

データベースの排他制御

TIS株式会社

テクノロジー＆イノベーション本部

テクノロジー&エンジニアリングセンター



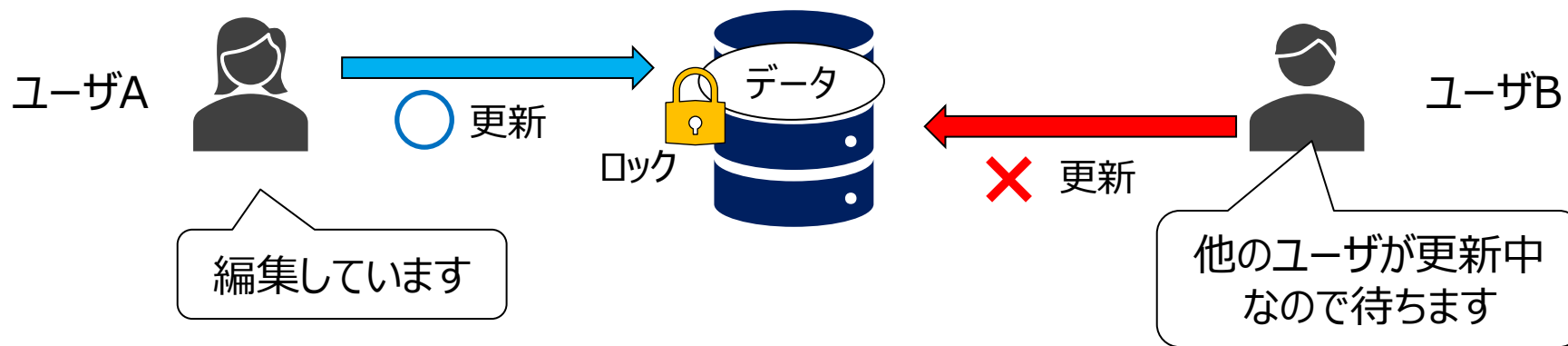
- OracleおよびJava、MySQLは、オラクルおよびその関連会社の登録商標です。
 - DB2は世界の多くの国で登録された International Business Machines Corporation の商標です。
 - Microsoft SQL Serverはマイクロソフトの製品です。Microsoft Excelは Microsoft Corporationの商標または登録商標です。
- なお、本文中ではそれぞれ、SQL Server、Excelと表記しています。
- その他の社名、製品名などは、一般にそれぞれの会社の商標または登録商標です。
- なお、本文中では、TMマーク、Rマークは明記しておりません。

- 排他制御設計とは
- 排他制御を実現するための技術要素
 - 物理ロック
 - 論理ロック
- 排他制御設計の考え方(事例を通して解説)

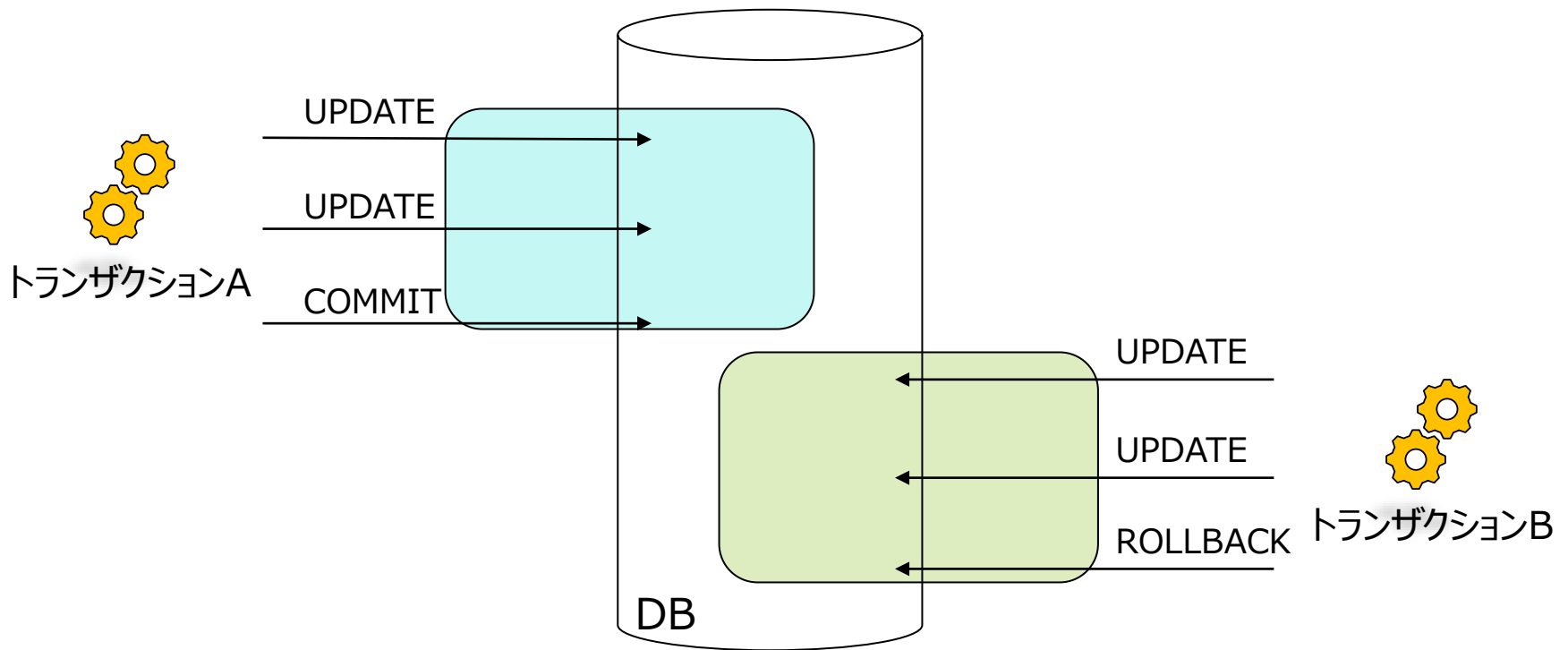
本講座では、データベースに関する排他制御を取り扱います。

排他制御設計とは

複数のユーザやトランザクションが同時に
同じデータを操作できないようにする仕組み



関連する複数の処理を一つの
処理単位としてまとめたもの。



- データ整合性の維持

- 処理が同時実行されても、データ不整合を絶対に防止しなければならない。

忘れがち。
データ整合性だけを考えると、
性能が出ないことがある。

- スループットの最大化

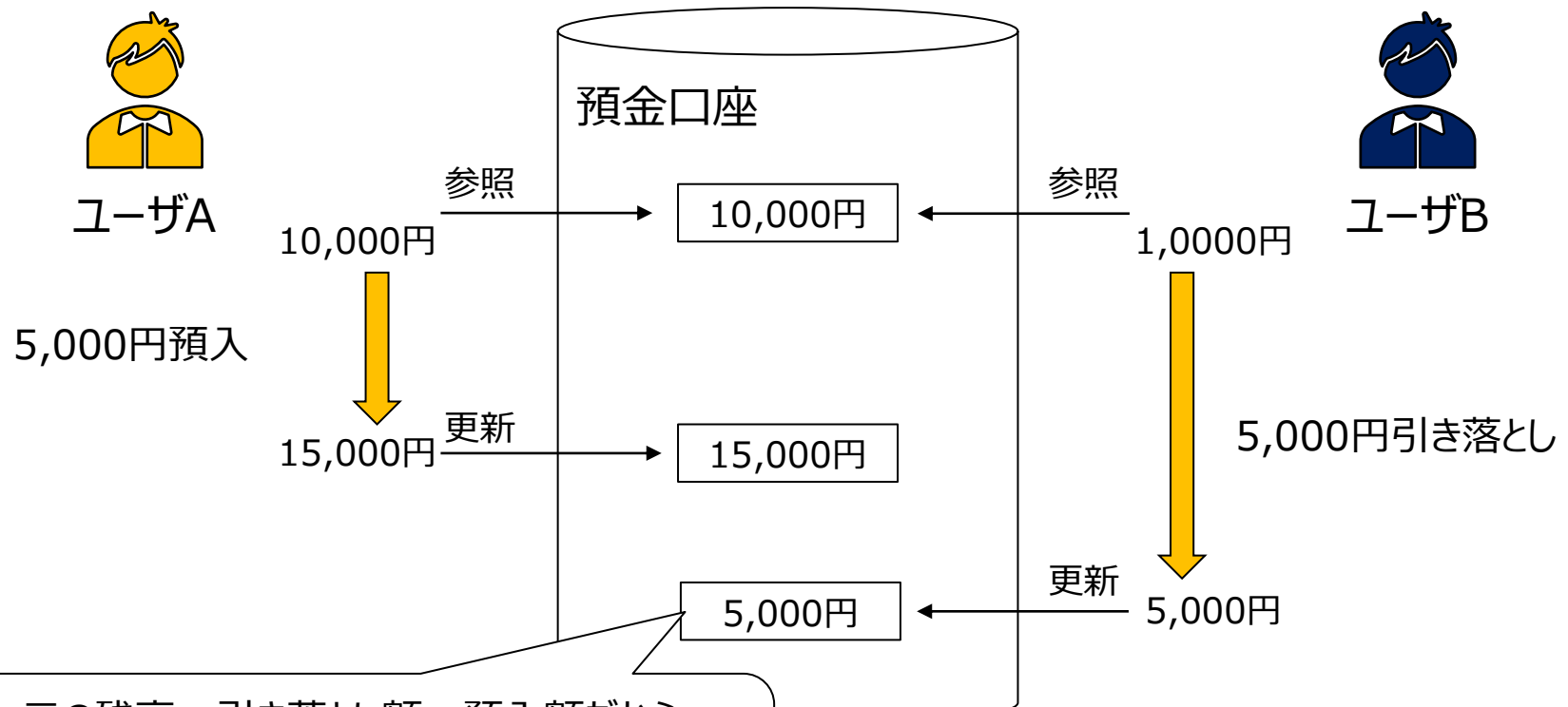
- データ整合性を維持することを前提としつつ、性能要件として求められるスループットを実現しなければならない。

※スループット：単位時間あたりに処理できる量のこと。性能を表す指標の一つ。

なぜ排他制御設計を学ぶのか(1/5)

排他制御設計がされていないと…

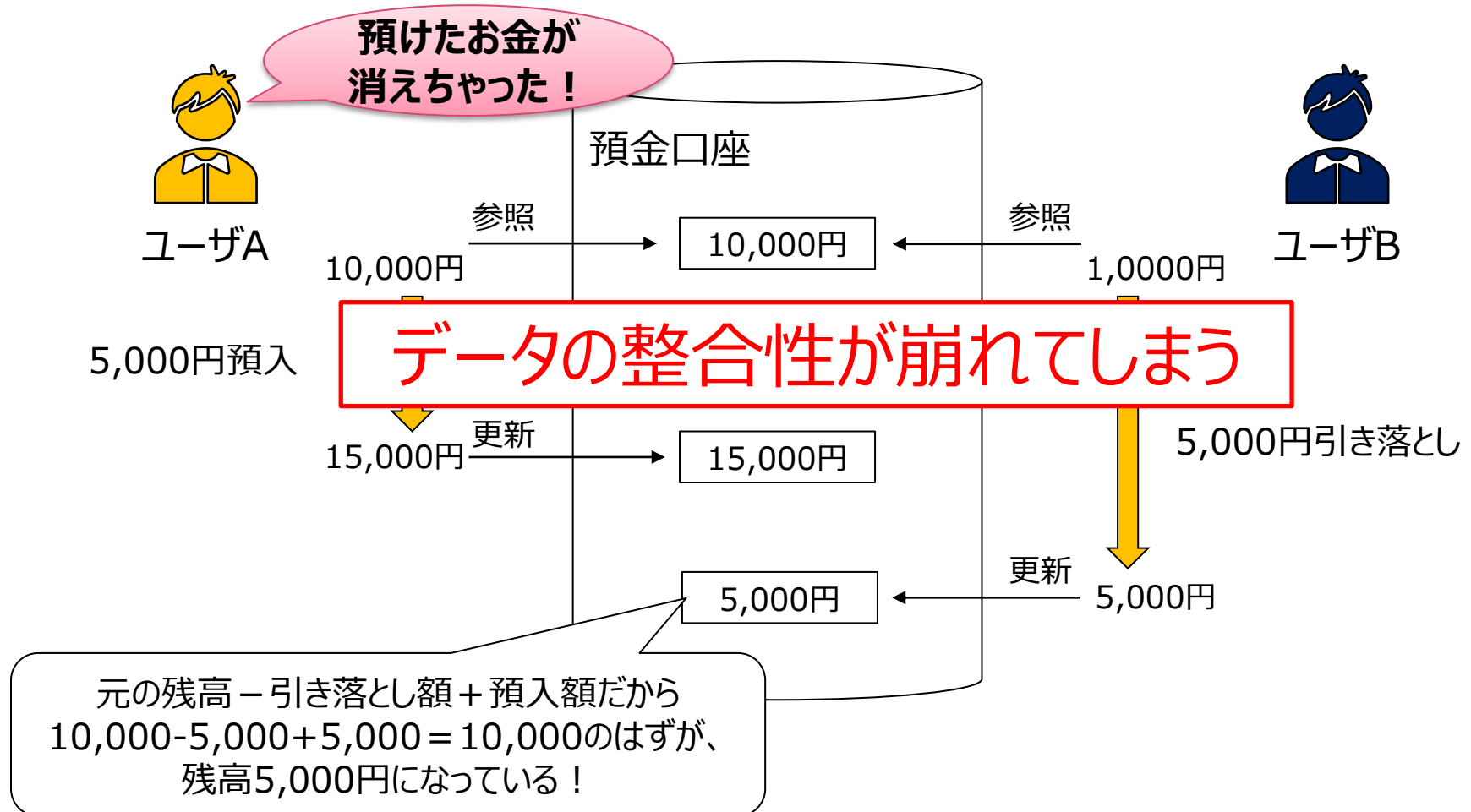
①排他制御そのものがされていないパターン



なぜ排他制御設計を学ぶのか(2/5)

排他制御設計がされていないと…

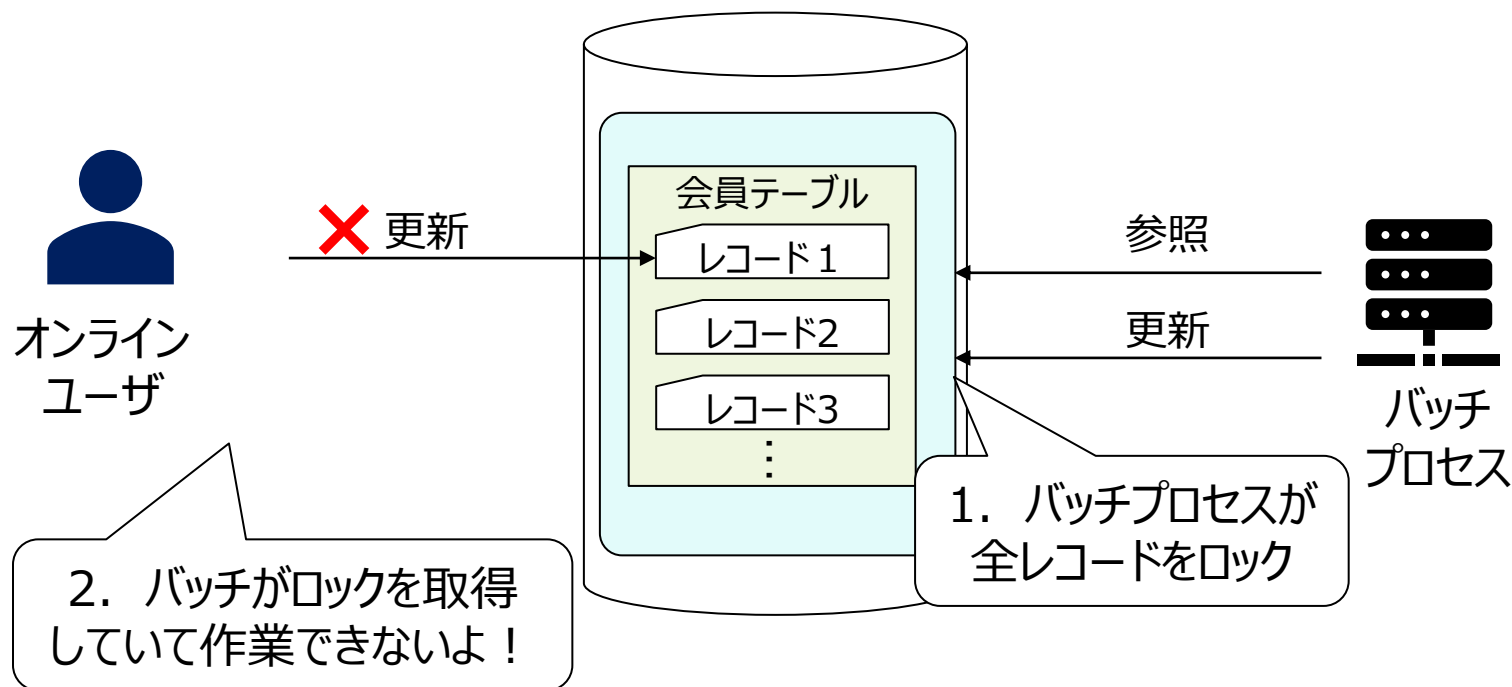
①排他制御そのものがされていないパターン



なぜ排他制御設計を学ぶのか(3/5)

排他制御設計がされていないと…

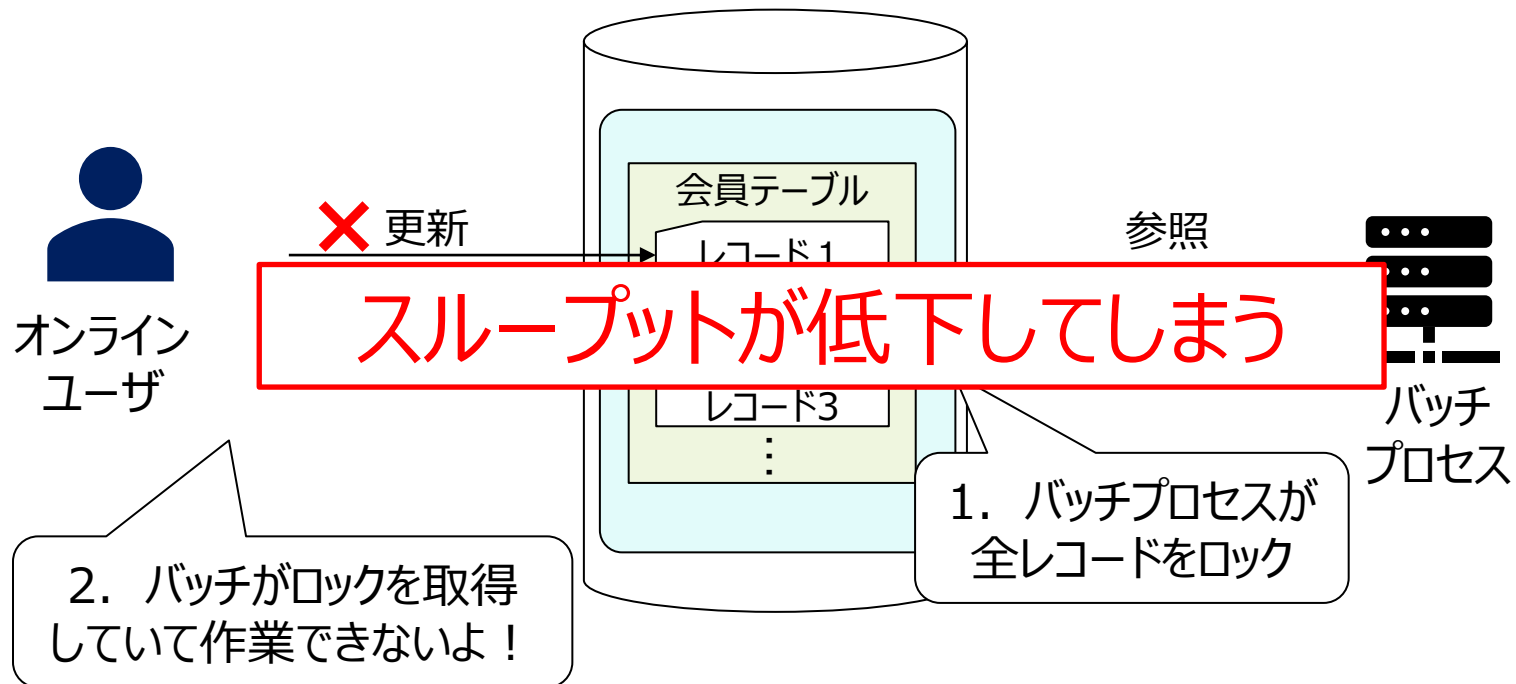
②排他制御はされているが、設計が甘いパターン



なぜ排他制御設計を学ぶのか(4/5)

排他制御設計がされていないと…

②排他制御はされているが、設計が甘いパターン



なぜ排他制御設計を学ぶのか(5/5)

このように、

- 排他制御がされていなければ、データ不整合が起こる
- 排他制御がされていても、設計が不十分だとスループットが低下する
と言った問題が起こります。

これらを防ぐために、排他制御について学んでいきましょう。

本講座では、まず排他制御を実現するための仕組みとして

- DBMSで実現される物理ロック
- アプリケーションレベルで実現される論理ロック

を説明します。

その後、排他制御設計の考え方について、事例を用いて解説します。

排他制御を実現するための技術要素

排他制御を実現するための仕組みは大きく2つに分けられます。

- **物理ロック**

- DBMSで実現されているロック機構。
- 共有ロック、更新ロック、排他ロックなどがある。

- **論理ロック**

- アプリケーションレイヤーで実現するロック機構。
- 複数のトランザクションに跨るロックを実現する。
- 楽観的ロック、悲観的ロックがある。

実際のシステムでは、これらを**組み合わせ**て**排他制御を実現**します。
どう組み合わせるか？ が設計にあたります。

それでは、物理ロック、論理ロックについてみて行きましょう。

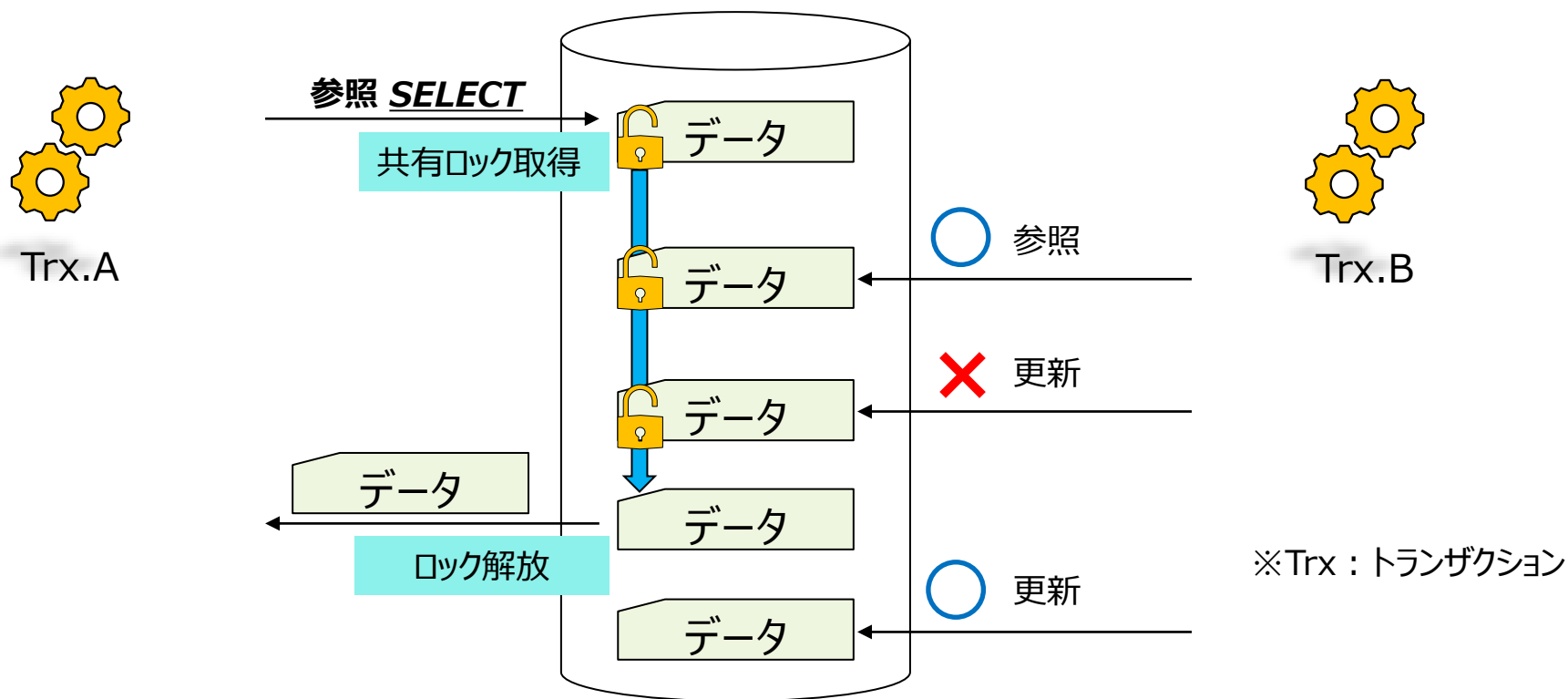
排他制御を実現するための技術要素 ～ 物理ロック ～

DBMSによって物理ロックの分類や数は異なります。しかし、ほとんどのDBMSでも以下の3つは存在し、実務上もこれらを押さえておけば大丈夫です。

- 共有ロック
- 排他ロック
- 更新ロック

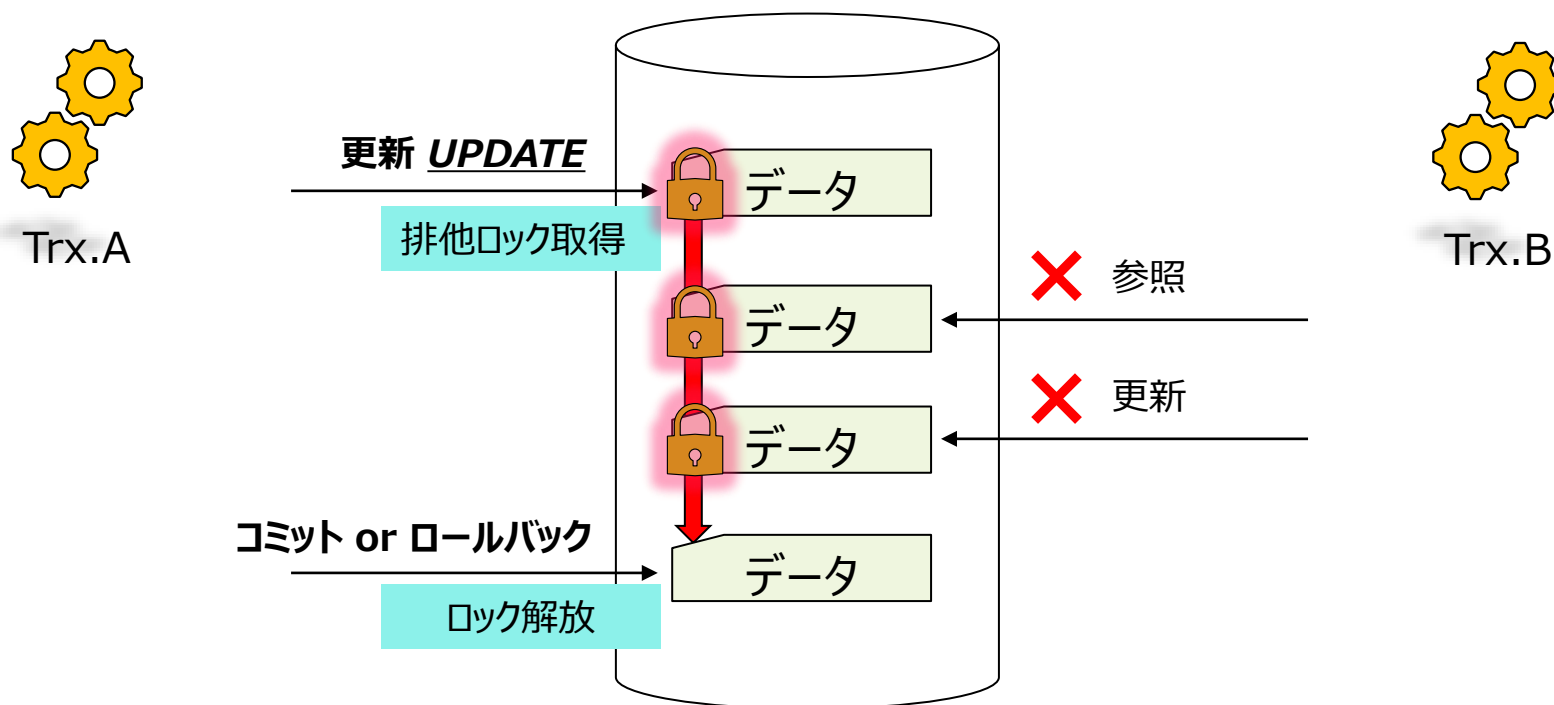
物理ロックの種類 – 共有ロック

自トランザクションがそのデータを読み出していることを、他のトランザクションに示すために取得するロック。他トランザクションは参照のみ可能。



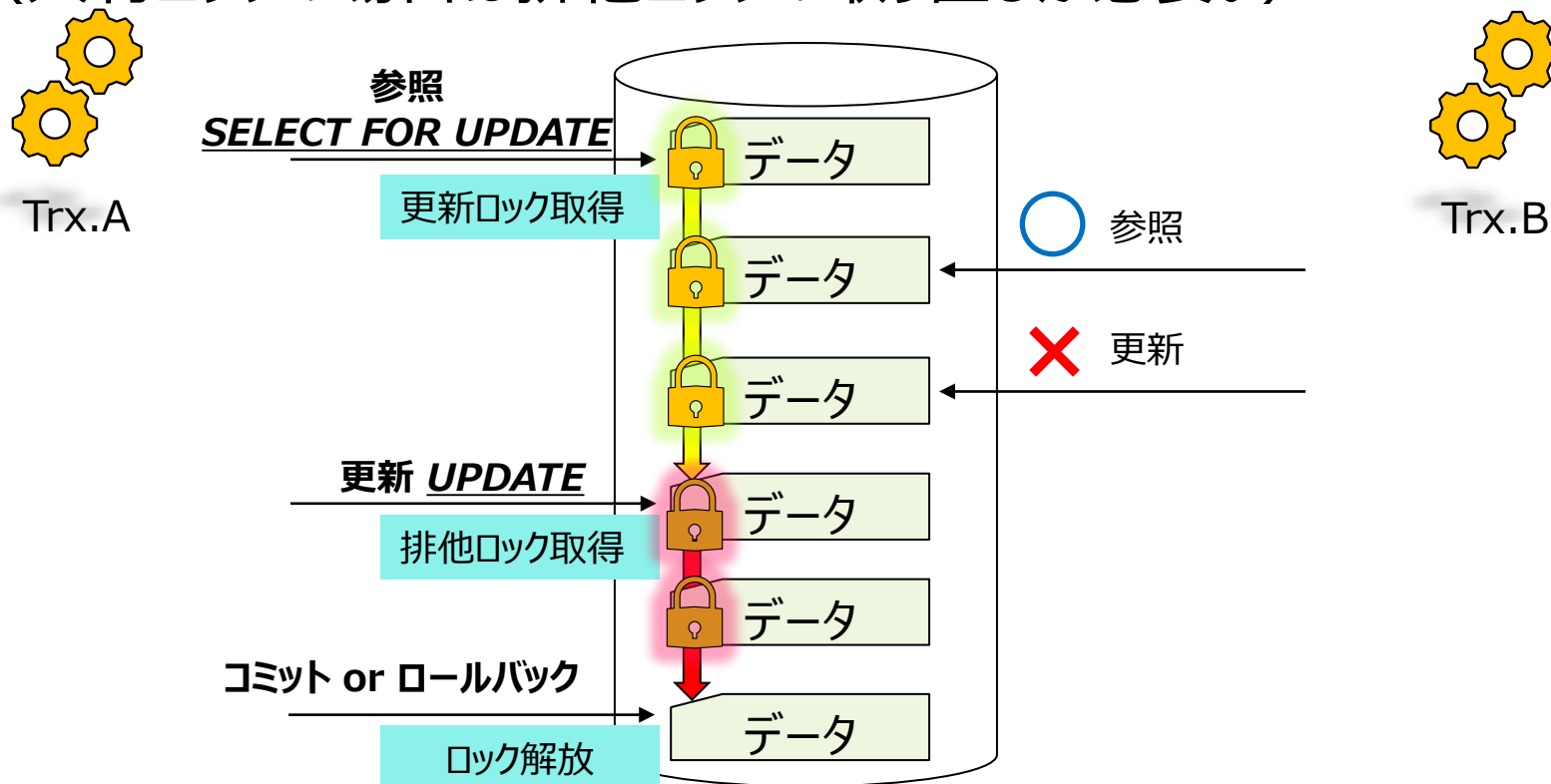
物理ロックの種類 – 排他ロック

自トランザクションがそのデータを更新していることを、他のトランザクションに示すために取得するロック。他トランザクションは参照も更新もできない。



物理ロックの種類 – 更新ロック

自トランザクションがそのデータを後で更新する予定であることを、他のトランザクションに示すために取得するロック。他トランザクションは参照のみ可。更新を実行すると、そのまま排他ロックに格上げされる。
 (共有ロックの場合は排他ロックの取り直しが必要。)



ロックの互換性(1/2)

あるトランザクションがロックを取得している（ロックを「かける」といいます）とき、他のトランザクションはどのロックを取得することができるのか？という関係を、**ロックの互換性**と呼び、下表のようになっています。

自トランザクションが新たに取得しようとしているロック	他のトランザクションによりすでにかけているロック		
	共有	更新	排他
共有	○	○	×
更新	○	×	×
排他	×	×	×

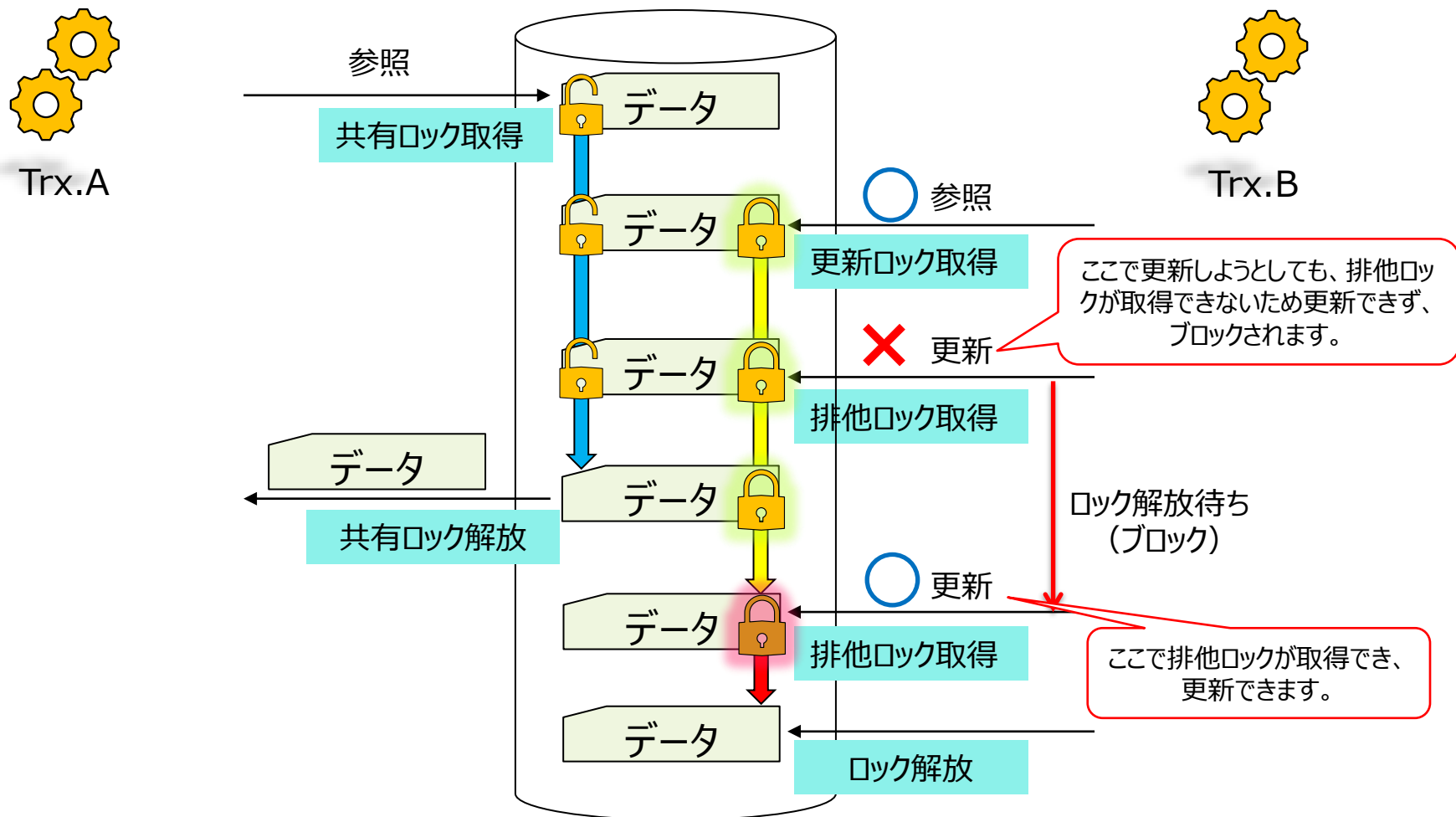
○：取得できる

×：取得できない

排他ロックがかけられているとき（更新中）、共有ロックが取得できない（そのレコードが見られない）のは経験に反すると感じる方もいらっしゃるかもしれません。原理的には取得できないのですが、実用上不便なのでDBMSメーカーが工夫して、共有ロックを取得できているかのように動作させています。

ロックの互換性(2/2)

ロックの互換性をイメージで示したのが下の図です。前ページの表を参照しながら、あるロックがかかっているとき、他のロックの取得はどうなるのか確認してください。



- トランザクションを同時実行したときの独立性を決定するレベル。
- 分離レベルが高いほど色々な不都合な読み取りを防ぐことができるが、スループットは低下する。

トランザクション分離レベル (ANSI/ISO SQL標準)	防止できる不都合な読み取り		
	Dirty Read	Non Repeatable Read	Phantom Read
SERIALIZABLE	○	○	○
REPEATABLE READ	○	○	×
READ COMMITTED	○	×	×
READ UNCOMMITTED	×	×	×

高
↑
分離レベル
↓
低

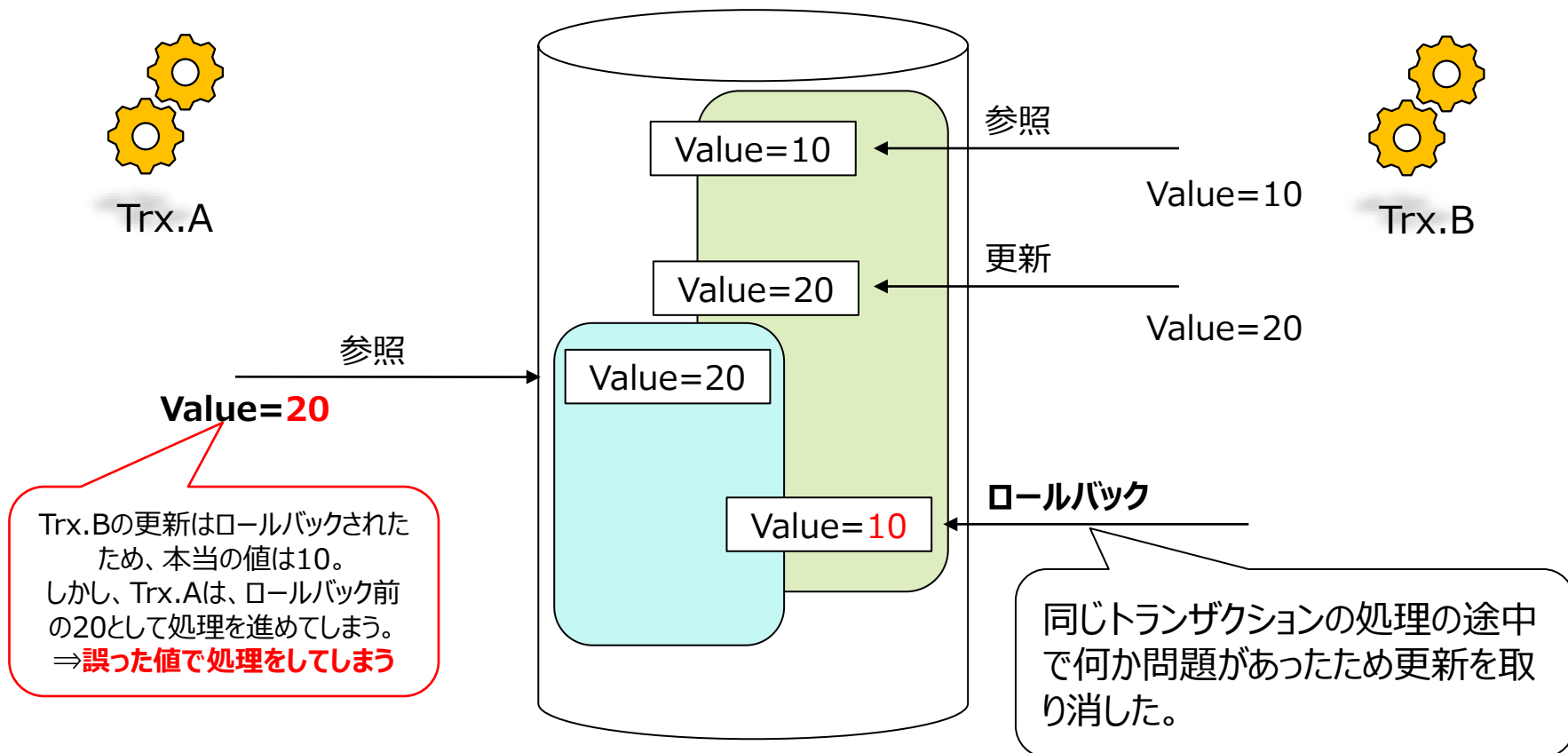
○ : 防止できる
× : 防止できない

- **SERIALIZABLE**
 - 直列化可能。トランザクションを時間的重なりなく逐次実行。
- **REPEATABLE READ**
 - 読み取り対象（他のトランザクションで変更されることはない）のデータを常に読み取る。
- **READ COMMITTED**
 - 確定した（コミット済）最新データを常に読み取る。
- **READ UNCOMMITTED**
 - 確定していないデータも読み取る。

「不都合な読み取り」について、次ページから説明していきます。

不都合な読み取り - Dirty Read

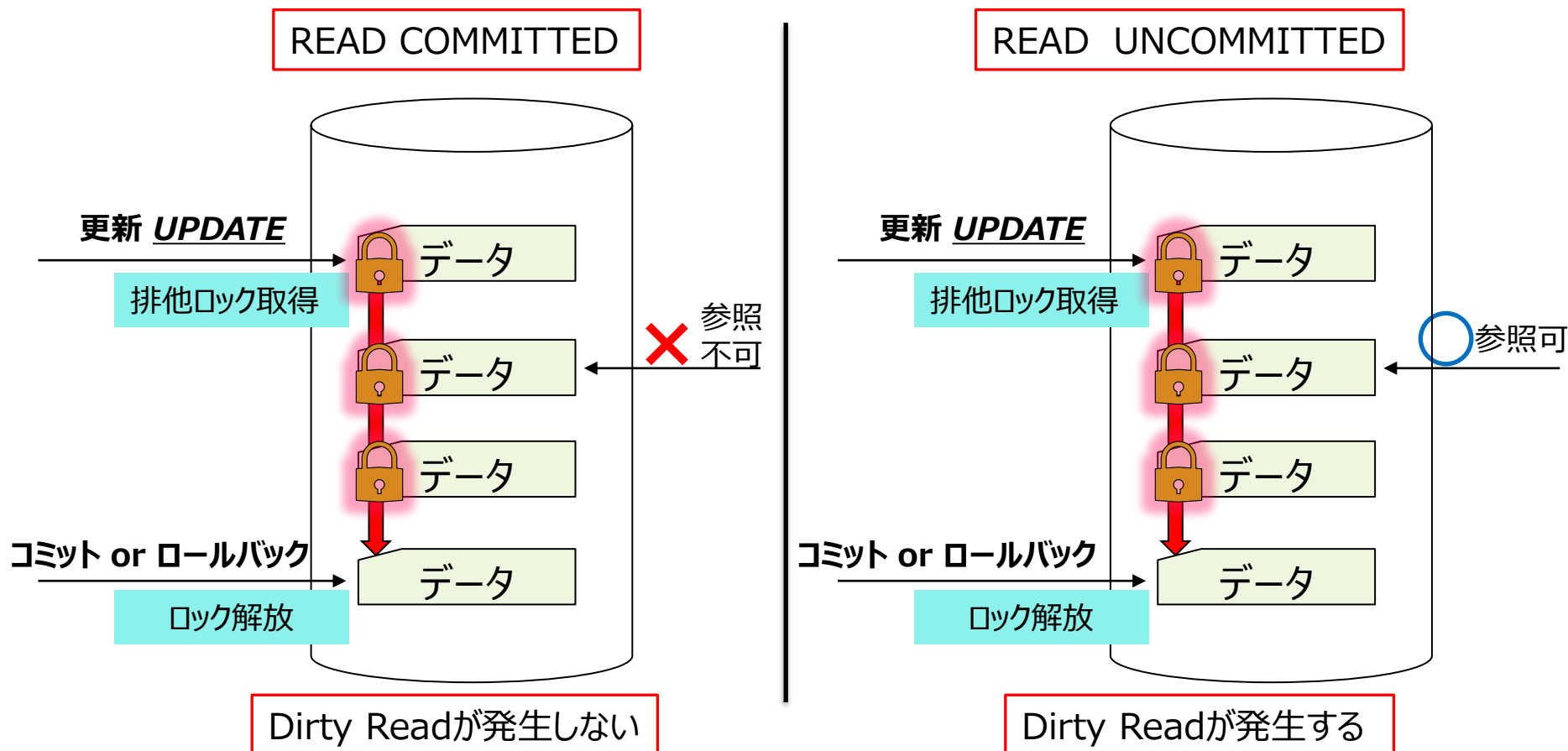
他トランザクションで更新しているコミット前のデータを読み出してしまう。**誤った値で処理**が行われる可能性がある為、**絶対に避けなければなりません。**



補足：ロックの互換性とDirty Read

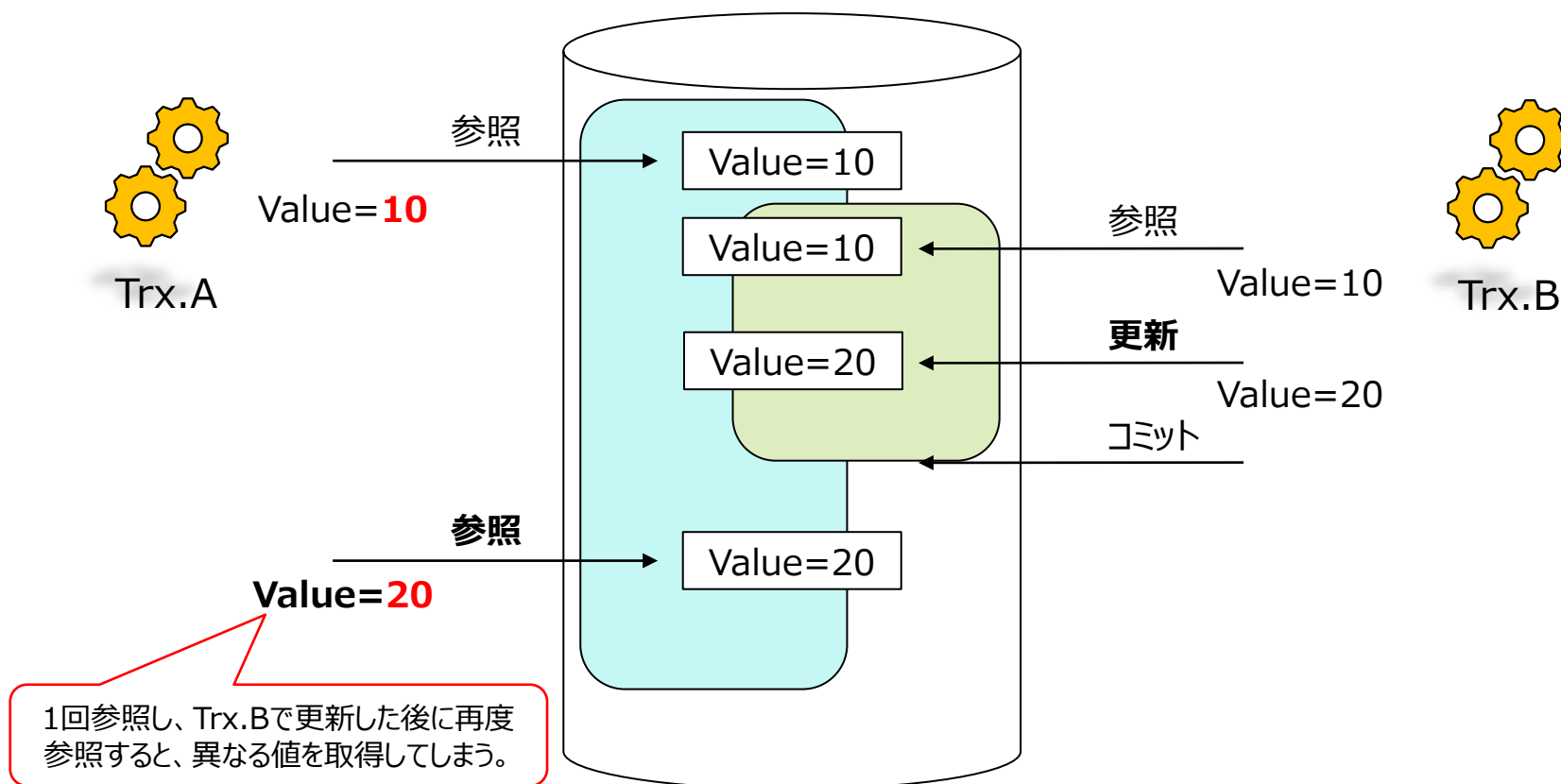
前掲のロックの互換性では「排他ロック中は共有ロックは取得できない」となっていました。この為、Dirty Readは起きようがないと思われるかもしれませんが。

実は、前掲のロックの互換性はREAD COMMITTEDを前提としていました。READ UNCOMMITTEDでは、ロックの互換性が変わり「排他ロック中でも共有ロックは取得できる」となります。

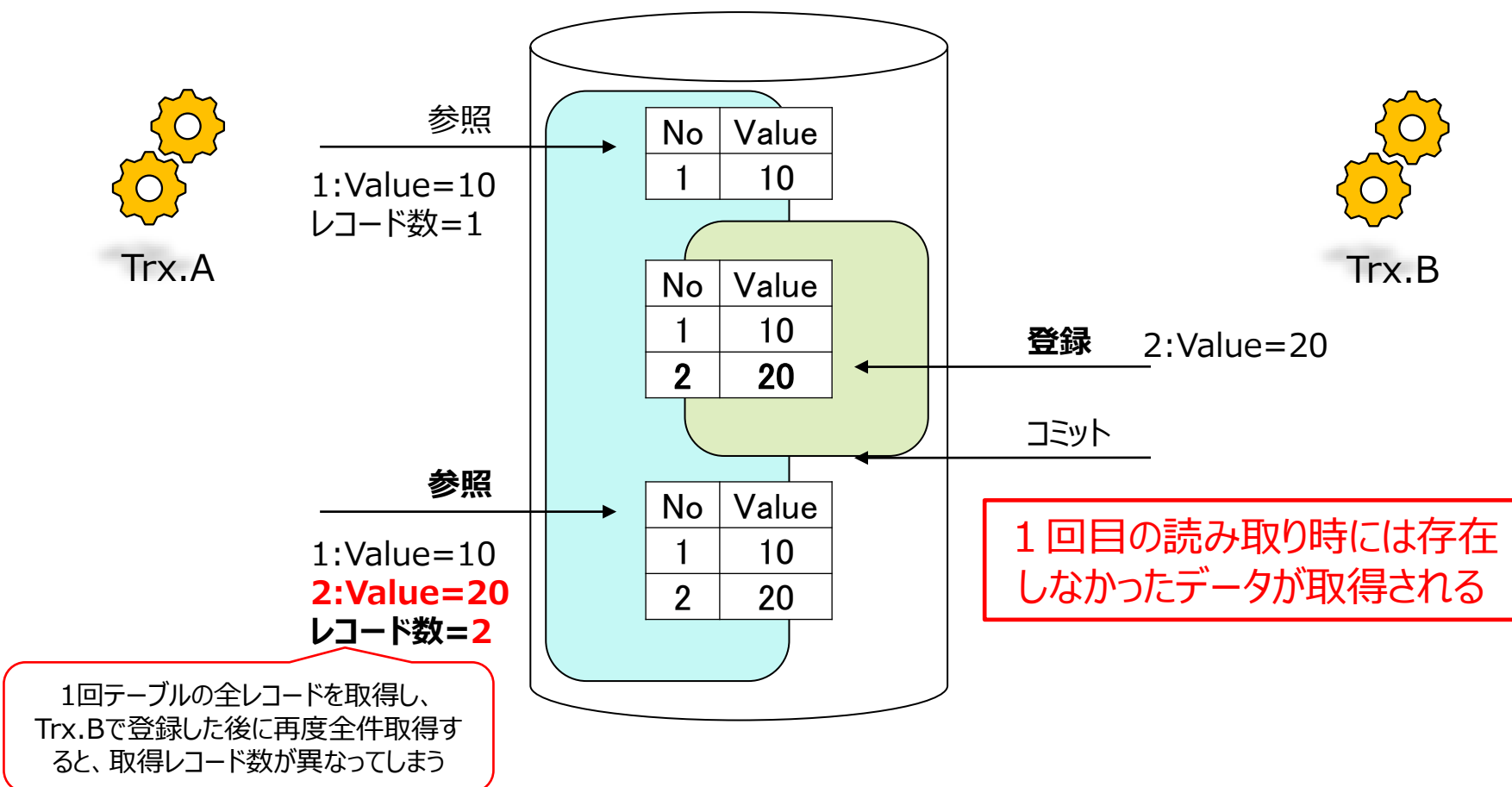


不都合な読み取り - Non Repeatable Read (反復不能読み取り)

あるトランザクションで同じ行を2回読み込んだ際、1回目と2回目のデータが異なってしまいます。



あるトランザクションにおいて、1回目と2回目の処理でレコード件数が異なってしまう。



トランザクション分離レベルの選択

- 基本的にはREAD COMMITTEDが選択される。
- 殆どのDBMSでREAD COMMITTEDがデフォルトに設定されている。(Oracle, SQL Server, DB2, ...)

分離レベル ↑高 ↓低	トランザクション分離レベル (ANSI/ISO SQL標準)	防止できる不都合な読み取り		
		Dirty Read	Non Repeatable Read	Phantom Read
	SERIALIZABLE	○	○	○
	REPEATABLE READ	○	○	×
	READ COMMITTED	○	×	×
	READ UNCOMMITTED	×	×	×

○ : 防止できる
× : 防止できない

検索時点でコミット済のデータのみを参照する。

例えばOracleの場合、以下のような仕組みで実現されています。

- 更新中のレコードはUNDO表領域から取得する。
（更新中の表領域からは読み込まない。）

⇒このため、未コミットの情報を取得することが無いので、
Dirty Readが発生しない。

なぜ、READ COMMITTEDが選択されるのでしょうか？

1. 最低限Dirty Readの発生は防ぐべき。

- 取り消される可能性のあるデータを前提に処理を実行すると、不整合が発生する恐れがある。

⇒ READ UNCOMMITTED はダメ

2. READ COMMITTED より高い分離レベルの場合、スループットが著しく低下する。

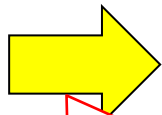
⇒ 性能上、REPEATABLE READ, SERIALIZABLE は好ましくない（避けたい）

⇒ READ COMMITTEDでは防げない不都合な読み取りが実際の業務処理で発生しうるのか考えてみる。

Non Repeatable ReadやPhantom Readは発生しているのか？



業務処理では、参照⇒チェック⇒更新の順で処理が実装されるため、同じトランザクションの中で同一データを検索することはない（Non Repeatable ReadやPhantom Readは発生しない）



排他制御的にも性能的にもバランスの取れる
READ COMMITTEDで問題ない。

READ COMMITTEDを選択した場合、「同じトランザクションの中で同一データを検索する」ような設計をしてはならない、ということでもある。

- Oracleでサポートしているトランザクション分離レベルは、READ COMMITTED と SERIALIZABLE。
- MySQLではデフォルトの分離レベルがREPEATABLE READとなっている。
- DBMSによってはロックエスカレーションがデフォルトで実行されるようになっている。(SQL Server、DB2など)
 - ロックエスカレーション
1つのテーブルで多くのレコードがロックされるなどの要因で、レコード単位でロックを管理するより、テーブル単位でロックを管理する（テーブル丸ごとロックする）方が効率的な場合に、レコード単位のロックがテーブル単位のロックに移行すること。
スループット低下やデッドロックにつながることもあるので、状況に応じて設定を変える必要がある。

DB2 9.7 & Oracle 11g 比較ハンドブック
翔泳社
翔泳社 著
(電子書籍)



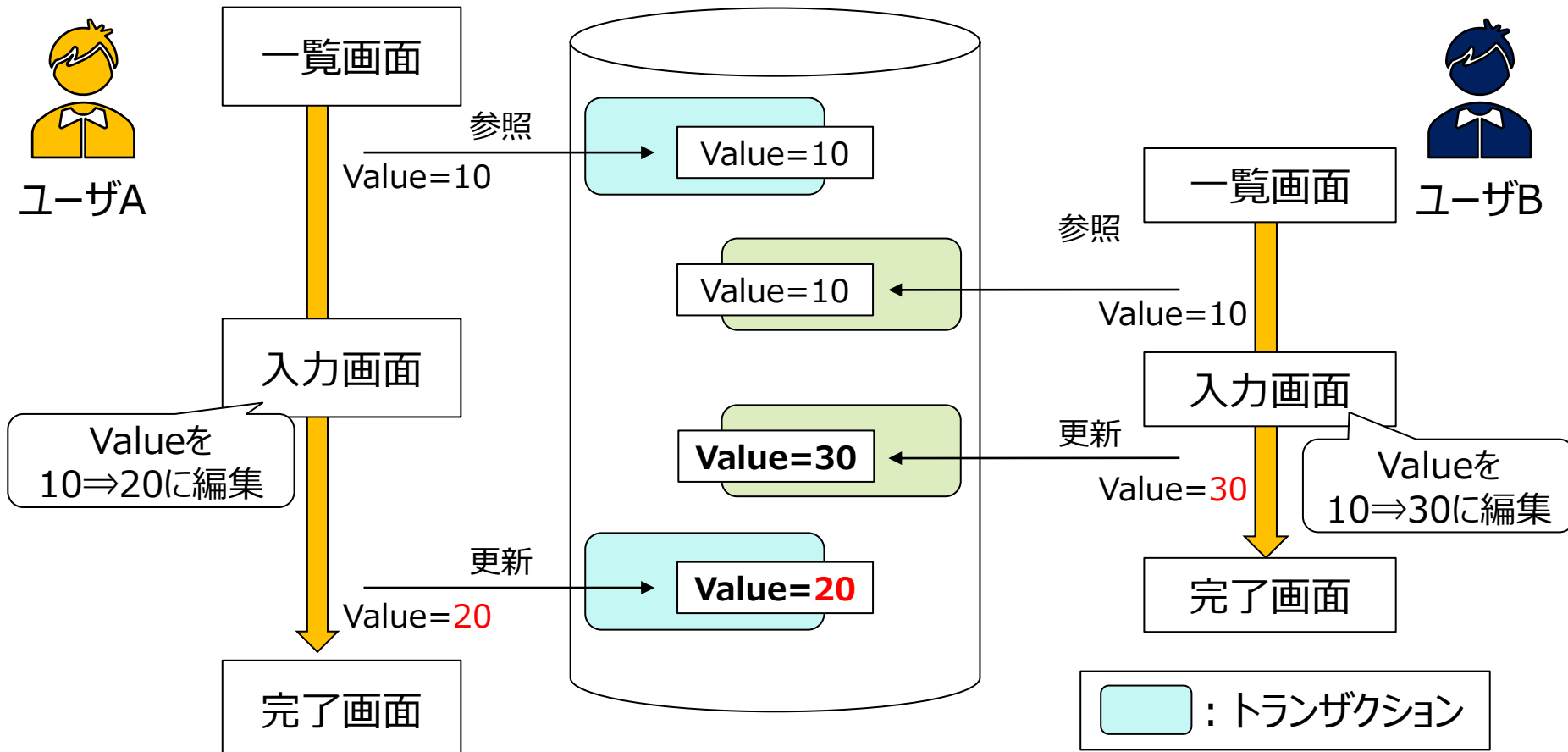
<https://enterprisezine.jp/sepub/ibmdownload>
2021年11月26日16時の最新情報を取得

排他制御を実現するための技術要素 ～ 論理ロック ～

- 物理ロック
 - DBMSで実現されているロック機構。
 - 共有ロック、更新ロック、排他ロック …
- 論理ロック
 - アプリケーションレイヤーで実現するロック機構。
 - 複数のトランザクションに跨るロックを実現。
 - 楽観的ロック、 悲観的ロック

物理ロックだけだと何が起こる？ (1/2)

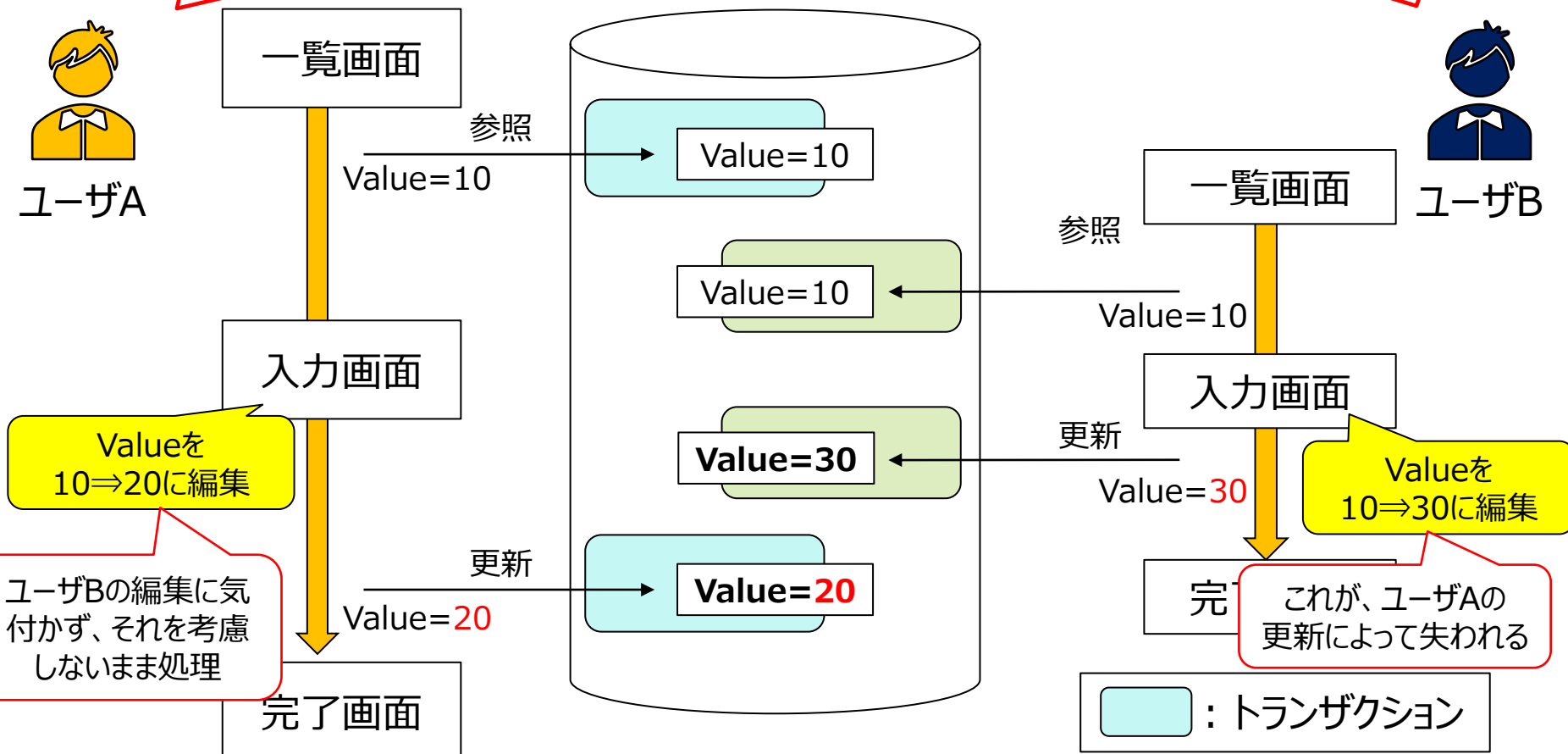
物理ロックは1トランザクションの中での制御です。しかし、多くのウェブアプリは1業務が複数トランザクションにまたがります。このため、物理ロックだけだと不都合が起こります。



物理ロックだけだと何が起こる？ (2/2)

最新のデータを把握せずに処理してしまう

ユーザBの更新が失われてしまう



複数のトランザクションから構成される業務処理について、下記を目指す。

- データ整合性の維持
- スループットの最大化

論理ロックの設計パターンは2つあります。

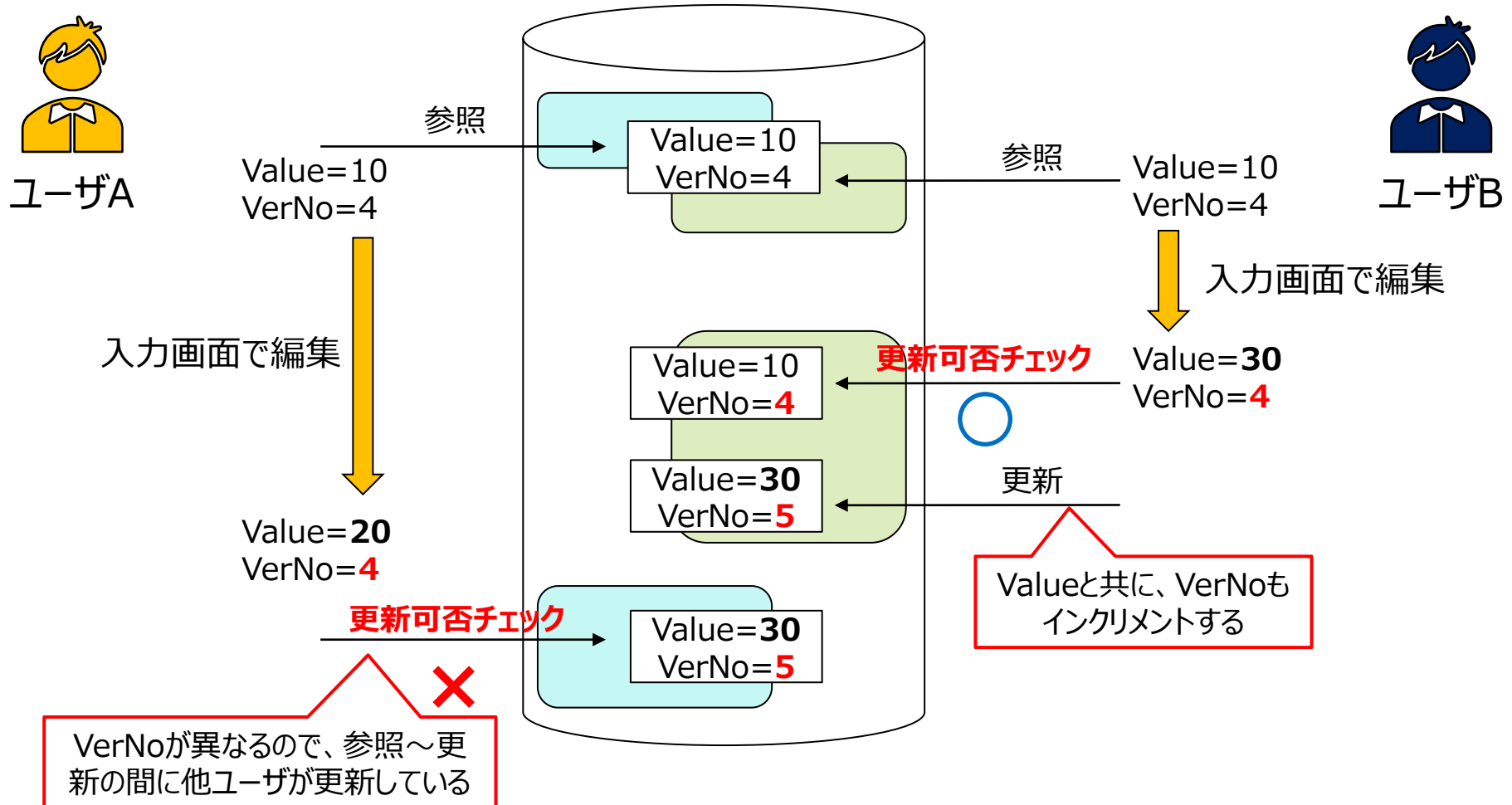
いずれも作り込みが必要です。

- 楽観的ロック方式
 - 更新を実行するときに、既に他のユーザが更新していないか確認する。
- 悲観的ロック方式
 - ユーザが編集作業を始める前に、他のユーザが更新できないようにロックしてしまう。

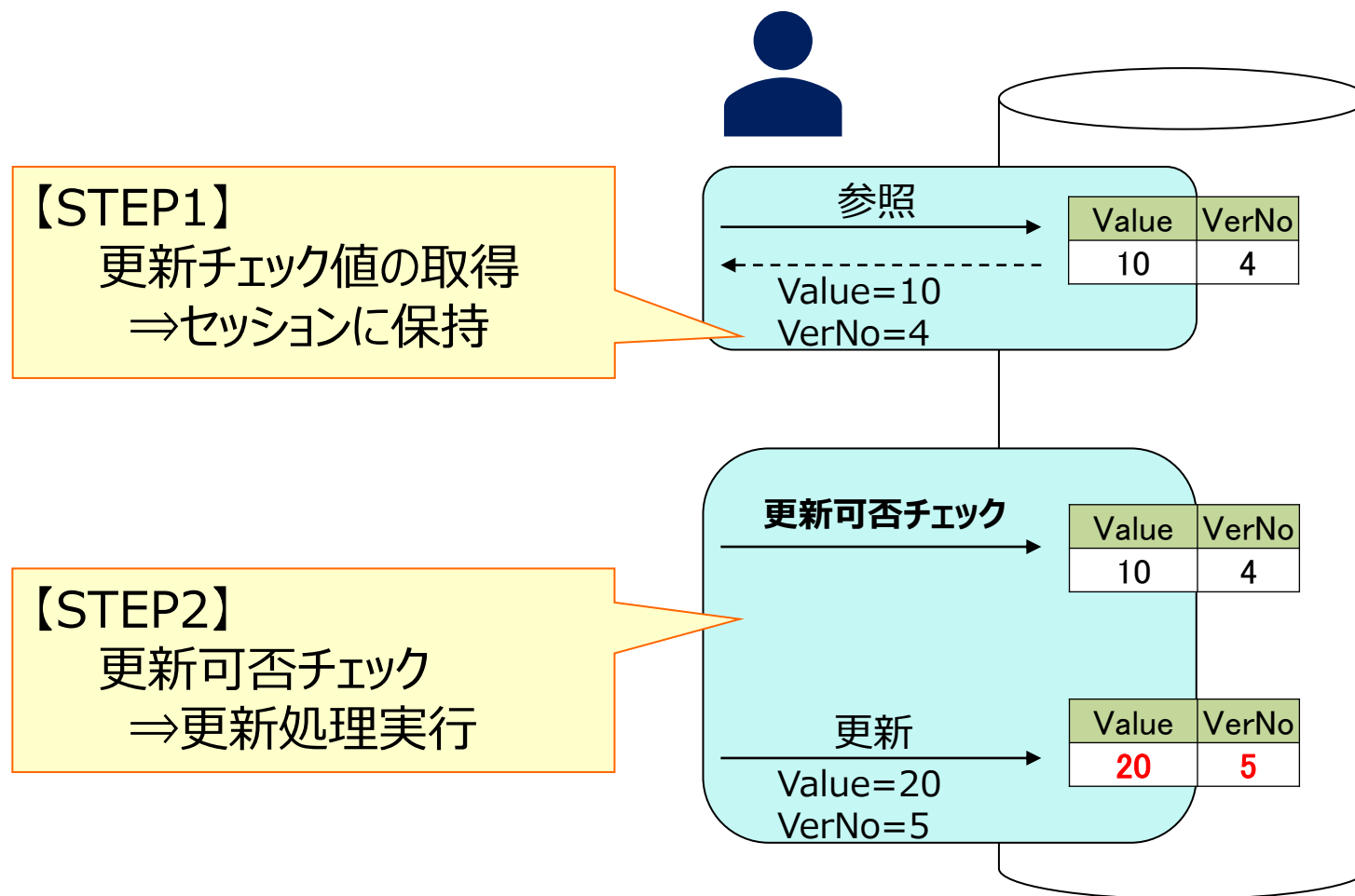
楽観的ロック方式からみて行きましょう。

楽観的ロック方式

楽観的ロック方式の概要です。編集後、更新を実行する際に更新可否チェック（他のユーザが更新していないかの確認）を行います。

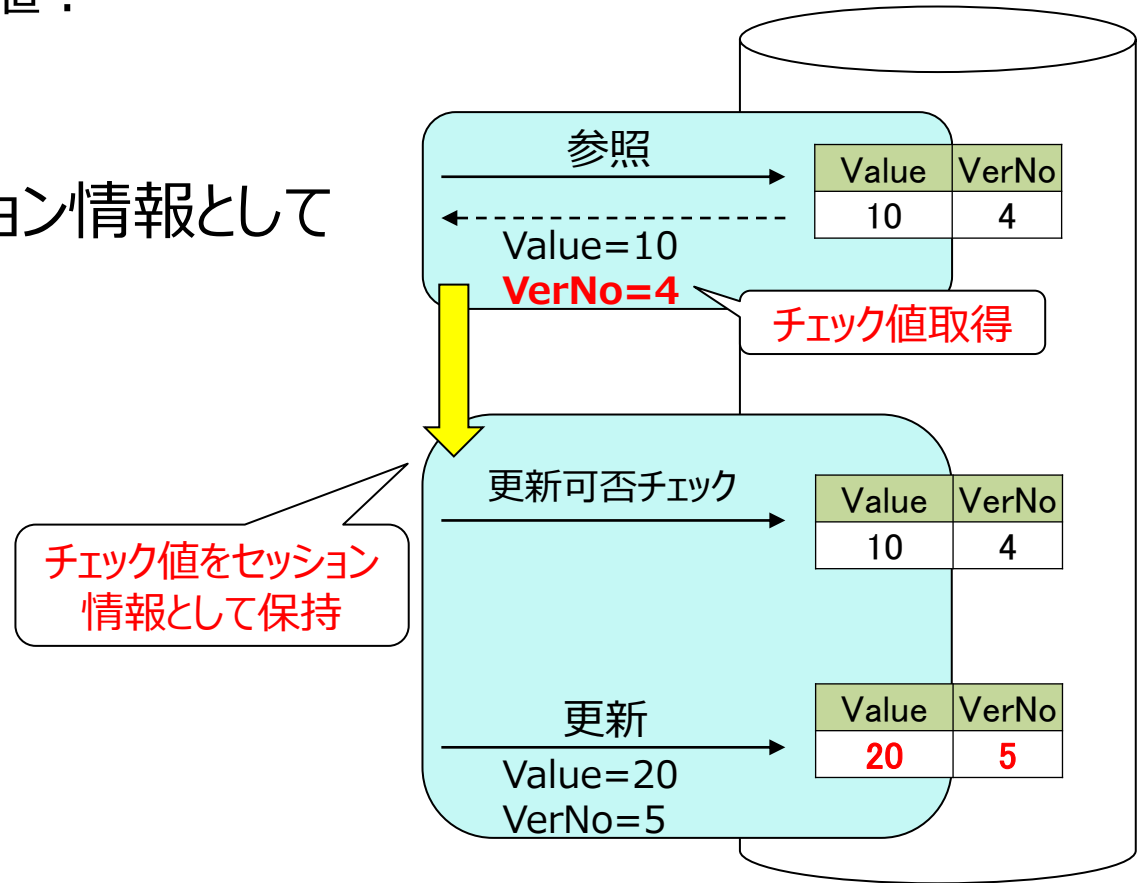


楽観的ロック方式の処理STEP



【STEP1】更新チェック値の取得

- 業務処理冒頭で更新チェックに利用する値を取得する。
更新チェックに利用する値：
 - バージョン番号
 - 最終更新日時、...
- 取得した値はセッション情報として保存する。



更新チェック値としてレコードごとに**バージョン番号**を持たせる。
データ取得時とデータ更新時、このバージョン番号が同じなら更新可能とする。

- メリット
 - 最終更新日時（次ページで紹介）のように同期や精度の問題がない。
- デメリット
 - バージョン番号カラムをテーブルに追加する必要がある。

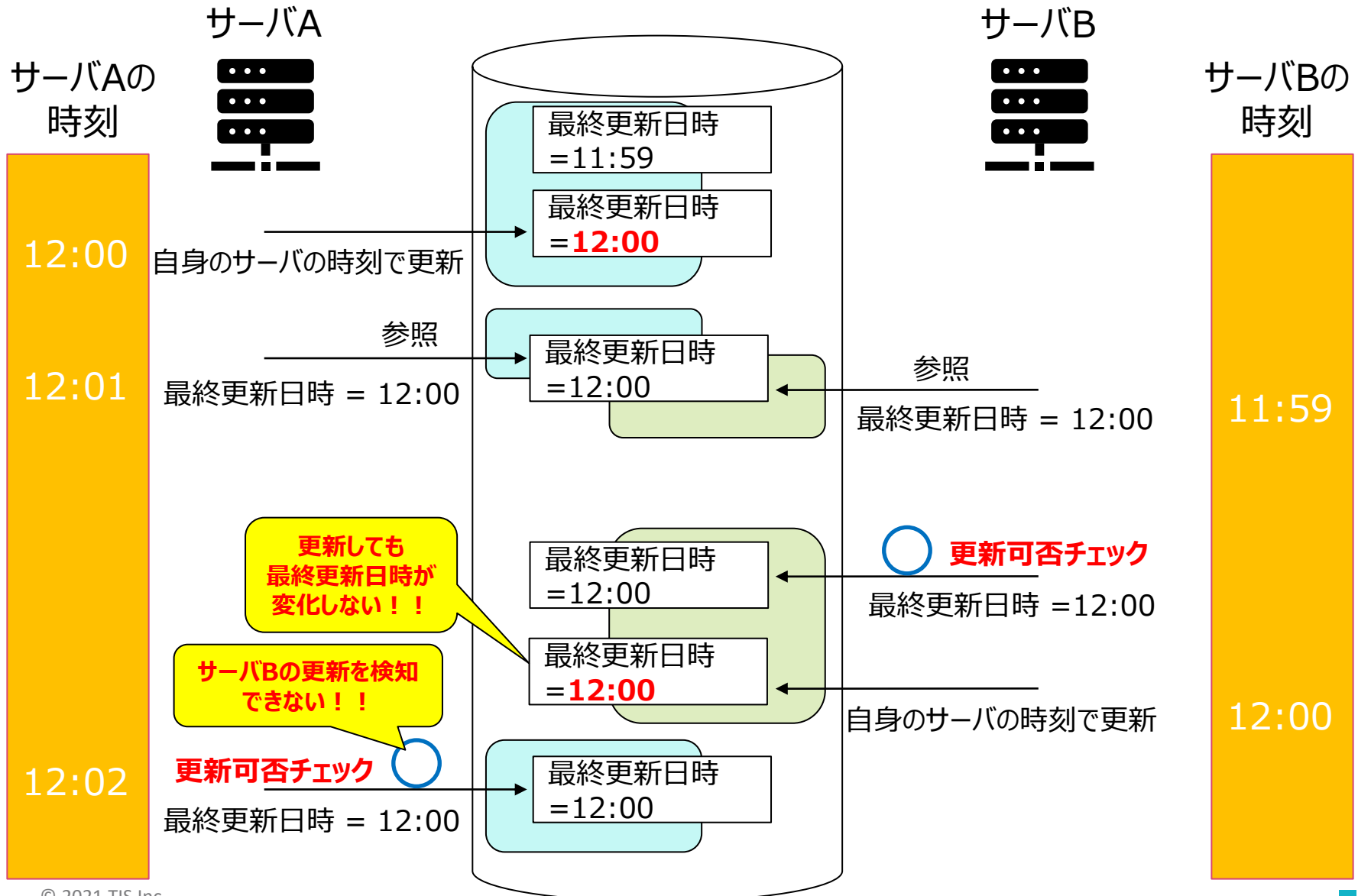
顧客ID	顧客名	<u>VerNo</u>
12345	山田太郎	24
23456	山田次郎	18
34567	山田三郎	3

レコードごとの**最終更新日時**を利用する。
データ取得時とデータ更新時で最終更新日時が同じなら更新可能とする。

- 日時には、同期と精度の問題が付きまとう。
 - 2つの更新を区別できないという問題。
 - 精度より短い間隔で更新が行われるとそれらを区別できない。
 - 同期については次ページで紹介。
- 敢えてこちらを採用するメリットは薄いですが、既存のパッケージ製品で採用している場合がある。

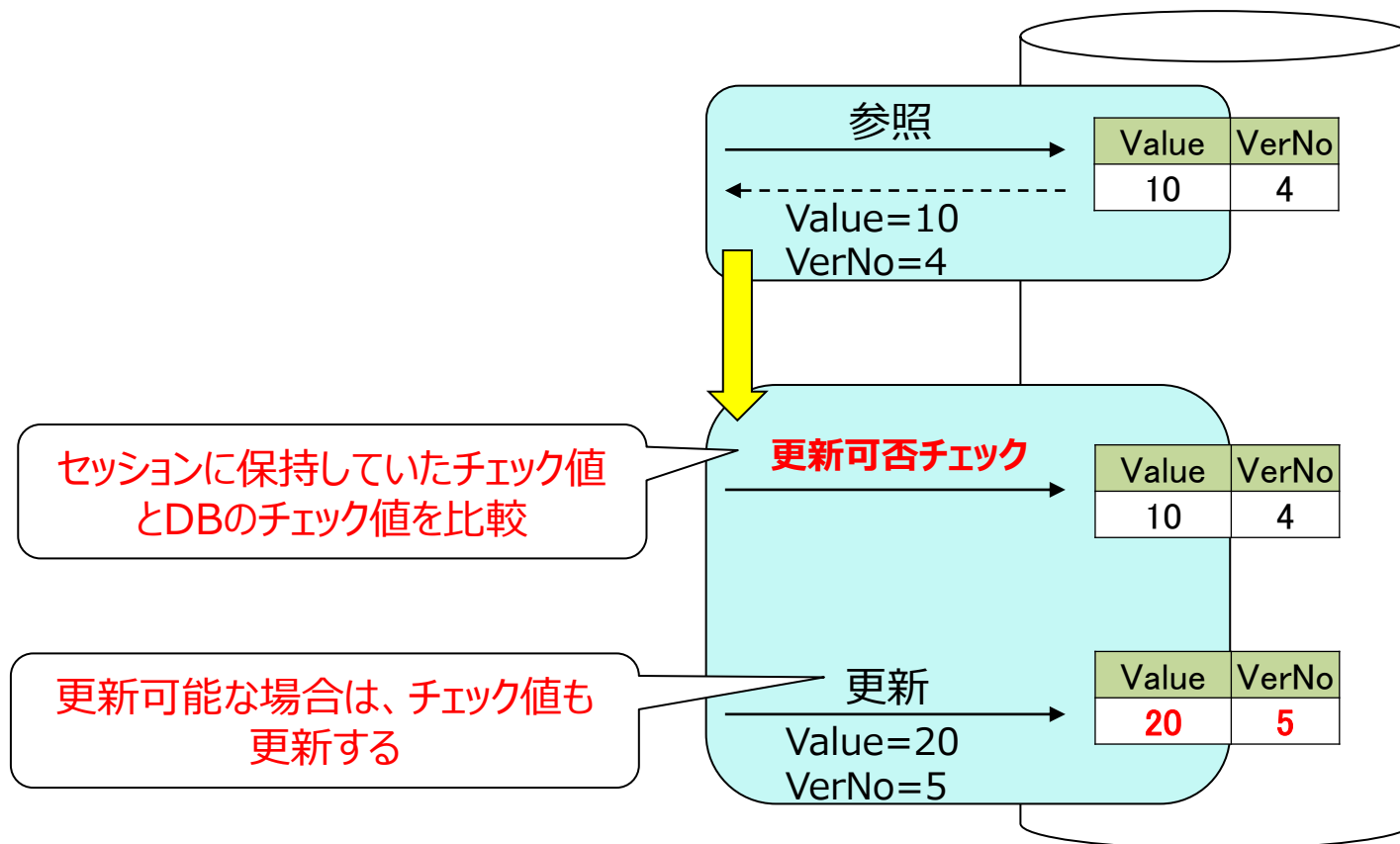
顧客ID	顧客名	最終更新日時
12345	山田太郎	16/07/18 12:32:16.005
23456	山田次郎	16/07/01 10:10:03.015
34567	山田三郎	16/07/30 22:33:17.105

最終更新日時を使ったときの同期の問題

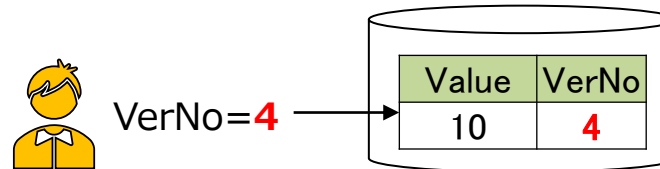


【STEP2】更新可否チェック(1/2)

更新時に更新チェック値の比較を行い、更新可否を判定する。

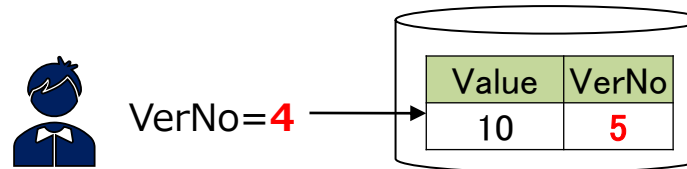


【STEP2】更新可否チェック(2/2)



更新チェック値が一致 ⇒ 誰も更新していない … **更新可能**

⇒ 更新処理へ



更新チェック値が不一致 ⇒ 誰かが先に更新した … **更新不可**

⇒ 排他エラーとする

更新チェック値のチェック方法は二通りあります。

更新可否のチェック方式(1/2)

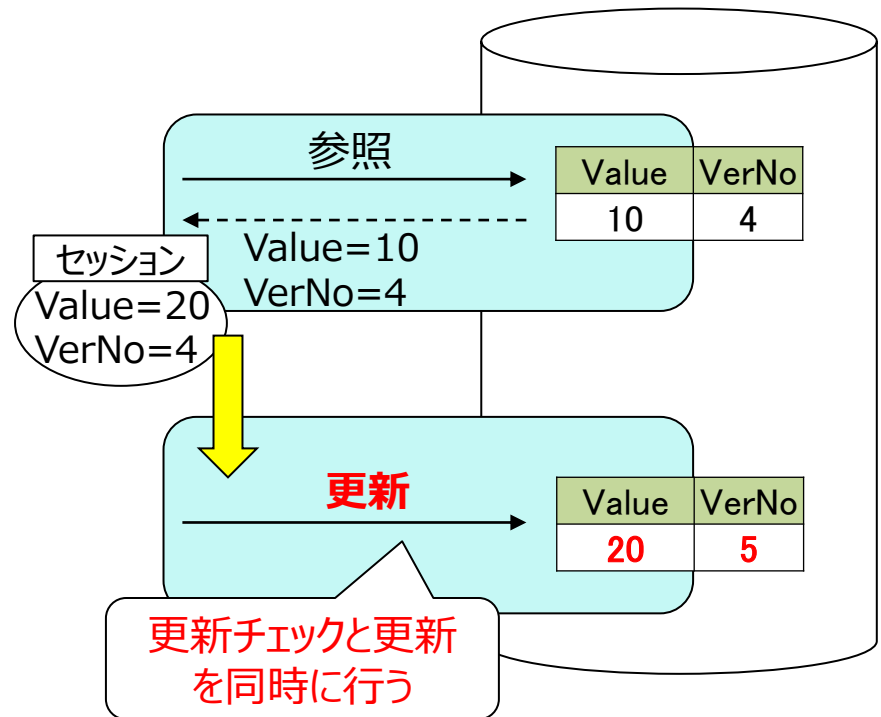
UPDATE文のWHERE句に更新チェック値を付与

更新件数が0件の場合は、更新衝突（排他エラー）とみなす。

バージョン番号が更新されていなければ、
WHERE句の条件に一致する。

```
UPDATE 顧客TABLE  
SET  Value='20', VerNo=5  
WHERE ID='12345' AND VerNo=4
```

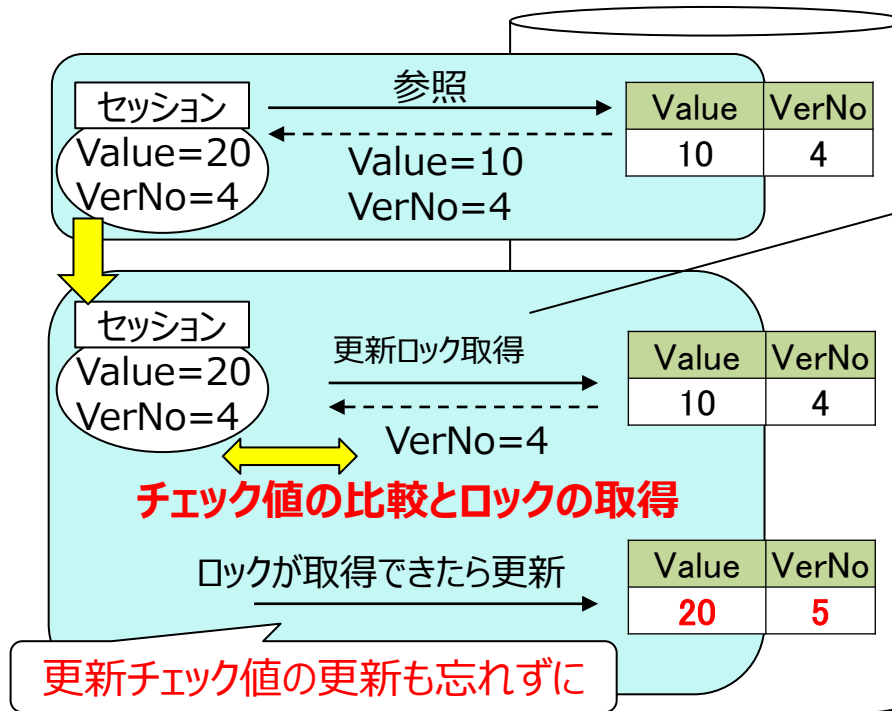
更新チェック値も
合わせて更新



更新可否のチェック方式(2/2)

更新チェック値が変更されていないかチェックしロックを取得

セッションに保持していた更新チェック値で検索とロックを行い（更新ロックの取得を試みる）、レコードが取得できれば更新を行う。



チェック値比較時に更新ロックを取得する。
以下はOracleの場合の例。

```
SELECT * FROM 顧客TBL  
WHERE 顧客ID=12345 AND VerNo=4  
FOR UPDATE
```

WHERE句に保持していた更新チェック値を指定

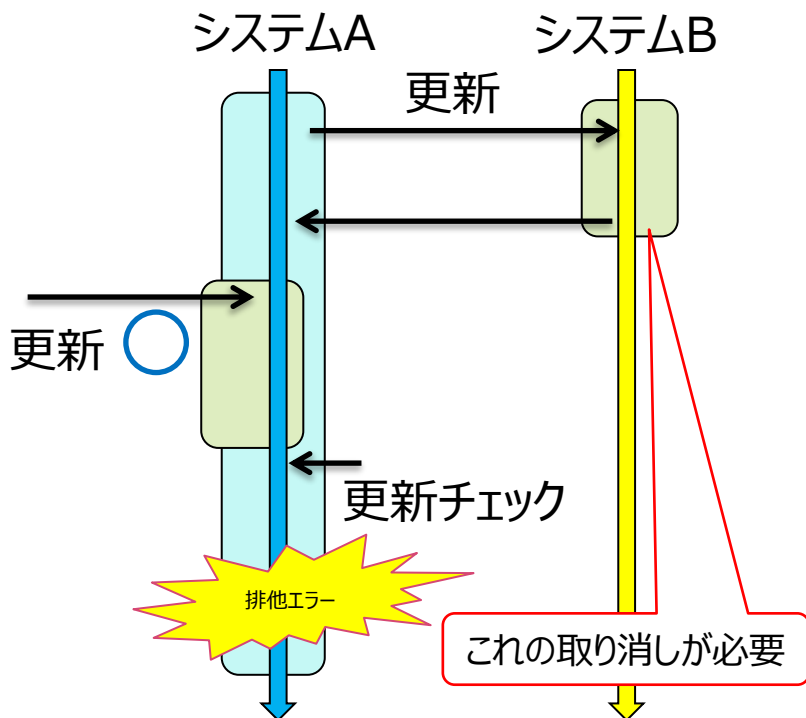
「FOR UPDATE」を付けることで更新ロック取得となる

更新可否チェック方式による違い

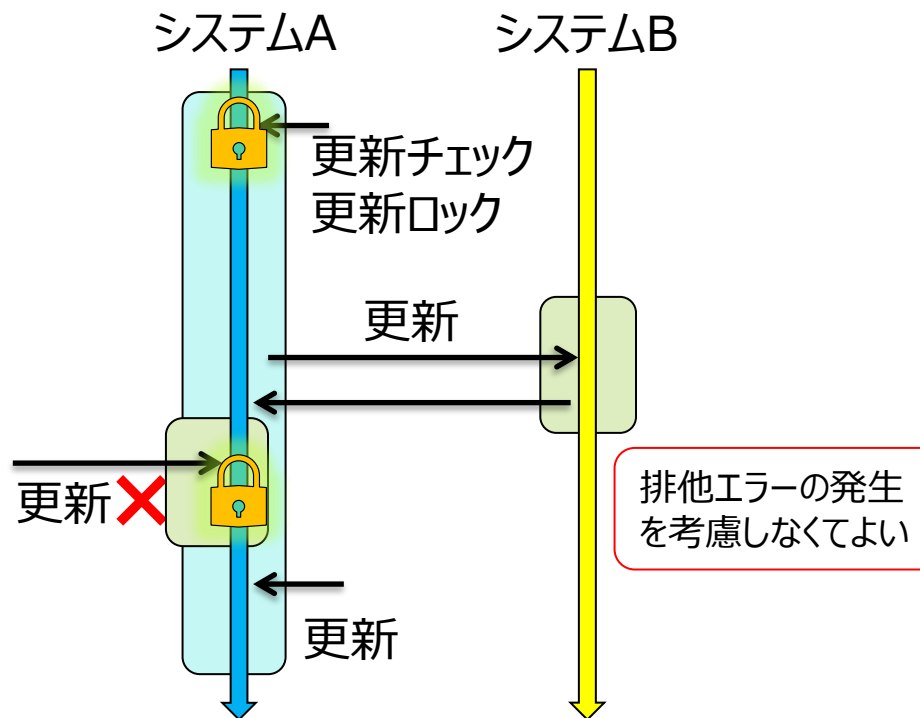
UPDATE文のWHERE句に更新チェック値を付与して、更新実行時にあわせて更新可否をチェックする方式は、実際に更新を実行するまでの間に、他のトランザクションが対象のレコードを更新できる。このため、更新を実行するまでの間に行った処理が無駄になることがある。

例えば、他のシステムに処理を依頼したあと、排他エラーが発生した場合、依頼した処理の取り消し処理が必要になる。

更新実行時に更新可否チェックを行う方式

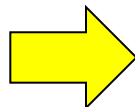


更新ロックを取得する方式

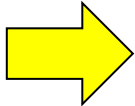
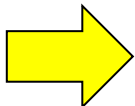


メリット

楽観的ロック方式の方が後述の悲観的ロック方式よりも実現コストが低い。

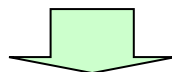
- ロックの仕組みづくりや、それを利用する業務プログラムの実装が容易。
-  • 汎用フレームワークが楽観的ロック方式の仕組みを提供している場合があり、その場合はさらに実装コストを低減できる。

デメリット

- データの更新衝突が発生した場合、通常は共通エラー画面に遷移させるため、負けたユーザの入力データがロストしてしまう。
 入力が無駄になる（やり直しが必要）。
- データ取得時の更新有無チェック値を、セッションに保持しておく必要がある。
 サーバーのリソースを消費する。

- 複数のユーザが同時に同じデータを更新する可能性が低い場合。
- ユーザが入力したデータが無駄になっても、ユーザの損害が少ない場合。

業務要件上許されるのであれば、
楽観ロックが実現コスト面で有利

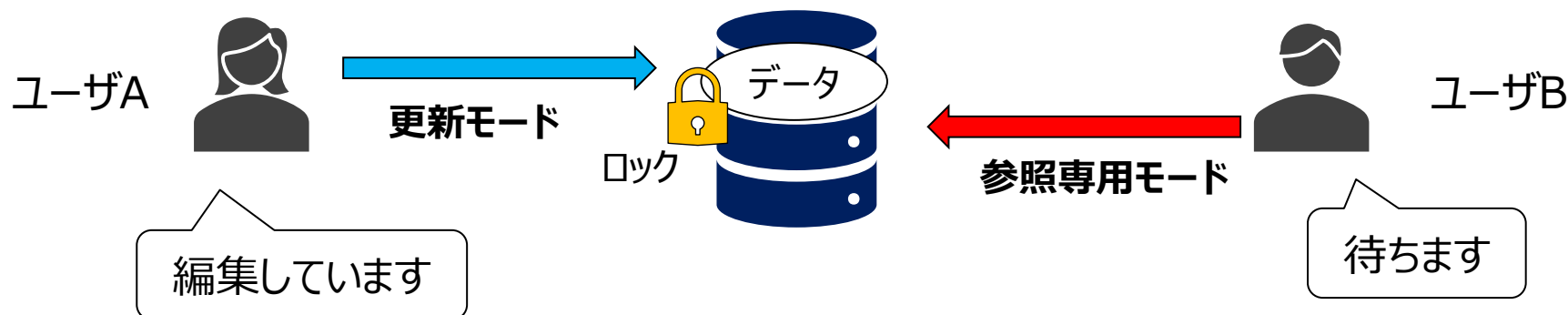


最初に考えるべきは、**楽観的ロック方式**

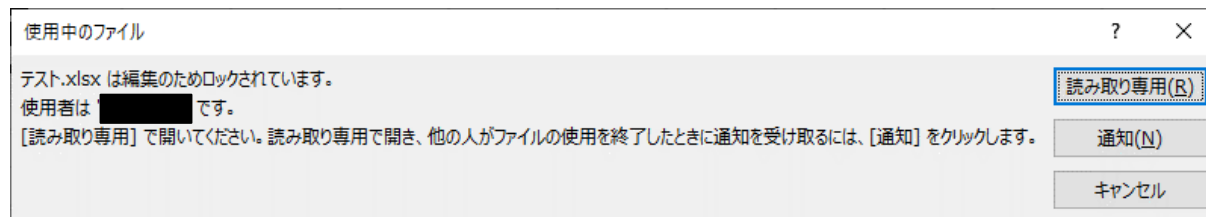
次に、悲観的ロック方式をみて行きましょう。

悲観的ロック方式(1/2)

悲観的ロック方式とは、操作開始時点でロックを取得し他のユーザが変更できないようにする方式です。



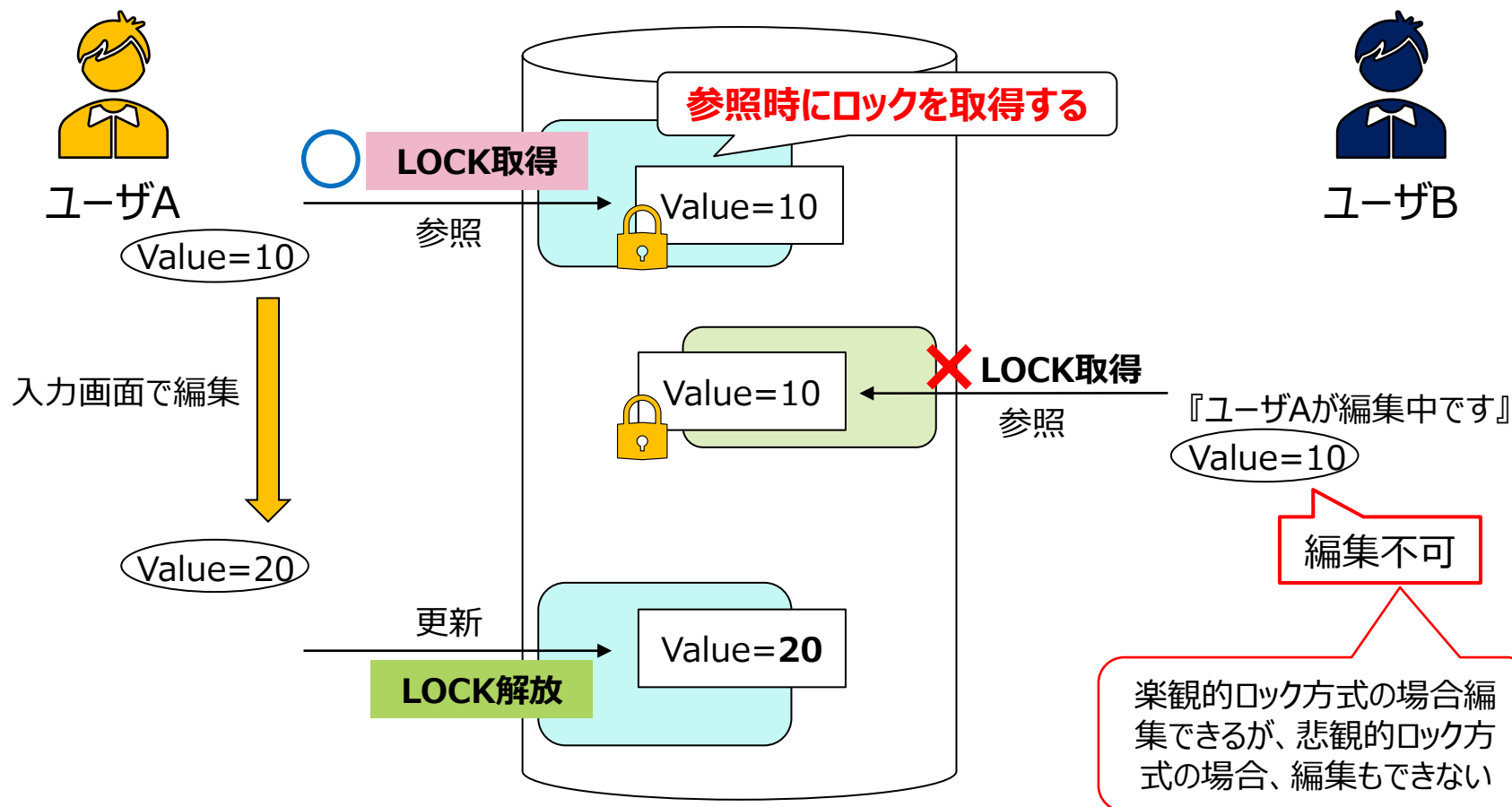
Excelみたいな感じ



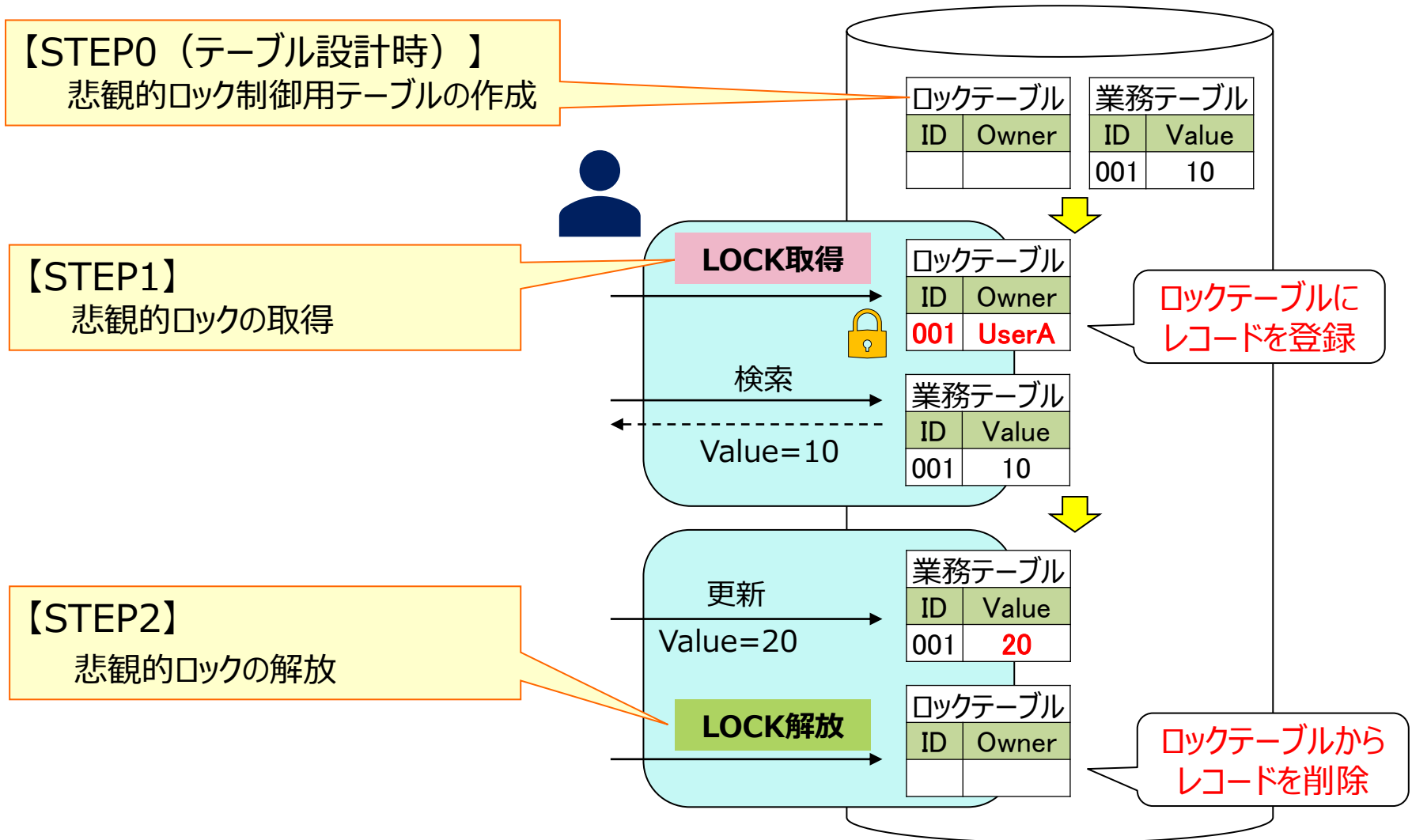
※使用者はマスクしています

悲観的ロック方式(2/2)

全体の流れは下図のようになります。楽観的ロック方式では、更新実行時に更新可否を確認していました。悲観的ロック方式では、データの参照時にロックを取得します。



悲観的ロック方式の処理STEP



ロックの取得・解放

悲観的ロック方式では、まず（データモデリング時に）悲観的ロック制御用のロックテーブルを作成しておきます。

どのレコードがロックされているのか、といった悲観的ロックの情報は、ロック対象レコードに紐付くロックテーブルのレコードで管理します。

ロックの取得・解放操作の実体は、ロックテーブルに対する以下の操作です。

- ロック取得：ロックテーブルにレコードを**登録**する
- ロック解放：ロックテーブルのレコードを**削除**する

企業テーブル（ロック対象レコード）

企業ID	企業名
12345	TIS
23456	INTEC
34567	AGREX

企業排他制御テーブル（ロックテーブル）

企業ID	ロックユーザ	有効期限
12345	A	09/11 12:02:11.005
34567	B	09/11 15:00:31.111

ロックを取得したユーザ

ロックの有効期限

WEBブラウザを利用する場合、悲観的ロックが確実に解放されるとは限りません。

- ロックしたまま別画面へ遷移した、ブラウザを閉じた、などなど。
サーバはこのようなブラウザのイベントを検知できない。

対策

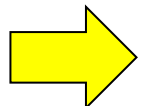
- ログイン、ログアウト時などにロックレコードを削除する。
- 有効期限を過ぎたロックレコードを、定時バッチなどで削除する仕組みを用意しておく。

これらを実現するために、ロックテーブルに有効期限を持たせる。

- 有効期限を過ぎたロックレコードを無視できるようにしておく
(有効期限が過ぎれば「横取り」が可能) 。
- 権限を持つユーザがロックを強制解除できる機能を用意する。

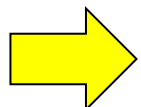
メリット

他のユーザによる更新処理の中断を、完全に防ぐことが出来る。

 ユーザの入力したデータが無駄にならない。

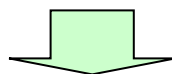
デメリット

- 悲観的ロックの仕掛けを一から検討して実装する必要がある。
- 確保され続けている悲観的ロックを解放する機構を用意する必要がある。

 作り込みの手間が発生する。

他のユーザによる更新処理の中断を、完全に防がなければならない場合

実現コストの面では楽観ロックの方が有利

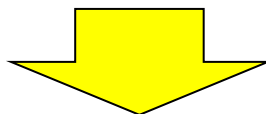


業務上必要な部分だけ、**悲観的ロック方式**

基本は楽観的ロックとして、業務上必要な部分があれば、悲観的ロックとする。

クレジットカード会員の入会管理システム

- 複数のオペレータが同一のデータを操作する可能性が高い。
⇒競合が発生しやすい。
- 入力画面の項目数が多く、入力完了までに時間がかかる。
⇒入力量が多く、ユーザの入力を無駄にできない。



楽観的ロックだと、競合が発生した場合に
時間をかけて入力した情報が無駄になってしまう。

各方式の比較

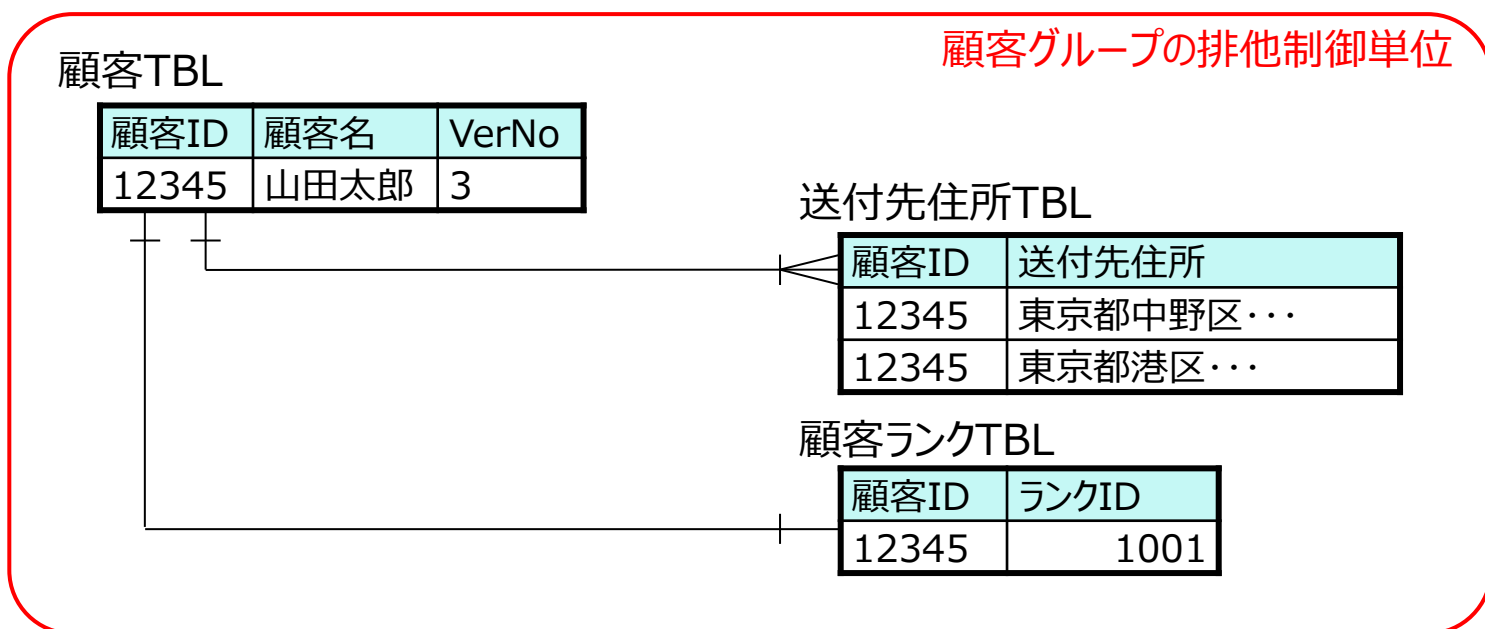
	楽観的ロック方式	悲観的ロック方式
メリット	<ul style="list-style-type: none">■ FWによってサポートされている場合は実現が容易	<ul style="list-style-type: none">■ 他のユーザによる更新処理の中断を、完全に防ぐことが出来る
デメリット	<ul style="list-style-type: none">■ 入力作業が無駄になる可能性がある■ 検索時の更新チェック値を保持しておく必要がある	<ul style="list-style-type: none">■ 仕掛けを検討、実装する必要がある■ 未使用のロックを解放する機構を別途用意する必要がある
使いどころ	<ul style="list-style-type: none">■ データ更新衝突の可能性は低いが、もし更新衝突が発生した場合には確実にデータ整合性を確保したい場合■ ユーザが入力途中のデータが無駄になっても、ユーザの損害が少ない場合	<ul style="list-style-type: none">■ 他のユーザによる更新処理の中断を、完全に防がなければならない場合

- 排他制御単位の設計
- 更新順序の設計

排他制御単位とは、まとめて排他制御を行うテーブルのグループです。データモデリング時に設計して決定します。

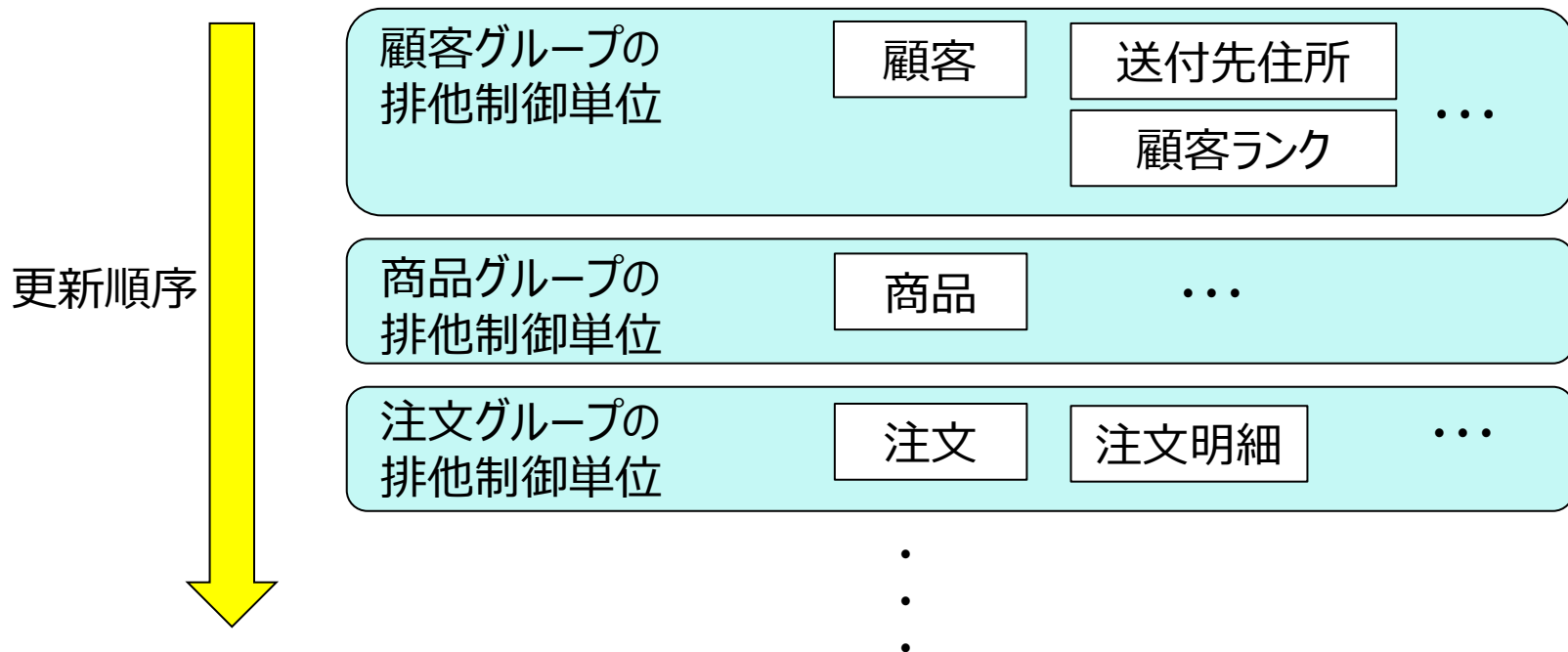
例えば、顧客、送付先住所、顧客ランクの3テーブルが業務上いつも同じトランザクションで更新されるのならば、各テーブルごとにそれぞれ排他制御を行う必要はなく、どれか一つのテーブルを代表としてそのテーブルに対して排他制御を行えば十分です（3つ一緒に更新されるから、例えば顧客テーブルの更新状況を、そのまま他の2つのテーブルの更新状況とみなせる）。

どのテーブルをまとめることができるのか、は業務要件とER図の観点から設計することになります。



デッドロックは、複数のトランザクションが勝手な順番でテーブル（排他制御単位）を更新することで発生します。

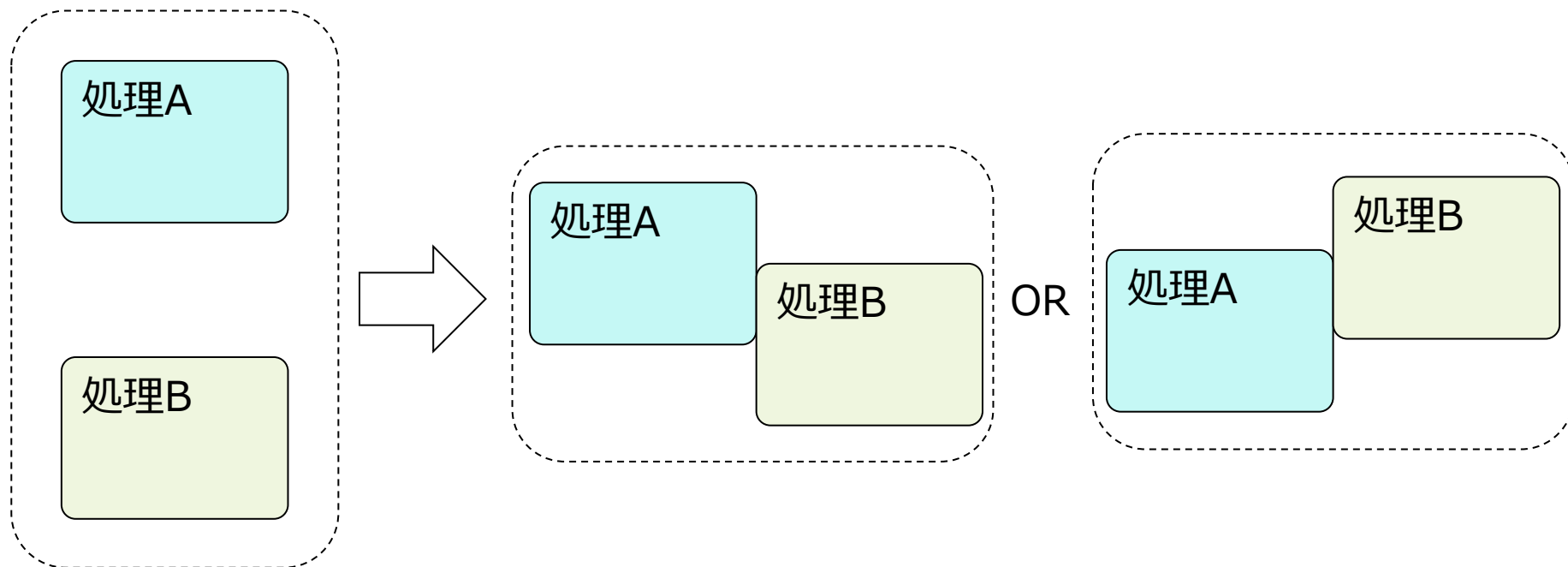
そこで、排他制御単位間での更新順序を決めることで、デッドロックを防止します。しかし、これはシステムの仕組みで強制できるものではなく、ルールです。したがって、**全ての開発者が更新順序を意識し、その順序を守る必要があります。**



排他制御設計の考え方

排他制御方式の設計手順(1/2)

- 競合が発生し得るあらゆる状況について排他制御方式を検討する。
- 具体的には、処理間の時間間隔が長い状況から競合パターンを考え始め、次第に短い状況へ踏み込んでいく。
 - 最初に処理の実行が重ならない状況から考える。
 - 次に処理が連続して実行されたり、実行が重なる状況を考えていく。



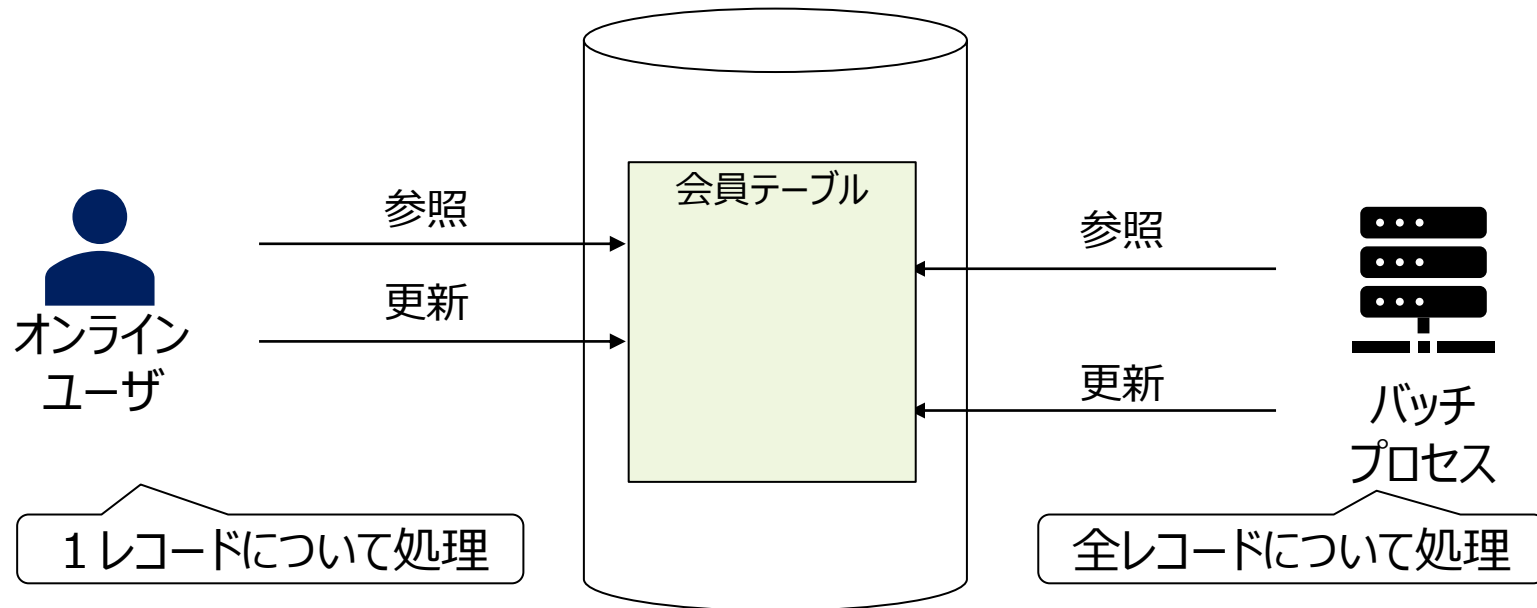
事例の紹介を通して、具体的な排他制御方式の設計手順を紹介します。

- Case1.オンライン処理とバッチ処理の並走
- Case2.常駐バッチの並列実行

- Case1.オンライン処理とバッチ処理の並走
- Case2.常駐バッチの並列実行

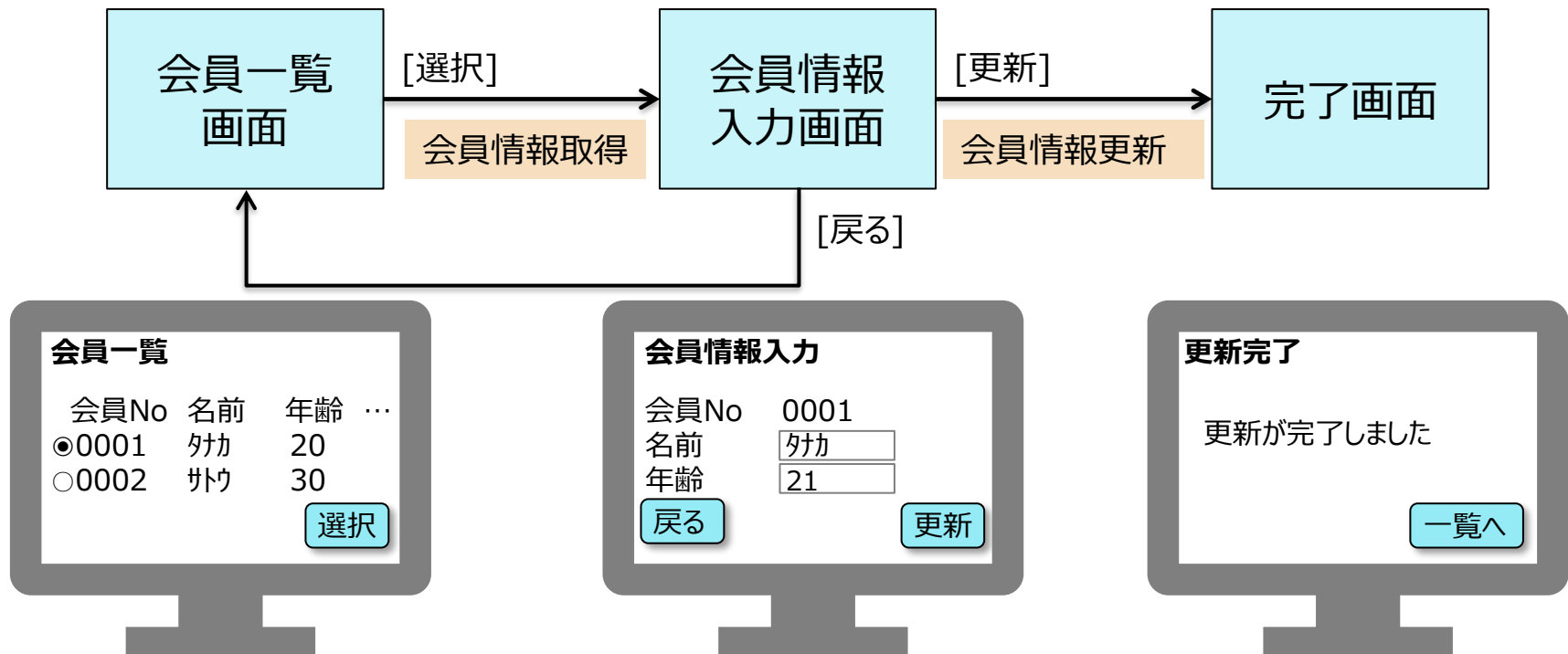
Case1.オンライン処理とバッチ処理の並走

排他制御設計での考慮点は、「オンライン処理とバッチ処理が同時刻に同じテーブルやレコードを参照・更新する可能性がある」ことです。



オンライン

ユーザは会員一覧画面で処理対象会員を選択し、
入力画面で更新情報を入力して会員情報を更新する。

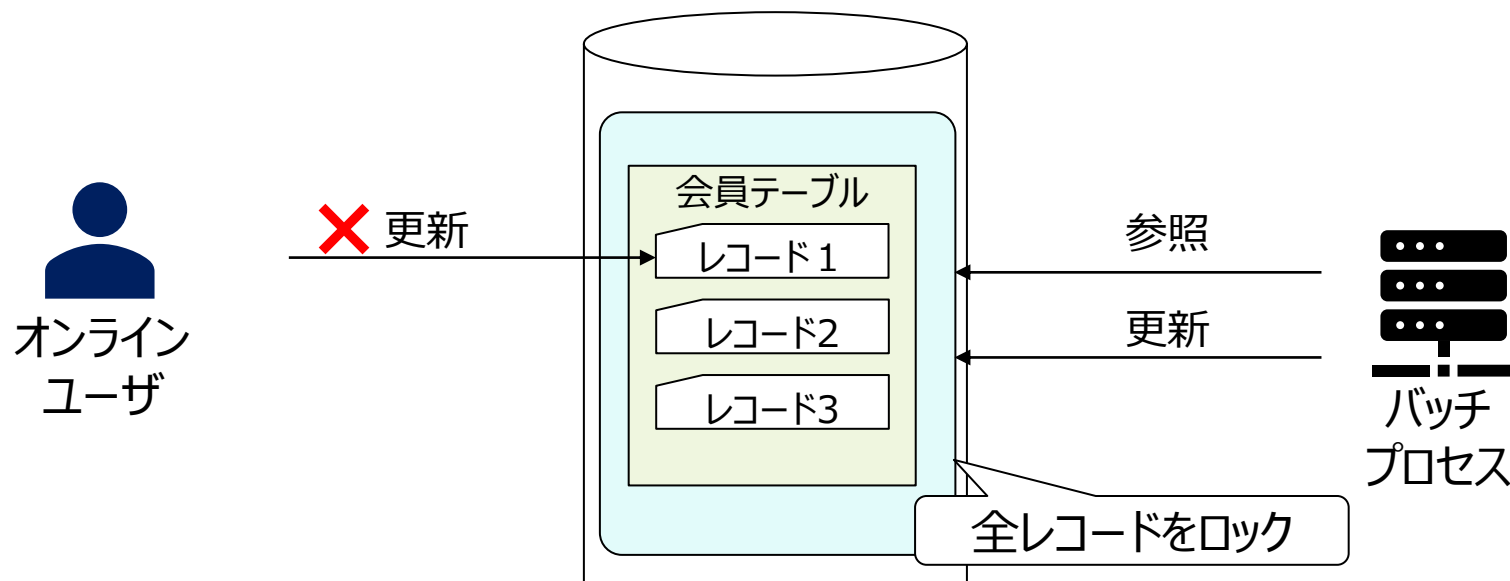


バッチ

- 会員テーブルの全レコードに対して更新処理を行う。
- 処理対象となる会員テーブルレコード数は約10万件。
- バッチの実行時間は約 1 時間、その間オンライン業務は停止しないこととする。

オンラインのユーザが待たされることがなく、
通常通り処理ができること

オンライン・バッチ競合時の考慮(1/6)

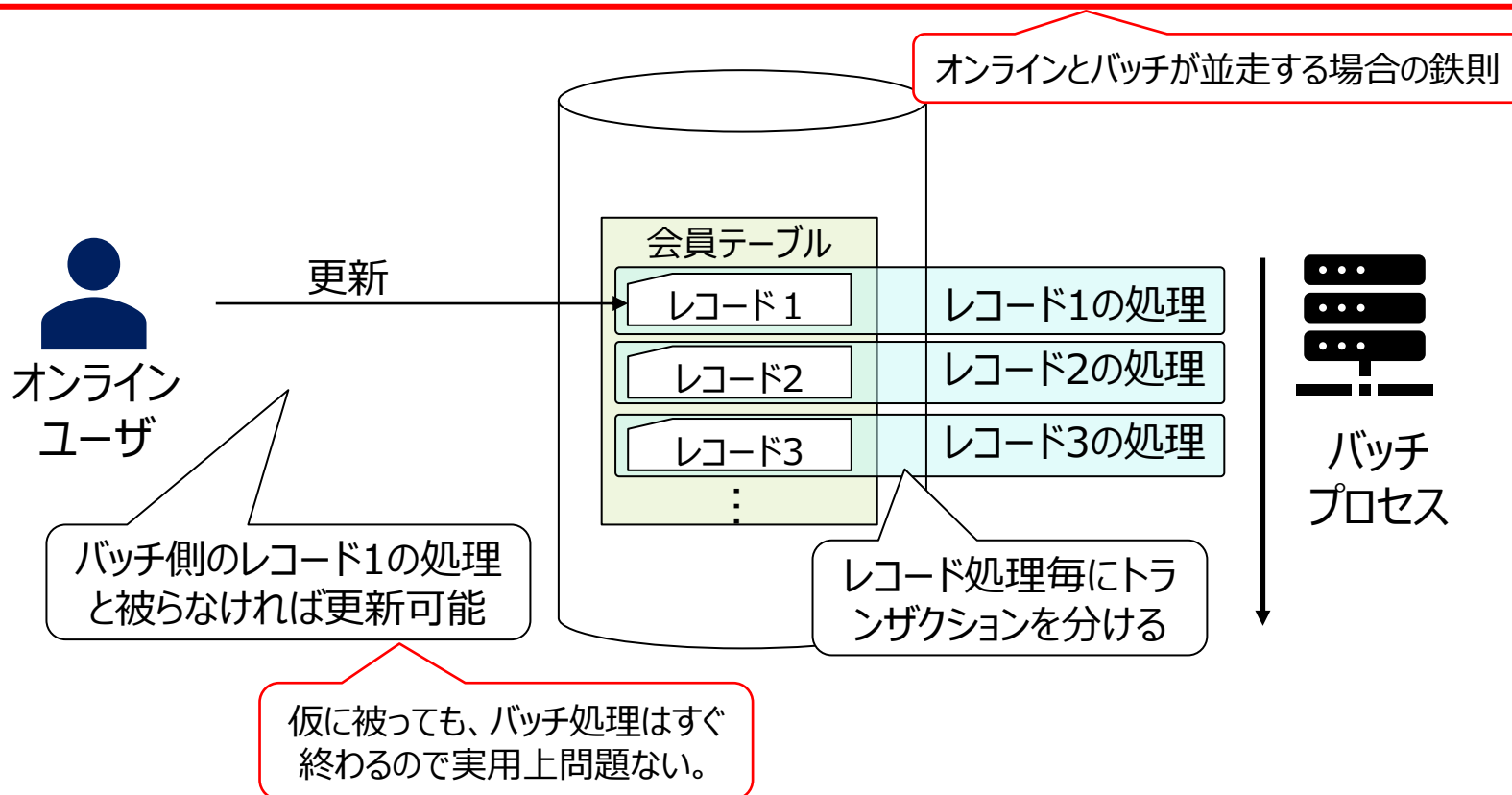


バッチが更新するために全レコードをロックしてしまうと、バッチ起動中、オンラインユーザはロックが取得できなくて作業できなくなる

バッチ実行中オンライン業務は停止しないこととする、という要件が満たせない

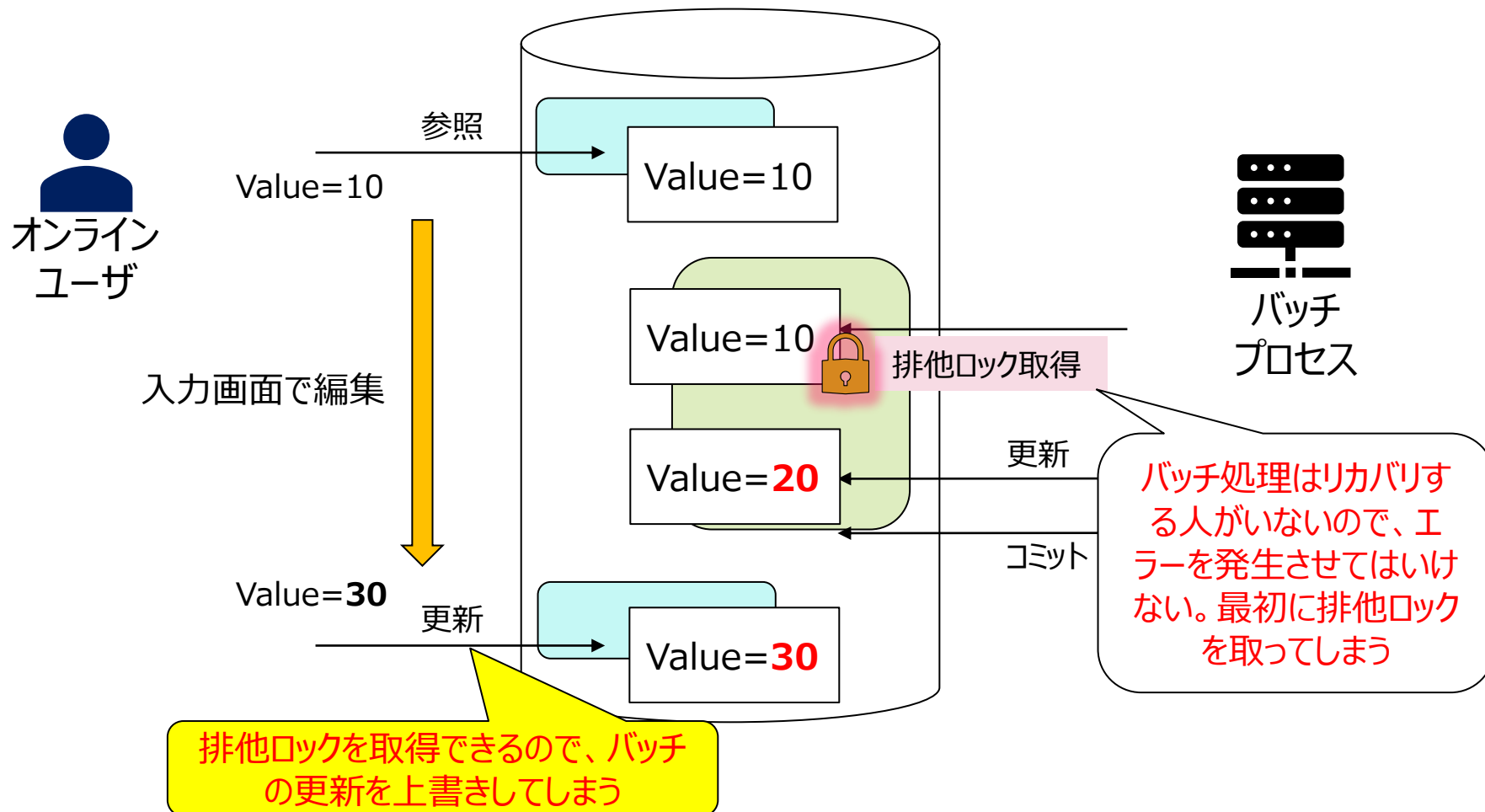
オンライン・バッチ競合時の考慮(2/6)

対策：レコード処理毎にトランザクションを分け、バッチが同時にロックするレコードを1件のみとする。



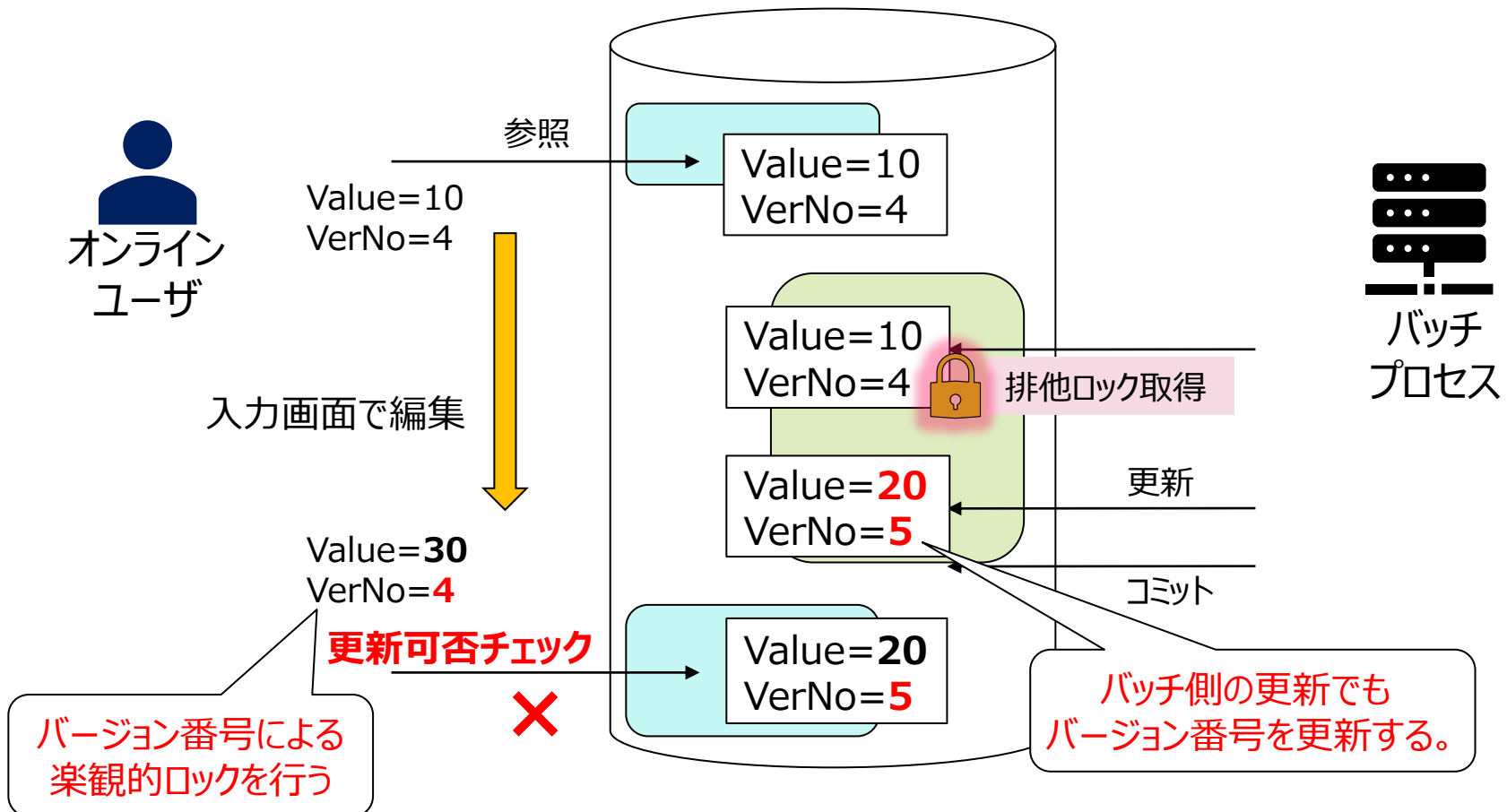
オンライン・バッチ競合時の考慮(3/6)

次に競合が起こらないか考えていきます。
まずは、オンラインの入力画面で編集中にバッチで更新が行われた場合から。



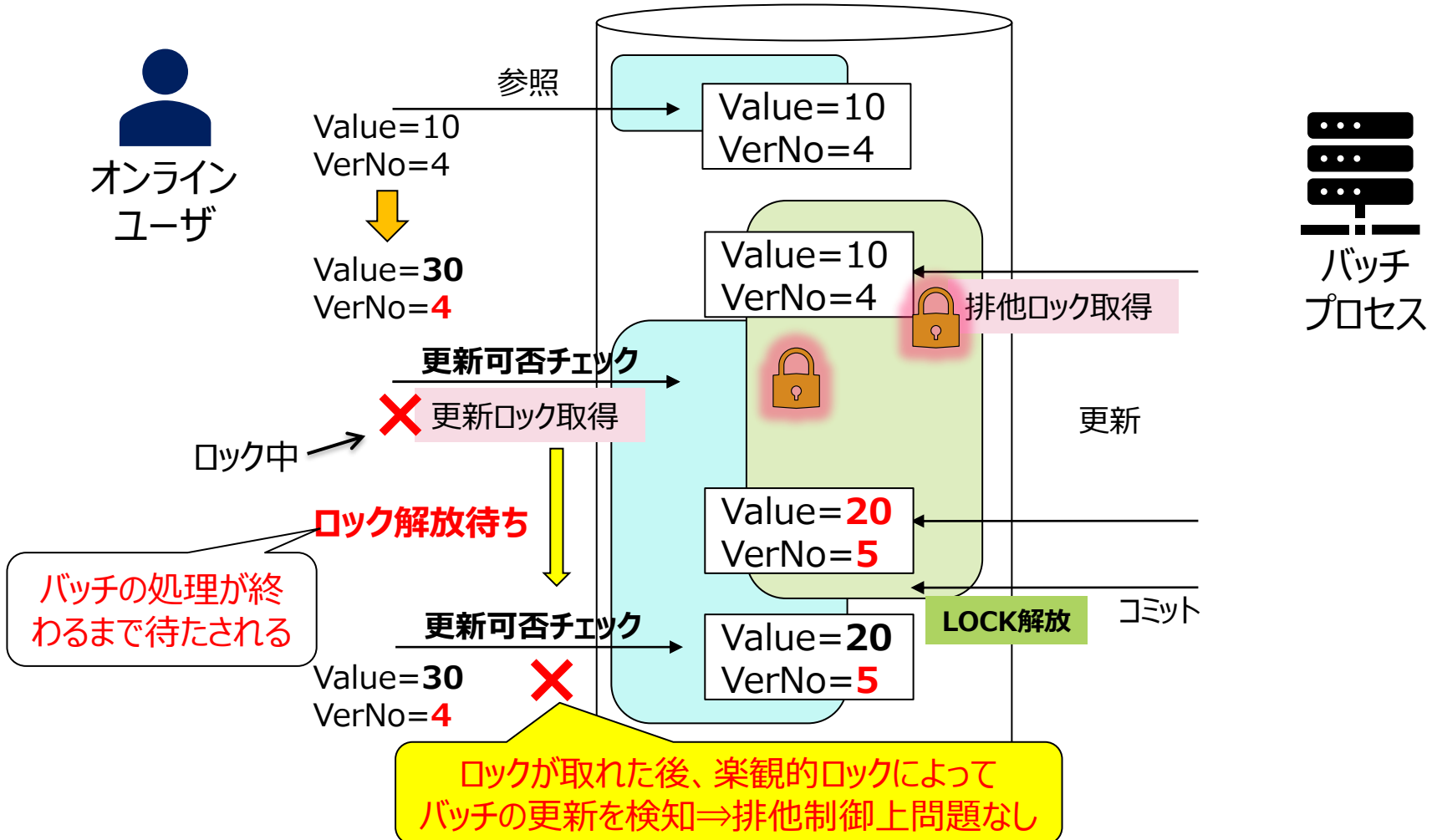
オンライン・バッチ競合時の考慮(4/6)

対策：バージョン番号による楽観的ロックを行うことで、バッチの更新を検知できるようにする。



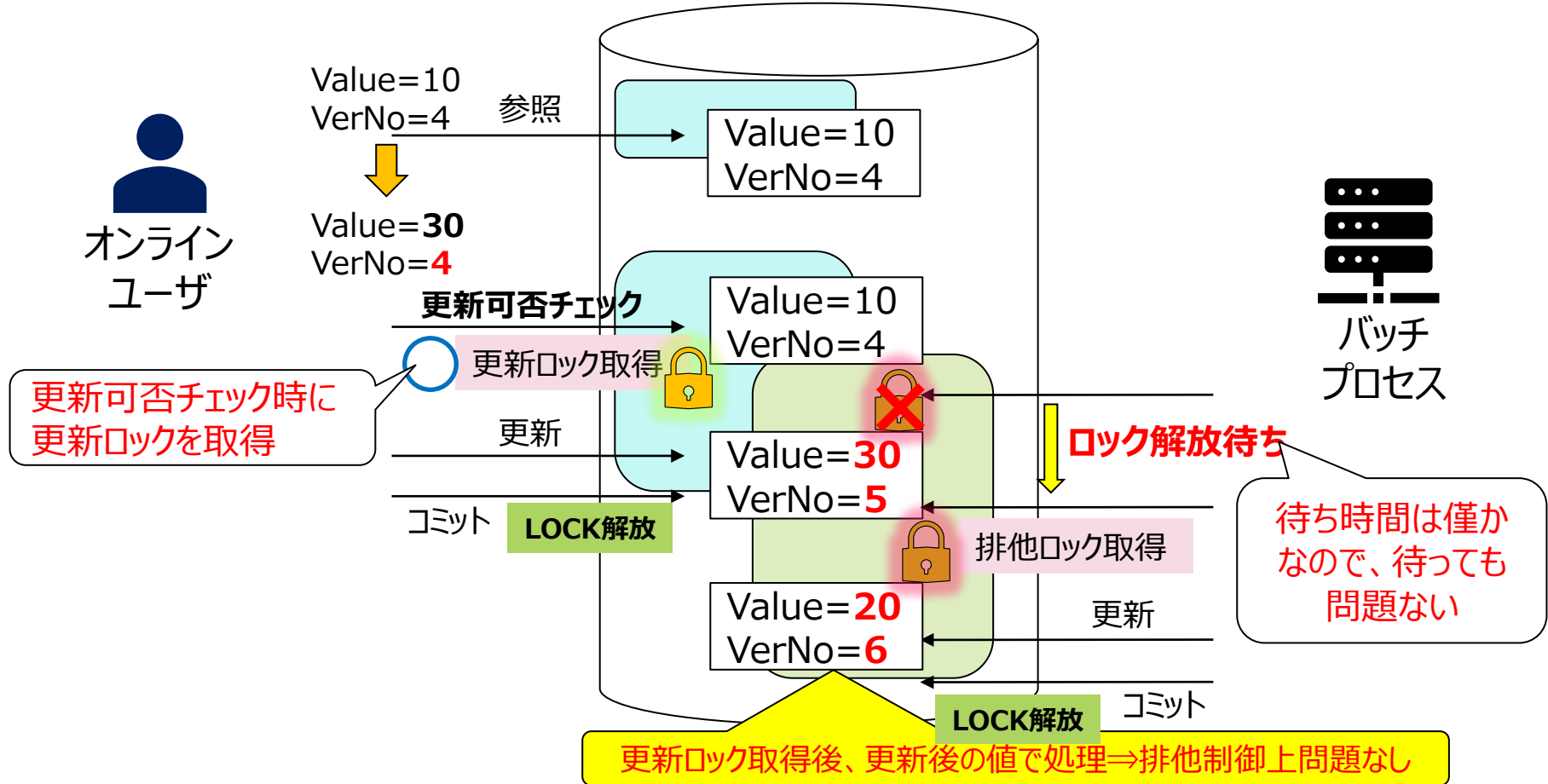
オンライン・バッチ競合時の考慮(5/6)

さらに処理の間隔を近づけ、オンラインの更新処理とバッチの更新処理が重なった場合を考えます。まずは、バッチ⇒オンラインの順で重なった場合から。



オンライン・バッチ競合時の考慮(6/6)

次に、オンライン⇒バッチの順で重なった場合です。

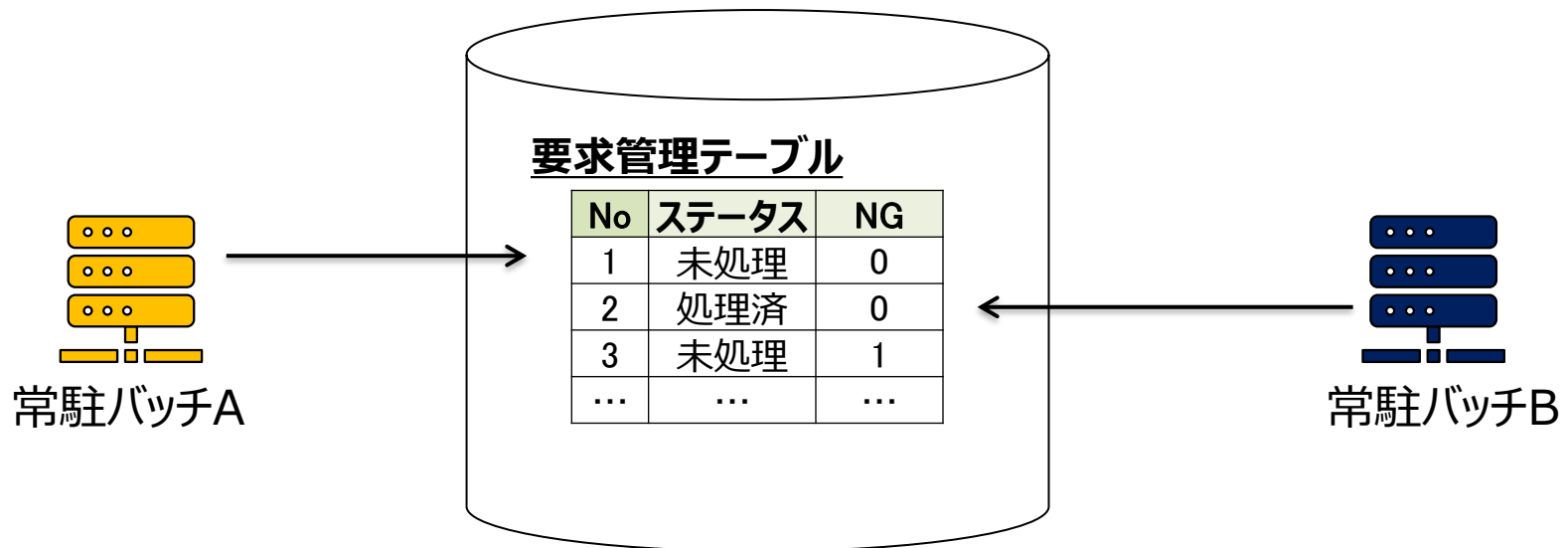


今回はオンラインとバッチが重なった時でも問題ありませんでした。ここまで検討して、「全体として大丈夫」ということができます。

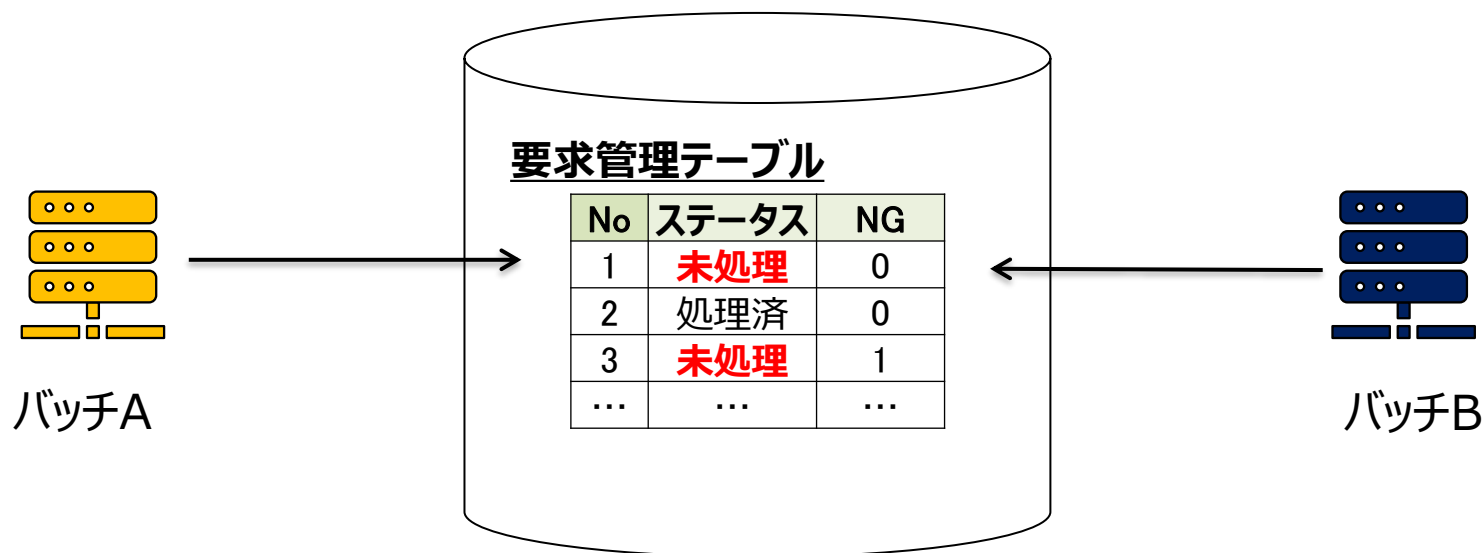
- Case1.オンライン処理とバッチ処理の並走
- Case2.常駐バッチの並列実行

Case2.常駐バッチの並列実行

2つのサーバで同じ処理を行う常駐バッチを並列実行し、同じ1つのテーブルに対して更新処理を行う。

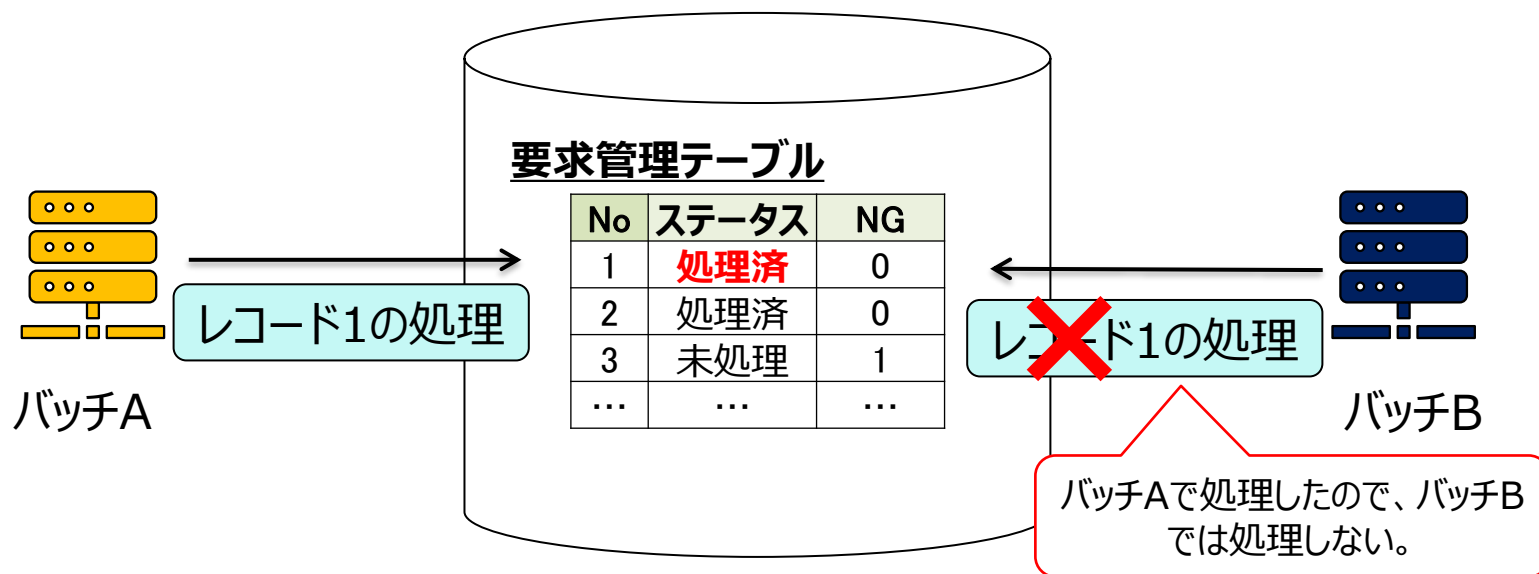


- テーブルに存在する**ステータスが「未処理」**となっている全てのレコードを取得する。
- 取得した全てのレコードに対してデータ処理を行う。
- 処理が完了したレコードは**ステータスを「処理済」に更新**する。

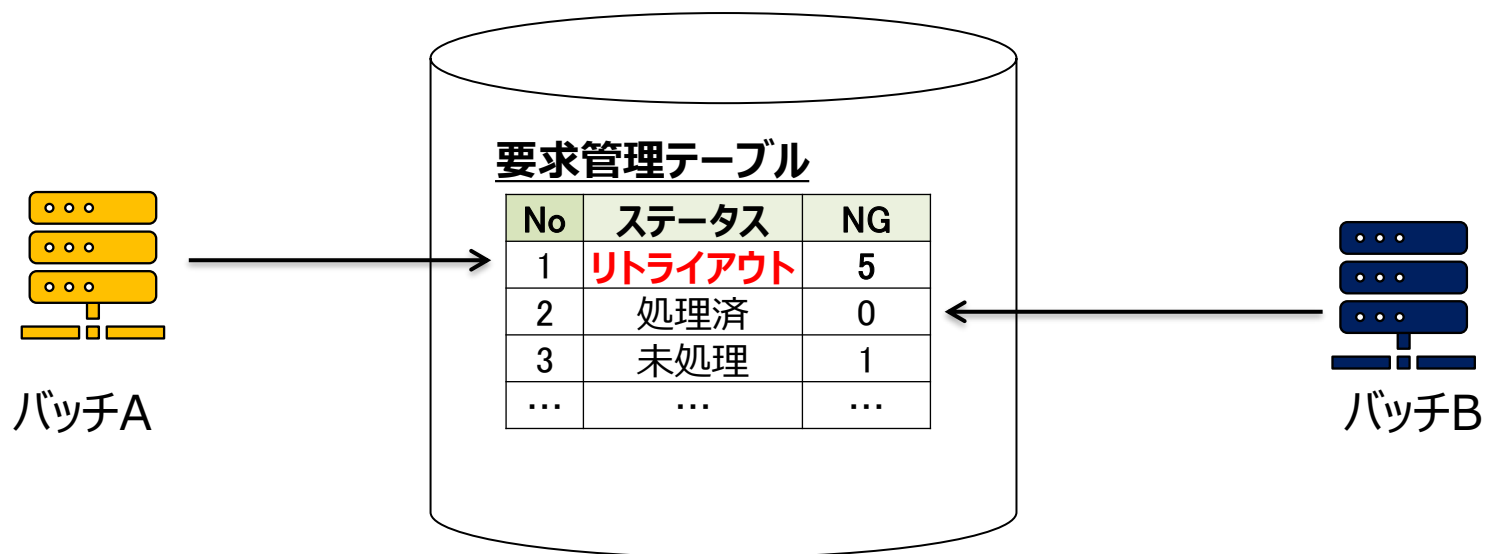


バッチAで処理を行ったレコードについて、バッチBで二重に処理を実行させないようにする。

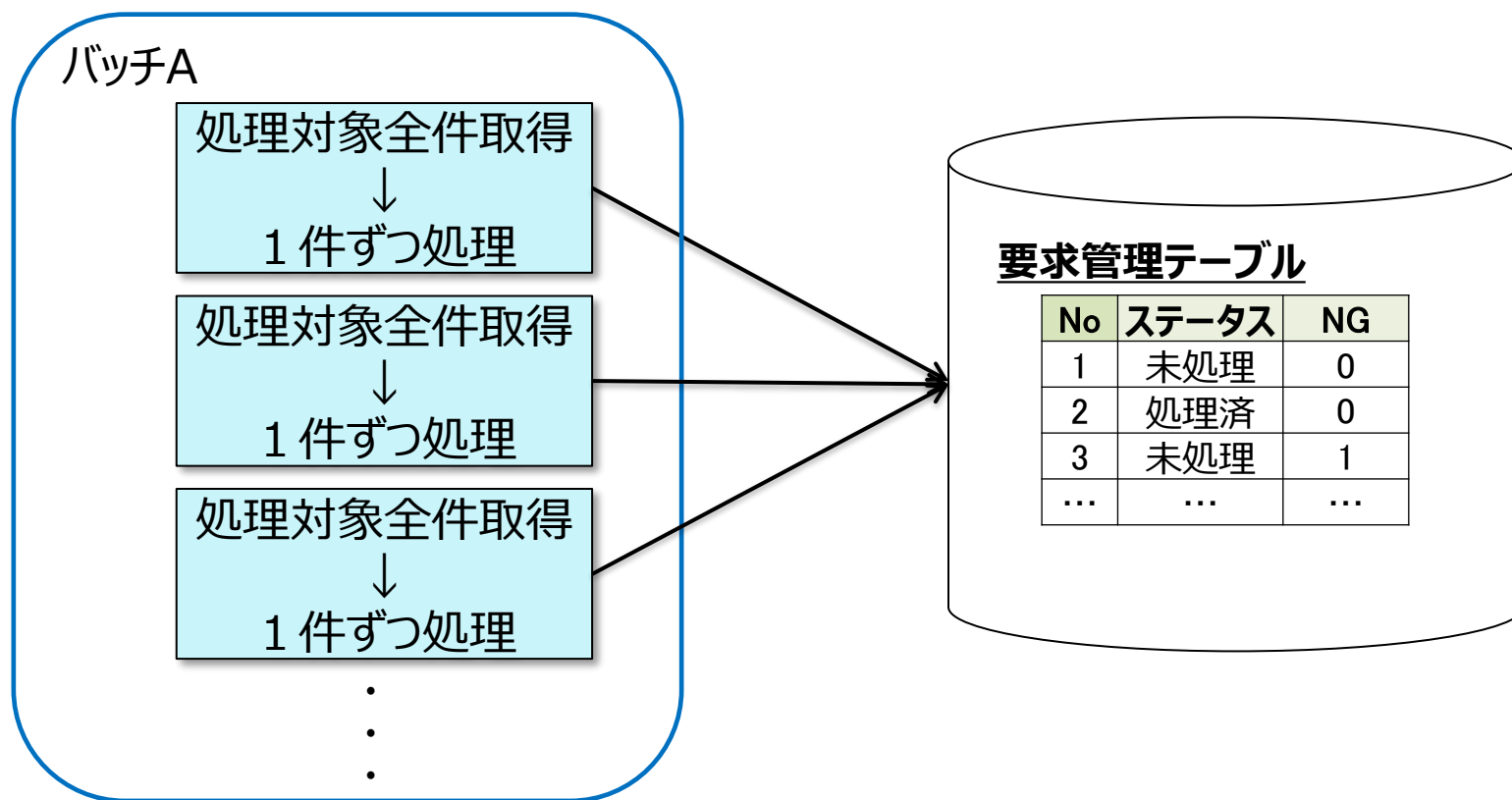
(逆も同じ。同じレコードを食い合わないようにする)



処理に失敗した場合、NG回数をインクリメントする。NG回数が5回に達したら、ステータスを「リトライアウト」に更新する。



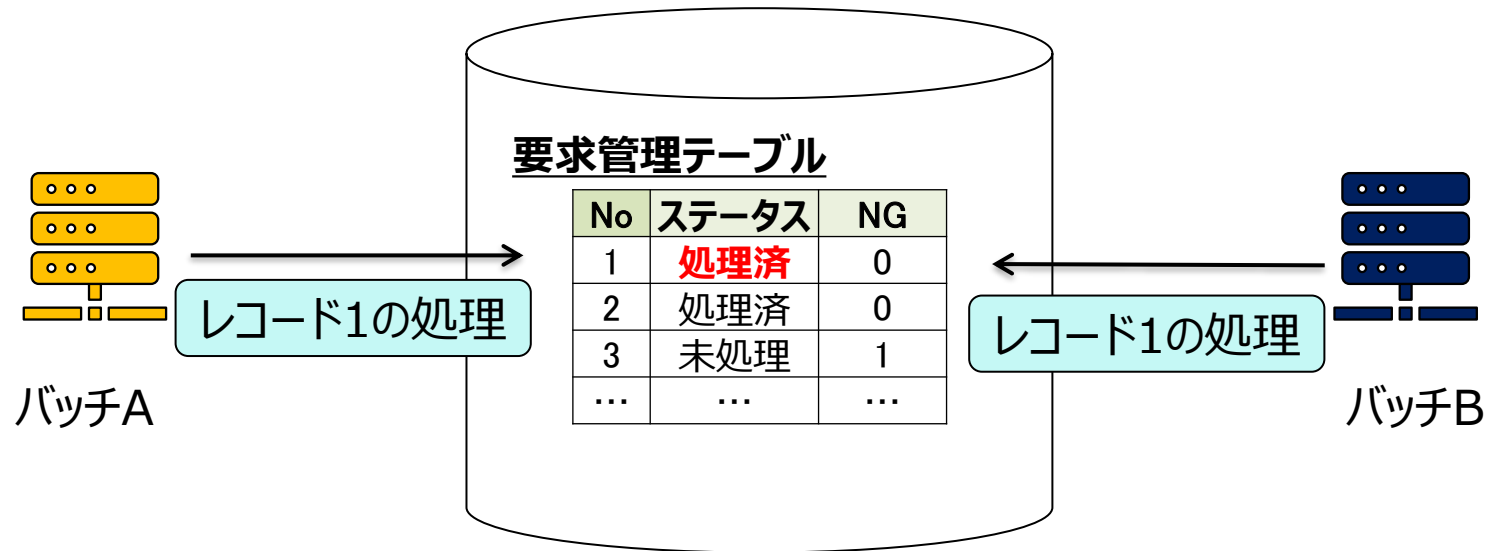
全件について処理が完了したら、再び対象レコードの全件取得から処理を開始する。



- 可用性の向上
 - 片方のバッチプロセスが停止しても、もう一方のサーバのバッチプロセスだけで全ての処理を遂行できる。
- スループットの向上
 - 2つのバッチプロセスで並列して更新処理を行うことにより、処理速度を向上させる。

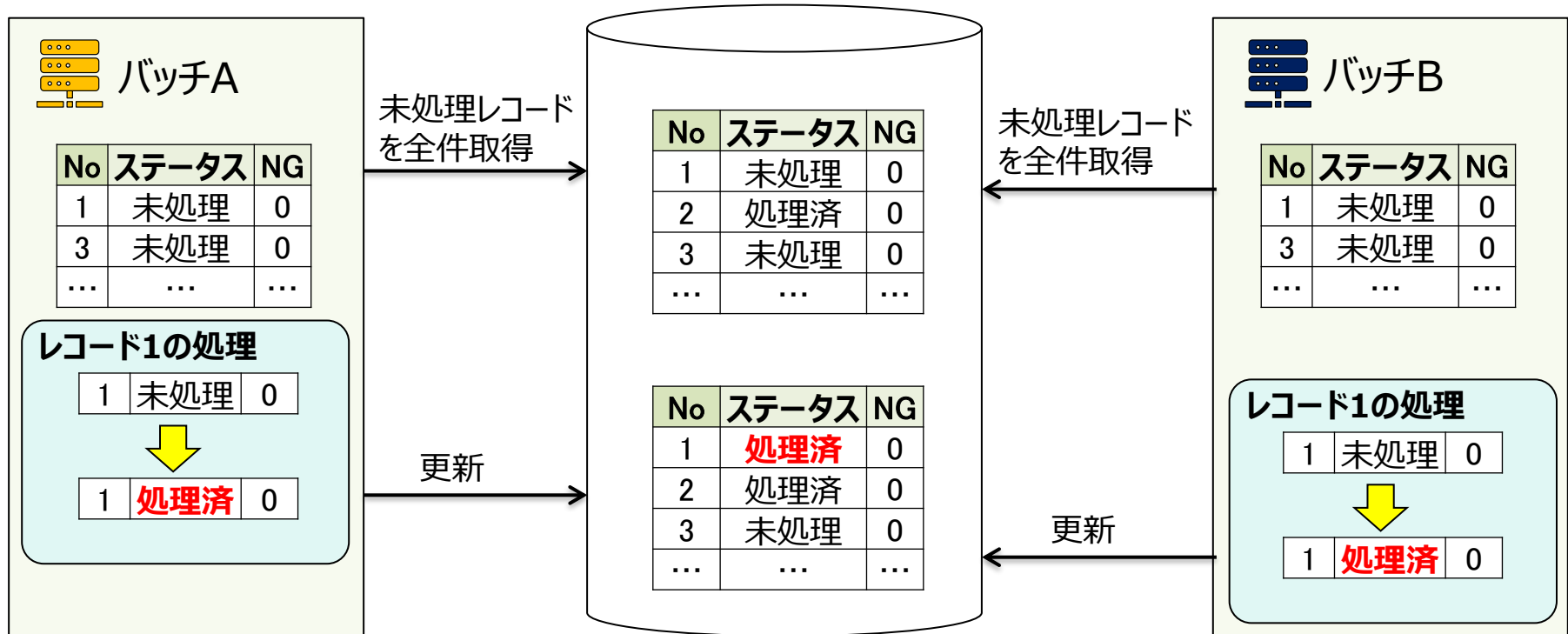
並列化により発生する問題

並列する複数のプロセスが同じレコードを食い合うという問題を解決する必要があります。



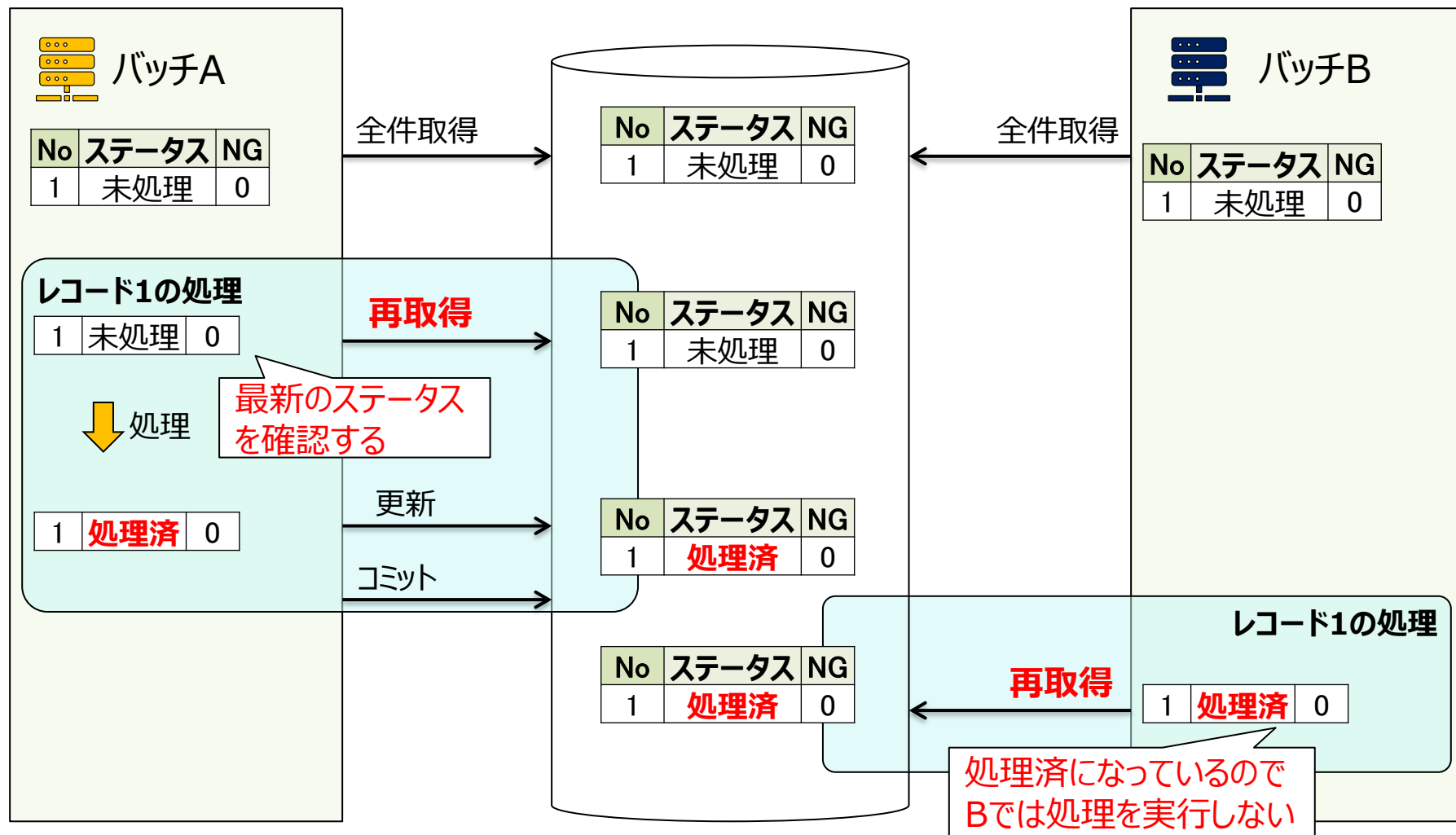
食い合う：
バッチA、バッチB両方で同じ
レコードを処理してしまうこと。

同じレコードを食い合わないようにする(1/4)



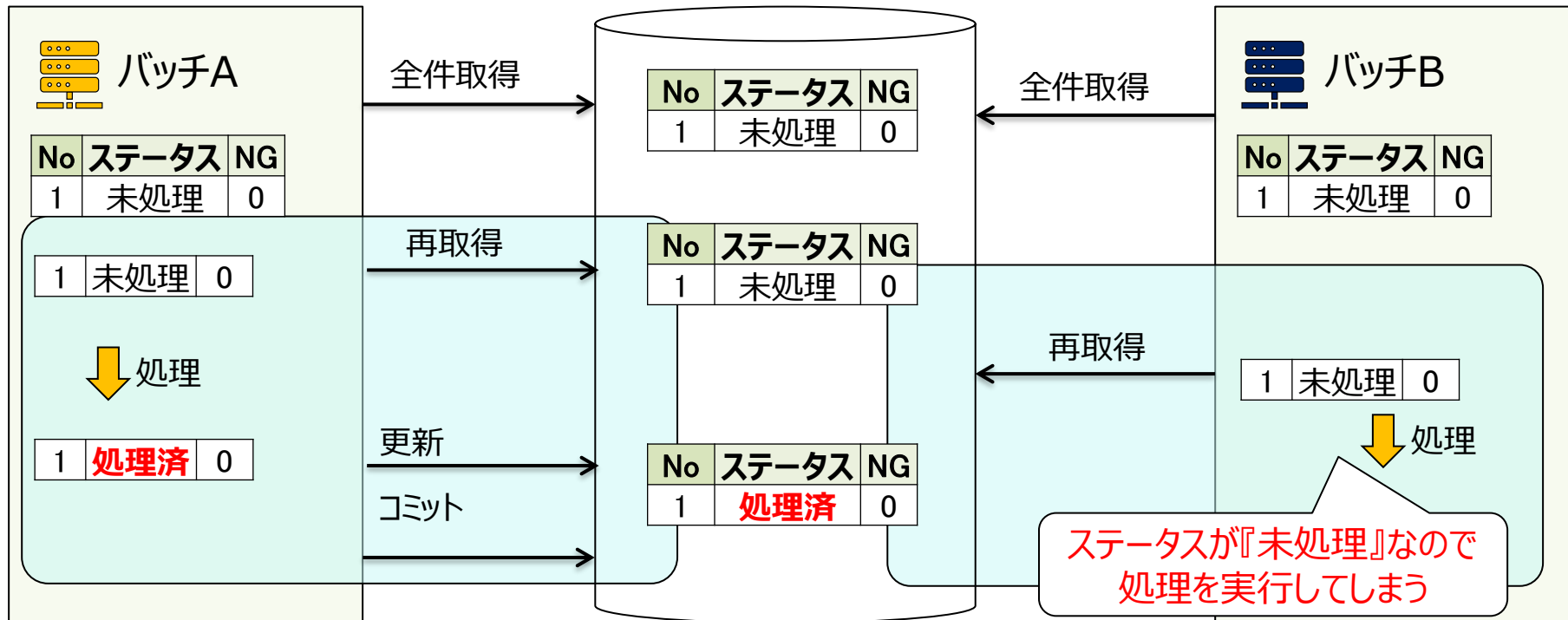
最新のステータスを確認しないから食い合いが発生します。
そこで、処理対象レコードを**再取得**して最新のステータスを確認するようにします。

同じレコードを食い合わないようにする(2/4)



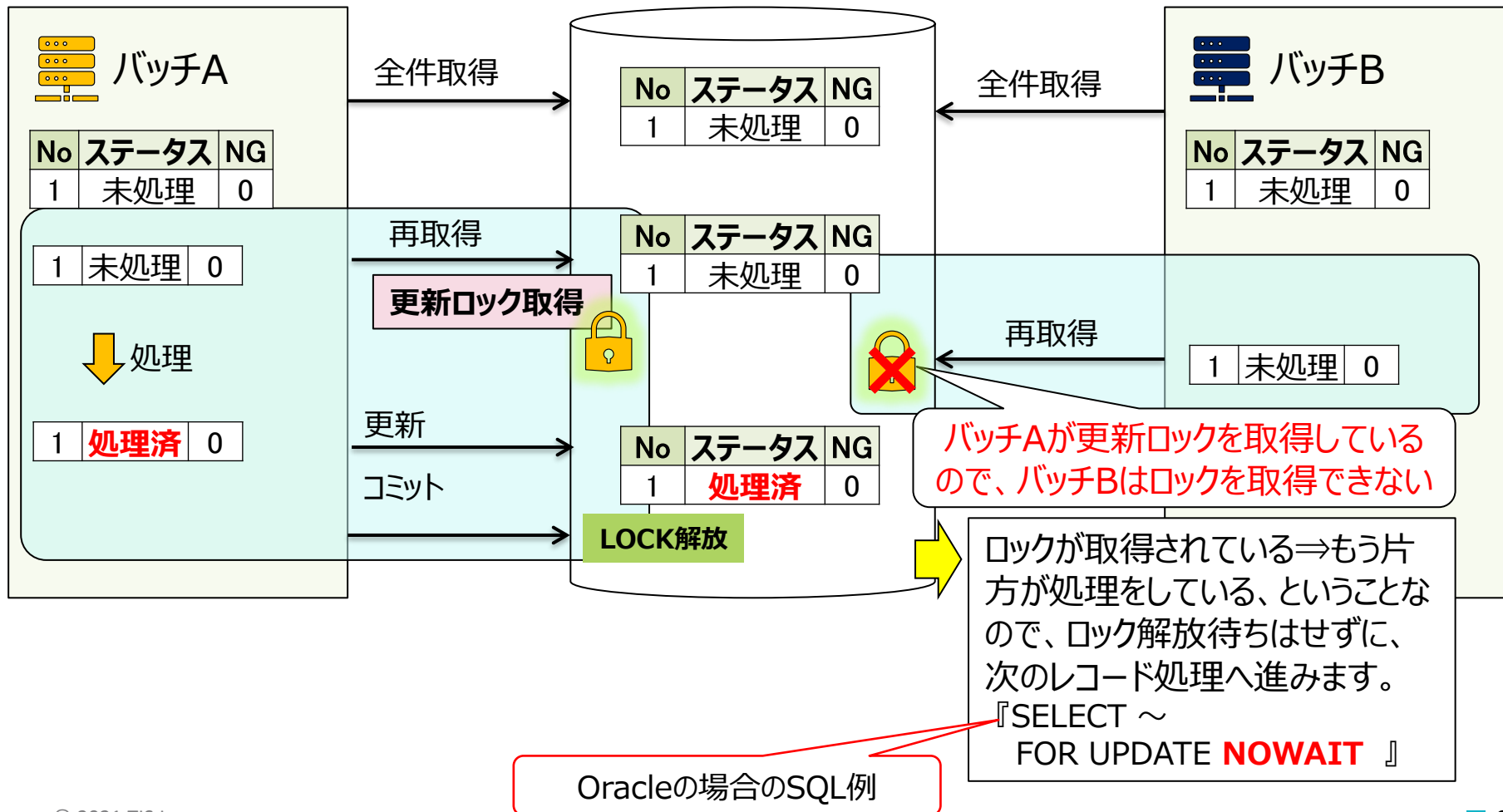
同じレコードを食い合わないようにする(3/4)

バッチAとバッチBの処理間隔を近づけ、処理が重なった時を考えます。
バッチAでレコード再取得してから更新を行うまでの間にバッチBで同じレコードを再取得した場合、バッチA、B両方で処理を実行してしまいます。



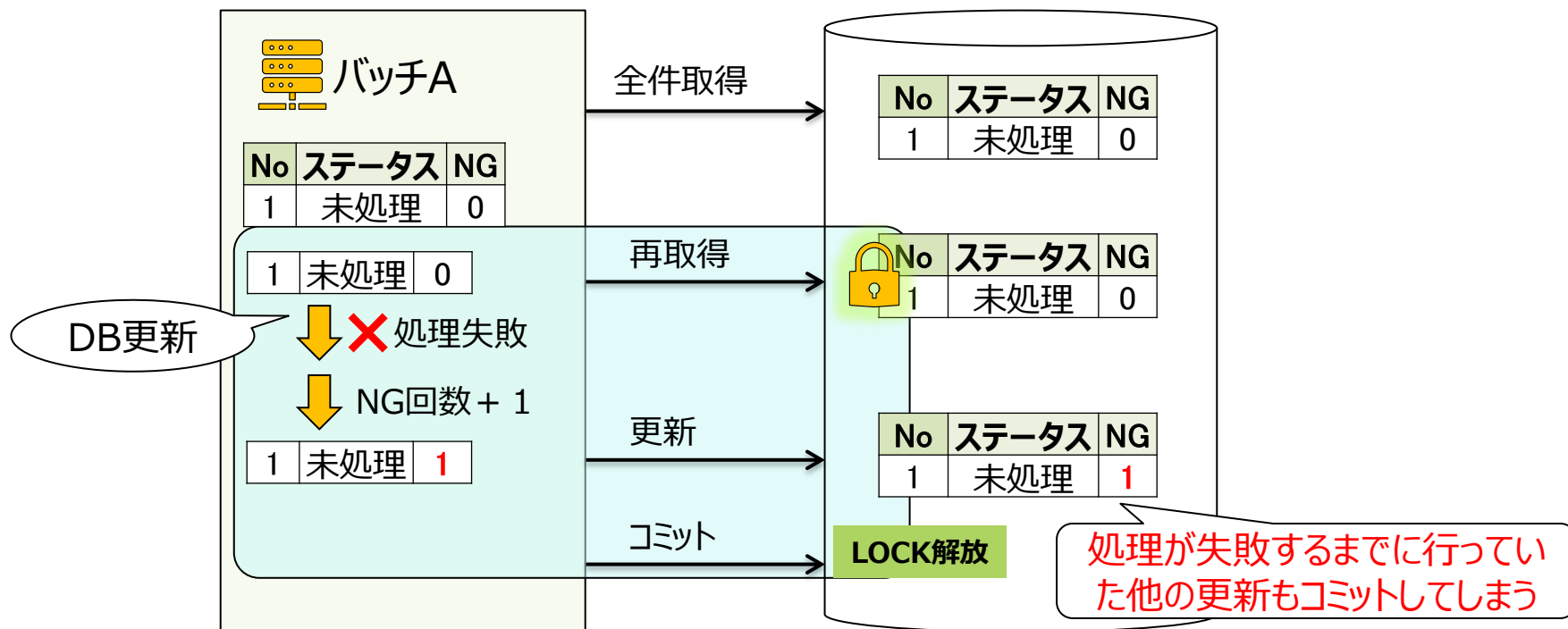
同じレコードを食い合わないようにする(4/4)

対策：再取得時にレコードの更新ロックを取得することで、片方のバッチのみが処理できるようにする。



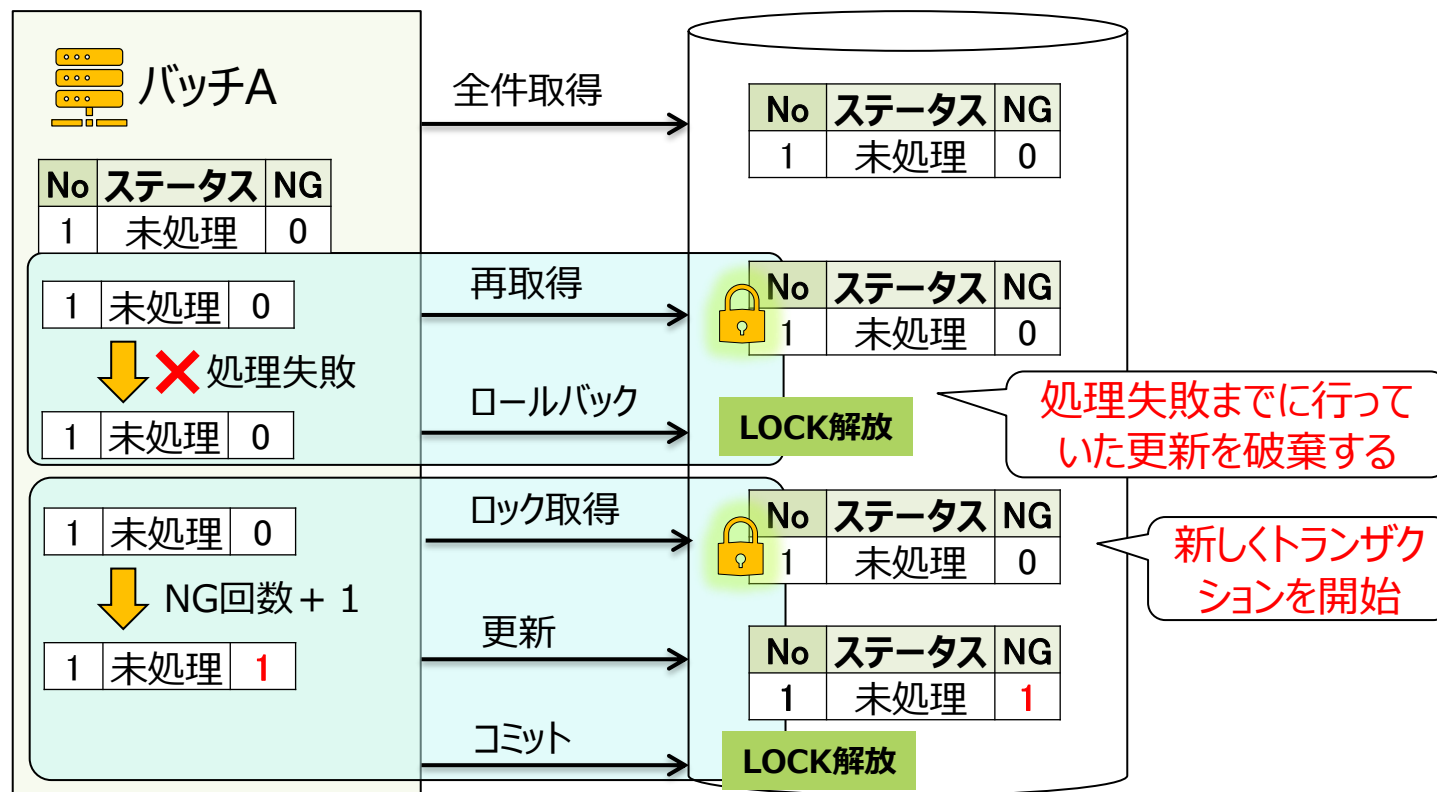
処理失敗レコードの更新(1/7)

処理失敗した場合、同じトランザクションでNG回数をインクリメントしてコミットすると、ロールバックすべき他の更新までコミットしてしまいます。



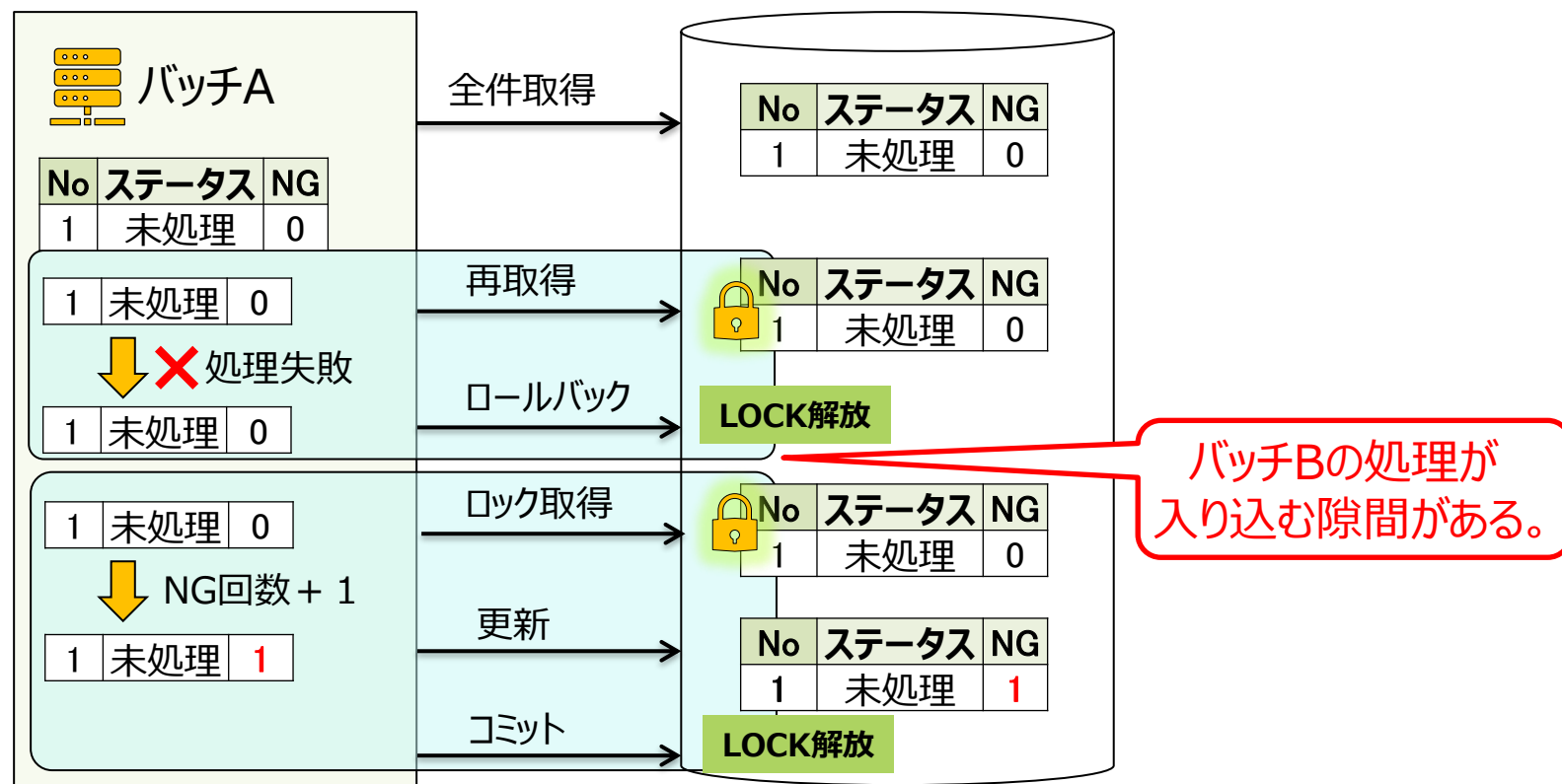
処理失敗レコードの更新(2/7)

対策：処理失敗した場合、実行中のトランザクションをロールバックし、
新しいトランザクションでNG回数をインクリメントする。



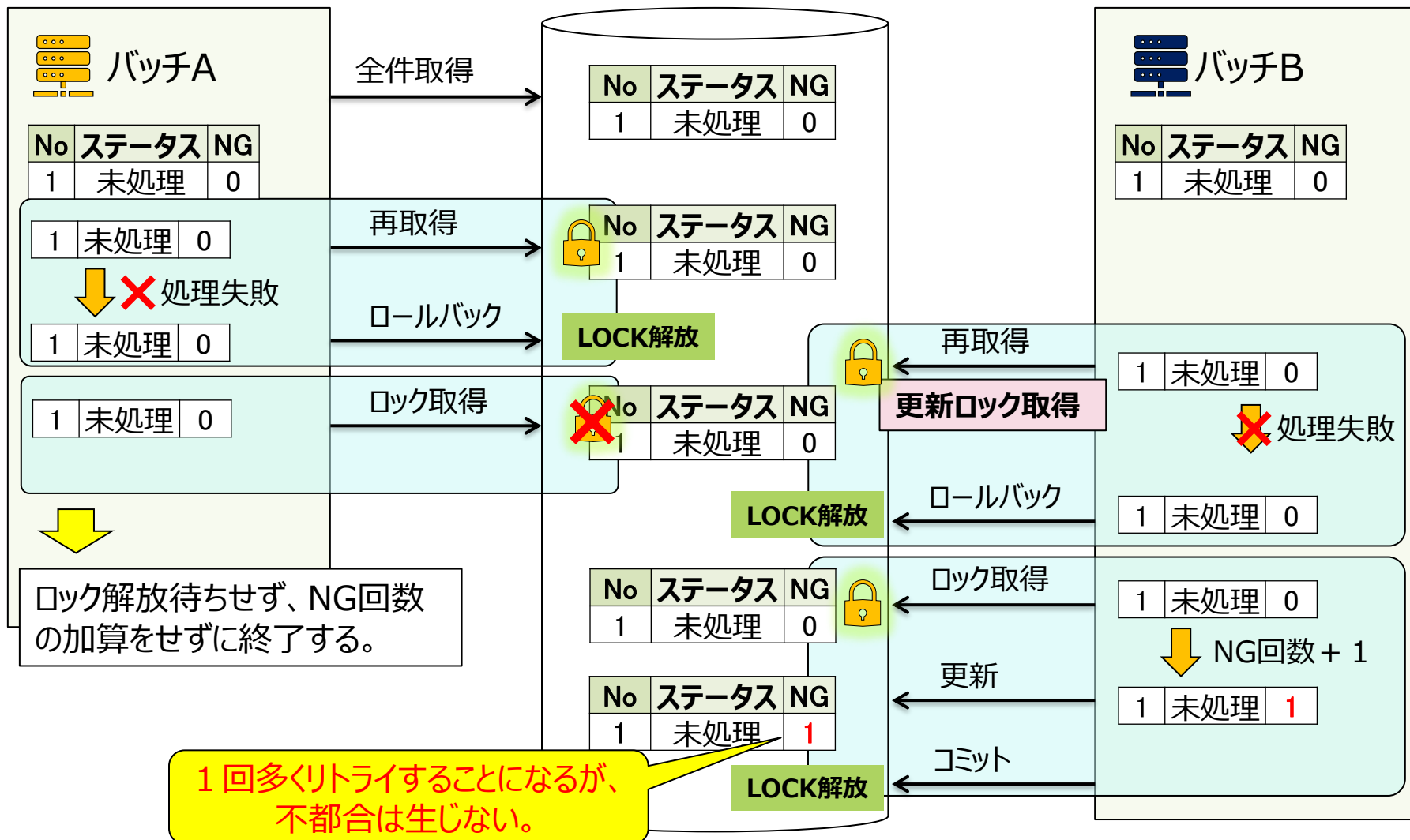
処理失敗レコードの更新(3/7)

しかしトランザクションを分けたため、バッチBが入り込む隙間が生まれます。この場合の検討が必要です。



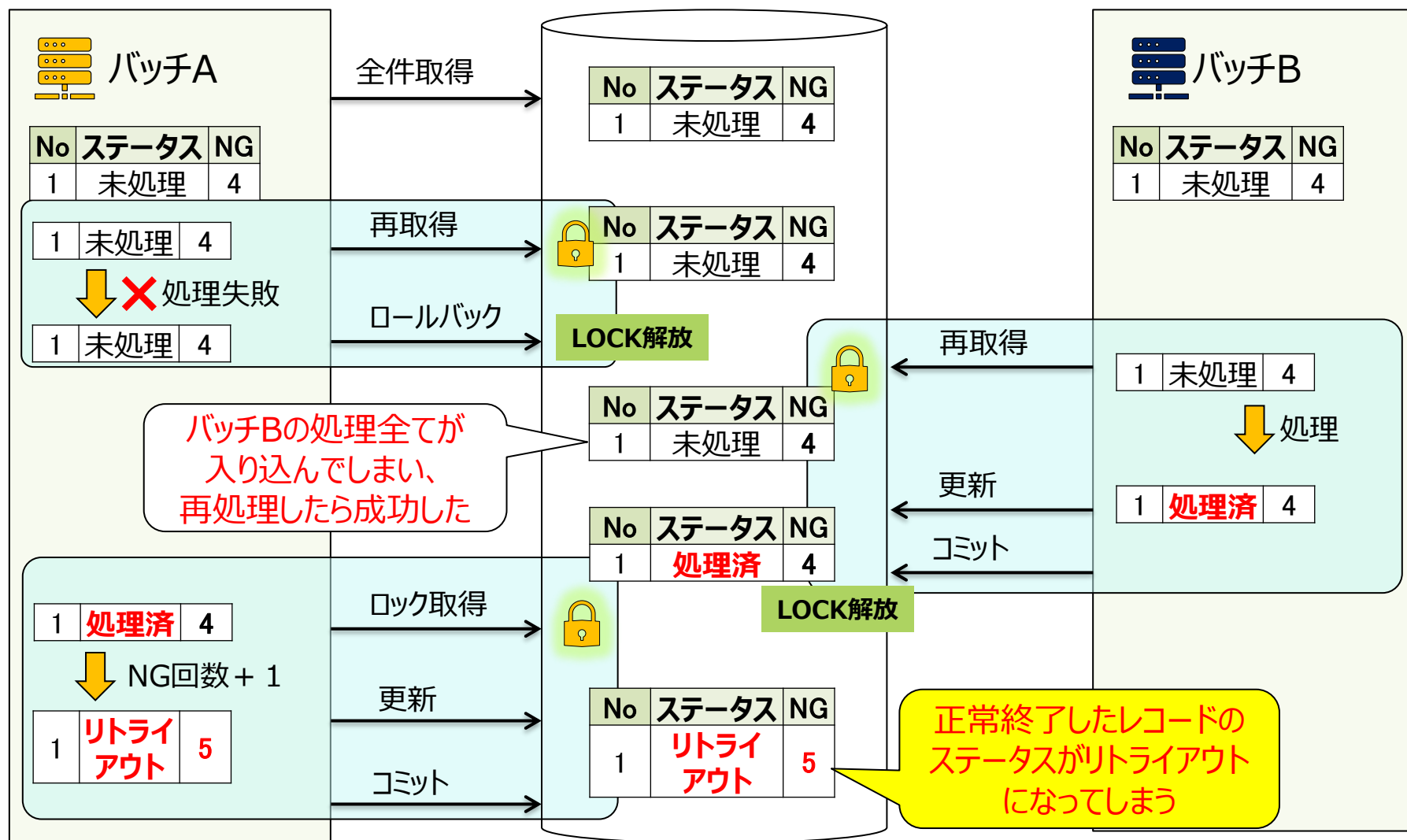
処理失敗レコードの更新(4/7)

バッチBが隙間で更新ロックを取得してしまった場合、不都合は生じません。



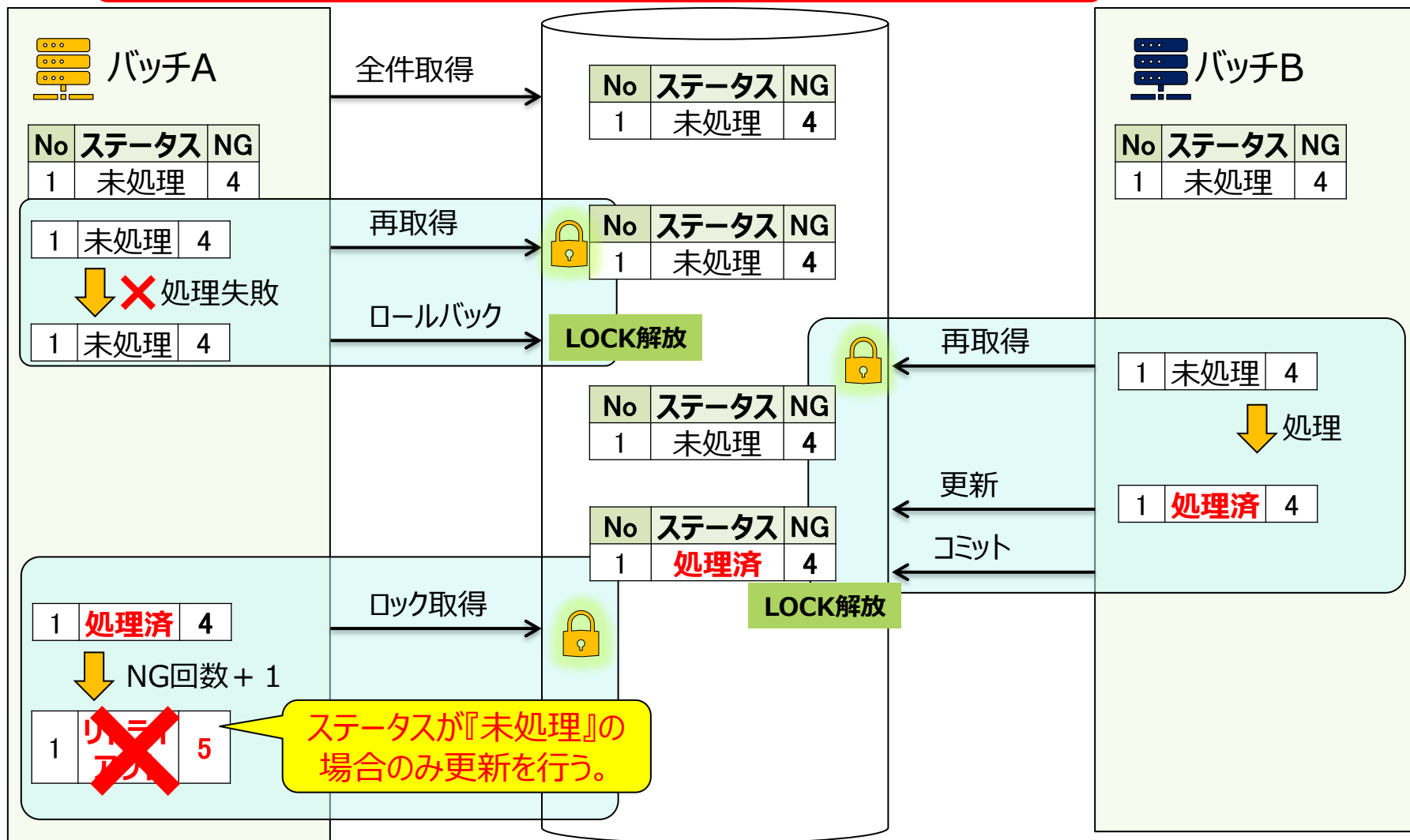
処理失敗レコードの更新(5/7)

バッチBの処理全てが入り込んでしまうと、正常終了してもリトライアウトになってしまいます。



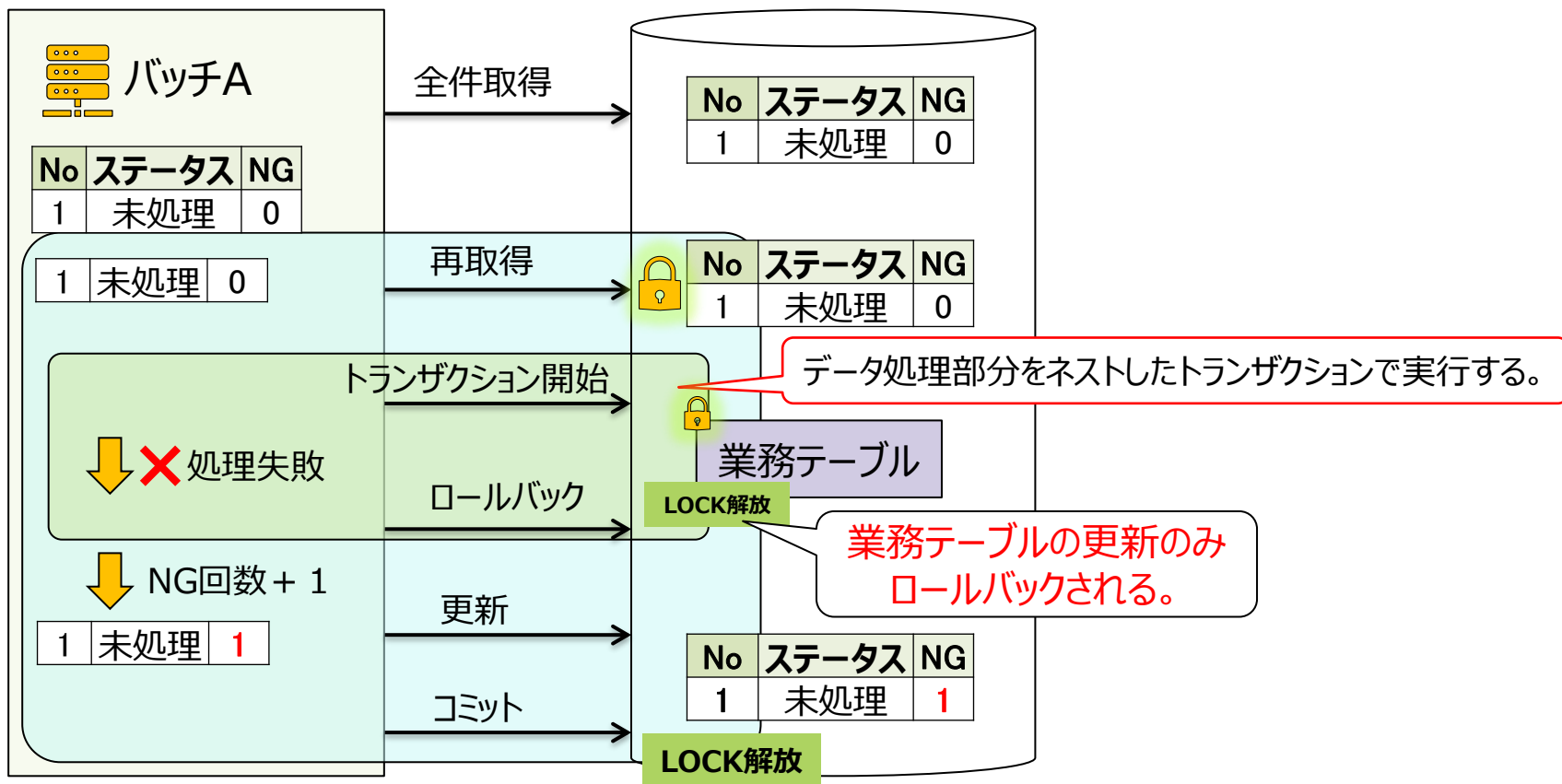
処理失敗レコードの更新(6/7)

対策：リトライアウトに更新する前に、ステータスの確認を行う。



処理失敗レコードの更新(7/7)

別の対策：トランザクションをネスト（トランザクションの中で別のトランザクションを実行）する。



- 排他制御設計とは
- 排他制御を実現するための技術要素
 - 物理ロック
 - 共有ロック、排他ロック、更新ロック
 - 論理ロック
 - 楽観的ロック方式、悲観的ロック方式
- 排他制御設計の考え方（事例を通して解説）
 - オンライン処理とバッチ処理の並走
 - 常駐バッチの並列実行

ITで、社会の願い叶えよう。



TIS INTEC
Group

付録

DB2 9.7 & Oracle 11g 比較ハンドブック
翔泳社
翔泳社 著
(電子書籍)



<https://enterprisezine.jp/sepub/ibmdownload>
2021年11月26日16時の最新情報を取得