

## [ SYSTEM DESIGN ]

### I. system design interview guide

#### 1. requirements clarifications (요구사항 구체화)

- a. 문제의 범위에 대해 명확히 질문하는 것이 좋음 - 정답이 하나도 없는 개방형 문제이기 때문
- b. 시스템 최종목표를 정의하는데 충분한 시간을 보내는 것이 중요
- c. 시스템의 어떤 부분에 집중해야 하는지 명확히 해야 함 - 큰 시스템을 설계하는데 30분 밖에 걸리지 않기 때문
- d. twitter의 경우
  - i. 서비스 사용자가 트윗을 게시하고 다른 사람을 팔로우 할 수 있나요?
  - ii. 또한 사용자 타임라인을 생성하고 표시하도록 설계해야 하나요?
  - iii. 트윗에 사진과 동영상이 포함되나요?
  - iv. 우리는 백엔드에만 집중하고 있나요? 아니면 프론트엔드도 개발하고 있나요?
  - v. 사용자가 트윗을 검색할 수 있나요?
  - vi. 인기 급상승 주제를 표시해야 하나요?
  - vii. 새로운 혹은 중요한 트윗을 위한 push notification이 있나요?
  - viii. 이런 모든 질문은 최종 디자인이 어떻게 보일지를 결정함

#### 2. system interface definition (시스템 인터페이스 정의)

- a. 예상되는 api를 정의 - 요구 사항이 잘못되지 않았는지 확인가능
- b. 요구사항을 완성한 후에는 항상 시스템 api 를 정의 하는 것이 좋음
- c. 시스템에서 예상되는 내용을 명시
- d. twitter의 경우
  - i. postTweet ( user\_id, tweet\_date, tweet\_location, user\_location, timestamp, ... )
  - ii. generateTimeline ( user\_id, current\_time, user\_location, ... )
  - iii. markTweetFavorite ( user\_id, tweet\_id, timestamp, ... )

#### 3. back-of-the-envelope estimation (시스템규모 추정)

- a. 시스템의 규모를 추정하는 것이 좋은 이유 : 스케일링, 파티셔닝, 로드밸런싱 및 캐싱에 초점을 맞출 때 도움이 됨
- b. tweeter를 설계한다고 가정
  - i. 예상 규모는 어떤가요? (새 트윗수, 트윗보기수, 초당 타임 라인 생성 수등)
  - ii. 얼마나 많은 스토리지가 필요한가요? (트윗에 사진과 비디오를 첨부 할 수 있다면 규모는 많이 달라짐)
  - iii. 네트워크 대역폭은 얼마나 사용하게 될까요? (서버 간 트래픽 관리 및 로드 균형 조정 방법을 결정하는데 중요)

#### 4. defining data model (데이터 모델 정의)

- a. 데이터 모델을 조기에 정의 하면 시스템의 여러 구성요소간에 데이터가 흐르는 방식이 명확해짐 - 추후 데이터 파티셔닝 및 관리를 안내
- b. 응시자는 다음을 식별 할 수 있어야 함
  - i. 시스템의 다양한 엔티티
  - ii. 서로 상호 작용하는 방법
  - iii. 스토리지, 운송, 암호화 같은 데이터 관리의 다른 측면을 식별 가능해야 함
- c. twitter의 경우
  - i. User : UserID, Name, Email, DateOfBirth, CreationDate, LastLogin 등
  - ii. Tweet : TweetID, Description, TweetLocation, NumberOfLikes, TimeStamp 등
  - iii. UserFollow : UserOneID, UserTwoID
  - iv. FavoriteTweets : UserID, TweetID, TimeStamp
- d. 예상질문
  - i. 어떤 데이터베이스 시스템을 사용해야 하나요?
  - ii. Cassandra와 같은 NoSQL이 우리의 요구에 가장 잘 맞습니까? 아니면 MySQL 같은 솔루션을 사용해야 하나요?
  - iii. 사진과 비디오를 저장하기 위해 어떤 종류의 블록 스토리지를 사용해야 하나요?

#### 5. high-level design (고급 디자인)

- a. 시스템의 핵심 구성 요소를 나타내는 5개 내외의 상자로 블록 다이어그램을 그림

- b. 실제 문제를 end to end로 해결하는데 필요한 충분한 구성 요소를  
식별해야 함
- c. tweeter의 경우
  - i. 상위레벨에서는 트래픽 분배를 위한 로드 밸런서, 모든 읽기/쓰기  
요청을 처리하기 위한 여러 애플리케이션 서버가 필요
  - ii. 쓰기와 비교하여 더 많은 읽기 트래픽이 있다고 가정하면 이러한  
시나리오를 처리하기 위해 별도의 서버를 갖도록 결정 할 수 있음
  - iii. 백엔드는 모든 트윗을 저장하고 많은 수의 읽기를 지원할 수 있는  
효율적인 데이터베이스가 필요
  - iv. 또한 사진과 비디오를 저장하기 위한 분산 파일 저장 시스템이  
필요 (1.jpg)

#### 6. detailed design (세부설계)

- a. 두세가지 구성요소를 더 깊게 파고들것
- b. 면접관의 피드백은 시스템의 어떤 부분에 대해 더 논의 해야 할지를  
안내함
- c. 서로 다른 접근법, 그것들의 장단점을 제시 할 수 있어야 함
- d. 본인이 선택한 접근법을 선호하는 이유를 설명해야 함
- e. 단일 답변이 없다는 것을 기억할것
- f. 시스템 제약 조건을 염두에 두고 다른 옵션 사이의 균형을 고려하는 것이  
중요함
- g. 예상질문
  - i. 방대한 양의 데이터를 저장하게 될텐데 데이터를 여러  
데이터베이스에 배포하기 위해 어떻게 데이터를 분할 해야  
할까요?
  - ii. 사용자의 모든 데이터를 동일한 데이터베이스에 저장해야 할까요?  
어떤 문제가 발생할 수 있을까요?
  - iii. 트윗을 많이 하거나 많은 사람들이 따르는 인기 사용자의 경우  
어떻게 처리하는 것이 좋을까요?
  - iv. 사용자의 타임라인에는 가장 최근의 관련 트윗이 포함 되어 있어야  
하는데 최신 트윗을 스캔하는데 최적화 된 방식으로 데이터를  
저장해야 할까요?
  - v. 캐시 속도를 얼마나 많이, 어느 계층에 도입해야 할까요?
  - vi. 더 나은 로드 밸런싱이 필요한 구성 요소는 무엇일까요?

7. identifying and resolving bottlenecks (병목현상 식별 및 해결)

a. 가능한 많은 병목현상과 이를 완화하기 위한 다양한 접근 방식에 대해

논의 해야 함

b. 예상질문

i. 시스템에 단일 장애지점이 있나요? 완화하기 위해 무엇을 하고  
있나요?

ii. 서버를 몇대 잃어도 사용자에게 서비스를 제공 할 수 있을 만큼  
데이터 복제본이 충분한가요?

iii. 몇 개의 장애로 인해 전체 시스템이 종료되지 않도록 실행중인  
다른 서비스의 사본이 충분한가요?

iv. 서비스 성능은 어떻게 모니터링 하나요? 중요한 구성요소가  
실패하거나 성능이 저하 될때마다 경고를 받나요?

8. summary (요약)

a. 인터뷰 중에 준비 및 구성하는 것이 설계 인터뷰를 성공하는 열쇠임

b. 위에서 언급한 단계는 시스템을 설계하는 동안 모든 다른 측면을 추적하고  
추적하도록 안내 해야함

## II. Designing a URL Shortening service like TinyURL

1. TinyURL - <http://goo.gl/r1lit3u> → <http://www.mnori.com/game/1.html>

2. URL 단축이 왜 필요한가요?

- a. 표시, 인쇄, 메시지, 트윗시 공간절약
- b. 사용자 입력 오류 줄임
- c. 통계값 얻음

3. requirements and goals of the system

a. functional requirements

- i. TinyURL 생성
- ii. TinyURL 입력 시 원래 URL로 redirection
- iii. 고객주문형 TinyURL (ex) <http://goog.le/wooick>
- iv. 기한만료가 있어야 함

b. non-functional requirements

- i. availability (가용성) 이 높아야 함 - 서비스가 중단되면 모든 URL redirection이 실패함
- ii. 대기시간 최소
- iii. real-time URL redirection
- iv. 생성전 TinyURL 은 예측할 수 없어야 함

c. extended requirements

- i. analytics - 얼마나 많이 redirection이 일어났나요?
- ii. 다른 서비스가 rest api 를 통해 access 할 수 있어야 함

4. capacity estimation and constraints

a. Spread Sheet 참고

5. system apis - soap, rest api

a. createURL ( api\_dev\_key, original\_url, custom\_alias=None, user\_name=None, expire\_date=None )

- i. api\_dev\_key (string) : account에 등록된 api 개발자 키, 사용량 제한에도 사용

- ii. custom\_alias (string) : 고객이 직접 url명을 정함

- iii. user\_name (string) : encoding시 사용

- iv. return (string) : TinyURL/error code

b. deleteURL ( api\_dev\_key, tiny\_url )

- i. return (string) : success message/error code

## 6. database design

### a. 고려사항

- i. 1B records
- ii. record size is small (less than 1k)
- iii. read-heavy

### b. database schema

#### i. Url

- TinyUrl : varchar(6), PK
- CustomAlias : varchar(6)
- OriginalURL : varchar(512)
- ExpireDate : datetime
- UserID : int, FK

#### ii. User

- UserID : int, PK
- Name : varchar(20)
- Email : varchar(32)
- DateOfBirth : datetime
- LastLogin : datetime

### c. 어떤 DB를 사용해야 할까요?

- i. record량이 방대하고 relationship이 적음 - NoSQL (확장위주)

## 7. basic system design and algorithm

### a. real-time url encoding

#### i. encoding

- a-z + 0-9 = 36  $\rightarrow$  base36
- base36 + A-Z = 62  $\rightarrow$  base62
- base62 + '.' + '=' = 64  $\rightarrow$  base64

#### ii. length

- 6 letter :  $64^6 \approx 68.7B$
- 8 letter :  $64^8 \approx 281T$  (Trillion)

#### iii. workflow ([2\\_x.jpg](#))

- MD5 사용하여 128bit hash value 생성
- base64 encoding

- 21 letter 생성 - 128 / 6 (base64 는 원데이터를 6bit 씩  
분할) = 21.xxx
- 첫 6자 (혹은 8자) 를 사용
- 키가 중복하는 경우
  - ✓ 인코딩에 사용되지 않은 다른 글자 사용
  - ✓ 몇몇 글자들의 위치를 바꿈

iv. 단점

- 여러 사용자가 동일 URL입력시 동일한 결과값 반환 -  
허용불가
- URL의 일부가 URL encoding 된 경우

v. 해결방안

- URL 생성 시 sequence 번호 추가 - sequence 번호가  
증가하면 서비스 성능이 느려짐
- URL 생성 시 UserID 추가

b. generating keys offline (오프라인으로 키생성)

- i. KGS (Key Generation Service, 독립형 키 생성 서비스) : 6개의  
문자열을 임의로 생성 후 데이터베이스에 저장, 모든 키는  
고유해야 함

ii. TinyURL 생성 시 KGS로 부터 생성한 키 중 하나를 씀

iii. 장점

- 인코딩 필요 없음
- 중복/충돌 없음

iv. (동시성) 문제를 일으킬 수 있나요?

- 키를 사용하면 표시

- ✓ 두개 (미사용, 사용) 테이블 사용
- ✓ KGS가 키를 제공하자 마자 사용된 키 테이블로  
이동
- ✓ KGS는 키를 메모리에 보관 - 키 제공 속도 빠름

• 단점

- ✓ KGS가 죽으면 미사용키는 모두 손실
- ✓ 여러서버에 동일한 키를 제공하면 안됨 - 잠금설정  
필요

v. Key-DB의 크기는 얼마나 되나요?

- 하나의 문자를 저장시 1byte 소요라 가정

$$\checkmark 6(\text{문자수}) * 68.7\text{B}(\text{key}) = 412\text{GB}$$

vi. KGS는 단일 장애지점 인가요?

- 네, 이를 해결하기 위해 replica of KGS 을 만듬
- 기본서버가 다운될때 마다 대기 서버가 키를 제공

vii. 각각의 application server 가 key-DB이 일부 키를 캐시 할 수 있나요?

- 네, 하지만 캐쉬된 모든 키를 사용하기 전에 applicaton 서버가 종료되면 해당 키가 손실

viii. 키 조회는 어떻게 하나요?

- database, key-value stores 에서 키를 조회
- original url 얻음 - HTTP302 redirection 을 통해 original url로 이동
- original url 못 얻음 - HTTP404 Not Found 리턴

ix. 맞춤 별칭에 크기 제한을 적용해야 하나요?

- 크기 제한이 좋음 - 일관된 URL database 확보

## 8. data partitioning and replication

### a. range based partitioning (범위 기반 분할)

- 해시 키의 첫 글자 기준으로 url을 별도 분할 영역에 저장  
(ex) a로 시작 -> partition1, b로 시작 -> partition2
- static partitioning scheme(정적 분할 구성표) - 예측 가능 방식으로 쓰기/읽기 하기 위함
- 단점 - 서버간의 불균형 (ex) e로 시작하는 문자가 너무 많음

### b. hash based partitioning (해시 기반 파티셔닝)

- hash function을 사용 (ex) key % number of server 를 서버 index에 mapping
- 단점 - 서버간의 불균형
- 해결방안 - consistent hashing (일관된 해싱)

## 9. cache

### a. Memcached 같은 상용 솔루션 사용 가능 - hash와 함께 전체 url 저장

b. 얼마나 많은 캐시를 가져야 하나요?

- 20% (80:20 rule) 로 시작, 클라이언트 사용패턴에 따라 조절

- c. cache evitction policy (캐시제거정책) : LRU (Least Recently Used, 가장 오래된 item 부터 제거)
- d. efficiency(효율성) 을 높이기 위해 복제하여 로드를 분산
- e. 각 캐시 복제본은 어떻게 업데이트 할 수 있나요?
  - i. cache miss (캐시누락) -> 캐시 업데이트 (새 항목을 DB로부터 가져와 캐시 복제본들에게 업데이트)

## 10. load balancer (LB)

- a. 위치
  - i. 클라이언트와 웹서버 사이
  - ii. 웹서버와 응용프로그램계층 사이
  - iii. 응용프로그램계층과 DB(or cache) 사이
- b. 초기
  - i. round-robin 사용 : simple, overhead가 발생하지 않음
  - ii. 단점 : 서버로드가 고려되지 않음
  - iii. 해결안 : intelligent LB solution (주기적으로 서버노드를 체크 한 후 트래픽에 반영)

## 11. purging or DB cleanup

- a. 사용자 지정 만료시간에 도달하면 링크는 어떻게 되나요?
  - i. 만료된 링크 제거를 위해 database를 검색하면 많은 부하가 걸림
  - ii. 만료된 링크를 천천히 제거 - 만료된 링크 요청시 링크 삭제 후 에러 리턴
  - iii. 스토리지 및 캐시에서 만료된 링크 제거를 위해 정리 서비스를 정기적으로 실행 할 수 있음 - 서비스는 매우 가벼워야 하며 사용자 트래픽이 적을 때만 실행
- b. 6개월 동안 방문하지 않은 링크를 제거해야 하나요?
  - i. 스토리지 가격에 따라 기간을 늘릴 수 있음

## 12. telemetry (원격 측정)

- a. 예상질문
  - i. TinyURL을 몇번 사용했는지, 사용자의 위치가 어디인지 확인 할 수 있나요?
  - ii. 통계자료를 어디에 저장 하나요?
- b. 추적할 가치가 있는 통계 : 방문자 국가, 액서스 날짜 및 시간, 사용빈도, 브라우저 또는 페이지에 액서스한 플랫폼 (OS)

13. security and permissions (보안 및 권한)

- a. 사용자가 개인 url을 만들거나 특정 사용자 집합만 url에 액세스하도록 허용할 수 있나요?
  - i. url과 함께 권한수준 (공개/개인) 저장
  - ii. 개인 url을 볼 수 있는 UserID를 저장하기 위한 추가 테이블 작성

### III. designing pastebin

1. pastebin
  - a. 해당 서비스에 데이터(text, image)를 저장하면 url로 리턴 - 해당 url로 저장된 text 접근
  - b. url을 이용하여 데이터를 쉽고 빠르게 공유
2. requirements and goals of the system
  - a. functional requirements
    - i. text를 저장하면 url을 리턴 (기능축소)
    - ii. 자동만료, 만료시간 선택
    - iii. custom url support
  - b. non-functional requirements
    - i. reliability 높음 - upload된 데이터는 손실 될 수 없음
    - ii. availability 높음- url 요청시 해당 text를 반드시 리턴
    - iii. real-time access with minimum latency
    - iv. 시스템의 생성한 url은 추측 불가능해야 함
  - c. extended requirements
    - i. analytic
3. design considerations
  - a. text 양을 제한 가능한가요? 네, 10mb정도가 적당
  - b. custom url에 글자수를 제한 해야 하나요?
    - i. 자유롭게 가능
    - ii. 글자수를 시스템 생성 url수와 동일하게 진행하는 것이 좋음 - DB일관성 유지
4. capacity estimation and constraints
  - a. Spread Sheet 참고
5. system apis - soap, rest api
  - a. addPaste (api\_dev\_key, paste\_data, custom\_url=none, user\_name=none, expire\_date=none)
    - i. return (string) : paste\_url / error code
  - b. getPaste (api\_dev\_key, paste\_url)
    - i. return (string) : paste\_date / error code
  - c. deletePaste (api\_dev\_key, paste\_url)
    - i. return (string) : success / error code

## 6. database design

### a. considerations

- i. record수가 아주 많음
- ii. metadata size는 작음 (1kb 미만)
- iii. contents (text) 크기는 평균 10kb, 최대 10mb
- iv. 사용량이 많음

### b. type

- i. Metadata Storage : 수직확장(관계많음, 용량적음), RDBMS  
(ex) MySQL, MS-SQL, Oracle
- ii. Contents Storage : 수평확장(관계적음, 용량 많음), Object Storage (ex) Amazon S3

### c. table

#### i. Paste

- URL: varchar(6), PK
- ContentsPath: varchar(512)
- UrlAlias : varchar(6)
- ExpireDate : datetime,
- UserID : int, FK

#### ii. User

- UserID : int, PK
- Name : varchar(20)
- Email : varchar(32)
- DateOfBirth : datetime
- LastLogin : datetime

## 7. high level design (3.1.jpg)

### a. application layer

- i. 읽기/쓰기 요청 처리
- ii. 백엔드 저장소와 통신

### b. storage layer

- i. metadata storage
- ii. contents storage

## 8. component design (3.2.jpg)

### a. application layer

- i. 쓰기요청 받음 -> 6자리 키생성 -> 컨텐츠와 키를 DB에 저장 -> 키 or 오류코드 반환
  - 새로운 키가 기존키와 중복될 경우 : 키를 다시 생성, 성공할 때까지 시도 - KGS로 단점 해결 가능
  - KGS (Key Generation Service)
    - ✓ 키를 미리 생성해서 DB Table에 넣음 - 필요할 때마다 꺼내 씀
    - ✓ 생성시간 단축, 중복/충돌 없음
  - CustomUrl 이 이미 DB에 있는 경우 오류코드 반환
  - 각 어플리케이션 서버가 Key\_DB의 일부를 Cache 할 수 있나요?
    - ✓ 가능하지만 해당 서버 종료시 Cache된 미사용키 손실
- ii. 읽기요청 받음 -> 키 검색 -> Uploaded Text or Error Code 리턴

#### b. datastore layer

- i. metadata storage : 수직확장 (관계많음, 용량적음), RDBMS Dynamo, Cassandra
- ii. contents storage : 수평확장 (관계적음, 용량많음), Object Storage (Amazon S3)

### 9. purge and DB cleanup

#### a. 만료된 컨텐츠 삭제

- i. (실시간 정리) : DB 사용량 증가 - 시스템속도 저하
- ii. (지연 정리)
  - 고객 요청 -> 만기된 record의 경우 error 리턴 후 삭제
  - 별도의 정리 서비스 실행 : 정기적, 서비스는 가벼워야 하고 트래픽이 적을때 실행
- iii. 기본 만료시간 설정, 만료된 URL키는 재사용 가능
- iv. 6개월 동안 사용되지 않은 URL이 있다면?
  - case by case
  - 유지 결정을 내리는 추세임 : 스토리지 비용 저렴

### 10. data partitioning and replication

#### a. data partitioning

- i. range based partitioning (범위기반분할)
    - URL 첫글자를 이용하여 저장할 파티션을 정함
    - 파티션이 불균형해짐 (ex) E로 시작하는 것이 많음
  - ii. hash based partitioning (해시기반분할)
    - hash function : key % number of server
    - 문제 : 확장불가, 파티션이 불균형해짐
    - 해결 : Consistent Hashing (일관된 해싱),  
repartitioning
  - b. replication
    - i. traffic 분산 : 시스템 응답속도 향상
    - ii. data 백업 : 원본이 손실한 경우 복사본으로 복원 가능
11. cache and load balancer
- a. cache
    - i. Memcache 같은 상용 솔루션 사용가능
    - ii. cache size : 20 % of total (based on 20:80 rule)
    - iii. cache evitction policy : LRU (Least Recently Used)
    - iv. cache server를 replication 해서 서버간 로드를 분산시키면  
(효율성) 이 높아짐
    - v. cache type
      - write through cache
        - ✓ hdd와 cache 모두 저장
        - ✓ 장점 : 일관된 sync
        - ✓ 단점 : write 느림
      - write around cache
        - ✓ hdd 만 저장, cache miss ( cache 에 원하는 데이터가 없음) 시에만 hdd에서 cache로 복사 후 서비스
        - ✓ 장점 : write 중간
        - ✓ 단점 : cache miss 일어날 경우만 read 느림
      - write back cache
        - ✓ cache만 저장, 정기적으로 cache update 본을 hdd에 sync
        - ✓ 장점 : write 빠름

✓ 단점 : 서버 다운시 데이터 훼손 ( cache에 있는 데이터가 hdd와 sync을 하지 못하고 증발)

b. load balancer 위치

- i. 클라이언트와 웹서버 사이
- ii. 웹서버와 어플리케이션서버 사이
- iii. 어플리케이션서버와 DB or Cache 사이

12. security and permissions

a. 사용자가 개인 URL을 만들거나 특정 사용자 집합이 URL에 액세스  
하도록 허용할 수 있나요?

- i. 레코드 (튜플을) 공개/비공개로 설정 -> 비공개의 경우 별도의  
테이블을 만들어 해당 레코드에 access 가능한 UserID를 저장 ->  
사용자 권한이 없다면 오류코드 리턴 (ex) HTTP 401 Error

## IV. Designing Instagram

1. instagram
  - a. SNS (Social Networking Service)
  - b. photo, movie 공개/비공개 공유
  - c. 타 유저 follow 가능
  - d. 다른 SNS 서비스와 연동
2. requirements and goals of the system
  - a. functional requirements
    - i. photo/movie upload/view
    - ii. search
    - iii. 다른 유저 follow
    - iv. newsfeed support
  - b. non-functional requirements
    - i. high reliability : update된 데이터는 절대 손실 불가
    - ii. high availability : 끊김없는 서비스
    - iii. real-time response with minumum latency : 200ms 이내
  - c. extended requirements
    - i. tag 입력/검색
    - ii. 댓글달기
3. design considerations
  - a. 무제한 photo/movie 업로드 지원 : storage의 효율적인 운용 필요
  - b. 빠른 응답, 100% data (신뢰성) 제공
4. capacity estimate and constraints
  - a. Spread Sheet 참고
5. high level design ([8.jpg](#))
  - a. 사진/동영상 업로드 : Object Storage (ex) Amazon S3
  - b. 사진 보기 및 검색 : Metadata Storage, RDBMS (ex) MySQL, MS-SQL, Oracle
6. database schema
  - a. Photo
    - i. PhotoID : int, PK
    - ii. PhotoPath : varchar(256)

- iii. PhotoLatitude : int,
- iv. PhotoLongitude : int,
- v. UserLatitude : int,
- vi. UserLongitude : int,
- vii. UserID : int, FK
- b. User
  - i. UserID : int, PK
  - ii. Name : varchar(20)
  - iii. Email : varchar(32)
  - iv. DateOfBirth : datetime
  - v. LastLogin : datetime
- c. Follow
  - i. UserOneID : int
  - ii. UserTwoID : int
- d. 뉴스피드는 최근사진을 먼저 보여줌 : PhotoID에 CreationDate를 반영  
후 Index 처리

## 7. component design

- a. 서버가 쓰기요청을 받으면 읽기 요청을 제공하기 어려움 : 쓰기(upload)  
프로세스는 느린 프로세스이기 때문에 서버의 사용 가능한 연결을 모두  
사용할 수 있음
- b. 해결 : 읽기서버와 쓰기서버를 분리 - 서버를 분리하면 독립적으로  
확장/최적화 가능 ([9.jpg](#))
- c. reliability and redundancy
  - i. replication 필요 : 업로드된 사진은 절대 손상되선 안됨
  - ii. 서비스 redundancy (복제)
    - 성능향상 : 앞에 LB를 두어 Traffic 분산
    - availability 향상 : 서버들 중 일부가 다운되어도 서비스  
가능 ([10.jpg](#))

## 8. data partitioning - hash based sharding

- a. key : UserID ( 같은 유저 사진을 같은 shard에 배치)
- b. hash function : UserID % number of server
- c. PhotoID : 각 shard의 sequence no 이용
- d. DB 앞에 LB 설치 : DB간 robin / down-time 처리

e. shard는 2배 정도 용량으로 운용 : 성능과 확장성이 좋아짐

f. 확장계획

i. 서비스 초기, 하나의 서버에 다수의 DB 인스턴스 생성

ii. 추후 확장 시마다 각각의 DB 인스턴스를 다른 서버로 이주

g. 단점

i. hot user 문제 : 팔로워가 많은 user는 어떻게 처리하나요?

ii. storage의 불균일한 배포 : 특정 사용자의 사진이 많다면?

iii. 하나의 shard에 한 사용자의 사진을 모두 넣을 수 없다면?

9. ranking and news feed generation

a. 팔로우 유저들 사진 중 가장 인기/관련성 높은 사진들을 최근순으로 정렬

b. (ex) 100장의 최신 사진을 가져온다고 가정

i. follower list 받음

ii. follower들의 최근 100장의 사진의 metadata들을 받음

iii. 순위 알고리즘 통해 뉴스피드에 뿌려질 사진 100장을 결정하여 반환

iv. 단점 : 대기시간이 길어짐 (실시간 여러 테이블 쿼리 후 결과 병합, 순위지정을 수행) -> 뉴스피드 사전선정 (뉴스피드를 정기적으로 생성하여 별도의 테이블에 저장)

c. 뉴스피드 사전선정

i. 최신 뉴스피드가 필요할 때마다 이미 만들어 놓은 뉴스피드가 저장된 테이블 Query

ii. 뉴스피드 생성시 최근에 생성한 뉴스피드 시간을 찾고 그 시점부터 새 뉴스피드를 만들어 테이블에 저장

iii. 서버기준, 뉴스피드를 클라이언트에게 보내는 방법

- pull

- ✓ 클라이언트가 정기적으로 요청. 서버는 변동사항이 없더라도 항상 응답해야 함 - 빈응답이 너무 많음

- ✓ 서버가 변동사항이 생기더라도 클라이언트에게 바로 보낼 수 없음 - 클라이언트의 요청을 받을 때까지 대기

- push

- ✓ long polling

- ✓ 변동된 데이터를 클라이언트에게 즉시보냄

- ✓ 단점 : hot user에게는 load가 많이 걸림
- hybird
  - ✓ follower가 많은 사용자만 pull
  - ✓ 나머지는 push처리
- d. news feed creation with sharded data
  - i. Photoid에 사진생성시간을 넣는다면 최근 사진을 더 빨리 찾을 수 있음 (Photoid가 PK이기 때문)
  - ii. Photoid = epoch time + auto sequence
  - iii. Phoeoid의 크기
    - 50년간 쓴다 가정
    - $86400 * 365 * 50 = 1.57B$  (31bit 필요)

## 10. cache

- a. 대규모 사진 전송 서비스 필요
- b. 사진 cache를 지역별로 설치
- c. 20% of total (20:80 rule)
- d. type
  - i. write through cache
  - ii. write around cache
  - iii. write back cache

## 11. load balancing

- a. 위치
  - i. 클라이언트와 웹서버 사이
  - ii. 웹서버와 어플리케이션 서버 사이
  - iii. 어플리케이션 서버와 DB서버 사이
- b. 초기
  - i. round-robin method
    - 장점
      - ✓ simple, overhead가 생기지 않음
      - ✓ 다운된 서버에는 트래픽을 전송하지 않음
    - 문제 : 서버 load를 체크못함
    - 해결 : 서버의 load를 실시간 체크, load가 많은 서버에는 트래픽 전송 금지 - intelligent LB

## V. Designing Dropbox

1. why cloud storage?
  - a. availability : 언제, 어디서나, 어떤장치에서나 파일을 access 할 수 있어야 함
  - b. reliability and durability : uploaded 된 파일은 손실되지 않아야 함
  - c. scalability : 스토리지 무제한 공급
2. requirements and goals of the system
  - a. 모든장치에서 파일 업로드/다운로드/공유
  - b. 장치간 자동 동기화
  - c. 대용량 파일 지원 (최대 1GB)
  - d. ACID-ity 지원 - transaction이 안전하게 수행된다는 것을 보장
    - i. atomicity (원자성) : all or nothing
    - ii. consistency (일관성) : integer가 string이 될 수 없음
    - iii. isolation (격리성) : transaction 사이에 다른 작업이 끼어들 수없음
    - iv. durability (내구성) : 입금후 입금내역이 사라지면 안됨
  - e. 오프라인 편집 : 오프라인에서 파일 추가/삭제/수정 가능 (추후 온라인시 동기화)
  - f. 파일 버전관리 (data snapshot 지원)
3. design considerations
  - a. 사용량 엄청 많음 : 엄청난 횟수의 파일 읽기/쓰기
  - b. read/write ratio : 1:1
  - c. 시스템 내에서 파일은 chunk로 저장 (ex) 4mb
    - i. 전송실패/업데이트 한 chunk만 재전송
    - ii. 중복된 check 제거 : 대역폭과 스토리지 공간 절약
    - iii. 클라이언트에 metadata 사본 저장 - 데이터 교환량 절약
    - iv. 변경이 아주 작을 경우 (chunk를 보내기도 아까울 정도) : diff 만 upload
4. capacity estimations and constraints
  - a. spread sheet 참고
5. high level design ([11.jpg](#))
  - a. device 특정 폴더 지정 -> cloud와 sync

- b. 장치간 자동 동기화 : 한 장치에서 수정된 사항은 다른 모든 장치에도 자동 수정
- c. file, metadata, 파일공유자 저장 - 파일 업로드/다운로드 서버 (block server) 와 metadata server 가 필요하고 update 발생시 모든 client에서 file sync를 알리는 메커니즘이 필요
- d. component
  - i. block server : file upload/download 담당
  - ii. metadata server : 클라이언트에 있는 metadata를 최신상태로 유지
  - iii. sync server : 모든 클라이언트의 sync를 맞춤
  - iv. storage
    - cloud storage : object storage (ex) amazon s3
    - metadata storage : RDBMS (ex) MySQL, MS-SQL, Oracle

## 6. component design

- a. client
  - i. client application
    - sync된 folder 모니터링
    - remote cloud storage 와 sync
    - file upload/download/update
  - ii. file transfer
    - file을 chunk로 나눔 - update/전송실패된 chunk만 전송
    - chunk 크기 : 공간활용도와 IOPS (Input Output operations Per Second)를 최적화
    - chunk size, network bandwidth, average storage file size 등의 정보를 metadata에 저장
  - iii. client가 metadata 사본을 보관해야 하나요?
    - offline update 수행가능
    - remote metadata를 update 하기 위한 왕복시간 절약가능
  - iv. client가 update를 확인 할 수 있는 방법은 어떤것들이 있나요?
    - client가 주기적으로 체크

- ✓ 반영(서버->클라이언트) 지연 발생
- ✓ server는 대부분 빈응답 반환 : 대역폭 낭비 및 서버 사용량 늘림
  - http long polling
    - ✓ 서버는 요청을 열린상태로 유지
    - ✓ 새로운 정보가 있으면 즉시 클라이언트에게 응답
- v. client component ([12.jpg](#))
  - internal metadata DB
    - ✓ 파일시스템의 모든 파일, 청크, 버전 및 위치 추적
  - chunker
    - ✓ 파일 -> chunk로 나눔
    - ✓ chunk -> 파일 재구성
    - ✓ chunk algorithm : 파일의 수정부분만 감지 -> 해당부분만 cloud storage로 전송 - 대역폭/동기화 시간 절약
  - watcher
    - ✓ 로컬 작업영역(sync 폴더) 모니터링
    - ✓ 사용자가 파일/폴더 생성/삭제/업데이트시 indexer에게 알림
  - indexer
    - ✓ watcher로 부터 수신한 이벤트 처리
    - ✓ updated file chunk 정보를 metadata DB에 저장
    - ✓ 해당 chunk들을 cloud storage로 upload/download
    - ✓ remote sync service와 통신 후 update 사항을 다른 client device (ipad, mobile, pc, etc)에게 broadcast 후 metadata DB를 update
  - server가 느릴경우 client는 어떻게 처리되나요?
    - ✓ client는 재시도 간격을 기하 급수적으로 늘림
  - mobile device (무선인터넷 사용) 경우 서버의 변경사항을 즉시 sync 해야 하나요?

- ✓ 아니요. 사용자 요청시에만 (대역폭과 device 저장공간을 절약하기 위해)

- b. metadata database

- i. file/chunk, client 및 작업영역(sync 폴더)에 대한 version 및 metadata 유지관리
- ii. NoSQL (scalability 와 performance 를 위해 ACID-ity 을 미지원) 로 구현할 경우 ACID-ity를 프로그래밍으로 구현해야 함
- iii. 다음 객체에 대한 정보 저장 : chunk, file, client, device, workspace (sync folder)

- c. sync service

- i. client update 가 있을 경우 file update 처리 후 다른 client sync
- ii. client local(metadata)DB 와 server metadata DB 를 sync
- iii. client 가 일정시간 offline 였을 경우 online 되는 즉시 long polling 요청 (update를 하기 위함) -> sync service는 metadata DB에서 일관성 check 후 update 진행
- iv. optimization
  - client 와 cloud storage 간 더 적은 data가 전송되도록 설계 (ex) differencing algorithm 사용
  - 두 버전간의 차이(변경된 파일 부분) 만 전송
  - 서버에 이미 같은 hash를 가진 chunk가 있는 경우 (다른 사용자의 경우라도) upload 를 최소하고 기존 chunk 사용
  - 효율적이고 확장 가능한 sync protocol 제공
    - ✓ 통신 미들웨어 사용 (client와 sync server 사이)
    - ✓ 메세징 미들웨어는 pull/push 전략으로 많은 수의 client를 지원 할 수 있는 확장 가능한 message queuing 및 update notification을 제공해야 함
    - ✓ 여러 sync service 가 global request queue에서 요청을 수신할 수 있으며 통신 미들웨어는 밸런스를 조절 할 수 있음

- d. message queue service (13.jpg)

- i. messaging middleware

- ii. client 와 sync service 간에 async message based communication
  - iii. 분산 서버간에는 비동기 느슨한 연결 메시지 기반 통신을 함
  - iv. 두개의 queue 구현
    - request queue
      - ✓ global queue - 모든 클라이언트가 공유
      - ✓ client가 metadata DB update request를 request queue로 전송 -> sync service 는 metadata update
    - response queue
      - ✓ 각 client에게 update message 전달
      - ✓ client 가 수신한 message는 대기열에서 바로 삭제되므로 update된 message를 device 마다 공유할 필요가 있다면 별도의 응답 대기열을 만들면 됨
  - e. cloud/block storage ([14.jpg](#))
    - i. 사용자가 update한 file 저장
    - ii. client는 storage와 직접 상호작용 하여 storage에서 object를 송수신
7. file processing workflow
- a. 클라이언트1이 chunk를 cloud storage에 upload
  - b. 클라이언트2가 metaupdate를 update 하고 변경사항 commit
  - c. 클라이언트1이 확인받고 변경사항에 대한 알림을 클라이언트2와 3에게 보냄
  - d. 클라이언트 2, 3은 metadata 변경사항 수신 후 update된 chunk를 download
8. data deduplication
- a. data deduplication
    - i. storage 사용률을 높이기 위해 중복된 data 삭제
    - ii. network data transfer에서도 적용가능
  - b. post-process deduplication
    - i. 새 chunk 저장 후 중복된 파일을 찾아 삭제
    - ii. 장점 : 스토리지 성능 저하 없음

- c. in-line deduplication
  - i. 저장시 중복제거 해시 계산을 실시간 수행
  - ii. 기 저장된 chunk를 식별하면 chunk에 대한 reference만 metadata에 추가
  - iii. 최소(최적)의 네트워크 및 스토리지 사용을 제공

9. metadata partitioning - partitioning schema (data를 다른 DB에 나누고 저장)

- a. vertical partitioning
  - i. 하나의 특정 기능과 관련된 테이블을 한 서버에 저장 (ex) User 관련 테이블을 한 DB에 저장하고 file/chunk 관련 테이블을 다른 DB에 저장
  - ii. 단점
    - 추후 확장 방법 없음
    - 두개의 DB에서 각각의 테이블을 join하면 성능 및 일관성 문제가 발생 (ex) User와 File 테이블 join

b. range based partitioning

- i. 파일 경로 첫 글자 기준
- ii. 정적 분할 체계 : 파일을 예측 가능한 방식으로 save/search 할 수 있음
- iii. 단점
  - 서버들의 분포가 일정하지 않음 (ex) e로 시작하는 파일 경로가 너무 많음

c. hash based partitioning

- i. 객체의 hash를 가져와 DB 선택
- ii. 단점
  - 서버들의 분포가 일정하지 않음 -> consistent hashing으로 해결

10. cache

- a. hot file/chunk 처리용 block storage cache
- b. memcached : 해당 id/hash 및 block server로 전체 chunk를 저장
- c. cache eviction policy : LRU (Least Recently Used)

11. load balancer

- a. 위치

- i. client와 block server 간
  - ii. client와 metadata service 간
  - b. 초기정책 : round-robin method
    - i. 장점
      - simple, overhead가 없음
      - 작동하지 않은 서버에게는 traffic를 보내지 않음
    - ii. 단점
      - 서버 로드를 체크하지 못함 - 로드가 많이 걸린 서버에도 traffic 전송
    - iii. 해결
      - 서버 로드 체크 가능한 intelligent LB 사용
12. security, permissions and file sharing
- a. metadata DB에 각 file 권한 저장

## VI. Designing Facebook Messenger (**수정완료**)

1. facebook messenger
  - a. Text based instant messaging service
  - b. Website 와 Mobile App 연동
2. requirements and goal of the system
  - a. functional requirements
    - i. one-on-one coversation
    - ii. user status (online/offline) notification
    - iii. persistent strorage of chat history
  - b. non-functional requirements
    - i. real-time chat with minum latency
    - ii. high consistent (일관성)  
(ex) 사용자의 디바이스들의 chat history는 모두 같아야 함
    - iii. high availability (가용성) < high consistent (일관성)
  - c. extended requirements
    - i. group chat
    - ii. push notification : offline user에게 message 전송
3. capacity estimation and constraints
  - a. spread sheet 참고
4. high level design (**15.jpg**)
  - a. workflow (**16\_x.jpg**)
    - i. user1 가 chat server에게 message 전송
    - ii. chat server 는 메시지 수신후 user1 에서 승인
    - iii. chat server 는 메시지를 DB에 저장하면서 user2 에게 전송
    - iv. user2 는 메시지를 수신하고 chat server에게 승인 전송
    - v. chat server 는 user1 에게 메시지가 배달되었음을 알림
5. component design
  - a. message handling
    - i. model
      - pull (server 관점)
        - ✓ client가 주기적으로 서버에 요청
        - ✓ client는 대부분 빈 응답을 받음

- ✓ server는 보류중인 메시지 처리기능 필요
- ✓ server 리소스 낭비 심함 (클라이언트의 잦은 요청, 빈응답)
- push (server 관점)
  - ✓ client는 server와 연결만 유지 (http long polling, WebSocket)
  - ✓ server는 보류중인 메시지 처리기능 필요없음 - 새 메시지가 생기는 즉시 클라이언트에게 전송하기 때문
  - ✓ workflow : client 요청 -> 요청을 열린 상태로 유지 (새 메시지가 있을때까지 기다림) -> 새 메시지가 서버에 도착하면 바로 client에게 전송 -> long polling 요청이 timeout 되면 client는 연결 요청을 다시 보냄

ii. 서버는 특정 클라이언트와의 연결을 어떻게 찾나요?

- hash table 이용
- key : UserID
- value : connection object (연결 객체)
- 서버가 메시지 수신후 hash Table에서 해당 사용자를 찾아 connection object를 찾고 메시지를 보냄

iii. 서버가 오프라인 사용자에게 메시지를 보내면 어떻게 되나요?

- 수신자가 연결을 끊은 경우 전송실패, 해당사항을 발신자에게 알림
- 수신자의 long polling 요청이 timeout 된 경우 발신자에게 메시지 전송을 다시 시도하도록 요청
- 재시도시 메시지를 다시 입력하지 않도록 처리

iv. 몇대의 chat server가 필요할까요?

- 500M의 연결, 서버 한대가 50K 연결을 처리한다고 가정 시  $500,000,000 / 50,000 = 10,000$  개 서버 필요

v. 어느 서버가 어떤 사용자와 연결되어 있는지 알 수 있나요?

- 채팅 서버 앞에 LB 설치 가능
- UserID를 연결되어 있는 서버에 매핑

vi. 서버는 메시지 요청을 어떻게 처리하나요?

- 새 메시지 수신 -> DB에 메시지 저장 -> 수신자에게 메시지 전송 -> 보낸사람에게 승인을 보냄
- 수신자 연결을 보유한 서버를 찾음 (hash table 이용) -> 해당 서버로 메시지를 전달하여 수신자에게 전송 -> 수신자가 메시지를 받으면 수신자는 서버에게 승인을 전송 -> 서버는 발신자에게 성공했음을 알림

vii. 메신저는 메시지 순서를 어떻게 유지하나요?

- 서버가 메시지를 받으면 메시지와 time stamp + 일련번호를 저장
- Edge Case : 두 user가 동시에 메시지를 전송시 순서가 보장되지 않음
- 두 클라이언트 모두 메시지 순서는 다르게 보이지만 각각의 유저가 소유한 Device에서는 동일하게 보임 - consistency 유지 가능

b. 데이터베이스에서 메시지 저장 및 검색

i. 메시지를 DB에 저장하는 법

- DB와 함께 작동하는 별도의 Thread를 시작
- DB에 비동기 요청을 보내 메세지를 저장

ii. DB 설계시 고려사항

- 어떻게 해야 DB 연결 풀을 효율적으로 사용할 수 있는지?
- 실패한 요청을 재시도하는 방법
- 재시도 후에도 실패한다면 해당 요청을 기록 할 위치
- 모든 문제 해결 후 기록된 요청을 재시도 하는 방법 (재시도 후 실패)

iii. 어떤 스토리지를 사용해야 하나요?

- 사용조건
  - ✓ 매우 빠른 속도로 소규모 업데이트 가능해야 함
  - ✓ 다양한 레코드를 빠르게 가져올 수 있어야 함
- HBase
  - ✓ message service에 적합
  - ✓ 하나의 키에 여러 값을 저장 (여러 값을 여러 열에 저장)
  - ✓ 열지향 키-값 NoSQL DB

- ✓ HBase는 데이터 그룹화하여 새 데이터를 메모리 버퍼에 저장하고 버퍼가 가득차면 디스크에 덤프 - 많은 작은 데이터를 빠르게 저장하는데 도움
- ✓ HBase 가변 크기 데이터를 저장하는데도 좋음
- ✓ HBase : Google의 BigTable을 모델로 하여 HDFS(Hadoop Distributed File System)에서 실행

- RDBMS

- ✓ message service에 적합하지 않음
- ✓ 메시지를 받거나 보낼때 DB Table에 행을 읽거나 저장하는 횟수가 많기 때문 (대기시간이 길고 DB에 막대한 부하 발생)

iv. 서버에서 데이터를 효율적으로 가져오려면 어떻게 하나요?

- 휴대폰 화면은 작기 때문에 PC에 비해 더 적은 메시지/대화내역이 필요
- 서버에서 데이터를 가져올 때 페이지 선정

c. Managing user's status ([17.jpg](#))

- 사용자의 online/offline 상태를 추적 후 모든 관련 사용자에게 알림
- 서버는 사용자 상태를 쉽게 파악 가능 - 서버가 모든 사용자의 connection object를 유지 관리하고 있기 때문
- 500M DAU가 있다고 가정, 상태 변경사항을 관련 online 사용자에게 broadcast 해야하는 경우 리소스가 많이 소모될 것으로 예상 됩니다. 리소스 소모를 줄이는 방법은?
  - client app이 시작할때만 친구들 현재상태를 가져옴
  - 메시지를 보낸 후 수신자가 offline 경우 보낸 사람에게 오류를 보내고 해당 user 상태를 offline 으로 처리
  - 사용자가 잠깐 online/offline 되는 경우도 있기 때문에 작업은 자주 수행되어서는 안되며 몇초 지연된 broadcast를 함 (자주 online, offline 되는 것을 방지)
  - 화면에 표시되는 사용자만 실시간 update
  - 다른 유저와 새 채팅을 시작할 때 친구들 현재상태를 가져옴

#### iv. summary

- client는 chat server에 접속해서 메시지를 보냄
- chat server는 수신자에게 메시지를 전달
- online 사용자는 서버와 연결을 유지하면서 메시지를 받음
- chat server는 새 메시지가 도착할 때마다 long polling request에 따라 수신자에게 메시지를 push
- 메시지는 HBase(작은 자료를 빠리 저장하고 범위기반 검색을 지원)에 저장
- chat server는 user의 status를 다른 관련 사용자에게 broadcast 해야 함
- client는 화면에 보이는 사용자의 상태 업데이트만 가져옴

### 6. Data partitioning (5년 3.7 PB)

#### a. UserID based partitioning

- i. UserID의 Hash를 기반으로 partitioning (ex) 하나의 shard가 4TB인 경우: 3.7 PB / 4 TB ≈ 900 shard
- ii. hash function : UserID % 900 (number of server)
- iii. 장점 : chat history를 빠르게 가져올 수 있음 (한 사용자의 모든 메시지를 동일한 DB에 저장하기 때문)
- iv. 확장 방법
  - 소수의 물리서버에 다수의 논리 파티션으로 시작
  - 스토리지 수요가 증가함에 따라 논리적 파티션을 분산시키기 위해 물리적 서버를 더 추가 할 수 있음

#### b. MessageID partitioning

- i. 쓸수없음
- ii. 모든 shard에 한 사용자의 메시지를 분산시켜서 저장시키는 경우 메시지를 가져오는데 너무 느림 (모든 shard를 검색 해야 하기 때문)

### 7. cache

- a. 사용자의 화면에 볼 수 있는 최근 대화들을 cache 할 수 있음
- b. 사용자의 모든 메시지가 하나의 shard에 저장되므로 cache 도 한 서버(shard)에 있어야 함

### 8. load balancing

- a. chat server 앞에 위치

- b. 각 UserID를 사용자 연결을 보유한 서버에 매핑한 다음 해당 서버로 요청을 보냄

- c. cache 서버를 위한 LB도 필요

9. fault tolerance and replication

- a. chat server가 다운되면 해당 연결을 다른 서버로 전송해야 하나요?

- i. TCP연결을 다른서버에 failover하는 것은 매우 어려움

- ii. 연결이 끊어지면 client가 다시 연결하도록 해야 함

- b. 메시지 사본을 여러개 저장해야 하나요?

- i. 네, 데이터의 여러 사본을 각각 다른 서버에 저장 (replication)

10. extended requirements

a. group chat

- i. chat server에 저장 가능한 별도의 group chat object 생성

- ii. group chat object
    - GroupChatID
    - chat에 소속된 사람들의 list

- iii. LB는 GroupChatID 기반으로 각 그룹 채팅 메시지를 보낼 수 있음

- iv. DB는 group chat을 GroupChatID 기반으로 분할 된 별도의 Table에 저장

b. push notification

- i. 현재는 online 사용자에게만 메시지를 보냄

- ii. offline 사용자는 발신자에게 fail을 보냄

- iii. push notification을 사용하면 offline 사용자에게도 메시지를 보낼 수 있음 - 해당 메시지를 제조업체 push notification server로 보내 알림

## VII. Designing Tweeter

1. twitter
  - a. sns
  - b. tweet (140자의 메시지) 를 게시하면 follower에게 broadcast
2. requirements and goals of the system
  - a. functional requirements
    - i. tweet 게시
    - ii. 다른 사용자를 follow
    - iii. tweet에 좋아요를 표시
    - iv. timeline 제공 : follow 하는 사용자들의 최신 tweet으로 구성
    - v. tweet에는 image와 video를 첨부
  - b. non-functional requirements
    - i. high availability (가용성)
    - ii. timeline 200ms 이내로 생성
    - iii. high consistency (일관성) < high availability (가용성) -  
사용자는 항상 tweet을 보지 않은 것으로 간주
  - c. extended requirements
    - i. tweet search / reply
    - ii. 인기주제 검색
    - iii. 다른 사용자를 tagging
    - iv. tweet notification
    - v. follower 추천
    - vi. moments
3. capacity estimation and constraint
  - a. spread sheet 참고
4. system apis - soap, rest api
  - a. tweet (api\_dev\_key, tweet\_data, tweet\_location, user\_location, media\_ids, maximum\_result\_to\_return)
    - i. tweet\_data (string) : 140자 이내
    - ii. tweet\_location (string) : optional, tweet이 작성된 위치
    - iii. media\_ids (number[]) : tweet에 첨부된 photo, video id list
    - iv. return : URL 혹은 HTTP 오류코드

5. high level design ([18.jpg](#))

- a. 읽기 무거운 시스템
- b. LB와 함께 다수에 application server 가 필요
- c. 새로운 tweet을 저장, 많은 양의 읽기를 지원하는 DB 필요 (ex) RDBMS
- d. 사진과 동영상을 저장하기 위한 File Storage 필요 (ex) Object Storage
- e. daily write load : 100M -> 1160 tweet / sec
- f. daily read load : 28B -> 325k tweet / sec

6. database schema

a. Tweet

- i. TweetID : int, PK
- ii. UserID : int, FK
- iii. Content : varchar (140)
- iv. TweetLatitude : int
- v. TweetLongitudue : int
- vi. UserLatitude : int
- vii. UserLongitude : int
- viii. CreationDate : datetime
- ix. NumberOfFavorites : int

b. User

- i. UserID : int, PK
- ii. Name : varchar(20)
- iii. Email : varchar(32)
- iv. DateOfBirth : datetime
- v. CreationDate : datetime
- vi. LastLogin : datetime

c. Follower

- i. UserOneID : int
- ii. UserTwoID : int

d. Favorite

- i. TweetID : int
- ii. UserID : int
- iii. CreationDate : datetime

## 7. data sharding

### a. sharding based on UserID

- i. 한 사용자의 모든 데이터를 하나의 shard 저장가능
- ii. Hash Function : UserID % 서버대수
- iii. 단점
  - hot user 발생시 해당 user 정보를 보유한 shard load 가 많이 걸림
  - shard 불균일성 : 증가하는 사용자의 데이터를 균일하게 유지하기 어려움
- iv. 해결방안
  - 일관된 해싱 (consistent hashing)
  - 데이터를 다시 파티션 / 재분배

### b. sharding based on TweetID

- i. Hash Function : TweetID % 서버대수
- ii. tweet 검색시 모든 서버를 query 해야 하며 각 서버는 일련의 tweet을 반환 -> 중앙서버가 반환값을 집계하여 사용자에게 반환
- iii. workflow
  - server는 user의 follower들을 검색
  - server는 모든 DB(shard)에게 query하여 tweet을 검색
  - 각 DB는 각 user(follower)들에 대한 tweet을 찾아 최근순으로 정렬 후 반환
  - server는 반환 값을 병합하고 재정렬하여 최상위 결과를 user에게 반환
  - 장점 : hot user 문제 해결 가능
  - 단점 : 모든 shard 전부 검색해야 됨 -> 대기시간이 길어짐 (DB 앞에 hot tweet을 저장하는 cache를 도입하면 성능이 향상됨)

### c. sharding based on CreationDate

- i. 생성시간 기준으로 tweet을 저장하면 모든 최신 tweet을 빠르게 가져올 수 있음. 왜냐하면 매우 작은 서버 집합만 쿼리하기 때문
- ii. 단점
  - traffic load가 분산되지 않음 - 모든 새 tweet이 하나의 서버로만 이동, 나머지 서버는 유휴상태가 됨

- 읽는 동안 최신 데이터를 보유한 서버는 이전 데이터를 보유한 서버에 비해 load가 높음

iii. TweetID와 CreationDate 기준으로 shard하면 어떤가요?

- Tweet CreationDate을 저장하지 않고 TweetID를 사용  
-> 최신 트윗을 빠르게 찾을 수 있음
- 각 TweetID를 고유하게 만들어야 함 - timestamp가 포함되어야 함

iv. 그렇게 되면 TweetID의 크기는 얼마인가요? 향후 50년간 얼마나 많은 bit가 필요한가요?

- $3600 * 24 * 365 * 50 \approx 1.6B \rightarrow 31\text{bit} 필요$
- 평균 초당 1160개의 새로운 트윗이 생성 -> auto incremented sequence를 위해 17bit 필요
- $31\text{bit} + 17\text{bit} = 48\text{bit}$  (TweetID 길이, 19.jpg)
- 내결함성(fault tolerance)과 성능향상(better performance)을 위해 auto incremented key를 생성하는 두개의 DB서버(하나는 짹수키, 또 하나는 홀수키를 생성)를 갖을 수 있음
- example (epoch second = 1483228800)
  - ✓ 1483228800 000001
  - ✓ 1483228800 000002
  - ✓ 1483228800 000003
  - ✓ 1483228800 000004
  - ✓ ...
- TweetID를 64bit (8byte)로 만들면 향후 1000년 동안 1분단위로 저장 가능

v. 해당방식(sharding based on CreationDate)으로 timeline 생성시 모든서버를 query 하지만 읽기 / 쓰기 속도는 훨씬 빠름

- CreationDate에 secondary Index가 없기 때문에 쓰기 대기시간이 짧아짐
- 읽는 동안 PK에 epoch time이 포함되어 있으므로 생성시간을 따로 filtering 할 필요가 없음

8. cache (20.jpg)

a. DB서버가 hot tweet와 user를 cache

- b. 전체 tweet 객체를 저장할 수 있는 Memcached 같은 상용 솔류션 사용 가능

- c. 캐시정책 : LRU (Least Recently Used)
- d. 캐시크기 : 사드의 일일 읽기 볼륨의 20% (20:80 rule)
- e. 최근 데이터를 캐시하면 어떻게 되나요? 성능이 향상됨

#### 9. timeline generation

- a. Designing Facebook's Newfeed 참조

#### 10. replication and fault tolerance

- a. 시스템은 읽기 용량이 크기때문에 각 DB 파티션마다 여러개의 보조 DB서버를 갖음
- b. 보조 서버는 트래픽만 사용
- c. 모든 쓰기는 먼저 주서버로 이동한 다음 보조서버로 복제
- d. 주서버가 다운될때마다 보조서버로 fail over 가능 - fault tolerance (내결함성) 제공

#### 11. load balancer

- a. 위치

- i. 클라이언트와 응용프로그램 서버간
- ii. 응용프로그램 서버와 DB복제 서버간
- iii. 집계서버와 캐시서버간

- b. 초기 LB 정책 : round-robin 방식

- i. 장점

- 구현간단, over head가 발생하지 않음
- 특정 서버가 동작하지 않으면 LB가 해당서버에 traffic을 보내지 않음 (회전에서 아예 빼버림) -> 동작하지 않은 서버는 트래픽 전송을 중지

- ii. 단점

- 서버로드를 고려하지 않음 - 서버에 과부하가 걸리거나 느려도 해당
- 서버에 Traffic을 줄이지 않음

#### 12. monitoring

- a. 일일 / 초당 새로운 tweet, 일일 최고점을 얼마인가요?
- b. timeline delivery stats, 매일/매초 얼마나 많은 tweet이 전달 되나요?
- c. 사용자가 timeline을 갱신할때 평균 지연시간은 얼마인가요?

### 13. extend requirements

- a. timeline을 어떻게 제공하나요?
  - i. 팔로워에게 최신 tweet을 가져와 시간별로 병합 및 정렬
  - ii. 화면 크기에 따라 tweet 화면에 보여줄 tweet 요청 갯수를 구하고 서버로 요청
  - iii. tweet을 cache해서 작업 속도를 높일 수 있음
  - iv. timeline(feeds)를 사전생성하여 효율성을 높일 수 있음
- b. retweet : DB에 각 tweet 객체를 사용하면 원래 tweet의 ID를 저장가능
- c. trending topics
  - i. 지난 N초동안 가장 빈번하게 발생하는 hash tag 또는 검색어를 cache하고 매 M초마다 계속 update 할 수 있음
  - ii. tweet, 검색어, retweet 또는 좋아요의 빈도에 따라 인기 주제 순위를 매길 수 있음
  - iii. 더 많은 사람들에게 보여지는 주제에 더 많은 가중치를 줌
- d. follow 추천은 어떻게 제공하나요?
  - i. user와 관련있는 누군가가 따르는 사람을 제안
  - ii. 추종자가 더 많은 사람을 가장 먼저 보여줌
- e. moment
  - i. 지난 1시간 혹은 2시간 동안 서로 다른 웹사이트에 대한 주요 뉴스를 얻고 관련 tweet을 파악하고 우선순위를 정한 다음 ML (Supervised Learning) 또는 Clustering을 사용하여 분류 (뉴스, 재무, 엔터테인먼트 등) 한 후 기사를 Moments에서 최신 주제로 표시
- f. search
  - i. indexing, 순위 및 tweet 검색

## VIII. Designing Youtube or Netflix

1. requirements and goals of the system
  - a. functional requirements
    - i. video upload / share / search / likes / dislikes
    - ii. total number of views and etc
    - iii. add / view comments on videos
  - b. non-functional requirements
    - i. high reliability : any video should not be lost
    - ii. high available > high consistency : if a user doesn't see a video for a while, it should be fine.
    - iii. real-time service : while watching video, user should not fee any lag
  - c. not in scope
    - i. video recommendation
    - ii. most popular video
    - iii. channels
    - iv. substriptions
    - v. watch later
    - vi. favorites
    - vii. etc
2. capacity estimation and constraints
  - a. Spread Sheet 참고
3. system apis : soap, rest apis
  - a. uploadVideo (api\_dev\_key, title, description, tag[], category\_id, default\_language, recording\_details, video\_contents)
    - i. recording\_details (string) : location where the video was recording
    - ii. video\_contents (stream) : video to uploaded
    - iii. return : string
      - success : HTTP 202 (request accepted)
      - fail : error code

- b. searchVideo (api\_dev\_key, search\_query, user\_location, maximum\_videos\_to\_return, page\_token)
  - i. search\_query (string) : a string containing the search terms
  - ii. maximum\_videos\_to\_return (number) : maximum number of results returned in one request
  - iii. page\_token (string) : page from result set
  - iv. return : json
    - list of video resources
    - video title, thumbnail, video creation date, view count
- c. streamVideo (api\_dev\_key, video\_id, offset, codec, resolution)
  - i. offset (number) : stream video from any offset
  - ii. return : stream
    - a media stream (a video chunk) from the given offset

#### 4. high level design (21.jpg)

- a. processing queue : upload된 video는 processing queue에 push
- b. encoder : 다양한 형식으로 encoding
- c. video and thumbnail : distributed file storage (object storage, amazon의 s3)에 video, thumbnail 저장
- d. user database : rdbms, user info (name, email, address 등) 저장
- e. video metadata storage : rdbms, 비디오에 관련된 정보 (title, description, file path, uploading user, total views, likes, dislikes comments 등) 저장

#### 5. database schema : RDMS (MySQL)

- a. Video
  - i. VideoID: int, PK
  - ii. Title: varchar (128)
  - iii. Description: varchar (512)
  - iv. Size: int
  - v. Thumbnail: varchar (128)
  - vi. UserID: int, FK

- vii. Likes: int
- viii. Dislikes: int
- ix. Views: int
- b. Comment
  - i. CommentID: int, PK
  - ii. VideoID: int, FK
  - iii. UserID: int, FK
  - iv. Comment: varchar (512)
  - v. CreationDate: datetime
- c. User
  - i. UserID: int, PK
  - ii. Name: varchar (20)
  - iii. Email: varchar (32)
  - iv. Address: varchar (256)
  - v. Age: int
  - vi. DateOfBirth: datetime
  - vii. Registration : tinyint(1)
  - viii. CreationDate : datetime
  - ix. LastLogin : datetime

## 6. component design (22.jpg)

- a. goal : 시스템은 읽기가 아주 무거우므로 비디오를 빠르게 검색하고 끊김없이 볼 수 있는 시스템을 만든다.
- b. read : write = 200 : 1
- c. 비디오는 어디에 저장 되나요?
  - i. distributed file storage system
  - ii. HDFS(hadoop distributed file system), GlusterFS
- d. how should we efficiently manage read traffic?
  - i. separate read traffic from write
  - ii. 비디오 사본이 여러개 이므로 읽기 트래픽을 다른 서버에 배포 할 수 있음
  - iii. metadata는 master-slave로 구성 가능
    - 먼저 master에 저장 -> 해당 master의 모든 slave에 저장

- 단점

- ✓ 새로운 비디오가 먼저 저장 메타데이터는 master에 먼저 저장
- ✓ 이 때는 master 와 slave들이 sync 되지 않음
- ✓ user가 slave에게 자료요청시 오래된 데이터를 반환
- ✓ Master와 slave간에 Sync 시간이 그리 오래 걸리지 않기 때문에 큰 문제가 되진 않음

- iv. thumbnail image는 어디에 저장 되나요?

- object storage : 모든 비디오에 5개의 thumbnail을 가지고 있다고 가정하면 읽기 트래픽이 엄청나게 됨 -> 분산처리를 해야 함
- storage 결정 전 고려사항
  - ✓ thumbnail size : 5kb
  - ✓ thumbnail read traffic은 movie에 비해 엄청나게 많음
- 모든 thumbnail이 disk에 있다면?
  - ✓ 디스크의 다른 위치에서 파일을 읽으려면 많은 검색을 수행 해야 함
  - ✓ 매우 비효율적이며 대기시간이 길어짐

- e. bigtable

- i. 여러 파일을 하나의 블록으로 결합하여 디스크에 저장
- ii. 소량의 데이터를 읽는데 매우 효율적임

- f. cache

- i. cache에 hot thumbnail을 넣으면 지연시간을 개선하는데 도움
- ii. thumbnail file size가 작으면 memory에 다 많이 caching 할 수 있음

- g. video upload : 연결이 끊어지면 동일한 지점에서 다시 시작해야 함

- h. video encoding workflow : a new video upload -> processing queue에 새로운 작업 추가 -> 비디오를 여러 형식으로 인코딩 -> 인코딩 완료 후 업로더에게 알리고 비디오를 공유

- 7. metadata sharding

- a. 수 많은 비디오를 보유, read load 가 매우 높음으로 read/write 작업을 효율적으로 수행 할 수 있도록 데이터를 분산처리해야 함
- b. UserID based sharding
  - i. 한 사용자의 모든 데이터를 한 서버(shard)에 저장
  - ii. hash function : UserID % 서버대수
  - iii. 사용자가 비디오를 query하는 동안 hash function으로 사용자 데이터를 보유한 서버를 찾은 다음 서버에서 읽을 수 있음
  - iv. 타이틀별로 비디오를 검색하기 위해 모든 서버를 퀼리 -> 각 서버는 비디오 세트들을 반환 -> 중앙서버는 집계하고 순위를 정한 후 사용자에게 반환
  - v. 단점
    - hot user 문제 : hot user를 보유한 서버(shard)는 많은 query가 load -> 병목현상이 발생, 전반적인 성능에 영향
    - 일부 사용자는 다른 사용자에 비해 많은 비디오를 저장 할 수 있음 -> 증가하는 사용자 데이터를 균일하게 서버에 배포하는 것이 어려움
    - 해결방법: 일관된 해싱(consistent hashing), 데이터를 재분할
- c. VideoID based sharding
  - i. hash function : VideoID % 서버대수
  - ii. 사용자의 비디오를 찾기 위해 모든 서버를 Query -> 각 서버는 비디오 세트를 반환 -> 중앙서버가 집계하고 순위를 정한 후 사용자에게 반환
  - iii. hot video 문제를 해결
  - iv. db앞에 hot video를 저장하는 cache를 도입하면 성능을 더욱 향상 시킬 수 있음

## 8. video deduplication

- a. data storage : 동일 video 사본을 여러개 저장 -> 저장공간낭비
- b. caching : 캐시에 중복된 자료가 있음 -> 캐시 효율성 저하
- c. network usage : 네트워크를 통해 캐싱시스템으로 전송해야 하는 데이터 양도 증가
- d. energy consumption : 많은 양의 스토리지, 비효율적인 캐시 및 네트워크 사용으로 에너지 낭비가 발생

e. 중복된 검색 결과, 더 긴 비디오 시작시간, 스트리밍이 중간에 끊김 형태로 발생

f. 해결방안 - 중복제거기술(deduplication)

i. inline deduplication

- 비디오 사본을 인코딩, 전송 및 저장하는데 많은 리소스를 절약
- 비디오 업로드 -> 구글의 비디오 일치 알고리즘(블록일치, 우상상관)을 이용해서 중복을 찾음 -> 업로드 중인 비디오의 사본이 이미 있는 경우 업로드를 중지, 단 새로 업로드 중인 비디오의 퀄리티가 높은 경우 계속 업로드, 기존 비디오의 하위 부분이면 누락된 부분만 업로드 가능

#### 9. load balancing

a. 캐시 서버간 일관된 해싱(consistent hashing)을 사용 - 캐시 서버가 로드 균형에 도움이 됨

b. 비디오를 정적 해시 기반 구성표를 사용하여 서버에 매핑하므로 각 비디오의 인기가 다르면 서버에 고르지 않은 load가 발생 할 수 있음 - 비디오가 인기를 얻으면 해당 비디오의 복제본이 있는 서버들이 다른 서버보다 더 많은 트래픽이 발생

c. 해결방법

- i. 해당서버는 동일한 캐시 위치에서 덜 사용중인 서버로 클라이언트를 redirection 시킴
- ii. dynamic http redirection 사용

#### 10. cache

a. 전 세계 분산된 사용자에게 서비스를 제공하려면 대규모 비디오 전송 시스템이 필요

b. 지리적으로 분산된 많은 수의 비디오 캐시 서버를 사용해서 콘텐츠를 사용자에게 더 가깝게(빠르게) 전달

c. 동영상 용량이 크므로, 사용자 성능을 극대화하고 캐시 서버의 부하를 균등하게 분배하는 전략이 필요

d. hot db row를 cache하기 위해 metadata server용 cache를 도입

e. db에 도달하기 전 memecached를 사용해서 데이터 및 application server cache 한다면 원하는 행이 있는지 빠르게 확인

f. 캐시 제거 정책 (cache eviction policy) : LRU (Least Recently Used)

11. cdn (content delivery network)

- a. 정의 : 사용자의 지리적 위치, 웹 페이지의 출처 및 컨텐츠 전달 서버를 기반으로 웹 컨텐츠를 사용자에게 전달하는 분산 서버 시스템
- b. hot video를 cdn으로 옮길 수 있음
  - i. cdn은 여러곳에서 컨텐츠를 복제 -> 비디오가 사용자에게 더 가까이 갈 가능성이 높아짐. hop이 적을 수록 비디오가 친근한 네트워크에서 스트리밍됨
  - ii. cdn은 캐시되지 않은 덜 인기있는 비디오 (하루에 1 ~ 20회 미만 조회)는 다양한 데이터 센터에서 제공
- c. fault tolerance (내결함성)
  - i. db server간 배포는 일관된 해싱 (consistent hashing)을 사용해야 함
  - ii. 일관된 해싱은 사용 불능 서버 교체 뿐만 아니라 서버간에 로드를 분산시키는데 도움

## IX. Designing Typeahead Suggestion

1. typeahead suggestion
  - a. 잘 알려지고 자주 검색되는 용어를 검색
  - b. 검색창에 입력하면 입력한 문자를 기반으로 query를 예측하고 query를 완료하기하기 위한 제안 목록이 제공
  - c. 사용자가 검색을 더 잘 표현하는데 도움 - 사용자를 안내하고 검색 쿼리를 구성하는데 도움
2. requirements and goals of the system
  - a. functional requirements
    - i. 사용자가 입력한 글자로 시작하는 빈도수 기반 상위 10개의 단어를 제안
  - b. non-functional requirements
    - i. 실시간으로 제안목록이 보여져야 함 (200ms 이내)
3. basic system design and algorithm
  - a. trie
    - i. memory에 index를 구성하는 매우 효율적인 데이터 구조 (DB X)
    - ii. 트리와 유사한 데이터 구조
    - iii. 각 노드가 문구의 문자를 순차적으로 저장하는 데 사용 ([23.jpg](#))
    - iv. 저장소 공간을 절약하기 위해 분기가 하나만 있으면 노드를 병합 ([24.jpg](#))
    - v. 대소문자를 구분하지 안아야 하나요?
      - 단순성과 검색 유스 케이스를 위해 구분하지 않는 것이 좋음
  - b. 최고의 제안을 찾는 방법은 무엇인가요? (주어진 접두사의 상위 10개의 단어를 찾는다고 가정)
    - i. 각 노드에서 종료된 검색 수를 저장
    - ii. 주어진 접두사에 대한 최고의 제안을 찾으려면 그 아래의 하위 트리를 탐색
  - c. 접두사가 주어지면 하위 트리를 통과하는데 시간이 얼마나 걸릴까요?
    - i. indexing 해야 할 문구들이 주어지면 trie는 거대해짐
    - ii. 하위 tree를 순회하는데 오래걸림
    - iii. 'system design interview questions' : 30 levels deep

- iv. 대기시간 200ms 이내여야 하기 때문에 솔류션의 효율성을  
개선해야 함
- d. 각 노드마다 최고의 제안(빈도수가 높은)을 저장할 수 있나요?
  - i. 네, 각 노드는 사용자에게 반환할 상위 10개의 문구를 저장
  - ii. 검색속도는 빨라지지만 추가 저장공간이 아주 많이 필요함
  - iii. space complexity 최적화 방안
    - 전체 문구를 저장하지 않고 leaf 노드에 참조값(pointer)  
만 저장하여 스토리지를 절약할 수 있음
- e. Trie는 어떻게 만드나요?
  - i. 각 부모노드는 자식노드를 iteration 혹은 recursion 하여 주요  
제안 및 개수를 계산
  - ii. 부모노드는 자녀 주요 제안을 결합하여 주요 제안을 결정
- f. Trie는 어떻게 업데이트 하나요?
  - i. 매일 5B 검색을 한다고 가정하면 초당 60K개의 쿼리가 발생
  - ii. 모든 쿼리를 update 하면 리소스가 많이 소요 -> 읽기 요청도  
방해 받음
  - iii. 해결방안 - 일정 간격마다 trie를 offline으로 update
    - 새로운 쿼리가 들어오면 로그를 기록하고 빈도를 체크
    - 모든 쿼리를 기록하거나 1000번째 쿼리마다 셱플링 및  
로깅을 수행
    - MapReduce로 지난 1시간 동안 검색된 모든 용어의  
빈도를 계산 -> 산출된 데이터로 trie를 업데이트
    - MapReduce workflow
      - ✓ 입력된 데이터를 잘게 쪼갬
      - ✓ 쪼갠 데이터를 Map 연산
      - ✓ Partitioner가 모음
      - ✓ reduce 처리
      - ✓ 정렬
      - ✓ 유저에게 리턴
    - update 는 offline으로 처리 : 읽기 쿼리가 update  
요청에 의해 차단되는 것을 방지
    - ✓ 각 서버에 trie 사본을 만들어 offline으로 업데이트  
-> 완료되면 사본을 시작하고 원본을 버림

- ✓ master-slave로 구성 -> master가 트래픽을 제공하는 동안 slave를 update -> update가 완료되면 slave를 new master로 만듦

#### 9. 자동 완성 제안의 빈도는 어떻게 업데이트 하나요?

- i. 각 노드마다 제안 빈도수를 저장하고 있기 때문에 그것들도 업데이트 해야 함
- ii. 모든 검색어를 처음부터 다시 계산하지 않고 빈도 차이만 업데이트 (ex) 지난 10일동안 검색한 모든 단어에 개수를 유지한다고 가정 더 이상 포함되지 않은 기간에서 개수를 빼고 포함된 새기간에 대한 개수를 더함
- iii. 각 항의 EMA (Exponential Moving Average, 지수 가중 이동평균)에 따라 빈도를 더하고 뺄 수 있음. EMA에는 최신 데이터에서 더 많은 가중치를 부여함
- iv. workflow
  - trie에 새로운 용어(단어)를 삽입 후 구문의 leaf 노드로 이동하여 빈도를 늘림
  - 각 노드에 상위 10개의 쿼리를 update
  - 다시 root까지 이동
  - 모든 부모에 대해 현재 쿼리가 상위 10개의 일부인지 확인 맞다면 해당 빈도를 update 후 새 용어를 삽입하고 빈도가 가장 낮은 용어를 제거
- v. Trie에서 용어를 어떻게 제거하나요?
  - 법적인 문제, 불법 복제 등으로 trie에 용어를 제거
  - 정기 업데이트가 발생하면 trie에 해당 용어를 제거 할 수 있음
- vi. 제안에 대한 다른 순위기준은 무엇인가요?
  - 빈도수 외에 신선도, 사용자 위치, 언어, 인구통계, 개인이력등과 같은 다른 요소들도 고려

#### 4. permanent storage of the trie

- a. trie를 file에 저장하는 방법 (snapshot 작성법, 25.jpg) - 서버 리부팅시 필요
  - i. trie의 snapshot을 정기적으로 찍어 파일로 저장
  - ii. root node 시작, level별로 level을 저장

- iii. 각 노드마다 포함된 문자와 보유한 자식 수를 저장
- iv. 각 노드 바로 뒤에 모든 자식을 배치 (preorder traverse)
- v. node value 뒤에 자식수를 붙임

5. scale estimation

- a. Spread Sheet 참고

6. data partition

- a. range based partitioning

- i. 첫 글자 기준으로 저장 할 서버를 선정
- ii. 자주 발생하지 않은 문자들을 하나의 서버에 저장
- iii. 예측 가능한 방식으로 저장하고 검색할 수 있도록 분할 체계 (hash function)을 정적으로 생성
- iv. 단점
  - 서버의 불균일성
  - 한 단어 ('E') 분할 영역에 overflow 되는 경우 발생 - 예측이 어렵다

- b. partition based on the maximum capacity of the server

- i. 서버의 최대 메모리 용량을 기준으로 나눔
  - 한서버에 용량이 가득 차면 다른 서버에 저장 (ex)

서버 1, A ~ AABC

서버 2, AABD ~ BXA

서버 3, BXB ~ CDA

...

사용자가 'A'입력 - 서버 1, 2에 큐리

사용자가 'AA'입력 - 서버 1, 2에 큐리

사용자가 'AAA'입력 - 서버 1에 큐리

- ii. LB를 trie 서버 앞에 위치 - mapping 을 저장하고 traffic을 리디렉션 시켜 시스템 성능을 높임

- iii. 여러 서버를 쿼리 하는 경우 LB와 trie서버 사이에 aggregator서버를 추가 - 여러 trie 결과를 집계하여 최상위 결과를 client에 반환
  - iv. 여전히 hot spot은 발생
  - v. cap으로 시작하는 용어에 대한 쿼리가 많으면 용어를 보유한 서버가 다른 서버에 비해 높은 load를 가짐
- c. partition based on the hash of the term
- i. 각 용어에 서버 번호를 생성하는 해시 함수를 전달, 해당 서버에 용어를 저장
  - ii. 용어 분포를 무작위로 만들어 hotspot을 최소화
  - iii. 단점 : 모든 서버에 요청한 다음 결과를 집계해야만 용어에 대한 자동완성 제안을 찾음

## 7. cache

- a. 가장 많이 검색되는 검색어를 caching하면 시스템 성능이 큰 도움이 됨
- b. 가장 자주 검색되는 용어와 그에 대한 제안을 담고 있는 trie서버 앞에 별도의 cache서버를 위치
- c. 간단한 계산, 개인화 또는 추세 데이터를 기반으로 각 제안에 대한 참여를 예측, 이러한 용어를 미리 cache할 수 있는 간단한 ML 모델도 구축 가능

## 8. replication and load balancer

- a. 둘다 모두 trie에 대한 복제분이 있어야 함
- b. 데이터 파티셔닝 체계를 추적하고 접두사 기반으로 트래픽을 replication 하는 load balancer가 필요

## 9. fault tolerance

- a. trie 서버가 다운되면 어떻게 하나요?

  - i. master-slave 구성을 갖음
  - ii. master가 죽으면 slave가 대신
  - iii. 백업되는 모든 서버는 마지막 snapshot을 기반으로 trie를 재 구성할 수 있음

## 10. typeahead client

- a. 최적화 방안
  - i. 사용자가 50ms동안 아무키도 누르지 않은 경우만 서버로 query
  - ii. 사용자가 지속적으로 입력시 진행중인 서버 요청을 취소
  - iii. 처음 두 문자를 입력할 때까지 대기

- iv. 서버에서 일부 데이터 미리 가져와 향후 요청에 대비
- v. 최근 제안 내역을 로컬에 저장 (최근 사용한 history는 재사용률이 높음)
- vi. 웹사이트를 열자마자 클라이언트를 서버와 미리 연결함 - 문자 입력 후 클라이언트가 서버와 연결하는 시간을 줄임
- vii. 서버는 효율성 (effciency)을 위해 캐시의 일부를 CDN 및 ISP (Internet Service Provider)로 push 할 수 있음

#### 11. personalization

- a. 사용자는 기록검색, 위치, 언어등을 기반으로 사전 제안을 받음
- b. 사용자의 개인기록을 서버에 별도로 저장하고 클라이언트에 cache

## X. Designing an API Rate Limiter

1. rate limiter
  - a. 요청의 수에 따라 사용량 조절 - 많은 수에 요청을 받고 있더라도 제한된 수의 요청만 처리
  - b. example
    - i. 사용자는 초당 1개의 메시지만 발송
    - ii. 사용자는 신용카드 사용 시 비밀번호를 3번 이상 틀렸을 경우 사용 중지
    - iii. IP당 20개의 계정만 만들 수 있음
2. rate limiter는 왜 필요한가요?
  - a. DDOS 공격, 무차별 암호시도, 무차별 신용카드거래등과 같은 악의적 행동으로 부터 서비스 및 사용자 보호
  - b. 수익 손실을 방지, 인프라 비용 절감, 스펠체단, 온라인 괴롭힘을 중지
  - c. 장점
    - i. client, script 오작동 방지 - 일부 entity에서 많은 요청이 발생 시 사전 차단하여 traffic이 과다하게 발생하는 것을 방지
    - ii. 보안 : 비밀번호 시도 횟수 제한
    - iii. 시스템을 학대하는 행동 및 설계 방지 - API 제한이 있다면 client 개발자가 server에게 불필요한 반복 요청 방지 가능
    - iv. 비용 및 리소스 사용을 통제
      - API 서비스에 대한 제어를 가능하게 하여 일반적이 않은 입력을 사전에 차단 (ex) 1초에 수천개의 게시물을 작성하는 것을 막음
    - v. 수익 - 요금별 사용량, 기능을 제한하여 고객이 더 비싼 요금을 지불하도록 유도 (이통사)
    - vi. 교통체증 제거
3. requirements and goals of the system
  - a. functional requirements
    - i. entity가 특정시간에 api로 전송할 수 있는 요청수를 제한 (ex) 초당 15개 이상 요청 불가
    - ii. api는 클러스터를 통해 access 할 수 있으므로 다른서버에서의 속도제한도 고려해야 함

- iii. 미리 정해진 임계값에 도달하면 서버는 에러메시지를 표시
- b. non-functional requirements
  - i. high availability : 외부 공격으로 부터 서비스를 보호하는 수문장의 역할을 하므로 항상 작동해야 함
  - ii. 사용자 환경에 영향을 미치는 상당한 지연을 초래해서는 안됨

#### 4. 어떻게 rate limiting을 할 수 있나요?

- a. application level, api level에서 limit 정의 가능
- b. limit가 넘어서면 서버는 HTTP 429 - Too many request 리턴

#### 5. throttling type

- a. hard throttling
  - i. api 요청 수는 throttle 제한을 초과 할 수 없음
- b. soft throttling
  - i. api 요청 제한을 특정 백분율을 초과하도록 설정  
(ex) 분당 100개의 메시지로 제한, 10% 초과하면 110개까지는 봐줌
- c. elastic or dynamic throttling
  - i. 시스템에 사용 가능한 리소스가 있는 경우 요청수가 임계값을 초과할 수 있음

#### 6. rate limiting algorithm ([26.jpg](#))

- a. fixed window algorithm (고정창 알고리즘) : 계단처럼 시간 프레임 단위 안에 있는 메시지만 가지고 계산  
(ex) 각 계단에는 2개 이상의 사과를 올려 놓으면 안됨
- b. rolling window algorithm (롤링 창 알고리즘) : 물이 흐르듯 연속적인 시간 프레임 안에 있는 메시지만 가지고 계산 (ex) 구멍 뚫은 종이 사이로 사과가 2개 이상 보이면 안됨

#### 7. high level design ([27.jpg](#))

- a. rate limiter는 storage (db, cache)가 제공할 요청과 거부할 요청을 결정
- b. workflow
  - i. 새로운 요청이 도착
  - ii. rate limiter가 승인 또는 거부 결정
  - iii. 승인된 요청만 storage에서 web server로 전달

#### 8. fixed window algorithm

- a. 시나리오 - 사용자당 요청 수를 분당 3개로 제한 ([28.jpg](#), [29.jpg](#))

- i. UserID가 hash table에 없는 경우
  - insert 하고 count를 1로 설정
  - startTime을 현재 시간으로 설정 후 요청을 허용
- ii. UserID가 hash table에 있는 경우
  - CurrentTime - StartTime  $\geq$  1 min
    - ✓ StartTime를 현재시간으로 update 하고 count를 1로 설정
  - CurrentTime - StartTime < 1 min
    - ✓ Count < 3 이면 Count를 1 늘리고 요청 허용
    - ✓ Count = 3 이면 요청을 거부
- iii. 단점
  - 1분 마지막 순간에 3개, 다음 1분 시작에 3개라면 2초에 6개로 승인되지만, 6개의 간격은 1분 미만이 됨 ([30.jpg](#)) -> sliding (rolling) window algorithm으로 해결 가능
  - Atomicity (원자성, [31.jpg](#))
    - ✓ 사용자의 현재 Count가 2이고 두 번 더 요청이 들어온다고 가정
    - ✓ 두개의 개별 프로세스가 이러한 요청을 각각 처리하고 둘 중 하나를 업데이트 하기 전에 Count를 동시에 읽은 경우 각 프로세스는 속도 제한에 도달하지 않았다고 판단
    - ✓ 해결방안 : Redis를 사용하여 key-value를 저장하는 경우 read update 작업중에 Redis 잠금을 사용 - 동일한 사용자의 동시 요청 속도가 늦추고 또다른 복잡성 계층을 도입해야 함 (비용이 발생). memcached를 사용할 경우 비슷한 문제가 발생

b. 모든 사용자 데이터를 저장하는데 얼마나 많은 메모리가 필요하나요?

- i. UserID : 8 (byte)
- ii. Count : 2 (최대 65k 숫자 표현 가능)
- iii. EpochTime : 4 (분, 초만 저장시 2 byte 로도 가능)
- iv. hash table에 20 byte의 overhead가 있다고 가정, 1M 사용자를 추적하는 경우 :  $(12 + 20) * 1M = 32mb$

- v. 원자성(Atomicity) 문제를 해결하기 위해 각 사용자 레코드를 접근한다고 하면 추가로 4byte 숫자가 필요 하므로  $(32 + 4) * 1M = 36mb$  의 메모리가 필요
- c. 분산 환경에서는 Redis, Memcached 솔루션 사용
  - i. 모든 데이터를 원격 Redis 서버에 저장
  - ii. 모든 Rate Limiter server는 요청을 처리하거나 조절하기 전에 server를 읽고 update 함
- 9. sliding window algorithm - 사용자 당 각 요청을 추적 할 수 있다면 슬라이딩 윈도우를 유지 할 수 있음. 각 요청의 TimeStamp를 hash table의 value field에 있는 Redis Sorted Set에 저장
  - a. 시나리오 - 사용자당 요청 수를 분당 3개로 제한 ([32.jpg](#), [33.jpg](#))
    - i. Sorted Set에서 CurrentTime - 1 minute 보다 오래된 TimeStamp를 모두 제거
    - ii. Sorted Set의 총 item 수를 계산 - 이 수가 3보다 큰 경우 요청 거부
    - iii. Sorted Set에 현재 시간을 삽입 후 요청 승인
  - b. 슬라이딩 윈도우에서 모든 사용자 데이터를 저장하는데 얼마나 많은 메모리가 필요하나요?
    - i. UserID : 8 (byte)
    - ii. EpochTime : 4
    - iii. assume
      - 시간당 500개의 요청 제한 속도가 필요하다 가정
      - Sorted Set Overhead : 20
      - Hash Table Overhead : 20
    - iv.  $8 + 4 + 20 * 500 + 20 = 12kb$
    - v. Sorted Set에서 최소한 2개의 pointer가 필요
      - 이전 요소에 대한 pointer
      - 다음 요소에 대한 pointer
      - 64bit system에서는 포인터는 8 byte 이므로 총 16 byte 필요
    - vi. 1M의 사용자를 추적하는 경우 필요 메모리는  $12kb * 1M \approx 12GB$

- c. sliding window algorithm은 fixed window algorithm에 비해 많은 메모리가 필요 -> scalability (확장성) issue 발생 -> 최적의 메모리 사용을 위해 두 알고리즘 결합 (sliding window with counters)

## 10. sliding window with counters

- a. 구현방법 (34.jpg)
  - i. Sorted Set에 TimeStamp와 Count 를 함께 저장
  - ii. TimeStamp를 일정간격(그림에서는 1분) 으로 저장
  - iii. Sorted Set에 있는 Count를 모두 더 해 Limit 적용
- b. Sliding window with counters에서 사용자 데이터는 모두 얼마인가요?
  - i. UserID : 8 (byte)
  - ii. EpochTime : 4
  - iii. Counter : 2
  - iv. assume
    - Limit : 500 request / hour
    - Overhead
      - ✓ Hash Table : 20
      - ✓ Redis Hash : 20
  - v. 매분마다 카운트를 유지하기 때문에 각 사용자당 60개의 항목이 필요
  - vi.  $(8 + 4 + 2 + 20) * 60 + 20 = 1.6\text{KB}$
  - vii. 1M 사용자를 추적하는 경우  $1.6\text{KB} * 1\text{M} \approx 1.6\text{GB}$

## 11. data sharding and caching - UserID based sharding

- a. Fault Tolerance (내결함성) 와 replication (복제) 를 위해 Consistent Hashing (일관된 해싱) 을 사용해야 함
- b. 서로 다른 api에 대해 서로 다른 제안을 설정하려면 api별로 / 사용자 별로 shard를 선택 할 수 있음
  - i.createUrl and deleteUrl of URL Shortener 처럼 api가 분할 된 경우 api shard에 대해 별도의 api limiter를 두는게 좋음
  - ii. api 에 hash based partitioning을 한다면 createUrl에 시간당 100개, 분당 3개 식으로 각 partition 속도를 제한 할 수 있음

c. cache 제거 정책 : LRU (Least Recently Used), 최근 활성 사용자를 caching 함으로서 큰 잇점을 얻음

- i. application server는 backend server에 도달하기 전에 cache에 원하는 record가 있는지 빠르게 확인 가능
- ii. rate limiter는 cache의 모든 counter와 time stamp만 update 해서 write-back cache의 잇점을 크게 얻음

12. IP나 유저별 제한을 할 수 있나요?

a. IP

- i. IP당 요청을 조절
- ii. good / bad 행동자를 구별하는데 최적은 아니지만 속도 제한이 전혀 없는거 보다는 좋음
- iii. 단점
  - 한명의 bad 사용자가 다른 사용자를 제한 - 단일 공용 IP를 사용하는 경우
  - 해커가 많은 IPv6 주소를 사용하는 경우 서버에서 해당 주소들을 추적하다 보면 memory overflow가 발생

b. User

- i. 사용자 인증 후 api에서 속도 제한을 수행
- ii. 인증되면 사용자에게 각 요청과 함께 전달할 token이 제공
- iii. 유효한 token이 있는 특정 api에 대한 한도를 추적
- iv. 만약 로그인 api 자체를 제한 해야 하는 경우는 어떻게 하나요?
  - 이것에 약점은 해커가 한도까지 잘못된 자격 증명을 입력하여 사용자에 대한 서비스 거부 공격을 시도에 취약
  - 공격 받은 중에 실제 사용자를 로그인 할 수 없음

c. Hybrid

- i. ip방식과 user 단위 rate limiter를 모두 수행단독으로 구현할 경우 약점이 있기 때문

## XI. Designing Twitter Search

1. twitter search - tweet을 검색하는 서비스
2. requirements and goal of the system
  - a. 효과적인 tweet 저장 / 탐색 하는 시스템 구축
3. capacity estimations and constraints
  - a. assume
    - i. average incoming tweets : 400M / day
    - ii. average size of tweet : 300 byte / tweet
  - b. estimation
    - i. storage
      - $400M * 300\text{byte} = 120\text{GB} / \text{day}$
      - $120\text{GB} / 86400\text{sec} \approx 1.38\text{MB} / \text{sec}$
4. system apis - soap or rest api
  - a. search (api\_dev\_key, search\_terms, maximum\_result\_to\_return, sort, page\_token)
    - i. sort (number) : optional, sort mode
      - Latest first (0 - default)
      - Best matched (1)
      - Most liked (2)
    - ii. page\_token (string) : resultSet에서 반환 되어야 할 페이지 지정
    - iii. return (json)
      - search\_terms에 match되는 tweets을 반환
      - UserID, Name, TweetText, TweetID, CreationDate, Likes, etc
5. high level design ([35.jpg](#))
  - a. 모든 상태를 db에 저장하고 어떤 단어가 어떤 tweet에 들어 있는지 추적할 수 있는 index를 생성해야 함
  - b. index 목적 : tweet 검색 속도 향상
6. component design
  - a. storage

- i. 매일 120GB의 새로운 데이터가 저장됨 -> 방대한 데이터를 고려, 분산서버에 효율적으로 저장할 수 있는 data partitioning scheme가 필요
  - ii. 향후 5년간 계획은 어떻게 되나요?
    - 120GB \* 365 \* 5 ~= 200TB
    - HDD 용량의 80% 만 사용한다고 가정 250TB의 HDD 용량이 필요
    - fault tolerance (내결합성) 을 위해 모든 tweet의 사본을 저장한다고 가정 -> 하드 요구 용량은 250TB \* 2 = 500TB 필요
    - 최신 서버가 4TB data 를 저장한다고 가정하면 125대의 서버가 필요
  - iii. tweet 저장 workflow
    - rdbms에 저장 -> TweetID와 TweetText라는 두개의 열이 있는 테이블에 저장 -> TweetID based Sharding -> hash function ( TweetID % number of server ) 로 storage server에 mapping
  - iv. unique한 TweetID를 어떻게 만들 수 있나요?
    - 매일 400M의 새로운 tweets을 저장한다면 5년동안 얼마나 많은 tweet을 예상할 수 있나요?  
✓ 400M \* 365 \* 5 ~= 730B
    - tweet 저장시 unique한 TweetID를 생성할 수 있는 server가 있다고 가정 - hash function 는 TweetID % 서버대수
- b. index
- i. tweet 쿼리는 word 단위로 구성되므로 어떤 단어가 어떤 tweet에 있는지 알려주는 index 를 생성
  - ii. assume
    - 모든 영어단어, 사람이름, 도시이름 등과 같은 유명한(사용빈도가 높은) 명사에 대한 색인을 만듦
    - 약 300K 영어단어와 200K 명사가 있음
    - 단어 평균길이 : 5 character
  - iii. estimation

- 모든 단어를 memory에 저장시  $500K * 5 = 2.5MB$
- 지난 2년동안의 모든 tweet의 index를 memory에 유지한다고 가정
  - ✓ 292B ( $730B / 5 * 2$ ) tweet / 2years
  - ✓ TweetID 를 5 byte라 가정하면  $292B * 5 = 1460GB$
- index 큰 분산 해시 테이블과 유사
  - ✓ key : word
  - ✓ value : TweetID
  - ✓ 트윗당 평균 단어수를 40 words / tweet 라 가정  
-> 전치사와 'an', 'and' 같은 작은 단어들은 index 하지 않으므로 15 words / tweet 라고 가정
  - ✓  $1460GB * 15 + 2.5MB \approx 21TB$
  - ✓ 서버에 144GB memory가 있다고 가정시 152대 ( $21000 / 144$ ) 의 서버가 필요

### c. database partitioning

#### i. word based sharding

- tweet의 모든 단어를 검색 -> 각 단어별 hash를 계산 후 해당 index 서버에 저장
- 특정 단어가 포함된 모든 tweet을 찾으려면 해당 단어가 저장된 server만을 query 하면 됨
- 단점
  - ✓ hot word 처리 문제 : 해당 단어를 보유한 서버에 load 가 많이 걸림 -> 이 부하는 시스템 전반에 영향을 줌 (느려짐)
  - ✓ 배포의 불균일성 : 각 단어의 비중이 다름 -> 균일한 분포 유지가 어려움
- 해결안
  - ✓ consistent hashing (일관된 해싱)
  - ✓ repartition (재 파티션, 처음부터 다시)

#### ii. tweet object based sharding

- TweetID를 hash function에 전달 하여 서버를 찾음 -> 해당 서버에서 tweet의 모든 단어를 indexing -> 특정

단어를 query하려면 모든 서버를 query 해야 함 -> 각 서버는 TweetID Set 반환 -> 중앙서버가 집계 후 사용자에게 반환

a. 36.jpg

7. fault tolerance

a. index server 죽으면 어떻게 되나요?

- i. master-slave로 구성되어 있다면 slave가 master를 대행
- ii. master와 slave 모두 동일한 index 사본을 가지고 있어야 함

b. 그렇다면 master 와 slave 모두 동시에 죽으면 어떻게 되나요?

- i. 서비스 정지
- ii. 복구

- 새 서버를 할당하고 동일한 index를 다시 작성 - 해당 서버에 어떤 word / tweet이 보관되어 있는지 모르기 때문
- tweet object based sharding을 사용하는 경우
  - ✓ brute-force solution : 전체 db를 모두 다시 검색, TweetID로 hash function을 통해 서버에 저장될 모든 tweet를 파악
- 단점
  - ✓ 비효율적임
  - ✓ 서버가 재구축 되는 동안 서비스를 할 수 없음
- Tweet과 index 서버간 mapping을 효율적으로 검색하려면 어떻게 해야 하나요?
  - ✓ 모든 TweetID를 index server에 mapping하는 reverse index를 만듦 -> 해당 정보는 index build server가 보관
  - ✓ key : index server
  - ✓ value : 해당 index server에 유지되는 TweetID 모두를 포함하는 hash set을 만드는 hash table (시스템 속도 대폭 향상)
  - ✓ fault tolerance (내결함성) 을 위해 index build server 복제본을 운영

8. cache

a. hot tweet 처리를 위해 db 앞에 cache 도입

- b. Memcached를 사용한다면 모든 tweet을 memory에 저장 가능
- c. application server는 backend db에 도달 전에 cache에 보관된 tweet이 있는지 신속하게 확인 (시스템 속도 대폭 향상)
- d. client의 사용패턴에 따라 필요한 캐시 서버 수를 조정 가능
- e. cache eviction policy : LRU (Least Recently Used)

## 9. load balancing

### a. 위치

- i. client 와 web server 간
- ii. web server와 application server 간
- iii. application server와 db server 간

### b. 초기정책 : round-robin 방식

#### i. 장점

- 구현이 간단하고 overhead가 발생하지 않음
- backend server간 수신요청을 균등하게 분배
- 사용불능서버를 교체하지 않고도 트래픽 전송을 중지

#### ii. 단점

- server load check를 못함
- 특정 서버에 과부하가 걸리더라도 트래픽을 보내는 것을 중단하지 않음

#### iii. 해결안

- backend server에 주기적으로 load를 체크, 이를 기반으로 traffic을 조정하는 지능적인 LB solution을 배치

## 10. ranking

### a. 소셜 그래프 거리, 인기도, 관련성 등으로 검색 결과의 순위를 매기려면 어떻게 해야 하나요?

- i. tweet의 인기, 댓글수에 따라 tweet 순위를 정한다고 가정
- ii. ranking algorithm
  - 좋아하는 사람 수등을 기반으로 한 인기 수 field를 만들고 indexing 하여 저장 가능
  - aggregator server는 이러한 모든 결과를 집합하고 정렬 (인기도순) 한 후 최상의 결과를 사용자에게 보냄



## XII. Designing a Web Crawler

1. web crawler
  - a. web을 체계적이고 자동화된 방식으로 탐색하는 software
  - b. 시작페이지에서 link를 recursion하여 document를 수집
  - c. 용도
    - i. search engine
      - web crawling 하여 최신 data를 제공
      - 모든 page를 download하여 더 빠른 검색을 수행할 수 있도록 index를 만듦
    - ii. web page / link test
    - iii. site structure와 contents가 변경 되는 시기를 모니터링 하기 위해
    - iv. 널리 사용되는 web site의 mirror site를 운용
    - v. 저작권 침해 (copyright infringements) 검색
    - vi. 웹에서 멀티미디어 파일에 저장된 내용을 이해하는 것과 같은 특수 목적 인덱스를 생성
2. requirements and goal of the system
  - a. Scalability
    - i. 웹 전체를 crawling 할 수 있어야 함
    - ii. 수억개의 웹 문서를 가져오는데 사용할 수 있도록 시스템이 확장 가능해야 함 - 모듈화
  - b. Extensibility
    - i. 새로운 기능이 추가될 것으로 예상 -> 모듈 방식으로 설계
    - ii. 추후에 download 해서 처리해야 하는 새로운 문서 유형이 있을 수 있음
3. design consideration
  - a. HTML page이 전용인가요? 아니면 source file, image, video 등과 같은 유형의 미디어도 가져와 저장해야 하나요?
    - i. 미디어 유형을 download 하기 위해 범용 crawler를 작성하는 경우 구문 분석 모듈을 미디어 별 각기 다른 모듈 유형으로 추출하는 것을 만들 수 있음

- ii. crawler html만 처리한다고 가정하지만 추후에 확장 가능한 형태이기 때문에 미디어 유형을 쉽게 추가 할 수 있도록 설계할 예정
- b. 어떤 protocol을 사용할까요?
  - i. HTTP만 사용할 예정
  - ii. 추후 FTP나 기타 프로토콜을 사용하는 것을 확장하는 것은 어렵지 않음 - 모듈방식으로 설계하기 때문
- c. RobotsExclusion 이 뭔가요?
  - i. 사이트 crawling 시 지정된 파일이나 디렉토리를 무시하도록 요청하는 기능  
(ex) 개인정보보호
  - ii. Robots Exclusion Protocol을 사용하려면 web crawler에서 실제 contents를 download하기 전에 web site에서 이러한 선언이 포함된 robot.txt 파일을 가져와야 함

#### 4. capacity estimation and constraints

- a. 4주 이내에 15B page를 crawling 하려면 초당 몇 page를 가져와야 하나요?
  - i.  $15B / 4 * 7 * 86400 \approx 6200 \text{ page / sec}$
- b. Storage는 어떤가요?
  - i. html text 만 다룬다고 가정 평균 페이지를 100KB로 가정
  - ii. 각 page마다 500 byte의 metadata 를 저장한다면  $15B * (100KB + 500byte) \approx 1.5PB$  필요
  - iii. 70% capacity model (storage system 용량의 70% 까지만 사용) 을 가정한다면  $1.5PB / 0.7 \approx 2.14 PB$  필요

#### 5. high level design

- a. basic algorithm : seed URL list를 입력하여 다음 단계를 반복적으로 실행
  - i. 방문하지 않은 url list에서 url을 선택
  - ii. host name으로 부터 ip 추출
  - iii. host에 연결해서 해당 문서를 다운로드
  - iv. 문서 내용을 분석, 새로 생기는 url을 url list에 추가
  - v. download 한 문서를 처리
    - 문서 저장 또는 내용 index 생성

b. crawling 하는 방식

- i. BFS (breath first search, 넓이 우선 탐색) <- 주로 사용됨
- ii. DFS (depth first search, 깊이 우선 탐색)
  - crawler가 이미 웹사이트와 연결을 설정한 경우
  - 웹사이트 내의 모든 url을 dfs만 사용하여 일부 리소스 (hand shaking overhead) 를 절약 할 수 있는 경우
- iii. 경로 오름차순 크롤링 (path ascending crawling)
  - 특정 웹사이트를 정기적으로 crawling 할 때 inbound link가 없는 많은 고립된 resource 나 resource를 검색하는 데 도움
  - 이 체계에서 crawler는 crawling 하려는 각 url이 모든 경로로 올라감

(ex)

<http://foo.com/a/b/page.html>

위의 seed url이 성공되면 /a/b/, a/, /를 크롤링 하는 방식

c. 효율적인 crawling이 왜 어려운지?

- i. web page가 너무 방대함
  - 대량의 웹페이지는 crawler가 전부가 아닌 일부만 다운로드 할 가능성이 있음 -> crawler는 다운로드 우선순위를 정할 만큼 지능적이여야 함
- ii. web page 쉽기 변경됨
  - 특정 웹사이트에서 crawler가 마지막 page를 다운로드 하려고 하는 순간 page가 변경되거나 새 page가 추가될 수 있음

d. crawler 최소 구성 요소

- i. URL frontier
  - download 하려는 url 목록을 저장
  - crawling 하려는 url의 우선순위를 선정
- ii. HTTP fetcher
  - 서버에서 웹페이지를 검색

- iii. Extractor
  - html 문서에서 link 추출
- iv. Duplicate Eliminator
  - 동일한 내용이 두번 추출 되지 않도록 함
- v. Datastore
  - 검색된 페이지, url 및 기타 metadata를 저장

e. 37.jpg

6. component design

- a. assume
  - i. crawler 단일서버에서 운용
  - ii. 모든 crawling이 여러 작업스레드에서 수행
  - iii. 각 작업스레드가 문서를 loop로 다운로드 처리하는 필요한 모든 단계를 수행
- b. workflow (38.jpg)
  - i. url frontier에서 url을 앞에꺼를 하나 뺀 후 해당 url로 이동하여 download
  - ii. url은 다운로드에 사용해야 하는 network protocol을 식별 (http)
  - iii. protocol은 추후 module 식으로 쉽게 추가 가능
  - iv. url schema에 따라 작업자는 적절한 protocol module을 호출하여 문서를 다운로드
  - v. 다운로드 후에는 dis (document input stream)에 배치
  - vi. 문서를 dis에 넣으면 다른 module이 문서를 여러번 다시 읽을 수 있음 (local 저장, 서버에 다시 접속 할 필요 없음)
  - vii. 문서가 dis에 작성되면 작업자 쓰레드는 dedupe test 호출, 해당 문서(url)이 이미 방문했는지 여부를 판단, 이전에 방문했다면 더이상 처리하지 않고 frontier에서 해당 url을 제거
  - viii. crawler는 download 한 문서를 처리
    - 문서의 MIME 타입에 따라 해당 타입에 연관된 각 처리 모듈의 process method를 호출
    - page에서 모든 link를 추출
    - link는 url로 변환, 사용자가 제공한 url filter에 대해 test되어 download 여부를 결정

- url 필터를 통과하면 작업 url 확인 test를 수행, url이 이전에 표시되었는지 확인
- url이 새 url이면 frontier에 추가

### c. 구성요소

#### i. url frontier

- download 해야 하는 모든 url을 보관하는 data structure
- seed set page로 부터 시작, web에 대한 폭 넓은 탐색을 수행
- 순회: FIFO, queue
- 각 서버에 crawling 작업스레드가 여러개 있다고 가정, hash function 이 각 url을 crawling하는 서버에 mapping 한다고 가정
- 분산 url frontier를 디자인 하는 동안 다음의 요구사항을 염두에 두어야 함
  - ✓ crawler는 많은 페이지를 다운로드 하여 서버에 과부하가 가해서는 안됨
  - ✓ 웹서버에 연결하는 computer는 소수여야 함
- 구현방안
  - ✓ 각 서버에 queue는 하위 queue 모음을 가질 수 있음
  - ✓ 각 작업자 thread에 별도의 하위 queue가 있으면 crawling을 위해 url을 제거
  - ✓ 새 url을 추가하는 경우 url이 있는 queue 하위 queue는 url 정식 host name에 의해 결정
  - ✓ hash function은 각 host name을 thread 번호로 mapping 가능
  - ✓ 최대 하나의 작업자 thread가 지정된 웹서버에서 문서를 download하고 queue 사용하여 web server에 과부하가 걸리지 않도록 함

#### ii. fetcher module

- 목적: http와 같은 적절한 network protocol을 사용하여 지정된 url에 해당 문서를 download

- web master는 robot.txt를 만들어 웹사이트의 특정부분을 crawler가 방문하지 않도록 제한
- 요청이 있을 때마다 이 파일은 다운로드 하지 않기 위해 crawler의 http 프로토콜 모듈을 로봇의 제외 규칙에 대한 고정 크기 캐시 매팅 호스트 이름을 유지 할 수 있음

iii. dis (document input stream)

- 여러개의 처리 모듈에서 동일한 문서를 처리 할 수 있음
- 문서를 여러번 download 하지 않기 위해 dis (document input stream) 라는 추상화를 사용, 문서를 local로 cache
- fetcher가 frontier에서 url을 추출 한 후 작업자는 해당 url을 관련 프로토콜 모듈로 전달하여 network 연결에서 dis를 초기화, 문서에 내용에 포함시킴

iv. document dedupe test

- 한 문서가 여러개의 url을 제공, 서버가 mirroring 되는 경우 -> 동일한 문서를 여러번 다운로드 -> 중복 제거 테스트
- 처리된 모든 문서는 64bit checksum(MD5, SHA)을 계산 database에 저장 한 후 이를 통해 중복 페이지 여부를 판별

v. url filters

- crawler가 특정 website를 무시 할 수 있도록 blacklist를 만드는데 사용
- 각 url을 frontier에 추가하기 전에 작업자 thread는 사용자 제공 url을 제공 하는데 filter를 정의 할 수 있음

vi. url dedupe test

- 저장공간을 절약하기 위해 url을 text 형태가 아닌 checksum 형태로 저장할 수 있음
- db의 작업수를 줄이기 위해 모든 thread가 공유하는 각 호스트에 자주 사용되는 url은 cache 저장
- url 저장소에 필요한 저장용량은 얼마인가요?
  - ✓ 15B의 url, 4byte 의 checksum 이라고 가정
  - ✓ 15B \* 4byte = 60GB 필요

vii. checkpointing

- 전체 web crawling을 완료하는데 몇 주가 걸림
- crawler는 오류를 방지하기 위해 정기적으로 상태의 snapshot을 disk에 저장
- 최신 검사점에서 중단되거나 중단된 crawling을 쉽게 다시 시작할 수 있음

7. fault tolerance

- a. crawling server간에 배포하려면 일관된 해싱 (consistent hashing)을 사용해야 함
- b. consistent hashing : 사용 불능 host를 교체, crawling server간 load를 분산시키는데 도움
- c. 모든 crawling server는 정기적으로 check point를 수행하고 FIFO 대기열을 disk에 저장
- d. server가 다운되면 교체할 수 있음
- e. consistent hashing은 load(부하)를 다른 서버로 이동시킴

8. data partitioning

- a. 3가지 종류의 데이터를 처리함
  - i. 방문할 url
  - ii. 중복 제거를 위한 url checksum
  - iii. 중복 제거를 위한 document checksum
- b. 호스트 이름 기준으로 url을 배포하므로 이러한 데이터를 동일한 호스트에 저장 할 수 있음 -> 각 호스트는 방문해야 할 url set, 이전에 방문한 모든 url의 checksum 및 download 한 모든 문서의 checksum을 저장
- c. 각 호스트는 주기적으로 검사를 수행, 모든 데이터 스냅샷으로 원격 서버에 dump -> 서버가 다운된 경우라도 마지막 스냅샷에서 data를 가져올 수 있음

9. crawler traps

- a. 웹에는 crawler trap, spam site 및 은폐된 컨텐츠가 많이 있음
- b. crawler traps - 무기한으로 crawling 하는 url 또는 url 집합
  - i. 의도하지 않은 경우 - 파일 시스템 내에 심볼링 링크가 주기적으로 생성
  - ii. 의도한 경우 - 무한한 웹 문서를 동적으로 생성하는 트랩을 작성

- c. 스팸 방지 트랩 : 스팸머가 전자 메일 주소를 찾는데 사용하는 crawler를 포착하도록 설계, 다른 사이트는 trap을 사용, 검색엔진 crawler를 포착, 검색 등급을 높임

### XIII. Designing Facebook's Newsfeed

1. newsfeed
  - a. website 중앙 위치
  - b. 팔로워의 게시물 표시
  - c. 팔로워 상태, 사진, 비디오, 링크, 앱 활동 및 좋아요 표시
2. requirements and goal of the system
  - a. functional requirements
    - i. 사용자, 페이지, 그룹 팔로우
    - ii. 팔로워 게시물을 기반으로 뉴스피드 생성
    - iii. 피드는 사진, 비디오, text 포함
  - b. non-functional requirements
    - i. real-time generation with minimum latency (2초 이내)
    - ii. 새 게시물을 push 하는 경우는 5초 이내
3. capacity estimation and constraints
  - a. assume
    - i. follow : user 당 300명, 200페이지
    - ii. dau : 300m
    - iii. 평균 타임라인 시청횟수 : 5회
    - iv. 평균 게시물 파일 크기 : 1kb
  - b. estimation
    - i. 뉴스피드 요청횟수
      - $300m * 5 = 1.5b \text{ request/day}$
      - $1.5b / 86400 = 17.5k \text{ request/sec}$
    - ii. memory
      - user data size :  $(300 + 200) * 1kb = 500\text{kbyte/user}$
      - memory usage :  $300m * 500\text{kb} = 150\text{tb/day}$
      - number of server (서버당 100gb memory 보유) :  $150\text{tb} / 100\text{gb} = 1500\text{대}$
4. system apis
  - a. getUserFeed (api\_dev\_key, user\_id, since\_id=none, count=none, max\_id=none, exclude\_replies=none)
    - i. since\_id (number) : 입력된 ID 이후 결과 반환 (since\_id ~ )

- ii. count (number) : 입력된 숫자 만큼의 결과 반환 (max 200)
- iii. max\_id (number) : 입력된 ID 이전 결과 반환 (~ max\_id)
- iv. exclude\_replies (boolean) : 회신에 반환값에 나타나는지 여부
- v. return (json) : 피드항목 목록 리턴

## 5. database design

### a. User

- i. UserID : int, PK
- ii. Name : varchar(20)
- iii. Email : varchar(32)
- iv. DateOfBirth : datetime
- v. CreationDate : datetime
- vi. LastLogin : datetime

### b. Entity

- i. EntityID: int, PK
- ii. Type : tinyint (true, false)
- iii. Category : smallint (2byte, -2^15 ~ 2^15 - 1)
- iv. Name : varchar(20)
- v. Email : varchar(32)
- vi. Phone : varchar(12)
- vii. Description : varchar(512)
- viii. CreationDate : datetime

### c. Follow

- i. UserID : int, PK
- ii. EntityOrFriendID : int, PK
- iii. type : tinyint (true or false)

### d. FeedItem

- i. FeedItemID : int, PK
- ii. UserID : int
- iii. EntityID : int
- iv. Path : varchar(256)
- v. Latitude : int
- vi. Longitude : int
- vii. CreationDate : datetime

- viii. NumberOfLikes : int
- e. FeedMedia
  - i. FeedItemID : int, PK
  - ii. MediaID : int
- f. Media
  - i. MediaID : int, PK
  - ii. Type : smallint (2byte, -2^15 ~ 2^15 - 1)
  - iii. Path : varchar(256)
  - iv. Description : varchar(256)
  - v. Latitude : int
  - vi. Longitude : int
  - vii. CreationDate : datetime

## 6. high level design

### a. 피드생성

- i. 사용자의 게시물, 사용자가 따르는 엔티티 (페이지/그룹/사용자)에서 생성
- ii. workflow (사용자 : jane)
  - jane 이 follow 하는 사용자 ID 및 Entity 검색
  - 검색 결과를 바탕으로 최신/인기/관련 있는 게시물 검색
  - 검색된 feeds를 cache에 저장 후 jane에게 반환

### b. 피드게시

- i. feed 끝에 도달하면 서버로 부터 다음 게시물 (20개) 요청
- ii. 사용자가 온라인 상태일때 새 게시물이 작성된다면?
  - 최신 항복이 있다는 알림 받거나 서버가 자동 push (새 게시물은 순위가 매겨져서 feed에 추가됨)

### c. 구성요소

- i. web server
- ii. application server
  - db에 post 저장
  - feed 검색 후 사용자에게 반환
- iii. metadata database & cache
  - user/page/group 의 metadata 저장
- iv. post database & cache

- post 의 metadata 저장
- v. news feed generation service
- 관련 게시물 수집 -> 순위지정 -> 뉴스피드 생성 -> 캐시에 저장
  - 실시간 update 수신 -> 새 게시물을 타임라인에 추가
- vi. feed notification service
- 사용자에게 새 게시물이 있음을 알림

d. 39.jpg

## 7. component design

### a. 피드 생성

```
( SELECT FeedItemID FROM FeedITEM WHERE UserID in
( SELECT EntityOrFriendID FROM Follow WHERE UserID = <current_user_id>
and type = 0 ) /* 0 : user */
UNION
( SELECT FeedItemID FROM FeedItem WHERE EntityID in
( SELECT EntityOrFriendID FROM Follow WHERE UserID = <current_user_id>
and type = 1 ) /* 1 : entity */
ORDER BY CreationDate DESC
LIMIT 100
```

#### i. 단점

- hot user 문제 : 다수의 게시물을 정렬/병합/순위지정 하기 때문에 많은 친구/팔로워를 가진 사용자는 속도 느림
- 사용자가 페이지를 로드할때마다 타임라인을 실시간 생성 : 느림

#### ii. 해결방안 - 뉴스피드 오프라인 생성

- 타임라인을 미리 생성하여 메모리에 저장하면 efficiency 가 높아짐

- key : UserID

- value

```
struct {
```

```
LinkedHashMap < FeedItemID, FeedItem > feedItems;
```

```
DateTime lastGenerated;
```

```
}
```

- iii. 메모리에 몇 개의 피드 항목을 저장해야 하나요?
    - (ex) 페이지당 게시물 20개, 평균페이지 열람수 10개라면  
 $20 \times 10 = 200$ 개 저장
    - 200개 이후의 게시물은 DB에서 가져옴
  - iv. 모든 사용자의 뉴스피드를 생성하고 메모리 (cache)에 보관해야 하나요?
    - 자주 로그인 하지 않은 사용자 : LRU 기반 캐시 (오랫동안 액세스하지 않은 사용자는 메모리에서 제거)
    - 더 효율적인 방식 : 사용자의 로그인 패턴을 파악 후 최적 예상방안으로 cache 운용  
(ex) 하루 중 어느 시간에 활동하고 어떤 요일에 뉴스피드에 액세스 하는지 등
- b. 피드 게시 : fan-out ( 팔로워 모두에게 게시물을 push 하는 process )
- i. pull (fan-out-on-load)
    - 방식 : 클라이언트가 정기적(1초에 한번)으로 요청, 서버는 바로 응답 해야 함
    - 단점 : 클라이언트에게 새 게시물이 즉시 제공 안됨  
(클라이언트로 부터 요청 받은 후에야만 제공 가능) -> 빈응답 다수 발생 (자원낭비)
  - ii. push (fan-out-on-write)
    - 방식 : http long polling
    - 장점
      - ✓ 새 게시물 즉시 제공
      - ✓ 피드를 가져올 때 follower 목록 check 할 필요 없음 : 읽기 작업 줄어듬
    - 단점
      - ✓ hot user 문제 : 다수의 follower에게 push 해야 하기 때문에 시스템 느려짐
  - iii. hybrid
    - 일반 user : push
    - hot user : pull
  - iv. 모바일 장치에서 실시간 뉴스피드 update는 불필요한 대역폭을 소비 (데이터요금 낭비) : 모바일의 경우 사용자가 선택하도록 유도

## 8. feed ranking

- a. 방식 : 중요도 (좋아요, 댓글, 공유, 최근, 이미지/비디오 유무) 에서 우선순위를 고려하여 가중치 결정 -> 결합 -> 순위 점수 계산
- b. 평가기준
  - i. stickiness (사용자 고착도, 얼마나 오래/자주 사용하는지)
  - ii. retention (유지율)
  - iii. ads revenue (광고수익)
  - iv. etc

## 9. data partitioning

- a. sharding post and metadata : UserID/PostID based shard
  - i. 분산처리가 필요한 이유
    - read : 높은 읽기 부하
    - write : 매일 upload/update 되는 다수의 게시물
  - ii. twitter와 유사
- b. sharding feed data : UserID based shard
  - i. hash function : UserID % number of server
  - ii. 한 사용자의 모든 데이터를 한 서버에 저장
  - iii. 한 서버에 한 사용자의 모든 데이터를 저장하는 것이 가능 : 한 사용자가 500개 이상의 FeedItemID를 저장하지 않기 때문

## XIV. Designing Yelp or Nearby Friends

1. proximity server (근접서버)
  - a. 주변 관광 명소(장소, 이벤트) 검색/저장
2. requirements and goal of the system
  - a. function requirements
    - i. 사용자는 장소를 추가/삭제/업데이트 할 수 있어야 함
    - ii. 위치(경도/위도)가 주어지면 주어진 반경 내에서 모든 인근 장소를 찾을 수 있어야 함
    - iii. 장소에 대한 feedback을 추가하거나 검토 할 수 있어야 함
    - iv. feedback : 사진, text, 등급 등
  - b. non-functional requirements
    - i. 최소한의 대기시간으로 실시간 검색 제공
    - ii. 많은 검색 부하 견딜 수 있어야 함
3. scale estimation
  - a. assume
    - i. 장소 : 500m
    - ii. QPS : 100k queries/sec
    - iii. 성장을 : 장소 수, QPS 20% increase/year
4. database schema
  - a. Location (byte)
    - i. LocationID : 8, PK
    - ii. Name : 256
    - iii. Latitude : 8
    - iv. Longitude : 8
    - v. Description : 512
    - vi. Category : 1, coffee shop, restorant, theater, etc
  - b. Review
    - i. ReviewID : 4, PK, 어떤 장소도  $2^{32}$ 개를 넘지 않는다 가정
    - ii. LocationID : 8
    - iii. Reviewtext : 512
    - iv. Rating : 1 ~ 10
5. system apis - soap, rest api

- a. search ( api\_dev\_key, search\_terms, user\_location, radius\_filter=none, category\_filter=none, sort=none, page\_token )
  - i. search term (string) : 검색어
  - ii. radius\_filter (number) : optional, 미터 단위의 검색 반경
  - iii. sort (number) : optional, 가장 일치하는 (0-기본값), 최소거리(1), 가장 높은 등급(2)
  - iv. page\_token (number) : 반환 page 수 지정
  - v. 반환값 : JSON
    - 검색어와 일치하는 업체 목록에 대한 정보
    - 업체명, 주소, 카테고리, 등급 및 썸네일

## 6. basic system design and algorithm

- a. 개요
  - i. dataset (장소, 리뷰) 을 저장하고 index 생성
  - ii. index를 효율적으로 읽어야 함 - 결과를 실시간으로 사용자에게 리턴, 같은 데이터를 다시 요청할 가능성이 큼
  - iii. 장소의 위치는 자주 바뀌지 않음 - 데이터가 자주 업데이트 되지 않음
- b. SQL Solution
  - i. RDBMS 사용
    - LocationID : PK
    - Latitude 와 Longitude에 index 처리
  - ii. SELECT \* FROM Location WHERE Latitude BETWEEN X-D and X+D and Longitude between Y-D and Y+D
  - iii. 두 점 사이의 거리를 찾기 위해 피타고拉斯 정리 이용
  - iv. 이 쿼리는 얼마나 효율적인가요?
    - 효율적이지 않음
    - 500M 의 Location이 저장될 것으로 예상
    - index가 따로 있기 때문에 두개의 list를 반환, 교차 수행하는 것은 비효율적임
    - 'X-D'와 'X+D', 'Y-D'와 'Y+D' 사이에 너무 많은 Location 이 있을 수 있음
- c. Grid (40.jpg)

- i. 전체지도를 작은 그리드로 분할 - Location 을 더 작은 set으로 그룹화
- ii. 각 grid는 특정 경도 및 위도 범위 안에 있는 모든 장소를 저장
- iii. 장소를 찾는데 단지 몇 개의 격자만 query 하면 됨
- iv. workflow
  - 주어진 위치와 경계를 기준으로 인접 그리드를 찾음
  - 인접 그리드를 통해 근처의 장소를 찾음
- v. Example
  - GridID : 4 byte
  - grid는 고정된 그리드 크기에서 정적으로 정의 되므로 모든 위치 (lat, long) 및 해당 그리드의 그리드 번호를 쉽게 찾을 수 있음
  - 각 위치마다 GridID를 저장하고 Index하면 더 빠른 검색이 가능
  - SELECT \* FROM Location WHERE Latitude between X-D and X+D and Longitude between Y-D and Y+D and GridID in (GridID0, GridID1, GridID2, ..., GridID8)
  - Query 의 runtime 이 빨라짐
- vi. index를 memory에 보관해야 하나요?
  - index를 memory에 저장 : 서비스 성능 향상
  - index를 hash table에 저장 : key - GridID, value - grid 안에 있는 장소 목록
- vii. index를 저장하려면 얼마나 많은 메모리가 필요하나요?
  - assume
    - ✓ 검색 반경을 10마일
    - ✓ 지구 총 면적 약 200M 평방마일
    - ✓  $200M / 10 = 20M$  개 grid 생성
  - estimate
    - ✓  $4 (\text{GridID}) * 20M (\text{grid 갯수}) + 8 (\text{LocationID}) * 500 M (\text{유저수}) \approx 4\text{GB}$
- viii. 많은 Location이 있는 grid는 느리게 실행됩니다. 해결 방안은?
  - grid에 location 이 많은 경우 grid 분할 (동적분할)

- 해결 과제

- ✓ 분할된 grid를 location에 mapping 하는 법
- ✓ 인접한 grid의 모든 grid를 찾는 법

- d. dynamic size grid

- i. 빠른 검색을 위해 grid에 500 개 미만의 location을 둔다고 가정  
-> grid가 제한에 도달하면 동일한 크기의 grid 4개로 나누고 그 사이에 location을 배포

- ii. 위 가정을 적용할 수 있는 data structure는 ? (41.jpg)

- QuadTree
- 각 노드가 4개씩의 자식을 가질 수 있는 Tree

- iii. QuadTree 구축 방안

- 전 세계를 나타내는 하나의 node로 시작
- 4개의 node로 나누고 그 사이에 location을 위치
- 모든 leaf node가 500 미만이 될때까지 node를 4개로 나눔

- iv. 특정 Location에 대한 grid는 어떻게 찾을 수 있나요?

- root node로 시작하여 아래로 검색
- 각 단계에서 현재 방문중인 node에 자식이 있는지 확인
  - ✓ 있는 경우 : 해당 location가 포함된 자식 노드로 이동
  - ✓ 없는 경우 : 해당 node가 원하는 node임

- v. 이웃 grid는 어떻게 찾나요?

- double linked list 이용 : leaf 노드만 location을 포함하므로 모든 leaf 노드를 double linked list로 연결  
-> 인접한 leaf node 사이에서 앞뒤로 반복하여 원하는 위치를 찾음
- 부모 노드를 이용
  - ✓ 각 노드에 pointer (부모를 가르킴) 를 두어 부모에 access 가능하도록 처리
  - ✓ 각 부모 노드에 모든 자식에 대한 포인터가 있으므로 노드의 형제를 쉽게 찾을 수 있음
  - ✓ 부모 포인터를 통해 인접한 그리드에 대한 검색을 계속 확장 할 수 있음

vi. search workflow는 어떻게 되나요?

- 사용자의 위치가 포함된 node를 찾음
- 해당 node에 원하는 장소가 충분하면 사용자에게 반환
- 그렇지 않으면 필요한 반경을 찾거나, 최대 반경을 기준으로 검색 소진 할 때까지 주변 노드 (상위 포인터 또는 double linked list를 통해) 확장

vii. QuadTree를 저장하는데 얼마나 많은 메모리가 필요하나요?

- 각 장소의 LocationID, Latitude, Longitude 만 cache 하는 경우  $24 * 500 \text{ M} = 12\text{GB}$
- number of total grid :  $500\text{M} / 500 = 1\text{M}$  (각 grid는 최대 500개의 location을 갖을 수 있고 500M의 location이 있음)
- leaf node - 1M, location data - 12GB, QuadTree에는 약 1/3의 내부 노드가 있으며 내부 노드에는 4개의 자식 pointer 가 있음. 각 pointer가 8 byte 라면 모든 내부 노드를 저장하는데 필요한 메모리는  $\checkmark 1\text{M} * 1 / 3 * 4 * 8 = 10\text{MB}$
- 필요 메모리 :  $12\text{GB} + 10\text{MB} = 12.01\text{GB}$  - 한 서버로 충분히 가능
- 시스템에 새로운 location을 어떻게 삽입하나요?
  - ✓ DB 및 QuadTree에 해당 location을 삽입
  - ✓ QuadTree가 분산서버에 있는 경우 새 장소의 grid % server 를 찾아서 추가

## 7. data partitioning

### a. sharding based on region

i. location은 같은 지역 (같은 우편번호등) 으로 나눔

ii. 문제점

- hot region 문제 : 해당 지역을 보유하는 서버에 많은 query가 발생 - 성능저하
- 지역이 성장하는 동안 location 을 server에 균일 분포 하는 것이 어려움

iii. 해결방안 : consistent hashing 혹은 데이터 재 파티션

### b. sharding based on LocationID

- i. hash function :  $\text{LocationID \% number of server}$
- ii. 근처 location 찾기 : 모든 서버를 query -> 중앙서버는 각 서버에서 반환된 set를 집계하여 사용자에게 반환
- iii. 다른 파티션에 다른 QuadTree 구조가 있나요?
  - Yes - 모든 파티션의 그리드에 동일한 수의 장소가 있다고 보장 할 수 없기 때문
  - 모든 서버에서 주어진 반경내에서 모든 인접 그리드를 검색하므로 다른 서버의 다른 트리 구조는 문제가 되지 않음

c. **42.jpg**

8. replication and falut tolerance

- a. replication 의 장점 - QuadTree 서버의 복제
  - i. 읽기 traffic을 분산
  - ii. 복제본(슬레이브)를 읽기 트래픽만 제공하는 master-slave 구성으로 사용
  - iii. 모든 쓰기 traffic는 먼저 master로 이동 한 후 slave에 적용
  - iv. slave에 최근 삽입된 장소가 없을 수 있지만 (몇 밀리초 지연) 이는 허용 가능
- b. QuadTree 서버가 죽으면 어떻게 되나요?
  - i. 각 서버의 보조 복제본을 가짐
  - ii. 주서버와 보조서버는 모두 동일한 QuadTree 구조를 갖음
- c. 주 서버와 보조서버가 동시에 죽으면 어떻게 되나요?
  - i. 새 서버를 할당하고 동일한 서버를 동일한 서버에 다시 구축
  - ii. 각각의 서버에 어떤 장소가 보관되어 있는지 어떻게 아나요?
    - brute-force solution : 전체 db를 hash function을 사용하여 서버에 저장될 모든 필수 장소를 파악 - 비효율적이고 느림, 서버가 재구축 되는 동안 서비스 멈춤
  - iii. 그렇다면 Locations과 QuadTree 서버 간에 mapping을 효율적으로 검색 할 수 있는 방법은 무었인가요?
    - 모든 location을 QuadTree 서버에 mapping 하는 역함수(hash function의 역함수)를 만듬
    - QuadTree Index 서버 배치
      - ✓ key : QuadTree Server 번호

- ✓ value : 해당 QuadTree Server에 보관되는 모든 location을 포함하는 HashMap (Location을 빠르게 추가 / 삭제 가능)
- ✓ information server 가 이를 통해 QuadTree를 구축 - LocationID 및 Lat / Long 을 각 위치에 저장
- ✓ QuadTree 서버 자체를 재구성 할때마다 QuadTree Index 서버에 저장해야 하는 모든 장소를 요청가능 - 성능 대폭 향상

d. fault tolerance를 위해 QuadTree Index 서버의 복제본도 있어야 함

#### 9. cache

- a. hot location을 처리하기 위해 cache 도입
- b. Memcached 같은 상용 솔루션 사용 가능
- c. application 서버는 back-end db에 도달하기 전 cache에 해당 작업 영역이 있는지 빠르게 확인 가능
- d. client 의 사용 패턴에 따라 필요한 cache server 수를 조정
- e. 캐시 제거 정책 : LRU (Least Recently Used)

#### 10. load balancing

- a. 위치
  - i. 클라이언트와 웹서버 사이
  - ii. 웹서버와 응용 프로그램 사이
  - iii. 응용 프로그램과 back-end 서버 사이
- b. round-robin approach (초기)
  - i. 구현이 간단하며 overhead가 발생하지 않음
  - ii. 서버가 작동하지 않으면 rotation에서 제외함으로 traffic을 보내지 않음
  - iii. 문제점 : server load를 고려하지 않음 - back-end 서버에 주기적으로 load를 check 하고 이를 기반으로 traffic을 조정하는 지능적인 lb 솔루션이 필요

#### 11. ranking

- a. 특정 반경에서 가장 인기 있는 장소를 어떻게 반환하나요?
  - i. 인기(별 갯수)를 db와 QuadTree에 저장

- ii. QuadTree 각 파티션에 별갯수 상위 100개 장소를 반환 -> aggregator서버는 반환된 모든 장소 중 상위 100 개의 장소를 결정
- b. QuadTree에서 장소를 검색하고 인기를 업데이트 할 수 있지만 많은 리소스가 필요 -> 검색요청 및 시스템 처리량에 영향 -> 시스템 부하가 최소 일때 하루에 한 두번 업데이트 하도록 결정

## XV. Designing Uber backend

1. requirements and goals of the system
  - a. Uber의 간단버전 개발
  - b. 사용자 유형 : drivers, customers
  - c. driver는 현재위치, 승객을 태울 수 있는지를 정기적으로 서비스에 알림
  - d. customer는 근처에 있는 모든 driver를 볼 수 있음
  - e. customer는 탑승 요청 -> 근처 drivers에게 customer가 pickup준비가 되었음을 알림
  - f. driver와 customer가 탑승을 수락하면 여행이 끝날 때까지 서로의 현재 위치를 지속적으로 볼 수 있음
  - g. 목적지에 도착하면 driver는 다음 주행을 위해 여행이 완료되었음을 표시
2. capacity assume
  - a. total
    - i. customer : 300M
    - ii. driver : 1M
  - b. dau
    - i. customer : 1M
    - ii. driver : 500K
  - c. daily rides : 1M
  - d. 모든 활성 driver가 3초마다 현재 위치를 통지
  - e. customer가 승차 요청을 하면 시스템은 driver에게 실시간 연락 가능
3. basic system design and algorithm
  - a. Yelp 과 대부분 동일 하며 QuadTree가 자주 update 되는 것만 다름
  - b. dynamic grid solution의 문제점
    - i. 모든 활성 driver는 3초마다 현재위치를 보고함으로 이를 반영하도록 data 구조를 update 한다면 많은 시간과 resource가 필요
      - driver를 새 위치로 update하려면 driver의 이전 위치 기반으로 올바른 grid를 찾아야 함
      - 새 위치가 현 grid에 속하지 않음 -> 현 grid에 driver 제거 후 올바른 grid로 이동 / 다시 삽입 -> 새 grid가 최대 한계에 도달하면 다시 분할

- ii. 근처의 모든 driver의 현재 위치를 해당 지역의 모든 활성 고객에게 전파 할 수 있는 빠른 메커니즘 필요. 또한 차량이 운행 중일때 driver와 customer 모두에게 차량의 현재 위치를 알려 주어야 함
- c. QuadTree 라도 빠른 update가 보장되진 않음
- d. driver가 자신의 위치를 보고 할때마다 QuadTree를 수정해야 하나요?
- i. QuadTree를 update 하지 않으면 일부 오래된 데이터가 있어 driver의 현재위치가 올바르게 반영되지 않음
  - ii. QuadTree 구축 목적 : 근처의 object를 빠르게 찾는 것
  - iii. 해결방안 - Hash Table
    - 모든 드라이버가 보고한 최신 위치를 hash table에 저장하고 QuadTree를 조금 덜 update
    - dirver의 현재위치가 15초 내에 QuadTree에 반영된다고 가정
    - DriverLocationHT : driver가 보고 한 현재 위치가 저장된 hash table
  - iv. DriverLocationHT 에 필요한 메모리는 얼마인가요?
    - DriverID : 3 (byte, 1M)
    - OldLatitude : 8
    - OldLongitude : 8
    - NewLatitude : 8
    - NewLongitude : 8 (total 35byte)
    - 1M (total driver) \* 35 byte = 35 MB
  - v. 모든 driver로 부터 위치를 수신하기 위해 얼마나 많은 대역폭이 필요하나요?
    - 3 (DriverID) + 8 (Latitude) + 8 (Longitude) = 19 byte
    - 19 byte \* 500K (활성 driver) = 3.5MB (3초마다 수신)
  - vi. DriverLocationHT를 여러 서버에 배포해야 하나요?
    - 확장성, 성능, 내결합성을 위해 여러서버에 배포해야 함
    - driver location server : DriverLocationHT 를 보유
      - ✓ driver 위치 정보를 받는 즉시 server는 모든 관심있는 고객에게 broadcast

✓ driver 위치를 새로 고치기 위해 server는 해당

QuadTree 서버에 알림

vii. driver 위치를 customer에게 어떻게 효율적으로 broadcast 할 수 있나요?

- push model : server는 모든 관련 사용자에게 위치를 push
- publisher/subscriber model
  - ✓ notification service
  - ✓ Uber 앱 실행 -> 근처 driver를 찾기 위해 서버에 query -> driver 목록 반환 전에 목록 내 모든 driver의 위치정보를 구독시킴
  - ✓ driver 위치를 알고자 하는 customer list를 유지 할 수 있으며, 해당 driver에 대한 DriverLocationHT에서 update 가 있을 때마다 driver의 현재위치를 모든 가입 customer에게 broadcast 시킬 수 있음

viii. 이러한 모든 구독을 저장하려면 얼마나 많은 memory가 필요하나요?

- assume
  - ✓ 일일 활성 고객 : 1M, 일일 활성 Driver : 500K
  - ✓ 평균 5명의 customer가 한명의 driver를 구독
  - ✓ 모든 정보를 hash table에 저장
- estimation
  - ✓  $500K * 3 (\text{DriverID}) + 500K * 5 (\text{average customer}) * 8 (\text{CustomerID}) \approx 21\text{MB}$

ix. driver 위치를 customer에게 알리려면 얼마나 많은 대역폭이 필요하나요?

- $5 (\text{average driver}) * 500K = 2.5\text{M}$
- $2.5\text{M} * (3 \text{ as DriverID} + 8 \text{ as latitude} + 8 \text{ as longitude}) = 47.5\text{MB/s}$

x. 알림 서비스를 효율적으로 구현하여 어떤 어떻게 해야 하나요?

- HTTP Long polling
- push notification

- xi. 현재 customer를 위해 new publisher/drivers는 어떻게 추가 되나요?
  - customer가 uber app을 처음 열면 근처의 driver를 subscription 하게 됨
- xii. customer가 보고 있는 지역에 새 driver가 들어오면 어떻게 되나요?
  - 새로운 customer/ driver subscription을 동적으로 추가하려면 customer가 보고 있는 영역을 추적해야 함
    - ✓ solution이 복잡해짐
- xiii. client가 server에서 주변 driver에 대한 정보를 가져오는 경우는 어떤가요?
  - client가 현재 위치를 보내면 server는 QuadTree에서 근처에 있는 모든 driver를 찾아 client로 반환 -> client 드라이버의 현재 위치를 화면에 update
  - client는 5초마다 query 하여 서버로의 왕복 횟수를 제한
  - push 모델 보다 더 단순
- xiv. grid 가 최대한도에 도달하자마자 grid를 다시 분할해야 하나요?
  - 분할하기 전 grid가 한도를 약간 초과하여 커지도록 cushion을 가질 수 있음 (43.jpg)
- xv. “Request Ride”的 workflow는 어떻게 되나요?
  - customer가 탑승 요청
  - aggregator 서버 중 하나가 요청을 받아 QuadTree 서버 근처의 driver를 반환하도록 요청
  - aggregator 서버는 모든 결과를 수집하고 등급별로 정렬
  - aggregator 서버는 최상위 (3명) 드라이버에게 동시에 알림을 보냄 -> 요청을 수락한 driver는 먼저 탈것에 할당, 다른 driver는 취소 요청을 받음 -> 3명의 driver 모두 응답하지 않으면 aggregator는 list에서 다음 3명의 driver에게 탑승을 요청
  - driver가 요청을 수락하면 고객에게 알림

#### 4. fault tolerance and replication

- a. driver 위치 서버 or 알림서버가 죽으면 어떻게 되나요?

- i. 서버의 복제본이 필요하므로 기본 서버가 죽으면 보조 서버가 제어할 수 있음
- ii. 빠른 I/O를 제공할 수 있는 SSD 같은 일부 영구 저장소에 저장 가능 - 주/보조 서버가 모두 죽으면 영구 저장소에서 데이터를 복구

## 5. ranking

- a. 주어진 반경 내에서 최고 등급의 driver를 어떻게 반환 할 수 있나요?
  - i. 주어진 반경 내에서 상위 10개의 driver를 검색하는 동안 QuadTree 각 partition에 최대 등급의 상위 10개의 driver를 반환하도록 요청
  - ii. aggregator server는 다른 partition에서 반환된 모든 driver 중에 상위 10개의 driver를 확인 할 수 있음
- b. addvanced issues
  - i. 느리고 연결이 끊긴 네트워크에서 client를 어떻게 처리하나요?
  - ii. customer가 탑승중일때 연결이 끊어지면 어떻게 하나요? 그러한 시나리오에서 요금청구를 어떻게 하나요?
  - iii. client가 항상 정보를 push하는 것과 server 항상 정보를 push 하는 것을 비교하면 어떤가요?

## XVI. Designing Ticketmaster

### 1. requirements and goals of the system

#### a. functional requirements

- i. 제휴 영화관이 위치한 다른 도시를 나열
- ii. 도시를 선택하면 해당 도시에서 출시된 영화를 표시
- iii. 영화를 선택하면 해당 영화를 실행하는 영화관과 사용 가능한 공연시간을 표시
- iv. 특정 영화관에서 공연을 선택하고 티켓을 예약
- v. 영화관의 좌석 배치를 보여주고 여러 좌석 선택 가능
- vi. 예약된 / 사용가능 좌석 구분 표시
- vii. 예약 완료를 위해 결제 전 5분 동안 좌석을 확보 할 수 있어야 함
- viii. 기다리는 고객은 공정하고 선착순으로 서비스를 받아야 함

#### b. non-functional requirements

- i. 동시성 (concurrent) 이 높아야 함 - 동일 좌석에 대해 여러번 요청이 있어도 공정하고 우아하게 처리 해야 함
- ii. 티켓 예약 = 금융거래 : 시스템이 안전하고 데이터베이스 ACID를 준수 해야 함

### 2. design considerations

- a. 사용자 인증이 꼭 필요하진 않음
- b. 부분 티켓 주문은 처리하지 않음 (ex) 모든 티켓을 얻거나 아무것도 얻지 못함
- c. 공정성 (fairness) 는 필수
- d. 시스템 남용(abuse)을 막기 위해 한번에 10석 이상 예약하지 못하도록 제한
- e. 시스템은 확장ability) 가능하도록 설계해야 하며 높은 가용성 (availability) 필요 - traffic 급증에 대처하기 위함

### 3. capacity estimation

#### a. assume

- i. views : 3B views/month
- ii. sells : 10M tickets/month
- iii. cities : 500
- iv. average each city has 10 cinemas

- v. average each cinema has 2000 seats
  - vi. shows : 2 shows/day
  - vii. seat booking size : 50 byte (ID, NumberOfSeats, ShowID, MovieID, SeatNumbers, SeatStatus, Timestamp, etc)
  - viii. movie info size : 50 byte
  - b. estimation
    - i.  $500 \text{ (cities)} * 10 \text{ (cinemas)} * 2000 \text{ (seats)} * 2 \text{ (shows)} * (50 + 50) \text{ byte} = 2\text{GB / day}$
    - ii. 3.5 TB / 5years
4. system apis - soap, rest api
- a. SearchMovie (api\_dev\_key, keyword, city, lat\_long, radius, start\_datetime, end\_datetime, postal\_code, includeSpellcheck, results\_per\_page, sorting\_order )
    - i. keyword : 검색할 키워드
    - ii. lat\_long (string) : 검색할 위도 및 경도
    - iii. start\_datetime (string) : 시작 날짜 시간
    - iv. end\_datetime (string) : 종료 날짜 시간
    - v. includeSpellcheck (enum : 'yes' or 'no') : 응답에 맞춤법 검사 제안을 포함
    - vi. results\_per\_page (number) : 페이지당 반환 할 결과수
    - vii. sorting\_order (string) : 검색 결과의 정렬 순서
    - viii. return (json)
  - b. ReserveSeats ( api\_dev\_key, session\_id, movie\_id, show\_id, seats\_to\_reserve [] )
    - i. session\_id (string) : 해당 예약을 추적하기 위한 사용자의 session ID, 예약시간(5분)이 만료되면 서버에서 사용자의 예약이 session\_id를 통해 제거됨
    - ii. show\_id (string) : 예약 표시
    - iii. seats\_to\_reserve (number) : 예약 할 좌석 ID 가 포함된 배열
    - iv. return (json)
      - 예약성공
      - 예약실패 - 전체표시
      - 예약실패 - 재시도, 이미 다른 사용자가 예약을 보유

## 5. database design

### a. consideration

- i. 각 도시는 여러개의 극장을 보유
- ii. 각 영화관은 여러개의 홀을 보유
- iii. 각 영화에는 여러개의 쇼를 보유
- iv. 각 쇼에는 여러개의 예약(seat)을 보유
- v. 사용자는 여러 번 예약(seat) 할 수 있음

### b. ERD 참고 ([44.jpg](#))

## 6. high level design

- a. web server : user session 관리
- b. application server : ticket 관리
- c. db server : data 저장
- d. cache server : 성능 (응답속도) 향상
- e. [45.jpg](#)

## 7. component design - 단일 서버에서 서비스를 제공한다고 가정

### a. ticket reservation workflow

- i. 영화 검색
- ii. 영화 선택
- iii. 시간 선택
- iv. 좌석 수 선택
- v. 선택된 좌석 수가 있는 경우 -> 극장 지도 표시
- vi. 선택된 자석 수가 없는 경우
  - 만석 (사용자에게 오류로 표시)
  - 사용자가 예약하려는 자리는 더이상 없지만 다른 show이 있으므로 극장 지도로 돌아감
  - 대기페이지로 이동
    - ✓ 필요한 좌석 수를 이용 할 수 있게 되면 극장지도 페이지로 이동
    - ✓ 기다리는 동안 모든 좌석이 예약되거나 예약 풀에 적은 좌석이 있으면 오류 메시지 표시
    - ✓ 대기를 취소하고 영화 검색 페이지로 이동
    - ✓ 사용자 세션이 만료되고 영화 검색 페이지로 돌아간 후 최대 1시간 동안 기다릴 수 있음

- vii. 좌석이 성공적으로 예약 된 경우 사용자는 결제 할 수 있는 5분이 주어짐. 5분이 지나면 해당 좌석은 다른 사용자에게 넘어감  
**(46\_x.jpg)**
- b. 서버는 아직 예약되지 않은 모든 활성 예약은 어떻게 추적하며 어떻게 모든 대기를 추적하나요?
- i. ActiveReservationService
    - 모든 활성 예약을 추적하고 만료된 예약을 시스템에서 제거
      - ✓ HashMap 종류의 데이터 구조 필요
      - ✓ HashMap head는 항상 가장 오래된 예약 레코드를 가르킴 -> 시간 초과에 도달하면 예약이 만료
      - ✓ key : ShowID
      - ✓ value : BookingID 및 예약 시작시간이 있는 HashMap
    - 외부 금융 서비스와 협력하여 사용자 지불을 처리
    - 예약이 완료/만료 될 때마다 WaitingUserService는 대기 중인 고객에게 서비스를 제공 할 수 있도록 신호를 받음
  - ii. WaitingUserService
    - 모든 대기중인 사용자 요청을 추적하고 필요한 좌석 수가 사용 가능한 즉시 가장 오래 기다린 대기자에게 좌석을 선택하도록 알림
    - HashTable
      - ✓ 대기중인 모든 사용자를 저장
      - ✓ key : CinemaID
      - ✓ value : UserID, 대기 시작시간이 있는 HashMap
  - iii. Client 는 Long Polling을 사용 하여 예약 상태에 대한 최신 정보 유지가능 - 좌석 사용 가능시 서버는 이 요청을 사용자에게 알림
  - iv. 예약 만료
    - Server에서 ActiveReservationsService는 활성 예약의 만료 (예약 시간 기준) 를 추적

- Client 와 Server 가 동기화 되지 않을 수 있는 Timer  
(만료시간 동안) 표시되므로 서버에서 5초의 버퍼를  
추가하여 손상된 환경으로 부터 보호

#### 8. concurrency

- a. 두명의 사용자가 동일한 좌석을 예약 할 수 없도록 동시성  
(concurrency) 을 처리
  - i. SQL DB Transaction 사용
  - ii. SQL-SERVER를 사용할 경우 Transaction Isolation Levels을  
사용하여 행을 잠그고 update 할 수 있음
  - iii. Transaction 내에서 행을 읽는 경우, 다른 사람이 update 할 수  
없도록 행에 대한 쓰기 잠금을 얻음
  - iv. Transaction 이 성공하면 ActiveReservationService에서  
예약 추적을 시작 할 수 있음

#### 9. fault tolerance

- a. ActiveReservationsService 혹은 WaitingUserService가 충돌하면  
어떻게 되나요?
  - i. ARS가 충돌할 때마다 '예약' table에서 모든 활성 예약을 읽을 수  
있음
    - 예약이 완료될때 까지 status를 reserved(1)로 유지
  - ii. master-slave 구성 : master 가 충돌할때 slave가 대신 할 수  
있음
    - 대기중인 사용자를 db에 저장하지 않으므로 WUS가  
충돌하면 master-slave 설정이 없는 한 data를 복구할  
수단이 없음
- b. db에 fault tolerance (내결합성) 을 갖도록 db를 master-slave  
설정을 함

#### 10. data partitioning

- a. database partitioning
  - i. MovieID로 partitioning 하면 영화의 모든 show 가 단일 서버에  
있게 됨
  - ii. hot movie의 경우 해당 서버에 많은 부하가 걸림
  - iii. 더 낮은 방법은 ShowID를 기반으로 partition 함 - load가  
다른서버에 분산됨

- b. ActiveReservationService 및 WaitingUserService partitioning
- i. web server : 모든 활성 사용자의 세션을 관리, 사용자와 모든 통신을 처리
  - ii. Consistent hashing을 사용하여 ShowID 기반으로 ARS 및 WUS 모두에 application server를 할당
  - iii. 특정 show의 모든 예약 및 대기 사용자는 특정 서버 set에 의해 처리
  - iv. workflow - consistent hashing의 LB를 위해 show에 3개의 서버를 할당한다고 가정
    - 예약을 제거하거나 만료한 것으로 표시하도록 DB를 update
    - 'Show\_Seat' 테이블에서 좌석 상태를 update
    - Linked HashMap에서 예약을 제거
    - 가장 오래 기다린 사용자를 파악하기 위해 해당 show의 대기 사용자를 보유하고 있는 모든 WUS 서버에 메시지를 broadcast - consistent hashing scheme는 이러한 사용자를 보유한 서버를 알려줌
    - 필요한 좌석을 사용할 수 있는 경우 가장 오래된 대기 사용자를 보유한 요청을 처리하도록 WUS 서버에 메시지를 보냄
  - v. 예약을 성공할 때마다 다음과 같은 일이 발생
    - 예약을 보유한 서버는 해당 show의 대기중인 사용자를 모든 서버에 메시지를 전송하여 해당 서버가 사용 가능한 좌석보다 더 많은 좌석을 필요로 하는 모든 대기중인 사용자를 만료시킴
    - 위의 메시지를 수신하면 대기중인 사용자를 보유한 모든 서버가 DB를 쿼리하여 현재 사용 가능한 여유 좌석수를 찾음 - DB cache는 이 쿼리를 한번만 실행하는 데 큰 도움이 됨
    - 사용 가능한 좌석보다 더 많은 좌석을 예약하려는 모든 대기 사용자를 만료 - WUS는 모든 대기중인 사용자의 HashMap을 Iteration 하여 처리

XVII.

1.