WAL-FTL: SQLite 의 WAL 기능을 효율적으로 지원하는 FTL 설계 방안

이두기, 노홍찬, 박상현
연세대학교 컴퓨터과학과
e-mail : {edoogie, fallsmal, sanghyun}@cs.yonsei.ac.kr

WAL-FTL: Designing an FTL that supports SQLite's WAL functionalities efficiently

Doogie Lee, Hongchan Roh, Sanghyun Park*
Dept. of Computer Science, Yonsei University

요 약

스마트 기기에서는 NAND 기반 저장장치 위에 SQLite 를 활용하여 데이터를 관리하는 방식이 널리 쓰이고 있다. SQLite 에서 트랜잭션의 원소성을 보장하기 위한 기법인 WAL 은 트랜잭션 처리의 동시성을 높일 수 있어 다중 쓰레드 환경에 적합하지만, 오버헤드가 큰 체크포인트 동작을 주기적으로 수행하는 문제가 있다. 본 논문에서는 WAL 의 아이디어를 저장장치에 도입하여 트랜잭션을 처리할 때 동시성은 높이면서 오버헤드는 줄일 수 있는 저장장치 FTL 을 제안한다.

1. 서론

오늘날 스마트폰과 태블릿을 비롯한 많은 스마트 기기들은 주요 저장장치로 eMMC 등의 NAND 기반 저장장치(이하 저장장치 표현)를 사용하고 있다. NAND 기반 저장장치들은 HDD 등 다른 저장장치들에 비해 다소 비싸지만, 칩 하나 정도 크기로 소형화/경량화가 가능하고, 전력 소모도 상대적으로 작으며, 특히 외부로부터의 충격에 강하다는 장점이 있다.

한편, 스마트 기기 응용 분야가 다양해지면서 데이터를 안정적이면서 효율적으로 관리할 필요성이 증가하였다. 하지만, 대부분의 스마트 기기는 사용할 수 있는 자원의 제약이 심해 일반적으로 쓰이는 DBMS 보다는 경량 DBMS 인 SQLite 가 많이 쓰이고 있다.

SQLite 도 트랜잭션 기능을 제공하고 있으며, 그 중에서 트랜잭션의 원소성(atomicity)을 보장하기 위해 롤백 저널(Roll-Back Journal[1], 이하 RBJ)과 선행 기입 로깅(Write-Ahead Logging[2], 이하 WAL)이라는 두 가지 방법을 제공하고 있다. 그 중에서 특히 WAL 은 기존에 주로 쓰였던 RBJ 에 비해 일반적으로 좀 더 나은 성능을 보이고, 특히 트랜잭션 간의 동시성(concurrency, 이하 동시성)을 높일 수 있어 특히 다중 쓰레드(multi-thread) 환경에 유리하며, 최근에 많이 활용되는 추세이다.

그러나, WAL 은 구조적으로 주기적인 체크포인트(checkpoint) 동작이 필요한데, 이 체크포인트 동작의 경우 로그 파일의 내용을 읽어서 DB 파일에 일일이 복사하는 과정이어서 수행하는데 비교적 긴 시간이 필요하며, 수행되는 동안 다른 쓰레드가 DB 에 접근할 수 없어 동시성도 떨어지게 된다.

그런데, 최근 몇몇 연구[3-6]에서는 저장 장치 수준에서 호스트가 트랜잭션의 원소성을 보장하기 위해 수행하는 백업/로깅 등을 없애고, 대신에 저장장치에서 이러한 기능을 간접적으로 제공하는 방법들이 제시되었다. 이러한 방법들은 NAND 기반 저장장치의 독특한 저장 방식을 활용한 것들로서, 이러한 아이디어들과 WAL 을 접목시킨다면, 체크포인트 오버헤드가 매우 작으면서도 트랜잭션 처리의 동시성도 높은 새로운 방식을 개발할 수 있을 것이다.

본 논문에서는 우선 SQLite 의 WAL 의 동작 방식과, NAND 기반 저장 장치들의 독특한 특징 및 이를 활용한 트랜잭션 처리 방법론들에 대해 간단히 살펴본다. 그리고 나서 WAL 과 저장장치 수준 트랜잭션 처리 방법을 접목시킨 새로운 FTL 을 제안하고자 한다.

2. 연구 배경

2.1. SQLite 에서의 Write-Ahead Logging (WAL)

SQLite 의 WAL 저장 방식의 동작 방식은 다음과 같다.

우선, 트랜잭션을 이용하여 응용 프로그램이 데이터를 업데이트할 경우, 원래의 DB 파일에 바로 반영하지 않고, 변경된 내용들을 별도의 로그 파일인 WAL 파일 뒷부분에 프레임[1] 단위로 추가(append)한다.

[1] WAL 파일의 최소 기록 단위. 업데이트할 페이지의 내용 앞에 프레임 헤더(24 바이트)가 붙는다.

이후, 트랜잭션이 커밋 되면 WAL 파일에 커밋 레코드(프레임)를 기록한다.

WAL 파일에 쌓이게 되는 변경된 내용은 체크포인트(checkpoint) 연산을 통해 DB 파일에 반영된다. 체크포인트 동작은 WAL 파일에 있는 프레임들 중에서 유효(valid)한 프레임들만 DB 파일에 복사하는 동작으로 일정한 주기마다 호출되며, 체크포인트 동작을 마치면 WAL 파일이 초기화된다.

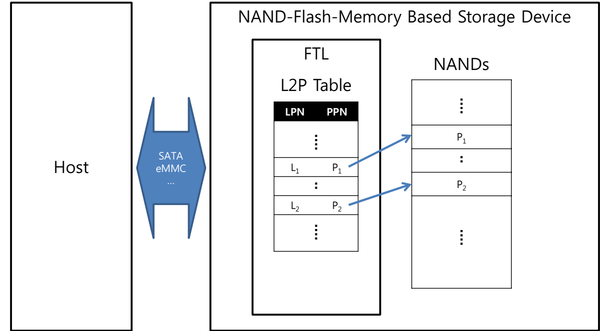
한편, 트랜잭션을 통한 DB 데이터 읽기 동작은 다음과 같이 수행된다. 우선, 모든 트랜잭션은 시작하는 시점의 WAL 파일의 유효 프레임 번호(valid frame number)를 기록한다. 이 유효 프레임 번호는 각각의 트랜잭션이 시작하는 시점에 WAL 파일 내에 있는 마지막 커밋 레코드의 위치를 가리키는 것인데, 각 트랜잭션에서 DB 에 있는 데이터를 읽을 때 SQLite 는 우선 WAL 파일의 시작부터 유효 프레임 번호까지의 프레임들 중에서 요청 받은 페이지가 있을 경우 WAL 파일에서 데이터를 읽어서 보내주고, 범위 안에 해당하는 프레임이 없을 때에는 DB 파일에서 페이지를 읽어서 보내준다.

SQLite 의 WAL 은 기존 방식인 RBJ 에 비해 특히 트랜잭션 동시성(concurrency) 측면에서 큰 장점이 있다. 즉, 다중-쓰레드 환경에서 하나의 DB 에 여러 트랜잭션이 동시에 접근하려고 할 때, 기존 RBJ 의 경우, 업데이트되는 내용은 DB 파일에 직접 반영되므로 한 트랜잭션이 읽기를 다 끝나기 전에는 다른 트랜잭션이 데이터를 업데이트할 수 없다. 반면, WAL 에서는 각 트랜잭션의 유효 프레임 번호를 기준으로 그 뒤에 쓰여진 프레임들은 무시하므로, 한 트랜잭션이 끝나지 않은 상태에서도 다른 트랜잭션들이 DB 를 업데이트할 수 있다. 마찬가지로, 한 트랜잭션이 데이터를 쓰고 있는 중이라도 다른 트랜잭션들은 각자의 유효 프레임 번호 앞쪽에 있는 데이터만 참조하므로 DB 를 업데이트 중인 트랜잭션에 영향을 받지 않고 읽기를 진행할 수 있다.

하지만, WAL 방식에도 몇 가지 단점은 존재한다. 우선, WAL 은 항상 WAL 파일에 유효한 페이지가 있는지 검사하고 나서 읽기 동작을 수행하므로, 데이터를 항상 DB 파일에서만 찾는 기존 RBJ 에 비해 읽기 동작이 다소 느려질 수 있다. 또한, WAL 은 유효한 데이터는 무조건 2 번(업데이트 시 WAL 파일 1 번, 체크포인트 시 DB 파일 1 번) 쓰도록 하고 있어 최악의 경우에는 실제 저장장치에 쓰이는 데이터의 양이 업데이트되는 데이터 크기의 2 배가 될 수 있다. 게다가, 주기적으로 필요한 체크포인트 동작으로 인해 상당한 오버헤드가 발생한다.

2.2. NAND 기반 저장장치의 데이터 업데이트 방식.

NAND 는 한 번 데이터를 쓰고(program) 나면 그 영역을 지우기(erase) 전까지는 덮어쓰기(over-write)가 불가능하다. 따라서 NAND 를 기반으로 하는 저장장치인 SSD 는 데이터를 업데이트 할 때 기존 데이터를 직접 수정하는 대신, 기존 데이터를 남겨두고 다른 빈 공간에 데이터를 옮겨 적고 주소에 대한 사상(mapping)만 변경하는 out-of-place 업데이트 방식을 채택하고 있다. 따라서 호스트가 특정 주소에 있는 데이터에 대한 읽기/쓰기 동작을 요청하면, 호스트의 주소(논리 주소)를 NAND 의 물리 주소로 변환(translation)하는 과정을 거친다. NAND 기반 저장 장치들은 이러한 변환 과정을 처리하기 위해 플래시 변환 계층(Flash Translation Layer, 이하 FTL)을 두어 논리-물리 주소 사상(Logical-to-Physical Address Mapping, 이하 L2P)을 담당하게 하고 있다. (그림 1)은 이러한 FTL 의 구조를 개념적으로 나타낸 것이다.



(그림 1) NAND 기반 저장장치 구조

이러한 out-of-place 업데이트 방식을 활용하면 저장장치 수준에서 효율적으로 트랜잭션의 원소성을 보장할 수 있다. 즉, 예를 들어 트랜잭션을 통해 업데이트된 데이터에 대한 L2P 주소 사상을 L2P 테이블에 바로 반영하지 않고 특정한 자료 구조에 저장하고 있다가, 트랜잭션이 커밋될 때 한꺼번에 L2P 테이블에 반영하는 등의 방식[4]으로 트랜잭션의 원소성을 비교적 간단하게 구현할 수 있다. 실제로 최근 이와 유사한 방식으로 저장장치 수준에서 트랜잭션을 처리하려는 다양한 방법들이 제시[3-6]되고 있다. 이러한 방식들의 경우, 기존의 저장장치에 의존하지 않는 다른 방식들이 호스트가 데이터를 서로 다른 위치에 2 번(백업, 로깅 등) 이상 적어야 하는 데 비해, 이미 저장장치에 적혀있는 데이터를 활용하여 데이터를 2 번 적지 않고 그보다 훨씬 크기가 작은 L2P 매핑 정보만 변경해서 트랜잭션 원소성을 보장할 수 있다. 이러한 방식들을 활용하면 저장장치에 대한 쓰기 횟수가 비약적으로 감소하여 트랜잭션 처리 및 복구 성능을 비약적으로 향상시킬 수 있다.

그러나, 기존의 시도들의 경우 쓰기 트랜잭션의 처리 및 복구 측면에서는 매우 효율적이나, 대부분 트랜잭션의 동시성은 고려하지 않고 있으며, 오히려 트랜잭션 전체 데이터를 한꺼번에 써야 한다 던지, 트랜잭션 읽기/쓰기 실행 순서에 제약을 두는 등으로 해서 트랜잭션의 동시성을 제약하고 있다.

3. WAL-FTL

3.1. 설계 목표 및 방향

본 논문에서는 FTL 의 설계를 변경하여 아래와 같은 2 가지 목표를 만족시키고자 한다.

a. 기존 WAL 방식보다 쓰기 부담을 줄인다.
b. 기존 WAL 방식의 장점인 트랜잭션의 동시성을 보장한다.

위의 2 가지 목표를 만족시키기 위해 다음과 같은 방향으로 FTL 을 설계한다.

1) 저장장치 기반 트랜잭션 처리: 데이터를 2 번 쓸 필요 없이 저장장치 내의 이전 페이지 정보를 활용하여 원소성을 확보한다.
2) WAL 읽기 처리: 저장장치에서 데이터를 읽을 때 특정 정보를 전달해서 진행중인 다른 트랜잭션으로부터 현재 트랜잭션을 격리시키면서 데이터의 일관성도 확보한다.

3.2. 기본 동작 설계

3.2.1. 트랜잭션 쓰기 처리

WAL-FTL 의 트랜잭션 쓰기 처리 방식은 다음과 같다.

1-0) 각 트랜잭션을 시작할 때, 트랜잭션 별로 고유한 트랜잭션 ID 를 부여한다.
1-1) 호스트는 트랜잭션 데이터를 쓸 때 WAL-FTL 에 트랜잭션 ID 를 같이 전달한다.
1-2) WAL-FTL 은 트랜잭션 데이터를 쓴 다음, 새로 업데이트 된 L2P 정보를 L2P 테이블에 바로 반영(기존 out-of-place 업데이트 방식)하지 않고, 별도의 트랜잭션 L2P(이하 WAL-L2P) 테이블에 따로 모아둔다.
1-3) 호스트가 트랜잭션에서 쓸 데이터를 다 쓴 다음에는 COMMIT 명령을 통해 트랜잭션이 완료되었음을 알린다.
1-3-1) 만일 트랜잭션 처리 중간에 호스트가 오류 상황을 발견하면 WAL-FTL 에게 ABORT 명령을 보낸다.
1-4) WAL-FTL 이 COMMIT 명령을 받으면 WAL-L2P 테이블에서 해당하는 L2P 정보를 찾아서 COMMIT 되었다고 표시한다. 이 때, 해당 WAL-L2P 엔트리에는 커밋 순서 번호(Commit Sequence Number, 이하 CSN)가 부여된다.
1-5) WAL-FTL 이 ABORT 명령을 받으면 WAL-L2P 테이블에서 해당하는 L2P 정보를 찾아서 제거한다.
1-6) 호스트는 WAL-FTL 의 트랜잭션 L2P 테이블을 정리하기 위해 CHECKPOINT 명령을 주기적으로 수행한다.
1-7) WAL-FTL 이 CHECKPOINT 명령을 받으면 WAL-L2P 테이블에서 COMMIT 된 L2P 정보들을 L2P 테이블에 반영하며, WAL-L2P 테이블에서는 이를 제거한다.

WAL-FTL 은 위와 같은 방식을 통해 호스트가 별도의 백업 혹은 로깅 작업을 하지 않아도 저장장치에 이미 기록되어 있는 데이터를 활용해서 트랜잭션 원소성을 구현할 수 있다. 만일 데이터 업데이트 도중 오류가 발생해서 트랜잭션이 비정상 종료되더라도, WAL-FTL 은 기존 L2P 테이블의 정보 등을 활용하여 DB 파일을 이전 상태로 복원(roll-back)할 수 있다.

또, 체크포인트 동작 시에도 전체 WAL 파일의 유효한 데이터들을 일일이 읽어서 다시 DB 파일에 적느라 상당히 많은 쓰기 동작이 발생하는데 비해, WAL-FTL 에서는 그보다 훨씬 작은 WAL-L2P 정보만 L2P 로 옮기므로 쓰기 연산이 매우 적게 발생한다.

3.2.2. 트랜잭션 읽기 처리

WAL-FTL 의 트랜잭션 읽기 처리 방식은 다음과 같다.

2-0) 각 트랜잭션을 시작할 때, WAL-FTL 로부터 최신 CSN 값을 읽어온다.
2-1) 호스트는 저장장치에서 데이터를 읽을 때 WAL-FTL 에게 CSN 과 읽을 페이지의 주소를 전달한다.
2-2) WAL-FTL 은 WAL-L2P 테이블에서 유효한 CSN 값을 가진 L2P 정보 중에서 논리 주소(logical address)가 일치하며 CSN 값이 전달받은 CSN 보다 작은 값을 가지는 WAL-L2P 를 찾는다.
2-3) 만일 위의 조건을 만족하는 WAL-L2P 엔트리가 있을 경우, 그 L2P 정보를 이용하여 NAND 에서 데이터를 읽어온다.
2-4) 만일 위의 조건을 만족하는 WAL-L2P 엔트리가 없을 경우, L2P 테이블에 있는 L2P 정보를 이용하여 NAND 에서 데이터를 읽어온다.

여기에서 CSN 이란 WAL-L2P 테이블에 있는 각각의 엔트리가 언제 커밋 되었는지를 나타내는 순서 정보이다. 이 값은 처음 기록되고 나서 커밋 되기 이전까지는 무효(invalid)한 값을 가지다가, 트랜잭션이 정상적으로 종료되고 난 직후 해당 트랜잭션이 커밋 된 순서에 따라 값을 할당 받는다.

WAL-FTL 은 CSN 값을 SQLite WAL 의 유효 프레임 번호와 같은 방식으로 활용한다. 즉, WAL-L2P 테이블에서 CSN 값이 현재 트랜잭션의 CSN 값보다 작다면 그 엔트리가 가지고 있는 L2P 정보는 현재 트랜잭션이 시작되기 이전에 커밋 된 정보들이며, 따라서 이를 L2P 변환에 활용한다. 반대로 CSN 값이 트랜잭션의 CSN 값보다 크거나, 무효한 값을 가지고 있다면 현재 트랜잭션이 시작된 이후 다른 트랜잭션에 의해 커밋 된 값이거나, 혹은 아직 커밋 되지 않은 트랜잭션이 업데이트한 값으로 간주해서 이를 무시하는 식으로 동시(concurrent) 접근 환경에서도 트랜잭션의 일관성을 보장할 수 있다.

3.3. 저장장치 자료 구조

TID	L	P	CSN
21	2000	301	10
21	2001	305	10
23	100	400	11
23	101	401	11
21	2003	306	10
24	400	10000	*INV*
24	401	10001	*INV*
⋮			

(그림 2) WAL-L2P 테이블

트랜잭션을 통해 업데이트되는 모든 데이터는 [4]

에서 처럼 L2P 테이블에 바로 반영되지 않고 **오류! 참조 원본을 찾을 수 없습니다.**)와 같은 구조체(WAL-L2P 테이블)에 저장된다.

WAL-L2P 테이블의 각 엔트리에는 다음과 같은 정보들이 저장된다.

- TID: 데이터를 업데이트하는 트랜잭션의 ID
- Logical Address: 트랜잭션 데이터의 논리적 주소(호스트 관점에서의 주소)
- Physical Address: 트랜잭션 데이터가 실제로 저장된 물리적 주소(NAND 상의 위치)
- CSN (commit sequence number): 커밋 순서 정보.

이 밖에도, WAL-FTL 은 현재 저장 장치 내의 최대 CSN 값을 계속 관리한다.

호스트-저장장치간 인터페이스

호스트-저장장치 간에는 기존의 일반적인 저장장치 인터페이스에 아래와 같은 기능을 추가한다.

- TXWRITE(TID, PAGE) 일반적인 저장장치의 WRITE()에 트랜잭션의 ID 정보를 추가한 것으로서, 트랜잭션을 통해 특정 페이지를 저장장치에 기록한다.
- TXREAD(CSN, PAGE) 일반적인 저장장치의 READ() 에 트랜잭션의 커밋 순서 번호(CSN)를 추가한 것이다.
- COMMIT(TID) 주어진 TID 에 해당하는 트랜잭션을 커밋 시킨다.
- ABORT(TID) 주어진 TID 에 해당하는 트랜잭션을 롤백 시킨다.
- CHKPT() 현재까지 커밋 된 정보를 L2P 테이블에 반영한다.
- GET_CSN() 현재 디바이스-CSN 값을 호스트에 리턴 한다.

3.4. 전체 시스템 구조

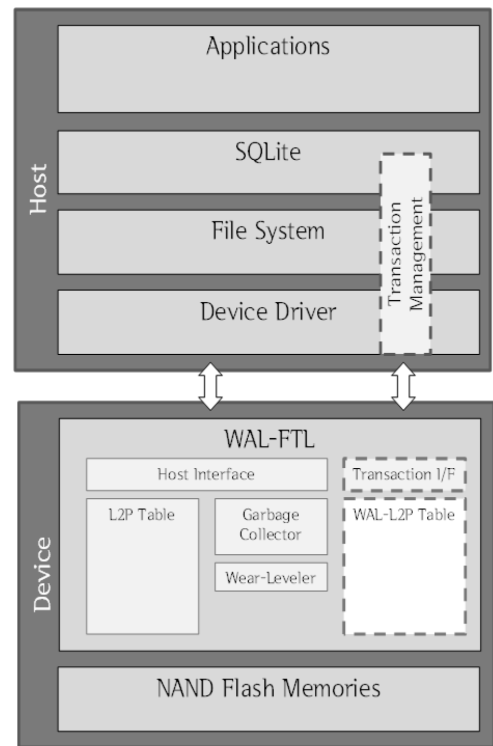
앞의 내용을 종합하여 구성되는 전체적인 시스템의 개념적인 구조는 **오류! 참조 원본을 찾을 수 없습니다.**)과 같다.

4. 결론 및 향후 연구

본 논문에서는 SQLite 의 WAL 의 장점인 높은 트랜잭션 동시성을 가지면서, 동시에 저장장치가 트랜잭션을 지원하여 WAL 의 단점인 체크포인트 오버헤드를 극복하고 트랜잭션 처리 성능을 높이는 FTL 에 대한 개념적인 설계를 제시하였다.

본 논문에서 제시한 아이디어대로 저장장치 및 호스트-저장장치 인터페이스를 구현하게 되면, 다중 쓰레드를 원활하게 지원하면서도 트랜잭션 처리 성능도 높은 새로운 DB 시스템을 구축할 수 있을 것이며, 그에 따라 우리는 좀 더 뛰어난 성능의 스마트 기기들을 만날 수 있게 될 것이다.

향후에는 본 논문에서 제시한 기본적인 설계를 바탕으로 하여 WAL-FTL 및 이를 활용하는 시스템을 구현하고, 이를 다양한 응용 사례에 적용하여 WAL-FTL 의 효율성을 검증할 것이다.



(그림 3) WAL-FTL 을 포함하는 전체 시스템의 구조 점선으로 표시된 부분이 새로 추가되어야 할 모듈들이다.

감사의 글

이 논문은 2012 년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (2012R1A2A1A01010775)

참고문헌

[1] SQLite. *Atomic Commit In SQLite*. Available: http://www.sqlite.org/atomiccommit.html

[2] SQLite. *Write-Ahead Logging*. Available: http://www.sqlite.org/wal.html

[3] P. Sunhwa, Y. Ji Hyun, and O. Seong-Yong, "Atomic write FTL for robust flash file system," in *Consumer Electronics, 2005. (ISCE 2005). Proceedings of the Ninth International Symposium on*, 2005, pp. 155-160.

[4] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: transactional FTL for SQLite databases," presented at the ACM SIGMOD International Conference on Management of Data, New York, New York, USA, 2013.

[5] O. Xiangyong, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block I/O: Rethinking traditional storage primitives," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, 2011, pp. 301-311.

[6] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional flash," presented at the 8th USENIX conference on Operating systems design and implementation, San Diego, California, 2008.