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

(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

<table>
<thead>
<tr>
<th>bucket (0)</th>
<th>key</th>
</tr>
</thead>
<tbody>
<tr>
<td>374</td>
<td>975</td>
</tr>
<tr>
<td>602</td>
<td>68</td>
</tr>
<tr>
<td>580</td>
<td>16</td>
</tr>
<tr>
<td>461</td>
<td></td>
</tr>
<tr>
<td>825</td>
<td>700</td>
</tr>
<tr>
<td>612</td>
<td>353</td>
</tr>
<tr>
<td>618</td>
<td></td>
</tr>
<tr>
<td>993</td>
<td>233</td>
</tr>
</tbody>
</table>

bucket = stable_hash(buckets, key)
Stable Hashing Resize

bucket = stable_hash(buckets, key)
Who uses Stable Hashing?

- **Caching/Routing:**
  - Microsoft Proxy Server
  - Saur

- **DHT/Storage:**
  - Gluster
  - Amazon DynamoDB
  - Cassandra
  - ceph
  - openstack
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

- Perfectly Stable
  - Consistent Hashing
  - Rendezvous Hashing

- Precisely Weighted
  - CARP
  - RUSH
  - CRUSH
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

\[
\begin{align*}
H(“0” \ || \ 612) &= 759 \\
H(“1” \ || \ 612) &= 481 \\
H(“2” \ || \ 612) &= 830 \\
H(“3” \ || \ 612) &= 879 \ (\text{golden}) \\
H(“4” \ || \ 612) &= 484
\end{align*}
\]
How Rendezvous Hashing Works

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

0

1

2

3

4

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

<table>
<thead>
<tr>
<th>Bucket ID</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>746</td>
</tr>
<tr>
<td>1</td>
<td>207</td>
</tr>
<tr>
<td>2</td>
<td>515</td>
</tr>
<tr>
<td>3</td>
<td>668</td>
</tr>
<tr>
<td>4</td>
<td>252</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
H(\text{"0" || 353}) &= 746 \\
H(\text{"1" || 353}) &= 207 \\
H(\text{"2" || 353}) &= 515 \\
H(\text{"3" || 353}) &= 668 \\
H(\text{"4" || 353}) &= 252
\end{align*}
\]
How CARP Works

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

| Key  | Hash Value | Score Multiplied
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>612</td>
<td>149.24</td>
</tr>
<tr>
<td>1</td>
<td>612</td>
<td>115.60</td>
</tr>
<tr>
<td>2</td>
<td>612</td>
<td>163.21</td>
</tr>
<tr>
<td>3</td>
<td>612</td>
<td>149.22</td>
</tr>
<tr>
<td>4</td>
<td>612</td>
<td>095.17</td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>Load Factor</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6000</td>
<td>200</td>
</tr>
<tr>
<td>0.4000</td>
<td>100</td>
</tr>
</tbody>
</table>
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
How RUSH/CRUSH work

The diagram shows a hierarchy where the root node is labeled 750. This node has two children: one labeled 500 and the other labeled 250. The node 500 has two children: one labeled 200 and the other labeled 300. The node 250 has one child labeled 2.

The percentages indicate the distribution of data or resources: 66% from the root to the 500 node, 33% from the root to the 2 node, 40% from the 500 node to the 200 node, and 60% from the 500 node to the 300 node.
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:**
  - Weighted Rendezvous Hash (‘14)
Classes of Stable Hashing Algorithms

Stable Hashing

Perfectly Stable

- Consistent Hashing
- Rendezvous Hashing

Precisely Weighted

- CARP
- RUSH
- CRUSH

WRH
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

\[
\begin{align*}
200 / -\log(H(\text{"0"} || 612) / \text{MAX\_HASH}) &= 725.29 \\
400 / -\log(H(\text{"1"} || 612) / \text{MAX\_HASH}) &= 546.43 \\
200 / -\log(H(\text{"2"} || 612) / \text{MAX\_HASH}) &= 1073.36 \\
100 / -\log(H(\text{"3"} || 612) / \text{MAX\_HASH}) &= 775.37 \\
200 / -\log(H(\text{"4"} || 612) / \text{MAX\_HASH}) &= 275.61
\end{align*}
\]
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

![Diagram showing keys transferred under CARP with weights and percentages.]

Weight:
0: 200
1: 100
2: 200

4.3%
35.7%
47.8%
Keys Transferred under WRH

Weight: 200 100 200
Simplicity of WRH

```python
#!/usr/bin/python
import mmh3
import math
import binascii
import hashlib

fifty_three_ones = (0xFFFFFFFFFFFFFFF >> (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(self, 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 blocks and $X$ be the set of hashable objects. Let $w_i \in \mathbb{R}$ represent the weight for block $i$, and $h_i : X \rightarrow [0, 1]$ be the hash function for block $i$. Assume $h_i(x)$ is a perfect hashing function - that is it maps $x \in X$ to a uniform, continuous random variable in $[0, 1]$. Define $f_i$ as

$$f_i(x) = \frac{w_i}{N_{h_i(x)}}$$

We define the champion function, $C$, as

$$C(x) = \alpha \max h_i(x)$$

**Theorem 1.** $Pr(C(x) = i) = \frac{w_i}{\sum_{j \in \{1, n\}} w_j}$

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

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

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

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

Then

$$Pr(C(x) = i) = \int_{h_i(x) = z} N_{h_i(x)}$$

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

$$= \int_{h_i(x) = z} \prod_{j \neq i} \frac{w_j}{\sum_{k \neq i} w_k}$$

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

$$Pr(C(x) = i) = \frac{w_i}{\sum_{j \neq i} w_j}$$

$\blacksquare$
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

Storage Pool A

Total Pool Size = 10 PB

Device Set Size = 10 PB, Fraction = 10 PB/10 PB = 100%
Evolution of a Storage Pool

Storage Pool A

Total Pool Size = 20 PB

Device Set Size = 10 PB, Fraction = 10 PB/20 PB = 50%

Device Set Size = 10 PB, Fraction = 10 PB/20 PB = 50%
Evolution of a Storage Pool

Storage Pool A

Total Pool Size = 40 PB

Device Set Size = 10 PB, Fraction = 10 PB/40 PB = 25%

Device Set Size = 10 PB, Fraction = 10 PB/40 PB = 25%

Device Set Size = 20 PB, Fraction = 20 PB/20 PB = 50%
Evolution of a Storage Pool

Storage Pool A

Total Pool Size = 55 PB

Device Set Size = 10 PB, Fraction = 10 PB/55 PB = 18%

Device Set Size = 10 PB, Fraction = 10 PB/55 PB = 18%

Device Set Size = 20 PB, Fraction = 20 PB/55 PB = 36%

Device Set Size = 15 PB, Fraction = 15 PB/55 PB = 27%
Evolution of a Storage Pool

Storage Pool A

Total Pool Size = 45 PB

- Device Set Size = 10 PB, Fraction = 10 PB/45 PB = 22%
- Device Set Size = 20 PB, Fraction = 20 PB/45 PB = 44%
- Device Set Size = 15 PB, Fraction = 15 PB/45 PB = 33%
Moving Data according to the WRH

![Graph showing total number of slices over time with event console and data points for Set 1, Set 2, and Set 3 averages.]
Storage Resource Map

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

```json
"storage_pool_map": {
    "657fe35a-a87a-44cf-b766-8e890aea7b2e": {
        "weight": "46000000000000000",
        "hash_seed": 67662243
    },
    "bfa3a243-c2f4-3a1c-afa9-cee4b56c1da1": {
        "weight": "22000000000000000",
        "hash_seed": 27781369
    }
}
```
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

- Consistent Hashing: [https://en.wikipedia.org/wiki/Consistent_hashing](https://en.wikipedia.org/wiki/Consistent_hashing)
- RUSH: [http://www.ssrc.ucsc.edu/Papers/honicky-ipdps04.pdf](http://www.ssrc.ucsc.edu/Papers/honicky-ipdps04.pdf)
- CRUSH: [http://www.crss.ucsc.edu/media/papers/weil-sc06.pdf](http://www.crss.ucsc.edu/media/papers/weil-sc06.pdf)
- OpenStack: [http://docs.openstack.org/developer/swift/ring.html](http://docs.openstack.org/developer/swift/ring.html)