DBMaster

データベース管理者参照編

SYSCOM

CASEMaker Inc./Corporate Headquarters

1680 Civic Center Drive
Santa Clara, CA 95050, U.S.A.

Contact Information:
CASEMaker US Division
E-mail : info@casemaker.com
Europe Division
E-mail : casemaker.europe@casemaker.com
Asia Division
E-mail : casemaker.asia@casemaker.com(Taiwan)
E-mail : info@casemaker.co.jp(Japan)

www.casemaker.com
www.casemaker.com/support

Document No. 645049-236180/DBM54J-M09302015-DBAG
発行日:2015-09-30

本文には記されていない新しい機能についての説明は、CASEMakerのDBMasterをインストールしてからREADME.TXTを読んでください。

登録商標
CASEMaker、CASEMakerのロゴは、CASEMaker社の商標または登録商標です。
DBMasterは、Syscom Computer Engineering社の商標または登録商標です。
Microsoft、MS-DOS、Windows、Windows NTは、Microsoft社の商標または登録商標です。
UNIXは、The Open Groupの商標または登録商標です。
ANSIは、American National Standards Institute, Incの商標または登録商標です。

ここで使用されているその他の製品名は、その所有者の商標または登録商標で、情報として記述しているだけです。SQLは、工業用語であって、いかなる企業、企業集団、組織、組織集団の所有物でもありません。

注意事項
本書で記述されるソフトウェアは、ソフトウェアと共に提供される使用許諾書に基づきます。

保証については、ご利用の販売店にお問い合わせ下さい。販売店は、特定用途への本コンピュータ製品の商品性や適合性について、代表または保証しません。販売店は、突然の衝撃、過度の熱、冷気、湿度等の外的な要因による本コンピュータ製品へ生じたいかなる損害に対しても責任を負いません。不正な電圧や不適合なハードウェアやソフトウェアによってもたらされた損失や損害も同様です。

本書の記載情報は、その内容について十分精査していますが、その誤りについて責任を負うものではありません。本書は、事前の通知無く変更することがあります。

目次

1 はじめに

データベース管理者参照編にようこそ。DBMasterは、強力かつ柔軟なSQLデータベース管理システム(DBMS)です。会話型の構造的問合せ言語(SQL)、Microsoftのオープンデータベース結合(ODBC) 互換インタフェース、およびC言語のための組込みSQL(ESQL/C)をサポートします。DBMasterはさらにジャバ・データベース・コネクティビティ(JDBC)の準拠のインターフェースおよびDBMaster COBOLインターフェース(DCI)をサポートします。公開アーキテクチャーであるODBCインタフェースは、多種多様なプログラミングツールを使用して顧客アプリケーションを構築し、既存のODBC-適合アプリケーションを用いてデータベースに問合せることを可能にします。

DBMasterは、シングルユーザーの個人データベースから、企業全体に分散するデータベースまでに容易にスケール化することができます。どのようなデータベース構成を選択しても、重要データの安全性は、DBMasterのセキュリティ、整合性、信頼性の先進的機能によって確実に保証されます。広範なクロス-プラットフォームのサポートは、現在あるハードウェアの効力を高め、需要の増大に応じてより強力なハードウェアに拡大し、グレードアップすることを可能にします。

DBMasterは、優れたマルチメディア処理機能を提供し、あらゆるタイプのマルチメディアデータを格納、探索、検索、操作を可能にします。バイナリラージオブジェクト(BLOB)は、DBMasterの先進的セキュリティと損傷リカバリ機構を全面的に利用して、マルチメディアデータの整合性を確実にします。ファイルオブジェクト(FO)は、マルチメディアデータを管理する一方で、元のアプリケーションで各ファイルを編集する機能を維持します。

本書は、DBMaster DBMS の概念と原理、DBMaster SQL 問合せ言語の形式と文法に慣れていないデータベース管理者を読者に想定しています。しかしながら、コンピュータの一般的な実務知識をもち、DBMaster を走らせるオペレーティングシステムを楽に使用できる読者を対象にします。オペレーティングシステムの情報は本書の範囲外です。その分野で問題が起きた場合は、オペレーティングシステムのマニュアルを参照してください。

本書は、DBMaster DBMS を使用する上で、データベース管理者が理解しなければならない概念と原理の一般的情報を記載し、データベースの作成、保守、最適化に必要な DBMaster SQL 文の使用方法を概説します。全体に例と図を用意し、明確に理解できるようにします。

データベースの操作性能は、DBMS の設定によって大きく影響されます。データベースの性能を最適化しチューンアップするには、データの格納位置、アクセス、索引構成、データの保護等の多くの決定が必要になります。本書は、データベース管理者あるいはアプリケーション開発者として決定した選択の効果を理解するためのバックグラウンドを提供します。
DBMaster がサポートする機能参照は、ほとんど SQL 文を用いて例示します。読者は SQL 言語に慣れてことを想定しています。

本書に記載する概念、命令、例の大部分は、DBMaster が提供する dmSQL を用いて提示します。dmSQL は、SQL 構文のコマンドラインツールです。データベース管理の一部の機能は、ごくまれに、DBMaster アプリケーションツールかユーティリティのどれかを使用して実行する場合があります。DBMaster が提供するアプリケーションツールとユーティリティの詳細な使用方法に関しては、「その他のマニュアル」を参照してください。

1.1 その他のマニュアル

DBMaster には、本マニュアル以外にも多くのユーザーガイドや参照編があります。特定のテーマについての詳細は、以下のマニュアルをご覧下さい。

- DBMasterの能力と機能性についての概要は、*「DBMaster入門編」*をご覧下さい。
- DBMasterの管理についての詳細は、*「JServer Managerユーザーガイド」*をご覧下さい。

- DBMasterの環境設定についての詳細は、*「JConfiguration Tool参照編」*をご覧下さい。
- DBMasterの機能についての詳細は、*「JDBA Toolユーザーガイド」*をご覧下さい。
- DBMasterで使用しているdmSQLのインターフェースについての詳細は、*「dmSQLユーザーガイド」*をご覧下さい。
- DBMasterで採用しているSQL言語についての詳細は、*「SQL文と関数参照編」*をご覧下さい。
- ESQLプログラムについての詳細は、*「ESQL/Cプログラマー参照編」*をご覧下さい。
- ODBCとJDBCプログラムについての詳細は、*「ODBCプログラマー参照編」*と*「JDBCプログラマー参照編」*をご覧下さい。
- エラーと警告メッセージについての詳細は、*「エラー・メッセージ参照編」*をご覧下さい。
- ネイティブDCI APIについての詳細は、*「DCI ユーザーガイド」*をご覧下さい。
- SQLストアドプロシージャ言語の詳細については、*「SQLストアドプロシージャ参照編」*を参照して下さい。

1.2 字体の規則

本書は、標準の字体規則を使用しているので、簡単かつ明確に読むことができます。

斜体	斜体は、ユーザー名や表名のような特定の情報を表します。斜体の文字そのものを入力せず、実際に使用する名前をそこに置き換えてください。斜体は、新しく登場した用語や文字を強調する場合にも使用します。
太字	太字は、ファイル名、データベース名、表名、カラム名、関数名やその他同様なケースに使用します。操作の手順

	においてメニューのコマンドを強調する場合にも、使用します。
キーワード	文中で使用する SQL 言語のキーワードは、すべて英大文字で表現します。
小さい英大文字	小さい英大文字は、キーボードのキーを示します。２つのキー間のプラス記号 (+) は、最初のキーを押したまま次のキーを押すことを示します。キーの間のコンマ(,)は、最初のキーを放してから次のキーを押すことを示します。
注	重要な情報を意味します。
➲ **プロシージャ**	一連の手順や連続的な事項を表します。ほとんどの作業は、この書式で解説されます。ユーザーが行う論理的な処理の順序です。
➲ **例**	解説をよりわかりやすくするために与えられる例です。一般的に画面に表示されるテキストと共に表示されます。
コマンドライン	画面に表示されるテキストを意味します。この書式は、一般的に dmSQL コマンドや dmconfig.ini ファイルの内容の入/出力を表示します。

2 概要

データベースを構成する物理ファイルのデータ編成は非常に複雑です。DBMS は、データのビューをコンピュータ上のデータベースの実装から切り離し、データベースを２次元の表の集まりと見ます。表の行およびカラムにデータの値があります。表は容易に視覚化され、データを柔軟にモデル化します。

DBMaster はデータ検索方法を何通りか提供します。日々のトランザクション処理や問合せには会話型 SQL を用い、アプリケーションを素早く容易に開発するには DBMaster のアプリケーション・プログラム・インタフェース(API)を用います。どのプラットフォームでも簡単に使用できる GUI ツールもあります。

2.1 特徴

リレーショナル・データベース管理システムとして、DBMaster は従来のデーータベース管理システムのすべての特性を継承すると同時に、多くの強力且つ先進的な機能を持っています。これらの付加機能は、単に DBMaster の操作性能を向上させるだけでなく、従来のデータベース管理システムには無い機能、特にマルチメディア・サポート分野の機能も提供します。

マルチメディアサポート

強力なマルチメディア管理機能がデータベース・エンジンに組み込まれており、テキスト、画像、音声、ビデオ、アニメーションを含む大量のマルチメディア・データを効率的に格納、操作します。マルチメディア管理機能は、ユーザが満足するために異なる方法でマルチメディア・データを格

納する柔軟性も提供します。マルチメディア機能には以下のものが含まれます：

- バイナリ・ラージオブジェクト(BLOB)とファイルオブジェクト(FO)
- 表内の複数のBLOBおよびFOカラム
- ファイルオブジェクトは既存のマルチメディア・ツールで編集可能
- 組み込みの全文検索エンジン

マルチメディア・データは、バイナリ・ラージオブジェクト(BLOB)として直接データベースに格納することができます。データは、在来型のデータタイプに用意されている安全性、信頼性、整合性によって全面的に保護されます。また、ファイルオブジェクトとして格納することにより、データをデータベース管理下に置きながら、第三者マルチメディアツールからのアクセスも可能にします。

64BIT対応

DBMaster は 64bit ポーティングの Windows x64 及び Linux x64 OS に対応しました。x86-64 アーキテクチャ CPU 上の 64bit Windows または Linux 環境に適切な 64bit 版 DBMaster をインストールする必要があります。

64bit 版では以下の制限があります。

- 32/64bitで作成されたデータベースは64/32bitのデータベースサーバでは起動できません。ストアドプロシージャやユーザ定義関数なども同様で互換性がありません。
- 32/64bitクライアントが64/32bitデータベースサーバに接続可能とはいえ、異なるOSアーキテクチャ下で使用する場合にはデータベースを移行する必要があります。
- UDF、ESQL/C、ストアドプロシージャのコンパイルとビルドには64bit Cコンパイラが必要です。.NETアプリケーションにはVS2005または64bitアプリケーションへコンパイルとリンクが必要です。JDBCまたはJAVAストアドプロシージャでは64bit JVM環境でJAVAプログラムをコンパイルします。

- DBMasterの共有メモリサイズは64bitマシンで2Gページ（或いは2G* PAGE SIZEバイト）です。32bitマシンで2Gバイトです。

JDBCサポート

DBMaster は、Java トランザクション API(JTA)機能と同様に JDBC 3.0 の特徴をサポートします。JDBC JTA は、BEA WebLogic™のようなポピュラーな Java AP サーバーへの接続しやすくなります。

JDBC と JDBA のインプリメントについて知るためには、製品ドキュメンテーションを参照してください。JDBC 仕様に関する情報は次のサイトで入手可能です：http://java.sun.com/products/jdbc/

JTA 仕様に関する情報は次のサイトで入手可能です：
http://java.sun.com/products/jta/

MICROSOFTトランザクション･サーバー(MTS)サポート

MTS(Microsoft トランザクション・サーバー)は、Windows NT の不可欠な要素です。又、Windows のオペレーティング・システムの一部に初期設定でインストールされています。MTS は、CICS、Tuxedo 等の他のプラットフォームで利用できる Windows NT 同等の機能を提供するために、トランザクション処理システム(TP)として開発されました。これらは純粋に、データソースの安定した環境を築くために設計されています。

DBMaster は MTS を経由してのトランザクション操作をサポートしています。

MTS で DBMaster を使用する際に、以下のことをご注意ください。

- Microsoft Data Access Components (**MDAC**) 2.6以上のバージョンとMTSと共同作業する必要があります。下記のサイトで最新版のMDACをダウンロードすることができます：
  http://www.microsoft.com/data
- **MDAC**2.5を使用する場合、**dmconfig.ini** ファイルの **DM_COMMON_OPTION** セクションに**DM_DifEn = 0** を追加する必要があります。

オープンインタフェース

ODBC 3.0 互換インタフェースと ANSI SQL-99 サポートを使用することによって、高性能のアプリケーションを素早く作成することができます。Visual C++、Visual Basic、Delphi、AcuBench 等の多種多様の開発ツールを使用してアプリケーションを構築します。DBMaster は特定の開発環境の制約はなく、既存のツールを無料で使用することができます。以下のオープンインタフェースが含まれます。

- ANSI-SQL99 準拠
- ODBC 3.0サポート
- ESQL/Cプリプロセッサ
- JDBC2.0サポート

ESQL/C プリプロセッサは、従来の C 開発環境で書かれたプログラムの開発プロセスを単純化します。高水準の組み込み SQL 問い合せ言語のパワーを使ってデータベース・アプリケーションを記述すると、DBMaster プリプロセッサが自動的に適切な ODBC 関数コールに変換します。

データ整合性

DBMaster には、従来のデータ整合性の全ての機能が備わっています。データ整合性は、主キーと外部キーの参照アクションを全てサポートすることによって保証されます。ユーザー定義データ型は、ドメイン、カラム制約、表制約と共に、正当な値のみフィールドに格納されることを保証します。

以下のデータ整合性機能があります。

- 主キーと外部キーの整合性チェック
- 全ての参照アクションのサポート
- 表制約とカラム制約
- ユーザー定義データ型
- カラム初期値

データ信頼性

先進的データ保護機能より、データは常に安全に保たれます。特性は自動損傷リカバリ、データベース一貫性チェック、自動バックアップ等を含みます。これらの機能は、オペレーティング・システムやディスクに障害があっても、データの整合性と安全性を確実にします。

以下のデータ信頼性機能があります。

- オンライン・トランザクション処理
- オンライン完全および増分バックアップ
- 自動損傷リカバリ
- 自動増分バックアップ
- 自動統計更新
- データベース一貫性チェック
- 複数ジャーナルファイル
- オプションのBLOBバックアップ

ストレージ管理

DBMaster の最新のストレージ管理は、簡明な管理機能と構成機能をもつ柔軟なデータストレージを提供します。表の行数あるいはデータベース内の表数には、実用上の制限はありません。表を複数のディスクに跨らせることさえできます。DBMaster は、表スキーマをオンラインで変更することを許し、必要性に応じて動的に調整できるアプリケーションの開発も増強します。

以下のストレージ管理機能があります。

- 自動拡張表領域と標準表領域
- UNIXプラットフォームのローデバイスをサポート
- 最大データベースサイズは256PB（petabytes：1 PB = 1024TB）

- 表数は無制限
- レコード数は無制限
- 表スキーマのオンライン再定義/編集

データベースのストレージ領域は、ディスクの上限まで自動的に拡張することができます。また、固定サイズのストレージ領域を設定することもできます。UNIX プラットフォームではローデバイスをサポートし、ファイルシステムをバイパスし最大性能で直接ローデバイスに書き込むことができます。

セキュリティ管理

DBMS は中央化したマルチ・ユーザーシステムなので、無認可のアクセスを防止し、ユーザーのアクセスを限定する何らかのセキュリティ管理が必要になります。ユーザーおよびグループレベルのセキュリティ権限は、データベースをアクセスする人を管理し、一方、表および個々のカラム権限は、ユーザーがアクセスする物を管理します。

以下のセキュリティ管理機能があります。

- ユーザーレベル、グループレベルのセキュリティ
- グループの階層
- 表と個々のカラムの権限管理
- ストアドコマンド、ストアドプロシージャの権限管理
- ネットワーク接続の暗号化

先進言語機能

先進的言語機能が伝統的なデータベース機能を補完します。ストアドコマンド、ストアドプロシージャ、トリガー、ユーザー定義関数を使用して、DBMaster の機能を容易に拡張しカスタマイズすることができます。ビジネス規則を直接データベースエンジンに組み入れ、ロジックをデータベース内に中央化し、保守管理しやすくします。

以下の先進的言語機能が含まれます：

- 組み込み関数
- ユーザー定義関数
- ストアドコマンド
- ストアドプロシージャ
- トリガー

2.2 データベースのモード

DBMaster には、データベースを起動させるときのモードが幾つかあります。各モードには、データベースに接続しアクセスする種々のオプションがあり、一つのコンピュータ上のシングルユーザー・システムから、複数のコンピュータに跨って分散する大規模マルチユーザー・システムまで、データベースをスケール化することができます。

データベース・サーバーが走行するプラットフォームにより、使用できるモードが異なります。DBMaster には、シングルユーザー、マルチユーザー、クライアント/サーバーの 3 つのモードがあります。

シングルユーザー・モード

シングルユーザー・モードは、UNIX と Linux プラットフォームでのみ使用することができます。このモードは、DBMaster を個人用データベースに単純化したバージョンです。このモードの主な利点は、ロック、セキュリティ、ネットワークのサポートが不要になり、アプリケーションサイズを小さくし、データベースオペレーションの実行速度を速くすることができることです。

このモードの制限は、シングル接続がデーターベースに確立できますが、ほかのサーバ或いはデーモンを実行できません。（例：バックアップ・サーバー、レプリケーション・サーバー、グローバル・トランザクション・サーバー。別の制限としては、ネットワークを使用することができないので、データベースは、ホストマシンのみからアクセスしなければなりません。

マルチユーザー・モード

マルチユーザー・モードは、Windows プラットフォームでのみ使用することができます。このモードの利点は、複数のデータベース接続が可能であり、DBMaster はセキュリティ、信頼性の機能を全て備えていることです。シングルユーザー・モードと同様、ネットワークをサポートしないので、すべてのデータベース接続は、ホストマシンからアクセスしなければなりません。

このモードの制限は、バックアップ・サーバー、レプリケーション・サーバー、グローバル・トランザクション・サーバー等の付属サーバーやデーモンを走らせることができないことです。

クライアント/サーバー・モード

クライアント/サーバー・モードは、すべてのプラットフォームで使用することができます。TCP/IP ネットワーク経由でホストコンピュータに接続することができるどのコンピュータからでも、データベースに複数接続することができます。DBMaster は、セキュリティ、信頼性、同時実行制御のすべての機能を備えています。更に、ネットワークを通るデータを暗号化することもできます。このモードは、バックアップ・サーバー、レプリケーション・サーバー、グローバル・トランザクション・サーバー等の付属サーバーやデーモンをすべてサポートします。

注　*一つの接続は同時に複数述語の実行が支持できません。*

2.3 DBMaster インターフェースとツール

DBMaster は、データベース管理に必要なアプリケーション・プログラム・インタフェースと種々のツールおよびデーターベース管理のユーティリティを全て備えています。ツールは、会話型 SQL 問い合せのコマンドラインツールから、上は、複数サーバーを管理するグラフィカルインタフェースツールにまで広がります。データベースに経験のない利用者にも、プラットフォーム間で一貫している簡明な管理機能とグラフィカルなツールを高く評価いただけると思います。

アプリケーション・プログラム・インタフェース

API は、直接データベースエンジンに作用する低水準ルーチンのライブラリです。API は、通常、C++や Visual Basic のような汎用プログラミング言語を用いてアプリケーション・ソフトウェアを開発するときに使用します。DBMaster は、ODBC 3.0 互換インタフェースを提供し、すべての中核レベルの関数と大部分の拡張レベルの関数をサポートしています。

DMSQLインターラクティブ問い合せツール

dmSQL は、DBMaster の全ての能力と機能を直接使用することができる文字ベースのインターラクティブ・インターフェースです。dmSQL を使用すると、データベースを操作し、問合せを行い、直ちに結果セットが確認できます。dmSQL は、在来型のプログラミング言語を用いてプログラムを作成せずに、データベースの全機能を有効活用できるツールです。

JDBA TOOL

JDBA Tool は、データベースを維持、監視するための交換グラフィカルで会話型のツールです。JDBA Tool は、DBMS の複雑さと問合せ言語を隠した、直感的で覚えやすく便利なインターフェースです。このツールを使えば、熟練していなユーザーでも問い合わせ言語を学習することなく、データベースにアクセスできるようになり、上級ユーザーは SQL を使った形式的なコマンドを入力する煩わしさを感じること無く、データベースをより速く管理、操作できるようになります。JDBA Tool には統計データや、監視機能を使って、誰がデータベースを使用しているかといった情報も見ることができます。

JSERVER MANAGER

JServer Manager は、データベースの作成、起動、バックアップ、リストアを行う直感的でグラフィカルなツールです。JServer Manager は、一度に全データベース・サーバーを作成、管理する中心的な役割を果たします。

JCONFIGURATION TOOL

JConfiguration Tool は、全データベースの環境設定のパラメータを管理するグラフィカルなツールです。このツールは、DBMaster の環境設定ファイルのキーワードを修正する簡単で直接的な方法を提供します。環境設定の各パラメータは、ユーザー・インターフェースの中に明確に定義されているので、マニュアルを参照したり、キーワードの定義を暗記したりする必要もありません。

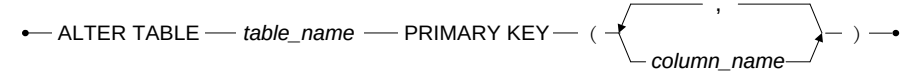
C言語のためのESQL

C 言語のための ESQL は、埋め込み SQL/C プログラムを編集し、プリプロセスに使用する、グラフィカルな双方向のツールです。複数の ESQL/C プログラムを管理し、それらを編集/プリプロセスするための使いやすいインターフェースを提供します。出力ウィンドウの各エラー文をクリックするだけで、プリプロセスの警告/エラーを検査することができます。

2.4 構文ダイアグラム

SQL 文の構文は、構文ダイアグラムで示します。構文ダイアグラムは、SQL 文を作成するときに、構文やすべてのオプションを覚えていないときの手助けになります。構文ダイアグラムの例を下に図示します。

構文ダイアグラムを使用するときは、始点から終点までの線をたどります。進路上にある SQL 文の要素は必要なものです。進路から外れた要素は選択するもので追加オプションや柔軟性を与えます。



斜体の字句は、データベースで使われている実際の名前を指定する場所です。この部分は、実際の名前で置き換えなければなりません。上のダイアグラムでは、表名 をデータベースにある表の名前で置き換えなければなりません。例えば、tutorial データベースでは、表名 を Customers に置き換え、この SQL 文を Customers 表に対して実行することができます。

矢印の向きにも注意する必要があります。SQL 文の中で項目のリストを与えることができるときには、構文ダイアグラムは、これを循環パスで示します。上のダイアグラムの矢印をたどって得られる循環パスが示すように、カラム名にはカンマで分離したカラム名リストも含まれます。

3 システムアーキテクチャ

この章では、シングルユーザー・モデルとクライアント/サーバー・モデルのアーキテクチャを詳細に説明します。まず DBMaster プロセスとデータベース*通信*制御域*(DCCA)*について解説します。DCCA は、起動した各データベースに必要なすべての情報を格納するエリアです。次に、2 つのモデルのアーキテクチャについて説明します。

3.1 DBMaster プロセス

DBMaster プロセスは、ユーザーが記述した SQL 文や他のデータベース機能に従って、データの格納と検索を処理します。DBMaster プロセスは、図3-1に示すいくつかの階層から構成されます。

アプリケーションは、図 3-1 のようにアプリケーション・プログラム・インタフェース(API)を通して DBMaster と通信します。API は、SQL 文または関数呼び出しを SQL エンジンに渡します。SQL エンジンは、SQL 文を解析しデータベース・エンジンが受理できる一連の関数呼び出しに変換します。データベースエンジンは、SQL エンジンから受け取った関数呼び出しを実行し、表にデータを格納、表からデータを回収します。

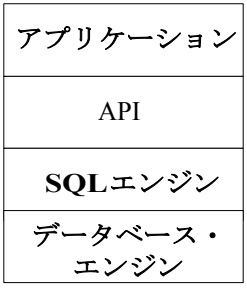


図 3-1 : DBMaster プロセス

SQL エンジンとデータベース・エンジンの役割は異なります。基本的に、SQL エンジンは、SQL の構文解析と問合せの最適化を扱います。一方、データベース・エンジンは、領域/バッファの管理、同時実行制御、障害リカバリ等々を処理します。すべてのモジュールが協調することで、データベースの一貫性が保たれます。パフォーマンスをチューニングするパラメータの多くは、データベース・エンジンに関係します。

シングルユーザーとクライアント/サーバーの 2 つのモデルは、API と SQL エンジンが同じように使用します。一方、データベース・エンジンは異なります。シングルユーザー・モデルが一人の利用者しか処理できないのに対して、クライアント/サーバー・モデルは複数の利用者を処理することができます。

クライアント/サーバー・モデルでは、クライアント側でアプリケーションと API が統合されて、サーバー側で SQL エンジンとデータベース・エンジンが統合されて運用します。API は、ネットワーク・プロトコルを通して SQL エンジンと通信します。

3.2 データベース通信制御域(DCCA)

DBMaster は、データベースを起動させると、まずデータベースのバッファプールや様々な種類の制御情報を格納する大きなメモリブロックを割り当てます。このメモリブロックをデータベース通信制御域(DCCA)と言います。DCCA は、3 種類のデータ、ページバッファ、ジャーナルバッファ、システム制御域*(SCA)*で構成されています。

DBMaster を操作する上で、特にクライアント/サーバー・モードで走行する場合、DCCA は非常に重要です。Microsoft Windows 環境および UNIX のシングルユーザー環境では、専用ヒープに DCCA を割り当てます。一方、UNIX のクライアント/サーバー環境では、専用ヒープに DCCA を割り当てることはできません。代わりに、UNIX の標準機能である共有メモリ機構を使用して DCCA を割り当てます。クライアント/サーバー・モードで運用されている全ての DBMaster プロセスは、DCCA を通じて相互に通信します。

DBMaster では、DCCA のサイズと使用を簡単にチューニングすることができます。DCCA をチューニングすると、DBMaster の全般的な性能に大きく影響します。DCCA の詳細については、18 章の「パフォーマンスのチューニング」で説明します。

3.3 シングルユーザー・モードのアーキテクチャ

シングルユーザー・モードは、1 ユーザーのみをサポートする DBMS です。同時実行制御の必要がないので、他のモデルよりも小さく高速になります。一人でデータベースを使用する場合は、シングルユーザー・モードがデータベース管理にとって最適です。図3-2 は、シングルユーザー・モードの DBMaster システム・アーキテクチャを示しています。

1 ユーザーのみがシングルユーザーのデータベースに接続することができるので、DCCA は専用ヒープから取得し、共有はしません。シングルユーザー・モードにはロックの機構はありません。データベース・エンジンは、パフォーマンスを上げるために、データベース走行中にデータベースの全データをメモリに保持します。適切なときに、変更したページをディスク（データファイルとジャーナルファイル）に書き戻します。**dmconfig.ini** ファイルは、DBMaster 自身の環境設定に必要な多くのパラメータを定義するためのテキストファイルです。

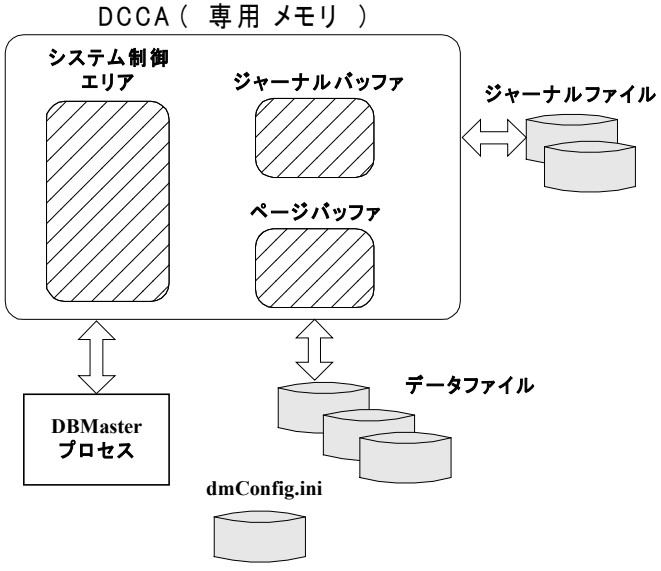


図 3-2：シングルユーザー・モード DBMaster のシステムアーキテクチャ

3.4 クライアント/サーバー・モードのアーキテクチャ

DBMaster はクライアント/サーバー・モードでも使用できます。このモデルでは、クライアント・アプリケーション・プロセスとデータベース・サーバー・プロセス（サーバー・プロセスとも呼ばれます）の 2 つのプロセスを含みます。典型的にクライアント・プロセスは、フロントエンド PC またはワークステーション側にあり、DBMaster の API ライブラリルーチンを使用して、サーバー・プロセスと通信します。サーバプロセスはローカルエリアネットワークの一部分として位置づけられます。クライアント/サーバー・構成での注意すべき点は、サーバーとクライアントを含むすべてのコンピューターに異なる種類のプラットフォームを採用できるということです。

DBMaster のクライアント/サーバー・モードにはネットワーク管理モジュールが含まれます。クライアント側とサーバー側の両方にネットワーク管理モジュールがインストールされます。ネットワークマネージャは、クライアントとサーバー間のデータ送信の役割を担います。クライアント/サーバー・モデルの重要な問題の一つは、ネットワークプロトコルです。DBMaster は、普通、TCP/IP(Transmission Control Protocol/Internet Protocol)ネットワーク・プロトコルのみをサポートしています。DBMaster のクライアント/サーバー・バージョンを走行させるときは、TCP/IP ネットワーク・ソフトウェアをインストールしておきます。UNIX、Windows のようなオペレーティング・システムには、TCP/IP が組み込まれているので、追加インストールする必要はありません。Windows 場合は、ネットワークに TCP/IP を選択し、それをシステムにインストールするだけです。図3-3 は、クライアント/サーバー・モードの DBMaster システム・アーキテクチャを示します。

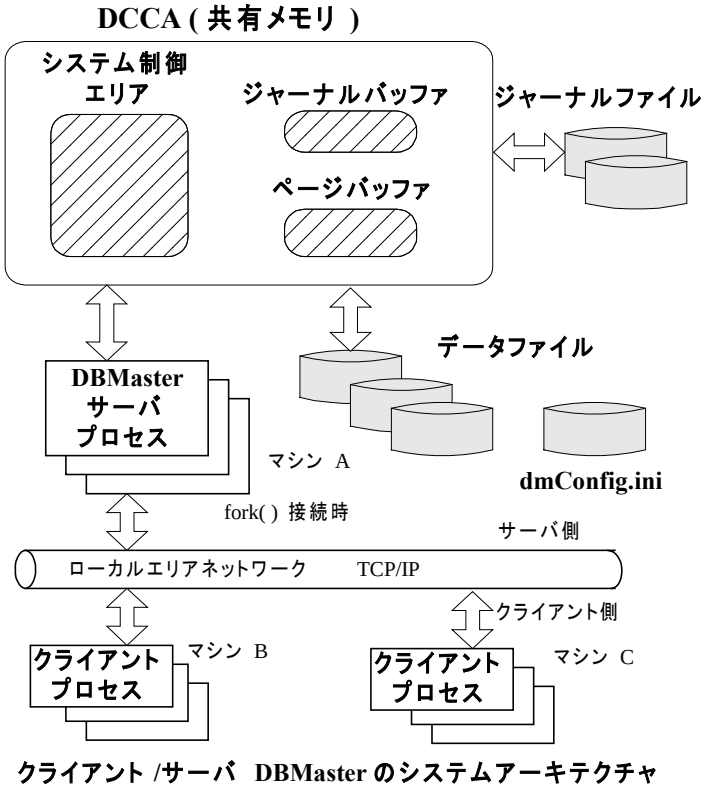


図 3-3 : クライアント/サーバー・モデルの DBMaster システムアーキテクチャ

UNIX システムでは、クライアント・プロセスがデータベース・サーバーに接続すると、DBMaster のネットワーク・サーバーは、別のネットワーク・サーバーへ分岐し、後に続く問合せを処理します。元のネットワーク・サーバー・プロセスは、別のクライアントからの接続を待ち続けます。Windows NT では、少し違うシナリオになります。

NT はマルチスレッド・システムなので、NT 版の DBMaster ネットワーク・サーバー(dmserver.exe)も、マルチスレッド・プログラムになります。このため、NT で走行中のサーバーにクライアント・プロセスが接続すると、DBMaster サーバー・プロセスは、プロセス領域に別のスレッドを作成して後に続く問合せを処理します。この場合、DCCA は、共有メモリではなく

専用メモリに割り当てられます。Windows NT のデータベースには、常に DBMaster サーバー・プロセスは 1 つしか存在しません。マルチスレッドをサポートするオペレーティング・システムは更に増えています。これまで調査では、マルチスレッド・プログラムの方がマルチプロセス・プログラムよりも効率が良いことが示されているので、DBMaster では可能な場合、マルチスレッドを使用します。

クライアント/サーバー・モデルには、3 つの構成要素、サーバー・プログラム、クライアント・プログラム、クライアント・ライブラリがあります。

サーバー・プログラム

DBMaster のサーバー・プログラム名は、**dmserver** です。サーバー・プログラムには、ネットワーク通信を扱うネットワークマネージャとデータ・アクセスを行うデータベース・エンジンが含まれます。このプログラムを最初に起動させなければ、クライアント・プログラムはデータベース・サーバーに接続できません。

クライアント・プログラム

DBMaster の SQL クライアント・プログラム名は、**dmsqlc** です。このクライアント・プログラムを使用してデータベースに接続し、SQL 文を与えてデータを処理します。

クライアント・ライブラリ

DBMaster のクライアント・ライブラリ名は、UNIX では **libdmapic.a**、Windows では **dmapi*<version>*.lib** です。クライアント・プログラムを開発したいユーザーは、そのプログラムをこのライブラリにリンクさせます。例えば、フロントエンド・アプリケーションを書くときに、ベンダーから提供されている様々な開発ツールを使用することができますが、アプリケーションを構築する時点で、これらのプログラムをライブラリとリンクして、アプリケーションがサーバー・プログラムと通信できるようにしなければなりません。

4 基本データベース管理

本章は、データベースの作成、データベースの起動、データベースの接続、データベースの終了といったデータベースの基本管理について解説します。データベース管理者は、これらの操作を dmSQL の操作と **dmconfig.ini** ファイルの編集で実行することもできますが、JConfiguration Tool と JServer Manager ユーティリティを使うことも可能です。

以下の節では、環境設定パラメータと基本データベース管理に必須のコマンドについて説明します。第 1 節では環境設定ファイルの役割とそのフォーマットについての要約です。2 節以降は、特定の設定の役目とこれらの設定がデータベース作成、起動、接続後のデータベース・パフォーマンスに与える影響について解説します。

4.1 環境設定ファイル - dmconfig.ini

DBMaster の操作には、データベース起動時に必要な環境設定パラメータがたくさんあります。これらのパラメータは、どのようにデータベースを起動するかを指定するためにデータベース・エンジンによって使用されます。ファイル格納ディレクトリ、ランタイム・メモリ割り当て、ネットワーク接続などは、環境設定パラメータを使用してセットするデータベースの機能のいくつかにすぎません。これらのパラメータは、**dmconfig.ini** ファイルに環境変数として保存されています。環境変数は、各キーワードにセットされる値と同じです。(本節の「dmconfig.ini のフォーマット」を参照)。ユーザーは、**dmconfig.ini** ファイルにあるパラメータをセットして、或いは JConfiguration Tool でデータベースをカスタマイズすることができます。JConfiguration Tool は、使いやすいグラフィカル・ユーザーインターフェー

スで環境設定パラメータの管理を単純化します。JConfiguration Tool についての詳細は、*「JConfiguration Tool 参照編」*をご覧下さい。パラメータ（キーワード）によっては、データベース作成時にセットする必要があります。その他については、データベース起動前にセットします。加えて、ある環境設定パラメータは、データベース作成後に変更することができず、エラーになります。以下の節では、環境設定ファイルのキーワードを直接編集して設定を管理する方法について説明します。**dmconfig.ini** にあるキーワードの完全な一覧については、本書の **dmconfig.ini** のキーワードを参照してください。

DMCONFIG.INIのディレクトリ

UNIX プラットフォームでは、DBMaster は以下の 3 つのロケーションの **dmconfig.ini** ファイルを順に検索します。

- カレントディレクトリ
- 環境変数DBMASTERで指定されるディレクトリ
- DBMaster のインストールディレクトリ**~DBMaster/Version**

DBMaster は、各ディレクトリを順に検索します。あるディレクトリにある **dmconfig.ini** ファイルに関連のデータベース・セクションが見つからない場合、DBMaster は次のディレクトリを検索します。

Microsoft Windows システムの場合は、異なります。DBMaster は以下の位置に **dmconfig.ini** ファイルのみを検索します。

- DBMaster環境変数から指定されたディレクトリ
- DBMasterのインストールディレクトリ。典型的なWindowsインストレーションの場合、このディレクトリはC:\DBMaster\Versionにあります

あるデータベースの環境設定パラメータの値が必要な場合、DBMaster は上記の 3 ディレクトリ（Microsoft Windows システムでは、インストールディレクトリ）を検索し、データベース名と同じセクション名をもつ **dmconfig.ini** ファイルを見つけます。このファイルは、テキストエディタで編集することができます。**dmconfig.ini** のファイルで追加、修正されたパラメータは、データベースを運用する際に参照されます。

データベース作成時に、いずれの **dmconfig.ini** ファイルにも対応するデータベース・セクションが無い場合、DBMaster は最初に見つかった **dmconfig.ini** にデータベース・セクションを作成します、。或いは、ローカル・ディレクトリ（Microsoft Windows の**インストール**ディレクトリ）に新しい **dmconfig.ini** ファイルが作成され、同様にデータベース・セクションが設定されます。

データベース起動時には、**dmconfig.ini** ファイルと対応するセクションが必ず存在します。存在しない場合は、エラーになります。別々の **dmconfig.ini** ファイルに様々なデータベース・セクションを設けたり、別々のディレクトリに複数の **dmconfig.ini** ファイルを配置したりすることも可能ですが、理想的な方法とはいえません。1 つの **dmconfig.ini** ファイルを使用するほうが、管理がより容易になります。

JConfiguration Tool は、**dmconfig.ini** ファイルにある全データベース・セクションを表示します。UNIX システムでは、JConfiguration tool は上述のディレクトリに示される全 **dmconfig.ini** ファイルの全セクションを表示します。

DMCONFIG.INIのフォーマット

dmconfig.ini ファイルは、セクションに分かれています。各セクション名は対応するデータベース名を表します。各セクションの下にあるキーワードは、データベースの環境設定を定義します。セミコロン以降の任意の文字列はコメントとみなされます。

➲ 例

dmconfig.ini ファイルのフォーマット:

```
[セクション名 1]
<キーワード 1> = <値 1>
<キーワード 2> = <値 2> <値 3> ; これはコメントです
; これはコメントです
...

[セクション名 2]
<キーワード 3> = <値 4> <値 5>
<キーワード 4> = <値 6>
```

ファイル名とサイズ

データベースは、オペレーティング・システムのファイルで構成されています。これらのファイルは、dmconfig.ini ファイルでキーワードを使って定義されています。<ファイル名>パラメータは、<値>パラメータの場所で使用します。<ファイル名>パラメータは、**firstdb.sdb** のような純粋なファイル名か、**mydb/firstdb.sdb** のような相対パスか、**/disk1/mydb/firstdb.sdb** のような絶対パスです(UNIX の場合は"/"、Microsoft Windows の場合は"\")。

<np>パラメータは、ページ数を表します。.ページサイズは DB_PgSiz キーワードで指定されていなければ、初期値は 8KB です。

ページ多く使用する場合には、ユーザはユニットに M (メガバイト) 或いは G (ギガバイト) を指定することができます。ユニットとしての M 或いは G が使用されていなければ、ユニットはページとなります。ユーザは M 或いは G で値を指定した場合、実際のサイズは指定されたサイズより 1 ページ分を少なくなります。例えば、ページサイズが 16 KB で、8M B を指定した場合、サイズは 8192 KB ではなく、8176KB となります。

➲ 例

ファイル名とサイズを表すフォーマットの一部:

```
[セクション名 1]
<キーワード 1> = <ファイル名>
<キーワード 2> = <ファイル名> <ファイル名>
<キーワード 1> = <np>
```

ファイルのディレクトリ

DBMaster のプログラムを実行しているユーザーが別々のディレクトリからアクセスする場合(ユーザー毎に「カレント・ディレクトリ」が異なる場合)、**dmconfig.ini** のファイル名は、全て絶対パスにする必要があります。

或いは、**DB_DbDir** 環境設定パラメータを使用します。このキーワードは、データベースの「ホーム・ディレクトリ」(またはデータベース・ディレクトリ)を指定します。

➲ 例 1

データベース・ディレクトリ名を、セクション名に表される初期設定 DB1 の替わりに db にセットします。更に、他のデータベース・ファイルは入れ替えたディレクトリや他のディスクに配置されます。

```
[DB1]
DB_DbDir = /disk1/db
DB_DbFil = mydb1
DB_JnFil = /disk2/usr/DB1.JNL
```

物理ファイル名は、以下のようになります。

```
DB_DbFil -- /disk1/db/mydb1
DB_JnFil -- /disk2/usr/DB1.JNL
DB_BbFil -- /disk1/db/DB1.SBB (using default file name)
```

➲ 例 2

DB_DbFil キーワードを使う:

```
[DB2]
DB_DbFil = mydb2
DB_JnFil = /disk2/usr/DB2.JNL
```

物理ファイル名は、以下のようになります。

```
DB_DbFil -- mydb2 (in current directory)
DB_JnFil -- /disk2/usr/DB2.JNL
DB_BbFil -- DB2.SBB (in current directory)
```

注　*この規則は、ユーザー定義ファイルにも適用されます。*

DMCONFIG.INIの重要なキーワード

以下に dmconfig.ini にある重要なキーワードを説明します。データベース作成とデータベース起動に必須のキーワードは、本章の後節で紹介します。dmconfig.ini のキーワードで全てのキーワードについて説明します。

- DB_DbDir=<ファイル名>—データベースが存在するディレクトリを指定します。
- DB_DbFil=<ファイル名>—<ファイル名>でシステム・データベース・ファイルのファイル名を指定します。

- **DB_PGSIZ=<4, 8, 16, 32>**—ページサイズを指定します(4KB, 8KB,16KB, または32KB)
- **DB_JnFil =<ファイル名>**—ジャーナルファイル名を指定します。
- **DB_JnlSz =<ページ数>**—ジャーナルファイルのサイズを指定します。
- **<論理ファイル>=<ファイル名> <ページ数>**—ユーザー定義の**<論理ファイル>**名にマップする物理 **<ファイル名>**と**<ページ数>**を指定します。
- **DB_NBufs =<ページ数>**—実行時のバッファサイズを指定します。
- **DB_SvAdr =<IPアドレス>**または**<ホスト名>**—データベース・サーバーのIPアドレスまたはホスト名を指定します。C/Sシステムでは、クライアント側でこのオプションを指定しなければなりません。
- **DB_PtNum =<ポート番号>**—クライアントとデータベース・サーバーの間の通信に使用するTCP/IPポート番号を指定します。
- **DB_MaxCo**=<数値>—データベースが扱うことができる最大接続数を指定します。

注 *パターンマッチは全て、〈論理ファイル〉を除いて、大文字と小文　　字を区別しません。*

DMCONFIG.INIの初期設定値

キーワードによっては、初期設定があります。**dmconfig.ini** の中でキーワードが設定されていない場合は、その初期設定値が採用されます。各キーワードの詳細と初期値については、**dmconfig.ini** のキーワードを参照してください。

環境変数をサポート

CD-ROM に保存された読み取り専用データベースに対して、ユーザーは **dmconfig.ini** ファイルにパスを指定しにくいです。,もし DBMaster はディフォルト環境変数**$APP_HOME** と**$APP_DRIVE** をサポートできるとこの問題は簡単になります。

- **$APP_HOME**: DBMaster ホームインストールディレクトリです。これはレジスターからDBMaster ホームHOME 情報が取れます。例えば、DBMaster がディレクトリ**D:\dbmaster\5.4**にインストールされると、**dmconfig.ini**ファイルを読み取る場合にDBMaster は自動に**$APP_HOME**を"**D:\dbmaster\5.4**"に替わります。
- **$APP_DRIVE**: この変数はLinuxシステムに空のストリングを返し、Windows システムのドライバレターにホームインストールディレクトリを返します。もしDBMaster がディレクトリ**D:\dbmaster\5.4**にインストールされると、**dmconfig.ini**ファイルを読み取る場合にDBMaster は自動に"**D:\dbmaster\5.4**"を"**D:**"に替わります。

DBMaster はシステム環境変数もサポートします、例えば操作システム環境変数に**$ TEMP = "C:\TEMP"**を定義された後、**dmconfig.ini** ファイルを読み取る場合に DBMaster は自動に**$ TEMP** を**"C: \ TEMP"**に替わります。

ディフォルト環境変数**$ APP_HOME** 或いは**$ APP_DRIVE** はシステム環境変数に定義されると、**dmconfig.ini** ファイルを読み取る場合に DBMaster はシステム環境変数にこの定義値が見つかりません、そうすると、システム環境変数値ではなく、ディフォルト環境変数値を使用します。

➲ **例**

ユーザーは CD-ROM があると、以下の設置によって、DBMaster ソフトウエアとデータベースを CD-ROM に置きます：

```
dmconfig.ini
[DBSAMPLE5]
DB_DBDIR=$APP_DRIVE\database
DB_FODIR=$APP_DRIVE\database\fo
DB_TPFIL=$TEMP\DBSAMPLE5.tmp
DB_SMODE=6
```

DMCONFIG.INIのサンプル・ファイル

次の例では、**dmconfig.ini** ファイルには 2 つのセクションが定義されています。1 つは ***Personnel*** データベースで、もう一方は ***LIBRARY*** データベースです。

➲ 例

典型的な dmconfig.ini：

```
[Personnel]
DB_DbFil = /disk1/bin/PERSONNEL.DB
DB_JnFil = /disk1/bin/PERSONNEL.JNL
f1.os = /disk1/bin/PERSONNEL.OS 100
f1.blob = /disk1/bin/PERSONNEL.BLOB 1000

DB_NBufs = 0              ; データバッファの個数
DB_NJnlB = 100            ; ジャーナル・バッファ数
DB_MaxCo = 100            ; 最大接続数
DB_JnlSz = 2000           ; ジャーナルファイルのサイズ(ページ)
DB_RTime = 0              ; リストア・ターゲット時間
DB_SvAdr = 192.72.116.130 ; サーバーの IP アドレス
DB_PtNum = 21000          ; サーバーのポート番号

[LIBRARY]
DB DbFil=/disk3/usr/lib/library.db
DB JnFil=/disk3/usr/lib/library.jnl
DB SvAdr = 192.72.116.137
DB PtNum = 26999
DB_JnlSz = 2000
```

4.2 データベースを作成する

新しいデータベースを作成する前に、まず計画をたてる必要があります。データベース作成前に、考慮すべき環境設定パラメータがいくつかあります。パラメータによっては、データベース作成後に変更できません。データベース作成時に設定する必要があるパラメータは以下のとおりです。

- データベース名
- 大文字と小文字の識別 (データベース内のあるスキーマ・オブジェクトを大文字と小文字を識別するかどうかを決定する)
- BLOBフレームサイズ (各BLOBフレームに割り当てるディスク領域のサイズ)

- 言語設定 (使用する文字セットを決定- ASCII、Big5等)
- 言語コードオーダー (文字型データのソートに使用するパターン)

その他の環境設定パラメータは、データベース作成後にも変更することができますが、データベース作成前に検討しておくことが大切です。そのパラメータには、以下のようなものがあります。

- 表領域名、ディレクトリ、サイズ、拡張性
- ジャーナルファイルの数
- ジャーナルファイル名、サイズ、ディレクトリ
- システムデータとBLOBファイル名、サイズ、ディレクトリ
- 初期設定ユーザーデータとBLOBファイル名、サイズ、ディレクトリ
- システム一時ファイル名とディレクトリ
- ユーザー定義ファイル名、サイズ、ディレクトリ
- DBMasterのログファイルのディレクトリ
- バックアップのディレクトリ
- 表レプリケーション・ログのディレクトリ
- ユーザー・ファイル・オブジェクトの使用
- ローデバイスの使用を可能にする(UNIXプラットフォームでのみ)
- クライアント/サーバー・データベースにする
- データベースIPアドレスとポート番号 (クライアント/サーバー・データベース)
- 初期設定ユーザーIDとパスワード
- メモリ割り当て

DBMaster には、ウィザードを使って簡単にデータベースを作成することができるツール JServer Manager がありますが、データベース管理者は、**dmconfig.ini** ファイルを編集し、dmSQL を使ってデータベースを作成するこ

ともできます。以下の節は、データベース作成の概要について解説します。JServer Manager のデータベース作成ウィザードの作成手順とほぼ同じです。

データベースのネーミング

データベースに名前を付ける前に、以下の規則について理解して下さい。

- データベース名の長さは、128文字以下。
- データベース名には、アンダースコア(_)を含む英数字を使用します。
- 日本語や中国語のようなダブルバイト文字は使用できません。
- 文字の並びに制限はありません。
- データベース名は大文字と小文字を識別しません。
- データベース名は、データベースに接続する全コンピュータ内で一意のものでなければなりません。

JServer Manager のデータベース作成ウィザード、或いは dmSQL を使ってデータベースに名前を付けることもできます。

➲ **例**

dmSQL を使ってデータベースを作成する:

```
dmSQL> create db <database name>;
dmSQL> terminate db;
dmSQL> quit;
```

スキーマ・オブジェクト名の大文字と小文字を識別する

データベース内の全識別子の大文字と小文字を識別するかを指定することができます。大文字と小文字を識別しない場合、全識別子は大文字になります。一旦データベースが作成されると、この設定を変更することはできません。キーワードをゼロに設定すると、データベースは大文字と小文字を識別します。初期設定の 1 に設定すると、大文字と小文字を識別しない設定でデータベースが作成されます。次の **dmconfig.ini** の変数は、データベースで大文字と小文字を識別するかどうかを指定します。

DB_IDCap = <値> (初期設定値 = 1)

ストレージ・パラメータの設定

シングル・データベースに関係のあるオペレーティング・システムのファイルには 10 種類あります。システムデータ・ファイルとシステム BLOB ファイル、初期設定ユーザーデータ・ファイルと初期設定ユーザーBLOB ファイル、システム・ジャーナル・ファイル、システム一時ファイル、ユーザー定義ファイル、DBMaster ログファイル、バックアップ・ファイル、表レプリケーション・ログファイルです。データベースを最初に作成した際、ユーザーは各ファイルに名前と場所を割り当てます。又は初期設定値が割り当てられます。データベース作成前に、これらファイルのデータベースでの役割を理解することは重要です。

本節で説明するパラメータの多くは、JConfiguration Tool のストレージのページで修正することができます。データベースのパラメータを変更する方法についての詳細は、*「JConfiguration Tool 参照編」*をご覧下さい。ファイルの管理についての詳細は、5.2節の*「ファイルの種類」*を参照して下さい。

データベース作成時、**dmconfig.ini** ファイルの関連設定に基づき、システム・データファイル、ジャーナル・ファイル、システム BLOB ファイルが生成されます。**DB_DbFil**、**DB_JnFil**、**DB_BbFil** が指定されていない場合、初期設定値になります。

初期設定値は以下のとおりです。

DB_DbFil – データベース名 + '.SDB'

DB_JnFil -- データベース名 + '.JNL'

DB_BbFil -- データベース名 + '.SBB'

データベース・ディレクトリの指定

データベース・ディレクトリは、データベースに関係するファイルを作成し、保存する初期設定の場所です。ファイルを絶対パスで指定した場合、そのパス名でファイルを参照します。ファイル名のみを指定した場合、DBMaster はデータベース・ディレクトリを探します。見つからない場合、

現在のディレクトリにあるものとみたし、そのファイル名のみを参照します。データベース・ディレクトリを指定するために、**dmconfig.ini** の以下のキーワードが使用されます。

DB_DbDir = <パス名> (初期設定: <DBMaster インストール・ディレクトリ>/bin/<データベース名>)

➲ **例**

データベース・ディレクトリを**/disk1/db** にセットする:

```
[DB1]
DB_DbDir = /disk1/db
```

システム表領域の作成

DBMaster のデータベースは、表領域と呼ばれるいくつかの論理域で構成されています。表領域を使うと、データベースを管理できる領域に分割することができます。論理ビューでは、表領域には最低 1 つの表と索引があります。物理ビューでは、表領域は 1 つ以上のファイルからなる物理ストレージです。新たに作成されたデータベースには、システム表領域と初期設定ユーザー表領域の 2 つの表領域があります。

システム表領域は、システム・データファイルとシステム BLOB ファイルからなります。システム表領域は、データベース全体のシステム表を記録するために使用します。データベース管理者は、システム表領域にシステム・データファイルとシステム BLOB ファイルの初期値を指定することができます。

ユーザー表領域を削除することはできますが、システム表領域は削除することができません。システム・データ・ファイルの初期サイズは、200 ページ(200*DB_PGSIZ KB)です。以下の **dmconfig.ini** のキーワードは、システム表領域を定義します。

システム・データ・ファイル: **DB_DbFil** = <ファイル名> (初期設定: " <データベース名>.SDB")

システム BLOB ファイル: **DB_BbFil** = <ファイル名> (初期設定: " <データベース名>.SBB")

<ファイル名>パラメータは、***firstdb.sdb*** のような単純なファイル名や、***mydb/firstdb.sdb*** のような相関パスや、***/disk1/mydb/firstdb.sdb*** のような絶対パスのいずれかです(UNIX では"/"、Microsoft Windows では"\"になります)。

➲ **例**

dmconfig.ini ファイルに次の行を入力すると、システム表領域は ***/disk1/mydb/***ディレクトリに保存されることになります。

```
DB_DbFil = /disk1/mydb/firstdb.sdb
DB_BbFil = /disk1/mydb/firstdb.sbb
```

ユーザー初期設定表領域の作成

初期設定ユーザー表領域は、最初 1 つのデータファイルと 1 つの BLOB ファイルで構成されています。ユーザー・データはこれらの表に保存されます。データベース管理者は、初期設定ユーザー表領域にデータファイルと BLOB ファイルの初期サイズと場所を指定することができます。データファイルのサイズは、ページ数で指定します(1 ページは 4K、8K、16K 、 32K に指定可能です)。BLOB ファイルのサイズは、フレーム数で指定します。フレーム・サイズは、ユーザーが指定し、この章の「BLOB フレーム・サイズの指定」で解説します。初期設定ユーザー表領域の初期設定は、自動拡張です。これは、表領域がデータで一杯になった場合、自動的にファイルが拡張される（つまり表領域が拡張される）ことを意味します。但し、ユーザー表を保存されるために追加の表領域を作成するほうがより柔軟で効果的です。

JDBA Tool は、新しい表領域を作成し、既存の表領域を管理するために役立つツールです。絶対パスを指定せずに、データファイルや BLOB ファイルが表領域に追加された場合、そのファイルはデータベース・ディレクトリに作成されます。他のユーザー表領域は削除することができますが、初期設定ユーザー表領域は削除することができません。**dmconfig.ini** にある次のキーワードは、初期設定ユーザー・データファイルとユーザーBLOB ファイルを指定します。

ユーザー・データファイル: **DB_UsrDb** = <ファイル名> (初期設定: " <データベース名>.DB")

ユーザーBLOB ファイル: **DB_UsrBb** = <ファイル名> (初期設定: " <データベース名>.BB")

<ファイル名> のパラメータは、***firstdb.sdb*** のような単純なファイル名か、***mydb/firstdb.sdb*** のような相関パスか、***/disk1/mydb/firstdb.sdb*** のような絶対パスのいずれかです(UNIX では"/"、Microsoft Windows では"\"になります)。

ジャーナル・ファイルの作成

ジャーナル・ファイルは、リアルタイムのデータベースへの全変更の履歴と各変更の状態です。8 つまでのジャーナル・ファイルを作成することができます。各ジャーナル・ファイルは、固定サイズです。全ジャーナル・ファイルがアクティブ・トランザクションで一杯になった時（例えば、トランザクションがコミットされず、占有されたジャーナル・ブロックを解放することができないが解放されない）、利用できるスペースがないので、現在のトランザクションは中止されます。これをジャーナル・フルといいます。最も長いトランザクションがジャーナル・ファイルの全ジャーナル・ブロックを使用しないようにして下さい。絶対パスを指定せずに、ジャーナル・ファイルを作成すると、データベース・ディレクトリに作成されます。データベース起動後にジャーナル・ファイルを修正することはできません。ジャーナル・ファイルの数を減らす、またはジャーナル・ファイルを追加する、或いはジャーナル・ファイルのサイズを変更する場合、新規ジャーナル・モードでデータベースを再起動します。新規ジャーナル・モードについての詳細は、4.3節の「データベースを起動する」を参照して下さい。また、ジャーナル・ファイルについての詳細は、5.2節の「ファイルの種類」をご覧下さい。以下の **dmconfig.ini** のキーワードは、ジャーナル・ファイル名、ディレクトリ、サイズを指定します。

ジャーナル・ファイル名: **DB_JnFil** = <ファイル名> (初期設定: "<データベース名>.JNL")

ジャーナル・ファイルのサイズ (ページ) **DB_JnlSz** = <サイズ> where サイズ =100 ページ～8G

➲ 例

次の **dmconfig.ini** ファイルの行は、**/mydb** ディレクトリの別々のディスクに各 500 ページの 2 つのジャーナル・ファイルを作るよう指定します。

```
DB_JnFil = /disk1/mydb/firstdb1.jnl /disk2/mydb/firstdb2.jnl
DB_JnlSz = <500>
```

システム一時ファイルの作成

システム一時ファイルは、データベースがアクティブの時、ソート結果のようなデータベースに関する情報を保管するために使用します。これらのファイルは必要なときに生成され、データベースの終了時に削除されます。絶対パスを指定せずに一時ファイルが作成された場合、データベース・ディレクトリに作成されます。8 つまでのシステム一時ファイルを指定することができます。各一時ファイルには 2 ギガバイトまで入れることができます。ディスク I/O パフォーマンスを改善するため、一時ファイルを別のディスクに置くことができます。一時ファイル全体(単一ファイルの場合最大 2GB)を入れるために十分なスペースをディスクに確保しなければ、エラーになります。データベース起動前に、JConfiguration Tool を使って、或いは **dmconfig.ini** を編集してシステム一時ファイルを指定することができます。次の **dmconfig.ini** のキーワードは、システム一時ファイル名とディレクトリを指定します。

DB_TpFil = <ファイル名>[<ファイル名>…] (初期設定: "<データベース名>.TMP")

BLOBフレーム・サイズの指定

BLOB フレームは、BLOB ファイルが使用する最小のストレージ単位です。BLOB ファイルは、LONG VARCHAR や LONG VARBINARY のようなラージ･オブジェクトを保存するために使用します。BLOB フレームのサイズは、データベース作成後に変更することはできません。最小フレーム・サイズは 8KB、最大フレーム・サイズは 256KB です。フレーム・サイズの決定は、ディスク利用とパフォーマンス間のトレードオフです。BLOB 全体を頻繁に回収する場合、BLOB 全体を含むようフレーム・サイズを調節することは、1 ディスクしかアクセスする必要がないので、より良いパフォーマンスに繋がります。但し、実際には様々な種類の BLOB データ・サイズが存在して

います。フレーム・サイズを最大の BLOB にあわせると、最小の BLOB を含むほかのフレームは、未使用のディスクスペースを抱えることになり、スペースの無駄です。替わりに、フレーム・サイズを最小の BLOB にあわせると、大きい BLOB をフェッチする際、複数のフレームにまたがって保存することになり、パフォーマンスが下がります。次の **dmconfig.ini** のキーワードは、BLOB フレーム・サイズを指定します。

DB_BFrSz = <nk>。フレームのパラメータ< nk >はキロバイトです。システム BLOB ファイルのサイズは、(ページサイズ+ (フレーム数 – 1) × nk)。詳細については、7章の「ラージオブジェクト管理」を参照して下さい。

➲ **例**

BLOB フレーム・サイズを 10KB にセットする:

```
DB_BFrSz = 10
```

自動拡張表領域に追加するページ数の設定

データファイルや BLOB ファイルの全ページが一杯になったとき、DBMaster は自動的にファイルのページ数やフレーム数を増やして表領域を拡張します。この設定は、ファイルが一杯になった時に追加するページ数やフレーム数を指定します。データベース管理者は、データベースをすばやく拡張しようとする場合、ファイルを追加する頻度を下げるために、大きい数を選択します。データベース起動前に、JConfiguration Tool を使って、或いは **dmconfig.ini** ファイルを編集して、この数を調節することができます。次の **dmconfig.ini** のキーワードは、自動拡張表領域に追加するページ数/フレーム数を指定します。

DB_ExtNp = <ページ数>、追加するページ数(初期設定: 20 ページ/フレーム)

ユーザー・ファイル・オブジェクトの利用

FILE データ型は、ユーザー・ファイル・オブジェクトやシステム・ファイル・オブジェクトとして保存することができます。ユーザー・ファイル・オブジェクトは、データベースが存在する PC にアクセスする外部ファイルです。言い換えると、ユーザー・ファイル・オブジェクトは、データベース外部の外部ファイルへのリンクに過ぎません。ユーザー・ファイル・オブジェクトを有効にすることは、FILE データ型のカラムをデータベース・

サーバーにアクセスする外部ファイルにリンクさせることです。必要に応じて、使用不可能または可能にすることができます。挿入したユーザー・ファイル・オブジェクトは、設定を OFF にした場合でもアクセスすることができます。データベース起動前に、JConfiguration Tool のストレージのページで、ユーザー・ファイル・オブジェクトを使用可能にすることができます。データベース起動前にキーワード値を変更することができます。キーワード値を 0 にすると、ファイル・オブジェクトを挿入することはできません。キーワード値を 1 にすると、ファイル・オブジェクトを挿入することができます。次の **dmconfig.ini** キーワードは、ファイル・オブジェクトの使用を指定します。

DB_UsrFo = <値> (初期設定: 0 / 使用不可能)

システム・ファイル・オブジェクトのディレクトリの作成

システム・ファイル/オブジェクトは、DBMaster によって生成、削除、管理されます。システム・ファイル/オブジェクトは全て、システム・ファイル・オブジェクトのディレクトリに配置されます。システム・ファイル・オブジェクトのディレクトリを変更すると、これまでに挿入されたシステム・ファイル・オブジェクトが存在するディレクトリは変更しません。データベース起動前に、JConfiguration Tool のストレージのページで、システム・ファイル・オブジェクト名とそのディレクトリを修正します。データベース起動前に、キーワード値を変更することができます。次の **dmconfig.ini** のキーワードは、システム・ファイル・オブジェクトのディレトリを指定します。

DB_FoDir = <パス名> (初期設定: "\<データベース・ディレクトリ>\fo")

ユーザー定義関数DLLファイルのディレクトリの作成

データベース管理者は、ユーザー定義関数(UDF)のダイナミック・リンク・ライブラリ(DLL)のディレクトリを指定することができます。UDF は、ダイナミック・リンク・ライブラリ(Windows オペレーティング・システムでは.DLL、UNIX オペレーティング・システムでは.so)に保存されるコンパイルされた関数です。ユーザー定義関数の DLL ファイルのディレクトリに保存された DLL は、DBMaster にアクセス可能で、SQL 文や ODBC アプリケーションで使用することができます。データベース起動時に UDF をロード

します。次の **dmconfig.ini** のキーワードは、UDF の DLL ファイルのディレクトリを指定します。

DB_LbDir = <ファイル名> (初期設定: 現在の作業ディレクトリ)

ログ･システムの起動

DBMaster は接続情報、ユーザ情報、実行時間、SQL コマンドといった情報を記録するためのログ･システムを提供します。システムは追加のデータベース情報取得に使用することができ、実行時間環境のエラー解決にも非常に有効です。

ログフォーマットはテキスト **CSV** フォーマットです。だから、**.log** ファイルを**.csv** ファイルにリネームする場合ユーザーはエクセルビューアーを使用してチェックできます。以下の表にログファイルの全てのカラムと各のカラムの内容を説明されます。

カラム名	説明
LOG_TIME	ログを書き入れるタイム
BEG_TIME	コマンドの開始時間
STATE	_、O、X、S。"_、 O、 X、S"によって未知、OK、エラー或いは遅くのような四つの状態を判断します。チェックシーケンスは RC を最初にチェックしてから実行時間をチェックします
RETCODE	戻りコード、0 或いはエラーコード
EXE_TIME	実行時間
SV_FUNC	現時点でのサーバー機能を実行する
CONNECT_ID	接続 ID
USERNAME	ユーザーの名前
LOGIN_TIME	ログインタイム
LOGIN_ADDR	ログイン IP アドレス
STMT_ID	句の ID
ERROR_ARG	エラー引数

OTHER_INFO	ほかの LGXXX 設定(ex: LGPLN)を開くと、この情報を LOGNAME.TXT にレコードされます。ユーザーは「INFO_XXX」を.TXT までマークしてチェックします
SQL_CMD	最後に実行された SQL コマンド

ログシステムはデータベース起動前の **dmconfig.ini** キーワード **DB_LGSVR** の設定、もしくは実行時間にストアドプロシージャ SETSYSTEMOPTION() の呼び出しにて有効になります。

ログ情報はレベルで分けられます。どの処理が記録され、いつ取得されるのかが各レベルで決まっています。ログ機能が有効のとき、DBMaster はログ取得オプションに従ってサーバーの処理を記録し、**DB_LGDIR** で指定されたディレクトリにログを保管します。DBMaster は DB 名とログの索引番号からログの名前を決定します。サーバログはログファイル名に当日の日付を入れることができます。ログファイル名は一意で上書きできません。保持したい日数を指定でき、デーモンが期限の切れたログファイルを削除します。**dmconfig.ini** のキーワード **DB_LGDAY** にて設定されます。複数のログファイルが肥大化することになるため、ストレージ容量の節約に古いログファイルの圧縮を行うことがよいとされます。この設定は **DB_LGZIP** に対応するものです。ログ取得開始時にはいくつかのシステム情報は DBNAME.LOG に記録され、ログ情報は DBNAME_currentdate_1.LOG に記録されます。ログファイルが 100MB もしくは DB_LGFSZ で指定したサイズに達した際、ログ情報は引き続いて DBNAME currentdate_2.LOG に記録されます。例:DBNAME_20080706_2.LOG, DBNAME_20080706_3.LOG, …DBNAME_currentdate_n.LOG DB_LGFNO とDB_LADAY で指定されていない場合、n は初期値で 20 です。

DB_LGPLN, DB_LGPAR, DB_LGLCK, DB_LGDAY, DB_LGZIP が起動されている場合や DMERROR.LOG が DBNAME_currentdate_n.TXT に出力される情報を含む場合はその他のログ情報が取得されます。初期ファイルサイズと循環ログの点で、DBNAME_currentdate_n.TXT は DBNAME_currentdate_n.LOG と同様に機能します。追加情報は DBNAME_currentdate_n.LOG の OTHER_INFO フィールド内と DBNAME_currentdate_n.TXT ログファイルに INFO_connection_id_number として記録さ

れます。connection_id_number は追加ログ情報のソーストレースに使用できます。 注意点として DBNAME_currentdate_n.LOG と DBNAME_currentdate_n.TXT のネーミングは並行しており、情報は常に同じ n の数値を持つそれぞれのログファイルに記録されます。これは DBNAME_currentdate_n.LOG と DBNAME_currentdate_n.TXT のサイズの合計によって処理されます。この合計がログファイルの最大サイズに達すると、両方のログが一杯であるとみなされ、引き続いた情報は直ちに DBNAME_currentdate_n+1.LOG と DBNAME_currentdate_n+1.TXT に出力されます。DB_LGSYS の有効時追加のログ情報が取得されます。

以下の **dmconfig.ini** キーワードはログシステムに関する設定です。

DB_LGDIR = <pathame> (初期設定: DBDIR/lgdir)

DB_LGSVR = <value> (初期値= 0/ 無効)

DB_LGERR = <value> (初期値= 3)

DB_LGSTM = <value> (初期値= 5 秒)

DB_LGFSZ = <value> (初期値=100 MB)

DB_LGFNO = <value> (初期値= 20)

DB_LGPLN = <value> (初期値= 0/無効)

DB_LGSYS = <value> (初期値= 0)

DB_LGSQL = <value> (初期値= 0/disabled)

DB_LGPAR = <value> (初期値= 0/disabled)

DB_LGLCK = <value> (初期値= 0/disabled)

DB_LGDAY=<value> (初期値= 30)

DB_LGZIP= value> (初期値= 1/有効)

➲ 例

10 秒以上掛かる遅いクエリで 10000 以上のエラーコードのログを取得、ログファイルを 5 日間保持し、閉じたログファイルを圧縮する場合、データベース起動前に以下のように **dmconfig.ini** を定義します:

```
[DBNAME]
.........
DB_LGSVR=3;
DB_LGERR=2;
DB_LGSTM=10;
DB_LGDAY=5;
DB_LGZIP=1;
```

もしくは SETSYSTEMOPTION を呼び出すことで同様の設定が可能です。

```
dmSQL> call SETSYSTEMOPTION('LGSVR', '3');
dmSQL> call SETSYSTEMOPTION('LGERR', '2');
dmSQL> call SETSYSTEMOPTION('LGSTM', '10');
dmSQL> call SETSYSTEMOPTION('LGDAY', '5');
dmSQL> call SETSYSTEMOPTION('LGZIP', '1');
```

ログシステムはサーバ側で使用されます。クライアント、ネットワークのエラーは記録されません。ログが有効のとき、特にログレベルが高い場合、サーバのパフォーマンスに影響があります。サーバログを保管できるだけの HD 容量があるかの注意も必要です。さもなくば HD フルが発生し、情報を損失することになります。

ローデバイス

DBMaster の物理ストレージのファイルシステムは柔軟性に富んでいます。ユーザーは、UNIX ファイルのみ、ローデバイスのみ、両方のファイルシステムのファイルを使用してデータベースを作成することができます。**dmconfig.ini** のファイル名が**/dev/**で始まるファイルは、ローデバイスとして取り扱われます。

ローデバイスの I/O オペレーションは通常の UNIX ファイルよりも高速なので、DBA は、積極的にローデバイスをデータベースファイルとして使用することを検討すべきです。ローデバイスは、データベースを作成する前に作成しておかなければなりません。ローデバイスの作成手順については、

UNIX のシステムマニュアルを参照してください。

ローデバイスのパーティションを区切らずに、複数のファイルをローデバイスに置くことができます。 ローデバイスに複数のファイルを置くには、次の制約を考慮する必要があります：

- 単一のローデバイスに複数のファイルを置くことができません。
- ローデバイス上で複数のファイルを設定しているとき、初期設定の後でファイルサイズを変更することはできません。
- 単一のローデバイスにおける合計ファイルの総サイズは8TBに制約されています。
- 自動拡張ファイルをローデバイス上に置くと、そのデバイスに他のファイルを置くことはできません。 自動拡張として設定したファイル以外の、**DB_DBFIL**、**DB_BBFIL**、**DB_USRDB**、**DB_USRBB**、**DB_TPFIL**ファイルはすべて自動拡張ファイルです。
- **DB_DBFIL**、**DB_BBFIL**、**DB_USRDB**、**DB_USRBB**、**DB_TPFIL**がローデバイスに設定されている場合ページ数という、1つのパラメータしか使用することはできません。 上記のファイルにオフセットを設定することはできません。このファイルは自動拡張ファイルですので、一つのみのローデバイスを占めて、ほかのファイルとローデバイスを共有できません。だから、オフセットが必要としません。

➲ **例**

これは有効です。 500 ページのファイルが作成されます。

```
DB_DBFIL = /dev/sda 500; 500 ページを持つファイルを作成します
```

これも有効ですが、30 ページのファイルが作成されます。パラメータ 500 は無視されます。

```
DB_BBFIL = /dev/sdb 30 500;
```

Microsoft Windows はローデバイスをサポートしません。

➲ **例 1**

```
[MYDB]
f1 = /dev/sda 0 500
```

```
f2 = /dev/sda 500 200
f3 = /dev/sdb 300
```

上記ローデバイスファイルを含む標準の表領域、**ts_raw** を作成するには:

```
dmSQL>CREATE TABLESPACE ts_raw DATAFILE f1, f2, f3
TYPE=DATA;
```

ローデバイスに 3 つのファイルが作成されます。もしページサイズは 4K に設定すると 最初のファイルは/dev/sda のアドレス 0 で始まる 500× 4 K = 2000k のサイズからなり、2 番目のファイルは/dev/sda のアドレス 500× 4 K = 2000 K で始まる 200× 4 K = 800 K のサイズからなっています。3 番目のファイルは、アドレス 0 で始まる 300× 4 K =1200 K のサイズからなっています。

➲ **例 2**

```
[MYDB2]
DB_JnlSz = 1000
DB_JnFil = J.1 /dev/sda 1000 /dev/sda 2000 J.2jnl
```

もしページサイズは 4 K を選ぶと 2 つの通常のジャーナルファイル J1.jnl、J2.jnl、および 1 つは/dev/sda のアドレス 4000 K で始まり、もう 1 つは/dev/sda のアドレス 8000 K で始まる 2 つのローデバイスジャーナルファイルが作成されます。

クライアント/サーバー・データベースの利用

データベースを、シングルユーザー・データベースとして、或いはマルチユーザー・データベースとして起動することができます。データベース作成前、データベースの初期の機能とどのユーザー・モードが適しているかを決定します。データベースを当初マルチユーザー・データベースとする場合、DBMaster サーバーを運用しているネットワークに合った IP アドレスと DNS 名をデータベースに設定します。同様に、データベース・サーバーが使用する TCP/IP ポート番号を指定します。クライアント側のデータベースは、データベースに接続するための情報を設定します。この設定は、データベース起動前であれば変更することができますが、快適な操作を行うためにデータベース作成前にこれらの設定を行うことを強くお勧めします。クライアントは、不正に環境設定したサーバー・データベースに接続することができません。双方の設定が無効である場合、データベースはシング

ルユーザー・モードで起動します。これらのパラメータは、JConfiguration Tool の接続のページ、又は次の **dmconfig.ini** のキーワードを編集して変更することができます。

IP アドレス/サーバー名: **DB_SvAdr** = **<IP アドレス>** 又は **<ホスト名>** (初期設定: ローカル・ホスト名、又は 127.0.0.1)

ポート番号: **DB_PtNum** = <ポート番号> (初期設定: 2300, 1024~65535)

ユーザー名とパスワードの初期設定値

初期設定のユーザー名とパスワードは、データベースに登録されているものでなければなりません。このキーワードは、データベース接続時にチェックされますが、データベースの起動時には調べられません。

➲ **例**

データベース接続時に使用する初期設定のユーザー名とパスワードを指定する:

```
DB_USRID = <ユーザー名>
DB_PASWD = <*****>
```

言語コード・オーダーを選択する

DBMaster は **DB_WSORT** にて定義する複数のワードソートオーダーを提供しています。例 : **DB_WSORT** はソートオーダーの大小文字区別を定義することができます。初期設定は バイナリのオーダーソートオーダーとなります。

DBMaster は、日本語に JIS、英語用に ASCII、繁体中国語に BIG5、簡体中国語に GB といった様々な文字セット(言語コード)をサポートしています。**dmconfig.ini** ファイルのキーワード **DB_LCode** は、DBMaster が使用する文字セットを指定します。各文字セットに、複数の並べ替えオーダーがあります。

例えば繁体中国語の場合、コード順、画数順、発音順等による並べ替えオーダーがあります。DBMaster の初期設定の並べ替えオーダーは、バイナリ

順です。新規データベースを作成時、キーワード **DB_Order** で指定したユーザー定義のオーダー定義ファイルが、並べ替え順序を変更します。

ソート・オーダーの設定

ソートオーダーは DBMaster がクエリに対してどのようにデータを返すか、DBMaster のクエリに含まれる GROUP BY、ORDER BY、DISTINCT 句のルールを決める設定です。 ソートオーダーは同時にどのように WHERE、DISTINCT 句を含んだクエリを確定させるかの決定もします。

dmconfig.ini のキーワード **DB_WSORT** にてワードソートオーダーを定義します。

DBMaster は大文字と小文字を区別しないソートオーダーに定義されている場合、文字の大小のみが異なっても同じ文字列とみなします。(例: 'John' = 'john)大文字小文字の区別をしないオーダーでもクエリの結果ではしばしば大文字小文字を区別を必要とすることがあります。以下の **dmconfig.ini** のパラメータはワードのソートオーダーを大文字小文字を区別するように定義しています。

DB_WSORT = <values>(初期値:0/バイナリソートオーダー)

1 は大文字小文字を区別しないソートオーダー

2 は大文字小文字区別のソートオーダー

下記は、新規データベースを作成する前に言語とソートオーダー・ファイルを設定する方法を表しています。

➲ **例**

言語タイプを繁体中国語に設定し、BIG5 を使う:

```
[MY_DB]
……
DB_LCODE =1                ; 繁体中国語の BIG5
DB_ORDER = big5_stroke.ord ; オーダー定義ファイル
……
```

キーワード DB_ORDER は、DBMaster インストール・ディレクトリの *shared/codeorder* サブディレクトリに配置されている、*big5_stroke.ord* と名づけられたユーザー定義のオーダー定義ファイルを表しています。オーダー定義ファイルは、DBMaster のソート結果に影響を与えるテキスト・ファイルです。このキーワードは、データベースの作成時に使用され、データベースに記録されて使用されることはありません。このキーワードが無いと、データベース作成時、ソート順は、バイナリ順になります。定義ファイルを指定すると、常にファイルが存在しなければデータベース起動時にエラーになります。

ユーザー・オーダー定義ファイル

オーダー定義は、ユーザー定義のテキスト・ファイルです。オーダー定義ファイルは、有効な文字の並びを決定します。名前の例は、*codename_ordertype.ord* のようになります。例えば *big5_stroke.ord* のように、Codename には言語名、ordertype には並び方を用います。

➲ 例

オーダー定義ファイル:

```
Comment: You can write information here.

[BEGIN]       // begin to arrange the character sequence

c             // ASCII 0x63
0x62          // Character 'b'
a             // ASCII 0x61

[SINGLE]      // Single-Byte Character Default Order

[DOUBLE]      // Double-Byte Character Default Order

0xA440        // one of Chinese characters
0xA441        // one of Chinese characters
0xA442        // one of Chinese characters
```

[BEGIN] キーワードの前の全行は、コメントとして利用されます。**//**や**/***の後の全文字もコメントです。[BEGIN]の後ろの各行は、一つの文字を表しま

す。定義する文字は、行の最初に置き、最低 1 つのスペースか改行マーク(¶)が後に続きます。オーダー定義ファイルの文字は、少ないものから多い順でリストされます。上記の例では、文字 **c** は **b** より少なく、**b** は **a** より少なくなります。

テキスト編集ソフトで編集できない文字は、16 進数でそれらを表すことができます。例えば、**a** という文字は、**a** と表すこともできますし、コード値 **0x61** を使用することもできます。見えない文字にも役に立ちます。

並べ替えオーダーの作成者は、いくつかの文字のみを指定して、その他は初期設定つまりバイナリで並べ替えるようにすることもできます。キーワード[SINGLE]と[DOUBLE]は、定義ファイルで指定されない、シングル文字セットとダブル文字セットを指すのに使用されます。キーワード[SINGLE]がオーダー定義ファイルに追加されない場合、定義されないシングル-バイト文字は、定義ファイルにあるその他の全ての前になります。キーワード[DOUBLE]がオーダー定義ファイルに追加されない場合、定義されないダブル-バイト文字は、定義ファイルの全文字の後になります。

DBMaster は、定義ファイルにエラーが見つかった場合、初期設定を使用します。例えば、[BEGIN]が無くなった場合、常に全文字に対し初期設定の並べ替えを適用します。同じ文字が 2 度以上現れた場合、最初のインスタンスが処理され、その他は無視されます。新規データベースを作成した後、作成者は並べ替え順序が正しいかどうかを注意深くチェックする必要があります。

分散型データベース環境では、全データベースは同じ並べ替えオーダー定義ファイルを使用する必要があります。他のマシンに全データベースをコピー又は移動する場合、必ず並べ替えオーダー定義ファイルもコピーして下さい。

データ通信制御域

*データ通信*制御域（*DCCA*）は、ほとんど全ての情報とデータが置かれるメモリブロックです。マルチユーザー・データベースの場合、DCCA は共有メモリに割り当てられ、プロセス間通信に使用されます。データベースを起動すると、データベースの全情報を保持するための DCCA が割り当てら

れます。DCCA は三つの部分—ページバッファ、ジャーナルバッファ、システム制御域に分けられます。

dmconfig.ini には、DCCA に関連するキーワードが幾つかあります。

- **DB_NBufs=<NP>**—ページバッファのページ数を指定します。初期値は 250です。
- **DB_NJnlB=<NP>**—ジャーナルバッファのページ数を指定します。初期値は 64 です。
- **DB_ScaSz=<NP>**—システム制御域のページ数を指定します。初期値は 100 です。
- **DB_MaxCo**=<number>—データベースが扱うことができる同時実行のトランザクションの最大数を指定します。また、データベース作成時や新規ジャーナル・モードで起動した時に、ジャーナル・ファイルをフォーマットするために使用されます。

DCCA の大きさは、ページバッファ、ジャーナルバッファ、システム制御域のサイズを加えることで概略推定することができます。ただし、システム制御域のサイズが小さすぎると、実際に必要なサイズよりも小さい推定になるかも知れません。指定した DCCA のサイズが十分に大きくない場合、DBMaster は、上記の初期値の代わりに、必要な情報を保持するための最小スペースを自動的に DCCA に割り当てます。

UNIX のマルチユーザー環境では、DCCA は共有メモリに割り当てられるので、DCCA のサイズがシステムの共有メモリサイズを超えることはできません。共有メモリを増大する方法については、UNIX マニュアルを参照してください。共有メモリを大きくするには、一般にカーネルの再構築が必要になります。DBMaster は、バッファとシステム制御域が大きければ大きい程スムースに走行します。

DCCA、ページバッファ、ジャーナルバッファ、システム制御域間の関係は、18 章の*「パフォーマンス・チューニング」*でより詳細に説明します。

JConfiguration Tool のキャッシュと制御のページでも、DCCA パラメータを設定することができます。詳細については、*「JConfiguration Tool 参照編」*をご覧下さい。

4.3 データベースを起動する

データベースを起動すると、オペレーティングシステムからリソースを獲得し、初期化し、利用者のデータベース接続を待ちます。パラメータによっては、データベース起動前に考慮する必要があります。そのパラメータは以下のとおりです。

- データベースの起動モード
- クライアント/サーバー・データベースを使用可能にする
- データベースのIPアドレスとポート番号(クライアント/サーバー・データベース用)
- 初期設定ユーザーIDとパスワード
- メモリ割り当て

dmSQL や JServer Manager を使って、データベースを起動することができます。dmSQL を使ったデータベース起動についての詳細は、以下の節をご覧下さい。JServer Manager を使ったデータベース起動の方法については、*「JServer Manager ユーザーガイド」*を参照して下さい。

シングルユーザー・データベースを起動する

シングルユーザー・データベースは、データベース接続の度に起動し、切断と共に終了します。

➲ **例**

dmSQL を使って、シングルユーザー・データベースを起動する：

```
dmSQL> START DB <データベース名> <ユーザー名> <パスワード> ;
.
< DML の実行 >
.
dmSQL> TERMINATE DB;
```

注 *DBA権限をもつユーザーのみデータベースを起動することができます。DBA権限については、8章の「セキュリティ管理」を参照してください。シングルユーザー・データベースは、1ユーザーのみデータベースにアクセスします。*

クライアント/サーバー・データベースを起動する

クライアント/サーバー・データベースは、DBA がデータベース・サーバーを起動し、別の（または同じ）マシンの全てのクライアントがネットワークを経由してデータベースに接続できるようにします。

クライアント/サーバー・データベースを起動するには、2 つのシステム情報が **dmconfig.ini** に必要になります。第一はサーバー機の IP アドレスです。全てのクライアントがサーバーと同じマシンにいるわけではなく、IP アドレスがネットワーク上の各マシンを識別する唯一の ID になります。サーバーの IP アドレスは **DB_SvAdr** で指定します。

第 2 はポート番号です。サーバー・プログラムは、**DB_PtNum** で指定されたポート番号と結合して接続を待ちます。データベース・サーバーと通信するすべてのクライアント・プログラムは、そのポート番号に接続しなければなりません。

➲ 例 1

サーバーIP アドレスとサーバー/クライアントポート番号を指定する：

```
DB_SVADR = <サーバの IP アドレス> （クライアント側）
DB_PTNUM = <ポート番号> （サーバー側とクライアント側）
```

➲ 例 2

dmServer を使って、サーバー機でクライアント/サーバー・データベースを起動させる:

```
UNIX> dmserver <データベース名>
```

➲ 例 3

ユーザー名とパスワードを入力して下さい。dmServer が起動して、クライアントが接続するのを待機する：

```
UNIX> dmserver [-f] [-t ポート番号] [-u ユーザー名 [-p パスワード]]
データベース名
```

Unix スイッチの説明：

- **-f:** サーバー・プログラムはフォアグラウンド・モードで走行します。（dmserverは通常バックグラウンド・モードで走行します。）
- **-t:** 使用するポート番号を指定します。dmconfig.iniに定義されるポート番号の代わりに、このポート番号を使用します。
- **-u:** ユーザー名を指定します。
- **-p:** パスワードを指定します。

コマンドラインでユーザー名とパスワードを指定しない場合は、dmconfig.iniの DB_UserId と DB_PasWd が参照されます。DB_UserId と DB_PasWd が設定されていない場合、ユーザー名とパスワードの画面入力を促します。

起動モード

dmconfig.ini の **DB_SMode** キーワードでデータベースの起動モードを指定します。**DB_SMode** キーワードには、起動モードに対応する次の 6 つの値があります。

- **1** — システムを普通に起動するノーマル起動。データベースが前回のセッションの際にクラッシュした場合、データベースを整合した安定した状態に戻すために、DBMasterは自動的にクラッシュ・リカバリを実行します。
- **2** — 新規ジャーナル。新しいジャーナル・ファイル名やディレクトリが **dmconfig.ini**ファイルにセットされた場合、データベースを新規ジャーナル・モードで起動させる必要があります。新規ジャーナル・ファイルやディレクトリは、JConfiguration Toolのストレージのページでも指定することができます。以前のジャーナル・ファイル名をそのまま使う場合、古いレコードは全て上書きされます。ユーザーがジャーナル・ファイルのサイズを変更、ジャーナル・ファイルを追加、ジャーナル・ファイル名を変更しようとする時に、この設定を選択しなければなりません。このオプションを選択する前に、増分または完全バックアップをすることが理想的です。

- 3 — バックアップしたデータベースのリストアは、データベースを起動するためにバックアップしたデータベース・ファイル（ジャーナル・ファイルを含む）を使います。DB_RTimeで指定した時点まで、操作をロールオーバーするために増分バックアップ・ファイルが使用されます。このキーワードが指定されていない場合、または最後の増分バックアップ以降の時点に指定されている場合、DB_RTimeの初期値が使用されます。ロールオーバーについての詳細は、「リカバリ、バックアップ、リストア」の章を参照して下さい。
- 4 — データベース・レプリケーションのソース・データベース。このモードでシステムを起動すると、ソース・データベースになります。データベース・レプリケーションについての詳細は、*「データ・レプリケーション」*の章を参照して下さい。
- 5 — データベース・レプリケーションのターゲット・データベース。このモードでシステムを起動すると、ターゲット・データベースになります。データベース・レプリケーションについての詳細は、*「データ・レプリケーション」*の章を参照して下さい。
- 6 — データベースは、読み込み専用で普通に起動します。但し、データベースは読み込み専用で、ユーザーには読み込み権限しか与えられません。読み込み専用モードでソース・データベースを起動すると、ユーザーはそれを修正することができません。

起動モードは、JConfiguration Tool のデータベース起動のページや、JServer Manager のデータベース起動の高度な設定ウィンドウでも指定することができます。

強制起動

何らかの事情で損傷したデータベースを起動すると、常にエラーメッセージが戻ります。この問題の解決策として、強制的にデータベースを起動させる「強制起動」モードがあります。環境変数 DB_ForcS を 1 に設定すると、データベースは強制的に起動されます。より詳細については、15 章の *「リカバリ、バックアップ、リストア」*をご覧下さい。

E-MAILエラーレポート・システム

一般的にエラーメッセージは、全て dmerror.log ファイルに保存されます。データベース管理者が同時に dmerror.log ファイルを確認しない限り、データベースのエラーによっては気付かないまま報告されているかもしれません。DBMaster には、e-mail エラーレポート・システムがあります。これは、システムによってデータベース管理者にエラーを通知させます。

エラーレポート・システムは、2 つの環境設定ファイルのキーワードで有効にすることができます。このキーワードは、JConfiguration Tool、或いはデータベース起動の際に JServer Manager で指定することができます。e-mail レポート・システムの振る舞いを決定するキーワードは、**DB_ERMRv** と **DB_ERMSv** です。DB_ERMRv は、e-mail エラーレポートの参加者を指定し、**DB_ERMSv** は、e-mail が経由する SMTP サーバーのアドレスを設定します。JConfiguration Tool、又は JServer Manager を使ったエラーレポート・システムの設定方法についての詳細は、*「JConfiguration Tool 参照編」*と*「JServer Manger ユーザーガイド」*を参照して下さい。

4.4 データベースの接続

この節は、起動したクライアント/サーバー・データベースとの接続方法を説明します。データベースの DML オペレーションを実行する前に、まずデータベースに接続します。切断後もクライアント/サーバー・データベースはまだアクティブです。データベースを終了するまで、更に接続することができます。

クライアント/サーバー接続用のパラメータには、ポート番号、サーバー・アドレス、接続タイムアウト時間、ロック・タイムアウト時間などがあります。**dmconfig.ini** ファイルのキーワード値を変更、または JConfiguration Tool を使って接続パラメータを変更することができます。

シングルユーザー・データベースの場合は、1 ユーザーしか接続できないので、データベースを使用する際にそれを起動するだけで、接続する必要はありません。詳細については、*「データベースを起動する」*の節を参照して下さい。

クライアント/サーバー・データベース

DB_SvAdr と **DB_PtNum** キーワードを **dmconfig.ini** ファイルで設定する必要があります。**DB_UsrId** と DB_PasWd キーワードが **dmconfig.ini** に定義されている場合は、CONNECT 構文の<***ユーザー名***>と<***パスワード***>を省略することができます。

➲ **例**

dmSQL でクライアント/サーバー・データベースに接続/切断する:

```
dmSQL> CONNECT TO <データベース名> <ユーザー名> <パスワード>;
.
< DML の実行 >
.
dmSQL> DISCONNECT;
dmSQL> QUIT;
```

接続タイムアウト

サーバー機の電源が切れている、或いはサーバーの IP アドレスを間違えている場合、クライアント/サーバーのデータベースに接続することができません。この場合、クライアントは、接続が確立するまで長時間待つかもしれません。**DB_CTimO** パラメータで接続タイムアウト時間を秒単位で明示的に設定することができます。このキーワードの初期値は 5 秒です。

ロックのタイムアウト

データベース接続をするときに、**dmconfig.ini** にロックのタイムアウトのキーワード **DB_LTimO** を秒単位で定義して、ロックできないときの待ち時間を指定することができます。

例えば、**DB_LTimO**=10 とした場合、ユーザーが 10 秒以上ロックを待つと、「ロックタイムアウト」のエラーが返ります。**DB_LTimO**=0 は、ロック待ちをしないことを意味します。**DB_LTimO**=-1 は、ロックが解除されるまで待ち続けることを意味します。ユーザーは、自分自身の **DB_LTimO** 値を指定することができます。

データの圧縮

ネットワークトラフィックの最も主な原因はデータベースのデータコンテンツです。データ圧縮はネットワークで転送データを減らし、パフォーマンスを向上させます。

データベース接続前にキーワード **DB_NETZC** の設定でネットワーク圧縮機能を有効にします。この機能が有効のときサーバから送信される際に圧縮されクライアントがデータを受け取ったときに展開されます。

➲ **例**

ネットワーク圧縮を起動するには、データーベースを接続する前に以下のように **dmconfig.ini** をセットします:

```
[DBNAME]
DB_NETZC = 1;
```

4.5 データベースを終了する

すべての作業が完了する場合、データベースを終了します。データベースを終了すると、DBMaster は DCCA のようなリソースを全て解放してオペレーティング・システムに戻します。このとき、アクティブ・トランザクションがデータベース・エンジンに残っていれば、それを中止します。

ただし、データベース・エンジンに接続が残っていても、接続プロセスを切断せずにデータベースを終了します。この場合、データベース管理者は手動でプロセスを停止しなければなりません。停止しないと、次にデータベースを起動するときに、「ファイルをロックできません。トランザクションロールバック」のエラーメッセージが出ます。

データベース管理者(DBA ユーザー)は、データベース終了前に全てのユーザーがログオフしていることを確認します。DBA はまずデータベースに接続してから、終了のための SQL 文を実行して、データベースを終了します。DBA のみデータベース終了の権限をもっています。

➲ 例

dmSQL でシングル又はクライアント/サーバー・データベースを終了する:

```
dmSQL> TERMINATE DB;
```

5 ストレージアーキテクチャ

この章は、DBMaster の論理レベルと物理レベルのストレージアーキテクチャについて解説します。

論理レベルは、利用者に理解しやすい方法でデータベースのデータ編成を提示します。物理レベルは、表領域内の情報に対応するオペレーティングシステムの物理ファイルによって構成されます。情報は DBMaster によって管理され、利用者には隠されています。

表領域とファイルを使用して、どのようにデータベースのストレージを管理するかについても説明します。

5.1 アーキテクチャ

DBMaster データベースは、表 *領域* と言われる一つ以上の論理的な区画で構成されます。表領域は、DBMaster の主要な論理ストレージ構造であり、論理的には、1 つ以上の表および索引があります（図5-1 を参照）。物理的には、表領域は 1 つ以上のオペレーティング・システム・ファイルで構成されます（図5-2 を参照）。

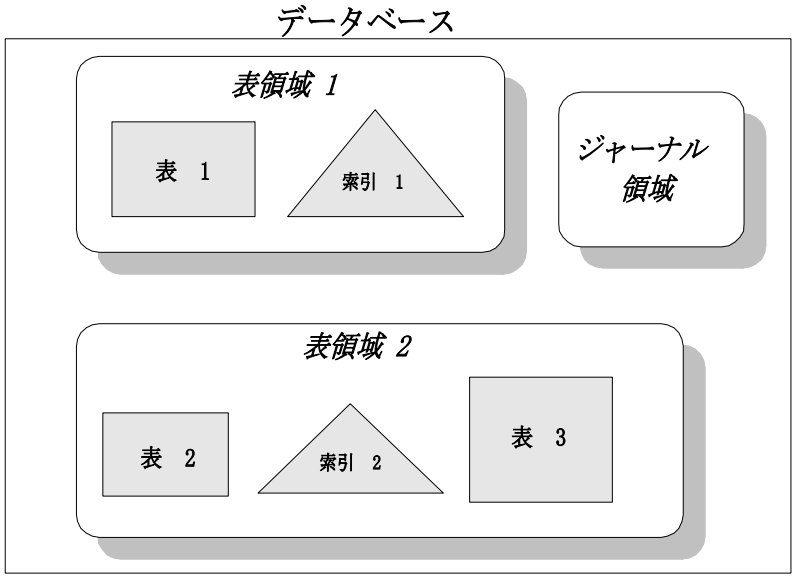


図 5-1：DBMaster データベースの論理ストレージ構成

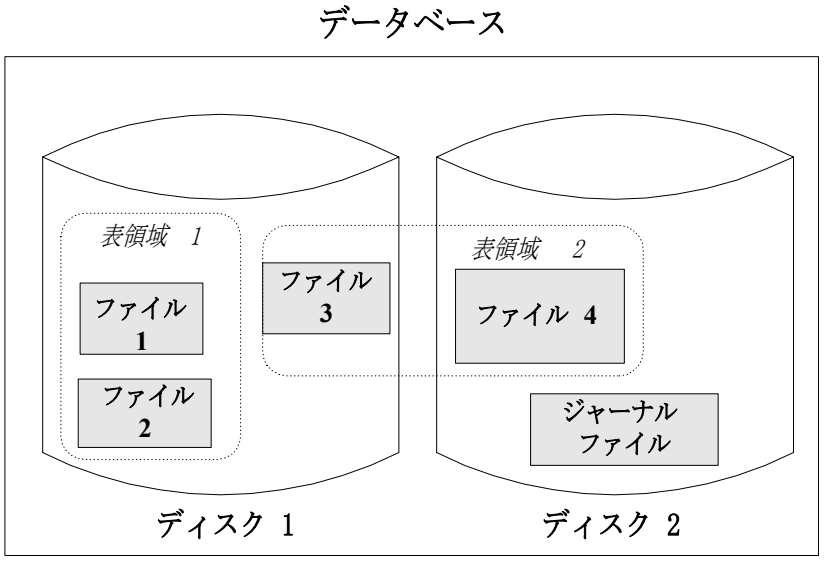


図 5-2：DBMaster データベースの物理ストレージ構成

5.2 ファイルの種類

DBMaster で使用するオペレーティング・システムのファイルには 10 種類あります。データベースの様々な用途を保存します。システム・データファイルとシステム BLOB ファイル、ユーザー・データファイルとユーザーBLOB ファイル、システム・ジャーナルファイル、システム一時ファイル、ユーザー定義ファイル、DBMaster ログファイル、バックアップファイル、表レプリケーション・ファイルです。システム・データファイルとシステム BLOB ファイル、ユーザー・データファイルとユーザーBLOB ファイルは、データベースの保存アーキテクチャと表領域に関するものです。ジャーナルファイルは、データベースで実行されるトランザクションの記録を保存するために重要な役割を果たし、データベースのバックアップとリカバリにも極めて重要なものです。

DBMaster は、データベースの性能を上げるために、２種類のファイル—データファイルとバイナリラージオブジェクト（BLOB）ファイルに分けてデータを格納します。BLOB データは、画像、音声、大量テキストのように 1 ページに収めきれない大きいデータオブジェクトで構成されます。データ行や索引キーはデータファイルに格納し、BLOB データは BLOB ファイルに格納します。DBMaster は、2 種類のファイルを別々の方法で管理して高いパフォーマンスを実現します。

ユーザー・データファイル

データファイルがページで構成されるのに対して、BLOB ファイルはフレームで構成されます。データファイルと BLOB ファイルの両方とも、最大サイズは 8TB です。しかし、フレームとページは、次の点で大きく異なります：

- データーベースを作成する時ページサイズを**dmconfig.ini**キーワード **DB_PGSIZ**にて4KB,8 KB, 16KB ,32KBに指定できます。
- ページは複数のレコードを含むことができますが、フレームは１件のみのBLOBデータを含みます。

ページは、データファイルが使用するストレージの最小単位です。表と索引は、同じページフォーマットでデータを格納します。データページは、4つのセクションに分けられます。

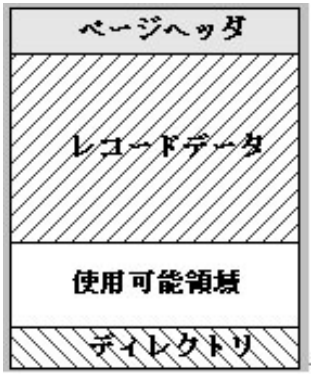


図 5-3：データページのフォーマット

ページヘッダは、DBMaster システム全般のページ情報を含みます。レコードデータ領域は、実際の表や索引のデータを含みます。表と索引は、行およびカラムで表示されます。レコードディレクトリは、ページ内のレコードに関する情報をもちます。使用可能領域はページ内の未使用領域です。

ユーザーBLOBファイル

BLOB フレームは、BLOB ファイルが使用するストレージの最小単位です。データベースを作成する前に、**dmconfig.ini** に BLOB フレームのサイズを指定することができます。フレームサイズは、8KB～256KB です。但し、Windows 3.1 環境では 8KB に固定されます。BLOB フレームは、3 つのセクション、フレームヘッダ、BLOB データ、使用可能領域に分けられます。BLOB ファイルの詳細情報については、7章の *「ラージオブジェクト管理」* を参照してください。

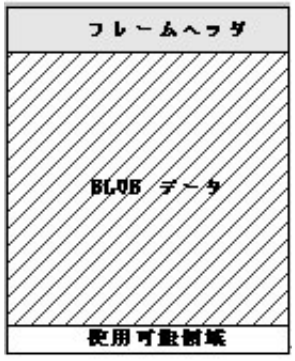


図 5-4 : フレームのフォーマット

フレームヘッダは、ページヘッダと同様、DBMaster システム全般のフレーム情報を含みます。BLOB データ領域は、BLOB データを 1 件のみ含みます。しかし、フレームサイズよりも大きい BLOB データは、複数のフレームに跨ります。使用可能領域はフレーム内の未使用領域です。

ジャーナルファイル

DBMaster のジャーナルは、複数の物理ジャーナルファイルから構成されます。全てのジャーナルファイルは同じサイズであり、DB_PGSIZ で決定します。アクティブ・トランザクションのデータベース変更アクションは、全てジャーナルに記録されます。ジャーナルは、論理的なジャーナルレコードで構成されます。複数レコードがブロックに収められたり、レコードが複数ブロックに跨ったりします。アクティブ・トランザクションのジャーナルレコードは、再利用できません。ジャーナルファイル全体は、ジャーナルレコードのリングを形成します。

DBMaster は、使用中のジャーナルファイルが一杯になると、自動的に次のジャーナルファイルに切り替えます。アクティブ・トランザクションが全ジャーナルファイルを一杯にすると、使用できるジャーナルブロックがなくなり、トランザクションはアボートされます。これを *ジャーナルフル* 言います。

ジャーナル・ファイルには、ジャーナルブロックの他に、ジャーナルステータスブロックがあります。ジャーナルステータスブロックは、データベ

ースのリカバリあるいはリストアに使用されます。リカバリとリストアは後の節で説明します。

DBMaster は、ジャーナルブロックバッファをメモリにもち、ジャーナルファイルのアクセスを高速にします。修正データをディスクに書き出す前に、*書き出し先行ログ*(WAL)プロトコルを使用してジャーナルレコードをディスクに書き出します。ジャーナルバッファは、バッファが一杯になるか、トランザクションがコミットされると、WAL プロトコルを用いてジャーナルファイルに掃き出されます。

DMCONFIG.INIのジャーナル・パラメータ

次のジャーナルパラメタを指定してデータベース性能を上げることができます。

- **DB_JnFil**—ジャーナルファイル名を指定します。8つまでのジャーナル・ファイル名を指定することができます。ジャーナルファイル名は、カンマまたは空白で区切ります。

➲ **例**

このデータベースには、パフォーマンスを上げるために異なるドライブに指定した、7 つのジャーナル・ファイルがあります。

```
DB_JNFIL=myDb.jn1, myDb.jn2, myDb.jn3, /disk1/usr/myDb.jn4,
         myDb.jn5, /disk2/usr/myDb.jn6, myDb.jn7
```

- **DB_JnlSz**—ジャーナル・ファイルのサイズをページ数で指定します(1ジャーナル・ページサイズはDB_PGSIZ で決定します)。ジャーナルの合計サイズは、次のようになります。

（ジャーナル・ファイル数x ジャーナル・ファイルのサイズ）KB

データベースを作成するときには、ジャーナルサイズを合理的に決定します。全てのジャーナルファイルが一杯になると、ジャーナルフルのためトランザクションがアボートされます。このため、ジャーナルサイズが小さいと、長いトランザクションはアボートされるかもしれません。トランザクションがデータベースを長い間操作する場合は、ジャーナルファイルのサイズを大きくするか、ジャーナルファイル数を増やすべきです。

- **DB_NJnlB**—ジャーナルバッファのサイズをページ数で指定します(1ジャーナル・ページサイズは**DB_PGSIZ** で決定します)。

ジャーナルファイルのサイズを変更する

データベース起動時にジャーナルフルのメッセージに頻繁に出会う場合、ジャーナル・ファイルを拡大するとデータベースのパフォーマンス改善になります。DBMaster 3.0 では、ジャーナル・ファイルのサイズを変更した後、指定した時点までデータベースをリストアすることができませんでした。しかし、バージョン 3.0 以降ではこれが可能になりました。ディスク障害からデータベースを保護するために、ジャーナル・ファイルのサイズを変更した後すぐに完全バックアップを実行します。

➲ **ジャーナル・ファイルのサイズを変更する：**

1. 必要なジャーナル・ファイルのサイズ数を決定するために、最大トランザクションを扱うために必要なディスクスペースを見積もります。
2. データベースを終了します。
3. dmconfig.iniのDB_JnFil、DB_JnlSzの2つのパラメータを再指定する。

   注 *これらの設定は、JServer Managerのデータベース起動ウィザードのストレージのページの高度な設定で変更することができます。*

4. 起動モードを新規ジャーナル・モード（**dmconfig.ini: DB_SMode=2**）にセットします。

   注 *これらの設定は、JServer Managerのデータベース起動ウィザードの データベース起動のページの高度な設定で変更することができます。*

5. データベースを再起動します。
6. 起動モードをノーマル・モード（**dmconfig.ini: DB_SMode=1**）にセットします。

   注 *この設定は、JConfiguration Toolのデータベース起動のページで変更することができます。*

7. データベースがBACKUP-DATA、又はBACKUP-DATA-AND-BLOBモードの場合、オンライン完全バックアップを実行します。

表領域

DBMaster のデータベースは、表*領域*と呼ばれる論理的な領域に区分けされます。表領域は、データベースを管理可能な領域に区分する論理的なストレージ領域です。各表領域には、最低 1 つのオペレーティング・システムのファイルがあります。表領域とファイルを使い始める前に、以下の用語を理解して下さい。

表領域の種類

表領域には、サイズが固定のものと自動的に拡張するものがあります。固定サイズの表領域を*標準表領域*と言い、自動的に拡張する表領域を*自動拡張表領域*と言います。更に、*システム表領域*と呼ばれる特殊な表領域があります。

システム表領域

DBMaster データベースには、システム表領域(SYSTABLESPACE)と、初期設定表領域(DEFTABLESPACE)と呼ばれる少なくとも 2 つの表領域があります。データベースの作成時に、*システムカタログ*表を記録するシステム表領域が生成されます。システムカタログ表は、データベース全体の情報を格納します。

初期設定表領域

ユーザーが特別に表領域を割り当てない場合、初期設定表領域にはユーザー表が格納されます。但し、別の表領域を作成して、ユーザー表を入れるほうがより柔軟で効率的です。

一時表領域

一時表領域（TMPTABLESPACE）は外部の一時表(ETT)をストアするため使用されます。一時表領域は自動拡張表領域です。これは二種類ファイルがあります：データファイルと BLOB ファイルです。データファイルのロジック名は DB_TMPDB で、物理名は DB_TMPDIR/DBNAME.TDB です；BLOB ファイルのロジック名は DB_TMPBB で、物理名は DB_TMPDIR/DBNAME.TBB です。

ユーザーは "create temporary table"または" select into"句を呼び出す場合、ETTsを生成して"TMPTABLESPACE"まで保存されます。ユーザーは"TMPTABLESPACE"（システムはディフォルトに ETT を TMPTABLESPACE に保存する）に一時表を作成できますが、全部の標準表ではありません。ユーザーは "ALTER TABLESPACE TMPTABLESPACE SET AUTOEXTEND OFF/ON;"と"ALTER DATAFILE DB_TMPDB/ DB_TMPBB ADD n PAGES;" をしますが、TMPTABLESPACE にファイルを追加できなくて、TMPTABLESPACE からファイルを削除もできません。データベースが作成する場合、TMPTABLESPACE が作成されます。データベースが起動すると、TMPTABLESPACE は初期値を再設置します。

- TMPTABLESPACEにのみ一時表を作成することができます。
- TMPTABLESPACE.にETTに標準表を作成することができません。
- TMPTABLESPACEにファイルを追加できなくて、TMPTABLESPACEからファイルを削除もできません。
- TMPTABLESPACEを削除することができません。

標準表領域

標準表領域は固定サイズで、最低 1 つのファイルから成ります。標準表領域のファイルが小さすぎて全データを入れることができない場合、手動で拡大することができます。標準表領域は、最大 32767 個のファイルを入れることができます。表領域の全ファイルの合計ページ数は 8TB 以下です。

自動拡張表領域

自動拡張表領域は、必要に応じて自動的に拡張する表領域です。自動拡張表領域のファイルは自動的に拡大します。追加した順と逆に表領域は拡張します。つまり、データスペースで拡張する必要がある場合、最後に追加されたデータファイルから拡張されることを意味します。

表領域を拡張させないようにする場合、自動拡張表領域を標準表領域に変更することができます。その逆も同様です。つまり、標準表領域を使い果たした時に自動拡張表領域に変更することができます。替わりに、新しいファイルを追加、或いは既存のファイルのサイズを大きくすることができ

ます。ローデバイス・ファイルは、標準表領域としてのみ使用します。自動拡張表領域としては使用できません。

データベース作成時、システム表領域と呼ばれる自動拡張表領域が生成されます。他の表領域を作成すると、その初期設定の標準表領域になります。システム表領域を無制限に大きくしたくない場合、標準表領域に変更することができます。

dmconfig.ini で各データファイルのページ数を指定します。データファイルのページ数は、ファイルが自動拡張表領域に属する場合はファイルの初期サイズになり、標準表領域に属する場合は実際のファイルサイズになります。

5.3 表領域とファイルの管理

データベースの表領域とファイルを管理するために、様々な要素を考慮する必要があります。例えば、新規データベースの作成時にデータベースのサイズと種類を決定する、追加表領域を作成する、自動拡張表領域を標準表領域に変更する、表領域にデータファイルを追加する、表領域にあるファイルのサイズを変更する、不要になったファイルおよび表領域を捨てる等です。表はほかの表領域に変更できます。

JDBA Tool や dmSQL 文と **dmconfig.ini** ファイルの編集を組み合わせて、表領域を管理することができます。JDBA Tool は、全表領域の管理ルーチンのための直感型のユーザー・インターフェースです。JDBA Tool を使った表領域の管理方法については、*「JDBA Tool ユーザーガイド」*を参照して下さい。

DBMaster データベースには、システム表領域と呼ばれる少なくとも一つの表領域があります。データベースの作成時に、5 つのファイルが生成されます。システム・データファイル、ユーザー・データファイル、システム BLOB ファイル、ユーザーBLOB ファイル、ジャーナル・ファイルです。システム・データファイルとシステム BLOB ファイルとジャーナル・ファイルは、システム表領域に配置されます。これらのファイルは、データベース全体のシステムカタログ表を記録するために使用されます。ユーザー・データファイルとユーザーBLOB ファイルは、初期設定ユーザー表領域に配置されます。

表領域を追加しない限り、ユーザー表は初期設定ユーザー表領域に配置されます。別途表領域を作成し、ユーザー表を保存する方が、柔軟で効率的です。

システムファイルとシステム表領域を初期設定する

新規データベースを作成すると、システム表領域と 3 つのシステムファイル（システム・データファイル、システム BLOB ファイル、ジャーナルファイル）が生成されます。これらのファイルは、データベース・スキーマとトランザクションの記録を保管するために使用されます。システム・データ、BLOB、ジャーナルの各ファイルには、データベース名に.SDB、.SBB、.JNL という拡張子が付けられます。ファイルのサイズを指定しない場合、各々200 × DB_PGSIZ KB、20KB、4000KB の初期値サイズになります。システム・ファイルに別の名前を使用する場合は、**dmconfig.ini** ファイルか JConfiguration Tool のストレージのページで指定します。

➲ **例**

dmconfig.ini ファイルにシステム・ファイル名を指定する:

```
[MY_DB]                          ;データベース名
DB_DBDIR = \disk1\usr            ;データベース・ディレクトリ
DB_DBFIL = datafile.sdb          ;システム・データファイル
DB_BBFIL = blobfile.sbb          ;システム・BLOB ファイル
DB_JNFIL = jrnlfile.jnl   ;ジャーナル・ファイル
```

CREATE DB 文を実行すると、これらの **dmconfig.ini** ファイルの値を用いて前に述べた 3 つのシステムファイルが生成されますが、初期値のファイル名の代わりに指定したファイル名が使用されます。この場合は、システム・データファイル名は **datafile.sbd**、システム BLOB ファイル名は、**blobfile.sbb**、ジャーナル・ファイル名は **jnlfile.jnl** になります。

システム表領域は自動拡張表領域なので、システム表領域のサイズは初期サイズであり、領域の上限ではありません。システム表領域のディスク容量を制限したい場合は、ALTER TABLESPACE 文を使用してシステム表領域を標準表領域に変更することができます。

標準のシステム表領域の全容量を使い切ってしまったときは、標準表領域にファイルを追加するか、またはファイルにページを追加して容量を拡大するか、表領域の種類を自動拡張に変更します。

ユーザーファイルとユーザー表領域を初期設定する

新規データベースを作成すると、初期設定ユーザー表領域と 2 つのファイル(ユーザー・データファイルとユーザーBLOB ファイル)が生成されます。これらのファイルは、ユーザー・データを保存するために使用されます。ユーザー・データと BLOB の各ファイルに、データベース名と拡張子.DB と.BB からなるファイル名で付けられます。サイズを定義しない場合、各々初期設定のサイズ 200 × DB_PGSIZ KB と 20KB で作成されます。初期設定ユーザー・ファイルに別の名前を使用する場合、**dmconfig.ini** ファイルか JConfiguration Tool のストレージのページで指定します。

➲ **例**

dmconfig.ini ファイルの初期設定ユーザー・ファイルの名前を指定する:

```
[MY_DB]                                    ;データベース名
DB_USRDB = /disk1/usr/f1.db  200           ;ユーザー・データファイル
DB_USRBB = /disk1/usr/f1.bb  20            ;ユーザー・BLOB ファイル
```

CREATE DB コマンドを実行すると、初期設定名の替わりに、**dmconfig.ini** ファイルの値を使用した 2 つのファイルが生成されます。この場合、ユーザー・データファイル名は **f1.db**、そのサイズは 200 ページになります。ユーザーBLOB ファイル名は **f1.bb**、そのサイズは 20 フレームです。

初期設定表領域は自動拡張の表領域ですので、その初期サイズには限界値はありません。

表領域を作成する

データファイルや BLOB ファイルを入れるために追加の表領域を作成することができます。dmSQL や JDBA Tool を使って、表領域を作成することができます。dmSQL を使った表領域作成についての詳細は、*「SQL 文と関数参照編」*を、JDBA Tool を使う場合は、*「JDBA Tool ユーザーガイド」*をご覧下さい。

表領域には少なくとも 1 個のデータファイルがなければなりません。そのファイルは、データファイルあるいは BLOB ファイルいずれのファイルでもかまいません。初期設定では、新規ファイルをデータファイルとして生成されます。BLOB ファイルを作成する場合は、作成時に BLOB と指定する必要があります。

表領域を作成する前に、**dmconfig.ini** ファイルに表領域に関連するデータファイルのファイル名とサイズを指定します。

➲ **例 1**

オペレーティング・システムのファイル名、ページ・サイズと共に、ファイル **f1**、**f2**、**f3** を **dmconfig.ini** で指定する*:*

```
[MY_DB]                          ;データベース名
f1 = /disk1/usr/f1.dat 1000      ;1000 ページのデータファイル
f2 = /disk2/usr/f2.dat 500       ; 500 ページのデータファイル
f3 = /disk1/usr/f3.blb 1000      ;1000 フレームの blob ファイル
```

別々のディスクにある 2 つのデータ・ファイルと 1 つの BLOB ファイルをもつ標準表領域 ts1 を作成する：

```
dmSQL> CREATE TABLESPACE ts1 DATAFILE f1, f2, f3 TYPE=BLOB;
```

➲ **例 2**

データファイルと BLOB ファイルを 1 個ずつもつ自動拡張表領域を作成します。データファイルの初期サイズは 500 ページ、BLOB ファイルの初期サイズは 20 ページです。データファイルあるいは BLOB ファイルが一杯になると、自動的に拡張します。

```
[MY_DB]                          ;データベース名
f4 = /usr/f4.dat 500             ;初期サイズ 500 ページのデータファイル
f5 = /usr/f5.blb 20              ;初期サイズ  20 ページの blob ファイル
```

これらのファイルをもつ新規表領域を作成する：

```
dmSQL> CREATE AUTOEXTEND TABLESPACE ts_aut DATAFILE f4 TYPE=DATA, f5 TYPE=BLOB;
```

ローデバイスファイル

UNIX システムの DBMaster は、物理ファイル名が**/dev/**で始まるファイルをローデバイスファイルとみなします。ローデバイスファイルは、通常のファイルよりも高速アクセスを可能にするので、データベースの性能を上げ

るために使用することができます。ローデバイスファイルは、表領域に関連づける前に、ディスクに作成しておかなければなりません。

➲ 例 3

5000 ページのローデバイスファイル f2 の物理ファイル名/dev/rawf2 を指定する：

```
[MY_DB]                    ;データベース名
f2= /dev/rawf2 5000        ;5000 ページのローデバイスファイル
```

上記のローデバイスファイル f2 をもつ標準表領域 **ts_raw** を作成する:

```
dmSQL> CREATE TABLESPACE ts_raw DATAFILE f2;
```

標準表領域を拡張する

標準表領域は 3 通りの方法で拡張することができます：

- 標準表領域に新しいファイルを追加します
- 標準表領域の既存ファイルにページを追加します
- 自動拡張をONにセットする

JDBA Tool を使って、或いは SQL 文と **dmconfig.ini** ファイルの編集を組み合わせでこれらの手順を実行することができます。以下は、**dmconfig.ini** ファイルを編集し、SQL 文を実行して標準表領域を拡張する方法の例です。

➲ 例

この SQL 文を入力する前に、論理ファイル名 file_blob に対応する物理ファイル名とファイルサイズを DBMaster に伝える必要があります。に伝える必要があります。そのためには、**dmconfig.ini** のデータベースセクションに次の文を追記する必要があります。file_blob はデータベース内で使用される論理ファイル名、file.blb はオペレーティングシステムで使用される物理ファイル名です。

```
file_blob = file.blb 120
```

標準表領域に新しいファイルを追加して標準表領域を 120 フレームに拡張する例を示します。BLOB ファイル file_blob を標準表領域 **ts_app** に追加する:

```
dmSQL> ALTER TABLESPACE ts_app ADD DATAFILE file_blob TYPE = BLOB;
```

標準表領域 **ts_app** にあるデータ・ファイル file_data に 100 ページを追加する：

```
dmSQL> ALTER DATAFILE file_data ADD 100 PAGES;
```

DBMaster は、ページを追加してファイルサイズを変更した後に、**dmconfig.ini** にあるファイルのページ数を新しい値に更新します。

自動拡張表領域の拡張

自動拡張表領域の拡張方式が以下のようになります。

- 常に最初のファイルから拡張されます。そうすると、パフォーマンスをアップすることができますが、表領域内のあらゆるファイルが均等に拡張されていません。
- 常に最小のファイルから拡張されます。そうすると、表領域でのあらゆるファイルが均等に拡張されますが、テーブルでの全ての行は順次にあらゆるファイルに分散されますので、パフォーマンスを下がる可能性があります。
- まずは最小のファイルから拡張され、現行ファイルのサイズが次の最小ファイルとDB_EXTHDの値の和を上回るまで、次の最小ファイルを拡張します。そうすると、パフォーマンス及びファイルサイズのバランスを取ることができます。
- 上記の如何なる方法についてですが、選ばれたファイルがディスクフル、ファイルシステム制限及びDBMasterストレージ制限などの原因で拡張できない場合、DBMasterは次のファイルを拡張することになります。次のファイルも前と同じ原因で拡張できないと、あらゆるファイルが拡張し終わるまで、DBMasterはずっと次のファイルを拡張し続けます。これに関する詳細については、**DB_EXTHD**をご参照ください。
- JConfiguration Toolを使って、或いはSQL文と**dmconfig.ini**ファイルの編集を組み合わせでこれらの機能を実行することができます。以下は、**dmconfig.ini**ファイルを編集し、SQL文を実行して自動拡張表領域を拡張する方法の例です。

➲ 例 1

キーワード **DB_EXTNP** 及び **DB_EXTHD** を使って、或いは SETSYSTEMOPTION('EXTHD','newValue')をコールして、自動拡張表領域の拡張範囲を指定します。DBMaster はユーザーが定義したストラテジーに従って、自動的に自動拡張表領域を拡張します。

dmconfig.ini ファイルで、自動拡張表領域を拡大するための DBMaster のサイズを 30 ページに指定し、ファイルを繰り返し拡張する閾値を 100 ページに指定します。

```
DB_EXTNP=30
DB_EXTHD=100
```

dmconfig.ini ファイルを変更することにより、下記の五つのファイルを追加します。

```
D1=/home/dbmaster/testdb/D1 10
B1=/home/dbmaster/testdb/B1 30
D2=/home/dbmaster/testdb/D2 50
B2=/home/dbmaster/testdb/B2 70
D3=/home/dbmaster/testdb/D3 100
```

dmSQL プロンプトで下記のコマンドを入力して、自動拡張表領域 TS を作成します。

```
dmSQL> CREATE AUTOEXTEND TABLESPACE TS DATAFILE D1 TYPE=DATA, B1 TYPE=BLOB,
D2 TYPE=DATA, B2 TYPE=BLOB, D3 TYPE=DATA;
```

表領域 **TS** でカラム（**c1 char(5000)**）がある表 **tb_t1** を作成して、当該表に幾つかの行を挿入してから、表領域 **TS** が自動的に拡張可能です。この規則に従って、ファイルが以下のように拡張します。

```
1st, extend the smallest file D1, add 30 pages. Now, D1=40Now, D1=70, D2=50,
D3=100
3rd, extend D1, add 30 pages. Now, D1=100, D2=50, D3=100
4th, extend D1, add 30 pages, Now, D1=130, D2=50, D3=100
5th, extend D1, add 30 pages, Now, D1=160, D2=50, D3=100
6th, extend D2, because D1 > D2+EXTHD. Add 30 pages, D1=160, D2=80, D3=100
7th, extend D2, until D2 > D3(the smallest)+EXTHD, then extend D3.
```

注　*表tb_t1にBLOBフィールドが存在しないため、ファイルB1及びB2が拡張できません。*

➲ 例 2

実行時に、ユーザーはシステムストアドプロシージャ **SETSYSTEMOPTION** をコールすることによって、システムオプション **EXTHD** が変更できます。

```
dmSQL> Call SETSYSTEMOPTION('EXTHD','1000'); // EXTHD を 10000 ページに変更します
```

実行時に、ユーザーはシステムストアドプロシージャ GETSYSTEMOPTION をコールすることによって、システムオプション **EXTHD** の値が確認できます。

```
dmSQL> Call GETSYSTEMOPTION('EXTHD',?)  // EXTHD の現在の値を報告します
```

表領域にファイルを追加する

表領域に新規ファイルを作成して追加することにより、標準の表領域または自動拡張表領域、またその結果としてデータベースのサイズが拡大されます。 データ行を挿入または更新できる領域を増加するには、データファイルを通常の表領域または自動拡張表領域に追加します。BLOB データに使用できるサイズを大きくしたいときは、BLOB ファイルを追加します。JDBA Tool を使うか、**dmconfig.ini** ファイルを修正するか、dmSQL にコマンドを入力して、表領域にファイルを追加することができます。以下は、**dmconfig.ini** ファイルを修正してファイルを追加する方法と、dmSQL にコマンドを入力する方法の概要です。表領域にファイルを追加するときは、表領域を作成するときと同様に、追加するファイル名とサイズを **dmconfig.ini** ファイルに指定しておきます。BLOB ファイルを追加するときは、BLOB ファイルタイプも指定しなければなりません。ファイルタイプを指定しないと、DBMaster は DB ファイルとして追加します。

➲ 例 1

3000 ページの DB ファイル ***f7*** の物理ファイル名 ***/disk1/usr/f7.dat*** を指定する:

```
[MY_DB]                         ;データベース名
f7 = /disk1/usr/f7.dat 3000     ;3000 ページの db ファイル
```

DB ファイル ***f7*** を表領域 **ts_reg** に追加する：

```
dmSQL> ALTER TABLESPACE ts_reg ADD DATAFILE f7;
```

➲ 例 2

5000 ページの BLOB ファイル名 ***f8*** の物理ファイル名 ***/disk1/usr/f8.blb*** を指定する：

```
[MY_DB]                          ;データベース名
f8 = /disk1/usr/f8.blb 5000      ;5000 フレームの blob ファイル
```

この BLOB ファイルを表領域 **ts_reg** に追加する：

```
dmSQL> ALTER TABLESPACE ts_reg ADD DATAFILE f8 TYPE=BLOB;
```

ファイル・タイプは、初期設定でデータとして記述、又は追加する必要があります。

表領域内のファイルにページを追加する

標準表領域にファイルを追加してデータベースを拡大するだけでなく、標準表領域のファイルサイズを大きくしてデータベースを拡大することもできます。自動拡張表領域のファイルサイズを変更して予めディスク領域を割り当てておき、パフォーマンスを上げることもできます。ファイルのサイズを変更すると、DBMaster は、**dmconfig.ini** にあるファイルのページ数を自動的に更新します。

➲ 例

ファイル ***f1*** に 100 ページ追加してサイズを拡大します(ファイル ***f1*** は、いずれかの表領域に属して存在していなければなりません)。

```
dmSQL> ALTER DATAFILE f1 ADD 100 PAGES;
```

標準表領域を自動拡張表領域に変更する

以下の場合に表領域を標準表領域から自動拡張に変更することができます。

- 標準表領域にデータを加えたいが、表領域が既にディスクで使用できるスペース一杯になっている。これを自動拡張表領域に変換し、別のディスクの表領域にファイルを追加する
- 標準表領域を自動拡張表領域に変更した後、そのサイズを広げるだめに表領域にファイルを追加することができます。

標準表領域を作成した後、JDBA Tool や dmSQL の ALTER TABLESPACE 文を使って自動拡張表領域に変更することができます。

➲ 例

標準表領域 **ts_reg** を自動拡張表領域に変更する:

```
dmSQL> ALTER TABLESPACE ts_reg SET AUTOEXTEND ON;
```

自動拡張表領域を標準表領域に変更する

以下の場合に表領域を自動拡張から標準表領域に変更することができます。

- 自動拡張表領域にデータを追加したいが、表領域がディスクを一杯まで使い切っている。自動拡張表を標準表領域に変更し、別のディスク上のファイルを表領域に追加する。
- 表領域が占めるディスク容量を制限したい（自動拡張表領域は、最大 2GB まで、ディスクの全使用可能領域で成長します）。
- 自動拡張表領域を標準表領域に変更した後、サイズを拡大するために表領域にファイルを追加することができます。

自動拡張表領域を作成した後に、JDBA Tool や dmSQL の ALTER TABLESPACE 文を使用して標準表領域に変更することができます。

➲ 例

自動拡張表領域 **ts_reg** を標準表領域に変更する:

```
dmSQL> ALTER TABLESPACE ts_reg SET AUTOEXTEND OFF;
```

表領域とファイルの縮小

他の用途にディスクを割り当てる必要がある場合、表領域のサイズを縮小することができます。表領域のサイズを縮小するためのコマンドは、SHRINK DATAFILE 文と SHRINK TABLESPACE 文です。SHRINK DATAFILE コマンドは、ユーザー定義ファイルに適用しますが、SHRINK TABLESPACE コマンドがユーザー定義表領域の全てのファイルに適用します。これらの操作は、JDBA Tool で行うこともできます。以下のセクションでは、dmSQL を用いた表領域のサイズの縮小方法について解説します。

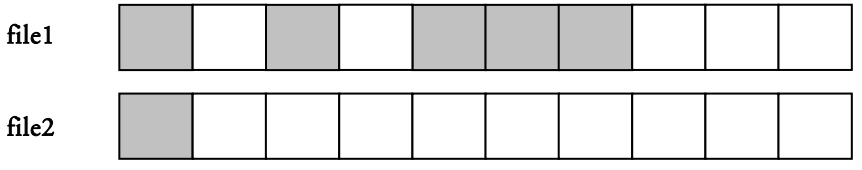
TRUNCATEONLYオプション

TRUNCATEONLY オプションで SHRINK コマンドを実行すると、ファイルの末尾の空きページを切り縮めます。ファイルの圧縮はしません。使用ページ間に空きページがある場合、そのファイルにそのまま残ります。データベース管理者は、ファイルの末尾の空きスペース全てを切り捨てる（WITH n FREE PAGES オプションを指定しない）こともできますし、指定したページ数のみを残す（WITH n FREE PAGES オプションを指定する）ようにすることも可能です。以下に 2 つのオプションについての例を紹介します。

WITH n FREE PAGESオプションを使用しない

TRUNCATEONLY オプションで SHRINK コマンドを実行し、WITH n FREE PAGES オプションを使用しないと、ファイルの末尾の空きページを切り縮めるだけです。

例えば、表領域 **ts_shrink** に **file1** と **file2** があり、灰色のブロックが使用ページを表し、白のブロックが空きページを表している場合、



表領域全体に対し SHRINK TABLESPACE 文を実行、或いは両ファイルに対し SHRINK DATAFILE 文を実行すると、両ファイルの最後の空きページは削除されます。TRUNCATEONLY オプションは、必ず指定します。次に実行例を紹介します。

➲ **例 1**

```
dmSQL> SHRINK TABLESPACE ts_shrink TRUNCATEONLY;
```

➲ **例 2**

```
dmSQL> SHRINK DATAFILE file1 TRUNCATEONLY;
dmSQL> SHRINK DATAFILE file2 TRUNCATEONLY;
```

両ファイルの最後のページを切り落とした後、ファイルの状態は以下のようになります。

結果

file2 の全ページが空いていても、DBMaster は最低 2 ページを残します（1 つは PE ページで、もう 1 つはデータページ）。

WITH n FREE PAGESオプションを使用する

WITH n FREE PAGES オプションは、ページ切り落としの後、ファイルに残すファイルの最後の空きスペースのページ数（PE ページを含まない）を指定します。

上述の file1 と file2 を用いて、次の各文を実行します。

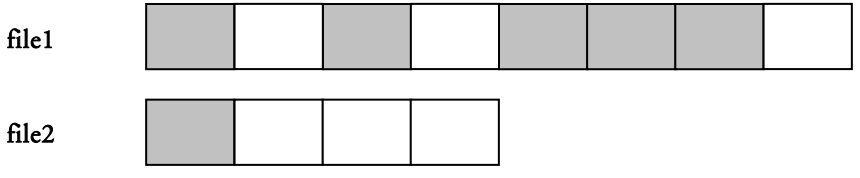
➲ 例 1

```
dmSQL> SHRINK TABLESPACE ts_shrink TRUNCATEONLY WITH 3 FREE PAGES;
```

➲ 例 2

```
dmSQL> SHRINK DATAFILE file1 TRUNCATEONLY WITH 3 FREE PAGES;
dmSQL> SHRINK DATAFILE file2 TRUNCATEONLY WITH 3 FREE PAGES;
```

結果



SHRINK TABLESPACE コマンドと WITH FREE PAGES オプションは、表領域の各ファイルに個々に適用されます。上記の場合、同じ表領域の各ファイルに 3 空きページが残ります。

ファイルに 50 空きページがある場合、WITH 80 FREE PAGES オプションを実行すると、何も起こりません。SHRINK コマンドの後、50 空きページがそのまま存在し、30 空きページ(80 - 50)を追加してファイルを拡大することにはなりません。

SHRINK コマンドは、自動コミットが ON の状態で実行する必要があります。TRUNCATEONLY オプションは、ロールバックされません。ユーザーは、クラッシュ回復の際もこのコマンドをロールバックすることができません。

COMPRESSONLYオプション

SHRINK TABLESPACE コマンドのみ COMPRESSONLY オプションをサポートしています。これは、表領域の各ファイルを圧縮します。圧縮や移動によって同じページのレコードを圧縮しません。単位はページです。前方の空きページに、ファイルの後ろにある使用ページを移動します。コマンドを実行すると、全ての空きページはファイルの後方に置かれ、前方に全使用ページが置かれます。

結果 1：

File1 には、5 つの連続しない使用ページがあります。

➲ **例**

使用ページをつなげる:

```
dmSQL> SHRINK TABLESPACE ts_shrink COMPRESSONLY;
```

結果 2:

SHRINK コマンドは自動コミット ON の状態で実行する必要があります。COMPRESSONLY オプションはロールバックされます。データベースがク

ラッシュした場合、クラッシュ回復の後、COMPRESSONLY 操作は、全て実行又は全て失敗します。

COMPRESSONLY オプションを使った SHRINK コマンドとバックアップは競合します。DBMaster では、同時にこれら 2 つのコマンドを実行することができません。

表領域の縮小と圧縮の制限

これらのコマンドの一般的な制限は以下のとおりです。

- SHRINKコマンドは、データとBLOBファイル上で実行できますが、ジャーナル・ファイルではできません
- DBA以上のユーザーのみSHRINKコマンドを実行できます
- SHRINKコマンドは、自動コミットがONにする必要があります
- SHRINKコマンドは、DBMaster 3.7に追加されましたので、DBMasterの初期のバージョンではこのコマンドは認識しません。つまり、一旦DBAがSHRINKコマンドを実行し、それで増分バックアップをすると、DBMasterの初期のバージョンではそのジャーナル・バックアップ・ファイルをリストアできません
- TRUNCATEONLYオプションは、ロールバックできません
- COMPRESSONLYオプションは、SYSTABLESPACE表領域を圧縮できません
- COMPRESSONLYオプションは、ユーザー表にOIDカラムがあるかどうかをチェックできません。このコマンドは、OIDの意味を認識しません。OIDは同じデータベースのものを指す場合もありますし、リモート・データベースのものを指す場合もあります。このコマンドはユーザー表のOIDカラムを修正しません
- COMPRESSONLYオプションとバックアップは、同時に実行できません

表領域を削除する

システム表領域と初期設定表領域以外の表領域は、空であるか必要な情報がない時、データベースから削除することができます。表領域を削除する

場合は、まず表領域内の全ての表を削除するか、表が空であることを確認します。表領域から表を削除する方法については、6章の「スキーマ・オブジェクト管理」を参照してください。

表領域を削除すると、表領域に属するファイルも自動的にデータベースから削除されます。ただし、オペレーティングシステムのファイルシステムからは削除されません。これらのファイルは、依然としてファイルシステムに存在します。ファイルが占有するディスクをシステムに戻すには、オペレーティングシステムの削除コマンドを使用します。表領域の物理ファイルがファイルシステムから削除されると、ファイル内のデータをリカバリすることはできません。表領域に関連するファイルの削除には十分注意してください。さもないと、大切なデータを失うかもしれません。

JDBA Tool や dmSQL の DROP TABLESPACE 文で表領域を削除することができます。

➲ **例**

表領域 **ts_aut** に関連する全てのファイルを削除する:

```
dmSQL> DROP TABLESPACE ts_aut;
```

表領域からファイルを削除する

JBDA Tool ツールまたは ALTERTABLESPACE 表領域名 DROP DATAFILE ファイル名コマンド dmSQL コマンドプロンプトで使用することにより、不要なデータファイルを削除することができます。後者を使用する場合、ALTER TABLESPACE tablespace-name DROP DATAFILE コマンドをコミットした後、物理データファイルと **dmconfig.ini** の情報を手動で削除してください。

不要なデータファイルは、次の条件を満たす表領域から削除できます:

a) データファイルがその表領域の唯一のデータファイルである場合、表領域からデータファイルを削除することはできません。

b) 削除するデータファイルは空でなければなりません。

c) システム、またはシステムの初期設定のデータファイル、または初期設定の表領域は削除できません。

➲ 例

表領域 **ts_aut** からデータファイル **f4** を削除する:

```
dmSQL> ALTER TABLESPACE ts_aut DROP DATAFILE f4;
```

読み取り専用テーブルスペース

読み取り専用テーブルスペースはテーブルスペースでの更新や新しいオブジェクトの作成を許可しないテーブルスペースです。

テーブルスペースを読み取り専用に設定すると、多くのメリットがあります:

バックアップの実行を削減できる。読み取り専用にした後、1 度のバックアップだけでよい。

- 復元が容易になる
- 読み取り専用テーブルスペースはロックの必要がないため更新可能のテーブルスペースよりもオーバーヘッドが少ない
- I/Oが減少する

➲ 例

テーブルスペース **ts_reg** を読み取り専用テーブルスペースに設定する方法

```
dmSQL> ALTER TABLESPACE ts_reg SET READ ONLY;
```

表領域 **ts_reg** を読み出し/書き込み可能表領域に変更：

```
dmSQL> ALTER TABLESPACE ts_reg SET READ WRITE;
```

表領域とファイルの情報を取得する

JDBA Tool を使うと、表領域の構造と指定した表領域の中のファイルを簡単に見ることができます。表領域は、全データベース・オブジェクトの論理ツリー構造の一部として表示されます。ツリーの表領域ノードを選択すると、データベース内にある全表領域がファイル詳細と共に表示されます。ツリーから表領域を選ぶと、表領域の全ファイルと共に、ファイルのサイ

ズ、物理ロケーション、データの種類、表領域の拡張性のような詳細が表示されます。

また、dmSQL を使ってシステム表 SYSTABLESPACE に問合せて表領域の情報を取得し、SYSFILE システム表からはユーザーBLOB ファイルとユーザー・データファイルの情報を取得することができます。

➲ **例 1**

表領域名、標準か自動拡張か、領域内のファイル数、合計ページ数のような表領域の情報を、システム表 SYSTABLESPACE から取得する：

```
dmSQL> SELECT * FROM SYSTABLESPACE;
```

➲ **例 2**

同様にファイル情報も取得できます。システム表 SYSFILE を検索し、ファイル名、ファイルの種類、データベース内部のファイル識別名、ファイルが属する表領域名、ファイルのページ数等の情報を取得する：

```
dmSQL> SELECT * from SYSFILE;
```

システムカタログ表 SYSTABLESPACE と SYSFILE については、システムカタログ参照を参照してください。

ファイルと表領域の整合性をチェックする

DBMaster には、様々なデータベースの部分の整合性をチェックする 6 つのコマンドがあります。これらのコマンドは、データベースが大きい場合は時間がかかり、ロックをかけますので、必要な時のみ使用します。ファイルと表領域の整合性は、これらのコマンドの 1 つを使ってチェックすることができます。CHECK FILE コマンドは、ファイルが壊れているかどうか、或いは表領域に正しい表が入っているかどうかをチェックします。

ファイルをチェックする

データファイルの各ページ（フレーム）の内容をチェックすることができます。通常、ディスク障害が発生したときには、ファイルが破壊されているかどうかチェックします。

➲ 例

データファイル **FILE1** の整合性をチェックする：

```
dmSQL> CHECK FILE FILE1;
```

表領域をチェックする

表領域に関連するファイルと表をチェックすることができます。個々のファイルと表は、CHECK FILE 文と CHECK TABLE 文と同じ方法でチェックされ、これらの SQL 文を直接実行したときと同じ結果を返します。

➲ 例

表領域 **ts_reg** の整合性をチェックする:

```
dmSQL> CHECK TABLESPACE ts_reg;
```

6 スキーマ・オブジェクト管理

本章では、DBMaster の種々のスキーマ・オブジェクト管理について解説します。スキーマ・オブジェクトには、表、ビュー、シノニム、索引、シリアル番号、データ整合性、ドメイン等が含まれます。

さらに、システム・カタログ表をブラウズしてスキーマ・オブジェクト情報を取得し、表と索引のディスク容量を見積る方法についても説明します。

スキーマ・オブジェクト管理は、dmSQL のコマンドや JDBA Tool で行うことができます。JDBA Tool は、直感型のグラフィカル・インターフェースで、主なデータベース管理業務を行うための使いやすいウィザードがあり、データベースの論理的な構造を表示しています。JDBA Tool を使うと、DBMaster を使いはじめたユーザーにとって、スキーマ・オブジェクト間の関係を理解する上で役に立ちます。熟練ユーザーは、論理的な画面を参照し、データベース・スキーマの作成と管理に役立てることができます。以下の節では、dmSQL を使ってデータベースのスキーマ・オブジェクトを管理する方法の例を紹介します。JDBA Tool を使ったスキーマ・オブジェクト管理についての詳細は、*「JDBA Tool ユーザーガイド」*を参照して下さい。DBMaster に SQL 言語を使用する方法についての詳細情報は*「SQL 文と関数参照編」*を参照してください。

6.1 スキーマを管理する

スキーマは名前領域(データベース・オブジェクトの論理グループ)です。スキーマには、表、ビュー、索引、コマンド、プロシージャ、ドメインとシノニムなどのスキーマ・オブジェクトが含まれます。

CREATE SCHEMA は新規スキーマを定義します。 スキーマを作成した後、スキーマ内部にオブジェクトを作成できます。 スキーマ所有者は与えられた権限の譲与者です。

スキーマの所有者は次のように決定されます:

- AUTHORIZATION条件項が指定されている場合、指定されたユーザー名はスキーマ所有者です。 スキーマ名を省略する場合、指定されたユーザー名はスキーマ名として使用されます。

例えば

```
dmSQL> CREATE SCHEMA AUTHORIZATION JEFFERY;
```

- AUTHORIZATION条件項が指定されていない場合、CREATE SCHEMA 文を発行するユーザーはスキーマ所有者です。

➲ 例 1

RESOURCE 権有するユーザーとして、JEFFERY は **SS1** と呼ばれるスキーマを作成します。**SCH_JEF** は初期設定の所有者です。

```
dmSQL> CREATE SCHEMA SCH_JEF;
```

➲ 例 2

DBA 権を有するユーザーの場合、所有者としてユーザーJEFFERY を有するスキーマを作成すると、ユーザー名 JEFFERY は初期設定のスキーマ名になります。

```
dmSQL> CREATE SCHEMA AUTHORIZATION JEFFERY;
```

➲ 例 3

DBA 権を有するユーザーの場合、所有者としてユーザーJEFFERY と共に **SCH_ForJEF** と呼ばれるスキーマが作成されます。

```
dmSQL> CREATE SCHEMA SCH_ForJEF AUTHORIZATION JEFFERY;
```

➲ 例 4

DBA 権限を持つユーザがスキーマ inventory を作成します。ユーザはその後に表と表に対して索引を作成します。最後に表の権限をユーザ JEFFERY に与えます。

```
dmSQL> CREATE SCHEMA inventory;
dmSQL> CREATE TABLE inventory.part (partNo smallint not null, quantity int);
dmSQL> CREATE INDEX partind ON inventory.part (partNo);
dmSQL> GRANT ALL ON inventory.part TO JEFFERY;
```

DROP SCHEMA を行うとデータベースからスキーマが削除されます。スキーマはその所有者もしくは DBA 権限のユーザによってのみ削除されます。注意点としてスキーマ所有者はスキーマにオブジェクトが残っている場合、スキーマの削除はできません。

➲ **例**

スキーマ SCH_JEF をデータベースから削除します。

```
dmSQL> DROP SCHEMA SCH_JEF;
```

注 *SYSADMは新規ユーザーに権限を与える場合, 新規ユーザー名はデータベースに任意のユーザーから作成したスキーマ名と異なります。*

情報スキーマ

DBMaster の各データベースには、INFORMATION_SCHEMA と呼ばれるスキーマが含まれています。スキーマには、データベースに属する各オブジェクトの説明を表示できる、ただし変更はできない一連のビューが含まれます。

DBMaster はメタデータを取得するための情報スキーマビューを備えています。これらのビューには、DBMaster のメタデータのシステムテーブルに従属しない内部ビューが含まれます。情報スキーマビューを使えば、システムテーブルに重大な変更が加えられても、アプリケーションは正しく機能します。DBMaster に含まれる情報スキーマビューは、INFORMATION_SCHEMA の SQL-92 規格定義に対応しています。

DBMaster は現在のサーバを参照する場合、3 部の命名規則をサポートしています。SQL-92 規格も 3 部の命名規則をサポートしています。しかし、両方の命名規則で使用される名前は異なります。それらのビューは、各データベースに含まれる INFORMATION_SCHEMA という名前の特別なスキーマで定義されます。各 INFORMATION_SCHEMA ビューには、その特定の

データベースに保存されるすべてのデータオブジェクトのメタデータが含まれます。この表は DBMaster と SQL-92 規格との関係を示しています。

DBMaster 名	相当する SQL-92 名
データベース	カタログ
所有者	スキーマ
オブジェクト	オブジェクト
ユーザー定義データ型	ドメイン

命名規則のマッピングはこれらの DBMaster SQL-92 対応ビューに適用されます。

それらのビューは、各データベースに含まれる INFORMATION_SCHEMA という名前の特別なスキーマで定義されます。各 INFORMATION_SCHEMA ビューには、その特定のデータベースに保存されるすべてのデータオブジェクトのメタデータが含まれます。

INFORMATION_SCEHMA ビューは以下の通りです。

COLUMN_DOMAIN_USAGE

COLUMN_PRIVILEGES

COLUMNS

DOMAINS

SCHEMATA

TABLE_PRIVILEGES

TABLES

VIEW_COLUMN_USAGE

VIEW_TABLE_USAGE

VIEWS

6.2 表管理

表は、DBMaster がデータを格納するストレージの論理単位です。表は、カラム（列）と行から構成されます。カラムは、フィールドあるいは属性とも呼ばれます。行は、レコードあるいはタプル（組）とも呼ばれます。

DBMaster の表は、所有者名と表名によって識別されます。

例えば、ユーザー *Jeff* と *Kevin* が表名 *friend* の表を作成すると、表名 *Jeff.friend* と *Kevin.friend* の 2 つ別の表が作成されます。

JDBA Tool では、データベースにある全表は論理ツリーの表ノードで表示されます。表をクリックすると、表スキーマを見ることができます。

表を作成する

表は、表名と 1～2000 のカラムの集まりで定義します。

各カラムは、以下で定義します。

- カラム名とカラムのデータ型または後節ドメイン管理で説明するドメイン
- カラムのサイズ（INTEGERデータ型のように予め決まっている場合以外）、カラムの精度とスケール（DECIMALデータ型のみ）、カラムの開始番号（SERIALデータ型のみ）

DBMaster は、カラムの定義に使用する数多くのデータ型をサポートします。数値型（SMALLINT、INTEGER、BIGINT、FLOAT、DOUBLE、DECIMAL、SERIAL、BIGSERIAL）、バイナリ型（BINARY、VARBINARY、CHAR、VARCHAR）、BLOB 型（LONG VARCHAR、LONG VARBINARY、FILE）、日付時刻型（DATE、TIME、TIMESTAMP）があります。データ型の詳細については、*「SQL 文と関数参照編」*をご覧下さい。

表を作成するためには、表名、カラム定義、所属する表領域名を指定します。表領域を指定しない場合、初期設定表領域に置かれます。JDBA Tool の表作成ウィザードや dmSQL のコマンドで表を作成することができます。JDBA Tool についての詳細は、*「JDBA Tool ユーザーガイド」*を参照して下さい。次の例は、dmSQL を使った表の作成方法の例です。SQL 文の CREATE TABLE の構文と使用方法については、*「SQL 文と関数参照編」*をご覧下さい。

➲ 例

表領域 ts_reg に表 tb_staff を作成する：

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20),
                                  ID INTEGER,
                                name CHAR(30),
                             joinDate DATE,
                               height FLOAT,
                               degree VARCHAR(200),
                              picture LONG VARCHAR) IN ts_reg;
```

DBMaster には、表作成時に適用することができる役立つ機能がたくさんあります。

- カラムに初期値を定義する。
- カラムをNULL不可に指定する。
- 表に主キー、又は外部キーを指定する
- ロックモード、フィルファクタ、非キャッシュ（NOCACHE）オプションを指定してデータベースの効率を改善する。
- 一時表に指定する。

カラム初期値

表のカラムに初期値を設定することができます。新しい行を挿入するときにカラムの値が指定されていないと、自動的にカラム初期値がカラムに挿入されます。

カラム初期値の指定はオプションです。初期値を指定しない場合は、NULL がカラム初期値になります。

初期値は、定数または組み込み関数です。組み込み関数の詳細については、*「SQL 文と関数参照編」*をご覧下さい。

DBMaster はキーワード USER、SYSTEM と ON UPDATE で挿入及び更新操作をしている際のディフォルトカラムの属性を設定することをサポートします。キーワード USER/SYSTEM が選択可能です。これらのキーワードは、ユーザーが INSERT/UPDATE ステートメントを使用してデフォルト値を持

つカラムの値を変更できるかどうかを指定します。キーワード USER がデフォルトとして使用されます。キーワード USER はユーザーがその値を変更できるのを指定し、キーワード SYSTEM はユーザーがその値を変更できないのを指定します。キーワード ON UPDATE は選択可能です。このキーワードは、その他のカラムの値が変更される場合に、デフォルト値を持つカラムの値が自動的に変更できるのを指定します。この三つのキーワードは主に表定義に用いられ、表定義に関する詳しい情報については、*「SQL 文と関数参照編」の CREATE TABLE、ALTER TABLE ADD COLUMN* 及び *ALTER TABLE MODIFY COLUMN* とのセクションをご参照ください。

INSERT/UPDATE 文を使用してカラムの値を指定する場合、データベースの表をロードしている間に SYSTEM DEFAULT 属性を持つカラムの値が上書きされるかどうかを指定することができます。このオプションを *ON* に設定する場合は、その値はデフォルト値に更新され、このオプションを *OFF* に設定する場合は、元の値がユーザーによって指定された値に更新されます。このオプションのデフォルト設定は *OFF* です。

➲ **例 1**

表 ***tb_staff*** のカラム ***nation*** の初期値を定数'R.O.C.'に指定し、カラム ***joinDate*** の初期値を組み込み関数 **curdate()**に指定する:

```
dmSQL> CREATE TABLE tb staff (nation CHAR(20) DEFAULT 'R.O.C',
                                  ID INTEGER,
                                name CHAR(30),
                            joinDate DATE DEFAULT CURDATE(),
                              height FLOAT,
                              degree VARCHAR(200),
                             picture LONG VARCHAR) IN ts_reg;
```

➲ **例 2a**

```
dmSQL> CREATE TABLE computer(id INT, buy_time TIMESTAMP DEFAULT '2012-03-04 12:12:12',
price int); //buy_time の属性は USER です。
dmSQL> INSERT INTO computer VALUES(1, '2012-10-10 10:10:20', 3400); //buy_time の値
はユーザーが指定する'2012-10-10 10:10:20'に変更されます。
1 rows inserted
dmSQL> INSERT INTO computer VALUES(2, '2012-10-11 10:10:20', 5400);
1 rows inserted
dmSQL> SELECT * FROM computer;
```

```
    ID              BUY TIME             PRICE
========== ========================== ==========
         1 2012-10-10 10:10:20               3400
         2 2012-10-11 10:10:20               5400
2 rows selected
dmSQL> UPDATE computer SET price=3200 WHERE id=1; //buy_time の値が更新されていません。
1 rows updated
dmSQL> SELECT * FROM computer;
    ID              BUY_TIME             PRICE
========== ========================== ==========
         1 2012-10-10 10:10:20               3200
         2 2012-10-11 10:10:20               5400
2 rows selected
```

➲ 例 2b

```
dmSQL> ALTER TABLE computer MODIFY (buy_time TO buy_time TIMESTAMP DEFAULT '2012-03-04
12:12:12' ON UPDATE); // buy_time の属性は USER および ON UPDATE です。
dmSQL> UPDATE computer SET price=3000 WHERE id=1; // buy_time の値は'2012-03-04
12:12:12'に変更されます。
1 rows updated
dmSQL> SELECT * FROM computer;
    ID              BUY_TIME             PRICE
========== ========================== ==========
         1 2012-03-04 12:12:12               3000
         2 2012-10-11 10:10:20               5400
2 rows selected
dmSQL> UPDATE computer SET price=3000, buy_time='2012-10-10' WHERE id=1;// buy_time
の値はユーザーが指定する'2012-10-10'に変更されます。
1 rows updated
dmSQL> SELECT * FROM computer;
    ID              BUY_TIME             PRICE
========== ========================== ==========
         1 2012-10-10 00:00:00               3000
         2 2012-10-11 10:10:20               5400
2 rows selected
```

➲ 例 2c

```
dmSQL> ALTER TABLE computer MODIFY (buy_time TO buy_time TIMESTAMP SYSTEM DEFAULT
'2012-03-04 12:12:12'); // buy_time の属性は SYSTEM
```

```
dmSQL> INSERT INTO computer VALUES(3, '2012-11-10 10:10:20', 4700); // buy_time の
値はユーザーが指定する'2012-11-10 10:10:20'に変更されません。
1 rows inserted
dmSQL> INSERT INTO computer VALUES(4, '2012-12-11 10:10:20', 2800);// buy_time の値
はユーザーが指定する'2012-12-10 10:10:20'に変更されません。
1 rows inserted
dmSQL> SELECT * FROM computer;
    ID              BUY_TIME              PRICE
========== ========================= ==========
         1 2012-10-10 00:00:00               3000
         2 2012-10-11 10:10:20               5400
         3 2012-03-04 12:12:12               4700
         4 2012-03-04 12:12:12               2800
4 rows selected
dmSQL> UPDATE computer SET price=4500 WHERE id=3; // buy_time の値が更新されていません。
1 rows updated
dmSQL> SELECT * FROM computer;
    ID              BUY_TIME              PRICE
========== ========================= ==========
         1 2012-10-10 00:00:00               3000
         2 2012-10-11 10:10:20               5400
         3 2012-03-04 12:12:12               4500
         4 2012-03-04 12:12:12               2800
4 rows selected
```

➲ 例 2d

```
dmSQL> ALTER TABLE computer MODIFY (buy_time TO buy_time TIMESTAMP SYSTEM DEFAULT
'2012-03-04 12:12:12' ON UPDATE); //buy_time の属性は SYSTEM 及び ON UPDATE です。
dmSQL> UPDATE computer SET price=4000, buy_time='2015-01-01' WHERE id=3; //buy_time
の値はディフォルト値'2012-03-04 12:12:12'に変更されます。
1 rows updated
dmSQL> SELECT * FROM computer;
    ID              BUY_TIME              PRICE
========== ========================= ==========
         1 2012-10-10 00:00:00               3000
         2 2012-10-11 10:10:20               5400
         3 2012-03-04 12:12:12               4000
         4 2012-03-04 12:12:12               2800
4 rows selected
```

NULL値制約

カラムや表に整合性制約と呼ばれる制限を設けることができます。1 つの例がカラムに定義する NOT NULL 整合性制約です。この制約は、カラムに NULL 値を入れることができないように制限します。

例えば、***tb_staff***表には必ず新入社員の ID と名前を入れるようにします。

➲ **例**

表 **tb_staff** に新入社員の ID と名前を作成する:

```
dmSQL> CREATE TABLE tb staff (nation CHAR(20) DEFAULT 'R.O.C',
                                 ID INTEGER NOT NULL,
                               name CHAR(30) NOT NULL,
                           joinDate DATE DEFAULT CURDATE(),
                             height FLOAT,
                             degree VARCHAR(200)) IN ts_reg;
```

主キーと外部キー

表所有者は、CREATE TABLE 文で主キー又は外部キーを指定することができます。主キーと外部キーの詳細については、6.9節の「データ整合性管理」を参照して下さい。

ロックモード

表のロックモードは、データベースをアクセスするときに自動的にオブジェクトに設定されるロックの種類を表します。DBMaster には、表（TABLE）、ページ（PAGE）、行（ROW）の 3 段階のロックモードがあります。表作成時にロックモードを指定しない場合、初期設定の行（ROW）ロックになります。表ロックのような高位のロックモードを設定すると、データベース・アクセスの同時実行性は低下しますが、使用するロックリソース（共有メモリ）は少なくなります。低位のロックモードを設定すると、データベース・アクセスの同時実行性は高くなりますが、使用するロックリソース（共有メモリ）は多くなります。例えば、ロックモードが表の時に行の挿入/修正を行うと、他の利用者はその表にアクセスできません。表全体に排他ロックがかけられるからです。ロックモードの詳細については、「ロック」を参照してください。

➲ 例

表にロックモードを指定する：

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                  ID INTEGER NOT NULL,
                                name CHAR(30) NOT NULL,
                            joinDate DATE DEFAULT CURDATE(),
                              height FLOAT,
                              degree VARCHAR(200)) IN ts reg
                      LOCK MODE ROW;
```

フィルファクタ

フィルファクタ（FILLFACTOR）は、データページに格納されているレコードを拡張するときのために空き領域を確保しておき、データページの使用効率を最適化する機能です。ページのフィルファクタが指定したパーセントに達すると、新規レコードは別のページに挿入するようにします。同じページ内でレコードが大きくなれるようにすることによって、1 つのレコードを複数ページから検索しないようにし、効率よくアクセスできるようになります。

➲ 例

表 **tb_staff** のフィルファクタを 80%にする:

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                  ID INTEGER NOT NULL,
                                name CHAR(30) NOT NULL,
                            joinDate DATE DEFAULT CURDATE(),
                              height FLOAT,
                              degree VARCHAR(200)) IN ts reg
                    LOCK MODE ROW
                    FILLFACTOR 80;
```

この場合、ページの 80%以上を使用すると、データページには新しい行を挿入しなくなります。FILLFACTOR は 50～100 に設定することができます。初期値は 100 です。

非キャッシュ

NOCACHE は、大きい表を検索してアクセスするときに役に立つオプションです。DBMaster は、データベースをアクセスするときに、検索データを共有メモリのページバッファにキャッシュしてディスク I/O が頻発するのを回避しますが、大きい表の検索では、依然としてディスク I/O が頻発することがあります。ページバッファの個数よりも大きいデータページをもつ表を検索するときは、全てのページバッファを使い切ってしまい、ディスク I/O が頻発します。

表を作成するときに NOCACHE オプションを指定すると、DBMaster は、表検索の間、検索データを 1 ページバッファ分のみキャッシュします。これによって、全てのページバッファが一つの表によって使われてしまうことを防止します。

➲ **例**

NOCACHE オプションを指定する：

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                 ID INTEGER NOT NULL,
                               name CHAR(30) NOT NULL,
                           joinDate DATE DEFAULT CURDATE(),
                             height FLOAT,
                             degree VARCHAR(200)) IN ts_reg
                    LOCK MODE ROW
                    FILLFACTOR 80
                    NOCACHE;
```

一時表

一時的にデータを格納するために一時表 を作成することができます。一時表は、セッションの間のみ存在します。データベースから切断すると、一時表は自動的に削除されます。一時表は、素早いデータ操作をサポートし、表作成者のみ使用することができます。SQL 構文で TEMPORARY キーワードの替わりに、TEMP または LOCAL TEMPORARY を使うこともできます。

➲ **例**

一時表 tb_student を作成する：

```
dmSQL> CREATE TEMPORARY TABLE tb_student (name CHAR(25) NOT NULL,
                                          birthday DATE,
                                            score INTEGER);
```

自動更新統計

表は頻繁に書き入れると統計値は非常に重要です。ユーザーは表を作成する場合、統計更新の時間間隔を指定すると、DBMaster は自動的に定時に統計が更新できます。

➲ **例 1**

表を作成して、または隔七日に自動的に更新します：

```
dmSQL> CREATE TABLE tb_student (name CHAR(25) NOT NULL,
                                birthday DATE,
                                  score INTEGER)
       UPDATE STATISTICS EVERY 7 DAYS;
```

データベースは実行している場合、ユーザーはシステムストアドプロシージャ SetSystemOption を使用できて指定された統計値が変更することができます。

➲ **例 2**

以下の例示はデータベースは実行している場合に更新統計サンプルを 60 に設定するため使用されます：

```
dmSQL> call setSystemOption('STSSP', '60');
```

データベースを起動した後統計が更新できます。またプロセッサーソースも必要のため、データベースの性能を影響されます。その状況を避けるため、適当な時間と時間間隔を選択して表の使用ピークを避けしたほうがいいです。

表のスキーマを確認する

dmSQL や JDBA Tool を使って、表スキーマを問い合せることができます。JDBA Tool には、表スキーマをグラフィカルに表示し、SQL 文を入力する事

無く、表スキーマを修正することができます。また、dmSQL コマンドの DEF TABLE を使って、表スキーマを直接問い合せることもできます。

➲ 例

表 **tb_staff** のスキーマを確認する:

```
dmSQL> DEF TABLE tb_staff;
CREATE TABLE SYSADM.TB_STAFF (
 NATION  CHAR(20) default 'R.O.C',
 ID  INTEGER not null,
 NAME  CHAR(30) not null,
 JOINDATE  DATE default CURDATE(),
 HEIGHT  FLOAT DEFAULT NULL ,
 DEGREE  VARCHAR(200) DEFAULT NULL)
 in TS_REG  LOCK MODE ROW  FILLFACTOR 80  NOCACHE;
```

表を変更する

表を作成した後に、以下の目的のために表を変更することができます。

- カラムを追加/削除する。
- カラム定義を修正する。
- フィルファクタの値を変更する。
- NOCACHEオプションをON/OFFにする。

dmSQL や JDBA Tool を使って、表のスキーマを変更することができます。

カラムを追加/削除する

表が空であるか否かにかかわらず、表にカラムを追加することができます。空表にカラムを追加するのは、表の最後にカラムを追加して表スキーマを拡張するのと同じです。修正権限のあるユーザーのみが新規カラムを追加することができます。

空でない表にカラムを追加するときは、表スキーマを拡張するだけでなく、追加カラムの全ての行に NULL 値を埋め込みます。従って、空でない表に NOT NULL 整合性制約をもつカラムを追加する場合は、既存のレコードに

代入する値を与えます。SQL 構文の詳細については、*「SQL 文と関数参照編」*をご覧下さい。

➲ **例 1**

表 **tb_staff** にカラム名 ***photo*** のカラムを追加する：

```
dmSQL> ALTER TABLE tb_staff ADD photo LONG VARCHAR;
```

➲ **例 2**

表の既存カラム名の後にカラム名 **city** を追加し、初期設定値を' Taipei'にセットする：

```
dmSQL> ALTER TABLE tb_staff ADD city CHAR(20) default 'Taipei' AFTER name;
```

➲ **例 3**

tb_staff 表にデータがあり、それに非 NULL カラムを追加する場合、**GIVE** キーワードを使って、新規に追加したカラムの既存レコード部分に値を代入することができます。**tb_staff** 表に非 NULL カラム **HireDate** を追加する：

```
dmSQL> ALTER TABLE tb_staff ADD (HireDate data NOT NULL give '2000-02-20');
```

➲ **例 4**

tb_staff 表からカラム photo を削除する：

```
dmSQL> ALTER TABLE tb_staff DROP photo;
```

カラム定義を修正する

カラム名、データ名、カラム順、初期設定値、カラム制約等のような表の既存カラムの定義を修正することができます。あるカラムのデータ型を変更する場合、新規データ型が元のデータ型と互換していることを確認して下さい。それ以外の場合、データ不整合のため修正操作は失敗します。例えば、CHAR データ型のカラムを DATE データ型に変更することはできません。

➲ **例 1**

tb_staff 表のカラム **city** のカラム名を修正する ；

```
dmSQL> ALTER TABLE tb_staff MODIFY city NAME TO emp_photo;
```

➲ 例 2

tb_staff 表のカラム height のデータ型を変更する：

```
dmSQL> ALTER TABLE tb_staff MODIFY height TYPE TO decimal(10,2);
```

➲ 例 3

カラム height のカラム順を変更し、カラム HireDate の前に置く：

```
dmSQL> ALTER TABLE tb_staff MODIFY height BEFORE HireDate;
```

➲ 例 4

カラム nation の初期設定値を変更する：

```
dmSQL> ALTER TABLE tb_staff MODIFY nation DEFAULT TO 'Taiwan';
```

➲ 例 5

カラム height の制約を修正する：

```
dmSQL> ALTER TABLE tb_staff MODIFY height CONSTRAINT TO CHECK value < 250;
```

ロックモードを変更する

データベース接続において高い同時実行性を実現させる場合、ロックモードを行ロックのような低レベルに設定します。但し、低レベルのロックモードは、より多くのリソースを消費します。表のロックモードの選択は、常に同時実行性とリソースのトレードオフです。詳細は、9.4節の「ロック」を参照して下さい。

➲ 例

表 *tb_staff* のロックモードを行に変更する：

```
dmSQL> ALTER TABLE tb_staff SET LOCK MODE ROW;
```

表のフィルファクタを変更する

表作成時に定義したフィルファクタは、いつでも修正することもできます。フィルファクタ・オプションについての詳細は、6.2節の「表管理」の「フィルファクタは表の作成時もしくは後の変更時に決定されます。フィルファクタのオプションについての詳細は「表を作成する」の「フィルファクタ」を参照ください。

➲ 例

表のフィルファクタの値を変更する：

```
dmSQL> ALTER TABLE tb_staff SET FILLFACTOR 90;
```

NOCACHEオプションをON/OFFにする

表作成時に定義した NOCACHE オプションは、いつでも変更することができます。詳細については*「表管理」の「非キャッシュ」*のセクションを参照して下さい。

➲ 例

表 ***tb_staff*** の NOCACHE オプションを OFF にする：

```
dmSQL> ALTER TABLE tb_staff SET NOCACHE OFF;
```

表を別の表領域に移動する

ユーザーは表をほかの表領域に移動することができます。同時に、索引とこの表は同じ表領域に存在すると、索引も移動できます。異なる表領域にある場合、索引がほかの表領域に移動できません。だから、ユーザーはほかの表領域に索引を再作成する必要があります。

また、ユーザーは **FASTCOPY ON** の設定を通して、快速的に一つの表をその他の表領域に移動でき、これは表移動時の効率を高めます。そして、移動時に、システムはバッファを利用せず、一回のみのログ操作で、直接に一つのデータページをその他のデータページにコピーすることができます。そうすると、ログへの操作が大幅に下がります。

一つの表をほかの表領域に移動することによって、この表がほかのディスクに保存されることができ、ディスクフルによる保存できないことを避けます。

上記の移動について、下記の幾つかの制限があります。

- システム表、一時表、ビューがほかの表領域に移動できません。
- 永久表がSYSTABLESPACE 或いはTMPTABLESPACE に移動できません。
- TMPTABLESPACE に永久表の索引が再作成できません。
- NON-TMPTABLESPACEに一時表の索引が再作成できません。

- 別の表領域にシステム表の索引が再作成できません。
- データを一つの表から同じ表にコピーできません。
- 表を一つの表領域から同じ表領域に移動できません。

➲ **例**

```
dmSQL> CREATE TABLE tb_staff (c1 int, c2 char(10)) in ts_reg; // create table tb_staff
in ts_reg
dmSQL> CREATE INDEX idx_desc ON tb_staff (c1); // defaultly store index in ts_reg where
the table tb_staff is stored
dmSQL> SET FASTCOPY OFF;
dmSQL> ALTER TABLE tb_staff MOVE TABLESPACE ts_app; // slowly move table tb_staff and
index idx_desc to ts_app
dmSQL> SET FASTCOPY ON;
dmSQL> ALTER TABLE tb_staff MOVE TABLESPACE ts_app; // quickly move table tb_staff
and index idx_desc to ts_app
dmSQL> REBUILD INDEX idx_desc FOR tb_staff IN ts_shrink; // rebuild index idx_desc
in ts_shrink
dmSQL> ALTER TABLE tb_staff MOVE TABLESPACE ts_aut; // only move table tb_staff to
ts_aut, index idx_desc no change
```

JSONCOLSタイプの使用

DBMaster は JSONCOLS タイプをサポートします。JSONCOLS タイプは JSON 式で、表にあるあらゆるダイナミックカラムを構造化された出力タイプに転化します。JSONCOLS タイプは表にある全てのダイナミックカラムをストアすることに用いられ、ダイナミックカラムと合わせて使用することができます。ダイナミックカラムの詳細については、*ダイナミックの使用*というセクションをご参照ください。表にあるカラム数が多い且つ殆どのカラムが NULL 値である、そして個別にそれらを操作すると手間をかける場合、JSONCOLS タイプを使用する必要があります。

JSONCOLS タイプは、CREATE TABLE または ALTER TABLE ステートメントで定義されます。SQL 構文の詳細については、*SQL 文と関数参照編*の *SQL 文*をご参照ください。JSONCOLS タイプが定義された後、普通のカラムとして使用されます。また、JSONCOLS カラムは LONG VARBINARY から派生し、DBMaster でラージオブジェクトのカラムにインデックスを作成する

ことが出来ないため、定義された JSONCOLS カラムでインデックスを作成することが出来ません。

JSONCOLS タイプを定義するには、CREATE TABLE または ALTER TABLE ステートメントでのキーワード*<JSONCOLS_type_name>* JSONCOLS を使用する必要があります。

DBMaster では、JSONCOLS タイプは JSONCOLS カラムとして表示されます。下記のフォーマットで JSONCOLS タイプを定義します。

```
{col1_name:col1_value,col2_name:col2_value,col3_name:col3_value....}
```

JSONCOLS タイプの値の示例は以下のようになります。

```
{"ID":1234,"NAME":"linda","PHONE":"1234567"}
```

注　*JSONCOLSタイプのJSON式で、NULL値を含むダイナミックカラムが省略されます。*

データを挿入または更新する際、間違うフォーマットで JSONCOLS タイプを定義する場合、"エラー(8077) : [DBMaster]無効な JSON フォーマット"が戻されます。

JSONCOLS カラムの定義に DATE タイプ、TIME タイプまたは TIMESTAMP タイプの値を含む場合、相応するキーでクエリした結果として、当該値が Epoch 時間として表示されます。DBMaster では、Epoch 時間は協定世界時 (UTC) での1970 年1 月 1 日真夜中（午前 0 時 0 分 0 秒）の時刻からの形式的な経過秒数に定義されます。

DBMaster は内部方式で JSONCOLS カラムのデータを保存しますので、クエリ時に、ダイナミックカラムの表示順序はその挿入順序と異なる可能性があります。

JSONCOLS カラムまたは JSONCOLS カラムを含む表を削除する場合、JSONCOLS カラムに保存されたダイナミックカラムがシステムに自動的に削除されます。

JSONCOLS タイプを使用する際に、以下のガイドラインを考える必要があります。

- JSONCOLSタイプを変更することができません。JSONCOLSタイプを変更するには、JSONCOLSタイプを削除して、再作成しなければなりません。
- 各表には一つのみのJSONCOLSタイプを存在することを許可します。
- JSONCOLSタイプで制約またはデフォルト値を定義することができません。

JSONCOLS タイプのセキュリティモデルは、普通のカラムと同様です。JSONCOLS カラムで SELECT、INSERT、UPDATE 及び DELETE ステートメントを実行するには、JSONCOLS カラムに相応する権限を有する必要があります。

JSONCOLS カラムのデータ操作は、単独的なダイナミックカラムの名称を使用、または JSONCOLS タイプの名称を参照、及び JSONCOLS タイプの JSON 式の使用による JSONCOLS タイプの値を指定することによって実行されます。ダイナミックカラムは任意の順序で JSONCOLS カラムに表示できます。

➲ **例**

下記のコマンドで、JSONCOLS タイプを含む表を作成します。

```
dmSQL> CREATE TABLE student(name CHAR(30), info JSONCOLS);
```

或いは、

```
dmSQL> CREATE TABLE student(name CHAR(30));
dmSQL> ALTER TABLE student ADD COLUMN info JSONCOLS;
```

JSONCOLS タイプの名称を使用して、表 student にデータを挿入します。

```
dmSQL> INSERT INTO student (name,info) VALUES
('jessia','{"desk_id":3,"birthday":"1986-09-19","score":90}');
1 rows inserted
dmSQL> INSERT INTO student (name,info) VALUES
('pine','{"desk_id":4,"birthday":"1987-03-03","score":95}');
1 rows inserted
```

"SELECT *"を使用して当該表をクエリします。

```
dmSQL> SET blobwidth 80;
dmSQL> SELECT * FROM student;
       NAME                           INFO
```

```
================ ============================================
jessia           {"score":90,"birthday":"1986-09-19","desk id":3}
pine             {"score":95,"birthday":"1987-03-03","desk id":4}
2 rows selected
```

JSONCOLS タイプの名称を使用して、当該表をクエリします。

```
dmSQL> SELECT name, info FROM student;
       NAME                          INFO
================ ============================================
jessia           {"score":90,"birthday":"1986-09-19","desk_id":3}
pine             {"score":95,"birthday":"1987-03-03","desk_id":4}
2 rows selected
```

JSONCOLS タイプの名称を使用して、当該表のデータを更新します。

```
dmSQL> UPDATE student SET info = '{"desk id":7, "birthday":"1986-09-19","score":88}'
WHERE name='jessia';
1 rows updated
```

birthday というカラムのデータのタイプを DATE タイプに修正します。

```
dmSQL> ALTER TABLE student ADD DYNAMIC COLUMN birthday DATE;
dmSQL> SELECT info FROM student;
                              INFO
================================================================
{"score":88,"birthday":"1986-09-19","desk_id":7}{"score":95,"birthday":"1987-03-0
3","desk_id":4}2 rows selected
dmSQL> INSERT INTO student (name,desk_id,birthday,score) VALUES
('mike','8','1985-02-15','92');
dmSQL> SELECT info FROM student;
                              INFO
================================================================
{"score":88,"birthday":"1986-09-19","desk_id":7}
{"score":95,"birthday":"1987-03-03","desk_id":4}
{"BIRTHDAY":477244800000,"DESK_ID":"8","SCORE":"92"}
3 rows selected
```

student という JSONCOLS カラムでテキストインデックスを作成します。

```
dmSQL> CREATE TEXT INDEX idx_stu ON student(INFO);
```

student という JSONCOLS カラムでビューを作成します。

```
dmSQL> CREATE VIEW view1 AS SELECT info FROM student;
```

```
dmSQL> SELECT * FROM view1;
                                INFO
=====================================================================
{"score":88,"birthday":"1986-09-19","desk_id":7}
{"score":95,"birthday":"1987-03-03","desk_id":4}
{"BIRTHDAY":477244800000,"DESK_ID":"8","SCORE":"92"}
3 rows selected
```

ダイナミックカラムの使用

DBMaster はダイナミックカラムをサポートします。ダイナミックカラムは表の定義に存在しませんが、JSON 文字列から派生されたキーで、そして表はその一つのカラムを JSONCOLS カラムとして宣言する場合にのみ使用されます。ダイナミックカラムは半構造化データ、数千の属性及びそのデータタイプを頻繁に変更するデータを保存することに用いられ、JSONCLOS タイプと合わせて使用することができます。JSONCLOS タイプの詳細については、*JSONCLOS タイプの使用*というセクションをご参照ください。表にあるカラム数が多い且つ殆どのカラムが NULL 値である場合、ダイナミックカラムを使用する必要があります。

ダイナミックカラムは JSONCOLS カラムにストアされ、ダイナミックカラムの記述情報は SYSDESCOL にストアされます。SYSDESCOL に関する詳細については、*DBMaster のシステムカタログ表*を、JSONCOLS カラムの詳細については、JSONCOLS タイプの使用というセクションをご参照ください。

ダイナミックカラムのセキュリティモデルは普通のカラムと大体同じで、ただし以下の特徴があります。

- 一つのダイナミックカラムで一つのインデックスが作成されます。
- ダイナミックカラムはデータタイプの修正のみをサポートします。
- ダイナミックカラムは以下のデータタイプをサポートします：SMALLINT、INT、FLOAT、DOUBLE、DATE、TIME、TIMESTAMP、CHAR、VARCHAR、NCHAR、NVARCHAR。
- ダイナミックカラムはNULL値にされることが可能です。
- ダイナミックカラムがディフォルト値を持ってはいけません。

- ダイナミックカラムがカラム制約を持ってはいけません。
- ダイナミックカラムはストアコマンドに使用されてはいけません。
- ダイナミックカラムはストアドプロシージャに使用されることができません。
- ダイナミックカラムはトリガーに使用されることができません。

JSONCOLS カラムは作成された後で、ダイナミックカラムは定義される必要が無し、直接に使用できます。ダイナミックカラムのデフォルトのデータタイプは varchar（256）で、ALTER TABLE ADD DYNAMIC COLUMN コマンドを使用して、このデフォルトのデータタイプをその他のデータタイプに変更することができますが、ダイナミックカラムを表に挿入する場合、当該コマンドを使用して、このダイナミックカラムのデータタイプを宣言することができます。このコマンドで宣言したデータタイプをその他のデータタイプに変更しようとする場合、ALTER TABLE MODIFY DYNAMIC COLUMN コマンドを使用する必要があります。また、ダイナミックカラムの記述情報を削除しようとする場合、ALTER TABLE DROP DYNAMIC COLUMN コマンドを使用する必要があります。SQL 構文の詳細については、*「SQL 文と関数参照編」*をご参照ください。ALTER TABLE ADD DYNAMIC COLUMN を実行せず、初めてデータを挿入して、当該コマンドでダイナミックカラムのデータタイプを宣言して、それからこの前挿入されたデータを宣言されたデータタイプに変換できない場合、クエリ時に当該データは NULL に表示され、エラーも戻されません。

また、パラメータでダイナミックカラムにデータを挿入するまたはダイナミックカラムのデータを更新する場合、DBMaster は VARCHAR タイプでデータの挿入のみをサポートします。この場合、暗黙的なデータ転換機能を使用して、その他のデータタイプでデータを挿入することができます。

ダイナミックカラムのデータ操作は、単独的なダイナミックカラムの名称を使用、または JSONCOLS タイプの名称を参照、及び JSONCOLS タイプの JSON 式の使用による JSONCOLS タイプの値を指定することによって実行されます。ダイナミックカラムは任意の順序で JSONCOLS カラムに表示できます。

➲ 例

下記の操作は表 **student** に基づきもので、詳しい情報については、*JSONCLOS タイプの使用*の例をご参照ください。

ダイナミックカラムの名称を使用して、表 **student** にデータを挿入します。

```
/* 暗黙的なデータ転換はデフォルトとして OFF になっています */
dmSQL> INSERT INTO student (name,score) VALUES(?,?);
dmSQL/Val> 'demi','85';      /* これは ok です */
1 rows inserted
dmSQL/Val> 'finly',82;      /* INT タイプは CHAR タイプに転換できません */
ERROR (9629): 値リスト・シンタックス・エラー
dmSQL/Val> END;
dmSQL> SET itcmd ON;
dmSQL> INSERT INTO student (name,score) VALUES(?,?);
dmSQL/Val> 'finly',82;      /* 暗黙的なデータ転換機能を使用します */
1 rows inserted
dmSQL/Val> END;
dmSQL> SET itcmd OFF;
dmSQL> INSERT INTO student (name,desk id,birthday,score)
VALUES('linda','1','1982-01-01','91');
1 rows inserted
dmSQL> INSERT INTO student (name,desk id,birthday,score)
VALUES('glow','2','1984-03-25','93');
1 rows inserted
dmSQL> INSERT INTO student (name,desk id,birthday,score)
VALUES('kitty','abc','1980-02-27','97');
1 rows inserted
```

"SELECT *"を使用して表 student をクエリします。

```
dmSQL> SELECT * FROM student;
     NAME                          INFO
============= ============================================
jessia         {"score":88,"birthday":"1986-09-19","desk_id":7}
pine           {"score":95,"birthday":"1987-03-03","desk_id":4}
mike           {"BIRTHDAY":477244800000,"DESK_ID":"8","SCORE":"92"}
demi           {"SCORE":"85"}
finly          {"SCORE":"82"}
linda          {"BIRTHDAY":378662400000,"DESK_ID":"1","SCORE":"91"}
```

```
glow          {"BIRTHDAY":448992000000,"DESK ID":"2","SCORE":"93"}
kitty         {"BIRTHDAY":320428800000,"DESK ID":"abc","SCORE":"97"}
8 rows selected
```

ダイナミックカラムの名称を使用して、表 **student** をクエリします。

```
dmSQL> SELECT name, desk_id, birthday, score FROM student;
     NAME         DESK_ID      BIRTHDAY              SCORE
==============  ==========  ============  ====================
jessia          7           19*           88
pine            4           19*           95
mike            8           19*           92
demi                NULL    NU*           85
finly               NULL    NU*           82
linda           1           19*           91
glow            2           19*           93
kitty          abc          19*           97
8 rows selected
```

ダイナミックカラムの名称を使用して、表 **student** のデータを更新または削除します。

```
dmSQL> UPDATE student SET score='88' WHERE name='linda';
1 rows updated
dmSQL> DELETE FROM student WHERE desk_id='2';
1 rows deleted
```

当該表にダイナミックカラムの説明を追加します。

```
dmSQL> ALTER TABLE student ADD DYNAMIC COLUMN desk id INT;
dmSQL> ALTER TABLE student ADD DYNAMIC COLUMN score DOUBLE;
```

データを表 **student** に挿入します。

```
dmSQL> INSTO student (name, desk id, age, score)
VALUES('jane','12','1982-05-07',96);
ERROR (6150): [DBMaster] INSTER/UPDATE 値のタイプとカラムのデータ型が矛盾しているか、比較オペランドの値と記述したカラムのデータ型が矛盾しています
dmSQL> INSERT INTO student (name, desk_id, age, score)
VALUES('jim',8,'1984-09-26',98);
1 rows inserted
dmSQL> SELECT name, desk_id, birthday, score FROM student;
     NAME         DESK_ID       BIRTHDAY            SCORE
==============  ============  ============  ====================
```

```
jessia                    7  1986-09-19    8.80000000000000e+001
pine                      4  1987-03-03    9.50000000000000e+001
mike                      8  1985-02-15    9.20000000000000e+001
demi               NULL         NULL       8.50000000000000e+001
finly              NULL         NULL       8.20000000000000e+001
linda                     1  1982-01-01    8.80000000000000e+001
kitty              NULL       1980-02-27    9.70000000000000e+001
jim                       8     NULL       9.80000000000000e+001
8 rows selected
```

score というダイナミックカラムのデータタイプを修正します。

```
dmSQL> ALTER TABLE student MODIFY DYNAMIC COLUMN score TYPE TO INT;
```

desk_id というダイナミックカラムでインデックスを作成します。

```
dmSQL> CREATE INDEX idx1 ON student(desk_id);
```

birthday というダイナミックカラムの記述情報を削除します。

```
dmSQL> ALTER TABLE student DROP DYNAMIC COLUMN birthday;
```

表をロックする

DBMaster には、データベースにアクセスするときに自動的にロックをかけるメカニズムがありますが、後から使用する SELECT 文や UPDATE 文のために、手動で表をロックすることもできます。通常、検索または更新している表をロックし、他の利用者に表を更新させないようにします。

表をロックするときは、幾つかのオプションを指定することができます。データを検索するときの共有ロック、更新するときの排他ロック、ロックを取得するときの WAIT または NO WAIT オプションがあります。これらの機能の詳細については、*「SQL 文と関数参照編」*をご覧下さい。表ロック、同時実行制御、トランザクション操作についての詳細は、9章の「同時実行制御」を参照して下さい。

➲ 例

後続の検索から表 **tb_staff** をロックします。直ちにロックできない場合は、ロックするのを待たずに失敗します。

```
dmSQL> LOCK TABLE tb_staff IN SHARE MODE NO WAIT;
```

表を削除する

表が不要になったときは、削除することができます。表を削除すると、全てのデータと表の索引も削除されます。表に割り当てられていた全てのページは開放されます。

➲ **例 1**

DROP TABLE コマンドを使用して、表 **tb_staff** を削除する：

```
dmSQL> DROP TABLE  tb_staff;
```

➲ **例 2**

DROP TABLE IF EXISTS マンドを使用して、表 **tb_staff** を削除する：

```
dmSQL> DROP TABLE IF EXISTS tb_staff;
```

6.3 ビュー管理

DBMaster では、ビューと呼ばれる仮想表を定義することができます。ビューは、既存の表を基にして定義され、ビュー名を付けてデータベースに保存されます。ビューの定義はデータベースに保存されますが、ビューで実際に見るデータが物理的に何処かに格納されているわけではありません。データはビューの基になる表に格納されており、表からビューの行が引き出されます。ビューは、1 つ以上の表（または他のビュー）を参照する問い合せによって定義します。

ビューは、データベースの非常に役に立つメカニズムです。例えば、複雑な問合せを一度ビューで定義しておけば、繰り返し使用することができ、問合せの手間を省くことができます。更に、ビューを使用してアクセスできる表の行やカラムを事前に決めておき、データベースのセキュリティを高めることもできます。

ビューが 1 つの表からの問合せで作成されている場合、そのビューを使用して、元の表の更新、挿入、削除することができます。この制限により、ビューが単一テーブルから提供されない場合、ビューが可能となるのは検索されたときのみです。

ビューを作成する

ビューは dmSQL または JDBA ツールで作成されます。ビューを作成する際に、ビューの名称、関連する表またはビューのクエリ指令を指定する必要があります。

ビューのカラム名リストを指定することができます。カラム名を指定しないときは、ビューは基礎とする表のカラム名が継承されます。

CREATE VIEW 構文を使用します。例えば、表 **tb_staff** の 2 カラムのみを利用者に見せるには、次の SQL 文でビュー**vi_staff** を作成することができます。利用者は、ビュー**vi_staff** を通して表 **tb_staff** の 2 つのカラム **name** と **ID** のみを見ることができます。

➲ **例**

CREATE VIEW で、表 **tb_staff** からビュー**vi_staff** を作成する：

```
dmSQL> CREATE VIEW vi_staff (empName, empId) AS
             SELECT name, ID FROM tb_staff;
```

CREATE OR REPLACE VIEW 構文を使用します。例えば、ビュー**vi_staff** が存在しました、表 ***tb_staff*** の 2 カラムのみ利用者に見せたかったが、3 カラムを見るためビューの定義を変更する必要があります。ビューの権限ではありません。以下のはコマンドでビューを代わります。ユーザーは **vi_staff** を通じて **tb_staff** から 3 カラムがビューできます。

➲ **例**

```
dmSQL> CREATE OR REPLACE VIEW vi staff (empName, empId, empAge) AS
             SELECT name, ID, age FROM tb_staff;
```

ビューのスキーマを確認する

dmSQL や JDBA Tool を使って、ビューのスキーマを問い合せることができます。JDBA Tool には、表スキーマをグラフィカルに表示し、SQL 文を入力する事無く、表スキーマを修正することができます。また、dmSQL コマンドの DEF VIEW を使って、ビューのスキーマを直接問い合せることもできます。

➲ 例

ビュー**vi_staff** のスキーマを確認する：

```
dmSQL> DEF VIEW vi_staff;
create view SYSADM.VI_STAFF(empname,empid) as select name,id from SYSADM.TB_STAFF ;
```

ビューを削除する

不要になったビューを削除することができます。ビューを削除すると、システムカタログに格納されている定義のみが削除されます。ビューの基になる表には、何の影響も与えません。

➲ 例

ビュー**vi_staff** を削除する：

```
dmSQL> DROP VIEW vi_staff;
```

6.4 シノニム管理

シノニムは、表またはビューの別名のことです。シノニムは単なる別名なので、システムカタログ内のシノニム定義以外のストレージを必要としません。

シノニムは、表あるいはビューの完全修飾名を簡便にするのに便利な機能です。表およびビューを識別するために、通常所有者名とオブジェクト名からなる完全修飾名が使用されます。シノニムを使用することによって、表またはビューの完全修飾名を指定しなくても、誰でも表またはビューにアクセスすることができます。シノニムには所有者名が無いので、全てのシノニムは、データベース全体で一意にしなければなりません。JDBA Tool、または dmSQL を使ってシノニムを作成/削除することができます。

シノニムを作成する

➲ 例 1

CREATE SYNONYM コマンドを使用します：

```
dmSQL> CREATE SYNONYM staff FOR SYSADM.tb_staff;
```

この文は､所有者が SYSADM である表 *tb_staff* の別名 *tb_staff* を作成します。全てのデータベース利用者は、シノニム staff を通して、表 SYSADM.tb_staff を直接参照することができます。

➲ 例 2

CREATE OR REPLACE コマンドを使用します：

```
dmSQL> CREATE OR REPLACE SYNONYM staff FOR SYSADM.tb_staff;
```

表 SYSADM.tb_staff の別名 staff が存在すると、staff を替わります。staff が存在しない場合、作成します。

シノニムを削除する

不要になったシノニムを削除することができます。シノニムを削除すると、シノニムの定義のみがシステムカタログから削除されます。

➲ 例

シノニム *tb_staff* を削除する：

```
dmSQL> DROP SYNONYM tb_staff;
```

6.5 索引管理

索引は行の高速ランダムアクセスを可能にします。表に索引を構築すると検索速度が向上します。

SELECT NAME FROM tb_staff WHERE id=306004 というクエリを実行したとき、ID カラムに索引が作成されていれば検索に掛かる時間を大幅に短縮することが可能です。

索引は一つ以上、最大 32 カラムから構成することができます。表の全てのカラムに定義できます。

索引は、一意か非一意のどちらかです。一意索引は、二つ以上の行が同じキー値をもつことを認めません。ただし、NULL をキー値に持つ行はいくつあってもかまいません。空でない表に一意索引を作成すると、DBMaster は全ての既存のキーが異なっているかどうかチェックします。もし重複する

キーがあれば、エラーメッセージを返します。一意索引が作成されている表に行を挿入すると、挿入行と同じキーをもつ行が無いことを確かめます。

索引を作成するときに、各索引カラムのソート順を昇順か降順かで指定することができます。例えば、表に 1、3、9、2、6 の５個のキー値があるとします。昇順索引キーの順序は 1、2、3、6、9 降順索引キーの順序は 9、6、3、2、1 になります。

索引の順序は、問合せデータの出力順に大きく影響します。

➲ 例

下記のクエリが実行される場合、

```
dmSQL>select name, age from friend_table where age > 20;
```

カラム ***age*** の降順索引を使用して、出力は以下のようになります：

```
      name                    age
----------------          ----------------
Jeff                              49
Kevin                             40
Jerry                             38
Hughes                            30
Cathy                             22
```

索引を作成するときには、表と同様、フィルファクタを指定することができます。フィルファクタは、索引ページにどのくらいの密度までキーを置くか表します。フィルファクタは 50%～100%の範囲で指定し、初期値は 100%です。索引を作成した後に頻繁にデータを更新する場合は粗いフィルファクタ、例えば 60%を設定します。表のデータを更新することがない場合は、フィルファクタを初期値 100%のままにします。

索引を作成する前に、全てのデータを表にロードしておくべきです。大量データの場合は、必ずロードしておきます。表にデータをロードする前に索引を作成すると、新しい行をロードする度に索引が更新されます。すぐ分かるように、大量データをロードした後に索引を作成した方が、ロード前に索引を作成するよりもはるかに効率的です。

索引を作成する

JDBA Tool や dmSQL の CREATE INDEX 文を使って、索引を作成することができます。索引名と索引カラムを指定して、表に索引を作成します。各カラムのソート順を昇順か降順かで指定することができます。ソート順の初期値は昇順です。

➲ **例 1**

表 **tb_staff** のカラム ***ID*** に降順索引 **idx_desc** を作成する：

```
dmSQL> CREATE INDEX idx_desc ON tb_staff (ID DESC);
```

➲ **例 2**

表 **tb_staff** のカラム ***ID*** に一意索引 **idx_uniq** を作成する：

```
dmSQL> CREATE UNIQUE INDEX idx2 ON tb_staff (ID);
```

➲ **例 3**

フィルファクタを指定して、索引を作成する：

```
dmSQL> CREATE INDEX idx_fill ON tb_staff (name, age DESC) FILLFACTOR 60;
```

➲ **例 4**

表領域 **ts_reg** に索引を作成する：

```
dmSQL> CREATE INDEX idx_reg ON tb_staff (name, age DESC) IN ts_reg FILLFACTOR 60;
```

式インデックスの作成

インデックスはシンプルなカラムだけでなく、式カラムやユーザー定義関数（UDF）カラム上でも作成できます。

➲ **例 1**

テーブル tb_salary の式 basepay+bonus 上でインデックス idx_expr を作成する方法：

```
dmSQL> CREATE INDEX idx_expr ON tb_salary (basepay+bonus);
```

➲ 例 2

テーブル **tb_staff** の UDF 部分文字列 **(nation,1,3)** 上でインデックス **idx_substr** を作成する方法：

```
dmSQL> CREATE INDEX idx_substr ON tb_staff (substring(nation,1,3) desc);
```

➲ 例 3

テーブル **tb_salary** の式および UDF **abs(bonus)** 上でインデックス **idx_udf** を作成する方法：

```
dmSQL> CREATE INDEX idx_udf ON tb_staff (basepay+abs(bonus)-tax desc);
```

XMLカラムに索引の作成

XML クエリ性能を向上させるため、XML カラムに対して特別な XML 索引を作成することができます。XML は XML UDF: **extract()** と **extractvalue()**をサポートします。以下の例は dmSQL で XML カラムに索引をどう作成するか説明します。詳細な構文と SQL コマンド CREATE INDEX 用法は *SQL コマンドと関数参考* にあります。

➲ 例 1

extract XML UDF を使って索引を作成する：

```
dmSQL> create index idx_extr on tb_extract (extract(id, '/order/items/item/@product',
NULL));
```

➲ 例 2

extractValue XML UDF を使って索引を作成する：

```
dmSQL> create index idx extrV on tb extract (extractValue(id,
'/order/items/item/@product', NULL));
```

extract()と extractvalue()の主な区別:

extract()

- 結果は複数の値あるいはユニーク値或いは 0 を使用できます
- asc / descは使用できません
- 一意索引は使用できません

extractValue()

- UDFの結果にユニーク値或いは 0 を使用できます

（UDF の結果が複数であれば、既存のタプルに索引の作成が失敗、タプルにデータの挿入もも失敗します）

- asc / descを使用できます
- 一意索引使用可能

索引を削除する

JDBA Tool や dmSQL の DROP INDEX 文を使って、索引を削除することができます。索引が主キーで他の表から参照されている場合は、索引を削除することはできません。主キーについての詳細は、「データ整合性管理」の節を参照して下さい。

➲ 例

表 **tb_staff** から索引 **idx_desc** を削除します：

```
dmSQL> DROP INDEX idx_desc FROM tb_staff;
```

索引を再作成する

JDBA Tool または dmSQL の REBUILD INDEX 文を使って、索引を再作成することができます。一般的に、索引が断片化されていて、その効率が低下した場合に索引を再作成します。索引の再作成は、古い索引を削除し、新しい索引を作成します。

ユーザーは表をほかの表領域に移動することができます。同時に、索引とこの表は同じ表領域に存在すると、索引も移動できます。異なる表領域にある場合、索引がほかの表領域に移動できません。だから、ユーザーはほかの表領域に索引を再作成する必要があります。

➲ 例 1

tb_staff 表の **idx_fill** 索引を再作成する：

```
dmSQL> REBUILD INDEX idx_fill FOR tb_staff;
```

➲ 例 2

```
dmSQL> REBUILD INDEX idx FOR tb_staff IN ts_reg;
```

6.6 自動インデックスの管理

クラウドデータベースの発展に従って、インデックスの管理は手動管理から自動管理に変更しました。ユーザーに実行されたクエリステートメントによって、自動インデックスデーモンが当該リクエストを分析することができ、もっとインテリジェントにインデックスを管理することになります。

DBMaster は自動インデックスをサポートします。これは非一意インデックスと類似していますが、当該インデックスは自動インデックスデーモンによって自動的に作成及び削除されることができます。AUTOCOMMIT ON を設定する場合、自動インデックスを作成する際、DBMaster は Update(U) ロックのみをリクエストします。これは DBMaster が他のユーザーが同時に表をクエリすることを許可しているのを意味します。

DBMaster は自動インデックスデーモンが自動インデックスを操作することをサポートします。当該動作は**コレクションメカニズム**及び**ハンドリングメカニズム**によって実現します。自動インデックスデーモンが起動後に、コレクションメカニズムがユーザーに実行された各クエリステートメントの実行計画を分析して、その結果（ゼロまたは複数のログ）を DMSCAN.LOG ファイルに記録します。ハンドリングメカニズムが **DB_IDXTM** 及び **DB_IDXTV** の設定又は dmSQL コマンド SYNC AUTO INDEX コマンドの実行又は JDBA ツールの同期自動インデックスウィザードの利用によって、ハンドリングを行います。ハンドリングメカニズムの主なジョブは下記のようになります : DMSCAN.LOG ファイルの読み取り、あらゆるログを分析して、その結果によって自動インデックスを作成する必要があるかを決めること、インデックス使用情報の更新統計及び **DB_IDXDP**（自動インデックスのみ削除され、その他のインデックスが削除されていない）に設定された時間期限を越えてまだ使用されていない自動インデックスを削除すること等。このプロセスの中に作成及び削除したインデックスのレコードが DMAUTOIDX.LOG ファイルに保存されていますが、自動インデックスの状況を確認することができます。

実際は、クライアントサイドのユーザーがクエリステートメントを実行する際、直ちにログを DMSCAN.LOG ファイルに書き込まなく、まずはログをクライアントサイドのバッファ(固定サイズは2560バイト)に保存して、バッファフルになっても DMSCAN.LOG ファイルに書き込みます。また、当該ユーザーはデータベースへの接続が切断する又は SYNC AUTO INDEX コマンドを実行する場合、バッファでのデータも DMSCAN.LOG ファイルに書き込まれます。そうすると、マルチユーザーが同時にデータを DMSCAN.LOG ファイルに書き込むことによる内容混雑を避けるだけでなく、I\O を集中し性能アップの方面にも役立ちます。

自動的に自動インデックスを作成及び削除するために、**dmconfig.ini** ファイルにてキーワード **DB_IDXSV**, **DB_IDXLG**, **DB_IDXTM**, **DB_IDXTV**, **DB_IDXDP** 及び **DB_IDXLN** を設定して、自動インデックスデーモンを制御する必要があります。データベースの運用に従って、DBA 又は SYSDBA 又は SYSADM オーソリティーを持つユーザーのみ、**setSystemOption()**をコールして、**IDXLG** 以外のオプションを設定することができます。

自動インデックスとその他のインデックスとの違いは、以下ようになります。

- 自動インデックスが自動インデックスデーモンによって自動的に作成されます。
- インデックスが他のインデックスと合併できる場合、或いは当該インデックスが設定の削除日数を越えてまだ使用されていない場合、自動インデックスが自動インデックスデーモンによって自動的に削除されます。
- ユーザーによって作成されたインデックスの最大列数は32ですが、自動インデックスデーモンによって作成された自動インデックスの最大列数は16です。
- COMMIT ONを設定し自動インデックスを作成するにはUロックが必要条件となりますが、他のインデックスを作成する場合、Xロックが必要となります。
- DBMasterは"SET LOADAUTOINDEX ON|OFF"構文及びODBCの機能SQLSetConnectOptionをサポートしていますが、ユーザーはこれにより、自動インデックスをロードするかどうかを決めることができます。

ODBC機能の詳細については、*「ODBCプログラマーガイド」*をご参照ください。

➲ **例 1**

システムストアドプロシージャ **SetSystemOption** で、自動インデックスサーバをオープンに設定する。

```
dmSQL> call setsystemoption('IDXSV','1');  //オートインデックスサーバーを起動させます
dmSQL> select * from tb_staff where joinDate="1986-07-20";   //デーモンで自動インデックスを作成します
dmSQL> sync auto index;    //自動インデックスデーモンをウェイクアップします
dmSQL> select * from sysindex;
dmSQL> select * from sysindexref;
```

➲ **例 2**

システムストアドプロシージャ **SetSystemOption** で日数を 60 にリセットして、自動インデックスを削除する。

```
dmSQL> call SETSYSTEMOPTION('IDXDP','60');
```

自動インデックスの作成

自動インデックスが自動インデックスデーモンにより自動的に作成され、或いは dmSQL コマンド CREATE AUTO INDEX により手動で作成されます。

手動でインデックスを作成する詳細な情報について、*「SQL コマンドと関数参照編」*の CREATE INDEX 構文をご参照ください。

自動インデックスデーモンがテーブルに自動インデックスを作成する際、自動インデックスのタイプ及び自動インデックスのカラム ID を指定し、昇順または降順などの各カラムのソート順を指定します。デフォルトのソート順は昇順です。

自動インデックスデーモンによって作成された自動インデックスの名称は AUTO、アンダーライン、カラム id 及びカラムの順序（カラムの順序が *D* 又は *null* でありますが、*D*は昇順を表しますが、*null* は降順を表します。）に構成されています。同じ名称を持つ二つのインデックスが合併できる場合は、新しいインデックスを作成する必要がありませんが、合併出来ない場合は、新しいインデックスを作成して、index_name_Rxxx に名付けます。

index_name は二つのインデックスの名称で、**xxx** はランダムな数字で、特別な要求がないです。

➲ **例 1**

カラム ID 及び NAME において、表 **tb_staff** に対して降順で自動インデックス **AUTO_1D_2** を作成します。dmsql コマンドにより、DESC オプションを使用することができます。

```
dmSQL> CREATE AUTO INDEX AUTO_ID_2 ON tb_staff (ID DESC, NAME);
```

➲ **例 2**

新しいインデックスの名称は古いインデックスの名称とほぼ同じ、そして合併できる場合、新しいインデックスの名称を **AUTO_ID_2__R321** に拡張することができます。ここの **321** はランダムな数字です。

```
dmSQL> CREATE AUTO INDEX AUTO_ID_2__R321 ON tb_staff (ID DESC, NAME);
```

自動インデックスの削除

自動インデックスデーモンが DB_IDXDP によって設定した削除日数を越えてまだ使用されていない自動インデックスを自動的に削除することができ、そして、dmSQL コマンド DROP AUTO INDEX を使用して、自動インデックスを削除することもできます。インデックスが主キーとするカラムにて作成される場合、或いは他の表に参照される場合、当該インデックスが削除できません。主キーに関する詳細については、*「データ整合性管理」*の節をご参照ください。

➲ **例**

表 **tb_staff** からインデックス AUTO_1D_2 を削除する。

```
dmSQL> DROP AUTO INDEX AUTO_1D_2 FROM tb_staff;
```

6.7 テキスト索引を管理する

テキスト索引は、カラムに複数の言葉やフレーズを含む表の行に高速アクセスを可能にする機能です。テキスト索引は、カラムに含まれるテキストの全てを含みますが、データはコード変換され、表から直接取得するより

も早く回収できるよう構築されています。一旦表にテキスト索引を作成すると、その操作はユーザーにとって透過的になります。DBMS は可能な限り索引を使って、全テキスト問合せのパフォーマンスを向上させます。

DBMaster は 2 つのテキスト索引方法を提供します：シグネ―チャおよびインバーテッド・ファイル(IVF)です。テキスト索引は、CHAR、VARCHAR、LONG VARCHAR、LONG VARBINARY および FILE データ・タイプを含むすべての文字タイプ・カラムにビルドすることができます。表は多くのテキスト索引を持つことができ、多数のカラムを使用して、テキスト索引を構築することができます。ユーザは JDBA ツールあるいは dmSQL コマンド CREATE [SIGNATURE | IVF] TEXT INDEX のいずれかの使用によりテキスト索引を作成することができます。

➲ **例**

データ・カラムで自動的にテキスト索引を使用する：

```
dmSQL> SELECT id FROM tb_staff WHERE data MATCH 'compute';
```

DBMaster の文字列演算には、MATCH、CONTAIN、LIKE がありますが、テキスト索引検索には、MATCH 演算子のみ適用できます。

DBMaster は 2 つの異なるタイプのテキスト索引を提供します：シグネ―チャおよびインバーテッド・ファイルです。シグネ―チャ・テキスト索引は少量のデータにより能率的です。インバーテッド・ファイルのテキスト索引は通常より多くのストレージ空間が必要ですが、大量のデータに対するクエリーのレスポンスがより速くなります。

シグネチャ・テキスト索引を作成する

テキスト索引方法がコマンドの中で指定されない場合、DBMaster はシグネチャ・テキスト索引を作成します。JDBA Tool や dmSQL の CREATE TEXT INDEX 文または CREATE SIGNATURE TEXT INDEX 文を使って、シグネチャ・テキスト索引を作成することができます。

➲ **例**

tb_staff 表のデータ・カラムにシグネチャ・テキスト索引 **tidx_name** を作成する：

```
dmSQL> CREATE TEXT INDEX tidx_name ON tb_staff(data);
```

シグネチャ・テキスト索引パラメータ

DBMaster は、シグネチャ・テキスト索引のパフォーマンスおよびストレージ・サイズを設定するのに便利なよう 2 つのパラメーターを供給します。

テキスト・サイズの合計 (MB) —すべてのソースドキュメントの推定サイズの合計。範囲は 1～200 メガバイト(MB)。初期設定値は 32 です。実際のテキスト・サイズの合計が 200 MB までと制限されていないことに注意してください。推定サイズが 200 より大きい場合は、200 に設定してください。しかしながら大量のデータに対してよりよいクエリーパフォーマンスを得るためには IVF テキスト索引を使用することを強く推奨します。

スケール—予測されるインデックス・サイズと合計のテキスト・サイズの比率。20(MB)にテキスト・サイズの合計をセットし、テキスト・インデックスが 10MB のストレージを使用することが想定される場合、スケールを 50(50%)にを設定することが推奨されます。。スケールが大きいほどより高い検索パフォーマンスを得られます。範囲は 10 の～200 です。また、デフォルト値は 40(40%)です。

➲ 例

テキスト索引を作成する対象は、データを約 40 メガバイト含んでいる **tb_staff** 表の **data** カラム上の **tidx_scale** です。私たちはテキスト索引がストレージ空間を約 20 メガバイト使用することを望みます：

```
dmSQL> CREATE SIGNATURE TEXT INDEX tidx_scale ON tb_staff(data)
       TOTAL TEXT SIZE 40 MB
       SCALE 50;
```

テキスト索引パラメータとして初期設定値を使用することができます。テキスト索引のパフォーマンスを上げるか、テキスト索引のサイズを縮小する場合、テキスト索引パラメータを変更して下さい。パラメータをセットして、テキスト索引のパフォーマンスを監視して下さい。それからパラメータを再度調節して下さい。

IVFテキスト索引を作成する

ユーザは、CREATE IVF TEXT INDEX コマンドの使用によりインバーテッドファイル・(IVF)テキスト索引を作成することができます。

➲ 例

tb_staff 表の data カラム上で IVF テキスト索引(ivfidx_name)を作成する:

```
dmSQL> CREATE IVF TEXT INDEX ivfidx_name ON tb_staff(data);
```

IVFテキスト索引パラメータ

IVF テキスト索引を作成するコマンドには 2 つのパラメータを使用します。

ストレージパス —インバーテッドファイルが存在する場所で、論理的な作業ディレクトリ。**dmconfig.ini** ファイルに論理的なディレクトリーを定義します。初期設定値は **DB_DbDir**、データ・ベースのホームディレクトリとなります。インバーテッドファイル索引の詳細ストレージ管理および指定する協定は、次のセクションに記述されています。

テキスト・サイズの合計(MB) —ドキュメントのサイズの合計の近似に索引が付けられるでしょう。サイズのユニットはメガバイト(MB)です。サイズに基づいて、DBMaster は、どれだけの分割がなされるか決定するでしょう。それは 1MB から 10000MB の間に及ぶかもしれません。デフォルト価値は 500MB です。

➲ 例

データを約 400 MB 含んでいる **tb_staff** 表の data カラム上のパス¥IVFDIR の中でインバーテッドファイル・テキスト索引 **ivfidx_name** を作成する:

最初に、データベースの **dmconfig.ini** セクション中の論理的なパスを加えてください。

```
MYPATH1 = \IVFDIR
```

下記コマンドを使用してください。

```
dmSQL> CREATE IVF TEXT INDEX ivfidx_name ON tb_staff(data)
    2> STORAGE PATH MYPATH1
    3> TOTAL TEXT SIZE 400 MB;
```

インバーテッドファイルのテキスト索引を作成している間、それは大量のメモリーリソースを必要とします。DBMaster は、テキスト索引の作成時の最大メモリ使用量を決定するため単純な規則に従います。DBMaster が自由なメモリを検知することができないか、自由なメモリーリソースが 128MB 未満の場合、最大のメモリ使用量は 64MB になります。そうでなければ、自由なメモリーリソースの半分が最大のメモリ使用量になります。ユーザは **dmconfig.ini** に、キーワード・エントリー**DB_IFMem** を加えることにより、メガバイト(MB)によってメモリ使用量の近似の上界を手動で指定することができます。

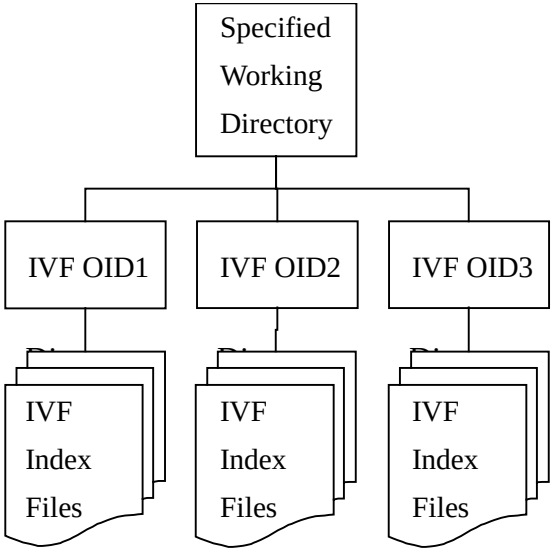
➲ 例

dmconfig.ini の中で IVF テキスト索引を作成時に 100MB のメモリ使用量を指定する:

```
DB_IFMEM = 100
```

ストレージの概要

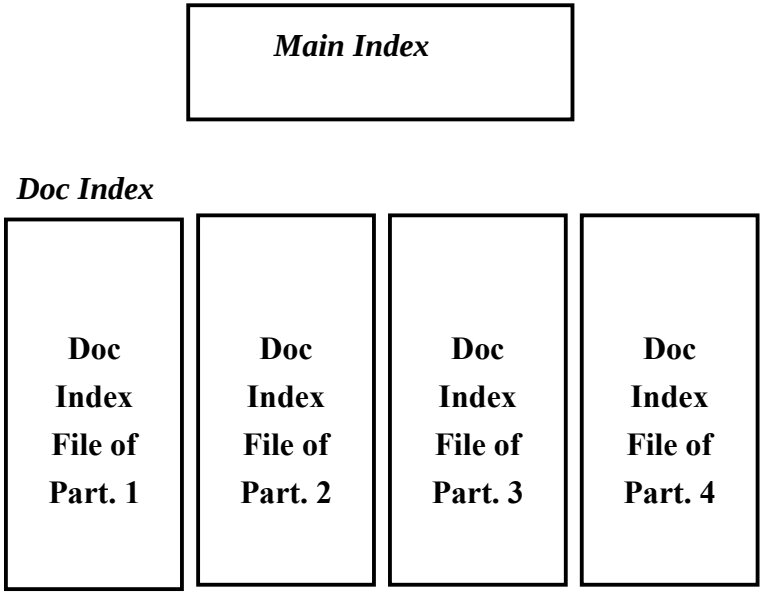
ストレージパス・パラメータによって指定された作業ディレクトリーに加えて、DBMaster は、異なる IVF 索引を管理するためにこのディレクトリー中でサブディレクトリーを生成するでしょう。各 IVF 索引はユニークな時間バージョンを行っています。したがって、DBMaster は、索引ファイルを格納するユニークなサブディレクトリーを生成するためにこのプロパティを使用することができます。サブディレクトリーを指定する方法はその後記述されます。ここで制限があります:これらのサブディレクトリーおよび IVF はユーザが IVF 索引を削除した場合、ロールバックができなくなります。



例えば、\DBMaster5.4 を指定された作業ディレクトリであるととします。そして､インデックス名前IVF1を備えたこの作業ディレクトリーの下のIVF、時間バージョン 1024476670 を作成します。その後、サブディレクトリー IVF1.1024476670 が作成されます。IVF はすべてサブディレクトリーに存在します。IVF には異なるターム型があり、1 バイト・ターム、ユニグラム・タームおよびバイグラム・タームの 3 種類があります。また、各インバーテッドファイルは、テキスト・サイズによっていくつかのパーティションを持っています。

下記は 4 つの分割を備えた IVF 索引ファイルの概念の構造です。シグネチャ・テキスト索引と IVF テキスト索引のどちらを選ぶことは、下記の要因に左右されます:

1. 索引サイズ–シグネチャー索引のサイズは、**スケール・**パラメーター(それはデフォルトによるデータ・サイズの合計の 40%)によってセットされた比



Inverted-File Structure with Four Partitions

率を超過しないでしょう。インバーテッドファイル索引の平均サイズはデータ・サイズの約 1.5 倍ですが、データの特性に依存して、2 倍あるいは 3 倍まで成長するかもしれません。

2. クエリーのレスポンス時間– 十分なメモリおよび処理力を備えた現代のパソコンにおいては、データ・サイズがギガバイトであってもインバーテッドファイルの索引には秒以下のレスポンス時間を期待することができます。特にデータ・サイズが大きくなっている時、シグネチャ索引は答えるのにより長くかかるでしょう。

3. データベースとの統合– BLOB オブジェクトとして格納されるシグネチャー索引と異なり、IVF 索引は外部ファイルとして格納されます。したがって、例えば IVF 索引の削除オペレーションはロールバックすることができません。

データの特性に対して最良の方法を見つけるために両方のタイプのテキスト索引を試してはいかがでしょう。経験法としては、100MB 未満のデータ・サイズについては、シグネチャ・テキスト索引は、質問に合理的に速く応答し、より少ないストレージ空間をとります。

多数カラム上でのテキスト索引の作成

テキスト索引は多数のカラムを使用して構築することができます。CONTAINS と連結オペレーター(||)を使用して、多重カラムテキストクエリーを実行してください。ユーザは、それらの索引あるいは単なる部分のすべてのカラム上でクエリーを実行することができます。すなわち、match クエリーのカラム・リストはテキスト索引を使用するために、テキスト索引のカラム・リストに「含まれて」います。テキスト索引がカラム・リスト上で作成されなくても、テキスト索引を使用せずに多重カラムテキストクエリー構文を使用することが可能です。

多数のカラム上で探索することは、各自探索して、その後すべてのカラムのデータを合併することと論理上等価です。.

➲ **例 1**

tb_document 表の author、subject および content カラムの IVF テキスト索引 ivfidx_multiple の作成：

```
dmSQL> CREATE IVF TEXT INDEX ivfidx_multiple ON tb_document(author,subject,content);
dmSQL> SELECT author FROM tb_document WHERE
    2> CONTAINS(author || subject || content, 'reagan');
```

➲ **例 2**

部分的なカラム・リスト上のクエリ：

```
dmSQL> SELECT author FROM tb_document WHERE
    2> CONTAINS(author || content, 'reagan');
dmSQL> SELECT author FROM tb_document WHERE CONTAINS(subject, 'reagan');
dmSQL> SELECT author FROM tb_document WHERE subject MATCH 'reagan';
```

➲ 例 3

この例において、カラム **subject** はテキスト索引 **ivfidx_multiple** に含まれています。しかし、**abstract** はそうではありません。したがって、このクエリーはテキスト索引を使用しないでしょう。

```
dmSQL> SELECT author FROM tb_document WHERE
    2> CONTAINS(subject || abstract, 'reagan');  // no text index used
```

➲ 例 4

この例は、多数のカラム上のクエリーの振る舞いを例証します。

```
dmSQL> CREATE TABLE t1 (c1 char(20), c2 char (20), c3 serial);
dmSQL> INSERT INTO t1 VALUES('apple orange', 'banana grape');
dmSQL> INSERT INTO t1 VALUES('grape orange', null);
dmSQL> CREATE TEXT INDEX ix on t1 (c1, c2);
dmSQL> SELECT c3 FROM t1 WHERE CONTAINS (c1 || c2, 'apple');
    C3
==========
    1
 1 rows selected
dmSQL> SELECT c3 FROM t1 WHERE CONTAINS (c1 || c2, 'orange & grape');
    C3
==========
    1
    2
 2 rows selected
```

メディア・タイプ上でのテキスト索引の作成

DBMaster のラージオブジェクト・カラムはメディア・タイプを登録することができます。例えば、LONG VARBINARY カラムは、その内容がマイクロソフト Word ファイルであることを知ることができます。その結果、DBMaster は、マイクロソフト Word ドキュメント上で全文検索を実行する適切な機能を起動することができます。メディアフォーマットをテキスト形式に変換するため DBMaster はメディア UDF を提供します。メディアフォーマットをチェックするための UDF (CHECKMEDIAFORMAT)もあります。表 6-1 は利用可能な異なるメディア・タイプおよび関連する SQL コマンドを要約します。

メディア・タイプ	データ・タイプ	ファイル・タイプ
Microsoft Word™,	MsWordType	MsWordFileType
HTML	HtmlType	HtmlFileType
XML	XmlType	XmlFileType
Microsoft PowerPoint	MsPPTType	MsPPTFileType
Microsoft Excel	MsExcelType	MsExcelFileType
PDF	PDFType	PDFFileType

表 6-1：メディア・タイプおよび対応する SQL コマンド

内部的には、MsWordType、MsPPTType、MsExcelType、PDFType は LONG VARBINARY オブジェクトとして扱われます。HtmlType と XmlType は LONG VARCHAR オブジェクトです。また、MsWordFileType、HtmlFileType、XmlFileType 、MsPPTFileType, MsExcelFileType and PDFFileType は FILE オブジェクトです。

メディアタイプカラム上での全文検索

メディアタイプにはテキスト索引を作成し、全文検索を実行することができます。ただしそれ以前にメディア形式のデータはプレーンのテキストに変換されている必要があります。DBMaster は新たなメディア形式データを認識しないのでメディア形式からテキスト形式への変換は不可能であり、メディア形式データの全文検索はできません。DBMaster はメディア形式データをテキストデータに変換するための下記メディア UDF を提供しております。

DOC、XLS、PPT、HTM、PDF。

- DOCTOTXT(BLOB) RETURNS NCLOB;
- XLSTOTXT(BLOB) RETURNS NCLOB;
- PPTTOTXT(BLOB) RETURNS NCLOB;
- HTMTOTXT(CLOB) RETURNS CLOB;
- PDFTOTXT(BLOB) RETURNS NCLOB;

ユーザは、ちょうど規則的なテキスト・カラムでのようにメディアタイプカラム上で全文検索を実行するためにMATCHとCONTAINSを使用することができます。

➲ 例 1

PowerPoint ドキュメントを UNICODE のテキスト BLOB データを含む一時 BLOB に変換します。

```
dmSQL> create table tb ppt(pptfile long varbinary);
dmSQL> insert into tb ppt values(?);
dmSQL/Val> &e:\udf\pptfile\pfile.ppt;
dmSQL/Val>end;
dmSQL> select PPTTOTXT(pptfile) from tb_ppt;
```

➲ 例 2

MSWord タイプ・カラムを備えた表を作成、あるデータを挿入し、検索します。

```
dmSQL> create table tb minutes(id int, doc MsWordFileType);
dmSQL> insert into tb minutes values(1, 'c:\meeting\20020403-1.doc');
dmSQL> select id from tb minutes where doc match 'Jeff';
    id
==========
         1
 1 rows selected
```

ユーザは、メディアタイプカラム上にテキスト索引を作成することができます。

➲ 例 3

表 **tb_minutes** のカラム **doc** 上にシグネ—チャ・テキスト索引を作成し探索します：

```
dmSQL> create text index ix_m on tb_minutes(doc);
dmSQL> select id from tb_minutes where doc match 'Jeff';
   id
==========
         1
 1 rows selected
```

カラム・データのメディア・タイプをチェックする

メディアタイプカラムが異なるタイプのデータを含むことはありえます。DBMaster は、メディアタイプカラムへのデータの挿入か更新中に内容の確認は行います; DBMaster は、カラムのデータが指定するメディア・フォーマットと一致するかどうかチェックするために内蔵の関数 CHECKMEDIAFORMAT を提供します。関数はタイプが一致する場合、1 を返し、さもなければ 0 を返します。DBMaster はオフィス 2007-2010 バージョンでのマルチメディアタイプをサポートします。

DBMaster は以下のメディアタイプフォーマットをサポートします： DOC、XLS、PPT、HTM、PDF。

- **DOC**：MicrosoftWord
- **XLS**：Microsoft Excel
- **PPT**：Microsoft Power Point
- **HTM**：Hypertext Markup Language
- **PDF**：Portable Document Format

注　*DBMasterはPDFのフォーマットは1.2～1.7をサポートします。*

➲ **例**

メディアタイプフォーマットが正しいかをチェックします：

```
dmSQL> Create table tb checkmedia(note long varbinary);
dmSQL> insert into tb checkmedia VALUES(?);
dmSQL/Val> &E:\DOCS\Media.doc;
dmSQL/Val> end;
dmSQL> select checkmediaformat(note,'doc') from tb_checkmedia;
```

返される値は 1, 0 または NULL です。

- 1が返されるときBLOBデータは指定されたメディア形式と一致します。
- 0が返されるときBLOBデータは指定されたメディア形式とは一致しません。
- NULLが返されるときBLOBはNULLです。

表のカラムに原型のメディアタイプ(システムドメイン)が定義され、メディア形式は正しい形式ではないとき、移行は失敗する可能性があります。適切でないメディアデータを取り除く、データ型を CLOB に変換する、BLOB データをメディア形式のチェックが行われないようにするなどでこの問題を解決します。

テキスト索引を削除する

JDBA Tool、又は dmSQL の DROP TEXT INDEX 文を使ってテキスト索引を削除することができます。

➲ 例

tb_staff 表からテキスト索引 tidx_name を削除する：

```
dmSQL> DROP TEXT INDEX tidx_name  FROM tb_staff;
```

テキスト索引の再作成

索引と異なり、テキスト索引は、新規レコードが挿入されたり、古いレコードが更新されたりする際に表の内容に影響されません。その替わり、それらを手動で再作成する必要があります。テキスト索引を再作成するまで、更新したデータはテキスト索引検索で検索されません。

➲ 例

```
dmSQL> CREATE TABLE tb_song (id int, name varchar(20));
dmSQL> INSERT INTO tb_song VALUES(1,'Endless Love');
1 rows inserted
dmSQL> CREATE TEXT INDEX ix_name ON song(name);
dmSQL> INSERT INTO tb_song VALUES(2,'Love Story');
1 rows inserted
dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love';
   id             name
=========== ====================
          1 Endless Love
 1 rows selected
```

パターン検索にマッチするレコードが 2 つありますが、1 つしか回収されません。

増分テキスト索引の再作成

REBUILD TEXT INDEX コマンドを使って、更新したデータ分のみ再作成することができます。全ての新しいレコードと更新したレコードを回収し、新しいシグネチャ・ベクタを作成し、テキスト索引の後方に新しいベクタを追加します。レコードの変更がわずかの場合、REBUILD TEXT INDEX コマンドは、最も早く再作成することができる方法です。

➲ **例**

テキスト索引の増分を再作成し、結果を表示する：

```
dmSQL> REBUILD TEXT INDEX tidx_name FOR song;
dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love';
     id              name
========== ====================
         1 Endless Love
         2 Love Story
2 rows selected
```

完全テキスト索引の再作成

完全テキスト索引の再作成は、削除したドキュメント・スペースを開放します。REBUILD TEXT INDEX 構文は、削除したレコードのシグネチャ・ベクタ・スペースを開放しません。多くのドキュメントが削除され更新された場合、テキスト索引を完全に再作成するため、CREATE TEXT UPDATE 構文を使うことをお勧めします。又、テキスト索引の要素をリセットするには、完全テキスト索引再作成をする必要があります。

➲ **例 1**

表 tb_song のテキスト索引 tidx_name e を完全に再作成する：

```
dmSQL> REBUILD TEXT INDEX tidx_name FOR tb_song;
```

テキスト索引のパラメータをリセットまたは違う種類のテキスト索引へ変更するためには、まず削除してから再作成する必要があります。

➲ **例 2**

tb_song 表のシグネチャ・テキスト索引 tidx_name を IVF 索引として完全に再作成する：

```
dmSQL> DROP TEXT INDEX tidx_name FROM tb_song;
dmSQL> CREATE IVF TEXT INDEX tidx_name ON tb_song(name);
```

➲ 例 3

tb_song 表のシグネチャ・テキスト索引 tidx_name を新しい total text size を使用して完全に再作成する：

```
dmSQL> DROP TEXT INDEX tidx_name FROM tb_song;
dmSQL> CREATE TEXT INDEX tidx_name FROM tb_song (name) TOTAL TEXT SIZE 60 MB;
```

ブールテキスト検索

MATCH は、簡単なテキストパターンのほか、DBMaster に提供された複雑なブール文字も検索できます。

検索パターンで以下のブール文字を指定することができます。

'&''– AND

'|' – OR

'! ' – NOT

' (' – 左カッコ

') ' – 右カッコ

ブール文字の優先順位は、カッコ > NOT > AND > OR です。MATCH パターンにブール文字が含まれている時、ブール文字間のほかの全文字は、単純な検索パターンとして処理されます。例えば、MATCH パターンが「coffee | tea | apple juice」である場合、検索パターンが「coffee」、「tea」、「apple juice」を含みます。

➲ 例 1

「love」と「friend」を含む文書を検索する：

```
dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love | friend';
```

➲ 例 2

「love」か「friend」を含み、「endless love」を含まない文書を検索する：

```
dmSQL> SELECT * FROM tb_song WHERE name MATCH '(love | friend) & !endless love';
```

➲ **例 3**

MATCH パターンのブール文字のような結果を取得するために SQL 文を使うブール演算子（AND、OR、NOT）を使います。但し、検索パターンの部分のみテキスト索引に適用されるので、パフォーマンスが低くなります。例えば、以下の SQL コマンドは文字列「friend」の検索に使用される場合にのみテキスト索引スキャンに適用され、文字列「love」を検索する場合に、標準的な非索引検索が使用できます。

```
dmSQL> SELECT * FROM tb_song WHERE name MATCH 'love' AND name MATCH 'friend';
```

あいまい検索

限られた知識のために正確ではないパターンを検索したいと考えるかもしれません。正確なフレーズのみ認められるのであるなら、問合せ「William Clinton」では、「William Jeffery Clinton」を検出できませんし、その逆も同様です。「William & Clinton」のような条件表現は、多くの見当違いの結果を戻すかもしれません。DBMaster には、ユーザーが多くの無意味な結果を受け取ることなく、不正確な問合せを実行することができるあいまいな検索機能があります。

「?intel pentium」のような、「?」に続くフレーズは、あいまいな表現とみなされます。あいまいな表現に使用できる単語数に決まりはありません。あいまいな表現での検索に使用する言葉は、検出された結果文では最大 4 つの言葉で分けられるかも知れません。例えば、「?intel pentium」は、「Intel will release its 1GHz Pentium III processor」、「?amd k7 athlon」では「AMD has renamed its K7 processor as Athlon」を検出します。

問合せの文字は、無視されるかもしれません。例えば、「?William Jeffery Clinton」は「William Clinton」と「William J Clinton」を検出するかもしれませんが、問合せの最初の文字が必ず現れます。そのため、問合せ「?William Clinton」は、「Bill Clinton」を検出しません。

あいまいな表現は、他のテキスト条件演算子と組み合わせることができます。

➲ **例**

```
dmSQL> SELECT content FROM tb_document WHERE content MATCH '?intel pentium & ?amd k6';
```

```
dmSQL> SELECT title FROM tb_document WHERE title MATCH 'al gore | ?george bush';
```

あいまいな表現のフレーズに、他の演算子を含むことはできません。このように、表現が「?intel pentium & amd k6」のような場合、「(?intel pentium) & (amd k6)と?(intel & pentium)」とみなされ、エラーになります。

相似全文検索

あいまい検索は、多くの無関係な結果を受け取ること無く、厳密でない問合せを実行させることができます。相似合致検索はあいまい検索に似ていますが、より正確です。問合せの文字列にある全文字が、必ず結果文に現れます。ティルデ・マーク(~)に続くフレーズは、相似表現とみなされます。例えば、「~amd sales 1ghz athlon」は、「AMD announced quarterly sales of its 1ghz Athlon chip」を検出しますが、「AMD announced quarterly sales of its Athlon chip」を検出しません。

相似合致検索の表現は、他のテキスト条件関数と合わせることができます。

➲ **例**

```
dmSQL> SELECT content FROM tb_document WHERE content MATCH '~intel pentium & ~amd k6';
dmSQL> SELECT title FROM tb_document WHERE title MATCH 'al gore | ~george bush';
```

あいまい/相似の検索規則

問合せ文字列と結果文字列のマッチには、以下の 4 つのルールが適用されます。

1. 問合せの最初の文字は、必ず表示されます。例えば、問合せ「?William Clinton」は「Bill Clinton」を検出しません。

2. 複数の単語は、付随する他の単語で分けられることがあります。例えば、「?intel pentium」は「Intel will release its 1GHz Pentium III processor」や、「?amd k7 athlon は AMD has renamed its K7 processor as Athlon」を検出します。現在、結果文の問合せ文字列の文字間の追加文字数は、4 つ以上になることはありません。

3. 問合せの文字の一部は、結果文に現れないかもしれません。例えば、「?William Jeffery Clinton」は、「William Clinton や William J Clinton」を検出します。省略される最大文字数は、公式で決まります。

最大省略単語数 = 単語数 – 切り下げ(単語数× 変数).

現在の変数は 0.75 です。

4. 問合せの全単語は、元の順序どおりに戻されます。例えば、「?amd 1ghz k7 athlon」は、「AMD will announce 1GHz Athlon」を検出しますが、「AMD Athlon, formerly known as K7」 を検出しません。

「~intel pentium」のような「~」(ティルデ・マーク)に続くフレーズは、相似表現とみなされます。相似検索は、ルール 1、2、4 に適合するあいまいな検索の特別なケースです。但し、単語の切り捨てをしません。

ユーザー定義ストップワード

キーワードベースのシステムとは対照的に、フルテキストの検索ソフトウェアはストップワードを除き、ドキュメントのすべての単語をインデックスします。ストップワードは、フルテキストの検索ソフトウェアが無関係のレコードを検索しないように、インデックスおよび検索処理中に無視するようプログラムされる用語です。一般に、ストップワードのリストには、英語では多用される冠詞、代名詞、形容詞、副詞、前置詞 (the, they, very, not, of など) が含まれます。この規則は中国語などの 2 バイトエンコード文字にも適用できます、例：的, 呢, 啊, and 哈です。

ストップワードリストの検索パス

DB_StpWd =<文字列>

このキーワードは、DBMaster のインストールディレクトリの共有/ストップワードサブディレクトリに保存されているストップワードリスト定義ファイルの名前を示しています。ストップワードリスト定義ファイルは、DBMaster のテキストインデックスの結果に影響する純粋なテキストファイルです。このキーワードは、データベースがテキストインデックスを作成および検索するときに使用されます。このキーワードがないと、データベースは LCode に基づく事前定義のストップワードリスト定義を検索します。

デフォルト値:

DB_Lcode	**ストップワードリスト定義**

0 英語 (ASCII)	en.tab
1 繁体字中国語 (BIG5)	tw.tab
2 日本語 (Shift JIS + 半角)	jp.tab
3 簡体字中国語 (GB)	cn.tab
4 ラテン語 1 コード (ISO-8859-1)	en.tab
5 ラテン語 2 コード (ISO-8859-2)	en.tab
6 キリル文字コード (ISO-8859-5)	en.tab
7 ギリシャ語コード (ISO-8859-7)	en.tab
8 日本語コード（EUC-JP）	jp.tab
9 簡体字中国語(GB18030)	cn.tab
10 UTF-8 (UTF-8)	en.tab

有効範囲: ユーザー定義のストップワードリスト定義ファイルのファイル名

以下も参照: DB_LCode

使用場所: サーバ側 (テキストインデックスの作成および検索用のみ)

デフォルトのストップワードリスト

設定を指定しない場合、DBMaster は LCODE に基づく事前定義ファイル内のデフォルトのストップワードをロードします。この機能は DBMaster の以前のバージョンのユーザーも使用できます。

DBMaster はローカルディレクトリの事前定義ファイルを検索し、ディレクトリをインストールします。

ユーザー定義ストップワードリスト

設定ファイル **DB_STPWD** からストップワードリストを指定できます。DBMaster はテキストインデックスの作成またはテキストインデックスからのオブジェクト検索時にファイルをロードします。

DBMaster はローカルまたはユーザー指定ディレクトリのファイルを検索し、ディレクトリをインストールします。

ストップワードリストを無効にする

ストップワードリストを無効にするには以下の 2 つの方法があります:

事前定義ファイルをリネームまたは削除します。

存在しないストップワードリストを設定ファイルに定義します。

6.8 メモリ表を管理する

ほとんどすべての意図および目的に対して、メモリ表は DBMaster の通常の表として同じように機能します。 相違は、メモリ表が一時表であり、その寿命サイクルが接続ベースであるというところにあります。 これは、データベースへの接続がある場合のみメモリ表が存在することを意味します。ユーザーがデータベースへの接続をサーブし、例えばその日の退出時間を記録してシステムからログアウトすると、そのユーザーのメモリ表とその内容は失われます。 メモリ表は、データベースに接続されている間のみ表示されます。 データベースへの接続が失われると、表示もされません。 通常の表とは異なり、メモリ表は作成されたマシンのメモリにのみ保存されます。 グループでは使用できず、データの選択と挿入しかできないため、データ機能の更新または削除をサポートしていません。 メモリ表は、次のトランザクション制御をサポートします: コミット、ロールバック、保存ポイントの定義、保存ポイントへロールバック、内部保存ポイント。

メモリ表を作成するには、dmSQL 構文 CREATE TABLE を使用します。SQL コマンド CREATE TABLE の構文と使用の詳細は、*「SQL 文と関数参照編」*をご覧ください。

➲ 例

メモリ表 **tb_memory** を作成する：

```
dmSQL> create hash index hidx on tb_memory (id, name) bucket 31;
```

ハッシュ索引の管理

メモリ表は作成されたマシンのメモリ上に保存されます。このためメモリ表はその他の表タイプで使用される B ツリー構造を使用することができません。 メモリ表にはハッシュ索引を定義することができます。ハッシュ索引は、メモリ表にのみ作成できます。 ハッシュ索引のメリットは、ハッシュ索引に保存されたデー

タに対し素早くアクセスができることです。 ハッシュインデックスは、イコール式とイコール結合のパフォーマンスも向上させます。 テーブルにハッシュ索引の作成には CREATE HASH INDEX index_name ON table_name (column_name、 …)[bucket n]を使用します。作成されたハッシュ索引の索引名の場所、メモリ表の表名、影響のあるメモリ表内のカラム名(この値はカラムの asc/desc を指定できません)と bucket n で作成されたハッシュ表のアレイサイズを指定します例えば、メモリ表が作成されているとき、シャープ索引 hidx を 31 のアレイサイズを有するカラム id と name を使用して、メモリ表 tb_memory に作成できます。

➲ 例

前の例からメモリ表 **tb_memory** にシャープ索引 **hidx** を作成する：

```
dmSQL> create hash index hidx on tb_memory (id, name) bucket 31;
```

6.9 データ整合性管理

データ整合性は、データがある種の基準に合致することを確かめる制約あるいは規則を適用することによって検証されます。例えば、ある項目の入力値が正しい値の範囲内（例：新卒採用者の年齢は 16～90 の間になければならない）にあることを検証するのは、データ整合性の一つの例です。

表に適用できるデータ整合性には、以下の小節で述べるような種々のタイプがあります。

NULL値制約（NOT NULL）

表の全てのカラムは、初期値では NULL 値が認められます。NOT NULL キーワードをもつカラムは、NULL 値を認めないカラムであることを示します。

一意索引

6.5節「索引管理」で述べた一意索引は、表の全ての行が指定したカラムまたはカラムの組に対して重複する値（NULL 値を除く）を取ることが無いように制限します。

一意制約

UNIQUE 制約をカラムや表全体に設定することができます。UNIQUE 制約は、カラムにある全ての行が全て異なる値をとるようにします。カラムにあるいかなる行や、UNIQUE 制約を設定したカラムも同じ値ではありません。

➲ **例**

カラムに UNIQUE 制約をつけた表を作成する：

```
dmSQL> CREATE TABLE tb student (Name CHAR(50) CONSTRAINT u UNIQUE
                          mathematics SMALLINT);,
```

チェック制約

チェック制約は、表の全ての行が、一つまたは一組のカラムに指定した CHECK 条件を真にしなければならないものです。CHECK 条件が偽になる INSERT または UPDATE 文を実行すると、文は失敗します。

チェック制約は、一つのカラム（カラム制約）または一組のカラム（表制約）に対して定義されます。

カラム制約

カラム制約は特定のカラムに定義されます。同じ表の他のカラムは影響しません。新しい行を挿入したとき、あるいは既存の行を更新したときは、各カラム制約が評価されます。

表制約

表制約は一組のカラムに対して定義します。表制約は、全てのカラム制約が真と評価された後に評価されます。新規の行の挿入あるいは既存の行の更新は、表制約が真と評価されたときだけ処理されます。

➲ **例 1**

カラム制約と表制約のある表を作成する：

```
dmSQL> CREATE TABLE tb_student (mathematics SMALLINT
                                  CHECK VALUE >= 0 AND VALUE <= 100,
                                chemistry SMALLINT
                                  CHECK VALUE >= 0 AND VALUE <= 100)
                                  CHECK mathematics + chemistry <= 200;
```

➲ **例 2**

SQL99 標準構文を使ってカラム制約と表制約のある表を作成する：

```
dmSQL> CREATE TABLE tb student (mathematics SMALLINT
              CONSTRAINT con1 CHECK mathematics>= 0 AND mathematics<= 100,
                              chemistry SMALLINT
              CONSTRAINT con2 CHECK chemistry>= 0 AND chemistry<= 100,
              CONSTRAINT con3 CHECK mathematics + chemistry <= 200);
```

キーワード VALUE は、カラム制約のカラムの値を表すのに使用します。表制約では、各カラムの値を表すためにカラム名を使用します。

主キー

表には、主キーを 1 つ設定することができます。主キーはカラムまたはカラムグループで構成されます。主キーの値は、表内の行を一意に識別します。主キーは、NULL 不可カラムの一意索引と同じです。主キーを作成すると、一意索引が表に作成されます。各表には、 1 つしか主キーを設けることはできません。

主キーを作成する

➲ **例 1**

ID カラムを主キーにして表を作成する：

```
dmSQL> CREATE TABLE  tb_student(
          ID     INTEGER  PRIMARY KEY,
          name   CHAR(30),
          nation CHAR(20)
       );
```

➲ 例 2

ID と name カラムに混合主キーを作成する：

```
dmSQL> CREATE TABLE  tb_student (
         ID     INTEGER,
         name   CHAR(30),
         nation CHAR(20),
         PRIMARY KEY (ID, name)
       ) in ts_reg;
```

➲ 例 3

tb_student 表に主キーを追加する：

```
dmSQL> ALTER TABLE  tb_student PRIMARY KEY (ID , name);
```

➲ 例 4

SQL99 標準構文を使って、表領域 ts_reg にある表 tb_student に主キーPK1 を追加する：

```
dmSQL> ALTER TABLE  tb_student ADD CONSTRAINT PK1 PRIMARY KEY (ID , name) IN ts_reg;
```

➲ 例 5

SQL99 標準構文を使って、ID カラムを主キーにした表を作成する：

```
dmSQL> CREATE TABLE  tb_student (
          ID     INTEGER  CONSTRAINT pk1 PRIMARY KEY,
          name   CHAR(30),
          nation CHAR(20)
       );
```

主キーを削除する

不要になった主キーを削除することができます。主キーを削除する前に、主キーを参照する全ての外部キーを削除しておかなければなりません。

➲ 例

表 tb_student の主キーを削除する：

```
dmSQL> ALTER TABLE  tb_student DROP PRIMARY KEY;
```

外部キー（参照整合性）

表のカラムが別の表の主キーと同じ値をもつとき、外部キーと呼びます。外部キーは 2 つの表の関係を表します。外部キーは、表内の 1 カラムまたはカラムグループに作成することができ、別の表のカラムまたはカラムグループを参照するのに使用します。参照されるカラムは、主キーか一意索引であり NULL 値をもつことができません。

外部キーをもつ表に新しい行を挿入する場合、外部キーのカラムが別の表のカラムを参照するので、参照されるカラムには、挿入しようとしている行のキー値が含まれていなければなりません。もし無ければ、その行を挿入することができません。

同様に、外部キーから参照される表の行を削除する場合は、削除する前に、外部キーをもつ表から同じキー値を全て削除しておかなければなりません。主キーあるいは外部キーは、いつでも作成／削除することができます。空でない表に主キーを作成すると、DBMaster はキーの一意性をチェックします。空でない表に外部キーを作成すると、DBMaster は全てのキー値が参照される表にあるかどうかをチェックします。

外部キーを作成する

外部キーは、参照するカラムと参照されるカラムを指定することによって、別の表を参照するのに使用します。参照するカラムと参照されるカラムは、互いに対応しなければなりません。対応するカラムは、同じタイプ、同じ長さのものです。参照されるカラムは NULL 不可ですが、参照するカラムは NULL 値でもかまいません。参照されるカラムを指定しないときは、参照される表の主キーが参照されるカラムとみなされます。

➲ 例 1

表 **tb_salary** に対して、表 ***tb_staff*** を参照する外部キーを作成する：

```
dmSQL> ALTER TABLE tb_salary FOREIGN KEY f1(ID, name) REFERENCES tb_staff;
```

➲ 例 2

あるいは **tb_salary** 表の作成時に、外部キーを指定する：

```
dmSQL> CREATE TABLE tb_salary (
           ID  INT,
         name  CHAR(30),
      basepay  INT,
        bonus  INT,
          tax  INT,
       FOREIGN KEY f1 (ID, name) REFERENCES tb_staff);
```

➲ 例 3

SQL99 標準構文を使って、表 **tb_example** 作成の際に外部キー**f1** を指定す：：

```
dmSQL> CREATE TABLE tb_example (
       c1 int,
       c2 int CONSTRAINT f1 REFERENCES tb_other (c1) ON DELETE SET NULL);
```

表 ***tb_staff***に主キーが設定してある場合、参照されるカラムを指定しなくても、別の表に外部キーを作成することができます。

外部キーを削除する

外部キーで定義される関係が不要になった場合、JDBA Tool か dmSQL の DROP FOREIGN KEY 文を使用して外部キーを削除することができます。

➲ 例

表 **tb_salary** から外部キーを削除する：

```
dmSQL> ALTER TABLE tb_salary DROP FOREIGN KEY f1;
```

6.10 シリアル番号管理

DBMaster には、自動的にシリアル番号を生成する機能があります。この機能は、マルチユーザー環境で、ディスク I/O またはトランザクションロックのオーバーヘッド無しに一意的な連続番号を生成して使用するときに、特に有効になります。

DBMaster のシリアル番号は、符号付き 32 ビットの整数です。シリアル番号は、SERIAL データ型をもつカラムで生成されます。1 つの表に作成できる SERIAL カラムは 1 つのみです。

表を作成するときに、シリアル番号カラムの開始番号を指定することができます。開始番号を指定しない場合は、初期設定値の 1 から始まります。

新しい行を挿入し、シリアル番号カラムに NULL 値が与えられたときにシリアル番号が生成されます。NULL 値の代わりに整数値を与えると、シリアル番号は生成されません。最後に生成したシリアル番号より大きい整数値を与えると、与えた整数値から始まるシリアル番号を生成するようにリセットされます。シリアル番号カラムには、DEFAULT 値あるいはカラム制約を定義することはできません。

シリアル番号カラムを作成する

JDBA Tool、又は dmSQL を使って SERIAL カラムを作成することができます。シリアル番号カラムは、SERIAL 或いは BIGSERIAL キーワードとオプションの開始番号で定義します。

➲ **例**

開始番号 1001 を指定する SERIAL タイプのカラム ***ID*** をもつ表 ***tb_staff*** を作成する：

```
dmSQL> CREATE TABLE tb_staff (nation CHAR(20) DEFAULT 'R.O.C',
                                ID SERIAL(1001),
                              name CHAR(30) NOT NULL,
                          joinDate DATE DEFAULT CURDATE(),
                            height FLOAT,
                            degree VARCHAR(200)) IN ts_reg;
```

シリアル番号を生成する

シリアル番号は、SERIAL カラムに NULL 値が挿入された時に、自動的に生成されます。

➲ **例**

表 ***tb_staff*** に新しい行を挿入し、カラム ***ID*** にシリアル番号を生成させる：

```
dmSQL> INSERT INTO tb staff VALUES
              ('U.S.A', NULL, 'Jeff', NULL, 6.6, 'Director');
```

シリアル番号を取得する

各接続のシステム表 SYSCONINFO の LAST_SERIAL カラムに最後に生成されたシリアル番号が保存されています。シリアル番号を含むレコードが挿入された後、LAST_SERIAL からシリアル番号を取得することができます。

➲ **例**

挿入されたレコードの生成されたばかりのシリアル番号を取得する：

```
dmSQL> select LAST SERIAL from SYSCONINFO;
LAST SERIAL
===========
        200
 1 rows selected
```

シリアル番号をリセットする

シリアル型のカラムのカウンタをリセットすることができます。表を修正することなく、シリアル型のカラムで新規シーケンスを開始することができます。

➲ **例**

表 *tb_staff* のシリアルのカウンタ値を現在の値から 3000 に修正する：

```
dmSQL> ALTER TABLE tb_staff SET SERIAL 3000;
```

6.11 ドメイン管理

ドメインは、カラムの定義に使用することができる整合性制約の一種です。ドメインはカラムのデータ型を指定しますが、カラム初期値とカラム制約を指定することもあります。ドメインを使用してカラムを定義すると、カラムはドメインのプロパティ（データ型、初期値、カラム制約）を継承し、これらを明示的に指定する必要がなくなります。

ドメインを使用してカラム初期値とカラム制約を指定すると、通常のカラム定義でこれらを指定するのと同じ結果が得られます。カラム定義でカラム初期値を指定すると、ドメインで指定したカラム初期値は無効になります。カラム定義でカラム制約を指定すると、ドメインで指定したカラム制約に加えて使用されます。

ドメインを使用してカラムを定義するときにカラム制約を追加指定する場合は、追加したカラム制約がドメインのカラム制約と競合しないことを確かめなければなりません。

DBMaster は、カラム制約の競合をチェックしないので、どんな値も入力できないカラム制約を定義してしまうかもしれません。SERIAL タイプを除いて、DBMaster がサポートする全てのデータ型をドメインに使用することができます。

ドメインを作成する

ドメインは、ドメイン名、データ型と、オプションの初期値、制約によって定義されます。JDBA Tool、又は dmSQL の CREATE DOMAIN 文を使ってドメインを作成することができます。例えば、**title**（映画、CD、ビデオテープ）カラムが全て 35 文字以下の VARCHAR データ型をもち、NULL 値を認めないようなドメインを作成するとします。

➲ **例 1**

キーワード VALUE は、ドメインで定義されるカラムの値を表すのに使用されます。CREATE TABLE 文で使用できるようにするため、ドメイン ***title_type*** を作成する：

```
dmSQL> CREATE DOMAIN title_type VARCHAR(35) CHECK VALUE IS NOT NULL;
```

➲ **例 2**

CREATE TABLE 文で、ドメインを使ってカラムを定義する：

```
dmSQL> CREATE TABLE movie_titles (title title_type, ..., ...);
```

TEXT CONVERTERでドメインを作成する

メディア型は特有のフォーマットを持つドメインです。CREATE DOMAIN 句の中の TEXT CONVERTER 文にてドメインを作成することができます。

DBMaster はテキスト索引作成と PURETEXT() UDF のために TEXT CONVERTER 式を CLOB、NCLOB、BLOB、FILE データをテキストデータへの変換に使用します。TEXT CONVERTER の関数名は引数型と関連のある BLOB のみ含みます。返答される結果のタイプは CLOB もしくは NCLOB 型となり、そうでなければ DBMaster はエラーを返します。TEXT CONVERTER 文では 32767 以上のドメインを作成することができません。

注 *式、引数のない関数、TEXT CONVERTER句よりも多い引数を含む関数を定義することはできません。*

➲ **例**

ドメイン **MSWORDTYPE** を定義します：

```
dmSQL> CREATE DOMAIN MSWORDTYPE AS BLOB
         TEXT CONVERTER DOCTOTXT
         CHECK VALUE IS NULL OR CHECKMEDIAFORMAT(VALUE,'DOC') = 1;
```

ドメイン **MSWORDTYPE** で表 tb_MT を作成します：

```
dmSQL> CREATE TABLE tb_MT (C1 MSWORDTYPE);
```

ドメインを削除する

ドメインは、ドメインを参照するカラムが無い場合は削除することができます。JDBA Tool、又は dmSQL の DROP DOMAIN 文を使って、ドメインを削除することができます。

➲ **例**

DROP DOMAIN 文を使ってドメインを削除する：

```
dmSQL> DROP DOMAIN title_type;
```

6.12 オブジェクトのアンロード/ロード

データベースのデータを外部のファイルに保存する必要があるかもしれません。そのような場合は、UNLOAD 文と LOAD 文を利用します。データベースからアンロードしたオブジェクトは、データベースから削除されることはありません。それらは 1 つ以上の外部テキストファイルに保存されて

いるだけです。オブジェクトをデータベースにロードすると、そのオブジェクトのスキーマも再生成されます。

オブジェクトのアンロード

アンロードは、データベースの内容を外部のテキストファイルに転送するために使用する dmSQL に備わった機能です。アンロード処理を行うと、2 つのテキストファイルが生成されます。1 つは拡張子が**.s0** のファイルでデータベース・オブジェクトを創設するためのスクリプトを保存し、もう 1 つは拡張子が**.bn** のファイルで、BLOB データを保存します。

アンロード構文には、次の 8 つのオプションがあります。データベースのアンロード、表のアンロード、スキーマのアンロード、データのアンロード、プロジェクトのアンロード、モジュールのアンロード、ストアド・プロシージャのアンロード、プロシージャ定義のアンロードです。オブジェクトのアンロードは、その対象オブジェクトへの SELECT 権限を有するユーザーしか行うことができません。例えば、ある表の SELECT 権限を持っているユーザーは、その表の中身しかアンロードできません。DBA、SYSDBA 権限を持つユーザーまたは SYSADM のみデータベースのアンロードを実行することができます。

UNLOAD [DB | DATABASE]

DBA、SYSDBA 権限を持つユーザーまたは SYSADM はデータベースの内容を外部テキストファイルにアンロードすることができます。このファイルには、セキュリティ、表領域、定義、索引、シノニム、データ等の情報があります。1 つのデータベースにつき、少なくともスクリプトと BLOB データの 2 つの外部ファイルが生成されます。

➲ **例**

```
dmSQL> UNLOAD DB TO empdb;
```

外部テキストファイル名は、**empdb** です。初期設定では、これらのファイルは現在の作業ディレクトリに生成されます。上記の記述では、少なくとも **empdb.s0** と **empdb.b0** の 2 つのファイルが生成されます。アンロードした BLOB ファイルの **empdb.b0** がオペレーティング・システムの許容サイズを超えた場合、dmSQL はファイル **empdb.b1**、**empdb.b2**、…、**empdb.bn**,と

順に最大 99 まで作成します。スクリプト・ファイル **emodb.s0** は、オペレーティング・システムの許容サイズ以内で常に 1 つしか生成されません。

UNLOAD TABLE

このオプションは、外部ファイルに表をアンロードし、その定義、シノニム、索引、主キー、外部キー、表のデータを記録します。所有者と表名に使用するワイルドカードの「_」と「%」は、DOS の「?」と「*」に対応します。「_」は、1 文字を表し、「%」は複数の文字を意味します。

UNLOAD SCHEMA

このオプションの使い方は、UNLOAD TABLE に似ています。これは表の定義のみアンロードし、表のデータをアンロードすることはできません。UNLOAD TABLE 文と同じワイルドカードを使用します。

UNLOAD DATA

このオプションは、表の全てのデータをアンロードし、表の定義をアンロードしません。UNLOAD DATA は、前の 2 つのオプションと同じワイルドカードを使用します。アンロードする表の SELECT 権限を有するユーザーのみ、UNLOAD DATA 文を実行することができます。

DBMaster 3.6 以降のバージョンでは、データのアンロードに追加の構文が使用できるようになりました。

```
dmSQL> UNLOAD DATA FROM (select 文) TO <ファイル名>;
```

select 文が join の場合、そのプロジェクション・カラムは同じ表のものでなければなりません。DDL 文、削除、挿入、更新文は実行できません。

➲ **例 1**

有効な構文：

```
dmSQL> UNLOAD DATA FROM (select t1.c1, t1.c2 from t1, t2 where t1.c1= t2.c1) TO f1;
```

➲ **例 2**

無効な構文：

```
dmSQL> UNLOAD DATA FROM (select t1.c1, t2.c1 from t1, t2 where t1.c1 = t2.c1) TO f1;
```

プロジェクション・カラムには、集計関数や組み込み関数を使用できません。

➲ **例 3**

無効な構文：

```
dmSQL> UNLOAD DATA FROM (select avg(c1) from t1) TO f1;
dmSQL> UNLOAD DATA FROM (select now() from t1) TO f1;
```

ビューとシノニムは、指定することができます。

➲ **例 4**

有効な構文：

```
dmSQL> UNLOAD DATA FROM (select * from s1 where c1 > 10) TO f1;
dmSQL> UNLOAD DATA FROM (select * from v1 where c1 < 10) TO f1;
```

UNLOAD PROJECT

このオプションは、外部テキストファイルにプロジェクトをアンロードします。

UNLOAD MODULE

このオプションは、外部ファイルにモジュールをアンロードします。

UNLOAD [PROC | PROCEDURE]

このオプションは、外部ファイルにストアド・プロシージャをアンロードします。

UNLOAD [PROC DEFINITION | PROCEDURE DEFINITION]

このオプションは、外部ファイルにストアド・プロシージャの定義をアンロードします。

➲ **例 1**

現ユーザーで表 **e tab** をアンロードする。表名にスペースがある場合は、ダブルクォーテーションで括ります：

```
dmSQL> UNLOAD TABLE FROM "e tab" TO empfile;
```

➲ **例 2**

emptab、**empname** のような **emp** で始まる SYSADM が所有する表を全てアンロードする：:

```
dmSQL> UNLOAD TABLE FROM SYSADM.emp% TO empfile;
```

➲ **例 3**

表名が **ktab** の表スキーマを全てアンロードする：

```
dmSQL> UNLOAD SCHEMA FROM %.ktab TO kfile;
```

ワイルドカードが入った表名をアンロードする場合は、名前にエスケープ文字（\）かダブルクォーテーション（""）を加えます。

➲ **例 4**

abc%という名前の表からデータをアンロードする：

```
dmSQL> UNLOAD DATA FROM abc\% TO abcfile;
dmSQL> UNLOAD DATA FROM "abc%" TO abcfile;
```

オブジェクトのロード

LOAD 文は、テキストファイルにアンロードしたデータベース・オブジェクトを、データベースに転送するために使用する dmSQL の機能です。ロード文には 7 つのオプションがあります。データベースのロード、表のロード、スキーマのロード、データのロード、プロジェクトのロード、モジュールのロード、ストアド・プロシージャのロードです。ファイルのアンロードとロードには、同じオプションを使用します。つまり、アンロードしたオプションで、テキストファイルからデータベースにロードします。テキストファイルをロードする際、自動的にトランザクションをコミットするためにコマンド数<n>をセットします。初期設定数は 1000 です。<n>のサイズは、トランザクションの成否、或いはローディングの速度に影響します。<n>値が大きいとジャーナルがすぐに一杯になり、トランザクションのエラーを引き起します。<n>値が小さいと、コミットされるトランザクション数が少なくなり、ローディングの速度が落ちます。ロード処理の際にエラーが発生すると、ログファイルにエラーメッセージが記録され、ロード処理は続行します。このログファイルは、実行済みのコマンドを元に戻す

ためにシステムによって利用され、ロードされる外部テキストファイルと同じディレクトリに保存されます。

LOAD [DB | DATABASE]

LOAD [DB | DATABASE]文は、データベースの中身を新しいデータベースに転送するために使用します。まず、データベースを外部テキストファイルに移すためにアンロードします。そして LOAD DB 文で、データベースの中身をテキストファイルからロードします。データベースをロードする前に、新しいデータベースを作成します。新しいデータベース名には、古いデータベース名と違う名前を指定することができます。DBA、SYSDBA 権限を持つユーザーまたは SYSADM のみこのコマンドを実行することができます。

➲ 例

次の LOAD DB の SET オプションは、DBMaster 3.6 以降のバージョンに加えられました。

```
dmSQL> SET LOAD DB [safe | fast];
```

LOAD DB を SAFE にセットすると、データベースはノーマル・モードで運用されます。ロードの際にエラーが発生すると、ロード・ユーティリティは、最後にコミットした文をロールバックし、エラーメッセージが表示され、ロード・ユーティリティのログファイルに書き込まれます。LOAD DB を FAST にセットすると、DBMaster 3.6 より前のバージョンのユーティリティをロードする方法は、全ロード処理作業は非ジャーナル・モードで行われます。LOAD DB を FAST にセットすると、ロード機能をスピードアップしますが、エラーが発生した場合はデータベースを非ジャーナル・モードで終了させます。例えば、ロードするファイルに表領域の作成があり、**dmconfig.ini** ファイルに定義されていないと想定する場合、LOAD DB に SAFE オプションを使用すると、エラー・メッセージ「ERROR(8002): [DBMaster] Config ファイルにキーワードの入力が必要です」が表示され、そのロード文はロールバックされます。また、LOAD DB に FAST オプションを使用すると、エラーメッセージ「ERROR(30017): [DBMaster] 非ジャーナルモードでエラーが発生したので、データベースを終了します」が表示されます。

注　*初期設定は、SET LOAD DB SAFE です。*

LOAD TABLE

このオプションは、スキーマとデータを含む表の中身をテキストファイルからロードします。テキストファイルから表をロードする際は、表名が一意であることを確認して下さい。

LOAD SCHEMA

このオプションは、テキストファイルにある表のスキーマをロードし、データをロードしません。テキストファイルからスキーマをロードする際、表名が一意であることを確認して下さい。

LOAD DATA

外部テキストファイルからデータをロードする際は、対応する表が存在しなければなりません。DBMaster 3.6 より前のバージョンでは、LOAD DATA 処理の際にエラーが発生すると、最後にコミットしたコマンドまでロールバックされました。

➲ 例

DBMaster 3.6 以降のバージョンでは次のオプションを使用することができます。

```
dmSQL> Set load data skip [error] | stop [on error];
```

LOAD DATA SKIP ERROR にセットすると、データをロードする際、以下のエラー・メッセージは無視されます。

ERROR(401)　一意キーの制約違反。

ERROR(410)　参照制約違反：親キーに値がありません。

ERROR(6521)　表又はビューが存在しません。

ERROR(6002)　構文エラー。

ERROR(6015)　不完全な SQL 文です。

エラーは無視され、ロード・ユーティリティはそれ以降のコマンドの実行を再開します。上記のエラーは、データ・ロードの際の最も一般的なエラーです。LOAD DATA STOP や STOP ON ERROR にセットすると、エラーの際に LOAD 文全体がロールバックされます。初期設定は、LOAD DATA

SKIP ERROR です。データ・ロードの際に発生した全エラー・メッセージは、ログファイルに記録されます。

LOAD MODULE

このオプションは、外部ファイルからモジュールをロードします。

LOAD PROJECT

このオプションは、外部ファイルからプロジェクトをロードします。

LOAD [PROC | PROCEDURE]

このオプションは、外部ファイルからストアド・プロシージャをロードします。

➲ **例 1**

empdb というファイルからデータベースをロードし、ロードの際に 100 コマンドづつ自動的にコミットする。システムは、同じディレクトリに **empdb.log** という名前のログファイルを生成します。

```
dmSQL> LOAD DB FROM empdb 100;
```

➲ **例 2**

empfile というファイルから表をロードし、ロードの際に 50 コマンドづつ自動的にコミットする：

```
dmSQL> LOAD TABLE FROM empfile 50;
```

➲ **例 3**

datafile というファイルからデータをロードし、ロードの際に初期設定の 1000 コマンドづつ自動的にコミットする：

```
dmSQL> LOAD DATA FROM datafile;
```

6.13 システム表の確認

DBMaster では、様々なシステム表に全てのスキーマ・オブジェクトの詳細情報を保管しています。システム表の詳細については、*システムカタログ参照*を参照してください。

スキーマオブジェクト情報	システム・カタログ表名
表	SYSTABLE
カラム	SYSCOLUMN
ビュー	SYSVIEWDATA
シノニム	SYSSYNONYM
索引	SYSINDEX
ドメイン	SYSDOMAIN
シリアル番号	SYSCONINFO
表制約	SYSTABLE
カラム制約	SYSCOLUMN
ドメイン制約	SYSDOMAIN

表 6-3：システムカタログ表のスキーマ情報

6.14 ディスク容量の計算

これまで述べたように、スキーマオブジェクトの中で物理的にディスク領域を占有するのは、表と索引のみです。ディスク領域を管理するには、表と索引を作成する前に、これらのサイズと表領域を決定しなければなりません。見積り段階で、表を置く表領域をどのように構成するか、将来データベースがどれだけハードウェアを必要とするかについて、明確な構図をもたなければなりません。

一般に、データベースの表を幾つかの表領域に分割すると、全ての表を一つの大きな表領域に置くよりも、高いパフォーマンスが得られます。逆に、小さい表領域を多く設けると、管理が複雑になります。このため、表領域を設計するためには、必要なディスク容量を明確に計算しておきます。

表サイズの見積り

ここでは、表と索引のサイズを見積る方法について解説します。

表と索引のサイズは、次の式で見積ります。

表サイズ ＝ 行サイズ x 行数 x **1.05**

索引サイズ ＝ キーサイズ x 行数 x **1.20**

この二つの式から、表領域に含まれる全ての表と索引のサイズを合計して、表領域のサイズを見積もることができます。係数の 1.05 と 1.20 は、システムが必要とするリソースオーバーヘッドの見積りです。行サイズとキーサイズには、内部レコードヘッダのサイズが含まれます。以下の小節で、行サイズとキーサイズの計算法を説明します。

行サイズ

DBMaster では、BLOB データを除いて、行サイズは 3996 バイトを超えることはできません。行サイズは、各カラムのサイズと内部レコードヘッダサイズから決まります。

内部行ヘッダのサイズは、次の式で計算します。

内部レコードヘッダサイズ ＝ （カラム数 ＋ 1 ）x 4

各データ型は、下記のカラムサイズをもちます。

タイプ	カラムのサイズ
BIGINT	8
BINARY(n)	N
BIGSERIAL	8
CHAR(n)	N
SMALLINT	2
INTEGER	4
FLOAT	4
SERIAL	4
DOUBLE	8
DECIMAL(p,s)	[(p+1)/2]+2
TIME	4
DATE	4

TIMESTAMP	12
OID	16
VARCHAR(n)	1-3992
FILE	20
LONG VARBINARY	20+X
LONG VARCHAR	20+X

表 6-4：データ型のサイズ

注 *VARCHARは、任意の文字をもつことができる可変長のデータ型です。BLOB型 (LONG VARCHARまたはLONG VARBINARY)は、48バイトのみデータファイルに取られます。実際のデータは、BLOBファイルまたはデータファイルに格納します。詳細は7章の「ラージオブジェクト管理」を参照してください。カラムの値がNULLのときはスペースを必要としません。*

➲ **例**

5 カラムの表を作成する：

```
dmSQL> CREATE TABLE tb_staff(ID INTEGER NOT NULL,
                             name CHAR(30) NOT NULL,
                           height FLOAT,
                           degree VARCHAR(200),
                          picture LONG VARCHAR);
```

表作成後、次の行を表に挿入したときのレコードサイズは以下のように計算します：

(3001, "Jeff Yang", 175.5, "Stanford PhD.", [pic1]) ここで pic1 は写真画像です。

データ項目	データ型	サイズ
ID	integer	4 バイト
Name	char	30 バイト
Height	float	4 バイト
Degree	varchar	13 バイト

picture	long varchar	48 バイト
行ヘッダー	—	24 バイト
	合計	123 バイト

＝ (4＋30＋4＋13＋48)＋(5＋1)×4

＝ **123** バイト

(3002, "George Wang", 180.0, "NCTU Ms.", NULL)

データ項目	データ型	サイズ
ID	integer	4 バイト
name	char	30 バイト
height	float	4 バイト
degree	varchar	8 バイト
picture	long varchar	0 バイト
行ヘッダー	—	24 バイト
	合計	70 バイト

＝ (4＋30＋4＋8＋0)＋(5＋1)×4

＝ **70** バイト

DBMasterは、行を挿入・更新するときに、レコードサイズがMAXTUPLEN[1]バイト以下であることを確かめます。また、表を作成するときに、最小レコードサイズがMAXTUPLEN以下であることも確認します。

この例の最小レコードサイズは、次のように計算します。

[1] MAXTUPLEN :ページサイズが 4KB のとき値は 3968、8KB では 8064、16KB では 16256、32KB のとき 32640 となります。

最小レコードサイズ ＝ **(4＋30＋0＋0＋0)＋(5＋1)×4**

＝ **58** バイト

最小レコードサイズが MAXTUPLEN バイト以下なので、この表を作成することができます。

キーサイズ

キーのストレージサイズが索引カラムと内部レコードヘッダから計算される点は、キーもレコードも同じです。ただし、内部行識別子のために、16 バイトを余分に必要とします。また、内部レコードヘッダのサイズは、行識別子のために 4 バイト大きくなります。従って、内部レコードヘッダの計算式は次のようになります。

内部レコードヘッダサイズ ＝（カラム数＋１＋１）×4

例えば、SMALLINT タイプのカラムに索引を作成する場合、各キーのサイズは次のようになります。

キーサイズ＝ **2＋16＋(1＋1＋1)×4**

＝ 30 バイト

この場合、キーカラムに 2 バイト、内部行識別子に 16 バイト、レコードヘッダに 12 バイトが使用されます。

表領域と表の見積り

次に表領域と表のサイズの見積り例を挙げます。表領域には、表 ***A***、***B***、***C*** と表 ***A*** の索引 ***D*** があるとします。表 ***A*** には INTEGER、INTEGER、CHAR(10) のカラムがあります。表 ***B*** には SMALLINT、CHAR(10)、FLOAT、VARCHAR(200)のカラムがあります。表 ***C*** には SMALLINT、INTEGER、LONG VARCHAR のカラムがあります。索引 ***D*** は表 ***A*** の最初のカラムに作成されます。表 ***A***、***B***、***C*** には、各々1500、3000、250 行あるとします。

レコードサイズとキーサイズは、以下で計算されます。表 ***B*** の VARCHAR カラムの平均長を 80 バイトと仮定します。表 ***C*** の BLOB カラムは 48 バイトです。

表 *A* :　　レコードサイズ＝ (4＋10)＋3×4
＝ 26 バイト

表 *B* :　　レコードサイズ＝ (2＋10＋4＋80)＋5×4
＝ 116 バイト

表 *C* :　　レコードサイズ＝ (2＋4＋48)＋4×4
＝ 70 バイト

表 *C* の BLOB 項目の平均サイズを 9000 バイトとすれば、BLOB ファイルのフレームサイズを 11KB にする必要があります。BLOB データの詳細情報については、7章の「ラージオブジェクト管理」を参照してください。

索引 D :　　キーサイズ＝ 4＋16＋3×4
＝ 32 バイト

各表のサイズは、以下で計算されます。表 *A* のサイズは索引 *D* のサイズも含む点に注意してください。

表 *A* :　　表サイズ＝ (26×1500×1.05)＋(32×1500×1.2)
＝98550 バイト

表 *B* :　　表サイズ＝ 116×3000×1.05
＝ 365400 バイト

表 *C* :　　表サイズ＝ 70×250×1.05
＝ 18375 バイト

表 *C* の BLOB ファイルのサイズは、250 フレーム（行毎に 1 フレームが必要）です。

この見積り数値から、上記の表と索引を格納するためには、少なくとも 1 個のデータファイル（482325 バイト）と BLOB ファイル（250 フレーム、フレームサイズ 11KB）をもつ表領域を作成する必要があります。

表領域のサイズは、表領域を作成するときに見積り、後になって表領域にファイルを追加したり、ファイルを拡張したりする手間を省くようにします。

6.15 データベース整合性をチェックする

DBMaster には、データベース整合性をチェックする様々な SQL 構文があります。データベースの不整合とは、例えば、表に無いキーが索引にある、親表に無いキーが外部表にある等があります。DBMaster では、6 種類の異なるレベルで整合性をチェックすることができます。データベースが大きい場合、整合性のチェックには時間がかかると共に、これらの SQL 文にはロックが掛けられます。必要なときだけ使用するようにしてください。

索引をチェックする

DBMaster では、索引構造および索引と表の関係をチェックすることができます。索引構造（B ツリー）が正しいか、データが指定順に並んでいるか、索引キーがデータレコードと正確にマッチしているか、等々をチェックします。

索引に問題があると推測される時は、SQL 文を使用して問題があるかどうか検証することができます。索引が壊れているときは、索引を削除して再作成し、壊れた索引を直します。

➲ **例**

表 **tb_staff** の索引 **idx_desc** の整合性をチェックする：

```
dmSQL> CHECK INDEX tb_staff. idx_desc;
```

表をチェックする

表に関連する全てのレコード、索引、BLOB データをチェックし、外部表と親表の関係をチェックすることができます。表に不整合があるときは、全

てのデータをアンロードし、表を削除して再作成し、全レコードを再挿入します。

➲ 例

表 **tb_staff** の整合性をチェックする：

```
dmSQL> CHECK TABLE tb_staff;
```

システム表をチェックする

システム表の整合性をチェックすることができます。システム表にエラーが発生すると、データベースの破壊状況は深刻です。

➲ 例

システム表の整合性をチェックする：

```
dmSQL> CHECK CATALOG;
```

データベースをチェックする

データベース全体（システム表と全ての表領域を含む）をチェックすることができます。

➲ 例

データベース全体の整合性をチェックする：

```
dmSQL> CHECK DB;
```

破壊された部分がある場合、データベースのバックアップを取ってあれば、最新のバックアップを使用してデータベースをリストアすることができます。詳細については、15 章の「リカバリ、バックアップ、リストア」を参照してください。

データベースのバックアップを取っていないときは、索引が破壊されている場合は、索引を削除して再作成します。他の種類の破壊が発生した場合は、直ちにデータベースのバックアップ（全てのデータとジャーナルを含む）を取るべきです。次に、データベースを終了して再起動し、CHECK 文を再度実行します。DBMaster が自動的にクラッシュしたデータベースをリカバリして、ある種の破壊は修復されているかも知れません。しかし依然

として不整合がある場合は、CASEMaker テクニカルサポート員にお問合せください。

ユーザーファイルのチェック

データベースがウォームスタートの方式で起動時に、DBMaster はユーザーがユーザーファイルをチェックすることを許可します。ユーザーが当該機能を ON にする場合、DBMaster はデータベースがウォームスタートの方式で起動時に全てのユーザーファイルがディスクに存在するかを確認し、存在しない場合に、DBMaster は DBA 以上の権限を持つユーザーが移動されたファイルを操作することを回避するために、エラーメッセージを出して、ユーザーに知らせます。当該エラーメッセージがログファイル DMEVENT.LOG に記録されます。ユーザーは **DB_CHKFL** を設定することによって、この機能を有効にすることができます。

注　*この機能はシングルユーザモードをサポートしていません。*

6.16　スキーマ・オブジェクトの統計を更新する

スキーマ・オブジェクト(表、索引、カラム)の古い統計値は、DBMaster 最適化に非効率な SQL 文の計画を選択させる結果となります。統計値を更新した後に、大量のデータがデータベースに挿入された場合、その値を再び更新します。

統計更新と SQL 最適化の詳細については、19 章の「問い合わせの最適化」の「統計」を参照して下さい。

➲ **例 1**

スキーマオブジェクトに対して統計を更新する：

```
dmSQL> update statistics;
```

データベースサイズが巨大な場合、すべてのスキーマオブジェクトの統計値更新に長時間を要します。別の方法として前回の更新から変更処理のあった特定のスキーマオブジェクトにのみ統計更新を実行することやサンプル率を指定して実行することができます。

➲ 例 2

表の統計を更新する：

```
dmSQL> update statistics table1, table2, user1.table3;
```

➲ 例 3

tb_staff の索引 idx_desc の統計を更新する：

```
dmSQL> update statistics tb_staff (index idx_desc);
```

➲ 例 4

表領域 **ts_reg** の統計を更新する：

```
dmSQL> update tablespace statistics ts_reg;
```

➲ 例 5

表 **tb_staff** の索引 **idx_desc** と **idx_fill** に対して統計を更新する：

```
dmSQL> update statistics tb_staff (index idx_desc, idx_fill);
```

データベースは実行している場合、ユーザーはシステムストアドプロシージャ SetSystemOption を使用できて指定された統計値が変更することができます。

➲ 例 6

以下の例示はデータベースは実行している場合に更新統計サンプルを 60 に設定するため使用されます：

```
dmSQL> call setSystemOption('STSSP', '60');
```

更新統計と SQL オプティマイザーについての詳細情報は第 18 章パフォーマンスのチューニングを参考してください。

7 ラージオブジェクト管理

文書テキスト、画像、音声、ビデオのような可変長のデータオブジェクトのことをラージオブジェクト（LO）と言います。DBMaster は、ラージオブジェクトを非常に柔軟に取り扱い、非構造的データのための優れたラージオブジェクト機構を提供します。

DBMaster では、表に入れることができる LO の個数に制限がありません。また、LO カラムの合計サイズの制限もありません。これは、LO カラムの容量の上限が 2GB であることを意味します。DBMaster は、SQL 言語を拡張してラージオブジェクトを直接アクセスできますので、特別な構文を学ぶ必要がありません。LO カラムの全てのアクセスは透過的な SQL 文のため、ラージオブジェクトの使用法を簡単に覚えることができます。更に、ユーザーは SQL 文あるいは ODBC インタフェースを使用して、ファイルにある LO データを入出力することができます。

LO は常に 1 つの単位としてディスクに書き込まれますが、ユーザーは LO 全体を読み込むことも、その一部を読むこともできます。SELECT、UPDATE、INSERT、DELETE 文に LO を指定することができます。LO は、NULL 値かどうかテストする論理式で使用することができます。更に、MATCH 関数で LO のパターンマッチ探索を実行することができます。MATCH 関数は、LO カラムにのみ使用し、ワイルド文字が使用できない点を除けば、LIKE 関数と同じ働きをします。

LO オブジェクトは、算術式あるいは文字列式の演算には使用できません。また、以下の中でも使用することもできません。

- 集計関数

- IN、ANY、EXIST、LIKE述語
- GROUP BY句
- ORDER BY句

LO には、データベースに格納されるバイナリ・ラージオブジェクト（BLOB）と外部ファイルとしてファイルシステムに格納されるファイルオブジェクト（FO）の 2 種類があります。

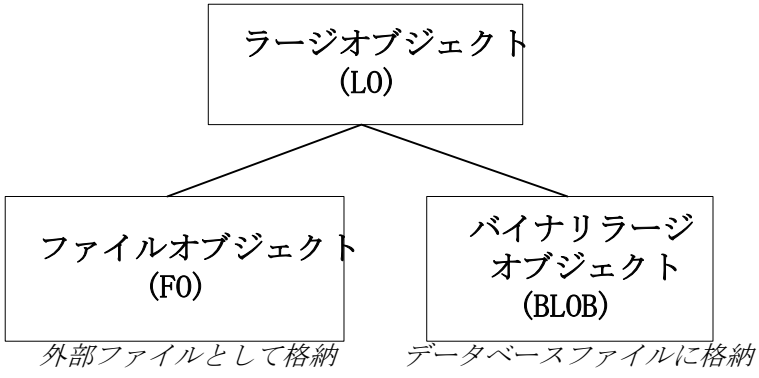


図 7-1：DBMaster がサポートするラージオブジェクト

データベースファイルに格納される BLOB は、DBMaster を通じてのみアクセスすることができ、DBMaster にそなわっているトランザクション制御、ログ情報、リカバリ等のデータ整合性が維持されます。BLOB はレコードの更新中、同じテ―ブルのタプル間でのみ共有できます。一方、FO は別の表間でも共有することができます。更に、データベース以外のアプリケーションと共有する必要がある時も、FO を使用すると一層柔軟になります。

7.1 BLOB 管理

BLOB には、LONG VARCHAR と LONG VARBINARY の 2 種類があります。LONG VARCHAR タイプは、メモ、長いテキスト、HTML ソースファイル、プログラム・ソースファイルのようなテキストデータを格納します。これ

に対して、LONG VARBINARY タイプは、画像、音声、スプレッドシート、プログラム・モジュール等のバイナリデータを格納します。

BLOB データは、サイズによりデータファイルと BLOB ファイルのいずれかに格納されます。データファイルのフォーマットは決まっていますが、BLOB ファイルはカスタマイズすることができるので、パフォーマンス向上とディスク効率化を実現します。

BLOB のログを取ると、ジャーナル領域を大量に占有しパフォーマンスを低下させるので、オプションになっています。BLOB ジャーナルを OFF にすると、ログ容量を節約しパフォーマンスは改善します。但し、BLOB ログを取らない場合、バックアップからリストアしたデータベースにある BLOB データの内容の正確さは、保証されません。BLOB のログを取る場合は、ジャーナルの容量が BLOB データを収容するのに十分なサイズであることを確認しておきます。

BLOB容量をカスタマイズする

DBMaster は、BLOB データを何処に格納すべきかを自動的に判断します。LONG VARCHAR と LONG VARBINARY カラムのサイズが小さく、行の合計サイズが最大サイズを超えない場合、データベースの他のデータと同様に、データベースのカラムに配置されます。レコードを取り出す時に、BLOB データも取り出すことができるので、効率が上がります。

行の合計サイズが最大サイズを超える場合は、BLOB データは分離して格納されます。この場合、BLOB データを取得する（間接 BLOB と言います）ためには、データ行の取り出しと BLOB データの取り出し 2 種類のディスク操作が必要になります。

間接BLOBは、BLOBサイズによって、表と同じ表領域のデータファイルかBLOBファイルのいずれかに格納されます。BLOBサイズが 16240[2] (16 KBページサイズ)バイト以下の場合は、間接BLOBカラムのデータは、データファイルに格納されます。それ以外のときは、BLOBファイルに格納されます。

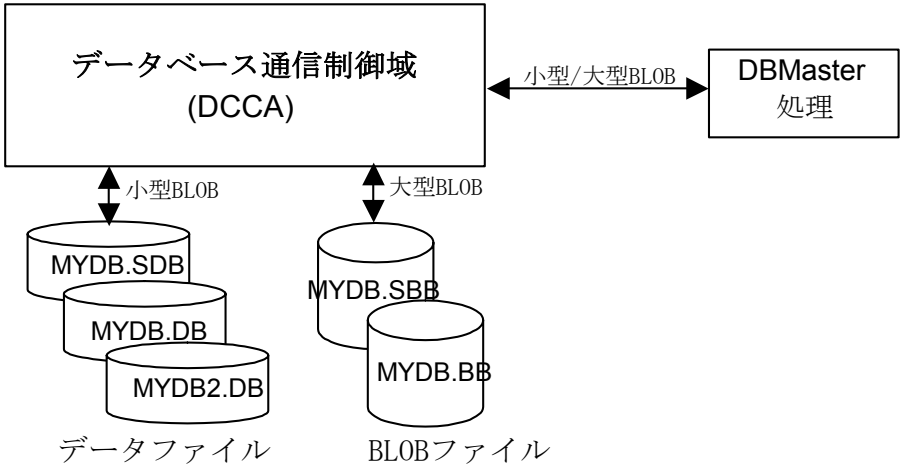


図 7-2 : DCCA を経由した DBMaster の BLOB データへのアクセス

データファイルはページで構成され、BLOB ファイルはフレームで構成されています。ページとフレームには、2 つの大きな違いがあります。

- ページのサイズは4KB,8 KB, 16KB ,32KBから選択できます。データーベースを作成する時またはフレームのサイズはカスタマイズするとき、キーワードDB_PGSIZでページのサイズを定義します。

[2]この値はそれぞれページサイズが 4KB の場合、3952 バイトとなり、ページサイズ 8KB のとき 16240 となります。16KB と 32KB のページサイズでは 32624 バイトです。

- 1ページに複数行のレコードを格納することができますが、1フレームには1つのBLOBしか入れることができません。

データベース作成時に BLOB ファイルのフレームサイズをカスタマイズして、パフォーマンスとディスク効率を改善することができます。フレームサイズをカスタマイズするためには、データベース作成時に、**dmconfig.ini** の**DB_BfrSz** キーワードにサイズをキロバイトで指定します。**DB_BfrSz** の初期値は 32 です。データベース作成時に設定する環境設定パラメータについての詳細は、4.2節の「データベースを作成する」を参照して下さい。

➲ **例 1**

dmconfig.ini ファイルの **DB_BfrSz** キーワードに、BLOB フレームサイズを指定する：

```
DB_BFRSZ = 16                          ; BLOB フレームサイズ = 16K バイト
```

DB_BfrSz は、8～256 の範囲内に指定することができます。

データベース内の BLOB ファイルは、全て同じフレームサイズです。データベース作成後に、BLOB フレームサイズを変更することはできません。DBMaster は、データベースのシステム表にフレームサイズを残しています。データベース起動時には、システム表からフレームサイズが参照され、dmconfig.ini にある **DB_BfrSz** キーワードは無視されます。

➲ **例 2**

システム表 SYSINFO にフレームサイズを問い合わせる：

```
dmSQL> SELECT INFO, VALUE FROM SYSINFO WHERE INFO = 'FRAME SIZE';

              INFO                             VALUE
============================== ==============================
FRAME_SIZE                     10240

1 rows selected
```

フレームサイズは、パフォーマンスとディスク使用容量の間のトレードオフで決定します。BLOB 全体を頻繁に検索する場合、フレームサイズを BLOB 全体が入るサイズに調節すると、ディスクアクセスは一回で済むのでパフォーマンスは良くなります。しかしながら、BLOB データのサイズには、大

きなばらつきがあるかもしれません。最大の BLOB に合わせてフレームサイズを決定すると、小さい BLOB が入っているフレームの使用されないディスク部分が無駄になります。

逆に、最小の BLOB に合せてフレームサイズを決めると、大きい BLOB は複数のフレームから取り出さなければならないためにパフォーマンスが低下します。

フレームには、フレーム情報を記録するヘッダ部分があります。例えば、フレームサイズを 8KB にしても、BLOB データが使用できる容量は 8192 バイトより小さくなります。さらに、BLOB の最初のフレームは、各 BLOB 情報（他のフレーム場所等）の記録用に約 1.8KB を別に確保するので、使用できる容量が一層小さくなります。従って、BLOB データの実際のサイズを 8192 バイトとすると、BLOB の最初の 6.2KB を最初のフレームに格納し、残りの BLOB を 2 番目のフレームに格納します。

1 グループはBE ページとNBE[3] PE ブロックで形成します。BEページは PAGE_SIZE KBデータページです。NPE[4]+1 フレームでPE ブロックを形成します。最初のフレームは、PAGE_SIZE[5]KBのPEページです。残りのNPEフレームはデータ用に使用され、**DB_BfrSz**で指定するサイズです。

---

[3] NBE は BE ページにコントロールされる PE ブロック数を定義します。それぞれ 4KB のページサイズでは 2004、8KB では 2026、16KB では 2716、32KB では 2723 となります。

[4] NPE は PE ページにコントロールされるページ数を定義します。それぞれ 4KB のページサイズでは 165、8KB では 333、16KB では 671、32KB では 1347 となります。

[5]ページサイズは **dmconfig.ini** のキーワード DB_PGSIZ にて定義されます。

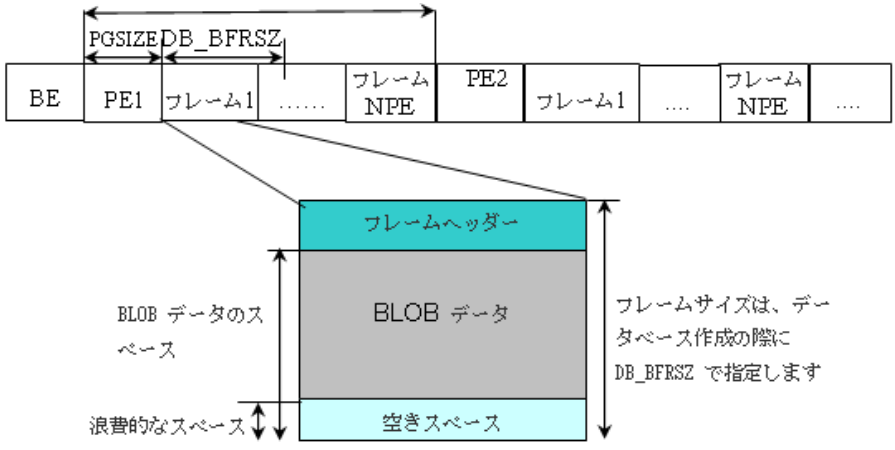


図 7-3 : BLOB ファイルの構造

フレーム総数、PEページ数、BE ページとデータフレームを基に、BLOB ファイルのサイズを計算することができます。

以下の公式は、近似の BLOB のサイズ(KB)の計算方法です。

BEページ数＝[**総**フレーム数/676685[6]]、([] means gain the nearest bigger integer)

PE ページ数＝[(総フレーム数–BE ページ数)/NPE+1]

BLOB ファイルサイズ = (BE ページ数 + PE ページ数) x PAGE SIZE KB + (フレーム総数 - BE ページ数 - PE ページ数) x DB_BFRSZ KB)

例えば、ページサイズは 8KB 、BLOB フレーム・サイズが 32KB の場合、3 フレームの BLOB ファイルのサイズは、次のようになります。

BE ページ数 =[3/676685]

= 1

[6]この値はページサイズに依存します。4 KB ページサイズでは 332665、8 KB ページサイズでは 676685、16 KB ページサイズでは 1825153、32 ページサイズでは 3670605 です。

PE ページ数 =[(3-1)/333+1]

= 1

BLOB ファイルサイズ = (1+1)*8+(3-1-1)*32

= 48 KB

DB_BbFil は、システム表領域、SYSTABLESPACE のシステム BLOB ファイル名を定義します。このファイル・サイズを指定することはできません。システム BLOB ファイルの初期設定ファイル名は、データベース名+'.SBB' です。

DB_UsrBb は、初期設定表領域、DEFTABLESPACE の初期設定ユーザーBLOB ファイル名とそのサイズを定義します。

既存のユーザー表領域に新しい BLOB ファイルを追加する方法についての詳細は、5.3節の「表領域にファイルを追加する」のセクションを参照して下さい。

BLOBを生成する

BLOB カラムは、データ型が LONG VARCHAR または LONG VARBINARY である点を除けば、他のカラムと同じです。

➲ 例

2 つの BLOB カラム ***note*** と ***photo*** を作成する：

```
dmSQL> CREATE TABLE tb_staff (id INTEGER, note LONG VARCHAR, photo LONG VARBINARY);
```

ホスト変数を使って ab.txt ファイルと画像ファイル img001.gif から BLOB データを挿入します：

```
dmSQL> INSERT INTO tb staff VALUES(2,?,?);
dmSQL/Val> &ab.txt, &img001.gif(2,4);
dmSQL/Val> END;
```

LONG VARBINARY カラムの内容は、16 進数で表示されます。表をブラウズすると、次の結果が得られます：

```
dmSQL> SELECT * FROM tb_staff;
 id           note              photo
=====      ==========     ================
```

```
   2       <script lan     ffd8ffe000104a464
```

ユーザーの指定したファイルに BLOB データを取り出すこともできます。BLOB データの挿入と取り出し方法については、*「JDBA Tool ユーザーガイド」*、及び*「ODBC プログラマーガイド」*を参照してください。

BLOBを更新する

BLOB は、常に全体をディスクに書き出します。BLOB カラムを更新すると、まず元の BLOB が削除され、それから新規 BLOB として新しいデータが挿入されます。

➲ **例**

UPDATE 文を使って、BLOB カラムの内容を更新する：

```
dmSQL> UPDATE tb_staff SET note = 'Hello !' WHERE id > 0;
dmSQL> SELECT * FROM tb_staff;
 id           note               photo
=====      ==========     ================
   1       Hello !         31323334353637
   2       Hello !         33343536
```

ユーザーから見れば、各行に必ず BLOB データがあります。しかし、DBMaster は ID > 0 の全ての行で共通する BLOB データを 1 つのみ作成し、行間でそのデータを共有してディスクを節約します。DBMaster には、1 つの BLOB を参照している行数を記録する内部カウンタがあります。共有の BLOB にリンクしている行の BLOB カラムを更新すると、新しい BLOB が生成され、共有 BLOB のカウンタが 1 つ減ります。これにより、BLOB カラムに施した修正が他の行に影響するのを防ぎます。これを疎結合と呼びます。疎結合はディスク使用を効率よくしますが、BLOB は同じ表にある行同士でしか共有することができません。どの行からもリンクされなくなった BLOB は、自動的に削除されます。

BLOBカラムの述語演算

BLOB は、NULL 値かどうかをチェックする CONTAIN、MATCH、条件式でのみ使用することができます。

➲ 例 1

表 *tb_staff* からカラム *note* が非 NULL である全データを取り出す：

```
dmSQL> SELECT * FROM tb_staff WHERE note IS NOT NULL;
```

DBMaster は、BLOB にもパターンマッチを使用することができます。CONTAIN と MATCH は、ワイルドカード文字を使用できない点を除くと、LIKE 関数と同じです。CONTAIN と MATCH の違いは、前者が部分文字一致に対し、後者が完全文字一致です。

➲ 例 2

notes カラムに「Database Administrator」という句を含む全従業員を見つける：

```
dmSQL> SELECT * FROM tb_staff WHERE note MATCH 'Database Administrator';
```

7.2 ファイルオブジェクト管理

ファイルオブジェクト（FO）は、外部ファイルとして格納され、他のアプリケーションからも直接ファイルにアクセスすることができます。そのため、データを他のアプリケーションでも使用するときに、このタイプを利用すると便利です。現在、ほとんどのマルチメディア・ツールは、そのツールで指定した完全な形式のファイルしか、マルチメディア・データを扱うことができません。マルチメディア・データを BLOB ファイルやデータファイルに格納すると、DBMaster からデータを取り出してから、対応ツールで処理できるようなファイルに再構成することが必要です。一方、マルチメディア・データを FO として格納すると、DBMaster からファイル名を取得し、対応するマルチメディア・ツールにそのファイル名を渡すだけです。

FO には、システム FO とユーザーFO の 2 種類があります。ユーザーがクライアント側でデータを挿入し、DBMaster を通じて、**dmconfig.ini** ファイルの DB_FoDir で指定した外部ファイルに保存される際に、システム・ファイルオブジェクトは生成されます。システム・ファイルオブジェクトは、DBMaster によって作成され、顧客の源の拡張子が付けられます。ユーザー・ファイルオブジェクトは、単にデータベースのカラムにリンクされた既存のファイルです。サーバーのオペレーティング・システムを通じて、DBMaster がアクセスできる端末にあるいずれかのファイルです。

システム FO およびユーザーFO の大きな違いは、システム FO が DBMaster によって自動的に生成されるということです。システム FO は、参照するカラムが無くなると削除されます。つまり、ユーザーはシステム FO のストレージを管理する必要がありません。システム FO のもう 1 つの利点は、データがサーバー側にコピーされるので、ユーザーがサーバー側からデータを管理できることです。DBMaster のバックアップとリストア機能は、システム FO も保護します。

これに対して、ユーザーFO は参照カラムが無くなっても削除されません。ユーザーFO の最大の利点は、DBMaster データベースのカラムを既存のファイルに直接リンクすることができることです。CD-ROM のファイルのようなデータをコピーする必要はありません。これによって、ディスクを大幅に節約することができ、複数のレコード間でファイルを容易に共有することができるようになります、但し、ファイルが DBMaster の外で削除された場合、このファイルにリンクしている全カラムは無効になります。ユーザーFO としてリンクされるファイルは、読み込み可能にしておきます。

ユーザーFO は、データベースからアクセス可能である必要があります。それらのファイルはサーバー側の多くのディレクトリに分散されることができます。ユーザーFO ディレクトリを指定する替わりに、**dmconfig.ini** ファイルの DB_UsrFO キーワードを 1 に指定し、ユーザーFO を使用するようにできます。ユーザーFO は初期設定では使用できません。

組み込み関数 **filename()**と **filelen()**を使用して、FO のファイル名とファイルサイズを取得することができます。

ファイルオブジェクトのパスをカスタマイズする

DBMaster はシステム FO を保存するために、複数のファイルオブエジェクトのサブディレクトリを生成します。これらのサブディレクトリは、**dmconfig.ini** ファイルのキーワード **DB_FoDir** で指定したファイルオブジェクトのディレクトリの中に存在します。サブディレクトリにあるファイルオブジェクトの数がしきい値に達した時、新しいサブディレクトリが生成されます。しきい値は、**dmconfig.ini** ファイルのキーワード **DB_FoSub** で指定できます。有効値は、100～10,000 です。

ファイルオブジェクト名のフォーマットは ZZxxxxxx.ext で、xxxxxx は 6 桁ベースの 36 シリアル番号です。ext は、ファイルオブエジェクトの拡張子です。ファイルオブエジェクトの拡張子は、SET EXTNAME コマンドに依存します。詳細は、「システム・ファイルオブジェクトの拡張子名」を参照ください。

サブディレクトリのネーミング規則は、サブディレクトリにある最初のファイルオブジェクトの名称に基づく規則に従います。フォーマットは SUBxxxxxx で、xxxxxx はディレクトリに最初に挿入されたファイルオブジェクトの 6 桁ベースの 36 シリアル番号です。

ファイルオブジェクトの管理を単純化させるために、複数のデータベースで 1 つの FO ディレクトリを共有することも可能ですが、データベースのバックアップの際に不便になるので、理想的ではありません。ファイルオブエジェクトのパスは、データベースの起動前、或いはランタイム時に環境設定パラメータを修正することで変更することができます。

ファイルオブジェクトのパスをオフラインで設定する

システム FO を格納する場所は、**dmconfig.ini** ファイルの **DB_FoDir** で指定することができます。**DB_FoDir** は、既存ディレクトリの絶対パスです。また、ディレクトリの書き込み許可が DBMaster に与えられている必要があります。

➲ **例 1**

/disk1/usr/fo ディレクトリに生成されたシステム FO を配置する時は、次の文を dmconfig.ini に指定します：

```
DB_FODIR = /disk1/usr/fo
```

➲ **例 2**

ユーザー・ファイルオブジェクトを使用可能にする：

```
DB_USRFO = 1        ; ユーザー・ファイルオブジェクトを使用可能にする
```

ファイルオブジェクトのパスをオンラインで設定する

DBMaster には、データベース起動時にシステム・ファイルオブジェクトのディレクトリを修正するための体系的な手順があります。この操作によって、以下の 3 つの要素の設定を新しい値に変更します。

- **ランタイムFOディレクトリ** — 変更を行った後、システム・ファイルオブジェクトは、全て新しいFOディレクトリに保存されます。
- **DB_FoDir** —データベースを再起動すると、新しいFOディレクトリを使用し始めます。
- **$DB_FODIR 別名** —FOの別名の初期設定。**dmconfig.ini**ファイルのキーワード**DB_FoDir**の設定に対応しています。

➲ **例 1**

次のコマンドを実行し、FO ディレクトリを**/home/DBMaster/mydb/fo** に変更する：

```
dmSQL> call SETSYSTEMOPTION('fodir', '/home/DBMaster/mydb/fo');
```

ファイルオブジェクトのディレクトリの変更に加え、FO ディレクトリのパスの現在の設定をシステムに問い合わせることができます。

➲ **例 2**

現在の FO ディレクトリの設定が戻ります：

```
dmSQL> call GETSYSTEMOPTION('fodir', ?);
OPTION_VALUE: /home/DBMaster/mydb/fo
```

ファイルオブジェクトを生成する

DBMaster でファイルオブジェクトを生成するためには、いくつかのステップが必要です。まず、表に FILE データ型のカラムを作成します。システム FO とユーザーFO のいずれも、FILE データ型のカラムに挿入されます。

➲ **例 1**

ファイルオブジェクトのカラム ***photo*** のある表 ***person*** を作成する：

```
dmSQL> CREATE TABLE person (name CHAR(10), photo FILE);
```

➲ 例 2

サーバー側で FO データが挿入されると、既存のファイルが FO カラムにリンクされ、ユーザーFO が生成されます。クライアント側で FO データが挿入されると、そのファイルをクライアント側からサーバー側の FO ディレクトリにコピーすることでシステム FO が生成します。

```
dmSQL> INSERT INTO person VALUES ('cathy','/disk1/image/cathy.bmp')
    2>;   // ユーザーFO として格納
dmSQL> INSERT INTO person VALUES ('jeff',?);
dmSQL/Val> &jeff.gif;  // システム FO として格納
dmSQL/Val> END;
```

➲ 例 3

FO カラムの要素、ファイル名、ファイルサイズを取り出すことができます。FO カラム **cathy.bmp** を取り出す：

```
dmSQL> SELECT photo, FILENAME(photo), FILELEN(photo) FROM person ;
  photo          filename(photo)         filelen(photo)
=========    ======================     ==============
012034451      /disk1/image/cathy.bmp        21100
349045821      /disk1/usr/fo/ZZ000000.bmp    12034
```

FO カラムのより詳細な操作法については、*「JDBA Tool ユーザーガイド」*と*「ODBC プログラマーガイド」*を参照してください。

システム・ファイルオブジェクトの拡張子名

SET EXTNAME 文を使って、システム FO の拡張子名を指定することができます。

➲ 例 1

システム FO の拡張子名をセットする：

```
dmSQL> SET EXTNAME TO <拡張子名>;
```

拡張子名には、次の 2 種類を指定することができます。

- 'bmp'、'avi'、'jpg'のような7文字を超えない文字列。
- SOURCEオプションを使うと、クライアントのソースファイルと同じ拡張子になります。

➲ 例 2

SET EXTNAME コマンドを使う：

```
dmSQL> CREATE TABLE tb_example (c1 INT, f1 FILE);
dmSQL> INSERT INTO tb_example (c1, f1) VALUES (?, ?);
dmSQL/Val> SET EXTNAME TO FOB;
dmSQL/Val> 1, &readme.txt;              //拡張子名 : '.FOB'
1 rows inserted1
dmSQL/Val> SET EXTNAME TO doc;
dmSQL/Val> 2, &readme.txt;              //拡張子名 : '.doc'
1 rows inserted
dmSQL/Val> SET EXTNAME TO SOURCE;
dmSQL/Val> 3, &readme.txt;              //拡張子名 : '.txt'
dmSQL/Val> END;
dmSQL> SELECT c1, FILENAME(f1) FROM tb example;
        c1                                  filename(f1)
================= ==================================================
        1                             /usr1/fo/ZZ000001.FOB
        2                             /usr1/fo/ZZ000002.doc
        3                             /usr1/fo/ZZ000003.txt
3 rows selected
```

ファイルオブジェクトを更新する

UPDATE 文を使って、FO カラムの内容を更新します。FO カラムは新しいファイルに置き換えられます。

FO の挿入と同様、FO カラムが新しいシステム FO を更新、或いはユーザーFO にリンクされます。

➲ 例

photo カラムを既存のファイル */disk2/image/common.bmp* にリンクさせる：

```
dmSQL> UPDATE tb_person SET photo = '/disk2/image/common.bmp' WHERE name = 'cathy ';
```

替わりに、クライアント側のファイルから新しいデータを入力することもできます。詳細については、*「JDBA Tool ユーザーガイド」*と*「ODBC プログラマーガイド」*を参照してください。

UPDATE の結果が複数行になる場合でも、ディスクを節約するために、ファイルは 1 つしか生成されず、行同士で共有します。DBMaster 内部に、ファイルを参照する行数を記録するカウンタがあります。外部のアプリケーション・プログラムでファイルの内容が修正された場合、全ての行がその修正を認識します。

UPDATE や DELETE 操作の結果、システム FO にリンクする行が無くなると、トランザクションのコミット後に自動的にその FO は削除されます。但し、ユーザーFO は DBMaster が生成したものではないので、参照する行がなくなっても削除されることはありません。

ファイルオブジェクト名を変更する

ディスクが一杯になる、ディスクを再作成する等の理由で、FO のディレクトリや名前の変更が必要になることがあります。MOVE FILE OBJECT 文で、FO の名前やパスを変更することができます。但し、この文を使用する前に、実際のファイルを新しいディレクトリに移動しておきます。DBMaster は、移動する場所にファイルが存在するかどうかを確認します。

➲ **例 1**

FO のディレクトリを変更する：

```
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/ZZ000000.FOB' TO '/disk3/pub/photo1.bmp';
```

複数の FO を別のディレクトリに移動することもできます。移動元のファイル名を指定するために、「*」文字を 1 つ使うことができますが、移動先ディレクトリには使用できません。DBMaster は、再帰的なファイル移動をサポートしません。サブディレクトリ以外の全てのファイルを別のディレクトリに移動するには、移動元ディレクトリの最後に「/」または「/*」文字を追記して指定します。

➲ **例 2**

/disk1/usr/fo に 4 つのファイル *ABC1.fob*、*ABC2.fob*、*ABC3.fob*、*ABC4.fob* がある場合、これらを */disk3/pub* に移動する文は下記のいずれかです：

```
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/ ' TO '/disk3/pub/ ';
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/* ' TO '/disk3/pub/ ';
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/*.FOB ' TO '/disk3/pub/ ';
```

```
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/A* ' TO '/disk3/pub/ ';
```

➲ **例 3**

ABC1.fob を***/disk1/usr/fo*** から***/disk3/pub*** へ移動する文は下記のいずれかです：

```
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/*1.FOB ' TO '/disk3/pub/ ';
dmSQL> MOVE FILE OBJECT '/disk1/usr/fo/A*1.FOB ' TO '/disk3/pub/ ';
```

ファイルオブジェクトの長さの取得

組み込み関数 FILELENEX と FILELEN にて FO の長さを取得できます。詳細は*「SQL 文と関数参照編」*をご参考ください。

ファイルオブジェクトの述語演算

BLOB と同様、FO が NULL かチェックすることができます。CONTAIN 関数と MATCH 関数を使用して、パターン検索を行うこともできます。更に、組み込み関数 filelen()を用いて算術式や、組み込み関数 fileexist()を使った条件式、更に組み込み関数 filename()を用いて文字列式で FO を使用することができます。

オペレーティング・システムでファイルが削除されたり、名前が変更したりした場合、組み込み関数 fileexist()でそのファイルの存在をチェックすることができます。値が 0 の場合は、FO ファイルが存在しないことを意味し、1 はまだ存在することを意味し、NULL はその行が NULL 値であることを意味します。

➲ **例 1**

photo カラムのファイル名が***.gif*** の拡張子である行を検索する：

```
dmSQL> SELECT * FROM tb_person WHERE FILENAME(photo) LIKE '%.gif';
```

➲ **例 2**

photo カラムのサイズが 100KB より大きい行を取り出す：

```
dmSQL> SELECT * FROM tb_person WHERE FILELEN(photo) > 102400;
```

➲ 例 3

既存ファイルの行を取り出す：

```
dmSQL> SELECT * FROM tb_person WHERE FILEEXIST(photo)=1;
```

ファイルオブジェクトUNC名

Microsoft Windows 環境のファイルオブジェクトのパスとディレクトリ名に Universal Naming Convention(UNC)ファイル名を使うことができます。これにより、Microsoft Windows のプラットフォーム上で DBMaster のサーバーを運用する際、パスとディレクトリ名を簡単に指定することができます。

➲ 例 1

dmconfig.ini ファイルにシステム FO のディレクトリを指定する際に、UNC 名**\\ntmachine\e\fo** を使う：

```
DB_FODIR = \\NTMACHINE\E\FO
```

➲ 例 2

ファイルオブジェクトを指定するために UNC 名を使う：

```
dmSQL> CREATE TABLE tb example (c1 INT, c2 FILE);
dmSQL> INSERT INTO tb example VALUES (?, ?);
dmSQL/Val> 1, '\\NTMACHINE\D\DB\memo1.txt';
1 rows inserted
dmSQL/Val> 2, &c:\temp\memo2.txt;
1 rows inserted
dmSQL/Val> END;

dmSQL> SELECT c1, FILENAME(c2) FROM t1;
   c1                        FILENAME(c2)
======== ================================================
    1            \\NTMACHINE\D\DB\memo1.txt
    2            \\NTMACHINE\E\FO\ZZ000001.txt

2 rows selected
```

ファイルオブジェクト・パスの初期設定別名

DBMaster では、**dmconfig.ini** ファイルのキーワード **DB_DbDir** と **DB_FoDir** に指定したディレクトリを基に、$DB_DBDIR と$DB_FODIR の 2 つの別名を使うことができます。これらの別名は、ファイルオブジェクトのパスに実際のパスを指定する替わりに利用します。別名パスを通じて、挿入/更新/削除を行うことができます。別名を使うことで、ファイルオブジェクト操作に使用するパスを簡単に指定することができます。

dmconfig.ini ファイルのキーワード **DB_DbDir** と **DB_FoDir** が、それぞれ別名$DB_DBDIR と$DB_FODIR を意味します。

➲ **例**

dmconfig.ini ファイルのキーワード **DB_FoDir** に設定されているパス：

```
DB_FODIR = "/usr1/cctsai/tmp/employeedata/FO"
```

ファイルオブジェクト・パスの別名を使って、FILE データ型のカラム **file** に値を挿入する：

```
dmSQL> create table tb_example (c1 INT ,file FILE);
dmSQL> insert into tb_example values (2, '$DB_FODIR/PHOTO471.JPG');
```

上記の例では、挿入されるファイル **PHOTO471.JPG** は、実際には絶対パスの/usr1/cctsai/tmp/employeedata/FO/にあるファイル PHOTO471.JPG にリンクされます。

ファイルオブジェクトとアプリケーション

ODBC は、DBMaster でのみサポートされている FILE データ型を認識しません。Inprise/Borland Delphi や Microsoft Visual Basic のような開発ツールは、FILE データ型を使っていません。FILE データ型のデータにアクセスするために、これらのツールを使う場合、**DB_FoTyp** キーワードを設定して、FILE データ型を変換する必要があります。**DB_FoTyp** を 1 にセットすると、FILE データは、LONG VARBINARY データ型に変換されます。**DB_FoTyp** が 0 の場合、他のデータ型に変換されませんので、ツールで FILE データ型を使うことはできません。

➲ 例

FILE データ型から LONG VARBINARY に変換する：

```
DB_FOTYP = 1
```

7.3 ラージオブジェクトのジャーナル

BLOB（LONG VARCHAR や LONG VARBINARY）のデータのログは、大量のディスク領域を必要とし、パフォーマンスの低下をもたらします。DBMaster では、指定した表領域の BLOB のログを取るかどうかを選択することができます。DBMaster では、ファイルオブジェクト(FILE)のログをとることはできません。

初期設定では、BLOB データの内容のログは取られませんが、データベースの起動から終了までの間、BLOB データの整合性は保たれます。システム障害が発生した場合でも、BLOB データはリカバリ後に矛盾の無い状態になります。

但し、増分バックアップからデータベースをリカバリする際、BLOB データの整合性は保証されません。BLOB データをジャーナルに記録するためには 2 段階のステップがあります。まず、**dmconfig.ini** ファイルのパラメータ **DB_BMode** を、BLOB のトランザクションを記録するようにセットします。次に、バックアップする BLOB データがある表領域は、BACKUP BLOB ON オプションで作成されていることを確認します。

BLOBジャーナルのログを取る

BLOB のログを取得するために次の 2 つの前提条件があります。

- **dmconfig.ini**ファイルのDB_BModeキーワードの値が、2（データとBLOBをバックアップする）にセットされていること
- ログを取るBLOBファイルが、BACKUP BLOB ONオプションで作成した表領域に存在すること

DB_BMODEの値を設定する

キーワード DB_BMode は、データベースのバックアップ・モードを指定します。値を 0 にセットすると NON-BACKUP モードになり、1 は BACKUP-DATA モードになり、2 は BACKUP-DATA-AND-BLOB モードになります。

- **NON-BACKUP (0)** モード:増分バックアップをしません。
- **BACKUP-DATA (1) モード**: システム表領域とユーザー表領域にあるデータの増分バックアップをしますが、ユーザー表領域のBLOBの増分バックアップはしません。
- **BACKUP-DATA AND-BLOB (2) モード**: システム表領域とユーザー表領域にあるデータと、BACKUP BLOB ONフラグで作成したユーザー表領域のBLOBの増分バックアップをします。但し、BACKUP BLOB OFFフラグで作成したユーザー表領域にあるBLOBの増分バックアップをしません。

BLOB のログを取る場合は、**dmconfig.ini** ファイルに次の 1 行を追加します。

```
DB_BMODE = 2    ;BLOB を含む全データのログを取る
```

データベースのバックアップ・モードの詳細については、15 章の「リカバリ、バックアップ、リストア」を参照して下さい。

CREATE TABLESPACEバックアップのオプションを設定する

個々の表領域のバックアップ・モードは、その作成時に設定します。CREATE TABLESPACE 文のコマンドは次のとおりです。

```
CREATE [AUTOEXTEND] TABLESPACE tablespace_name [backup_mode]
DATAFILE [tsfile , tsfile, ...];
```

そのうち、

```
backup mode ::= BACKUP BLOB OFF | BACKUP BLOB ON
tsfile ::= file_name TYPE = DATA | file_name TYPE = BLOB
```

BACKUP BLOB ON フラグで作成した表領域に、重要な BLOB を置くことができますし、システムのパフォーマンスを優先して、BACKUP BLOB OFF フラグで作成した紛失する可能性がある表領域に BLOB を置くことも重要なことです。表領域を作成する際には、その優先順位を考慮する必要があります。

表領域を作成する前に、**dmconfig.ini** ファイルにてデータ及び BLOB ファイルを指定する必要です。JConfiguration Tool での**ユーザーファイルタブ**によって実現できます。データ及び BLOB ファイル作成の詳細については、*「JConfiguration Tool 参照編」*をご参照ください。

➲ **例 1**

BACKUP BLOB OFF で表領域 **ts_reg** を作成し、BACKUP BLOB ON で **ts_aut** を作成する：

```
dmSQL> CREATE TABLESPACE ts_reg BACKUP BLOB OFF
            2> DATAFILE f1 TYPE = DATA, f2 TYPE = BLOB;
dmSQL> CREATE TABLESPACE ts_aut BACKUP BLOB ON
            2> DATAFILE f3 TYPE = DATA, f4 TYPE = BLOB;
```

➲ **例 2**

システム表 SYSTABLESPACE の BK_MODE から各表領域のバックアップ・モードを取得することができます。値が 1 の場合は BACKUP BLOB が OFF、2 は ON を意味します。表領域のバックアップ・モードを問い合せる：

```
dmSQL> SELECT TS NAME, BK MODE FROM SYSTEM.SYSTABLESPACE;
     TS NAME        BK MODE
==============  ===========
SYSTABLESPACE            2
DEFTABLESPACE            2
ts_reg                   1
ts_aut                   2
4 rows selected
```

データベースと表領域のバックアップ・モードの相互関係は次のとおりです。

バックアップ・モード	表領域バックアップ・モード	ユーザー定義表領域(データ)	ユーザー定義表領域 (BLOB)	システム表領域(データとBLOB)
NON BACKUP (DB_BMODE=0)	---	バックアップしない	バックアップしない	バックアップしない
BACKUP DATA (DB_BMODE=1)	---	バックアップする	バックアップしない	バックアップする
BACKUP DATA AND BLOB (DB_BMODE=2)	BACKUP BLOB OFF	バックアップする	バックアップしない	バックアップする
	BACKUP BLOB ON	バックアップする	バックアップする	バックアップする

ジャーナルに BLOB データを記録する場合、ジャーナル・ファイルが充分な大きさかどうかチェックしておきます。サイズが十分でない場合は、ジャーナルが既に一杯である旨を伝えるメッセージを受け取ります。ジャーナルファイルのサイズの調整方法については、5章の「ジャーナルファイルのサイズを変更する」を参照して下さい。

データファイル、BLOB ファイル、表領域についての概念は、5章の「ストレージアーキテクチャ」を参照して下さい。

CREATE TABLESPACE コマンドに関する事項は、*「SQL 文と関数参照編」*をご覧下さい。

SYSTABLESPACE 表に関する詳細は、システムカタログ参照の *システム・カタログ*を参照して下さい。

ファイルオブジェクトのジャーナルのログを取る

DBMaster は、ファイルオブジェクト(FO)のジャーナル・ログ取得をサポートしていません。データベースをバックアップする際、データベースにある全てのファイルオブジェクトのバックアップも必ず行って下さい。

➲ 例

システム表 SYSFILEOBJ から全 FO のファイル名を取得する：

```
dmSQL> SELECT FILE_NAME FROM SYSFILEOBJ;
```

全 FO をバックアップ・ストレージにコピーして下さい。バックアップからデータベースをリストアする際、同様に FO もコピーして下さい。ファイルのパスや、ファイル名を変更した場合、SYSFILEOBJ 表にファイル名を更新するため、MOVE FILE OBJECT コマンドを使って下さい。

7.4 ラージオブジェクトと SELECT INTO 文

SELECT INTO 文は、選択したデータを指定した表に挿入します。この命令文を使うと、BLOB とファイルオブジェクトを 1 つの表から別の表に移動することができます。分散型データベース(DDB)環境で使用することができます。

ローカルからローカルの SELECT INTO 文では、BLOB データのコピーを作るか、或いはシステム FO/ユーザーFO の共有カウンタを増やす必要があります。

DDB 環境では、BLOB データをある場所から別の場所にコピーしますが、ファイル・オブジェクトの場合には考慮の余地があります。DBMaster では、DDB 環境でファイル・オブジェクト処理を扱うために、分散ファイル・オブジェクト複製モード（SET DFO DUPMODE コマンド）を使います。

SET DFO DUPMODE

DFO DUPMODE は、ファイル・オブジェクトをターゲット・データベースにコピーするかどうかをデータベースに伝えます。DFO DUPMODE には、NULL モードと COPY モードの 2 種類あります。

➲ 例

DFO DUPMODE の NULL モードと COPY モードの構文：

```
dmSQL> SET DFO DUPMODE NULL;
dmSQL> SET DFO DUPMODE COPY;
```

SET DFO DUPMODE NULL

DDB モードでは、2 種類のケースが考えられます。

- ソース・データベースとターゲット・データベースが同じで、ローカル・データベース又は両方のリモート・データベースがあります。それらは同じデータベースなので、DBMasterはファイル・オブジェクトの共有カウンターを増やすだけです。
- ソース・データベースとターゲット・データベースが異なる場合。ターゲットのFILEカラムをNULLにセットします。このように、ソース・データベースのファイル・オブジェクトは、送信されません。

SET DFO DUPMODE COPY

ファイルオブジェクトには 3 種類のケースが考えられます。

- ユーザーFOでは、DBMasterは、ソース・ファイル名のみターゲット・データベースに渡します。ユーザーは、ターゲット・データベースがそれらにアクセスできる場所にファイルをコピーする必要があります。新規ディレクトリがソース・データベースと異なる場合、UPDATE文かMOVE FILE OBJECT文でターゲット・データベースにあるファイル名を変更します。
- ２種類のデータベース間のシステムFOの場合、DBMasterは、ターゲット・データベースに新規システムFOを作成し、ソース・データベースの内容をそれにコピーします。
- 同じデータベースにあるシステムFOの場合、ローカルからローカル、又はリモートからリモートに、DBMasterは共有カウンタを増やすだけです。

制限

DFO DUPMODE モードは、BLOB(LONG VARCHAR と LONG VARBINARY) で実行している SELECT INTO に影響を与えません。常に、BLOB データ型を DDB 環境若しくは通常の環境にコピーします。

DDB 環境では、SELECT INTO コマンドがユーザーFO で実行される場合、DFO DUPMODE のオプションはコピーです。ユーザーは、ターゲット・データベースにあるリンク・ファイルの位置を把握しておく必要があります。ソースとターゲットのファイル・パスが異なる場合は、ファイルをソースからターゲットにコピーし、これらのカラムに UPDATE か MOVE FILE OBJECT コマンドを実行します。

ユーザーが上記演算子を実行しない場合は、ファイルオブジェクトを問合せた際に、ファイルが完全なパスに存在しない、又はファイルのパスが正しくないため、エラー・メッセージが返されます。

システム FO をリモート・データベースからローカル・データベースに移すする際、DBMaster は共有情報の記録を残します。この情報は、一つの SELECT INTO 内に保管されます。それゆえ、ファイルの重複によりスペースを浪費するという問題が存在します。加えて、リモート・データベースへのシステム FO の移動は、重複するファイルを生成します。

SET EXTNAME オプションは、SELECT INTO コマンドの結果に影響しません。ソースとターゲット・データベースにあるファイルオブジェクトの拡張子は同じです。例えば、ソース・データベースのファイル名は 'ZZ000001.BMP'であれば、ターゲットのファイルオブジェクトのターゲット名は'ZZXXXXXX.BMP'のようになります。

➲ 例

DBMaster は、CHAR、VARCHAR、BINARY のようなデータをファイル名としてみなします。そのため、ユーザーは **db2** が **db1** のビューから'/etc/hosts' ファイルにアクセスできるようする必要があります。**db2:t2.c2** が FILE データ型の FILE データで CHAR データを参照する。

```
dmSQL> SELECT c1, '/etc/hosts' FROM db1:t1 INTO db2:t2(c1, c2);
```

以下の表は、ターゲット・データベースの FILE データ型のカラムを考慮して、異なるソースのデータ型での SELECT INTO コマンドの影響を要約しています:

ソース DB 種類	環境	SET DFO DUPMODE	結果
文字列式 CHAR VARCHAR BINARY	非 DDB か DDB 環境	…	ソース:ファイル名を渡す ターゲット:新規ユーザー・ファイル・オブジェクトを挿入する

ソース DB 種類	環境	SET DFO DUPMODE	結果
FILE	ソースとターゲットが同一のデータベース	…	ファイル・オブジェクトの共有カウンタを増やす
	ソースとターゲット・データベースが異なる	NULL	ターゲット: NULL 値を挿入する
		COPY	ソースがユーザーFO オブジェクトの場合: ソース:ファイル名を渡す ターゲット:新規ユーザーFO を挿入する ソースがシステム FO の場合: ソース:FO の内容を渡す ターゲット:新規システム FO オブジェクトを挿入する
LONG VARCHAR LONG VARBINARY その他	…	…	サポートしていません

8 セキュリティ管理

本章では、DBMaster データベースのセキュリティ方針を立案するためのガイドラインを解説します。データベースのセキュリティの設定、ユーザーの権限レベルの設定、表権限の設定に関する情報も説明します。

8.1 セキュリティ方針

DBMaster には 2 種類のセキュリティがあります。

- **データベース権限**—DBMasterにログインできるユーザーと実行可能なアクションを定めます。
- **オブジェクト権限**—表、カラム、ビュー、ドメイン、シノニム等のDBMasterオブジェクトへのアクセス権を管理します。.

8.2 データベース権限

データベース権限とは、誰にデータベースへのアクセスを認め、何を実行させるかを決定することです。DBMaster は、*ユーザー名*と*パスワード*でデータベースへのアクセスを管理します。DBMaster は表8-1に示すように、五つレベルのユーザー権限を持っています。

SYSADM 権限は DBMaster の最高レベルの権限です。各データベース内に、SYSADM 権限を持つユーザーが一つのみです。SYSADM は SYSDBA、DBA、RESOURCE または CONNECT 権限を他のユーザーに与えることができ、ほかのユーザーに ACL（アクセスコントロールリスト）を設定することもで

き、そしてデータベース内の SYSDBA 及び DBA と同じレベルの権限を持っています。

SYSDBA 権限を持つユーザーは、SYSADM または SYSDBA 以外の権限を持つユーザーに CONNECT、RESOURCE 及び DBA 権限を与えるまたは取り消すことができ、これらのユーザーのパスワードを変更することもでき、低権限ユーザーに ACL（アクセスコントロールリスト）を設定することもできます。また、SYSDBA 権限を持つユーザーは DBA 権限を持つユーザーのあらゆる権限を有しますが、そして、SYSADM のみ他ユーザーに SYSDBA 権限を与えるまたは取り消すことができます。SYSDBA 権限を持つユーザーに対して、SYSADM はその SYSDBA 権限を取り消す場合には、当該ユーザーは DBA 権限を持っていますが、DBA 権限を取り消す場合には、当該ユーザーは SYSDBA 権限及び DBA 権限を持っていません。

DBA 権限レベルは、データベースの全てのオブジェクトに対する全ての権限をもち、SYSADM、SYSDBA または DBA 権限を持つ以外の全ユーザーに対して、任意のオブジェクト権限を与え、変更し、取り消すことができます。DBA は、表領域やファイルのような新規リソースを作成し、データベースの起動/終了/バックアップ等のデータベース管理オペレーションを行うことができます。

RESOURCE（リソース）権限をもつユーザーは、新規表やビューを作成し、所有する表の権限を他のユーザーに与えることができます。

CONNECT（接続）権限もつユーザーのみ、権限が与えられたオブジェクトにアクセスすることはできますが、新規表やビューを作成することはできません。CONNECT ユーザーは、システム表の情報を検索することができます。

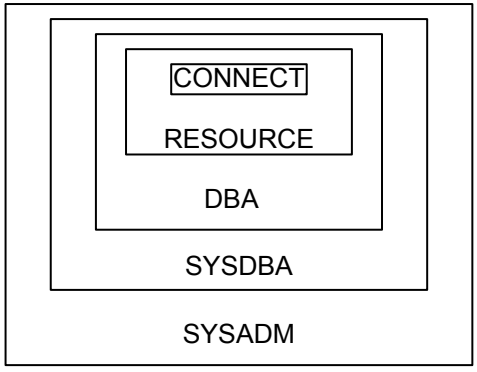


図 8-1：DBMaster データベースの権限レベルの構造

レベル	権限
SYSADM	全ユーザー（SYSADM 以外）にセキュリティ権限レベル（CONNECT/RESOURCE/DBA/SYSDBA）を与える/取り消す。 全ユーザーのパスワードを変更する。 SYSDBA 権限レベルの全権限を有する。
SYSDBA	全ユーザー（SYSADM 及び SYSDBA 権限を持つユーザー以外）にセキュリティ権限レベル（CONNECT/RESOURCE/DBA）を与える/取り消す。 全ユーザー（SYSADM 及び SYSDBA 権限を持つユーザー以外）のパスワードを変更する。 DBA 権限レベルの全権限を有する。
DBA	全ての表（SYSTEM 表以外）の全権限を有する。 全てのユーザー/グループのオブジェクト権限を付与/変更/取り消す。 ユーザーをグループに追加/削除する データベースの起動と終了、表領域の作成/削除/変更、データベースのバックアップ等のデータベース管理コマンドに対する実行権限を有する。 CONNECT と RESOURCE 権限レベルの全権限を有する。

RESOURCE	表、ビュー、ドメイン、シノニムを作成し、作成した表、ビュー、ドメイン、シノニムを削除する。 所有する表/ビュー権限を他ユーザーに与える/取消す。 与えられた任意の表権限を有する。 CONNECT 権限レベルの全権限を有する。
CONNECT	データベースにログインする。 SYSTEM 表を検索する。 与えられた任意の表権限を有する。 CONNECT 権限レベルは必ず最初に与えられる権限です。

表 8-1：DBMaster データベースの権限レベル

ユーザー管理

DBMaster には、ユーザーを管理するための種々の SQL 文があります。これらの SQL 文を使用して、新規ユーザーをデータベースに追加、既存のユーザーをデータベースから削除、ユーザーのパスワードを設定/変更、ユーザーに与えられている権限レベルの変更を行うことができます。

ユーザーを追加する

新しいユーザーが DBMaster にログインする前に、SYSADM は GRANT コマンドを使用して、ユーザーにユーザー名とパスワードを指定しなければなりません。

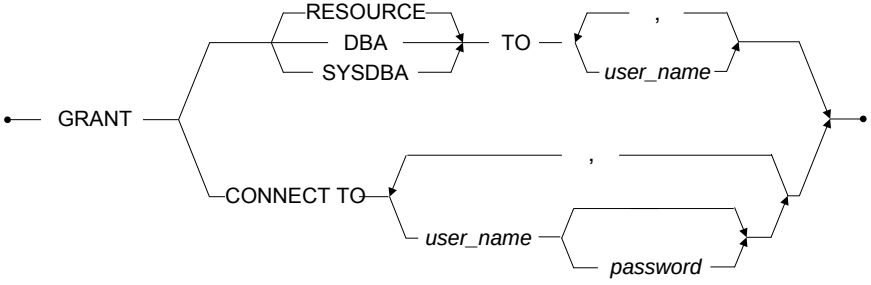


図 8-2：GRANT コマンドの構文

GRANT コマンドはユーザー権限の付与に用いられます。SYSADM のみその他のユーザーに権限を与えることができます。但し、SYSADM は自分の権限を他のユーザーに与えることができません。従いまして、各データベースに SYSADM ユーザー名及び SYSADM 権限を持つユーザーは一つのみです。SYSADM はデータベースを作成するときの初期設定のユーザーで、そのユーザー名 SYSADM を変更することができません。SYSADM のみそのパスワードが変更できます。

SYSADM は、CONNECT、RESOURCE、DBA、SYSDBA 及び ACL 権限を他のユーザーに与えることができます。GRANT コマンドによって他のユーザーに RESOURCE、DBA または SYSYDBA 権限を与える場合、その新しい権限は次回にデータベースに接続するときに有効になります。

SYSADM または SYSDBA 権限を持つユーザーは CONNECT 権限を持つユーザーにパスワードを与えることができます。パスワードを指定しない場合、ユーザーがデータベースにログインするときにパスワードを必要としないという意味になります。16 バイト以下の任意の SQL 識別子をパスワードにすることができます。

➲ **例 1**

ユーザー***Jeff***に CONNECT 権限を与え、パスワードを ***jeff123*** にする。

```
dmSQL> GRANT CONNECT TO Jeff jeff123;
```

➲ **例 2**

ユーザー***Jeff***の権限を RESOURCE レベルにアップする。

```
dmSQL> GRANT RESOURCE TO Jeff;
```

➲ **例 3**

ユーザー***Jeff***の権限を DBA レベルにアップする。

```
dmSQL> GRANT DBA TO Jeff;
```

➲ **例 4**

ユーザー***Jeff*** の権限を SYSDBA レベルにアップする。

```
dmSQL> GRANT SYSDBA TO Jeff;
```

パスワードを変更する

ALTER PASSWORD コマンドを使用して、ユーザーのパスワードを変更することができます。

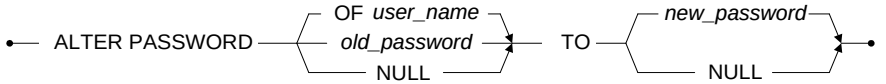


図 8-3 : ALTER PASSWORD コマンドの構文

下記の二つの方法でユーザーパスワードを変更することができます:

- ALTER PASSWORD <旧パスワード> TO <新パスワード> 文は、ユーザーが自分自身のパスワードを変更するときに使用します。<旧パスワード>は、データベースに格納されている元のパスワードと一致しなければなりません。
- ALTER PASSWORD OF <ユーザー名> TO <新パスワード>文は、SYSADMが任意のユーザーのパスワードを変更するときに使用します。旧パスワードは必要ありません。

➲ **例 1**

パスワードの無いユーザー***Jeff***が、パスワード ***xyz@#***を付ける。

```
dmSQL> ALTER PASSWORD NULL TO "xyz@#";
```

➲ **例 2**

SYSDBA がユーザー**Jeff** のパスワードを **abc@#**に変更する。

```
dmSQL> ALTER PASSWORD OF Jeff TO "abc@#";
```

➲ **例 3**

SYSADM が、ユーザー***Jeff***のパスワードを ***xyz@#***に変更する。

```
dmSQL> ALTER PASSWORD OF Jeff TO "xyz@#";
```

ユーザーの削除/ユーザー権限レベルの変更

REVOKE コマンドを使用して、データベースでのユーザー権限を取り消すことができます。

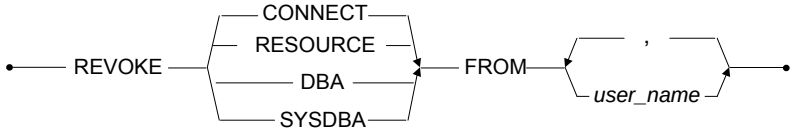


図 8-4 : REVOKE コマンドの構文

RESOURCE、DBA または SYSDBA 権限の取り消し動作は、次回にデータベースに接続する際に有効になります。

SYSADM は、SYSADM 以外のユーザーに CONNECT、RESOURCE、 DBA、SYSDBA 及び ACL（アクセスコントロールリスト）を取り消すことができます。

SYSDBA 権限を持つユーザーは、SYSDBA 以外のユーザーに CONNECT、RESOURCE、DBA 及び低権限ユーザーの ACL（アクセスコントロールリスト）を取り消すことができます。

➲ **例 1**

ユーザー***Jeff***の SYSDBA 権限を取り消す。

```
dmSQL> REVOKE SYSDBA FROM Jeff;
```

上記のコマンドを実行した後、ユーザー***Jeff***は SYSDBA 権限をもはや持っていませんが、DBA 権限があります。

➲ **例 2**

ユーザー***Jeff***の DBA 権限を取り消す。

```
dmSQL> REVOKE DBA FROM Jeff;
```

上記のコマンドを実行した後、ユーザー***Jeff***は DBA 権限をもはや持っていませんが、CONNECT 権限があります。

➲ **例 3**

Jeffの CONNECT 権限とログインする資格を取り消す：

```
dmSQL> REVOKE CONNECT FROM Jeff;
```

権限	説明
SYSDBA	ユーザーの SYSDBA 権限の取り消しは、当該ユーザーがその他のユーザーにセキュリティ権限レベル（CONNECT/RESOURCE/DBA）を付与または取り消すことができなく、そして、その他のユーザーのパスワードも変更できないと指します。 ユーザーは DBA 権限を持っています。 このユーザーによって作成された表、ビュー、ドメイン、シノニムは全部データベースに残ります。
DBA	ユーザーの DBA 権限の取り消しは、当該ユーザーは表を作成または削除することができなく、そして、その他のユーザーの権限を付与または取り消すこともできないと指します。 RESOURCE 権限の無いユーザーは CONNECT 権限を持っています。 このユーザーによって作成された表、ビュー、ドメイン、シノニムは全部データベースに残ります。
RESOURCE	ユーザーの RESOURCE 権限の取り消しは、表の作成／削除ができないと指します。 DBA 権限を持つユーザーの権限範囲は変わりません。DBA 権限を持たないユーザーは CONNECT 権限を持っています。 このユーザーによって作成された表、ビュー、ドメイン、シノニムは全部データベースに残ります。
CONNECT	ユーザーの CONNECT 権限の取り消しは、当該ユーザーがデータベースにログインすることができないと指します。 ユーザーはデータベースでの表、ビューに対する所有権が全部取り消されます。 このユーザーによって作成された表、ビュー、ドメイン、シノニムは全部データベースに残ります。

表 8-2：DBMaster データベースの権限レベルの取り消し

グループ管理

複数のユーザーあるいはグループを 1 つにまとめることによって、権限の管理を簡略化することができます。グループにまとめると、1 つの SQL 文で、グループのメンバー全員に同時にデータベース権限を与えることができるようになります。グループは、ユーザーとは異なりますが、ユーザーと同様に取り扱うことができます。ある権限をグループに与えれば、全てのグループメンバーにその権限が与えられます。

SYSADM 、SYSDBA または DBA 権限を持つユーザーのみ下記のことをすることができます。

- グループを作成する。
- グループにメンバーを追加する。
- グループからメンバーを削除する。
- グループを削除する。

グループを作成する

新規グループは、CREATE GROUP 文を使用して作成します。

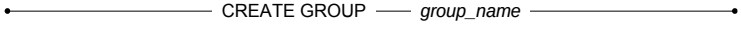


図 8-4 : CREATE GROUP コマンドの構文

*< group name >*は、DBMaster がグループを識別するための一意の名前です。SYSTEM、PUBLIC、GROUP、および既存のユーザー名、グループ名をグループ名に指定することはできません。

➲ **例**

commitee という名前のグループを作成する：

```
dmSQL> CREATE GROUP commitee;
```

グループにメンバーを追加する

グループを作成したら、ADD <ユーザー名/グループ名> TO GROUP 文を使用してグループにメンバーを追加することができます。

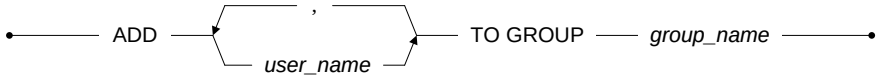


図 8-6 : ADD … TO GROUP コマンドの構文

既存のユーザー名または既存のグループ名をグループメンバーにします。グループ自身をグループメンバーにすることはできません。

➲ **例**

DBA は、ユーザー***Jeff***とグループ ***RD*** をグループ ***commitee*** に追加し、グループ ***commitee*** に表 ***CASEMaker.*** **TB_STAFF** の SELECT 権限を与える：

```
dmSQL> ADD Jeff, RD TO GROUP commitee;
dmSQL> GRANT SELECT ON CASEMaker.TB_STAFF TO commitee;
```

commitee の全員に表 ***CASEMaker. TB_STAF***F の SELECT 権限が与えられます。

グループからメンバーを削除する

REMOVE 〈ユーザー名/グループ名〉 FROM GROUP 文を使用して、指定したグループからユーザーを削除することができます。

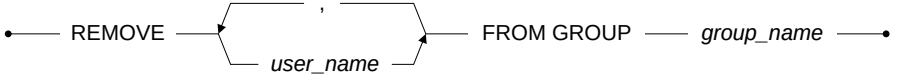


図 8-7 : REMOVE … FROM GROUP コマンドの構文

グループから削除されたメンバーは、グループに与えられている権限を全て失います。しかし、直接メンバーに与えられた権限はそのまま残ります。

➲ **例**

ユーザー***Jeff***をグループ ***commitee*** から削除する：

```
dmSQL> REMOVE Jeff FROM GROUP commitee;
```

これにより、ユーザー*Jeff*はグループ ***commitee*** から削除され、表 ***CASEMaker.****TB_STAFF* の SELECT 権限を失います。

グループを削除する

DROP GROUP 文は、指定したグループをデータベースから削除します。結果として、グループ全員に与えられたグループ権限もなくなります。

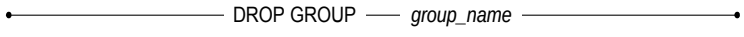


図 8-8 : DROP GROUP 文の構文

➲ **例**

グループ ***commitee*** をデータベースから削除する :

```
dmSQL> DROP GROUP commitee;
```

IPアドレス認証

特定の IP アドレス（例えば、192.72.112.*）でのみデータベースに接続しようとする場合、または特定の IP アドレス（例えば、192.168.0.*）とデータベースとの接続を禁止しようとする場合、IP アドレス認証機能を有効にすると、データベースにアクセスできるクライアントの IP アドレスまたはアクセスできない IP アドレスがコントロールできます。すべてのユーザー設定はシステムカタログ SYSACL に保存されます。

カタログには USER_NAME と IP_ADDRESS の 2 つのカラムが含まれます。

- USER_NAME: ユーザーが接続に使用する名前の設定
- IP_ADDRESS: データベースの接続を可能にするIPアドレス

ユーザー名 **PUBLIC** は予約されています。ユーザー名 **PUBLIC** を使用すると、すべてのユーザーはデータベースに接続するのに指定された設定を満たさなければなりません。

データベースが作成された後、ビューSYSORDERACL は自動的に作成され、全てのユーザーの IP アドレスの情報を表示します。IP アドレスがデータベ

ースに接続できるかどうかをチェックするためにこれらの情報を選択することができます。

IP認証を有効にする

dmconfig.ini ファイルのキーワード DB_STACL を使って IP 確認を有効にできます。IP 認証設定はデータベース起動前に行う必要があります。

- **DB_STACL =1**: IP認証を有効にする
- **DB_STACL =0:** IP認証を無効にする (デフォルト)

認証ルールの作成

IP 認証は以下のような二つのルールがあります：ホワイトリストベースとブラックリストベースです。同じ IP アドレスの場合、その二つのルールによる 制約結果は下記の表に示します。

オーダー / マッチ	ホワイトリストベス	ブラックリストベス
許可される IP アドレスリストにのみマッチする	許可	許可
ブロックされる IP アドレスリストにのみマッチする	ブロック	許可
いずれかにマッチする	ブロック	許可
両方ともにマッチする	ブロック	許可

ホワイトリストベースに設定した場合、SYSAUTHSER のカラム ACLORDER の値は 0 で、ブラックリストベースに設定した場合、SYSAUTHSER のカラム ACLORDER の値は 1 です。

多くの IP アドレスがデータベースに接続するのを許可すると同時にある特定の IP アドレスがデータベースに接続するのを禁止するには、ホワイトリ

ストベースはより適切で、多くの IP アドレスがデータベースに接続するのを禁止すると同時にある特定の IP アドレスがデータベースに接続するのを許可するには、ブラックリストベースはより適切です。そのディフォルトルールはホワイトリストベースです。

➲ **例 1a**

Glowはブラックリストベースを選択し、それを自分の IP 認証ルールとして、クライアントが 127.0.0.1 以外の IP セグメント 127.0.0.*のあらゆる IP アドレスでデータベースに接続することを禁止します。

```
dmSQL> GRANT BLOCK TO Glow '127.0.0.*';
dmSQL> GRANT ALLOW TO Glow '127.0.0.1';
```

➲ **例 1b**

Jeff はホワイトリストベースを選択し、それを自分の IP 認証ルールとして、クライアントが 192.168.0.3 以外の IP セグメント 192.168.0.*のあらゆる IP アドレスでデータベースに接続することを許可します。

```
dmSQL> GRANT ALLOW TO Jeff '192.168.0.*';
dmSQL> GRANT BLOCK TO Jeff '192.168.0.3';
```

ユーザーは一つのみの IP 認証ルールが選択できます。ユーザーの IP 認証ルールを変更するには、**ALTER ACL ORDER** ステートメントを使用します。このルールを変更する前に、すべての付与制約を取り消すことをお勧めします。制約に関する詳細については、以下のセクション：*制約の作成*、*制約の取り消し*をご参照ください。

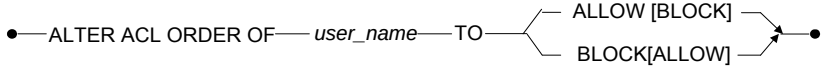


図 8-8： ALTER ACL ORDER

➲ **例**

```
dmSQL> REVOKE BLOCK FROM Vivian ALL;
dmSQL> REVOKE ALLOW FROM Vivian ALL;
dmSQL> ALTER ACL ORDER OF Vivian TO ALLOW BLOCK;
```

制約の作成

IP 認証ルールを設定した後、GRANT ALLOW ステートメント及び GRANT BLOCK ステートメントを使用して、指定の IP 認証ルールの制約を作成することができます。

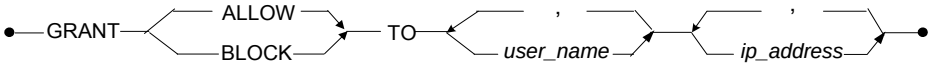


図 8-9： ユーザーリスト IP リストに 許可/ブロック権限を付与

GRANT ALLOW ステートメントは GRANT ACCESS ステートメントと同じです。

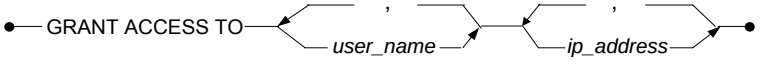


図 8-10：ユーザーリスト/IP リストにアクセス権限を付与

➲ **例**

```
dmSQL> GRANT ACCESS TO vivian,joe '192.72.5.23','140.21.55.*';
dmSQL> GRANT ALLOW TO jane,jetty '192.72.12.20','140.15.45.*';
dmSQL> GRANT BLOCK TO pine,jim '192.70.16.20','139.15.45.*';
```

IP アドレスに関する ALLOW 権限のみがグループ PUBLIC に与えられますので、BLOCK 権限をグループ PUBLIC に与える場合、ERROR（6890）が戻されます。また、一般的には、幾つかの IP アドレスに関する ALLOW 権限をグループ PUBLIC に与えた場合、グループ PUBLIC に属するユーザーはホワイトリストベースまたはブラックリストベースを追加して、自分に新しい制約を追加します。

制約の取り消し

指定の IP 認証ルールの制約を取り消すには、REVOKE ALLOW ステートメント及び REVOKE BLOCK ステートメントを使用します。

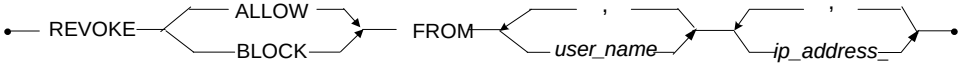


図 8-11：ユーザリスト/IP リストから許可/ブロック権限を取り消す

REVOKE ALLOW ステートメントは REVOKE ACCESS ステートメントと同じです。

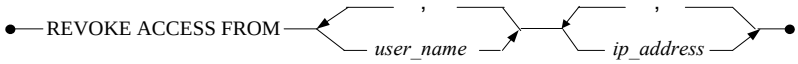


図 8-12：ユーザーリスト/IP リストからアクセス権限を取り消す

順次に指定の IP 認証ルールのあらゆる制約取り消すには、"REVOKE ALLOW/BLOCK FROM user_name ALL"ステートメントを使用します。ここの"ALL"はあらゆる IP アドレスを示しています。

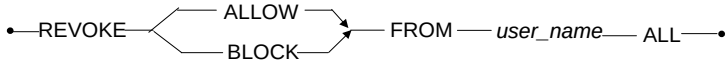


図 8-13 ：あらゆるユーザ名から許可/ブロック権限を取り消す

➲ **例**

```
dmSQL> REVOKE ACCESS FROM vivian,joe '192.72.77.*','140.44.88.23';
dmSQL> REVOKE ALLOW FROM jane,jetty '192.72.12.20','140.15.45.*';
dmSQL> REVOKE BLOCK FROM pine,jim '192.70.16.20','139.15.45.*';
dmSQL> REVOKE BLOCK FROM glow ALL;
```

8.3 オブジェクト権限

*オブジェクト*とは、データベースにある任意の表、ビュー、表/ビューのカラム、ドメイン、シノニムのことです。DBMaster では、ユーザーにオブジェクト権限を与える（GRANT）、取り消す（REVOKE）ことによって、オブジェクトのセキュリティを管理します。

ドメインは、データベースの全ユーザーが参照でき、作成者のみ削除できます。シノニムの権限は元の表に基づきます。ビュー、ドメイン、シノニムの詳細な定義については、6章の「スキーマ・オブジェクト管理」を参照してください。

オブジェクト権限を与える

オブジェクトの作成者はオブジェクトの所有者になり、そのオブジェクトの全ての権限をもちます。オブジェクトの所有者は、GRANT <オブジェクト> 文を用いて、他のユーザーにオブジェクト権限を与えることができます。

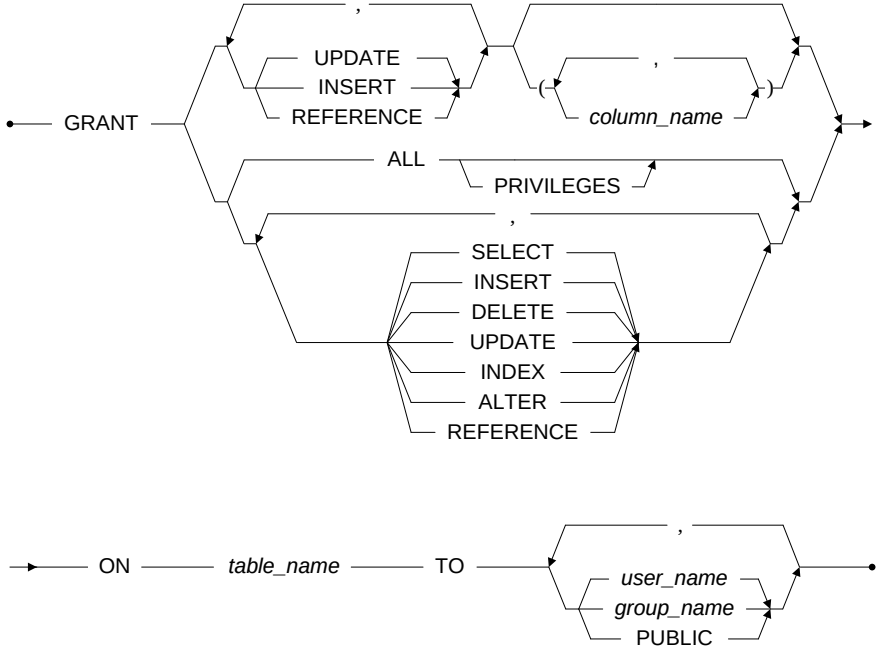


図 8-14 : GRANT(オブジェクト権限)文の構文

DBA 権限のユーザーは、オブジェクトの所有者である無しに関わらず、全ての表やビューに対する権限を与えることができます。RESOURCE 権限の（DBA 権限の無い）ユーザーは、自分が作成した表やビューに対する権限のみを与えることができます。表8-2は、DBMaster で使用している全権限の一覧です。

データベース情報が破壊されるのを防ぐために、INSERT、UPDATE、DELETE 権限には、細心の注意を払って管理する必要があります。ALTER と INDEX 権限は、データベース設計者に限定すべきです。

UPDATE、INSERT、REFERENCE 権限は、特定のカラムに限定することができます。カラム名は、ON 句で指定した表のカラムを修飾名無しで指定します。

権限	説明
SELECT	表/ビューを検索する権限
INSERT	表に行を挿入する権限 （オプション）指定したカラムのみを挿入する
DELETE	表から行を削除する権限
UPDATE	表を更新する権限 （オプション）指定したカラムのみを更新する
INDEX	表索引を作成/削除する権限
ALTER	表定義を変更する権限
REFERENCE	（参照先表の主キーを参照する） 外部キーを参照元表に作成する権限
ALL [PRIVILEGES]	表またはビューに対する上記全ての権限 （PRIVILEGES キーワードはオプション）

表 8-3：DBMaster の表レベルの権限

GRANT <オブジェクト権限>文で指定するユーザーは、少なくとも CONNECT 権限を持っています。グループは、CREATE GROUP 文で作成したグループ名です。PUBLIC キーワードは、全てのユーザーを意味します。PUBLIC に権限を与えることは、ユーザー全員が指定した表権限をもつことを意味します。

➲ **例 1**

Jeff が、自分が作成した表 ***TB_INFO*** を閲覧する権限を ***Cathy*** に与える：

```
dmSQL> GRANT SELECT ON TB_INFO TO Cathy;
```

➲ **例 2**

DBA が、***Jeff*** が作成した表 ***TB_INFO*** を閲覧する権限を ***Cathy*** に与える：

```
dmSQL> GRANT SELECT ON Jeff.TB_INFO TO Cathy;
```

➲ 例 3

DBA が、*Jeff* が作成した表 *TB_INFO* のカラム *phoneno* の INSERT と UPDATE 権限を *Cathy* に与える：

```
dmSQL> GRANT INSERT, UPDATE (phoneno) ON Jeff.TB_INFO TO Cathy;
```

現状態では、*Cathy* にはこのカラムからデータを削除する権限はありません。

➲ 例 4

PUBLIC キーワードで、表 *Jeff.* TB_INFO のデータを閲覧する権限をデータベースの全ユーザーに与える：

```
dmSQL> GRANT SELECT ON Jeff. TB_INFO TO PUBLIC;
```

オブジェクト権限を取り消す

REVOKE <オブジェクト権限>文は、ユーザーに与えた権限を取り消します。構文は、図8-12 のとおりです。

REVOKE <オブジェクト権限>文で指定する権限は、GRANT <オブジェクト権限>文にある権限と同じです。構文ダイアグラム中の、ユーザー名はデータベース権限が与えられたユーザー、グループ名はユーザーの集合体、PUBLIC キーワードはデータベースの全ユーザーを表します。

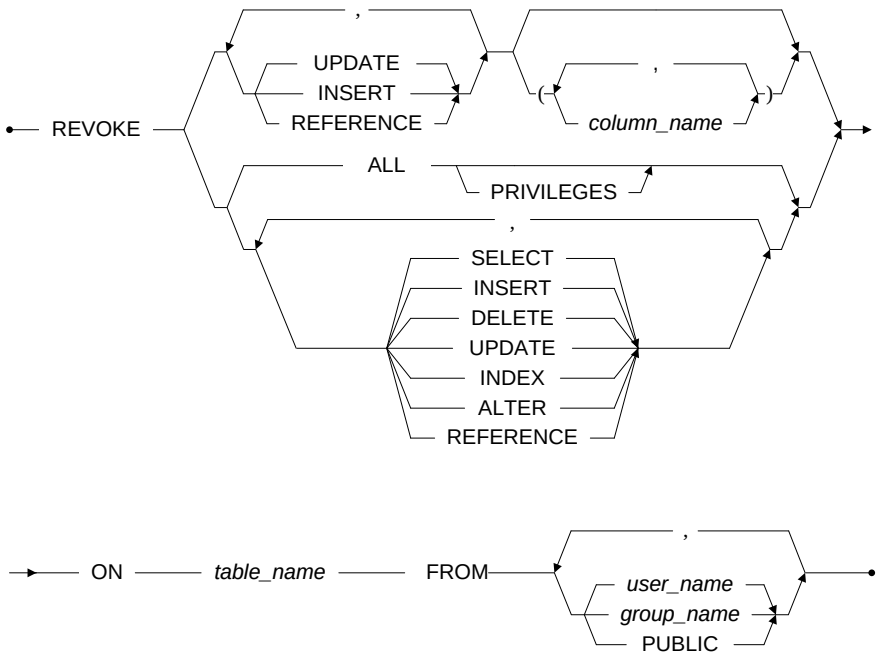


図 8-15：REVOKE(オブジェクト権限)文の構文

➲ **例 1**

Cathy がもっている表 **TB_INFO** の SELECT 権限を取り消す：

```
dmSQL> REVOKE SELECT ON TB_INFO FROM Cathy;
```

➲ **例 2**

Cathy がもっている表 ***Jeff.*** **TB_INFO** の SELECT 権限を取り消す：

```
dmSQL> REVOKE SELECT on Jeff. TB_INFO FROM Cathy;
```

➲ **例 3**

group1 *に与えられている表* ***Jeff.*** **TB_INFO** の ***phoneno*** カラムの UPDATE 権限を取り消す：

```
dmSQL> REVOKE UPDATE (phoneno) on Jeff. TB_INFO FROM group1;
```

➲ **例 4**

PUBLIC に与えられている表 **TB_INFO** の全ての権限を取り消す：

```
dmSQL> REVOKE ALL ON TB_INFO FROM PUBLIC;
```

➲ 例 5

表 TB_INFO の INSERT、UPDATE、SELECT 権限を *Cathy* と *group2* の全メンバーから取り消す：

```
dmSQL> REVOKE INSERT, UPDATE, SELECT ON TB_INFO FROM Cathy, group2;
```

8.4 セキュリティのシステム表

権限レベル、権限、グループに関する全ての情報は、以下のシステムカタログに格納されています。

- SYSAUTHUSER—ユーザー毎の権限レベル
- SYSAUTHTABLE—表権限
- SYSAUTHCOL—INSERT、UPDATE、REFERENCE権限が制限されている表カラム
- SYSAUTHGROUP—グループ名、グループ作成者、グループのメンバー数
- SYSAUTH—グループ名、グループ作成者、グループのメンバー数
- SYSACL—ユーザーのIP確認規則

システム表は、SYSTEM が所有します。システム表は、SYSADM を含めいかなるユーザーも変更することができません。DBMaster のシステム表の詳細については、システムカタログ参照を参照してください。

9 同時実行制御

この章では、トランザクションと同時実行制御の概念を説明します。概念に加えて、ロック機構を用いてマルチユーザー環境における同時アクセスとデータの正確性をどのように維持するかを説明します。9.1節はトランザクションの概念とトランザクション管理に用いられる機能を説明します。9.3節はデータベース・システムにおける同時実行制御の必要性を議論します。9.4節は DBMaster の同時実行制御テクニックを説明します。

9.1 トランザクション

データベースで言うトランザクションは、1 つ以上の SQL 文で構成される作業のまとまりです。トランザクションは不可分のオペレーションです。トランザクションは、一連の SQL 文全体を完了するか、あるいは全く何もしないかのどちらかを意味します。トランザクションには、不可分、永続的、一貫、分離、直列化の属性があります。

トランザクションの状態

トランザクションは、以下のいずれかの状態にあります。

- **アクティブ**—トランザクションが実行を開始すると、直ちにアクティブ状態に入ります。アクティブ状態では、種々のデータベース操作を実行することができます。
- **部分コミット**—トランザクションが、最後のSQL文（COMMIT WORK のような）に達すると、部分コミット状態に入ります。この時点で、トランザクションの実行は終了しますが、実際の出力でエラーが発生して

アボートされることがあり得ます。この場合、出力結果はディスクに書き込まれません。つまり、ハードウェア障害によってトランザクションが失敗するかもしれません。

- **コミット**—トランザクションの実行が成功すると、コミット状態になります。
- **失敗**—トランザクションが正常に続行できない場合、失敗状態になります。失敗状態は、アクティブ状態におけるハードウェアまたはロジックのエラー、ユーザーのアボート指示によって起こります。
- **アボート**—トランザクションが不成功終了すると、アボート状態になります。この場合、トランザクションがデータベースに行ったすべての変更はロールバックされます。

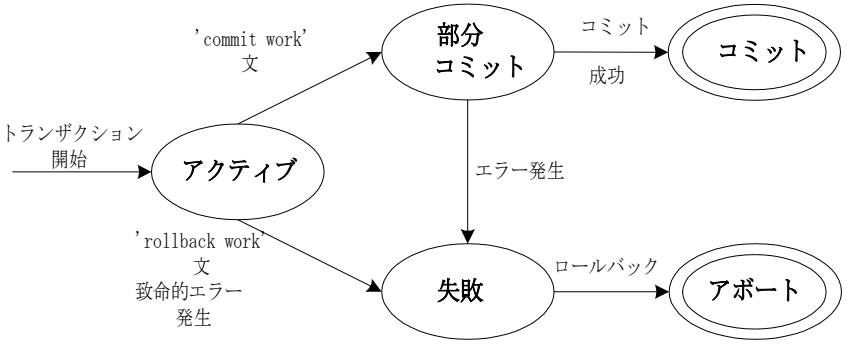


図 9-1：トランザクションの状態遷移図

トランザクションの管理

データベース接続を行うと、自動的にトランザクションが開始し、アクティブ状態になります。トランザクションを終了すると、DBMaster は自動的に新しいトランザクションを開始します。

DBMaster は SQL 文を実行するごとに自動的にコミットします。これを自動コミット・モードといいます。このモードでは、トランザクションの寿命と 1 つの SQL 文の寿命は同じになります。トランザクションは SQL 文が終わると終了し、次の SQL 文のトランザクションが始まります。各々の SQL 文が、独立した 1 つのトランザクションになります。

一連の SQL 文を実行し終わるまでトランザクションをコミットしたくない場合は、SET AUTOCOMMIT OFF 文によって手動コミット・モードに変更することができます。手動コミット・モードでは、COMMIT WORK 文を用いてトランザクションをコミットします。トランザクションを終了するまで、必要な SQL 文を実行し続けることができます。データベースの変更をコミットする場合は COMMIT WORK 文を実行し、データベースの変更を放棄する場合は ROLLBACK WORK 文を実行して、トランザクションを終了します。

自動コミット・モードに戻す場合は、SET AUTOCOMMIT ON 文を実行します。トランザクション・モードの初期値は、自動コミット・モードです。

注　*トランザクションが終了すると、トランザクションに割り当てられた全てのリソースは開放されます。*

セーブポイントを使う

セーブポイントは、トランザクションの途中で任意に宣言することができる中間点です。セーブポイントを使用すると、セーブポイント以降に実行したトランザクションの作業をロールバックすることができます。

例えば、トランザクションが一連の SQL 文を実行しているときに、20 番目の文でエラーが起こったとします。15 番目と 16 番目の文の間にセーブポイントを設けておけば、最初の 15 個の文を保護することができます。セーブポイントまでロールバックし、エラーを修復して 16 番目以降の SQL 文を再試行します。トランザクションをアボートし全ての文を再試行する必要はありません。図9-2はこの例を示しています。

15 番目と 16 番目の文の間にセーブポイントを設けていない場合、トランザクションをアボートし、最初から 15 番目までの文を再実行しなければなりません。これは便利とはいえず、時間がかかります。セーブポイントは、この問題を完全に解決します。

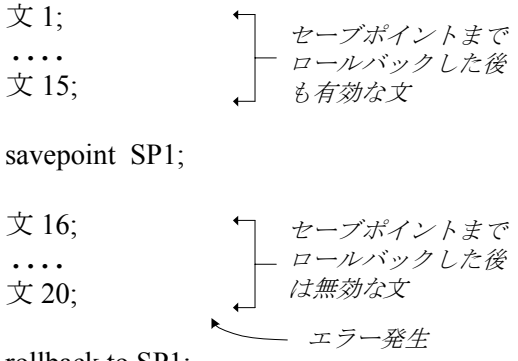


図 9-2：セーブポイントの使用法

SAVEPOINT と ROLLBACK TO…文を使用して、セーブポイントをマークし特定のセーブポイントまでロールバックすることができます。

➲ **例 1**

SAVEPOINT 文：

```
dmSQL> SAVEPOINT <savepoint name>;
```

➲ **例 2**

ROLLBACK TO … 文：

```
dmSQL> ROLLBACK TO <savepoint name>;
```

<セーブポイント名>はユーザーが指定します。セーブポイントまでロールバックすると、セーブポイント以降に取得したロック等のシステムリソースは開放されます。

9.2 トランザクション隔離レベル

同時実行トランザクションの問題

トランザクションが同じデーターベース上で同時に実行される場合ダーティリード(dirty read) 、ノンリピータブルリード(none repeatable read)、ファントムリード(phantom read)と呼ばれる状況が発生します。それぞれ説明します。

特に注意がない限り、以下のデータを元に解説いたします。

表テーブル 1 のカラム c1 に 1、3、5 の値があります。表に二つのトランザクション Tx1、Tx2 を同時に実行します。)

ダーティリード

解説: コミットされていない別トランザクションのデータをもう一方のトランザクションが読み込むこと。

➲ **例**

```
T1                    T2
------------          ------------
Insert 4
Select c1<5
……                   ……
Commit or rollback
```

Tx1 が"select c1 <5"を実行した結果、c1 = 1、3、4 ですが、T2 がロールバックする可能性があるため、この結果は不正なものとなり得ます。

ノンリピータブルリード

解説: トランザクションが既に一度取得したデータが別のトランザクションによって変更され、そのデータを再度取得するもの。

➲ **例**

```
T1                        T2
------------              -----------------
Select c1<5
```

```
                          Update c1=2 where c1=1
                          Commit
Select c1<5
```

T1 の一度目の SELECT では、c1 = 1, 3 が結果として返されます。T2 が値 1 を 2 に UPDATE し、この変更を COMMIT します。T1 が同じクエリを実行すると 2,3 という結果が返されます。T1 は同じクエリを 2 度実行していますがそれぞれ違う値が返ってきています。

ファントムリード

解説: 二つの同じ述語の参照は異なる結果を返します。2 度目の参照では元の結果にはない結果を少なくとも 1 項目返します。

➲ **例**

```
T1                        T2
------------              -----------------
Select c1<5
                          Insert 4
                          Commit
Select c1<5
```

まず、T1 が SELECT 文を実行し、C1=1,3 という結果が返されます。次に T2 が値 4 を INSERT し COMMIT します。その後 T1 が同じクエリを実行すると結果は 1,3,4 となります。T1 は 2 度同じクエリを実行しましたがそれぞれ別の値が返されています。2 度目の結果のレコードには 1 度目の結果に含まれていないものがあります。この例では 2 度目の実行時の値 4 がファントムです。

四つのトランザクション隔離レベル

3 つの同時実行上の問題に関わらず、ANSI/ISO SQL では 4 つの同時実行レベルを定義しています。

隔離レベル	ダーティリード	ノンリーピータブルリード	ファントムリード
リード・ノンコミ	可能	可能	可能

ット			
リード･コミット	不可能	可能	可能
リピータブルリード	不可能	不可能	可能
シリアライザブル	不可能	不可能	不可能

DBMASTERにトランザクション隔離レベルを設置する

DBMaster はトランザクション分離の設定方法を **dmconfig.ini** 設定、ODBC 関数、dmSQL からの SQL 文による 3 種類を用意しています。

DMCONFIG キーワード

関連キーワードは **DB_ISOLV** です。

```
DB_ISOLV {1,2,3,4}
1 : READ UNCOMMITTED
2 : READ COMMITTED
3 : REPEATABLE READ
4 : SERIALIZABLE
```

初期値は 1 です.

DB_ISOLV では各トランザクションのデフォルトの分離レベルを定義します。例えば DB_ISOLV = 3 であれば各トランザクションの分離レベルはリピータブルリードとなります。

ODBC関数

SQLSetConnectionOption は同時実行トランザクション分離レベルを定義するために、SQLGetConnectionOption は分離レベルを取得するために使用されます。:

```
SQLSetConnectOption( HDBC, SQL_ATTR_TXN_ISOLATION, level)
Level : {
SQL_TXN_READ_UNCOMMITTED,
SQL_TXN_READ_COMMITTED,
SQL_TXN_REPEATABLE_READ,
SQL_TXN_SERIALIZABLE
```

```
}
SQLGetConnectOption( HDBC, SQL_ATTR_TXN_ISOLATION, &level)
```

SQL構文

分離レベルの情報を取得します：

➲ **例**

```
dmSQL> set transaction isolation level READ UNCOMMITTED;
dmSQL> call getsystemoption('isolv',?);
OPTION_VALUE :SQL_TRANSACTION_READ_UNCOMMITTED
```

9.3 マルチユーザー環境

複数の利用者がデータベースをアクセスするときは、同じデータを同時にアクセスするときに起こる状況について考慮する必要があります。

セッション

利用者と DBMaster 間の通信経路のことを*接続*と言います。通信経路は、共有メモリまたはネットワークを利用して確立します。

➲ **例**

データベースとの接続を確立する：

```
dmSQL> CONNECT TO database_name user_name password;
```

データベースに接続する特定の接続のことをセッションと言います。セッションは、データベースに接続した時から切断する時まで継続します。1 つのセッションでは、一度に 1 つのアクティブ・トランザクションしか行うことができません。

同時実行制御の必要性

マルチユーザー・データベースのシステム環境では、複数の利用者が同時にデータベースに接続することができます。結果として、多くのトランザクションにより同じデータベースを同時に更新することになります。

同時実行を制御するメカニズムが何もない場合、以下のようなデータ不整合が様々な状況で発生します。

- 更新紛失問題
- 一時更新問題
- 不正合計問題

更新紛失問題

更新紛失問題は、2 つのトランザクションで同じデータを同時に更新するときに発生します。

➲ **例**

トランザクション T1 と T2 が X の値を読み込み、異なる計算をして修正します。各トランザクションは、異なる X の値をもつことになります。T2 が書き込む前に、T1 が X の値をデータベースに書き込みます。次に T2 がデータベースに書き込まれた T1 の X の値に上書きします。T1 が書き込んだ値は無くなります：

```
      T1                      T2
--------------          --------------
read(X);
                        read(X);
X = X - N;
                        X = X + M;、
Write(X);
                        write(X);
```

一時更新問題

一時更新問題は、トランザクションがデータを更新し、別のトランザクションが同じデータを更新した後に、ロールバックした時に発生します。

➲ **例**

トランザクション T1 は、X の値を読み込んで修正し、データベースに書き込み、他の文の実行を続けます。この間に、トランザクション T2 が X の値

を読み込み、新しい値に修正してデータベースに書き込みます。一方、トランザクション T1 は、途中で失敗し、全ての値をデータベース更新前の状態にするために、ロールバックします。データベース管理システムは、X の値を元に戻して T2 が書き込んだ値に上書きします。トランザクション T2 が計算した X の値は、一時的にしか存在しません。

```
      T1                      T2
--------------          --------------
read(X);
X = X - N;
write(X);
                        read(X);
                        X = X + M;
                        write(X);
rollback;
```

不正合計問題

不正合計問題は、トランザクションがレコード合計を集計中に、別のトランザクションが同じレコードを更新するときに発生します。

➲ 例

トランザクション T1 で X と Y の値を集計し、同時にトランザクション T2 が同じ X と Y を更新します。トランザクション T2 は、T1 が X の値を集計する前に X を更新し、T1 が Y の値を集計した後に Y を更新します。トランザクション T1 は、更新前と更新後の値を集計することになります。両トランザクションが終了したとき、集計された合計は、データベースにある値の合計とは異なることになります。

```
      T1                      T2
--------------          --------------
sum = 0;
                        read(X);
                        X = X - N;
                        write(X);
read(X);
sum = sum + X;
```

```
read(Y);
sum = sum + Y;
                          read(Y);
                          Y = Y + N;
                          write(Y);
```

同時実行性の問題を解決するには、ロックやタイムスタンプなど種々のテクニックがあります。次節では、トランザクションの同時実行を制御するために、DBMaster で使用しているロックについて説明します。

9.4 ロック

この節では、まずロックの概念を説明します。次に、DBMaster のロックのメカニズムを解説し、ロック単位とロックモードを説明します。最後に、デッドロックをどのように取り扱うかを示します。

ダイナミックカラムのデータタイプを変更します。**table tb_student** の詳細については、*カラムを追加/削除する*をご参照ください。

```
dmSQL> ALTER TABLE tb_student modify dynamic column score type to varchar(20);
```

ロックの概念

一般に、マルチユーザーのデータベース・システムは、同時トランザクションのアクセスの同期をとるために、種々の形態のロックを使用します。トランザクションは、表やレコードのようなデータにアクセスする前に、これらのデータをロックする必要があります。

DBMaster のロックは完全に自動化されており、ユーザーが何かをする必要はありません。全ての SQL 文で、暗黙のうちにロックがかけられます。データベース内のどのデータにも、ユーザーが明示的にロックする必要はありません。

共有ロックと排他ロック

一般に、マルチユーザーのデータベースでは、複数の読み込みと、1 つの書き込み操作を認める 3 種類のロックが使用されます。

- **共有ロック(S)**—共有ロックは、トランザクションがデータの読み込み操作を伴うことを意味します。複数のトランザクションで、同時に同じデータの共有ロックをかけることができ、高い同時実行性を実現することができます。
- **更新ロック (U)**—更新ロックは、トランザクションがデータの更新操作を伴うこと或いは操作目的を持つことを意味します。このロックは共有ロックと共存できますが排他ロックと共存できません。一つのオブジェクトは一回に一つの更新ロックを持ちます。
- **排他ロック(X)**—排他ロックは、トランザクションがデータの更新操作を伴うことを意味します。排他ロックが開放されるまでは、データにアクセスできるトランザクションは1つのみです。

2フェーズロック

2 フェーズロック・プロトコルは、トランザクションを確実に連番化するために用います。2 フェーズロック・プロトコルでは、各トランザクションは、アンロック要求を出す前に、全てのロック要求を出しておかなければなりません。名前が示すように、このプロトコルは 2 つのフェーズに分けられます。

- **拡大（成長）フェーズ**—このフェーズでは、必要なあらゆるロック要求を出します。このフェーズでは、アンロック要求はできません。
- **縮小フェーズ**—このフェーズでは、トランザクションに拡大フェーズで獲得したロックを解除させます。このフェーズでは、新規のロック要求はできません。

DBMaster では、同時実行性を制御するために、トランザクションを連番化することによって、2 フェーズロック・プロトコルを使用しています。

デッドロック

デッドロックは、2 つ以上のトランザクションが、他のトランザクションによってロックされたデータの解放を待機しているときに発生します。

➲ 例

T1 は T2 が X の共有ロックを解除するのを待ち、T2 は T1 が Y の共有ロックを解除するのを待ちます。その結果、デッドロックが発生してシステムは無限に待機します。

```
T1                                  T2
------------                        ------------
共有ロック(Y);
read(Y);
共有ロック(X);
read(X);
排他ロック(X)
(T1はT2の解除を待つ)                排他ロック(Y);
(T2はT1の解除を待つ)
```

ロックの単位

DBMaster には、表（リレーション）、ページ、行（タプル、レコード）の 3 段階のロック単位があります。表はページで構成され、ページは行で構成されています。

高いレベルのロックをかけると、自動的に低いレベルにも適用されます。例えば、表に排他ロック(X ロック)をかけると、表の中にある全てのページと行にも排他ロックが適用されます。このため、他のユーザーは表に含まれるページや行にアクセスできなくなります。但し、行に排他ロックをかけても、同時に他の行に排他ロックをかけることは可能です。排他ロックを使用するときに、同レベルの 2 つのオブジェクトが干渉することはありません。DBMaster のロック単位(レベル)を図9-3 に示します。

図 9-3：ロック単位

高いレベルのロック単位を使用すると、同時実行性の度合いは下がります。但し、共有メモリ等のシステムリソースの使用を小さくなります。ロック

単位の選択は、同時実行性とリソース使用の間のトレードオフになります。初期設定のロック単位は、行です。それ以外のロック単位を使用する必要がある場合は、表作成時に指定することができます。詳細については、5章の「ストレージアーキテクチャ」を参照してください。

ロックの種類

DBMaster がサポートする主なロックモード（タイプ）は、共有(S)ロック、更新ロック（U）と排他(X)ロックです。複数のユーザーが同時にデータに共有ロックをかけることはできますが、1 つのオブジェクトに対して排他ロック或いは更新ロックをかけることができるのは 1 ユーザーのみです。共有ロックと排他ロック更新ロック以外に、内包ロックと呼ばれるロックモードも使用することができます。

データをロックすると、高位のオブジェクトが自動的に内包ロックされます。例えば、ある行に共有ロックをかけると、その行を含むページに内包共有(IS)ロックがかけられ、その行を含む表に内包共有ロックがかけられます。

DBMaster がサポートする内包ロックモードには、以下のものがあります。

- IS—低位で共有ロックが指定されていることを示します。
- IU—低位で更新ロックが指定されていることを示します。
- IX—低位で排他ロックが指定されていることを示します。
- SIX—低位で排他ロックが指定され、同位で共有ロックが指定されていることを示します。共有ロックとIXロックの組み合わせです。
- SIU—低位で更新ロックが指定され、同位で共有ロックが指定されていることを示します。共有ロックとIUロックの組み合わせです。
- UIX—低位で排他ロックが指定され、同位で更新ロックが指定されていることを示します。共有ロックとIXロックの組み合わせです。

表9-1 は、ロックモードの共存性をマトリックスで示します。マトリックスの真は、それぞれのロックモードが両立すること、同時に 1 つのデータに存在できることを意味します。偽は、それぞれのロックモードが両立できないこと、同時に存在することができないことを意味します。

ロック要求が既存のロックと競合する場合、既存のロックが解除されるか、ロック要求の時間切れになるまで、ロック要求は実行されません。ロック待機が時間切れになると、「ロックタイムアウト」のエラーメッセージが返されます。ロック待ち時間の初期設定値は 5 秒です。但し、**dmconfig.ini** ファイルにある **DB_LTimO** キーワードに、別の値を指定することができます。

➲ **例**

待ち時間を 8 秒に設定する：

```
DB_LTIMO = 8;
```

	N	IS	S	IU	SIU	IX	U	SIX	UIX	X
N	真	真	真	真	真	真	真	真	真	真
IS	真	真	真	真	真	真	真	真	真	偽
S	真	真	真	真	真	偽	真	偽	偽	偽
IU	真	真	真	真	真	真	偽	真	偽	偽
SIU	真	真	真	真	真	偽	偽	偽	偽	偽
IX	真	真	偽	真	偽	真	偽	偽	偽	偽
U	真	真	真	偽	偽	偽	偽	偽	偽	偽
SIX	真	真	偽	真	偽	偽	偽	偽	偽	偽
UIX	真	真	偽	偽	偽	偽	偽	偽	偽	偽
X	真	偽	偽	偽	偽	偽	偽	偽	偽	偽

表 9-1：ロックモードの共存性マトリックス

デッドロックの取り扱い

DBMaster は、待ちグラフを解析することによって、自動的にデッドロックを検出します。デッドロックが検出されると、デッドロックを解消するためにアボートされ犠牲になるトランザクションがあります。

➲ 例

デッドロック問題の例では、トランザクション T2 が Y に排他ロックをかけたときにデッドロックが発生し検出されます。この場合、トランザクション T2 がアボートされます。トランザクション T2 を実行したユーザーは、「デッドロックのためトランザクションアボート」のエラーメッセージを受け取ります。

```
      T1                          T2
--------------                ------------
共有ロック(Y);
read(Y);
                              共有ロック(X);
                              read(X);
排他ロック(X);
(T1 は T2 の解除を待つ)          排他ロック(Y);
                              (T2 は T1 の解除を待つ)

                              T2 は DBMaster によってアボートされる
```

10 トリガー

DBMaster のデータベース・サーバーのトリガーは、非常に役に立つ強力な機能です。トリガーは、特定のイベントが発生したときに、予め定義した文を自動的に実行させるために使用します。どのユーザーあるいはアプリケーションプログラムが生成したイベントであるかは問いません。

トリガーは、標準の SQL 文では不可能な方法でデータベースをカスタマイズすることを可能にします。データベースは、ユーザーやアプリケーションプログラムが明示的なアクションを指示しなくても、複雑で類型的でないデータベース操作を一貫して制御するようになります。

以下の用途にトリガーを使用することができます。

- ビジネスルールを組み込む
- データベース作業の追跡記録を作成する
- 既存のデータから別の計算値を導きだす
- 複数の表にデータを複製する
- セキュリティの認証手順を実施する
- データ整合性を制御する
- 類型的でない整合性制約を定義す

追跡が困難で変更が難しい複雑な相互依存性をデータベース内に形成しないように、トリガーの使用は抑制します。一般に、標準の SQL 文では作成できない機能あるいは整合性制約を組み込む場合にのみ、トリガーを使用します。

10.1 トリガーの構成要素

トリガーの定義は、システムカタログに格納されています。

トリガーは、6 つの要素から構成されます。

トリガー名—トリガーを一意に識別する名前。

トリガーアクションタイム—トリガーの起動時点（イベントの前か後）。

トリガーイベント—表にデータを挿入するなど、データベースに発生する特定の状況。

トリガー表—トリガーイベントを実行する表名。

トリガーアクション—トリガーイベントが発生したときに、自動的に実行される SQL 文またはストアド・プロシージャ。

トリガータイプ—トリガーのタイプ。

作成するトリガーには、これら全ての構成要素が備わっていなければなりません。更にオプションとして、REFERENCING 句があります。

トリガー名

トリガー名は、表に関連付けられるトリガーを一意的に識別します。トリガー名は、128 文字までの英数字、アンダースコア文字、記号#と$からなります。先頭文字に数字は使えません。空白は使用できません。

トリガーアクションタイム

トリガーアクションタイムは、トリガーを誘発する SQL 文を実行する前、或いは実行する後のどちらにトリガー・アクションを起動するかを指定します。トリガーアクションタイムは、BEFORE 又は AFTER キーワードで指定します。BEFORE キーワードは、トリガーイベントの SQL 文の実行前にトリガー・アクションが起動することを示し、AFTER キーワードはトリガーイベントの SQL 文の実行後にトリガー・アクションが起動することを示します。各トリガーには、どちらかのトリガータイムを指定します。

トリガーイベント

トリガーイベントは、トリガーを誘発するデータベース操作です。トリガーイベントは、トリガー表に対する INSERT、UPDATE、DELETE 文のいずれかです。トリガーにはトリガーイベントを 1 つのみ指定しますが、複数のトリガーに、複数のトリガーイベントを指定することができます。

トリガー表

トリガー表は、トリガーイベントを実行する表です。トリガーは、トリガー表に関連付けられます。トリガー表は、実際に存在する表です。一時表、ビュー、シノニムをトリガー表にすることはできません。トリガーには、1 つのトリガー表しか指定することはできません。

トリガーアクション

トリガーアクションは、トリガーが起動したときに実行する INSERT、UPDATE、DELETE、EXECUTE PROCEDURE 文です。トリガーには、1 つのトリガーアクションしか指定することはできません。

トリガータイプ

トリガータイプは、各トリガーイベントに応じて何回トリガーを起動させるかを指定します。行トリガーと文トリガーの 2 種類のトリガーがあります。行トリガーは FOR EACH ROW キーワードで指定し、トリガーイベントによって修正される行毎に起動します。文トリガーは FOR EACH STATEMENT キーワードで指定し、トリガーイベントに対して 1 度だけトリガーアクションを起動します。

REFERENCING句

REFERENCING 句は、カラムの新旧の値に対する相関名を定義します。初期設定の相関名 OLD、NEW が、同じ名前をもつ表と競合して使用できない場合に用います。

10.2 トリガー操作

ユーザーやアプリケーション・プログラムがトリガーイベントを表に実行する度に、トリガーを起動すべきか見極められ、トリガーイベントが定義されている場合は、トリガーアクションが実行されます。データベース内でトリガーは起動するので、全てのアプリケーションにわたり、データの一貫性は保たれます。特定のイベントが発生したときには、関連するアクションも必ず実行することが保証されます。

トリガーは、ドメイン整合性、カラム整合性、参照整合性、非類型的な制約を定義するのに使用することができます。但し、宣言型の整合性制御でそれらを行うことも可能です。
トリガーには所有者がありません。トリガーは表に関連付けられます。

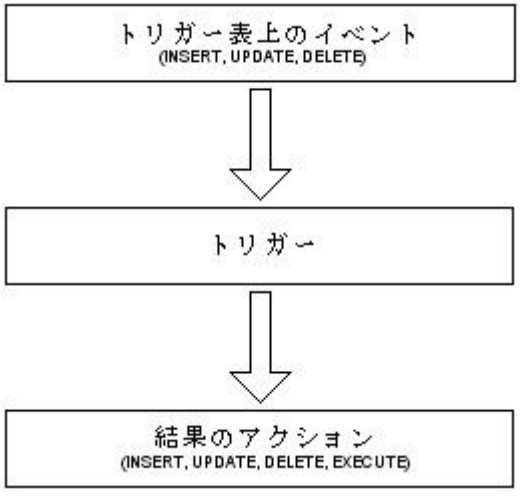


図 10-1：トリガーイベントとアクション

10.3 トリガーを作成する

CREATE TRIGGER 文は、トリガーを作成し、特定の表に関連付けます。DBA 権限以上のユーザーか、関連付ける表の所有者が、トリガーを作成するこ

とができます。又、トリガー定義で参照する全てのオブジェクトに対し、必要なオブジェクト権限を持っている必要があります。

基本的な必要事項

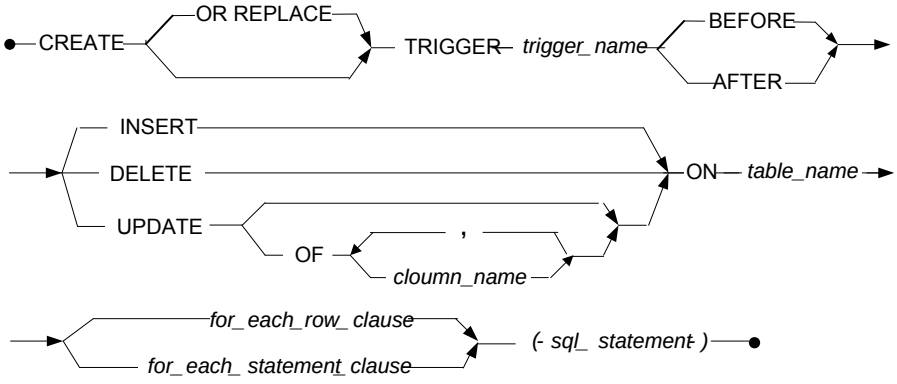
CREATE TRIGGER 文には、以下の要素が必ず含まれます。

- トリガー名
- トリガーアクションタイム (実行前/実行後)
- トリガーイベント
- トリガー表
- トリガーアクション
- トリガータイプ (行トリガー/文トリガー)

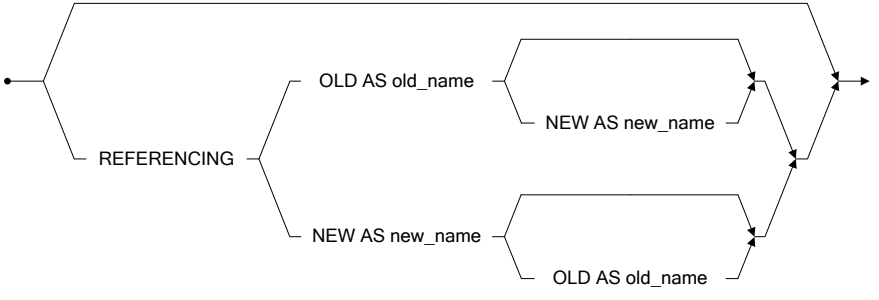
セキュリティ権限

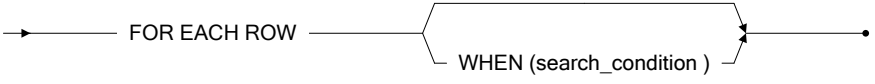
トリガーアクションにある SQL 文は、トリガーイベントを実行するユーザーの権限ではなく、トリガー表の所有者と同じ権限で作動します。トリガーが存在する場合、トリガーイベントを実行することができるユーザーであれば、トリガーを使用することができます。

CREATE TRIGGER構文



FOR EACH ROW句





FOR EACH STATEMENT句

図 10-2 : CREATE TRIGGER 文の構文

OR REPLACE: OR REPALCE で既存なトリガーを再び作成します。この句を使用して削除しないで既存なトリガーの定義を変更でき、このトリガーは再び作成できます。

➲ **例 1**

tb_staff 表にトリガーを作成する：

```
dmSQL> CREATE OR REPLACE TRIGGER tr_staff_insert AFTER INSERT ON tb_staff
       FOR EACH ROW WHEN (new.ID > 0)
       (INSERT INTO tb_salary(new.ID, new.Name,NULL, NULL, NULL));
```

トリガーアクションタイムを定義する

表への 1 つのイベントに対して、トリガータイムとトリガータイプを組み合わせることによって、4 種類のトリガーBEFORE/FOR EACH ROW、AFTER/FOR EACH ROW、BEFORE/FOR EACH STATEMENT、AFTER/FOR EACH STATEMENT を作成することができます。

BEFORE/FOR EACH STATEMENT トリガーは、トリガーを誘発する文を実行する前、つまりトリガーイベントを実行する前に、トリガーアクションが 1 度だけ実行されます。AFTER/FOR EACH STATEMENT トリガーは、トリガーを誘発する文が完了した後に、1 度だけトリガーアクションが実行されます。BEFORE/AFTER FOR EACH STATEMENT トリガーは、トリガーを引き起こす文が何も行を処理しない場合でも実行されます。

INSERT/DELETEイベントのトリガーアクションタイム

以下の例は、INSERT / DELETE トリガーイベントの前後に起動させるトリガーの作成方法です。<*SQL 文*>は、トリガーアクションを表しています。

➲ **例 1**

表 **tb** への INSERT イベントに対し 4 つのトリガーを定義する：

```
CREATE TRIGGER tr1 BEFORE INSERT ON tb FOR EACH STATEMENT <sql statement>
CREATE TRIGGER tr2 BEFORE INSERT ON tb FOR EACH ROW <sql statement>
CREATE TRIGGER tr3 AFTER INSERT ON tb FOR EACH ROW <sql statement>
CREATE TRIGGER tr4 AFTER INSERT ON tb FOR EACH STATEMENT <sql_statement>
```

➲ **例 2**

表 tb への DELETE イベントに対し 4 つのトリガーを定義する:

```
CREATE TRIGGER tr1 BEFORE DELETE ON tb FOR EACH STATEMENT <sql_statement>
CREATE TRIGGER tr2 BEFORE DELETE ON tb FOR EACH ROW <sql_statement>
CREATE TRIGGER tr3 AFTER DELETE ON tb FOR EACH ROW <sql_statement>
```

```
CREATE TRIGGER tr4 AFTER DELETE ON tb FOR EACH STATEMENT <sql_statement>
```

UPDATEトリガーイベントのトリガーアクションタイム

UPDATE イベントの場合は状況が異なります。表が更新されると必ず起動する UPDATE 表トリガーを作成するか、指定したカラムが更新されたときのみ起動する UPDATE カラムトリガーを作成するかを選びます。いずれのトリガーも単一の表に作成します。UPDATE カラムトリガーには複数のカラムを指定することができますが、これらのカラムは、互いに排他的でなければなりません。

➲ **例**

basepay、bonus、id、name の 4 つのカラムがもつ表 tb_salary のカラム basepay, bonus にトリガー*tr* UpdateColumn を作成する：

```
dmSQL> CREATE TRIGGER tr UpdateColumn AFTER UPDATE OF basepay, bonus ON tb salary
       FOR EACH ROW
       (INSERT INTO tb_OldSalary VALUES (old.basepay, old.bonus));
```

bonus カラムを指定する 2 つ目の UPDATE カラムトリガー**tr_UpdateBonus** を作成すると、**bonus** は既にトリガーに登場しているので、この SQL 文は失敗します。

```
dmSQL> CREATE TRIGGER tr UpdateBonus AFTER UPDATE OF bonus,tax ON tb salary
       FOR EACH ROW
       (INSERT INTO tb_oldTax VALUES (old.bonus, old.tax));ERROR (6150): [DBMaster]
insert/update 値のタイプとカラム・データのタイプが矛盾しているか、又は、比較オペランドの値と記述
語にあるカラム・データのタイプが矛盾しています
```

表に 4 つのカラムがある場合、同じトリガータイプ(例えば、BEFORE/FOR EACH ROW トリガー)に対して、最大 4 つの UPDATE カラムトリガー、或いは 1 つの表更新トリガーを定義することができます。

FOR EACH ROW / FOR EACH STATEMENT句

FOR EACH STATEMENT 句は、各トリガーイベントに対しトリガーアクションを一度だけ起動させるよう指定します。トリガーイベント文が、結果としてどの行も処理しなかった場合でも、トリガーは起動します。

FOR EACH ROW 句は、トリガーイベントが修正した各行に対し、行ごとにトリガーアクションが一度起動するように指定します。トリガーイベントがどの行も修正しない場合、トリガーは起動しません。OLD と NEW キーワードは、トリガーアクションにトリガー表の値を指定する際に、トリガー表の変更前と変更後の値を識別するために使用します。OLD キーワードは、トリガーイベントが実行される前のトリガー表の値を意味します。NEW キーワードは、トリガーイベントが実行された後のトリガー表の値を意味します。

➲ **例 1**

次の文は、表 ***Sales*** に UPDATE カラムトリガーを作成する方法を示しています。***totSales*** フィールドは、2 つのフィールド ***unitPrice*** と ***unitSale*** から算出されます。***unitPrice*** と ***unitSale*** はトリガーカラムです。

```
dmSQL> CREATE TRIGGER tr TotalSale AFTER UPDATE OF unitPrice, unitSale ON Sales
                    FOR EACH ROW
                   (UPDATE Sales SET totSales = new.unitPrice * new.unitSale);
```

➲ **例 2**

この例では、4 つのトリガーがあります。

```
dmSQL> CREATE TRIGGER tr_BeforeUpdatePro BEFORE UPDATE ON tb_Orders
                     FOR EACH STATEMENT
                    (EXECUTE PROCEDURE checkPrivilege);

dmSQL> CREATE TRIGGER tr_BeforeUpdate BEFORE UPDATE ON tb_Orders
                     FOR EACH ROW
                    (INSERT INTO Log_Old_Value (old.customer, old.amount));

dmSQL> CREATE TRIGGER tr_AfterUpdate AFTER UPDATE ON tb_Orders
                     FOR EACH ROW
                    (INSERT INTO Log_New_Value (new.customer, new.amount));

dmSQL> CREATE TRIGGER tr_AfterUpdatePor AFTER UPDATE ON tb_Orders
                     FOR EACH STATEMENT
                    (EXECUTE PROCEDURE Log_Time);
```

ユーザーが ***Orders*** 表の 2 つの行を変更する UPDATE 文を実行した場合、その影響と実行順序は以下のようになります。

1. プロシージャ checkPrivilege の呼び出し
2. Log_Old_Value 表に 1 行挿入
3. 1 行更新
4. Log_New_Value 表に 1 行挿入
5. Log_Old_Value 表に 1 行挿入
6. 1 行更新
7. Log_New_Value 表に 1 行挿入
8. プロシージャ *Log_Time* の呼び出し

REFERENCING句を使う

行トリガーの際、トリガーアクションに指定するカラムの値を、トリガーイベントの実行前にするのか、実行後にするのか、その*<SQL 文>*の中で明確にする必要があります。例えば、販売品の価格を更新する際に、古い価格と新しい価格の記録を残す場合は、「FOR EACH ROW / FOR EACH STATEMENT 句」セクションの例 2 のようにキーワード OLD と NEW を使って、変更前と変更後の値を指定します。

但し､表の中にNEWやOLDといった名前のカラムがある場合があります。このような場合、替わりに相関名を定義するために REFERENCING 句を使用します。REFERENCING 句は、カラム名の新旧の値を意味する 2 つの接頭語を定義します。これらの接頭語は、相関名と呼ばれています。キーワード OLD と NEW で、相関名を定義します。

➲ **例**

```
dmSQL> CREATE TRIGGER tr_log_price AFTER UPDATE OF price ON New
                                   REFERENCING OLD as pre NEW as post
                                   FOR EACH ROW
                                   (INSERT INTO logTbl
                                    VALUES (item_no, today(), pre.price,
                                    post.price));
```

この例では、トリガー表の名前が ***New*** なので、相関名 ***pre*** と ***post*** をアクショントリガー文に使用します。REFERENCING 句は、行トリガーにのみ有効なので、文トリガーで指定することはできません。

トリガーイベントが INSERT の場合、新たに挿入されたレコードには変更前の値はありませんので、変更前の値を使用することはできません。同様に、トリガーイベントが DELETE の場合、削除したレコードに新しい値はありませんので、変更後の値を利用することはできません。UPDATE イベントトリガーでは、変更前と変更後いずれの値も利用することができます。

WHEN条件句を使用する

行トリガーに WHEN 条件句を加えると、条件式の結果によってトリガーアクションを実行させることができます。WHEN 条件句は、キーワード WHEN と（）で括った条件式を、アクションタイムの後ろ、トリガーアクションの前に記述します。WHEN 条件句は、行トリガーでのみ使用し、文トリガーでは使用することができません。

➲ **例 1**

表 **tb_logComplain** に顧客の苦情の記録を残すトリガーを作成する。条件式 call_code = '**c**'の'**c**'は、苦情電話を意味します。

```
dmSQL> CREATE TRIGGER tr log complain AFTER INSERT ON Customer Call
                                      FOR EACH ROW
                                      WHEN (new.call code = 'c')
                                      (INSERT INTO tb logComplain
                                       VALUES (Today(), Cus_Name));
```

トリガー定義の中に WHEN 条件句が含まれているときは、行毎に WHEN 句が判断されます。WHEN 条件句が TRUE の行に対しては、トリガーアクションが起動します。WHEN 条件句が FALSE または UNKNOWN の行に対しては、トリガーアクションは起動しません。

WHEN 条件の結果は、トリガーアクションの実行にのみ影響します。トリガーを引き起こす文には、何の影響も与えません。

➲ **例 2**

表 **tb_staff** に全ての INSERT、UPDATE、DELETE 文を記録する 3 つのトリガーを作成する：

```
dmSQL> CREATE TRIGGER tr_staff _insert AFTER INSERT ON tb_staff
                                      FOR EACH ROW
```

```
                                    (INSERT INTO tb salary
                                     VALUES (new.Id, new.Name, NULL, NULL, NULL));

dmSQL> CREATE TRIGGER tr staff  update AFTER UPDATE ON tb staff
                                   FOR EACH ROW
                                   (INSERT INTO tb_salary
                                    VALUES (ld.Id, old.Name,new.Id, new.Name));

dmSQL> CREATE TRIGGER tr_staff _update AFTER DELETE ON tb_staff
                                   FOR EACH ROW
                                   (INSERT INTO tb_salary
                                    VALUES (old.Id, old.Name,
                                    NULL, NULL));
```

➲ **例 3**

主キーが変更されたときに、カスケードして外部キーを変更させる(***deptNo*** カラムを表 **tb_dept** の主キー、***DeptNo*** カラムを表 **tb_staff** の外部キーと想定):

```
dmSQL> CREATE TRIGGER trig_upd_dept BEFORE UPDATE OF deptNo ON tb_dept
                                   FOR EACH ROW
                                   WHEN (NEW.deptNo <> OLD.deptNo)
                                   (UPDATE tb_staff SET tb_staff.ID = NEW.deptNo
                                    WHERE tb_staff.ID = OLD.deptNo);
```

➲ **例 4**

主キーを削除した時に、カスケードして全ての外部キーを削除させる:

```
dmSQL> CREATE TRIGGER tr dept delete BEFORE DELETE ON tb dept
                                   FOR EACH ROW
                                   (DELETE FROM tb staff
                                    WHERE tb_staff.ID  = OLD.deptNo);
```

➲ **例 5**

主キーを更新した時に、カスケードして全ての外部キーを NULL にする:

```
dmSQL> CREATE TRIGGER tr_dept_delete BEFORE UPDATE ON tb_dept
                                   FOR EACH ROW
                                   (UPDATE tb_staff set DeptNo = NULL
                                    WHERE tb_staff.ID = OLD.deptNo);
```

➲ 例 6

部品の在庫が底をついた時に、自動的に再発注され、**tb_pending_orders** 表に部品番号と数量を記録させるトリガーを作成する:

tb_Inventory 表: part_no int, parts_on_hand int, reorder_level int, reorder_qty int

tb_pending_orders 表: part_no int, qty int, order_date date

```
dmSQL> CREATE TRIGGER tr reorder AFTER UPDATE OF parts on hand ON tb Inventory
                                 FOR EACH ROW
                                 WHEN (new.parts on hand < new.reorder level)
                                 (INSERT INTO tb pending orders
                                  VALUES (new.part_no, new.reorder_qty, today()));
```

トリガーアクションを指定する

トリガーアクションは、トリガーイベントが発生した時に実行される SQL 文です。トリガーアクションは、INSERT、DELETE、UPDATE、EXECUTE PROCEDURE 文のいずれかです。これ以外の文を使うことはできません。ストアド・プロシージャに、COMMIT、ROLLBACK、SAVEPOINT トランザクション制御文を入れることはできません。各トリガーには（）で括ったトリガーアクションを 1 つ指定します。

➲ 例

表 **tb_staff** にトリガーを作成する:

```
dmSQL> CREATE TRIGGER tr staff insert AFTER INSERT ON emp
                                  FOR EACH ROW WHEN (new.empNo > 0)
                                  (INSERT INTO tb salary(new.ID (new.empName,
                                   new.empAddress, new.Manager));
```

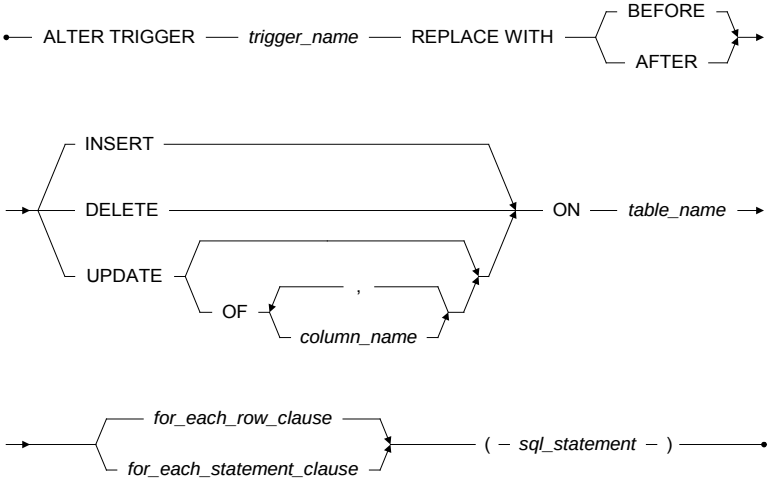
この例では、トリガー名は **tr_staff_insert** です。表 **tb_staff**.で INSERT 文が実行された後に、トリガーが起動することを意味する AFTER キーワードが指定しています。トリガーを誘発するトリガーイベントの SQL 文は、INSERT で、トリガー表は **tb_salary** です。トリガータイプは、行トリガーです。

```
dmSQL> CREATE TRIGGER trDelAcct AFTER DELETE ON tb_salary
                                FOR EACH ROW
                                (INSERT INTO tb_oldsalry
                                 VALUES (Old. name));
```

この例では、トリガーtr_salry_Del は、tb_salary 表から 1 つのレコードが削除された時、削除された顧客名を tb_old_salry 表に追加します。一時表、ビュー、システム表にトリガーを作成することはできません。

10.4 トリガーを修正する

トリガーを部分的に修正することはできません。しかし、トリガーの定義を置き換えることはできます。トリガー定義を修正するには、ALTER TRIGGER 文を使用します。



FOR EACH ROW句

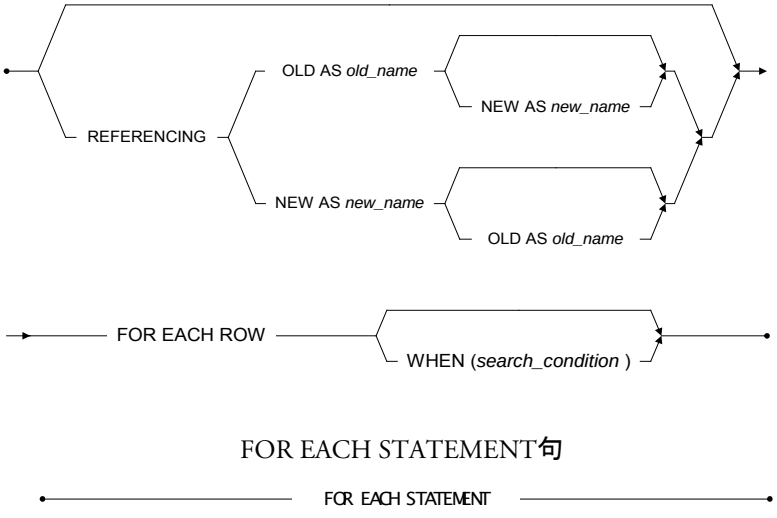


図 10-3 : ALTER TRIGGER 文の構文

トリガーアクションを置き換える

トリガーアクションを置き換える場合、ALTER TRIGGER トリガー名 REPLACE WITH 文を使います。

➲ **例**

マネージャが退職する場合、表 **tb_staff** からマネージャのデータを削除すると共に、表 **tb_manager** からも同じデータを削除するトリガーを作成する:

```
dmSQL> CREATE TRIGGER tr_staff_del AFTER DELETE ON tb_staff
                            FOR EACH ROW
                            ( DELETE FROM tb_manager WHERE pId = old.Id );
```

「そのマネージャがプロジェクトマネージャの時のみ **tb_manager** 表から削除する」といった条件をトリガーアクションに加えるように、トリガーを修正する:

```
dmSQL> ALTER TRIGGER tr_staff_del REPLACE WITH AFTER DELETE ON tb_staff
                           FOR EACH ROW
                           ( DELETE FROM tb_manager
                             WHERE Id = old.Id
                             AND title = 'Project Mananger');
```

修正する替わりに、トリガーを削除して再作成することもできます。

10.5 トリガーを削除する

トリガーをデータベースから削除するには、DROP TRIGGER 文を使用します。

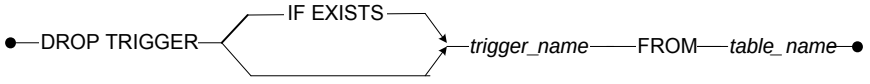


図 10-4 : DROP TRIGGER 文の構文

トリガーを削除する

表を削除すると、その表を参照しているトリガーに影響します。表スキーマが変更された場合、次にトリガーを実行するときは、新しい表定義に合せようとしますが、イベントあるいはアクションで指定するカラムが消失していると、トリガーを実行することはできず、トリガーを誘発する SQL 文も失敗します。この場合は、トリガーを削除するか、トリガー定義を新しい表定義に合せて修正します。トリガーを削除するためには、削除するトリガー名と関連付けられている表名を指定します。

➲ **例 1**

myTable 表から DROP コマンドを使用してトリガー**myTrigger** を削除する:

```
dmSQL> DROP TRIGGER myTrigger FROM myTable;
```

➲ **例 2**

command: **myTable** 表から DROP IF EXISTS コマンドを使用してトリガー **myTrigger** を削除する

```
dmSQL> DROP TRIGGER IF EXISTS myTrigger FROM myTable;
```

➲ **例 3**

表 tb_staff にトリガーtr_staff_upd を作成する:

```
dmSQL> CREATE TRIGGER tr1 AFTER UPDATE ON t1
                          FOR EACH ROW ( DELETE FROM t2 WHERE c1 = old.c1 );
```

この場合、表 **salary** のカラム **id** が削除されるか、そのデータ型が変更されると、トリガーを誘発する文（update on **tb_staff**）を実行する時に、トリガー**tr_staff_upd** を起動しようとしますが、実行エラーが発生します。

10.6 トリガーを使用する

本節では、トリガーのいくつかの使用方法について説明します。

ストアド・プロシージャをアクションに使用する

トリガーの最も強力な機能は、ストアド・プロシージャをトリガーアクションで呼び出せることです。EXECUTE PROCEDURE 文は、トリガー表のデータをストアド・プロシージャに渡すことができます。

➲ **例**

トリガーを作成し、EXECUTE PROCEDURE を使う:

```
dmSQL> CREATE TRIGGER tr_sales AFTER UPDATE OF price ON tb_Sales
                                FOR EACH ROW
                                (EXECUTE PROCEDURE
                                logPrice(item_no, new.price, old.price));
```

ユーザーは、引数リストでストアド・プロシージャに値を渡すことができます。行トリガーのアクションからストアド・プロシージャを呼び出すときは、カラムの値を渡すのに OLD と NEW の相関名を使用することができます。文トリガーのアクションからストアド・プロシージャを呼び出すときは、定数値のみ渡すことができます。

ストアド・プロシージャを使うかどうか関わらず、トリガーアクションでトリガー表のトリガーカラム以外のカラムを更新することができます。トリガーによって起動するストアド・プロシージャに、BEGIN WORK、

COMMIT WORK、ROLLBACK WORK、SAVEPOINT、DDL 文のようなトランザクション制御文を使用できません。

トリガーアクションとしてのストアド・プロシージャを、複数行を返すカーソルを処理するプロシージャにすることはできません。

トリガーの実行順序

トリガーの実行順序は、トリガーカラムのカラム番号によって決められます。トリガーカラムリストの中で一番小さいカラム番号をもつトリガーから起動し始めます。次の例では、a=column1、b=column 2、c=column 3、d= column 4 です。

➲ 例

SQL 文 UPDATE t1 SET b=b+1, c=c+1 は、トリガー***trig1*** と ***trig2*** を起動させます。トリガーカラム ***a***、***c*** をもつトリガー***trig1*** の方が、トリガーカラム ***b***、***d*** をもつトリガー***trig2*** よりも小さいカラム番号をもっているので、トリガー***trig2*** の前にトリガー***trig1*** が起動します。

```
dmSQL> CREATE TRIGGER trig1 AFTER UPDATE OF a,c ON t1
                            FOR EACH STATEMENT (UPDATE t2 set c1=c1+1);

dmSQL> CREATE TRIGGER trig2 AFTER UPDATE OF b,d ON t1
                            FOR EACH STATEMENT (UPDATE t2 set c2=c2+1);
```

セキュリティとトリガー

トリガーを起動させるユーザーには、トリガーイベントを実行する権限が必要です。さもないと、トリガーを引き起こすことができません。但し、トリガーアクションを実行する権限は必要ありません。何故ならば、トリガーアクションにある SQL 文は、トリガーを引き起こすユーザーの権限ではなく、トリガー所有者の権限により作動するからです。トリガーが作成されると、トリガーを誘発するトリガーイベントを実行できるユーザーであれば、トリガーを実行することができます。

➲ 例

ユーザー*B*は表 *T1* と *T2* の両方を更新でき、ユーザー*A* は *T1* のみ更新することができ *T2* は更新できないとします。ここで、ユーザー*B* が表 *T1* に UPDATE トリガーを作成し、トリガーアクションで *T2* を更新するとします。ユーザー*A* が *T1* を更新すると、トリガーアクション（*T2* の更新）は正常に実行されます。何故ならば、トリガーアクションはユーザー*B* の権限で起動するからです。このセキュリティ規則は、トリガーの使用者にトリガーアクションの実行権限を求める規則より、より単純であり合理的です。

カーソルとトリガー

カーソル内の UPDATE および DELETE 文は、単独の UPDATE および DELETE 文とは異なる動きをします。全てのトリガーは、WHERE CURRENT OF 句が付いた、各 UPDATE 文または DELETE 文で実行されます。

例えば、カーソルに対して 4 行が変更された場合、BEFORE/FOR EACH STATEMENT、BEFORE/FOR EACH ROW、AFTER/FOR EACH STATEMENT、AFTER/FOR EACH ROW トリガーは、各行毎に 1 回、計 4 回実行されます。

トリガーのカスケード

トリガーの実行により、さらに別のトリガーが誘発される場合があります。また、参照整合性を確かにするためにカスケードを利用することもあります。DBMaster では、最大 64 までトリガーをカスケードすることができます。

➲ 例

tb_customer 表から顧客を削除したときに、顧客関連のレコードを *tb_order* 表から削除するトリガーを実行、カスケードして販売関連のレコードを *tb_item* 表から削除するトリガーを引き起こします。

```
dmSQL> CREATE TRIGGER tr cas1 AFTER DELETE ON tb customer
                          FOR EACH ROW
                          (DELETE FROM orders WHERE cust num = old.cust num);

dmSQL> CREATE TRIGGER tr cas2 AFTER DELETE ON tb orders
                          FOR EACH ROW
                          (DELETE FROM tb_items WHERE order_num = old.order_num);
```

カスケードが循環するトリガーを作成した場合でも、作成時にはエラーになりませんが、トリガーがカスケードレベルの最大限度まで実行された時に、エラーになります。

10.7 トリガーをイネーブル／ディセーブルにする

作成したトリガーは、イネーブル(使用可能)・モードになっています。つまり、イベントが発生するとトリガーアクションが実行されます。

次のような場合に、トリガーをディセーブル(使用不可能)にします。

- 大量のデータをロードする時に、トリガーを一時的にディセーブルにしてロード操作をスピードアップする。
- トリガーが参照しているオブジェクトが、使用できない。

➲ **例 1**

表 ***Mytable*** のトリガー***Mytrigger*** をディセーブルにする：

```
dmSQL> ALTER TRIGGER Mytrigger ON Mytable DISABLE;
```

➲ **例 2**

表 ***Mytable*** のトリガー***Mytrigger*** をイネーブルにする:

```
dmSQL> ALTER TRIGGER Mytrigger ON Mytable ENABLE;
```

要約すると、トリガーには 2 つのモードがあります：

- **イネーブル**—トリガー作成時のモードです。トリガーイベントが発生すると、トリガーアクションが実行されます。
- **ディセーブル**—このモードでは、トリガーイベントが発生してもトリガーアクションは実行されません。

10.8 トリガー作成の必要権限

表にトリガーを作成するには、表の所有者であるか、DBA 権限をもっている必要があります。トリガー作成者には、トリガー定義で参照する全てのオブジェクトを作成することができる権限が必要です。

DBMaster では、トリガー自身には所有者が無く、トリガーは表に関連付けられています。表の所有者または DBA は、表に関連付けられたトリガーに対する全ての権限をもっており、トリガーを作成、削除、変更することができます。

トリガーアクション内の SQL 文は、トリガー文を実行するユーザーの権限ドメインではなく、トリガー作成者の権限ドメインで作動します。

11 ストアド・コマンド

ストアド・コマンドは、データベースに格納されているコンパイル済みの SQL DML 文です。実行形式になっているので、同じ SQL 文を何度もコンパイル、最適化することなく実行することができます。頻繁に使用する SQL 文のストアド・コマンドを作成しておき、繰り返しストアド・コマンドを実行することによって、より良いパフォーマンスを得ることができます。ストアド・コマンドは、1 つの SQL 文から作られているプログラムロジックをもたないストアド・プロシージャとみなすこともできます。

11.1 ストアド・コマンドを作成する

ストアド・コマンドの作成には、CREATE COMMAND 文を使用します。

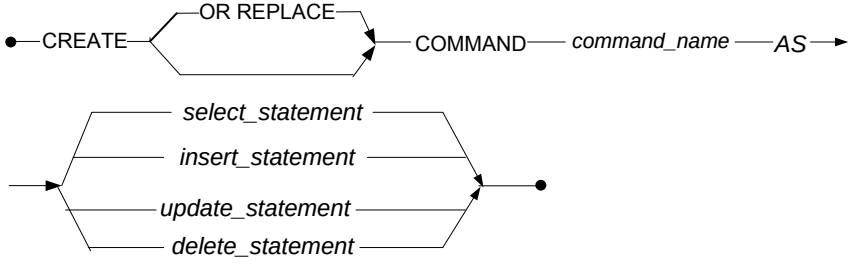


図 11-1：CREATE COMMAND 文の構文

OR REPLACE: OR REPALCE で既存なトリガーを再び作成します。この句を使用して削除しないで既存なトリガーの定義を変更でき、このトリガーは再び作成できます。

作成するストアド・コマンドの SQL 文に、入力パラメタを使用することができます。入力パラメタの値は、ストアド・コマンドを実行するときに与えます。

➲ **例 1**

同じ SQL DML 文のストアドコマンド **sc_student_insert** を作成するため、**tb_student (id** INT, **score** INT, **name** CHAR(32)**)** と定義される表を使用します：

```
dmSQL> INSERT INTO tb_student VALUES (1, ?, ?);
```

➲ **例 2**

そして、CREATE COMMAND を使用します：

```
dmSQL> CREATE COMMAND sc_student_insert AS INSERT INTO tb_student VALUES (1, ?, ?);
```

➲ **例 3**

CREATE OR REPLACE COMMAND を使用します：

```
dmSQL> CREATE OR REPLACE COMMAND sc student insert AS INSERT INTO tb student VALUES
(1, ?, ?);
```

➲ **例 4**

その他の DML 文のストアド・コマンドを作成する：

```
dmSQL> CREATE COMMAND sc student select AS SELECT id,name FROM tb student;
dmSQL> CREATE COMMAND sc_student update AS UPDATE tb_student SET id = id+1 WHERE
score> ?;
dmSQL> CREATE COMMAND sc_student delete AS DELETE FROM tb_student WHERE score > ?;
```

ストアド・コマンドを作成すると、dmSQL またはアプリケーション・プログラムの中で直接コマンドを実行することができます。入力パラメータのあるストアド・コマンドを実行するときは、パラメータマーク（?）、定数、NULL、DEFAULT、引数のない組み込み関数をパラメータ値に指定します。

このとき、ストアド・コマンドの入力パラメータの個数と与える個数は必ず同じでなければなりません。

11.2 ストアド・コマンドを実行する

EXECUTE COMMAND 文を使用してストアド・コマンドを実行します。

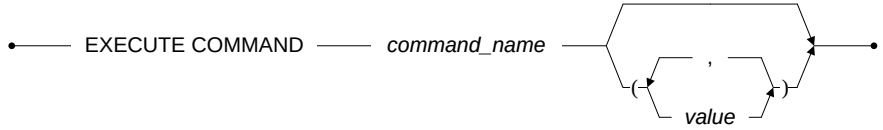


図 11-2 : EXECUTE COMMAND 文の構文

➲ **例 1**

```
dmSQL> EXECUTE COMMAND sc_student_insert (200, 'john') ;
```

➲ **例 2**

```
dmSQL> EXECUTE COMMAND sc_student_insert (DEFAULT, ?);
```

➲ **例 3**

```
dmSQL> EXECUTE COMMAND sc_student_insert (?, NULL);
```

➲ **例 4**

```
dmSQL> EXECUTE COMMAND sc_student_insert (?, ?);
```

11.3 ストアドコマンドを再構造する

ストアド･コマンドを再構造するため REBUILD COMMAND を使用します。

図 11-3 REBUILD COMMAND 文の構文

➲ **例**

```
dmSQL> REBUILD COMMAND sc_student_insert;
```

11.4 ストアド・コマンドを削除する

不要になったストアド・コマンドは、DROP COMMAND 文を使用して削除することができます。

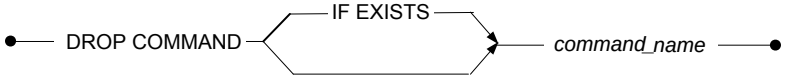


図 11-4 : DROP COMMAND 文の構文

➲ **例 1**

```
dmSQL> DROP COMMAND sc_student_insert;
```

➲ **例 2**

```
dmSQL> DROP COMMAND IF EXISTS sc_student_insert;
```

11.5 ストアド・コマンドのセキュリティ

ストアド・コマンドのセキュリティは、他のスキーマ・オブジェクトと同様に取り扱われます。ストアド・コマンドを作成して使用するときは、セキュリティとオブジェクト権限を考慮しなければなりません。

RESOURCE 権限以上のユーザーのみ、ストアド・コマンドを作成することができます。また、実行権限をもつ SQL DML 文のみをストアド・コマンドにすることができます。ストアド・コマンドは、作成者が所有します。

➲ **例**

リソース権をもつユーザー**joe** がストアド･コマンド **sc_CheckDate** を作成する。

```
dmSQL> CREATE COMMAND sc_CheckDate AS SELECT FirstName, LastName, Hiredate FROM
SYSADM.Employee WHERE HireDate > '1995-01-01';
```

employee 表は **SYSADM** が所有しているので、システム管理者はユーザー**joe** がストアド・コマンドを作成する前に、ユーザー**joe** に **SYSADM. employee** 表に対する選択権限を与えなければいけません。

ストアド・コマンドを実行するためには、実行権限が必要です。ストアド・コマンドを使用するユーザーに、ストアド・コマンドの実行権限を与えます。ただし、必要な権限を有するユーザー（DBA、SYSDBA 、SYSADM、ストアド・コマンドの所有者、実行権限が与えられたユーザー）のみストアド・コマンドの実行権限を与えたり、取り消したりすることができます。

DBA は、全てのストアド・コマンドを実行する権限を持っています。ストアド・コマンドの所有者は、ストアド・コマンドの実行権限、他のユーザーに実行権限を与える/取り消す権限をもっています。

実行権限を与える

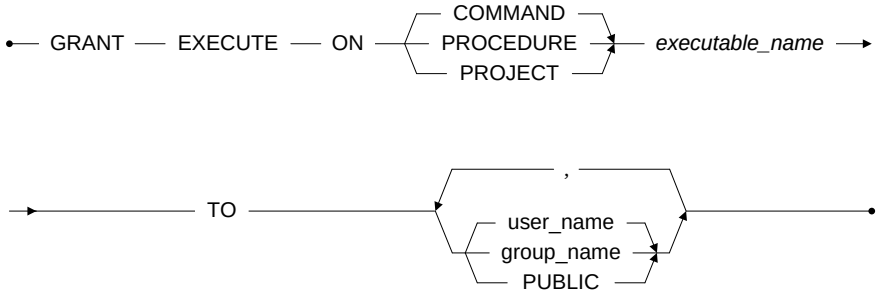


図 11-5 : GRANT EXECUTE 権限の構文

➲ **例**

sc_student_insert のコマンドを実行する権限(EXECUTE 権限)を John に与える：

```
dmSQL> GRANT EXECUTE ON COMMAND sc_student_insert TO John;
```

実行権限を取り消す

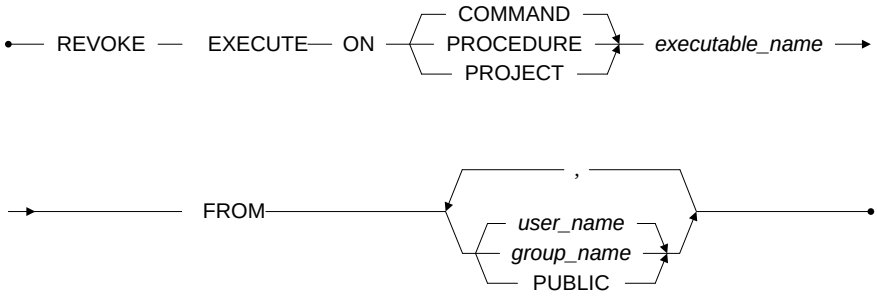


図 11-6 : REVOKE EXECUTE 権限の構文

➲ **例**

ユーザーJohn のストアド・コマンド sc_student_insert の実行権限を取り消す:

```
dmSQL> REVOKE EXECUTE ON COMMAND sc_student_insert FROM John;
```

11.6 ストアド・コマンドのライフサイクル

ストアド・コマンドで参照する表が削除または変更されると、ストアド・コマンドは不正になります。不正になったストアド・コマンドは、使用することができません。以前記述した古いカラム情報を参照するプログラムは、実行時に予想できない結果を引き起こします。

ストアド・コマンドの利点は、SQL コマンドを繰り返し実行する際のパフォーマンスを向上させることにあります。DBMaster は、UPDATE STATISTICS などで、実行計画が更新されているかどうかをチェックします。UPDATE STATISTICS コマンドを実行すると、ストアド・コマンドの全実行計画は、より良いパフォーマンスを達成するために更新されます。

11.7 ストアド・コマンドの情報を取得する

システム表 SYSCMDINFO からストアド･コマンドの情報を取得することができます。次の SQL 文でストアド・コマンドの情報を取得することができます。

```
dmSQL> SELECT * from SYSCMDINFO;
```

12 ストアド・プロシージャ

ストアド・プロシージャは特殊なユーザー定義関数です。DBMaster はストアド・プロシージャで三種類の言語 : ESQL/C、Java と SQL をサポートします。ストアド・プロシージャは、一度作成されると、実行形式のデータベース・オブジェクトとしてデータベースに保存されます。SQL 文を何度もコンパイルし最適化しなくて済むので、頻繁に繰り返される仕事の効率を上げることができます。ストアド・プロシージャは、会話型 SQL 文としてを実行することができ、また、アプリケーション・プログラム、トリガー、他のストアド・プロシージャから呼び出すことができます。

ストアド・プロシージャを利用することによって、データベースのパフォーマンス改善、アプリケーションの単純化、データベース・アクセスの制限と監視等の広範囲の目的を達成することができます。

ストアド・プロシージャは実行可能オブジェクトとしてデータベースに保存されているので、データベース上で利用される全てのアプリケーションが使用することができます。複数のアプリケーションが同じストアド・プロシージャを使用することができるので、アプリケーションの開発期間を短縮することもできます。

12.1 ESQL ストアドプロシージャ

ESQL ストアドプロシージャは ESQL/C プログラムです。ストアド・プロシージャは、C 関数やシステムコールの呼び出しなど、C アプリケーションで可能な機能を実行することができます。

ストアド・プロシージャの ESQL/C プログラムは、CREATE PROCEDURE 文、宣言セクション（必要な場合）、コードセクションから構成されます。プログラムにホスト変数が使われていない場合、宣言セクションは省略することができます。

➲ 例

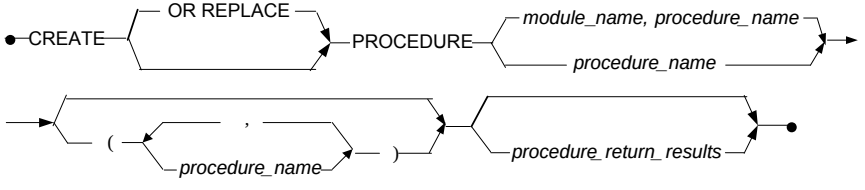
１個の入力パラメータ、１個の出力パラメータ、戻り値（STATUS）のあるストアドプロシージャ **sp_Aphone** を作成する:

```
EXEC SQL CREATE PROCEDURE sp_Aphone (CHAR(13) name, CHAR(13) phone OUTPUT)
         RETURNS STATUS;
{
    EXEC SQL BEGIN CODE SECTION;

        EXEC SQL SELECT PHONE FROM TBL WHERE NAME = :name INTO :phone;

        EXEC SQL RETURNS STATUS SQLCODE;

    EXEC SQL END CODE SECTION;
}
```

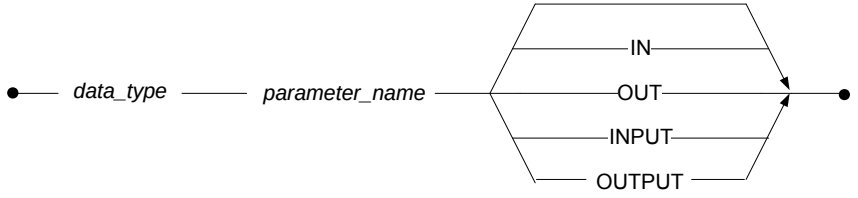
このプログラムの構造は、以下の節で説明します。

CREATE PROCEDURE構文

プロシージャ定義の最初に、CREATE PROCEDURE 文を記述します。



<プロシージャ・パラメータ> 句



<プロシージャ戻り値> 句

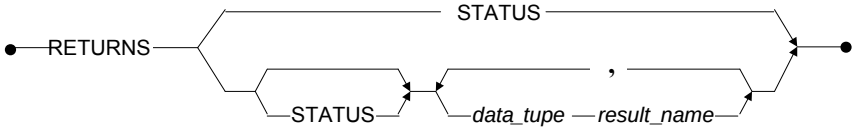


図 12-1：CREATE PROCEDURE 文の構文

OR REPLACE: OR REPLACE で既存なプロシージャを再び作成します。この句を使用して削除しないで既存なプロシージャの定義を変更でき、このプロシージャは再び作成できます。

プロシージャはアボートする場合、以前のプロシージャはもう削除したことを注意してください。

➲ 例

プロシージャ文作成の例:

```
dmSQL> CREATE PROCEDURE sp example1 (INTEGER n IN) RETURNS STATUS;
dmSQL> CREATE PROCEDURE sp example2 (INTEGER n1 IN, INTEGER n2 OUTPUT) RETURNS
CHAR(12) nm;
dmSQL> CREATE PROCEDURE sp example3 (CHAR(10) par1 OUTPUT, SMALLINT par2)
       RETURNS STATUS, TIMESTAMP ret1, FLOAT ret2;
dmSQL> CREATE OR REPLACE PROCEDURE sp example4(CHAR(10) par1 OUTPUT, SMALLINT par2)
RETURNS STATUS, TIMESTAMP ret1, FLOAT ret2;
```

CREATE PROCEDURE 文では、プロシージャ名と I/O パラメータの名前と種類を必ず指定します。

パラメータを使用する

パラメータが必要なときは、括弧内にパラメータのタイプと名前のリストを与えます。パラメータ名の後に、IN/OUT（または INPUT/OUTPUT）のパラメータ属性を付けます。パラメータ属性を指定しない場合、IN が初期設定値として使用されます。入力パラメータは、プロシージャに値を渡すときに使用します。例 1 のプロシージャには、入力パラメータ ***name*** があります。プロシージャを実行するときは、入力パラメータの値を与えなければなりません。

出力パラメータは、プロシージャ実行後の結果（結果セットではなく）を取得するときに使用します。例 1 のプロシージャ ***a_phone*** には、出力パラメータ ***phone*** があります。出力結果を受け取るために、出力パラメータにバッファを割り当てます。例 1 のプロシージャを実行すると、入力パラメータで指定した人の電話番号をバッファから取得することができます。

ストアド・プロシージャは、結果セットの行をデータベースから検索することもできます。この場合は、結果リストを指定します。プロシージャが検索結果を返さない場合は、結果リストは必要ありません。結果リストは、RETURNS キーワードと、その後に続くタイプと名前から成ります。

STATUS キーワードは、プロシージャ実行後に戻される整数値を表示するために使用します。

➲ 例

入力パラメータと値を使ってプロシージャを作成する：

```
EXEC SQL CREATE PROCEDURE sp Select (FLOAT ifl) RETURNS STATUS,
                                                  FLOAT fl,
                                                 DOUBLE db;
{
    EXEC SQL BEGIN CODE SECTION;
    EXEC SQL RETURNS STATUS SQLCODE;
    EXEC SQL RETURNS SELECT fl, db FROM t8 WHERE fl < :ifl into :fl, :db;
    EXEC SQL END CODE SECTION;
```

```
}
```

DBMaster では現在、次のデータ型の入出力パラメータを使用することができます。INTEGER、SMALLINT、CHAR()、DATE、TIME、TIMESTAMP、FLOAT、DOUBLE。

RETURN SELECT文

プロシージャは、結果セットを返すことができます。ストアド・プロシージャ実行の呼び出し側に結果セットを引き渡すために、ホスト変数を使用します。ストアド・プロシージャのコードに、RETURNS キーワードを指定すると、プリプロセッサが C コードに関するホスト変数を生成します。RETURNS キーワードは、結果セットを作り出す select 文の前に置きます。

➲ **例**

この例は CREATE PROCEDURE 文と SELECT 文の二個所に RETURNS があり、互いに対になっています。結果セットを返す場合は、CREATE PROCEDURE 文で RETURNS を用いて出力パラメータを宣言し、SELECT 文の前に RETURNS キーワードを置きます。

```
EXEC SQL CREATE PROCEDURE  sp Allphone RETURNS CHAR(12) name, CHAR(12) phone;
{
    EXEC SQL BEGIN CODE SECTION;
    EXEC SQL RETURNS SELECT NAME, PHONE FROM TBL INTO :name, :phone;
    EXEC SQL END CODE SECTION;
}
```

モジュール名

ストアド・プロシージャを作成すると、プロシージャ名と所有者名がその初期設定ダイナミック・リンク・ライブラリ名となります。作成者は、そのプロシージャ名のみを使って、呼び出し、削除をすることができます。完全なプロシージャ名(*所有者.プロシージャ名)*を使うと、いずれのユーザーもプロシージャを呼び出すことができます。

ユーザーは、初期設定のダイナミック・リンク・ライブラリ名以外の名前を使用するために、CREATE PROCEDURE 文の中にモジュール名を指定することもできます。この場合、プロシージャを作成したユーザーであって

も、完全なプロシージャ名(*モジュール名.所有者.プロシージャ名*)でプロシージャを呼び出し、削除する必要があります。

変数宣言

ストアド・プロシージャのホスト変数は、ESQL/C と同じ方法で宣言します。宣言セクションは、ESQL/C プログラムの中では無く、コード・セクションの前に置きます。C 変数は、宣言セクションの前後いずれに置くことも可能ですが、必ずコード・セクションの前に置きます。

コードセクション

変数宣言以外の全ての文は、コードセクションに置きます。コードセクションの前に宣言以外の文を置くと、コンパイルエラーや不正な結果を招きます。コードセクションの後に置かれた文は実行されません。

ストアド・プロシージャの環境設定

ストアド・プロシージャを作成すると、対応するダイナミック・リンク・ライブラリがサーバーに生成され保存されます。ライブラリ・ファイルは、初期設定で DBMaster サーバーの作業ディレクトリに配置されます。環境設定ファイルのキーワード **DB_SPDir** に、ストアド・プロシージャのライブラリ・ファイルを配置するパスを指定することができます。

ストアド・プロシージャを作成/実行する際にデータベース・サーバーから送信された、エラーメッセージ・ファイルとトレース・ログファイルを受け取るために、クライアント側のキーワード **DB_SPLog** にディレクトリを指定します。

➲ **例 1**

dmconfig.ini ファイルで、ストアド・プロシージャのダイナミック・リンク・ライブラリのためのパスを、**c:\usr\jerry\data\SP** にセットする:

```
DB_SPLOG=c:\usr\jerry\data\SP
```

➲ **例 2**

dmconfig.ini ファイルで、ストアド・プロシージャ・ログファイルのディレクトリを **c:\usr\jerry\data\SP** にセットする:

```
DB_SPLOG=c:\usr\jerry\data\SP
```

ファイルからストアド・プロシージャを作成する

まず、ストアド・プロシージャを記述し、それをファイルに保存します。この新規ストアド・プロシージャを dmSQL か JDBATool のようなツールを使ってデータベースに挿入します。

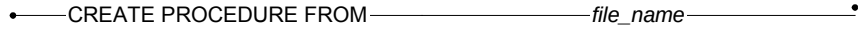


図 12-2：CREATE PROCEDURE FROM<ファイル名>文の構文

➲ **例**

ファイルからストアド・プロシージャを作成する：

```
dmSQL> CREATE PROCEDURE FROM 'proc1.ec';
dmSQL> CREATE PROCEDURE FROM '.\esql\sp\proc2.ec';
dmSQL> CREATE PROCEDURE FROM 'c:\users\jerry\sp\proc3.ec';
```

➲ **替わりに JDBATool を使う：**

1. ツリーのストアド・プロシージャをクリックします。
2. ［作成］ボタンをクリックします。［ストアド・プロシージャを作成する］ウィンドウが表示されます。
3. ［インポート］ボタンをクリックします。［開く］ウィンドウが表示されます。
4. サーバーやネットワーク・ドライブにある他のデータベースのSPDIRディレクトリを含むどのようなソースからでもファイルをインポートすることができます。［ファイル名］欄にファイル名を入力、或いはブラウザしてパスを選択して下さい。

注 *インポートするファイルは、C++コードを含むASCII形式のファイルです。*

5. ［開く］をクリックします。

6. インポートするファイルが適切にフォーマットされた(ASCII)テキスト・ファイル場合、［ストアド・プロシージャを作成する］ウィンドウが再び表示されます。別の場所にストアド・プロシージャを保存する場合は[新規保存]をクリック、コンパイルしてデータベースにストアド・プロシージャを保存する場合は、［OK］をクリックします。

   注 *このストアド・プロシージャ作成の際にエラーが発生した場合、ウィンドウ下部に表示されます。*

ストアドプロシージャの実行

ストアド・プロシージャは、dmSQL、C プログラム（ODBC または ESQL）、他のストアド・プロシージャ、トリガーアクションから呼び出すことができます。

DMSQL

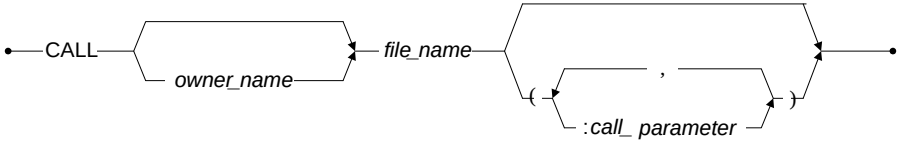


図 12-3 : dmSQL 内の CALL 文の構文

➲ **例 1**

dmSQL でストアド・プロシージャを実行する：

```
dmSQL> CALL sp_example1 (3);                          // プロシージャ p1 を実行
dmSQL> CALL SYSADM. sp_example2 (5, ?);               // SYSADM のプロシージャ p2 を実
行
dmSQL/Val> 100;                                       // パラメータ値を入力
dmSQL> ? = CALL SYSADM. sp_example2 (5, 100);         // プロシージャ p2 を実行
し 、戻り状態を取得
```

➲ **例 2**

プロシージャが結果セットを返す場合、dmSQL は出力パラメタを処理して自動的に画面に表示します。結果セットは、SELECT 文を入力したときと同じように画面に表示されます：

```
dmSQL> call sp_Aphone ('jeff',?);
       PHONE
       ==============
       408-255-2689

dmSQL> call sp_Allphone;
       NAME             PHONE
       ==========================
       Jerry           02-775-8615
       Jeff            408-255-2689
```

ESQL

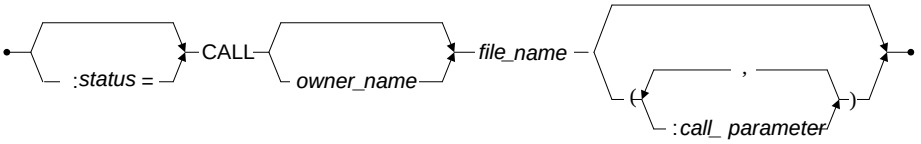


図 12-4：ESQL 内の CALL 文の構文

➲ **例**

ESQL プログラムでストアド・プロシージャを実行する：

```
EXEC SQL :s = CALL sp_example1 (3);
EXEC SQL CALL SYSADM. sp_example2 (5, :n2) INTO :nm;
EXEC SQL :s = CALL jack. sp_example3 (:par1, 7) INTO :ret1, :ret2;
```

ESQL プログラムで使われる構文は、dmSQL に似ています。ステータス、出力パラメータ、結果セット値を受け取るときは、ホスト変数を利用します。

ネストされたストアド・プロシージャを実行する

ESQL/C ストアド・プロシージャ内にあるストアド・プロシージャを ESQL/C プログラムでストアド・プロシージャを実行するのと全く同じ方法で呼び

出すことができます。唯一の例外は、ESQL プログラムでは RETURNS キーワード使用することができませんが、ストアド・プロシージャが他のストアド・プロシージャを呼び出す時に使用できることです。

ストアド・プロシージャ ***sel_all_phone*** が複数行の結果セットを返すとします。ESQL プログラムは、通常カーソルを使用して ***sel_all_phone*** の各行を取り出します。ストアド・プロシージャ ***sp2*** でも同じ方法でデータを取り出し調べることができますが、***sel_all_phone*** の結果セット全体を直接呼び出し側に返すこともできます。

➲ **例**

ストアド・プロシージャ ***sp2*** 内で次の文を実行する：

```
EXEC SQL RETURNS CALL sel_all_phone INTO :oName, :oPhone;
```

ストアド・プロシージャが他のストアド・プロシージャの結果セットを返す場合、呼び出し側は、結果セットと同じ結果リストを指定するか、最初の n 個の結果カラムと同じ結果リストを指定しなければなりません。

ODBCでストアド・プロシージャを実行する

ODBC プログラムからストアド・プロシージャを呼び出すこともできます。この場合、プロシージャのパラメータに対してパラメータをバインドし、プロシージャが返す結果セットに対してカラムをバインドする必要があります。ODBC プログラムでは、結果セットの部分カラムをバインドすることができます。プロシージャ実行後、出力パラメタはホスト変数に返されます。結果セットは、SELECT 文と同様に取得します。

➲ **例 1**

プロシージャ **sp_proc1** の宣言：

```
dmSQL> CREATE PROCEDURE sp proc1 (CHAR(12) p1, CHAR(12) p2 OUTPUT) RETURNS INTEGER
r1;
{
EXEC SQL BEGIN CODE SECTION;
EXEC SQL SELECT c2 FROM t1 WHERE c1 = :p1 INTO :p2;
EXEC SQL RETURNS SELECT c1 FROM t2 INTO :r1;
EXEC SQL END CODE SECTION;
}
```

➲ 例 2

sp_proc1 を呼び出す ODBC プログラム：

```
SQLPrepare(cmdp,(UCHAR*)"call sp_proc1 (?, ?)", SQL_NTS);

strcpy(bpname, "12345");

SQLBindParameter(cmdp, 1, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
                     20, 0, &p1, 20, NULL);
SQLBindParameter(cmdp, 2, SQL_PARAM_INPUT_OUTPUT, SQL_C_CHAR, SQL_CHAR,
                     20, 0, &p2, 20, NULL);

SQLBindCol(cmdp, 1, SQL_C_LONG, &i, sizeof(long), NULL);
SQLExecute(cmdp);                                    /* get p2           */

while ((rc=SQLFetch(cmdp))!=SQL_NO_DATA_FOUND)       /* fetch result set */
```

ストアド・プロシージャ実行をトレースする

DBMaster には、デバッグのためにストアド・プロシージャの実行をトレースするトレース機能があります。

➲ 例

TRACE コマンドを使う：

```
EXEC SQL TRACE ON;                          // トレース開始
EXEC SQL SELECT c1 FROM t1 INTO :var1;
EXEC SQL TRACE ("var1 = %d\n", var1);       // var1 の値をトレース
EXEC SQL TRACE OFF;                         // トレース終了
```

TRACE 機能を ON にし、トレース用の変数を配置し、メッセージを出力します。ストアド・プロシージャの実行後、全トレース情報はクライアント側の環境設定ファイルのキーワード **DB_SPLog** で指定したディレクトリにあるファイル **_spusr.log** に記録されます。

12.2 JAVA ストアドプロシージャ

Java ストアドプロシージを使用しての開発計画は多数あると思います。今日の Java の普及を考えると、開発チームのメンバーが ESQL よりも Java に熟

達しているケースが考えられます。DBMaster は Java ストアドプロシージャをサポートしているので、Java プログラマーは自分の好きな言語でコード化できます。経験のある ESQL 開発者が Java を使うと、Java 言語のメリットを生かして、データベースアプリケーションの機能を拡張できます。また、Java を使うと、既存のコードを再利用して、生産性を劇的に拡大できます。

更に、DBMaster は Java ストアドプロシージャとしての Java メソッドをサポートしています。DBMaster は DriverManager.getConnection(url, …) の URL 引数を **jdbc:default:connection** に置き換えます。すべての java クラスを使えば、すべての JDBC クラスを含む Java ストアドプロシージャとして Java メソッドを実行できます。関連する.jar ファイルを登録し、Java ストアドプロシージャを作成、実行、削除する新しい構文を備えています。

CREATE JAVA ストアドプロシージャ構文は以下のです：

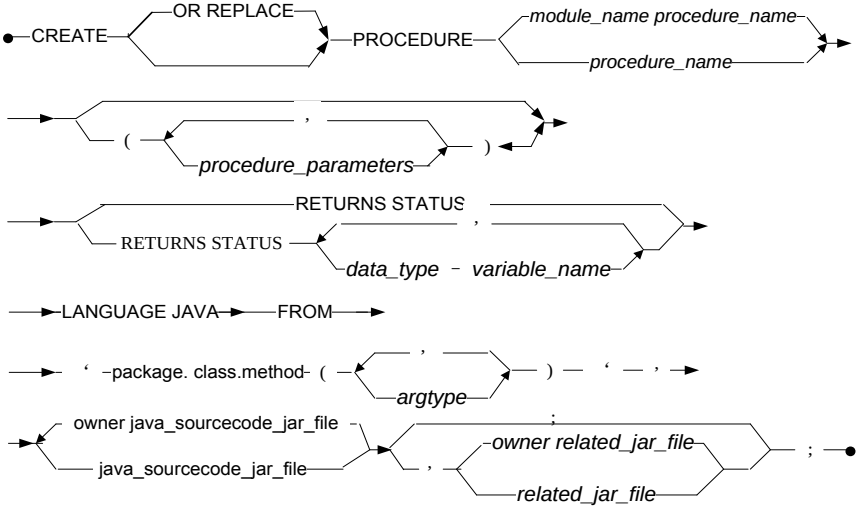


図 12-5：CREATE JAVA STORED PROCEDURE 文の構文

➲ 例

DBMaster は/home/usr/mary/sp/aa.jar の java メソッド xx.yy.AA(文字列) をサポートしていますが、メソッド AA(文字列)を実行するにはやはり /home/usr/john/sp/bb.jar が必要です。

例えば、aa.jar ファイルと bb.jar ファイルをデータベースに登録するには、最初にユーザーsysadm の/home/usr/mary/sp/aa.jar を /home/sysadm/spdir/jar/SYSADM/に置かなければなりません。

DB_SPDIR は dmconfig.ini で"/home/sysadm/spdir"と定義されるので、ユーザーは jar ファイルをデータベースに追加する前に、jar ファイルを DB_SPDIR/jar/uppercase_username/ディレクトリに移動しなければなりません。

➲ 例

aa.jar にはメソッド xx.yy.AA(文字列)がありますが、メソッド AA(文字列)を実行するにはやはり bb.jar が必要です。

aa.jar ファイルと bb.jar ファイルをデータベースに登録する方法：

```
>>ADD JARFILE xaa aa.jar;
>>ADD JARFILE xbb bb.jar;
```

注 *ユーザーは"add jarfile"文字列を実行する前に、物理的ファイル、すなわち aa.jarおよび bb.jar,をディレクトリDB_SPDIR/jar/uppercase_username/に移動させなければなりません。DB_SPDIR はdmconfig.iniのDBMasterストアドプロシージャのディレクトリを定義するキーワードです。*

AA(文字列)メソッドによる java ストアドプロシージャ"JSP_AA(char(10) par1) " の作成方法：

```
>>CREATE PROCEDURE JSP AA (char(10) par1) RETURNS STATUS LANGUAGE JAVA FROM
xx.yy.AA(String), xaa, xbb;
```

java ストアドプロシージャ"JSP_AA"の実行方法：

```
>>EXECUTE PROCEDURE JSP_AA ('aaaaaa');
>>CALL JSP_AA ('bbb');
```

java ストアドプロシージャ"JSP_AA"の削除方法：

```
>>DROP PROCEDURE JSP_AA;
```

データベースから aa.jar ファイルと bb.jar ファイルの登録を解除する方法：

```
>>REMOVE JARFILE xaa;
>>REMOVE JARFILE xbb;
```

➲ JAVA の関連したプロシージャ作成文字列

データベースに.jar ファイルを登録する方法：

```
ADD JARFILE logical_file_name physical_jarfile_name;
```

データベースから.jar ファイルの登録を解除する方法：

```
REMOVE JARFILE logical_file_name;
```

CREATE PROCEDURE コマンドで java ストアドプロシージャを作成します：

```
CREATE  [OR REPLACE] PROCEDURE procedure-name
      [(procedure-parameter [, procedure-parameter ...])]
      {
        [RETURNS STATUS]
        | [RETURNS [STATUS,] procedure-result [,procedure-result ...]]
         }
 LANGUAGE JAVA FROM  'package.class.method([ ' argtype[,argtype…] ' ])' ,
[owner.]java-sourcecode-jar-file [, owner.related-jar-file];
```

OR REPLACE: OR REPLACE で既存なプロシージャを再び作成します。この句を使用して削除しないで既存なプロシージャの定義を変更でき、このプロシージャは再び作成できます。

java ストアドプロシージャの作成方法：

```
CREATE PROCEDURE procedure-name
      [(procedure-parameter [, procedure-parameter ...])]
      {
        [RETURNS STATUS]
        | [RETURNS [STATUS,] procedure-result [,procedure-result ...]]
         }
 LANGUAGE JAVA FROM  'package.class.method([ ' argtype[,argtype…] ' ])' ,
[owner.]java-sourcecode-jar-file [, owner.related-jar-file];
```

java ストアドプロシージャの実行方法：

```
EXECUTE PROCEDURE [owner.]procedure-name;
EXECUTE PROC [owner.]procedure-name;
[? =] CALL  [owner.]procedure-name  [(procedure-parameter-value [,
procedure-parameter-value ...])]
```

java ストアドプロシージャの削除方法：

```
DROP PROCEDURE [owner.]procedure-name;
```

プロシージャのロード/アンロード方法：

```
UNLOAD PROCEDURE/PORC FROM [owner_patt.]proc_patt TO unload_filename;
LOAD PROCEDURE/PORC FROM unload_filename;
```

jar ファイルのロード/アンロード方法：

```
UNLOAD JARFILE FROM [owner_patt.]jarfile_patt TO unload_filename;
LOAD JARFILE FROM unload_filename;
```

注 *ユーザーはjarファイルをロードする前に、物理的 jarファイルを新 DB_SPDIR/jar/uppercase_username/ディレクトリに移動しなければなりません。*

JAVAストアドプロシージャの実行

Java ストアドプロシージャを使用する説明は、以下の例を使用して提供されます。

➲ **例 1 (INPUT パラメータ) :**

Java ストアドプロシージャを使用してテーブル tb_staff に 1 つのタプルを挿入します。

1. javaメソッドaddEmployee(int,String) を作成して、テーブルtb_staffに1つのタプルを挿入します。その後、Javaメソッドをコンパイルして、クラスをAA.jarファイルにzipします。

```
public static void addEmployee(int empid, String name)
{
        Connection conn = DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement pstmt = conn.prepareStatement("insert into tb staff
values(?,?)");
        pstmt.setInt(1, empid);
        pstmt.setString(2, name);
        pstmt.execute();
}
```

2. 以下のメソッドのJavaストアドプロシージャを作成します：
addEmployee(int,String)

SQL を実行して新規ジャーファイルを追加します：

```
dmSQL> ADD JARFILE logical_AA AA.jar;
```

以下の SQL に一つを実行して Java ストアドプロシージャを作成します：

```
dmSQL> CREATE PROCEDURE JSP_addEmp (int id, char(10) name) RETURNS STATUS LANGUAGE
JAVA FROM 'xx.yy.addEmployee(int,String)', logical_AA;
```

或いは：

```
dmSQL> CREATE OR REPLACE PROCEDURE JSP_addEmp (int id, char(10) name) RETURNS STATUS
LANGUAGE JAVA FROM 'xx.yy.addEmployee(int,String)', logical_AA;
```

3. Javaストアドプロシージャを実行します。

SQL を実行してスト Java SP を実行します：

```
dmSQL> EXECUTE PROCEDURE JSP_addEmp(1234, 'jeff');
```

➲ **例 2 (OUTPUT パラメータ)：**

Java アドプロシージャを使って **tb_staff** から述語 (empid) を伴う従業員名を 1 つ選択します。

1. Javaメソッドone Employee(int,byte[]) を作成して、テーブル**tb_staff**から述語 (emp id) を伴う従業員名を1つ入手します。その後、Javaメソッドをコンパイルして、クラスをBB.jarファイルにzipします。

```
public static void oneEmployee(int id, byte[] name)
{
        Connection conn = DriverManager.getConnection("jdbc:default:connection");
        PreparedStatement pstmt = conn.prepareStatement("select name from tb_staff
        where id = ? ");
        pstmt.setInt(1, id);
        ResultSet rs = pstmt.executeQuery();
        Rs.next();
        String empName = rs.getString(1);
        Name = empName.getBytes();
}
```

2. 以下のメソッドのJavaストアドプロシージャを作成します:
oneEmployee(int,String)

SQL を実行して NEW のジャーファイルを追加します：

```
dmSQL> ADD JARFILE logical_BB BB.jar;
```

SQL を実行してスト Java SP を作成します：

```
dmSQL> CREATE PROCEDURE JSP_oneEmp (int id, char(10) name OUTPUT) RETURNS STATUS
LANGUAGE JAVA FROM 'xx.yy.oneEmployee(int,byte[])', logical_BB;
```

3. Javaアドプロシージャを実行します。

SQL を実行してスト Java SP を実行します：

```
dmSQL> EXECUTE PROCEDURE JSP_oneEmp(1234, ?);
```

➲ **例 3 (結果セット)：**

Java ストアドプロシージャを使用してテーブル EMPLOYEE から 1 つの結果セットを選択します。

1. Javaメソッド rsEmployee() を作成してテーブル**tb_staff**から従業員名を1つ入手します。その後、Javaメソッドをコンパイルして、クラスをCC.jarファイルにzipします。

```
public static ResultSet rsEmployee()
{
        Connection conn = DriverManager.getConnection("jdbc:default:connection");
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("select id, name from tb staff");
        Return rs;
}
```

2. 以下のメソッドのJavaストアドプロシージャを作成します:
rsEmployee(int,String)

SQL を実行して NEW のジャーファイルを追加します：

```
dmSQL> ADD JARFILE logical_CC CC.jar;
```

SQL を実行して Java ストアドプロシージャを作成します：

```
dmSQL> CREATE PROCEDURE JSP_rsEmp RETURNS STATUS, int outId, char(10) outName LANGUAGE
JAVA FROM 'xx.yy.rsEmployee()', logical_CC;
```

3. Javaアドプロシージャを実行します。

SQL を実行して Java ストアドプロシージャを実行します：

```
dmSQL> EXECUTE PROCEDURE JSP_rsEmp();
```

4. 一般のフェッチ (または拡張フェッチ) メソッドを使って結果セットをフェッチします。

入力/出力引数

DBMaster のストアドプロシージャ入力/出力引数は以下のデータ型をサポートしています：

BINARY	VARCHAR
CHAR	FLOAT
REAL	DOUBLE
SMALLINT	INTEGER
TIMESTAMP	DATE
TIME	DECIMAL

これらは DBMaster がサポートしている 7 つの基本的な Java 型および配列です：

JAVA TYPE	ARRAY
byte	byte [ ]
short	short [ ]
int	int [ ]
long	long [ ]
float	float [ ]
double	double [ ]
char	char [ ]

以下のクラスもサポートされています：

JAVA CLASS	ARRAY
Byte	Byte[ ]
Short	Short[ ]
Integer	Integer [ ]
Long	Long[ ]

Float	Float[ ]
Double	Double[ ]
Character	Character[ ]
String	String[ ]

12.3 SQL ストアド・プロシージャ

ストアドプロシージャを作成するのに、ESQL と JAVA を使用するのは、かなりおかしなことです。今日、ストアドプロシージャつまり SQL ストアドプロシージャを作成するのに直接 SQL 構文を使用します。

SQL ストアド・プロシージャは SQL 文のみで論理的構造を埋め込んで実現可能なストアド・プロシージャです。これはサーバーに保存される SQL 文のセットです。一度作成すると、クライアントは独立して SQL 文を保持する必要がなく、代わりに SQL ストアド・プロシージャと連携することができます。ストアド・プロシージャは永久ストアド・プロシージャと一時ストアド・プロシージャを含みます。詳細な情報は*「DBMaster ストアド・プロシージャユーザー参照編」*を参考してください。

アーキテクチャ

SQL ストアド・プロシージャのコアは複合文です。この複合文はキーワード BEGIN から始めて END まで終了します。以下は SQL ストアド・プロシージャ構文の構成：

```
BEGIN                        #block header
Variable declarations
Condition declarations
Cursor declarations
Condition handler declarations
Assignment, flow of control, SQL statements and other compound statements
END;                         #block end
```

SQL ストアド・プロシージャは一つ或いは一つ以上の選択可能な複合文（またはブロック）から構成します。このブロックはシングルの SQL ストアド・プロシージャにネストし連続に引用できます。ブロックは選択可能変数、

条件、処理宣言でオーダーがあります。SQL 制御文、ほかの SQL 文及びカーソル宣言のプロシージャロジックを実現する前、この内容は説明しなければなりません。カーソルは SQL ストアド・プロシージャ体を含む SQL 文セットにどこでも宣言できます。

SQLストアド・プロシージャを作成する構文

ファイルからSQLストアド・プロシージャを作成する

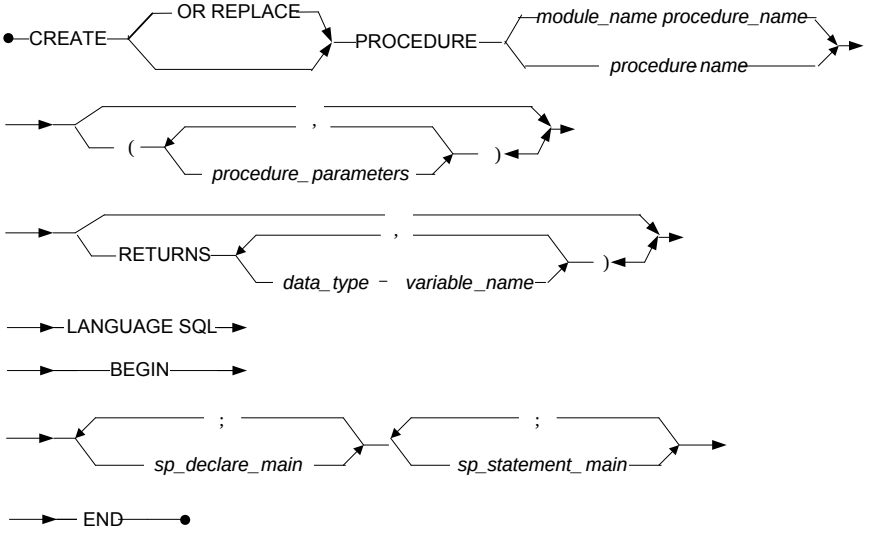


図 12-6 : SQL ストアド・プロシージャを作成する構文

OR REPLACE: OR REPLACE で既存なプロシージャを再び作成します。この句を使用して削除しないで既存なプロシージャの定義を変更でき、このプロシージャは再び作成できます。

注 *プロシージャにCREATE OR REPLACE COMMANDとCREATE OR REPLACE PROCEDUREコマンドをサポートしません。*

注 *AUTOCOMMITはOFFに設定すると、CREATE OR REPLACE PROCEDUREコマンドをサポートしません。*

DBMaster SQL ストアド・プロシージャは外部ファイル(*.sp)をコールして作成されます、dmSQL コマンド・ライン・ツールに直接にこれを書き入れません。

➲ **例 1**

ファイルからストアド・プロシージャを作成：

```
dmSQL> CREATE PROCEDURE FROM 'CRETB.SP';
dmSQL> CREATE PROCEDURE FROM '.\SPDIR\CRETB.SP';
dmSQL> CREATE PROCEDURE FROM 'D:\DATABASE\SPDIR\CRETB.SP';
```

➲ **例 2**

ファイルから SQL ストアド・プロシージャを作成：

```
dmSQL> CREATE OR REPLACE PROCEDURE FROM 'CRETB.SP';
dmSQL> CREATE OR REPLACE PROCEDURE FROM '.\SPDIR\CRETB.SP';
dmSQL> CREATE OR REPLACE PROCEDURE FROM 'D:\DATABASE\SPDIR\CRETB.SP';
```

スクリプトに SQL ストアドプロシージャを作成する

ユーザーはファイルから SQL ストアド・プロシージャを作成できるだけではなく、dmSQL を使用して作成できます。ユーザーは自分で呼び出す、削除して権限を持つ SQL ストアド・プロシージャが実行できます。詳細な情報について *QL ストアド・プロシージャ*の関連章節を参考してください。

SQLSP は一つ以上の SQL 句を含み、この句は'; 'で終了します。だから、dmSQL はブロック区切り記号を支持しなければなりません。ブロック区切り記号は A-Z、 @、%、^になれて、二つ以上のキャラクタ七つ以下のを含みます。ブロック区切り記号に' ; 'は入力終了という意味ではありません。ユーザーは SQLSP を書き入れる前ブロック区切り記号を設定しなければなりません、そうしないと、エラーが起こします。

➲ **例**

スクリプトにストアドプロシージャ構文を作成します。

```
dmSQL> set block delimiter @@;
dmSQL> @@
    2> create procedure sp in script2
    3> language sql
    4> begin
```

```
    5> insert into t1 values(1);
    6> end;
    7> @@
dmSQL> set block delimiter;
```

引数の使用

SQL ストアド・プロシージャ支持の SQL 引数は SQL 値を進出プロシージャに渡すため使用されます。

実行ロジックは特別な入力または入力スカラー・セットに条件を制限する場合、一つまたは多くの出力スカラー値を返すが結果セットを返したくない場合、SQL ストアド・プロシージャの引数は非常に重要です。

変数の宣言

SQL ストアド・プロシージャ支持のローカル変数はユーザーが SQL 値を配置、復旧することが許せます。この SQL 値は SQL ストアド・プロシージャロジックから支持されます。

SQL ストアド・プロシージャの変数は DECLARE 文を通じて定義されます。DECLARE 文はルーチンに以下の項目を定義します：ローカル変数、条件とハンドル、カーソル。DECLARE は BEGIN ... END **複合**文の内部にのみ位置できます、他の文の前になります。宣言は下記のようなオーダーに従わなければなりません：カーソルの宣言はハンドルを宣言する前に行われるべきで、変数と条件の宣言はハンドル或いはカーソルを宣言する前に行われれます。

DBMaster は 2 種類の SQL 変数をサポートします：**接続変数（CV）**と**ステートメント変数（SV）**。この 2 種類の変数が dmSQL コマンドの拡張性及び移行性を強化するために使用されます。下記のセクションでは、CV 及び SV に関する内容と例を説明します。

接続変数（CV）

*CV*はローカル接続時にのみ定義できる接続変数です。ローカル接続における接続変数は特別な接続変数で、その他の接続での接続変数と違います。

すなわち、接続変数はローカル接続に属する接続にのみ使用され、その他の接続により取得または使用されることができません。

ユーザーにとって、接続変数はローカル接続での SQL コマンドのグローバル変数で、dmSQL コマンドラインツール及び SQLSP に用いられます。ユーザーが一つの文に接続変数に値を指定して、それをその他の文に参照されます。これは一つの文から値をその他の文に渡すことを許すと表します。接続が一旦データベースから切断されると、あらゆる接続変数は自動的に解放されます。CV は事前に定義されたデータタイプで、それを使用する前に、変数の設定に従い、そのデータタイプを指定することができます。初期化されていない変数を参照する場合、エラー（6344）[DBMaster]「無効な変数名です」が戻されます。

➲ 例 1

dmSQL コマンドラインツールで CV を宣言及び使用することが出来ます。下記の例で、**@a**、**@aa** 及び**@b** はそれぞれ接続変数で、INSERT、UPDATE、DELETE 操作をするだけでなく、WHERE 句または udf 関数のコール等に用いられます。

```
dmSQL> declare set int @a = 1;
dmSQL> select @b;
ERROR (6344): [DBMaster] 無効な変数名です:B
dmSQL> declare set int @aa = NULL;
dmSQL> select @aa;
          @AA
======================
NULL
dmSQL> declare set int @b = 2;
dmSQL> select @a, @b;
    @A          @B
=========== ===========
1           2

dmSQL> select * from t1 where c1=@a;
dmSQL> insert into t1 values(@b+100);
dmSQL> update t2 set c2 = @b+2 where c1 = @a+1;
dmSQL> delete from t1 where c1=@b;
dmSQL> select sum(c1) into @b from t1;
```

➲ 例 2

SQLSP にて、CV は普通の変数と同じように、宣言及び使用されることができます。下記の例の**@val2** 及び **@val3** は接続変数です。

```
dmSQL> set block delimiter @@;
dmSQL> @@ create procedure tsp2
    2> language sql
    3> begin
    4> declare set int @val2 =100+100;
    5> declare set int @val3 =length('test');
    6> set @val3 =length('test');
    7> end; @@
```

ステートメント変数（SV）

*SV*はステートメント変数で、一つの文で宣言されることができていますが、指定された文にのみ用いられます。異なるコマンドでのステートメント変数はそれぞれ独立な変数で、SQLSP にのみ用いられます。

ステートメント変数の定義は SQLSP のローカル変数の定義とほぼ同じです。SQL 文が一旦終了すると、ステートメント変数は自動的に解放されます。

➲ 例

CV と **SV** との定義上の区別は下記のようになります。

```
dmSQL> set block delimiter @@;
dmSQL> declare set int @aa = 100;
dmSQL> create table tab(c1 int);
dmSQL> @@ create procedure tsp(out c1 int)
    2> language sql
    3> begin
    4> declare vals int default 100;  --vals is sv
    5> insert into tab values(@aa);   --aa is cv
    6> insert into tab values(vals);  --vals is sv
    7> set c1 = @aa;                  --c1 is sv
    8> end; @@
```

カーソル

SQL ストアド・プロシージャのカーソルは結果セット（データ行のセット）を定義して、また基礎行に基づいてシングル行に複雑なロジックを実行します。同じ方法で SQL ストアド・プロシージャは結果セットも定義できて、これを直接**呼出し側**に或いはクライアント・アプリケーションに返します。

カーソルは行セットの中に一行に指すポインターとします。同一タイムだけで一行が参照できますが、必要があると結果セットにほかの行に移動できます。

DECLARE CURSOR 文はカーソルを定義します。カーソルを使用すると以下の SQL 文を使用しなければなりません：DECLARE CURSOR、OPEN、FETCH、CLOSE。

代入文

代入文は SQL 変数または SQL 引数に値を割り当てるため使用されます。変数を宣言する場合、SET 文、CURSOR FOR SELECT FROM 文を通じて或いは初期値をとして変数に値を割り当てます。文字、表現式、クエリ結果、特別レジスターの値みんなを変数に割り当てます。変数の値は SQL ストアド・プロシージャ、SQL ストアド・プロシージャにほかの変数に分配できて、また SQL ストアド・プロシージャ文を実行するルーチンに引数として参照できます。

SET 変数文はローカル変数、出力引数及び新規なトランザクション変数に値を割り当てます。トランザクションはこれを制御します。SET 代入文は簡単な表現式と複雑な表現式を引きうけます。

注 *変数に値を割り当てると、代入長さは1024未満になります。*

簡単表現

簡単な表現は以下のタイプがあります。数字タイプ：INTEGER、BIGINT、SMALLINT、DOUBLE、FLOAT、DECIMAL ; 文字データ・タイプ : CHAR、NCHAR、VARCHAR、NVARCHAR ; BINARY データ・タイプと timestamp タイプ：DATE、TIME、TIMESTAMP。

簡単な表現は'+'、'-'、'*'、'/'、変数、定数、値、ストリングを含みます。簡単表現の実行効率は複雑表現より高いですので、複数循環文に適用して実行スピードを上げます。SQL 関数についての詳細な情報は*「SQL 文と関数参照編」*をご参照ください。

複雑表現

複雑表現は簡単表現に全ての代入値を含むだけではなく、SQL 関数も含みます。例えば組み込み関数とユーザー定義関数。

制御流れの構文

SQL 制御文から提供する変数サポートとフロー制御文は実行順序を制御するため使用されます。IF と CASE のような文は条件付き、SQL 制御文のブロックを実行します。WHILE と REPEAT のようなほかの文はタスクが終了するまで重複文を実行するため使用されます。

SQL 制御文以下の種類を割り当てます：変数関連文、条件文(CASE、IF)、ループ文(FOR、LOOP、WHILE、REPEAT)、転換制御文(ITERATE、LEAVE)、ラベルと SQL ストアドプロシージャ**複合**文。

戻り結果セット

SQL ストアド・プロシージャに、カーソルは結果セットの行循環より使用範囲がもっと広いです。これは結果セットをコール・プログラムに返すことができます。

SQLストアド・プロシージャのステータスを戻る

ステータスコードはストアドプロシージャが成功に実行されるかどうかを表します。ストアドプロシージャにステータスコードを定義することができません。

ステータスコード：

-1 ：ストアドプロシージャの実行が失敗です

0：ストアドプロシージャの実行が成功です

1：ストアドプロシージャの実行中に警告が出る

ストアドプロシージャのステータスを戻ろうとする場合、' LANGUAGE SQL '文の前に' RETURN STATUS 'を追加する必要があります。

➲ **例**

ほかの SQL ストアドプロシージャをコールします：

```
CREATE PROCEDURE call_test
RETURNS STATUS
LANGUAGE SQL
BEGIN
                DECLARE cur CURSOR WITH RETURN FOR select * from call_tb;
                OPEN cur;
END;
CREATE PROCEDURE CASE TEST 1(IN inval INT, OUT outval1 INT, OUT outval2 INT)
LANGUAGE SQL
BEGIN
        SET outval1 = 1;
        SET outval2 = 2;
END;
CREATE PROCEDURE call1(IN inval INT, OUT outval1 INT, OUT outval2 INT)
LANGUAGE SQL
BEGIN
        CALL CASE TEST 1(inval, outval1, outval2);
END;
```

SQLストアド・プロシージャを実行

SQL ストアド・プロシージャは DBMaster コマンドライン（dmSQL ツール）からの文の実行を通じて、または直接にグラフィカル・ユーザー・インタフェース・ツール（JDBA ツール）で CREATE PROCEDURE 文を使用して実行できます。

CALL 文はプロシージャをコールします。この文はアプリケーション・プログラムに埋め込まれて、又は動的な SQL 文を通じて実行されます。これは動的に準備される実行可能な文です。

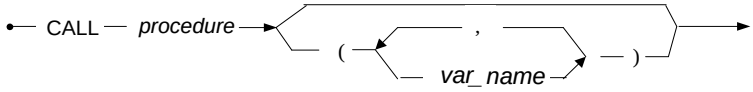


図 12-7 : dmSQL で CALL 文の構文

12.4 ストアド・プロシージャを削除する

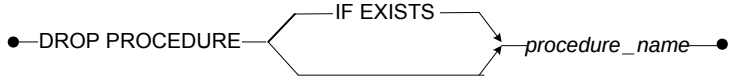


図 12-8 DROP PROCEDURE 文の構文

➲ **例 1**

ストアド・プロシージャ **sp_proc1** とストアド・プロシージャ **user1.sp_proc2** を削除する：

```
dmSQL> DROP PROCEDURE sp_proc1;
dmSQL> DROP PROCEDURE user1.sp_proc2;
```

➲ **例 2**

存在すると、ストアド・プロシージャ **sp_proc1** とストアド・プロシージャ **user1.sp_proc2** を削除する：

```
dmSQL> DROP PROCEDURE IF EXISTS sp_proc1;
dmSQL> DROP PROCEDURE IF EXISTS user1.sp_proc2;
```

12.5 ストアド・プロシージャ情報を取得する

➲ **例 1**

システム表 SYSPROCINFO からストアド・プロシージャ情報を取得する：

```
dmSQL> SELECT * FROM SYSPROCINFO;
```

➲ **例 2**

システム表 SYSPROCPARAM からストアド・プロシージャのパラメータ情報を取得する：

```
dmSQL> SELECT * FROM SYSPROCPARAM;
```

注 *プログラム内でプロシージャとパラメータの情報を取得するには、ODBC 関数 SQLProcedure()とSQLProcedureColumns()を使用します。*

12.6 ストアド・プロシージャのセキュリティ

ストアド・プロシージャを実行することができるのは、DBA 以上のユーザーとプロシージャの所有者のみです。その他のユーザーは、実行権限が与えられているユーザーのみ実行することができます。所有者と DBA 以上のユーザーのみ、他のユーザーにストアドプロシージャの実行権限を与えることができます。

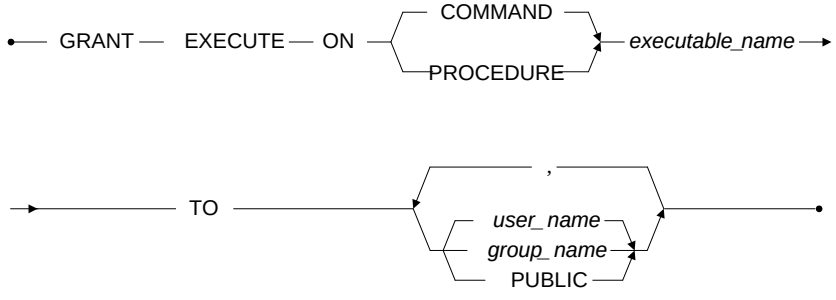


図 12-9 : GRANT EXECUTE 文の構文

所有者と DBA 以上のユーザーは、他のユーザーからストアド・プロシージャの実行権限を取り消すこともできます。

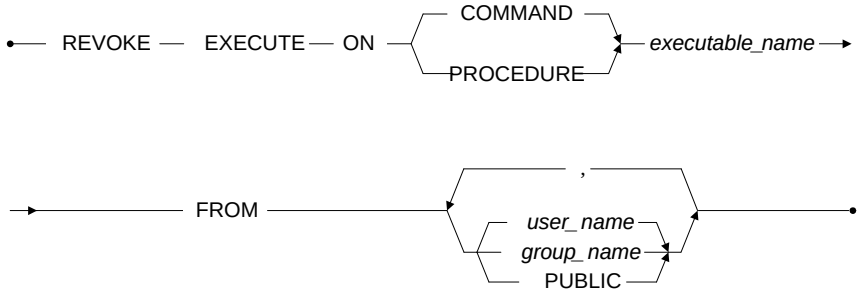


図 12-10 : REVOKE EXECUTE 文の構文

➲ **例 1**

user1 がストアド・プロシージャ sp_proc1 を作成し、***user2*** に実行権限を与える :

```
dmSQL> GRANT EXECUTE ON PROCEDURE sp_proc1 TO user2;
```

➲ **例 2**

user1 がストアド・プロシージャ sp_proc1 を作成し、実行権限を PUBLIC にする :

```
dmSQL> GRANT EXECUTE ON PROCEDURE sp_proc1 TO PUBLIC;
```

➲ **例 3**

user1 がストアド・プロシージャ sp_proc1 への ***user2*** の実行権限を取り消す :

```
dmSQL> REVOKE EXECUTE ON PROCEDURE sp_proc1 FROM user2;
```

➲ **例 4**

user1 がストアド・プロシージャ sp_proc1 への PUBLIC 実行権限を取り消す :

```
dmSQL> REVOKE EXECUTE ON PROCEDURE sp_proc1 FROM PUBLIC;
```

13　スケジュール

オルガニゼーションには複数のタスクがあり、手動的にそれを一つずつ処理するのは非常に難しいです。管理タスクを簡素化にし、複雑なスケジューリングのニーズのため豊富な機能を提供するために、DBMaster では高度なジョブスケジュール機能を追加しています。一つのスケジュールは、タスクがいつ実行されるべきかを示しています。その中に開始時間、終了時間およびタイムテーブルが含まれています。開始時間はスケジュールの開始日時を示し、終了時間はスケジュールの有効期限の終了日時を示し、タイムテーブルは、スケジュールがいつ実行されるかを示します。

スケジュールはユーザーがデータベースの環境で様々な任務の発生日時及び発生場所を制御するのを許可します。これらのタスクがかなり時間が掛かり、そして複雑なのです。そのため、スケジュールによって、これらのタスクの管理と計画を高めることができます。また、手動介入無しでたくさんのルーチンデータベースのタスクの実行を確保することは、ランニングコストが下げられ、更に信頼できるルーチンが実現でき、最大限度に人為的なミスも減らしられ、必要な時間システムも短縮できます。

DBMaster は下記のストアドプロシージャのコレクションを持つ高度なジョブスケジューリング機能を提供します：START_DMSCHSVR、STOP_DMSCHSVR、SCHEDULE_CREATE、SCHEDULE_ALTER、SCHELOG_CLEAN、SCHEDULE_DROP、SCHEDULE_RELOAD、SCHEDULE_ENABLE、SCHEDULE_DISABLE、TASK_CREATE、TASK_ALTER 及び TASK_DROP。これらのプロシージャの詳細につきましては、*「SQL 文と関数参照編」*をご参照ください。RESOURCE 権限を持つユーザーはタスク/スケジュールの作成、変更または削除をすることができ、自分のスケジュール機能を有効/無効にすることもできます；DBA 以上の権限を持つユ

ーザーはタスク/スケジュールの作成、変更または削除をすることができ、あらゆるユーザーのスケジュールを有効/無効にすることもでき、そして、dmschsvr の起動及び停止、全てのスケジュールのリロードをすることもできます。また、ユーザーは自分自身のタスクに基づくスケジュールのみ作成できます。

13.1 dmschsvr

デーモン **dmschsvr** は、主に SQL 文、プロシージャまたは実行可能のプログラムの実行に用いられ、これらはユーザーが事前に設計したものです。ユーザーはルーチンタスクのスケジュールが作成でき、**dmschsvr** は自動的にこれらのタスクを実行して、日常メンテナンスを減らします。

dmschsvr は定期的に SYSSCHEDULE 及び SYSTASK に保存された情報に基づきジョブを実行します。SCHEDULE_CREATE、SCHEDULE_ALTER、SCHEDULE_DROP はそれぞれスケジュールの作成、変更と削除に用いられ、SCHEDULE_ENABLE 及び SCHEDULE_DISABLE はスケジュール機能を無効/有効にすることに用いられます。TASK_CREATE、TASK_ALTER 及び TASK_DROP はそれぞれタスクの作成、変更と削除に用いられます。これらのプロシージャをコールした後、SYSSCHEDULE と SYSTASK に格納されている情報が変更されます。

dmschsvr が実行している間に、まずは ODBC を介して特別なユーザーを使用してデータベースに接続し、その後、SYSSCHEDULE に保存されたあらゆるスケジュール情報をスキャンして使用可のスケジュールを確認し、最後にシステムにあるスケジュール情報を読み取ります。これらの情報によって、**dmschsvr** は現在時刻と実行時間の間の時間間隔がもっとも短いタスクを計算し、その後、当該タスクのデータをキューに読み取り、最後に、実行時間がなると、当該タスクが実行されます。SYSSCHEDULE のデータが変更される場合、**dmschsvr** が自動的にあらゆるスケジュールをリロードすることになります。

コマンド **dmschsvr–ddb_name** で **DB_SCHSV** の値を 1 に設定するまたは *start_dmschsvr('taskNum', 'logPath')*をコールすることによって、**dmschsvr** を起動することができます。START_DMSCHSVR に関する詳細につきましては、*「SQL 文と関数参照編」*をご参照ください。また、コマンド **dmschsvr –d** に

はその他のパラメータがあります：-n は同時に実行中のタスクの最大数を指し、-p はスケジュールログの保存ディレクトリを指します。*stop_dmschsvr* のみをコールすることによって、**dmschsvr** を停止することが出来ます。*stop_dmschsvr* をコールした後、**dmschsvr** は 1 分間後に停止することになります。

13.2 スケジュールの作成

下記のコマンドでスケジュールを作成します。

```
dmSQL> call schedule create('schedule name', 'task name', 'timetable' , 'starttime',
'endtime');
```

➲ **例**

タスク **insert_t1** を作成して、表 **t1** に値を挿入します。

```
dmSQL> call task_create('insert_t1', 'SQL_STATEMENT', 'insert into t1 values(1,2) ');
```

タスク insert_t1 にスケジュール insert_into_t1 を作成します。

```
dmSQL> call schedule_create('insert_into_t1', 'insert_t1', '10 0,1 * * *', '2012-12-12
12:00:00', '2015-12-12 12:00:00'); //タスク 'insert_t1'は 2012-12-12 12:00 から
2015-12-12 12:00 までの毎日の 0:10 及び 1:10 に実行されます。
```

13.3 スケジュールの変更

下記のコマンドでスケジュールを変更します。

```
dmSQL> call schedule alter('schedule name', 'task name',
'timetable' ,'starttime','endtime');
```

➲ **例**

スケジュール **insert_into_t1** を変更します。下記の例では、実行計画**"10 0,1 * * *"**を**"20 2,3 * * *"**に変更します。スケジュール **insert_into_t1** に関する詳しい情報につきましては、セクション 13.2 *スケジュールの作成*の例をご参照ください。

```
dmSQL> call schedule_alter('insert_into_t1', 'insert_t1', '20 2,3 * * *', '2012-12-12
12:00:00', '2015-12-12 12:00:00'); // タスク'insert_t1'は 2012-12-12 12:00 から
2015-12-12 12:00 までの毎日の 2:20 及び 3:20 に実行されます。
```

13.4 スケジュールの削除

下記のコマンドでスケジュールを削除します。

```
dmSQL> call schedule_drop('schedule_name');
```

➲ **例**

スケジュール **insert_into_t1** を削除します。スケジュール **insert_into_t1** に関する詳しい情報につきましては、セクション 13.2 *スケジュールの作成*の例をご参照ください。

```
dmSQL> call schedule_drop('insert_into_t1');
```

13.5 スケジュールのリロード

下記のコマンドであらゆる有効のスケジュールをリロードします。

```
dmSQL> call schedule_reload;
```

➲ **例**

使用可のあらゆるスケジュールをシステムにリロードします。

```
dmSQL> call schedule_reload;
```

14 ユーザー定義関数のコーディング

DBMaster では、プログラマが各ユーザー定義関数(UDF)を構築することができます。一旦 DBMaster に UDF を書き込むと、新しい DBMaster の組み込み関数のように扱われます。新しいユーザー定義関数の作成はわかりやすく、以下に示した一般手順のとおりです。

➲ **ユーザー定義関数を作成する：**

1. C言語（UDFインターフェース）でユーザー定義関数を記述します。
   a) インクルード文を記述します。
   b) 関数ヘッダーを記述します。
   c) 関数が与える引数を記述します。
   d) 必要に応じ割り当てメモリを定義します。
   e) 必要であれば、エラー・コードを定義します。
2. UDFのダイナミック・リンク・ライブラリを構築します。
3. UDFに与えるデータ配列と共に、DBMasterでUDFを作成します。

14.1 UDF インターフェース

UDF 作成の最初のステップは、C 言語でコーディングすることです。以下の節では、C 言語での UDF の例を紹介し、DBMaster の UDF 特有の各コード要素について説明します。

サンプル

INTEGER データ型を文字列に変換する UDF、**INT2STR()**を作成するには、それを含むダイナミック・リンク・ライブラリを構築する必要があります。

```
dmSQL> SELECT INT2STR(c1) FROM t1;      // c1 は integer データ型
```

以下の C ソースコード *template.c* は、ユーザー定義関数 **INT2STR()**のスナップショットです。

```
#include <memory.h>
#include <string.h>
#include <stdio.h>
#include "libudf.h"

/* integer データ型を文字列データ型に転送する */
#ifdef WIN32
__declspec(dllexport)
#endif
int INT2STR(int narg, VAL args[])
{
  char *ptag;
  int len;
  char p1[11];
  int rc;

  if (args[0].type != NULL_TYP)
     {
     sprintf(p1, "%d", args[0].u.ival);
     len = strlen(p1);
     if (rc = _UDFAllocMem(args, &ptag, len))
         return rc;
     memcpy(ptag, p1, len);
     args[0].type = CHAR_TYP;
     args[0].len = len;
     args[0].u.xval = ptag;
     }
  return _RetVal(args, args[0]);
}
```

LIBUDF.Hを含む

DBMaster では、UDF のコーディングに必要ないくつかの定数、データ型、一般的なインターフェースを定義します。

UDF のコーディングの前に、必ず libudf.h を含む必要があります：

```
#include "libudf.h"
```

パラメータを経由する

SQL 文で使用する UDF の引数は、C 言語では UDF コードのパラメータ *args* にパッケージされます。*args* 配列を通じて、UDF は入力データを取得します。args は UDF 制御ブロックとも呼ばれ、常に DBMaster の一般的なインターフェースの最初の引数として使用されます。BLOB 一般インターフェースのような一般インターフェースについては、後ほど説明します。

C 関数の各 UDF 見出し：

```
int FUNCTION_NAME(int narg, VAL args[])
{
...
}
```

注 *args[]は、配列を意味します。1引数のみパスする関数には、*argsのポインタ形式を使用します。*

nargは、関数が渡す引数の数を定義します。UDF **MYSUBSTRING (c1, c2, c3)** がSQL文にコールされる場合、**c1**情報はargs[0]で、**c2**はargs[1]で、**c3**はargs[2]によって渡されます。nargの値は、配列のサイズが3であることを示します。

➲ **例 1**

c1 に'abcdefghijklmn'の値を使うと、*args[0]*は以下のようになります：

```
args[0].type   = CHAR TYP
args[0].len    = 14
args[0].u.xval = "abcdefghijklmn"
```

➲ 例 2

c2 に整数 30 の値を使うと、*args[1]*は以下のようになります：

```
args[1].type   = INT TYP
args[1].len    = 4
args[1].u.ival = 30
```

CHAR_TYP と INT_TYP に加えて、BIN_TYP、FLT_TYP、OID_TYP、BLOB_TYP、DEC_TYP、NULL_TYP が ***libudf.h*** に定義されている定数です。

```
#define BIN_TYP   0x0000                /* bit string     data type*/
#define CHAR_TYP  0x1000                /* character      data type*/
#define INT_TYP   0x2000                /* integer        data type*/
#define FLT_TYP   0x3000                /* floating point data type*/
#define OID_TYP   0x4000                /* OID            data type*/
#define BLOB_TYP  0x5000                /* BLOB           data type*/
#define DEC_TYP   0x6000                /* decimal        data type*/
#define NULL_TYP  0xF000                /* set if column is null   */
```

➲ 例 3

NULL_TYP で、入力データが NULL かどうかを確認することができます。

```
if (args[0].type == NULL_TYP)
{
    /* 入力データはNULL */
}
else
{
   /* 入力データは非NULL */
}
```

libudf.h に定義されている VAL のデータ構造体：

```
typedef struct tVAL {
   i16 type;                    /* data type          */
   i15 len;                     /* data length        */
   union {
      i31    ival;             /* long integer data  */
      i15    sival;            /* short integer data */
      double fval;             /* double data        */
      float  sfval;            /* float data         */
```

```
      dec t  dval;            /* decimal data       */
      char   *xval;           /* pointer to data    */
   } u;
} VAL;
```

libudf.h にある DECIMAL データ型に使用する構造体 *dec_t* :

```
typedef struct
{
    i8 pre;
    i8 sca;
    i8 dgt[20];
    i8 exp;
   i8 junk;
} dec_t10;
typedef dec_t10 dec_t;
```

UDF は、VAL 型を通して入力データをパスするだけでなく、それを通じて出力データを戻します。データを戻す方法については、後ほど説明します。

割り当てスペース

C 関数で、メモリを割り当て、関数を終わる前にそれを解放する必要があるかもしれません。文字列や一時 BLOB ID のような戻された値は、メモリを割り当て、UDF 関数に残し、空きメモリ・スペースで DBMaster に役立てる必要があります。

➲ 例

arg は UDF 制御ブロック、*ppt* は割り当てたメモリ・ブロックを取得するポインタで、*nb* は希望する割り当てサイズです。この関数はメモリを割り当て、DBMaster が取り扱うまで維持します。

```
int _UDFAllocMem(VAL *arg, char **ppt, int nb);
```

_UDFAllocMem()に割り当てられたメモリ・スペースへのポインタ、*args[0].u.xval* を通じて、DBMaster は結果が戻された後にそれが解放されたことを知る :

```
if (rc = _UDFAllocMem(args, &ptag, 10))
   return rc; /* return error code */
memcpy(ptag, "0123456789", 10);
args[0].type = CHAR_TYP;
```

```
args[0].len = len;
args[0].u.xval = ptag;
```

結果を戻す

2 種類の戻り値があります。一つは、エラーコードで、もう一方は、配列タイプ VAL を経由した UDF の結果です。エラーコードは DBMaster に返りますが、その値はユーザーにはわかりません。エラーメッセージのみ表示されます。以下でどのようにエラーコードが返されるのかを解説します。

C 関数の UDF の見出し：

```
int FUNCTION_NAME(int narg, VAL *args);
```

FUNCTION_NAME()が 0 以外の値を戻した場合、何らかの問題があります。戻り値が 0 の場合は、関数には問題ありません。

UDF から戻す前に、以下の宣言で UDF から DBMaster にインポートされた結果を渡すために_RetVal()をコールします：

```
int _RetVal(VAL *arg, VAL rtn);
```

最初の引数 *arg* は UDF 制御ブロック、2 番目の *rtn* は戻り値です。以下のコードは整数 30 を戻します。

```
int rc;                      /* error code */
VAL rtn;
rtn.type   = INT_TYP;
rtn.len    = 4;
rtn.u.ival = 30;
rc = _RetVal(arg, rtn);      /* pass result back to DBMaster         */
return rc;                   /* return error code (0 means no error) */
```

14.2 UDF ダイナミック・リンク・ライブラリの構築

DBMaster には、ダイナミック・リンク・ライブラリを構築するために、ソース・ファイルと一緒にリンクするライブラリ *dmudf.lib* があります。Microsoft Windows と UNIX 環境のダイナミック・リンク・ライブラリは、異なるので、別々に説明します。

MICROSOFT WINDOWS環境でのDLL

DBMaster はソースコード template.c 及び WIN32 ビットのユーザー用のコンパイラ環境テンプレートメークファイル udf42.mak(Microsoft VC++ 4.2 バージョン用)、udf50.mak (Microsoft VC++ 5.0 バージョン用)、udf60.mak (Microsoft VC++ 6.0 バージョン用)及び WIN32 ビットユーザーと WIN64 ビットのユーザー用のコンパイラ環境テンプレートメークファイル udf80.mak (Microsoft Visual studio 2005 及びそれ以上のバージョン用)をサポートし、ユーザーがこれを参照することができます。これらのソースコード及びテンプレートメークファイルがディレクトリ/udf_templates の下にあります。ユーザーはテンプレートの C ソースファイルによって UDF を定義することができます。

➲ 以下の文では、udf60.mak を使用します。

1. dmudf.libファイルの位置を確認し、必要な変更を加えるためにVisual C++に用意されたIDEを使用します。
2. テンプレートファイルudf60.makを指定のディレクトリにコピーし、改めて名前を付けます。
3. <ファイル>メニュから<ワークスペースを開く>オプションを選択し、テンプレートメークファイルのプロジェクトワークスペースを開きます。
4. <プロジェクトワークスペース>メニュから<ファイルビュー>オプションを選択し、template.cをクリックして、削除ボタンを押してそれを削除します。
5. ツールバーで<プロジェクト>オプションを選択して、<プロジェクトに追加>、<ファイル>を選択し、自分の.cファイルをプロジェクトワークスペースのテンプレートメークファイルに挿入します。
6. <プロジェクト> -> <設定>で、WIN32 Debugを選択します。プロジェクト設定<一般>で、出力ディレクトリが変更できます。テンプレートメークファイルにて、60Debを中間及び出力ファイルのディレクトリとして設定します。

7. プロジェクト設定<リンク>アイテムでは、カテゴリ・アイテム<一般>で、出力.dllファイル名を直接<出力ファイル名>で変更することができます。又、DBMasterが各人の作業ディレクトリに<オブジェクト/ライブラリ・モジュール> をサポートしている*dmudf.lib* ファイルのリンク・パスを変更する必要があります。

上記が完了した後、各人の **dll make** ファイルを作成することができます。似たような手順で、WIN32 Release version の dll ファイルを作成することができます。

VC++のパワー・ユーザーは、同様の手順と 4 バイトの構造 member alignment を設定することで、dll 作成ファイルを作成することもできます。VC ++ 6.0 IDE **project workspace** では、C/C++ メニュー・アイテムを選択して、**Category** ダイアログ・ボックスでは、<**Code Generation**>を選択して下さい。そこにある構造 member alignment オプションを、結果として 4 バイトを選択します。

collect dll を書く際に、設定を残すために **make file** テンプレートを使用して下さい。**make file** で **template.c** を初期設定の C ファイル名として使用したくない場合は、***udf60.mak*** から ***template.c*** を削除し、***udf60.mak*** プロジェクト作業スペースに独自の C ファイルを挿入して下さい。

➲ **例**

DBMaster **template.c** に、用意された **libudf.h** ファイルを必ず含め、各人の関数をエクスポートして下さい。VC++ プログラマ・ガイドブック、又は以下のエクスポート関数手法を使用して下さい。

```
__declspec(dllexport) datatype YOUR_FUNCTION_NAME( ..... )
```

替わりに、各人の関数をエクスポートするためプロジェクト作業スペースに **def file** を作成し、UDF のための関数名が C ソースコードの大文字であることに留意して下さい。

上記を完了した後、**debug/release version dll** ファイルをビルドすることができます。例えば、***udf60.dll*** ファイルを作成します。

➲ **以下の文では、udf60.mak を使用します：**

1. **udf80.mak 、udf80.def、template.c**及び**udf60.dsp**を必要なディレクトリにコピーして、名前をudf60.dspからudf80.dspに変更します。

2. **udf80.def**の編集により、自分の. cファイルで**template.c**を置換することができます。
3. **udf80.def**の編集により、出力ディレクトリを変更することができます。テンプレートメークファイルにて、60Debを中間及び出力ファイルのディレクトリとして設定します。
4. **udf80.def**の編集により、output .dllファイルの名称を**udf80.mak**に変更できます。

上記のステップを完了した後、cmd によって自分の **dll make** ファイルを作成することができます。

cmd を開いて、cd コマンドで必要なディレクトリに切り替えて、下記のコマンドを実行します。

```
@call bat vs env
@NMAKE /NOLOGO /C /S /f udf80.mak /x make.err CFG="udf80 - Win32 Debug"> make.msg
```

bat_vs_env：この値は Visual Studio のバージョンとオペレーティングシステムによって決定されます。例えば、C コンバイル環境及びオペレーティングシステムはそれぞれ VS2005 と 32 ビットの場合、"*C:|Program Files|Microsoft Visual Studio 8|VC|bin|vcvars32.bat*"で bat_vs_env を置換することができます。

これに関する詳細については、下記の表をご参照ください。

VS バージョン	OS	bat_vs_env
VS2005	32bit	C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 8\VC\bin\amd64\vcvarsamd64.bat
VS2008	32bit	C:\Program Files\Microsoft Visual Studio 9.0\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\bin\amd64\vcvarsamd64.bat
VS2010	32bit	C:\Program Files\Microsoft Visual Studio

		10.0\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\amd64\vcvars64.bat
VS2012	32bit	C:\Program Files\Microsoft Visual Studio 11.0\VC\bin\vcvars32.bat
	64bit	C:\Program Files (x86)\Microsoft Visual Studio 11.0\VC\bin\amd64\vcvars64.bat

dll の Release バージョンをビルドするために、Release で cmd コマンドライン "@NMAKE /NOLOGO /C /S /f udf80.mak /x make.err CFG="udf80 - Win32 Debug "> make.msg"の Debug を置換することができます。

UNIXのUDF soファイル

UNIX のダイナミック・ライブラリ、**so** ファイルを作成することができます。

➲ **例**

udf. c ファイルと名づけた例で UDF C ソースコードを記述する。完成後、UNIX ベースの OS で UDF 関数を使用する。

```
$ cc -c udf.c
$ ld -o libudf.so udf.o -lm
 dmSQL> CREATE FUNCTION libudf.INT2STR(INT) RETURNS CHAR(10);
```

注 *上記の例にあるldコマンドのオプションは、UNIXで変ることがあります。–Gであったり、–sharedであったり、それ以外のものであったりします。共有ライブラリ作成でldコマンド使用方法をチェックするためにUNIXマニュアル、又はman pagesを参照して下さい。*

14.3 UDF の作成/使用/削除

次は DBMaster 内のユーザー定義関数を説明します。下記の部分ではユーザー定義関数の作成、クエリ及び削除に関する構文を述べます。

UDFの作成

➲ 構文

```
dmSQL> CREATE FUNCTION <udf_dll_name.function_name> (<function_datatype>) RETURNS
<function_output_datatype>;
```

UDFの問合せ

➲ 構文

```
dmSQL> SELECT 関数名(カラム名) FROM 表名;
```

UDFの削除

➲ 構文

```
dmSQL> DROP FUNCTION 関数名;
```

サンプル

UDF ファイルの使用方法は以下のようになります。

➲ 例 1

表スキーマが number INT, name CHAR (10)の表 tb_UDF があるデータベース DMDEMO を使用する：

```
dmSQL> SELECT * FROM tb_UDF;
number    name
=======  =========
10         1
20         2
30         3

3 rows selected
```

DBMaster での例 *template.c* を使って、ここで *udf60.dll* を問題無く作成することができます。

dmconfig.ini ファイルの DMDEMO セクションに DB_LbDir キーワードを 1 行追加する：

```
[DMDEMO]
DB_DBDir = D:\UDFDEMO
DB_FODIR = D:\UDFDEMO\FO
DB_LBDIR = D:\UDF\60Deb   ; 追加した行
```

DB_LbDir の詳細については、*dmconfig.ini* のキーワードを参照して下さい。
DB_LbDir を設定するか、UDF の初期設定ディレクトリの**<DBMaster ディレクトリ>¥shared¥udf** に ***udf60.dll*** *を置いて下さい。*

➲ **例 2**

データベース **DMDEMO** を起動し、UDF 関数を作成します。例では、*udf_dll_名*は **udf60**、*関数名*は **INT2STR**、*入力データ型*は INT、*出力データ型*は CHAR(10)です。

```
dmSQL> CREATE FUNCTION udf60.INT2STR(INT) RETURNS CHAR(10);
```

UDF 関数 **INT2STR** の結果は、*関数名*が **INT2STR**、**tb_UDF** のスキーマに従い*カラム名*が **number**、*表名*が **tb_UDF** になります。

```
dmSQL> SELECT INT2STR(number) FROM tb UDF;

INT2STR(c1)
===========
10
20
30

3 rows selected
```

➲ **例 3**

同じダイナミック・リンク・ファイルの UDF 関数 **STR2INT** の例：

```
dmSQL> CREATE FUNCTION udf60.STR2INT(CHAR(10)) RETURNS INT;
dmSQL> SELECT STR2INT(c2) FROM tb_UDF;

 STR2INT(c2)
============
1
2
```

```
3

3 rows selected
```

➲ 例 4

UDF 関数を削除する場合、UDF 関数名を削除するだけで、dll 名を添える必要はありません。UDF 関数を削除すると、データベースが終了するのを待って消去されます。データベースが終了するまで、関数は残っています。

```
dmSQL> DROP FUNCTION INT2STR;
```

14.4 XML が有効な UDF の作成

FLEXML

Kristoffer Rose の Flexml は GNU 一般公有使用許諾に基づき配布されている、XML プロセスジェネレータです。DTD ファイルを使い LEX ファイルを生成します。Flexml はhttp://flexml.sourceforge.net/から取得できます。

LEXファイルの生成

```
$ flexml name.dtd
```

カスタマイズ YY_INPUTの追加

原型の LEX 入力は FILE 入力となります。LEX ファイルの使用には書き換えが必要です。UDF BLOB は入力ソースとなります。下記の例はカスタマイズされた YY_INPUT 追加による書き換えのデモです。

➲ 例

以下のように YY_INPUT 追加により LEX ファイルの定義部分を書き換えます。定義部分はファイルの冒頭部分から"%{ “ と “}%" までです。

```
#include "libudf.h"

typedef struct udf_file
        {
        VAL *args;
        i31 handle;
        i31 rc;
```

```
  i31 left;
        i31 rlen;
        } UDF FILE;

#undef YY_INPUT
#define YY_INPUT(buf,result,max_size)  {\
  UDF_FILE * uf =(UDF_FILE *)yyin; \
        errno=0; \
        if ( uf->left <= 0 )\
        {\
          result = (uf->rlen=0);\
        }\
        if ( (uf->rc = _UDFBbRead(uf->args, uf->handle, max_size, &(uf->rlen),
buf))!=0 ) \
  { \
          errno = uf->rc; \
          result = 0;\
        }\
        else\
        {\
    uf->left -= uf->rlen;\
          result = uf->rlen;\
        }\
}\
```

LEX ファイルの最後に UDF 関数を追加：

```
#ifdef WIN32
  declspec(dllexport)
#endif
int XXX VALIDATE(int nArg, VAL args[])
{
  BBObj bbin;
  UDF FILE uf;
  int rc = 0;
  int rc2 = 0;
  memset(&uf, 0, sizeof(UDF FILE));
  memcpy((char *)(&bbin), args[0].u.xval, BBOBJ SIZE);
  uf.args = args;
  rc = _UDFBbOpen( args, bbin, &(uf.handle));
  if( rc != 0 )
```

```
    goto EXIT;
  if (rc =  UDFBbSize(args, bbin, &(uf.left)) )
        {
    goto EXIT;
        }
  yyin = (void *)&uf;
  rc = validdtd00udf();
  rc2 = _UDFBbClose(args, uf.handle);
EXIT:
  if( args[0].type != NULL_TYP ) // null column data
        {
          args[0].type = INT_TYP;
          args[0].len = 4;
          args[0].u.ival = (rc == 0? 1:0);
        }
  return _RetVal(args, args[0]);
}
```

DLL/SOの作成

flex *名*.l

cc –c –DBUILD_DLL lex.name.c –IDBMaster-installed-dir/include

UDFの作成

CREATE FUNCTION dllname.udfname(BLOB) returns int;

チェック制約を定義したカラムの作成

CREATE TABLE table-name(c1 XMLTYPE CHECK udfname(value) = 1);

DBMASTER DTDを確認するUDFの発生器

特定の DTD を使用可能な UDF をコマンドラインツールで生成します。DTD のファイル名は必須です。定義されていなければエラーとなります。出力ディレクトリはオプションです。定義されていないときはファイルはカレントディレクトリに生成されます。プレフィックスもオプションで、定義するとプレフィックスをファイル名の中に使用しファイルが生成されます。定義しなければ DTD ファイル名には拡張子が付きません。

```
$ dmxmludfmk -dtd dtd-file-name [-o output-directory] [-p prefix]
```

数ファイルが生成されます。

- Lexファイル：＜ユーザ指定プレフィックス.l＞
- Yacc ファイル ：＜ユーザ指定プレフィックス.y＞
- UDF関数ファイル：＜ユーザ指定プレフィックス.c＞udf.c と[ユーザ指定プレフィックス]udf.h.
- UDF関数ファイルは<ユーザ指定プレフィックス>_VALIDATEとして名付けられます。
- hash.c と hash.h はhash 関数を提供します。
- UNIX でのMakefile或いはWindowsプラットフォームで のMakefile.msvc.

➲ **例 1**

```
Make <user-specified-prefix>.so        ;for UNIX
```

➲ **例 2**

```
Nmake /f Makefile.msvc                 ;for Windows visual studio 2005 or 2008
```

➲ **例 3**

```
nmake /f Makefile.msvc COMPILER=VC60   ;for Windows visual studio 6.0
```

➲ **例 4**

```
nmake /f Makefile.msvc OSTYPE=$OSTYPE  ; for cygwin environment
```

注 *dmxmludfmkはASCII,BIG5,gb,Shift-JIS,UTF8をサポートしており、そのほかのflexmlはDTD内部定義のコンテンツ置き換えのみサポートしています。*

初期設定確認

I_VALIDATE は XML 文をチェックするための初期設定確認方法を提供します。I_VALIDATE は DTD と XML スキーマに対してはチェックを提供していません。I_VALIDATE は libmedhia ライブラリの一部です。

14.5 UDF BLOB 一般インターフェース

今日、ユーザーにとってマルチメディアは、重要で有益なものです。DBMaster は、ファイル操作手法を使って BLOB にアクセスする一般インターフェースをサポートしています。そのため、プラグラマーは簡単に BLOB タイプのデータ用の UDF を記述できることができます。データベースに保存することができる BLOB タイプのデータ型は、FILE、LONG VARCHAR、LONG VARBINARY です。

DBMaster の新しい機能の多くは、一時結果を処理するために一時 BLOB を必要とします。DBMaster では、プログラマーが UDF をより簡単に記述できるようにするため一時 BLOB も使用できます。プログラマーは標準 BLOB を開き、データを読み込み、変換関数その他を実行し、新規一時 BLOB に結果を保存し、UDF にそれを戻します。API は、通常の BLOB カラム同様この一時 BLOB をフェッチします。

BLOB一般インターフェース関数

DBMaster には、プログラマーが UDF を記述するための BLOB 一般インターフェース関数があります。DBA は、データベース起動前に、一時 BLOB ファイル用に **dmconfig.ini** ファイルの **DB_FoDir** をセットする必要があります。一時 BLOB は、**DB_FoDir** ディレクトリにある外部ファイルに作成されます。ファイル名のフォーマットは、"**__??????.TMP**"です。"? "は、[0-9, A-Z]いずれかの文字を意味します。このフォーマットの全ファイルは、データベースを終了し再起動する際に、削除されます。

_UDFBbOpen()

bbObj を使って BLOB を開き、**pHandle** を通じて操作を戻します。**BbObj** は、UDF の入力配列で BLOB を使って、**Arg[i]**で回収することができます。**BOLB** が問題無く開いた場合、この関数は 0 を戻し、それ以外ではエラー・コードを戻します。

```
int _UDFBbOpen(VAL *Arg, BBObj bbObj, i31 *pHandle);
```

_UDFBbRead()

この関数は操作に指定する BLOB を読み込みます。この関数をコールする前に、読み込みデータを取得するために関数**_UDFAllocMem()**を使って、**szBuf** とバッファ(**pBuf**)を割り当てます。戻されたデータは、**pBuf** に保存され、サイズ実際の読み込みは **szRead** にあります。**szBuf** は、non-positive の場合、文字は読み込まれません。:

```
int _UDFBbRead(VAL *Arg, i31 handle, i31 szBuf, i31 *szRead, char *pBuf);
```

_UDFBbSeek()

関数、**BLOB** の次の出力演算の位置をセットするために使用します。新規位置は、*libudf.h* で定義される **SEEK_BB_BEG**、**SEEK_BB_CUR**、**SEEK_BB_END** の値を使って、**ptrname** に従って、開始部分、現在の位置、又はファイルの終わりからオフセット・バイトの位置にあります。この関数は、**_UDFBbOpen()**と**_UDFBbClose()**間でのみ実行され、**_UDFBbCreate()**と**_UDFBbClose()**間では作動しません。:

```
int _UDFBbSeek(VAL *Arg, i31 handle, i63 offset, i16 ptrname);
```

_UDFBbCurOffset

この関数は、オープン **BLOB** 又は **pOffset** で **BLOB** にあるオフセットの現在の位置を戻します。多くとも 2G-1、いっそう offset＞＝2G の場合、この関数を戻します。

```
int _UDFBbCurOffset(VAL *Arg, i31 handle, i31 *pOffset);
```

_UDFBbCurOffsetEx

Unlike _UDFBbCurOffset と違うのは, この関数は、オープン **BLOB** 又は **pOffset** で **BLOB** にあるオフセットの現在の位置を戻します。

```
int _UDFBbCurOffsetEx(VAL *Arg, i31 handle, i63 *pOffset);
```

_UDFBbClose()

_UDFBbOpen() で開いた、又は**_UDFBbCreate()**で作成した BLOB を閉じます。

```
int _UDFBbClose(VAL *Arg, i31 handle);
```

_UDFBbCreate()

一時 BLOB を作成し、_UDFBbWrite()のハンドルを戻します。コーラーは、pBbObj で指定され、_UDFBbCreate()、_UDFBbWrite()、_UDFBbClose()で書かれた BBObj 構造のためのスペースを用意する必要があります。BBObj は、BBObj 引数を使って_UDFBbDrop()と呼ばれる一時 BLOB を削除する場合に、この一時 BLOB を識別するために使用します。

成功した場合、pHandle は_UDFBbWrite() で書かれ、_UDFBbClose()で閉じたオープン・ファイルのハンドルに似た BLOB ハンドルを戻します。

替わりに、ファイル(**BB_TEMP_FO**)やメモリ(**BB_TEMP_MEM**).で作成される一時 BLOB を指定します。これは、メモリの制限のために一時 BLOB がメモリで作成されことには意味するものではありません。メモリの元の一時 BLOB や入力データがサイズの制限を越える場合、メモリの一時 BLOB は、システムでファイルに変換される可能性があります。プログラマーは、コーディングの際にこの機能に依存することはできません。

成功し、エラー・コードが戻されない場合、関数は 0 を戻します。

新規一時 BLOB を読み込む前、_UDFBbClose()を使ってそれを閉じてから、_UDFBbOpen()を使って再び開きます。読み込みのために一度閉じて再び開かない限り、一時 BLOB で_UDFBbSeek() を使用することはできません。

```
int _UDFBbCreate(VAL *Arg, BBObj *pBbObj, i31 *pHandle, i31 Opt);
```

_UDFBbWrite()

一時 BLOB 作成のために_UDFBbCreate()を使った後、_UDFBbWrite()を使って、それにデータを書き込みます。ハンドルは、_UDFBbCreate()から、pBuf はデータの入力を指示し、長さは szBuf です。成功すると、関数は 0 を戻し、それ以外はエラー・コードが戻されます。

```
int _UDFBbWrite(VAL *Arg, i31 handle, i31 szBuf, char *pBuf);
```

_UDFBbDrop()

通常、一時 BLOB が UDF から戻され、システムがその寿命を制御する場合、それを削除する必要がありません。但し、作成した BLOB を戻さない場合、一時 BLOB を削除するためにこの関数を使用します。この関数は、標準 BLOB では実行できません。実行した場合、ERR_BLOB_INV_BLOB エラー

を受け取ります。成功した場合、関数は **0** を戻し、それ以外はエラー・コードが戻されます。

```
int _UDFBbDrop(VAL *Arg, BBObj bbObj);
```

_UDFBBSIZE()

この関数は、**pLen** で **BLOB** のデータ・サイズを戻します。**BbObj** は、標準 BLOB か一時 BLOB です。多くとも 2G-1、いっそう offset＞＝2G の場合、この関数を戻します。成功した場合、関数は **0** を戻し、それ以外はエラー・コードが戻されます。

```
int _UDFBbSize(VAL *Arg, BBObj bbObj, i31 *pLen);
```

_UDFBBSIZEEX()

_UDFBbSize と違います,この関数は pLen で **BLOB の実際データサイズを戻します。**. **BbObj** は標準な BLOB でも臨時的な BLOB でもいいです。成功すると関数が 0 を戻し、そうしないと、エラーコードが戻されます。

```
int _UDFBbSizeEx(VAL *Arg, BBObj bbObj, i63 *pLen);
```

サンプル

以下の例では、ユーザー定義関数の作成方法を示しています。ユーザー定義関数 **MYCONVERT** は、VARCHAR 形式で入力し、一時 BLOB として出力します。

➲ **ユーザー定義関数 MYCONVERT を作成する：**

1. **myudf.c**(ソースコードは後述に従います)を使って、UNIXにダイナミック・ライブラリを作成します。

```
cc -g -c myudf.c
ld -G -o myudf.so myudf.o
```

2. データベースを起動します。

3. 次のコマンドを入力します。

```
dmSQL> CREATE FUNCTION myudf.myconvert(VARCHAR(100))  // input string
    2>  RETURNS LONG VARCHAR;                         // output BLOB
dmSQL> SELECT myconvert(c1) FROM mytable;             // output temp BLOB
```

UDF MYCONVERT のソースコードは以下のようになります。

```
#include "libudf.h"
int MYCONVERT(int nArg, VAL args[])
{
  int     rc = 0, trc;             /* return code                      */
  BBObj   tmpobj;          /* output temp BLOB                */
  i31     handle;          /* handle of created temp BLOB     */
  boolean fgCreate = FALSE;        /* temp BLOB has been created?     */
  char   *pInData, pOutData[4096];/* input/output data buffer        */
  i31     nInData, nOutData;       /* input/output data buffer length */

  if (args[0].type == NULL_TYP)
    goto cleanup;

  pInData = args[0].u.xval;        /* get input data                   */
  nInData = args[0].len;           /* input data length                */

  /* create a temp BLOB in file */
  if (rc = _UDFBbCreate(args, &tmpobj, &handle, BB_TEMP_FO))
    goto cleanup;
  fgCreate = TRUE;

  /* any real processing function */
  RealConvert(pInData, nInData, pOutData, &nOutData);

  /* write result data to temp BLOB */
  if (rc = _UDFBbWrite(args, handle, nOutData, pOutData))
    goto cleanup;

  /* close created temp BLOB (NOTE: temp BLOB is still alive) */
  if (rc = _UDFBbClose(args, handle))
    goto cleanup;

  args[0].type = BLOB_TYP;
  args[0].len = BBID_SIZE;
  args[0].u.xval = (char *)&tmpobj;
                 /* _RetVal() does a copy from this local buffer  */

cleanup:
  if (rc)
    {
```

```
    /* error handle */
    if (fgCreate)
      {
      _UDFBbClose(args, handle);      /* close created temp BLOB     */
      trc = _UDFBbDrop(args, tmpobj); /* drop it because of failure */
      if (trc > rc)
        rc = trc;
      }
    return rc;
    }
  else
    return _RetVal(args, args[0]);

}/* MYCONVERT() */
```

トラブルシューティング

BLOB 一般インターフェースを使って BLOB UDF を記述する際、トラブルシュートのために以下を使用します。

ERROR (327): BLOBカラムは、まだ開かれてないか、生成されていません

関数は、他の BLOB 関数インターフェースを使用する前に、BLOB をオープンする場合は**_UDFBbOpen()**、新規一時 BLOB を作成する場合は、**_UDFBbCreate()**を使用する必要があります。

ERROR (328): BLOBカラムのオフセット値が無効です

UDF が**_UDFBbSeek()**を使って、長さが 5 しか無い BOLB を長さ 10 でオフセットするケース。

ERROR (331): このBLOBは生成されている状態ではありません

_UDFBbWrite()は、**_UDFBbCreate()**で作成した一時 BLOB 上でのみ実行できます。それを閉じることはできません。例えば、**_UDFBbOpen()**で開いた BLOB でこれを使用すると、エラーになります。

ERROR (330): このBLOBはオープンされている状態ではありません

_UDFBbRead()は、**_UDFBbOpen()**で開いた BLOB(一時 BLOB を含む)でのみ実行できます。

ERROR (332): BLOBオブジェクトはクローズされていません

BLOB を開くために**_UDFBbOpen()**か**_UDFBbCreate()**が使用されている場合、プログラマは開いた BLOB を閉じるために**_UDFBbClose()**を呼ぶ必要があります。

ERROR (322): CONFIGファイルにファイル・オブジェクトのディレクトリがありません;ファイル・オブジェクトの挿入はできません

一時 BLOB が使用されている場合、**dmconfig.ini** *ファイル*のキーワード **DB_FoDir** を必ずセットします。セットしない場合、一時 BLOB 作成しようとしても失敗し、エラーになります。

14.6 UDF に関連する dmconfig.ini のキーワード

DB_STRSZ

dmconfig.ini ファイルのキーワードで UDF に関連するものには、**DB_LbDir** と **DB_FoDir** に加え、DB_StrSz があります。

```
DB_STRSZ=<値>
```

このキーワードは、ユーザー定義関数(UDF)の使用時に、STRING タイプのデータの戻り長さを定義します。UDF は固定長さのデータのみを戻ることができるため、長すぎの文字列の受けを避けるには、このキーワードを使用して STRING タイプのデータの長さを制限します。当該キーワードのディフォルト値は 255 で、値の範囲は 1 から 4,096 です。当該キーワードはク

ライアントサイドとサーバサイドにて使用するもので、クライアントサイドでは優先使用の権限を持っています。

15 リカバリ、バックアップ、リストア

データベース管理システムには、ハードウェアおよびソフトウェアの障害が発生することがあります。考えたくはないのですが、DBMS が何のまえぶれも無くこのような障害の影響を受けることがあり得ます。DBMS には、障害発生後にデータベース情報をリカバリする何らかの手段が必須です。実際、ファイルを基本とするシステムを DBMS に置き換える主なメリットの一つは、DBMS にリカバリ手段があることなのです。

DBMaster には、データ紛失や障害によるダウンタイムを防ぐために、先進的なデータ保護機能が組み込まれています。先進的なリカバリ、バックアップ、リストア機能を備えることによって、DBMaster は、データベースの信頼性とデータの一貫性を確実なものにしています。

15.1 データベース障害のタイプ

データベース障害は、一般にシステム障害とメディア障害の 2 つのタイプに分けられます。どちらの障害が発生しても、被害を受けたデータベースはデータ不整合あるいはデータ紛失の可能性があります。DBMS には、データベース障害をリカバリし、損傷したデータベースをバックアップファイルで置き換えるリストアのための手段が必須です。

システム障害

システム障害（インスタンス障害とも言います）は、コンピュータ・システムの主メモリの障害です。システム障害は、電源の不良、アプリケーションまたはオペレーティング・システムのクラッシュ、メモリエラー、その他の原因で発生します。システム障害の結果、突然データベース管理システムが停止します。

システム障害が発生すると、アプリケーションと、アクティブ・トランザクションが異常停止します。進行中のトランザクションと、完了したもののディスクには書き込まれていないトランザクションの正確な状態を判断するのは不可能なので、システム障害が発生した後に、この種のトランザクションをリカバリしなければなりません。システム障害は、トランザクションログまたはジャーナルファイルを使用して保護するのが最も一般的な方法です。

メディア障害

メディア障害（ディスク障害とも言います）は、コンピュータ・システムのディスクストレージの障害です。メディア障害は、通常、ディスク自身の物理的な外傷、例えば、ヘッドクラッシュ、火災、限界以上の振動や衝撃の被災によって発生します。

メディア障害が発生すると、通常、損傷したディスクにあるデータの紛失を回避する方法はありません。メディア障害によりファイルが物理的に損傷すると、データベースのリストアが必要になります。このときバックアップとリストア機能が備わっていると、データベースを正しくリストアすることができます。

15.2 データベース障害のリカバリ

データベース障害後のリカバリの目的は、コミットされたトランザクションをデータベースに確実に反映し、コミットされていないトランザクションをデータベースに反映しないこと、障害に起因する問題に煩わされること無く速やかに通常のオペレーションに復帰させることです。

DBMaster は、ジャーナルファイルとチェックポイントを使用して、リカバリを実現します。これら双方を活用して、できるだけ短時間に、可能な限りユーザーへの影響を小さくし、全てのトランザクションをリカバリします。

ジャーナルファイル

ジャーナルファイルは、データベースへの変更の全履歴とその変更の状態をリアルタイムで記録します。ジャーナルファイルに記録された変更履歴によってシステム障害をリカバリし、完了したトランザクションが行った変更でディスクに書き込まれていない操作を再試行し、異常終了したトランザクションが行った変更を元に戻します。

データベースがバックアップ・モードで走行しているときは、リストアに必要な追加情報もジャーナルファイルに記録されます。この情報は、ジャーナルファイルのバックアップが取られるまで保持され、バックアップを取った後には、新しいトランザクション用に開放されます。

リストア処理中に、バックアップ・ジャーナルファイルの情報がデータベースのバックアップコピーに追加されます。これによって、完全バックアップ後のデータベース変更を記録したジャーナルファイルのみバックアップすればよいことになります。

チェックポイントイベント

チェックポイントは、データベースの状態をクリーンにするシステムのイベントです。DBMaster は、全てのジャーナルレコードと全ての汚れたデータページをメモリバッファからディスクに書き出し、バックアップやリカバリで不要なジャーナルブロックを再生します。DBMaster は、最も古いアクティブ・トランザクションの起動前に完了している非アクティブ・トランザクションのジャーナルブロックを再生します。

チェックポイントを取ると、インスタンス障害後の起動時間が短縮されます。DBMaster は、最後にチェックポイントを取った時刻とチェックポイントを取るときに活動している全てのトランザクションのリストをジャーナルファイルヘッダに書き出します。DBMaster は、データベースリカバリ時

にこの情報を使用して、元に戻す/再試行/無視するトランザクションを決定します。

DBMaster は、ジャーナルファイルがフルになると自動的にチェックポイントを取り、ジャーナルブロックを再生して再利用しようとします。現在のトランザクションを完了させるのに必要な領域を再生することができないときは、トランザクションをアボートします。DBMaster は、データベースの起動時、終了時、オンラインバックアップ時に、自動的にチェックポイントを取ります。

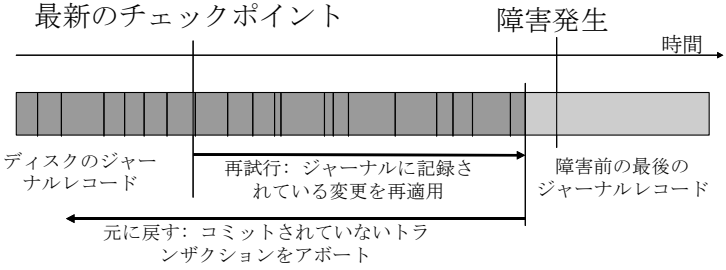
データベース管理者は、CHECKPOINT 文を実行して手動でチェックポイントを取ることができます。チェックポイントを取る最適間隔は、データベース内のアクティビティ頻度、トランザクションの平均サイズ、ジャーナルファイルの個数とサイズに依存します。これらの要因はデータベースによって大きく異なるので、最適間隔は経験を通してのみ決めることができます。チェックポイントは、ジャーナルフルの可能性を減らすと共に、データベースの起動/終了/バックアップの時間を短縮します。

チェックポイントを取るために要する時間は、最後にチェックポイントを取ってからのトランザクション数とサイズによっては非常に長くなります。チェックポイント時のアクティブ・トランザクションは、DBMaster が再生できるジャーナルレコードを調べている間は待機させられますが、ジャーナルレコードや汚れたデータページを実際にディスクに書き出している間は待機する必要はありません。

リカバリの手順

システム障害後にデータベースを起動した時、あるいは起動時にエラーが発生したときには、自動的にリカバリ処理が行われます。リカバリ処理の

際、DBMaster は常に再試行と元に戻すの 2 つのステップを実行します。



リカバリ処理の第一のステップは、ジャーナルに記録されているデータベース変更を全て再試行（再適用）することです。再試行ステップが必要なのは、システム障害の発生前に完了したトランザクションによる変更が全てディスクに書き出されてはいない可能性があるためです。完了したトランザクションのデータベース変更はジャーナルに格納されているので、再試行ステップでデータベースに書き出すことができます。再試行ステップを終えたときには、コミットされた全てのトランザクションの変更だけでなく、コミットされていない全てのトランザクションの変更もデータベースに含まれます。

リカバリ処理の第二ステップは、システム障害の発生前には完了していないトランザクションの全ての変更を元に戻す（ロールバック）することです。元に戻すステップが必要なのは、システム障害時に実行中のトランザクションは、状態があいまいなので続行することができないからです。未完了のトランザクションは削除しなければなりません。トランザクションは、成功してデータを変更するか、失敗してデータを変更しないかのどちらかであり、その定義から自己完結しています。元に戻すステップを終えたデータベースには、コミットされた全てのトランザクションの変更だけが含まれ、コミットされないトランザクションの変更は含まれません。

自動リカバリで修復できない不整合を引き起こすシステム障害やメディア障害があっても、DBMaster は、何とかしてデータベースを起動できるようにします。データベースの起動に失敗した場合、通常は、バックアップコピーからデータベースをリストアしなければなりません。データベースのバックアップを取っていない場合は、**dmconfig.ini** ファイルの **DB_ForcS** キーワードを使用して強制起動モードを設定し、強制的にデータベースを起

動させることができます。強制的にデータベースを起動させると、損傷していないデータをアンロードすることができます。

データベースを強制的に起動する

エラーが発生したときにデータベースを通常通り起動すると、自動的にデータベースのリカバリ処理が実行されます。エラーが発生しても、多くの場合、データベースは正常に起動します。データベースが起動できないようであれば、何かのディスク障害の可能性があります。ディスク障害を修復するには、最新のバックアップからデータベースをリストアしなければなりません。データベースのバックアップが無く、データベースを起動できない場合は、*強制起動*モードを使用します。

DBMaster には、強制起動オプションがあります。強制起動に必要なことは、**dmconfig.ini** ファイルの **DB_ForcS** キーワードで、強制起動モードを ON にすることだけです。このキーワードを 1 に設定すると強制起動モードが有効になり、0 にすると無効になります。強制起動モードが ON のときは、データベースを起動する際のエラーは無視されます。

強制起動でもデータベースを起動することができない場合、残された手段は以下の処理を実行することです。この処理を行う前に、全てのデータとジャーナルのバックアップを取っておきます。

➲ **バックアップの後、次の処理を実行する：**

1. **dmconfig.ini**の強制起動モードをOFFにします。(**DB_ForcS** = 0)
2. **dmconfig.ini**の起動モードを新規ジャーナル・モードにします。(**DB_SMode** = 2)
3. データベースを再起動します。
4. **dmconfig.ini**の起動モードをノーマル・モードに戻します。(**DB_SMode** = 1)

これでデータベースは起動できるはずです。新規ジャーナル・モードは、データベースのリカバリ処理を行わずに、強制的にデータベースを起動します。データベースが起動できないような深刻なエラーが発生している場合、データは不整合の状態になっています。

データベースを強制的に起動した後は、データベースの整合性をチェックします。データベース整合性チェックについては、6.12 節の「データベース整合性をチェックする」を参照してください。

15.3 バックアップの種類

バックアップは、メディア障害からデータベースを保護するために使用します。メディア障害が発生したときは、データベースのファイルが損傷して使えなくなるかもしれません。そのようなときは、最新のバックアップを使用して損傷ファイルを置き換え、データベースを再構築します。

データベースのバックアップには、3 つのタイプ、完全バックアップ、差分バックアップと増分バックアップがあります。

完全バックアップ

完全バックアップは、ある時点のデータベース全体のコピー、つまり全てのデータファイルとジャーナルファイルのコピーを取ります。オプションとして、dmconfig.ini ファイル自身のバックアップコピーを取り、データベースの設定情報を保存することもできます。完全バックアップは、オンラインあるいはオフラインのいずれの状態でも取ることができます。

完全バックアップは、データベース全体をバックアップするので大量のストレージを必要としますが、損傷したファイルにバックアップ・ファイルを上書きするだけで、比較的素早くデータベースをリストアすることができます。完全バックアップは、バックアップコピーを作成した時点までデータベースをリストアすることができます。

完全バックアップが成功すると、完全バックアップ ID が割り当てられます。完全バックアップ ID は、時刻/日付です。バックアップ ID は、完全バックアップ、差分バックアップと増分バックアップがバックアップ順に関連していることを明確にするために使用されます。差分バックアップは最新の完全バックアップをした後変更されたデータのみ記録します。差分バックアップからデータベースをリストアするため差分ベースが必要です。つまり、一つの差分バックアップを使用してデータベースが再構成できません。つまり、現在ある完全バックアップと次の完全バックアップ間に全ての増

分バックアップが存在するようにします。順序に逆らって、増分バックアップをリストアしようとすると、エラーになります。バックアップ順序は、DBMaster によって管理されています。データベースの修正、リストア、新規ジャーナルモードでのデータベース起動、バックアップ・モードの変更を行うと、新しい完全バックアップが必要です。

完全バックアップを実行する方法は、3 つあります。まず、バックアップ・サーバーを使用する方法です。詳細については、14.6 節の*「バックアップ・サーバー」*を参照して下さい。バックアップ・サーバーによる完全バックアップには、dmSQL または JServer Manager を使う方法があります。2 つ目の方法は、インタラクティブ完全バックアップです。この方法では、バックアップ・サーバーを起動させる必要がありません。JServer Manager を使用してこのタイプの完全バックアップを実行します。インタラクティブ完全バックアップを実行するための詳細については、*「JServer Manager ユーザーガイド」*を参照して下さい。3 つ目の方法は、オフライン完全バックアップです。詳細については、15.5 節の*「オフライン完全バックアップ」*を参照して下さい。

差分バックアップ

完全バックアップは全てのファイルをコピーするバックアップです。これは時間、領域がかかる操作です。この状況を避けるため、新規なバックアップタイプが必要になります：これは差分バックアップです。

差分バックアップは最新の完全バックアップのデータを基礎とします。つまり差分のベースです。そのため、差分バックアップを実行する前、差分ベースが存在しなければなりません。

差分バックアップは差分ベースを変更されたデータのみ含みます。普通に、差分ベースは連続な差分バックアップに使用されます。リストアタイムで完全バックアップと差分バックアップはこの完全バックアップを基づいて完全データベースを生成します。

DBMaster 差分バックアップはデータファイル(全部の DB と BB)のみコピーしますが、ジャーナルではありません。ジャーナルファイルは頻繁に変更されるからです。それで、差分バックアップを実行する場合、有効なジャーナルブロックのみコピーされます。

差分バックアップは最新の完全バックアップ以後変更されたデータのみレコードします。差分データベースバックアップは完全バックアップより小さいので、ユーザーは多くの頻繁なバックアップをすることができます。頻繁なバックアップはデータをなくすリスクが減少できます。差分データベースバックアップを使用する場合：

- データベースで、最後の完全バックアップからのデータのごく一部は、わずかながら比較的変化されます。もし、同じデータが何回も修正された場合は、差分バックアップはとくに有効です。
- ユーザーは頻繁なバックアップをしたいですが、頻繁な完全バックアップをしたくない場合。
- データベースをリストアする時、ユーザーはトランザクションジャーナルバックアップをロールする時間を減少したい場合。

差分バックアップを実行する方法は二つあります。一つはバックアップサーバーを使用することです。バックアップサーバーはデータベースを起動する前、同じのキーワードで設置しなければなりません。これについての詳細なのは 15.6 章バックアップサーバーに説明します。もう一つの方法はオンラインで差分バックアップです。推薦の方法は JServer Manager で実行できます。差分バックアップは*「JServer Manager ユーザーガイド」*を参照してください。

増分バックアップ

増分バックアップは、最後の増分、差分或いは完全バックアップ以降に変更されたジャーナルのコピーのみ作成するバックアップです。増分バックアップは差分、完全バックアップを実行した後実行されます。新規な完全バックアップをバックアップシーケンスで実行します。以後の増分バックアップはこのシーケンスの一部分で、他の完全バックアップと差分バックアップを使用されることができません。バックアップモードが開いていつので、増分バックアップは全部のトランザクションをレコードするジャーナルファイルから構成します。データベースが普通モード(**DB_SMODE** = 1)で実行する場合、完全バックアップ或いは差分バックアップを終了した前増分バックアップを実行できません ; レプリケーションモード(DB_SMODE = 4)で、完全バックアップ或いは差分バックアップが必要なく増分バックア

ップが実行できます。増分バックアップにはデータベースの変更分しかないので、データベースをリストアするには、増分バックアップを取る以前のデータベースの完全バックアップが必要になります。増分バックアップは、データベースがオンライン状態のときにのみ取ることができます。

増分バックアップは、ジャーナルファイルのみ取るので少量のストレージで済みます。しかしながら、リストアするときには、バックアップしたジャーナルファイルにある全てのトランザクションをロールオーバーする必要があり、完全バックアップからのリストアよりも時間がかかります。増分バックアップと差分バックアップ、完全バックアップを使用すると、直前の完全バックアップ或いは差分バックアップから増分バックアップまでのデータベースをリストアすることができます。

増分バックアップを実行する方法は、2 つあります(現在までの増分バックアップは、別の種類のバックアップとしています)。1 つ目の方法は、バックアップ・サーバーによる増分バックアップです。この方法を使うためには、バックアップ・サーバーを起動させなければなりません。バックアップ・サーバーを使った増分バックアップの実行については、14.6 節の*「増分バックアップ」*を参照して下さい。2 つ目の方法は、インタラクティブ増分バックアップです。この方法では、バックアップ・サーバーを起動させる必要がありません。JServer Manager を使用して、このインタラクティブ増分バックアップを実行します。詳細については、*「JServer Manager ユーザーガイド」*を参照して下さい。

オフライン・バックアップ

オフライン・バックアップは、データベースを終了した後に行うバックアップです。データベース管理者は、データベースを終了する予定時刻を決めて全てのユーザーに通知し、データベースから切断させます。オフライン・バックアップは、データベースを終了する前に、ユーザーが全てのトランザクションを終了し、データベースから切断しなければならないので、ユーザーにとっては不便です。オフラインの際は、完全バックアップのみ取ることができます。

オフライン・バックアップは、データベース終了後にオペレーティング・システムを使ってもデータベース・ファイルのコピーを取ることができる

ので、DBMS に必須の機能というわけではありません。DBMaster の場合も、データベース管理者がそのような方法でオフライン・バックアップを実行することもできますが、より使いやすいグラフィカル・インターフェースの JServer Manager を使って、オフライン・バックアップを行うこともできます。

オンライン・バックアップ

オンライン・バックアップは、データベースの実行中にバックアップを行います。ユーザーはデータベースから切断する必要がなく、データベースを終了する必要もありません。オンライン・バックアップは、ユーザー側では何もする必要が無いのでユーザーにとって便利です。オンライン時には、完全バックアップ、差分バックアップおよび増分バックアップを取ることができます。

DBMaster は、dmSQL とオペレーティング・システムを使用して手動でオンライン・バックアップを取る方法と、より使いやすいグラフィカル・インターフェースの JServer Manager を使ってオフライン・バックアップを実行する方法があります。

現在までのオンライン増分バックアップ

DBMaster には、現在までのオンライン増分バックアップというバックアップもあります。

オンライン増分バックアップと、複数のジャーナルファイルを用いる現在までのオンライン増分バックアップの違いは、些細ではありますが重要です。オンライン増分バックアップでは、DBMaster は最後のバックアップ以降に使用された全ジャーナルファイルをバックアップしますが、使用中のジャーナルファイルはバックアップしません。現在までのオンライン増分バックアップでは、DBMaster は使用中のジャーナルファイルも含めたすべてのジャーナルファイルをバックアップします。これは、オンライン増分バックアップが最後にコミットされたトランザクションが最後のフルジャーナルファイルに書き込まれた時点までデータベースをリストアするのに対し、現在までのオンライン増分バックアップは、使用中のジャーナルフ

ァイルがバックアップされた時点までデータベースをリストアすることを意味します。

ジャーナルファイルが 1 つしかないデータベースでは、オンライン増分バックアップと現在までのオンライン増分バックアップは、全く同じです。この場合、唯一のジャーナルファイルが使用中のジャーナルファイルです。このジャーナルファイルが、いずれの増分バックアップの際でもバックアップされます。

現在までのオンライン増分バックアップは、JServer Manager を使用して実行することができます。詳細については、*「JServer Manager ユーザーガイド」*を参照して下さい。

15.4 バックアップ・モード

バックアップ・モードは、オンライン増分バックアップを取るか、増分バックアップで取るデータの種類を指定します。また、非アクティブ・トランザクションに属するジャーナルブロックを何時開放するかを指定します。3 つのバックアップ・モード、NONBACKUP、BACKUP-DATA、BACKUP-DATA-AND-BLOB があります。

バックアップモード	表領域バックアップ・モード	ユーザー表領域(データ)	ユーザー表領域(BLOB)	システム表領域(データとBLOB)
Non Backup		×	×	×
Backup Data		○	×	○
Backup Data and BLOB	Backup BLOB Off	○	×	○
	Backup BLOB On	○	○	○

NONBACKUPモード

NONBACKUP は、完全バックアップ以降に挿入・更新したデータを全て保護しません。このモードでは、オンライン増分バックアップは取ることができません。システム障害はジャーナルから完全にリカバリすることがで

きますが、メディア障害は、データを紛失するかもしれません。アクティブ・トランザクションが使用していないジャーナルブロックは、チェックポイント後に直ちに再利用されます。ジャーナルブロックが上書きされると、最後の完全バックアップ時のデータベースしかリストアすることができなくなります。

BACKUP-DATAモード

BACKUP-DATA モードは、完全バックアップ以降に挿入・更新した全てのデータ（BLOB データを除く）を保護します。このモードでは、オンライン増分バックアップを取ることができますが、BLOB 以外のデータのみバックアップファイルに格納されます。システム障害はジャーナルから完全にリカバリすることができ、メディア障害は部分的にリカバリすることができます。直前のバックアップを使用して、メディア障害までのデータベースをリストアすることができますが、BLOB データの変更は失われます。アクティブ・トランザクションが使用していないジャーナルブロックは、チェックポイントを取り、且つジャーナルファイルがバックアップされた後に再利用されます。

BACKUP-DATA-AND-BLOBモード

BACKUP-DATA-AND-BLOB モードは、完全バックアップ以降に挿入・更新した全てのデータ（BLOB データを含む）を保護します。このモードでは、オンライン増分バックアップを取ることができ、全てのデータがバックアップファイルに格納されます。システム障害はジャーナルから完全にリカバリすることができ、ディスク障害も完全にリカバリすることができます。直前のバックアップを使用して、メディア障害時のデータベースを完全に（BLOB データを含めて）リストアすることができます。 アクティブ・トランザクションが使用していないジャーナルブロックは、チェックポイントを取り、且つジャーナルファイルがバックアップされた後に再利用されます。

表領域のBLOBバックアップモード

DBMaster は、通常、バックアップモードの設定をデータベース全体に適用します。データベースの表領域は、全て同じバックアップモードになります。バックアップモードが BACKUP-DATA-AND-BLOB の場合、全てのデータ変更（BLOB データを含め）はジャーナルに記録されます。BLOB データをジャーナルに記録すると、大きいサイズのバックアップファイルを頻繁に作成してジャーナル領域をすぐ使い果たしてしまいます。

この方式は、BLOB データを一つも失うことができないときには必要ですが、多くの場合、バックアップしなくてもよい BLOB データがあります。このような場合、どのバックアップモードを選択するかの判断が難しくなります。そのために、DBMaster は、表領域を作成するときに、表領域のバックアップモードを指定できるようにしています。

表領域にある BLOB データをバックアップする場合は、BACKUP BLOB ON オプションを使用して CREATE TABLESPACE 文を実行します。表領域の BLOB データをバックアップしない場合は、BACKUP BLOB OFF オプションを使用して CREATE TABLESPACE 文を実行します。表領域のバックアップモードは、データベースのバックアップモードと表領域のオプションの組み合わせによって次のようになります：

- データベースがBACKUP-DATA-AND-BLOBモードで走行し、表領域がBACKUP BLOB ONオプションで作成された場合、表領域のBLOBデータはバックアップされます。
- データベースがBACKUP-DATA-AND-BLOBモードで走行し、表領域がBACKUP BLOB OFF オプションで作成された場合、表領域の BLOB データはバックアップされません。
- データベースがBACKUP-DATAモードで走行する場合は、表領域作成時のオプションに関わらず、表領域のBLOBデータはバックアップされません。

表領域を作成するときに BACKUP BLOB ON または BACKUP BLOB OFF オプションを指定しないと、初期設定値として BACKUP BLOB ON になりま

す。BACKUP-DATA-AND-BLOB モードでデータベースが 走行していれば、表領域の全ての BLOB データの変更がジャーナルファイルに記録されます。

注 *差分バックアップを実行する場合、読み込み専用表領域にデータファイルはバックアップしません。表領域を読み込み専用に変更した後新しい完全バックアップは必要になります。*

ファイルオブジェクトのバックアップ・モード

データベースの一般データと BLOB データのバックアップに加え、ファイルオブジェクトのバックアップを選択することができます。ファイルオブジェクトは、バックアップデーモンによって自動完全バックアップの際にのみバックアップされます。まずバックアップ・サーバーを起動し、完全バックアップ・スケジュールをセットし、バックアップ・ディレクトリをセットする必要があります。完全バックアップの設定についての詳細は、15.6節の *「バックアップ・サーバー」* を参照して下さい。

ファイルオブジェクトには、ユーザー・ファイルオブジェクトとシステム・ファイルオブジェクトの 2 種類があります。システム・ファイルオブジェクトのみバックアップすることもできますし、両方ともバックアップする、或いはしないことも可能です。**dmconfig.ini** のキーワード **DB_BkFoM** は、ファイルオブジェクトのバックアップ・モードを指定します。

- **DB_BkFoM = 0**: ファイルオブジェクトをバックアップしない。
- **DB_BkFoM = 1**: システム・ファイルオブジェクトのみバックアップする。
- **DB_BkFoM** = 2: システム・ファイルオブジェクトとユーザー・ファイルオブジェクト双方ともバックアップする。

ファイルオブジェクトをバックアップする時(**DB_BkFoM** = 1、2)、バックアップ・サーバーは、ファイルオブジェクトの全外部ファイルを **DB_BkDir** キーワードで指定したディレクトリのサブディレクトリの"*fo*"にコピーします。スケジュールは、**DB_FBkTm** と **DB_FBkTv** で指定した完全バックアップのスケジュールに基づきます。

➲ 例

関連キーワードを含む **dmconfig.ini** ファイルからの引用：

```
[MyDB]
DB_BkSvr = 1                          ; バックアップ・サーバーを起動させる
DB_FBKTm = 01/05/01 00:00:00          ; 2001 年 5 月 1 日の深夜に開始
DB_FBKTV = 1-00:00:00                 ; 1 日間隔
DB_BkDir = /home/DBMaster/backup      ; バックアップ・ディレクトリ
DB_BkFoM = 2                          ; システム FO とユーザーFO 両方をバックアップする
```

ファイル・オブジェクトのバックアップ・モードが 2 なので、バックアップ・サーバーはデータベースの全外部ファイルオブジェクトを、"**/home/DBMaster/backup/FO**"ディレクトリにコピーします。**FO** サブディレクトリが存在しない場合、バックアップ・サーバーはそれを生成します。

FO サブディレクトリにあるファイルは、連続する番号に名前を付け替えられます。例えば、元の外部ファイルの名前が"**/DBMaster/mydb/ FO /ZZ000123.bmp**"の場合、バックアップ・サーバーはそれを **FO** サブディレクトリにコピーし、344 番目のファイルオブジェクトを意味する、'**fo0000000344.bak**'という名前に付け替えます。ソースの完全なファイル名とその新しい名前の間のマッピングは、ファイルオブジェクトのマッピング一覧ファイル **dmFoMap.his** に保存されます。ファイルオブジェクトのマッピング一覧ファイルについての詳細は、14.7節の「バックアップ履歴ファイル」を参照して下さい。

バックアップ・サーバーはまた、古いバージョンのファイルオブジェクトを **DB_BkOdr** で指定した古いバックアップ・ディレクトリにある **FO** サブディレクトリに移動します。

データベース管理者は、ファイルオブジェクトのバックアップを使用可能にすると、完全バックアップにより時間がかかる点を考慮する必要があります。完全バックアップ全体のコストには、(1)**DB_BkOdr** にセットしている場合、以前の完全バックアップのコピー; (2)全データベース・ファイルのコピー; (3)全ジャーナル・ファイルのコピー;(4)**DB_BkFoM** にセットしている場合、全ファイルオブジェクトのコピーが含まれます。また、バックアップ・エラーを防ぐために、**DB_BkDir** で指定したバックアップ・ディレクトリに全バックアップ・ファイルのために十分なスペースがあることを確認して下さい。

ストアドプロシージャのバックアップモード

ユーザーはデータベースでの BLOB データ及びファイルオブジェクトを定期的にバックアップする以外に、ストアドプロシージャのバックアップも選択できます。ストアドプロシージャのバックアップはバックアップデーモンによって自動的に完全バックアップを行う際にのみ実行されます。それを実行するには、ユーザーはまずバックアップサーバーを開いて、バックアップのスケジュール及びディレクトリを設定する必要があります。完全バックアップに関する詳細については、15.6 節のバックアップサーバーをご参照ください。

DB_BKSPM の設定によってストアドプロシージャのバックアップモードを指定することができます。

- **DB_BKSPM =** 0: ストアドプロシージャをバックアップしません
- **DB_BKSPM** = 1: 全てのストアドプロシージャをバックアップします

➲ 例

関連するキーワードを含む **dmconfig.ini** ファイルからの抜粋を以下のように示します。

```
[MyDB]
DB_BkSvr = 1                        ; バックアップサーバを起動させます
DB_FBKTm = 14/05/01 00:00:00        ; 2014 年の 5 月 1 日の午前 0 時から始まります
DB_FBKTV = 1-00:00:00               ; 間隔は毎日です
DB_BkDir = /home/dbmaster/backup     ; バックアップディレクトリ
DB_BKSPM = 1                        ; あらゆるストアドプロシージャをバックアップします
```

- ストアドプロシージャのバックアップファイルはデフォルトとして、キーワード**DB_BkDir**により指定されたプロシージャを格納するディレクトリの下のサブディレクトリにあります。ユーザーは**dmconfig.ini**ファイルにてキーワード**DB_BkOdr**を設定する場合、完全バックアップを行う過程で、前回のストアドプロシージャのバックアップシーケンスが**SP**というサブディレクトリ（**DB_BkOdr**により指定された古いバックアップディレクトリの下にある）に移動され、その後、デフォルトのディレクトリからこれらのバックアップシーケンスを削除します。

- ストアドプロシージャのバックアップ処理においては、バックアップサーバーはまずは*dmSpBk.his*という名前のファイルを生成し、それから、ストアドプロシージャのファイルをコピーし、その同時に、ファイル拡張子.s0と.b0を使用するファイル及びバックアップしたストアドプロシージャのロードに用いられるファイルが作成されます。ESQLストアドプロシージャ及びSQLストアドプロシージャについては、バックアップする必要があるファイルはソースファイル及びオブジェクトファイルで、JAVAストアドプロシージャについては、バックアップする必要があるファイルはオブジェクトファイルのみです。ストアドプロシージャのリビルドが便利なように、ESQLストアドプロシージャのソースファイルはspnameowner.ecのフォーマットでリネームされ、SQLストアドプロシージャのソースファイルはspnameowner.spのフォーマットでリネームされます。例えば、ストアドプロシージャ名は**y1**で、そしてSYSADMに属する場合、コピーしたソースファイル名は**y1SYSADM.sp**にリネームされます。
- ファイル**dmSpBk.his**はバックアップ情報の記録に用いられます。ストアドプロシージャのバックアップリストファイルの詳細については、15.7節の*バックアップの履歴ファイル*をご参照ください。
- データベース管理者は、オブジェクトファイルのバックアップ時に完全バックアップのために多くの時間を必要とすることを考えなければなりません。
- 完全な完全バックアップは下記の五つのことを含めます：（1）**DB_BkOdr**が設定される場合、以前の完全バックアップをコピーする、（2）あらゆるデータベースファイルのコピー、（3）あらゆるジャーナルファイルのコピー、（4）**DB_BkFoM**が設定される場合、あらゆるオブジェクトファイルをコピーする、（5）**DB_BKSPM**が設定される場合、あらゆるストアドプロシージャをコピーする。また、バックアップの失敗を防ぐために、ファイルを完全バックアップをする時、**DB_BkDir**により指定されたバックアップディレクトリのディスクスペースが十分であることを確保しなければなりません。

バックアップファイルの圧縮

データベースファイルは非常に大きくなることがあるので、バックアップファイルを保存するには大量の空きスペースが必要です。DBMaster は現在バックアップファイルの圧縮をサポートしています。この機能を使用すると、**dmconfig.ini** にキーワード **DB_BKZIP** を設定することで、またはデータベースが実行している中にシステムプロシージャ **SetSystemOption** を使用して BKZIP も変更できます。

- **DB_BKZIP =**1: バックアップファイルを圧縮します。
- **DB_BKZIP =**0: バックアップファイルは圧縮されません。(デフォルト)

圧縮形式は GZIP なので、GZIP 互換ツールを使えば、圧縮されたファイルの読み取りができます。

注 *バックアップファイルを有効にするようにDB_BKZIPを設定しても、FOファイルは圧縮されません。*

バックアップモードを設定する

バックアップモードは、いろいろな方法で設定することができます。**dmconfig.ini** 環境設定ファイルを編集する方法と、JServer Manager グラフィカルツールを使用する方法のいずれかの使い易い方法を採用して下さい。

データベースのバックアップモードを高位のバックアップ保護に変更する（例、NONBACKUP から BACKUP-DATA 、BACKUP-DATA から BACKUP-DATA-AND-BLOB）と、ジャーナルの使用法に影響を与えます。ジャーナルは、バックアップモードを変更する前には記録していないデータ変更を記録しはじめます。結果として、バックアップモードを変更する場合は、完全バックアップ或いは差分バックアップを取ることが必要になります。これによって、リストア処理の過程でジャーナルファイルでデータベースを更新するための出発点が整います。

データベースのバックアップモードを低位のバックアップ保護に変更する（例、BACKUP-DATA または BACKUP-DATA-AND-BLOB から NONBACKUP）場合は、単にデータ変更のジャーナル記録を止めるだけなので、何もする必要がありません。DBMaster は、直前の完全バックアップ或いは差分バックアップを出発点として使用し、リストア処理の過程でバックアップジャーナルファイルによってデータベースを更新します。しかしながら、低位のバックアップ保護に変更した後にデータベースをリストアすると、データ変更が幾つか失われるかもしれません。

バックアップモードは、**dmconfig.ini** ファイルか JServer Manager を使用してオフラインで変更することができます。バックアップモードはジャーナルの使用法に影響を与えるので、新しいバックアップモードでデータベースを起動する前に、オフライン完全バックアップを取る必要があります。オフラインでバックアップモードを変更するときは、自由にモードを変更することができますが、低位から高位のバックアップ保護に変更するときは、必ず完全バックアップを取らなければなりません。詳細については、次節の*オフライン完全バックアップ*を参照してください。

バックアップモードは、dmSQL を使用してオンラインで変更することができます。バックアップモードの変更はジャーナルの使用法に影響を与えるので、低位から高位（例、NONBACKUP から BACKUP-DATA へ、あるいは BACKUP-DATA から BACKUP-DATA-AND-BLOB へ）のバックアップ保護の変更は、完全バックアップの開始（BEGIN BACKUP）と終了（END BACKUP）の間に変更しなければなりません。

ユーザーはいつでも BACKUP-DATA-AND-BLOB から NONBACKUP モードまで変更できます。実行中に、NONBACKUP から BACKUP-DATA-AND-BLOB モードまで、或いは BACKUP-DATA-AND-BLOB から BACKUP-DATA-AND-BLOB まで直接に変更できません。

➲ 例

dmSQL を使ってバックアップ・モードを変更する：

```
dmSQL> BEGIN BACKUP;
dmSQL> SET DATA BACKUP ON;
dmSQL> END BACKUP DATAFILE;
dmSQL> END BACKUP JOURNAL;
```

替わりに次の方法を使う：

```
dmSQL> BEGIN BACKUP;
dmSQL> END BACKUP DATAFILE;
dmSQL> SET DATA BACKUP ON;
dmSQL> END BACKUP JOURNAL;
```

高位から低位のバックアップ保護に変更するときは、先ず NONBACKUP モードに変更しなければなりません。例えば、BACKUP-DATA-AND-BLOB から BACKUP-DATA モードに変更するには、最初に NONBACKUP モードに変更し、次に低位から高位へのバックアップ保護の変更規則に従って BACKUP-DATA モードに変更します。NONBACKUP への変更は、完全バックアップの開始と終了の間でなくてもいつでも行うことができます。

DMCONFIG.INIファイルを使ってバックアップ・モードを設定する

データベースがオフラインのときは、**dmconfig.ini** ファイルの **DB_BMode** キーワードを使用して直接バックアップモードを変更することができます。次にデータベースを起動するときに、新しいバックアップモードが使用されます。オンライン中に **DB_BMode** キーワードの値を変更しても、データベースを再起動するまで有効ではありません。NONBACKUP から BACKUP-DATA または BACKUP-DATA-AND-BLOB モード、BACKUP-DATA から BACKUP-DATA-AND-BLOB モードに変更するときは、オフライン完全バックアップを取るのを忘れないでください。

➲ **dmconfig.ini ファイルでバックアップモードを設定する：**

1. データベース・サーバーで、テキスト・エディタを使って**dmconfig.ini** ファイルを開きます。
2. バックアップ・モードを変更するデータベース・セクションを見つけます。
3. **DB_BMode**キーワードの値を次のいずれかにします：

   ```
   0 - NONBACKUP mode
   1 - BACKUP-DATA mode
   2 - BACKUP-DATA-AND-BLOB mode
   ```

4. データベースを再起動して、新しいバックアップモードが有効になります。

バックアップ・モードを変更するデータベース・セクションに **DB_BMode** キーワードが無いときは、**DB_BMode** キーワードを追加します。キーワードは、データベース・セクションの任意の位置に追加することができます。キーワードが無い場合は、初期値 0（NONBACKUP モード）が用いられます。

DMSQLを使ってバックアップモードを設定する

データベースがオンライン中にバックアップ・モードを変更する場合は、dmSQL で SET 文を使用します。SET 文は、オンライン完全バックアップ或いは差分バックアップの間に実行します。設定した新しいバックアップ・モードは直ちに有効になります。

➲ **dmSQL でバックアップモードを設定する：**

1. バックアップ・モードを変更するデータベースに接続します。
2. **BEGIN BACKUP**コマンドを使用してオンライン完全バックアップを開始します。
3. 完全バックアップを取っている間に、次の**SET**文のいずれかを実行してバックアップモードを変更します：

```
dmSQL> SET BACKUP OFF;
dmSQL> SET DATA BACKUP ON;
dmSQL> SET BLOB BACKUP ON;
```

4. オンライン完全バックアップを終了します。

SET BACKUP OFF 文は NONBACKUP モードに、SET DATA BACKUP ON 文は BACKUP-DATA モードに、SET BLOB BACKUP ON 文は BACKUP-DATA-AND-BLOB モードに対応します。

JSERVER MANAGERを使ってバックアップモードを設定する

データベースがオフラインの場合、JServer Manager グラフィックユーティリティを使用してバックアップ・モードを変更することができます。JServer Manager は、**dmconfig.ini** ファイルの **DB_BMode** キーワードの値を自動的に

変更します。新しいバックアップモードは、次回にデータベースを起動するときに使用されます。データベースがオンラインの場合、データベースを再起動するまで、**DB_BMode** キーワードの値は有効になりません。データベースのバックアップモードは NONBACKUP から BACKUP-DATA または BACKUP-DATA-AND-BLOB に、BACKUP-DATA から BACKUP-DATA-AND-BLOB モードに変更する場合、この前にオフライン完全バックアップをしなければなりません。JServer Manager でバックアップモードを設定する方法については、*「JServer Manager ユーザーガイド」*を参照して下さい。

15.5 オフライン完全バックアップ

オフライン完全バックアップは、オペレーティング・システムのコマンドを使用してデータベースをバックアップします。DBMaster でもこのオプションを提供していますが、バックアップ・サーバーの使用を推奨します。オフライン完全バックアップは、データベースが終了している必要があります。更に、バックアップの手順がより複雑です。

オフライン完全バックアップを取るためには、オペレーティング・システム上でのデータベース・ファイルの読み取り許可とバックアップ・ディレクトリへの書き込み許可があることが前提です。データベースの終了には、DBA、SYSDBA または SYSADM レベルのセキュリティ権限が必要です。

NON-BACKUP、BACKUP-DATA、BACKUP-DATA-AND-BLOB のいずれのバックアップモードでも、オフライン完全バックアップを取ることができます。オフライン完全バックアップを使用することによって、データベースを終了した時点までデータベースをリストアすることができます。

JServer Manager を使用したオフライン完全バックアップではファイルオブジェクトはバックアップされないことに留意してください。ファイルオブジェクトは手動でコピーします。オフライン完全バックアップからデータベースをリストアする場合、必ず正確にファイルとディレクトリ構造を複製して下さい。JServer Manager を使用したオフライン完全バックアップの実行方法については、*「JServer Manager ユーザーガイド」*を参照して下さい。

DMSQLを使ってオフライン完全バックアップをする

➲ dmSQL でオフライン完全バックアップを取る：

1. 全てのユーザーにデータベースを終了する時刻と、それ以前にデータベースから切断するように告知します。
2. データベースが起動している場合は、TERMINATE DB文でデータベースを終了します。その際にエラーが発生した場合は、問題を解決してデータベースを再起動し、再度終了します。
3. dmconfig.iniファイルを調べ、バックアップが必要な全てのファイルのディレクトリ（ファイルオブジェクトを含む）をリストアップします。
4. オペレーティング・システムを使用して、データベース・ファイル（データファイル、ジャーナルファイル、ファイルオブジェクト、dmconfig.iniを含む）をバックアップ・ディレクトリまたはバックアップ端末にコピーします。

JSERVER MANAGERを使ってオフライン完全バックアップをする

データベースがオフライン時に、JServer Manager グラフィカル· ユーティリティを使用して、オフライン完全バックアップを実行することができます。JServer Manager を使用したオフライン完全バックアップではファイルオブジェクトはバックアップされないことに留意してください。ファイルオブジェクトは手動でコピーします。オフライン完全バックアップからデータベースをリストアする場合、必ず正確にファイルとディレクトリ構造を複製して下さい。JServer Manager を使用したオフライン完全バックアップの実行方法については、*「JServer Manager ユーザーガイド」*を参照して下さい。

15.6 バックアップ・サーバー

DBMaster は手動でデータベースのバックアップを取る種々のツールや方法を提供していますが、定期的にバックアップを取る方法も知っておく必要があります。如何に信頼できる人でも、ときには忘れることがあります。

データベースをバックアップする知識によっては、データの安全が損なわれることがあります。これを解決するために、DBMaster は、バックアップ・サーバーを使用して完全に自動化したオンライン完全、差分と増分バックアップを取る便利な方法を提供します。

注 *バックアップサーバーはオンラインバックアップのみ実行できます。データベースを起動した後、バックアップサーバーが存在するからです。*

バックアップ・サーバーは、バックグラウンドで走行し、定期的なスケジュールで、またはジャーナルファイルがフルになったときに、あるいはこの両方でオンライン完全、差分と増分バックアップを取ります。バックアップ・サーバーは、データベース・サーバーと相互に通信して、いつバックアップを取るか、増分バックアップのタイプは何か、どのバックアップ・オプションを変更するかを柔軟に判断します。バックアップ・サーバーは、データベース・サーバーと同時に起動し、バックアップ・サーバーを停止させるか、データベースを終了するまで走行し続けます。

完全バックアップを実行する時、バックアップ・サーバーは、前回の完全バックアップをバックアップ・ディレクトリから古いディレクトリにコピーします。そして、ジャーナルファイルと **dmconfig.ini** を含むデータベースの全ファイルをバックアップ・ディレクトリにコピーし、以前の完全バックアップを上書きします。

差分バックアップを実行する時、バックアップ・サーバーはデータ(DB と BB)ファイルのみコピーします。ジャーナルファイルは頻繁に変更されるからです。それで、差分バックアップを実行する場合、有効なジャーナルブロックのみコピーされます。読み込み専用表領域にデータファイルは差分バックアップ以外です。

増分バックアップを実行する時、バックアップ・サーバーは、必要なジャーナルファイルをバックアップ・ディレクトリにコピーします。ユーザーは、バックアップ・ファイル名のフォーマットを指定してファイル名をセットします。

バックアップ・サーバーには、種々の構成オプションがあります。構成オプションは、バックアップ・ファイル名フォーマット、バックアップ・デ

ィレクトリの場所、バックアップを取るスケジュール、時間間隔と完全バックアップ以後の差分バックアップの最大数、バックアップを取るときのジャーナルファイルの充満度、バックアップ・ファイルの保存方法等を指定します。

データベースが実行している中に dmSQL SetSystemOption ストアドプロシージャを使用してバックアップ関連設定が行うことができます。つまり、データベースが実行している中に SetSystemOption ストアドプロシージャを使用して BKSVR、BKDIR、BKITV、DBKTV、BKTIM、BKFUL、BKFOM、BKZIP、BKCMP、BKRTS、BKCHK、 FBKTM、 FBKTV、DBKMX、BKODR、BKFRM を変更されることができます。

バックアップ・サーバーを起動する

ユーザーは DB_BkSvr キーワードを設定した後に、バックアップ・サーバーを明示的に起動させる必要はありません。DBMaster は、データベースを起動させると同時にバックアップ・サーバーも起動します。初期値は、バックアップ・サーバーを起動しません。データベースがマルチユーザーモードで起動する場合バックアップサーバーが起動します。

DB_BkSvr でバックアップサーバー状態を制御することができます。**DB_BKSvr** が 0 に設定されると、バックアップサーバーはインアクティブになる。**DB_BKSvr** が 1 に設定されると、バックアップサーバーはアクティブににになる。**dmconfig.ini** ファイルで **DB_BkSvr** キーワードが 1 に設定されること、またはデータベースが起動された後、**setsystemoption**('bksvr','1')を呼び出して BkSvr を変更することを通じて、バックアップサーバーをアクティブにします。

バックアップサーバーはアクティブである、また **dmconfig.ini** の設定ファイルで適当なバックアップパラメータを設定する場合では、任意のクライアントツール、或いはユーザーアプリケーションで使用できるシステムのストアドプロシージャ **SetSystemOption** を呼び出して、バックアップを起動することができます。

完全、差分、増分バックアップをする構文は：

```
dmSQL> Call SetSystemOption('STARTBACKUP','1'); //do full backup
dmSQL> Call SetSystemOption('STARTBACKUP','2'); //do incremential backup
```

```
dmSQL> Call SetSystemOption('STARTBACKUP','3'); //do differential backup
```

DMCONFIG.INIを使ってバックアップ・サーバーを起動する

データベースがオフライン時に、**dmconfig.ini** ファイルの **DB_BkSvr** キーワードで直接バックアップサーバーを起動させることができます。バックアップサーバーは、次にデータベースが起動する際に同時に起動します。データベースがオンラインのときは、**dmconfig.ini** ファイルのキーワード **DB_BkSvr** の値を変更してもデータベースを再起動するまで有効にはなりません。

➲ **dmconfig.ini ファイルでバックアップ・サーバーを起動させる：**

1. データベース・サーバーで、任意のASCIIテキスト・エディタを使って **dmconfig.ini**ファイルを開きます。
2. バックアップ・サーバーを起動させるデータベース・セクションを見つけます。
3. バックアップモードがBACKUP-DATAまたはBACKUP-DATA-AND-BLOBであるのを確保するべきです。**DB_Bmode** を1に設定する場合、バックアップモードはBACKUP-DATAで、2に設定する場合、バックアップモードはBACKUP-DATA-AND-BLOBです。
4. **DB_BkSvr**キーワードを1に設定し、バックアップ・サーバーを起動します。
5. データベースを再起動して、バックアップします。

DMSQLを使ってバックアップ・サーバーを起動する

データベースがオンラインするかどうか、ユーザーは dmSQL コマンドラインに以下のコマンドを使って動的なバックアップサーバーが起動させることができます。

```
dmSQL> Call SetSystemOption('BKSVR','1');
```

ユーザーは *Call SetSystemOption('BkSvr', '1')*を使用して BkSvr を変更することができます。その同時に、*Call SetSystemOptionW('option','value')*を使用して **dmconfig.ini** ファイルの **DB_BkSvr** キーワードの値を自動的に書き込まれます。

バックアップサーバーは起動される場合、**dmconfig.ini** ファイルに適当的なバックアップ変数を設定すると、ユーザーはシステムストアドプロシージャ **SetSystemOption** をコールして、クライアントツールまたはユーザーアプリケーションで使用できたストアドプロシージャをバックアップできます。

```
dmSQL> Call SetSystemOption('STARTBACKUP','1') ; //do full backup
dmSQL> Call SetSystemOption('STARTBACKUP','2'); //do incremential backup
dmSQL> Call SetSystemOption('STARTBACKUP','3'); //do differential backup
```

完全バックアップの時間間隔を変更する構文は：

```
dmSQL> Call SetSystemOption('bkitv', 'Interval');
```

JSERVER MANAGERを使ってバックアップ・サーバーを起動する

データベースがオンラインするかどうかとかかわらず、ユーザーは JServer Manager グラフィックユーティリティを使用してバックアップ・サーバーのランタイム設定をすることができます。JServer Manager は **dmconfig.ini** ファイルのキーワード **DB_BkSvr** の値を自動的に変更します。データベースがオフラインする場合、次回データベースを起動する際にバックアップサーバーも起動します。オンライン状態で JServer Manager でバックアップ・サーバーを起動する方法については、*「JServer Manager ユーザーガイド」* をご参照下さい。

差分バックアップファイル名のフォーマットを設定する

バックアップファイル名のフォーマットは、バックアップ・サーバーが差分バックアップに名前を付ける際に使用するフォーマットを指定することができます。以下は差分バックアップファイル名のフォーマットです：

DTimeStamp_DataFileName.dif(2) —差分バックアップ識別、これは必要なオプションです。

TimeStamp —Timestamp は 1970(00:00:00 GMT)1 月の秒数です。

DataFileName —データファイルが属するデータベース名。

.dif — 全ての差分バックアップファイル・オブジェクトには、拡張子 **.dif** が付きます。差分バックアップソースファイルが完全バックアップに既存しないと、ファイルの拡張子は**.dif2**でなければなりません。

2009/12/01 14:11 にはじめの差分バックアップを実行した後、また差分バックアップファイル名を生成します。ファイル名は D1259647860_DBNAME.BB.dif、D1259647860_DBNAME.DB.dif、D1259647860_DBNAME.SBB.dif と D1259647860_DBNAME.SDB.dif です。ジャーナルファイル名は D1259647860_DBNAME.JNL です。

増分バックアップファイル名のフォーマットを設定する

バックアップのファイル名形式は、バックアップ・サーバーが増分バックアップに名前を付ける際に使用するフォーマットを指定することができます。バックアップ・ファイル名には、テキスト定数も、特殊な文字列で構成されたフォーマット・シーケンス（エスケープ・シーケンス）も含むことができます。

増分バックアップのファイル名は、少なくとも次の 3 つの特殊な文字列で構成されています。完全バックアップ ID、データベース名、バックアップ ID 番号です。バックアップ・サーバーは、バックアップ・シーケンスに増分ファイルの名前を付ける際に完全バックアップ ID を割当てます。データベースをリストアする時、DBMaster は完全バックアップ ID を使用してこれに属する増分バックアップファイルを適切に再生成します。バックアップ ID 番号は、バックアップ・シーケンスにある増分バックアップの相対位置を識別します。

フォーマット・シーケンスは、エスケープ文字、サイズ、フォーマット文字の 3 つの部分から構成されています。有効なフォーマット・シーケンスは次のとおりです。

%［*x*］F—完全バックアップ ID。変数 ***x*** は以下の 1～4 のフォーマットで表現される値のいずれかです。

1:完全バックアップ ID は YYYYMMDD で表されます。例 20010917

2: 完全バックアップ ID は MMDD で表されます。例 0917

3: 完全バックアップ ID は MMDDhhmm で表されます。例 09171305

4: 完全バックアップ ID は DDhhmmss で表されます。例 17130558

%［*n*］B—ジャーナル・ファイル・バックアップ識別番号。

%［*n*］N—ジャーナル・ファイルが属するデータベース名。

%［*n*］Y—ジャーナル・ファイルがバックアップされた年。

%［*n*］M—ジャーナル・ファイルがバックアップされた月。

%［*n*］D—ジャーナル・ファイルがバックアップされた日。

エスケープ文字は%記号で表し、フォーマットシーケンスの始めを示します。バックアップファイル名のテキスト文字に%記号を含めるときは、2 つの%記号（即ち%%）にします。%記号の後には、数字または F、Y、M、D、B、N のフォーマット文字が続きます。これ以外の文字が%記号の後にあると、DBMaster は不正なバックアップファイル名フォーマットとみなしエラーを返します。

サイズは、フォーマットシーケンスが生成する文字列の長さを 1 ～ 9 で指定します。フォーマットシーケンスが指定した長さより短い文字列を生成するときは、0 を埋めて指定した長さに合せます。データベース名は名前の右側に 0 を埋め、その他は左側に 0 を埋めます。フォーマットシーケンスが指定した長さより長い文字列を生成するときは、切り捨てます。データベース名は右側を切り捨て、その他は左側を切り捨てます。長さを囲う[ ]は、長さの指定がオプションであることを示します。実際にフォーマットシーケンスを与えるときに、[と] を入力してはいけません。長さを指定しないときは、フォーマットシーケンスが生成する文字列全体を使用します。

フォーマット文字は、フォーマットシーケンスが生成する文字列のタイプを指定します。フォーマット文字は、F、Y、M、D、B、N のいずれかでなければなりません。他の文字を使用すると、DBMaster は不正なバックアップファイル名フォーマットとみなしエラーを返します。正しいフォーマット文字であっても、エスケープ文字の直後あるいはエスケープ文字と数字の直後になければ、テキスト文字とみなされます。

年・月・日はシステムの日付から取られるので、システムの日付が合っていれば、年・月・日も正しくなります。バックアップ識別番号は、バックアップを取る一連のジャーナルファイルの順序番号です。DBMaster は、バックアップ・サーバーが各ジャーナルファイルをバックアップするごとに、自動的に識別番号を付けます。

バックアップファイル名フォーマットは、**dmconfig.ini** ファイルの **DB_BkFrm** キーワードで指定します。バックアップファイル名フォーマットを指定しないときは、初期設定のフォーマット <I><TimeStamp><_><DB_BKFRM>, 例： I1234567890_%2F%4N%4B.JNL を使用します。バックアップファイル名フォーマットが生成するファイル名の長さは、255 文字以下でなければなりません。

バックアップファイル名フォーマットは、いろいろな方法で設定することができます。環境設定テキスト・ファイルを直接編集するか、或いは JServer Manager ツールを使用するか、いずれかの使いやすい方法で行います。

DMCONFIG.INIを使ってバックアップファイル名のフォーマットを設定する

データベースがオフラインのときは、**dmconfig.ini** ファイルの **DB_BkFrm** キーワードに直接バックアップのファイル名形式を設定することができます。バックアップ・サーバーは、次にデータベースが起動するときに、設定したバックアップのファイル名形式を全てのバックアップ・ジャーナルファイルに適用します。データベースがオンラインのときに **DB_BkDir** キーワードの値を変更しても、データベースを再起動しなければ有効にはなりません。

➲ **dmconfig.ini でバックアップのファイル名形式を設定する：**

1. データベース・サーバーのコンピュータ上でASCIIテキストエディタを使用して**dmconfig.ini**ファイルを開きます。
2. バックアップのファイル名形式を設定するデータベースのセクションを見つけます。
3. **DB_BkFrm**キーワードにバックアップのファイル名形式の文字列を設定します。

   注 *文字列には任意のフォーマットシーケンスとテキスト文字を含めることができますが、結果として得られるファイル名は256文字以下にしなければなりません。*

4. データベースを再起動すると、設定したバックアップのファイル名形式を使用開始します。

DMSQLを使ってバックアップファイル名のフォーマットを設定する

データベースが実行している中に **dmSQL SetSystemOption** ストアドプロシージャを使用してバックアップファイル名のフォーマットが変更できます。普通の構文は：

```
dmSQL> Call SetSystemOption('bkfrm', 'name');
```

➲ **例**

バックアップファイル名のフォーマットは I1234567890_%2F%4N%4B.JNL に変更すると、dmSQL コマンドプロンプトに以下のラインを入力してください。

```
dmSQL> Call SetSystemOption('bkfrm', 'I1234567890_%2F%4N%4B.JNL');
```

JSERVER MANAGERを使ってバックアップファイル名のフォーマットを設定する

データベースがオンラインであるかオフラインであるかに関わらず、JServer Manager グラフィックユーティリティを使用してバックアップファイル名のフォーマットを設定することができます。JServer Manager は、**dmconfig.ini** ファイルのキーワード **DB_BkFrm** の値を自動的に変更します。バックアップ・サーバーは、次にデータベースを起動する際に、このバックアップのファイル名形式を全てのバックアップ・ジャーナルファイルに適用します。JServer Manager でバックアップファイル名のフォーマットを設定する方法については、*「JServer Manager ユーザーガイド」*を参照して下さい。

バックアップ・ディレクトリを設定する

バックアップ・ディレクトリは、バックアップ・ジャーナルファイルを何処に置くかを指定します。DBMaster はシングルバックアップファイルパスとマルチバックアップファイルパスをサポートします。バックアップサーバーは自動に BkDir を作成します。しかし、異なるディスクに一つ以上のバ

ックアップディレクトリを選択するべき、データベース・ファイルとバックアップファイルの両方が失われてしまうのを防ぐことができます。
バックアップ・ディレクトリは、**dmconfig.ini** ファイルの **DB_BkDir** キーワードで定義します。**DB_BkDir** キーワードには、バックアップ・ディレクトリの絶対パスまたは相対パスを指定します。バックアップ・ディレクトリを指定しない場合は、初期設定の「backup」ディレクトリがデータベース・ディレクトリの下に作成されます。（データベース・ディレクトリは、**dmconfig.ini** ファイルの **DB_DbDir** キーワードで指定します。）バックアップ・ディレクトリのパス名は、256 字以内でなければなりません。

ただし、DBMaster データベースはレプリケーションモード（マスターとスレーブ）で実行する場合、BKDIR はシングルパスになるべきです。ユーザーは BKDIR をマルチパスを設定すると、始めてのパスが有効になり、パスサイズは無視になります。複数のデータベースが同じディレクトリにある場合は、初期設定のバックアップ・ディレクトリを利用しないようにします。この場合、データベースのバックアップ履歴情報が別のデータベースによって上書きされて、一方または双方のバックアップが使用不能の状態になります。このような問題をさけるためには、データベースを別々のデータベース・ディレクトリに入れるという方法と、各データベースのバックアップ・ディレクトリを別々に指定するという方法がありますが、前者の方法がより望ましいです。こうすることで、各ファイルがどのデータベースに属するかを正確に知ることができるようになります。

バックアップ・ディレクトリは、いろいろな方法で設定することができます。**dmconfig.ini** 環境設定ファイルを編集する方法と、JServer Manager グラフィカルツールを使用する方法のいずれかの使い易い方法を採用して下さい。

DMCONFIG.INIを使ってバックアップ･ディレクトリを設定する

データベースのオフライン時に、**dmconfig.ini** ファイルの **DB_BkDir** キーワードに直接バックアップ・ディレクトリを指定することができます。次にデータベースが起動するときに、バックアップ・サーバーが設定したディレクトリをバックアップ・ディレクトリとして使用します。データベース

のオンライン時に、**DB_BkDir** キーワードの値を変更しても、データベースを再起動するまで、その変更は有効にはなりません。

➲ **dmconfig.ini ファイルでバックアップ・ディレクトリを設定する：**

1. データベース・サーバーで、テキスト・エディタを使って**dmconfig.ini** ファイルを開きます。
2. バックアップ・ディレクトリを設定するデータベースの構成セクションを見つけます。
3. **DB_BkDir**キーワードに既存のバックアップ・ディレクトリ名の文字列を指定します。
4. データベースを再起動して、新しいバックアップディレクトリが有効になります。

DMSQLを使ってバックアップディレクトリを設定する

SetSystemOption を使用すると、データベースの起動中のみバックアップディレクトリを変更することができます。当該コマンドの一般的な構文は、次のとおりです。

```
CALL SetSystemOption('bkdir', 'path')
```

path には、新しいバックアップ・ディレクトリを指定します。指定する文字列のサイズは、256 文字以下です。

➲ **例**

dmSQL コマンド・プロンプトに次のように入力し、ディレクトリのパスを *E:/storage/database/backup/WebDB* に変更します。

```
dmSQL> CALL SetSystemOption('bkdir', 'E:/storage/database/backup/WebDB');
```

JSERVER MANAGERを使ってバックアップディレクトリを設定する

データベースがオフラインする場合、JServer Manager グラフィックユーティリティを使用して、オフラインバックアップのディレクトリを設定することができます。JServer Manager は **dmconfig.ini** ファイルのキーワード DB_BkDir の値を自動的に変更します。次回データベースが起動する際に、

バックアップサーバーはこのディレクトリをバックアップディレクトリとして使用します。データベースがオンラインする場合、JServer Manager の「ランタイムの設定」ダイアログを使用してバックアップディレクトリを直ちに変更する或いはデータベースが対話型バックアップをする際にデータベースを再起動時にそれを変更することが設定できます。データベースがオンラインかオフラインかとかかわらず、JServer Manager は新しいバックアップディレクトリにバックアップ履歴ファイルのコピーを作成します。JServer Manager でバックアップ・ディレクトリを設定する方法については、*「JServer Manager ユーザーガイド」*を参照して下さい。

複数のバックアップ・パスを設定する

DBMaster は、ユーザーのために複数のバックアップ・ファイルもサポートします。 この機能はユーザーがバックアップ・パスに保存しようとするときには便利ですが、バックアップ・パスは完了するバックアップに対して十分な領域を提供することはありません。 複数のバックアップ・オプションが設定されている場合、DBMaster は 2 次バックアップ場所にバックアップするように残りのデータを切り替えるため、バックアップを正しく実行できます。 完全バックアップまたは、差分、増分バックアップの複数のバックアップ・パスを使用できます。 複数のバックアップ・パスを使用して情報をバックアップしているとき、DBMaster には次の制約があります:

- データベース・システムがファイルのバックアップを試みているとき、各ファイルに対して1つずつパスにファイルを保存しようとします。 例えば、ファイルをバックアップ・ディレクトリ1に保存しているとき、ディレクトリにファイルを保存するのに必要な領域がない場合、ファイルはバックアップ・ディレクトリ2に切り替えられます。 すべてのバックアップ・ディレクトリが一杯であると、エラー・メッセージが返されます。
- スレーブ・サイトには、1つのバックアップ・ディレクトリをファイルのバックアップに使用できます。
- FOは、最初のバックアップ・ディレクトリにバックアップする必要があります。

- バックアップ・パスの最大数は32です。

➲ **例**

複数のバックアップ・パスを設定しているとき、DBMaster は次の構造に適合します：

```
DB_BKDIR = <BKDIR 1> <SIZE 1> < BKDIR 2> <SIZE 2> < BKDIR 3> <SIZE 3>…
< BKDIR n > : n のバックアップパス
< SIZE n > : n のバックアップパスのサイズ(単位あたり 8k)
```

従って、データベース **DB1** に対して複数のバックアップ・パスを設定する必要があります。

```
DB_BKDIR = /home/usr/dbmaster/bk 5000 /home2/backup 1000
```

home/usr/dbmaster/bk の空き領域がフルになる場合、データベースは home2/backup にバックアップされます。

古いディレクトリを設定する

古いディレクトリはシングルディレクトリまたはグループディレクトリ（32 以下）で、最後のバックアップシーケンスの前のバックアップシーケンスを保存するため使用されます。バックアップ・サーバーが以前の完全バックアップ・ファイルを配置する場所を指定します。メディア障害の際にデータベースとバックアップ・ファイルの両方を消失することが無いように、データベースとは別のディスクを選択する必要があります。

dmconfig.ini ファイルのキーワード **DB_BkOdr** で、古いディレクトリを設定します。指定しない場合、バックアップ・サーバーは以前のバックアップ・ファイルを捨てます。

DMCONFIG.INIを使って古いディレクトリを設定する

dmconfig.ini ファイルのキーワード **DB_BkOdr** に、バックアップ・サーバーが使用する古いディレクトリを直接セットすることができます。次回データベースを起動する際、バックアップ・サーバーは古いディレクトリとしてこのディレクトリを使用します。データベースがオンラインであれば、データベースを再起動するまで、キーワード **DB_BkOdr** の値の変更は無効です。

DMSQLを使って古いディレクトリを設定する

データベースが実行している中に **dmSQL SetSystemOption** ストアドプロシージャを使用して古いディレクトリを設定することができます。当該コマンドの一般的な構文は下のようになります：

```
dmSQL> Call SetSystemOption('bkodr', 'path')
```

Path は新規な古いバックアップディレクトリの完全パスです。*Path* にストリングの長さは 256 文字を超えることが許可しません。

➲ **例**

古いディレクトリのパスを *E:/storage/database/backup/WebDB* に変更すると、dmSQL コマンドプロンプトに以下のラインを入力してください。

```
dmSQL> Call SetSystemOption('bkodr', 'E:/storage/database/backup/WebDB');
```

JSERVER MANAGERツールを使って古いディレクトリを設定する

データベースがオフラインの場合、JServer Manager のグラフィカルツールを使って、古いファイルのディレクトリの位置を設定することができます。JServer Manager ツールは、**dmconfig.ini** ファイルの **DB_BkOdr** キーワードの値を自動的に変更します。データベースを再起動する際に、バックアップサーバーはこのディレクトリをバックアップディレクトリにします。データベースがオンラインの場合、次回にデータベースが再起動するまで、JServer Manager は直ちに古いバックアップディレクトリを変更するまたは必要に応じて当該変更を遅らせることができます。JServer Manager で古いバックアップディレクトリを設定する方法については、*「JServer Manager ユーザーガイド」* をご参照下さい。

差分バックアップの設定

差分バックアップ・スケジュールは、バックアップ・サーバーがオンライン差分バックアップを実行するタイミングを指定します。スケジュールは、バックアップ開始日時と、それ以降にバックアップが実行される時間間隔で構成されています。バックアップ開始日時は、バックアップ・サーバー

が最初の差分バックアップを実行する日付と時刻を表しています。時間間隔は、次回の増分バックアップまでの時間を表しています。

バックアップ開始日時は、**dmconfig.ini** ファイルのキーワード **DB_FBKTM** で指定します。キーワード **DB_FBKTM** に、yy/mm/dd hh:mm:ss 形式で日付と時刻を入力して下さい。初期設定値はありません。但し、バックアップ・サーバーを使用するために JServer Manager を使用する場合、JServer Manager は初期設定値を設け、この値を **dmconfig.ini** ファイルに書き込みます。

時間間隔は、**dmconfig.ini** ファイルのキーワード **DB_DBKTV** で指定します。始めのバックアップは **DB_FBKTM** + **DB_DBKTV** で終了されます。キーワード **DB_DBKTV** に、d-hh:mm:ss 形式で時間間隔を入力して下さい。初期設定値はありません。但し、バックアップ・サーバーを使用するために JServer Manager を使用する場合、JServer Manager は初期設定値 1-00:00:00 を設定し、この値をファイルに書き込みます。

完全バックアップ以後の差分バックアップの最大数はキーワード **DB_DBKMX** で指定します。差分バックアップの数は完全バックアップ以後で **DB_DBKMX** を超える場合、バックアップサーバーは古い差分バックアップを削除します。データベースは実行している場合、システムプロシージャ SetSystemOption を使用して **DBKMX** を変更できます。そして、キーワード **DB_BKCHK** は差分バックアップをする前、データベースをチェックするかを指定します。データベースは実行している中に、システムプロシージャ SetSystemOption を使用して **DB_BKCHK** を変更できます。

DMCONFIG.INIを使って差分バックアップを設定する

データベースがオフラインの場合、**dmconfig.ini** ファイルのキーワード **DB_BkTim** と **DB_BkItv** を使ってバックアップ・サーバーが利用するスケジュールを設定することができます。次回データベースを起動する時、バックアップ・サーバーは、差分バックアップ・スケジュールにこの設定を使います。データベースがオンラインの場合、データベースを再起動するまで、キーワード **DB_BkTim** と **DB_BkItv** の値の変更は無効です。

➲ **dmconfig.ini を使った差分バックアップスケジュールを設定：**

1. データベース・サーバーで、ASCIIテキスト・エディタを使って **dmconfig.ini**ファイルを開きます。

2. バックアップ・スケジュールを設定するデータベースのセクションを見つけます。
3. yy/mm/dd hh:mm:ss形式を使って、日付と時刻のキーワードDB_FBKTMの値を指定します。
4. ndays-HH:MM:SS形式を使って、時間間隔のキーワードDB_DBKTVの値を指定します。
5. データベースを再起動して、新しいバックアップのスケジュールが有効になります。

DMSQLを使って差分バックアップの設定を変更する

バックアップサーバーを起動するため SetSystemOption コマンドが使用できます。当該コマンドの一般的な構文は下のようになります：

```
dmSQL> CALL SETSYSTEMOPTION('BKSVR','1');
```

バックアップサーバーを起動した場合、システムストアドプロシージャ SetSystemOption をコールして、バックアップサーバーに差分バックアップを実行させます。

```
dmSQL> Call SetSystemOption('STARTBACKUP','3');
```

差分バックアップの時間間隔を変更する構文は下のようになります：

```
dmSQL> Call SetSystemOption('dbktv', 'Interval');
```

JSERVER MANAGERを使って差分バックアップの設定を変更する

データベースがオフラインの場合、JServer Manager グラフィカルユーティリティを使用して差分バックアップのスケジュールを変更することができます。JServer Manager は、自動的に dmconfig.ini ファイルのキーワード DB_FBKTM と DB_DBKTV の値を変更します。次回データベースを起動する時、バックアップ・サーバーはこれらの設定を新しい差分バックアップ・のスケジュールとして使用します。データベースがオンラインの場合、次回にデータベースが再起動するまで、JServer Manager は直ちにバックアップのスケジュールを変更するまたは必要に応じて当該変更を遅らせることができます。JServer Manager で差分バックアップの時間スケジュールを設定する方法については、*「JServer Manager ユーザーガイド」*をご参照下さい。

増分バックアップの設定

増分バックアップ・スケジュールは、バックアップ・サーバーがオンライン増分バックアップを実行するタイミングを指定します。スケジュールは、バックアップ開始日時と、それ以降にバックアップが実行される時間間隔で構成されています。バックアップ開始日時は、バックアップ・サーバーが最初の増分バックアップを実行する日付と時刻を表しています。時間間隔は、次回の増分バックアップまでの時間を表しています。

定期スケジュールに加え、後述するジャーナル・トリガー値も合わせて使用することが可能です。ジャーナル・トリガー値は、ジャーナルファイルが指定した割合に達した時に、データベースのバックアップを行います。これら 2 種類のバックアップを組み合わせることができます。定期スケジュールを定義しない場合、バックアップ・サーバーは、特定のタイミングでデータベースをバックアップすることはありません。但し、ジャーナル・ファイルが一杯になった場合には、増分バックアップを実行します。

バックアップ開始日時は、**dmconfig.ini** ファイルのキーワード **DB_BkTim** で指定します。キーワード **DB_BkTim** に、yy/mm/dd hh:mm:ss 形式で日付と時刻を入力して下さい。初期設定値はありません。但し、バックアップ・サーバーを使用するために JServer Manager を使用する場合、JServer Manager は初期設定値を設け、この値を **dmconfig.ini** ファイルに書き込みます。

時間間隔は、**dmconfig.ini** ファイルのキーワード **DB_BkItv** で指定します。キーワード **DB_BkItv** に、d-hh:mm:ss 形式で時間間隔を入力して下さい。初期設定値はありません。但し、バックアップ・サーバーを使用するために JServer Manager を使用する場合、JServer Manager は初期設定値 1-00:00:00 を設定し、この値を **dmconfig.ini** ファイルに書き込みます。

DBMaster には、増分バックアップ・のスケジュールを設定する方法がいくつかあります。**dmconfig.ini** 環境設定ファイルを編集する方法、JServer Manager を使用する方法のいずれかの使い易い方法を採用して下さい。

DMCONFIG.INIを使って増分バックアップの設定を変更する

データベースがオフラインの場合、**dmconfig.ini** ファイルのキーワード **DB_BkTim** と **DB_BkItv** を使って直接バックアップ･サーバーが利用するスケジュールを設定することができます。次回データベースを起動する時、バックアップ・サーバーは、増分バックアップ・スケジュールにこの設定を使います。データベースがオンラインの場合、データベースを再起動するまで、キーワード **DB_BkTim** と **DB_BkItv** の値の変更は無効です。

➲ **dmconfig.ini ファイルを使ってバックアップ・スケジュールを設定する：**

1. データベース・サーバーで、テキスト・エディタを使って**dmconfig.ini** ファイルを開きます。
2. バックアップ・スケジュールを設定するデータベースのセクションを見つけます。
3. yy/mm/dd hh:mm:ss形式を使って、日付と時刻のキーワード**DB_BkTim** の値を指定します。
4. d-hh:mm:ss形式を使って、時間間隔のキーワード**DB_BkItv**の値を指定します。
5. データベースを再起動して、新しいバックアップのスケジュールが有効になります。

DMSQLを使って増分バックアップの設定を変更する

SetSystemOption 文を使用すると、データベースの起動中に増分バックアップの起動時刻と起動間隔を変更することができます。増分バックアップの起動時刻を変更する一般的な構文は、次のとおりです。

```
CALL SetSystemOption('bktim', 'StartTime')
```

増分バックアップの起動間隔を変更する一般的な構文は、次のとおりです。

```
CALL SetSystemOption('bkitv', 'Interval')
```

StartTime は、増分バックアップが最初に起動する時刻を意味し、そのフォーマットは YY:MM:DD HH:MM:SS です。*Interval* は、増分バックアップが作動する時間間隔を意味し、そのフォーマットは D-HH:MM:SS です。

バックアップサーバーを起動した場合、システムストアドプロシージャ **SetSystemOption** をコールして、バックアップサーバーに増分バックアップを実行させます。

```
dmSQL> Call SetSystemOption('STARTBACKUP','2')
```

➲ **例**

dmSQL コマンド・プロンプトに次のように入力し、増分バックアップの間隔を 1 時間に設定します。

```
dmSQL> CALL SetSystemOption('bkitv', '0-1:00:00');
```

JSERVER MANAGERを使って増分バックアップの設定を変更する

データベースがオフラインの場合、JServer Manager グラフィックユーティリティを使用して増分バックアップのスケジュールを設定することができます。JServer Manager は、自動的に **dmconfig.ini** ファイルのキーワード **DB_BkTim** と **DB_BkItv** の値を変更します。次回にデータベースが起動する時、バックアップ・サーバーはこれらの設定を新しい増分バックアップのスケジュールとして使用します。データベースがオンラインの場合、次回にデータベースが再起動するまで、JServer Manager は直ちにバックアップのスケジュールを変更するまたは必要に応じて当該変更を遅らせることができます。JServer Manager で増分バックアップを設定する方法については、*「JServer Manager ユーザーガイド」*をご参照下さい。

ジャーナル・トリガー値の設定

ジャーナル・トリガー値は、ジャーナルファイルが指定した割合まで書き込まれたときに、バックアップ・サーバーがオンライン増分バックアップを取るジャーナルファイルの割合(%)の値です。ジャーナル・トリガー値とバックアップ・スケジュールを組み合わせて、ジャーナルファイルが指定パーセントに達した時点に加え、定期的にデータベースをバックアップすることができます。

ジャーナル・トリガー値は、**dmconfig.ini** ファイルの DB_BkFul キーワードで指定します。**DB_BkFul** キーワードの値は、50～100 の範囲内の整数値または 0 です。50～100 は、バックアップの実行を引き起すジャーナルファイ

ルに書き込まれた割合(%)を表します。0 はジャーナルファイルが完全に一杯になったときにバックアップを取ります。0 と 100 は実質的に同じです。どちらもジャーナルファイルが完全に一杯（100%）になったときにバックアップを取ります。ジャーナル・トリガー値を指定しない場合、バックアップ・サーバーは初期値 0 を採用します。

ジャーナル・トリガー値は、いろいろな方法で設定することができます。**dmconfig.ini** 環境設定ファイルを編集する方法と、JServer Manager グラフィカルツールを使用する方法のいずれかの使い易い方法を採用して下さい。

DMCONFIG.INIを使ってジャーナル･トリガー値を設定する

データベースがオフラインのときは、**dmconfig.ini** ファイルの **DB_BkFul** キーワードに直接ジャーナル・トリガー値を設定することができます。バックアップ・サーバーは、次にデータベースが起動するときに、設定したジャーナル・トリガー値を使用します。データベースがオンラインのときに **DB_BkFul** キーワードの値を変更しても、データベースを再起動するまで有効にはなりません。

➲ **dmconfig.ini ファイルでジャーナルトリガー値を設定する：**

1. データベース・サーバーで、テキスト・エディタを使って**dmconfig.ini** ファイルを開きます。
2. ジャーナル・トリガー値を設定するデータベースのセクションを見つけます。
3. **DB_BkFul**キーワードの値を、50～100の範囲内の整数値、又は0にします。
4. データベースを再起動して、新しいジャーナルトリガーの値が有効になります。

DMSQLを使ってジャーナル・トリガー値を設定する

SetSystemOption 文を使用すると、データベースの起動中のみジャーナル・トリガー値を変更することができます。典型的な構文は、次のとおりです。

```
CALL SetSystemOption('bkful', 'n')
```

n には、0 或いは 50～100 の間の数値を指定します。*n* を 0 に設定すると、ジャーナル・ファイルが一杯になった時にバックアップ・サーバーが作動

します。*n* を 50～100 の間の数値に指定すると、ジャーナル・ファイルの割合が指定した値に達した時に、バックアップ・サーバーが作動します。

➲ **例**

dmSQL コマンド・プロンプトに次のように入力し、ジャーナル・トリガー値を 75%に設定します。

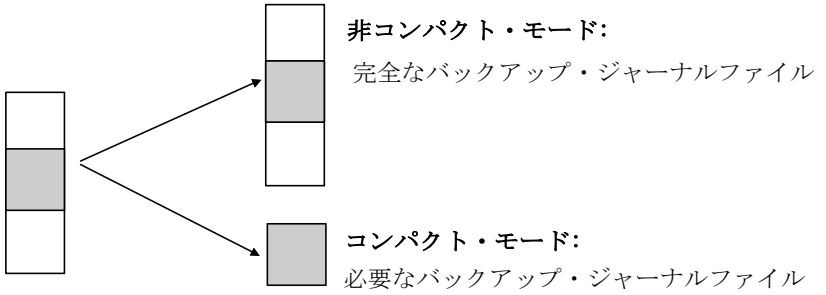
```
dmSQL> Call SetSystemOption('bkful', '75');
```

JSERVER MANAGERを使ってジャーナル・トリガー値を設定する

データベースがオフラインのときは、JServer Manager グラフィックユーティリティを使用してジャーナルトリガー値を設定することができます。JServer Manager は、**dmconfig.ini** ファイルの **DB_BkFul** キーワードの値を自動的に変更します。バックアップ・サーバーは、次回にデータベースが起動する場合、この設定を新しいジャーナルトリガー値として使用します。データベースがオンラインの場合、次回にデータベースが再起動するまで、JServer Manager は直ちにジャーナルトリガーの値を変更するまたは必要に応じて当該変更を遅らせることができます。JServer Manager でジャーナル・トリガー値を設定する方法については、*「JServer Manager ユーザーガイド」*をご参照下さい。

コンパクト・バックアップ・モードの設定

コンパクト・バックアップ・モードは、バックアップ・サーバーがオンライン増分或いは差分バックアップを取るときに、ジャーナルファイル全体をバックアップせずに、フルジャーナルブロックのみバックアップします。コンパクト・バックアップは、データベースのリストアに必要の無いジャーナルブロックは無視し、必要なジャーナルブロックのみバックアップします。コンパクト・バックアップ・モードを使用すると、バックアップ領域を節約しますが、データベースのリストアにはより多くの時間がかかります。



コンパクト・バックアップ・モードは、dmconfig.ini ファイルの DB_BkCmp キーワードで指定します。DB_BkCmp キーワードの値は 0 或いは 1 です。1 はコンパクト・バックアップ・モードを有効にし、0 は無効にします。このキーワードの値を指定しない場合は、初期値 1（有効）を使用します。

コンパクト・バックアップ・モードは、いろいろな方法で設定することができます。dmconfig.ini 環境設定ファイルを編集する方法と、JServer Manager グラフィカルツールを使用する方法のいずれかの使い易い方法を採用して下さい。

DMCONFIG.INIを使ってコンパクト・バックアップモードを設定する

データベースがオフラインのときは、dmconfig.ini ファイルの DB_BkCmp キーワードで直接コンパクト・バックアップ・モードを設定することができます。バックアップ・サーバーは、次にデータベースが起動するときに、設定したコンパクト・バックアップ・モードを使用します。データベースがオンラインのときに DB_BkCmp キーワードの値を変更しても、データベースを再起動するまで有効にはなりません。

➲ **dmconfig.ini ファイルでコンパクト・バックアップ・モードを設定する：**

1. データベース・サーバーで、テキスト・エディタを使って dmconfig.ini ファイルを開きます。
2. コンパクト・バックアップ・モードを設定するデータベースのセクションを見つけます。

3. コンパクト・バックアップ・モードを有効にする場合はDB_BkCmpキーワードの値を1に、無効にする場合は0にします。
4. データベースを再起動して、新しいコンパクトバックアップモードが有効になります。

DMSQLを使ってコンパクト・バックアップモードの設定

データベースが実行している中に **dmSQL SetSystemOption** ストアドプロシージャを使用してコンパクト・バックアップモードを変更することができます。成功的なバックアップはジャーナル· ファイル内にすべてのジャーナル· ブロックがある必要としません。キーワード **DB_BKCMP** を 1 に設定すると、バックアップサーバーはジャーナルブロックのみバックアップします、構文は以下のようになります：

```
dmSQL> Call SetSystemOption('bkcmp', '1');
```

JSERVER MANAGERを使ってコンパクトバックアップモードを設定する

データベースがオフラインの場合、JServer Manager グラフィカルユーティリティを使用してコンパクトバックアップモードを設定することができます。JServer Manager は、**dmconfig.ini** ファイルの **DB_BkCmp** キーワードの値を自動的に変更します。バックアップサーバーは、次回にデータベースが起動する場合、この設定を新しいコンパクトバックアップモードとして使用します。データベースがオンラインの場合、次回にデータベースが再起動するまで、JServer Manager は直ちにコンパクトバックアップモードを変更するまたは必要に応じて当該変更を遅らせることができます。JServer Manager でコンパクトバックアップモードを設定する方法については、*「JServer Manager ユーザーガイド」* をご参照下さい。

完全バックアップのスケジュール

完全バックアップ・スケジュールは、バックアップ・サーバーがオンライン完全バックアップを実行する日時を指定します。そのスケジュールは、開始日時と時間間隔の 2 つの部分で構成されています。バックアップ開始日時は、バックアップ・サーバーが最初の完全バックアップを実行する日

付と時刻を決定します。時間間隔は、それ以降の完全バックアップの間隔を決定します。

データベースのバックアップに、増分と完全（差分）バックアップ・スケジュールを組み合わせることができます。完全バックアップ・スケジュールを指定しない場合、バックアップ・サーバーは定期的に完全バックアップを行いません。

バックアップ開始日時は、**dmconfig.ini** ファイルのキーワード **DB_FBkTm** で指定します。キーワード **DB_FBkTm** に日付と時刻を **yy/mm/dd hh:mm:ss** 形式で入力して下さい。初期設定値はありません。

時間間隔は、**dmconfig.ini** ファイルのキーワード **DB_FBkTv** で指定します。キーワード **DB_FBKTV** に時間間隔を **d-hh:mm:ss** フォーマットで入力して下さい。初期設定値はありません。

最後、キーワード **DB_BKCHK** は完全バックアップと差分バックアップする前にデータベースをチェックするかを指定します。キーワード **DB_BKRTS** は完全バックアップを実行する場合にバックアップサーバーが読み取り表領域ファイルを含むかどうか指定します。この二つの機能を使用すると、ユーザーは **dmconfig.ini** にキーワード **DB_ BKCHK** と **DB_BKRTS** を設定できます、またはデータベースが実行している中にシステムプロシージャ SetSystemOption を使用して **DB_ BKCHK** と **DB_BKRTS** を変更します。

DMCONFIG.INIを使って完全バックアップ･スケジュールを設定する

データベースがオフラインの場合、**dmconfig.ini** ファイルのキーワード **DB_FBkTm** と **DB_FBkTv** に直接、完全バックアップ・スケジュールをセットすることができます。次回データベースを起動する際、バックアップ・サーバーは完全バックアップ・スケジュールにこれらの設定を使用します。データベースがオンラインの場合、キーワード **DB_FBkTm** と **DB_FBkTv** の変更はデータベースを再起動するまで反映されません。

➲ dmconfig.ini ファイルを使ってバックアップ・スケジュールを設定する：

1. データベース・サーバーで、テキスト・エディタを使ってdmconfig.iniファイルを開きます。
2. バックアップ・スケジュールを設定するデータベース・セクションを見つけます。
3. yy/mm/dd hh:mm:ss形式を使って、日付と時刻のキーワードDB_FBkTmの値を指定します。d-hh:mm:ss形式を使って、時間間隔のキーワードDB_FBkTvの値を指定します。
4. データベースを再起動して、新しい完全バックアップのスケジュールが有効になります。

DMSQLを使って完全バックアップ・スケジュールを設定する

データベースが実行している場合に完全バックアップの起動時間と時間間隔を変更するためプロシージャ SetSystemOption が使用されます。完全バックアップの起動時間を変更する構文は下のようになります：

```
dmSQL> Call SetSystemOption('fbktm', 'StartTime');
```

完全バックアップの時間間隔を変更する構文は下のようになります：

```
dmSQL> Call SetSystemOption('fbktv', 'Interval');
```

StartTime は初めての完全バックアップの起動時間です。フォーマットは YY:MM:DD HH:MM:SS です。 *Interval* は完全バックアップの発生間隔です、フォーマットは D-HH:MM:SS になります。

バックアップサーバーは起動した後、システムストアドプロシージャ SetSystemOption をコールして完全バックアップが始めます。

```
dmSQL> Call SetSystemOption('STARTBACKUP','1') ;
```

➲ 例

完全バックアップの時間間隔は一時間を設定すると、dmSQL コマンドプロンプトに以下のラインを入力してください：

```
dmSQL>Call SetSystemOption('fbktv', '0-1:00:00');
```

JSERVER MANAGERを使って完全バックアップ・スケジュールを設定する

データベースがオフラインの場合、JServer Manager ツールを使用して完全バックアップのスケジュールを設定することができます。JServer Manager は自動的に **dmconfig.ini** ファイルの **DB_FBkTm** と **DB_FBkTv** キーワードの値を変更します。次回にデータベースが起動する際に、新しい完全バックアップ・スケジュールが適用されます。JServer Manager で完全バックアップ・のスケジュールを設定する方法については、*「JServer Manager ユーザーガイド」*をご参照下さい。

ファイルオブジェクトのバックアップ・モード

ファイルオブジェクト・バックアップ・モードは、完全バックアップの際にファイルオブジェクトをバックアップさせるかどうかを決定します。システム・ファイルオブジェクトのみバックアップ、又はシステム・ファイルオブジェクトとユーザー・ファイルオブジェクト両方をバックアップのいずれかを選択することもできます。

ファイルオブジェクト・バックアップ・モードを設定する方法はいくつかあります。データベースの起動時に環境設定ファイルのキーワード **DB_BkFoM** が参照されますが、ランタイム時に dmSQL や JServer Manager を使ってその設定を変更することもできます。

バックアップ・サーバーは、以前のバックアップを **DB_BkOdr** で指定した古いディレクトリに移動します。

ファイルオブジェクトのバックアップを使用可能にすると、完全バックアップにより時間がかかります。完全バックアップ全体のコストには、(1)**DB_BkOdr** にセットしている場合、以前の完全バックアップのコピー、(2)全データベース・ファイルのコピー、(3)全ジャーナル・ファイルのコピー、(4)**DB_BkFoM** にセットしている場合、全ファイルオブジェクトのコピーが含まれます。また、バックアップ・エラーを防ぐために、**DB_BkDir**(必要な場合は **DB_BkOdr** も)で指定したバックアップ・ディレクトリに全バックアップ・ファイルのために十分なスペースがあることを確認して下さい。

ファイルオブジェクトは、完全バックアップが実行された時にバックアップ・ディレクトリに生成された FO ディレクトリにコピーされます。ファイルオブジェクトは、バックアップ・ファイルオブジェクト・ディレクトリにコピーされる際に、順番に名前が付けられます。*fo* サブディレクトリにあるファイルの名前は、FO で始まり 10 桁の通し番号が付きます。バックアップしたファイル・オブジェクトには、全て拡張子.BAK が付きます。元のファイル名とパスとバックアップしたファイル名の間のマッピングは、ファイルオブジェクトのマッピング・ファイル ***dmFoMap.his*** に記録されます。

バックアップされたファイルオブジェクトのマッピング・ファイル

ファイルオブジェクトのマッピングファイル **dmFoMap.his** は、「**DB_BkDir/FO**」ディレクトリに生成されます。このファイルは、純粋な ASCII テキストファイルで、外部ファイル名とバックアップされたファイル名の記録に用いられます。そのフォーマットは以下のとおりです。

```
Database Name: DBSAMPLE5
Begin Backup FO Time: 2013/04/12 09:21:32
FO Backup Directory: C:\DBMaster\5.4\SAMPLES\DATABASE\backup\FO\
[Mapping List]
s, fo0000000000.bak, "C:\DBMaster\5.4\SAMPLES\DATABASE\backup\FO\ZZ000001.bmp"
u, fo0000000001.bak, "C:\DBMaster\5.4\SAMPLES\DATABASE\backup\FO\image.jpg"
....
s, fo0000002345.bak, "C:\DBMaster\5.4\SAMPLES\DATABASE\backup\FO\ZZ00AB32.txt"
```

[Mapping List]の前の内容は、ユーザーの参照のための説明です。[Mapping List]の後ろの各行は、ファイルオブジェクトの種類(s = システム・ファイルオブジェクト、u = ユーザー・ファイルオブジェクト)、**FO** サブディレクトリにある新しいファイル名、その元のファイル名とパスを表します。このマッピングファイルは、ファイルオブジェクトのリストア時に必要になります。

DMCONFIG.INIを使ってファイルオブジェクトのバックアップ・モードを設定する

dmconfig.ini のキーワード **DB_BkFoM** は、ファイルオブジェクトのバックアップ・モードを指定します。

- **DB_BkFoM** = 0：ファイルオブジェクトをバックアップしない。
- **DB_BkFoM** = 1：システム・ファイルオブジェクトのみバックアップする。
- **DB_BkFoM** = 2：システム・ファイルオブジェクトとユーザー・ファイルオブジェクト双方ともバックアップする。

ファイルオブジェクトをバックアップする時(**DB_BkFoM** = 1、2)、バックアップ・サーバーは、ファイルオブジェクトの全外部ファイルを **DB_BkDir** キーワードで指定したディレクトリのサブディレクトリの"*fo*"にコピーします。スケジュールは、**DB_FBkTm** と **DB_FBkTv** で指定した完全バックアップのスケジュールに基づきます。

➲ 例

関連キーワードを含む **dmconfig.ini** ファイルからの引用：

```
[MyDB]
DB_BkSvr = 1                        ; バックアップ・サーバーを起動させる
DB_FBKTm = 01/05/01 00:00:00        ; 2001 年 5 月 1 日の深夜に開始
DB_FBKTV = 1-00:00:00               ; 1 日間隔
DB_BkDir = /home/DBMaster/backup    ; バックアップ・ディレクトリ
DB_BkFoM = 2                        ; システム FO とユーザーFO 両方をバックアップする
```

ファイル・オブジェクトのバックアップ・モードが 2 なので、バックアップ・サーバーはデータベースの全外部ファイルオブジェクトを、"**/home/DBMaster/backup/FO**"ディレクトリにコピーします。**FO** サブディレクトリが存在しない場合、バックアップ・サーバーはそれを生成します。

DMSQLを使ってファイルオブジェクトのバックアップ・モードを設定する

SetSystemOption 文を使用すると、データベースの起動中のみファイルオブジェクトのバックアップ・モードを変更することができます。ファイルオブジェクトのバックアップ・モードを変更する一般的な構文は、次のとおりです。

```
dmSQL> Call SetSystemOption('bkfom', 'n');
```

n には、0 又は 1、或いは 2 を指定します。*n* を 0 に設定すると、ファイルオブジェクト・バックアップ・モードは OFF になります。*n* を 1 に設定すると、完全バックアップの際にシステム・ファイルオブジェクトを全てバックアップします。*n* を 2 に設定すると、完全バックアップの際にシステム・ファイルオブジェクトとユーザー・ファイルオブジェクトを全てバックアップします。

➲ 例

dmSQL コマンド・プロンプトに次のように入力し、システム・ファイルオブジェクトとユーザー・ファイルオブジェクトを全てバックアップするように、バックアップ・サーバーを設定します。

```
dmSQL> CALL SetSystemOption('bkfom', '2');
```

JSERVER MANAGERを使ってファイルオブジェクトのバックアップ・モードを設定する

ファイルオブジェクトのバックアップモードの設定は、完全バックアップをしている間にファイルオブジェクトのコピー方法に影響を与えます。**ファイルオブジェクトをバックアップしないモード**を選択する場合には、ファイルオブジェクトがコピーできません。**ファイルオブジェクトのみバックアップするモード**を選択する場合には、ファイルオブジェクトが自動完全バックアップのプロセスでコピーされます。**システムとユーザーファイルオブジェクトをバックアップするモード**を選択する場合には、システムファイルオブジェクト及びユーザーファイルオブジェクトが両方とも自動完全バックアップのプロセスでコピーされます。JServer Manager でファイルオブジェクトのバックアップモードを設定する方法については、*「JServer Manager ユーザーガイド」*をご参照下さい。

ストアドプロシージャのバックアップモード

ストアドプロシージャのバックアップモードは、データベース管理者がバックアップサーバが完全バックアップ時にストアドプロシージャをバックアップするかどうかを決定することができます。

ストアドプロシージャのバックアップモードの設定方法は幾つかあります。

データベースがスタートする場合、キーワード **DB_BKSPM** によってその設定方法を決めることができますが、dmSQL または JServer Manager ツールで実行している間に、それが変更されることがあります。

バックアップサーバに通して、前回のバックアップを **DB_BkOdr** により指定された古いバックアップディレクトリに移動することができます。

ストアドプロシージャのバックアップを開始すると、データベースは完全バックアップを完了するために多くの時間を必要とするようになります。この時間はデータベース内のファイルオブジェクトの数によって決めます。

完全な完全バックアップは下記の五つのことを含めます：（1）**DB_BkOdr** が設定される場合、以前の完全バックアップをコピーする、（2）あらゆるデータベースファイルのコピー、（3）あらゆるジャーナルファイルのコピー、（4）**DB_BkFoM** が設定される場合、あらゆるオブジェクトファイルをコピーする、（5）**DB_BKSPM** が設定される場合、あらゆるストアドプロシージャをコピーする。また、バックアップの失敗を防ぐために、ファイルを完全バックアップをする時、DB_BkDir により指定されたバックアップディレクトリのディスクスペースが十分であることを確保しなければなりません。

ストアドプロシージャが完全バックアップを行う際にバックアップディレクトリにて作成されたサブディレクトリ SP にコピーされます。その場合、ファイルオブジェクトがリネームされます。コピーされたストアドプロシージャのバックアップ情報はファイル **dmSpBk.his** に記録されます。

バックアップストアドプロシージャの情報ファイル

バックアップ情報は、データベース名、バックアップ時間及びバックアップされた行の情報記録用のバックアップリストで構成します。バックアッ

プ情報ファイル **dmSpBk.his** は **DB_BkDir** によって指定されたディレクトリの下にあるサブディレクトリ SP にて作成されます。このファイルはコピーされたストアドプロシージャのバックアップ情報を記録する純粋な ASCII テキストファイルで、そのフォーマットは下記のようになります。

```
Database Name: MYDB
Begin Backup SP time: 2014/06/20 09:13:06
[Backup List]
SQLSP, SYSADM, SP1, SP1SYSADM.luo, SP1SYSADM.sp
ESQLSP, SYSADM, A4, A4SYSADM.dll, A4SYSADM.ec
....
JAR, "", "", employee.jar, ""
```

「バックアップリスト」前の内容はただユーザー参照用の説明です。「バックアップリスト」後の各ラインはそれぞれ一つのレコードを表します。当該レコードにストアドプロシージャのタイプ、ストアドプロシージャの所有者、ストアドプロシージャ名、ストアドプロシージャのオブジェクトファイル名及びそのソースファイル名を含めます。このバックアップ情報ファイルはストアドプロシージャの回復のために必要であります。

DMCONFIG.INIファイルを使ってストアドプロシージャのバックアップモードを設定する

ファイルのキーワード **DB_BKSPM** は、ストアドプロシージャのバックアップモードを決定します。

- **DB_BKSPM** = 0:ストアドプロシージャをバックアップしません
- **DB_BKSPM** = 1:あらゆるストアドプロシージャをバックアップします

➲ 例

指定されたストアドプロシージャのバックアップパラメータは **dmconfig.ini** ファイルのエントリは以下のようになります。

```
[MyDB]
DB_BkSvr = 1                          ; バックアップサーバを起動します
DB_FBKTm = 14/05/01 00:00:00          ; 2014 年 5 月 1 日の午前零時に開始
DB_FBKTV = 1-00:00:00                 ; 時間間隔は毎日一回です
DB_BkDir = /home/dbmaster/backup      ; バックアップディレクトリ
DB_BKSPM = 1                          ; あらゆるストアドプロシージャをバックアップします
```

上記の例では、**DB_BKSPM** の値は 1 であるため、バックアップサーバがあらゆるストアドプロシージャを **DB_BkDir** によって指定されたディレクトリの下にあるサブディレクトリ **SP** にバックアップします。サブディレクトリ **SP** が存在しない場合は、バックアップサーバーはそれを作成します。

DMSQLを使ってストアドプロシージャのバックアップモードを設定する

データベースが動作している間に、プロシージャ **SetSystemOption** はストアドプロシージャのバックアップモードの変更に用いられます。ファイルオブジェクトのバックアップモードを変更する一般的な構文は以下のようになります。

```
dmSQL> Call SetSystemOption('BKSPM', 'n');
```

上記の例では、ユーザーは n に 0 または 1 を割り当てることができます。0 を割り当てる場合は、バックアップサーバはストアドプロシージャをバックアップしていませんが、1 を割り当てる場合は、バックアップサーバは完全バックアップを行っている間にあらゆるストアドプロシージャをバックアップしています。

➲ **例**

バックアップサーバを設定することによって、あらゆるストアドプロシージャで完全バックアップをします。dmSQL コマンドプロンプトに下記のコマンドを入力します。

```
dmSQL> Call SetSystemOption('BKSPM', '1');
```

JSERVER MANAGERツールを使ってストアドプロシージャのバックアップモードを設定する

データベースがオンラインするかどうかとかかわらず、ユーザーは JServer Manager グラフィックユーティリティを使用してバックアップストアドプロシージャを設定することができます。JServer Manager は **dmconfig.ini** ファイルのキーワード **DB_BKSPM** の値を自動的に変更します。データベースがオフラインする場合、次回データベースを起動する際にこの新しい設定が有効になります。データベース起動時にまたは JServer Manager の**ランタイム**

設定でストアドプロシージャのバックアップモードを設定する方法については、*「JServer Manager ユーザーガイド」*をご参照下さい。

バックアップ・サーバーをインアクティブにする

DBMaster はデータベースが起動する場合、自動にバックアップサーバーを起動します。バックアップサーバーはデイフォルトに無効です。**DB_BkSvr** でバックアップサーバーの状態を変更できます。**DB_BKSvr** は 0 を設定すると、バックアップサーバーはインアクティブになり、1 を設定してアクティブになります。**dmconfig.ini** ファイルのキーワード **DB_BkSvr** を 0 に設定しますまたはデータベースを起動した後、BkSvr with *call setsystemoption('bksvr','01')* を変更します。

DMCONFIG.INIを使ってバックアップ・サーバーをインアクティブにする

データベースがオフラインの場合、**dmconfig.ini** ファイルの **DB_BkSvr** キーワードを使用して直接バックアップサーバーを停止することができます。バックアップサーバーは、次回にデータベースが起動するときには起動しません。データベースがオンラインの場合、**DB_BkSvr** キーワードの値を変更してもデータベースが再起動するまで有効にはなりません。

➲ **dmconfig.ini ファイルでバックアップ・サーバーをインアクティブする：**

1. データベース・サーバーで、テキスト・エディタを使って**dmconfig.ini** ファイルを開きます。
2. バックアップ・サーバーをインアクティブするデータベース・セクションを見つけます。
3. **DB_BkSvr**キーワードの値を0にして、バックアップ・サーバーをインアクティブさせます。
4. データベースを再起動します。

DMSQLを使ってバックアップ・サーバーをインアクティブにする

データベースが動作している間に、プロシージャ **SetSystemOption** はバックアップサーバのステータスの変更に用いられます。バックアップサーバのステータスを変更する一般的な構文は以下のようになります。

```
dmSQL> Call SetSystemOption('bksvr', 'n');
```

ｎは 0 または 1 に設定可能です。0 に設定する場合は、バックアップサーバをアクティブにしないを表し、1 に設定する場合は、バックアップサーバをあくてぃぶにするのを表します。

➲ **例**

バックアップサーバをインアクティブにするには、dmSQL コマンドプロンプトに下記のコマンドを入力します。

```
dmSQL> Call SetSystemOption('bksvr', '0');
```

JSERVER MANAGERを使ってバックアップ・サーバーをインアクティブにする

データベースがオフラインの場合、JServer Manager グラフィカルユーティリティを使用してバックアップ・サーバーを停止することができます。JServer Manager は、**dmconfig.ini** ファイルの **DB_BkSvr** キーワードの値を自動的に変更します。バックアップ・サーバーは、次回にデータベースが起動したときには起動しません。データベースがオンラインの場合、バックアップ・サーバーを OFF にしても、データベースを再起動するまでその変更は有効になりません。JServer Manager でバックアップ・サーバーのインアクティブにする方法については、*「JServer Manager ユーザーガイド」*をご参照下さい。

15.7 バックアップ履歴ファイル

自動バックアップはバックアップサーバーを使ってバックアップのジャーナルファイルの情報を保存します、これらの情報を自動的にバックアップ履歴ファイルに保存します。

バックアップ履歴ファイルを配置する

バックアップ履歴ファイルはテキストファイルで、**dmconfig.ini** ファイルのキーワード **DB_BKDIR** の第一ディレクトリに置いています。これは ***dmBackup.his*** という名前のファイルで、オンラインバックアップ・ディレクトリに生成されます。バックアップ履歴ファイルは、データベースのリストア時に自動的に使用されます。但し、オフラインバックアップは ***offBackup.his*** という名前で記録されます。

バックアップ履歴ファイルを理解する

バックアップ履歴ファイルには、ID 番号のほか、ファイル名、バックアップした日付、時刻に分けられた全ての情報があります。DBMaster では、バックアップ履歴ファイルを使って、バックアップの順序をたどり、各段階で完全、差分バックアップと増分バックアップの整合性を取ります。

以下はバックアップ履歴ファイルの形式を示しています。

```
<バックアップ ID>: ジャーナルファイル名 -> アーカイブファイル名: 日時: イベント
```

各行は、<ジャーナルファイル名>のジャーナルファイルが、<アーカイブファイル名>のアーカイブファイルに、<イベント>の理由により、<日時>の時にコピーされたことを意味します。<イベント>はバックアップの理由を示し、JOURNAL-FULL、TIME-OUT、USER、ON-LINE-FULL-BACKUP-BEGIN、ON-LINE-FULL-BACKUP、ON-LINE-FULL-BACKUP-END、のいずれかです。JOURNAL-FULL は、ジャーナルが一杯になったので増分バックアップが行われたことを示します。TIME-OUT は、スケジュールに定めた予定時刻がきたので差分或いは増分バックアップが実行されたことを示します。USER は、ユーザーによる手動増分バックアップを表します。
ON-LINE-FULL-BACKUPxxxx は、完全バックアップを意味します。

バックアップ履歴ファイルを使用する

ジャーナルフルが頻発するようなら、バックアップ・ジャーナルフルのパーセントをもっと低くするか、バックアップ間隔を短くすべきことを意味します。バックアップ間隔が短すぎるかどうかは、バックアップ履歴ファイルをチェックすることによって見つけることができます。 バックアップ

履歴ファイルの中に同じジャーナルファイルが連続しているならば、バックアップ間隔が短すぎるかもしれません。この状況だと、各ファイルは少数の変更ブロックしか含まないのでディスクを無駄に消費します。これを避けるには、コンパクト・バックアップ・モードを有効にするか、バックアップ間隔を長くします。

多くのジャーナルファイルが毎回バックアップされるならば、バックアップ間隔が長すぎることを意味するかもしれません。この状況は、ディスク障害が発生したときに多くのデータが失われる可能性があるため危険です。増分バックアップを取るときにジャーナルファイルが 1 個バックアップされるのが理想的です。差分バックアップはデータファイルのみ目指しますが、ジャーナルファイルではありません。これによって、ストレージが節約されジャーナルデータを失うリスクが低くなります。

バックアップ・サーバーを使用しているときでも、定期的に完全バックアップを取ることによって、メディア障害のリストア時間をより短くすることができます。これは、必要なバックアップストレージの量も少なくします。

バックアップ・サーバーが走行しているときでも、手動で差分と増分バックアップを取ることができます。

ファイルオブジェクトのバックアップ履歴ファイル

ファイルオブジェクトのバックアップ履歴ファイル、**dmFoMap.his** には、環境設定パラメータに基づいてバックアップされた全ファイルオブジェクトの記録が保管されます。**dmFoMap.his** は、*<DB_BkDir>/FO* ディレクトリにあり、元の外部ファイル名とバックアップファイル名を記録した純粋な ASCII テキストファイルです。

ファイル・フォーマットは、以下のとおりです。

```
Database Name: MYDB
Begin Backup FO Time: 2001.5.13 2:33
FO Backup Directory: /DBMaster/mydb/backup/FO (i.e. DB BkDir/FO)
[Mapping List]
s, fo0000000000.bak, "/DBMaster/mydb/fo/ZZ000001.bmp"
u, fo0000000001.bak, "/home2/data/image.jpg"
....
```

```
s, fo0000002345.bak, "/DBMaster/mydb/fo/ZZ00AB32.txt"
```

最初のカラムの **s** や **u** は、それぞれシステム・ファイルオブジェクトとユーザー・ファイルオブジェクトを表します。2 番目のカラムは、バックアップ名を与え、3 番目のカラムは元のファイルオブジェクトの完全名と絶対パスを与えます。

ストアドプロシージャのバックアップ履歴ファイルの理解

ストアドプロシージャのバックアップ履歴ファイル **dmSpBk.his** は、ストアドプロシージャのバックアップ設定パラメータでバックアップされたストアドプロシージャのレコードを保持します。**dmSpBk.his** はストアドプロシージャのバックアップ情報を記録する純粋な ASCII テキストファイルで、**DB_BkDir** により指定されるディレクトリの下にあるサブディレクトリ SP にあります。

当該ファイルのフォーマットは下記のようになります。

```
Database Name: MYDB
Begin Backup SP time: 2014/06/20 09:13:06
[Backup List]
SQLSP, SYSADM, SP1, SP1SYSADM.luo, SP1SYSADM.sp
ESQLSP, SYSADM, A4, A4SYSADM.dll, A4SYSADM.ec
....
JAR, "", "", employee.jar, ""
```

上記の例では、最初のカラムはストアドプロシージャのタイプを表し、二番目のカラムはストアドプロシージャの所有者を表し、三番目のカラムはストアドプロシージャ名を表し、四番目及び五番目のカラムはそれぞれオブジェクトファイル名及びそのソースファイル名を表します。

15.8 レプリケーションデータベースにバックアップする

スレーブデータベース以外の普通データベースとマスターでは、ユーザーは完全バックアップ、差分バックアップ、および増分バックアップをすることができます。方法は以前と同じです。しかし、JServerManager は、マス

ターデータベースでインタラクティブに増分バックアップをすることができません。さらに、マスターデータベースでインタラクティブに完全バックアップをする場合、増分バックアップを消去することができません。

マスターデータベースでは、潜在的に、完全バックアップより先に多くの増分バックアップファイルがまだレプリケーションためにバックアップシーケンスに残されていることを注意してください。同時に、完全バックアップのために、レプリケーションサーバーは増分バックアップファイルを消去しない可能性があります。それで、長い期間に次の完全バックアップが実行されていない場合、大量のファイルが DB_BKDIR に存在しています。

一言で、レプリケーションサーバーはバックアップサーバーとよく協力する必要があって、彼らはお互いに妨げることができません。一方では、バックアップはレプリケーションを壊すことができません。言い換えるなら、完全、差分バックアップをする中或いは終了した場合、レプリケーションサーバーはスレーブサイトにすべてのトランザクションを複写することができます。つまり、レプリケーションはバックアップシーケンスを壊すことができません。

バックアップシーケンスを使用してマスターデータベースがリストアできます。ただし、リストアした後、レプリケーションデータベースは終了します。ユーザーはレプリケーションデータベースを続くと、すべてのスレーブデータベースが新しいマスタデータベースを代わらなければなりません。つまり、ユーザーはマスタデータベースファイルをコピーして全部のスレーブファイルを代わります。

レプリケーションデータベースにいくつの制限があります：

- マスタデータベースが起動する場合、**dmconfig.ini** ファイルの**BMODE**と**BKSVR**は起動状態を設定します。
- マスターとスレーブデータベースのランタイムに**BMODE**, **BKSVR**, **BKDIR** を変更することはできません。例えば、setsystemoption ('bkdir','new-bkdir')をコールしてエラーを返します。
- マスターとスレーブデータベースサイトに、**BKDIR**キーワードがシングルパスになるべきです。ユーザーはマルチパスを設定すると、第一番のパスを使用されて、パスサイズが無視になります。

- マスターデータベースにJServerManagerで増分バックアップをすることができません。
- スレーブデータベースに、完全、差分と増分バックアップをすることができません。

15.9 リストア選択肢

データベースのリストアは、直前の完全バックアップのデータベースを再構築し、バックアップされたジャーナルファイルの変更を施したデータベースを作成します。

リストア選択肢を分析する

利用できるリストア選択肢が何ですか。

この質問の回答は、データベースが BACKUP モードかどうかによって異なります。

- データベースがNONBACKUPモードのときは、直前の完全バックアップでリストアしデータベースを再起動することが、ディスク障害時の唯一の選択肢です。直前の完全バックアップ以降に行われた全ての作業は失われるので、再入力しなければなりません。この場合は以下の質問に答える必要はありません。
- データベースがBACKUP（BACKUP-DATAまたはBACKUP-DATA-AND-BLOB）モードのときは、損傷したデータベースを再構築する種々の選択肢があります。

リストアの準備をする

データベースをリストアする前に、以下の問いを確認する必要があります：

- どの時点のデータベースにリストアしますか？

ディスク障害が発生した時点のデータベースにリストアするのであれば、全てのジャーナルファイルを可能な限りバックアップします。これらのフ

ファイルは、現時点に最も近いデータベースにリストアするのに役に立ちます。

- 以前にどのファイルのバックアップを取りましたか？

最新の完全バックアップとそれ以降の差分と増分バックアップが何処にあるかを見つけます。例えば、毎月 30 日に完全バックアップを取り、毎月 15 日に差分バックアップを取り、10 日毎に増分バックアップを取るとします。5 月 25 日にシステムが損傷したならば、4 月 30 日の完全バックアップと、5 月 15 日の差分バックアップと、5 月 20 日の増分バックアップと、5 月 25 日の損傷ジャーナルファイルが必要になります。これらのファイルを見つけると、DBMaster は、5 月 25 日の障害の前の状態にデータベースをリストアすることができます。完全バックアップファイルと連なった差分ファイルと増分ファイルで構成された正確なバックアップ順はリストアの必要不可欠です。オンラインバックアップ順は dmbackup.his というファイル名のバックアップ履歴に記録され、オフラインバックアップ順は offbackup.his というファイル名のバックアップ履歴に記録され、このバックアップ履歴ファイルは非常に重要です。DBMaster は、データベースをリストアするときに、その情報を読み込みます。

リストアする

リストアプロセス時、DBMaster は以下の動作を行います：

- データファイル、BLOBファイル、ジャーナルファイルを含む完全バックアップファイルを**dmconfig.ini**ファイルのDB_DBDIRで指定されたディレクトリにコピー。このオペレーションは元のファイルを上書きします。そのためリストア実行前に元のデータベースファイルを別箇所へのコピーにて退避しておくことが望ましいです。少なくともジャーナルファイルが保存されていることを確認し、リストアに失敗した場合でもデータベースを現状に近い状態まで戻す可能性を残せます。
- 差分と或いは増分バックアップファイルをデータベースに適用
- リストアツール使用時の指定可能設定。
- **dmconfig.ini**のデータベースセクションをリストアするかの指定。リストアする場合、リストアされる**dmconfig.ini**のフルパスを指定。

- ユーザーはバックアップ順で普通の位置と違った位置にデータベースをリストアしたい場合、データファイル"path"、例えばDB_DBDIR、DB_DBFIL、 DB_USRDBなどに、或いはデータファイルとblobファイルのパスに再びキーワードを設定するべきです。もしバックアップ順は他の場所または新しいコンピュータに移動されると、データベースをリストアする場合、ユーザーは以下の状況をお考えください：

  a) 構成セクションは新規なデータベースの**dmconfig.ini**ファイルにあると、データベース名はバックアップデータベース名と一致にしなければなりません。

  b) 新規なキーワード（BKDIR或いは他のデータファイルとblobファイルのキーワード）を設置します。

  c) 一つ以上のjnlファイルがあると、ユーザーは**DB_JNFIL**の値をセットして、jnlファイルのナンバーはバックアップデータベースのjnlファイルと同じでなければなりません。

  d) バックアップファイルは複数のフォルダーにあると、ユーザーはキーワード**DB_BKDIR**（全てのバックアップファイルのフォルダーを含み、dmbackup.hisファイルは始めのBKDIRだとのことを確保する）を設置しなければなりません。

  注 *ユーザーはバックアップ順があるフォルダーから既存のdmconfig.iniファイルをコピーすることができます。関連キーワードを変更することで、新規のdmconfig.iniファイルも設置できます。*

- バックアップ履歴ファイルdmBackup.his或いは offBackup.hisのオンラインまたはオフラインのフルパス。dmBackup.his 或いはoffBackup.hisがデフォルトのディレクトリに置かれていない場合に指定。

- リストアタイム(RTime) RTimeはデータベースがリストアされる時間を表示し、バックアップ順が適切であるか判断し、差分と増分バックアップファイルをデータベースに適用します。リストアツールまたはシステム**dmconfig.ini**のキーワード**DB_RTime**にて設定するかリストアされるバックアップ**dmconfig.ini**に指定します。RTimeが定義されないとデフォルトの値として現在の時刻が使用されます。

DBMaster はリストアに対して JServer Manager Tool とコマンドラインツール上での Rollover による 2 種類の方法を用意しています。

JServer Manager の使用については*「JServer Manager ユーザガイド」*をご覧ください。ロールオーバーコマンドラインツールに関しては次の *Rollover によるデータベースリストア*をご覧ください。

ROLLOVERによるデータベースリストア

Rollover コマンドラインツールをデータベースのリストアに使用できます。方式は JServer Manager によるデータベースのリストアのものと同じです。

Rollover の使用方法は下のようになります：

rollover database_name [-i inifile] [-r rtime] [-h hisfile] [-m foMapfile] [-f FOtype] [-t rsSP]

六つの鉤括弧のオプション引数があります：

-i **dmconfig.ini** のフルパスを指定。**dmconfig.ini** のリストアを指定した場合、rollover は **dmconfig.ini** の指定に対応したシステム **dmconfig.ini** のデータベースセクションを置き換えます。そうでなければ **dmconfig.ini** はリストアされません。

-r データベースがリストアされる時間を示します。-r オプションは RTime を設定する一つ目の方法で、キーワード **DB_RTime** をシステム **dmconfig.ini** またはリストアデータベースを設定するバックアップ **dmconfig.ini** に定義する方法があります。-r オプションとキーワード **DB_RTime** のどちらも定義されていない場合は現在の時刻が RTIME として使用されます。

-h dmBackup.his 或いは offBackup.his の完全パスを指定する。初期値は "DB_BkDir/dmBackup.his"又は"DB_BkDir/offBackup.his"です。

-m dmFoMap.his の完全パスを指定する。初期値は *"DB_BkDir/FO/dmFoMap.his"*です。

-f FO ファイルのユーザ/システムでどちらをリストアするかを定義します。このオプションには 4 つの設定値があります。0 は FO ファイルをリストアしない;1 はシステム FO をリストア;2 はユーザ FO を、3 は全ての FO をリストアします。デフォルト値は ***3*** です。

-t　ストアドプロシージャをリストアするかを指定します。このオプションには二つの設定値があり、0 の場合は、リストアしたストアドプロシージャがないと表し、1 の場合は、あらゆるストアドプロシージャがリストアされると表します。当該設定値のデフォルト値は ***1*** です。

16　分散データベース

この章は、分散データベース、DBMasterの分散アーキテクチャ、分散データアクセス、分散データベースオブジェクト管理、分散トランザクション管理など、DBMasterの分散データベース管理機能を紹介します。現在、DBMasterではDDBモードでのサブクエリをサポートしておりません。

16.1　分散データベースの概要

典型的なクライアント/サーバー・データベース管理システムは、図16-1に示すようにネットワーク上の特定のコンピュータにデータベースを置き、このコンピュータが全てのクライアントの要求を処理する役割をもちます。

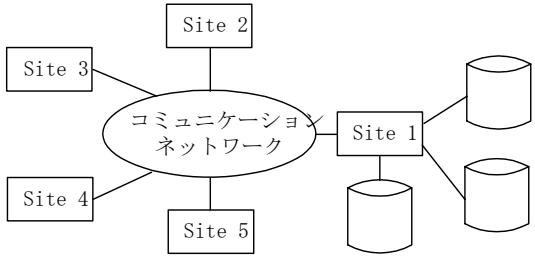


図 16-1：従来のクライアント/サーバー・データベース管理システム

分散データベースは、図15-2 に示すようにネットワーク上の複数コンピュータにデータベースのコピーを置き、各コンピュータが独立にデータベースクライアントをサポートします。分散データベース管理システムは、各

コンピュータのデータベースを管理し、データが物理的に何処にあるかを関知することなく透過的にデータをアクセスできるようにします。

DBMaster は、リモートデータベース接続、分散問い合せ、分散トランザクション管理などの機能を提供し、真の分散アーキテクチャをサポートする完全で強固な分散データベース管理システム（分散 DBMS）です。また、表とデータベースをレプリケーションし、自動的にデータを最新状態に保ちます。

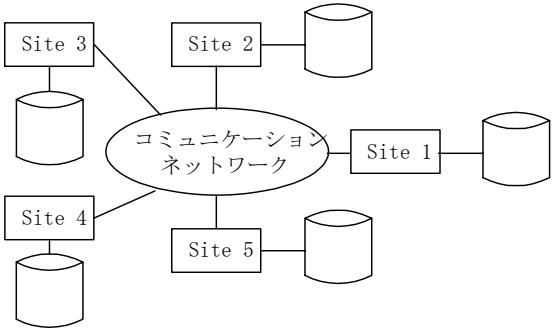


図 16-2：クライアント・サーバーに分散したデータベース

DBMaster の分散データベース環境では、ODBC 3.0 互換 API と分散データベースの異なる部分をアクセスする特別な SQL 問い合せを使用してアプリケーションプログラムを書くことができます。DBMaster は、全てのデータがローカルデータベースにあるかのように透過的にデータを統合し、結果を返します。

この章では、DBMaster のシステムアーキテクチャと分散データベース管理の基本機能を概観します。分散環境の構成、リモートデータリンクと分散トランザクション管理、分散問い合せの実行などを取り上げます。データベース管理者であるかアプリケーション開発者であるかに関わらず、DBMaster の分散アーキテクチャの簡明さと機能を概観できるようにします。

16.2 分散型データベース構造

DBMaster の分散データベース環境は、典型的なクライアント／サーバー・アーキテクチャの上に、マルチクライアント・アプリケーションとマルチデータベースサーバーを効率よくリンクして構築します。クライアント・アプリケーションはユーザー要求を処理して結果を表示し、データベースサーバーはデータを管理します。各クライアントは、コーディネータ・データベースと言われる単一のデータベースサーバーに接続し、コーディネータ・データベースを通して、参加データベースと呼ばれる他のリモート・データベースに接続することができます。

DBMaster は、階層的分散構造を使用してリモート・データベースに接続します。コーディネータ・データベースと参加データベースは階層的に分散し、参加データベースを通して、直接接続していないリモートデータベースのデータへもアクセス可能にします。この場合、参加データベースは、下の階層にある子データベースのコーディネータ・データベースとしての働きをし、ローカル・コーディネータデータベースと言われます。DBMaster の分散構造を図15-3 に例示します。

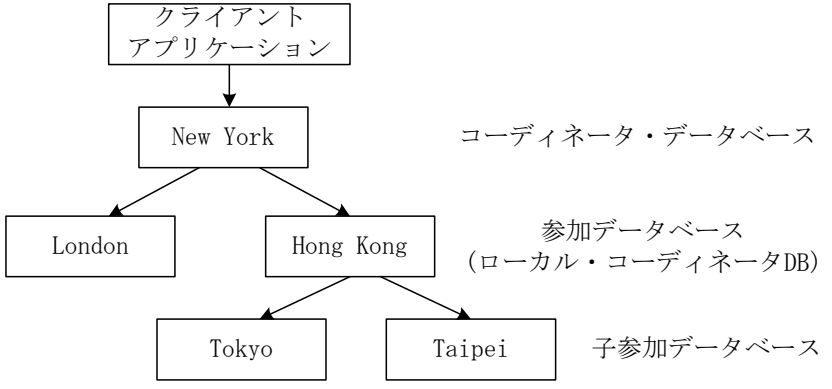


図 16-3 : DBMaster の分散構造

図 16-3 のクライアント AP は、New York データベースサーバーをコーディネータ・データベースにし、New York データベースサーバーに接続します。New York データベースを通して London と Hong Kong のデータにアクセスし、

London と Hong Kong のデータベースが参加データベースになります。Hong Kong のデータの一部が Tokyo または Taipei にある場合、Hong Kong データベースは Tokyo と Taipei の調整データベースになり、Tokyo と Taipei のデータベースは子参加データベースになります。Hong Kong データベースは、参加データベースであるだけでなく、ローカル・コーディネータ・データベースにもなります。

16.3 分散データベース環境

DBMaster の分散データベース環境のセットアップは非常に簡単です。必要なのは、**dmconfig.ini** ファイルにキーワードを幾つか追加し、分散データベース構成オプションを設定することだけです。これらのパラメータは、JConfiguration Tool を使ってセットすることもできます。詳細については、*「JConfiguration Tool ユーザーガイド」*を参照して下さい。

DBMaster の分散データベース環境は、以下に説明するキーワードの値を設定してセットアップします。DB_で始まるキーワードは、クライアントとコーディネータ・データベース間のクライアント/サーバー接続用です。DD_で始まるキーワードは、コーディネータ・データベースと参加データベース間の分散データベース接続用です。

- **DB_SvAdr=<IPアドレス/ホスト名>**—コーディネータ・データベー スのIPアドレスまたはホスト名を指定します。クライアント・アプリケーションが接続するコーディネータ・データベースの位置を知らせるために使用されます。
- **DB_PtNum=<ポート番号>**—クライアント・アプリケーションとコーディネータ・データベースが通信するために使用するポート番号です。
- **DD_DDBMd=<0/1>**—分散データベース・モードを有効/無効にします。初期値は0で、分散データベース・モードが無効であることを意味します。
- **DD_CTimo=<秒数>**—コーディネータ・データベースが参加データベースに接続する際の待ち時間を秒数で指定します。初期値は5秒です。

- DD_LTimo=<秒数>—コーディネータ・データベースが参加データベースのデータをロックする際の待ち時間を秒数で指定します。初期値は5秒です。
- DD_GTSvr=<0/1>—グローバル・トランザクション・リカバリ（GTRECO）デーモンを有効/無効にします。初期値は1で、有効を意味します。
- DD_GTItv=<D-hh:mm:ss>—中断グローバル・トランザクションを処理するときのグローバル・トランザクション・リカバリ（GTRECO）デーモンの待ち時間間隔を指定します。

DBMaster は、ネットワーク上または参加データベース上のエラーに起因する分散トランザクション障害の自動リカバリ機構をサポートします。自動リカバリ機構は、GTRECO デーモンによって処理されます。GTRECO デーモンは、分散データベースサーバがグローバルトランザクションに対して何か問題があるかどうかをチェックします。問題を検出すると、GTRECO デーモンは、中断グローバルトランザクションをリカバリしようとします。GTRECO デーモンは、dmconfig.ini ファイルの DD_GTSVR キーワードを使用して有効にします。

➲ 例

ロスアンゼルス支店とシアトル支店をもつ ABC 銀行を例にして、DBMaster がどのように分散データベースを管理するかを良く理解できるように説明します。各支店には、店独自の顧客データとビジネスデータがあります。ただし、行政財務データベースは、ロスアンゼルス支店で中央管理します。

ロスアンゼルス支店データベースサーバーの dmconfig.ini ファイル：

```
[BankTranx]                              ;ロスアンゼルス支店ビジネスデータベース
DB DbDir = c:\database
DB SvAdr = 192.168.0.1
DB PtNum = 21000
DD DDBMD = 1

[BankMIS]                                ;行政財務データベース
DB DbDir = c:\database
DB_SvAdr = 192.168.0.1
DB_PtNum = 30000
```

```
DD DDBMD = 1

[BankTranx@Seattle]                    ;シアトル支店ビジネスデータベース
DB SvAdr = 192.168.0.2
DB_PtNum = 21000
DD_CTIMO = 20
DD_LTIMO = 10
```

シアトル支店データベースサーバーの **dmconfig.ini** ファイル：

```
[BankTranx]                            ;シアトル支店ビジネスデータベース
DB_DbDir = c:\database
DB_SvAdr = 192.168.0.2
DB_PtNum = 21000
DD_DDBMD = 1

[BankMIS]                              ;ロスアンゼルス支店行政財務データベース
DB SvAdr = 192.168.0.1
DB PtNum = 30000
DD CTIMO = 20

[BankTranx@La]                         ;ロスアンゼルス支店ビジネスデータベース
DB SvAdr = 192.168.0.1
DB PtNum = 21000
DD CTIMO = 20
DD_LTIMO = 10
```

ロスアンゼルス支店クライアントの **dmconfig.ini** ファイル：

```
[BankTranx]                            ;ロスアンゼルス支店ビジネスデータベース
DB SvAdr = 192.168.0.1
DB_PtNum = 21000
```

シアトル支店クライアントの **dmconfig.ini** ファイル：

```
[BankTranx]                            ;シアトル支店ビジネスデータベース
DB_SvAdr = 192.168.0.2
DB_PtNum = 21000
```

分散データベースをサポートするには、上記の環境設定ファイルのローカルデータベース構成セクションに **DD_DDBMD**=1 を設定します。この例では、ロスアンゼルス支店とシアトル支店の **dmconfig.ini** ファイルの BankTranx 構成セクションにこのキーワードを設定します。

コーディネータ・データベースの環境設定ファイルには、参加データベースのデータベース構成セクションも加えます。また、参加データベースの環境設定ファイルには、コーディネータ・データベースのデータベース構成セクションを含めます。この例では、ロスアンゼルス支店データベースとシアトル支店データベースに、同じデータベース名[BankTranx]が使用されています。リモートデータベースのデータベース構成セクション名が、**dmconfig.ini** ファイルにあるローカルデータベースのデータベース構成セクション名と競合する場合があります。

分散データベースのこの種の問題を避けるために、ローカル **dmconfig.ini** ファイルにあるリモートデータベース名に、サーバーホスト記述を付加することによってローカルデータベース名と区別できるようにしています。リモート・データベース名は、次の形式になります：

```
データベース名@サーバーホスト記述
```

サーバーホスト記述は任意の識別文字列です。例えば、IP アドレス、データベースサーバのホスト名、ドメイン名、説明文等をサーバーホスト記述に使用することができます。この例では、ロスアンゼルス支店のクライアント・アプリケーションからシアトル支店のデータベースにアクセスするときは BankTranx@Seattle を使用し、シアトル支店のクライアント・アプリケーションからロスアンゼルス支店のデータベースをアクセスするときは BankTranx@La を使用します。

ロスアンゼルス支店とシアトル支店の環境設定ファイルには、ローカルデータベースとリモートデータベースのサーバーアドレスとポート番号をそれぞれの構成セクションに指定します。

この例では、ロスアンゼルス支店の環境設定ファイルは、BankTranx 構成セクションにローカルサーバーアドレスを指定し、BankTranx@Seattle 構成セクションにシアトル支店のサーバーアドレスを指定します。同様に、シアトル支店の環境設定ファイルは、BankTranx 構成セクションにシアトル支店のサーバーアドレスを指定し、BankTranx@La 構成セクションにロスアンゼルス支店のサーバーアドレスを指定します。

DD_CTimo と **DD_LTimo** のリモート接続パラメータは、コーディネータ・データベースの環境設定ファイルでは参加データベースの構成セクション

に、参加データベースの環境設定ファイルではコーディネータ・データベースの構成セクションに指定します。

ネットワーク上の各データベースサーバーは、分散データベースのオブジェクトを操作することができます。ユーザーは、通常のクライアント／サーバー・アーキテクチャと同様の方法で、コーディネータ・データベースを通して任意のデータベースサーバーにアクセスすることができます。リモートデータベースに問合せる SQL 文は、コーディネータ・データベースを通してリモートデータベースサーバーに渡されます。コーディネータ・データベースは、SQL 文をローカル部分とリモート部分に分解し、適切な SQL 文をリモートデータベースサーバーに送ります。コーディネータ・データベースは、リモートデータベースが結果を返すのを待ち、ローカルとリモートの全てのデータを統合し組み合わせた結果をユーザーに返します。

16.4 分散データベースのオブジェクト

DBMaster には、参加データベースにアクセスする方法がいくつかあります。

- 参加データベース名を直接指定する。
- コーディネータ・データベースに定義されているデータベースリンクを使用する。
- ビューやシノニムのようなリモートオブジェクトマッピングを通す。

最初の 2 つの方法の違いは、データベースリンクがリモートデータベース名の他にセキュリティ情報を伴う点です。データベースリンクは、リモートデータベースにアクセスするときに、ユーザー名とパスワードを指定させることができます。

分散式クエリと通常クエリの違いは、データベース・オブジェクトを指定する異なる構文のみです。但し、アクセスできるリモートデータベースのオブジェクトは、表、ビュー、シノニム及びストアドコマンドのみです。リモートデータベースのオブジェクトにアクセスする場合、リモートデータベース名かデータベースリンクを指定します。リモートデータベースのオブジェクトは、次の２通りの方法で指定することができます：

- リモートデータベース名：オブジェクト所有者.オブジェクト名

- データベースリンク名：オブジェクト所有者.オブジェクト名

➲ **例 1**

リモート・データベースのオブジェクト問合せる：

```
dmSQL> SELECT * FROM Bank:EmpTaple;
dmSQL> DELETE FROM Bank:EmpTable WHERE id = 101;
dmSQL> INSERT INTO Link1:mis.account VALUES (2003,'Kevin Liu','2327-0021');
```

➲ **例 2**

2 つの参加データベースにあるリモート・データベースのオブジェクトへアクセスする：

```
dmSQL> SELECT * FROM ABCBank@La:account a,
                     ABCBankMIS@Seattle:account b
                WHERE a.name = b.name;
```

➲ **例 3**

リモートデータベース DB1 に cmd 1 というストアドコマンドを作成して、データベース DB2 にてそのコマンドを実行する：

```
dmSQL> EXECUTE COMMAND DB1:cmd1(value);
```

データベース名でリモートデータベースに接続する

コーディネータ・データベースサーバーにあるデータベース名を通してリモートデータベースに接続することができます。この方法でデータにアクセスするには、コーディネータ・データベースサーバーの **dmconfig.ini** ファイルに定義されているリモートデータベース名を知る必要があります。

➲ **例 1**

ABC 銀行のロスアンゼルス支店のクライアント・アプリケーションから台北支店データベースにアクセスします。この例は、ユーザー名 SYSADM とパスワード aa で台北支店に接続しているように見えますが、実際は、ロスアンゼルス支店のコーディネータ・データベースに接続しています。コーディネータ・データベースは、接続するときに与えたユーザー名とパスワードを用いてリモートデータベースに接続します。

```
dmSQL> CONNECT TO BankTranx SYSADM aa;
```

```
dmSQL> SELECT * FROM BankTranx@Taipei:SYSADM.Account ORDER BY AccID;
```

➲ **例 2**

ジョインを使って、２つのリモート・データベース・オブジェクトにアクセスする：

```
dmSQL> SELECT * FROM BankMIS:SYSADM.Personnel ORDER BY PID;
dmSQL> SELECT Personnel.* FROM BankTranx@Taipei:Account A,
                               BankMIS:Personnel B
                          WHERE A.CustID = B.CustID;
```

データベースリンクでリモートデータベースに接続する

データベースリンクは、リモートデータベース接続に必要なログイン情報とパスワードを含むリモートデータベース接続を作成します。データベースリンクを使用することによって、コーディネータ・データベースのユーザー名とは別のユーザー名を使用してリモートデータベースに接続したり、パブリックリンクを使用してアカウントの無いリモートデータベースに接続したりすることができるようになります。また、データベースリンクは、分散データベース環境のデータを透過的にします。ログイン情報とパスワードを含むデータベースリンク定義の情報は、コーディネータ・データベースに格納されます。

データベースリンクを作成する

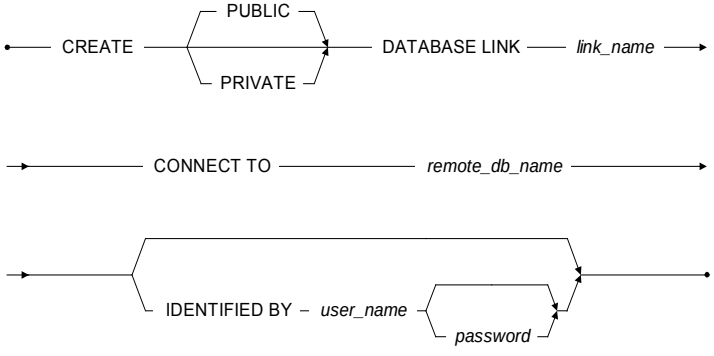


図 16-4：CREATE DATABASE 文の構文

パブリックリンクは、データベースの全ユーザーが使用することができ、SYSADM と DBA ユーザーのみが作成することができます。プライベートリンクは、作成ユーザーのみが使用することができます。パブリックリンクと同じ名前のプライベートリンクを作成した場合、プライベートリンクがパブリックリンクを上書きします。

リンクの種類を指定しない場合、初期設定のプライベートリンクが作成されます。IDENTIFY BY 句でユーザー名とパスワードを指定しない場合、作成者のユーザー名とパスワードが初期設定で使用されます。

リモート・データベース・オブジェクトとデータベース・リンク

➲ 例 1

どのようにデータベースリンクを使用してリモートデータベースオブジェクトをアクセスするかを以下に例示します。SYSADM は、データベースに接続してパブリックリンク Bank_Seattle を作成します。Bank_Seattle は、ユーザー名 SYSADM でシアトル支店データベースに接続します。SYSADM は、Bank_Seattle にある Account 表を更新して切断します。次に user1 が接続して Account 表の問合せを実行します：

```
dmSQL> CONNECT TO BankTranx SYSADM;
dmSQL> CREATE PUBLIC DATABASE LINK Bank_Seattle CONNECT TO BankTranx@Seattle
    2> IDENTIFIED BY SYSADM;
dmSQL> UPDATE Bank_Seattle:Account SET balance = balance + 100
    2> WHERE id = 1001;
dmSQL> DISCONNECT;
dmSQL> CONNECT TO BankTranx user1 pwd1;
dmSQL> SELECT * FROM Bank_Seattle:Account;
```

➲ 例 2

SYSADM はパブリックリンクで接続するときのユーザ名を指定していません。このため、user1 がパブリックリンクを使用して Bank_Seattle データベースに接続するには、リモートデータベースに user1 アカウントがあり、user1

は SYSADM.Account 表に問合せる権限をもっていなければなりません。そうでなければ、この例はエラーを引き起こします：

```
dmSQL> CONNECT TO BankTranx SYSADM;
dmSQL> CREATE PUBLIC DATABASE LINK Bank_Seattle CONNECT TO BankTranx@Seattle;
dmSQL> SELECT * FROM Bank_Seattle:Account;
dmSQL> DISCONNECT;
dmSQL> CONNECT TO BankTranx user1 pwd1;
dmSQL> SELECT * FROM Bank_Seattle:SYSADM.Account;
```

データベースリンク名とリモートデータベース名が同じ名前のときは、データベースリンク名が優先します。リモートデータベースを直接アクセスしたいときは、データベース名@"" 形式で明示的にリモートデータベース名であることを指定します。DBMaster は、データベースリンクの代わりに直接リモートデータベースをアクセスします。

リモートデータベースをアクセスする２通りの方法を例示します。一つはデータベースリンクを使用し、他の一つはデータベース名@""の形式で明示的にデータベース名を指定します。

➲ **例 3**

SYSADM がリンクを使ってリモート・データベースに接続する：

```
dmSQL> CONNECT TO BankTranx SYSADM;
dmSQL> CREATE PUBLIC DATABASE LINK BankMIS CONNECT TO BankMIS
    2> IDENTIFIED BY SYSADM;
dmSQL> DISCONNECT;
```

➲ **例 4**

user1 が BankMIS@"":SYSADM.Personnel フォームを使って、リモート・データベースに接続する：

```
dmSQL> CONNECT TO BankTranx user1 pwd1;
dmSQL> SELECT * FROM BankMIS:Personnel;           //データベースリンクを使用
dmSQL> SELECT * FROM BankMIS@"":SYSADM.Personnel; //リモートデータベース名を使用
```

注 *リモートデータベースオブジェクトにアクセスするときデータベースリンクを使用しUPDATEやDELETE操作を行いますがこのときサブクエリを使用することはできません。現在DBMasterでは未対応の動作です。*

データベースリンクを削除する

図 16-5 ：DROP DATABASE LINK 構文

パブリックリンクは、SYSADM と DBA ユーザーのみ削除することができます。プライベートリンクは、所有者のみ削除することができます。同じ名前のパブリックリンクとプライベートリンクがあるときに、リンクの種類を指定せずにデータベースリンクを削除しようとすると、プライベートリンクが削除されます。

➲ **例**

パブリック・データベースリンク BankMIS を削除する：

```
dmSQL> DROP PUBLIC DATABASE LINK BankMIS;
```

データベース・オブジェクトのマッピング

データベース・オブジェクトのマッピングは、分散データベース環境を更に位置透過的にします。データベース・オブジェクトのマッピングでリモート・データベース・オブジェクトをアクセスする方法は、ローカル・データベース・オブジェクトにアクセスする方法と同じです。データベース・オブジェクトのマッピングには、ビューあるいはシノニムを使用します。

シノニム

リモート・データベースのオブジェクトのシノニム定義は、リモート・データベースのオブジェクトに別名を付けることです。リモート・データベースのオブジェクトは、シノニム作成者の権限ではなく、接続ユーザーの権限でアクセスします。

➲ 例

シノニムを使って、リモート・データベースのオブジェクトにアクセスする：

```
dmSQL> CONNECT TO BankTranx user1;
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user2;
dmSQL> CREATE SYNONYM s1 FOR BankTranx:Account;
dmSQL> CREATE SYNONYM s2 FOR LK1:user2.Personnel;
dmSQL> SELECT * FROM s1;
     // SELECT * FROM BankTranx:user1.Account; (BankTranx, user1)
dmSQL> SELECT * FROM s2;
     // SELECT * FROM LK1:user2.Personnel; (BankMIS, user2)
dmSQL> DISCONNECT;
dmSQL> CONNECT TO BankTranx user3;
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user4;
dmSQL> SELECT * FROM s1;
       // SELECT * FROM BankTranx:user3.Account; (BankTranx, user3)
dmSQL> SELECT * FROM s2;
       // SELECT * FROM LK1:user2.Personnel; (BankMIS, user4)
```

コメント行は同等の SQL 文、接続データベースと接続ユーザー名を記しています。

ビュー

リモート・データベースのオブジェクトのビュー定義は、シノニムとは異なります。ビューは単なる別名ではなく、ビューの定義には、データベース名、ユーザー名、パスワード、オブジェクト所有者、オブジェクト名を含めることができます。リモート・データベースは、接続ユーザーの権限ではなく、ビュー作成者の権限でアクセスします。

➲ 例

ビューを使ってリモート・データベースのオブジェクトにアクセスする：

```
dmSQL> CONNECT TO BankTranx user1;
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user2;
dmSQL> CREATE VIEW v1 AS SELECT * FROM BankTranx:Account;
dmSQL> CREATE VIEW v2 AS SELECT * FROM LK1:user3.Personnel;
dmSQL> SELECT * FROM v1;
       // SELECT * FROM BankTranx:user1.Account; (BankTranx, user1)
```

```
dmSQL> SELECT * FROM v2;
       // SELECT * FROM BankMIS:user3.Personnel; (BankMIS, user2)
dmSQL> DISCONNECT;
dmSQL> CONNECT TO BankTranx user3;
dmSQL> CREATE DATABASE LINK LK1 CONNECT TO BankMIS IDENTIFIED BY user4;
dmSQL> SELECT * FROM v1;
       // SELECT * FROM BankTranx:user1.Account; (BankTranx, user1)
dmSQL> SELECT * FROM v2;
       // SELECT * FROM LK1:user3.Personnel; (BankMIS, user2)
```

コメント行は同等の SQL 文、接続データベースと接続ユーザー名を記しています。

データベースリンクをクローズする

SQL 文でリモートデータベースに一度でもアクセスすると、コーディネータ・データベースは、参加データベースへのリモート接続を確立します。リモート接続は、全てのユーザーがコーディネータ・データベースから切断するか、CLOSE DATABASE LINK 文で明示的にリンクを閉じるまでオープンしています。1 つのデータベース当たり 256 つまでしかリモート接続を確立することができないので、必要の無くなったリモート接続はクローズして開放します。

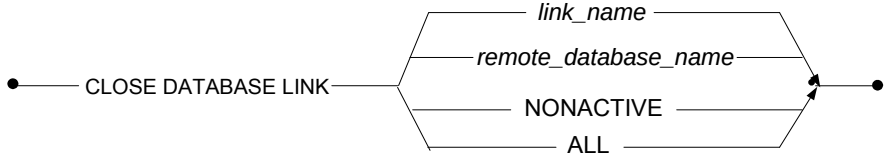


図 16-6 : CLOSE DATABSE LINK 構文

➲ **例 1**

リモートデータベース名 BankMIS を使ってデータベースリンクをクローズする：

```
dmSQL> CLOSE DATABASE LINK BankMIS;
```

➲ 例 2

データベース・リンク名 BankLink1 を使ってデータベース・リンクをクローズする：

```
dmSQL> CLOSE DATABASE LINK BankLink1;
```

CLOSE DATABASE LINK 文を実行すると、リモート接続カウンタが 1 減ります。カウンタが 0 になると、リモート接続は完全に切断され、リソースが開放されます。0 でなるまでは、リモート接続はオープンしたままです。

➲ 例 3

全データベース・リンクをクローズし、接続とリソースを開放する：

```
dmSQL> CLOSE DATABASE LINK ALL;
```

➲ 例 4

使用していない全リモート接続、NONACTIVE をクローズする：

```
dmSQL> CLOSE DATABASE LINK NONACTIVE;
```

データベースリンクのシステムカタログ表

データベースリンクに関連するシステムカタログ表は、SYSDBLINK と SYSOPENLINK があります。SYSDBLINK は全てのデータベースリンク名と定義を記録し、SYSOPENLINK はデータベース間でオープンしている接続を記録します。

16.5 分散トランザクション管理

DBMaster では、ユーザーに透過的な分散トランザクションのメカニズムを使用することができます。参加データベースでは、一切の分散トランザクションをコミットしません。

➲ 例

分散トランザクション制御がどのように行われるかを例示します：

```
dmSQL> CONNECT TO BankTranx user1;                    // ロスアンゼルスの ABC 銀行
dmSQL> SET AUTOCOMMIT OFF;
dmSQL> UPDATE BankTranx:Customer SET money=money-1000 where id=123;
```

```
dmSQL> UPDATE BankTranx@"Bank in Seattle":Customer SET money=money+1000
    2> WHERE id=123;
dmSQL> COMMIT;
```

コーディネータ・データベースがクライアント・アプリケーションによる全データベース操作を扱うので、コーディネータ・データベースは、分散システムカタログマネージャを通して命令の概要を理解します。ローカルサイトに属するトランザクションは、コーディネータ・データベースによって通常のクライアント/サーバー・トランザクションと同様の方法で処理されます。リモートサイトに属するトランザクションは、適切なリモートデータベースを参照している必要があります。コーディネータ・データベースは、各参加データベースと情報を交換し、ロールバックまたはコミットされるまで、トランザクション全体を調整します。

2フェーズコミット

データベース管理システムは、データ整合性を保つためにトランザクションを一体で管理します。トランザクションの全ての操作は、一体でコミット、或いはロールバックされます。従来のクライアント/サーバー・アーキテクチャでは、ジャーナルを使用して種々の変更を確実にロールバック、又はコミットします。

分散データベースアーキテクチャでは、仮アボート付き 2 フェーズコミットプロトコルを使用して、複数データベースサーバに分散するトランザクションを管理します。トランザクションが複数データベースのデータを修正するときは、コミットする前 2 フェーズコミットプロトコルを完了しなければなりません。2 フェーズコミット機構は、全てのサイトのコミットまたはロールバックをグローバルに保証します。また、リモートのシノニム、整合性制約、トリガーが実行したデータ操作オペレーションも保護します。トランザクションをコミットするには、全てのサブトランザクションが終了したことを確かめなければなりません。そうでないときは、トランザクションはアボートされます。同じ理由で、コミットできないサブトランザクションがあるときは、他のサブトランザクションも同じようにアボートします。

分散トランザクションリカバリ

全ての参加データベースにグローバルトランザクションのコミットを通知するときは、2 フェーズコミットプロトコルが使用されます。コーディネータ・データベースは、コミットフェーズに入る前に参加データベースのステータスをチェックし、サーバおよびネットワーク上に問題が無いことを確かめます。参加データベースに問題があるときは、他の参加データベースに該当するトランザクション部分をロールバックする通知を出し、グローバルトランザクション失敗のエラーを返します。参加データベースサーバまたはネットワーク上に何か問題があっても、２フェーズコミットプロトコルの準備フェーズが終了しているならば、グローバルトランザクションは成功とみなされます。DBMaster は、どの参加データベースサーバが該当するトランザクション部分をコミットできないかを SYSGLBTRANX システムカタログ表に記録します。また、クラッシュしたデータベースに対する中断トランザクションをもっているデータベースを SYSPENDTRANX システムカタログ表に記録します。

DBMaster には、分散トランザクション実行中のネットワーク障害あるいはサイト障害を処理する自動リカバリ機構があります。コーディネータ・データベースは、グローバルトランザクションリカバリ（GTRECO）デーモンを起動させます。このデーモンは、SYSGLBTRANX システムカタログ表を検索し、中断グローバルトランザクションを周期的にリカバリします。次に、クラッシュした参加データベースに接続し、グローバルトランザクションの該当部分をコミットまたはロールバックすることを通知します。

発見的グローバルトランザクション終了

2 フェーズコミット中にネットワーク障害またはサイト障害が発生すると、中断トランザクションは、ロックやジャーナルブロック等のリソースを保持したままにします。中断トランザクションは、グローバルリカバリ（GTRECO）デーモンが問題を解決するまで、これらのリソースを占有します。直ちにネットワークまたはサイトの障害を解決することができない場合、参加データベースユーザの一部は、占有されたリソースによってブロックされます。この問題を解決するために、DBMaster は発見的グローバルトランザクション終了をサポートします。発見的トランザクション終了

は、データベース管理者が行う独立したアクションであり、参加データベースの中断トランザクションを強制的にコミットまたはロールバックします。データベース管理者は、JDBATool を使用してこの問題を解決することができます。詳細については、*「JDBA Tool ユーザーガイド」*を参照して下さい。

➲ **中断トランザクションを手動で解決する：**

1. 参加データベースのSYSPENDTRANX表を見て、長時間中断しているトランザクションがあるかどうか調べます。
2. コーディネータ・データベースで、SYSGLBTRANXシステム表から中断トランザクションのコミット状態を調べます。例えば、"DB_1-3376aafd"と"DB_2-3376aafd:DB_3-3376ab0f#1"の中断トランザクションがあるとします。DB_1管理者に"DB_1-3376aafd"の状態を問い合せ、DB_3管理者に"DB_2-3376aafd:DB_3-3376ab0f#1"の状態を問い合せます。
3. 問い合せを受けた管理者は、SYSGLBTRANXを調べてトランザクションの状態を判断します。STATE 2（COMMIT）、又は3（PENDCOM）ならば'commit'と回答します。STATE 4（PENDABO）の場合は、'abort'と回答します。しかし、STATE 1（PREPARE）ならば、このトランザクション分岐はこのサイトで中断中であり、親サイトの管理者に状態の判断を委ねます。
4. 参加データベース管理者は、回答に基づきJServer Managerを使用して、発見的コミットまたはアボートを実行します。

調整データベースで行われたアクションと異なる発見的中断トランザクション終了を行うと、分散データは不整合になります。

17 データ・レプリケーション

データ・レプリケーションは、広義には複数のデータベースのオブジェクトに適応する処理を指します。

わずか数年前には、企業のデータはその本部に集中していました。情報を利用する遠隔地の部署は、本部に直接アクセスする接続を設けるか、本部の情報システム部門に印刷された書類を依頼する必要がありました。その接続は、高価で信頼性に乏しい上に数に限りがありました。一方書類は、柔軟性に欠け、多くの時間がかかりました。

オープン・システムは、安価で強力なコンピューティング・リソースをあらゆる企業にもたらしました。これらの新しいリソースを使って企業の情報を効率的に共有する能力は、企業集団にとって重要な競争力の強みになりました。今日企業が直面している問題は、「何故、企業データを分散させて、共有するのか？」では無く、むしろ「どのように効率的に情報を分散させるか？」ということです。レプリケーションは、分散された企業アプリケーションの多くで採用されるアーキテクチャとなってきています。

17.1 表レプリケーション

表レプリケーションとは

表レプリケーションは、リモート・サイトに表の全体、又は部分的なコピーを作成します。これにより、ユーザーが遠隔地でデータのコピーを使って作業することが可能です。使用するコピーは、別の場所のデータベースと同調されています。この方法では、各データベースは、遅いネットワーク接続上で他の機器にアクセスする必要がないので、データ要求にすぐ応

えることができ、より効率的です。このような要求は、本社と地域企業又は支店間で、頻繁に発生します。

例えば、台北のデータベース・サーバーの表 A から、東京のデータベース・サーバーにある表 B にレプリケーションを設けた後、表 A に加えられた修正は、表 B にレプリケートされます。東京のクライアントは、台北のデータベースに接続することなく、東京のデータベースにアクセスして同じデータを取得することができます。

データベース・レプリケーションと表レプリケーションの違い

データベース・レプリケーションと表レプリケーションの大きな違いは、データ・オブジェクトが異なる点です。前者は完全なデータベースで、後者は表です。ユーザーは、必要に応じてそれらを使い分けます。データベース・レプリケーションを選択すると、レプリケートする単位が全データベースなので、ターゲット・データベースは読み込み専用になります。

2種類の表レプリケーション

表レプリケーションには、2 種類あります。 1 つは、同期です。'同期'は、修正部分が即リモート・データベースに反映されることを意味します。元の表の修正と同時にリモート表も修正されます。DBMaster は、2 フェーズ・コミットとトリガーを用いて、同期表レプリケーションを実行します。つまり、レプリケーションを設けると、ソース・データベースの更新は全て、DDB(分散型データベース)のアクションとなります。これは、ソース・データベースのふるまいにも影響します。ターゲット・データベースにアクセスできない場合、ソース・データベースへの変更はエラーになります。

もう 1 つは、非同期です。ターゲット・データベースへの修正は、後から行われることを意味します。ソース・データベースとターゲット・データベースの時間差は、ユーザーが定義したスケジュールによります。変更は一旦ソース表に保存され、スケジュールに従ってターゲット表が修正されます。このレプリケーションでは、レプリケーションで対になる 2 つのデータベースが独立しているので、ネットワークを利用できない時でも通常通り作業をすることができます。

用語の定義

ソース表

データをレプリケートするソース・データベースにある表。

ターゲット表

データをレプリケートされるターゲット・データベースにある表。

パブリケーション

レプリケーションに使用するソース表のデータの集まり。

サブスクリプション

パブリケーションを受け取るターゲット表にあるデータの集まり。

フラグメント(断片)

水平パーティションとも呼ばれています。フラグメントは、データ・タプルの一定範囲のレプリケーションです。

プロジェクション

レプリケーションのために選択した元の表から選択したカラム。

レプリケーション・ドメイン

レプリケーション・フラグメント(水平パーティション)とプロジェクション(垂直パーティション)を合わせたものを、レプリケーション・ドメインと呼びます。レプリケートされる表のデータ範囲です。

ドメイン変更をレプリケートする際に問題があります。UPDATE 文をレプリケートする際に発生します。

➲ **例**

```
dmSQL> CREATE REPLICATION rp_case1 WITH PRIMARY AS tb_example WHERE c2 > 0 REPLICATE
TO db2:t1;
dmSQL> CREATE REPLICATION rp_case2 WITH PRIMARY AS tb_example WHERE c2 < 0 REPLICATE
TO db2:t1;
```

```
dmSQL> UPDATE tb_example SET rp_case1=-7 WHERE rp_case2=7;
```

レプリケーション・ドメイン rp2 は、c2>0 です、レプリケーション・ドメイン rp3 は、c2<0 です。レプリケート時、UPDATE 文をレプリケートするのみではありません。更新したタプル・レプリケーション・ドメインは rp2 から rp3 に変更され、続いてレプリケーションは rp2 に DELETE 文(delete c2=-7)を実行し、rp3 に INSERT 文(insert c2=7)を実行します。

データの初期化

レプリケーション作成時、自動的にターゲット・データベースのデータを初期化する方法を指定することができます。表レプリケーションを作成した後、ソース・データベースへのいかなる変更(挿入、削除、更新)も、ターゲット・データベースに影響します。

データ初期化には、以下の 4 つのオプションがあります。

- **CLEAR DATA:** 表レプリケーションの際、ターゲット・データベースから全データを削除します。
- **FLUSH DATA:** ターゲット表にソース表のフラグメントを満たす表の全データを挿入します。
- **CLEAR AND FLUSH DATA:** 「clears data」した後、「flushes data」を行います。
- **指定しない:** ターゲット表を残します。

表レプリケーションの作成

以下の構文ダイアグラムは、非同期と同期表レプリケーションを表しています。

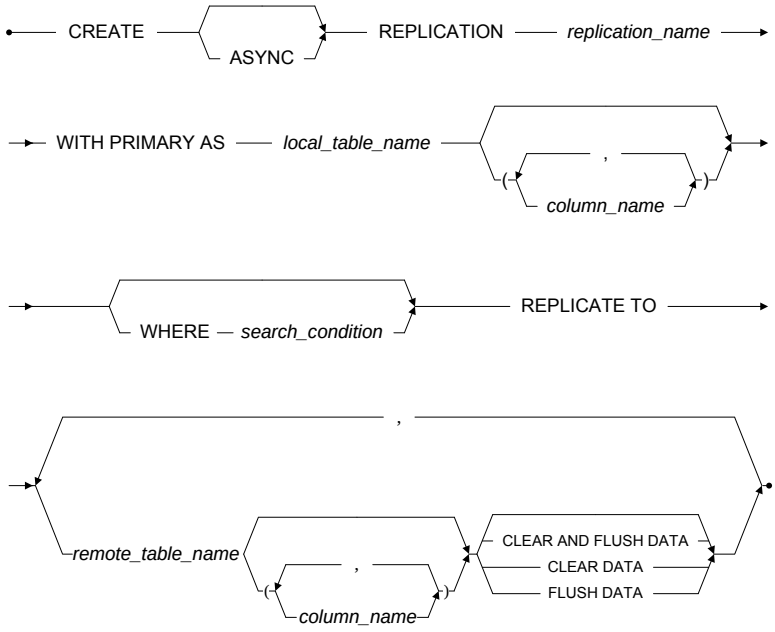


図 17-1：CREATE REPLICATION 文の構文

➲ **例**

データベース ***db30a*** に表 ***tb1***、データベース ***db30B*** に表 *tb2* があると想定します。***tb1*** のデータを ***tb2*** にコピーするレプリケーションを作成し、***tb2*** の現在のデータを修正させないようにする：

```
dmSQL> CREATE REPLICATION r1 WITH PRIMARY AS TB1 REPLICATE TO DB30B:TB2;
```

この例では、ターゲット・データベースを指定するために、データベース名を使っていますが、替わりデータベース・リンクを使うこともできます。

表レプリケーションの規則

- ソースとターゲット表のスキーマが必ず必要です。つまり表レプリケーション実行の際には、表を作成されません。
- 表のレプリケーション名は一意である必要があります。

- レプリケーションのサブスクライバは、<リンク名|データベース名>+ <表所有者名> + <表名>のフォームを使用して、一意にする必要があります。
- どの表の対象カラムにも、必ず主キーのカラムが必要です。
- 主キーのカラムは、必ずフラグメント・カラムに含まれる必要があります。
- 元の表の所有者、若しくはDBA以上の権限を持ったユーザーのみ、レプリケーションを作成、削除、修正する権限があります。
- ターゲット表にカラム識別子が存在しない場合、ターゲット・カラム名は元となる表名と同一にする必要があります。
- 主キー・カラムの数は、ソース表とターゲット表で同じ数にする必要があります。

➲ **例 1**

ソース・データベースから表 **tb_salary** を ***db1*** の表 **usr1.tb_salaryA** にレプリケートするパブリケーションを作成します。カラム名を定義しない場合、表 **tb_salary** の全カラムが **tb_salaryA** に存在し、カラムのデータ型が互換している必要があります。表 **tb_salary** の **id**, **name**,**basepay** を表 **usr1.tb_salaryA** のカラム **id**, **name**,**basepay** にレプリケートする：

```
dmSQL> CREATE REPLICATION rp salaryA with
       PRIMARY AS tb salry
       REPLICATE TO db1:usr1.tb_salaryA;
```

➲ **例 2**

db1 の **tb_salaryA** の(**Aid**, **Aname**)と、***db2*** の **tb_salaryB** の(**id**, **name**)に、カラム(**id**, **name**)の c1 > 100 のデータをレプリケートするために使用するパブリケーションを作成する。この文を実行すると、**tb_salary** の c1 > 100 の全データは、***db2*** の **tb_salaryB** と **tb_salaryA** の **Aid** と **Aname** にコピーされます：

```
dmSQL> CREATE REPLICATION rp_salaryAB with
                  PRIMARY AS tb_salary (id,name) where id > 100 ,
        REPLICATE TO db1:tb_salaryA (Aid,Aname),
                     db2:tb_salaryB flush data;
```

➲ 例 3

このコマンドは、まず *db1* の tb_salaryA から全データを削除し、*db1* の tb_salaryA の(Aid, Aname)にカラム(id, name)の c1 > 100 のデータをコピーするパブリケーションを作成する：

```
dmSQL> CREATE REPLICATION rp_salaryAClear with
                PRIMARY AS tb_salary (id,name) where id > 100 ,
        REPLICATE TO
                db1:tb_salaryA (Aid, Aname) clear data;
```

レプリケーションを削除する

このコマンドは、ソース表からレプリケーションを削除します。

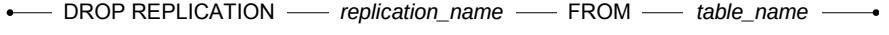


図 17-2：DROP REPLICATION 文の構文

➲ 例

表 tb_salary からレプリケーション rp_salaryA を削除する：

```
dmSQL> DROP REPLICATION rp_salaryA FROM tb_salary;
```

レプリケーションを修正する

既存のレプリケーションにターゲット表を追加したり、削除したりすることができます。次の構文ダイアグラムと例で方法を説明します。

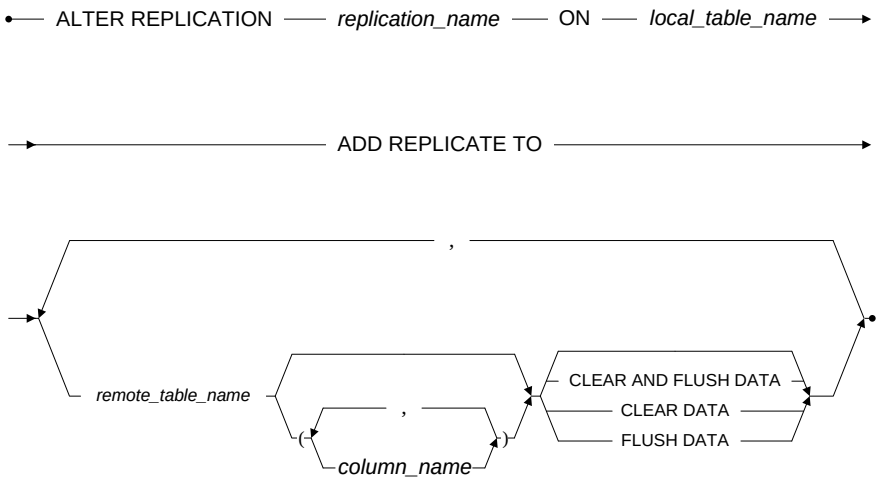


図 17-3 : ALTER REPLICATION … ADD REPLICATE TO 文の構文

➲ **例 1**

最初のコマンドは、データベース ***dbX*** の表 **tb_salaryX** にソース・データベースの表 **tb_salary** をコピーするレプリケーションを作成します。2 番目のコマンドは、表 **tb_salary** の **rp_AlterRp** レプリケーションに、***db1*** に **tb_salaryA**、***db4*** に **tb_salaryB** の 2 サブスクライバを追加します。

```
dmSQL> CREATE REPLICATION rp_AlterRp WITH PRIMARY AS tb_salary
       REPLICATE TO dbX:tb_salaryX;
dmSQL> ALTER REPLICATION rp_AlterRp ON tb_salary
       ADD REPLICATE TO db1: tb_salaryA,
                   db4: tb_salaryB (Bid, Bname) clear data;
```

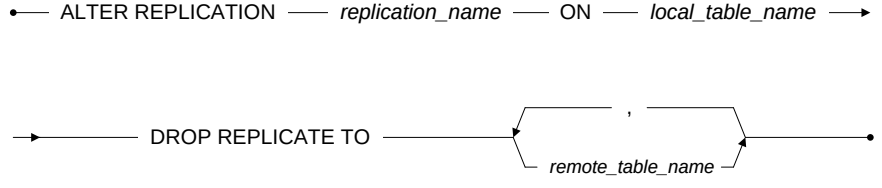


図 17-4 : ALTER REPLICATION … DROP REPLICATE TO 文の構文

➲ **例 2**

表 tb_salary のレプリケーション rp_AlterRp から、*db1* のサブスクライバ tb_salaryA を削除する。

```
dmSQL> ALTER REPLICATION r3 ON tb_salary
       DROP REPLICATE TO db1: tb_salaryA;
```

17.2 同期表レプリケーション

2 フェーズ・コミットで、分散したデータを同期することができます。相互に接続された全ての分散サイトで受け入れられた場合のみ、トランザクションは成功します。ネットワークをこえた精巧な「handshake（握手）」メカニズムは、分散したサイトで、各トランザクションの受け入れをコーディネートします。つまり、同期表レプリケーションを使用すると、いつデータを更新しても必ず同期されています。

同期表レプリケーションの設定

DBMaster は、同期表レプリケーションに 2 フェーズ・コミットを使用します。ソースとターゲット・データベースは、分散型データベース(DDB)モードである必要があります。まずデータベースを DDB モード(DD_DDbMd=1)にセットし、**dmconfig.ini** ファイルにデータベースのセッションを追加します。詳細については、16 章の「分散データベース」を参照して下さい。

表レプリケーション作成後、ソース表へのいかなる変更（挿入、削除、更新）も、ターゲット・データベースへ反映されます。

17.3 非同期表レプリケーション

同期表レプリケーションが、ソース表の修正と同時にターゲット表を修正する一方、非同期表レプリケーションはソース表に変更を保存し、スケジュールに従ってターゲット表を修正します。

DBMaster は、ソース表に変更を保存するためにレプリケーション・ログというファイルを使用します。ソース表への変更は、レプリケーション・ログに保存され、予め定義したスケジュールに基づいてターゲット表にレプリケートします。レプリケーション・ログを使用すると、DBMaster はソー

スDBトランザクションとターゲットDBトランザクションを独立して取り扱うことができるようになります。リモート接続が使用できない場合でもソース表を更新することができます。これにより、DBMasterは障害復旧まで再試行し続けるので、非同期表レプリケーションはネットワークやリモート・データベースの障害に対して柔軟になります。

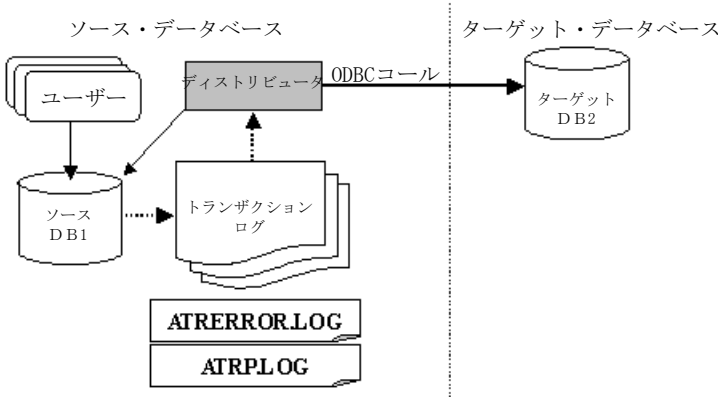


図 17-5：非同期表レプリケーションのアーキテクチャ

非同期表レプリケーションは、データ・レプリケーションを操作するためにレプリケーション・ログ・システム、バックグラウンド・デーモン、ディストリビュータを使用します。レプリケーション・ログは、DBMasterのジャーナル・ファイルではありません。レプリケーション・ログのレベルは、ジャーナルより高く、表レプリケーションのためにのみ使用されます。ジャーナルの内容は、物理的なデータの変更ですが、レプリケーション・ログは、レプリケーション表に適用するコマンドのようなものです。

ソース・データベースが起動している時、DBMasterはレプリケーション・ログファイルにソース表の修正のログを取ります。予定時刻に、ディストリビュータは、レプリケーション・ログどおりにターゲット・データベースのレプリケートする表の全変更を再び実行します。

通常ディストリビュータ・デーモンは、遠隔地のデータベースと通信するためにODBC関数コールを使用します。そのため、Oracle、SQL Server、Informixのような異種のデータベース・サーバーに、表をレプリケートする

ことができます。異種表レプリケーションは、この章の後方で説明します。高速非同期表レプリケーションは、ODBC 関数コールを使用しません。これについても同様に後ほど説明します。

➲ **非同期表レプリケーションを作成する：**

1. 非同期表レプリケーションを使用可能にします。
2. ターゲット・データベースにスケジュールを作成します。
3. スケジュールに従い、非同期表レプリケーションを作成します。

非同期表レプリケーションを使用可能にする

ソース・データベースには、ディストリビュータ・デーモンがあります。ディストリビュータは、定期的にターゲット・データベースに接続し、表レプリケーションを実行します。

ソース・データベースの **dmconfig.ini** ファイルのキーワード DB_AtrMd は、ディストリビュータ・デーモンを起動させるかどうかを指定します。ディストリビュータ・デーモンを起動させないと、データベースは非同期表レプリケーションのソースにはなりません。

ソース・データベースの **dmconfig.ini** ファイルのキーワード RP_LgDir は、非同期表レプリケーションのレプリケーション・ログをどこに配置するかを指定します。レプリケーション・ログファイルは、バイナリです。ユーザーは任意にそれらを削除することはできません。RP_LgDir の初期設定ディレクトリは、データベースのホーム・ディレクトリの TRPLOG という名前のサブディレクトリです。

スキーマをチェックし、表レプリケーションを作成する際、ソースとターゲット・データベースの分散型モードを ON(DD_DDbMd = 1)にする必要があります。スケジュールを NO CHECK にセットすると、DBMaster はスキーマのチェックを省略し、分散型データベース・モードは OFF になります。

➲ **例**

ソース・データベースの **dmconfig.ini** ファイルを使って、データベース ***srcdb*** の表をリモート・データベース ***destdb*** にレプリケートする：

```
[SRCDB]
DB DBDIR = /disk1/DBMaster/src
DB USRBB = /disk1/DBMaster/src/SRCDB.BB 3
DB USRDB = /disk1/DBMaster/src/SRCDB.DB 150
RP_LGDIR = /disk1/DBMaster/src/trplog
DB_ATRMD = 1
DD_DDBMD = 1
DB_SVADR = srcpc
DB_PTNUM = 22222

[DESTDB]
DB_SVADR = destpc
DB_PTNUM = 33333
```

ターゲット・データベースの **dmconfig.ini** ファイル：

```
[SRCDB]
DB SvAdr = srcpc
DB PtNum = 22222

[DESTDB]
DB DBDir = /disk3/DBMaster/dest
DB USRBB = /disk3/DBMaster/dest/DESTDB.BB 3
DB USRDB = /disk3/DBMaster/dest/DESTDB.DB 150
DD DDBMD = 1
DB SVADR = destpc
DB_PTNUM = 33333
```

非同期表レプリケーションの実行のために、ディストリビュータ・デーモンが ODBC ドライバ・マネージャを使用するため、ソース・データベースが Microsoft Windows 環境で運用されている場合、リモート・データベースの ODBC データ・ソース名(DSN)を設定する必要があります。

スケジュール(作成と削除)

ターゲット表への非同期レプリケーションを作成する前に、DBA 権限またはそれ以上を持つユーザーは、*スケジュール*を設定します。スケジュールには、開始日時、間隔、ターゲット・データベースへの接続に使用するアカウントとパスワードを指定します。1 つのソース・データベースに別々のターゲット・データベースのための複数のスケジュールを作成することが

できますが、1 つのターゲット・データベースに複数のスケジュールを設けることはできません。

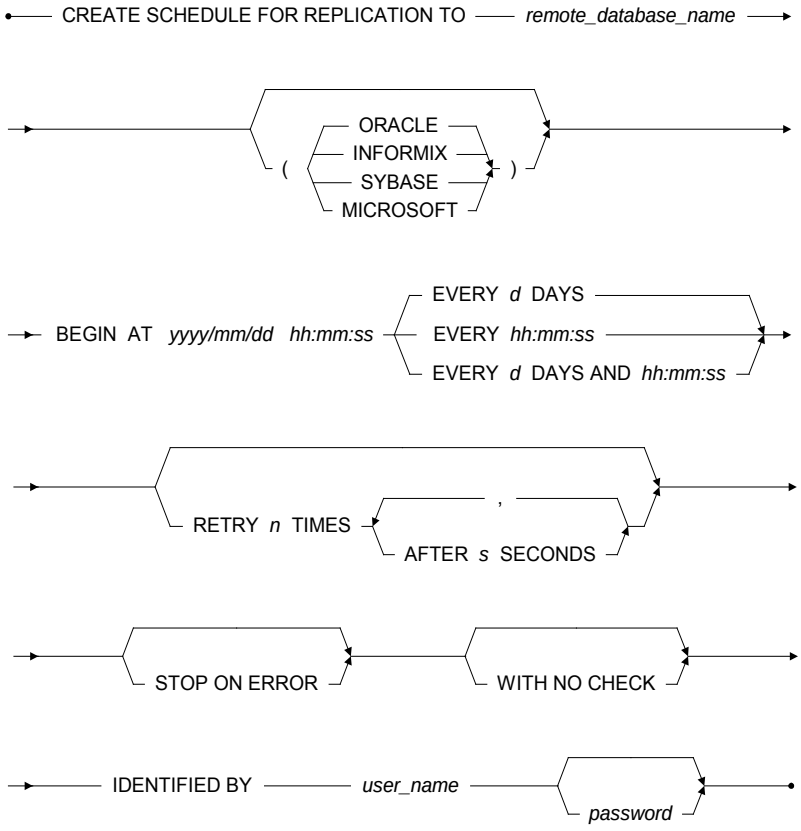


図 17-6 : CREATE SCHEDULE 文の構文

キーワード IDENTIFIED BY は、リモート・データベースへ接続し、表レプリケーションを実行するために、ディストリビュータ・デーモンが使用するリモート・アカウントです。このアカウントには、ターゲット表に挿入、削除、更新を実行することができる権限が必要です。

スケジュールが必要無くなり、いずれのレプリケーションでも使用されていない場合、DBA 権限のユーザーは、ソース・データベースでスケジュールを削除することができます。

図 17-7 : DROP SCHEDULE 文の構文

➲ **例 1**

データベース ***destdb*** へのレプリケーション・スケジュールを作成する：

```
dmSQL>   CREATE SCHEDULE FOR REPLICATION TO destdb
         BEGIN AT 2000/1/1 00:00:00
         EVERY 12:00:00
         IDENTIFIED BY User Password;
```

ディストリビュータ・デーモンは、2000 年 1 月 1 日以降 12 時間おきに、非同期レプリケーションを実行するために起動します。データベース・ディレクトリにあるディストリビュータ・メッセージ・ログ ATRP.LOG には、ディストリビュータ・デーモンの起動日時と状態が記録されています。

➲ **例 2**

スケジュールを削除する：

```
dmSQL> DROP SCHEDULE FOR REPLICATION TO destdb;
```

非同期表レプリケーションを作成する

同じスケジュールを利用する全ての非同期表レプリケーションは、同時に実行されます。非同期表レプリケーションのメカニズムは、LONG VARCHAR、LONG VARBINARY、FILE データ型を含む全てのデータ型をサポートします。

非同期表レプリケーションの作成は、同期表レプリケーションの作成とほぼ同じです。キーワード ASYNC のみ追加します。

➲ 例 1

データベース *srcdb* に、ターゲット・データベース *destdb* にスケジュールに基づいてレプリケートするレプリケーション rp_AsyRP を作成する：

```
dmSQL> CREATE ASYNC REPLICATION rp_AsyRP
       WITH PRIMARY AS tb_salary
       REPLICATE TO destdb: tb_salaryA
       CLEAR AND FLUSH DATA;
```

ユーザーは、ターゲット・データベース側でデータを初期化するために、CLEAR DATA、FLUSH DATA、CLEAR AND FLUSH DATA を指定することができます。レプリケーションを作成する時、確認と初期化のためにターゲット・データベースへの接続用に、データベース・リンクを使用することができます。このアクションは、*destdb* ターゲット・データベース名かデータベース・リンク名を使って、或いは現在のユーザー・アカウント経由で実行されます。

非同期表レプリケーション作成後、レプリケーションの作業は、ディストリビュータ・デーモンに移動します。ターゲット・データベースへの接続に使用されるアカウントは、CREATE SCHEDULE 文の IDENTIFIED オプションで指定したアカウントに変更されます。

非同期表レプリケーションの状態は、ソース・データベースのホーム・ディレクトリにあるディストリビュータ・メッセージ・ログ ATRP.LOG に記録されます。ディストリビュータ・メッセージ・ログは、純粋なテキスト形式で、ディストリビュータ・デーモンの起動日時と状態を記録します。

➲ 例 2

ディストリビュータ・メッセージ・ログの内容：

```
2000/02/09 10:02:30 : start up
2000/02/09 10:02:33 : replicate transactions before 2000/02/09
                      10:02:29 (log:1.856152) to DESTDB
```

NO CASCADEオプション

キーワード NO CASCADE は、オプションです。非同期レプリケーションの場合のみ指定することができます。このキーワードは、カスケード・レプリケーションを指定します。コマンドは、最も高いレベルから低いレベル

へと連鎖して実行されます。典型的なカスケード・レプリケーションの例は、A から B に、さらに C にデータをレプリケートします。典型的な非カスケード・モデルは、B にデータをレプリケートし、B はいずれにもデータをレプリケートしません。このタイプを使う場合は、NO CASCADE オプションを ON にします。初期設定は、CASCADE です。

NO CASCADE は、2 つのサイト間でのみ表レプリケーションを行います。

➲ 例

レプリケーション *Rp1* は *DB1:t1* から *DB2:t2* に、*Rp2* は *DB2:t2* から *DB3:t3* にレプリケートするような例で、*Rp1* が NO CASCADE モードの場合、*DB1:t1* の変更は、*DB2:t2* にレプリケートされますが、*DB2* は *DB3:t3* への同様の変更をレプリケートすることを中止します。*Rp1* が CASCADE モードの場合、*DB1:t1* への変更は *DB2:t2* へ、更に *DB2:t2* から *DB3:t3* にレプリケートされます。

エラー操作

データ・レプリケーションの過程で、ディストリビュータが遭遇するエラーには、警告、接続エラー、データ・エラー、文エラー、トランザクション・エラーの 5 種類があります。

警告

例えば、CHAR(10)データ型が CHAR(5)カラム型にレプリケートされた場合、データ切り落としの警告があります。ディストリビュータ・デーモンは、このようなエラーを無視します。

接続エラー

ディストリビュータ・デーモンは、リモート・データベース・サーバーに接続できない場合、そのスケジュールを放棄し次回まで待機します。全ての作業はそれまで保留されます。

データ・エラー

非同期表レプリケーションは、柔軟なデータベースの対なので、他の誰かが先にターゲット・データベースを更新する可能性があります。例えば、

ディストリビュータ・デーモンがターゲット・データベースにレコードを挿入しようとする際に、すでにそれが存在するような場合。また別の状況では、ディストリビュータ・デーモンがレコードを削除しようとする際に、それが存在しないというような場合。STOP ON ERROR オプションを使用して、このようなエラーの際にディストリビュータ・デーモンを停止させることができます。ディストリビュータの初期設定では、これらのエラーを無視されます。

注 *上述したデータ・エラーは、整合性違反エラーや結果行エラーを含みます。前者は、関数SQLError()を呼び出し、エラー23000を戻します。後者は、関数SQLRowCount()を呼び出し、結果は1つではありません。ODBC関数についての詳細は、「ODBCプログラマーガイド」を参照して下さい。*

文エラー

ディストリビュータ・デーモンが、SQL 文を実行してロック・タイムアウト・エラーに遭遇した時は、待機した後に再試行するために、RETRY *<n>* TIMES オプションを使用します。AFTER *<s>* SECONDS オプションは、次回の試行まで待機する時間を指定します。

トランザクション・エラー

例えば、デッド・ロックは、エラーのトランザクションをロールバックします。ディストリビュータ・デーモンが、トランザクションをロールバックするエラーに遭遇した場合、各トランザクションを再試行します。それでも結果がエラーである場合、ディストリビュータは次回のスケジュールまでこれらのアクションを保留します。

レプリケーション間に発生したエラーの記録は、ATRERROR.LOG という名前のテキスト・ファイルで確認することができます。このファイルは、データベース・ディレクトリにあります。

スケジュール(中断と再開)

東京にあるデータベースにデータのレプリケートを試みる例で、東京が祝日のためにデータベースが起動していないことをあらかじめ知っている場合、データベースの準備ができるまで、スケジュールを一時停止することができます。

DBA 権限のユーザーが、スケジュールを中断、再開することができます。スケジュールが中断されると、ディストリビュータ・デーモンは、レプリケーションのための接続を停止します。

➲ **例 1**

ターゲット・データベース ***destdb*** へのスケジュールを中断する：

```
dmSQL> SUSPEND SCHEDULE FOR REPLICATION TO destdb;
```

➲ **例 2**

ターゲット・データベース ***destdb*** へのスケジュールを再開する：

```
dmSQL> RESUME SCHEDULE FOR REPLICATION TO destdb;
```

スケジュールを同期させる

データを同調させる必要がある時があります。これを解決する方法をスケジュール同期と呼びます。スケジュール同期は、ディストリビュータ・デーモンに強制的に特定のデータベースに直ちにソース上の変更を実行させます。ユーザーは、ディストリビュータ・デーモン起動のスケジュールまで、待機する必要がありません。

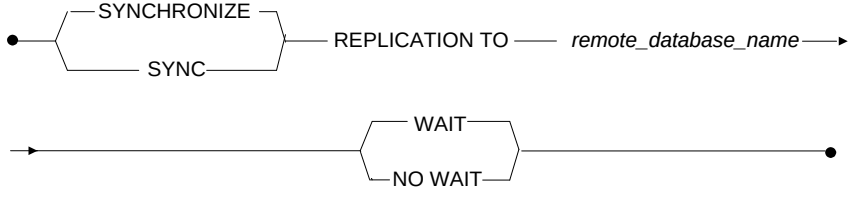


図 17-8 : SYNCHRONIZE REPLICATION 文の構文

➲ **例 1**

ターゲット・データベース ***destdb*** にレプリケートする：

```
dmSQL> SYNC REPLICATION TO destdb WAIT;
```

初期設定の WAIT オプションは、ディストリビュータ・デーモンが全ての変更を終了するまで待機します。このコマンドは、レプリケーションの完了後のみ戻ります。NO WAIT オプションは、ディストリビュータ・デーモンにその作業を直ちに実行させ、SYNC コマンドが直ぐに戻されます。

➲ **例 2**

NO WAIT オプションで SYNC REPLICATION TO を使用する：

```
dmSQL> SYNC REPLICATION TO destdb NO WAIT;
```

スケジュールを変更する

スケジュール作成後、DBA 権限を持つユーザーは、ディストリビュータの起動間隔、リモート接続に使用するアカウント、RETRY オプション、STOP/IGNORE ON ERROR オプション等のスケジュールの内容を修正することができます。

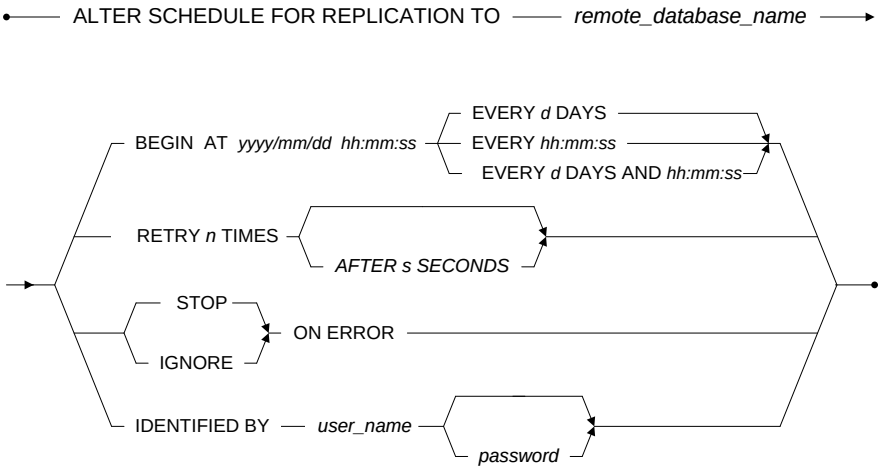


図 17-9：ALTER SCHEDULE 文の構文

➲ 例 1

IGNORE ON ERROR オプションを追加して、ターゲット・データベース ***destdb*** へのレプリケーションのスケジュールを修正する：

```
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb IGNORE ON ERROR;
```

➲ 例 2

```
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb
       IDENTIFIED BY User2 Password2;
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb STOP ON ERROR;
dmSQL> ALTER SCHEDULE FOR REPLICATION TO destdb RETRY 5 TIMES AFTER 3
       SECONDS;
```

異種非同期表レプリケーション

DBMaster は、他の DBMaster データベースだけでなく、Oracle、Microsoft SQL Server のデータベースへの非同期表レプリケーションも可能です。このようなレプリケーションは、異種表レプリケーションと呼ばれ、異種環境で他のデータベースと共存することができます。

DBMaster は、第 3 者のターゲット・データベースにデータを送信する前に、レプリケートしたデータを予め処理する必要があります。つまり異種環境のスケジュール作成の際に、ORACLE、MICROSOFT キーワードを使って、レプリケートする DBMS の種類を定義します。

DBMaster は非同期表レプリケーションを実行するために ODBC ドライバ・マネージャを使用しているので、DBMaster サーバーは Windows NT、若しくは Windows2000 に設ける必要があります。又、ターゲット・データベース名の定義に、リンク名を指定することはできません。第 3 者のターゲット・データベースは、Windows、UNIX、Linux のいずれかのプラットフォームに配置することができます。

異種表レプリケーションのスケジュールを作成する時、DBMaster がスキーマのチェックをしない様にする場合は、WITH NO CHECK キーワードを使用し、ユーザー自身がスキーマのチェックを行います。ターゲット表にあるカラムとデータ型が、ソース表にあるカラムとデータ型に互換していることを確認して下さい。

異種表レプリケーションを作成する時は、CLEAR DATA、FLUSH DATA、CLEAR AND FLUSH DATA キーワードを使用することはできません。レプリケーションを開始する前に、ターゲット・データベースのデータを手動で削除/挿入して、表を初期化された状態にします。

➲ **例 1**

ユーザー***orcuser***、パスワード ***mypassword*** で ODBC データソース名が ***orcdb*** の Oracle データベースに接続する：

```
dmSQL> CREATE SCHEDULE FOR REPLICATION TO orcdb (ORACLE)
       BEGIN AT 2000/01/01 00:00:00 EVERY 2 DAYS
       WITH NO CHECK
       IDENTIFIED BY orcuser mypassword;
```

➲ **例 2**

表 **tb_salary** に異種レプリケーションを作成する：

```
dmSQL> CREATE ASYNC REPLICATION rp HeteroRp
       WITH PRIMARY AS tb salary
       REPLICATE TO orcdb:orcuser.tb_salary;
```

高速非同期表レプリケーション

非同期表レプリケーションは、WAN 環境で低パファーマンスの原因となりえるターゲット・データベースと通信するために、ODBC 関数コールを使用します。DBMaster には、WAN での高パフォーマンスを実現する、高速非同期表レプリケーションと呼ばれるもう一つのメカニズムがあります。これは、コマンドをまとめ、ネットワーク間を移動するパッケージに格納します。

他の DBMS が、このプロトコルをサポートしていない為、高速非同期表レプリケーションは異種機で実行できません。又、高速スケジュール作成時に STOP ON ERROR オプションは使用できません。

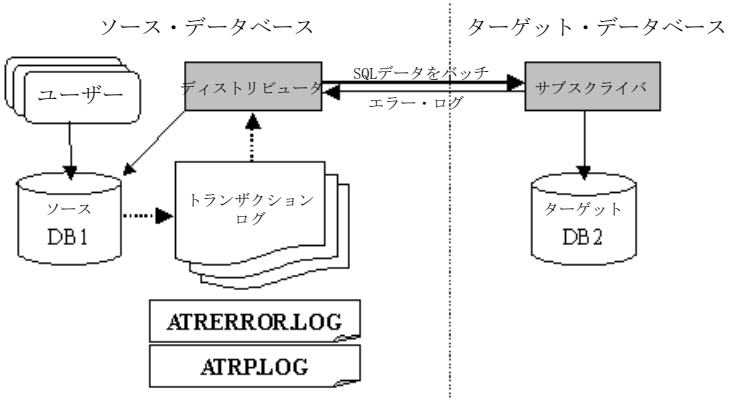


図 17-10：高速非同期表レプリケーションのアーキテクチャ

ソース・データベースにディストリビュータ・デーモン、ターゲット・データベースにサブスクライバ・デーモンがあります。それらが、高速非同期表レプリケーションを実行します。ディストリビュータは、ODBC コール経由でソース表からターゲット表に直接変更を適用しません。替わりに、SQL 文と、ソース表で適用される関連データをパッケージにし、ターゲット・データベースでサブスクライバ・デーモンにパケットを送信するだけです。ターゲット・データベースでは、パケットを取得した後、サブスクライバ・デーモンはターゲット表にそのコマンドを実行します。

高速レプリケーション設定

➲ **高速レプリケーションを作成する：**

1. ソース・データベースでディストリビュータ、ターゲット・データベースでサブスクライバ・デーモンを使用可能にします。
2. ターゲット・データベースで高速スケジュールを作成します。
3. スケジュールに基づいて、非同期表レプリケーションを作成します。

サブスクライバ・デーモンを使用可能にする

サブスクライバ・デーモンを起動するために、ターゲット・データベースの **dmconfig.ini** ファイルのキーワード **DB_EtrPt** をセットします。ディスト

リビュータ・デーモンとサブスクライバ・デーモン間の通信チャネルのポート番号を指定します。

➲ 例

ソース・データベース ***srcdb*** からターゲット・データベース ***destdb*** に高速レプリケーションを使ってレプリケートする場合、ターゲット・データベースでサブスクライバ・デーモンを起動させる必要があります。

ターゲット・データベースの **dmconfig.ini** ファイルのサンプル。

```
[SRCDB]
DB_SVADR = srcpc    ; tell the target database where the source
DB_PTNUM = 22222    ; database is
[DESTDB]
DB_DBDIR = /disk3/DBMaster/dest
DB_USRBB = /disk3/DBMaster/dest/DESTDB.BB 3
DB_USRDB = /disk3/DBMaster/dest/DESTDB.DB 150
DD_DDBMD = 1
DB_SVADR = destpc
DB_PTNUM = 33333
DB_ETRPT = 44444    ; port number used by Subscriber Daemon
```

ソース・データベースのキーワード **DB_AtrMd** は、ディストリビュータ・デーモンを起動するために使用します。ターゲット・データベースのキーワード DB_EtrPt で、ディストリビュータ・デーモンに、サブスクライバ・デーモンがどのポート番号を使用しているのかを明示する必要があります。

ソース・データベースの **dmconfig.ini** ファイルのサンプル。

```
[SRCDB]
DB_DBDIR = /disk1/DBMaster/src
DB_USRBB = /disk1/DBMaster/src/SRCDB.BB 3
DB_USRDB = /disk1/DBMaster/src/SRCDB.DB 150
RP_LGDIR = /disk1/DBMaster/src/trplog
DB_ATRMD = 1
DD_DDBMD = 1
DB_SVADR = srcpc
DB_PTNUM = 22222
[DESTDB]
DB_SVADR = destpc
DB_PTNUM = 33333
```

```
DB_ETRPT = 44444    ; port number used by Subscriber Daemon
```

高速非同期表レプリケーションのスケジュール

高速非同期表レプリケーションは、CREATE SCHEDULE コマンドの EXPRESS オプションを使用します。

➲ **例**

高速非同期表レプリケーションのスケジュールを作成する：

```
dmSQL> CREATE SCHEDULE FOR EXPRESS REPLICATION TO destdb
       BEGIN AT 2000/1/1 00:00:00
       EVERY 12:00:00
       IDENTIFIED BY User Password;
```

高速非同期表レプリケーションを作成する

この手順は、非同期表レプリケーションと同じです。詳細については、前述の「非同期表レプリケーションを作成する」、又は *「SQL 文と関数参照編」* をご覧下さい。

17.4 データベース・レプリケーション

多くの中小企業や組織が、一台のファイル・サーバー、若しくは本社に全てのデータを保管し、サーバーに直接接続する端末が必要です。このアーキテクチャでは、アプリケーション・システムが、全ての端末が同じビルや場所にある場合では、スムーズに運用されます。但し、遠隔地の支店がデータベースにアクセスする場合、データ転送の速度とネットワーク帯域のためにそのパフォーマンスは低くなりました。

DBMaster のデータベース・レプリケーションは、データの共有を可能にし、アクセス速度を向上させます。データベース・レプリケーションは、設定した時間に、ソース・データベースからターゲット・データベースへデータをコピーします。言い換えると、データを追加/修正/削除するといったソース・データベース上の変更を自動的にターゲット・データベースにコピーします。データベース・レプリケーションには、3 つの利点があります。まず、直接遠隔地でデータにアクセスできるので、アプリケーション・システムのパフォーマンスが向上します。次に、ソース・データベースとタ

ーゲット・データベースは同様にデータ更新されます。つまり、データ共有という目的が効率的に達成されます。3 つ目は、アプリケーションがターゲット・データベースに接続できない場合、ソース・データベースに接続して業務を継続することが可能です。データ・レプリケーションを使用するメリットがある反面、データ用のストレージや、レプリケーションの操作の処理が多くなります。

データベース・レプリケーションの基本

このセクションでは、データベース・レプリケーションの流れを図16-11 で説明します。データベース・レプリケーションは、ジャーナル・バックアップ・サーバーで運用します。データベースのバックアップとリストアの詳細については、15 章の「リカバリ、バックアップ、リストア」を参照して下さい。

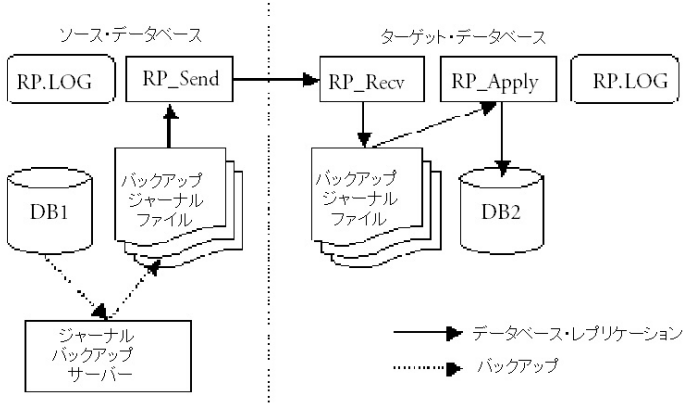


図 17-11 : データベース・レプリケーションの流れ

データベース・レプリケーションは、4 つのサーバーで実行します。ジャーナル・バックアップ・サーバー、**RP_Send** サーバー、**RP_Recv** サーバー、**RP_Apply** サーバーです。ジャーナル・バックアップ・サーバーと **RP_Send** サーバーはソース・データベースで起動します。**RP_Recv** サーバーと **RP_Apply** サーバーは、ターゲット・データベースで起動します。

データベースの最初のステップは、ターゲット・データベースをソース・データベースと同じ状態にすることです。それ以降のデータ挿入、削除、スキーマ作成等の全変更は、ソース・データベースで行います。その後、ソース・データベースで運用されているジャーナル・バックアップ・サーバーは、定期的にバックアップ・ジャーナルに変更を書き込みます。

RP_Send サーバーは、定期的にソース・データベースのバックアップ・ジャーナルをターゲットの RP_Recv サーバーに送信します。RP_Recv は、受信したバックアップ・ジャーナルをターゲット・データベースに適用するよう、RP_Apply に通知します。

ターゲット・データベースは、全ユーザーに対して読み込み専用ですので、RP_Apply のみ、ターゲット・データベースを更新することができます。

データベース・レプリケーションの設定

➲ **データベース・レプリケーションを手動で設定する：**

1. ソース・データベースをターゲット・データベースにコピーします。
2. ソース・データベースにジャーナル・バックアップ・サーバーと RP_Sendサーバーをセットします。
3. ターゲット・データベースにRP_Recvサーバーと RP_Applyサーバーをセットします。
4. ソースとターゲット・データベースを起動します。
5. レプリケーション・ログ(RP.LOG)とエラー・ログ(DMERROR.LOG)を確認します。

完全バックアップ

前述のように、データベース・レプリケーションの最初のステップは、ターゲット・データベースをソース・データベースと同一にすることです。ターゲット・データベースとソース・データベースの機器のバイト・オーダーを同一であることを確認して下さい。例えば、ソース・データベースが x86 機で運用されている場合、ターゲット・データベースの機器は、x86 互換オーダーにする必要があります。Sun Sparc のバイト・オーダーは、x86

のものとは異なります。そのため、ターゲット・データベースは、Sparc 機では運用できません。その逆も同様です。

➲ ソース・データベースをコピーする：

1. ソース・データベースを終了します。
2. ソース・データベースのデータ・ファイル、BLOBファイルとジャーナル・ファイルをコピーします。
3. ターゲット・データベース機にファイルのコピーを移動します。
4. 対応するファイル・ディレクトリを変更するために、ターゲットの dmconfig.iniの環境設定ファイルを修正します。

手順 1 と手順 2 を合わせると、オフライン完全バックアップになります。詳細については、14.5節の「オフライン完全バックアップ」を参照して下さい。

ここでは、手動オフライン完全バックアップの方法を説明します。このセクションで使用する例は、全てソース・データベースが x86 プロセッサと Linux か FreeBSD で運用されていることを想定しています。

➲ 例

ソース・データベースのファイルをコピーし、論理名を取得するために SYSFILE システム表に問い合わせる：

```
dmSQL> select * from SYSFILE;
```

dmconfig.ini ファイルで物理ファイル名を確認する：

```
[MYDB]     ;; ソース・データベースの環境設定, CPU：x86, OS: FreeBSD
DB_DBDIR = /home/DBMaster/mydb
DB_USRBB = /home/DBMaster/mydb/MYDB.BB 3
DB_USRDB = /home/DBMaster/mydb/MYDB.DB 150
FILE1 = /home/DBMaster/mydb/data/FILE1.DB 50
FILE2 = /home/DBMaster/mydb/data/FILE2.DB 50
DB_JNFIL = JN1.JNL JN2.JNL
...
```

ソース・データベースを終了した後、ターゲット・データベースを運用している PC にソース・データベースをコピーして下さい。ここでは、ターゲ

ット・データベース機は、Windows NT を運用している x86 と想定しています。例えば、コピーするファイルには、システム・ファイル MYDB.SDB と MYDB.SBB、初期設定ユーザー・ファイル MYDB.DB と MYDB.BB、ジャーナル・ファイル JN1.JNL と JN2.JNL、全ユーザー定義ファイル FILE1.DB、FILE2.DB 等が含まれます。

次に、**dmconfig.ini** 環境設定ファイルのソース・データベースのセクションを、ターゲット・データベースの **dmconfig.ini** にコピーして下さい。そして、絶対パスとパス名の規則が異種プラットフォーム間で違う可能性があるので、物理ファイル名を論理的に一致させるために、ターゲット・データベースの **dmconfig.ini** を修正して下さい。

ターゲット・データベースの **dmconfig.ini** ファイルの引用は以下のようになります。

```
[MYDB]     ;; ターゲット・データベースの環境設定, CPU : x86, OS: MS Windows NT
DB_DBDIR = d:\DBMaster\db\mydb
DB_USRBB = d:\DBMaster\db\MYDB.BB 3
DB_USRDB = d:\DBMaster\db\MYDB.DB 150
FILE1    = d:\DBMaster\db\mydb\FILE1.DB 50
FILE2    = d:\DBMaster\db\mydb\FILE2.DB 50
DB_JNFIL = JN1.JNL JN2.JNL
...
```

DBMaster では、1 つのソース・データベース当たり、256 つまでターゲット・データベースを使用できますので、複数のターゲット・データベースがある場合、システム管理者は各ターゲット・データベースに対して上記の手順を繰り返します。

ソース・データベースのジャーナル・バックアップ・サーバーを設定する

データベースの全変更はジャーナル・ファイルにログされるので、DBMaster ではレプリケーションのためにソース・データのバックアップ・ジャーナル・ファイルを使用します。ジャーナル・バックアップ・サーバーによって増分バックアップが実行された後、**RP_Send** サーバーはバックアップ・ジャーナル・ファイルをターゲット・データベースに送信します。その後、ジャーナル・ファイルにログした全トランザクションを実行することができ、ソース・データベースの全変更は、ターゲット側にも反映されます。

➲ 例

ジャーナル・バックアップ・サーバーを起動するには、ソース・データベースのみ必要です。サーバーを起動し、バックアップ・ディレクトリとスケジュールを指定するために **dmconfig.ini** ファイルを使用する：

```
DB_BMODE = 1                          ; BACKUP-DATA モードでデータベースを起動
DB_BKSVR = 1                          ; ジャーナル・バックアップ・サーバーを起動
DB_BKDIR = /home/DBMaster/mydb/bkdir ; バックアップ・ジャーナルファイルのディレクトリ
DB_BKTIM = 00/01/01 00:00:00          ; バックアップ開始日時
DB_BKITV = 0-12:00:00                 ; 12 時間毎にジャーナル・バックアップを実行
```

DB_BMode は、BACKUP-DATA (1)、又は BACKUP-DATA-AND-BLOB(2)モードになります。但し、DB_BMode が BACKUP-DATA-AND-BLOB モードの場合でも、TABLESPACE 作成の際に BACKUP BLOB ON を設定する必要があります。それ以外の場合では、BLOB データはターゲット・データベースにコピーされません。

データ送信と受信

データベース・レプリケーションの過程には、3 つの固有サーバーRP_SEND と **RP_RECV** と **RP_APPLY** が関係します(図16-11 参照)。**RP_SEND** は、ソース・データベースに位置し、ターゲット側にバックアップ・ジャーナル・ファイルを送信します。**RP_RECV** と **RP_APPLY** は、ターゲット・データベースに存在します。**RP_RECV** は、ソース・データベースからバックアップ・ジャーナル・ファイルを受信し、**RP_APPLY** はこれらのジャーナル・ファイルを実行します。**RP_Send** の起動時間は、ソース・データベースと同じです。**RP_RECV** と **RP_APPLY** も、ターゲット・データベースと同時に起動し、終了します。

ジャーナル・バックアップ・サーバーが増分バックアップを完了した後、**RP_SEND** は、未送信のバックアップ・ファイルを、バックアップ・ディレクトリ(**DB_BkDir**)からターゲット・データベースに送信します。一方、ターゲット・データベースの **RP_RECV** は、ソース・データベースから送信されたバックアップ・ファイルを全て受信し、ターゲット・データベースのバックアップ・ディレクトリ(**DB_BkDir)**にそれらのファイルを配置します。送信と受信が行われた後、**RP_APPLY** はバックアップ・ジャーナル・

ファイルに記録されている変更をターゲット・データベースに実行し、データベース・レプリケーションの一連の流れが完了します。

ソース・データベースにRP_SENDサーバーを設定する

RP_SEND と **RP_RECV** は、各々送信と受信を行います。つまり、**RP_SEND** は、**RP_RECV** がある機器の IP アドレスとポート番号を知る必要があります。データ・レプリケーションの際の **RP_SEND** と **RP_RECV** 間の通信のために、ポート番号を明確にして下さい。又、データベース・サーバーで使用されている番号と異なる番号を指定する必要があります。ソース・データベースの **dmconfig.ini** のキーワード **RP_SlAdr** を使用して、**RP_SEND** がレプリケーション・データをどこに送信するかを指定します。

RP_SlAdr の構文：

```
RP_SLADR = {address[:port number]}
```

初期設定ポート番号は、23001 です。

➲ 例 1

ソース・データベースは、256 つまでのターゲット・データベースを使用できます。カンマかスペースで、ターゲット・データベースの情報を区切ります。以下の例は、3 つのターゲット・データベース、192.168.9.222 (ポート番号 5100)、Mars (ポート番号 5101)、Scorpio (初期設定のポート番号 23001) を指定する：

```
RP_SLADR = 192.168.9.222:5100, Mars:5101, Scorpio
```

データをコピーする場所を定義した後、RP_SEND がデータベース・レプリケーションの実行を開始する日時と時間間隔を設定します。スケジュールを設立すると、RP_SEND は定期的にデータ・レプリケーションを実行します。

➲ 例 2

dmconfig.ini ファイルに、開始日時を 2000/01/01/ AM01:00、時間間隔を 1 日に設定する：

```
DB_SMODE = 4                  ; ソース・データベースとして起動し、RP_SEND サーバーも起動
RP_BTIME = 00/01/01 01:00:00 ; レプリケーション開始日時
RP_ITERV = 1-00:00:00          ; 1 日毎にレプリケーション実行
```

```
RP_SLADR = 192.168.9.222:5100, Mars:5101, Scorpio ; 3 つの機器にレプリケート
RP_RETRY = 3                  ; ネットワーク接続エラーの際の再試行の回数
RP_CLEAR = 1      ; ターゲット側にバックアップ・ジャーナルファイルを送信した後、削除
```

環境設定ファイルの **RP_BTime** と **RP_Iterv** は、データ・レプリケーションのスケジュールを定義するために使用します。**RP_BTime** は、バックアップ・ジャーナル・ファイルをターゲット側への送信を開始する日時です。**RP_BTime** のフォーマットは、<年/月/日 時間:分:秒>です。指定しない場合は、**RP_BTime** はソース・データベースの起動日時になります。**RP_Iterv** は、システムがデータベース・レプリケーションを自動的に実行する間隔を表しています。**RP_Iterv** のフォーマットは、<日-時間:分:秒>です。初期設定値は、1 日です。有効値の範囲は、0 から 24855 です。

注 *データベース起動前にこれらの値を設定します。*

RP_ReTry は、ネットワーク接続に失敗した場合、再試行する回数を指定します。**RP_Clear** は、バックアップ・ジャーナル・ファイルをターゲット側に送信した後に、削除するかどうかを定義します。**RP_Clear** を 1 にセットすると、バックアップ・ジャーナル・ファイルを削除します。初期設定値は 0 です。バックアップ・ジャーナル・ファイルがデータベース・レプリケーションにのみ使用される場合、これらのファイルを削除することは、ストレージ領域の節約になります。但し、ハードウェアに障害が発生した際に、バックアップ・ジャーナル・ファイルからデータをリストアすることができません。この場合、ソース・データベースをリストアするためにターゲット・データベースに完全バックアップを行う必要があります。つまり、ソース・データベースのバックアップにも、バックアップ・ジャーナル・ファイルを必要とするので、**RP_Clear** を 0 にセットすることをお勧めします。

ターゲット側でRP_RECVとRP_APPLYサーバーを設定する

RP_RECV と **RP_APPLY** サーバーを起動するために、ターゲット・データベースの起動モード **DB_SMode** を 5 にセットします。

➲ 例 1

ターゲット・データベースは、1 つのソース・データベースからのみレプリケーションを受信します。**RP_Primy** でソース・データベースの機器名、又はアドレスを指定する：

```
RP_PRIMY = FreeBSD        ; 機器 FreeBSD からレプリケーション・データを受信
```

➲ 例 2

RP_RECV サーバーは、**RP_SEND** からデータを受信するために、**DB_PtNum** と異なるポート番号を使用する。例えば、NTPC という機器にターゲット・データベースがあると想定した場合のキーワード **RP_PtNum** の設定：

```
RP_PTNUM = 5100             ; RP_SEND と RP_RECV 接続用のポート番号
DB_PTNUM = 3333             ; ユーザー・データベース・アクセス用のポート番号
```

➲ 例 3

ソース・データベースに、ターゲット・データベースの機器名、又はアドレスとポート番号を指定するキーワード **RP_SlAdr** を設定する：

```
RP_SLADR = NTPC:5100        ; ターゲット側の機器名とアドレス
```

加えて、ターゲット・データベースに、ソース・データベースから受信したバックアップ・ジャーナル・ファイルを保存する **DB_BkDir** バックアップ・ディレクトリを設定する必要があります。**RP_APPLY** が、バックアップ・ジャーナル・ファイルを使って、ターゲット・データベースに変更を適用した後、それらは自動的に削除されます。

読み込み専用のターゲット・データベース

ターゲット・データベースは、ソース・データベースと必ず同一です。データ定義(Create Table と Alter Table のような DDL)や、データ更新(INSERT、UPDATE、DELETE のような)を受け入れません。つまり、ターゲット・データベースは、読み込み専用ということになります。

データベース・レプリケーションの処理中に、ターゲット・データベースは、データをリストアするためにソース・データベースから受信したバックアップ・ジャーナル・ファイルを使用します。システムは、データ・リストアの処理中、データをロックしません。それゆえ、ターゲット・データベースへの問合せは、基本的に一種のダーティ・リードです。言い換え

ると、**RP_APPLY** がデータをリストアしている為に誤ったデータを読み込む可能性があります。

ソース/ターゲット・データベースを起動する

ソース・データベースとターゲット・データベースの起動モードは異なります。**dmconfig.ini** ファイルの **DB_SMode** を使って、設定します。ソース・データベース・モードの起動モード DB_SMode は 4 です。ターゲット・データベース・モードの起動モード **DB_SMode** は 5 です。

ソースとターゲット・データベースは別々に起動します。特別な順序はありません。

dmconfig.ini にあるレプリケーションのための全キーワードの要約は、のちほど説明します。

➲ 例

下記の例のソース・データベース名は FreeBSD、ターゲット・データベース名は NTPC です。ソース・データベースの必要最低限の設定です：

```
[MYDB] ;; ソース・データベースの環境設定, CPU：x86, OS：FreeBSD, Name：FreeBSD
;; データベース関連設定
DB_DBDir = /home/DBMaster/mydb
DB_USRBB = /home/DBMaster/mydb/MYDB.BB 3
DB_USRDB = /home/DBMaster/mydb/MYDB.DB 150
FILE1 = /home/DBMaster/mydb/data/FILE1.DB 50
FILE2 = /home/DBMaster/mydb/data/FILE2.DB 50
DB_JNFIL = JN1.JNL JN2.JNL
DB_SVADR = FreeBSD                  ; ソース・データベースの機器名
DB_PTNUM = 3333                     ; ソース・データベースのポート番号
;; ジャーナル・バックアップ・関連設定
DB_BMODE = 1                        ; データベースを BACKUP-DATA モードで起動
DB_BKSVR = 1                        ; ジャーナル・バックアップ・サーバー起動
DB_BKDIR = /home/DBMaster/mydb/bkdir ; バックアップ・ジャーナルファイルのディレクトリ
DB_BKTIM = 00/01/01 00:00:00        ; バックアップ開始日時
DB_BKITV = 0-12:00:00               ; 12 時間毎にジャーナル・バックアップ実行
;; RP_SEND サーバー関連設定
DB_SMODE = 4                        ; ソース・データベースとして起動、
                                    ; 同時に RP_SEND サーバーも起動
RP_BTIME = 00/01/01 01:00:00        ; レプリケーション開始日時
```

```
RP_ITERV = 1-00:00:00                ; 1 日毎にレプリケーション実行
RP_SLADR = NTPC:5100                 ;ポート番号 5100 で NTPC にレプリケート
RP_RETRY = 3                        ; ネットワーク接続エラーの際、3 回再試行する
RP_CLEAR = 1                        ; 送信後、バックアップ・ジャーナルファイルを削除
```

下記は、ターゲット・データベースの必要最低限の設定です。

```
[MYDB];; ターゲット・データベースの環境設定., CPU : x86, OS : MS Windows NT, Name : NTPC
;; データベース関連設定
DB_DBDir = d:\DBMaster\db\mydb
DB_USRBB = d:\DBMaster\db\MYDB.BB 3
DB_USRDB = d:\DBMaster\db\MYDB.DB 150
FILE1    = d:\DBMaster\db\mydb\FILE1.DB 50
FILE2    = d:\DBMaster\db\mydb\FILE2.DB 50
DB_JNFIL = JN1.JNL JN2.JNL
DB_SVADR = NTPC
DB_PTNUM = 3333
;; RP_RECV と RP_APPLY サーバー関連設定
DB_SMODE = 5              ; ターゲット・データベースとして起動,
                          ; 同時に RP_RECV と RP_APPLY サーバーも起動
RP_PRIMY = FreeBSD        ; この機器からのみレプリケーション・データを受信
RP_PTNUM = 5100           ; RP_SEND と RP_RECV 接続用のポート番号
DB_BKDIR = e:\mydb\bkdir  ; 一時バックアップ・ジャーナルファイルのディレクトリ
```

直ちにデータベース・レプリケーションを実行する

データベース・レプリケーション固有のサーバーが、データの変更を検出し、自動的にデータをレプリケートすることは既に述べました。

➲ 例

DBMaster に直ちにデータベース・レプリケーションを実行させる：

```
dmSQL> Set Flush;
```

ソースとターゲット・データベース間でデータを同調させる必要がある時に、このコマンドを使用して下さい。

このコマンドを実行すると、ジャーナル・バックアップ・ファイルは直ちに増分バックアップを実行し、現在のジャーナル・ファイルをバックアップします。その後、3 つの固有サーバー(RP_SEND、RP_RECV、RP_APPLY)が手順通りにデータをレプリケートします。

レプリケーション・ログ(RP.LOG)とエラー・ログ(DMERROR.LOG)を確認する

データベース・レプリケーションの過程で、ネットワーク障害やエラー・メッセージがあった場合、現在のデータベース・ディレクトリにログファイル DMERROR.LOG が生成されます：

```
yy/mm/dd hh:mm:ss Daemon name:Error number:Error message
```

➲ **例 1**

DMERROR.LOG：

```
97/12/31 11:40:59 - RP_SEND:rc = 1503, cannot connect to server
192.72.116.130
97/12/31 11:43:36 - RP_SEND:rc = 1503, cannot connect to server
192.72.116.130
97/12/31 11:45:00 - RP_SEND:rc = 1503, cannot connect to server
192.72.116.130
97/12/31 11:50:00 - RP_SEND:rc = 1503, cannot connect to server
192.72.116.130
97/12/31 11:50:45 - RP SEND:rc = 1503, cannot connect to server
192.72.116.130
```

ソースでもターゲット・データベースでも、DMERROR.LOG が生成される可能性があります。

レプリケーションが成功した場合でも、システムは RP.LOG という名前のログファイルを生成します。そのフォーマットは以下のようになります。

ソース・データベースでは以下のとおりです：

```
RP_SEND:RPID id ~ id sent at yy/mm/dd hh:mm:ss
```

ターゲット・データベースでは以下のとおりです：

```
RP RECV:RPID id ~ id sent at yy/mm/dd hh:mm:ss
RP_APPLY:RPID id ~ id applied at yy/mm/dd hh:mm:ss
```

レプリケーション・ログ RP.LOG は、ソースとターゲット・データベースのサーバー上にあります。いずれのバックアップ・ジャーナル・ファイルにも id があります。上記のフォーマットでは、RPID はどのジャーナル・ファイルが処理されるかを示します。RP_SEND 行の RPID は送信されたものを示し、RP_RECV 行の RPID は受信されたものを指し、RP_APPLY の RPID

はリストアされたものを意味します。通常、RPID はバックアップ・ジャーナル・ファイル名に含まれます。

➲ **例 2**

ソース・レプリケーションの RP.LOG は以下のようになります：

```
RP_SEND : RPID  7 ~ 10 sent to 192.72.116.130 at 97/12/16 15:36:17
RP_SEND : RPID 11 ~ 11 sent to 192.72.116.130 at 97/12/16 15:59:42
RP_SEND : RPID 12 ~ 12 sent to 192.72.116.130 at 97/12/31 11:52:28
```

➲ **例 3**

ターゲット・データベースの RP.LOG は以下のようになります：

```
RP_RECV :  RPID  7 ~ 10 received at 97/12/16 15:35:53
RP APPLY : RPID  7 ~ 10 applied at 97/12/16 15:35:55
RP RECV :  RPID 11 ~ 11 received at 97/12/16 15:59:18
RP APPLY : RPID 11 ~ 11 applied at 97/12/16 15:59:18
RP RECV :  RPID 12 ~ 12 received at 97/12/31 11:52:01
RP_APPLY : RPID 12 ~ 12 applied at 97/12/31 11:52:02
```

RP.LOG は、**RP_SEND**、**RP_RECV**、**RP_APPLY**、3 種類の固有サーバーによって実行される全アクションを記録するために使用されます。ファイルは、レプリケーション全参加データベースのデータベース・ディレクトリ **DB_DbDir** にあります。データベース管理者は、レプリケーションをスムーズに運用するために、これらのファイルを定期的に監視する必要があります。

例えば、リモート・データベースへの接続がいつも失敗する場合、ネットワークが正常に動いているか、リモート・データベースがエラーの際に使用されているかどうかをチェックして下さい。

JSERVER MANAGER環境の設定

データベースをレプリケートするために、データベースの初期環境を構築する必要があります。前節のように、直接に **dmconfig.ini** ファイルを編集する場合は、テキスト編集ソフトを使用します。この節では、DBMaster の JServer Manager を使って、データベース・のレプリケーション環境を設定します。

完全バックアップを行う

まず、ソース・データベースの完全バックアップを実行します。次に、そのバックアップをターゲット・データベースにコピーします。オフライン完全バックアップの詳細については、14.5節の「オフライン完全バックアップ」を参照して下さい。

ソース・データベースの設定

データベース・レプリケーションのサーバーを起動するためには、バックアップ・サーバーを起動させる必要があります。詳細については、15 章の*「リカバリ、バックアップ、リストア」*を参照して下さい。

➲ **ソース・データベース環境を設定する：**

1. ソース・データベースで、JServer Managerを起動します。
2. **［データベースの起動］**をクリックします。
3. **［データベース起動の高度な設定］**ウィンドウの**［レプリケーション］**タグをクリックします。
4. データベース・レプリケーションを使用可能にします。
   a) **［ターゲット・データベースのIPとポート番号］**の欄に入力します。
   b) **［データベース・レプリケーションの開始日時］**の欄に日付と時間を入力します。
   c) **［エラー時の再試行の回数］**の欄に数値を入力します。
   d) 必要であれば、**［レプリケーション後にバックアップ・ジャーナル・ファイルを削除］**のチェックボックスをクリックします。
   e) **［データベース・レプリケーション実行の間隔］**の欄に、各レプリケーション間の間隔の日数、時間、分、秒を入力します。
5. **［保存］**ボタンをクリックします。
6. **［データベースの起動］**ウィンドウに戻る場合は、**［取消］**ボタンをクリックします。

ターゲット・データベースの設定

ソース・データベースの完全バックアップをターゲット・データベースにコピーした後、JServer manager を使ってターゲット・データベースの環境設定を設定します。

1. ターゲット・データベースで、JServer Managerを起動します。
2. ［**データベースの起動**］をクリックします。
3. ［**データベース起動の高度な設定**］ウィンドウの［**レプリケーション**］タグをクリックします。
4. データベース・レプリケーションを使用可能にします。
   a) ［**ソース・データベースのIPアドレス**］の欄に入力します。
   b) ［**ターゲットDB受信デーモンのポート番号**］にRP_RECV用のポート番号を入力します。
5. ［**保存**］ボタンをクリックします。
6. ［**データベースの起動**］ウィンドウに戻る場合は、［**取消**］ボタンをクリックして下さい。

データベース環境設定ファイル

この節は、データベース・レプリケーションに関連する **dmconfig.ini** ファイルのキーワードの要約です。

ソース・データベース環境設定

データ・レプリケーションのソース・データベースのキーワード:

- **DB_SMode** **DB_SMode**を4にセットすると、このデータベースがソース・モードで起動することを意味します。
- **RP_BTime** ターゲット・データベースにソース・データベースの完全バックアップ・ファイルの送信を開始する時間を設定します。フォーマットは、yr/mon/day hr:min:secです。初期設定値は、ソース・データベースの起動時間です。例、97/12/31 12:00:00。

- **RP_Iterv**　バックアップ・ジャーナル・ファイルを送信する時間間隔を指定します。フォーマットは、day-hr:min:secです。例えば、1-12:00:00は、1日半おきにバックアップを送信することを意味します。日数の有効値の範囲は、0から24855です。
- **RP_ReTry**　ネットワーク・エラーの際に接続を再試行する回数を指定します。
- **RP_Clear**　送信後、ジャーナル・バックアップ・ファイルを削除するかどうかを指定します。値を1にするとファイルを削除し、値を0にすると削除しません。初期設定値は0です。値を1にすると、ハードウェアに障害がある場合に、ターゲット・データベースをリストアに使用しない限り、ソース・データベースをリストアすることができません。
- **RP_SlaDr**　この値は、ターゲット・データベースのアドレス(又は機器番号)とポート番号を指定します。各ソース・データベースにつき、8つまでのターゲット・データベースを指定できます。

RP_SlaDr の構文：

```
RP_SLADR = {Address[:Port Number]}
```

増分バックアップを実行するために、ソース・データベースでジャーナル・バックアップ・サーバーを起動する必要があります。:

- **DB_BMode** 1(BACKUP-DATA)、又は2(BACKUP-DATA-AND-BLOB)。
- **DB_BkSvr**　ジャーナル・バックアップ・サーバーを起動させるには、**DB_BkSvr**を1にセットして下さい。
- **DB_BkDir**　ジャーナル・バックアップ・ファイルを保存するディレクトリ
- **DB_BkTim** ジャーナル・バックアップ・サーバーの起動日時。フォーマットは、yr/mon/day hr:min:secです。初期設定値は、ソース・データベースの起動日時です。
- **DB_BkItv**　増分バックアップを実行する時間間隔。フォーマットは、day-hr:min:secです。例えば、0-12:00:00は、12時間毎に実行すること意味します。

ターゲット・データベース環境設定

データ・レプリケーションのターゲット・データベースのキーワードは: 以下のようになります。

- **DB_SMode** **DB_SMode**を5にセットすると、このデータベースをターゲット・データベースで起動することを意味します。
- **RP_Primy**　ソース・データベースのアドレス、又は機器名。
- **RP_PtNum** **RP_Recv**が使用するポート番号、この値は**DB_PtNum**と異なり、ソース・データベースのポート番号RP_SlaDrと同一のものを指定します。
- **DB_BkDir**　ソース・データベースから受信した一時バックアップ・ジャーナル・ファイルを保存するディレクトリ。初期設定ディレクトリは、**DB_FoDir**で指定されています。

データベース・レプリケーションの制限

データベース・レプリケーションの制限要約は以下のようになります：

- バイト順がソースとターゲット機で同一である必要があります。
- ターゲット・データベースは、読み込み専用です。
- 1つのソース・データベースに対し、256つまでのターゲット・データベースを指定することができます。
- バックアップシーケンスを使用してマスターデータベースがリストアできます。ただし、リストアした後、レプリケーションデータベースは終了します。ユーザーはレプリケーションデータベースを続くと、すべてのスレーブデータベースが新しいマスタデータベースを代わらなければなりません。つまり、ユーザーはマスタデータベースファイルをコピーして全部のスレーブファイルを代わります。
- 完全バックアップのために、レプリケーションサーバーは増分バックアップファイルを消去しない可能性があります。それで、長い期間に次の完全バックアップが実行されていない場合、大量のファイルが**DB_BKDIR**に存在しています。

- FILEデータ型のレプリケーションは、現在のところ使用できません。
- LONG VARCHARやLONG VARBINARYデータ型のようなカラムのBLOBデータをレプリケートする場合は、DB_BMODEを2(BACKUP-DATA-AND-BLOB)にセットし、TABLESPACEの作成の際にBACKUP BLOB ONのオプションをセットして下さい。

18 パフォーマンスのチューニング

DBMaster は、高度にチューニングすることができるデータベース・システムです。DBMaster をチューニングすることによって、そのパフォーマンスを各々満足できる水準まで改善することができます。この章は、チューニングの目的とチューニングの方法を説明します。また、パフォーマンスの分析方法を例示します。

18.1 チューニングの手順

DBMaster をチューニングする前に、パフォーマンス改良の目的を明確にします。互いに競合する目的があるかも知れません。その場合、どちらの目的が重要かを決定します。DBMaster をチューニングするための幾つかのポイントを以下に列記します。

- SQL文のパフォーマンスを改善する。
- データベース・アプリケーションのパフォーマンスを改善する。
- 同時実行処理のパフォーマンスを改善する。
- リソース使用を最適化する。

目的を定めたら、以下の手順で DBMaster をチューニングします。

- データベースのパフォーマンスを監視する。
- I/Oをチューニングする。

- メモリ割り当てをチューニングする。
- 同時実行処理をチューニングする。
- データベースのパフォーマンスを監視し、以前の統計と比較する。

各ステップのチューニングが、別のステップに影響を与えることがあります。上記の順にチューニングすることによって、他のチューニングへの影響を小さくすることができます。全てのチューニングを実行したら、DBMaster のパフォーマンスを監視し、最良のパフォーマンスが得られたかどうかを確かめます。

DBMaster をチューニングする前に、SQL 文の効率が良いかどうか、データベースのアプリケーションが良く設計されているかどうかを確かめます。非効率的な SQL 文や不出来なアプリケーションは、チューニングでは改善することができない悪影響をデータベース・パフォーマンスに及ぼします。効率的な SQL 文とアプリケーションについては、*「SQL 文と関数参照編」*と *「ODBC プログラマーガイド」*を参照してください。

18.2 データベースを監視する

この節は、データベースのリソース、操作、接続、同時実行性等の状態に関する情報をどのように監視するかを説明します。データベース接続を切断する方法もあわせて紹介します。

注 *ログインするためユーザーはDBA権限を持たなければなりません。*

監視表

データベースの状態は、4 つのシステムカタログ表、SYSINFO、SYSUSER、SYSLOCK、SYSWAIT に格納されています。

SYSINFO 表には、データベース・システム値が格納されています。例えば、総 DCCA サイズ、使用可能 DCCA サイズ、最大トランザクション数、ページバッファ数などです。また、システムアクション統計値、例えば、アクティブ・トランザクション数、起動トランザクション数、ロック要求数、セマフォ要求数、物理ディスク I/O 数、ジャーナルレコード I/O 数なども格

納します。SYSINFO 表を参照して、データベース・システムの状態を監視し、データベースのチューニングに役立てることができます。

SYSUSER 表は、データベースの接続情報、例えば、接続 ID、ユーザー名、ログイン名、ログイン IP アドレス、実行した DML オペレーション数を格納します。SYSUSER 表は、どのユーザーがデータベースを利用しているかを監視するために役立てることができます。

SYSLOCK 表は、オブジェクトのロック情報（ロックされたオブジェクト ID、ロック状態、ロック単位、ロックしている接続 ID）等を格納します。SYSLOCK 表は、どのオブジェクトがどの接続によってロックされているか、どのユーザーがどのオブジェクトをロックしているかを監視するために利用することができます。

SYSWAIT 表には、各接続の待ち状態情報（待機している接続 ID、待機させている接続 ID）があります。SYSWAIT 表を利用して、接続の同時実行性の状態を監視することができます。アイドル接続或いはデッド接続によってロックされているリソースを待機している場合、SYSWAIT 表を利用して、どの接続がオブジェクトをロックしているかを見つけることができます。さらにその接続を切断して、ロックされたリソースを開放することができます。

システムカタログ表は、通常の表と同様に参照します。

➲ **例**

SYSUSER 表をブラウズする：

```
dmSQL> SELECT * FROM SYSUSER;
```

これらのシステムカタログ表の詳細については、*システムカタログ参照*を参照してください。

接続を切断する

接続がリソースを抱えたまま長時間アイドル状態になっていたり、リソースを緊急に必要としたりする場合に、その接続を切断しなければならないことがあります。また、データベースを終了するためには、使用中の全て

の接続を切断しなければなりません。データベース接続を切断するときは、SYSUSER 表をブラウズして接続 ID を調べます。

➲ **例 1**

ユーザー*Eddie* を切断するために、*Eddie* の接続 ID を取得する：

```
dmSQL> SELECT CONNECTION_ID FROM SYSUSER WHERE USER_NAME = 'Eddie';

CONNECTION_ID
=============
352501
```

➲ **例 2**

取得した接続 ID を使って、接続を切断する：

```
dmSQL> KILL CONNECTION 352501;
```

18.3 I/O をチューニングする

DBMaster の大部分の時間はディスク I/O に費やされます。

以下を実行することによって､ディスク I/O のボトルネックを取り除くことができます。

- データ区分を決定する。
- ジャーナルファイル区分を決定する。
- データファイルとジャーナルファイルを別のディスクに分離する。
- ローデバイスを使用する。
- 自動拡張表領域に事前に領域を割り当てる。
- I/Oデーモンとチェックポイント・デーモンを使用する。

データ区分を決定する

表領域を使用してデータを区切り、全てのデータを一箇所に格納しないようにすることができます。適切な表領域を使用することによって、ストレージ管理機能や表の全検索は非常に効率よくなります。類似するデータが

ある小さい表は 1 つの表領域にまとめ、大きい表はその表専用の表領域に配置します。

ディスク・ストライピングを使用することによって、ディスク I/O の速度を改善することができます。ストライピングとは、連続するディスク・セクタを複数のディスクに仕切る方法です。この方法は、大きい表のデータを複数のディスクに分割するときに使用し、多くのプロセスが並行して同じデータファイルにアクセスするときに、ディスク競合が発生しないようにするために役立ちます。

ジャーナルファイル区分を決定する

DBMaster には、複数のジャーナルファイルを使用することができます。単一のジャーナルファイルは管理が簡単ですが、複数のジャーナルファイルを利用する場合も利点があります。複数のジャーナルファイルを使用すると、バックアップモードで走行中のデータベースの増分バックアップを取るときに、一杯になったジャーナルファイルのみバックアップすることによって増分バックアップのパフォーマンスを改善することができます。また、各ジャーナルファイルを別々のディスクに分散することによって、ディスク I/O のパフォーマンスを上げることができます。

ジャーナルファイルのサイズは、トランザクションのニーズを調べて決定します。但し、ジャーナルフルでバックアップを取るときは、ジャーナルファイルのサイズによってバックアップ間隔も影響を受けます。大きいジャーナルファイルは、バックアップ間隔を長くします。

ジャーナルファイルとデータファイルを分離する

ジャーナルファイルとデータファイルを別々のディスクに分離すると、それぞれのファイルに並行してアクセスすることができ、ディスク I/O のパフォーマンスを上げることができます。速度の違うディスクがあるときは、どちらのファイルを速いディスクに置くかを検討します。一般に、オンライン・トランザクション処理 (OLTP) アプリケーションを頻繁に使う場合、ジャーナルファイルを速いディスクに置き、意思決定支援システムのように長い問合せのアプリケーションを実行する場合は、データファイルを速いディスクに置きます。

ローデバイスを使用する

UNIX システムの DBMaster では、データファイルおよびジャーナルファイルを格納するローデバイスを構築することができます。DBMaster の優れたバッファ機構は、UNIX ファイルよりもローデバイスから読み/書きする方が速くします。ローデバイスの作成方法については、オペレーティング・システムのマニュアルを参照するか、システム管理者に相談してください。ローデバイスを使用する上で不便になるのは、DBMaster による表領域の自動拡張機能を利用できないことです。このため、ローデバイスを使用する場合は、十分に計画する必要があります。

自動拡張表領域に事前に領域を割り当てる

DBMaster には、表領域を自動的に拡張する機能があり、簡単に管理することができます。しかし、ページ拡張には時間を要するので、表領域の必要サイズを見積ることができる場合は、表領域の作成時にサイズを決めておく方がパフォーマンスは良くなります。ALTER FILE 文を使用して、作成後にファイルのページ数を増やすことも可能です。表領域のサイズを事前に割り当てることによって、全容量を使い尽くしたディスクにある表領域を拡張する際に発生するディスクフルエラーを避けることもできます。

I/Oとチェックポイント・デーモンを使う

I/Oデーモン

DBMaster は、定期的に最新の使用ページバッファからディスクに汚れたページを書き出すために I/O デーモンを使用します。これは、ページバッファにデータページをスワップする際に発生するオーバーヘッドを減らします。I/O デーモンを制御するために、**dmconfig.ini** ファイルのキーワードを使用します。

DB_IoSvr—I/O デーモンを作動、又は停止させます。I/O デーモンを作動させる場合はキーワードの値を 1 にセットし､停止させる場合は 0 にします。

➲ **例**

dmconfig.ini ファイル：

```
[MYDB]
…
DB_IOSVR = 1
```

MYDB データベースには、DCCA に 400(**DB_Nbufs**)ページバッファがあります。10 秒毎に、I/O デーモンは以下の処理を行います。

- 最新の使用ページバッファをスキャンします。
- スキャン実行中、汚れたページを回収します。
- 回収した汚れたページをディスクに書き込みます。

チェックポイント・デーモン

DBMaster には、I/O デーモンに基づいて、定期的にチェックポイントを取るチェックポイント・デーモンがあります。これは、コマンドの際や、ジャーナルファイルが一杯の時、或いはデータベースの起動と終了時に発生するチェックポイントを待機する時間を縮小します。チェックポイント・デーモンは、I/O デーモンの副機能です。チェックポイント・デーモンと I/O デーモンは同時に作動します。

チェックポイント・デーモンを使用するためには、キーワード **DB_IoSrv** で I/O デーモンを使用可能にして下さい。I/O デーモンを ON にすると、データベースの起動後に自動的に 10 分毎にチェックポイントが取られます。

➲ **例**

dmconfig.ini でチェックポイント・デーモンを起動し、I/O デーモンを止める：

```
[MYDB]
…
DB_IOSVR = 1                    ; I/O とチェックポイントデーモンを使用可能にする
```

I/O デーモンとチェックポイント・デーモンには、I/O リソースが費やされます。データベース・サーバー起動後、I/O デーモンとチェックポイント・デーモンによって生成されたエラーメッセージは、ERROR.LOG ファイルに書き込まれます。

18.4 メモリ割り当てをチューニングする

DBMaster は、一時的な情報をメモリバッファに格納し、永続的な情報をディスクに格納します。メモリはディスクよりも速くデータを検索することができるので、メモリバッファからデータを取得することができればパフォーマンスが上がります。DBMaster の各メモリ構造のサイズは、データベースのパフォーマンスに影響しますが、メモリがパフォーマンスに関係するのは、十分なメモリが無いときだけです。

この節では、データベースが使用するメモリのチューニングに焦点を当てます。必要な DCCA サイズをどのように計算するか、或いはページバッファ、ジャーナルバッファ、システム制御域をどのように監視し十分なメモリを割当てるかを説明します。

➲ **以下の順にメモリをチューニングすることによって、最高のパフォーマンスを得ることができます：**

1. オペレーティング・システムをチューニングする。
2. DCCAメモリサイズをチューニングする。
3. ページバッファをチューニングする。
4. ジャーナルバッファをチューニングする。
5. SCAをチューニングする。

DBMaster のメモリ必要量は、使用するアプリケーションによって変わります。アプリケーション・プログラムと SQL 文をチューニングした後に、メモリ割り当てをチューニングします。

オペレーティング・システムをチューニングする

オペレーティング・システムをチューニングし、メモリスワップを少なくすると共に、システムが効率的にスムーズに走行することを確かめます。

物理メモリとディスク上の仮想メモリファイル間のメモリスワップは、大量の時間を必要とします。走行中のプロセス用に十分な物理メモリがあることが重要です。オペレーティング・システムの状態は、オペレーティン

グ・システムのユーティリティを使用して測定することができます。ページスワップ率が極端に高い場合は、物理メモリが十分ではないことを示しています。このような場合は、不用のプロセスを削除するか、メモリをシステムに追加します。

DCCAメモリをチューニングする

*データベース通信制御域(DCCA)*は、DBMaster サーバーによって割当てられる共有メモリの集まりです。DBMaster を起動するたびに、DCCA が割り当てられ、初期化されます。

UNIX のクライアント/サーバー・モデルの場合は、UNIX の共有メモリ・プールに DCCA を割り当てます。DCCA サイズが、オペレーティング・システムに認められている共有メモリの最大サイズを超えていないことを確かめます。DCCA の必要サイズがオペレーティング・システムの限界より大きいときは、共有メモリの最大サイズを拡張します。方法については、オペレーティング・システムの管理マニュアルを参照してください。

DBMaster の共有メモリサイズは 64bit マシンで 2G ページ（或いは 2G * PAGE SIZE バイト）。でも、32bit マシンで 2G バイトです。これはページバッファーサイズ、ジャーナルバッファーサイズと SCA サイズを含みます。

64bitLinux OS で必要な共有メモリは 2G を超えるとまた十分な RAM があると、ユーザーは **shmmax** サイズを設置しなければなりません。***/etc/sysctl.conf*** がエディットできるまたは **kernel.shmmax = n** のセットを通じて最大共有メモリサイズを指定します。ユーザーは自分で***/etc/sysctl.conf***.の変更権を確立しなければなりません。

➲ **例**

```
kernel.shmmax = 8405194752.
```

DCCAの環境を設定する

DCCA には、プロセス通信制御ブロック、同時実行制御ブロックと、データページ、ジャーナルブロック、カタログ用のキャッシュバッファがあります。DBMaster は、同時実行制御ブロックと DBMaster プロセスの通信状態

を DCCA に保持します。DBMaster プロセスは、DCCA のキャッシュバッファを経由して、共通のディスクデータにアクセスします。

DCCA の各要素のサイズは、データベースの起動前に、**dmconfig.ini** のパラメータに適切な値を与えて設定します。

➲ **例 1**

dmconfig.ini ファイルで DCCA を環境設定する：

```
DB NBUFS = 200
DB NJNLB = 50
DB_SCASZ = 50
```

DB_NBufs はページバッファ数（ページサイズが 4KB にすると 4096 バイト/バッファ）、**DB_NJnlB** はジャーナルバッファ数（ページサイズが 4KB にすると 4096 バイト/バッファ）、**DB_ScaSZ** は SCA のページ数（4096 バイト/ページページサイズが 4KB にすると）を指定します。DBMaster は、データベースを起動するときにのみ、これらの DCCA パラメータを読みます。パラメータの値を変更する場合は、データベースを終了し、**dmconfig.ini** のパラメータの値を変更し、データベースを再起動します。これらのパラメータについては、**dmconfig.ini** のキーワードを参照してください。

DCCA に割当てられるメモリは、**DB_NBufs**、**DB_NJnlB**、**DB_ScaSZ** パラメータのサイズの合計です。

➲ **例 2**

ページサイズが 4KB にすると DCCA の総サイズを計算する：

```
DCCA の合計サイズ  = (200 + 50 + 50) * 4 KB
                  = 1200 KB
```

十分なDCCA物理メモリを割り当てる

DCCA は DBMaster プロセスが最も頻繁にアクセスするリソースです。従って、オペレーティング・システムが頻繁に DCCA をディスクにスワップしないだけの十分な物理メモリが必要になります。さもないと、データベースのパフォーマンスは非常に悪くなります。オペレーティング・システムのユーティリティを使用して、ページスワップ率を測定することができます。

➲ 例

SYSINFO システム表から DCCA に割り当てられたメモリを表示する：

```
dmSQL> select INFO, VALUE from SYSINFO where INFO = 'DCCA_SIZE'
or INFO = 'FREE_DCCA_SIZE';

              INFO                            VALUE
============================== ==============================
DCCA SIZE                      1228800
FREE DCCA SIZE                 189024

2 rows selected
```

DCCA_SIZE—DCCA メモリのバイト数

FREE_DCCA_SIZE—DCCA に残っている未使用メモリのバイト数

DCCA の未使用メモリは、ロック制御ブロックのような動的制御ブロックが使用するために確保されています。

一般的にシステム・パフォーマンスは、バッファ数が大きい方が良くなります。但し、物理メモリに比べて DCCA が大きすぎると、システムパフォーマンスは悪くなります。このため、十分なメモリを DCCA に割り当てるとともに、物理メモリに合った DCCA サイズにする必要があります。

ページバッファキャッシュをチューニングする

DBMaster は、ページバッファのために共有メモリを使用します。バッファキャッシュは、データアクセスと同時実行制御を高速にします。ページバッファ数は自動的に環境設定されます。**dmconfig.ini** ファイルの **DB_Nbufs** キーワードを 0 に設定すると、ページバッファ数を自動的にセットするようにします。DBMaster に物理メモリの使用を検出させるシステムのページバッファ数を動的に調節します。その数は、Windows 95/98 では 500 ページ以上、Windows NT/2000/や Unix で 2000 ページ以上です。システムの物理メモリ使用を検出できない場合、最小量が割り当てられます。

適切なページバッファ・サイズに調整することで、問合せのパフォーマンスが大幅に向上します。以下の小節で、バッファキャッシュのパフォーマンスを監視し、バッファヒット率を計算する方法について説明します。

➲ **バッファキャッシュのパフォーマンスを改善する：**

1. スキーマ・オブジェクトの統計値を更新します。
2. 大きい表にはNOCACHEをセットします。
3. クラスタが劣化した索引のデータを再編成します。
4. キャッシュバッファを拡張します。
5. チェックポイントの影響を減らします。

ページバッファキャッシュのパフォーマンスを監視する

バッファキャッシュのアクセス統計値は、SYSINFO システム表に格納されます。

➲ **例**

以下 SQL の文を使って、バッファ・キャッシュ値を取得する：

```
dmSQL> select INFO, VALUE from SYSINFO where INFO = 'NUM_PAGE_BUF';

              INFO                            VALUE
============================== ==============================
NUM_PAGE_BUF                   1000

1 rows selected

dmSQL> select INFO, VALUE from SYSINFO where INFO = 'NUM_PHYSICAL_READ'
or INFO = 'NUM_LOGICAL_READ'
or INFO = 'NUM_PHYSICAL_WRITE'
or INFO = 'NUM_LOGICAL_WRITE';

              INFO                            VALUE
============================== ==============================
NUM PHYSICAL READ              13207
NUM LOGICAL READ               331595
NUM PHYSICAL WRITE             7361
NUM LOGICAL WRITE              127423

4 rows selected
```

NUM_PAGE_BUF—バッファキャッシュのページ数

NUM_PHYSICAL_READ—ディスクから読んだページ数

NUM_LOGICAL_READ—バッファキャッシュから読んだページ数

NUM_PHYSICAL_WRITE—ディスクに書いたページ数

NUM_LOGICAL_WRITE—バッファキャッシュに書いたページ数

ページバッファのヒット率は、以下の式で計算します：

$$\text{読み出しヒット率} = 1 - \left(\frac{\text{NUM\_PHYSICAL\_READ}}{\text{NUM\_LOGICAL\_READ}}\right)$$

$$\text{書き込みヒット率} = 1 - \left(\frac{\text{NUM\_PHYSICAL\_WRITE}}{\text{NUM\_LOGICAL\_WRITE}}\right)$$

上記の例のヒット率は次のように計算します：

$$\text{読み出しヒット率} = 1 - \left(\frac{13207}{331595}\right)$$
$$= 0.9600$$
$$= 96.0\%$$

$$\text{書き込みヒット率} = 1 - \left(\frac{7361}{127423}\right)$$
$$= 0.942$$
$$= 94.2\%$$

読み込み/書き出しのヒット率から、バッファキャッシュのパフォーマンスをどのように改善するかを決めます。ヒット率が低すぎる場合は、以下の小節で説明する方法を使用して DBMaster をチューニングします。

常にヒット率が高い場合、例えば 99%以上のときは、キャッシュは十分な大きさであると言えます。この場合、キャッシュサイズを減らしてアプリケーションにメモリを空けてみることができます。この変更の前後には、キャッシュ・パフォーマンスを監視して、良いパフォーマンスが維持されていることを確認します。

古い統計値

読み込み/書き出しのヒット率が低すぎるのは、スキーマ・オブジェクト（表、索引、カラム）の統計値が古いためかもしれません。誤った統計値は、非効率的なプランで SQL 文を最適化する原因になります。統計を更新した後に、大量のデータをデータベースに挿入した場合は、再度統計値を更新する必要があります。

➲ **例 1**

全てのスキーマ・オブジェクトの統計値を更新する：

```
dmSQL> UPDATTE STATISTICS;
```

データベースが極端に大きい場合、全てのスキーマ・オブジェクトの統計値を更新するには、非常に時間がかかります。代わりに、特定のスキーマ・オブジェクトの統計値のみサンプリング率を指定して更新することができます。

➲ **例 2**

特定のスキーマ・オブジェクトを更新する：

```
dmSQL> UPDATE STATISTICS table1, table2, user1.table3 SAMPLE = 30;
```

スキーマ・オブジェクトの統計値を更新したら、「ページバッファキャッシュのパフォーマンスを監視する」で述べた方法を使用して再度ページバッファキャッシュのパフォーマンスを監視します。

キャッシュをスワップアウトする

DBMaster は、*Least Recently Used（LRU）*規則を用いてスワップするページバッファを決定します。LRU は、ページバッファの中の頻繁にアクセスのあるページを残し、アクセス頻度の低いページをスワップします。但し、大きい表を検索すると、1 つの表スキャンのためだけに、全てのページバッファがスワップアウトされてしまうことがあります。

例えば、データベースに 200 ページのバッファがある場合、250 ページの表を参照すると、250 ページ全体がページバッファに読み込まれ、これまで頻繁に使用された 200 ページが廃棄されるかもしれません。最悪の場合、全検索の後に他のデータにアクセスすると、ディスクから 200 ページを読見

込むかもしれません。ところが、表を NOCACHE（非キャッシュ）モードにしておくと、全検索で読み込まれたページは LRU 列の終わりに置かれ、最も頻繁に使用された 200 ページの内 199 ページは、そのままバッファキャッシュに残ります。

通常、ページバッファ数を超えるページ数のある表は、NOCACHE モードにしておく必要があります。めったに使用しない表や、表のページ数がページバッファ数とほぼ同じ表も NOCACHE モードに設定します。

➲ **例 1**

表のページ数とキャッシュモードを調べる：

```
dmSQL> select TABLE OWNER, TABLE NAME, NUM PAGE, CACHEMODE from SYSTEM.SYSTABLE where
TABLE OWNER != 'SYSTEM';

TABLE OWNER     TABLE NAME      NUM PAGE   CACHEMODE
=========== ================== =========== =========
       BOSS             salary           5 T
        MIS              asset          45 T
        MIS         department           3 T
        MIS           employee          29 T
        MIS           worktime         450 T
      TRADE           customer         350 T
      TRADE          inventory         167 T
      TRADE              order         112 T
      TRADE        transaction        1345 F

9 rows selected
```

NUM_PAGE—表のページ数

CACHEMODE—表のキャッシュモード、T はキャッシュを意味し、F は NO**CACHE** を意味します。

この例では、表 **TRADE.transaction** が NOCACHE に設定されており、他の表はキャッシュします。ページバッファが 200 しかなければ、**MIS.worktime** と **TRADE.customer** 表も NOCACHE に設定すべきです。**TRADE.order** と **TRADE.inventory** 表を稀にしか使用しない場合は、同様に NOCACHE に設定します。

➲ 例 2

表のキャッシュモードを NOCACHE にする：:

```
dmSQL> ALTER TABLE MIS.worktime SET NOCACHE ON;
```

表に索引が作成されていないか、問合せの述語が索引以外のカラムを参照する場合も、DBMaster は表を全検索します。全検索を避けるためには、可能な限り効率的な SQL 文を書き、索引カラムを使用するようにします。

クラスタ性能劣化のレコード

索引クラスタは、索引キー順に大量のレコードを読み取ったり検索条件で索引カラムを参照したりする時、バッファキャッシュのパフォーマンスに影響する重要な要因になります。

➲ 例 1

tb_customer 表の全カラムを検索し、主キー**custid** でソートする：

```
dmSQL> SELECT * FROM CUSTOMER ORDER BY tb_customer ;
```

customer は 3500 件のレコードが 350 ページに分散して格納されており、システムには 200 ページバッファがあるとします。レコードが ***custid*** によってクラスタ化されており、クラスタが非常に良い（全てのページでキー順に並べられている）と、350 ページをディスクから読むだけで済みます。しかしクラスタが悪い（同じページにキー順のレコードがない）と、最悪の場合、3500 ページを読む（各レコードをディスクから読む）ことになります。索引クラスタの状態を調べるときは、最初に表の統計値を更新しておきます。

➲ 例 2

表 **tb_customer** のカラム **custid** に索引 **custid_index** のクラスタカウントを調べる：

```
dmSQL> SELECT CLSTR COUNT FROM SYSTEM.SYSINDEX
                          WHERE TABLE OWNER = 'TRADE'
                          AND TABLE NAME = 'customer'
                          AND INDEX_NAME = 'custid_index';
```

結果は以下のようになります。

```
CLSTR_COUNT
```

```
==========
        385

1 rows selected
```

CLSTR_COUNT—クラスタカウントは、バッファをほとんど使用せずに完全に索引検索で読み取ることができるデータページ数を表します。上記の例では、customer 表全体を検索し custid のカラム順に並べるには、385 ページのみディスク読み込みを行います。

➲ 例 3

ページ数と行数を取り出す：

```
dmSQL> select NUM PAGE,NUM ROW from SYSTEM.SYSTABLE
                                where TABLE OWNER = 'TRADE'
                                and TABLE_NAME = 'customer';
```

結果は以下のようになります：

```
NUM_PAGE    NUM_ROW
========== ==========
        350        4375

1 rows selected
```

NUM_PAGE—表のページ数

NUM_ROW—表のレコード数

CLSTR_COUNT、NUM_PAGE、NUM_ROW を用いて、以下の式でクラスタ率を見積ることができます：

$$\text{クラスタ率} = \frac{(\text{CLSTR\_COUNT} - \text{NUM\_PAGE})}{\text{NUM\_ROW}}$$

上記の例では、クラスタ率は次のようになります。

$$\text{クラスタ率} = \frac{(385-350)}{4375}$$
$$= 0.008$$
$$= 0.8\%$$

クラスタ率は 0～100%の間です。CLSTR_COUNT が NUM_PAGE よりも僅かに小さいときは、クラスタ率 0 として取り扱います。クラスタ率 0 は、データが索引で完全にクラスタ化されていることを意味します。クラスタ率が高すぎる、例えば 20%（高いかどうかは表サイズ、平均レコードサイズ等により変わります）以上の場合は、索引クラスタが劣化しています。索引クラスタが劣化していると、索引検索がより適当であると思えるような SQL 文でさえ、最適化によって表が全検索されることになるかも知れません。

➲ **頻繁に使用する索引のクラスタを改善する：**

1. 表の全データをアンロードします（索引順に）。
2. アンロードしたデータの順序をそろえます。
3. 表の索引を削除します。
4. 表の全データを削除します。
5. 表にデータをリロードします。
6. 表の索引を再編成します。

データのリロードした後は、索引は完全にクラスタ化されるはずです。ここで、表は一つの索引でのみクラスタ化される点に注意して下さい。表に複数の索引が編成されているときは、最も重要な索引でクラスタ化します。一般的に最も重要の索引は、主キーです。データのアンロードとリロードには多くの時間とストレージを必要とするので、索引クラスタをチューニングするのは、頻繁に参照される非常に大きい表だけにします。

データページバッファの不足

データベースアクセスに十分なデータページバッファが割当てられていないときは、DCCA にページバッファを追加します。

➲ ページバッファ数を変更する：

1. データベース・サーバーを停止します。
2. dmconfig.iniのDB_NBufsの値を大きくします。
3. データベースを再起動します。

データバッファを拡張したときは、一定期間データベースを走行し、再度バッファキャッシュのパフォーマンスを監視します。バッファヒット率が上がれば、バッファページの追加によってパフォーマンスが改善されたことになります。そうでなければ、更にバッファキャッシュにページを追加するか、システムパフォーマンスを低下させる他の理由をチェックします。

チェックポイントが頻繁に発生する

書き込みヒット率が読み出しヒット率に比べて低すぎる場合、チェックポイントがあまりにも頻繁に取られるかもしれません。

チェックポイントを取ると、全ての汚れたページバッファはディスクに書き出されます。チェックポイントは大量の CPU 時間を必要とするので、チェックポイント・デーモンに一定間隔でチェックポイントを実行させるようにします。定期的にチェックポイントを取るもう 1 つの利点は、システム障害後にデータベースを再起動してリカバリする時間を短縮することです。

定期的にチェックポイントを取る他に、NON-BACKUP モードで走行中に使用可能なジャーナル領域が無くなった時、あるいは BACKUP モードで走行中に増分バックアップが取られる時に、自動的にチェックポイントが取られます。チェックポイントを取る間隔を長くするためには、ジャーナルサイズを大きくします。

➲ 例

下記の例では、実行されたチェックポイントの数を説明します。

```
dmSQL> select INFO, VALUE from SYSINFO where INFO = 'NUM_CHECKPOINT';

             INFO                            VALUE
============================== ==============================
NUM_CHECKPOINT                 26
```

```
1 rows selected
```

再度キャッシュバッファのパフォーマンスを監視する

上述の方法でシステムをチューニングした後、キャッシュバッファのパフォーマンスを監視します。

➲ **キャッシュバッファのパフォーマンスを監視する：**

1. データベース情報が安定した状態になるまで、一定期間データベースを走行させます。
2. 以下のSQL文でSYSINFOシステム表の統計値をリセットします。

```
dmSQL> SET SYSINFO CLEAR;
```

3. しばらくにデータベースを実行します。
4. SYSINFO表から読み込み/書き出し数を取得してヒット率をチェックします。

ジャーナル・バッファをチューニングする

ジャーナル・バッファは、最新の使用済みジャーナルブロックを格納します。ジャーナル・バッファが十分にあれば、データ更新時にジャーナルブロックをディスクに書き込む際と、トランザクションをロールバックする時にジャーナルブロックをディスクから読む出す時間が減少します。

多くのレコードを修正（挿入、削除、更新）する長いトランザクションを実行することが少ない場合は、この節を読み飛ばしてもかまいません。そうでない場合は、ジャーナルバッファがシステムに十分あるかを調べます。最適なジャーナルバッファ数は、同時に走行する最長のトランザクションに必要なジャーナルブロックの合計です。

➲ **ジャーナル・バッファ数を見積る：**

1. データベースのアクティブ・ユーザーが一人であることを確認します。
2. 下記のSQL文でSYSINFO表にあるカウンタをクリアします。

```
dmSQL> SET SYSINFO CLEAR;
```

3. ほとんどのレコードを更新するトランザクションを実行します。

4. 次のSQL文を実行して使用したジャーナルブロック数を調べます。

```
dmSQL> select INFO, VALUE from SYSINFO where INFO = 'NUM JNL BLK WRITE';

             INFO                           VALUE
============================== ==============================
NUM JNL BLK WRITE              626

1 rows selected
```

注 *NUM_JNL_BLK_WRITE—このトランザクションが使用したブロック数。この例で使用するジャーナルブロックのサイズは512バイトです。上記の例の場合、約80ページのジャーナルバッファが必要になります（1ページは4KBにします）。*

ジャーナルバッファの使用状況を調べるもう一つの尺度は、ジャーナルバッファのフラッシュ率です。ジャーナルバッファのフラッシュ率は、ジャーナルを書くときにディスクに掃き出したジャーナルバッファの割合です。ジャーナルバッファのフラッシュ率が高すぎる（例えば、50%以上）場合は、ジャーナルバッファ数を増やす必要があります。

➲ 例

ジャーナルバッファのフラッシュ率を計算する：

```
dmSQL> select INFO, VALUE from SYSINFO where INFO = 'NUM JNL BLK WRITE'
                                        or INFO = 'NUM JNL FRC WRITE ';

             INFO                           VALUE
============================== ==============================
NUM JNL BLK WRITE              41438
NUM JNL FRC WRITE              159

2 rows selected
```

NUM_JNL_BLK_WRITE—バッファに書いたジャーナルブロック数。

NUM_JNL_FRC_WRITE—ジャーナルバッファを強制的にディスクに書いた回数。

DB_NJnlB を 50 ページ（400 ジャーナルバッファ）とします。以下の例では、ジャーナルフラッシュ率（0.65）が少し高すぎます。ジャーナルバッファを追加してジャーナルバッファのパフォーマンスを改善すべきです。

$$\text{ジャーナルフラッシュ率} = \frac{(\text{NUM\_JNL\_BLK\_WRITE}/\text{NUM\_JNL\_FRC\_WRITE})}{(\text{DB\_NJNLB} \times 8)}$$

$$= \frac{(41438/159)}{(50 \times 8)}$$

$$= 0.65$$

システム制御域(SCA)をチューニングする

DCCA のキャッシュバッファや制御ブロック（セッション情報とトランザクション情報のような）は、データベースの起動時に固定サイズが割り当てられます。しかし、同時実行制御ブロックは、データベースの起動中に動的に割り当てられます。これらの制御ブロックは DCCA に割り当てられ、サイズは DB_ScaSz に指定されます。

SCA 領域に動的なメモリを割り当てることができない時には、エラーメッセージ「データベースが要求した共有メモリがデータベース起動時の設定を超えています」が表示されます。通常このエラーは、あまりにも多くのロックを使用する長いトランザクションに起因します。この現象が頻発するようならば、以下に例示する方法で解決することができます。

長いトランザクションを避ける

長いトランザクションは、多くのロック制御ブロックとジャーナルブロックを占有します。長いトランザクションが実行しているときに、上記エラーが発生する場合は、そのトランザクションを複数の短いトランザクションに分けることができないかを検討します。

大きい表の多数のロックを避ける

索引スキャンを使用して、大きい表から多くのレコードを選択すると、多くのロックリソースが使用されます。トランザクションで使用するロックリソースを減らすためには、表スキャンを開始する前に、ロックモードを上げます。

例えば、表の設定ロックモードが行（ROW）ならば、ロックモードをページあるいは表にします。これによって、リソースの使用が少なくなりますが、同時実行性はある程度低下します。

SCAサイズを大きくする

上記 2 つの条件に当てはまらない場合は、SCA のサイズを大きくします。**dmconfig.ini** ファイルの **DB_ScaSz** の値を設定し直し、データベースを再起動します。

カタログキャッシュをチューニングする

カタログキャッシュは SCA に格納されています。スキーマ・オブジェクトを変更することが殆ど無い場合は、データ・ディクショナリのターボモードを ON にする（**dmconfig.ini** で **DB_Turbo**=1 にする）ことができます。ターボモードを ON にすると、DBMaster はカタログキャッシュの寿命を長くし、オンライン・トランザクション処理（OLTP）プログラムのパフォーマンスを改善します。

18.5 同時実行処理をチューニングする

リソース競合は、マルチユーザーのデータベースシステムで、2 つ以上のプロセスが同時に並行して同じデータベースリソースにアクセスしようとするときに発生します。2 つ以上のプロセスが互いに一方のリソースを待つことによって、デッドロックと呼ばれる状況が発生することもあります。リソース競合は、プロセスのデータベースアクセスを待たせるのでシステムパフォーマンスを低下させます。

次の方法でリソース競合を検出し減らすことができます。

- ロック競合を減らす。
- プロセス数を制限する。
- CPUアフィニティーを設定する。
- Job優先級を設定する。

ロック競合を減らす

DBMaster プロセスは、データベースからデータをアクセスするときに、自動的に対象オブジェクト（行、ページ、表）をロックします。2 つのプロセスが同じオブジェクトをロックすると、一方は待機させられます。複数のプロセスが互いに他のプロセスのロック解除を待つと、デッドロックが発生します。デッドロックが発生すると、デッドロックを発生させた最後のトランザクションがロールバックされ犠牲になります。デッドロックはシステムパフォーマンスを低下させるので、ロック統計を監視しデッドロックを回避するようにします。

➲ 例

デッドロックの状態を確認する：

```
dmSQL> select INFO, VALUE from SYSINFO where INFO = 'NUM LOCK REQUEST'
                                          or INFO = 'NUM DEADLOCK'
                                          or INFO = 'NUM STARTED TRANX';

              INFO                              VALUE
================================ ================================
NUM STARTED TRANX                9287
NUM LOCK REQUEST                 772967
NUM DEADLOCK                     181

3 rows selected
```

NUM_LOCK_REQUEST—ロック要求回数

NUM_DEADLOCK—デッドロックの発生回数

NUM_STARTED_TRANX—実行したトランザクション数

この例では、平均して、51（9287/181）トランザクションに 1 回デッドロックが発生し、1 トランザクション当たり約 83（772967/9287）のロックを要求しています。

デッドロックの頻度が高い場合は、スキーマ設計、SQL 文、アプリケーションを調査します。表の初期設定ロックモードを低く（行ロックのように）

すると、ロック競合を減らすことはできますが、より多くのロックリソースを必要とするようになります。

もう一つの方法は、データ検索後にデータが同じ状態を保つ必要が無い場合に、ブラウズモードを ON にして表を読むことです。この方法は、データを更新せずに単にデータを見る、あるいはデータを計算するときに有効です。ブラウズモードは、要求されたデータのスナップショットを提供し、データを読み込むとすぐにロックを解除します。同時実行性が高くなり、ロックリソースの消費を少なくする利点があります。

プロセス数を制限する

DBMaster では、1 サーバーあたり 4800 まで同時接続することができます。メモリ、CPU パワーのようなサーバー・リソースが十分でない場合、リソースの競合を避けるために、最大数を制限します。環境設定パラメータ **DB_MaxCo** は、データベースの最大接続数に影響します。

データベースの作成時、ジャーナル・ファイルは特定の接続数にフォーマットされています。ジャーナル・ファイルを、各接続用にトランザクション情報配列を確保できるようにする必要があります。ジャーナル・ファイルに応じて使用することができる接続数の数は、*ハード接続数*とも呼ばれています。この値は、データベース作成時に、**DB_MaxCo** の値によって決定します。*ハード接続数*の最小値が 240、最大値が 4840 で、必ず 40 の倍数でなければなりません。**DB_MaxCo** を 40 の倍数でない数値に設定した場合、*ハード接続数*は 40 の倍数に丸められます。ハード接続数は、ジャーナル・ファイルの限界です。それゆえ、その値を変更すると、**DB_MAXCO** の再設定及びデータベースを新規ジャーナル・モード（**DB_SMODE**=2）で再起動する必要があります。

*ハード接続数*を計算するために使用される式は下記のようになります。

ハード接続数 = (DB_MAXCO + 保留された接続数 + 40 - 1)/40 * 40

現在 DBMaster はデーモン、管理用などの内部接続をサポートするために 20 個の保留接続数を持っています。

ハード接続数は、直接 DCCA のサイズに影響を与えません。これは、*ソフト接続数*と呼ばれる値によって決定します。大きい接続数は、サーバとク

ライアントの両方でより多くのメモリが必要であることを意味します。従いまして、データベースの*ハード接続数*が多くても、ユーザーはメモリを節約するために小さいソフト接続数を使用してデータベースを起動することができます。データベース起動時に、*ソフト接続数*が DB_MaxCo の値によって決定されます。ソフト接続数は、DCCA がサポートする接続数と、結果として DCCA のメモリ使用を決定します。ソフト接続数は、ハード接続数より少ないか同じです。ソフト接続数を変更するためには、**DB_MaxCo** を変更した後、データベースを通常に再起動します。

*ソフト接続数*を計算するために使用される式は下記のようになります。

ソフト接続数 = DB_MAXCO + 保留された接続数

➲ **例 1**

次の環境設定ファイルでは、**DB1** のハード接続数は 240、**DB2** は 1120 です。

```
[DB1]
DB_MaxCo = 50     ;; ハード接続数は 240
                  ;; ソフト接続数は 70
[DB2]
DB_MaxCo = 1100   ;; ハード接続数は 1120
                  ;; ソフト接続数は 1120
```

➲ **例 2**

データベース起動後、**DB1** の新しいハード接続数は 320 になります。

```
[DB1]
DB_SMode = 2              ;; 新規ジャーナル・モードで起動
DB_MaxCo = 280            ;; 新しいハード接続数は 320
```

➲ **例 3**

DB2 が例 1 のように作成されていると想定した場合、**dmconfig.ini** ファイルへ次のように入力すると、ハード接続数は 1120、ソフト接続数は 40 になります。

```
[DB2]
DB_SMode = 1              ;; ノーマル起動モード
DB_MaxCo = 20             ;; 新しいソフト接続数は 40
```

CPUアフィニティーを設定する

マルチタスクの性能を高めるため、操作システムはプロセスとスレッドを異なる CPU にディスパッチします。操作システムから考えて有効でも、各プロセッサーキャッシュがデータを繰り返してロードして、高いシステムロードのせいで DBMaster 性能を低下にする可能性があります。もし各プロセッサーは最後に実行する CPU に実行すると、DBMaster 性能が良くなります。ユーザーはシステムユーザーのクエリを通じてプロセスアフィニティーを取れます。システムストアドプロシージャ STEAFFINITY を使用して新しいアフィニティーマスクを設定します。この方法で、プロセスは指定した CPU にのみ実行されます。

古い操作システムと比べて、CPU アフィニティーは Linux 2.6、Windows NT と Windows 98 のような現代操作システムの特性を持ちます。二つの CPU 種類があります：ユーザー変更できないソフトアフィニティー、ユーザー変更できるハードアフィニティー。

CPU アフィニティーはアフィニティーマスクによって定義されます。アフィニティーマスクはビットベクトルで、一つのビットは一つのプロセッサーを代表します。DBMaster はアフィニティーマスクを char(64)として定義しますので、最高は 64 CPU を設定します。

➲ **例 1**

8-CPU システムのアフィニティーマスク値。（高い位置に連続的な 0 は無視になります）。

10 進法の値	バイナリビットマスク	実行 CPU
1	'1'	0
3	'11'	0 と 1
7	'111'	0、1 と 2
15	'1111'	0、1、2 と 3
31	'11111'	0、1、2、3 と 4

63	'111111'	0、1、2、3、4 と 5
127	'1111111'	0、1、2、3、4、5 と 6
255	'11111111'	0、1、2、3、4、5、6 と 7

GETCPUNUMBER、SETAFFINITY を使用して、ユーザーはランタイムに DBMaster を再起動する必要が無く、現在のシステム状態を取れ、接続の CPU アフィニティーを設定することができます。

GETCPUNUMBER の原型は以下のようになります：

```
GETCPUNUMBER (INT CPU_NUMBER OUTPUT)
```

CPU_NUMBER：出力バラメータ、ロジックプロセッサの数

SETAFFINITY の原型は以下のようになります：

```
SETAFFINITY (INT CONNECTION_ID INPUT,  CHAR(64) AFFINITY_MASK INPUT)
```

CONNECTION_ID：入力引数、接続の ID 或いはサーバーです。ユーザーは"select connection_id from sysuser"またはシステムモニターをチェックすることを通じでこれを取ります。windows では、これはスレッドの ID を示して、Unix-like システムでは、これはプロセス ID を示します。

AFFINITY_MASK：入力引数、CPU アフィニティマスクです。有効なアフィニティマスクは 1 或いは 0 から構成されます。'1' は CPU が有効だという意味で、 '0'は無効だという意味です。

➲ 例 2

ユーザーはアフィニティマスクを設定する前、システム情報をとる必要があります、例えば、サーバーの CPU の数、接続の CPU 用法、正確的なアフィニティマスク。

GETCPUNUMBER を呼び出して CPU の数を取ります：

```
dmSQL> call GETCPUNUMBER(?);
```

接続の CPU 用法、正確的なアフィニティマスクを取ります：

```
dmSQL> select connection_id, affinity_mask, priority_level, cpu_usage from sysuser;
```

システムユーザーに CPU アフィニティを設定します。CPU 0 と 1 にこの接続が実行することが許せます：

```
dmSQL> select connection id , user name from sysuser;

    CONNECT*              USER NAME
===============   ======================
30420                  BACKUP_SERVER
30418                  SYSADM

2 rows selected
dmSQL> call setaffinity(30418,'11');
```

注 *SysadmのみSETAFFINITYをコールすることができます。*

接続にクエリシステムユーザーを通じてCPU アフィニティを取ります：

```
dmSQL> select affinity_mask from sysuser where connection_id = ?;
```

JOB優先級を設定する

プロセッサとスレッドの優先級は、マルチ-タクス操作システムの基本的な機能です。スケジューラは、システムのロードをチェックし、必要なら、ジョブの優先級レベルを変更します。DBMaster は、システム全体のパフォーマンスを向上させるために、ユーザーに一つの接続の優先権を設定させることができます。例えば、長い時間がかかるジョブは CPU タイムの大部分を占める、その結果として、他の接続は長い時間を待つ、他の仕方がありません。ユーザーは長期ジョブの優先級を減少する場合、他の接続はより多くの CPU タイムを取得して実行できます。そして、システムの全体パフォーマンスを向上させることができます。ユーザーは、接続のパフォーマンスを向上するために、いくつかの重要の接続に新しい値を設定することができ、その間、他の接続は低いパフォーマンスになっています。

GETCPUNUMBER と SETAFFINITY システムストアドプロシージャを使用して、ユーザーは今のシステム声明を取ることができて、ランタイムに DBMaster を再起動する必要が無く、 接続の CPU アフィニティを設定できます。

SETPRIORITY の原型は以下のようになります：

```
SETPRIORITY (INT CONNECTION_ID INPUT,INT PRIORITY_LEVEL INPUT)
```

CONNECTION_ID：入力引数、接続の ID 或いはサーバーです。ユーザーは"select connection_id from sysuser"またはシステムモニターをチェックすることを通じでこれを取ります。windows では、これはスレッドの ID を示して、Unix-like システムではこれはプロセス ID を示します。

PRIORITY_LEVE：入力引数、五つのレベルがあります。ディフォルト優先レベルは 3 です。有効な優先級は'1'、'2'、'3'、'4'と'5'で、'1'は最低レベル、'2'は低いレベル、'3'は普通レベル、'4'は高いレベル、'5'は最高優先級という意味です。

注　*Linux にroot権限が必要ですので、ユーザーは高い優先級を設定することができません。だから、Linux に低い優先が設定できます。でも、Windows に制限がありません。*

➲ 例

ユーザーはアフィニティマスクを設定する前、システム情報をとる必要があります、例えば、サーバーの CPU の数、接続の CPU 用法、優先級。

GETCPUNUMBER を呼び出して CPU の数を取ります：

```
dmSQL> call GETCPUNUMBER(?);
```

接続の CPU 用法、優先級を取ります：

```
dmSQL> select connection_id, affinity_mask, priority_level, cpu_usage from sysuser;
```

システムユーザーに優先級を設定します：

```
dmSQL> select connection id , user name from sysuser;

    CONNECT*              USER NAME
===============   ======================
30420                  BACKUP SERVER
30418                  SYSADM

2 rows selected
dmSQL> call setpriority(30418,3);
```

注　*SYSADMのみSETPRIORITYをコールすることができます。*

精確な接続にクエリシステムユーザーを通じて優先級を取ります：

```
dmSQL> select priority_level from sysuser where connection_id = ?;
```

19 問合せの最適化

この章は、DBMaster の問合せの最適化について説明します。問合せの最適化とは、SQL の問合せに最適な内部実行方法を選択することで、問合せをより速く効率的にします。

この章は、以下のトピックスを記載しています。

- 問合わせ最適化がどういうもので、それが必要な理由。その目的を理解すると、SQL問合わせでの役割がわかります。
- 問合せ実行計画(QEP)が何で、どのようにQEPを読み込むか。QEPについて理解すると、DBMasterのSQLコマンドの実行方法がわかります。
- 問合わせ最適化の操作方法。問合わせ最適化がQEPを見つける方法を理解すると、関連するSQL問合せを書き直すことでより効率的なQWPを見つけることができます。
- コスト機能とは何か。QEPの操作にどのぐらいの回数要するかを理解すると、問合せ最適化が適切な操作を選択する方法がわかります。問合せ最適化がより良い操作を見つけるためにDBMasterにあるコマンドを使用します。
- 統計値が何で、これらの値をどのように利用するか。問合せ最適化における統計値の使用を理解すると、問合せ最適化が実行計画を選択した理由がわかります。
- 問合せの実行速度をどのように向上させるか。効果的な問合せの記述方法を理解すると、問合せ文を書き直すことで実行性能を改善することができます。

19.1 問合せの最適化とは

SELECT、INSERT、DELETE、UPDATE のようなデータ操作言語(DML)は、問合せ最適化の非常に重要な要素です。DBMaster には、SQL の問合せを実行する多くの方法があります。問合せ最適化の目標は、最も効率的な実行計画を見つけることで、その主要な役割は、各操作とその順序を決定することにあります。

➲ **最も効果的な操作を見つける：**

1. 表からデータの読み込み – 順序どおり、又は索引スキャンで読み込みます。
2. 表の結合 – 表は、ネステッド・ループ、又はソート・マージ結合で結合できます。
3. ソート –ソートを必要とするかタイミングが、操作前か操作後か。ソートの回避方法。

問合せ最適化は、ジョインする表のシーケンスを最適化するために、外部ジョインに影響される行数を見積もります。データ特色を熟知したユーザーによっては、問合せ最適化が見つける以上に効果的な方法を見つけることができるかもしれません。

DBMaster の問合せ最適化は、各計画に対し行数を算定し、どのぐらいのディスクページ I/O が必要であるか、一つの表にかかる CPU 時間といった、実行計画のあらゆる可能性を見積もります。その中から、最もコストの低い計画を見つけます。

DBMaster が問合せ実行計画を見つける際に、いくつかの主要な操作を考慮します。

- **表スキャン –** 又はシーケンシャル・スキャンと呼ばれます。これは、データベースのデータ・ページから各行を、シーケンシャル順で受け取ることを意味します。
- **索引スキャン –** データを回収するための順序は、索引ページで指定されたデータ・ページのアドレスを参照します。

- **ネステッド・ジョイン** –マージのために、2つ以上の表を行ごとに比較します。
- **マージ結合** – マージのために、２つの表を並べ替え、行ごとに比較します。
- **ソート** – ソートを実行します。
- **一時表** – 問合せ実行の過程で、一時表を作成します。

➲ **例 1**

ORDER BY 句を使ってソートする：

```
dmSQL> SELECT * FROM tb_staff, tb_salary WHERE tb_salary.basepay=3000 AND
tb_staff.id = tb_salary.id ORDER BY tb_staff.name;
```

➲ **例 2**

問合せ実行計画 1：

```
ort tb staff.name
     merge join tb staff.id = tb salary.id
        index scan tb staff on idx ID(id)
        sort tb salary.id
            table scan tb_salary, filter tb_salary.basepay=3000
```

➲ **例 3**

問合せ実行計画 2：

```
nested join
     index scan tb_staff on idx_name(name)
      table scan tb_salary, filter tb_salary.basepay and tb_staff.id =
tb_salary.id
```

19.2 最適化の操作方法

DBMaster の最適化は以下の方法で、問合せの最適化処理を実行します。

- 問合せを分析し、複数の要素にWHERE句を分解します。
- あらゆる実行シーケンスとそのジョイン・シーケンスを検索します。

- ネステッド・ジョインを使用するか、ソート・マージ結合を使用するかを決定します。
- 表スキャンを使用するか、索引スキャンを使用するかを決定します。
- ソート順を決定します。

最適化の入力

見積もりの精度は、最適化が成功するかどうかを決定付ける重要な要素です。但し、最適化に必要な時間を見積もるための情報は限られています。実際の実行時間と比較するとほんのわずかです。最適化に必要な全ての情報は、システム・カタログ表にあります。この情報を活用し、陳腐化させないために、UPDATE STATISTICS コマンドを使用して下さい。詳細については、19.4節の *「統計」* を参照して下さい。

システム・カタログ表に表示されているデータは以下のとおりです。

- 表の行数
- 表で使用されているデータ・ページ数
- 表の行の平均バイト数
- カラムが使用する平均バイト数
- 各索引カラムの別個の値
- 各カラムの2番目に大きい値と2番目に小さい値; 極端に大きいか小さい値を選択して精度に影響することを避けるため、最大値、最小値を省きます。
- B-tree索引で使用されている索引スキャンのページ数
- B-tree索引のレベル(高さ)
- B-tree索引のリーフ・ページ数
- B-tree索引のクラスタ・カウント

この情報を使用する最適化の場所は、データの値が一律に分散されているところです。データの分散が偏っていて一律でない場合、最適化は不適切な計画を選択します。

要素

最適化の最初の作業は、WHERE 句の全表現をチェックすることです。これらの表現を要素と呼ばれるいくつかの小さい独立した式に分解します。

➲ **例 1**

最適化は、WHERE 句を２つの要素 **tb_staff.id = tb_salary.id** と **tb_salary.basepay=3000** に分解する：

```
dmSQL> select * from tb_staff, tb_salary where tb_staff.id = tb_salary.id and
tb_salary.basepay=3000;
```

➲ **例 2**

１つの要素 **tb_staff.id = tb_salary.id** 又は **tb_salary.basepay=3000** に WHERE 句を使用する：

```
dmSQL> select * from tb staff, tb salary where tb staff.id = tb salary.id or
tb_salary.basepay=3000;
```

➲ **例 3**

２つの要素 **tb_staff.id = tb_salary.id** と **tb_salary.basepay=3000** または **tb_staff.name='joy'**に WHERE 句を使用する：

```
dmSQL> select * from tb_staff, tb_salary where tb_staff.id = tb_salary.id and
(tb_salary.basepay=3000 or tb_staff.name='joy');
```

➲ **例 4**

１つの要素 **tb_staff.id = tb_salary.id** または **tb_staff.name='joy'**に WHERE 句を使用する：

```
dmSQL> select * from tb_staff, tb_salary where tb_staff.id = tb_salary.id and
tb_salary.basepay=3000 or tb_staff.name='joy';
```

上記の例から、式がバイナリ演算「**and**」を含んでいる時、複数の要素に分けられることがわかります。但し、バイナリ演算「**or**」を含んでいるとき、分解することができません。

要素を見つけるために、最適化は各要素の選択度を評価する必要があります。選択度は、各要素で検出されたデータ率です。その値は 0 と 1 の間です。表 t1 には 100 行あります。

➲ 例 5

表 t1 で問合せに 5 行を使用する、**tb_staff**, **tb_staff.id=3** は 5/100、つまり 0.05 です。

```
dmSQL> select * from tb_staff where tb_staff.id=3;
```

文章に複数の要素がある場合、それらは各々独立しているので、この文章の選択度が、これらの要素の成果です。

ジョイン・シーケンス

ジョイン・シーケンスは、マージされる元の表のアクセス順です。ジョイン・シーケンスが異なると、実行シーケンスと実行時間も異なります。いずれにしても、常に正しい結果を回収します。

➲ 例 1

```
dmSQL> select * from tb_staff, tb_salary where tb_staff.id = tb_salary.id;
```

問合せ実行計画 1：

```
nested join
     table scan tb_staff
      table scan tb_salary, filter tb_staff.id = tb_salary.id
```

問合せ実行計画 2：

```
nested join
     table scan tb salary
      index scan tb_staff on tb_staff (id), filter tb_staff.id = tb_salary.id
```

➲ 例 2

```
dmSQL> select * from tb staff, tb salary,tb dept where tb staff.id = tb salary.id and
tb_salary.id=tb_dept.id;
```

問合せ実行結果、3! (=6)ジョイン・シーケンスがあります：

```
(tb_staff, tb_salary), tb_dept
(tb_staff, tb_dept), tb_salary
(tb_salary, tb_staff), tb_dept
```

```
(tb salary, tb dept), tb staff
(tb dept, tb staff), tb salary
(tb_dept, tb_salary), tb_staff
```

DBMaster は、これらの全ジョイン・シーケンスを検索し、コストを算出し、最適な方法を選択します。

ネステッド・ジョインとマージ結合

ネステッド・ジョインとマージ結合の２つの方法があります。

- ネステッド・ジョインは、結合のために２層のネステッド・ループを使用します。その時間コンプレックスの分析アルゴリズムは、**$O(n^2)$**です。
- マージ結合は、前もって２つの表をソートし、行ごとにソートした順序にこれら２つの表を統合します。ソートの時間コンプレックスは、**$O(n \times \log(n))$**です。ソート済みのデータに、**$O(n)$**のジョインを実行する時間コンプレックスがあります。ソート・マージ結合は同等の結合にのみ使用されます。

時間コンプレックスの観点から見ると、マージ・ジョインはネステッド・ジョインよりも優れています。但し、２つの表にある行数の差が大きい場合のような例外があります。これに関わらず、最適化はコスト関数と統計的な値でジョインを実行する最適な方法を決定します。

表スキャンと索引スキャン

表スキャンは、行ごとに表から行を順に取得します。例えば、条件 **age > 50** に合う表の全ての行を見つけて、各データ・ページから各行を受け取ります。更に、希望のデータを回収するために、マッチする条件と各行を比較します。

もう一方のスキャンは、索引スキャンと呼ばれています。これは、表のカラムに索引を作成し、索引を参照して必要な全データを見つけます。DBMaster で使用する索引方法は、B-tree です。索引スキャンの使用に必要な条件は、使用するカラムに索引を編成することです。

ソート

問合せ最適化のもう一つの重要な操作は、結合の前後にどのようにソートをするか、又はいかにソートを回避するかということです。

➲ **例**

ソートを作成する：

```
dmSQL> select * from tb_staff, tb_salary where tb_staff.id=tb_salary.id order by
tb_staff.basepay;
```

問合せ実行計画 1、最適化はマージ後にソートを実行する：

```
sort tb staff.basepay
     merge join tb staff.id=tb salary.id
          index scan tb_staff on idx_id(id)
          sort tb_salary.id
               table scan tb_salary
```

問合せ実行計画 2、最適化はマージ前にソートを実行する：

```
 nested join
     index scan tb_staff on idx_base(basepay)
      table scan tb_salary, filter tb_staff.id=tb_salary.id
```

19.3 問合せの時間コスト

ディスクからデータを読み出す時間とカラム値を比較する時間は、問合せ実行における２つの重要な部分です。

CPUコスト

データベース・サーバーは、メモリでデータを処理します。メモリに行を読み込み、テストのためのフィルタ表現を使用します。まず、メモリに２つの表からデータを読み出し、それからジョイン条件をテストします。加えて、データベース・サーバーは、各行から選択したカラムのデータを回収する必要があります。ソートやキーワード、又は「**like**」や「**match**」のようなワイルドカードが使用される場合、より時間を費やします。

I/Oコスト

メモリで行をチェックする以上に、ディスクからの行の読み出しに時間がかかりますので、最適化の主な目的は、ディスクからの読み出しをへらすことです。

データベース・サーバーのディスク・ストレージを処理する基本単位は、ページと呼ばれます。ページは、クラスタされたブロックで構成されています。ページのサイズは、データベース・サーバーに関係します。DBMasterのこのサイズは、4KB,8KB,16KB または 32KB がサポートできます。ページ行の容量は、行のサイズに関係します。データ・ページは 4KB ならには、通常 10 から 100 行あります。索引ページのエンティティは、キー値と 4 バイトのポインタを含みます。索引ページには、通常 100 から 1000 エントリがあります。

データベース・サーバーは、処理のためにディスクから読み込むディスク・ページのコピーを保存するメモリ・スペースが必要です。メモリ・スペースに限りがあるので、いくつかのページは再度読み込まれるかもしれません。メモリ・スペースは、ページ・バッファと呼ばれています。必要とされるページが、ページ・バッファにある場合、サーバーはディスクからは行を読み込まず、パフォーマンスは上がります。データベース・サーバーとオペレーティング・システムは、ディスクのページ数とページ・バッファの数を決定します。ページを読み込む実際のコストは均一でないので、その見積もりは困難です。

バッファは、以下の要素の組み合わせです。:

- **バッファ** – ページ・バッファにターゲット・ページがある可能性があります。この場合、アクセス・コストは、ほとんど無視されます。
- **競合** – 複数のアプリケーション・プログラムが、ディスクのようなハードウェア装置を使用しようとする際に、データベース・サーバーからの要求が遅れます。
- **シーク時間** – これは最もディスクで時間を要する動作です。つまり、希望のデータの位置に読み込み/書き込みヘッドを移動するのに費やす時間です。ディスクの速さとディスク読み込み/書き込みヘッドの最初の

位置に関係します。そのバリエーションもシーク時間に多いに依存します。

- **呼び出し時間** – ローテーション遅延時間としても知られています。ディスクのスピードと読み込み/書き込みヘッドの場所に関係します。

表スキャンのコスト

表で全データをスキャンするために時間がかかります。問合せの文がある無いに関わらず、ページに含まれる全データを比較する必要があります。表スキャンのコストは、データ・ページの数と同等です。

索引スキャンのコスト

索引スキャンは、B-tree 索引ページを経由してデータを読み込みます。2 種類の索引スキャンがあります。一つは、B-tree が参照するデータ・ページを読み込みます。もう一方は、索引リーフから直接データを読み込みます。これはリーフ・スキャンと呼ばれています。

➲ **例**

表 **tb_staff** カラム **id** と **name** を使って、**c1** に編成された索引をスキャンする:

```
dmSQL> select * from tb_staff where id > 0;
```

次の方法でも同様に実行できます：

```
dmSQL> select id from tb_staff where id > 0;
```

リーフ・スキャンを使用します。リーフ・ページには必要な全データがあるので、データ・ページからデータを読み込む必要がありません。

- 全データを読み込む時、索引スキャンのコスト：

  コスト = B tree レベル I/O + リーフ・ページ数 I/O + クラスタ・カウンタ

- 全データを読み込み、リーフ・スキャンのみ必要とする場合、索引スキャンのコスト：

  コスト = B tree レベル I/O + リーフ・ページ数 I/O

- 行を読み込む時、索引スキャンのコスト：

コスト = B tree レベル I/O + 1 リーフ・ページ I/O + 1 データ・ページ I/O

- 行を読み込み、リーフ・スキャンのみ必要とする場合、索引スキャンのコスト：

  コスト = B tree レベル I/O + 1 リーフ・ページ I/O

- 部分データを読み込む時、索引スキャンのコスト：

  コスト = B tree レベル I/O + (リーフ・ページ数× S) + (クラスタ・カウント×S)

  S は、選択度を表します。

ソートのコスト

ディスクからメモリへのデータ読み込みは、唯一時間がかかる作業です。コスト算出は、ソートされるカラム数が c、ソートされるキーのバイトが w、ソートされる行の数が n の場合、"$c \times w \times n \times \log_2(n)$ "に比例します。

ネステッド・ジョインのコスト

ネステッド・ジョインでデータ・ページにアクセスするためには、２つ以上のループが必要です。ネステッド・ジョインでは、外部表は内部表とは異なります。一般的に、ネステッド・ジョインのコストは以下のように計算します：

外部表 I/0 + 内部表 I/0 x 外部表にある行の数

マージ結合のコスト

マージ結合を実行する前に、表をソートする必要があります。既にマージ・キーを使って統合するために２つの表をソートしたと想定します。マージ結合のコストは、これら２つの表の I/O の合計です。マージ・キーでソートを実行しない場合でも、ソートのコストを追加する必要があります。

19.4 統計

統計は、表データの量と分散を表します。最適なアクセス計画を見つけるためのコスト関数の情報が含まれます。但し、表データが挿入/削除/更新されている場合、統計は陳腐化します。統計値を更新して問合せの効果を上げるため、UPDATE STATISTICS 文を実行します。

統計の種類

DBMaster は、以下の統計を取ります。

表について

- **nPg** – データのページ数
- **nRow** – データの行数
- **avLen** – 行の平均バイト

カラムについて

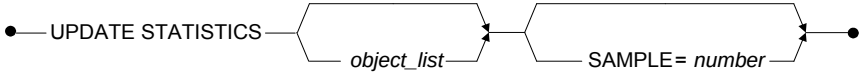
- **distVal** – 異なる値の数
- **avLen** – 各カラムの平均バイト
- **loVal** – カラムの2番目に小さい値
- **hiVal** – カラムの2番目に大きい値

索引について

- **nPg** – 索引のページ数
- **nLevel** – 索引ツリーのレベル数
- **nLeaf** – 索引ツリーのリーフ数
- **distKey** – 異なるキーの数
- **distC1** – 最初の索引カラムの異なるキー数
- **distC2** – 最初の2索引カラムの異なるキー数

- **distC3** – 最初の3索引カラムの異なるキー数
- **nPgKey** – 各キーの索引のページ数
- **cCount** – カウントされたクラスタの数; 索引にアクセスするデータ・ページの数

UPDATE STATISTICS構文



オブジェクトリスト句

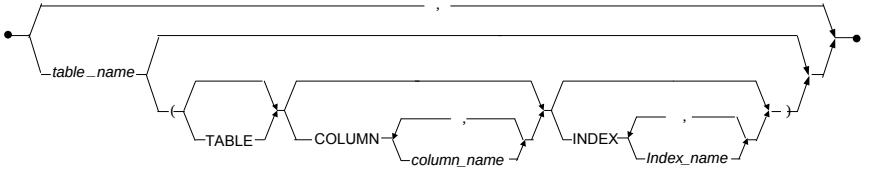


図 19-1：UPDATE STATISTICS 文の構文

- **SAMPLE—**サンプル率を意味します。1から100の間の整数です。

➲ **例 1**

最適化のサンプル率で、カラム、索引、システム表を含む全ての表の統計を更新する：

```
dmSQL> UPDATE STATISTICS;
```

➲ **例 2**

サンプル率 30 で、カラム、索引、システム表を含む全ての表の統計を更新する：

```
dmSQL> UPDATE STATISTICS sample=30;
```

➲ 例 3

表 jeff.tb_staff の統計を更新する：

```
dmSQL> UPDATE STATISTICS jeff.tb_staff;
```

➲ 例 4

表 jeff.tb_staff の索引 idx_id の統計を更新する：

```
dmSQL> UPDATE STATISTICS jeff.tb_staff (TABLE COLUMN(name, age) INDEX(idx_id));
```

➲ 例 5

表 jeff.tb_staff と表 jeff.tb_dept の統計を更新する：

```
dmSQL> update statistics jeff.tb_staff, jeff.tb_dept;
```

自動更新統計デーモン

DBMaster には、自動的にデータベース全体の統計を更新するためのデーモンもあります。全表にある全ての統計が再生成されるわけではありませんが、表の統計がどのぐらい以前に更新されたか、或いは最新の統計更新時以降にどのぐらい表が変更されたか等に基づく最適化されたサンプル率で、再生成されます。ユーザーは自分で更新統計モードが選択でき、異なる表に異なるサンプル率も設定でき、更新統計の初期時間と間隔時間も変更できます。ユーザーはシステムテーブル SYSUSER を問い合せて更新統計状態を取ります。この状態はキャラクタストリングでカラム SQL_CMD にストアドされます。そして、更新統計コマンドが実行されていると、ユーザーはプロシージャ **setSystemOption()**を通じてこれをアボートできて、この接続が切断できません。

更新統計サーバーは存在しなくまたは終了した場合、更新統計デッモンが実行しないことを注意してください。そして、ユーザーは一時表に表統計を設定することができません。

ユーザーは更新の性能を高めるため三つの方法があります：**dmconfig.ini** を設定、各表を設定、**setSystemOption**、更新統計状態を取得と更新統計コマンドを終了。

DMCONFIG を設定

DBMaster は更新統計デッモンをアクティブにするため、配置キーワードがあります。**DB_StSvr** を 1 に設置すると、自動更新統計デッモンをアクティブにします；**DB_STMOD** はデータベースの増分更新統計モードを設定します。**DB_STSTM** は更新統計時間を開きます；**DB_STSTV** は更新統計デッモン間隔を指定します；**DB_STSSP** は更新統計サンプル率を設置します。

➲ **例 1**

データベースを起動する前、**dmconfig.ini** ファイルに更新統計デッモンに関するキーワードを配置します：

```
[DBNAME]
; Here omit other keywords
DB_STSVR = 0
DB_STMOD = 1
DB_STSTM = 2010-10-10 10:00:00
DB_STSTV = 12:00:00
DB_STSSP = 70
```

今、データベースを起動して、更新統計デッモンはスケジュールによって自動に統計を更新します。

➲ **例 2**

ランタイムに更新統計デッモンに関しての引数を設置することを許せます：

```
dmSQL> call setSystemOption('STSVR','1');          //activate backup server
dmSQL> call setSystemOption('STMOD','1');                  //reset DB_STMOD
dmSQL> call setSystemOption('STSTM','2009/6/6 20:30:00');   //reset DB STSTM
dmSQL> call setSystemOption('STSTV','7-00:00:00');         //reset DB STSTV
dmSQL> call setSystemOption('STSSP','60');                  //reset DB STSSP
```

各表の設定

更新統計デーモンの**表設定モード**を起動する時、つまり **DB_STMOD** の値を 1 に設定する場合に、ユーザーは SQL 文 UPDATE STATISTICS SET を実行することによって、各表の更新統計オプションを設定することができます。SQL 文 UPDATE STATISTICS SET でのモードの値を 1 に設定する場合、

当該表の更新統計サンプルレートは UPDATE STATISTICS SET での SAMPLE の値によって決めます；SQL 文 UPDATE STATISTICS SET での MODE の値を 0 に設定する場合、当該表の更新統計サンプルレートは **dmconfig.ini** ファイルでの **DB_STSSP** の値によって決めます。

全てのテーブルの設定情報はシステムテーブル SYSTABLE に保存されています。テーブルの更新統計モードが UPD_STS_MODE カラムに保存され、テーブルの更新統計サンプルレートが UPD_STS_SAMPLE カラムに保存されています。

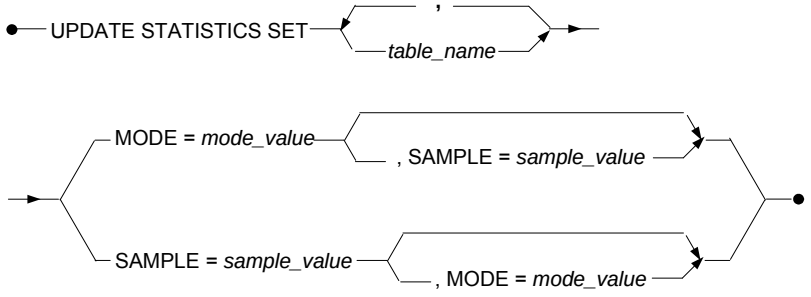


図 19-2：UPDATE STATISTICS SET の構文

➲ **例 1**

表 **jeff.tb_staff** の更新統計モード及びサンプルレートを設定します。

```
dmSQL> UPDATE STATISTICS SET jeff.tb_staff MODE = 1,  SAMPLE = 80;
dmSQL> select TABLE_NAME, TABLE_OWNER, UPD_STS_MODE, UPD_STS_SAMPLE from SYSTABLE;
         TABLE_NAME             TABLE_OWNER        UPD_STS_MODE   UPD_STS_SAMPLE
====================== ====================== ============== ===============
TB STAFF                JEFF                   -1             80
1 rows selected
```

➲ **例 2**

表 **jeff.tb_staff** 及び表 **jim.tb_salary** の更新統計モード及びサンプルレートを設定します。

```
dmSQL> UPDATE STATISTICS SET jeff.tb staff, jim.tb salary MODE = 1, SAMPLE = 60;
dmSQL> select TABLE_NAME, TABLE_OWNER, UPD_STS_MODE, UPD_STS_SAMPLE from SYSTABLE;
```

```
        TABLE NAME           TABLE OWNER         UPD STS MODE   UPD STS SAMPLE
====================== ====================== ============== ===============
TB STAFF               JEFF                   1              60
TB SALARY              JIM                    1              60
2 rows selected
```

SETSYSTEMOPTION

自動的に更新及び統計するシステムオプションはそれぞれ STSVR、STMOD、STSTM、STSTV 及び STSSP です。データベースが実行している間に、システムストアドプロシージャ **setSystemOption()**または **setSystemOptionW()**をコールして、上記のシステムオプションを設定することができ、そしてそれを **dmconfig.ini** ファイルに書き入れることもできます。また、システムストアドプロシージャ **getSystemOption()**をコールして、システムオプションの情報を取得します。システムストアドプロシージャ **setSystemOption()**、**setSystemOptionW()**及び **getSystemOption()**の使用方法の詳細については、*「SQL 文と関数参照編」*と*「ODBC プログラム参照編」*をご参照ください。

「」「」''''

➲ **例 1**

データベースが実行している間にシステムオプション **STSVR** を 0 に設定します。

```
dmSQL> call setSystemOptionW('STSVR', '0');
```

データベースが実行している間にシステムオプション **STSVR** を 1 に設定して **dmconfig.ini** ファイルの既存なデータベースセクションに **DB_STSVR = 1** を書き入れます。

```
dmSQL> call setSystemOptionW('STSVR', '1');
```

データベースが実行している間にシステムオプション **STSVR** の値を取ります。

```
dmSQL> call getSystemOption('STSVR', ?);
```

➲ **例 2**

データベースが実行している間にシステムオプション **STSSP** を 70 に設定します。

```
dmSQL> call setSystemOptionW('STSSP', '70');
```

データベースが実行している間にシステムオプション STSSP を設定して **dmconfig.ini** ファイルの既存なデータベースセクションに **DB_STSSP = 30** を書き入れます。

```
dmSQL> call setSystemOptionW('STSSP', '30');
```

データベースが実行している間にシステムオプション **STSSP** の値を取ります。

```
dmSQL> call getSystemOption('STSSP', ?);
```

注　*全てのランタイムに配置するキーワードが変更するわけではありません。*

更新統計の状態

システムテーブル SYSUSER をクエリすることによって更新統計状態を取得できます。これはキャラクタストリングで **SQL_CMD** カラムにストアドされます。

以下は更新統計情報です：

```
[EXEC] update statistics command // (Total, start_time, execute_time, remain_time,
complete_percent) (table_name, start_time, execute_time, remain_time,
complete_percent) (index_name, start_time, execute_time, remain_time,
complete_percent)
```

この更新統計情報は三つの部分を含みます：合計、テーブル、索引。合計は全部の更新統計オブジェクトという意味です。テーブルは一つの表統計で、表統計に属する索引とデータページを含みます。索引は索引統計、またはデータページ統計ということです。

➲ **例**

```
dmSQL> SELECT CONNECTION_ID, SQL_CMD FROM SYSUSER;
CONNECTION_ID           SQL_CMD
============= ============================================================
3264          [EXEC] Update t1 set c1 = c1 + 1;
3267          [EXEC] update statistics // for table SYSADM.HUNDRED index HUNDRED_CODE
start at 2011/02/21 09:19:54
2 rows selected
```

更新統計のモニタ及びアボート

DBMaster では、更新統計の状態を表示するのをサポートしています。表 SYSUSER のカラム SQL_CMD をクエリすることによって、更新統計のプロセスがモニタできます。また、*setSystemOption('STS_ABORT', 'connection_id')*をコールして実行中の更新統計をアボートすることができます。表 SYSUSER のカラム SQL_CMD 及びシステムストアドプロシージャ *setSystemOption()*の使用方法については、*「JDBA ユーザー参照編」*及び*「SQL 文と関数参照編」*をご参照ください。

➲ **例 1**

SYSUSER 表のカラム SQL_CMD をクエリして更新統計プロセスをモニターします：

```
dmSQL> select SQL_CMD from SYSUSER;
```

➲ **例 2**

接続 ID が 14076 である更新統計コマンドをアボートします：

```
dmSQL> call setSystemOption('STS_ABORT', '14076');
```

➲ **例 3**

値 0 は特別な接続 ID です。更新統計に関する全部の接続をアボートするという意味です。

```
dmSQL> call setSystemOption('STS_ABORT', '0');
```

統計のロードとアンロード

UNLOAD STATISTICS 文を使って、外部テキスト・ファイルに統計値をダンプすることができます。又、LOAD STATISTICS 文で、外部テキスト・ファイルからデータベースに統計値をコピーすることもできます。

➲ **例 1**

UNLOAD STATISTICS を使う：

```
dmSQL> UNLOAD STATISTICS TO file1;//データベースの統計を file1 にダンプ
```

➲ 例 2

LOAD STATISTICS を使う：

```
dmSQL> LOAD STATISTICS FROM file1;//外部テキスト file1 から統計を読み込む
```

経験豊富なユーザーであれば、ファイルの統計を修正し、データベースにそれを入力することによって問合せの効率を向上することができます。

➲ 例 3

UNLOAD STATISTICS で生成された外部テキストファイルは以下のようになります：

```
DBname = TESTDB

TBowner = jeff
TBname = tb_staff
TBpage = 5
TBrows = 30
Tbavlen = 50

COname = age
COtype = INTEGER
COdist = 12
COavlen = 4
COlow = 25
COhigh = 42

IXname = idxage
IXpages = 5
IXlevel = 2
IXleaf = 3
IXdist = 12
IXdistC1 = 12
IXdistC2 = 12
IXdistC3 = 12
IXpgkey = 8
IXcount = 7
```

19.5 問合せの高速化実行

以下の修正を行って、問合せを高速化することができます。

- データ行の読み込みを少なくする。
- ソートを回避、又はデータ行とカラムのソートを少なくする。
- データの読み込みにシークエンシャルを使用する。

データ・モデル

データ・モデルの定義は、データベースにある全ての表、ビュー、索引、とりわけ索引の存在を含みます。索引が結合やソートやビューのような条件で使用されるかどうかを指定します。

問合せ計画

問合せ実行計画をチェックするために、SET DUMP PLAN ON 文を使用することができます。

実行計画の特徴は、以下のとおりです。

- **Index**—索引が使用されたかどうか、又は使用方法を見るために出力データをチェックします。
- **Filter**—どのぐらいのデータを述語が検出できるか、述語要素でチェックします。
- **Query**—アクセス計画が最適かどうかを見た後、問合せをチェックします。

➲ **例**

実行計画を見る：

```
dmSQL> SET DUMP PLAN ON;
```

索引チェック

問合せカラムに適切な索引があるかどうかをチェックします。問合せ効率を向上させるために、以下の節で解説する方法を使用して下さい。

フィルター・カラム

これは、効率的な問合せのためにソース情報の一部に過ぎません。SELECT 文の WHERE 句を使用して、出力情報の量をコントロールすることができます。

以下に、いくつかの高度な WHERE 句の使用方法を説明します。

相互に関連する副問合せを回避する

複数のカラムが WHERE 句の主問合せと副問合せで使われる時に、相互に関連する副問合せが発生します。主問合せに含まれる各データ行の複数の副問合せで、異なる結果が戻されます。各行にあるカラムのデータが、副問合せにある前の行のカラムと異なる場合、主問合せから取得した各業の新規問合せを実行することと同じです。

時間のかかる副問合せを見つけた場合、まず相互に関連する副問合せかどうかをまずチェックして下さい。もし、そうであれば、条件を変えるように問合せを記述し直して下さい。問合せを記述し直すのが容易ではない場合、データ行の数を減らす別の方法を試して下さい。

難解な定型表現を回避する

キーワード **like** は、定型表現として知られる、ワイルド・カードの比較として使用されます。ワイルド・カードは文章の初期で使用され、データベース・サーバーは、索引フィルタを使用できないので、各行をチェックします。これにより、DBMaster にシークエンシャルに表の全ての行にアクセスし、チェックするようにします。

➲ **例**

ワイルド・カード"*"と共にキーワード"like"を使う：

```
dmSQL> SELECT * FROM tb_salary WHERE name LIKE '*st';
```

問合せ結果

問合せが実際何を行うのかを理解したら、同じ結果を取得するために他のの同様の問合せを見つけることができるようになります。より効率的に問合せを書き直すためのいくつかのヒントを紹介します。

- ビューを使ってジョインを書き直す。
- ソートを回避する、又は減らす。
- 大きな表へは順序どおりにアクセスしないようにする。
- ユニオンを使用して、順序どおりのアクセスをさける。

一時表

一時オーダー表の利用は、問合せを高速化するために効果的です。同時に、最適化の演算を簡潔にして複数のカラムでのソート演算を回避することにも役に立ちます。

- 複数のカラムでのソートを回避する一時表を使用する。
- 非シーケンシャルなアクセスのソートに置き換える。

19.6 構文ベースの問合せ最適化

最適化は、コスト関数と統計で自動的に問合せ実行計画を選択します。但し、データ分散が偏っている特殊な場合には、最適化は不適切な問合せ実行計画を選択する可能性があります。この問題を解決するために、DBMasterには*構文ベースの問合せ最適化*と呼ばれる最適化の手法があります。

問合せに使用するスキャンの種類と索引スキャンに使用する索引を、任意に指定することができます。加えて、データベースの統計を最近更新していない場合でも、自動的に最も効率の良いスキャンの種類を決定します。使用する索引の種類には、5 種類のケースを指定できます。

強制索引スキャン

強制索引スキャンの構文：

```
tablename (INDEX [=] idxname [ASC|DESC])
```

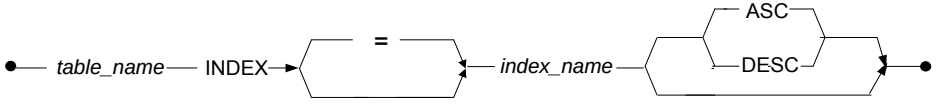


図 19-3：インデックススキャン強制の構文

➲ **例 1**

表スキャンの強制実行を意味する索引 0 を指定する：

```
dmSQL> SELECT * FROM tb_staff (INDEX=0);
```

➲ **例 2**

主キーで索引スキャンの強制実行を意味する索引 1 を指定する：

```
dmSQL> SELECT * FROM tb_staff (INDEX=1);
```

➲ **例 3**

索引 ***idx1*** で索引スキャンを強制実行する：

```
dmSQL> SELECT * FROM tb_staff (INDEX idx1);
```

➲ **例 4**

表 **tb_staff** には最適化にスキャンの種類を決定させ、表 **tb_salary** には索引 **idx_id** で索引スキャンを強制実行する：

```
dmSQL> SELECT * FROM tb_staff, tb_salary (INDEX idx1);
```

強制索引スキャンと「別名」

索引スキャンを強制実行し、表に別名を付ける構文：

```
tablename (INDEX [=] idxname) aliasname
```

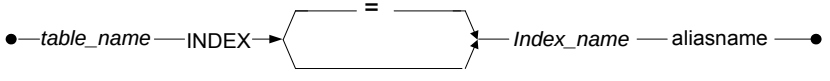


図 19-4：インデックススキャン強制およびエイリアスの構文

➲ **例**

索引 **idx_id** で索引スキャンを強制実行し、表に別名を付ける：

```
dmSQL> SELECT * FROM tb_staff (INDEX idx_id) a, tb_staff b WHERE a.id = b.id;
```

強制索引スキャンと「シノニム」

シノニムを使った索引スキャンを強制実行する構文：

```
synonymname (INDEX [=] idxname)
```

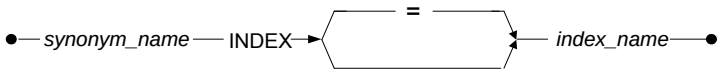


図 19-5：インデックススキャン強制およびシノニムの構文

➲ **例**

シノニム **staff**:を使って、索引 **idx_id** で索引スキャンを強制実行する：

```
dmSQL> SELECT * FROM staff (INDEX idx_id);
```

強制索引スキャンと「ビュー」

ビュー作成時に索引スキャンを強制実行させる構文：

```
viewname (INDEX [=] idxname)
```

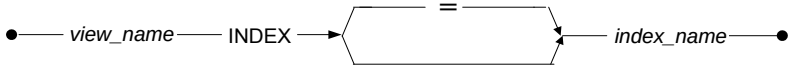


図 19-6：インデックススキャン強制およびビューの構文

➲ **例 1**

ビュー**vi_staff** を作成する際に、索引 **idx_id** で索引スキャンを強制実行する：

```
dmSQL> CREATE VIEW vi_staff as SELECT * FROM t1 (INDEX idx1);
```

ビュー選択時に、索引スキャンを強制実行することはできません。

➲ **例 2**

エラーが発生する誤った使用例：

```
dmSQL> SELECT * FROM vi_staff (INDEX idx_id) ;
```

強制テキスト索引スキャン

テキスト索引スキャンを強制実行する構文:

```
tablename (TEXT INDEX [=] idxname)
```

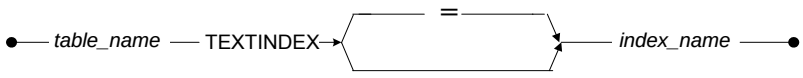


図 19-7 : テキストスキャン強制の構文

➲ **例**

索引 ***tidx1*** でテキスト索引スキャンを強制実行する :

```
dmSQL> SELECT * FROM tb_staff (TEXT INDEX tidx1);
```

強制ループ結合 (ネステッド結合)

2 つのテーブルの間でネステッド結合を強制するのに使用される一般的な構文:

```
tablename { INNER | OUTER } LOOP JOIN tablename
```

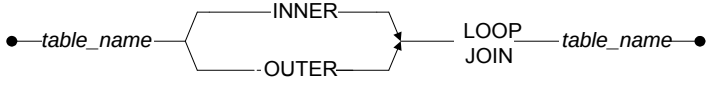


図 19-8 : ループ結合強制の構文

注 *このタイプの強制結合には、INNER JOINまたはOUTER JOIN構文を使用してください。*

➲ **例 1**

```
dmSQL> SELECT * FROM tb_staff INNER LOOP JOIN tb_salary ON tb_staff.id=tb_salary.id;
```

➲ **例 2**

```
dmSQL> SELECT * FROM tb_staff OUTER LOOP JOIN tb_salary ON tb_staff.id=tb_salary.id;
```

強制マージ結合

2 つのテーブルの間でマージ結合を強制するのに使用される一般的な構文:

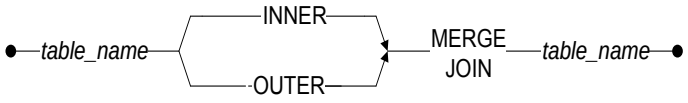


図 19-9：マージ結合強制の構文

注 *結合がマージ結合を使用できない場合、マージ結合の強制は使用でません が、エラーメッセージは返されません。*

➲ **例 1**

```
dmSQL> SELECT * FROM tb_staff INNER MERGE JOIN tb_salary ON tb_staff.id= tb_salary.id;
```

➲ **例 2**

```
dmSQL> SELECT * FROM tb_staff OUTER MERGE JOIN tb_salary ON tb_staff.id= tb_salary.id;
```

強制結合シーケンス

すべてのテーブルの結合シーケンスを強制するので、結合シーケンスはスワップできません。結合シーケンスの強制に使用される一般的な構文:

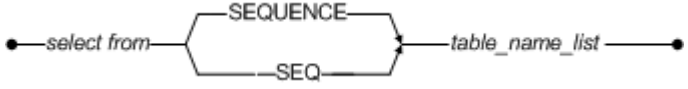


図 19-10：結合シーケンス強制の構文

➲ **例 1**

```
dmSQL> select * from sequence tb_staff, tb_salary, tb_dept where tb_staff.id=
tb_salary.id and tb_salary.basepay=tb_dept.basepay;
```

➲ **例 2**

```
dmSQL> select * from seq tb_staff inner join tb_salary on tb_staff.id= tb_salary.id
inner join tb_dept on tb_staff.basepay=tb_detp.basepay;
```

強制GROUP BYメソッド

結合シーケンスの強制に使用される一般的な構文:

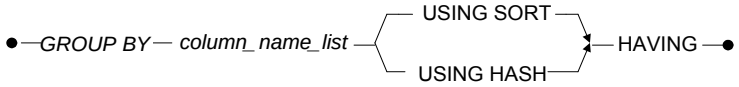


図 19-11 Group by メソッド強制の構文

➲ **例 1**

```
dmSQL> select id,name,count(*) from tb_staff group by id,name using hash;
```

➲ **例 2**

```
dmSQL> select id,name,count(*)  from tb salary group by id,name using sort having
sum(basepay)>0;
```

19.7 ダンプ計画を読み込む方法

遅い問合せをチェックする最初の手順は、実行計画を読み込むことです。DBMaster では、実行計画関数のダンプと読み込みができます。

ダンプ計画には 3 つの SQL 文があります。

```
dmSQL> SET DUMP PLAN ON;
```

ダンプ計画オプションを ON にします。遅い問合せは、計画をダンプしてからコマンドを実行します。

```
dmSQL> SET DUMP PLAN OFF;
```

ダンプ計画オプションを OFF にします。遅い問合せは、コマンドを実行するだけでダンプしません。これが初期設定です。

```
dmSQL> SET DUMP PLAN ONLY;
```

計画ダンプのみを ON にします。コマンドを実行しません。

一瞥すると、ダンプ計画が ON と呼ばれるいくつかのブロックで構成されていることがわかります。問合せ最適化は、いくつかの ON ブロックに分けます。各ブロックは、論理的な最適化の単位です。最適化は、各 ON ブロックを最適化します。単純な結合問合せには、通常一つの ON ブロック

しかありませんが、副問合せのような複合問合せは、複数の ON ブロックを生成します。

最適化は、各 ON ブロックのコストに基づいて最適な実行計画を見つけます。ON ブロックをいくつかの PL ブロックに分けます。各 PL ブロックは、スキャンやジョイン等の演算を表します。

この章の前述の節で説明している以下の名称を習熟する必要があります。

- 表スキャン
- 索引スキャン
- ネステッド・ジョイン
- マージ・ジョイン
- 要素

表スキャン

➲ **例**

表スキャン **tb_staff** のダンプ計画をセットする：

```
dmSQL> SET DUMP PLAN ON;
dmSQL> SELECT * FROM tb_staff WHERE c1>1;
```

結果、表スキャン **tb_staff** のダンプ・スキャン：

```
----- begin dump plan -----

{ON Block 0}
ON Type     : SCAN

[PL Block 0]
Method     : Scan
Table Name : tb_staff
Type       : Table Scan
Order      : <none>
Factors    : (1) tb_staff.id > 1
I/O Cost   : 101.0
CPU Cost   : 25.3
Sub Cost   : 0.0
```

```
Result Rows: 330.0

----- end dump plan -----
```

最初の 2 行は、ON ブロックの情報を示しています。

{ON Block 0} – ブロック ID が 0 の ON ブロック。

ON Type: SCAN – ON ブロックの種類はスキャン。

ON ブロックには、一つの PL ブロックが含まれます。

[PL Block 0] – ブロック ID が 0 の PL ブロック。

Method: Scan – この PL ブロックは、スキャン演算を実行します。

Table Name: tb_staff – 定義された表 t1 でのスキャン。

Type: Table Scan – スキャンの種類が、表スキャン。

Order: <none> -- スキャン順序、表スキャンでの使用はありません。

Factors: (1) tb_staff.id > 1 – このスキャンは、フィルタ t1.c1 > 1 を使用します。

I/O Cost: 101.0 – 試算した I/O コストは、101.0 ページです。

CPU Cost: 25.3 – 試算した CPU コストは、25.3 ページです。

Sub Cost: 0.0 –試算した PL ブロックの副ブロックのためのコストの合計。

Result Rows: 330.0 – スキャンとフィルタの後に試算した結果行。

索引スキャン

➲ **例**

WHERE を使って、表 **tb_salary** から **id** と **name** のダンプ計画をセットする：

```
dmSQL> set dump plan on;
dmSQL> select id,name from tb_salary where id>1 and name='john';
```

結果、WHERE を使った表 **tb_salary** から **id** と **name** のダンプ・スキャン：

```
----- begin dump plan -----

{ON Block 0}
ON Type     : SCAN
```

```
[PL Block 0]
Method     : Scan
Table Name : tb salary
Scan Type  : Index Scan on idx21(name, id)
Order      : ASC
Index EQFA#: 1
Index FA#  : 2
Index FACOL: 1, 2
Index Cost : 2
Factors    : (1) tb_salary.name = 'john'
           : (2) tb_salary.id > 1
I/O Cost   : 2.0
CPU Cost   : 0.6
Sub Cost   : 0.0
Result Rows: 13.0

----- end dump plan -----
```

最初の 2 行は、ON ブロックの情報を示しています。

{ON Block 0} – ブロック ID が 0 の ON ブロック。

ON Type: SCAN – ON ブロックの種類はスキャンです。

又 ON ブロックは、一つの PL ブロックを含んでいます。

[PL Block 0] – ブロック ID が 0 の PL ブロック。

Method: Scan – ブロックは、スキャンを実行します。

Table Name : tb_salary– 表 tb_salary のスキャン。

Scan Type: Index Scan on idx21(name, id) – スキャンの種類は索引で、索引カラム name, id を使って、索引 *idx12* を適用します。

Order: ASC – 昇順索引スキャン順。

Index EQFA#: 1 – 索引スキャンに適用した要素数と同じ、この例では tb_salary.name = 'john'を使用しています。

Index FA#: 2 – 索引スキャンで適用した要素数、この例では tb_salary.name = 'john' and tb_salary..id > 1 を使用しています。

Index FACOL: 1, 2 – 索引カラムからマッピングする要素 ID。この例では、最初の索引カラム c2 を要素(1) tb_salary.name = 'john'に、2 番目の索引カラム c1 を、要素(2) tb_salary..id > 1 にマップします。

Index Cost: 2 – 試算した索引ページ・コストは 2 です。

Factors: (1) tb_salary.name = 'john'

(2) tb_salary.id > 1 – フィルタ tb_salary.name = 'john' と tb_salary.id > 1 を適用します。

I/O Cost: 2.0 – 試算した I/O コストは 2 ページです。

CPU Cost: 0.6 – 試算した CPU コストは 0.6 です。

Sub Cost: 0.0 –試算した PL ブロックの副ブロックのためのコストの合計

Result Rows: 13.0 – スキャンとフィルタの後に試算した結果行。

同等結合

➲ **例**

WHERE を使って、**tb_staff** と **tb_salary** からダンプ計画をセットする：

```
dmSQL> set dump plan on;
dmSQL> select * from tb_staff, tb_salary where tb_staff.id=tb_salary.id;
```

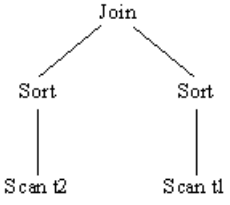
結果、WHERE を使った **tb_staff** と **tb_salary** からのダンプ・スキャン：

```
----- begin dump plan -----

{ON Block 0}
ON Type     : JOIN

[PL Block 0]
Method      : Join
Type        : Merge Join
Factors     : (1) tb_staff.id = tb_salary.id
I/O Cost    : 8.5
CPU Cost    : 573.8
Sub Cost    : 231.6
Result Rows: 500.0
Sub Block 1: [PL Block 1]
```

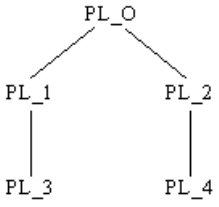
```
Sub Block 2: [PL Block 2]

[PL Block 1]
Method     : Sort
I/O Cost   : 4.2
CPU Cost   : 274.4
Sub Cost   : 120.0
Result Rows: 1000.0
SUB Block  : [PL Block 3]

[PL Block 3]
Method     : Scan
Table Name : tb_salary
Type       : Table Scan
Order      : <none>
Factors    : <none>
I/O Cost   : 101.0
CPU Cost   : 25.3
Sub Cost   : 0.0
Result Rows: 1000.0

[PL Block 2]
Method     : Sort
I/O Cost   : 4.2
CPU Cost   : 274.4
Sub Cost   : 120.0
Result Rows: 1000.0
SUB Block  : [PL Block 4]

[PL Block 4]
Method     : Scan
Table Name : tb_staff
Type       : Table Scan
Order      : <none>
Factors    : <none>
I/O Cost   : 101.0
CPU Cost   : 25.3
Sub Cost   : 0.0
Result Rows: 1000.0
```

```
----- end dump plan -----
```

この例では、複数の PL ブロックがあります。PL ブロックの関係は、副ブロックの情報を使って結合されています。



ブロックを表す単純なツリー



各ノードを名称に置き換える

ジョイン・ブロックの説明:

[PL Block 0] – ブロック ID が 0 の PL ブロック

Method: Join – ブロックは、ジョインです

Type: Merge Join – ジョインの種類はマージです

Factors: (1) tb_staff.id = tb_salary.id – ジョインブロックを使用して、ジョインフィルタ t1.c2 = t2.c2 を適用します

I/O Cost: 8.5—試算した I/O コストは 8.5 ページです

CPU Cost: 573.8 – 試算した CPU コストは 573.8 ページです

Sub Cost: 231.6 – 試算した PL ブロックの副ブロックのためのコストの合計

Result Rows: 500.0 – 試算したジョイン・ブロック後の結果行

Sub Block 1: [PL Block 1] – ブロックの最初の子は[PL Block 1]にリンクしています

Sub Block 2: [PL Block 2] – ブロックの 2 番目の子は[PL Block 2]にリンクしています

ソート・ブロックの説明：

[PL Block 1] – ブロック ID が 1 の PL ブロック

Method: Sort – ソート・ブロック

I/O Cost: 4.2 – 試算した I/O コストは 4.2 ユニットです

CPU Cost: 274.4 – 試算した CPU コストは 274.4 ユニットです

Sub Cost: 120.0 – 試算した PL ブロックの副ブロックのためのコストの合計

Result Rows: 1000.0 – 試算したソート・ブロック後の結果行

SUB Block: [PL Block 3] – このブロックの子ブロックは[PL Block 3]にリンクします

ユーザーが直面する最も一般的なケースを列記しました。ダンプ計画では多くの変更が明らかですが、全て同じ要素、I/O コスト、CPU コスト、結果行で構成されています。ダンプ計画が複雑すぎる場合、他の手法を試すために前述した構文ベースの最適化を使用して下さい。

A. dmconfig.ini のキーワード

A.1 概要

データベース・エンジンの起動時や、データベース・サーバーへの接続時に種々のパラメータが初期化され、DBMaster の環境が設定されます。これらのパラメータは、ASCII テキストファイル **dmconfig.ini** から読み込まれます。このファイルは、DBMaster の環境を設定するキーワードと対応する値から構成されています。このファイルは ASCII 形式なので、必要に応じてテキストエディタを使ってパラメータを変更することができます。

ほとんどのキーワードは、データベースを起動する時に必要になります。データベース起動後に変更したキーワードは、データベースを再起動するまで有効にはなりません。

一方、ユーザーがデータベースに接続するときに必要になるキーワードも幾つかあります。これらのキーワードは、サーバーが起動した後も、接続コマンドを実行する前であれば、パラメータの値を変更し、新しい設定がそのセッションで有効になります。

環境設定パラメータは、DBMaster のパフォーマンスに重要な役割を果たします。このため、各パラメータの効果に注意し、DBMaster をスムースに作動させる最適値を見積る必要があります。又、データベースのファイル同様、定期的に **dmconfig.ini** ファイルのバックアップを取ることが理想的です。

A.2 dmconfig.ini ファイル形式

dmconfig.ini ファイルは ASCII テキストファイルです。ASCII テキストファイルを開いたり、保存したりすることができるテキストエディタで編集することができます。dmconfig.ini ファイルは、複数のデータベース・セクションで構成されます。各セクションは、ある特定のデータベースを起動するために用いられる環境設定情報の集まりです。各セクションは、セクション名で始まり、キーワードと値のリストが続きます。

➲ 例

dmconfig.ini ファイルの形式を以下に示します。

```
[セクション名 1]
キーワード 1 = 値 1          ; コメントを書きます
キーワード 2 = 値 2
        .
        .
[セクション名 2]
キーワード 3 = 値 3 値 4      ; 空白かコンマで値を区切ります
キーワード 4 = 値 5
        .
        .
```

セクション名

各セクション名は、それぞれデータベース名に対応しています。データベースは、起動するときに対応するセクションにある環境設定オプションを使用します。セクション名は、([ )で始まりデータベース名が続き( ])で終わります。([ )は行の先頭に置きます。

キーワード

セクション名の後に、キーワードとその値のリストが続きます。キーワードと値は、セクション名に対応するデータベースが起動するときに使用されます。キーワード=値という文字列で、各キーワードに値を割り当てます。キーワードによって、その値は整数または文字列の値をとります。

コメント

セミコロン(;)の後ろの文字列や記号は、コメントとみなされ無視されます。

➲ 例

dmconfig.ini ファイルの実例は以下のようになります。

```
[SDB]
DB_DbFil=SDB.DB
DB_JnFil=SDE.JNL
DB_SMode=1              ;ノーマル起動モード

DB_BMode=1
DB_BkSvr=1
DB_BkTim=96/03/19 00:00:00
DB_BkItv=7-00:00:00
DB_NBufs=100
DB_NJnlB=200
DB_MaxCo=100
DB_JnlSZ=20000
file1=SDE.FIL 40

[EMP]
DB_DBFIL=EMP.DB
DB_JNFIL=EMP.JNL
DB_SMode=1              ;ノーマル起動モード
DB_NBufs=100
DB_NJnlB=400
DB_MaxCo=100
DB_JnlSZ=20000
file1=EMP.FL1 100
file2=EMP.FL2 200
```

この例では、dmconfig.ini に SDB データベースと EMP データベースの 2 つのセクションがあります。

A.3 dmconfig.ini のディレクトリ

UNIX システムの場合、DBMaster は **dmconfig.ini** ファイル用に 3 つのディレクトリを探します。

➲ **ディレクトリと検索順序は以下のとおりです。**

1. 現在のディレクトリ
2. 環境変数DBMASTERで指定したディレクトリ
3. インストールディレクト：~DBMaster/Version

Microsoft Windows プラットフォームの場合、**dmconfig.ini** ファイルは Windows をインストールしたディレクトリに格納されます。これは典型的な **C:\DBMaster*Version*** です。
データベースの起動時に、DBMaster はデータベースに対応するセクションがある **dmconfig.ini** ファイルを見つけて、リストのディレクトリをスキャンします。セクション名のある **dmconfig.ini** ファイルを見つけると、そのセクションに定義されたキーワードが使用されます。最初のディレクトリに **dmconfig.ini** ファイルにセクション名が無い場合は、セクション名が見つかるまで、続くディレクトリで **dmconfig.ini** ファイルのサーチを続けます。

A.4 キーワードの初期設定値

DBMaster にパラメータが必要な時は、**dmconfig.ini** ファイルの該当セクションで対応するキーワードが検索されます。ユーザー定義ファイルを除いて、セクション名とキーワードは、大文字と小文字を区別せずにパターンマッチします。**dmconfig.ini** にキーワードが存在しない場合は、キーワードの初期設定値が使われます。ほとんどのキーワードに初期設定値があります。詳細については、*dmconfig.ini キーワード参照*の節をご覧下さい。

A.5 dmconfig.ini を作成する

データベースを作成する前に、テキストエディタを使用して **dmconfig.ini**

にデータベースのセクションを作成し、データベース作成時にパラメータが有効になるようにします。データベース作成時に対応セクションが何処にも無い場合、最初に見つけた dmconfig.ini ファイルに自動的に対応セクションが設けられます。dmconfig.ini ファイルそのものが存在しない場合は、新たに dmconfig.ini が生成されます。従って、データベースを起動する際には、データベースの対応セクションが必ずあります。見つからない場合は、エラーを返します。

A.6 キーワード参照

DB_ATCMT=<値>

このキーワードは、自動コミットモードの ON または OFF を指定します。1 に設定すると自動コミットモードを ON にし、0 に設定すると自動コミットモードを OFF にします。自動コミットモードが ON のときは、各 SQL 文を実行した後に自動的に COMMIT WORK 文が発行されます。このキーワードはクライアント側で設定します。

初期値： 1

有効値の範囲： 0、1

関連キーワード： DB_LTimO

使用する場所: クライアント側

DB_ATRMD=<値>

このキーワードは、データベースが非同期表レプリケーションのソースデータベースとして指定します。この値を 1 にセットすると、ソース表の操作のログをとると同時にディストリビュータデーモンを使用可能にします。つまり、レプリケーションのソースデータベースになります。

初期値: 0

有効値の範囲 : 0、1

関連キーワード: DB_EtrPt、RP_LgDir

使用する場所: サーバー側

DB_BbFil=<文字列>

このキーワードは、システム BLOB ファイル名を指定します。ファイルは、BLOB データの挿入時に必要に応じて拡張されます。

初期値： データベース名.SBB。例、db.SBB。

有効値の範囲：文字列の長さ < 256

関連キーワード： DB_DbDir、DB_DbFil、DB_UsrBb、DB_UsrDb

使用する場所: サーバー側

DB_BfrSz=<値>

このキーワードは、BLOB フレームのサイズをキロバイトで指定します。このキーワードは、データベース作成時に使用されます。

初期値： 32 KB

有効値の範囲： 8-256

関連キーワード： DB_BbFil

使用する場所: サーバー側（データベース作成時のみ）

DB_BkChk=<値>

このキーワードは、完全バックアップと差分バックアップの前にデータベースをチェックするかどうかを指定します。このキーワードを 0 に設定する場合、完全バックアップと差分バックアップの前に DBMaster はデータベースをチェックしません ; 1 に設定する場合、完全バックアップと差分バックアップの前に DBMaster はデータベースをチェックします。データベースに障害が発生する場合、バックアップサーバーはエラーメセッジを出力してバックアップを終了することになります ; 値が 2 の場合、完全バックアップと差分バックアップの前に DBMaster はデータベースをチェックします。データベースが破損している場合、バックアップサーバーはエラーメセッジを出力して、BKDIR/BADDB ディレクトリに破損したデータベースをバックアップし続けます。バックアップサーバーは障害が発生しているデー

タベースであると検知すると、一度だけバックアップします。その後データベースをチェックして、問題がない場合、バックアップサーバーは BKDIR/BADDB ディレクトリにある正常でないバックアップを削除し、正常なバックアップをし続ます。さもなければ、エラーメッセージのみを出力し、このバックアップを停止します。

増分バックアップファイルは常に前回の完全バックアップ或いは差分バックアップの後に出力されます。したがって、データベースが損害されることを検知する場合、増分バックアップファイルは BKDIR/BADDB ディレクトリに置かれ、データベースが OK になる場合、BKDIR ディレクトリに置かれます。

初期値: 0

有効値の範囲: 0、1、2

関連キーワード: DB_DbKtv、DB_DbKmx

使用する場所: サーバー側

DB_BkCmp=<値>

このキーワードは、コンパクトバックアップモードを使用するかを指定します。ジャーナルファイルにある各ジャーナルブロックが必ずしもバックアップするわけではありません。このキーワードを 1 に設定すると、バックアップサーバーは、バックアップ必要のジャーナルブロックのみをバックアップしてディスクを節約します。詳細につきましては、15 章の *データベースのバックアップ、リカバリ、リストア*をご参照下さい。

初期値： 1

有効値の範囲： 0、1

関連キーワード： DB_BkSvr

使用する場所: サーバー側

DB_BKDIR=<文字列>

このキーワードは、データベースの最新バックアップシーケンスを格納される既存ディレクトリ或いは一グループのディレクトリ（32 以下）を指定します。有効なバックアップシーケンスは一つの完全バックアップ、一つまたは多数の差分バックアップ（オプション）、シリーズの増分バックアップ（オプション）から構成します。**DB_DbDir** 以外のディレクトリを指定することも可能です。「リカバリ、バックアップ、リストア」の章も参照して下さい。

有効値の範囲： 文字列の長さ < 256

関連キーワード： DB_BkSvr、DB_Bmode

使用する場所: サーバー側

➲ **例**

< BKDIR n >: n のバックアップパス

< SIZE n >: n のバックアップパスのサイズ

DB_BKDIR = <BKDIR 1> <サイズ 1> < BKDIR 2> <サイズ 2> < BKDIR 3> <サイズ 3>…

```
[MYDB]
….
DB_BKDIR = /home/usr/dbmaster/bk 5000 /home2/backup 1000
….
```

/home/usr/dbmaster/bk が一杯のとき、ファイルを /home2/backup にバックアップします。

DB_BKFOM=<値>

このキーワードは、FO（ファイルオブジェクト）のバックアップモードを指定します。FO は、データベースの完全バックアップの際にのみバックアップされます。値を 0 に設定すると、FO のバックアップ機能を OFF にします。値 1 に設定すると、システム FO がバックアップされます。値 2 に設定すると、システム FO とユーザーFO の両方がバックアップされます。

初期値: 0

有効値の範囲: 0: FO をバックアップしない。

1: システム FO をバックアップする。

2: システム FO とユーザーFO をバックアップする。

関連キーワード: DB_BkSvr、DB_FBKTm、DB_FBKTV、DB_BkDir

使用する場所: サーバー側

DB_BKFRM=<値>

このオプションは、バックアップサーバーが増分バックアップに名前を付ける際に使用するフォーマットを指定することができます。バックアップ・ファイル名フォーマットは*<I><Timestamp><_><DB_BKFRM>*です。タイムスタンプは 10 桁の有効な時間数字データで、また<DB_BKFRM>にはテキスト定数も、特殊な文字列で構成されたフォーマット・シーケンス（例：エスケープ・シーケンス）も含むことができます。

増分バックアップのファイル名は、少なくとも次の 3 つの特殊な文字列で構成されています。完全バックアップ ID、データベース名、バックアップ ID 番号です。バックアップ・サーバーは、バックアップ・シーケンスに増分ファイルの名前を付ける際に完全バックアップ ID を割当てます。データベースをリストアする時、DBMaster は完全バックアップ ID を使用してこれに属する増分バックアップファイルを適切に再生成します。バックアップ ID 番号は、バックアップ・シーケンスにある増分バックアップの相対位置を識別します。

フォーマット・シーケンスは、エスケープ文字、サイズ、フォーマット文字の 3 つの部分から構成されています。有効なフォーマット・シーケンスは次のとおりです。

%［*n*］Y—ジャーナル・ファイルがバックアップされた年。

%［*n*］M—ジャーナル・ファイルがバックアップされた月。

%［*n*］D—ジャーナル・ファイルがバックアップされた日。

%［*n*］B—バックアップ識別番号。

%［*n*］N—ジャーナルファイルの対応するデータベースの名称。

➲ **例**

```
DB_BkFrm = %N.%B
```

データベース名が test1 の場合、増分バックアップ・ファイル名は test1.1, test1.2…になります。

初期値: *I<timestamp>*_%4N%4B.jnl

関連キーワード: DB_BkSvr、DB_BkTim、DB_BkItv

使用する場所: サーバー側

DB_BKFUL=<値>

このキーワードは、増分バックアップの実行を誘発する、ジャーナルファイルの書き込み密度（%）を指定します。値を 0 にすると、ジャーナルファイルが 100%のときにバックアップ・サーバーが起動します。50-100 の範囲内の値は、全てのジャーナルファイルの合計使用量が指定した割合に達したときにバックアップ・サーバーを起動させます。例えば、2 つのジャーナルファイルに各々500 ジャーナル・ブロックあり、**DB_BkFul** を 80 に設定する場合、**500×2×0.8＝800** ブロックを使用する毎に、バックアップ・サーバーが自動的に増分バックアップをします。

初期値： 90

有効値の範囲： 0、50 ～ 100

関連キーワード： DB_BkSvr、DB_BkTim、DB_BkItv

使用する場所: サーバー側

DB_BKITV=<文字列>

このキーワードは、自動的にバックアップを取るときの時間間隔を指定します。詳細につきましては、**DB_BkTim** を参照してください。

初期値： なし（DB_BkItv を設定していない場合、バックアップする計画が無いです）

関連キーワード： DB_BkSvr、DB_BkTim、DB_BMode

使用する場所: サーバー側

DB_BKODR=<文字列>

このキーワードは、ディレクトリ或いは一グループのディレクトリ（32 以下）を指定します。このディレクトリは古くて、バックアップシーケンスを保存するため使用されます。有効なバックアップシーケンスは一つの完全バックアップ、一つまたは多数の差分バックアップ（オプション）、シリーズの増分バックアップ（オプション）から構成します。詳細は 15 章の *リカバリ*、*バックアップ*、*リストア*を参照して下さい。

初期値: なし

有効値の範囲: 文字列の長さ < 256

関連キーワード: DB_BkSvr、DB_BMode、DB_FBkTm、DB_FBkTv

使用する場所: サーバー側

➲ **例**

DB_BKODR = <BKDIR 1> <サイズ 1> < BKDIR 2> <サイズ 2> < BKDIR 3> < サイズ 3>…

< BKDIR n > : n のバックアップパス

< SIZE n > : n のバックアップパスのサイズ(単位あたり 8 KB)

```
[MYDB]
….
DB_BKODR = /home/usr/dbmaster/bk 5000 /home2/backup 1000
……
```

DB_BKRTS=<値>

このキーワードはバックアップサーバがフルバックアップをする時に読み取り専用のテーブルスペースを含むかどうかを指定します。デフォルト値は 1 で、読み取り専用のテーブルスペースファイルをバックアップします。読み取り専用のテーブルスペースファイルがバックアップ済みな時、

DB_BkRTs に 0 を割り当てることができます。キーワードに 0 を割り当てると、テーブルスペースを読み取り専用テーブルスペースに設定した後にフルバックアップを行う必要がありますが、ご注意ください。最新のファイルをバックアップしない場合、データベースに読み取り専用テーブルスペースが含まれるフルバックアップを復元しようとすると、深刻なエラーが発生することがあります。 1 のデフォルト値を使用してください、そうしなければ、ここには特有な理由で別のことをします。

初期値:1

有効値の範囲: 0、1

関連キーワード: DB_BkSvr

使用する場所:サーバー側

DB_BKSPM=<値>

このキーワードは、完全バックアップを実行している間にストアドプロシージャをバックアップするかどうかを指定します。その値を 1 または 0 に設定でき、デフォルト値は 0 です。**DB_BkSPm** に 0 を割り当てる場合は、ストアドプロシージャがバックアップされていないと表し、**DB_BkSPm** に 1 を割り当てる場合は、あらゆるストアドプロシージャがバックアップされると表します。

初期値: 0

有効値の範囲: 0、1

使用する場所: サーバー側

DB_BKSVR=<値>

このキーワードは、データベースを起動した後、バックアップサーバの状態を制御するため使用されます。**DB_BKSvr** は 0 にすると、バックアップサーバーはインアクティブです ; **DB_BKSvr** は 1 にすると、バックアップサーバーはアクティブです。バックアップサーバーをアクティブするため、ユーザーは **dmconfig.ini** ファイルに **DB_BkSvr** を 1 に設置する或いはデータベースを起動した後、*call setsystemoptio('bksvr','1')*を使用して BkSvr を変更します。詳細な情報は 15 章の *リカバリ、バックアップ、リストア*を参照して下さい。

初期値: 0

有効値の範囲： 0、1

関連キーワード： DB_BkCmp、DB_BkDir、DB_BkFul、DB_BkTim、DB_BkItv

使用する場所: サーバー側

DB_BKTIM=<文字列>

このキーワードは、**DB_BkItv** と共にバックアップ・サーバーのスケジュールを指定します。**DB_BkTim** は、バックアップ・サーバーが最初に増分バックアップを取る日時を指定します。増分バックアップは、**DB_BkItv** で指定された時間間隔で取られます。

➲ **例**

```
DB_BkTim = 96/05/01 00:00:00    ;1996 年 5 月 1 日からバックアップを開始
DB_BkItv = 1-12:30:00           ;1 日と 12 時間 30 分のバックアップ間隔
```

DB_BkTim と **DB_BkItv** は、バックアップ・サーバーが起動される場合のみ参照できます。詳細につきましては、15 章の*データベースのバックアップ、リカバリ、リストア*をご参照下さい。

初期値: なし （DB_BkTim を設定していない場合、バックアップする計画が無いです)

有効値の範囲：1970-01-01 00:00:01 ~ 2037-12-31 23:59:59

関連キーワード： DB_BkItv、DB_BkSvr、DB_BMode

使用する場所: サーバー側

DB_BKZIP=<値>

このキーワードは、バックアップサーバがフルバックアップ時にバックアップファイルを圧縮するかどうかを指定します。

キーワードを 1 に設定すると、フルバックアップで圧縮ファイルが生成します。キーワードを 0 に設定すると、フルバックアップでファイルを圧縮しません。

初期値: 0

有効値の範囲: 0、1

関連キーワード: DB_BkSvr、DB_BkCmp、DB_BkDir、DB_BkFul、DB_BkTim、DB_BkItv

使用する場所: サーバー側

DB_BMODE=<値>

このキーワードは、データベースのバックアップ・モードを指定します。0 は NON-BACKUP モード、1 は BACKUP-DATA モード、2 は BACKUP-DATA-AND-BLOB モードです。このキーワードは増分バックアップにのみ役に立ちます。1 或いは 2 に設定すると、ユーザーは増分バックアップをすることができます。値は 0 を設定する場合、ジャーナルは一部のジョブを保存します。詳細は 15 章の*リカバリ*、*バックアップ*、*リストア*を参照して下さい。

初期値: 0

有効値の範囲： 0 、 1 、2

関連キーワード： DB_BkSvr

使用する場所: サーバー側

DB_BROWS=<値>

このキーワードは、SELECT 文のロックの種類を指定します。値を 0 にセットすると、DBMaster が SELECT 文の結果セットに S ロックをかけることを意味します。1 にセットすると、SELECT 文の結果セットにロックしないことを意味します。この値は、データベースに接続している時に必要です。

初期値: 1

有効値の範囲: 0、1

使用する場所: クライアント側

DB_CBMOD=<値>

このキーワードは、トランザクションが終了した後のカーソルの振る舞いを指定します。1 はトランザクションのコミット後に全てのオープンカーソルをクローズします。2 と 3 はトランザクションのコミット後に全てのオープンカーソルをオープンのままにします。2 はトランザクションのコミット後に全てのロックが解放されます。3 は全ロックが解放されますが、排他ロックは共有ロックになります。いずれの場合も、トランザクションがアボートされた場合はカーソルをクローズします。

初期設定値: 2

有効値の範囲： 1、2、3

使用する場所: クライアント側

DB_CHKFL=<値>

このキーワードは、データベースがウォームスタートの方式で起動する際にユーザーファイルをチェックするかどうかを指定します。その値を 1 または 0 に設定でき、デフォルト値は 1 です。0 に設定する場合は、当該機能が無効になり、サーバーがユーザーファイルをチェックしませんが、1 に設定する場合は、当該機能が有効になります。データベースが起動時に、幾つかの構成アイテムまたはファイルが存在しない場合、警告メッセージが出力され、ユーザーに知らせますが、当該メッセージはファイル *DMEVENT.LOG* に記録されています。

初期値: 1

有効値の範囲: 0,1

使用する場所: サーバー側

DB_CLILCODE=<文字列>

このキーワードはクライアント側の言語コードを定義します。多言語データベースを使用して、またデータベースサーバの LCODE が UTF-8 のとき、

クライアント側では、UTF-8 のデータベースに接続するための任意のローカルコードを使用することができます。

サーバ側の LCODE が UTF-8 でない場合、CLILCODE の値はサーバ側のコードと同じでなければなりません。

SELECT GETSYSINFO('CLILCODE')コマンドを使用してクライアント側の言語コード設定状況を返します。

初期値: サーバ側の LCODE が UTF-8 でない場合、初期値はサーバ側の LCODE に従います。サーバ側の LCODE が UTF-8 の場合、Windows 環境のクライアント側は Windows 言語環境を CLILCODE のデフォルトの値として取得し、Windows 言語環境が未対応のものであれば ASCII となります。

Linux プラットフォームのクライアント側では環境変数 **LANG** を CLILCODE のデフォルトの値にとります。LANG が設定されていないまたは未対応の場合は ASCII となります。**LANG** の形式は *lang* または *lang.lcode* となり、lang.lcode は *lcode* を直接取得するため;その他の場合、DBMaster はその言語のデフォルトの *lcode* を取得し、CLILCODE のデフォルトの値とします。

有効値の範囲：

ASCII (英語)
BIG5 (繁体中国語)
Shift-JIS (日本語 Shift-JIS + 半角文字)
GBK (簡体中国語)
ISO-8859-1 (ラテン 1 コード)
ISO-8859-2 (ラテン 2 コード)
ISO-8859-5 (キリル コード)
ISO-8859-7 (ギリシアコード)
EUC-JP (日本語コード)
GB18030 (簡体中国語)
UTF-8 (UTF-8)

ISO-8859-{3、4、9、10、13、14、15、16}、KOI8-R、KOI8-U、 KOI8-RU、CP{1250、1251、1252、1253、1254、1257}、 CP{850、866}、Mac{ローマ語、中央ヨーロッパ語、アイスランド語、クロアチア語、ルーマニア語 }、

Mac{キリル語、ウクライナ語、ギリシア語、トルコ語 }、 Macintosh(ヨーロッパ言語)

ISO-8859-{6、8},、CP{1255、1256},、CP862、 Mac{ヘブライ語、アラビア語} (セム言語)

CP932、ISO-2022-JP、ISO-2022-JP-2、ISO-2022-JP-1(日本語)

EUC-CN、CP936 、EUC-TW、CP950(中国語)

EUC-KR、CP949 、JOHAB(韓国語)

Georgian-Academy、Georgian-PS(グルジア語)

KOI8-T(タジク語)

PT154(カザフ語)

TIS-620、CP874 、MacThai(タイ語)

MuleLao-1、CP1133(ラオス語)

VISCII、TCVN 、CP1258(ベトナム語)

関連キーワード：DB_LCode、DB_ErrLCODE

使用する場所：クライアント側

➲ **例 1**

```
DB_CliLCODE=GBk;
```

➲ **例 2**

```
DB_CliLCODE= PT154;、
```

DB_CmChe=<値>

このキーワードは、クライアントのコマンドキャッシュを ON にするかどうかを指定します。この値を 1 に設定する場合、クライアントのコマンドキャッシュを起動するのを表し、0 に設定する場合、そのコマンドキャッシュを無効にすることを意味します。クライアントのコマンドキャッシュが運行する時、この前に実行した SQL コマンドに関する情報がキャッシュに保存され、その後の実行する SQL コマンドと同じ場合に、それを再利用

することができます。この方法により、パフォーマンスの向上が達成されます。

初期値: 0

有効値の範囲: 0、1

使用する場所: クライアント側

DB_CTBLM=<値>

このキーワードはテーブル作成時のデフォルトのロックモードを指定します。

このキーワードを 0 に設定する場合、デフォルトのロックモードはページレベルのロックモードであるのを表します。この場合で、テーブル作成時にロックモードを指定しないと、ロックモードはページレベルとなります。

値を 1 に設定する場合、デフォルトのロックモードが行レベルのロックモードであるのを表します。この場合で、テープル作成時にロックモードを指定しないと、ロックモードは行レベルとなります。

初期値: 1

有効値の範囲: 0、1

使用する場所: サーバ側

DB_CTIMO=<値>

このキーワードは、クライアントがデータベースサーバーに接続する際の接続タイムアウトの時間（秒数）を指定します。データベースが起動していない場合やサーバーの IP アドレスが誤っている場合、ユーザーは接続がタイムアウトするまで、しばらく待つことを余儀なくされます。ユーザーはこのキーワードの値を設定することによって、待ち時間を短くします。

初期値: 5（秒）

有効値の範囲： 5 ~ 1:00:00（1 時間）

使用する場所: クライアント側

DB_DAIFM=<文字列>

このキーワードは、SQL 文の日付入力フォーマットを指定します。詳細については、*ODBC プログラマーガイド*— 付録 B を参照してください。

初期値： なし（全ての入力フォーマットを受理します）

有効値の範囲： mm/dd/yy

mm-dd-yy

dd/mon/yy

dd-mon-yy

mm/dd/yyyy

mm-dd-yyyy

yyyy/mm/dd

yyyy-mm-dd

dd/mon/yyyy

dd-mon-yyyy

dd.mm.yyyy

関連キーワード： DB_DaoFm

使用する場所: クライアント側またはサーバー側（クライアント側に優先権がある）

DB_DAOFM=<文字列>

このキーワードは、SQL 文の日付出力フォーマットを指定します。詳細については、詳細については、*ODBC プログラマーガイド*— 付録 B を参照してください。

初期値： yyyy-mm-dd

有効値の範囲： DB_DaiFm

関連キーワード： DB_DaiFm

使用する場所: クライアント側またはサーバー側（クライアント側に優先権がある）

DB_DBDIR=<文字列>

このキーワードは、データベースのファイルを格納するディレクトリを指定します。ディレクトリは、相対パスまたは絶対パスで指定します。

DBMaster には、7 種類のファイルがあります：

- **DB_DbFil**で定義されるシステム・データファイル
- **DB_UsrDb**で定義される初期設定ユーザー・データファイル
- **DB_JnFil**で定義されるシステム・ジャーナルファイル
- **DB_BbFil**で定義されるシステムBLOBファイル
- **DB_UsrBb**で定義される初期設定ユーザーBLOBファイル
- **DB_TpFil**で定義されるシステム一時ファイル
- ユーザー定義ファイル

これらのキーワードは、絶対パス名、相対パス名、ファイル名で定義することができます。パス名で定義した場合、定義したファイルを参照するためにそのパス名が使用されます。ファイル名で定義した場合、DB_DbDir キーワードをサーチします。DB_DbDir が定義されていれば、その文字列をファイル名の前に付けてファイルを参照します。DB_DbDir が定義されていない場合、ファイルは現在のディレクトリにあるものとみなされます。

➲ **例 1**

```
[DB1]
DB_DbDir = /disk1/db
DB_DbFil = mydb1
DB_JnFil = /disk2/usr/DB1.JNL
```

物理ファイル名は次のようになります：

```
DB_DbFil -- /disk1/db/mydb1
DB_JnFil -- /disk2/usr/DB1.JNL
```

```
DB_BbFil -- /disk1/db/DB1.BB （初期設定のファイル名を使用）
```

➲ **例 2**

```
[DB2]
DB_DbFil = mydb2
DB_JnFil = /disk2/usr/DB2.JNL
```

物理ファイル名は次のようになります：

```
DB_DbFil -- mydb2 （現在のディレクトリにある）
DB_JnFil -- /disk2/usr/DB2.JNL
DB_BbFil -- DB2.BB （現在のディレクトリにある）
```

初期値： なし（現在のディレクトリ）

有効値の範囲： 文字列の長さ < 256

関連キーワード： DB_DbFil、DB_JnFil、DB_BbFil、DB_TpFil、DB_UsrDb、DB_UsrBb

使用する場所: サーバー側

DB_DBFIL=<文字列>

このキーワードは、システム・データファイルの物理ファイル名（オペレーティング・システムが使用するファイル名）を指定します。

初期値： データベース名.SDB。例、db.SDB

有効値の範囲： 文字列の長さ < 256

関連キーワード： DB_BbFil、DB_DbDir、DB_UsrBb、DB_UsrDb

使用する場所: サーバー側

DB_DBKMX=<値>

このキーワードは、完全バックアップをした後、差分バックアップの最大数を指定します。仮に差分バックアップの数が **DB_DbKmx** を上回る場合、バックアップサーバーは古い差分バックアップを削除します。値を 0 に設定した場合、このキーワードは使用できません、つまり、差分バックアップの数は無制限です。

初期値： 10

有効値の範囲: 0 ~ 65535

関連キーワード: DB_ FBkTm、DB_ FBkTv、DB_DbKtv

使用する場合：サーバー側

DB_DBKTV=<文字列>

このキーワードは差分バックアップの時間間隔を指定します。はじめの差分バックアップは DB_FBKTM + DB_DBKTV です。フォーマットは **nDays-hh:mm:ss** です。詳細な情報は **DB_FBkTm** と **DB_FBkTv** をご覧下さい。

初期値: 無し(DB_DbKtv を設定していない場合、完全バックアップの計画が無いです)

有効値の範囲: 0-00:00:01 ~ 24854-23:59:59

関連キーワード: DB_ FBkTm、DB_ FBkTv、DB_DbKmx

使用する場合：サーバー側

DB_DSCMT=<値>

このキーワードは、アプリケーションがデータベースから切断されたときにトランザクションをコミットするかどうかを指定します。値を 0 に設定した場合、クライアントのアプリケーションが SQLDisconnect コマンドを発行しても、DBMaster はデータベースから切断される前にコミットを発行しません。切断前にオートコミットを設定しなかった場合やコミットを発行しなかった場合、トランザクションはロールバックします。

値を 1 に設定した場合、クライアントのアプリケーションが SQLDisconnect コマンドを発行すると、DBMaster もデータベースから切断される前にコミットを発行します。切断前にオートコミットを設定しなかった場合やコミットを発行しなかった場合もトランザクションはコミットします。

初期値: 0

有効値の範囲: 0 (データベースからの接続を切断する前にコミットしない)

1 (データベースからの接続を切断する前にコミットする)

使用する場所: クライアント側

DB_DTCLT=<値>

このキーワードは、DBMaster のクライアントサイドとサーバーサイドの最大アイドルタイムアウト時間の間隔を指定します。このキーワードの初期値は 0 ですが、即ち当該機能はディフォルトでは無効になっています。クライアントサイドの機械の電源が突然に切れる場合またはネットワークが正常に作動していない場合に、サーバーはクライアントに割り当てたリソースを解放しないです。

当該キーワードを設定することによって、この問題を解消することができます。サーバーはクライアントが機能していないことを自動検出すると、クライアントの全てのリソースを解放します。エリアネットワークが安定しないと、DBA はクライアントアイドル・タイムアウトを設定します、サーバーは期間にクライアントを検出します。クライアントが機能していないとクライアントの全てのリソースを開放します。

DBMaster クライアントとサーバー側の間に最大なアイドルタイムは **DB_DtClt**（クライアントで設置）と **DB_ITimo**（サーバー側で設置）によってサーバーがこの二つの値にもっと小さい値を選択してチェックタイムとします、ルールは以下の通りです：

- ユーザーはこの二つのキーワードに一つのキーワードを設置すると、DBMasterクライアントとサーバー側のコミュニケーションのアイドルタイムは設置されたキーワードの値になります。つまり、ユーザーはクライアントで**DB_DtClt**を設定しますが、サーバーで**DB_Itimo**を設定しないと、DBMasterクライアントとサーバー側のコミュニケーションの最大のアイドルタイムはキーワード**DB_DtClt**の設置値になります。ユーザーはクライアントで**DB_DtClt**を設定しなく、サーバー側で**DB_ITimo**を設置すれば、DBMasterクライアントとサーバー側のコミュニケーションの最大のアイドルタイムはキーワード**DB_Itimo**の値になります。

- ユーザーは同時にこの二つのキーワードを設定すると、以下の状況が起こします：

  a) **DB_DtClt**の値は**DB_Itimo**の値より小さいと、DBMasterのクライアントとサーバーの側のコミュニケーションの最大のアイドルタイムはキーワード**DB_DtClt**の値になります。

  b) **DB_DtClt**の値は**DB_Itimo**の値より大きいと、DBMasterのクライアントとサーバーの側のコミュニケーションの最大のアイドルタイムはキーワード**DB_ITimo**の値になります。同時に**DB_DtClt**の設置値は**DB_Itimo**の値をリセットされます。

- ユーザーはいずれクライアントで**DB_DtClt**を設定しなく、サーバーでも**DB_Itimo**を設定しないと、DBMasterクライアントとサーバー側のコミュニケーションの最大のアイドルタイムは無制限になります。

初期値： 0（秒）- 使用不能

有効値の範囲： 0、5 ~ 1:00:00（1 時間）

関連キーワード: DB_ITimO

使用する場所: クライアント側

DB_ERMRv=<文字列>

このキーワードは、データベースにエラーが発生した際に生成される e－mail アドレスのリストが通知を受けることを指定します。8 つまでの e-mail アドレスを指定できます。各アドレス文字列は、カンマで区切ります。参加者のアドレスを指定しない場合、エラーレポート・システムは無効になり、e-mail は生成されません。

初期値: なし

関連キーワード: DB_ ERMSv

使用する場所: サーバー側

DB_ERMSv=<文字列>

このキーワードは、e-mail のメッセージを送信するための SMTP サーバーを指定します。SMTP サーバーを 1 つのみ指定します。SMTP サーバーが指定されず、e-mail のアドレスのみ指定されている場合、このキーワードには「ローカルホスト」が使用されます。

初期値: ローカルホスト

関連キーワード: DB_ERMRv

使用する場所: サーバー側

DB_ErrLCODE=<文字列>

このキーワードはクライアントで表示されるエラーメッセージの文字セットを設定します。多言語データベース環境ではクライアント側独自の出力エラーメッセージのロケールコードを設定できます。エラー表は *DBMaster5.4/shared/locale/locale.lang/*ディレクトリにあります。この値はローカル設定と文字セットと組み合わせて使用されます。

SELECT GETSYSINFO('ERRLCODE')コマンドでクライアント側のエラーメッセージ文字セットを返すことができます。

➲ **例**

```
DB ErrLCODE='ja';
DB ErrLCODE='ja.EUC JP';
DB_ErrLCODE='ja.UTF-8';
```

初期値: Windows ではレジストリから LANG の値を取得しデフォルトの値に適用します。レジストリに言語設定がない場合、オペレーションシステムの言語コードに従います。Linux のクライアントではデフォルトの値は環境変数の LANG に依存します。LANG が未設定または未対応の場合、DBMaster はエラーメッセージの文字セットを ASCII にします。

ロケール設定値: en、ja、zh_CN、zh_TW

使用可能文字セット：

ASCII (英語)
BIG5 (繁体中国語)
Shift-JIS (日本語 Shift-JIS + 半角文字)
GBK (簡体中国語)
ISO-8859-1 (ラテン 1 コード)
ISO-8859-2 (ラテン 2 コード)
ISO-8859-5 (キリル コード)
ISO-8859-7 (ギリシアコード)
EUC-JP (日本語コード)
GB18030 (簡体中国語)
UTF-8 (UTF-8)

有効値：

EN_XXX
EN_XXX.ASCII
EN_XXX. ISO-8859-1
EN_XXX.ISO-8859-2
EN_XXX.ISO-8859-5
EN_XXX.ISO-8859-7
EN_XXX.UTF-8
EN
EN.ASCII
EN.ISO-8859-1
EN.ISO-8859-2
EN.ISO-8859-5
EN.ISO-8859-7
EN.UTF-8
JA_XXX
JA_XXX.SHIFT-JIS
JA_XXX.SHIFT_JIS
JA_XXX.UTF-8
JA_XXX.EUCJP
JA_XXX.EUC-JP
JA

JA.SHIFT-JIS
JA.SHIFT_JIS
JA.UTF-8
JA.EUCJP
JA.EUC-JP
ZH_CN
ZH_CN.GBK
ZH_CN.UTF-8
ZH_CN.GB18030
ZH_TW
ZH_TW.BIG5
ZH_TW.UTF-8

関連キーワード：DB_LCode、DB_CliLCODE

使用する場所：クライアント側

DB_ETRPT=<値>

このキーワードは、高速非同期表レプリケーションに使用されます。データベース・サーバーにサブスクライバ・デーモンが取り付く TCP/IP ポート番号を整数で指定します。ソース・データベース側では、このキーワードは、ターゲット・データベースのサブスクライバにどのように接続するかを指示します。ターゲット・データベース側では、このキーワードは、サブスクライバ・デーモンを起動させます。

初期値: なし

有効値の範囲: 1024~65535

関連キーワード: DB_AtrMd、RP_LgDir

使用する場所: サーバー側(データベースのソースとターゲット両方)

DB_EXTHD=<値>

このキーワードは、ファイルが繰り返して拡張される際のしきい値を指定します。その値を－1 に設定する場合は、自動拡張表領域は常に最初のファ

イルから自動的に拡張されると表し、0 に設定する場合は、自動拡張表領域は常に最小のファイルから自動的に拡張されると表し、1~4294967296/DB_PgSiz、1M~4096M または 1G~4G に設定する場合は、自動拡張表領域はまずは最小のファイルから自動的に拡張し、現行ファイルのサイズが次の最小ファイルと DB_ExtHd の値の和を上回るまで、次の最小ファイルを拡張します。そうすると、パフォーマンス及びファイルサイズのバランスを取ることができます。

初期値: 100M

有効値の範囲: -1、0、1 ~ 4294967296/DB_PGSIZ、1M ~ 4096M、1G ~ 4G

関連キーワード: DB_ExtNp

使用する場所: サーバー側

DB_EXTNP=<値>

このキーワードは、自動拡張する表領域のサイズを定義します。自動拡張表領域が使い果たされた時、DBMaster は自動的に拡張します。この値は、拡張するページ数/フレーム数を指定します。

初期値: 20 (ページ/フレーム)

有効値の範囲: 1 ~ *32767*

使用する場所: サーバー側

DB_FBKTM=<文字列>

このキーワードは、**DB_FBkTv** と共にオンライン完全バックアップを実行するためにバックアップ・サーバーのスケジュールを定義します。**DB_FBkTm** は、バックアップ・サーバーが完全バックアップを実行する最初の日時を指定します。オンライン完全バックアップは、以降 **DB_FBkTv** で指定した間隔毎に実行されます。

➲ **例**

```
DB_FBKTm = 96/05/01 00:00:00    ;1996 年 5 月 1 日開始
DB_FBKTV = 1-12:30:00           ;1 日 12 時間半間隔
```

キーワード **DB_FBkTm** と **DB_FBkTv** は、バックアップ・サーバーにのみ使用されます。

初期値: なし

有効値の範囲：1970-01-01 00:00:01 ~ 2037-12-31 23:59:59

関連キーワード: DB_FBkTv、DB_BkSvr、DB_BkOdr

使用する場所: サーバー側

DB_FBkTv=<文字列>

このキーワードは、完全バックアップの時間間隔を指定します。詳細については、**DB_FBkTm** を参照して下さい。

初期値: なし (DB_FBkTv が設定されていない場合､完全バックアップをする計画がないです)

有効値の範囲：0-00:00:01 ~ 24854-23:59:59

関連キーワード: DB_BkSvr、DB_FBkTm、DB_BkOdr

使用する場所: サーバー側

DB_FltDb=<文字列>

このキーワードは、カラムが内部ストレージおよび値の範囲として 4 (REAL) バイトを使用するか 8 (DOUBLE) バイトを使用するかを指定します。タイプ名も REAL または DOUBLE に変わります。

値を 0 に設定すると、FLOAT カラムのストレージが 4 バイトに、タイプ名が REAL に指定されます。

値を 1 に設定すると、FLOAT カラムのストレージが 8 バイトに、タイプ名が DOUBLE に指定されます。

初期値: 1

有効値の範囲: 0、1

使用する場所: サーバー側

DB_FoDir=<文字列>

このキーワードは、ファイルオブジェクトを置く絶対パスを指定します。

➲ **例 1**

```
DB_FoDir = /usr/DBMaster/fileobj              ;UNIX の場合
```

➲ **例 2**

```
DB_FoDir = c:\DBMaster\fileobj                ;DOS の場合
```

➲ **例 3**

```
DB_FoDir = \\NTMachine\DBMaster\fileobj       ;Microsoft UNC 名の場合
```

ファイルオブジェクトは、**DB_FoDir** が設定されているときのみ挿入することができます。このキーワードはデータベースの起動時に使用されます。

初期値: なし（ファイルオブジェクトを挿入することはできません）

有効値の範囲： 文字列の長さ < 256

関連キーワード： DB_UsrFo

使用する場所: サーバー側

DB_ForcS=<値>

このキーワードは、データベースの強制起動を指定します。この値を 1 に設定すると、データベースの起動中にエラーが発生しても強制的に起動します。

初期値: 0

有効値の範囲： 0、1

関連キーワード： DB_SMode

使用する場所: サーバー側

DB_FORUX=<値>

このキーワードは、サーバー側の「select … for update」文のロックの種類を指定します。

DBMaster は、「select … for update」文の結果セットに U ロックをかけます。この値を 1 にセットする特殊なアプリケーションの場合は、「select … for update」文の結果セットに X ロックをかけさせます。この設定は、データベース起動時に必要です。

初期値: 0

有効値の範囲: 0、1

使用する場所: サーバー側

DB_FOSUB=<値>

このキーワードは、各システム・ファイルオブジェクトのサブディレクトリに格納されるファイルオブジェクトの最大数を指定します。サブディレクトリはキーワード **DB_FoDir** で指定するディレクトリに生成されます。ファイルオブジェクトの新しいサブディレクトリは、サブディレクトリにあるファイルオブジェクトの数がしきい値を超えた際に、自動的に生成されます。

➲ **例**

ファイルオブジェクトのサブディレクトリ当たりのファイル数を 500 に設定する：

```
DB_FoSub = 500
```

DB_FoDir が設定されている場合のみ、新しいシステム・ファイルオブジェクトにデータを挿入することができます。この設定は、データベース起動時に必要です。

初期値: 0 (ファイルオブジェクトは、DB_FoDir で指定したディレクトリに保存されます。)

有効値の範囲: 100 ~ 10000、0 (FO ディレクトリには、サブディレクトリがありません。)

関連キーワード: DB_FoDir

使用する場所: サーバー側

DB_FoTyp=<値>

このキーワードは、FILE データ型を ODBC データ型にマップするかを指定します。ODBC は、DBMaster でのみ使用する FILE データ型をサポートしていません。Borland Delphi や Microsoft Visual Basic のような開発ツールは、FILE データ型をサポートしません。このようなツールに FILE データ型のデータにアクセスさせる場合、DB_FoTyp を 1 にセットして DBMaster で FILE データ型を LONG VARBINARY にマップする必要があります。DB_FoTyp が 0 の場合は、マップしません。

初期値: 1

有効値の範囲：0 (マップしない)

1 (FILE データ型を LONG VARBINARY にマップする)

使用する場所: クライアント側

DB_GcChk=<値>

このキーワードは、グループ・コミットのトランザクション・プロトコルを初期化する TPS(transaction per second)の最小数を指定します。

DBMaster は、常に現在のトランザクション数をチェックします。1 秒あたりのトランザクション数は、1 秒あたりの sync 要求の数と同等です。DBMaster は、TPS のしきい値として **DB_GcChk** キーワードを使用します。サーバーの TPS がこのしきい値に達した時、グループ・コミット・プロトコルが作動します。TPS がしきい値以下の時、グループ・コミットは無効になります。例えば、**DB_GcChk** が 20 の場合、1 秒あたりのトランザクションが 20 を超えた場合、グループ・コミット・プロトコルが起動します。

DB_GcChk はキーワード、DBMaster に動的にグループ・コミット・プロトコルを動的に切り替えます。トランザクション・アクティビティは、非常

に頻繁な時と、非常に低い時があるように一定ではないので、不必要に待機が無いようにプロトコルを切り替えます。

初期値: 20

有効値の範囲: >0

関連キーワード: DB_GcWtm、DB_GcMxw

使用する場所: サーバー側

DB_GcMxw=<値>

このキーワードは、グループ・コミットを実行させるトランザクションの数を指定します。グループ・コミットを待機する最大待ち時間を指定する **DB_GcWtm** と共に使用します。

グループ・コミットのプロトコルを有効にする場合（詳細は **DB_GcChk** を参照のこと）、DBMaster は sync 要求ごとにチェックします。以下の条件のいずれかに合致する場合、tube sync プロセスが処理されます。或いはグループ・コミットが実行される前まで他の sync 要求を待機します

- トランザクションが**DB_GcWtm**キーワードで最大待機時間に達した場合
- グループ・コミットを待機しているトランザクションの数がDB_GcMxwキーワードで指定された値を超えた場合

➲ **例**

最大待機時間は 30 ミリ秒です。待機するトランザクションの最大数は 5 です。DBMaster は、少なくとも 1 つのトランザクションの待機時間が 30 を超えた場合、或いは待機しているトランザクションが 5 つになった場合、sync 操作が実行されます。**DB_GcMxw**＝ 0 にすると、グループコミットが完全に使用不能です。

初期値: 0(使用不可)

有効値の範囲: >0

関連キーワード: DB_GcChk、DB_GcWtm

使用する場所: サーバー側

DB_GcWTM=<値>

このキーワードは、グループ・コミットを待機するトランザクションの最大グループコミット待機時間を指定します。待機時間を長くすると、トランザクションの応答時間を縮小しますが、グループ・コミット全体にかかる時間は増加します。

このキーワードは、**DB_GcMxw** と共に使用します。詳細については、**DB_GcMxw** を参照して下さい。

初期値: 10 (ミリ秒)

有効値の範囲: 待機時間 > 0

関連キーワード: DB_GcChk、DB_GcMxw

使用する場所: サーバー側

DB_IDCAP=<値>

このキーワードは、データベースにある全識別子が大文字小文字を識別するかどうかを指定します。値を 0 にセットすると、全ての識別子は、大文字小文字を識別します。値を 1 にセットすると、全ての識別子は大文字小文字を識別しないことを意味します。このモードの場合、全ての識別子は特定の場所では大文字に変換されます。このキーワードは、データベース作成時にのみセットすることができます。つまり、既存データベースのこのキーワードを変更しても無効です。

初期値: 1

有効値の範囲: 0 (大文字小文字を識別する)

1 (大文字小文字を識別しない)

使用する場所: サーバー側

DB_IDXDP=<値>

このキーワードは、自動インデックスデーモンによって自動的にインデックスを削除する閾値を指定します。自動インデックスが **DB_IdxDp** に指定された削除日数を越えてまだ使用されていない場合、自動インデックスデーモンでそれを自動的に削除することになります。データベースを実行する際に、ユーザーは setSystemOption()をコールすることによって、**IDXDP** の値を設定することができます。

初期値: 30（日）

有効値の範囲: 0 - 365（日）

関連キーワード: DB_IdxLg、DB_IdxLn、DB_IdxSv、DB_IdxTm、DB_IdxTv

使用する場所: サーバー側

DB_IDXLG =<文字列>

このキーワードは、自動インデックスのログファイル保存用のディレクトリを指定します。そのディフォルトパスはデータベースのインストールディレクトリです。

初期値: データベースのディレクトリ

有効値の範囲: 文字列の長さ < 256

関連キーワード: DB_IdxDp、DB_IdxLn、DB_IdxSv、DB_IdxTm、DB_IdxTv

使用する場所: サーバー側

DB_IDXLN=<値>

このキーワードは、自動インデックスデーモンによって自動的にインデックスを作成する閾値を指定します。スキャンログの数（これらのログは同じテーブル id、テーブルバージョン及びカラム ID リストを持っている）が **DB_IdxLn** に指定された値を超えると、自動インデックスデーモンがシリアルログに応じて自動インデックスを作成します。データベースを実行

する際に、ユーザーは setSystemOption()をコールすることによって、**IDXLN** の値を設定することができます。

初期値: 1

有効値の範囲: 1 ~ 65535

関連キーワード: DB_IdxDp、DB_IdxLg、DB_IdxSv、DB_IdxTm、DB_IdxTv

使用する場所: サーバー側

DB_IDXSV=<値>

このキーワードは、自動インデックスサーバーをアクティブ化するために使用されます。その値を 1 に設定する場合は、サーバーが起動していると表し、0 に設定する場合は、サーバーが起動していないと表します。データベースを実行する際に、ユーザーは setSystemOption()をコールすることによって、**IDXTM** の値を設定することができます。

初期値: 0

有効値の範囲: 0、1

関連キーワード: DB_IdxDp、DB_IdxLg、DB_IdxLn、DB_IdxTm、DB_IdxTv

使用する場所: サーバー側

DB_IDXTM=<文字列>

このキーワードは、自動インデックスデーモンが自動インデックスを実行する最初の時間を指定します。**DB_IdxTm** のフォーマットは **yyyy-mm-dd hh:mm:ss** です。データベースを実行する際に、ユーザーは setSystemOption()をコールすることによって、**IDXSV** の値を設定することができます。

初期値: 1970-01-01 00:00:00

有効値の範囲: 1970-01-01 00:00:00 ~ 2037-12-31 23:59:59

関連キーワード: DB_IdxDp、DB_IdxLg、DB_IdxLn、DB_IdxSv、DB_IdxTv

使用する場所: サーバー側

DB_IDXTV=<文字列>

このキーワードは、自動インデックスデーモンの時間間隔を指定します。"1-12:30:00"のような値は、時間間隔が一日＋12 時間＋30 分間です。データベースを実行する際に、ユーザーは setSystemOption()をコールすることによって、IDXTV の値を設定することができます。

初期値: 0-01:00:00

有効値の範囲: 00:00:00 ~ 24854-23:59:59

関連キーワード: DB_IdxDp、DB_IdxLg、DB_IdxLn、DB_IdxSv、DB_IdxTm

使用する場所: サーバー側

DB_IFMEM =<値>

このキーワードは、IVF テキストインデックス検索ルーチン使用にの DBMaster が分配するメモリ使用量近似の上界の最大量を MB でを指定します。インバーテッドファイルのテキスト索引を作成している間、それは大量のメモリーリソースを必要とします。DBMaster は、テキスト索引の作成時の最大メモリ使用量を決定するため単純な規則に従います。DBMaster が自由なメモリを検知することができないか、自由なメモリーリソースが 128MB 未満の場合、最大のメモリ使用量は 64MB になります。そうでなければ、自由なメモリーリソースの半分が最大のメモリ使用量になります。ユーザは dmconfig.ini に、キーワードエントリーDB_IFMem を加えることにより、メガバイト(MB)によってメモリ使用量の近似の上界を手動で指定することができます。DBMaster は、通常オペレーティングシステムが利用可能と判断したメモリの半分を動的に割り当て管理します。DBMaster が利用可能なメモリの量を検知することができない場合や、128MB 以上の利用が可能であると検知した場合､分配する最大メモリを 64MB にセットします。このキーワードを使用して任意の値を指定すると、DBMaster はその値を越えるメモリ量を割り当てることはありません。オペレーティングシステムがメモリ使用情報を提供しない場合、もしくは改善された IVF テキストインデックスクエリーの性能を最大限に使う為より大きなのキャッシュサイズが必要であれば、このキーワードの使用が推奨されます。

➲ 例

IVF テキストインデックスバッファー用に分配するメモリ最大量を 256MB に指定します。

```
DB_IFMem = 256
```

初期値: 無し。DBMaster は自動的にバッファーを配置します。

有効値の範囲: 64 ~ (オペレーティングシステムによって許可された最大値)

使用する場所: サーバー側

DB_IOSVR=<値>

このキーワードは、I/O サーバとチェックポイント・デーモンを起動させるかどうかを指定します。値を 1 にすると、データベース・サーバー起動後、I/O とチェックポイント・デーモンを起動させます。キーワードは、データベース起動時に使用されます。

➲ 例 1

I/O とチェックポイント・デーモンを起動する：

```
DB_IOSvr = 1
```

初期値: 1

有効値の範囲: 0、1

関連キーワード: DB_Nbufs

使用する場所: サーバー側

DB_ISOLV=<値>

このキーワードはユーザーがデータベースに接続するときのデフォルトのトランザクションの分離性を指定します。

有効値とオプションは以下の通りです。

- 1: 未コミット読み取り
- 2: コミット読み取り

- 3: 繰り返し可能読み取り
- 4: 直列化可能

初期値: 1

有効値の範囲: 1-4

使用する場所: クライアント側

DB_ITCMD=<値>

このキーワードは、暗黙的なデータ変換をオンまたはオフにするかどうかを指定します。この値を 1 に設定すると、暗黙的なデータ変換機能が有効になり、0 に設定すると、当該機能が無効になっています。

初期値: 0

有効値の範囲: 0，1

使用する場所：サーバー側

DB_ITIMO=<値>

このキーワードは、アイドルタイムアウトの時間間隔(秒)を指定します。DBMaster は、指定したタイムアウト間隔以上データベース操作が無い接続を自動的に切断します。この機能は、アイドル接続に強制的にバッファ、ページ、ロック、メモリ等のデータベースの全リソースを解放させます。値を 0 にセットすると、この機能を OFF にします。この値が **DB_DtCtl** より小さい場合は、自動的にこの値は DB_DtCtl にリセットされます。このキーワードの初期値は 0 ですが、0 にするとこの機能が閉じられます。

DBMaster クライアントとサーバー側の間に最大なアイドルタイムは **DB_DtClt**（クライアントで設置）と **DB_ITimo**（サーバー側で設置）の設定値によって決めますが、ルールは以下のようになります：

- ユーザーはこの二つのキーワードに一つのキーワードを設置すると、DBMasterクライアントとサーバー側のコミュニケーションのアイドルタイムは設置されたキーワードの値になります。つまり、ユーザーはクライアントで**DB_DtClt**を設定しますが、サーバーで**DB_Itimo**を設定しないと、DBMasterクライアントとサーバー側のコミュニケ

ーションの最大のアイドルタイムはキーワードDB_DtCltの設置値になります。ユーザーはクライアントでDB_DtCltを設定しなく、サーバー側でDB_ITimoを設置すれば、DBMasterクライアントとサーバー側のコミュニケーションの最大のアイドルタイムはキーワードDB_Itimoの値になります。

- ユーザーは同時にこの二つのキーワードを設定すると、以下の状況が起こします：

  a) DB_DtCltの値はDB_Itimoの値より小さいと、DBMasterのクライアントとサーバーの側のコミュニケーションの最大のアイドルタイムはキーワードDB_DtCltの値になります。

  b) DB_DtCltの値はDB_Itimoの値より大きいと、DBMasterのクライアントとサーバーの側のコミュニケーションの最大のアイドルタイムはキーワードDB_ITimoの値になります。同時にDB_DtCltの設置値はDB_Itimoの値をリセットされます。

- ユーザーはいずれクライアントでDB_DtCltを設定しなく、サーバーでもDB_Itimoを設定しないと、DBMasterクライアントとサーバー側のコミュニケーションの最大のアイドルタイムは無制限になります。

初期値: 0 (使用しない)

有効値の範囲: 0 ~ 4294967 (秒)

関連キーワード: DB_DtClt

使用する場所: サーバー側

DB_JNFIL=<文字列>

このキーワードは、ジャーナルファイル名を指定します。指定したファイルの個数は、データベースに割り当てるジャーナルファイルの個数を表します。最大 8 個のジャーナルファイル名を指定することができます。

初期値: データベース名.JNL。例、DB.JNL.

有効値の範囲： 文字列の長さ < 256

関連キーワード： DB_JnlSz、DB_DbDir

使用する場所: サーバー側

DB_JNLSZ=<値>

このキーワードは、ジャーナルファイルのサイズを指定します。ユーザーはユニットが M（メガバイト） G（ギガバイト）にて設定することができます。M 、G が使われないとユニットはページです。ユーザーは値が M または G にすると、実際サイズは指定サイズよりも 1 ページ分小さくなります。例： ページのサイズは 16 KB でユーザーは **DB_JnlSz** を 8M に設定すると、ジャーナルファイルのサイズは、8192 KB ではなく、8176 KB となります。

初期値: 1000（ジャーナルファイルの初期値サイズは 1000×DB_PGSIZ KB になります)

有効値の範囲： 100 ページ~ 8G

関連キーワード： DB_JnlSz、DB_DbDir

使用する場所: サーバー側

DB_LBDIR=<文字列>

このキーワードは、ユーザー定義関数(UDFs)、又は Windows のダイナミックリンクライブラリ *(DLLs)*、又は Unix ファイルのがデータベースの起動時にロードされるディレクトリを指定します。

初期値: DBMaster の作業ディレクトリ

有効値の範囲: 文字列の長さ < 256

関連キーワード: DB_JnlSz、DB_DbDir

使用する場所: サーバー側

DB_LCDEC=<値>

このキーワードは、データベースがシステムに小数点設置キャラクタを使用するかをチェックするため使用されます。キーワードの値は 1 になると、チェック機能を開きます。0 になるとチェック機能が閉じます。

初期値は 0 で、"."を DBMaster のディフォルト称す点設置キャラクターとします。このキーワードを 1 になる場合、DBMaster はシステムの小数点設置キャラクターをチェックして、チェックしたキャラクターを DBMaster に小数点キャラクターとします。つまり、もしシステムに使用する小数点キャラクターは","なら、DBMaster も","を使用します；もしシステムに使用する小数点キャラクターは"."なら、DBMaster も"."を使用します。システムに小数点キャラクタが","を使用すると、ユーザーはこのキーワードを 1 を設定することを推薦します。

初期値: 0

有効値の範囲: 0，1

関連キーワード: DB_LCode

使用する場所：クライアント側

DB_LCODE=<値>

このキーワードは、サーバー側の言語コードを指定します。言語コードは、問合せの中の LIKE 演算の結果に影響します。0 は ASCII 互換言語、値 1 は BIG5 コード互換言語、値 2 はシフト JIS 互換言語、値 3 は GB コード互換言語を表します。詳細については、*「SQL 文と関数参照編」*をご覧ください。このキーワードは、データベースの起動時に使用されます。

SELECT GETSYSINFO('LCODE')コマンドにてサーバー側の言語コード設定が返されます。

初期値: 0

有効値の範囲： 0－ 英語 (ASCII)

1 － 繁体字中国語 (BIG5)

2 － 日本語 (シフト JIS)

3 － 簡体字中国語 (GB)

4－ ラテン文字 1 (ISO-8859-1)

5 － ラテン文字 2 (ISO-8859-2)

6－ キリル文字 (ISO-8859-5)

7 － ギリシア文字 (ISO-8859-7)

8 － 日本語文字 (EUC-JP)

9－ 簡体中国語文字 (GB18030)

10－ UTF-8（UTF-8）

関連キーワード: DB_CliLCODE、DB_ErrLCODE

使用する場所: サーバー側

DB_LETPT=<値>

このキーワードは、ページ・ロックから表ロックに切り替える時の*ロック拡張しきい値*を指定します。ページ・ロック数がしきい値以上になると、自動的に表ロックに拡張します。

初期値： 60

有効値の範囲： 3 - 32767

関連キーワード： DB_LetRP

使用する場所: サーバー側

DB_LETRP=<値>

このキーワードは、レコード（行）ロックからページ・ロックに切り替える時のロック拡張しきい値を指定します。同じページに行ロック数がしきい値以上になると、DBMaster は自動的にページ・ロックに拡張します。

初期値: 30

有効値の範囲： 3 - 32767

関連キーワード： DB_LetPT

使用する場所: サーバー側

DB_LGDAY=<値>

このキーワードはログファイルの有効日数を指定します。**DB_STSVR** で指定した日数に達した期限切れのログファイルはデーモンによって削除されます。**DB_LGDAY** = 0 のとき、ログシステムには元来のルール(ファイル名に日付なし)が適用されます。**DB_LGDAY** が 0 より大きい場合、**DB_LGFNO** の設定は無効になります。

初期値：30

有効値の範囲：1~365

関連キーワード：DB_LgZip、DB_ DB_LgSvr、DB_ LgFNO

使用する場所: サーバー側

DB_LGDIR=<文字列>

このキーワードはサーバのログファイルを保存するディレクトリを指定します。

初期値: *DB_DbDir/lgdir* でディレクトリを指定

有効値の範囲:文字列の長さ < 256

関連キーワード: DB_DbDir、DB_LgSvr

使用する場所: サーバー側

DB_LGERR=<値>

DB_LgSvr の値 が 1-4 の場合、このキーワードはエラーログレベルを指定します. 値を 0 にすると、エラーコード>30000 のコアダンプとデータベース クラッシュエラーのログを取ります；値を 1 にすると、エラーコード >20000 の切断エラーとデータベース クラッシュエラーのログを取ります；値を 2 にすると、エラーコード>10000 のアボート、切断エラーとデータベース クラッシュエラーのログを取ります；値を 3 にすると、エラーコード>100 の通常エラー、アボート、切断、データベース 損傷エラーのログを取ります；値を 4 にすると、エラーコード>0 の警告とすべてのエラーのログを取ります。

初期値: 3

有効値の範囲: 0、1、2、3、4

関連キーワード: DB_LgSvr

使用する場所: サーバー側

DB_LGFNO=<値>

DBMaster は DBNAME_1.LOG としてログを保存します。ログファイル・サイズが初期値 100 MB 或いは DB_LgFSz で指定された値に達すると、ログが以下のように記録されます。例：DBNAME_2.LOG、DBNAME4_3.LOG …DBNAME_n.LOG。n は生成するログファイル数です。このキーワードは n の値を指定します。

初期値:20

有効値の範囲: 2 ~ 255

関連キーワード: DB_LgSvr、DB_LgFSz、DB_LgDay

使用する場所: サーバー側

DB_LGFSZ=<値>

このキーワードはログファイルとテキストファイルのサイズを指定しますが、その単位はメガバイトです。DBMaster はログシステムに記録された情報をログファイル DBNAME_1.LOG に保存します。ログファイルが指定するサイズに達すると、ログ情報が次のログファイルに記録されます。例えば、 DBNAME4_2.LOG、DBNAME4_3.LOG…DBNAME4_n.LOG、ここの n は DB_LgFNo によって指定された値ですが、DB_LgFNo に指定されていない場合、デフォルトとして 20 となります。ファイルの数はデフォルトの 20 個に達した場合または DB_LgFNo に指定された数に達した場合まで、後ログ情報が DBNAME_1.LOG に再記録されます。

初期値:100

有効値の範囲: 10 ~ 1500

関連キーワード: DB_LgSvr、DB_LgFNo

使用する場所: サーバー側

DB_LGLCK=<値>

このキーワードは特別なロック情報のログ取得を指定します。ロック・タイム・アウト或いはデッドロックの時にログが出力されます。値を 0 にすると、ログを取得しません、値を 1 にすると、ログを取得します。

初期値: 0

有効値の範囲: 0、1

関連キーワード: DB_LgSvr

使用する場所: サーバー側

DB_LGPAR=<値>

このキーワードは入力引数値のログ取得状態を指定します。値を 0 にすると、ログ機能が無効になり、値を 1 にすると、入力引数値のログを取ることができます；値を 2 にすると、入力引数値とストアドプロシージャにて実行された SQL コマンドを同時にログを取ります。値を 3 にすると、入力引数値とトリガーSQL コマンドを同時にログを取ります。値を 4 にすると、ストアドプロシージャとトリガーSQL コマンドと入力引数値を同時にログを取ります。

初期値: 0

有効値の範囲: 0、1、2、3、4

関連キーワード: DB_LgSvr

使用する場所: サーバー側

DB_LGPLN=<値>

このキーワードは select、update 、delete 文のログ状態を指定します。値を 0 にすると、この効能が閉じられますが、値を 1 にすると、実行プランのログを取得します。

初期値: 0

有効値の範囲: 0、1

関連キーワード: DB_LgSvr

使用する場所: サーバー側

DB_LGSQL=<値>

このキーワードは **DB_LgSvr** = 4 設定時の SQL コマンドを記録することに使用されます。値を 0 にすると、SQL コマンドを記録しません；値を 1 にすると、**DB_LgSvr** = 4 の場合、 select 文以外の SQL コマンドを記録します。値を 2 にすると、**DB_LgSvr**=4 の場合、全ての SQL コマンドを記録します。

初期値: 2

有効値の範囲: 0、1、2

関連キーワード: DB_LgSvr

使用する場所: サーバー側

DB_LGSTM=<値>

このキーワードはログ取得間隔時間を指定します。時間の掛かる処理を記録する、**DB_LgSvr** の設定が 2、3 、 4 の時のみ必要です。

初期値: 5 (秒)

有効値の範囲: 1 ~ 65536 (秒)

関連キーワード: DB_LgSvr

使用する場所: サーバー側

DB_LGSVR=<値>

このキーワードではサーバのログの取得レベルを定義します。値が 0 の場合はログサーバーが閉じます。値が 1 ではログサーバーがエラー情報をレコードします、デフォルトのエラーレベルは **DB_LgErr** で定義された。値が 2 の場合はログサーバーは処理の遅いコマンドをレコードして、デフォルトの時間は **DB_LgSTm** で設定されます。値が 3 ではエラーと処理の遅いコマンドの両方のログを取得できます。値 4 では切断、COMMIT、ROLLBACK、SQL コマンド、処理の遅いコマンドのログを取得。SQL ログの取得状況は **DB_LgSQL** によって定義されます。値 5 ではデータベースが切断する場合、全部のログを取得できます。値 6 では接続、切断する場合の全部のログを取得することが可能です。

ログシステムに関する詳細はセクション 4.2 の *ログシステムの起動*をご参照ください。

初期値: 0

有効値の範囲: 0、1、2、3、4、5、6

関連キーワード: DB_LgErr、DB_LgSTm、DB_LgSQL、DB_LgFSz、DB_LgFNo、DB_LgDir、DB_LgPLn、DB_LgPar、DB_LgLck、DB_LgSys

使用する場所: サーバー側

DB_LGSYS=<値>

ログシステムを起動すると、このキーワードはどんなインフォメーションのログを取るかを指定します。値は 0 にすると、実行タイム、エラーコード、サービスファクション、接続 ID、ユーザー名、処理 ID，ログイン情報、エラー引数、SQL 文などのログを取ります ; 値 1 は **DB_LgSys=0** に設定した場合と同様で、SYSUSER と SYSINFO のログも取得します。値は 2 では **DB_LgSys**=1 とほぼ同じで加えてメモリの検知が可能なときにはすべての SYSTEM メモリのログを取得することができます。

初期値: 0

有効値の範囲: 0、1 、2

関連キーワード: DB_LgSvr、DB_LgSQL、DB_LgFSz、DB_LgErr、DB_LgSTm

使用する場所: サーバー側

DB_LGZIP=<値>

このオプションはログファイルの圧縮に使用され、ストレージを節約します。

初期値: 0

有効値の範囲: 0、1

使用する場所: サーバ側

関連キーワード：DB_LgSvr、DB_LgDay、DB_LgFNo

DB_LTIMO=<値>

このキーワードは、ロック・タイムアウト時間を秒数で指定します。例：ほかのトランザクションに配分した既に他のトランザクションによって割り当てられた表やタプルといったデータベース・オブジェクトをロックする場合、オブジェクトが開放されるまで待機させられます。

値を指定すると DBMaster は指定した時間の間オブジェクトがロックできるようになるのを待ちますが、待ち時間を超過したときは、「ロック・タイムアウト」エラーメッセージが返されます。或いは、ロックタイムアウトが終了する前にロックが取得されるまでです。タイムアウトを無効にするには、このキーワードを-1 に設定します。この場合は、ロックができるまで待ち続けます。

全く待ちたくないときは、0 に設定します。このキーワードは、データベースの起動時ではなく、接続時に使用されます。クライアント/サーバー・モードの場合、各クライアントは自分自身の dmconfig.ini をもっているので、クライアント毎にロック・タイムアウトを設定することができます。

初期値： 5（秒）

有効値の範囲： -1~ 65535

使用する場所: クライアント側

DB_MAXCO=<値>

このキーワードは、新規ジャーナル・モードでデータベースを作成、或いは起動する場合に、ハード接続数を指定します。一方、データベースをノーマル起動モードで起動する場合は、ソフト接続数を指定します。ソフト接続数は、データベース・システムで同時に処理することができる最大トランザクション数を意味します。また、1 接続で 1 トランザクションしか扱うことができないので、データベース・システムに設けることができる最大同時接続数という意味もあります。

ハード接続数は、ジャーナルファイルに記録可能な最大接続数を定義します。ハード接続数は、240 から 4800 間の 40 の倍数です。*ハード接続数*が DBMaster のライセンスに関連することがあります。データベースの作成時、または新しいジャーナル操作によるデータベースの起動時、値を **dmconfig.ini** の **DB_MaxCo** に割り当てないと、*ハード接続数*は最大 (**DB_MaxCo** のデフォルト値、ライセンスの最大ユーザー数) になります。一方、データベースの作成時、または新しいジャーナル操作によるデータベースの起動時、値を **DB_MaxCo** に割り当てると、*ハード接続数*は最大 (**DB_MaxCo** のデフォルト値、割り当てられた値) になります。つまり、ハード接続数を判断するために、**DB_MaxCo** は 40(又は 240)に最も近い倍数値に丸められます。ハード接続数とソフト接続数とデータベースのパフォーマンスに与える影響についての詳細は、18.5 節の*同時実行処理のチューニング*をご参照下さい。

初期値: 240

有効値の範囲: 2~4800

関連キーワード:DB_SMode

使用する場所: サーバー側

DB_MTIMO=<値>

このキーワードはクライアントのラッチタイムアウト値を秒単位で指定します。クライアントのラッチは各 API 関数の開始時期に設定され、API 関数

の終了時に解除されます。データベースへの接続または操作を同一セッションで行うマルチスレッドアプリケーションの問題を避けるには、この機能を使用します。

キーワードを 0 に設定すると、クライアントラッチの解除を待たないように指定されます。0 以上の数にキーワードを設定すると、複数のスレッドが同一コネクションを使って API 関数に同時に接続してコールする時に問題が発生するのを最小限に抑えるように DBMaster がクライアントラッチの解除を待つように指定されます。

初期値: 0

有効値の範囲: 0 ~ 65535

使用する場所: クライアント側

DB_MXCMD=<値>

このキーワードは ODBC アプリケーションが扱える最大ステートメント数または DCI COBOL アプリケーションがオープンする最大のテーブル数(値-1)を定義します。値に"n"を置くと ODBC アプリケーションは一接続に対して最大 n ステートメントまで割り当てることができます。DCI では最大(n-1)テーブル+1 ステートメントを SQL 実行のためにオープンできます。

初期値: 257

有効値の範囲:1 ~ 32767

使用する場所:サーバー側

DB_NBUFS=<値>

このキーワードは、データバッファのサイズを指定します。ユーザーはユニットが M（メガバイト） G（ギガバイト）にて設定することができます。M 、G が使われないとユニットはページです。ユーザーは値を M または G で指定すると、実際サイズが指定されたサイズより 1 ページ分小さくなります。例： ページのサイズは 16K で DB_NBufs を 8M に設定すると、ジャーナルファイルのサイズは 8192K ではなく、8176K となります。一般に、

バッファが多いほど、DBMaster は効率よく走行します。このキーワードは、データベースの起動時に使用されます。

DBMaster が物理メモリ使用を検出できるプラットフォームで値を 0 にすると、自動的にバッファのサイズが設定されます。DBMaster が物理メモリ情報を取得できない場合、2000 ページにセットされます。

➲ **例**

データベース起動後、SYSINFO 表に問合せると、データベースが使用しているバッファの数が判断できます。次のコマンドは、データベースが現在 500 ページのバッファを使用していることを表しています。:

```
dmSQL> select * from SYSINFO where INFO = 'NUM PAGE BUF';

 ID       INFO            VALUE
==== ============== ==================
0107  NUM PAGE BUF         500

1 rows selected
```

最適値の判断については、18 章の*「パフォーマンスのチューニング」*の*「メモリ割り当てのチューニング」*を参照して下さい。

初期値： 0（自動設定）

有効値の範囲： 0、15（システムに依存）

関連キーワード： DB_NJnlB、DB_ScaSz

使用する場所: サーバー側

DB_NETEC=<値>

このキーワードは、ネットワークの暗号化の on/off を指定します。ネットワーク暗号化が ON の場合、DBMaster のサーバーとクライアント間のネットワークの全データは、暗号化されます。DBMaster の暗号化技術は、DES と RSA の混合型です。

初期値: 0 (暗号化しない)

有効値の範囲: 0 (暗号化しない)、1 (暗号化する)

使用する場所: サーバー側

DB_NETZC=<値>

サーバとクライアントの間でデータを送る時、このキーワードは圧縮ファクションを開くか on/off で指定しますこの機能を有効にすると、サーバからのデータを圧縮し転信され、クライアントがデータを受け取って展開するというものです。データ送信量を減らし、転送速度を短縮します。**DB_NetZc**=1 にすると機能が有効になり、**DB_NetZc**=0 の場合は無効になります。

初期値: 0 (off)

有効値の範囲: 0 (off) 、1 (on)

使用する場所:クライアント側

DB_NJNLB=<値>

このキーワードは、共有メモリのジャーナルバッファの個数を指定します。ジャーナルファイルのサイズと間違えないでください。共有メモリに取られるジャーナルバッファの個数です。

ユーザーはユニットが M（メガバイト） G（ギガバイト）にて設定することができます。M 、 G が使われないとユニットはページです。ユーザーは値を M または G で指定すると、実際サイズが指定されたサイズより 1 ページ分小さくなります。例： ページのサイズは 16K で **DB_ NJnlB** を 8M に設定すると、ジャーナルファイルのサイズは 8192K ではなく、8176K となります。このキーワードは、データベースの起動時に使用されます。

初期値：64 (64* DB_PgSiz KB)

有効値の範囲： 16（システムに依存）

関連キーワード： DB_JnlSz、DB_NBufs、DB_ScaSz、DB_PgSiz

使用する場所: サーバー側

DB_OPTRT =<値>

このキーワードはクエリオプティマイザが内部及び外部結合においてネスト結合またはマージ結合を選択する根拠を指定します。そのデフォルト値は 0 です。このキーワードに 1 を設定する場合、オプティマイザは実行時間の代わりに応答時間を考慮するのを表し、0 を設定する場合、オプティマイザは応答時間の代わりに実行時間を考慮するのを表します。

初期値：0

有効値の範囲：0、 1

使用する場所：サーバー側

DB_ORDER=<文字列>

このキーワードは、DBMaster のインストール・ディレクトリのサブディレクトリ **shared/codeorder** に格納されたオーダー定義ファイル名を指定します。オーダー定義ファイルは、DBMaster のソート結果に影響する、純粋なテキスト・ファイルです。このキーワードは、データベース作成時に使用され、それ以降無用になります。このキーワードを定義しない場合、デフォルトソート・シーケンスは、バイナリ・シーケンスになります。

初期値: なし

有効値の範囲:ユーザー定義オーダー定義ファイルのファイル名

関連キーワード: DB_LCode

使用する場所: サーバー側(データベース作成時のみ)

DB_PASWD=<文字列>

このキーワードは、初期設定のログイン・ユーザーID のパスワードを指定します。初期設定のログイン・ユーザーID が指定されていないときは、この値は無視されます。このキーワードは、データベースの起動時と接続時に使用されます。

初期値： Null 文字列

有効値の範囲： 文字列 <=16

関連キーワード： DB_UsrId

使用する場所: クライアント側

DB_PGSIZ = <値>

このキーワードがページサイズを指定します。DBMaster ではユーザーはデータのページサイズを選択できます、サイズは 4 K、8 K、16 K 及び 32 K です。このキーワードはデータベースを作成時のみ有効です。

初期値: 8

有効値の範囲: 4、8 、16 、32

使用する場所:サーバー側（データベースを作成する場合のみ）

DB_PTNUM=<値>

このキーワードは、データベース・サーバーの TCP/IP ポート番号を指定します。このキーワードは、クライアント側では接続時に使用され、サーバー側では起動時に使用されます。ポート番号が、全てのクライアントとサーバー・マシンで厳密に一致していなければ、接続は不可能になります。

初期値： 無し

有効値の範囲： 1024 ~ 65535

関連キーワード： DB_SvAdr

使用する場所: サーバーとクライアント側両方

DB_RESWD=<値>

DBMaster の予約語として指定されている単語をオブジェクト名として使用することができます（予約語の一覧については*「SQL 文と関数参照編」*を参考のこと）。**DB_ResWd** を 1 にセットした場合、予約語を用いたオブジェクトを追加しようとするとエラーになります。**DB_ResWd** を 0 にセットすると、エラーになりません。このキーワードは、予約語を含んだオブジェクトのインポートを可能にすることです。

初期値: 1

有効値の範囲: 0、1

使用する場所: サーバー側

DB_RMPAD=<値>

このキーワードは、CHAR データのスペース埋め込みを削除するかどうかを指定します。値を 0 にすると、結果セットの CHAR データのスペース埋め込みを全て残すことを意味します。値を 1 にすると、ユーザー・バッファにコピーする前に、CHAR データのスペース埋め込みを削除することを意味します。ユーザー・アプリケーションは、データ挿入時に DBMS で生成された埋め込みスペースがない固定長の CHAR データを回収することができるようになります。

初期値: 0

有効値の範囲: 0、1

使用する場所: クライアント側

DB_RSTSN=<値>

このキーワードは最大な serial 値になる時自動に serial 数は原始値を設定する。

初期値: 0

有効値の範囲: 0、1

使用する場所: サーバー側

DB_RTIME=<文字列>

このキーワードは、データベースをリストアするときの目標時点を指定します。データベースをリストアするときは、バックアップ・ファイルの最も早い時点から DB_RTime で指定した時点まで、バックアップ・ファイルをロールフォワードします。DB_RTime を指定しない場合、バックアップ・ファイルの最後（バックアップを取った時点）までリストアします。

バックアップ時点よりも後に **DB_RTime** 時点を指定すると、バックアップ時点が **DB_RTime** として使用されます。

初期値： 0 （ 70/1/1 00:00:00 ）

有効値の範囲： YY/MM/DD hh:mm:ss

使用する場所: サーバー側

DB_SCASZ=<値>

このキーワードは、システム制御エリア（SCA）のサイズを指定します。ユーザーはユニットが M（メガバイト） G（ギガバイト）にて設定することができます。M 、 G が使われないとユニットはページです。ユーザーは値を M または G で指定すると、実際サイズが指定されたサイズより 1 ページ分小さくなります。例 : ページのサイズは 16K で **DB_JnlSz** を 8M に設定すると、ジャーナルファイルのサイズは 8192K ではなく、8176K となります。SCA の必要最小メモリが **DB_ScaSz** の値よりも大きいときは、必要最小メモリを SCA に割り当てられ、この値は無視されます。このキーワードは、データベースの起動時に使用されます。

初期値：200（200 ×DB_PGSIZ KB)

有効値の範囲： 1~（システムに依存）

関連キーワード： DB_NbufS、DB_NJnlB

使用する場所: サーバー側

DB_SCHSV=<値>

このキーワードは、dmschsvr サービスを有効にするかどうかを指定します。このキーワードを 1 に設定する場合、dmschsvr サービスが有効になり、0 に設定する場合、dmschsvr サービスが無効になります。

初期値：0

有効値の範囲：0、1

関連キーワード： DB_TskNo

使用する場所：サーバー側

DB_SMODE=<値>

このキーワードは、データベースの起動モードを指定します。6 つの起動モードがあります。

- **ノーマル起動**—システムをノーマルに起動します。データベースがクラッシュしたときは、自動的にクラッシュをリカバリし整合性の取れた安定したデータベースにします。
- **新規ジャーナル起動**—システムをノーマルに起動しますが、ジャーナルファイル（DB_JNFILキーワードで指定）を新規に作成します。同じ名前のファイルがあるときは上書きします。
- **ロールオーバー起動**—バックアップファイル（ジャーナルファイルを含めて）を使用してデータベースを起動します。このモードはデータベースのリストアに使用され、**DB_RTime**で指定された時点までデータベース・オペレーションをロールオーバーします。ロールオーバーについての詳細は、*「データベースのリカバリ、バックアップ、リストア」*の章を参照して下さい。
- **ソース・データベースとして起動**—このモードは、データベースレプリケーションで使用します。このモードでシステムを起動すると、ソース・データベースになります。詳細については、*「データベース・レプリケーション」*の章を参照して下さい。
- **ターゲット・データベースとして起動**—このモードは、データベース・レプリケーションで使用します。このモードでシステムを起動すると、ターゲット・データベースになります。詳細については、*「データベース・レプリケーション」*の章を参照して下さい。
- **読み込み専用データベースを起動**— システムをノーマルに起動します。但しデータベースは読み込み専用で、読み込み権限しか与えられません。読み込み専用モードで書き込み許可を与えてデータベースを起動しても、ユーザーは修正することができません。

初期値： 1

有効値の範囲： 1 （ノーマル起動）

2 （新規ジャーナル起動）

3 （ロールオーバー起動）

4 （ソース・データベース起動）

5 （ターゲット・データベース起動）

6 （読み込み専用データベース起動）

関連キーワード： DB_Forcs

使用する場所: サーバー側

DB_SPDIR=<文字列>

このキーワードは、ストアド・プロシージャのファイルのパスを指定します。ストアド・プロシージャ・ファイルには、生成したダイナミック・リンク・ライブラリ・ファイルとストアド・プロシージャ作成時に生成された一時ファイル全てが含まれます。絶対パス名を指定します。

➲ **DB_SPDir の例**

```
DB_SPDir = /usr/DBMaster/data/spdir          ;UNIX の場合
```

➲ **DB_SPDir の例**

```
DB_SPDir = c:\DBMaster\data\spdir            ;Windows の場合
```

初期値: <データベースディレクトリ>

有効値の範囲: 文字列の長さ < 256

関連キーワード: DB_SPInc

使用する場所: サーバー側

DB_SPINC=<文字列>

このキーワードは、ストアド・プロシージャのインクルード・ファイルのパスを指定します。生成したストアド・プロシージャに、追加のインクルード・ファイルが必要な場合に利用されます。絶対パス名を指定します。

➲ DB_SPInc の例

```
DB_SPInc = /usr/DBMaster/data/sp\include          ;UNIX の場合
```

➲ DB_SPInc の例

```
DB_SPInc = c:\DBMaster\data\sp\include            ;Windows の場合
```

初期値: (dmserver が走行している現在のディレクトリ)

有効値の範囲: 文字列の長さ < 256

関連キーワード: DB_SPDir

使用する場所: サーバー側

DB_SPLOG=<文字列>

このキーワードは、ストアド・プロシージャのログファイルのパスを指定します。ストアド・プロシージャのログファイルには、ストアド・プロシージャ作成時にデータベースから送信されたエラー・ログファイルと、ストアド・プロシージャ実行のためのトレース・ログファイルが含まれます。絶対パス名を指定します。

➲ DB_SPLog の例

```
DB_SPLog = /usr/joe/mydata/splog                  ;UNIX の場合
```

➲ DB_SPLog の例

```
DB_SPLog = c:\user\joe\mydata\splog               ;Windows の場合
```

初期値: (クライアント・アプリケーションが走行している現在のディレクトリ)

有効値の範囲: 文字列の長さ < 256

使用する場所: クライアント側

DB_SQLST=<値>

このキーワードは、SQL コマンドモニターの表示モードを指定します。SYSUSER システム表の SQL_CMD と TIME_OF_SQL_CMD の表示内容に影響します。SQL コマンドモニターは、SQL コマンドを実行している際に

精確またはおおまかな SQL 情報を取得することができます。精確な情報の取得は、おおまかな情報の取得と比べて、CPU の消費時間が多いです。CPU の過負荷を回避するために、SQL コマンドモニターを閉じることができます。

初期値: 1

有効値の範囲: 0 (SQL コマンドモニターを切る)

1 (SQL コマンドとおおまかな SQL コマンドの実行時間を表示)

2 (SQL コマンドと精確な SQL コマンドの実行時間を表示)

使用する場所: サーバー側

DB_STACL=<値>

このキーワードは、ユーザーがデータベースに接続するときにデータベースサーバがユーザーの IP アドレスを確認するかどうかを指定します。

キーワードを 1 に設定すると、ACL の確認が有効になります。キーワードを 0 に設定すると、ACL の確認が無効になります。

初期値: 0

有効値の範囲: 0、1

使用する場所: サーバー側

DB_STMOD =<値>

このキーワードは、更新統計デッモンモードを指定します。キーワードを 0 に設定すると、更新統計デッモンがノーマルモードで起動されて、サンプルレートは **dmconfig.ini** のキーワード **DB_StsSp** の値を使用します ; 1 に設定すると、各表の設定モードで更新統計デッモンを起動して、サンプルレートは各表のモードと各表のサンプルレートに決定されます。ユーザーはコマンド"SET TABLE STATISTICS"を使用して各表のモードとサンプルレートを設定することができます。

初期値: 1

有効値の範囲: 0、1

関連キーワード: DB_StSvr、DB_StsSp、DB_StsTm、DB_StsTv

使用する場所: サーバー側

DB_STPWD=<文字列>

このキーワードは、DBMaster のインストールディレクトリの*共有/ストップワード*サブディレクトリに保存されているストップワードリスト定義ファイルの名前を示しています。ストップワードリスト定義ファイルは、DBMaster のテキストインデックスの結果に影響するテキストファイルです。このキーワードは、データベースがテキストインデックスを作成および検索するときに使用されます。このキーワードがないと、データベースは LCode に基づく事前定義のストップワードリスト定義を検索します。

➲ **例**

```
[DB1]
DB_LCode = 2
DB_StpWd = /home/usr/jp.tab
```

初期値:

DB_LCode	ストップワードリスト定義
0: 英語 (ASCII)	en.tab
1: 繁体字中国語 (BIG5)	tw.tab
2: 日本語 (Shift JIS + 半角)	jp.tab
3: 簡体字中国語 (GB)	cn.tab
4: ラテン語 1 コード (ISO-8859-1)	en.tab
5: ラテン語 2 コード (ISO-8859-2)	en.tab
6: キリル文字コード (ISO-8859-5)	en.tab
7: ギリシャ語コード (ISO-8859-7)	en.tab
8：日本語文字 (EUC-JP)	jp.tab

9：簡体中国語文字 (GB18030)　　cn.tab

10：UTF-8(UTF-8)　　en.tab

有効値の範囲: ユーザー定義のストップワードリスト定義ファイルのファイル名

関連のキーワード: DB_LCode

使用する場所: サーバー側

DB_STROP=<値>

このキーワードは、文字列連結演算子(||)を適用する前に、パディングスペースを削除するかどうかを指定します。値を 0 にすると、文字列連結演算子を適用する前に、固定長 CHAR データ型のパディングスペースを残すことを意味します。値を 1 にすると、文字列連結演算子を適用する前に、パディングスペースを削除することを意味します。

このキーワードは、クライアントとサーバーいずれでも設定することができます。このキーワードの値が、クライアント側の **dmconfig.ini** ファイルに設定されていない場合、サーバー側の **dmconfig.ini** ファイルに依存します。サーバーの初期設定値は 0 です。

初期値: 0

有効値の範囲: 0、1

使用する場所: クライアントとサーバー側

DB_STRSZ=<値>

このキーワードは、ユーザー定義関数(UDF)にのみ使用される STRING データ型の戻りデータの長さを指定します。UDF は、固定長のデータしか戻すことができないので、このキーワードにより、クライアントが長すぎる文字列を受け取ることを避けるために、STRING のサイズを制限することができます。

初期値: 255

有効値の範囲: 1 ~ 4096

使用する場所: クライアントかサーバー側(クライアントに優先権があります)

DB_STSSP=<値>

このキーワードは更新統計ページのデータのサンプルレートを指定します。このキーワードを-1 に設定する場合、知能的にサンプルレートが取得できますが、0 に設定する場合、統計値を更新しません。また、ユーザーは自主的にサンプルレートを設定することもでき、値の範囲は 1～100 です。

初期値: 100

有効値の範囲: -1、0、1 ~ 100

関連のキーワード：DB_StSvr、DB_StsTv、DB_StsTm、DB_StMod

使用する場所: サーバー側

DB_STSTM=<文字列>

このキーワードは更新統計デッモンの初期時間を指定します。**DB_StsTm** のフォーマットは **yyyy-mm-dd hh:mm:ss** です。

初期値: 1970-01-01 03:00:00

有効値の範囲: 1970-01-01 00:00:01 ~ 2037-12-31 23:59:59

関連のキーワード：DB_StSvr、DB_StsTv、DB_StsSP、DB_StMod

使用する場所: サーバー側

DB_STSTV=<文字列>

このキーワードは更新統計デッモンの時間間隔を指定します。値 "1-12:30:00" は時間間隔が一日 12 時間 30 分だという意味です。

初期値:1-00:00:00

有効値の範囲:0-00:00:01~24854-23:59:59

関連のキーワード: DB_StSvr、DB_StsTm、DB_StsSP、DB_StMod

使用する場所: サーバー側

DB_STSVR=<値>

このキーワードは、自動統計更新サーバーを作動させるために使用します。値を 1 に設定するとサーバーが起動します。値を 0 に設定するとサーバーが起動しないことを意味します。自動統計更新サーバーが作動する場合、**DB_StsTm** と **DB_StsTv** の値によってデータベースの統計を再計算します。

初期値: 1

有効値の範囲: 0、1

関連のキーワード: DB_StsTv、DB_StsTm、DB_StsSP、DB_StMod

使用する場所: サーバー側

DB_SVADR=<文字列>

このキーワードは、サーバー・マシンの TCP/IP アドレス、またはマシンのホスト名の文字列を指定します。DNS（ドメイン名サービス）が正しくクライアント・マシンに設定されているときは、ドメイン名を指定することもできます。このキーワードは、全てのクライアント・マシンで接続時に必要になります。TCP/IP アドレスが正しくないと接続は失敗します。詳細については、ネットワーク管理者に尋ねるか、TCP/IP ネットワークのマニュアルを参照してください。

初期値： なし

有効値の範囲： a、b、c、d またはホスト（ドメイン）名（1<=a、b、c、d <=254）

関連キーワード： DB_PtNum

使用する場所: サーバーとクライアント側両方

DB_SVLOG =<値>

このキーワードは、dmserver コマンドラインのテキストがファイルに保存するか指定します。この機能が起動した後、*<データベースディレクトリ>*\<*データベース名*>.log は ASCII ファイルフォーマットとして保存されます。DBA 権限を持つユーザーはこの機能を使用して接続エラーを扱うことがで

きます。値 1 はファイルが保存されたと表示します、0 は保存されないという意味です。

初期値: 0

有効値の範囲: 0、1

使用する場所: サーバー側

DB_TCPIP=<値>

このキーワードは、dmserver が TCP/IP ネットワークプロトコルのみを使用するかどうかを指定します。その値を 0 に設定する場合、dmsever は名前付きパイプ及び TCP/IP ネットワークプロトコルを使用して接続しますが、1 に設定する場合、dmsever は TCP/IP ネットワークプロトコルのみを使用し接続します。

初期値: 0

有効値の範囲: 0、1

使用する場所: サーバー側

DB_TMIFM=<文字列>

このキーワードは、SQL 文の時刻入力フォーマットを指定します。詳細については、*「ODBC プログラマーガイド」の付録 B* を参照してください。

初期値： なし（全ての時刻入力フォーマットを受理します）

有効値の範囲： hh:mm:ss.fff

hh:mm:ss

hh:mm

hh

hh:mm:ss.fff tt

hh:mm tt

hh tt

tt hh:mm:ss.fff

tt hh:mm:ss

tt hh:mm

tt hh

関連キーワード： DB_TmoFm

使用する場所: クライアントかサーバー側(クライアント側に優先権があります)

DB_TMOFM=<文字列>

このキーワードは、SQL 文の時刻出力フォーマットを指定します。詳細については、*「ODBC プログラマーガイド」*を参照してください。

初期値：hh:mm:ss

有効値の範囲： DB_TmiFm の有効値の範囲と同じ

関連キーワード：DB_TmiFm

使用する場所: クライアントかサーバー側(クライアント側に優先権があります)

DB_TMPDIR=<文字列>

このキーワードは TMPTABLESPACE のファイルの保存ディレクトリを表示します。システムはデータファイルと BLOB ファイルをこのディレクトリに作成します。データファイルのロジック名は **DB_TMPDB** で、物理名は **DB_TMPDir/DBNAME.TDB** です；BLOB ファイルのロジック名は **DB_TMPBB** で、物理名は **DB_TMPDir/DBNAME.TBB** です。

DB_TMPDB と **DB_TMPBB** はキーワードではなく、予約語です。**dmconfig.ini** ファイルに **DB_TMPDB** と **DB_TMPBB** を設定することができません。

初期値：DB_DbDir/tmpDir

有効値の範囲：DB_DbDir

使用する場所：サーバー側

DB_TPFIL=<文字列>

このキーワードは、システム一時ファイルの名前を指定します。 ファイルサイズの制限は 4GB です。 ユーザーは、最大 8 つのシステム一時ファイルを指定できます。

初期値 ：*.TMP* のファイル拡張子のあるデータベース名。

有効範囲：1 スペースの後の 1 つのコンマで区切られた、任意の長さの最大 8 つの文

関連キーワード： DB_DbFil、DB_BbFil

使用する場所：サーバーサイド

DB_TSKNO=<値>

このキーワードは同じ時期に dmschsvr によって喚起されるタスクの数量を示しています。

初期値：30

有効値の範囲：1~50

関連キーワード: DB_SchSv

使用する場所:サーバー側

DB_TURBO=<値>

このキーワードは、ノーマル・カタログ・キャッシュ操作で DBMaster を走行させるかどうかを指定します。データベース構造を変更することがほとんど無いアプリケーションの場合、**DB_Turbo** モードを使用してデータへのアクセスを高速化することができます。詳細については、*「パフォーマンスのチューニング」*の章を参照してください。このキーワードは、データベースの起動時に使用されます。

初期値： 0

有効値の範囲： 0、1

使用する場所:サーバー側

DB_USRBB=<文字列>

このキーワード、特別なユーザー定義ファイル名は、オペレーティングシステムが使用する初期設定ユーザーBLOB ファイルの物理名を指定します。

この物理名の後、ユーザーはファイルサイズを定義します。ここには三つのユニットオプションがあります：フレーム、M（メガバイト）、G（ギガバイト）。M 或いは G を使用する時、実際値は定義サイズより 1 フレームを少なくなります。初期値ユニットはフレームです。

➲ **例**

20 フレームの初期設定ユーザーBLOB ファイルを定義する:

```
[MY_DB]                                    ;データベース名
DB_USRBB = /disk1/usr/f1.bb  20            ;blob ファイル
```

初期値:*データベース名.BB が（3 フレーム）。例えば、*db.BB。

有効値の範囲:文字列の長さ < 256

関連キーワード: DB_BbFil、DB_DbDir、DB_DbFil、DB_UsrDb、ユーザー定義ファイル名。

使用する場所: サーバー側

DB_USRDB=<文字列>

このキーワード、特別なユーザー定義ファイル名は、オペレーティング・システムが使用する初期設定ユーザー・データファイルの物理名を指定します。

この物理名の後、ユーザーはファイルサイズを定義します。ここには三つのユニットオプションがあります：ページ、M（メガバイト）、G（ギガバイト）。M 或いは G を使用する時、実際値は定義サイズより 1 フレームを少なくなります。初期値ユニットはページです。

➲ 例 1

200 ページの初期設定ユーザー・データファイル名を定義する。

```
[MY_DB]                                    ;データベース名
DB_USRDB = /disk1/usr/f1.db  200           ;データファイル
```

初期値:*データベース名.DB（200 ページ）。例えば*、db.DB。

有効値の範囲:文字列の長さ < 80

関連キーワード: DB_BbFil、DB_DbDir、DB_DbFil、DB_UsrBb、ユーザー定義ファイル名

使用する場所: サーバー側

DB_USRFO=<値>

このキーワードは、ユーザー・ファイルオブジェクトをデータベースに挿入するかどうかを指定します。値を 1 にすると、ユーザー・ファイルオブジェクトは使用可能にします。詳細については、「ラージオブジェクト管理」の章を参照してください。このキーワードは、データベースの起動時に必要になります。

初期値： 0

有効値の範囲： 0、1

関連キーワード： DB_FoDir

使用する場所:サーバー側

DB_USRID=<文字列>

このキーワードは、データベースの起動時または接続時のログインに使用される初期設定ユーザーID を指定します。

初期値： Null 文字列

有効値の範囲：長さ 128 以下の文字列

関連キーワード： DB_PasWd

使用する場所: クライアント側

DB_WSORT = <値>

このキーワードはワードソートオーダーの大文字と小文字の識別を指定します。初期値は 0 にします。値は 0 にすると、ワード･ソート･コードは大文字と小文字が識別しないです；値は 2 にすると、ワードソートコードは大文字と小文字の識別です。

初期値: 0

有効値の範囲: 0、1、2

関連キーワード: DB_LCode、DB_Order

使用する場所: サーバ側

DD_CTIMO=<値>

このキーワードは、DDB 環境でリモート・データベースに接続する際の接続タイムアウト時間を指定します。DDB 環境では、コーディネータ・データベースのサーバーは、リモート・データベースに分散型の接続を設ける必要があるかもしれません。

初期値: 5 (秒)

有効値の範囲: >= 1

関連キーワード: DD_DDBMd、DD_LTimO、DB_CTimO

使用する場所: サーバー側

DD_DDBMD=<値>

このキーワードは、DDB(分散データベース)機能が、データベース・サーバーで使用するかどうかを指定します。DDB 操作又は表レプリケーション機能を使用する場合は、このキーワードの値を 1 にし、ON にします。

初期値: 0

有効値の範囲: 0、1

関連キーワード: DD_GTSvr

使用する場所: サーバー側

DD_GTITV=<文字列>

このキーワードは、中断グローバル・トランザクションを解決する GTRECO デーモンのスケジュールを指定します。GTRECO サーバーが ON の時のみ使用します。入力フォーマットは、**D-hh:mm:ss** です。

初期値: 00:10:00

有効値の範囲: 0 日-24855 日

関連キーワード: DD_GTSvr

使用する場所: サーバー側

DD_GTSVR=<値>

このキーワードは、DDB モードが ON の時に、GTRECO(グローバル・トランザクション・リカバリ)デーモンを起動するかどうかを指定します。GTRECO デーモンは、DBMaster データベース・サーバーを経由する中断グローバル・トランザクションを自動的に解決します。

初期値: 1

有効値の範囲: 0、1

関連キーワード: DD_DDBmd、DD_GTItv

使用する場所: サーバー側

DD_LTIMO=<値>

このキーワードは、DDB 操作時に、分散データへのアクセスの際のロック・タイムアウト時間を指定します。これは、サーバーとサーバー間のデータ・アクセスにのみ影響します。タイムアウト時間の詳細は、**DB_LTimO** を参照して下さい。

初期値：5 (秒)

有効値の範囲： >= -1

関連キーワード： DD_DDBmd、DD_CTimO、DB_LTimO

使用する場所: サーバー側

DM_DIFEN=<値>

このキーワードは、必要な時に新規環境ハンドルを割り当てるかどうかを指定するために、DM_COMMON_OPTION のグローバル・セクションにセットする必要があります。dmconfig.ini ファイルにある DM_COMMON_OPTION のグローバル・セクションは、複数データベース間のグローバル設定に使用します。DM_DIFEN のようなキーワードは、それが定義された dmconfig.ini ファイルの全データベースに適用されます。特殊なケースで DBMaster のカスタマーサポートが言及しない限り、これを変更しないで下さい。

初期値: 1

有効値の範囲: 0、1

使用する場所: クライアント側

LG_NPFUN=<文字列>

このキーワードは、DM_COMMON_OPTION セクションでのみセットし、ログされない関数を指定します。この値は、空白かカンマ(,)で仕切られた ODBC 関数名です。このキーワードは、LG_PTFUN が dmconfig.ini ファイルで定義されていない場合のみ有効です。このキーワードを一旦指定すると、文字列に列記される関数は、ログされません。

初期値: "" (空白、全関数はログされます。)

有効値の範囲: 関数リスト文字列 (例 "SQLError、SQLGetDiagRec")

関連キーワード: LG_Path、LG_PTFun、LG_Trace、LG_Time

使用する場所: クライアント側

LG_PATH=<文字列>

このキーワードは、DM_COMMON_OPTION セクションでのみセットされ、出力ファイルのファイル・パス名を指定します。

初期値: c:\odbclog.log (Win32) 、./odbclog.log (UNIX)

有効値の範囲: 文字列の長さ < 256

関連キーワード: LG_NPFun、LG_PTFun、LG_Time、LG_Trace

使用する場所: クライアント側

LG_PTFUN=<文字列>

このキーワードは、DM_COMMON_OPTION セクションでセットし、ログを取る関数を指定します。この値は、空白かカンマ(,)で仕切られた ODBC 関数名です。このキーワードの値を一旦指定すると、文字列に列記された関数のみ、ログされます。LG_PTFun と LG_NPFun がセットされた場合、LG_PTFun にのみ影響します。

初期値:なし (関数は全てログされます。)

有効値の範囲: 関数リスト文字列 (例 "SQLError、SQLGetDiagRec")

関連キーワード: LG_NPFun、LG_Path、LG_Time、LG_Trace

使用する場所: クライアント側

LG_TIME=<値>

このキーワードは、DM_COMMON_OPTION セクションでセットし、各関数に費やす時間をログするかどうかを指定します。この値を 1 にセットすると、出力ログファイルに各関数に費やす時間をログし、0 にセットすると、時間をログしません。この情報を使って、ODBC プログラムにあるパフォーマンスのボトルネックを見つけることができます。

初期値: 0

有効値の範囲: 0、1

関連キーワード: LG_NPFun、LG_Path、LG_PTFun、LG_Trace

使用する場所: クライアント側

LG_TRACE=<値>

このキーワードは、DM_COMMON_OPTION セクションでセットし、ODBC log を ON にするか OFF にするかを指定します。この値を 1 に設定すると、ODBC log を ON にし、0 にすると OFF にします。ODBC ログが ON の時、コールされた ODBC 関数、入力パラメータ、出力パラメータ、戻されたコード又はエラー情報は、指定したファイルにログされます。詳細については、以下のキーワードを参照して下さい。

初期値: 0

有効値の範囲: 0、1

関連キーワード: LG_NPFun、LG_Path、LG_PTFun、LG_Time

使用する場所: クライアント側

RP_BTIME=<値>

このキーワードは、データベース・レプリケーションの開始日時を指定します。フォーマットは、YYYY/MM/DD hh:mm:ss、例えば 2000/1/1 01:30:00 のように使用します。データベース・レプリケーションの間隔は、RP_Iterv で指定することができます。

初期値: ソース・データベースの起動時間

有効値の範囲: YYYY/MM/DD hh:mm:ss (例 2000/1/1 00:00:00)

関連キーワード: DB_SMode、RP_Clear、RP_Iterv、RP_Primy、RP_PtNum、RP_ReTry、RP_SlAdr

使用する場所: ソース・データベースのサーバー側

RP_CLEAR=<値>

このキーワードは、データベース・レプリケーションに使用します。リモート・データベースにバックアップ・ファイルをレプリケートした後、バ

ックアップ・ファイルを消去するかどうかを指定します。値を 1 にすると、ファイルを消去し、0 にするとそれらを残します。

初期値: 0 (クリアしない)

有効値の範囲: 0 (クリアしない)、1 (クリアする)

関連キーワード: DB_SMode、RP_BTime、RP_Iterv、RP_Primy、RP_PtNum、RP_ReTry、RP_SlAdr

使用する場所: ソース・データベースのサーバー側

RP_ITERV=<値>

このキーワードは、データベースレプリケーションのスケジュールを指定します。フォーマットは dd-hh:mm:ss で、例えば 1-12:00:00（1 日半）のように使用します。**RP_BTime** でデータベースレプリケーションの開始日時を指定することができます。

初期値: 1-00:00:00 (1 日)

有効値の範囲: 0 日 ~ 24855 日

関連キーワード: DB_SMode、RP_BTime、RP_Clear、RP_Primy、RP_PtNum、RP_ReTry、RP_SlAdr

使用する場所: ソース・データベースのサーバー側

RP_LGDIR=<文字列>

このキーワードは、非同期表レプリケーションのログファイルを置くディレクトリを指定します。このレプリケーション・ログファイルは、バイナリで、ユーザーは任意にそれらを削除することができません。

初期値:データベース・ホーム・ディレクトリの下の **TRPLOG** という名前のサブディレクトリ

有効値の範囲: 文字列の長さ < 256

関連キーワード: DB_AtrMd、DB_EtrPt

使用する場所: ソース・データベースのサーバー側

RP_PRIMY=<文字列>

このキーワードは、データベース・レプリケーションに使用し、ソース・データベースのアドレスを指定します。

初期値: なし

有効値の範囲: a、b、c、d 又はホスト (ドメイン)名 (1<=a、b、c、d <=254)

関連キーワード: DB_SMode、RP_BTime、RP_Clear、RP_Iterv、RP_PtNum、RP_ReTry、RP_SlAdr

使用する場所: ターゲットデータベースのサーバー側

RP_PTNUM=<値>

このキーワードは、データベース・レプリケーションに使用し、ターゲット・データベースの RP_RECV デーモンのポート番号を指定します。ターゲット・データベースで使用する **DB_PtNum** と異なる番号で、ソース・データベースの **RP_SlAdr** で指定されるポート番号と同一にします。

初期値: 23001

有効値の範囲: 1024~65535

関連キーワード: DB_SMode、RP_BTime、RP_Clear、RP_Iterv、RP_Primy、RP_ReTry、RP_SlAdr

使用する場所: ターゲットデータベースのサーバー側

RP_RESET=<値>

このキーワードは、データベース起動の際に非同期表レプリケーション・システムをリセットするよう指定します。値を 1 に設定すると、未送信の非同期表レプリケーションの全ログは消去され、**RP_LgDir**(初期設定は DB_DbDir/TRPLOG)ディレクトリにある全**.TRP** ファイルは削除され、データベース起動の際に非同期表レプリケーションの状態はリセットされます。つまり、未送信の非同期表レプリケーション・データは無視されます。次にデータベースを起動した際に非同期表レプリケーションがリセットされ

るのを防ぐために、データベース起動後に **RP_Reset** の値は、0 にリセットされます。

初期値: 0

有効値の範囲: 0、1

関連キーワード: RP_LgDir

使用する場所: ソース・データベースのサーバー側

RP_RETRY=<値>

このキーワードは、データベース・レプリケーションに使用し、ネットワーク障害の際、リモート・データベースに接続を試行する回数を指定します。

初期値: 0

有効値の範囲: 0 ~2147483647

関連キーワード: DB_SMode、RP_BTime、RP_Clear、RP_Iterv、RP_Primy、RP_PtNum、RP_SlAdr

使用する場所: ソース・データベースのサーバー側

RP_SLADR=<文字列>

このキーワードは、データベース・レプリケーションに使用し、ソース・データベースがデータを送信するターゲット・データベースを指定します。DBMaster は、各ソース・データベースに 1 から 8 ターゲット・データベースまで指定することができます。

➲ **例 1**

RP_SlAdr の構文は以下のようになります。

```
RP_SLADR = アドレス[:ポート番号] {, アドレス[:ポート番号]}
```

初期設定のポート番号は 23001 です。カンマやスペースで、ターゲットデータベース用の情報を仕切ることができます。

➲ **例 2**

RP_SlAdr ポート番号は以下のようになります。

```
RP_SLADR = 192.168.9.222:5100, Server2:5101, Server3
```

この例では、3 つのターゲットデータベースがあります。1 つは、ポート番号が 5100 の 192.168.9.222、2 つ目は、ポート番号が 5101 の Server2、3 つ目は、初期設定のポート番号 23001 の Server3 です。

初期値: なし

関連キーワード: DB_SMode、RP_BTime、RP_Clear、RP_Iterv、RP_Primy、RP_PtNum、RP_ReTry

使用する場所: ソース・データベースのサーバー側

ユーザー定義ファイル名=<物理ファイル名> <ページ数>

ユーザー定義ファイルは、表領域を使用し尽くしたときにデータファイルや BLOB ファイルを作成して表領域に追加するときに使用します。基本的には、DBMaster ファイルを作成するときは、パス名のない論理ファイル名を指定します。ユーザー定義ファイルは、論理ファイル名を物理ファイル名（オペレーティング・システムのファイル名）に対応づけます。

➲ **dmconfig.ini** でファイルにマップする：

```
FILE1 = /disk1/usr/datafile 100
```

ユーザーは DBMaster に対して論理ファイル ***FILE1*** を指定しますが、DBMaster は 100 ページ（データベースを作成する時、ページのサイズは DB_PGSIZ にて設定します。）の物理ファイル***/disk1/usr/datafile*** を作成します。このファイルを他のディレクトリに移動するときは、***dmconfig.ini*** の物理ファイル名のみ変更するだけで、プログラムや SQL スクリプトを変更する必要はありません。ユーザー定義ファイルに対しても、DB_DbDir のルールが適用される点に注意してください。

有効値の範囲： ファイル名 —文字列の長さ < 256

ページ数 — 3 ~ 2147483647

関連キーワード： DB_DbDir、DB_UsrBb、DB_UsrDb

使用する場所: サーバー側

B. システムカタログ参照

リレーショナルデータベースの定義に「データベースの全ての情報はユーザーのデータと同様に論理レベルで表現されなければならない」ということがあります。データベースの情報は*システムカタログ*に格納されます。認証されたユーザーは、SQL で表のデータをアクセスするのと同じ方法でデータベース情報をアクセスすることができます。システムカタログ参照は、アルファベット順にシステムカタログの表とビューを説明します。システムカタログ表に問い合わせることによって、データベースの詳細な状態を知ることができます。

B.1 システムカタログ

システムカタログは、データベースのオブジェクト情報をもつ表の集まりです。システムカタログは*データディクショナリ*とも言われます。

全てのシステムカタログは SYSTEM が所有し、CONNECT 権限をもつ全てのユーザーがアクセスすることができます。ただし、システムカタログは SYSTEM に属するので、システム表やシステム定義カラムを削除したり、システム表の行を INSERT、DELETE したりすることはできません。

DBA、SYSDBA と SYSADM 権限のユーザーは全てのシステムカタログ表が読めます。RESOURC と CONNECT 権限を持つユーザーは以下の十つのシステムカタログ表を読むことができません： SYSAUTHGROUP、SYSCONFIG、SYSFILE、SYSFILEOBJ、SYSGLBTRANX、SYSLOCK、SYSPENDTRANX、SYSTRPJOB、SYSTRPPOS、SYSWAIT。

権限によって以下の 31 個表に読む権限も違います:SYSACL、SYSAUTHCOL、SYSAUTHEXE、SYSAUTHMEMBER、SYSAUTHTABLE、SYSAUTHUSER、SYSCMDINFO、SYSCOLUMN、SYSCONINFO、SYSDBLINK、SYSFOREIGNKEY、SYSINDEX、SYSINDEXREF、SYSINFO、SYSJARFILE、SYSJAVAARGU、 SYSOPENLINK、SYSPROCINFO、SYSPROCJAVA、SYSPROCPARAM、SYSPROJECT、SYSPUBLISH、SYSSCHEMA、SYSSUBSCRIBE、SYSSYNONYM、SYSTABLE、SYSTABLESPACE、SYSTEXTINDEX、SYSTRIGGER、SYSUSER、SYSVIEWDATA。

全てのユーザーは下記の六つのシステムカタログ表を読むことができます:SYSDOMAIN、SYSSCHEDULE、SYSSCHELOG、SYSTASK、SYSTRPDEST、SYSUSERFUNC。

システムカタログ表の数は 47 です。

2.2 DBMaster のシステムカタログ表

以下に DBMaster がもつ全てのシステムカタログ表と、各表の内容の要約をリストします。

表名	内容
SYSACL	IP アドレス権限情報
SYSAUTHCOL	カラム権限情報
SYSAUTHEXE	実行可能オブジェクト権限情報
SYSAUTHGROUP	グループ情報
SYSAUTHMEMBER	グループメンバー情報
SYSAUTHTABLE	表権限情報
SYSAUTHUSER	セキュリティレベル情報
SYSCMDINFO	ストアドコマンド情報
SYSCONINFO	コンソール情報
SYSCOLUMN	カラム情報

表名	内容
SYSCONFIG	構成情報
SYSCONINFO	接続情報
SYSDBLINK	データベースリンク情報
SYSDOMAIN	ドメイン情報
SYSFILE	ファイル情報
SYSFILEOBJ	ファイルオブジェクト情報
SYSFOREIGNKEY	外部キー情報
SYSGLBTRANX	DDB グローバル・トランザクション情報
SYSINDEX	索引情報
SYSINDEXREF	インデックス及び自動インデックスの情報
SYSINFO	データベース・システム情報
SYSJARFILE	JAR 情報
SYSJAVAARGU	Java の引数情報
SYSLOCK	ロック情報
SYSOPENLINK	オープンリンク情報
SYSPENDTRANX	中断分散トランザクション情報
SYSPROCINFO	ストアド・プロシージャ情報
SYSPROCJAVA	JavaSP 情報
SYSPROCPARAM	ストアド・プロシージャのパラメータ情報
SYSPROJECT	ESQL プロジェクト情報
SYSPUBLISH	表レプリケーションのソース情報
SYSSCHEDULE	スケジュールタスク情報
SYSSCHELOG	実行されたスケジュールタスクを記録する情報

表名	内容
SYSSUBSCRIBE	表レプリケーションのターゲット情報
SYSSYNONYM	シノニム情報
SYSTABLE	表情報
SYSTABLESPACE	表領域情報
SYSTASK	スケジュールのアクション情報
SYSTEXTINDEX	テキスト索引情報
SYSTRIGGER	トリガー情報
SYSTRPDEST	表レプリケーション情報
SYSTRPJOB	レプリケートされる全作業を記録するための情報
SYSTRPPOS	トランザクションのログファイルを簡潔にするディストリビュータ情報
SYSUSER	データベースにログインしたユーザーの情報
SYSUSERFUNC	ユーザー定義関数の情報
SYSVIEWDATA	ビュー情報
SYSSCHEMA	スキーマ情報
SYSWAIT	接続待ち情報

SYSACL

SYSACL テーブルはユーザーが接続できる IP アドレスを一覧表示します。RESOURCE 或いは CONNECT 権限を持つユーザーは自分関連の情報のみ読めます。SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

テーブルにはユーザー名と IP アドレスが含まれます。ユーザー名 "PUBLIC" は すべてのユーザーが設定を満足しなければならないことを意味します。"*" はすべての IP アドレスが接続可能であることを意味します。

カラム名	説明
USER_NAME	ユーザーの名前
ADDRESS	IP アドレス
PRIVILEGE	許可—この IP アドレスが許可される。 ブロックーこの IP アドレスがブロックされる。

SYSAUTHCOL

SYSAUTHCOL 表には、オブジェクト権限がユーザーに与えられている表の全てのカラム情報があります。ユーザーは特定的なカラムの操作権限を持つ、また RESOURCE 或いは CONNECT 権限を持つと、このカラムの権限が与えられている情報が読めます。SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。ユーザーが表の全てのカラムに対して INSERT、UPDATE、REFERENCE 権限をもっている（SYSAUTHTABLE 表の INS_ALL、UPD_ALL、REF_ALL の値が 1）場合は、SYSAUTHCOL の INS、UPD、REF の値は無視されます。

カラム名	**解説**
COLUMN_NAME	権限が与えられているカラム名。
TABLE_NAME	カラムが属する表名。
GRANTEE	カラム権限が与えられているユーザー名またはグループ名。
TABLE_OWNER	表を作成したユーザー名。
INS	**1** — ユーザーには指定カラムにデータを挿入する権限があります。 **0** — ユーザーには指定カラムにデータを挿入する権限がありません。

カラム名	解説
UPD	**1** — ユーザーには指定カラムのデータを更新する権限があります。 **0** — ユーザーには指定カラムのデータを更新する権限がありません。
REF	**1** — ユーザーには指定カラムを参照する制約を作成する権限があります。 **0** — ユーザーには指定カラムを参照する制約を作成する権限がありません。

SYSAUTHEXE

SYSAUTHEXE 表には、実行可能オブジェクトの権限情報があります。ユーザーは特定的なカラムの操作権限を持つ、また RESOURCE 或いは CONNECT 権限を持つと、このカラムの権限が与えられている情報が読めます。SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
OBJNAME	実行可能オブジェクト名。
OWNER	実行可能オブジェクトを作成したユーザー。
OBJTYPE	「Procedure」、「Command」、「Project」等の実行可能オブジェクトの種類。
GRANTEE	実行可能オブジェクト権限を持つユーザー名。

SYSAUTHGROUP

SYSAUTHGROUP 表には、データベースに定義されている全てのグループの情報があります。RESOURCE 或いは CONNECT 権限を持つユーザーは全ての情報を見ることができません。システムはエラー6829 を返して select 権限がないと表示します。DBA 以上の権限を持つユーザーはグループが作成

できます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
GROUP_NAME	グループ名。
GROUP_OWNER	グループを作成したユーザー。
NUM_MEMBERS	グループのメンバー数。

SYSAUTHMEMBER

SYSAUTHMEMBER 表には、グループに属する全てのメンバーのリストがあります。RESOURCE 或いは CONNECT 権限を持つユーザーは自分と関連のグループ情報のみ読めます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
MEMBER_NAME	グループに属するメンバーの名前。
GROUP_NAME	グループ名。

SYSAUTHTABLE

SYSAUTHTABLE 表には、表に対して認められる権限と、権限が与えられたユーザーの情報があります。ユーザーは特定的なカラムの操作権限を持つ、また RESOURCE 或いは CONNECT 権限を持つと、このカラムの権限が与えられている情報が読めます。SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の権限情報が読めます。

カラム名	解説
TABLE_NAME	権限が与えられている表またはビューの名前。
GRANTEE	表権限が与えられているユーザーの名前。

カラム名	解説
TABLE_OWNER	表またはビューを作成したユーザー。
NUM_RPI_COLS	表またはビューの中の権限が与えられているカラムの個数。
SEL_ALL	1—ユーザーには表またはビューの全カラムのデータを検索する権限があります。 0—ユーザーには表またはビューの全カラムのデータを検索する権限がありません。
DEL_ALL	1—ユーザーには表またはビューの全カラムのデータを削除する権限があります。 0—ユーザーには表またはビューの全カラムのデータを削除する権限がありません。
INS	1—ユーザーには表またはビューの特定のカラムにデータを挿入する権限があります。 0—ユーザーには表またはビューの特定のカラムにデータを挿入する権限がありません。
INS_ALL	1—ユーザーには表またはビューの全てのカラムにデータを挿入する権限があります。 0—ユーザーには表またはビューの全てのカラムにデータを挿入する権限がありません。特定のカラムに挿入する権限はあります（INS を参照）。
UPD	1—ユーザーには表またはビューの特定カラムのデータを更新する権限があります。 0—ユーザーには表またはビューの特定カラムのデータを更新する権限はありません。
UPD_ALL	1—ユーザーには表またはビューの全カラムのデータを更新する権限があります。 0—ユーザーには表またはビューの全カラムのデータを更新する権限がありません。特定のカラムを更新する権限はあります（UPD を参照）。

カラム名	解説
ALT_ALL	**1** — ユーザーには表またはビューの定義を変更する権限があります。 **0** — ユーザーには表またはビューの定義を変更する権限がありません。
IDX_ALL	**1** — ユーザーには表の索引を作成、削除する権限があります。 **0** — ユーザーには表の索引を作成、削除する権限がありません。
REF	**1** — ユーザーには表またはビューの特定カラムを参照する制約を作成する権限があります。 **0** — ユーザーには表またはビューの特定カラムを参照する制約を作成する権限がありません。
REF_ALL	**1** — ユーザーには表またはビューの全カラムを参照する制約を作成する権限があります。 **0** — ユーザーには表またはビューの全てのカラムを参照する制約を作成する権限がありません。特定のカラムを参照する権限はあります（REF を参照）。

SYSAUTHUSER

SYSAUTHUSER 表には、データベースに登録されている全てのユーザー名とその権限レベルの情報があります。RESOURCE 或いは CONNECT 権限を持つユーザーは自分の情報が読めます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
USER_NAME	CONNECT 権限が与えられているユーザーの ID。データベース・ユーザーは、CONNECT 権限が与えらると正当とみなされます。

DBA	0—DBA 権限を持っていません。 1—DBA 権限を持っています。 2—SYSDBA 権限を持っています。
RESOURCE	0—RESOURCE 権限を持っていません。 1—RESOURCE 権限を持っています。
ACLORDER	0—ホワイトリストベース。 1—ブラックリストベース。

SYSCMDINFO

SYSCMDINFO 表には、ストアド・コマンド情報があります。ユーザーは特定的な操作権限を持つ、また RESOURCE 或いは CONNECT 権限を持つと、このストアドコマンドの権限が与えられている情報が読めます、そして作成情報も取れます。SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
MODULENAME	モジュール名。（このカラムは、ESQL アプリケーションやストアド・プロシージャで使用します。純粋なストアド・コマンドの場合は、無視されます。）
CMDNAME	ストアド・コマンド名。
CMDOWNER	ストアド・コマンドの所有者。
STATEMENT	元の SQL 文。
NUM_PARM	パラメータ数。
STATUS	1— 有効 0—無効
REBTIME	ストアド・コマンドを再作成する時間。
CMDPLAN	格納されたストアド・コマンドの実行計画文字列。

SYSCOLUMN

SYSCOLUMN 表には、全ての表とビューにある全てのカラム情報があります。ユーザーは特定的なカラムの操作権限を持つ、また RESOURCE 或いは CONNECT 権限を持つと、自分の表のカラム情報とこのカラムの権限が与えられている情報が読めます。SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。システムカタログ表のカラムも含まれます。SCALE と RADIX が適用されないデータ型のときは、SCALE と RADIX のカラムには、-1 が返されます。

カラム名	解説
COLUMN_NAME	カラム名。
TABLE_NAME	カラムが属する表名。
TABLE_OWNER	表を作成したユーザー名。
COLUMN_ORDER	表内のカラムの順序番号。
NULLABLE	**1**—NULL 値が認められます。 **0**—NULL 値が認められません。
TYPE_NAME	カラムのデータ型。BINARY、CHAR 、NCHAR 、DATE 、DECIMAL 、DOUBLE 、FILE 、FLOAT 、INTEGER 、LONG VARCHAR、NCLOB 、LONG VARBINARY、SERIAL 、SMALLINT 、TIME 、TIMESTAMP 、VARCHAR 、NVARCHAR のいずれかです。
PRECISION	カラムの精度。
SCALE	カラムのスケール。
RADIX	カラムの基数
ASCII_DEF	カラムの ASCII 形式の初期設定値。
CONSTR	カラム制約。
REMARKS	カラムの記述。

SYSCONFIG

SYSCONFIG 表には全部のサーバー構成設定をリストします。RESOURCE 或いは CONNECT 権限を持つユーザーは当該表の如何なる情報を読み込むことができません。システムはエラー6829 を返して select 権限がないと表示します。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	**解説**
KEYWORD	dmconfig.ini に使用されるキーワード。
VALUE	現在の設定値

SYSCONINFO

SYSCONINFO 表には、データベースの接続に関する情報があります。RESOURCE 或いは CONNECT 権限を持つユーザーは自分の接続情報のみ読めます。SYSDAM または DBA、SYSDBA 権限を持つユーザーも自分の接続情報のみ読めますので、ご注意ください。

カラム名	**解説**
CONNECTION_ID	接続 ID
INFO1	予備
LAST_SERIAL	更新した SERIAL データ型のカラムで作動している最後のシリアル番号
LAST_OID	最後に挿入されたレコードのオブジェクト ID (OID)
INFO2	予備
INFO3	予備

SYSDBLINK

SYSDBLINK 表には、リモート・データベースリンク情報があります。RESOURCE 或いは CONNECT 権限を持つユーザーは作成した情報を取れます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
OWNER	リンクの所有者。
DB_LINK	リンク名。
DBSVR	リモート・データベース情報のあるデータベース・セクション。
USER_NAME	リモート・データベースのユーザー名。

SYSDESCOL

SYSDESCOL 表には、ダイナミックカラムの情報が含まれています。RESOURCE 或いは CONNECT 権限を持つユーザーは、自分に作成された、及び権限が与えられたダイナミックカラムの情報を読むことが出来ます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
COLUMN_NAME	ダイナミックカラムの名称。
TABLE_NAME	JSONCOLS タイプを所有する表の名称。
TABLE_OWNER	表の作成者の名前。
DATA_TYPE	ダイナミックカラムのタイプ。
COLUMN_NUM	ダイナミックカラムの数。
LENGTH	ダイナミックカラムの長さ。

SYSDOMAIN

SYSDOMAIN 表には、データベースに定義されているドメイン情報があります。SYSDAM または RESOURCE、CONNECT 、DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
DOMAIN_NAME	ドメイン名。
DOMAIN_OWNER	ドメインを定義したユーザー名。
ASCII_DEF	ドメインの初期設定値（ASCII 形式）。
TYPE_NAME	ドメインのデータ型。BINARY、CHAR 、NCHAR 、DATE 、DECIMAL 、DOUBLE 、FILE 、FLOAT 、INTEGER 、LONG VARCHAR、NCLOB、LONG VARBINARY、SERIAL 、SMALLINT 、TIME 、TIMESTAMP 、VARCHAR 、NVARCHAR のいずれかです。
DATA_LEN	ドメインのデータ型のサイズ。
PRECISION	ドメインの精度。
SCALE	ドメインのスケール。
CONSTR	ドメイン制約。
TEXT_CONVERTER	テキスト索引と PURETEXT() UDF を作成するため、これは CLOB、NCLOB 、BLOB or FILE データを純テキストに転換します

SYSFILE

SYSFILE 表には、データベースのファイル情報があります。RESOURCE 或いは CONNECT 権限を持つユーザーは当該表の如何なる情報を読み込むことができません。システムはエラー6829 を返して select 権限がないと表示します。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
FILE_NAME	論理ファイル名。
FILE_TYPE	ファイルの種類： 1（データファイル） 2（BLOB ファイル）
FILE_OID	OID のファイル
TS_NAME	ファイルが属する表領域名。
FILE_NPAGES	ファイルのページ数。AUTOEXTEND 表領域では、FILE_NPAGES が物理ファイルのページ数より少なくなるかもしれません。
RAWDEV_OFFSET	現バージョンでは未サポートです。
CREATE_TIME	ファイルを作成する時間

SYSFILEOBJ

SYSFILEOBJ 表には、データベースのファイルオブジェクト情報があります。システムとユーザーのファイルオブジェクト双方が含まれます。RESOURCE 或いは CONNECT 権限を持つユーザーは当該表の如何なる情報を読み込むことができません。システムはエラー6829 を返して select 権限がないと表示します。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
FILE_TYPE	00—システム・ファイルオブジェクト。 01—ユーザー・ファイルオブジェクト。
SHARE	ファイルオブジェクトを共有するレコード数。
FILE_NAME	ファイルオブジェクトの所在を示す絶対パス名。

SYSFOREIGNKEY

SYSFOREIGNKEY 表には、データベースの全ての外部キー情報があります。ユーザーは特定的な表操作権限を持つ、また RESOURCE 或いは CONNECT 権限を持つと、表の外部キーの権限が与えられている情報が読めます、そして作成情報も取れます。SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
FK_TBL_NAME	子表名（外部キーが定義されている表）。
PK_TBL_NAME	外部キーの親表名。
FK_TBL_OWNER	子表の所有者。
PK_TBL_OWNER	親表の所有者。
FK_NAME	外部キー名。
UPD_ACT	更新の参照アクション。 **0** — アクション無し **1** — NULL にセット **2** — カスケード **3** — 初期設定値にセット
DEL_ACT	削除の参照アクション。 **0** — アクション無し **1** — NULL にセット **2** — カスケード **3** — 初期設定値にセット

SYSGLBTRANX

SYSGLBTRANX 表は、グローバル・トランザクション情報をもちます。RESOURCE 或いは CONNECT 権限を持つユーザーは当該表の如何なる情報を読み込むことができません。システムはエラー6829 を返して select 権限がないと表示します。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
STATE	グローバル・トランザクションの状態。 0 (ISSUE) — トランザクション・ブランチが発行されましたが、全ての参加データベースの準備ができていません。 1 (PREPARE) — 参加データベースは準備されましたが、コミットするかアボートするかを判断する親参加データベースを待機しています。 2 (COMMIT) — 参加データベースはグローバル・トランザクションをコミットすることを決定しました。 3 (PEND_TO_COMMIT) — クラッシュ回復後、このトランザクション・ブランチはコミット行列に追加され、コミットされるのを待っています。 4 (PEND_TO_ABORT) — クラッシュ回復後、このトランザクション・ブランチはアボート行列に追加され、アボートされるのを待っています。
PARTICIPANT	グローバル・トランザクションの参加データベース。
GLBTRANXID	グローバル・トランザクション ID。

SYSINDEX

SYSINDEX 表には、データベースの索引情報があります。NUM_PAGE、NUM_LEVEL、NUM_LEAF、DIST_KEY、NUM_PAGE_KEY、CLSTR_COUNT カラムの値が-1 の場合は、これらの値が適用できないことを意味します。ユーザーは特定的な索引の操作権限を持つ、また RESOURCE 或いは CONNECT 権限を持つと、この索引の権限が与えられている情報が読めます、そして索引を作成する情報も取れます。SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
INDEX_NAME	索引名。
TABLE_NAME	索引が定義されている表の名前。
TS_NAME	インデックスの格納に用いられる表領域を指定します。
TABLE_OWNER	索引が定義されている表の所有者。
UNIQUE	索引の一意性フラグ： **0** — 一意ではない **1** — 一意 **3** — 主キー **4** —自動インデックス
NUM_COL	索引キーに含まれるカラムの個数。
INDEX_OID	索引の OID。
NUM_PAGE	索引のページ数。
NUM_LEVEL	レベル数。
NUM_LEAF	リーフページのページ数。
DIST_KEY	異なるキーの個数。
NUM_PAGE_KEY	キー当たりのページ数。

カラム名	解説
CLSTR_COUNT	クラスタ・カウント；索引を使用してデータページをアクセスするときのページ I/O 回数。これはバッファ数に関連します。
CREATE_TIME	索引を作成した時間。

SYSINDEXREF

SYSINDEXREF 表には、データベースのインデックス情報及び自動インデックス情報があります。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
INDEX_NAME	インデックス名。
TABLE_NAME	インデックスの所有する表の名称。
TABLE_OWNER	インデックスの所有する表の所有者。
UNIQUE	インデックスの一意性状態のフラグ。 **0** — 一意ではない **1** — 一意 **3** — 主キー **4** — 自動インデックス
TOTAL_COUNT	使用される自動インデックスの総数。
LASTREF_TIME	自動インデックスの最後のリファレンス時間。

SYSINFO

SYSINFO 表には、データベースの現在の状態に関する情報があります。RESOURCE 或いは CONNECT 権限を持つユーザーは情報が読めます。

SYSINFO.ID	SYSINFO.INFO
0108	DB_PAGE_SIZE
0711	VERSION
0712	FILE_VERSION

SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

SYSINFO 表のスキーマは、他のシステム表と異なります。スキーマは以下のとおりです。

```
SYSTEM.SYSINFO (char(4) ID, varchar(32) INFO, varchar(32) VALUE);
```

各カラムの意味は以下のとおりです。

- ID: アイテムID。システム情報は、このIDに基づいて分類されます。最初の2文字はカテゴリを表し、続く2文字はカテゴリ内のアイテムを表します。例えば、NUM_LOGICAL_READを表すID '0105'の'01'は、ページとI/Oカテゴリに属することを意味し、'05'はページとI/Oカテゴリ内の順序を表します。IDを使うと、SYSINFOのソートや抽出ができます。
- INFO: システム情報のアイテム名。例えば、'NUM_LOGICAL_READ' は、論理ディスク読み込み数を意味します。
- VALUE: システム情報の全ての値は、VARCHARデータで戻されます。

➲ 例

以下の文は、論理ディスク読み込み数を表示します。

```
dmSQL> select ID, INFO, VALUE from SYSTEM.SYSINFO where INFO = 'NUM LOGICAL READ';
ID       INFO                              VALUE
====     ================        ========================================
0105     NUM LOGICAL READ          338

1 rows selected
```

以下のリストは、SYSINFO カタログにある全アイテムです。

ページとI/O情報

SYSINFO.ID	SYSINFO.INFO	解説
0101	NUM_IDX_SPLIT	発生を分割する索引ページ数。
0102	NUM_PAGE_COMPRESS	圧縮されたデータページ数。例、ページ再編成。
0103	NUM_PHYSICAL_READ	物理ディスク読み込みの数。I/O 単位はページ。
0104	NUM_PHYSICAL_WRITE	物理ディスク書き込みの数。I/O 単位はページ。
0105	NUM_LOGICAL_READ	論理読み込みの数。I/O 単位はページ。
0106	NUM_LOGICAL_WRITE	論理書き込みの数。I/O 単位はページ。
0107	NUM_PAGE_BUF	ページ・バッファ数。単位はページ。
0108	DB_PAGE_SIZE	データベースのページサイズ（バイト）
0109	DB_SCA_SIZE	システム制御エリアサイズ (ページ)
0110	NUM_JOURNAL_BUF	ジャーナルバッファ数 (ブロック)
0111	DB_SYSCB_SIZE	内部システム制御ブロックサイズ (バイト)

SYSINFO.ID	SYSINFO.INFO	解説
0112	READ_HIT_RATIO	ページバッファがヒット率を読み込む
0113	WRITE_HIT_RATIO	ページバッファがヒット率を書き出す

注 *1ページサイズがキーワードDB_PGSIZによって決めますが、4K、8K、16K或いは32Kに設定されます。*

ジャーナル情報

SYSINFO.ID	SYSINFO.INFO	解説
0201	NUM_JNL_BLK_READ	ジャーナル・ファイルから読み込まれるジャーナル・ブロックの数。
0202	NUM_JNL_BLK_WRITE	ジャーナル・ファイルに書き込まれるジャーナル・ブロックの数。
0203	NUM_JNL_REC_WRITE	生成されたジャーナル・レコードの数。新規ジャーナル・レコードは、まずジャーナル・バッファに配置されます。
0204	NUM_JNL_FRC_WRITE	ジャーナル強制書き込みの数。この数は、ディスクにフラッシュされる汚れたジャーナル・バッファの I/O 数です。
0205	NUM_JOURNAL_FILE	ジャーナル・ファイルの数。

SYSINFO.ID	SYSINFO.INFO	解説
0206	NUM_JOURNAL_BLOCKS	ファイルにあるジャーナル・ブロックの数。データベースにある合計ジャーナル・ブロック数は、 NUM_JOURNAL_FILE* NUM_JOURNAL_BLOCKS
0207	NUM_JNR_BLOCK_FREE	空きジャーナル・ブロック数。
0208	CURRENT_JOURNAL_FN	現在使用されているジャーナル・ファイルのファイル数。
0209	CURRENT_JOURNAL_BN	ジャーナル・ファイルの現在のブロック数。ジャーナル・ファイルの各ジャーナル・ブロックには、以下で表される一意のアドレスがあります。 CURRENT_JOURNAL_FN と CURRENT_JOURNAL_BN。 ジャーナル・ファイルのブロック数は、0 から数えられます。
0210	JOURNAL_FLUSH_RATE	（DMServer に書き込まれたジャーナルブロックの数/フラッシュの回数）/バッファーにジャーナルファイルの数。(%)

注 *1ブロック = 512バイト。*

トランザクション情報

SYSINFO.ID	SYSINFO.INFO	解説
0301	NUM_STARTED_TRANX	開始したトランザクション数
0302	NUM_COMMITED_TRANX	コミットしたトランザクシ

SYSINFO.ID	SYSINFO.INFO	解説
		ョン数
0303	NUM_ABORTED_TRANX	中止したトランザクション数
0304	NUM_CHECKPOINT	チェックポイントの数
0305	NUM_COMMIT_WAITER	グループ・コミットを待機しているトランザクション数。

ロック情報

SYSINFO.ID	SYSINFO.INFO	解説
0401	NUM_ROW_LOCK_UPG	拡張したページ・ロックの数(ページ・ロックに拡張した行ロック)
0402	NUM_PAGE_LOCK_UPG	拡張した表ロックの数(表ロックに拡張したページ・ロック)
0403	NUM_LOCK_TIMEOUT	タイムアウトのために、ロックできなかった数
0404	NUM_LOCK_WAIT	ロック待機数
0405	NUM_LOCK_REQUEST	ロックが要求された数
0406	NUM_DEADLOCK	検出されたデッドロックの数

接続情報

SYSINFO.ID	SYSINFO.INFO	解説
0501	NUM_MAX_HARD_CONNECT	データベースに認められている最大接続数。(接続のハードの限界。つまり、データベースが新規ジャーナルで起動、或いは新規データベース作成時の **DB_MaxCo)**。
0502	NUM_MAX_SOFT_CONNECT	一時的に認められている接続の最大数(接続のソフトの限界。つまり、ノ

SYSINFO.ID	SYSINFO.INFO	解説
		ーマル起動時の DB_MaxCo)。接続のソフトの限界は、接続のハードの限界以下です。(前バージョンの *NUM_MAX_TRANX*)
0503	NUM_CONNECT	現在のアクティブ接続数(前バージョンの *NUM_ACT_TRANX*)
0504	NUM_PEAK_CONNECT	一時的なアクティブ接続の最大数(アクティブ接続のピーク数)。

データ操作情報

SYSINFO.ID	SYSINFO.INFO	解説
0601	NUM_SQL_SELECT	SELECT 操作の数
0602	NUM_SQL_INSERT	INSERT 操作(INSERT INTO を含む)の数
0603	NUM_SQL_UPDATE	UPDATE 操作の数.
0604	NUM_SQL_DELETE	DELETE 操作の数
0605	NUM_SQL_PREPARE	サーバーに呼び出す SQLPrepare()の数
0606	NUM_SQL_EXECUTE	サーバーに呼び出す SQLExecute()の数
0607	NUM_SQL_EXEDIRECT	サーバーに呼び出す SQLExecDirect()の数
0608	NUM_SQL_FETCH	ネットワークを経由するフェッチしたデータの数

データベース情報

SYSINFO.ID	SYSINFO.INFO	解説
0701	SYSINFO_RESET_TIME	SYSINFO のカウンタが再起動し

SYSINFO.ID	SYSINFO.INFO	解説
		た日時(新) これは、SYSINFO がリセットされた日時を記録するために使用します。この設定は、以下の条件で発生します。 1. dmSQL> set SYSINFO clear; 2. 1 つのカウンタが一杯になり、全カウンタをリセット。以下の時にチェックされます。 2-1. SYSINFO 表を選択するたびにチェック。 2-2. I/O デーモンによって約 5 秒間隔でチェック。
0702	DCCA_SIZE	DCCA の合計サイズ。単位はバイト。
0703	FREE_DCCA_SIZE	使用可能な DCCA のサイズ。単位はバイト。
0704	DDB_MODE	分散型データベース・モード： **ON** —使用可能 **OFF** —使用不可能。
0705	BACKUP_MODE	バックアップ・モード： . **NON-BACKUP** — 非バックアップ・モード(DB_BMode = 0) . **BACKUP-DATA**— データのみバックアップ・モード(DB_BMode = 1) . **BACKUP-DATA-AND-BLOB** — データと BLOB のバックアップ・モード(DB_BMode = 2)
0706	USER_FO_MODE	ユーザーFO モード： **ON** —使用可能

SYSINFO.ID	SYSINFO.INFO	解説
		OFF—使用不可能。
0707	READ_ONLY_MODE	読み込み専用モード： ON —使用可能 OFF—使用不可能。
0708	FRAME_SIZE	BLOB フレーム・サイズ。単位はバイト。
0709	CREATE_DB_TIME	データベースの作成日時。
0710	START_DB_TIME	データベースの起動日時。
0711	VERSION	DBMaster のバージョン。
0712	FILE_VERSION	データベースのファイル・バージョン。
0713	FORCE_NEW_JNL_TIME	新規ジャーナルで強制起動した日時。
0714	START_NO_JNL_TIME	ジャーナルを OFF にした日時。
0715	END_NO_JNL_TIME	ジャーナルを ON にした日時。
0716	MAX_ITT_SIZE	内部一時表に許可できる最大なメモリサイズ。(バイト)
0717	CURRENT_ITT_SIZE	内部一時表に現在のサイズ。(バイト)
0718	FULL_BACKUP_COST	最後の完全バックアップ時間コスト。
0719	DIFF_BACKUP_COST	最後の差分バックアップ時間コスト。
0720	DIFF_BACKUP_PCT	(最後の差分バックアップサイズ)/(データベースサイズ) * 100%

システム情報

SYSINFO.ID	SYSINFO.INFO	解説
0801	CPU_USAGE	短期間の平均 CPU 負荷(約 5 秒)

SYSINFO.ID	SYSINFO.INFO	解説
		(*新*). (**0 ~ 100 %**) サポートのプラットフォーム：Solaris、LINUX 、Windows 2000 (最初の CPU のみカウントし、pdh.dll ライブラリが必要)。 このアイテムを有効にするために、I/O デーモンを起動する必要がありますので、ご注意ください。
0802	TOTAL_MEMORY	合計物理メモリ。単位はバイト。 サポートのプラットフォーム：Solaris、LINUX 、Windows NT/2000、POSIX 標準をサポートしている UNIX。
0803	TOTAL_FREE_MEMORY	現在の空き物理メモリ(新)。単位はバイト。 サポートのプラットフォーム：Solaris、LINUX 、Windows 、POSIX 標準をサポートしている UNIX。
0804	TOTAL_SWAP_SPACE	合計スワップ・スペース。単位はバイト。 サポートのプラットフォーム：Solaris、LINUX 、Windows。
0805	TOTAL_FREE_SWAP_SPACE	現在の空きスワップ領域(*新*)。単位は**バイト**。 サポートのプラットフォーム：Solaris、LINUX、Windows。

サポートされていないバージョンの場合、その値は NULL になります。SYSINFO カタログは、累計されたカウンタの集まりです。SYSINFO をリセットする方法が 2 つあります。

1. SET SYSINFO CLEAR文の実行。

2. 1つのカウンタがオーバーフローした時、SYSINFOにある全カウンタがリセットされます。以下の時に、オーバーフローがチェックします。
   SYSINFO表が選択された時。
   I/Oデーモンによって5秒毎にチェックされます。

以下の文を実行して、リセットした日時を取得することができます。

```
dmSQL> select VALUE from SYSTEM.SYSINFO where INFO = 'SYSINFO_RESET_TIME';
```

SYSJARFILE

SYSJARFILE 表に Java の JAR パッケージの情報があります。RESOURCE または CONNECT 権限を持つユーザーは、自分が作成した、または他のユーザーより対象権限を取得している表またはビューの情報を読み込むことができます。SYSADM または DBA、SYSDBA 権限を持つユーザーはあらゆる情報が読めます。

カラム名	解説
JAR_NAME	JAR 名。
JAR_OWNER	JAR の所有者。
JARFILE_NAME	JAR ファイルの名称。

SYSJAVAARGU

SYSJAVAARGU 表に Java の変数情報があります。RESOURCE または CONNECT 権限を持つユーザーは、自分が作成した、または他のユーザーより対象権限を取得している表またはビューの情報を読み込むことができます。SYSADM または DBA、SYSDBA 権限を持つユーザーはあらゆる情報が読めます。

カラム名	解説
PROC_NAME	プロシージャの名称。
PROC_OWNER	プロシージャの所有者。

カラム名	解説
DATATYPE_NAME	データタイプの名称。
ORDER	JAVA 変数のオーダー。
DATATYPE	データのタイプ。

SYSLOCK

SYSLOCK 表には、データベースオブジェクトのロック情報があります。RESOURCE 或いは CONNECT 権限を持つユーザーは当該表の如何なる情報を読み込むことができません。システムはエラー6829 を返して select 権限がないと表示します。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

注 *ロックレベルはSYSTEM、TABLE、PAGEまたはTUPLEで、ロック状態はGRANTED、WAITINGまたはCONVERTで、ロックモードはNONE、IS、S、IX、SIXまたはXです。*

カラム名	解説
LK_OBJECT_ID	ロックされたオブジェクトの OID。
TABLE_ID	ロックされたオブジェクトがある表の OID。
LK_GRAN	ロック単位。SYSTEM、TABLE、PAGE、TUPLE。
HOLD_LK_CONNECTION	オブジェクトのロックしている接続 ID。
LK_STATUS	ロック状態。GRANTED、WAITING、CONVERT。
LK_CURRENT_MODE	オブジェクトの現在のロックモード。
LK_NEW_MODE	オブジェクトの新しいロックモード。

SYSOPENLINK

SYSOPENLINK 表には、オープン・データベースリンク情報があります。RESOURCE 或いは CONNECT 権限を持つユーザーは自分が作成した情報が読めます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
DB_LINK	オープンしているデータベースリンク名。
DBSVR	データベースサーバー名。
USER_NAME	ユーザー名。
TXN_STATUS	トランザクションの状態。 'R' — 読み込み 'W' — 書き込み 'N' — トランザクション無し

SYSPENDTRANX

SYSPENDTRANX 表には、分散データベース環境でコミットされなかったトランザクション情報があります。RESOURCE 或いは CONNECT 権限を持つユーザーは当該表の如何なる情報を読み込むことができません。システムはエラー6829 を返して select 権限がないと表示します。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
XIDFORMAT	グローバル・コーディネータの種類を意味するフォーマット ID、DBMaster は 22873 です。
PREPAREDTIME	コミット・トランザクションが準備された時刻。
GLBTRANXID	グローバル・トランザクション ID。

SYSPROCINFO

SYSPROCINFO 表には、ストアドプロシージャ情報があります。RESOURCE 或いは CONNECT 権限を持つユーザーは自分が作成した、または他のユーザーより対象権限を取得しているプロシージャ情報が読めます。SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

ローカルの一時プロシージャについては、ユーザは現在の接続で作成されているプロセスのみ確認できます。

グローバルの一時プロシージャについては、あらゆるユーザー（異なる接続も含め）は当該プロシージャのライフサイクル内に当該プロシージャが削除されるまで、その情報を確認することができます。

カラム名	解説
QUALIFIER	修飾名。
PROC_OWNER	プロシージャの所有者。
NAME	プロシージャ名。
NUM_INPUT_PARAMS	入力パラメータの個数。
NUM_OUTPUT_PARAMS	出力パラメータの個数。
NUM_RESULT_SETS	結果セットの数。
REMARKS	注釈。
PROC_TYPE	プロシージャの種類。 1 (SQL_PT_PROCEDURE)—プロシージャ 2 (SQL_PT_FUNCTION)—関数
TEMP_TYPE	一時表領域のタイプ。 0 —標準 1 —グローバル 2 —ローカル
SID	接続 ID。

SYSPROCJAVA

SYSPROCJAVA 表に Java のプロシージャ情報があります。RESOURCE または CONNECT 権限を持つユーザーは、自分が作成した、またはビューの情報及び他のユーザーより対象権限を取得している表またはビューの情報を読み込むことができます。SYSADM または DBA、SYSDBA 権限を持つユーザーはあらゆる情報が読めます。

カラム名	解説
PROC_NAME	プロシージャ名。
PROC_OWNER	プロシージャの所有者。
CLASS_ID	Class ID。
NUM_ARGU	変数ナンバー。
NUM_RELATED_JARFILE	関連する JAR ファイルの数。
JAR_NAME	JAVA プロシージャの名称。
JAR_OWNER	JAVA プロシージャの所有者。

SYSPROCPARAM

SYSPROCPARAM 表には、ストアドプロシージャのパラメータ情報があります。RESOURCE または CONNECT 権限を持つユーザーは、自分が作成した、または他のユーザーより対象権限を取得しているプロシージャ関数の情報を読み込むことができます。SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

ローカルの一時プロシージャについては、ユーザは現在の接続で作成されているプロセスのみ確認できます。

グローバルの一時プロシージャについては、あらゆるユーザー（異なる接続も含め）は当該プロシージャのライフサイクル内に当該プロシージャが削除されるまで、その情報を確認することができます。

カラム名	解説
QUALIFIER	修飾名。
OWNER	プロシージャの所有者。
PROC_NAME	プロシージャ名。
PARAM_NAME	パラメータ名。
PARAM_TYPE	パラメータの種類。 1 (SQL_PARAM_INPUT) — 入力 3 (SQL_PARAM_OUTPUT) — 出力 4 (SQL_RETURN_VALUE) — 戻り値 5 (SQL_RESULT_COL) — 結果セット
DATA_TYPE	データ型。
TYPE_NAME	タイプ名。
PRECISION	精度。
LENGTH	サイズ。
SCALE	スケール。
RADIX	基数。
NULLABLE	Null 値の可否。 1 — Null 値が認められる 0 — Null 値は認められない
REMARKS	注釈。
SID	接続 ID。

SYSPROJECT

SYSPROJECT 表には、ESQL プロジェクト情報があります。RESOURCE または CONNECT 権限を持つユーザーは、自分が作成したプロシージャの ESQL プロジェクトの情報を読み込むことができます。SYSADM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
PROJECT_NAME	プロジェクト名。
PROJECT_OWNER	プロジェクト所有者。
MODULE_NAME	モジュール名。
MODULE_OWNER	モジュールの所有者。
MODULE_SOURCE	モジュールソース。
REF_CMD	内部用

SYSPUBLISH

SYSPUBLISH 表には、表レプリケーションのソース情報があります。RESOURCE 及び CONNECT 権限を持つユーザーはユーザー表のレプリケーション情報を読み込むことができます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
REPLICATION_NAME	レプリケーション名。
TYPE	S — 同期 A — 非同期
TABLE_OWNER	レプリケートされる表の所有者。
TABLE_NAME	レプリケートされる表名。
NUM_PROJECT	プロジェクション・カラムの数。
FRAGMENT	フラグメント文字列。
NUM_SUBSCRIBER	サブスクライバの数。

SYSSCHEDULE

SYSSCHEDULE 表にスケジュールタスクの情報があります。データベースが作成された後、システムは自動的にスケジュール **schelogcl** に関するレコ

ードを毎日の午前 1:10 にタスク **tasklogcl** を実行する SYSSCHEDULE に追加します。このスケジュールはディフォルトとして、無効になっています。DBA 以上の権限を持つユーザーのみ、SCHEDULE_ENABLE または SCHEDULE_DISABLE を使用して、当該スケジュールを有効/無効にすることができます。

カラム名	解説
SCHEDULE_OWNER	スケジュールの所有者。
SCHEDULE_NAME	スケジュール名。
TASK_NAME	スケジュールに名づけされたタスクの名称。
TIMETABLE	スケジュールと前のスケジュールとの間隔。
START_TIME	スケジュールの実行開始時刻。
END_TIME	スケジュールの実行完了時刻。
STATUS	スケジュールの状態。 **0** —無効 **1** — 有効

SYSSCHELOG

SYSSCHELOG 表にスケジュールタスクのレコードを実行する情報があります。定期的に当該表に格納されている過大のログをクリアし、最近の数日のログを保存することをお勧めします。

カラム名	解説
SCHEDULE_OWNER	スケジュールの所有者。
SCHEDULE_NAME	スケジュールタスクの名称。
PROCESS_CONNECTION_ID	タスクの接続 ID。

カラム名	解説
BEGIN_TIME	タスクの実行開始時刻。
END_TIME	タスクの実行完了時刻。
STATUS	タスクの実行状態。
ERROR_MSG	タスクが実行失敗時に出力したエラーメッセージ。

SYSSCHEMA

SYSSCHEMA 表にはスキーマ名とスキーマ所有者との関係情報があります。RESOURCE及びCONNECT権限を持つユーザーはユーザースキーマ情報が読めます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
SCHEMA_NAME	スキーマの名。
SCHEMA_OWNER	スキーマの所有者。

SYSSUBSCRIBE

SYSSUBSCRIBE 表には、表レプリケーションのターゲット情報があります。RESOURCE 及び CONNECT 権限を持つユーザーは、ユーザー表及び関連する表の情報が読みこめます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
BASE_TABLE_OWNER	ベース表の所有者。
BASE_TABLE_NAME	ベース表名。
REPLICATION_NAME	レプリケーション名。

カラム名	解説
DB_LINK	データベースリンク名。
TABLE_OWNER	表の所有者。
TABLE_NAME	表名。

SYSSYNONYM

SYSSYNONYM 表には、データベースに定義されているシノニムの情報があります。RESOURCE 及び CONNECT 権限を持つユーザーは、自分が作成したビューまたは表のシノニム情報が読めます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
SNAME	シノニム名。
OWNER	シノニムの所有者。
TV_NAME	シノニムのソース表名/ビュー名。
TV_OWNER	表/ビューの所有者。
TV_LINK	リモート・データベースにある表又はビューのリンク名
TV_SERVER	リモート・データベースにある表又はビューのデータベース名

SYSTABLE

SYSTABLE 表には、データベースの全ての表情報があります。RESOURCE 及び CONNECT 権限を持つユーザーは、自分が作成したまたは他のユーザーから対象権限を取得しているビューまたは表の情報が読めます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説

カラム名	解説
TABLE_NAME	表名。
TABLE_OWNER	表の所有者。
TABLE_TYPE	表タイプ：SYSTEM TABLE、SYSTEM VIEW、TABLE、VIEW。
LOCKMODE	表のロックモード： **T** — 表ロック **P** — ページロック **R** — 行ロック 初期設定ロックモードは行ロックです。
CACHEMODE	表全検索のキャッシュモード： **T** — キャッシュする（真）。 **F** — キャッシュしない（偽）。
TS_NAME	表が格納される表領域名。
TABLE_OID	表の OID。
NUM_COL	表内のカラム数。
NUM_INDEX	表の索引数。
NUM_PAGE	表のページ数。初期値は -1 です。表の統計を更新すると、NUM_PAGE の真の値が返ります。
NUM_FRAME	表内の BLOB フレーム数。
NUM_ROW	表の行数。初期値は-1 です。表の統計を更新すると、NUM_ROW の真の値が返ります。
NUM_INDIRECT_ROW	間接行の数。
AVERAGE_LENGTH	表データの平均桁数。初期値は-1 です。表の統計を更新すると、AVERAGE_LENGTH の真の値が返されます。
CREATE_TIME	表を作成した時刻。
UPD_STS_TIME	表の統計を更新した最後の時刻。

カラム名	解説
CONSTR	表制約。
FILLFACTOR	fillfactor は使用済みページの上限パーセントを指定します。上限を超すと新規データの挿入を停止します。ページの空き部分はデータ更新のために使用されます。初期値は 100（%）です。
SERIAL_COL_ID	表の n 番目はシリアル番号カラムです。
SERIAL_START_NO	シリアル番号の開始番号。初期値は 1 です。
REMARKS	表の説明。
NUM_TRIG	表にあるトリガーの個数。
NUM_TEXTINDEX	表にあるテキスト索引の個数。
NUM_PUBLICATION	表のパブリケーション数。
NUM_DEST	非同期レプリケーションのターゲット・データベースの個数。
UPD_STS_MODE	表の更新統計モード。
UPD_STS_SAMPLE	表の更新統計サンプル率。

SYSTABLESPACE

SYSTABLESPACE 表には、データベースにある全ての表領域の情報があります。CONNECT 権限を持つユーザーは当該表の如何なる情報を読み込むことができません。システムはエラー6829 を返して select 権限がないと表示します。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
TS_NAME	表領域名。

TS_TYPE	表領域のタイプ： テーブルスペースのタイプ **0** (fixed(W)) — 標準 **1** (ext(W)) — 自動拡張 **2** (fixed(R)) — 読み取り専用 (標準) 3 **(ext(R)) — 読み取り専用 (自動拡張)**
NUM_OID	表領域の ID。
NUM_FILES	表領域にあるファイルの数。
NUM_PAGES	表領域のページ数。表領域が自動拡張のときは、NUM_PAGES の値が表領域の実際のページ数よりも小さいこともあります。
NUM_FREE_PAGES	表領域に残されている使用可能ページ数。
NUM_PE	PE 数。
NUM_FRAMES	表領域にある BLOB のフレーム数。
NUM_FREE_FRAMES	表領域の利用できる空き BLOB フレーム数。
CREATE_TIME	表領域の作成時刻。
NUM_FREE_FRAMES	表領域にある使用可能の空きフレームの数。

SYSTASK

SYSTASK 表にスケジュールのアクションの情報を含めます。データベースが作成された後、システムは自動的にタスク **tasklogcl** に関するレコードを SYSTASK に追加します。このタスクは SCHELOG_CLEAN をコールすることによって、ディフォルトとして作成時間が最新のログの作成時間より早い 20 日のログをクリアします。DBA 以上の権限を持つユーザーのみ、クリアしようとするロゴの作成時間と最新のスケジュールログの作成時間の間の日数（即ち reserve_day の値）を変更することができます。

カラム名	解説
TASK_OWNER	タスクの所有者。

TASK_NAME	タスク名。
TASK_TYPE	タスクのタイプ。
ACTIONS	タスクのアクション。

SYSTEXTINDEX

SYSTEXTINDEX 表には、テキスト索引情報があります。RESOURCE 及び CONNECT 権限を持つユーザーは、自分が作成したまたは他のユーザーから対象権限を取得しているテキスト索引の情報が読めます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
TEXTINDEX_NAME	テキスト索引名。
TABLE_NAME	表の名前。
TABLE_OWNER	表の所有者。
INDEX_OID	索引の OID。
TYPE	2 つのタイプのテキスト索引：シグネチャテキスト索引と IVF テキスト索引。
FACTOR1	内部管理データ。
FACTOR2	内部管理データ。
FACTOR3	内部管理データ。
FACTOR4	内部管理データ。
FACTOR5	内部管理データ。
FACTOR6	内部管理データ。
FACTOR7	内部管理データ。
FACTOR8	内部管理データ。
FACTOR9	内部管理データ。
FACTOR10	内部管理データ。
CREATE_TIME	テキスト索引作成日時。

カラム名	解説
REBUILD_TIME	テキスト索引再作成日時。

SYSTRIGGER

SYSTRIGGER表には、トリガー情報があります。RESOURCE及びCONNECT権限を持つユーザーは、自分が作成した表のトリガー情報が読めます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
TBNAME	表名。
TBOWNER	表所有者。
TRIGNAME	トリガー名。
TRIGEVENT	トリガーイベント： 1 — 挿入イベント 2 — 削除イベント 3 — 更新イベント 4 — カラム更新イベント
NUM_COL	カラム数。
SCOL_NUM	トリがーで更新された最も低いカラム番号
TRIGTYPE	トリガータイプ： 1 — BEFORE と FOR EACH STATEMENT 2 — BEFORE と FOR EACH ROW 4 — AFTER と FOR EACH STATEMENT 8 — AFTER と FOR EACH ROW
STATUS	状態： 1 —イネーブル 2 —ディセーブル

カラム名	解説
OLD	古い値。
NEW	新しい値。
MODE	1—有効トリガー 0— 無効トリがー
TRIGDEF	トリガーの定義。

SYSTRPDEST

SYSTRPDEST 表には、非同期表レプリケーションに使用されるスケジュール情報があります。RESOURCE 権限を持つユーザーは非同期表レプリケーションを通じてスケジュールに情報をとる必要があります。SYSADM または RESOURCE、CONNECT、DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
SVRNAME	リモートデータベース名。
USER_NAME	リモートデータベースのユーザーアカウント。
STATUS	リモートデータベースの状態 0— 正常 1—エラー 2—再試行 3—中断
BEGTIME	レプリケーションの開始日時
INTERVAL	レプリケーションの間隔

SYSTRPJOB

SYSTRPJOB 表には、非同期表レプリケーションに使用されるログ情報があります。RESOURCE 及び CONNECT 権限を持つユーザーは、当該表の如何

なる情報を読み込むことができません。システムはエラー6829 を返して select 権限がないと表示します。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
DESTINATION	データがレプリケートされるデータベース。
FN	トランザクションのログレコードのファイル番号。
OFFSET	トランザクションのログレコードのオフセット。

SYSTRPPOS

SYSTRPPOS 表には、非同期表レプリケーションに使用される情報があります。RESOURCE 及び CONNECT 権限を持つユーザーは、当該表の如何なる情報を読み込むことができません。システムはエラー6829 を返して select 権限がないと表示します。SYSDAM または DBA、SYSYDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
RECORD_ID	内部用。
POSARRAY	内部用。

SYSUSER

SYSUSER 表には、現在データベースに接続している全てのユーザーの状態に関する情報があります。RESOURCE 及び CONNECT 権限を持つユーザーは自分の情報が読めます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。データベースへの接続を切断する前に、SYSUSER 表に切断しよう接続の ID をクエリすべきです。ログインホスト名がネットワークに登録されていない場合、LOGIN_HOST は **anonymous** になります。更新統計状態を監視する場合、SYSUSER 表の SQL_CMD カラムをクエリすべきです。

カラム名	解説
CONNECTION_ID	接続 ID。
USER_NAME	ログイン・ユーザー名。
LOGIN_TIME	ログインした時刻。
LOGIN_IP_ADDR	ログイン IP アドレス。
LOGIN_HOST	ログイン・ホスト名。
NUM_SCAN	select オペレーション数。
NUM_INSERT	insert オペレーション数。
NUM_UPDATE	update オペレーション数。
NUM_DELETE	delete オペレーション数。
NUM_TRANX	処理したトランザクション数。
NUM_JBYTE_PER_TRAN	トランザクション当たりのジャーナルバイト数。
FIRST_W_JNL_FN	アクティブ・トランザクションの最初のジャーナルファイル番号
FIRST_W_JNL_BN	アクティブ・トランザクションの最初のジャーナル・ブロック番号
NUM_BYTE_JNR_DATA	アクティブ・トランザクションで使用される総ジャーナル・バイト
NUM_J_BLOCK_DURATN	アクティブ・トランザクションで使用される最初のジャーナル・ブロックと最新のものとの間隔

SQL_CMD	実行した最新の SQL 文とその状態。状態は以下のようになります。 **[PRE]** — SQL 文を準備中。 **[EXEC]** — SQL 文は、SQLExecute コールから実行中。 **[EXDIR]** — SQL 文は、SQLExecDirect コールから実行中。 **[FETCH]** — その操作は、データのフェッチ段階にあります。 **[EXIT]** — SQL 文は、準備、実行、フェッチ操作を終了しました。
TIME_OF_SQL_CMD	SQL 文が実行された最新の時刻。
AFFINITY_MASK	CPU アフィニティー。
PRIORITY_LEVEL	接続の優先級。
CPU_USAGE	CPU の使用率。

SYSUSERFUNC

SYSUSERFUNC 表には、ユーザー定義関数と組み込み関数の情報があります。UDF 関数を実行する場合、SYSUSERFUNC でのデータを使用する必要がありますので、CONNECT 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
MODE	関数のタイプ： **1** — 組み込み関数 **0** — 非組み込み関数
FILE_NAME	組み込み関数があるファイルのファイル名。
FUNC_NAME	組み込み関数名。

カラム名	解説
LANGUAGE_TYPE	UDF のタイプ： **0** —C UDF **1** —LUAUDF
TIME_STAMP	UDF が作成される時間。
RETURN_TYPE	組み込み関数が返す値のデータ型。
NUM_OF_PARAMETER	関数にあるパラメータの個数。
PARAMETER	各パラメータのデータ型。パラメータの個数は NUM_OF_PARAMETER の値で与えられます。

SYSVIEWDATA

SYSVIEWDATA 表には、データベースのビュー情報があります。RESOURCE または CONNECT 権限を持つユーザーは、自分が作成した、または他のユーザーより対象権限を取得しているビューの情報を読み込むことができます。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
VIEW_NAME	ビュー名。
VIEW_OWNER	ビューの所有者。
STATUS	**0** — 無効なビュー **1** — 有効なビュー
VIEW_DEFINITION	ビューの定義。

SYSWAIT

SYSWAIT 表には、同じオブジェクトの別のロックが解除されるのを待っているロック状態を知らせます。RESOURCE 或いは CONNECT 権限を持つユーザーは当該表の如何なる情報を読み込むことができません。システムは

エラー6829 を返して select 権限がないと表示します。SYSDAM または DBA、SYSDBA 権限を持つユーザーは全部の情報が読めます。

カラム名	解説
WAITING_CONNECTION	ロック解除を待機している接続 ID。
WAITED_CONNECTION	ロック解除を待機させている接続 ID。

C. システムの制限

DBMaster のデータベースには、使用するオブジェクト名、索引、表、メモリバッファ等のサイズ、ファイル数、同時実行トランザクション数等に制限があります。これらの制限を以下に要約します。

C.1 名前の制限

SQL 言語の ANSI/ISO 規格は、一意的な名前によってデータベース・オブジェクトを識別することを規定し、名前を付けるデータベース・オブジェクトを定義しています。これらの名前は SQL 文を実行する際に、どのオブジェクトを使用するのかを識別するために使用されます。名前を必要とするオブジェクトには次のものがあります。

- 表
- カラム
- ユーザー

ANSI/ISO 規格の SQL データベース・オブジェクトの名前は、ユーザーのパスワードを除き、128 文字以下の英数字からなります。ANSI/ISO 規格では、SQL データベースのオブジェクトの最大長は 128 文字で、英数字のみ含むことができます。スペースや句読点の文字を含めることはできません。

DBMaster は、名前に使用できる文字の範囲を広げ、更に幾つかのデータベース・オブジェクトにも名前をサポートしています。名前を付けることができるその他のデータベース・オブジェクトには次のものがあります。

- 索引

- カーソル
- 表領域
- 主キー/外部キー

DBMaster のデータベースの名前は、1～128 文字の英数字、アンダースコア(_)文字です。アンダースコア文字は、先頭文字を含めて何処にあってもかまいません。

ユーザーのパスワードを除く全てのオブジェクト名は、1～128 文字の英数字、2 バイト文字（日本語）、スペース、アンダースコア文字、記号 $ と # です。先頭文字も含め文字の順序に制限はありません。名前にスペースをいれるときは、ダブルクオート（" "）で囲います。接尾のスペースは無視されます。ユーザーのパスワードもこの規則に従いますが、最大 128 文字に制限されます。

DBMaster がサポートするデータベース・オブジェクト名のサイズ制限を以下の表に示します。

項目	最小	最大
データベース名	1	128
表領域名	1	128
表名	1	128
カラム名	1	128
索引名	1	128
カーソル名	1	128
ユーザー名	1	128
パスワード	1[1]	16
物理ファイル名（パスを含む）	1	256[2]

[1] ユーザーがパスワードを設定しないときは、パスワードの長さは NULL です。
[2] NULL 端末を含みます。

項目	最小	最大
論理ファイル名	1	128
SQL 文	—	2097152

表 C-1： データベース・オブジェクト名の長さの最小/最大 (文字)

3.2 ストレージの制限

下記の表は、DBMaster データベース・オブジェクトのストレージ制限を示しています。全ての項目に最大値が示されていますが、これは論理的な制限と理解すべきです。項目には、物理システムの制限（システムのメモリ、ディスク領域等）とオペレーティング・システムの制限（システムリソースや他の制限）も課せられているを忘れないでください。全ての制限は、特に注意しない限り、DBMaster がサポートする全てのプラットフォームで共通です。

項目	最小	最大
データベースのサイズ	—	256PB
データベースのファイル数	1	32767
データベースの表領域数	1	32767
表領域のファイル数	1	32767
表領域の表数	0	無制限[3]

[3] 表と索引の個数は、現在オペレーティング・システムのみによって制限されます。

項目	最小	最大
データファイルのページ数	3	2^{31}-1
表のカラム数	1	2000 (表の最大カラム数がページサイズに依存)
表のレコード（行）サイズ	0	968 (4KB ページサイズ)[4]8064 (8KB ページサイズ)[4]16256 (16KB ページサイズ)[4] 32640 (32KB ページサイズ)[4]
表の索引数	0	無制限[3]
索引のカラム数	1	32
索引のキーの長さ	0	4000
索引のカラム ID	1	無制限[4]
システム一時ファイル数	1	8
ジャーナルファイル数	1	8
ジャーナルファイルのページ数	100 ページ	8 GB
プロジェクション・カラム数	1	表カラム最大数と同じ
GROUP BY カラム数	1	表カラム最大数と同じ
ORDER BY カラム数	1	カラム最大数と同じ
ODBC バインドパラメータ数	0	表カラム最大数と同じ
SQL ソースの個数	1	127

[4]表のすべてのカラムに索引を作成できます。

項目	最小	最大
BLOB ファイルのバイト数	0	8TB
データバッファのページ数	15	OS に依存
ジャーナルバッファのページ数	16	OS に依存
スキャンの述語の操作回数	1	式に依存、最大 1022
式または述語の定数データ	0	64K
式または述語のホストデータ	0	64K

表 C-2： データベース・オブジェクトの最小と最大サイズ（バイト）

3.3 処理上の制限

次の表は、DBMaster データベースの処理上の制限を示しています。

項目	最小	最大
データベースの同時実行トランザクション（接続）の個数	0	4800
CHAR、BINARY データ項目の長さ	0	3968 (4KB ページサイズ)[4] 8064 (8KB ページサイズ)[4] 16256 (16KB ページサイズ)[4] 32640 (32KB ページサイズ)[4]
VARCHAR データ項目の長さ	0	3968 (4KB ページサイズ)[4] 8064 (8KB ページサイズ)[4] 16256 (16KB ページサイズ)[4] 32640 (32KB ページサイズ)[4]

LONG VARCHAR、LONG VARBINARY データ項目の長さ	0	8TB
選択されたコマンドの射影リストにある項目の個数	1	表カラム最大数と同じ

表 C-3：処理上の最小と最大サイズ