# 5주차_6장

## 키-값 저장소

- 키는 일반 텍스트일 수 있고 해시값일 수도 있다.
- 키가 짧을수록 성능이 좋다.
- 값은 리스트, 객체, 문자열 등 무엇이든 올 수 있다.

put(key, value)로 쌍을 저장소에 저장하고 get(key)로 값을 꺼낸다.

## 문제 이해 및 설계 범위 확정

우리가 결정해야할 것은 다음의 것들이 있다.

- 읽기, 쓰기, 메모리 사용량 사이의 균형
- 데이터의 일관성과 가용성에서의 타협적결정

을 결정해야함

그리고 이것을 결정하기 위해 다음의 특성을 알고 있다.

- 키=값 쌍의 크기는 10KB 이하이다.
- 큰 데이터를 저장할 수 있어야 한다.
- 높은 가용성 → 장애상황을  빠르게 응답하기 위해서
- 높은 규모 확장성 → 트래픽 양에 따라 자동적으로 서버 증설/삭제가 이뤄져야한다.
- 데이터 일관성 수준은 조정이 가능해야한다
- 응답 지연시간이 짧아야한다.

### 단일 키-값 저장소

키-값 페어 전부를 메모리에 해시 테이블로 저장하는 방법을 쓰는 것이 가장 쉽다. 하지만 메모리가 쉽게 고갈되기 때문에 아래 개선책을 고려한다.

- 데이터 압축

- 자주 쓰이는 데이터만 메모리에 남기고 나머지는 디스크에 저장
- 분산 키-값 저장소로 여러 개의 메모리 쓰기

# 분산 키-값 저장소

분산-키 값 저장소는 분산 해시 테이블이라고한다. 키-값 쌍을 여러 서버에 분산시키는 탓이다. 분산 시스템을 설계할 때는 CAP 정리를 이해하고 있어야한다.

# CAP

- 데이터 **일관성** : 클라이언트는 어떤 노드에 접근했는지 상관없이 같은 데이터를 봐야 한다.
- **가용성** : 일부 노드에 장애가 발생하더라도 항상 응답할 수 있어야 한다.
- **파티션 감내** : 파티션(네트워크에 장애)이 생기더라도 시스템은 동작해야 한다.

CAP 정리란 `세 가지 요구사항 중 오로지 두 가지만 만족할 수 있다`는 것을 의미한다. 즉 하나는 반드시 희생해야한다.

- 충족하는 요구사항에 따라 CP, AP, CA 시스템이라 한다.
- CP 시스템: 일관성, 파티션 감내를 지원하는 저장소.
- AP 시스템: 가용성과 파티션 감내를 지원하는 저장소.
- CA 시스템: 일관성과 가용성을 지원하는 저장소. 파티션 감내는 지원하지 않는다. 그러나  네트워크 장애는 피할 수 없는 일이므로 분산시스템은 반드시 파티션 문제를 감내할 수 있게 설계해야한다. 즉, 실세계에서 CA는 성립하지 않는다.

## 이상적 상태

이상적 환경이라면 네트워크가 파티션되는 상황은 절대로 일어나지 않을 것이다. n1에 기록된 데이터는 자동적으로 n2와 n3에 복제된다. 데이터 일관성과 가용성도 만족된다.

## 실세계에서는 다르다

분산시스템은 파티션 문제를 피할 수 없다. 파티션 문제가 발생하면 우리는 일관성과 가용성 사이에서 하나를 택해야한다.

n3에 장애가 발생한 상황. 클라이언트가 n1이나 n2에 쓰기 연산을 하면 n3에는 전달되지 않는다. 반대의 경우도 마찬가지. 서로가 오래된 사본을 가지게 될 수 있다.

- 일관성을 택한다면 n1과 n2에 대해서도 쓰기연산을 중단시킨다.
- 가용성을 택한다면 계속 읽기 연산을 허용해야한다. n1, n2에서의 쓰기연산도 허용할 것이다. 파티션 문제가 해결되면 새 데이터를 n3에 전송할 것이다.

면접 문제의 요구사항에 따라 CAP 정리를 적절히 적용하자!!

# 시스템 컴포넌트

이번 절에서는 키-값 저장소 구현에 사용될 핵심 컴포넌트들 및 기술들을 살펴볼 것이다.

## 데이터 파티션

대규모의 애플리케이션의 경우, 전체 데이터를 작은 파티션으로 분할한 다음 여러대의 서버에 저장해야한다.

데이터를 파티션 단위로 나눌 때 다음 문제를 해결해야한다.

- 데이터를 여러 서버에 고르게 분산할 수 있는가
- 노드가 추가되거나 삭제될 때 데이터의 이동을 최소화할 수 있는가
-

5장에서의 안정해시가 이 문제에 적합한 해결책이된다.

- 안정해시 : 해시링에 서버를 배치하고 어떤 키-값쌍을 어떤 서버에 저장할 지 결정하려면 우선 해당키를 같은 링위에 배치하고 링을 시계방향으로 순회하다가 만나는 첫번째 서버가 바로 키-값을 저장할 서버이다.

## 데이터 다중화

고가용성과 안정성을 확보하려면 비동기적인 **다중화**가 필요하다.

다중화할 서버를 선정하는 방식은 다음과 같다! 해시 링 위에 저장할 키 값을 배치하고 만나는 N개의 서버에 데이터 사본을 보관하는 것이다.

- 데이터 유실을 방지하기 위해 데이터를 N 대의 서버에 비동기적으로 복제한다.

- 안정 해시를 응용해서 바로 다음 노드 뿐만이 아니라 다음 N개의 노드에 데이터를 저장한다.
- 가상 노드를 쓰면 실질적으로 N대의 서버에 저장되지 않을 수 있기 때문에 조율해야 한다.
- 고속 연결망이 구축된 다른 데이터 센터에 데이터 사본을 보관한다.

## 데이터 일관성

읽기 쓰기의 일관성을 보장하기 위해 **정족수 합의**(Quorum Consensus) 프로토콜을 사용한다.

- N : 데이터 사본 개수
- W : 쓰기 연산의 정족수. W 개 노드가 쓰기를 성공했다고 알려야 한다
- R : 읽기 연산의 정족수. R 개 노드가 읽기를 성공했다고 알려야 한다

중재자(coordinator)는 클라이언트의 읽기/쓰기 요청을 전파하고 성공 여부를 판단한다.

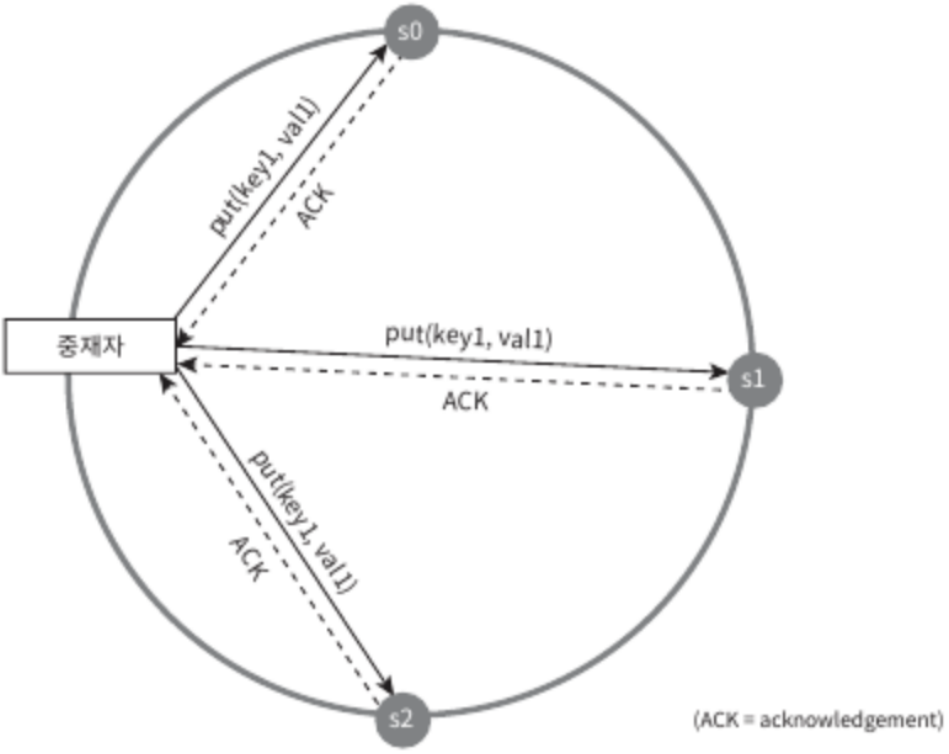


그림 6-6

W, R, N의 값을 정하는 것은 응답 지연과 데이터 일관성 사이의 타협점을 찾는 전형적인 과정이다.

W, R이 작은 경우 응답속도는 빠를 것이다. W,R이 큰 경우는 데이터 일관성 수준은 향상되지만 응답은 느려질 것이다.

보통 요구 사항에 따라 N, W, R을 다음처럼 정할 수 있다.

- 빠른 읽기 연산에 최적화된 시스템: R=1, W=N
- 빠른 쓰기 연산에 최적화된 시스템: W=1, R=N
- 강한 일관성이 보장됨: W + R > N
- 강한 일관성이 보장되지 않음: W + R <= N

## 일관성 모델

일관성 모델은 데이터의 일관성의 수준을 결정하는데, 종류가 다양하다.

- 강한 일관성: 모든 읽기 연산은 가장 최근에 갱신된 결과를 반환한다. 다시 말해서 클라이언트는 절대로 낡은 데이터를 보지 못한다.
- 약한 일관성: 읽기 연산은 가장 최근에 갱신된 결과를 반환하지 못할 수 있다.
- 최종 일관성: 약한일관성에 속함. 갱신 결과가 모든 사본에 반영이 되는 모델

이번 장에서는 최종 일관성 모델을 따를 것이다.

## 비 일관성 해소 기법: 데이터 버저닝

최종 일관성 모델을 따를 경우 일관성이 깨질 수 있다. 이 문제는 클라이언트가 해결해야하는데, 이 때 버저닝과 벡터시계를 사용한다.

버저닝은 데이터 변경 시마다 데이터의 새로운 버전을 만드는 것이다. 각 버전의 데이터는 불변하다.

서버 1, 2가 있고 두 서버가 같은 키 값에 대해 다른 버전의 데이터를 가지고 있다면, 이 둘의 충돌을 어떻게 해결해야할까?

바로 백터시계를 이용하는 것이다. 벡터 시계는 [서버, 버전]의 순서쌍을 데이터에 메단것이다. 어떤 버전이 선행 버전인지, 후행 버전인지, 아니면 다른 버전과 충돌이 있는지 판별하는데 쓰인다.

- 벡터 시계 = [서버, 버전] 순서쌍
- 모든 노드는 하나의 데이터당 벡터 시계 를 가진다.

  D([S1,v1],,⋯,[Sn,vn])

    - **D** : 하나의 데이터에 대한 일종의 메타데이터
    - **S** : D에 쓰기 연산을 수행한 서버. 모든 S는 고유하다.
    - **v** : 한 서버가 여러 번의 쓰기 연산을 수행할 때 증가시키는 버전
- 같은 데이터의 벡터 시계가 노드끼리 다르다면, 노드(S)별 가장 최신의 버전(v)을 합해 놓은 새로운 벡터 시계를 만든다.

**단점**

- 충돌 감지 및 해소 로직이 클라이언트에 들어가 있어 구현이 복잡해진다.
- 오래된 [서버, 버전] 순서쌍을 제거하지 않으면 너무 많은 데이터가 저장된다.
    - 이렇게 하면 버전 간 선후 관계를 100% 확정할 수 없어 충돌 해소의 효율성이 낮아진다.
    - 하지만 DynamoDB에서는 아직 이 문제가 발생하지 않았다.

## 장애 처리

대다수 대규모 시스템에서 장애는 그저 불가피하기만 한 것이 아니라 아주 흔하게 벌어지는 사건이다. 따라서 장애를 어떻게 처리할 것이냐 하는 것은 굉장히 중요한 문제다. 이번 절에서 우리는 우선 장애 감지 기법들을 먼저 살펴보고, 그 다음으로 장애 해소 전략들을 짚어 볼 것이다.

## 장애 감지

분산 시스템에서는 보통 두 대 이상의 서버가 서버 A의 장애를 보고해야 해당 서버에 실제로 장애가 발생했다고 간주하게 된다.

- 모든 노드 사이에 멀티캐스팅 채널을 구축하는 것이 서버 장애를 감지하는 가장 손쉬운 방법이다. 하지만 서버의 수가 많아진다면 비효율적일 것이다.

## 가십 프로토콜 (gossip protocol)

### 동작원리

- 각 노드는 멤버십 목록을 유지한다. 멤버십 목록은 `각 멤버 ID, 박동 카운터`쌍의 목록이다.
- 각 노드는 주기적으로 자신의 박동카운터를 증가시킨다.
- 각 노드는 무작위로 선정된 노드들에게 주기적으로 자기 박동 카운터 목록을 보낸다.
- 박동 카운터 목록을 받은 노드는 멤버십 목록을 최신값으로 갱신한다.
- 어떤 멤버의 박동 카운터 값이 지정된 시간동안 갱신되지 않으면 해당 멤버는 장애 상태인 것으로 간주된다. (아무 서버도 해당 서버에게서 리스트를 전달받지 못했다는 뜻)

## 일시적 장애 처리

가십 프로토콜로 장애를 감지했다면,데이터 일관성의 정족수 합의를 사용해 해결한다.

- **엄격한 정족수** : 읽기와 쓰기 연산을 금지하고 장애를 처리한다.
- **느슨한 정족수**
  - 장애 상태의 서버를 제외하고 W개의 쓰기 연산 서버, R개의 읽기 연산 서버를 선택한다.
  - 중재자는 장애 서버로 가는 요청을 다른 서버들로 보내서 처리한다.
  - 이 동안의 변경사항은 서버를 복구하면 일괄 반영한다.
  - 이를 위해 임시 쓰기 연산 서버에 단서(hint)를 남긴다. 이를 임시위탁이라고 한다.

## 영구 장애 처리 : 안티 엔트로피 프로토콜

- 안티 엔트로피 프로토콜 : 사본의 망가진 상태를 탐지하고 갱신하는 프로토콜
- *해시 트리(merkle tree)**를 이용해 탐색 속도를 빠르게 함
- Data Nodes : 모든 노드의 데이터는 적절한 파티션으로 나누어서 저장
- 각 데이터의 해시를 트리 형태로 올리기 때문에 트리의 루트부터 값을 비교하면 어디서 데이터가 불일치하는지 쉽게 찾아낼 수 있음

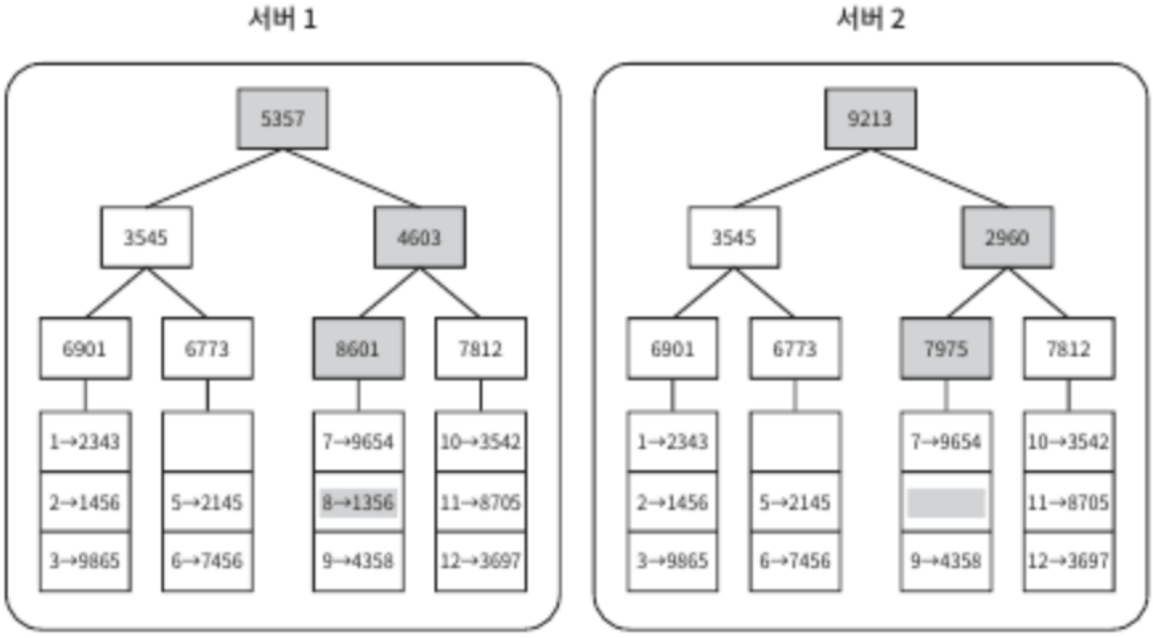


그림 6-16

### 데이터 센터 장애 처리

- 데이터를 여러 데이터 센터에 다중화하는 것이 중요

# 시스템 아키텍처 다이어그램

- 클라이언트는 키-값 저장소가 제공하는 두 가지 API, get(key) 및 put(key, value)와 통신한다.
- 중재자(coordinator)는 클라이언트에세 키-값 저장소에 대한 프락시 역할을 하는 노드
- 노드는 안정 해시의 해시 링 위에 분포

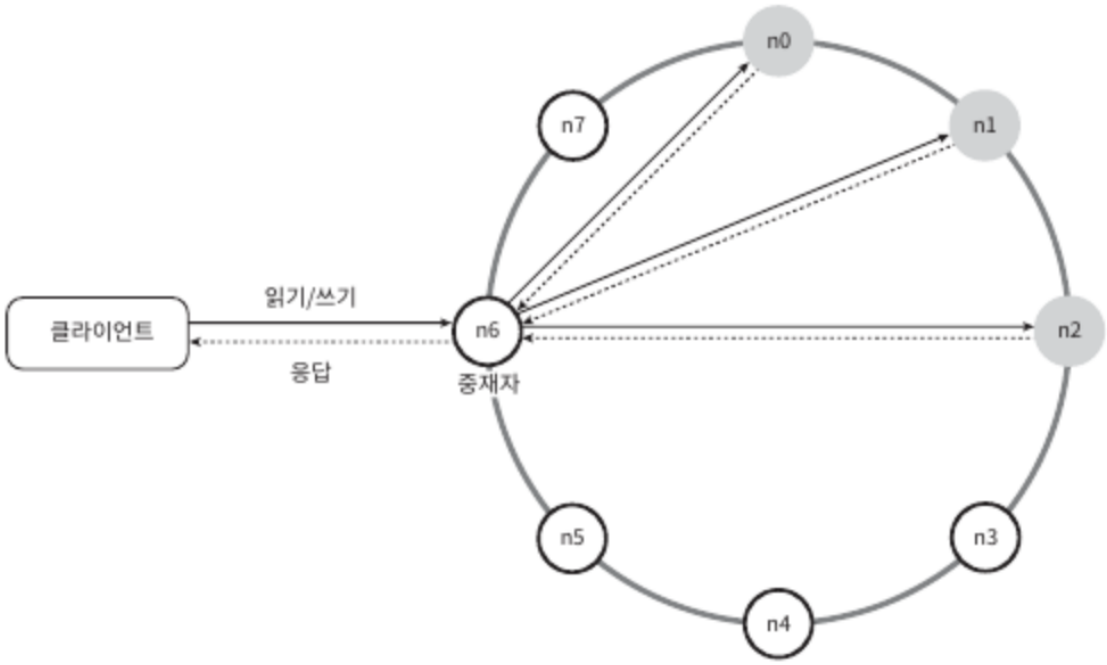


그림 6-17

- 노드를 자동으로 추가, 삭제할 수 있도록 시스템은 완전히 분산
- 데이터는 여러 노드에 다중화
- 모든 노드가 같은 책임을 지므로 SPOF(Single Point of Failure)는 존재하지 않음
- 완전 분산 설계이므로, 모든 노드는 아래와 같은 기능 전부 지원해야 함
  - 클라이언트 API

- 장애 감지
- 데이터 충돌 해소
- 장애 복구 매커니즘
- 다중화
- 저장소 엔진

# 쓰기 경로

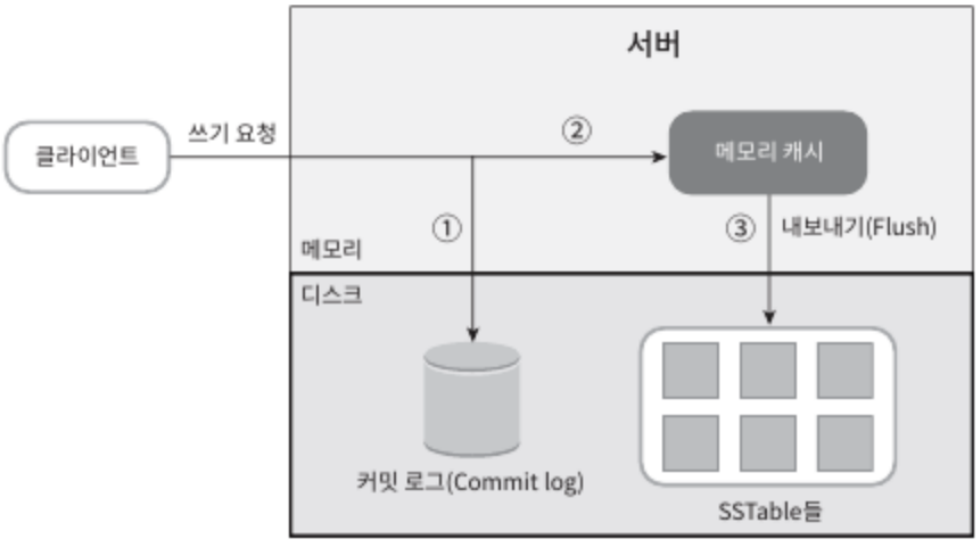


그림 6-19

1. 쓰기 요청이 커밋 로그 파일에 기록됨
2. 데이터가 메모리 캐시에 기록됨
3. 메모리 캐시가 가득차거나, 임계치에 도달하면 데이터는 디스크에 있는 SSTable에 기록됨.(Sorted-Srting Table의 약어, <키, 값>의 순서쌍을 정렬된 리스트 형태로 관리하는 테이블)

# 읽기 경로

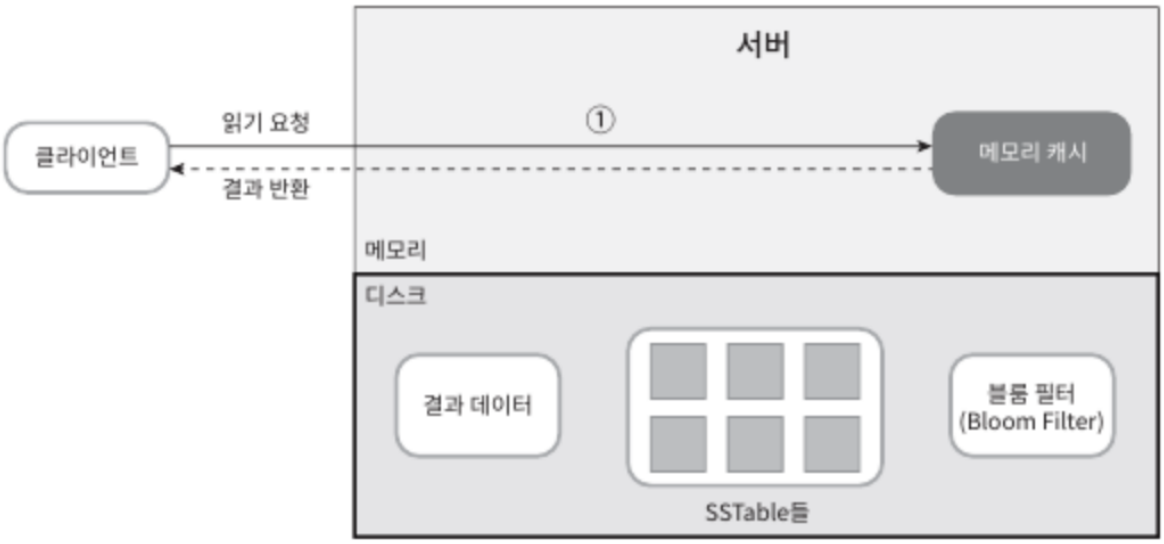


그림 6-20

- 데이터가 메모리 캐시에 있는지부터 살핌 → 지금은 캐시에 있는 사진
- 메모리에 없는 경우는 디스크에서 가져옴
  - 어느 SSTable에 찾는 키가 있는지 알아낼 효율적인 방법이 필요함
  - 블룸 필터 사용됨

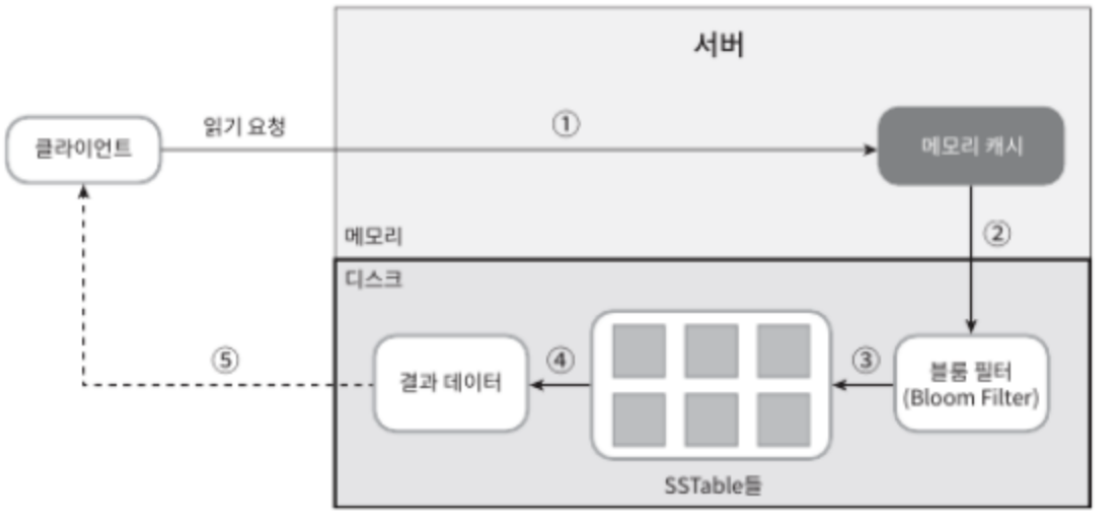


그림 6-21

1. 데이터가 메모리에 있는지 검사한다. 없으면 2로 간다.

2. 데이터가 메모리에 없으므로 블룸 필터를 검사한다.
3. 블룸 필터를 통해 **어떤 SSTable에 키가 보관되어 있는지 알아낸다**.
4. SSTable 에서 데이터를 가져온다.
5. 해당 데이터를 클라이언트에게 반환한다.