SQL Remote for ASE 環境から Mobile Link への移行

もくじ

1.0 前提条件

このドキュメントは、読者が SQL Remote for ASE に関する実用的な知識を持っていることを前提としています。ただし、Mobile Link についての知識の有無は問いません。このドキュメント、特にコード・サンプルは、Mobile Link に熟知していない方でも技術的な詳細を理解できる内容となっています。このドキュメントでは、リモート・データベースと統合データベースの違い、パブリケーション、リモート・ユーザ、統合ユーザとサブスクリプション、カラムによるサブスクライブ、デフォルトの競合解決などの SQL Remote の概念について言及しますが、その詳しい説明は行いません。また、SQL Remote for ASE を中心に説明を進めますが、SQL Remote for ASA 環境を Mobile Link に移行する場合は、その多くの概念をそのまま適用することができます。

1.1 記載されているコードおよびドキュメントについて

このドキュメントに記載されているすべてのコードは、Windows 2000 (Service Pack 3) で稼働する Adaptive Server Anywhere バージョン 9.0.2.2451 および Windows 2000 (Service Pack 4) で稼働する Adaptive Server Enterprise バージョン 12.5.1 ESD #2 (EBF 11665) で動作がテストされています。また、Adaptive Server Anywhere マニュアルに言及している箇所は、Adaptive Server Anywhere バージョン 9.0.2.2451 に付属のマニュアルに基づいています。記載されているどのコードも特定バージョンまたは特定ビルドの Adaptive Server Anywhere または Adaptive Server Enterprise に固有なものではありませんが、コードを実稼働環境に展開する場合は、必ず事前にご使用の環境でテストしてください。

1.2 Mobile Link の概要

Mobile Link は、統合データベースと呼ばれるメイン・データベースとさまざまなリモート・データベース間の双方向の同期を可能にするセッションベースの同期システムです。統合データベースには ODBC に準拠した任意のデータベースを使用でき、ここにあらゆるデータのマスタが保持されます。リモート・データベースには Adaptive Server Anywhere または Ultra Light データベースのいずれかを使用できます。

通常、同期は、Mobile Link リモート・クライアントが Mobile Link 同期サーバへの接続を確立した時点から開始されます。同期中、Mobile Link クライアントは、前回の同期以降にリモート・データベースに対して行われた変更をアップロードします。Mobile Link 同期サーバはこのデータを受け取ると、統合データベースを更新し、統合データベースの変更をリモート・データベースにダウンロードします。

Mobile Link 同期システムを構成する各コンポーネントの詳細については、製品マニュアルを参照してください。『*MobiLink Administration Guide*』の「Synchronization Basics」では、Mobile Link の動作の概要がわかりやすく説明されています。このドキュメントを読み進める前に、この章を読むことをおすすめします。少なくとも、この章に目を通し、このドキュメントで使用されている用語および概念を理解しておいてください。すべての iAnywhere Solutions 製品の製品マニュアルは、http://www.ianywhere.jp/dl/dl_prd.html からオンラインで利用できます。

1.3 なぜ移行が必要なのか

現在抱えているニーズが SQL Remote for ASE ですべて満たされているのであれば、急いで移行する必要はないでしょう。Mobile Link に新しい機能 (おそらくより優れた機能) があるからといって、必ずしも移行が必要というわけではありません。現状で何も問題がなければ、その解決も必要ないわけです。ただし、外部要因の変化が、より新たなテクノロジへの移行の引き金になることもあります。

Mobile Link には、SQL Remote for ASE にはない機能が数多くあります。主な機能の違いを次にいくつか紹介します。

- 理論上、Mobile Link では任意の ODBC 準拠の統合データベースを使用できます。一方、SQL Remote が統合データベースとして使用できるのは Adaptive Server Enterprise および Adaptive Server Anywhere のみです。このドキュメントの執筆時点で Mobile Link でサポートされている統合データベースは、Adaptive Server Anywhere、Adaptive Server Enterprise、Microsoft SQL Server、Oracle、および IBM DB/2 です。
- Mobike Link では、Adaptive Server Anywhere リモート・データベースまたは Ultra Light データベースとの同期が可能です。一方、SQL Remote for ASE では、Adaptive Server Anywhere リモート・データベースを使用したレプリケートのみが可能です。
- Mobile Link では、スクリプトを作成することで、データをどのように統合データベースに適用し、どのデータをリモートに送信するかを定義することができるため、データのアップロードとダウンロードをより柔軟に行うことができます。
- Mobile Link 環境では、アップロードとダウンロードを 1 回の同期で行えるため、データのレイテンシが短くなります。一方、SQL Remote 環境では、メッセージは共有メッセージ・システムに書き込まれますが、そのメッセージが反対側のレプリケーション・システムでいつ取り出されるかについての保証は一切ありません。
- Mobile Link では、128 ビット暗号化により同期ストリームを簡単に暗号化できます。一方、SQL Remote のメッセージには、暗号化ではなく難読化が行われ、メッセージに暗号化を付加するには、SQL Remote 用のカスタム・エンコーディングおよびデコーディング DLL を作成する必要があります。
- Mobile Link では、統合データベースで発生したイベントに基づいて、同期を開始するようリモート・データベースに要求することができます。一般に、このプロセスは**プッシュ同期**と呼ばれます。

Sybase および iAnywhere Solutions は、Adaptive Server Enterprise データベースから Adaptive Server Anywhere データベースにデータを移動することのできる 3 つのテクノロジを提供しています。

1. Replication Server (接続ベース)
2. Mobile Link (セッション・ベース)
3. SQL Remote for ASE (メッセージ・ベース)

常時接続された少数のデータベース間で高速な双方向レプリケーションを低レイテンシで行う場合は、明らかに Replication Server が最適でしょう。一方、接続されていない環境向けのソリューションとしては、Mobile Link と SQL Remote for ASE が似たような役割を果たします。

iAnywhere Solutions は、次期メジャー・リリースの SQL Anywhere Studio において SQL Remote for ASE の生産終了を発表することを決定しました。SQL Remote for ASE の現行ユーザは、Remote for ASE の代わりに Mobile Link を使用することで移行できます。SQL Remote for ASE の生産終了が発表されるからといって、SQL Remote for ASA の状況がいかなる形でも影響を受けることはありません。SQL Remote による Adaptive Server Anywhere データベース間のレプリケーションが SQL Anywhere Studio の強力かつ有望な製品であることに変わりはありません。

1.4 移行目標

ある製品から別の製品への移行を検討する際に、また展開先サイトに DBA が不要であるという SQL Anywhere Studio の当初からの理念を貫く上でも、iAnywhere Solutions 製品間の移行がリモート・ユーザにとっても簡単に行えるものでなければなりません。リモート・ユーザが移行による変化を意識せずに済み、エンド・ユーザのトレーニング・コストを伴うことなく移行できなければなりません。また、移行を段階的に進めることも必要です。システム内に配備されているすべてのユーザが決められた日にアップグレードを実行すると想定するのは非現実的であり、現行ユーザがアップグレードしていないという理由だけで新しいデータにアクセスできないような状況を避けることも重要です。機能を限定してしまう移行パスも好ましくありません。実装する新しいシステムでは、従来のシステムで可能だったすべてのこと、またできればさらに多くのことを実行できなければなりません。以降の項は、ここに掲げた移行目標に沿った内容となっています。

このドキュメントの残り部分は 2 つの項に分かれています。最初の項では、現在 Adaptive Server Anywhere リモート・データベースを使用し、リモート・データベースに対して SQL Remote for ASA を実行しているユーザを、Mobile Link を使用し、Mobile Link 統合データベースと同期する環境へと移行するために、リモート・サイトで必要な変更について扱います。2 番目の項では、統合データベースに対して必要な変更について扱います。

2.0 リモート・データベースでの変更

リモート・ユーザにとってできるかぎり簡単に移行できるようにするために、以降に示す手順では、リモート・ユーザに新たなデータベースの展開は要求しません。リモート・ユーザが引き続き既存のリモート・データベースを使用できるようにすることで、移行はリモート・ユーザにとって大幅に簡単になるはずです。

2.1 リモート・サイトに展開する新たなソフトウェア

リモート・データベースに dbmlsync.exe 実行プログラムが存在しない場合は、新たな Adaptive Server Anywhere ソフトウェアの配備が必要になる可能性があります。リモート・ユーザに新たなソフトウェを展開する必要がある場合、より新しいバージョンの Adaptive Server Anywhere にアップグレードしようと考える方もいるでしょう。Mobile Link クライアントを実行するために展開する必要がある新たな Adaptive Server Anywhere ファイルは、dbmlsync.exe とそれに関連するストリーム DLL (dbmlsock9.dll または dbmlhttp9.dll) のみです。展開の詳細については、『*ASA Programming Guide*』の「Deploying Databases and Applications」を参照してください。

2.2 SQL Remote と Mobile Link の比較

リモート・データベースに必要な変更について詳しく説明する前に、レプリケーションと同期の概念、リモート・サイトにおけるその相違点または類似点を説明しておきましょう。

SQL Remote	**Mobile Link**
SQL Remote は、リモート・データベース上のパブリケーションに基づいて、統合データベースに送信する必要のあるデータを判断します。標準ではリモートで行われたすべての変更が統合データベースに送信されるため、通常、リモート上のパブリケーションには、カラムまたはサブクエリによるサブスクリプショ	Mobile Link クライアントでも、リモート・データベース上のパブリケーションにより、同期する必要のあるデータが判断されます。パブリケーションに WHERE 句を指定することで、統合データベースに送信するローを制限することもできます。実際、SQL Remote クライアントのものと同じパブリケーショ

ンはありません。WHERE 句を使用することで、リモートから統合データベースに送信するローを制限することもできます。	ンを Mobile Link クライアントでも使用できます。
SQL Remote クライアントでは、変更の送信先データベースを識別するために、リモート・データベースに統合ユーザが定義されます。この統合ユーザは、リモート・データベースに定義されたパブリケーションにサブスクライブします。	Mobile Link クライアントでは、リモート・データベースに同期ユーザが定義されます。同期ユーザは同期サブスクリプションによってパブリケーションに関連付けられます。
メッセージを書き込む共有メッセージ・システムの場所とその方法を SQL Remote が認識できるよう、メッセージ・システム・パラメータがリモート・データベースに定義されます。	Adaptive Server Anywhere を使用する Mobile Link クライアントは Mobile Link 同期サーバに直接接続するため、メッセージ・システムの場所を指定する代わりに、使用している通信ストリーム・タイプと Mobile Link 同期サーバの場所を指定します。

次のサンプル・コードは、リモート・データベースに SQL Remote 情報を定義する方法を示しています。
`CREATE PUBLICATION p1 ( TABLE t1 );`
`GRANT PUBLISH to u1;`
`CREATE REMOTE MESSAGE TYPE 'file'`
`  ADDRESS 'u1';`
`GRANT CONSOLIDATED TO cons TYPE`
`  file ADDRESS 'cons';`
`CREATE SUBSCRIPTION TO cons FOR p1;`
`SET REMOTE file OPTION`
`  "public"."root_directory" =`
`  'r:nnmsgs';`
	次のサンプル・コードは、リモート・データベースに Mobile Link 情報を定義する方法を示しています。
	`CREATE PUBLICATION p1 ( TABLE t1 );`
	`CREATE SYNCHRONIZATION USER u1;`
	`CREATE SYNCHRONIZATION SUBSCRIPTION`
	`  FOR u1`
	`  TO p1`
	`  TYPE 'TCPIP'`
	`  ADDRESS`
	`  'host=10.2.3.1;port=600'`
	`  OPTION 'sv=v1';`

統合データベースにどの変更を送信するかを判断する際に、SQL Remote は Adaptive Server Anywhere トランザクション・ログをスキャンし、このトランザクション・ログから統合データベースにレプリケートする必要のある操作をフィルタリングします。リモート・データベースに対して実行された各操作 (COMMIT 文を含む) がメッセージとして統合データベースに送信されます。すなわち、同じローが 1000 回更新された場合、1000 個の更新が統合データベースに送信され、操作の順序がリモート・データベースと統合データベース間で維持されます。	統合データベースにどの変更を送信するかを判断する際に、Mobile Link クライアントは Adaptive Server Anywhere トランザクション・ログをスキャンし、このトランザクション・ログから統合データベースに同期する必要のある操作をフィルタリングします。Mobile Link クライアントによってスキャンされた操作のうち、同一のローに対する操作は 1 つの操作にまとめられます。すなわち、同じローに対する 1000 回の更新は、1 つの更新として送信されます。デフォルトでは、リモートから送信されるすべての変更が単一のトランザクションで統合データベースに適用されます。各操作が独立したトランザクションで、トランザクションによるアップロードが使用されている場合を除き、リモートでの操作順序を保証する方法はありません。

上記の表内のサンプル・コードで、同期ユーザがリモート・データベースのパブリッシャと同じ名前 (u1) で定義されているのは偶然ではありません。その理由は後に明らかになります。

2.3 リモート・サイトでの移行手順

1. リモートで dbremote を実行し、未処理のすべての変更を統合データベースに送信します。これにより、統合データベースから送信済みのすべての変更がリモート・データベースに適用されます。次の手順に進む前に、dbremote 出力ログでエラーがないか確認してください。

2. 統合ユーザを削除します。これにより、このリモートで SQL Remote メッセージを受信したり適用したりできなくなります。
3. 新しい同期ユーザとサブスクリプションを作成します。これにより、リモート・データベースで新たに行われたすべての変更が、SQL Remote ではなく Mobile Link によって同期されるようになります。同期ユーザの名前には、リモート・データベースの *現在のパブリッシャ*を使用します。
4. dbmlsync を実行し、最初の同期を実行します。

上記の手順は、Windows バッチ・ファイルで簡単に自動化できます。次に示す Windows バッチ・ファイルと関連 SQL ファイルは、上記の手順を単純に実装したものです。このバッチ・ファイルは Windows NT オペレーティング・システム群のシェルに固有です。

サンプル・バッチ・ファイル

```
@rem
@rem Run dbremote to send changes and pick up outstanding
       messages
@rem
start /wait dbremote -c "eng=remote;uid=DBA;pwd=SQL" -k -v -ot
       out.txt
@rem
@rem Parse the dbremote output file for errors
@rem
type out.txt | grep "E. " > check.txt
type out.txt | grep "Not Applying Messages" >> check.txt
type out.txt | grep "Not Applying Operations" >> check.txt
type out.txt | grep "Missing Message" >> check.txt
@rem
@rem See if check.txt contains any information in it, and if
@rem it does, do not execute migrate.sql
@rem Note: The FOR command should all be on a single line
@rem
FOR /F "tokens=3* delims= " %%A IN
   ('DIR check.txt /-C /N ^| FIND /I "check.txt"') DO SET
       ACTSIZE=%%A
IF 0 NEQ %ACTSIZE% GOTO error_exists
@rem
@rem Run the SQL File to migrate the remote to use MobiLink
@rem
dbisql -nogui -c " eng=remote;uid=DBA;pwd=SQL " read migrate.sql
goto end
:error_exists
@echo "An error occurred when running dbremote"
type check.txt
:end
```

サンプル SQL ファイル

```
create procedure do_migrate ()
begin
  declare @stmt long varchar;
  revoke consolidate from cons;
  set @stmt = 'create synchronization user ' || CURRENT
        PUBLISHER;
  execute immediate @stmt;
  set @stmt = 'CREATE SYNCHRONIZATION SUBSCRIPTION TO "p1" ';
  set @stmt = @stmt || 'FOR "' || CURRENT PUBLISHER || '" ';
  set @stmt = @stmt || 'TYPE ''TCPIP'' ';
  set @stmt = @stmt || 'ADDRESS ''host=10.2.3.1;port=600'' ';
  set @stmt = @stmt || 'option SV=''v1'' ';
  execute immediate @stmt;
end;
call do_migrate();
drop procedure do_migrate;
```

2.4 起こりうる問題

2.4.1 最後に送信された SQL Remote メッセージが失われる

前述のバッチ・ファイルと SQL ファイルにも、データが失われる可能性があります。移行プロセスの手順 1 で、最後の SQL Remote メッセージが統合データベースに送信され、出力ファイルの解析により SQL Remote の実行時にエラーが発生しなかったことが確認されますが、統合データベースにメッセージが正常に適用されたかどうかを確認する処理は何も実行されていません。何らかの理由でこれらのメッセージが失われるか統合データベースに適用されなかった場合、統合ユーザはすでにリモート・データベースから削除されているため、メッセージが再送されることはありません。そのため、メッセージが失われた場合、これらのメッセージに含まれる操作は統合データベースに適用されません。

この問題を回避するには、統合ユーザの SYS.SYSREMOTEUSER テーブル内の log_sent 値が confirmed_received 値と等しい場合にのみ、統合ユーザを削除します。この間に、SQL Remote からさらにメッセージが送信されると統合ユーザの log_sent 値が変更されてしまうため、リモート・データベースに対して変更が行われないようにしなければなりません。リモートで受信専用モードで SQL Remote を実行し、SQL Remote がメッセージを受信するたびに log_sent 値と confirmed_received を比較します。これらの値が等しければ、メッセージは統合データベースに適用されています。

この手法が本当に有効なのは、統合データベースの送信頻度が非常に高い場合のみです。統合データベースで SQL Remote を 1 日 1 回のみ実行するような状況では、統合データベースから確認メッセージが届くまで長い時間待つことになりかねません。

2.4.2 統合データベースで生成されるメッセージが増える

手順 1 で最後のメッセージを受信するためにリモート・データベースで SQL Remote を実行してから、手順 4 で初めて dbmlsync を実行するまでの間に、統合データベースで追加メッセージが生成される可能性があります。このプロセスはバッチ・ファイルで自動化される可能性が高いため、この問題が起きる可能性は非常に低いですが、統合データベースでの送信頻度が非常に高い場合は、その可能性がわずかに高くなります。このわずかな可能性に備え、統合データベースでの SQL Remote の送信頻度を高くすることで、この問題が生じる可能性をさらに低減することもできます。

また、統合データベース上で SQL Remote により生成される実際のメッセージは、メッセージ・システムにも残されます。まれに SQL Remote メッセージの内容を読み取る必要が生じた場合は、iAnywhere テクニカル・サポートを利用することもできます。

2.4.3 解決策

お気づきかも知れませんが、前述の 2 つの問題では、一方の問題の解決策により SQL Remote の送信頻度が高くなり、もう一方の問題の解決策により統合データベースでの SQL Remote の送信頻度が低くなります。明らかに、両方を同時に行うことはできません。

データが*失われない*ようにする最善の方法は、ユーザの新しいリモート・データベースを抽出することです。このドキュメントでは、すべてのリモート・データベースを再抽出しなくても済む方法を紹介します。ここで説明するプロセスはエンド・ユーザにほとんど影響を与えず、レプリケーション・システムの管理者に必要な作業もほとんどありませんが、データが失われる可能性がわずかにあります。

2.5 リモート・アプリケーションに対する変更

リモート・クライアント上でどのように SQL Remote を実行していたかによっては、dbremote を実行していたアプリケーションを、代わりに dbmlsync を実行するよう変更することが必要になる場合があります。

dbremote をサービスとして実行していた場合、dbremote の代わりに dbmlsync を実行するよう新しいサービスを定義し、これまで使用していた SQL Remote サービスを削除または無効にする必要があります。このプロセスは、dbsvc コマンド・ライン・ユーティリティを使用してバッチ・ファイルで自動化することができます。ユーザがバッチ・ファイルをクリックしてレプリケーションを開始していた場合は、SQL Remote の代わりに dbmlsync を実行するようバッチ・ファイルを変更する必要があります。

DBTOOLS API によって dbremote を呼び出していた場合、DBRemoteSQL() の代わりに DBSynchronizeLog() を呼び出すようコードを変更し、DBSynchronizeLog() に渡す新しい構造を定義する必要があります。

外部プロセス (dbremote など) を生成するアプリケーションを使用していた場合は、dbmlsync を生成するようアプリケーションのコードを変更する必要があります。

アプリケーションに対して変更を行う場合、Adaptive Server Anywhere バージョン 9.0.1 で導入された DBMLSync 統合コンポーネントを使用して同期プロセスを起動することができます。DBMLSync 統合コンポーネントは、Visual Basic、PowerBuilder、Delphi、または ActiveX コンポーネントを利用可能なその他のアプリケーション開発ツールにインポートすることのできる ActiveX プラグインです。ActiveX コンポーネントを .NET Compact Framework プログラミング環境にインポートできるサードパーティ製ツールもあります。DBMLSync 統合コンポーネントには、ビジュアル・バージョンと非ビジュアル・バージョンがあります。ビジュアル・コンポーネントでは、dbmlsync 実行プログラムに非常によく似たダイアログ・ボックスが表示され、このコンポーネントへの入力を設定することができます。非ビジュアル・コンポーネントでは、必要なプログラミング作業は増えますが、同期中にアップロードおよびダウンロードされるすべてのローにアクセスできるなど、カスタマイズの柔軟性が大幅に高まります。DBMLSync 統合コンポーネントの使用の詳細については、『*MobiLink Clients*』の「DBMLSync Integration Component」を参照してください。

3.0 統合データベースでの変更

以降の項では、統合データベースで必要な変更について説明します。

3.1 統合データベースの設定

%ASANY%¥MobiLink¥setup ディレクトリには、syncase125.sql という名前のファイルがあります。このファイルは、Mobile Link で使用する統合データベースに対して実行する必要があります。このスクリプトにより、アップロードおよびダウンロード・スクリプトを保管する Mobile Link システム・テーブルと、リモート・サイトの追跡進捗情報が作成されます。このスクリプトを実行する前に、スクリプトを変更し、適切なデータベースが変更されるよう先頭に use db_name コマンドを追加してください。通常、このスクリプトはマスタ・データベースに対してではなく、同期対象のテーブルが含まれるデータベースに対して実行します。このスクリプトを実行するときは、Mobile Link が同期の実行時に接続するときと同じユーザとして Adaptive Server Enterprise サーバに接続している必要があります。このスクリプトで作成される Mobile Link システム・テーブルなど、同期に関与するすべてのテーブルに対するパーミッションは、このユーザが選択、挿入、更新、および削除します。また、同期スクリプトからストアド・プロシージャを呼び出す場合は、このユーザにこれらのストアド・プロシージャの実行パーミッションが必要です。

Mobile Link は ODBC を使用して統合データベースに接続するため、Mobile Link が統合データベースに接続する際に使用する ODBC DSN を作成する必要があります。ただし、すべての ODBC ドライバが同じように作成されているわけではないため、DSN を作成する前に、iAnywhere Solutions の Web サイトの「SQL Anywhere Mobile Link の推奨 ODBC ドライバ」Web ページに目を通してください。このページのリンクは http://www.ianywhere.jp/tech/odbc_mobilink.html です。SQL Anywhere Studio のインストール・プロセスにより、サポートされている多くの統合データベースに対応した iAnywhere 提供の DataDirect OBDC ドライバがインストールされます。これらのドライバはリリース前にテストされているため、他の ODBC ドライバよりこれらのドライバを使用することをおすすめします。

3.2 互換性のあるアップロード・スクリプトの作成

SQL Remote では、生成され、適用されるメッセージに実際の SQL 文が含まれています。一方、Mobile Link クライアントが生成する統合データベースへのアップロード・ストリームには SQL 文が含まれていません。このストリームは、アップロードされるテーブルのリスト (およびその構造) と、これらのテーブルのローに対して行われた変更のリストで構成されています。更新がアップロードされる場合は、ローの変更前後のイメージがアップロードに含まれます。アップロードされる操作が挿入または削除の場合は、変更後または変更前のいずれかのイメージのみがアップロードに含まれます。これらのアップロード・ストリームで渡されるデータに対して実行するアクションを決定するためのアップロード・スクリプトを定義する必要があります。

3.2.1 アップロード・スクリプトの例

リモートおよび統合データベースの両方に以下のテーブル構造が存在するとします。

```
CREATE TABLE t1 (
  pkey INTEGER PRIMARY KEY,
  c1 INTEGER,
  c2 VARCHAR(100),
  c3 NUMERIC(10,2)
)
go
```

このテーブルには 4 つのカラムが含まれているため、アップロード・ストリームの各ローに対して、関連付けられたアップロード・ストリームが呼び出され、各カラムのデータが同期スクリプトに渡されます。スクリプトを作成する際は、このデータを疑問符で表します。これは ODBC や JDBC API と非常によく似ています。

Upload_Insert script:

```
insert into t1 (pkey,c1,c2,c3) values ( ?, ?, ?, ? );
```

Upload_Update script:

```
update t1 set c1 = ?, c2 = ?, c3 = ? where pkey = ?;
```

Upload_Delete script:

```
delete from t1 where pkey = ?;
```

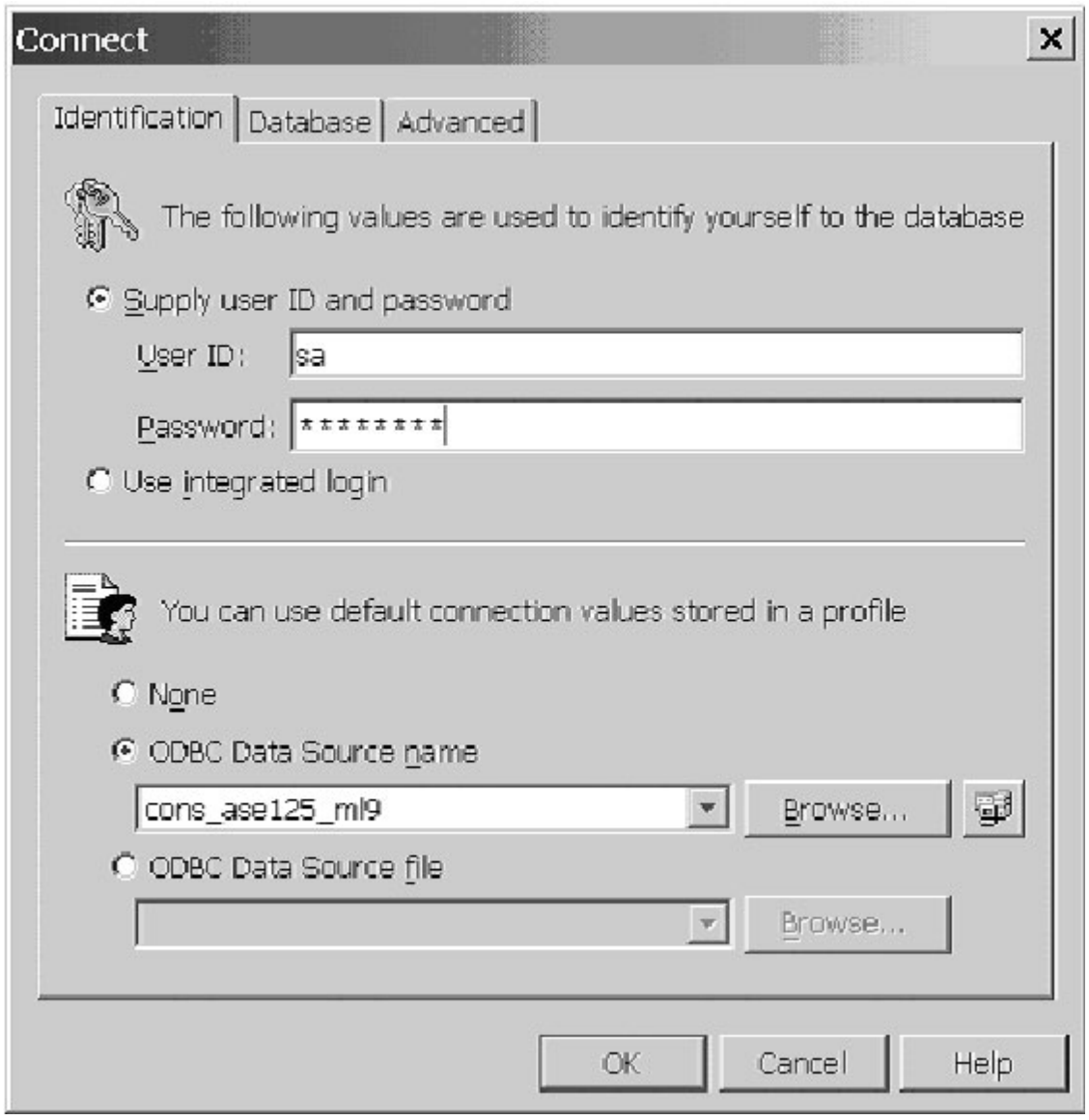
これらのスクリプトにおけるパラメータの順序に注意してください。upload_insert スクリプトでは、4 つのすべてのカラムがその存在順に渡されますが、upload_update スクリプトでは、データ・カラムの後にプライマリ・カラムが渡されます。このように、update 文では常にプライマリ・キー・カラムが最後に参照されるため、記述が容易です。upload_delete スクリプトではプライマリ・キー・カラムのみが渡され、データ・カラムは渡されません。

アップロード・スクリプトとして多くの状況で利用できるデフォルト・スクリプトを自動的に生成することができます。手順の詳細については、『Mobile Link 管理ガイド』の「スクリプトの自動生成」を参照してください。

3.2.2 同期スクリプトの定義

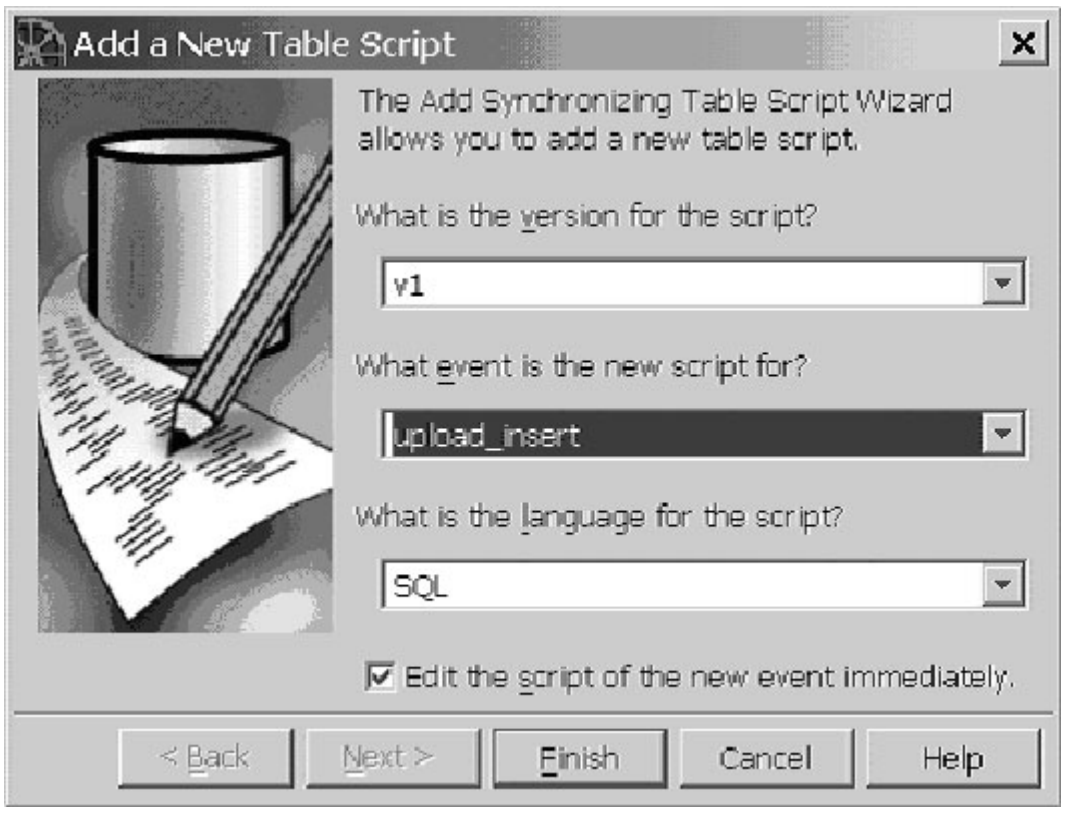
同期スクリプトは、Sybase Central を使用して定義することも、統合データベースに対して *syncase125.sql* スクリプトを実行したときに作成されるストアド・プロシージャを実行して追加することもできます。

Sybase Central を使用する場合、まず統合データベースに接続する必要があります。[Tools] ー [Connect] を選択し、次に Mobile Link Synchronization 9 プラグインを使用して接続するよう選択します。統合データベースへの接続用に作成した ODBC DSN を選択します。次に、DSN にユーザ ID とパスワードが指定されていない場合はオプションでこれらの値を指定し、[OK] をクリックします。

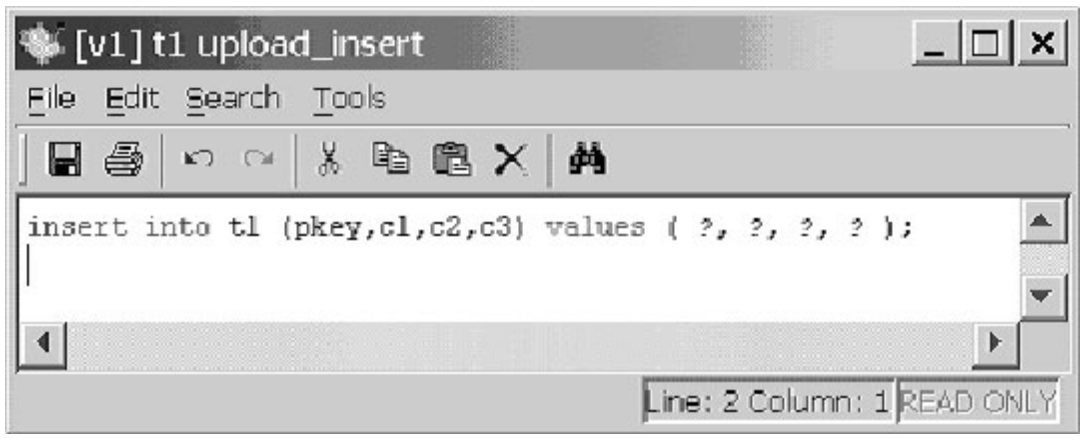


スクリプトは**スクリプト・バージョン**と呼ばれるグループに整理されます。Mobile Link クライアントは、特定のバージョンを指定することにより、どの同期スクリプト・セットを使用してアップロード・ストリームの処理とダウンロード・ストリームの準備を行うかを選択することができます。スクリプト・バージョンを統合データベースに追加するには、Versions フォルダを開き、Add Version ウィザードをダブルクリックして画面の指示に従い、新しいスクリプト・バージョンを定義します。

次の手順では、統合データベースにおいて同期に関与するすべてのテーブルのリストを作成します。Synchronized Tables フォルダを開いて右側フレームで Add Synchronized Table ウィザードをダブルクリックし、同期対象の各テーブルについてウィザードの指示に従います。テーブルがリストに追加されたら、左側ペインで Synchronized Tables フォルダ内のそのテーブル名をクリックし、右側ペインで Add Table Script ウィザードをクリックします。追加する新しいスクリプトのスクリプト・バージョン、イベント、および言語を選択し、スクリプトを今すぐ編集するためのオプションを選択してから、[Finish] ボタンをクリックします。



表示されるエディタ・ウィンドウでスクリプトの内容を入力して [File] ー [Close] を選択し、確認メッセージが表示されたらスクリプトを保存するよう選択します。



Sybase Central の代わりに、定義されている Mobile Link ストアド・プロシージャを使用する場合は、次に示す SQL により、t1 テーブルに対して 3 つの基本アップロード・イベントが定義されます。この SQL をテキスト・ファイルに保存して、ソース・コントロール・システムでチェックイン、維持することもできます。

```
exec ml_add_table_script 'v1', 't1', 'upload_insert',
  'insert into t1 (pkey,c1,c2,c3) values ( ?, ?, ?, ? )'
go
exec ml_add_table_script 'v1', 't1', 'upload_update',
  'update t1 set c1 = ?, c2 = ?, c3 = ? where pkey = ?'
go
exec ml_add_table_script 'v1', 't1', 'upload_delete',
  'delete from t1 where pkey = ?'
go
commit work
go
```

以降の項では、ストアド・プロシージャを使用して同期スクリプトを定義します。

3.3 互換性のあるダウンロード・スクリプトの作成

ダウンロード・スクリプトは、リモート・クライアントに送信するデータを決定するものです。特定テーブルに対するダウンロード・スクリプトとして最も単純なものは、すべてのローをすべてのリモート・クライアントに送信する SELECT 文のみで構成されます。

download_cursor script:

```
select pkey,c1,c2,c3 from t1
```

このスクリプトを定義するには、次のコードを実行します。

```
exec ml_add_table_script 'v1', 't1', 'download_cursor',
  'select pkey,c1,c2,c3 from t1'
go
commit work
go
```

上記の download_cursor には、同期のたびに、最後のダウンロード以降に変更されたローだけでなく、すべてのローがすべてのリモート・ユーザに送信されるという問題があります。移行の主な目標の 1 つとして、どの機能も失わないことを掲げましたが、上記の非常に単純なダウンロード・カーソルでは、リモートへの変更のみの送信という機能が失われてしまいます。

3.3.1 以前ダウンロードされたローの除外

リモート・データベースからの同期要求時のアップロード・ストリームでは、すでにダウンロードされたローの除外に役立つ 2 つの情報も渡されます。この情報とは、同期ユーザの名前と、このユーザが最後に正常にダウンロードを開始した日時です。この日時は LastDownload タイムスタンプと呼ばれ、リモートから渡されますが、統合データベースにより生成されます。この手法により、リモート・データベースと統合データベースのタイム・ゾーンが異なる場合や、クロックが統合データベースと同期されていない場合でも、同期を適切に機能させることができます。Mobile Link は、ダウンロード・ストリームを生成する直前に、統合データベースに日時を問い合わせ、リモートに渡すダウンロード・ストリームにこの日時を追加します。ダウンロード・ストリームがリモート・データベースに正常に適用されると、リモート上のシステム・テーブルが更新されます。リモートが次回同期するときには、新たな LastDownload タイムスタンプが Mobile Link に戻されます。

最後に正常に実行されたダウンロード日時を使用すると、各同期において変更されたローのみをリモートに送信することができます。これを行うには、各同期テーブルに datetime 型の last_modified カラムを追加し、同期のたびにローがダウンロードされないようにします。このカラムには、現在日時として定義されている DEFAULT 値を設定してください。

```
ALTER TABLE t1 ADD last_modified datetime default getdate() null
go
```

このカラムの初期値は、すべてのローについて '1900-01-01 00:01:00' に設定してください。その理由は後でわかります。また、ローが変更されるたびに last_modified カラムが更新されるようテーブルに更新トリガを追加する必要があります。

```
UPDATE t1 SET last_modified = '1900-01-01 00:01:00'
go
commit
go

CREATE TRIGGER au_t1 ON t1 FOR UPDATE
AS
BEGIN
UPDATE t1
SET last_modified = getdate()
WHERE pkey IN ( SELECT pkey FROM inserted )
END
go
```

これで、最後に正常に実行された同期以降に変更されたローのみをダウンロードするように、次のように download_cursor を修正することができます。

```
exec ml_add_table_script 'v1', 't1', 'download_cursor',
  'select pkey,c1,c2,c3 from t1 where last_modified >= ?'
go
commit work
go
```

last_modified カラムが統合データベースにのみ存在し、リモートに対して同期されないことに注意してください。このスキーマにより、変更が統合データベースで行われ、リモートでは行われません。ここで、統合データベースに定義されている、現在の SQL Remote ユーザに対するパブリケーションを変更することが必要になる場合があります。レプリケート対象がすべてのカラムではなくなったため、SQL Remote ユーザにレプリケートされるカラムのリストを含める必要があります (まだ行っていない場合)。

```
exec sp_add_article_col 'p1', 't1', 'pkey'
exec sp_add_article_col 'p1', 't1', 'c1'
exec sp_add_article_col 'p1', 't1', 'c2'
exec sp_add_article_col 'p1', 't1', 'c3'
go
```

3.3.2 シャドウ・テーブルによる以前ダウンロードされたローの除外

場合によっては、既存のアプリケーションによるデータベースへのアクセス方法が原因でベース・テーブルを変更できないことがあります。この場合、ベース・テーブルを変更する代わりに、テーブル内の各ローについて最後に変更された日時を保管するシャドウ・テーブルを追加することができます。

```
CREATE TABLE t1_st (
  pkey integer primary key,
  last_modified datetime default getdate()
)
go
```

このテーブルには、ベース・テーブルのローごとに 1 つのローを含め、last_modified 値を '1900-01-01 00:01:00' に設定してください。また、シャドウ・テーブルを使用する場合は、ベース・テーブルでの変更を追跡するために 3 つのトリガを定義する必要があります。

```
insert into t1_st select pkey, '1900-01-01 00:01:00' from t1
go
commit work
go

create trigger ai_t1 on t1 for insert as
begin
  insert into t1_st select pkey, getdate() from inserted
end
go

create trigger au_t1 on t1 for update as
begin
  update t1_st set last_modified = getdate()
  where pkey in ( select pkey from inserted )
end
go

create trigger ad_t1 on t1 for delete as
begin
  delete from t1_st where pkey in ( select pkey from deleted )
end
go
```

シャドウ・テーブルを使用するときは、download_cursor により 2 つのテーブルを結合し、以前ダウンロードされたローを除外する必要があります。

```
exec ml_add_table_script 'v1', 't1', 'download_cursor',
  'select t1.pkey,t1.c1,t1.c2,t1.c3 from t1,t1_st
    where t1.pkey = t1_st.pkey
      and t1_st.last_modified >= ?'
go
commit work
go
```

以降の項では、last_modified カラムがベース・テーブルに追加されているものとし、last_modified カラムのシャドウ・テーブルについてはこれ以上説明しません。

3.3.3 削除のダウンロード

Mobile Link は統合データベースのトランザクション・ログをスキャンせず、SQL 文に基づいてデータがダウンロードされるため、ダウンロード・ストリームで削除をリモート・データベースに送信するためには、別のメカニズムが必要になります。リモート・データベースから削除するローを追跡するため

に、download_delete_cursor という別のダウンロード・カーソルが使用されます。リモート・データベースで削除するローは、**シャドウ・テーブル**と呼ばれるテーブルによって追跡できます。これまでのサンプル・コードで使用してきたテーブル t1 を使用して、t1_del というテーブルを作成しました。

```
CREATE TABLE t1_del (
  pkey integer primary key,
  del_time datetime default getdate()
)
go
```

ここで、 t1 に対して、ローが削除されるたびに t1_del にローを挿入する削除トリガを定義します。このトリガでは、ローが削除された日時を追跡します。これにより、t1 に対する download_delete_cursor は次のようになります。

```
create trigger ad_t1 on t1 for delete as
begin
  insert into t1_del select pkey, getdate() from deleted
end
go

exec ml_add_table_script 'v1', 't1', 'download_delete_cursor',
   'select pkey from t1_del where del_time >= ?'
go
```

リモート・データベースで共有ローに対する削除を許可すると、SQL Remote と同様の参照整合性 (RI) の問題が発生する可能性が生じます。

たとえば、リモートでデータの更新および削除をユーザに許可した場合、特定のローに対する更新と削除をそれぞれ別々のユーザが同時に実行する可能性があります。最初に削除が適用された場合、更新が届いても、更新されるローは 0 個です。削除は更新を実行したリモートに即座に送信されるため、実際にはこれによってデータの不一致は生じませんが、更新に重要なビジネス情報が含まれていたとしても、その情報は簡単に失われてしまいます。

リモートで削除を許可することから生じるもう 1 つの問題は、外部キー・リレーションシップに含まれる親レコードの削除をリモート・ユーザに許可した場合に、さらに深刻なものになる可能性があります。たとえば、あるリモート (RemoteA) がすべての子レコードに続いて親レコードを削除し、これらの変更を統合データベースに同期したとします。その一方で、別のリモート・データベース (RemoteB) が、RemoteA により削除された親レコードを参照する子レコードを挿入したとします。ここで RemoteA が同期を行うと、子レコードと親レコードが統合データベースから削除されます。RemoteB が同期を行うと、子レコードの挿入により外部キー違反が発生します。

分散環境のリモート・サイトで削除を許可する場合は、十分な注意が必要です。

3.3.4 リモート・ユーザごとのローのパーティション化

SQL Remote を使用する場合、統合データベースのパブリケーション定義でカラムによるサブスクリプションを指定して、リモート・ユーザごとにローをパーティション化します。一方、Mobile Link を使用する場合は、download_cursor で同じカラムを使用してローをパーティション化します。ここで、t2

という新しいテーブルを定義します。このテーブルには、以前ダウンロードされたローを除外するための last_modified カラムと、同期しているリモート・データベースに基づいてローをパーティション化するための rem_user カラムが含まれています。

```
CREATE TABLE t2 (
  pkey INTEGER PRIMARY KEY,
  c1 INTEGER,
  c2 VARCHAR(100),
  c3 NUMERIC(10,2),
  last_modified datetime default getdate(),
  rem_user VARCHAR(128)
)
go

CREATE TRIGGER au_t2 ON t2 FOR UPDATE
AS
BEGIN
  UPDATE t2
  SET last_modified = getdate()
  WHERE pkey IN ( SELECT pkey FROM inserted )
END
go

sp_create_publication p1
sp_add_remote_table t2
sp_add_article p1, t2, NULL, rem_user
sp_add_article_col p1, t2, pkey
sp_add_article_col p1, t2, c1
sp_add_article_col p1, t2, c2
sp_add_article_col p1, t2, c3
sp_add_article_col p1, t2, rem_user
go
```

同期中、アップロード・ストリームで同期ユーザの名前が渡され、この値が download_cursor を含む多くの同期スクリプトに渡されます。ダウンロード・カーソルの WHERE 句で rem_user カラムを指定すれば、テーブル t2 に対して、パブリケーションの subscribe by 句と同等の機能を提供するダウンロード・カーソルを作成できます。

```
exec ml_add_table_script 'v1', 't2', 'download_cursor',
   'select pkey,c1,c2,c3,rem_user from t2
   where last_modified >= ?
   and rem_user = ?'
go
```

上記の download_cursor では、SQL Remote と同様に、ダウンロード以降に変更された、特定リモート・ユーザ用のローのみがリモート・データベースに同期されます。疑問符はパラメータの位置を表し、download_cursor 内の最初の疑問符は LastDownload タイムスタンプに置き換えられ、2 番目の疑問符は同期ユーザの名前に置き換えられます。Mobile Link User パラメータのみを使用する場合も、download_cursor に 2 つの疑問符

を含める必要があります。LastDownload タイムスタンプは null にできないため、? is not null を追加しても 2 つの疑問符を参照することができますが、select 文の結果は変更されません。

```
exec ml_add_table_script 'v1', 't2', 'download_cursor',
   'select pkey,c1,c2,c3 from t2 where ? is not null and rem_user = ?'
go
```

3.3.5 削除のダウンロードとリモート・ユーザごとのデータのパーティション化

特定のテーブルについて異なるデータが各リモート・サイトで保持され、削除をリモート・ユーザにダウンロードする場合は、削除のダウンロード・シャドウ・テーブルにより、削除されたローをどのリモート・ユーザが受け取るかを指定することも必要です。ここでは、これまでのサンプル・コードで使用してきた t2 テーブルを引き続き使用しますが、多少異なったシャドウ・テーブルと削除トリガを定義する必要があります。

```
CREATE TABLE t2_del (
  pkey integer primary key,
  rem_user varchar(128) NOT NULL,
  del_time datetime default getdate()
)
go
```

t2 テーブルに対する削除トリガでは、削除の送信先リモート・データベースの名前を挿入する必要があります。

```
create trigger ad_t2 on t2 for delete as
begin
  insert into t2_del select pkey, rem_user, getdate() from
       deleted
end
```

ここで、テーブル t2 の rem_user カラムが変更される状況を考慮する必要があります。この状況が発生した場合、新しい rem_user 値に関連付けられたリモート・データベースにこのローの挿入を送信し、古い rem_user 値に関連付けられたリモート・データベースにこのローの削除を送信する必要があります。この変更により、更新トリガが起動し、テーブル t2 に last_modified カラムが設定されるため、既存の download_cursor を使用してこのローの挿入が新しいリモート・データベースにダウンロードされます。ただし、そのローの削除は、このローの以前の所有者であるリモート・ユーザに送信されるようにします。これを行うために、ローの以前の所有者であるリモート・ユーザ用のテーブル t2_del にローが挿入されるようテーブル t2 に対する更新トリガを変更します。

```
CREATE TRIGGER au_t2 ON t2 FOR UPDATE
AS
  IF UPDATE ( rem_user )
    BEGIN
      INSERT INTO t2_del
        SELECT deleted.pkey, deleted.rem_user, getdate()
          FROM deleted, inserted
         WHERE deleted.pkey = inserted.pkey
           AND deleted.rem_user <> inserted.rem_user
```

```
    END
  UPDATE t2
  SET last_modified = getdate()
  WHERE pkey IN ( SELECT pkey FROM inserted )
go
```

最後に、download_delete_cursor でも、削除が正しいユーザにのみ送信されるようにする必要があります。

```
exec ml_add_table_script 'v1', 't2', 'download_delete_cursor',
   'select pkey from t2_del where del_time >= ? and rem_user = ?'
go
```

3.3.6 サブスクリプション・カラムを含まないテーブルのパーティション化

多くの場合、テーブルにサブスクリプション・カラムが存在しなくても、テーブルのローをパーティション化する必要があります。この解決策でも、テリトリ再アラインメント問題と呼ばれる問題を考慮する必要があります。これは、別テーブルでの変更の結果として、サブスクリプション・カラムを含まないローをリモート間で移行する必要があるというものです。この状況、および SQL Remote for ASE を使用している場合のその解決策については、Adaptive Server Anywhere のマニュアルに詳しい説明があります。この概念に馴染みがない方は、先に進む前に『*SQL Remote ユーザーズ・ガイド*』の「サブスクリプション式を含まないテーブルの分割」をお読みください。

3.3.6.1 Contact サンプルのスキーマ

この項では、まずマニュアルに掲載されているサンプル (Contact サンプル) を、Mobile Link が SQL Remote for ASE とまったく同じ機能を実行できるよう変更します。SQL Remote for Adaptive Server Enterprise サンプルで使用されているスキーマを次に示します。その後に、Mobile Link で使用するためにこのスキーマをどのように変更する必要があるかを説明します。

```
--
-- Create the Tables
--
CREATE TABLE SalesRep (
   rep_key CHAR(12) NOT NULL,
   name CHAR(40) NOT NULL,
   PRIMARY KEY (rep_key)
)
go

CREATE TABLE Customer (
   cust_key CHAR(12) NOT NULL,
   name CHAR(40) NOT NULL,
   rep_key CHAR(12) NOT NULL,
   FOREIGN KEY ( rep_key )
   REFERENCES SalesRep,
   PRIMARY KEY (cust_key)
)
```

```
go

CREATE TABLE Contact (
   contact_key CHAR( 12 ) NOT NULL,
   name CHAR( 40 ) NOT NULL,
   cust_key CHAR( 12 ) NOT NULL,
   subscription_list CHAR( 12 ) NULL,
   FOREIGN KEY ( cust_key )
   REFERENCES Customer ( cust_key ),
   PRIMARY KEY ( contact_key )
)
go


--
-- Define the SQL Remote Definitions
--
exec sp_add_remote_table 'SalesRep'
exec sp_add_remote_table 'Customer'
exec sp_add_remote_table 'Contact'
go
exec sp_create_publication 'SalesRepData'
go
exec sp_add_article 'SalesRepData', 'SalesRep'
  exec sp_add_article 'SalesRepData', 'Customer', NULL, 'rep_key'
  exec sp_add_article 'SalesRepData',
                  'Contact', NULL, 'subscription_list'
  go

  --
  -- Set the subscription_list column when a new row is added
  -- to the Contact table
  --
  CREATE TRIGGER set_contact_sub_list
  ON Contact
  FOR INSERT
  AS
  BEGIN
     UPDATE Contact
     SET Contact.subscription_list = (
          SELECT rep_key
          FROM Customer
          WHERE Contact.cust_key = Customer.cust_key )
     WHERE Contact.contact_key IN ( SELECT contact_key FROM
          inserted )
  END
go
```

```
  --
  -- Maintain the subscription_list column on the Contact table
  -- when the cust_key column is modified
  --
  CREATE TRIGGER update_contact_sub_list
  ON Contact
  FOR UPDATE
  AS
  IF UPDATE ( cust_key )
     BEGIN
       UPDATE Contact
       SET subscription_list = Customer.rep_key
       FROM Contact, Customer
       WHERE Contact.cust_key = Customer.cust_key
     END
go

  --
  -- Maintain the subscription_list column on the Contact table
  -- when the rep_key column on the Customer table is changed
  --
  CREATE TRIGGER transfer_contact_with_customer
  ON Customer
  FOR UPDATE
  AS
  IF UPDATE ( rep_key )
  BEGIN
     UPDATE Contact
     SET Contact.subscription_list = (
        SELECT rep_key
        FROM Customer
        WHERE Contact.cust_key = Customer.cust_key )
     WHERE Contact.cust_key IN ( SELECT cust_key FROM inserted )
  END
  go
```

3.3.6.2 Mobile Link を使用するためのスキーマの変更

この項では、前項のスキーマを変更して各テーブルに last_modified カラムを含め、削除のダウンロードを処理するシャドウ・テーブルを追加し、アップロードおよびダウンロードを処理する Mobile Link スクリプトを作成します。

```
  CREATE TABLE SalesRep_del (
    rep_key CHAR( 12 ) PRIMARY KEY,
    del_time DATETIME DEFAULT getdate()
  )
```

```
go

ALTER TABLE SalesRep ADD last_modified datetime default
       getdate() null
go
UPDATE SalesRep SET last_modified = '1900-01-01 00:01:00'
go
commit
go

CREATE TRIGGER ad_SalesRep on SalesRep for delete as
BEGIN
  INSERT INTO SalesRep_del SELECT rep_key, getdate() FROM
       deleted
END
go

CREATE TRIGGER au_SalesRep ON SalesRep FOR UPDATE
AS
BEGIN
  UPDATE SalesRep
  SET last_modified = getdate()
  WHERE rep_key IN ( SELECT rep_key FROM inserted )
END
go
exec sp_add_article_col 'SalesRepData', 'SalesRep', 'rep_key'
exec sp_add_article_col 'SalesRepData', 'SalesRep', 'name'
go
exec ml_add_table_script 'v1', 'SalesRep', 'upload_insert',
  'insert into SalesRep values ( ?, ? ) '
go
exec ml_add_table_script 'v1', 'SalesRep', 'upload_update',
  'update SalesRep set name = ? where rep_key = ? '
go
exec ml_add_table_script 'v1', 'SalesRep', 'upload_delete',
  'delete from SalesRep where rep_key = ? '
go
exec ml_add_table_script 'v1', 'SalesRep', 'download_cursor',
  'select rep_key, name from SalesRep where last_modified >= ? '
go
exec ml_add_table_script 'v1', 'SalesRep', 'download_delete_
       cursor',
  'select rep_key from SalesRep_del where del_time >= ? '
go
commit work
go
```

```
CREATE TABLE Customer_del (
  cust_key CHAR( 12 ) PRIMARY KEY,
  rep_key CHAR(12) NOT NULL,
  del_time DATETIME DEFAULT getdate()
)
go

ALTER TABLE Customer ADD last_modified datetime default
       getdate() null
go
UPDATE Customer SET last_modified = '1900-01-01 00:01:00'
go
commit
go
CREATE TRIGGER ad_Customer ON Customer FOR DELETE AS
BEGIN
  INSERT INTO Customer_del
    SELECT cust_key, rep_key, getdate() FROM deleted
END
go

DROP TRIGGER transfer_contact_with_customer
go
CREATE TRIGGER transfer_contact_with_customer
ON Customer
FOR UPDATE
AS
  IF UPDATE ( rep_key )
  BEGIN
    UPDATE Contact
    SET Contact.subscription_list = (
      SELECT rep_key
      FROM Customer
      WHERE Contact.cust_key = Customer.cust_key )
    WHERE Contact.cust_key IN ( SELECT cust_key FROM inserted )
    INSERT INTO Customer_del
      SELECT deleted.cust_key, deleted.rep_key, getdate()
        FROM deleted, inserted
       WHERE deleted.cust_key = inserted.cust_key
         AND deleted.rep_key <> inserted.rep_key
  END
  UPDATE Customer
  SET last_modified = getdate()
  WHERE cust_key IN ( SELECT cust_key FROM inserted )
go
exec sp_add_article_col 'SalesRepData', 'Customer', 'cust_key'
exec sp_add_article_col 'SalesRepData', 'Customer', 'name'
```

```
exec sp_add_article_col 'SalesRepData', 'Customer', 'rep_key'
go
exec ml_add_table_script 'v1', 'Customer', 'upload_insert',
  'insert into Customer (cust_key,name,rep_key) values ( ?, ?, ? )'
go
exec ml_add_table_script 'v1', 'Customer', 'upload_update',
  'update Customer set name = ?, rep_key = ? where cust_key = ?'
go
exec ml_add_table_script 'v1', 'Customer', 'upload_delete',
  'delete from Customer where cust_key = ? '
go
exec ml_add_table_script 'v1', 'Customer', 'download_cursor',
  'select cust_key, name, rep_key from Customer
   where last_modified >= ? and rep_key = ?'
go
exec ml_add_table_script 'v1', 'Customer', 'download_delete_
       cursor',
  'select cust_key from Customer_del
   where del_time >= ? and rep_key = ?'
go
commit work
go
```

```
CREATE TABLE Contact_del (
  contact_key CHAR( 12 ) PRIMARY KEY,
  subscription_list CHAR(12) NOT NULL,
  del_time DATETIME DEFAULT getdate()
)
go
ALTER TABLE Contact ADD last_modified datetime default getdate()
       null
go
UPDATE Contact SET last_modified = '1900-01-01 00:01:00'
go
commit
go

CREATE TRIGGER ad_Contact ON Contact FOR DELETE AS
BEGIN
  INSERT INTO Contact_del
    SELECT contact_key, subscription_list, getdate() FROM
        deleted
END
go
DROP TRIGGER update_contact_sub_list
go
```

```
CREATE TRIGGER update_contact_sub_list
ON Contact
FOR UPDATE
AS
  IF UPDATE ( cust_key )
     BEGIN
       UPDATE Contact
       SET subscription_list = Customer.rep_key
       FROM Contact, Customer
       WHERE Contact.cust_key = Customer.cust_key
       INSERT INTO Contact_del
         SELECT deleted.contact_key,
              deleted.subscription_list,
              getdate()
          FROM deleted, Customer c_del, inserted, Customer c_ins
         WHERE deleted.cust_key = c_del.cust_key
           AND inserted.cust_key = c_ins.cust_key
           AND deleted.contact_key = inserted.contact_key
           AND deleted.cust_key <> inserted.cust_key
           AND c_del.rep_key <> c_ins.rep_key
     END
  UPDATE Contact
  SET last_modified = getdate()
   WHERE contact_key IN ( SELECT contact_key FROM inserted )
go

exec sp_add_article_col 'SalesRepData', 'Contact', 'contact_key'
exec sp_add_article_col 'SalesRepData', 'Contact', 'name'
exec sp_add_article_col 'SalesRepData', 'Contact', 'cust_key'
exec sp_add_article_col 'SalesRepData', 'Contact',
       'subscription_list'
go

exec ml_add_table_script 'v1', 'Contact', 'upload_insert',
  'insert into Contact (contact_key, name, cust_key,
       subscription_list)
   values ( ?, ?, ?, ? ) '
go

exec ml_add_table_script 'v1', 'Contact', 'upload_update',
  'update Contact set name = ?, cust_key = ?, subscription_list = ?
   where contact_key = ? '
go

exec ml_add_table_script 'v1', 'Contact', 'upload_delete',
  'delete from Contact where contact_key = ? '
go
```

```
exec ml_add_table_script 'v1', 'Contact', 'download_cursor',
  'select contact_key, name, cust_key, subscription_list from
       Contact
   where last_modified >= ? and subscription_list = ?'
go

exec ml_add_table_script 'v1', 'Contact', 'download_delete_
       cursor',
'select contact_key from Contact_del
 where del_time >= ? and subscription_list = ?'
go
commit work
go
```

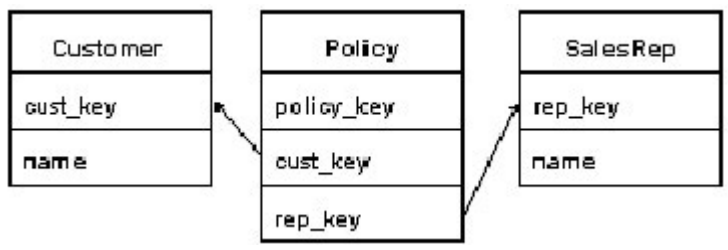
この項で扱う内容は、Contact テーブルに対する更新トリガと、Customer テーブルで rep_key が更新されても Contact_del テーブルにエントリが追加されないことを除き、これまでの項ですべて説明済みです。

Contact テーブルに対する更新トリガでは、cust_key テーブルが変更された結果、そのローの所有者でなくなったリモートに削除を送信する必要があります。ここで重要なのは、cust_key が変更された場合でも、削除されたテーブルの subscription_list カラムの値が削除を受け取るべきであるとは断定できないことです。2 名の顧客を 1 名の営業担当者が担当しており、これらの 2 名の顧客間で移動するよう連絡先が変更された場合、削除を送信すべきでは*ありません*。この変更によって、連絡先がある rep_key から別の rep_key へと実際に移動されることを確認する必要があります。その理由から、Customer テーブルを削除されたテーブルに、また Customer テーブルの別インスタンスを挿入されたテーブルに結合し、2 つのインスタンスの Customer テーブルの rep_key 値を比較して本当に削除を送信すべきかどうかを確認する必要があります。

もう 1 つの未解決の問題は、Customer テーブルの rep_key カラムが変更された場合に、Contact テーブルのリモートにどのように削除を送信するかです。Customer テーブルに対する更新トリガにより、Customer_del テーブルに古い rep_key 値のためのローを追加できますが、Contact_del テーブルには値が追加されません。これは、dbmlsync (および Ultra Light) がリモート上で密かに RI 違反を処理するためです。Customer テーブルに対する削除がリモートで行われ、ダウンロード・ストリーム適用の最後のコミット時に RI 違反が発生しそうになると、Contact テーブルのローが dbmlsync により削除されます。

3.3.7 複数のサブスクリプションによるローの共有

この項では、リモートにあるローが複数のリモート・データベースに存在する可能性がある状況について説明します。前項と同様に、必ずしもテーブルにリモート・ユーザ名が存在するわけではありませんが、2 つのテーブル間に多対多の関係があるときに、どのローがどのリモートに属しているかを管理する手段が必要です。このドキュメントでは、Policy を使用し、各リモート・データベースに送信する Customer テーブル内のローを判断するサンプルについて説明します。



Policy テーブルには、各ポリシーに対するローが含まれています。各ポリシーは特定の営業担当者によって特定顧客用に作成されたものです。顧客と営業担当者の間には多対多の関係があり、特定の販売担当者／顧客のペア間に複数のポリシーが作成されている可能性があります。Customer テーブルのどのローも 0 名、1 名、または複数名の営業担当者と共有することが必要になる可能性があります。この問題を SQL Remote for ASE で解決するためのスキーマを次の項に示します。その後で、このスキーマを Mobile Link で動作するよう変更する方法について説明します。

3.3.7.1 Policy サンプルのスキーマ

このドキュメントで扱うサンプルでは、Policy テーブルの rep_key カラムで更新が発生する可能性が考慮されていないことに注意してください。

```
--
-- Create Tables
--
CREATE TABLE SalesRep (
   rep_key CHAR( 12 ) NOT NULL,
   name CHAR( 40 ) NOT NULL,
   PRIMARY KEY ( rep_key )
)
go

CREATE TABLE Customer (
   cust_key CHAR( 12 ) NOT NULL,
   name CHAR( 40 ) NOT NULL,
   subscription_list VARCHAR( 255 ) NULL,
   PRIMARY KEY ( cust_key )
)
go

CREATE TABLE Policy (
   policy_key INTEGER NOT NULL,
   cust_key CHAR( 12 ) NOT NULL,
   rep_key CHAR( 12 ) NOT NULL,
   FOREIGN KEY ( cust_key ) REFERENCES Customer ( cust_key ),
   FOREIGN KEY ( rep_key ) REFERENCES SalesRep ( rep_key ),
   PRIMARY KEY ( policy_key )
)
go
--
-- Define the SQL Remote Definitions
--
exec sp_add_remote_table 'SalesRep'
```

```
exec sp_add_remote_table 'Policy'
exec sp_add_remote_table 'Customer'
go

exec sp_create_publication 'SalesRepData'
exec sp_add_article 'SalesRepData', 'SalesRep'
exec sp_add_article 'SalesRepData', 'Policy', NULL, 'rep_key'
exec sp_add_article 'SalesRepData', 'Customer', NULL,
       'subscription_list'
exec sp_add_article_col 'SalesRepData', 'Customer', 'cust_key'
exec sp_add_article_col 'SalesRepData', 'Customer', 'name'
go
--
-- This stored procedure is used to maintain the subscription_list
-- column in the Customer Table
--
CREATE PROCEDURE SubscribeCustomer
  @cust_key CHAR(12)
AS
BEGIN
   -- Rep returns the rep list for customer @cust_key
   DECLARE Rep CURSOR FOR
     SELECT DISTINCT RTRIM( rep_key )
     FROM Policy
     WHERE cust_key = @cust_key
DECLARE @rep_key CHAR(12)
DECLARE @subscription_list VARCHAR(255)
  -- build comma-separated list of rep_key
-- values for this Customer
OPEN Rep
FETCH Rep INTO @rep_key
IF @@sqlstatus = 0 BEGIN
  SELECT @subscription_list = @rep_key
  WHILE 1=1 BEGIN
     FETCH Rep INTO @rep_key
     IF @@sqlstatus != 0 BREAK
     SELECT @subscription_list =
        @subscription_list + ',' + @rep_key
  END
END
ELSE BEGIN
  SELECT @subscription_list = ''
END
-- update the subscription_list in the
-- Customer table
UPDATE Customer
SET subscription_list = @subscription_list
```

```
 WHERE cust_key = @cust_key
END
go

--
-- Maintain the subscription_list column on the Customer
-- Customer table when a new Policy is created
--
CREATE TRIGGER InsPolicy
ON Policy
FOR INSERT
AS
BEGIN
   -- Cust returns those customers inserted
   DECLARE Cust CURSOR FOR
      SELECT DISTINCT cust_key
      FROM inserted
   DECLARE @cust_key CHAR(12)
   OPEN Cust
   -- Update the rep list for each Customer
   -- with a new rep
   WHILE 1=1 BEGIN
      FETCH Cust INTO @cust_key
      IF @@sqlstatus != 0 BREAK
      EXEC SubscribeCustomer @cust_key
   END
END
go
--
-- Maintain the subscription_list column on the Customer
-- Customer table when a Policy is deleted
--
CREATE TRIGGER DelPolicy
ON Policy
FOR DELETE
AS
BEGIN
   -- Cust returns those customers deleted
   DECLARE Cust CURSOR FOR
      SELECT DISTINCT cust_key
      FROM deleted
   DECLARE @cust_key CHAR(12)
   OPEN Cust
   -- Update the rep list for each Customer
   -- losing a rep
   WHILE 1=1 BEGIN
      FETCH Cust INTO @cust_key
```

```
      IF @@sqlstatus != 0 BREAK
      EXEC SubscribeCustomer @cust_key
   END
END
go
```

3.3.7.2 Mobile Link を使用するためのスキーマの変更

このサンプルについても、説明を単純にするために、Policy テーブルの cust_key または rep_key カラムは更新されないものと仮定します。理論上、これは Policy テーブルに対する更新トリガが必要ないことを意味しますが、ここでは完全を期して更新トリガが記述されています。

```
CREATE TABLE SalesRep_del (
   rep_key CHAR( 12 ) PRIMARY KEY,
   del_time DATETIME DEFAULT getdate()
)
go
ALTER TABLE SalesRep ADD last_modified datetime default
       getdate() null
go
UPDATE SalesRep SET last_modified = '1900-01-01 00:01:00'
go
commit
go
```

```
CREATE TRIGGER ad_SalesRep on SalesRep for delete as
BEGIN
  INSERT INTO SalesRep_del SELECT rep_key, getdate() FROM
       deleted
  END
go
CREATE TRIGGER au_SalesRep ON SalesRep FOR UPDATE
AS
BEGIN
  UPDATE SalesRep
  SET last_modified = getdate()
  WHERE rep_key IN ( SELECT rep_key FROM inserted )
END
go
exec sp_add_article_col 'SalesRepData', 'SalesRep', 'rep_key'
exec sp_add_article_col 'SalesRepData', 'SalesRep', 'name'
go
exec ml_add_table_script 'v1', 'SalesRep', 'upload_insert',
  'insert into SalesRep values ( ?, ? ) '
go
exec ml_add_table_script 'v1', 'SalesRep', 'upload_update',
  'update SalesRep set name = ? where rep_key = ? '
```

```
go
exec ml_add_table_script 'v1', 'SalesRep', 'upload_delete',
  'delete from SalesRep where rep_key = ? '
go
exec ml_add_table_script 'v1', 'SalesRep', 'download_cursor',
  'select rep_key, name from SalesRep where last_modified >= ?'
go
exec ml_add_table_script 'v1', 'SalesRep', 'download_delete_cursor',
   'select rep_key from SalesRep_del where del_time >= ? '
go
commit work
go

CREATE TABLE Customer_del (
  cust_key CHAR( 12 ) NOT NULL,
  rep_key CHAR( 12 ) NOT NULL,
  del_time DATETIME DEFAULT getdate(),
  PRIMARY KEY ( cust_key, rep_key, del_time )
)
go
ALTER TABLE Customer ADD last_modified datetime default
       getdate() null
go
UPDATE Customer SET last_modified = '1900-01-01 00:01:00'
go
commit
go

CREATE TRIGGER au_Customer ON Customer FOR UPDATE
AS
  UPDATE Customer
  SET last_modified = getdate()
  WHERE cust_key IN ( SELECT cust_key FROM inserted )
go

exec ml_add_table_script 'v1', 'Customer', 'upload_insert',
  'insert into Customer (cust_key,name) values ( ?, ? ) '
go
exec ml_add_table_script 'v1', 'Customer', 'upload_update',
  'update Customer set name = ? where cust_key = ? '
go
exec ml_add_table_script 'v1', 'Customer', 'upload_delete',
  'delete from Customer where cust_key = ? '
go
exec ml_add_table_script 'v1', 'Customer', 'download_cursor',
  'select cust_key, name from Customer
   where last_modified >= ? and charindex( ?, subscription_list
```

```
       ) > 0'
go
exec ml_add_table_script 'v1', 'Customer', 'download_delete_
       cursor',
   'select cust_key from Customer_del
    where del_time >= ? and rep_key = ?'
go
commit work
go

CREATE TABLE Policy_del (
  policy_key INTEGER PRIMARY KEY,
  rep_key CHAR( 12 ) NOT NULL,
  del_time DATETIME DEFAULT getdate()
)
go
ALTER TABLE Policy ADD last_modified datetime default getdate()
       null
go
UPDATE Policy SET last_modified = '1900-01-01 00:01:00'
go
exec sp_add_article_col 'SalesRepData', 'Policy', 'policy_key'
exec sp_add_article_col 'SalesRepData', 'Policy', 'cust_key'
exec sp_add_article_col 'SalesRepData', 'Policy', 'rep_key'
go
COMMIT
go
CREATE TRIGGER au_Policy ON Policy FOR UPDATE
AS
  UPDATE Policy
  SET last_modified = getdate()
  WHERE policy_key IN ( SELECT policy_key FROM inserted )
go
DROP TRIGGER DelPolicy
go

CREATE TRIGGER DelPolicy
ON Policy
FOR DELETE
AS
BEGIN
   -- Cust returns those customers deleted
   DECLARE Cust CURSOR FOR
      SELECT DISTINCT cust_key
      FROM deleted
   DECLARE CustRep CURSOR FOR
      SELECT cust_key, rep_key
```

```
   FROM deleted
DECLARE @cust_key CHAR(12)
DECLARE @rep_key CHAR(12)
-- Add a row to Policy_del table
INSERT INTO Policy_del
   SELECT policy_key, rep_key, getdate() FROM deleted
-- See if we need to add any rows to the Customer_del table
OPEN CustRep
WHILE 1=1
BEGIN
   FETCH CustRep INTO @cust_key, @rep_key
   IF @@sqlstatus != 0 BREAK
   -- See if we just deleted the last policy that link this
   -- customer to this sales rep
   IF NOT EXISTS ( SELECT 1 FROM Policy
                   WHERE cust_key = @cust_key
                     AND rep_key = @rep_key )
   BEGIN
      INSERT INTO Customer_del
      VALUES ( @cust_key, @rep_key, getdate() )
   END
END
OPEN Cust
   -- Update the rep list for each Customer
   -- losing a rep
   WHILE 1=1 BEGIN
      FETCH Cust INTO @cust_key
      IF @@sqlstatus != 0 BREAK
      EXEC SubscribeCustomer @cust_key
   END
END
go

exec ml_add_table_script 'v1', 'Policy', 'upload_insert',
   'insert into Policy (policy_key,cust_key,rep_key) values ( ?,
        ?, ? )'
go
exec ml_add_table_script 'v1', 'Policy', 'upload_update',
   'update Policy set cust_key = ?, rep_key = ?
    where policy_key = ? and 1=0'
go
exec ml_add_table_script 'v1', 'Policy', 'upload_delete',
   'delete from Policy where policy_key = ? '
go
exec ml_add_table_script 'v1', 'Policy', 'download_cursor',
   'select policy_key, cust_key, rep_key from Policy
    where last_modified >= ? and rep_key = ?'
```

```
go
exec ml_add_table_script 'v1', 'Policy', 'download_delete_
       cursor',
   'select policy_key from Policy_del
    where del_time >= ? and rep_key = ?'
go
commit work
go

DROP PROCEDURE SubscribeCustomer
go
CREATE PROCEDURE SubscribeCustomer
  @cust_key CHAR(12)
AS
BEGIN
   -- Rep returns the rep list for customer @cust_key
   DECLARE Rep CURSOR FOR
      SELECT DISTINCT RTRIM( rep_key )
      FROM Policy
      WHERE cust_key = @cust_key
   DECLARE @rep_key CHAR(12)
   DECLARE @subscription_list VARCHAR(255)
   -- build comma-separated list of rep_key
   -- values for this Customer
   OPEN Rep
   FETCH Rep INTO @rep_key
   IF @@sqlstatus = 0 BEGIN
      SELECT @subscription_list = @rep_key
      WHILE 1=1 BEGIN
         FETCH Rep INTO @rep_key
         IF @@sqlstatus != 0 BREAK
         SELECT @subscription_list =
            @subscription_list + ',' + @rep_key
         END
      END
   ELSE BEGIN
      SELECT @subscription_list = ''
  END

  -- update the subscription_list and last_modified
  -- columns in the Customer table
  UPDATE Customer
  SET subscription_list = @subscription_list,
     last_modified = getdate()
  WHERE cust_key = @cust_key
END
go
```

この項で示したコードについてもそのほとんどが説明済みですが、ここでいくつかの点についてさらに説明しておきましょう。

customer_del テーブルには、cust_key、rep_key、および del_time カラムを含む複合プライマリ・キーがあります。Customer テーブルの同じローに対する削除が 2 つのリモート・データベースに送信される可能性があったことから、cust_key と rep_key の両方のカラムが必要になります。また、営業担当者が最後の顧客を失い、新しいポリシーの結果としてその顧客を再度獲得し、その後その顧客を再度失う可能性があったことから、この問題を解決するためにプライマリ・キーに del_time カラムも必要です。

Customer_del テーブルにローを追加するための、Customer テーブルに対する削除トリガはありません。削除しようとしている顧客を参照するローが Policy テーブルに存在するかぎり、Customer テーブルでの削除は RI 違反で失敗します。Customer テーブル内のローを削除できるのは、そのローを参照するローが Policy テーブルに存在しない場合のみです。Policy の最後のローが削除され、これによって営業担当者と顧客間のリレーションシップが失われた場合、Policy テーブルに対する削除トリガによって Customer_del テーブルにローが追加されます。統合データベースで Customer テーブルのローを削除できるということは、そのローがすべてのリモート・サイトで削除されているということです。

Policy テーブルに対する挿入トリガは何も変更されていませんが、Policy テーブルに対する挿入トリガと削除トリガで呼び出される SubscribeCustomer ストアド・プロシージャは多少変更されています。Customer テーブルの last_modified カラムは subscription_list と同時に変更されるため、subscription_list に追加された新しい rep_key は自動的に download_cursor によって適用されます。

Policy テーブルに対する削除トリガが変更され、削除されたテーブル内のローを循環する別のカーソルが追加されています。Policy テーブル内の削除された各ローについて、特定顧客のポリシーを失った営業担当者に Customer テーブルに対する削除が必要かどうかを確認してください。この営業担当者がこの顧客に対して多数のポリシーを持っている可能性もあります。多数あるポリシーの 1 つが削除されても、顧客の削除にはならない場合もあります。削除は、営業担当者と顧客を関連付ける最後のポリシーが削除された場合のみ送信されます。

Policy テーブルの upload_update カーソルには、ローの更新を回避するために WHERE 1=0 句が含まれています。Mobile Link はスクリプトが存在しないためにデータが失われる可能性があることを訴えるため、このイベントを単純に削除することはできません。ここに示した単純なサンプルでは、このテーブルに対する更新は許可されていないため、リモートから送信される可能性のあるすべての更新 (これについても許可されていない) は無視されます。ご使用のアプリケーションがリレーションシップ・テーブル内の更新可能な追加カラムを使用し、rep_key および cust_key カラムの変更が許可されていない場合、アップロード・スクリプトでもこれらのルールに従うことをおすすめします。rep_key および cust_key 値をパラメータとして使用するストアド・プロシージャを作成することもできますが、このストアド・プロシージャ自体は統合データベース上のこれらの値を更新しません。理論上、リモートにあるアプリケーションによりこれらの更新が許可されていない場合、その値が変更されることはありませんが、起こり得ないことではあっても十分な注意を払い、その悪影響を防ぐに越したことはありません。

3.3.8 競合解決スクリプト

SQL Remote for ASE 用の競合解決スクリプトを定義するには、レプリケーション・テーブルのリストにテーブルを追加するときに、resolve_procedure、old_row、および remote_row テーブルも指定します。old_row テーブルには統合データベースにおける更新前の値、remote_row テーブルにはリモート・データベースから送信された以前の値が保管され、このテーブル自体はリモートから送信された新しい値で更新されています。次のコードは、これまでのサンプル・コードで使用してきた t1 テーブルを使用する単純なサンプルです。このサンプルでは、競合時に優先されるロー、すなわち競合時の優先度を決定するカラムの値が最も大きいローを判断するために、c3 カラムが使用されています。

```
CREATE TABLE t1 (
  pkey INTEGER PRIMARY KEY,
  c1 INTEGER,
  c2 VARCHAR(100),
 c3 NUMERIC(10,2),
 last_modified DATETIME DEFAULT getdate()
)
go

CREATE TABLE t1_old (
  pkey INTEGER PRIMARY KEY,
  c1 INTEGER,
  c2 VARCHAR(100),
  c3 NUMERIC(10,2),
 last_modified DATETIME
)
go

CREATE TABLE t1_remote (
  pkey INTEGER PRIMARY KEY,
  c1 INTEGER,
  c2 VARCHAR(100),
  c3 NUMERIC(10,2),
  last_modified DATETIME
)
go

CREATE PROCEDURE sp_resolve_t1
AS
BEGIN
  DECLARE @lost_c3 NUMERIC(10,2)
  DECLARE @won_c3 NUMERIC(10,2)
  DECLARE @rem_c3 NUMERIC(10,2)
  DECLARE @lost_c2 VARCHAR(100)
  DECLARE @won_c2 VARCHAR(100)
  DECLARE @rem_c2 VARCHAR(100)
  DECLARE @lost_c1 INTEGER
  DECLARE @won_c1 INTEGER
  DECLARE @rem_c1 INTEGER
  DECLARE @pkey INTEGER
  SELECT @lost_c1=c1, @lost_c2=c2, @lost_c3=c3, @pkey=pkey FROM
        t1_old
  SELECT @won_c1=c1, @won_c2=c2, @won_c3=c3 FROM t1 WHERE
        pkey=@pkey
  SELECT @rem_c1=c1, @rem_c2=c2, @rem_c3=c3 FROM t1_remote
  IF @lost_c3 > @rem_c3
    BEGIN
```

```
      IF @lost_c3 > @won_c3
        BEGIN
          UPDATE t1 SET c1=@lost_c1, c2=@lost_c2, c3=@lost_c3
           WHERE pkey=@pkey
        END
    END
    ELSE
      BEGIN
        IF @rem_c3 > @won_c3
          BEGIN
            UPDATE t1 SET c1=@rem_c1, c2=@rem_c2, c3=@rem_c3
             WHERE pkey=@pkey
          END
      END
END
go

exec sp_create_publication p1
exec sp_add_remote_table 't1', 'sp_resolve_t1', 't1_old', 't1_remote'
exec sp_add_article 'p1', 't1'
exec sp_add_article_col 'p1', 't1', 'pkey'
exec sp_add_article_col 'p1', 't1', 'c1'
exec sp_add_article_col 'p1', 't1', 'c2'
exec sp_add_article_col 'p1', 't1', 'c3'
go
commit work
go
```

Mobile Link で実行される競合解決プロセスは、SQL Remote for ASE の競合解決プロセスと非常によく似ています。Mobile Link を使用する場合、更新しようとしているローの現在の状態と、アップロード・ストリームに保管されているローの更新前イメージを比較する upload_fetch カーソルを定義します。これらの 2 つの値が同じでない場合、競合が発生しようとしており、次の 3 つのイベントが起動します (定義されている場合)。

- **upload_new_row_insert** :このイベントには、リモート・データベースの新しいロー値が渡されます。
- **upload_old_row_insert** :このイベントには、リモート・データベースの古いロー値が渡されます。
- **resolve_conflict** :このイベントは、競合の原因となる各ローに対して呼び出されます。

Mobile Link と SQL Remote for ASE における競合解決の実質的な違いは、Mobile Link の競合解決では、更新がまだ行われていない状態で、各プロセスで 3 つの同じローにアクセスして競合の解決方法を決定できることです。優先されるローを決定するロジックはすでに作成されているため、面倒な作業はほとんど終わっています。理想としては、同じテーブルとストアド・プロシージャを再利用するところですが、SQL Remote for ASE により、同時に競合解決を行うスレッドを 1 つに限定するためにチェックインされているため、SQL Remote for ASE と同時に Mobile Link で競合解決を行おうとすると問題が発生します。Mobile Link の競合解決を実装するコードを次に示します。新しいテーブルが作成され、新しいストアド・プロシージャが使用されていますが、ロジックはすべて同じであるため、ほとんどの作業がカットアンドペースト操作です。

```
CREATE PROCEDURE sp_resolve_ml_t1
AS
BEGIN
  DECLARE @cur_c3 NUMERIC(10,2)
  DECLARE @new_c3 NUMERIC(10,2)
  DECLARE @old_c3 NUMERIC(10,2)
  DECLARE @cur_c2 VARCHAR(100)
  DECLARE @new_c2 VARCHAR(100)
  DECLARE @old_c2 VARCHAR(100)
  DECLARE @cur_c1 INTEGER
  DECLARE @new_c1 INTEGER
  DECLARE @old_c1 INTEGER
  DECLARE @pkey INTEGER

  SELECT @new_c1=c1, @new_c2=c2, @new_c3=c3, @pkey=pkey FROM
        #t1_ml_new
  SELECT @old_c1=c1, @old_c2=c2, @old_c3=c3 FROM #t1_ml_old
    WHERE pkey=@pkey
  SELECT @cur_c1=c1, @cur_c2=c2, @cur_c3=c3 FROM t1 WHERE
        pkey=@pkey
  IF @new_c3 > @old_c3
    BEGIN
      IF @new_c3 > @cur_c3
        BEGIN
          UPDATE t1 SET c1=@new_c1, c2=@new_c2, c3=@new_c3
           WHERE pkey=@pkey
        END
    END
  ELSE
    BEGIN
      IF @old_c3 > @cur_c3
        BEGIN
          UPDATE t1 SET c1=@old_c1, c2=@old_c2, c3=@old_c3
           WHERE pkey=@pkey
        END
    END
  DELETE FROM #t1_ml_new
  DELETE FROM #t1_ml_old
END
Go

exec ml_add_connection_script 'v1', 'begin_connection',
  'CREATE TABLE #t1_ml_new (
    pkey INTEGER PRIMARY KEY,
    c1 INTEGER,
    c2 VARCHAR(100),
    c3 NUMERIC(10,2)
```

```
  )
  CREATE TABLE #t1_ml_old (
    pkey INTEGER PRIMARY KEY,
    c1 INTEGER,
    c2 VARCHAR(100),
    c3 NUMERIC(10,2)
  )'
go

exec ml_add_table_script 'v1', 't1', 'upload_fetch',
  'select pkey,c1,c2,c3 from t1 where pkey = ?'
go
exec ml_add_table_script 'v1', 't1', 'upload_new_row_insert',
  'insert into #t1_ml_new values ( ?, ?, ?, ? )'
go
exec ml_add_table_script 'v1', 't1', 'upload_old_row_insert',
  'insert into #t1_ml_old values ( ?, ?, ?, ? )'
go
exec ml_add_table_script 'v1', 't1', 'resolve_conflict',
  'exec sp_resolve_ml_t1'
go
```

3.4 移行されたリモートからの初めての同期

リモートが初めて同期するときの LastDownload タイムスタンプは '1900-01-01 00:00:00' です。空のリモート・データベースを展開した場合、そのデータベースが初めて同期を行うと、1900 年 1 月 1 日以降に変更されたすべてのデータがリモートにダウンロードされます。これは、リモート・データベースにすべてのデータを取り込むための効果的な方法です。ここでは、すでにデータが取り込まれた状態のリモート・データベースを使用しているため、dbmlsync の初回実行時に Mobile Link による全データのダウンロードは不要です。

リモート・データベースの移行についての説明では、SQL Remote によってすでにダウンロードされたすべてのデータが Mobile Link からの次のダウンロードに含まれないようにするために、初回同期時に統合データベースに対して何が行われているかについては、詳しく説明しませんでした。Mobile Link リモート・データベースが初めて同期するときに渡す last_download の値は '1900-01-01 00:00:00' です。この last_download 値から、これが最初の同期であることがわかり、この同期中にいくつかの追加手順を踏むことができます。

すべての追加手順は modify_last_download_timestamp イベントで行われます。まず、last_download の値が '1900-01-01 00:00:00' かどうかを確認します。この値である場合は、渡された ml_username が sr_remoteuser テーブルに存在するユーザ名であるかどうかを確認します。これに該当する場合は、このユーザの sr_remoteuser テーブルから time_sent カラムの値を取得します。これは、このユーザに最後に SQL Remote メッセージが送信された日時を表します。渡された last_download 値 ('1900-01-01 00:00:00') を sr_remoteuser テーブルの日時に変更し、SQL Remote ユーザにこれ以上 SQL Remote メッセージが送信されないよう sr_remoteuser テーブルからこのユーザを削除します。このコードは次のようになります。

```
CREATE PROCEDURE sp_mldt
  @ldts DATETIME OUTPUT,
```

```
  @ml_user VARCHAR(128)
AS
BEGIN
  IF @ldts = '1900-01-01 00:00:00'
  BEGIN
    IF EXISTS ( SELECT 1 FROM sr_remoteuser
              WHERE user_name(user_id) = @ml_user )
    BEGIN
      SELECT @ldts=time_sent
        FROM sr_remoteuser
       WHERE user_id = user_id(@ml_user)
DELETE FROM sr_subscription WHERE user_id = user_id( @ml_user )
DELETE FROM sr_remoteoption WHERE user_id = user_id( @ml_user )
     DELETE FROM sr_remoteuser WHERE user_id = user_id( @ml_user )
    END -- if exist
  END -- if @ldts
END -- end proc

exec ml_add_connection_script 'v1', 'modify_last_download_timestamp',
  '{call sp_mldt ?, ?}'
go
```

リモート・サイトに展開する新しいデータベースを SQL Remote ではなく Mobile Link で同期する場合は、Mobile Link ユーザの新しいリモート・データベースが正常に同期できることを確認することも重要です。新たな Mobile Link リモート・データベースと SQL Remote データベースの大きな違いは、新しい Mobile Link リモート・データベースにはデータが存在しないことです。渡される last_download タイムスタンプが '1900-01-01 00:00:00' であるため、すべてのデータが同期されます。最近移行された SQL Remote データベースか、新たな Mobile Link データベースかは、sr_remoteuser テーブルにローが存在するかどうかを調べることで見分けることができます。ローが存在しない場合は、modify_last_download_timestamp イベントで last_download タイムスタンプ値を変更しないでください。同期の残りの処理では、値 '1900-01-01 00:00:00' が使用されます。

既存のすべてのテーブルについて新しい last_modified カラムの初期値が '1900-01-01 00:01:00' に設定されたため、初回同期時にすべてのローが新しい Mobile Link クライアントにダウンロードされます。値が '1900-01-01 00:01:00' である理由は、この値が '1900-01-01 00:00:00' より大きく、sr_remoteuser テーブルの time_sent の最小値より確実に小さいためです。

4.0 まとめ

ある製品を別の製品へと移行する作業は、新しい製品を単にインストールするほど容易ではありません。新しい製品の構成が必要になり、また新しい製品を構成するために行った変更によって問題が生じないよう移行元の製品を構成することが必要になる場合もあります。SQL Remote for ASE から Mobile Link への移行はまさにそのケースですが、それによって得られるメリットはデメリットをはるかに超えています。このドキュメントでは、第 1.3 項で掲げた移行目標を念頭に置いて、SQL Remote for ASE 環境から Mobile Link 環境への移行を成功させるために必要な手順を示してきました。管理者が統合データベースで行わなければならない作業はありますが、この移行中、リモート・サイトへの影響はほとんどありません。

さらに支援が必要な場合も、多数のオプションが用意されています。iAnywhere プロフェッショナル・サービスが長年に渡る経験を活かし、お客様に応じたサービスを通して移行を成功へと導きます。iAnywhere プロフェッショナル・サービスでは、次のようなサービスを提供しています。

- レプリケーション・ルールの解析と検討
- 現在の SQL Remote for ASE ソリューションと同等またはそれを上回る Mobile Link ソリューションの設計
- Mobile Link スクリプトの開発とテスト
- カスタマイズされた知識移転プロセスの提供
- Mobile Link 同期トレーニングの実施

法的注意