SDC
STORAGE DEVELOPER CONFERENCE
SNIA ■ SANTA CLARA, 2015

New Hashing Algorithms for Data Storage

Jason Resch

Cleversafe

Applications of Hashing

- Hashing is useful generally:
  - Provides O(1) lookup
  - Key → Value storage/retrieval
- Could use hashing to decide…
  - Storage node in storage system or database
  - Proxy server that has a cache
  - Task assignment in distributed computing

Hashing in Distributed Systems

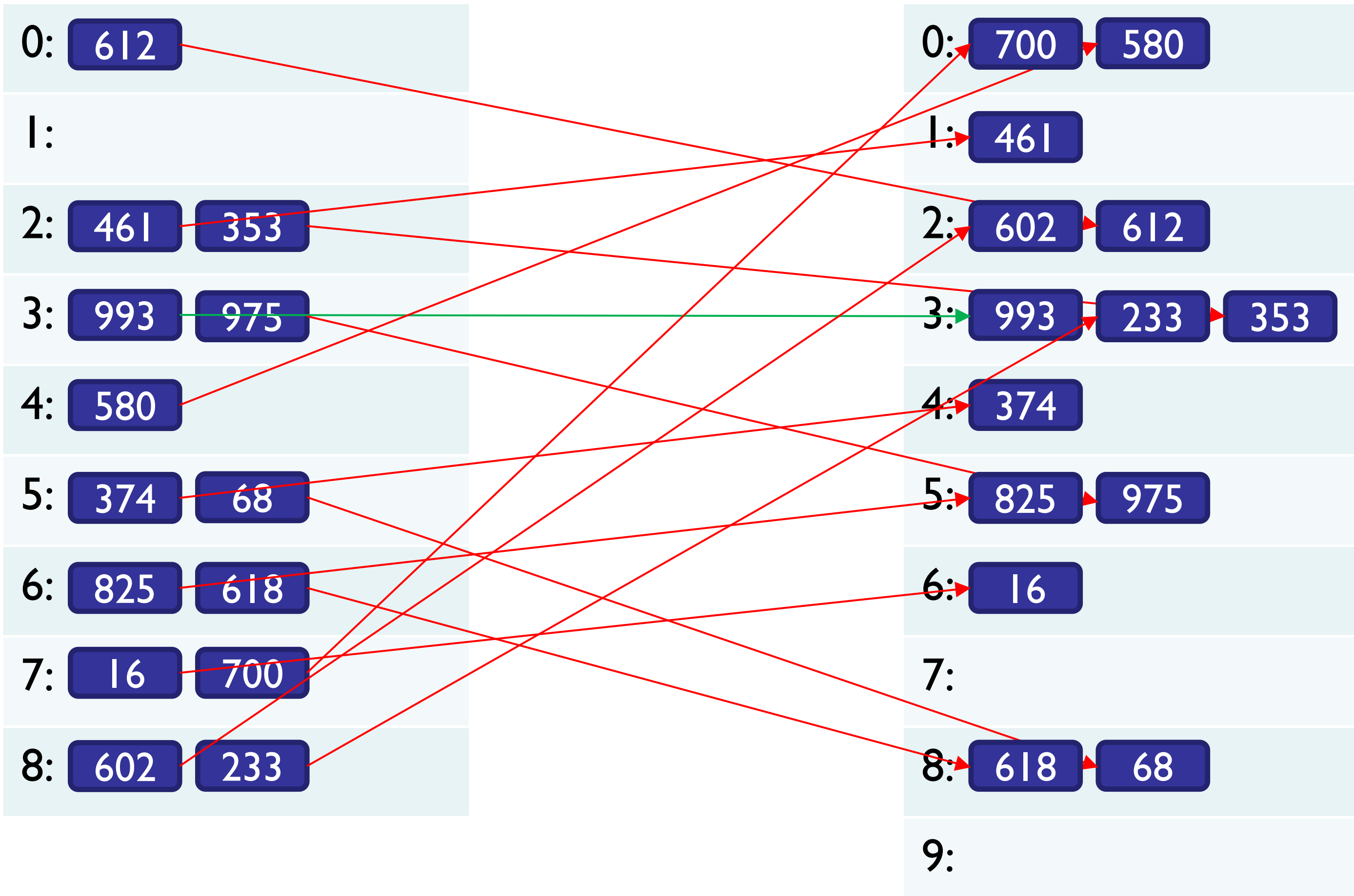
- Distributed Storage
  - If buckets are “storage nodes”, we can use hashing so readers and writers select the same storage locations for the same names
- Distributed Caching
  - If buckets are “caching servers”, we can use hashing to maximize reuse of the same caching servers for the same URLs

Conventional Hash Table Resize

0:	612	
1:		
2:	461	353
3:	993	975
4:	580	
5:	374	68
6:	825	618
7:	16	700
8:	602	233

bucket = hash(key) % num_buckets

Conventional Hash Table Resize



bucket = hash(key) % num_buckets

Stable Hashing Defined

- When a conventional Hash Table is resized, most keys are remapped to different buckets
  - bucket = hash(key) % num_buckets
  - Almost all keys move if num_bucket changes

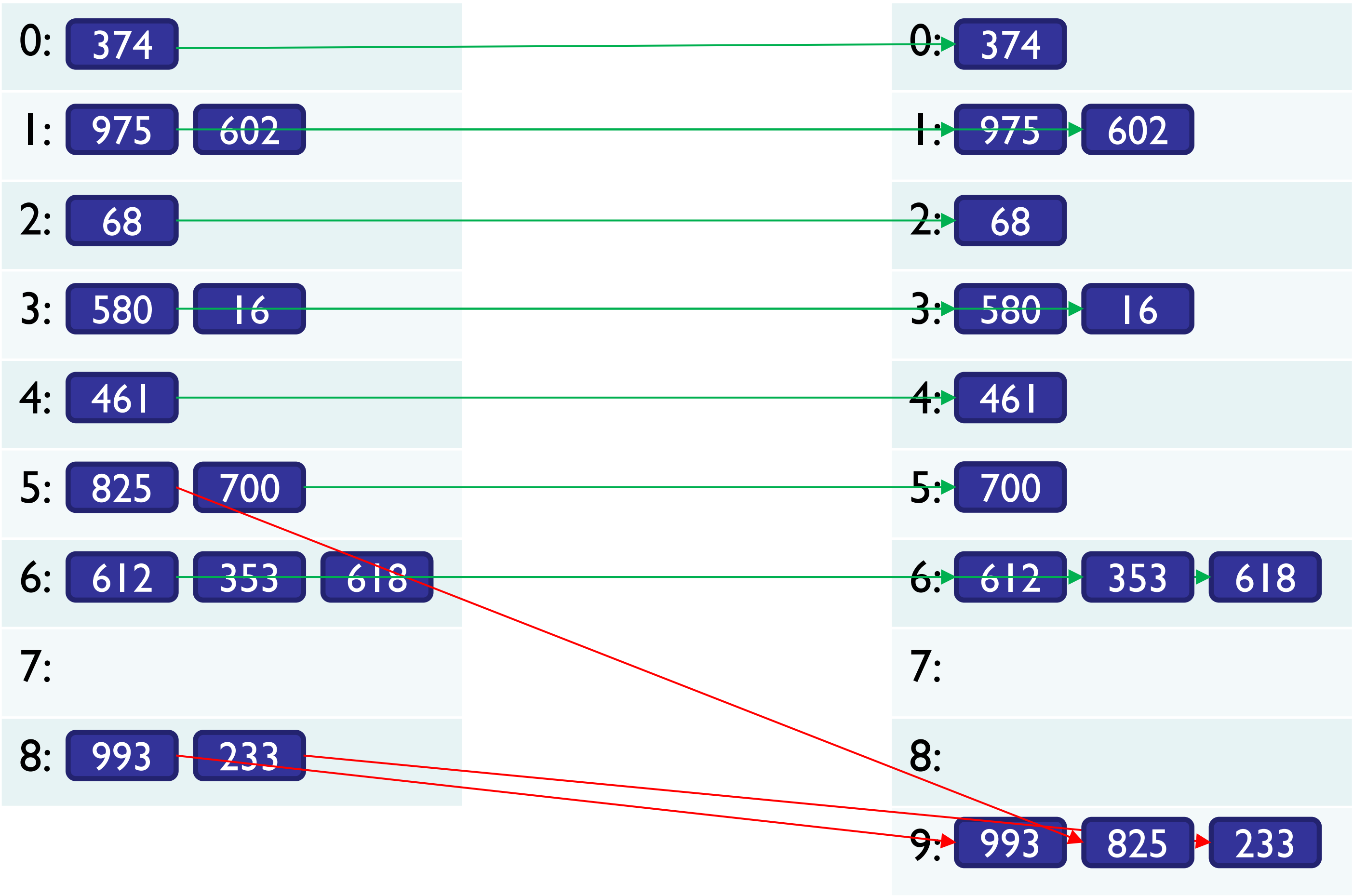
- Stable Hashing
  - Enables Hash Tables with greater stability
  - Minimizes disruption when resizing/scaling

Stable Hashing Resize

0:	374		
1:	975	602	
2:	68		
3:	580	16	
4:	461		
5:	825	700	
6:	612	353	618
7:			
8:	993	233	

bucket = stable_hash(buckets, key)

Stable Hashing Resize



bucket = stable_hash(buckets, key)

Who uses Stable Hashing?

- Caching/Routing:

Microsoft Proxy Server

Squid

- DHT/Storage:

Gluster

amazon DynamoDB

Cassandra

ceph

openstack CLOUD SOFTWARE

When is Stable Hashing Preferable?

- When the system is stateful
- And recreating or transferring state is expensive

- For in-memory Hash Tables remapping is cheap
  - Requires copying a pointer in RAM
- For Distributed Hash Tables remapping is costly
  - Moving a key requires transfer over a network

Stable Hashing with Global Namespaces

- Last year we presented about unlimited scale:
  - Main lesson: it requires eliminating points of contention, including metadata systems
  - We achieved this with a “Global Namespace”

- Namespace is fixed, but system is dynamic…
  - We needed an algorithm that could adapt to changes in the system, *and do so efficiently!*

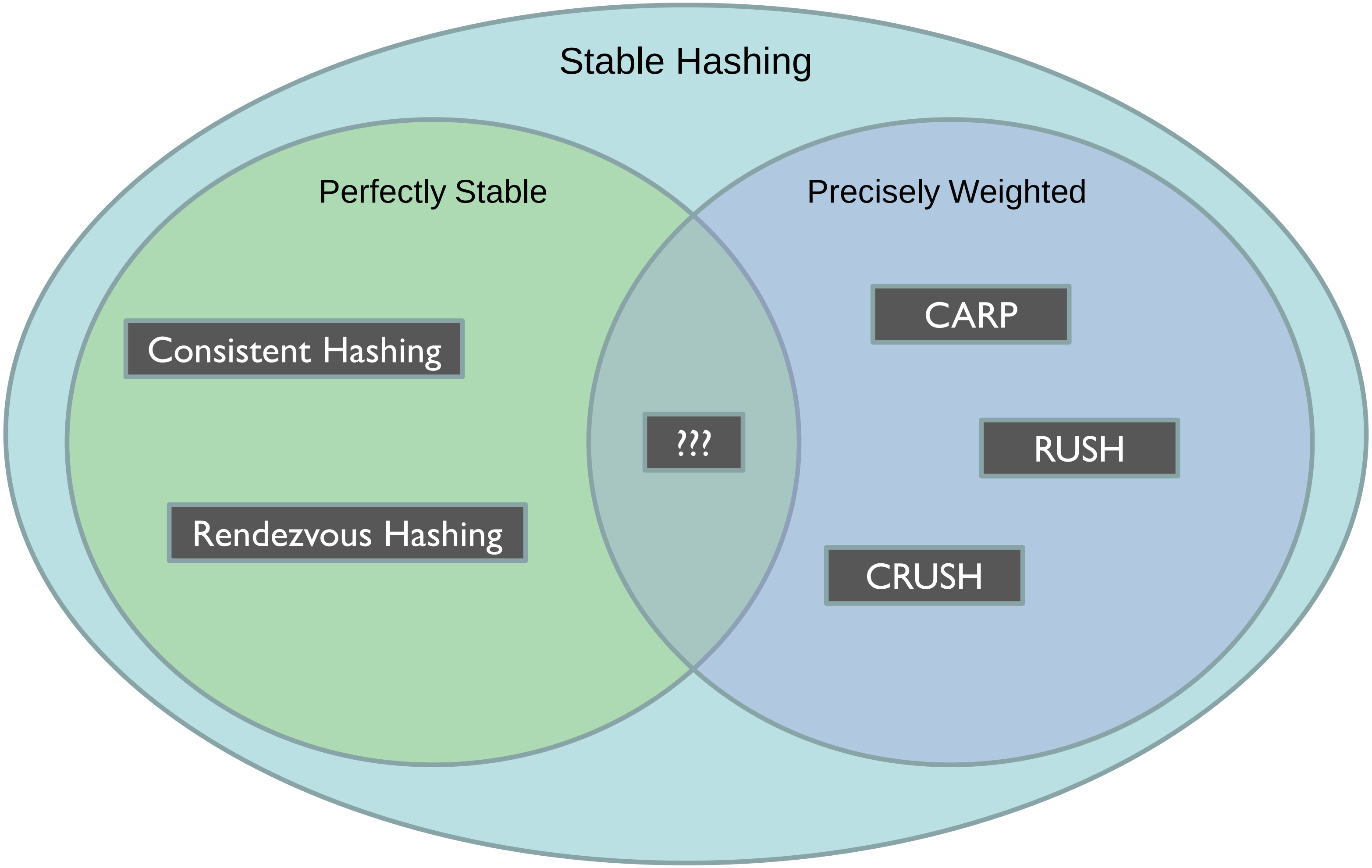
Our motivations for using Stable Hashing

- Helps balance:
  - Storage (read/write) load across nodes
  - Storage utilization across nodes
- Minimizes disruption for:
  - Addition of new nodes
  - Resizing of existing nodes (disk addition)
  - Removing or repurposing nodes
  - Replacing obsolete nodes with new ones

But what algorithm to use…

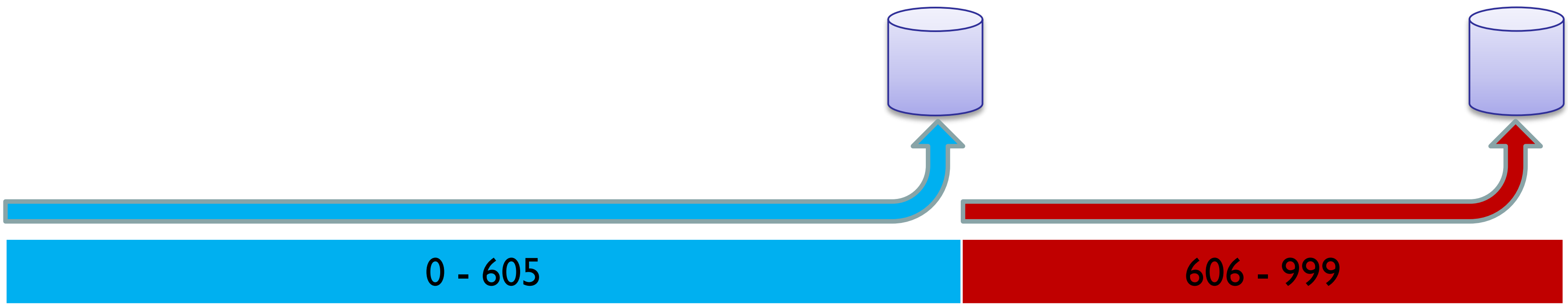
- Perfect Stable Hashing:
  - Rendezvous Hashing (‘96)
  - Consistent Hashing (‘97)
- Weighted Stable Hashing:
  - CARP (‘98)
  - RUSH/CRUSH (‘04/’06)

Classes of Stable Hashing Algorithms



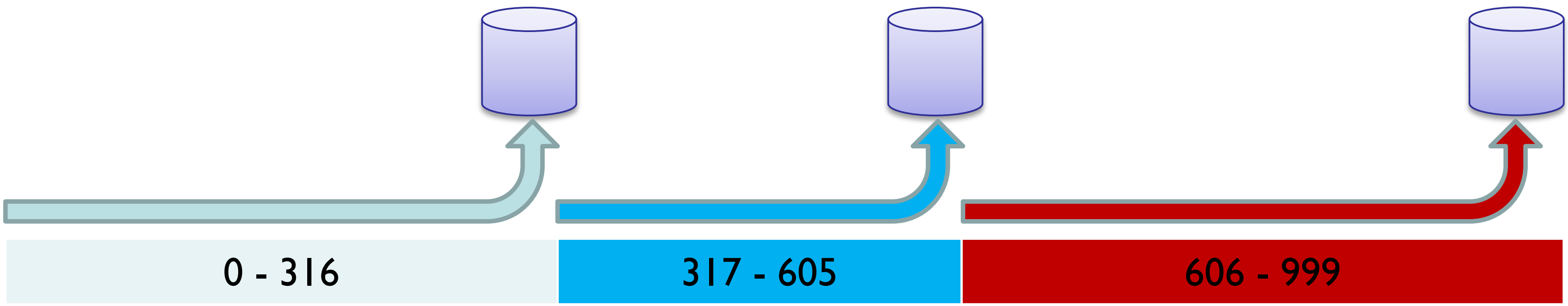
How Consistent Hashing Works

- Buckets inserted in random positions
- Keys map to the next node greater than that key



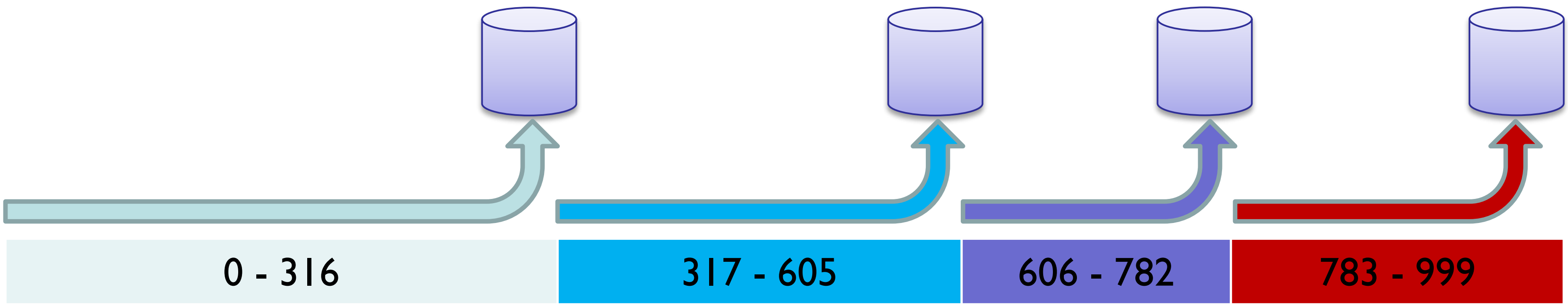
How Consistent Hashing Works

- Buckets inserted in random positions
- Keys map to the next node greater than that key



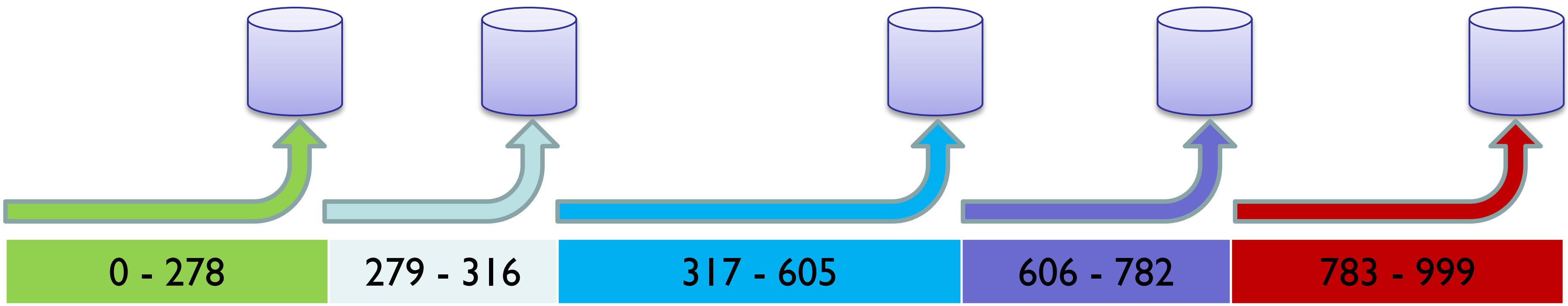
How Consistent Hashing Works

- Buckets inserted in random positions
- Keys map to the next node greater than that key



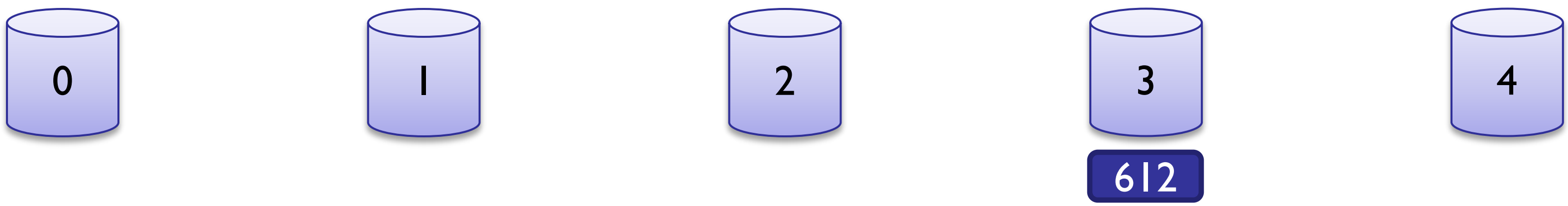
How Consistent Hashing Works

- Buckets inserted in random positions
- Keys map to the next node greater than that key



How Rendezvous Hashing Works

- Hash(Bucket ID || Key) → Score
- Bucket with the highest score wins



H("0" || 612) = 759

H("1" || 612) = 481

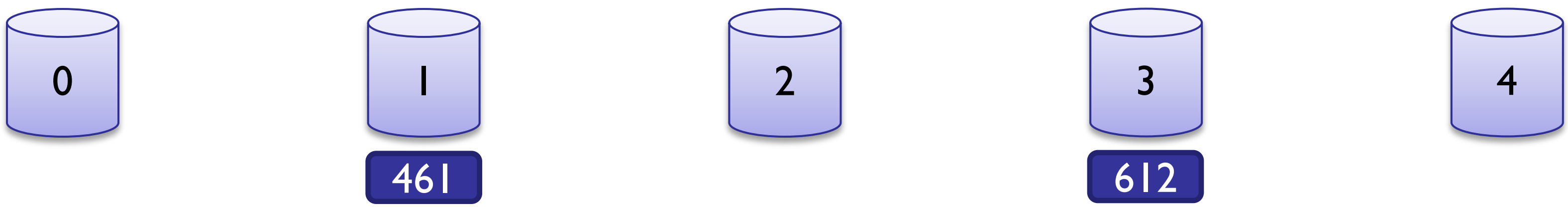
H("2" || 612) = 830

H("3" || 612) = 879

H("4" || 612) = 484

How Rendezvous Hashing Works

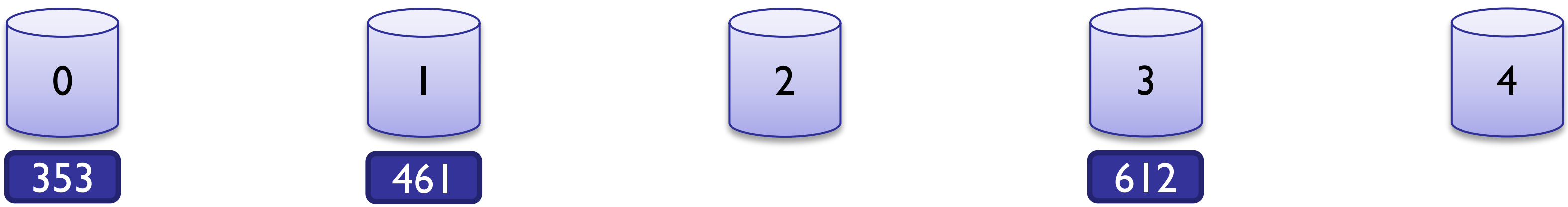
- Hash(Bucket ID || Key) → Score
- Bucket with the highest score wins



H("0" || 461) = 707
H("1" || 461) = 854
H("2" || 461) = 370
H("3" || 461) = 065
H("4" || 461) = 804

How Rendezvous Hashing Works

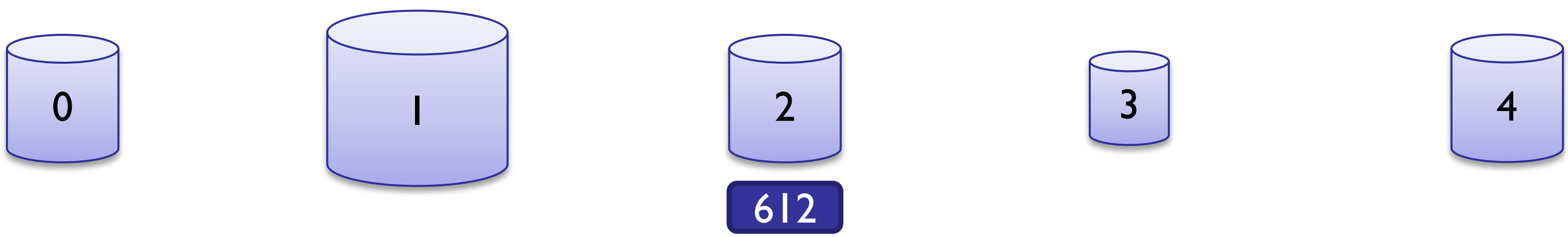
- Hash(Bucket ID || Key) → Score
- Bucket with the highest score wins



H("0" || 353) = 746
H("1" || 353) = 207
H("2" || 353) = 515
H("3" || 353) = 668
H("4" || 353) = 252

How CARP Works

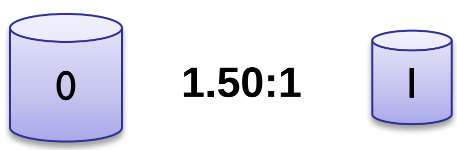
- CARP is Rendezvous Hashing, with one change
- Scores are multiplied with a “Load Factor”



H(“0” || 612) × 0.197 = 149.24
H(“1” || 612) × 0.240 = 115.60
H(“2” || 612) × 0.197 = 163.21
H(“3” || 612) × 0.170 = 149.22
H(“4” || 612) × 0.197 = 095.17

Why CARP isn't Perfectly Stable

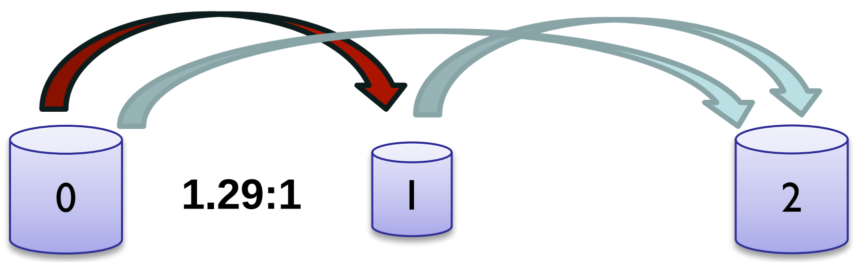
- Load factors in CARP must be relatively scaled
  - If any node's weighting changes, or if any node is added or removed, then all load factors must be recomputed



Load Factor:	0.6000	0.4000
Weight:	200	100

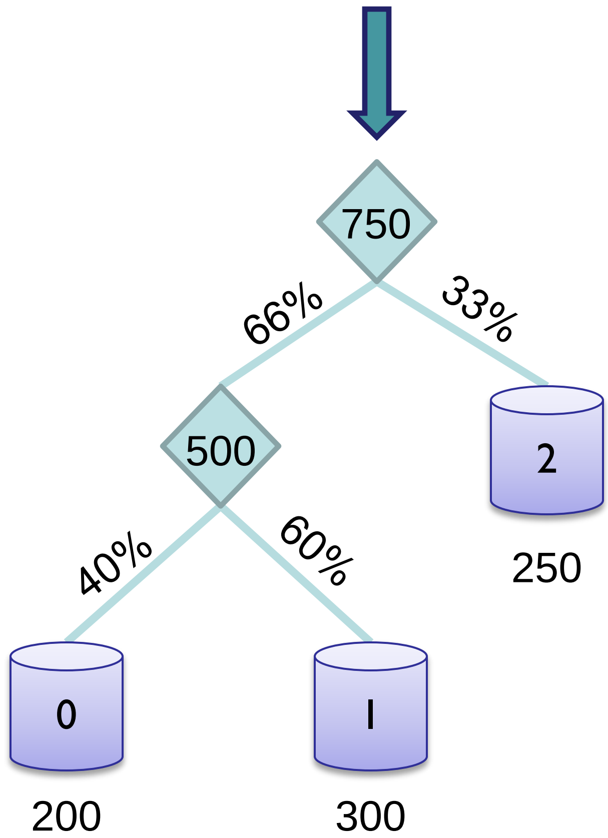
Why CARP isn't Perfectly Stable

- Load factors in CARP must be relatively scaled
  - If any node's weighting changes, or if any node is added or removed, then all load factors must be recomputed



Load Factor:	0.3604	0.2792	0.3604
Weight:	200	100	200

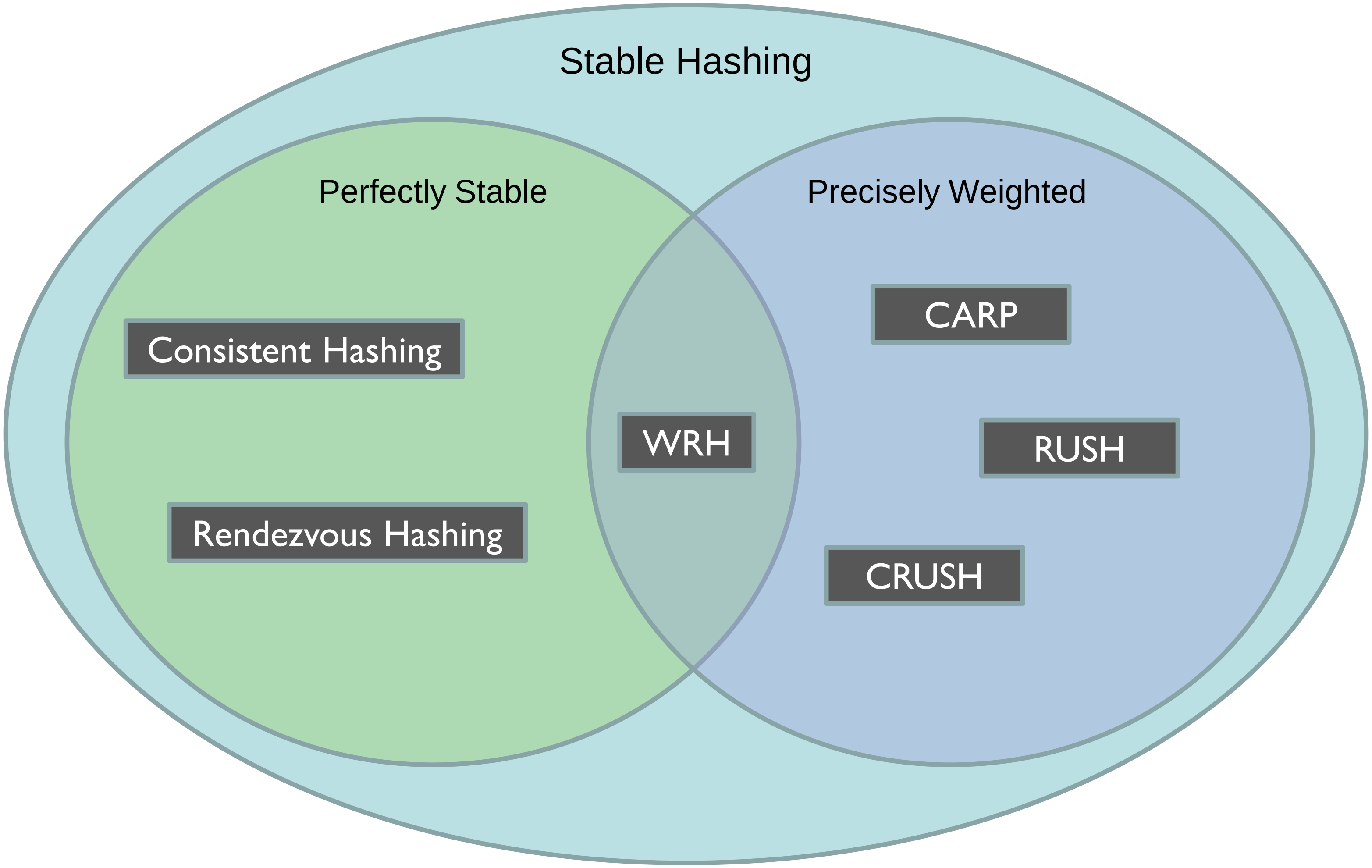
How RUSH/CRUSH work



Evolution of Stable Hashing

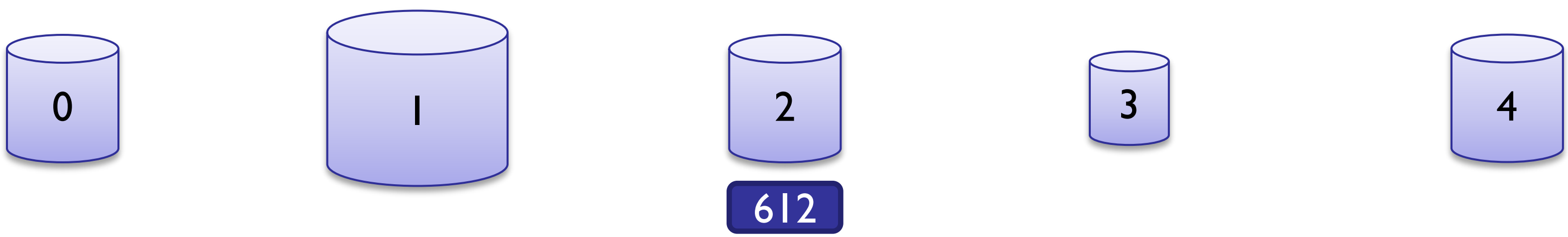
- Perfect Stable Hashing:
  - Rendezvous Hashing (‘96)
  - Consistent Hashing (‘97)
- Weighted Stable Hashing:
  - CARP (‘98)
  - RUSH/CRUSH (‘04/‘06)
- Perfect Weighted Stable Hashing:
  - <u>Weighted Rendezvous Hash (‘14)</u> NEW!

Classes of Stable Hashing Algorithms



How Weighted Rendezvous Hashing Works

- WRH adjusts scores before weighting them
- Unlike CARP, scores aren't relatively scaled



200 / -Log(H("0" || 612) / MAX_HASH) = 725.29

400 / -Log(H("1" || 612) / MAX_HASH) = 546.43

200 / -Log(H("2" || 612) / MAX_HASH) = 1073.36

100 / -Log(H("3" || 612) / MAX_HASH) = 775.37

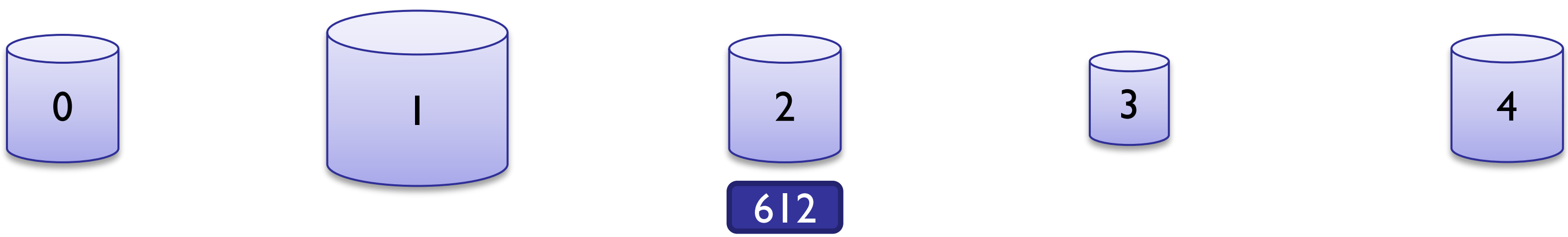
200 / -Log(H("4" || 612) / MAX_HASH) = 275.61

Why WRH is perfectly stable

- When a node is added, removed, or changed:
  - Only the scores for that node change
    - It may win some keys (if weight increased)
    - It may lose some keys (if weight decreased)
- For the unchanged nodes:
  - Scores for all of them remain unchanged
    - No wasted data transfer occurs between nodes
    - Minimum data moves to recover equilibrium

Weight Change with WRH

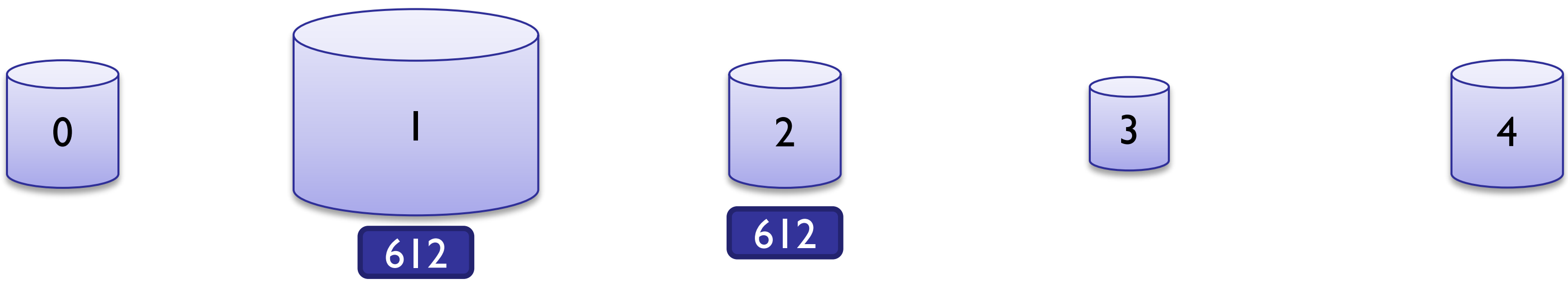
- WRH adjusts scores before weighting them
- Unlike CARP, scores aren't relatively scaled



200 / -Log(H("0" || 612) / MAX_HASH) = 725.29
400 / -Log(H("1" || 612) / MAX_HASH) = 546.43
200 / -Log(H("2" || 612) / MAX_HASH) = 1073.36
100 / -Log(H("3" || 612) / MAX_HASH) = 775.37
200 / -Log(H("4" || 612) / MAX_HASH) = 275.61

Weight Change with WRH

- WRH adjusts scores before weighting them
- Unlike CARP, scores aren't relatively scaled



200 / -Log(H("0" || 612) / MAX_HASH) = 725.29

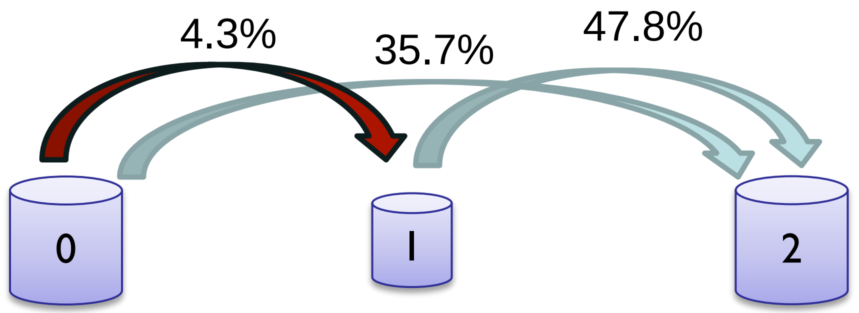
800 / -Log(H("1" || 612) / MAX_HASH) = 1092.86

200 / -Log(H("2" || 612) / MAX_HASH) = 1073.36

100 / -Log(H("3" || 612) / MAX_HASH) = 775.37

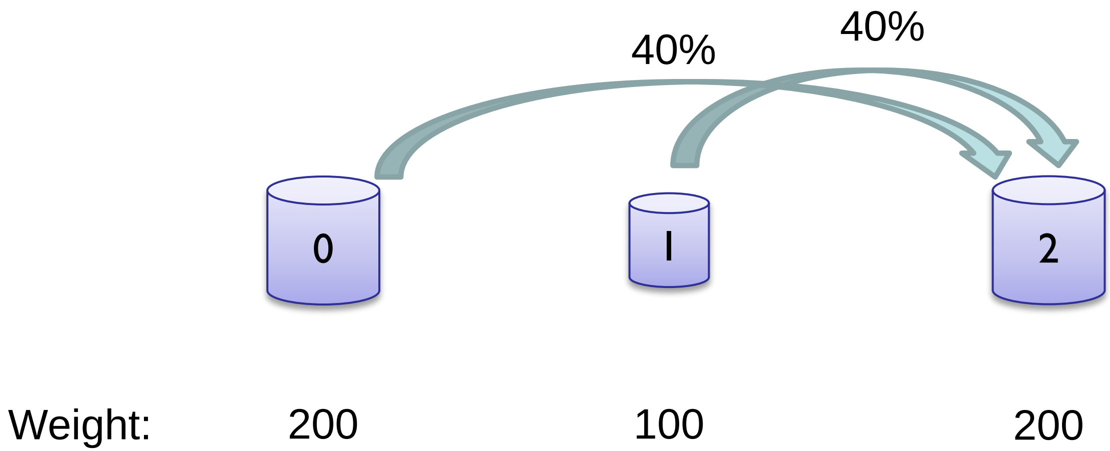
200 / -Log(H("4" || 612) / MAX_HASH) = 275.61

Keys Transferred under CARP



Weight: 200 100 200

Keys Transferred under WRH



Simplicity of WRH

```python
#!/usr/bin/python

import mmh3
import math
import binascii
import hashlib

fifty_three_ones = (0xFFFFFFFFFFFFFFFF >> (64 - 53))
fifty_three_zeros = float(1 << 53)

def int_to_float(value):
  return (value & fifty_three_ones) / fifty_three_zeros

class Bucket(object):
  def __init__(self, name, seed, weight):
    self.name, self.seed, self.weight = name, seed, weight

  def compute_weighted_score(self, name):
    hash_1, hash_2 = mmh3.hash64(str(name), 0xFFFFFFFF & self.seed)
    hash_f = int_to_float(hash_2)
    score = 1.0 / -math.log(hash_f)
    return self.weight * score

  def __str__(self):
    return "[" + self.name + " (" + str(self.seed) + ", " + str(self.weight) + ")]"

def determine_responsible_bucket(buckets, name):
  highest_score, champion = -1, None
  for bucket in buckets:
    score = bucket.compute_weighted_score(name)
    if score > highest_score:
      champion, highest_score = bucket, score
  return champion
```

Where the magic happens

Proof of Correctness

Let i $\in \{1..n\}$ be buckets and X be the set of hashable objects. Let $w_i \in \mathbb{R}_0^+$ represent the weight for bucket i, and $h_i : X \to [0,1]$ be the hash function for bucket i. Assume $h_i(x)$ is a perfect hashing function - that is it maps $x \in X$ to a uniform, continuous random variable on $[0,1]$. Define f_i as

$$f_i(y) = \frac{-w_i}{ln(y)}$$

We define the champion function, C, as

$$C(x) = \arg\max_i f_i(h_i(x))$$

Theorem 1. $Pr[C(x) = i] = \frac{w_i}{\sum_{j=1}^n w_j}$

Proof. Let $x \in X$ and $h_i(x) = z$

$$f_i(a) > f_i(b) \iff a > b$$

$$f_j(h_j(x)) = f_i(z) \iff h_j(x) = f_j^{-1}(f_i(z))$$

$$Pr[f_j(h_j(x)) < f_i(h_i(x)) \mid h_i(x) = z] = Pr[h_j(x) < f_j^{-1}(f_i(z)]$$

$$= f_j^{-1}(f_i(z))$$

Then

$$Pr[C(x) = i \mid h_i(x) = z] = \prod_{j \neq i}^{n} f_j^{-1}(f_i(z))$$

$$Pr[C(x) = i] = \int_{h_i(x)=0}^{1} \prod_{j \neq i}^{n} f_j^{-1}(f_i(h_i(x))$$

$$= \int_{h_i(x)=0}^{1} \prod_{j \neq i}^{n} e^{\frac{-w_j}{-w_i/ln(h_i(x)}} = \int_{h_i(x)=0}^{1} h_i(x)^{\sum_{j \neq i}^{n} \frac{w_j}{w_i}}$$

$$= \frac{1}{1 + \sum_{j \neq i}^{n} \frac{w_j}{w_i}} = \frac{w_i}{\sum_{j=1}^{n} w_j}$$

$$Pr[C(x) = i] = \frac{w_i}{\sum_{j=1}^{n} w_j}$$

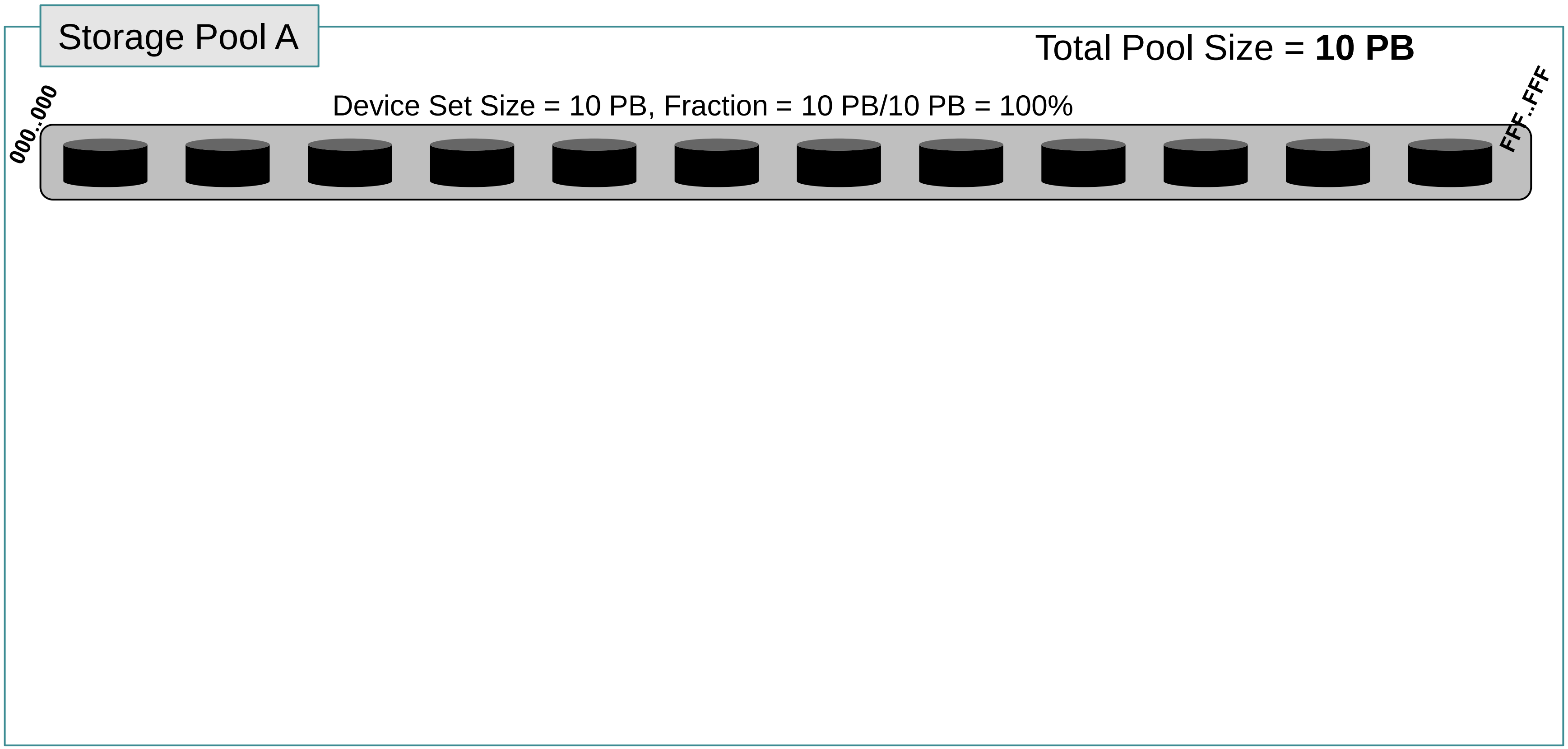
$\square$

1

How we use the WRH

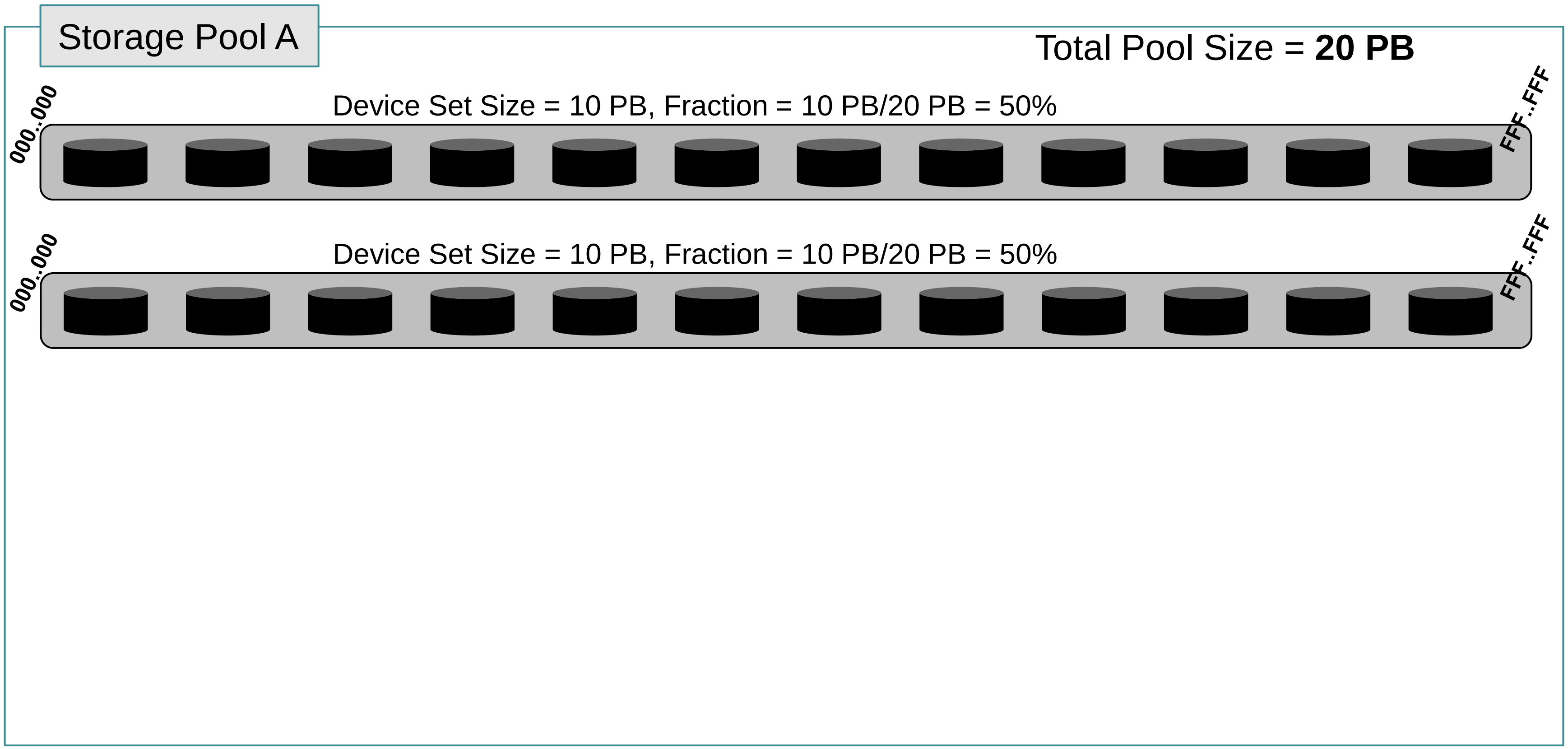
- Our system is grown by sets of devices
  - Each set is composed of devices spread across fault domains (racks, sites, etc.)
- Devices have a lifecycle:
  - Added, possibly expanded, then retired
- The WRH selects which “device set” to write a given object to or read a given object from

Evolution of a Storage Pool

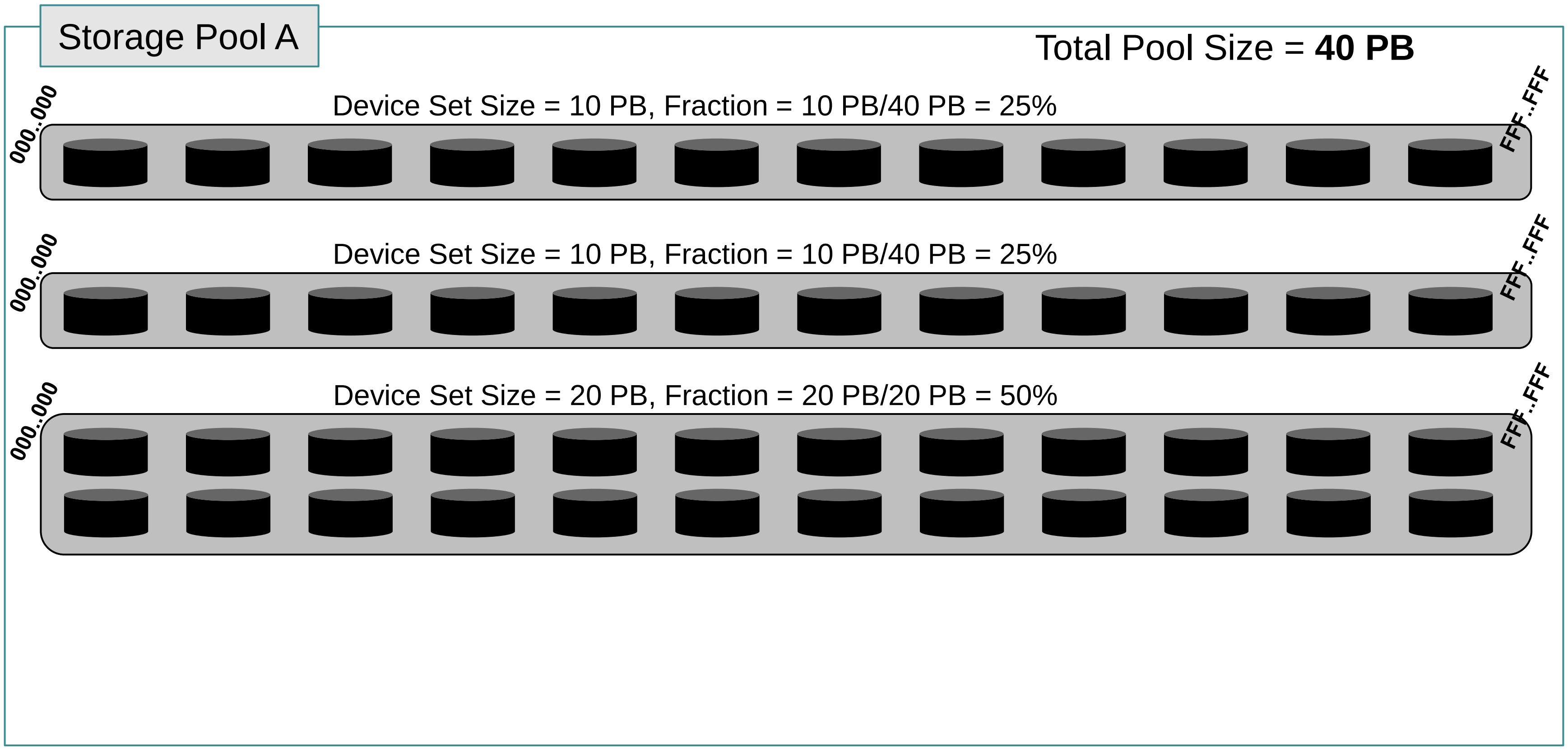


9/22/2015

Evolution of a Storage Pool

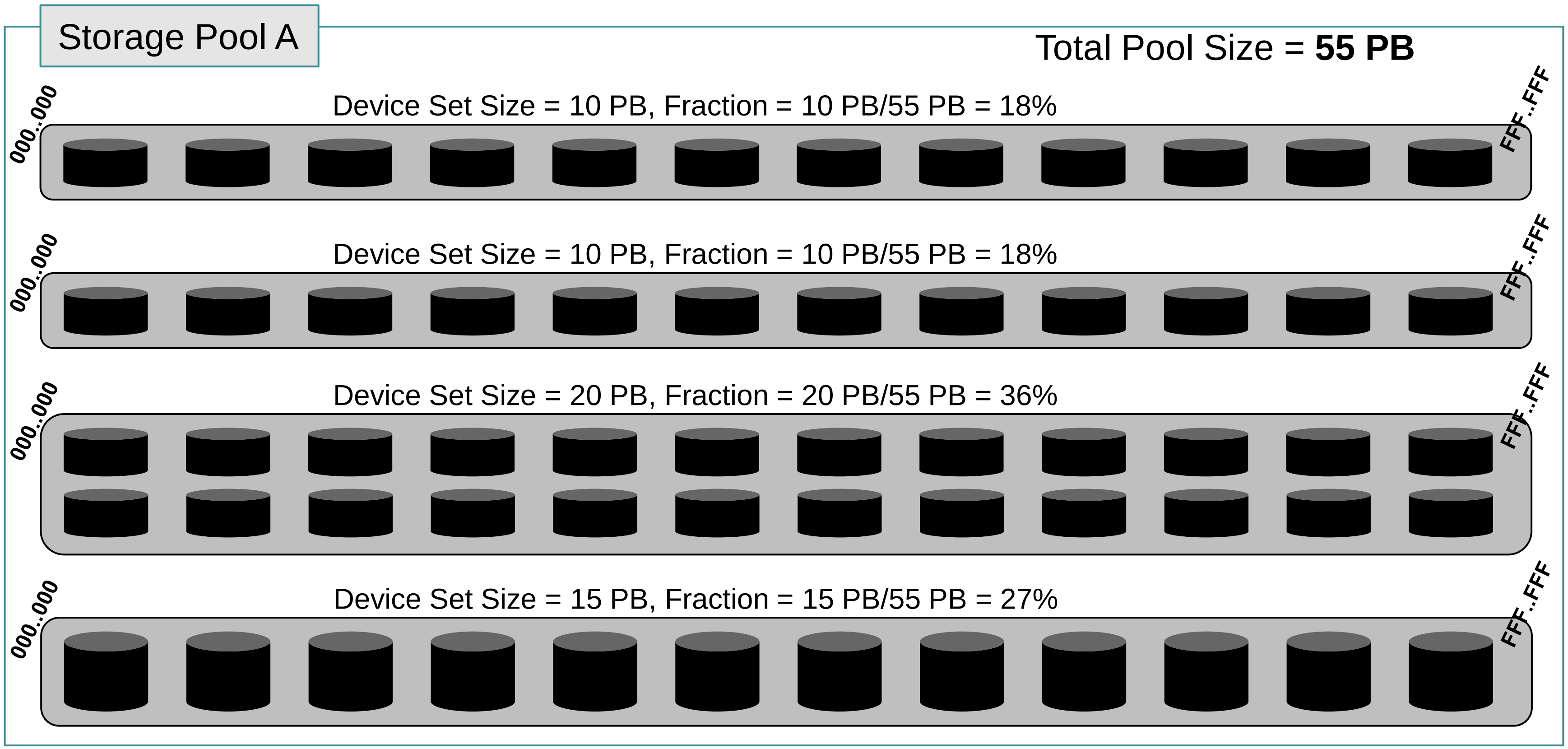


Evolution of a Storage Pool



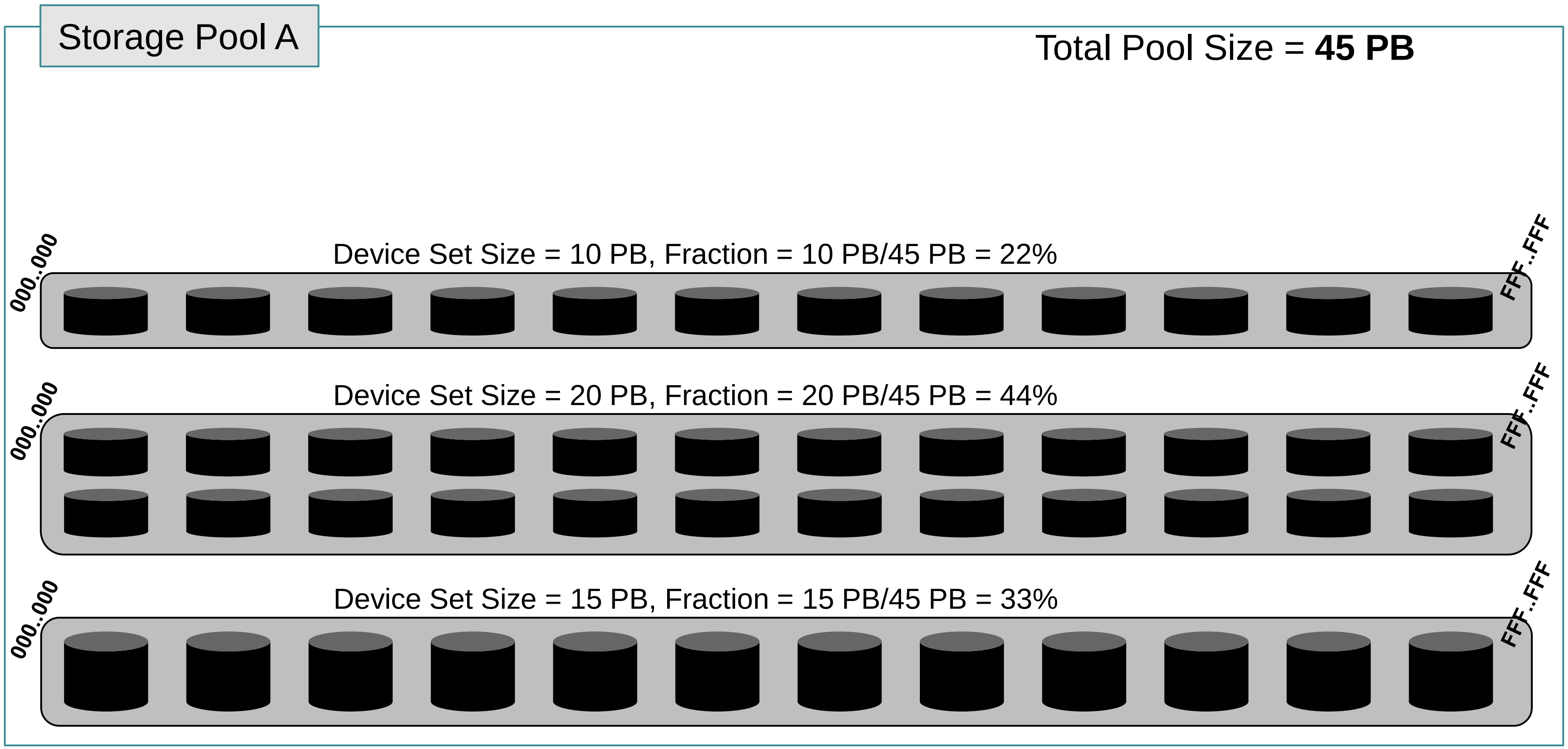
9/22/2015

Evolution of a Storage Pool



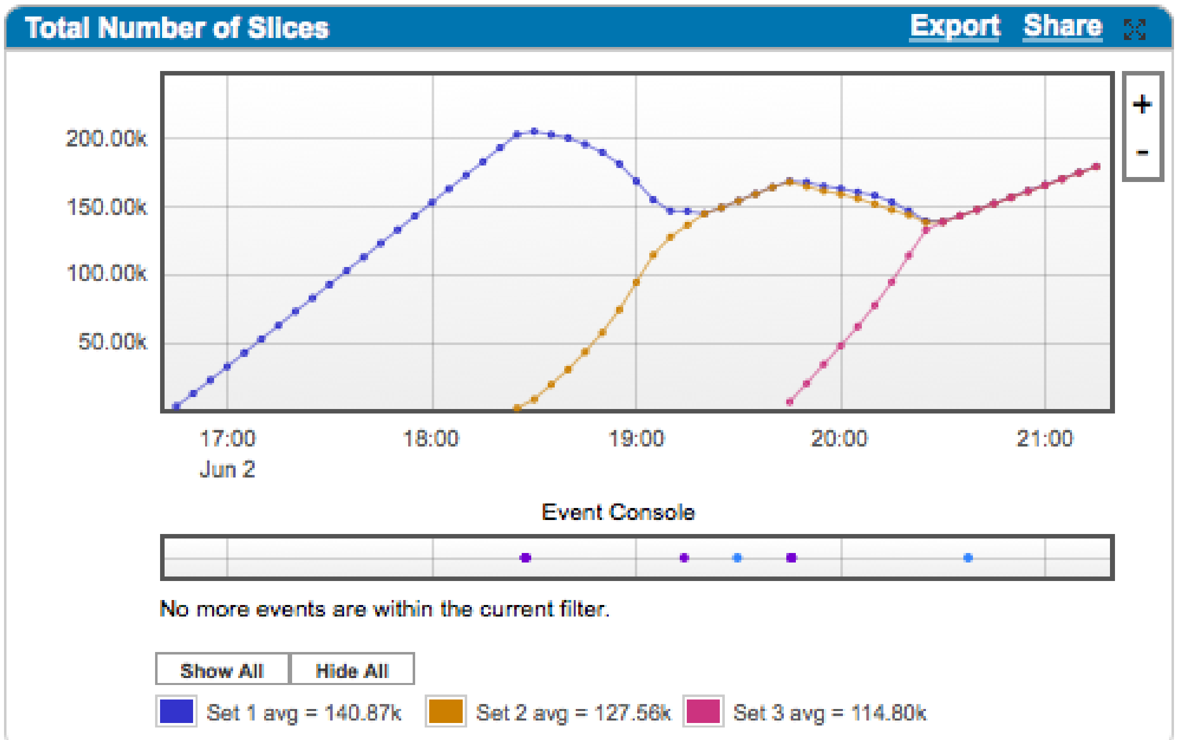
9/22/2015

Evolution of a Storage Pool



9/22/2015

Moving Data according to the WRH



9/22/2015

Storage Resource Map

- Shows relative capacities of device sets each of which is independently reliable storage


```
"storage_pool_map": {
    "657fe35a-a87a-44cf-b766-8e890aea7b2e": {
        "weight": "4600000000000000",
        "hash_seed": 67662243
    },
    "bfa3a243-c2f4-3a1c-afa9-cee4b56c1da1": {
        "weight": "2200000000000000"
        "hash_seed": 27781369
    }
}
```

Efficient Replacement: reuse seed

- The Hash Seed enables a clever trick:
  - When retiring a device set with replacement, we-use the same hash seed for the new set
  - Since it seeds hashes in the same way, it computes identical scores as the old set
  - When the retired set's weight is set to 0, all keys move from the old set to the new one

Other ways to use WRH

- We see many potential applications:
  - Performing work
    - Take on rebuilding tasks from a work queue
    - Assign compute jobs according to CPU capacity
  - Route access requests to “Access nodes”
    - Reduces contention, maximizes cache hits
  - Map data to drives within a storage node
    - When a drive fails, remap data to other drives

Thank you! Any Questions?

References

- Rendezvous Hashing: https://en.wikipedia.org/wiki/Rendezvous_hashing
- Consistent Hashing: https://en.wikipedia.org/wiki/Consistent_hashing
- CARP: https://tools.ietf.org/html/draft-vinod-carp-v1-03
- RUSH: http://www.ssrc.ucsc.edu/Papers/honicky-ipdps04.pdf
- CRUSH: http://www.crss.ucsc.edu/media/papers/weil-sc06.pdf
- CEPH: http://ceph.com/docs/master/rados/operations/crush-map/
- OpenStack: http://docs.openstack.org/developer/swift/ring.html
- GlusterFS: http://blog.gluster.org/2012/03/glusterfs-algorithms-distribution/
- Cassandra: http://blog.imaginea.com/consistent-hashing-in-cassandra/
- DynamoDB: http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf