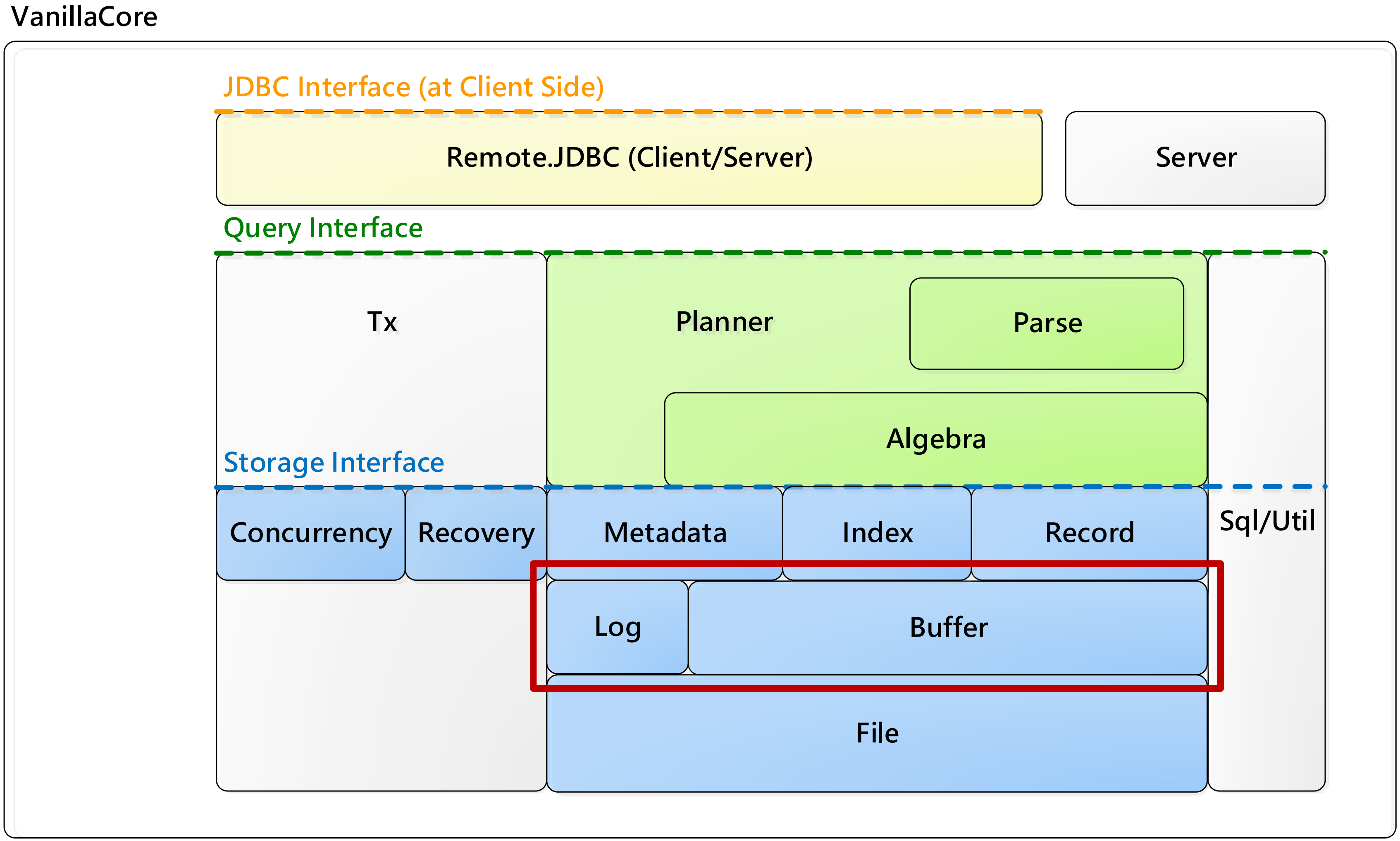
Memory Management

Shan-Hung Wu & DataLab
CS, NTHU

Memory Management



Outline

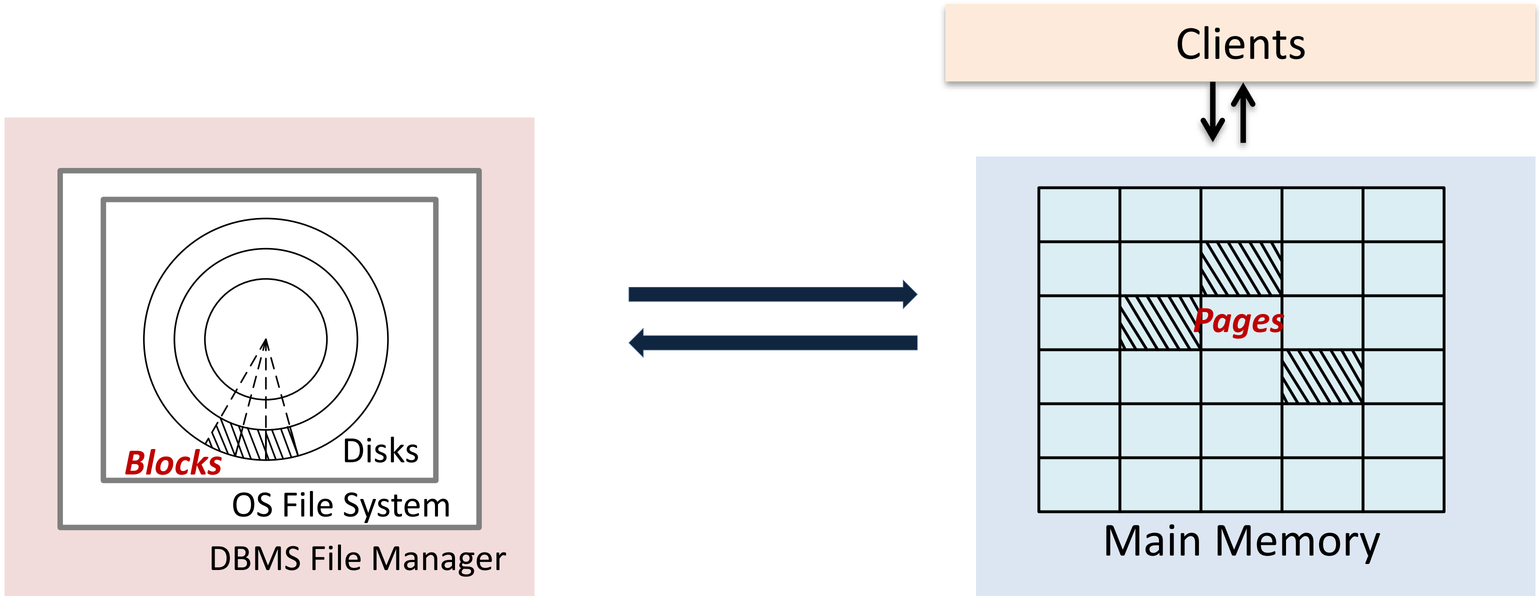
- Overview
- Buffering User Data
- Caching Logs
- Log Manager in VanillaCore
- Buffer Manager in VanillaCore

Outline

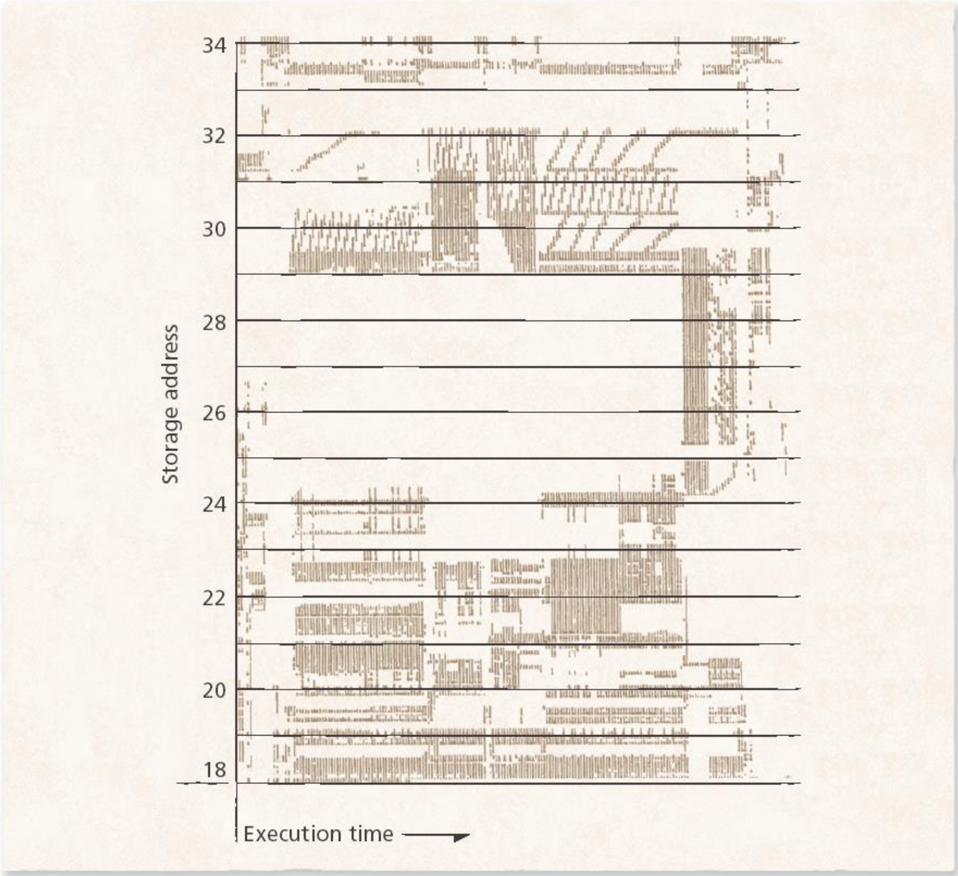
- Overview
- Buffering User Data
- Caching Logs
- Log Manager in VanillaCore
- Buffer Manager in VanillaCore

Consequences of Slow I/Os

- Architecture that minimizes I/Os:
  - Block access to/from disks
  - Self-managed caching of blocks
  - Choose the plan that costs least (fewest block I/Os)



Storage Access Patterns



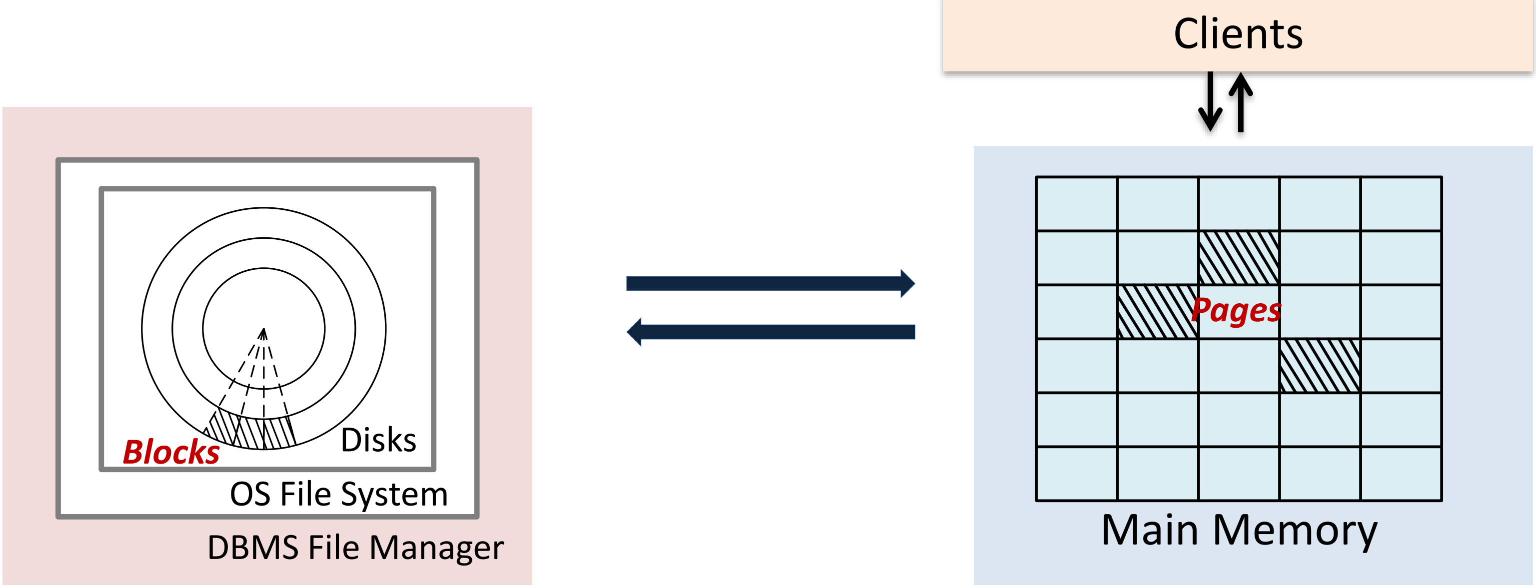
IBM Systems Journal, 1971

Storage Access Patterns

- ***Spatial locality***: each client (e.g., scan) focuses on a small number of blocks a time
  - Despite ending up with huge block accesses
  - E.g., to produce the next output record, a product scan needs only two blocks a time (left and right)
- ***Temporal locality***: recently used blocks are likely to be used in the near future
  - E.g., blocks of catalogs

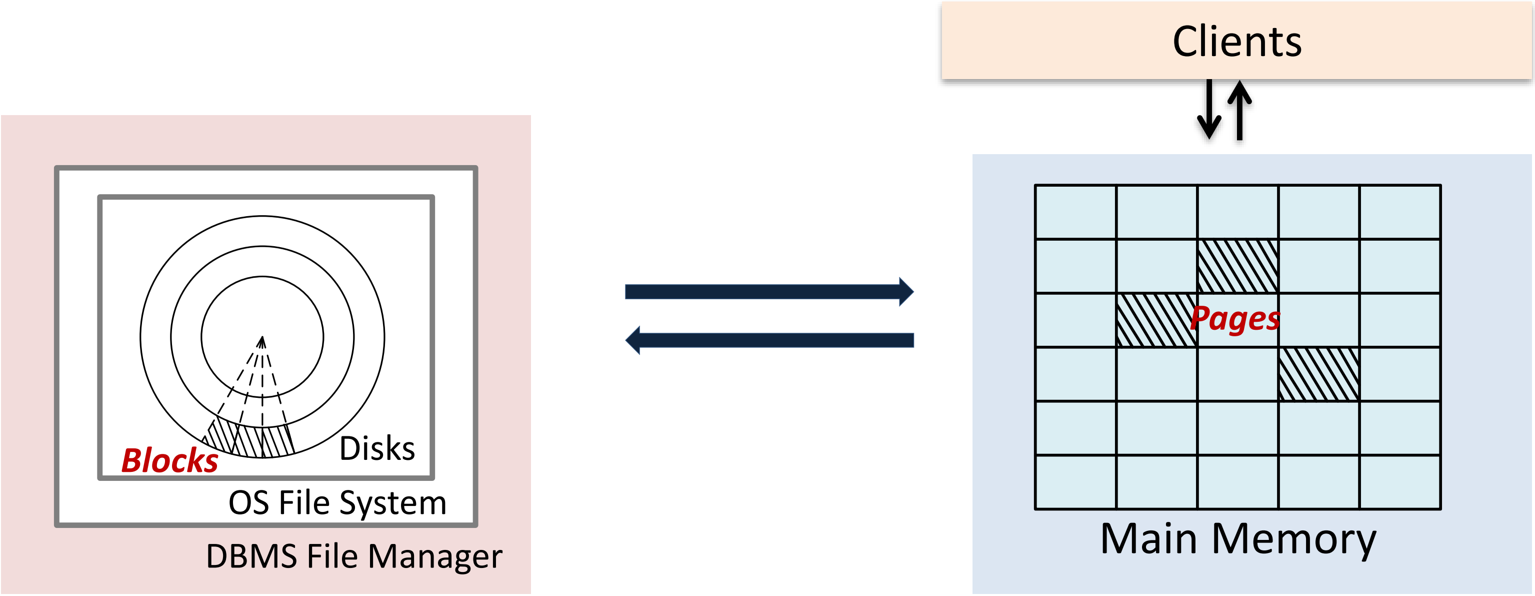
Minimizing Disk Access by Caching

- Idea: to reserve a pool of pages that keep the contents of most currently used blocks
  - To ***swap in***/***out*** blocks only when there's no empty page left in the pool



Benefits

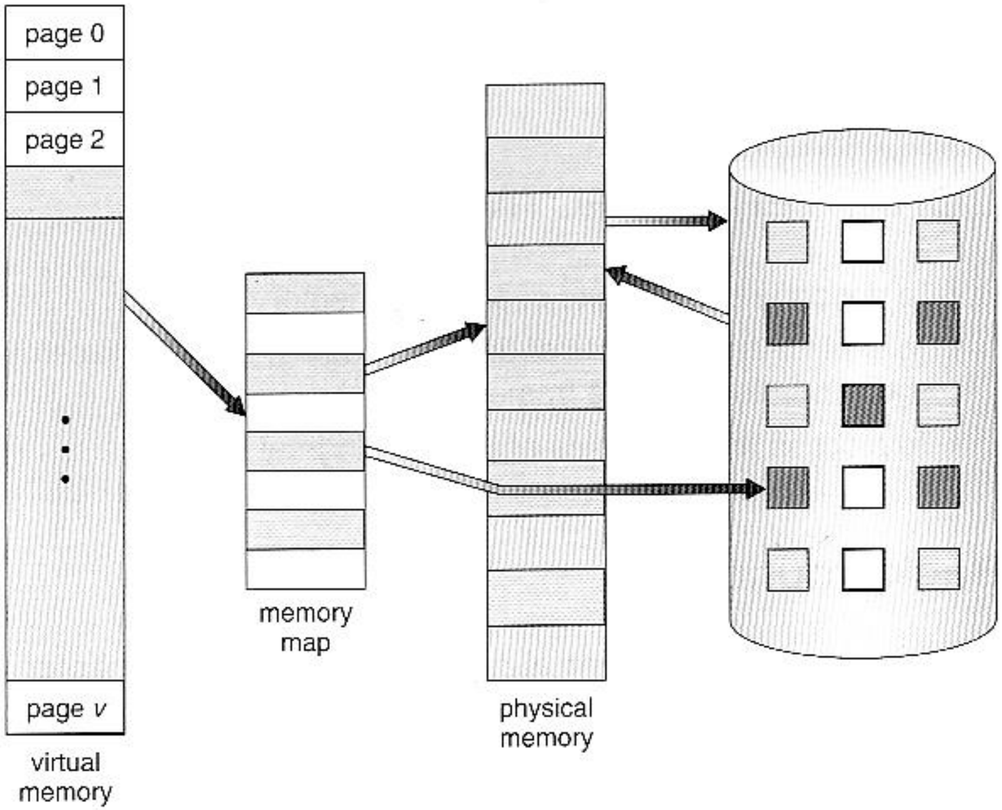
- Economic: only small memory space required
- Saves reads (if a requested block hits a page)
- Saves writes: all values set to a block only need to be written once upon swapping



Why not virtual memory?

Virtual Memory

- Modern OSs support ***virtual memory***
- Illusion: a very large address space for each process
  - – Larger than physical memory



From http://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/images/Chapter9/9_01_VirtualMemoryLarger.jpg

Don’t Rely on Virtual Memory (1/2)

- Problem 1: bad page replacement algorithms
  - E.g., FIFO, LRU, etc.
- OS has no idea which blocks will probably be used by a process in the near future
  - E.g., DBMS knows a user is likely to read the next record in a block (via scan.next())
  - But OS doesn’t

Don't Rely on Virtual Memory (2/2)

- Problem 2: uncontrolled delayed writes
  - Swapping is automatic
- When powered off, dirty pages may gone
  - Impairs the DBMS ability to recover after a system crash
  - Hurts durability of committed transactions
- Immediate writes?
  - Impairs the caching
  - Data may still corrupt due to partial writes upon crash
- Meta-writes (of logs) are needed

Self-Managed Pages in DBMS

- Pros:
- Controlled swapping
  - Fewer I/Os than VM via better replacement strategy
  - DBMS can tell which page must/cannot be flushed
- Supports meta-writes
  - DBMS can write logs to recover from crashes

What Blocks to Cache?

- Those of user data (DBs, including catalogs)
  - Pages for these blocks are managed by the ***buffer manager***
- Those of logs
  - In meta-writes
  - Pages managed by the ***log manager***

Outline

- Overview
- Buffering User Data
- Caching Logs
- Log Manager in VanillaCore
- Buffer Manager in VanillaCore

Access Pattern to User Blocks

- ***Random*** block reads and writes
  - From clients directly
  - Even from sequential scans (if above OS file system)
- Concurrent access to ***multiple*** blocks by multiple threads
  - Each thread per, e.g., JDBC client
- ***Predictable*** access to certain blocks
  - Each scan needs certain blocks a time
  - In particular, a table scan need one block a time and can forget what just read

Buffer Manger

- To reduce I/Os, the buffer manager allocates a pool of pages, called ***buffer pool***
  - Caching multiple blocks
  - Implement swapping
- Pages should be the direct I/O buffers held by the OS
  - Avoids swapping by VM
  - Eliminates the redundancy of double buffering
  - E.g., `ByteBuffer.allocateDirect()`

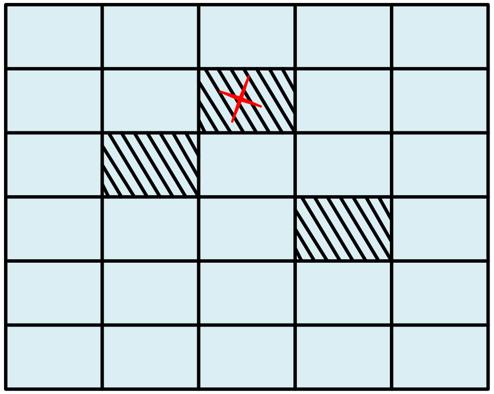
How do make use of predictable block accesses to further reduce I/Os?

Pinning Blocks

- Each table scan needs one block a time
  - The semantic of blocks is hidden behind the associated `RecordFile`
- It is the `RecordFile` instances that talk to memory manager about which blocks are needed
  - One instance per thread/client
- Through ***pinning***

Pinning Blocks

- When a `RecordFile` needs a block
  1. Asks buffer manager to ***pin*** (read-in) a block in some page
  2. Client accesses the contents of the page
  3. When the client is done with the block, it tells the buffer manager to ***unpin*** the block
- When swapping, only pages containing the ***unpinned*** blocks can be swapped out



Pinning Pages

- Results of pinning:
  1. A hit, no I/O
  2. Swapping: there exists at least one ***candidate page*** in the buffer pool holding unpinned block
     - Need to flush the page contents back to disk if the page is dirty
     - Which candidate page? ***replacement strategies***
     - Then read in the desired block
  3. Waiting: all pages in the buffer pool are pinned
     - Wait until some other unpins a page

Buffers

- Each page in the buffer pool needs to associate with additional information:
  - Is contained block pinned?
  - Is contained block modified (dirty)?
  - Information required by the replacement strategy
- A ***buffer*** wraps a page and hold this information
- A block can be pinned and accessed by multiple clients
  - Buffer must be thread safe (same as page)
  - DBMS needs other mechanism (i.e., concurrency control) to serialize conflict operations to a buffer

Example API

BufferMgr

<<final>> # BUFFER_SIZE : int

+ BufferMgr()
<<synchronized>> + pin(blk : BlockId, txNum : long) : Buffer
<<synchronized>> + pinNew(filename : String, fmtr : PageFormatter, txnum : long) : Buffer
<<synchronized>> + unpin(txnum : long, buffs : Buffer[])
+ flushAll(txnum : long)
+ available() : int

Buffer

~ Buffer()
<<synchronized>> + getVal(offset : int, type : Type) : Constant
<<synchronized>> + setVal(offset : int, val : Constant , txnum : long, lsn : long)
<<synchronized>> + block() : BlockId
<<synchronized>> ~ flush()
<<synchronized>> ~ pin()
<<synchronized>> ~ unpin()
<<synchronized>> ~ isPinned() : boolean
<<synchronized>> ~ isModifiedBy(txNum : long) : boolean
<<synchronized>> ~ assignToBlock(b : BlockId)
<<synchronized>> ~ assignToNew(filename : String, fmtr : PageFormatter)

- A block can be pinned multiple times
- There’s no guarantee that `pin()`’s on the same block will return the same buffer instance

Buffer Replacement Strategies

- All buffers in the buffer pool begin unallocated
- Once all buffers are loaded, buffer manager has to replace the unpinned block in some ***candidate buffer*** to serve new pin request
- Best candidate?
  - The buffer containing block that will be unused for the longest time
  - Maximizes the hit rate of pins

Buffer Replacement Strategies

- However, as in VM, access of blocks in unpinned buffers is not determinable
- Heuristics needed:
  - Naïve
  - FIFO
  - LRU
  - Clock
- Some commercial systems use different heuristics for different ***buffer type***
  - E.g., catalog buffers, index buffers, buffers for full table scan, etc.

Example

- A sequence of operations
  - `pin(10); pin(20); pin(30); pin(40); unpin(20); pin(50); unpin(40); unpin(10); unpin(30); unpin(50);`
- There are 4 buffers in buffer pool

Buffer	0	1	2	3
Block Id				
time read in				
time unpinned				

Example

- A sequence of operations
  - `pin(10); pin(20); pin(30); pin(40); unpin(20); pin(50); unpin(40); unpin(10); unpin(30); unpin(50);`
- There are 4 buffers in buffer pool

Buffer	**0** ✗	**1**	**2** ✗	**3** ✗
Block Id	**10**	**20**	**30**	**40**
time read in	**1**	**2**	**3**	**4**
time unpinned		**5**		

Example

- A sequence of operations
  - `pin(10); pin(20); pin(30); pin(40); unpin(20); pin(50); unpin(40); unpin(10); unpin(30); unpin(50);`
- There are 4 buffers in buffer pool

Buffer	**0** ✗	**1**	**2** ✗	**3** ✗
Block Id	**10**	**20**	**30**	**40**
time read in	**1**	**2**	**3**	**4**
time unpinned		**5**		

Example

- A sequence of operations
  - `pin(10); pin(20); pin(30); pin(40); unpin(20); pin(50); unpin(40); unpin(10); unpin(30); unpin(50);`
- There are 4 buffers in buffer pool

Buffer	0 ✗	1 ✗	2 ✗	3 ✗
Block Id	10	50	30	40
time read in	1	6	3	4
time unpinned		5		

Example

- A sequence of operations
  - `pin(10); pin(20); pin(30); pin(40); unpin(20); pin(50); unpin(40); unpin(10); unpin(30); unpin(50);`
- There are 4 buffers in buffer pool

Buffer	0	1	2	3
Block Id	10	50	30	40
time read in	1	6	3	4
time unpinned	8	10	9	7

Example

- Suppose that there are two more pin requests coming:
  - `pin(60); pin(70);`
- Let’s see how different replacement strategies work

Buffer	0	1	2	3
Block Id	10	50	30	40
time read in	1	6	3	4
time unpinned	8	10	9	7

The Naïve Strategy

- Travers the buffer pool sequentially from beginning
- Replaces the first unpinned buffer met
  - `pin(60); pin(70);`
- Easy to implement, but?

Buffer	0	1	2	3
Block Id	60	70	30	40
time read in	11	12	3	4
time unpinned	8	10	9	7

The Naïve Strategy

- Problem: buffers are not evenly utilized
  - `pin(60); unpin(60); pin(70); unpin(70); pin(60); unpin(60); ...`
- Low hit rate
  - Some buffers may contain stale data

Buffer	0	1	2	3
Block Id	60	50	30	40
time read in	15	6	3	4
time unpinned	16	10	9	7

The FIFO Strategy

- Chooses the buffer that contains the least-recently-read-in block
  - Each buffer records the ***time a block is read in***
- Unpinned buffers can be maintained in a priority queue
  - Finds the target unpinned buffer in O(1) time

Buffer	**0**	**1**	**2**	**3**
Block Id	10	50	30	40
time read in	1	6	3	4
time unpinned	8	10	9	7

The FIFO Strategy

- Chooses the buffer that contains the least-recently-read-in block
  - Each buffer records the ***time a block is read in***
- Unpinned buffers can be maintained in a priority queue
  - Finds the target unpinned buffer in O(1) time

Buffer	**0**	**1**	**2**	**3**
Block Id	**60**	**50**	**70**	**40**
time read in	**11**	**6**	**12**	**4**
time unpinned	**8**	**10**	**9**	**7**

The FIFO Strategy

- Assumption: the older blocks are less likely to be used in the future
- Valid?
- Not true for, e.g., catalog blocks

Buffer	0	1	2	3
Block Id	10	50	30	40
time read in	1	6	3	4
time unpinned	8	10	9	7

The LRU Strategy

- Chooses the buffer that contains the least recently used block
  - Each buffer records the ***time the block is unpinned***

Buffer	0	1	2	3
Block Id	10	50	30	40
time read in	1	6	3	4
time unpinned	8	10	9	7

The LRU Strategy

- Choose the buffer that contains the least recently used block
  - Each buffer records the ***time the block is unpinned***

Buffer	0	1	2	3
Block Id	60	50	30	70
time read in	11	6	3	12
time unpinned	8	10	9	7

The LRU Strategy

- Assumption: blocks that are not used in the near past will unlikely be used in the near future
  - Valid generally
  - Avoids replacing commonly used pages
- But still not optimal for full table scan
- Most commercial systems use simple enhancements to LRU

Buffer	0	1	2	3
Block Id	60	50	30	70
time read in	11	6	3	12
time unpinned	8	10	9	7

LRU Variants

- In Oracle DBMS, the LRU queue has two logical regions
  - – Cold region in front of the hot region
- ***Cold: LRU***; ***hot: FIFO***
- For full table scan
  - – Puts the just read page into the head (at LRU end)

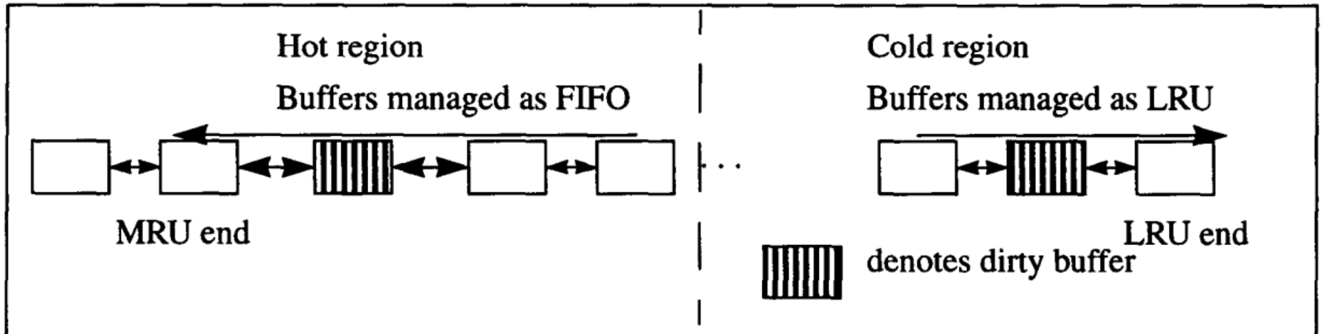
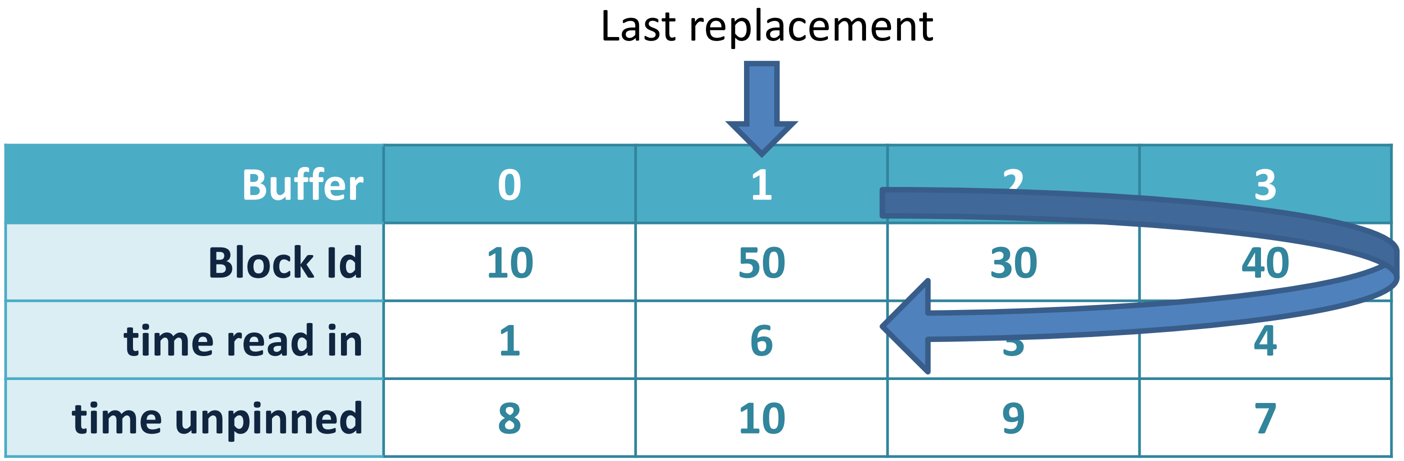


Figure 2: LRU chain management.

W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton, "The oracle universal server buffer," VLDB, 1997.

The Clock Strategy

- Similar to Naïve strategy, but always start traversal from the previous replacement position
- Uses the unpinned buffers as evenly as possible
  - With LRU flavor
- Easy to implement



Last replacement

Buffer	0	1	2	3
Block Id	10	50	30	40
time read in	1	6	3	4
time unpinned	8	10	9	7

The Clock Strategy

- Similar to Naïve strategy, but always start traversal from the previous replacement position
- Uses the unpinned buffers as evenly as possible
  - With LRU flavor
- Easy to implement

Last replacement

Buffer	0	1	2	3
Block Id	10	50	60	70
time read in	1	6	11	12
time unpinned	8	10	9	7

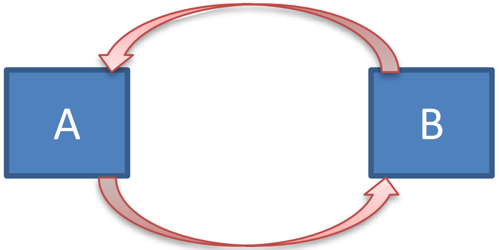
How many pages in buffer pool?

Pool Size

- The set of all blocks that are currently accessed by clients is called the ***working set***
- Ideally, the buffer pool should be larger than the working set
  - Otherwise, ***deadlock*** may happen

Deadlock

- What if there is no candidate buffer when pinning?
  - Buffer manager tells the client to wait
  - Notifies (wakes up) the client to pin again when some other unpins a block
- Deadlock
  - Clients *A* and *B* both want to use two buffers and there remain only two candidate buffers
  - If they both have got one buffer and attempt to get another one, deadlock happens
  - Circularly waiting the others to unpin



Deadlock

- How to detect deadlock?
  - No buffer becomes available for an exceptionally long time
  - E.g., much longer than executing a query
- How to deal with deadlock?
  - Forces at least one client to
  1. First unpin all blocks it holds
  2. Then re-pins these blocks one-by-one

Waiting: An Example

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12

A	B	C
	…	
		pin(12)

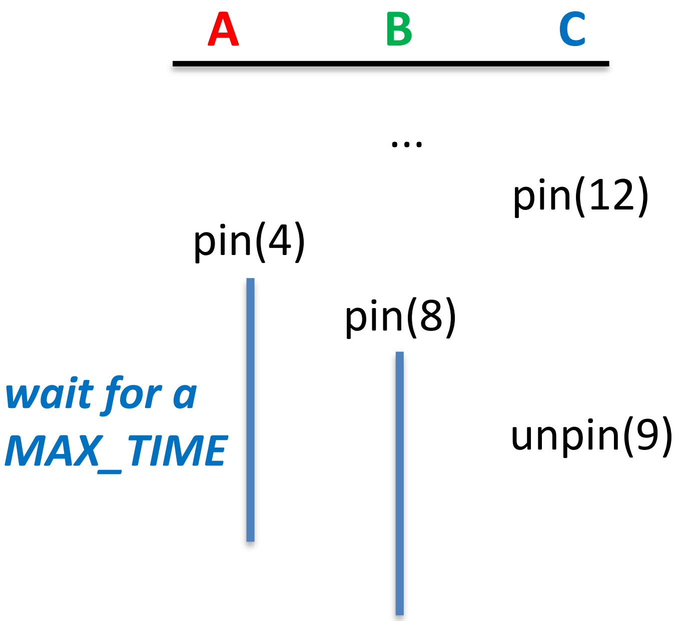
Buffer pool

1	2	3	5	6	7	9	10	11	

Waiting list

Waiting: An Example

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12



Buffer pool

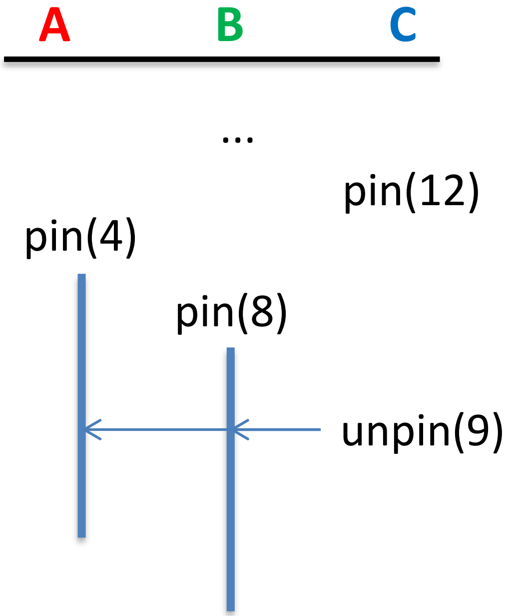
1	2	3	5	6	7	9	10	11	12

Waiting list

A	B			

Waiting: An Example

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12



Buffer pool

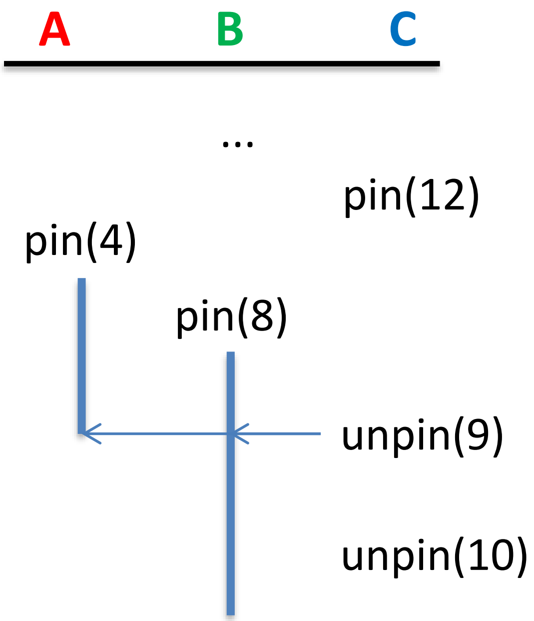
1	2	3	5	6	7		10	11	12

Waiting list

A	B			

Waiting: An Example

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12



Buffer pool

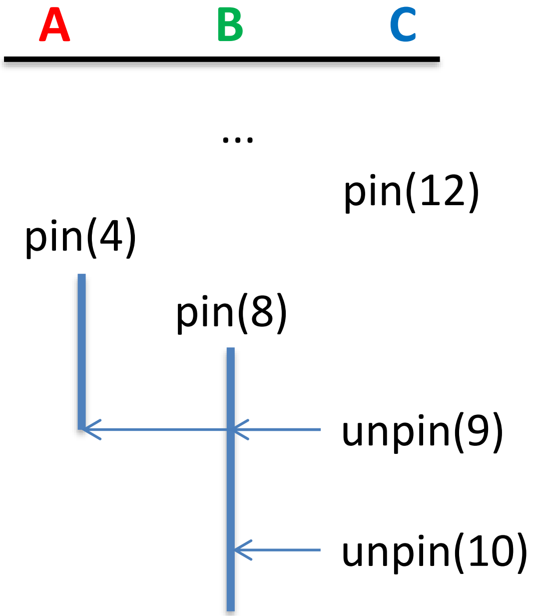
1	2	3	5	6	7	4	10	11	12

Waiting list

B				

Waiting: An Example

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12



Buffer pool

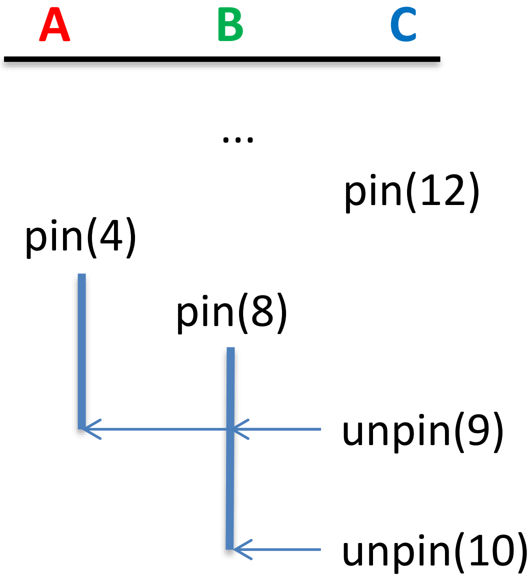
1	2	3	5	6	7	4		11	12

Waiting list

B				

Waiting: An Example

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12



Buffer pool

1	2	3	5	6	7	4	8	11	12

Waiting list

Waiting: Deadlock Case

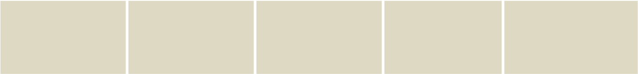
A	B	C
	…	
		pin(12)

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12, 13

Buffer pool

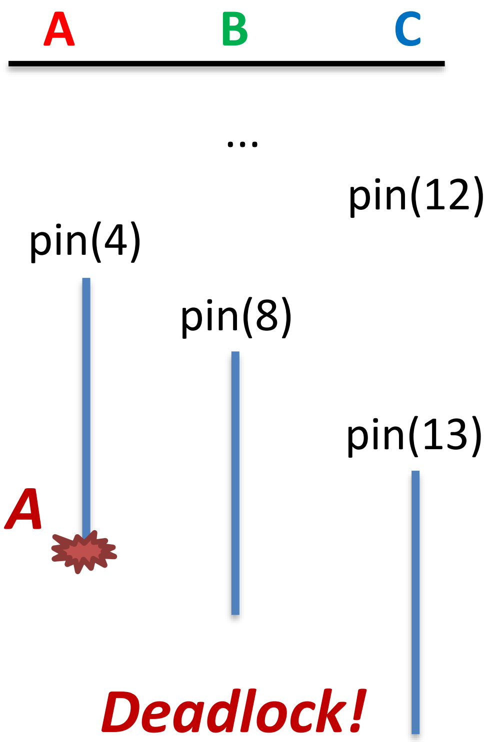
1	2	3	5	6	7	9	10	11	

Waiting list



Waiting: Deadlock Case

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12, 13



Buffer pool

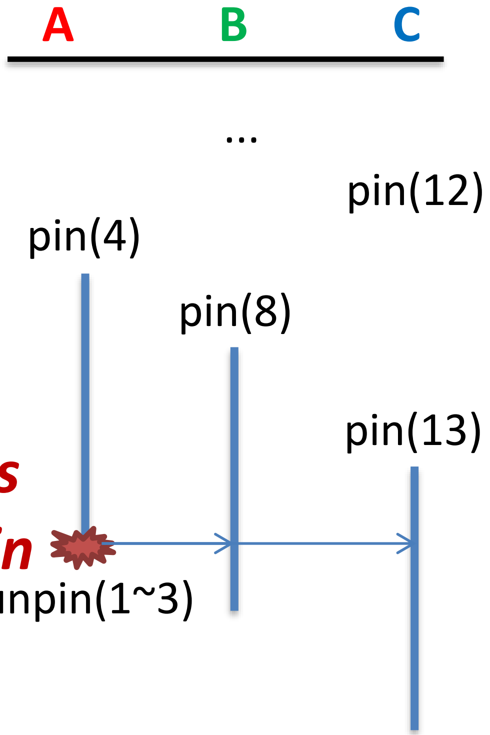
1	2	3	5	6	7	9	10	11	12

Waiting list

A	B	C		

Waiting: Deadlock Case

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12, 13



Unpin all holding pages then re-pin again

Buffer pool

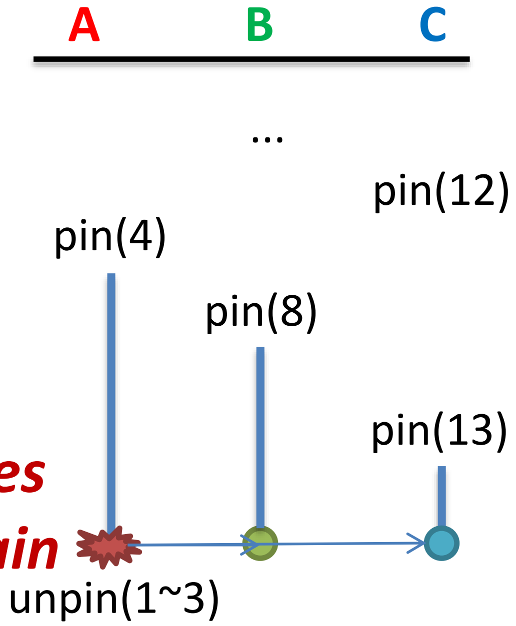
			5	6	7	9	10	11	12

Waiting list

B	C			

Waiting: Deadlock Case

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12, 13



Unpin all holding pages then re-pin again

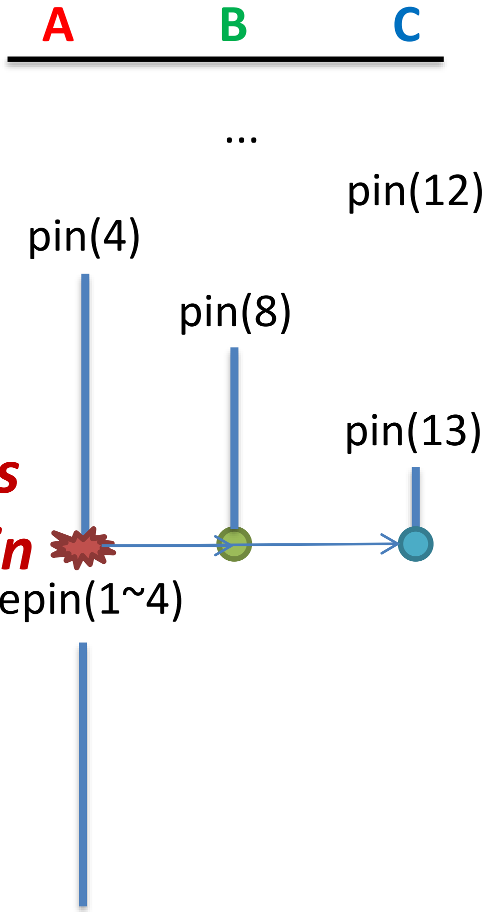
Buffer pool

13		8	5	6	7	9	10	11	12

Waiting list

Waiting: Deadlock Case

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12, 13



Unpin all holding pages then re-pin again

Buffer pool

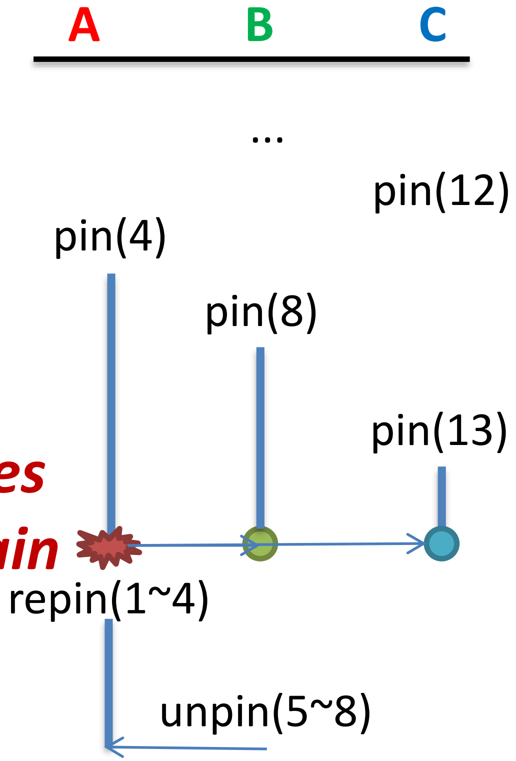
13	1	8	5	6	7	9	10	11	12

Waiting list

A				

Waiting: Deadlock Case

- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12, 13



Unpin all holding pages then re-pin again

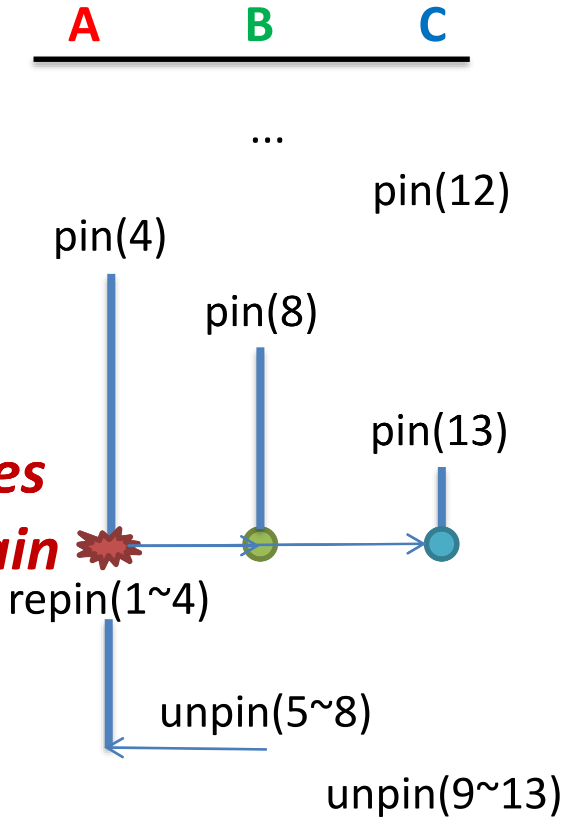
Buffer pool

13	1		2	3	4	9	10	11	12

Waiting list

Waiting: Deadlock Case

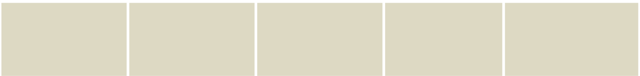
- Buffer pool size: 10
- Block access from three clients:
  - Client A: 1, 2, 3, 4
  - Client B: 5, 6, 7, 8
  - Client C: 9, 10, 11, 12, 13



Unpin all holding pages then re-pin again

Buffer pool

Waiting list



How about Self-Deadlock?

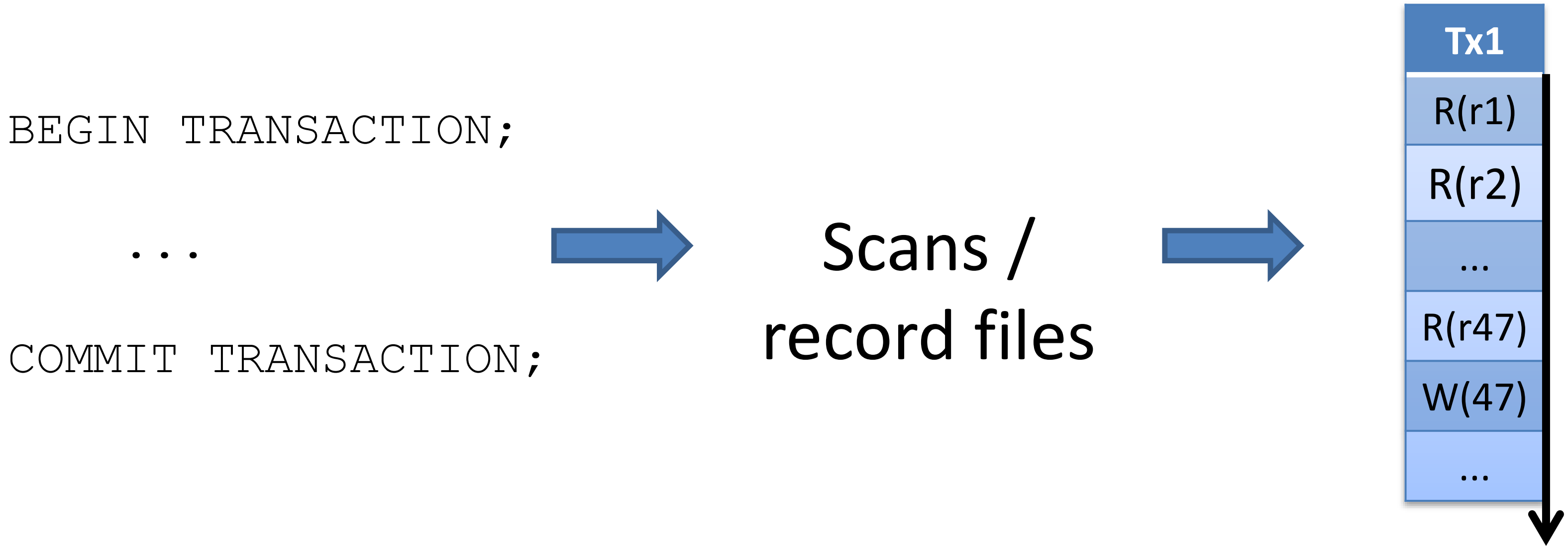
- A client that pins more blocks than a pool can hold
- Happens when
  - The pool is too small
  - The client is malicious (luckily, we write the clients/`RecordFile` ourselves)
- How to handle this?
  - A (fixed-sized) buffer manager has no choice but throwing an exception
- The pool should be large enough to at least hold the working set of a single client
- A good client should pin blocks sparingly
  - Unpins a block immediately when done
  - Call `close()` after iterating a `ResultSet` in JDBC

Outline

- Overview
- Buffering User Data
- Caching Logs
- Log Manager in VanillaCore
- Buffer Manager in VanillaCore

Why logging?

Transactions Revisited



```
BEGIN TRANSACTION;

    ...

COMMIT TRANSACTION;
```

Scans / record files

ACID

- A database ensures the ***ACID*** properties of txs
- ***Atomicity***
  - All operations in a transaction either succeed (transaction commits) or fail (transaction rollback) together
- ***Consistency***
  - After/before each transaction (which commits or rollback), your data do not violate any rule you have set
- ***Isolation***
  - Multiple transactions can run concurrently, but cannot interfere with each other
- ***Durability***
  - Once a transaction commits, any change it made lives in DB permanently (unless overridden by other transactions)

How?

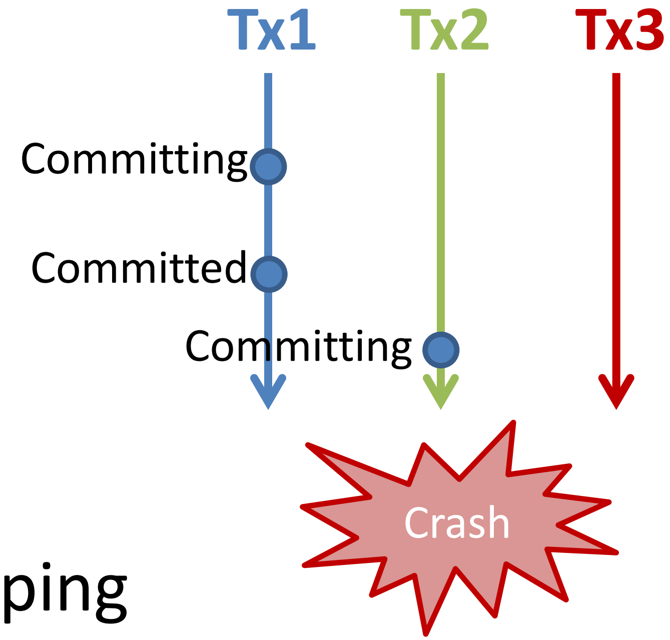
Naïve C and I

- Observation: there is no tx that accesses data across DBs
- To ensure C and I, each tx can simply lock the entire DB it belongs
    - Acquire lock at start
    - Release lock when committed or rolled back
- Txs for different DBs can execute concurrently

Naïve A and D

- D given buffers?
- ***Flush*** all dirty buffers of a tx before committing the tx
  - Return to DBMS client ***after*** tx commit

Naïve A and D



- What if system crashes and then recovers?
- To ensure A, DBMS needs to rollback uncommitted txs (2 and 3) at sart-up
  - Why 3? flushes due to swapping
- Problems:
  - How to determine which txs to rollback?
  - How to rollback all actions made by a tx?

Naïve A and D

- Idea: ***Write-Ahead-Logging*** (***WAL***)
  - Record a ***log*** of each modification made by a tx
    - E.g., <SETVAL, <TX>, <BLK>, <OFFSET>, <VAL_TYPE>, <OLD_VAL> >
    - ***In memory*** to save I/Os (discussed later)
  - To commit a tx,
    1. Write all associated logs to a log file ***before*** flushing a buffer
    2. After flushing, write a <COMMIT, <TX>> log to the log file
  - To swap a dirty buffer (in BufferMgr)
    - All logs must be flushed ***before*** flushing a user block

Naïve A and D

- Which txs to rollback?
  - Observation: txs with COMMIT logs must have flushed all their dirty blocks
  - Ans: those without COMMIT logs in the log file
- How to rollback a tx?
  - Observation: only 3 possibilities for each action ***on disk***:
    1. With log and block
    2. With log, but without block
    3. Without log and block
  - Ans: simply ***undo*** actions that are logged to disk, flush all affected blocks, and then writes a <ROLLBACK, <TX>> log
  - Applicable to self-rollback decided by a tx

Naïve A and D

- Assumption of WAL: each block-write either succeeds or fails entirely on a disk, despite power failure
  - I.e., no corrupted log block after crash
  - Modern disks usually store enough power to finish the ongoing sector-write upon power-off
  - Valid if block size == sector size or a journaling file system (e.g., EXT3/4, NTFS) is used
    - Block/physical vs. metadata/logical journals

Caching Logs

- Like user blocks, the blocks of the log file are cached
  - Each tx operation is logged ***into memory***
  - Log blocks are flushed only on
    - Tx commit
    - Buffer swapping
- Avoids excessive I/Os

Do we need a buffer pool for the log blocks?

Access Patterns: A Comparison

- User blocks
  - Of multiple files
  - Random reads, writes, and appends
  - Concurrent access by multiple worker threads (each thread per JDBC client)
- Log blocks
  - Of a ***single*** log file (why not one file per tx?)
  - Always ***appends***, by multiple worker threads
  - Always ***sequential backward reads***, by a single recovery thread at start-up

Do we need a buffer pool for the log blocks?

No! One Buffer Is Enough

- For (sequential forward) appends
  - All worker threads "pin" the tail block of the same file
  - ***Exactly one*** buffer is needed
- For the sequential backward reads
  - The recovery thread "pins" the block being read
  - There is only one recovery thread
  - ***Exactly one*** buffer is needed
- DBMS needs an additional ***log manager***
  - To implement this specialized memory management strategy for log blocks

Example API

LogMgr

<<final>> + LAST_POS : int
<<final>> + logFile : String

+ LogMgr()
<<synchronized>> + flush(lsn : long)
<<synchronized>> + iterator() : Iterator<BasicLogRecord>
<<synchronized>> + append(rec : Constant[]) : long

BasicLogRecord

+ BasicLogRecord(pg : Page, pos : int)
+ nextVal(type : Type) : Constant

- Each log record has an unique identifier called ***Log Sequence Number*** (***LSN***)
  - Typically block ID + starting position
- `flush(lsn)` flushes all log records with LSNs no larger than `lsn`

Cache Management for Read

- Provides a log iterator that iterates the log records backward from tail
- Internally, the iterator allocates a page, which always holds the block where the current log record resides
- Optimal: more pages do not help in saving I/Os

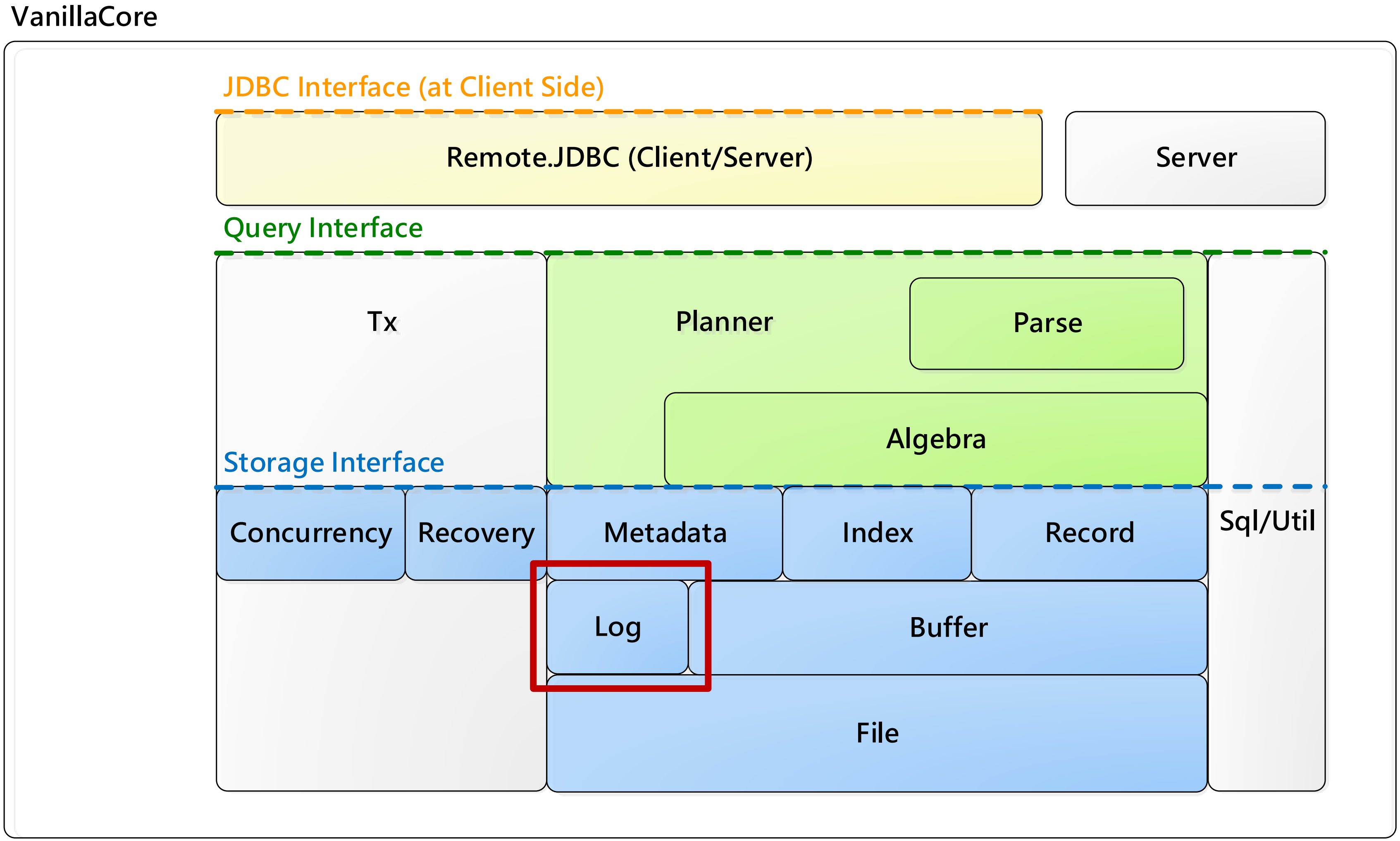
Cache Management for Append

- Permanently allocate a page, *P*, to hold the tail block of the log file
- When `append(rec)` is called:
    1. If there is no room in *P*, then write the page *P* back to disk and clear its contents
    2. Add the new log record to *P*
- When `flush(lsn)` is called:
    1. If that log record is in *P*, then write *P* to disk
    2. Else, do nothing
- Optimal: more pages do not help in saving I/Os

Outline

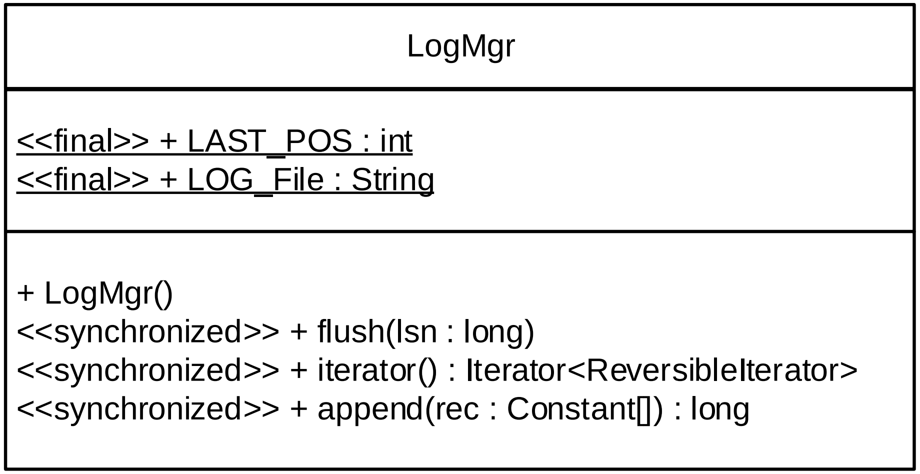
- Overview
- Buffering User Data
- Caching Logs
- Log Manager in VanillaCore
- Buffer Manager in VanillaCore

Log Manager in VanillaCore



`LogMgr`

- In `storage.log` package



LogMgr

<<final>> + LAST_POS : int
<<final>> + LOG_File : String

+ LogMgr()
<<synchronized>> + flush(lsn : long)
<<synchronized>> + iterator() : Iterator<ReversibleIterator>
<<synchronized>> + append(rec : Constant[]) : long

`LogMgr`

- Singleton
- Constructed during system startup
  - Via `VanillaDb.initFileAndLogMgr(dbname)`
- Obtained via `VanillaDb.logMgr()`
- The method `append` appends a log record to the log file, and returns the record’s LSN as long
  - No guarantee that the record will get written to disk
- A client can force a specific log record, ***and all its predecessors***, to disk by calling `flush`

LSNs

- Recall that an LSN identifies a log record
  - Typically block ID + starting position
- VanillaCore simplifies the LSN to be a ***block number***
  - Recall: block ID = file name + block number
- All log records in a block are assigned the same LSN, therefore flushed together

BasicLogRecord

- An iterator of values in an log record
- The log manager only implements the memory management strategy
  - Does ***not*** understand the contents of the log records
  - It is the ***recovery manager*** that defines the semantic of a log record

BasicLogRecord

+ BasicLogRecord(pg : Page, pos : int)
+ nextVal(Type) : Constant

`LogIterator`

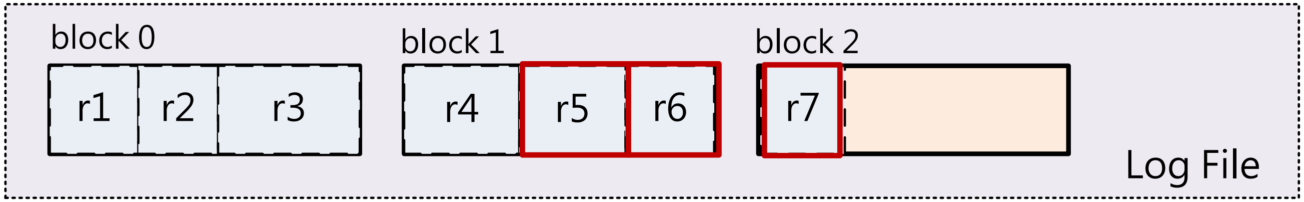
- A client can read the records in the log file by calling the method `iterator` in `LogMgr`
  - Returns a `LogIterator` instance

LogIterator

+ LogIterator(blk : BlockId)
+ hasNext() : boolean
+ next() : BasicLogRecord
+ hasPrevious() : boolean
+ previous() : BasicLogRecord
+ remove()

`LogIterator`

- Calling `next` returns the next `BasicLogRecord` in ***reverse*** order from tail
  - This is how the recovery manager wants to see the logs



Using `LogMgr`

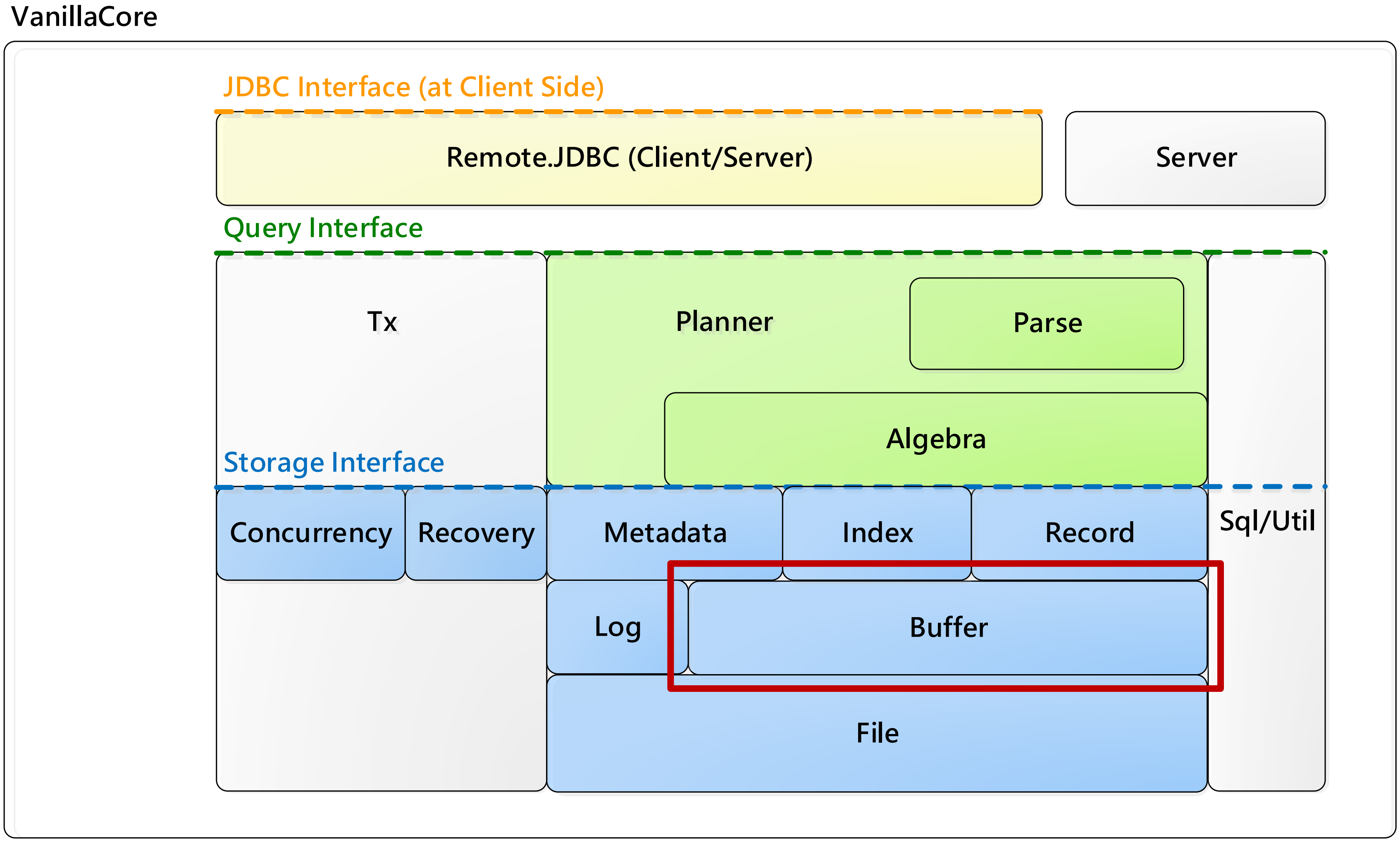
```
VanillaDb.initFileAndLogMgr("studentdb");
LogMgr logmgr = VanillaDb.logMgr();
long lsn1 = logmgr.append(new Constant[] { new IntegerConstant(1),
        new VarcharConstant("abc") });
long lsn2 = logmgr.append(new Constant[] { new IntegerConstant(2),
        new VarcharConstant("kri") });
long lsn3 = logmgr.append(new Constant[] { new IntegerConstant(3),
        new VarcharConstant("net") });
logmgr.flush(lsn3);

Iterator<BasicLogRecord> iter = logmgr.iterator();
while (iter.hasNext()) {
    BasicLogRecord rec = iter.next();
    Constant c1 = rec.nextVal(Type.INTEGER);
    Constant c2 = rec.nextVal(Type.VARCHAR);
    System.out.println("[" + c1 + ", " + c2 + "]");
}
```

```
Output:
[3, net]
[2, kri]
[1, abc]
```

Outline

- Overview
- Buffering User Data
- Caching Logs
- Log Manager in VanillaCore
- Buffer Manager in VanillaCore

Buffer Manager in VanillaCore



`BufferMgr`

- Each transaction has its own `BufferMgr`
- Constructed while creating a transaction
  - Via `transactionMgr.newTransaction(…)`
- Obtained via `transaction.bufferMgr()`

BufferMgr : TransactionLifecycleListener

<<final>> # BUFFER_POOL_SIZE : int

+ BufferMgr()
+ onTxCommit(tx : Transaction)
+ onTxRollback(tx : Transaction)
+ onTxEndStatement(tx : Transaction)
<<synchronized>> + pin(blk : BlockId)
<<synchronized>> + pinNew(filename : String, fmtr : PageFormatter) : Buffer
<<synchronized>> + unpin(buffs : Buffer[])
+ flush()
+ flushAll()
+ available() : int

`BufferMgr`

- A `BufferMgr` of a transaction takes care which buffers are pinned by the transaction and make it waiting when there is no available buffer
- `flush()` flushes each buffer modified by the specified tx
- `available()` returns the number of buffers holding unpinned buffers

`BufferPoolMgr`

- A `BufferPoolMgr` is a singleton object and it is hidden in `buffer` package to the outside world
- It manages a buffer pool for all pages and implements the ***clock*** buffer replacement strategy
  - The details of disk access is unknown to client

`Buffer`

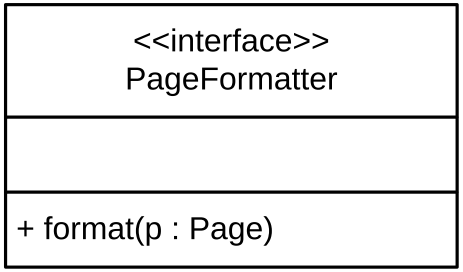
- Wraps a page and stores
  - ID of the holding block
  - Pin count
  - Modified information
  - Log information
- ***Supports WAL***
  - `setVal()` requires an LSN
    - Must be preceded by `LogMgr.append()`
  - `flush()` calls `LogMgr.flush(maxLsn)`
    - Called by `BufferMgr` upon swapping

Buffer

~ Buffer()
<<synchronized>> + getVal(offset : int, type : Type) : Constant
<<synchronized>> + setVal(offset : int, val : Constant , txnum : long, lsn : long)
<<synchronized>> + block() : BlockId
<<synchronized>> ~ flush()
<<synchronized>> ~ pin()
<<synchronized>> ~ unpin()
<<synchronized>> ~ isPinned() : boolean
<<synchronized>> ~ isModifiedBy(txNum : long) : boolean
<<synchronized>> ~ assignToBlock(b : BlockId)
<<synchronized>> ~ assignToNew (filename : String, fmtr : PageFormatter)

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



<<interface>>
PageFormatter

+ format(p : Page)

```
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);
    }
}
```

Using the Buffer Manager

- Reading value from a buffer

```
// Initialize VanillaDB ...

Transaction tx =
VanillaDb.txMgr().newTransaction(Connection.TRANSACTION_SERIALIZABLE,
                false);
BufferMgr bufferMgr = tx.bufferMgr();

BlockId blk = new BlockId("student.tbl", 0);
Buffer buff = bufferMgr.pin(blk);
Type snameType = Type.VARCHAR(20);
Constant sname = buff.getVal(46, snameType);

System.out.println(sname);

bufferMgr.unpin(buff);
```

Using the Buffer Manager

```
// Initialize VanillaDB ...

Transaction tx =
VanillaDb.txMgr().newTransaction(Connection.TRANSACTION_SERIALIZABLE,
false);
BufferMgr bufferMgr = tx.bufferMgr();
LogMgr logMgr = VanillaDb.logMgr();
long myTxnNum = 1;

BlockId blk = new BlockId("student.tbl", 0);
Buffer buff = bufferMgr.pin(blk);
Type snameType = Type.VARCHAR(20);
Constant sname = buff.getVal(46, snameType);
Constant[] logRec = new Constant[] { new BigIntConstant(myTxnNum), new
VarcharConstant("student.tbl"),
new BigIntConstant(blk.number()), new IntegerConstant(46), sname };

long lsn = logMgr.append(logRec);
buff.setVal(46, new VarcharConstant("kay").castTo(snameType), myTxnNum,
lsn);
bufferMgr.unpin(buff);

// [WAL] when buff.flush() is called due to swapping or tx commit,
// logMgr.flush(lsn) is called first by buff
```

- Writing value into a buffer

You Have Assignment!

Assignment 3 (1/2)

- The current File and Buffer Managers are slow
  - Mainly due to the synchronization (for thread-safety)
- ***Optimize them*** to improve performance

- We provide you a basic implementation of these modules which have bad performance
  - You need to modify them to reach higher throughput or lower latency in our benchmark
- You need to come out at least one optimization for each module
  - `storage.file`
  - `storage.buffer`

Assignment 3 (2/2)

- Use the micro-benchmark we provided to compare performance between the basic and your implementation

- Report
  - Show the throughput and latency of benchmark between those implementation
  - Show what you exactly do for optimization
  - Discuss why your optimization works

Hints

- Critical sections are used in File/BufferManagers to protect some shared resource
- ***Breaks*** a single critical section into multiple ones
  - Avoids unnecessary transaction/thread stalls
- ***Reduce the coverage*** of critical sections
  - reduce the stall duration

References

- W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton, The oracle universal server buffer, *VLDB*, 1997.
- M. Cyran., Oracle Database Concepts, 10g Release 2 (10.2), 2005.
- Edward Sciore., *Database Design and Implementation*, chapter 13.
- Hellerstein, J. M., Stonebraker, M., and Hamilton, J. Architecture of a database system. *Foundations and Trends in Databases 1*, 2, 2007.
- Hussein M. Abdel-Wahab, CS 471 – Operating Systems Slides, http://www.cs.odu.edu/~cs471w/