과제 #4

2012311596 윤지아 · 2012312945 김동민 · 2015310489 박준호

프로젝트 요약

목적

본 과제의 목적은 기존 Sqlite을 check point를 이용한 no-force 방식으로 변경하고 database의 종료나 비정상적인 상태가 발생할 경우에 physiological log를 이용하여 recovery함으로써 원하는 database를 유지하는 것이다. 기존 sqlite는 force 정책을 사용하여 commit이 발생할 경우 변경된 내용을 disk에 바로 flush하는 방식을 사용한다. 우리는 본 과제에서 no-force 정책으로 바꾸고 상수의 check-point를 사용하여 일정 수준 이상 쌓일 때까지 대기하였다가 한번에 flush하여 성능을 높이고자 하였다. 또한 기존의 쿼리문을 저장하는 logical logging 방식과는 다르게 page number를 함께 저장하는 physiological logging 방식을 이용하여 sqlite cell 구조에 직접 접근하여 recovery를 실행한다. 결과적으로 기존의 sqlite 대비 실행 시간을 측정하고 비교한다. 또한 최종 작성된 결과물 외에 다른 방식으로 접근했던 소스코드와 그 결과에 대해서 부록(Appendix)에서 별도로 다룬다.

Physio-logical Logging 구현

Physiological Logging을 사용하기 위해 제일 먼저 쿼리 결과를 물리적으로 저장시키지 않고 로그파일에만 저장을 해야한다. 우리는 로그파일에 저장할 요소로 log size, log sequence number, opcode, page number, redo log size, redo log data, undo log size, undo log data를 저장한다.

opcode는 어떤 SQL문을 수행하였는지 나타내는데, 0은 Begin, 1은 Insert 2는 Update 3은 Delete 4는 Commit을 나타낸다. redo log data와 undo log data는 cell data를 저장한다.

<그림 1>은 Sqlite3Log 함수이다. 파라미터로 로그에 입력할 값을 받고, memcpy를 통해 log_buffer에 로그 정보를 저장한다. 그리고 commit(opcode=4)이 일어나면 msync를 통해 로그파일에 쓰게된다. Is_open=1일 때, return을 하는 것은 recovery가 일어날 때 btree를 설정하기 위해 쿼리문을 실행하는데, 그 쿼리문에 대한 로깅은 생략하기 위해 사용하는 변수이다. sqlite3Log 함수가 호출될 때마다 p_check 변수가 1씩 증가하는데, 이는 checkpoint에 사용되는 변수이다.

src/btreeInt.h

```
718.void sqlite3Log(Pgno pgno,int opcode, int redo_size, const char
   *redo_log, int undo_size,const char *undo_log){
719.         if(is_open == 1)
720.                 return;
```

```
721.         if(origin_log_buffer == NULL)
722.                 origin_log_buffer = log_buffer;
723.        static void * old_log_buffer;
724.         if(opcode == 0){
725.                 old_log_buffer = log_buffer;
726.         }
727.         int log_size = sizeof(Pgno) + sizeof(int)*3 + sizeof(int)*2 +
   undo_size + redo_size;
728.         void* log = malloc(log_size);
729.         printf("log : %u %d %d %d %d %d %d\n", lastLsn, pgno, opcode,
   redo_size, undo_size, log_size, p_check);
730.         p_check++;
731.         int tmp_size = 0;
732.         memcpy(log+tmp_size, &log_size, sizeof(int));
733.         tmp_size+=sizeof(int);
734.         memcpy(log+tmp_size, &lastLsn, sizeof(int));
735.         tmp_size+= sizeof(int);
736.         lastLsn+=log_size;
737.         memcpy(log+tmp_size, &opcode, sizeof(int));
738.         tmp_size+= sizeof(int);
739.         memcpy(log+tmp_size, &pgno, sizeof(Pgno));
740.         tmp_size+= sizeof(Pgno);
741.         memcpy(log+tmp_size, &redo_size,sizeof(int));
742.         memcpy(log+tmp_size, &redo_size,sizeof(int));
743.         tmp_size+= sizeof(int);
744.         memcpy(log+tmp_size, redo_log, redo_size);
745.         tmp_size+= redo_size;
746.         memcpy(log+tmp_size, &undo_size,sizeof(int));
747.         tmp_size+= sizeof(int);
748.         memcpy(log+tmp_size, undo_log, undo_size);
749.         memcpy(log_buffer, log, log_size);
750.         if(opcode == 4){
751.                 msync(old_log_buffer, log_buffer - old_log_buffer +
   log_size , MS_SYNC);
752.                 old_log_buffer = log_buffer + log_size;
753.         }
754.         log_buffer+=log_size;
755.         free(log);
756.         if(pragma_check != 3)
757.                 pragma_check = 0;
758.};
```

<그림 1> sqlite3Log 함수 구현

<그림2>는 btree.c 파일 안의 sqlite3BtreeInsert() 함수에서 로깅을 하는 부분이다. 8048-8050에서 newCell의 메모리를 할당하고 fillInCell()함수로 데이터를 채워넣는다. 8062-8073에서 제거대상인 oldCell을 찾아서 데이터를 비우고 메모리를 반환한다. 메모리 반환전에 undo_log에 oldCell의 데이터를 저장한다. 8082-8096에서 newCell의 데이터를 redo_log에 저장하고 insertCell로 Page에 데이터를 추가한다. 그리고 sqlite3Log() 함수를 호출함으로써 로깅을 완료한다. loc값은 update와 insert 쿼리문 둘다 sqlite3BtreeInsert() 함수에서 일어나는데 loc가 0인 경우 update이므로 opcode로 2를 부여하고 loc가 1인 경우 insert이므로 opcode로 1을 부여한다.

src/btree.c

```
7962.int sqlite3BtreeInsert(BtCursor *pCur, const BtreePayload *pX, int
   appendBias, int seekResult){
7963.     ……
8048.  assert( newCell!=0 );
8049.  rc = fillInCell(pPage, newCell, pX, &szNew);
8050.  if( rc ) goto end_insert;
8051.  assert( szNew==pPage->xCellSize(pPage, newCell) );
8052.  assert( szNew <= MX_CELL_SIZE(pBt) );
8053.  idx = pCur->aiIdx[pCur->iPage];
8054.  if( loc==0 ){
8055.    u16 szOld;
8056.    assert( idx<pPage->nCell );
8057.    rc = sqlite3PagerWrite(pPage->pDbPage);
8058.    if( rc ){
8059.      goto end_insert;
8060.    }
8061.    oldCell = findCell(pPage, idx);
8062.    if( !pPage->leaf ){
8063.      memcpy(newCell, oldCell, 4);
8064.    }
8065.
8066.    undo_s = szOld;
8067.    undo_log = (char*)malloc(undo_s);
8068.    memcpy(undo_log, oldCell,  undo_s);
8069.
8070.    rc = clearCell(pPage, oldCell, &szOld);
8071.
8072.    dropCell(pPage, idx, szOld, &rc);
8073.    if( rc ) goto end_insert;
8074.  }else if( loc<0 && pPage->nCell>0 ){
8075.    assert( pPage->leaf );
8076.    idx = ++pCur->aiIdx[pCur->iPage];
8077.  }else{
8078.    assert( pPage->leaf );
8079.  }
8080.
8081.  redo_s = szNew + sizeof(int);
8082.  redo_log = (char*)malloc(redo_s);
8083.  memcpy(redo_log, &idx, sizeof(int));
8084.  memcpy(redo_log+sizeof(int), newCell, redo_s-sizeof(int));
8085.
8086.  if(loc != 0){
8087.          undo_s = 0;
8088.          undo_log = (char*)malloc(1);
8089.  }
8090.
8091.  sqlite3Log(pPage->pgno, loc==0?2:1, redo_s, redo_log, undo_s,
   undo_log);
8092.  free(redo_log);
8093.  free(undo_log);
8094.
8095.  insertCell(pPage, idx, newCell, szNew, 0, 0, &rc);
8096.     ……
```

<그림 2> sqlite3BtreeInsert 함수 구현

No-Force 정책와 checkpoint 구현

src/btree.c

```
3868.int sqlite3BtreeCommit(Btree *p){
3869.  sqlite3BtreeEnter(p);
3870.   rc = sqlite3BtreeCommitPhaseOne(p, 0);
3871.   if( rc==SQLITE_OK ){
3872.     rc = sqlite3BtreeCommitPhaseTwo(p, 0);
3873.   }
3874.   sqlite3BtreeLeave(p);
3875.   return rc;
3876.}
```

<그림 3> sqlite3BtreeCommit 함수

sqlite에서는 commit시에 <그림 3>과 같이 sqlite3BtreeCommitPhaseOne, sqlite3BtreeCommitPhaseTwo를 차례대로 호출한다. <그림 4>의 CommitPhaseOne은 sqlite3PagerCommitPhaseOne을 불러서 RBJ모드일 경우에는 RBJ에 data를 쓰고, DB로 옮기는 작업을 한다. 반면 WAL모드일 경우에는 wal frame을 Wal-file에 쓰는 작업을 처리한다. 이 때 <그림 5>의 sqlite3PagerCommitPhaseOne에서는 PcacheDirtyList를 통해 Dirty된 page들을 가져오며, 그러한 page들을 pagerWalFrames를 통해 Wal frame으로 변환하고 disk에 쓰는 과정을 거친다.

src/btree.c

```
3730.int sqlite3BtreeCommitPhaseOne(Btree *p, const char *zMaster){
3731.   int rc = SQLITE_OK;
3732.   if(p_check >= 1000 || pragma_check >= 1){
3733.   if( p->inTrans==TRANS_WRITE ){
3734.     BtShared *pBt = p->pBt;
3735.     sqlite3BtreeEnter(p);
3736.#ifndef SQLITE_OMIT_AUTOVACUUM
3737.     if( pBt->autoVacuum ){
3738.       rc = autoVacuumCommit(pBt);
3739.       if( rc!=SQLITE_OK ){
3740.         sqlite3BtreeLeave(p);
3741.         return rc;
3742.       }
3743.     }
3744.     if( pBt->bDoTruncate ){
3745.       sqlite3PagerTruncateImage(pBt->pPager, pBt->nPage);
3746.     }
3747.#endif
3748.     rc = sqlite3PagerCommitPhaseOne(pBt->pPager, zMaster, 0);
3749.     sqlite3BtreeLeave(p);
3750.   }
3751.   }
3752.   return rc;
3753.}
```

<그림 4> sqlite3BtreeCommitPhaseOne 함수 구현

src/pager.c

```
6185.int sqlite3PagerCommitPhaseOne(Pager *pPager, const char *zMaster, int
   noSync){
……
6219.    if( pagerUseWal(pPager) ){
6220.      PgHdr *pList = sqlite3PcacheDirtyList(pPager->pPCache);
6221.      PgHdr *pPageOne = 0;
6222.      if( pList==0 ){
6223.        /* Must have at least one page for the WAL commit flag.
6224.        ** Ticket [2d1a5c67dfc2363e44f29d9bbd57f] 2011-05-18 */
6225.        rc = sqlite3PagerGet(pPager, 1, &pPageOne, 0);
6226.        pList = pPageOne;
6227.        pList->pDirty = 0;
6228.      }
6229.      assert( rc==SQLITE_OK );
6230.      if( ALWAYS(pList) ){
6231.        rc = pagerWalFrames(pPager, pList, pPager->dbSize, 1);
6232.      }
……
```

<그림 5> sqlite3PagerCommitPhaseOne 함수

CommitPhaseTwo에서는 Commit을 하는동안 Btree와 BtShared 잡고 있었던 Lock들을 풀어준다. sqlite3PagerCommitPhaseTwo에서는 pPager->eState를 PAGER_READER로 바꾸어 Writer lock을 풀어준다.

src/btree.c

```
3821.int sqlite3BtreeCommitPhaseTwo(Btree *p, int bCleanup){
3822.  if(p_check >= 1000 ||  pragma_check >=1){
3823.    if(pragma_check == 3)
3824.      pragma_check = 0;
3825.      p_check = 0;
3826.
3827.  if( p->inTrans==TRANS_NONE ) return SQLITE_OK;
3828.  sqlite3BtreeEnter(p);
3829.  btreeIntegrity(p);
3830.
3831.  /* If the handle has a write-transaction open, commit the shared-
   btrees
3832.  ** transaction and set the shared state to TRANS_READ.
3833.  */
3834.  if( p->inTrans==TRANS_WRITE ){
3835.    int rc;
3836.    BtShared *pBt = p->pBt;
3837.    assert( pBt->inTransaction==TRANS_WRITE );
3838.    assert( pBt->nTransaction>0 );
3839.    rc = sqlite3PagerCommitPhaseTwo(pBt->pPager);
3840.……
3841.}
```

<그림 6> sqlite3BtreeCommitPhaseTwo 함수 구현

Checkpoint는 No-force 정책에서 일정량의 데이터가 메모리에 쓰인 경우에 한꺼번에 데이터 베데이터 저장하는 것을 뜻한다. sqlite3에서 sqlite3BtreeCommit() 함수를 통해 commit이 일어나는데 이 때, 로깅을 할 때마다 증가했던 p_check 변수의 값이 일정치에 도달 할때까지 commit이 일어나지 않게 하는데 예외로 pragma 관련 쿼리문이 입력됐을 경우에는 commit이 일어나게 한다.(예를 들어 pragma journal_mode = wal) 이 점을 이용하여 우리가 commit 타이밍을 제어할 수 있게된다.

우리는 Sqlite의 Force정책을 No-force로 바꾸고, Log force at commit을 구현하기 위해서 p_check와 pragma_check라는 변수를 두어 앞서 설명한 것처럼 <그림 1>의 p_check는 logging 하는 함수에서 증가 시켜주고, pragma_check는 pragma 값이 변할 때 마다 Commit을 불러 Memory의 Data가 disk에 sync하도록 구현하였다. <그림 3>에서 3732와 <그림 4>에서 3823 처럼 CommitPhaseOne과 CommitPhaseTwo에서 볼 수 있듯이, p_check의 값이 1000이 넘어가기 전까지는 disk와 Memory의 내용이 sync되지 않는다. Log force at commit 같은 경우 <그림 1>에서 750과 같이 Commit opcode가 들어온 경우에 Log file을 Msync를 통해 내려보내도록 구현하였다. sync를 실행한 후에는 checkpoint를 새로 설정하고 로그 파일에 들어있는 정보가 불필요 하기 때문에 비워준다.

Recovery 구현

기존에 No-force 정책에서 sqlite를 사용 중 예상치 못한 종료가 일어났을 경우 변경된 데이터를 데이터베이스에 저장하지 못한 경우 다시 sqlite를 실행하면 변경된 데이터들이 저장되어 있지않다. Physiological Logging에서 Recovory는 이런 상황을 피하기위해 로그파일에 저장된 로그를 바탕으로 redo, undo를 시행해서 종료가 되기전 정상적인 데이터로 복원하는 역할을 한다.

Sqlite3_open()는 main.c에서 sqlite가 실행될 때 호출되는 함수이다. log file을 열어서 log_buffer에 로그 파일 데이터를 복사하고 openDatabase() 함수를 호출한다. openDatabase() 함수에서 recovery를 수행하는 부분이다. sqlite3_open() 함수를 통해 호출되며 로그 파일에서 데이터를 복사한 log_buffer안의 데이터를 한줄 한줄 복사하여 recovery를 시행한다. 2966-2971에서 쿼리문을 통해 Btree를 초기화하고 읽어 들이는데, 이 쿼리문에 대한 로깅을 하지 않기위에 is_open 변수를 활성화 시킨다. 2979-3002에서 log_butter의 로그들을 한줄 한줄 읽어서 변수들에 입력을 하는데, 현재 개발이 insert에 대한 redo만 개발 되어있어서 opcode가 1인 경우에만 recovory를 진행한다. 3003-3013에서는 log에서 가져온 데이터를 통해 insertCell에 쓰이는 newCell에 메모리를 할당하고 데이터를 복사한다. 그리고 insertCell을 수행함으로써 테이블에 데이터를 추가한다. 3014-3015에는 recovery를 통해 입력된 데이터들이 commit을 통해 데이터베이스에 쓰게 하기위해 상태를 dirty로 변경시킨다.

Recovery 작업이 끝난 뒤 pragma에 대한 쿼리문을 실행함으로써 commit일 일어나게 강제한다. 그리고 log_buffer를 초기화 함으로써 recovery는 끝나게 된다.

src/main.c

```
3141.int sqlite3_open(
3142.  const char *zFilename,
3143.  sqlite3 **ppDb
3144.  ){
3145.        char logFilename[40];
3146.        strncpy(logFilename,zFilename,36);
3147.        int logNameLen = strlen(logFilename);
3148.        strcat(logFilename, ".log");
3149.        log_fd = open(logFilename, O_RDWR | O_CREAT, 0644);
3150.        if(log_fd < 0){
3151.                fprintf(stderr, "LOG FILE OPEN ERROR\n");
3152.        }else{
3153.                ftruncate(log_fd,1024*4096);
3154.        }
3155.        origin_log_buffer = log_buffer = (void*) mmap(NULL, 1024*4096,
   PROT_READ | PROT_WRITE, MAP_SHARED, log_fd,0);
3156.        //memset(log_buffer, 0x00, 1024*4096);
3157.        //msync(log_buffer, 1024*4096, MS_SYNC);
3158.        if(log_buffer == MAP_FAILED){
3159.                fprintf(stderr, "LOG FILE MAPPING ERROR\n");
3160.        }
3161.        return openDatabase(zFilename, ppDb,
3162.                        SQLITE_OPEN_READWRITE | SQLITE_OPEN_CREATE, 0);
3163.}
```

<그림 7> sqlite3_open 함수 구현

src/main.c

```
static int openDatabase(
  const char *zFilename, /* Database filename UTF-8 encoded */
  sqlite3 **ppDb,        /* OUT: Returned database handle */
  unsigned int flags,    /* Operational flags */
  const char *zVfs       /* Name of the VFS to use */
){
……
2966.  is_open = 1;
2967.  char tempsql0[100]="PRAGMA journal_mode=wal;";
2968.  char tempsql1[100]="select * from test;";
2969.  sqlite3_exec(db,tempsql0,0,0, &zErrMsg);
2970.  sqlite3_exec(db,tempsql1,0,0, &zErrMsg);
2971.  is_open = 0;
2972.
2973.  int lastLsn, log_size, opcode, redo_size, undo_size, idx;
2974.  char * redo_log, *undo_log;
2975.  MemPage* pPage;
2976.  u8* newCell;
2977.  Pgno pgno;
2978.  while(1){
2979.          memcpy(&log_size,log_buffer,sizeof(int));
```

```
2980.           printf("REcovery log %d\n",log_size);
2981.           if(log_size == 0)
2982.                   break;
2983.           log_buffer+=sizeof(int);
2984.           memcpy(&lastLsn,log_buffer,sizeof(int));
2985.           log_buffer+=sizeof(int);
2986.           memcpy(&opcode,log_buffer,sizeof(int));
2987.           log_buffer+=sizeof(int);
2988.           memcpy(&pgno,log_buffer, sizeof(Pgno));
2989.           log_buffer+=sizeof(Pgno);
2990.           memcpy(&redo_size,log_buffer,sizeof(int));
2991.           log_buffer+=sizeof(int);
2992.           redo_log = (char*)malloc(sizeof(char)*redo_size);
2993.           memcpy(redo_log, log_buffer,sizeof(char)*redo_size);
2994.           log_buffer+=sizeof(char)*redo_size;
2995.           memcpy(&undo_size,log_buffer,sizeof(int));
2996.           log_buffer+=sizeof(int);
2997.           undo_log = (char*)malloc(sizeof(char)*undo_size);
2998.           memcpy(undo_log, log_buffer,sizeof(char)*undo_size);
2999.           log_buffer+=sizeof(char)*undo_size;
3000.           if(opcode != 1)
3001.                   continue;
3002.           sqlite3BtreeEnter(db->aDb[0].pBt);
3003.           allocateTempSpace(db->aDb[0].pBt->pBt);
3004.           newCell = db->aDb[0].pBt->pBt->pTmpSpace;
3005.           btreeGetPage(db->aDb[0].pBt->pBt, pgno, &(pPage), 0);
3006.
3007.
3008.           memcpy(&idx, redo_log, sizeof(int));
3009.           memcpy(newCell, redo_log + sizeof(int), redo_size -
   sizeof(int));
3010.
3011.           insertCell(pPage, idx, newCell, redo_size-sizeof(int), 0, 0,
   &rc);
3012.           sqlite3BtreeLeave(db->aDb[0].pBt);
3013.           pPage->pDbPage->pPager->eState = PAGER_WRITER_FINISHED;
3014.           sqlite3PcacheMakeDirty(pPage->pDbPage);
3015.   }
3016.
3017.   db->aDb[0].pBt->inTrans=TRANS_WRITE;
3018.   pragma_check = 1;
3019.   is_open = 1;
3020.   sqlite3_exec(db,tempsql0,0,0, &zErrMsg);
3021.   is_open = 0;
3022.   db->aDb[0].pBt->inTrans=TRANS_NONE;
3023.   log_buffer = origin_log_buffer;
3024.   memset(log_buffer, 0x00, 1024*4096);
3025.   msync(log_buffer, 1024*4096, MS_SYNC);
3026.……
3027.}
```

<그림 8> openDatabase 함수 구현

Overflow 처리

sqlite에서는 처음에 한 테이블당 한 개의 페이지를 가지고 있고, 이는 Btree를 통해 관리를 하는데 페이지가 가득차는 경우에 split이 발생하여 한 테이블에서 쓰는 페이지가 늘어나게 된다. 이런 경우에 page number가 달라지기 때문에 physiological하게 logging 해서 recovery를 하기가 어려운데, 그 문제를 해결하기위해 우리는 split이 발생하는 쿼리문에서 바로 checkpoint를 시행하게 한다. Sqlite3BtreeInsert() 함수에서 데이터를 추가하다보면 페이지에 오버플로우가 발생하여 nOverflow 변수가 활성화된다. 그런 경우에 btree의 balance가 일어나게된다.

checkpoint를 부분에서 pragma_check 변수를 이용하여 강제로 checkpoint를 일어나게 할 수 있는데 pragma_check 변수에 기존의 pragma에 대해 부여하는 1을 주면 sqlite3BtreeCommit()에 도달하기전에 pragma_check 변수의 값이 0으로 변하여 checkpoint가 일어나지 않는 문제가 발생하였다. 그래서 overflow가 발생했을 때를 나타내는 3을 부여하여 sqlite3BtreeCommit()에 도달하기전에 pragma_check가 0이 되는 문제를 해결하였다. 이를 통해 sqlit이 발생하는 쿼리문이 마치면서 checkpoint도 자동으로 일어나서 굳이 overflow가 일어날 때에 대한 로깅을 할 필요성이 없어졌다.

src/btree.c

```
3141.int sqlite3BtreeInsert(BtCursor *pCur, const BtreePayload *pX, int
   appendBias, int seekResult){
3142.     ……
8120.  pCur->info.nSize = 0;
8121.  if( pPage->nOverflow ){
8122.    assert( rc==SQLITE_OK );
8123.    pCur->curFlags &= ~(BTCF_ValidNKey);
8124.    rc = balance(pCur);
8125.
8126.    printf("\n***overflow***\n");
8127.    /* Must make sure nOverflow is reset to zero even if the balance()
8128.    ** fails. Internal data structure corruption will result otherwise.
8129.    ** Also, set the cursor state to invalid. This stops
   saveCursorPosition()
8130.    ** from trying to save the current position of the cursor.  */
8131.    pCur->apPage[pCur->iPage]->nOverflow = 0;
8132.    pCur->eState = CURSOR_INVALID;
8133.    pragma_check = 3;
8134.  }
8135.  assert( pCur->apPage[pCur->iPage]->nOverflow==0 );
8136.     ……
```

<그림 9> overflow 처리

결과 분석

time 측정

우리는 trigger를 두어서 test table의 첫번째 컬럼은 index로, 두번째 컬럼의 값은 반복의 시작 번호로 지정하고 1000까지 서로 다른 수를 자동으로 입력하도록 하여 실험하였다. 이 때 <그림 11>, <그림 12>와 같은 shell script와 sql을 만들어서 비교적 수월하게 실험하도록 자동화하였다. 하지만 recursive에 대한 최대치가 sqlite에서 정해져 있기 때문에 500개씩만 입력하였다. 결과적으로 60000개의 값을 test 테이블에 삽입하여 그 시간을 측정하였다.

```
ga@ubuntu:~/sqlite/sqlite3_logging/sqlite-src-3140100_$ ./sqlite3 test.db
SQLite version 3.14.1 2016-08-11 18:53:32
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE test (a int, b int, CONSTRAINT pk_test PRIMARY KEY (a));
CREATE TRIGGER test1_ins_trigger after insert on test
when new.b < 1000 begin
insert into test values(new.a+1,new.b+1);
end;
sqlite>
```

<그림 10> trigger 설정

test.sh

```
time ./sqlite3 test.db < test_query.sql
```

<그림 11> 자동 실험 설정

test_query.sql

```
pragma recursive_triggers = 1;
delete from test;
insert into test values (0, 500);
insert into test values (501, 499);
insert into test values (1001, 499);
insert into test values (1501, 499);
insert into test values (2001, 499);
……
insert into test values (59501, 499);
```

<그림 12> trigger 선언 및 간략화된 insert query

no_force	force
real0m0.487s user0m0.396s sys 0m0.012s	real0m0.478s user0m0.380s sys 0m0.032s

<그림 13> 실험 결과

측정 결과는 <그림 13>과 같다. no_force의 경우 overflow가 발생할 경우 한번에 sync 하기 때문에 중간 과정에서 disk에 직접 쓰는 과정은 생략되기 때문에 성능 향상을 기대했었다. 하지만 이를 위해 중간 단계에서 log를 남기면서 log 파일로의 sync를 진행하기 때문에 결과적으로는 실행시간이 비슷하게 나왔다고 판단된다.

APPENDIX

Cell 구조와 Cell 단위의 Data Manage 함수

<그림 14>와 <그림 15>는 FillinCell의 한 부분으로 BtreePayload *pX를 이용해 new Cell을 구성하고 있는 부분이다. 먼저 pPage->intKey란 pPage가 index를 담고있는 page인가에 대한 변수인데, intKey가 0이면 index를 담고 있는 Page이고, index가 1이면 일반적인 Data를 담고 있는 Page가 된다. Header에는 뒤의 Payload의 길이가 어느정도인지를 나타내는 nPayload, Cell의 번호가 몇번인지를 나타내는 nKey등이 들어있다. 그 뒤에는 Payload에 Data를 넣기 위한 작업을 한다.

```
/* Fill in the header. */
nHeader = pPage->childPtrSize;
if( pPage->intKey ){
  nPayload = pX->nData + pX->nZero;
  pSrc = pX->pData;
  nSrc = pX->nData;
  assert( pPage->intKeyLeaf ); /* fillInCell() only called for leaves */
  nHeader += putVarint32(&pCell[nHeader], nPayload);
  nHeader += putVarint(&pCell[nHeader], *(u64*)&pX->nKey);
}else{
  assert( pX->nData==0 );
  assert( pX->nZero==0 );
  assert( pX->nKey<=0x7fffffff && pX->pKey!=0 );
  nSrc = nPayload = (int)pX->nKey;
  pSrc = pX->pKey;
  nHeader += putVarint32(&pCell[nHeader], nPayload);
}
```

<그림 14> FillInCell 일부분

```
/* Fill in the payload */
if( nPayload<=pPage->maxLocal ){
  n = nHeader + nPayload;
  testcase( n==3 );
  testcase( n==4 );
  if( n<4 ) n = 4;
  *pnSize = n;
  spaceLeft = nPayload;
  pPrior = pCell;
}else{
  int mn = pPage->minLocal;
  n = mn + (nPayload - mn) % (pPage->pBt->usableSize - 4);
  testcase( n==pPage->maxLocal );
  testcase( n==pPage->maxLocal+1 );
  if( n > pPage->maxLocal ) n = mn;
  spaceLeft = n;
  *pnSize = n + nHeader + 4;
  pPrior = &pCell[nHeader+n];
}
pPayload = &pCell[nHeader];
```

<그림 15> FillInCell 일부분

nPayload는 payload의 길이, pPayload는 payload에 들어갈 data, spaceLeft는 pPayload에 남은 공간을 의미한다. pPayload가 가득 찰시 일단은 Ovfl(overflow)Page를 두어서 그곳에 쓰고 나중에 split이 일어날 수 있도록 한다. 반면 그렇지 않다면 남은 Data들을 pPayload에 쓴다.

```
**   nPayload          Total payload size in bytes
**   pPayload          Begin writing payload here
**   spaceLeft         Space available at pPayload.  If nPayload>spaceLeft,
**                     that means content must spill into overflow pages.
**   *pnSize           Size of the local cell (not counting overflow pages)
**   pPrior            Where to write the pgno of the first overflow page
**
```

<그림 16> 각 성분에 대한 설명

spaceLeft에 의해 split이 날 수 있다는 점과 nKey가 변화할 수 있는 상황(Primary Key가 지워지고 새로들어오는 상황)이 있지 않을까 라고 생각해서 우리 조는 pX를 logging하여 recovery를 하고자 했다. (최근 실험해 본 결과 nKey값은 Primary Key와는 관계가 없고 Table에 들어온 순서를 의미하는 값임을 알게 되었음.) 하지만, Cell단위 함수에서 split에 대한 처리를 해주지 않고 그러한 처리에 대한

책임을 balance에 넘겨버리기 때문에 balance를 따로 호출해야만 했고 이부분은 balance를 할 때, Force를 함으로써 해결할 수 있었다.

```
  /* Write the payload into the local Cell and any extra into overflow pages */
  while( nPayload>0 ){
    if( spaceLeft==0 ){
#ifndef SQLITE_OMIT_AUTOVACUUM
      Pgno pgnoPtrmap = pgnoOvfl; /* Overflow page pointer-map entry page */
      if( pBt->autoVacuum ){
        do{
          pgnoOvfl++;
        } while(
          PTRMAP_ISPAGE(pBt, pgnoOvfl) || pgnoOvfl==PENDING_BYTE_PAGE(pBt)
        );
      }
#endif
```

<그림 17> 기존의 overflow 처리 과정