학습 내용 정리 - 3월 2주차

박시준

Table of content

Algorithm – Floyd Warshal

[프로그래머스] 순위

- 그래프라고 적힌 문제

- 권투 대회 결과가 이차원 배열로 주어짐
- 앞: 이긴 사람
- 뒤: 진 사람

입출력 예

n	results	return
5	[[4, 3], [4, 2], [3, 2], [1, 2], [2, 5]]	2

순위

문제 설명

n명의 권투선수가 권투 대회에 참여했고 각각 1번부터 n번까지 번호를 받았습니다. 권투 경기는 1대1 방식으로 진행이 되고, 만약 A 선수가 B 선수보다 실력이 좋다면 A 선수는 B 선수를 항상 이깁니다. 심판은 주어진 경기 결과를 가지고 선수들의 순위를 매기려 합니다. 하지만 몇몇 경기 결과를 분실하여 정확하게 순위를 매길 수 없습니다.

선수의 수 n, 경기 결과를 담은 2차원 배열 results가 매개변수로 주어질 때 정확하게 순위를 매길 수 있는 선수의 수를 return 하도록 solution 함수를 작성해주세요.

제한사항

- 선수의 수는 1명 이상 100명 이하입니다.
- 경기 결과는 1개 이상 4,500개 이하입니다.
- results 배열 각 행 [A, B]는 A 선수가 B 선수를 이겼다는 의미입니다.
- 모든 경기 결과에는 모순이 없습니다.

[프로그래머스] 딕셔너리로 시도 -> 실패

- 단순 딕셔너리를 써서 풀어보려고 했으나 잘 되지 않음
- 알고 보니 플로이드-와샬 알고리즘을 사용하는 문제

```python
from collections import defaultdict
def solution(n, results):
    answer = 0
    win = defaultdict(list)
    lose =  defaultdict(list)
    for i in range(len(results)):
        win[results[i][0]].append(results[i][1])
        lose[results[i][1]].append(results[i][0])
    for i in range(n):
        if len(win[i]) + len(lose[i]) == n - 1:
            print(i)
    return answer
```

Floyd-Warshall Algorithm

- 플로이드 와샬 알고리즘: 모든 노드 간 최단 경로를 구하는 문제
- 다익스트라 알고리즘이 한 노드에서 다른 모든 노드로 가는 최단 경로를 구하는 알고리즘이라면 플로이드 와샬 알고리즘은 한 번 실행하여 모든 노드 간 최단 거리를 구할 수 있음.
- 음의 가중치 간선이 있어도 사용할 수 있다.

Floyd-Warshall Algorithm 구현 방법

- 구현 방법은 매우 단순함. 반복문을 세 번 중첩하면 된다. O(n^3)

1. 우선 그래프를 2차원 인접 행렬로 나타낸다. 인접한 노드가 있으면 간선의 가중치만큼 값을 가지고, 인접 노드가 없으면 무한대의 값을 가진다.
2. 노드 S와 노드 E와의 거리를 구하기 위해 그 사이 노드 M을 모두 탐색하면서 d[s][e] > d[s][m] + d[m][e] 라면 d[s][e] = d[s][m] + d[m][e]로 업데이트 해준다.

```
void Floyd_Warshall() {
  for(m=1; m<=N; m++)
    for(s=1; s<=N; s++)
      for(e=1; e<=N; e++)
        if (d[s][e] > d[s][m] + d[m][e])
                    d[s][e] = d[s][m] + d[m][e];
}
```

첫번째 풀이 방법 - 플로이드 와샬

- 최단거리를 구하는 플로이드 와샬 알고리즘을 응용하는 문제
- 단 여기서는 행렬에 간선 거리를 입력하는 게 아니라 '승(1)과 패(-1)'를 입력한다. 2차원 인접행렬의 초기값은 0으로 해주고, results를 반영하여 이긴 사람은 1로 진 사람은 -1로 초기 셋팅
- 반복문 3번 돌리면서 A -> M 이고 M -> B면 A -> B이고 B는 A에게 진 것으로 설정한다.
- O(n^3). 시간은 오래 걸림

```python
def solution(n, results):
    answer = 0
    matrix = [[0 for _ in range(n)] for _ in range(n)]
    for a, b in results:
        matrix[a - 1][b - 1] = 1
        matrix[b - 1][a - 1] = -1
    for m in range(n):
        for a in range(n):
            for b in range(n):
                if a == b or matrix[a][b] in [1, -1]:
                    continue
                if matrix[a][m] == matrix[m][b] == 1:
                    matrix[a][b] = 1
                    matrix[b][a] = -1
    for row in matrix:
        if row.count(0) == 1:
            answer += 1
    return answer
```

```
테스트 4 〉 통과 (1.34ms, 10.2MB)
테스트 5 〉 통과 (2.51ms, 10.1MB)
테스트 6 〉 통과 (6.40ms, 10.1MB)
테스트 7 〉 통과 (40.97ms, 10.3MB)
테스트 8 〉 통과 (58.30ms, 10.5MB)
테스트 9 〉 통과 (75.59ms, 10.5MB)
테스트 10 〉 통과 (73.74ms, 10.4MB)
```

두번째 풀이 방법 – Hash와 Set을 사용

- 플로이드 와샬을 쓰지 않고도 풀 수 있음
- 딕셔너리와 **set**를 함께 쓰면 풀 수 있었음.
- 두번째 루프인 업데이트 루프를 돌리면 된다.
- A->B이고 B->C이면 A->C로 업데이트 하는 점은 플로이드-와샬과 같다.
- 반복문을 두 번 중첩해서 훨씬 더 빠르지만, 공간은 더 많이 쓰고, 변수 사용이 상당히 복잡함

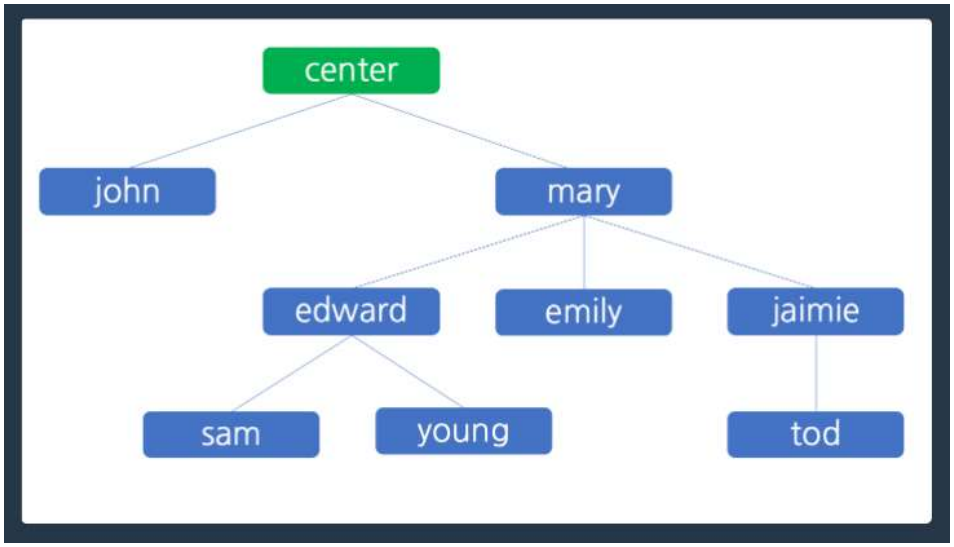
```python
from collections import defaultdict
def solution(n, results):
    answer = 0
    win = defaultdict(set)
    lose =  defaultdict(set)
    for w, l in results:
        win[l].add(w)
        lose[w].add(l)

    for i in range(1, n + 1):
        for winner in win[i]:
            lose[winner].update(lose[i])
        for loser in lose[i]:
            win[loser].update(win[i])

    for i in range(1, n + 1):
        if len(win[i]) + len(lose[i]) == n - 1:
            answer += 1
    return answer
```

```
테스트 4 : 통과 (0.04ms, 10.2MB)
테스트 5 : 통과 (0.28ms, 10.2MB)
테스트 6 : 통과 (0.55ms, 10.2MB)
테스트 7 : 통과 (3.05ms, 10.4MB)
테스트 8 : 통과 (3.50ms, 10.8MB)
테스트 9 : 통과 (8.35ms, 11.2MB)
테스트 10 : 통과 (5.53ms, 11.2MB)
```

Algorithm – Union find

[프로그래머스] 다단계 칫솔 판매

- Enroll 배열에 각 사람의 이름 주어짐
- Referral 배열에 각 사람의 보스가 주어짐
- Seller 배열에 판매에 성공한 사람 이름이 주어짐
- Amount 에 판매에 성공한 사람이 몇 개를 팔았는지 주어짐
- 보스는 부하의 이득의 10%를 가져감
- 각 사람들의 이득이 얼마 되는지 계산하여 반환



enroll	referral	seller	amount	result
["john", "mary", "edward", "sam", "emily", "jaimie", "tod", "young"]	["-", "-", "mary", "edward", "mary", "mary", "jaimie", "edward"]	["young", "john", "tod", "emily", "mary"]	[12, 4, 2, 5, 10]	[360, 958, 108, 0, 450, 18, 180, 1080]

[프로그래머스] 딕셔너리로 시도 -> 실패

- 딕셔너리를 활용하여 반복문 한번씩 돌려서 값을 계산했으나 오답이 나옴
- 어느 child를 먼저 검색하냐에 따라서 결과값이 달라지게 된다. 정렬 순서가 매우 중요해짐
- 부모 하나만 바꾸면 안 되고 타고 올라가서 루트까지 다 영향을 줘야 한다.

```
실행 결과    실행한 결괏값 [360,950,120,0,450,20,180,1080]이 기댓값 [360,95
             8,108,0,450,18,180,1080]과 다릅니다.
    출력     {'young': 1080, 'john': 360, 'tod': 180, 'emily': 450, 'mar
             y': 950, '-': 140, 'jaimie': 20, 'edward': 120}
```

```python
def solution(enroll, referral, seller, amount):
    answer = []
    profit = {}
    parents = {}
    for i in range(len(seller)):
        profit[seller[i]] = amount[i] * 100
    for i in range(len(enroll)):
        parents[enroll[i]] = referral[i]
    for child in parents:
        if child in profit:
            if parents[child] in profit:
                profit[parents[child]] += int(profit[child] * 0.1)
            else:
                profit[parents[child]] = int(profit[child] * 0.1)
            profit[child] = int(profit[child] * 0.9)
    for name in enroll:
        if name in profit:
            answer.append(profit[name])
        else:
            answer.append(0)

    return answer
```

[프로그래머스] 풀이 방법: Union-find

- Union-find를 구현할 필요는 없지만, Union-find의 개념은 가져와야 한다.
- **부모가 루트일 때까지 연쇄적으로 값을 입력하기**
- defaultdict를 쓰면 answer에 기본값이 없어도 바로 값을 더할 수 있음
- 그리고 Union-find 알고리즘의 핵심은 **'재귀함수'**. 재귀가 있어야 루트까지 연산할 수 있음

```python
from collections import defaultdict
def solution(enroll, referral, seller, amount):
    answer = []
    enroll_dic = defaultdict(str)
    sell_dic = defaultdict(int)

    def settle(node, money):
        if node == "-" or money < 1:
            return
        sell_dic[node] += money - (money // 10)
        settle(enroll_dic[node], money // 10)

    for i in range(len(enroll)):
        enroll_dic[enroll[i]] = referral[i]
    for i in range(len(seller)):
        settle(seller[i], amount[i] * 100)
    for person in enroll:
        answer.append(sell_dic[person])

    return answer
```

Algorithm – two pointer

[프로그래머스] 풍선 터뜨리기

- 풍선이 일렬로 나열되어 있음.
- 다음 과정을 거쳐서 풍선이 하나만 남을 때까지 계속 터뜨린다.

1. 임의의 인접한 두 풍선을 선택하여 둘 중 하나를 터뜨린다.

2. 풍선을 터뜨리고 빈 공간이 생기면 빈 공간이 없어지도록 풍선을 모은다.

- 두 풍선을 터뜨릴 때 항상 큰 값을 가지는 풍선을 터뜨려야 한다. 예외적으로 딱 한 번만 더 작은 값의 풍선을 터뜨릴 수 있다.
- 이런 방식으로 풍선을 터뜨릴 때 마지막까지 살아남을 수 있는 풍선의 개수를 반환하라.

a	result
[9,-1,-5]	3
[-16,27,65,-2,58,-92,-71,-68,-61,-33]	6

[프로그래머스] 풍선 터뜨리기 - 문제 분석

- 최후까지 남을 수 있는 풍선은 -16, -92, -71, -68, -61, -33이 써진 풍선으로 모두 6개입니다.

```
[-16,27,65,-2,58,-92,-71,-68,-61,-33]    6
```

- 좌평면, 우평면 동시에 기준 풍선보다 작은 값의 풍선이 있으면 그 기준 풍선은 살아남지 못한다.
- '-2' 풍선보다 작은 풍선이 좌평면과 우평면에 모두 있으므로 '-2' 풍선은 살아남지 못함.

[프로그래머스] 반복문 + two pointer

- 일단 모든 풍선을 반복문을 돌려서 기준 풍선으로 잡는다.
- 반복문 내 반복문으로 pointe를 움직여준다. 기준 풍선보다 더 작은 값이 있으면 small 체크를 활성화 시킨다.
- 양평면에 기준 풍선보다 더 작은 풍선이 있다면 answer를 감소시킴
- 하지만 시간 초과 발생함

```python
def solution(a):
    answer = len(a)

    for i in range(len(a)):
        left = i - 1
        left_small = False
        right = i + 1
        right_small = False
        while left >= 0:
            if a[left] < a[i]:
                left_small = True
                break
            left -= 1
        while right < len(a):
            if a[right] < a[i]:
                right_small = True
                break
            right += 1
        if left_small == True and right_small == True:
            answer -= 1

    return answer
```

```
통과 (0.01ms, 10.1MB)
통과 (0.01ms, 10.2MB)
통과 (1.64ms, 9.99MB)
통과 (255.88ms, 14.2MB)
통과 (1389.71ms, 31.6MB)
통과 (2215.37ms, 42.5MB)
통과 (3122.90ms, 53.4MB)
통과 (9988.47ms, 53.4MB)
실패 (시간 초과)
실패 (시간 초과)
실패 (시간 초과)
실패 (시간 초과)
실패 (시간 초과)
실패 (시간 초과)
실패 (시간 초과)
```

[프로그래머스] 풀이법 – two pointer

- Two pointer를 쓰는 것은 맞다. 하지만 반복문 내 반복문으로 돌릴 필요 없음
- Left 출발 포인터와 right 출발 포인터를 사용해서 동시에 탐색하면서 더 작은 값이 나오면 그 작은 값을 minimum 값으로 갱신해주고 answer를 증가시키면 된다.
- 내가 놓친 것: 양쪽 끝에 있는 풍선은 반드시 살아남는다는 사실. 중앙에서 출발할 필요 없이 양 끝에서 동시에 출발하면 됨.
- **값 갱신만 잘하면 반복문 하나로 끝나는 문제!**

```python
def solution(a):
    answer = 2

    left = a[0]
    right = a[-1]

    for i in range(len(a)):
        if a[i] < left:
            left = a[i]
            answer += 1
        if a[-1 - i] < right:
            right = a[-1 - i]
            answer += 1

    return answer if left != right else answer - 1
```

통과 (0.01ms, 10.2MB)
통과 (0.00ms, 10.1MB)
통과 (0.10ms, 10.3MB)
통과 (10.07ms, 14.2MB)
통과 (50.60ms, 31.6MB)
통과 (103.40ms, 42.6MB)
통과 (180.68ms, 53.4MB)
통과 (106.33ms, 53.3MB)
통과 (105.50ms, 53.3MB)
통과 (125.78ms, 53.4MB)
통과 (181.31ms, 53.4MB)
통과 (139.36ms, 53.3MB)
통과 (140.70ms, 53.5MB)
통과 (156.14ms, 53.4MB)
통과 (191.41ms, 53.4MB)

Server health check

AWS ELB health check

- AWS ELB는 그 자체가 Health check 기능을 지원한다.
- 다음은 ELB의 기능을 정리한 것

3. 로드밸런서의 기능?

1) 워크로드를 가상 서버와 같은 다수의 컴퓨팅 리소스로 **분산**

2) 요청의 전체적인 흐름을 방해하지 않고 필요에 따라 로드 밸런서에서 **컴퓨팅 리소스를 추가 및 제거**

3) **health check(상태확인)**

로드밸런서가 트래픽을 보낼 대상(인스턴스)의 상태확인

로드 밸런서가 정상적인 대상에만 요청을 보내도록 컴퓨팅 리소스의 상태를 모니터링하는 상태 확인

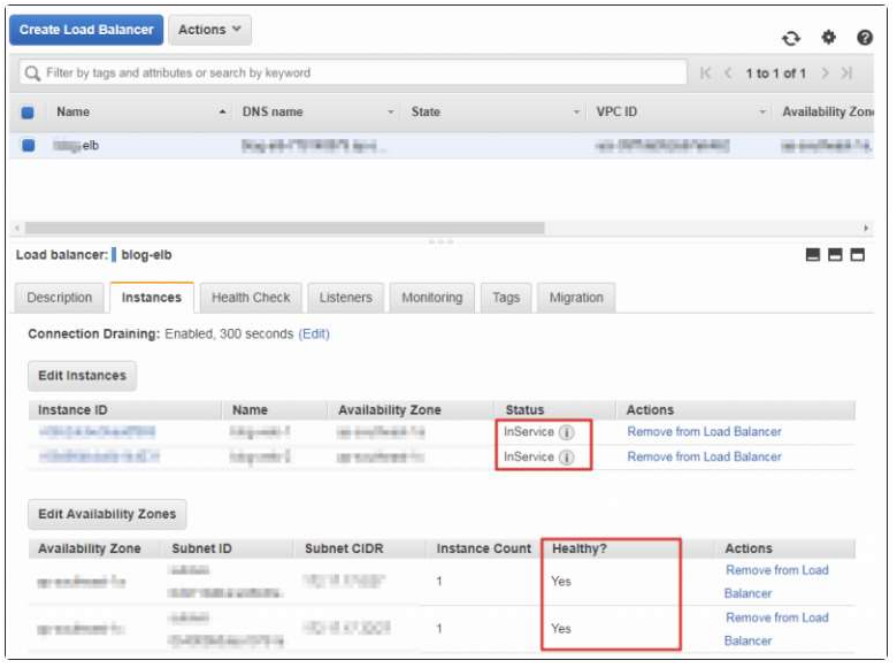
비정상적인 인스턴스를 감지하고 정상적인 인스턴스에만 트래픽을 라우팅

4) 컴퓨팅 리소스가 주요 작업에 집중할 수 있도록 암호화 및 복호화 작업을 로드밸런스로 오프로드

AWS ELB health check 확인 양상

Health Check (상태 검사)

- 직접 트래픽을 발생시켜 **인스턴스가 살아있는지 체크**하는 기능이다.
- 타겟 그룹에 대한 헬스 체크를 통해 현재 **정상적으로 작동하는 인스턴스로만 트래픽을 분배**한다.
- 인스턴스의 상태를 자동으로 감지해서 **오류가 있는 시스템은 배제** → 인스턴스가 회복되면 LB가 자동으로 감지하여 인스턴스에 트래픽을 보내준다.
  - 이를 통해 장애가 전파되는것을 방지하여 고가용성을 확보할 수 있다..
  - 상태 확인 개선을 통해 상세한 오류 코드를 구성할 수 있다.
  - 새로운 지표로 EC2 인스턴스에서 실행되는 각 서비스의 트래픽을 파악할 수 있다.
- 헬스 체크는 두가지 상태로 나뉘어진다
  - InService(서비스 살음) / OutofService(서비스 죽음)
- 제한 시간이나 간격, 성공 판단 횟수 등을 정의할 수 있다.



AWS ELB 동작 방식

- ELB 동작에 Health check 기능이 그냥 포함되어 있음

4. 로드밸런서 동작방식

1) 하나이상의 '**리스너**'를 지정해서 클라이언트로부터 들어오는 트래픽을 허용

* **Listener(리스너)** : **요청을 듣고 분배하는 역할 (**연결 요청을 확인하는 프로세스)

클라이언트와 로드 밸런서 간의 연결을 위한 프로토콜 및 포트 번호

사용자가 구성한 프로토콜 및 포트 번호에 매칭된 인스턴스에 요청을 전달하도록 함.

2) 들어온 요청을 하나이상의 가용영역에 있는 타겟대상(EC인스턴스)으로 보냄(라우팅함)

3) 로드밸런서에 등록된 대상의 health check

4) 타겟대상이 비정상(unhealthy)으로 감지되면, 해당 타겟대상으로 트래픽 보내는거 중단

5) 타겟대상이 정상(health)y으로 감지되면, 다시 타겟대상으로 트래픽 보냄

ELB는 서버가 요청을 받을 수 있을때만 요청을 전해준다!

ELB에서는 서버가 정상적으로 활동 중인지 확인을 위해서 Health Check 과정을 거치게 됩니다. 예를 들어 서버에서 GET/health라는 요청에 대해서 상태코드 200반응을 응답하도록 설정해 두고, ELB에 Health Check 경로를 GET/health로 등록해두면 ELB는 주기적으로 서버들에게 GET/health 요청을 보내게 됩니다. 이때 예상과 다른 응답이 오면 해당 서버가 정상적인 반응이 올 때까지 요청을 전달하지 않습니다.

물론, 검사 주기 및 몇 번이나 연속으로 비정상 코드 응답을 받을 때 비정상 상태로 판단할지 등을 설정할 수 있습니다.(정상에 대한 설정도 가능합니다)

AWS ELB health check 설정하는 법

- 이렇게 4단계: 라우팅 구성에서 상태 검사에 경로를 설정할 수 있게 해두었다.
- 샘플 코드에서 /health로 들어오는 요청에 대해서는 상태코드 200을 반환하도록 해놓았기 때문

AWS ELB health check 설정

- 상태 설정에 대한 고급 설정을 할 수 있음
- 정상 임계값과 비정상 임계값을 설정할 수 있음
- 제한 시간 내에 응답이 안 오면 실패처리하고 이것이 누적이 되면 unhealthy로 처리해서 요청을 보내지 않음

Cassandra 특징

Cassandra partition key, cluster key

- Primary key에 해당하는 column 이 “key”에 대응, 나머지 column 이 “value”에 대응한다.
- Primary key에는 partition key와 cluster key가 있다.
- Partition key는 primary key의 첫 번째 부분
- Cluster key는 partition key를 제외한 나머지 부분
- Partition key는 여러 column을 튜플로 묶어서 사용할 수 있다.
- Cluster key도 여러 column을 사용할 수 있는데, 여러 column으로 구성하면, 첫 번째 열로 일차 정렬, 두 번째 열로 이차 정렬.. 순차적으로 정렬된다.

```
CREATE TABLE test_keyspace.test_table_ex1 (
    code text,
    location text,
    sequence text,
    description text,
    PRIMARY KEY (code, location, name)
);

CREATE TABLE test_keyspace.test_table_ex2 (
    code text,
    location text,
    sequence text,
    description text,
    PRIMARY KEY ((code, id), location, name)
);
```

CQL 예시

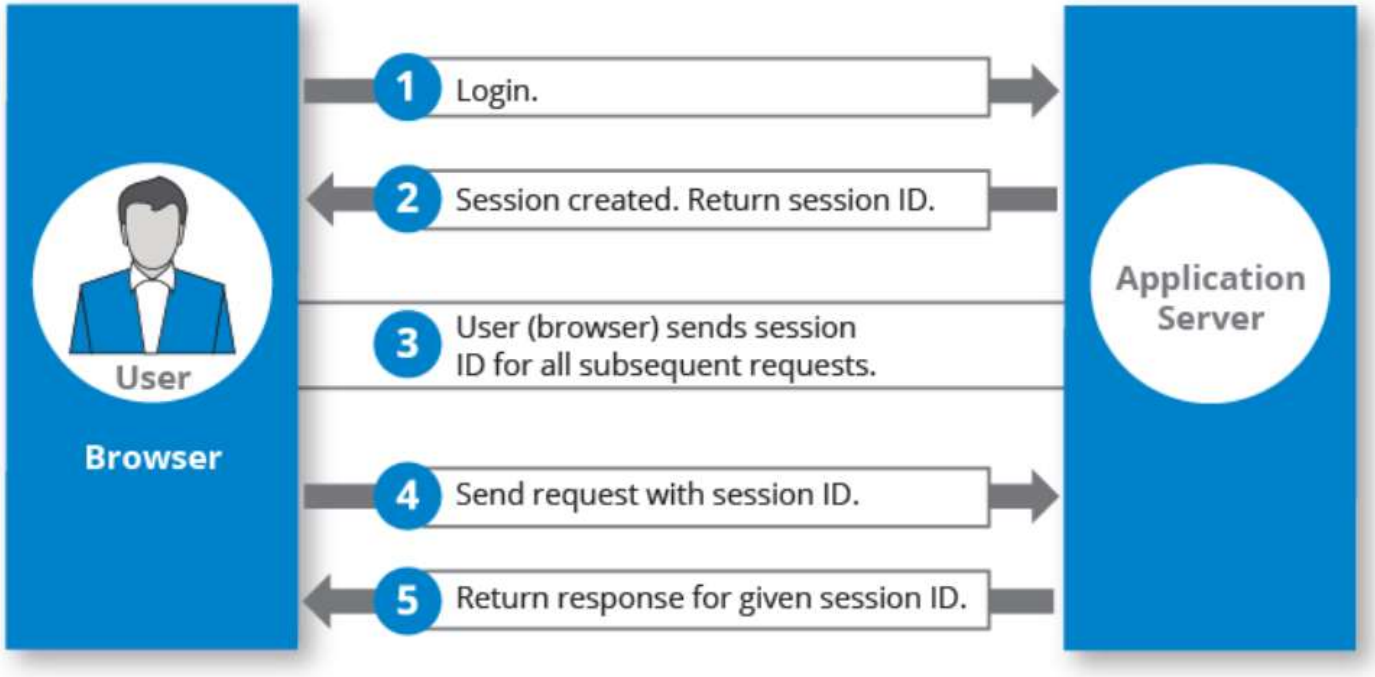
Cassandra partitioning

- 같은 partition key를 가진 데이터들을 파티션이라고 부른다. partition key는 해시 알고리즘에 의해 특정 토큰으로 변환되고, 토큰을 기준으로 클러스터 내 특정 노드에 데이터를 위치시킨다.
- 같은 partition key를 가지는 모든 row들은 같은 디스크에 저장된다.
- 하나의 partition에 2억 개의 column을 저장할 수 있다.
- 하나의 테이블에 10 개의 column 이 있다면, 파티션 당 200만 row를 저장할 수 있다. 하지만 row가 10만이 넘어가면 read 성능이 떨어진다.

JWT, JWE, JWS, JWK

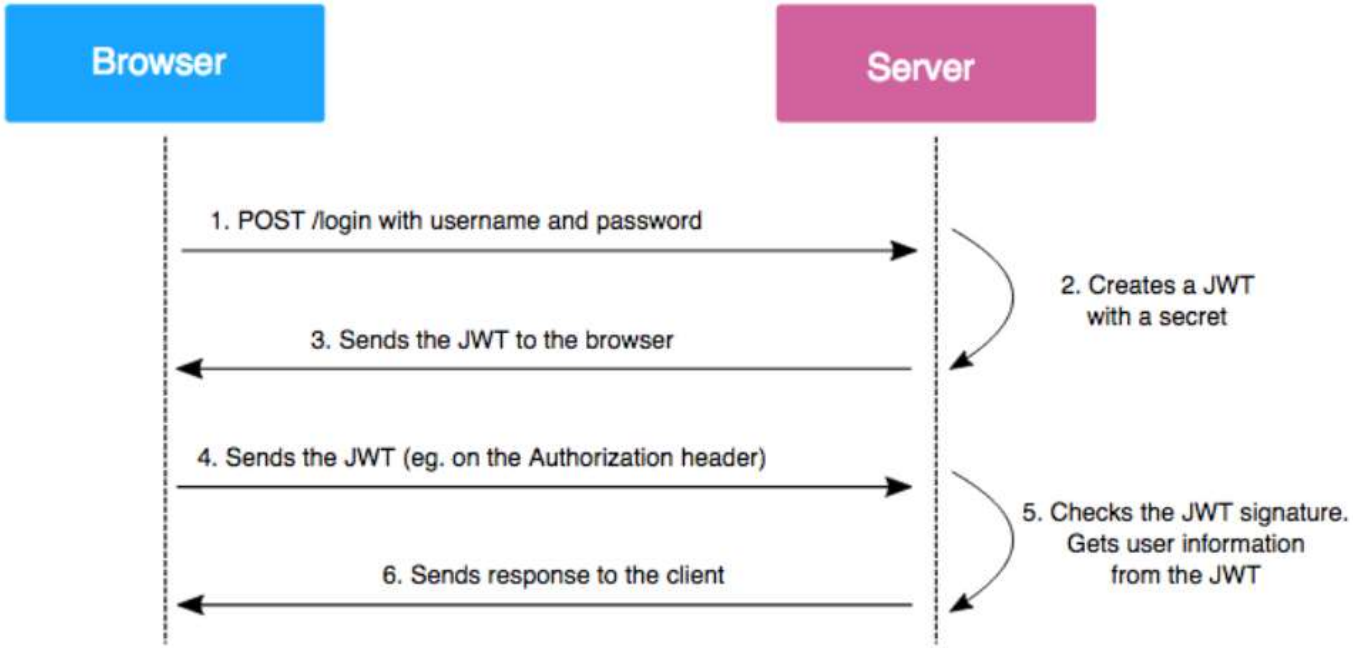
기존 session 인증 방식

- 기존의 세션 로그인 방식
- 로그인을 하면 서버에서 Session Id가 생성되어 Front End에 쿠키로 전달되어 저장된다.



JWT 인증 방식

- 서버에서 세션이 아닌 JWT를 생성하여 브라우저에 보내고 브라우저에는 이것을 저장했다가 이후 요청에 JWT를 함께 날려보낸다.
- 이것만 보면 세션과 동일해 보이지만 세션과 큰 차이가 있다.



JWT에서 내용 저장은 클라이언트에

- 서버에 정보를 저장하던 세션과는 달리 JWT는 클라이언트 측에 정보를 저장한다.
- 클라이언트 측에 정보를 저장한다는 것이 보안적으로 염려스러운 부분이 있다. 그래서 JWT는 인증에 필요한 정보를 암호화해서 보낸다.
- JSON 데이터를 Base64 URL-safe Encode 방식으로 직렬화한다.
- 토큰 내부에는 위변조 방지를 위해 개인키를 통한 전자서명도 있다.
- 사용자가 JWT를 서버에 보내면 서버는 이 서명을 검증하는 과정을 거쳐서 검증이 완료되면 해당하는 응답을 전송해준다.

JWT 구성요소(decoded)

- JWT는 Header, Payload, Signature로 구성된다.
- Header
  - alg: 해싱에 사용될 알고리즘
  - typ: 토큰의 타입
- Payload: 토큰에 담을 정보
  - 고유 ID 값
  - 유효기간
- Signature: Header와 Payload를 더해서 비밀키로 Hashing하여 생성한다. 토큰의 위조 여부를 판단할 때 사용한다.

<Header>

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

<Payload>

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

<Signature>

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) secret base64 encoded
```

JWT 구성요소(encoded)

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
zdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJ
SMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

- 위 그림은 인코딩된 JWT 데이터를 표현한 것이다.
- JWT는 구분자 '.'를 사용하여 세가지 부분으로 구분된다. 앞에서부터 Header, Payload, Signature를 의미한다.
- Header, Payload, Signature는 Base64 URL safe Encode 방식으로 암호화되어 전송된다.

```
JWT = B64(Header).B64(Payload).B64(Sig)
```

Signature의 생성 방식

```
Sig = ECDSA(SHA256(B64(Header).B64(Payload)), PrivateKey)
```

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) ☐ secret base64 encoded
```

- 전자서명은 base64UrlEncode 방식으로 각각 암호화된 Header와 Payload를 합쳐서 비밀키로 해싱해서 생성한다.
- Header와 Payload는 단순 인코딩된 값이라서 제 3자가 복호화 할 수 있지만, Signature를 생성할 때는 비밀키를 넣어서 암호화하기 때문에 제 3자가 복호화할 수 없다.
- Signature에는 비대칭 암호화 알고리즘을 사용한다. 암호화에는 비밀키를 사용하고 복호화에는 공개키를 사용한다.
- 그래서 Signature는 Token의 위조 여부를 판단하는 데 사용된다.

JWT의 장점

- 토큰에 기본 정보, 토큰을 검증하는 서명이 자체적으로 포함되어 있다.
- 인증 정보를 저장하기 위한 별도의 저장소가 필요 없다. 서버는 무상태를 유지해도 된다.
- 토큰을 기반으로 다른 로그인 시스템에 접근할 수도 있다.
- 규격이 정해져 있기 때문에 다양한 환경(웹, 모바일)에서 사용할 수 있다.

JWT의 단점

- 쿠키 세션과 다르게 JWT는 토큰의 Byte가 큰 편이라 네트워크에 부하가 많이 걸린다.
- Payload 자체는 암호화할 수 없어서 고객의 중요한 정보는 담으면 안 된다.
- 토큰은 한 번 발급되면 만료시간이 될 때까지 계속 유효하다. 그래서 탈취 당하면 답이 없음. 서버에서는 토큰에 대한 정보를 들고 있지 않으므로 특정 토큰의 접속을 강제로 만료할 수 없다.

JWT 보안 전략

- JWT 단점을 보완하기 위해 보안 전략이 필요하다.

1. 짧은 만료기한 설정 – 사용자가 자주 로그인해야 하는 단점이 있다.
2. Sliding session – 서비스를 지속적으로 사용하는 사용자의 토큰 만료기한을 늘려주는 방법을 사용함
3. Refresh Token – Access Token과 Refresh Token을 구분하여 사용한다. Access Token이 만료되면 Refresh Token을 사용하여 Access Token의 재발급을 요청한다. Refresh Token을 별도의 저장소에 저장해야 하므로 JWT의 메리트를 다 가져갈 수 없음.

JWS와 JWE 그리고 JOSE

- 사실 JWT는 인터페이스 역할의 추상적 개념이고 실제 구현은 JWS와 JWE로 나뉜다.
- JWS(JSON Web Signature)는 디지털 서명을 사용하는 방식이고, claim의 내용이 노출되지만, 클라이언트가 Claim의 데이터를 사용할 수 있다. 보편적으로 JWT라고 하면 JWS를 의미한다.
- JWE(JSON Web Encryption)은 Claim 자체를 암호화해서 클라이언트가 내용을 파악할 수 없다. 보안이 더 뛰어나지만, 클라이언트에서 활용성이 떨어지므로 잘 사용하지 않는 방식
- JOSE는 JWS와 JWE 둘 다 사용하는 방식

JWK

- JSON Web Key는 JWT의 public key를 표현하기 위한 JSON 양식
- AWS Cognito, Auth0 등 서비스에서 JWT 서명에 사용되는 public key를 제공하기 위해 사용함. 서비스에서 정의된 URL에 접근하면 JWK를 다운로드 할 수 있다.
- 서버에서 토큰을 발급하고 나서 해당 토큰이 유효한지 검사할 때 JWK를 사용해서 서명의 일치 여부를 판단한다.