학습 내용 정리 – 2월 3주차

박시준

Table of content

Algorithm – binary search

[프로그래머스] 징검다리 건너기

[본 문제는 정확성과 효율성 테스트 각각 점수가 있는 문제입니다.]

카카오 초등학교의 "니니즈 친구들"이 "라이언" 선생님과 함께 가을 소풍을 가는 중에 **징검다리**가 있는 개울을 만나서 건너편으로 건너려고 합니다. "라이언" 선생님은 "니니즈 친구들"이 무사히 징검다리를 건널 수 있도록 다음과 같이 규칙을 만들었습니다.

- 징검다리는 일렬로 놓여 있고 각 징검다리의 디딤돌에는 모두 숫자가 적혀 있으며 디딤돌의 숫자는 한 번 밟을 때마다 1씩 줄어듭니다.
- 디딤돌의 숫자가 0이 되면 더 이상 밟을 수 없으며 이때는 그 다음 디딤돌로 한번에 여러 칸을 건너 뛸 수 있습니다.
- 단, 다음으로 밟을 수 있는 디딤돌이 여러 개인 경우 무조건 가장 가까운 디딤돌로만 건너뛸 수 있습니다.

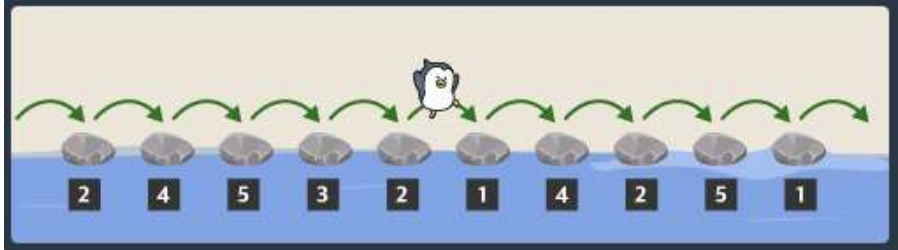
"니니즈 친구들"은 개울의 왼쪽에 있으며, 개울의 오른쪽 건너편에 도착해야 징검다리를 건넌 것으로 인정합니다.
"니니즈 친구들"은 한 번에 한 명씩 징검다리를 건너야 하며, 한 친구가 징검다리를 모두 건넌 후에 그 다음 친구가 건너기 시작합니다.

디딤돌에 적힌 숫자가 순서대로 담긴 배열 stones와 한 번에 건너뛸 수 있는 디딤돌의 최대 칸수 k가 매개변수로 주어질 때, 최대 몇 명까지 징검다리를 건널 수 있는지 return 하도록 solution 함수를 완성해주세요.

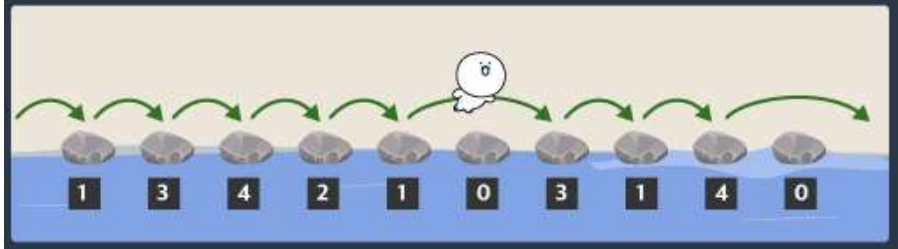
stones	k	result
[2, 4, 5, 3, 2, 1, 4, 2, 5, 1]	3	3

[프로그래머스] 징검다리 건너기

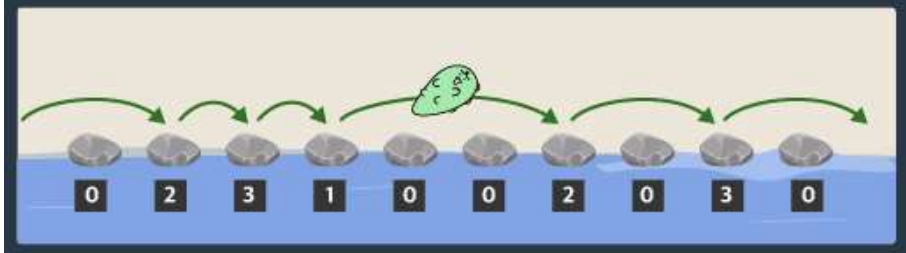
첫 번째 친구는 다음과 같이 징검다리를 건널 수 있습니다.



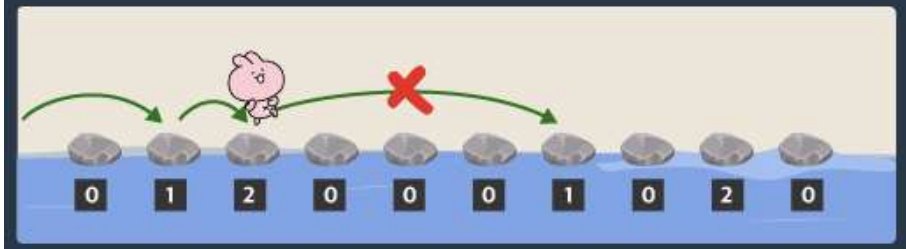
첫 번째 친구가 징검다리를 건넌 후 디딤돌에 적힌 숫자는 아래 그림과 같습니다.
두 번째 친구도 아래 그림과 같이 징검다리를 건널 수 있습니다.



두 번째 친구가 징검다리를 건넌 후 디딤돌에 적힌 숫자는 아래 그림과 같습니다.
세 번째 친구도 아래 그림과 같이 징검다리를 건널 수 있습니다.



세 번째 친구가 징검다리를 건넌 후 디딤돌에 적힌 숫자는 아래 그림과 같습니다.
네 번째 친구가 징검다리를 건너려면, 세 번째 디딤돌에서 일곱 번째 디딤돌로 네 칸을 건너뛰어야 합니다. 하지만 k = 3 이므로 건너뛸 수 없습니다.



따라서 최대 3명이 디딤돌을 모두 건널 수 있습니다.

첫 번째 시도 – 모든 징검다리 하나씩 검사하기

- 루프 하나를 돌면서 각 원소의 값(징검다리)을 검사, 만약 값이 0이면 soak 변수를 1씩 증가 시킴
- Soak 변수가 k와 동일하면 루프 끝냄. (더 이상 못 건너니까) people 개수 리턴
- 정확성 테스트는 통과했으나, 효율성 테스트 모두 실패.
- 접근법이 잘못됨

```python
def solution(stones, k):

    people = 0
    soak = 0
    i = 0
    while soak < k:
        if i == len(stones):
            i = 0
            people += 1
            continue
        if stones[i] == 0:
            soak += 1
            i += 1
            continue
        stones[i] -= 1
        soak = 0
        i += 1

    return people
```

```
테스트 14 〉 통과 (76.20ms, 10.1MB)
테스트 15 〉 통과 (148.95ms, 10.3MB)
테스트 16 〉 통과 (182.86ms, 10.1MB)
테스트 17 〉 통과 (242.69ms, 10.2MB)
테스트 18 〉 통과 (0.20ms, 10.2MB)
테스트 19 〉 통과 (0.90ms, 10.1MB)
테스트 20 〉 통과 (5.68ms, 10.2MB)
테스트 21 〉 통과 (43.38ms, 10MB)
테스트 22 〉 통과 (103.88ms, 10.3MB)
테스트 23 〉 통과 (205.91ms, 9.98MB)
테스트 24 〉 통과 (198.61ms, 10.1MB)
테스트 25 〉 통과 (0.02ms, 10MB)
```

```
효율성  테스트
테스트 1 〉 실패 (시간 초과)
테스트 2 〉 실패 (시간 초과)
테스트 3 〉 실패 (시간 초과)
테스트 4 〉 실패 (시간 초과)
테스트 5 〉 실패 (시간 초과)
테스트 6 〉 실패 (시간 초과)
테스트 7 〉 실패 (시간 초과)
테스트 8 〉 실패 (시간 초과)
```

두 번째 시도 – 이분 탐색 return mid

- 디딤돌 값 들 중 최대 값을 가져온다. 0과 최댓값의 중간 값을 people 수로 가정한다. (mid라고 표시)
- Mid를 모든 디딤돌 값에서 뺀다고 가정한다.
- 뺐을 때 디딤돌 값이 0이 되면 "soak" 개수를 올린다. Soak 개수가 k와 같아지면 루프 빠져나옴 (못 건너니까)
- Soak가 k와 같을 경우 high를 낮춰서, k보다 작으면 low를 올려서 이분탐색 재시작.
- **하지만 실패**

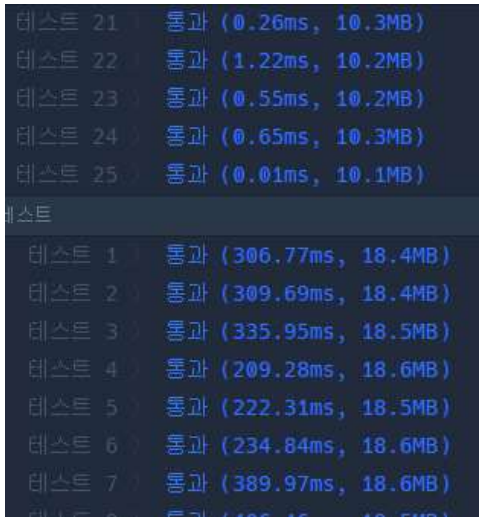
```
while low <= high:
    soak = 0
    mid = (high + low) // 2
    for stone in stones:
        if stone - mid <= 0:
            soak += 1
        else:
            soak = 0
        if soak >= k:
            break

    if soak >= k:
        high = mid - 1
    else:
        low = mid + 1

return mid
```

```
테스트 2 : 통과 (0.01ms, 10.4MB)
테스트 3 : 실패 (0.01ms, 10.1MB)
테스트 4 : 실패 (0.03ms, 10.3MB)
테스트 5 : 통과 (0.04ms, 10.1MB)
테스트 6 : 실패 (0.43ms, 10.3MB)
테스트 7 : 실패 (1.10ms, 10.2MB)
```

세 번째 시도 – 이분 탐색 return low

- Return 값은 left로
- Mid를 그대로 return 하면, 실패하지만, low(mid+1)해주면 통과함
- 최대로 건널 수 있는 사람 수가 mid와 동일할 수 있음. 하지만 최대로 건널 수 있는 사람 수가 mid인 경우 루프에서는 fail처리가 되고 high를 낮추는 작업이 들어감 그러므로 루프를 다 돌고 나온 mid는 최대치를 가질 수 없음
- 그래서 루프가 다 끝난 상태에서 low에 mid + 1을 해준 값이 최대값을 가지고 있는 것임

```
while low <= high:
    soak = 0
    mid = (high + low) // 2
    for stone in stones:
        if stone - mid <= 0:
            soak += 1
        else:
            soak = 0
        if soak >= k:
            break
    if soak >= k:
        high = mid - 1
    else:
        low = mid + 1

return low
```



예시) [4, 3, 5, 1, 2, 7, 9, 3, 2, 6]

네 번째 시도 – sliding window 힌트

- 힌트에서 슬라이딩 윈도우 문제라는 것을 알게 됨
- 모든 윈도우 내 최댓값을 구함
- 그 최댓값들 중에 최솟값이 답
- 왜냐면 그 최솟값보다 큰 값을 빼면 특정 윈도우에선 못 건넘.

이 문제에 sliding window maximum 기법이 쓰일 수 있는 이유...

저처럼 이해를 못해 고생하시는 분이 있다면, 도움이 되었으면 합니다.

leetcode 239번처럼 길이가 k 인 각 sliding window 에서 최대값을 구한 후 그 값들중 제일 작은 값이 답입니다.
(그 답의 값을 x라고 하겠습니다.)

왜냐하면,

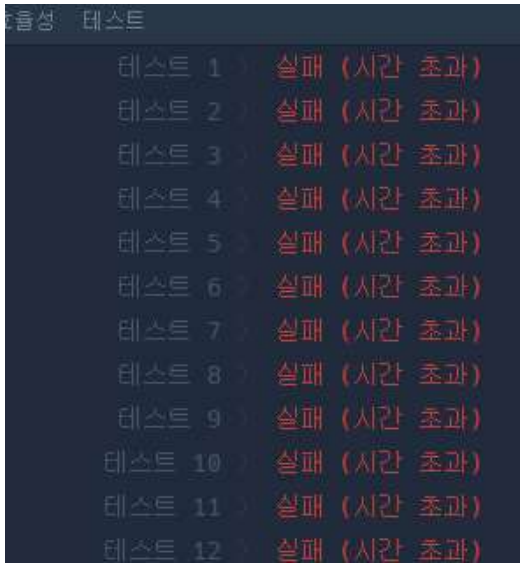
모든 stone 값에서 x 를 뺐을 때, 임의의 sliding window 에 속하는 stone 은 반드시 음이 아닌 값을 적어도 하나 갖습니다.
만약 x 보다 큰 값을 빼게 되면, 어떤 sliding window 에서는 모든 stone 값이 음수일 것입니다.
이렇게 되면 징검다리를 건널 수 없습니다. (k 보다 멀리 점프할 수 없어서)
그래서 x 가 최대 한계입니다.

(저의 지능은 여기가 한계인가 봅니다;;)

네 번째 시도 – sliding window

- 윈도우를 정해서 그 윈도우 내에서 최댓값을 구함
- 윈도우를 한 칸 씩 움직이면서 모든 경우의 최댓값 중에서 최솟값을 구함
- Max() 와 min()을 매 루프마다 쓰니까 시간초과 나는 듯.
- Heap을 쓰면 풀 수 있겠지만, 굳이 어렵게 만들 필요가?

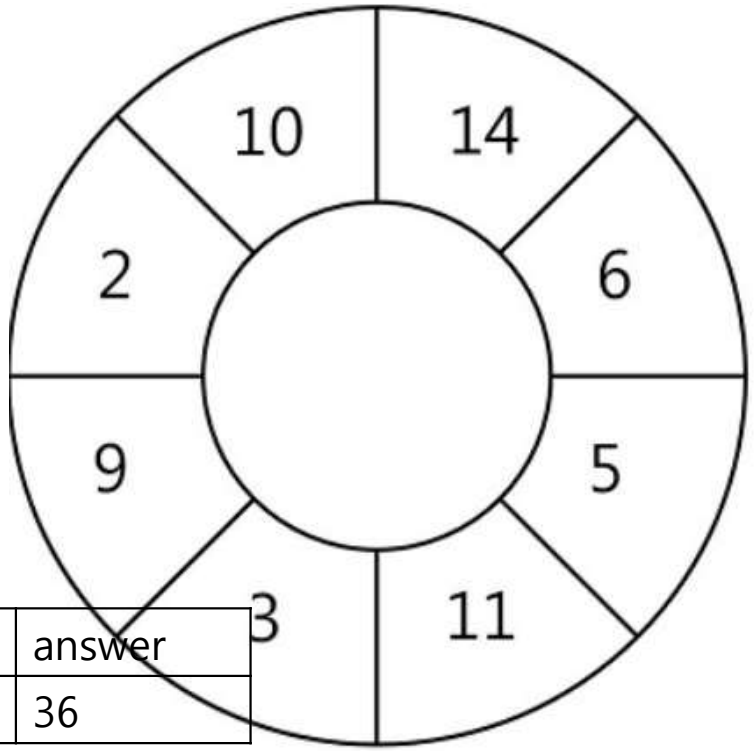
```
left = 0
right = k
maxi = 0
mini = math.inf

while right < len(stones):
    window = stones[left:right]
    maxi = max(window)
    mini = min(maxi, mini)
    left += 1
    right += 1
```



Algorithm – Dynamic programming

[프로그래머스] 스티커 모으기(2)

- 스티커가 원형으로 연결되어 있음
- 하나를 뜯으면 양 옆에 있는 것을 못 뜯음
- 뜯을 수 있는 경우의 수 중에서 얻을 수 있는 최댓값을 구해야 함.
- 아래는 예시



sticker	answer
[14, 6, 5, 11, 3, 9, 2, 10]	36
[1, 3, 2, 5, 4]	8

sticker	answer
[14, 6, 5, 11, 3, 9, 2, 10]	36

첫번째 시도 – dynamic programming

- 보자마자 다이나믹 프로그래밍이라는 생각이 들었음.
- Memory[0] = 첫번째 값
- Memory[1] = 두번째 값
- 하지만 점화식을 세우는 것에 실패
- Max를 사용하지 않고 조건문으로 처리 하려고 해서 실패 했던 것 같음

```
for i in range(2, len(sticker) - 1):
    if sticker[i] + dp[i-2] > dp[i-1]:
        dp[i] = sticker[i] + dp[i - 2]
```

두번째 시도 – DP with max()

- Max를 사용하니 전부 해결
- 이런 유형의 문제는 Max()라는 함수를 반드시 사용해야

- sticker는 원형으로 연결된 스티커의 각 칸에 적힌 숫자가 순서대로 들어있는 배열로, 길이(N)는 1 이상 100,000 이하입니다.
- sticker의 각 원소는 스티커의 각 칸에 적힌 숫자이며, 각 칸에 적힌 숫자는 1 이상 100 이하의 자연수입니다.
- 원형의 스티커 모양을 위해 sticker 배열의 첫 번째 원소와 마지막 원소가 서로 연결되어있다고 간주합니다.

```
if len(sticker) <= 2:
    return max(sticker)

#case1: tear from first sticker
dp[0] = sticker[0]
dp[1] = sticker[0]
for i in range(2, len(sticker) - 1):
    dp[i] = max(dp[i - 1], dp[i - 2] + sticker[i])
max1 = dp[len(sticker) - 2]

#case2: tear from second sticker
dp[0] = 0
dp[1] = sticker[1]
for i in range(2, len(sticker)):
    dp[i] = max(dp[i - 1], dp[i - 2] + sticker[i])
max2 = dp[len(sticker) - 1]

return max(max1, max2)
```

Max() 사용 전 고려사항

- Max()는 O(n)의 시간 소요.
- 징검다리 문제의 경우 배열의 길이가 20만 이하
- 스티커 모으기의 경우 배열의 길이가 10만
- 징검다리 문제의 경우 max()내 비교할 원소가 k개(최대 20만개)
- 스티커 모으기 문제의 경우 max 내 비교할 원소가 2개

[제한사항]

- 징검다리를 건너야 하는 니니즈 친구들의 수는 무제한 이라고 간주합니다.
- stones 배열의 크기는 1 이상 200,000 이하입니다.
- stones 배열 각 원소들의 값은 1 이상 200,000,000 이하인 자연수입니다.
- k는 1 이상 stones의 길이 이하인 자연수입니다.

- sticker는 원형으로 연결된 스티커의 각 칸에 적힌 숫자가 순서대로 들어있는 배열로, 길이(N)는 1 이상 100,000 이하입니다.
- sticker의 각 원소는 스티커의 각 칸에 적힌 숫자이며, 각 칸에 적힌 숫자는 1 이상 100 이하의 자연수입니다.
- 원형의 스티커 모양을 위해 sticker 배열의 첫 번째 원소와 마지막 원소가 서로 연결되어있다고 간주합니다.

Algorithm – Heap

[프로그래머스] 디스크 컨트롤러

하드디스크는 한 번에 하나의 작업만 수행할 수 있습니다. 디스크 컨트롤러를 구현하는 방법은 여러 가지가 있습니다. 가장 일반적인 방법은 요청이 들어온 순서대로 처리하는 것입니다.

한 번에 하나의 요청만을 수행할 수 있기 때문에 각각의 작업을 요청받은 순서대로 처리하면 다음과 같이 처리 됩니다.

- 작업의 평균 처리 시간이 최소로 나오도록 했을 때 그 최솟값을 반환해야 하는 문제
- 들어온 순서대로 요청을 처리하면 종료까지 걸린 시간의 평균은 10ms(= (3 + 11 + 16) / 3)

- 0ms 시점에 3ms가 소요되는 A작업 요청
- 1ms 시점에 9ms가 소요되는 B작업 요청
- 2ms 시점에 6ms가 소요되는 C작업 요청

[프로그래머스] 디스크 컨트롤러

- 작업의 평균 처리 시간이 최소로 나오도록 했을 때 그 최솟값을 반환해야 하는 문제
- 각 작업의 요청부터 종료까지 걸린 시간의 평균은 9ms(= (3 + 7 + 17) / 3)

- 0ms 시점에 3ms가 소요되는 A작업 요청
- 1ms 시점에 9ms가 소요되는 B작업 요청
- 2ms 시점에 6ms가 소요되는 C작업 요청

[프로그래머스] 디스크 컨트롤러

- 일단 보자마자 sort를 활용해야 함을 직감.
- 그리고 heap 모듈을 사용해서 최솟값부터 꺼내 오도록 한다.
- Heap을 넣을 때 값을 교차해서 넣어야 한다. (Heap은 첫 번째 인덱스를 보고 sorting 작업을 진행함)
- 하지만 문제를 풀진 못했다.

```python
import heapq
def solution(jobs):
    jobs.sort(key=lambda x: x[0])
    answer, second, i = 0, 0, 0
    heap = []
    while i < len(jobs):
        for j in jobs[i:]:
            if j[0] <= second:
                heapq.heappush(heap, [j[1], j[0]])
        if len(heap) > 0:
            task = heapq.heappop(heap)
            second += task[0]
            answer += (second - task[1])
            i += 1
        else:
            second += 1
    return (answer // len(jobs))
```

[프로그래머스] 디스크 컨트롤러

- Start 변수를 활용해야 하는 문제.
- Start를 사용하지 않으면 이미 작업 대기줄에 등록했던 jobs를 중복해서 등록하게 된다. Start를 사용해야 heap에 이미 한번 넣었던 변수를 제외할 수 있음

```
def solution(jobs):
    jobs.sort(key=lambda x: x[0])
    answer, second, i = 0, 0, 0
    start = -1
    heap = []
    while i < len(jobs):
        for j in jobs[i:]:
            if start < j[0] <= second:
                heapq.heappush(heap, [j[1], j[0]])
        if len(heap) > 0:
            task = heapq.heappop(heap)
            start = second
            second += task[0]
            answer += (second - task[1])
            i += 1
        else:
            second += 1
    return (answer // len(jobs))
```

Cookie

Cookie

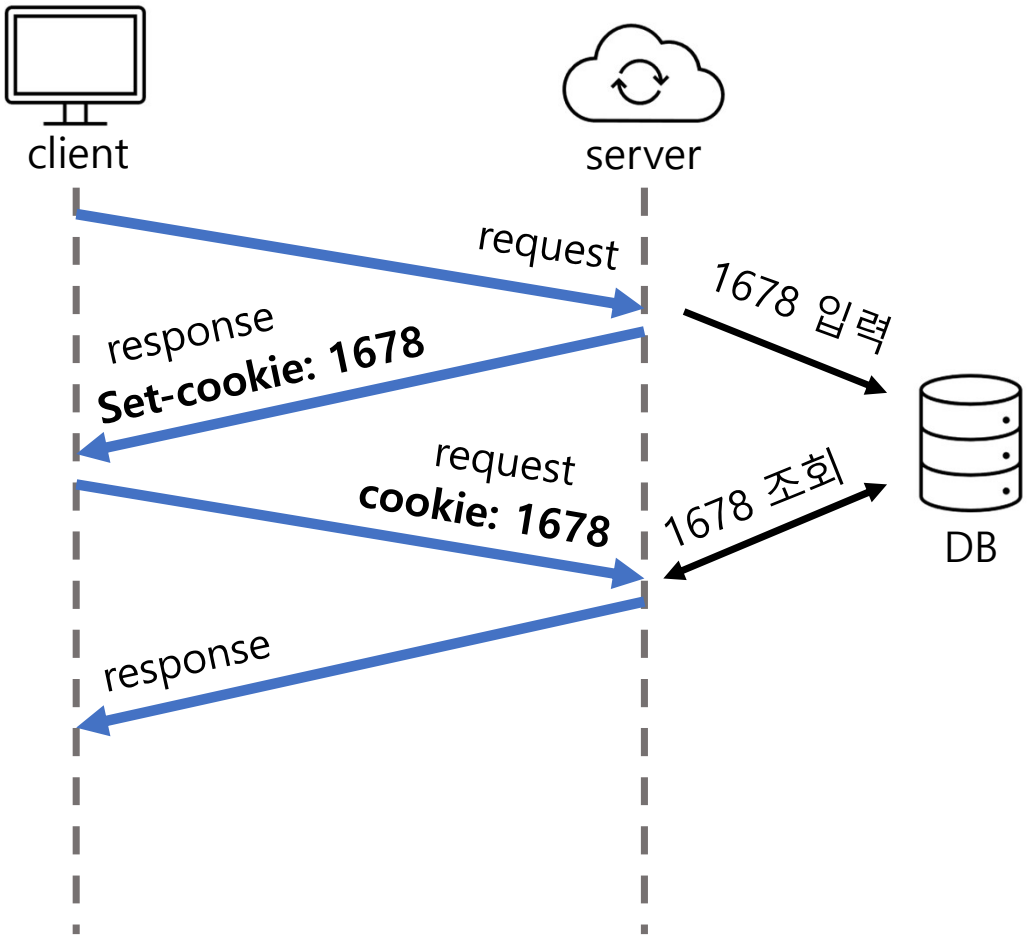
- HTTP is stateless. There is something we need to use to identify specific user.
- Cookie: small piece of data sent from server to a user's web browser. When browser send this data later with HTTP request, the server can distinguish who the user is.
- Use Case Example:

- to store user preferences, such as login information or items added to a shopping cart.

- to track user activity and gather information about how a website is being used.

Cookie operation

1. 브라우저가 서버에 request
2. 서버는 response에 Set-cookie로 헤더와 식별번호를 보낸다.
3. 브라우저는 디바이스 내 저장소에 쿠키 정보를 저장한다.
4. 그 후 브라우저는 request 보낼 때 쿠키 정보와 함께 보낸다.
5. 서버에서 특정 사용자 식별 가능



Cookie 저장소 위치

- Cookie의 저장 위치는 운영체제와 브라우저 별로 다르다.
- 구글 크롬은 다음 위치에 쿠키 저장

"C:\Users<username>\AppData\Local\Google\Chrome\User Data\Default\Cookies"

- 쿠키 파일 이름은 "cookie.txt" 또는 "cookie.jar"

Cookie 활용

- 쿠키는 특정 사용자를 기억할 수 있는 수단 (로그인을 하지 않아도, 설령 그 사용자의 이름을 몰라도 그 사용자의 쿠키는 서버에서 기억하고 있다. DB를 사용해서)
- 쿠키를 사용하면 비로그인한 사용자도 장바구니를 사용할 수 있고, 이 말은 곧 비로그인 사용자의 주문도 받을 수 있다는 뜻
- 그러다가 비로그인 사용자가 갑자기 로그인을 하면, 로그인 정보와 쿠키 정보를 매칭해서 장바구니 정보를 그대로 연관시킬 수 있음

Cache

cache

- Cache: 자주 사용되는 데이터를 임시로 복사해두는 장소
- Cache의 종류

- CPU cache
- Disk cache
- Proxy server cache
- Browser cache
- Server-side cache storage

Web-cache(proxy server)

- Web-cache

Browser cache(client-side cache)

- Cache expiration date
- 웹 브라우저가 특정 리소스에 최초로 요청했을 경우 서버는 아래와 같이 Cache-Control 헤더가 포함된 응답을 보낸다. 이런 헤더가 포함된 응답을 받으면 브라우저는 3600초 동안 응답을 캐시에 저장한다.

HTTP/1.1 200 OK

Content-Type: text/html

Cache-Control: max-age=3600

Content-Length: 157

- 3600초 안에 동일한 요청을 보내면 웹서버에 요청을 보내는 게 아니라, 캐시에 저장된 응답을 가지고 온다.

Server-side cache

- Local cache
- application에서 바로 접근. 빠름. 데이터 동기화 문제 있음
- Global cache
- 서버와 분리된 별도 저장소(cache server). 느림. 데이터 통일. Redis등

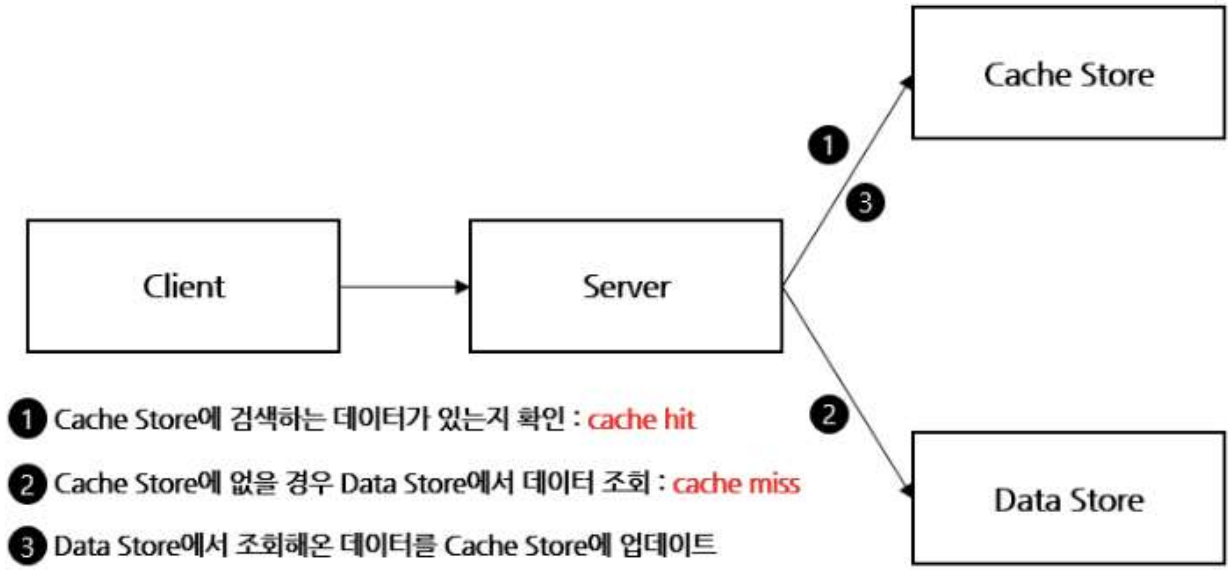
Private cache vs public cache

- Private cache: only browser can save it
- Public cache: every client and server can save it

Server-side cache 사용 전략

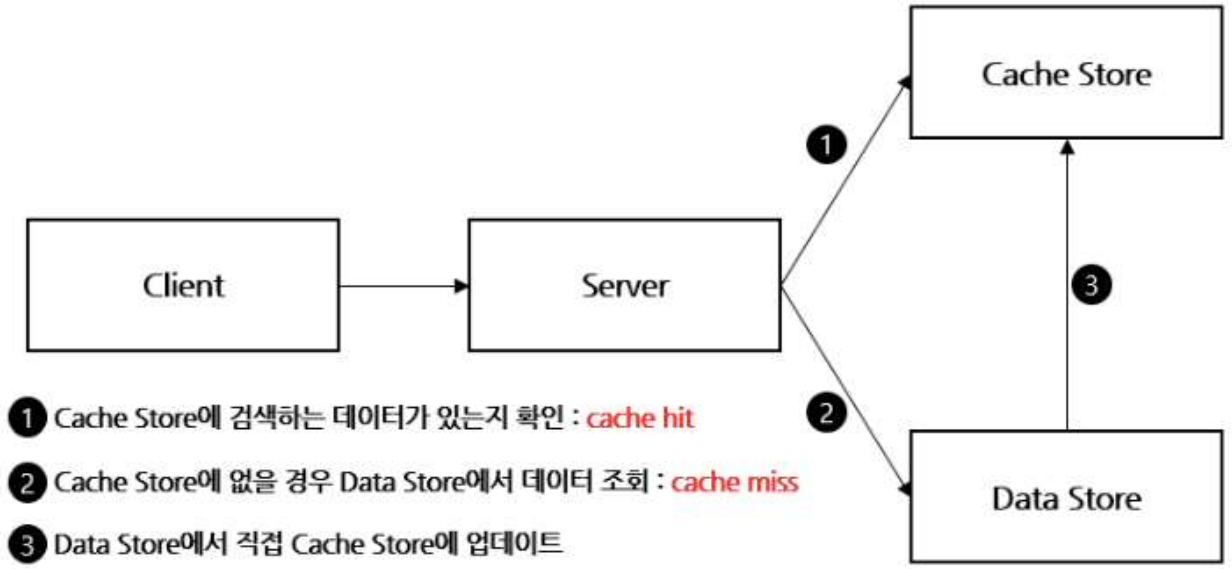
- Look aside

- 읽기에 적합, cache warming이 필요함



Server-side cache 사용 전략

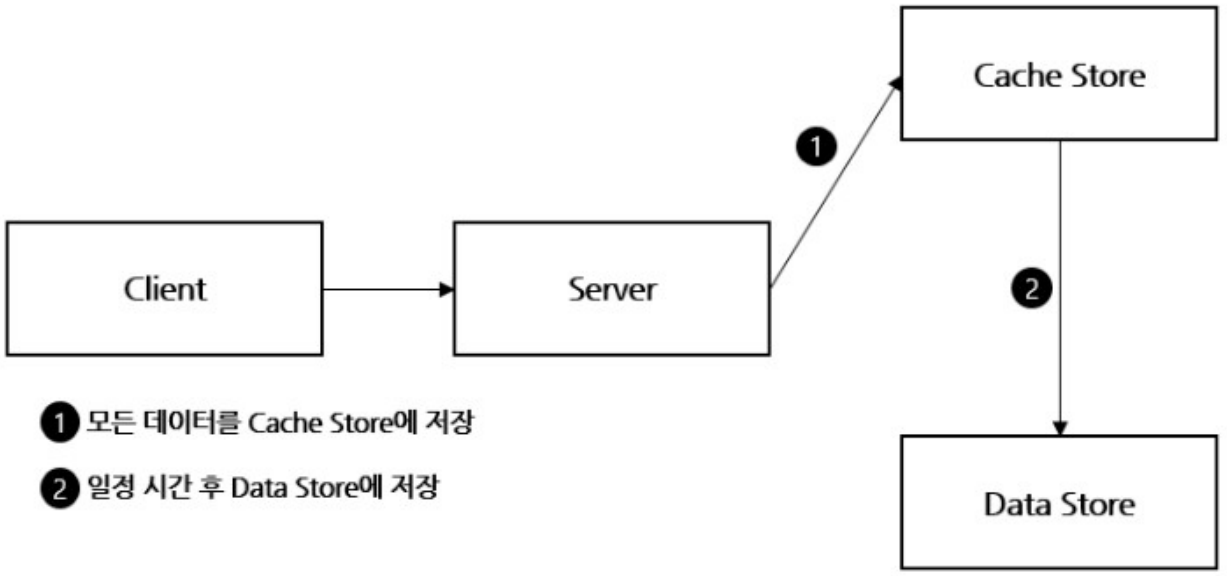
- Read through
- 읽기에 적합



Server-side cache 사용 전략

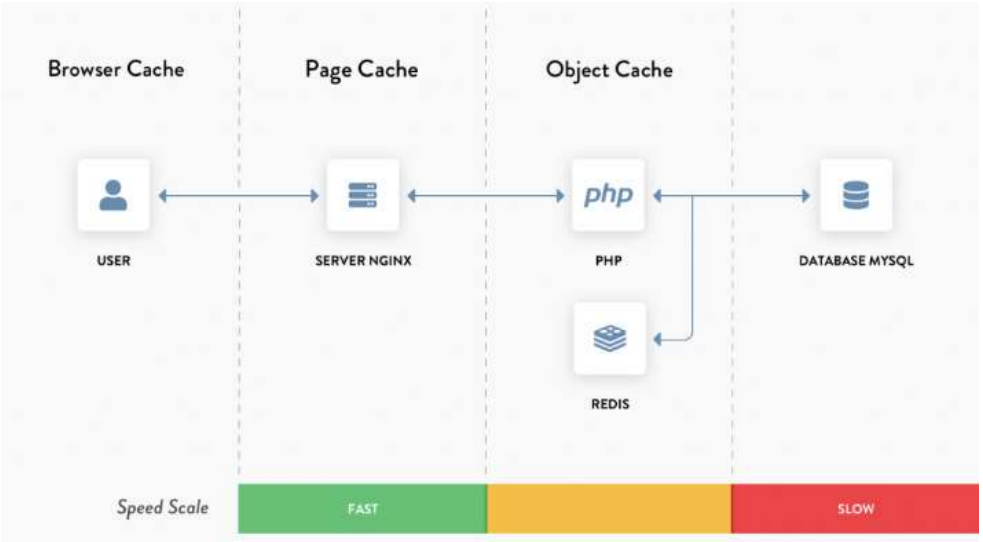
- Write back

- 캐시 장애 시 데이터 유실 가능성, 쓰기 성능 좋음



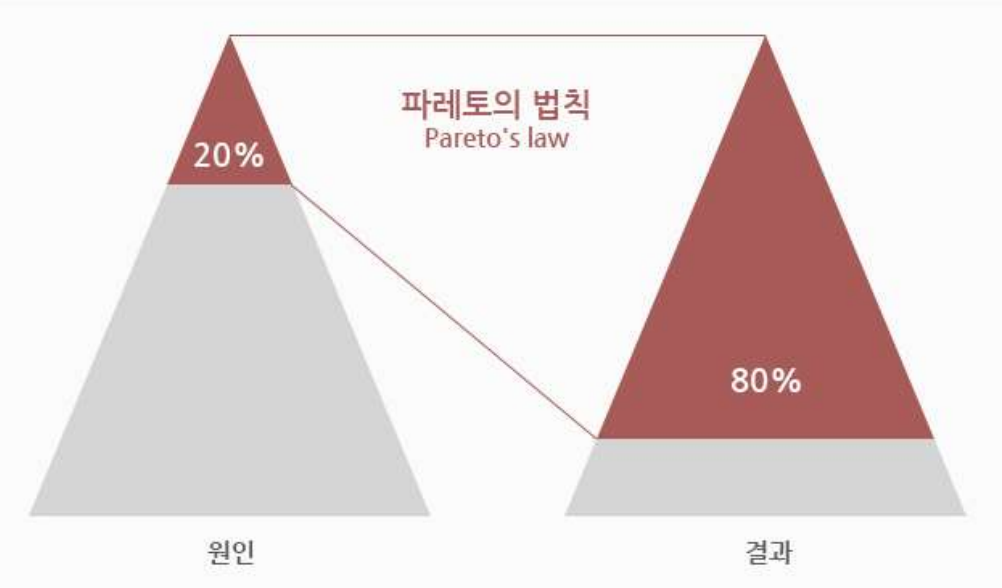
내가 오해했던 것

- "Redis는 Cache DB다"라고 생각함.
- Redis는 In-memory DB이고 Cache로 쓸 수도 있고, 영구 저장소 용도로 사용할 수도 있다.
- Cache는 역할에 따른 명칭일 뿐. Redis 자체가 Cache인 것은 아님.
- 다양한 위치에 Cache가 존재할 수 있다.



Cache의 fundamental

- 20%의 리소스가 80% 요청을 감당한다.
- 20% 기능에 cache를 사용함으로써 리소스 전송량을 대폭 줄인다.
- 한마디로 자주 사용하는 것들을 cache에 넣는다는 것.

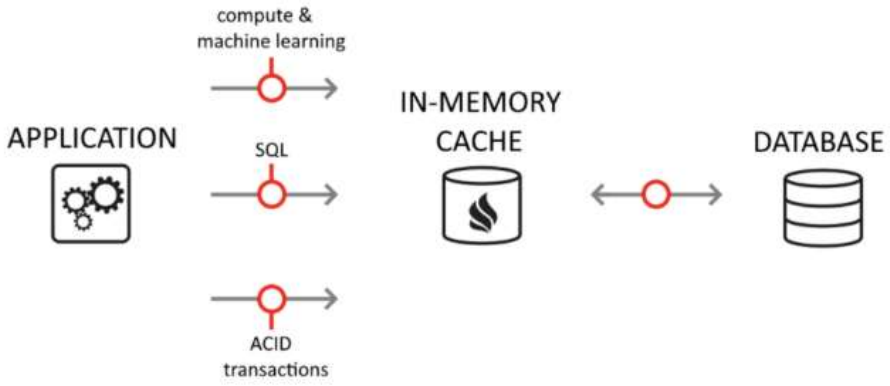


Cache의 사용 예시

- 페이스북 사용 예시
- Login을 위한 유저 정보
- 첫 화면에 보여줄 피드 몇 개
- 친구 관계 등

In-memory DB

- **Can be used as Cache**
- Redis, Memcached
- in-memory DB, cache, message broker
- Store data to RAM
- Faster but lower capacity
- 디스크가 아닌 메모리에 데이터 저장
- Key-value storage



Memcached

- String으로만 구성되어 있어서 Redis보다 빠름
- Multithread
- Persistence 지원 안 됨
- 복제 지원 안 됨
- 트랜잭션 지원 안 됨
- data eviction 방식: LRU만 가능

Redis

- hash, set, list, string 등 다양한 데이터 구조 저장
- Single thread – Atomic 보장
- Persistence – 디스크에 저장할 수 있음. 서버가 꺼지더라도 다시 데이터 불러들일 수 있음
- Replication – 하나의 인스턴스에서 다른 인스턴스로 복제할 수 있음
- 트랜잭션 지원 함
- data eviction 방식: 6가지 방식 존재
- 주요 활용 용도: remote data store. Rece condition 발생 가능성이 있는 곳에 쓰는 게 좋음. (Single thread의 이점)

Redis의 메모리 관리

- RAM공간이 부족하면 디스크와의 swap이 발생하는데 이것은 latency를 야기한다. 그리므로 메모리 관리가 매우 중요하다.
- 메모리 단편화 문제를 해결하기 위해 이왕이면 다양한 크기의 데이터 사용을 줄이고, 유사한 크기의 데이터를 저장하는 게 좋음
- 싱글 스레드이므로 동시 처리 명령어는 1개. 그러므로 O(n) 명령어는 지양하는 게 좋음. (FLUSHALL, FLUSHDB, Delete Collections, Get All Collections)
- 트랜잭션 지원

Large system requirement

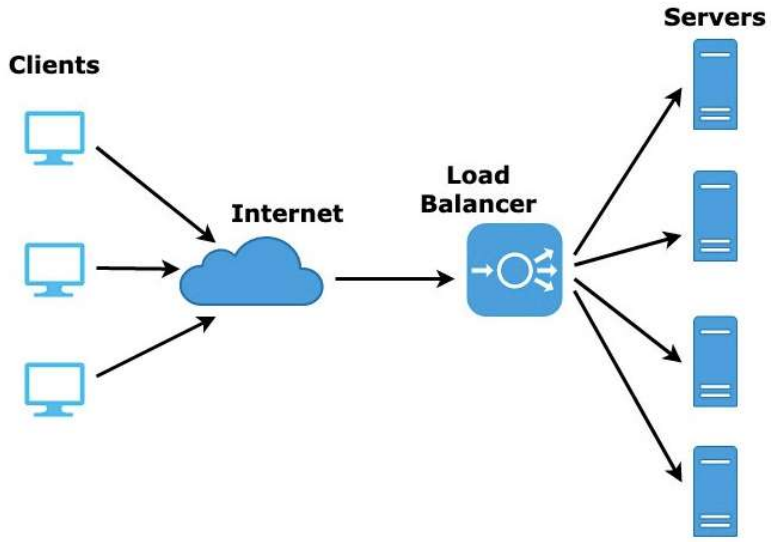
Scale up vs scale out

- Scale up: 서버에 고 사양 자원을 추가하는 행위
- Scale out: 더 많은 서버를 추가하는 행위
- Scale up이 좀 더 단순하고 쉬움. 하지만 scale up 방법에는 한계가 존재. 메모리 무한으로 넣을 수 없음. 서버 한 대 장애 나면 전체 서비스 중단.
- Scale out 했을 때, 특정 서버에 부하가 쏠리는 문제가 있을 수 있다. => load balancer 를 사용함.

Load balancer

- Load balancer를 사용하면 웹 서버는 클라이언트의 접속을 직접 처리하지 않는다.
- 로드 밸런서만 공개 ip 주소를 가지고 나머지 서버는 사설 ip주소를 가지고 있는다.
- Load balancer가 있으면 서비스 규모가 커졌을 때 서버를 하나 더 추가하면 된다.

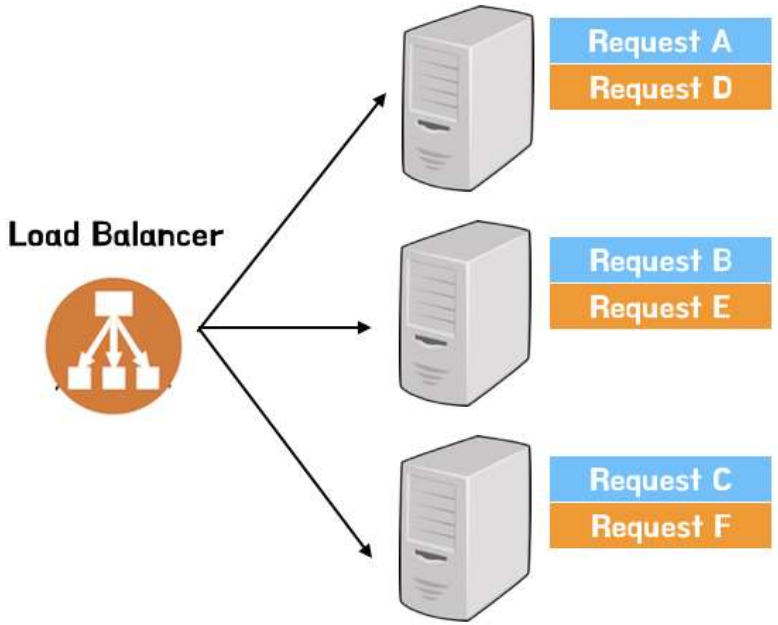
Nginx를 사용하여 로드밸런싱 할 수 있다.



Load balancing algorithm

- Load balancing에서 어느 서버에 클라이언트의 요청을 할당할지 분배하는 방식이 여러가지 존재함.
- 단순 round robin – 요청을 들어오는 순서대로 골고루 분배해주는 것
  - 가중 round robin
  - Least connection

  등등



sticky session

- 그런데 어떤 load balancing 방법을 사용하던 간에 세션과 관련되어 문제가 발생할 수 있음
- 이전에 요청해서 받은 session은 지난 번과 동일한 서버를 방문해야 유효한 것. (이전에 로그인을 했는데 또 로그인을 해야..)
- Sticky session을 사용하면, 클라이언트에게 첫 응답을 준 서버에게 클라이언트는 껌딱지처럼 붙어있게 된다. 특정 세션의 요청을 그 세션을 처음 처리한 서버에게만 주는 것임.
- 하지만 이것의 단점도 있는데 특정 서버에만 요청이 과부하될 수도 있고, 만약 특정 서버가 죽는다면 거기 붙어있던 세션들은 모두 죽는다.

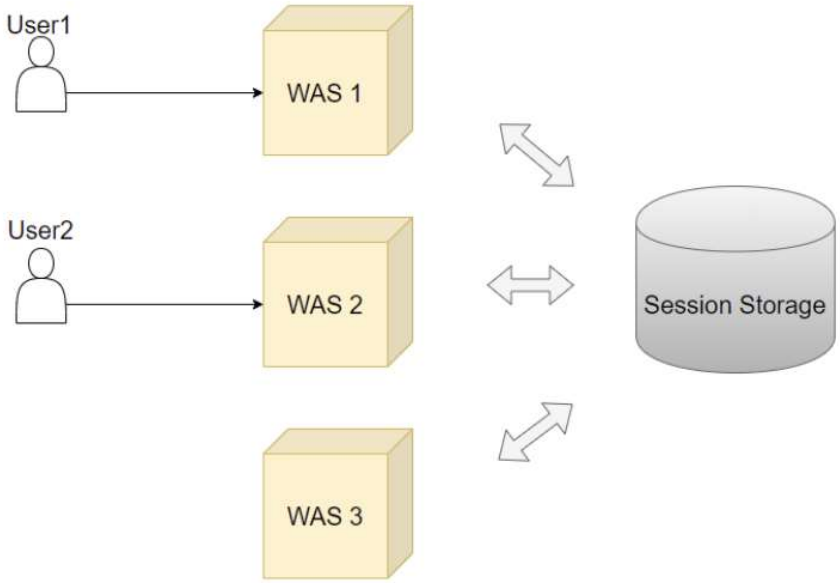
Session clustering

- Sticky session의 단점을 cover
- 각 WAS는 세션을 각각 가지고 있지만, 이를 하나로 묶어서 하나의 클러스터로 관리한다.
- All to All Session Replication – 모든 변경사항이 나머지 저장소에도 복제가 됨. 전부 복제하면 저장소 용량 부족. 대규모에는 부적합
- Primary-secondary Replication - primary는 secondary에 key만 복제.

언제나 Primary에 접근을 해야 하는 단점 존재.

Session storage 분리 (무상태 아키텍처)

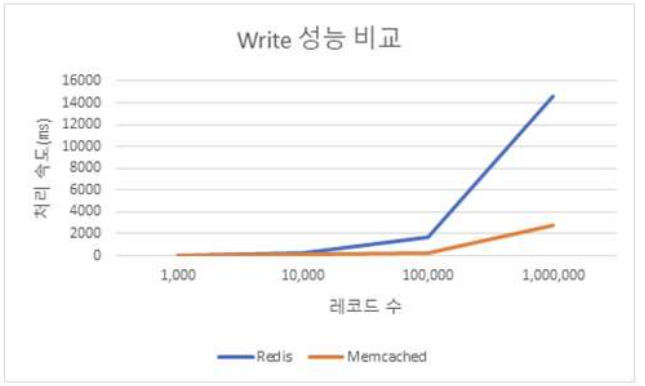
- 세션 저장소를 아예 서버와 분리한다. (DB와도 분리한다.)
- 서버가 세션 정보를 들고 있지 않으므로 무상태 아키텍처라고 한다.
- Sticky session처럼 특정 서버에 트래픽이 몰리는 문제는 없음
- 서버 하나가 장애가 나도 서비스 유지할 수 있음
- 세션 저장소로 가장 많이 사용되는 것이 바로 Redis와 MemCached
- 물론 그냥 Disk DB(NoSQL)를 사용할 수도 있다.



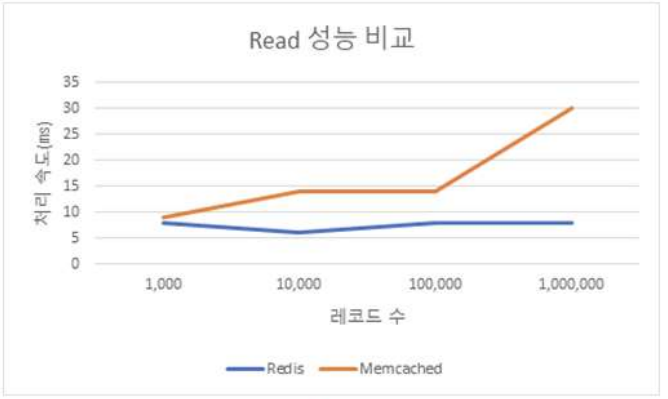
Session storage 채택

- Redis와 Memcached 각자 동작 방식이 달라서 장단점이 있으나, 쓰기 읽기 성능만 비교해본다면..
- Write 성능은 Memcached 승
- Read 성능은 Redis 승
- Session은 읽는 작업이 압도적으로 많으므로 Redis 사용이 적합함
- 그리고 spring은 redis API를 지원함

https://www.sciencedirect.com/science/article/pii/S1319157816300453



쓰기 성능 비교



읽기 성능 비교

데이터베이스 다중화

- 웹 서버 뿐만 아니라 데이터베이스도 백업이 필요함
- 데이터 서버를 지역적으로 다중화해서 재해에 따른 리스크를 감소, 쿼리를 병렬로 처리할 수 있으므로 성능이 좋아진다.
- 주로 master-slave 관계 설정. 데이터 **원본은 주 서버**에, **사본은 부서버**에 저장하는 방식
- 쓰기는 master에서만 가능. Slave는 그 사본을 전달 받기만 하고, 데이터 읽기만 지원한다.

데이터베이스 샤딩

- 데이터베이스를 증설하는 방법 -> 수평적 확장 (샤딩)
- 다중화랑 다른 점은 똑같은 정보를 복제하는 게 아님. 각 데이터베이스에 들어있는 데이터가 다름. 모든 샤드는 같은 스키마를 쓰지만 샤드 사이의 데이터 중복은 없다.
- 샤딩을 할 때 가장 중요한 것은 샤딩 키를 어떻게 정하는지다

Ex) user_id%4

- 샤딩을 하면 여러 샤드에 걸친 데이터를 조인하기가 어렵다. 이를 해결하기 위해서 데이터베이스를 비정규화하여 하나의 테이블에서 질의가 수행될 수 있도록 한다.

메시지 큐

- 메시지 큐는 메시지의 무손실을 보장하는, 비동기 통신을 지원하는 컴포넌트.
- 생산자가 메시지 큐를 발행하면, 소비자는 그 메시지를 받아 알맞은 동작을 수행한다.
- 소비자 서버의 프로세스가 다운되어 있어도 생산자는 메시지를 발행할 수 있고, 소비자는 생산자가 가용한 상태가 아니더라도 메시지를 수신할 수 있다.
- 웹서버는 시간이 오래 걸리는 작업(사진 보정)을 메시지 큐에 넣는다 사진 보정 작업 프로세스들은 이 작업을 메시지 큐에서 꺼내어 비동기적으로 완료한다. 이렇게 하면 생산 서버와 소비 서버는 각자 독립적으로 움직일 수 있다.

자동화

- 시스템이 크고 복잡해지면 생산성을 높이기 위해 자동화 도구를 활용해야 한다.
- 빌드, 테스트, 배포 등의 절차를 자동화 할 수 있다.
- Jenkins, Docker

캐시

- 자주 참조되는 데이터를 메모리 안에 두고 빨리 처리되도록 하는 저장소. 새로 고침 자주해도 DB를 호출하지 않도록 하기 위해서 사용함.
- 데이터 쓰기는 자주 일어나지 않지만, 읽기가 자주 일어난다면 사용할만함.
- 캐시는 DB의 데이터를 대체하여 제공하기 때문에 데이터 일관성을 위해서 적당한 만료기한을 가져야 한다. 너무 길면, 데이터 차이 남.
- 캐시도 웹 서버나 DB처럼 다운될 수 있다. 캐시도 분산하는 게 좋음
- 캐시가 꽉차면 기존 데이터를 삭제해야한다. 캐시 삭제 정책도 신경을 쓴다.

Calculate QPS and Server, DB

개략적인 규모 추정

- QPS(query per second) = 일단 능동 사용자(DAU) / 24시간 / 3600초 * 사용자당 평균 요청 수
- 최대 QPS = 전체 사용자 수 / 24시간 / 3600초 * 사용자당 평균 요청 수
- 일간 미디어 저장을 위한 저장소 요구량 = DAU * 사용자당 평균 요청 수 * 요청에서 미디어 사용률 * 미디어별 할당 메모리 크기(MB)
- 5년간 미디어 보관하기 위한 저장소 요구량 = 5 * 365 * 일간 미디어 저장을 위한 저장소 요구량

서버 대수 계산 예시

- 1초에 10만개 요청
- 서버는 32개의 멀티코어
- 각각의 요청처리에 100ms 소요

How to calculate how many servers are needed for an specific QPS (queries per second)

5

Back of envelope calculations: Imagine that I have some requirements and after some calculations, I get that I will get about 100k request per second to my server. How many servers would I need? Let's say server has 32CPUs and each request takes 100ms

Thanks

architecture system-design

서버 대수 계산 예시

- CPU 위주 경우 320 요청 처리
- 메모리 위주 RAM은 16GB Memory 소모 각 40MB 4000 요청 처리

If the request is CPU bound , the generic formula as below can be used :

Max number of requests per second = Number of CPU Cores / Average request(task) time(seconds)

If you have a server with 32 CPU cores and if every task consumes 100 ms then, then you can expect the CPU to handle approximately 32 CPU cores / 0.1 seconds = 320 requests per second.

If the request is memory bound, the generic formula as below can be used:

Max number of requests per second = (Total RAM / worker memory) * (1 / task time)

If the total RAM memory is 16Gb and the worker/process memory consumption is 40Mb and the task consumes 100ms, then the maximum number of request that you can expect the CPU can handle shall be 4000 requests per second.

RDBMS vs NoSQL

ACID

- **A – Atomicity:** "All or nothing"

transaction의 시작과 종료사이에 일어난 data의 변경은 정상적이면 모두 저장(Commit)되어야 하고, 문제가 있으면 모두 취소(Rollback)되어야 한다.

- **C – Consistency:** "항상 유효한 데이터만 저장"

Data 저장 시 엄격한 규칙을 적용하여 DB가 안 깨지도록 보장하는 성질

- **I – Isolation:** "다른 트랜잭션에 영향 최소화"

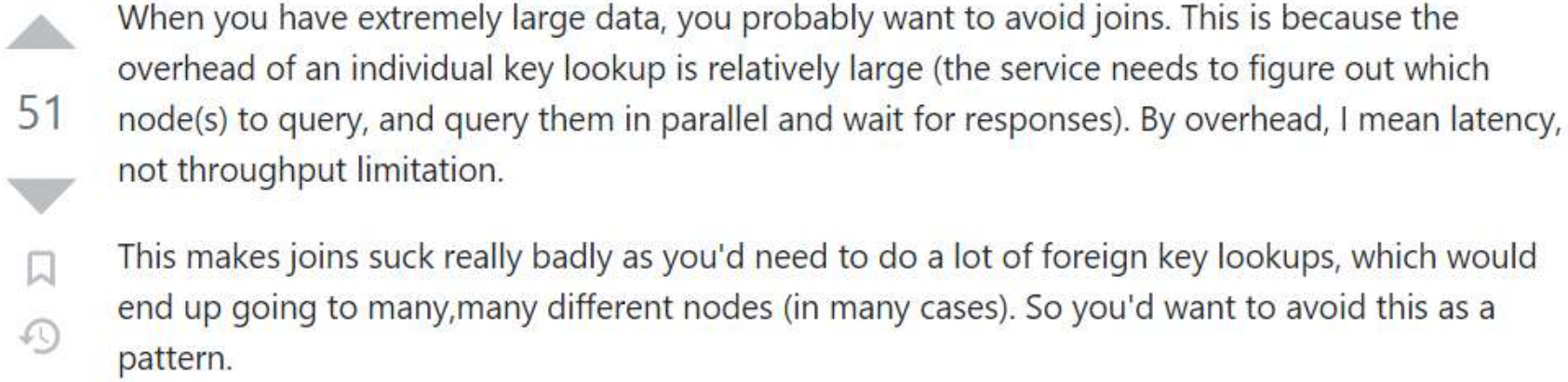
여러 개의 transaction이 동일한 DB를 CRUD할 때 중간 처리 결과를 참조하지 못하게 하여 오류를 방지하는 성질

- **D – Durability:** "영구보관 보장"

Commit된 데이터는 장애가 나도 저장되며 수동으로 지우지 않는 한 영구적으로 보관된다는 성질

RDBMS Join의 단점

- 데이터 규모가 매우 큰 경우



51

When you have extremely large data, you probably want to avoid joins. This is because the overhead of an individual key lookup is relatively large (the service needs to figure out which node(s) to query, and query them in parallel and wait for responses). By overhead, I mean latency, not throughput limitation.

This makes joins suck really badly as you'd need to do a lot of foreign key lookups, which would end up going to many,many different nodes (in many cases). So you'd want to avoid this as a pattern.

Denomalization

- Denomalization은 nomalization의 반댓말로 같은 데이터를 중복해서 저장하는 방식을 말함.
- NoSQL에서는 join 기능이 지원되지 않으므로(정규화되어있지 않으므로) 두 테이블 join하여 데이터 가져오려면 어플리케이션에서 두 번의 쿼리가 사용된다.
- 이런 두 번의 I/O를 방지하기 위해서 처음부터 별도의 데이터 테이블을 만들어 필요한 데이터들을 join시켜 기록해두는 것이다. 그러면 한 번의 쿼리로도 원하는 데이터를 가져올 수 있다.
- 단점으로는 데이터 일관성의 문제가 발생할 수 있음. 여러 테이블을 동시에 업데이트 해줘야하는데, 특정 테이블을 업데이트하다가 서버가 죽으면 데이터 불일치 문제 발생. 그리고 별도의 테이블을 생성하기 때문에 스토리지 사용량이 증가한다.

Calculate number of server

RPS

- RPS(request per second): the figure to measure the network throughput. Main metric.
- Those requests can be pure HTTP requests or can be other kind of server requests. Database queries, fetch the mail, bank transactions, etc.
- QPS(query per second)

I/O bound (memory bound)

- There are two types of request: CPU bound and I/O bound(memory bound)
- Typically, requests are limited by I/O
- CPU is doing nothing most of the time

RPS = (memory / worker memory) * (1 / Task time)

CPU bound

- image processing or doing calculations are CPU bound
- Having a lot of workers does not help, as only one can work at the same time per core
- The limit here is CPU power and number of cores

RPS = Num. cores * (1 /Task time)

Rule of thumb

- image processing or doing calculations are CPU bound
- Having a lot of workers does not help, as only one can work at the same time per core
- The limit here is CPU power and number of cores

RPS = Num. cores * (1 /Task time)

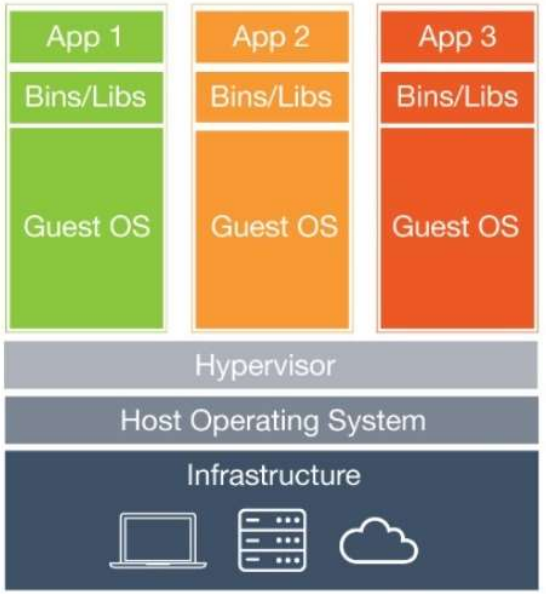
Docker

Docker의 용도

- Docker: open-source container platform
- 용도: environmental disparity 를 해결해준다.
- 예를 들어 내 컴퓨터는 윈도우인데 서버는 리눅스. 둘의 OS 환경이 달라서 코드가 동작을 안 할 수도 있음. 이것이 environmental disparity. Docker를 사용하면 다른 기기에서도 개발했던 것과 같은 서버 환경을 구축할 수 있다.

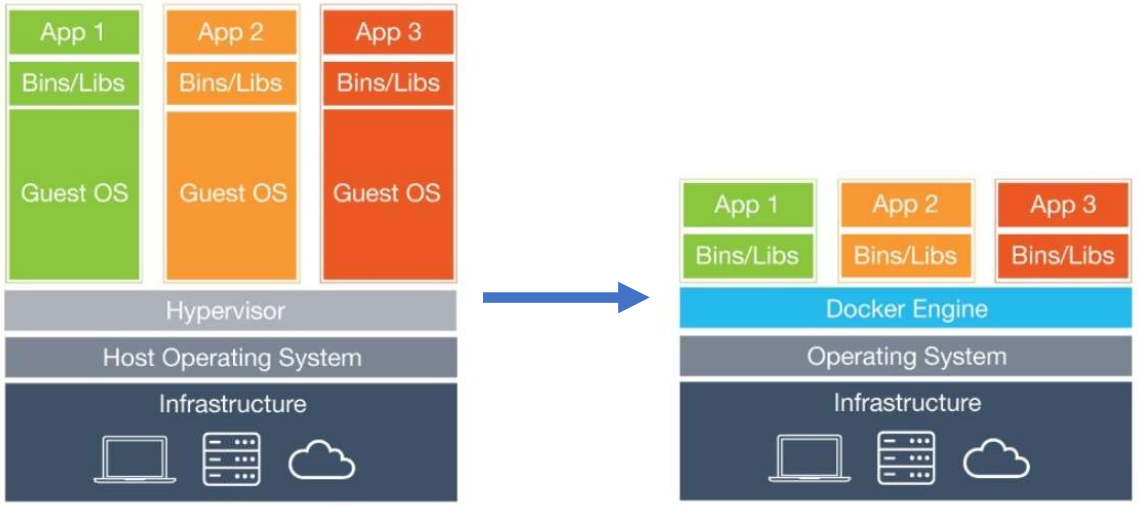
Docker를 사용 이전

- 서버와 개발 컴퓨터의 OS가 다르다면 각 서버에 개발환경과 유사한 상태로 만들도록 별도의 가상머신을 구동하고 거기에 OS를 설치해서 돌릴 수 있다. 다만 이렇게 OS를 올리면 용량도 잡아먹고 속도도 느려짐.



Docker를 사용 이후

- 도커를 사용하면 OS 없이도 사용환경을 그대로 따라 구축해서 동작시킬 수 있다. 서버는 그냥 실행파일과 라이브러리만 있으면 된다. 이렇게 별도의 환경을 구축해주는 것을 컨테이너라고 한다. 이렇게 컨테이너를 구축해주는 프로그램 중에 하나가 바로 "도커"

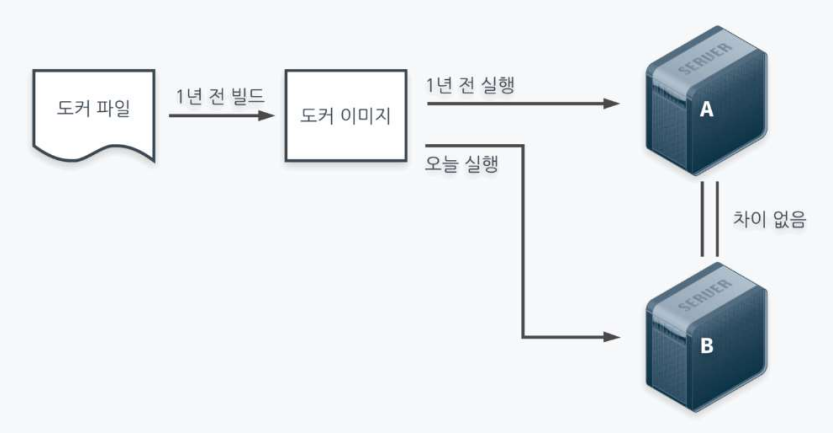


Docker의 사용법

1. 개발 컴퓨터와 서버에 Docker를 설치.
2. Docker 파일을 생성해서 구현하고 싶은 환경을 설정함. (우분투, 자바, git 등등)
3. Docker는 그 파일을 보고 설정대로 프로그램을 다운 받고 환경을 설정한다.
4. 하나의 서버는 여러 환경설정(컨테이너)을 가질 수 있고, 각 설정들은 나머지 다른 설정들과는 독립적이라서 영향 받지 않는다. 모든 컨테이너는 독립적으로 운용할 수 있음

Docker의 image와 container

- 도커 파일 = 서버의 운영 기록을 코드화 한 것.
- 도커 이미지 = 도커 파일 + 실행 시점
- 도커 컨테이너 = 도커 이미지 + 환경 변수
- 도커 이미지에는 실행 시점 정보가 포함되므로 서버 구성에 차이가 없어진다.
- 도커 이미지를 실행한 것이 도커 컨테이너



Docker image와 container

- Program이 여러 개의 process를 실행시킬 수 있는 것처럼 하나의 docker image가 여러 개의 container를 실행 시킬 수 있다
- Docker hub에서 필요한 image를 다운 받을 수 있다.