PMDK 라이브러리를 적용한 SQLite 데이터베이스 엔진

박종혁 ○ 김경민 오기환 이상원

성균관대학교

{akindo19, lufovic77, wurikiji, swlee}@skku.edu

SQLite Database Engine with PMDK Library

Jong-Hyeok Park○ Kyungmin Kim Gihwan Oh Sang-Won Lee

Sungkyunkwan University

요 약

지속성 메모리(Persistent Memory)는 휘발성 메모리와 같이 빠른 I/O 속도를 가지며, 비 휘발성 메모리처럼 데이터 지속성을 보장한다. DBMS는 지속성 메모리를 효율적으로 활용하기 위해 Intel에서 제공한 PMDK 라이브러리와 같은 소프트웨어 지원을 활용한다. 본 논문에서는 지속성 메모리를 저장장치로 사용할 때 효율적인 트랜잭션 처리를 위해 SQLite에 PMDK 라이브러리를 적용한 SQLite/PMM을 구현하고, TPC-C벤치마크를 통해 성능향상을 확인하였다. 또한, PMDK 라이브러리를 적용하지 않은 메모리 맵핑 I/O 버전과의 성능 비교를 통해 PMDK 라이브러리의 효율성을 확인하였다.

1. 서 론

DRAM과 같은 휘발성(Volatile) 메모리는 빠른 속도의 읽기와 쓰기가 가능하지만, 전원 결함(Power Failure)과 같은 시스템 충돌이 발생하는 경우, 데이터 손실이 발생한다. 반면, SSD(Solid State Drive)와 같은 비 휘발성(Non-Volatile) 메모리는 읽기와 쓰기 속도가 느린 대신, 업데이트가 발생한 데이터에 대해 지속성(Durability)을 보장한다. DBMS(Data Base Management System)는 두 저장장치의 특성을 최대한 활용하기 위해 빠른 I/O 처리를 할 수 있도록 휘발성 메모리를 데이터 캐싱을 위한 버퍼로 활용하고, 데이터의 지속성을 보장하는 비휘발성 메모리를 저장장치로 활용한다.

NVDIMM(Non-Volatile Dual Inline Memory Module)과 같은 지속성 메모리(Persistent Memory)는 DRAM과 비슷한 읽기 쓰기 속도를 가지며, 데이터의 지속성을 보장할 수 있는 비휘발성을 가진다.[1] 또한, 바이트 단위 접근(Byte Addressable)이 가능하여 블록단위가 아닌 바이트 단위로 I/O를 할 수 있다. DBMS는 휘발성 메모리와 비휘발성 메모리의 장점을 모두 갖춘 지속성 메모리를 효율적으로 사용하기 위해 다양한 소프트웨어 지원을 활용한다. 페이지 캐시와 블록 레이어를 우회하여 메모리 맵핑(Memory Mapping) 을 할 수 있는 EXT4 파일시스템의 DAX 옵션[2]과 Intel에서 오픈소스로 공개한 PMDK(Persistent Memory Development Kit) 라이브러리가 있다[3].

본 논문에서는 PMDK에서 제공하는 Libpmem 라이브러리를 활용하여 NVDIMM을 SQLite의 데이터 및 로그 저장장치로 적용한 SQLite/PMM (Persistent Memory Mapped)을 구현하고, TPC-C 벤치마크[4]로 성능 평가를 수행하였다. 또한, PMDK 라이브러리를 사용하지 않고, Intel에서 제공하는 메모리 접근 순서 보장 인스트럭션을 활용한 메모리 맵(memory mapped) 파일 버전과 성능을 비교하여 PMDK 라이브러리의 효율성을 확인하였다.

본 논문의 구성은 다음과 같다. 2장에서는 Intel에서 제공하는 PMDK 라이브러리에 대해 알아본다. 3장에서는 PMDK에서 제공하는 Libpmem 라이브러리를 적용한 SQLite/PMM의 구현과정 대해 알아본다. 4장에서는 기존의 SQLite와 NVDIMM기반의 SQLite의 성능을 비교한다. 마지막으로 5장에서는 결론을 제시하고 논문을 마무리한다.

2. Persistent Memory Development Kit

PMDK는 NVRAM 어플리케이션 개발을 지원하는 표준적인 오픈소스 라이브러리들의 집합이다. 특히, 트랜잭션 관리 및 지속성 메모리를 인지하는 동적할당을 지원하는 것이 특징이다.

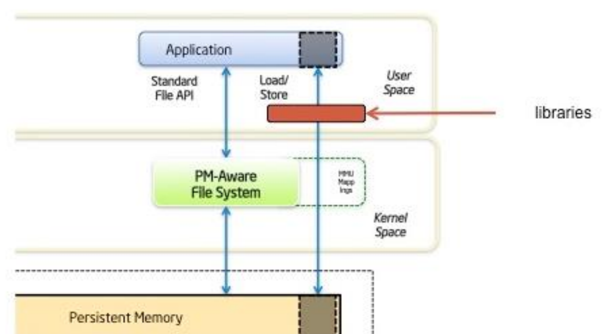

그림 1. PMDK 라이브러리 구조

그림 1은 PMDK 라이브러리의 동작방식을 나타낸다. 지속성 메모리는 지속성 메모리를 지원하는 파일 시스템(PM-Aware Filesystem)을 통해 직접적으로 어플리케이션과 연동된다. 또한, DAX 옵션과 같이 커널의 개입 없이 유저 레벨에서 PMDK 라이브러리를 통해 mmap() 시스템 콜을 이용하여 어플리케이션이 직접적인 load 및 store 연산을 수행한다.

3. SQLite/PMM

그림 2(a)는 기존의 SQLite의 시스템 구조이다. 휘발성 메모리인 DRAM을 버퍼로 사용하고, 데이터 및 로그 파일에 대해 블록 레이어 상에서 File I/O 시스템 콜 함수를 통해 I/O를 수행한다. **그림 2(b)**는 본 논문에서 구현한 SQLite/PMM 시스템 구조이다. NVDIMM상에서 DAX 옵션을 활성화한 ext4 파일시스템을 마운트하고, File I/O가 아닌 Libpmem 라이브러리 함수를 활용해 메모리 맵 I/O를 수행한다. Libpmem 라이브러리는 트랜잭션 기능을 지원하지 않기 때문에 SQLite/PMM에서는 기존의 SQLite에서 제공하는 Rollback Journal 모드[5]를 활용하였다.

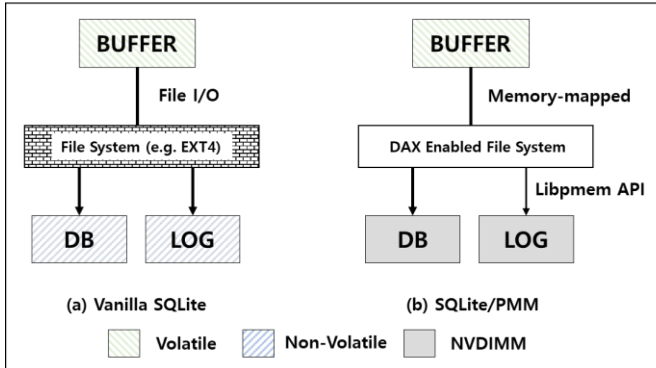

그림 2. 기존 SQLite 와 SQLite/PMM 시스템 구조

표 1은 SQLite에서 파일 읽기 및 쓰기 연산에 사용된 File I/O 함수와 그에 대응하는 Libpmem 라이브러리 함수이다[6]. SQLite/PMM은 SQLite의 동작 과정을 크게 수정하지 않고, 데이터 및 로그 파일에 대한 I/O 연산에 Libpmem 라이브러리 함수를 적용하였다. pmem_memcpy_nodrain()함수는 NVDIMM에 할당된 메모리 맵 파일을 업데이트하고 무조건 저장장치의 캐시를 플러시하는 것이 아니라, 애플리케이션 단에서 지속성 보장을 해야 할 경우에만 pmem_drain()함수를 호출하여 불필요한 메모리 연산의 오버헤드를 제거한다. PMDK 라이브러리 적용함으로써 File I/O 기반의 SQLite의 I/O 동작을 NVDIMM에서 효율적으로 동작할 수 있도록 변환할 수 있었다.

	File I/O (POSIX)	Libpmem (PMDK)
Open	fd=**open**(path,flag, ..)	pmem=**pmem_map_file**(path, len, &mapped) 매개변수로 주어진 경로(path)의 파일에 길이(len)만큼 비휘발성 메모리 영역을 맵핑하는 함수
Write	**write**(fd, buf, count)	**pmem_memcpy_nodrain**(pmem, buf, count) 비휘발성 메모리 영역에 데이터(buf)를 주어진 길이(count)만큼는 함수 하드웨어 버퍼까지 쓰여지는 곳은 보장하지 않음.
Sync	**fdatasync**(fd)	**pmem_drain**() 비휘발성 메모리 영역에 있는 내용을 하드웨어(예: NVDIMM)의 버퍼에 동기화하는 함수
Read	**read**(fd, buf, count)	**memcpy**(buf, pmem, count) pmem영역의 데이터를 메모리 영역으로(buf)로 주어진 길이(count)만큼 복사하는 함수
Close	**close**(fd)	**pmem_unmap**(pmem, len) 지정된 범위(len)의 모든 비휘발성 메모리영역에 대한 맵핑을 삭제하는 함수

표 1. File I/O 함수와 Libpmem 라이브러리 함수

4. 성능 평가

본 논문에는 TPC-C 벤치마크를 통해 기존의 SQLite와 PMDK를 사용하지 않은 메모리 맵 파일을 적용한 버전과 성능을 비교하였다. 저장장치는 Netlist NVvault DDR4 16GB를 사용하였다. 데이터베이스 크기는 10 warehouse이며, 버퍼 크기는 SQLite 기본 설정 값인 2MB로 설정하였다. 자세한 실험환경은 **표 2**와 같다.

OS	Ubuntu 16.04.2 LTS 4.15.0-24
Processor	Intel® Xeon E5-2640 2.60GHz (32 Core)
Memory	32 GB
NVDIMM	Netlist NVvault DDR4 16GB
Benchmark	TPC-C (10 warehouse 1GB)

표 2. 실험환경

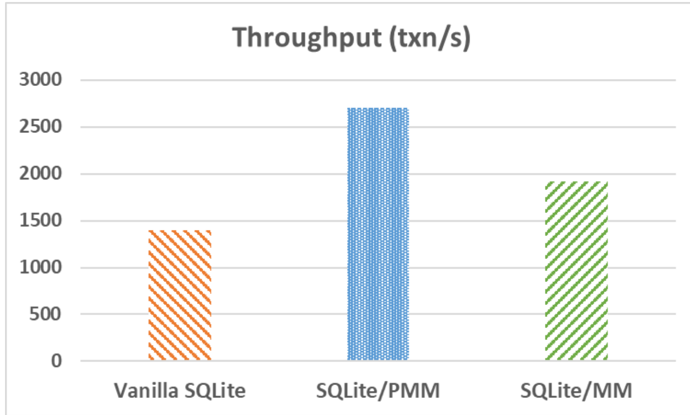

그림 3. TPC-C 실험 결과

Vanilla SQLite의 경우, NVDIMM에 DAX 옵션을 비활성한 뒤, 데이터베이스 파일과 로그 파일을 저장하였다. PMDK를 적용하지 않고 메모리 맵 I/O를 활용한 버전(SQLite/MM)의 경우, 미리 맵핑 영역을 할당하였다. 성능 측정 결과는 **그림 3**과 같다. 기존의 SQLite에 비해 2배 이상, SQLite/MM에 비해 1.4배 성능향상을 확인하였다.

오버헤드가 큰 File I/O가 아닌, 메모리 맵 I/O연산을 수행하기 때문에 Vanilla SQLite보다 트랜잭션 처리량이 증가하였다. SQLite/MM의 경우, 업데이트가 발생할 때마다 맵핑 영역에 대한 잦은 메모리 복사 연산과 메모리 연산의 순서를 보장하기 위한 메모리 배리어 연산을 수행하지만, PMDK 라이브러리는 메모리 복사 연산 및 메모리 배리어 연산의 수행을 효율적으로 처리하기 때문에 성능이 향상되었다.

5. 결론

본 논문에서는 SQLite에 Libpmem 라이브러리를 적용한 SQLite/PMM을 구현하고 기존의 SQLite와 PMDK 라이브러리가 아닌 메모리 맵 I/O를 활용한 버전과 성능을 비교하였다.

성능측정 결과, 기존의 SQLite에 비해 2배 이상 성능 향상을 확인하였고, PMDK를 적용하지 않은 메모리 맵 I/O 방식보다 약 1.4배 성능 향상을 보이는 등 PMDK의 효율성을 확인하였다.

향후연구로는 메모리 복사 오버헤드를 최소화하기 위해 트랜잭션 기능이 지원되는 PMDK의 libpmemobj 등의 라이브러리를 활용하여, 비 휘발성 메모리 없이, 지속성 메모리상에서만 동작하는 데이터베이스 엔진 연구를 수행할 것이다.

참고문헌

[1]Joy Arulraj and Andrew Pavlo. 2017. How to Build a Non-Volatile Memory Database Management System. In Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17).
[2]"Persistent Memory Development Kit Github",https://github.com/pmem/pmdk (2018-08-20 방문)
[3]"Direct Access for files", https://www.kernel.org/doc/Documentation/filesystems/dax.txt (2018-08-20 방문)
[4]"Python TPC-C Github" https://github.com/apavlo/py-tpcc (2018-08-22 방문)
[5]"Temporary Files Used By SQLite", https://www.sqlite.org/tempfiles.html#rollback_journals (2018-08-20 방문)

사사

이 성과는 2018년도 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. 2018R1A2B2005502).

본 연구는 미래창조과학부 및 정보통신기술진흥센터의 SW컴퓨팅산업원천기술개발사업(SW스타랩)의 연구결과로 수행되었음 (IITP-2015-0-00314).

본 연구는 과학기술정보통신부 및 정보통신기술진흥센터의 SW중심대학지원사업의 연구결과로 수행되었음"(2015-0-00914)