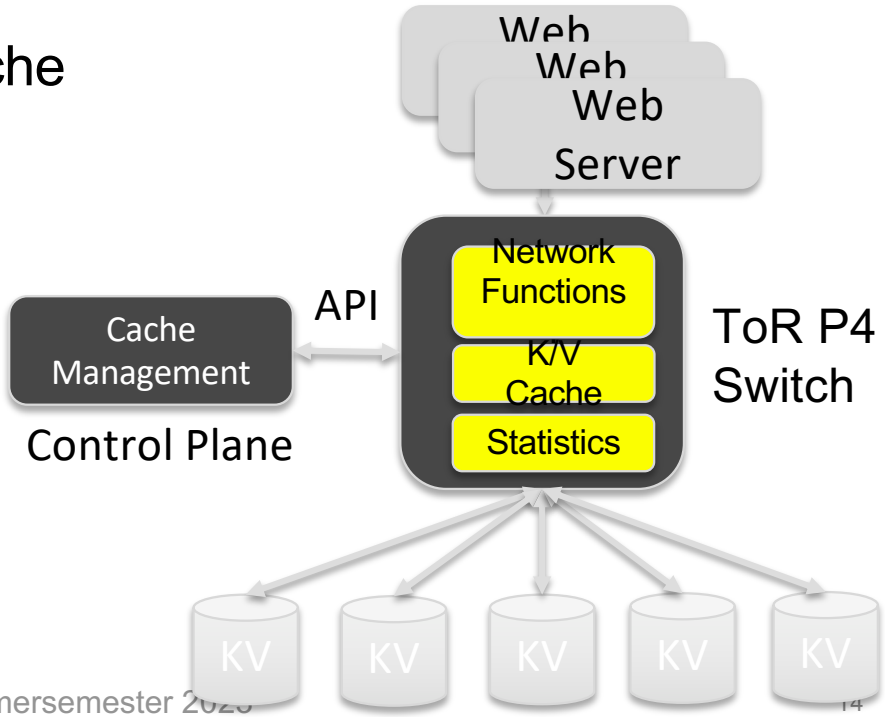
Design Requirements

- **Requirements**
  - Size: O(N log N), e.g. 10.000 for 100 billion items on 100 servers
  - Cache throughput >= backend aggregate throughput
    - E.g. SSD: each O(100) kQPS → O(10) mQPS (e.g. In-memory cache)
    - E.g. In-memory key/value (Redis, memcached) each O(10) mQPS → O(1) bQPS (e.g. In dataplane cache)
- **Key challenges**
  - Need to parse custom key/values in packets
  - Store and serve variable length values
  - Cache-update strategies
  - How to track hot items? → Query statistics



Architecture

- **Data Plane**
  - Programmable parser, parses requests and key/values
    - Wire protocol: OP, Seq, Key, Value
    - UDP for Read, TCP for Write //this is a problem as most KV are TCP
  - Maintains query statistics for efficient cache replacement
  - Use registers to store variable length key/value mappings
  - Use actions and tables to serve querys from local key/value cache
- **Control Plane**
  - Cache replacement strategy
    - Insert hot items, evict unpopular ones
  - Manage memory allocation



How to store Variable length values?

- **Use P4 Register Arrays**
- **Challenges:**
  - No loops, No strings
  - Compact representation required, minimize:
    - Table entries
    - Action data size
    - Metadata across table
- **Idea: Split value over multiple arrays**
  - Use bitmap to indicate which arrays
  - Use index to identify which slots in array

Pseudocode for fixed length

```
action process_array(idx):
  if pkt.op == read:
    pkt.value = array[idx]
elif pkt.op == cache_update:
  array[idx] = pkt.value
```

How to store Variable length values?

Example: splitting a 96 bit string into 3 parts, each fitting 32 bit register size → 3 Arrays → populated by CP

Lookup Table

Match	Pkt.key==A	Pkt.key==B	Pkt.key==C	Pkt.key==D
Action	Bitmap =111 Index=0	Bitmap = 110	Bitmap = 010	Bitmap = 101
Pkt.value	A0 A1 A2	B0 B1	C0	D0 D1

Where to insert new key into cache → Binpacking Problem

Value Table 0

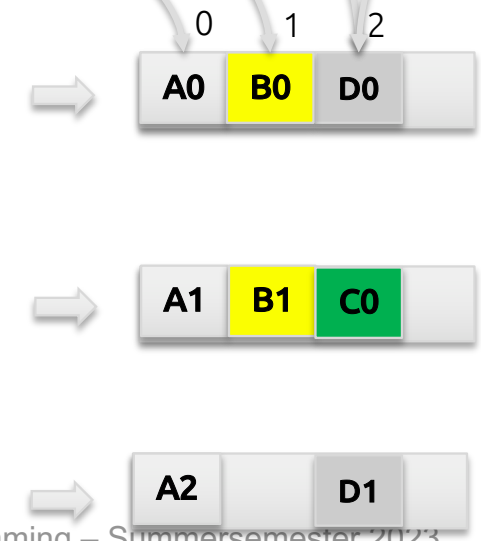
Match	Bitmap[0] == 1
Action	process_array_0 (index)

Value Table 1

Match	Bitmap[1] == 1
Action	process_array_1 (index)

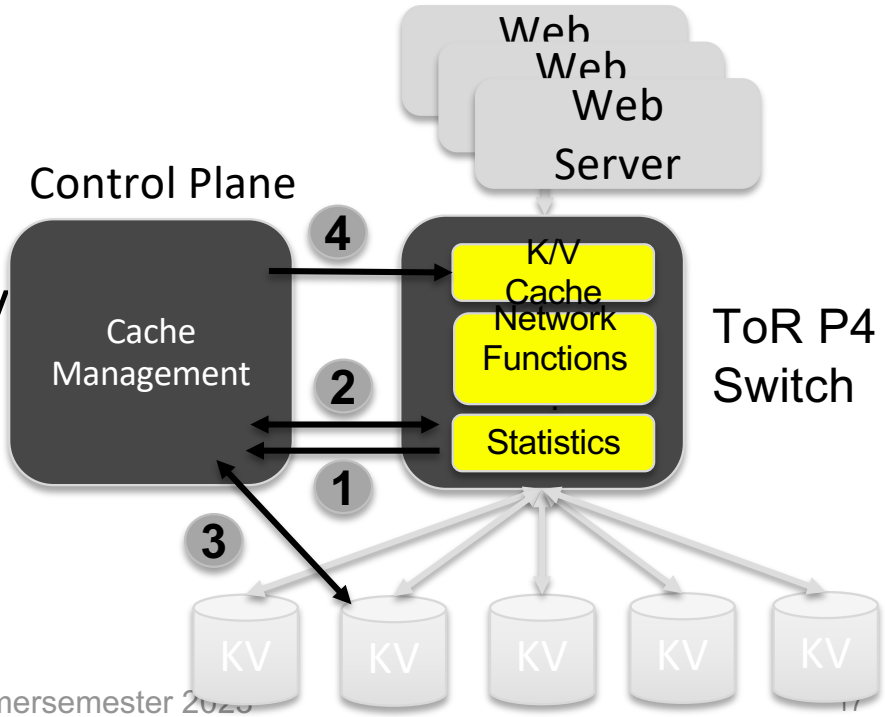
Value Table 2

Match	Bitmap[2] == 1
Action	process_array_2 (index)



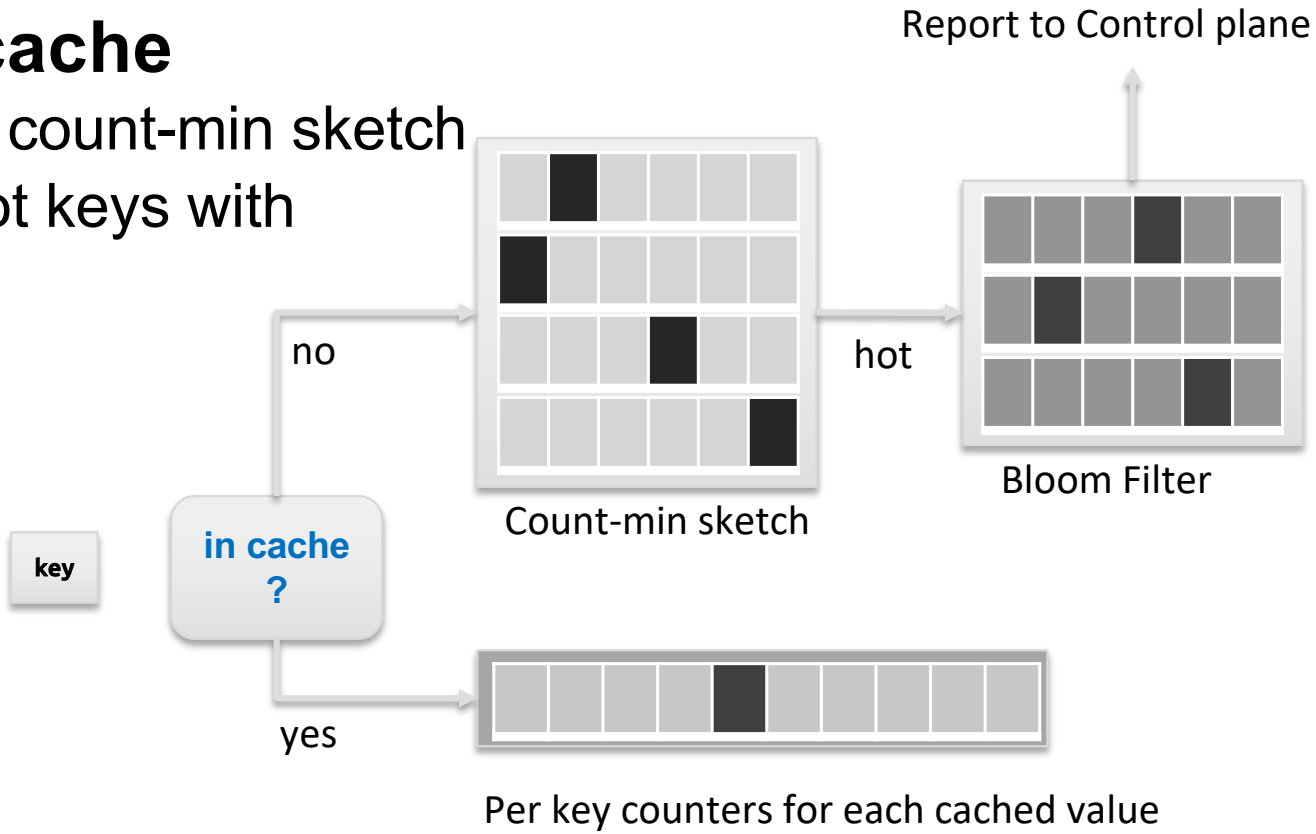
Cache Update

- **The cache needs to**
  - Track the hottest O(N log N) items with limited insertion rate
  - React quickly to changes in workload with minimal updates
- **How?**
  1. P4 data plane reports hot keys
  2. Control plane compares new hot and sampled cached keys
  3. Control plane fetches values for newly to be inserted keys
  4. Control plane inserts new keys and evicts old ones



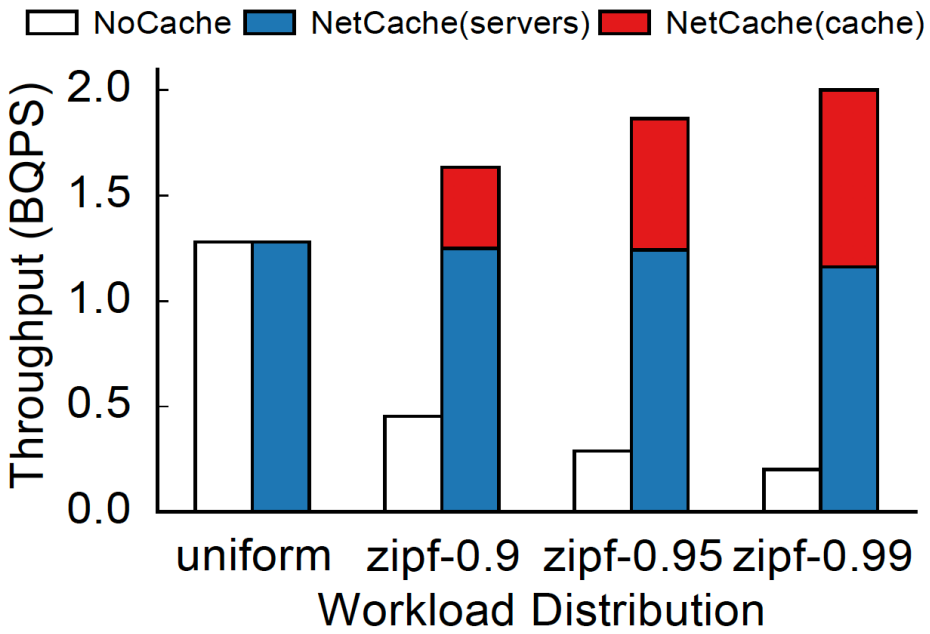
Cache Statistics

- **Per-key counter array for cached keys**
- **If the key is not in cache**
  - Report hot keys with count-min sketch
  - Remove duplicate hot keys with Bloom filter



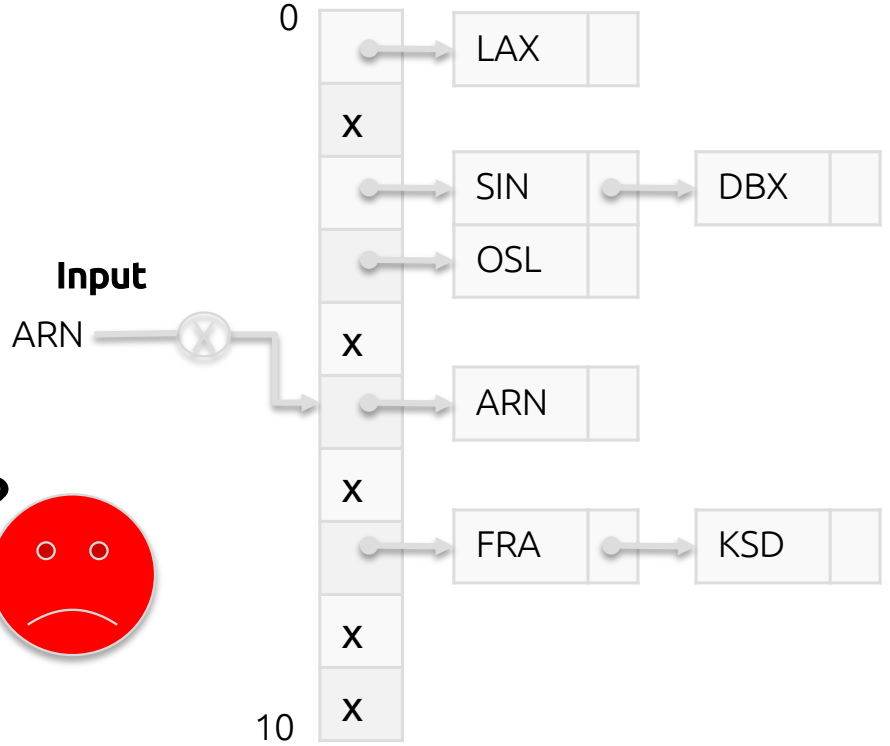
Implementation/Evaluation

- **Implemented on Barefoot Tofino**
  - 2K lines of code
  - Cache 64 kItems, 16 byte key, 128 byte value
  - 128 x 16 core xeon servers, 40 Gbps NIC
    - TommyDS
    - 10 mQPS, 7 microsec
- **Takeaway**
  - 3…10 performance improvement
  - Reacts to wide range of workload dynamics



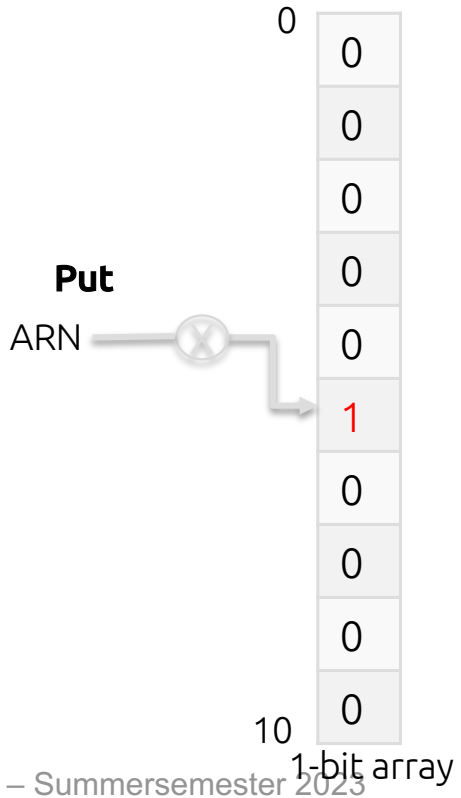
Probabilistic Datastructures

- **For counting not cached values, need to implement something like a set: set of non-cached keys**
- **Can use hash function**
  - Need to deal with hash-collisions
  - Typically use linked lists
  - Non-deterministic runtime
  - Deterministic output and fast
  - Listsize for N elements and M cells
    - Average: N/M
    - Worst case: N
- **Feasible to be implemented in P4?**
  - Only works for small N, Need for loops!
  - Not suited for NetCache



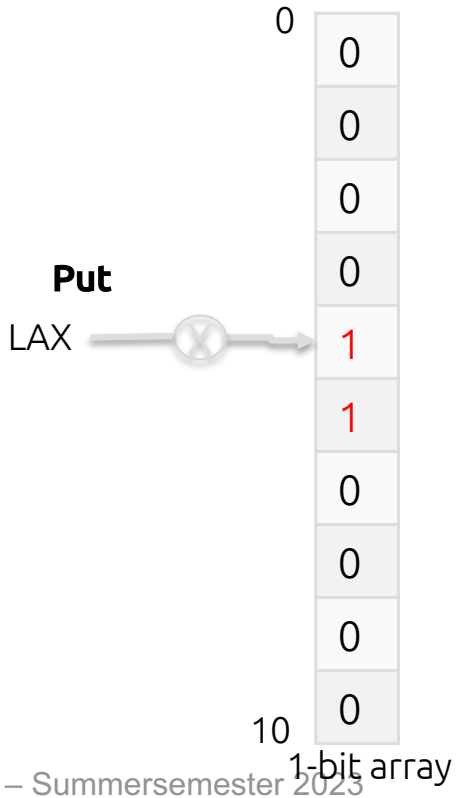
Membership Queries

- **Use 1-bit array to insert item and record/query for membership**



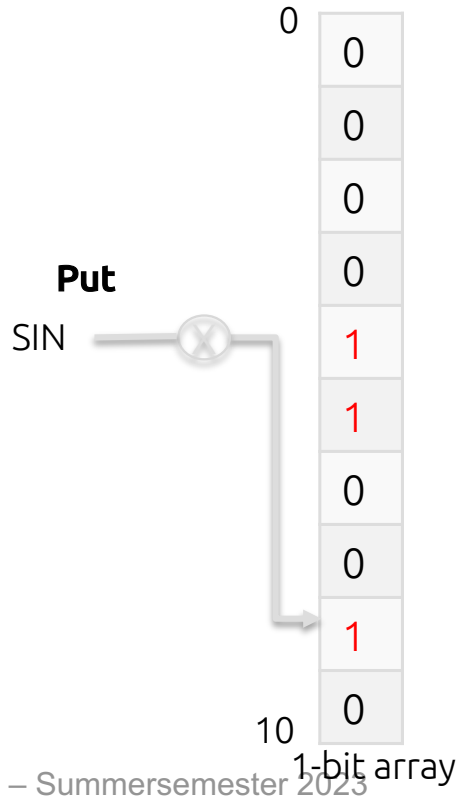
Membership Queries

- **Use 1-bit array to insert item and record/query for membership**



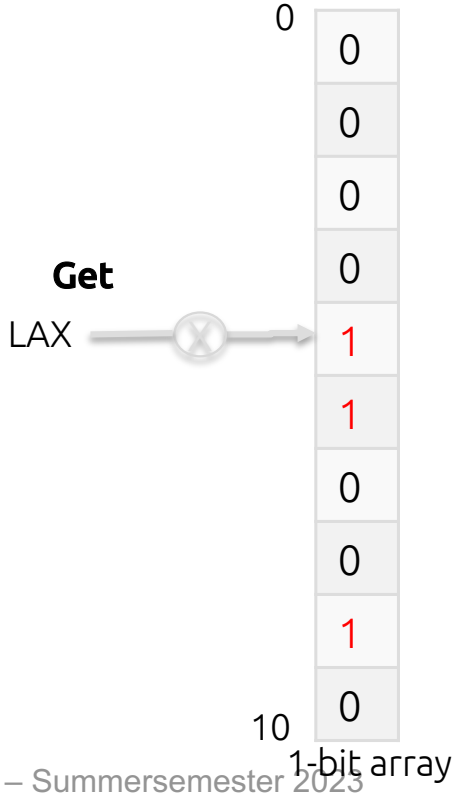
Membership Queries

- **Use 1-bit array to insert item and record/query for membership**



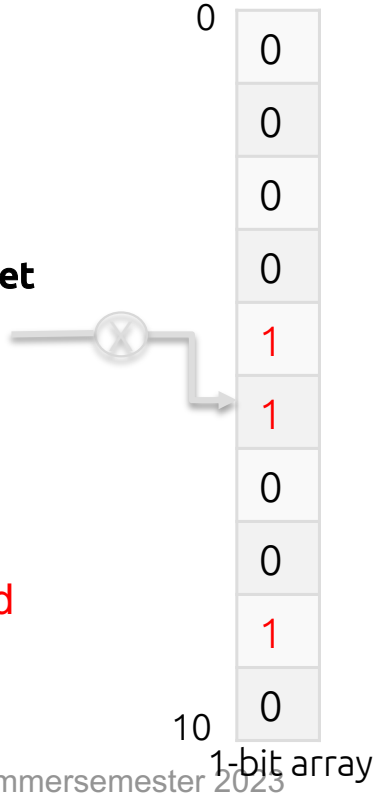
Membership Queries

- Use 1-bit array to insert item and record/query for membership
- Get returns 1, if hit



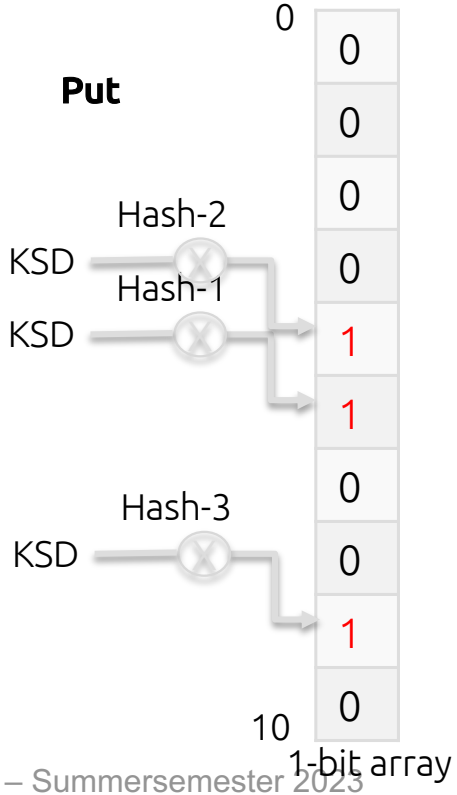
Membership Queries

- **Use 1-bit array to insert item and record/query for membership**
- **Get returns 1, if hit**
- **Problem:**
  - Collisions
  - Probability for false positives
    - assume N elements and M cells (here M=10)
    - P(False Negative)=0, why?
    - P(element to be mapped to cell x) = 1/M
    - P(element not to be mapped to cell x) = 1-1/M
    - P(cell==0)= $(1-1/M)^N$
    - P(False positive)= $1-(1-1/M)^N$
    - N=1000, M = 100000: P(False positive) = 1%
    - Need 100 times more cells than elements and still high FPR



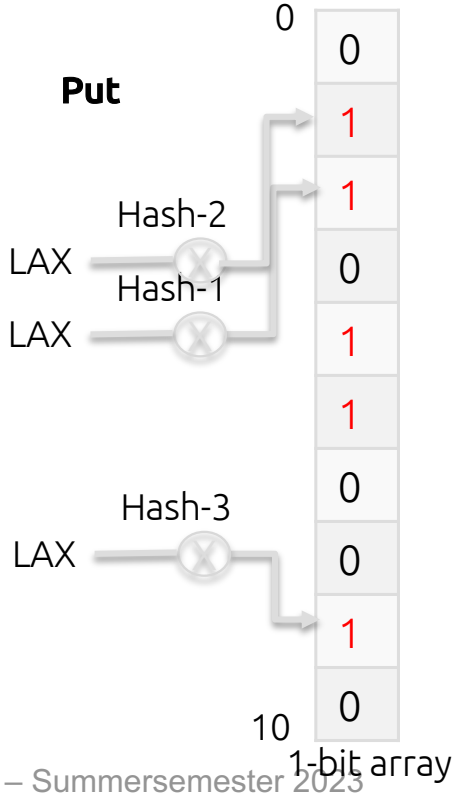
Bloom Filter

- **Use 1-bit array to insert item and record/query for membership of an item in a set of items**
- **Instead 1, use 3 (k?) different hash funtions**



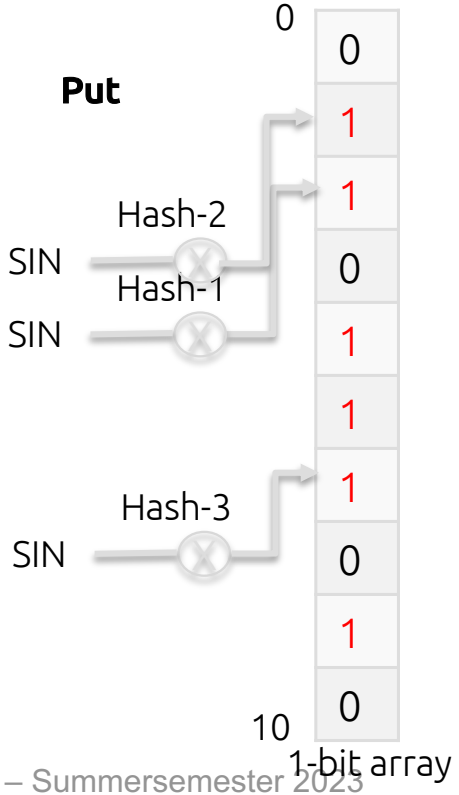
Bloom Filter

- **Use 1-bit array to insert item and record/query for membership of an item in a set of items**
- **Instead 1, use 3 different hash funtions**



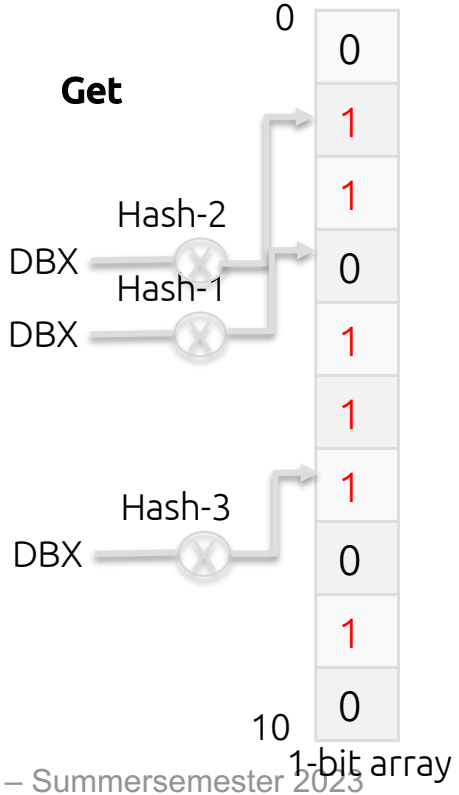
Bloom Filter

- **Use 1-bit array to insert item and record/query for membership of an item in a set of items**
- **Instead 1, use 3 different hash funtions**
- **Query:**
  - Element in set if all hash values map to cell ==1
  - Element is NOT in set if at least one hash value returns 0



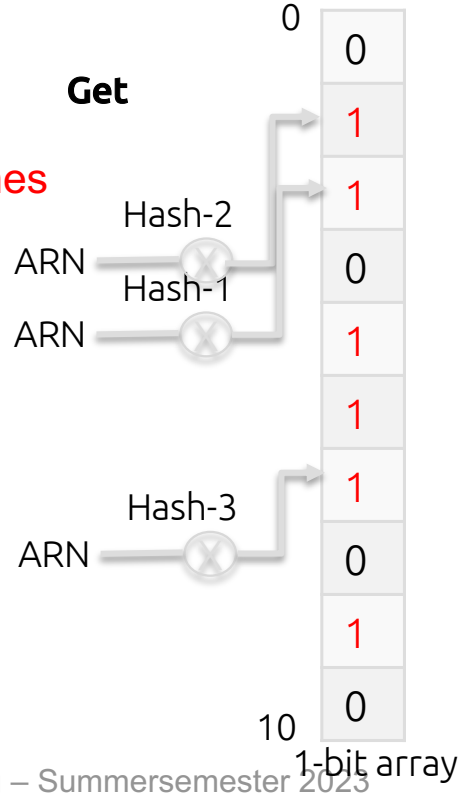
Bloom Filter

- **Use 1-bit array to insert item and record/query for membership of an item in a set of items**
- **Instead 1, use 3 different hash funtions**
- **Query:**
  - Element in set if all hash values map to cell ==1
  - Element is NOT in set if at least one hash value returns 0
    - DBX not in set



Bloom Filter

- **Use 1-bit array to insert item and record/query for membership of an item in a set of items**
- **Instead 1, use 3 different hash funtions**
- **Query:**
  - Probability for false positives
    - assume N elements and M cells, H Hashes
    - P(False Negative)=0
    - P(element to be mapped to cell x) = 1/M
    - P(element not to be mapped to cell x) = 1-1/M
    - $P(cell==0)= (1-1/M)^{HN}$
    - $P(False\ positive)= (1-(1-1/M)^{HN})^{H}$
    - N=1000, M = 100000, H=7: P(False positive) = 0%
- **Trade-off compute for storage:**
  - Need 7 times hash computation



Bloom Filter

- **For given FPR, can trade-off memory space and compute**
- **assume N elements and M cells, H Hashes**
  - **p**=P (False positive)= $(1-(1-1/M)^{HN})^{H}$ can be approximated with $(1-e^{-HN/M})^{H}$ →MIN!
  - Min H = ln2*(M/N)
  - More: https://hur.st/bloomfilter/?n=1000000000&p=0.001&m=1000000&k=

Bloom filters are space-efficient probablistic data structures used to test whether an element is a member of a set.
If all bits are set, the element *probably* already exists, with a false positive rate of **p**; if any of the bits are not set, the element *certainly* does not exist.

Bloom Filter in P4

- **Can we implement Bloom filter in P4 using hash functions and registers?**
- **V1 model code**

```
enum HashAlgorithm {
        crc32,
        crc32_custom,
        crc16,
        s,
        random,
        identity,
        csum16,
        xor16
}
```

```
extern register<T> {
        register(bit<32> size);

        void read(out T result, in bit<32> index);
        void write(in bit<32> index, in T value);
}
```

```
extern void hash<O, T, D, M>(out O result,
        in HashAlgorithm algo, in T base,
        in D data, in M max);
```

https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

Bloom Filter in P4

```
/***
 * Calculate a hash function of the value specified by the data
 * Parameter.  The value written to the out parameter named result
 * Will always be in the range [base, base+max-1] inclusive, if max >=
 * 1.  If max=0, the value written to result will always be base.
 *
 * Note that the types of all of the parameters may be the same as, or
 * Different from, each other, and thus their bit widths are allowed
 * To be different.
 *
 * @Param o          must be a type bit<w>
 * @Param D          must be a tuple type where all the fields are bit-fields (type bit<w> or int<w>) or varbits.
 * @Param t          must be a type bit<w>
 * @Param M          must be a type bit<w>
 */
@Pure
extern void hash<o, T, D, m>(out O result, in hashalgorithm algo, in T base, in D data, in M max);
```

```
extern void hash<O, T, D, M>(out O result,
                             in HashAlgorithm algo, in T base,
                             in D data, in M max);
```

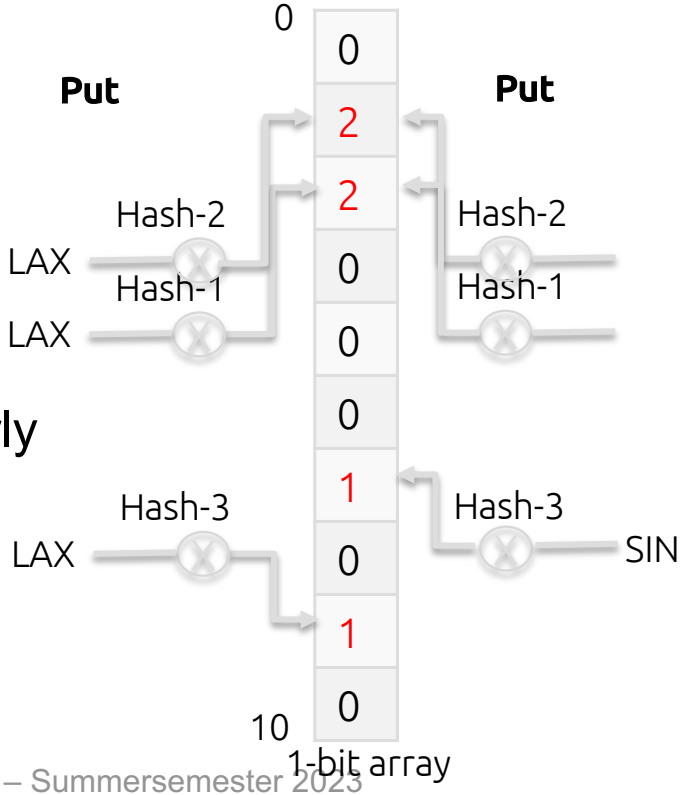
https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

Bloom Filter in P4

```
control MyIngress(…) {
  register  <bit<1>>(M) bloom; //record unique IP src-dstprefix pairs
              apply {
              //void hash(out O result, in hashalgorithm algo, in T base, in D data, in M max);
                          hash(meta.result1, crc16, 0, {meta.dstPrefix, packet.ip.srcIP}, M);
                          hash(meta.result2, crc32, 0, {meta.dstPrefix, packet.ip.srcIP}, M);
                          if (meta.to_insert == 1) {
                                      bloom.write(meta.result1, 1);
                                      bloom.write(meta.result2, 1);
                          }
                          if (meta.to_query == 1) {
                                      bloom.read(meta.get1, meta.result1);
                                      bloom.read(meta.get2, meta.result2);
                                      if (meta.get1 == 0 || meta.get2 == 0) {
                                                  meta.is_inList = 0;
                                      }
                                      else {
                                                  meta.is_inList = 1;
              }
}}}
```

Deletions?

- **Handling deletions with resetting (writing 0)?**
- **Deleting SIN will also delete LAX (why?)**
- **Use counting bloom filter instead**
  - Instead of using array of 1-bit cells, use e.g. 4 bit cells
  - To insert element, increment counter
  - To remove an element, decrement counter
  - Uses more memory
  - Counters must be dimensioned properly to avoid overflow



Counting Bloom Filter in P4

```
control MyIngress(…) {
  register  <bit<32>>(M) bloom; //add unique IP src-dst prefix pairs
              apply {
                        hash(meta.result1, crc16, 0, {meta.dstPrefix, packet.ip.srcIP}, M);
                        hash(meta.result2, crc32, 0, {meta.dstPrefix, packet.ip.srcIP}, M);

                        //check if element in the set
                        bloom.read(meta.get1, meta.result1);
                        bloom.read(meta.get2, meta.result2);

                        if (meta.get1 == 0 || meta.get2 == 0) { // not in set: add it
                                  bloom.write(meta.result1, meta.get1 + 1);
                                  bloom.write(meta.result2, meta.get2 + 1);
                        }
              }
}
```

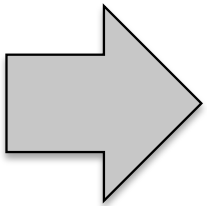
Counting Bloom Filter in P4

```
control MyIngress(…) {
  register  <bit<32>>(M) bloom; //remove unique IP src-dst prefix pairs
              apply {
                        hash(meta.result1, crc16, 0, {meta.dstPrefix, packet.ip.srcIP}, M);
                        hash(meta.result2, crc32, 0, {meta.dstPrefix, packet.ip.srcIP}, M);

                        //delete only if element in the set
                        bloom.read(meta.get1, meta.result1);
                        bloom.read(meta.get2, meta.result2);

                        if (meta.get1 > 0 && meta.get2 > 0) { // in set: remove it
                                  bloom.write(meta.result1, meta.get1 - 1);
                                  bloom.write(meta.result2, meta.get2 - 1);
                        }
              }
}
```

Sketches

Bloom filters and Counting Bloom Filters are streaming algorithms answering specific questions approximately
-"Have I seen this key in my stream?"
-"What keys are in my stream?"

How about counting instances → Sketches
-"How frequent have been key lookups for X?"
-"What are the hottest keys?"



How often does an element appear in a stream → Streamprocessing

- Packet flows
- Search engines
- Databases
- IoT
- key value stores
- ….

Count-Min Sketch: Counting frequencies

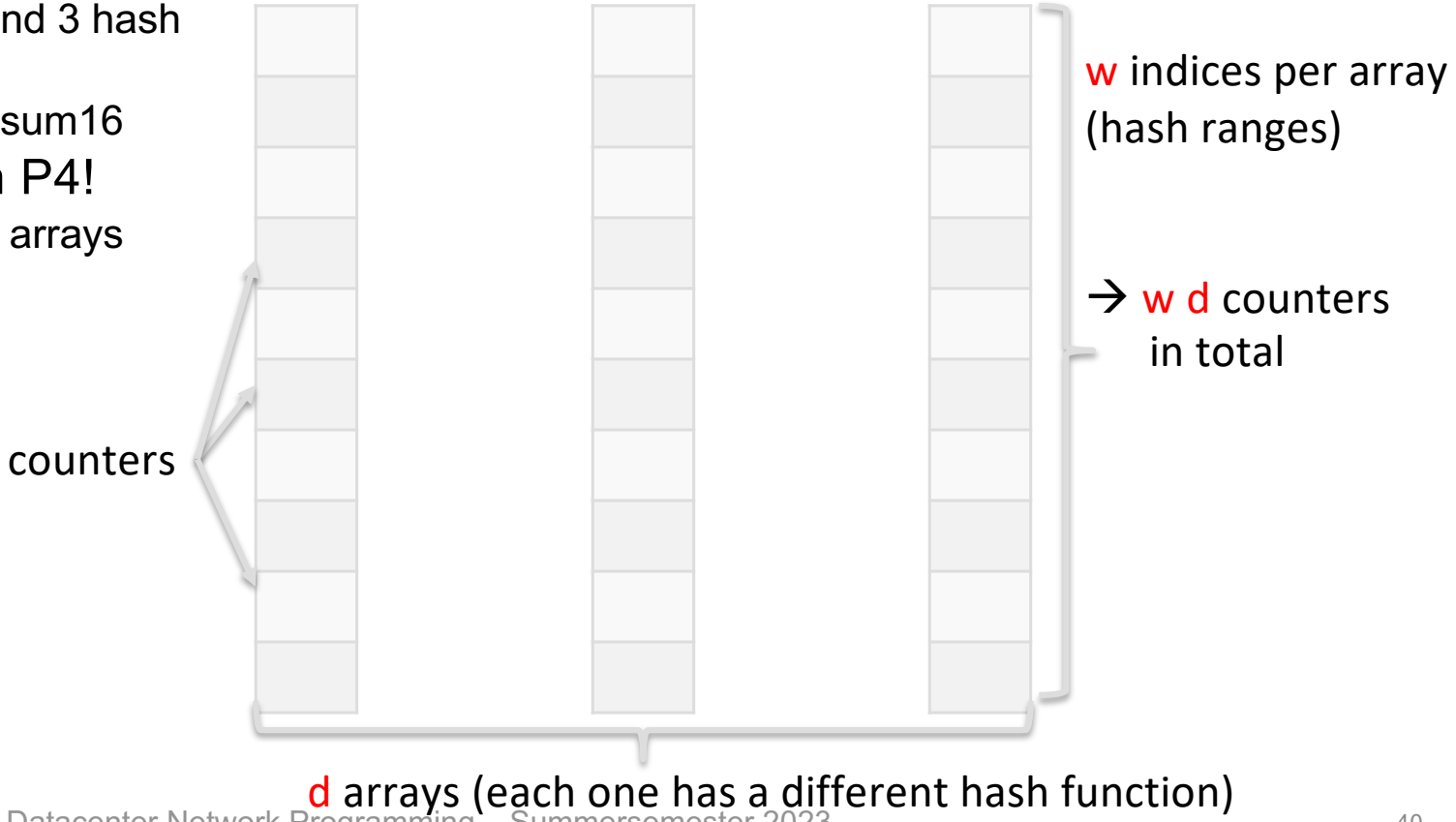
- **Vector of frequencies of all distinct elements x_i**
- **Worst case: for exact counting requires linear space to store**
  - Data stream has n distinct elements (no duplicates)
  - Requires n counters? Recall Bloom filter
    - Allow small set of false positives
    - Quickly filter only on those elements that might be in the set

$$\begin{pmatrix} X_1 \\ X_2 \\ .. \end{pmatrix}$$

- **Sketches:**
  - Approximate frequencies of data stream elements, allow small miscounting to save space
  - CountMin sketch: similar to counting Bloom filter but is designed to have known error bounds for frequency queries
    - Estimation error exceeds $\varepsilon \|x\|_1$ with a probability smaller than δ
    - For ε=0.01, $\|x\|_1$ = 100.000, δ = 0.05 → Probability for any estimate to be off by more than 1000 is < 5%

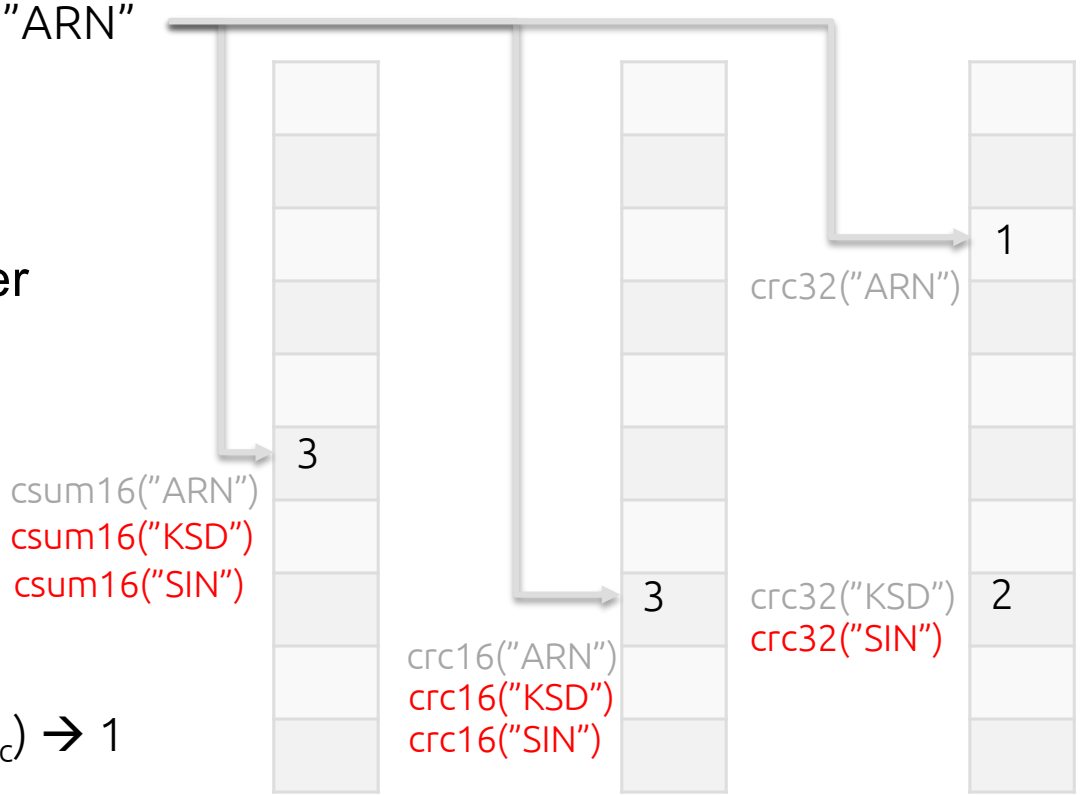
Count-Min Sketch: Counting frequencies

- **Design:**
  - use multiple arrays and hashes
    - E.g. 3 arrays and 3 hash functions:
      crc16, crc32, csum16
  - can implement in P4!
    - In Netcache: 4 arrays



Count-Min Sketch: Counting frequencies

- **Problem:**
  - Hash collisions
    - Over-count
- **Solution:**
  - Return the minimum
  - To minimize errors, proper dimensioning required



Get "ARN"
return min (x_a, x_b, x_c) → 1

Get "KSD"
return min (x_a, x_b, x_c) → 2 → Error!

Count-Min Sketch: Dimensioning

- **Recipe: (related work, wikipedia,…)**
  - Choose $d = \left\lceil log_2 \frac{1}{\delta} \right\rceil$ number of hash functions and $w = \left\lceil \frac{2}{\varepsilon} \right\rceil$ cells
  - Then $\hat{x}_i - x_i \geq \varepsilon \|\chi\|_1 \; with \; probability \; less \; than \; \delta$
- **Example:**
  - $\varepsilon = 0.01, \|\chi\|_1 = 10.000 \;\; then \; \varepsilon \|\chi\|_1 = 100$ → 200 cells, 4 hash functions for $\delta = 0.0625$ → 4 register arrays, each one 200 entries
  - Assume two flows x_a, x_b with $\|\chi\|_a = 1000, \|\chi\|_b = 50$
  - Error relative to total stream size is 1%, relative to flow a: 10%, relative to flow b: 200%
- **Conclusion:**
  - Sketches compute statistical summaries and perform better for high frequency elements
  - Sketches trade-off memory/compute usage with error margins with guarantees

http://web.stanford.edu/class/cs369g/files/lectures/lec7.pdf

Conclusion

- **P4 powerful tool for**
  - Data plane caching
  - Speeding up key value stores which are at the backend of modern data centers
- **Can implement probabilistic data structures in P4**
  - Bloom filter
  - Count-min sketch
  - Those are good for stream processing, e.g. Heavy-hitter, with many other applications including Bitcoin clients, malicious web site detection and in CDNs