5장 - 안정 해시 설계

수평적 규모 확장성을 달성하기 위해서는 요청 또는 데이터를 서버에 균등하게 나누는 것이 중요하다.

안정 해시는 이 목표를 달성하기 위해 보편적으로 사용하는 기술이다.

이 해시 기술을 풀려고 하는 문제부터 좀 더 자세히 살펴보자.

해시 키 재배치(rehash) 문제

N개의 캐시 서버에서 부하를 균등하게 나누는 보편적 방법은 아래의 **해시 함수를 사용**하는 것이다.

```
serverIndex = hash(key) % N (N은 서버의 개수이다)
```

- 이렇게 하면 좋지만 아래에서 단점에 대해 설명함 (서버가 증설되거나 줄었을 때 문제 발생, 안정 해시로 해결)

예제로 어떻게 동작하는지 더 자세히 알아보자. (총 4대의 서버를 사용한다고 가정)

- 다음 표는 각각의 키에 대해서 해시 값과 서버 인덱스를 계산한 예제

키	해시	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

표 5-1

- 특정한 키가 보관된 서버를 알아내기 위해, 나머지(modular) 연산을 f(key) % 4와 같이 적용

  ex) hash(key0) % 4 = 1이면, 클라이언트는 캐시에 보관된 데이터를 가져오기 위해 서버 1에 접속

- 다음 그림은 위 예제에서 키 값이 서버에 어떻게 분산되는지 보여준다.

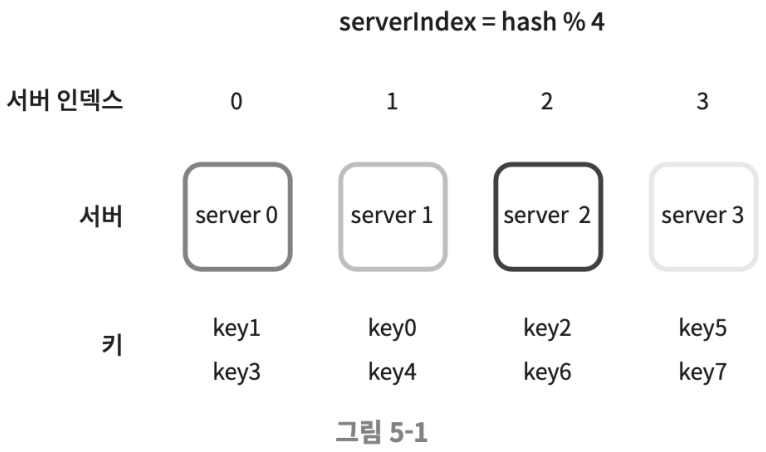

그림 5-1

  - 이 방법은 서버 풀(server pool)의 크기가 고정되어 있을 때, 그리고 데이터 분포가 균등할 때 잘 동작
  - 서버가 추가되거나 기존 서버가 삭제되면 문제 발생
    - ex) 1번 서버가 장애를 일으켜 동작을 중단했다고 했을 때, 서버 풀의 크기는 3으로 변함
      그 결과로, 키에 대한 해시 값은 변하지 않지만 나머지(%) 연산을 적용하여 계산한 서버 인덱스 값은 달라질 것이다.
    - 즉, 어떤 서버가 죽으면 대부분 캐시 클라이언트가 데이터가 없는 엉뚱한 서버에 접속하게 된다.
      이 결과로 대규모 캐시 미스가 발생한다.
    - **안정해시는 이 문제를 효과적으로 해결하는 기술**이다.

안정 해시(consistent hash)

- 해시 테이블 크기가 조정될 때 **평균적으로 오직 `k/n` 개의 키만 재배치하는 해시 기술**
  - `k` : 키의 개수
  - `n` : 슬롯(slot)의 개수

- 이와는 달리 대부분의 전통적 해시 테이블은 슬롯의 수가 바뀌면 거의 대부분 키를 재배치함

해시 공간과 해시 링

안정 해시의 동작 원리

- 해시 함수 f는 SHA-1을 사용한다고 가정, 함수의 출력 값 범위는 x0, x1, ... xn과 같다고 가정
- SHA-1의 해시 공간(hash space) 범위는 `0 ~ 2^160 - 1`
- 따라서 x0은 0, xn은 2^160-1이며, 나머지 x1부터 xn-1까지는 그 사이의 값을 갖는다.
- `왼쪽 그림`은 해시 공간을 **리스트**로 표현, `오른쪽 그림`은 해시 공간의 양쪽을 접어 **링**으로 만든 것이다.

그림 5-3

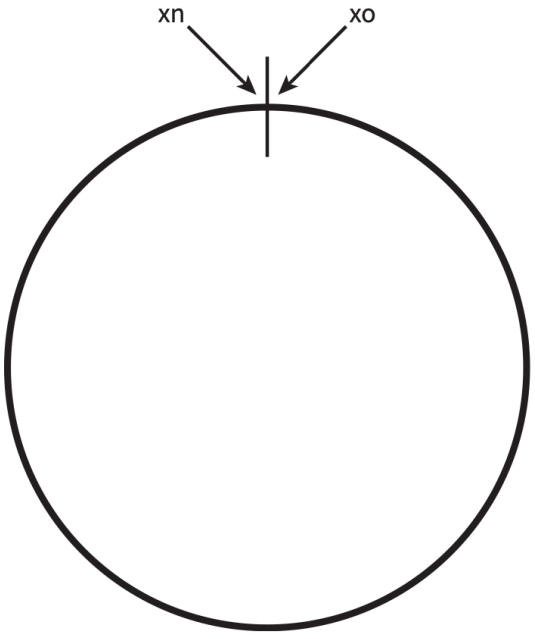

그림 5-4

- SHA-1 범위의 공간을 미리 만들어서 배치하는 건가? (80p)

해시 서버

해시 함수 `f`를 사용하면 서버 IP나 이름을 이 **링 위의 어떤 위치에 대응**시킬 수 있다.

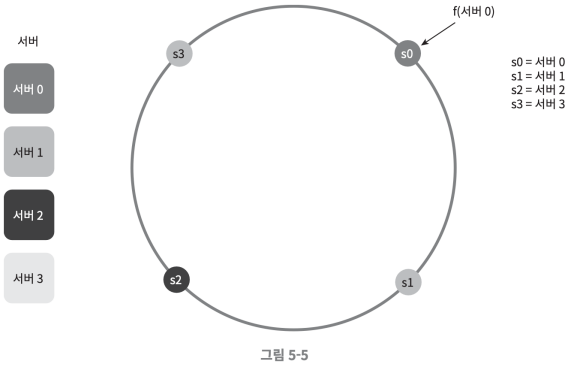

그림 5-5

4개의 서버를 해시 링 위에 배치한 결과

해시 키

여기 사용된 해시 함수는 `해시 키 재배치 문제`에 언급된 **함수와는 다르며, 나머지 연산은 사용하지 않는다**.

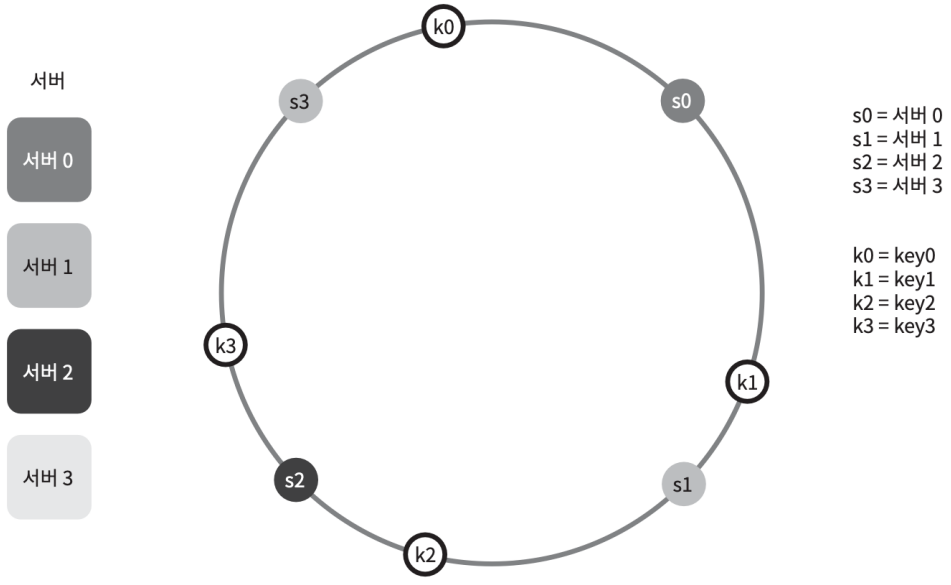

그림 5-6

캐시할 키 key0, ..., key3 또한 해시 링 위의 어느 지점에 배치할 수 있음

서버 추가

- 서버를 추가하더라도 키 가운데 일부만 재배치하면 된다. (다음 그림을 참고)

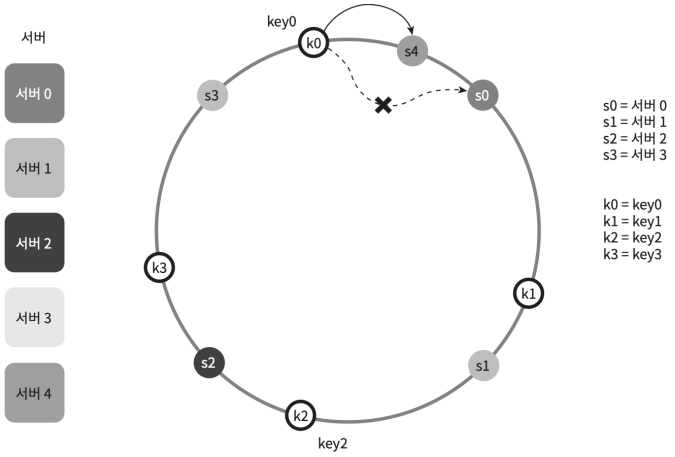

그림 5-8

- 서버 4가 추가된 뒤 key0만 재배치 되고, k1, k2, k3는 같은 서버에 남는다.
- 서버 4가 추가되기 전, key0은 서버 0에 저장되어 있었다.
  하지만 서버 4가 추가된 뒤 key0은 서버 4에 저장된다.
  (key0의 위치에서 시계 방향으로 순회했을 때 처음으로 만나게 되는 서버가 서버 4이기 때문)
  다른 키들은 재배치되지 않는다.

서버 제거

- 추가와 마찬가지로 서버가 제거되면 키 가운데 일부만 재배치 된다.

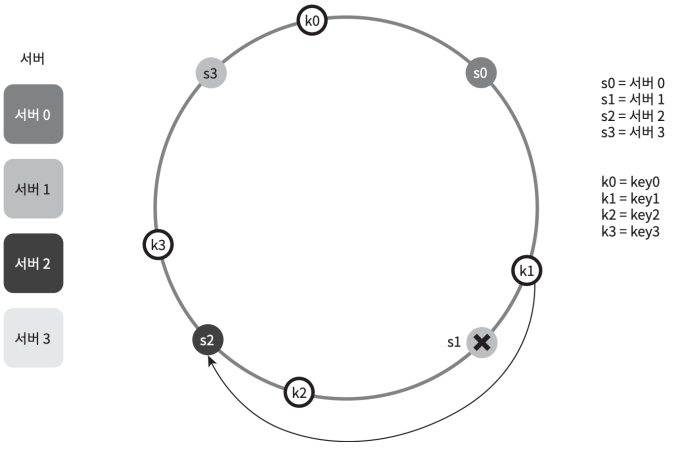

그림 5-9

- 서버 1이 삭제되엇을 때 key1만 서버 2로 재배치 된다. 나머지 키는 영향이 없음

기본 구현법의 두 가지 문제

- 안정 해시 알고리즘은 MIT에서 처음 제안되었고, 기본 절차는 다음과 같다.
  - 서버와 키를 균등 분포(uniform distribution) 해시 함수를 사용해 해시 링에 배치
  - 키의 위치에서 링을 시계 방향으로 탐색하다 만나는 최초의 서버가 키가 저장될 서버

이 접근법은 두 가지 문제가 있음

1. **서버가 추가되거나 삭제되는 상황을 감안하면 파티션(partition)의 크기를 균등하게 유지하는 게 불가능**
   - 파티션 : 인접한 서버 사이의 해시 공간
   - 어떤 서버는 작은 해시 공간을 할당 받고, 어떤 서버는 큰 해시 공간을 할당 받는 상황 발생 가능

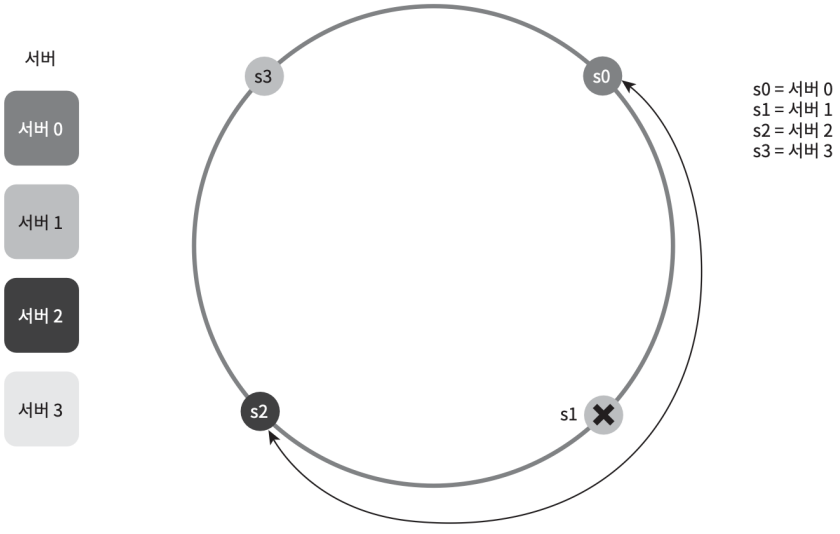

그림 5-10

s1이 삭제돼 s2의 파티션이 다른 파티션의 비해 두 배로 커지게 되는 상황

2. **키의 균등 분포(uniform distribution)를 달성하기 어려움**

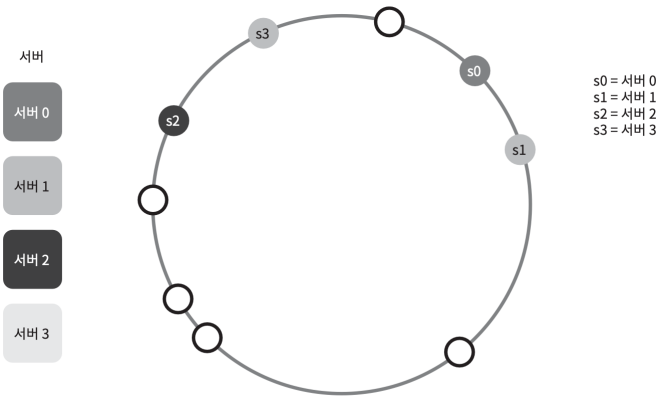

그림 5-11

- 서버1과 서버3은 아무 데이터가 없고, 대부분의 키는 서버2에 보관될 것이다.
- 이 문제를 해결하기 위해 제안된 기법이 **가상 노드(virtual node)** 또는 **복제(replica)**라는 기법이다.

가상 노드, 복제

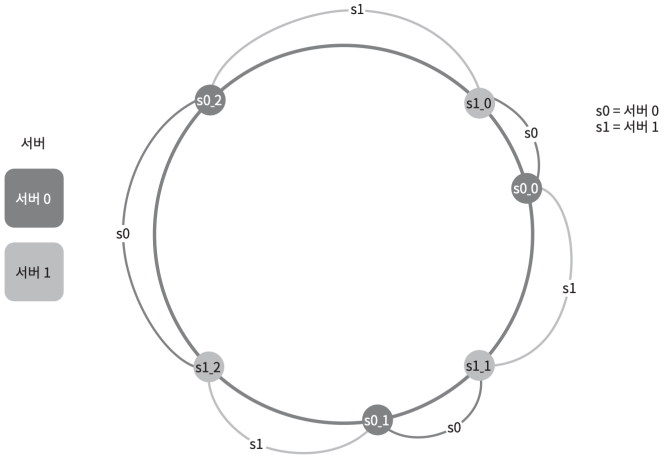

그림 5-12

- 실제 노드 또는 서버를 가리키는 노드로서, 하나의 서버는 링 위에 여러 개의 가상 노드를 가질 수 있음
- 키의 위치로부터 시계방향으로 링을 탐색하다 만나는 최초의 가상 노드가 해당 키가 저장될 서버가 됨

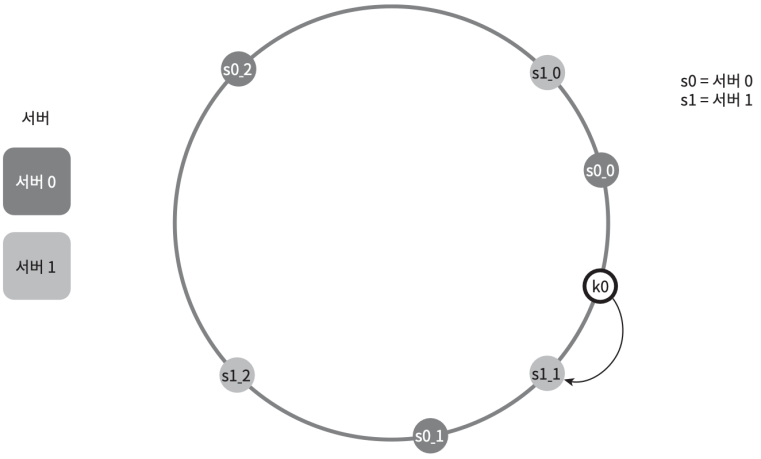

그림 5-13

k0가 저장되는 서버는 서버 1 (시계 방향으로 최초)

- **가상 노드의 개수를 늘리면 키의 분포는 점점 더 균등**해진다.
  표준 편차(standard deviation)가 작아져서 데이터가 고르게 분포되기 때문
  - 하지만 **가상 노드 데이터를 저장할 공간은 더 많이 필요**하게 된다.
    적절한 타협적 결정(tradeoff)이 필요하다.

재배치할 키 결정

> 서버가 추가되거나 제거되면 데이터 일부는 재배치해야 된다. 어느 범위의 키들이 재배치되어야 할까?

다음 그림처럼 서버 4가 추가되었을 때 이에 영향 받는 범위는

`s4(새로 추가된 노드)`부터 그 **반시계 방향**에 있는 `첫 번째 서버 s3`까지이다.
즉 s3부터 s4 사이에 있는 키들을 s4로 재배치해야 한다.

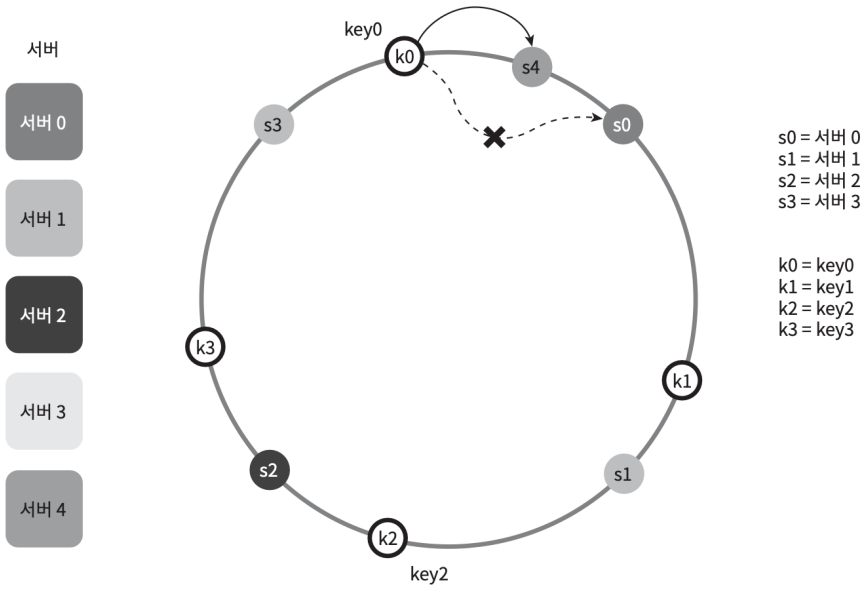

그림 5-14

다음 그림처럼 서버 s1이 삭제되면

`s1부터(삭제된 노드)` 그 **반시계 방향**에 있는 `최초 서버 s0` 사이에 있는 키들이 재배치

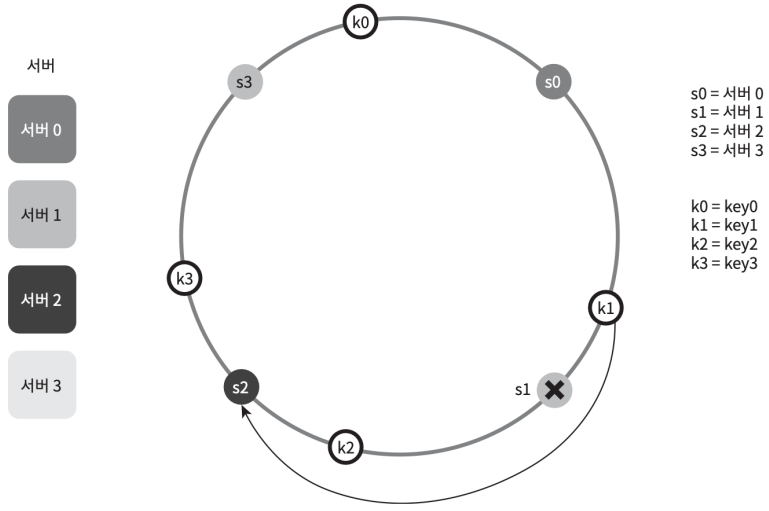

그림 5-15

마치며

안정 해시의 이점

- 서버가 추가되거나 삭제될 때 재배치되는 키의 수가 최소화된다.

- 데이터가 보다 균등하게 분포되므로 수평적 규모 확장성을 달성하기 쉽다.
- 핫스팟(hotspot) 키 문제를 줄인다.
    - 특정한 샤드(shard)에 대한 접근이 지나치게 빈번하면 서버 과부하 문제가 생길 수 있다.
      케이티 페리, 저스틴 비버와 같은 유명인의 데이터가 전부 같은 샤드에 몰리는 상황을 생각해보면 됨

      안정 해시는 데이터를 좀 더 균등하게 분배하므로 이런 문제가 생길 가능성을 줄인다.

실제 안정 해시를 사용하는 사례

- 아마존 다이나모 데이터베이스(DynamoDB)의 파티셔닝 관련 컴포넌트
- 아파치 카산드라(Apache Cassandra) 클러스터에서의 데이터 파티셔닝
- 디스코드(Discord) 채팅 애플리케이션
- 아카마이(Akamai) CDN
- 매그레프(Meglev) 네트워크 부하 분산기
- 책에선 나오진 않지만 NGINX의 Load Balancing - 여기 블로그 참고, 공식 문서는 여기
    - Round Robin and weighted Round Robin
    - Consistent (ketama) Hash ← 이 부분에서 안전 해시를 사용
- 등

▼ **Chapter 5: DESIGN CONSISTENT HASHING**

1. Consistent hashing
2. Consistent Hashing
3. Dynamo: Amazon’s Highly Available Key-value Store
4. Cassandra - A Decentralized Structured Storage System
5. How Discord Scaled Elixir to 5,000,000 Concurrent Users

6. CS168: The Modern Algorithmic Toolbox Lecture #1: Introduction and Consistent Hashing
7. Maglev: A Fast and Reliable Software Network Load Balancer