5장 : 안정 해시 설계

난이도	⭐⭐⭐
학습날짜	@2025년 12월 10일

개요

해당 챕터는 **수평적 규모 확장성**을 달성하기 위해서 요청 또는 데이터를 서버에 균등하게 나는 것을 목적으로 어떻게 설계해야 하는지에 대한 과정을 담은 챕터이다.

해시 키 문제 - 부하 분산

특정 키를 토대로 해시값을 추출하고 이를 토대로 부하를 분산하는 로직을 지난 첫번째 챕터에서 살펴보았다.

예를 들어, **로드밸런서**에서 데이터를 해시함수를 토대로 분산시켜주거나 샤딩을 할 때도 마찬가지이다.

이때, 데이터를 균등하게 분산 및 분포시켜야 안정적인 서비스를 제공할 수 있다.

만약 데이터가 균등하게 분산되지 않는다면 어떤 문제가 발생할까?

쉽게 이해하기 위해, mod4 연산을 사용하여 데이터를 분산했다고 가정해보자.

키 (Key)	해시 (Hash)	해시 % 4 (서버 인덱스)
key0	18358617	1
key1	26143584	0
key2	18131146	2
key3	35863496	0
key4	34085809	1
key5	27581703	3
key6	38164978	2
key7	22530351	3

이를 정리해보면 다음과 같이 정리된다. 즉 균등분배된다.

서버	키	개수
server0	key1, key3	2
server1	key0, key4	2
server2	key2, key6	2
server3	key5, key7	2

문제는 서버가 추가되거나 삭제되었을 때 발생한다.

1. **서버가 삭제된 경우**

   특정 서버 1대가 장애를 일으켜 동작을 중단했다고 가정해보자. 이때는 기존 `mod4` 연산이 mod3 연산으로 바뀌어서 같은 키가 기존과 다른 서버로 매칭되는 문제가 발생한다.

키 (Key)	해시 (Hash)	해시 % 4 (서버 인덱스)	해시 % 4 (서버 인덱스)	변경 여부
key0	18358617	1	0	⭕

키 (Key)	해시 (Hash)	해시 % 4 (서버 인덱스)	해시 % 4 (서버 인덱스)	변경 여부
key1	26143584	0	0	❌
key2	18131146	2	1	⭕
key3	35863496	0	2	⭕
key4	34085809	1	1	❌
key5	27581703	3	0	⭕
key6	38164978	2	1	⭕
key7	22530351	3	0	⭕

즉, 2개의 키를 제외하고 모든 키가 다른 서버로 요청을 보내는 결과가 발생한다.

2. **서버가 추가된 경우**

서버가 추가된 경우도 삭제된 경우와 동일하다. `mod4` 연산을 `mod5` 연산으로 변경하면서 완전히 다른 서버로의 매칭이 발생한다.

키 (Key)	해시 (Hash)	해시 % 4 (서버 인덱스)	해시 % 5 (서버 인덱스)	변경 여부
key0	18358617	1	2	⭕
key1	26143584	0	4	⭕
key2	18131146	2	1	⭕
key3	35863496	0	1	⭕
key4	34085809	1	4	⭕
key5	27581703	3	3	❌
key6	38164978	2	3	⭕
key7	22530351	3	1	⭕

이렇게 데이터가 다른 서버로의 매칭이 발생하는 상황으로 인해

- 캐시 서버라면 **Cache Stampede(캐시 대량 미스)** 발생해서 DB가 바로 뻗어버린다.
- 샤딩된 DB라면 전체 데이터 **Re-shuffling(재분배)** 하느라 네트워크 대역폭 다 잡아먹고 서비스가 멈춘다.

안정 해시

이러한 해시의 균등 분배 문제를 해결하기 위해 등장한 개념이 안정 해시이다.

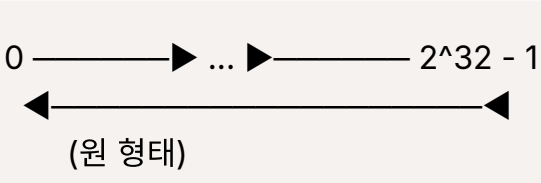
안정 해시는 해시 테이블 크기가 조정될 때 평균적으로 오직 k/n개의 키만 재배치하는 해시 기술이다.

- k : 키의 개수
- n : 슬롯의 개수

해시 공간과 해시 링

1) **해시 공간을 원(circle) 형태로 사용**

일반적인 해싱은 key → hash(key) → N으로 나눈 나머지(index)를 사용하지만, 안정 해시는 `0 ~ 2³²−1` 같은 범위를 원형(해시 링)으로 생각합니다.



2) 노드를 해시 링 위에 배치

각 서버의 이름 또는 IP를 해싱하여 링 위에 매핑합니다.

예시:

- Server A → hash("A") = 위치 20

- Server B → hash("B") = 위치 80
- Server C → hash("C") = 위치 150

3) 키(key)를 역시 해시 링 위에 배치

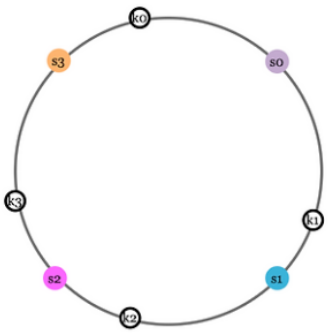
저장할 데이터의 key도 해시 함수로 링 위에 매핑합니다.

예시:

- key "user123" → hash("user123") = 위치 73

4) 키는 링에서 "시계방향으로 가장 가까운 서버"에 저장

이때, 파티션(partition)이란 인접한 서버 사이의 해시 공간이다.



[서버 조회]

해당 Hash Key가 속한 해시 공가은 시계방향으로 서버를 탐색하며 만난 첫번째 서버로 조회를 시도한다.

[서버 추가]

새로운 노드가 추가되면, 추가된 노드를 토대로 파티션을 재계산하고 파티션이 변경되는 서버들은 조회 서버를 변경한다.

[서버 삭제]

서버 추가 로직과 마찬가지로, 서버가 삭제되면 새로운 파티션을 토대로 서버들을 조회 노드를 변경한다.

문제점

해당 방식으로 안정 해시를 구현하였을때에는 몇가지 문제점이 발생한다.

1. **파티션의 크기**를 균등하게 유지하는 것이 불가능하다.

   노드의 위치 관계에 따라, 어떤 서버는 굉장히 작은 해시 공간을 할당 받고, 어떤 서버는 굉장히 큰 해시 공간을 할당받을 수 있기 때문이다.

2. **키의 균등 분포**(unfirom distribution)를 달성하기가 어렵다.

   파티션의 크기가 균등하지 않는 것과 비슷하게, Key 또한 균등하게 각 서버에 매핑될 수 없다는 점이다.

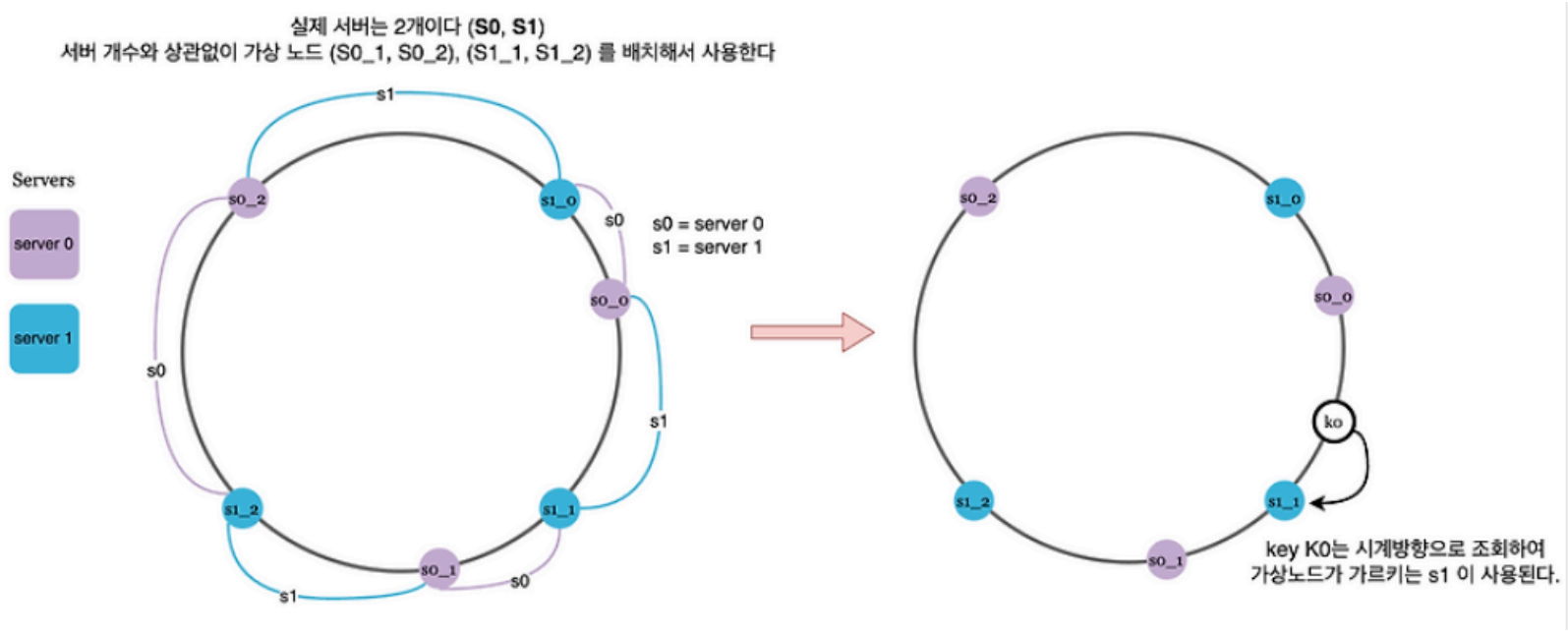
이를 극복하기 위해 가상 노드 (Virtual Node) 개념이 등장한다.

가상노드를 활용한 안정 해시

가상노드란 실제 노드 또는 서버를 가리키는 노드로서, 하나의 서버는 링 위에 여러 개의 가상 노드를 가질 수 있다.

즉, 실제 서버 (노드)를 1:1로 사용하지 않고, 가상의 노드를 만들어서 서버 1대가 여러개의 가상 노드를 가질 수 있도록 구현하는 방식이다.

> 마치 서버 한대가 여러 개의 노드를 가질 수 있게끔 설계하는 방식이라고 생각한다.



각 서버가 생성할 가상 노드의 개수는 구현에 따라 달라질 수 있다.

[조회 방식]

기존에는 시계방향으로 회전하며 가장 처음 만난 서버를 조회하지만, 이제는 가장 처음 만난 가상 노드를 조회한다.

가상 노드 방식의 장점

- 서버가 추가되거나 삭제될 때 재배치되는 키의 수가 최소화된다
- 데이터가 보다 균등하게 분포하게 되므로, 수평적 규모 확장성을 달성하기에 쉽다
- 특정한 샤드 혹은 서버에 과부하가 발생하는 핫스팟 키 문제를 줄여준다.

💡 가상 노드는 좋나?

- 좀 무식하게 서버를 늘리는 것 같은데 다른 방식은 없나?
  - 메모리가 많아지는데 **적당히 타협**하자

🤔 **노드 하나가 터지면 가상 노드 전체가 터진다?**

🤔 물리 노드와 가상 노드의 크기가 다르나?