3일차에서 할 이야기

- DB Pro Tips
- API Pro Tips

이번 파트에서 할 이야기

- Schema Design에서 유의할 점
- alter table이나 update 쿼리 등 파괴적인 행동에서 유의할 점
- 몇가지 고민들
- Index에 대한 이야기 등

data types

타입을 잘 선택하자

- 잘못 선택한 타입은 쿼리를 느리게 만들고 용량을 낭비함
- 사용되는 타입의 대부분은 문자열, 정수
    - 얘네들만 잘 알아두면 더 좋은 CREATE TABLE문을 만들 수 있다

data types

`CHAR` VS `VARCHAR`

- `CHAR(100)` 에 `'abc'` 가 들어가는 경우 → 계속 100만큼의 용량을 점유
- `VARCHAR(100)` 에 `'abc'` 가 들어가는 경우 → 길이를 3으로 줄이므로 그만큼 용량을 아낌
- `VARCHAR` 는 resize 과정이 포함되므로 insert 시간이 늘어남
- 주민번호, 전화번호 등 길이의 편차가 작은 문자열에는 `CHAR`
- 이름, 주소, 게시글 제목/내용 등 길이의 편차가 큰 문자열에는 `VARCHAR`
- `CHAR` 의 max length는 255, `VARCHAR` 의 max length는 65535
- `CHAR(256)` 이상을 쓰고 싶다 : 어쩔 수 없지만 `VARCHAR` 사용

data types

`VARCHAR(16)` 이나 `VARCHAR(9999)` 나 똑같다?

- `VARCHAR` 는 어차피 resize 될테니 `VARCHAR(16)` 과 `VARCHAR(9999)` 는 차이가 없다?
  - 길이가 2인 문자열만 넣는다? → **resize되므로 insert된 row가 차지하는 용량에는 차이가 없다!**
  - `LENGTH` 함수에 넘겨지거나 값 비교를 하거나 할 때 max length에 해당하는 메모리를 준비
    - `VARCHAR(16)` 은 1바이트의 정수를 준비
    - `VARCHAR(9999)` 는 2바이트의 정수를 준비
    - 때문에 max length가 클수록 메모리를 더 많이 사용할 수 있음 (눈에 띌 정도는 아니지만)

data types

`VARCHAR(16)` 이나 `VARCHAR(9999)` 나 똑같다?

- max length가 큰 문자열은 index로 사용되었을 때 문제를 일으킬 수 있다
  - 사이즈가 커서 index로 등록이 불가능하거나
  - INSERT/UPDATE/DELETE 퍼포먼스가 크게 저하되거나
    - 뒤에서 더 얘기하겠지만 대충 설명하면..
    - index는 데이터를 순차적으로 저장할 수 있도록, 데이터에 변동이 생길 때마다 정렬을 수행
    - 정렬을 위해 데이터를 비교할텐데, 길이가 길면 그만큼 비교가 오래 걸림
- → **가능한 작게 준비하자**

data types

TEXT 타입들

- `TINYTEXT`, `TEXT`, `MEDIUMTEXT`, `LONGTEXT` 존재

```
  TINYTEXT |           255 (2^8-1)  bytes
      TEXT |        65,535 (2^16-1) bytes = 64 KiB
MEDIUMTEXT |    16,777,215 (2^24-1) bytes = 16 MiB
  LONGTEXT | 4,294,967,295 (2^32-1) bytes =  4 GiB
```

- `CHAR`, `VARCHAR` 와 다르게 최대 길이를 지정할 수 없음
- `TEXT` 류 타입들의 동작은 `VARCHAR` 와 유사 : resize를 수행
- `CHAR` 와 `VARCHAR` 는 index에 사용할 수 있지만 `TEXT` 류 타입들은 일부만 등록하거나 해야 함
- 인덱스 없는 상황에선 `VARCHAR` 나 `TEXT` 나 비슷한 속도
  - MySQL – Benchmarking Varchar vs Text
  - 그러나 `VARCHAR` 는 인덱스를 걸수라도 있기 때문에 최적화 여지는 있다
- 결론 : `TEXT` 류 타입들은 제약이 많으므로 되도록 `CHAR`, `VARCHAR` 를 먼저 검토

data types

INT 타입들

- `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT`, `BIGINT` 존재

```
  TINYINT |          1 byte  (8 bit)  = -128 ~ 127
 SMALLINT |          2 bytes (16 bit) = -32768 ~ 32767
MEDIUMINT |          3 bytes (24 bit) = -8388608 ~ 8388607
      INT |          4 bytes (32 bit) = -2147483648 ~ 2147483647
   BIGINT |          8 bytes (64 bit) = -2^63 ~ 2^63-1
```

- 나이(age)같은 것에 대충 `INT` 써서 4바이트 쓰는 것보다 `TINYINT`를 쓰면 storage를 아낄 수 있다

data types

INT UNSIGNED

- INT 타입 뒤에 UNSIGNED를 붙이면 음수 표현을 위한 비트를 아끼므로 maximum value를 더 확장할 수 있음
  - e.g. `SMALLINT` : -32768 ~ 32767, `UNSIGNED SMALLINT` : 0~65535
- `INT AUTO_INCREMENT` 를 PK로 사용하는 경우 `UNSIGNED` 를 붙여 2배의 범위를 확보할 수 있음

data types

INT(11)은 뭐유?

- 타입을 `INT` 가 아닌 `INT(11)` 처럼 정의하는 경우가 있음
- **`INT(x)` will make difference only in term of display**
- `INT(x)` 의 `x` 는 최대값을 명시하는 용도가 아님
  - 예를 들어 `INT(3)` 에 `100000` 을 넣어도 정상 작동
- `ZEROFILL` 옵션이 들어가 있을 때 `0` 을 어디까지 채울지를 명시하는 것
  - 만약 `INT(5) ZEROFILL` 컬럼이 있고 여기 `150` 이 들어가 있다면, SELECT 시 `00150` 으로 표시
- `ZEROFILL` 옵션 없이 `INT(x)` 처럼 정의된 컬럼은 `INT` 와 다를 것이 없음

data types

MySQL에 BOOL 타입은 없다

- `**boolean` isn't a distinct datatype in MySQL; it's just a synonym for `tinyint` **
  - `BOOL` 혹은 `BOOLEAN` 이라는 타입은 실제로 존재하는 것이 아님
- `CREATE TABLE` 문에 `BOOL` 혹은 `BOOLEAN` 타입의 컬럼을 명시한 뒤,
- `SHOW CREATE TABLE` 문으로 확인하면 `TINYINT(1)` 로 치환되는 것을 확인할 수 있음
- → `-128 ~ 127` 범위의 값이 허용되므로, 1, 0만 허용되는 것을 원했다면 문제
- 대신 `BIT(1)` 타입을 쓰는 것을 추천

data types

size 정하기

- 표준에 근거
  - 이메일 : RFC 3696에 따라 최대 길이는 320자 – `VARCHAR(320)`
  - 주민등록번호 : 앞 6글자 + 뒤 7글자 – `CHAR(13)`
- 기획에 근거
  - 닉네임 : 최대 20글자로 설정할 수 있다 – `VARCHAR(20)`
- INSERT 전처리가 들어가는 경우
  - 비밀번호는 암호화(hashing)를 거치면서 길이가 fix될 수 있음 – `CHAR` 사용 가능
  - 전화번호는 hypen을 제외 – `CHAR(13)` 대신 `CHAR(11)` 사용 가능

data types

size 정하기

- 국제화를 위한 노력
  - 전화번호를 hypen(-) 제외하고 `010xxxxxxxx` 포맷으로 단정짓고 `CHAR(11)`
    - 전화번호가 `010xxxxxxxx` 패턴이 아닌 외국 유저는 어떡한담?
    - 국가번호(한국은 +82)가 있어서 이게 붙은 번호가 실제로 유효한 번호
    - → `+8210....` 처럼 국가번호를 포함한 전화번호를 받게 만든다면 더 낫다
    - E.164 스펙에 따르면 국가번호를 포함해 전화번호 최대 길이는 15글자(e.g. `+8210...`)
      - → `VARCHAR(15)`
- 애매한 것
  - 이름, 주소, 나이 등 maximum을 쉽게 알기 어려운 것
  - → 적당히 크게 잡아놓고 나중에 문제가 생기면 대응
    - 참고할 것을 최대한 찾자 : What is a reasonable length limit on person "Name" fields?
  - 또는 기획에서 limitation 걸어둔 것이 있다면

key

자연(natural) key와 대체(surrogate) key

- Primary Key로 무엇을 사용할 것인가?
  - 사용자의 email을 PK로 둘 수도 있고, 따로 `INT AUTO INCREMENT` 같은 것을 PK로 둘 수도 있음
  - 각각을 자연 key, 인조 key라고 함
- 자연 key
  - 그 자체로 unique한 것
    - e.g. 사업자등록번호, 전화번호, 국가코드, 이메일 등
  - 중복이 있는 대상은 생각보다 많음 : 주민등록번호, 자동차 번호
- 대체 key
  - 로직 상으로 직접 만들어주는 key
    - e.g. UUID, AUTO INCREMENT INT 등

key

자연(natural) key와 대체(surrogate) key

- 대체 key는 언제 쓰는가?
  - 자연 key로 할만한 데이터가 없는 경우
    - `게시글` 은 자연 key로 쓸만한 것이 따로 없으니 만들어줘야 함
  - unique한 데이터가 있지만 자연 key로 쓰기 불안한 경우
    - `전화번호` 는 unique하지만 개인정보 보호를 위해 추후 받지 않게 될수도 있음
    - `전화번호` 는 사용자가 변경을 원할 수 있음
      - → PK를 변경하는 것은 굉장히 까다로움
        - FK로 연결되어 있는 것을 다 변경해줘야 하나,
        - `ON UPDATE CASCADE` 설정이 안 되어있는 경우가 있다면 어려워짐
- 결론 : 자연 key를 사용하기로 너무 쉽게 결정하지 말자
- 팁 : 자연 key일 것 같은 데이터도 '정말로 unique할까? ' 알아보기

key

AUTO INCREMENT INT PK의 이름은 무엇이 좋을까?

- `AUTO_INCREMENT` 옵션이 들어간 `INT` 타입의 컬럼은 인조 key의 대표적인 사례
- 사람 따라 다르게 네이밍함
  - id, pk, seq, ...
- `id` 가 많이 사용되는 모습이지만 취향 차이
- **하나로 정하고 통일하자**

고민

time 데이터는 어떻게 저장할까?

- 회원가입 시각, 게시글 작성 시각 등
- 타입은 `DATETIME` 을 쓰기로 쉽게 결정하지만 어느 timezone을 따를지도 생각해줘야 함
  - 추천 : UTC 기준
- 데이터는 지역화시키기보다, 범용적인 standard에 맞추는 것이 좋음
- 사용자의 위치에 따라 timezone을 바꿔 보여주는 방법?
  - 참고 : ISO 8601

고민

time 데이터는 어떻게 저장할까? 〉 unix timestamp

- unix timestamp : 1970-01-01 00:00:00 UTC로부터의 경과 시간을 초 단위로 표현한 것
- e.g.
  - 1970-01-01 01:00:00 UTC는 unix timestamp로 `3600`
  - 2021-10-10 09:00:00 UTC는 unix timestamp로 `1633899600`
- `DATETIME` 타입의 컬럼에 들어간 값은 겉으로 보기엔 어느 timezone을 썼는지 알 수 없음
  - `2021-10-10 15:00:00` : `UTC` 일수도 `Asia/Seoul` 일수도 `america/los_angeles` 일수도
- `TIMESTAMP` 타입의 컬럼에 unix timestamp 값을 넣는 것도 좋은 방법
- **주의할 점 : unix timestamp는 `1970-01-01 00:00:00 UTC` 이전의 시각을 표현할 수 없음**

고민

varchar Fields – Is a Power of Two More Efficient?

- 최대 500글자의 문자열을 저장하고 싶은 경우 `VARCHAR(500)`
- 그런데 `VARCHAR(512)` 와 같이 2의 거듭제곱에 해당하는 size를 설정하는 경우도 종종 있음
- size에 2의 거듭제곱을 사용했을 때 DBMS가 최적화하는 것은 **전혀 없다**
  - 오히려 낭비(`CHAR(10)` 이면 될 거 `CHAR(16)` 을 쓰면 그만큼 불필요하게 스토리지를 소모)
- 단, 길이를 잴 때 몇 byte의 정수를 준비하느냐에 차이는 있음
  - `VARCHAR(255)` 대상으로는 1바이트의 정수를, `VARCHAR(256)` 은 1바이트(8비트)로 표현할 수 없으므로 2바이트의 정수를 준비
  - `VARCHAR(32767)` 은 2바이트, `VARCHAR(32768)` 은 3바이트
  - ...

재밌는 트릭들

soft delete

- 게시글 삭제, 유저 탈퇴 등 삭제 operation은 `DELETE` query를 이용
  - hard delete라고 함
- soft delete : 테이블에 `status` , `deleted` 같은 bool 컬럼을 두고, 삭제 operation은 `UPDATE` query
- hard delete에 비해 쉽게 복구 가능함
  - hard delete의 경우 삭제 시점을 모르면 어떤 백업을 올려야할지 알기 힘듬
  - 백업을 올리는 것 자체가 오래 걸림(경험 상 2TB 30분)
- 그러나 query할 때 WHERE절을 추가로 걸도록 신경써줘야 하므로 실수가 발생할 가능성 있음
- 논쟁이 꽤 있었음 : Are soft deletes a good idea?

재밌는 트릭들

ON UPDATE CURRENT_TIMESTAMP

- 가장 최근 업데이트 시각을 기록하기 위해 `DATETIME` 타입의 컬럼을 추가하는 경우가 있음
- 컬럼에 `ON UPDATE` 라는 옵션을 통해, row가 update되었을 때 평가되는 표현식을 담을 수 있다

```
CREATE TABLE `example` (
                ...
                updated_at DATETIME ON UPDATE CURRENT_TIMESTAMP
);
```

- 굳이 로직 단에서 현재 시각을 set할 필요 없이, 자동으로 update되므로 유용함

재밌는 트릭들

DROP TABLE보다 ALTER TABLE RENAME TO

- 서비스가 성장하며 어떤 테이블은 필요없어지게 될 수 있음
- `DROP TABLE` 하는 것도 방법이지만 복구가 어렵다
- `ALTER TABLE ... RENAME TO ...` 문을 이용해 이름만 바꿔서 문제가 없는지 지켜보자
  - 문제가 있으면 다시 rename해주면 되는거고
  - 문제가 없으면 그때 DROP 진행

Online DDL

LOCK은 언제나 우리를 괴롭힌다

- Lock?
  - 어떤 query가 진행되는 동안 테이블의 특정 영역, 혹은 테이블 전체를 대상으로 읽기나 쓰기가 불가능하게 되는 것
    - UPDATE 쿼리의 대상이 되는 row는 다른 트랜잭션이 SELECT만 가능함
  - Lock이 발생하는 쿼리도 있고, 아닌 것도 있음
    - → `record lock gap lock` , `mysql lock` 같은 키워드!
- `ALTER TABLE` 에서 대표적으로 하는 일
  - `ADD COLUMN` , `MODIFY COLUMN` , `DROP COLUMN`
  - → 모두 테이블 전체에 Lock을 발생시킴
- `ALTER TABLE` 문은 테이블 사이즈에 비례해 오래 걸린다
  - 플랜비의 사례) row 200만개(4GB) 테이블의 DATETIME 타입 컬럼을 index로 추가하는 데 7분 걸림

Online DDL

LOCK은 언제나 우리를 괴롭힌다

- 그럼 어떻게 하죠?
  - Online DDL!
- ALTER TABLE문 뒤에 `ALGORITHM`, `LOCK` 옵션을 넣으면 됨

```
ALTER TABLE tbl_name ADD COLUMN column_name ..., ALGORITHM=INPLACE, LOCK=NONE;
```

- 기본적인 원리
  i. 테이블의 COPY를 만들어서 여기에 ALTER TABLE 동작 수행
  ii. 그동안 들어오는 INSERT/UPDATE/DELETE는 원본에 반영하기도 하며 로그를 따로 모아둠(row log buffer라고 함)
  iii. ALTER TABLE이 종료되면 (2)에서 모아뒀던 row log buffer의 내용을 반영
  iv. 이렇게 만들어진 새로운 테이블로 참조를 바꿔치기!
- 모든 조작에 대해 유효한 것은 아님
  - 버전 따라서도 다르니 공식 문서 참조

Online DDL

DDL이 아닌 곳에서도 LOCK이?

- 플랜비의 사례 2
  i. 어떤 기능에 날짜(day)까지만 제공하던 것을 초(second) 단위까지 제공해주도록 해야 했음
  ii. 기존 `DATE` 타입 컬럼을 `DATETIME` 으로 변경하면 API response format이 바뀌어서 하위호환성이 망가짐
  iii. 그래서 새로운 `DATETIME` 타입 컬럼을 만들기로 결정하고 Online DDL 잘 해서 멋지게 컬럼 추가 완료
  iv. 요 컬럼에 기존 `DATE` 타입 컬럼의 데이터를 복사하려고 UPDATE문을 돌렸는데 LOCK 발생;;
  v. LOCK이 생길거라고 예상은 했지만, 이미 update 처리가 완료된 row는 LOCK이 해제될 줄 알았음
    - 그러나 트랜잭션이 끝날 때까지는 계속 LOCK을 물고 있음
- UPDATE query는 트랜잭션이 종료될 때까지 그 대상이 되는 row 전체에 LOCK을 건다
  - 앞의 사례와 같은 backfill 작업은 row 수십~수백개 정도씩 잡아서 작은 UPDATE를 여러번 하는 것이 좋음

Index

Index는 무엇인고..

- 막 흩뿌려진 dataset보다 정렬된 dataset은 값을 찾기 쉽다

```
a = [5, 4, 7, 8, 1, 3, 2, 6]
b = [1, 2, 3, 4, 5, 6, 7, 8]
```

- 여기서 `3` 을 찾고 싶다면?
  - `a` 의 경우 : 값에 규칙성이 없으므로 첫 번째 값부터 확인해가며 순차 탐색
  - `b` 의 경우 : 값이 정렬되어 있으므로 이진탐색 가능
  - `Scattered read vs Sequential read`
- 무엇과 같은지 보아하니
  - `SELECT * FROM table WHERE id = 3;`

Index

Index는 무엇인고..

- Index는 데이터 변경마다 정렬을 수행해서, 나중에 SELECT를 빠르게 되도록 한다
  - INSERT 속도가 느려지고 + 정렬된 데이터를 저장하기 위해 추가적인 스토리지가 필요
  - Index의 여부는 테이블이 커질수록 SELECT 퍼포먼스에 영향을 많이 준다
  - 플랜비의 사례 3
    - row 15만개 테이블에서 Index 안 걸려있는 컬럼 대상으로 WHERE절 걸었더니 115초 걸림
    - Index를 걸고 나서 똑같은 쿼리를 실행해 보니 2초 걸림
- `ALTER TABLE ADD INDEX` 가 오래 걸리는 이유?
  - row 수십, 수백만 개를 다 정렬한다고 생각하면 오래 걸릴수밖에 없다

Index

우리는 알게 모르게 Index를 쓰고 있다

- PK, UQ, FK
  - 자동으로 Index가 걸림
  - MySQL에서 Key == Index
    - What is the difference between "ADD KEY" and "ADD INDEX" in MySQL?
  - Unique constraint만 명시하고 index는 안 건다? → 불가능

Index

Clustered vs Non-clustered Index

- 물리적으로 정렬되냐 vs 논리적으로 정렬되냐
- Clustered Index
  - Primary Key가 대표적인 clustered index
  - 모든 테이블의 데이터는, clustered index에 해당하는 컬럼의 값으로 정렬해 그대로 disk에 저장
    - 하드디스크를 뜯어서 쭉~ 읽으면 실제로 정렬된 상태일 것 (물론 꼭 그렇진 않음 ㅎㅎ)
    - 참고 : 공간 지역성
- Non-Clustered Index (Secondary Index)
  - UQ, FK가 대표적인 non-clustered index, index를 별도로 추가하는 것도 여기에 포함
  - 구성된 데이터는 별도의 공간을 사용
    - 데이터의 모양은 인덱스의 종류에 따라 다름
    - B-Tree, R-Tree, Fractal Tree, ...

Index

Index를 추가하는 경우

- Key들은 자동으로 Index가 만들어짐
- 쉽게 생각하기 : WHERE절에서 자주 쓸 것 같은 컬럼
  - 쿠팡에서 기간(정확히는 연도)을 설정해 주문내역을 불러오는 기능
- 더 깊게 생각하기 : 많은 양의 데이터에 비싼 쿼리를 수행할 것 같은 컬럼
  - LIKE query는 비싸지만 대상 row가 적다면 그렇게 느리지 않음
  - 주문일자에 인덱스를 거는 것은 필요없을 수 있음
    - `WHERE user_id=... AND order_time BETWEEN ... AND ...`
    - user id를 통해서 BETWEEN query가 들어갈 데이터 양을 줄이고 시작할 수 있기 때문
- 그냥 쿼리가 느려지면 추가하도록 하자
  - 참고 : query plan

Index

알아두면 좋은 심화지식

- Index는 정렬 퍼포먼스를 위해, 선형적으로 정렬하는 대신 Tree같은 비선형 자료구조를 사용
  - row가 수십만개 있는데 INSERT 하나 할때마다 그걸 싹 다 정렬하는건 말도 안됨
- Index를 타는 쿼리는 O(1)
  - 왜 그런지는 B-Tree Index를 공부해보면 알 수 있다
  - covering index vs full scan

Index

알아두면 좋은 심화지식

- 컬럼에 Index를 걸어둬도 쿼리에 따라 아예 안 쓰는 경우도 있다
- LIKE query
  - `LIKE '...%'` 의 경우 index를 탐
    - 명시한 키워드로 시작하는 문자열의 모임을 찾고, 거기서 like query 수행
  - `LIKE '%...'`, `LIKE '%...%'` 형태의 텍스트 검색
    - index를 타지 않음
    - `FULLTEXT` index를 걸어두고 `MATCH()`, `AGAINST()` 쓰면 매우 빠르다
    - How to speed up SELECT .. LIKE queries in MySQL on multiple columns?
    - MySQL 5.7 FullText Search 이용하기 with ngram

정리

- 타입은 작을수록 좋다
- `VARCHAR(2)` 와 `VARCHAR(9999)` 는 resize를 거치고 나면 결국 용량 차이는 없다
- `INT(x)` 는 `ZEROFILL` 옵션이 없을 때 그냥 `INT` 와 같다
- `BOOL` , `BOOLEAN` 은 `TINYINT(1)` 로 해석된다
  - 정말 bool값을 원한다면 `BIT(1)` 을 사용하자
- size는 근거있게 정의하자
- 자연 key를 맹신하지 말자
- PK의 이름은 적당한 것으로 통일하자

정리

- datetime을 저장하려면 UTC 기준으로 하자
- timestamp를 사용하는 경우 `1970-01-01 00:00:00 UTC` 이전은 표현할 수 없다는 점을 유의하자
- 2의 거듭제곱 길이를 사용하는 것은 의미 없다
- soft delete
- `ON UPDATE CURRENT_TIMESTAMP` 알아두자
- `DROP TABLE` 보다 `RENAME` 을 하는 것이 복구하기 좋다
- `ALTER TABLE` 시 ONLINE DDL을 하자
- UPDATE 쿼리를 날리는 경우에도 LOCK을 생각하자

정리

- 인덱스는 순차탐색보다 이진탐색이 더 빠르다는 개념의 연장선
- 인덱스는 INSERT 퍼포먼스 + 스토리지 ↔ SELECT 퍼포먼스를 tradeoff
- clustered index, non-clustered index
- PK, UQ, FK 등 Key들은 자동으로 index에 등록되며, WHERE절에 자주 쓸 것 같으면 따로 등록하자
- DB는 배울 것이 참 많다..
  - 보통, 대충 쓰다가 서비스가 커지고 최적화를 하기 시작하면서 공부하기 시작함
  - 적재된 데이터의 양이 적고, 트래픽이 적으면 Index, Lock, 고립레벨같은 것이 별로 문제가 안 됨
    - 데이터 양이 적으면 full scan해도 빠르게 동작하기 때문
  - 더 알고싶다면 : [영어 PDF] High Performance MySQL, 3rd Edition