



Magic eDeveloper V10 遅延トランザクション

本書および添付サンプル(以下、本製品)の著作権は、マジックソフトウェアジャパン株式会社(MSJ)にあります。MSJの書面による事前の許可なしでは、いかなる条件下でも、本製品のいかなる部分も、電子的、機械的、撮影、録音、その他のいかなる手段によっても、コピー、検索システムへの記憶、電送を行うことはできません。

本製品の内容につきましては、万全を期して作成していますが、万一誤りや不正確な記述があったとしても、MSE (Magic Software Enterprises Ltd.) および MSJ はいかなる責任、債務も負いません。本製品を使用した結果、または使用不可能な結果生じた間接的、偶発的、副次的な損害(営利損失、業務中断、業務情報の損失などの損害も含む)に関し、事前に損害の可能性が勧告されていた場合であっても、MSE および MSJ、その管理者、役員、従業員、代理人は、いかなる場合にも一切責任を負いません。MSE および MSJ は、本製品の商業価値や特定の用途に対する適合性の保証を含め、明示的あるいは黙示的な保証は一切していません。

本製品に記載の内容は、将来予告なしに変更することがあります。

サードパーティ各社商標の引用は、MSE および MSJ の製品に対する互換性に関するの情報提供のみを目的となされるものです。一般に、会社名、製品名は各社の商標または登録商標です。

本製品において、説明のためにサンプルとして引用されている会社名、製品名、住所、人物は、特に断り書きのない限り、すべて架空のものであり、実在のものについて言及するものではありません。

初版 2008年3月28日

マジックソフトウェア・ジャパン株式会社

目次

1 本書について.....	5
2 遅延トランザクションの基本.....	6
2.1 遅延トランザクションとは？.....	7
2.2 遅延トランザクションの特徴.....	8
2.2.1 トランザクションの基本要件.....	8
2.2.2 分離レベル.....	8
2.2.3 遅延トランザクションの利点.....	8
2.3 物理トランザクション.....	9
2.4 遅延トランザクション.....	10
2.5 トランザクションの設定.....	11
2.6 ロールバック.....	12
3 排他制御.....	13
3.1 不正更新の問題.....	14
3.2 物理トランザクションの場合.....	15
3.3 遅延トランザクションの場合.....	16
3.4 更新レコードの識別.....	17
3.5 更新レコードの識別特性の動作.....	18
3.5.1 「位置」の場合.....	18
3.5.2 「位置と選択項目」の場合.....	18
3.5.3 「位置と更新項目」の場合.....	18
3.6 差分更新.....	19
3.7 差分更新の設定.....	20
4 一時テーブルの利用.....	21
4.1 物理トランザクションの場合.....	22
4.2 物理トランザクションを使った場合のプログラム構造.....	23
4.3 遅延トランザクションを利用するやりかた.....	24
4.4 遅延トランザクションを使った場合のプログラム構造.....	25
4.5 物理トランザクションから遅延トランザクションへの移植.....	26
4.6 物理トランザクションでの重複チェックのタイミング.....	28
4.7 遅延トランザクションでの重複チェックのタイミング.....	29
5 トランザクションのネスト.....	30
5.1 トランザクションのネストはいつ起こるか.....	31
5.2 ネストトランザクションの動作例1.....	32
5.3 ネストトランザクションの動作例2.....	33
5.4 ネストしたトランザクションのロールバック.....	34
5.5 ロールバック後の動作.....	35
5.6 ネストしたトランザクションのコミット後のロールバック.....	36
5.7 ネストトランザクションの分離.....	37
5.8 ネストしたトランザクションの設定.....	39
5.8.1 例1：明細更新バッチタスク.....	39
5.8.2 例2：新受注番号発番.....	39

6 外部キーと同期パラメータ	41
6.1 外部キー制約とは.....	42
6.2 外部キー制約の定義.....	43
6.3 ヘッダ・明細レコード間の外部キー制約.....	44
6.4 物理トランザクションの場合.....	45
6.4.1 レコード登録時.....	45
6.4.2 レコード削除時.....	45
6.5 遅延トランザクションと同期パラメータ.....	46
6.6 同期 = No の場合の動作.....	47
6.7 同期 = Yes の場合の動作.....	48
6.8 サブフォームの場合.....	49
7 トランザクションキャッシュのログ	50
7.1 トランザクションキャッシュのログの表示設定.....	51
7.2 トランザクションキャッシュのログの監視.....	52

1 本書について

本書では、Magic eDeveloper V10 の独自のデータ管理機能である遅延トランザクションの基礎を学ぶことを目的としています。

本書の読者は、Magic eDeveloper V10 の基本的な操作・設定・プログラミング等についてすでによく知っていることを前提にしています。また、SQL データベースを使った Magic システム開発についても理解していることを前提にしています。

これらの前提知識については、以下の書籍が参考になります。いずれも、弊社ホームページのスキルアップセンター <http://www.magicsoftware.co.jp/training/introduction/introduction.html> よりダウンロードすることができます。

書籍名	内容
Getting Started V10	Magic eDeveloper V10 を始めて利用される方を対象に、スタンドアロンのオンラインアプリケーションをステップバイステップで作りながら Magic の基本を学んでいきます。Magic の初歩から、タスクの動作、フォームの設計、データソースの定義、イベント指向エンジン、1対1リレーション、1対多リレーション、バッチタスク、帳票印刷、メニュー作成までを学びます。
Magic eDeveloper V10 チュートリアル SQL 編	本チュートリアルでは、SQL データベースを使って Magic アプリケーションを作成するための基本事項を勉強します。SQL データベースとしては SQL Server 2005 を使い、インストール、Magic のデータベースの設定、データソースリポジトリの扱い、Pervasive からの移行、ロックとトランザクション、一時ファイルを使ったプログラミング手法などについて学びます。
Magic eDeveloper V10 コーディングサンプル	本書はより本格的なアプリケーションに近い Magic アプリケーションのコーディングサンプルです。Getting Started V10、Magic eDeveloper V10 チュートリアル SQL 編を終了し、より上級の Magic 開発者となることを目指している方を対象にしています。

また、遅延トランザクションについては、以下の資料も参考にしてください。

書籍名	内容
データ管理と Magic eDeveloper	遅延トランザクションについての概論。製品 CD の Online フォルダに、Data Management.pdf として格納されています。また、弊社ホームページのホワイトペーパーのページ http://www.magicsoftware.co.jp/products/brochureandwhitepaper.html からダウンロードすることができます。
リファレンスヘルプ	製品添付のヘルプファイルに、リファレンスマニュアルがあります。この「データ管理 → 遅延トランザクション」に詳しい説明があります。

2 遅延トランザクションの基本

本章では、遅延トランザクションの基本として、概念とその利点について説明します。



Magic では、遅延トランザクションと区別するために、DBMS が提供するトランザクション機能を **物理トランザクション** と呼びます。

2.1 遅延トランザクションとは？

遅延トランザクションというのは、Magic が独自に実装しているトランザクション機能です。

トランザクションの利用は、データの保全のために必須と言ってもよいものとなりました。通常のアプリケーションでは、DBMS が提供しているトランザクション機能を利用します。

また、トランザクションの利用に関連して必ず出てくる問題がデータのロックの問題です。これは複数ユーザが同時に利用している環境での、同時並行利用性とデータの不正更新防止という重要な二つの要件に、決定的な影響を与えます。

従って、ロックとトランザクションを正しく設計することはアプリケーションの安定動作に非常に重要なこととなります。

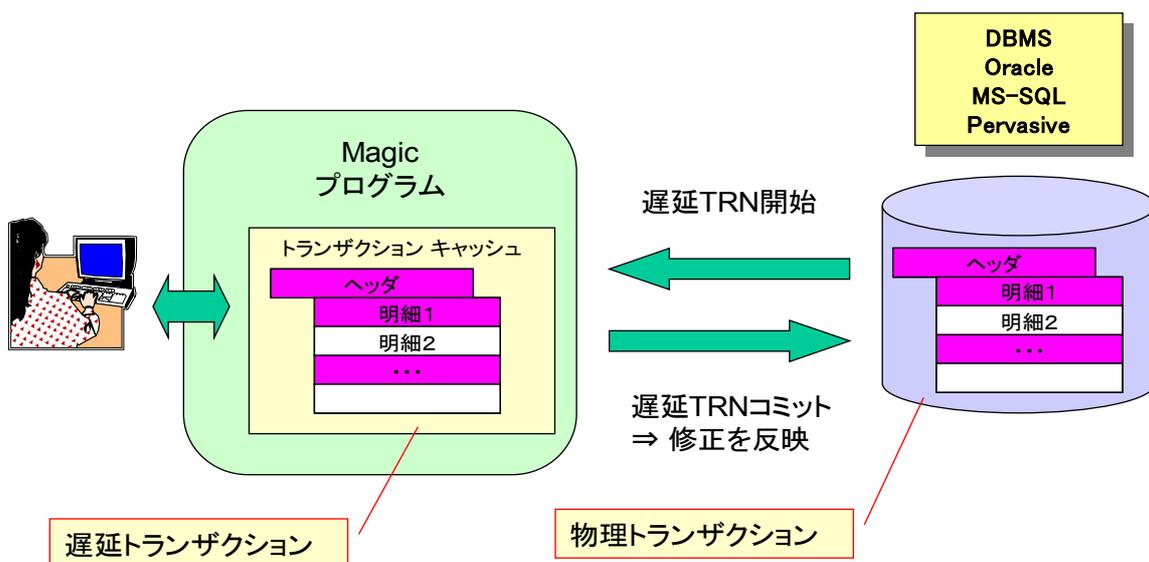
しかし、DBMS が提供するトランザクション機能は、

- 各社各様の拡張や細かなオプションがあり、それにより微妙に動作が異なる。
- 制御のための SQL 文は、DBMS ごとに全く異なっている。

という現実があり、正しく設計するには、各 DBMS ごとのトランザクション機能についての正確かつ詳細な理解が必須となり、開発者の負担が大きくなっていました。

Magic の遅延トランザクションは、このような問題を解決するために、Magic が独自に実装したトランザクション機能で、次のように機能します：

- 遅延トランザクションは、タスク、グループ（バッチタスクの場合のみ）、レコードレベルで開始・終了（コミット、またはロールバック）できます。
- 遅延トランザクション開始時には、DBMS へのトランザクションは開始されません。
- 遅延トランザクション中に発生したデータの修正（作成、削除を含む）は、すべて Magic のメモリ内のトランザクションキャッシュに格納されます。この間、DBMS へは、DML 文（UPDATE、INSERT、DELETE 文など）が発行されません。また、読み込んだレコードに対するロックも掛けられません。
- 遅延トランザクションがコミットされた時点で、トランザクションキャッシュ内にあるすべての修正内容が、一度に、DBMS に反映されます。具体的には、Magic エンジンが次のことを行います。
 - DBMS のトランザクション（物理トランザクション）を開始します。
 - DML 文を発行して、DBMS のデータを修正します。
 - DBMS の物理トランザクションをコミットします。
- 遅延トランザクションがロールバックされる場合には、トランザクションキャッシュの内容が破棄されます。



2.2 遅延トランザクションの特徴

2.2.1 トランザクションの基本要件

遅延トランザクションは、一般的な概念で言う「トランザクション」の4つの基本要件をすべて満たしています。

- **原子性 (ATOMICITY)** : トランザクションは、それ以上分割することのできない最小の作業単位である、ということです。トランザクションを構成する処理の結果がすべて有効になるか、またはすべて無効になるかのいずれかでなければなりません。
- **一貫性 (CONSISTENCY)** : トランザクションの実行前と実行後でデータの整合性を持ち、データの一貫性を確保しなければなりません。
- **隔離性 (ISOLATION)** : 処理対象が同じデータである複数のトランザクションを一度に実行する場合は、それぞれのトランザクションは隔離された状態でデータの変更を行わなければなりません。すなわち、一方の変更と他方の変更とが混在されたような結果になってはならないということです。
- **持続性 (DURABILITY)** : トランザクションがコミットされ、ユーザに成功が通知された後には、その変更はいつまでも残り、取り消されない、ということです。これはシステム障害に対する耐性があることを意味します。

2.2.2 分離レベル

遅延トランザクションは、それぞれが完全に独立していて、コミット前のデータが他のトランザクションから見えることはありません。すなわち、いわゆる「ダーティリード」が起りません。このため、遅延トランザクションは「コミット済み読み取り (READ COMMITTED)」の分離レベルを実現しています。

2.2.3 遅延トランザクションの利点

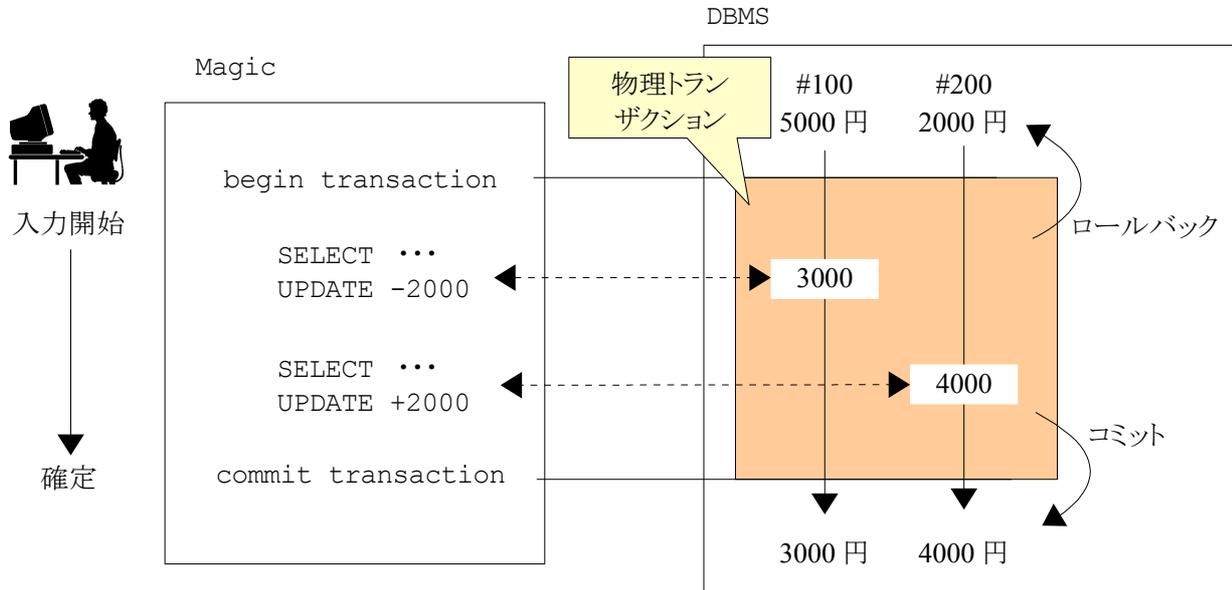
Magic はもともとデータベース独立性が高く、個々の DBMS の違いについてあまり煩わされずにプログラムを作ることができるのですが、遅延トランザクションを利用することにより、通常の物理トランザクションを使ったプログラム開発に比べ、次のような利点を得ることができます。

- トランザクションを利用するプログラム開発が容易になる: トランザクションキャッシュが一時テーブルの代わりとなるので、一時テーブルの作成・管理が不要になります。また、DBMS 独自のトランザクションの微妙な違いに煩わされずにプログラム開発を行うことができます。
- 楽観的ロックを簡単に実現できる: 遅延トランザクションは、いわゆる楽観的ロックを基本にしています。これにより、複数ユーザ環境での実行において、不正更新を防ぎつつ、並列度を高くすることができます。
- DBMS に対する負荷を最小限にすることができる: 遅延トランザクションでは、トランザクション中で修正のあったレコードを自動的に判別して、必要最小限の DML 文を発行します。また、物理トランザクションは、遅延トランザクションのコミットのタイミングで、ごく短時間で終了します。
- トランザクションのネストが可能になる: トランザクションの中で別のトランザクションを開始/終了するという、トランザクションのネスティングができるようになります。
- 外部キー制約 (外部キー制約) に対応できる: ヘッダ・明細構造のテーブルでは、外部キー制約が設定されることがありますが、遅延トランザクションを使えば、この制約のかかっているテーブルに対しても、自動的に適切な順序で DML 文が発行されます。

2.3 物理トランザクション

ここで、遅延トランザクションの動作との対比を明らかにするために、物理トランザクション(DBMS が実装するトランザクション機能)の動作について、簡単におさらいしておきます。

下図は、銀行口座#100 から#200 へ 2000 円を振り込む場合のトランザクションを簡単に示したものです。



1. 最初に、利用者が振込みを始めようとしたタイミングで、トランザクションが開始されます。トランザクションの開始は、「begin transaction」などの DML 文により、DBMS に通知されます。
2. まず、振込み元の口座#100 の現在の残高を SELECT 文で読み込みます。ここでは 5000 円があります。
3. ここから、UPDATE 文を使って、2000 円を減額します。
4. 次に、振込み先の口座#200 の現在の残高を SELECT 文で読み込みます。ここでは 2000 円があります。
5. ここに、UPDATE 文を使って、2000 円増額します。
6. 最後にトランザクションをコミットします。コミットは「commit transaction」などの DML 文によりなされます。

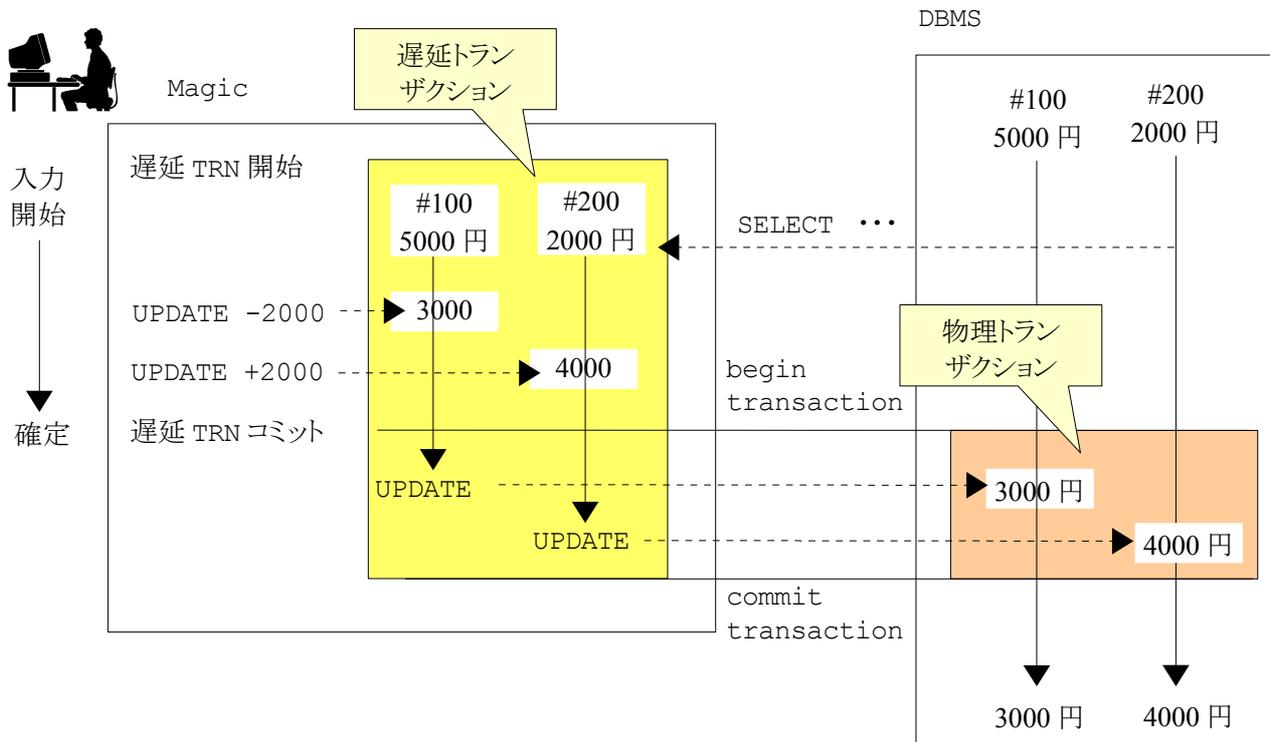
このような一連の操作により、1 回の振込みが実行されることとなります。

万一、トランザクションの途中になんらかのエラーが発生したり、あるいはコンピュータが動作を停止した場合には、トランザクションはロールバックされます。このときには、すべての DML 文は破棄され、口座はいずれもトランザクション開始直前の状態のままとなっています。

上の例では、ユーザが振込みを始めようとした時点でトランザクションが開始され、確定した時点でトランザクションがコミットされています。従って、トランザクションは人手で入力するための時間の間続くこととなります。これは、数十秒で済むかもしれないし、数分かかるかもしれないし、あるいは、途中で割り込み作業が入って何時間もかかるかもしれません。

2.4 遅延トランザクション

同じ振り込み作業を、遅延トランザクションを使って実現した場合の動作は、下図のようになります。



1. 最初に、利用者が振込みを始めようとしたタイミングで、遅延トランザクションが開始されます。この際、Magic がトランザクションキャッシュの内容を初期化するだけで、DBMS へは物理トランザクションの開始が通知されません。
2. 振込み元の口座#100 の現在の残高を SELECT 文で読み込みます。ここでは 5000 円があります。このレコードをトランザクションキャッシュに保存しておきます。
3. トランザクションキャッシュのレコードから 2000 円を減額し、3000 円とします。
4. 次に、振込み先の口座#200 の現在の残高を SELECT 文で読み込み、やはりトランザクションキャッシュに保存します。ここでは 2000 円があります。
5. ここに、UPDATE 文を使って、2000 円増額し、4000 円とします。
6. ユーザが入力を確定し、遅延トランザクションをコミットするタイミングで、Magic は DBMS に対するデータ変更を行います。すなわち、次のことを実行します。
 - ① 物理トランザクションを開始します。
 - ② UPDATE 文で、トランザクションキャッシュに保存されていたデータの変更分を、DBMS に反映させます。
 - ③ 物理トランザクションをコミットします。

このとき、遅延トランザクションは、ユーザの入力開始から確定までの間(長時間になるかもしれない)続きますが、DBMS に対する物理トランザクションは、ほぼ一瞬のうちに終了します。

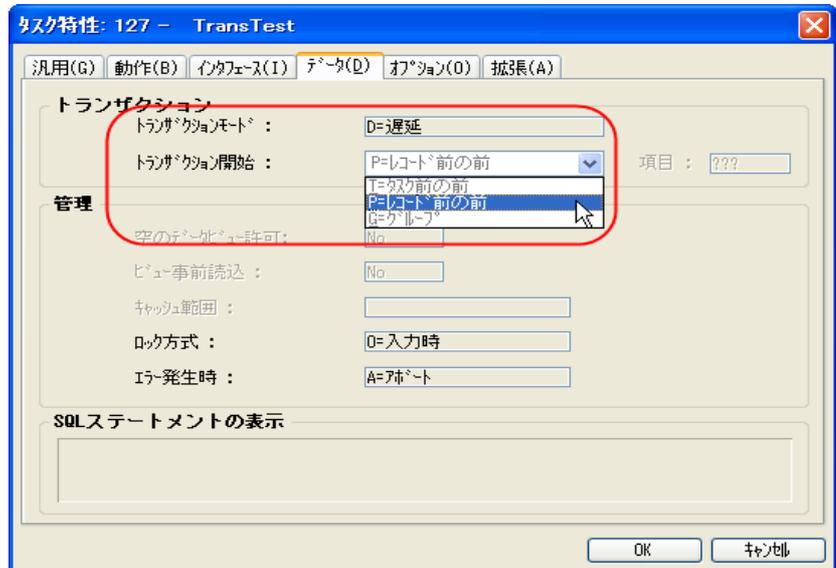
2.5 トランザクションの設定

Magic では、トランザクションの設定をタスク特性の「データ」タブで設定します。(下図)

ここで設定できるトランザクション関係のパラメータは、次の二つがあります。

- トランザクションモード：物理、遅延、その他のモードを設定します。
- トランザクション開始：トランザクションの開始のタイミングを設定します。

トランザクション開始パラメータの設定オプションは、トランザクションモードの設定によって異なります。以下に、「トランザクションモード」と「トランザクション開始」パラメータの組み合わせを示します。



トランザクションモード	トランザクション開始
D=遅延	T=タスク前
	P=レコード前
	G=グループ (バッチタスクのみ)
N=ネスト遅延	T=タスク前
	P=レコード前
	G=グループ (バッチタスクのみ)
W=親と同一	(設定はできるが、実行時は無効なので省略)
P=物理	T=タスク前
	G=グループ (バッチタスクのみ)
	L=レコードロック時
	P=レコード前
	S=レコード後
	U=レコード更新時
	N=なし

このパラメータの組み合わせについては、第5章 トランザクションのネスト で再度説明します。

2.6 ロールバック

遅延、物理いずれのトランザクションにおいても、Magic エンジンでは、回復不可能なエラーが発生した場合に自動的にトランザクションをロールバックします。

また、Magic 開発者が、明示的にトランザクションをロールバックさせたい場合もあります。たとえば、「取り消し」ボタンに対するイベントハンドラの中で、トランザクションをすべて取り消して、初期状態に戻す、という機能を実装する場合などです。

この場合には、Magic の組み込み関数 Rollback を使います。Rollback 関数の仕様は、以下のとおりです。

構文: Rollback (論理値, ネストレベル)

機能: トランザクションのロールバック。ネストレベルを指定し、その位置までトランザクションをロールバックします。

パラメータ: **論理値:** 確認メッセージの表示の有無を指定します。
True … ロールバックの際に「トランザクションをロールバックしますか?」という確認メッセージが表示されます。
False … ロールバックの際に確認のメッセージは表示されません。
ネストレベル: 数値。どのレベルまでロールバックするかを示す以下の数値です。
1 … 最も内側のネストトランザクションをロールバックします。
2 … 最も内側のすぐ上位のトランザクションをロールバックします。
0 … 最も外側のトランザクションをロールバックします。

戻り値: 論理値

- True…トランザクションがロールバックされた場合
- False…トランザクションが始まっていなかった場合

例: Rollback ('TRUE' LOG, 0)

3 排他制御

マルチユーザ環境でいつも課題となることに、不正更新の防止があります。これは、二人のユーザが同時に同じレコードを更新してしまうと、最初の人を書き込んだレコードが、後の人の書きこんだレコードで上書きされてしまい、データが正しくなくなってしまう問題です。

不正更新を防止するには、「ロック」のメカニズムを使いますが、ロックのメカニズムとして、主として次のような2種類に大別できます。

- 悲観的ロック (Pessimistic Lock)
- 楽観的ロック (Optimistic Lock)

Magic では、物理トランザクションにおいては悲観的ロックを使い、遅延トランザクションにおいては楽観的ロックを使います。

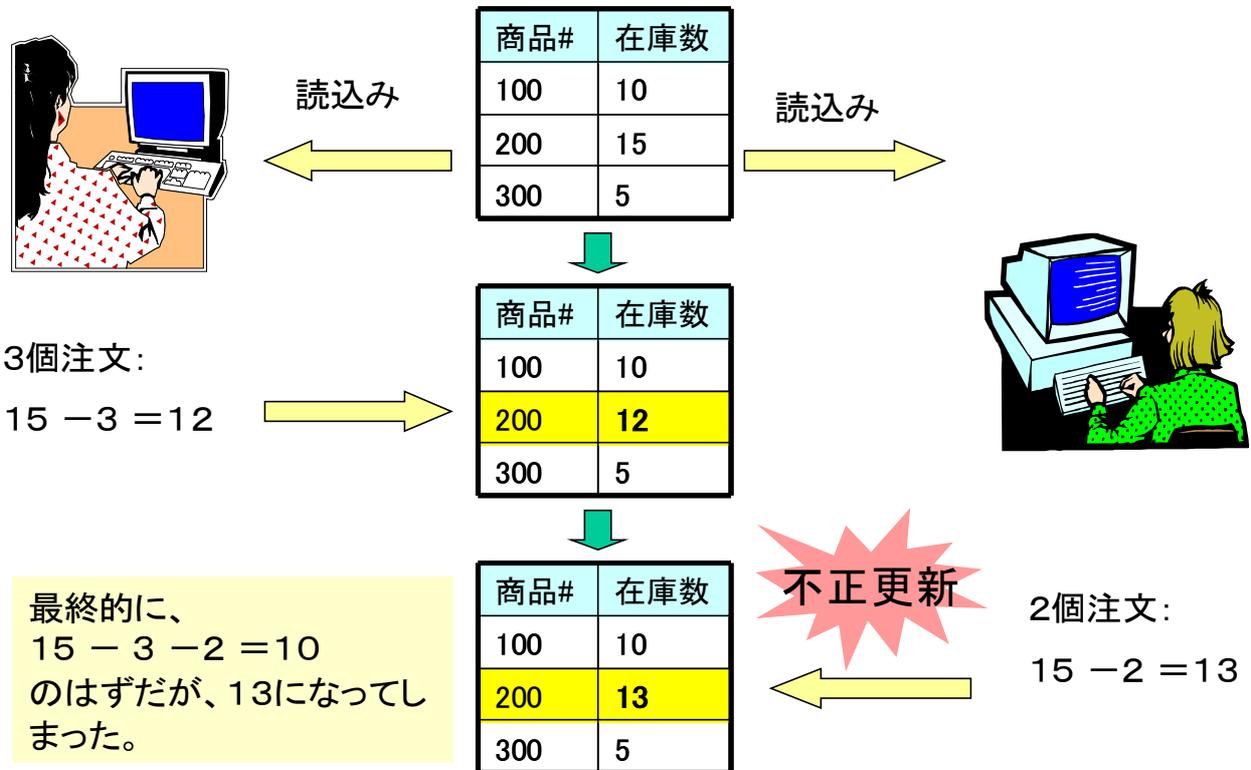
本章では、不正更新の問題と、Magic における悲観的/楽観的ロック、および関連する話題について説明します。



オンラインタスクでは、遅延トランザクションでも、Magic ロックを使って悲観的ロックをかけることができますが、ここでは説明を省略します。

3.1 不正更新の問題

最初に、複数ユーザが同時実行している場合の不正更新の問題とは、どのような問題かを説明します。下図は、二人のユーザが同一の商品に対して注文を受けている場面です。



次のような順序でレコードが更新されたとしましょう。

1. ユーザ1(左側)が、商品#200の現在の在庫数を読み込む。ここでは15個残っている。
2. ユーザ2(右側)が、同じく商品#200の在庫数を読み込む。やはり15個残っている。
3. ユーザ1が、3個の注文を受けたので、在庫数を $15 - 3 = 12$ として、書き込む。
4. ユーザ2は、2個の注文を受けたので、在庫数を $15 - 2 = 13$ として、書き込む。

本当は、それぞれ3個、2個の注文を受けたので、 $15 - 3 - 2 = 10$ が残り在庫数でなければならないのに、レコード書き込みのタイミングによっては、このような不正な結果になってしまうことがあります。これが不正更新の問題です。

このような問題の起こる原因は、上を見てするわかつおり、ユーザ2がユーザ1のレコード更新結果を無視して、自分の計算結果で上書きしてしまうところにあります。

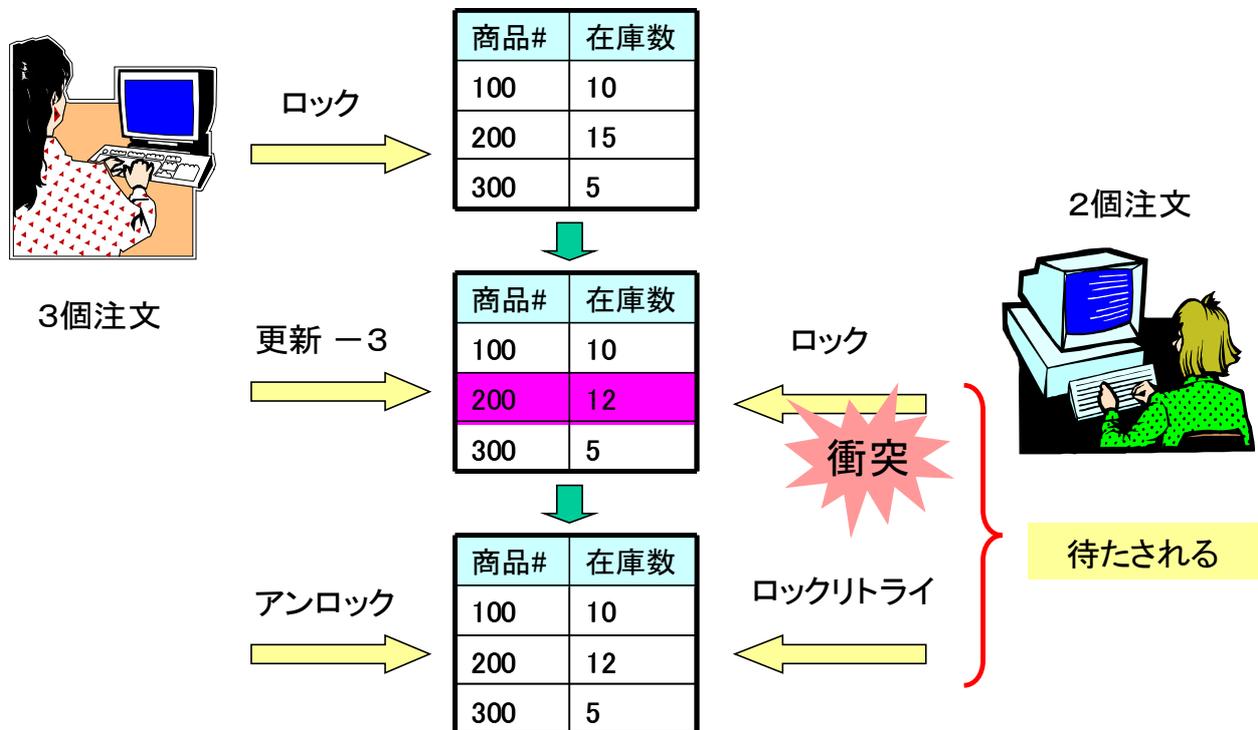
従って、この問題を回避するには、次のいずれかの方法が考えられます。

- 自分が更新を行おうとするときには、あらかじめ、レコードの排他的権利を設定し、他の人が変更できないようにしてしまう。
- 更新を行うタイミングで、自分が読み込んだ後に他の人がレコードを変更していないかチェックする。

前者が「悲観的ロック」という方法であり、後者が「楽観的ロック」という方法です。

3.2 物理トランザクションの場合

Magic では、物理トランザクションを使っている場合、悲観的ロックの方法を用い、他のユーザが変更できないように、レコードに排他的なロックをかけます。



例えば、最初のユーザが商品 #200 のレコードを読み込むと同時に、レコードロックをかけます。その後、在庫数を-3してレコードの更新を行います。

このときに、後のユーザが同じく商品 #200を読み込むと同時にロックをかけようとすると、ロックの衝突が置きます。ロックの衝突が起きた場合には、ロックが解除されるまで待たされることとなります。

最初のユーザが更新完了して、レコードをアンロックしたタイミングで、初めて後のユーザがロック・更新を行えるようになります。

このようにして、排他的なロックをあらかじめかけておくことで、不正更新が起こらないようにします。

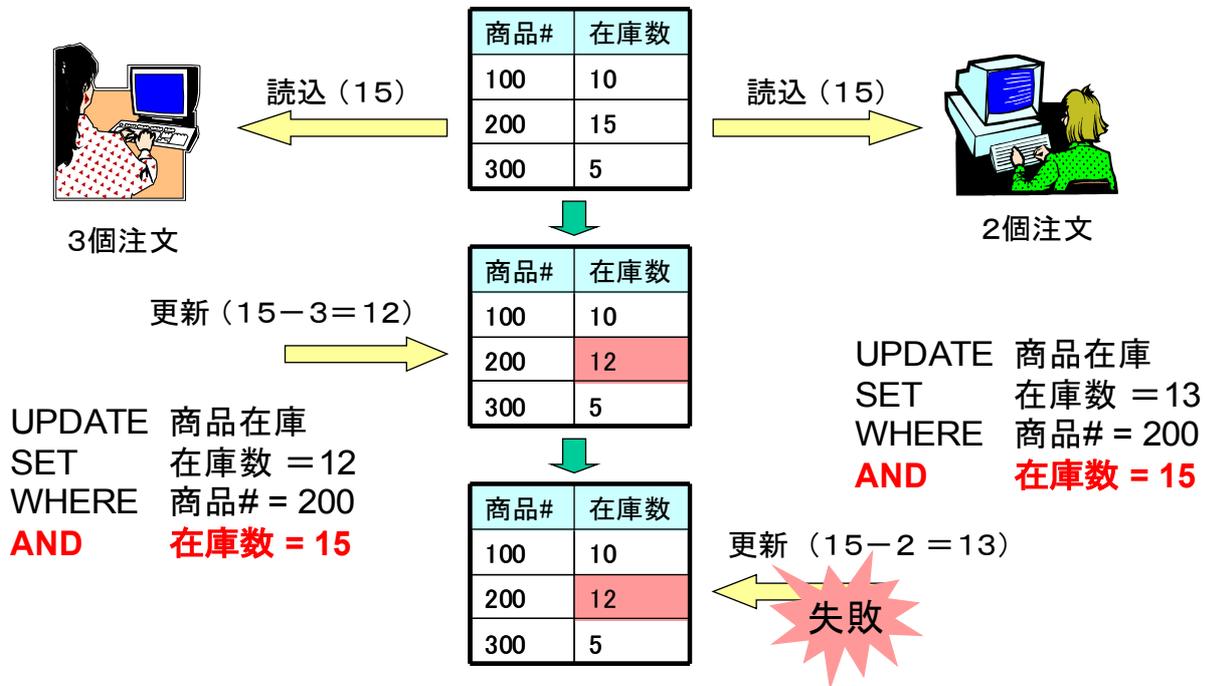
この方式では、不正更新は起こる可能性がなくなりますが、問題点として、ロックの間、他のユーザが待たされてしまう、ということがあげられます。特に、ユーザが多い場合に、誰がどのように使っているかわからない状況で、長時間ロックされたままのレコードのために、他の人の作業が止まってしまうのは困ったこととなります。いろいろな部署に電話をかけて「誰か商品 #200を使いっぱなしにしてませんか？」と問い合わせしなければならないこととなります。

3.3 遅延トランザクションの場合

Magic の遅延トランザクションでは、「楽観的ロック」という方法でこの問題に対応しています。

楽観的ロックというのは、ロックの衝突、不正更新の確率は少ない、という前提で、ロックはかけずに処理を続けさせて、最後にレコードの更新の段階に至って、他の人がレコードを更新してしまっていないかを確認する、という手法です。

下図では、先の例と同じく、二人の人が商品 #200 を同時に更新しようとしています。



このとき、ロックをかけないので、それぞれのユーザはレコードを読み込むことができます。

先のユーザが、-3 してレコードを書き込むとします。このときに、更新のための SQL 文としては、次のようなものが発行されます。

```
UPDATE 商品在庫 SET 在庫数 = 12 WHERE 商品 = 200 AND 在庫数 = 15
```

ここで注目すべきは、WHERE 句の中で「在庫数 = 15」という条件が付加されていることです。これは、最初に読み込んできたときの値が、現在の値と変わっていないことを確認するためのものです。

今の例では、まだ誰もレコードを更新していないので、この UPDATE 文は成功し、レコードが更新されます。

次に、後のユーザ2が-2してレコードを更新しようとしたとします。このときに発行される UPDATE 文は、同じく、次のようになります。

```
UPDATE 商品在庫 SET 在庫数 = 13 WHERE 商品 = 200 AND 在庫数 = 15
```

これを実行しようとする、WHERE 句の「在庫数 = 15」の条件に引っかかって、UPDATE 文が失敗します。先のユーザがレコードを更新してしまって、在庫数が12になってしまっているからです。

このようにして、楽観的ロックの方法では、ロックをかけずに不正更新を検出することができるようになります。

3.4 更新レコードの識別

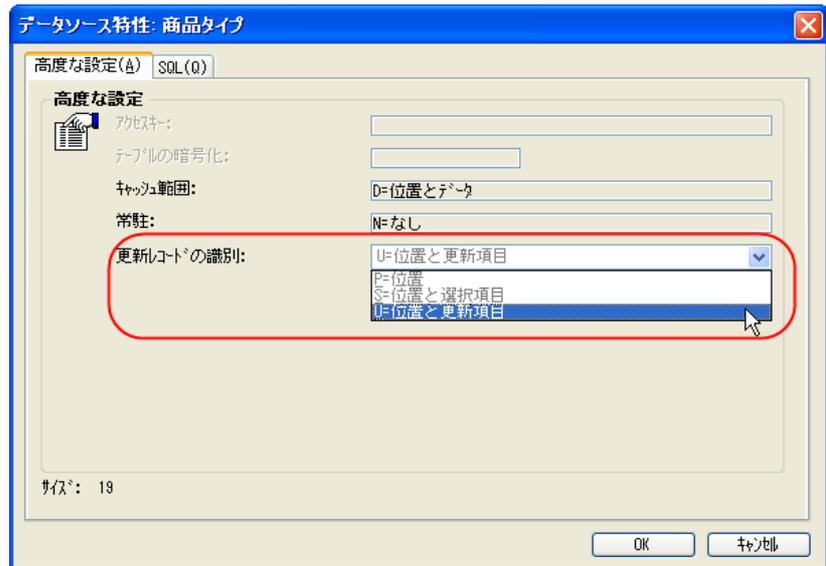
楽観的ロックの場合、更新しようとするレコードが、他のユーザによって更新されているかを確認するために、前述の例では「在庫数」のみをチェックしていました。これはこの例では必要最小限で妥当な設定です。

Magic では、更新されるレコードを識別するために、次のオプションが用意されています。

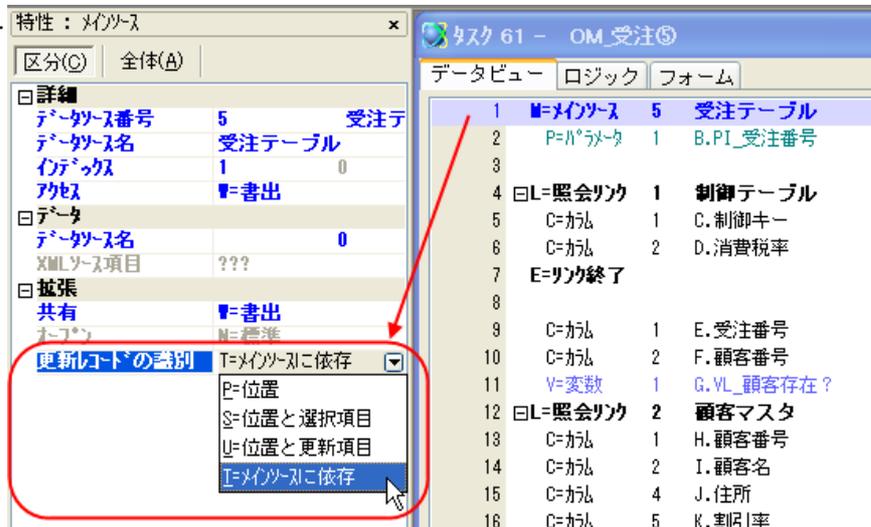
- 位置 (Pervasive の場合のデフォルト)
- 位置と選択項目
- 位置と更新項目 (SQL DBMS のデフォルト)

このオプションは、次のところで設定できます。

データリポジトリ: テーブル特性の「更新レコードの識別」特性



タスク: データビューで「メインソース」行、あるいは「リンク」行の特性の「更新レコードの識別」特性



3.5 更新レコードの識別特性の動作

「更新レコードの識別特性」の設定により、遅延トランザクションでの UPDATE 文が次のように変わってきます。

3.5.1 「位置」の場合

「位置」の場合には、次のような UPDATE 文となります。

```
UPDATE 商品在庫 SET 在庫数 = 12 WHERE 商品 = 200
```

すなわち、他のユーザによるデータ変更の確認を行いませんので、不正更新が起こったとしても、それを検出することができません。従って、この設定は、複数ユーザによる更新の衝突が起こらないと(アプリケーションの設計上、あるいは運用上)わかっている場合にだけ設定するようにしてください。この設定だと SQL 文が単純になるので、SQL 文の処理のオーバーヘッドが若干減ります。

3.5.2 「位置と選択項目」の場合

「位置と選択項目」の場合には、タスクのデータビューで選択されている項目すべてについて、データ変更がかかっていないかをチェックするような、次のような UPDATE 文となります。

```
UPDATE 商品在庫 SET 在庫数 = 12 WHERE 商品 = 200 AND 商品名 = 'ブードル' AND 商品タイプ = 'D'  
AND 単価 = 10200 AND 在庫数 = 15
```

この場合には、在庫数以外のカラムについても、他のユーザが更新していないかをチェックします。チェックとしては完全ですが、場合によっては必要ないかもしれません。

たとえば、ユーザ1が在庫数を更新し、ユーザ2が商品名を変更したとしても、エラーとする必要はありませんが、「位置と選択項目」の設定になっていた場合にはエラーとなってしまいます。

3.5.3 「位置と更新項目」の場合

「位置と更新項目」は SQL テーブルの場合のデフォルトの設定であり、通常、これが必要最小限の設定です。この設定の場合には、値に変更のあったカラムについてだけ、他のユーザによる変更がされていないかのチェックが行われます。

たとえば、タスクで在庫数だけ変更された場合には、次のようになります。

```
UPDATE 商品在庫 SET 在庫数 = 12 WHERE 商品 = 200 AND 在庫数 = 15
```

この場合、ユーザ2が在庫数を変更していたらエラーが検出されますが、商品名を変更していてもエラーとはなりません。

3.6 差分更新

楽観的ロックの方式では、不正更新を検出することはできませんが、その後の対応はアプリケーションに任されています。通常の場合には自動的に回復することはできないので、エラーメッセージを出した後、ロールバックして最初からユーザに入力をしなおしてもらう、というような対応となるでしょう。

しかしながら、できることならば、このような状況でエラーも出さず、しかも不正更新も行われたいというのがベストです。

このために、Magic には、「差分更新」という便利なオプションがあります。「差分更新」について、先と同じ例で説明します。(下図参照)

まず、二人のユーザがそれぞれ商品 #200 の記録を読み込みます。ここまでは同じです。

先のユーザ1が-3して、記録を更新します。このとき、「差分更新」の方法を使った場合には、記録更新のための UPDATE 文が次のようになります。

```
UPDATE 商品在庫 SET 在庫数 = 在庫数 - 3 WHERE 商品 = 200
```

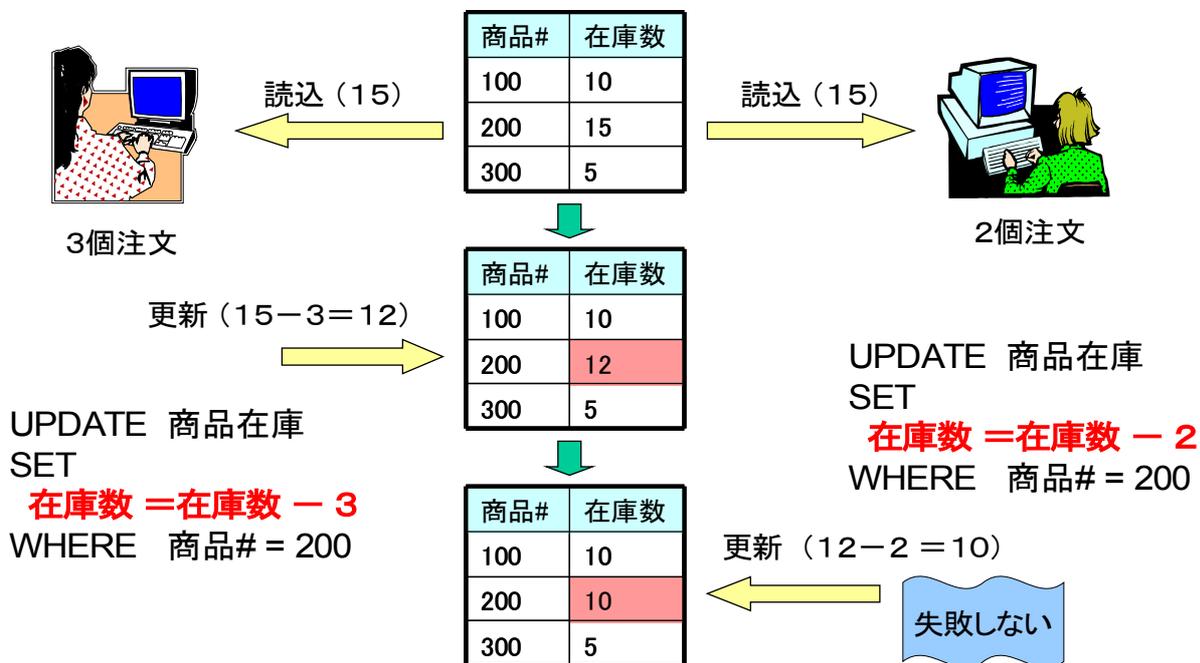
ここに見るように、UPDATE 文での在庫数の新しい値を設定する式として、12 という絶対値を設定するのではなく、「在庫数 = 在庫数 - 3」すなわち、「現在の在庫数から、差分の -3 を引いた値をセットせよ」という形になっています。この例では、その結果、在庫数は 12 になります。

後のユーザ2が 同じ記録を -2 しようとする、同じように、このような UPDATE 文が発行されます。

```
UPDATE 商品在庫 SET 在庫数 = 在庫数 - 2 WHERE 商品 = 200
```

この場合には、在庫数の新しい値は、現在値を基準にして差分である -2 を行った値として計算されるので、正しく、10 となります。

このように差分更新のオプションを使って更新を行えば、ロックをせずに同時更新をしても、正しい結果を得られるようになります。

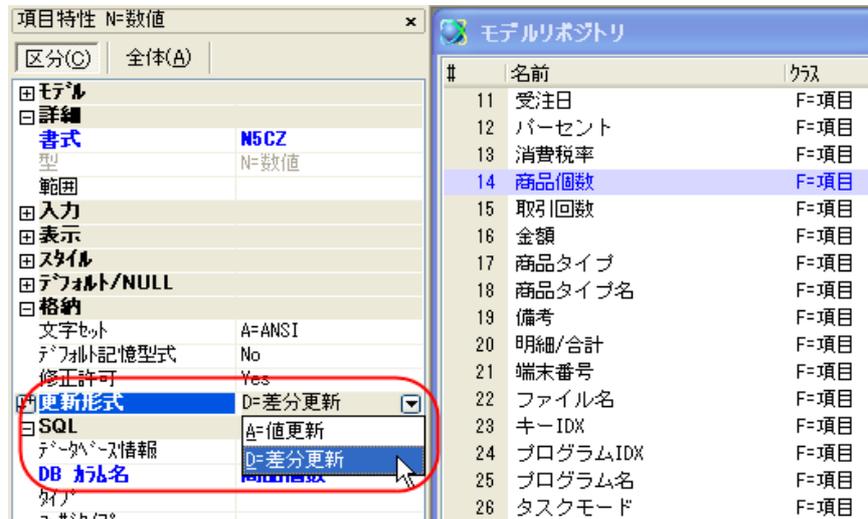


差分更新は、「在庫数」のような、累積値を表す数値型のカラムでだけ有効です。また、データベース SQL の DBMS でなければなりません。Pervasive や Memory は ISAM DBMS ですので、使えません。

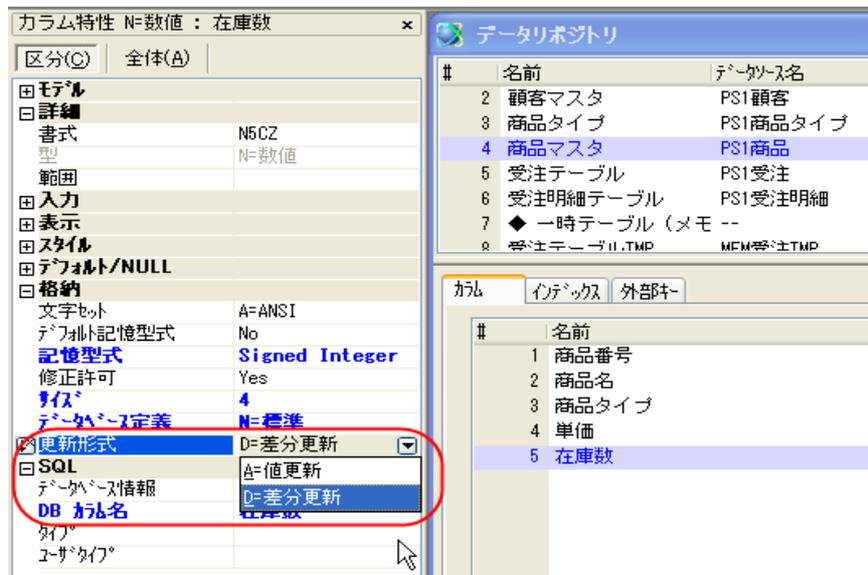
3.7 差分更新の設定

Magic で差分更新を利用するためには、次の箇所で、カラム単位で指定することができます。

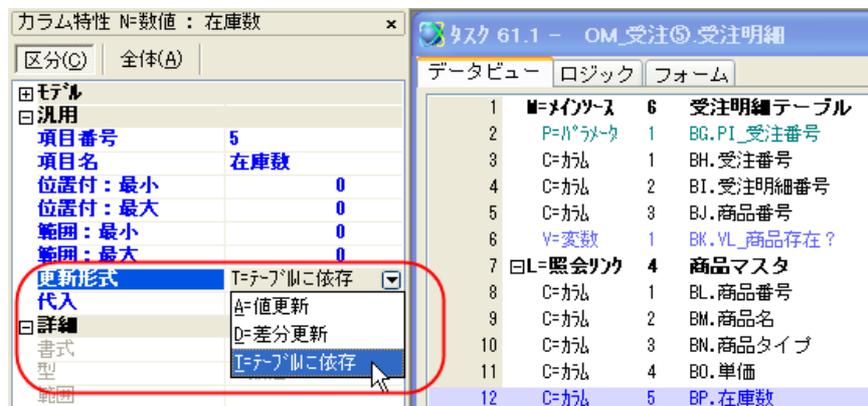
モデルリポジトリ: 数値項目モデルの「更新形式」特性で、「D=差分更新」を指定します。この特性のデフォルトは、「A=値更新」です。



データリポジトリ: カラム特性の「更新形式」特性で、「D=差分更新」を指定します。この特性のデフォルトは、「A=値更新」です。



タスク: データビュー上で、カラムの「更新形式」特性で設定します。この特性のデフォルトは「T=テーブルに依存」で、テーブルリポジトリで指定されている値が採用されます。テーブルリポジトリでの指定にかかわらず、「D=差分更新」を明示的に指定することもできます。



4 一時テーブルの利用

マルチユーザ環境では、データの一貫性を確保しながらロック待ちなどでユーザが待たされるのを極力防ぐことが必要です。

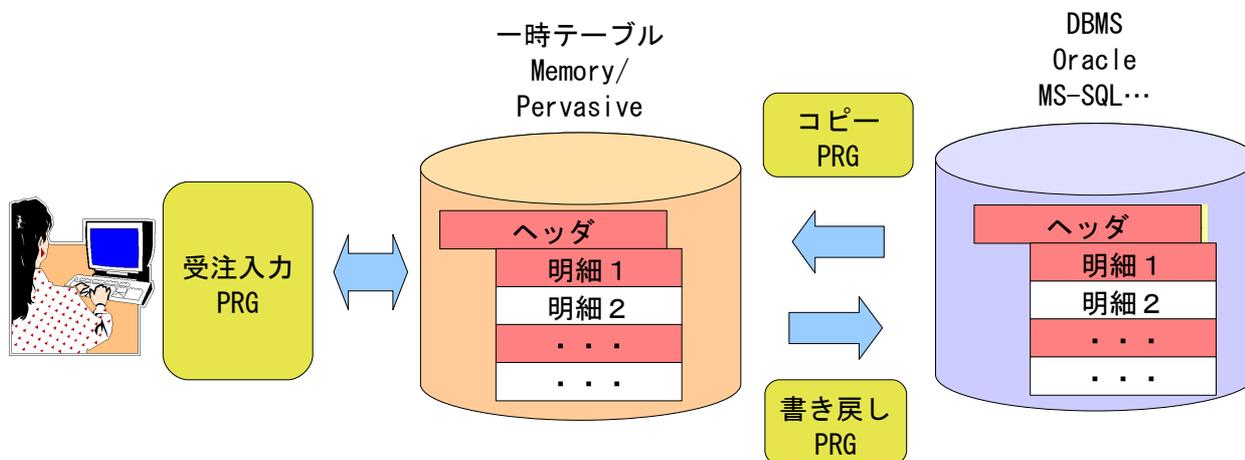
受注、発注、入在庫管理などのアプリケーションでは、いわゆる、ヘッダ・明細型の階層的なデータ構造が非常に多く使われます。Magic では、このデータ構造を操作するために、親子のタスク構造を使うのが普通です。

ヘッダ・明細型の親子タスクを作成する場合、多くのテーブルが関連するようになるのが普通であり、タスクの構造や排他制御の関連も複雑になります。このことから Magic のオンラインのシステムでは、ユーザごとの一時テーブルを使う方法が一般的に行われてきました。

本章では、従来の物理トランザクションと一時テーブルを使うオンラインアプリケーションの概要について説明した後、遅延トランザクションを使うとどのように作ることができるかを説明し、最後に物理トランザクションと一時テーブルを使うプログラムを、遅延トランザクションを使うプログラムに移植する上で気をつけるべきことを説明します。

4.1 物理トランザクションの場合

物理トランザクションを使う場合には、一時テーブルを使って、ロックによる並列実行性の低下を防いでいました。下図は、この方法を概念的に図示したものです。



- 受注のヘッダ・明細レコードは DBMS に格納されていますが、一時テーブルには Pervasive や Memory などを使い、ほぼ同様のデータ構造で定義しておきます。
- 最初に、コピープログラムが DBMS から必要なレコードを一時テーブルにコピーします。
- 次に、ユーザの入力によって、レコードが追加・修正・削除されていきます。
- この時点では、修正されるのは一時テーブルのレコードだけで、もとの DBMS のレコードは変更されていません。
- 最後にユーザが入力を確定すると、一時テーブルの内容が、書き戻しプログラムによって DBMS に反映されます。

もし、明細行やヘッダ行の変更に伴い、集計データなども更新する必要がある場合には、書き戻しプログラムでの処理の一部として実装します。

この方法では、一時テーブルに対する細かな制御を行うことができますが、コピーおよび書き戻しのバッチプログラムを別途作成する必要があります。これは通常単純なバッチプログラムではありますが、やはり生産性を下げる要因の一つとなります。

4.2 物理トランザクションを使った場合のプログラム構造

物理トランザクションと一時トランザクションを使った、ヘッダ・明細の親子プログラムの作り方は、大略次のようになります。

データリポトリ: ヘッダと明細のテーブルと同様なカラム定義の一時テーブルを定義します。

ここでは、一時テーブルのデータベースを Memory にしています。

#	名前	データソース名	データベース	フォルダ	公開名
3	商品タイプ	PS1商品タイプ	MSSQL2005		
4	商品マスタ	PS1商品	MSSQL2005		
5	受注テーブル	PS1受注	MSSQL2005		
6	受注明細テーブル	PS1受注明細	MSSQL2005		
7	◆ 一時テーブル (メモリ)	--	Memory		
8	受注テーブルTMP	MEM受注TMP	Memory		
9	受注明細テーブルTMP	MEM受注明細TMP	Memory		

#	名前	型	書式
1	受注番号	N=数値	8P0Z
2	顧客番号	N=数値	5Z
3	受注日	D=日付	YYYY/MM/DD
4	最終明細番号	N=数値	3Z
5	明細合計額	N=数値	N10CZ
6	受注割引額	N=数値	N10CZ
7	消費税額	N=数値	N10CZ
8	受注合計額	N=数値	N10CZ

プログラム: 一時ファイルの管理のため、いくつかのバッチプログラムを利用します。(赤枠の中)

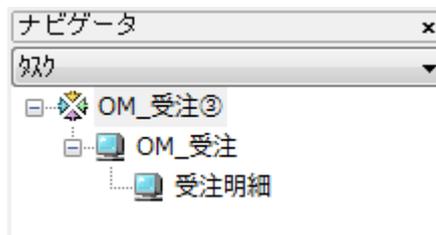
オンラインの入力プログラムは、52番「OM_受注③」です。

#	名前	フォルダ	公開名
45	-- 物理/一時Memファイル利用 --	ONL/物理/MEMテーブル	
46	BQ_受注存在チェック	ONL/物理/MEMテーブル	
47	OT_受注存在チェック	ONL/物理/MEMテーブル	
48	BC_受注TMPコピー	ONL/物理/MEMテーブル	
49	BM_受注番号発番	ONL/物理/MEMテーブル	
50	BC_受注TMP書戻し	ONL/物理/MEMテーブル	
51	BD_受注削除	ONL/物理/MEMテーブル	
52	OM_受注③	ONL/物理/MEMテーブル	ONL_受注2
53		ONL/物理/MEMテーブル	

入力プログラム: プログラムは、ルートにバッチタスクを置き、子タスクと孫タスクがヘッダ・明細タスクとなっています。

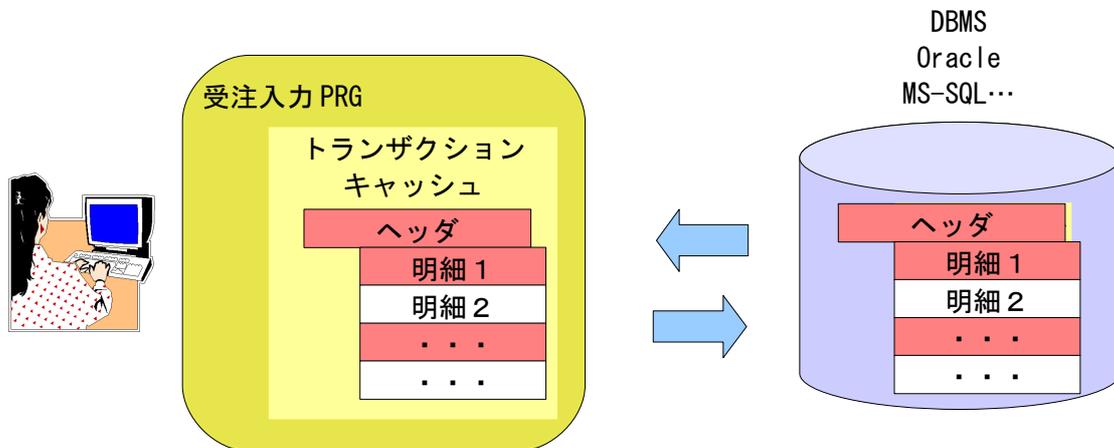
ルートのバッチタスクでは、一時テーブルの読み込み、書き込み、その他、印刷、削除、検索などの処理を担当します。

ルートのバッチタスクはトランザクションなし、子タスクでレコードレベルトランザクション、孫タスクは「親と同一」の設定とします。



4.3 遅延トランザクションを利用するやりかた

遅延トランザクションを使うと、一時ファイルの作成・管理を Magic が自動的に行ってくれるようになります。先ほどの受入力プログラムを、遅延トランザクションを使って実装すると下図のようになります。



- この場合には、Pervasive や Memory の一時テーブルというものを作りません。その代わりに、Magic が一時テーブルの代わりにトランザクションキャッシュを使って、レコードを管理します。
- 最初にプログラムがレコードフェッチをすると、DBMS のレコードがトランザクションキャッシュに読み込まれます。
- ユーザが入力・修正すると、このトランザクションキャッシュの中のレコードが修正されていきます。
- 最後にユーザが入力を確定すると、遅延トランザクションがコミットされ、トランザクションキャッシュの中の修正内容が DBMS に反映されます。

このように、遅延トランザクションを使えば、一時テーブルの定義や、コピー・書き戻しプログラムの作成が必要なくなり、プログラム作成が簡単になります。

ヘッダ・明細行の変更に伴う集計データなどの更新は、ヘッダレコードあるいは明細レコードのレコード後処理で行うことになります。この集計データへの変更も、トランザクションキャッシュに蓄えられて、遅延トランザクションのコミットのタイミングで、ヘッダ・明細レコードの書き出しと合わせて DBMS に反映されます。

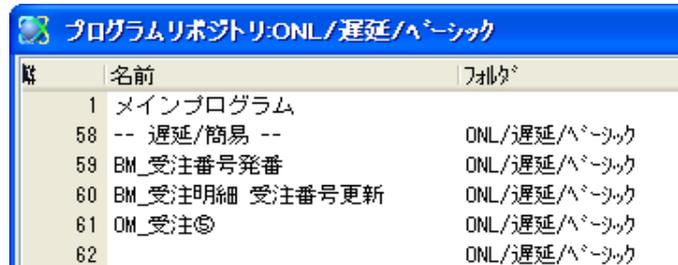
4.4 遅延トランザクションを使った場合のプログラム構造

遅延トランザクションを使ったプログラムの作り方は、大略次のようになります。

データリポジトリ: 遅延トランザクションを使う場合には、前節の説明からもわかるように、一時テーブルを利用しません。従って、データリポジトリに一時テーブルを定義する必要がありません。

プログラム: 一時ファイルの管理のためのバッチプログラムは必要ありません。

オンラインの入カプログラムは、61番「OM_受注⑤」です。

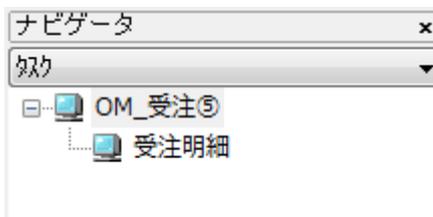


順	名前	フォルダ
1	メインプログラム	
58	-- 遅延/簡易 --	ONL/遅延/ハネック
59	BM_受注番号発番	ONL/遅延/ハネック
60	BM_受注明細 受注番号更新	ONL/遅延/ハネック
61	OM_受注⑤	ONL/遅延/ハネック
62		ONL/遅延/ハネック

入カプログラム: プログラムは、必要最小限の親子オンラインタスクだけで済みます。

親タスクで遅延トランザクションをレコードレベルで開始し、子タスクは「親と同一」にします。

印刷などの処理は、親タスクのイベントハンドラで実行します。



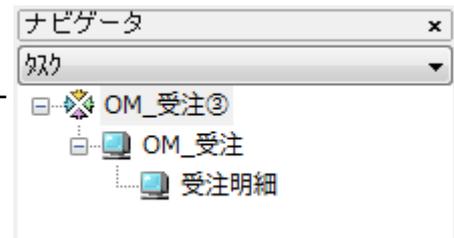
4.5 物理トランザクションから遅延トランザクションへの移植

すでに物理トランザクションを利用し、一時ファイルを使うように作成されたプログラムを、遅延トランザクションを利用するように移植しようとするとき、工数を削減するために、できるだけプログラムに変更を加えずにすませたいと思うでしょう。

遅延トランザクションは、それ自体でトランザクションキャッシュという、一時テーブルと同様な目的の機能が備わっているため、それにさらに一時テーブルを使う、というような形になるため、屋上屋を重ねるようなこととなりますが、誤動作を引き起こすことはありません。工数削減のため修正箇所を極力少なくしたいのならば、そのような形で移植することも可能です。

このときには、トランザクションの設定に注意する必要があります。ここではトランザクションの設定について説明します。

4.2「物理トランザクションを使った場合のプログラム構造」で説明したように、一時テーブルを使うヘッダ・明細プログラムは、ルートにバッチタスクを持つ親・子・孫の3段階の構造になります(右図)。一時テーブルを操作(コピー／書き戻し)するバッチタスクは、ルートタスクから呼び出します。



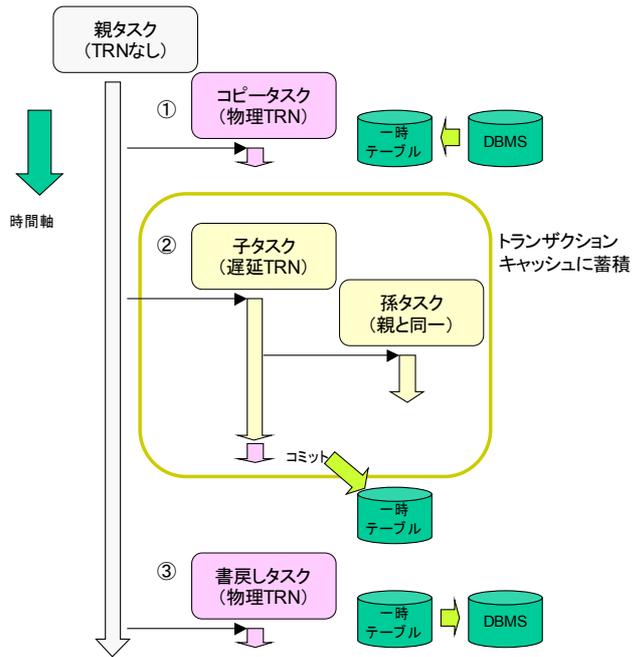
このときに、トランザクションの構造は次のようにしておきます。

タスク	トランザクション設定
ルートタスク	<p>なし</p>
子タスク(ヘッダレコード)	<p>レコードレベル 遅延トランザクション</p>
孫タスク(明細レコード)	<p>親と同一</p>

この設定にすると、右図のような流れで実行が進むので、正しくデータが処理されます。

- ① 最初に、親タスクが、一時テーブルへのコピーをするバッチタスクを呼び出します。このタスクはタスクレベルの物理トランザクションで、タスク終了とともにデータがコミットされます。
- ② 子と孫のオンラインタスクでは、遅延トランザクションで実行します。入力データはトランザクションキャッシュに蓄積されますが、子タスクが終了するタイミングで一時テーブルにコミットされます。
- ③ 親タスクから書き戻しのバッチタスクを呼び出します。これで、一時テーブル中のデータがDBMSに反映されます。

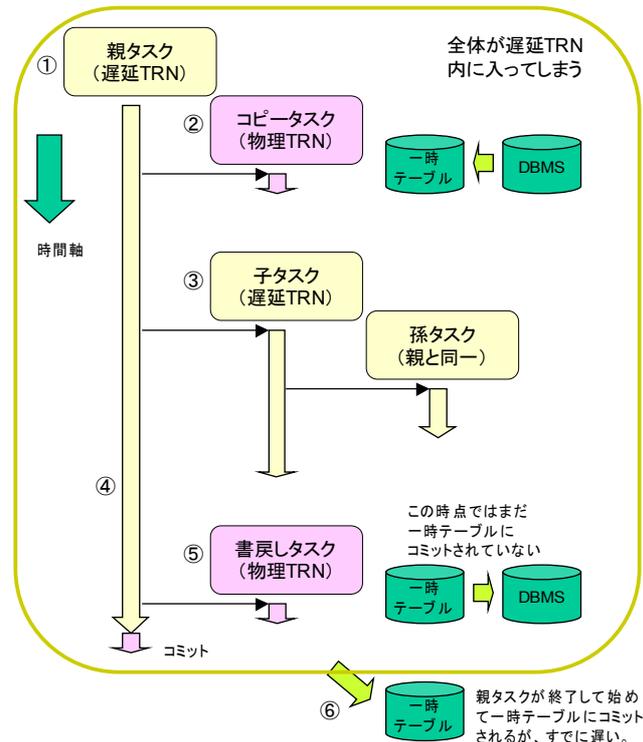
一時テーブルと遅延トランザクションの正しい設定



もし、ルートの子タスクのトランザクションが「なし」ではなく、「遅延」(レコードあるいはタスクレベル)に設定されていると、右図のような動作となり、データが正しく更新されません。これは、親タスクのトランザクションの設定が、タスクレベルの遅延トランザクションになっていた場合の例です。

- ① 親タスクで遅延トランザクションが始まります。
- ② コピータスクが呼び出されます。タスクレベルの物理トランザクションなので、タスクが終了した時点で一時テーブルにデータがコミットされます。
- ③ 子タスクと孫タスクで、ユーザがデータを入力します。その内容はトランザクションキャッシュに蓄積されます。
- ④ 子タスクが終了して親タスクに戻っても、まだ遅延トランザクションの中なので、トランザクションキャッシュのデータがコミットされません。
- ⑤ ここで書き戻しタスクを呼び出しても、一時テーブルの内容は更新されていません。従って、ユーザの入力内容はDBMSに反映されません。
- ⑥ 最後に、親タスクが終了するときに、トランザクションキャッシュの内容が一時テーブルにコミットされますが、すでに遅い。

一時テーブルと遅延トランザクションの誤った設定



4.6 物理トランザクションでの重複チェックのタイミング

Magic エンジンには重複レコードを自動的にチェックする機能が備わっていますが、これは物理トランザクションの中でだけ有効な機能です。遅延トランザクションでは重複チェックが自動で行われません。

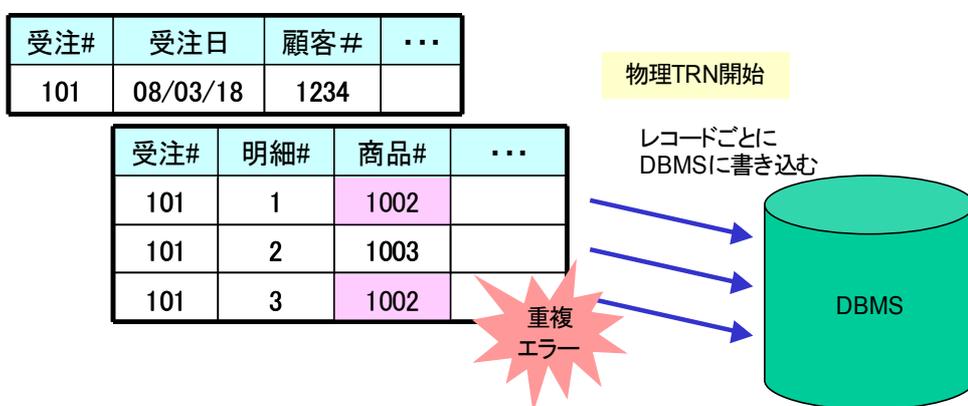
物理トランザクションから遅延トランザクションに移植する場合、この点に注意する必要があります。

本節と次節では、この点について説明します。

たとえば、明細レコードで、〈受注番号、商品番号〉の組み合わせが一意的なインデックスとして定義されていたとします。この場合、明細レコード中に同一の商品を選択することができません。

この状況で、下図のように受注のヘッダ・明細入力を行うと、物理トランザクションを使ったプログラムでは次のようになります。

物理トランザクションでのキー重複チェック



- ① 受注ヘッダレコードが開始されるタイミング(親タスクで、ユーザが入力を開始したとき)に、物理トランザクションが開始されます。
- ② ユーザが子タスクに移り、最初の受注明細行(商品#=1002)を入力してから、次の行に移動したとき、Magic はDBMS に明細レコードを書き込みます。
- ③ 次の受注明細行を入力するとき、商品#を入力したタイミングで、Magic エンジンが重複レコードをチェックします。上の例では、2 番目のレコードの商品#が 1003 なので、重複がなく、そのまま処理が進みます。
- ④ 次に、3 行目を入力するとき、商品#1002 を入力したタイミングで、Magic エンジンが重複チェックを行い、重複が見つかるので、エラーメッセージを出します。

このように、物理トランザクションの中では、重複キーを入力するとすぐに、重複が検出され、ユーザに警告されます。



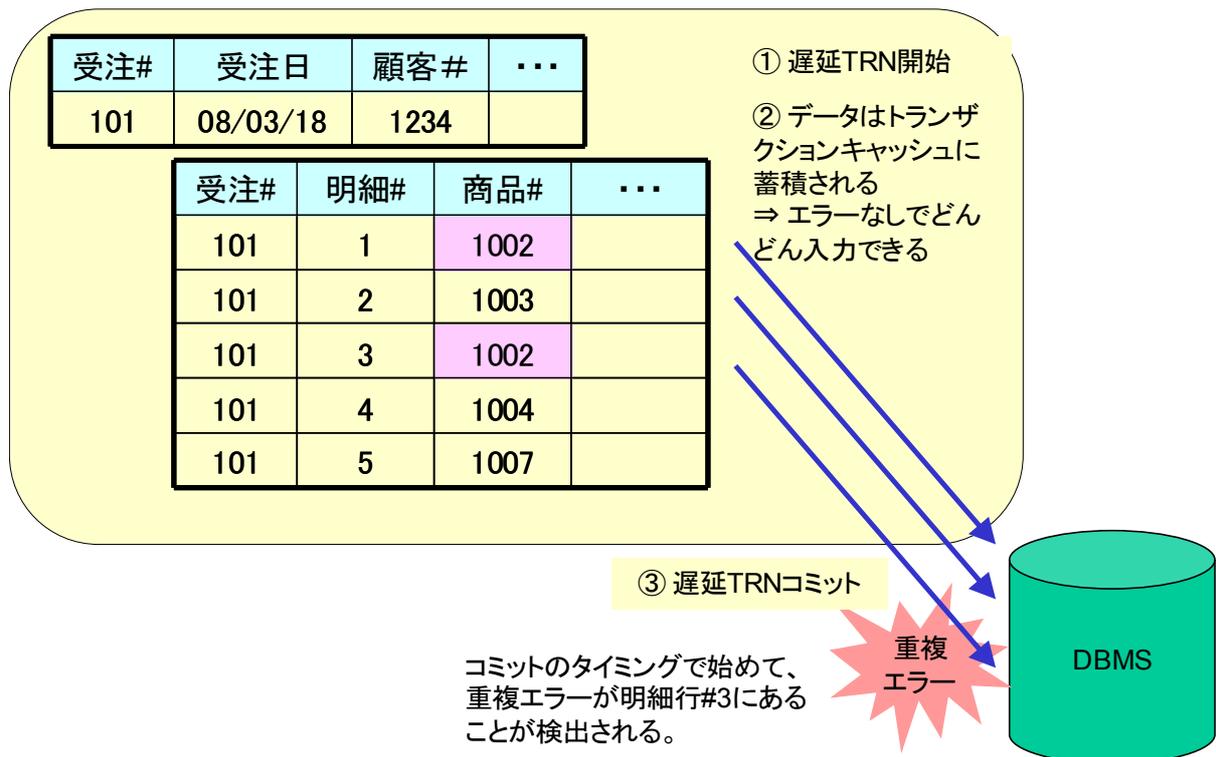
Magic エンジンの重複チェックは、DBMS へのレコードアクセスを伴うので、「データベース」テーブルの設定で無効にすることもできます。もし、重複チェックを行わないように設定されていた場合には、上の④において、ユーザが商品番号を入力した直後にはエラーチェックが行われません。しかし、レコード後処理の後、DBMS に書き込みを行う時点で、DBMS から重複エラーが報告されるので、いずれにしても、次のレコードに進む前に、重複エラーが検出されます。

4.7 遅延トランザクションでの重複チェックのタイミング

遅延トランザクション内では、レコードの重複チェックが自動的に行われません。また、遅延トランザクション中はトランザクションキャッシュ中にデータが蓄積されるため、レコード書き込みのタイミングでもDBMSのエラーが発生しません。遅延トランザクションがコミットされ、トランザクションキャッシュのデータがDBMSに書き込まれるタイミングで始めて、DBMSからキー重複エラーが検出されます。

前節と同じ例で、遅延トランザクションで行った場合の動作を見てみましょう。

遅延トランザクションでのキー重複チェック



- ① 受注ヘッダレコードが開始されるタイミングで、遅延トランザクションが開始されます。
- ② ユーザの入力は、ヘッダレコードも明細レコードもすべて、トランザクションキャッシュに蓄積されます。この間、キー重複のチェックはされません。従って、3行目で同一商品番号を入力しても、エラーにはならず、引き続き明細行を入力し続けることができます。
- ③ 最後に、親タスクに戻り、受注ヘッダレコードが終了して、遅延トランザクションがコミットされます。このとき、トランザクションキャッシュの内容がまとめてDBMSに書き込まれますが、明細行の3行目を書き込もうとした段階で、重複エラーがDBMSにより検出されます。

このような動作になっているため、遅延トランザクションを使った場合にもキー重複を行いたい場合には、自分でキーチェックのプログラムロジックを作りこむ必要があります。具体的には、Stat(0,'C'MODE)を条件として、データリンクを行い、リンクの戻り値がFalseであることを確認する、というようなロジックとなります。

5 トランザクションのネスト

遅延トランザクションはネストすることができます。すなわち、すでに遅延トランザクションが開始されている状態で、そのトランザクションを終了することなく、別の独立したトランザクションを開始・コミットすることができます。

本章では、遅延トランザクションのネストに関して、ネストの可能な組み合わせ、ネスト時の動作、ネストを使う上での注意事項などについて説明します。

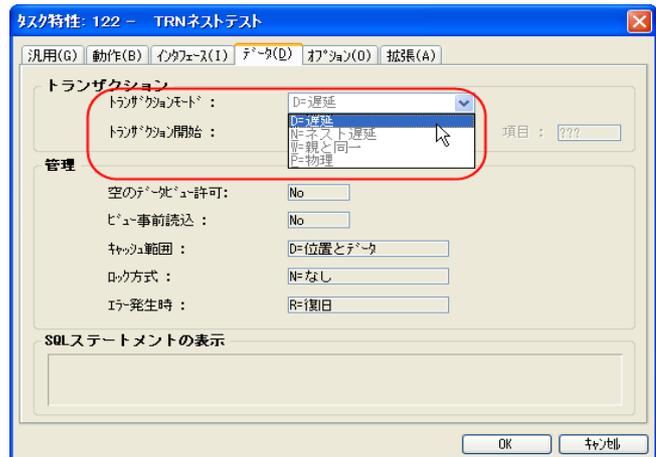
5.1 トランザクションのネストはいつ起こるか

2.5「トランザクションの設定」で説明したように、タスクのトランザクション設定は、タスク特性の「データ」タブで行います(下図参照)。

ここで見えるように、「トランザクションモード」の設定には、

- D=遅延
- N=ネスト遅延
- W=親と同一
- P=物理

の4つの選択肢があります。



トランザクションの設定はタスク単位で行うことができるので、すでに遅延トランザクションで実行しているタスクから、別のタスクを呼び出した場合に、新たなトランザクションが開始されることがあります。呼び出し元のトランザクションの状態と、呼び出し先のタスクのトランザクションモードの設定の組み合わせの結果をマトリクスで示したのが下図です。

呼び出し元のトランザクション状態 ↓	呼び出し先の TRN 設定			
	遅延	ネスト遅延	親と同一	物理
トランザクションなし	遅延	遅延	(同一)	物理
物理トランザクション	(エラー)	(エラー)	(同一)	(同一)
遅延トランザクション	(同一)	遅延(ネスト)	(同一)	物理(ネスト)

ここで、

- (エラー) というのは、Magic で許可されておらず、実行時にエラーが発生する組み合わせです。
- (同一) というのは、新しいトランザクションは発生せず、以前の状態がそのまま継続することを意味します。
- これ以外の組み合わせの場合には、マトリクスに書いてあるモードで、新しいトランザクションが発生します。

この中で、(ネスト) と注記されているモードがありますが、これは、ネストしたトランザクションが発生することを意味しています。すなわち、

- 遅延トランザクション → ネスト遅延トランザクション
- 遅延トランザクション → 物理トランザクション

というネストが可能です。

特に、ネスト遅延トランザクションは遅延トランザクションの一種ですので、別のタスクでさらにネストをさせることもできます。遅延トランザクションのネストのレベルには制限がありません。



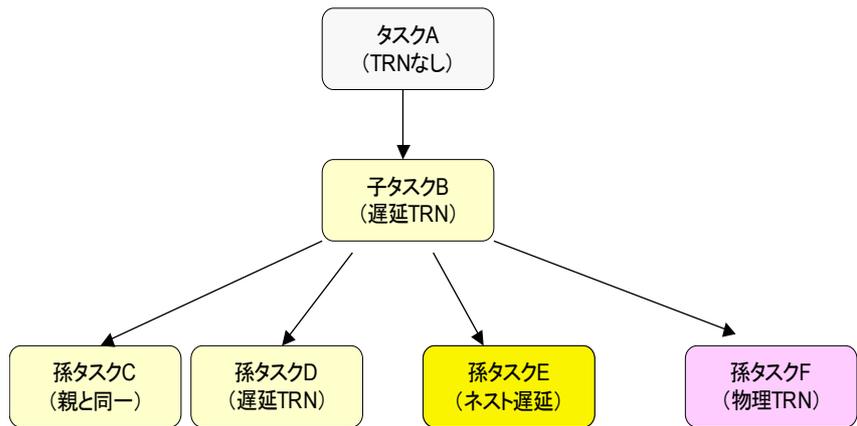
物理トランザクションのネストはできません。すでに物理トランザクションが開始されている状態で、「P=物理」のトランザクションモードのタスクを呼び出しても、新たなトランザクションは発生せず、そのまま同一の物理トランザクションの中で実行されます。

5.2 ネストトランザクションの動作例1

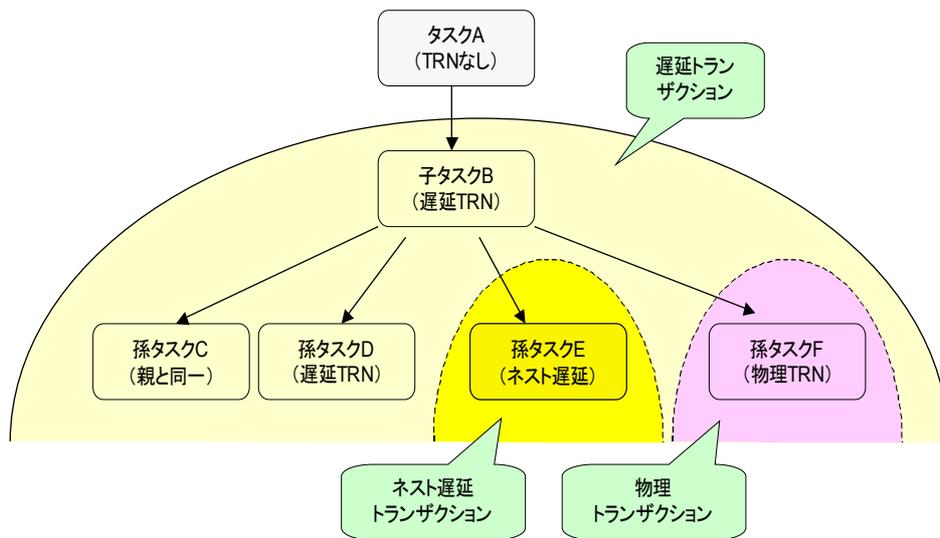
トランザクションのネストを具体的にみていきましょう。

例として、右図のようなタスク構造のプログラムがあったとします。それぞれのトランザクションモードの設定は、タスク名の下に「()」で注記されています。

ここでは、簡単のため、タスクレベルのトランザクションと仮定します。



このプログラムを、タスクAからタスクB、タスクC、... という順序で実行させていった場合、トランザクションの範囲がどのようになるかを図示したものが下図です。

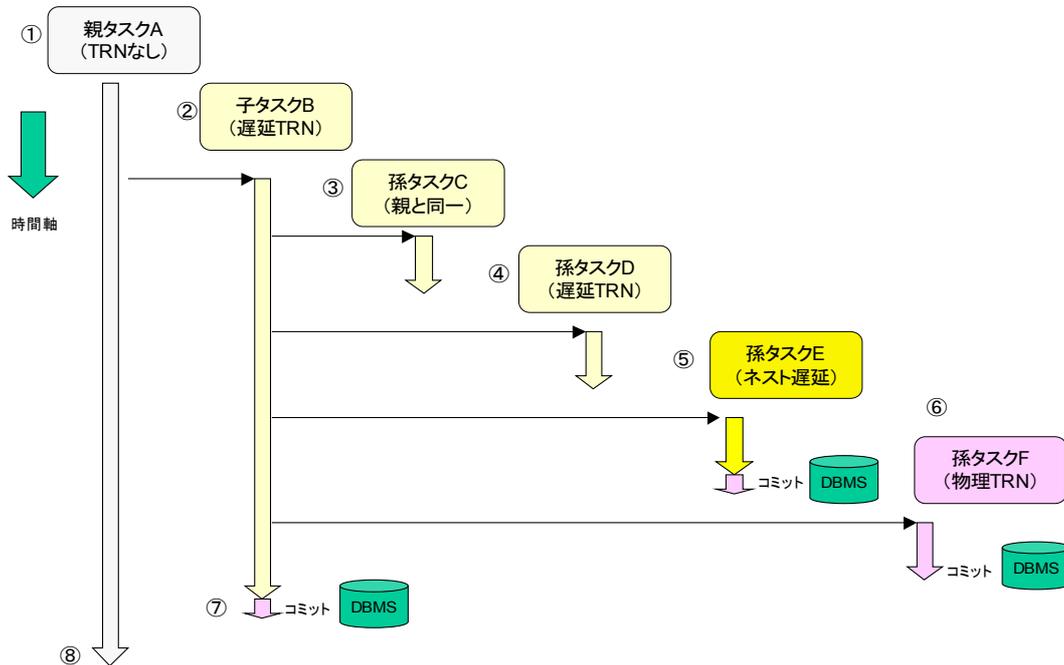


すなわち、

- タスクAはトランザクションなしで動作します。
- タスクB、C、Dは同一の遅延トランザクションの中で動作します。
- タスクEは、ネストされた遅延トランザクションとして動作します。この遅延トランザクションは、タスクB、C、Dの遅延トランザクションとは完全に独立した(分離された)トランザクションとして動作します。
- タスクFは、物理トランザクションとして動作します。この物理トランザクションも、他の遅延トランザクションとは完全に独立したトランザクションとして動作します。

5.3 ネストランザクションの動作例 2

前節の例で使ったタスクを実行したとき状況を、時間軸を縦軸にして図示したのが、下図です。



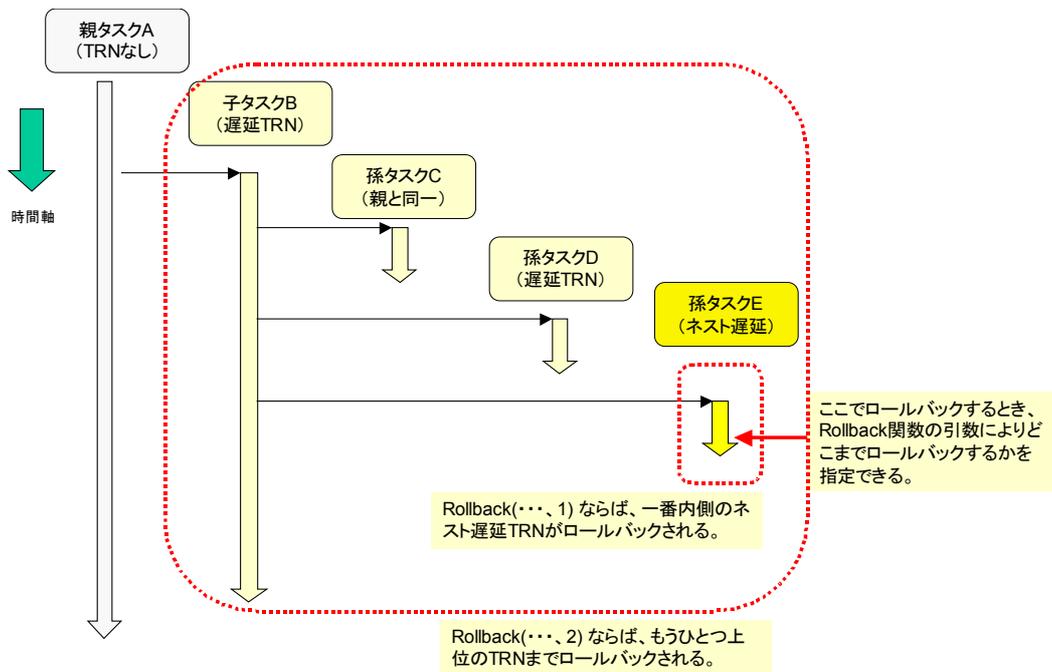
- ① 最初に、タスク A が起動されます。この時点では、トランザクションは開始されていません。
- ② 次に、タスク B が起動されます。ここで遅延トランザクションが開始されます。
- ③ タスク C は「親と同一」のトランザクションモードなので、新たなトランザクションは発生しません。タスク B の遅延トランザクションの中で動作します。
- ④ タスク D は「遅延」のトランザクションモードですが、すでにタスク B で遅延トランザクションが開始されているので、新たなトランザクションが発生しません。タスク B の遅延トランザクションの中で動作します。
- ⑤ タスク E は、「ネスト遅延」のトランザクションモードですので、新しい遅延トランザクション(ネストされた遅延トランザクション)が発生します。タスク E が終了する時点で、このネストされた遅延トランザクションがコミットされます。すなわち、このタスク E 内で行われたデータ変更が、DBMS に反映されます。
- ⑥ 次に、タスク F は「物理」のトランザクションモードですので、新しい物理トランザクション(遅延トランザクションの中にネストされた物理トランザクション)が発生します。すなわち、ここで DBMS に対して、トランザクション開始が通知されます。タスク F の中で行われたデータ操作は、そのつど DBMS に送られ、タスク F の終了時に、DBMS に対して、トランザクションのコミットが通知されます。
- ⑦ タスク B に戻り、タスク B が終了する時点で、遅延トランザクションがコミットされます。すなわち、タスク B、C、D の中で行われたデータ操作の内容が、すべて DBMS に反映されます。
- ⑧ タスク A に戻り、終了します。ここではトランザクションがないので、DBMS に対する操作は起こりません。

5.4 ネストしたトランザクションのロールバック

トランザクションをロールバックすると、トランザクション中のデータ操作内容がすべて取り消されますが、ネストされたトランザクションがある場合には、どこまでロールバックされるかが問題になります。

ここではまず、ネストされたトランザクション内でロールバックが起こった場合にどこまでロールバックが起こるかを説明します。

下図は、前節で使ったプログラム例の中で、ネストされた遅延トランザクションのタスクEの中で、Rollback 関数が実行されたときに、どこまでロールバックされるかを図示したものです。



2.6「ロールバック」で説明したように、Rollback 関数は、第2引数として、トランザクションのネストレベルを指定することができます。今のように、ネストされたトランザクションの中で Rollback 関数が実行される場合には、この引数の値により動作が変わります。

- ネストレベルが1の場合 (Rollback(..., 1) が実行された場合) には、一番内側のネストトランザクションだけがロールバックされます。すなわち、タスクEの中で行われたデータ操作だけがロールバックされ、リスタートされます。
- ネストレベル2の場合 (Rollback(..., 2) が実行された場合) には、もうひとつ上のレベルのトランザクションまでがロールバックされます。すなわち、この例ではタスクBまでさかのぼり、トランザクションキャッシュの内容が破棄され、リスタートします。
- Rollback 関数の第2引数には、0を指定することもできます。この場合、すべてのトランザクションがロールバックされます。この例では、一番外側のトランザクションは、タスクBで開始された遅延トランザクションなので、タスクBまでさかのぼってロールバックがされます。すなわち、この場合には Rollback(..., 2) と同じ結果となります。



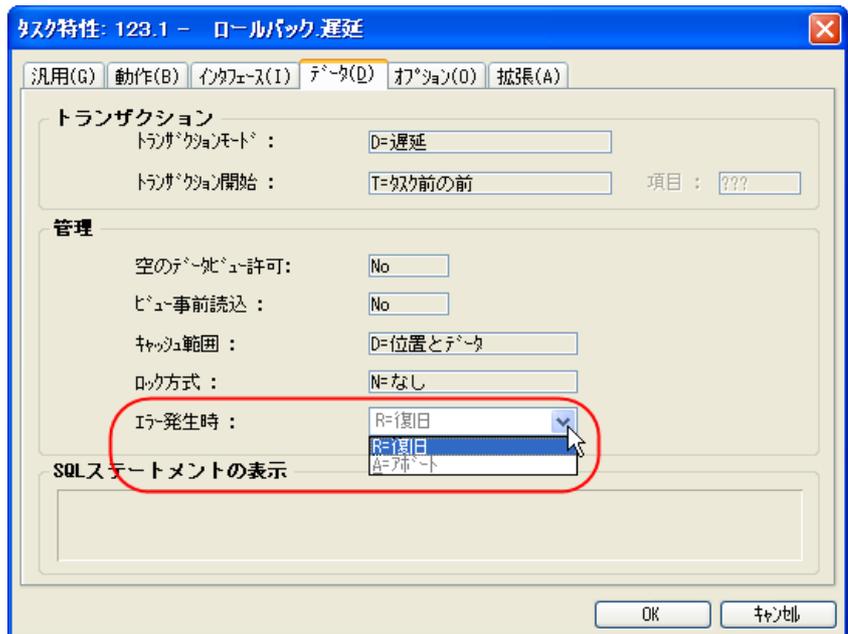
ロールバック後にリスタートするか否かは、タスク特性で設定できます(次節参照)。上の説明は、デフォルトの動作(リスタート)を前提にしたものです。

5.5 ロールバック後の動作

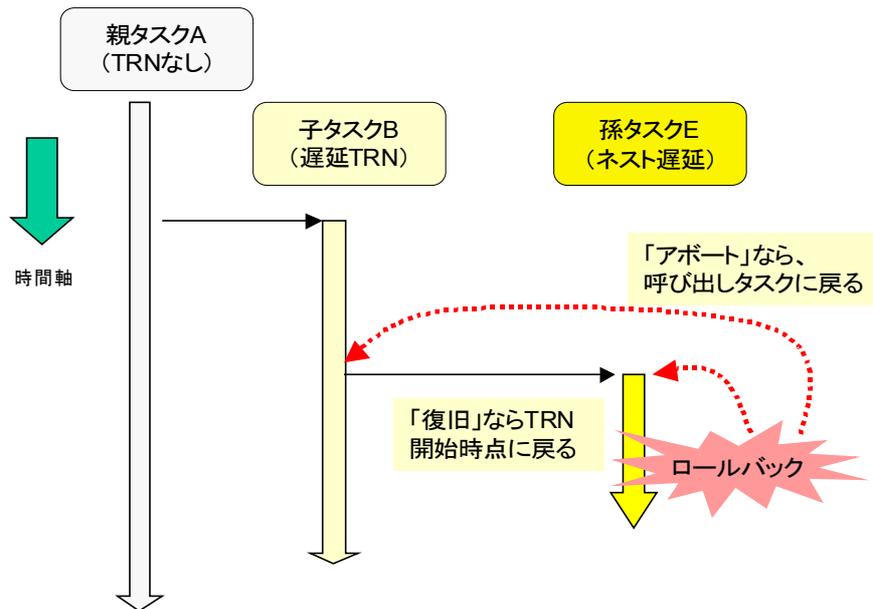
トランザクションがロールバックした場合、トランザクションキャッシュの内容がリセットされ、トランザクションが開始された時点にまで戻りますが、その後はどのように動作するでしょうか？

これについては、次の二つのオプションをタスク特性の「エラー発生時」パラメータで設定することが可能です。(右図)

- R=復旧: トランザクションを開始時点にまで戻って、リトライする。
- A=アボート: このタスクを終了する。



たとえば、下図のように、タスクB ⇒ タスクE とネストされた遅延トランザクションの中で、1レベルロールバックさせるために `Rollback(..., 1)` が実行された場合には、トランザクションキャッシュがリセットされた後、次のように動作します。

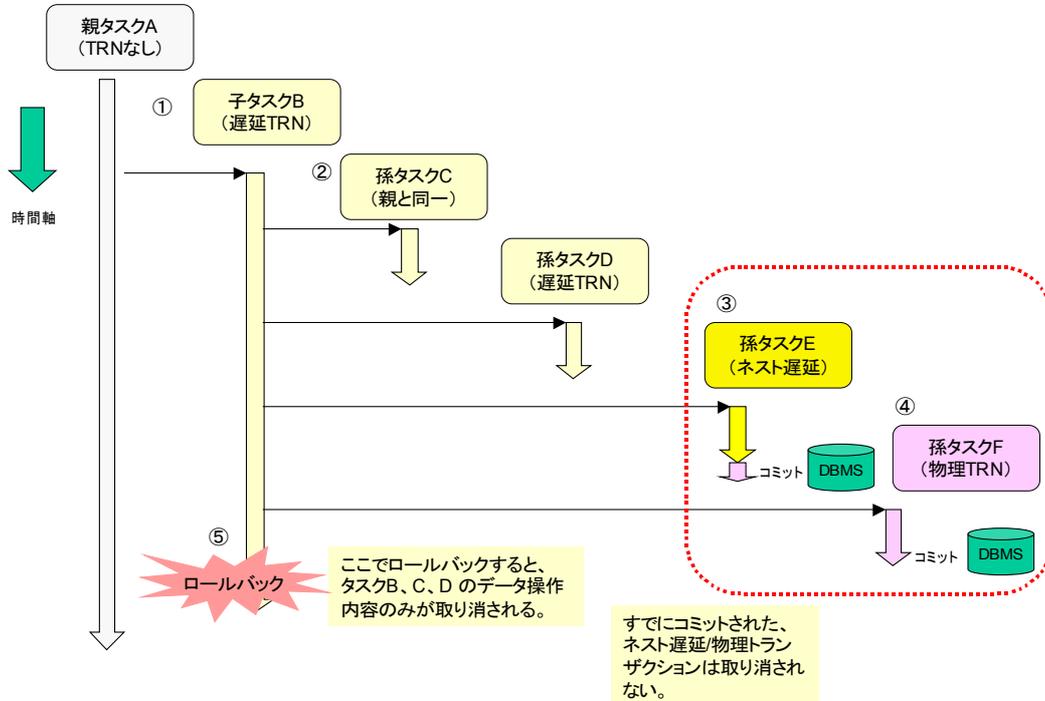


- タスクEの「エラー発生時」パラメータが「R=復旧」の場合には、トランザクション開始の時点(タスクEの最初)に戻ります。タスクEのトランザクション開始がレコードレベルであれば、レコードが再度フェッチされ、レコード前処理を行い、ユーザの入力待ちになります。トランザクション開始がタスクレベルであれば、タスクEが初期状態に戻り、タスク前処理⇒レコードフェッチ⇒レコード前処理を行い、ユーザ入力待ちの状態になります。
- タスクEの「エラー発生時」パラメータが「A=アボート」の場合には、タスクEが終了し、タスクBに戻ります。より正確には、タスクBで、タスクEを呼び出した「コール」コマンドの直後に進み、実行を再開します。

5.6 ネストしたトランザクションのコミット後のロールバック

ネストトランザクションのあるプログラムでは、ネストしたトランザクションがコミットされた後に、外側のトランザクションでロールバックが起こる、ということがありえます。このような場合には、すでにコミットされたネストトランザクションの内容は取り消されません。

たとえば、次のような流れが実行されたとします。(下図)



- ① タスクBで遅延トランザクションが開始される。
- ② タスクC、タスクDは、タスクBの遅延トランザクションの中で実行される。
- ③ 孫タスクEでネスト遅延トランザクションが開始され、コミットされる。
- ④ 孫タスクFで物理トランザクションが開始され、コミットされる。
- ⑤ その後、タスクBに戻ってから、ここでロールバックが起こる。

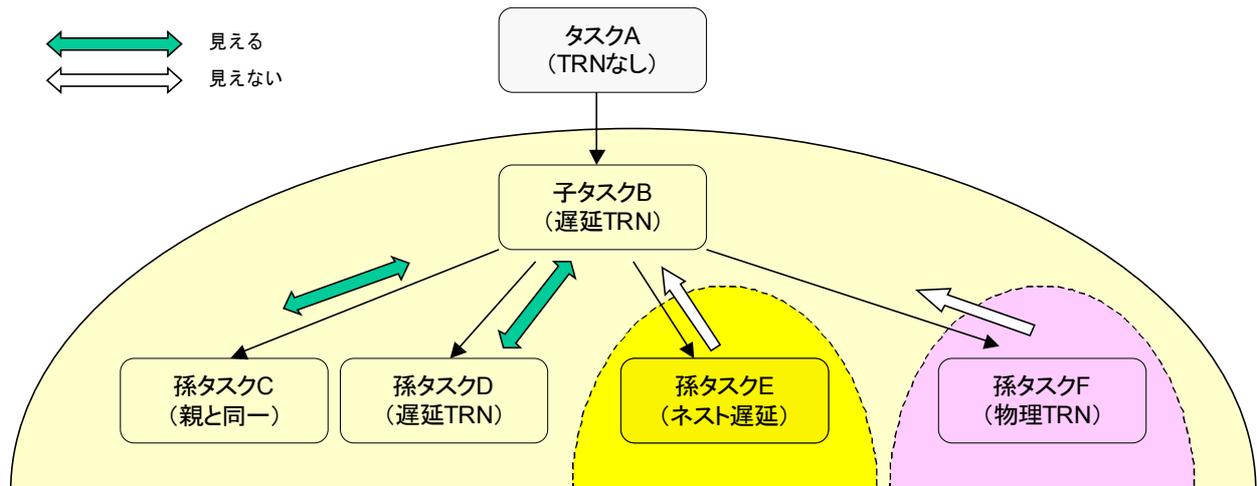
この場合には、タスクB、C、Dにおいて行われたデータ操作内容のみが⑤のロールバックによって取り消されます。③でタスクEで行われDBMSにコミットされたデータ操作、および、④でタスクFで行われDBMSにコミットされたデータ操作の内容については、すでに確定したもののなので、取り消されることはありません。

5.7 ネストトランザクションの分離

遅延トランザクションは、それぞれが完全に独立していて、コミット前のデータが他のトランザクションから見えることはありません。すなわち、いわゆる「ダーティリード」が起こりません。このため、遅延トランザクションは「コミット済み読み取り(READ COMMITTED)」の分離レベルを実現しています。

このことは、ネストトランザクションについても同じです。すなわち、ネストトランザクションはそれぞれが独立したトランザクションであるために、お互いに分離されていて、一方でのデータの変更内容は、コミットされるまで、他のトランザクションからは見ることはできません。

例として、下図のようなネストトランザクション設定のプログラムがあるとします。



ここでは、

- タスクB、C、D は同一の遅延トランザクション
- タスクE は、ネスト遅延トランザクション
- タスクF は、(遅延トランザクションにネストした)物理トランザクション

の中でそれぞれ実行しています。

この場合には、図で示したように、同一トランザクション内の変更は見えますが、他のトランザクションの中の未コミットデータ内容は見ることはできません。

実際に、実行させてみたのが次ページの図です。



遅延

商品番号	商品名	商品タイプ	単価	在庫数
1002	フード	D	10,200	40
1003	フォックス テリア	D	4,080	50
1004	カリヤ	B	3,060	50
1005	ルカイ	F	21,420	50
1006	チワ	D	2,040	50
1007	ケッペ	F	816	50
1008	ルマージュ	F	8,160	50
1009	パンショウ	B	816	50
1010	ルスター	R	102	50

親と同一 遅延 ネスト遅延 物理 終了

ここでデータを
50→40に修正

親と同一

商品番号	商品名	商品タイプ	単価	在庫数
1002	フード	D	10,200	40
1003	フォックス テリア	D	4,080	50

「親と同一」あるいは
「遅延」→「遅延」では、同一
遅延 TRNに入る
⇒ 修正データが見える
(40になっている)

遅延(孫)

商品番号	商品名	商品タイプ	単価	在庫数
1002	フード	D	10,200	40
1003	フォックス テリア	D	4,080	50

ネスト遅延

商品番号	商品名	商品タイプ	単価	在庫数
1002	フード	D	10,200	50
1003	フォックス テリア	D	4,080	50

「ネスト遅延」あるいは
「物理」では、別 TRN で実
行する
⇒ 修正データが見えない
(50のまま)

物理

商品番号	商品名	商品タイプ	単価	在庫数
1002	フード	D	10,200	50
1003	フォックス テリア	D	4,080	50

5.8 ネストしたトランザクションの設定

前節のように、ネストしたトランザクションでは、他のトランザクション中の未コミットデータを見ることができません。このことから、実際にプログラムを作る上で、次のような注意が必要になります。

5.8.1 例1: 明細更新バッチタスク

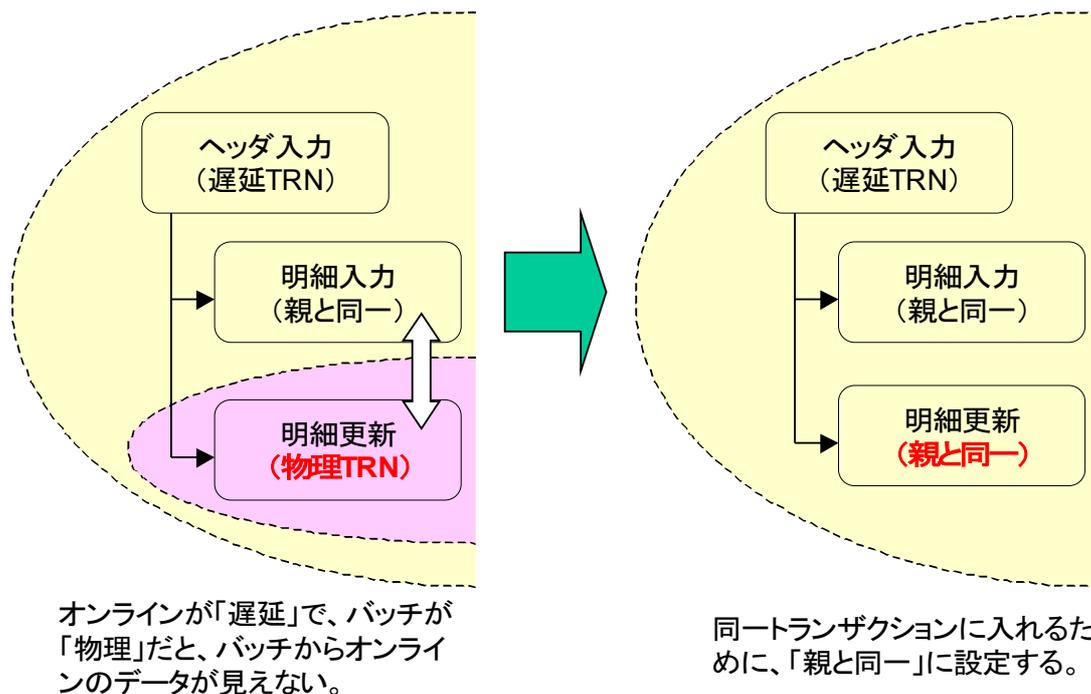
ヘッダ・明細の入力プログラムにおいて、「明細更新」バッチサブタスクで、行番号のつけなおし、在庫データの反映などを行う場合を考えてみます。

このとき、「明細更新」のトランザクション設定が「物理」(バッチタスクのデフォルト設定)になっていると、ヘッダ・明細入力のオンラインタスクは遅延トランザクションで動作するのに対し、明細更新バッチタスクはネストされた物理トランザクションとして、つまり別トランザクションとして動作するようになります。

そうすると、明細入力オンラインタスクでユーザが入力したデータが、明細更新バッチタスクでは見えません。そのため、このバッチタスクによって、意図したようなデータ更新がされなくなってしまいます。

このようになっていると、集計データなどが更新されていなかったり、あるはずのない0のデータが登録されていたりする結果となります。

このような場合には、バッチタスクのトランザクション設定も「親と同一」とします。こうすると、バッチタスクはオンラインタスクと同一の遅延トランザクションの中で実行されるようになり、ユーザの入力データがバッチタスクでも見えるようになり、正しく更新処理を行うことができるようになります。



5.8.2 例2: 新受注番号発番

遅延トランザクションのタスクから呼び出されるバッチタスクは、上の例で見たように、普通は同一のトランザクション内で動作させることが必要ですので、「親と同一」にすることが多いと思います。

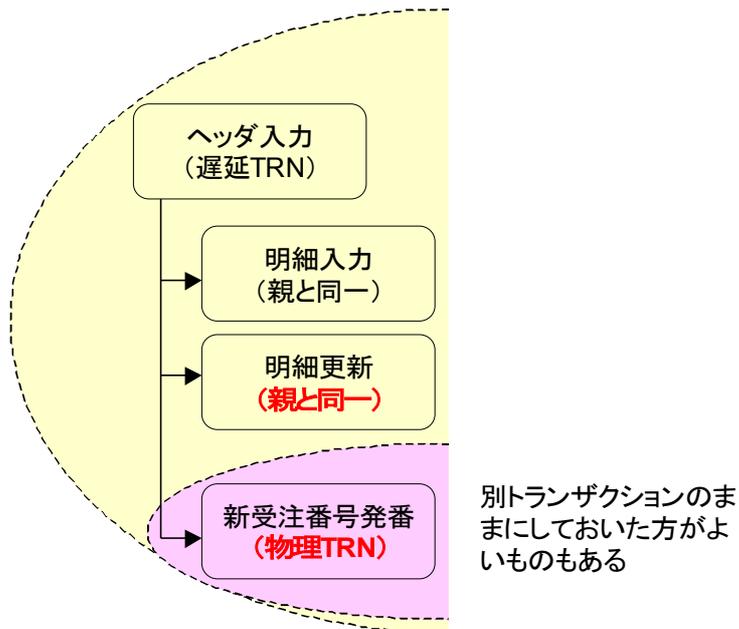
ただし、意図的にネストした物理トランザクションにしておいた方がよいこともあります。

たとえば、新規受注登録の場合に、新しい受注番号を発番するバッチタスクがこれに該当します。よく見られるやりかたとして、アプリケーション全体の管理テーブルなどに最終受注番号を記録しておき、発番する場合には、これに +1 してやる方法があります。

このようなバッチタスクでは、次のような要求事項があります：

- 多くのユーザが同時に利用していることを考えると、間違っ同一受注番号を発行しないように、管理テーブルは厳密に排他が行われている必要があります。
- 管理テーブル(最終受注番号を管理しているレコード)はアプリケーション全体にひとつだけなので、これに対するロック時間は、必要最小限に抑えておく必要があります。
- 一方、このバッチタスクでは、ユーザが入力したデータを参照する必要がありませんから、遅延トランザクションでの未コミットデータが見える必要はありません。

従って、このバッチタスクは、遅延トランザクションの中にネストする物理トランザクションとして実行させるのがベストです。すなわち、トランザクションモードは「物理」とします。



6 外部キーと同期パラメータ

SQL DBMS では、依存関係のあるテーブル間に、外部キー制約を定義することが可能です。外部キー制約は、データの整合性を確保するため、不正な参照関係をチェックし、万が一不正なデータが挿入されたり、不正な状態にデータが更新・削除されたりした場合には、例外を発生させて、不正挿入・更新・削除を防止する機能です。

外部キー制約は、ヘッダ・明細型の階層構造を持つテーブルにも設定する場合があります。この場合には、INSERT 文によるレコードの挿入順序に注意する必要があります。

本章では、最初に、外部キーの基本概念と、Magic での設定方法について説明します。続いて、ヘッダ・明細型の親子タスクで正しい順序で INSERT 文を発行するための方法について説明します。

6.1 外部キー制約とは

外部キー(Foreign Key)とは、リレーショナルデータベースの概念のひとつで、テーブル間のデータの依存性を定義するものです。特に、あるテーブルの一意キーの値を参照しているカラムにおいて、参照先に当該レコードが存在していることを保障するものです。

このようなデータの依存性は、小さなアプリケーションでも多く見られます。たとえば、下図はペットショップデモでのデータ依存性(の一部)です。

たとえば、次のテーブルのカラムは、レコードを一意に識別する一意キーです。

- 顧客マスタにおける顧客#
- 商品マスタにおける商品#
- 受注テーブルにおける受注#

そして、これらのカラムのデータを参照している、別テーブルのカラムがいくつかあります。

- 受注テーブルにおける顧客# (顧客マスタの顧客#を参照している)
- 受注明細テーブルにおける受注# (受注テーブルの受注#を参照している)
- 受注明細テーブルにおける商品# (商品マスタの商品#を参照している)

このようなカラムを「外部キー」と呼びます。

外部キー制約というのは、外部キーが参照しているテーブルで、その値のレコードが存在していなければならない、という制約です。

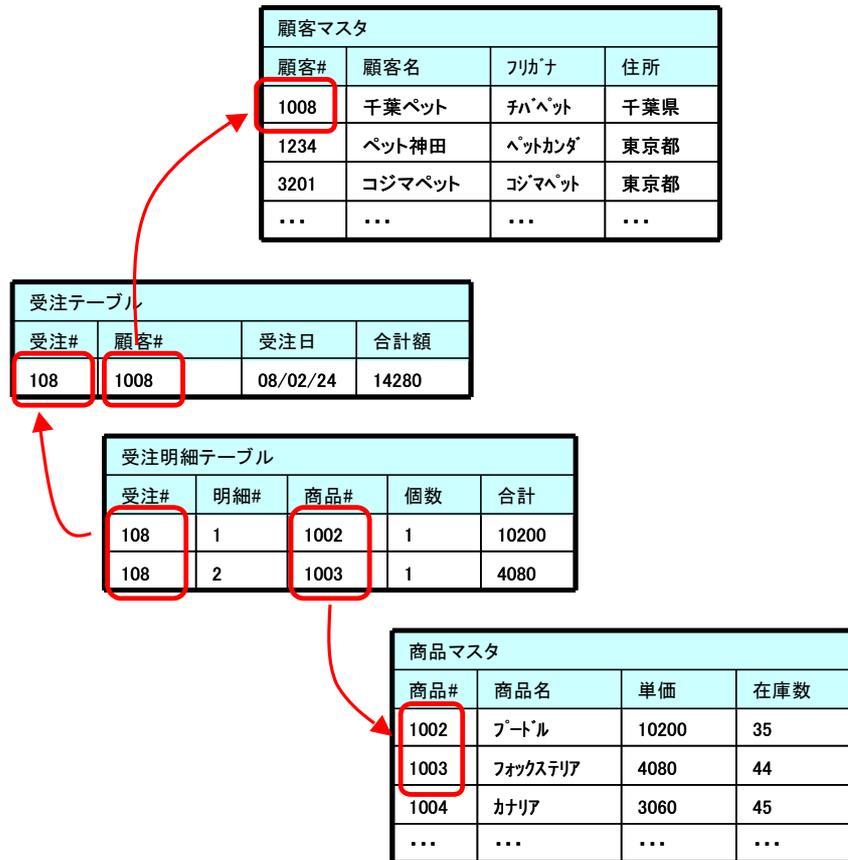
たとえば、上図の例では受注テーブルで顧客#1008を参照していますが、この場合、顧客1008のレコードが、顧客マスタに存在していなければなりません。もし1008というレコードが存在していなければ、この受注レコードは、存在していない(登録されていない)顧客を参照していることになり、これは不正なデータと考えられます。

このような不正なデータが万一発生したとすると、それは次のような原因が考えられます。

- オペレータが、顧客マスタに顧客登録を行っていない。
- プログラムのエラー(バグ)が潜在している。
- 間違って(以前は存在していたのに)顧客1008のレコードを削除してしまった。

そのほかにも可能性があるかもしれませんが、いずれにしても、そのまま放っておくことはできず、なんらかの対策が必要になる事態です。

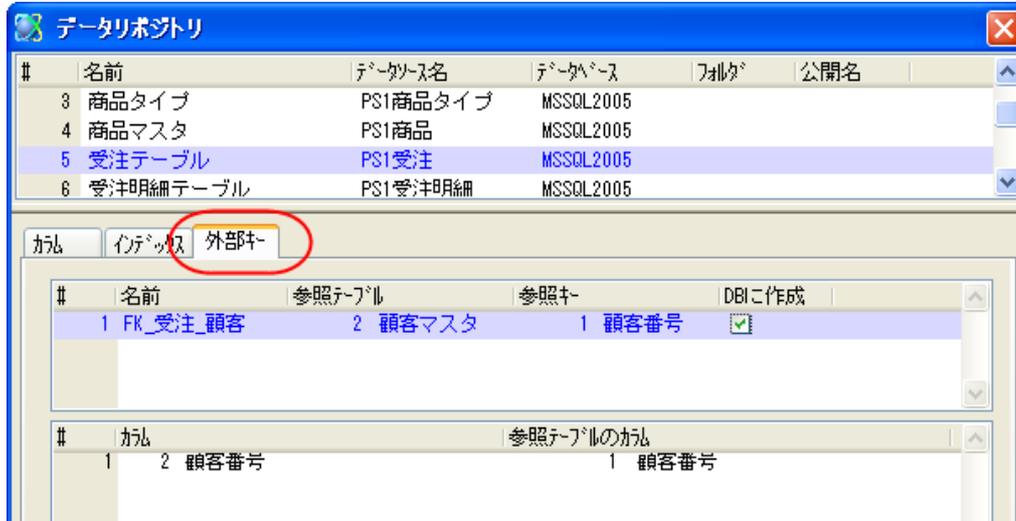
DBMS 管理者がこのようなデータの依存性の関係を「制約」として定義しておけば、DBMS が常に制約を監視するようになるので、万一不正データが発生しそうになったら、エラーを発生させ、不正な状態にならないように保護できるようになります。



6.2 外部キー制約の定義

Magic Studio において、外部キー制約を定義するには、データリポジトリの「外部キー」タブで行います。

たとえば、受注テーブルにおいて、顧客番号カラムから、顧客マスタの顧客番号キーを参照している場合には、下図のように定義します。



1. データリポジトリで、「受注テーブル」のエントリにカーソルを合わせます。
2. 「外部キー」タブを選びます。
3. F4 で1行作成、適当な名前(上図では「FK_受注_顧客」)をつけます。この名前は、DBMS 上に作成されるオブジェクトの名前になるので、データベース上で一意になるような名前にしてください。
4. 参照テーブルとしてテーブル2番「顧客マスタ」、参照キーとして、キー1番の「顧客番号」を設定します。
5. 「DBに作成」の欄には、チェックを入れてください。これにより、テーブル作成時に、この外部キー制約を定義するSQL文が生成されるようになります。
6. 下段の「カラム」欄には、受注テーブルのカラム2「顧客番号」を設定します。

これにより、DBMS 上に、「受注テーブル」の「顧客番号」→「顧客マスタ」の「顧客番号」という参照関係を制約とする、外部キー制約が定義されるようになります。

具体的には、Magic がテーブルを作成する段階で、次のような SQL が発行されて、この制約が DBMS 上に定義されます。

```
ALTER TABLE MAGIC..PS1 受注
ADD CONSTRAINT FK_受注_顧客
FOREIGN KEY (顧客番号)
REFERENCES MAGIC..PS1 顧客 (顧客番号)
```



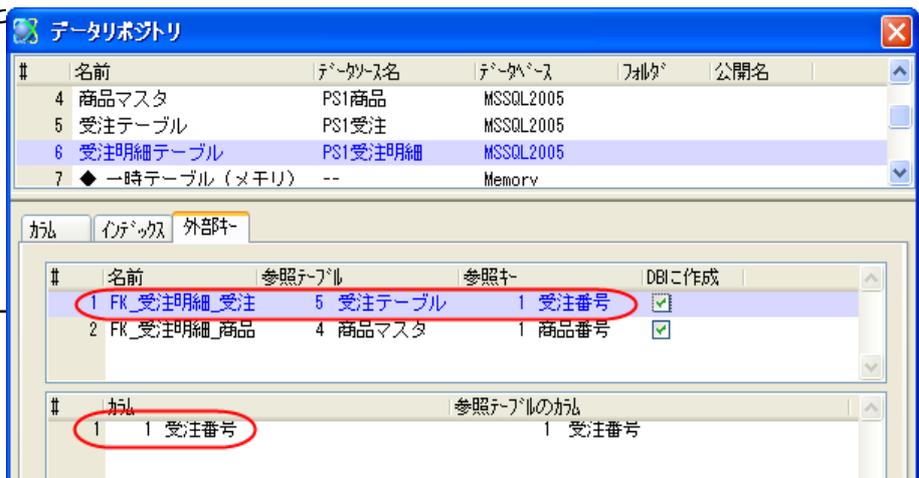
すでにデータが存在するテーブルに対して、あとから外部キー制約を定義すると、制約に違反するレコードが DBMS により削除されてしまいますので、注意してください。

6.3 ヘッダ・明細レコード間の外部キー制約

ヘッダ・明細型の階層構造を持ったレコード間にも、外部キー制約をかけることがあります。

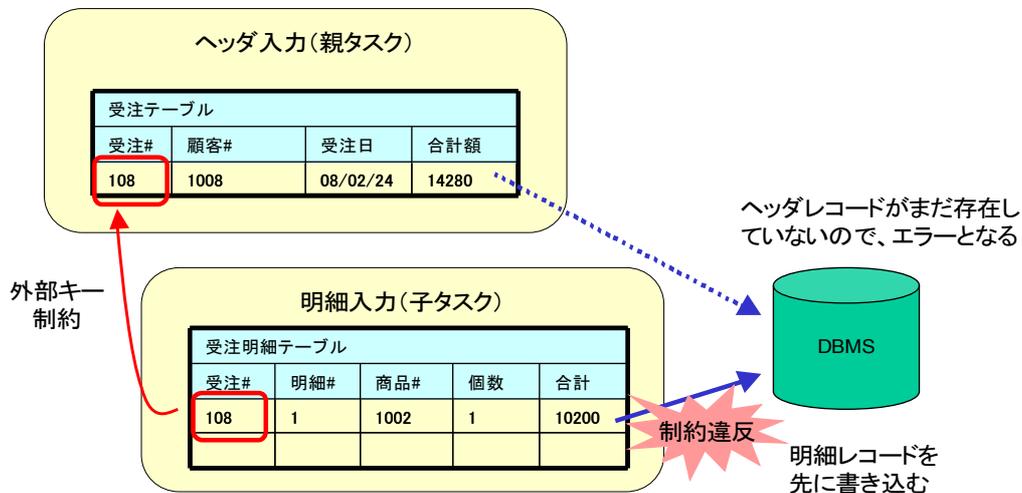
たとえば、先の例の受注テーブルと受注明細テーブル間には、「受注番号」での外部キー制約があります。すなわち、受注明細行で参照している受注番号(たとえば 108)は、受注テーブル中にも存在していなければなりません。

この参照キー制約は、受注明細テーブルの外部キーとして定義されます。(右図)



このように定義されているときに、Magic の親子タスクでレコードを新規登録すると、次のような問題が起こります。

1. 最初にユーザがヘッダデータを入力します。この時点ではまだDBMSにヘッダレコードがINSERTされません。
2. 子タスクに移って、明細レコードの1行目を入力し、2行目に移ります。このタイミングで、DBMSに明細レコードがINSERT文で書き込まれます。
3. しかし、明細レコード108を登録しようとしても、対応するヘッダレコードがまだDBMSにINSERTされていないので、外部キー制約に引っかかります。この結果、制約違反のエラーが発生してしまいます。



この問題を回避する方法を、次に説明します。

6.4 遅延トランザクションと同期パラメータ

遅延トランザクションを使った場合には、「同期」パラメータを Yes に設定することにより、外部キー制約の問題を解決できます。この方法は、遅延トランザクションの特性をうまく利用した方法で、一時テーブルを使わないでも、パラメータひとつで設定できる便利な方法です。

説明のために、受入力プログラムを一部変更し、サブフォームは使わず、「受注明細」ボタンを押すと子タスクが呼び出される、というようにします。

フォームに「明細」ボタンを貼り付け、そのイベントハンドラとして、サブタスクを呼び出すコールコマンドを入れます。

区分(C)	全体(A)	詳細
タスクID	1	受注明
パラメータ	1	
フォーム	0	
ロック	No	0
同期	No	0
リセット	Yes	
リ方向	No	
条件		

データビュー	ロジック	フォーム
23 アクション	E=式	13 Rollback ('FALSE'LOC
24 イベント	u_実行E	コマンド PB
25 コール	S=サブタスク	1 受注明細

「同期」パラメータは、このコールコマンドの特性のひとつとして定義されます。デフォルトでは No となっていますが、このパラメータが No の場合と Yes の場合とで、動作の違いを見ていきましょう。

6.5 同期 = No の場合の動作

コールコマンドの「同期」特性は、デフォルトでは No です。この場合には、「6.3 ヘッダ・明細レコード間の外部キー制約」で説明したように、DBMS への INSERT などのデータ操作 SQL 文は、すべて Magic エンジンの実行順序の通りに行われます。

この状態で実際に受注データを新規入力してみると、「確定」ボタンを押した時点(遅延トランザクションがコミットされる時点)で、右図のような制約違反エラーが発生します。

rc-ps-01

ファイル(E) 編集(E) 受注/物理トランザクション 受注/遅延トランザクション その他 オペレーション(O)

OM_受注

受注番号 00000117
受注日 2008/03/24

顧客情報
顧客番号 1008 千葉ペットショップ
住所 千葉県千葉市高柳 1234-1
割引率 9.00 条件 30日後支払い

DBエラー

説明:
MAGIC00 : INSERT ステートメントは FOREIGN KEY 制約 "FK_受注明細_受注番号" と競合しています。競合が発生したのは、データベース "MAGIC"、テーブル "dbo.PS1受注", column "受注番号" です。

千葉ペットショップは12年来のお得意様です。対応には十分に気を付けて下さい。

明細合計額 14,280
受注割引額 1,285
消費税額 650
受注合計額 13,645

修正 照会 登録 確定 取消
受注検索 印刷 削除 終了

制約違反. データソース: PS1受注明細 登録

データベースゲートウェイのログを見てみると、最初に次のような INSERT 文が発行され、そこでエラーが起きていることがわかります。

```
INSERT INTO MAGIC..PS1 受注明細
```

```
(受注番号, 受注明細番号, 商品番号, 商品タイプ, 数量, 単価, 合計)
```

```
VALUES ( 113 1, 1002, 'D', 1, 10200, 10200)
```

この SQL 文の問題は、受注番号 113 の受注レコードがまだ INSERT されていないので、外部キー制約のかかっている受注明細レコードで、受注番号 113 のレコードを INSERT しようとしていることです。

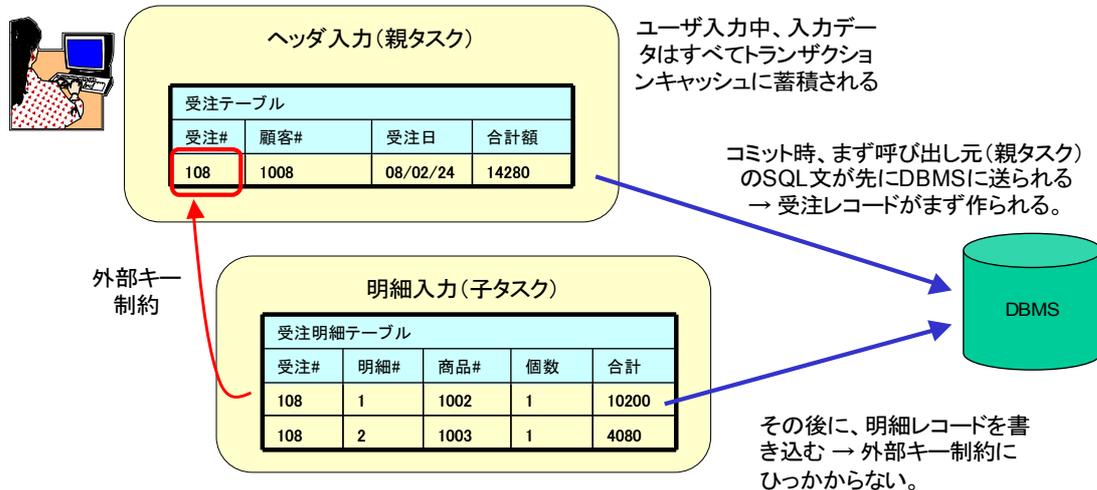


正確には、ゲートウェイのログはこれとは違う形の SQL 文が記録されていますが、ここでは説明のために、わかりやすく簡単化しています。

6.6 同期=Yesの場合の動作

「同期」パラメータの値が Yes になっていた場合には、呼び出し元のタスク(親タスク)で行われたデータ操作の SQL 文が先に DBMS に送られ、その後、呼び出し元のタスク(上の場合にはサブタスク 受注明細))でのデータ操作の SQL 文が DBMS に送られます。

同期=Yesのときの動作



この場合、発行される SQL 文は次のようなものになります。これを見てわかるように、受注テーブル(PS1 受注)に対する INSERT 文が先に発行され、その後に受注明細テーブル(PS1 受注明細)に対する INSERT 文が発行されているので、外部キー制約に違反することなく、処理が成功しています。

```
INSERT INTO MAGIC..PS1 受注
```

```
(受注番号, 顧客番号, 受注日, 最終明細番号, 明細合計額, 受注割引額, 消費税額, 受注合計額)  
VALUES (115,1008,'2008-03-24 00:00:00:000',2,14280,1285,650,13645)
```

```
INSERT INTO MAGIC..PS1 受注明細
```

```
(受注番号, 受注明細番号, 商品番号, 商品タイプ, 数量, 単価, 合計)  
VALUES (115,1,1002,'D',1,10200,10200)
```

```
UPDATE MAGIC..PS1 商品
```

```
SET 在庫数 = ISNULL(在庫数, 0) + (-1)  
WHERE 商品番号 = 1002
```

```
INSERT INTO MAGIC..PS1 受注明細
```

```
(受注番号, 受注明細番号, 商品番号, 商品タイプ, 数量, 単価, 合計)  
VALUES (115,2,1003,'D',1,4080,4080)
```

```
UPDATE MAGIC..PS1 商品
```

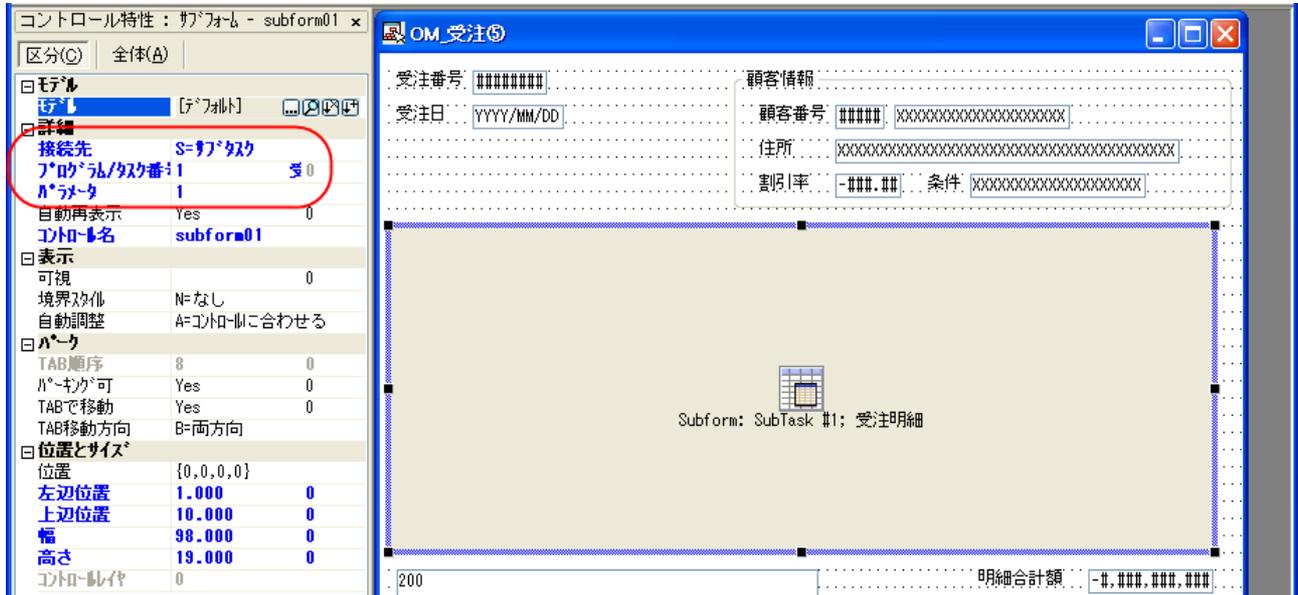
```
SET 在庫数 = ISNULL(在庫数, 0) + (-1)  
WHERE 商品番号 = 1003
```

```
UPDATE MAGIC..PS1 顧客
```

```
SET 受注累計額 = ISNULL(受注累計額, 0) + 1, 取引回数 = ISNULL(取引回数, 0) + 13645  
WHERE 顧客番号 = 1008
```

6.7 サブフォームの場合

ヘッダ・明細型のデータ構造のある場合には、サブフォームを使った親子タスクを使うのがより一般的です。この場合には、フォームエディタ上で、サブフォームの特性として、タスク/プログラム番号とパラメータとを指定します。



この場合には、「コール」コマンドを使わないので、コール特性としての「同期」パラメータを設定することができませんが、同期 = Yes と設定されたのと同じ動作をします。このため、とくに設定を行わなくとも、外部キー制約に違反することなく、ヘッダ・明細データの登録を行うことができます。

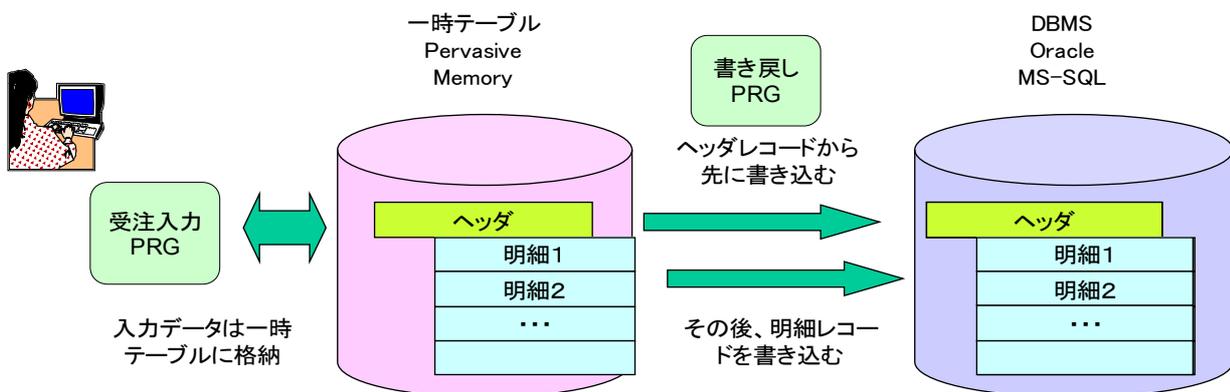
6.8 物理ランザクションの場合

以上は、遅延ランザクションを使った場合の話でしたが、本節では、比較のために、物理ランザクションを使っているプログラムで、この外部キー制約に違反しないようにする方法について説明します。

物理ランザクションでは、「同期」パラメータを使えないので、一時テーブルを使う必要があります。そして、以下に説明するように、登録・削除時に一時テーブル中のレコードを正しい順序で挿入・削除するよう、プログラムを設計します。

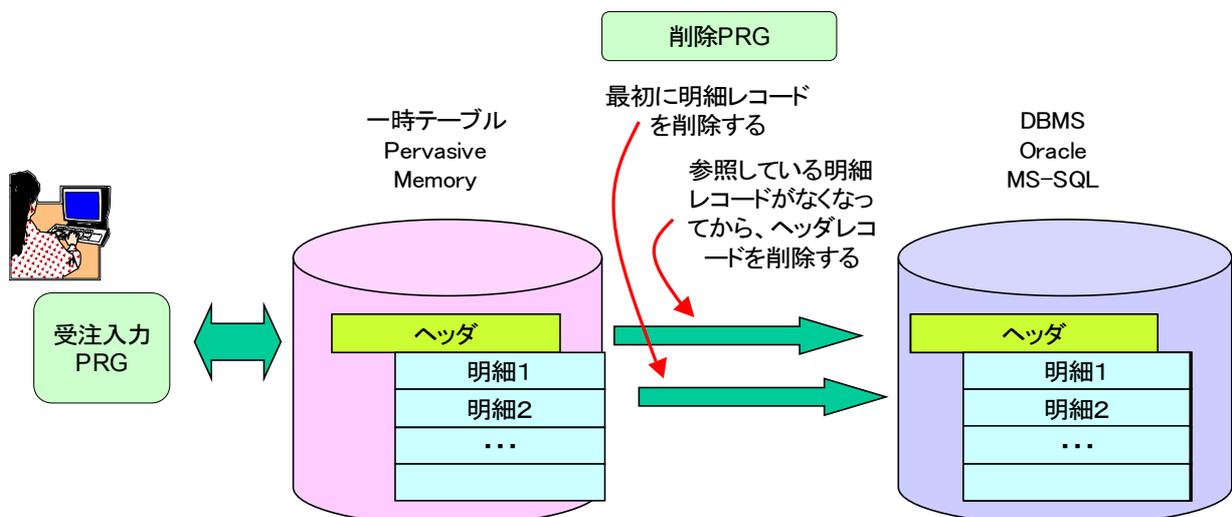
6.8.1 レコード登録時

新規レコード登録時、ユーザの入力はすべて一時テーブルに格納しておきます。確定時には、一時テーブルのレコードをDBMSに書き戻しますが、このときに、書き戻しバッチタスクの実行順序をよく考えて、ヘッダレコードを先に書き込み、そのあとに明細レコードを書き込むようにします。



6.8.2 レコード削除時

一方、外部キー制約は、レコードの登録時だけでなく、レコードの削除時にもチェックされます。このため、ヘッダ・明細レコードを削除する場合には、レコード作成の場合とは逆に、最初に明細レコードをすべて削除し、その後にヘッダレコードを削除する必要があります。これを逆にして、ヘッダレコードを先に削除してしまうと、明細レコードでまだ参照しているのにヘッダレコードを削除してしまうことになり、外部キー制約にひっかかり、エラーとなってしまいます。



7 トランザクションキャッシュのログ

Magic のプログラムをデバッグする際、Magic が発行する SQL 文を監視することにより、プログラムの挙動を検査して、意図した通りにデータアクセスが行われているかを確認することが有効です。

物理トランザクションを使っている場合には、ゲートウェイのログを有効にして、Magic のゲートウェイが DBMS に対して発行する SQL 文を監視することができます。あるいは、DBMS ベンダーから提供される、各 DBMS 専用のトレースユーティリティなどを使って、Magic が発行する SQL 文を確認できます。

しかし、Magic エンジンが遅延トランザクションで動作している場合には、すべてのデータがいったんトランザクションキャッシュに蓄積され、コミットのタイミングになって始めて、一度にすべての SQL 文が DBMS に対して発行されます。このため、デバッグのためにプログラムの処理を進めながら、リアルタイムで発行される SQL 文を監視することができません。

この問題を解決するために、Magic のデバッガでは、トランザクションキャッシュに対するデータ操作の内容を、リアルタイムに、アクティビティモニタに表示させることができるようになっています。

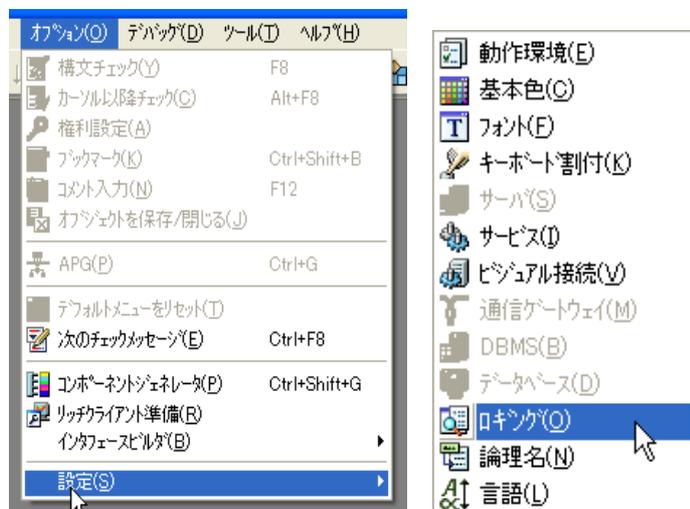
7.1 トランザクションキャッシュのログの表示設定

アクティビティモニタを開く: Studio でアクティビティモニタを開くには、メニュー 表示(V)⇒アクティビティモニタ を選択します。

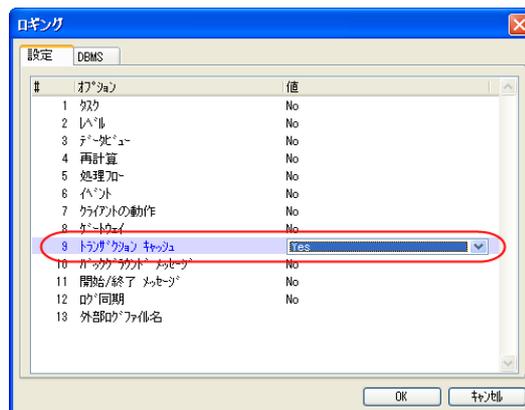


トランザクションキャッシュのログ表示: トランザクションキャッシュに対するデータ操作の内容を、リアルタイムに、アクティビティモニタに表示させるには、

- ① メニュー オプション(O) ⇒ 設定(S) ⇒ ログिंग を選択して、ログングダイアログを開きます。



- ② 「トランザクションキャッシュ」を Yes に設定します。



7.2 トランザクションキャッシュのログの監視

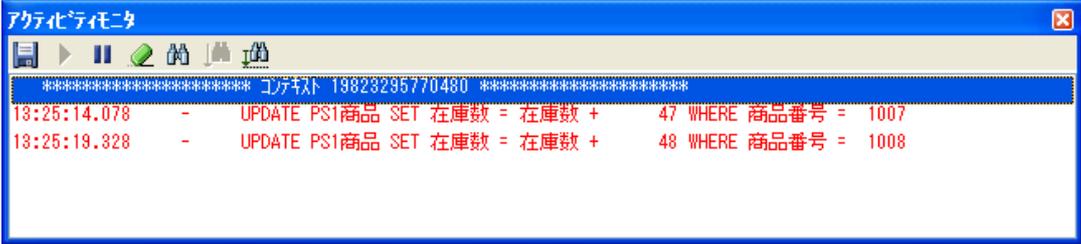
実際にトランザクションキャッシュを表示させてみます。

簡単な例として、商品ファイルを APG で表示させ、修正モードで在庫数を変更する場合をみてみます。



商品番号	商品名	商品タイプ	単価	在庫数
1002	アートル	D	10,200	50
1003	フォックス テリア	D	4,080	50
1004	カリヤ	B	3,060	50
1005	ルカグイ	F	21,420	50
1006	イワ	D	2,040	50
1007	カッポ	F	816	47
1008	ルマツヤク	F	8,160	48
1009	ブツョウ	B	816	50
1010	ルムスター	R	102	50
1011	コアラ	S	4,488	50
1013	オム	B	306	50
1103	ジュウマツ	B	204	50

商品番号 1007 および 1008 について、在庫数を変更した場合、アクティビティモニタには次のようなログが、リアルタイムで記録されます。



```
***** コネクト 19823295770480 *****
13:25:14.078 - UPDATE PS1商品 SET 在庫数 = 在庫数 + 47 WHERE 商品番号 = 1007
13:25:19.328 - UPDATE PS1商品 SET 在庫数 = 在庫数 + 48 WHERE 商品番号 = 1008
```

これにより、データの操作がプログラムで正しく行われているかを確認することができます。



アクティビティログには、トランザクションキャッシュに対するデータ操作内容が、SQL 文の形式で (UPDATE ... とか、INSERT ... とかの形式で) 表示されますが、これはあくまで視覚化のためであり、実際に内部的に、あるいは DBMS に対して、このような SQL 文が発行されているわけではありません。

Magic eDeveloper V10



Magic eDeveloper V10

遅延トランザクション

Copyright © 2008, Magic Software Japan K.K.,
All rights reserved.

第 1 版

2008 年 3 月 28 日

発行

〒151-0053 東京都渋谷区代々木三丁目二十五番地三号

あいおい損保新宿ビル 14 階

マジック ソフトウェア・ジャパン (株)

<http://www.magicsoftware.co.jp/>
