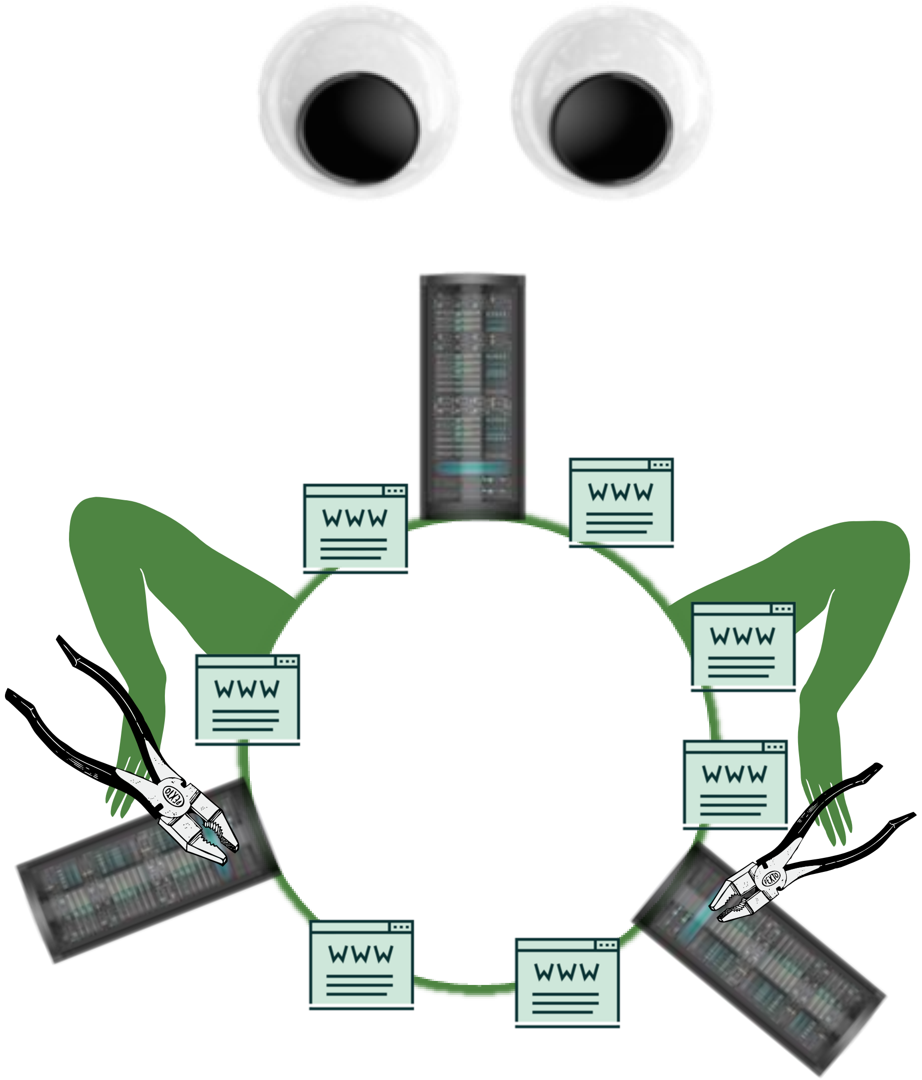
Hash & Adjust: Competitive Demand-Aware Consistent Hashing

Arash Pourdamghani
Joint work with Chen Avin, Robert Sama, Maryam Shiran, Stefan Schmid

OPODIS'24



consistent **Hash**ing **&** self-**Adjust**ing

Motivation

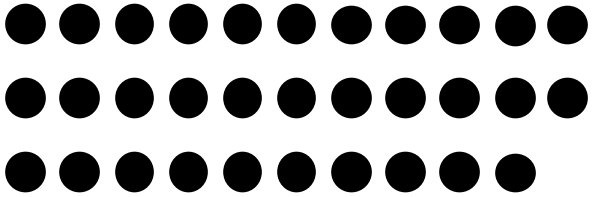
- **Consistent Hashing** [Karger et al., STOC 1997] is an essential part of today's load balancers:
  - Amazon DynamoDB, Apache Cassandra, Vimeo Skyfire

Motivation

- **Consistent Hashing** [Karger et al., STOC 1997] is an essential part of today's load balancers:
  - Amazon DynamoDB, Apache Cassandra, Vimeo Skyfire
- It is closely related to **balls into bins** problem:

Motivation

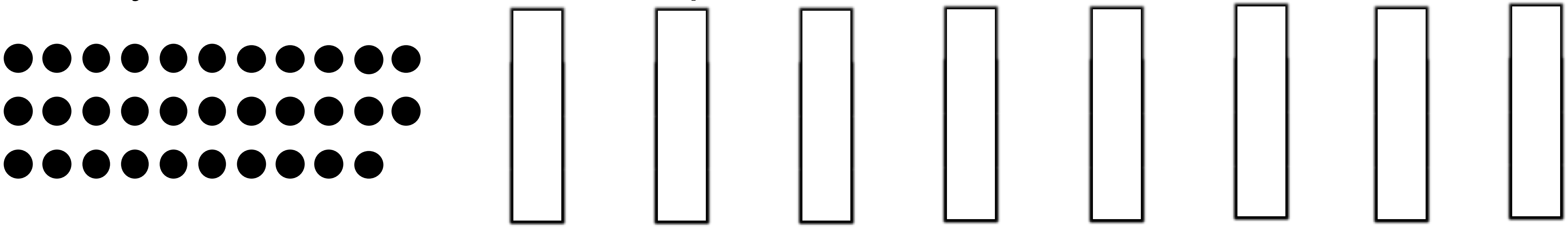
- **Consistent Hashing** [Karger et al., STOC 1997] is an essential part of today's load balancers:
  - Amazon DynamoDB, Apache Cassandra, Vimeo Skyfire
- It is closely related to **balls into bins** problem:



A set of m **items**(balls)

Motivation

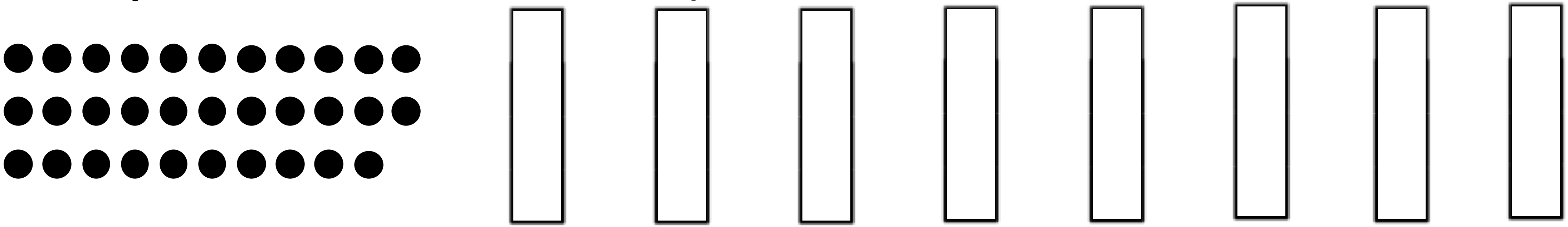
- **Consistent Hashing** [Karger et al., STOC 1997] is an essential part of today's load balancers:
  - Amazon DynamoDB, Apache Cassandra, Vimeo Skyfire
- It is closely related to **balls into bins** problem:



A set of m **items**(balls) need to be assigned to n **servers**(bins)

Motivation

- **Consistent Hashing** [Karger et al., STOC 1997] is an essential part of today's load balancers:
  - Amazon DynamoDB, Apache Cassandra, Vimeo Skyfire
- It is closely related to **balls into bins** problem:

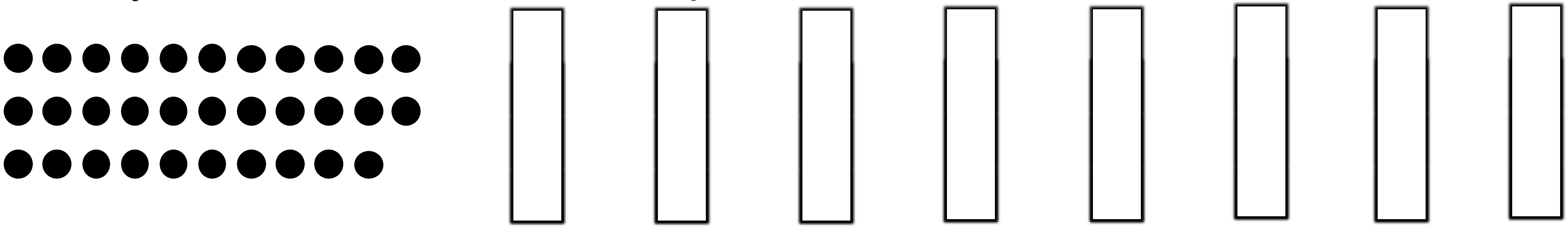


A set of m **items**(balls) need to be assigned to n **servers**(bins)

- Goals:
  - Minimize load (the number of items in each server) of servers AND access cost (number of accessed servers, accessing first server costs 1)

Motivation

- **Consistent Hashing** [Karger et al., STOC 1997] is an essential part of today's load balancers:
  - Amazon DynamoDB, Apache Cassandra, Vimeo Skyfire
- It is closely related to **balls into bins** problem:

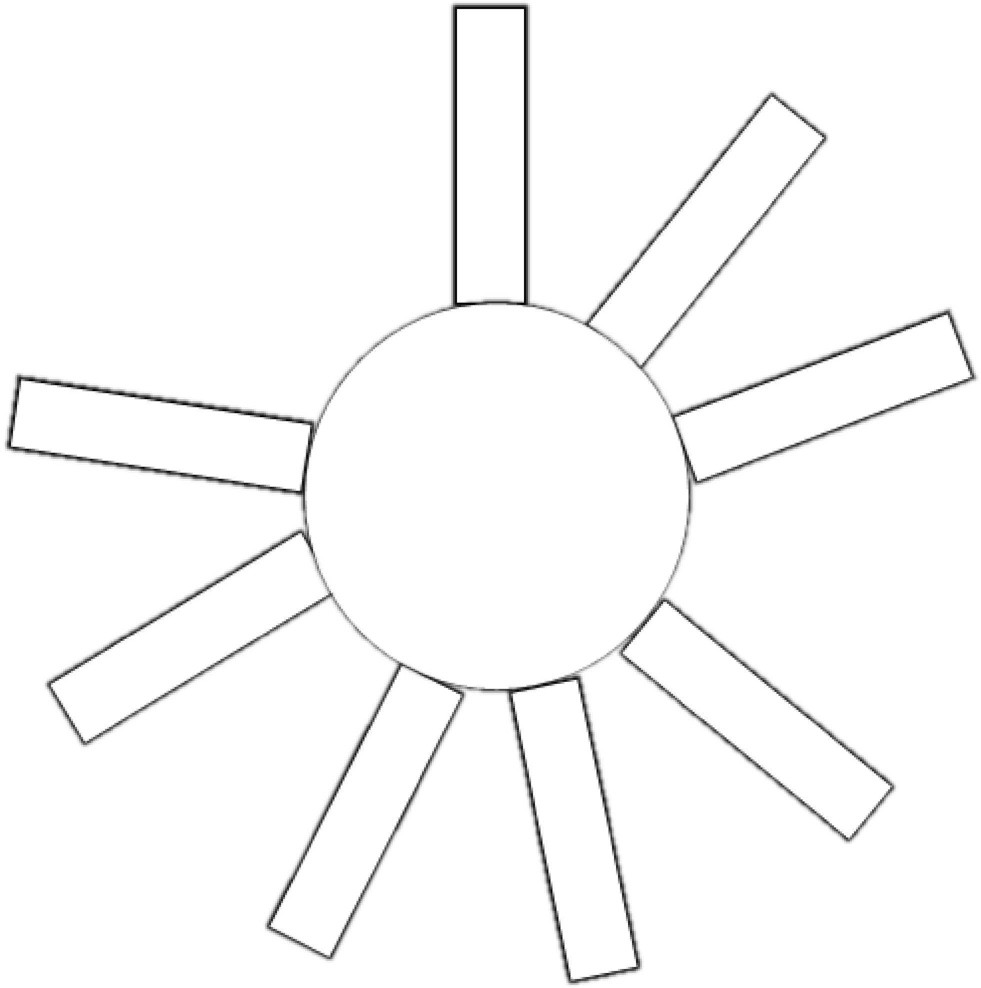


A set of m **items**(balls) need to be assigned to n **servers**(bins)

- Goals:
  - Minimize load (the number of items in each server) of servers AND access cost (number of accessed servers, accessing first server costs 1)
  - Supporting dynamic item/server insertion/deletion, and item access (search)

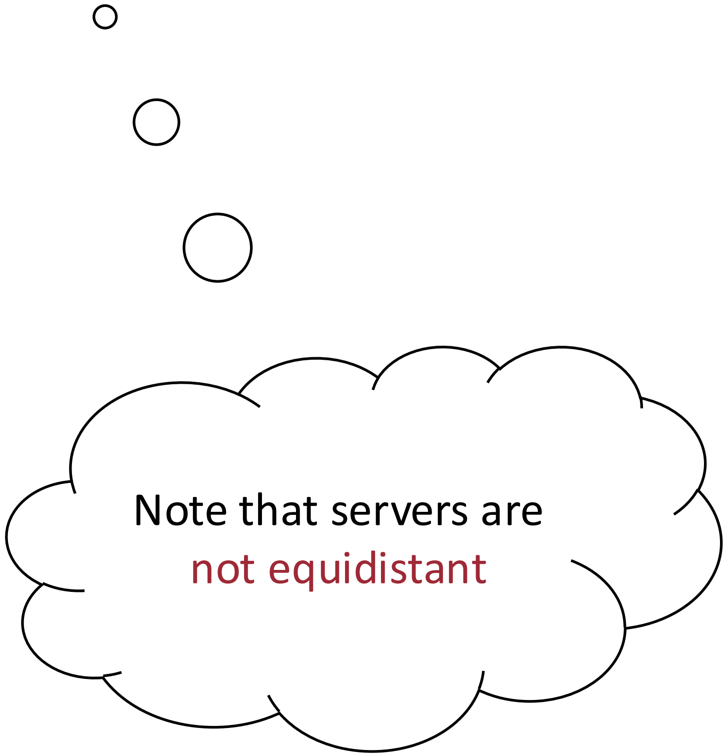
Consistent Hashing Model

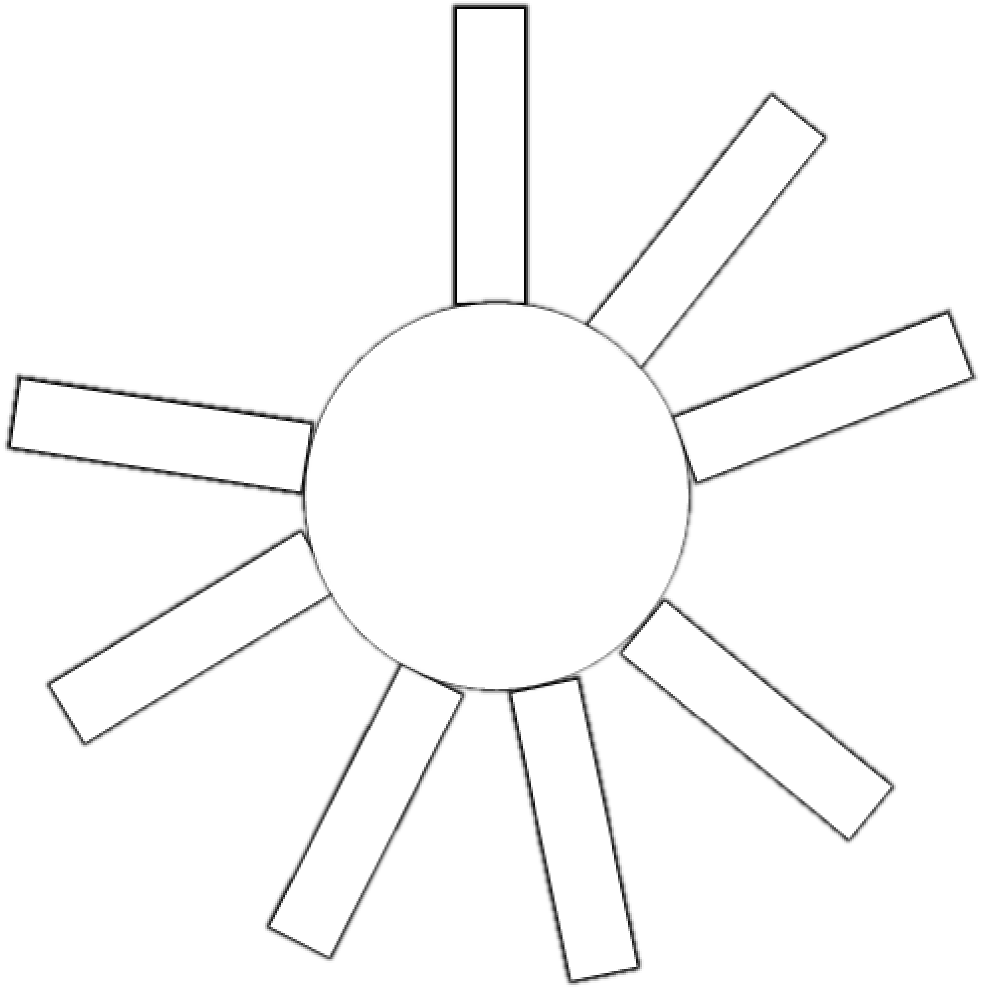
- Hash servers to a circle



Consistent Hashing Model

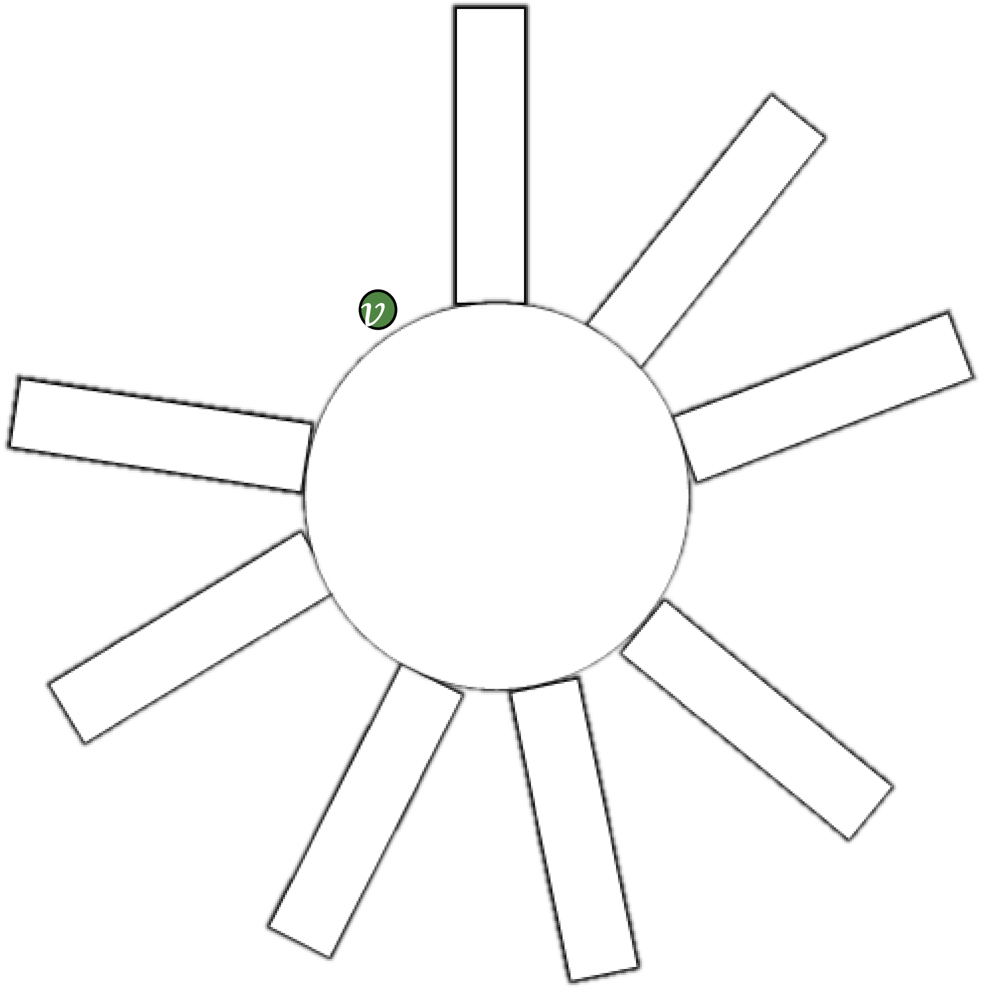
- Hash servers to a circle





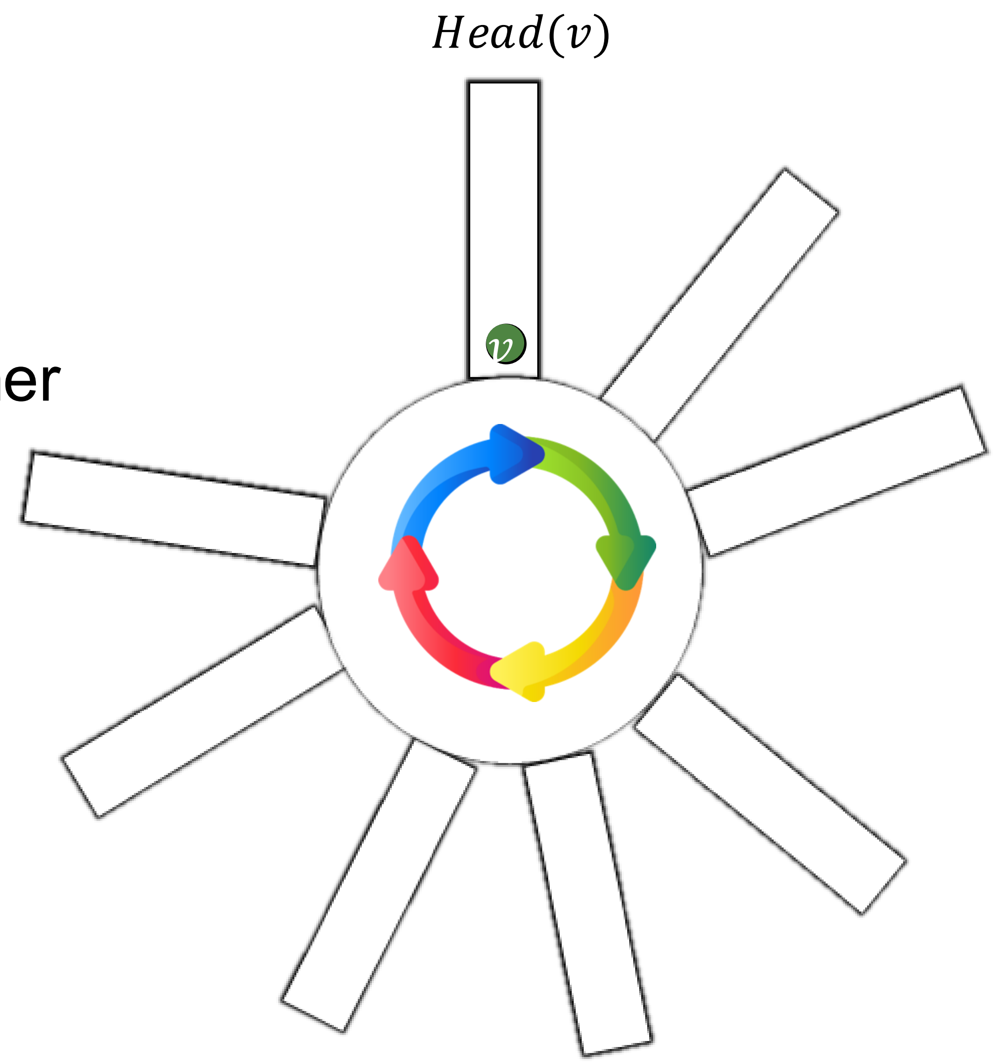
Consistent Hashing Model

- Hash servers to a circle
- Hash items to a circle



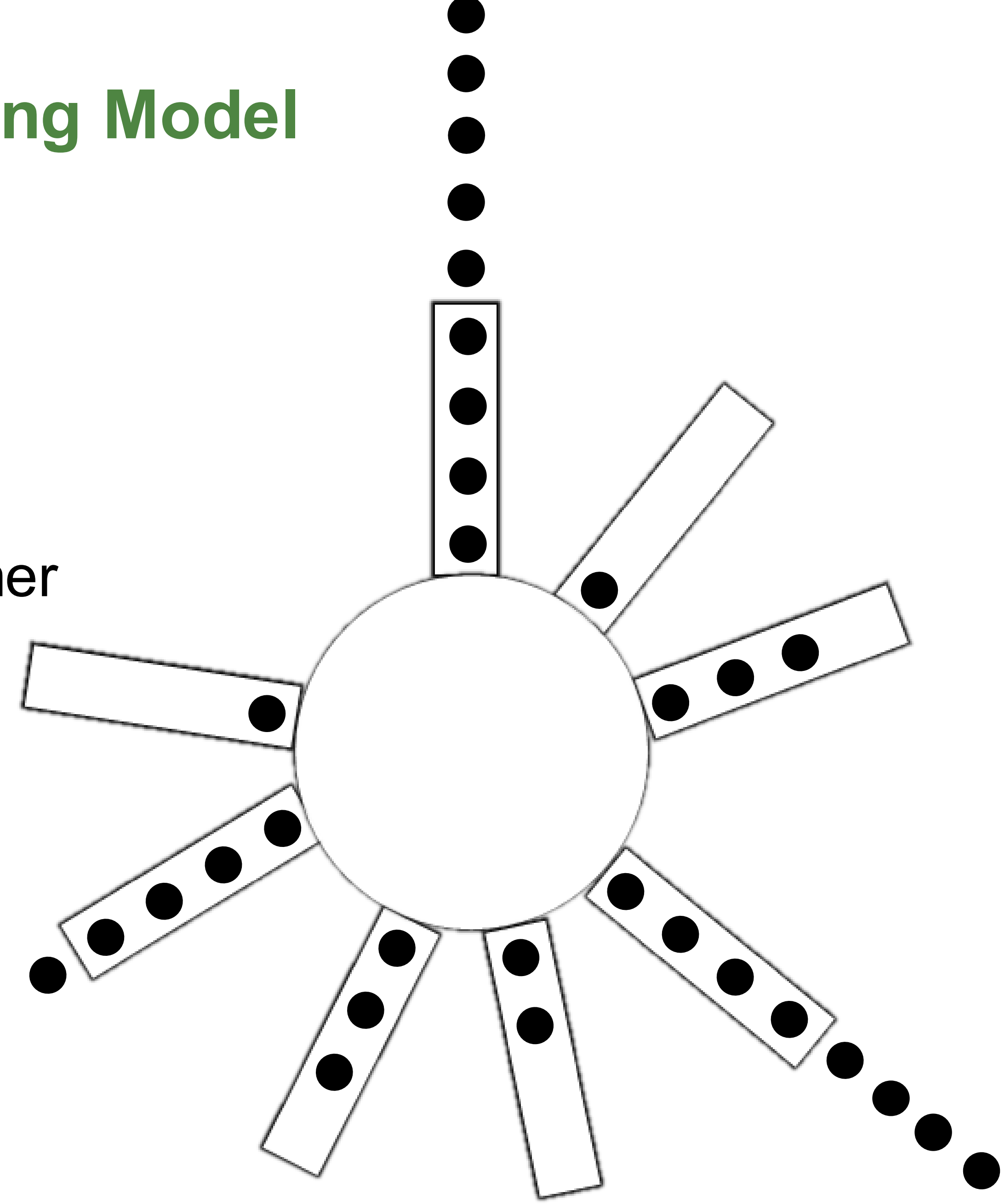
Consistent Hashing Model

- Hash servers to a circle
- Hash items to a circle
- Assign items to servers in a clockwise manner



Consistent Hashing Model

- Hash servers to a circle
- Hash items to a circle
- Assign items to servers in a clockwise manner



Consistent Hashing Model

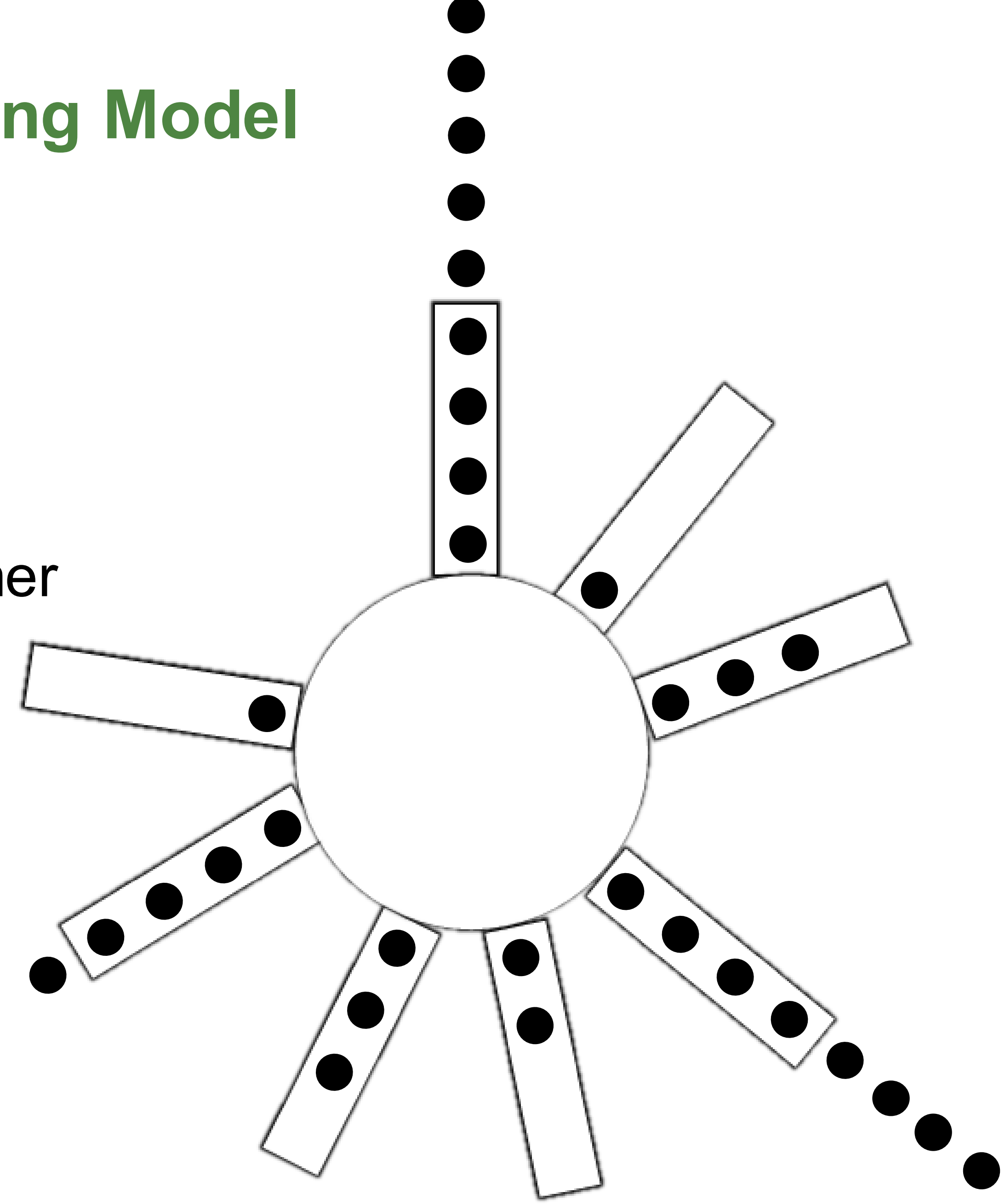
- Hash servers to a circle

- Hash items to a circle

- Assign items to servers in a clockwise manner

Challenge:

- $O(\frac{\log n}{\log \log n})$ w.h.p. gap between min and max load



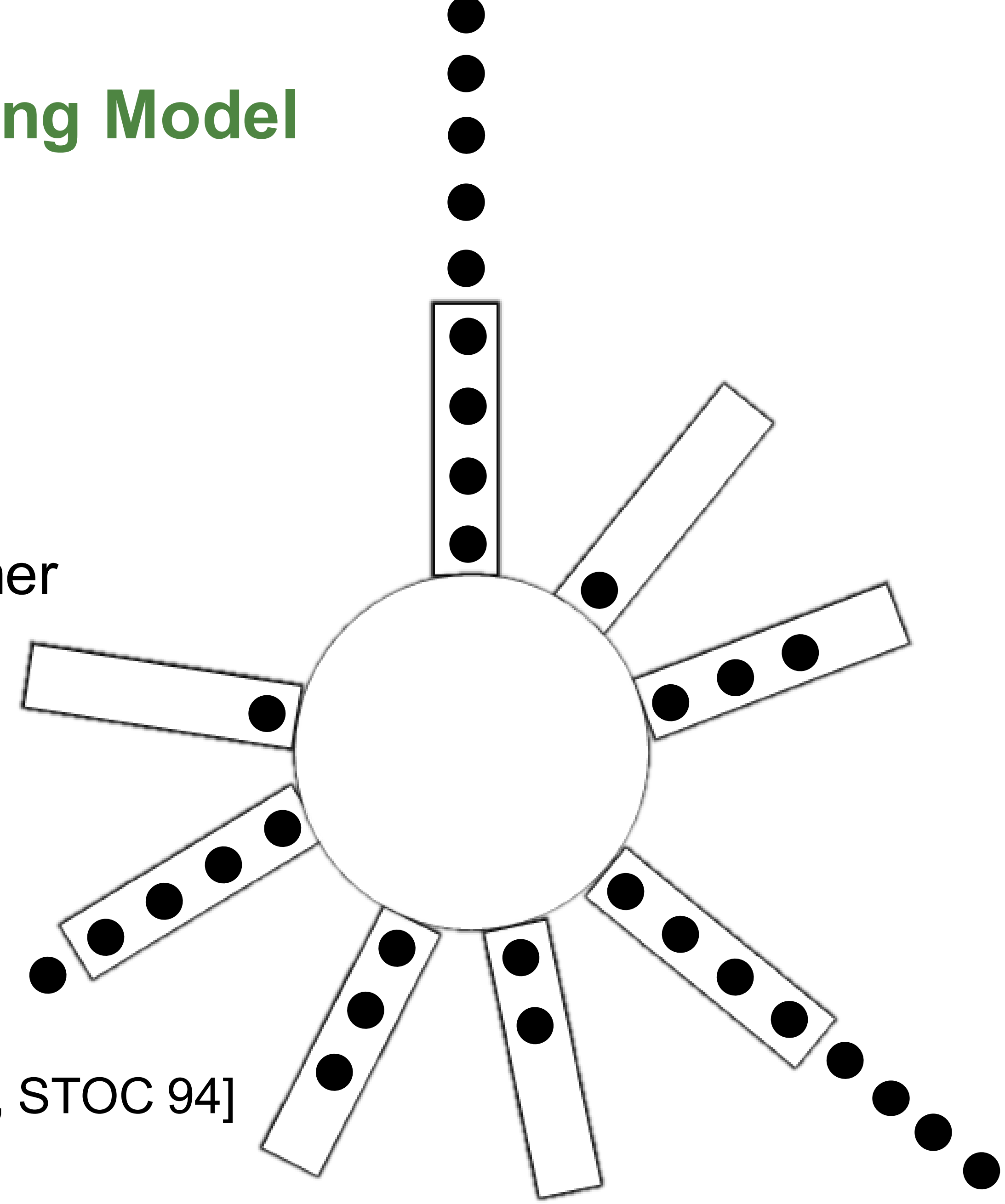
Consistent Hashing Model

- Hash servers to a circle
- Hash items to a circle
- Assign items to servers in a clockwise manner

Challenge:

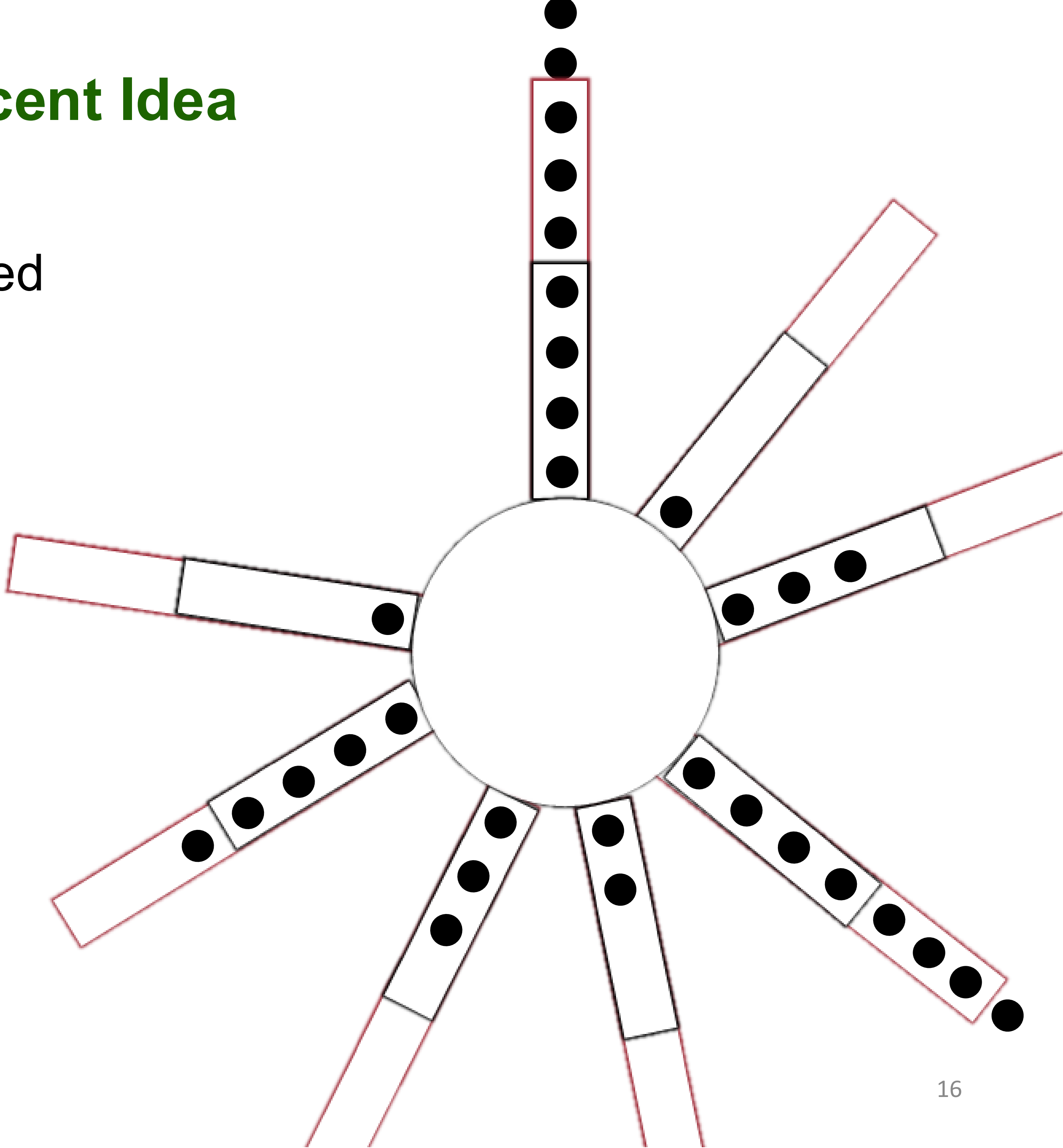
- $O(\frac{\log n}{\log \log n})$ w.h.p. gap between min and max load

- Other works reduced the gap:
  - E.g., by using “power of two choices” [Azar et al., STOC 94]
  - But the gap is still non-constant



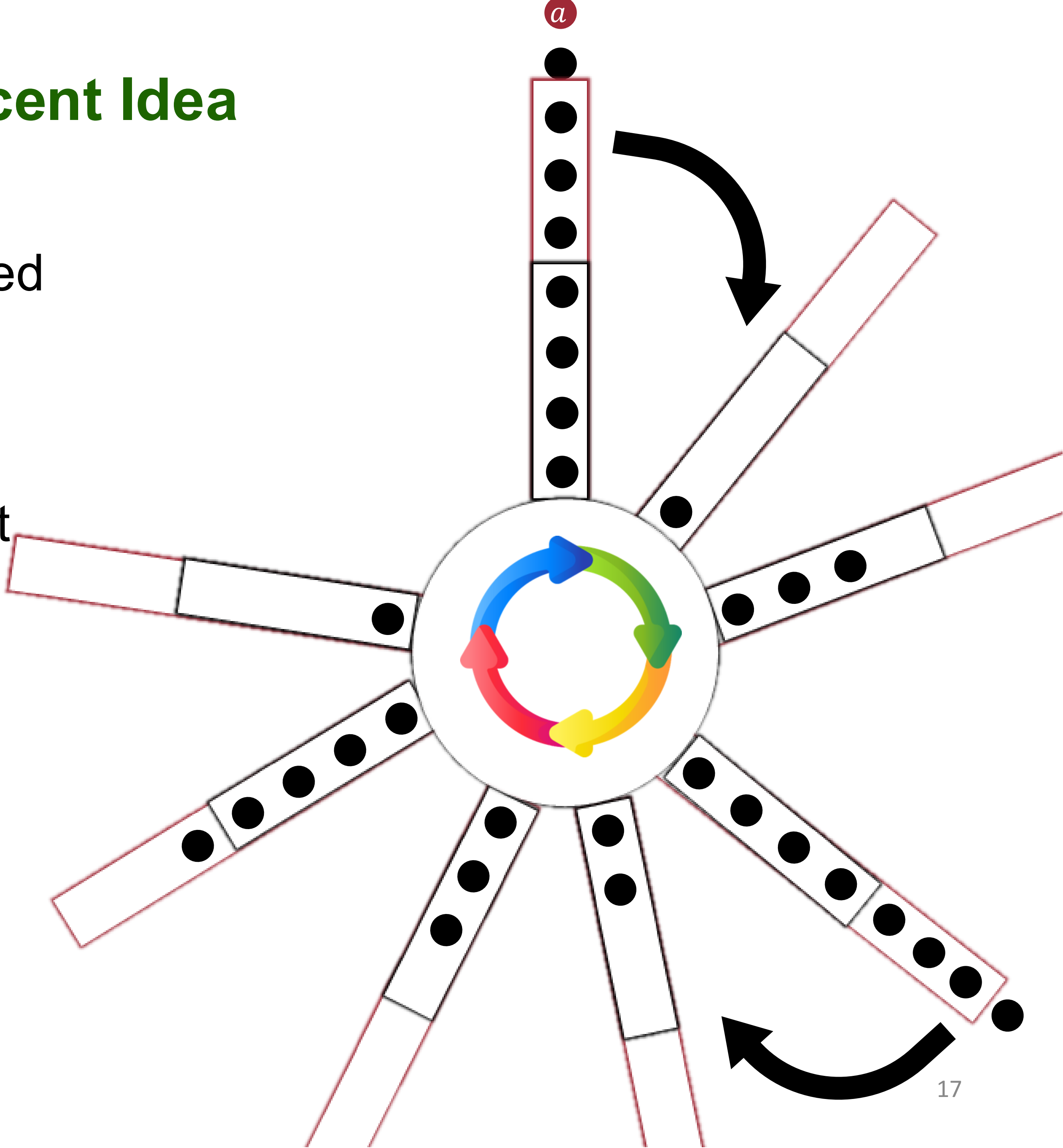
A Recent Idea

- [Mirrokni et al., SODA 2018] suggested bounding capacity to $c = \left\lceil \frac{m}{n} \right\rceil * \beta$



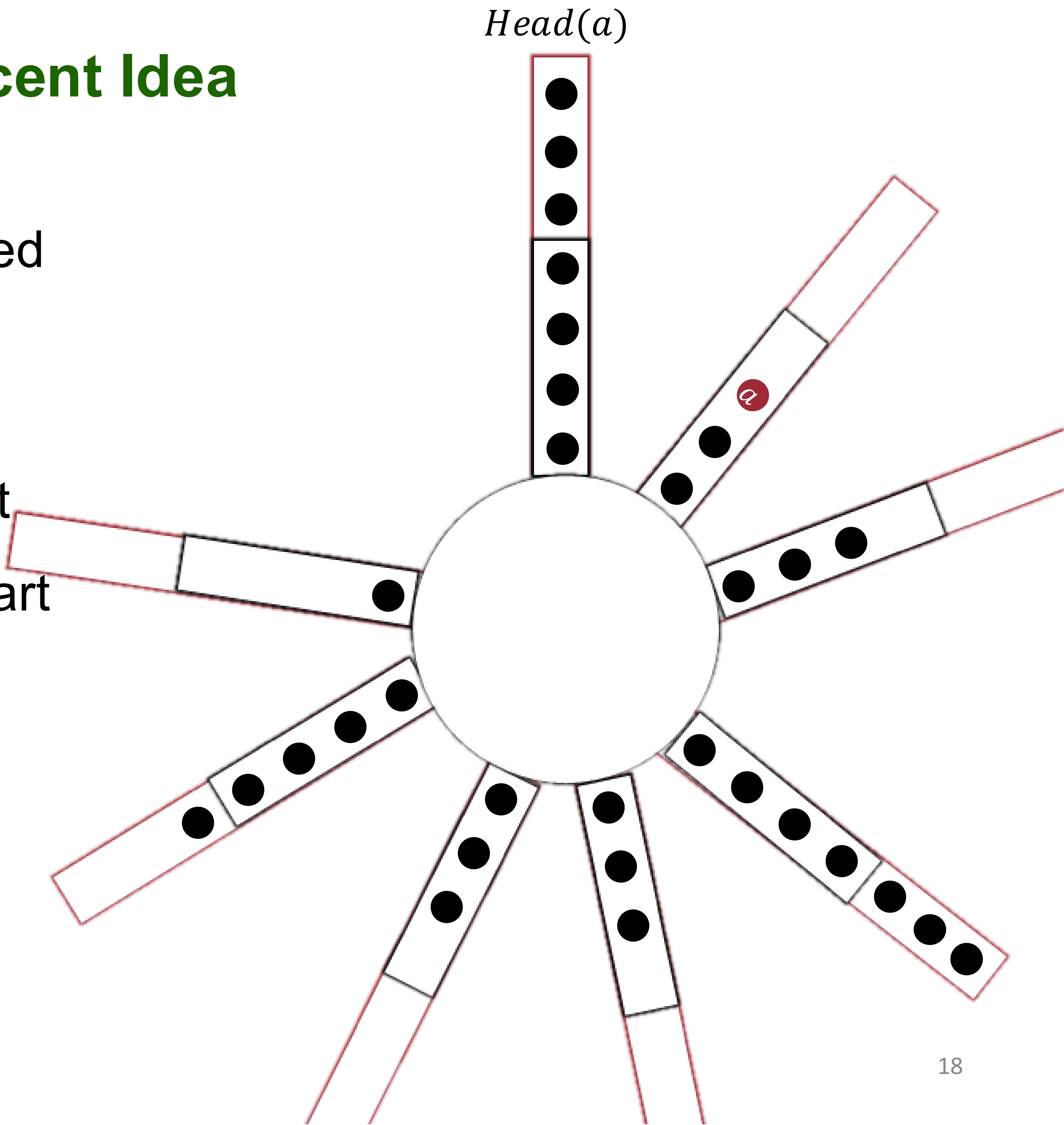
A Recent Idea

- [Mirrokni et al., SODA 2018] suggested bounding capacity to $c = \left\lceil \frac{m}{n} \right\rceil * \beta$
- Allow extra items to be moved to next servers



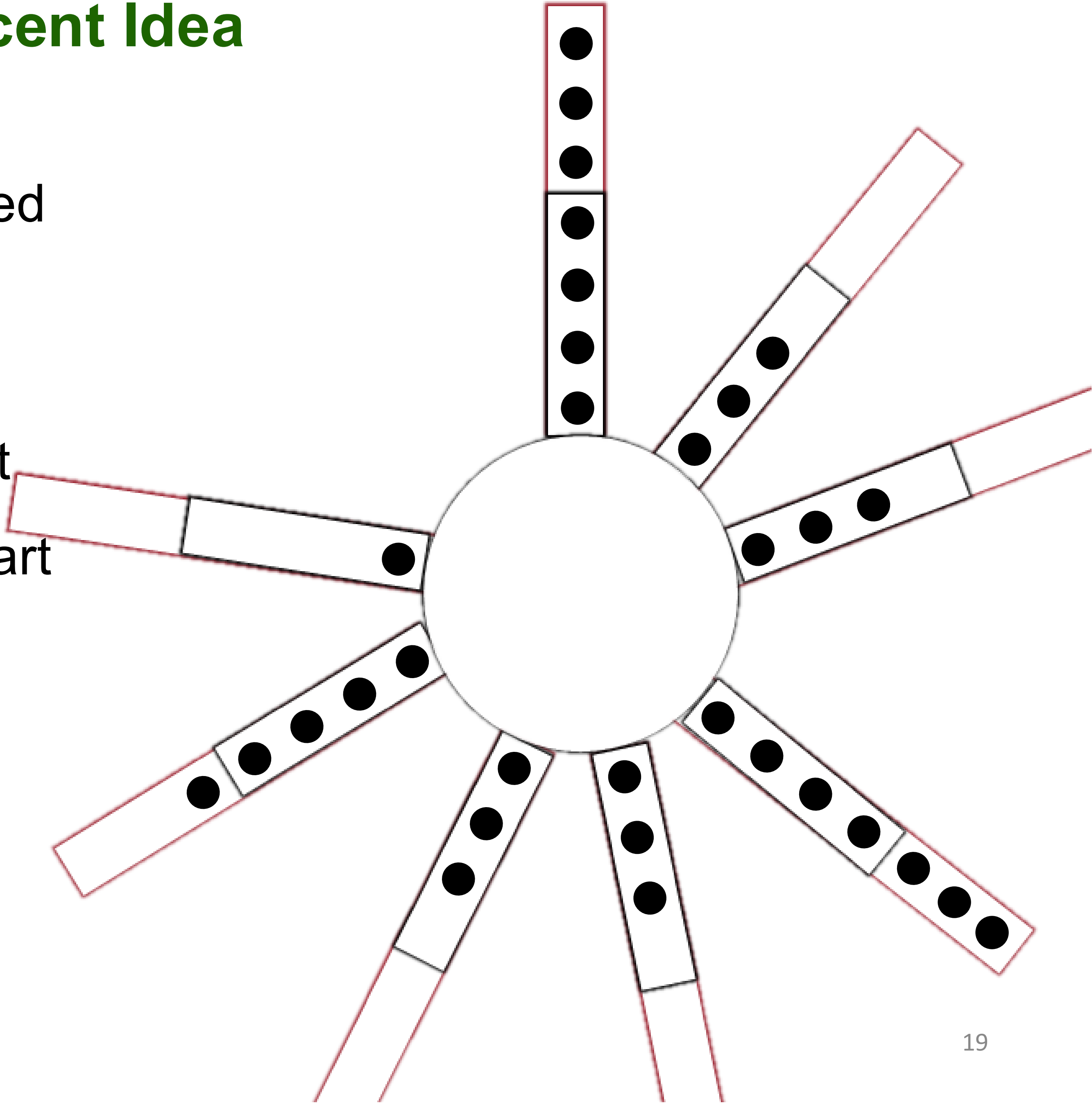
A Recent Idea

- [Mirrokni et al., SODA 2018] suggested bounding capacity to $c = \left[\frac{m}{n}\right] * \beta$
- Allow extra items to be moved to next servers, and to access a node, we start from its head



A Recent Idea

- [Mirrokni et al., SODA 2018] suggested bounding capacity to $c = \left\lceil \frac{m}{n} \right\rceil * \beta$
- Allow extra items to be moved to next servers, and to access a node, we start from its head

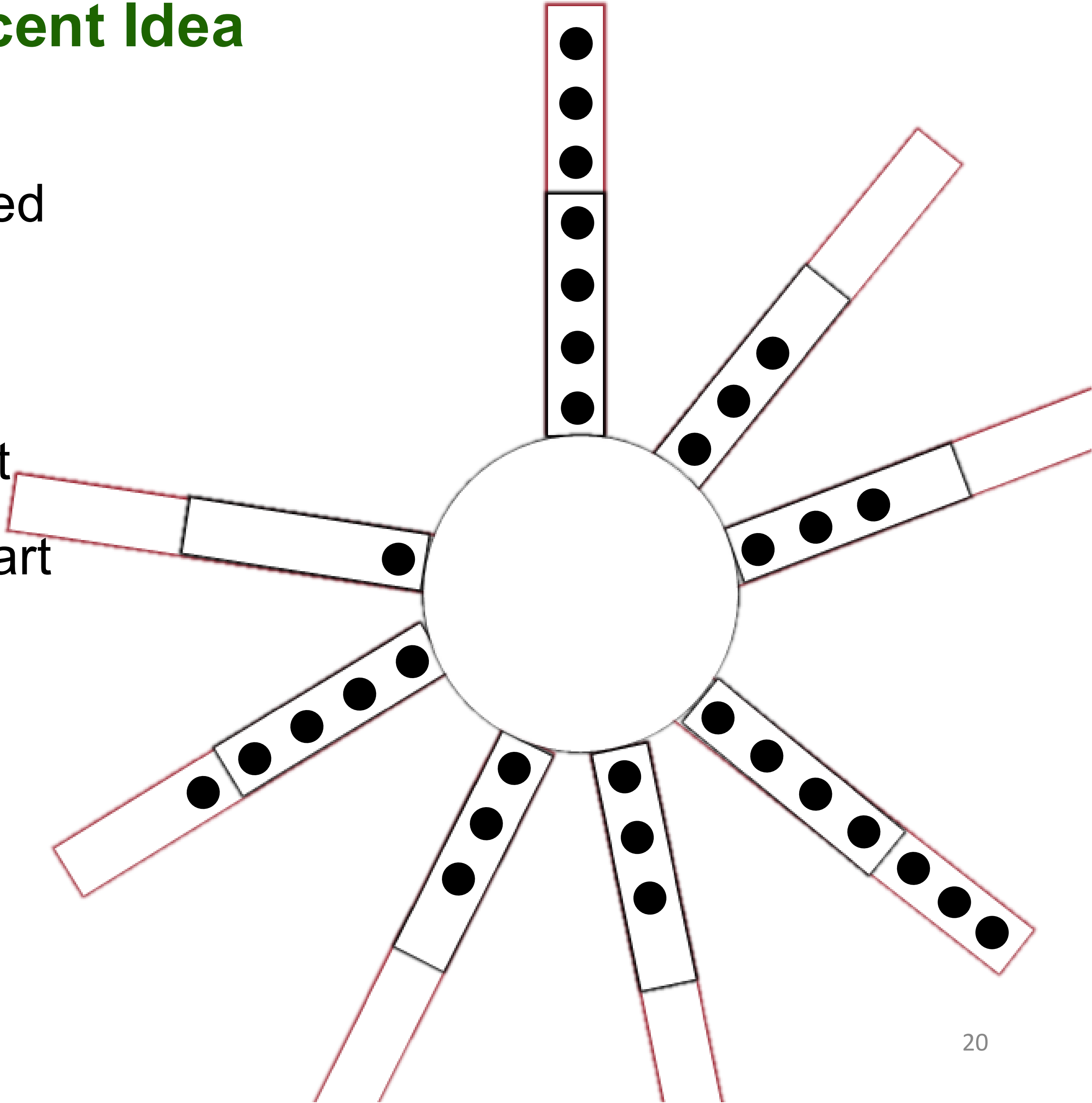


Challenges:

- Storage utilization is far from optimal

A Recent Idea

- [Mirrokni et al., SODA 2018] suggested bounding capacity to $c = \left\lceil \frac{m}{n} \right\rceil * \beta$
- Allow extra items to be moved to next servers, and to access a node, we start from its head

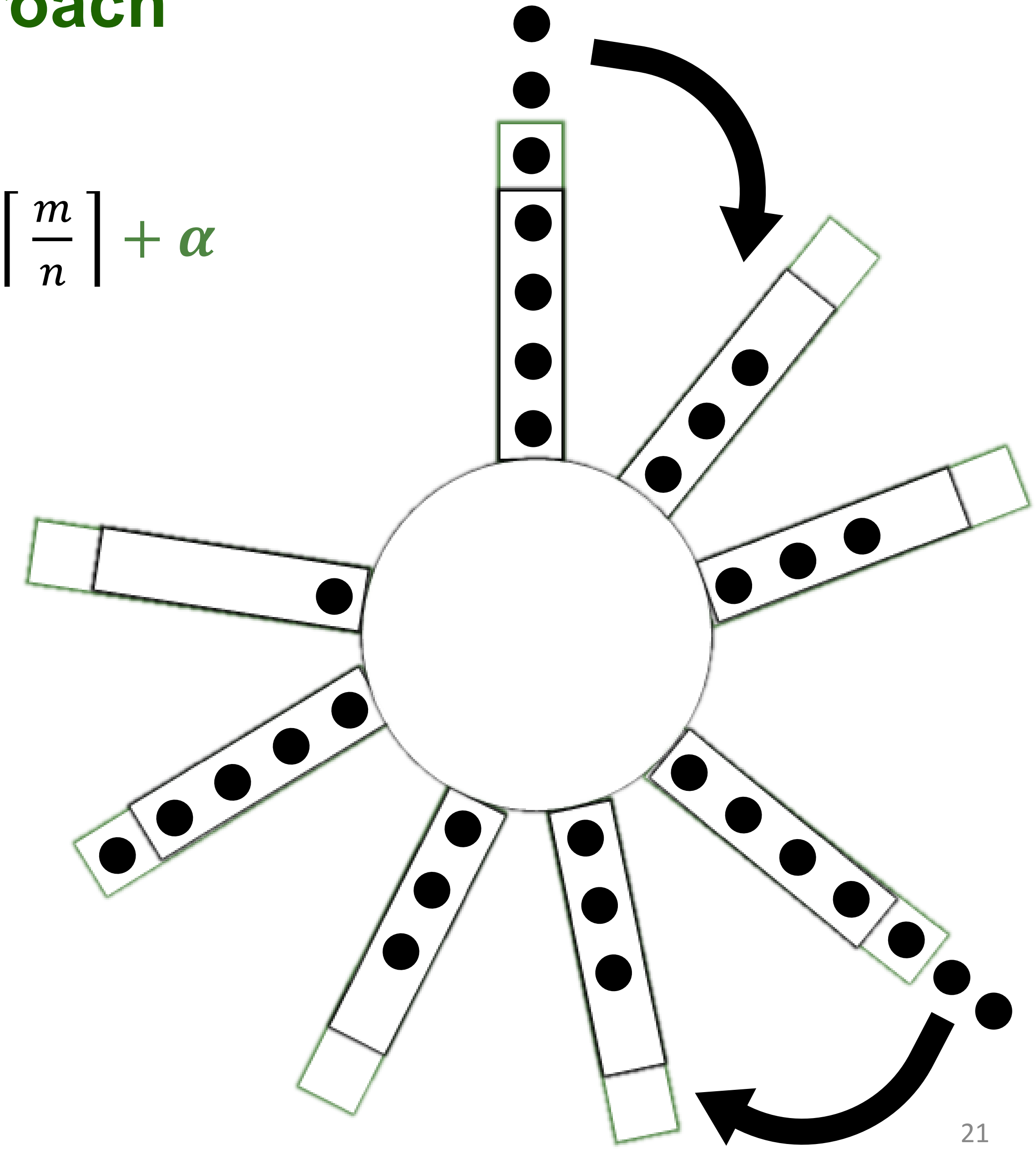


Challenges:

- Storage utilization is far from optimal
- Access cost is increased significantly

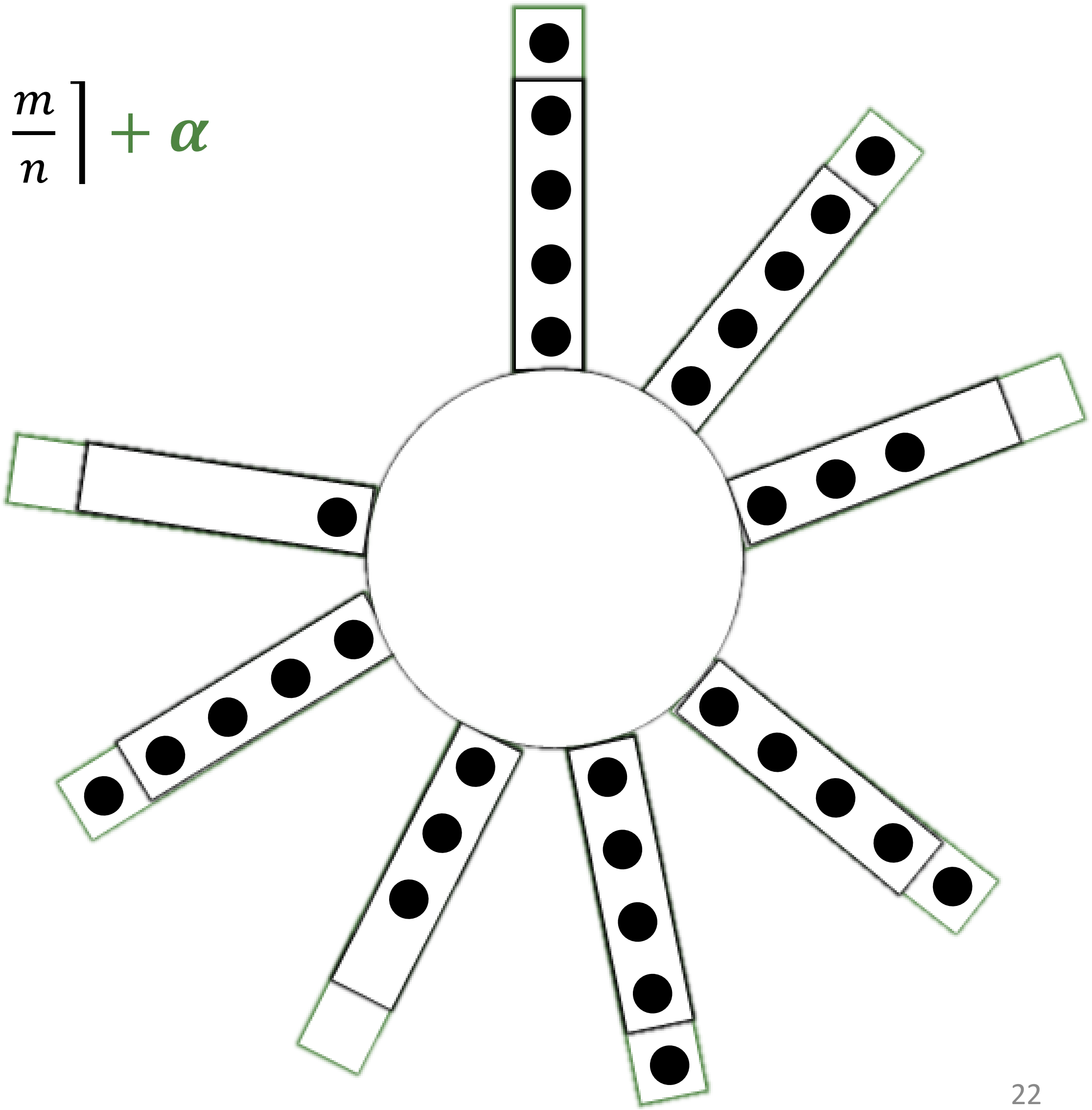
Our Approach

- Each server has a bounded capacity $c = \left\lceil \frac{m}{n} \right\rceil + \alpha$



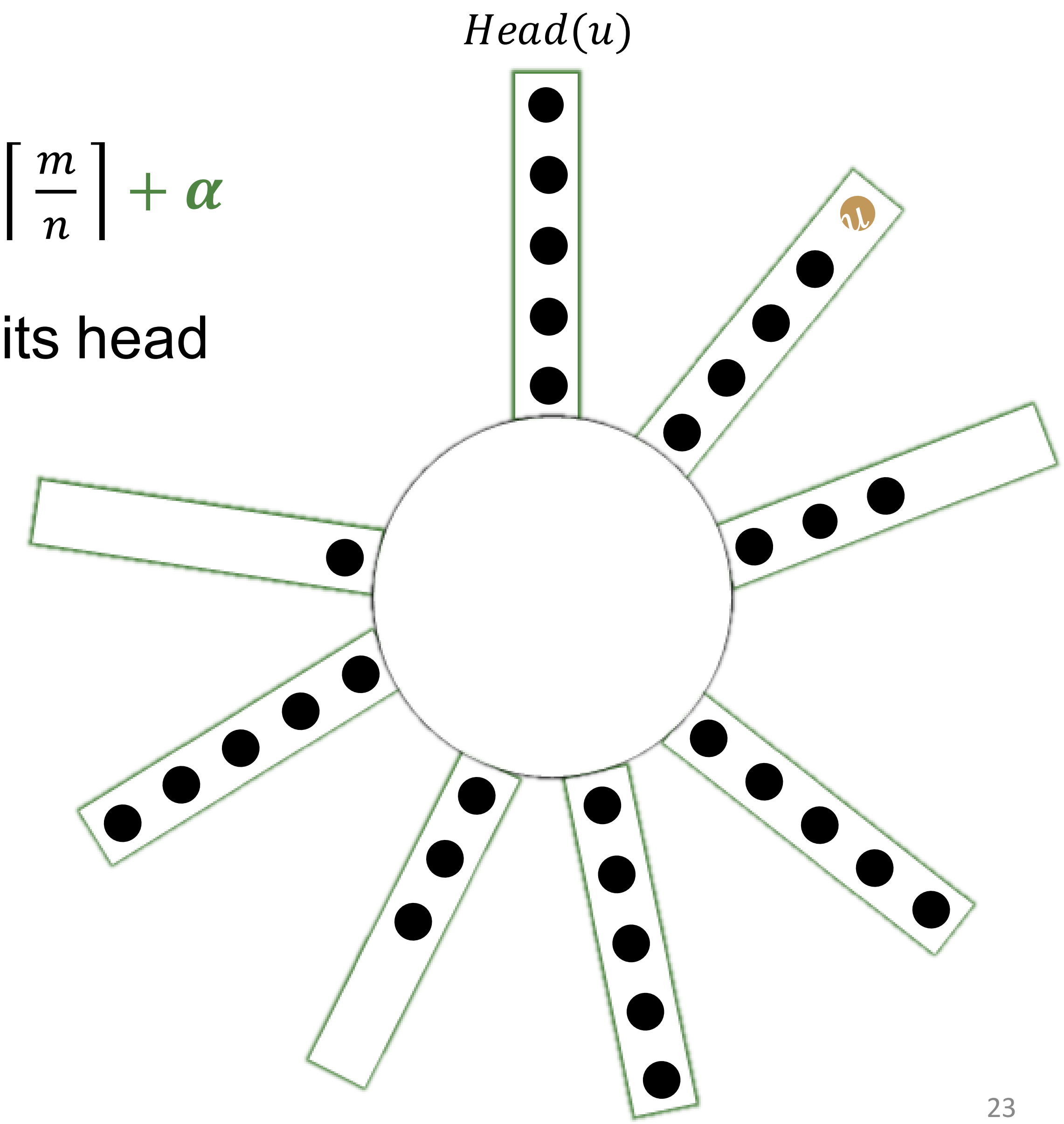
Our Approach

- Each server has a bounded capacity $c = \left\lceil \frac{m}{n} \right\rceil + \boldsymbol{\alpha}$



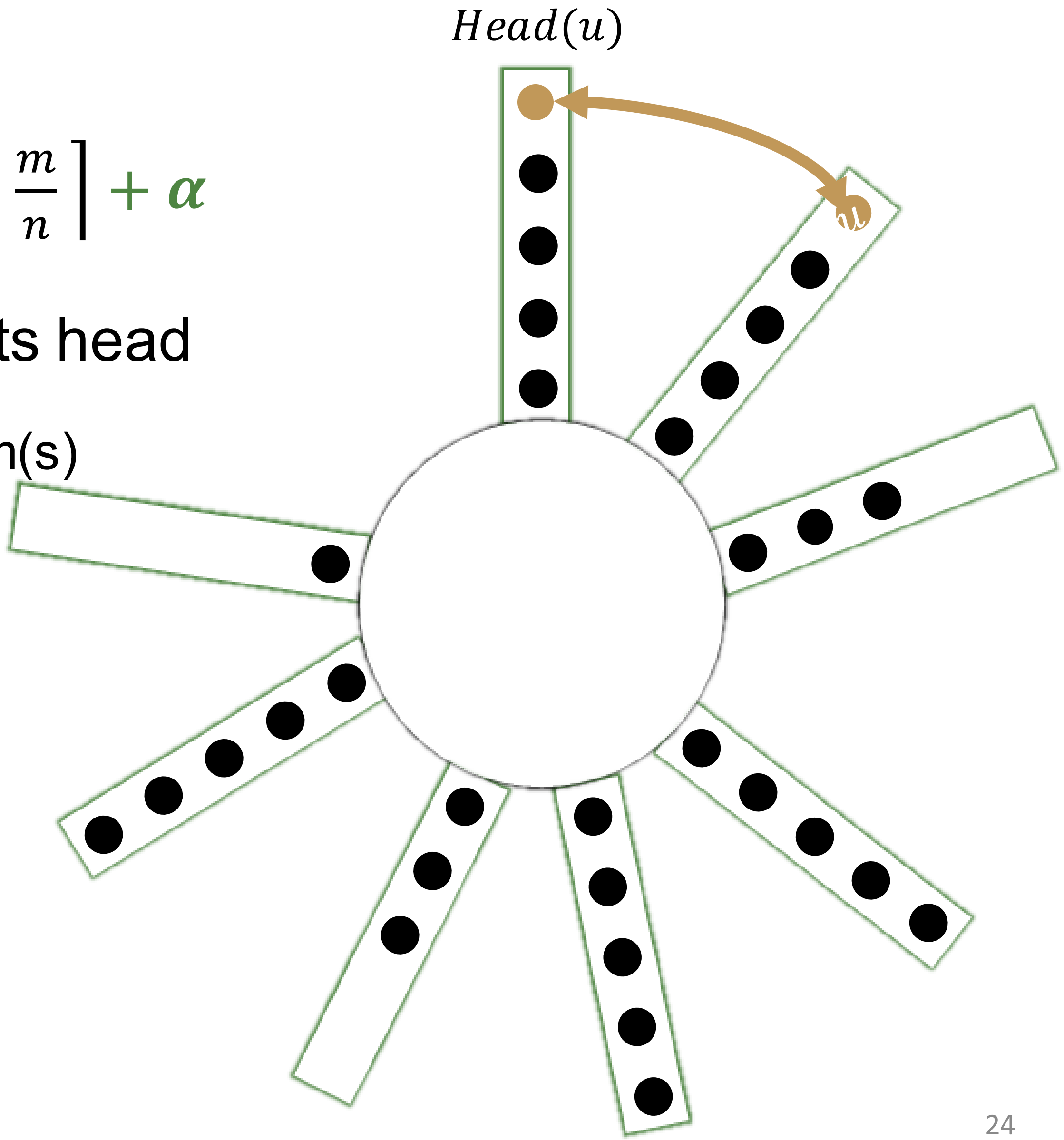
Our Approach

- Each server has a bounded capacity $c = \left\lceil \frac{m}{n} \right\rceil + \boldsymbol{\alpha}$
- Bring the recently accessed item back to its head



Our Approach

- Each server has a bounded capacity $c = \left\lceil \frac{m}{n} \right\rceil + \alpha$
- Bring the recently accessed item back to its head
  - By swapping with (the) least recently used item(s)

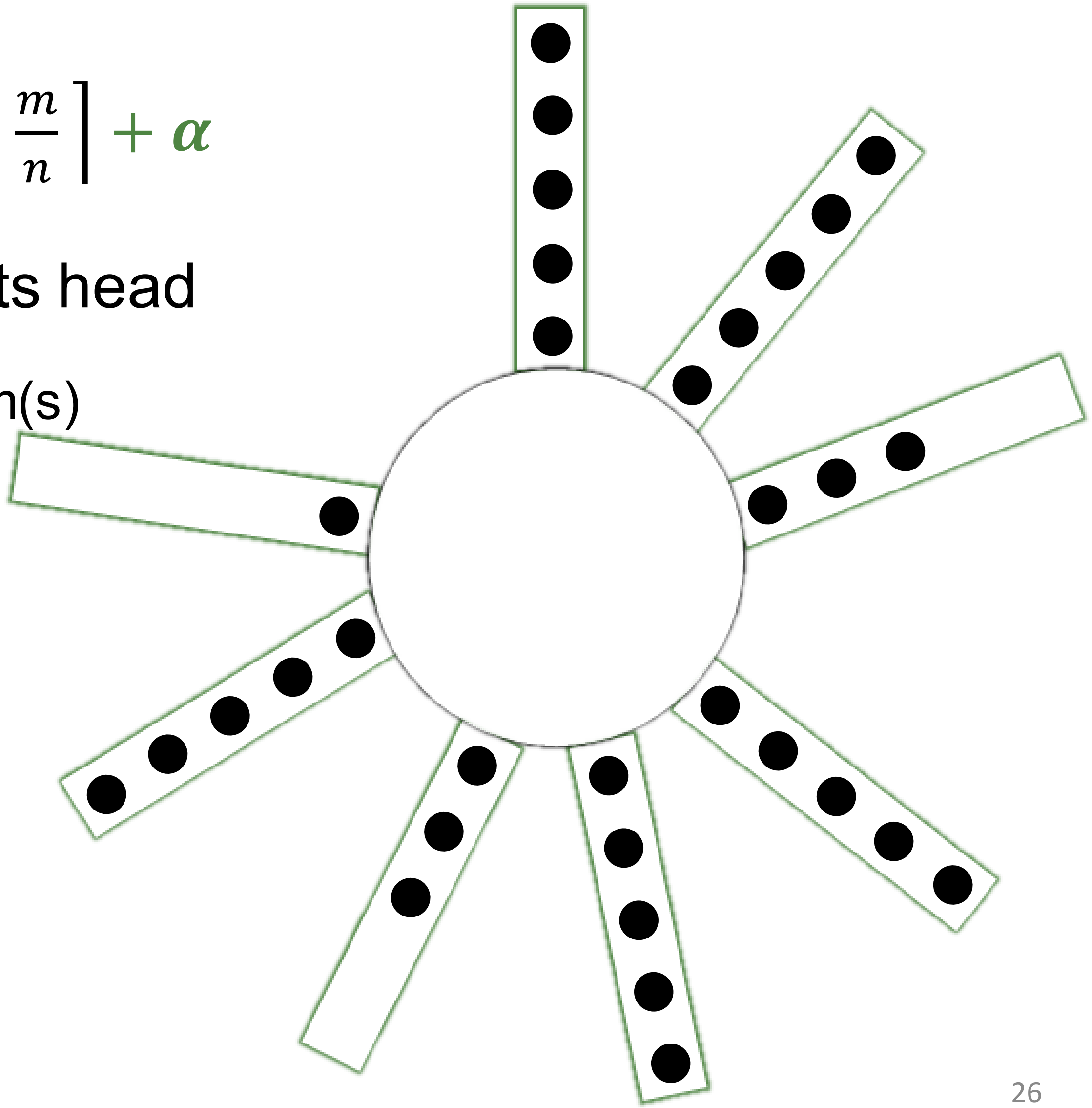


Our Approach

- Each server has a bounded capacity $c = \left\lceil \frac{m}{n} \right\rceil + \boldsymbol{\alpha}$
- Bring the recently accessed item back to its head
  - By swapping with (the) least recently used item(s)

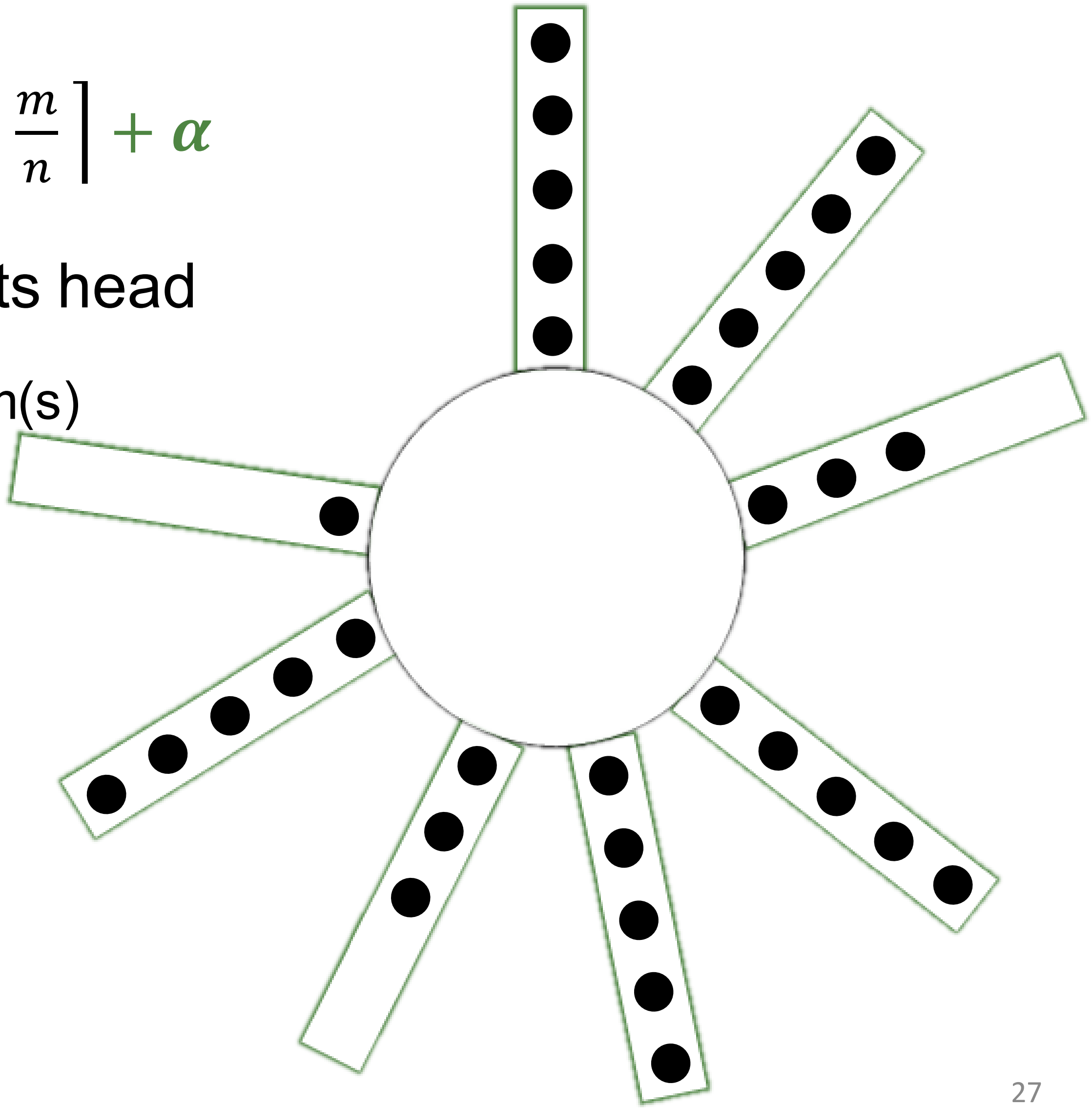
Benefits:

- Storage utilization is near-optimal



Our Approach

- Each server has a bounded capacity $c = \left\lceil \frac{m}{n} \right\rceil + \boldsymbol{\alpha}$
- Bring the recently accessed item back to its head
  - By swapping with (the) least recently used item(s)



Benefits:

- Storage utilization is near-optimal
- Our approach is constant competitive*
  - $Cost$: access + reconfiguration cost
  - $Cost_{ALG} \leq \alpha \cdot Cost_{OPT}$

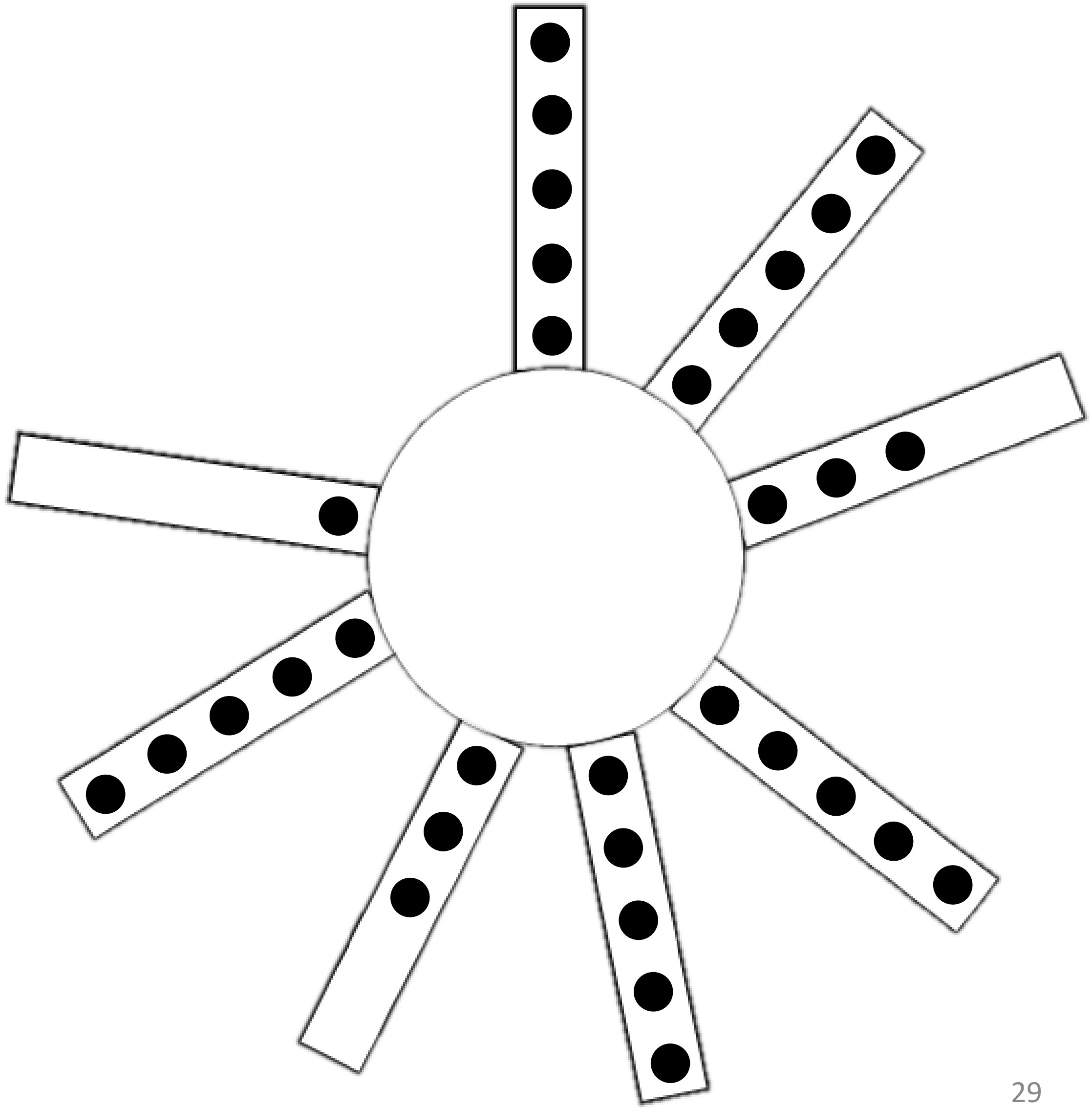
  * Given well-behaved inputs

Related work Recap

Data structure	Access Cost	Storage Utilization
Traditional [Karger et al., STOC 1997]	Low	Low
With Bounded Loads [Mirrokni et al., SODA 2018]	High	Medium
Our Work [Hash & Adjust]	Low	High

Analysis: Access Operations

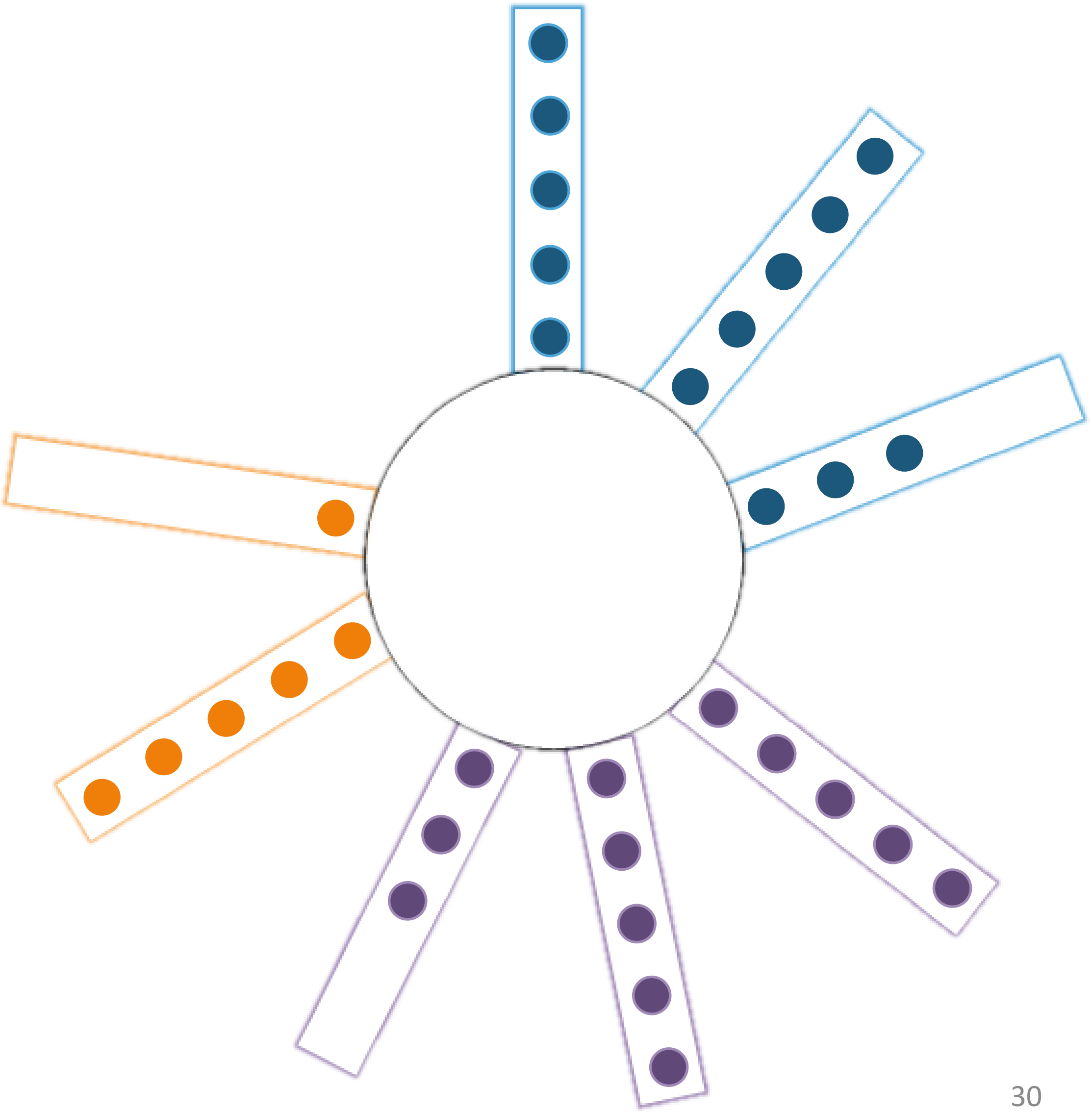
Thm 1: Considering access requests, our algorithm is $2 \cdot (1 + \omega)$ competitive, where ω is the constant cost of moving items between two adjacent servers.



Analysis: Access Operations

Thm 1: Considering access requests, our algorithm is $2 \cdot (1 + \omega)$ competitive, where ω is the constant cost of moving items between two adjacent servers.

Lem 1: Decomposing into *Serverlists* is always possible

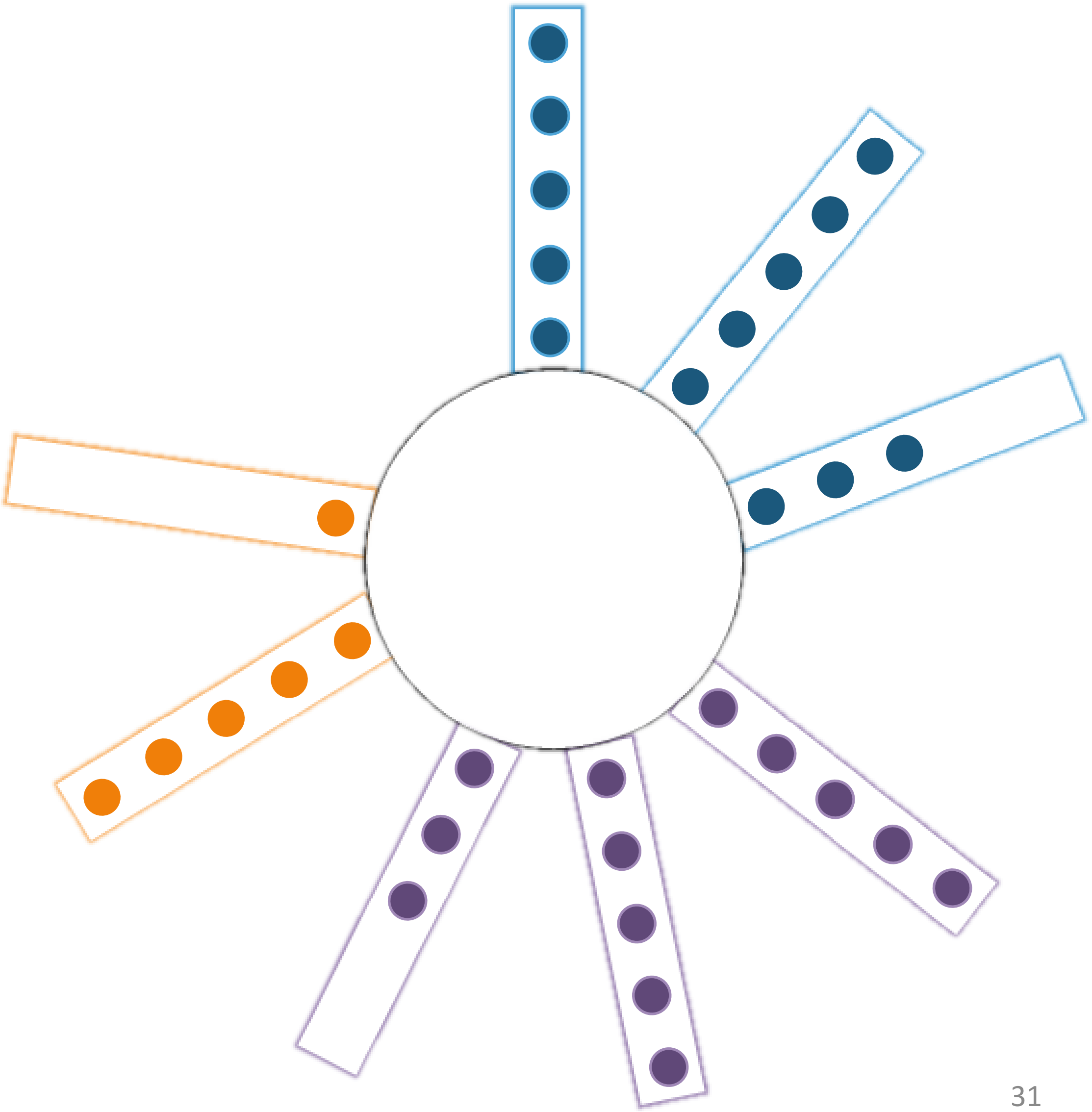


Analysis: Access Operations

Thm 1: Considering access requests, our algorithm is $2 \cdot (1 + \omega)$ competitive, where ω is the constant cost of moving items between two adjacent servers.

Lem 1: Decomposing into *Serverlists* is always possible

Proof idea: because of extra capacity, we always have non-full servers, and we can not jump over them

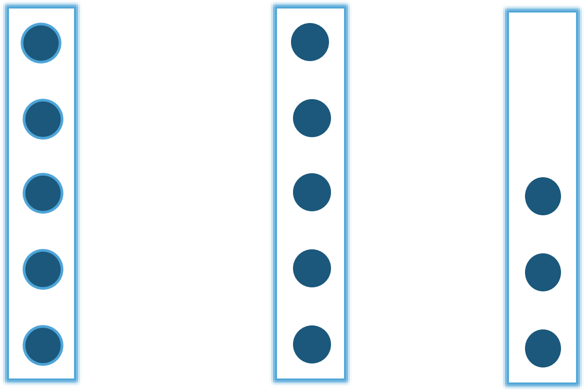


Analysis: Access Operations

Thm 1: Considering access requests, our algorithm is $2 \cdot (1 + \omega)$ competitive, where ω is the constant cost of moving items between two adjacent servers.

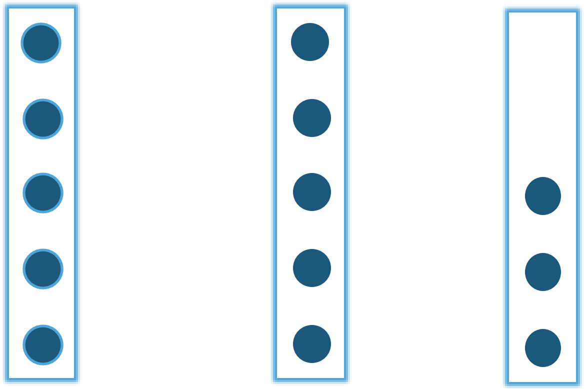
Lem 1: Decomposing into *Serverlists* is always possible

Lem 2: Access cost inside each *Serverlist* is constant competitive



Analysis: Access Operations

Thm 1: Considering access requests, our algorithm is $2 \cdot (1+\omega)$ competitive, where ω is the constant cost of moving items between two adjacent servers.



Lem 1: Decomposing into *Serverlists* is always possible

Lem 2: Access cost inside each *Serverlist* is constant competitive

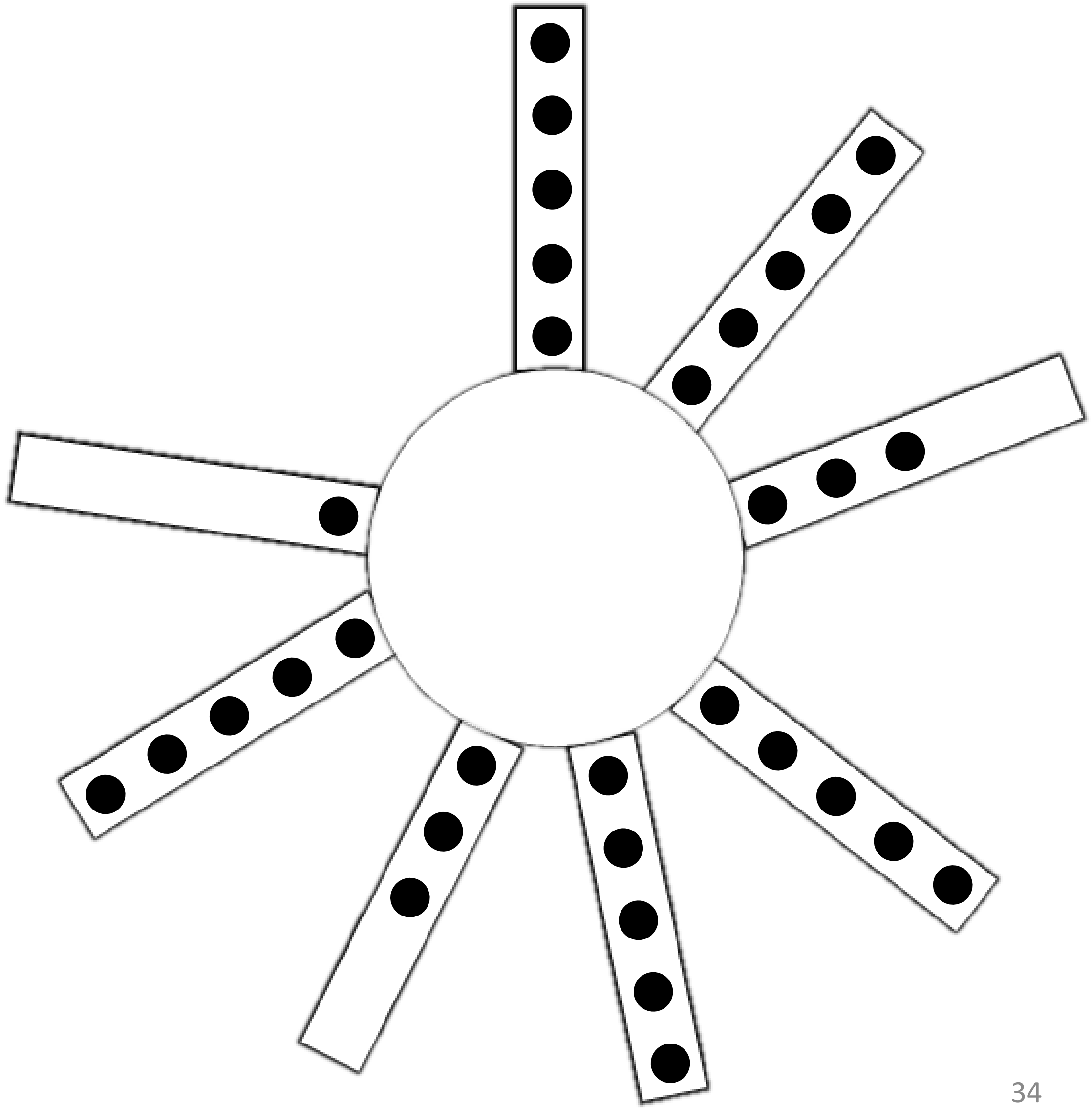
Proof idea: Potential function analysis based on inversions

Analysis: Other Operations

Well-behaved request sequence:

A sequence that insertion/deletions happen after each $\sum_{i=1}^{n-1} e^{\frac{\alpha^2}{m^2}(i+1)}$ access request

Thm 2: Hash & Adjust is constant competitive, in expectation, considering a well-behaved request sequence.



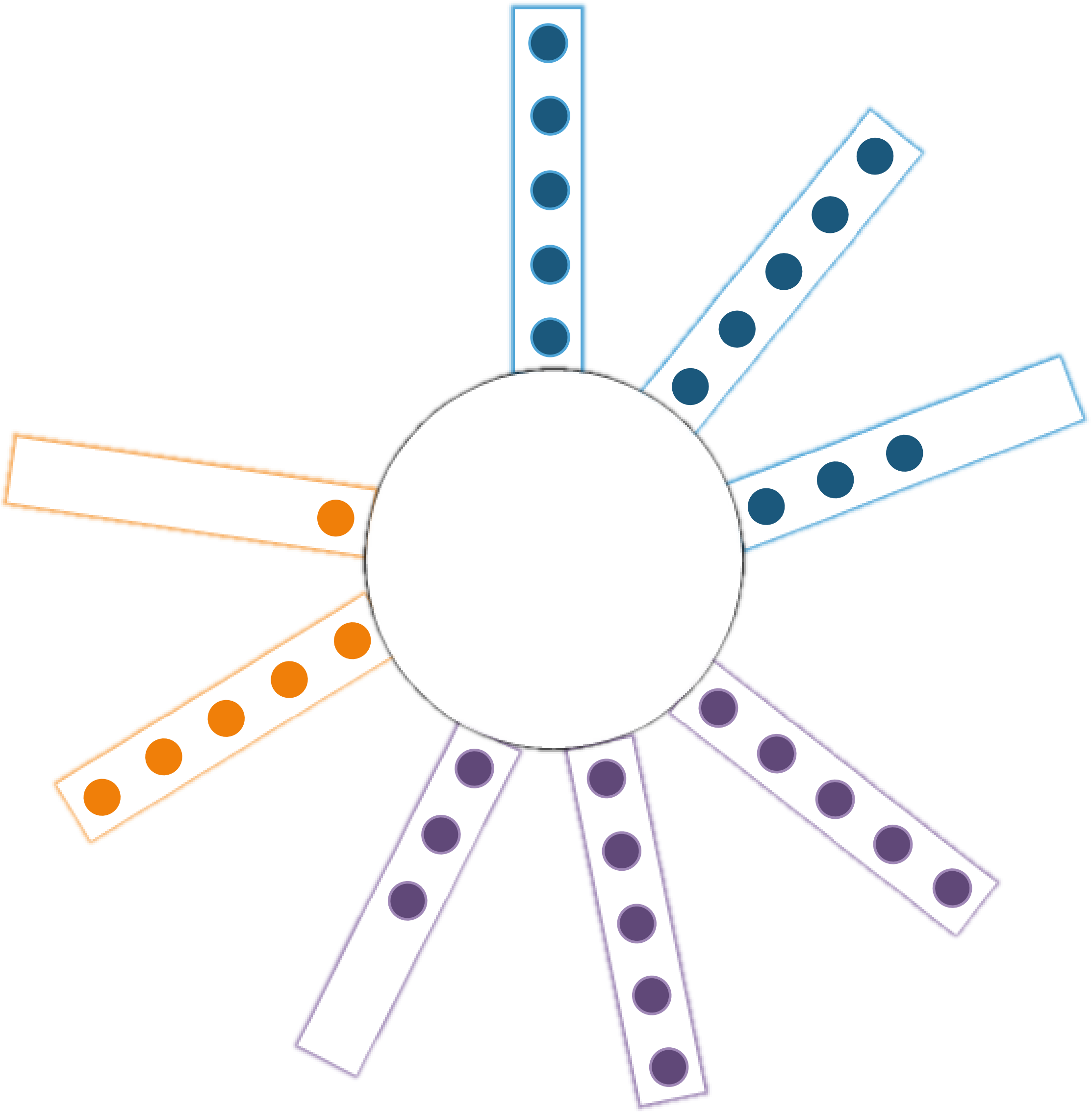
Analysis: Other Operations

Well-behaved request sequence:

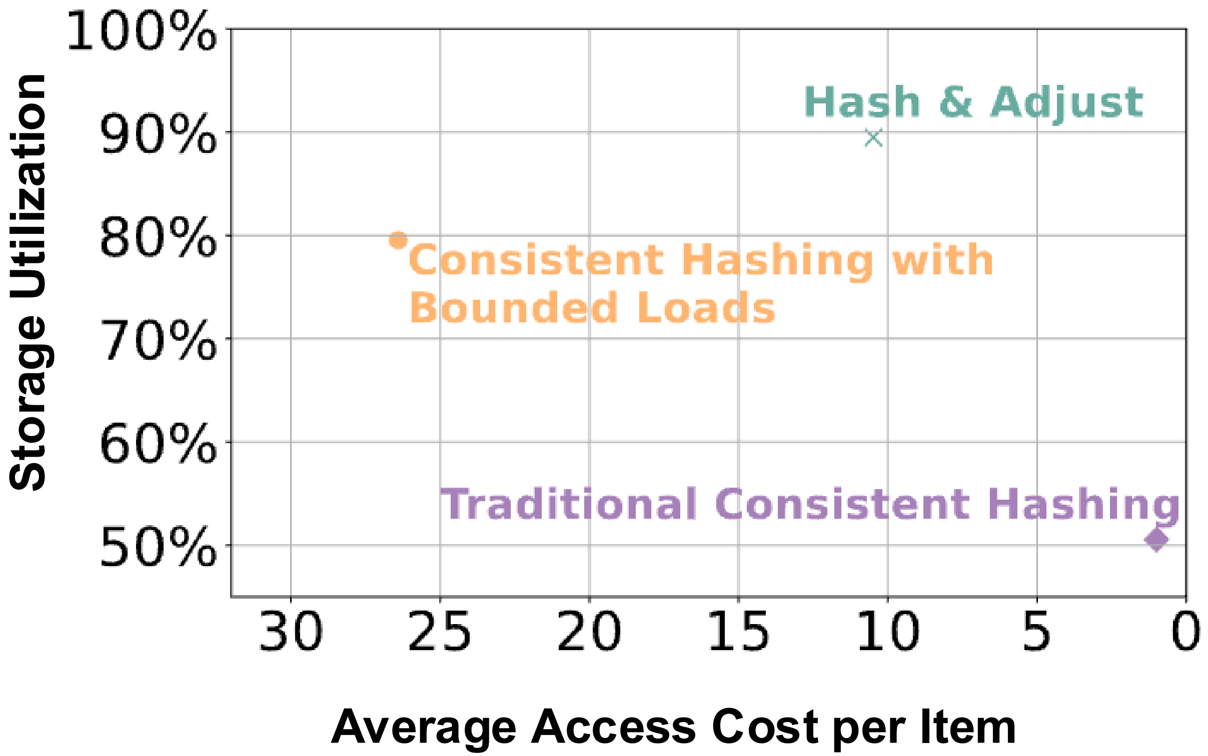
A sequence that insertion/deletions happen after each $\sum_{i=1}^{n-1} e^{\frac{\alpha^2}{m^2}(i+1)}$ access request

Thm 2: Hash & Adjust is constant competitive, in expectation, considering a well-behaved request sequence.

Proof sketch: Expected maximum length of an a *Serverlists is* $\sum_{i=1}^{n-1} e^{\frac{\alpha^2}{m^2}(i+1)}$

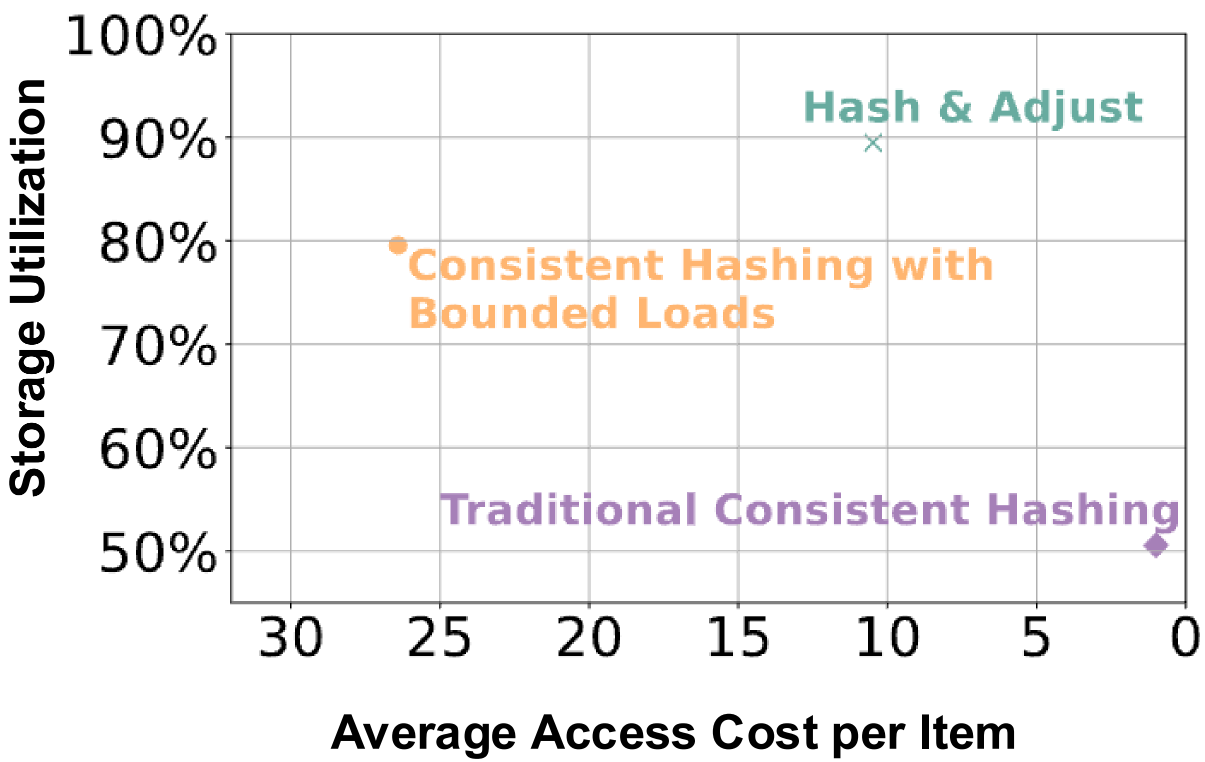


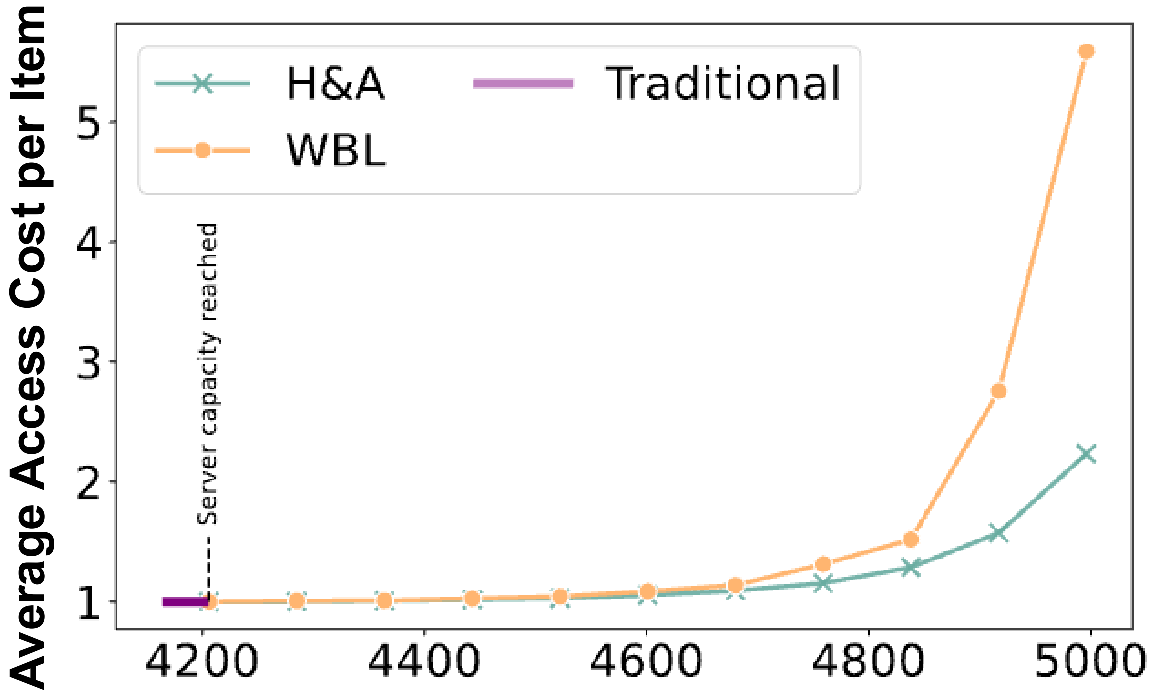
Some Empirical Results



Based on a dataset from Avazu, considering $\alpha = 4$ four our algorithm and $\beta = 1.25$ for ”with bounded load” algorithm

Some Empirical Results





Based on a dataset from Avazu, considering $\alpha = 4$ four our algorithm and $\beta = 1.25$ for ”with bounded load” algorithm

Based on a dataset from CAIDA, and fixing capacity of all algorithms

Future Work

- Rendering other distributed data structures self-adjusting
- Incorporating the algorithm in open-source load balancers like HAproxy

Full paper:
https://arxiv.org/pdf/2411.11665

Our group's website:
tu.berlin/en/eninet

My website:
pourdamghani.net