학습 내용 정리 - 3월 1주차

박시준

Table of content

Algorithm – binary search

[프로그래머스] 입국심사

- 문제에 binary search라고 쓰여있었음에도 불구하고 풀지 못했던 문제
- 어떤 것을 기준으로 놓고 이분 탐색 해야 하는지 몰라서 풀지 못했음

입국심사

문제 설명

n명이 입국심사를 위해 줄을 서서 기다리고 있습니다. 각 입국심사대에 있는 심사관마다 심사하는데 걸리는 시간은 다릅니다.

처음에 모든 심사대는 비어있습니다. 한 심사대에서는 동시에 한 명만 심사를 할 수 있습니다. 가장 앞에 서 있는 사람은 비어 있는 심사대로 가서 심사를 받을 수 있습니다. 하지만 더 빨리 끝나는 심사대가 있으면 기다렸다가 그곳으로 가서 심사를 받을 수도 있습니다.

모든 사람이 심사를 받는데 걸리는 시간을 최소로 하고 싶습니다.

입국심사를 기다리는 사람 수 n, 각 심사관이 한 명을 심사하는데 걸리는 시간이 담긴 배열 times가 매개변수로 주어질 때, 모든 사람이 심사를 받는데 걸리는 시간의 최솟값을 return 하도록 solution 함수를 작성해주세요.

제한사항

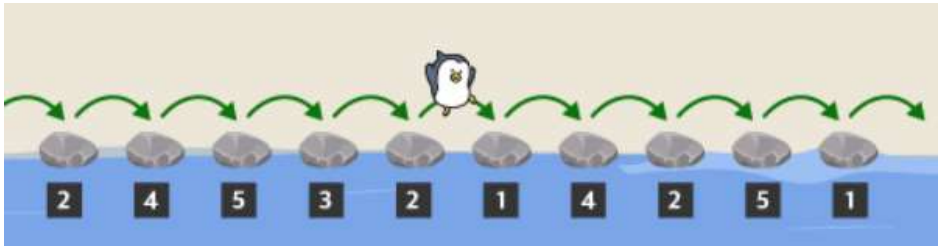
- 입국심사를 기다리는 사람은 1명 이상 1,000,000,000명 이하입니다.
- 각 심사관이 한 명을 심사하는데 걸리는 시간은 1분 이상 1,000,000,000분 이하입니다.
- 심사관은 1명 이상 100,000명 이하입니다.

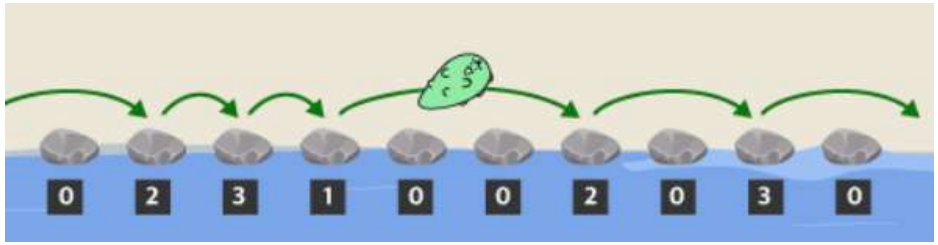
입출력 예

n	times	return
6	[7, 10]	28

이진탐색 - 징검다리 건너기

- 이 문제 역시 '징검다리 건너기' 문제 처럼 주어진 배열을 정렬해서 이분 탐색하는 것이 아니라 별도의 가상의 세로축 배열을 가정하고 두어 이진탐색해야 하는 문제였다.



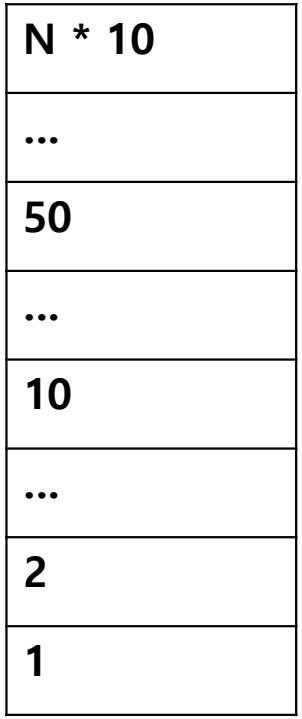


200,000,000
…
20,000
…
100
…
2
1

2	4	5	3	2	1	4	2	5	1

[프로그래머스] 입국심사

- 이 문제 역시 '징검다리 건너기' 문제 처럼 주어진 배열을 정렬해서 이분 탐색하는 것이 아니라 별도의 가상의 세로축 배열을 가정하고 두어 이진탐색해야 하는 문제였다.
- 최댓값은 가장 오래 걸리는 심사관에게 모든 사람들이 심사를 받는다고 했을 때 소요되는 시간으로 설정한다.
- N명 = 심사 받아야 하는 사람 수
- 10분 = 가장 오래 걸리는 심사관
- 여기까지는 오케이. 하지만 이 이상은 못 함.



7	10

못 풀었던 이유

- While 문 안에 어떤 알고리즘이 들어가야 하는지 알 수 없었음.
- 각 인원수를 어떻게 분배하는가에 따라 소요시간이 달라지는데, 이것을 어떻게 분배하는가에 초점을 맞추다 보니, 너무 복잡하게 생각한듯
- 사람이 실제 배정할 때 방식처럼 생각하다 보니 mid를 제대로 활용할 방법이 떠오르지 않았던 것.

그 때 당시 생각:
'어느 심사대에 사람을 우선적으로 배정하면서 시간을 늘려가야 하지?'

```python
def solution(n, times):
    spent = 0
    left = 0
    right = n * max(times)

    while left <= right:
        mid = left + (right - left) // 2
```

주어진 국민 6명, 심사대 [7, 10] ➡ 7 * 4 = 28, 10 * 2 = 20 ➡ Max: 28분

주어진 국민 7명, 심사대 [7, 10] ➡ 7 * 4 = 28, 10 * 2 = 30 ➡ Max: 30분

주어진 국민 8명, 심사대 [7, 10] ➡ 7 * 5 = 35, 10 * 2 = 30 ➡ Max: 35분

못 풀었던 이유

- 이 문제에는 함정이 있다. '1분을 더 기다린 후' 라는 문장을 보고 각 심사대에 어떻게 사람을 배정할지에 대해 초점을 맞추면 이 문제는 풀 수가 없다.
- 하지만 이 문제를 풀려면 남은 인원수를 어느 심사대에 배정할지에 초점을 두는 게 아니라, **주어진 시간 안에 각 심사대는 몇 명의 사람을 심사할 수 있는지에 초점을 둬야 함.**

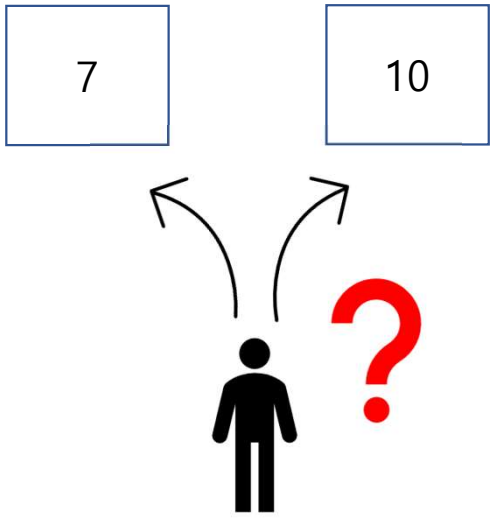
가장 첫 두 사람은 바로 심사를 받으러 갑니다.

7분이 되었을 때, 첫 번째 심사대가 비고 3번째 사람이 심사를 받습니다.

10분이 되었을 때, 두 번째 심사대가 비고 4번째 사람이 심사를 받습니다.

14분이 되었을 때, 첫 번째 심사대가 비고 5번째 사람이 심사를 받습니다.

20분이 되었을 때, 두 번째 심사대가 비지만 6번째 사람이 그곳에서 심사를 받지 않고 1분을 더 기다린 후에 첫 번째 심사대에서 심사를 받으면 28분에 모든 사람의 심사가 끝납니다.



입국심사 while 안에 들어가야 하는 알고리즘

- 총 소요시간을 먼저 고정한다. (mid)
- 총 소요시간에서 각 심사관마다 걸리는 시간을 나누면 각 심사관마다 심사할 수 있는 인원 수가 나온다.
- 각 심사관마다 심사할 수 있는 인원수를 모두 합쳐서 총 인원 수를 구한다.
- 총 인원 수가 주어진 n보다 크면 mid를 줄여서 이진 탐색을 진행,
- 총 인원 수가 주어진 n보다 작으면 mid를 키워서 이진 탐색을 진행한다.
- 발상만 제대로 하면 쉬운 문제

```python
def solution(n, times):
    left = 1
    right = n * max(times)

    while left <= right:
        mid = left + (right - left) // 2
        total = 0

        for time in times:
            total += mid // time

        if total >= n:
            right = mid - 1
        else:
            left = mid + 1

    return left
```

Algorithm – palindrome

[프로그래머스] 가장 긴 팰린드롬

- 배열 s가 주어짐
- 앞뒤를 뒤집었을 때 같으면 팰린드롬
- 가장 긴 팰린드롬 길이를 반환하는 문제
- 분명 예전에 leetcode에서 풀었던 문제인데, 발상이 떠오르지 않아서 못 풀었던 문제 (투포인터였는지 기억이 가물가물)

가장 긴 팰린드롬

문제 설명

앞뒤를 뒤집어도 똑같은 문자열을 팰린드롬(palindrome)이라고 합니다.
문자열 s가 주어질 때, s의 부분문자열(Substring)중 가장 긴 팰린드롬의 길이를 return 하는 solution 함수를 완성해 주세요.

예를들면, 문자열 s가 "abcdcba"이면 7을 return하고 "abacde"이면 3을 return합니다.

제한사항

- 문자열 s의 길이 : 2,500 이하의 자연수
- 문자열 s는 알파벳 소문자로만 구성

[프로그래머스] 1차 시도 - 스택

- 스택을 사용한 풀이
- 하지만 대부분 실패
- 일단 짝수 팰린드롬은 이걸로 못 푼다.
- 홀수 팰린드롬이라 하더라도
- “abcbawabcba”에서 11이 아닌 5를 반환하는 잘못된 코드

```python
def solution(s):
    answer = 1
    count = 1
    stack = []

    for i in range(len(s) - 1):
        if len(stack) and s[i + 1] == stack[-1]:
            stack.pop()
            count += 2
        else:
            stack.append(s[i])
            count = 1
        answer = max(count, answer)

    return answer
```

```
테스트 1 : 통과 (0.01ms, 10.2MB)
테스트 2 : 통과 (0.01ms, 10MB)
테스트 3 : 통과 (0.03ms, 10.1MB)
테스트 4 : 실패 (0.03ms, 10.2MB)
테스트 5 : 실패 (0.03ms, 10.1MB)
테스트 6 : 실패 (0.03ms, 10.1MB)
테스트 7 : 실패 (0.03ms, 10.3MB)
테스트 8 : 실패 (0.03ms, 10.1MB)
테스트 9 : 실패 (0.03ms, 10.1MB)
테스트 10 : 실패 (0.03ms, 10.1MB)
테스트 11 : 실패 (0.03ms, 10.3MB)
테스트 12 : 실패 (0.03ms, 10.1MB)
테스트 13 : 실패 (0.03ms, 10.1MB)
테스트 14 : 실패 (0.06ms, 10.1MB)
테스트 15 : 실패 (0.06ms, 10.3MB)
테스트 16 : 실패 (0.06ms, 10.1MB)
```

문제 풀면서 삽질했던거 공유

1. 짝수 길이의 팰린드럼도 된다 (abba)
2. 홀수 길이일 때 가운데 문자가 좌우 대칭에 쓰인 문자와 같아도 된다 (aba, aaa = 길이 3인 펠린드럼)

[프로그래머스] 2차 시도 - 투포인터

문자열을 0부터 끝까지 탐색 합니다.

1. 현재 문자열의 위치를 중심으로 좌우 비교를 합니다. 같은 경우 같은 문자열의 최대 길이를 저장 합니다.(홀수개)
2. 현재 문자열의 오른쪽에 있는 문자와 비교를 합니다. 같은 경우 두 개의 문자를 중심으로 좌우 비교를 합니다. 같은 문자열의 최대 길이를 저장합니다.(짝수개)
3. 탐색 중 좌우 다른 문자열이 나오면 탐색을 종료합니다.

- 다른 사람이 적은 힌트를 보고 투포인터 알고리즘이었다는 것을 떠올림
- 하지만 실패. “aaaaaaa”와 같은 경우 무조건 짝수 검사를 해버린다.

```
실패 (0.01ms, 10.4MB)
실패 (0.01ms, 10.2MB)
실패 (0.40ms, 10.3MB)
실패 (0.41ms, 10.4MB)
실패 (0.38ms, 10.2MB)
실패 (0.39ms, 10.2MB)
실패 (0.38ms, 10.3MB)
```

```python
for i in range(len(s) - 1):
    if s[i] == s[i + 1]:
        left = i - 1
        right = i + 2
        count = 2
    else:
        left = i - 1
        right = i + 1
        count = 1
    while left >= 0 and right < len(s):
        if s[left] == s[right]:
            count += 2
        left -= 1
        right += 1
    answer = max(answer, count)
```

[프로그래머스] 3차 시도 - 투포인터

- 다른 사람의 풀이를 참고.
- 홀짝 경우를 나누지 말고 그냥 모두다 구해야 함.

```
int isPalindrome(string s, int left, int right) {
    while(left >= 0 && right < s.size()) {
        if (s[left] != s[right]) break;
        left--;
        right++;
    }
    return right - left - 1;
}

int solution(string s)
{
    int answer=0;
    for (int i = 0; i < s.length(); i++) {
        int odd = isPalindrome(s, i, i);
        int even = isPalindrome(s, i - 1, i);
        int is_max = max(odd, even);
        answer = max(answer, even);
    }
    return answer;
}
```

[프로그래머스] 3차 시도 - 투포인터

- 홀짝 경우를 나누지 말고 그냥 모두다 구해야 함.
- 그리고 count += 2 형식이 아니라 포인터의 위치를 활용해서 개수를 센다.
- Right – left – 1

팰린드롬 개수가 짝수인 경우

팰린드롬 개수가 홀수인 경우

```
for i in range(len(s)):
    left = i
    right = i + 1
    even_case = 0
    while left >= 0 and right < len(s):
        if s[left] != s[right]:
            break
        left -= 1
        right += 1
    even_case = right - left - 1
    left = i
    right = i
    odd_case = 0
    while left >= 0 and right < len(s):
        if s[left] != s[right]:
            break
        left -= 1
        right += 1
    odd_case = right - left - 1
    current_max = max(odd_case, even_case)
    answer = max(answer, current_max)
```

Algorithm – Complete search

[프로그래머스] 자물쇠와 열쇠

- 배열이 주어짐
- Key의 경우 열쇠가 있는 곳에 값이 1로 지정 (나머지는 0으로)
- Lock의 경우 구멍이 있는 곳에 값이 0으로 지정 (나머지는 1로)
- 열쇠는 90도, 180도, 270도 회전이 가능하고, 상하좌우로 자유롭게 이동할 수 있음

입출력 예

key	lock	result
[[0, 0, 0], [1, 0, 0], [0, 1, 1]]	[[1, 1, 1], [1, 1, 0], [1, 0, 1]]	true

[프로그래머스] 1차 시도 – brute force

- 열쇠가 위치할 수 있는 경우의 수가 너무 많음
- 하지만 제한사항을 보니 M과 N의 제한 사항이 충분히 작음
- For문을 돌려서 Brute Force로 하나하나씩 대입해서 넣어보면 될 것 같음

제한사항

- key는 M x M(3 ≤ M ≤ 20, M은 자연수)크기 2차원 배열입니다.
- lock은 N x N(3 ≤ N ≤ 20, N은 자연수)크기 2차원 배열입니다.
- M은 항상 N 이하입니다.
- key와 lock의 원소는 0 또는 1로 이루어져 있습니다.
  - 0은 홈 부분, 1은 돌기 부분을 나타냅니다.

[프로그래머스] 1차 시도 – brute force

```python
key_cor = []
key_rotate = [[],[],[],[]]
lock_cor = []

for i in range(len(key)):
    for j in range(len(key[0])):
        if key[i][j] == 1:
            key_cor.append((i, j))

for i in range(len(lock)):
    for j in range(len(lock[0])):
        if lock[i][j] == 0:
            lock_cor.append((i, j))
count_lock = len(lock_cor)
```

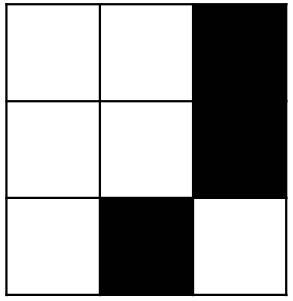
```python
for row, col in key_cor:
    key_rotate[0].append((row, col))
    key_rotate[1].append((col, len(key_cor) - 1 - row))
    key_rotate[2].append((len(key_cor) - 1 - row, len(key_cor) - 1 - col))
    key_rotate[3].append((len(key_cor) - 1 - col, row))

for arr in key_rotate:
    count = 0
    for row, col in arr:
        if (row, col) in lock_cor:
            count += 1
    if count == count_lock:
        return True

return False
```

[프로그래머스] 1차 시도 실패

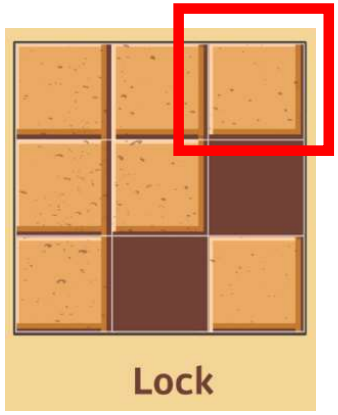
- 통과하면 안 되는 경우에서 통과를 해버림. 회전만 시키고, 상하좌우 이동은 하지도 않았는데, 열쇠가 맞아떨어졌다고 결과가 나온 것.
- 초기화 때, 비어 있는 lock 좌표만 담았기 때문에 key와 lock이 겹쳐서 열쇠가 들어가지 않는 경우를 고려하지 않았음

```
for i in range(len(lock)):
    for j in range(len(lock[0])):
        if lock[i][j] == 0:
            lock_cor.append((i, j))
```

```
for arr in key_rotate:
    count = 0
    for row, col in arr:
        if (row, col) in lock_cor:
            count += 1
    if count == count_lock:
        return True
```



Key

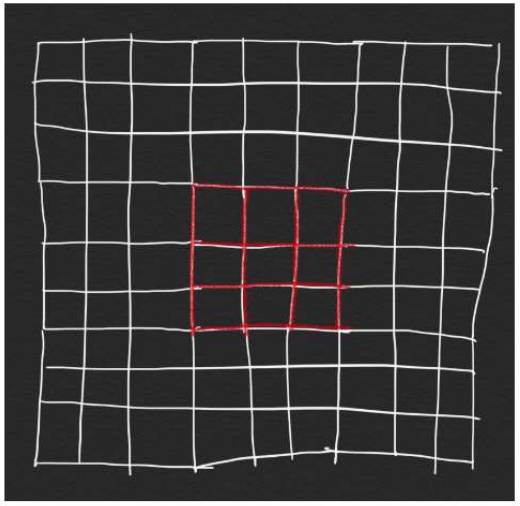


Lock

```
[[0, 0, 0], [1, 0, 0], [0, 1, 1]], [[1, 1, 1], [1, 1, 0], [1, 0, 1]]
true
테스트를 통과하였습니다.
```

[프로그래머스] 다른 풀이 참고

- 기본적으로 주어진 배열을 확장해서 푸는 문제가 대다수
- Key의 값과 lock의 값을 더해서 전체 배열의 값을 탐색해서 1이 나오면 True. 값이 하나라도 1이 아닌 0이나 2가 나온다면 False.



```python
def solution(key, lock):
    M = len(key)
    N = len(lock)
    # 3배 더 큰 자물쇠 생성
    new_lock = [[1] * N * 3 for _ in range(N * 3)]
    # 기존 자물쇠를 새로운 자물쇠 가운데에 위치
    for ix in range(N):
        for iy in range(N):
            new_lock[ix + N][iy + N] = lock[ix][iy]
```

```python
def check_match(lock, key):
    for i in range(len(lock)):
        for j in range(len(lock)):
            if lock[i][j] + key[i][j] != 1:
                return False
    return True
```

[프로그래머스] 다른 풀이 참고

- 새로운 배열 공간 board를 만들고 자물쇠를 중앙에 둔다.
- 열쇠가 상하좌우로 움직일 수 있는 충분한 배열 공간을 생성한다. (가로세로: 2M + N 또는 3N) (N은 자물쇠 크기)
- Rotate를 먼저 해서 열쇠 방향을 먼저 결정
- 반복문으로 상하좌우 움직였을 경우를 모두 탐색 (Brute force)

```
def solution(key, lock):
    M, N = len(key), len(lock)
    # 자물쇠를 중심으로 열쇠가 상하좌우 움직일 수 있는 공간 만들기
    board = [[0] * (M * 2 + N) for _ in range(M * 2 + N)]

    # 자물쇠를 중앙에 배치
    for i in range(N):
        for j in range(N):
            board[M+i][M+j] = lock[i][j]
```

```
rotated_key = key
# 회전하여 넣어보기 (4번 루프)
for _ in range(4):
    rotated_key = rotate90(rotated_key)
    for row in range(1, M + N):
        for col in range(1, M + N):
            attach(row, col, M, rotated_key, board)
            if (check(board, M, N)):
                return True
            detach(row, col, M, rotated_key, board)
return False
```

[프로그래머스] 다른 풀이 참고

- Attach: 열쇠를 넣었을 때. 전체 배열 board에 열쇠 값만큼 더한다.
- Detach: 열쇠를 뺐을 때. Board에서 열쇠 값만큼 뺀다.
- Board에서 자물쇠 영역만 검사한다. 해당 영역 내 값이 1이 아니면 False, 값이 전부 1이면 True
- 내장함수 zip()과 리스트 조작으로 배열 회전

```python
def attach(row, col, M, key, board):
    for i in range(M):
        for j in range(M):
            board[row + i][col + j] += key[i][j]

def detach(row, col, M, key, board):
    for i in range(M):
        for j in range(M):
            board[row + i][col + j] -= key[i][j]

def check(board, M, N):
    for i in range(N):
        for j in range(N):
            if board[M + i][M + j] != 1:
                return False
    return True

def rotate90(arr):
    return list(zip(*arr[::-1]))
```

서브넷, CIDR, private IP

Public IP vs private IP

- Public IP: ISP가 제공하는 IP. 전 세계에서 유일한 IP 주소를 가짐. 다른 PC로부터 접근할 수 있음
- Private IP: 일반 가정이나 회사 내에서 할당된 네트워크 주소. IPv4 주소 부족으로 서브네팅된 주소. 라우터에 의해 할당되며, 외부에서 바로 접근할 수 없고, 라우터를 거쳐야 접근할 수 있다.

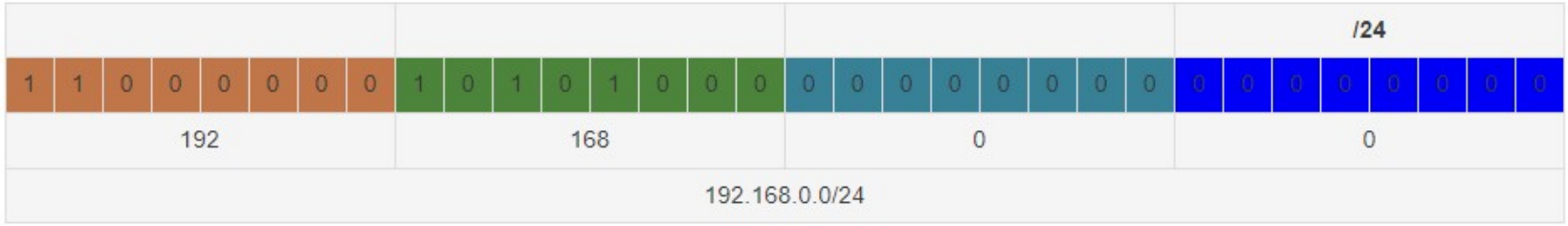
CIDR(Classless Inter-Domain Routing)

- AWS, GCP, Azure 등 퍼블릭 클라우드 등을 사용할 때는 네트워크 설정에서 VPC 및 Subnet 을 생성하여 네트워크를 구성하게 된다.
- CIDR는 그냥 서브넷을 표기하는 방법 중 하나. 단 한 줄로 네트워크의 범위를 알 수 있게 하는 방법이다.
- IP 주소를 Class A,B,C 등과 같이 규격화된 구분없이 좌측부터 비트 단위로 넷 마스크를 취하는 방식
- N은 0~32범위를 가진다.
- N 비트 수만큼 주소가 같다는 것을 의미한다.

A.B.C.D/N

CIDR(Classless Inter-Domain Routing)

- IP 주소는 4개의 옥텟을 가진다.
- 아래 CIDR 표기 방식은 좌측에서 부터 24비트는 동일하고 나머지 8 비트가 가변적인 것을 의미함.
- 그러므로 이 표기 방식은 192.168.0.0 ~ 192.168.0.255를 의미함.

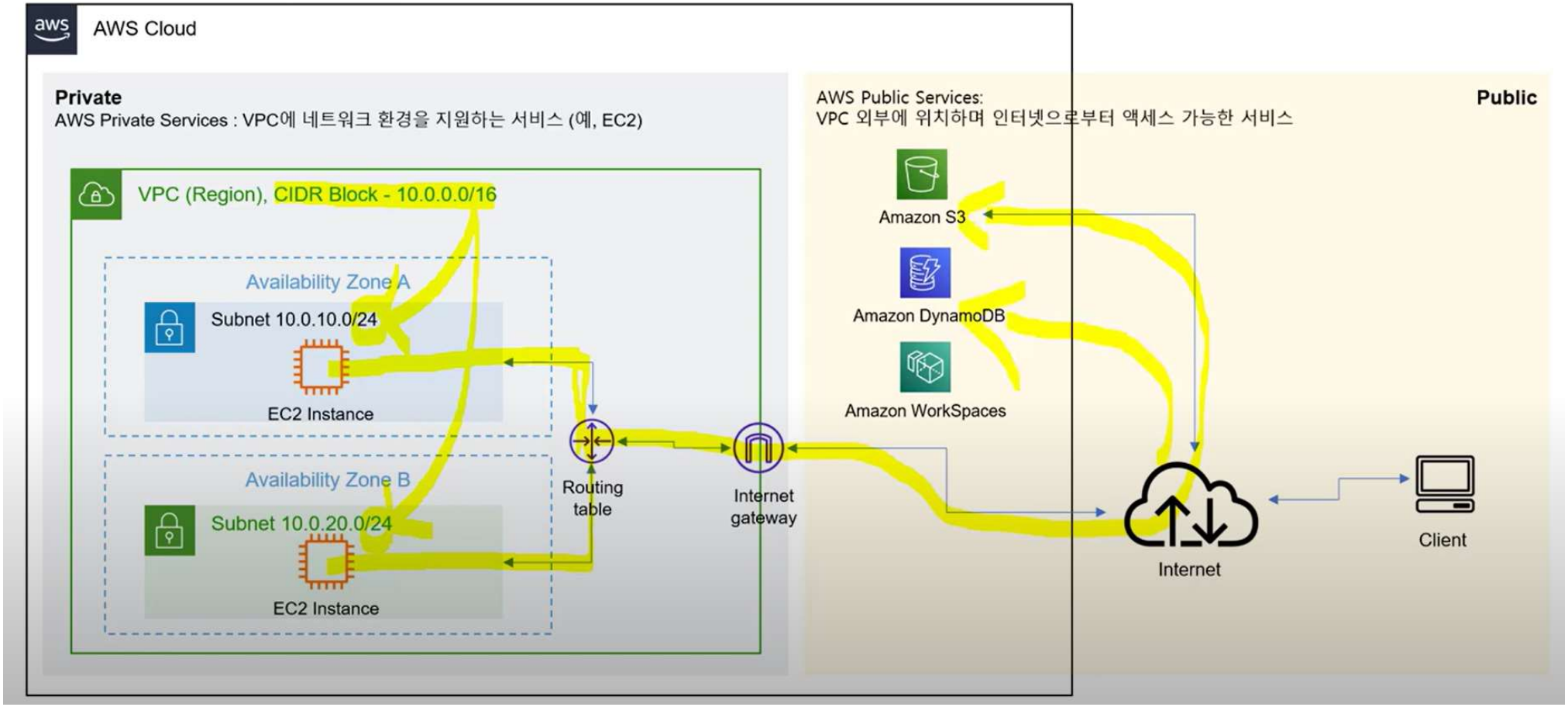


AWS – VPC, private link

AWS VPC 기본 개념

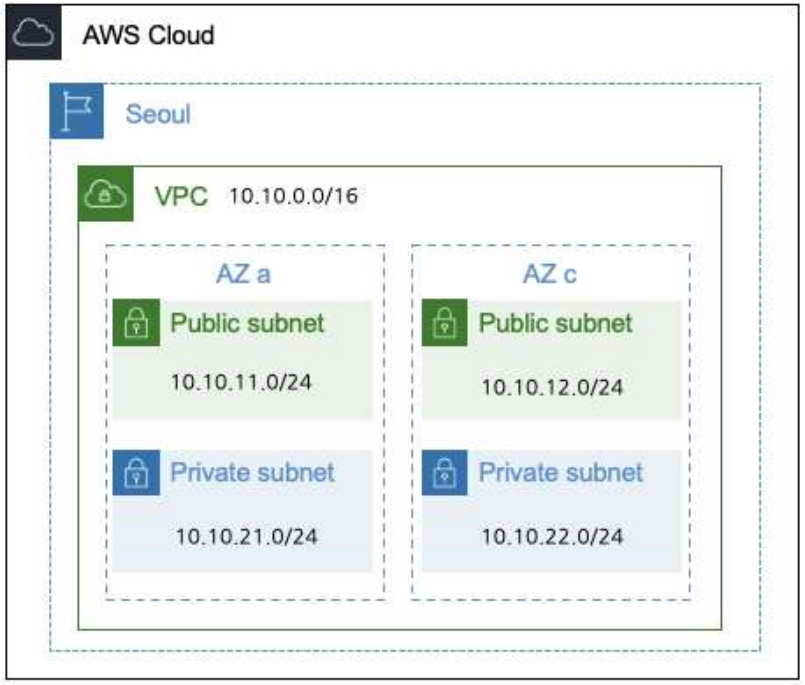
- VPC: 사용자가 정의한 가상 네트워크
- VPC는 private 네트워크이므로 private IP를 사용한다.
- AWS Cloud 안에 VPC를 생성할 수 있음
- VPC 안에 가용영역(AZ)를 생성할 수 있고, 이 가용영역에는 네트워크의 서브넷이 할당된다.
- 하나의 서브넷 안에 원하는 컴퓨터를 둘 수 있음 (EC2, lambda 등)
- 서로 다른 서브넷은 routing table을 통해 통신할 수 있다.
- VPC 내부 요소들은 internet gateway를 통해 외부 인터넷과 통신할 수 있다.

AWS VPC


AWS Cloud
Private
AWS Private Services : VPC에 네트워크 환경을 지원하는 서비스 (예, EC2)
VPC (Region), CIDR Block - 10.0.0.0/16
Availability Zone A
Subnet 10.0.10.0/24
EC2 Instance
Availability Zone B
Subnet 10.0.20.0/24
EC2 Instance
Routing table
Internet gateway
AWS Public Services:
VPC 외부에 위치하며 인터넷으로부터 액세스 가능한 서비스
Public
Amazon S3
Amazon DynamoDB
Amazon WorkSpaces
Internet
Client

리전 – VPC – 가용영역 - 서브넷

- 하나의 리전 안에 여러 VPC를 둘 수 있고
- 하나의 VPC 안에 여러 가용영역을 둘 수 있고
- 하나의 가용영역 안에 여러 서브넷을 둘 수 있고,
- 하나의 서브넷 안에 여러 서버(EC2)를 둘 수 있다.



AWS 리전과 AZ

- 리전: 서비스를 제공하고 있는 거점 국가
- 가용 영역: 리전 내에는 여러 개의 가용영역이 존재한다. 각 가용영역끼리는 전용선으로 연결되어 있어서 인터넷에 비해 통신이 빠르다.
- 서브넷을 만들 때 포인트는 동일한 역할을 하는 서브넷을 여러 AZ에 복제해서 넣는 것이다. 이렇게 하면 특정 AZ에 장애가 발생해도 다른 AZ가 cover칠 수 있다.
- 다중 AZ <- 이것이 AWS 설계의 가장 기본

AWS subnet

- 서브넷: EC2 등 서버를 구동하기 위해서 VPC 내부에 만든 주소 범위. VPC에 설정한 CIDR 블록에 다시 더 작은 CIDR 블록을 할당할 수 있다.
- 각 서브넷에는 하나의 라우팅 테이블만 설정할 수 있다.
- 각 서브넷에는 하나의 ACL만 설정할 수 있다. 하나의 서브넷에 작성할 수 있는 서브넷은 최대 200개.
- 주의점: 필요 이상으로 서브넷을 분할할 경우 주소의 낭비로 이어진다.
- Public subnet과 private subnet이 있는데, 이것은 해당 서브넷이 무엇을 경우해서 연결하는가에 따라 결정된다.

AWS routing table

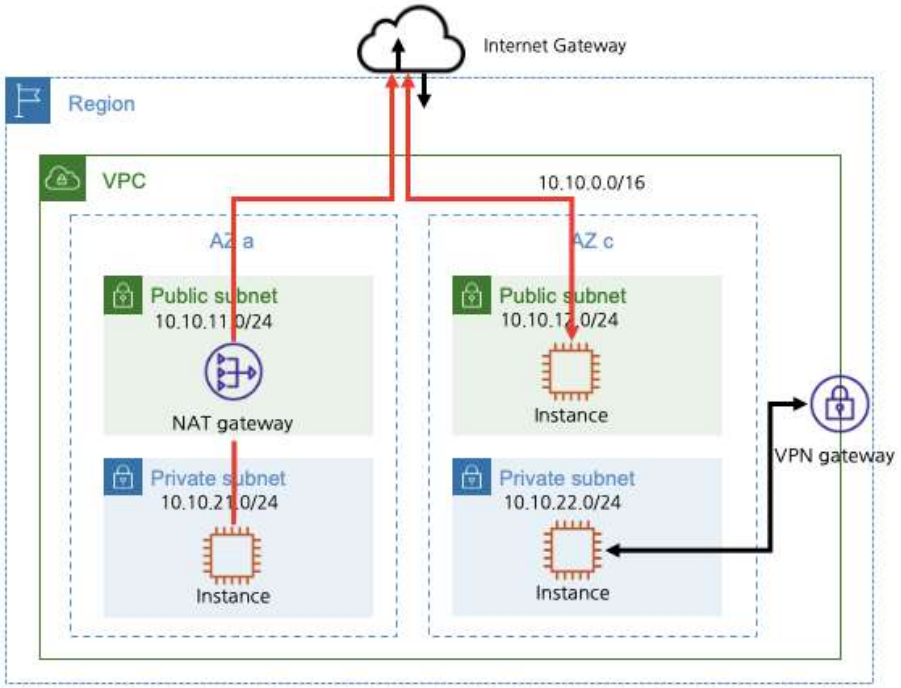
- 서로 다른 서브넷이 이를 통해 통신할 수 있다.
- 하나의 라우팅 테이블은 복수의 서브넷에 공유되어 사용할 수 있지만, 하나의 서브넷은 복수의 라우팅 테이블을 적용할 수 없다.
- VPC에는 메인 라우팅 테이블이 있고, 서브넷 생성할 때 별도의 라우팅 테이블을 설정하지 않는 경우, 메인 라우팅 테이블로 지정된다.

AWS network ACL

- 서브넷에 대한 트래픽을 제어하는데 사용한다.
- 일종의 방화벽 장비
- 인바운드(외부에서 VPC로 접근), 아웃바운드(VPC에서 밖으로) 양방향으로 트래픽을 제한할 수 있다.

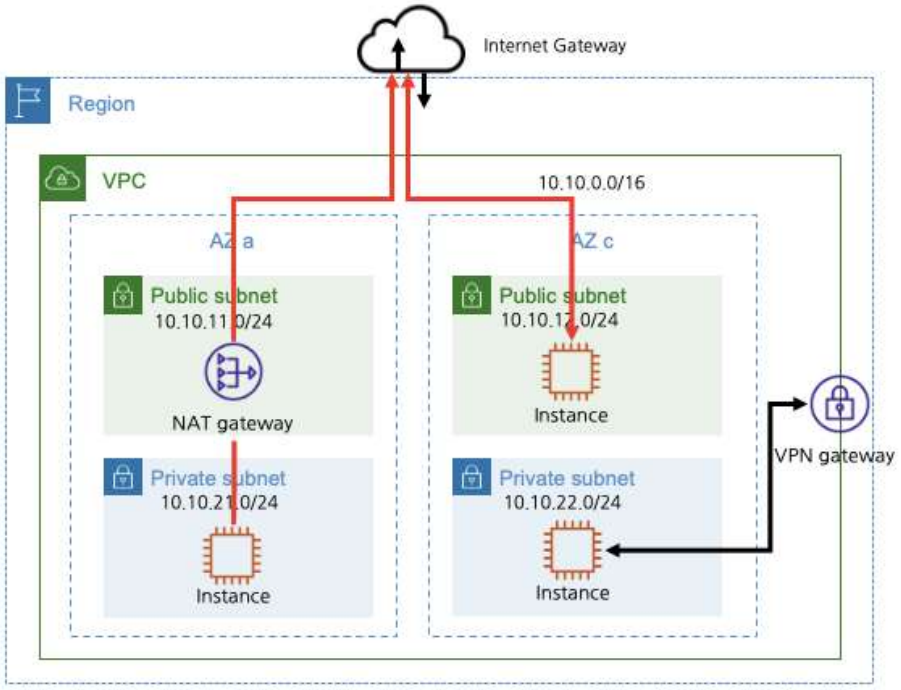
AWS 인터넷 게이트웨이

- 인터넷 게이트웨이: VPC와 internet을 연결하는 출입구. 각 VPC에 하나만 설정할 수 있다.
- 라우팅 테이블이 게이트웨이와 연결되면 서브넷 내부 요소들은 인터넷과 통신할 수 있게 된다.
- 인터넷 게이트웨이와 바로 연결되면 public subnet.
- 인터넷 게이트웨이와 연결되지 않으면 private subnet. 하지만 NAT 게이트웨이를 경유해서 인터넷 게이트웨이와 연결될 수도 있다.



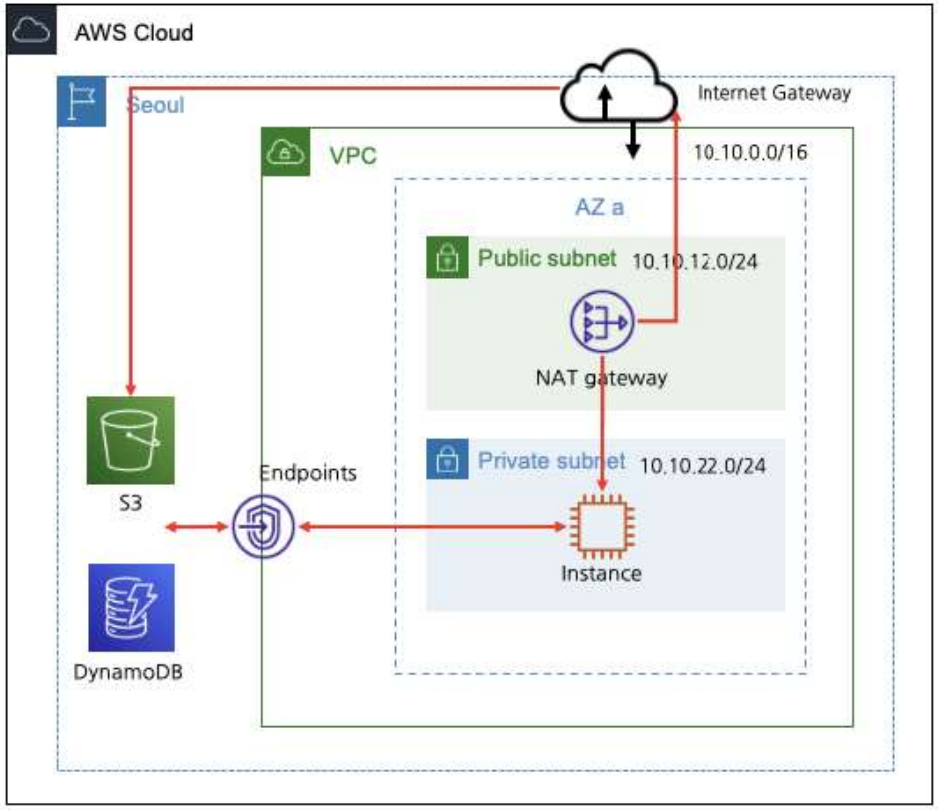
AWS 가상 프라이빗 게이트웨이

- 가상 프라이빗 게이트웨이(VPN gateway): VPC가 VPN이나 Direct Connect에 접속하기 위한 게이트웨이.
- VPNGateway도 각 VPC에 하나만 설정할 수 있다.



VPC Endpoint

- VPC Endpoint는 VPC 내부 요소가 인터넷을 경유하지 않고 특정 AWS 지원 서비스에 접근하는 방법이다.
- VPC link라고 하기도 함
- 세가지 타입 엔드포인트가 존재
- Interface endpoint
- Gateway endpoint
- Gateway load balancer endpoint

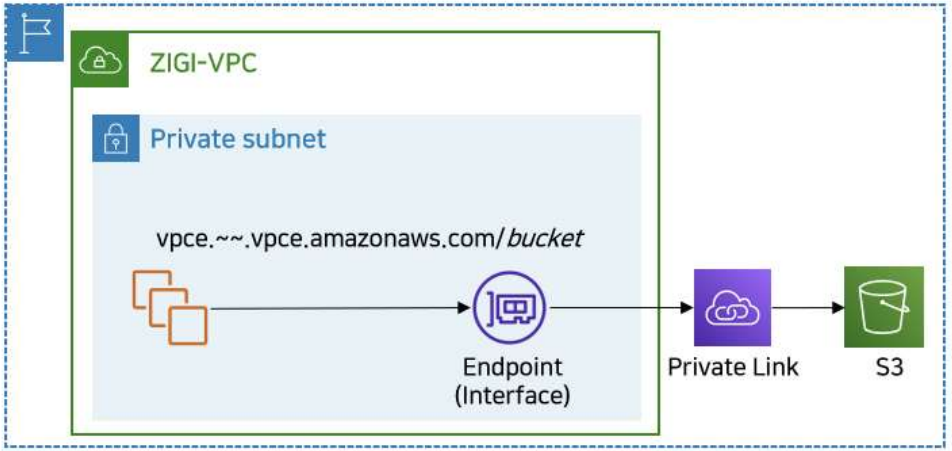


PrivateLink

- Interface Endpoint는 AWS PrivateLink에 의해 구동된다.

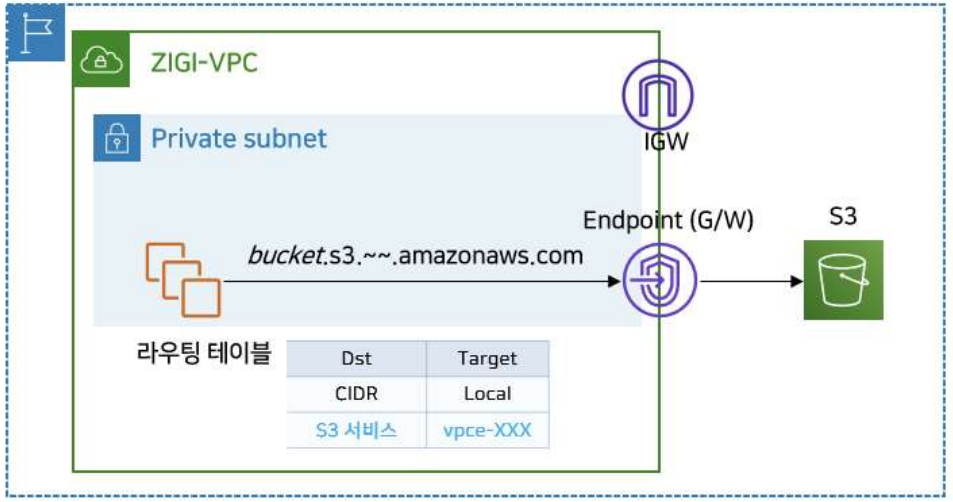
Interface Endpoint

- Elastic network interface(ENI)이다. security group에 붙여서 사용한다.
- PrivateLink를 통해 AWS 서비스 또는 AWS에서 지원하는 마켓플레이스에 연결할 수 있다.
- 요금이 부과된다.
- 서브넷 내에서 생성되어 사용된다.



Gateway Endpoint

- VPC를 DynamoDB나 S3에 연결할 때 사용한다.
- Private Link를 사용하지 않는다.
- 사용에 요금이 부과되지 않는다.
- VPC 내부에 생성되어 사용된다.
- 해당 endpoint를 사용할 서브넷을 지정하여 사용할 수 있다.
- 서브넷에 연결만 하면 서브넷에 있는 모든 요소를 다 연결한다. gateway endpoint 접근을 위한 별도의 인스턴스를 생성할 필요 없다.



Interface Endpoint vs Gateway Endpoint

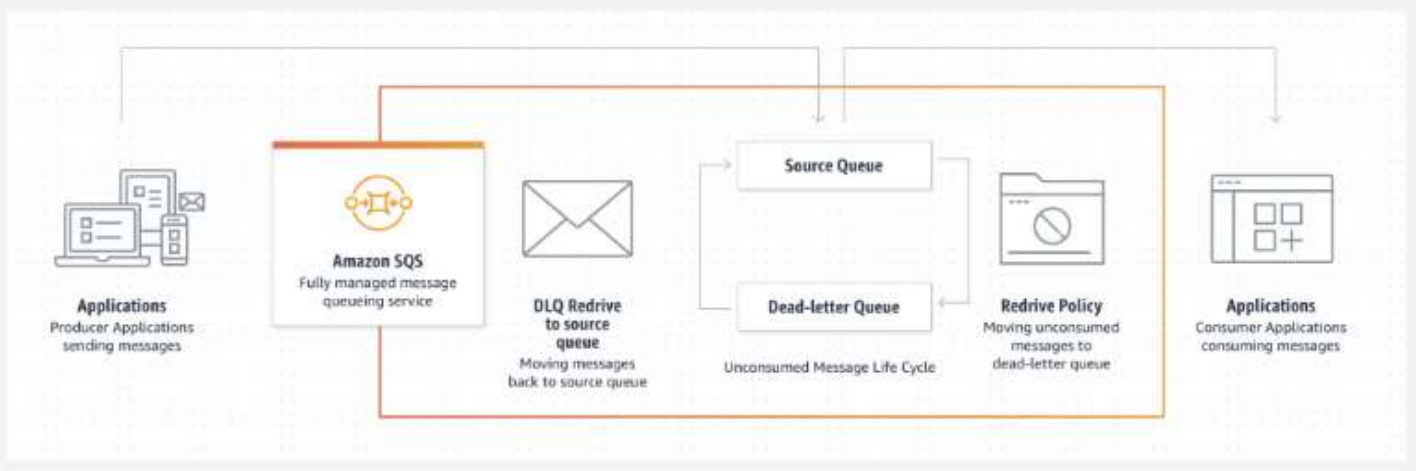
- Gateway endpoint는 VPC 안에서 작동되고
- Interface endpoint는 보안 그룹에 붙어서 작동된다.

AWS - DLQ

Dead letter Queue 역할

- 메시지들은 가끔 유실될 수도 있다. 예상치 못한 에러 상황 있을 수 있음. Consumer의 하드웨어 에러 때문에 메시지 페이로드가 망가질 수도 있음
- AWS에서 특정 수만큼의 메시지가 Consumer에서 처리되는 데 실패하면, 해당 메시지는 dead letter queue로 넘어간다.
- DLQ에 쌓인 메시지들을 보고 어플리케이션 버그를 찾아낼 수 있음

AWS DLQ



- Amazon SQS에 있는 하나의 기능
- Source Queue에서 일정 횟수 전송에 실패한 메시지들을 Dead letter queue로 보내준다.

AWS DLQ 동작 방식, 사용법

- maxReceiveCount를 설정할 수 있다. 이 수는 SQS에서 받으려고 retry할 수 있게 허용해주는 횟수를 의미한다. 이 수를 넘어가면 메시지는 Dead letter queue로 넘어간다.
- 버그를 찾아내서 수정한 이후로 AWS console에서 redrive해서 해당 메시지들을 다시 source queue로 집어 넣을 수 있다.
- AWS SDK나 console을 이용해서 dead letter queue를 지정할 수 있다. DLQ 기능을 사용하기 위해 특정 큐를 반드시 dead letter queue로 지정해줘야 한다.
- ApproximateAgeOfOldestMessage 메트릭: 해당 큐에서 삭제되지 않고 가장 오래 남아있는 메시지의 존재 기간

큐 타입에 따라 메시지 실패 처리

- Standard queue

  - retention period가 끝나기 전까지는 계속 재시도를 한다. 하지만 이렇게 계속 재시도를 하게 놔두면 하드웨어에 무리를 주므로 적당한 maxReceiveCount를 설정해서 DLQ로 빠지게 한다.

- FIFO queue

  - 메시지들을 순서대로 정확히 한 번 처리하기 때문에 문제가 있는 메시지가 있는 그룹은 계속 사용 불가 상태. FIFO queue는 FIFO DLQ를 사용해야 하는데, 복구할 때 메시지 순서를 제대로 복구하는 message processing을 거쳐야 한다. 만약 복구 과정이 따로 없다면 FIFO의 경우 DLQ를 사용하지 말고 그냥 계속 재시도하도록 놔둬야 한다.

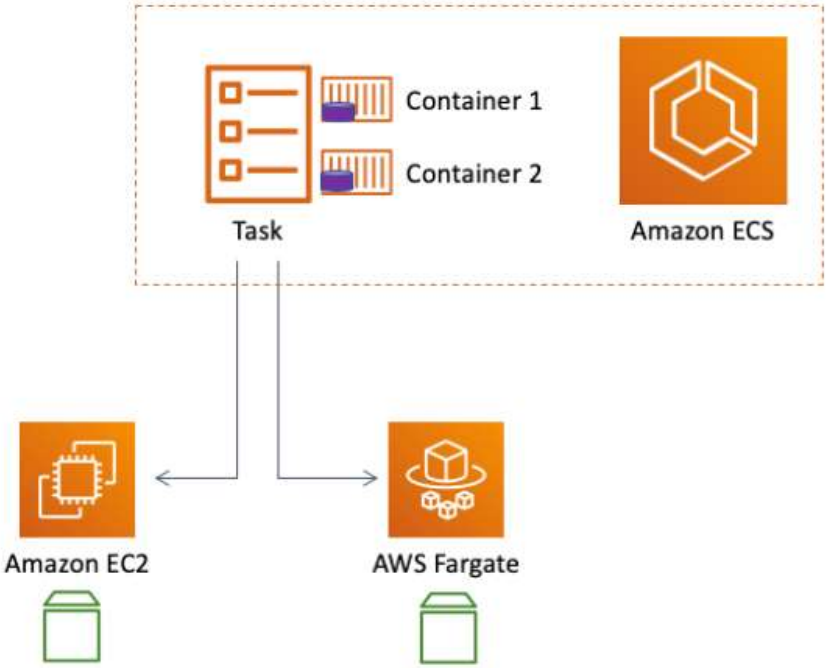
AWS DLQ 사용 시 주의사항

- 콘솔로 현재 SQS에서 진행 중인 메시지를 보면, 횟수가 카운트되어 maxReceiveCount가 올라간다. 일정 횟수가 되면, 해당 메시지는 Dead Letter Queue로 보내진다. 그러므로 함부로 큐의 메시지를 console 을 통해 보지 말 것.
- DLQ의 retention period 는 항상 소스 큐의 retention period 보다 길게 설정해야한다.
- NumberOfMessageSent 와 NumberOfMessagesReceived가 다를 수 있다. 수동으로 메시지 보내면, NumberOfMessageSent가 올라간지만, 처리 실패로 DLQ에 들어가면 해당 변수가 올라가지 않는다.

AWS – ECS

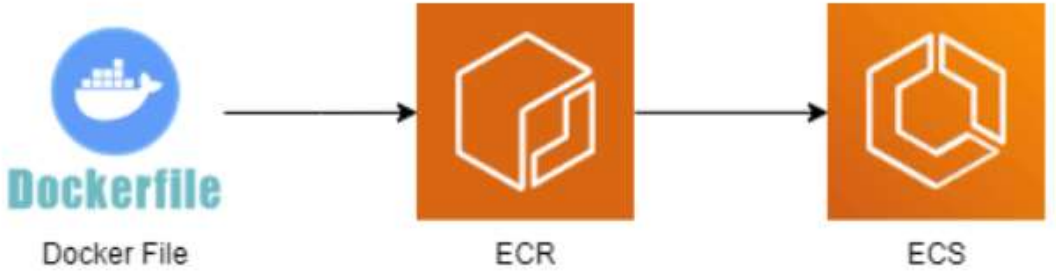
AWS ECS

- Elastic container service는 **컨테이너의 라이프사이클을 관리**하는 AWS의 컨테이너 오케스트레이션 서비스다.
- **컨테이너 라이프사이클**엔 'Container Start', 'Re-Schedule', 'Load Balancing' 등이 포함된다.
- 쉽게 말해서 도커 이미지 꺼내와서 Task 별로 Container 돌리고 그렇게 해서 EC2 또는 Fargate 생성해서 서버 운영함
- 비슷한 툴로는 kubernetes나 docker swam이 있음



AWS ECR

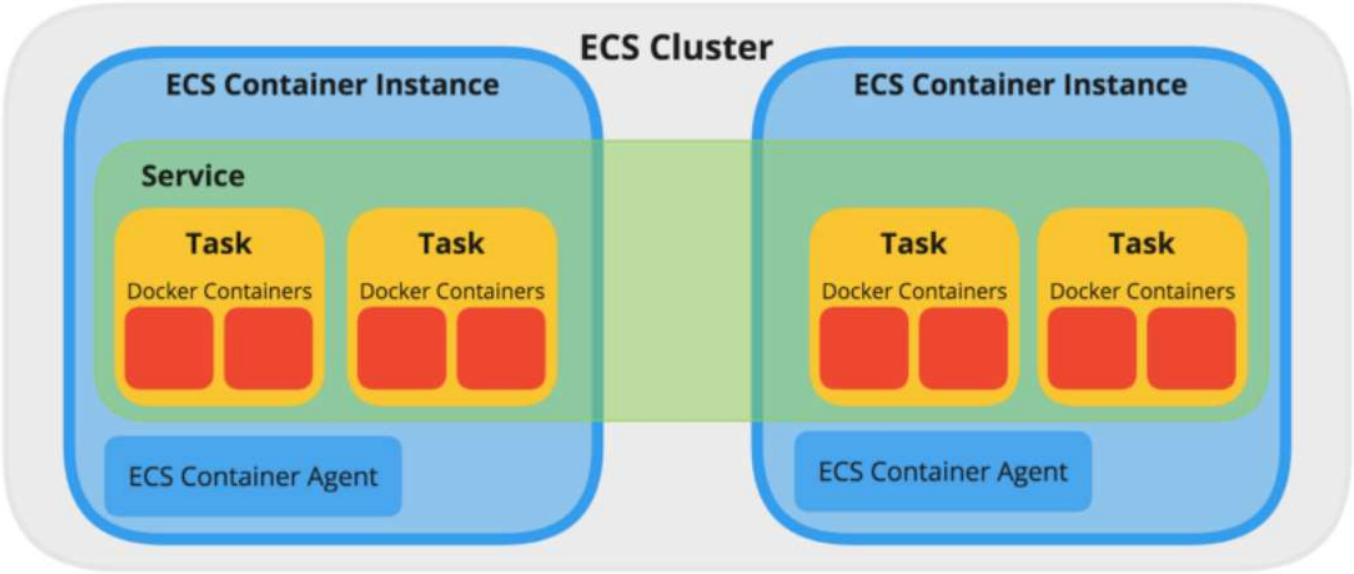
- ECS를 알기 전에 ECR을 먼저 알아야 한다.
- Elastic Container Registry는 도커 이미지를 저장하는 리포지토리.
- 여기에 도커 이미지를 저장하는 이유: 새로운 버전의 이미지가 올라오면 자동으로 클러스터에 넘어가게 설정할 수도 있는 등 다양한 기능을 누릴 수 있다.
- 또는 AWS IAM이랑 연계되어서 이미지를 push pull 할 수 있는 권한을 관리할 수도 있다.



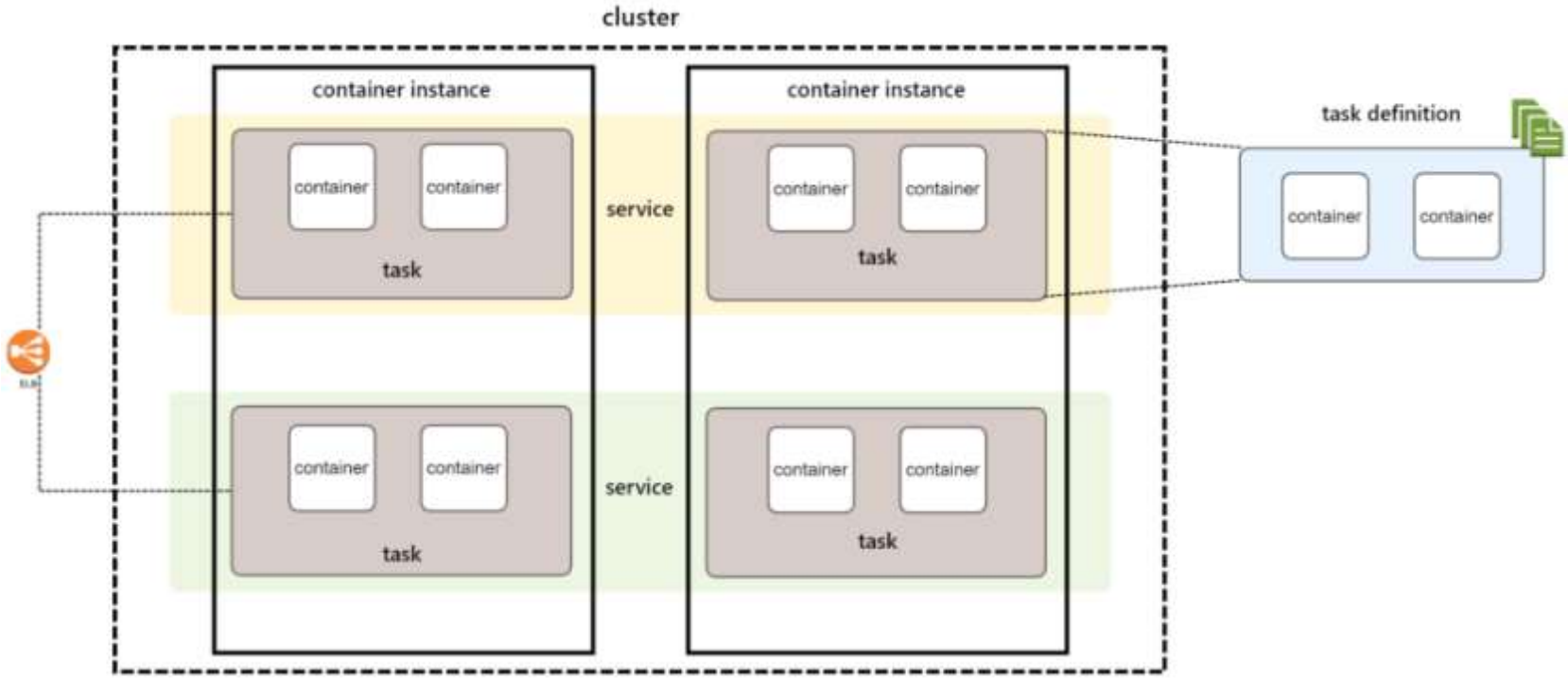
AWS ECS 구성요소

- Cluster: ECS 기본 단위. 컨테이너를 실행할 수 있는 가상공간. 논리적으로 묶인 그룹을 의미함. 보통 프로젝트 단위로 실행
- Task: 생성한 컨테이너를 구동하는 최소 단위. 하나의 task 내에 여러 컨테이너가 있을 수 있음
- Task definition: Task를 실행할 때, 컨테이너 네트워크 모드, Task 역할, 도커 이미지, CPU 메모리 제한 등 설정이 필요한데, 이런 설정을 매번 하는 게 아니라 하나의 집합 단위로 두고 필요할 때 사용하는데, 이 때 이 집합 단위가 Task definition이다.
- Service: cluster 는 두 가지 방식으로 Task를 실행할 수 있는데, 하나가 Task definition으로 직접 Task 실행, 다른 하나는 service 를 정의하여 여러 Task를 동시에 실행하는 것이다.

AWS ECS 구성요소 그림으로 정리



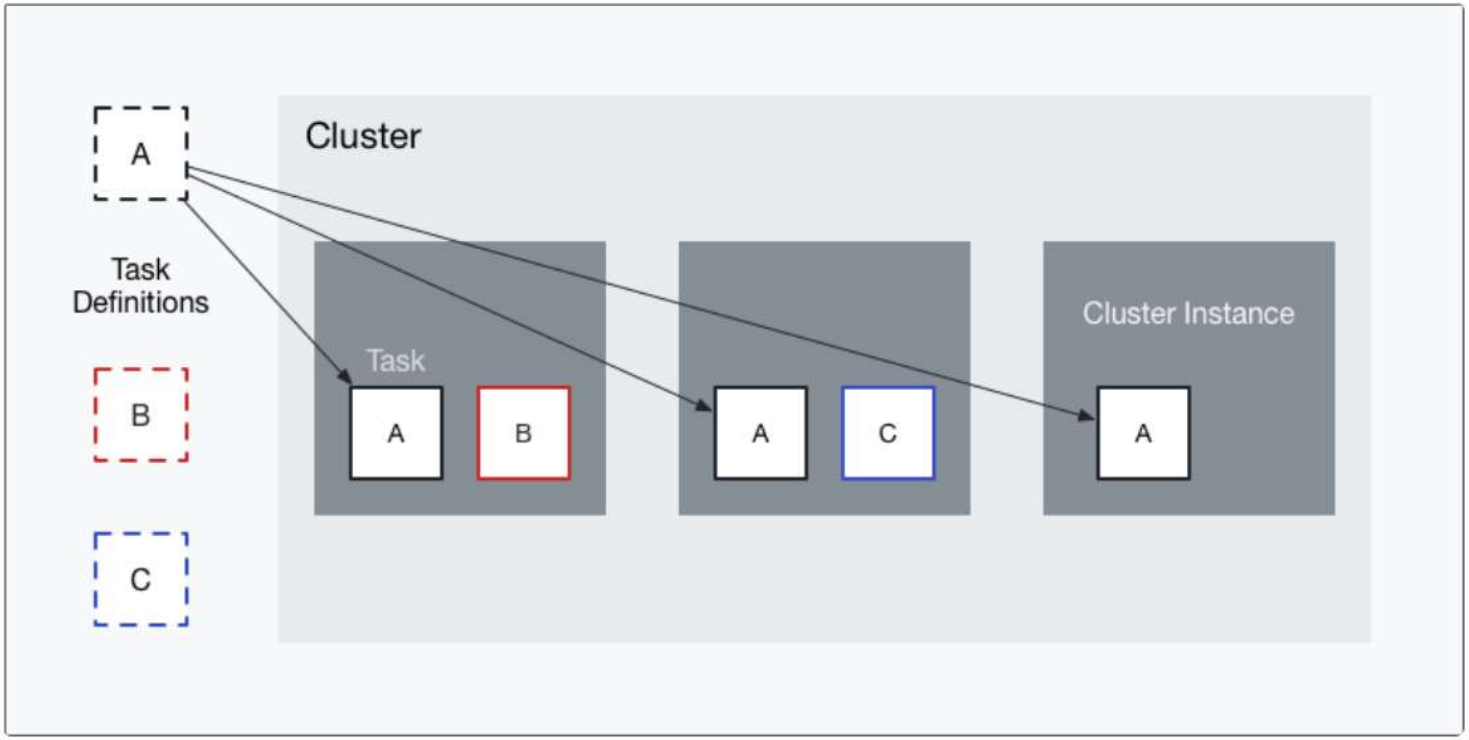
AWS ECS 구성요소 그림으로 정리



- ECS 최소 단위는 Task. 하나의 Task에 여러 Container로 구성됨

Task와 Task definition

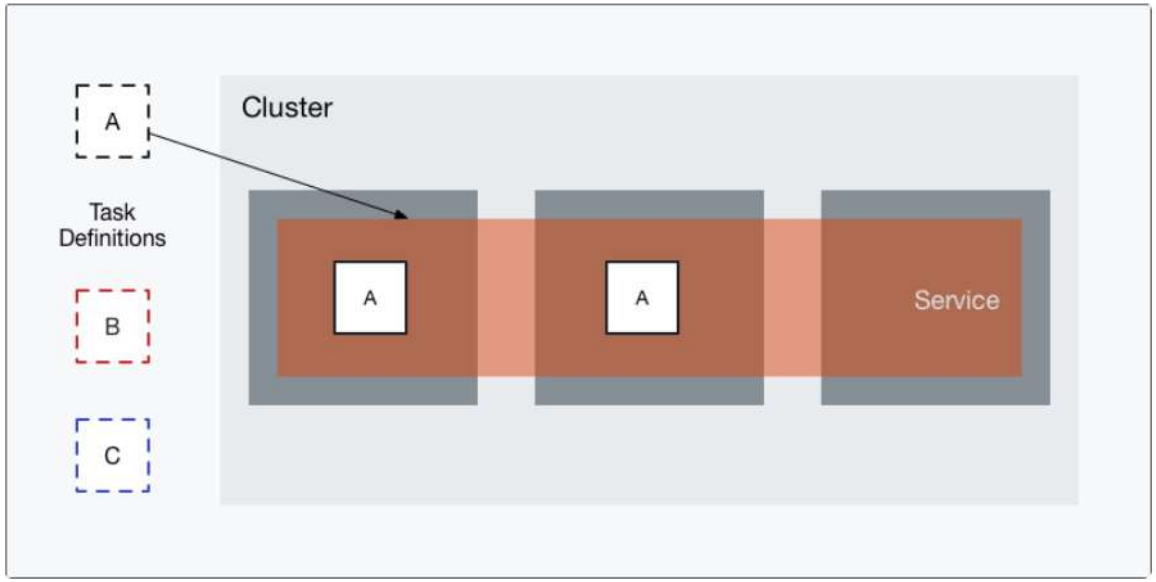
- Task는 Cluster에 종속적이지만, Task definition 은 Cluster에 종속적이지 않다.



Service의 사용법 및 종류

- 서비스를 정의하는 방법

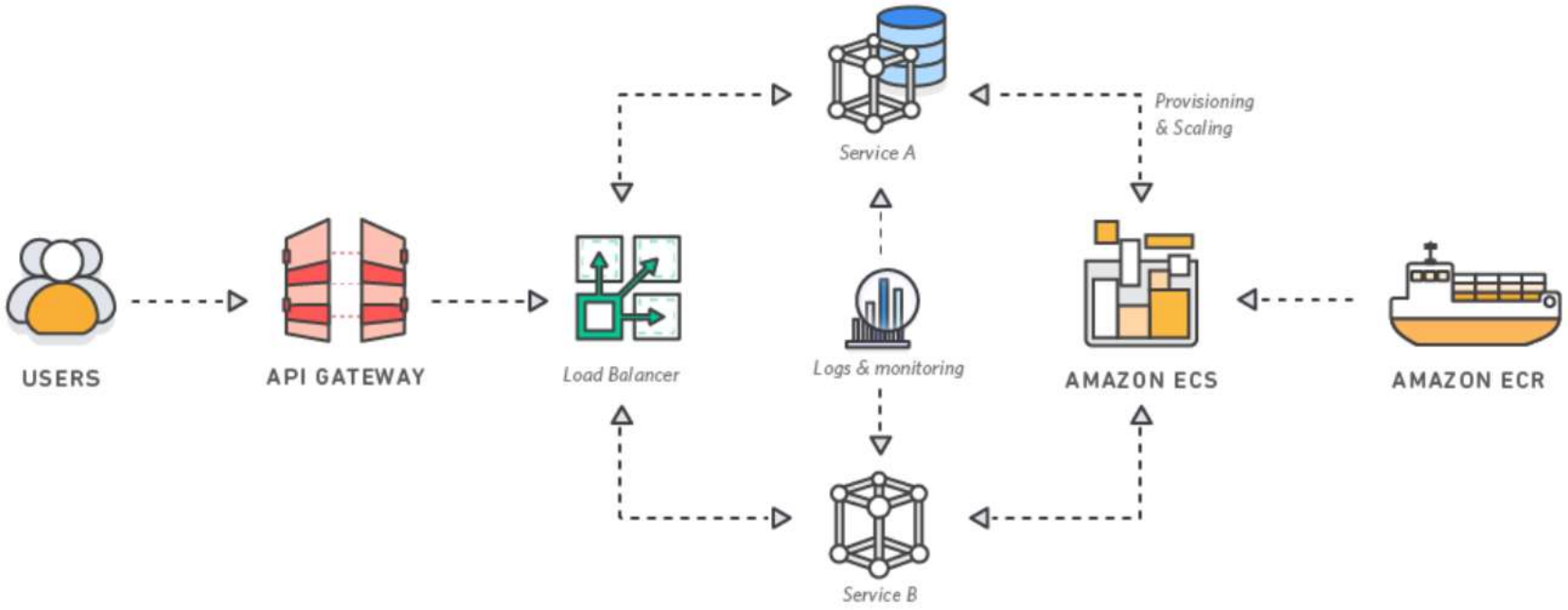
- Replica 타입: 실행하려는 Task 개수를 지정하면 서비스가 클러스터에서 이 개수만큼 Task가 실행되도록 관리해준다.
- Daemon 타입: 모든 컨테이너 인스턴스에서 관련 Task가 하나씩 실행된다.



ECS 동작 순서

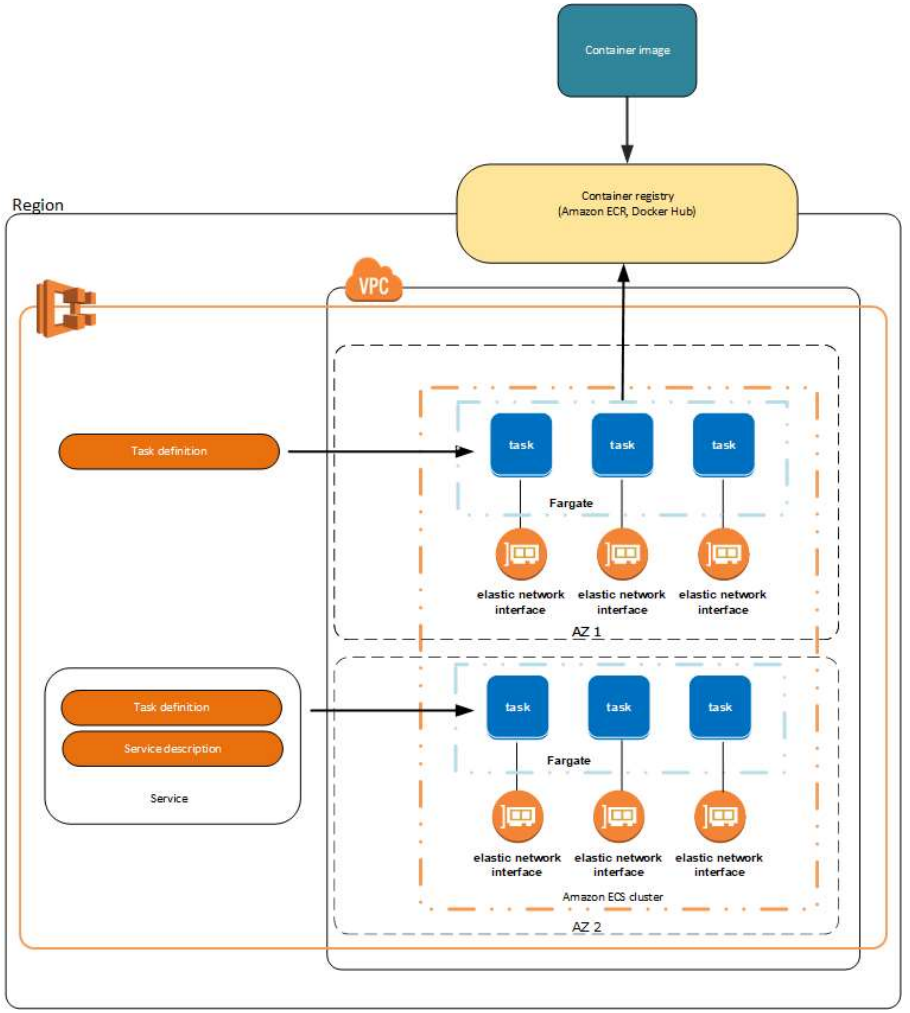
1. 컨테이너의 이미지를 저장소(ECR)에 커밋
2. task definition에서 사용할 이미지 및 시작 유형, 리소스 정의
3. cluster 생성
4. service가 task definition을 참고하여 task 생성
5. elb에 들어온 요청에 따라 오토 스케일링 및 로드 밸런싱

MSA에서 ECR, ECS를 사용한 보편적 구조



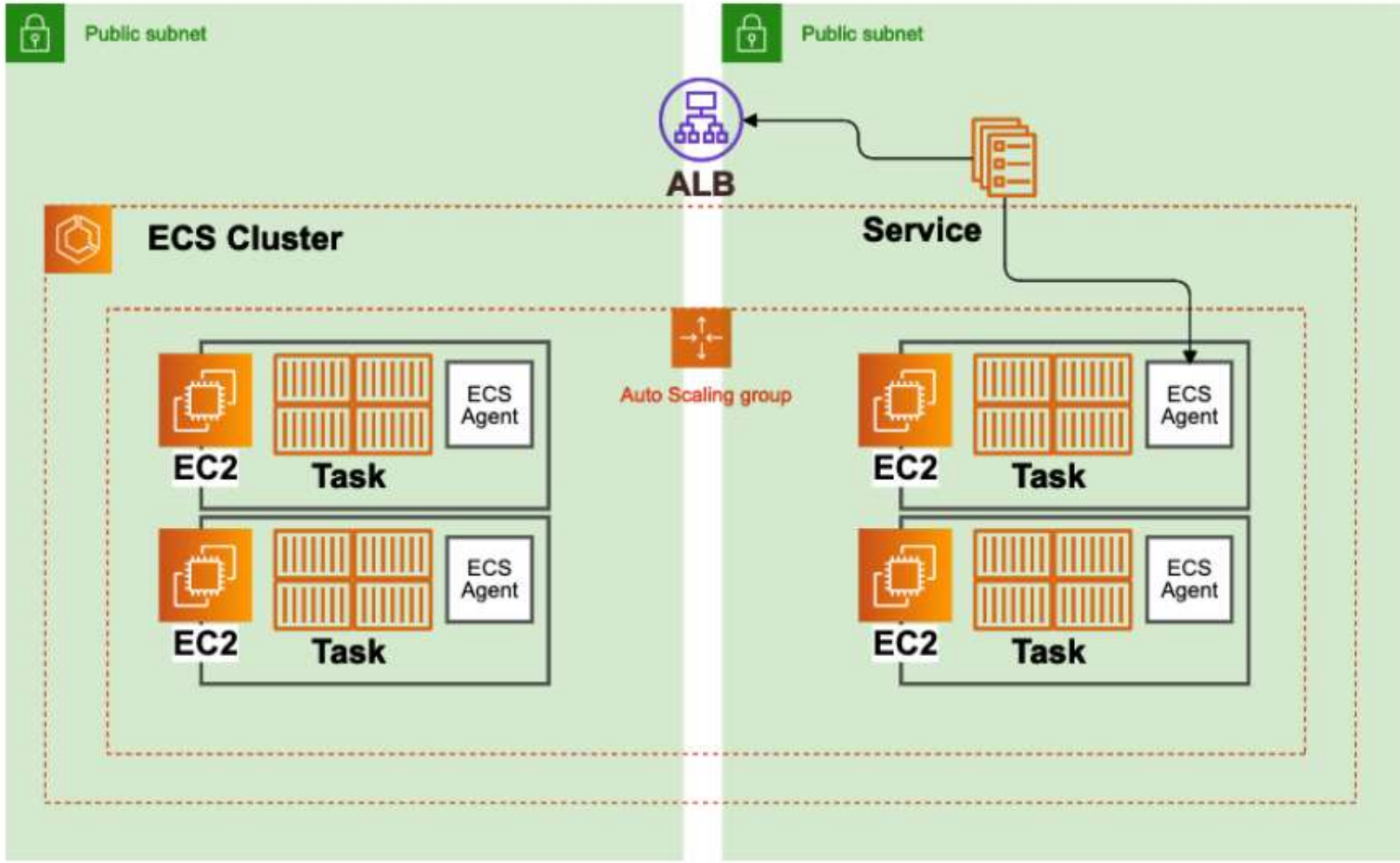
AWS VPC 와 ECS 위치

- ECS 설정은 외부에 위치해 있다.
- 외부에 위치한 ECS는 Task definition과 service description을 가지고 있다.
- VPC 내부에는 AZ가 있고 각 AZ에 Fargate나 EC2 등 서버 컴퓨터가 있다. 이 서버 컴퓨터 안에서 Task가 돌아간다.
- 여러 AZ 영역에 걸쳐서 ECS cluster가 형성될 수 있다.



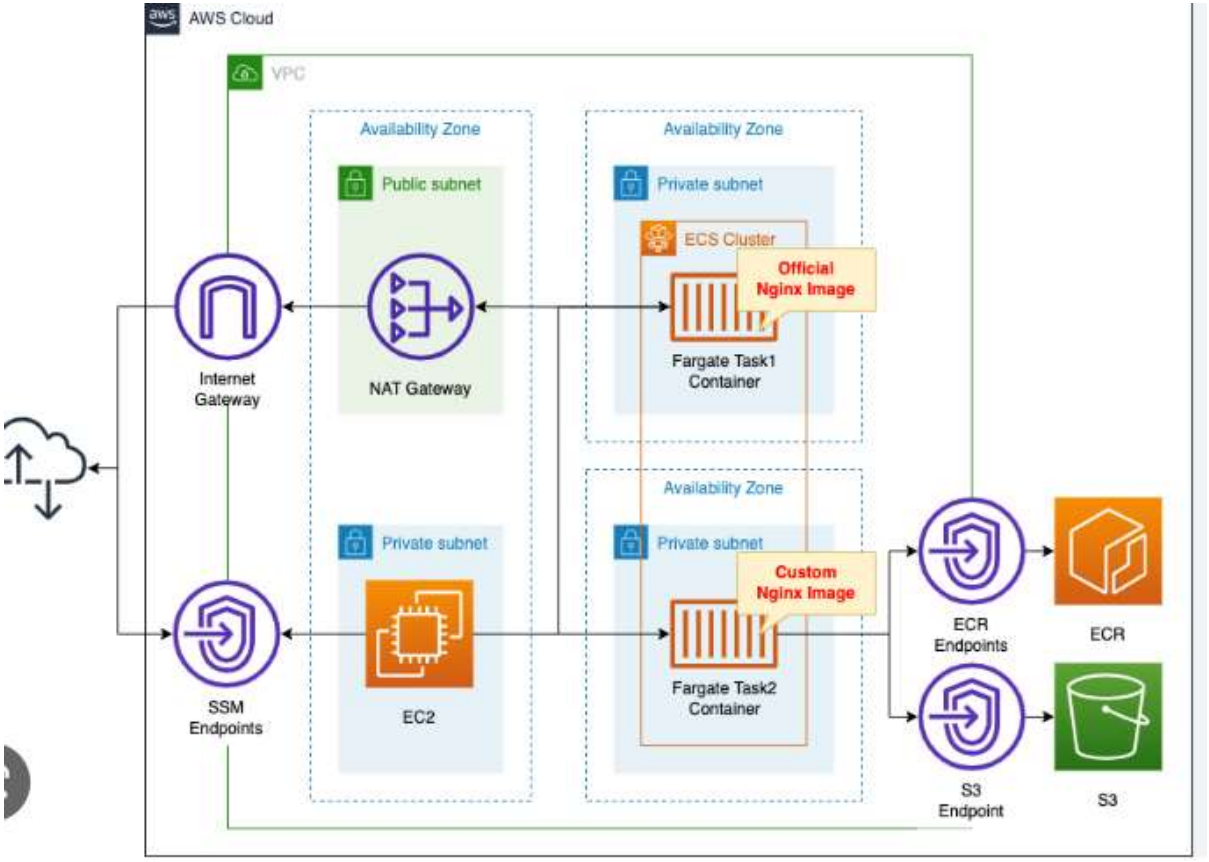
AWS 서브넷 내 ECS cluster

- VPC 내부에는 여러 서브넷이 있다.
- 각 서브넷 내부에 Task가 있고 이 Task가 EC2를 구동한다.
- ECS cluster는 여러 서브넷에 걸쳐 있을 수 있다.



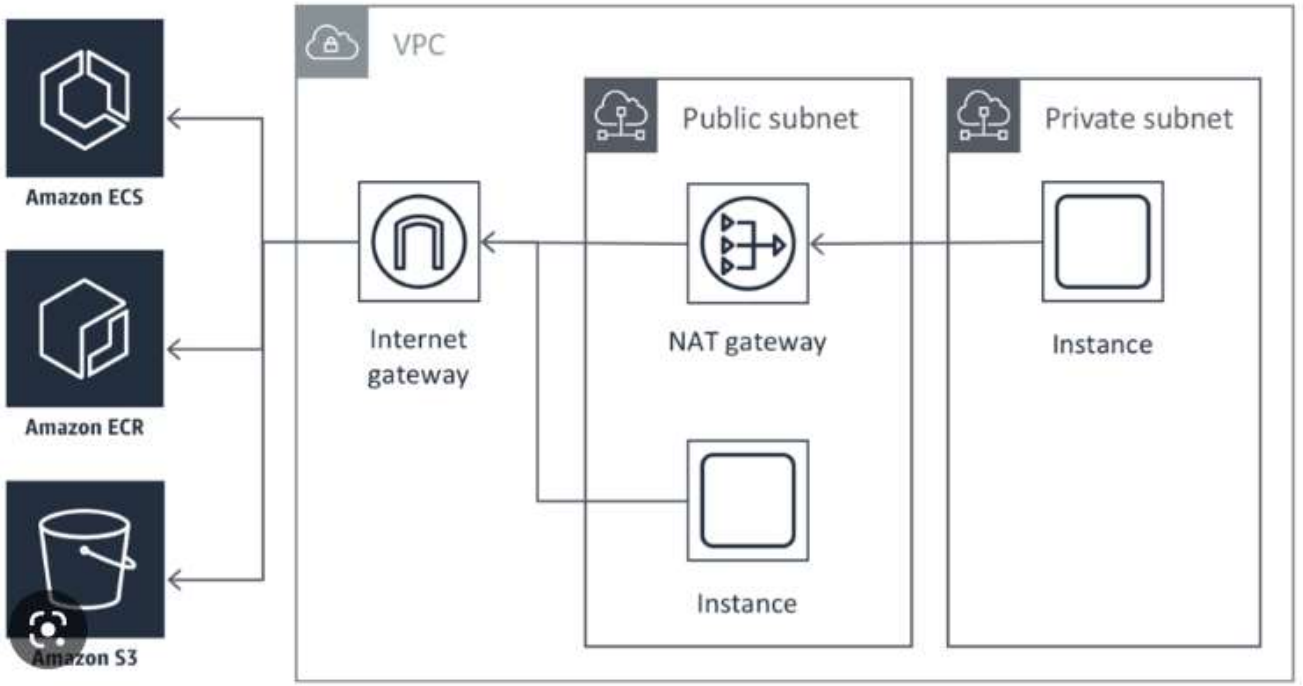
AWS 서브넷 내 ECS cluster

- ECS cluster는 여러 서브넷에 걸쳐 있을 수 있다.
- ECS cluster는 여러 AZ 영역에 걸쳐 있을 수 있다.



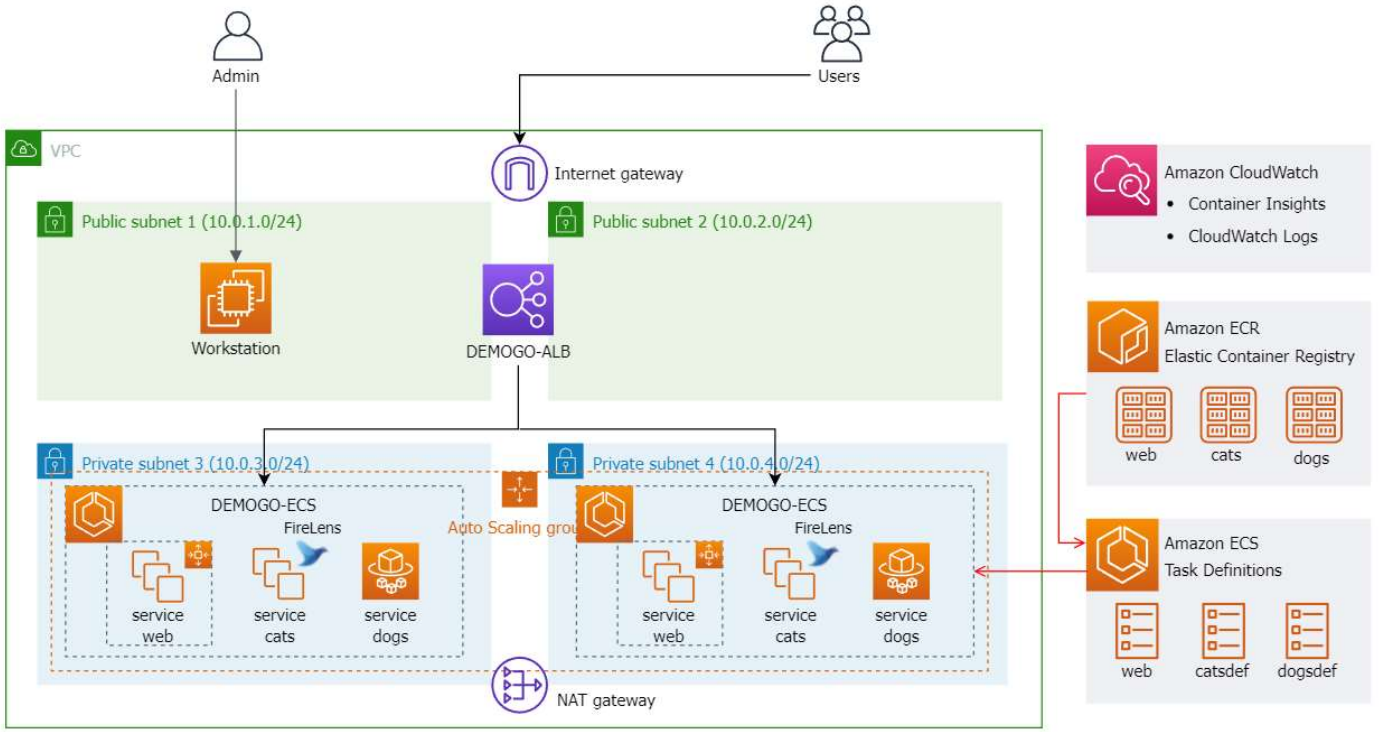
AWS 서브넷 내 ECS instance

- ECS instance는 subnet 내에 있다.



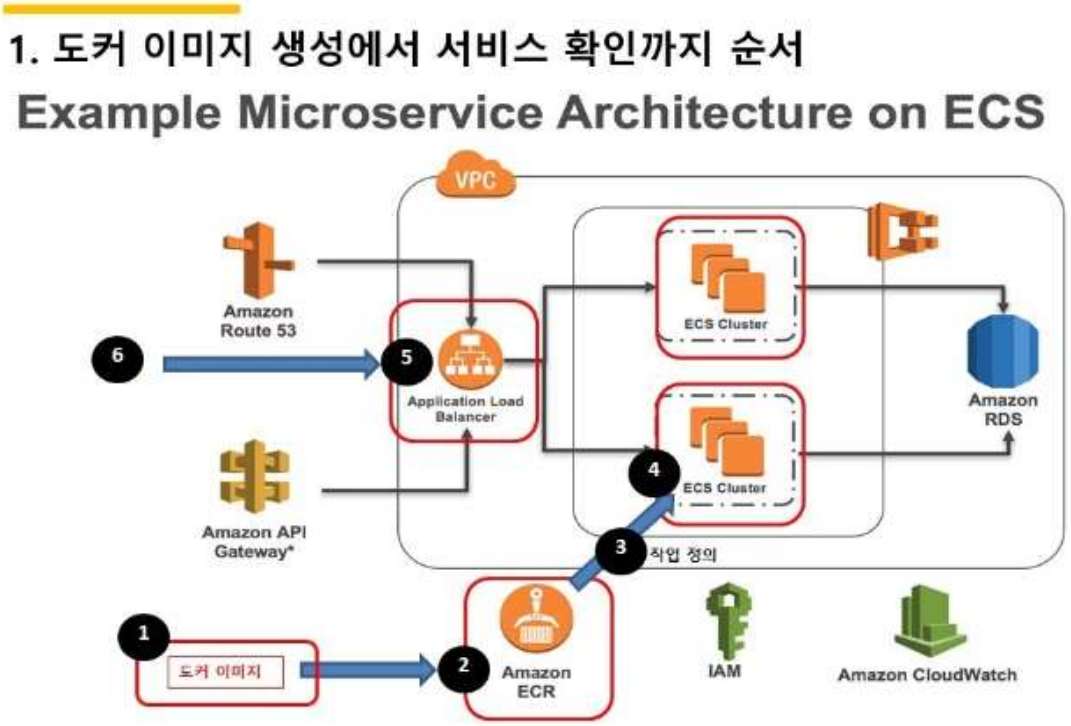
AWS ECS 응용한 시스템 구조 예시

- ECS와 ECR은 VPC 외부인 AWS 서비스에 존재하고 여기서 이미지를 등록하여 컨테이너를 생성하고 Task definition을 관리한다.
- 서브넷 내부에 ECS cluster가 존재하고 그 안에 Task가 service로 묶여있다.



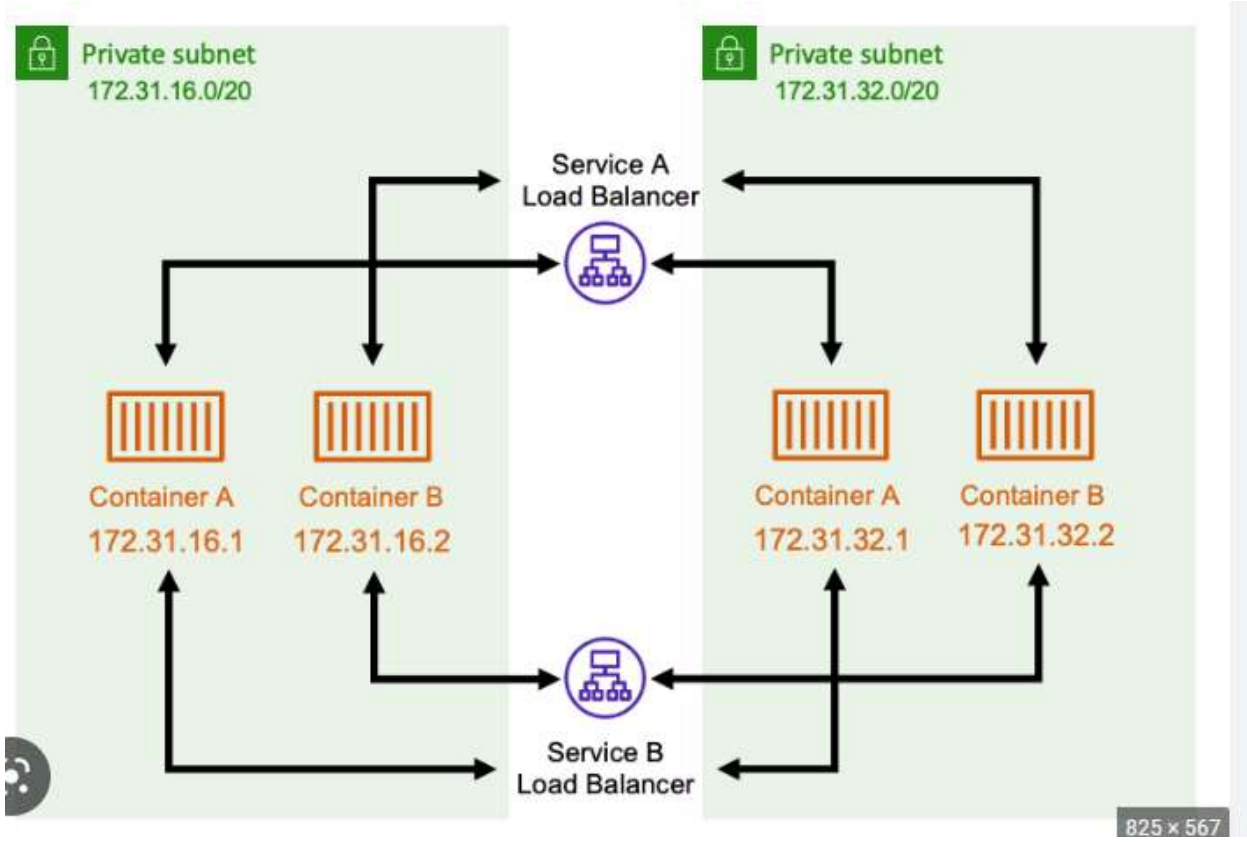
로드밸런서와 ECS cluster

- ALB와 ECS cluster가 서로 연결된다.



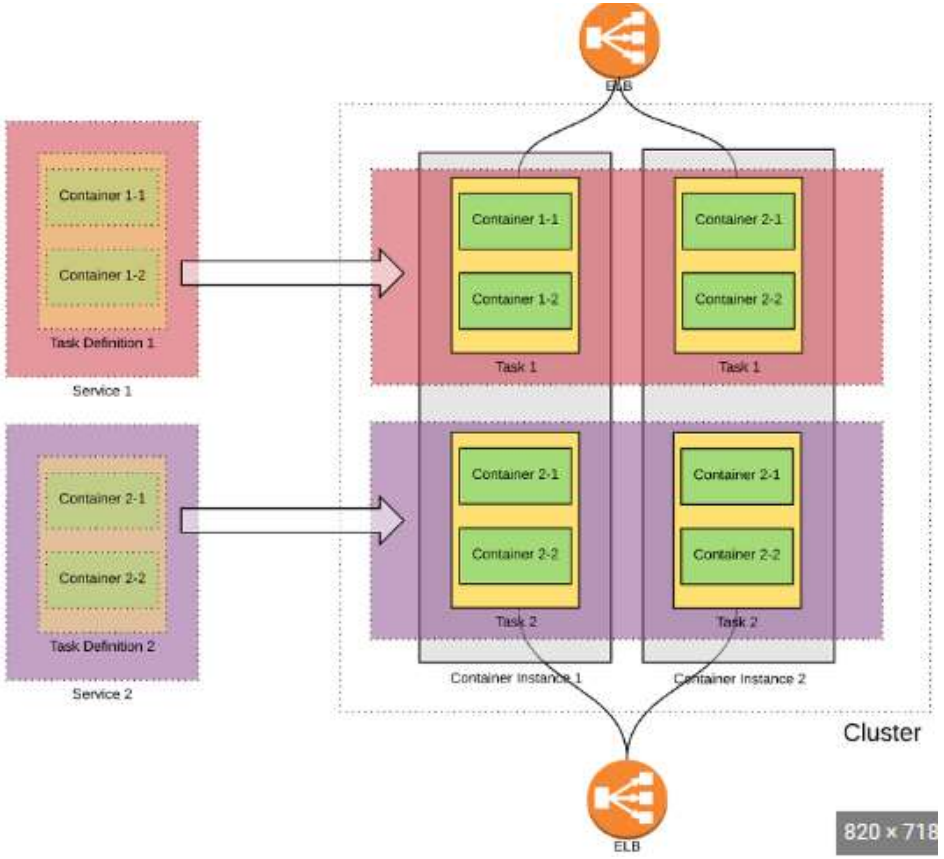
Service와 subnet

- Service는 서로 다른 subnet들에 걸쳐 있다.
- 컨테이너 사이에 로드밸런서가 있기도 함.
- 로드밸런서 위치는 그냥 임의로 정하면 되는듯

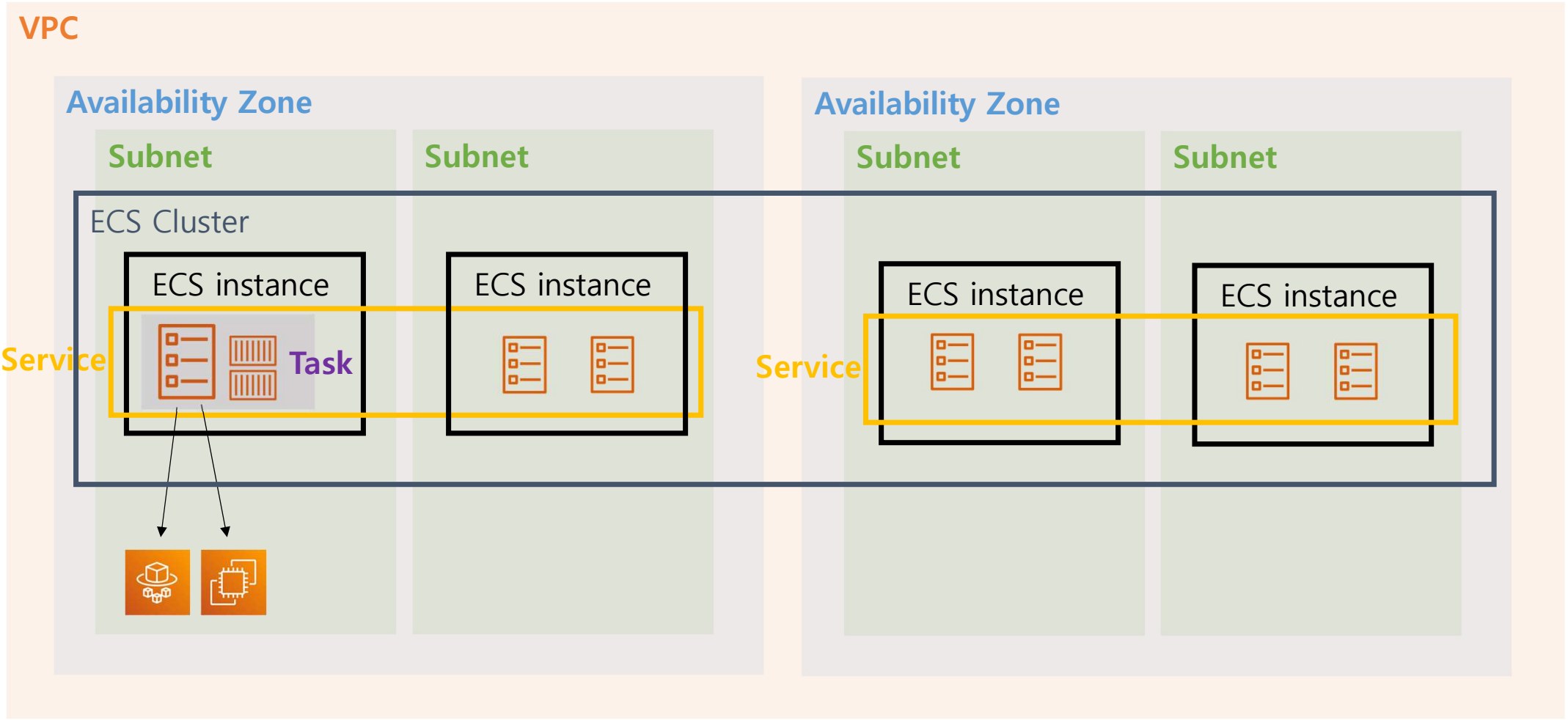


Service와 subnet

- Task 사이에 로드밸런서가 있기도 함
- 로드밸런서 위치는 그냥 임의로 정하면 되는듯



AWS ECS 구성요소 with VPC



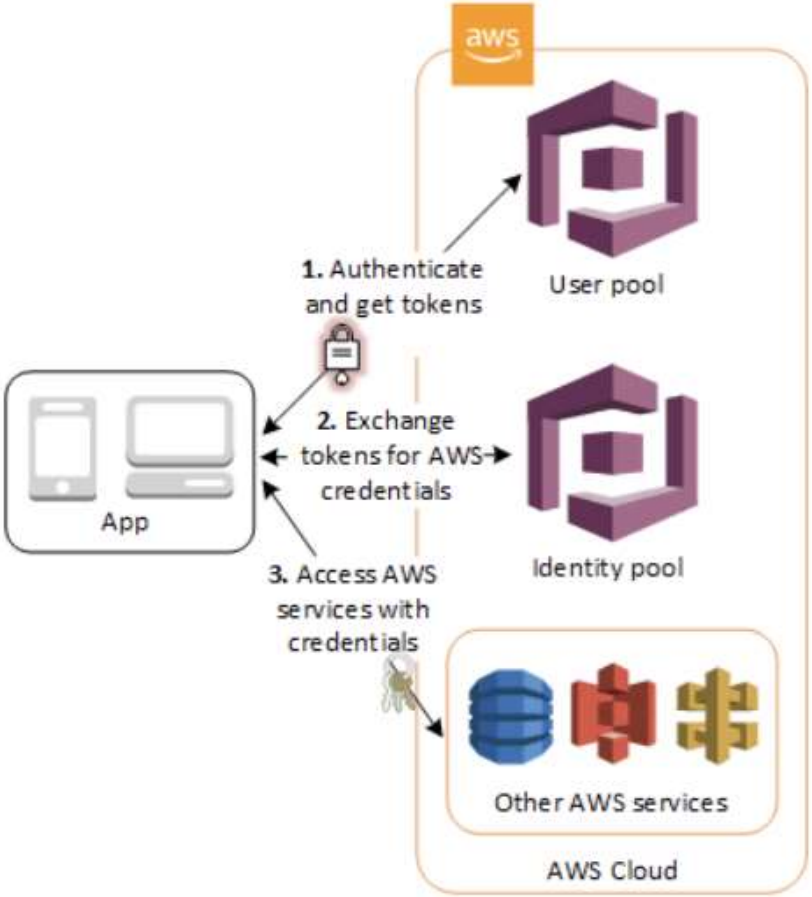
AWS - Cognito

Cognito

- Authentication, Authorization, user 관리를 지원하는 AWS 서비스
- 두 가지 메인 요소: 1. user pool 2. identity pool
- User pool: 앱 사용자의 가입 및 로그인 옵션을 제공
- Identity pool: 다른 AWS 서비스에 대한 사용자 액세스 권한을 부여

Cognito 동작 시나리오

1. 사용자가 user pool을 통해 로그인을 하고, authentication 성공해서 user pool token을 받는다.
2. Identity pool을 통해 User pool token이 AWS credential과 교환된다.
3. 사용자는 AWS credential을 사용해서 S3같은 AWS 서비스에 접근한다.



User pool 기능

- Sign-up and sign-in
- Built in 로그인 UI
- 사용자 프로필 관리
- MFA 기능
- 계정 탈취 보호
- 전화 및 이메일 확인과 같은 보안 기능

Identity pool 기능

- 특정 사용자에게 고유 자격을 부여해서 사용자가 AWS 인프라에 접근할 수 있도록 하는 기능. 다음 사용자를 포함한다.

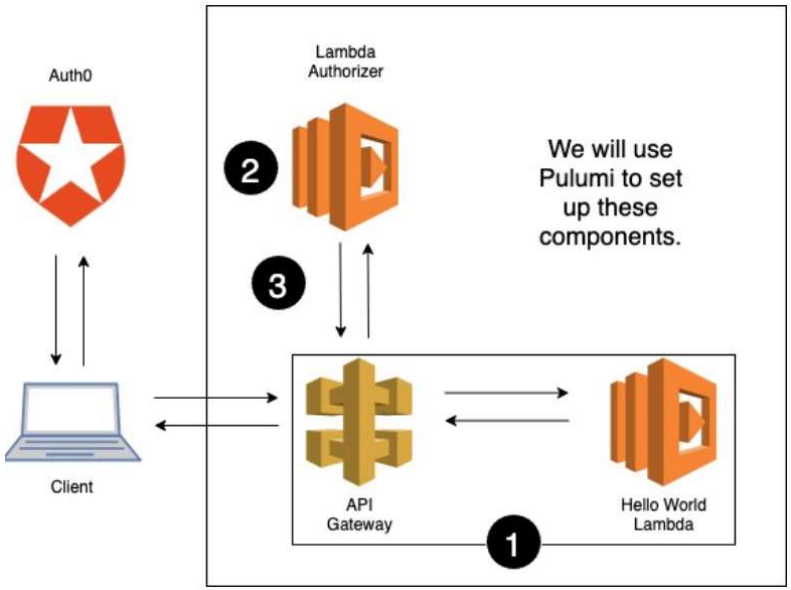
- Cognito user pool 사용자
- Facebook, Google, Apple, SAML 인증된 사용자
- 기존의 인증 프로세스를 통해 인증된 사용자

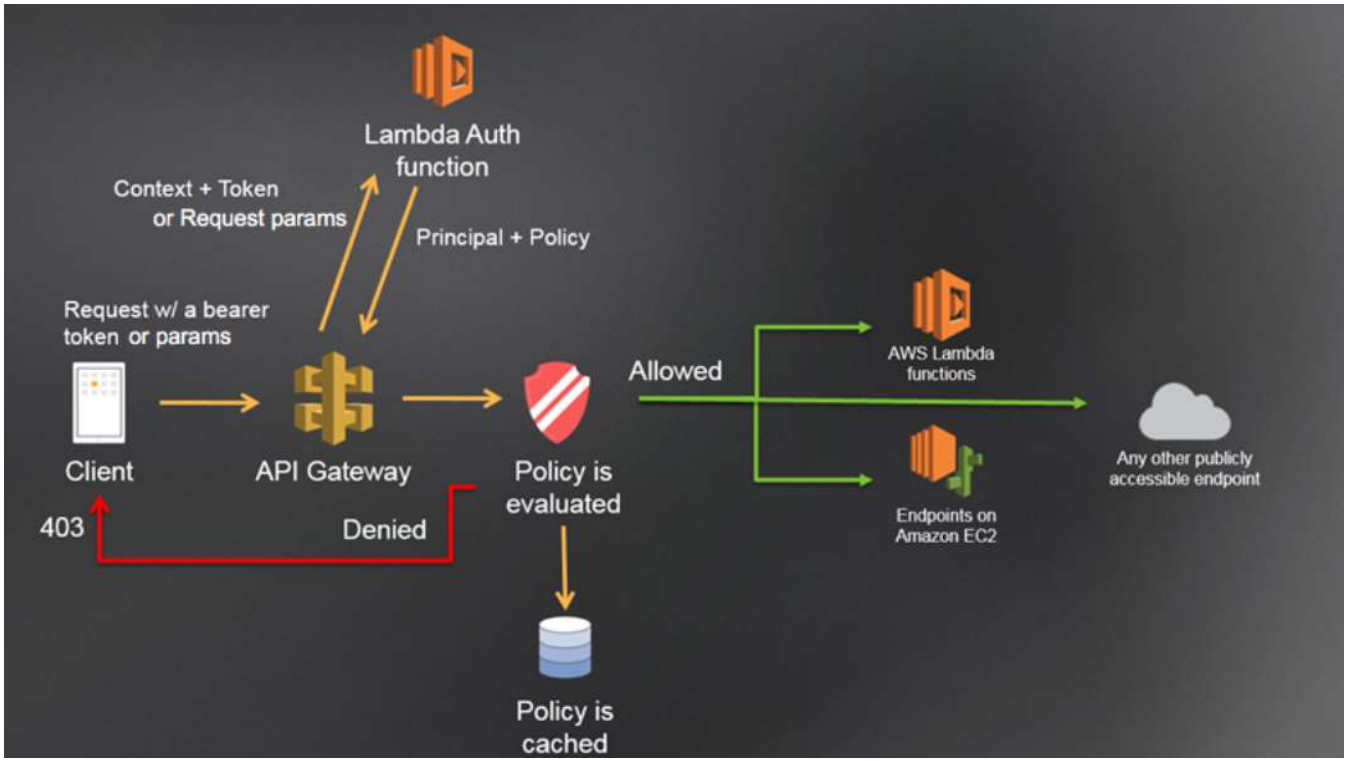
- 유저 프로필 정보를 저장하기 위해서 identity pool은 user pool에 연결되어야 한다.

auth 처리하는 방법 3가지

- API Gateway로 들어오는 요청에 대한 인증을 처리하기 위한 방법은 3가지가 있다.
- token 기반의 Lambda authorizer
- request 기반의 Lambda authorizer
- AWS Cognito user pool을 사용하는 authorizer
- AWS Cognito는 유저 관리와 인증에 대한 많은 부분을 처리해주는 서비스로 기능도 많고 알아야 할 것도 많다. 하지만 제대로 된 서비스를 운영하는 것이 아니라 간단한 개인 프로젝트를 진행할 때 사용하기에는 너무 알아야 할 것도 많고 관리할 것도 많아서 차라리 직접 만든다고 해도 좀 더 간단한 무언가를 쓰는 것이 낫다는 생각이 든다. 때문에 Cognito는 잠시 접어두고 Lambda authorizer로 아주 간단한 수준의 authorizer를 만드는 쪽을 선택.

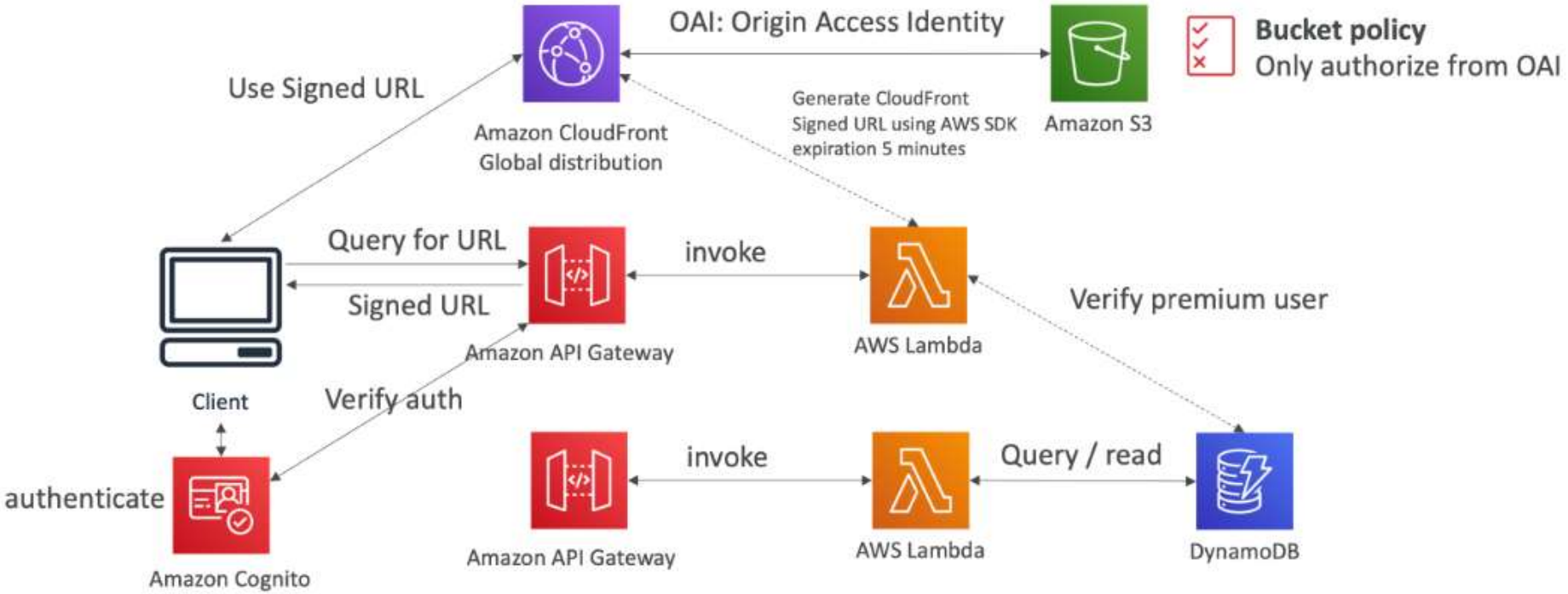
Lambda authorizer 사용하는 구조 예시





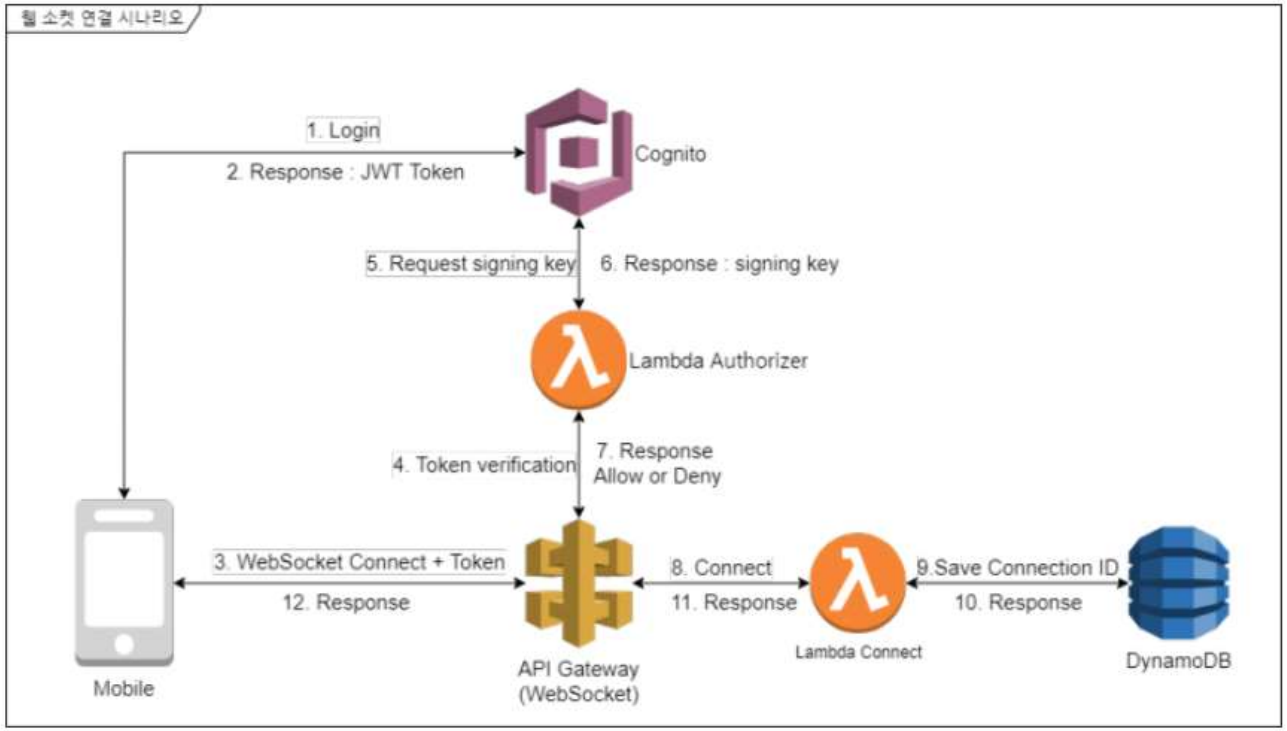
Cognito로 처리하는 구조 예시(1)

생성된 Signed URL은 람다로부터 API Gateway로 전달되고 API Gateway는 이를 클라이언트에 전달한다. 그러면 클라이언트는 CloudFront에 Signed URL을 이용해 접근할 수 있다.



Cognito로 처리하는 구조 예시(2)

게이트웨이(AWS API Gateway)의 HTTP API의 경우 코그니토(AWS Cognito)를 바로 연결해 인증 처리를 할 수 있다. 때문에 람다(AWS Lambda) 함수를 구현할 때 별다른 권한 처리를 하지 않고도 안전하게 권한이 있는 사용자로 받아들여 기능을 구현하면 된다. 하지만 웹소켓을 사용하려면 아래처럼 Lambda Authorizer 중간 다리를 사용해야 한다.



Cognito vs Custom authorizer

Congnito

• 장점

1. AWS가 대부분 알아서 처리한다. 개발자가 Authentication에서 실수할 일이 별로 없음
2. IAM을 통한 AWS 리소스에 대한 세분화된 액세스 제어
3. CSRF 공격을 자동적으로 막아줌. IAM은 5분이 지난 모든 signed 요청을 거부함

단점

1. 클라이언트 측에서 AWS SDK를 사용해야 함. 개발 복잡성 증가
2. 필요한 액세스 권한은 API 게이트웨이에 대한 것뿐이므로 리소스에 대한 세부적인 액세스 제어는 실제로 필요하지 않을 수 있음.

Cognito vs Custom authorizer

Custom authorizer

장점

1. 내가 원하는 방식대로 Authentication 과정을 만들 수 있음
2. AWS SDK에 대한 추가 고려사항이 없음

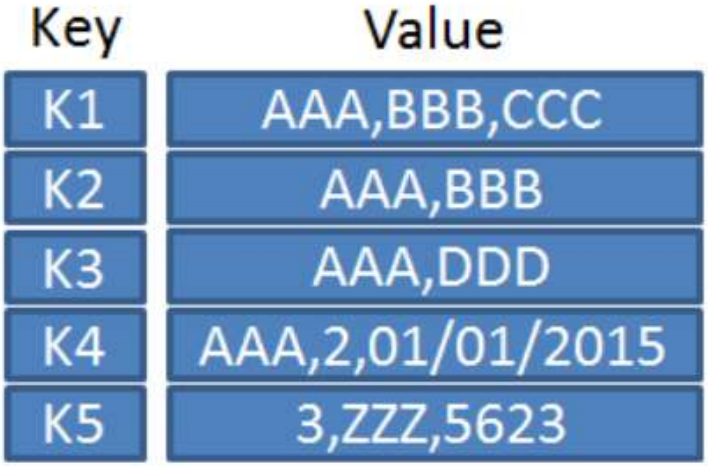
단점

1. Authentication은 정말 어렵고 많은 경우의 수를 고려해야 해서 머리 아픔
2. AWS에서 기능을 다 만들었는데, 굳이 생고생을 해야함.

DB – key-value DB

Key-Value DB

- 가장 단순한 DB
- 구조가 단순하기 때문에 속도도 빠르고 확장성이 좋다.
- 간단한 조회를 위해서 사용하므로 SQL과 같은 질의언어가 없다.
- 하지만 key 값을 통한 검색이 아닌 value를 통한 데이터 조회는 어렵다.
- Key 검색만 허용되므로 key 디자인을 잘 해야함
- 복잡한 데이터 처리에는 부적합하다.



DB – Document DB

Document DB

- Key-Value Database와 같이 데이터를 키-값 형태로 저장
- 키-값 모델과 다른 점이라면 Value가 문서 형태로 저장됨. (문서란 semi-structured entity이며 보통 JSON이나 XML 같은 표준 형식)
- 스키마가 없다. 문서를 추가하면 그게 그냥 스키마가 된다.
- 그래서 각 문서 별로 다른 필드를 가질 수 있어서 데이터 형태가 일정하지 않음. 어플리케이션 코드에서 필드 관리가 중요하다.
- 필수 속성(Null을 허용하지 않는 속성)에 대한 관리도 애플리케이션 레벨에서 관리가 이루어져야 한다.

Mongo DB와 CouchBase의 쿼리 언어

- MongoDB는 독특한 쿼리 언어가 있는데 보통의 SQL이랑 많이 다름

```
-- SQL
SELECT * FROM STUDENT WHERE name = 'Ryan';

-- MongoDB query language
db.STUDENT.find({name:"Ryan"})

-- Couchbase N1QL
SELECT * FROM STUDENT WHERE name = 'Ryan';
```

Mongo DB와 CouchBase의 저장형태

- MongoDB는 JSON이 아니라 BSON 형태로 데이터를 저장한다. 텍스트를 저장하는 비용보다 훨씬 저렴하게 저장할 수 있음

```
{"hello": "world"} →
\x16\x00\x00\x00           // total document size
\x02                       // 0x02 = type String
hello\x00                  // field name
\x06\x00\x00\x00world\x00  // field value
\x00                       // 0x00 = type EOO ('end of object')

{"BSON": ["awesome", 5.05, 1986]} →
\x31\x00\x00\x00
 \x04BSON\x00
 \x26\x00\x00\x00
 \x02\x30\x00\x08\x00\x00\x00awesome\x00
 \x01\x31\x00\x33\x33\x33\x33\x33\x33\x14\x40
 \x10\x32\x00\xc2\x07\x00\x00
 \x00
 \x00
```

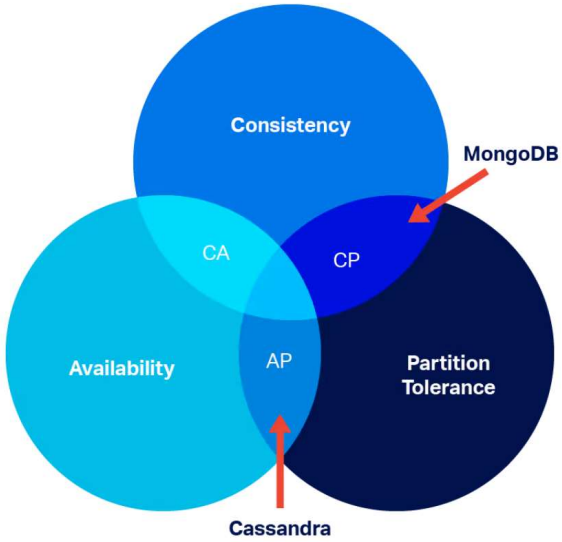
- CouchBase는 그냥 JSON 형태로 데이터를 저장한다.

MongoDB 속도가 빠른 이유

- MongoDB: write할 때 memory 에 먼저 write, 1분 단위로 frushing해서 Disk에 집어넣음. Write가 매우 빠르다. Read는 file의 index를 메모리에 로딩해놓고 찾는다. (memory mapped file 방식) 하지만 치명적인 단점이 있는데, flushing이 실패하면, 데이터가 유실될 수도 있다. 그리고 write read 구조상 memory 사용이 많다.

Mongo DB의 데이터 일관성

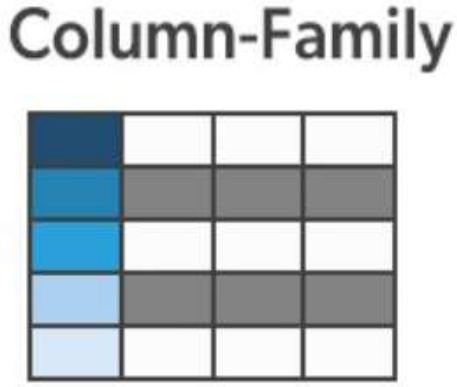
- Mongo DB는 가용성을 포기한 분산 데이터베이스 시스템
- 프라이머리 노드가 중단되었다면 세컨더리 노드 중 하나가 프라이머리 노드로 승격되어야 한다. 새로운 프라이머리 노드가 선출되는 동안 시스템은 모든 쓰기 작업은 잠시 사용 불가능(unavailable)한 상태가 된다. 따라서 MongoDB는 CP 시스템으로 분류된다

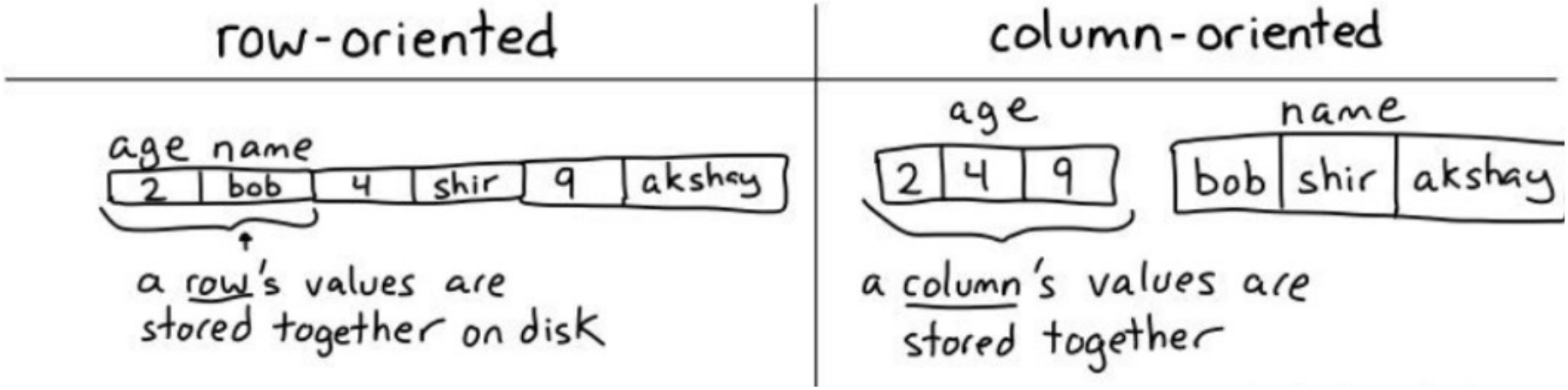


DB – Column base DB

Column Oriented Database

- 관계형 데이터베이스(RDB) 는 일반적으로 트랜잭션 응용 프로그램의 경우 데이터 row를 저장하는 데 최적화되어 있지만 Column Oriented Database는 일반적으로 Column의 빠른 검색에 최적화되어 있다.





RDBMS와 비교

- 관계형 데이터베이스는 row 단위로 값을 입력.

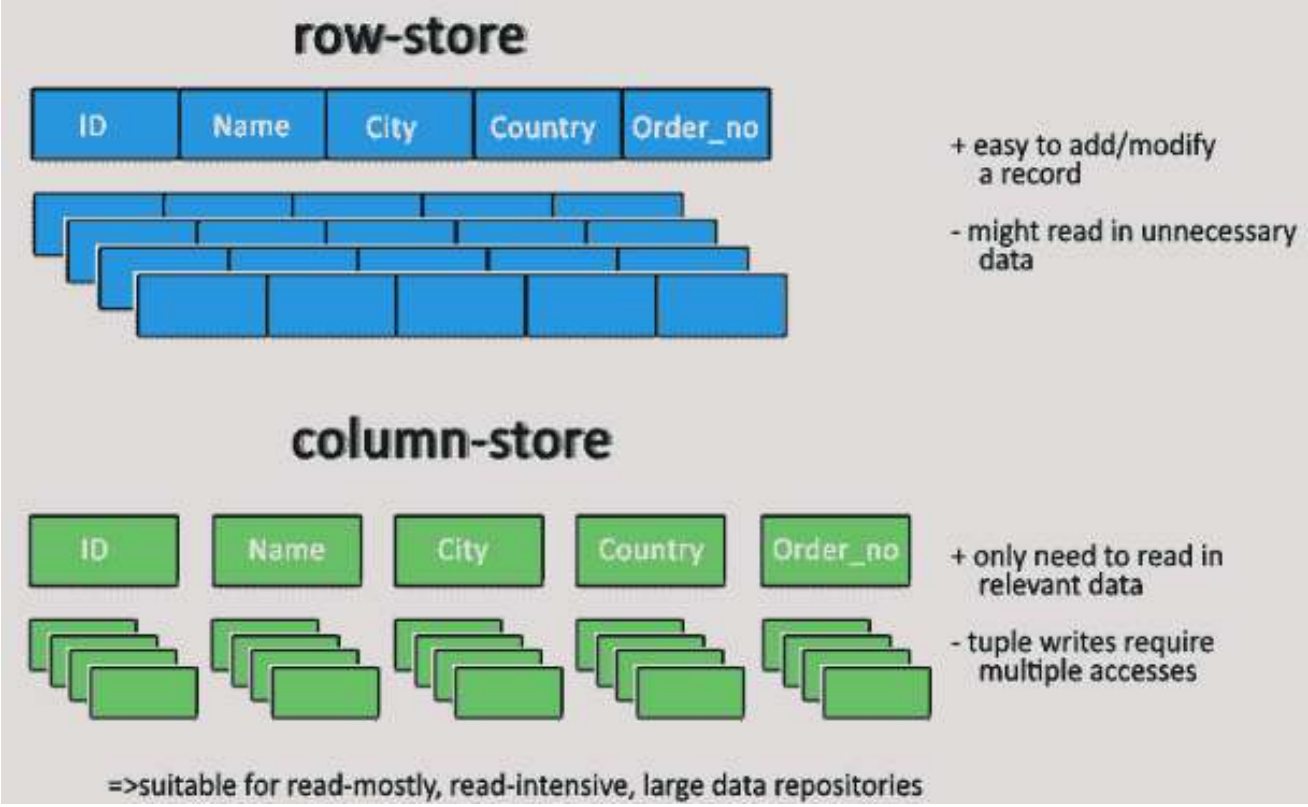
```
mysql> CREATE TABLE authors (id INT, name VARCHAR(40), email VARCHAR(40));
mysql> INSERT INTO authors (id,name,email) VALUES(1,"wonyoung","wy@abc.com");
```

- Column형은 column 단위로 값을 입력.
- 얼핏 보면 RDBMS와 비슷해 보이지만 디스크 차원에서 본다면 확연한 차이를 보임

```
cql> CREATE TABLE author_table ( id text PRIMARY KEY, name text, email text );
cql> INSERT INTO author_table (id, name , descript ) VALUES ( 'myid_0', 'wonyoung', 'wy@abc.com');
```

RDBMS와 비교

- RDBMS와 Columnar DB의 저장 방식 비교



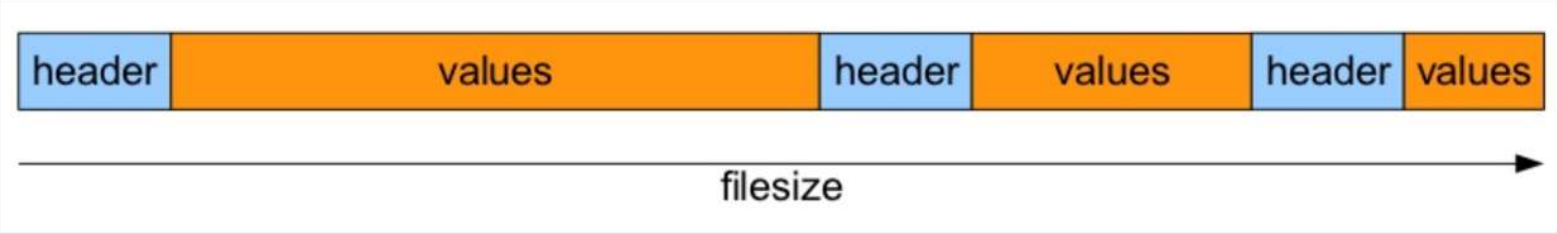
RDBMS의 장단점

[장점]

- 데이터의 끝에 행을 추가하기만 하면 되기 때문에 저장 속도가 빠르다.
- RDBMS의 row 방식은 이렇게 균일하지 못한 데이터 길이 때문에 disk 가 position을 찾는데 더 오랜 시간을 사용하게 된다.

[단점]

- 열 단위로 데이터를 탐색할 때 속도가 느리다. 필요치도 않은 열을 탐색하느라 시간을 낭비하게 됨



- `varchars` 라는 타입을 생각해보면 매우 다양한 value의 크기를 확인할 수 있다.
- 이는 곧 Position을 계산하기 어렵게 만들며 빠르게 Record를 읽을 수 없게 만든다.

RDBMS에 비해 Columnar DB가 유리한 점

[장점]

- 데이터 집합을 만드는 경우 추가 메모리 소모 없이 필요한 결과를 출력할 수 있으며, 이에 따라 성능이 크게 향상된다.
- 같은 열에는 유사한 데이터가 반복되기 때문에, 매우 작게 압축 할 수 있음. 압축되지 않은 행 지향 DB와 비교하여 1/10 이하로 압축 가능
- 열 단위로 데이터를 끌어 가지고 와서 특정 열 속성에 대해 데이터를 분석할 때 매우 유용하다. <- 그래서 매우 방대한 데이터를 처리할 때 분석에 매우 유리함.

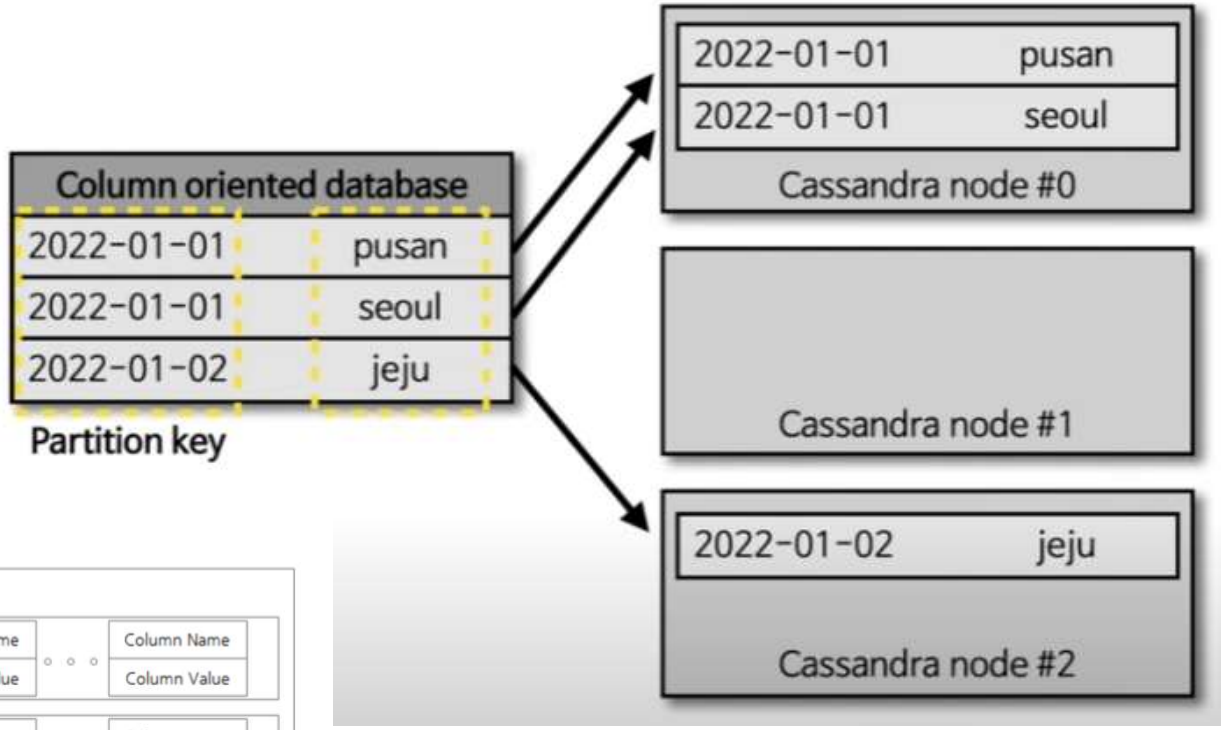
[단점]

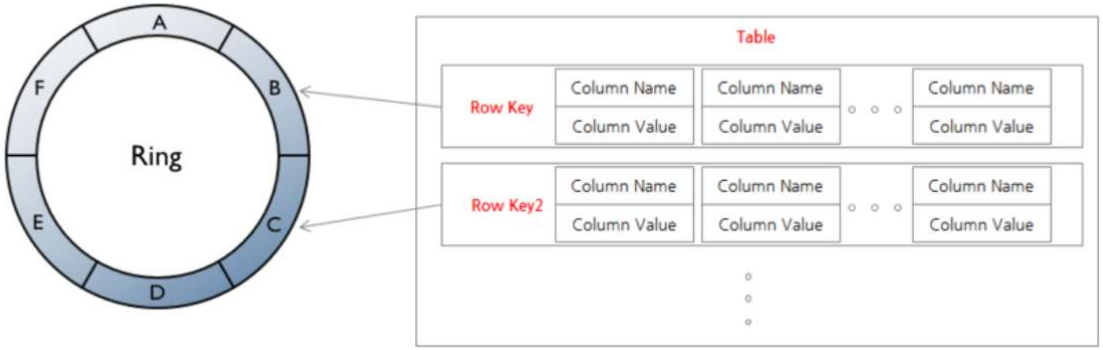
- 행을 새로 추가하는 것이 상대적으로 느리다. 데이터 추가 작업을 수행할 때, 각 데이터의 마지막 위치를 확인해야하는 작업이 필요하다.

Matt	Dave	Tim	Los Angeles	San Francisco	Oakland	27	30	33

Column형 DB의 partion key

- 파티션 키에 의해 특정 노드로 저장되게 된다.
- 데이터가 늘어나면 노드를 늘릴 수 있음. -> DB 선형으로 scale-out 장점





NoSQL에서 Column DB의 장점

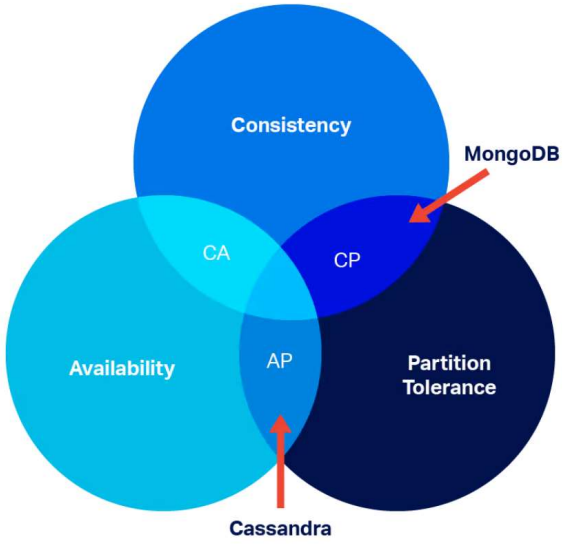
- Columnal Storage 는 전체 디스크 I/O 요구 사항을 크게 줄이고 디스크에서 로드해야 하는 데이터의 양을 줄인다.
- 분산 DB: 각 노드는 모두 프라이머리이다. 각 노드들은 p2p 프로토콜을 통한 동기화 작업을 수행한다. 단일 장애 지점 문제가 없다.
- 결함허용: 한 개의 노드에 장애가 발생하면, 탐지 및 교체 또는 복원할 수 있다.
- 임계점을 넘는 다량의 데이터를 처리할 때는 Mongo DB보다 좋은 성능을 낸다.

NoSQL에서 Column DB의 단점

- 이미 생성된 테이블의 파티션 키를 수정할 수 없음 (처음 설계 때부터 Plan B까지 제대로 설계해야 함.)
- 설계를 잘못 해서 테이블을 삭제해야 하는 순간이 오면 매우 난감해짐
- 파티션 키가 아닌 데이터를 where로 필터링하기 어렵다. Secondary index를 지원하긴 하지만, 전체 클러스터를 스캔하기 때문에 상용환경에서 사용하는 건 매우 제한됨
- 무엇보다 사용이 어렵다.
- 복잡한 쿼리 처리하기 어렵다.

Cassandra의 데이터 일관성

- 카산드라는 Peer-to-peer 시스템이다. 즉 카산드라는 프라이머리 노드 없이 모든 노드가 읽기 작업과 쓰기 작업을 수행할 수 있고 복제본을 분리된 다른 노드에 저장한다.
- 노드가 다른 노드와 통신할 수 없어도 해당 노드는 여전히 읽기 작업과 쓰기 작업을 수행할 수 있으나 데이터가 다른 노드와 맞지 않는 상태, 즉 일관성이 깨진 상태가 된다.
- 카산드라는 일관성을 포기한 대신 높은 가용성을 확보한 AP 시스템으로 분류된다.



Hbase를 안 쓰는 이유

- Cassandra도 어려운 편인데, Hbase의 learning curve가 더 높다.
- Cassandra는 CQL을 제공하지만, Hbase는 그런거 없다. Hbase shell을 사용해서 쿼리해야한다. 더 복잡하다.
- 클러스터 구축 면에서도 Hbase가 더 난이도가 높다. 단독 클러스터 구성이 불가하고, 최소 5대의 데이터 노드와 하나의 네임노드를 필요

DB – graph DB

Graph DB

- Graph이론을 활용한 것. 데이터가 그래프 형태로 저장된다.
- 그래프는 node와 relation으로 구성된다.
- 다른 NoSQL과 다르게 트랜잭션을 지원한다.
- 질의는 그래프 순회를 통해 이뤄진다. index free adjacency라고 하는데, 인덱스를 사용하지 않고 연결된 노드를 찾을 수 있다. 관계를 표현할 때 RDBMS보다 더 직관적으로 표현한다.
- RDBMS에서 join을 많이 사용하면 성능 저하가 필연적인데, graph는 관계로 다른 노드와 이어지므로 이런 것이 덜하다.
- 스키마가 없음

Graph DB 활용할 곳

- 복잡한 쿼리를 처리하기 용이하다.
- 패턴이 일치하지 않는 것들은 무시하고 넘어가기 때문에 연결된 패턴을 찾는데 있어서는 매우 빠른 속도를 보장
- 노드의 데이터 크기와 쿼리의 성능이 독립적이기 때문에 데이터 사이즈가 늘어난다고 해서 성능의 저하가 발생하진 않음
- 복합성이 낮은 단순한 구조의 간단한 질의를 사용할 경우 굳이 GraphDB를 사용할 이유가 없음
- 그래프 구조를 가지는 소셜 네트워크의 DB로 쓰기 적합하다.
- 질의 언어를 배우기 어렵다.

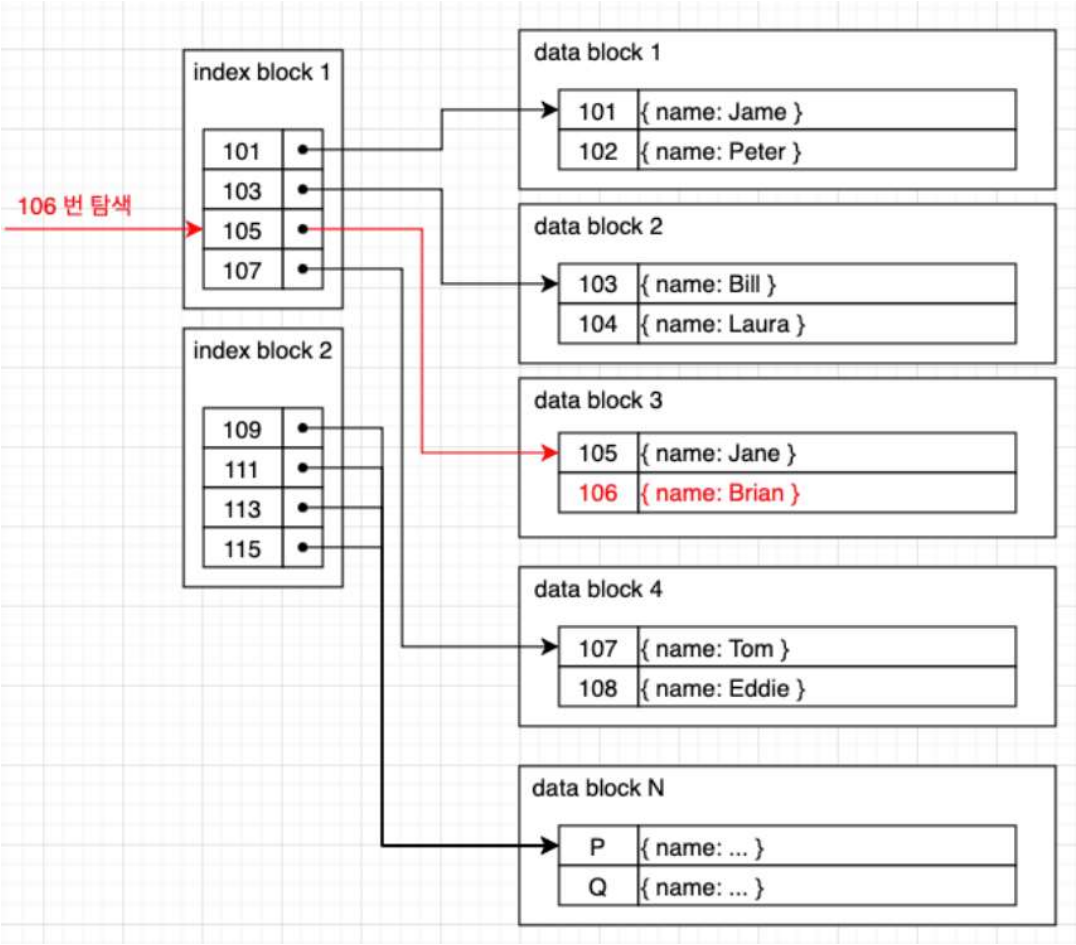
DB – index

DB에서 index란?

- Index란 Data와 별개로 저장되는, DB의 Disk 탐색속도를 높여주는 부가적인 자료구조
- 원래 Disk I/O에 필요한 시간은 memory I/O에 비해 훨씬 오래 걸린다.
- Disk에 데이터가 많으면 모든 Block을 탐색하는데 시간이 매우 오래 걸린다. 이것을 index 자료구조를 통해 Disk I/O 횟수를 줄여서 시간을 단축시키는 것이다.
- Index 파일은 실제 data에 비해 파일 공간을 덜 차지하므로 같은 용량 내에 더 많은 데이터를 담을 수 있고, 그래서 한 번 스캔할 때 더 많이 스캔할 수 있다. 이것이 정보를 찾는데 시간을 단축시켜준다.

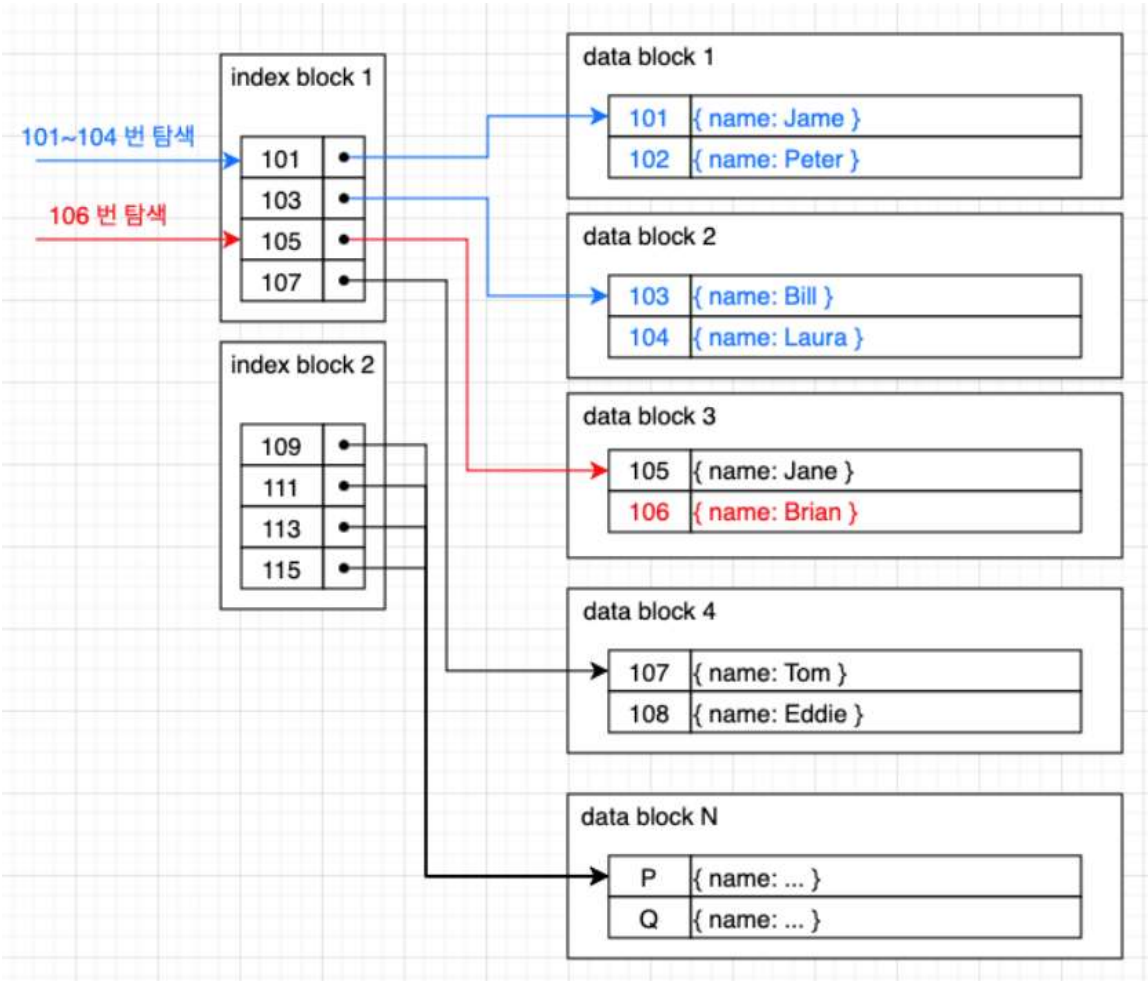
DB에서 index란?

- Index를 이용한다면, [index block1 -> (index block2) -> data block3-> 2 번째 레코드] 이렇게 탐색하게 되면 총 I/O는 2~3번 일어난다. 다른 레코드를 검색한다고 해도 총 2~3번이 보장될 것이다.
- index를 이용하지 않으면, 최선의 경우 바로 data block3에 random access 하게 되어 1번의 I/O만으로 id=106을 검색성공 하겠지만, 최악의 경우는 data block1, data block2, ... data block N 이렇게 모든 N개의 disk block을 scan 해야 할 수도 있다.



Cluster index (primary index)

- Primary key는 테이블당 하나만 가질 수 있음
- 데이터를 넣을 때 key에 따라 순서가 정렬되면서 넣어지고
- 데이터를 읽을 때에도 특정 key를 선택하면 어떤 Block에 데이터가 저장되어 있는지 바로 알기 때문에 읽는 속도가 빠르다.



Secondary indexes

- Primary key이외에도 필요한 정렬 기준이 있을 경우 사용하는 index다.
- 테이블 당 여러 개 있을 수 있고 unique하지 않아도 된다.
- 순서대로 정렬되어 있지 않으므로 탐색에 시간이 더 걸리는 편