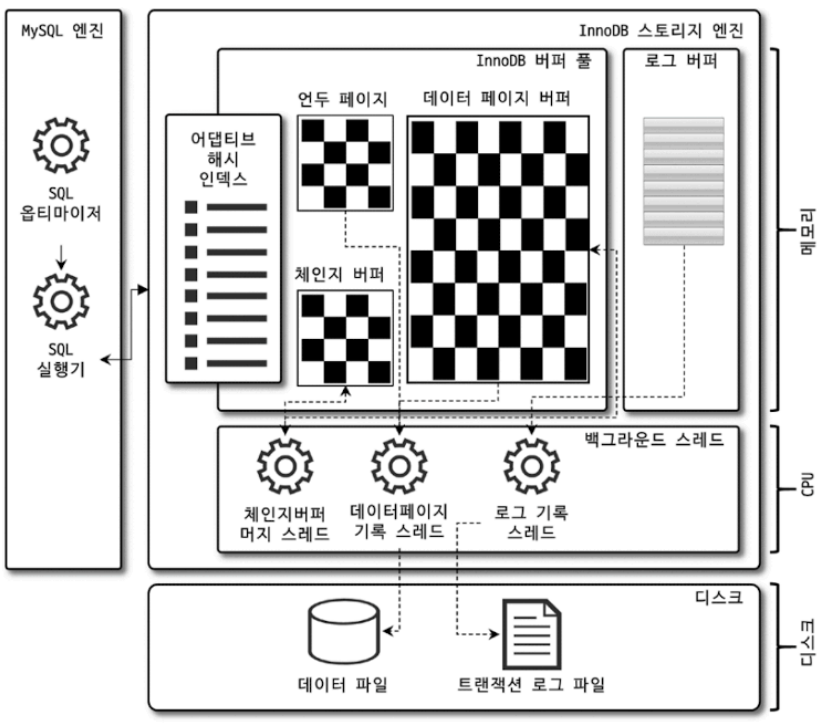
4.2 InnoDB 스토리지 엔진 아키텍처

4.2 InnoDB 스토리지 엔진 아키텍처



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

- **프라이머리 키에 의한 클러스터링**

  InnoDB의 모든 테이블은 기본적으로 PK 값 순서대로 디스크에 저장된다.

  모든 세컨더리 인덱스는 레코드의 주소 대신 PK 값을 논리적인 주소로 사용한다.

  PK가 클러스터링 인덱스이기 때문에 PK에 대한 range scan은 상당히 빠르며, 쿼리 실행계획에서 다른 보조 인덱스보다 PK가 선택될 확률이 높다.

  MyISAM* 스토리지 엔진에서는 클러스터링 키** 를 지원하지 않고, PK는 유니크 제약을 가질 뿐 세컨더리 인덱스와 구조적으로 아무런 차이가 없다.

  *MyISAM 데티블 구조와 인덱스는 4.3.3 데이터 파일과 프라이머리 키 (인덱스) 구조 참조

  ** 8.8 클러스터링 인덱스 참조

- **외래 키 지원**

  외래 키 지원은 InnoDB 스토리지 엔진 레벨에서 지원하는 기능으로 MyISAM이나 MEMORY 테이블에서는 사용할 수 없다.

  외래 키는 DB 서버 운영의 불편함 때문에 서비스용 DB에는 생성하지 않는 경우도 자주 있다.

  `foreign_key_checks` 시스템 변수를 OFF로 설정하면 FK 체크 작업을 일시적으로 멈출 수 있다.

  fk는 부모 테이블과 자식 테이블 모두 해당 칼럼에 인덱스 생성이 필요하다. 또한, 변경 시 반드시 부모와 자식 테이블에 데이터가 있는지 체크해야 한다. 이로 인해 lock이 여러 테이블로 전파되어 데드락이 발생하는 경우가 많으므로 개발할 때 주의할 필요갸 있다.

- **MVCC (Multi Version Concurrency Control)**

  하나의 레코드에 대해 2 개의 버전이 유지되고, 필요에 따라 보여지는 데이터가 달라지는 구조.

  lock을 사용하지 않는 일관된 읽기를 제공하는 것을 목적으로 한다.

  일반적으로 레코드 레벨의 트랜젝션을 지원하는 DBMS가 제공하는 기능이다. InnoDB는 Undo log를 이용해 이 기능을 구현한다.

  ex) 테이블 데이터가 변경되었지만 커밋되지 않은 시점에 다른 트랜젝션에서 해당 레코드를 조회할 경우 격리수준에 따라 다음과 같이 동작한다.

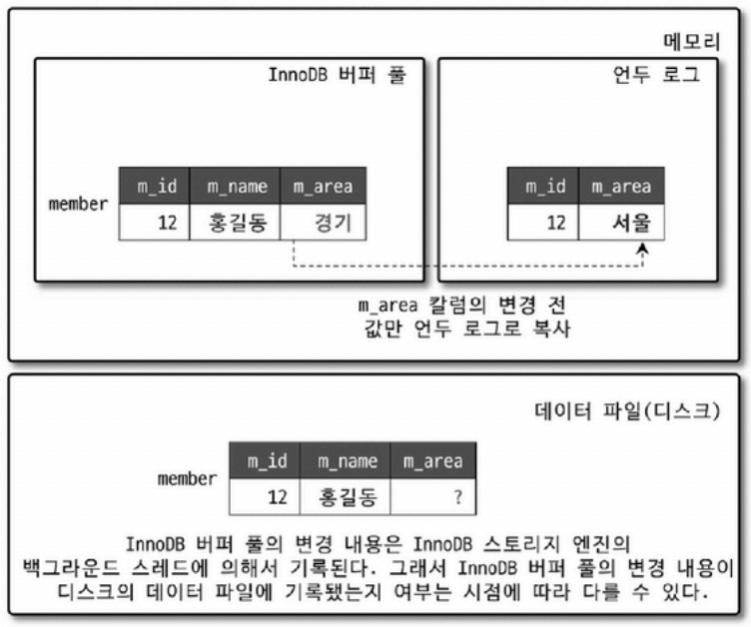


그림 4.11 UPDATE 후 InnoDB 버퍼 풀과 데이터 파일 및 언두 영역의 변화

1) **READ_UNCOMMITTED**

→ InnoDB 버퍼 풀이 가진 변경된 데이터를 읽어 반환

2) **READ_COMMITED 또는 그 이상 (REPEATEBLE_READ, SERIALIZABLE)**

→ 변경되기 전 내용을 보관하고 있는 Undo 영역의 데이터를 반환

- **잠금 없는 일관된 읽기 (Non-Locking Consistent Read)**

  MVCC를 통해 InnoDB 스토리지 엔진은 읽기 작업에 대해 lock을 걸지 않는다. 그렇기 때문에 격리 수준이 SERIALIZABLE이 아니라면, INSERT 작업이 없는 순수 읽기 작업은 다른 트랜젝션의 lock을 기다릴 필요가 없이 바로 실행된다.

  일관된 읽기를 위해서는 Undo 로그를 유지해야 하므로 트랜젝션이 오랫동안 활성상태를 유지하는 경우 MySQL 서버가 느려지거나 문제가 발생할 수도 있다. 따라서 트랜젝션 시작 후 가능한 빨리 롤백 또는 커밋을 통해 트랜젝션을 완료하는 것이 좋다.

- **자동 데드락 감지**

  InnoDB 스토리지 엔진은 내부적으로 데드락을 체크하기 위해 잠금 대기 목록 그래프 (Wait-for-List)를 관리한다.

  InnoDB 스토리지 엔진은 데드락 감지 스레드를 가지고 있어서 주기적으로 잠금 대기 그래프를 검사해 교착상태에 빠진 트랜젝션들을 찾아 그 중 하나를 강제종료 한다. 이때, 강제종료할 트랜젝션의 선택 기준은 Undo 로그의 양이며, 로그 양이 더 적은 트랜젝션이 강제종료 대상이 된다.

  동시 처리 스레드가 매우 많은 경우 데드락 감지 스레드는 매우 많은 CPU 자원을 소모할 수 있다.

  `innodb_deadlock_detect` 시스템 변수를 OFF로 설정하면 데드락 감지 스레드가 작동하지 않도록 할 수 있다.

- **자동화된 장애 복구**

  InnoDB 데이터 파일은 기본적으로 MySQL 서버가 시작될 때 항상 자동 복구를 수행한다. 이 과정에서 자동으로 복구될 수 없는 손상이 있다면 자동 복구를 멈추고 MySQL 서버는 종료되어 버린다. 이때는 MySQL 서버 설정파일의 `innodb_force_recovery` 시스템 변수를 설정해서 MySQL 서버를 시작해야 한다. 설정 가능한 값은 1~6까지이다.

  그래도 MySQL 서버가 시작되지 않으면 백업 파일로 DB를 새로 구축하고 바이너리 로그를 사용해 최대한 장애 시점까지의 데이터를 복구해야 한다.

- **InnoDB 버퍼 풀**

  InnoDB 스토리지 엔진에서 가장 핵심적인 부분으로, 디스크 I/O를 최소화하고 성능을 극대화하는 역할을 한다. 1) 디스크의 데이터 파일이나 인덱스 정보를 메모리에 캐싱하고, 2) 쓰기 작업을 지연시켜 일괄 작업으로 처리할 수 있게 해주는 버퍼 역할을 한다.

  서버 메모리가 허용하는 만큼 크게 설정하면 할수록 쿼리의 성능이 빨라진다. → 캐싱 성능 향상

  (디스크의 모든 데이터 파일이 버퍼 풀에 적재될 정도의 버퍼 풀 공간이라면 버퍼 풀 크기를 늘려도 더 성능이 좋아지지는 않을 것이다.)

  - **버퍼 풀의 크기 설정**

OS와 각 클라이언트 스레드가 사용할 메모리도 충분히 고려해서 설정해야 한다.
MySQL 서버가 사용하는 레코드 버퍼* 공간은 별도록 설정할 수 없으며, 전체 커넥션 개수와 각 커넥션에서 읽고 쓰는 테이블의 개수에 따라서 결정된다.

MySQL 5.7 버전부터 InnoDB의 버퍼 풀 크기를 동적으로 조절할 수 있게 되었다. 가능하면 버퍼 풀의 크기를 적절히 작은 값으로 설정해서 조금씩 증가시키며 최적점을 찾는 것이 좋다.

InnoDB 버퍼 풀은 내부적으로 128MB 청크 단위로 관리된다. 즉, 버퍼 풀의 크기를 늘리거나 줄일 때 128MB 단위로 처리된다.

`innodb_buffer_pool_size` 시스템 변수로 크기 설정 가능.

`innodb_buffer_pool_instance` 시스템 변수를 통해 버퍼 풀을 여러 개로 분리해 관리할 수 있으며, 각각의 버퍼 풀은 버퍼 풀 인스턴스라고 표현한다. 기본 8개로 초기화되고, 전체 버퍼 풀을 위한 메모리 크기가 1GB 미만이면 버퍼 풀 인스턴스는 1개만 생성된다.

ex) RAM 8GB 미만 : 전체 메모리의 50%정도만 버퍼풀로 설정

8GB 이상인 경우 50%부터 시작해서 조금씩 올려가면서 최적점을 찾는다.

50GB 이상인 경우 13 ~ 30GB정도만 남겨두고 나머지를 InnoDB 버퍼 풀로 할당.

ex) 버퍼 풀로 할당할 수 있는 메모리 공간이 40GB 이하라면 버퍼 풀 인스턴스 수는 기본 값은 8을 유지하고, 메모리가 크다면 인스턴스당 5GB정도가 되도록 인스턴스 개수를 설정하는 것이 좋다.

*레코드 버퍼: 각 클라이언트 세션에서 테이블의 레터드를 읽고 쓸 때 버퍼로 사용하는 공간

- **버퍼 풀의 구조**

  InnoDB 스토리지 엔진은 버퍼 풀 메모리 공간을 페이지 단위로 쪼개어 사용한다.

  InnoDB 스토리지 엔진은 버퍼 풀의 페이지 조각을 관리하기 위해 크게 다음 3가지의 자료구조를 관리한다.

  - **LRU (Least Recently Used) List**

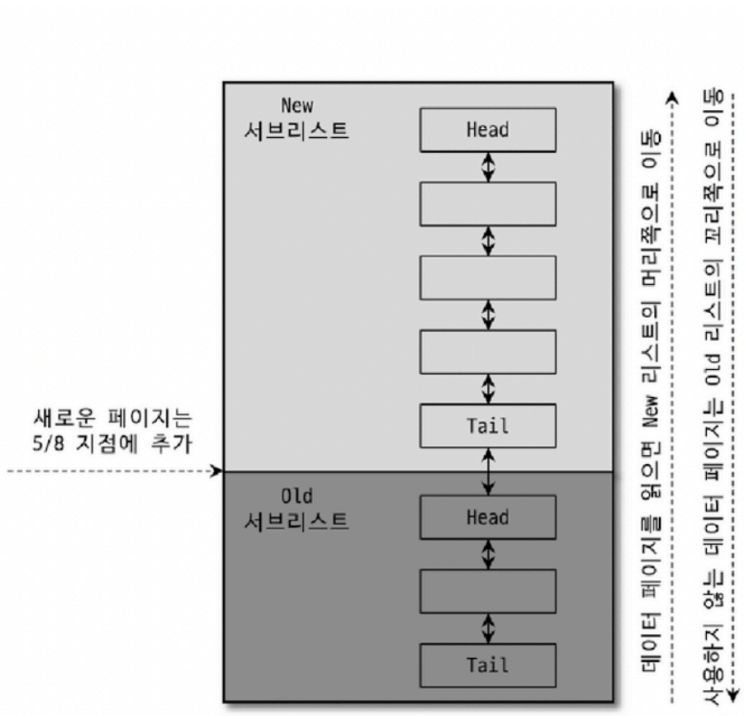


그림 4.13 버퍼 풀 관리를 위한 LRU 리스트 구조

디스크로부터 한 번 읽어온 페이지를 최대한 오래 InnoDB 버퍼 풀 메모리에 유지하여 디스크 읽기를 최소화하는 것을 목적으로 한다.

LRU와 MRU (Most Recently Used) 리스트가 결합된 형태이다.

Old 서브 리스트 영역 → LRU

New 서브 리스트 영역 → MRU

1) 버퍼 풀에 데이터 페이지가 있다면 해당 페이지 포인터를 MRU 방향으로 승급

2) 버퍼 풀에 없다면 디스크에서 필요한 페이지를 메모리에 적재 후 LRU 헤더 부분에 추가.

LRU 헤더의 데이터가 실제로 읽히면 MRU 헤더로 이동.

Aging을 통해 오래 사용되지 않은 페이지는 LRU 끝으로 밀려나며, 버퍼 풀에서 제거됨 (**Eviction**).

3) 필요한 데이터가 자주 접근됐다면 해당 페이지 인덱스 키를 어댑티브 해시 인덱스에 추가.

- **Flush List**

  : 디스크로 동기화되지 않은 데이터를 가진 페이지 (더티 페이지)가 변경된 시점 기준의 페이지 목록을 관리한다. 디스크에서 읽은 상태 그대로 변경이 없다면 플러시 리스트에서 관리되지 않는다.

- **Free List**

  : 실제 사용자 데이터로 채워지지 않은 비어있는 페이지들의 목록. 새롭게 디스크의 데이터 페이지를 읽어와야 하는 경우 사용된다.

- **버퍼 풀과 리두 로그**

  데이터가 변경되면 InnoDB는 변경 내용을 Redo 로그에 기록하고 버퍼 풀의 데이터 페이지에도 변경 내용을 반영한다. Redo 로그의 각 엔트리는 특정 데이터 페이지와 연결된다.

  Redo 로그가 디스크로 기록되었다고 데이터 페이지가 디스크로 기록되었다는 것을 항상 보장하지는 않으며, 반대의 경우도 발생할 수 있다. InnoDB 스토리지 엔진은 주기적으로 **체크포인트***를 발생시켜 디스크의 Redo 로그와 데이터 페이지의 상태를 동기화하게 된다.

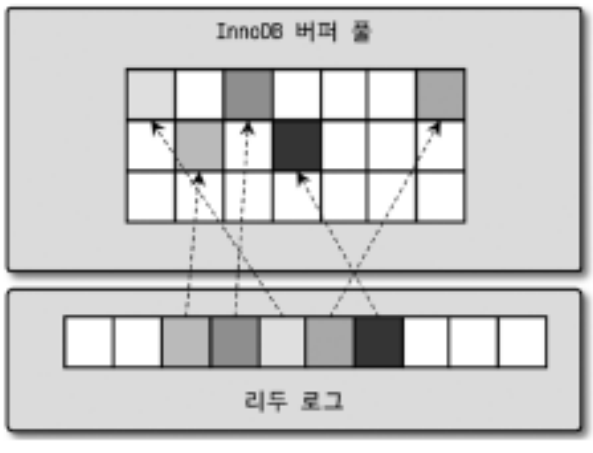


그림 4.14 InnoDB 버퍼 풀과 리두 로그의 관계

데이터 변경이 계속 발생하면 Redo 로그 엔트리가 새로운 로그 엔트리로 덮어씌워질 수 있기 때문에 전체 Redo 로그 파일에서 재사용 가능한 공간과 바로 재사용 불가능한 공간을 구분해서 관리한다. 재사용 불가능한 공간을 **활성 리두 로그 (Active Redo Log)**라고 한다.

체크포인트 에이지**는 활성 리두 로그 공간의 크기를 나타낸다.

- **캐싱 성능 → 버퍼 풀 크기 증가**

  → 버퍼 풀(InnoDB Buffer Pool)이 클수록 더 많은 데이터를 메모리에 저장하여 디스크 I/O를 줄이고 속도를 높임

- **버퍼링 성능 → 버퍼 풀과 Redo 로그 크기 비율 조정 + 더티 페이지 관리**

  → 더티 페이지를 많이 가지고 있을 수록 디스크 쓰기 작업을 버퍼링하여 여러번의 디스크 쓰기를 한 번으로 줄일 수 있다.

  → Redo 로그가 커야 변경된 데이터를 충분히 저장할 수 있다

  → 더티 페이지 비율이 많을수록 디스크 쓰기 폭발 (**Disk IO Burst**) 현상이 발생할 가능성이 높아진다.

  InnoDB 성능을 극대화 하기 위해서는 위 두가지를 고려하여 버퍼 풀 크기와 Redo 로그 크기를 설정해야 한다. 기본적으로 InnoDB 스토리지 엔진은 전체 버퍼 풀이 가진 페이지의 90%까지 더티 페이지를 가질 수 있으며, 시스템 설정 변수를 이용해 더티 페이지의 비율을 조정할 수 있다.

  * 체크포인트: MySQL 서버 시작 시 InnoDB 스토리지 엔진이 Redo 로그의 어느 부분부터 복구를 실행할지 판단하는 기준점이 된다.

  ** 체크포인트 에이지: 가장 최근 체크포인트의 LSN (Long Sequence Number)과 마지막 Redo 로그 엔트리의 LSN 차이.

- **버퍼 풀 플러시 (Buffer Pool Flush)**

  MySQL 5.6버전까지는 급작스럽게 디스크 기록이 폭증해서 사용자 쿼리 처리 성능에 영향을 미치는 등 더티 페이지 플러시 기능이 좋지 않았다. 하지만 버전이 업그레이드 되면서 예전과 같은 디스크 쓰기 폭증 현상은 발생하지 않으며 특별히 서비스를 운영할 때 성

능 문제가 발생하지 않는다면 관련 시스템 변수들을 조정할 필요는 없다.

- **Flush_list 플러시**

  : Redo 공간 재활용을 위해 오래된 Redo 로그 엔트리가 사용하는 공간을 비워야 한다. 이때, 오래된 Redo 공간을 비우기 전 버퍼 풀의 더티 페이지가 먼저 디스크로 동기화되어야 한다. InnoDB 스토리지 엔진은 주기적으로 플러시 리스트 블러기 함수를 호출해서 플로시 리스트에 오래전에 변경된 데이터 페이지 순서대로 디스크에 동기화하는 작업을 수행한다.

  어댑티브 플러시를 통해 더티 페이지 비율이나 설정 값에 의존하지 않고 Redo 로그 속도 증가에 따라 적절한 수준의 더티 페이지가 버퍼 풀에 유지될 수 있도록 디스크 쓰기를 실행한다. `innodb_adaptive_flushing` 시스템 변수로 On/Off 가능.

- **LRU_list 플러시**

  InnoDB 스토리지 엔진은 LRU 리스트에서 사용 빈도가 낮은 더티 페이지를 제거한다.

  1) LRU 리스트 끝부분부터 시작해서 최대 `innodb_lru_scan_depth` 시스템 변수에 설정된 개수만큼 페이지들을 스캔한다.

  2) 스캔하며 더티 페이지는 디스크에 동기화하고 클린 페이지는 즉시 free 리스트로 페이지를 옮긴다.

- **버퍼 풀 상태 백업 및 복구**

  버퍼 풀이 잘 **워밍업***된 상태에서는 그렇지 않은 경우보다 몇십 배의 쿼리 속도 차이를 보이는 것이 일반적이다.

  MySQL 5.5 버전에서는 MySQL 서버를 셧다운했다가 다시 시작하는 경우 강제 워밍업을 위해 주요 테이블과 인덱스에 대해 풀 스캔을 한 번씩 실행 후 서비스를 오픈했다.

  MySQL 5.6 버전부터는 버퍼 풀 덤프 및 적재 기능이 도입되어 `innodb_buffer_pool_dump_now` 시스템 변수를 이용해 현재 InnoDB 버퍼 풀의 상태를 백업할 수 있고, `innodb_buffer_pool_load_now` 시스템 변수를 이용해 백업된 버퍼 풀 상태를 복구할 수 있다.

  버퍼 풀 백업 작업은 LRU 리스트에 적재된 페이지의 메타데이터만 저장하기 때문에 매우 빠르다. 그러나 버퍼 풀을 다시 복구하는 작업은 상당히 많은 디스크 읽기를 필요로 하기 때문에 InnoDB 버퍼 풀 크기에 따라 상당한 시간이 걸릴 수도 있다.

  `innodb_buffer_pool_load_abort` 시스템 변수를 통해 복구 중단 가능.

  * Warming Up: 디스크의 데이터가 버퍼 풀에 적재되어 있는 상태.

- **버퍼 풀의 적재 내용 확인**

  MySQL 5.6 버전부터 `information_schema` DB의 `innodb_buffer_page` 테이블을 이용해 InnoDB 버퍼 풀의 메모리에 어떤 테이블의 페이지들이 적재돼 있는지 확인할 수 있었지만 버퍼 풀이 큰 경우 테이블 조회가 상당히 큰 부하를 일으켜 서비스 쿼리가 많이 느려지는 문제가 있었다. 그래서 실제 서비스용 MySQL 서버에서는 버퍼 풀의 상태를 확인하는 것이 거의 불가능했다.

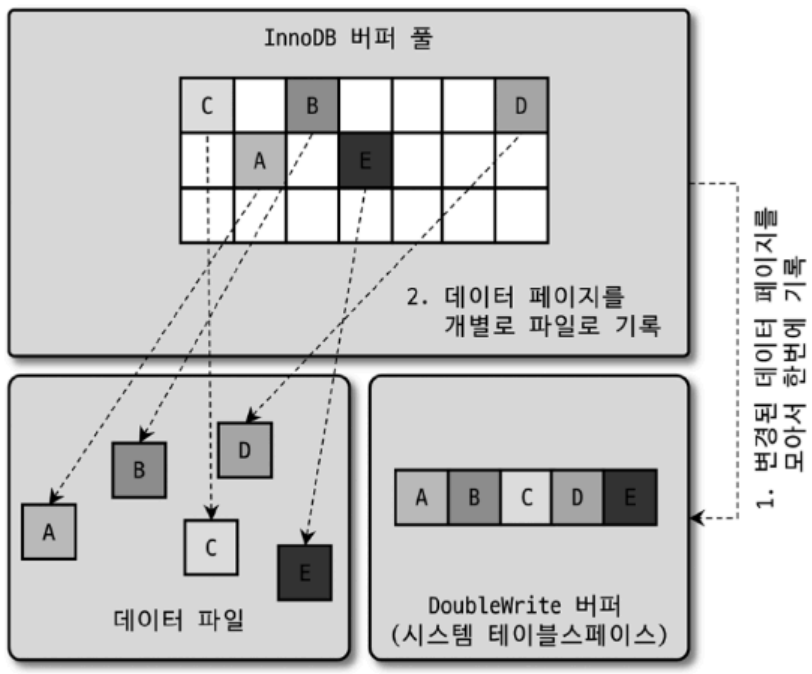
  MySQL 8.0 버전에서는 이런 문제를 해결하기 위해 `information_schema` DB에 `innodb_cachaed_indexes` 테이블이 새로 추가됐다.

- **Double Write Buffer**

  InnoDB 스토리지 엔진의 Redo 로그는 공간의 낭비를 막기 위해 페이지의 변경된 내용만 기록한다. 이로 인해 InnoDB의 스토리지 엔진에서 더티 페이지를 디스크 파일로 flush할 때 일부분만 기록되는 경우 그 페이지의 내용은 복구하지 못 할 수도 있다.

  이렇게 페이지가 일부분만 기록되는 현상을 **Partial-page** 또는 **Torn-page**라고 한다. HW 오작동이나 시스템의 비정상 종료 등으로 발생할 수 있다.

  InnoDB 스토리지 엔진에서는 이 같은 문제를 막기 위해 Double-Write 기법을 이용한다.



1) InnoDB에서 'A' ~ 'E'까지의 더티 페이지를 디스크로 플러시한다.

2) InoDB 스토리지 엔진은 실제 데이터 파일에 변경 내용을 기록하기 전 'A' ~ 'E'까지의 더티 페이지를 우선 묶어서 한 번의 디스크 쓰기로 시스템 테이블 스페이스의 Double Write 버퍼에 기록한다.

3) InnoDB 스토리지 엔진은 각 더티 페이지를 파일의 적당한 위치에 하나씩 랜덤으로 쓰기를 실행한다.

기록 도중 비정상적으로 작업이 종료된다면 InnoDB 스토리지 엔진은 재시작 시 Double Write 버퍼의 내용과 데이터 파일의 페이지들을 모두 비교해서 다른 내용을 담고 있는 페이지가 있으면 Double Write 버퍼의 내용을 데이터 파일로 복사한다.

실제 파일에 더티 페이지가 정상적으로 기록되면 Double Write 버퍼 공간에 기록된 변경 내용은 더이상 필요 없어진다.

Double Wirte 기능 사용 여부는 `innodb_doublewrite` 시스템 변수로 제어 가능.