Database Systems

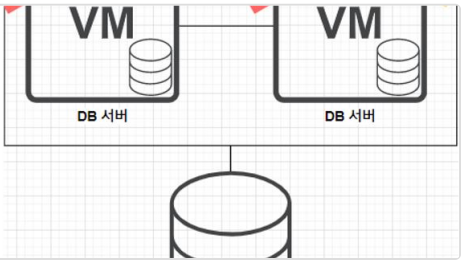
# Index	1
CreatedAt	
Person	
Status	
Tags	Computer Science Database Systems
UpdatedAt	

References

면접 시리즈3 - 데이터베이스

데이터베이스 관리 시스템 으로 여러 사용자가 데이터베이스에 접근하여 사용할 수 있도록 해주는 소프트웨어 파일 시스템의 데이터 중복, 비일관성, 검색 등의 문제 를 해결하기 위해 파일 시스템이 OS마다 다

https://backtony.github.io/interview/2021-12-08-interview-18/



---

1. DBMS

- 데이터베이스 관리 시스템 으로 여러 사용자가 데이터베이스에 접근하여 사용할 수 있도록 해주는 소프트웨어

2. DB를 사용하는 이유

- 파일 시스템의 데이터 중복, 비일관성, 검색 등의 문제를 해결하기 위해
- 파일 시스템이 OS마다 다를 수 있기 때문에 OS에 종속적인 파일 시스템을 이용하는 것은 프로그램의 확장성을 해침

3. 스키마

- 데이터베이스의 구조와 제약 조건에 관한 전반적인 명세를 기술한 메타데이터 집합

4. 테이블

- 행과 열로 이루어진 데이터 집합

5. 행

- 테이블을 구성하는 데이터들 중 가로로 묶은 데이터 셋
- 일반적으로 한 행은 한 객체에 대한 정보를 갖음
- 튜플 또는 레코드라고 부름

6. 열

- 테이블을 구성하는 데이터들 중 세로로 묶은 데이터 셋으로 **속성** 이라고 부름

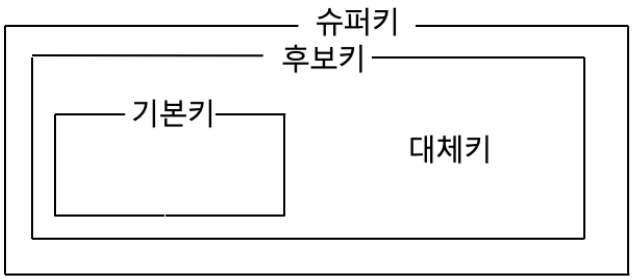
7. 도메인

- 데이터베이스 필드에 채워질 수 있는 값들의 집합
- 예를 들어, 도메인이 1과 10 사이의 정수인데 11이 들어가거나 “고양이”가 들어갈 수 없음

8. 뷰

- 하나 이상의 테이블에서 유도된, 매모리에 물리적으로 존재하지 않는 가상 테이블
- 특정 사용자로부터 특정 속성을 숨기는 기능으로 뷰를 정의하여 그 뷰를 테이블처럼 사용
- 인덱스를 가실 수 없고, 뷰의 정의를 변경할 수 없음
- 기본 키 포함하고 정의할 경우, 삽입, 삭제, 갱신 가능

9. 키



- 검색, 정렬 시 튜플을 구분하는 기준이 되는 속성
- 유일성: 키로 튜플을 유일하게 식별할 수 있음
- 최소성: 튜플을 구분하는데 꼭 필요한 속성들로만 구성

9.1. 후보 키

- 테이블을 구성하는 속성 중에서 튜플을 유일하게 식별할 수 있는 속성들의 부분 집합
    - 기본 키로 사용할 수 있는 속성들
- 모든 테이블은 하나 이상의 후보 키를 가짐(기본키)
- 유일성과 최소성 만족

9.2. 기본 키

- 후보 키 중에서 선택한 Primary Key
- 특정 튜플을 유일하게 식별 가능
- 중복 값과 NULL 불가
- 유일성과 최소성 만족

9.3. 대체 키

- 후보 키가 두개 이상일 때, 기본 키를 제외한 나머지 후보 키

9.4. 슈퍼 키

- 고유하게 식별하는 모든 후보키의 조합하는 키
- 유일성은 만족하지만, 최소성은 만족하지 않음

9.5. 외래 키

- 다른 릴레이션(테이블)의 속성, 참조 관계를 표현하는데 사용하는 키
- 테이블의 열 중 다른 테이블의 기본키를 참조하는 열
- 테이블 간의 연결, 중복 방지, 무결성 유지

10. 트랜잭션

- 데이터베이스의 상태를 변화시키는 하나의 논리적인 작업 단위
- 논리적인 작업의 쿼리 캐수와 관계 없이 논리적인 작업 셋 자체가 100% 적용되거나 아무것도 적용되지 않아야 함을 보장

10.1. 트랜잭션 특징: ACID

- Atomicity(원자성): 트랜잭션을 구성하는 연산 전체가 모두 정상적으로 실행되거나 모두 취소되어야 한다.
- Consistency(일관성): 트랜잭션이 실행을 성공적으로 완료하면 언제나 일관성 있는 데이터베이스 상태로 유지한다.
- Isolation(고립성): 두 개 이상의 트랜잭션이 동시에 발생할 때, 서로의 연산에 영향을 주면 안된다.
- Durability(영구성): 커밋된 트랜잭션의 내용은 영구히 반영된다.

10.2. 트랜잭션 상태

- 활동(Active): 트랜잭션이 실행 중인 상태

- 장애(Fail): 트랜잭션이 실행에 오류가 발생하여 중단한 상태
- 철회(Aborted): 트랜잭션이 비정상적으로 종료되어 Rollback 수행하는 상태
- 부분 완료(Partially Commit): 트랜잭션이 마지막 연산까지 실행했지만, Commit 연산이 실행되기 직전 상태
- 완료(Commmitted): 트랜잭션이 성공적으로 종료되어 commit 연산을 실행한 후의 상태

10.3. 트랜잭션 격리수준

- Read Uncommitted
  - 다른 트랜잭션에서 커밋되지 않은 내용에 접근 가능 (Dirty Read)
  - 락 발생 X
- Read Committed
  - 커밋된 내용만 접근 가능
  - 한 트랜잭션 내에서 검색 결과가 비일관적인 현상 발생 (Nonrepeatable read)
  - 락 발생 X
- Repeatable Read
  - 커밋이 완료된 데이터만 읽을 수 있으며, 트랜잭션 범위 내에서 조회한 내용이 항상 동일함을 보장
  - 일정범위의 레코드를 두번 이상 읽을 때, 첫 번재 쿼리에서 없던 유령 레코드가 두 번째 쿼리에서 나타나는 현상을 Phantom read
  - 락 발생
- Serializable
  - 한 트랜잭션에서 사용하는 데이터는 다른 트랜잭션이 접근 불가능
  - 락 발생

11. Commit

- 트랜잭션이 성공하여 트랜잭션 결과를 영구적으로 반영하는 연산

12. Rollback

- 트랜잭션의 실행을 취소하였음을 알리는 연산
- 트랜잭션이 수행한 결과를 원래의 상태로 원상 복귀시키는 연산

13. 동시성 제어

- 동시성 제어는 동시에 여러개의 트랜잭션이 수행될 때, 트랜잭션들이 DB의 일관성을 파괴하지 않도록 트랜잭션 간의 상호작용을 제어하는 것을 의미한다.

13.1. Locking

- 트랜잭션이 데이터에 접근하기 전에 Lock을 요청해서 Lock이 허락되면 해당 데이터에 접근할 수 있도록 하는 기법
- 종류
  - 비관적 락(Pessimistic lock)
    - 공유락(Shared Lock) : 사용중인 데이터를 다른 트랜잭션이 읽기 허용, 쓰기 비허용
    - 베타락(Exclusive Lock) : 사용중인 데이터를 다른 트랜잭션이 읽기, 쓰기 비허용
    - **데이터 수정 즉시 트랜잭션 충돌을 감지할 수 있다.**
    - 롤백을 개발자가 일일이 하는 것이 힘든 경우, 충돌이 일어났을 때 롤백 비용이 많이 드는 경우, 주문 시에 쿠폰 사용, 알림 제공, 주문서 작성 등의 여러 기능이 한 트랜잭션에 묶여 있는 경우에 적합하다.
  - 낙관적 락(Optimistic lock)
    - 데이터 갱신 시 충돌이 발생하지 않을 것으로 가정하여 락을 걸지 않는 방식 -> **락이 아닌 버전 관리 기능을 통해서 트랜잭션 격리성 관리**
    - Version 컬럼을 별도로 추가해서 충돌 방지 -> Version 정보를 사용하면 최초 커밋만 인정된다.
      - 벌크 연산은 버전을 무시하기 때문에 벌크 연산에서는 버전을 증가시키려면 버전 필드를 강제로 증가시켜야 한다.
    - DB 가 제공하는 락 기능을 사용하지 않고 **JPA 가 제공하는 버전 관리 기능을 사용** -> 애플리케이션에서 제공하는 락
    - **커밋 전까지는 충돌을 알 수 없다.**
    - **충돌이 나면 롤백 처리는 개발자의 몫이다.**
  - 둘의 사용을 판단하는 기준을 읽기와 수정 비율이 어디에 가까운지 보고 판단해야 한다. 수정의 비율이 높다면 Pessimistic을 사용하고 읽기의 비중이 높다면 Optimistic을 사용한다.
- 낙관적락 옵션

- None
  - @Version 이 적용된 필드만 있으면 적용
  - 용도 : 조회한 엔티티를 수정할 때 다른 트랜잭션에 의해 변경(삭제) 되지 않아야 한다.
  - 동작 : 엔티티를 수정할 때 버전을 체크하면서 버즌을 증가시킨다. 만약 버전값이 현재 버전이 아니면 예외가 발생한다.
  - 이점 : 두 번의 갱신 분실 문제를 예방한다.
- OPTIMISTIC
  - @Version만 적용하면 엔티티를 수정해야 버전을 체크하지만 해당 옵션을 추가하면 엔티티를 조회만 해도 버전을 체크한다. 즉, 한 번 조회한 엔티티는 트랜잭션을 종료할 때까지 다른 트랜잭션에서 변경하지 않음을 보장한다.
  - 용도 : 조회 시점부터 트랜잭션이 끝날 때까지 조회한 엔티티가 변경되지 않음을 보장
  - 동작 : 트랜잭션을 커밋할 때 버전 정보를 조회해서 현재 엔티티의 버전과 같은지 검증한다. 아니면 예외 발생
  - 이점 : dirty read와 non-repeatable read를 방지한다.
- OPTIMISTIC_FORCE_IMCREMENT
  - 낙관적 락을 사용하면서 버전 정보를 강제로 증가시킨다.
  - 논리적인 단위의 엔티티 묶음을 관리한다.
  - 예를 들어 게시물과 첨부 파일중 첨부파일이 연관관계 주인인 상태에서 게시물 수정에서 첨부파일만 추가하면 게시물의 버전은 증가하지 않는다. 즉, 게시물은 물리적으로 변경되지 않았지만 논리적으로는 변경되었기 때문에 게시물 버전도 강제로 증가할 때 사용한다.
  - 동작 : 엔티티를 수정하지 않아도 트랜잭션을 커밋할 때 update 쿼리를 사용해 버전 정보를 강제로 증가시킨다. 이때 데이터베이스 버전이 엔티티의 버전과 다르면 예외를 발생시킨다. 추가로 엔티티를 수정하면 수정 시 버전 update가 발생하여 총 2번의 증가가 나타날 수 있다.
  - 이점 : 강제로 버전을 증가시켜 논리적인 단위의 엔티티 묶음을 버전 관리 할 수 있다.

13.2. 갱신 분실 문제

- A와 B가 동시에 제목이 같은 공지사항을 동시에 수정한다고 했을 때 A가 먼저 수정을 완료하고 B가 이후에 완료버튼을 눌렀다면 B의 수정사항만 남게되는데 이것을 갱신 분실 문제라고 한다.갱신 분실 문제는 데이터베이스 트랜잭션의 범위를 넘어서는 문제로 트랜잭션으로만 해결할 수는 없고 3가지 선택 방법이 있다.
  - 마지막 커밋만 인정하기
    - 위에서는 B의 커밋만 인정한다.
    - 기본값
  - 최초 커밋만 인정하기
    - 위에서는 A의 수정을 인정하고 B의 수정이 완료될 때 오류가 발생한다.
  - 충돌하는 갱신 내용 병합하기
    - 사용자 A와 사용자 B의 수정사항을 병합한다.

13.3. 격리 수준 제어(MVCC) 대신 락을 사용하는 이유

- 낙관적 락이나 비관적 락은 다른 트랜잭션이 수정하는 것 자체를 막아버린다. 반면에 MVCC는 레코드에 잠금을 걸지 않고 트랜잭션 격리 레벨에 따라 일관된 읽기를 제공한다. 따라서 두 개의 트랜잭션이 동시에 수정할 때 처음의 수정사항만 반영하도록 하여 갱신 분실 문제를 예방하기 위해서는 락을 사용한다. MVCC가 일관된 읽기를 사용할 수 있는 이유는 변경되기 이전의 내용을 보관하고 있는 undo 로그에서 데이터를 가져오기 때문이다.

13.4. 낙관적 락 vs Repeatable Read

- Repeatable read는 선행 트랜잭션이 종료시까지 다른 트랜잭션이 update, delete하지 못하도록 완전히 락을 걸어버린다.
- 반면에 낙관적인 락은 애플리케이션 단에서 락을 걸지 않아 트랜잭션 자체를 blocking 하지 않으면서도 다른 트랜잭션이 수정하는 것을 막아준다.
- 락을 거는 것 자체가 성능에 영향을 줄 수 있기 때문에 격리 레벨을 조정하는 것보다 낙관적인 락을 사용하는게 더 좋다.

14. 무결성 제약조건

- 데이터베이스의 정확성, 일관성을 보장하기 위해 저장, 삭제, 수정 등을 제약하기 위한 조건
- 개체 무결성: 기본 키는 null, 중복값 불가능
- 참조 무결성: 외래 키는 null이거나 참조 테이블의 기본 키 값과 동일해야 함

15. 조인

- 두 개 이상의 테이블이나 데이터베이스를 연결하여 데이터를 검색하는 방법
- 적어도 하나의 컬럼을 서로 공유하고 있어야 한다.

15.1. 종류

- INNER JOIN
  - 기준 테이블과 join한 테이블의 **중복된 값** 을 보여준다.
  - **교집합** 과 같아고 보면 된다.
- LEFT OUTER JOIN
  - 기준 테이블의 값 + join 테이블과 기준테이블의 중복된 값
  - 기준 테이블은 다 보여주고, join 테이블은 기준 테이블과 중복되는 값만 붙여준다고 생각하면 된다.
  - 기준 테이블과 중복되는 값이 없다면 join 테이블의 내용은 NULL로 표기된다.
- RIGHT OUTER JOIN
  - LEFT OUTER JOIN의 반대이다.
- FULL OUTER JOIN
  - 기준, Join 테이블의 합집합
  - 전부 다 보여주고 빈값들은 NULL로 채워진다.
- CROSS JOIN
  - 크로스 곱이라고도 하는데 모든 경우의 수를 전부 표현해주는 방식이다.
  - 각 테이블의 데이터 개수가 N,M 이라면 결과는 N*M의 데이터 개수가 나온다.
- SELF JOIN
  - CROSS JOIN의 대상이 자기 자신인 것

16. 트리거

- DML이 수행되었을 때, 자동으로 실행되게 정의한 프로시저
  - 프로시저란 쿼리문을 마치 하나의 메서드 형식으로 만들고 어떠한 동작을 일괄적으로 처리하는 용도

17. SQL

- DML : 데이터 조작
  - Select, Insert, Update, Delete
- DDL : 데이터(구조, 객체) 정의
  - Create, Drop(테이블 삭제), Truncate(테이블 데이터 삭제, 테이블 초기화), Alter
- DCL : 권한 제어
  - Grant, Revoke

18. 힌트

- 옵티마이저가 항상 최적의 실행 경로 실행을 보장하지 않기 때문에 직접 최적의 실행 경로를 작성해주는 것

19. 인덱스

- 추가적인 쓰기와 저장 공간사용을 통해 데이터베이스의 검색 속도 향상시키기 위한 방법
- 칼럼의 값(key)와 해당 레코드가 저장된 주소를 키와 값의 쌍으로 인덱스 정의
- 일반적으로 B+ 트리 자료구조를 사용
- 장단점
  - 검색 속도 향상
  - 데이터의 추가, 삭제, 수정의 경우 인덱스도 변경하고 정렬해야 하므로 성능 저하
  - 추가적인 저장 공간 필요
- 알고리즘
  - B 트리 알고리즘 : 컬럼의 값을 변형하지 않고 원래의 값으로 인덱싱, 등호 뿐만 아니라 부등호 연산에도 적용 가능
  - Hash 알고리즘 : 해시값을 이용한 인덱싱

19.1. Cluster 인덱스

- 인덱스로 지정한 컬럼을 기준으로 물리적으로 정렬하는 인덱스
- 한 테이블당 1개 (Primary Key)
- 위에서 언급한 일반적인 인덱스로 검색속도는 빠르지만, 입력,수정,삭제는 느림

19.2. Non-Cluster 인덱스

- 데이터 자체는 정렬되지 않고, 인덱스값을 기준으로 정렬하여 새로 인덱스 페이지를 만드는 인덱스
- 새로 인덱스 페이지를 만들고 리프 페이지에 실제 데이터가 위치한 주소를 가리킴
- 한 테이블 당 여러 개 가능
- 검색 속도는 Cluster에 비해 느리지만, 입력,수정,삭제가 빠름

19.3. 멀티 인덱스

- 두개 이상의 필드를 조합하여 생성한 인덱스
- 그냥 두개 인덱스로 조회하면 되지 않을까?
  - mysql은 단일 쿼리를 실행할 때 **하나의 테이블 당 하나의 인덱스만 사용** 할 수 있다.
  - 둘중 인덱싱데이터 내 행이 적은 것을 먼저 참조하게 된다.

19.4. scan

- Table Full Scan
  - 테이블에 속한 블록 전체를 읽어서 사용자가 원하는 데이터를 찾는다.
- Index Range Scan
  - 인덱스 컬럼이 가공되지 않은 상태로 조건절에 있을 때 수행된다.
  - 인덱스에서 일정량을 스캔하면서 얻은 ROWID를 사용해 테이블 레코드를 찾는다.
- Index Full Scan
  - 인덱스 컬럼이 조건에 없으면 Index Range Scan이 불가능하므로, 옵티마이저는 Table Full Scan을 고려한다.
  - 그런데 대용량 테이블이라 Table Full Scan에 대한 부담이 너무 크면 Index를 활용해야할 필요가 있다.
  - 인덱스의 전체 크기는 테이블의 전체 크기보다 훨씬 적으므로, Index Range Scan을 할 수 없을 때, Table Full Scan보다는 Index Full Scan을 고려한다.

19.5. 자료구조: B 트리, B+ 트리, 해시 테이블, 해시인덱스 vs B 트리 인덱스

- B 트리
  - 이진 트리를 확장해서 많은 자식을 갖을 수 있는 균형 트리
  - key들이 항상 오름차순으로 정렬되어 구성
  - Branch와 Leaf 노드가 key와 data를 저장

- B+ 트리
  - B트리를 확장해서 데이터의 빠른 접근을 위한 인덱스 역할만 하는 비단말 노드를 추가한 트리(리프들이 연결되어 있음)
  - Branch 노드는 key만 저장
  - key들이 항상 오름차순으로 정렬되어 구성
    - 하나의 노드에 더 많은 key를 담을 수 있게 되므로 트리의 높이가 B 트리에 비해 더 낮아진다.(cache hit를 높임)
  - Leaf 노드는 Key와 Data를 저장하고 Linked List로 연결되어 있음(검색에 유용)
    - 풀 스캔 시 B트리는 모든 노드를 확인해야하지만, B+ 트리의 경우 리프노드에 연결된 연결리스트로 선형 탐색이 가능하다.
- 해시 테이블
  - 칼럼 값으로 생성된 해시를 기반으로 인덱스 구현
  - O(1)로 매우 빠름
  - 해시 테이블은 동등 연산에 특화된 자료구조 이기 때문에 select 조건에서 부등호 연산 사용시 성능 저하
  - 동등 비교에서는 효과적
  - 해시는 데이터를 고정된 데이터의 크기로 변환시키는 것을 말함
- 해시 인덱스 vs b 트리 인덱스
  - 동등 비교에서는 해시 인덱스가 효과적
  - 범위 검색에서는 b 트리 인덱스가 정렬되어 있어서 효과적

20. 정규화

- 이상현상이 존재하는 테이블(릴레이션)을 분해하여 여러 개의 테이블(릴레이션)으로 생성하는 과정 이상현상을 제거하기 위해 테이블을 분리하는 작업
  - 이상현상
    - 삽입 이상: 불필요한 데이터를 추가해야만 삽입이 가능한 상황
    - 갱신 이상: 전체의 데이터 중 일부만 변경하여 데이터가 불일치하는 상황
    - 삭제 이상: 삭제로 인해 꼭 필요한 데이터까지 삭제되는 상황
- 1 정규화: 테이블의 컬럼이 하나의 값을 갖도록 테이블을 분해하는 것(도메인이 원자값만 포함)

- 2 정규화: 기본키의 부분집합이 결정자가 안되도록 분리(완전 함수적 종속)
- 3 정규화: 이행종속 제거 (a -> b, b -> c 일때, a->c 가 성립되는 것을 분리, 즉, a-> b랑 b->c 테이블로 분리)
- BCNF: 값을 정하는 결정자가 후보키가 되도록 테이블 분해(모든 결정키가 후보키)

21. 함수의 종속성

- X -> Y: X값을 알면 Y를 알 수 있고, X값에 의해 Y값이 달라지면 Y는 X에 함수적 종속이다.

22. 반정규화

- 반정규화는 성능 향상을 위해 중복, 통합하는 기법으로 조인으로 인한 성능 저하가 예상되는 경우 반정규화를 사용한다.

23. 커넥션 풀

- 클라이언트 요청에 따라 각 애플리케이션의 스레드에서 데이터베이스에 접근하기 위해서는 커넥션이 필요하다. 커넥션 풀이란 미리 일정 수의 Connection을 만들어 풀에 보관해 두는 것 을 의미한다.
    - 사용자의 요청에 따라 Connection을 생성하다 보면 많은 수의 연결이 발생했을 때 서버에 과부하가 걸리게 되므로 방지하기 위함
    - 커넥션 풀을 사용하면 생성, 소멸에 시간 소요가 없어지기 때문에 효율적
    - 한번에 사용할 수 있는 커넥션 수가 제어되어 애플리케이션이 쉽게 죽지 않음

24. RDB vs NoSQL

- 관계형 DB
    - **정해진 스키마에 따라 데이터를 테이블에 저장하는 데이터베이스**
    - **데이터 구조를 보장하고 중복을 피할 수 있다.**
    - SQL을 사용하면 RDBMS에서 데이터를 저장, 수정, 삭제 및 검색 할 수 있다.
    - 데이터는 관계를 통해 여러 테이블에 분산된다.
    - 수직적 확장이 가능하다(단순히 서버의 성능을 향상시키는 것)
    - 사용처
        - **관계를 맺고 있는 데이터가 자주 변경되는 경우**
        - 변경될 여지가 없고, 명확한 스키마가 사용자와 데이터에게 중요한 경우

- NoSQL
  - **스키마가 없거나 느슨한 스키마로 데이터 간의 관계없이 자유로운 형태로 데이터를 저장하는 데이터베이스**
  - 유연하기 때문에 언제든지 데이터를 조정하고 새로운 필드를 추가할 수 있다.
  - 중복을 계속 업데이트해야하는 단점이 있다.
  - **수평적 확장** 으로 **트래픽 분산 및 대용량 처리** 가 가능하다.
  - 레코드를 문서(documents)라고 부른다.
  - Eventual Consistency -Consistency를 보장해주지 못하기 때문에 나온 개념으로, Consistency를 완전히 보장하지는 않지만, 결과적으로 언젠가는 Conssistency가 보장됨을 의미
  - 사용처
    - **읽기를 자주 하지만 데이터 변경은 자주 없는경우**
    - 정확한 데이터 구조를 알 수 없거나, 변경/확장될 수 있는 경우
    - 데이터베이스를 수평적으로 확장해야하는 경우(막대한 양의 데이터를 다루는 경우)

25. Redis, MongoDB, Memcached

- redis
  - **싱글 스레드 인메모리 DB로 key-value 형태로 데이터를 저장** 하며 주로 캐시에 사용
  - **스냅샷, AOF를 통해 백업 가능**
    - 스냅샷 기능을 통해 디스크에 백업하거나, AOF를 통해 명령 쿼리를 저장해두고 서버가 셧다운되면 재실행 방식으로 휘발을 막음
    - AOF :모든 작업을 log 파일에 기록해두고 서버가 재실행되면 순차적으로 연산을 수행하면서 데이터를 복구하는 방식
  - 자료구조 지원
- MongoDB
  - **JSON과 유사한 문서(document)를 사용하여 스키마 없는 데이터를 저장한다.**
  - 물리 디스크 사용
- Memcached

    - **멀티스레드** 를 지원하는 고성능 분산 메모리 캐싱 시스템
    - 데이터 복제를 지원하지 않는다.
- Memcached는 데이터베이스로드를 줄이고 웹 애플리케이션의 속도를 높이는데 유용하고, Redis는 확장 가능한 웹 애플리케이션을 구축하고 고급 데이터 구조가 필요할 때 사용할 수 있다.

26. Elastic Search

- 자바 기반의 오픈소스 검색 엔진
- ELK 스택으로 lastic search, Logstach, kibana 로 묶어서 사용함
- 역색인(inverted index)로 데이터를 저장해서 전문(Full-text)검색 시에 RDBMS보다 성능이 뛰어남
    - 역색인이란 책 뒤에 보면 단어별로 위치가 어딘지 명시해놓은 방식
    - RDBMS는 인덱스가 B+ tree 방식으로 구현되어 있음

26.1. Elastic Search 키워드 검색 vs RDBMS %like%

- **RDBMS**
    - **LIKE의 경우 와일드카드로 시작하는 경우, 인덱스를 사용하지 않기 때문에 전체 검색이 진행되어 느리다.**
    - LIKE의 경우 고정된 문자로 시작되는 경우, 인덱스가 동작한다.
- **Elastic search**
    - 전체 검색이 아니라 역색인을 기반으로 검색하기 때문에 빠르다.

27. 클러스터링

- DB 클러스터링은 DB스토리지는 공유하고 동일한 DB 서버를 다중화(여러 대)하는 방식 이다.동기 방식으로 동기화한다.
    - Active-Active 방식
        - 여러 대의 DB서버가 트래픽을 분산해서 받는다.
        - 여러 대의 서버가 DB 스토리지를 공유하기 때문에 병목이 생길 수 있다.
    - Active-Standby 방식
        - 한쪽은 Standby 상태로 두어 Active 상태의 서버가 죽으면 FailOver되어 전환하는 방식
        - FailOver이 이루어지는 동안 손실이 존재한다.

28. 레플리케이션

- DB 레플리케이션은 DB 서버와 DB 스토리지를 다중화하는 방식 이다.
  - 조회작업은 Slave에서 INSERT, DELETE, UPDATE 작업은 Master에서 수행하면서 트래픽 분산
  - Slave로 데이터를 옮길 때 비동기 방식으로 동작하기 때문에 일관성 있는 데이터를 얻지 못할 수 도 있다.
  - Master 노드가 다운되면 복구 및 대처가 까다롭다.

29. 수직 파티셔닝

- 큰 Table이나 인덱스를 관리하기 쉬운 단위로 분리하는 방식
- 예를 들어, 사람이라는 테이블이 너무 커지게 되면서 사람을 북유럽인, 아시아인, 서유럽인 이렇게 테이블로 분리하는 것이라고 볼 수 있다.
- 장점
  - Insert 시 분리된 파티션으로 분산시켜 경합을 줄임
  - 읽기/쓰기 향상
  - 파티션 별로 백업 및 복구 가능
  - 데이터 전체 검색 시 필요한 부분만 탐색해서 성능 증가
- 단점
  - 인덱스와 테이블을 별도로 파티셔닝 할 수 없다. 즉, 테이블과 인덱스를 같이 파티셔닝 해야한다.
  - 테이블 간 조인 비용 증가

30. 샤딩(수평 파티셔닝)

- 같은 테이블 스키마를 가진 데이터를 다수의 데이터베이스에 분산하여 저장하는 방식
- 데이터를 잘 분산시키기 위해 고려해야할 것이 Shard Key이고 이를 정하는 방식으로 Hash Sharding, Dynamic Sharding, Entity Group 방식이 있다.
- 샤딩은 복잡도가 매우 높아지므로 다른 방식을 우선적으로 고려해야 한다.

31. SQL Injection

- 해커에 의해 조작된 SQL쿼리문이 데이터베이스에 그대로 전달되어 비정상적 명령을 실행시키는 공격 기법

을 말한다. 입력값에 대한 검사, Error Message 노출을 하지 않도록 하여 대응한다.

32. 행의 개수가 많은 테이블 설계

- 파티셔닝 또는 샤딩으로 테이블을 분리하거나 레플리카를 사용해 write, read 트래픽을 분산시킬 수 있다.
- 카디널리티가 높은 것을 기준으로 인덱스 설정
  - 인덱스도 공간을 차지하기 때문에 테이블당 최대 4개가 적당하다.
  - 조건절로 열을 같이 자주 사용하는 것이 있다면 멀티 인덱스를 고려한다.
- 쿼리 성능 향상
  - Entity 대신 DTO를 사용해 조회쿼리 최적화
    - 이미 알고 있는 값인 경우 as 표현식으로 대체
  - 연관된 Entity의 Save를 위해서는 반대편 Entity의 Id값만 있으면 되므로 Id만 조회해서 연관관계를 맺는다.
  - 모든 컬럼이 인덱스로 이뤄진 커버링 인덱스 방식으로 쿼리 작성
  - 더보기(slice) 방식의 페이징 쿼리의 경우 offset를 사용하지 않는 nooffset 방식으로 쿼리 최적화
  - 같은 테이블에 여러 데이터를 insert 할 경우, 아무 세팅도 하지 않았다면 insert 쿼리가 여러번 나가게 되며 이를 배치 insert를 사용해 최적화
    - MySQL의 경우 Identity전략은 Insert를 실행하기 전까지는 ID에 할당된 값을 알 수 없기 때문에 Table 전략을 사용해야 하고 전략을 Identity로 유지하고 싶다면 JdbcTemplate을 사용한다.
  - exists 쿼리 같은 경우 조금만 복잡해지면 JPA 네이밍 쿼리로는 사용이 불가능하고 @Query로 직접 쿼리를 작성해야 하는데 JPQL는 select exist절을 지원하지 않기 때문에 count를 사용해야 함 -> Querydsl의 selectOne과 FetchFirst를 사용하면 JPA 네이밍 쿼리처럼 최적화 가능

33. Statement, PreparedStatement

- Statment는 **SQL문을 실행할 수 있는 객체** 를 의미하고 둘다 첫 실행 시에 아래와 같은 과정을 거쳐 DB에 쿼리가 실행된다.
  1. Parsing(구문 분석)
  2. Compile
  3. Execute

- Statement는 매번 쿼리를 수행할 때마다 3단계를 거치게 되지만, PreparedStatement는 처음 한 번만 3단계를 거치고 이후에는 캐시에 담아서 재사용한다.간략하게 설명하면, statement는 매번 컴파일을 해야하지만, PreparedStatement는 캐싱해서 재사용하기 때문에 PreparedStatement가 성능이 더 좋다. PreparedStatement를 사용할 경우 바인딩에서 문자열을 이어서 사용하게 되면 SQL Injection 공격에 취약하므로 ?를 사용해서 바인딩해야 한다.

34. RabbitMQ, Kafka

- 서비스가 점점 발전하고 규모가 커지게 되면서 서로 통신하고 데이터를 교환하는 방법이 필요해졌다.따라서 필요한 데이터를 담은 “메시지”라는 것을 한쪽에서 생성(produce)하면 다른 쪽에서 소비(consume)하는 구조를 사용하게 되었다.이 역할을 하는 것이 RabbitMQ와 Kafka이다.둘 다 한 곳에서 메시지를 넣어주면 필요한 곳에서 메시지를 꺼내 소비하는 방식으로 되어있지만 차이가 있다.
- RabbitMQ
  - 전통적인 메시지 브로커
  - 생산자와 소비자간의 보장되는 메시지 전달에 초점을 맞춰 브로커 중심적인 특징
  - 컨슈머가 메시지를 가져가면 큐에는 더 이상 남지 않고 사라진다. 따라서 소비자와 메시지 브로커의 결합력이 높아지게 되어 트래픽이 증가하면 수평적으로 확장하는데 어렵다.
  - 이벤트 메시지가 성공적으로 전달되었다고 판단될 경우 이 메시지가 큐에서 삭제되어버리기 때문에 후에 다시 이벤트를 재생하기가 어렵다.
- Kafka
  - 최신 기술인 이벤트 스트리밍 플랫폼
  - 토픽을 컨슈머가 가져간 후에도 이벤트 스트림에서 계속 토픽을 유지하기 때문에 오류 수정이 필요하거나 앱을 리빌드 하는 등의 상황에서 이벤트를 다시 재생시킬 수 있다.
  - 레코드들을 consume 해도 레코들이 삭제되지 않기 때문에 RabbitMQ에 비해 유연하고 느슨한 결합을 가져가게 되고 유연한 확장이 가능해진다.
- 결론
  - RabbitMQ의 경우 kafka에 비해 좀 더 쉽지만 컨슈머와 메시지 브로커간의 결합도가 높기 때문에 트래픽이 작으면서 비즈니스가 후에 확장되지 않을 확률이 높다면 RabbitMQ를 사용하는 것이 좋다.
  - 대규모 트래픽이 예상되고, 추후 확장이 예상된다면 kafka를 선택하는 것이 좋다.