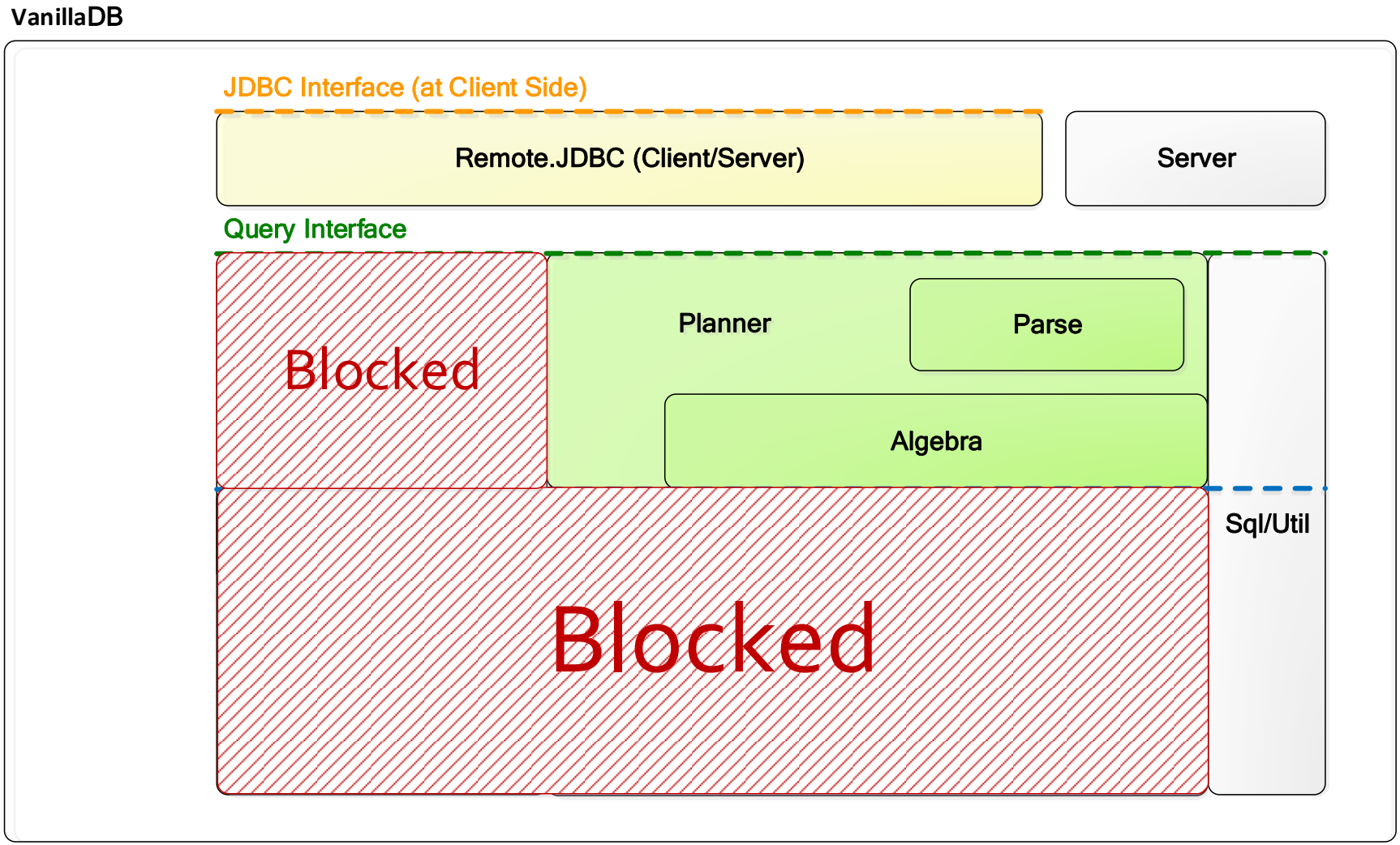
VanillaCore Walkthrough Part 3

Cloud Databases

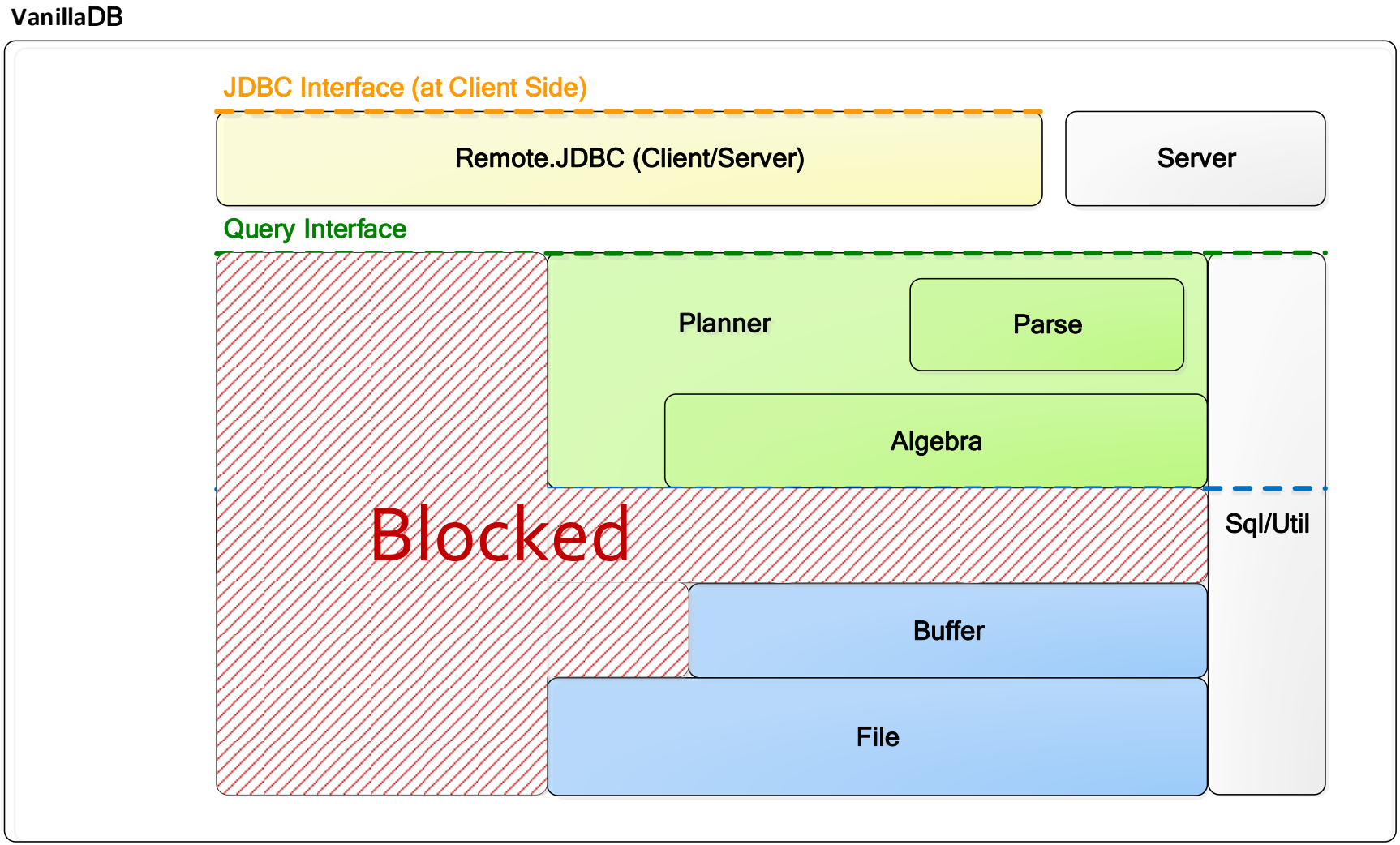
DataLab

CS, NTHU

Last Time


VanillaDB
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Blocked
Planner
Parse
Algebra
Sql/Util
Blocked

This Time


VanillaDB
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Planner
Parse
Algebra
Blocked
Sql/Util
Buffer
File

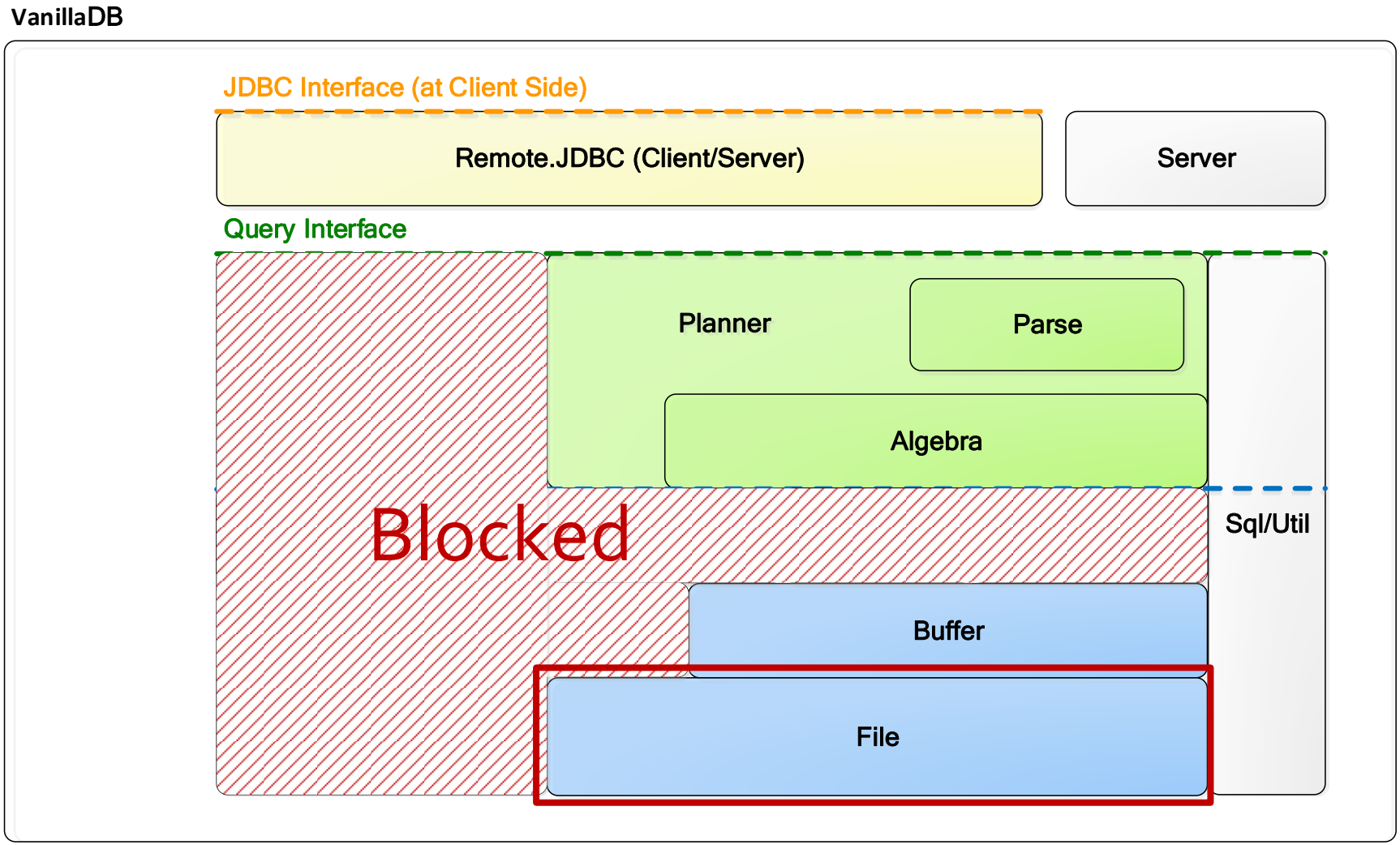
Outline

- File package
- Buffer package

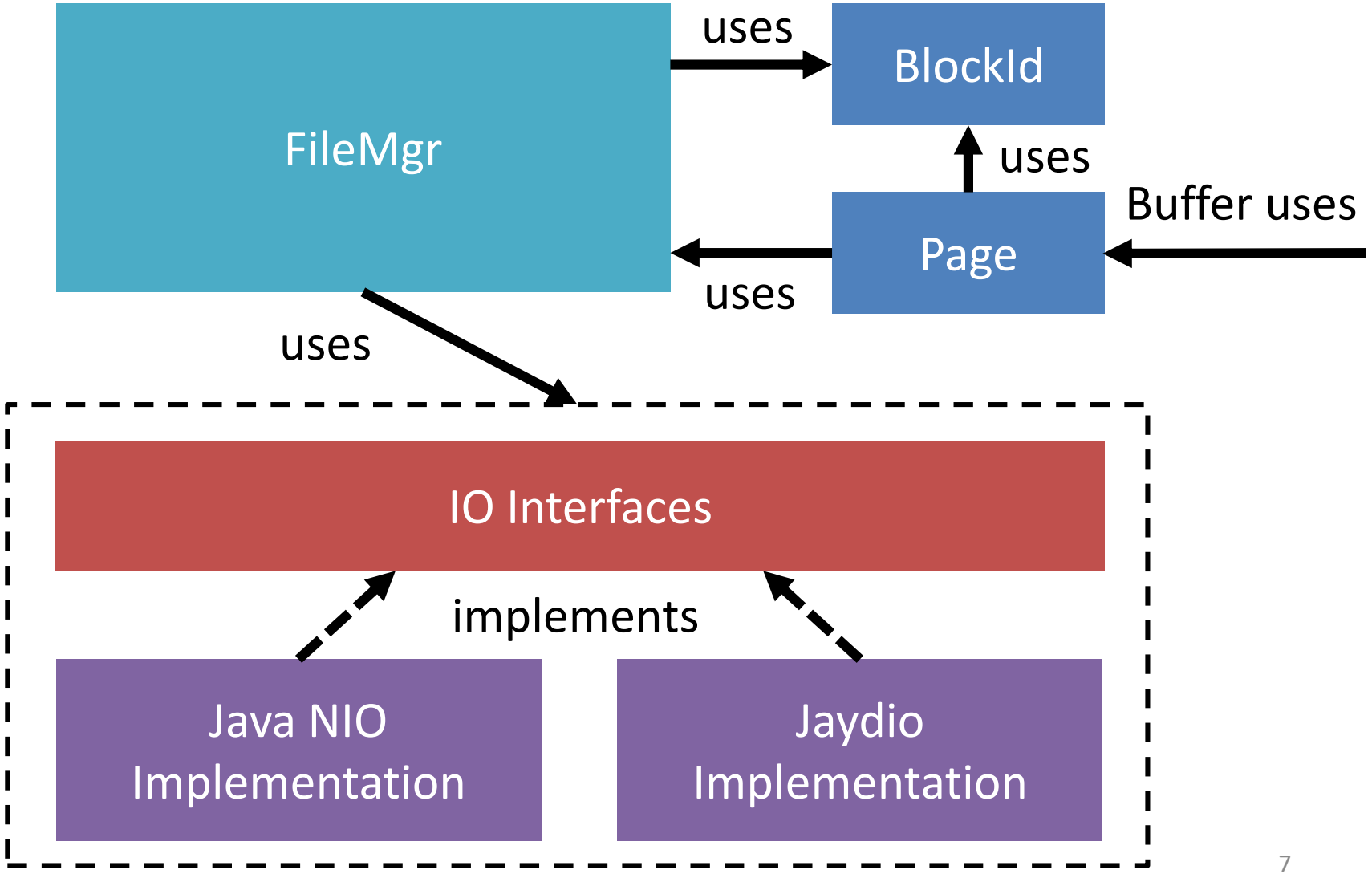
Outline

- File package
- Buffer package

Where are we?

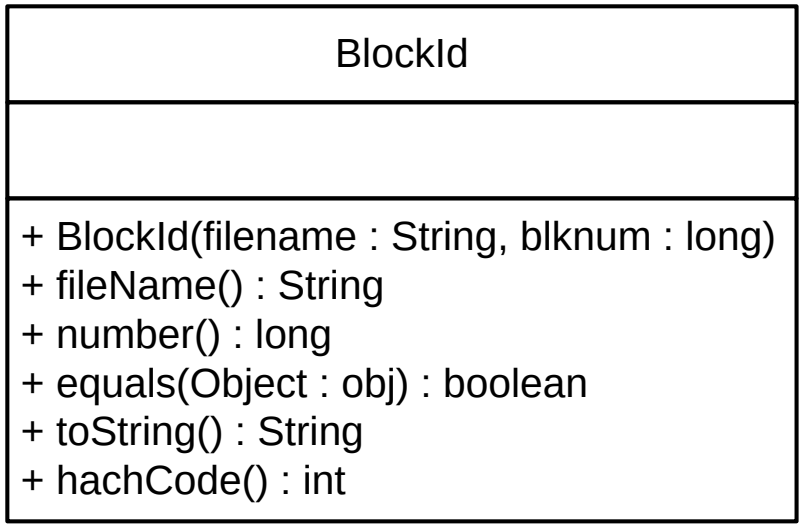

VanillaDB
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Planner
Parse
Algebra
Blocked
Sql/Util
Buffer
File

`file` Package

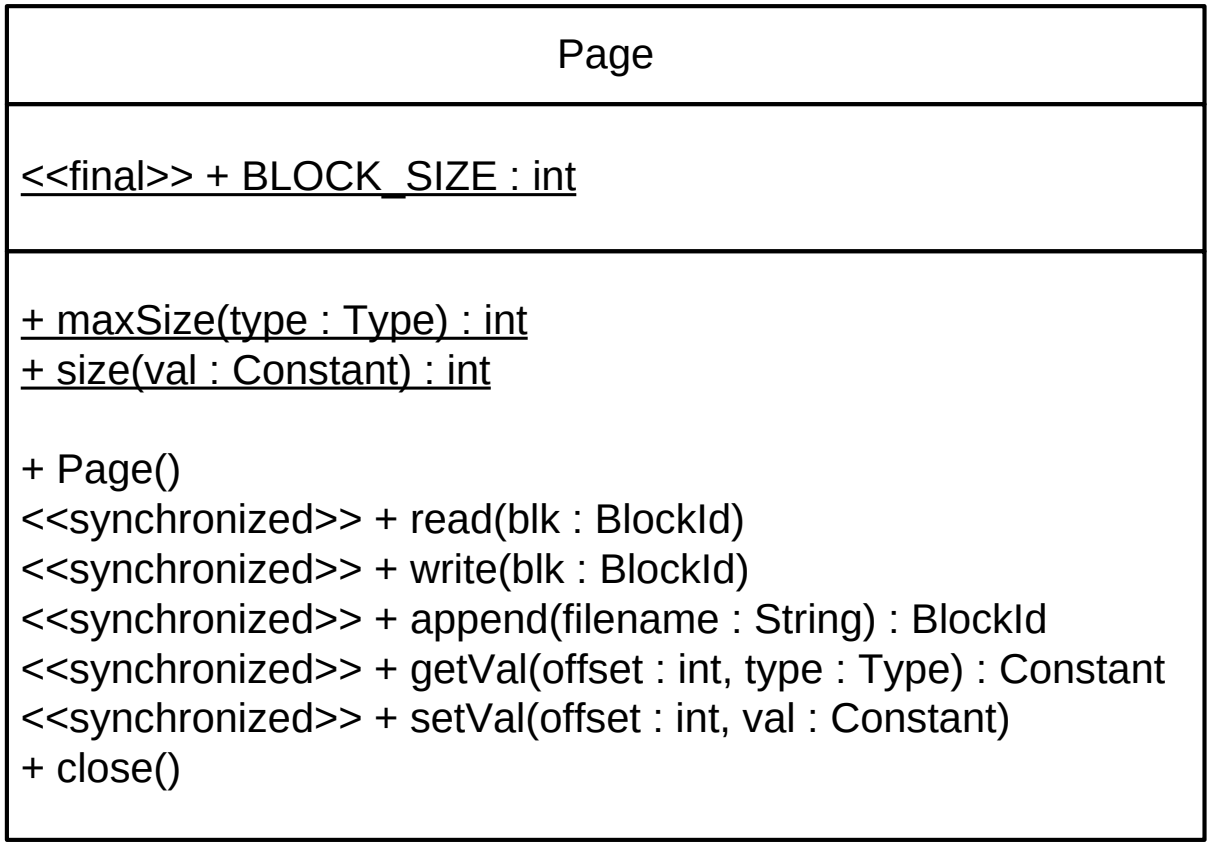


BlockId

```
public class BlockId {
    private String fileName;
    private long blkNum;

    public BlockId(String fileName, long blkNum) {
        this.fileName = fileName;
        this.blkNum = blkNum;
    }

    public String fileName() {
        return fileName;
    }

    public long number() {
        return blkNum;
    }
    ...
}
```



Page


Page
<<final>> + BLOCK_SIZE : int
+ maxSize(type : Type) : int
+ size(val : Constant) : int
+ Page()
<<synchronized>> + read(blk : BlockId)
<<synchronized>> + write(blk : BlockId)
<<synchronized>> + append(filename : String) : BlockId
<<synchronized>> + getVal(offset : int, type : Type) : Constant
<<synchronized>> + setVal(offset : int, val : Constant)
+ close()

Page

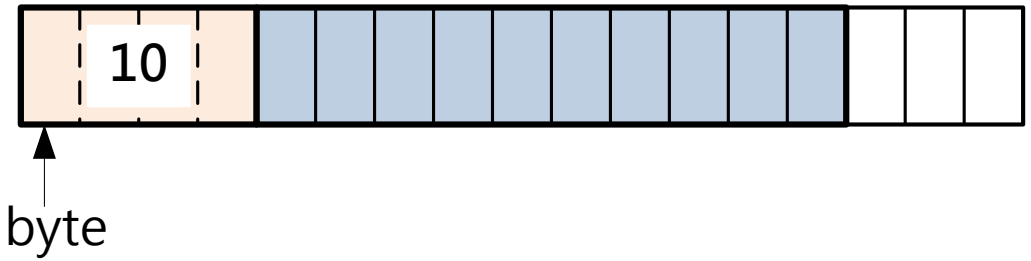
- Backed by `IoBuffer`

```
private IoBuffer contents = IoAllocator.newIoBuffer(BLOCK_SIZE);
```

- Translate constants using `Constant.asBytes()`
  - Fixed length for numeric type constants (e.g., 4 bytes for `IntegerConstant`)
  - Variable length for `VarcharConstant`
- How to reconstruct a varchar constant in getter?

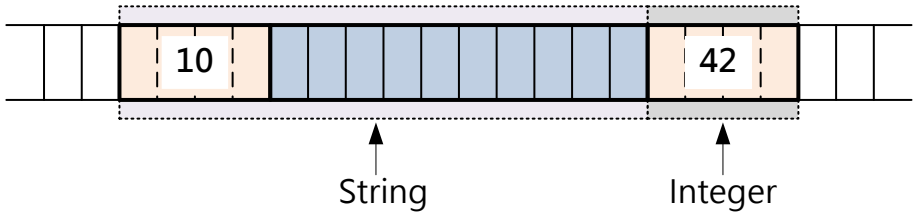
Storing A Varchar

- `Page` stores a Varchar in two parts
  - The first is the length of those bytes
  - The second is the bytes from `asByte()`



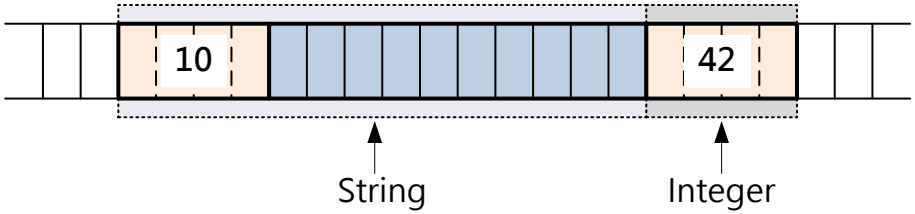
setVal

```
public synchronized void setVal(int offset, Constant val) {
    byte[] byteval = val.asBytes();

    // Append the size of value if it is not fixed size
    if (!val.getType().isFixedSize()) {
        // check the field capacity and value size
        if (offset + ByteHelper.INT_SIZE + byteval.length > BLOCK_SIZE)
            throw new BufferOverflowException();

        byte[] sizeBytes = ByteHelper.toBytes(byteval.length);
        contents.put(offset, sizeBytes);
        offset += sizeBytes.length;
    }

    // Put bytes
    contents.put(offset, byteval);
}
```



getVal

```
public synchronized Constant getVal(int offset, Type type) {
    int size;
    byte[] byteVal = null;

    // Check the length of bytes
    if (type.isFixedSize()) {
        size = type.maxSize();
    } else {
        byteVal = new byte[ByteHelper.INT_SIZE];
        contents.get(offset, byteVal);
        size = ByteHelper.toInteger(byteVal);
        offset += ByteHelper.INT_SIZE;
    }

    // Get bytes and translate it to Constant
    byteVal = new byte[size];
    contents.get(offset, byteVal);
    return Constant.newInstance(type, byteVal);
}
```



Sizing Information

- There are static APIs providing sizing information in Page

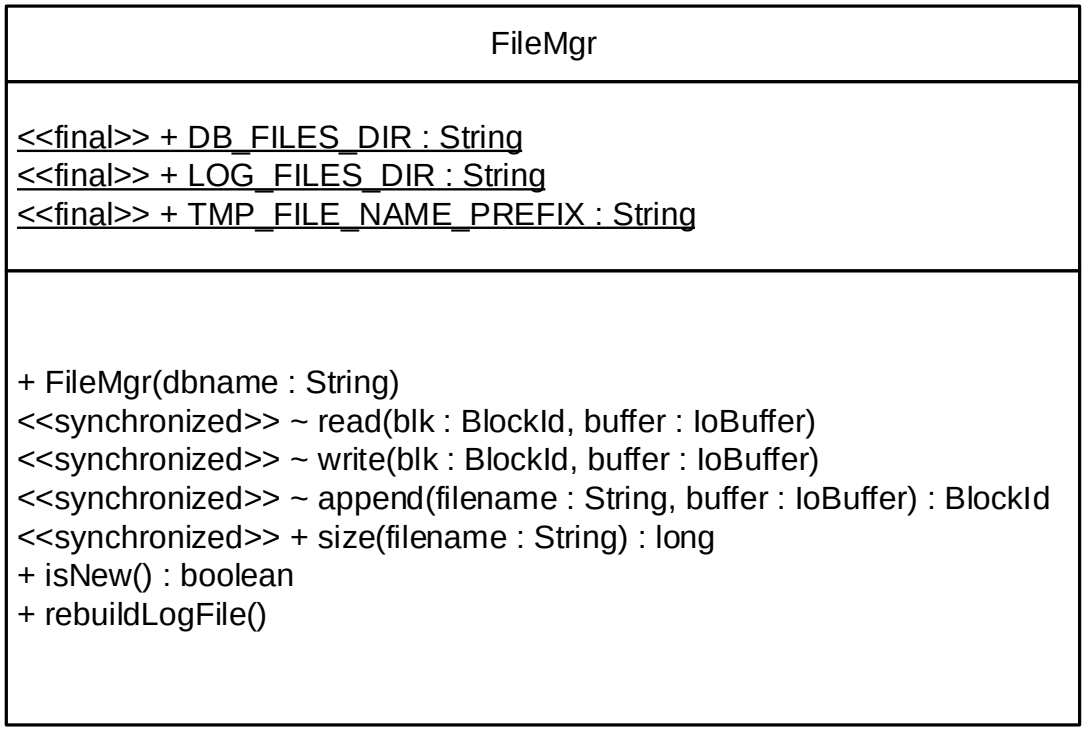
```
public static int maxSize(Type type) {
     return type.isFixedSize() ? type.maxSize() : ByteHelper.INT_SIZE
               + type.maxSize();
}

public static int size(Constant val) {
     return val.getType().isFixedSize() ? val.size() : ByteHelper.INT_SIZE
               + val.size();
}
```

File I/Os

```
public Page() {
}

public synchronized void read(BlockId blk) {
    fileMgr.read(blk, contents);
}

public synchronized void write(BlockId blk) {
    fileMgr.write(blk, contents);
}

public synchronized BlockId append(String fileName) {
    return fileMgr.append(fileName, contents);
}
```

FileMgr

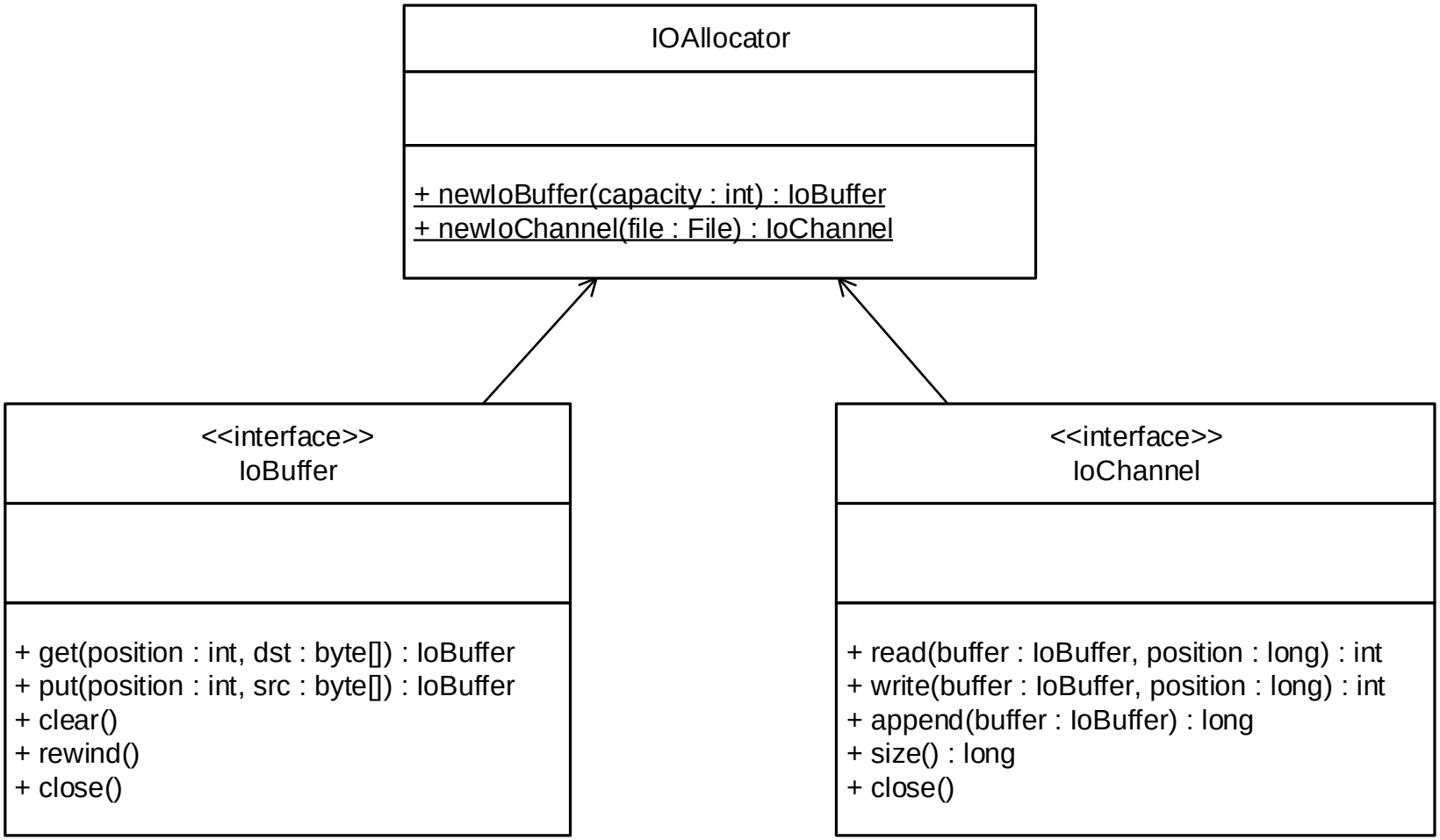
- Handles the actual I/Os
- Keeps the `IoChannel` instances of all opened files



`FileMgr`

- A page delegates `read`, `write` and, `append` to `FileMgr`
- Note that the file manager always reads/writes/appends a ***block-sized*** number of bytes from/to a file
  - Exactly one disk access per call

file.io


IOAllocator
+ newIoBuffer(capacity : int) : IoBuffer
+ newIoChannel(file : File) : IoChannel
<<interface>>
IoBuffer
+ get(position : int, dst : byte[]) : IoBuffer
+ put(position : int, src : byte[]) : IoBuffer
+ clear()
+ rewind()
+ close()
<<interface>>
IoChannel
+ read(buffer : IoBuffer, position : long) : int
+ write(buffer : IoBuffer, position : long) : int
+ append(buffer : IoBuffer) : long
+ size() : long
+ close()

`IoChannel` in Java NIO

- Opens a file by creating a new `RandomAccessFile` instance and then obtain its file channel via `getChannel()`
- Files are open in "rws" mode when using Java NIO
  - The "rw" means that the file is open for reading an writing
  - The "s" means that the OS should not delay disk I/O in order to optimize disk performance; instead, every *write* operation must be written immediately to the disk

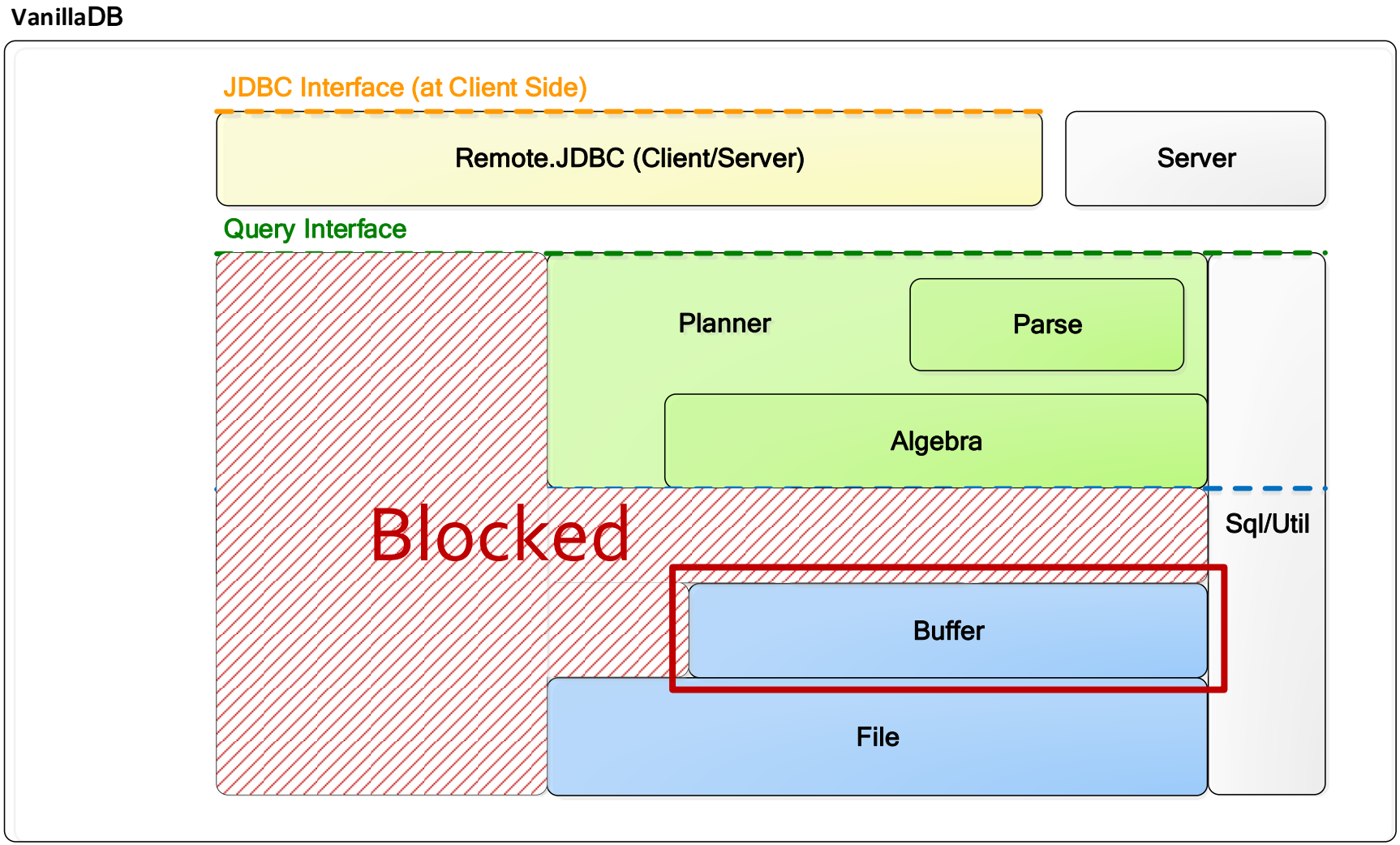
`IoBuffer` in Java NIO

- We don't want the memory space of `ByteBuffer` be swapped out by OS
- `ByteBuffer` has two factory methods: `allocate` and `allocateDirect`
  - `allocateDirect` tells JVM to use one of the OS's I/O buffers to hold the bytes
  - ***Not*** in Java programmable buffer, no garbage collection
  - Eliminates the redundancy of ***double buffering***

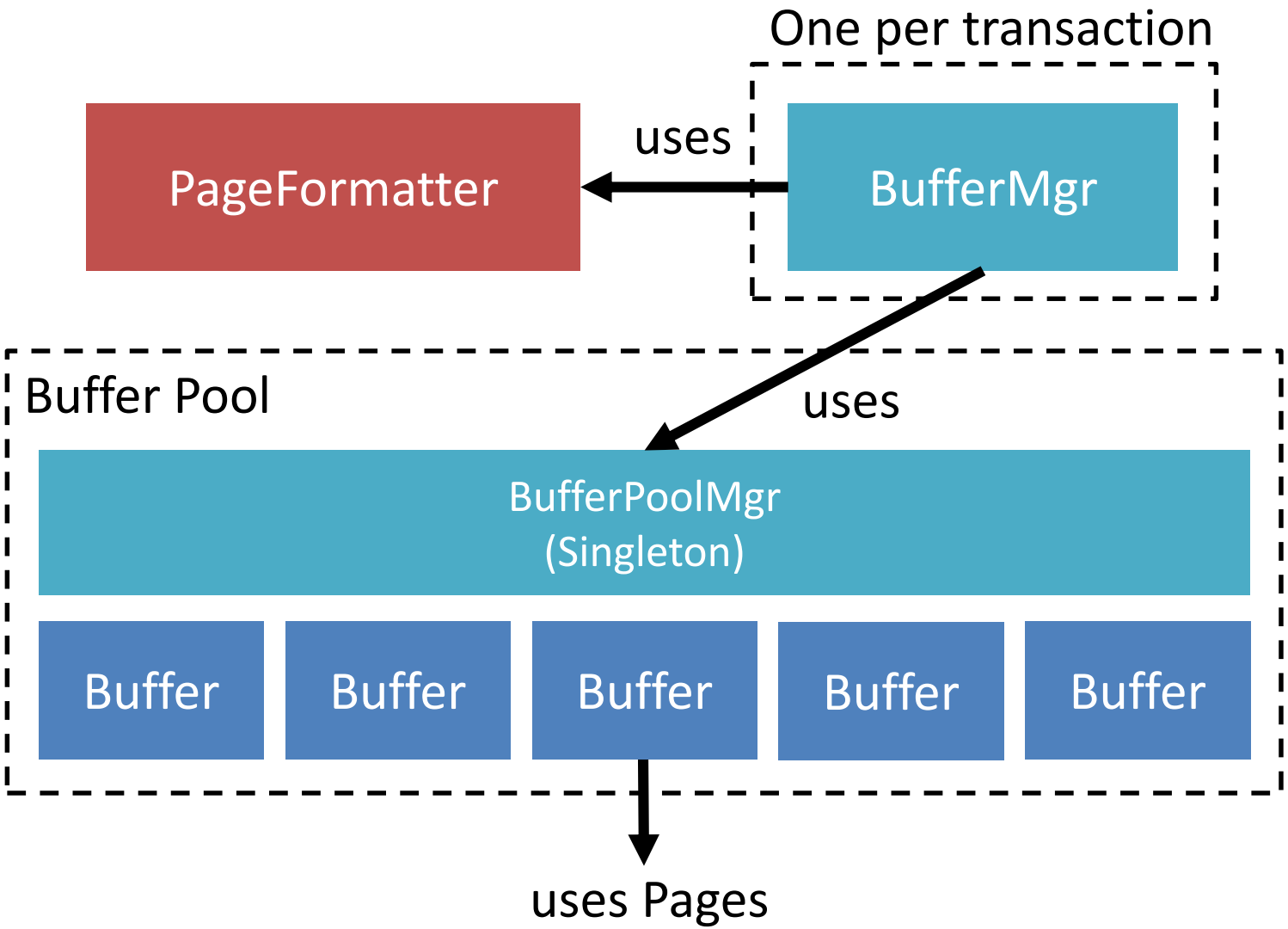
Outline

- File package
- Buffer package

Where are we?


VanillaDB
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Planner
Parse
Algebra
Sql/Util
Blocked
Buffer
File

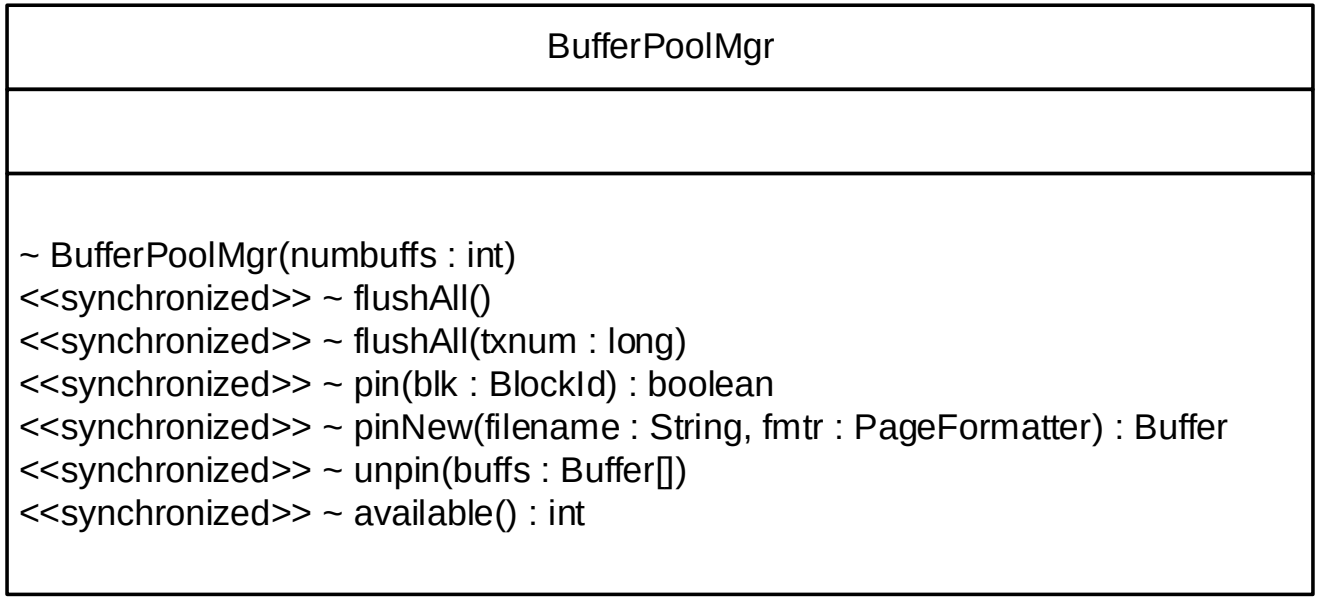
`buffer` Package



`BufferMgr` vs. `BufferPoolMgr`

- Each transaction has its own `BufferMgr`, but there is only one `BufferPoolMgr`
- Responsibility
  - `BufferPoolMgr` manages the buffer pool
  - `BufferMgr` handles waiting for pinning and manages pinned buffers for each transaction

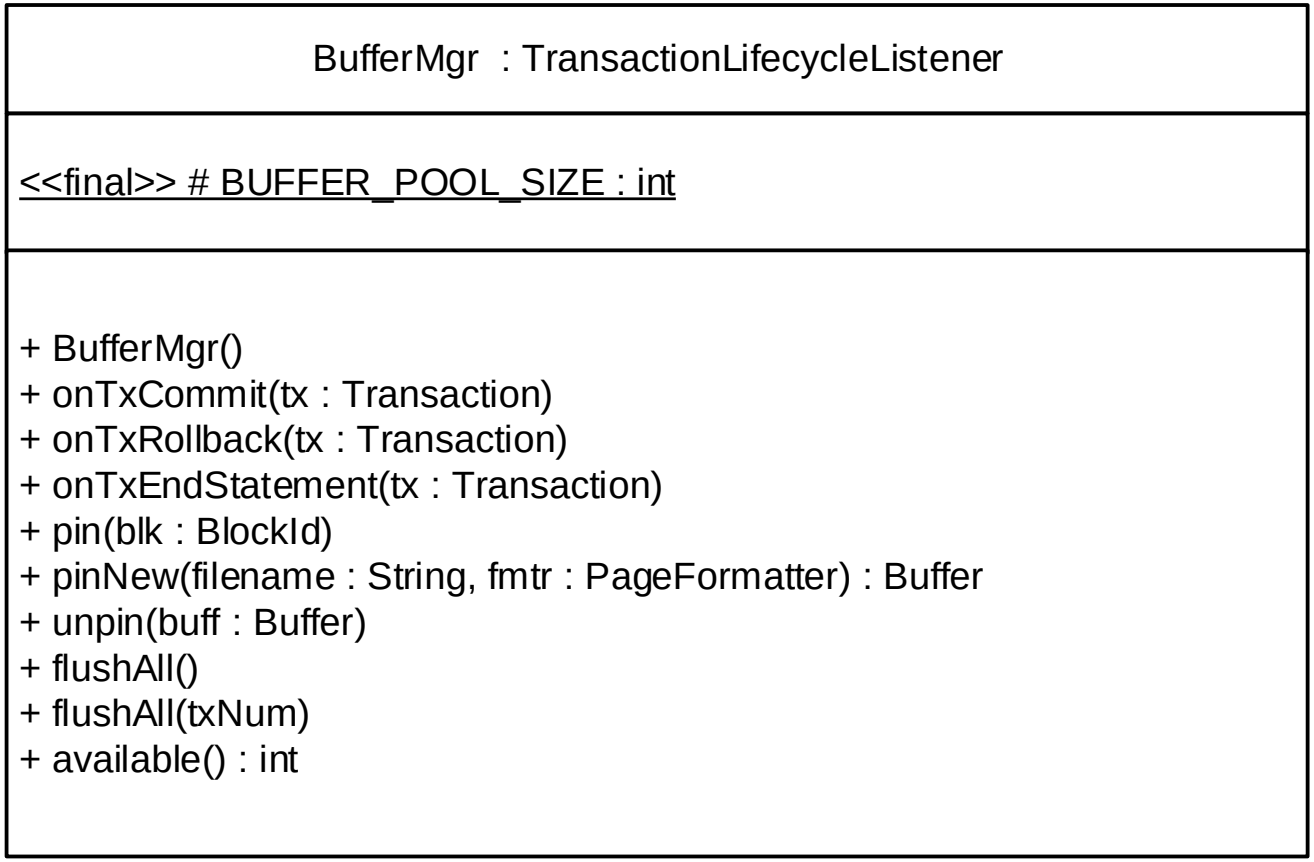
BufferPoolMgr



`BufferPoolMgr`

- Singleton
- Finds a hit for a `pin()`
- Implements the ***clock*** replacement strategy
- The `pin()` ***returns null immediately*** if there's no candidate buffer
  - Then, the `BufferMgr` make the calling thread waiting and retrying later

BufferMgr


BufferMgr : TransactionLifecycleListener
<<final>> # BUFFER_POOL_SIZE : int
+ BufferMgr()
+ onTxCommit(tx : Transaction)
+ onTxRollback(tx : Transaction)
+ onTxEndStatement(tx : Transaction)
+ pin(blk : BlockId)
+ pinNew(filename : String, fmtr : PageFormatter) : Buffer
+ unpin(buff : Buffer)
+ flushAll()
+ flushAll(txNum)
+ available() : int

`BufferMgr`

- Created when constructing a transaction
- A `BufferMgr` manages the pinned buffers and the pinning counts of a transaction
- `BufferMgr.pin()` makes the calling thread to wait if there's no candidate buffer for replacement
  - How?

Java `wait()` and `notifyAll()` Methods

- In Java, every object has a waiting list
  - `obj.wait(timeout)` puts the caller thread into the waiting list of `obj`
- The thread will be removed from the list and ready for execution in two conditions:
  - Another thread call `obj.notifyAll()`
  - Timeout elapsed

Java `wait()` and `notifyAll()` Methods

- If…
  1. `obj.wait()` is surrounded by a synchronized block, and
  2. there are multiple threads in `obj`'s waiting list,
- Then when `notifyAll()` is called, ***all*** waiting threads will compete on the lock to enter the synchronized block
  - ***No*** FIFO guarantee which thread will be notified first, and which will acquire the lock first
  - Only one thread wins the lock, others ***blocked*** until the winner releases the lock

BufferMgr

- `pin()`: if `BufferPoolMgr` returns null, put the current thread into `BufferPoolMgr`'s waiting list

```
buff = bufferPool.pin(blk);
while (buff == null && !waitingTooLong(timestamp)) {
    bufferPool.wait(MAX_TIME);
    buff = bufferPool.pin(blk);
}
```

- `unpin(buff)`: notify all threads in `BufferPoolMgr`'s waiting list
  - Only one thread will pin successfully due to the synchronization

```
public Buffer pin(BlockId blk) {
      synchronized (bufferPool) {
            PinnedBuffer pinnedBuff = pinnedBuffers.get(blk);
            if (pinnedBuff != null) {
                  pinnedBuff.pinnedCount++;
                  return pinnedBuff.buffer;
            }
            if (pinnedBuffers.size() == BUFFER_POOL_SIZE)
                  throw new BufferAbortException();
            try {
                  Buffer buff;
                  long timestamp = System.currentTimeMillis();
                  buff = bufferPool.pin(blk);
                  if (buff == null) {
                        waitingThreads.add(Thread.currentThread());
                        while (buff == null && !waitingTooLong(timestamp)) {
                              bufferPool.wait(MAX_TIME);
                              if (waitingThreads.get(0).equals(Thread.currentThread()))
                                    buff = bufferPool.pin(blk);
                        }
                        waitingThreads.remove(Thread.currentThread());
                        bufferPool.notifyAll();
                  }
                  if (buff == null) {
                        repin();
                        buff = pin(blk);
                  } else {
                        pinnedBuffers.put(buff.block(), new PinnedBuffer(buff));
                  }
                  return buff;
            } catch (InterruptedException e) {
                  throw new BufferAbortException();
            }
      }
}
```

```
public Buffer pin(BlockId blk) {
     synchronized (bufferPool) {
          PinnedBuffer pinnedBuff = pinnedBuffers.get(blk);
          if (pinnedBuff != null) {
               pinnedBuff.pinnedCount++;
               return pinnedBuff.buffer;
          }
          if (pinnedBuffers.size() == BUFFER_POOL_SIZE)
               throw new BufferAbortException();
          try {
               Buffer buff;
               long timestamp = System.currentTimeMillis();
               buff = bufferPool.pin(blk);
               if (buff == null) {
                    waitingThreads.add(Thread.currentThread());
                    while (buff == null && !waitingTooLong(timestamp)) {
                         bufferPool.wait(MAX_TIME);
                         if (waitingThreads.get(0).equals(Thread.currentThread()))
                              buff = bufferPool.pin(blk);
                    }
                    waitingThreads.remove(Thread.currentThread());
                    bufferPool.notifyAll();
               }
               if (buff == null) {
                    repin();
                    buff = pin(blk);
               } else {
                    pinnedBuffers.put(buff.block(), new PinnedBuffer(buff));
               }
               return buff;
          } catch (InterruptedException e) {
               throw new BufferAbortException();
          }
     }
}
```

Synchronize on the buffer pool (singleton)

```
public Buffer pin(BlockId blk) {
      synchronized (bufferPool) {
            PinnedBuffer pinnedBuff = pinnedBuffers.get(blk);
            if (pinnedBuff != null) {
                  pinnedBuff.pinnedCount++;
                  return pinnedBuff.buffer;
            }
            if (pinnedBuffers.size() == BUFFER_POOL_SIZE)
                  throw new BufferAbortException();
            try {
                  Buffer buff;
                  long timestamp = System.currentTimeMillis();
                  buff = bufferPool.pin(blk);
                  if (buff == null) {
                        waitingThreads.add(Thread.currentThread());
                        while (buff == null && !waitingTooLong(timestamp)) {
                              bufferPool.wait(MAX_TIME);
                              if (waitingThreads.get(0).equals(Thread.currentThread()))
                                    buff = bufferPool.pin(blk);
                        }
                        waitingThreads.remove(Thread.currentThread());
                        bufferPool.notifyAll();
                  }
                  if (buff == null) {
                        repin();
                        buff = pin(blk);
                  } else {
                        pinnedBuffers.put(buff.block(), new PinnedBuffer(buff));
                  }
                  return buff;
            } catch (InterruptedException e) {
                  throw new BufferAbortException();
            }
      }
}
```

Find the given block from the pinned buffers of this transaction

```
public Buffer pin(BlockId blk) {
      synchronized (bufferPool) {
            PinnedBuffer pinnedBuff = pinnedBuffers.get(blk);
            if (pinnedBuff != null) {
                  pinnedBuff.pinnedCount++;
                  return pinnedBuff.buffer;
            }
            if (pinnedBuffers.size() == BUFFER_POOL_SIZE)
                  throw new BufferAbortException();
            try {
                  Buffer buff;
                  long timestamp = System.currentTimeMillis();
                  buff = bufferPool.pin(blk);
                  if (buff == null) {
                        waitingThreads.add(Thread.currentThread());
                        while (buff == null && !waitingTooLong(timestamp)) {
                              bufferPool.wait(MAX_TIME);
                              if (waitingThreads.get(0).equals(Thread.currentThread()))
                                    buff = bufferPool.pin(blk);
                        }
                        waitingThreads.remove(Thread.currentThread());
                        bufferPool.notifyAll();
                  }
                  if (buff == null) {
                        repin();
                        buff = pin(blk);
                  } else {
                        pinnedBuffers.put(buff.block(), new PinnedBuffer(buff));
                  }
                  return buff;
            } catch (InterruptedException e) {
                  throw new BufferAbortException();
            }
      }
}
```

Pins the requested block

Add the buffer to the pinned list of this transaction

```
public Buffer pin(BlockId blk) {
      synchronized (bufferPool) {
            PinnedBuffer pinnedBuff = pinnedBuffers.get(blk);
            if (pinnedBuff != null) {
                  pinnedBuff.pinnedCount++;
                  return pinnedBuff.buffer;
            }
            if (pinnedBuffers.size() == BUFFER_POOL_SIZE)
                  throw new BufferAbortException();
            try {
                  Buffer buff;
                  long timestamp = System.currentTimeMillis();
                  buff = bufferPool.pin(blk);
                  if (buff == null) {
                        waitingThreads.add(Thread.currentThread());
                        while (buff == null && !waitingTooLong(timestamp)) {
                              bufferPool.wait(MAX_TIME);
                              if (waitingThreads.get(0).equals(Thread.currentThread()))
                                    buff = bufferPool.pin(blk);
                        }
                        waitingThreads.remove(Thread.currentThread());
                        bufferPool.notifyAll();
                  }
                  if (buff == null) {
                        repin();
                        buff = pin(blk);
                  } else {
                        pinnedBuffers.put(buff.block(), new PinnedBuffer(buff));
                  }
                  return buff;
            } catch (InterruptedException e) {
                  throw new BufferAbortException();
            }
      }
}
```

If there was not any available buffer, make the thread waiting

The thread in the head of the list can pin

Wake up other thread again

```
public Buffer pin(BlockId blk) {
      synchronized (bufferPool) {
            PinnedBuffer pinnedBuff = pinnedBuffers.get(blk);
            if (pinnedBuff != null) {
                  pinnedBuff.pinnedCount++;
                  return pinnedBuff.buffer;
            }
            if (pinnedBuffers.size() == BUFFER_POOL_SIZE)
                  throw new BufferAbortException();
            try {
                  Buffer buff;
                  long timestamp = System.currentTimeMillis();
                  buff = bufferPool.pin(blk);
                  if (buff == null) {
                        waitingThreads.add(Thread.currentThread());
                        while (buff == null && !waitingTooLong(timestamp)) {
                              bufferPool.wait(MAX_TIME);
                              if (waitingThreads.get(0).equals(Thread.currentThread()))
                                    buff = bufferPool.pin(blk);
                        }
                        waitingThreads.remove(Thread.currentThread());
                        bufferPool.notifyAll();
                  }
                  if (buff == null) {
                        repin();
                        buff = pin(blk);
                  } else {
                        pinnedBuffers.put(buff.block(), new PinnedBuffer(buff));
                  }
                  return buff;
            } catch (InterruptedException e) {
                  throw new BufferAbortException();
            }
      }
}
```

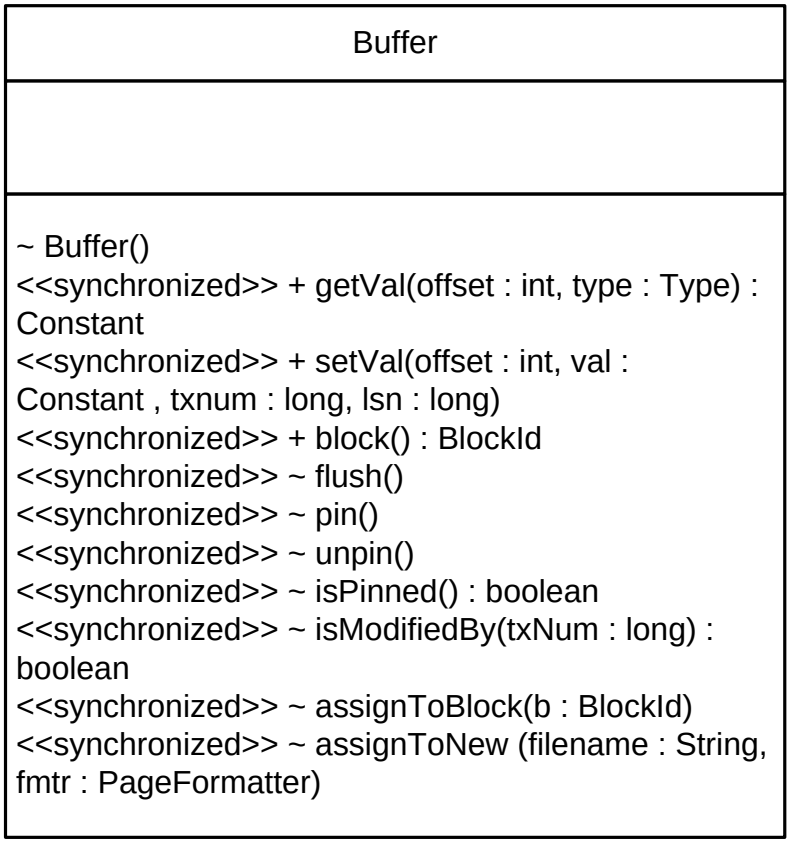
Waitting too long? There might be deadlock. Re-pin all blocks

```
public Buffer pin(BlockId blk) {
      synchronized (bufferPool) {
            PinnedBuffer pinnedBuff = pinnedBuffers.get(blk);
            if (pinnedBuff != null) {
                  pinnedBuff.pinnedCount++;
                  return pinnedBuff.buffer;
            }
            if (pinnedBuffers.size() == BUFFER_POOL_SIZE)
                  throw new BufferAbortException();
            try {
                  Buffer buff;
                  long timestamp = System.currentTimeMillis();
                  buff = bufferPool.pin(blk);
                  if (buff == null) {
                        waitingThreads.add(Thread.currentThread());
                        while (buff == null && !waitingTooLong(timestamp)) {
                              bufferPool.wait(MAX_TIME);
                              if (waitingThreads.get(0).equals(Thread.currentThread()))
                                    buff = bufferPool.pin(blk);
                        }
                        waitingThreads.remove(Thread.currentThread());
                        bufferPool.notifyAll();
                  }
                  if (buff == null) {
                        repin();
                        buff = pin(blk);
                  } else {
                        pinnedBuffers.put(buff.block(), new PinnedBuffer(buff));
                  }
                  return buff;
            } catch (InterruptedException e) {
                  throw new BufferAbortException();
            }
      }
}
```

Self-deadlock: throw exception

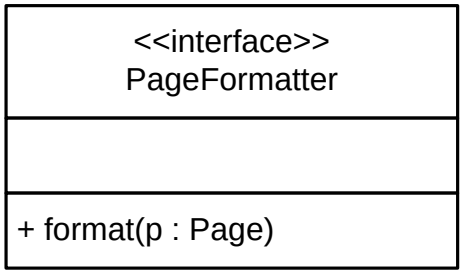
Buffer

- Wraps a page and stores
  - ID of the holding block
  - Pin count
  - Modified information
  - Log information
- ***Supports WAL***
  - `setVal()` requires an LSN
    - Must be preceded by `LogMgr.append()`
  - `flush()` calls `LogMgr.flush(maxLsn)`
    - Called by `BufferMgr` upon swapping



PageFormatter

- The `pinNew(fmtr)` method of `BufferMgr` appends a new block to a file
- `PageFormatter` initializes the block
  - To be extended in packages (`storage.record` and `storage.index.btree`) where the semantics of records are defined




```
class ZeroIntFormatter implements PageFormatter {
    public void format(Page p) {
        Constant zero = new IntegerConstant(0);
        int recsize = Page.size(zero);
        for (int i = 0; i + recsize <= Page.BLOCK_SIZE; i += recsize)
            p.setVal(i, zero);
    }
}
```