VT Gerçeklenmesi Ders Notları-#3

Remote: Kullanıcıdan gelen JDBC isteklerini karşılar.

Planner: SQL ifadesi için işleme planı oluşturur ve karşılık gelen ilşkisel cebir ifadesini oluşturur.

Parse: SQL ifadesindeki tablo, nitelik ve ifadeleri ayrıştırır.

Query: Algebra ile ifade edilen sorguları gerçekler.

Metadata: Tablolara ait katalog bilgilerini organize eder.

Record: disk sayfalarına yazma/okumayı kayıt seviyesinde gerçekler.

Transaction&Recovery: Eşzamanlılık için gerekli olan disk sayfa erişimi kısıtlamalarını organize eder ve veri kurtarma için kayıt_defteri (*log*) dosyalarına bilgi girer.

Buffer: En sık/son erişilen disk sayfalarını ana hafıza tampon bölgede tutmak için gerekli işlemleri yapar.

Log: Kayıt_defterine bilgi yazılmasını ve taranması işlemlerini düzenler.

File: Dosya blokları ile ana hafıza sayfaları arasında bilgi transferini organize eder.

- VTYS'nin hafıza yönetimi prensipleri
- *SimpleDB'de* log yönetimi
- *SimpleDB'de* hafıza yönetimi algoritmaları

- *Kaynak: "database design and implementation" by Edward Sciore, 2009, John Wiley*

VTYS Hafıza Yönetimi

- Yönetim'de amaç : Disk erişim sayısının azaltılması
- İşletim sisteminin sözde hafıza(*virtual mem.*) yönetimi yetmez mi?
  - Cevap: 2 önemli nedenden dolayı kesinlikle yetmez:
    - VTYS'nin kurtarma sisteminin (*recovery manag.*), sayfaların diske yazılmasında bazı sırayı takip etmesi gerekir. Örneğin, değişikliğe maruz kalan *veri bloğu* ve bu değişiklik için gerekli olan log kaydını içeren *log bloğunu* düşünelim. Sizce bu bloklardan hangisi daha önce diske yazılmalıdır? Peki, işletim sistemi bunu sağlayabilir mi?
      - İşletim sistemi, bir bloğun şu an için kullanılıp/kullanılmadığından habersiz. Takip ettiği şey, yerleştirme algoritmasına göre sayfa hakkında bazı istatistikler..Oysa VTYS bir *hareket(transaction*) süresince hangi sayfaların kullanacağını biliyor ve bunların ana hafıza kalmasını istiyor...
- VTYS kendi tampon havuzunu (*buffer pool*), veri tabanı kullanımına ve sistemin diğer modüllerine göre en optimum olarak kullanır.
- Şu an için elimizde iki tür bilgi var: "**kullanıcı verisi**" ve "**log**". Bunları yöneten modüller:
  - Log yönetim modülü (*simpledb.log*)
  - Tampon yönetim modülü (*simpledb.buffer*)

Log Yönetim modülü

- VT durumlarının takip edilmesi için;
  - Her değişiklik bir log kaydı ile saklanmalı
  - Her log kaydı her zaman log dosyasının sonuna eklenmeli
- Log kayıtlarını kim okur?
  - Log yönetim modülü DEĞİL!
  - VTYS kurtarma yönetimi (*recovery manager*) modulü okur ve değerlendirir. Zaten log kayıtlarını, LogMgr'a yazdıran da gene kurtarma modulüdür.
- Aşağıdaki log yönetimi algoritmasını inceleyin ve **sorunu bulun**:

1. Allocate a page in memory.
2. Read the last block of the log file into that page.
3. If the log record fits in the page, then:
   a) Add the log record to the end of the page.
   b) Write the page back to disk.
4. If the log record does not fit in the page, then:
   a) Allocate a new, empty page.
   b) Add the log record to that page.
   c) Append the page to a new block at the end of the log file.

Figure 13-1
A simple (but inefficient) algorithm for appending a new record to the log

Log Yönetim algoritması

1. Permanently allocate one memory page to hold the contents of the last block of the log file. Call this page P.
2. When a new log record is submitted:
   a) If there is no room in P, then:
      Write P to disk and clear its contents.
   b) Add the new log record to P.
3. When the database system requests that a particular log record be written to disk:
   a) If that log record is in P, then:
      Write P to disk.

Figure 13-3
The optimal log management algorithm

- Buna göre 2 durumda log sayfası diske yazılması gerekiyor:
  1. Sayfa kapasitesi dolduğu zaman
  2. Log sayafası içerisindeki bir log kaydı, VTYS kurtarma modulü tarafından diske yazılmaya zorlandığı zaman..
- Buna göre bir log sayfası, birden çok defa diske yazılmak zorunda kalabilir mi?

SimpleDB log yönetim modulü

LogMgr

```
public LogMgr(String logfile);
public int    append(Object[] rec);
public void   flush(int lsn);
public Iterator<BasicLogRecord> iterator();
```

BasicLogRecord

```
public BasicLogRecord(Page pg, int pos);
public int    nextInt();
public String nextString();
```

Figure 13-4
The API for the SimpleDB log manager

```
SimpleDB.initFileAndLogMgr("studentdb");
LogMgr logmgr = SimpleDB.logMgr();
int lsn1 = logmgr.append(new Object[]{"a", "b"});
int lsn2 = logmgr.append(new Object[]{"c", "d"});
int lsn3 = logmgr.append(new Object[]{"e", "f"});
logmgr.flush(lsn3);

Iterator<BasicLogRecord> iter = logmgr.iterator();
while (iter.hasNext()) {
   BasicLogRecord rec = iter.next();
   String v1 = rec.nextString();
   String v2 = rec.nextString();
   System.out.println("[" + v1 + ", " + v2 + "]");
}
```

(a) An example of the code fragment

- Bir SimpleDB örneği(instance), sadece bir adet LogMgr nesnesine sahiptir.
- Log kaydı, içerisinde int ve String tipinde veri tutan değişken uzunluklu bir kayıt.
- Eklenen herbir log kaydına ait ***lsn*** numarası vardır. Şu an için, bu ***lsn*** numarası, log kaydının içinde bulunduğu sayfaya karşılık gelen bloğun numarasıdır.
- *flush(lsn)* fonksiyonu *lsn* ve öncesindeki bütün log kayıtlarını diske yazar.
- *iterator()* fonksiyonu log kayıtlarını sondan başa doğru tarar. (*çünkü kurtarma modülü böyle istiyor..)*
- Yandaki örnek kodun çıktısı ne olur?

SimpleDB LogMgr gerçeklenmesi

```
public class LogMgr implements Iterable<BasicLogRecord> {
   public static final int LAST_POS = 0;

   private String logfile;
   private Page mypage = new Page();
   private Block currentblk;
   private int currentpos;

   public LogMgr(String logfile) {
      this.logfile = logfile;
      int logsize = SimpleDB.fileMgr().size(logfile);
      if (logsize == 0)
         appendNewBlock();
      else {
         currentblk = new Block(logfile, logsize-1);
         mypage.read(currentblk);
         currentpos = getLastRecordPosition() + INT_SIZE;
      }
   }

   public void flush(int lsn) {
      if (lsn >= currentLSN())
         flush();
   }

   public Iterator<BasicLogRecord> iterator() {
      flush();
      return new LogIterator(currentblk);
   }

   public synchronized int append(Object[] rec) {
      int recsize = INT_SIZE;
      for (Object obj : rec)
         recsize += size(obj);
      if (currentpos + recsize >= BLOCK_SIZE){
         flush();
         appendNewBlock();
      }
      for (Object obj : rec)
         appendVal(obj);
      finalizeRecord();
      return currentLSN();
   }
```

Figure 13-6
The code for the SimpleDB class *LogMgr*

OS'nin tamponlarını kullanıyor

Eski blokları boşuna yazmıyor...

Log kayıtlarının geriye doğru birbirine zincirleme bağlanmasını gerçekleştirir. *(Bu aşağıdaki örnekte gösterilmiştir.)*

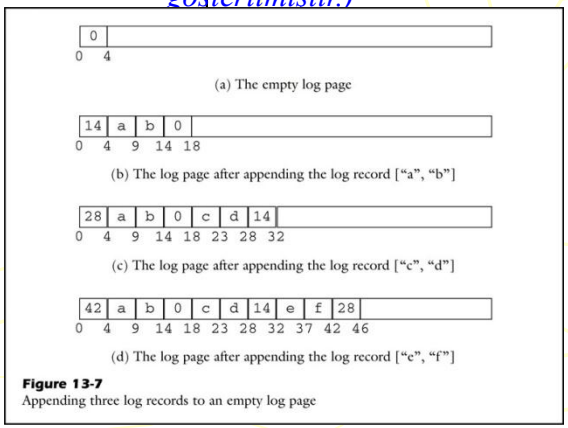

Figure 13-7
Appending three log records to an empty log page

```
   private void appendVal(Object val) {
      if (val instanceof String)
         mypage.setString(currentpos, (String)val);
      else
         mypage.setInt(currentpos, (Integer)val);
      currentpos += size(val);
   }

   private int size(Object val) {
      if (val instanceof String) {
         String sval = (String) val;
         return STR_SIZE(sval.length());
      }
      else
         return INT_SIZE;
   }

   private int currentLSN() {
      return currentblk.number();
   }

   private void flush() {
      mypage.write(currentblk);
   }

   private void appendNewBlock() {
      setLastRecordPosition(0);
      currentblk = mypage.append(logfile);
      currentpos = INT_SIZE;
   }

   private void finalizeRecord() {
      int lastpos = getLastRecordPosition();
      mypage.setInt(currentpos, lastpos);
      setLastRecordPosition(currentpos);
      currentpos += INT_SIZE;
   }

   private int getLastRecordPosition() {
      return mypage.getInt(LAST_POS);
   }

   private void setLastRecordPosition(int pos) {
      mypage.setInt(LAST_POS, pos);
   }
}
```

Figure 13-6 *(Continued)*

LogIterator sınıfının gerçeklenmesi

```
class LogIterator implements Iterator<BasicLogRecord> {
   private Block blk;
   private Page pg = new Page();
   private int currentrec;

   LogIterator(Block blk) {
      this.blk = blk;
      pg.read(blk);
      currentrec = pg.getInt(LogMgr.LAST_POS);
   }

   public boolean hasNext() {
      return currentrec > 0 || blk.number() > 0;
   }

   public BasicLogRecord next() {
      if (currentrec == 0)
         moveToNextBlock();
      currentrec = pg.getInt(currentrec);
      return new BasicLogRecord(pg, currentrec+INT_SIZE);
   }

   public void remove() {
      throw new UnsupportedOperationException();
   }

   private void moveToNextBlock() {
      blk = new Block(blk.fileName(), blk.number()-1);
      pg.read(blk);
      currentrec = pg.getInt(LogMgr.LAST_POS);
   }
}
```

Figure 13-8
The code for the SimpleDB class *LogIterator*

```
public class BasicLogRecord {
   private Page pg;
   private int pos;

   public BasicLogRecord(Page pg, int pos) {
      this.pg = pg;
      this.pos = pos;
   }

   public int nextInt() {
      int result = pg.getInt(pos);
      pos += INT_SIZE;
      return result;
   }

   public String nextString() {
      String result = pg.getString(pos);
      pos += STR_SIZE(result.length());
      return result;
   }
}
```

Figure 13-9
The code for the SimpleDB class *BasicLogRecord*

Tampon Yönetimi

- Kullanıcı veri sayfalarının **"tampon havuzunda"** tutulması ve bu havuzun koordinasyonu
- **tampon havuzu:** VTYS tarafından belirlenen sabit sayıda tampondan oluşan ana hafıza bölgesi. Her tampon bir sayfayi içermekte, her sayfa da bir disk bloğunun bilgisini tutmaktadır.(*Esasında, olay OS'nin tampon bölgesinden istenen alanın VTYS'nin kontrolüne verilmesi*)
- Protokol:
  - İstemci, istediği bloğa karşılık gelecek sayfanın tampon havuzunda **pin** edilmesini tampon yöneticisinden talep eder.
  - İstemci, bu sayfayı istediği gibi kullanır.(okuma/yazma)
  - İstemci, bu sayfa ile işini bitirince, tampon yöneticisinden sayfanın **unpin** edilmesini talep ister.
- **Pin edilmiş tampon**, bir istemci tarafından kullanılıyor anlamına geliyor. O zaman tampon yöneticisi bu tamponu başka bir bloğa atayamaz..

  Unpin edilmiş tampon ise, herhangi bir istemci tarafından kullanılmıyor anlamında. O zaman tampon yöneticisi bu tamponu başka bir bloğa atayabilir...
- O zaman, İstemcinin tampon talebi, şu 4 durumdan biri ile neticelenir:
  1. İstenen bloğun içeriği zaten tampon havuzunda olması ve
     a) Pin edilmiş
     b) Unpin edilmiş
  2. İstenen bloğun içeriği tampon havuzunda değil,
     a) En az bir unpin edilmiş tampon var.
     b) Bütün tamponlar pin edilmiş
- Her durumu(1a,1b,2a,2b) dikkatle analiz edelim!

Tampon yerdeğişim stratejileri

- Bir tampon isteği geldiğinde, Eğer tampon havuzunda birden çok **unpin** durumda olan tampon varsa(2a), yerdeğişim için bunlardan hangisi seçilecek?
  - Amaç: "En az disk erişim sayısı.." En iyi seçim kesin olarak bilinemez, o yüzden **unpin** durumdaki tamponlar arasında başarılı bir tahmin yapılmalı..*(işte tampon yöneticisinin **pin/unpin** durumunu takip etmesi, bu noktada kötü bir seçim yapılmasına engel oluyor. Yoksa OS'na bıraksak pin durumunda olabilen bir sayfayı seçebilirdi)*
- Çok sayıda tampon yerdeğişim stratejisinden sadece 4 tanesi: **Sade (naive), FIFO, LRU, Clock**
- Aşağıdaki 4 tamponluk havuz için verilen senaryoda, hangi blokların hangi zamanda **pin/unpin** edildiği şematize ediliyor. Şekilde sadece **pin(50),** bir yerdeğişime sebep olmakta bu değişimde 4 farklı strateji için de aynıdır;çünkü sadece 1 adet **unpin** durumunda tampon vardı *(1 no'lu tampon)*

```
pin(10); pin(20); pin(30); pin(40); unpin(20);
pin(50); unpin(40); unpin(10); unpin(30); unpin(50);
```

(a) A sequence of ten *pin/unpin* operations

Buffer:	0	1	2	3
block#	10	50	30	40
time read in	1	6	3	4
time unpinned	8	10	9	7

(b) The resulting state of the buffer pool

Figure 13-11
The effect of some *pin/unpin* operations on a pool of 4 buffers

11.zamanda pin(60)

12.zamanda pin(70)

İstekleri her bir stateji için nasıl bir yerdeğişime sebep verir?

- *Sade : 60 → tampon 0*
  70 → tampon 1
- *FIFO: 60 → tampon 0*
  70 → tampon 2
- *LRU: 60 → tampon 3*
  70 → tampon 0
- *Clock: 60 → tampon 2*

• Pin(60),unpin(60),pin(70),unpin(70),.....senaryosu için SADE stratejisi iyi midir?

• Katalog bilgisi tutan tamponlar için FIFO kullanımı iyi midir?

• LRU adil bir strateji mi? Clock, maksimum adil kullanım sağlıyor mu?

CLOCK stratejisi

- SADE, LRU ve LFU (*least frequently used*) yöntemlerinin iyi taraflarını alalım..
- Hangi tampon seçiliyor?
  1. Saat gibi düşünülen tampon havuzunda dön, karşılaştığın ilk unpin tamponu seç.
  2. Dönmeye en son yerdeğişimin olduğu tampondan sonraki tampondan başla.
- 1. kriter ile SADE'ye benziyor. Adil bir yaklaşım..
- 2. kriter ile LRU'ya benziyor. Çünkü en son yerdeğişen tampon, yeni bir yerdeğişime aday olması için bir tur dönmesi lazım.
- 2. kriter ile LFU'a benziyor. Çünkü, en son yerdeğişen tampona tekrar sıra geldiği zaman; eğer bu tampon çok sık kullnılan bir tampon ise yine **pin** durumunu muhafa ediyor olacaktır. Böylece seçilmesi olasılığı düşmüş oluyor.

SimpleDB'de Tampon Yönetimi

BufferMgr

```
public BufferMgr(int numbuffs);
public Buffer pin(Block blk);
public Buffer pinNew(String filename,
                     PageFormatter fmtr);
public void   unpin(Buffer buff);
public void   flushAll(int txnum);
public int    available();
```

Buffer

```
public int     getInt(int offset);
public String  getString(int offset);
public void    setInt(int offset, int val,
                      int txnum, int lsn);
public void    setString(int offset, String val,
                         int txnum, int lsn);
public Block   block();
```

PageFormatter

```
public void   format(Page p);
```

- Bir SimpleDB örneği(instance), sadece bir adet BufferMgr nesnesine

```
SimpleDB.initFileLogAndBufferMgr("studentdb");
BufferMgr bm = SimpleDB.bufferMgr();

Block blk = new Block("student.tbl", 0);
Buffer buff = bm.pin(blk);
String sname = buff.getString(46);
int gradyr = buff.getInt(38);
System.out.println(sname + " has gradyear " + gradyr);
```

```
SimpleDB.initFileLogAndBufferMgr("studentdb");
BufferMgr bm = SimpleDB.bufferMgr();
LogMgr lm = SimpleDB.logMgr();
int mytxnum = 1; // assume we are transaction 1

Block blk = new Block("student.tbl", 0);
Buffer buff = bm.pin(blk);
int gradyr = buff.getInt(38);
Object[] logrec  = new Object[]
                {mytxnum, "student.tbl", 0, 38, gradyr};
int lsn = lm.append(logrec);
buff.setInt(38, gradyr+1, mytxnum, lsn);
bm.unpin(buff);
```

Önce log kaydının yerini tespit etmem gerek...

Sayfa(*Page*) Formatları

- simpleDB (veya bir VTYS) farklı tipte sayfalara sahip olabilir: Veri sayfaları, index sayfaları, log sayfaları,..
- *PageFormatter* nesnesindeki *format(Page p); pinNew(...)* ile dosyaya yeni eklenecek olan sayfanın istenilen formatta olmasını sağlar.

```
class ABCStringFormatter implements PageFormatter {
   public void format(Page p) {
      int recsize = STR_SIZE("abc");
      for (int i=0; i+recsize <= BLOCK_SIZE; i+=recsize)
         p.setString(i, "abc");
   }
}
```

Figure 13-15
A simple (but utterly useless) page for

```
SimpleDB.initFileLogAndBufferMgr("studentdb");
BufferMgr bm = SimpleDB.bufferMgr();
PageFormatter pf = new ABCStringFormatter();
Buffer buff = bm.pinNew("junk", pf);
String s  = buff.getString(0); //will be "abc"
int blknum = buff.block().number();
bm.unpin(buff);
System.out.println("The first value in block "
                   + blknum + " is " + s);
```

Figure 13-16
Adding a new block to the file *junk*

Tampon çekişmesi (*contention*)

- Bütün tamponların **pin** durumunda olması halinde, tampon yöneticisi, bir **pin** veya **pinNew** isteğine hemen cevap veremez. Bunun için hemen yerine getirlemeyen istekler bekleme listesinde (*waitList*) bekletilir. *(Kullanıcı bu seviyedeki bir bekletilmeden tabiki habersizdir.)*
- Tampon çekişmelerinde, kilitlenme (*deadlock*) senaryoları yaşanabilir. Örneğin, 2 tamponluk bir havuzdan, 2 istemcinin her birinden 2 tampon isteği olsun. Her bir istemci bir tamponu kaparsa, ikinci tampon için her ikisi de bekleme listesine alınacak=>kilitlenme!
- Kilitlenmeye mani olan veya çözen algoritmalar karmaşık bir konu. SimpleDB'de bu problem, önceden tayin edilen "istemci bekleme süresinin" aşılmasında, istemciye Exception (*BufferAbortException*) gönderilmesi ile çözülmüştür. Bu durumda istemcinin tx'ı, *rollback* ile yeniden başlatılır.

SimpleDB'de Tampon

- Tampon sınıf şunları takip eder,
  - içerdiği sayfa,
  - karşılık gelen disk bloğu
  - pin durumu (üstünde kaç pin var)
  - Sayfayı en son değiştiren tx.
  - Sayfadaki en son değişikliğe ait LSN
- Bununla beraber değişmiş sayfaların diske yazılmasını sağlamalıdır. Bunun ne zaman yapar?
  - Değişikli olduğu anda?
  - Tampon unpin olduğu zaman?(1b)
  - Tampon başka bir bloğa atandığı zaman?(2a)
  - Kurtarma modülü istediği zaman

```
public class Buffer {
   private Page contents = new Page();
   private Block blk = null;
   private int pins = 0;
   private int modifiedBy = -1;
   private int logSequenceNumber;

   public int getInt(int offset) {
      return contents.getInt(offset);
   }

   public String getString(int offset) {
      return contents.getString(offset);
   }

   public void setInt(int offset, int val,
                      int txnum, int lsn) {
      modifiedBy = txnum;
      if (lsn >= 0)
         logSequenceNumber = lsn;
      contents.setInt(offset, val);
   }

   public void setString(int offset, String val,
                         int txnum, int lsn) {
      modifiedBy = txnum;
      if (lsn >= 0)
         logSequenceNumber = lsn;
      contents.setString(offset, val);
   }
```

Figure 13-17
The code for the SimpleDB class *Buffer*

```
   public Block block() {
      return blk;
   }

   void flush() {
      if (modifiedBy >= 0) {
         SimpleDB.logMgr().flush(logSequenceNumber);
         contents.write(blk);
      }
      modifiedBy = -1;
   }

   void pin() {
      pins++;
   }

   void unpin() {
      pins--;
   }

   boolean isPinned() {
      return pins > 0;
   }

   boolean isModifiedBy(int txnum) {
      return txnum == modifiedBy;
   }

   void assignToBlock(Block b) {
      flush();
      blk = b;
      contents.read(blk);
      pins = 0;
   }

   void assignToNew(String filename, PageFormatter fmtr) {
      flush();
      fmtr.format(contents);
      blk = contents.append(filename);
      pins = 0;
   }
}
```

Figure 13-17 *(Continued)*

SimpleDB'de Tampon yönetimi *(basit)*

```
class BasicBufferMgr {
   private Buffer[] bufferpool;
   private int numAvailable;

   BasicBufferMgr(int numbuffs) {
      bufferpool = new Buffer[numbuffs];
      numAvailable = numbuffs;
      for (int i=0; i<numbuffs; i++)
         bufferpool[i] = new Buffer();
   }

   synchronized void flushAll(int txnum) {
      for (Buffer buff : bufferpool)
         if (buff.isModifiedBy(txnum))
            buff.flush();
   }

   synchronized Buffer pin(Block blk) {
      Buffer buff = findExistingBuffer(blk);
      if (buff == null) {
         buff = chooseUnpinnedBuffer();
         if (buff == null)
            return null;
         buff.assignToBlock(blk);
      }
      if (!buff.isPinned())
         numAvailable--;
      buff.pin();
      return buff;
   }

   synchronized Buffer pinNew(String filename,
                              PageFormatter fmtr) {
      Buffer buff = chooseUnpinnedBuffer();
      if (buff == null)
         return null;
      buff.assignToNew(filename, fmtr);
      numAvailable--;
      buff.pin();
      return buff;
   }

   synchronized void unpin(Buffer buff) {
      buff.unpin();
      if (!buff.isPinned())
         numAvailable++;
   }

   int available() {
      return numAvailable;
   }
   private Buffer findExistingBuffer(Block blk) {
      for (Buffer buff : bufferpool) {
         Block b = buff.block();
         if (b != null && b.equals(blk))
            return buff;
      }
      return null;
   }

   private Buffer chooseUnpinnedBuffer() {
      for (Buffer buff : bufferpool)
         if (!buff.isPinned())
            return buff;
      return null;
   }
}
```

Sade yerdeğiştirme staratejisi

Figure 13-18
The code for the SimpleDB class *BasicBufferMgr*

SimpleDB'de Tampon yönetimi *(bekleme listesi)*

```
public class BufferMgr {
   private static final long MAX_TIME = 10000;
   private BasicBufferMgr bufferMgr;

   public BufferMgr(int numbuffers) {
      bufferMgr = new BasicBufferMgr(numbuffers);
   }

   public synchronized Buffer pin(Block blk) {
      try {
         long timestamp = System.currentTimeMillis();
         Buffer buff = bufferMgr.pin(blk);
         while (buff == null &&
               !waitingTooLong(timestamp)) {
            wait(MAX_TIME);
            buff = bufferMgr.pin(blk);
         }
         if (buff == null)
            throw new BufferAbortException();
         return buff;
      }
      catch(InterruptedException e) {
         throw new BufferAbortException();
      }
   }

   public synchronized Buffer pinNew(String filename,
                                     PageFormatter fmtr) {
      try {
         long timestamp = System.currentTimeMillis();
         Buffer buff = bufferMgr.pinNew(filename, fmtr);
         while (buff == null &&
               !waitingTooLong(timestamp)) {
            wait(MAX_TIME);
            buff = bufferMgr.pinNew(filename, fmtr);
         }
         if (buff == null)
            throw new BufferAbortException();
         return buff;
      }
      catch(InterruptedException e) {
         throw new BufferAbortException();
      }
   }

   public synchronized void unpin(Buffer buff) {
      bufferMgr.unpin(buff);
      if (!buff.isPinned())
         notifyAll();
   }

   public void flushAll(int txnum) {
      bufferMgr.flushAll(txnum);
   }

   public int available() {
      return bufferMgr.available();
   }

   private boolean waitingTooLong(long starttime) {
      long now = System.currentTimeMillis();
      return now - starttime > MAX_TIME;
   }
}
```

Figure 13-19
The code for the SimpleDB class *BufferMgr*

Figure 13-19 *(Continued)*