コミット管理およびコミットされないレコードの読み取り

GeneXus™

Transaction タイプと Procedure タイプの GeneXus オブジェクトには、Yes または No の値をとる [Commit on exit] プロパティがあります。これにより、生成されたプログラムが自動コミットを実行するかどうかを決めることができます。

[Commit on exit] プロパティ

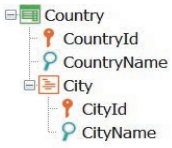
[Commit on exit] = Yes

プロシージャー

```
For each Country
     Where CountryId = 2
     CountryName = "New country name"
endfor
```

生成されたプログラムのソースの最後に自動コミットが含まれます。

トランザクション

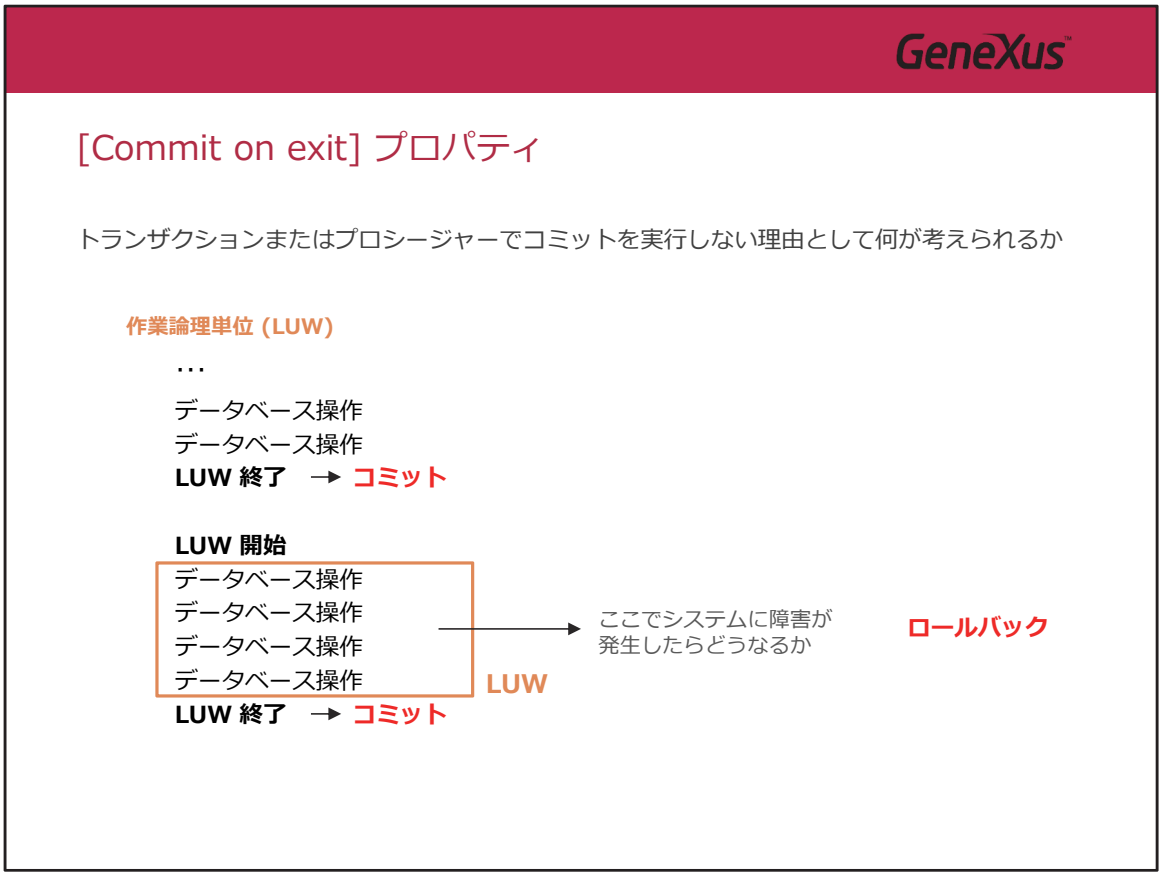


データベースでの変更後に、生成されたプログラムに自動コミットが含まれます。

既定では、生成されたプログラムが Transaction タイプと Procedure タイプのオブジェクトに関連付けられているものなら、GeneXus によって Commit ステートメントが含められます。

- プロシージャーの場合は、生成されたプログラムのソースの最後に自動コミットが含まれます。
- トランザクションの場合は、データベースで変更が行われた後に、生成されたプログラムに自動コミットが含まれます。具体的には、データの挿入、変更または削除の後で、AfterComplete のトリガーのタイミングに条件付けられたルールに関連付けられたコードおよび AfterTrn イベントに関連付けられたコードが実行される**直前**です。

たとえば、第 2 レベルとして City を持つ Country トランザクションがあり、フォームを介して 5 つの国を挿入する場合、コミットが 5 回実行されます。そのタイミングは、各国およびその都市の情報を保存した後で、AfterComplete のタイミングに条件付けられたルールが実行される前になります。



トランザクションまたはプロシージャーでコミットを実行しない理由として何が考えられるでしょうか。

それは、作業論理単位 (LUW) のカスタマイズのためです。

つまり、LUW を拡張する必要がある場合です。たとえば、複数のプロシージャー、あるいは 1 つまたは複数のプロシージャーを持つトランザクションが 1 つの LUW を構成しており、すべてを含めてコミットを実行する必要がある場合がこれに該当します。

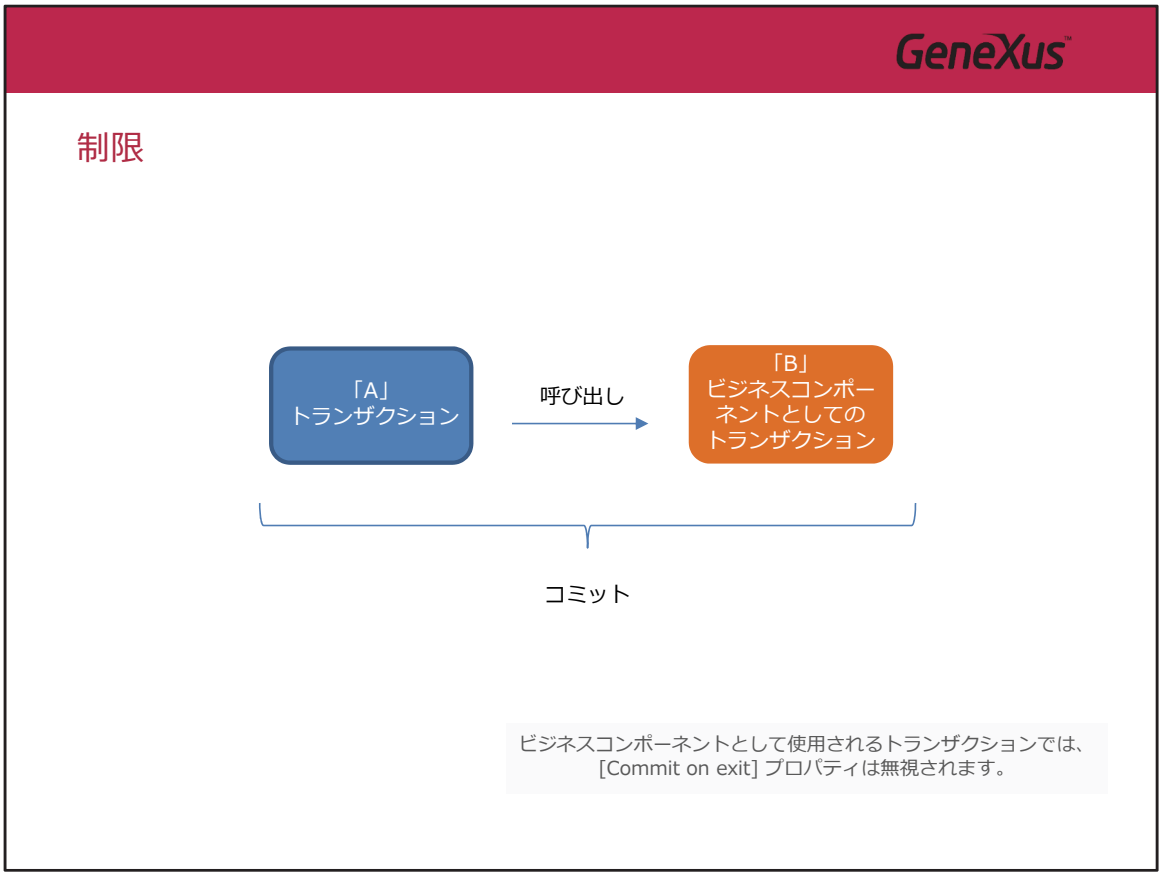
GeneXus

制限

これについてはいくつか重要な制限があります。

Web 環境では、各トランザクションがコミットできるのは、データベースに対して実行されるそのトランザクションの一連の操作、およびそれによって呼び出されるプロシージャーのみです。別のトランザクションにより実行される操作は含まれません。

つまり、トランザクションが別のトランザクションを呼び出す場合、1 つのトランザクションにより実行されるコミットは、もう 1 つのトランザクションにより挿入、変更または削除されるレコードには適用されません。したがって、2 つの異なるトランザクションを同じ LUW に含めることはできません。



2 つの異なるトランザクションにより操作を実行する必要があり、2 つのトランザクションの間で 1 つの LUW を作成したい場合、ビジネスコンポーネントの概念を使用して実行することが解決策となります。両方のトランザクションに関連付けられた操作を実行した後に Commit コマンドを含めます。

ただし、ビジネスコンポーネントとして使用されるトランザクションでは、[Commit on exit] プロパティは無視されます。

つまり、トランザクションの [Commit on exit] プロパティが Yes に設定されていても、そのトランザクションをビジネスコンポーネントとして使用する場合、コミットは自動的には実行されません。Commit コマンドを明示的に宣言する必要があります。

このような動作により、複数のトランザクションの間に作業論理単位を指定して、必要な箇所に Commit コマンドを含めることができます。

GeneXus

[Commit on exit] プロパティ - 例

1

[Commit on exit] = Yes


```
For each Country
     Where CountryId = 2
     CountryName = "New country name"
endfor
```

[Commit on exit] プロパティが考慮され、GeneXus により Commit が自動的に追加されます。

4 つの具体的な例を取り上げ、動作について分析します。

最初の例です。

Country トランザクションがあるとします。プロシージャーのソースを見てください。プロシージャーの [Commit on exit] プロパティは Yes に設定されています。

For each でデータベースに対して更新が実行されるため、GeneXus により Commit が自動的に追加され、CountryId = 2 の値を持つ国が新しい名前に更新されます。

[Commit on exit] プロパティ - 例

2

[Business Component] = True

[Commit on exit] = Yes


```
&Country.CountryName = "Brazil"
&Country.Insert()
```

[Commit on exit] プロパティが無視され、国はコミットされません。

2 つ目の例を見てみましょう。

同じ Country トランザクションがビジネスコンポーネントとして設定されているとします。プロシージャーのソースが示されています。

CountryId 項目属性は自動採番で、プロシージャーの [Commit on exit] プロパティは Yes に設定されています。

どのように動作するでしょうか。

プロシージャーはビジネスコンポーネントによる操作のみを実行するため、プロシージャーの [Commit on exit] プロパティが Yes に設定されていても、これは無視されて国はコミットされません。コミットするためには、Commit コマンドを明示的に追加する必要があります。

GeneXus

[Commit on exit] プロパティ - 例

3

[Business Component] = True

[Commit on exit] = Yes

```
&Country.CountryName = "Brazil"
&Country.Insert()

For each Country
     Where CountryId = 2
     CountryName = "New country name"
endfor
```

[Commit on exit] プロパティが考慮され、
ビジネスコンポーネントと For each 操作の両方がコミットされます。

次の例に進みましょう。

ここでも、同じ Country トランザクションがビジネスコンポーネントとして設定されているとします。プロシージャーのソースが示されています。

CountryId 項目属性は自動採番で、プロシージャーの [Commit on exit] プロパティは Yes に設定されています。

この例では、プロシージャーのソースにビジネスコンポーネントによる操作があり、データベースの更新を実行する For each があるため、[Commit on exit] プロパティの Yes の値が考慮されます。そして、ビジネスコンポーネントによる操作と For each の操作の両方がコミットされます。

したがって、Brazil という名前の国が挿入され、さらに値 CountryId = 2 を持つ国の名前が変更されます。

[Commit on exit] プロパティ - 例

4

[Business Component] = True

[Commit on exit] = Yes

```
For each Country
     Where CountryId < 3
     &Country.Load(CountryId)
     &Country.CountryName = "New country name"
     &Country.Update()
endfor
```

[Commit on exit] プロパティは無視されます。

最後の例を見てみましょう。

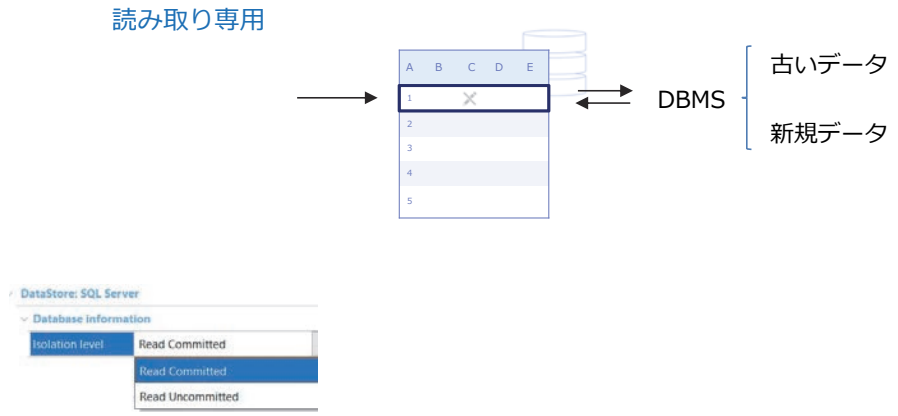
この場合も、ビジネスコンポーネントとして設定されている同じ Country トランザクションについて考えます。プロシージャーのソースが示されています。

CountryId 項目属性は自動採番で、プロシージャーの [Commit on exit] プロパティは Yes に設定されています。

この場合も、[Commit on exit] プロパティの値 Yes は無視されます。For each で直接ではなく、ビジネスコンポーネントを介して更新が試みられるためです。このビジネスコンポーネントが存在しない場合、For each 自体はなんのアクションも実行しません。
したがって、Commit コマンドを明示的に追加する必要があります。

4 つの例を確認しました。一般的には、[Commit on exit] プロパティが Yes に設定されているオブジェクトの場合、完了時にコミットが実行されると考えられます。ただし、ビジネスコンポーネントを介してのみでなく、このオブジェクトによりデータベースの更新が実行される必要があります。

[Isolation level] プロパティ



それでは、ほかのトピックについても考えてみましょう。

ここでは並行性制御について詳しくは触れません。ただし、複数のユーザーがデータベースに対して操作を実行するとき、読み取る情報が別の書き込みプログラムによりブロックされた場合、表示される値は DBMS によって異なります。クエリされているデータの古い値と新しい値のどちらを表示するかは DBMS によって異なります。

この動作はデータストアレベルで [Isolation level] プロパティにより制御されます。

このプロパティを使用して、プログラムにより実行される変更のアイソレーションレベルを指定できます。有効値は次のとおりです。

- [Read Committed]: コミットが実行されるまで、ほかのユーザーが行った変更はプログラムに表示されません。これはプロパティの既定値です。
- [Read Uncommitted]: この場合は、コミットの実行前であっても、ほかのユーザーが行った変更がプログラムに表示されます。

アイソレーションレベルの指定は、読み取りと並行性制御に影響します。既定値の [Read Committed] オプションを指定すると、高いレベルの整合性を確保できますが、データベースのロックが多くなります。