Real MySQL - 아키텍쳐

MySQL 엔진

MySQL 엔진은 클라이언트로부터의 접속 및 쿼리 요청을 처리하는 커넥션 핸들러와 SQL 파서 및 전처리기, 쿼리의 최적화된 실행을 위한 옵티마이저가 중심을 이룬다. 또한 MySQL 은 표준 SQL(ANSI SQL) 문법을 지원하기 때문에 표준 문법에 따라 작성된 쿼리는 타 DMBS와 호환되어 실행될 수 있다.

스토리지 엔진

요청된 SQL 문장을 분석하거나 최적화하는 등 DMBS의 두뇌에 해당하는 처리를 수행하고, 실제 데이터를 디스크 스토리지에 저장하거나 디스크 스토리지로부터 데이터를 읽어오는 부분은 스토리지 엔진이 전담한다.

핸들러 API

MySQL 엔진의 쿼리 실행기에서 데이터를 쓰거나 읽어야 할 때는 각 스토리지 엔진에 쓰기 또는 읽기를 요청하는데, 이러한 요청을 핸들러 요청이라 한다.

```
//핸들러 API를 통해 얼마나 많은 데이터 작업이 있었는지 확인하는 명령어.
SHOW GLOBAL STATUS LIKE 'Handler%';
```

MySQL 스레딩 구조

MySQL 서버는 프로세스 기반이 아니라 스레드 기반으로 작동하며, 크게 Foreground 와 Background 스레드로 구분할 수 있다.

Foreground Thread (클라이언트 스레드)

포그라운드 스레드는 최소한 MySQL 서버에 접속된 클라이언트의 수만큼 존재하며, 주로 각 클라이언트 사용자가 요청하는 쿼리 문장을 처리한다. 포그라운드 스레드는 데이터를 MySQL의 데이터 버퍼나 캐시로부터 가져오며, 버퍼나 캐시에 없는 경우에는 직접 디스크의 데이터나 인덱스 파일로부터 데이터를 읽어와서 작업을 처리한다.

Background Thread

InnoDB는 다음과 같이 여러 가지 작업이 백그라운드로 처리된다.

- 인서트 버퍼를 병합하는 스레드
- 로그를 디스크로 기록하는 스레드
- InnoDB 버퍼 풀의 데이터를 디스크에 기록하는 스레드
- 데이터를 버퍼로 읽어오는 스레드
- 잠금이나 데드락을 모니터링하는 스레드

글로벌 메모리 영역

글로벌 메모리 영역의 모든 메모리 공간은 MySQL 서버가 시작되면서 운영체제로부터 할당된다.

일반적으로 클라이언트 스레드의 수와 무관하게 하나의 메모리 공간만 할당된다.

대표적인 글로벌 메모리 영역은 다음과 같다.

- 테이블 캐시
- InnoDB 버퍼 풀
- InnoDB 어댑티브 해시 인덱스
- InnoDB 리두 로그 버퍼

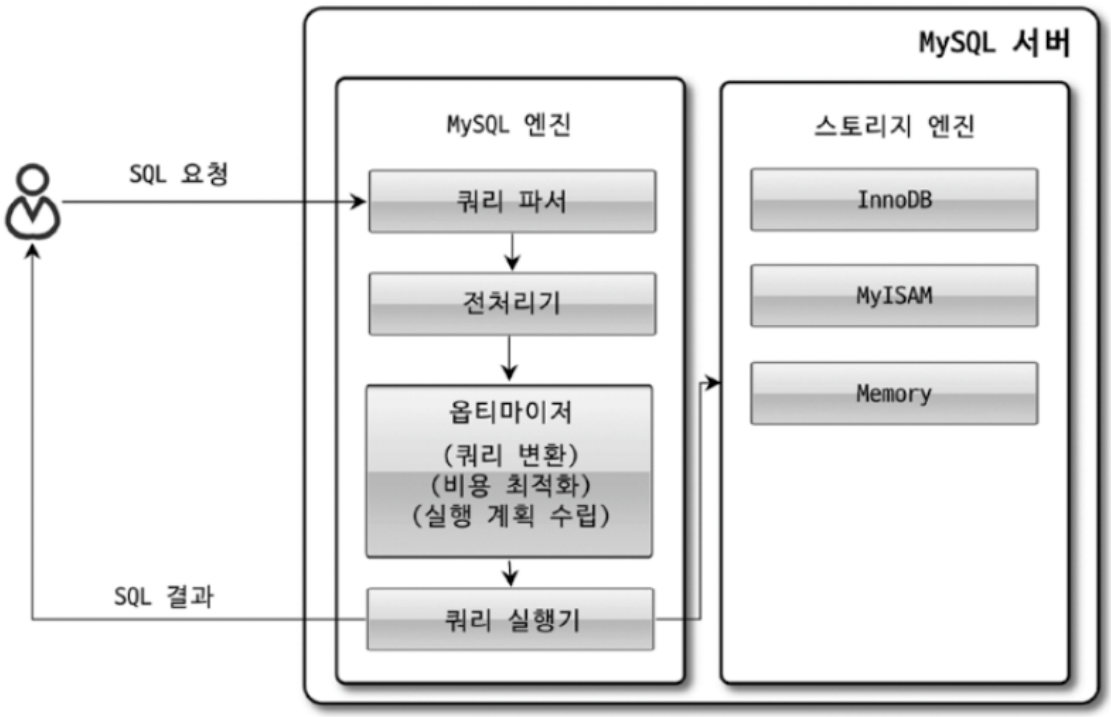
로컬 메모리 영역

세션 메모리 영역이라고도 표현한다. MySQL 서버상에 존재하는 클라이언트 스레드가 쿼리를 처리하는 데 사용하는 메모리 영역이다. 클라이언트가 MySQL 서버에 접속하면 MySQL 서버에서는 클라이언트 커넥션으로부터의 요청을 처리하기 위해 스레드를 하나씩 할당하게 된다 (그래서 클라이언트 메모리 영역이라고도 불린다).

대표적인 로컬 메모리 영역은 다음과 같다.

- 정렬 버퍼 (Sort Buffer)
- 조인 버퍼
- 바이너리 로그 캐시
- 네트워크 버퍼

쿼리 실행 구조



쿼리 파서

쿼리 파서는 사용자 요청으로 들어온 쿼리 문장을 토큰으로 분리해 트리 형태의 구조로 만들어낸다. 쿼리 문장의 기본 문법 오류는 이 과정에서 발견된다.

전처리기

파서 과정에서 만들어진 파서 트리를 기반으로 구조적인 문제점이 있는지 확인한다. 각 토큰을 테이블 이름이나 칼럼 이름, 또는 내장 함수와 같은 개체를 매핑해 해당 객체의 존재 여부와 객체의 접근 권한 등을 확인한다.

옵티마이저

옵티마이저는 쿼리 문장을 저렴한 비용으로 어떻게 가장 빠르게 처리할지를 결정하는 역할이다.

실행 엔진

옵티마이저가 두뇌라면 실행 엔진과 핸들러는 손과 발에 비유할 수 있다.

옵티마이저가 GROUP BY를 처리하기 위해 임시 테이블을 사용하기로 결정했다고 해보자:

1. 실행 엔진이 핸들러에게 임시 테이블을 만들라고 요청
2. 다시 실행 엔진은 WHERE 절에 일치하는 레코드를 읽어오라고 핸들러에게 요청
3. 읽어온 레코드들을 1번에서 준비한 임시 테이블로 저장하라고 다시 핸들러에게 요청

4. 데이터가 준비된 임시 테이블에서 필요한 방식으로 데이터를 읽어 오라고 핸들러에게 다시 요청
5. 최종적으로 실행 엔진은 결과를 사용자나 다른 모듈로 넘김

실행 엔진은 만들어진 계획대로 각 핸들러에게 요청해서 받은 결과를 또 다른 핸들러 요청의 입력으로 연결하는 역할을 수행한다.

핸들러 (스토리지 엔진)

핸들러는 실행 엔진의 요청에 따라 데이터를 디스크로 저장하고 디스크로부터 읽어 오는 역할을 담당한다.

스레드 풀 (Thread Pool)

스레드 풀은 내부적으로 사용자의 요청을 처리하는 스레드 개수를 줄여서 동시 처리되는 요청이 많다 하더라도 MySQL 서버의 CPU가 제한된 개수의 스레드 처리에만 집중할 수 있게 해서 서버의 자원 소모를 줄이는 것이 목적이다.

스레드 그룹의 모든 스레드가 일을 처리하고 있다면 스레드 풀은 해당 스레드 그룹에 새로운 작업 스레드를 추가할지, 아니면 기존 작업 스레드가 처리를 완료할 때까지 기다릴지 여부를 판단해야 한다.

트랜잭션 지원 메타데이터

MySQL 8.0 버전부터는 테이블의 구조 정보나 스토어드 프로그램의 코드 관련 정보를 모두 InnoDB의 테이블에 저장하도록 개선됐다. MySQL 서버가 작동하는 데 기본적으로 필요한 시스템 테이블 또한 모두 InnoDB 스토리지 엔진을 사용하도록 개선했으며, 시스템 테이블과 테이블 딕셔너리 정보를 모두 모아서 mysql DB에 저장하고 있다.

InnoDB 스토리지 엔진 아키텍처

InnoDB는 MySQL에서 사용할 수 있는 스토리지 엔진 중 거의 유일하게 레코드 기반의 잠금을 제공하며, 그 때문에 높은 동시성 처리가 가능하고 안정적이며 성능이 뛰어나다.

프라이머리 키에 의한 클러스터링

InnoDB의 모든 테이블은 기본적으로 프라이머리 키 값의 순서대로 디스크에 저장되며, 모든 세컨더리 인덱스는 레코드의 주소 대신 프라이머리 키의 값을 논리적인 주소로 사용한다.

외래 키 지원

InnoDB에서 외래 키는 부모 테이블과 자식 테이블 모두 해당 칼럼에 인덱스 생성이 필요하고, 변경 시에는 반드시 부모 테이블이나 자식 테이블에 데이터가 있는지 체크하는 작업이 필요하므로 잠금이 여러 테이블로 전파되고, 그로 인해 데드락이 발생할 때가 많으므로 개발할때도 외래 키의 존재에 주의하는 것이 좋다.

```
SET foreign_key_checks=OFF;     //글로벌 변수를 변경한다, 사용하지 말자!
SET SESSION foreign_key_checks=OFF;   //작업중인 세션에만 적용하도록 하자!
```

위 커멘드를 실행하면 외래 키 관계에 대한 체크 작업을 일시적으로 멈출 수 있다. 레코드 적재나 삭제 등의 작업도 부가적인 체크가 필요 없기 때문에 훨씬 빠르게 처리할 수 있다. 하지만 다시 활성화 하기 전에는 반드시 일관성을 맞춰준 후 다시 외래키 체크 기능을 활성화해야 한다.

MVCC (Multi Version Concurrency Control)

아직 COMMIT이나 ROLLBACK이 되지 않은 상태에서 다른 사용자가 SELECT 쿼리로 작업 중인 레코드를 조회하면 어디에 있는 데이터를 조회할까?

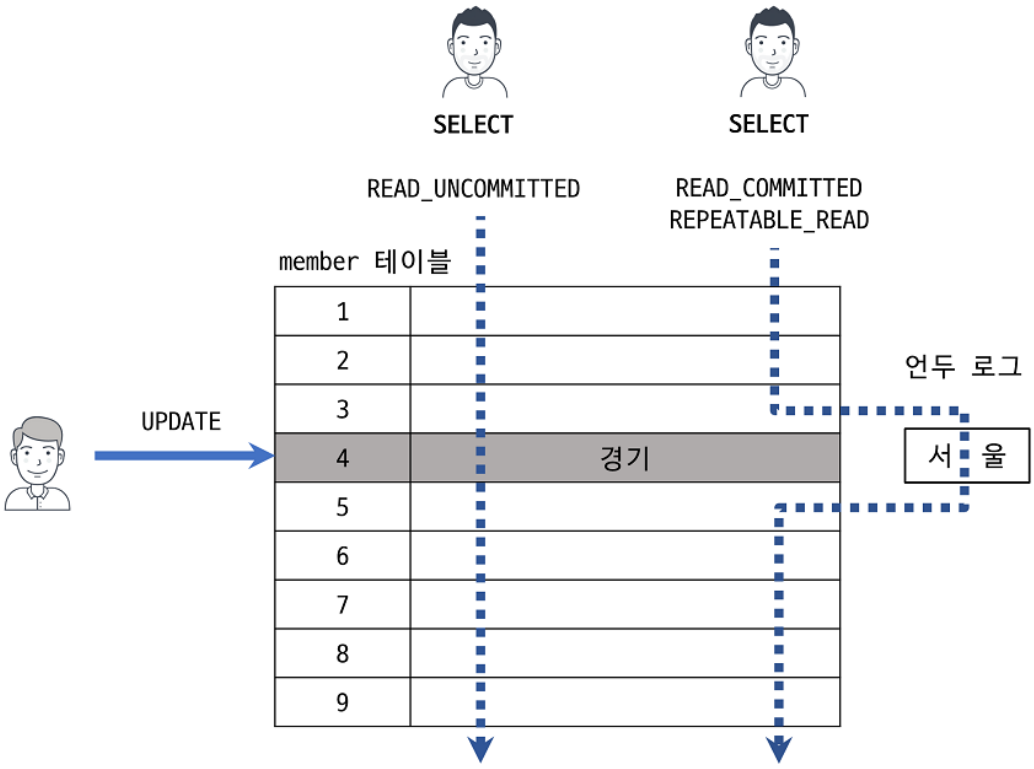
답은 MySQL 서버의 시스템 변수 (transaction_isolation)에 설정된 격리 수준에 따라 다르다는 것이다. 격리 수준이 READ_UNCOMMITTED인 경우에는 InnoDB 버퍼 풀이 현재 가지고 있는 변경된 데이터를 읽어서 반환한다. 즉, 데이터가 커밋됐든 아니든 변경된 상태의 데이터를 반환한다.

그렇지 않고 READ_COMMITTED나 그 이상의 격리 수준 (REPEATABLE_READ, SERIALIZABLE)인 경우에는 아직 커밋되지 않았기 때문에 InnoDB 버퍼 풀이나 데이터 파일에 있는 내용 대신 변경되기 이전의 내용을 보관하고 있는 언두 영역의 데이터를 반환한다.

이러한 과정을 DMBS에서는 MVCC라고 표현한다. 하나의 레코드에 대해 2개의 버전이 유지되고, 필요에 따라 어느 데이터가 보여지는지 여러가지 상황에 따라 달라지는 구조다.

잠금 없는 일관된 읽기 (Non-Locking Consistent Read)

InnoDB 스토리지 엔진은 MVCC 기술을 이용해 잠금을 걸지 않고 읽기 작업을 수행한다. 그렇기 때문에 다른 트랜잭션이 가지고 있는 잠금을 기다리지 않고 읽기 작업이 가능하다. 특정 사용자가 레코드를 변경하고 아직 커밋을 수행하지 않았다 하더라도 이 변경 트랜잭션이 다른 사용자의 SELECT 작업을 방해하지 않는다.



자동 데드락 감지

InnoDB 스토리지 엔진은 내부적으로 잠금이 교착 상태에 빠지지 않았는지 체크하기 위해 잠금 대기 목록을 그래프 형태로 관리한다. InnoDB 엔진의 데드락 감지 스레드가 주기적으로 잠금 대기 그래프를 검사해 교착 상태에 빠진 트랜잭션들을 찾아서 그중 하나를 강제 종료하고, 주로 언두 로그 레코드를 더 적게 가진 트랜잭션이 종료된다.

InnoDB 스토리지 엔진은 상위 레이어인 MySQL 엔진에서 관리되는 테이블 잠금은 볼 수가 없어서 데드락 감지가 불확실할 수 있다. 하지만 시스템 변수중 innodb_table_locks를 활성화하면 테이블 레벨의 잠금까지 감지할 수 있게 되므로 활성화 하도록 하자.

만약 서비스에 동시 처리 스레드가 매우 많아지거나 각 트랜잭션이 가진 잠금의 개수가 많아지면 데드락 감지 스레드가 느려진다. 이렇게 되면 innodb_deadlock_detect를 OFF로 설정해서 비활성화하고 innodb_lock_wait_timeout을 기본값인 50초보다 낮추는것을 권장한다.

InnoDB 버퍼 풀

InnoDB 스토리지 엔진에서 가장 핵심적인 부분으로, 디스크의 데이터 파일이나 인덱스 정보를 메모리에 캐시해 두는 공간이다. 쓰기 작업을 지연시켜 일괄 작업으로 처리할 수 있게

해주는 버퍼 역할도 같이 한다.

버퍼 풀의 크기 설정

레코드 버퍼는 각 클라이언트 세션에서 테이블의 레코드를 읽고 쓸 때 버퍼로 사용하는 공간을 말한다. 커넥션이 많고 사용하는 테이블이 많다면 레코드 버퍼 용도로 사용되는 메모리 공간이 많이 필요해질 수도 있다. 이 크기를 동적으로 조절할때는 적절히 작은 값으로 설정해서 조금씩 상황을 봐 가면서 증가시키는 것이 좋다.

버퍼 풀을 더 크게 변경하는 작업은 시스템 영향도가 크지 않지만, 줄이는 작업은 서비스 영향도가 매우 크므로 가능하면 크기를 줄이는 작업은 하지 않도록 주의하자.

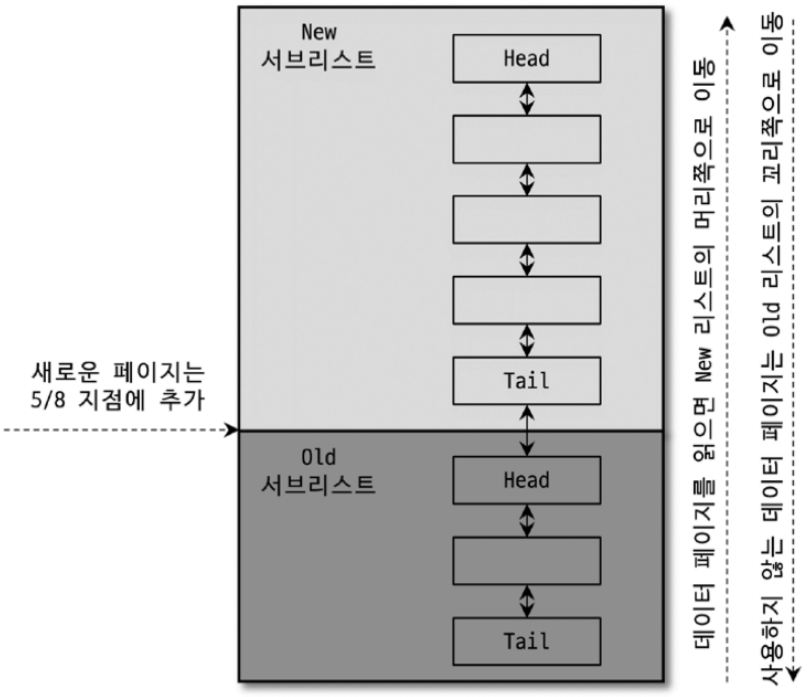
버퍼 풀은 여러개로 쪼개어 관리할 수 있으며, 여러개의 작은 버퍼 풀로 쪼개지면서 개별 버퍼 풀 전체를 관리하는 잠금 자체도 경합이 분산되는 효과를 갖게 된다.
innodb_buffer_pool_instances 시스템 변수를 이용해 버퍼 풀을 여러 개로 분리해서 관리할 수 있는데, 각 버퍼 풀을 버퍼 풀 인스턴스라고 표현한다.

버퍼 풀의 구조

InnoDB 스토리지 엔진은 버퍼 풀이라는 거대한 메모리 공간을 페이지 크기의 조각으로 쪼개어 InnoDB 스토리지 엔진이 데이터를 필요로 할 때 해당 데이터 페이지를 읽어서 각 조각에 저장한다. 버퍼 풀의 페이지 크기 조각을 관리하기 위해 InnoDB 스토리지 엔진은 크게 LRU (Least Recently Used) 리스트와 Flush 리스트, 그리고 Free 리스트라는 3개의 자료 구조를 관리한다.

Free 리스트는 실제 사용자 데이터로 채워지지 않은 비어 있는 페이지들의 목록이며, 사용자의 쿼리가 새롭게 디스크의 데이터 페이지를 읽어와야 하는 경우 사용된다.

LRU는 아래와 같은 구조를 띠고 있는데, 엄밀하게 말하면 LRU와 MRU 리스트가 결합된 형태이다.



LRU 리스트를 관리하는 목적은 디스크로부터 한 번 읽어온 페이지를 최대한 오랫동안 InnoDB 버퍼풀의 메모리에 유지해서 디스크 읽기를 최소화하는 것이다. 그래서 처음 한 번 읽힌 데이터 페이지가 자주 사용된다면 그 데이터 페이지는 MRU 영역에서 계속 살아남게 되고, 반대로 거의 사용되지 않는다면 새롭게 읽히는 데이터 페이지들에 밀려서 LRU의 끝으로 밀려나 결국 제거될 것이다.

Flush 리스트는 디스크로 동기화되지 않은 데이터를 가진 데이터 페이지(이를 '더티 페이지'라고 한다)의 변경 시점 기준의 페이지 목록을 관리한다. 디스크에서 읽은 상태 그대로 전혀 변경이 없다면 Flush 리스트에 관리되지 않지만, 데이터 변경이 가해진 데이터 페이지는 Flush 리스트에 관리되고 특정 시점이 되면 디스크로 기록돼야 한다.

버퍼 풀과 리두 로그

InnoDB의 버퍼 풀은 서버의 메모리가 허용하는 만큼 크게 설정하면 할수록 쿼리의 성능이 빨라진다. 물론 이미 디스크의 모든 데이터 파일이 버퍼 풀에 적재될 정도라면 늘려도 도움이 되지 않는다.

이런 InnoDB 버퍼 풀의 쓰기 기능까지 향상시키려면 버퍼 풀과 리두 로그와의 관계를 먼저 이해해야 한다. 버퍼 풀은 디스크에서 읽은 상태로 전혀 변경되지 않은 클린 페이지와 함께 더티 페이지도 가지고 있다. 더티 페이지는 언젠가는 디스크로 기록돼야 하지만 버퍼 풀에 무한정 머무를 수 있는 것은 아니다. 리두 로그는 1개 이상의 고정 크기 파일을 연결해서 순환 고리처럼 사용하기 때문에, 데이터 변경이 계속 발생하면 리두 로그 파일에 기록됐던 로그 엔트리는 어느 순간 다시 새로운 로그 엔트리로 덮어 쓰인다.

적절한 리두 로그 파일의 크기를 선택하기 어렵다면 버퍼 풀의 크기가 100GB 이하의 MySQL 서버에서는 리두 로그 파일의 전체 크기를 대략 5~10GB 수준으로 선택하고 필요할 때마다 늘려가면서 최적화를 하는 것이 좋다.

플러시 리스트 플러시

InnoDB 스토리지 엔진은 주기적으로 플러시 리스트 플러시 함수를 호출해서 플러시 리스트에서 오래전에 변경된 데이터 페이지 순서대로 디스크에 동기화한다. 이 역할을 수행하는 스레드를 클리너 스레드(Cleaner Thread)라고 한다.

일반적으로 버퍼 풀은 더티 페이지를 많이 가지고 있을수록 디스크 쓰기 작업을 버퍼링함으로써 여러 번의 디스크 쓰기를 한 번으로 줄이는 효과를 극대화할 수 있다. 하지만 여기서 한 가지 문제점이 발생한다. 버퍼 풀에 더티 페이지가 많으면 많을수록 디스크 쓰기 폭발(Disk IO Burst) 현상이 발생할 가능성이 높아지기 때문이다. 이러한 문제를 완화하기 위해 스토리지 엔진에서는 innodb_max_dirty_pages_pct_lwm이라는 시스템 설정 변수를 이용해 일정 수준 이상의 더티 페이지가 발생하면 조금씩 더티 페이지를 디스크로 기록하게 하고 있다.

LRU 리스트 플러시

LRU 리스트에서 사용 빈도가 낮은 데이터 페이지들을 제거하기 위해서 LRU 리스트 플러시 함수가 사용된다. LRU 리스트의 끝부분부터 시작해서 최대 innodb_lru_scan_depth 시스템 변수에 설정된 개수만큼의 페이지들을 스캔하면서 더티 페이지는 디스크에 동기화하고, 클린 페이지는 즉시 Free 리스트 페이지로 옮긴다.

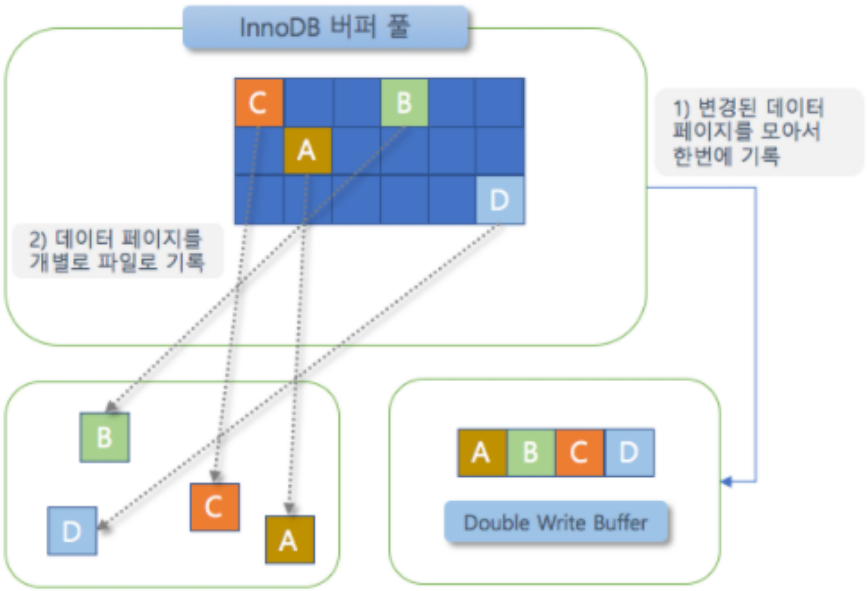
버퍼 풀 상태 백업 및 복구

버퍼 풀의 백업과 복구를 자동화하려면 innodb_buffer_pool_dump_at_shutdown과 innodb_buffer_pool_load_at_startup 설정을 MySQL 서버의 설정 파일에 넣어두면 된다.

Double Write Buffer

InnoDB 스토리지 엔진의 리두 로그는 리두 로그 공간의 낭비를 막기 위해 페이지의 변경된 내용만 기록한다. 이로 인해 더티 페이지를 디스크 파일로 플러시할 때 일부만 기록되는 문제가 발생하면 해당 페이지의 내용을 복구하지 못할 수도 있다. 이렇게 페이지가 일부만 기록되는 현상을 Partial-page 또는 Torn-page라고 하는데, 이런 현상은 하드웨어의 오작동이나 시스템의 비정상 종료 등으로 발생할 수 있다.

이러한 문제를 막기 위해 Double-Write 기법이 이용된다.



우선 InnoDB 스토리지 엔진은 실제 데이터 파일에 변경 내용을 기록하기 전에 더티 페이지를 묶어서 한 번의 디스크 쓰기로 시스템 테이블스페이스의 DoubleWrite 버퍼에 기록한다. 그리고 InnoDB 스토리지 엔진은 각 더티 페이지를 파일의 적당한 위치에 하나씩 랜덤으로 쓰기를 실행한다. DoubleWrite 버퍼의 내용은 실제 데이터 파일의 쓰기가 중간에 실패할 때만 원래의 목적으로 사용된다.

만일 페이지 기록중 운영체제가 비정상적으로 종료되었다면, InnoDB 스토리지 엔진은 재시작될 때 DoubleWrite 버퍼의 내용과 데이터 파일의 페이지들을 모두 비교해서 다른 내용을 담고 있는 페이지가 있으면 DoubleWrite 버퍼의 내용을 데이터 파일의 페이지로 복사한다.

그렇기 때문에 데이터의 무결성이 매우 중요한 서비스에서는 DoubleWrite의 활성화를 고려하는 것이 좋다.

언두 로그 (Undo Log)

InnoDB 스토리지 엔진은 트랜잭션과 격리 수준을 보장하기 위해 DML(INSERT, UPDATE, DELETE)로 변경되기 이전 버전의 데이터를 별도로 백업하고 이를 언두 로그라고 한다.

- **트랜잭션 보장**

  트랜잭션이 롤백되면 트랜잭션 도중 변경된 데이터를 변경 전 데이터로 복구해야 하는데, 이때 언두 로그에 백업해둔 이전 버전의 데이터를 이용해 복구한다.

- **격리 수준 보장**

  특정 커넥션에서 데이터를 변경하는 도중에 다른 커넥션에서 데이터를 조회하면 트랜잭션 격리 수준에 맞게 변경중인 레코드를 읽지 않고 언두 로그에 백업해둔 데이터를 읽어

서 반환하기도 한다.

언두 로그 모니터링

언두 영역은 데이터를 변경했을 때 변경되기 전의 데이터를 보관하는 곳이다. 만약 아래 쿼리가 실행됐다고 해보자:

```
UPDATE member SET name='홍길동' WHERE member_id=1;
```

위 문장이 실행되면 트랜잭션을 커밋하지 않아도 실제 데이터 파일(데이터/인덱스 버퍼) 내용은 '홍길동'으로 변경된다. 그리고 변경되기 전의 값이 '벽계수'였다면, 언두 영역에는 '벽계수'라는 값이 백업되고, 이 상태에서 사용자가 커밋하면 현재 상태가 그대로 유지되고, 롤백하면 언두 영역의 백업된 데이터를 다시 데이터 파일로 복구한다.

언두 로그의 데이터는 이렇게 크게 두가지 용도로 사용된다.

1. 위에서 언급한 트랜잭션의 롤백 대비용.
2. 트랜잭션의 격리 수준을 유지하면서 높은 동시성을 제공하기 위해.

언두 테이블스페이스 관리

언두 로그가 저장되는 공간을 언두 테이블스페이스(Undo Tablespace)라고 한다. 버전 별로 많은 변화가 있었는데 MySQL 8.0으로 업그레이드 되면서 언두 로그는 항상 시스템 테이블스페이스 외부의 별도 로그 파일에 기록되도록 개선됐다.

하나의 언두 테이블스페이스는 1개 이상 128개 이하의 롤백 세그먼트를 가지며, 롤백 세그먼트는 1개 이상의 언두 슬롯을 가진다. 언두 로그 설정값은 기본값을 유지하는게 좋다, 물론 일반적인 서비스에서 이 정도까지 동시 트랜잭션이 필요하진 않겠지만 언두 로그 슬롯이 부족한 경우에는 트랜잭션을 시작할 수 없는 심각한 문제가 발생하기 때문이다.

체인지 버퍼 (Change Buffer)

RDBMS에서 레코드가 INSERT되거나 UPDATE될 때는 데이터 파일을 변경하는 작업뿐 아니라 해당 테이블에 포함된 인덱스를 업데이트하는 작업도 필요하다. 하지만 이 작업은 랜덤하게 디스크를 읽는 작업이 필요하므로 테이블에 인덱스가 많다면 상당히 많은 자원을 소모하게 된다.

그래서 InnoDB는 변경해야 할 인덱스 페이지가 버퍼 풀에 있으면 바로 업데이트를 수행하지만 그렇지 않고 디스크로부터 읽어와서 업데이트해야 한다면 이를 임시 공간에 저장해 두고 바로 사용자에게 결과를 반환해서 성능을 향상시킨다. 이때 사용하는 임시 메모리 공간을 체인지 버퍼라고 한다.

사용자에게 결과를 전달하기 전에 반드시 중복 여부를 체크해야 하는 유니크 인덱스는 체인지 버퍼를 사용할 수 없다. 체인지 버퍼에 임시로 저장된 인덱스 레코드 조각은 이후 백그라운드 스레드에 의해 병합되는데, 이 스레드를 체인지 버퍼 머지 스레드라고 한다.

체인지 버퍼는 기본적으로 InnoDB 버퍼 풀로 설정된 메모리 공간의 25%까지 사용할 수 있게 설정돼 있고, 필요하다면 50까지 사용하게 설정할 수 있다.

리두 로그 및 로그 버퍼

Redo Log는 트랜잭션의 4가지 요소인 ACID 중에서 D(Durable)에 해당하는 영속성과 가장 밀접한 연관이 있다. 리두 로그는 MySQL 서버가 비정상적으로 종료됐을 때 데이터 파일에 기록되지 못한 데이터를 잃지 않게 해주는 안정장치다.

MySQL 서버가 비정상 종료되는 경우 InnoDB 스토리지 엔진의 데이터 파일은 다음과 같은 두 가지 종류의 일관되지 않은 데이터를 가질 수 있다:

1. 커밋됐지만 데이터 파일에 기록되지 않은 데이터
2. 롤백됐지만 데이터 파일에 이미 기록된 데이터

1번의 경우 리두 로그에 저장된 데이터를 데이터 파일에 다시 복사하면 된다, 하지만 2번의 경우에는 리두 로그로는 해결이 불가능하다. 이때는 변경되기 전 데이터를 가진 언두 로그의 내용을 가져와 데이터 파일에 복사하면 된다. 하지만 2번의 경우에도 최소한 그 변경이 커밋됐는지, 롤백됐는지, 아니면 트랜잭션의 실행 중간 상태였는지를 확인하기 위해 리두 로그가 필요하다.

그렇기 때문에 리두 로그는 트랜잭션이 커밋되면 즉시 디스크로 기록되도록 시스템 변수를 설정하는 것을 권장한다. 그래야만 서버가 비정상적으로 종료됐을 때 직전까지의 트랜잭션 커밋 내용이 리두로그에 기록될 수 있고, 장애 직전까지 복구가 가능하기 때문이다.

어댑티브 해시 인덱스

일반적으로 '인덱스'라고 하면 이는 테이블에 사용자가 생성해둔 B-Tree 인덱스를 의미한다. 하지만 여기서 언급하는 Adaptive Hash Index는 사용자가 수동으로 생성하는 인덱스가 아니라 InnoDB 스토리지 엔진에서 사용자가 자주 요청하는 데이터에 대해 자동으로 생성하는 인덱스이다.

B-Tree 인덱스에서 특정 값을 찾기 위해서는 B-Tree의 루트 노드를 거쳐서 브랜치 노드, 그리고 최종적으로 리프 노드까지 찾아가야 원하는 레코드를 읽을 수 있다. 이러한 작업을 동시에 몇천 개의 스레드로 실행하면 컴퓨터의 CPU는 엄청난 프로세스 스케줄링을 하게 되고 자연히 쿼리의 성능은 떨어진다.

이러한 B-Tree 검색 시간을 줄여주기 위해 InnoDB 스토리지 엔진은 자주 읽히는 데이터 페이지의 키 값을 이용해 해시 인덱스를 만들고, 필요할 때마다 어댑티브 해시 인덱스를 검색해서 레코드가 저장된 데이터 페이지를 즉시 찾아갈 수 있다.

해시 인덱스는 '인덱스 키 값'과 해당 인덱스 키 값이 저장된 '데이터 페이지 주소'의 쌍으로 관리되는데, 인덱스 키 값은 'B-Tree 인덱스의 고유번호와 B-Tree 인덱스의 실제 키 값' 조합으로 생성된다.

어댑티브 해시 인덱스가 도움이 되는 경우는 다음과 같다:

1. 디스크의 데이터가 InnoDB 버퍼 풀 크기와 비슷한 경우 (디스크 읽기가 많지 않은 경우)
2. 동등 조건 검색이 많은 경우 (동등 비교와 IN 연산자)
3. 매우 큰 데이터를 가진 테이블의 레코드를 폭넓게 읽는 경우

아래 경우에는 도움이 되지 않는다:

1. 디스크 읽기가 많은 경우
2. 특정 패턴의 쿼리가 많은 경우 (조인이나 LIKE 패턴 검색)
3. 쿼리가 데이터 중에서 일부 데이터에만 집중되는 경우

한가지 확실한 것은 어댑티브 해시 인덱스는 데이터 페이지를 메모리(버퍼 풀) 내에서 접근하는 것을 빠르게 만드는 기능이기 때문에 데이터 페이지를 디스크에서 읽어오는 경우가 빈번한 데이터베이스 서버에서는 아무런 도움이 되지 않는다.

MySQL 로그 파일

MySQL 서버의 상태나 부하를 일으키는 원인을 찾기 위해서는 다음에 설명하는 로그 파일들을 이용하는 것이 좋다.

에러 로그 파일

MySQL이 실행되는 도중에 발생하는 에러나 경고 메시지가 출력되는 로그 파일이다. 에러 로그 파일의 위치는 MySQL 설정 파일(my.cnf)에서 log_error라는 이름의 파라미터로 정의된 경로에 생성된다. 설정 파일에 별도로 정의되지 않은 경우에는 데이터 디렉터리(datadir 파라미터에 설정된 디렉터리)에 .err라는 확장자가 붙은 파일로 생성된다.

제너럴 쿼리 로그파일(General Log)

서버에서 실행되는 쿼리로 어떤 것들이 있는지 전체 목록을 뽑아서 검토해 볼 때가 있는데, 이때는 쿼리 로그를 활성화해서 쿼리를 쿼리 로그 파일로 기록하게 한 다음, 그 파일을 검토하면 된다. 쿼리 로그 파일의 경로는 general_log_file이라는 이름의 파라미터에 설정돼 있다. 또한 쿼리 로그를 파일이 아닌 테이블에 저장하도록 설정할 수 있으므로 이 경우에는 파일이 아닌 테이블을 SQL로 조회해서 검토해야한다.

슬로우 쿼리 로그

MySQL 서버의 쿼리 튜닝은 크게 서비스가 적용되기 전에 전체적으로 튜닝하는 경우와 서비스 운영중에 MySQL 서버의 전체적인 성능 저하를 검사하거나 정기적인 점검을 위한 튜닝으로 나눌 수 있다. 전자의 경우에는 모든 쿼리가 대상이라 모두 튜닝하면 되지만, 후자의 경우에는 어떤 쿼리가 문제의 쿼리인지 판단하기가 상당히 어렵다. 이럴때 슬로으 쿼리 로그가 도움이 된다.

슬로우 쿼리 로그 파일에는 long_query_time 시스템 변수에 설정한 시간 이상의 시간이 소요된 쿼리가 모두 기록된다. 슬로우 쿼리 로그는 쿼리가 실행된 후 실제 소요된 시간을 기준으로 기록되기 대문에 쿼리가 정상적으로 실행이 완료돼야 슬로으 쿼리 로그에 기록된다.