

Firebird RDBMS 傾向と対策 2

林 務 (Firebird 日本ユーザー会)

はじめに

本書は、PS ネットワークより平成 15 年 5 月 5 日に初版が出版された『Pocket Tech Note Firebird RDBMS 傾向と対策 2』の原稿を基にして PDF 化したものです。出版の際には B6 サイズの横長の体裁でしたが、PDF 化するにあたって、A4 縦のサイズへと変更し、必要最低限の編集を加えています。

本書の内容は、Firebird の 1.0.2 がリリースされた当初のものですが、基本的な SQL の構文やデータ型に関する知識は、いまでも大きく変わっていません。管理ツールとしてご紹介した IBOConsole や Marathon はその後開発が中断しており、現在では Firebird Project 公式の管理ツールとして FlabeRobin が開発・公開されています。Firebird 日本ユーザー会の加藤大受氏が日本語を行っており、Firebird 日本ユーザー会のページからもダウンロードすることが出来ますので、今後はこちらをお使い下さい。

2007 年 11 月に版元であった PS ネットワークの武田浩一氏が若くして急逝され、PS ネットワークが廃業となったため、多少でも Firebird の普及のお役に立てればとの想いから、ここに執筆者としての責任においてこれを PDF 化してお配りすることにいたしました。Delphi マガジンや nifty serve の FDELPHI でご活躍された武田浩一さんが、Firebird の普及にも尽力されていたことに感謝するとともに、ご冥福をお祈り申し上げます。(2008 年 12 月 11 日ウルトラの街にて 林 務)

まえがき

前著「Firebird RDBMS 傾向と対策 1」に引き続いて、オープンソースの RDBMS、Firebird の SQL に関する解説をさせていただきます。前著では Firebird の概要とインストール方法、オープンソースの GUI 管理コンソールである IBOConsole を使用した管理方法、そして同じくオープンソースの GUI 管理コンソールである Marathon を使用したデータベースの作成方法をご紹介させていただきました。また、Delphi + dbExpress + Firebird による簡易見積システムの作成を通して Delphi から Firebird を利用したアプリケーションの作成を行う手順をご紹介させていただきました。

本書では、Firebird の SQL の解説を詳細に行っていきたいと思います。Firebird は、ポーランド社の InterBase から派生したオープンソース・プロジェクトなので、InterBase の SQL とほぼ 100%の交換性を保っていますが、独自の拡張も行われています。本書は、Firebird を利用する上でのマニュアルとして利用できるよう、SQL の標準的な仕様を逐条的に取り上げ、Firebird 独自の拡張についてわかりやすく囲みにて強調するようにしました。また、出来るだけ具体的な例をあげてのサンプルを提示しました。読者のみなさんが、業務において Firebird を利用される際の、一助となれば幸いです。

なお、当初予定していたストアド・プロシージャとトリガー、UDF については、分量の関係から別稿とすることになりました。本書では、DDL(Data Definition Language)、DML(Data Manipulation Language)、内部関数とその他の SQL コマンドについて解説しています。また、Firebird のトランザクションを実例を挙げて解説することを試みました。

目次

はじめに	1
まえがき	1
<i>Yet another Open Source Database</i>	1
FIREBIRD RDBMS 傾向と対策 2	1
FIREBIRD の SQL	4
ISQL コマンドの使用法	5
ISQL の対話型モード	7
<i>isql SHOW</i> コマンド	7
<i>isql SET</i> コマンド	11
<i>isql</i> その他のコマンド	15
FIREBIRD のオブジェクト命名規則	16
FIREBIRD のセキュリティ	17
GRANT 文	17
REVOKE 文	18
FIREBIRD のトランザクション	19
FIREBIRD のトランザクションコマンド	20
SET TRANSACTION	20
COMMIT	20
ROLLBACK	20
FIREBIRD のトランザクション実行例	21
隔離レベル READ COMMITTED	21
隔離レベル SNAPSHOT	22
隔離レベル SNAPSHOT TABLE STABILITY	24
FIREBIRD SQL – CREATE 系コマンド	26
CREATE DATABASE	26
CREATE DOMAIN	28
CREATE GENERATOR	30
DROP GENERATOR	31
GEN_ID()	32
SET GENERATOR	33
CREATE TABLE	34
RECREATE TABLE	37
CREATE VIEW	38
CREATE EXCEPTION	41
CREATE INDEX	42
CREATE ROLE	43
CREATE SHADOW	44
FIREBIRD SQL – ALTER 系コマンド	45
ALTER DATABASE	45
ALTER DOMAIN	45
ALTER TABLE	46
ALTER EXCEPTION	48
ALTER INDEX	48
FIREBIRD SQL – DML 系コマンド INSERT、UPDATE、SELECT、DELETE 等	50
INSERT 文	50
UPDATE	51
SELECT	51
DELETE	55
FIREBIRD SQL – 内部関数、その他の構文	57
内部関数	57
AVG() 関数	57
COUNT 関数	57
MAX 関数	57
MIN 関数	58
CAST 関数	58
EXTRACT 関数	58

<i>UPPER</i> 関数	59
<i>SUBSTRING</i> 関数	59
その他の構文	59
<i>CONNECT</i> 文	59
<i>SET STATISTICS</i> 文	59
FIREBIRD SQL – DROP 文系	61
DROP DATABASE	61
DROP DOMIN	61
DROP EXCETION	61
DROP INDEX	61
DROP ROLE	61
DROP SHADOW	61
DROP TABLE	61
DROP VIEW	61
APPENDIX A FIREBIRD のデータ型	62
数値データ型	62
整数データ型	62
浮動小数点データ型	62
固定小数点データ型	62
数値データ型の演算上の注意点	62
文字データ型	63
日付時刻型	64
BLOB データ型	65
APPENDIX B ALTER TABLE 文で有効に変換可能なデータ型一覧表	66
APPENDIX C FIREBIRD のキャラクタセットとコレーションオーダー	67

Firebird の SQL

Firebird の SQL は、ANSI/ISO で標準化されている標準 SQL に準拠しています。Firebird は SQL-89 と SQL-92 のほとんど全てと、SQL-99 の機能のいくつかを実行可能です。

特に Firebird は、SELECT ステートメントにおいて内部結合 (JOIN) と外部結合 (OUTER JOIN)、左右の外部結合 (LEFT、RIGHT) すべてをサポートし、ビュー内の UNION を含む UNION をサポートしています。また、GROUP BY 句と ORDER BY 句をサポートし、単純なビューのダイレクトな更新と、その他のビューのトリガーによる更新をサポートしています。

サブクエリーは、WHERE 句や IN 演算子等において利用可能ですが、FROM 句では利用することができません。代わりに、結果セットを返すストアードプロシージャを FROM 句で利用することが可能です。

通常、商用の RDBMS で可能なことはほとんど実行可能なので、商用システムからの移行が容易になっています。

Firebird では、通常の SQL とは別に、埋め込み SQL と動的 SQL (DSQL)、およびプロシージャ/トリガー言語の 4 つの SQL 文法が存在しますが、本書では埋め込み SQL と動的 SQL (DSQL) については触れないことにします。これらは、Firebird に付属の gpre.exe というプリ・コンパイラを利用したプログラミング等で使用するもので、本書が主に対象とする Delphi や VB のプログラマなどが使用するものではないからです。

こうしたマニュアルではアルファベット順にコマンドの解説を行うのが一般的ですが、本書では、利用頻度の高い順に DDL 文から DML 文へと解説を進めることにします。また、ストアードプロシージャとトリガーは分量の関係上、別稿にて解説することになりました、申し訳ありませんが、そちらを参照してください。

なお、SQL 文の動作確認には firebird-win32 1.0.2-Release (Firebird-1.0.2.908-Win32.exe) を Windows2000 にインストールして利用し、標準で付属する isql.exe によって行いました。Firebird プロジェクトのホームページからダウンロードして利用してください。

筆者が日本語化を行っているオープンソースの GUI 管理コンソール IBCConsole や IBCOnsole の WISQL は、isql.exe と若干動作が異なる場合がありますので、注意してください。もちろん、Firebird 自体の動作が変わるわけではありません。

Firebird プロジェクトのホームページ

<http://firebird.sourceforge.net/>

※表記方法に関して

{ } : カッコで囲まれた要素から一つを選択します

[] : カッコで囲まれた要素を使用する事が出来ます

| : 複数の要素の区切として使用しています。

<> : 詳細な情報を別途表記しています。

!!FB!! : Firebird 独自の拡張を解説しています。

isql コマンドの使用手法

まずはじめに、SQL 文を実行するために使用する isql.exe の使用手法について解説します。isql のインストール位置は、<Firebird インストールフォルダ>\bin になります。

isql には対話型モードと非対話型モードがあります。対話型モードは、データベースへの操作を isql のプロンプト上でを行い、結果が画面上に表示されます。非対話型モードでは、事前に用意した SQL スクリプトを使用してデータベースに対する操作を行ったり、データベースの DDL (データ定義言語) を表示・保存させたりします。

まずは、コマンドプロンプト上で USAGE を表示させてみましょう。(適当に改行を入れてあります)

```
C:\Firebird\bin>isql.exe -?  
Unknown switch: ?  
isql [<database>] [-e] [-t <terminator>]  
[-I <inputfile>]  
[-o <outputfile>]  
[-x|-a] [-d <target db>]  
[-u <user>]  
[-p <password>] [-page <pagelength>] [-n]  
[-m] [-q] [-s <sql_dialect>] [-r <rolename>]  
[-c <num cache buffers>] [-z]  
-nowarnings -noautocommit
```

指定したオプション-?は理解されないの、結果的には USAGE を表示することになりますが、USAGE を表示するオプションがないというのはいちよといただけません。

各オプションの意味は以下のとおりです。

<database>

有効なデータベースファイル名を指定します。リモートサーバーのデータベースを指定する場合は以下のように記述します。

Win32 環境

[サーバー名[/ポート番号]:] ドライブ名\パス名
例.

SVR01/3050: C:\DB\SAMPLE.GDB

Linux・UNIX 環境

[サーバー名[/ポート番号]:]/パス名
例.

SVR02/3050: /HOME/DB/SAMPLE.GDB

-e

対話型モードにおいて、ユーザーの入力した SQL 文をエコー表示するためのオプションです。あまり利用価値はないと思います。

-t <terminator>

isql における SQL 文の区切は、標準で「;」(セミコロン) になりますが、このオプションによって別の文字列に変更することが可能です。試した限りでは、2 バイト文字も使用できますが、通常は「!!」や「##」の使用を想定しています。

terminator は、トリガー・プロシージャ言語において文の区切が「;」なので、SET TERM 文 (後述) を使用して変更し、最後に元に戻すというのが定石となっています。このオプションで、別の文字にしておけばそうしたテクニックは必要ないともいえますが、かえってわかりにくいのであまり使うことはないでしょう。

-I <inputfile>

-I は-input と記述することも出来ます。inputfile には、SQL 文を記述したテキストファイルを指定します。

input ファイル中に、isql コマンドの Input を記述して別ファイルを読み出すことも可能です。

isql は最初のファイルの終わりまで処理を行うと、コミットして終了します。

-o <outputfile>

-o は-output と記述することも出来ます。

outputfile で指定されたファイルにコマンドの出力結果を書き出します。

-x | -a

-x は-ex[tract]と記述することも出来ます。

指定したデータベースに対して DDL 文を抽出して出力します。DDL 文のうち、CREATE DATABASE 文はコメントアウトされた形式で出力されます。

InterBase のドキュメントでは、これら-a は全てのデータベースオブジェクトを抽出し、-x はそうではないと記述し、2 つの間に違いがあるようになっていますが、Firebird のサンプルデータベースである Employee.gdb に対して両者を実行してみたところ、全く相違がありませんでした。

オプションの表示においても、[-x|-a]となっているので、この 2 つは等価であると考えてよいでしょう。

-d <target db>

[-x|-a]オプションによって、DDL 文を抽出する際に、CREATE DATABASE 文をコメントアウトせずに、データベース名を<target db>に置き換えて出力します。あまり使用することはないと思います。

-u <user>
データベースへの接続に使用するユーザー名を指定します。-p オプションと共に使用します。

-p <password>
データベースへの接続に使用するパスワードを指定します。-u オプションと共に使用します。

-page <pagelength>
-page は-pag[e]lengthと記述することも出来ます。
<pagelength>で指定された行数ごとに列のヘッダーを表示します。このオプションを指定しない場合の、デフォルトは20行です。

-n
-n は-noauto と記述することも出来ます。
isql では、デフォルトで DDL 文は個別のトランザクションによって自動的にコミットされます。-n オプションを使用すると、この自動コミットがオフになるので、手動で制御たい場合に使用してください。

-m
-m は-merge_stderr と記述することも出来ます。
シェルスクリプトやバッチファイルから isql を実行する際に、エラー出力を標準出力にマージします。結果を一つのファイルにまとめたい時に使用します。

-q
このオプションに関する記述はどこにもありません。ib-support メーリングリストで尋ねたところ、Quiet モードではないかという指摘をいただきました。-q を指定して isql を実行した場合と指定しない場合で以下のような変化があります。

```
C:\>isql
Use CONNECT or CREATE DATABASE to specify a database
SQL> quit;
```

```
C:\>isql -q
SQL> quit;
```

-s <sql_dialect>
-s は-sqldialect と記述することも出来ます。
<sql_dialect>に1~3を指定することで、isql の実行時のDialectを設定できます。このオプションによる指定は、isql を終了するか、SET SQL DIALECT コマンドを使用してDialectを指定するまで有効です。

-r <rolename>
-r は-role と記述することも出来ます。
データベースへの接続にあたって、ユーザーに<rolename>の特権を与えます。

-c <num cache buffers>
データベースへの接続に使用するキャッシュサイズを指定することが出来ます。<num cache buffers>は一度に使用できるデータベースページ数を示します。デフォルトは75で最小値は45です。最大値はプラットフォームによって異なります。

-z
isql.exe のバージョン情報を表示します。

-nowarnings
isql の警告を抑制します。警告が表示されるのは以下の時です。

- 何も処理が発生しないSQL文を実行したとき。
- Dialect1 と Dialect3 で違う結果になるSQL文を実行したとき。
- API の呼出が、将来のバージョンで変更される予定となっている場合。
- データベースのシャットダウンが一時停止している場合。

-noautocommit
-n オプションと同様です。

実際の使用例を挙げると以下ようになります。

<ローカルデータベースへの接続>
isql d:\db\test.gdb -u SYSDBA -p masterkey

<リモートデータベースへの接続>
isql svr001:d:\db\test.gdb -u SYSDBA -p masterkey

<ローカルデータベースのメタデータを出力>
isql d:\db\test.gdb -x -u SYSDBA -p masterkey

<ローカルデータベースのメタデータをファイルへ出力>
isql d:\db\test.gdb -x -u SYSDBA -p masterkey -o d:\db\test.sql

<上記メタデータのデータベース名を変更>
isql d:\db\test.gdb -x -u SYSDBA -p masterkey -o d:\db\test.sql -d d:\db\test2.gdb

<SQL スクリプトファイルを実行>
isql -i d:\db\test.sql -u SYSDBA -p masterkey

<上記スクリプト実行時にメッセージを出さない>
isql -i d:\db\test.sql -u SYSDBA -p masterkey -q

isql の対話型モード

さて、すでに準備されている SQL スクリプトファイルを実行する場合やデータベースのメタデータを出力する場合以外は、isql の対話型モードを利用することになります。前述したローカル/リモートデータベースへの接続を行った場合、isql のコマンドプロンプトが表示されユーザーからの入力待ちとなります。

```
C:\>isql d:\db\test.gdb -u SYSDBA -p masterkey
Database: d:\db\test.gdb, User: SYSDBA
SQL>
```

なにもオプションをつけずに isql を起動した場合は、データベースの作成/接続を促すプロンプトが表示されます。

```
C:\>isql
Use CONNECT or CREATE DATABASE to specify a database
SQL>
```

この isql コマンドプロンプト上では、CREATE・ALTER・DROP・GRANT・REVOKE 等の DDL(Data Definition Language)文と、SELECT・INSERT・UPDATE・DELETE 等の DML(Data Manipulation Language)文、つまり通常の SQL コマンドと isql 独自のコマンドを実行することが出来ます。

SQL コマンドについては、別途解説をしていますので、isql コマンドについて述べていきたいと思えます。isql コマンドは大きく区分して、SHOW コマンド・SET コマンド・その他のコマンドの3つに分けることが出来ます。SHOW コマンドはデータベースのメタデータを表示する目的で使われます。SET コマンドは isql の環境設定のために使われます。その他のコマンドとしては、isql の終了やヘルプの表示コマンドなどがあります。

以下、各 isql コマンドを解説します。実行例をあげるために、サンプルデータベースである、employee.gdb を使用しました。employee.gdb は Firebird のインストールフォルダ直下の examples フォルダにインストールされています。

isql SHOW コマンド

SHOW CHECK テーブル名;

指定されたテーブルの CHECK 制約を表示します。表示されるのは、ユーザーが定義したメタデータのみです。

```
SQL> show check customer
CON> ;
CONSTRAINT INTEG_59:
CHECK (on_hold IS NULL OR on_hold = '*')
```

SHOW [DATABASE | DB];

データベースの情報を表示します。DB は DATABASE の省略形で、どちらも同じ結果を表示します。ここでは表示されていませんが、データベースのシャドウを作成している場合、シャドウセット番号も表示されます。

```
SQL> show db;
Database: employee.gdb
Owner: SYSDBA
PAGE_SIZE 4096
Number of DB pages allocated = 273
Sweep interval = 20000
Forced Writes are ON
Transaction - oldest = 427
Transaction - oldest active = 428
Transaction - oldest snapshot = 428
Transaction - Next = 434
```

SHOW [DOMAINS | DOMAIN ドメイン名];

DOMAINS を指定した場合、既存のドメインの一覧を表示します。また、DOMAIN ドメイン名を指定した場合は、そのドメインの情報を表示します。

```
SQL> show domains;
ADDRESSLINE BUDGET
```

COUNTRYNAME	CUSTNO
DEPTNO	EMPNO
FIRSTNAME	JOBCODE
JOBGRADE	LASTNAME
PHONENUMBER	PONUMBER
PRODTYPE	PROJNO
SALARY	TESTS

```
SQL> show domain addressline;
ADDRESSLINE          VARCHAR(30) Nullable
```

SHOW [EXCEPTIONS | EXCEPTION 例外名];

EXCEPTIONS を指定した場合、データベースでユーザーが定義した例外の一覧を表示します。EXCEPTION 例外名を指定した場合、その例外のテキストを表示します。

```
SQL> show exceptions;
```

```
Exception Name      Used by, Type
=====
CUSTOMER_CHECK     SHIP_ORDER, Stored procedure
  Overdue balance -- can not ship.
```

```
CUSTOMER_ON_HOLD   SHIP_ORDER, Stored procedure
  This customer is on hold.
```

```
ORDER_ALREADY_SHIPPED SHIP_ORDER, Stored procedure
  Order status is "shipped."
```

```
REASSIGN_SALES     DELETE_EMPLOYEE, Stored procedure
  Reassign the sales records before deleting this employee.
```

```
UNKNOWN_EMP_ID     ADD_EMP_PROJ, Stored procedure
  Invalid employee number or project id.
```

```
SQL> show exception customer_check;
```

```
Exception Name      Used by, Type
=====
CUSTOMER_CHECK     SHIP_ORDER, Stored procedure
  Overdue balance -- can not ship.
```

SHOW [FILTERS | FILTER BLOB フィルタ名];

FILTERS を指定した場合、BLOB フィルタの一覧を表示します。FILTER BLOB フィルタ名を指定した場合、その BLOB フィルタの情報を表示します。

※employee.gdb では BLOB フィルタは定義されていません。以下の例は、あくまでも仮定のものであります。

```
SQL> show filters;
```

```
JPEG_FILTER
```

```
SQL> show filter JPEG_FILTER;
```

```
BLOB Filter: JPEG_FILTER
Input subtype: -1 Output subtype -2
Filter library is: jpeg_filter
Entry point is: FILTER_LIB
```

SHOW [FUNCTIONS | FUNCTION ユーザー定義関数名];

FUNCTIONS を指定した場合、ユーザー定義関数 (UDF) の一覧を表示します。FUNCTION ユーザー定義関数名を指定した場合、その UDF の情報を表示します。

```
SQL> show filters;
```

```
There are no filters in this database
```

```
SQL> show functions;
```

ANSILIKE	CREATEUID
FREEMUTEX	STRFIRST
STRLAST	STRLCASE
STRLENGTH	STRTRIM
STRUCASE	SUBSTR
WAITMUTEX	WRITEDEBUG

```
SQL> show function CREATEUID;
```

```
Function CREATEUID:  
Function library is XLIBUDF.DLL  
Entry point is CreateUID  
Returns CSTRING(32) CHARACTER SET NONE
```

※XLIBUDF.DLLはSoft Complete Development社が開発して配布しているFREEのUDFライブラリです。
URL: <http://www.softcomplete.com/>

```
SHOW [GENERATORS | GENERATOR ジェネレーター名];
```

GENERATORSを指定した場合、ジェネレーターの一覧を表示します。GENERATOR ジェネレーター名を指定した場合、そのジェネレーターの情報が表示されます。

!!FB!! InterBase6に付属するisqlでは、マニュアルによるとジェネレーターの名前と、その次の値が表示されるとあります。一方、Firebird付属のisqlで表示されるのは、ジェネレーターの現在値となります。「次の値」というのは、GEN_ID()関数が、増分値を指定出来るようになって以上、意味のない数値であるように思います。Firebirdでは、明確に「current value is」と表記して、それが現在値であることを示しています。

```
SQL> show generators;  
Generator CUST_NO_GEN, current value is 1015  
Generator EMP_NO_GEN, current value is 145  
SQL> show generator CUST_NO_GEN;  
Generator CUST_NO_GEN, current value is 1015  
SQL>
```

本当に現在値なのかどうか試してみます。

```
SQL> select gen_id(cust_no_gen, 0)  
from rdb$database;
```

```
GEN_ID  
=====
```

```
1015
```

```
SHOW GRANT オブジェクト名;
```

オブジェクトにはセキュリティ特権の対象となる、テーブル・ビュー・ストアプロシージャを指定します。また、オブジェクト名にSQLロールを指定すると、SQLロールに含まれるユーザーを表示します。

```
SQL> show grant country;  
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON COUNTRY TO PUBLIC WITH GRANT OPTION
```

※employee.gdbにはSQLロールが作成されていません。以下の例では、SQLロールを作成して表示しています。

```
SQL> create role r_admin;  
SQL> grant all on country to r_admin;  
SQL> grant r_admin to tomneko;  
SQL> show grant r_admin;  
GRANT R_ADMIN TO TOMNEKO
```

```
SHOW [INDICES | INDEX
```

```
[インデックス名 | テーブル名]];
```

INDECESとINDEXは同義です。インデックス名やテーブル名を指定しない場合、データベースのすべてのインデックスを表示します。

```
SQL> show indicies;  
Command error: show indicies  
SQL> show indices;  
RDBSPRIMARY1 UNIQUE INDEX ON COUNTRY(COUNTRY)  
CUSTNAMEX INDEX ON CUSTOMER(CUSTOMER)
```

～中略～

```
RDBSPRIMARY24 UNIQUE INDEX ON SALES(PO_NUMBER)  
SALESTATX INDEX ON SALES(ORDER_STATUS, PAID)  
SQL> show index;  
RDBSPRIMARY1 UNIQUE INDEX ON COUNTRY(COUNTRY)  
CUSTNAMEX INDEX ON CUSTOMER(CUSTOMER)
```

～中略～

```
RDBSPRIMARY24 UNIQUE INDEX ON SALES(PO_NUMBER)
SALESTATX INDEX ON SALES(ORDER_STATUS, PAID)
SQL> show index salestatx;
SALESTATX INDEX ON SALES(ORDER_STATUS, PAID)
SQL> show index country;
RDBSPRIMARY1 UNIQUE INDEX ON COUNTRY(COUNTRY)
```

SHOW [PROCEDURES | PROCEDURE プロシージャ名];

PROCEDURES を指定した場合、データベースで定義されているすべてのストアードプロシージャを一覧表示します。PROCEDURE プロシージャ名を指定した場合、そのプロシージャの情報を表示します。

表示される情報には、そのプロシージャと依存関係のあるデータベースオブジェクトも含まれます。

```
SQL> show procedures;
```

Procedure Name	Dependency, Type
ADD_EMP_PROJ	EMPLOYEE_PROJECT, Table UNKNOWN_EMP_ID, Exception

～中略～

```
SHOW_LANGS          JOB, Table
SUB_TOT_BUDGET      DEPARTMENT, Table
```

SHOW [ROLES | ROLE SQL ロール名];

ROLES を指定すると、データベースで定義されているすべての SQL ロールを一覧表示します。ROLE ロール名を指定すると、その SQL ロールに属するユーザー（メンバ）を表示します。

```
SQL> show roles;
```

```
  R_ADMIN
```

```
SQL> show role r_admin;
```

```
TOMNEKO
```

SHOW SQL DIALECT;

現在のクライアントの SQL ダイアレクトとデータベースで設定されている SQL ダイアレクトを表示します。

```
SQL> show sql dialect;
```

```
Client SQL dialect is set to: 3 and database SQL dialect is: 3
```

SHOW SYSTEM [TABLES];

データベースのシステムテーブル及びシステムビューを表示します。

```
SQL> show system;
```

```
RDB$CHARACTER_SETS  RDB$CHECK_CONSTRAINTS
RDB$COLLATIONS      RDB$DATABASE
```

～中略～

```
RDB$USER_PRIVILEGES RDB$VIEW_RELATIONS
```

SHOW [TABLES | TABLE テーブル名];

TABLES を指定した場合、テーブルとビューの一覧を表示します。TABLE テーブル名を指定した場合、そのテーブル/ビューの情報を表示します。

```
SQL> show tables;
```

COUNTRY	CUSTOMER
DEPARTMENT	EMPLOYEE
EMPLOYEE_PROJECT	IBESCATGEGS
IBESCHAMPS	IBESDOC
IBESDOC_ATTRIBUTES	IBESDOC_RELATIONS
IBESWORK_TABLES	JOB
PHONE_LIST	PROJECT
PROJ_DEPT_BUDGET	SALARY_HISTORY
SALES	

```
SQL> show table country;
```

```
COUNTRY          (COUNTRYNAME) VARCHAR(15) Not Null
```

CURRENCY VARCHAR(10) Not Null
CONSTRAINT INTEG_2:
 Primary key (COUNTRY)

※一覧表示されたテーブル/ビューのうち、どれがビューであるかは、**SHOW VIEWS** コマンドを利用してください。

SHOW [TRIGGERS | TRIGGER トリガー名];

TRIGGERS を指定した場合、データベースで定義されているすべてのトリガーを一覧表示します。また、そのトリガーと依存関係のあるテーブルも表示されます。**TRIGGER** トリガー名を指定した場合、そのトリガーに関する情報を表示します。この場合、トリガーの定義（コード）自体も表示されます。

```
SQL> show triggers;
Table name                                    Trigger name
=====
CUSTOMER                                    SET_CUST_NO
EMPLOYEE                                    SAVE_SALARY_CHANGE
EMPLOYEE                                    SET_EMP_NO
SALES                                        POST_NEW_ORDER
SQL> show trigger SET_CUST_NO;
```

```
Triggers on Table CUSTOMER:
SET_CUST_NO, Sequence: 0, Type: BEFORE INSERT, Active
AS
BEGIN
    new.cust_no = gen_id(cust_no_gen, 1);
END
+++++
```

SHOW VERSION;

ISQL のバージョン、Firebird サーバー及び接続中のデータベースの ODS(On Disk Structure) のバージョンを表示します。

```
SQL> show version;
ISQL Version: WI-V1.0.2.908 Firebird 1.0
Firebird/x86/Windows NT (access method), version "WI-V6.2.908 Firebird 1.0"
on disk structure version 10.0
```

SHOW [VIEWS | VIEW ビュー名];

VIEWS を指定した場合、データベースで定義されているすべてのビューの一覧を表示します。**VIEW** ビュー名を指定した場合、そのビューの情報を表示します。

```
SQL> show views;
    PHONE_LIST
SQL> show view PHONE_LIST;
EMP_NO                    (EMPNO) SMALLINT Not Null
FIRST_NAME                (FIRSTNAME) VARCHAR(15) Not Null
LAST_NAME                (LASTNAME) VARCHAR(20) Not Null
PHONE_EXT                VARCHAR(4) Nullable
LOCATION                    VARCHAR(15) Nullable
PHONE_NO                 (PHONENUMBER) VARCHAR(20) Nullable
View Source:
=====
SELECT
    emp_no, first_name, last_name, phone_ext, location, phone_no
FROM employee, department
WHERE employee.dept_no = department.dept_no
```

isql SET コマンド

SET;

SET コマンドは、現在設定されている各種の **isql** 設定を表示します。

```
SQL> set
CON> ;
Print statistics:            OFF
Echo commands:              OFF
List format:                OFF
Row Count:                  OFF
Autocommit DDL:            ON
Access Plan:                OFF
```

```
Access Plan only:      OFF
Display BLOB type:    1
Terminator:           ;
Time:                 OFF
Warnings:             ON
```

SET コマンドの一覧は `HELP SET` コマンドで表示させることができます。

```
SQL> help set;
```

```
Set commands:
```

```
SET -- display current SET options
SET AUTODDL -- toggle autocommit of DDL statements
SET BLOB [ALL|<n>] -- display BLOBs of
                    subtype <n> or ALL
SET BLOB -- turn off BLOB display
SET COUNT -- toggle count of selected rows on/off
SET ECHO -- toggle command echo on/off
SET LIST -- toggle column or table display format
SET NAMES <csname> -- set name of
                    runtime character set
SET PLAN -- toggle display of query access plan
SET PLANONLY -- toggle display of
               query plan without executing
SET SQL DIALECT <n> -- set sql dialect to <n>
SET STATS -- toggle display of
            performance statistics
SET TIME -- toggle display of
           timestamp with DATE values
SET TERM <string> -- change statement
                  terminator string
SET WIDTH <col> [<n>] -- set/unset print width
                       to <n> for column <col>
```

All commands may be abbreviated to letters in CAPitals

最終行の表示にあるとおり、大文字で示された部分で各コマンドを省略することができます。

SET AUTODDL [ON | OFF]; (デフォルト ON)

DDL 文の実行後に自動コミットするかどうかを設定します。OFF の場合は、明示的にコミットする必要があります。

SET BLOB [ALL | <n>]; (デフォルト n=1)

ALL を指定した場合、すべてのサブタイプの BLOB を表示します。<n> を指定した場合、指定したサブタイプの BLOB のみを表示します。デフォルトは 1 となっており、isql はテキスト BLOB のみを表示します。n=0 は、未知の BLOB サブタイプに使用し、その他のサブタイプには特定の番号を指定します。

SET BLOB [ON | OFF]; (デフォルト ON);

ON を指定した場合、BLOB データが指定したサブタイプと必要な場合は適切な BLOB フィルタを使用して表示されます。OFF を指定した場合、サブタイプにかかわらず、BLOB は表示されません。

一度 OFF に指定すると、SET コマンドの表示は以下のようになります。

```
Display BLOB type:      NONE
```

この状態から、`SET BLOB ON;` を実行すると、サブタイプは 0 に初期化されます。OFF にする前に、特定のサブタイプを指定した場合は、ON にした後に、再度設定する必要があります。

SET COUNT [ON | OFF]; (デフォルト OFF)

ON を指定した場合、SELECT 文で行を選択したときに、選択された行数を表示します。

```
SQL> select * from country;
```

```
COUNTRY      CURRENCY
=====
USA           Dollar
~中略~
Fiji         FDollar
```

```
SQL> set count on;
```

```
SQL> select * from country;
```

```
COUNTRY          CURRENCY
=====
USA              Dollar
~中略~
Fiji            FDollar
```

```
Records affected: 14
```

SET ECHO [ON | OFF]; (デフォルト ON)

入力したコマンドを、指定された出力先（ディスプレイ/ファイル）に出力するかどうかを指定します。SQL スクリプトファイルを実行し、実行結果をファイルへ出力するような場合に、OFF を指定して、SQL コマンドを出力させないようにするためなどに使用します。

SET LIST [ON | OFF]; (デフォルト OFF)

SELECT 文の結果をリスト形式で表示するかどうかを指定します。OFF を指定した場合、これまでに示したような表形式で表示されます。ON を指定した場合、以下のような表示となります。

```
SQL> set list on;
SQL> select * from country;
```

```
COUNTRY          USA
CURRENCY         Dollar
~中略~
COUNTRY          Fiji
CURRENCY         FDollar
```

SET NAMES [キャラクタセット名]; (デフォルト NONE)

データベースへの接続で使用するキャラクタセット名を指定します。キャラクタセットの指定は、データベースに接続する前に行ってください。

デフォルトの NONE でデータの追加・更新を行った場合、データベースへ送られるデータは、データベースで指定されたキャラクタセットとの間で何らの変換も行われず、そのまま格納されます。その結果、他のコード体系で接続した場合、またはなんらかのコード変換を行う場合にエラーが発生してしまいます。

また、キャラクタセットが指定しされていない場合、コレーションオーダーなどの機能を使用することが出来なくなるので、データベースに接続するときは必ず指定するようにします。Windows の場合は SJIS_0208 を Linux を含む UNIX の場合は EUCJ_0208 を指定するようにしましょう。

SET PLAN [ON | OFF]; (デフォルト OFF)

ON を指定した場合、SELECT 文の実行時に、オプティマイザが指定したクエリープランを表示します。

```
SQL> set plan on;
SQL> select * from country;
```

```
PLAN (COUNTRY NATURAL)
```

```
COUNTRY          CURRENCY
=====
USA              Dollar
~中略~
Fiji            FDollar
```

SET PLANONLY [ON | OFF]; (デフォルト OFF)

!!FB!! ON を指定した場合、SELECT 文の実行時に、オプティマイザが指定したクエリープランのみを表示します。

```
SQL> set planonly on;
SQL> select * from country;
```

```
PLAN (COUNTRY NATURAL)
```

SET SQL DIALECT n; (デフォルト 3)

データベースへの接続に使用する、クライアントの SQL DIALECT を指定します。SQL DIALECT は 1, 2, 3 を指定することが出来ます。それぞれの意味は以下の通りです。ちなみに、ダイアレクトとは方言の意味です。

1: InterBase5.5 以前のバージョンとの交換性のために使用します。

2: ダイアレクト 1 からダイアレクト 3 への移行のために使用される暫定的ダイアレクト。

3: 区切付識別子、64 ビット数値、TIMESTAMP 型の導入に伴う DATE・TIME 型の変更など、Firebird 本来の機能を使用できるダイアレク

ト。

SET STATS [ON | OFF]; (デフォルト OFF)

ON を指定した場合、SELECT 文の実行時に、システムのパフォーマンスを示す統計情報を表示します。

```
SQL> set stats on;
```

```
SQL> select * from country;
```

```
COUNTRY          CURRENCY
=====
USA              Dollar
~中略~
Fiji            FDollar
```

```
Current memory = 9144320
Delta memory = 0
Max memory = 9249880
Elapsed time= 0.00 sec
Buffers = 2048
Reads = 0
Writes 0
Fetches = 31
```

ここで表示される統計情報は、それぞれ以下の通りです。

Current memory: 現在使用可能なメモリ量 (バイト)

Delta memory: 使用され変化したメモリ量 (バイト)

Max memory: 使用可能な最大メモリ量 (バイト)

Elapsed memory: 経過時刻

Buffers: キャッシュバッファの容量 (バイト)

Reads: 読み取り要求回数

Writes: 書き込み要求回数

Fetches: 実行されたフェッチ回数

SET TIME [ON | OFF]; (デフォルト OFF)

DATE 型のデータの時刻部分を表示するかどうかを設定します。ダイアレクト 3 の場合、DATE 型には時刻情報は含まれないので、ON でも OFF でも変化はありません。

```
SQL> create table t_test(id integer not null primary key, dt date);
```

```
SQL warning code = 301
```

```
-DATE data type is now called TIMESTAMP
```

```
SQL> insert into t_test values(1, '2003/03/03');
```

```
SQL> select * from t_test;
```

```
          ID          DT
=====
          1  3-MAR-2003 00:00:00.0000
```

```
SQL> set time off;
```

```
SQL> select * from t_test;
```

```
          ID          DT
=====
          1  3-MAR-2003
```

※上記の例では DIALECT 1 のデータベースに DIALECT 1 で接続しています。

SET TERM ターミネータ; (デフォルト ;)

isql がコマンドの区切を識別するために必要とするターミネータを指定します。通常、ストアードプロシージャやトリガーの作成時に使用します。

ストアードやトリガーは、セミコロンによって区切られた文の集合として定義されますが、isql からはターミネータによって識別される 1 行のコマンドとして解釈されなくてはなりません。詳しくは後述しますが、^や!!などにターミネータを一時的に変更して、ストアードやトリガーを作成することになります。

SET WIDTH <col> [<n>];

SELECT 文の実行時に、列の表示幅を設定するコマンドのはずですが、機能していないようです？

isql その他のコマンド

ADD テーブル名;

!!FB!! テーブルへのデータの追加を対話形式で行います。一見便利のように見えますが、日付データの入力時におかしな挙動が見受けられます。

```
SQL> add t_test;
```

```
Enter data or NULL for each column. RETURN to end.
```

```
Enter ID>2
```

```
Enter DT as M/D/Y>3/3/2003
```

```
Enter ID>
```

```
Input parsing problem
```

```
SQL> select * from t_test2;
```

```
          ID          DT
=====
1 2003-03-03
2 3903-03-03
```

※見ての通り、年数が 1900 年から数えて何年目になってしまっています。いろいろ試しましたが、どうもこれはバグのようで、SourceForge のバグリストにあがっています。文字型のデータでの改行が入力出来ない問題などもあるようです。

BLOBDUMP BLOB_ID ファイル名;

BLOBDUMP コマンドは、指定された BLOB データをファイルに格納するために使用します。テキスト型以外の BLOB データは、isql で表示することが出来ないため、ファイルへ出力して確認することになります。テキスト型の BLOB の場合も、指定したブロックを一括してファイルに保存することができます。

EDIT [ファイル名];

環境変数 EDITOR で指定される外部エディタを使用して、SQL スクリプトファイルまたは、一括実行されるコマンドラインを編集します。環境変数を指定しない場合、InterBase6 では notepad.exe が実行されるようですが、Firebird では「mep」というエディタが呼ばれてしまい、結果的にエラーになります。EDIT コマンドを使用する場合、必ず環境変数 EDITOR を設定してください。

EXIT;

現在アクティブなトランザクションをコミットして、データベースへの接続を閉じて、isql を終了します。

HELP;

isql のコマンドの一覧と、簡単な解説を表示します。

INPUT ファイル名;

指定したファイルから、SQL スクリプトを読み込んで実行します。

SQL スクリプトファイルに、INPUT 文を含めることでネストした実行もサポートされています。複数の文を実行する場合、呼び出されたスクリプトが終了すると、一つ前のスクリプトに実行が戻されます。サブルーチンのように使用できます。

ファイル名を二重引用符で囲むとスペースを含んだファイルを使用することが出来ます。非対話型の利用となるので、EDIT 文は利用できません。

OUTPUT [ファイル名];

指定したファイル名または標準出力に出力先を切り替えます。ファイル名を指定しない場合、標準出力を指定したことになります。

INPUT と同様に、ファイル名を二重引用符で囲むとスペースを含んだファイルを使用することが出来ます。

エラーメッセージはファイルに出力することが出来ません。

QUIT;

現在アクティブなトランザクションをロールバックして、データベースへの接続を閉じて、isql を終了します。

SHELL [OS のシェルコマンド];

OS のシェルコマンドを実行します。何も指定しないときは、シェルが子プロセスとして実行されます。子プロセスから抜けるには、各シェルの終了コマンド（通常は EXIT）を実行します。

Firebirdのオブジェクト命名規則

Firebirdでテーブル、フィールド、トリガー等の名前を付けるさいには、以下の命名規則に従わなくてはなりません。

- ・ スペースを含めることはできません。
 - ・ 大文字と小文字は区別されません。
 - ・ Firebirdの予約語と同じであってははいけません。

 - ・ ただし、Dialect3のデータベースで、オブジェクトの名前を二重引用符で囲った場合はこの限りではありません。その場合、次のようなことが可能となります。
このような名前を区切付識別子(Delimited Identifier)と呼びます。
- a) Firebirdの予約語を使用できます。
 - b) 名前にスペースを含めることができます。ただし、名前の最後の空白は無視されます。
 - c) 大文字と小文字が区別されます。

区切付識別子は、大文字と小文字が区別された名前を好むユーザーにとって朗報となるかもしれませんが、一方で厄介な問題を抱えています。

標準SQLでは大文字と小文字は区別されないため、他のRDBMSで大文字と小文字を意識しないプログラミングになれてきた人々を戸惑わせることになります。

ISQLでは二重引用符を付けない名前に付いてはすべて大文字に変換してから実行されます。この辺はIBConsoleやIBOConsoleでも同様です。ただし、Dialect3のデータベースでは大文字に変換した上で、二重引用符が付加されて実行されてしまうようです。

そのため、DelphiのdbExpressでSQL文を発行する際に小文字の名前がはねられてしまい、エラーが出るということが起こります。

筆者が確認したのはDelphi6のdbExpressですが、SELECT文は小文字でも通るのに、INSERT文は大文字で無いとだめという現象に出会い、原因を解明するのに手間取りました。

また、筆者が日本語化して公開しているIBOConsoleでは、GUIを使用してテーブル等の作成ができるのですが、この場合そのまま二重引用符が付加されてしまうため、大文字小文字混在のオブジェクト名を使用すると、後になっていちいちすべて二重引用符で囲まないとSQLが通らないという事態に陥ってしまいます。

そこで、SQLについては原則として大文字で書くということを徹底するに限るというのが結論です。ユーザーの自由度を高めるための設計が、逆にユーザーを縛っているような気がして残念です。

※ところで、本書ではサンプルのSQLを見やすくするためにオブジェクト名については大文字小文字混在で表記しています。上記主張とは逆になりますが、表現としてはやはりこちらの方がわかりやすいですね。

Firebirdのセキュリティ

Firebirdでは、データベースへのアクセスはユーザー名とパスワードによって管理されています。また、ANSI SQLで規定されているSQLロールをサポートし、グループレベルでのセキュリティにも対応しています。

これらの情報は、Firebirdのインストールディレクトリにあるisc4.gdbに保持されています。サーバー上のデータベースに接続するためには、gsec.exeやIBOConsole等のユーティリティを利用してユーザー名とパスワードを登録する必要があります。

Firebirdのユーザー名は半角で31文字までの長さで大文字と小文字は区別されません。パスワードは半角で8文字までで大文字と小文字が区別されます。デフォルトで特権ユーザーのSYSDBAが登録されています。SYSDBAのパスワードは、よく知られているように「masterkey」ですが、Firebirdをインストールしたら速やかに他のパスワードに変更するよう注意してください。

データベース上の各オブジェクトに対する操作上の特権は、GRANTとREVOKEの各SQLコマンドで行います。また、各種のユーティリティを利用して簡単に設定することも出来ます。

GRANT文

GRANT文は、各データベースオブジェクトに対する特権をユーザー/ロール等に割り当てます。テーブル、ビュー、ストアドプロシージャの各データベースオブジェクトは、作成した時点では作成者のみが各オブジェクトに対するすべての特権を所有しています。作成者は、GRANT文を使用して、各特権を他のユーザー/ロール等に付与することが出来ます。

GRANT

```
<特権> ON [TABLE] { テーブル名 | ビュー名 }
  TO { <オブジェクト>
      | <ユーザーリスト>
      | GROUP UNIX グループ名 }
  | EXECUTE ON PROCEDURE プロシージャ名
  TO { <オブジェクト>
      | <ユーザーリスト>}
  | <許可済ロール> TO {PUBLIC | <被許可リスト>};
<特権> = ALL [PRIVILEGES] | <特権リスト>
<特権リスト> =
{ SELECT
| DELETE
| INSERT
| UPDATE [( 列名 [, 列名 ...])]
| REFERENCES [( 列名 [, 列名 ...])] }
[, <特権リスト> ...]
<オブジェクト> =
{ PROCEDURE プロシージャ名
| TRIGGER トリガー名
| VIEW ビュー名
| PUBLIC }
[, <object> - ]
<ユーザーリスト> =
{ [USER] ユーザー名
| ロール名
| UNIX ユーザー名 }
[, <ユーザーリスト> ...]
[WITH GRANT OPTION]
<ロールリスト> =
ロール名 [, ロール名 ...]
<許可済ロール> =
ロール名 [, ロール名 ...]
<被許可リスト> =
[USER] ユーザー名
[, [USER] ユーザー名 ...]
[WITH ADMIN OPTION]
```

- 各特権の詳細は以下のようになります。

特権	実行可能動作
ALL	SELECT/DELETE/INSERT/UPDATE/REFERENCES
SELECT	テーブル/ビューから行を取得する
DELETE	テーブル/ビューから行を削除する
INSERT	テーブル/ビューに行を挿入する
UPDATE	テーブル/ビューの1列以上の値を変更する。列を一部に制限することができる
EXECUTE	ストアドプロシージャを実行する
REFERENCES	外部キーで指定された列を参照する。一時キーに含まれるすべての列にREFERENCES特権が指定されてなければならない

- 各ユーザーは許可された特権に従って、各データベースオブジェクトにアクセスすることができます。すべてのユーザーに対して

許可を与えるには、**PUBLIC** キーワードを使用します。**PUBLIC** に対して与えられた特権は、現在存在するすべてのユーザーと以降に追加されるすべてのユーザーで有効です。

ただし、**GRANT hogehoge TO PUBLIC** で与えられた特権は **REVOKE hogehoge FROM PUBLIC** でのみ取り消すことが可能です。

- **WITH GRANT OPTION** を指定すると、許可されたユーザーはその特権を他のユーザーに許可できるようになります。許可できるのは、自分が所有している特権だけに限られます。
- **SQL** ロールが所有している特権の譲渡を許可するためには、**WITH ADMIN OPTION** を指定します。

例：

```
GRANT SELECT, REFERENCES ON T_Test TO potesi;
GRANT ALL ON T_Test TO PROCEDURE SP_Test;
```

```
GRANT ALL ON T_Test TO PUBLIC;
```

REVOKE 文

REVOKE 文は、指定したデータベースオブジェクトに対すユーザー/ロール等の特権を取り消します。

REVOKE

```
[GRANT OPTION FOR] <特権> ON [TABLE]
{ テーブル名 | ビュー名 }
FROM { <オブジェクト>
      | <ユーザーリスト>
      | <ロールリスト>
      | GROUP UNIX グループ名 }
| EXECUTE ON PROCEDURE procname
  FROM { <オブジェクト>
        | <ユーザーリスト> }
| <許可済ロール> FROM {PUBLIC | <被許可リスト>};
<特権> = ALL [PRIVILEGES] | <特権リスト>
<特権リスト> =
{ SELECT
  | DELETE
  | INSERT
  | UPDATE [( 列名 [, 列名 ...])]
  | REFERENCES [( 列名 [, 列名 ...])] }
[, <特権リスト> ...]
<オブジェクト> =
{ PROCEDURE プロシージャ名
  | TRIGGER トリガー名
  | VIEW ビュー名
  | PUBLIC }
[, <object> - ]
<ユーザーリスト> =
{ [USER] ユーザー名
  | ロール名
  | UNIX ユーザー名 }
[, <ユーザーリスト> ...]
[WITH GRANT OPTION]
<ロールリスト> =
ロール名 [, ロール名 ...]
<許可済ロール> =
ロール名 [, ロール名 ...]
<被許可リスト> =
[USER] ユーザー名
[, [USER] ユーザー名 ...]
[WITH ADMIN OPTION]
```

- 各ユーザー/ロール等が所有している特権を取り消すことができるのは、その特権を与えたユーザーだけです。同じ特権を複数のユーザー等から与えられている場合、**REVOKE** を実行するユーザーが与えた特権のみが削除されます。
GRANT ALL で与えた特権は、**REVOKE ALL** でないと取り消すことが出来ません。例えば、**GRANT ALL** と **GRANT SELECT** を許可されているユーザーから、**REVOKE SELECT** を行っても **SELECT** 特権は残ります。逆に、**REVOKE ALL** を行くと、**SELECT** 特権も取り消されません。
- **GRANT hogehoge TO PUBLIC** で与えた特権に対して、一部のユーザーのある特権だけを取り消すと言うことは出来ません。PostgreSQL では、**GRANT hogehoge TO PUBLIC** としてから、**REVOKE hogehoge FROM NOBODY** などとすることに意味がありますが、Firebird では、このようなことをしても意味がありません。(エラーにはなりません)

Firebirdのトランザクション

Firebirdは高度なトランザクション管理機能を備えたリレーショナルデータベースです。Firebirdのトランザクション機能は履歴型アーキテクチャー(Multi Generation Architecture)によって実現されています。履歴型アーキテクチャーはPostgreSQLでバージョン6.5から採用されているMVCC(Multi Version Concurrency Control)と同様の機能で、Firebirdの出発点となっているアーキテクチャーでもあるので、PostgreSQLはようやくFirebirdに追いついたところと言えるかもしれません。

トランザクションは以下の4つの特徴によって定義されます。そのイニシャルによってACID特性と呼ばれています。

Atomicity(原子性)

トランザクションはそれ以上分割することの出来ない最小単位でなければなりません。各トランザクションを構成する処理のそれぞれは、すべて有効となるか、すべて無効となるかのどちらかでなければなりません。

処理1と処理2が一つのトランザクションである場合、片方だけが行われるということがあってはなりません。

一貫性(CONSISTENCY)

トランザクションの処理対象となるデータは、その実行の前後で整合性を保持し、一貫していなければなりません。

隔離性(ISOLATION)

複数のトランザクションの処理対象となるデータが同一である場合、各トランザクションはお互いに隔離された状態で処理を実行しなければなりません。

論理的には、相互に完全に独立した状態を保持できるのが最も望ましいわけですが、実際には各トランザクションの相互作用に応じていくつかの隔離レベルが規定されています。

持続性(DURABILITY)

トランザクションが正常に終了した場合、確定した処理内容は維持されなくてはなりません。現実的には、メモリからディスクへと変更が書き込まれることを意味します。

ANSI SQL標準では、「READ UNCOMMITTED」、「READ COMMITTED」、「REPEATABLE READ」、および「SERIALIZABLE」という4つのトランザクション隔離レベルが定義されています。

これらは、それぞれ複数のトランザクションの同時実行によって起こりうる現象に対して、どのような対応が行われているかによって区分されています。複数のトランザクションの同時実行によって起こりうる現象は以下の4つが規定されています。

ロスト・アップデート

トランザクション中に変更した値が、並行して実行されている他のトランザクションにより別の値に変更されてしまい、コミットしたとしても更新が行われなかったことになってしまう現象を示します。これは、最低限の隔離レベルである「READ UNCOMMITTED」でも起こってはならない現象とされています。逆に言えば、「ロスト・アップデート」が起こることということは、それはトランザクションではないということになります。

ダーティ・リード

トランザクション中に変更した値を、他のトランザクションが読み取ってしまい、その後ロールバックした場合に、確定しなかった値を結果的に読み取ってしまうことになる現象を示します。コミットされていない値を読み取ることから生じるこの現象は、「READ UNCOMMITTED」でのみ起こりえます。

ノン・リピータブル・リード

トランザクション中に同じ行を複数回読み取った場合に、その途中で他のトランザクションによって値が変更されコミットされたことによって、読み取る値が変わってしまう現象を示します。他のトランザクションによるコミットの結果に影響を受けるため、「READ UNCOMMITTED」「READ COMMITTED」において起こりえます。

ファントム・インサート

トランザクション中に同一の条件で複数の行を複数回選択した場合に、その途中で他のトランザクションによって行が挿入されたため、選択した行の結果が変わってしまう現象を示します。「READ UNCOMMITTED」「READ COMMITTED」「REPEATABLE READ」で起こりえます。

Firebirdはこのうち、「READ UNCOMMITTED」だけサポートしていません。MS SQL ServerやIBM DB2は4つすべてを備えています。Oracleは「READ COMMITTED」と「SERIALIZABLE」しかサポートしていません。Oracleの資料によれば、「それで十分だから」ということです。実際の運用において、「READ UNCOMMITTED」を使用する意味があるのは読み取り専用データベースくらいしかありませんので、理論的には存在していても現実的には意味の無い機能といえるでしょう。

履歴型アーキテクチャーによって、Firebirdは高度なトランザクション処理機能を備えています。もっとも高い分離レベルである「SERIALIZABLE」は、Firebirdでは「SNAPSHOT」と呼ばれますが、「READ COMMITTED」では避けることの出来ない他のトランザクションによってコミットされた値の影響を完全に排除することが出来ます。

この場合、通常の行レベルロックでは読み取り中の行は他のトランザクションからは書き込みが行えないようになっています。Firebirdでは履歴型アーキテクチャーによって、複数のトランザクションが同一の行に対して読み取りと書き込みを同時に行うことが出来ます。また、テーブル全体にロックをかけて読取専用のトランザクションとして利用できる「SNAPSHOT TABLE STABILITY」という分離レベルも利用できます。

Firebird のトランザクションコマンド

Firebird のトランザクション関連のコマンドは、**SET TRANSACTION** でトランザクションを開始し、**COMMIT** で確定、**ROLLBACK** で破棄となっています。SQL 標準では、**BEGIN TRANSACTION / COMMIT TRANSACTION / ROLLBACK TRANSACTION** となっていますので、どちらかといえば **Oracle** と同じで独自の構文になっています。

SET TRANSACTION

SET TRANSACTION 文はトランザクションの開始を宣言します。

SET TRANSACTION

```
[READ WRITE | READ ONLY]
[WAIT | NO WAIT]
[[ISOLATION LEVEL]
 {SNAPSHOT [TABLE STABILITY]
 | READ COMMITTED
 | [[NO] RECORD_VERSION]}]
[RESERVING <予約句> ];
<予約句> = テーブル名 [, テーブル名 ...]
[FOR [SHARED | PROTECTED]
 {READ | WRITE}] [, <予約句>]
```

- **READ WRITE** と **READ ONLY** はトランザクションがデータベースに対して、読み書きを行うのか読み取り専用なのかを指定します。省略した場合の規定値は **READ WRITE** です。
- **WAIT** と **NO WAIT** は、更新の衝突などに際して他のトランザクションの終了を待つのか、それともデッドロックを報告するのかを指定します。並行して動作する他のトランザクションが更新した行を対象とした読み取りまたは更新動作などを行った場合、通常はロックがかかり他のトランザクションのコミット又はロールバックを待たなくてはなりません。**NO WAIT** モードの場合、衝突する更新に対してはデッドロックエラーを返しますが、読み取りで待たされることはありません。省略した場合の規定値は **WAIT** です。
- **ISOLATION LEVEL** は、トランザクションの隔離レベルを指定します。詳しくは前述しました。省略したときの規定値は **SNAPSHOT** です。
READ COMMITTED を指定した場合、**[NO] RECORD VERSION** を指定することが出来ます。**NO RECORD VERSION** を指定した場合、常に最新の行を読み取ります。他のトランザクションによるコミット/ロールバックされていないデータがあるときに、**WAIT** モードではコミット/ロールバックを待機します。
RECORD VERSION を指定した場合、トランザクション開始時点でのコミット済データのみを対象として読み取りを開始します。
- **RESERVING** 句は、テーブルをトランザクションで使用するために予約します。例えば **PROTECTED READ** と **PROTECTED WRITE** を指定した場合、そのテーブルは他のトランザクションから読み取りも書き込みもできなくなります。

例：

```
SQL> SET TRANSACTION READ ONLY WAIT
ISOLATION LEVEL READ COMMITTED
NO RECORD_VERSION
RESERVING T_TEST FOR PROTECTED READ ,
T_TEST FOR PROTECTED WRITE;
```

※全部指定してみると上記のようになります。

COMMIT

COMMIT 文は、トランザクションによって行われた変更内容を確定し、データベースに永続化します。具体的にはディスクへの書き込みが行われ、以降のトランザクションで確定した内容として読み取りを行うことが可能となります。

COMMIT [WORK] [RETAIN [SNAPSHOT]];

- **WORK** 句は旧バージョンとの交換性のために存在します。
- **RETAIN [SNAPSHOT]** を指定した場合、カーソルは閉じられません。省略した場合のデフォルトの動作では、**COMMIT** 後にカーソルが閉じられます。

ROLLBACK

ROLLBACK 文は、トランザクションが開始される前の状態にデータベースを復元します。

ROLLBACK [WORK];

- **WORK** 句は旧バージョンとの交換性のために存在します。

Firebirdのトランザクション実行例

まず、`isql` を2つ起動して、同じデータベースに接続して、テスト用のテーブルを作成します。
2つの`isql` はここでは、`isql (1)`、`isql (2)`と表記しますが、実際にはコマンドプロンプトを2つ起動して、並べて実行することになります。また、見やすくするために適当に改行を入れてあります。

```
[isql (1)]
D:\DB>isql test.gdb -u SYSDBA -p hogehoge
Database: test.gdb, User: SYSDBA
SQL> create table t_test (
    id integer not null primary key,
    name char(10));
```

```
[isql (2)]
D:\DB>isql test.gdb -u SYSDBA -p masterkey
Database: test.gdb, User: SYSDBA
SQL>
```

隔離レベル READ COMMITTED

つぎに、それぞれの`isql` でトランザクションを開始します。まずは、`Isolation Level READ COMMITTED` を試してみましょう。ロック待ちで止まってしまうのをさけるため、`NO WAIT` オプションをつけておきます。

```
[isql (1) (2)]
SQL> set transaction no wait isolation level read committed;
Commit current transaction (y/n)?y
Committing.
SQL>
```

デフォルト・トランザクションをコミットするかどうか訪ねてくるので、`y` と答えておきましょう。`Firebird` はすべてのデータベース操作が何らかのトランザクションに属していなければならないため、`isql` ではデフォルトトランザクションが実行されています。また、`Firebird` はネストしたトランザクションをサポートしていないため、デフォルト・トランザクションの処理を完了しないと、次のトランザクションを開始することが出来ません。

では、`isql (1)` でテーブルにデータを挿入してみます。

```
[isql (1)]
SQL> insert into t_test values(1, 'tom');
SQL> insert into t_test values(2, 'jelly');
SQL> insert into t_test values(3, 'potesi');
SQL>
```

この状態で、`isql (2)` でテーブルを `SELECT` してみます。

```
[isql (2)]
SQL> select * from t_test;

      ID NAME
=====
Statement failed, SQLCODE = -901

lock conflict on no wait transaction
-deadlock
SQL>
```

コミットされていないデータがあるため、デッドロックが生じて、エラーが発生します。`WAIT` モードの場合は、`isql (1)` のトランザクションがコミットするまで待たされることになります。後で見ますが、`SNAPSHOT` の場合、`isql (2)` のトランザクション開始時点のテーブルの内容が問題なく表示されます。

それでは、`isql (1)` のトランザクションをコミットしてから、もう一度 `isql (2)` で `SELECT` をしてみます。

```
[isql (1)]
SQL> commit;
SQL>

[isql (2)]
SQL> select * from t_test;
```

```
      ID NAME
=====
```

```
1 tom
2 jelly
3 potesi
```

SQL>

isql (2)では、トランザクションのコミットを行っていないにも関わらず、isql (1)で挿入された行がSELECTされてしまいます。これが、**ファントム・インサート**です。トランザクション開始時点では、テーブルには全く存在しませんでした。ところが、別のトランザクションによって挿入された行が、いつの間にかSELECT出来てしまうわけです。

それでは、もう一度isql (1)でREAD COMMITTEDのトランザクションを開始して、テーブルの内容を更新してみることにします。更新後に、内容を確認します。

[isql (1)]

```
SQL> set transaction no wait isolation level read committed;
SQL> update t_test set name = 'tomneko' where id = 1;
SQL> select * from t_test;
```

```
      ID NAME
=====
1 tomneko
2 jelly
3 potesi
```

SQL>

この時点でisql (2)がSELECTを行うと再びdeadlockが発生してしまいます。

[isql (2)]

```
SQL> select * from t_test;
```

```
      ID NAME
=====
Statement failed, SQLCODE = -901
```

```
lock conflict on no wait transaction
-deadlock
SQL>
```

isql (1)をコミットしてから、再びSELECTを実行します。

[isql (1)]

```
SQL> commit;
SQL>
```

[isql (2)]

```
SQL> select * from t_test;
```

```
      ID NAME
=====
1 tomneko
2 jelly
3 potesi
```

SQL>

ここまで、isql (2)は一度もコミットを行っていません。にもかかわらず、ID=1の行の内容が変わってしまっています。これが、**ノン・リピータブル・リード**です。1回目の読み取りでは、'tom'だった内容が、2回目の読み取りでは'tomneko'に変化してしまっているわけです。

隔離レベル SNAPSHOT

さて、両方のトランザクションをコミットして、SNAPSHOT トランザクションを開始してみます。

[isql (1)]

```
SQL> set transaction no wait isolation level snapshot;
SQL>
```

```
[isql(2)]
SQL> commit;
SQL> set transaction no wait isolation level snapshot;
SQL>
```

それでは、まず isql (1) で行の挿入を行い、isql (2) で SELECT してみることにします。

```
[isql(1)]
SQL> insert into t_test values(4, 'hogeri');
SQL>
```

```
[isql(2)]
SQL> select * from t_test;
```

```
          ID NAME
=====
          1 tomeko
          2 jelly
          3 potesi
```

```
SQL>
```

こんどは、デッドロックを引き起こさずに SELECT 文が実行され、トランザクション開始時点のテーブルの内容が表示されます。isql (1) のトランザクションをコミットしてみるとどうなるでしょうか。コミット後に、再度 isql (2) で SELECT 文を実行してみます。

```
[isql(1)]
SQL> commit;
SQL>
```

```
[isql(2)]
SQL> select * from t_test;
```

```
          ID NAME
=====
          1 tomeko
          2 jelly
          3 potesi
```

```
SQL>
```

isql (2) のトランザクション中では、isql (1) のトランザクションによって挿入された行が見えていません。つまり、ファントム・インサートは起こっていません。

さらに、isql (1) でもう一度別のトランザクションを開始して、行の更新を行ってみます。

```
[isql(1)]
SQL> set transaction no wait isolation level snapshot;
SQL> update t_test set name = 'tom' where id = 1;
SQL> select * from t_test;
```

```
          ID NAME
=====
          1 tom
          2 jelly
          3 potesi
          4 hogeri
```

```
SQL>
```

```
[isql(2)]
SQL> select * from t_test;
```

```
          ID NAME
=====
```

```
1 tomeko
2 jelly
3 potesi
```

SQL>

ここでも、isql (2) で実行した SELECT 文ではまったくテーブルの内容が変化していないことがわかります。つまり、**ノン・リピータブル・リードは起こっていません**。isql (1) の SELECT 文の結果と照らし合わせると、ずいぶん違う内容になっています。

このことから、Firebird のトランザクションにおける、**ISOLATION LEVEL SNAPSHOT はシリアライザブルである**ということがわかります。

隔離レベル *SNAPSHOT TABLE STABILITY*

最後に、SNAPSHOT TABLE STABILITY のトランザクションを実行してみます。

```
[isql(1)]
SQL> set transaction no wait isolation level snapshot table stability;
SQL> update t_test set name = 'tomeko' where id = 1;
Statement failed, SQLCODE = -901
```

```
lock conflict on no wait transaction
SQL> insert into t_test values(5, 'foo');
Statement failed, SQLCODE = -901
```

```
lock conflict on no wait transaction
SQL>
```

挿入・更新とも不可能なことがわかります。他のトランザクションからはどうでしょうか。isql (2) で **READ COMMITTED** のトランザクションを開始して挿入と更新を試してみます。

```
[isql(2)]
SQL> set transaction no wait isolation level read committed;
SQL> insert into t_test values(5, 'foo');
Statement failed, SQLCODE = -901
```

```
lock conflict on no wait transaction
SQL> update t_test set name='tomeko' where id = 1;
Statement failed, SQLCODE = -901
```

```
lock conflict on no wait transaction
SQL> select * from t_test;
```

```
      ID NAME
=====
1 tom
2 jelly
3 potesi
4 hogeri
```

SQL>

やはり、どちらもデッドロックとなり挿入・更新を行うことが出来ません。しかし、読み取りは可能です。さらに、isql (2) で **SNAPSHOT** のトランザクションを開始して、同様に試してみます。

```
[isql(2)]
SQL> set transaction no wait isolation level snapshot;
SQL> insert into t_test values(5, 'foo');
Statement failed, SQLCODE = -901
```

```
lock conflict on no wait transaction
SQL> update t_test set name = 'tomeko' where id = 1;
Statement failed, SQLCODE = -901
```

```
lock conflict on no wait transaction
SQL> select * from t_test;
```

```
      ID NAME
```

=====

1 tom
2 jelly
3 potesi
4 hogeri

SQL> delete from t_test where id = 4;
Statement failed, SQLCODE = -901

lock conflict on no wait transaction
SQL>

DELETE 文も試してみましたが、すべて実行することができません。**TABLE STABILITY** オプションがテーブル全体をロックすることがわかります。

Firebird SQL - CREATE 系コマンド

DDL(Data Definition Language)系コマンド=データ定義言語に含まれる、CREATE/ALTER/DROP等のコマンド群は、データベースの構造を定義するために使用されます。CREATE系コマンドは、各データベースオブジェクトを作成するために使用します。

CREATE DATABASE

CREATE DATABASE文は、新しいデータベースを作成します。

```
CREATE {DATABASE | SCHEMA} 'ファイル名'  
[USER 'ユーザー名' [PASSWORD 'パスワード']]  
[PAGE_SIZE [=] int ] [LENGTH [=] 整数値 [PAGE[S]]]  
[DEFAULT CHARACTER SET キャラクタセット名 ]  
[ <二次ファイル情報 >];  
<二次ファイル情報 > =  
FILE 'ファイル名' [ <ファイル情報>]  
[ <二次ファイル情報>]  
<ファイル情報> =  
{LENGTH [=] int [PAGE[S]] |  
STARTING [AT [PAGE]] int }  
[ <ファイル情報>]
```

- DATABASE と SCHEMA は等価です。どちらを使っても構いません。

- ファイル名の記述方法は以下の通りです。

Win32 環境

[サーバー名[/ポート番号]:]ドライブ名\パス名

例.

SVR01/3050: C:\DB\SAMPLE.GDB

Linux・UNIX 環境

[サーバー名[/ポート番号]:]/パス名

例.

SVR02/3050: /HOME/DB/SAMPLE.GDB

!!FB!! ポート番号の指定は任意です。省略した場合は、デフォルトポートの 3050 で通信します。

- USER 'ユーザー名' は isql の起動時に -u オプションで指定してあれば省略することが出来ます。ユーザー名はデータベースを作成するサーバーのセキュリティデータベースで有効なユーザー名を指定します。InterBase のドキュメントには以下のような記述がありますが、isql の起動時に指定したユーザー名・パスワードと違うユーザー名を指定してもデータベースは作成されます。

「Windows のクライアントアプリケーションは、サーバーとの接続に使用したユーザー名を指定しなければなりません。」

- ユーザー名とパスワードはサーバーのセキュリティデータベースに登録されている有効な組み合わせを指定します。
- PAGE_SIZE はデータベースのページサイズを指定します。指定できる値は、1024・2048・4096・8192・16384 のいずれかです。

!!FB!! ページサイズのデフォルトは InterBase6 の 1024 から 4096 へと変更されています。また、ページサイズとして 16384 を指定できるようになりました。ページサイズを大きくすると、ファイルの利用効率は下がりますが、パフォーマンスが改善される可能性があります。

- LENGTH は 1 次ファイルまたは 2 次ファイルのデータベースページ数を指定します。PAGE_SIZE で指定したデータベースページの大きさに対して、何ページまでのファイルとするかを指定します。ここでの指定によって OS のファイルサイズ上限を超える場合は、指定は無視されます。また、Firebird はここでの指定に関わらず、データベースの最後のファイルを OS のファイルサイズ上限まで拡張します。

!!FB!! ファイルサイズの上限は 64bit/0 の導入により変更されました。InterBase では 2GB (Windows では 4GB) となっていました。新しいファイルサイズの制限は以下のとおりです。

FAT16 for Win9x/ME, max file size = 2GB - 1byte

FAT16 for WinNT/2000, max file size = 4GB - 1byte

FAT32 for Win9x/ME/2000, max file size = 4GB - 1byte

NTFS for WinNT/2000, max file size = 16,384GB - 1byte

Linux と UNIX に関しては、サポートするファイルシステムが多様なので、リリースノートでは以下の URL を見てほしいと書いてあります。

http://www.suse.de/~aj/linux_lfs.html

せっかくですから一部をご紹介しますと、以下のようになります。

Filesystem	File Size Limit	Filesystem Size Limit
ext2 with 1 KB blocksize	16448 MB (~ 16 GB)	2048 GB (= 2 TB)
ext2 with 2 KB blocksize	256 GB	8192 GB (= 8 TB)
ext2 with 4 KB blocksize	2048 GB (= 2 TB)	16384 GB (= 16 TB)
ReiserFS 3.5	4 GB	16384 GB (= 16 TB)
ReiserFS 3.6 (as in Linux 2.4)	2 ⁶⁰ Bytes (= 1 EB)	16384 GB (= 16 TB)
XFS	2 ⁶³ Bytes (= 8 EB)	2048 GB (= 2 TB) (Linux kernel limitation)
JFS with 512 Bytes blocksize	4194304 GB (= 4 PB)	512 TB
JFS with 4KB blocksize	33554432 GB (= 32 PB)	4 PB

- DEFAULT CHARACTER SET** はデータベースのデフォルトキャラクタセットを指定します。**Firebird** は、データベース (列)・クライアントのそれぞれにキャラクタセットを指定することができ、相互に変換することが可能です。

ただし、キャラクタセットを指定しない場合のデフォルトである **NONE** で格納されたデータについては、コードの変換は行われずにそのまま格納されるため、後からキャラクタセットを指定したり、キャラクタセットを指定した列へデータを移行したりすることが出来なくなります。

日本語環境の場合、**Windows** では **SJIS_0208** を **Linux・UNIX** では **EUCJ_0208** を指定することが推奨されています。また、ユニコードを格納する **UNICODE_FSS** もありますので、これを指定しても良いでしょう。

UNICODE_FSS では最大で 1 文字当たり 3 バイトを使用するので、1 列に格納できる文字数は **VARCHAR** の最大長 **32K** バイトから逆算して約 **10K** 個の文字となります。最大 2 バイトの **SJIS_0208** や **EUCJ_0208** と比較して、その点だけ不利になりますが、これほど長い文字列を格納する場合は **Blob** を使用したほうがよいでしょう。ちなみに、1 行の最大長は **64KB** でしかありません。

※**Firebird** で使用可能なキャラクタセットとコレーションオーダーの一覧は **APPENDIX C** を参照してください。

CREATE DATABASE 文の使用例を挙げると以下のようになります。

<例 1: 単一ファイルでデータベースを作成>

```
CREATE DATABASE 'd:\db\test.gdb'
user 'SYSDBA' password 'masterkey'
page_size 16384 default character set SJIS_0208;
```

<例 2: 3 つのファイルでデータベースを作成>

```
CREATE DATABASE 'd:\db\test.gdb'
user 'SYSDBA' password 'masterkey'
page_size 16384 default character set SJIS_0208
file 'd:\db\test_2.gdb' file 'd:\db\test_3.gdb';
```

<例 3: リモートサーバーにデータベースを作成>

```
CREATE DATABASE 'svr001:d:\db\test.gdb'
user 'SYSDBA' password 'masterkey'
page_size 16384 default character set SJIS_0208
```

CREATE DOMAIN

CREATE DOMAIN 文は、新しいドメインを作成します。ドメインとは列の雛型であり、CREATE TABLE 文や ALTER TABLE 文で列のデータ型と指定できるものです。

ドメインを作成して、列のデータ型として利用することで各テーブルの列定義を簡素化することが出来、設計や変更が容易となります。

```
CREATE DOMAIN ドメイン名 [AS] <データ型>
[DEFAULT ( リテラル値
        | NULL
        | USER
        | CURRENT_USER
        | CURRENT_ROLE)]
[NOT NULL]
[CHECK <DOMAIN チェック制約>]
[COLLATE コレクションオーダー];
<データ型> =
{ SMALL INT | INTEGER | FLOAT | DOUBLE PRECISION}
[<配列定義>]
{| DATA | TIME | TIMESTAMP}[<配列定義>]
{| DECIMAL | NUMERIC} [(有効桁数, 精度)]
[<配列定義>]
{| CHAR | CHARACTER
 | CHARACTER VARYING
 | VARCHAR} [(文字数)]
[CHARACTER SET キャラクタセット名][<配列定義>]
| BLOB [SUB TYPE {数値 | サブタイプ名}]
[SEGMENT SIZE 数値]
[CHARACTER SET キャラクタセット名]
| BLOB [セグメント長, [サブタイプ名]]
<配列定義>
[[ x : ] y [, [ x : ] y ... ] ]
<DOMAIN チェック制約 > =
VALUE <演算子> 値
| VALUE [NOT] BETWEEN 値 AND 値
| VALUE [NOT] LIKE 値[ESCAPE 値 ]
| VALUE [NOT] IN ( 値 [, 値 ... ] )
| VALUE IS [NOT] NULL
| VALUE [NOT] CONTAINING 値
| VALUE [NOT] STARTING [WITH] 値
| ( <DOMAIN チェック制約 > )
| NOT <DOMAIN チェック制約 >
| <DOMAIN チェック制約 > OR <DOMAIN チェック制約 >
| <DOMAIN チェック制約 > AND <DOMAIN チェック制約 >
<演算子> =
{ = | < | > | <= | >= | !< | !> | <> | != }
```

- ドメイン名には各データベースのドメインで一意的な名前を指定します。筆者のお勧めは、DOM_ という接頭辞をつけた簡易な名前です。

ドメイン名の例：

DOM_INT4、DOM_INT64、DOM_TEL、DOM_NAME 等

- データ型には後述する、SQL データ型を指定します。NCHAR とその派生型については、省略しました。詳しくは APPENDIX A を参照してください。
- DEFAULT 句を使用して、規定値を指定することが出来ます。指定できるのは以下の要素です。

リテラル値：文字列、数値、日付のいずれか。

NULL：NULL 値

USER：現在データベースに接続しているユーザー名

CURRENT_USER：同上

CURRENT_ROLE：現在データベースに接続しているユーザーの SQL ロール名

!!FB!!CURRENT_USER と CURRENT_ROLE は Firebird で新たに導入された、コンテキスト変数です。CURRENT_USER については、USER と同義ですが、CURRENT_ROLE は、SQL ロールが存在しない場合には、NONE が指定されるようになっています。InterBase ではエラーになりますが、Firebird ではエラーになりません。

- リテラル値に 'TODAY' を指定すると今日の日付が、'NOW' を指定すると現在の時刻が挿入されます。また、'YESTERDAY' と

'TOMORROW' もそれぞれ、昨日と明日の日付が挿入されます。また、`current_date` と `current_time`、`current_timestamp` も使用することが出来ます。

この機能については、ここで見かけますが、InterBase のマニュアルにも記載がなく、いわゆる **Undocumented** な機能となっています。

使用した結果は以下のようになります。次のようなテーブルに対して、それぞれの値を `INSERT` した結果をみます。ダイレクト 3 のデータベースでは、`DATE` 型は時間情報を持ちません。

```
T_TEST (ID INTEGER,
        _DATE DATE,
        _TIMESTAMP TIMESTAMP)
```

```
INSERT INTO T_TEST(1,'NOW','NOW');
INSERT INTO T_TEST(2,'TODAY','TODAY');
```

```
SELECT * FROM T_TEST;
```

```
ID      _DATE      _TIMESTAMP
-----
1       2003/1/24  2003/1/24  1:00:00
2       2003/1/24  2003/1/24
```

- `CHECK` 句では、列制約を定義します。`CREATE TABLE` 文で `CHECK` 制約を指定する場合には、テーブル毎の定義となるので、各列の名前を使用して制約を定義するのですが、ドメインで `CHECK` 制約を指定する場合には、ドメイン名ではなく「**VALUE**」というキーワードを使用します。

`CHECK` 制約で使用できる演算子や構文は前記したとおりです。ドメインの `CHECK` 制約では、他のドメインを参照したり、列名を指定したりすることはできません。また、ドメインの `CHECK` 制約は 1 つだけしか指定することができません。

ここで指定した `CHECK` 制約を、列の定義で無効にすることはできませんので注意してください。テーブルの作成時には、`CHECK` 制約を追加することが出来るだけです。

- `CHECK` 制約で使用される `LIKE` 句は `SELECT` 文でもよく使用されますが、`ESCAPE` 句の使用方法はなかなかわかりづらいものがあります。

Firebird の SQL では、標準 SQL に準拠して、`LIKE` 句のワイルドカードとして「%」と「_」を使用することが出来ます。「%」は全ての文字列に、「_」は任意の 1 文字に相当します。正規表現に置き換えると、「%」は「.+」に、「_」は「.»に相当します。「%」は 0 文字以上の任意の文字列にマッチするので、空文字であってもマッチします。しかし、**Null** にはマッチしません。

さて、ここで「%」と「_」の入った文字列に正確にマッチさせたい場合を考えます。「%」と「_」をワイルドカードとしてではなく、単なる文字として扱う場合に `ESCAPE` 句を使用します。

```
SELECT field FROM table WHERE field LIKE '__\%' ESCAPE '\';
```

この `SELECT` 文は、`'ab%c'` や `'de%f'` とマッチします。気をつけるのは、`field` の長さが文字列よりも長い場合、空白が付加されてしまうので、`UDF` の `rtrim()` 等を使用する必要があるということです。

```
SELECT field FROM table WHERE rtrim(field) LIKE '__\%' ESCAPE '\';
```

PostgreSQL では、「\」がデフォルトのエスケープキャラクターなのですが、Firebird の場合は指定がない限りエスケープ出来ないようです。

CREATE GENERATOR

CREATE GENERATOR 文はデータベースにジェネレーターを作成します。ジェネレーターは、一連の一意な整数値を必要とする場合に利用するもので、トリガーと組み合わせてオートナンバー型を実現すること等に使用されています。ジェネレーターのデフォルト値は **0** に設定されます。

CREATE GENERATOR ジェネレーター名

- ジェネレーター名はデータベースで一意でなくてはなりません。他のオブジェクトとは名前が重複しても構いませんが、わかりにくいので、**GEN_hogehoge** 等と命名することをお勧めします。
- ジェネレーターの値は **GEN_ID()** 関数によって取り出されますが、複数のトランザクションから取り出す場合にも値が重複することはありません。もっとも、**gen_id()** 関数は、ジェネレーター名と増分値を引数として取りますので、増分値を **0** にすれば同じ値だけを取り出すことも出来ます。
- ジェネレーターの値は **SET GENERATOR** 文によって再設定することが可能です。
- 増分値にマイナスを設定することも出来るので、**SET GENERATOR** 文を使用しなくても、**GENERATOR** を指定した値に再設定することも可能です。この方法はトリガー/ストアプロシージャーでも使えます。

DROP GENERATOR

!!FB!! Firebird では、**DROP GENERATOR** 文がサポートされました。InterBase ではジェネレーターを削除するためにはシステムテーブルに対して直接 SQL 文を発行しなくてはなりませんでした。Firebird ではより安全にジェネレーターを **DROP** することが出来ます。InterBase では、ジェネレーターを **DROP** するためには以下のようにしなくてはなりませんでした。

```
DELETE FROM RDBSGENERATOR  
WHERE RDBSGENERATOR_NAME = ジェネレーター名;
```

Firebird では以下のように記述します。

```
DROP GENERATOR ジェネレーター名;
```

GEN_ID()

GEN_ID()関数はジェネレーターから値を取り出すために使用されます。トリガー、プロシージャ、SQL文で使用することが出来ます。

GEN_ID(ジェネレーター名, 増分値)

- ジェネレーター名には既に作成されているジェネレーターを指定します。
- 増分値には $-(2^{63}) \sim 2^{63}-1$ を指定することが出来ます。0を指定すると同じ値を取り出し続けることが出来ます。
- ジェネレーターから値を一つ取り出すためには、以下のようなSQL文を使用します。

```
SELECT GEN_ID(ジェネレーター名, 1)
FROM rdb$database
```

rdb\$databaseは、システムテーブルの一つで、データベースのデフォルトキャラクタセット等が格納されています。このテーブルは、データが1行しかないことになっています。SELECT文は、テーブルの各行に対してFETCH動作を行うので、1行しかないテーブルに対してSELECTを行うと、一つだけ値を取り出すことが出来ます。FROM句にはなんらかのテーブルを指定すればいいので、この方法はトリック的なものです。もちろん、一行しかないテーブルを自分で用意してもOKです。

- ストアドプロシージャを使用してジェネレータを取得するには以下のようにします。

```
SET TERM !! ;
CREATE PROCEDURE SP_GEN_hoge
RETURNS (int_hoge INTEGER)
AS
BEGIN
    int_hoge = GEN_ID(GEN_hoge);
END;
```

- ジェネレーターをトリガーと組み合わせるとオートナンバー型を実現するためには、以下のようにします。

```
SET TERM !! ;
CREATE TRIGGER TRG_SET_hoge_ID FOR T_hoge
BEFORE INSERT POSITION 0 AS
BEGIN
    NEW hoge_ID = GEN_ID(GEN_hoge_ID, 1);
END !!
SET TERM ; !!
```

SET GENERATOR

SET GENERATOR 文はジェネレーターに任意の値を設定するために使用します。一連の一意な値を取得することがジェネレーターを利用する最大の目的ですから、SET GENERATOR 文はデータを頻繁にクリアして番号を振りなおす場合や、一定の範囲で循環した一連の番号を得たい場合などに利用することになります。

SET GENERATOR ジェネレーター名 TO 整数値

- GEN_ID () 関数は、増分値を加算した値を返すので、ジェネレーターで 100 からの連続した値を取り出す場合、SET GENERATOR 文でジェネレーターを 99 に設定して、GEN_ID () の増分値を 1 にします。
あるいは、0 からの連続した値を取り出す場合には、-1 に設定して、GEN_ID () 関数の増分値を 1 に設定します。

```
CREATE GENERATOR GEN_HOGE;
SET GENERATOR GEN_HOGE TO -1;
SELECT GEN_ID(GEN_HOGE, 1) FROM rdb$database;
      GEN_ID
=====
              0
```

- SET GENERATOR 文はストアードプロシージャ／トリガーで使用することはできませんので注意してください。

CREATE TABLE

CREATE TABLE 文は、データベースを設計し実装するための最大のポイントとなるコマンドです。当然、オプションも多義にわたっています。

```
CREATE TABLE テーブル名 (
<列定義>[, <列定義> | <テーブル制約>, . . .]);
<列定義> = 列名
  { <データ型>
  | COMPUTED [BY] (<計算式?>)
  | ドメイン名}
  [DEFAULT { リテラル値
            | NULL
            | USER
            | CURRENT USER
            | CURRENT ROLE}]
  [NOT NULL]
  [<列制約>]
  [COLLATE <コレーションオーダー>]
<データ型> =
  { SMALL INT | INTEGER | FLOAT | DOUBLE PRECISION}|
  { DATA | TIME | TIMESTAMP}[<配列定義>]|
  { DECIMAL | NUMERIC}
  [(有効桁数, 精度)][<配列定義>]|
  { CHAR | CHARACTER
  | CHARACTER VARYING
  | VARCHAR}
  [(文字数)][<配列定義>]
  [CHARACTER SET キャラクタセット名]|
  {BLOB [SUB TYPE {数値 | サブタイプ名}]
  [SEGMENT SIZE 数値]
  [CHARACTER SET キャラクタセット名]|
  BLOB [セグメント長, [サブタイプ名]}
<配列定義> =
  [[ x : ] y [, [ x : ] y ... ] ]
<計算式> =
  単一の値になる、有効な SQL 式
<列制約> = [CONSTRAINT 制約名]
  { UNIQUE
  | PRIMARY KEY
  | REFERENCES 別テーブル名[(列名[, 列名 . . .])]}
  [ON DELETE { NO ACTION
              | CASCADE
              | SET DEFAULT
              | SET NULL}]
  [ON UPDATE { NO ACTION
              | CASCADE
              | SET DEFAULT
              | SET NULL}]
  | CHECK (<検索条件>)}
<テーブル制約> = [CONSTRAINT 制約名]
  {{PRIMARY KEY | UNIQUE} (列名[, 列名 . . .])
  | FOREIGN KEY (列名[, 列名 . . .])
  REFERENCES 別テーブル名[(列名[, 列名 . . .])]}
  [ON DELETE { NO ACTION
              | CASCADE
              | SET DEFAULT
              | SET NULL}]
  [ON UPDATE { NO ACTION
              | CASCADE
              | SET DEFAULT
              | SET NULL}]
  | CHECK (<検索条件>)}
<検索条件> =
  <値> <演算子> {<値> | <単項 SELECT 文>
  | <値> [NOT] BETWEEN <値> AND <値>
  | <値> [NOT] LIKE <値>[ESCAPE <値> ]
  | <値> [NOT] IN ( <値> [, <値> ... ]
  | <単項 SELECT 文>)
```

```

| <値> IS [NOT] NULL
| <値> {>= | <=} <値>
| <値> [NOT] {= | < | >} <値>
| {ALL | SOME | ANY} (単列 SELECT 文)
| EXISTS (SELECT 表現)
| SINGULAR (SELECT 表現)
| <値> [NOT] CONTAINING <値>
| <値> [NOT] STARTING [WITH] <値>
| ( <検索条件>)
| NOT <検索条件>
| <検索条件> OR <検索条件>
| <検索条件> AND <検索条件>
<値> = {列名 [<配列定義>]
| <定数> | <計算式> | <内部関数>
| UDF ([<値> [, <値>...]])
| NULL | USER | CURRENT_USER | CURRENT_ROLE
| RDBSDB_KEY}
[COLLATE コレレーションオーダー]
<定数> = 数値
| '文字列'
| キャラクタセット名 '文字列'
<内部関数> = COUNT (* | [ALL] <値> | DISTINCT <値>)
| SUM ([ALL] <値> | DISTINCT <値>)
| AVG ([ALL] <値> | DISTINCT <値>)
| MAX ([ALL] <値> | DISTINCT <値>)
| MN ([ALL] <値> | DISTINCT <値>)
| CAST (<値> AS <データ型>)
| UPPER (<値>)
| GEN_ID (ジェネレーター名, 数値)
<演算子> =
{ = | < | > | <= | >= | !< | !> | <> | != }
<単項 SELECT 文> =
1 行・1 列の値を返す SELECT 文
<単列 SELECT 文> =
1 列のみで、0 行以上の行を返す SELECT 文
<SELECT 表現> =
0 行以上の行を返す、<値>のリストに対する SELECT 文

```

- ・ 列定義はドメインの定義と基本的に同じです。CREATE TABLE 文の列定義では、データ型にドメインを指定することができます。

例：

```

<通常 of データ型指定>
CREATE TABLE TableA (
  Col1 INTEGER NOT NULL PRIMARY KEY,
  Col2 CHAR(10) CHARACTER SET ASCII);

```

```

<ドメインによる指定>
CREATE DOMAIN DOM_INT4
  INTEGER NOT NULL;
CREATE DOMAIN DOM_SHTNAME
  CHAR(10) CHARACTER SET ASCII;
CREATE TABLE TableB(
  Col1 DOM_INT4 PRIMARY KEY,
  Col2 DOM_SHTNAME);

```

- ・ 列制約は UNIQUE 制約、PRIMARY KEY 制約、REFERENCES 制約を定義することが出来ます。UNIQUE 制約と PRIMARY KEY 制約は NOT NULL 制約と一緒に使用しなくてはなりません。UNIQUE キーに一つだけ NULL を許す RDBMS もあるので、注意してください。UNIQUE 制約、PRIMARY KEY 制約ともに、他のテーブルにある FOREIGN KEY 制約・REFERENCES 制約から参照することが出来ます。逆にいえば、UNIQUE 制約、PRIMARY KEY 制約をつけないと FOREIGN KEY として指定することができません。UNIQUE 制約と PRIMARY KEY 制約の違いは、PRIMARY KEY が一つだけしか設定できないが、UNIQUE 制約は複数設定できるということと、PRIMARY KEY が FOREIGN KEY 及び REFERENCES 制約のデフォルトの参照先になるということです。
- ・ REFERENCES 制約は ON [DELETE | UPDATE] 句によって、参照先のテーブルの値を連動して変更することができます。NO ACTION はデフォルトの設定で、何もしません。CASCADE は、DELETE に対しては参照先を連鎖削除し、UPDATE に対しては連鎖更新します。SET DEFAULT は DEFAULT 句で指定された値にセットします。SET NULL は NULL 値をセットします。

例：

REFERENCES 制約が設定された 2 つのテーブルを用意します。

```
CREATE TABLE TableA (  
  Col1 INTEGER NOT NULL PRIMARY KEY);
```

```
CREATE TABLE TableB (  
  Col1 INTEGER  
  REFERENCES TableA (Col1)  
  ON UPDATE CASCADE  
  ON DELETE CASCADE);
```

上記のテーブルに以下のようなデータを追加します。

TableA Col1	TableB Col1
-----	-----
1	1
2	1
3	2
	3
	3

このような 2 つのテーブルで、TableA で Col1=1 の行を Col1=4 に UPDATE します。

```
UPDATE TableA SET Col1 = 4  
WHERE Col1 = 1;
```

TableA Col1	TableB Col1
-----	-----
4	4
2	4
3	2
	3
	3

さらに、TableA で Col1=2 の行を DELETE します。

```
DELETE FROM TableA WHERE Col1 = 2;
```

TableA Col1	TableB Col1
-----	-----
4	4
3	4
	3
	3

- 各データ型の指定では配列型を指定することも出来ます。配列型の使用には直接 API を利用するか、IBObjects などの配列型に対応したコンポーネントを利用します。ISQL 等を利用して SQL 文から配列型を扱うことはできません。

```
CREATE TABLE TableA(  
  ID INTEGER NOT NULL PRIMARY KEY,  
  ARRAY1 INTEGER [0:1] NOT NULL);
```

- COMPUTED BY 句は計算項目を定義します。計算項目とは、ある種の VIEW ともいえる機能で、テーブルそのものに式によって計算されたフィールドを作成してしまう機能です。

例：

```
CREATE TABLE TableA(  
  ID INTEGER NOT NULL PRIMARY KEY,  
  FirstName VARCHAR(10) NOT NULL,  
  LastName VARCHAR(10) NOT NULL,  
  FullName COMPUTED BY  
  (LASTNAME|| ', ' || FIRSTNAME)  
);
```

```
INSERT INTO TableA VALUES
```

```
(1, 'Tsutomu', 'Hayashi');
INSERT INTO TableA VALUES
(2, 'Kouichi', 'Takeda');
```

この例では、別個に入力された **FirstName** と **LastName** を結合して **FullName** 列を作成しています。**COMPUTED BY** 句のところ、**LASTNAME** と **FIRSTNAME** を大文字にしているのは、**ISQL** のクセのためです。計算項目ではこの他、四則演算と **CAST()** 関数、**UPPER()** 関数を使用できます。内部関数のうち複数行を対象とした集計関数と、**EXTRACT** 関数は使用できません。

- テーブル制約は（複数の）列に適用する制約を定義します。列制約と同様に、**PRIMARY KEY** 制約と **UNIQUE** 制約、**REFERENCES** 制約を指定することが出来ます。**REFERENCES** 制約は列制約と違い、**FOREIGN KEY** 句を使って **REFERENCES** 制約を定義する（複数の）列を指定します。また、**CHECK** 制約を指定することも出来ます。

例:

```
CREATE TABLE TableA (
  ID1 INTEGER NOT NULL,
  ID2 INTEGER NOT NULL,
  FirstName VARCHAR(10) NOT NULL,
  LastName VARCHAR(10) NOT NULL,
  PRIMARY KEY (ID1, ID2),
  UNIQUE (FirstName, LastName)
);
```

```
CREATE TABLE TableB (
  ID1 INTEGER NOT NULL,
  ID2 INTEGER NOT NULL,
  ADDRESS VARCHAR(100),
  TEL VARCHAR(15),
  FOREIGN KEY (ID1, ID2)
  REFERENCES TableA (ID1, ID2)
);
```

上記の例では、TableA の ID1・ID2 の組み合わせが主キーとなり、FirstName と LastName もユニークなので同姓同名のデータを排除します。

また、TableB の ID1・ID2 は TableA の ID1・ID2 への外部キーとなっていますので、TableA に存在しない組み合わせを TableB に INSERT したり、UPDATE したりすることはできません。

RECREATE TABLE

!!FB!! RECREATE TABLE 文は、すでに存在しているテーブルを DROP することなく、再作成します。文法としては、CREATE TABLE 文と同一です。

RECREATE TABLE 文を実行すると、既存のテーブルのデータはすべて失われるので注意してください。また、他のテーブルやストアドプロシージャ等から参照されている場合は、エラーとなり RECREATE TABLE 文を実行できません。

CREATE VIEW

CREATE VIEW文は、一つ以上のテーブルから新しいビューを作成します。ビューは元になるテーブルに対する **SELECT** 文などによって構成されます。データベースにはビューの定義のみが格納され、データは必要に応じて元テーブルから抽出されます。なお、**Firebird** は単純なビューに対しては、直接の更新が可能となっています。また、より複雑なビューに対してはトリガーを設定することで更新に対応することが可能です。

```
CREATE VIEW ビュー名
[( ビュー列名 [, ビュー列名 ...])]
AS <SELECT 文>
[WITH CHECK OPTION];
```

- ビュー列名の数は、対応する **SELECT** 文の列数と一致していません。あるいは、全く指定しないかのどちらかとなります。

例：

```
CREATE TABLE TableA (
  ID INTEGER NOT NULL,
  FirstName VARCHAR(10) NOT NULL,
  LastName VARCHAR(10) NOT NULL,
  PRIMARY KEY (ID),
  UNIQUE (FirstName, LastName)
);
```

```
CREATE VIEW V_TableA
(ID, FULLNAME) AS
SELECT ID, FIRSTNAME || ', ' || LASTNAME FROM TableA;
```

```
SELECT * FROM TABLEA;
```

```
          ID FIRSTNAME  LASTNAME
=====
          1 Tsutomu    Hayashi
          2 Kouichi   Takeda
```

```
SELECT * FROM V_TABLEA;
          ID FULLNAME
=====
```

```
          1 Tsutomu, Hayashi
          2 Kouichi, Takeda
```

- **SELECT** 文で **ORDER BY** 句を使用することはできません。**VIEW** に対する **SELECT** 文で並び順を指定してください。
- 列を指定しないで **SELECT *** を使用したときには、列の並び順は元になっているテーブルの並び順と同様になります。
- **WITH CHECK OPTION** 句を指定すると、ビューの行を追加または更新する前に、**WHERE** 句で指定された条件によってチェックされます。条件に適合しない行は挿入や更新をすることができません。**WITH CHECK OPTION** 句は、読み取り専用ビューには使用できません。

```
CREATE TABLE TableA (
  ID INTEGER NOT NULL PRIMARY KEY,
  FirstName Varchar(10),
  LastName Varchar(10)
);
```

```
CREATE VIEW V_TableA AS
SELECT ID, FirstName FROM TableA
WHERE ID < 10;
```

```
INSERT INTO V_TableA Values (11, 'Jelly');
```

この場合、行の挿入は成功しますが、**V_TableA** を **SELECT** しても挿入したデータは表示されません。

```
SELECT * FROM V_TableA;
          ID FIRSTNAME
=====
```

```
          1 Tsutomu
          2 Kouichi
```

```

SELECT * FROM TableA;
      ID FIRSTNAME      LASTNAME
=====
      1 Tsutomu        <null>
      2 Kouichi        <null>
     11 Jelly         <null>

```

テーブルには行が挿入されています。

ビューに WITH CHECK OPTION を指定するとビューで参照できないデータの挿入は禁じられます。

```

CREATE VIEW V_TableA2 AS
SELECT ID, FirstName FROM TableA
WHERE ID < 10
WITH CHECK OPTION;

INSERT INTO V_TableA2 Values (12, 'Potesi');

```

Statement failed, SQLCODE = -297

Operation violates CHECK constraint on view or table V_TABLEA2

- ・ ビューの元になるテーブルとして結果セットを返すストアド・プロシージャを含めることはできませんので注意してください。
- ・ ビューが更新可能となるのは以下の場合です。

- ①一つのテーブルの一部（または全て）の場合
- ②更新可能なビューの一部（または全て）の場合
- ③ビューの定義から除外された元テーブルの列が、どれも NULL 値を格納できる場合
- ④ビューの SELECT 文に以下の要素が含まれていない場合。

サブクエリー

DISTINCT 句

HAVING 句

集合関数

JOIN で結合されたテーブル

ユーザー定義関数

ストアドプロシージャ

UNION 句で結合されたテーブル

列同士の演算によって指定された列

例：

```

CREATE TABLE TableA (
ID INTEGER NOT NULL PRIMARY KEY,
FirstName Varchar(10),
LastName Varchar(10)
);

```

```

CREATE VIEW V_TableA AS
SELECT ID, FirstName FROM TableA;

```

```

INSERT INTO V_TableA Values (1, 'Tsutomu');
INSERT INTO V_TableA Values (2, 'Kouichi');

```

```

SELECT * FROM V_TableA;
      ID FIRSTNAME
=====
      1 Tsutomu
      2 Kouichi

```

```

SELECT * FROM TableA;
      ID FIRSTNAME      LASTNAME
=====
      1 Tsutomu        <null>
      2 Kouichi        <null>

```

この例では、元になるテーブルの一部をビューとして取り出して、行の挿入を行ってみました。INSERT 文は実行され、データが挿入されていることが確認できます。また、ビューから除外された列に NULL 値が格納されていることがわかります。

CREATE EXCEPTION

CREATE EXCEPTION 文は、トリガー・ストアプロシージャで使用されるユーザー定義の例外と例外メッセージを作成します。

CREATE EXCEPTION 例外名 '例外メッセージ';

- ・ 例外名はデータベースで一意的な名前とします。メッセージは 78 文字までです。
- ・ 例外はデータベースでグローバルに使用することが出来るので、すべてのトリガーとストアプロシージャで処理を共通化できます。
- ・ トリガーまたはストアプロシージャで例外が発行された場合、以下の動作を行います。
 - 1) 例外を発行したトリガーまたはストアプロシージャを終了して、実行を中断します。
 - 2) 呼び出し元のアプリケーションに対して例外メッセージを返します。isql の場合、出力先を変更しなければ、画面に表示されません。
- ・ 例外はトリガーまたはストアプロシージャの中で WHEN 文でトラップして処理するか、プログラマによって任意の条件で発行することが出来ます。

例：

```
CREATE EXCEPTION DUPE_ID 'Duplication ID Error';
```

```
CREATE PROCEDURE SP_hogehoge
```

```
AS
```

```
BEGIN
```

```
    . . .
```

```
    WHEN SQLCODE -803 DO
```

```
        EXCEPTION DUPE_ID;
```

```
    . . .
```

```
END
```

- ・ 例外メッセージに日本語を使用することも出来ますので、よりユーザーに理解しやすいエラーメッセージを用意することが可能です。

CREATE INDEX

CREATE INDEX 文は、テーブルの一つ以上の列に対して、対応するインデックスを作成します。インデックスを作成するとデータの検索速度を上げることが出来ます。**SELECT** 文の **WHERE** 句で指定する列にインデックスを作成すると、パフォーマンスを向上させることが出来ます。

しかし、インデックスは挿入・更新については、インデックス更新のオーバーヘッドの文だけ遅くなります。大量のデータ挿入時などには、いったん **ALTER INDEX** 文を使用してインデックスの更新を停止するなどの措置を講じたほうが良い場合もあります。

ALTER INDEX 文を使用して、インデックスを停止した後に使用可能にするとインデックスが再構築されます。同様に **SET STATISTICS** 文を使用すると、指定したインデックスの選択性を再構築することが出来ます。

CREATE [UNIQUE] [ASC[ENDING] | DESC[ENDING]] INDEX インデックス名 **ON** テーブル名 {列名 [, 列名...]};

- ・ **BLOB** 型及び配列型が指定されている列にはインデックスを作成することはできません。
- ・ すでに重複した値が格納されている列と、**NULL** 値が格納されている列には **UNIQUE** を指定することはできません。
- ・ **ASC[ENDING]** (昇順: A→Z) と **DESC[ENDING]** (降順: Z→A) は、インデックスのソート順を指定するために使用します。**SELECT** 文で **ORDER BY** 句に指定する順序でインデックスを作成しておくことで、データの抽出を高速化することができます。昇順と降順の両方のインデックスを作成しておくことで、どちらのソート順にも対応するようにすることも可能です。
- ・ **Firebird** では、必要なインデックスはオプティマイザが自動的に選択します。**SELECT** 文の **PLAN** オプションを使用して、手動でインデックスを指定することも可能ですが、**Firebird** は高度なオプティマイザを備えているので、通常の使用ではプログラマがこれを使用することはありません。
Firebird がインデックスを利用するのは以下の場合です。
 - a) 主キーと外部キー
 - b) テーブルの **JOIN** で使用される結合キー
 - c) **DISTINCT** と **GROUP BY**、**ORDER BY** 句で指定されたソートキー
 - d) **WHERE** 句で指定された検索キー
- ・ 主キーと外部キーにはシステムが自動的にインデックスを作成するので、プログラマは **JOIN** キーと **ORDER BY** で使用されるソートキーにインデックスを作成するのが一般的です。
- ・ **Firebird** は以下の場合には指定された列にインデックスが合っても使用しませんので、注意してください。
 - a) **CONTAINING**、**LIKE**、及び<>の不等式処理で指定された列
 - b) **COUNT()**、**SUM()** などの集計関数で指定された列
 - c) **UPPER()** などのその他の関数で指定された列

CREATE ROLE

CREATE ROLE 文は SQL ロールを作成します。SQL ロールには、ユーザーと同じように **GRANT** 文を使用して各権限を付与することができます。また、**REVOKE** 文を使用して権限を取り消すことができます。

作成した SQL ロールを使用して、各ユーザーに指定の権限を付与することが出来ます。こうすることで、複雑な権限の組み合わせを単一のロールとして表現することができ、セキュリティ管理上の煩雑さを軽減することができます。

CREATE ROLE ロール名;

- ・ ロール名はデータベース内の各ロールにおいて一意でなければなりません。

例 :

```
CREATE ROLE R_admin;
```

```
GRANT ALL ON TableA TO R_admin;
```

```
GRANT R_admin to tonneko, potesi, makineko;
```

CREATE SHADOW

CREATE SHADOW 文はデータベースと同一のコピーとなるシャドウファイルを作成します。データベースのシャドウは、元になったデータベースとは別のファイルシステム上に配置することが可能で、そのためファイルシステムの障害によるデータベースの破壊を防ぎます。

シャドウは **Firebird** の原型となった **DEC** の **jrd** というデータベースの開発時に、**Jim Starkey** 氏によって考案されました。シャドウは **Firebird** の ツーフーズ・コミット機能によって、常に元になるデータベースとの整合性が保たれています。これを可能にするための **Multi Generation Architecture** によって、**Firebird** は高度なトランザクション管理機能を手に入れることができたのです。

シャドウ機能については、今日ではほとんど使用されていないようです。**UPS** や **RAID** の使用によってデータベースの物理的な障害・消失が防げるようになってきたためですが、そうした手段を講じることができない場合に手軽なデータベースの保護手段として活用しても良いのではないのでしょうか。

CREATE SHADOW シャドウセット番号

```
[AUTO | MANUAL] [CONDITIONAL]
'<ファイル名>' [LENGTH [=] 整数値 [PAGE[S]]]
[ <二次ファイル情報>];
<二次ファイル情報 > = FILE 'ファイル名'
[ <ファイル情報> [ <二次ファイル情報 >]
<ファイル情報> = LENGTH [=] 整数値 [PAGE[S]]
| STARTING [AT [PAGE]] 整数値 <ファイル情報>
```

- ・ シャドウセット番号は、シャドウの識別に用いられる一意の整数値を指定します。
- ・ **AUTO** と **MANUAL** オプションは、シャドウファイルが使用不可能な場合の、データベースへのアクセスを指定します。デフォルトは **AUTO** です。
- ・ **AUTO** モードでは、データベースへの接続とアクセスは正常に行えます。また、シャドウへの参照が削除され、シャドウファイルへの接続が切断されます。
- ・ **MANUAL** モードでは、シャドウファイルが使用可能となるか、シャドウへの全ての参照が削除されるまで、データベースへの接続とアクセスは行えません。
- ・ **CONDITIONAL** オプションを指定すると、データベースファイルに障害があり、シャドウファイルをアクティブにしてデータベースと置き換えた場合や、シャドウファイルに障害があった場合に、**Firebird** は新しいシャドウファイルを自動的に作成します。(筆者が試した限りでは、新しいシャドウファイルは作成されませんでした。)
- ・ データベースが破損した場合に、シャドウファイルをアクティブにしてデータベースと置き換えるには **gfix.exe** を使用します。

例：

```
gfix -a -user SYSDBA -password hoge hoge test.shd
```

シャドウをアクティブにした後で、シャドウファイルの名称を元のデータベース名に変更しておくことが推奨されています。

- ・ **MANUAL** モードでシャドウファイルが破損・消失した場合にシャドウファイルへの参照をデータベースから削除するには、同様に **gfix.exe** を使用します。

例：

```
gfix -kill -user SYSDBA -password hoge hoge test.gdb
```

- ・ シャドウファイルの拡張子は、通常 **.shd** とするようです。他の拡張子であっても、また拡張子がなくても問題はありません。
- ・ **LENGTH** オプションや **PAGES** オプション等については、**CREATE DATABASE** 文と同様です。ファイルサイズを制限した場合には、最後のファイルが **OS** の制限いっぱいまで拡張されます。
- ・ データベースが大きくなり、シャドウにファイルを追加しなくてはならない場合には、新しいマルチファイルのシャドウを追加してから、以前のシャドウを **DROP SHADOW** 文で削除するようにしてください。一つのデータベースに複数のシャドウを設定することができるので、削除してから追加という手順でなくても大丈夫ですが、ディスクスペースには注意してください。

Firebird SQL - ALTER 系コマンド

ALTER 系コマンドは、CREATE 系コマンドによって作成された各データベースオブジェクトの構造や状態を変更するために使用します。その性質上、頻繁に使用するものではありませんが、必要に応じて利用することになります。

ALTER DATABASE

ALTER DATABASE 文は、現在のデータベースに 2 次ファイルを追加するために使用します。それ以外の変更は行うことが出来ません。デフォルトキャラクタセットの変更や、PAGESIZE の変更などは gbak などのユーティリティを使用して、バックアップ・リストアをする際に変更することが出来ます。

```
ALTER { DATABASE | SCHIMA }
  ADD <二次ファイル情報>
<二次ファイル情報 > = FILE '<ファイル名>'
  [ <ファイル情報> ] [ <二次ファイル情報> ]
<ファイル情報> = LENGTH [=] 整数値 [PAGE[S]]
  | STARTING [AT [PAGE]] 整数値 <ファイル情報>
```

例：

```
ALTER DATABASE ADD FILE 'Test2. gdb'
  STARTING AT PAGE 8001 LENGTH 8000
  ADD FILE 'Test3. gdb';
```

ALTER DOMAIN

ALTER DOMAIN 文は、指定したドメインの定義を変更します。既存のドメインの任意の項目を変更することが可能ですが、NOT LNUL 制約だけは変更することが出来ません。NOT NULL 制約を変更する場合は、いったんドメインを削除してから再作成するしかありません。

ALTER DOMAIN

```
{ドメイン名 | 旧ドメイン名 TO 新ドメイン名}
SET DEFAULT ( リテラル値
  | NULL
  | USER
  | CURRENT_USER
  | CURRENT_ROLE)
| DROP DEFAULT
| ADD [CONSTRAINT] CHECK <DOMAIN チェック制約>
| DROP CONSTRAINT
| 新列名
| TYPE <データ型>;
<データ型> =
{ SMALL INT | INTEGER | FLOAT | DOUBLE PRECISION}
  [<配列定義>]
| { DATA | TIME | TIMESTAMP } [<配列定義>]
| { DECIMAL | NUMERIC } [(有効桁数, 精度)]
  [<配列定義>]
| { CHAR | CHARACTER
  | CHARACTER VARYING
  | VARCHAR } [(文字数)]
  [CHARACTER SET キャラクタセット名] [<配列定義>]
| BLOB [SUB TYPE {数値 | サブタイプ名}]
  [SEGMENT SIZE 数値]
  [CHARACTER SET キャラクタセット名]
| BLOB [セグメント長, [サブタイプ名]]
<配列定義>
[[ [ x : ] y [, [ x : ] y ... ] ]
<DOMAIN チェック制約 > =
VALUE <演算子> 値
| VALUE [NOT] BETWEEN 値 AND 値
| VALUE [NOT] LIKE 値 [ESCAPE 値 ]
| VALUE [NOT] IN ( 値 [, 値 ... ] )
| VALUE IS [NOT] NULL
| VALUE [NOT] CONTAINING 値
| VALUE [NOT] STARTING [WITH] 値
| ( <DOMAIN チェック制約 > )
| NOT <DOMAIN チェック制約 >
| <DOMAIN チェック制約 > OR <DOMAIN チェック制約 >
| <DOMAIN チェック制約 > AND <DOMAIN チェック制約 >
<演算子> =
{ = | < | > | <= | >= | !< | !> | <> | != }
```

例：

```
CREATE DOMAIN ZIPCODE AS CHAR(8);
ALTER DOMAIN ZIPCODE SET DEFAULT NULL;
```

ALTER TABLE

ALTER TABLE 文は、指定されたテーブルの列や参照整合性制約を追加・削除・変更するために使用します。1つの ALTER TABLE 文で複数の変更を行うことが可能です。

```
ALTER TABLE テーブル名 <変更操作> [, <変更操作>]
<変更操作> = ADD <列定義>
| ADD <テーブル制約>
| ALTER [COLUMN] 列名 <列変更操作>
| DROP 列名
| DROP CONSTRAINT 制約名
<列変更操作> = TO 新列名
| TYPE 新列データ型
| POSITION 新列位置
<列定義> = 列名 { <データ型>
| COMPUTED [BY] (<計算式>)
| ドメイン名}
[DEFAULT { リテラル値
| NULL
| USER
| CURRENT USER
| CURRENT ROLE}]
[NOT NULL]
[<列制約>]
[COLLATE <コレーションオーダー>]
<データ型> =
{ SMALL INT | INTEGER | FLOAT | DOUBLE PRECISION}
{ DATA | TIME | TIMESTAMP}[<配列定義>]
{ DECIMAL | NUMERIC} [(有効桁数, 精度)][<配列定義>]
{ CHAR | CHARACTER
| CHARACTER VARYING
| VARCHAR}
[(文字数)][<配列定義>]
[CHARACTER SET キャラクタセット名]
{BLOB [SUB TYPE {数値 | サブタイプ名}]
[SEGMENT SIZE 数値]
[CHARACTER SET キャラクタセット名]}
BLOB [セグメント長, [サブタイプ名]]
<配列定義> =
[[ x :] y [, [ x :] y ... ] ]
<計算式> =
単一の値になる、有効な SQL 式
<列制約> = [CONSTRAINT 制約名]
{ UNIQUE
| PRIMARY KEY
| REFERENCES 別テーブル名 [(列名 [, 列名 . . .])]
[ON DELETE { NO ACTION
| CASCADE
| SET DEFAULT
| SET NULL}]
[ON UPDATE { NO ACTION
| CASCADE
| SET DEFAULT
| SET NULL}]
| CHECK (<検索条件>)}
<テーブル制約> = [CONSTRAINT 制約名]
{{PRIMARY KEY | UNIQUE} (列名 [, 列名 . . .])
| FOREIGN KEY (列名 [, 列名 . . .])
REFERENCES 別テーブル名 [(列名 [, 列名 . . .])]
[ON DELETE { NO ACTION
| CASCADE
| SET DEFAULT
| SET NULL}]
[ON UPDATE { NO ACTION
```

```

| CASCADE
| SET DEFAULT
| SET NULL}]
| CHECK (<検索条件>)}
<検索条件> =
<値> <演算子> {<値> | <単項 SELECT 文>
| <値> [NOT] BETWEEN <値> AND <値>
| <値> [NOT] LIKE <値>[ESCAPE <値> ]
| <値> [NOT] IN ( <値> [, <値> ...]
| <単列 SELECT 文>)
| <値> IS [NOT] NULL
| <値> {>= | <=} <値>
| <値> [NOT] {= | < | >} <値>
| {ALL | SOME | ANY} (単列 SELECT 文)
| EXISTS (SELECT 表現)
| SINGULAR (SELECT 表現)
| <値> [NOT] CONTAINING <値>
| <値> [NOT] STARTING [WITH] <値>
| ( <検索条件>)
| NOT <検索条件>
| <検索条件> OR <検索条件>
| <検索条件> AND <検索条件>
<値> = {列名 [ <配列定義>]
| <定数> | <計算式> | <内部関数>
| UDF ([<値> [, <値>...]])
| NULL | USER | CURRENT_USER | CURRENT_ROLE
| RDBSDB_KEY}
[COLLATE コレクションオーダー]
<定数> = 数値
| '文字列'
| CHARSETNAME 'キャラクタセット名'
<内部関数> = COUNT (* | [ALL] <値> | DISTINCT <値>)
| SUM ([ALL] <値> | DISTINCT <値>)
| AVG ([ALL] <値> | DISTINCT <値>)
| MAX ([ALL] <値> | DISTINCT <値>)
| MN ([ALL] <値> | DISTINCT <値>)
| CAST (<値> AS <データ型>)
| UPPER (<値>)
| GEN_ID (ジェネレーター名, 数値)
<演算子> =
{ = | < | > | <= | >= | != | < | !> | <> | != }
<単項 SELECT 文> = 1 行・1 列の値を返す SELECT 文
<単列 SELECT 文> =
1 列のみで、0 行以上の行を返す SELECT 文
<SELECT 表現> =
0 行以上の行を返す、<値>のリストに対する SELECT 文

```

- テーブルにすでに存在するデータと矛盾する **PRIMARY KEY** 制約や **UNIQUE** 制約を追加しようとした場合、**ALTER TABLE** 文は失敗します。また、以下にあげる列を削除・変更しようとしたときも失敗します。
 - 列が **UNIQUE/PRIMARY KEY/FOREIGN KEY** 制約の一部に使用されている場合。
 - 列が **CHKCKE** 制約で使用されている場合。
 - 列が計算項目の中で使用されている場合。
 - 列が **VIEW** やトリガー/ストアド等の、他のデータベースオブジェクトから参照されている場合。
- 列に対する変更や削除を行う場合、既存のデータを失う可能性がありますので、必ずバックアップを取ってから作業をするようにして下さい。

Interbase の日本語版ドキュメントでは、「列の変更や削除を行うと、その列に格納されているデータはすべて失われます」という記述がありますが、英語版ドキュメント **IB6Beta** では、「**When a column is dropped, all data stored in it is lost.**」となっており、変更時にはデータは保存されるのが正解だと思います。

しかし、こうした作業に失敗は許されません。必ずバックアップを取りましょう。

※本書執筆のためテストを繰り返している最中に、実際にデータを失ったとした考えられない状況に出会いました。再現することができないのですが、**ALTER TABLE** 文を繰り返し実行していると、こういう事もあるようです。
- 各列のデータ型を変更する際には、既存のデータを失うような変更はすることが出来ません。例えば、**CHAR(4)** を **CHAR(2)** に変更することは出来ません。たとえ、データが空であっても駄目です。**CHAR(4)** を **CHAR(5)** に変更することは可能です。有効なデータ型の変換規則は **APPENDIX B** を参照して下さい。

例 :

```
ALTER TABLE T_Test
  ADD Name2 CHAR(10) DEFAULT '' NOT NULL ;

ALTER TABLE T_Test ALTER Name2 TO Name3;

ALTER TABLE T_Test ADD CONSTRAINT UK_Name_Name2
  UNIQUE (Name, Name3);

ALTER TABLE T_Test DROP CONSTRAINT UK_Name_Name2;

ALTER TABLE T_Test DROP Name3;
```

※制約名を指定しない場合、システムで自動的に一意の制約名がつけられます。これがわからないと制約を削除することは出来ません。各制約の管理は、システムテーブルの **RDB&RELATION_CONSTRAINTS** でされているので、これを参照します。

```
SQL> SELECT RDBSCONSTRAINT_NAME
CON> FROM RDBSRELATION_CONSTRAINTS
CON> WHERE RDBSRELATION_NAME = 'T_TEST'
CON> AND RDBSCONSTRAINT_TYPE = 'UNIQUE';
```

```
RDBSCONSTRAINT_NAME
=====
```

```
UK_NAME_NAME2
```

ALTER EXCEPTION

ALTER EXCEPTION 文は既存の例外のメッセージのみを変更することが出来ます。

ALTER EXCEPTION 例外名 '例外メッセージ';

- ・ 例外メッセージには、日本語を含めることが可能です。

ALTER INDEX

ALTER INDEX 文は既存のインデックスの使用可能/使用停止を設定するために使用します。

ALTER INDEX インデックス名 {**ACTIVE** | **INACTIVE**};

- ・ インデックスの使用を停止してから、再度使用可能にすると、インデックスが再作成されて検索のパフォーマンスが向上します。**SET STATISTICS** 文を使用しても同様の効果を得ることが出来ます。
- ・ 大量の行を挿入・削除・更新する場合に、インデックスを一時的に停止することでパフォーマンスを向上させることが出来ます。その場合、作業を開始する前にインデックスの使用を停止して、終了後に再度使用可能にしてください。もっとも、よほど大量のデータ更新でない限り、この必要は無いと思います。
- ・ 現在使用中のインデックスを使用停止にした場合、実際に停止するのは使用が終了してからになります。
- ・ **PRIMARY KEY/FOREIGN KEY** 制約が設定されているインデックスについては、使用を停止することが出来ません。

※筆者が試した限りでは、**CREATE TABLE** 文で指定された **UNIQUE** 制約のインデックスは停止することが出来ませんでしたが、**ALTER TABLE** 文で後から付け加えた **UNIQUE** 制約のインデックスは停止/再開することが出来ました。

Firebird SQL - DML 系コマンド INSERT、UPDATE、SELECT、DELETE 等

DML(Data Manipulation Language)系コマンドは、その名の示すとおりデータを操作するためのコマンド群です。CREATE 文や ALTER 文がデータベースの構造や状態を定義/変更するために使用されることと比して、データベースにデータを格納し、選択/更新し、削除するために使用されます。

複雑で難解な SELECT 文を競う SQLOR(SQL マニア?)も存在するように、その使用方法やテクニックは多義にわたります。

INSERT 文

INSERT 文は、指定したテーブルまたはビューに、新しい行を挿入します。

INSERT INTO テーブル名 | ビュー名

[(列名 [, 列名 ...])]}

{VALUES (<値> [, <値> .]) | <SELECT 表現>;}

<値> = { <定数> | <単項 SELECT 文>

| <SQL 関数> | UDF ([<値> [, <値> ...]])

| NULL | USER | RDBSDb_KEY

} [COLLATE コレレーションオーダー]

<定数> = 数値

| '文字列'

| キャラクタセット名 '文字列'

<SQL 関数> = CAST (<値> AS <データ型>)

| UPPER (<値>)

| GEN_ID (ジェネレーター名, <値>)

<SELECT 表現> = 挿入先のテーブルで必要とされる列数が一致し、各列のデータ型が一致又は許容されるデータ型であるような SELECT 文。通常、同様の構造のテーブル同士での INSERT に使用する。

<単項 SELECT 文> = 1 列のみで、0 行以上の列を返す有効な SELECT 文

- 挿入先の列を指定しない場合、INSERT 文ですべての列のデータを指定しなくてはなりません。また、その場合のデータの並び順は、テーブルの列の順に従います。

例:

```
CREATE TABLE A_TABLE(  
    ID INTEGER NOT NULL PRIMARY KEY,  
    NAME CHAR(10) DEFAULT 'tomneko',  
    DT TIMESTAMP);
```

```
INSERT INTO A_TABLE VALUES(1, 'tom', 'TODAY');
```

```
SELECT * FROM A_TABLE;
```

ID	NAME	DT
1	tom	2003-02-22 00:00:00

- 挿入先の列を指定した場合、指定しなかった列にデフォルト値が設定されていれば、デフォルト値が挿入されます。デフォルト値が設定されていなければ、NULL 値が挿入されます。

```
INSERT INTO A_TABLE(ID, DT) VALUES(2, 'YESTERDAY');
```

```
INSERT INTO A_TABLE (ID, NAME) VALUES(3, 'jelly');
```

```
SELECT * FROM A_TABLE;
```

ID	NAME	DT
1	tom	2003-02-23 00:00:00
2	tomneko	2003-02-22 00:00:00
3	jelly	<null>

- VALUES 句でデータを指定した場合、1 行のデータが挿入されます。
- 複数のデータを一度に挿入する場合、SELECT 文によって挿入するデータを指定します。その場合、挿入先の列数と同じ列数を返す SELECT 文でなければなりません。

```
INSERT INTO B_TABLE SELECT * FROM A_TABLE;
```

- 挿入元になるテーブルに、挿入先と同じテーブルを指定することも出来ませんが、挿入が無限に繰り返される恐れがあるため勧められません。

例 :

```
CREATE TABLE C_TABLE(  
ID INTEGER, NAME CHAR(10));  
  
INSERT INTO C_TABLE VALUES(1, 'TOMNEKO');  
INSERT INTO C_TABLE VALUES(2, 'JELLY');  
INSERT INTO C_TABLE VALUES(3, 'POTESI');  
  
INSERT INTO C_TABLE SELECT * FROM C_TABLE;
```

※最後の行を実行すると、プロンプトに戻って来ないで、無限ループに陥ります。あっという間にデータベースファイルが数百メガバイトになりますので、注意してください。(というか実行しないようにして下さい)

UPDATE

UPDATE 文は、テーブル又は更新可能なビューの行の全体又は一部を更新します。

```
UPDATE { テーブル名 | ビュー名 }  
SET 列名 = <値> [, 列名 = <値> ... ]  
[WHERE <検索条件>  
<値> = { 列名 [ <配列要素> ]  
| <定数> | <計算式> | <内部関数>  
| UDF ([ <値> [, <値> ... ]])  
| NULL  
| USER  
| CURRENT_USER  
| CURRENT_ROLE}  
[COLLATE COLLATION ]  
<配列要素> = [[X:]Y [, [X:]Y ... ] ]  
<定数> = 数値  
| '文字列'  
| CHARSETNAME 'キャラクタセット名'  
<内部関数> = CAST ( <値> AS <データ型> )  
| UPPER ( <値> )  
| GEN_ID ( ジェネレーター名 , 数値 )  
<計算式> = 単一の値になる有効な SQL 式  
<検索条件> = CREATE TABLE 文を参照して下さい。
```

- ・ 検索条件を指定せずに UPDATE 文を実行すると、すべての行が対象となります。
- ・ 直接指定可能な内部関数以外の内部関数も、単項 SELECT 文の一部として使用することが可能です。

例 :

```
UPDATE T_Test SET Name = 'tomeko'  
WHERE Name = 'toml';
```

SELECT

SELECT 文はテーブルからデータを抽出するために使用されます。WHERE 句ではサブクエリーを使用できますが、FROM 句では使用できません。代わりに、行を返すストアドプロシージャを利用することが可能です。

```
SELECT [FIRST 数値][SKIP 数値]  
[DISTINCT | ALL]  
{* | <値> [, <値> ... ]}  
FROM <テーブル参照> [, <テーブル参照> ... ]  
[WHERE <検索条件>]  
[GROUP BY 列名 [COLLATE コレクションオーダー ]  
[, 列名 [COLLATE コレクションオーダー ] ... ]  
[HAVING <検索条件>]  
[UNION [ALL] <SELECT 表現>]  
[PLAN <PLAN 表現>]  
[ORDER BY <ソート表現>]  
[FOR UPDATE [OF 列名 [, 列名 ... ]]];  
<値> = {列名 [ <配列定義> ]  
| <定数> | <計算式> | <内部関数>  
| UDF ([<値> [, <値> .. ]])  
| NULL | USER | CURRENT_USER | CURRENT_ROLE  
| RDBSDB_KEY}  
[COLLATE コレクションオーダー]  
[AS アlias名 ]
```

<配列定義> =
 [[x :] y [, [x :] y ...]]

<定数> = 数値
 | '文字列'
 | CHARSETNAME 'キャラクタセット名'

<内部関数> = COUNT (* | [ALL] <値> | DISTINCT <値>)
 | SUM ([ALL] <値> | DISTINCT <値>)
 | AVG ([ALL] <値> | DISTINCT <値>)
 | MAX ([ALL] <値> | DISTINCT <値>)
 | MIN ([ALL] <値> | DISTINCT <値>)
 | CAST (<値> AS <データ型>)
 | UPPER (<値>)
 | GEN_ID (ジェネレーター名, 数値)

<テーブル参照> = <結合テーブル>
 | テーブル名
 | ビュー名
 | ストアドプロシージャ名 [(<値> [, <値> ...])]
 [アリヤス名]

<結合テーブル> =
 <テーブル参照> <結合型> <テーブル参照>
 ON <検索条件>
 | (<結合テーブル>)

<結合型> = [INNER] JOIN
 | {LEFT | RIGHT | FULL } [OUTER]} JOIN

<検索条件> =
 <値> <演算子> {<値> | <単項 SELECT 文>}
 | <値> [NOT] BETWEEN <値> AND <値>
 | <値> [NOT] LIKE <値>[ESCAPE <値>]
 | <値> [NOT] IN (<値> [, <値> ...]
 | <単項 SELECT 文>)
 | <値> IS [NOT] NULL
 | <値> {>= | <=} <値>
 | <値> [NOT] {= | < | >} <値>
 | <値> <演算子>
 {ALL | SOME | ANY} (単項 SELECT 文)
 | EXISTS (SELECT 表現)
 | SINGULAR (SELECT 表現)
 | <値> [NOT] CONTAINING <値>
 | <値> [NOT] STARTING [WITH] <値>
 | (<検索条件>)
 | NOT <検索条件>
 | <検索条件> OR <検索条件>
 | <検索条件> AND <検索条件>

<PLAN 表現> = [JOIN | [SORT] [MERGE]]
 ({ <PLAN 項目> | <PLAN 表現> }
 [, { <PLAN 項目> | <PLAN 表現> } ...])

<PLAN 項目> = { TABLE | アリヤス名 }
 {NATURAL | INDEX (<インデックス名>
 [, <インデックス名> ...])
 | ORDER <インデックス名>}

<ソート表現> = { 列名 | 列番号 }
 [COLLATE コレクションオーダー]
 [ASC[ENDING] | DESC[ENDING]]
 [, <ソート表現> ...]

<計算式> = 単一の値になる有効な SQL 式

<単項 SELECT 文> = 1 行・1 列の値を返す SELECT 文

<単列 SELECT 文> =
 1 列のみで、0 行以上の行を返す SELECT 文

<SELECT 表現> =
 0 行以上の行を返す、<値>のリストに対する SELECT 文

- ・ 列指定に*を使用すると、すべての列を指定したことになります。

例:

```
SELECT * FROM T_TEST;
```

- ・ **!!FB!!** FIRST 句と SKIP 句は Firebird で拡張された構文です。FIRST 数値 で、SELECT 文の結果の先頭から指定した行数を返します。また、SKIP 句は先頭から指定した行数を文字通りスキップして結果を返します。

数値には、有効な正の数を返す数値及び計算式を指定することが可能です。残念ながら集計関数やサブクエリーを指定することは出来ません。EXTRACT() 関数や CAST() 関数は使用可能です。また、() でくくることで UDF も使用可能です。

サブクエリーを指定することが出来れば、総行数の 10% を返すというようなことも可能になりますが、この機能は ver. 1.5 で実装される予定です。

一方、UDF が使用可能と言うことは、GROUP BY 句で解説する UDF ラッパーというテクニックを使用してサブクエリーを行うことが出来ます。

例：

```
SELECT FIRST 10 SKIP 5 * FROM EMPLOYEE;
SELECT FIRST (10 * 2) * FROM EMPLOYEE;
SELECT SKIP (2 - 3) * FROM EMPLOYEE;
```

```
SELECT FIRST (BIN_OR((SELECT COUNT(*)
FROM T_TEST)/2, 1)) * FROM T_TEST;
```

※このクエリーの結果は、全体の半分の行を結果として返します。

- FROM 句で指定するテーブル参照には、テーブル、ビュー、行を返すストアドプロシージャを指定することが出来ます。各オブジェクトは JOIN 構文によって結合することや、ネストした結合を指定することが可能です。

例：

```
SELECT * FROM T_Test JOIN T_Test2
ON T_Test.ID = T_Test2.ID;
```

```
SELECT * FROM (T_Test T1 LEFT JOIN T_Test2 T2
ON T1.ID = T2.ID) T12 JOIN T_Test3 T3
ON T12.ID = T3.ID;
```

- WHERE 句を指定した場合、検索条件に一致する行だけが返されます。検索条件には単純な演算子による<値>の比較から、集合演算まで様々な条件を指定することが可能です。

BETWEEN 演算子は<値>が指定した範囲内に存在するかどうかを検証します。

例：

```
SELECT * FROM T_Test
WHERE DT BETWEEN '2003/1/1' AND '2003/2/28';
```

LIKE 演算子は<値>が指定した<値>に含まれるかどうかを検証します。対象となるのは文字型のデータ、CHAR、VARCHAR 等です。ワイルドカードとして%と_が使用可能です。%は0文字以上の任意の文字列を、_は任意の1文字とマッチします。

例：

```
SELECT * FROM T_TEST
WHERE Name LIKE 'tom%';
```

IN 演算子は、<値>が指定した集合に含まれるかどうかを検証します。() で囲まれた値のリストまたは1列で0行以上の行を返す SELECT 文が比較対象となります。() の中を空にするとエラーになります。

例：

```
SELECT * FROM T_Test
WHERE Name IN ('tom', 'jelly');
```

```
SELECT T1.* FROM T_Test T1
WHERE T1.Name IN
(SELECT T2.Name FROM T_Test T2
WHERE T2.Name = 'tom');
```

<値>を NULL 値と比較する場合は、IS [NOT] NULL 演算子を使用します。= や <> は NULL には使用できないので注意して下さい。

例：

```
SELECT * FROM T_Test
WHERE Name2 IS NOT NULL;
```

ALL/ANY/SOME 演算子を使った比較演算では、指定した SELECT 文が返す値の集合に対して、それぞれ ALL はすべての要素に対する比較が真であれば真を返します。ANY/SOME は同義ですが、比較対象の集合に対して、そのいずれかの要素に対する比較が真であれば真を返します。

例：

```
2 > ALL ( 1, 3, 5) → False
6 > ALL ( 1, 3, 5) → True
2 > ANY ( 1, 5, 9) → True
1 > ANY ( 2, 4, 6) → False
```

※わかりやすくするため、比較対象を数値で表していますが、実際には SELECT 文によるサブクエリーで無ければなりません。

EXISTS 演算子は、指定したサブクエリーに行が 1 行でも存在するかどうかを検証します。行があれば真を返します。
逆に、SINGULAR 演算子は、指定したサブクエリーが 1 行だけであるかどうかを検証します。行が 1 行だけであれば真を返します。

例：

```
SELECT T1.* FROM T_Test T1
WHERE EXISTS (SELECT 1 FROM T_Test T2
              WHERE T2.ID > 1);
```

上記の例では、T_Test の ID が 1 より大きい行が 1 つでもあれば、すべての行が返されます。

```
SELECT T1.* FROM T_Test T1
WHERE SINGULAR (SELECT 1 FROM T_Test T2
               WHERE T2.ID > 1);
```

この例では、T_Test の ID が 1 より大きい行が複数あれば、1 行も返りません。

※EXISTS 演算子と SINGULAR 演算子のサブクエリーでは、行が存在するかどうかだけが必要なので、SELECT 1 とすることによってデータを選択するオーバーヘッドを最小にすることが出来ます。

EXIST 演算子と IN 演算子は入れ替えが可能です。筆者の経験では、IN 演算子を使用して異様に遅いことがありましたが、EXISTS 演算子に変更して、多少ましになった事があります。実際、Firebird の IN 演算子は遅いので有名なようです。その件で ib-support ML でも勧められましたが、できるだけ、EXISTS を使用するようにした方がよいかもかもしれません。

例：

```
SELECT T1.* FROM T_Test T1
WHERE T1.ID IN (SELECT T2.ID FROM T_Test2 T2);
```

```
SELECT T1.* FROM T_Test T1
WHERE EXISTS (SELECT 1 FROM T_Test2 T2
              WHERE T1.ID = T2.ID);
```

上記の例は同じ結果を返します。IN 演算子による比較対象の集合に値が含まれると言うことは、結局、その条件を内側に持ち込んだときに行が存在するかどうかということ代替できます。

ちなみに、今回の場合、T_Test1 でも T_Test2 でも ID が重複していなければ、EXISTS を SINGULAR に変更しても結果は同じです。T1.ID=T2.ID の行はそれぞれ 1 行しか存在しないからです。

また、IN 演算子は ALL/ANY/SOME 演算子とも置き換えが可能です。その場合、以下のようになります。

```
<値> IN (値, 値...) → <値> = ANY (値, 値...)
<値> NOT IN (値, 値...) → <値> <> ALL(値, 値...)
```

- STARTING WITH 演算子と CONTAINING 演算子は共に文字列に対する比較演算を実行します。STARTING WITH 表現は<値>が特定の文字または文字列で始まっているかどうかを検証します。CONTAINING 演算子は指定された文字または文字列が、<値>のどこかに含まれていれば真を返します。
これらは、LIKE 演算子と置き換えが可能です。

```
<値> STARTING WITH '文字列'
→ <値> LIKE '文字列%'
<値> CONTAINING '文字列'
→ <値> LIKE '%文字列%'
```

- GROUP BY 句では、列名のリストに従って検索の結果を同一の値でグループ化します。
!!FB!! Firebird では GROUP BY 句に UDF を使用した列を指定することが出来ます。

例：

```
SELECT STRLEN(RTRIM(RDBSRELATION_NAME)),
COUNT(*)
FROM RDBSRELATIONS
GROUP BY STRLEN(RTRIM(RDBSRELATION_NAME))
ORDER BY 1
```

上記の例は、リリースノート記載のものですが、Firebird に付属の ib_udf.dll に含まれる UDF 関数のうち、STRLEN と RTRIM を利用して同じ文字列長のデータがいくつあるかを集計しています。リリースノートでは、ORDER BY 2 となっていますが、1 列目でソートした方が結果がわかりやすいので変更してあります。

!!FB!! また、GROUP BY UDF 機能の副次的作用として、やはり GROUP BY 句で使用することが出来なかった内部関数についても UDF を利用して、GROUP BY が可能となりました。これを UDF ラッパーといいます。

例えば、これまで年間を通して月次の集計をしたいと思った場合、InterBase では DATE 型/TIMESTAMP 型から月次を取り出して GROUP BY で集計するということが出来ませんでした。Firebird では可能です。(いばるようなことでもないか・・・)

```
SELECT MAX(EXTRACT(MONTH FROM SALESDATE)),
```

```

COUNT(SALESCOUNT)
FROM T_Table T1
GROUP BY
  BIN_OR(
    (SELECT EXTRACT(MONTH FROM SALESDATE)
     FROM T_Table T2
     WHERE T1.ID = T2.ID), 1)
ORDER BY 1;

```

この例では、T_Tableには約3万6千件のデータが入っており、売上日時がTIMESTAMP型で、売上個数がSALESCOUNTにINTEGER型で格納されています。が、この月別集計は約2秒で完了しました。

余談になりますが、実際のテーブルでは、主キーは3つに分かれているのですが、これを上記のように1つだけ指定してインデックスが効かない状態で実行すると、40秒以上かかりました。

ここで、BIN_OR(<値>, <値>)というUDFを使用していることに注意して下さい。これも同じく、ib_udf.dllに含まれるUDFで、バイナリレベルでのOR演算を行った結果を返します。従って、BIN_OR(<値>, 1)の結果は、常に<値>そのものになります。ある種のトリックです。

- HAVING句には、WHERE句と同様の検索条件を指定することが出来ます。ただし、WHERE句ではSUM()やMAX()などの複数行を対象とした集計関数を指定することができませんが、HAVING句では可能となっています。

その場合、集計の対象となるのはグループ化された各行の集合となります。

例：

```

SELECT Name, COUNT(*)
FROM T_Test
GROUP BY Name
HAVING COUNT(*) >= 2

```

- UNION [ALL]句は、複数のSELECT文を結合するために使用します。[ALL]を指定しない場合、重複する行は省かれます。UNION ALLとした場合、重複も含めてすべての行が返されます。

また、UNIONで結合されるSELECT文の選択列はデータ型も含めて同一でなければなりません。しかし、不足している列をNULLで補うことで結合することも可能です。

例：

```

SELECT ID, Name FROM T_Test
UNION
SELECT ID, Name FROM T_Test2;

SELECT ID, Name, NULL FROM T_Test
UNION ALL
SELECT NULL, Name, ID FROM T_Test2;

```

- PLAN句を指定すると、SELECT文を実行する際に使用するインデックス等を独自に指定することが出来ます。しかし、Firebirdのオプティマイザはほぼ自動的に最良のクエリーPLANを決定するので、PLAN句を使用することはまずありません。

この点、他のRDBMSとの大きな違いかもしれません。Firebirdにおけるチューニングは、インデックスの適切な作成と、最適なSELECT文の組み立てにかかっています。

また、通常のSQL文での処理がボトルネックになる場合、トリガーやストアプロシージャの利用を考えた方がよいでしょう。特に、ストアプロシージャにおいては、rdbSdb_keyを利用した、もっとも高速な行の指定が可能なので、数十分かかるようなSQL文があるならば、ストアプロシージャの使用を検討して下さい。

- ORDER BY句では、通常列名を使用してSELECT文の結果に対するソートの順序を指定します。集計関数などを使用した場合は、列名での指定ができませんので、1からはじまる列番号で指定することになります。

また、デフォルトのソート順序はASC[ENDING]=昇順ですが、DESC[ENDING]を指定することで、降順のソートも可能です。

さらに、COLLATE句でコレーションオーダーを指定すると、キャラクタセットによっては複数保持しているソート順を使用することが出来ます。

例：

```

SELECT * FROM T_Test ORDER BY ID;

SELECT * FROM T_Test ORDER BY 1, 2;

```

- FOR UPDATE句は、他のRDBMSでは検索結果に対する明示的なロックを提供します。Firebirdでは、FOR UPDATE句は埋め込みSQLにおける、更新対象を指定するために使用されるのみで、明示的なロックをかけることは出来ません。

ISQL等においては、FOR UPDATE句を指定してもエラーにはなりませんが、無視されます。FOR UPDATE句による明示的なロックは、Firebirdのver. 1.5以降で提供される予定です。

ver. 1.0でテーブルに対する明示的なロックが必要な場合は、SET TRANSACTION文でRESERVING句を使用してください。

DELETE

DELETE文は、テーブル又は更新可能なビューから、検索条件に従って行を削除します。

DELETE FROM [テーブル名 | ビュー名]
[WHERE <検索条件>];

- ・ **WHERE** 句を指定しないと、テーブル/ビューのすべての行を削除するので、注意して下さい。

例：

DELETE FROM T_Test WHERE Name = 'hageri';

Firebird SQL - 内部関数、その他の構文

Firebird は他の RDBMS と違い、様々な関数を UDF(ユーザー定義関数)として提供しているので、内部関数についてはほんのいくつかしか実装していません。

それらは、AVG()・COUNT()・MAX()・MIN()・SUM()の各集合関数と、CAST()・EXTRACT()・UPPER()の各変換関数になります。また、ジェネレーターから値を取得して結果を返す GEN_ID() も内部関数に区分されますが、これについてはジェネレーターの項で解説しました。!!FB!! Firebird では、SUBSTRING() 関数が追加されました。

InterBase では集合関数ですら GROUP BY 句に指定することが出来なかったのが、集計を取る場合などに苦労しました。Firebird では、UDF を GROUP BY 句に指定することが出来ますし、UDF を利用して内部関数を指定することもできます。UDF の使用範囲が広がったことで、Firebird はさらに大きな可能性を手に入れました。

UDF については、本書ではスペースの問題からふれることが出来ませんが、別途稿を改めて解説したいと思います。

内部関数

AVG() 関数

AVG() 関数は、指定した列または式の値から平均値を返します。対象となるのは、数値型のデータのみとなります。

AVG([ALL] 値 | DISTINCT 値)

- ALL を指定した場合、すべての値が集計の対象となります。ALL はデフォルト値なので、値のみを指定した場合はすべての値が集計の対象となります。
- DISTINCT を指定した場合、重複する値を除外してから集計を行います。
- NULL または不定の値については計算から除外されます。カウント自体がされませんので、注意して下さい。結果が NULL になるわけではありません。
- 集計の結果については、元になるデータ型の精度まで切り捨てられます。整数型のデータの平均値をとった場合、結果は整数となります。
- SELECT 文が 0 行を返した場合、AVG() 関数は NULL を返します。

例：

```
SELECT AVG(SALARY), COUNT(*), MAX(SALARY), MIN(SALARY), SUM(SALARY) FROM EMPLOYEE;
```

AVG	COUNT	MAX	MIN	SUM
2750534.952	42	9900000	22935	115522468

※上記の例ではサンプルデータベース、EMPLOYEE.GDB に対して集計関数を使用してみました。

COUNT 関数

COUNT() 関数は、SELECT 文で検索条件に一致する行数を返します。

COUNT (* | [ALL] <値> | DISTINCT <値>)

- * を指定した場合、指定した検索条件に一致する行数を、NULL 値も含めて集計して返します。
- [ALL] <値>を指定した場合、NULL を除外した行数を集計して返します。<値>には列又は有効な SQL 式を指定することが出来ます。ALL は<値>を指定した場合のデフォルトなので、<値>のみを指定した場合、NULL を除外した行数を返します。
- DISTINCT <値>を指定した場合、NULL を除外した上で、重複した値を排除して行数をカウントします。

例：

```
SELECT COUNT(*) FROM EMPLOYEE;
```

・・・42 行

```
SELECT COUNT(PHONE_EXT) FROM EMPLOYEE;
```

・・・39 行

```
SELECT COUNT(DISTINCT PHONE_EXT) FROM EMPLOYEE;
```

・・・38 行

```
SELECT COUNT(1) FROM EMPLOYEE;
```

・・・42 行

```
SELECT COUNT(NULL) FROM EMPLOYEE;
```

・・・エラー

MAX 関数

MAX() 関数は、SELECT 文において指定した列の最大値を返します。NULL 値は除外されます。

MAX ([ALL] <値> | DISTINCT <値>)

- 有効な行の数が 0 行の場合は、NULL 値を返します。

- **ALL** を指定した場合、全ての値から最大値を探します。**DISTINCT** を指定した場合、重複した値を除外してから最大値を返します。論理的に考えると、結果はどちらも同じでしょう。
- <値>には、列・定数・式・非集合関数・UDF を指定することが可能です。
- **CHAR**、**VARCHAR**、テキスト型の **BLOB** に対して **MAX()** 関数を実行した場合、その列に設定されているキャラクタセットとコレクションオーダーに従って、最大値が返されます。

MIN 関数

MIN() 関数は、**SELECT** 文において指定した列の最小値を返します。**NULL** 値は除外されます。

MIN ([**ALL**] <値> | **DISTINCT** <値>)

- 有効な行の数が **0** 行の場合は、**NULL** 値を返します。
- **ALL** を指定した場合、全ての値から最小値を探します。**DISTINCT** を指定した場合、重複した値を除外してから最小値を返します。論理的に考えると、結果はどちらも同じでしょう。
- <値>には、列・定数・式・非集合関数・UDF を指定することが可能です。
- **CHAR**、**VARCHAR**、テキスト型の **BLOB** に対して **MAX()** 関数を実行した場合、その列に設定されているキャラクタセットとコレクションオーダーに従って、最小値が返されます。

CAST 関数

CAST() 関数は、指定された<値>のデータ型を他のデータ型に変換します。

CAST (<値> **AS** データ型)

- <値>には、列・定数・式・非集合関数・UDF を指定することが可能です。
- データ型の変換規則は、**ALTER TABLE** 文におけるデータ型の変換と違い、以下のようになります。なお、**BLOB** を除く全てのデータ型は文字型/可変長文字型に変換することが出来ますので、一覧から省略してあります。

数値型 → 他の数値型
 文字型、可変長文字型 →
 日付型、タイムスタンプ型、時刻型
 日付型 → タイムスタンプ型
 時刻型 → タイムスタンプ型
 タイムスタンプ型 → 日付型、時刻型
BLOB 型 → 変換不可能

- 変換不可能な変換を実行すると、エラーが返されます。例えば数値型を日付型に変換したり、文字型でも日付を表さない適当な文字列 ('tonneko') などの日付型に変換すると、当然エラーになります。
- 可変長文字列型/**VARCHAR** への変換時には、文字列の長さを指定しなくてはなりません。また、その長さは変換元のデータを完全に格納できる大きさでなければなりません。そうでない場合、エラーが返されます。

例：

```
SELECT CAST('tonneko' AS VARCHAR(10))
FROM RDBSDATABASE;
CAST
=====
tonneko
```

```
SELECT CAST('tonneko' AS VARCHAR(3))
FROM RDBSDATABASE;
arithmetic exception, numeric overflow, or string truncation
```

EXTRACT 関数

EXTRACT() 関数は、**DATE**・**TIME**・**TIMESTAMP** 型から日付や時刻に関する情報を取り出します。

EXTRACT (キーワード **FROM** <値>)

- 指定可能なキーワードを以下の表に示します

キーワード	データ型	取りうる範囲
YEAR	SMALLINT	0 ~ 5400
MONTH	SMALLINT	1 ~ 12

DAY	SMALLINT	1 ~ 31
HOUR	SMALLINT	0 ~ 23
MINUTE	SMALLINT	0 ~ 59
SECOND	DECIMAL(6, 4)	0 ~ 59.9
WEEKDAY	SMALLINT	1 ~ 6
YEARDAY	SMALLINT	1 ~ 366

- それぞれのデータ型の中に存在しない部分を取り出そうとした場合、エラーが返されます。
- WEEKDAY は 0=日曜日、1=月曜日となっており、ISO8601 規格とも、Delphi の DayOfWeek() 関数とも結果が違うことに注意して下さい。それぞれの違いを以下に示します。

ISO8601 : 1=月曜日、2=火曜日、...、7=日曜日
Firebird : 0=日曜日、1=月曜日、...、
DayOfWeek : 1=日曜日、2=月曜日、...

※Delphi には、ISO8601 規格に準拠した DayOfTheWeek() 関数も用意されています。

EXTRACT() 関数の結果を ISO8601 に合わせるためには以下のようにする必要があります。

EXTRACT(WEEKDAY FROM 日付 - 1) + 1

UPPER 関数

UPPER() 関数は、文字列中の文字をすべて大文字に変換します。当然ですが、大文字小文字の区別のないキャラクタセット (SJIS_0208 など) に使用した場合、変換は行われません。

UPPER (<値>)

- <値>には列名、定数、式、関数、文字型を返す UDF を指定することが出来ます。

例:

SELECT UPPER(Name) FROM T_Test;

SUBSTRING 関数

SUBSTRING 関数は、文字列から部分文字列を取り出します。

SUBSTRING(<文字列> FROM <開始位置> [FOR <長さ>])

- <文字列>には、列名や文字型の定数、単項 SELECT 文、文字型の結果を返す計算式、UDF や内部関数を指定できます。
- <開始位置>と<長さ>は正の整数のみを指定可能です。<開始位置>は 1 から始まり、<長さ>には 0 を指定することも可能です。

例:

SELECT SUBSTRING('tomeko' FROM 1 FOR 2) FROM RDBSDATABASE;

その他の構文

CONNECT 文

CONNECT 文は、データベースに接続するために使用します。

CONNECT 'ファイル名'

[USER 'ユーザー名'] [PASSWORD 'パスワード']

[CACHE 数値] [ROLE 'ロール名'];

- ファイル名の指定方法は、CREATE DATABASE 文を参照してください。
- CACHE 句を指定した場合、プログラムが一度に使用できるデータベースページ数が決まります。デフォルトは 75 で、最小値は 45 となります。指定可能な最大値は 65535 となります。
Firebird のデータベースキャッシュは、サーバーレベル / データベースレベル / 接続レベルで設定が可能ですが、それぞれで設定された値の最大値が実際に適用されます。増やすことは出来ても減らすことは出来ません。

※isql で設定を変更しましたが、特に変化は無いようです。また、SET STAS ON で表示させたところ、Buffers には 4096 が表示されました。これは、ibconfig で設定されているクライアントマップサイズなので、どれが正しいのかいまいち怪しいです。

#DATABASE_CACHE_PAGES 2048

#SERVER_CLIENT_MAPPING 4096

SET STATISTICS 文

SET STATISTICS 文は、インデックスの選択性を再計算します。

SET STATISTICS INDEX インデックス名;

- インデックスの選択性とは、テーブルへのアクセス時に **Firebird** のオブティマイザが使用する、テーブル中の行を識別できるような数値を計算したものです。
この値はメモリにキャッシュされ、オブティマイザがクエリーに対する最適化を行うために使用されます。テーブルのインデックスに含まれる列の値で、重複する値が大幅に増減する場合、インデックスの選択性をさげ違算するとパフォーマンスが向上します。
- インデックスの選択性は、**ALTER INDEX** 文でいったん **INACTIVE** にした後、**ACTIVE** にすることでも再計算されます。
- **Gerhard Knapp** 氏作の **SET STATISTICS** を使用するとすべてのインデックスの選択性を簡単に再計算することが出来ます。**Knapp** 氏によると、**INFORMIX** にも同様の「**UPDATE STATISTICS**」文があり、これを事項した後ですばらしくパフォーマンスが向上したそうです。URL を以下に示します。

<http://www.clientel.at/public/>

Firebird SQL - DROP 文系

DROP 文は、各データベースオブジェクトをデータベースから削除するために使用します。各データベースオブジェクトが他のデータベースオブジェクト等によって使用されているといった依存関係がある場合、削除することは出来ません。

DROP 文によって各データベースオブジェクトを削除することが出来るのは、そのデータベースの所有者と **SYSDBA**、**UNIX** システムではシステムの **root** 権限を所有するユーザーのみです。

DROP DATABASE

DROP DATABASE 文は、現在接続されているデータベースを削除します。二次ファイルやシャドウファイル、ログファイルも含めてすべて削除されるので注意してください。また、当然ですが、格納されているデータもすべて失います。バックアップファイルは削除されません。

DROP DATABASE;

DROP DOMAIN

DROP DOMAIN 文は、指定したドメインをデータベースから削除します。テーブルの列の定義ですでに使用されている場合、エラーとなります。

DROP EXCETION

DROP EXCEPTION 文は、指定した例外をデータベースから削除します。例外がトリガーやストアドプロシージャで使用されている場合、エラーとなります。

DROP DOMAIN ドメイン名;

DROP INDEX

DROP INDEX 文は、指定したインデックスをデータベースから削除します。使用中のインデックスは、使用が終了するまで削除されません。削除できるのは、ユーザー定義のインデックスだけです。

DROP INDEX インデックス名;

DROP ROLE

DROP ROLE 文は、データベースから **SQL** ロールを削除します。ロールを削除すると、そのロールによってユーザーに与えられていた各データベースオブジェクトに対する特権も削除されます。

DROP ROLE ロール名;

DROP SHADOW

DROP SHADOW文は、指定したシャドウセットをデータベースから削除します。シャドウセット番号は、**ISQL** で **SHOW DATABASE** コマンドを使用して確認します。

DROP SHADOW シャドウセット番号;

DROP TABLE

DROP TABLE 文は、指定したテーブルをデータベースから削除します。テーブルのメタデータ / データ / インデックスが削除されます。また、このテーブルに作成されたトリガーも同時に削除されます。

すでに開始されているトランザクション中で参照されているテーブルは、使用後に削除されます。

DROP TABLE テーブル名;

DROP VIEW

DROP VIEW文は、指定したビューの定義をデータベースから削除します。

DROP VIEW ビュー名;

Appendix A Firebirdのデータ型

数値データ型

データ型	サイズ	範囲/有効桁数	説明
SMALLINT	16 ビット	-32,768 ~ 32,767	符号付短整数値型。
INTEGER	32 ビット	-2,147,483,648 ~ 2,147,483,647	符号付長整数値型。
FLOAT	32 ビット	$1.175 \times 10^{-38} \sim 3.402 \times 10^{38}$	IEEE 単精度浮動小数点型。有効桁数 7 桁。
DOUBLE PRECISION	64 ビット	$2.225 \times 10^{-308} \sim 1.797 \times 10^{308}$	IEEE 倍精度浮動小数点型。有効桁数 15 桁。
DECIMAL (precision, scale)	可変長 (16, 32 または 64 ビット)	precision=1~18。正確に格納される有効桁数を指定する。 scale=格納可能な小数点以下の桁数 (precision 以下でなければならない)	特定の小数点以下の桁数を持つ数値。
NUMERIC (precision, scale)	可変長 (16, 32 または 64 ビット)	precision=1~18。正確に格納される有効桁数を指定する。 scale=格納可能な小数点以下の桁数 (precision 以下でなければならない)	特定の小数点以下の桁数を持つ数値。

Firebird のサポートする数値型は、16 ビット・32 ビットの整数型 (INTEGER、SMALLINT)、単精度・倍精度の浮動小数点型 (FLOAT、DOUBLE PRECISION)、書式付固定小数点型 (DECIMAL、NUMERIC) となります。

整数データ型

Firebird では INTEGER と SMALLINT の 2 つの整数型を使用することが出来ます。それぞれの有効桁数は上記の通りです。整数データ型では以下の演算を行うことが出来ます。

- 比較演算子 (=, <, >, >=, <=) を使った比較を行えます。
- CONTAINING, STARTING WITH, LIKE のような他の演算子は、数値として文字列の比較を行います。
- 算術演算子を使って複数の整数での加減乗除を行えます。
- 複数のデータ型で算術演算を行うとき、Firebird は INTEGER、FLOAT、CHAR データ型の間で自動的に型変換を行います。数値データを他のデータ型と比較する演算では、Firebird はまずその他のデータ型を数値型に変換してから、比較を行います。
- テーブルのレコードは、SELECT 文の ORDER BY 句に整数データ型を指定することで、降順または昇順にソートできます。

浮動小数点データ型

Firebird には FLOAT と DOUBLE PRECISION の 2 つの浮動小数点データ型があります。FLOAT は 32 ビット単精度の浮動小数点となり、約 7 桁の有効桁数となります。DOUBLE PRECISION は 64 ビット倍精度の浮動小数点で、約 15 桁の有効桁数となります。浮動小数点データ型においては、小数点以下の桁数は増減する (浮動) ので、同じ列に 12.345 と 1.23 という値を格納することが可能です。

固定小数点データ型

Firebird は、通貨を扱う場合などに利用する固定小数点の数値データ型として、NUMERIC と DECIMAL の 2 つの SQL データ型をサポートしています。どちらのデータ型でも有効桁数と小数点以下の桁数を指定することが出来ます。

- 有効桁数 (precision) は、整数部と小数部をあわせた最大の桁数です。有効桁数として指定できる範囲は 1~18 となります。
- 小数点以下の桁数 (scale) は、小数点より右側になる数値の桁数です。小数点以下の桁数に指定できる範囲は 0~precision までとなります。scale は必ず precision 以下でなければなりません。
- NUMERIC と DECIMAL の相違は、NUMERIC(p, s) が厳密に p 桁が格納され、小数点以下の桁数が厳密に s 桁となりますが、DECIMAL(p, s) では p 桁以上が格納され、小数点以下が厳密に s 桁となります。
- NUMERIC と DECIMAL の 2 つの固定小数点データ型はダイレクト 1 とダイレクト 3 で、実際の格納方法が違ってきます。そのため、ダイレクト 1 からダイレクト 3 へデータベースを移行する際は一旦バックアップしてリストアするなどの手順が必要となります。

数値データ型の演算上の注意点

Firebird では、ダイレクト 1 とダイレクト 3 で、数値データ型の算術演算の結果が異なってきます。

- ダイレクト 1 では二つの整数型数値または二つの固定小数点型数値の除算の商は DOUBLE PRECISION 型の浮動小数点数値となります。
- ダイレクト 3 では、除算の商の小数点以下の有効桁数は、除数と被除数のスケールの合計となります。したがって、二つの整数型数値の除算の結果は整数型数値となります。
- 上記の結果として、1/3 を実行した場合、ダイレクト 1 では 0.3333333333333333e0 となりますが、ダイレクト 3 では 0 となってしまいます。整数型数値同士の除算では注意してください。

文字データ型

データ型	サイズ	範囲/有効桁数	説明
CHAR(n)	n 字	1~32,767 バイト キャラクタセットの文字サイズにより、32KB に納まる文字数。	固定長の CHAR 型またはテキスト文字列型。 CHARACTER キーワードも使用可。
VARCHAR(n)	n 文字	1 ~ 32,765 バイト。 キャラクタセットの文字サイズにより、32KB に納まる文字数。	可変長の CHAR 型 (テキスト文字列型)。 VARCHAR のかわりに、CHAR VARYING または、CHARACTER VARYING も使用できる。

Firebird では、上記の CHAR 型と VARCHAR 型の他に、NCHAR 型と NVARCHAR 型の 4 つの文字データ型をサポートしています。NCHAR 型・NVARCHAR 型は、ISO8859_1 キャラクタセットを使用する文字列型なので、本質的には固定長文字データ型と可変長文字データ型の 2 つだけをサポートしているといえます。

列のデータ型を指定する時に、CHARACTER SET オプションでキャラクタセットを指定することが出来ます。また、各キャラクタセットで定義されているコレーションオーダー（照合順序）を指定することが出来ます。

文字データ型で使用されるバイト数は、各キャラクタセットで必要とされるバイト数によって決まります。SJIS_0208 のような 2 バイトのキャラクタセットであれば、1 文字に 2 バイトが使用され、したがって最大文字数は 16,383 文字となります。これ以上の文字数を必要とする場合は、列を分けるか BLOB を使用してください。

- 固定長文字データ型では、指定された文字数に満たないデータについて Firebird は空白を付け加えて文字数を満たします。指定された文字数を超えたデータは切り捨てられます。デフォルトの文字数は 1 なので、CHAR は CHAR(1) と同じです。データの格納にあたっては、末尾の空白は圧縮されるので、CHAR 型が必ずしもディスクスペースを無駄にするわけではありません。
- 可変長文字データ型ではデフォルト値がないので、文字数 n は必ず指定しなくてはなりません。可変長文字データ型ではよりディスクスペースを節約できるので、検索が早くなりますが、挿入が遅くなる可能性が有ります。末尾に空白を付加したくない場合には、可変長文字データ型を使用してください。

日付時刻型

データ型	サイズ	範囲/有効桁数	説明
DATE	32 ビット	西暦 100 年 1 月 1 日～32768 年 2 月 29 日	年月日の日付情報のみを表す。
TIME	32 ビット	0:00 AM ～ 23:59.9999 PM	一日の午前零時からの時刻を 1/1000 秒単位で表す。
TIMESTAMP	64 ビット	西暦 100 年 1 月 1 日～32768 年 2 月 29 日	DATE と TIME の情報を組み合わせたデータ型。

Firebird では、DATE・TIME・TIMESTAMP の 3 つの日付時刻データ型をサポートしています。

- DATE は、日付を 32 ビットのデータとして格納します。西暦 100 年 1 月 1 日から 32768 年 2 月 29 日の範囲を有効な日付としてサポートします。
- TIME は、時刻を 32 ビットのデータとして格納します。00:00 AMから 23:59.9999 PM の範囲を有効な日付として認識します。最小単位は 1/1000 秒となります。
- TIMESTAMP は、2 つの 32 ビットのデータとして格納され、DATE と TIME を組み合わせたものとなっています。

BLOB データ型

データ型	サイズ	範囲/有効桁数	説明
BLOB	可変長		グラフィック、文字、音声データなどのサイズが動的に決まるデータ型。 内容はサブタイプによって決定される。

Firebird は動的にサイズの変更が可能な **BLOB (Binary Large Object)** データ型をサポートしています。BLOB データ型には、グラフィック、サウンド、動画、文字列などあらゆるデータを格納することが出来ます。また、**BLOB** フィルタを使用することによって、データの入出力に伴って形式の変換を行うことなども可能です。

APENDIX B
ALTER TABLE 文で有効に変換可能なデータ型一覧表

変換先												
変換元	BLOB	CHAR	DATE	Dec.	Dbl.	Flot.	Int.	Num.	Tstm	TIME	Sml.	Var.
BLOB												
CHAR		○										○
DATE		○	○						○			
Dec.		○		○				○				○
Dbl.		○			○	○						○
Flot.		○			○	○						○
Int.		○		○	○		○	○				○
Num		○						○				○
Tstm		○							○	○		
TIME		○							○	○		
Sml.		○		○	○	○	○	○			○	○
Var.		○										○

※各データ型はそれぞれ以下のように省略してあります。Dec.=DECIMAL、Dbl.=DOUBLE、Flot.=FLOAT、Int.=INTEGER、Num.=NUMERIC、Tstm.=TIMESTAMP、Sml.=SMALLINT、Var.=VARCHAR

※左端の列で変換元のデータ型を特定して、右へたどると変換可能なデータ型の列に○をつけてあります。

APENDIX C

Firebirdのキャラクタセットとコレーションオーダー

Firebird では、データベース・クライアントプログラム・CHAR 列・VARCHAR 列・テキスト型 BLOB 列に対して、キャラクタセットを指定することが出来ます。また、上記の各列に対してはその並び順（コレーションオーダー）を指定することも可能です。複数のコレーションオーダーをサポートするキャラクタセットも存在し、必要に応じて使い分けることができます。例えば、英単語の並び替えを行いたい場合、SJIS_0208 や ASCII などでは大文字小文字が区別されてしまい、ABC...abc...と並んでしまいます。その場合、IS08859_1 キャラクタセットでコレーションオーダー EN_US を指定して SELECT すると aAbBcC... とすることが出来ます。

!!FB!! IS08859_2 は Firebird 独自の拡張で、チェコ語に対応しています。また、WIN_1250 にコレーションオーダー PXW_HUN が追加されています、これは大文字小文字を区別しません。

この情報はシステムテーブル、RDBSCHARACTER_SETS と RDBSCOLLATIONS に格納されているので、以下の SELECT 文で表示させることが出来ます。

```
SELECT RCH.RDBSCHARACTER_SET_NAME, RCH.RDBSCHARACTER_SET_ID, RCH.RDBSBYTES_PER_CHARACTER,
       RCL.RDBSCOLLATION_NAME
FROM RDBSCHARACTER_SETS RCH, RDBSCOLLATIONS RCL
WHERE RCH.RDBSCHARACTER_SET_ID = RCL.RDBSCHARACTER_SET_ID
ORDER BY 1, 5;
```

キャラクタ セット	キャラクタ セット ID	最大文字サイズ	最小文字サイズ	コレーションオーダー
ASCII	2	1 バイト	1 バイト	ASCII
BIG_5	56	2 バイト	1 バイト	BIG_5
CYRL	50	1 バイト	1 バイト	CYRL
				DB_RUS
				PDOX_CYRL
DOS437	10	1 バイト	1 バイト	DOS437
				DB_DEU437
				DB_ESP437
				DB_FIN437
				DB_FRA437
				DB_ITA437
				DB_NLD437
				DB_SVE437
				DB_UK437
				DB_US437
				PDOX_ASCII
				PDOX_INTL
				PDOX_SWEDFIN
DOS850	11	1 バイト	1 バイト	DOS850
				DB_DEU850
				DB_ESP850
				DB_FRA850
				DB_FRC850
				DB_ITA850
				DB_NLD850
				DB_PTB850
				DB_SVE850
				DB_UK850
				DB_US850
DOS852	45	1 バイト	1 バイト	DOS852
				DB_CSY
				DB_PLK
				DB_SLO
				PDOX_CSY
				PDOX_HUN
				PDOX_PLK

				PDOX_SLO
DOS857	46	1 バイト	1 バイト	DOS857
				DB_TRK
DOS860	13	1 バイト	1 バイト	DOS860
				DB_PTG860
DOS861	47	1 バイト	1 バイト	DOS861
				PDOX_ISL
DOS863	14	1 バイト	1 バイト	DOS863
				DB_FRC863
DOS865	12	1 バイト	1 バイト	DOS865
				DB_DAN865
				DB_NOR865
				PDOX_NORDAN4
EUCJ_0208	62	1 バイト	1 バイト	EUCJ_0208
GB_2312	572	1 バイト	1 バイト	GB_2312
ISO8859_1	21	1 バイト	1 バイト	ISO8859_1
				DA_DA
				DE_DE
				DU_NL
				EN_UK
				EN_US
				ES_ES
				FI_FI
				FR_CA
				FR_FR
				IS_IS
				IT_IT
				NO_NO
				PT_PT
				V_SV
ISO8859_2	22	1 バイト	1 バイト	ISO8859_2
				CS_CZ
KSC_5601	442	1 バイト	1 バイト	KSC_5601
				KSC DICTIONARY
NEXT	56	1 バイト	1 バイト	NEXT
				NXT_DEU
				NXT_FRA
				NXT_ITA
				NXT_US
NONE	0	1 バイト	1 バイト	NONE
OCTETS	1	1 バイト	1 バイト	OCTETS
SJIS_0208	52	1 バイト	1 バイト	SJIS_0208
UNICODE_FSS	33	1 バイト	1 バイト	UNICODE_FSS
WIN1250	51	1 バイト	1 バイト	WIN1250
				PXW_CSY
				PXW_HUN
				PXW_HUNDC
				PXW_PLK
				PXW_SLO
WIN1251	52	1 バイト	1 バイト	WIN1251

				PXW_CYRL
WIN1252	53	1 バイト	1 バイト	WIN1252
				PXW_INTL
				PXW_INTL850
				PXW_NORDAN4
				PXW_SPAN
				PXW_SWEDFIN
WIN1253	54	1 バイト	1 バイト	WIN1253
				PXW_GREEK
WIN1254	55	1 バイト	1 バイト	WIN1254
				PXW_TURK