mybook

目次

2.1 まえがき

これはまえがきです。

この原稿は VFM (Vivliostyle Flavored Markdown) で書かれています。詳細はドキュメントを確認してください。

3.1 ソースコード

```
function main() {}
```

3.1.1 キャプション付き

app.js

```
function main() {}
```

or

app.js

```
function main() {}
```

3.2 後注

VFM は GitHub リポジトリ[1]で開発されています。 issue は GitHub[2]で管理されています。 後注はインラインで記述することもできます[3]。

3.3 Frontmatter

Markdown ファイルの冒頭で、メタデータを設定することができます。詳細は Frontmatter part in VFM document を確認してください。

3.4 改行

デフォルトでは、空行を 1 行はさむことで改行が行われます。Frontmatter で `hardLineBreaks` を `true` にすると、空行なしで改行が行われます。

はじめまして。

Vivliostyle Flavored Markdown（略して VFM）の世界へようこそ。VFM は出版物の執筆に適した Markdown 方言であり、Vivliostyle プロジェクトのために策定・実装されました。

3.5 画像

3.5.1 キャプション

Vivliostyle Logo

Vivliostyle Logo

3.6 数式

インライン： $x = y$

ディスプレイ：

$$1 + 1 = 2$$

3.7 HTML

Hey

3.7.1 Markdown と HTML の併用

- hoge
- fuga

3.8 ルビ

This is Ruby(ルビ)

---

1. VFM↩
2. Issues↩
3. これは後注です。↩

4.1 あとがき

これはあとがきです。

データベースのロックについて、資料を読んだり実際に試してみたので、学んだことを整理してみようと思います。はじめにロックについての基本的な知識を整理して、最終的にはデッドロックとその対策について説明します。

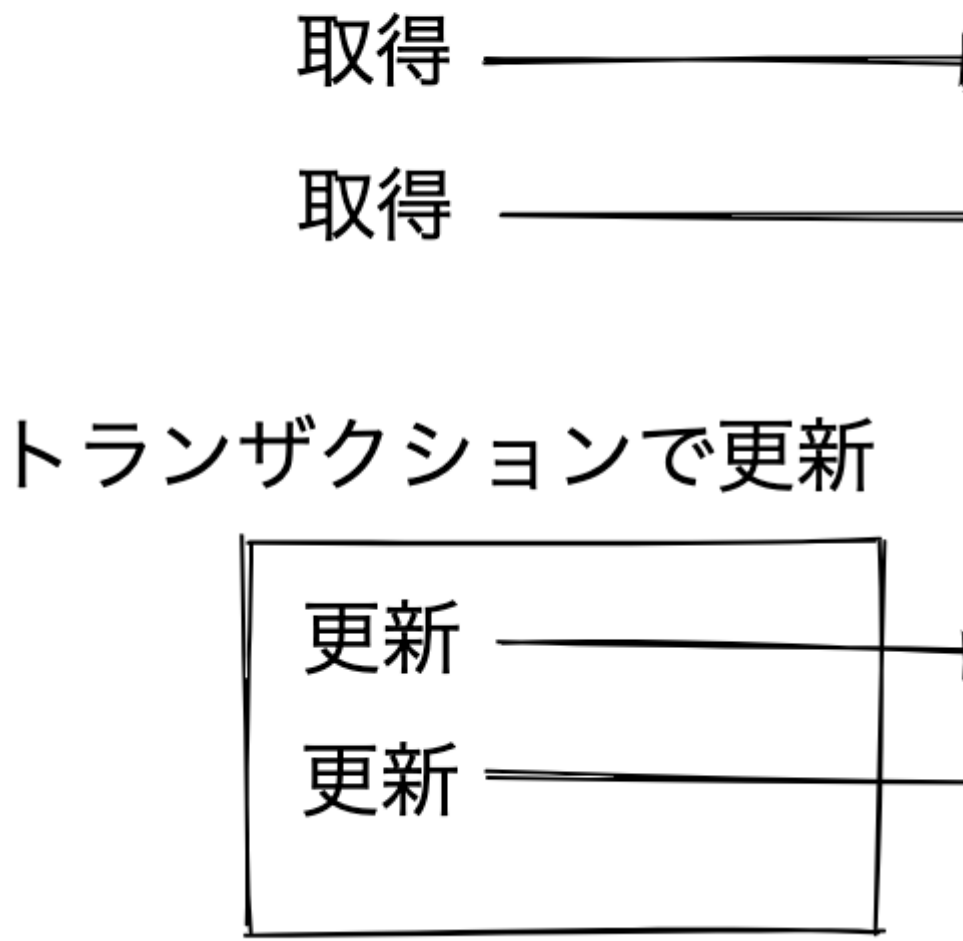
5.1 使用したソフトウェアのバージョン

- MySQL 8.0.31

5.2 ロックとは何か（概要）

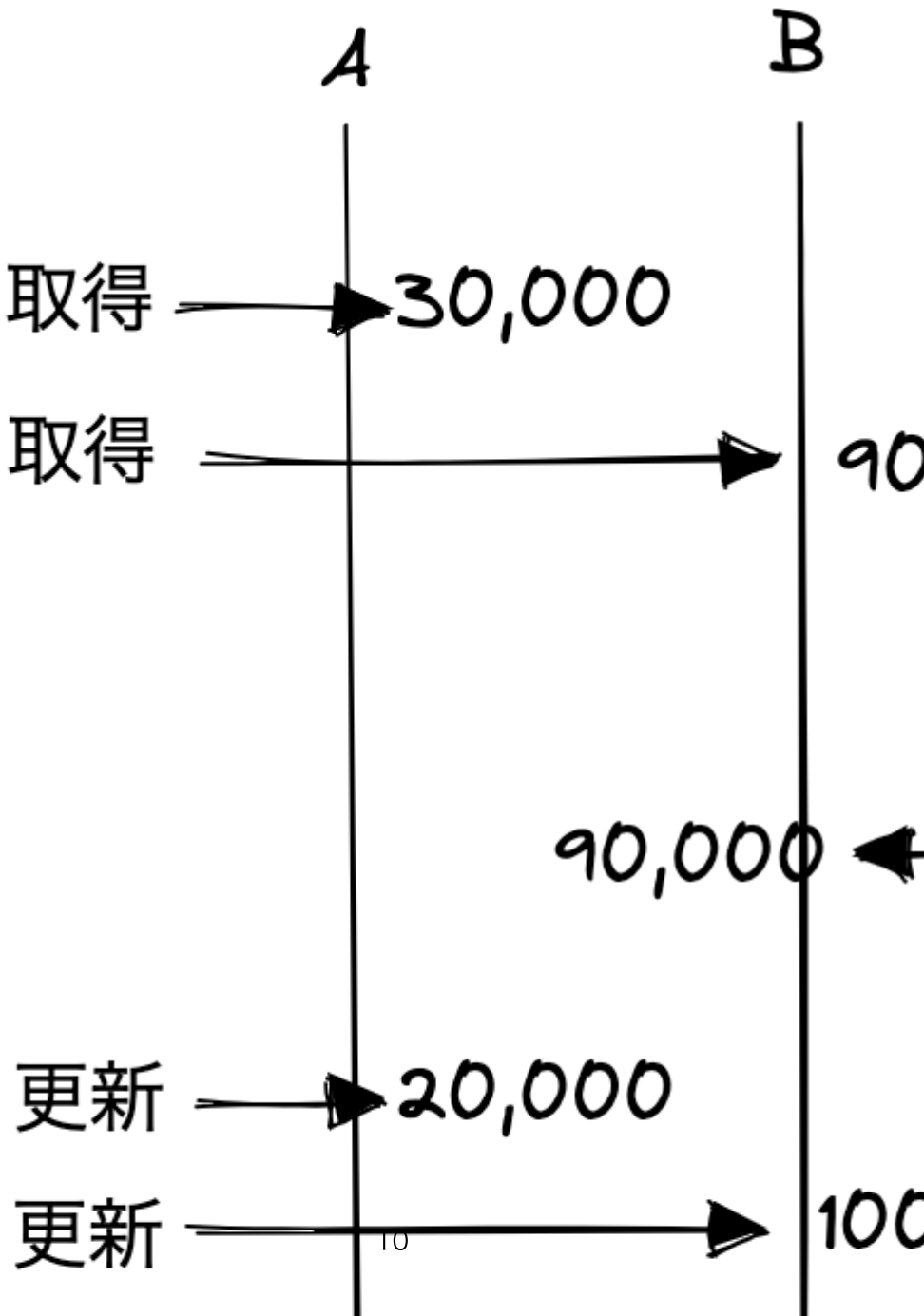
ロックはデータの更新を正しく行うための仕組みの一つで、あるデータに対する更新処理を制御するためのものです。ここで、データを正しく更新するとはどういうことかを説明するために、口座への振込を例に考えてみます。

例えば、口座Ａから口座Ｂに１万円の振込を行うとします。このとき、口座Ａの出金処理と口座Ｂの入金処理は、必ず両方が成功しなければなりません。このための仕組みが、みなさんご存じの通りトランザクションです。


取得
取得
トランザクションで更新
更新
更新

トランザクションによる更新

　また、口座Ａから口座Ｂに振り込むのと同じタイミングに、口座Ｃから口座Ｂに１万円振り込むとします。このとき、参照と更新が次のような順序で行われると、Ｂが２万円増えるべきところが１万円しか増えないことが起こってしまいます。


A
B
取得
30,000
取得
90
90,000
更新
20,000
更新
100

正しく更新できていない

このような同時更新による問題を防ぐための仕組みがロックです。

5.3 ロックとは何か（詳細）

ロックとは「あるデータに対する更新処理を制御するための仕組み」でした。データに対する更新処理を制御するというのは、同時に行われると問題が起きる処理を順番に実行するということです。そうするためには、あるデータを変更できるトランザクションを１つまでに制限します。

5.3.1 ２種類のロック

ロックには排他ロックと共有ロックの２種類があります。先述の「変更できるトランザクションを１つまでに制限する機能」は、排他ロックのことです。排他ロックは、「これからこのデータを変更するので、他のトランザクションは変更しないでね」ということを表します。

共有ロックは、トランザクション内で読み取りを行なっていることを示すためのロックです。つまり、「今このデータを読み取っているので、他のトランザクションは変更しないでね」ということを表します。

排他ロックと共有ロックは、ロックをかけられたデータが他のトランザクションから変更できないという点は同じです。異なる点は、排他ロックがデータを変更することが目的なのに対し、共有ロックはそうではない点です。

排他ロックと共有ロックの共通点や違いをまとめてみると、次のようになります（トランザクションをTXと略しています）。

	他のTXから読み取れる	**他のTXから変更できる**	**自分のTXから変更できる**	**他のTXから排他ロックを取得できる**	**他のTXから共有ロックを取得できる**
排他ロック	o	x	o	x	x

	他のTXから読み取れる	他のTXから変更できる	自分のTXから変更できる	他のTXから排他ロックを取得できる	他のTXから共有ロックを取得できる
共有ロック	o	x	△	x	o

排他ロックはデータを順番に更新するための仕組みなので、複数のトランザクションが取得することはできません。一方で、共有ロックは複数のトランザクションが取得できます。

5.3.2 ロックの取得するにはどうすればよいか

ロックを取得する代表的な方法には、次の２つがあります。

- トランザクション内でロック読み取りする（`SELECT ... FOR UPDATE`、`SELECT ... FOR SHARE`）
- `UPDATE`や`DELETE`を実行する

これらの方法では、検索条件に主キーなどの一意なインデックスを使った場合は、該当するレコードにロックがかけられます。

:::message 範囲検索を使った場合や、一意でないインデックスを使った場合は異なる挙動になりますが、ここでは簡単のためにレコードロックのみを考えることにします。 :::

5.3.3 ロックを取得してみる

それでは、実際に排他ロックと共有ロックを取得してみます。

まずは排他ロックを取得してみます。

```
mysql(T1)> begin;
Query OK, 0 rows affected (0.00 sec)

mysql(T1)> select * from numbers where id = 1 for upda
+----+-------+
| id | value |
```

```
+----+-------+
|  1 |    30 |
+----+-------+
1 row in set (0.00 sec)
```

別のセッションで、同じレコードに対して排他ロックの取得を試してみます。

```
mysql(T2)> begin;
Query OK, 0 rows affected (0.00 sec)

mysql(T2)> select * from numbers where id = 1 for upda
```

すると、実行結果がすぐ返ってきません。このように、ロックが取得できない場合はロックが待機状態（解放待ち）になります。待機状態のロックは、`innodb_lock_wait_timeout`秒（デフォルトでは50秒）が経過すると取得に失敗します。

ロックを解放するために、T1のトランザクションを終了します。すると、T2のクエリの実行結果が表示され、ロックも取得できました。

```
mysql(T1)> commit;
Query OK, 0 rows affected (0.00 sec)

mysql(T2)> select * from numbers where id = 1 for upda
+----+-------+
| id | value |
+----+-------+
|  1 |    30 |
+----+-------+
1 row in set (5.47 sec)
```

次は共有ロックです。共有ロックは、複数のトランザクションから取得できます。

```
mysql(T1)> begin;
Query OK, 0 rows affected (0.00 sec)

mysql(T1)> select * from numbers where id = 1 for share
+----+-------+
| id | value |
+----+-------+
|  1 |    30 |
+----+-------+
1 row in set (0.00 sec)

mysql(T2)> begin;
Query OK, 0 rows affected (0.01 sec)

mysql(T2)> select * from numbers where id = 1 for share
+----+-------+
| id | value |
+----+-------+
|  1 |    30 |
+----+-------+
1 row in set (0.00 sec)
```

ロックの取得状態は `performance_schema.data_locks` テーブルで確認できます。`S, REC_NOT_GAP` となっているのがレコードの共有ロックです。

```
mysql(T1)> select THREAD_ID ,OBJECT_NAME, INDEX_NAME,
+-----------+-------------+------------+-----------+---
| THREAD_ID | OBJECT_NAME | INDEX_NAME | LOCK_TYPE | LO
```

```
+-----------+-------------+------------+-----------+---
|        48 | numbers     | NULL       | TABLE     | I
|        48 | numbers     | PRIMARY    | RECORD    | S
|        50 | numbers     | NULL       | TABLE     | I
|        50 | numbers     | PRIMARY    | RECORD    | S
+-----------+-------------+------------+-----------+---
4 rows in set (0.00 sec)
```

カラムの詳細については、InnoDBの行ロック状態を確認する［その1］| gihyo.jp などを参照してみて下さい。

5.3.4 ここまでの整理

ここまでの内容を、Q & A形式で整理してみます。

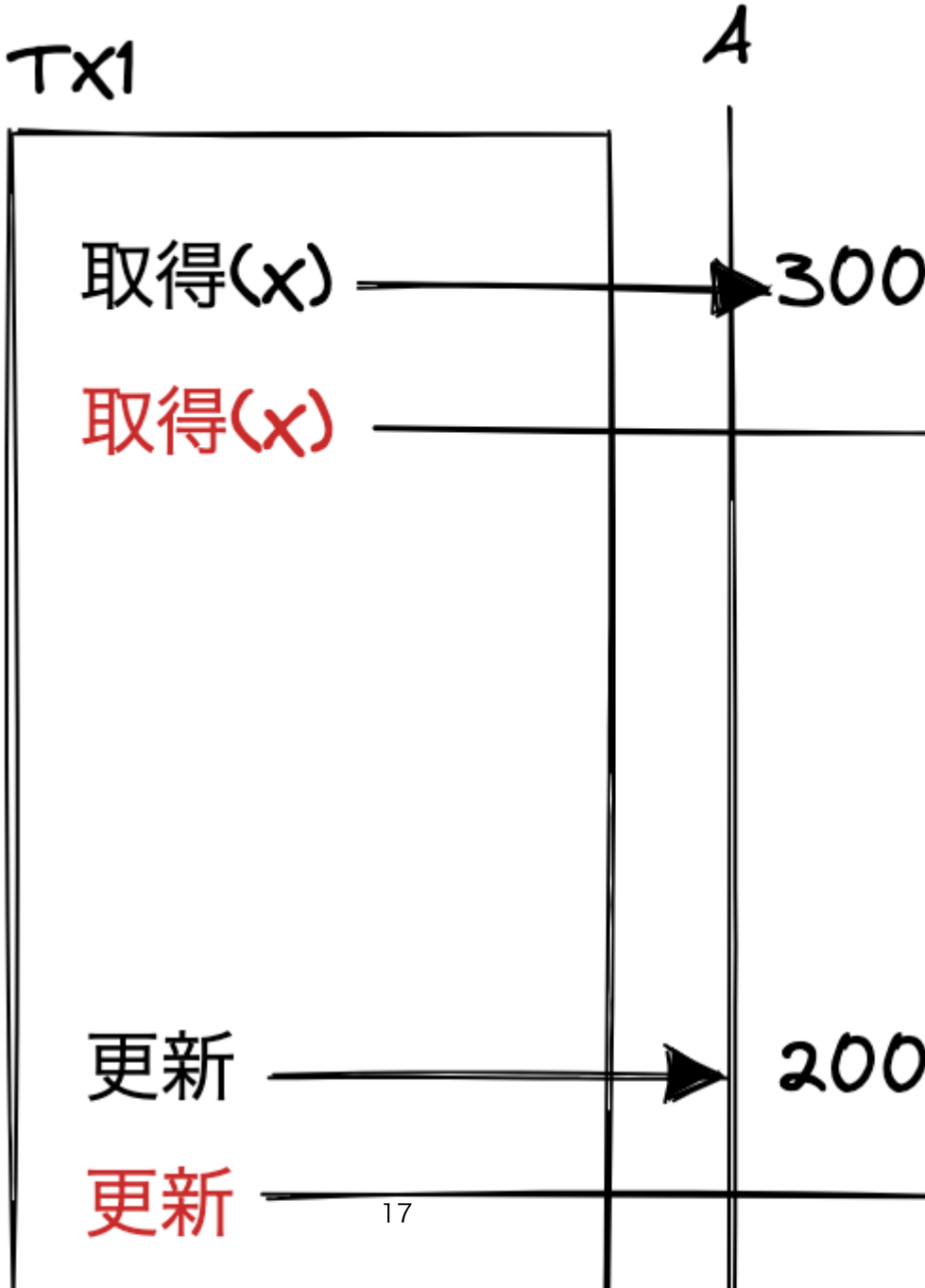
5.4 最初の問題をロックで解決する

それでは、最初の振込の問題をロックを使って解決してみましょう。どのような問題だったかというと、口座A→口座Bへの入金と、口座C→口座Bへの入金が同じタイミングで起こり、口座Bの入金金額がおかしくなってしまうという問題です。

解決するためには、入金処理を開始するときに2つの口座の排他ロックを取得すればよいです（`{}`はプレースホルダを表すこととします）。

```
begin;
select * from accounts where id = {id_from} for update
select * from accounts where id = {id_to} for update;
update accounts set balance = balance - {amount} where
update accounts set balance = balance + {amount} where
commit;
```

　こうすることで、同じ口座に関する処理は順番に行われるため、口座Ｂの金額がおかしくなることを防げます（赤線がＢの更新処理で、順番に行われていることがわかります）。


Tx1
A
取得(x)
300
取得(x)
更新
200
更新

ロックで解決

5.5 デッドロックとは何か

ロックを使うことでデータを正しく更新できるようになりました。しかし、ロックはまた別の問題を引き起こす可能性があります。それはデッドロックです。

デッドロックとは、複数のトランザクションが互いに相手が所有するロックが解放されるのを待機して、処理が進まなくなってしまうことをいいます。

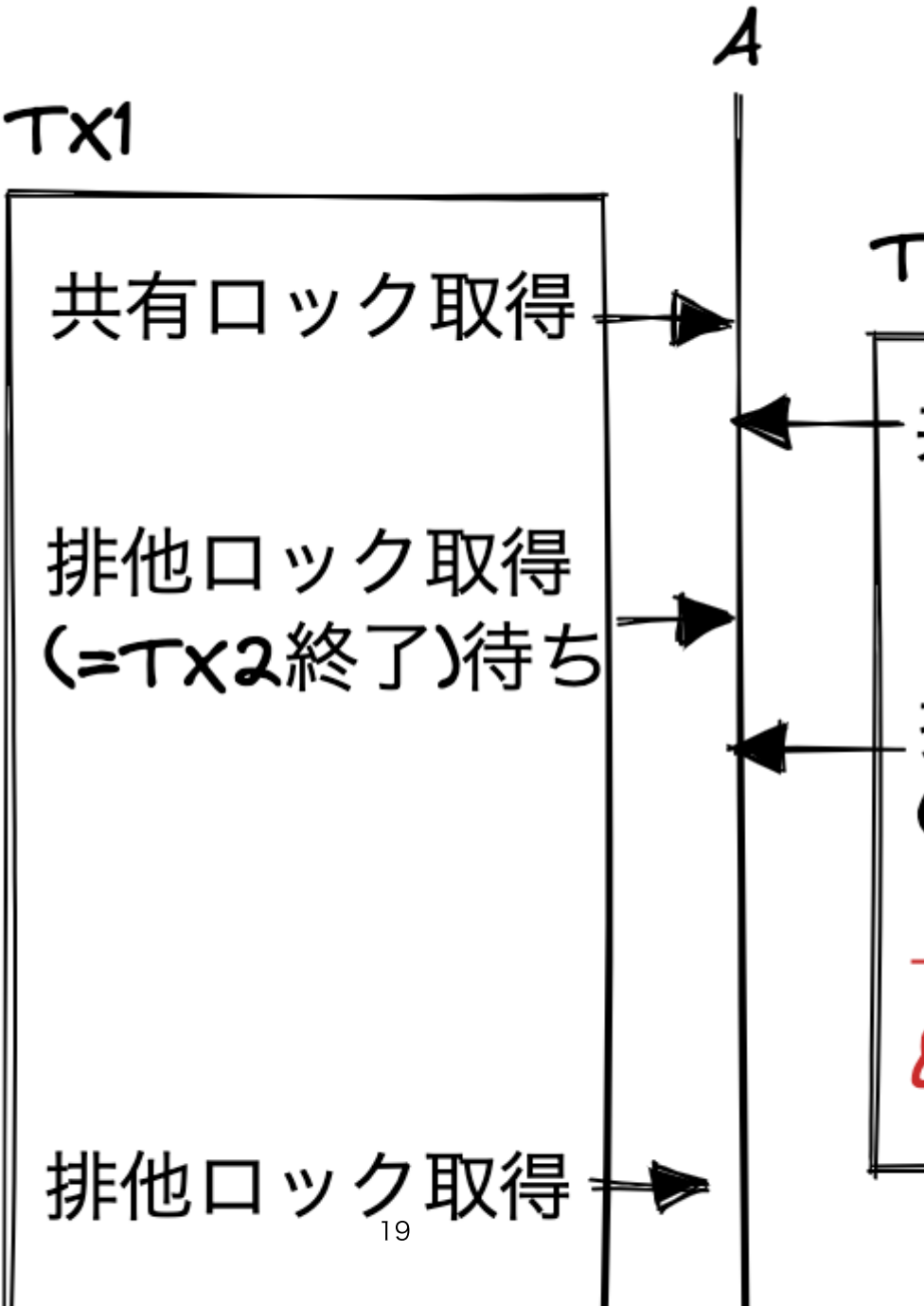
定義から分かるように、デッドロックは共有ロックだけを使う場合は起きません（共有ロックだけではロックの待機が起こらないため）。つまり、共有ロック・排他ロックが両方使われる場合と、排他ロックが複数使われる場合に発生します。

少々無理やりになってしまいますが、以下の単純なテーブルを使ってデッドロックを発生させてみます。

```
CREATE TABLE numbers (
  id INT PRIMARY KEY AUTO_INCREMENT,
  value INT NOT NULL
);
```

5.5.1 共有ロックと排他ロックによるデッドロック（変換デッドロック）

T1 と T2 でそれぞれ共有ロックを取得したあと、T1 と T2 の両方で排他ロックを取得しようとするとデッドロックになります。排他ロックを取得するときに、共有ロックの解放待ちになるためです。


A
Tx1
共有ロック取得
排他ロック取得
(=Tx2終了)待ち
排他ロック取得

共有ロックと排他ロックによるデッドロック
:::details 実行例

```
 mysql(A)> begin;
Query OK, 0 rows affected (0.00 sec)

mysql(A)> select * from numbers where id = 1 for share
+----+-------+
| id | value |
+----+-------+
|  1 |    30 |
+----+-------+
1 row in set (0.00 sec)

mysql(B)> begin;
Query OK, 0 rows affected (0.00 sec)

mysql(B)> select * from numbers where id = 1 for share
+----+-------+
| id | value |
+----+-------+
|  1 |    30 |
+----+-------+
1 row in set (0.00 sec)

-- ロック待ちになり、 相手がデッドロックがロールバックされてから
mysql(A)> update numbers set value = 100 where id = 1;
Query OK, 1 row affected (4.34 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

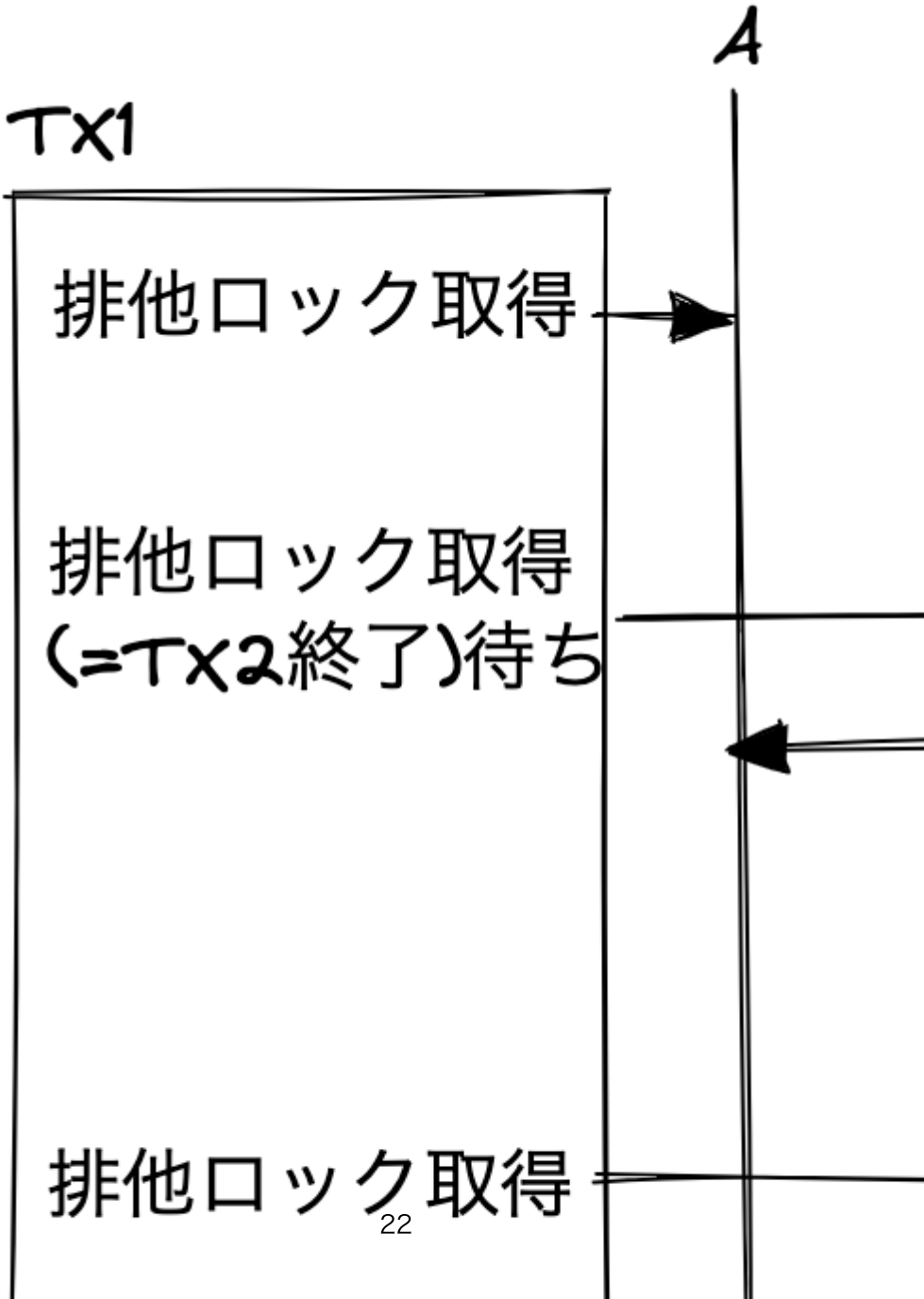
```
mysql(B)> update numbers set value = 100 where id = 1;
ERROR 1213 (40001): Deadlock found when trying to get
```

:::

なお、共有ロックを取得している状態で排他ロックを取得しようとして起こるデッドロックを「変換デッドロック」と呼ぶようです。

5.5.2 複数の排他ロックによるデッドロック（サイクルデッドロック）

T1 と T2 でそれぞれ別のデータの排他ロックを取得したあと、相手側が持つデータの排他ロックを取得しようとするとデッドロックになります。


A
TX1
排他ロック取得
排他ロック取得
(=TX2終了)待ち
排他ロック取得

複数の排他ロックによるデッドロック

:::details 実行例

```
 mysql(A)> begin;
Query OK, 0 rows affected (0.00 sec)

mysql(B)> begin;
Query OK, 0 rows affected (0.00 sec)

mysql(A)> select * from numbers where id = 1 for updat
+----+-------+
| id | value |
+----+-------+
|  1 |    30 |
+----+-------+
1 row in set (0.02 sec)

mysql(B)> select * from numbers where id = 2 for updat
+----+-------+
| id | value |
+----+-------+
|  2 |    10 |
+----+-------+
1 row in set (0.00 sec)

mysql(A)> select * from numbers where id = 2 for updat
-- ロック待ちになり、Bがデッドロックでロールバックされてから表示
+----+-------+
| id | value |
+----+-------+
|  2 |    10 |
```

```
+----+-------+
1 row in set (6.82 sec)

mysql(B)> select * from numbers where id = 1 for updat
ERROR 1213 (40001): Deadlock found when trying to get
```

:::

このようなデッドロックを、排他ロックの待機がサイクル状になるため「サイクルデッドロック」と呼びます。

5.6 デッドロックが起こらないようにするためには

デッドロックが起こる原因は、共有ロックを取得したデータを更新してしまうことや、トランザクションでの更新順序が一定でないことです。つまり、更新する場合は排他ロックを取得することと、トランザクションでの更新順序を一定にすることを守れば防げるはずです。

デッドロックの理解を深めるために、私が現在開発している在庫管理システムで遭遇しそうだった例を紹介します。

5.6.1 在庫管理システムでのデッドロック

商品には複数の在庫があり、在庫は現在の在庫数を持っています。在庫→商品への外部キーと、入荷→在庫への外部キーには、外部キー制約が設定されています。

:::details DDL

```
 -- 商品
CREATE TABLE products (
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  name VARCHAR(191) NOT NULL
);

-- 在庫
```

```
CREATE TABLE inventories (
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  product_id INT NOT NULL,
  current_quantity INT NOT NULL,

  FOREIGN KEY (product_id) REFERENCES products(id) ON I
);

-- 入荷
CREATE TABLE arrivals (
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  inventory_id INT NOT NULL,
  quantity INT NOT NULL,
  arrived_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAI

  FOREIGN KEY (inventory_id) REFERENCES inventories(id
);

insert into products (id, name) values (1, 'ハサミ'), (2
insert into inventories (product_id, current_quantity)
```

:::

入荷があるごとに入荷レコードを登録し、関連する在庫の在庫数を更新します。具体的には次のようなコードです。

```
begin;
insert arrivals (inventory_id, quantity) values ({inver
update inventories set current_quantity = current_quan
commit;
```

UPDATE文で排他ロックがかけらるため、在庫数の更新は正しく行えます。しかし、これは同時に実行されたときにデッドロックが発生する可能性があります。実際に起こしてみましょう。

```
 -- T1で入荷を登録
mysql(T1)> begin;
Query OK, 0 rows affected (0.00 sec)

mysql(T1)> insert into arrivals (inventory_id, quantity
Query OK, 1 row affected (0.04 sec)

-- T2で入荷を登録
mysql(T2)> begin;
Query OK, 0 rows affected (0.00 sec)

mysql(T2)> insert into arrivals (inventory_id, quantity
Query OK, 1 row affected (0.02 sec)

--- T1で在庫を更新しようとすると、ロック待ちになる(???)
mysql(T1)> update inventories set current_quantity = cu

---T2でも在庫を更新すると、デッドロックになる
mysql(T2)> update inventories set current_quantity = cu
ERROR 1213 (40001): Deadlock found when trying to get
```

一体なぜ、在庫の更新の部分でロック待ちになるのでしょうか？原因は、**入荷にINSERTするときに、関連する在庫に外部キーによる共有ロックが設定されるから**です。

デッドロックが発生したのは、共有ロックがかかったデータに、2つのトランザクションから排他ロックを取得しようとしたためです。つまり、変換デッドロックが発生しています。

外部キーによる共有ロックの取得については、MySQLのドキュメントに次のように書かれています。

FOREIGN KEY 制約がテーブル上で定義されている場合は、制約条件をチェックする必要がある挿入、更新、または削除が行われると、制約をチェックするために、参照されるレコード上に共有レコードレベルロックが設定されます。 MySQL :: MySQL 8.0 リファレンスマニュアル :: 15.7.3 InnoDB のさまざまな SQL ステートメントで設定されたロック

共有ロックは、「データを参照するので更新しないでねというロック」でした。そのため、子テーブルへのINSERTで親テーブルの関連レコードに共有ロックがかかるのは、外部キー制約の挙動としては妥当だと考えられます。

デッドロックの原因は、共有ロックを取得した在庫レコードを更新していることです。つまり、最初に更新のための排他ロックを取得すれば、デッドロックを防げます。

```
begin;
-- 最初に排他ロックを取得する
select id from inventories where id = {inventory_id} fo
-- 排他ロックを取得しているため、ここで共有ロックは取得されない
insert arrivals (inventory_id, quantity) values ({inver
update inventories set current_quantity = current_quan
commit;
```

5.6.2 その他のデッドロックの例

Zennの記事では、次の2つの記事が勉強になりました。どちらの例も、意図しない共有ロックの取得によって変換デッドロックが起きています。

https://zenn.dev/tockn/articles/4268398c8ec9a9

https://zenn.dev/shuntagami/articles/ea44a20911b817

デッドロックの対策として、「親レコードを排他ロックする」「再試行する」「デッドロックを許容する」「Redis Mutexを使う」「分離レベルを下げる」などの方法が紹介されています。

5.7 まとめ

- データを正しく更新するためのデータベースの仕組みには、トランザクションとロックがある
- ロックは、あるデータに対する更新処理を制御するための仕組み
- ロックには、データの更新を行うための排他ロックと、データを読み取っていることを示すための共有ロックがある
- ロックしようとしているデータが既にロックされている場合は、ロックの解放待ち（トランザクションの終了待ち）になる。ただし、共有ロックは複数取得できる。
- レコードロックを取得する代表的な方法には、ロック読み取り（SELECT … FOR UPDATE、SELECT … FOR SHARE）と、UPDATE文・DELETE文がある。UPDATEやDELETEでは、レコードに排他ロックが設定される。
- 複数のトランザクションが互いに終了待ちになることをデッドロックという。デッドロックが発生すると、片方のトランザクションがロールバックで強制終了される。
- デッドロックの原因には、データの更新を行うのに共有ロックを取得していること（変換デッドロック）や、データを更新する順序が一定でないこと（サイクルデッドロック）がある。

5.8 参考

- Database-Exclusion-Control.pdf