ARIES-Style physiological logging

in SQLite

성균관대학교 컴퓨터공학과

이유비, 장산, 정성윤

목 차

1. SQLite
2. 현재 SQLite의 recovery modes에 대한 문제점
3. Physiological Logging 소개
4. 구현
5. 성능 측정
6. 결론
7. SQLite

1) 소개

SQLite 뿐 아니라 일반적인 데이터베이스 시스템은 비휘발성 저장 장치인 디스크에 데이터를 저장하며 전체 데이터베이스의 일부분을 메인 메모리로 불러와 유지한다. 디스크에 저장되는 데이터의 단위는 페이지(page)라는 형태로 저장되며, 데이터베이스에 작업을 처리할 때 페이지 단위로 입출력을 수행한다. 일반적인 데이터베이스 시스템들과 마찬가지로 sqlite3에는 페이지 버퍼 관리자가 존재한다. 이러한 버퍼 관리자의 역할을 수행하는 모듈은 페이저(Pager) 모듈이다. sqlite3 내부의 페이저 모듈은 트리(tree) 모듈과 OS 인터페이스 사이에서 페이지에 대한 읽기, 쓰기, 캐싱 뿐만이 아니라 트랜잭션 관리, 로그, 락 매니저 로서의 역할도 수행한다.

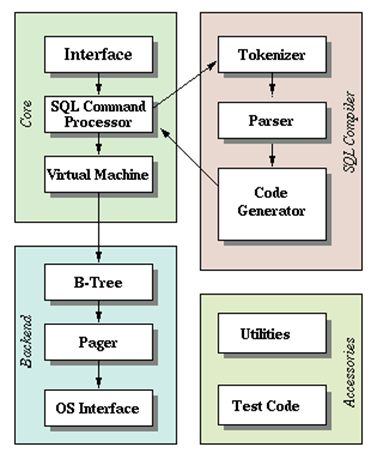


Figure 1: SQLite 구조[[1]](#footnote-1)

2) ACID & Recovery

ACID란, Atomicty, Consistency, Isolation, Durability 의 약자로, 데이터베이스 시스템이 가져야할 필수 특성들을 말한다. 이 때, Recovery manager는 Atomicity & Durability를 보장해야 한다.

Recovery는 commit된 상태와 디스크에 실제 저장되어 있는 데이터의 상태가 일치하지 않을 때, commit된 상태로 일치시킬 수 있도록 복구하는 것을 말한다. Recovery 과정에는 UNDO, REDO operation이 있는데 이는 다음과 같다.

UNDO: 이미 디스크에 반영되어 있는 것을 취소하여 그전 상태로 돌아갈 수 있도록 하는 과정

REDO: 디스크에 아직 반영되지 않았으나 꼭 반영되야할 것을 다시 수행하는 과정

Recovery mode에 따라, UNDO, REDO에 관한 구현이 달라지게 되는데, recovery mode는 Force 정책, Steal 정책에 따라 분류된다. Force, Steal을 간단히 설명하면 다음과 같다.

데이터베이스 페이지 버퍼 관리 정책에 따라 트랜잭션의 ACID 성질을 만족시키기 위해 UNDO 또는 REDO 기능이 요구된다.

아래의 경우는 버퍼 관리자의 관리 정책에 따라 분류한 것이다.

1. 수정된 페이지를 교체할 할 수 있는가?

a. Steal: 수정된 페이지를 디스크에 쓸 수 있다.

b. No-steal: 수정된 페이지는 최소한 트랜잭션 종료까지 버퍼에 유지한다.

2. 트랜잭션 커밋 시점에 페이지를 디스크에 반영하는가?

a. Force: 수정된 페이지 커밋 시점에 디스크 반영

b. No-force: 수정된 페이지 커밋 시점에 반영하지 않음

위의 경우 중 참고할 사항은 1.a 와 2.b 이다.

1의 경우, 페이지 캐시가 다 찼을 시 생각해 볼 수 있는 시나리오이다. 즉 캐시 영역이 모두 찼기 때문에 새로운 공간을 할당할 필요가 있고 (데이터베이스의 모든 작업은 페이지를 기반으로 하며 이를 위해 반드시 메모리에 캐시된 페이지 상에서만 동작하기 때문에 해당 페이지에 새로운 영역을 할당해 주어야 한다.) 만일 더 이상 새로운 영역을 할당할 수 없을 만큼 캐시가 꽉 찼다면 기존에 있는 페이지들 중에서 알고리즘에 기반하여 중요도가 낮은 것을 새로운 페이지를 위한 공간으로 대체한다. 이때 대체되는 페이지의 내용이 변경된 사항이 없으면 그냥 메모리 영역에서 해제해 주면 되지만 만일 update나 delete 등의 명령을 통해 사용자가 테이블의 내용을 변경하였다면 페이지의 내용이 변경되어 디스크에 있는 원본 페이지와 차이가 있으므로 이 변경된 페이지의 내용을 디스크에 써줘야 한다. a의 경우, 수정은 되었지만 디스크에 쓰여지지 않은 페이지에 대해 페이지 캐시에서 이를 디스크에 써서 공간을 확보하는 방식을 의미하며 b의 경우 수정되었지만 아직 디스크에 반영되지 않은 페이지들은 해당 페이지를 다루는 트랜잭션이 종료될 때 까지 페이지 캐시에 있는 것을 보장하는 방식이다. 여기서 a의 경우 문제는 어떤 트랜잭션에서 페이지를 수정 한 후 만일 그것이 steal 정책에 의해 디스크에 쓰여졌는데 해당 트랜잭션에서 abort 명령이나 혹은 시스템 크래쉬로 인해 트랜잭션 수행 전의 상태로 되돌려야 할 경우 디스크에 쓰여졌던 내용을 원상복귀 시키는 동작이 반드시 필요하게 되며 이를 UNDO 라고 한다.

2의 경우, 트랜잭션이 종료 후 해당 내용을 디스크에 쓰는 시점에 따라 force 와 no-force 정책으로 나눌 수 있다. force는 수정된 페이지가 존재한다면 트랜잭션이 종료된 그 시점에 바로 페이지를 디스크에 써주며 no-force의 경우 수정된 페이지를 트랜잭션의 종료 시점에 바로 반영하는 게 아니라 해당 정보들을 모아두었다가 나중에 특정 시점(체크 포인트)에 이를 디스크에 반영하는 정책이다. b의 경우에 수정된 페이지들의 정보들을 가지고 있을 때 시스템 크래쉬 등의 장애가 발생하여 복귀 된다면 디스크에는 해당 트랜잭션이 반영되지 않은 상태이기에 이를 다시 한번 반영 시켜줘야 하는 동작이 필요하며 이를 REDO라고 한다. REDO 동작을 수행하기 위해 변경된 페이지들의 정보를 따로 관리할 로그(log) 라는 파일이 필요하며 해당 파일에 쓰여져 있는 페이지들의 변경 정보는 특정 시점에 한꺼번에 내려 써주어 트랜잭션을 보장해야 한다.

위와 같이 데이터베이스 시스템의 버퍼관리 정책에 따라 트랜잭션 성질을 만족시키기 위해 추가적인 기능 (UNDO, REDO)이 필요로 하다. 일반적인 데이터베이스 시스템들의 페이지 버퍼 관리자들은 steal + no-force 조합을 사용한다. 그러나 sqlite3의 경우 steal + force 정책을 사용하여 트랜잭션의 커밋 시점에서 바로 디스크에 해당 페이지들의 변경점을 바로 쓰는 정책을 사용하고 있다.[[2]](#footnote-2)

1. 현재 recovery modes에 대한 문제점

SQLite는 Force, Steal 정책을 취하고 있는데, Force 정책에 의한 write amplification이 일어날 수 있다는 점이다. write amplification이란, 디스크에 쓰기 연산이 실제 변경되어야 하는 부분에 비하여 오버헤드가 큰 경우를 말한다. 쉽게 말해, 100B미만의 아주 작은 변화도 한 페이지의 변화를 만들어냈다면, 하나의 4KB 페이지 전체를 write해야한다는 것이다.

Force 정책으로 인하여 recovery에 대한 구현이 간단해지거나 불필요해지는 장점이 있으나, 매 페이지에 대한 변화를 write해야한다는 점에서 성능저하가 발생할 수 있다. (참고로 현재 주요한 DBMS들은 no-force, steal 정책을 적용한다는 점에서 차이가 있다.)

1. Physiological Logging 소개

위에서 말한 REDO와 UNDO 기능을 구현하기 위해 데이터베이스 시스템에서는 로그라는 파일을 두어 변경사항을 저장하고 데이터베이스의 저장 및 트랜잭션의 복구에 사용한다. 이 때 로그를 작성하는 방식에 따라 크게 Physical Logging과 Logical Logging 그리고 이 둘의 하이브리드 형태인 Physiological Logging으로 구분 할 수 있으며 이번 과제에서 우리의 목적은 Physiological Logging을 구현하여 변경사항들을 저장하고 이를 통해 force 정책을 no-force로 변경하는 것이다. 우선 각각의 로그 방식의 차이점을 비교해 보자.

(1) Physical logging

피지컬 로깅은 몇번 페이지의 물리적 오프셋 및 몇개의 글자가 바뀌는 지를 표시하는 방식이다. 예를 들어 Page42; Image at 367, 2; before:’ke’; after:’ca’ 라는 문장을 로그 파일에 저장하였다 하면 이는 페이지 42번에서 오프셋 367에서 부터 시작하여 2 만큼의 길이의 정보가 처음에는 ke 였다가 ca로 바뀌었음을 저장 한다는 의미이다. 로그 레코드 하나당 단 하나의 페이지의 변경점 만을 의미한다.

(2) Logical logging

로지컬 로깅은 고수준의 명령어 기반의 로깅 방식으로 어떤 테이블의 어떤 값이 무엇에서 무엇으로 변경하라는 명령어 자체를 로깅한다. 예를들어 CameraLingo; update(0, ‘kemera’=>’camera’) 라고 한다면 이는 CameraLingo 라는 테이블의 0번째 레코드의 값을 kemera에서 camera로 바꾼다는 정보의 로그 레코드이다. 피지컬 로그와의 큰 차이점은 해당 로그 레코드가 하나의 페이지에만 관여하지 않을 수도 있다는 것이다. 즉 테이블의 해당 값이 다수의 페이지에 걸쳐 존재할 가능성이 있으므로 하나의 로그 레코드 값에 의해 다수의 페이지가 변경되어질 수 있다.

(3) Physiological logging

우리가 이용하고자 하는 방식은 피지올로지컬 로그 방식으로 피지컬 로깅과 로지컬 로깅 두가지를 합쳐놓은 방식이다. 예를 들어 Page42; update(0, ‘Kemera’ => ‘camera’) 와 같은 형식의 로그레코드를 남기는데 테이블 단위가 아니라 페이지 단위로 변경 사항을 기록하지만 해당 페이지의 내용이 변경되기 위해 어떤 논리적 동작이 사용되었는지를 기록하는 방식이다.

위와 같이 분류한 로그의 종류 중 무엇을 선택할 것인가는 데이터베이스 시스템이 사용되어지는 환경과 밀접한 연관이 있다. 로지컬 로깅은 레코드에 대한 추상화 단계가 높기에 모든 페이지 단위의 변경점을 기록하지 않아 로그 레코드의 개수가 짧아져 로그 파일을 읽는 데 걸리는 IO 시간이 줄어드는 하나의 로그 레코드를 처리하는 소비되는 프로세싱 타임이 증가한다. 반면 피지컬 로깅은 테이블의 한 레코드를 변경하는 것에도 다수의 페이지의 정보가 변경될 가능성이 있어서 로그 레코드의 개수가 많고 따라서 로그파일이 커지며 I/O에 대한 시간이 길어지는 대신 한 로그 레코드를 처리하는 프로세싱 타임이 줄어드는 효과를 가진다. 피지올로지컬 로깅은 두 로깅 기법 사이의 타협점으로 프로세싱 타임과 IO 시간 값들이 서로 다른 두 로그 기법의 중간쯤에 위치하여있다. 우리의 구현 방식은 이러한 physiological logging 기법을 따른다.[[3]](#footnote-3)

1. 구현

insert, update, delete 트랜잭션들과 관련된 함수가 호출될 시에 정해놓은 로그 양식에 따라 로그를 남긴 후, SQLite를 재실행 시, 로그파일을 읽으면서 recovey가 진행될 수 있도록 하였다.

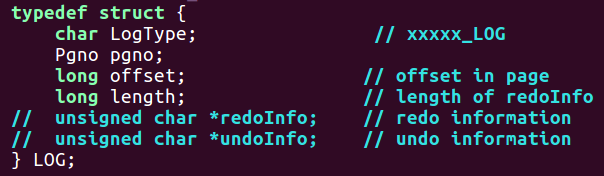


Figure 2: 로그 구조

로그는 다음과 같은 구성요소로 이루어져있다.

* LogType: 해당 로그가 어떠한 종류인지 나타내는 변수
* pgno: 해당 트랜잭션이 관련된 페이지번호를 저장하는 변수
* offset: page안에서의 offset을 나타냄
* length: 현재 4개의 구성요소 외의 실제 데이터에 대한 길이를 담고 있는 변수

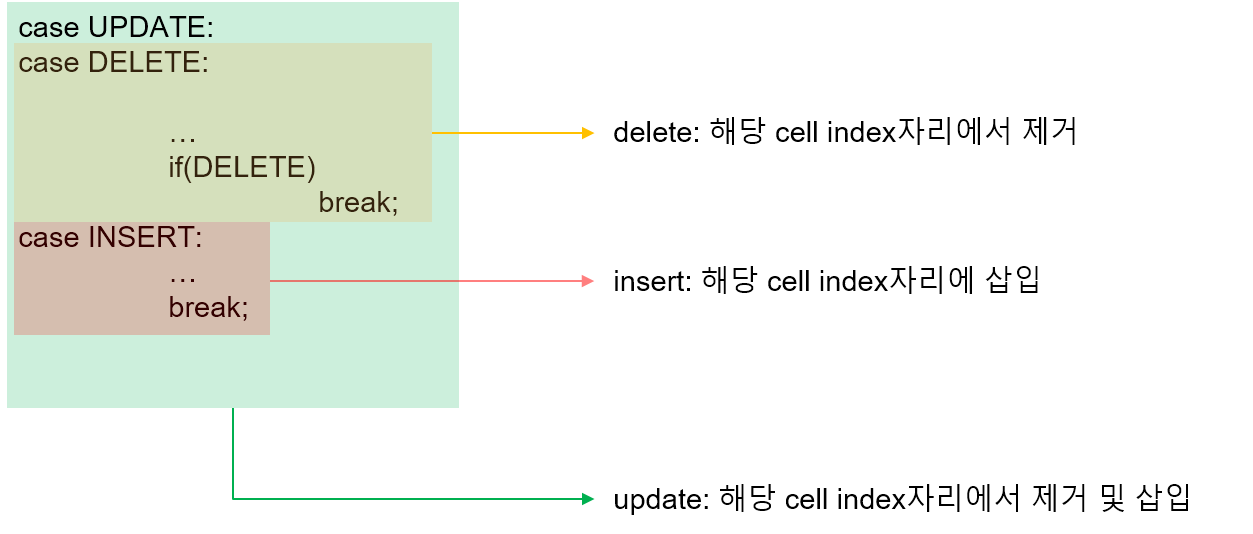


Figure 3: Recovery 방식

recovery는 SQLite가 첫 실행됐을 시, 로그 파일을 읽으며 cell에 대해서 LogType에 맞게 변화를 반영해주는 방식으로 진행한다. 자세한 구현은 소스코드 및 ppt 자료에 남긴다.

1. 성능 측정

성능측정은 다음과 같은 환경에서 이루어졌다.

* 구동환경

|  |  |  |
| --- | --- | --- |
|  | Original WAL | Physiological logging |
| OS | Linux ubuntu 4.4.0 | Linux ubuntu 4.4.0 |
| Storage type | external HDD | external HDD |
| CPU | i5-5200 2.26GHz | i5-5200 2.26GHz |
| Memory | 1GB | 1GB |

* 실험환경

하나의 column만 갖고있는 schema에 대하여 삽입되어 있는 99개의 tuple을 update하여 Original WAL에서는 항상 write가 일어날 수 있도록 하였고, Physiological logging에서는 로그에 대해서만 write 연산이 일어날 수 있도록 하였다.

* 테스트 결과

Original WAL:



Physiological logging:



1. 결론

아쉽게도 우리가 원했던 결과와는 달리, physiological logging 방식이 현저하게 느리다는 것을 보여주었다. 이는 우리가 로그를 남길 때, 계속해서 매핑했던 mmap 을 이용하여 로그파일에 써주기 위해 msync()를 반복적으로 호출했던 것이 오히려 역효과를 불러일으킨 것으로 보인다. SQLite의 일부분 코드만 수정하여 성능향상을 꾀해보려 하였으나, 실제 구현도 성능고려가 미흡하였고, 전체적인 SQLite의 구조를 알맞게 수정하지 못했다는 점이 이번에 나온 안좋은 성능결과를 보여주는 것으로 보인다.

<그림 목차>

[Figure 1: SQLite 구조 3](#_Toc470180332)

1. SQLite 구조: sqlite.org [↑](#footnote-ref-1)
2. DBMS 는 어떻게 트랜잭션을 관리할까? 오이석, Naver D2, http://d2.naver.com/helloworld/407507 [↑](#footnote-ref-2)
3. 14.604 What to log, Physical, Logical, and Physiological Logging, Trade-Offs…, Jens Dittrich, <https://www.youtube.com/watch?v=MR-89Lq5jCo&list=LLGGmNpPiAHQjTdQAuwmHFEg&index=16&t=288s> [↑](#footnote-ref-3)