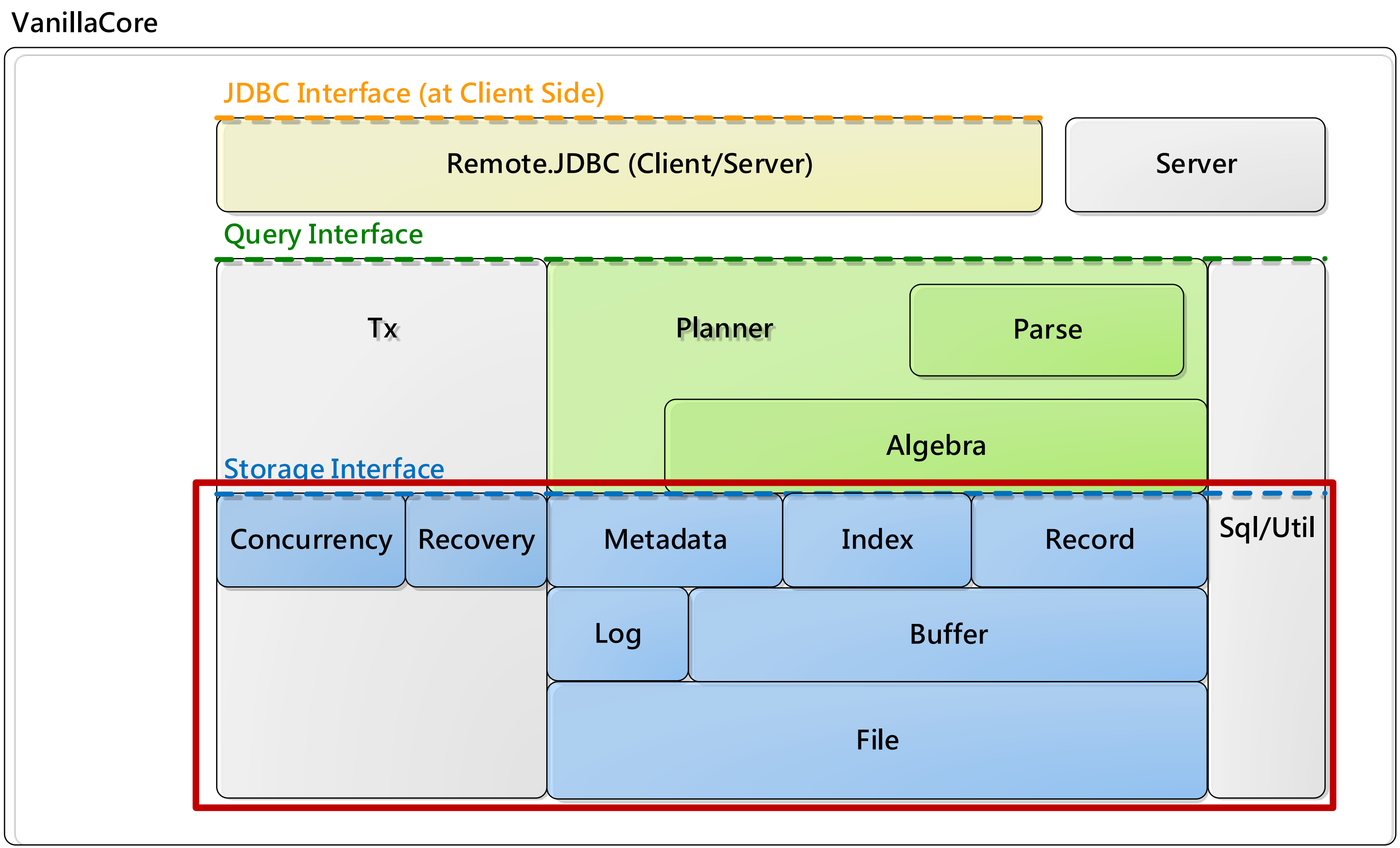
Data Access and File Management

Shan-Hung Wu & DataLab
CS, NTHU

Storage Engine



Outline

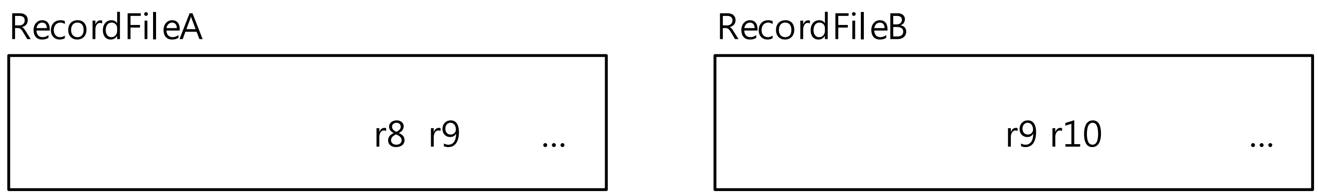
- Storage engine and data access
- Disk access
  - Block-level interface
  - File-level interface
- File Management in VanillaCore
  - `BlockID`, `Page`, and `FileMgr`
  - I/O interfaces

Outline

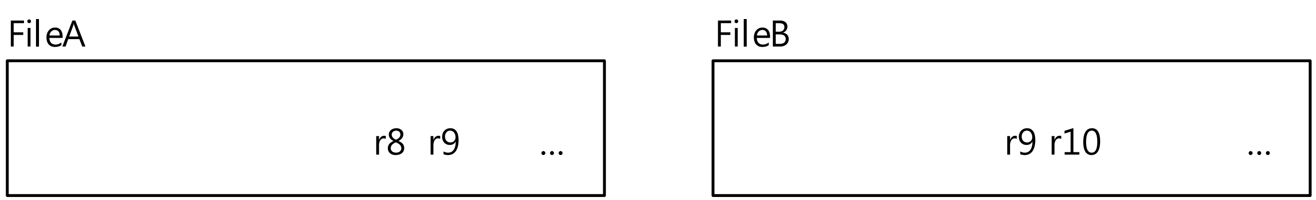
- Storage engine and data access
- Disk access
  - Block-level interface
  - File-level interface
- File Management in VanillaCore
  - `BlockID`, `Page`, and `FileMgr`
  - I/O interfaces

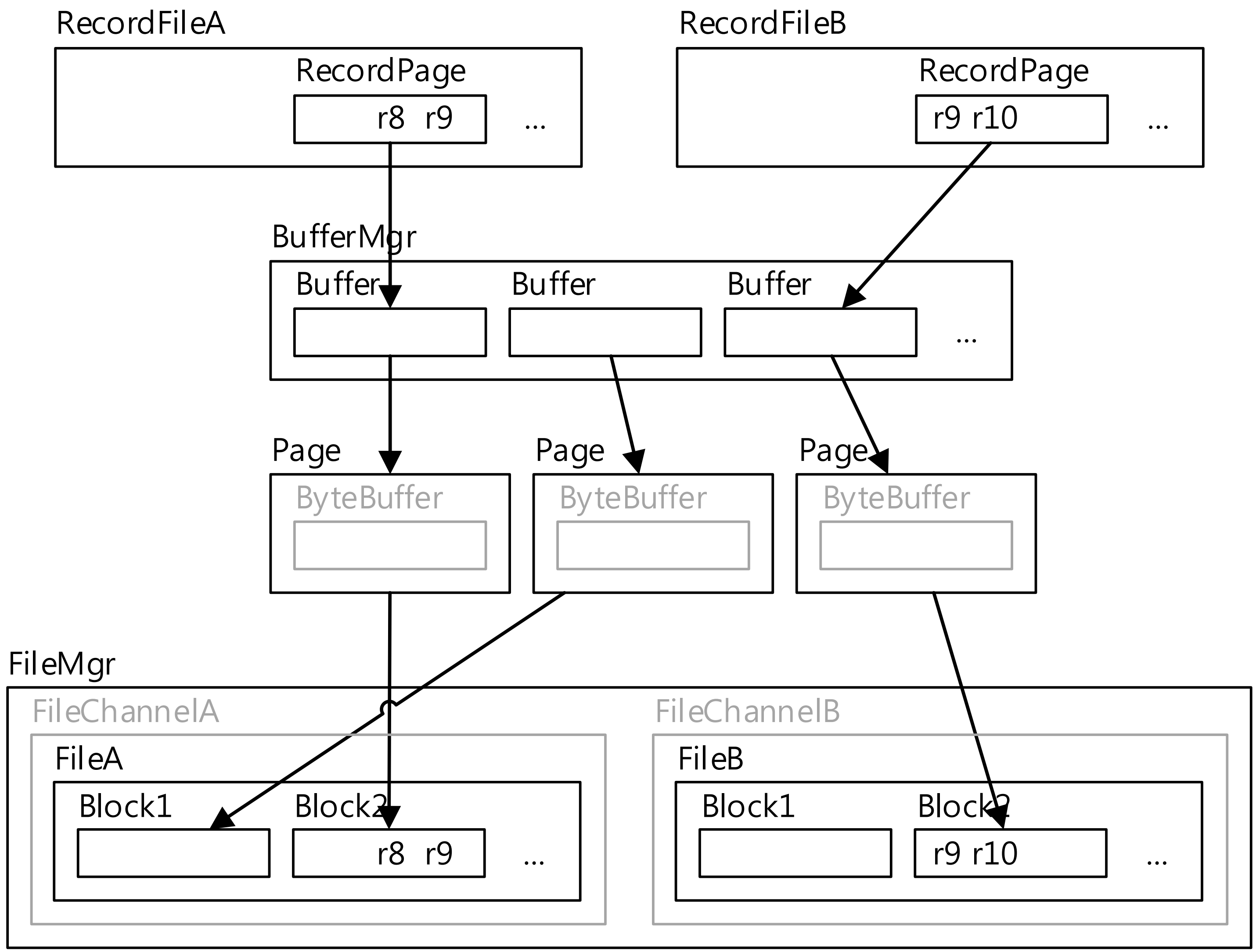
Storage Engine

- Main functions:
- Data access
  - File access (`TableInfo`, `RecordFile`)
  - Metadata access (`CatalogMgr`)
  - Index access (`IndexInfo`, `Index`)
- Transaction management
  - C and I (`ConcurrencyMgr`)
  - A and D (`RecoveryMgr`)



How does a `RecordFile` map to an Actual File on Disk?




RecordFileA
RecordPage
r8 r9
...
RecordFileB
RecordPage
r9 r10
...
BufferMgr
Buffer
Buffer
Buffer
...
Page
ByteBuffer
Page
ByteBuffer
Page
ByteBuffer
FileMgr
FileChannelA
FileA
Block1
Block2
r8 r9
...
FileChannelB
FileB
Block1
Block2
r9 r10
...

Data Access Layers (Bottom Up)

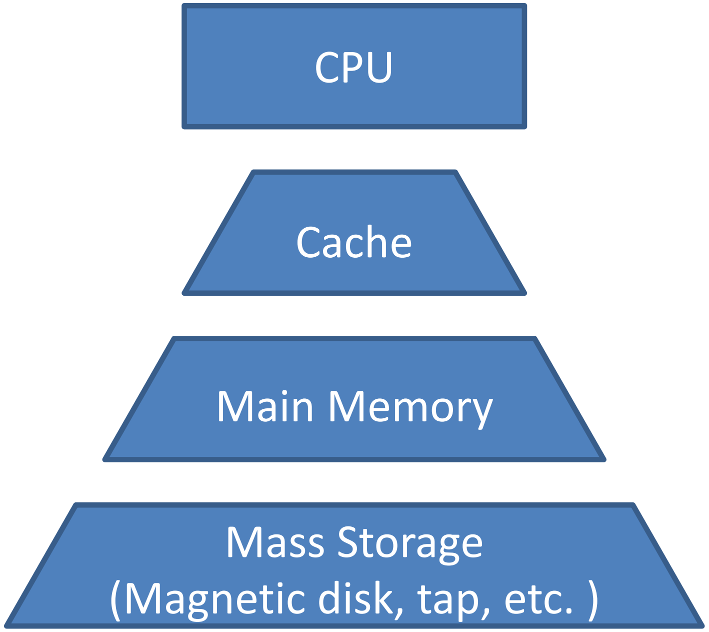
- In `storage.file` package: `Page` and `FileMgr`
  - ***Access disks*** as fast as passible
- In `storage.buffer` package: `Buffer` and `BufferMgr`
  - ***Cache pages***
  - Work with recover manager to ensure A and D
- In `storage.record` package: `RecordPage` and `RecordFile`
  - ***Arrange records in pages***
  - ***Pin/unpin buffers***
  - Work with recover manager to ensure A and D
  - Work with concurrency manager to ensure C and I
- `Index`
- `CatalogMgr`

Outline

- Storage engine and data access
- Disk access
  - Block-level interface
  - File-level interface
- File Management in VanillaCore
  - `BlockID`, `Page`, and `FileMgr`
  - I/O interfaces

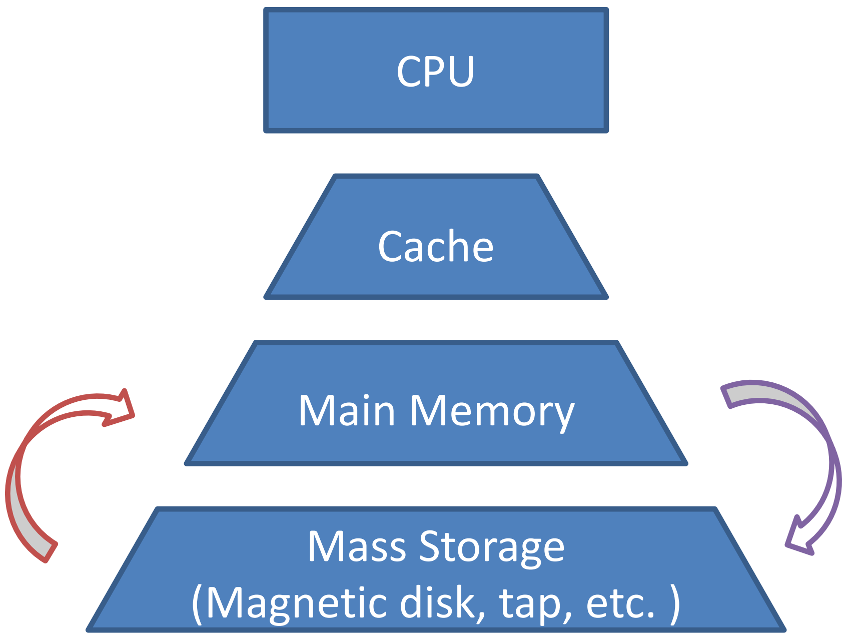
Why Disks?

- The contents of a database must be kept in ***persistent storages***
  - So that the data will not lost if the system goes down, ensuring D



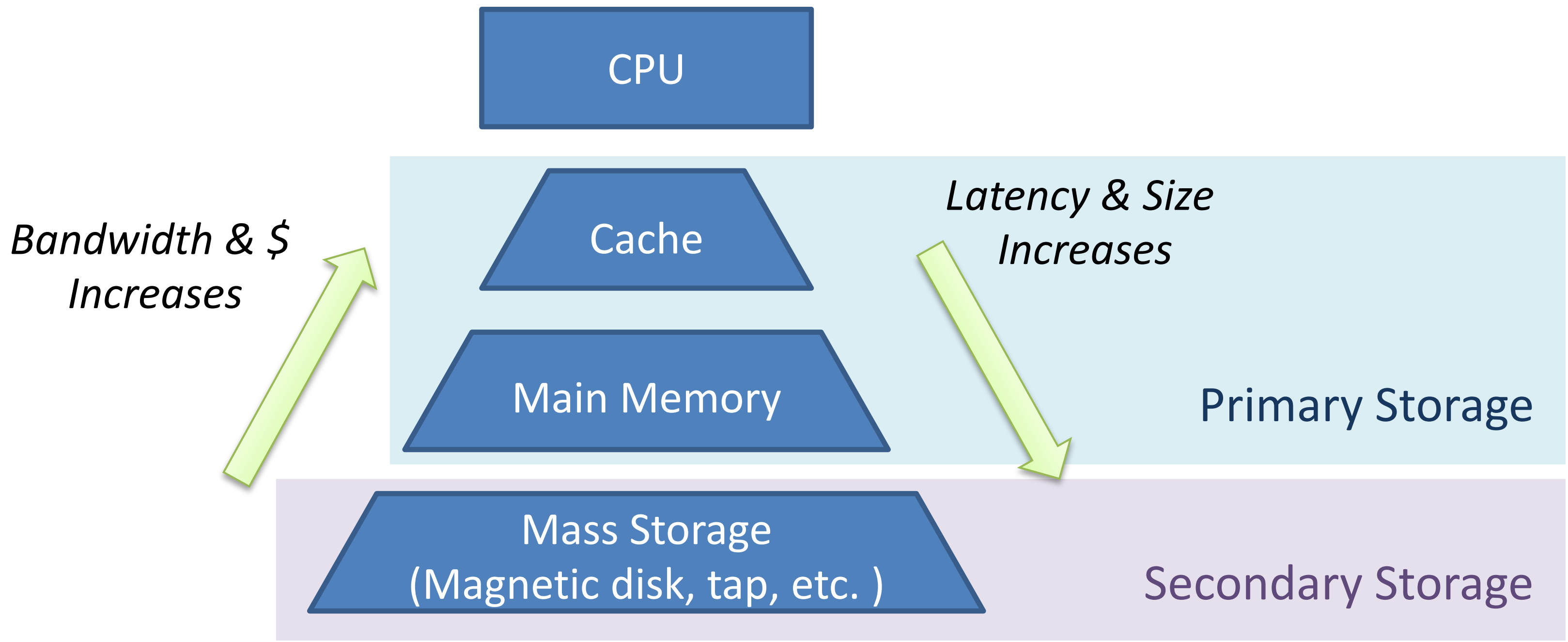
Disk and File Management

- I/O operations:
  - ***Read***: transfer data from disk to main memory (RAM)
  - ***Write***: transfer data from RAM to disk



Speed and $

- Primary storage is fast but small
- Secondary storage is large but ***slow***

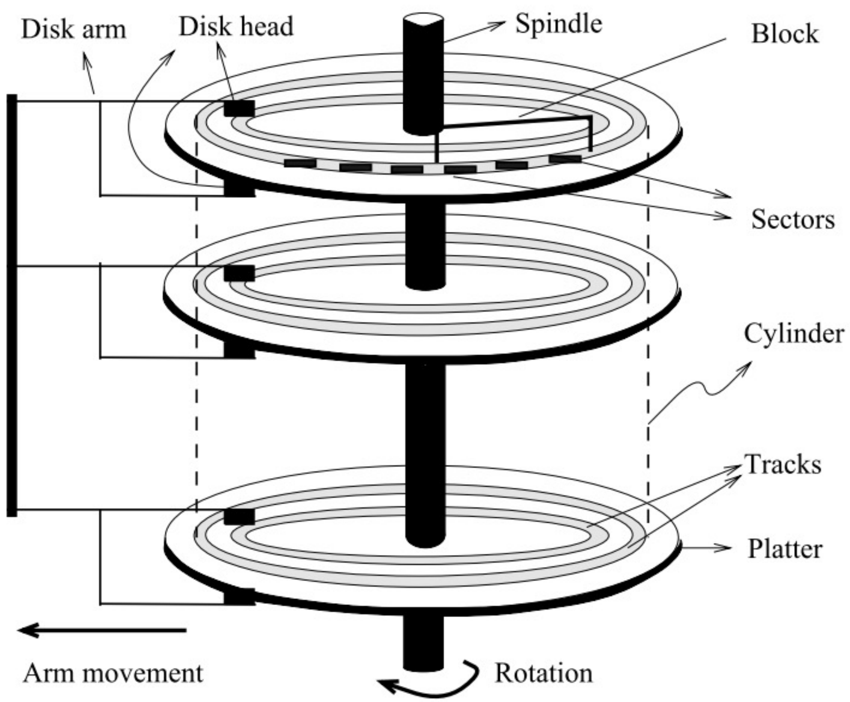


How Slow?

- Typically, accessing a block requires
  - ~60ns on RAMs
  - ~6ms on HDDs
  - ~0.06ms on SSDs
- HDDs are 100,000 times slower than RAMs!
- SSDs are 1,000 times slower than RAMs!

Understanding Magnetic Disks

- Data are stored on disk in units called ***sectors***
- ***Sequential access*** is faster than ***random access***
  - The disk arm movement is slow
- Access time is the sum of the ***seek time***, ***rotational delay***, and ***transfer time***



From Database Management System 2/e, Ramakrishnan.

Access Delay

- Seek time: 1~20ms
- Rotational delay: 0~10ms
- Transfer rate is about 1ms per 4KB page
- Seek time and rotational delay dominate

How about SSDs?

- Typically under 0.1ms delay for random access
- Sequential access may still be faster than random access
  - SSDs ***always read/write an entire block*** even when only a small portion is needed
- But if reads/writes are all comparable in size to a block, there will be no much performance difference

OS’s Disk Access APIs

- OS provides two disk access APIs:
- ***Block-level*** interface
  - A disk is formatted and mounted as a raw disk
  - Seen as a collection of blocks
- ***File-level*** interface
  - A disk is formatted and accessed by following a particular protocol
    - E.g., FAT, NTFS, EXT, NFS, etc.
  - Seen as a collection of files (and directories)

Outline

- Storage engine and data access
- Disk access
  - Block-level interface
  - File-level interface
- File Management in VanillaCore
  - `BlockID`, `Page`, and `FileMgr`
  - I/O interfaces

Block-Level Abstraction

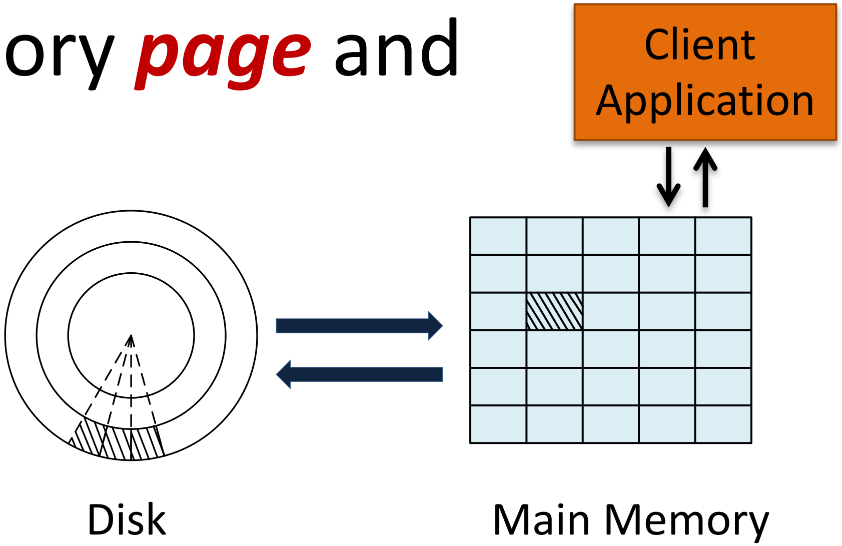
- Disks may have different hardware characteristics
  - In particular, different sector sizes
- OS hides the sectors behind ***blocks***
  - The unit of I/O above OS
  - Size determined by OS

Translation

- OS maintains the mapping between blocks and sectors
- Single-layer translation:
  – Upon each call, OS translates from the ***block number*** (starting from 0) to the actual sector address

Block-Level Interface

- The contents of a block cannot be accessed directly from the disk
  - May be mapped to more than one sectors
- Instead, the sectors comprising the block must first be read into a memory ***page*** and accessed from there
- ***Page:*** a block-size area in main memory



API

- `readblock(n, p)`
  - reads the bytes at block *n* into page *p* of memory
- `writeblock(n, p)`
  - writes the bytes in page *p* to block *n* of the disk

- OS also tracks of which blocks on disk are available for allocation
- `allocate(k, n)`
  - finds *k* contiguous unused blocks on disk and marks them as used
  - New blocks should be located as close to block *n* as possible
- `deallocate(k, n)`
  - marks the *k* contiguous blocks starting with block *n* as unused

Outline

- Storage engine and data access
- Disk access
  - Block-level interface
  - File-level interface
- File Management in VanillaCore
  - `BlockID`, `Page`, and `FileMgr`
  - I/O interfaces

File-Level Abstraction

- OS provides another, higher-level interface to the disk, called the ***file system***
- A file is a sequence of bytes
- Clients can read/write any number of bytes starting at any position in the file
- ***No notion of block*** at this level

File-Level Interface

- E.g., the Java class `RandomAccessFile`
- To increment 4 bytes stored in the file “file1” at offset 700:

```
RandomAccessFile f = new RandomAccessFile("file1", "rws");

f.seek(700);
int n = f.readInt();  // after reading pointer moves to 704

f.seek(700);
f.writeInt(n + 1);

f.close();
```

Block Access?

- Yes!
  - What does the "s" mode mean?

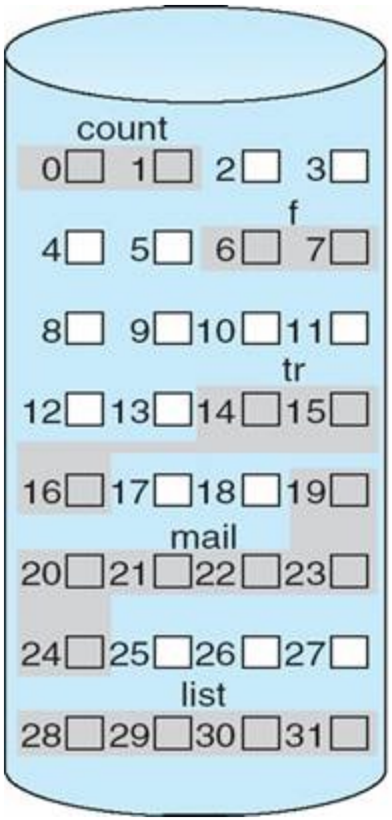
```
RandomAccessFile f =
    new RandomAccessFile("file1", "rws");
...
f.writeInt(...);
```

- OS hides the pages, called ***I/O buffers***, for file I/Os
- OS also hides the blocks of a file

Hidden Blocks of a File

- OS treats a file as a sequence of ***logical blocks***
  - For example, if blocks are 4096 bytes long
  - Byte 700 is in logical block 0
  - Byte 7992 is in logical block 1
- Logical blocks **≠** physical blocks (that format a disk)
- Why?

Continuous Allocation



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

- Stores each file in continuous physical blocks
- Cons:
  - Internal fragmentation
  - External fragmentation

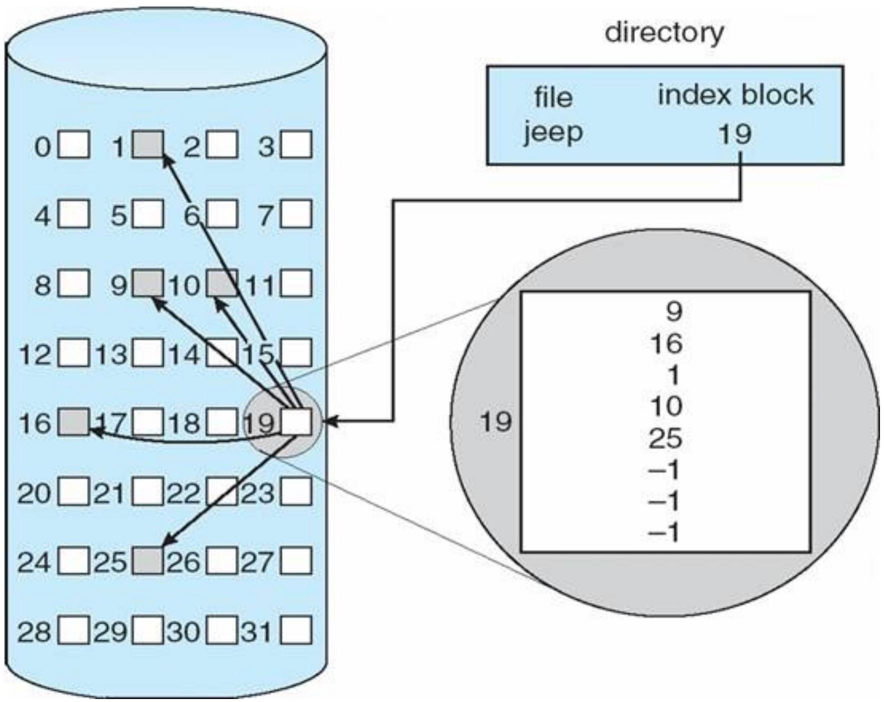
From Hussein M. Abdel-Wahab , CS 471 – Operating Systems Slides. http://www.cs.odu.edu/~cs471w/

Extent-Based Allocation

- Stores a file as a fixed-length sequence of ***extents***
  – An extent is a continuous chunk of physical blocks
- Only mitigates external fragmentation
  – Problem not solved

Indexed Allocation

- Keeps a special ***index block*** for each file
  - Which records of the physical blocks allocated to the file



From Hussein M. Abdel-Wahab, CS 471 – Operating Systems Slides. http://www.cs.odu.edu/~cs471w/

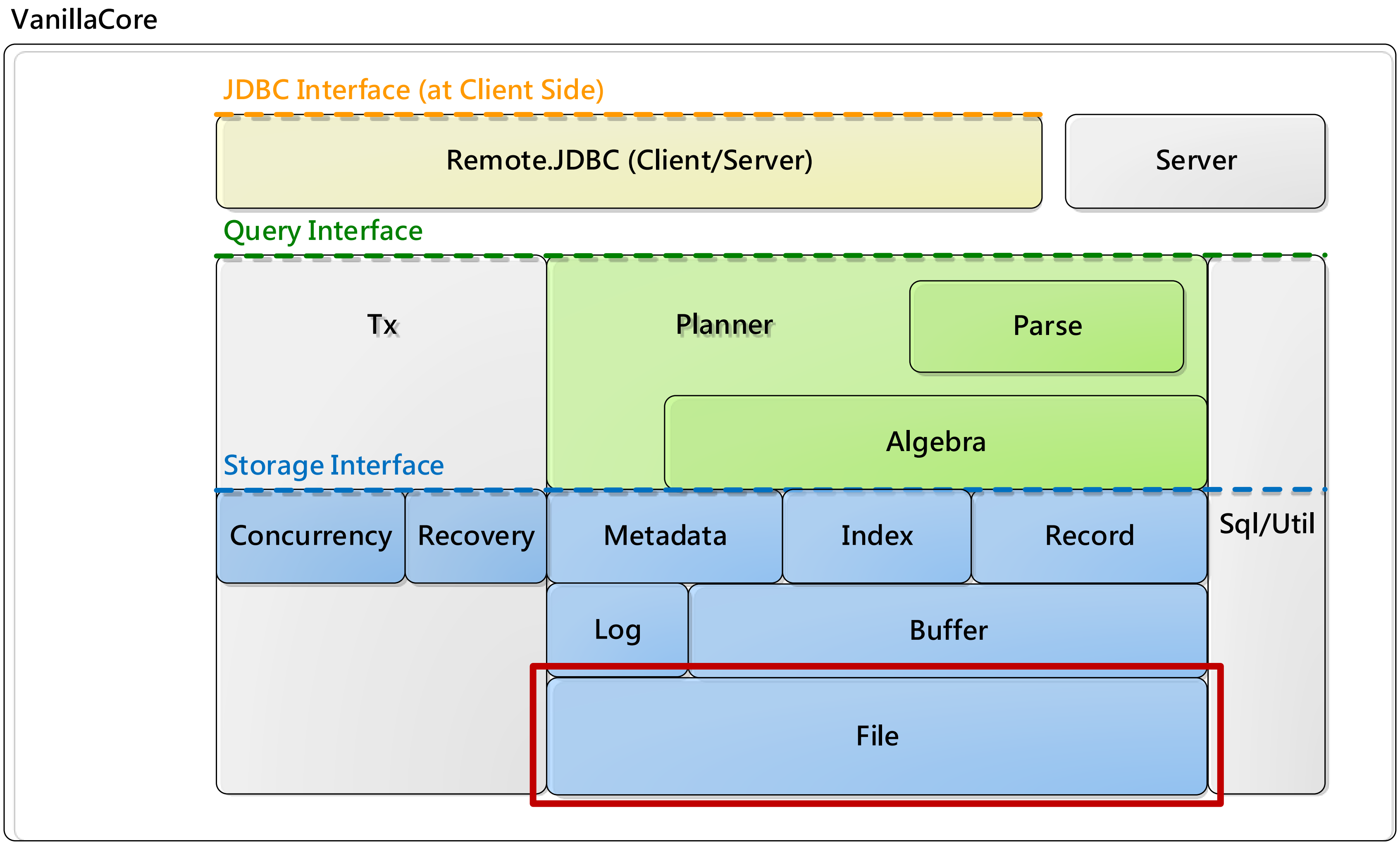
Translation

- OS maintains the mapping between logical and physical blocks
  - Specific to file system implementation
- When `seek` is called
- Layer 1: byte position → logical block
- Layer 2: logical block → physical block
- Layer 3: physical block → sectors

Outline

- Storage engine and data access
- Disk access
  - Block-level interface
  - File-level interface
- File Management in VanillaCore
  - `BlockID`, `Page`, and `FileMgr`
  - I/O interfaces

File Manager



Design Goal

- To access data in disks as fast as possible

- Two choices:
  - Based on the low-level block API
  - Based on the file system
- At which level?

Block-Level Based

- Pros:
  - Full control of physical positions of data
    - E.g., blocks accessed together can be stored nearby on disk, or
    - Most frequent blocks at middle tracks, etc.
  - Avoids OS limitations
    - E.g., larger files (even spanning multiple disks)

Block-Level Based

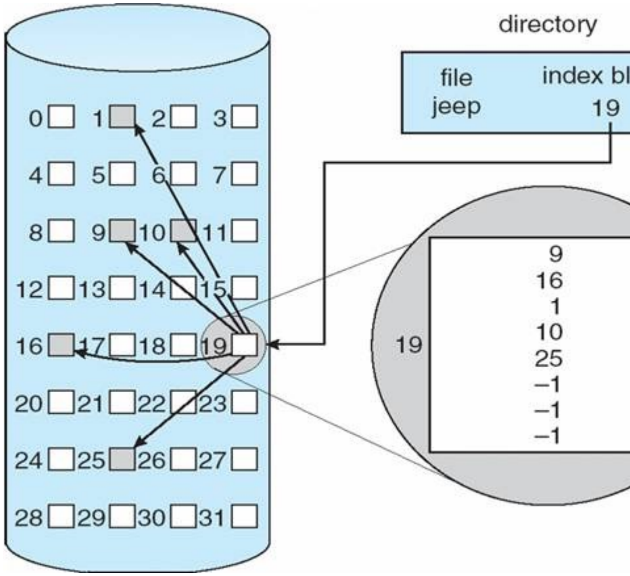
- Cons:
  - ***Complex*** to implement
    - Needs to manage the entire disk partitions and its free space
  - Inconvenient to some utilities such as (file) backups
  - “Raw disk” access is often OS-specific, which hurts portability
- Adopted by some commercial database systems that offer extreme performance

File-Level Based

- Pros:
  - Easy and convenient
- Cons:
  - Loses control to physical data placement
  - Loses track of pages (and their replacement)
  - Some implementations (e.g., postponed or reordered writes) ***destroy correctness*** (e.g., WAL)
- DBMS must flush by itself to guarantee ACID

VanillaCore's Choice

- A compromised strategy: at file-level, but access logical blocks directly
- Pros:
  - Simple
  - Manageable locality within a block
  - Manageable flush time (for correctness)
- Cons:
  - Needs to assume random disk access at all time
  - ***Even in sequential scans***
- Fast → minimizing #I/Os
- Adopted by many DBMS too
  - Microsoft Access, Oracle, etc.



Files

- A VanillaCore database is stored in several files under the database directory
  - One file for each table and index
    - Including catalog files
    - E.g., xxx.tbl, tblcat.tbl
  - Log files
    - E.g., vanilladb.log

Outline

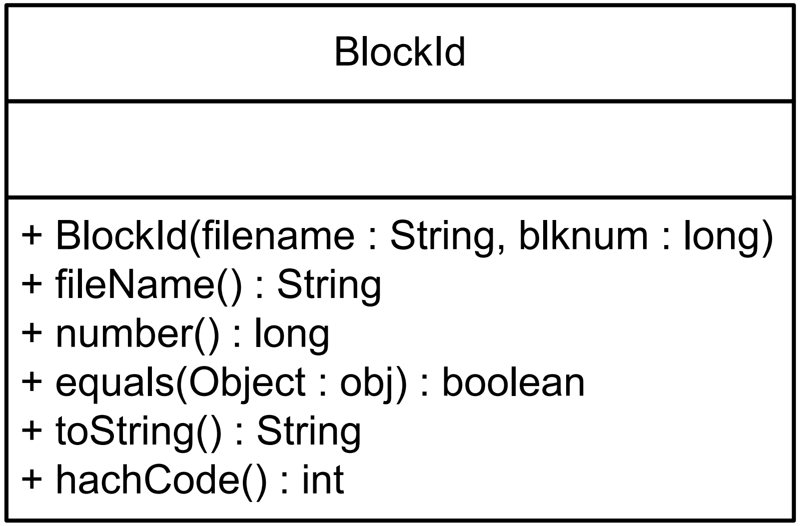
- Storage engine and data access
- Disk access
  - Block-level interface
  - File-level interface
- File Management in VanillaCore
  - `BlockID`, `Page`, and `FileMgr`
  - I/O interfaces

File Management

- `BlockId`, `Page` and `FileMgr`
- In package: `org.vanilladb.core.storage.file`

BlockId

- Immutable
- Identifies a specific logical block
  - A file name + logical block number
- For example,
  - `BlockId blk = new BlockId("std.tbl", 23);`



Page

- Holds the contents of a block
  - Backed by an I/O buffer in OS
- ***Not*** tied to a specific block
- Read/write/append an entire block a time
- Set values are ***not*** flushed until `write()`

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()

FileMgr

- Singleton, shared by all `Page` instances
- Handles the actual I/Os
- Keeps all opened files of a database
  - Each file is opened once and shared by all worker threads

```
FileMgr
---------------------------------------------
<<final>> + HOME_DIR : String
<<final>> + LOG_FILE_BASE_DIR : String
<<final>> + TMP_FILE_NAME_PREFIX : String
---------------------------------------------
+ FileMgr(dbname : String)
~ read(blk : BlockId, bb : IoBuffer)
~ write(blk : BlockId, bb : IoBuffer)
~ append(filename : String, bb : IoBuffer) : BlockId
+ size(filename : String) : long
+ isNew() : boolean
```

Using the VanillaCore File Manager

```java
VanillaDb.initFileMgr("studentdb");
FileMgr fm = VanillaDb.fileMgr();

BlockId blk1 = new BlockId("student.tbl", 0);
Page p1 = new Page();
p1.read(blk1);
Constant sid = p1.getVal(34, Type.INTEGER);
Type snameType = Type.VARCHAR(20);
Constant sname = p1.getVal(38, snameType);
System.out.println("student " + sid + " is " + sname);

Page p2 = new Page();
p2.setVal(34, new IntegerConstant(25));
Constant newName = new VarcharConstant("Rob").castTo(snameType);
p2.setVal(38, newName);
BlockId blk2 = p2.append("student.tbl");
```

Outline

- Storage engine and data access
- Disk access
  - Block-level interface
  - File-level interface
- File Management in VanillaCore
  - `BlockID`, `Page`, and `FileMgr`
  - I/O interfaces

I/O Interfaces

- Between VanillaCore and JVM/OS
- Two choices (both at file level):
  - Java New I/O
  - Jaydio (`O_Direct`, Linux only)
- To switch between these implementations, change the value of `USING_O_Direct` property in `vanilladb.properties` file

Java New I/O

- Each page wraps a `ByteBuffer` instance to store bytes
- `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***

Jaydio

- Provides similar interfaces to Java New I/O
- But with `O_Direct`
  - Some file systems (on Linux) ***cache*** file pages in its buffers for better performance
  - `O_Direct` tells those file systems ***not*** to cache file pages as we will implement our own caching policy (to be discussed in the next lecture)
  - Only available on Linux

Assigned Reading

- Java new I/O
  - In `java.nio`
- Classes:
  - `ByteBuffer`
  - `FileChannel`

References

- Ramakrishnan Gehrke, Database management System 3/e, chapters 8 and 9
- Edward Sciore, Database Design and Implementation, chapter 12
- Hellerstein, J. M., Stonebraker, M., and Hamilton, J., Architecture of a database system, 2007
- Hussein M. Abdel-Wahab, CS 471 – Operating Systems Slides, http://www.cs.odu.edu/~cs471w/