Living Without Record Locking

By David Adams

TN 06-38

Overview

4th Dimension および 4D Client/4D Server は、マルチプロセスまたはマルチユーザデータベースに特有の複雑な処理の多くを意識しなくてもよい設計になっています。データベースエンジンレベルでレコードロックが管理されるメカニズムにより、特別なプログラミングを意識的にしなくても、マルチプロセスまたはマルチユーザデータベースを開発することができます。しかし、4D の自動レコードロックシステムにすべてを任せるわけにいかない場合もあります：

- Web または SOAP を使用したレコード更新システム。
- 配列または変数を使用したレコード更新システム。具体的には、AreaList、4D View、リストボックスでレコードデータを表示・更新するシステム。
- 外部ファイルを使用したレコード更新システム。たとえば複数のサテライトオフィスから中央のデータベースにレコード更新情報をファイルで転送するようなシステム。

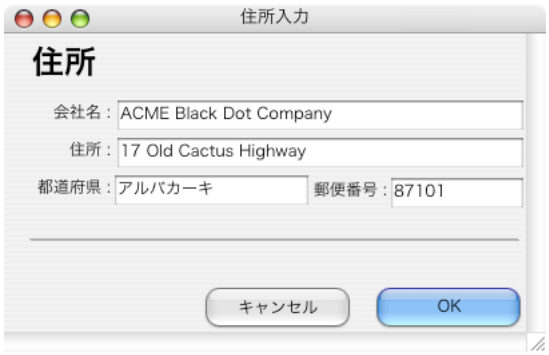
上記のような場合、レコードは中央のデータベースエンジンにおいてロックされた状態でキープされているわけではありません。そのため、他のユーザまたはプロセスによって削除または更新される可能性があります。そのような可能性がある以上、次の質問に答えられるカスタムレコード管理プログラムが必要です：

- レコードは他のユーザまたはプロセスによってロックされているか。
- レコードはすでに削除されたものか。
- 扱おうとしているレコードデータは最新のものか、それとも昔のものか。

このテクニカルノートは、上記の質問に対する答えを得る方法、また明快で一貫性があり、信頼できるレコード更新ポリシーを導入する方法を論じています。本テクニカルノートを読めば、この問題に対応するプログラムが意外にも簡単であることが理解できるはずです。しかし、詳細に進む前に、4th Dimension の自動レコードロックシステムの動作を確認しておきましょう。この点をはっきりさせることで、カスタムソリューションの方向性が分かり、自動ロックシステムで代用できるものを作ろうとして無駄な時間を費やさなくても済むからです。

Review: Automatic Record Locking Features

4th Dimension および 4D Client/4D Server のレコードロックは、標準の 4D コマンドを使用していれば、自動的に作動しています。このテクニカルノートで説明に使用するのは単純な住所録データです。サンプルレコードは次のようなものです：



レコードを編集できるものとして表示する前に、4th Dimension および 4D Client/4D Server はそのレコードは他のユーザまたはプロセスによって編集されているかを調べます。コンフリクトが検出された場合、次のようなアラートが表示されます：

ユーザモード、あるいは MODIFY SELECTION などの標準コマンドでレコードが表示されている限り、このメカニズムを実現するための追加プログラミングは不要です。レコードのロックは 4th Dimension では自動的に管理されており、特別に意識されていないことも少なくありません。このシステムは 4th Dimension および 4D Client/4D Server がオリジナルのレコードデータを直接表示しているのであれば、完璧に動作します。しかし、これから説明するように、実際にはそのようなわけではありません。

4th Dimension Never Shows Records

4th Dimension および 4D Client では、レコードはユーザモード、または MODIFY SELECTION や DISPLAY SELECTION のようなコマンドで表示されます。いずれの方法でも、実際に表示されているのはレコードのコピーであり、レコードそのものではありません。通常、レコードとそのコピーは同等のものです。シングルプロセス、シングルユーザデータベースであれば、まず違いを意識する必要はありません。問題は、単一のレコードのコピーが同時に複数の箇所で使用される場合に表面化します。これは 4D Server のトリガ、およびレコードにアクセスできるすべてのプロセスについて言える問題です。

Record Locking and Record Copying

4th Dimension/4D Server は、4D プロセスを使用し、レコードのロックをデータベースエンジンレベルで管理しています。あるプロセスがレコードを表示しようとするとき、データベースエンジンがオリジナルレコードを読み込み、プロセスに対してそのコピーを送信します。元のレコードはデータベースエンジン側に残っています。仮にネットワーク通信が途切れてしまい。4D Client がレコードの更新を保存できなければ、レコードは元のままです。4D Client が更新していたのはレコードのコピーであり、オリジナルのデータではないからです。このシステムは、4D のプロセスに対して作動します。あるプロセスがレコードをロックされたものとして保有している場合、ロックはそのプロセスの動向に依存しています。そのプロセスが終了すれば、データベースエンジンがロックしていたレコードはすべてロックを解かれ、プロセスのローカルレコードコピーは破棄されます。4th Dimension および 4D Client/4D Server では、レコードは常に 4D プロセスによって保有されますが、SOAP または Web 接続の場合はそうではありません。そのようなコンテキストでは、リクエストの処理が完了した時点でプロセスがアボートあるいは再利用されるため、レコードを保有するプロセスがなくなってしまいます。標準の 4D/4D Open/4D Client インタフェースであっても、レコードを配列や変数にコピーした場合、元のレコードはロックされません。

Writing a Custom Record Locking System

レコードをロックするためにはプロセスが必要、という基本メカニズムを理解すると、デベロッパはレコードロック用のプロセスを起動することで SOAP、Web その他のシチュエーション用にカスタムレコードロック管理システムを構築しようという誘惑に駆られるかもしれません。事実、4D の Web コンテキストモードはそのようなアプローチを採用しています。しかし、個人的な見解として、よほど特別なメリットがあるのでない限り、その種のカスタムレコードロック管理システムを開発することはあまり推奨できません。導入の煩雑さを別にしても、そのような意図的なロックをどれだけの時間、維持するべきかという面倒な問題があります。1 分、5 分、24 時間...どのような設定値にしたとしても、ある場合には長過ぎ、ある場合には短すぎるでしょう。結論からいえば、そもそもカスタムレコードロック管理システムは不要です。

Tools for Writing A Custom Record Locking System

カスタムレコードロック管理システムは不要、という今回の結論を受け入れがたく思う人もいることでしょう。読者もそのような意見の持ち主であれば、是非、一度カスタムレコードロック管理コードを作成してみてください。結果に満足できるかどうかは別としても、カスタムレコードロック管理システムの開発はたいへん勉強になるからです。一度でもそのような経験を積んでいれば、4D の内部メカニズムに対する理解が深まり、次から非常に効率良く開発ができるようになるはずです。この課題に取り組む前に次のアドバイスを考慮してください：

始める前に簡単で分かりやすいテストデータベースを作成してください。既存のデータベースを使用するよりも、そのほうがコードのテストが容易です。

PUSH RECORD コマンドはカレントプロセスを含め、すべてのプロセスに対してレコードをロックします。レコードロックのテストする場合、この仕組みを活用すると便利です。

トランザクション中にロックされたレコードは、トランザクションを中止または確定するまでロックされています。つまりレコードの集合をまとめてロックすることができます。

Record Modification Example

サンプルとして使用するレコードは次のようなものです：

ACME Black Dot Company
17 Old Cactus Highway
アルバカーキ 87101

このデータベースのレコードは、倉庫の Web インタフェース、自宅オフィスの 4D Client、外出先の PDA から SOAP 経由で更新できると想定しましょう。倉庫の従業員(Web 使用)と営業担当(SOAP 使用)が同時にレコードにアクセスしました。それぞれの値は次のとおりです：

倉庫	営業担当	サーバ
ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101	ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101	ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101

値はいずれも同じです。さて、次に営業担当のジョアンが顧客の新住所は 28 New County Road であると知りました。彼女はレコードを更新します。

倉庫	営業担当	サーバ
ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101	ACME Black Dot Company 28 New County Road アルバカーキ 87101	ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101

ジョアンがレコードを保存した結果、サーバの値が更新され、送り返された新しい値がローカルコピーとなります。この時点で営業担当とサーバが一致し、倉庫の値は古くなっています：

倉庫	営業担当	サーバ
ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101	ACME Black Dot Company 28 New County Road アルバカーキ 87101	ACME Black Dot Company 28 New County Road アルバカーキ 87101

一方、倉庫では、アーネストが ACME 向けの出荷準備を終えました。発送先と画面を見比べ、住所が間違っていないことを確認した後、レコードを保存します。その結果、今度は倉庫とサーバが古い情報で一致し、営業担当のジョアンが更新した値は失われてしまいます：

倉庫	営業担当	サーバ
ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101	ACME Black Dot Company 28 New County Road アルバカーキ 87101	ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101

これは問題点を浮き彫りにするための極端に単純な例です。もっと複雑なシナリオも考えられますが、基本的な問題はどれも同じであり、マルチプロセス/マルチユーザデータベースでひとつのレコードが同時に複数の場所で開かれる可能性があることに起因しています。そのような状況下でレコードを更新し、間違った順序でレコードを上書き保存すると問題になるのです。この例題を踏まえた上で、当初の課題である自動レコードロックの管理下にない場合の対処法に論題を戻すことにしましょう。

Managing Multiple Edits

以下はテクニカルノートの冒頭で提起した問題、およびプログラム的観点からみた解決法です：

- レコードは他のユーザまたはプロセスによってロックされているか。

4D 的な方法で他のプロセスまたはユーザがロックしていれば Locked 関数が True を返します。

- レコードは削除されたものか。

削除されたのであれば、見つからないはずなので、QUERY で確認することができます。ただし、4D の評価ルールで重複しないユニークな値のキーフィールドがある場合に限られます。

- 扱おうとしているレコードデータは最新のものか、それとも昔の情報か。

前述の例における ACME Black Dot の住所のように、レコードが更新前の古いデータを示している状況の検出には、若干の事前準備とカスタムコーディングが求められます。次に取り上げているのはこの点です。

Overview

更新前の古いレコード(ステールレコード)を検出するためには、レコードのカレントコピーをサーバの値と比較できる方法がなければなりません。前述の例では、次のようなステールレコードが倉庫からサーバに送信されていました：

倉庫	サーバ
ACME Black Dot Company	ACME Black Dot Company
17 Old Cactus Highway	28 New County Road
アルバカーキ 87101	アルバカーキ 87101

倉庫のレコードがステールであることを検出するひとつの方法は、倉庫が受け取ったレコードのコピーを更新データと一緒に送信するというものです。そのようにすれば、サーバは更新を実施する前にほんとうの値とコピーの値を比較でき、ステールレコードを検出できます。送信量が倍になり、ひどく非効率的であるとはいえ、基本コンセプトは間違っていません。元の値を比較すれば、そのレコードがどのオリジナルからコピーされたのかを知ることができます。倉庫の持っている元のレコードとサーバのレコードを比較すれば、違いは明らかです：

ACME Black Dot Company 17 Old Cactus Highway Albuquerque 87101
ACME Black Dot Company 28 New County Road Albuquerque 87101

倉庫の保有しているコピーが陳腐化していることが分かります。この方法唯一の難点は、レコード全体の送受信と処理が求められる部分にあります。それよりも優れているのは、簡単に計算でき、データ量の少ない署名情報、たとえばハッシュやシーケンシャルな管理番号です。

Creating Signatures with a Hash

ハッシュは、ブロックデータからコンパクトな署名を作成する方法として定着しています。元のデータをすべて転送する代わりに、元のデータからハッシュを作成して転送すれば、サイズが節約できます。ハッシュが一致しないということは、元のデータも一致しないということです。このテクニックは、ステールレコードの検出には充分なのですが、必要以上に処理能力を使ってしまうところが問題です。ハッシュの生成には、レコード全体をスキャンし、数々の数値演算を実施することが関係しています。やはりユニークなレコード値を管理するための簡単、確実、高速なソリューションは、数値タイプのカウンタです。

Note　ハッシュに関する詳細な考察については、テクニカルノート 05-43「HashTools コンポーネント」、05-43「ハッシュを使用して検索を最適化する」をご覧ください。

Adding Version Numbers to Records

レコードのコピー同士を比較しなくてはならない場合、テーブルに重複不可のバージョン管理

番号を追加すると便利です。このフィールドは、レコードが保存されるたびに加算されるようにしておきます。前述の例に適用すると、次のようになります：

倉庫	営業担当	サーバ
ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101 更新#1	ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101 更新#1	ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101 更新#1

値はいずれも同じです。さて、次に営業担当のジョアンが顧客の新住所は 28 New County Road であると知りました。彼女はレコードを更新します。

倉庫	営業担当	サーバ
ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101 更新#1	ACME Black Dot Company 28 New County Road アルバカーキ 87101 更新#1	ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101 更新#1

ジョアンがレコードを保存した結果、サーバの値が更新され、送り返された新しい値がローカルコピーとなります。この時点で営業担当とサーバが一致し、倉庫の値は古くなっています：

倉庫	営業担当	サーバ
ACME Black Dot Company 17 Old Cactus Highway アルバカーキ 87101 更新#1	ACME Black Dot Company 28 New County Road アルバカーキ 87101 更新#2	ACME Black Dot Company 28 New County Road アルバカーキ 87101 更新#2

一方、倉庫では、アーネストが ACME 向けの出荷準備を終えました。発送先と画面を見比べ、住所が間違っていないことを確認した後、レコードを保存します。サーバは倉庫から送信されたレコードを解析し、倉庫が更新#1(古い情報)を使用していることを検出します。サーバはアーネストに対し、レコードの情報が古いことを通知するエラーメッセージを送信します。

メカニズムは簡潔ですが、これだけで実に複雑なシナリオにも対応できることに気づきます。サーバサイドで更新/バージョン管理番号が正しく管理されている限り、エンドユーザがどのような手段でレコードを入手したのか、どれほどの期間そのレコードを保有していたのか、そしてどのような手段で更新を試みたのかに関係なく、ステールレコードをはじきだすことができるのです。インポート/エクスポート、SOAP、Web、あるいは他のどんな方法で更新するとしても簡単なレコード更新カウンタを設けるだけで大丈夫です。(このアイデアを提供してくれたシドニー市・LEAP Legal Software の Mark Burgess 氏に感謝します。)

Maintaining Update Values

レコード更新情報を管理するもっとも合理的かつ簡単な場所はトリガです：

```
Case of
   ¥ (Database event=On Saving New Record Event )
      [Address]Update_Number:=1
   ¥ (Database event=On Saving Existing Record Event )
      [Address]Update_Number:=[Address]Update_Number+1
End case
```

Detecting Stale Records: Where to Put the Code

残る問題は、ステールレコードを検出するコードをどこに記述するかという一点だけです。レコードの更新を扱う場所、すなわち SOAP 接続、Web 接続、インポートに関係したコード、あるいは 4th Dimension のプロジェクトメソッドに記述することができます。あるいは、そのコードもトリガに組み込んでしまうのが一番手っ取り早いかもしれません：

```
C_LONGINT($0;$ErrorCode)
$ErrorCode:=0
Case of
   ¥ (Database event=On Saving New Record Event )
      [Address]Update_Number:=1
   ¥ (Database event=On Saving Existing Record Event )
      If (Old([Address]Update_Number)#[Address]Update_Number)  ` 更新管理番号がない (0)
         ` あるいは陳腐化したレコードなので保存を拒否する
         $ErrorCode:=-16000  ` カスタムトリガエラーコードは -32,000 から -15,000 .を使用する
      Else
         [Address]Update_Number:=[Address]Update_Number+1
      End if
End case
```

レコード保存時のメソッドにコードを記述するとしても、セーフガードとして既存レコード保存時のトリガを作成するのがベストです。

Variations on a Theme

今回のテクニカルノートでは、倍長整数フィールドをレコード更新情報の管理に使用しました。data-time 形式のタイムスタンプを使用することもできますし、時系列にしたがってシーケンシャルにレコードのコピー同士を比較できる方法であれば、どんな方法を用いても構いません。とはいえ、よほど特別な理由がない限り、倍長整数を使用するのが無難だと思います。

Summary

4th Dimension の自動レコードロックシステムは、マルチプロセス/マルチユーザデータベースの開発にかかる負担を大幅に軽減するものです。自動的なレコードロックが利用できないような場合でも、同時に開かれたレコードの二重更新は簡単な工夫で管理することができます。レコードが削除されていないことはコマンドで調べることができ、別のプロセスまたはユーザに更新されてしまい、情報が古くなってしまったステールレコードは、倍長整数タイプのフィールドをひとつ追加して管理するだけで容易に検出することができるからです。