4R−6

トリガ／ストアードモジュールの実装方式

北澤 敦[1], 鴨井 功[1], 麦谷 尊雄[1],
[1]日本電気(株)
斎藤幸義[2]
[2]日本電気ソフトウェア(株)

1 はじめに

リレーショナルデータベースシステムにおける保全制約機能はデータベースの状態を、論理的に整合性のとれた状態に保持する機能であり、代表的なものには、primary key, not null, unique, check, assertion等の制約が存在する[1]。これ等の制約の保守規則は、通常、制約違反を監視し、違反が発生した場合にそのトランザクションをロールバックする方法がとられる。リファレンシャルインテグリティの様に、制約を実現するためにアクションを伴う機能も定義されつつあるが[1]、制約の種類や、アクションが限定されており、一般化された機能提供とはいえない。従って、一般的な制約を論理的に解決するためには、アプリケーションプログラムに必要なアクションを記述することが必要となる。この問題を解決するために、一般的な制約をプロダクションルールとしてデータベースに格納し、論理的に解決する方式として、トリガ機能による実装が提案されている[2]。一方、近年のデータベース管理システムも、様々な機能拡張に伴い、スキーマ情報が複雑化してきており、システム内での情報整合性を保証するための機能も必要となってきている。そこで我々は、リレーショナルデータベース管理システムの基本機能としてトリガ機能を位置付け、SQL言語をベースとした、プロダクションルール記述を実装した。本機能は、我々のプロダクトであるRIQSII V2の定義情報の管理や、表制約機能の実現に利用され、システムの拡張性や生産性の向上に貢献している。トリガ機能の実装に当たっては、性能を重視し、ストアードモジュール方式をトリガ機能に拡張して適用している。本稿では、トリガ機能の概要と、ストアードモジュールによる実装方式を報告する。さらに、トリガによるルールの伝播の問題とルールの遅延について考察する。

2 トリガ機能

トリガ機能は、データベースに対する変更を契機としたプロダクションルールで表される。

```
CREATE TRIGGER trigger-name action-time
  REFERENCING tuple-variables
   production-rule .....

production rule:
  {WHEN (condition)|WHENEVER}
      (action,.....)
```

ここで、action-timeは、トリガが起動されるタイミングを記述する。トリガの起動のタイミングとして、ある表の列に対するinsert,delete, updateおよび、各命令の動作の前後を指示する。トリガは、各命令毎に行単位に起動される。例えば、探索型のupdate命令では、探索条件に適合した導出表が複数の行を持つとすれば、トリガも複数回呼び出されることになる。

```
{BEFORE|AFTER} {INSERT|UPDATE|DELETE}
   ON table-name  OF column-name
```

conditionはSQLの探索条件を記述する。探索条件が真と評価された場合にactionが起動される。actionを無条件に起動する場合には、WHEN文の変わりにWHENEVER文を記述する。actionには任意のSQL文または、RAISE EXCEPTION文を記述する。

```
RAISE EXCEPTION 'message-string'
```

プロダクションルールのリストは、各々のルールの適応順を示す。ルールの適応は、最後のproduction-ruleの評価が完了した場合か、RAISE EXCEPTION文を評価した場合に終了する。トリガ機能の特徴は、データベースの変更状況を変数として知ることができる点である。tuple-variablesには、insert,delete, updateの各命令に対応した変数(組変数)を定義する。

```
REFEENCING OLD tuple-variable-name,
           NEW tuple-variable-name
```

OLD,NEWに対応するtuple-variable-nameで示される組変数の定義域は、1行からなる以下の表である。

1. insertの場合:OLDは無効。NEWは追加対象の値を列の値として持つ表。追加の対象とならない列の値はNULLである。
2. updateの場合:OLDは変更対象となる行の変更前の値を列の値として持つ表。NEWは、変更後の行の値を列の値として持つ表。
3. deleteの場合:OLDは削除対象となる行の各列の値を列の値として持つ表。NEWは無効。

これらの組変数は、トリガのcondition,action部で参照できる。

3 ストアードモジュール

RIQSII V2のデータベースアクセスは、モジュールを介して行われる。モジュールを定義する方法は以下の4通りある。

1. モジュール定義言語を利用して、明示的に生成する場合。
2. 埋め込みSQL文を持つプログラムのコンパイル単位に暗黙に生成される場合。
3. 対話形式の実行時に一時的に生成される場合。
4. トリガに対応して生成される場合。

モジュールは[1]に定義されているように、複数のプロシジャから構成される。各プロシジャは、SQL文から構成され、入力パラメータの定義を持つことができる。トリガをサポートするために、プロシジャ記述として、トリガ制御用の制御文および、探索条件を利用可能としている。

3.1 モジュールコンパイラと再生成

我々の方法で特徴的なのは、データ操作に関する全てのプロシジャがコンパイルされ、基本命令として登録されることである。insert,delete, updateといったデータ操作文に対応するプロシジャも基本命令セットとして定義される。特に、更新命令に付随して発生すべきルールは、全てトリガとしてルール化され、トリガの基本命令セットとして実現される。例えば、内部的な構造としてサポートされるインデックスは、特別な表とみなされ、トリガのactionとしてメンテナンスされる。常にインデックスを登録するケースではWHENEVERであるし、特別な条件の場合のみインデックスを登録するようなケースでは対応するconditionが与えられる。

我々のこの実装方式では、論理的な構造(すなわち表)に対する更新時に発生する物理的な構造への変換作業を、トリガとしてルール化することで、自動的に実装できる特徴がある。例えば、仮想表の更新や、物理的にチェインを有する表の更新などが考えられる。

一方、特定の更新命令に関連するトリガは、モジュールコンパイル時に決定され、トリガ基本命令セットとして展開されるので、実行時に関連するトリガを解決する必要がないため、実行時の性能が良いという特徴もある。逆に、ルールの発生／削除に伴い、トリガの追加／削除が発生するケースでは、モジュールの再コンパイルが必要となる。データベース定義の変更は、定義情報自身の更新ルールから発生するトリガによって通知され、実行に先だって対応するモジュールを再コンパイルすることで、この問題を解決している。もちろんこの場合でも、アプリケーションプログラムは再コン

Implementation of Triggers and Stored Modules
Atsushi Kitazawa[1], Isao Kamoi[1], Takao Bakuya[1],
Yukiyoshi Saito[2]
[1]NEC Corporation, [2]NEC Software, Ltd.

バイルする必要はない。これは、モジュールがアプリケーションプログラムから独立しているからである。

3.2 モジュール構造の利点

モジュール構造には、SQL文の共有利用や、権限の代行といったモジュール自身の利点の他に、モジュールをコンパイルすることによる、資源管理の一元化の利点がある。すなわち、モジュールのコンパイル時に、全てのプロシジャで利用する資源と、参照／更新形態を知ることができる。これは、モジュールをトランザクションに対応させた場合、モジュール毎にISOLATION LEVELを設定しておくことで、各資源に対するロック制御方式を事前に決定できることを意味する。また、各モジュールに関連する資源のオープン処理を一括して行うことも可能となる。特に、モジュールコンパイラはそのモジュールに関連する全ての資源を格納資源レベルまで解決しているため、実行時の論理資源から物理資源への変換も不要である。これらの資源管理は、プロシジャのモジュールへのマージ機能として実現される。

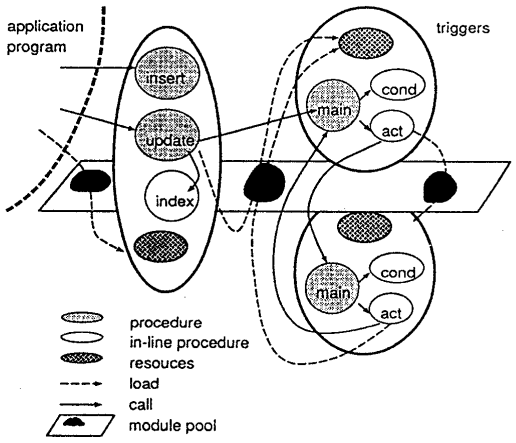

図1:stored module

3.3 トリガ命令セットの展開と起動

図1に、ストアードモジュールを利用したトリガの実現方式を示す。トリガに固有のモジュールコンパイラの機能は、基本命令セットの展開である。基本命令セットの展開には、インライン型と呼び出し型があり、インライン型では、トリガに関連する全てのプロシジャの命令セットがマージ機能によって呼び出し側のプロシジャにマージされる。一方、呼び出し型では、トリガモジュールのロード命令と、プロシジャの実行命令が設定される。一般に、ルールのネストが発生する可能性のあるケースでは、ルールのループが発生する可能性もあるため、呼び出し型が採用される。また、ネストしたルールの追加／削除によるモジュールのコンパイルコストが低いという特徴もある。これに対して、NOT NULLのチェックや、インデックスのメンテナンスの様に、ルールがネストする可能性のない場合は実行性能の良いインライン型を選択できる。 各トリガのconditionおよびaction は、各々プロシジャに対応する。また、トリガ定義自身もプロシジャである。このプロシジャは、メインプロシジャと呼ばれ、プロダクションルールのフローコントロールを行う。これらのプロシジャはインライン型として展開される。呼び出し型では、トリガは、ロード基本命令と、呼び出し基本命令で実行される。ロード基本命令は、モジュールに対して発行され、呼び出し基本命令はプロシジャに対して発行される。ロード基本命令では、モジュールのモジュールプールへの展開と、関連資源のオープンおよび、スタックの拡張が行われ、呼び出し基本命令では、パラメータのリンクおよびプロシジャの実行が行われる。トリガのループが発生した場合は、プール上のモジュールが再利用される。

3.4 ルールの伝播と誤り

トリガによるルールの伝播に伴う誤りの種類として、以下の3種類がある。

1. ルールの適用でループが発生する場合。update,insert命令で更新を波及して元もと更新した表の項目を再度更新する場合。例えば、リファレンシャルインテグリティで、ON UPDATE CASCADEがループを構成する場合。
2. RAISE EXCEPTIONに対応するconditionが常に真である場合。例えば、リファレンシャルインテグリティで、自分自身のプライマリキーを参照する場合。
3. ルールの適用順によって、結果が異なる場合。例えば、リファレンシャルインテグリティで同一の表に、ON DELETE CASCADEと、ON DELETE SET NULLが同時に波及するような場合。

トリガの定義時に誤りを検出するためには、ルールが伝播するか否かをconditionの真偽値として定義時に判定する必要がある。この判定には、トリガをその元となる制約に再度変換し、conditionに付加する必要がある。

3.5 ルールの遅延

トランザクション内でのルール適用の遅延は、性能の改善をもたらす場合がある。これは、更新量の多いトランザクションで、一括処理による二次記憶へのアクセス回数の削減が狙いである。ルールの適用を遅延するための考慮点は、以下の3点である。

1. トランザクション内で発生するトリガ起動の組変数に関してのみ、処理が可能なこと。すなわち、表データ全てを再度読みだして確認するのでは意味がない。(ルールの組変数に対する置き換えが可能かどうかの判定方法は[3]に示されている。)
2. ルールを適応することによって発生するルールも、トランザクション内で発生するトリガであること。従って、遅延処理中に新たな処理入力データが発生する可能性があること。
3. ルールを適応するタイミングは、少なくともトランザクションの終了時であるが、もし、ルールがそのトランザクションで利用する表に対するアクションを含むならば、トランザクションでその表を参照する場合にルールが適応されなければならないこと。

我々は、一括処理による性能を考慮して、インデックスに関連するルールを遅延する指定を可能としている。インデックスは、ルールを再生成することがなく、従って、2番目の考慮点を考慮する必要が無く、しかも、ソートによる一括処理で更新の高速化が可能である。

4 おわりに

RIQSII V 2におけるトリガ機能とその実装方式で、システムの拡張性、性能面での考慮点について示した。今後、データベースシステムがサポートする構造の拡張に伴い、ルール記述の機能拡張も必要になると考えられる。特に、機能の柔軟性の観点から、ユーザ定義関数のサポートが重要である。

参考文献

[1] ISO,Database Language SQL

[2] Jenifer,W.,Roberta,J.,C.,and Bruce,G.,L.:Implementing Set-Oriented Production Rules as an Extension to Starburst,Proc.17th VLDB Conf., Barcelona,Sept.(1991).

[3] Stefan,C.,Jenifer,W.:Deriving Production Rules for Constraint Maintenance, Proc.,16th VLDB Conf.,Brisban(1990).