# データベースの仕組みを独自実装して理解してみよう

Yoshisaur

# 自己紹介

名前: Yoshisaur(ヨシザウルス)

大学: 琉球大学工学部 B4

学科: 知能情報コース

研究室: 並列信頼研

X(Twitter): @ie_Yoshisaur

趣味/特技: 英話, 将棋

# 研究室紹介

名前: 並列信頼研


研究分野: 低レイヤー

・独自言語 -> Continuation based C

・独自OS -> GearsOS

# 発表の目的

・データベースの内部実装の知識の共有

・データベースを自作して得られた経験の共有

・チャレンジの題材としてのDB自作を布教

# データベースを自作しようとしたきっかけ

・イノシシ本を理解したかった

・データの量や複雑さ、変化が課題となるアプリケーションを信頼性、拡張性、保守性を守りながらデザインするための技術書

・600ページ以上あり、内容も高度で当時の自分では理解ができなかった


データ指向アプリケーションデザイン

# データベースを自作しようとしたきっかけ

・イノシシ本の内容にデータベース内部の実装の話が多く載せられていた

- ・2章　データモデルとクエリ言語
- ・3章　ストレージと抽出
- ・7章　トランザクション

・「自作DBを経験すれば、イノシシ本を読めるようになる!」と思い、自作DBを始めた


データ指向アプリケーションデザイン

# 自作DBにおける目標

・Rustを使う

・並行処理を含むトランザクション

・SQL対応

・1ヶ月で0から完成させる

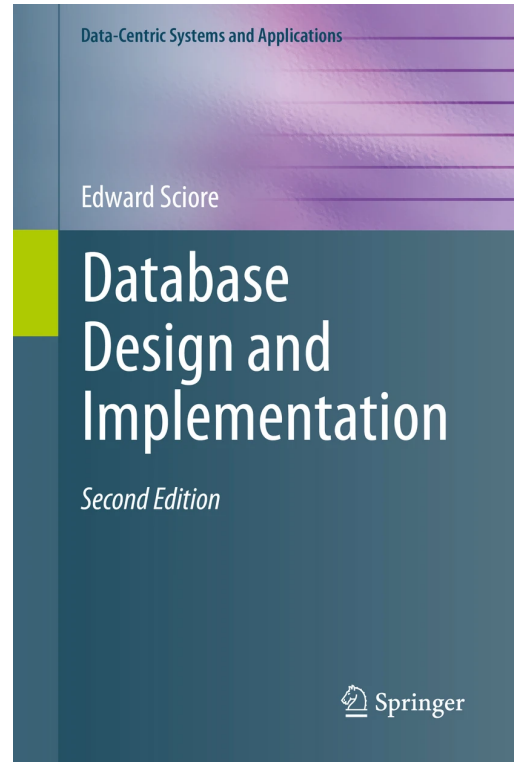
・コードを大量に書く経験を得る

・自分だけのクールな名前のソフトを作る → OxideDB

# 参考にするDB

・Database Design and Implementationという本でJavaで実装されているSimpleDBを参考にする

・Rust特有の言語ルールに適応させて再実装



# 設計方針

・扱える属性が4バイトのIntegerと可変長のVarchar(ASCII)

・ロックの単位はブロック(後に解説)

・SQLを処理する

・リカバリー方式はundo(後に解説)

# DB実装時に直面した問題点

・参考にしたSimpleDBでは循環参照を含む実装だったが、Rustでそれを実現しようとすると参照カウンタが機能しなくなり、メモリーリークが起きる

・可変シングルトンがRustで実装できない

・デッドロック検出器をせずにデバッグに苦労した

# DB実装時に直面した問題点

・参考にしたSimpleDBでは循環参照を含む実装だったが、Rustでそれを実現しようとすると参照カウンタが機能しなくなり、メモリーリークが起きる

・可変シングルトンがRustで実装できない

・デッドロック検出器をせずにデバッグに苦労した

あとで解説します

# DB実装のモジュール構成

- file - Diskにブロック単位で読み書き
- buffer - ブロックのバッファの管理
- log - トランザクションのログの読み書き
- transaction - リカバリーや並行性制御
- metadata - テーブルのスキーマやインデックスの情報の管理
- record - ディスク上のレコードデータの位置の管理
- query - 関係代数クエリの実行
- parse - SQLの解析
- plan - parseで解析したデータを関係代数クエリツリーに変換
- index - BTreeを用いたレコードの特定
- optimizer - 関係代数クエリを最適化する

# 解説するDB実装の詳細

- file
- buffer
- log
- transaction

# file

・DBのとして最低限求められる機能

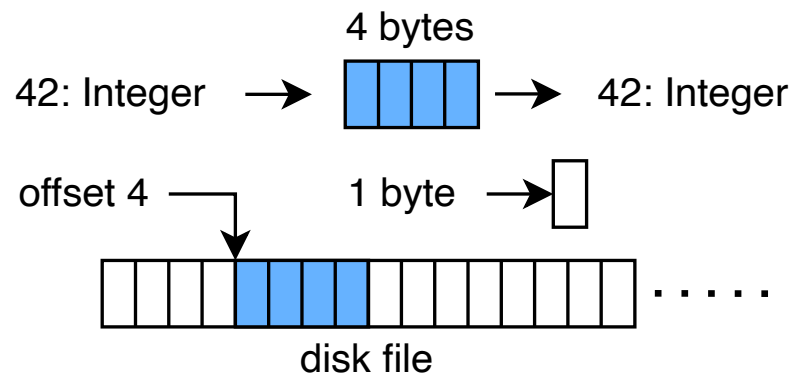
  ・ディスクにデータをwriteする

  ・ディスクからデータをreadする

# file

## 整数のwrite/read

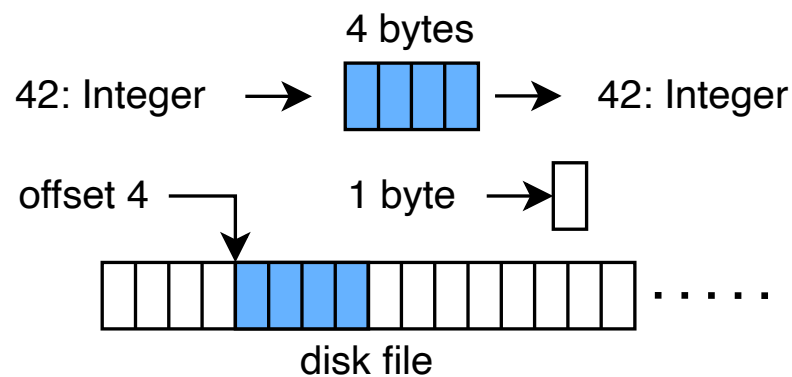
```
pub fn get_int(&mut self, offset: usize) -> Result<i32, Pa
    self.byte_buffer.set_position(offset as u64);
    let mut bytes = [0; I32_SIZE];
    self.byte_buffer
        .read_exact(&mut bytes)
        .map_err(PageError::IoError)?;
    Ok(i32::from_le_bytes(bytes))
}

pub fn set_int(&mut self, offset: usize, value: i32) -> Re
    self.byte_buffer.set_position(offset as u64);
    self.byte_buffer
        .write_all(&value.to_le_bytes())
        .map_err(PageError::IoError)
}
```



# file

## 整数のwrite/read

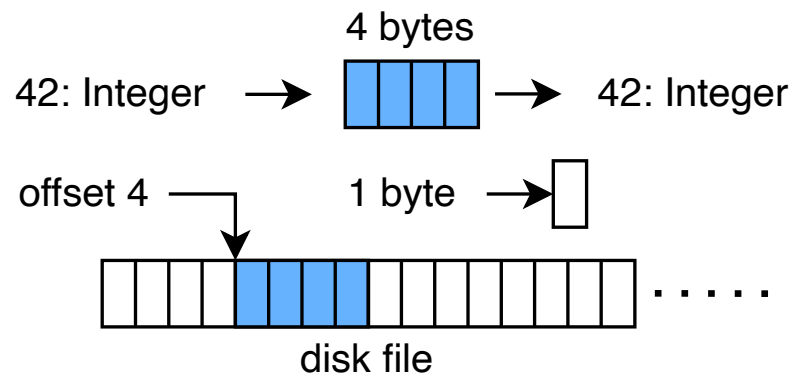
```
pub fn get_int(&mut self, offset: usize) -> Result<i32, Pa
    self.byte_buffer.set_position(offset as u64);
    let mut bytes = [0; I32_SIZE];
    self.byte_buffer
        .read_exact(&mut bytes)
        .map_err(PageError::IoError)?;
    Ok(i32::from_le_bytes(bytes))
}

pub fn set_int(&mut self, offset: usize, value: i32) -> Re
    self.byte_buffer.set_position(offset as u64);
    self.byte_buffer
        .write_all(&value.to_le_bytes())
        .map_err(PageError::IoError)
}
```



# file

## 整数のwrite/read

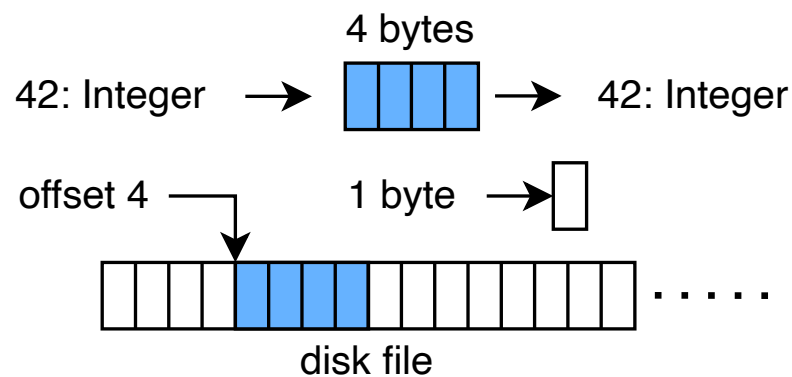
```
pub fn get_int(&mut self, offset: usize) -> Result<i32, Pa
    self.byte_buffer.set_position(offset as u64);
    let mut bytes = [0; I32_SIZE];
    self.byte_buffer
        .read_exact(&mut bytes)
        .map_err(PageError::IoError)?;
    Ok(i32::from_le_bytes(bytes))
}

pub fn set_int(&mut self, offset: usize, value: i32) -> Re
    self.byte_buffer.set_position(offset as u64);
    self.byte_buffer
        .write_all(&value.to_le_bytes())
        .map_err(PageError::IoError)
}
```



# file

## 整数のwrite/read

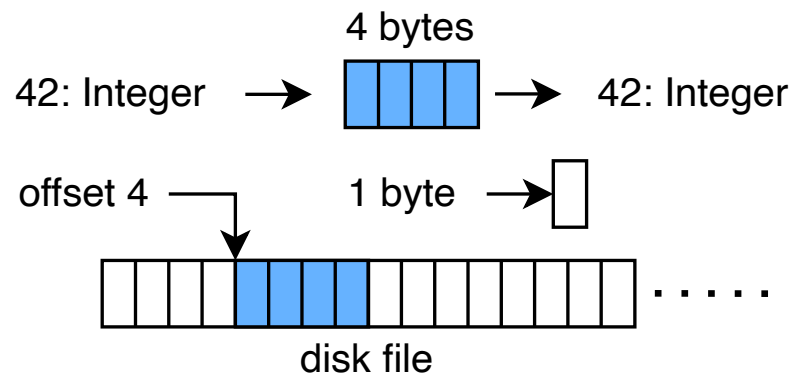
```
pub fn get_int(&mut self, offset: usize) -> Result<i32, Pa
    self.byte_buffer.set_position(offset as u64);
    let mut bytes = [0; I32_SIZE];
    self.byte_buffer
        .read_exact(&mut bytes)
        .map_err(PageError::IoError)?;
    Ok(i32::from_le_bytes(bytes))
}

pub fn set_int(&mut self, offset: usize, value: i32) -> Re
    self.byte_buffer.set_position(offset as u64);
    self.byte_buffer
        .write_all(&value.to_le_bytes())
        .map_err(PageError::IoError)
}
```



# file

## 整数のwrite/read

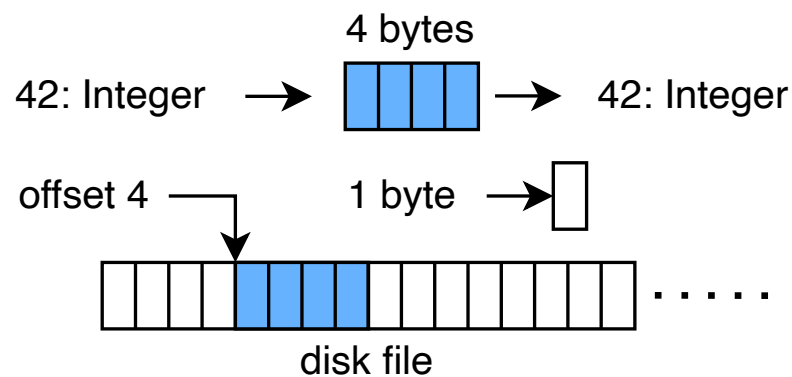
```
pub fn get_int(&mut self, offset: usize) -> Result<i32, Pa
    self.byte_buffer.set_position(offset as u64);
    let mut bytes = [0; I32_SIZE];
    self.byte_buffer
        .read_exact(&mut bytes)
        .map_err(PageError::IoError)?;
    Ok(i32::from_le_bytes(bytes))
}

pub fn set_int(&mut self, offset: usize, value: i32) -> Re
    self.byte_buffer.set_position(offset as u64);
    self.byte_buffer
        .write_all(&value.to_le_bytes())
        .map_err(PageError::IoError)
}
```



# file

## 整数のwrite/read

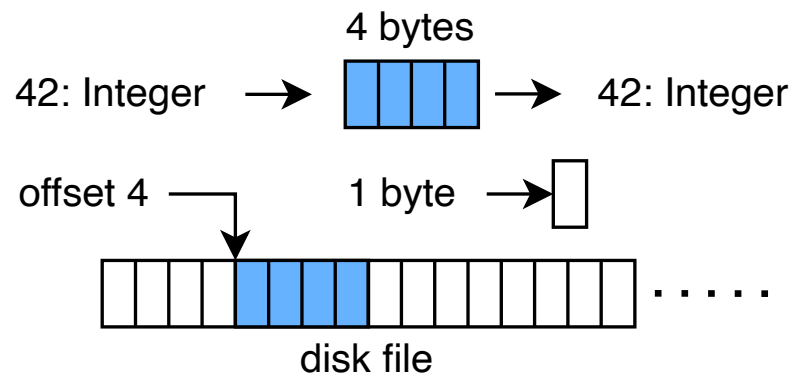
```
pub fn get_int(&mut self, offset: usize) -> Result<i32, Pa
    self.byte_buffer.set_position(offset as u64);
    let mut bytes = [0; I32_SIZE];
    self.byte_buffer
        .read_exact(&mut bytes)
        .map_err(PageError::IoError)?;
    Ok(i32::from_le_bytes(bytes))
}

pub fn set_int(&mut self, offset: usize, value: i32) -> Re
    self.byte_buffer.set_position(offset as u64);
    self.byte_buffer
        .write_all(&value.to_le_bytes())
        .map_err(PageError::IoError)
}
```



# file

## 整数のwrite/read

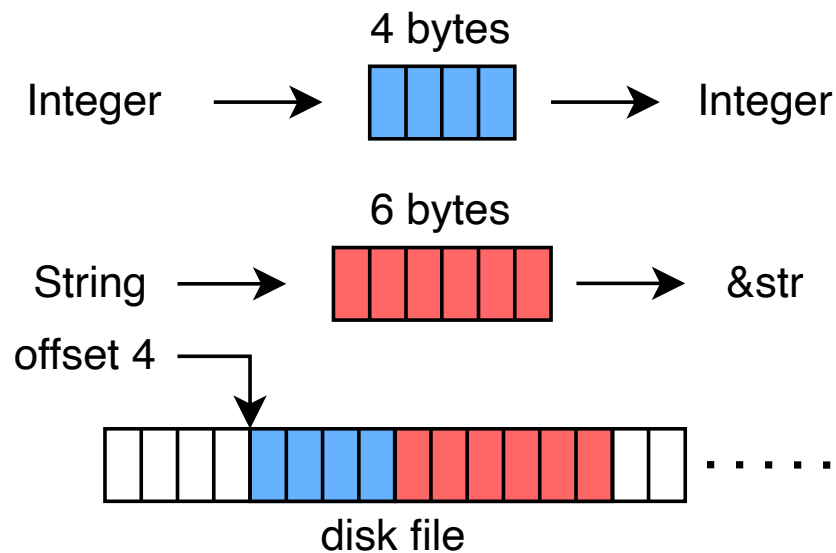
```
pub fn get_int(&mut self, offset: usize) -> Result<i32, Pa
    self.byte_buffer.set_position(offset as u64);
    let mut bytes = [0; I32_SIZE];
    self.byte_buffer
        .read_exact(&mut bytes)
        .map_err(PageError::IoError)?;
    Ok(i32::from_le_bytes(bytes))
}

pub fn set_int(&mut self, offset: usize, value: i32) -> Re
    self.byte_buffer.set_position(offset as u64);
    self.byte_buffer
        .write_all(&value.to_le_bytes())
        .map_err(PageError::IoError)
}
```



# file

可変長文字列のread/write

可変長なデータは、
最初のoffsetにデータの
バイト長を整数で書き込む



# file

「ブロック」について

・整数や可変長文字列がオフセットの位置に制約なく書き込むようにしたら実装が難しい

・ディスクファイル内を等間隔で区分けしたほうが実装が楽

・ディスク上の区分けの単位を「ブロック」という

  ・ブロックのサイズはしばしば4096 bytes

# file

## なぜブロックサイズが4096 bytesか?

- OSのファイルシステムでも「ブロック」が存在
  - 例えばLinuxのファイルシステム ext4も4096 bytesのブロック
  - 4096 bytesからずれると、読み書き時にオーバーヘッドが生じる
  - DBのブロックが4097 bytesのサイズだと、
    ファイルシステム側で8192 bytesを読み書きすることになる

# buffer

・「file」でディスクに対して整数と文字列を読み書きできるようになった! Happy!

・メモリー上にブロックのバッファを作ることで読み書きを高速化する

# buffer

・「file」でディスクに対して整数と文字列を読み書きできるようになった! Happy!

　・とはならない

・メモリー上にブロックのバッファを作ることで読み書きを高速化する

# buffer

・「file」でディスクに対して整数と文字列を読み書きできるようになった! Happy!

  ・とはならない
  ・Disk I/Oは遅すぎる

・メモリー上にブロックのバッファを作ることで読み書きを高速化する

# buffer

・「file」でディスクに対して整数と文字列を読み書きできるようになった! Happy!

- ・とはならない
- ・Disk I/Oは遅すぎる
- ・HDDは大半が7200rpm(毎秒120回転)、データの読み書きが最大120回

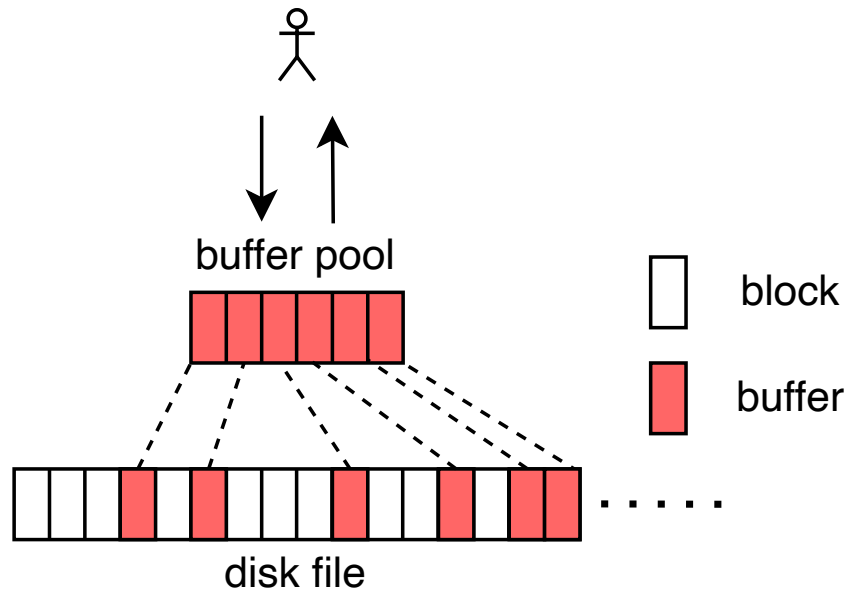
・メモリー上にブロックのバッファを作ることで読み書きを高速化する

# buffer

・バッファを経由してデータの読み書きを行う

・ユーザーからはデータの読み書きが高速化したように見える

・happy!



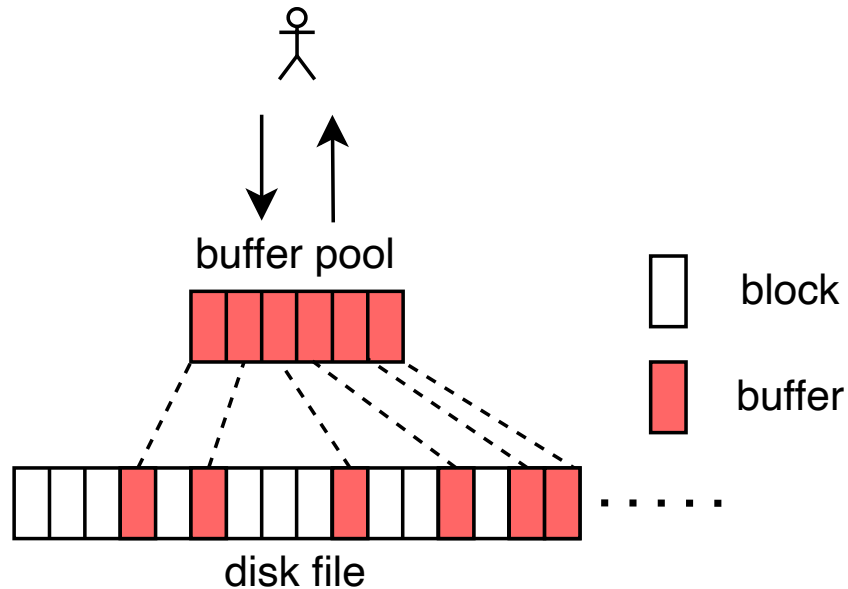
# buffer

・バッファを経由してデータの読み書きを行う

・ユーザーからはデータの読み書きが高速化したように見える

・happy!

  ・とはならない

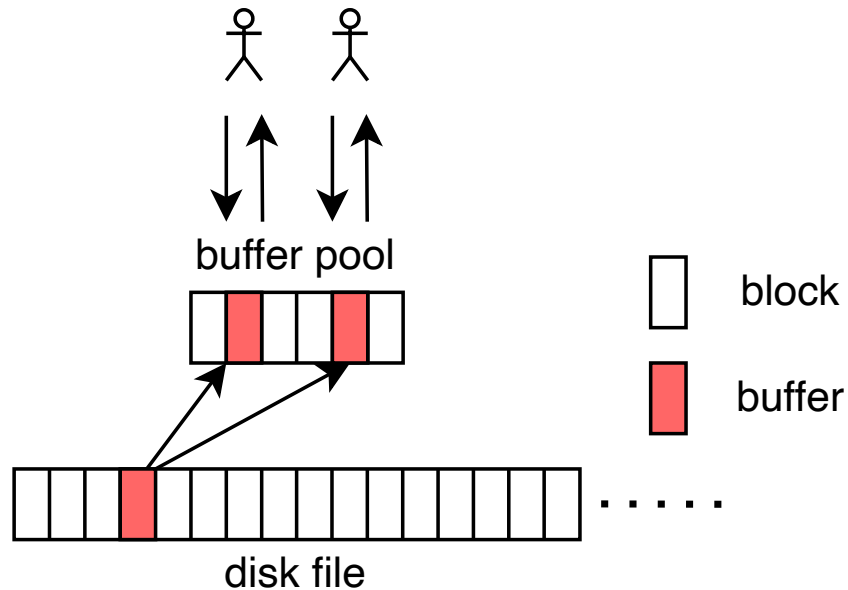


# buffer

・複数のスレッドがバッファにアクセスして読み書きができる場合は、データの整合性がカオスになる

・実装的には単一のブロックに対して2つ以上のバッファが割り振られることも可能

・「transaction」がこの問題を解決する(後に解説)



# log

・ログファイルにログ(レコード)を読み書きする

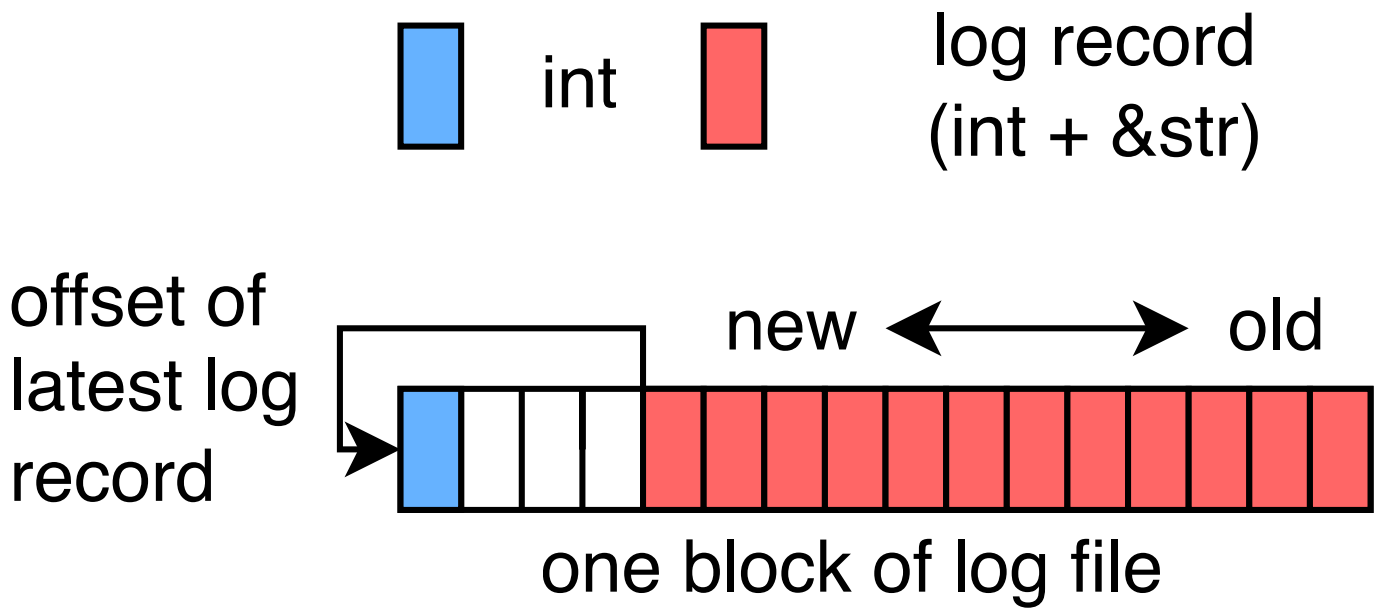
・DBがクラッシュした場合のリカバリーとトランザクションのロールバックに使われる
  ・ロールバックはトランザクションがデータ操作を永続化(コミットという)しないで、キャンセルすること

・リカバリーする際は、最新のログレコードから読み込んでundoするので、ファイル内のログレコードを逆順に回すイテレータも実装している

# log

## logイテレータの仕組み

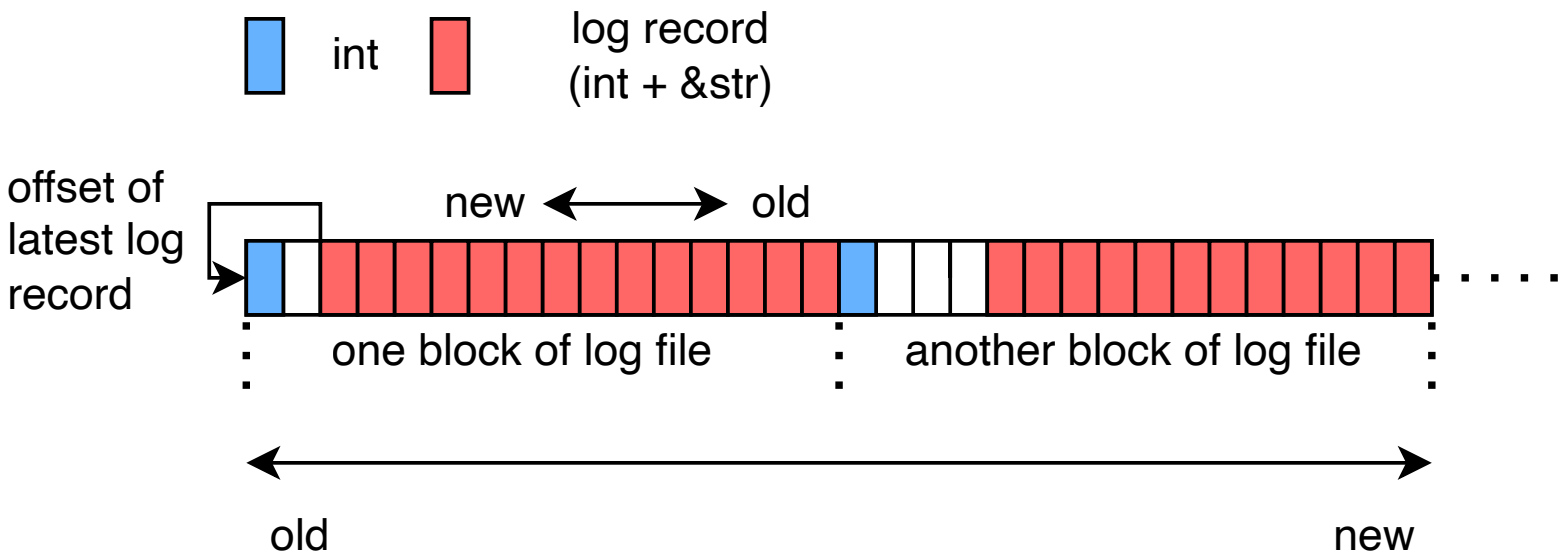
ブロック内の先頭側のブロックは新しいレコードを持つ



# log

## logイテレータの仕組み

ファイルの先頭側のブロックは古いレコードを持つ



# transaction

transactionの役割が2つある

・リカバリー

・DBのクラッシュ時やトランザクションのロールバックを行う際に中途半端なデータの変更を取り消してデータの整合性を保つ

・並行性制御

・同時に同一のデータに複数の読み書きが行われても矛盾が起きない順序でデータを操作してデータの整合静を保つ

# transaction - recovery

transactionはリカバリーのためにログレコードを書く

・transactionがDB(disk file)に何らかの操作を加えるときにログレコードをログファイルに書き込む

<START>, <COMMIT>, <ROLLBACK>, <CHECKPOINT>,
<SETINT transaction_id block_id offset value>,
<SETSTRING transaction_id block_id offset value>,

・リカバリーでは<CHECKPOINT>にたどり着くまで最新のレコードログから辿ってデータの変更の操作をundoし続ける

# transaction - recovery

クラッシュ時のリカバリーのシュミレーション

クラッシュ時のリカバリーでは、ディスクに永続化されたコミットされていないデータ操作をundoする

CHECKPOINT前にCOMMITしていないトランザクションはundoされる

```
1. START 1
2. SETINT 1 1 10 1234
3. SETSTRING 1 2 20 "Hello"
4. COMMT 1
5. START 2
6. SETINT 2 3 30 5678
7. SETSTRING 2 4 40 "World"
8. ROLLBACK 2
9. START 3
10. SETINT 3 5 50 91011
11. SETSTRING 3 6 60 "Database"
12. CHECKPOINT
13. START 4
14. SETINT 4 7 70 1213
15. SETSTRING 4 8 80 "Recovery"
```

# transaction - recovery

クラッシュ時のリカバリーのシュミレーション

クラッシュ時のリカバリーでは、ディスクに永続化されたコミットされていないデータ操作をundoする

CHECKPOINT前にCOMMITしていないトランザクションはundoされる

```
1. START 1
2. SETINT 1 1 10 1234
3. SETSTRING 1 2 20 "Hello"
4. COMMT 1
5. START 2
6. SETINT 2 3 30 5678
7. SETSTRING 2 4 40 "World"
8. ROLLBACK 2
9. START 3
10. SETINT 3 5 50 91011
11. SETSTRING 3 6 60 "Database"
12. CHECKPOINT
13. START 4
14. SETINT 4 7 70 1213
15. SETSTRING 4 8 80 "Recovery"
```

# transaction - recovery

クラッシュ時のリカバリーのシュミレーション

クラッシュ時のリカバリーでは、ディスクに永続化されたコミットされていないデータ操作をundoする

CHECKPOINT前にCOMMITしていないトランザクションはundoされる

```
1. START 1
2. SETINT 1 1 10 1234
3. SETSTRING 1 2 20 "Hello"
4. COMMT 1
5. START 2
6. SETINT 2 3 30 5678
7. SETSTRING 2 4 40 "World"
8. ROLLBACK 2
9. START 3
10. SETINT 3 5 50 91011
11. SETSTRING 3 6 60 "Database"
12. CHECKPOINT
13. START 4
14. SETINT 4 7 70 1213
15. SETSTRING 4 8 80 "Recovery"
```

# transaction - recovery

クラッシュ時のリカバリーのシュミレーション

クラッシュ時のリカバリーでは、ディスクに永続化されたコミットされていないデータ操作をundoする

CHECKPOINT前にCOMMITしていないトランザクションはundoされる

```
1. START 1
2. SETINT 1 1 10 1234
3. SETSTRING 1 2 20 "Hello"
4. COMMT 1
5. START 2
6. SETINT 2 3 30 5678
7. SETSTRING 2 4 40 "World"
8. ROLLBACK 2
9. START 3
10. SETINT 3 5 50 91011
11. SETSTRING 3 6 60 "Database"
12. CHECKPOINT
13. START 4
14. SETINT 4 7 70 1213
15. SETSTRING 4 8 80 "Recovery"
```

# transaction - recovery

## Rustで実装するときの注意点 - 循環参照

SimpleDBの実装ではtransactionとrecoveryを行うRecoveryMgrがお互いに参照を持っている

```
public class Transaction {
   private static int nextTxNum = 0;
   private static final int END_OF_FILE = -1;
   private RecoveryMgr    recoveryMgr;
   private ConcurrencyMgr concurMgr;
   private BufferMgr bm;
   private FileMgr fm;
   private int txnum;
   private BufferList mybuffers;
}
```

```
public class RecoveryMgr {
   private LogMgr lm;
   private BufferMgr bm;
   private Transaction tx;
   private int txnum;
}
```

# transaction - recovery

## Rustで実装するときの注意点 - 循環参照

SimpleDBの実装ではtransactionとrecoveryを行うRecoveryMgrがお互いに参照を持っている
-> Rustだとアンチパターン、RustはGCがない代わりに参照カウンタが0になったら解放する仕組みで、循環参照を実装すると永遠に解放されないままメモリリークする

```
public class Transaction {
   private static int nextTxNum = 0;
   private static final int END_OF_FILE = -1;
   private RecoveryMgr    recoveryMgr;
   private ConcurrencyMgr concurMgr;
   private BufferMgr bm;
   private FileMgr fm;
   private int txnum;
   private BufferList mybuffers;
}
```

```
public class RecoveryMgr {
   private LogMgr lm;
   private BufferMgr bm;
   private Transaction tx;
   private int txnum;
}
```

# transaction - recovery

## Rustで実装するときの注意点 - 循環参照

SimpleDBの実装ではtransactionとrecoveryを行うRecoveryMgrがお互いに参照を持っている
-> Rustだとアンチパターン、RustはGCがない代わりに参照カウンタが0になったら解放する仕組みで、循環参照を実装すると永遠に解放されないままメモリリークする

```
public class Transaction {
   private static int nextTxNum = 0;
   private static final int END_OF_FILE = -1;
   private RecoveryMgr    recoveryMgr;
   private ConcurrencyMgr concurMgr;
   private BufferMgr bm;
   private FileMgr fm;
   private int txnum;
   private BufferList mybuffers;
}
```

```
public class RecoveryMgr {
   private LogMgr lm;
   private BufferMgr bm;
   private Transaction tx;
   private int txnum;
}
```

# transaction - recovery

## Rustで実装するときの注意点 - 循環参照

SimpleDBの実装ではtransactionとrecoveryを行うRecoveryMgrがお互いに参照を持っている
-> Rustだとアンチパターン、RustはGCがない代わりに参照カウンタが0になったら解放する仕組みで、循環参照を実装すると永遠に解放されないままメモリリークする

```
public class Transaction {
   private static int nextTxNum = 0;
   private static final int END_OF_FILE = -1;
   private RecoveryMgr    recoveryMgr;
   private ConcurrencyMgr concurMgr;
   private BufferMgr bm;
   private FileMgr fm;
   private int txnum;
   private BufferList mybuffers;
}
```

```
public class RecoveryMgr {
   private LogMgr lm;
   private BufferMgr bm;
   private Transaction tx;
   private int txnum;
}
```

# transaction - recovery

## Rustの実装

```
pub struct Transaction {
    file_manager: Arc<Mutex<FileManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    log_manager: Arc<Mutex<LogManager>>,
    transaction_number: i32,
    concurrency_manager: Arc<Mutex<ConcurrencyManager>>,
    buffer_list: Arc<Mutex<BufferList>>,
    recovery_manager: Arc<Mutex<RecoveryManager>>,
}
```

```
pub struct RecoveryManager {
    log_manager: Arc<Mutex<LogManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    transaction_number: i32,
}

pub fn rollback(&self, transaction: &mut Transaction) -> R
    //省略
}

pub fn recover(&self, transaction: &mut Transaction) -> Re
    //省略
}
```

# transaction - recovery

## Rustの実装

```
pub struct Transaction {
    file_manager: Arc<Mutex<FileManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    log_manager: Arc<Mutex<LogManager>>,
    transaction_number: i32,
    concurrency_manager: Arc<Mutex<ConcurrencyManager>>,
    buffer_list: Arc<Mutex<BufferList>>,
    recovery_manager: Arc<Mutex<RecoveryManager>>,
}
```

```
pub struct RecoveryManager {
    log_manager: Arc<Mutex<LogManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    transaction_number: i32,
}

pub fn rollback(&self, transaction: &mut Transaction) -> R
    //省略
}

pub fn recover(&self, transaction: &mut Transaction) -> Re
    //省略
}
```

# transaction - recovery

## Rustの実装

```rust
pub struct Transaction {
    file_manager: Arc<Mutex<FileManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    log_manager: Arc<Mutex<LogManager>>,
    transaction_number: i32,
    concurrency_manager: Arc<Mutex<ConcurrencyManager>>,
    buffer_list: Arc<Mutex<BufferList>>,
    recovery_manager: Arc<Mutex<RecoveryManager>>,
}
```

```rust
pub struct RecoveryManager {
    log_manager: Arc<Mutex<LogManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    transaction_number: i32,
}

pub fn rollback(&self, transaction: &mut Transaction) -> R
    //省略
}

pub fn recover(&self, transaction: &mut Transaction) -> Re
    //省略
}
```

# transaction - recovery

## Rustの実装

```
pub struct Transaction {
    file_manager: Arc<Mutex<FileManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    log_manager: Arc<Mutex<LogManager>>,
    transaction_number: i32,
    concurrency_manager: Arc<Mutex<ConcurrencyManager>>,
    buffer_list: Arc<Mutex<BufferList>>,
    recovery_manager: Arc<Mutex<RecoveryManager>>,
}
```

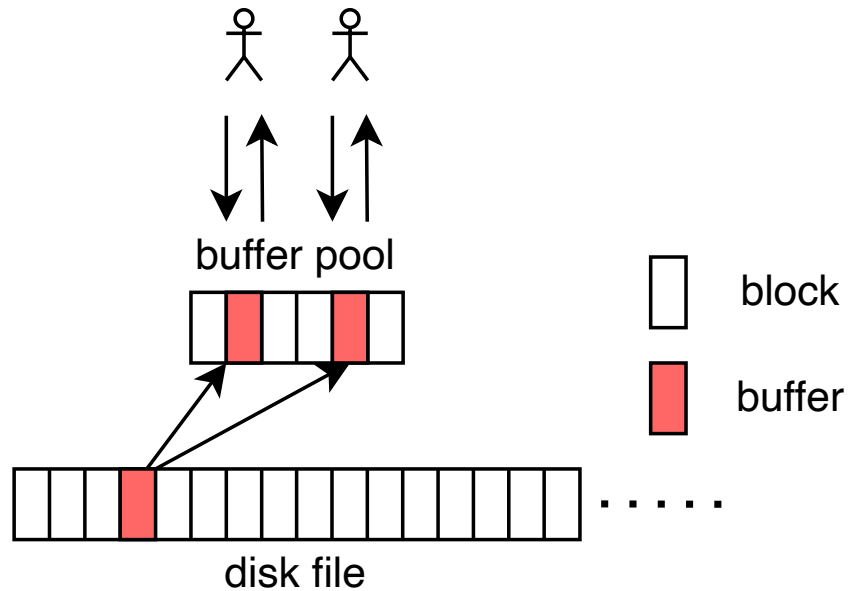
```
pub struct RecoveryManager {
    log_manager: Arc<Mutex<LogManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    transaction_number: i32,
}

pub fn rollback(&self, transaction: &mut Transaction) -> R
    //省略
}

pub fn recover(&self, transaction: &mut Transaction) -> Re
    //省略
}
```

# transaction - concurrency

・実装的には単一のブロックに対して2つ以上のバッファが割り振られることも可能

・書き込んだデータに矛盾が生じてしまう

・矛盾したデータを保存するDBに意味はなくなる

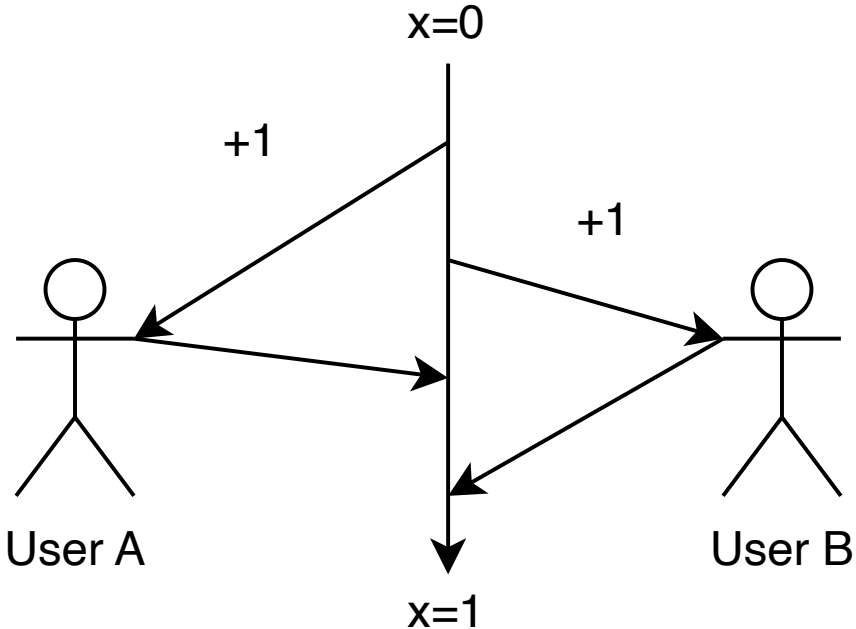


# transaction - concurrency

・データが壊れるシナリオ

・データに対して、2人以上の人がデータを読んでる状態で書き込みを行う

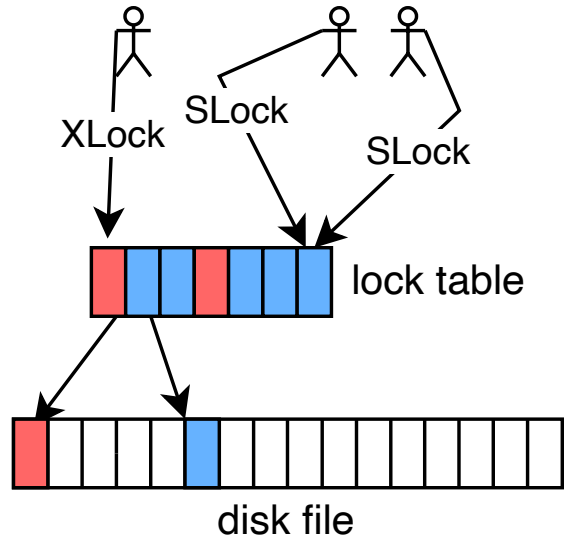
・xは2人によってインクリメントされ2になるはずだが、読み込みと書き込みの順序によってはこのような矛盾が生じる



# transaction - concurrency

・lock_tableがブロックにかかっているロックの数を管理

・locksがブロックにかかっているロックの種類を管理



# transaction - concurrency

・lock_tableがブロックにかかっているロックの数を管理

・locksがブロックにかかっているロックの種類を管理

```
enum LockType {
    Shared,
    Exclusive,
}

pub struct ConcurrencyManager {
    lock_table: Arc<Mutex<LockTable>>,
    locks: HashMap<BlockId, LockType>,
}

pub struct LockTable {
    locks: HashMap<BlockId, i32>,
}
```

# transaction - concurrency

・lock_tableがブロックにかかっているロックの数を管理

・locksがブロックにかかっているロックの種類を管理

```
enum LockType {
    Shared,
    Exclusive,
}

pub struct ConcurrencyManager {
    lock_table: Arc<Mutex<LockTable>>,
    locks: HashMap<BlockId, LockType>,
}

pub struct LockTable {
    locks: HashMap<BlockId, i32>,
}
```

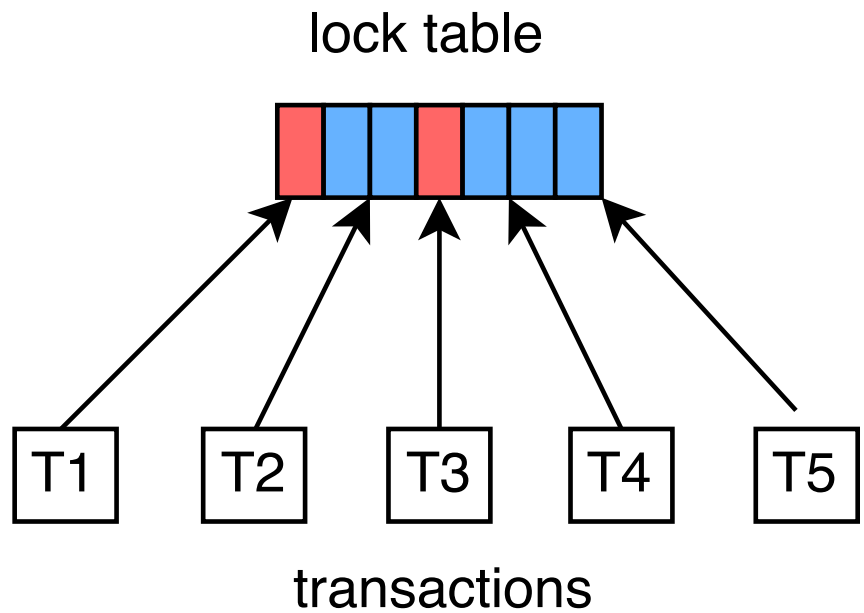
# transaction - concurrency

## Rustで実装するときの注意点 - 可変シングルトン

・以下のjavaの実装はシングルトンである、しかも変更も加えられるので可変シングルトン

・ConcurrencyMgrはTransactionがメンバーとして持つ

```
public class ConcurrencyMgr {
   private static LockTable locktbl = new LockTable();
   private Map<BlockId,String> locks  = new HashMap<BlockI
}
```



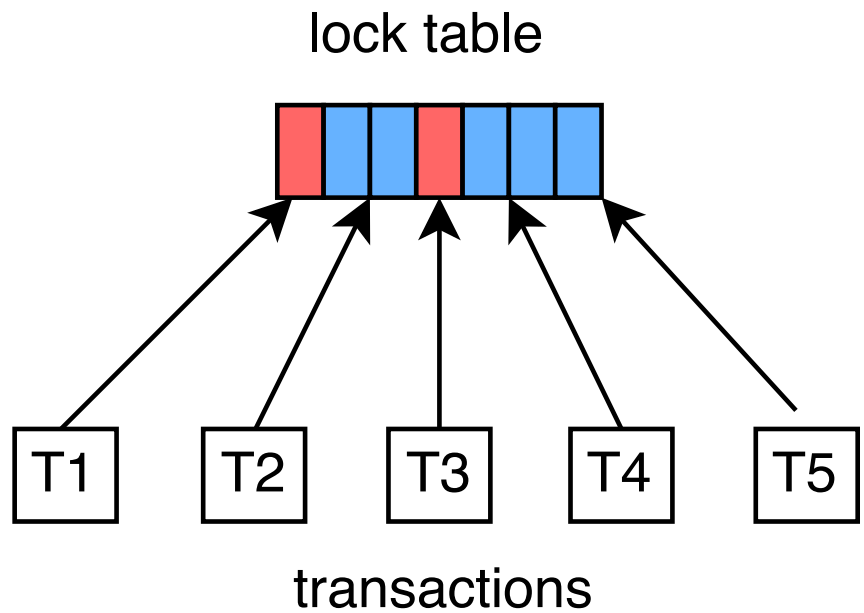
# transaction - concurrency

## Rustで実装するときの注意点 - 可変シングルトン

・以下のjavaの実装はシングルトンである、しかも変更も加えられるので可変シングルトン

・ConcurrencyMgrはTransactionがメンバーとして持つ

```
public class ConcurrencyMgr {
   private static LockTable locktbl = new LockTable();
   private Map<BlockId,String> locks  = new HashMap<BlockI
}
```



# transaction - concurrency

## Rustで実装するときの注意点 - 可変シングルトン

・Rustでは可変シングルトンの実装はunsafeを使わないと実装ができない

・ConcurrencyMgrはTransactionがメンバーとして持つ

・Rustでは可読性が落ちるが、transactionを生成するDB自体がlock tableを持つようにしている

```
pub struct OxideDB {
    block_size: usize,
    file_manager: Arc<Mutex<FileManager>>,
    log_manager: Arc<Mutex<LogManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    lock_table: Arc<Mutex<LockTable>>,
    metadata_manager: Option<Arc<MetadataManager>>,
    planner: Option<Arc<Mutex<Planner>>>,
}

pub fn new_transaction(&self) -> Transaction {
    Transaction::new(
        self.file_manager.clone(),
        self.log_manager.clone(),
        self.buffer_manager.clone(),
        self.lock_table.clone(),
    )
}
```

# transaction - concurrency

## Rustで実装するときの注意点 - 可変シングルトン

・Rustでは可変シングルトンの実装はunsafeを使わないと実装ができない

・ConcurrencyMgrはTransactionがメンバーとして持つ

・Rustでは可読性が落ちるが、transactionを生成するDB自体がlock tableを持つようにしている

```
pub struct OxideDB {
    block_size: usize,
    file_manager: Arc<Mutex<FileManager>>,
    log_manager: Arc<Mutex<LogManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    lock_table: Arc<Mutex<LockTable>>,
    metadata_manager: Option<Arc<MetadataManager>>,
    planner: Option<Arc<Mutex<Planner>>>,
}

pub fn new_transaction(&self) -> Transaction {
    Transaction::new(
        self.file_manager.clone(),
        self.log_manager.clone(),
        self.buffer_manager.clone(),
        self.lock_table.clone(),
    )
}
```

# transaction - concurrency

## Rustで実装するときの注意点 - 可変シングルトン

・Rustでは可変シングルトンの実装はunsafeを使わないと実装ができない

・ConcurrencyMgrはTransactionがメンバーとして持つ

・Rustでは可読性が落ちるが、transactionを生成するDB自体がlock tableを持つようにしている

```
pub struct OxideDB {
    block_size: usize,
    file_manager: Arc<Mutex<FileManager>>,
    log_manager: Arc<Mutex<LogManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    lock_table: Arc<Mutex<LockTable>>,
    metadata_manager: Option<Arc<MetadataManager>>,
    planner: Option<Arc<Mutex<Planner>>>,
}

pub fn new_transaction(&self) -> Transaction {
    Transaction::new(
        self.file_manager.clone(),
        self.log_manager.clone(),
        self.buffer_manager.clone(),
        self.lock_table.clone(),
    )
}
```

# transaction - concurrency

## Rustで実装するときの注意点 - 可変シングルトン

・Rustでは可変シングルトンの実装はunsafeを使わないと実装ができない

・ConcurrencyMgrはTransactionがメンバーとして持つ

・Rustでは可読性が落ちるが、transactionを生成するDB自体がlock tableを持つようにしている

```
pub struct OxideDB {
    block_size: usize,
    file_manager: Arc<Mutex<FileManager>>,
    log_manager: Arc<Mutex<LogManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    lock_table: Arc<Mutex<LockTable>>,
    metadata_manager: Option<Arc<MetadataManager>>,
    planner: Option<Arc<Mutex<Planner>>>,
}

pub fn new_transaction(&self) -> Transaction {
    Transaction::new(
        self.file_manager.clone(),
        self.log_manager.clone(),
        self.buffer_manager.clone(),
        self.lock_table.clone(),
    )
}
```

# DB実装の成果

自作結果/成果

・ソースコードの行数が14,000行を超えた(人生最高)
  ・テストが3000行ほど

・SimpleDBでも実装されているユニットテストと同じテストをOxideDBに90%以上実装、すべてPASS(25個)

・対話式コンソールでSQLが実行できるようになった

・Rust完全に理解した

# OxideDBのデモ

# OxideDBの実装課題

・インターフェースがシングルスレッドなので折角のtransactionの実装が活かされていない

・テストコードの量が足りない、実装と同じくらいコード量が必要
  ・あと8000行ほど

・デッドロック検知機を実装していない
  ・transactionの実装のデッドロック関連のデバッグで1週間溶かした

# まとめ

・SQLを実行できるところまで実装できた

・1万以上のRustのプログラミング経験を得た

# まとめ

・SQLを実行できるところまで実装できた

・1万以上のRustのプログラミング経験を得た

・理解したかったイノシシ本、今でも普通に難しい

# 将来の構想

・Rustのモデル検査機

・Rustのデッドロック検査機が欲しい

・Continuation based Rust

# ご清聴ありがとうございました

OxideDB

発表資料/マインドマップ