RustによるDBMSの実装

佐野 巧曜[1,a)]　河野 真治[1,b)]

,

概要：本研究では、Java で実装されている教育用の DBMS を Rust で再実装することを試みた。Rust で DBMS を再実装する際の課題として、Rust の言語ルールと DBMS のアーキテクチャの違いに起因する問題が存在した。これらの問題の解決方法と、Rust で DBMS を再実装することで得られる学習効果について議論する。

キーワード：DBMS, Rust, トランザクション, 並列処理

Implementation of DBMS in Rust

Yoshiaki Sano[1,a)]　Kono Shinji[1,b)]

,

Abstract: In this study, we attempted to reimplement an educational DBMS implemented in Java using Rust. There were problems due to differences between Rust's language rules and the architecture of the DBMS when reimplementing the DBMS in Rust. We discuss solutions to these problems and the learning effects obtained by reimplementing the DBMS in Rust.

Keywords: DBMS, Rust, Transaction, Parallel Processing

1. DBMSを実装により理解する

データベース管理システム（DBMS）は、データベースの作成、操作、保護を行うための、データを扱うアプリケーションに必ず使われるソフトウェアである。

DBMS を理解するために、実装を通して学ぶことが有効であると考えた。具体的には、データの永続化、インデキシング、クエリ処理、トランザクション管理、並行制御など、DBMS の核心的な機能を自身で実装することで、それらの仕組みや動作原理を直接的に学ぶことが期待される。

そこで、Java[2] で実装されている教育用の DBMS を Rust で再実装することを試みた。

Java は高レベルの機能を提供し、アプリケーション開発に適している。一方、Rust[1] は低レベルの機能を提供し、安全性、並行性、実行速度に焦点を当てて設計されているので、システムプログラミングに適している。

Rust で DBMS を実装することで、内部構造について低レベルの操作まで理解することが可能と

[1] 琉球大学工学部工学科知能情報コース
[a)] yoshisaur@cr.ie.u-ryukyu.ac.jp
[b)] kono@ie.u-ryukyu.ac.jp

なる。

2. 教育用 DBMS

本研究では、Rust で DBMS を実装するにあたり、「Database Design and Implementation」[3] というデータベース設計に関する書籍で実装されている Java で実装された SimpleDB[4] という教育用の DBMS を参考にした。

2.1 ディスク指向型アーキテクチャ

DBMS には、メモリ指向型アーキテクチャとディスク指向型アーキテクチャの二つのアーキテクチャが存在する。

SimpleDB はストレージデバイスが HDD を想定したディスク指向型アーキテクチャである。ディスク指向型アーキテクチャでは、データはディスクに格納され、必要に応じてメモリに読み込まれる。

Disk I/O だけでは、データベースの処理速度が遅くなるため、バッファプールを実装している。バッファプールは、データベースのデータを複数のブロック (バイト列の区分) に分割し、それぞれのブロックをバッファに保持する。データベースのクエリを実行する際に、必要なデータをディスクからバッファに読み込み、結果をバッファに書き込む。これにより、データベースの処理速度を向上させることができる。

図 1 は、バッファプールとディスクファイルの関係を示している。データベースのデータはディスクに格納され、必要に応じてバッファプールに読み込まれる。

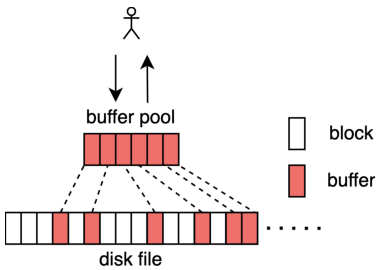

図 1 バッファプールとディスクファイル

2.2 対応するデータ型

SimpleDB は対応するデータの型は整数と ASCII の varchar のみに限定されている。これらのデータはバイト列に変換され、その後ファイルに書き込まれる。整数は固定長で 4 バイトを占め、varchar は可変長であるが、その先頭には 4 バイトの整数データが付与される。この 4 バイトの整数データには、文字列情報を含むバイト列の長さが格納されている。これにより、可変長のデータの操作も可能となる。対応するデータ型は少ないものの、固定長、可変長どちらのデータも処理できる DBMS を実装するにはこれで十分である。

以下の図 2 は整数データの読み書きを示している。

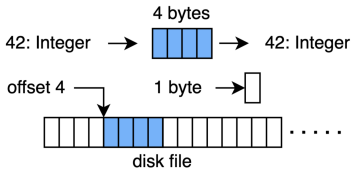

図 2 整数のデータの読み書き

また、図 3 は varchar データの読み書きを示している。

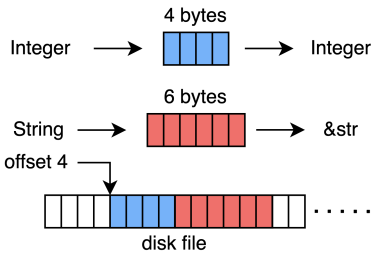

図 3 varchar のデータの読み書き

2.3 データロックの単位

DBMS では、データを一時的に他の操作から保護するためにデータロックをする。

SimpleDB では、データロックの単位はブロックと定義される。

一般的な DBMS では、データロックは通常、レコード単位で行われる。これにより、同時に行わ

れる複数の操作が同じレコードを同時に変更することを防ぎ、データの整合性を保つことができる。

しかし、SimpleDB では、データロックはブロック単位で行われる。これは、一連のバイト（つまり、ファイルの一部）を一度にロックすることを意味する。このアプローチは、レコード単位のロックに比べて粒度が粗いため、同時実行性が低下するというデメリットがある。しかし、このブロック単位でのロックは、実装が簡単になるというメリットがある。

2.4 リカバリ方式

リカバリ方式の簡単な実装方法には 2 つの方法が存在する。

一つは undo のみのアルゴリズムで、これはデータを復元する際にディスクに書き込まれたデータをログファイルの最新のログから遡って undo(元に戻す) することでデータを復元する方法である。

もう一つは redo のみのアルゴリズムで、これはデータを復元する際にログファイルの最古のログから順に redo(再実行する) することでデータを復元する方法である。

図 4 は undo ログのリカバリの例を示している。ログの行の番号が大きいほど新しいログで、強調されたログが undo されることを示している。リカバリ時点の CHECKPOINT 以降に書き込まれたログは、undo の対象となる。

```
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"
```

図 4 undo ログのリカバリの例

3. Rust と Java の比較

本研究では、Java で実装された SimpleDB と同じアーキテクチャの DBMS を Rust で実装するにあたって、Rust と Java の違いについて説明する。具体的には、メモリ管理、データ構造とメソッド、同期処理、並行処理機能の 4 つの観点から比較を行う。

3.1 メモリ管理

Rust は、所有権と借用の概念を導入し、メモリ安全性を保証する。これにより、ガベージコレクタを必要とせずに高速な実行速度を実現する。一方、Java はガベージコレクタを使用してメモリを管理する。

- Rust のメモリ管理の例:

```
1 let s = String::from("hello"); // sがスコープに入る
2 {
3     let r = &s; // rがスコープに入る, sは借用される
4 } // rがスコープから出る, 借用が終わるs
5 // sがスコープから出る, はドロップされるs
```

- Java のメモリ管理の例:

```
1 public class Main {
2     public static void main(String[] args) {
3         String s = new String("hello"); // sがスコープに入る
4         {
5             String r = s; // rがスコープに入る, sは参照される
6         } // rがスコープから出る
7     } // sがスコープから出る, はガベージコレクタにより後でクリーンアップされるs
8 }
```

3.2 データ構造とメソッド

Rust では、データ構造を定義するために struct

を、一連の関連するメソッドのシグネチャを定義するためにtraitを使用する。一方、Javaでは、データフィールドとメソッドを一緒に定義するためにclassを、一連の関連するメソッドのシグネチャを定義するためにinterfaceを使用する。

- Rustのデータ構造とメソッドの例:

```
struct Rectangle {
    width: u32,
    height: u32,
}

trait Area {
    fn area(&self) -> u32;
}

impl Area for Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}
```

- Javaのデータ構造とメソッドの例:

```
public interface Area {
    int area();
}

public class Rectangle implements Area {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public int area() {
        return this.width * this.height;
    }
}
```

3.3 同期処理

Rustでは、Synchronizedキーワードは存在せず、代わりにArc<Mutex<T>>を使用してデータの同期を行う。一方、JavaではSynchronizedキーワードを使用してオブジェクトの同期を行う。以下にそれぞれのインクリメント操作のコード例を示す。

- Rustの同期処理の例:

```
use std::sync::{Arc, Mutex};
use std::thread;

let counter = Arc::new(Mutex::new(0));
let mut handles = vec![];

for _ in 0..10 {
    let counter = Arc::clone(&counter);
    let handle = thread::spawn(move || {
        let mut num = counter.lock().unwrap();
        *num += 1;
    });
    handles.push(handle);
}

for handle in handles {
    handle.join().unwrap();
}
```

- Javaの同期処理の例:

```
import java.util.ArrayList;
import java.util.List;

public class SynchronizedCounter {
    private int c = 0;

    public synchronized void
```

```
        ↪increment() {
8           c++;
9       }
10
11      public synchronized int value()
            ↪ {
12          return c;
13      }
14  }
15
16  List<Thread> threads = new
        ↪ArrayList<>();
17  SynchronizedCounter counter = new
        ↪SynchronizedCounter();
18
19  for (int i = 0; i < 10; i++) {
20      Thread thread = new Thread(
            ↪counter::increment);
21      threads.add(thread);
22      thread.start();
23  }
24
25  for (Thread thread : threads) {
26      thread.join();
27  }
```

3.4 並行処理機能

Rust では、スレッドプールや Synchronized キューのような高レベルの並行処理機能は標準ライブラリには含まれていないが、Java ではこれらの機能が標準ライブラリに含まれている。

4. Rust による実装の問題点と解決

Rust で SimpleDB と同じアーキテクチャの DBMS を実装するにあたり、二つの問題点が生じた。どちらも Rust の厳密な言語ルールと DBMS のアーキテクチャの違いに起因するもので、しかも、どちらも DBMS の重要な機能であるトランザクションの実装に関わるものであった。最初の問題は、リカバリの実装中にインスタンス間で循環参照が必要だったが、これがメモリーリークを引き起こす可能性があったことである。二つ目の問題は、データロックの管理に可変シングルトンの実装が必要だったが、Rust ではその実装が困難であったことである。

実装では、トランザクションという構造体がデータのリカバリを担うリカバリマネージャとデータロックの管理を担うコンカレンシマネージャという構造体を持っている。

```
1  pub struct Transaction {
2      file_manager: Arc<Mutex<
           ↪FileManager>>,
3      buffer_manager: Arc<Mutex<
           ↪BufferManager>>,
4      log_manager: Arc<Mutex<LogManager
           ↪>>,
5      transaction_number: i32,
6      concurrency_manager: Arc<Mutex<
           ↪ConcurrencyManager>>,
7      buffer_list: Arc<Mutex<BufferList
           ↪>>,
8      recovery_manager: Arc<Mutex<
           ↪RecoveryManager>>,
9  }
```

4.1 循環参照

SimpleDB ではトランザクションとリカバリマネージャは循環参照をしている。これは、トランザクションがリカバリマネージャを参照し、リカバリマネージャがトランザクションを参照していることを意味する。

- Java の Transaction クラス:

```
1  public class Transaction {
2      private static int nextTxNum =
           ↪0;
3      private static final int
           ↪END_OF_FILE = -1;
4      private RecoveryMgr
           ↪recoveryMgr; // リカバリマ
           ネージャへの参照
5      private ConcurrencyMgr concurMgr
           ↪;
```

```
    private BufferMgr bm;
    private FileMgr fm;
    private int txnum;
    private BufferList mybuffers;
}
```

- Java の RecoveryMgr クラス:

```
public class RecoveryMgr {
    private LogMgr lm;
    private BufferMgr bm;
    private Transaction tx; // トランザクションへの参照
    private int txnum;
}
```

Rust では、複数のインスタンスやスレッドが一つのインスタンスにアクセスする場合、Arc<Mutex<T>>を使用する。Arc は複数の所有者を持つ不変のポインタであり、Mutex はスレッド間の同期を行う。Arc は参照カウントが 0 になったときにのみ、保持しているリソースを解放する。従って、循環参照が存在すると、参照カウントが最低でも 1 に留まり、0 にならず、結果としてメモリリークが発生する可能性がある。[5] なので、Rust のコンパイラは Arc<Mutex<T>>が循環参照を許さないように設計されている。

実際の Rust でのリカバリマネージャの実装では、メンバ変数としてトランザクションを持たせず、メソッドの引数としてトランザクションを渡すことで、循環参照を回避した。

```
// リカバリマネージャはトランザクションをメンバ変数として持たない
pub struct RecoveryManager {
    log_manager: Arc<Mutex<LogManager>>,
    buffer_manager: Arc<Mutex<BufferManager>>,
    transaction_number: i32,
}
impl RecoveryManager {
...
    // メソッドがトランザクションを引数として受け取る。これにより循環参照を避けている
    pub fn recover(&self, transaction: &mut Transaction) -> Result<(), RecoveryError> {
        ...
    }
...
}
```

4.2 可変シングルトン

SimpleDB では、データロックの管理に可変シングルトンのロックテーブルを実装している。ロックテーブルは、データブロックの ID をキーとして、そのブロックに対するロックの数を管理する。ロックの種類は、共有ロックと排他ロックの二種類である。共有ロックは、複数のトランザクションが同時にデータブロックを読み込むこと (READ) を許可する。排他ロックは、一つのトランザクションが書き込み (WRITE) を行う間、他のトランザクションが同じデータブロックにアクセスすることを禁止する。ロックテーブルで管理されているロックの数と種類をもとに、データブロックのロック状態を管理する。図 5 はロックテーブルのデータ構造を示している。

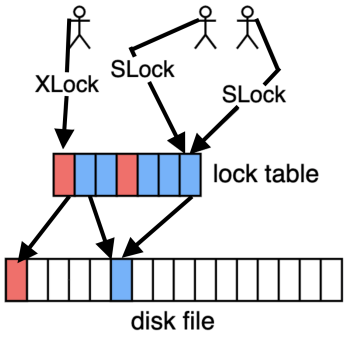

図 5 ロックテーブルのデータ構造

```
// コンカレンシマネージャでは、可変シングルトンのロックテーブルを使用してデータロックを管理する
public class ConcurrencyMgr {
    private static LockTable locktbl = new LockTable();
```

```
    private Map<BlockId, String> locks
        ↪= new HashMap<BlockId, String
        ↪>();
}
```

Java では、static キーワードを使用して、シングルトンを実装することができる。

一方、Rust では、static キーワードは存在するが、可変シングルトンを実装することはできない。

Rust のコンパイラは、可変シングルトンがスレッドセーフでないことを保証できないからである。

解決策として、コンカレンシマネージャを生成する際に、ロックテーブルを上位の構造体から渡すことで、可変シングルトンを実装した。

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

impl ConcurrencyManager {
    // コンカレンシマネージャのコンストラクタにロックテーブルを引数として渡すことで、可変シングルトンを実装
    pub fn new(lock_table: Arc<Mutex<
        ↪LockTable>>) -> Self {
        Self {
            lock_table,
            locks: HashMap::new(),
        }
    }
}
```

ConcurrencyManager の上位の構造体は Transaction であるが、Transaction のコンストラクトにもロックテーブルを引数として渡す必要がある。

```
impl Transaction {
    pub fn new(
        file_manager: Arc<Mutex<
            ↪FileManager>>,
        log_manager: Arc<Mutex<
            ↪LogManager>>,
        buffer_manager: Arc<Mutex<
            ↪BufferManager>>,
        lock_table: Arc<Mutex<
            ↪LockTable>>, // ロックテーブルを引数として渡す
    ) -> Result<Self, TransactionError
        ↪> {
        ...
    }
}
```

トランザクションの生成は、最上位の構造体である SimpleDB で行われる。つまり、SimpleDB のインスタンスを生成する際に、初めてロックテーブルが生成される。

Java の実装では、SimpleDB から Transaction、Transaction から ConcurrencyManager にロックテーブルを渡す冗長なバケツリレーの実装を可変スケルトンを使うことで回避しているが、Rust では、可変シングルトンを実装できないため、このような冗長な実装を行う必要がある。

5. DBMS の実装による成果

Rust を用いた SimpleDB の再実装は、約 1 ヶ月という期間を必要とした。全体で約 14,000 行のコードが記述され、そのうち実装部分が 11,000 行、テストコードが 3,000 行を占めた。テストコードは、元の SimpleDB と同一の内容を持ち、全てのテストが成功した。Rust の言語ルールと元の SimpleDB のアーキテクチャとの間には衝突が存在したが、Rust の言語ルールを優先することでこれらの問題を解決した。この結果、Rust を用いて DBMS を現実的な時間とコード量で実装でき、DBMS の仕組みが学習できることが示された。

6. Rust と将来の DBMS

今回の Rust での DBMS 実装では、トランザクションの実装に関わるケーススタディを行った。しかし、DBMS におけるトランザクションは、ほんの一部でしかない。

DBMS のアーキテクチャは、現実のハードウェアの制約によって決定される。アーキテクチャの

参考にした SimpleDB は、ストレージデバイスの前提が HDD であったが、最近は SSD も主流になりつつある。また、SSD の性能を最大限に引き出す NVMe というドライバインターフェースも登場していて、ハードウェアの制約自体が劇的に変化している。例えば、NVMe を In kernel で Mapping して、ヒープ上のブロックを制御するといった設計のデータベースも作ることができる。この変化に対応するには、ハードウェアと相互作用している OS に直接アクセスできる言語が必要である。Rust は Forign Function Interface(FFI) をサポートしているので、C などの言語との相互運用が可能であり、OS に直接アクセスできる。無論、Rust の FFI はメモリ安全性を保証しないので、FFI を使用する場合は、メモリ安全性の保証を強制しない unsafe キーワードを使用する必要がある [6] が、根本にあるコンピュータのハードウェアは本質的に unsafe なので、unsafe キーワードを Rust プログラミングにおいて使うことはなにも悪いことではない。Rust は、メモリ安全性を保証しないコードを、メモリ安全性を保証するコードから分離することができるので、将来の DBMS の実装において、ハードウェアとの相互作用を行う部分に C などの言語を使用し、ソフトウェアの部分に Rust を使用するなどの使い分けが有効であると考えられる。

参考文献

[1] J.Blandy and J.Orendorff: プログラミング Rust, オライリー・ジャパン ⓒ 2018.

[2] P.Niemeyer and J.Knudsen: 詳解 Java プログラミング 第 2 版 VOLUME 1, オライリー・ジャパン ⓒ 2003.

[3] Edward Sciore: Database Design and Implementation, Second Edition, Springer ⓒ 2020.

[4] Edward Sciore: ”SimpleDB: A Simple Java-Based Multiuser System for Teaching Database Internals”, Boston College Computer Science Dept, Chestnut Hill, MA, 617-552-3928.

[5] ”Reference Cycles - The Rust Programming Language”. https://doc.rust-lang.org/book/ch15-06-reference-cycles.html. 2023-11-30.

[6] ”Type layout – The Rust Reference”. https://doc.rust-lang.org/reference/type-layout.html. 2023-11-30.