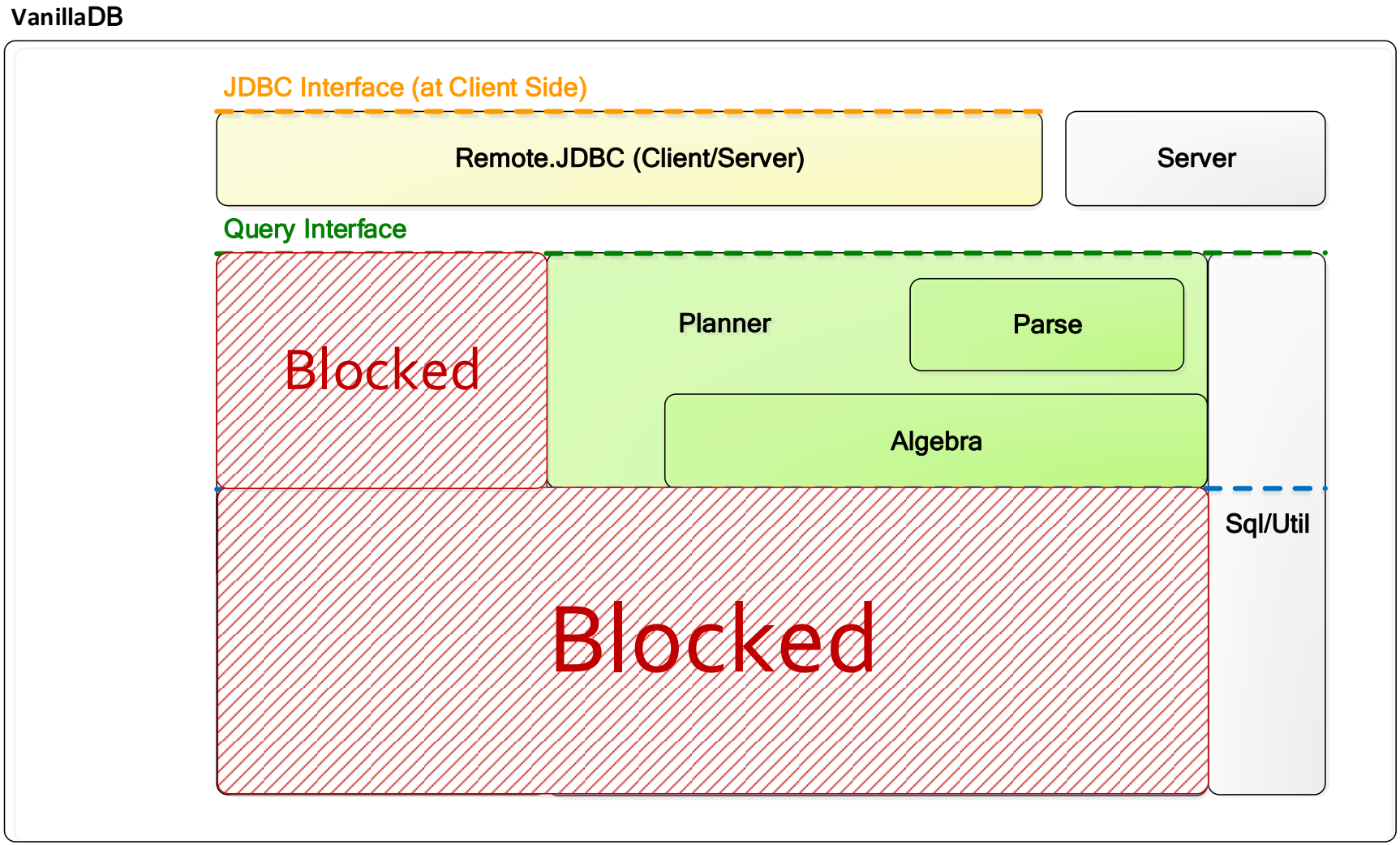
VanillaCore Walkthrough Part 3

Cloud Databases

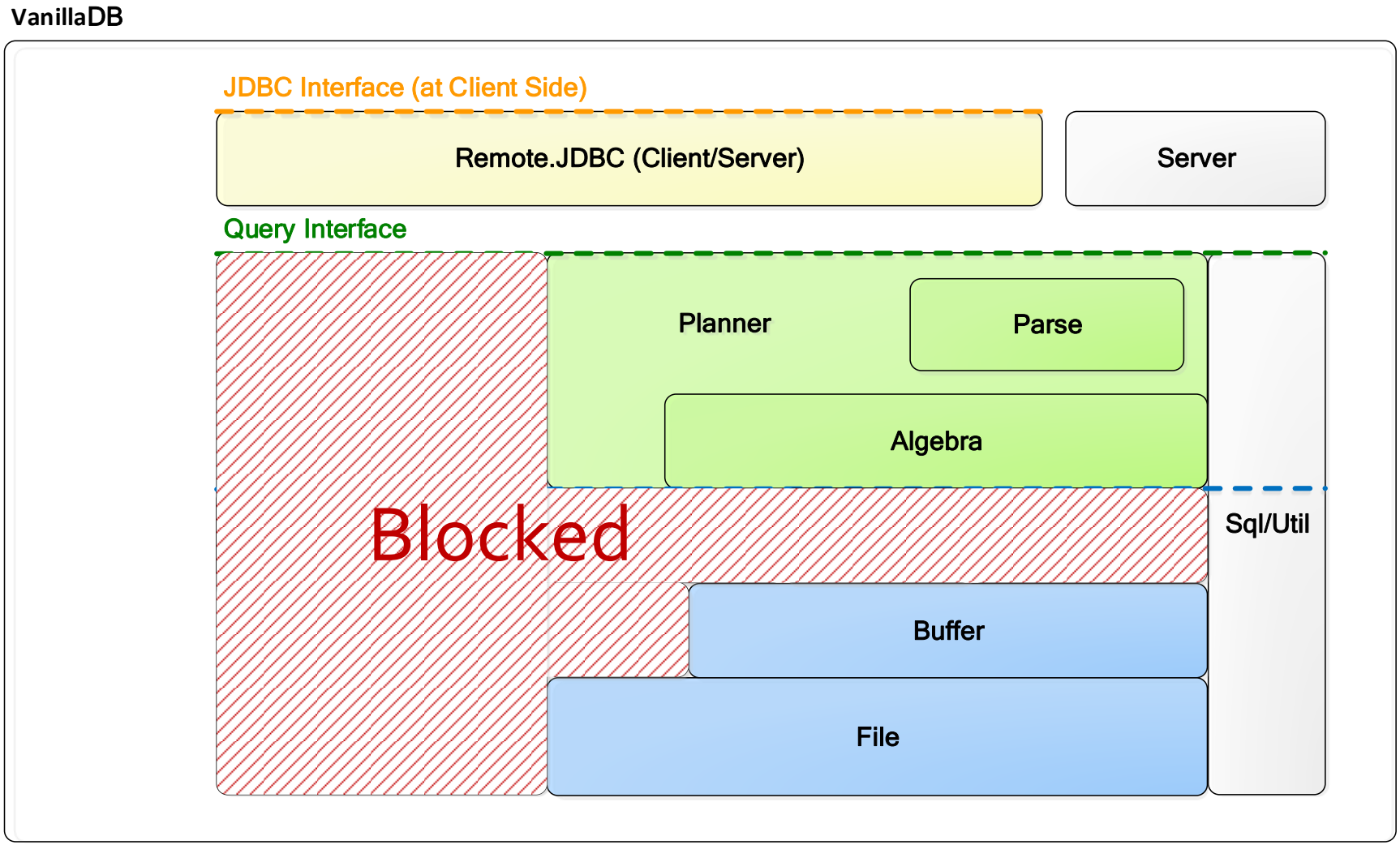
DataLab

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

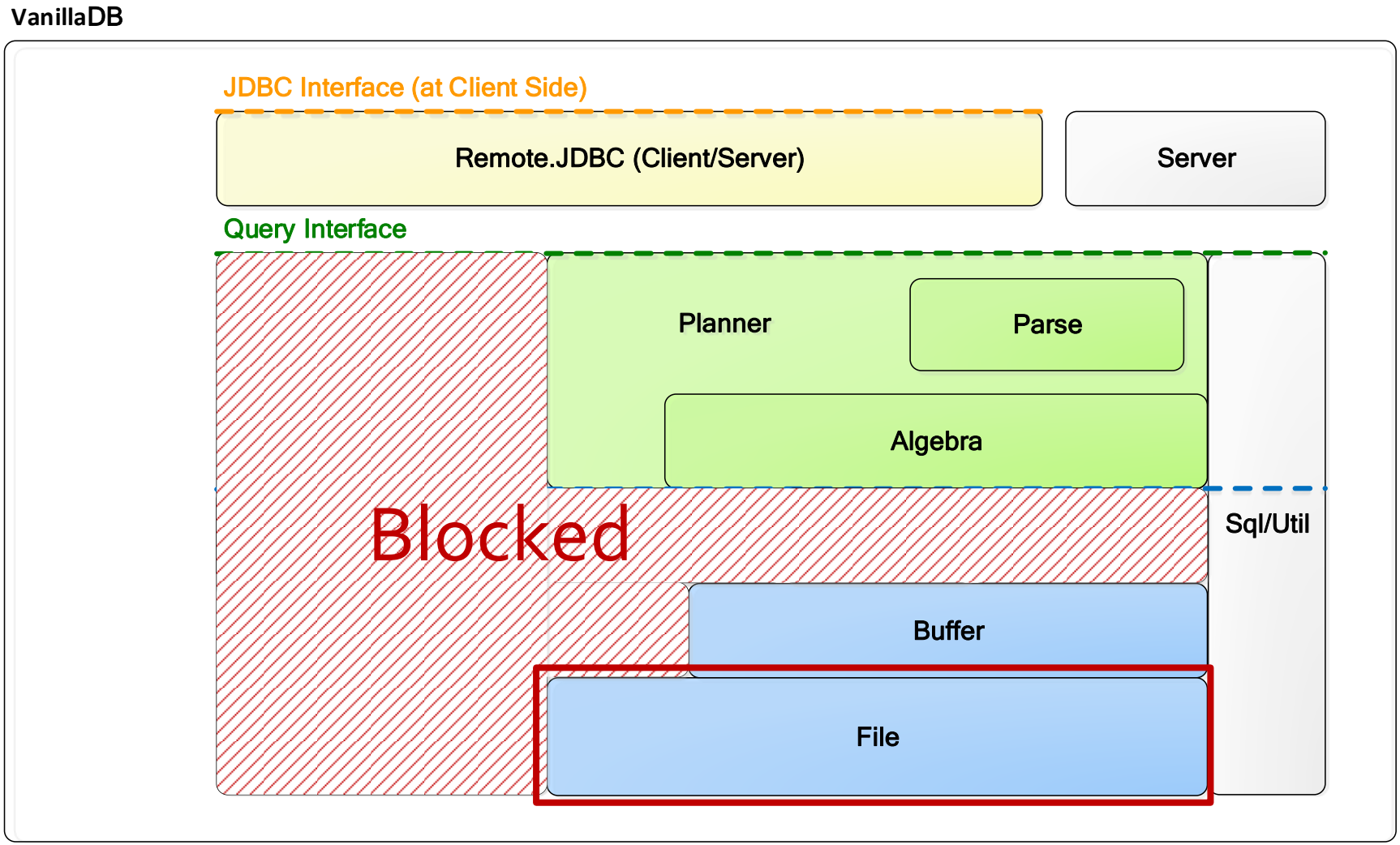
Outline

- File package
- Buffer package

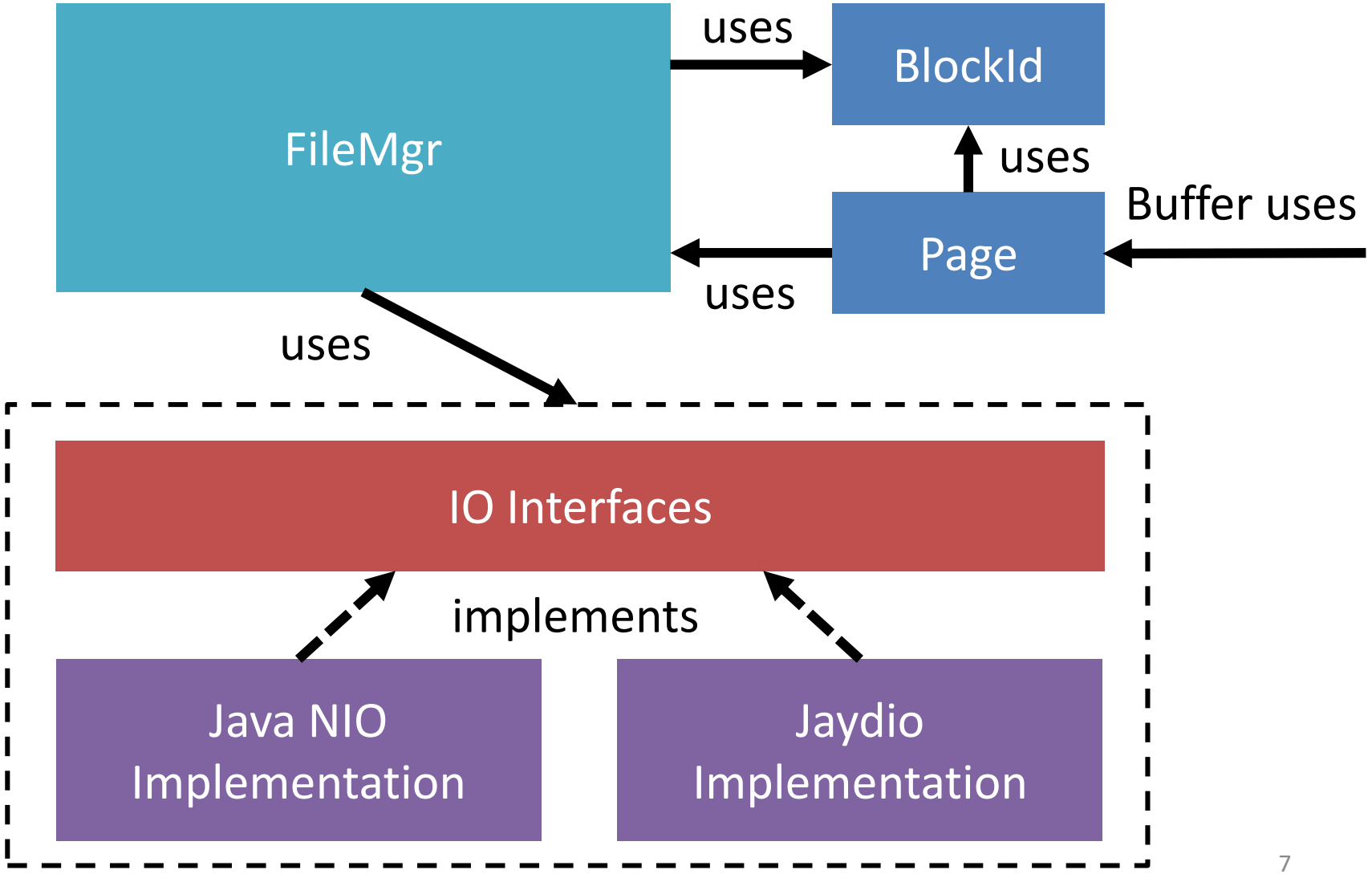
Outline

- File package
- Buffer package

Where are we?

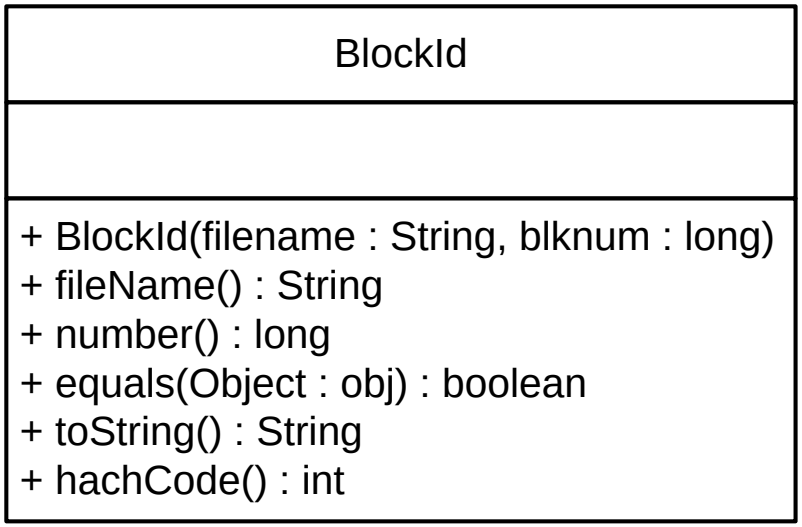

VanillaDB
JDBC Interface (at Client Side)
Remote.JDBC (Client/Server)
Server
Query Interface
Planner
Parse
Algebra
Blocked
Sql/Util
Buffer
File

`file` Package

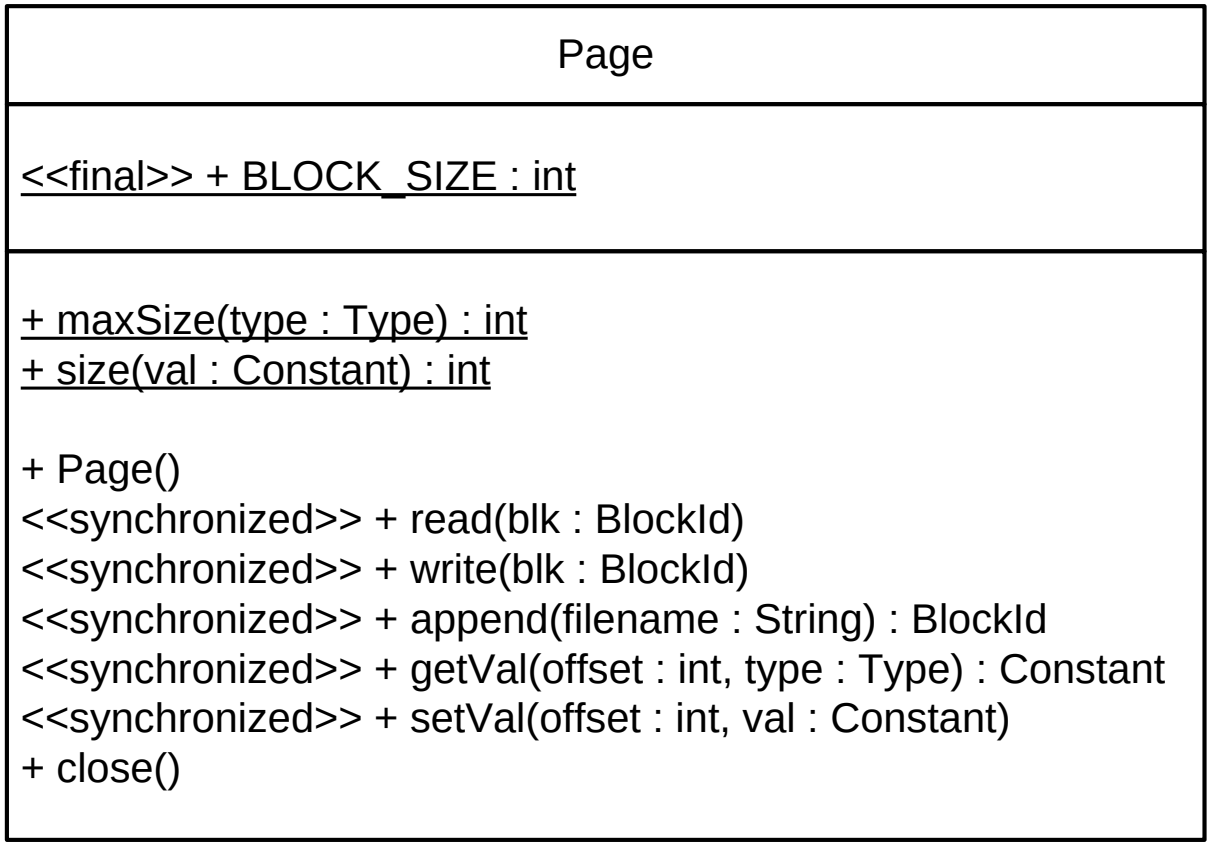


BlockId

```
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

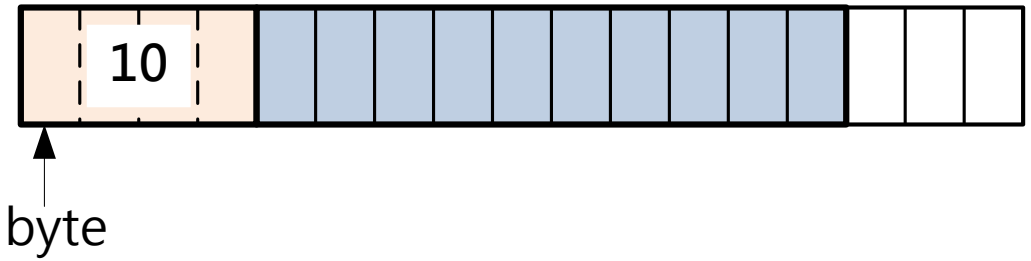
- Backed by `IoBuffer`

```
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?

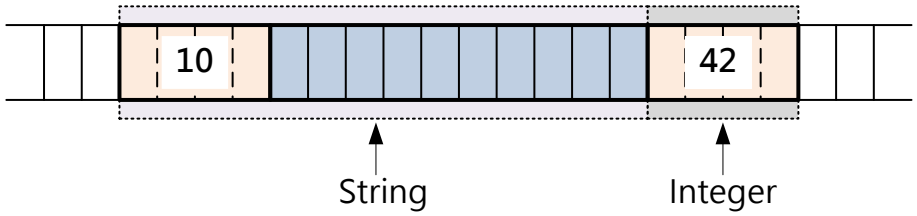
Storing A Varchar

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



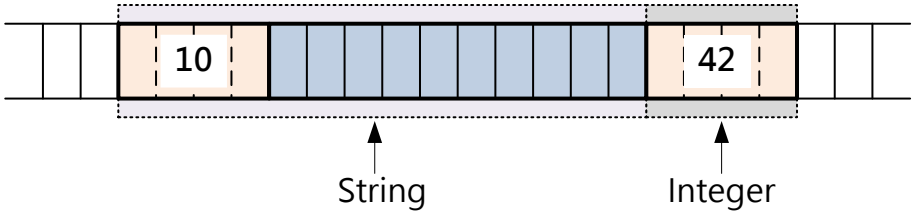
setVal

```
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

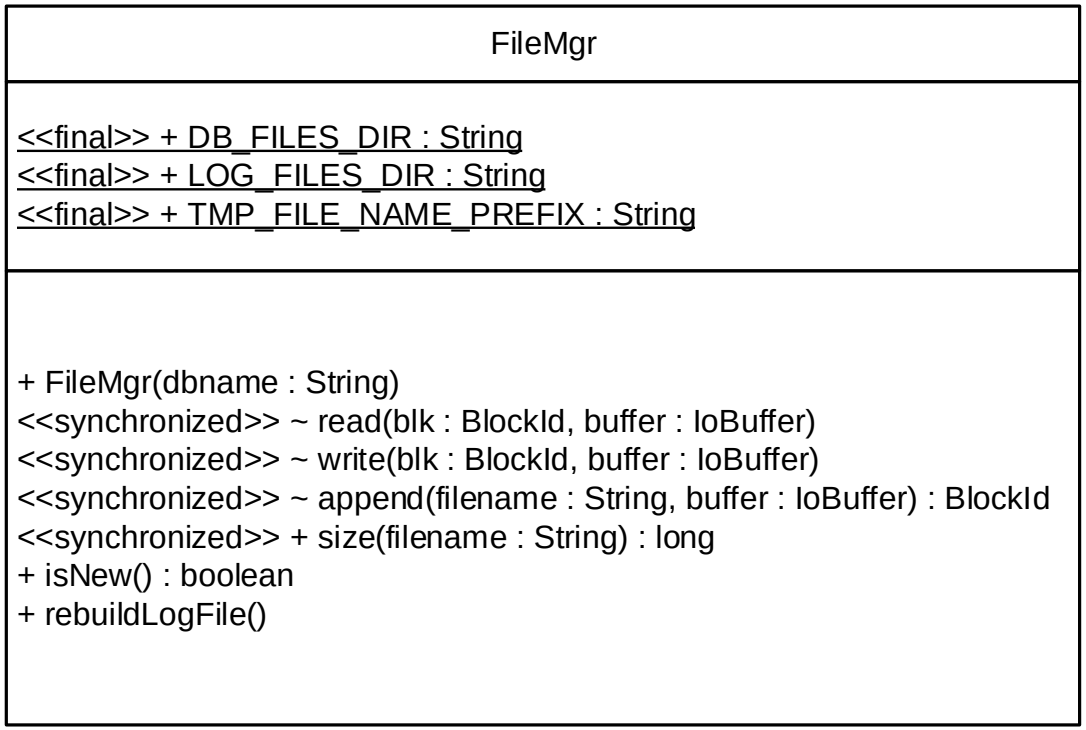
```
public static int maxSize(Type type) {
     return type.isFixedSize() ? type.maxSize() : ByteHelper.INT_SIZE
               + type.maxSize();
}

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

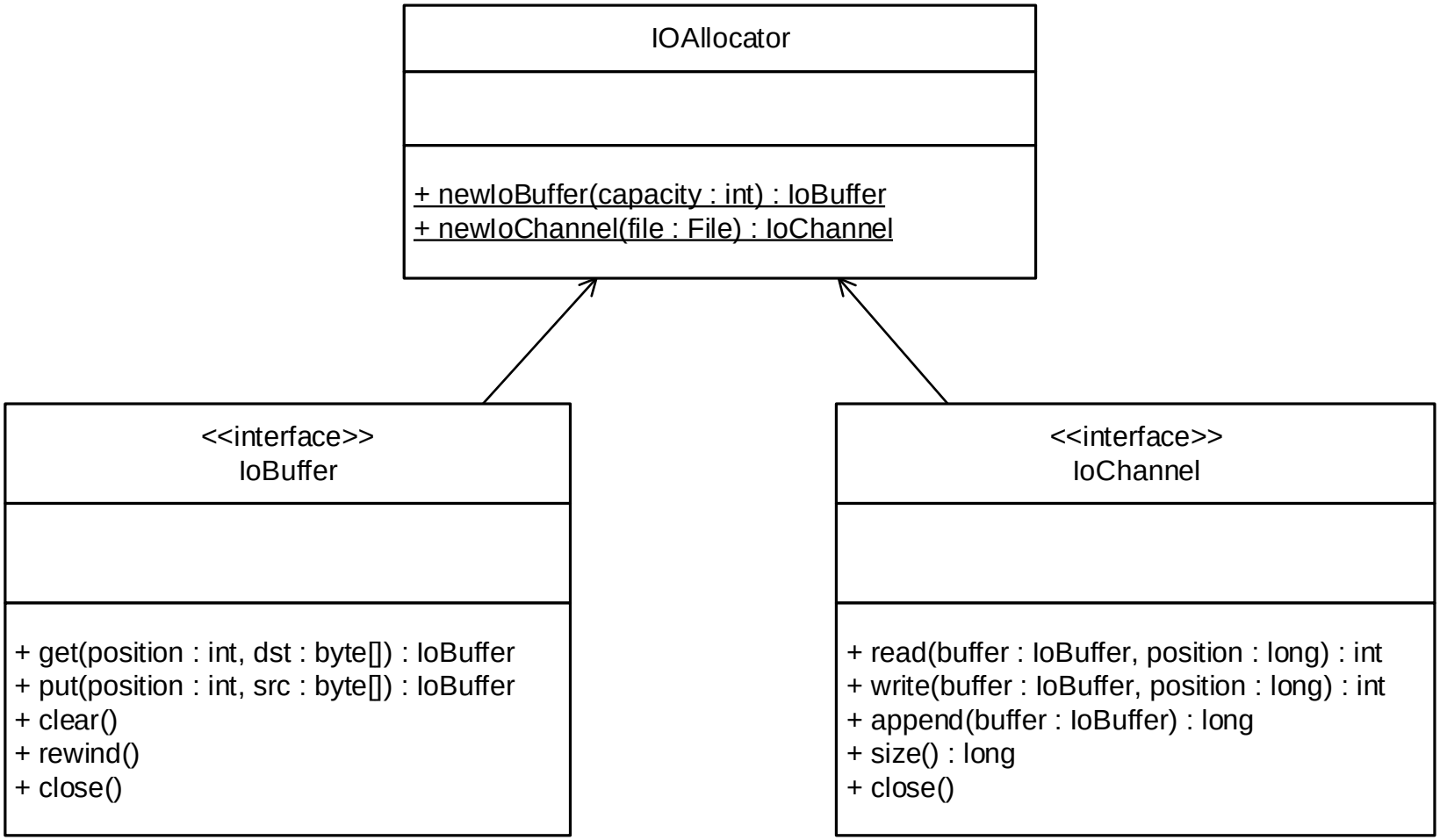
- Handles the actual I/Os
- Keeps the `IoChannel` instances of all opened files



`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 (TBA)