Simple Scenario

Suppose you're building a big web cache that holds copies of web pages your users have downloaded:

How do you allocate pages/images to the cache servers?

Static Partitioning

- Items A–C go to this server/bucket/bin, D–Fd-f go to that server/bucket/bin, ...
- Requires **planning**
  - If you used the server name, what if “`cowpatties.com`” had 1’000’000 pages, but “`zebras.com`” had only 10?
  - This may cause **load imbalance**
- Could fill up the bins as they arrive
  - Requires tracking the location of every object at the front-end.
  - May be reasonable design for huge objects

Conventional Hashing

- Recall that a **hash function** maps elements of a (usually super-big) universe U, like URLs, to "buckets", such as 32-bit values
  - A "good" hash function is **easy** to remember and evaluate.
  - For all practical purposes, a "good" hash function behaves like a **totally random** function.
- Given a "good" hash function, we can set

  bucket = hash(x) mod num_buckets
- Now the server we use is a deterministic function of the item
  - e.g., sha1(URL) → 160 bit ID % 20 → a server ID

Conventional Hashing

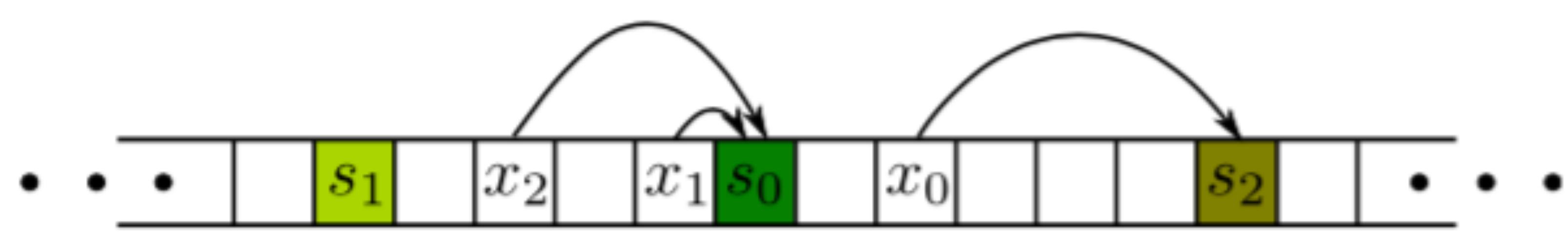
- Recall that a **hash function** maps elements of a (usually super-big) universe U, like URLs, to "buckets", such as 32-bit values
  - A "good" hash function is **easy** to remember and evaluate.
  - For all practical purposes, a "good" hash function behaves like a **totally random** function.
- Given a "good" hash function, we can set

  bucket = hash(x) mod num_buckets

- Now the server we use is a deterministic function of the item
  - e.g., sha1(URL) → 160 bit ID % 20 → a server ID
- But what happens if we want to **add or remove a server**?

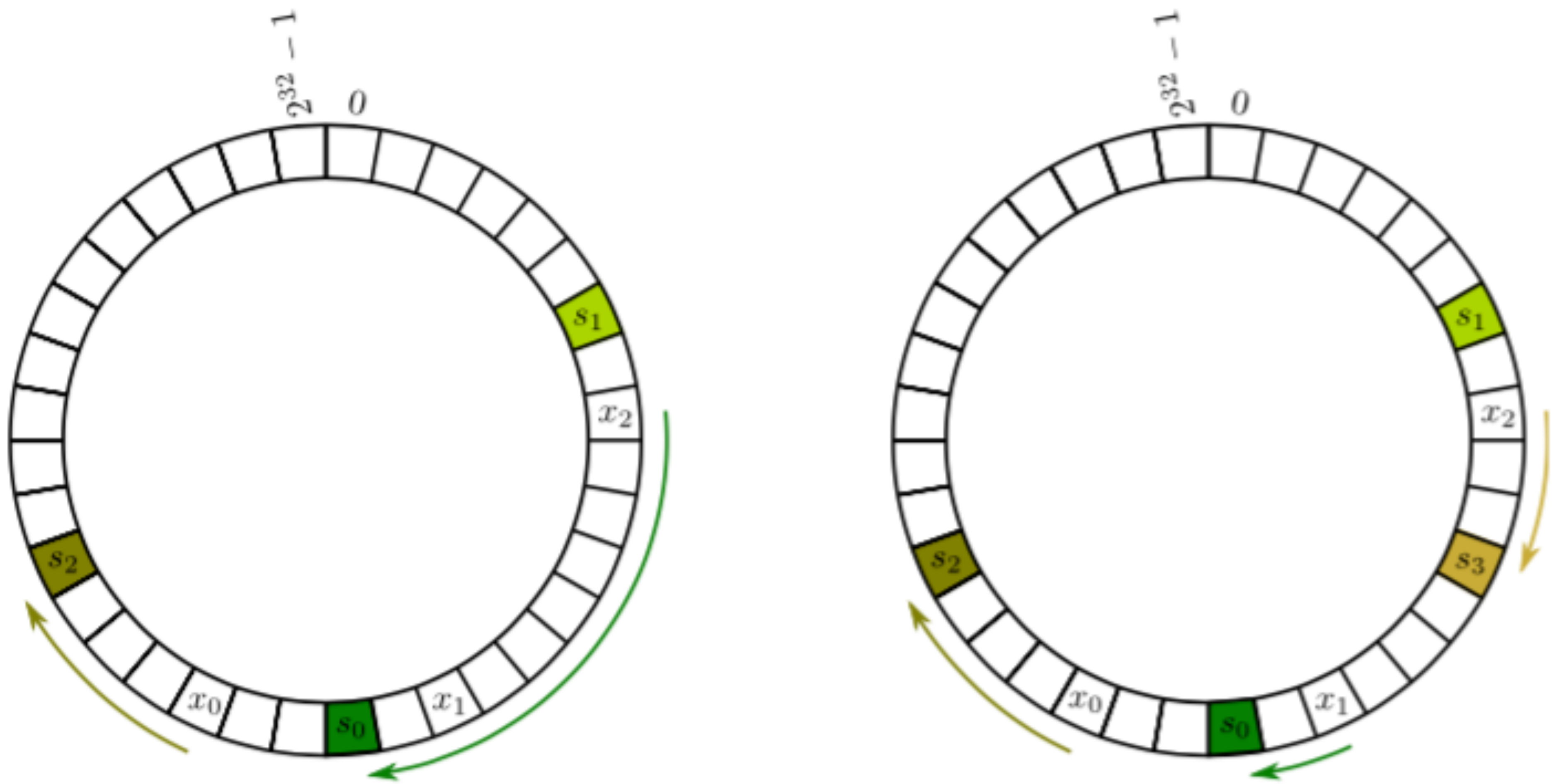
Consistent Hashing

- The key idea is: in addition to hashing the names of all objects (URLs) x, like before, **we also hash the names of all the servers** s. The object and server names need to be hashed to the same range, such as 32-bit values.



- Given an object x that hashes to the bucket h(x), **we scan buckets to the right** of h(x) until we find a bucket h(s) to which the name of some server s hashes.
- We **wrap around** the array, if necessary.

Consistent Hashing



- Hash of object = closest clockwise bucket ("*successor*")
- N servers partition the circle into N segments, with each server responsible for all objects in one of these segments.

Properties

- **Balance**: assuming reasonable hash functions, by symmetry, the expected load on each of the N servers is exactly a 1/N fraction of the objects.
- **Smoothness**: suppose we add a new server s — which objects have to move? **Only the objects stored at s**. When a server is added, the expected number of items that move to the newly added server is (#items)/(1+#servers).
- **Complexity**: to implement Lookup and Insert we can use a ~~hash table~~, a ~~heap~~, a **balanced binary search tree**, with O(log(n)) lookup and insert implementations.

Implementation

- **To insert an item x**:

  (a) Find the successor of $h_i(x)$ in the BST (if it has no successor in the BST then return the machine with the smallest h_m value)

  (b) Store x in the returned machine.

- **To delete an item x**:

  (a) Find the successor of $h_i(x)$ in the BST (if it has no successor in the BST then return the machine with the smallest h_m value)

  (b) Delete x in the returned machine.

- BST: a Binary Search Tree whose keys are the values assigned to the machines.
- h_i: the function hashing items to the interval [0, 1].
- h_m: the function hashing machines to the interval [0, 1].

Implementation

- **To insert a new machine Y**:

There may be some existing items that should be stored in the new machine Y, but these items now are all stored in the successor of $h_m(Y)$ (or the machine with the smallest h_m if $h_m(Y)$ is the largest value).

(a) Find the successor of $h_m(Y)$ in the BST (if it has no successor in the BST then return the machine with the smallest h_m value)

(b) Move all items whose h_i value is less than $h_m(Y)$ to the newly inserted machine Y .

- **To delete an existing machine Y**:

(a) Find the successor of $h_m(Y)$ in the BST (if it has no successor in the BST then return the machine with the smallest h_m value)

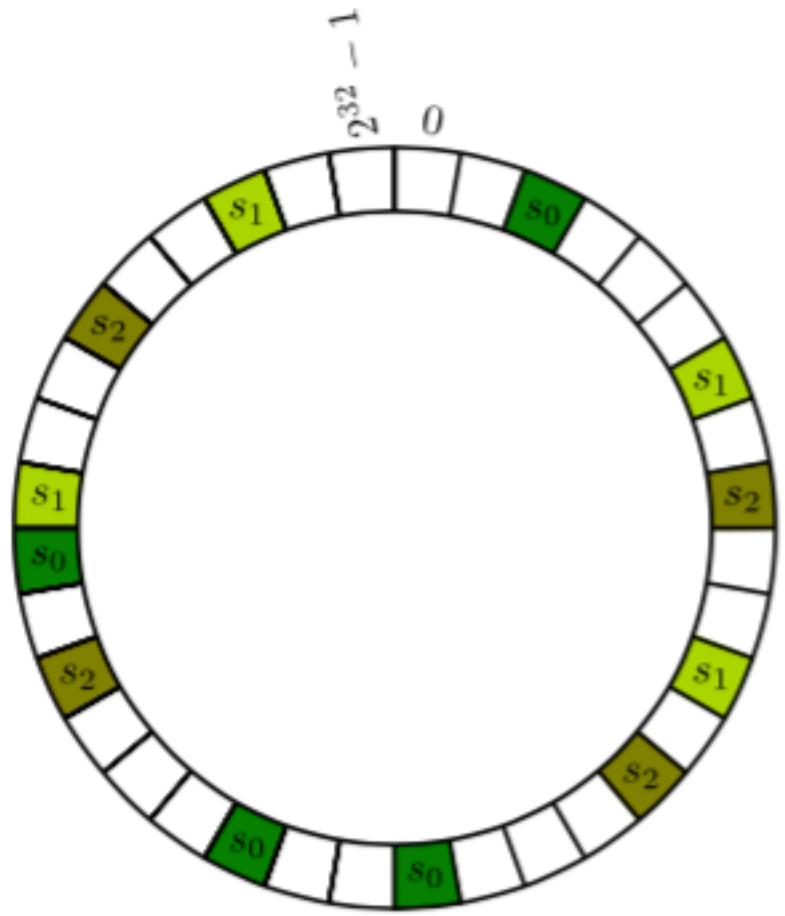
(b) Move all items in Y to the returned machine.

- BST: a Binary Search Tree whose keys are the values assigned to the machines.
- $h_{i:}$ the function hashing items to the interval [0, 1].
- h_m: the function hashing machines to the interval [0, 1].

Virtual Nodes

- While the **expected load** of each server is a 1/N fraction of the N objects, the **actual load** of each server will vary.
  - If you pick N random points on the circle, you're very unlikely to get a perfect partition of the circle into equal-sized segments.
- To **reduce imbalance**, systems often represent each physical node as k different buckets, sometimes called "**virtual nodes**" (but really, it's just multiple buckets).
  - For example, we can hash a server with K different hash functions on the same co-domain.
- Objects are assigned as before.

Virtual Nodes



- With N servers and K virtual nodes per server, by symmetry, each virtual node still expects to get a 1/(KN) fraction of the objects.
- This replication increases the number of keys stored in the balanced binary search by a factor of K, but it reduces the variance in load across servers significantly.
- Choosing $K \approx \log_2(N)$ is large enough to obtain reasonably balanced loads.

Use of consistent hashing

- The implementation of consistent hashing first appeared in a research paper in 1997 (STOC).
- In 1999, the trailer “Star Wars: The Phantom Menace” release put apple.com servers offline, while akamai.com, implementing consistent hashing, was able to serve a unauthorised copy.
- Consistent hashing is re-purposed in 2001 to address technical challenges that arise in peer-to-peer (P2P) networks (e.g., Chord and BitTorrent).
- In 2006 Amazon implements its internal Dynamo system using consistent hashing.