同期と複製

Replication and Synchronization

4D Developer Conference Tokyo 2012

目次

SQL 版

概要

4D v12 では，新しい SQL コマンドでデータベースの同期と複製ができるようになりました。SQL 版の API を使用すれば，4D v12 以降のデータベースは，コマンドひとつでテーブルの同期を取ることができます。このコマンドが登場する以前，データベースの同期は大仕事でした。デベロッパは，レコード・ロック，レコードの更新，衝突の解決など，アプリケーション層をすべて管理することに加え，Web Services，4D Internet Commands，4D Open，あるいは定期的なレコードの書き出しと読み込みなど，通信層とトランスポート・プロトコルを実装しなければなりませんでした。SQL 版の同期と複製は，データをネットワーク経由で交換するコーディングの煩わしさを除去するだけではありません。データの管理がエンジンレベルで処理されることが一番の強みです。このセクションでは，SQL 版のテーブル同期と複製について説明します。

同期とミラーリング

4D 2004 には，ログファイルの分割と統合をコーディングで実行することにより，論理ミラーサーバーを構築することができました。具体的には，完全に同一のデータベースを別々のマシンで実行し，一方（マスター）のデータベースは，定期的に New log file コマンドを実行することにより，適当なサイズのログファイルを作成します。バックアップは永久に実行しません。（ログファイルがバックアップの役目を果たします。）作成されたログファイルは，何らかの手段でもうひとつのサーバー（ミラー）に転送し，そこで INTEGRATE LOG FILE コマンドを実行することにより，データを同期させることができました。

この場合，ミラーサーバーは，完全に従属の立場であり，一切のデータ更新が許されませんでした。また，データの妥当性を検証するのは，マスターサーバーの仕事であり，ミラーサーバーは，与えられたログファイルの内容を単純に実行するだけの「無能な」アプリケーションでした。マスターサーバーが使用できなくなった場合，それまで控えていたミラーサーバーがマスターサーバーとなり，そのときはじめて「機能的な」アプリケーションになります。しかし，次回のメンテナンスでミラーサーバーの内容がマスターに転写され，マスターサーバーが完全に復帰した時点で，ミラーサーバーはふたたび控えの役目に徹することになります。

SQL 版の同期は，あるデータベースから別のデータベースにレコードがコピーされるという意味では，ミラーリングに似ていますが，どちらのデータベースも機能的なアプリケーションであり，自由にレコードを読み書きできる点がミラーリングとは違います。つまり，「マスター」に対して受動的で主体性のない「ミラー」という関係性ではないということです。当然，バックアップも自由に実行することができ，マスターとミラーの立場を是正するために再起動をする必要もありません。また，SQL 版の同期は，テーブル毎，フィールド毎のレコードコピーであり，ミラーリングのようにデータベース全体が同一である必要はありません。

配列とフィールド

SQL 版の同期は，フィールドと配列，ふたつの用法が存在します。前者は，レコードの値とスタンプ（後述）に基づき，自動的にデータベースを更新します。このとき，リレーションやトリガなどの論理的な制約は実行されません。データの妥当性は，供給側が検証したものと判断できるからです。後者は，レコードの値とスタンプを配列に返すだけであり，自動的にデータベースが更新されることはありません。これは，デベロッパが完全に制御できるという面では優れていますが，データベースエンジンレベルで実装されている SQL 同期の魅力の大部分を失うことになります。

複製と同期

あるデータベースから別のデータベースにレコードをコピーすることが複製だとすれば，同期は続けて反対方向の複製を実行することに相当します。同期は，二度，複製を実行することであり，両者は基本的に

同じオペレーションです。この記事では，コマンドに関する説明などを除き，特に区別することなく，同期（あるいは複製）という用語を使用しています。なお，複製（REPLICATE）とは違い，同期（SYNCHRONIZE）コマンドは，配列にデータを受け取るという用法はありません。配列の使用は，最終的に複製を実行するべきかどうかをプログラムで制御することがおもな目的ですが，一部のレコードを複製から除外することはそもそも同期の目的に反するので，同期では常にフィールドにデータを受け取ります。また，配列を使用するかどうか関わりなく，同期の代わりに複製を両方向に実行することはできません。複製を実行すれば，一方のデータベースはさらに更新されたことになり，再度，複製が必要となって，無限ループに陥るからです。

ローカルとリモート

このセッションでは，「ローカル」および「リモート」という用語を頻繁に使用します。いずれも，古くから 4D プログラミングで使用されている用語ですが，同期と複製の文脈では，通常とは意味が異なることに留意してください。

クライアント・サーバー型のアプリケーションであれば，サーバー（データベース）側がリモート，接続クライアント（ユーザー）側がローカルでした。同期と複製では，そのように述べることはできません。同期と複製の文脈では，SQL コマンドの REPLICATE あるいは SYNCHRONIZE を発行した側がローカル，その相手がリモートです。データを供給するのはリモート側なので，別の見方をするならば，コマンドを発行した側がデスティネーション，その相手がソースである，ということもできます。あるいは，コマンドを発行した側がスレーブ，その相手がマスターということになります。

コマンドを発行する側がローカルなのですから，同期と複製をプログラミングする場合，実際にデベロッパーが制御するのは，一貫してローカル/デスティネーション/スレーブのほうです。リモート/ソース/マスターは，最初に複製を有効にすることを除けば，一切，制御することがありません。リモート側の動作は，完全に自動的です。フィールド版の用法では，ローカル側も，エラー処理などの例外を除けば，基本的に全自動です。配列版の用法，および HTTP 版の API（後述）のみ，ローカル側のデータベースの管理がデベロッパの手に委ねられることになります。

セットアップ

ストラクチャーエディター

同期を実行するためには，ストラクチャーエディターでテーブルプロパティ「複製を有効にする」を選択する必要があります。このプロパティは，デフォルトで無効になっており，テーブルにプライマリーキーが設定されていなければ，選択することができません。同期は，プライマリーキーに基づいてレコードを特定するため，プライマリーキーの設定されていないテーブルは，同期することができないからです。値の重複しないフィールドであれば，どのフィールドでもプライマリーキーにすることができます。あるいは，複数のフィールドを組み合わせてプライマリーキーにすることもできます。しかし，推奨されているのは，UUID フィールドを新設し，そのフィールドをプライマリーキーにすることです。

UUID タイプのフィールドを追加するためには，文字列タイプのフィールドを追加し，プロパティリストで UUID プロパティを選択します。このように設定されたフィールドは，内部的には整数タイプの UUID ですが，ランゲージでは文字列（16 進数表現に変換された UUID）でアクセスできるようになります。正しい方法で生成された UUID は，本質的に重複不可なので，プライマリーキーに向いています。また，さまざまな言語やシステムでサポートされているという面でも便利です。

プライマリーキーを設定するためには，ひとつ以上のフィールドを選択し，コンテキストメニューの「主キーを作成」を選択します。既存のフィールドで，重複する値が存在する場合，主キーに設定することはできません。逆に，主キーに設定できれば，「重複不可」のプロパティを敢えて有効にする必要はありません。別のフィールドでプライマリーキーを作成すれば，以前のものは破棄されます。プライマリーキーを取り除くには，フィールドではなく，テーブルを選択して，コンテキストメニューの「主キーを取り除く」を選択します。

プライマリーキーが設定されたテーブルは，「複製を有効にする」プロパティが選択できます。複製が有効にされている間，そのテーブルは，別のフィールドをプライマリーキーにすることができません。プライマリーキー以外のフィールドは，引き続きタイプを変更したり，削除したりすることができます。

SQL

プライマリーキーの作成，UUID フィールドを追加すること，および複製を有効することは，いずれも SQL コマンドで実行できます。

```
ALTER TABLE テーブル名 ADD フィールド名 UUID AUTO_GENERATE
ALTER TABLE テーブル名 ADD PRIMARY KEY (フィールド名)
ALTER TABLE テーブル名 ENABLE REPLICATE
```

シンクヘッダーファイルとシンクデータファイル

複製が有効にされたデータベースは，ストラクチャファイルと同じディレクトリに 4DSyncHeader と 4DSyncData という拡張子のファイルが作成されます。これらのファイルには，複製を実行するために必要であり，デフォルトの設定でバックアップの対象にも含まれています。

これらのファイルを削除することは，複製をリセットすることになりません。ただ，以後の複製を破綻させることになるだけです。実際，複製が有効にされたテーブルがひとつでも残されているのであれば，いずれのファイルも削除するべきではありません。ファイルを安全に削除できるのは，複製が有効にされたテーブルがひとつもない場合だけです。複製のスタンプ（後述）をリセットするのであれば，ストラクチャエディターあるいは SQL コマンドで複製を無効にした後，再度，有効にしてください。

レコードのタッチ

複製が有効にされたテーブルは，以後，レコードの作成，更新，削除が実行されるたびにスタンプがインクリメントされ，SQL で同期ができるようになります。ただし，既存のレコードは，スタンプが存在しないので，複製の対象にはなりません。もし，既存のレコードを複製に含めたいのであれば，何らかの方法で既存のレコードにもスタンプが追加されるようにすることが必要です。たとえば，APPLY TO SELECTION で現在の値を代入すれば，レコードに「タッチ」し，スタンプを追加することができます。

メカニズム

バーチャルフィールド

複製が有効にされたテーブルには，SQL コマンドだけでアクセスできるバーチャルフィールドが存在します。バーチャルフィールドは，同期を実行するために必要な情報を内部的に管理するためのものなので，（配列版ではない用法で）同期を複製を実行する上でその内容を特に意識する必要はありません。しかし，バーチャルフィールドの役割を知ることは，同期と複製のメカニズムを理解する上で有用です。また，配列版の API では，バーチャルフィールドの内容が重要な位置を占めています。

```
__ROW_ID【Int32】
```

4D コマンドの Record number と同じ，つまりローカルデータベースのレコード番号です。同期と複製では，プライマリーキーでレコードを特定するので，レコード番号は更新されるべきレコードを判断するための決定的な情報ではありません。実際，削除されたレコードの番号は再利用されるので，プライマリーキーが違う（したがって同期では別物とみなされるべき）レコードがかつて存在した別のレコードと同じ番号を付され

ることがあり得ます。レコード番号は，レコードを特定するものというよりは，レコードの現在位置（場所）を特定する情報であるということです。

なお，_ROW_ID バーチャルフィールドは，複製と同期以外の SQL コマンド（SELECT など）でも参照することができ，SQL コマンドと 4D コマンドの橋渡しができる，という面でも便利です。当然，_ROW_ID に値を書き込むことはできません。

__ROW_STAMP【Int64】

同期では，レコードが更新された順序が重要であり，それを管理するために使用されているのがスタンプです。スタンプは，1 から始まってインクリメントされる 64 ビット整数のシーケンシャル番号です。テーブルのシーケンシャルと同じように，番号がひとつずつ増えてゆくとは限りませんが，最近に発行されたものほどおおきな番号が発行されることは保証されています。

スタンプは，複製が有効にされたテーブル毎に存在し，そのテーブルのレコードが更新（作成）または削除されるたびに，値がインクリメントされてゆきます。スタンプは，更新（作成）または削除されたレコードのプライマリーキーとともに保存され，その両者は「いつ」「何が」更新されたのかを知らせるものです。テーブルのスタンプは，最後に発行されたスタンプであり，それは最後に実行された更新または削除に対応しています。レコードのスタンプは，レコードに対するアクションの発生した順番を示すものであり，特定のアクションが他のアクションよりも前あるいは後に実行されたのかを示しています。

スタンプは，タイムスタンプではありません。つまり，特定のアクションが「いつ」発生したという情報ではないということです。また，更新の履歴でもありません。スタンプは，レコードが更新されるたびにインクリメントされてゆきますが，テーブル毎のスタンプは，そのテーブルで発生した最後のアクションに対応するものだけであり，レコード毎のスタンプは，そのレコードで発生した最後のアクションに対応するものだけなので，特定のレコードが現在の状態に至った経緯がスタンプで追跡できるわけではありません。

100 個のレコードを 1 回，更新した場合，100 個のスタンプが発行され，テーブルのスタンプは，少なくとも 100 番，インクリメントされることになります。1 個のレコードを 100 回，更新した場合，テーブルのスタンプは同じだけインクリメントされますが，レコードのスタンプは，最後の 1 個を除き，すべて後続のアクションに上書きされるので，そのレコードに対して 1 個だけ存在することになります。

バーチャルフィールドなので，値の書き込みはできませんが，スタンプの値はどの SQL コマンドでも参照することができます。スタンプの値は，64 ビット整数の範囲内で際限なく上昇する仕様です。これをリセットする必要は，通常，発生しないはずですが，一旦，複製を無効にしてから再度，有効にすれば，スタンプはふたたび 1 からカウントされるようになります。

__ROW_ACTION【Int16】

スタンプに対応するアクションのタイプを示す情報であり，1（更新）または 2（削除）いずれかの値を取ります。同期と複製では，作成（追加）と更新（変更）を区別する必要がないため，どちらも 1（更新）というアクションタイプに分類されます。このバーチャルフィールドにアクセスできるのは，同期あるいは複製のコマンドだけです。とりわけ，配列版の用法では，アクションの値に基づき，実際にローカルのデータベースを更新することになります。フィールド版の用法では，ローカルとリモート，それぞれのデータベースエンジンが自動的に適切なアクションを実行します。

同期と制約

複製および同期でテーブルが更新されるとき，そのテーブルに設定された制約は無視されます。つまり，トリガ，削除制御，自動インクリメント，自動 UUID，重複不可，ヌル値不可などのプロパティは無視され，直接，データベースが更新されるということです。これは，無茶に思えるかもしれませんが，実際のところ，当然のことです。複製は，状態をコピーする行為であり，その状態に至った経緯を再現する行為ではありません。複製されたレコードが論理的に無効なものだとしても，それは複製が制約は無視したためにそうなった

のではなく，論理的に無効なレコードが供給されたことが原因です。制約など，データの妥当性を検証するためのロジックは，同期を使用する場合，双方のシステムに実装されているべきです。また，複製はテーブル単位のオペーレーションであるため，プライマリーキーよりも先に外部キーが作成されることもあり得るわけで，この段階で制約を適用することには無理があります。そのため，複製はレコードを複製することだけに専念し，制約を適用することは，それぞれのアプリケーションに委ねられています。

通常，レコードの妥当性はリモート側で検証されており，複製後，それをローカル側で改めて検証する必要はないはずです。複製後，リモート側が独自に妥当性を検証する必要があるとすれば，それは制約，つまりデータベースのロジックがリモート側と同じではない場合です。そのようなケースでは，むしろ複製に配列を使用したほうが合理的かもしれません。配列の基づいてローカルデータベースを更新するときには，通常どおり制約やトリガが有効だからです。

最後に，制約が無効になるのは，同期と複製により実行されるデータベースの更新だけです。同期を実行中であっても，そのテーブルに対するアクセス自体は許されており，他のプロセスによるデータベースの更新は，引き続き制約が有効であることに留意してください。

同期とトランザクション

同期のリモート側，つまり SYNCHRONIZE コマンドを発行したほうではないデータベースは，トランザクションの中で同期を実行します。一方，ローカル側，つまり SYNCHRONIZE コマンドを発行したほうのデータベースは，データベース設定の「自動コミットトランザクション」あるいは明示的なトランザクションを開始しない限り，トランザクションの中で同期を実行しません。

ページネーションされた同期を実行している途中，リモート側で同期中のレコードが更新された場合，それまでの更新はキャンセルし，同じスタンプで同期をやり直す必要があります。そのようなわけで，同期を実行するローカル側のデータベースも，トランザクションを開始することが勧められています。また，リレーションが設定された複数のテーブルを同期している途中，ローカル側で同期中のレコードが更新された場合，データベースの整合性が危機にさらされるので，トランクションを開始するだけでなく，セレクションをロックすることが大切です。4D コマンドの SET QUERY AND LOCK，あるいは SQL コマンドの SELECT FOR UPDATE でこれを実行することができます。

SQL

SQL 版の同期は，4D データベースエンジンの間で，直接，実行されるものです。つまり，命令は，4D コマンドではなく，SQL コマンドで記述し，接続は，ODBC ではなく，SQL パススルーで確立する必要があります。また，SYNCHONIZE および REPLICATE コマンドが使用できるのは，Begin SQL/End SQL の中だけです。文字列で命令文を構築するのであれば，SQL EXECUTE ではなく，EXECUTE IMMEDIATE で実行しなければなりません。

参考:さまざまな SQL の記述スタイル

Begin SQL/End SQL の中に SQL 文を記述

Begin SQL/End SQL の中で EXECUTE IMMEDIATE を実行

SQL EXECUTE コマンド

SQL EXECUTE SCRIPT コマンド

1 番目のスタイルでは，メソッドに直接，SQL 文を記述します。SELECT 文の WHERE 句や INTO 句など，テーブル名・コマンド名・関数名以外の箇所には，コロン(:)記号に続けて 4D オブジェクト(変数，配列，ポインタ)をバインドすることができます。ローカル変数もバインドできます。さらに，ODBC 接続でなければ，INTO LISBOX や SYNCHRONIZE など，4D 特有の SQL コマンドが使用できます。もっともシンプルかつ

強力なスタイルですが，パラメータ化された部分以外の SQL 文は固定であり，OFFSET や LIMIT でリクエストを分割しない限り，結果セットは常に全体が返されます。

2 番目のスタイルでは，文字列で構築された SQL 文をコマンドに渡すので，条件付けされた，ダイナミックな SQL 文を実行することができます。コロン(:)記号に続けて 4D オブジェクトをバインドすることができますが，コンパイルモードでは，ローカル変数を使用することができません。OFFSET や LIMIT でリクエストを分割しない限り，結果セットは常に全体が返されます。ローカル変数の制約にさえ注意を払えば，もっとも汎用的なスタイルです。

3 番目のスタイルも，文字列で構築された SQL 文をコマンドに渡すので，条件付けされた，ダイナミックな SQL 文を実行することができます。4D オブジェクトは，クエスチョンマーク(?)と SQL SET PARAMETER コマンドでバインドすることができます。ローカル変数も使用できます。Begin SQL/End SQL とは違い，結果セットは一定サイズに分けて取り出すことができます。4D 特有の SQL コマンドは使用できませんが，ODBC 接続であれば，もっとも汎用的かつ強力なスタイルです。

4 番目のスタイルでは，外部に標準テキストで保存された SQL スクリプトを実行します。ローカルデータベース(内部あるいはエクスターナルデータベース)が対象であり，ODBC 接続あるいは 4D 同士のパススルーで接続されたリモートデータベースとの対話には使用できません。おもに，インポート・エクスポートなどのバッチ処理において威力を発揮します。

いずれしても，SQL 版の同期は，Begin SQL/End SQL の構文で実行する必要があります。SQL の統一的なログインコマンドは SQL LOGIN ですが，プロトコルに「4D:」あるいは「IP:」を指定し，アスタリスクを渡すことによって，4D 同士のダイレクト接続(パススルー)を確立し，かつそのセッションに Begin SQL/End SQL の命令を適用することができます。詳細は，SQL LOGIN コマンドのドキュメントを参照してください。

REPLICATE

シンタックス

一番シンプルな複製の構文は，下記の形を取ります。

```
REPLICATE フィールド名, フィールド名,...
FROM テーブル名
FOR REMOTE STAMP :in リモートスタンプ,
LATEST REMOTE STAMP :out リモートスタンプ,
INTO テーブル名 (フィールド名, フィールド名,...);
```

REPLICATE に続くフィールド名は，リモート/ソース/マスター側で有効なものでなければなりません。そのテーブルに存在するフィールドの一部だけを複製することもできます。また，フィールドを列挙する順序は不同であっても構いません。

FROM に続くテーブル名は，リモート/ソース/マスター側で有効なものでなければなりません。

FOR REMOTE STAMP に続く番号は，同期されるレコードを絞り込むためのフィルターです。始めての同期であれば，通常，1(最小のスタンプ)を渡し，複製が有効にされた後に実行されたすべての更新(アクション)をリクエストします。複製に成功すれば，LATEST REMOTE STAMP に最新のスタンプが返され，この値が次回の同期で FOR REMOTE STAMP に渡すべき番号となります。したがって，ローカル/デスティネーション/スレーブ側は，この数字を次回の同期まで保持することが必要です。

INTO に続くテーブル名およびフィールド名は，ローカル/デスティネーション/スレーブ側のものです。その数とタイプは，REPLICATE で指定したリモート/ソース/マスター側のフィールドリストに対応していなければなりません。テーブル名の代わりに配列を列挙することにより，ローカル側の更新を完全に制御することができます（配列版 API）。

複製の構文を拡張することにより，ページネーションを実施することができます。

```
REPLICATE フィールド名, フィールド名,...
FROM テーブル名
LIMIT 数値
OFFSET 数値
FOR REMOTE STAMP :in リモートスタンプ,
LATEST REMOTE STAMP :out リモートスタンプ,
INTO テーブル名 (フィールド名, フィールド名,...);
```

LIMIT に続く数値は，一度のリクエストで受け取るアクションの上限を指定するものです。テーブル全体，あるいは大量のレコードを更新する場合，この引数を指定し，空の結果セットが返されるまで命令を繰り返すことにより，複製を分割することができます。たとえば，LIMIT 100，OFFSET 100 を指定した場合，結果セットの 101 番から 200 番目が返されることになります。

OFFSET に続く数値は，指定したスタンプに対応する結果セットに対し，受け取るアクションの先頭からの位置を指定するものです。具体的には，直前の REPLICATE で返された結果セット数の累計を指定することになります。なお，LIMIT と OFFSET を併用することにより，一回の複製（特定のスタンプに対する複製）を分割することはできますが，その一部だけを受け入れるという選択肢はないことに留意してください。分割は，飽くまで一度に返されるデータサイズを抑えるための手段であり，最終的にはすべての更新を受け入れる，あるいはすべての更新をキャンセルする，という選択肢しかありません。

LIMIT と OFFSET を併用し，複製を分割する場合，途中でスタンプが無効になる（前回よりも新しい LATEST REMOTE STAMP が返される）という可能性もあるため，ローカル側は，複製をトランザクションの中で開始することが推奨されています。

複製の構文をさらに拡張することにより，同期されるレコードを絞り込むことができます。

```
REPLICATE フィールド名, フィールド名,...
FROM テーブル名
WHERE 条件
LIMIT 数値
OFFSET 数値
FOR REMOTE STAMP :in リモートスタンプ,
LATEST REMOTE STAMP :out リモートスタンプ,
INTO テーブル名 (フィールド名, フィールド名,...);
```

WHERE に続く SQL は，リモートからローカルへコピーされるレコードを絞り込むための条件式です。SELECT 文で有効な構文であれば，どれでも使用することができます。複製で WHERE 句を使用することには，いくらかの危険が伴います。WHERE 句で取り出されるレコードが流動的である場合，身寄りの無い（オーファン）レコードが発生する恐れがあるからです。たとえば，以前の複製では，WHERE 句に「引っか

かった」レコードが，途中から同じ WHERE 句で検出されないようになったときにそのような事態が起こります。良い例は，そのレコードがリモート側で削除されたときです。そのような場合，「削除」のアクションが発行されないため，問題のレコードはいつまでもローカル側に残ることになり，ローカルとリモートは同期が外れ，複製とは呼べないものになってしまうので，注意が必要です。なお，LIMIT や OFFSET も，数え間違いをしたり，キャンセルするべき状況でキャンセルしなかったりすると，オーファンレコードが発生することになりかねません。

複製の構文をさらに拡張することにより，衝突の問題を解決することができます。

```
REPLICATE フィールド名, フィールド名,...
FROM テーブル名
WHERE 条件
LIMIT 数値
OFFSET 数値
FOR REMOTE STAMP :in リモートスタンプ,
LOCAL STAMP :in ローカルスタンプ,
LOCAL OVER REMOTE|REMOTE OVER LOCAL,
LATEST REMOTE STAMP :out リモートスタンプ,
LATEST LOCAL STAMP :out ローカルスタンプ,
INTO テーブル名 (フィールド名, フィールド名,...);
```

衝突とは，同一のレコードが，複製と複製の間に，リモートとローカルの両方で更新された状況を指します。「同一のレコード」とは，プライマリーキーの値が同じであるということです。「複製と複製の間に」とは，リモート側で当該レコードのスタンプが，複製を要求したローカル側のリモートスタンプ（前回の複製で返されたリモートスタンプ）よりも進められているということです。「両方で更新された」とは，ローカル側でも当該レコードのスタンプが進められており，その値が前回の複製で返されたローカルスタンプ（コマンドに渡されたローカルスタンプ）よりも上であるということです。

ここで大事なことは，ローカルスタンプとリモートスタンプ，ふたつのスタンプを比較することにより，衝突が内部的に検出されるということです。ローカルスタンプとリモートスタンプは，それぞれのデータベースで別々にカウントされているので，どちらの更新がより新しいものであるのか，客観的に判断することができません。分かるのは，同一のレコードが，複製と複製の間に，リモートとローカルの両方で更新された，という衝突の事実であり，データベースを同期するという理念上，どちらかの更新を最新の状態として受けれなければならない，ということです。

LOCAL OVER REMOTE とは，そのような場合，ローカル/デスティネーション/スレーブが勝利することを決めるものです。REMOTE OVER LOCAL とは，反対にリモート/ソース/マスターの更新が勝ち残るというルールです。

REPLICATE の場合，データのコピーは一方通行（リモートからローカルへ）なので，どのようなルールであっても，リモートのデータが複製によって書き変わることはありません。LOCAL OVER REMOTE は，ローカルが勝利するというルールですが，それは，ローカル側のデータがリモートから供給されたデータによって書き変わらないという意味に過ぎないのであり，ローカルの値がリモートに書き込まれるというわけではありません。この結果，ローカルとリモートは，一時的に当該レコードが一致しない状態になります。その後，リモート側でデータがさらに更新され，衝突することなくリモートに複製された時点で，両者はふたたび一致するようになります。

REMOTE OVER LOCAL は，衝突が検出された場合，リモートのデータでローカルを一方的に更新するという意味なので，実際には最初から勝敗が決定しているのであり，ローカルスタンプを指定するまでもありません。REPLICATE の場合，REMOTE OVER LOCAL ルールは，配列版の API を使用し，「同期により，ローカルデータベースはリモートデータベースのレコードで上書きされますが，よろしいですか」といったダイアログを表示したい場合に便利です。

SYNCHRONIZE

シンタックス

同期の構文は，複製よりもずっとシンプルなものです。

SYNCHRONIZE LOCAL TABLE テーブル名 (フィールド名, フィールド名,...)
WITH REMOTE TABLE テーブル名(フィールド名, フィールド名,...)
FOR REMOTE STAMP :in リモートスタンプ,
LOCAL STAMP :in ローカルスタンプ,
LOCAL OVER REMOTE|REMOTE OVER LOCAL,
LATEST REMOTE STAMP :out リモートスタンプ,
LATEST LOCAL STAMP :out ローカルスタンプ;

REPLICATE コマンドでは最後の INTO 句で指定したローカル側のテーブル名とフィールド名が，一番最初に指定されていることに注目してください。複製同様，テーブルに存在するすべてのフィールドを列挙する必要はありません。冒頭で述べたように，同期に配列版の API はありません。

WITH REMOTE TABLE テーブル名およびフィールド名は，リモート/ソース/マスター側のものです。ここに列挙するフィールドの数とタイプは，ローカル側のものと一致していなければなりませんが，両者でテーブル名やフィールド名が一致している必要はありません。

FOR REMOTE STAMP, LATEST REMOTE STAMP, LOCAL STAMP, LATEST LOCAL STAMP の役割は，REPLICATE のときとまったく同じです。しかし，レコードを一方向にコピーする複製とは違い，同期は両方向にコピーを実施し，両者を一致させるのが目的なので，レコードを絞り込むための WHERE 句，データを分割して取り出すための LIMIT と OFFSET が SYNCHRONIZE で使用されることはありません。

REPLICATE あるいは SYNCHRONIZE の間隔が長くなるほど，起こりやすくなります。したがって，衝突を避けるための最良の方法は，なんといってもこまめに同期を実行することです。

HTTP 版

概要

HTTP 経由の同期は，Windows，Linux，Mac OS X のようなデスクトップ・プラットフォーム，あるいは Android や iOS のようなモバイル・プラットフォーム，あるいは Web アプリケーションなど，さまざまなプラットフォームの上に構築されたアプリケーションが，v12 以降の 4D とデータを交換するための有力な手段です。このセクションでは，テーブル同期と複製の HTTP 版 API（4DSYNC）について説明します。

SQL vs HTTP

4D v12 では，専用の SQL コマンドが追加され，v12 以降の 4D であれば，テーブル同士のデータが同期できるようになりました。ネットワーク経由の同期と複製は，データベースエンジンと SQL サーバーのレベルで実装されているので，デベロッパーは，テーブル単位の簡単な設定をするだけで，すぐにセットアップすることができます。ピクチャや BLOB などのバイナリフィールドも同期させることができます。命令は SQL 言語で記述しますが，ネットワーク接続には，SQL サーバーとのダイレクト TCP 通信を使用するので，ODBC ドライバーは不要です。SSL で通信を暗号化することもできます。リクエストの送受信や交換データの構造解析など，煩雑なネットワーク関連のプロトコルは，4D のフレームワークで管理されているので，v12 以降の 4D でテーブル同士のデータを同期させるのであれば，この方法がもっとも簡単です。

HTTP 経由の同期は，データベースエンジンと SQL サーバーのレベルで実装されている上述の機能を，4D 以外のプラットフォームから利用できるようにしたものです。SQL サーバーとのダイレクト TCP 通信の代わりに，HTTP サーバーをインタフェースに使用し，幅広いシステムからネットワーク接続できるようにされていることが特徴です。SSL で通信を暗号化することもできます。プロプライアタリな SQL 命令やデータ構造の代わりに，JSON を使用するため，さまざまなプログラミング言語からアクセスすることができます。（ただし，JSON 形式の制約上，ピクチャや BLOB などのバイナリフィールドは同期させることができません。）汎用的な仕様であるため，リクエストの送受信や交換データの構造解析など，クライアントアプリケーション側の処理は，それぞれのクライアントが各プログラミング言語で実装する必要があります。とはいえ，サーバー側の処理は，データベースエンジンと SQL サーバーのレベルで実装されているので，v12 以降の 4D とそうでないアプリケーションで，テーブル同士のデータを同期させるのであれば，この方法がもっとも簡単です。

セットアップ

HTTP サーバー

HTTP 経由の同期を使用するために必要なことは，HTTP サーバーを起動することを別にすれば，複製と同期（SQL 版）と基本的に同じです。つまり，対象テーブルにプライマリーキーを設定し，複製を有効にし，APPLY TO SELECTION などの方法で既存のレコードにタッチ（対象に含めたい場合）すれば，複製の準備は完了です。

Web サーバーを起動し，下記の URL にブラウザでアクセスしてみてください。

```
http://127.0.0.1/4DSYNC/$catalog
```

標準テキストで，複製を有効にされたテーブルの名前と，レコード数が返されるはずです。

注記：このテストで明らかなように，複製が有効にされたデータベースは，Web サーバーを起動した途端，複製が有効にされたテーブルには誰でも HTTP 経由でアクセスできるようになります。これを防止するためには，後述するいずれかの手段でアクセスを制御することが必要です。

プライマリーキー

4D 言語は，歴史的な経緯から，しばしばレコード番号を参照することがありますが，同期と複製を含め，SQL では，プライマリーキーでレコードを特定します。プライマリーキーは，テーブルで重複しないフィールドの値，あるいはフィールドの値の組み合わせです。4D は，今後，レコード番号よりもプライマリーキーを重視するようになります。プライマリーキーは，4D v11 では SQL 言語で定義することができました。v12 では，SQL 言語に加えてストラクチャーエディターでもプライマリーキーが設定できるようになりました。

プライマリーキーの条件が満たされてさえいれば，どのようなフィールドや値の組み合わせをプライマリーキーにしても構いませんが，プラットフォームや言語を超えた同期や複製を構築するつもりであれば，一意性が保証されており，さまざまな言語で実装されている UUID をプライマリーキーにするのが，もっともシンプルで良い方法かもしれません。プライマリーキーが UUID であれば，同期と複製のクライアントアプリケーションもローカルでリモートと同じタイプの UUID を発行することができます。

JSON レコード

HTTP 経由の同期では，JSON 形式で双方向にデータを転送します。JSON なので，文字コードは UTF-8 です。文字列やテキストのレコード値，テーブル名，フィールド名は，必要に応じてエスケープします。文字ベースのフォーマットなので，ブール，整数，倍長整数，実数はもちろん，Integer 64 bits や Float のような SQL データタイプも含めることができます。日付は，RFC822 フォーマットで表現します。（例：Sun, 08 Jan 2012 13:00:00 GMT）時間は，D:H:M:S:MS フォーマットで表現します。（例：0:3:45:0:0 = 午前 3 時 45 分）

テーブルリスト

暫定的な API ですが，ストラクチャ情報を HTTP 経由でリクエストすることができます。つまり，下記のようなリクエストを発行すれば，指定したテーブルの情報を疑似 XML 形式で取り出すことができます。

```
GET /4DSYNC/$catalog/テーブル名
```

たとえば，ダイナミックな同期アプリケーションを構築したい場合，この情報に基づき，リモートデータベースのフィールドを選択することができます。あるいは，リモート側のストラクチャ定義が変化した場合にも，動的に対応することができます。もっとも，通常の同期クライアントは，特定のアプリケーションとだけ同期を実行するはずであり，テーブル名やフィールド名は分かっているはずなので，基本的に API でストラクチャ情報を取り出す必要はないはずです。

注記：この API から返されるのは，4D の「書き出し」ダイアログで「XML 形式」を選択したときと同じ，擬似的な XML 形式です。フィールド名やテーブル名がそのまま XML の要素名になるため，空白や XML で許されない名前が文字列に含まれている場合，クライアント側は XML の解析に失敗します。安全のため，同期クライアントは，別の方法で 4D データベースのテーブル名やフィールド名を入手できるようにしたほうが良いかもしれません。なお，将来的には，JSON 形式の API が提供される予定です。

GET と POST

SQL 同期では，SYNCHRONIZE あるいは REPLICATE の SQL コマンドを SQL サーバーに要求しますが，HTTP 同期では，GET あるいは POST の HTTP コマンドを HTTP サーバーに要求します。同期と複製でコマンドが分かれているわけではなく，アップロードとダウンロードでコマンドが分かれていることに留意して

ください。HTTP で複製を実行したい場合，GET で複製に必要な情報を取得し，その内容をローカルに適用します。複製に必要な情報は，サーバーが自動的に作成しますが，ローカルデータベースの更新は，クライアントが自分で行なう必要があります。HTTP で同期を実行したい場合，複製を実行した後，POST で同期に必要な情報をリモートに送信します。リモートデータベースの更新は，すべてサーバーが自動的に行ないますが，同期に必要な情報の作成は，クライアントが自分で行なう必要があります。つまり，HTTP 同期で使用する GET あるいは POST コマンドは，SYNCHRONIZE/REPLICATE のサーバー側だけを自動化したものであり，クライアント側の処理（JSON オブジェクトの解析と作成，HTTP リクエストの処理，データベースの更新）は何もしないということです。

GET

URL

HTTP 経由の同期では，特別な URL（4DSYNC）に必須あるいは任意の引数をパスあるいはパラメータとして追加した URL を HTTP サーバーにリクエストします。

GET /4DSYNC/テーブル名/フィールド名 1{,フィールド名 2},...

HTTP 経由の同期では，最初に GET コマンドを発行し，リモート側のデータベースから同期の対象となるレコードを取り出します。一度のリクエストで同期できるのは，ひとつのテーブルだけであり，そのテーブル名は/4DSYNC/に続く URL コンポーネントで指定します。そのテーブルに存在するフィールドをすべて同期する必要はありません。レコードを取り出したいフィールドは，テーブル名に続く URL コンポーネントにカンマ区切り列挙します。最後のフィールド名を記述した後，クエスチョンマーク（?）に続けて必須または任意のパラメーターを渡します。パラメータ名は，それぞれドル記号（$）から始まります。したがって，最終的な URL は，次のようなものになります。

http://localhost/4DSYNC/Meetings/Name,Meeting_Date,Start_Time,End_Time,Purpose?$stamp=0&$format=json&$top=3&$offset=5

パラメーター

stamp【必須】

SQL コマンドの REPLICATE や SYNCHRONIZE で指定する，FOR REMOTE STAMP と同じスタンプ，つまり，同期を実行するたびに返された，LATEST REMOTE STAMP です。このスタンプよりも古い（番号の若い）スタンプのレコードは，リモート側から返されません。複製が有効にされた後，更新または削除されたレコードは，1 以上のスタンプが付けられているので，スタンプに 0 を指定すれば，複製が有効にされた後に更新または削除されたレコードがすべて入手できます。ただし，複製が有効にされる前から存在するレコードは，スタンプ自体が付けられていないので，スタンプに 0 を指定したとしても，同期の対象外です。そのようなレコードも同期に含めるのであれば，SQL 版の同期と同じように，-1 あるいは-2 を指定します。

format【必須】

HTTP 経由の同期でサポートされているのは，JSON 形式だけです。

top【任意】

SQL コマンドの REPLICATE の LIMIT と同じように，一度のリクエストで返されるレコードの最大数を指定することができます。SQL 版/HTTP 版ともに，一度の同期で処理するレコード数は，道理にかなった範囲内にとどめることが強く勧められています。もっとも，ページネーションされた同期を実行したとしても，残っ

ているレコード数が報告されるわけではないので，最終的にサイズ 0 のレスポンスが返されるまで，リクエストを繰り返さなければなりません。

skip【任意】

SQL コマンドの REPLICATE の OFFSET と同じように，特定のリモートスタンプに対するレスポンスの途中からレコードを入手することができます。top と併用することにより，ページネーションされた同期を実行することができます。

サンプルコード

```
// URL を構築
var url = '4DSYNC/Meetings/Name,Meeting_Date?$stamp=3&$format=json';

// HTTP リクエストを作成
var request = new XMLHttpRequest();

// リクエストが実行された後にコールされる関数を設定
request.onreadystatechange = myHandler;

// GET コマンドを指定
request.open('GET', url, true);

// リクエストを実行
request.send(null);
```

レスポンス

HTTP 経由の同期では，単一の JSON オブジェクトがサーバーから返されます。このレスポンスは，同期に関連したヘッダー情報，およびレコードデータの配列で構成されています。

__SENT

このレスポンスに含まれているレコード数です。これは，リクエストされたスタンプに対応するレコードの総数であるとは限りません。ページネーションされた同期の場合，最終的にサイズが 0 のレスポンスが変えされるまで，リクエストを繰り返す必要があります。

__LATEST_STAMP

次回のリクエストで stamp に渡すべきリモートスタンプ，つまり，SQL コマンドの REPLICATE や SYNCHRONIZE で指定する，LATEST REMOTE STAMP と同じスタンプです。ページネーションされた同期の場合，連続したリクエストの途中でスタンプが変化する可能性があります。そのような場合，それまでのページネーションされた同期をすべてキャンセルし，返されたリモートスタンプを破棄し，リクエストに使用したスタンプで同期を最初からやり直すことが必要です。ページネーションされた同期が成功とみなされるの

は，一連のリクエストで返されたリモートスタンプが変化しなかったときだけであるべきです。クライアント側は，実際に成功した同期のリモートスタンプを次回のために保持するようにしてください。

__ACTIONS

レコードデータの配列です。同期では，delete あるいは update タイプの JSON オブジェクトがレコード数だけ返されます。いずれのタイプにも，レコードを特定するためのプライマリーキー，およびスタンプとタイムスタンプが含まれており，クライアント側は，このデータに基づき，ローカルデータベースのレコードを特定，更新（作成）あるいは削除することが期待されています。フィールド値が JSON オブジェクトで含まれているのは，更新タイプのレコードデータだけです。

POST

URL

POST の URL は，GET とまったく同じです。

POST /4DSYNC/テーブル名/フィールド名 1{,フィールド名 2},...

パラメーター

POST では，GET と同じように stamp と format が必須パラメーターです。ページネーションは，クライアント側でリクエストを分割することにより，実現されるものなので，top や skip は不要です。

リクエスト

クライアントは，更新レコードを GET で返されるレスポンスと同じような JSON フォーマットで POST リクエストの HTTP ボディでサーバーに送信します。SQL 版の API では，フィールドの順序は自由ですが，HTTP 版の場合，JSON オブジェクトがシーケンシャルに解析されるため，フィールドの順序はストラクチャ定義と合致しなければなりません。また，サーバーに更新が反映されるようにするためには，直前の GET でサーバーから返されたリモートスタンプと同じ値を_LATEST_STAMP に渡すことが必要です。なお，スタンプの値そのものは整数ですが，JSON オブジェクトの中では string で表現されていることに留意してください。

サンプルコード

```
// ボディを構築
var postBody = {
"__SENT"      : 1,
"__LATEST_STAMP"  : "3" // number ではなく string
"__ACTIONS" : [{     // 削除アクション 1 個の配列
"__ACTION" : "delete",
"__PRIMARY_KEY_COUNT" : 1,
"__PRIMARY_KEY" : {
"id"     : generateUUID()      // generateUUID 関数は定義されていると仮定
},
"__STAMP"     : "3",
```

```
"__TIMESTAMP" : new Date().toUTCString() // javascript で生成
}]
}

// URL を構築
var url = '4DSYNC/Meetings/Name,Meeting_Date?$stamp=3&$format=json';

// HTTP リクエストを作成
var request = new XMLHttpRequest();

// JSON オブジェクトを string に変換
var postBodyAsString = JSON.stringify(postBody);

// リクエストが実行された後にコールされる関数を設定
request.onreadystatechange = myHandler;

// POST コマンドを指定
request.open('POST', url, true);

// リクエストを実行
request.send(postBodyAsString);
```

セキュリティ

HTTP 経由の同期は，通信層に 4D HTTP サーバーを使用し，アプリケーション層に内部 SQL サーバーを使用する，複合的なプロトコルです。そのため，HTTP サーバーのセキュリティ体系と SQL サーバーのセキュリティ体系のいずれか，あるいは両方を適用することができます。

4D パスワードシステム

データベース設定の「Web」ページで「BASIC 認証のパスワード」と「4D パスワードを含む」を有効にするだけでセットアップが完了する方式です。クライアントは，HTTP の BASIC 認証パスワードを送信し，サーバーは，その受信したユーザー名とパスワードがツールボックス/ユーザーで登録されたクライアント/サーバー用のユーザー名とパスワードと合致すれば，自動的にアクセスを許可します。別途，サーバー側で認証システムを実装する必要はありません。

スキーマ

セキュリティを高めるため，V11.3 で登場した，SQL サーバーのアクセス管理システムを併用することもできます。HTTP クライアントは，前述の 4D パスワード方式で自動アクセス認証させますが，使用する 4D ユーザーには，あらかじめ限定的なテーブルに対する読み（書き）アクセス権だけを設定しておけば，安心です。

スキーマを SQL 言語で設定すること以外には，別途，サーバー側で認証システムを実装する必要はありません。

BASIC パスワードシステム

データベース設定の「Web」ページで「BASIC 認証のパスワード」だけを有効にし，「4D パスワードを含む」を選択しなければ，クライアント/サーバー用のユーザー名とパスワードの代わりに，任意のパスワードシステムを実装することができます。具体的には，HTTP クライアントから送信された BASIC 認証のユーザー名とパスワードが，On Web Authentication データベースメソッドの$5 と$6 に代入されるので，その情報に基づいてアクセスの認証を行ないます。クライアント IP アドレス（$3）や HTTP ヘッダーなど，他の情報も認証の根拠に加えることができます。

DIGEST パスワードシステム

v11 で登場した，よりセキュリティの強化された HTTP パスワードシステムです。実際のユーザー名とパスワードの代わりに，クライアントはハッシュされたダイジェスト情報を送信し，サーバーは Validate Digest Web Password コマンドを使用してそのハッシュを検証します。BASIC 認証のようにユーザー名とパスワードが簡単に割り出されることはありません。HTTP アクセス用の限定的なユーザー名とパスワードであり，任意のパスワード認証コードを記述することができるという面でも安全です。ただし，クライアントアプリケーション側でダイジェストパスワードに対応する必要があります。

参考資料

4DSYNC を無効にする

4DSYNC は，4D の Web サーバーが起動していれば，自動的に処理される特別な URL です。4D v13 では，データベース設定により，URL を無効にすることもできますが，v12 以前では，On Web Authentication データベースメソッドで/4DSYNC/から始まる URL をフィルターすることが必要です。

参考：特別な URL を無効にする方法は，URL のタイプによって違います。

/4DSTATS　　：Designer にパスワードを設定。
/4DHTMLSTATS　：Designer にパスワードを設定。
/4DCACHECLEAR　：Designer にパスワードを設定。
/4DWSDL/　　：データベース設定「Web サービスリクエストを許可する」
/4DSYNC/　　：On Web Authentication。またはデータベース設定「4DSYNC URL を許可する」
/4DSOAP/　　：On Web Authentication。またはメソッドプロパティ「Web サービスとして提供」
/4DACTION/　　：On Web Authentication。またはメソッドプロパティ「4D HTML タグおよび URL（4DACTION…）で利用可能」

v11 以前で JSON と HTTP

HTTP 版の API は，おもに 4D 以外のリモートアプリケーションを想定しています。4D 同士の同期であれば，SQL 版の API で同期が実現できるからです。ただし，4D であっても，v11 以前は SQL でアクセスができないので，HTTP 経由で同期したいと考えるかもしれません。その場合，JSON オブジェクトの作成と解析，HTTP リクエストが標準コマンドでは用意されていないことが問題になります。JSON とネットワークの処理は，サードパーティ製プラグインの NTK，JSON だけであれば，Pelican Engineering のコンポーネントが利用できます。

www.pluggers.nl
www.pelicaneng.ca

同期に必要なライセンス

SQL 版の同期は，SQL サーバーを外部に公開することが関係しているので，必然的に，リモート/ソース/マスター側は，4D Server に限定されます。SQL 接続には空いている 4D Client Expansion ライセンス，あるいは SQL Unlimited Expansion ライセンスが使用されます。OEM Desktop も，SQL サーバーを公開することができますが，デスクトップアプリケーションをサーバー公開する場合，接続できる SQL クライアントは，固定の IP アドレスあるいはローカルホストに限定されることになります。クライアントは，4D SQL Desktop，4D Server，あるいは OEM Desktop です。Unlimited Desktop は，データの読み取りは無制限，書き込みは 24 時間に 2 回までの「デスクトップ」制限があることに留意してください。

HTTP 版の同期は，HTTP サーバーを外部に公開することが関係しているので，リモート/ソース/マスター側は，4D Web Application Server，あるいは 4D Server に 4D Web Application Expansion あるいは 4D Web Services Expansion のどちらかを追加したものになります。OEM Desktop も，HTTP サーバーを公開することができます。クライアントは，JavaScript アプリケーション，あるいは JSON 形式が処理できる一般的な HTTP クライアントアプリケーションです。