CS 7172
Parallel and Distributed Computation

Distributed Storage

Kun Suo

Computer Science, Kennesaw State University

https://kevinsuo.github.io/

Outline

- Distributed Storage
- Hashing
- Consistent hashing
- Consistent Hashing with Bounded Loads
- Consistent Hashing with Virtual Nodes

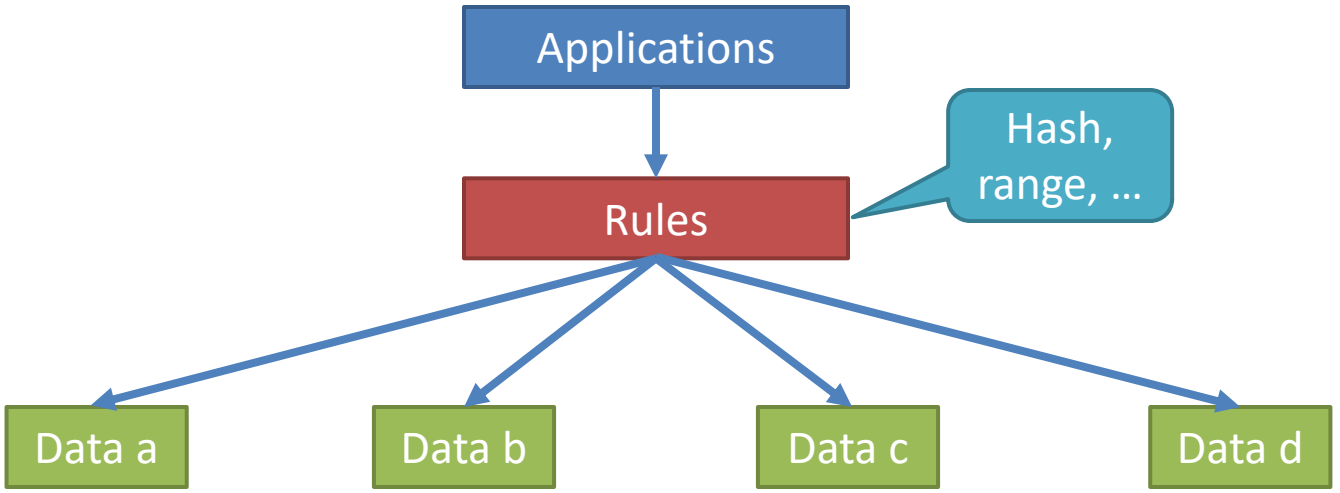
Why Distributed Storage?

- With the increase of data volume and user access, single-machine performance can no longer meet requirements

- Distributed storage has become a common method to provide large-scale applications with large-capacity, high-performance, highly available, and highly scalable storage services.

Distributed Storage

- Key idea:
  - Write: store the data on different machines according to certain rules.
  - Read: looking for certain machines according to certain rules and get the stored data.



Distributed Storage in Daily Life

Distributed Storage in Daily Life

- Customer

--> Product tags/boards

--> Product Shelf

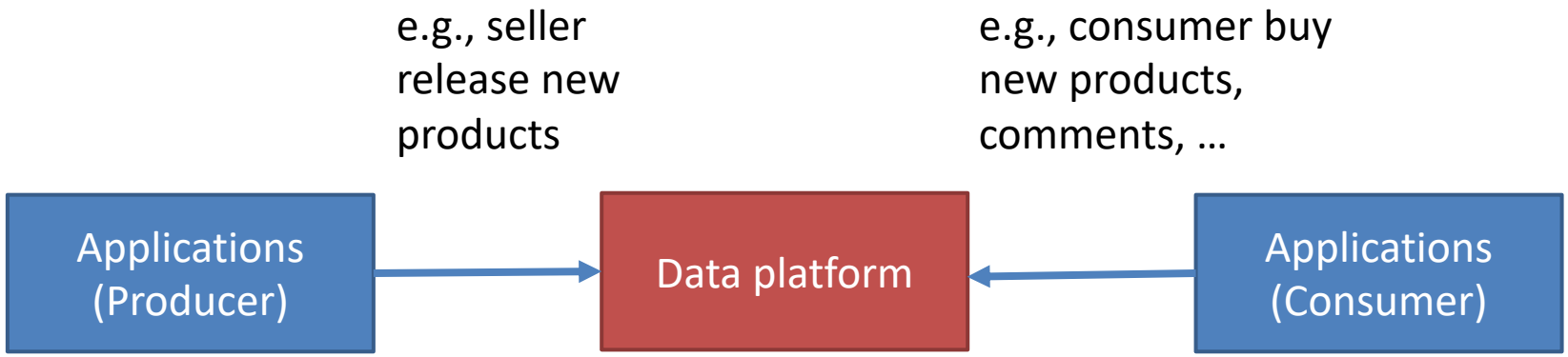
- Application

--> Rules, e.g., index

--> Data storage node

1. Application: Read/Write Data

- Application:
  - Producer: generate original data
  - Consumer: use and update the data
- Data:
  - Structured data, semi-structured data, unstructured data



1. Application: Read/Write Data

- **Structured data** refers to any data that resides in a fixed field within a record or file. This includes data contained in relational databases and spreadsheets.

id	name	age	gender
1	lyh	12	male
2	liangyh	13	female

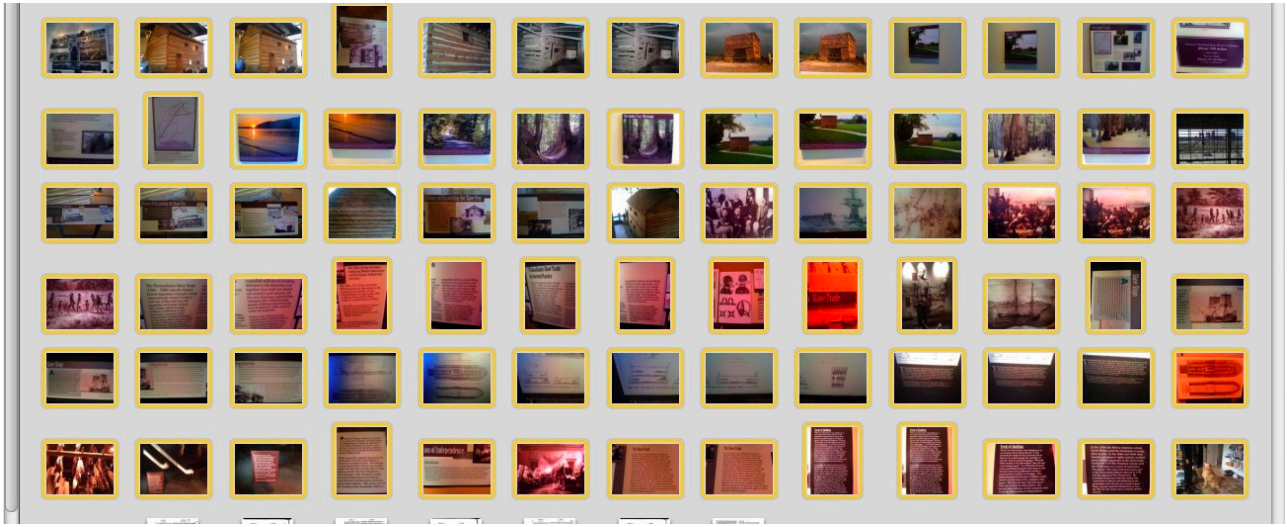
1. Application: Read/Write Data

- **Semi-structured data** is a form of structured data that does not obey the formal structure of data models associated with relational databases or other forms of data tables, but nonetheless contains tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data.

```
<person>
    <name>A</name>
    <age>13</age>
    <gender>female</gender>
</person>
```

1. Application: Read/Write Data

- **Unstructured data** is information that either does not have a pre-defined data model or is not organized in a pre-defined manner.



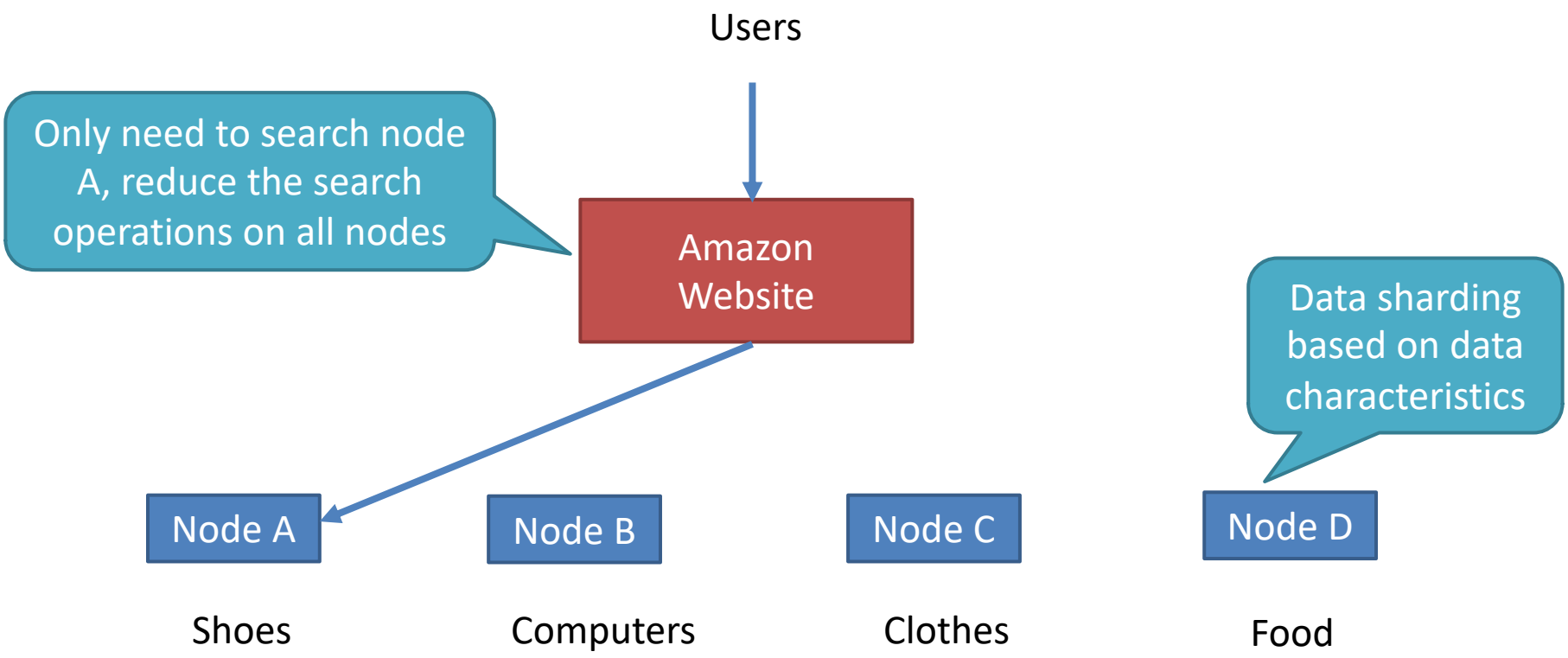
2. Rules/index: get the location of data

- Data sharding refers to a distributed storage system that stores data in the corresponding storage node or obtains the desired data in the corresponding storage node according to certain rules.

- Benefits:
  - Reduce storage and access pressure on a single storage node
  - Quickly find the storage node where the data resides, thereby greatly reducing search latency

2. Rules/index: get the location of data

- Data sharding example



2. Rules/index: get the location of data

- Suppose you have hundreds of gigabytes data to store across multiple nodes, how to do that?

- Three dimensions of design principle:
  - data uniformity (even distribution)
  - data stability
  - node heterogeneity

3. Nodes: store the data

- Structured data: distributed database, stored by tables. Example: MySQL Sharding, Microsoft SQL Azure, Google Spanner.

- Semi-structured data: distributed key/value store database, stored by key/values. Example: Redis, Memcache.

- Unstructured data: use files/blocks/objects to store. Example: Ceph, GFS, HDFS, Swift

3. Nodes: store the data

- Disk storage is large, but the IO overhead is significant, and the access speed is low. It is often used to store data that is not used frequently.

- The memory capacity is small, but the access speed is fast. It is used to store data that needs to be accessed frequently.

Data uniformity of design principle

- The ***data stored*** in different storage nodes should be balanced as much as possible to avoid overloading the storage pressure of one or some nodes, while other nodes have little data.
  - For example, 100G data, 4 nodes, it is desired that each node stores 25G data.

- ***User access*** must be balanced to avoid a situation in which one or more nodes have a large number of visits while other nodes are left with few visits.
  - For example, 1000 requests, 4 nodes. Each node processes 250 requests.

Data stability of design principle

- When a storage node fails and needs to be removed or augmented, the data should be kept as stable as possible, and no large-scale data migration will occur
  - Example, 100G data. At the beginning, there were 4 nodes of the same type (nodes 1 to 4). Each node stored 25G data. Now node 2 fails, which means that each node needs to store 100G / 3 data.

Node heterogeneity of design principle

- The hardware configuration of different storage nodes vary widely. For example, some nodes have high hardware configurations that can store large amounts of data and can withstand more requests; however, some nodes have poor hardware configurations and cannot store too much data and users cannot access too much

- A good data distribution algorithm should consider node heterogeneity

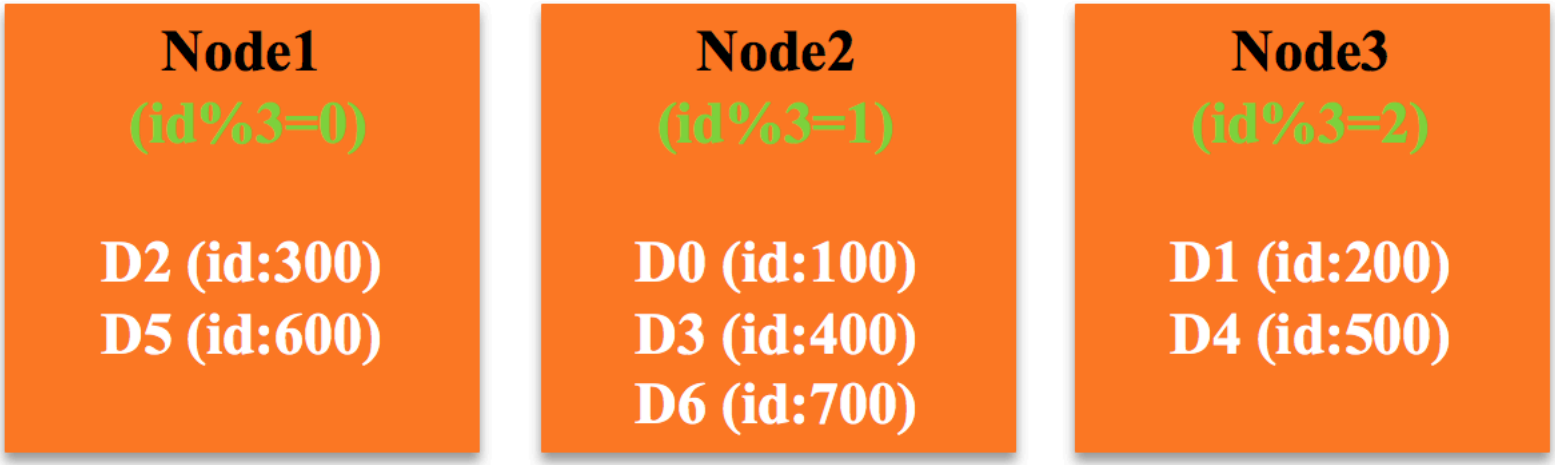
Hashing

- First determine a hash function, and then calculate the corresponding storage node by calculation

- Example:
  - hash function: $f = id \ \% \ NUM\_of\_node$
  - Node: N1, N2, N3
  - Data: D0:{ id:100, name:‘a0’}, D1:{ id:200, name:‘a1’}, D2:{ id:300, name:‘a2’}, D3:{ id:400, name:‘a3’}, D4:{ id:500, name:‘a4’}, D5:{ id:600, name:‘a5’}, D6:{ id:700, name:‘a6’}

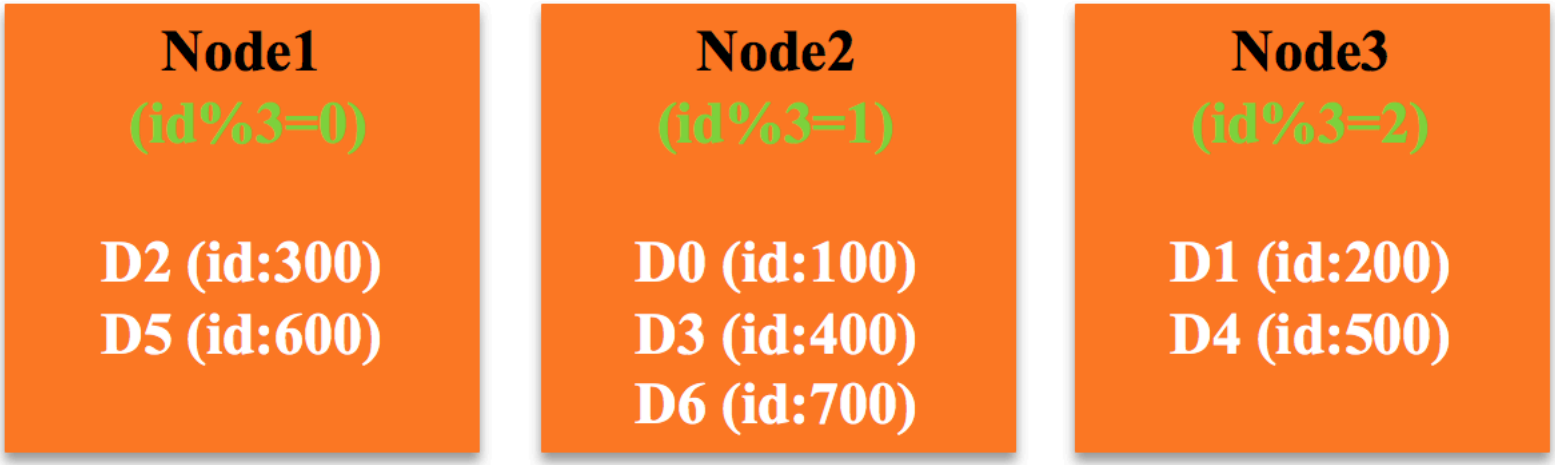
Hashing

- Example:
  - hash function: $f = id \,\%\, NUM\_of\_node$
  - Node: N1, N2, N3
  - Data: D0:{ id:100, name:‘a0’}, D1:{ id:200, name:‘a1’}, D2:{ id:300, name:‘a2’}, D3:{ id:400, name:‘a3’}, D4:{ id:500, name:‘a4’}, D5:{ id:600, name:‘a5’}, D6:{ id:700, name:‘a6’}



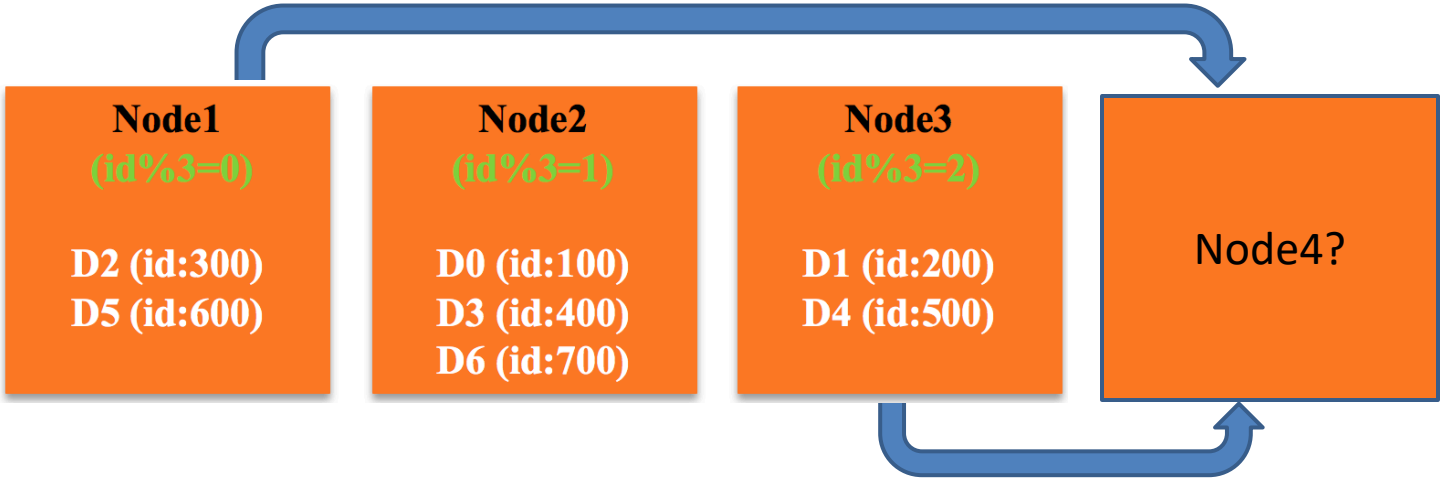
Hashing

- Advantages:
  - Guarantee data uniformity (e.g., the data will be stored across the nodes evenly)



Hashing

- Disadvantages:
  - Poor stability (if we need to add one new storage nodes, then the hashing function becomes $id \% 4$, and all data needs to be calculated again, which introduces significant data movement)



Hashing

- Scenario:
  - The hash method is suitable for scenarios where the number of nodes of the same type is relatively fixed
- Example:
  - redis

Consistent hashing

- Map the storage nodes and data to a hash ring. The data is looking clockwise in the ring and select the first node as its storage node.

- Example:
  - Node: N1, N2, N3
  - Data: D0:{ id:100, name:‘a0’}, D1:{ id:200, name:‘a1’}, D2:{ id:300, name:‘a2’}, D3:{ id:400, name:‘a3’}, D4:{ id:500, name:‘a4’}, D5:{ id:600, name:‘a5’}, D6:{ id:700, name:‘a6’}

Consistent hashing

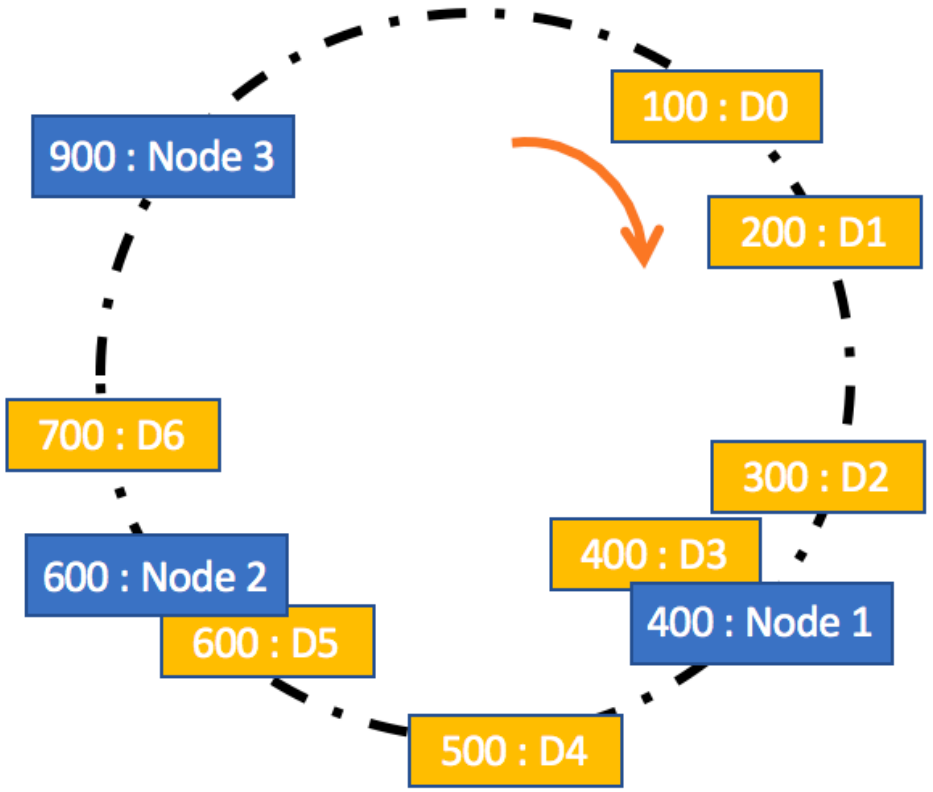
- Suppose the value of nodes and data in hash ring are as follows:

Nodes or data	Value
D0	100
D1	200
D2	300
D3	400
D4	500
D5	600
D6	700
N1	400
N2	600
N3	900

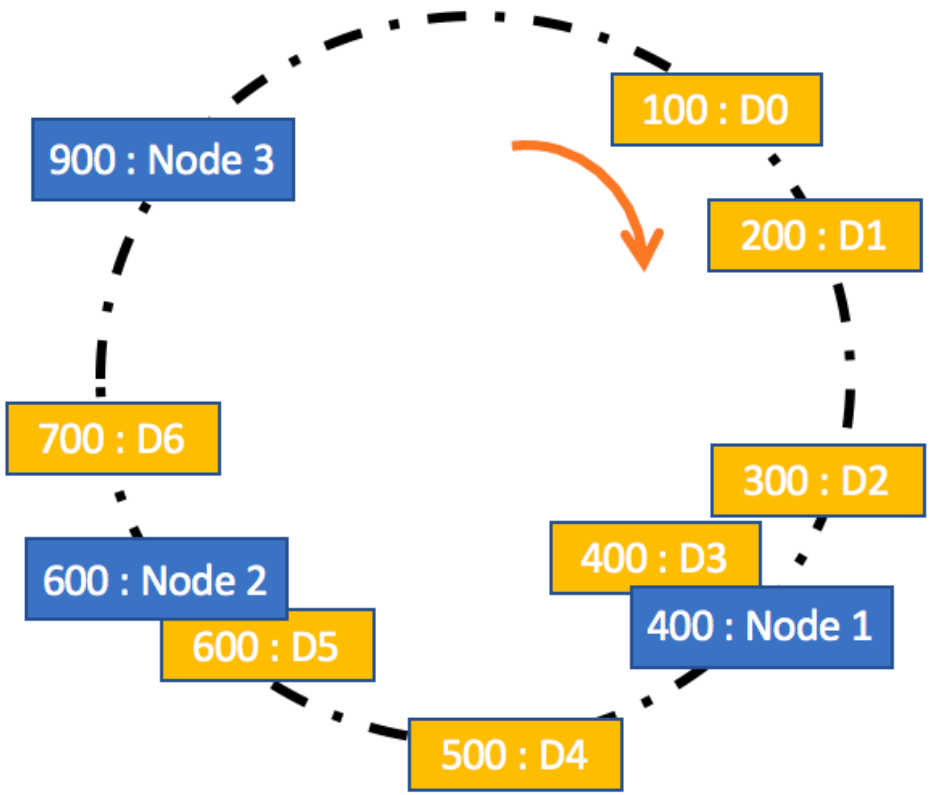
Consistent hashing

- Suppose the value of nodes and data in hash ring are as follows:

Nodes or data	Value
D0	100
D1	200
D2	300
D3	400
D4	500
D5	600
D6	700
N1	400
N2	600
N3	900



Consistent hashing



Node1
D0
D1
D2
D3

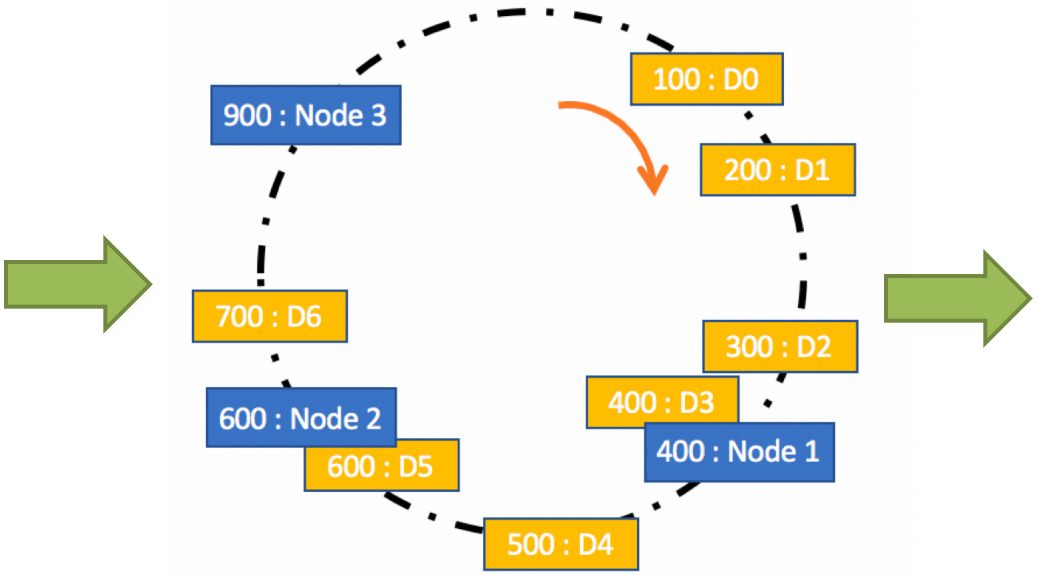
Node2
D4
D5

Node3
D6

Consistent hashing

- Consistent hashing is an improvement of hashing
- Two layers of hashing, which improve the stability

Nodes or data
D0
D1
D2
D3
D4
D5
D6
N1
N2
N3

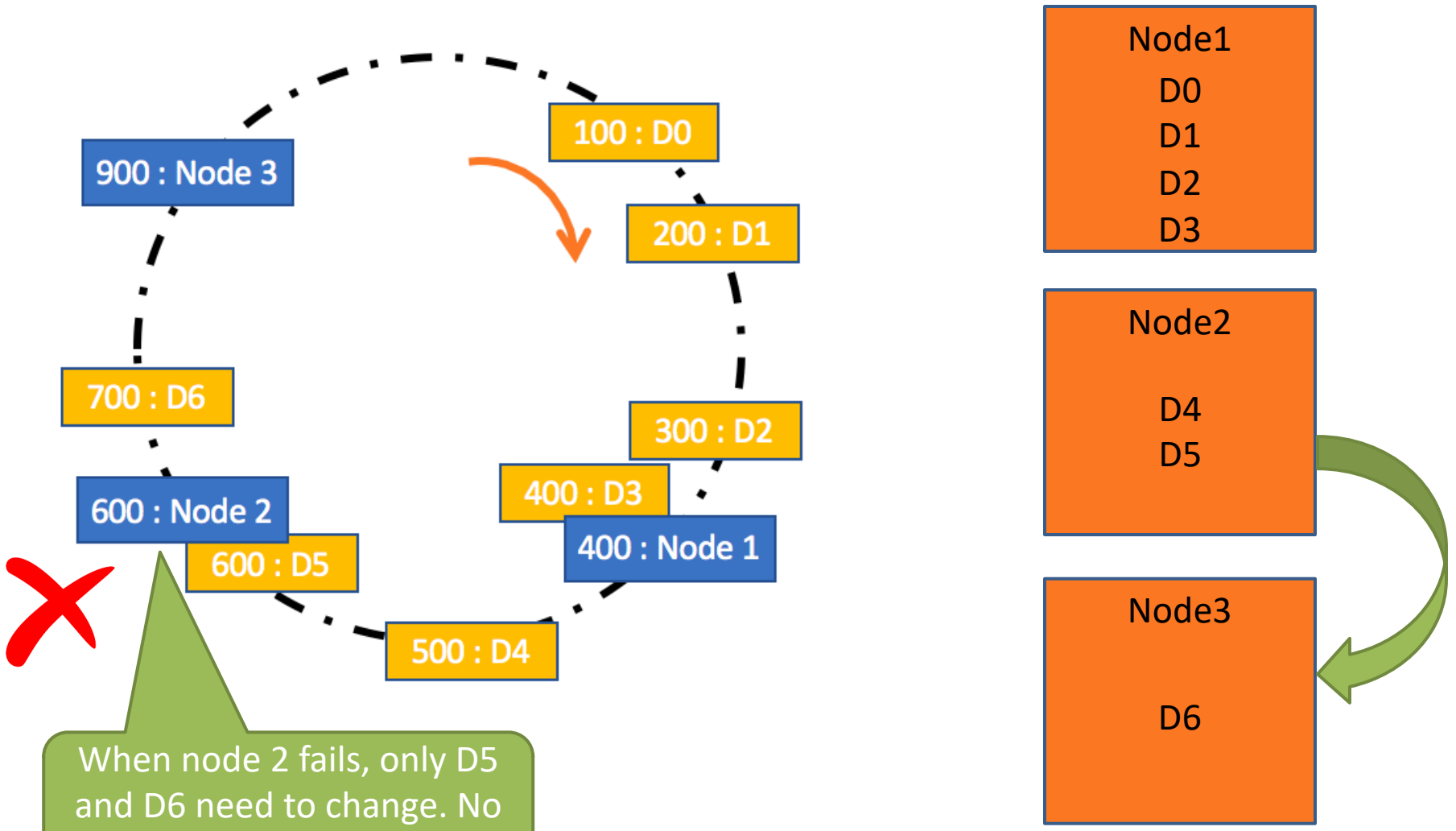


Node1
D0
D1
D2
D3

Node2
D4
D5

Node3
D6

Why consistent hashing has better stability?



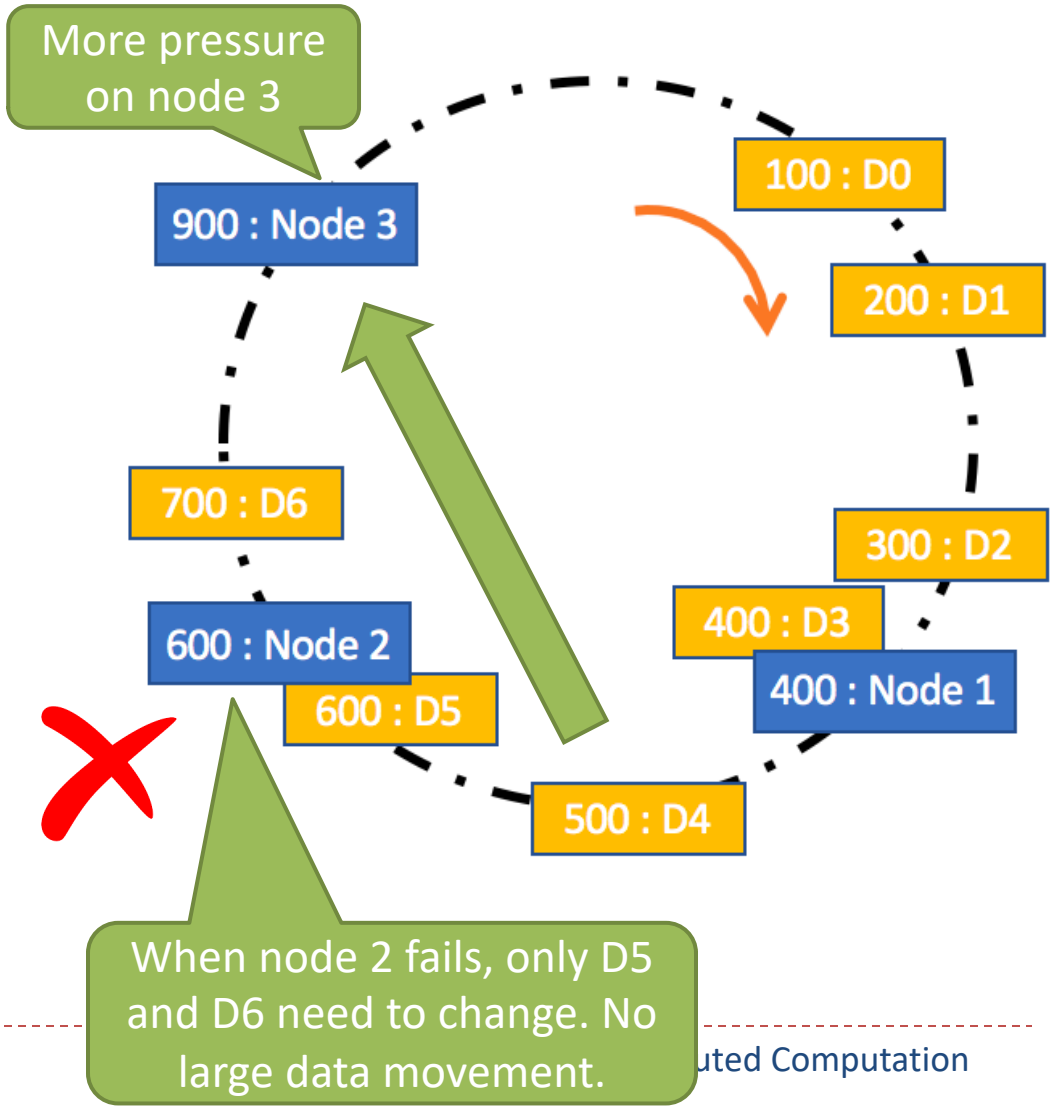
Consistent hashing

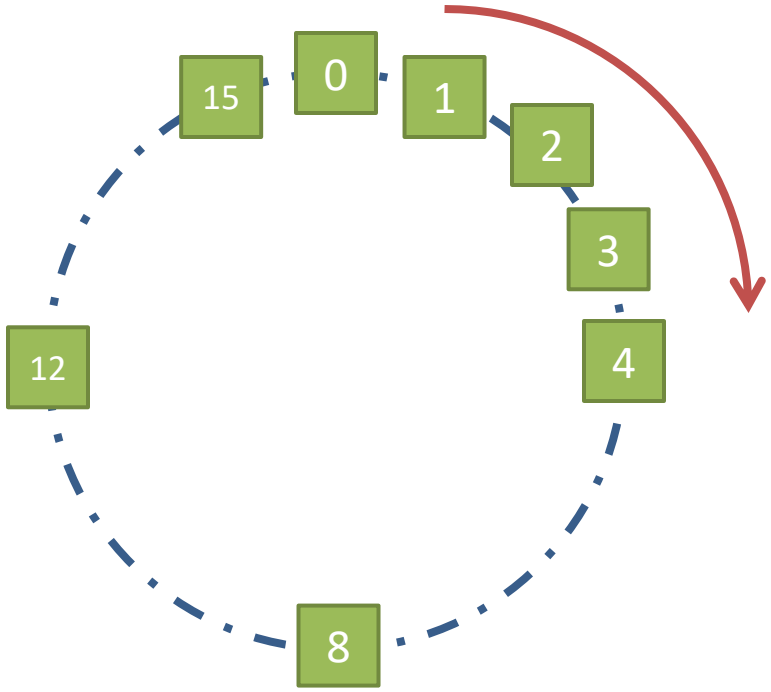
- Scenario:
  - Suitable for scenarios where nodes are the same type and node size will change

- Example:
  - Cassandra

Consistent hashing

- Problem:
  - Data uniformity. The load on subsequent nodes will become larger if the previous node fails.

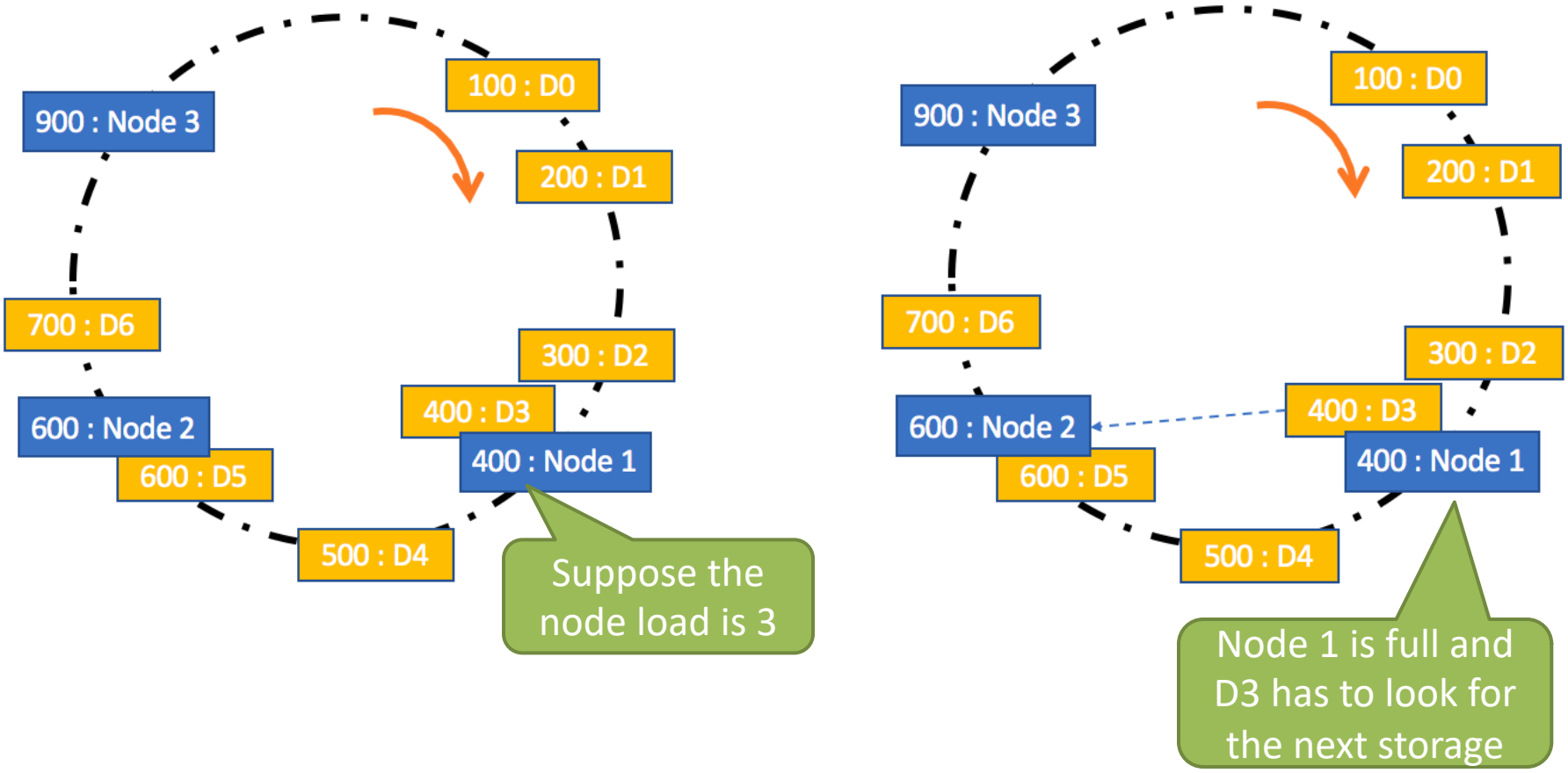



15
0
1
2
3
4
8
12

Consistent Hashing with Bounded Loads

- A storage upper limit value is set for each storage node to control the uneven data caused by storage node addition or removal.
- When the data finds the corresponding storage node according to the consistent hash algorithm, it first determines whether the storage node has reached the storage limit; if the storage limit has been reached, it continues to search for the node after the storage node clockwise.

Consistent Hashing with Bounded Loads



Consistent Hashing with Bounded Loads

- Paper: https://epubs.siam.org/doi/pdf/10.1137/1.9781611975031.39

Consistent Hashing with Bounded Loads

Vahab Mirrokni[1], Mikkel Thorup*[2], and Morteza Zadimoghaddam[1]

[1]Google Research, New York. {mirrokni, zadim}@google.com
[2]University of Copenhagen. mikkel2thorup@gmail.com

November 15, 2017

Abstract

In dynamic load balancing, we wish to allocate a set of clients (balls) to a set of servers (bins) with the goal of minimizing the maximum load of any server and also minimizing the number of moves after adding or removing a server or a client[1]. We want a hashing-style solution where we given the ID of a client can efficiently find its server in a distributed dynamic environment. In such a dynamic environment, both servers and clients may be added and/or removed from the system in any order. The most popular solutions for such dynamic settings are Consistent Hashing [KLL+97, SML+03] latter bound is the most challenging to prove. It implies that for superconstant c, we only pay a negligible cost in extra moves. We also get the same bounds for the simpler problem where, instead of a user specified balancing parameter, we have a fixed bin capacity C for all bins, and define $c = 1 + \varepsilon = Cn/m$.

1 Introduction

Load balancing in dynamic environments is a central problem in designing several networking systems and web services [SML+03, KLL+97]. We wish to allocate *clients* (also

Consistent Hashing with Bounded Loads

- Scenario:
  - Suitable for scenarios where nodes are the same type and node size will change
- Example:
  - Google cloud, Vimeo

Google Cloud

Heterogeneous Nodes

- Homogeneous nodes
  - Hash
  - Consistent Hashing
  - Consistent Hashing with Bounded Loads

- Heterogeneous Nodes?

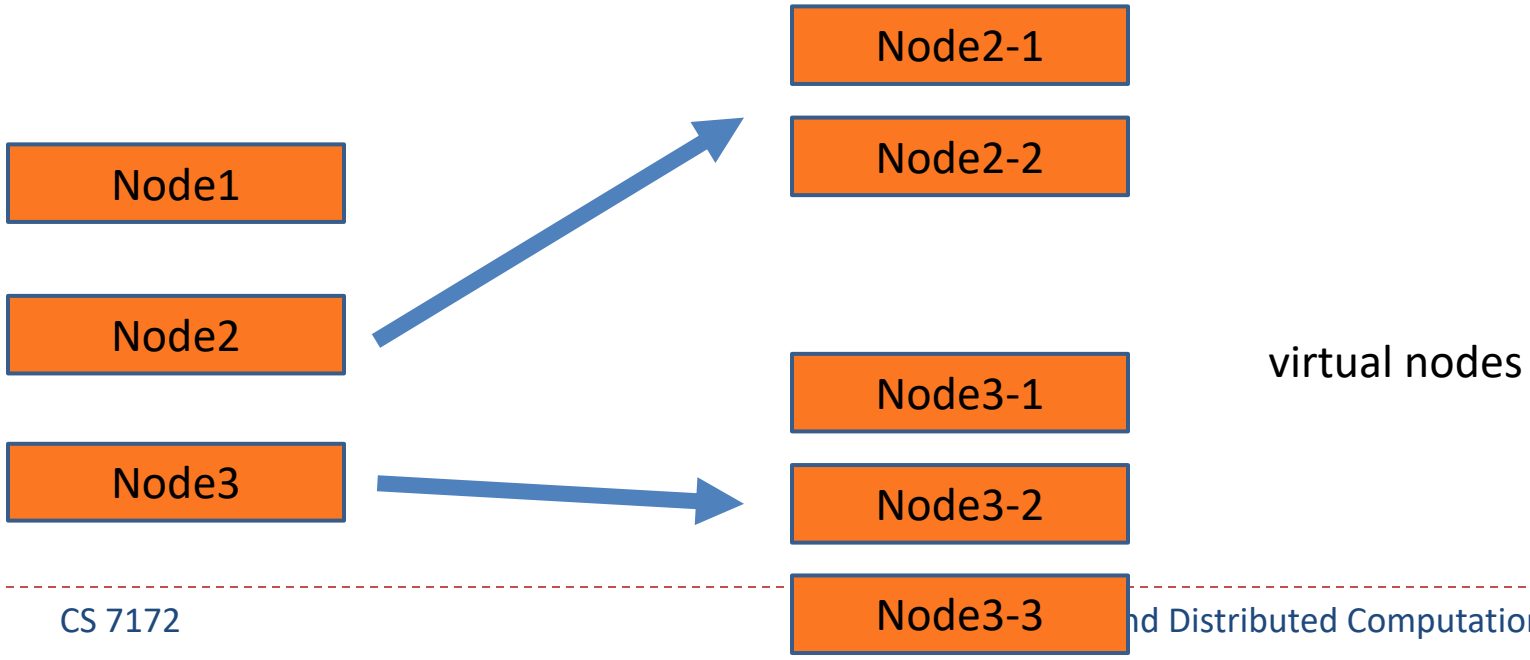
Consistent hashing with virtual nodes

- According to the performance of each node, a different number of virtual nodes are divided for each node, and these virtual nodes are mapped into a hash ring, and then data mapping and storage are performed according to a consistent hash algorithm

- Example:
  - Node: N1, N2, N3. (N3 = 3*N1, N2 = 2* N1)
  - Data: D0:{ id:100, name:‘a0’}, D1:{ id:200, name:‘a1’}, D2:{ id:300, name:‘a2’}, D3:{ id:400, name:‘a3’}, D4:{ id:500, name:‘a4’}, D5:{ id:600, name:‘a5’}, D6:{ id:700, name:‘a6’}

Consistent hashing with virtual nodes

- Example:
  - Node: N1, N2, N3. (N3 = 3*N1, N2 = 2* N1)
  - Data: D0:{ id:100, name:‘a0’}, D1:{ id:200, name:‘a1’}, D2:{ id:300, name:‘a2’}, D3:{ id:400, name:‘a3’}, D4:{ id:500, name:‘a4’}, D5:{ id:600, name:‘a5’}, D6:{ id:700, name:‘a6’}



Consistent hashing with virtual nodes

- Suppose the value of nodes and data in hash ring are as follows:

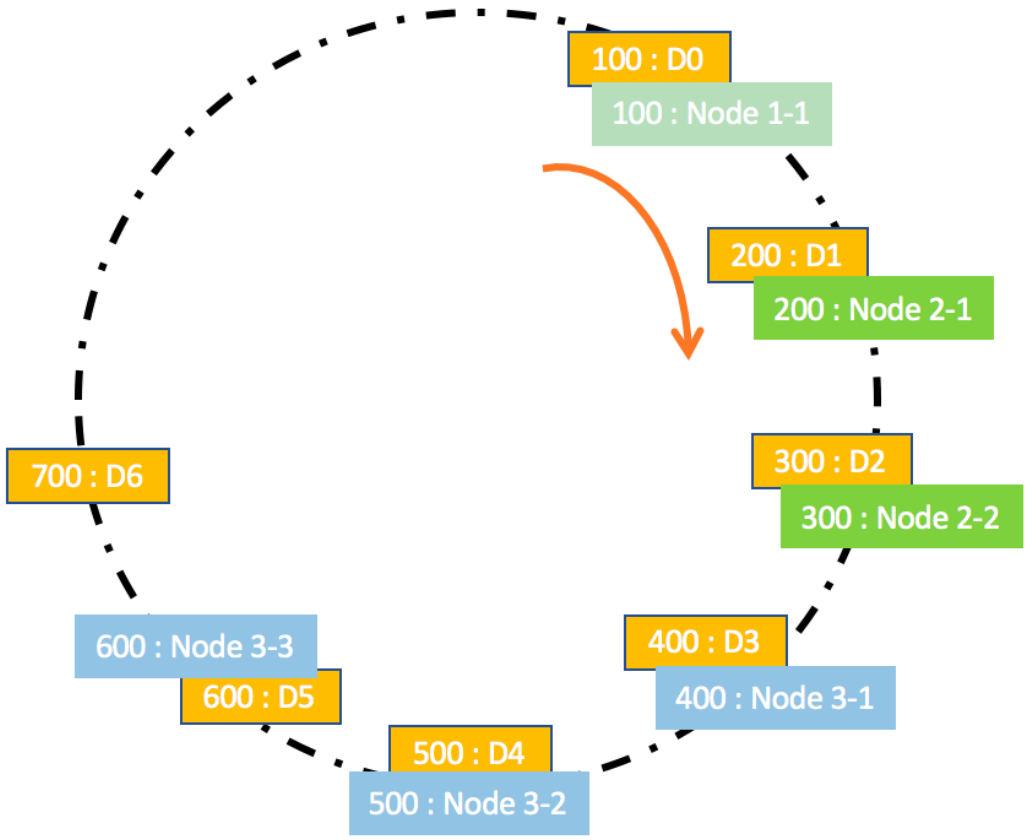
Nodes or data	Value
D0	100
D1	200
D2	300
D3	400
D4	500
D5	600
D6	700

Nodes or data	Value
Node1-1	100
Node2-1	200
Node2-2	300
Node3-1	400
Node3-2	500
Node3-3	600

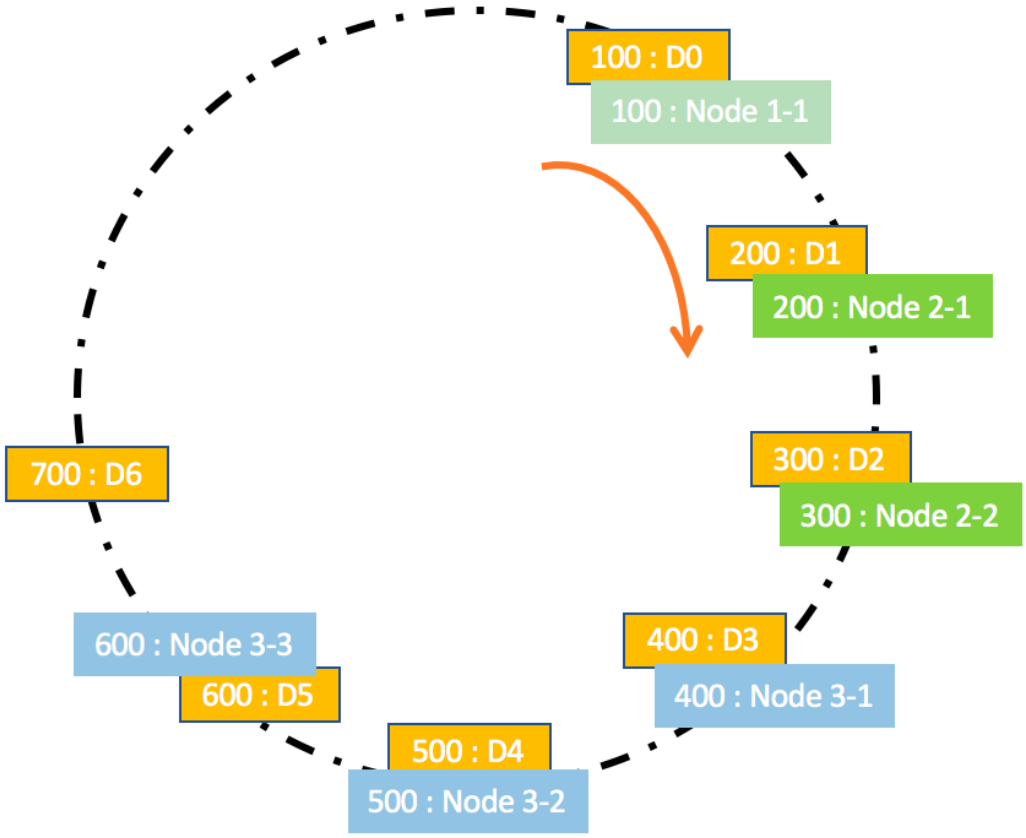
Consistent hashing with virtual nodes

Nodes or data	Value
D0	100
D1	200
D2	300
D3	400
D4	500
D5	600
D6	700

Nodes or data	Value
Node1-1	100
Node2-1	200
Node2-2	300
Node3-1	400
Node3-2	500
Node3-3	600



Consistent hashing with virtual nodes



Node1	D0, D6
Node2	D1, D2
Node3	D3, D4, D5

Consistent hashing with virtual nodes

- Scenario:
  - Suitable for scenarios where nodes are the different type and node size will change

- Example:
  - Memcached

Comparison

	data uniformity	data stability	node heterogeneity	Application
Hashing	Even distributed	Not stable	Same type	Redis cluster
Consistent hashing	Even distributed, but some nodes might have heavy load	Stable	Same type	Cassandra cluster
Consistent Hashing with Bounded Loads	Even distributed with load limits	Stable	Same type	Google cloud
Consistent hashing with virtual nodes	Even distributed	Stable	Different type	Memcached cluster