

Informix Red Brick Migration Guide Ver 2 (5.x から 6.0.2 への移行)

2000 年 3 月版

まえがき

このガイドは、Red Brick Warehouse 5.x をご利用のお客様が Informix Red Brick Warehouse 6.0 リリースへ移行する際に必要な移行に関する情報を記載しています。

6.0 データベースサーバの新機能

リリース 6.0 の Red Brick サーバには、次の新機能が搭載されています。

- VARCHAR データ型 (文字データの可変長格納領域)
- EXPORT 操作。クエリの結果をファイルへ送信したり、出力を別のプログラムへパイプ転送したりすることができます。
- クエリ・プライオリティ・コンカレンシ拡張機能
 - 凍結クエリ・リビジョン — データベースのロードとクエリを同時実行し、かつクエリの読み取りリビジョンを凍結する機能。
 - 並列バージョン管理ロード — バージョニングがオンである場合に、PTMU によりデータをロードする機能
- パフォーマンス拡張機能
 - 参照整合性チェック — プライマリ・キー・インデックスの使用効率を向上
 - DELETE ステートメントおよび UPDATE ステートメントでのジョインをサポート
 - Vista クエリ・リライト・サポートを拡張 — BREAK BY クエリおよび RESET BY クエリのリライトを実現。検索項目リストでの集約を使用した CASE 式、UNION 操作の結果に対して制約のあるクエリが可能。
- CHECK TABLE コマンドおよび CHECK INDEX コマンド (5.1.7 から搭載されたコマンド)

Parallel Table Management Utility (PTMU)、Vista、Client Connector Pack は、6.0 より前のリリースではすべて別売のオプション製品でしたが、6.0 サーバにはこれらの製品が含まれています。

新機能の詳細については、6.0 ソフトウェアに付属の **Answers OnLine** マニュアル CD-ROM を参照してください。

コネクティビティ

Client Connector Pack バージョン 2.0 には、ODBC Driver と新しい JDBC Driver が含まれています。

- 2.0 ODBC Driver は 5.1.x データベースおよび 6.0 データベースに対応しています。
- JDBC Driver は、Informix Red Brick Warehouse 6.0に対応しています。

この 2 種類のドライバは 1 枚の CD-ROM に収録されています。ドライバは、CD-ROM から個別にインストールします。詳細については、『*Client Connector Pack Installation Guide*.』を参照してください。

Administrator ツール

Administrator ツールである Informix Red Brick Warehouse Administrator は、サーバとは別にインストールします。

Administrator バージョン 2.0 は、6.0 データベースにのみ対応しており、VARCHAR データ型などの 6.0 の新機能をサポートしています。

Administrator 1.0.x バージョンは 6.0 データベースには対応していません。

対応製品一覧

データベースを作成するサーバ	対応製品
Red Brick Warehouse 5.1	Client Connector Pack 1.0 および 2.0 Administrator 1.0
Informix Red Brick Warehouse 6.0	Client Connector Pack 2.0 JDBC Driver 2.0 Administrator 2.0

6.0 マニュアル・セット

以下の Red Brick マニュアル・セットの変更内容に注意してください。

- 『*Warehouse Administrator's Guide*』のWindows NTおよびUNIX版はバージョン6.0『*Informix Red Brick Warehouse Administrator's Guide*』に対応しています。このガイドには、UNIX または Windows NT をご利用のお客様を対象にした情報が収録されています。
- 6.0 マニュアル・セットの新規マニュアルは、『*JDBC Connectivity Guide*』のみです。
- 『*Documentation Addendum*』(リリース 5.1.5 対応)の内容が、6.0 マニュアル・セットに統合されました。

アップグレード手順

リリース 6.0では、VARCHAR データ型の導入に伴うシステム・カタログの変更を反映するために、ご使用の 5.x データベースをアップグレードする必要があります。

リリース 5.0.3 以降のデータベースは、6.0 へ直接アップグレードすることができます。

重要: バージョニング機能を使用している5.1.4以降のデータベースへバージョンアップする場合、バージョンアップ前にバージョニングを停止しバージョンログを削除しなければいけません。これを行わないとバージョンアップに失敗します。

5.x データベースから 6.0 へのバージョンアップ

1. バージョンアップしたい5.x データベースのフルバックアップを作成する。
2. 5.x データベースを 5.x サーバへ接続し、バージョニングを停止し、バージョンログを削除する(データベースにバージョニング機能を設定していた場合)
3. 6.0 サーバをインストールする
4. 新しいrbw.configファイルにデータベース名を追加する
5. 新しくインストールしたところを差し示すように環境変数を変更する
6. 6.0 サーバで5.x データベースに接続し以下のTMUコマンドを実行することでデータベースをバージョンアップする

```
% rb_tmu -d db_name filename db_username password
```

db_name はデータベースの論理名、*filename* は以下の文字列が記述されているコントロール・ファイルです。

```
UPGRADE;
```

7. バージョンアップした6.0 データベースのフルバックアップを作成する

アップグレード処理について

UPGRADE 操作により、既存のスキーマをそのインデックスとともにメンテナンスします。6.0 の新機能を利用するために変更を行う場合には、アップグレード後に変更作業を行います。

VARCHAR データ型の導入により、(可変長文字データの格納領域である VARCHAR データ型を受け入れるために) 新しい“テーブル・フォーマット”が必要です。

UPGRADE 操作では、既存のスキーマはこの新しいテーブル・フォーマットに変更されません。ただし、アップグレード後の 6.0 データベースを管理するときに、次の操作を実行するとテーブル全体が変換されます。

- ALTER TABLE ADD COLUMN
- ALTER TABLE DROP COLUMN

ターゲット・テーブルのサイズが大きいと、上記の操作には時間がかかることがあります。

セグメントまたは PSU の追加などテーブル格納領域を変更する操作では、テーブル全体が 5.x フォーマットで保存されます。テーブルで 5.x フォーマットと 6.0 フォーマットが混在することはありません。

可変長文字データ

リリース 6.0 の Red Brick サーバにおける最も重要な変更点であり、かつ移行に関して重要な唯一の変更点は、可変長文字データのサポートです。新しい VARCHAR データ型を使用すれば、すべての値に対して固定長の文字列を適用する代わりに、各文字列をその実際の長さに従って列に格納することができます。

6.0 へアップグレードすると、既存の CHAR 列を VARCHAR に変換することで、文字データを格納するディスク領域を大幅に節約できることがあります。さらに、可変長列を使用すると、ロード時およびクエリ実行時に処理する必要のあるデータの量が減少します。この処理データ量の減少は、クエリとロードの実行パフォーマンスの向上につながります。

アプリケーションで可変長データ格納領域の使用が有効であると判断した場合は、CHAR から VARCHAR への変換を、テーブルごとに実行すること (列ごとの変換ではない点に注意) を推奨します。変換するテーブルのサイズによっては、変換処理に時間がかかることがあります。特に、テーブルに複数のインデックスが設定されている場合、あるいはテーブルが他のテーブルから複数のインデックスにより参照されている場合に処理時間がかかる可能性があります。変換処理の一部として、このようなインデックスの一部またはすべてを削除し、再作成する必要があります。

重要: 変換処理やその他の 6.0 管理タスクを開始する前に、アップグレード後の 6.0 データベースのフル・バックアップを実行してください。

変換処理の例

Store というディメンジョン・テーブルに、次の列があるとします。

列名	データ型
Storekey	INT
Mktkey	INT
Store_Type	CHAR(10)
Store_Name	CHAR(30)
Street	CHAR(30)
City	CHAR(20)
State	CHAR(5)
Zip	CHAR(10).

Store ディメンジョン・テーブルは Sales テーブルにより参照されています。Storekey は Store テーブルのプライマリ・キーであり、Sales テーブルのフォーリン・キーです。次のクエリは、格納領域が固定長であるために、固定長 30 文字の列 Street の約半分に不必要な空白が埋め込まれることを示しています。

```
RISQL> select avg(length(trim(street))) from store;  
16.777777
```

Store テーブルの中で、Street の他に長さが増える可能性がある列は Store_Name と City です。

```
RISQL> select avg(length(trim(store_name))) from store;  
16.388888  
RISQL> select avg(length(trim(city))) from store;  
8.333333
```

Store テーブルのその他のCHAR 型列には、10 文字以下の一定の長さの値が格納されます。したがって、可変長格納領域を設定する必要はありません。

Store テーブルのサイズが、データ型の変換によりディスク領域を節約できる程度の大きさであるとするれば、**Store** テーブルを再作成し、**Store_Name**、**Street**、**City** を可変長格納領域に設定することができます。変換処理を効率的かつ正常に行うために、テーブル全体を削除、再作成、再ロードすることを推奨します。

Store テーブルを再作成する

1. ターゲット・テーブル (この場合 **Store**) を **EXTERNAL** フォーマットでアンロードします。
2. ターゲット・テーブルが参照先テーブルの場合、参照元テーブル (この例では **Sales**) からフォーリン・キー制約とインデックスを削除します。
3. 編集した生成 **DDL** ファイルを使用して、ターゲット・テーブルを削除してから再作成します。
4. 編集した生成 **TMU** ファイルを使用して、このテーブルを再ロードします。
5. テーブルの既存のインデックスを再作成します。
6. 再作成結果を確認するため、テーブルに対してクエリを実行します。
7. 参照先テーブルのフォーリン・キー制約を復元し、このテーブルのインデックスを再作成します。

ステップ 1 — Store テーブルをアンロードする

TMU コントロールファイルを次のように作成します。

```
unload store external
ddlfile 'store.create'
tmufile 'store.load'
outputfile 'store.output';
```

EXTERNAL フォーマットを使用していることを確認してください。これにより、デフォルトで文字データが固定幅文字列に設定されます。EXTERNAL VARIABLE フォーマットは使用しないでください。EXTERNAL VARIABLE フォーマットは、可変長文字列がすでに格納されているデータをアンロードする場合に使用します。

TMU または PTMU を使用してアンロード操作を次のように実行します。

```
% rb_tmu store_unload.tmu system manager
```

store_unload.tmu は、コントロールファイルの名前です。

テーブルをアンロードする前に、テーブルの行数を確認するため、テーブルに対して `select count(*)` クエリを実行できます。この場合、アンロード実行時に TMU 情報メッセージをチェックすれば、テーブル全体が正常にアンロードされていることを確認できます。

ステップ 2 — 参照元テーブルから STAR インデックスとフォーリン・キー制約を削除する

変換するテーブルを参照するテーブルがある場合、変換するテーブルの削除と再作成を行う前に、参照元テーブルから該当する STAR インデックスとフォーリン・キー制約を削除する必要があります。この例では Sales テーブルが参照元テーブルです。

STAR インデックスとフォーリン・キー制約を削除する

1. Store テーブルを参照する STAR インデックスを削除します。

```
RISQL> drop index sales_star_idx;
```

2. Store テーブルを参照するフォーリン・キー制約を削除します。

```
RISQL> alter table sales drop constraint sales_store_fk;
```

ステップ 3 — アンロードしたテーブルを削除してから再作成する

1. アンロードしたテーブルを削除します。

```
RISQL> drop table store;
```

2. 3 つの CHAR 列を VARCHAR へ変換するため、生成 DLL ファイル (store.create) を編集します。
 - それぞれの VARCHAR 列の最大長を指定します。
 - この場合、各 VARCHAR 列に FILLFACTOR 値を指定することができます。デフォルトは 10 % です。FILLFACTOR 値は、列の値の平均長を示します。たとえば、最大長が 1000、平均長が 100 の列値の場合、FILLFACTOR を 10 % (デフォルト) に設定する必要があります。
3. 参照元テーブルで STAR インデックスを再作成できるようにするため、この DLL ファイルに MAXROWS PER SEGMENT も指定します。

完成した CREATE TABLE 文は次のとおりです。FILLFACTOR が各 VARCHAR 列定義の最後に指定されている点に注意してください。

```
CREATE TABLE STORE (  
STOREKEY INTEGER NOT NULL UNIQUE,  
MKTKEY INTEGER NOT NULL,  
STORE_TYPE CHARACTER(10) NOT NULL,  
STORE_NAME VARCHAR(30) NOT NULL  
WITH FILLFACTOR 50,  
STREET VARCHAR(30) NOT NULL WITH FILLFACTOR 50,  
CITY VARCHAR(20) NOT NULL WITH FILLFACTOR 40,  
STATE CHARACTER(5) NOT NULL,  
ZIP CHARACTER(10) NOT NULL,  
PRIMARY KEY(STOREKEY),  
CONSTRAINT STORE_FKC FOREIGN KEY(MKTKEY)  
REFERENCES MARKET (MKTKEY) ON DELETE NO ACTION)  
MAXROWS PER SEGMENT 2500;
```

FILLFACTOR の設定値 50 % および 40% は、この手順で前述した長さ (trim) クエリの実行結果にしたがって設定されています。

重要: VARCHAR 列の使用により実際に節約できるディスク領域の容量は、FILLFACTOR の設定値の正確さに基づいています。FILLFACTOR の設定値が高すぎると、節約できる PSU 領域は非常に少なくなります。

ステップ 4 — 新しいテーブルを再ロードする

VARCHAR 列のある Store テーブルがデータベースに作成されました。次に、生成 TMU ファイルを使用して Store テーブルを再ロードします。

Store テーブルを再ロードする

1. 変換された 3 つの列に対して RTRIM 関数を指定するため、生成ファイルを編集します。RTRIM は、アンロードされた固定幅値から末尾の空白を削除するために使用します。編集後の TMU スクリプトは次のようになります。

```
LOAD DATA INPUTFILE 'store.output'
RECORDLEN 136
INSERT
NLS_LOCALE 'English_UnitedStates.US-ASCII@Binary'
INTO TABLE STORE (
STOREKEY POSITION(2) INTEGER EXTERNAL(11)
    NULLIF(1)='% ',
MKTKEY POSITION(14) INTEGER EXTERNAL(11)
    NULLIF(13)='% ',
STORE_TYPE POSITION(26) CHARACTER(10) NULLIF(25)='% ',
STORE_NAME POSITION(37) CHARACTER(30) RTRIM
    NULLIF(36)='% ',
STREET POSITION(68) CHARACTER(30) RTRIM
    NULLIF(67)='% ',
CITY POSITION(99) CHARACTER(20) RTRIM NULLIF(98)='% ',
STATE POSITION(120) CHARACTER(5) NULLIF(119)='% ',
ZIP POSITION(126) CHARACTER(10) NULLIF(125)='% ');
```

2. 編集したファイルを TMU により呼び出します。

```
% rb_tmu store.load system manager.
```

ステップ 5 — 新しいテーブルのインデックスを作成する

削除前にテーブルに設定されていたインデックスを再作成します (ただし プライマリ・キー・インデックスは除きます)。

```
RISQL> create target index store_tgt_idx on store(store_type);
```

重要: *VARCHAR* 列に対して *TARGET* インデックスを作成することはできません。*TARGET* インデックス付き *CHAR* 列を *VARCHAR* 列に変換した後で、この列に *B-TREE* インデックスを作成する必要があります。

ステップ 6 — データを確認する

クエリを使用して再ロード結果を調べます。

```
select min(length(store_name)) as col1,  
       max(length(store_name)) as col2  
from store;
```

```
COL1 COL2  
11    25
```

このクエリの実行結果により、*Store_Name* 列には 11 ~ 25 文字の可変長値が格納されていることが確認されます。

今後ロードを実行すると、*Store_Name* 列には 30 文字以下の文字列が格納されます。

ステップ 7 — 参照元テーブルのフォーリン・キー制約とインデックスを復元する

最後に、Sales テーブルから削除したフォーリン・キー制約を復元し、適切な STAR インデックスを再作成します。

フォーリン・キー制約とインデックスを復元する

1. 制約を追加します。

```
RISQL> alter table sales add constraint sales_store_fk  
        foreign key(storekey) references store(storekey)
```

2. インデックスを再作成します。

```
RISQL> create star index sales_star_idx on sales  
        (sales_date_fk, sales_product_fk, sales_store_fk,  
         sales_promo_fk);
```