# 4D - SQLリファレンス

# チュートリアル

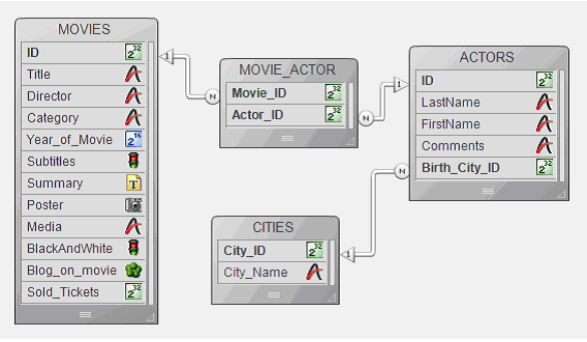
# 概要

SQLはコンピュータデータベースのデータを作成、整理、管理、取得するためのツールです。SQL自身はデータベース管理システムでも、スタンドアロンの製品でもありません。しかしながらSQLはデータベース管理システムのなかで欠くことができないものであり、言語およびシステムとの通信のために使用されるツールです。

このチュートリアルの目的は、4Dコード中でSQLを管理する方法や、SQLコマンドでデータを取り出す方法、引数の渡し方、SQLクエリ後の結果取得方法を示すことにあります。SQLそのものについては説明していません。これについてはインターネット上で有用な情報を入手することができます。

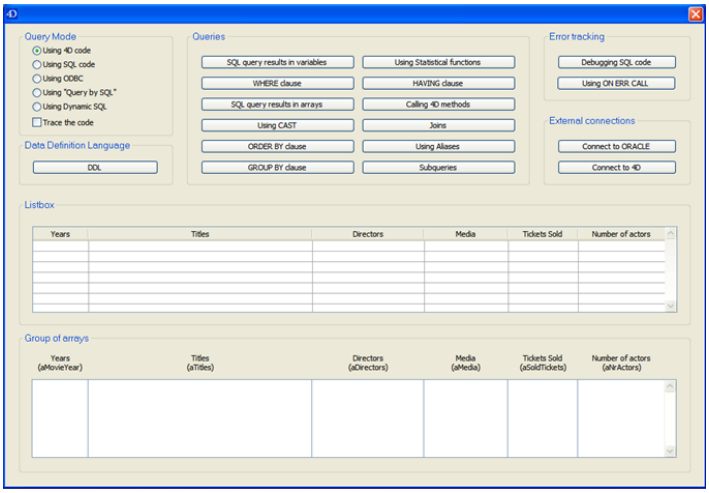
## このチュートリアルで使用される例題データベースについて

このドキュメントで使用されるすべての例題は"4D SQL Code Samples"というデータベースを使用しています。
ストラクチャーは以下のようになっています:




MOVIESテーブルにはタイトル、ディレクター、カテゴリー (アクション、アニメ、コメディ、サスペンス、ドラマその他)、公開年、サブタイトルの有無、概要、ポスターのピクチャ、メディアタイプ (DVD、VHS、DivX)、白黒、販売チケット数などによる50あまりの情報が登録されています。
ACTORSテーブルには映画の主演者に関 する、ID、名前、コメント、出演者の出身地IDなどの情報が格納されています。
CITIESテーブルには、都市の名前やIDなどの情報が格納さ れています。出演者の出身地として参照されます。
MOVIE_ACTORテーブルは、MOVIESとACTORSの多対多リレーションを表現する ために使用されます。

チュートリアル中で使用されるすべての例題は、以下のウィンドウからアクセスできます。このウィンドウは "**Demo SQL**" > "**Show Samples**" メ ニューで表示させます:



## SQLクエリの結果を変数に受け取る

---

まずとてもシンプルなクエリから始めましょう: ビデオライブラリにいくつ映画が登録されているかを取得します。4Dランゲージでは以下のようにコードを書きます:

```
C_LONGINT($AllMovies)
$AllMovies:=0
ALL RECORDS([MOVIES])
$AllMovies:=Records in selection([MOVIES])
ALERT("The Video Library contains "+String($AllMovies)+" movies")
```

- SQLエンジンを使用して同様のことを行う最初の方法は、クエリ文を**Begin SQL**と**End SQL**タグの間に置くことです。この方法では以下のようになります:

```
C_LONGINT($AllMovies)
$AllMovies:=0
Begin SQL
   SELECT COUNT(*)
   FROM MOVIES
   INTO <<$AllMovies>>
End SQL
ALERT("The Video Library contains "+String($AllMovies)+" movies")
```

- 御覧の通り、クエリの結果を変数にとることができます (今回の場合$AllMovies)。変数が"<<"と">>"で囲まれていることに注目してください。
有効な4D表現式 (変数、フィールド、配列、式など) を参照する別法は、表現式の前にコロン":"を置くことです:

```
C_LONGINT($AllMovies)
$AllMovies:=0
Begin SQL
   SELECT COUNT(*)
   FROM MOVIES
   INTO :$AllMovies
End SQL
ALERT("The Video Library contains "+String($AllMovies)+" movies")
```

インタープロセス変数の場合は少し異なる記法が必要なことに注意してください。インタープロセス変数は"["と"]"の間に記述しなければなりません:

```
C_LONGINT($AllMovies)
$AllMovies:=0
Begin SQL
   SELECT COUNT(*)
   FROM MOVIES
   INTO <<$AllMovies>>
End SQL
ALERT("The Video Library contains "+String($AllMovies)+" movies")
```

- SQLエンジンを使用する二つ目の方法は統合された汎用SQL (ODBC互換) コマンドを利用することです。この場合、コードは以下のようになります:

```
C_LONGINT($AllMovies)
$AllMovies:=0
` 内部SQLエンジンに接続を初期化する
SQL LOGIN(SQL_INTERNAL;"";"")
` クエリを実行し、$AllMovies変数に結果を返します。
SQL EXECUTE("SELECT COUNT(*) FROM MOVIES";$AllMovies)
` 見つかったすべてのレコードを取り出す
SQL LOAD RECORD(SQL_All_Records)
` 接続を閉じる
SQL LOGOUT
ALERT("The Video Library contains "+String($AllMovies)+" movies")
```

汎用SQLコマンドに関する詳細は4Dランゲージリファレンスマニュアルの**SQL**を参照してください。

- SQLエンジンを使用する三番目の方法は4Dの**QUERY BY SQL**コマンドを使用することです。この場合、コードは以下のようになります:

```
C_LONGINT($AllMovies)
$AllMovies:=0
QUERY BY SQL([MOVIES];"ID <> 0")
$AllMovies:=Records in selection([MOVIES])
ALERT("The Video Library contains "+String($AllMovies)+" movies")
```

実際のところ、**QUERY BY SQL**コマンドは以下のようなシンプルな*SELECT*クエリを実行するために使用できます:

```
SELECT *
FROM myTable
WHERE <SQL_Formula>
```

*myTable*は第一引数に渡されるテーブル名で、**SQL_Formula**は第二引数に渡されるクエリ文字列です:

```
QUERY BY SQL(myTable;SQL_Formula)
```

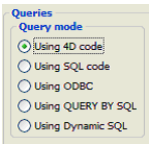
今回の場合**WHERE**句がありません。そこで"ID <> 0"を使用しています。同等のSQLは以下のようになります:

```
SELECT *
FROM MOVIES
WHERE ID <> 0
```

- SQLエンジンを使用する四番目の方法は、ダイナミックなSQLコマンド*EXECUTE IMMEDIATE*を使用することです。クエリは以下のようになります:

```
C_LONGINT($AllMovies)
$AllMovies:=0
C_TEXT($tQueryTxt)
$tQueryTxt:="SELECT COUNT(*) FROM MOVIES INTO :$AllMovies"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
ALERT("The Video Library contains "+String($AllMovies)+" movies")
```

以上の例題をテストするには、4D SQL Code Samplesデータベースを起動し、メインダイアログボックスを開きます。ダイアログの左にクエリモードを選択するエリアがあります: 標準の4Dコード、SQLコード、ODBCコマンド、**QUERY BY SQL**コマンド、またはダイナミックSQLから選択できます:




その後**SQL query results in variables**ボタンをクリックします。

### WHERE句の使用

---

ビデオライブラリから、1960年以降に公開された映画の数を調べる方法を見てみましょう。
4Dのコードは以下のようになります:

```
C_LONGINT($NoMovies)

$NoMovies:=0
REDUCE SELECTION([MOVIES];0)
QUERY([MOVIES];[MOVIES]Year_of_Movie>=1960)
$NoMovies:=Records in selection([MOVIES])
ALERT("The Video Library contains "+String($NoMovies)+" movies more recent or equal to 1960")
```

- SQLコードでは、同じクエリが以下のようになります:

```
C_LONGINT($NoMovies)

$NoMovies:=0
REDUCE SELECTION([MOVIES];0)
Begin SQL
   SELECT COUNT(*)
   FROM MOVIES
   WHERE Year_of_Movie >= 1960
   INTO :$NoMovies;
End SQL
ALERT("The Video Library contains "+String($NoMovies)+" movies more recent or equal to
1960")
```

- 汎用SQLコマンドを使用した場合、上のクエリは以下のようになります:

```
C_LONGINT($NoMovies)
$NoMovies:=0
REDUCE SELECTION([MOVIES];0)

SQL LOGIN(SQL_INTERNAL;"";"")
SQL EXECUTE("SELECT COUNT(*) FROM MOVIES WHERE Year_of_Movie >= 1960";$NoMovies)
SQL LOAD RECORD(SQL All Records)
SQL LOGOUT
ALERT("The Video Library contains "+String($NoMovies)+" movies more recent or equal to
1960")
```

- **QUERY BY SQL** コマンドを使用した場合、上のクエリは以下のようになります:

```
C_LONGINT($NoMovies)

$NoMovies:=0
REDUCE SELECTION([MOVIES];0)
QUERY BY SQL([MOVIES];"Year_of_Movie >= 1960")
$NoMovies:=Records in selection([MOVIES])
ALERT("The Video Library contains "+String($NoMovies)+" movies more recent or equal to
1960")
```

- SQLの *EXECUTE IMMEDIATE* コマンドを使用した場合、上のクエリは以下のようになります:

```
C_LONGINT($NoMovies)
C_TEXT($tQueryTxt)

$NoMovies:=0
REDUCE SELECTION([MOVIES];0)
$tQueryTxt:="SELECT COUNT(*) FROM MOVIES WHERE Year_of_Movie >= 1960 INTO :$NoMovies;"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
ALERT("The Video Libraontains "+String($NoMovies)+" movies more recent or equal to 1960")
```

前節のとおり、この例題をテストするには**4D SQL Code Samples**データベースを起動し、メインウィンドウを開きます。
クエリモードを選択して**WHERE clause**ボタンをクリックします。

## SQLクエリの結果を配列に受け取る

---

SQLクエリの引数に (年をハードコードするのではなく) 年を格納した変数を渡し、1960年以降に公開された映画の情報を取得する方法を見てみましょう。取得する情報はタイトルの他、公開年、ディレクター、メディア、販売チケット数です。取得した情報は配列やリストボックスに受け取ります。

- 4Dコードは以下のようになります:

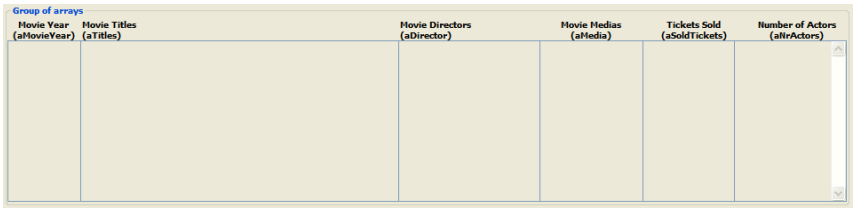
```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1960
QUERY([MOVIES];[MOVIES]Year_of_Movie>=$MovieYear)
SELECTION TO
ARRAY([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)
  ` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- SQLコードでは上記のクエリが以下のようになります:

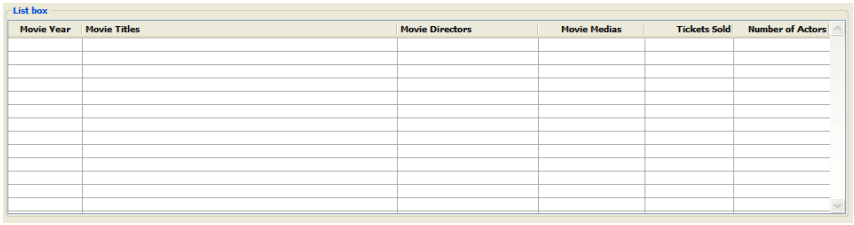
```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)

$MovieYear:=1960
Begin SQL
   SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets
   FROM MOVIES
   WHERE Year_of_Movie >= :$MovieYear
   INTO :aMovieYear, :aTitles, :aDirectors, :aMedias, :aSoldTickets;
End SQL
  ` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

ご覧頂いたとおり:

- 変数 ($MovieYear) を引数としてSQLクエリに渡すことができます。値を受け取る際と同じ記法です。
- SQLクエリ結果は配列aMovieYear、aTitles、aDirectories、aMedias、そしてaSoldTicketsに格納されます。この結果はメインウィンドウ中で二つの方法で表示されます:
  - グループ化されたスクロールエリア:




  - 同じ名称の列名を持つリストボックス:




- 汎用SQLコマンドの場合:

```
ARRAY LONGINT(aSoldTickets;0)
```

```
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)
C_TEXT($tQueryTxt)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1960
SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
SQL EXECUTE($tQueryTxt;aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets)
SQL LOAD RECORD(SQL All Records)
SQL LOGOUT
` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- **QUERY BY SQL**コマンドの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1960
QUERY BY SQL([MOVIES];"Year_of_Movie >= :$MovieYear")
SELECTION TO
ARRAY([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)
` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- SQLの*EXECUTE IMMEDIATE*コマンドを使用した場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)
C_TEXT($tQueryTxt)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1960
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aTitles, :aDirectors, :aMedias,
:aSoldTickets;"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

これらをテストするには、**4D SQL Code Samples**データベースを起動してメインウィンドウを表示させます。クエリモードを選択して**SQL query results in arrays**ボタンをクリックします。

## CASTの使用

---

SQL標準は、式の中で異なるタイプのデータを結合させることについて、極めて厳しいルールを定めています。通常DBMSが自動変換を行います。しかしSQL標準は、数値と文字データの比較を行おうとした場合、DBMSはエラーを生成しなければならないとしています。このコンテキストにおいて、*CAST*式はとても重要です。使用するプログラムのデータ型が、SQL標準でサポートされる型と一致しない場合は特にそうです。
**SQLクエリの結果を配列に受け取る**のクエリを*CAST*式を使用するよう若干変更したものを以下に示します。

- 4Dコードの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=Num("1960")
QUERY([MOVIES];[MOVIES]Year_of_Movie>=$MovieYear)
SELECTION TO
ARRAY([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)
  ` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- SQLコードの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)

Begin SQL
   SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets
   FROM MOVIES
   WHERE Year_of_Movie >= CAST('1960' AS INT)
   INTO :aMovieYear, :aTitles, :aDirectors, :aMedias, :aSoldTickets;
End SQL
  ` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- 汎用SQLコマンドの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_TEXT($tQueryTxt)
  
REDUCE SELECTION([MOVIES];0)
SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= CAST('1960' AS INT)"
SQL EXECUTE($tQueryTxt;aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets)
SQL LOAD RECORD(SQL All Records)
SQL LOGOUT
  ` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- **QUERY BY SQL**コマンドの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
```

```
ARRAY TEXT(aMedias;0)

REDUCE SELECTION([MOVIES];0)
QUERY BY SQL([MOVIES];"Year_of_Movie >= CAST('1960' AS INT)")
SELECTION TO
ARRAY([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)
` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- SQLの*EXECUTE IMMEDIATE*コマンドの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_TEXT($tQueryTxt)

REDUCE SELECTION([MOVIES];0)
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= CAST('1960' AS INT)"
$tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aTitles, :aDirectors, :aMedias,
:aSoldTickets;"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

これらをテストするには、**4D SQL Code Samples**データベースを起動してメインウィンドウを表示させます。クエリモードを選択して**Using CAST**ボタンをクリックします。

## ORDER BY句の使用

---

ここでは、1960年以降に公開されたすべての映画に対し、公開年、タイトル、ディレクタ、メディア、チケット販売数の情報を取得します。結果は公開年で並び替えます。

- 4Dコードは以下のようになります:

```
ARRAY LONGINT(aNrActors;0)
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1960
QUERY([MOVIES];[MOVIES]Year_of_Movie>=$MovieYear)
SELECTION TO
ARRAY([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)
SORT ARRAY(aMovieYear;aTitles;aDirectors;aMedias;>)
```

- SQLコードの場合:

```
ARRAY LONGINT(aNrActors;0)
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1960
Begin SQL
   SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets
   FROM MOVIES
   WHERE Year_of_Movie >= :$MovieYear
   ORDER BY 1
   INTO :aMovieYear, :aTitles, :aDirectors, :aMedias,  :aSoldTickets;
End SQL
```

- 汎用SQLコマンドの場合:

```
C_TEXT($tQueryTxt)
ARRAY LONGINT(aNrActors;0)
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)
      
REDUCE SELECTION([MOVIES];0)
$MovieYear:=1960
SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
SQL EXECUTE($tQueryTxt;aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets)
SQL LOAD RECORD(SQL All Records)
SQL LOGOUT
```

- **QUERY BY SQL**コマンドの場合:

```
ARRAY LONGINT(aNrActors;0)
```

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)
    
REDUCE SELECTION([MOVIES];0)
$MovieYear:=1960
QUERY BY SQL([MOVIES];"Year_of_Movie >= :$MovieYear")
SELECTION TO
ARRAY([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)
SORT ARRAY(aMovieYear;aTitles;aDirectors;aMedias;>)
```

- SQLの*EXECUTE IMMEDIATE*コマンドの場合:

```
ARRAY LONGINT(aNrActors;0)
C_TEXT($tQueryTxt)
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1960
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
$tQueryTxt:=$tQueryTxt+"  INTO :aMovieYear, :aTitles, :aDirectors, :aMedias,
:aSoldTickets;"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
```

これらをテストするには、**4D SQL Code Samples**データベースを起動してメインウィンドウを表示させます。クエリモードを選択して**ORDER BY clause**ボタンをクリックします。

## GROUP BY句の使用

---

ここでは1979年以降、販売されたチケット数に関する情報を取得します。結果は年で並び替えます。
これを行うために、1979年以降に公開された映画の販売チケット数を年ごとに計算します。そして結果を年で並び替えます。

- 4Dコードの場合、以下のようになります:

```
  ` 標準の4Dコードを使用
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aSoldTickets;0)
C_LONGINT($MovieYear;$vCrtMovieYear;$i)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1979
QUERY([MOVIES];[MOVIES]Year_of_Movie>=$MovieYear)
ORDER BY([MOVIES];[MOVIES]Year_of_Movie;>)
$vCrtMovieYear:=0
$vInd:=Size of array(aMovieYear)
For($i;1;Records in selection([MOVIES]))
   If([MOVIES]Year_of_Movie#$vCrtMovieYear)
      $vCrtMovieYear:=[MOVIES]Year_of_Movie
      $vInd:=$vInd+1
      INSERT IN ARRAY(aMovieYear;$vInd;1)
      aMovieYear{$vInd}:=$vCrtMovieYear
      INSERT IN ARRAY(aSoldTickets;$vInd;1)
   End if
   aSoldTickets{$vInd}:=aSoldTickets{$vInd}+[MOVIES]Sold_Tickets
   NEXT RECORD([MOVIES])
End for
  ` 情報を表示するために、リストボックスの残りの列を初期化
ARRAY TEXT(aTitles;Size of array(aMovieYear))
ARRAY TEXT(aDirectors;Size of array(aMovieYear))
ARRAY TEXT(aMedias;Size of array(aMovieYear))
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- SQLコードの場合、以下のようになります:

```
  ` 4D SQLを使用
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aSoldTickets;0)
C_LONGINT($MovieYear)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1979
Begin SQL
   SELECT Year_of_Movie, SUM(Sold_Tickets)
   FROM MOVIES
   WHERE Year_of_Movie >= :$MovieYear
   GROUP BY Year_of_Movie
   ORDER BY 1
   INTO :aMovieYear,  :aSoldTickets;
End SQL
  ` 情報を表示するために、リストボックスの残りの列を初期化
ARRAY TEXT(aTitles;Size of array(aMovieYear))
ARRAY TEXT(aDirectors;Size of array(aMovieYear))
ARRAY TEXT(aMedias;Size of array(aMovieYear))
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- 汎用SQLコマンドの場合、以下のようになります:

```
  ` ODBCコマンドを使用
C_TEXT($tQueryTxt)
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
C_LONGINT($MovieYear)
 
REDUCE SELECTION([MOVIES];0)
$MovieYear:=1979
```

```
SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" GROUP BY Year_of_Movie"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
SQL EXECUTE($tQueryTxt;aMovieYear;aSoldTickets)
SQL LOAD RECORD(SQL All Records)
SQL LOGOUT
` 情報を表示するために、リストボックスの残りの列を初期化
ARRAY TEXT(aTitles;Size of array(aMovieYear))
ARRAY TEXT(aDirectors;Size of array(aMovieYear))
ARRAY TEXT(aMedias;Size of array(aMovieYear))
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- **QUERY BY SQL**コマンドの場合、以下のようになります:

```
` QUERY BY SQLを使用
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
C_LONGINT($MovieYear)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1979
QUERY BY SQL([MOVIES];"Year_of_Movie >= :$MovieYear")
ORDER BY([MOVIES];[MOVIES]Year_of_Movie;>)
$vCrtMovieYear:=0
$vInd:=Size of array(aMovieYear)
For($i;1;Records in selection([MOVIES]))
   If([MOVIES]Year_of_Movie#$vCrtMovieYear)
      $vCrtMovieYear:=[MOVIES]Year_of_Movie
      $vInd:=$vInd+1
      INSERT IN ARRAY(aMovieYear;$vInd;1)
      aMovieYear{$vInd}:=$vCrtMovieYear
      INSERT IN ARRAY(aSoldTickets;$vInd;1)
   End if
   aSoldTickets{$vInd}:=aSoldTickets{$vInd}+[MOVIES]Sold_Tickets
   NEXT RECORD([MOVIES])
End for
` 情報を表示するために、リストボックスの残りの列を初期化
ARRAY TEXT(aTitles;Size of array(aMovieYear))
ARRAY TEXT(aDirectors;Size of array(aMovieYear))
ARRAY TEXT(aMedias;Size of array(aMovieYear))
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- SQLの*EXECUTE IMMEDIATE*コマンドの場合、以下のようになります:

```
` EXECUTE IMMEDIATE動的なSQLを使用
C_TEXT($tQueryTxt)
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
C_LONGINT($MovieYear)

$MovieYear:=1979
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" GROUP BY Year_of_Movie"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
$tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aSoldTickets;"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
` 情報を表示するために、リストボックスの残りの列を初期化
ARRAY TEXT(aTitles;Size of array(aMovieYear))
ARRAY TEXT(aDirectors;Size of array(aMovieYear))
ARRAY TEXT(aMedias;Size of array(aMovieYear))
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

これらをテストするには、"**4D SQL Code Samples**"データベースを起動してメインウィンドウを表示させます。クエリ

モードを選択して "**GROUP BY clause**" ボタンをクリックします。

## 統計関数の使用

---

ときに特定の値について統計を取りたくなることがあります。SQLには*MIN*, *MAX*, *AVG*, *SUM*などの集約関数が用意されています。集約関数を使用して、年ごとのチケット販売数の情報を取得します。結果は年でソートされます。
これを行うには、映画ごとのすべてのチケット販売異数を総計し、結果を年で並び替えます。

- 4Dコードの場合、以下のようになります:

```
C_LONGINT($vMin;$vMax;$vSum)
C_REAL($vAverage)
C_TEXT($AlertTxt)

REDUCE SELECTION([MOVIES];0)
$vMin:=0
$vMax:=0
$vAverage:=0
$vSum:=0
ALL RECORDS([MOVIES])
$vMin:=Min([MOVIES]Sold_Tickets)
$vMax:=Max([MOVIES]Sold_Tickets)
$vAverage:=Average([MOVIES]Sold_Tickets)
$vSum:=Sum([MOVIES]Sold_Tickets)
```

- SQLコードの場合:

```
C_LONGINT($vMin;$vMax;$vSum)
C_REAL($vAverage)
C_TEXT($AlertTxt)
  
$vMin:=0
$vMax:=0
$vAverage:=0
$vSum:=0
Begin SQL
   SELECT MIN(Sold_Tickets),
   MAX(Sold_Tickets),
   AVG(Sold_Tickets),
   SUM(Sold_Tickets)
   FROM MOVIES
   INTO :$vMin, :$vMax, :$vAverage, :$vSum;
End SQL
```

- 汎用SQLコマンドの場合:

```
C_LONGINT($vMin;$vMax;$vSum)
C_REAL($vAverage)
C_TEXT($tQueryTxt)
C_TEXT($AlertTxt)
  
$vMin:=0
$vMax:=0
$vAverage:=0
$vSum:=0
SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT MIN(Sold_Tickets), MAX(Sold_Tickets), AVG(Sold_Tickets),
SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
SQL EXECUTE($tQueryTxt;$vMin;$vMax;$vAverage;$vSum)
SQL LOAD RECORD(SQL_All_Records)
SQL LOGOUT
```

- SQLの*EXECUTE IMMEDIATE*コマンドの場合:

```
C_LONGINT($vMin;$vMax;$vSum)
C_REAL($vAverage)
C_TEXT($tQueryTxt)
C_TEXT($AlertTxt)
```

```
$vMin:=0
$vMax:=0
$vAverage:=0
$vSum:=0
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT MIN(Sold_Tickets), MAX(Sold_Tickets), AVG(Sold_Tickets),
SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" INTO :$vMin, :$vMax, :$vAverage, :$vSum;"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
```

これらをテストするには、**4D SQL Code Samples**データベースを起動してメインウィンドウを表示させます。クエリモードを選択して**Using Aggregate functions**ボタンをクリックします。

### HAVING句の使用

---

ここでは1979年以降、販売されたチケットの総数を年ごとに取得します。ただし販売数が10,000,000を上回る年は除きます。結果は年で並び替えられます。
これを行うために、1979年以降の年ごとにすべての映画のチケット総販売数を計算し、販売数が10,000,000を上回る年を取り除き、年で並び替えます。

- 4Dコードによるクエリは以下のようになります:

```
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aSoldTickets;0)
C_LONGINT($MovieYear;$vCrtMovieYear;$i;$MinSoldTickets;$vInd)
 
REDUCE SELECTION([MOVIES];0)
$MovieYear:=1979
$MinSoldTickets:=10000000
QUERY([MOVIES];[MOVIES]Year_of_Movie>=$MovieYear)
ORDER BY([MOVIES];[MOVIES]Year_of_Movie;>)
$vCrtMovieYear:=0
$vInd:=Size of array(aMovieYear)
For($i;1;Records in selection([MOVIES]))
   If([MOVIES]Year_of_Movie#$vCrtMovieYear)
      $vCrtMovieYear:=[MOVIES]Year_of_Movie
      If(aSoldTickets{$vInd}<$MinSoldTickets)
         $vInd:=$vInd+1
         INSERT IN ARRAY(aMovieYear;$vInd;1)
         aMovieYear{$vInd}:=$vCrtMovieYear
         INSERT IN ARRAY(aSoldTickets;$vInd;1)
      Else
         aSoldTickets{$vInd}:=0
      End if
   End if
   aSoldTickets{$vInd}:=aSoldTickets{$vInd}+[MOVIES]Sold_Tickets
   NEXT RECORD([MOVIES])
End for
If(aSoldTickets{$vInd}>=$MinSoldTickets)
   DELETE FROM ARRAY(aSoldTickets;$vInd;1)
   DELETE FROM ARRAY(aMovieYear;$vInd;1)
End if
 // 情報を表示するために、残りのリストボックス列を初期化します
ARRAY TEXT(aTitles;Size of array(aMovieYear))
ARRAY TEXT(aDirectors;Size of array(aMovieYear))
ARRAY TEXT(aMedias;Size of array(aMovieYear))
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- SQLコードを使用すると、上のクエリは以下のようになります:

```
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aSoldTickets;0)
C_LONGINT($MovieYear;$MinSoldTickets)

$MovieYear:=1979
$MinSoldTickets:=10000000
Begin SQL
   SELECT Year_of_Movie, SUM(Sold_Tickets)
   FROM MOVIES
   WHERE Year_of_Movie >= :$MovieYear
   GROUP BY Year_of_Movie
   HAVING SUM(Sold_Tickets) < :$MinSoldTickets
   ORDER BY 1
   INTO :aMovieYear,  :aSoldTickets;
End SQL
 // 情報を表示するために、残りのリストボックス列を初期化します
ARRAY TEXT(aTitles;Size of array(aMovieYear))
ARRAY TEXT(aDirectors;Size of array(aMovieYear))
ARRAY TEXT(aMedias;Size of array(aMovieYear))
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- 汎用SQLコマンドを使用すると、上のクエリは以下のようになります:

```
C_TEXT($tQueryTxt)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aSoldTickets;0)
C_LONGINT($MovieYear;$MinSoldTickets)
 
$MovieYear:=1979
$MinSoldTickets:=10000000
SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" GROUP BY Year_of_Movie"
$tQueryTxt:=$tQueryTxt+" HAVING SUM(Sold_Tickets) < :$MinSoldTickets"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
SQL EXECUTE($tQueryTxt;aMovieYear;aSoldTickets)
SQL LOAD RECORD(SQL_All_Records)
SQL LOGOUT
 // 情報を表示するために、残りのリストボックス列を初期化します
ARRAY TEXT(aTitles;Size of array(aMovieYear))
ARRAY TEXT(aDirectors;Size of array(aMovieYear))
ARRAY TEXT(aMedias;Size of array(aMovieYear))
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- **QUERY BY SQL**コマンドを使用すると、上のクエリは以下のようになります:

```
C_TEXT($tQueryTxt)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aSoldTickets;0)
C_LONGINT($MovieYear;$MinSoldTickets;$vCrtMovieYear;$vInd;$i)
    
REDUCE SELECTION([MOVIES];0)
$MovieYear:=1979
$MinSoldTickets:=10000000
QUERY BY SQL([MOVIES];"Year_of_Movie >= :$MovieYear")
ORDER BY([MOVIES];[MOVIES]Year_of_Movie;>)
$vCrtMovieYear:=0
$vInd:=Size of array(aMovieYear)
For($i;1;Records in selection([MOVIES]))
   If([MOVIES]Year_of_Movie#$vCrtMovieYear)
      $vCrtMovieYear:=[MOVIES]Year_of_Movie
      If(aSoldTickets{$vInd}<$MinSoldTickets)
         $vInd:=$vInd+1
         INSERT IN ARRAY(aMovieYear;$vInd;1)
         aMovieYear{$vInd}:=$vCrtMovieYear
         INSERT IN ARRAY(aSoldTickets;$vInd;1)
      Else
         aSoldTickets{$vInd}:=0
      End if
   End if
   aSoldTickets{$vInd}:=aSoldTickets{$vInd}+[MOVIES]Sold_Tickets
   NEXT RECORD([MOVIES])
End for
If(aSoldTickets{$vInd}>=$MinSoldTickets)
   DELETE FROM ARRAY(aSoldTickets;$vInd;1)
   DELETE FROM ARRAY(aMovieYear;$vInd;1)
End if
 // 情報を表示するために、残りのリストボックス列を初期化します
ARRAY TEXT(aTitles;Size of array(aMovieYear))
ARRAY TEXT(aDirectors;Size of array(aMovieYear))
ARRAY TEXT(aMedias;Size of array(aMovieYear))
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- SQLの*EXECUTE IMMEDIATE*コマンドを使用すると、上のクエリは以下のようになります:

```
C_TEXT($tQueryTxt)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aSoldTickets;0)
C_LONGINT($MovieYear;$MinSoldTickets)

$MovieYear:=1979
$MinSoldTickets:=10000000
$tQueryTxt:=""
```

```
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" GROUP BY Year_of_Movie"
$tQueryTxt:=$tQueryTxt+" HAVING SUM(Sold_Tickets) < :$MinSoldTickets"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
$tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aSoldTickets;"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
 // 情報を表示するために、残りのリストボックス列を初期化します
ARRAY TEXT(aTitles;Size of array(aMovieYear))
ARRAY TEXT(aDirectors;Size of array(aMovieYear))
ARRAY TEXT(aMedias;Size of array(aMovieYear))
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

これらをテストするには、"4D SQL Code Samples"データベースを起動してメインウィンドウを表示させます。そしてクエリモードを選択して**HAVING clause**ボタンをクリックします。

## SQLコードから4Dメソッドを呼び出す

---

ここでは、映画ごとに出演者に関する何らかの情報を取得します。たとえば最低7人の出演者がいる映画を探すといったようなものです。結果は年で並び替えられます。
これを行うために、4Dメソッド (Find_Nr_Of_Actors) を使用します。このメソッドは映画のIDを受け取り、その映画の出演者数を返します:

```
  `(F) Find_Nr_Of_Actors
C_LONGINT($0;$1;$vMovie_ID)
$vMovie_ID:=$1

QUERY([MOVIE_ACTOR];[MOVIE_ACTOR]Movie_ID=$vMovie_ID)
$0:=Records in selection([MOVIE_ACTOR])
```

- 4Dコードは以下のようになります:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aNrActors;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($NrOfActors;$i;$vInd)

$vInd:=0
$NrOfActors:=7
ALL RECORDS([MOVIES])
For($i;1;Records in selection([MOVIES]))
   $vCrtActors:=Find_Nr_Of_Actors([MOVIES]ID)
   If($vCrtActors>=$NrOfActors)
      $vInd:=$vInd+1
      INSERT IN ARRAY(aMovieYear;$vInd;1)
      aMovieYear{$vInd}:=[MOVIES]Year_of_Movie
      INSERT IN ARRAY(aTitles;$vInd;1)
      aTitles{$vInd}:=[MOVIES]Title
      INSERT IN ARRAY(aDirectors;$vInd;1)
      aDirectors{$vInd}:=[MOVIES]Director
      INSERT IN ARRAY(aMedias;$vInd;1)
      aMedias{$vInd}:=[MOVIES]Media
      INSERT IN ARRAY(aSoldTickets;$vInd;1)
      aSoldTickets{$vInd}:=[MOVIES]Sold_Tickets
      INSERT IN ARRAY(aNrActors;$vInd;1)
      aNrActors{$vInd}:=$vCrtActors
   End if
   NEXT RECORD([MOVIES])
End for
SORT ARRAY(aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets;aNrActors;>)
```

- SQLコードの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aNrActors;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($NrOfActors;$i;$vInd)

$vInd:=0
$NrOfActors:=7
Begin SQL
   SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets, {fn Find_Nr_Of_Actors(ID)
AS NUMERIC}
   FROM MOVIES
   WHERE {fn Find_Nr_Of_Actors(ID) AS NUMERIC} >= :$NrOfActors
   ORDER BY 1
   INTO :aMovieYear, :aTitles, :aDirectors, :aMedias, :aSoldTickets, :aNrActors;
End SQL
```

ここで示すとおり、SQLコード内部で以下のシンタックスを使用することにより4Dメソッドを呼び出すことができます:

```
{fn 4DFunctionName AS 4DFunctionResultType}
```

- 汎用SQLコマンドの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aNrActors;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($NrOfActors;$i;$vInd)
C_TEXT($tQueryTxt)

$vInd:=0
$NrOfActors:=7
SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets, {fn
Find_Nr_Of_Actors(ID) AS NUMERIC}"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE {fn Find_Nr_Of_Actors(ID) AS NUMERIC} >= :$NrOfActors"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
SQL EXECUTE($tQueryTxt;aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets;aNrActors)
SQL LOAD RECORD(SQL All Records)
SQL LOGOUT
```

- **QUERY BY SQL**コマンドの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aNrActors;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($NrOfActors;$i;$vInd)

$vInd:=0
$NrOfActors:=7
QUERY BY SQL([MOVIES];"{fn Find_Nr_Of_Actors(ID) AS NUMERIC} >= :$NrOfActors")
For($i;1;Records in selection([MOVIES]))
   $vInd:=$vInd+1
   INSERT IN ARRAY(aMovieYear;$vInd;1)
   aMovieYear{$vInd}:=[MOVIES]Year_of_Movie
   INSERT IN ARRAY(aTitles;$vInd;1)
   aTitles{$vInd}:=[MOVIES]Title
   INSERT IN ARRAY(aDirectors;$vInd;1)
   aDirectors{$vInd}:=[MOVIES]Director
   INSERT IN ARRAY(aMedias;$vInd;1)
   aMedias{$vInd}:=[MOVIES]Media
   INSERT IN ARRAY(aSoldTickets;$vInd;1)
   aSoldTickets{$vInd}:=[MOVIES]Sold_Tickets
   INSERT IN ARRAY(aNrActors;$vInd;1)
   aNrActors{$vInd}:=Find_Nr_Of_Actors([MOVIES]ID)
   NEXT RECORD([MOVIES])
End for
SORT ARRAY(aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets;aNrActors;>)
```

- SQLの*EXECUTE IMMEDIATE*コマンドの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aNrActors;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($NrOfActors;$i;$vInd)
C_TEXT($tQueryTxt)

$vInd:=0
```

```
 $NrOfActors:=7
 $tQueryTxt:=""
 $tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets, {fn
Find_Nr_Of_Actors(ID) AS NUMERIC}"
 $tQueryTxt:=$tQueryTxt+" FROM MOVIES"
 $tQueryTxt:=$tQueryTxt+" WHERE {fn Find_Nr_Of_Actors(ID) AS NUMERIC} >= :$NrOfActors"
 $tQueryTxt:=$tQueryTxt+" ORDER BY 1"
 $tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aTitles, :aDirectors, :aMedias,
:aSoldTickets,"+" :aNrActors;"
 Begin SQL
    EXECUTE IMMEDIATE :$tQueryTxt;
 End SQL
```

これらをテストするには"**4D SQL Code Samples**"データベースを起動してメインウィンドウを表示させます。クエリモードを選択して **Calling 4D methods**ボタンをクリックします。

## 結合

---

ここでは、出演者ごとの出身地を取得します。出演者のリストはACTORSテーブルにあり、都市のリストはCITIESテーブルにあります。このク エリを実行するには、ACTORSとCITIES二つのテーブルを結合しなければなりません。

- 4Dコードは以下のようになります:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
C_LONGINT($i;$vInd)

$vInd:=0
ALL RECORDS([ACTORS])
For($i;1;Records in selection([ACTORS]))
   $vInd:=$vInd+1
   INSERT IN ARRAY(aTitles;$vInd;1)
   aTitles{$vInd}:=[ACTORS]FirstName+" "+[ACTORS]LastName
   RELATE ONE([ACTORS]Birth_City_ID)
   INSERT IN ARRAY(aDirectors;$vInd;1)
   aDirectors{$vInd}:=[CITIES]City_Name
   NEXT RECORD([ACTORS])
End for
` 情報を表示するために他のリストボックス列を初期化します
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aSoldTickets;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
MULTI SORT ARRAY(aDirectors;>;aTitles;>;aMovieYear;aMedias;aSoldTickets;aNrActors)
```

- SQLコードの場合:

```
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)

Begin SQL
   SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName), CITIES.City_Name
   FROM ACTORS, CITIES
   WHERE ACTORS.Birth_City_ID=CITIES.City_ID
   ORDER BY 2,1
    INTO :aTitles, :aDirectors;
End SQL
` 情報を表示するために他のリストボックス列を初期化します
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aSoldTickets;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
```

- 汎用SQLコマンドの場合:

```
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
C_TEXT($tQueryTxt)

SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName),
CITIES.City_Name"
$tQueryTxt:=$tQueryTxt+" FROM ACTORS, CITIES"
$tQueryTxt:=$tQueryTxt+" WHERE ACTORS.Birth_City_ID=CITIES.City_ID"
$tQueryTxt:=$tQueryTxt+" ORDER BY 2,1"
SQL EXECUTE($tQueryTxt;aTitles;aDirectors)
SQL LOAD RECORD(SQL All Records)
SQL LOGOUT
` 情報を表示するために他のリストボックス列を初期化します
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aSoldTickets;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
```

- **QUERY BY SQL**コマンドでは、このクエリを実行できません。なぜなら このコマンドの一番目の引数に二つ以上のテーブルを渡せないためです。
- SQLの*EXECUTE IMMEDIATE*コマンドの場合:

```
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
C_TEXT($tQueryTxt)

$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName),
CITIES.City_Name"
$tQueryTxt:=$tQueryTxt+" FROM ACTORS, CITIES"
$tQueryTxt:=$tQueryTxt+" WHERE ACTORS.Birth_City_ID=CITIES.City_ID"
$tQueryTxt:=$tQueryTxt+" ORDER BY 2,1"
$tQueryTxt:=$tQueryTxt+" INTO :aTitles, :aDirectors"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
` 情報を表示するために他のリストボックス列を初期化します
 ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aSoldTickets;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
```

これらをテストするに は、**4D SQL Code Samples**データベースを起動してメインウィンドウを表示させます。クエリモードを選択して**Joins** ボタンをクリックします。

## エイリアスの使用

---

SQLクエリがとても複雑で、読み難いほどに長い名前を含む場合、可読性を向上させるためエイリアスを使用できます。
以下は、先の例を二つのエイリアスを使用して書き直したものです。ACTORSテーブルのエイリアスがAct、CITIESテーブルのエイリアスが Citです。

- 4Dコードは以下のとおりです:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
C_LONGINT($i;$vInd)

$vInd:=0
ALL RECORDS([ACTORS])
For($i;1;Records in selection([ACTORS]))
   $vInd:=$vInd+1
   INSERT IN ARRAY(aTitles;$vInd;1)
   aTitles{$vInd}:=[ACTORS]FirstName+" "+[ACTORS]LastName
   RELATE ONE([ACTORS]Birth_City_ID)
   INSERT IN ARRAY(aDirectors;$vInd;1)
   aDirectors{$vInd}:=[CITIES]City_Name
   NEXT RECORD([ACTORS])
End for
` 情報を表示するために他のリストボックス列を初期化します
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aSoldTickets;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
MULTI SORT ARRAY(aDirectors;>;aTitles;>;aMovieYear;aMedias;aSoldTickets;aNrActors)
```

- SQLの場合:

```
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)

Begin SQL
   SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName), CITIES.City_Name
   FROM ACTORS AS 'Act', CITIES AS 'Cit'
   WHERE Act.Birth_City_ID=Cit.City_ID
   ORDER BY 2,1
   INTO :aTitles, :aDirectors;
End SQL
` 情報を表示するために他のリストボックス列を初期化します
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aSoldTickets;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
```

- 汎用SQLコマンドの場合:

```
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
C_TEXT($tQueryTxt)

SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName),
CITIES.City_Name"
$tQueryTxt:=$tQueryTxt+" FROM ACTORS AS 'Act', CITIES AS 'Cit'"
$tQueryTxt:=$tQueryTxt+" WHERE Act.Birth_City_ID=Cit.City_ID"
$tQueryTxt:=$tQueryTxt+" ORDER BY 2,1"
SQL EXECUTE($tQueryTxt;aTitles;aDirectors)
SQL LOAD RECORD(SQL All Records)
SQL LOGOUT
` 情報を表示するために他のリストボックス列を初期化します
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aSoldTickets;Size of array(aTitles))
```

```
ARRAY LONGINT(aNrActors;Size of array(aTitles))
```

- **QUERY BY SQL**コマンドでは、このクエリを実行できません。なぜならこのコマンドの一番目の引数に二つ以上のテーブルを渡せないためです。
- SQLの*EXECUTE IMMEDIATE*コマンドの場合:

```
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
C_TEXT($tQueryTxt)
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName),
CITIES.City_Name"
$tQueryTxt:=$tQueryTxt+" FROM ACTORS AS 'Act', CITIES AS 'Cit'"
$tQueryTxt:=$tQueryTxt+" WHERE Act.Birth_City_ID=Cit.City_ID"
$tQueryTxt:=$tQueryTxt+" ORDER BY 2,1"
$tQueryTxt:=$tQueryTxt+" INTO :aTitles, :aDirectors"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aSoldTickets;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
```

これらをテストするには、**4D SQL Code Samples**データベースを起動してメインウィンドウを表示させます。クエリモードを選択して**Using Aliases**ボタンをクリックします。

## サブクエリ

---

ここではチケット販売に関する統計情報を取得します。チケットの総販売平均よりもチケットが売れた映画を取得します。このクエリをSQLで実行するために、クエリの中でクエリを行います。これをサブクエリと呼びます。

- 4Dコードは以下のようになります:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY TEXT(aTitles;0)
C_LONGINT($i;$vInd;$vAvgSoldTickets)

$vInd:=0
ALL RECORDS([MOVIES])
$vAvgSoldTickets:=Average([MOVIES]Sold_Tickets)
For($i;1;Records in selection([MOVIES]))
   If([MOVIES]Sold_Tickets>$vAvgSoldTickets)
      $vInd:=$vInd+1
      INSERT IN ARRAY(aTitles;$vInd;1)
      aTitles{$vInd}:=[MOVIES]Title
      INSERT IN ARRAY(aSoldTickets;$vInd;1)
      aSoldTickets{$vInd}:=[MOVIES]Sold_Tickets
   End if
   NEXT RECORD([MOVIES])
End for
` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aDirectors;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
SORT ARRAY(aTitles;aDirectors;aMovieYear;aMedias;aSoldTickets;aNrActors;>)
```

- SQLの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY TEXT(aTitles;0)
Begin SQL
   SELECT Title, Sold_Tickets
   FROM MOVIES
   WHERE Sold_Tickets > (SELECT AVG(Sold_Tickets) FROM MOVIES)
   ORDER BY 1
   INTO :aTitles, :aSoldTickets;
End SQL
` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aDirectors;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
SORT ARRAY(aTitles;aDirectors;aMovieYear;aMedias;aSoldTickets;aNrActors;>)
```

- 汎用SQLコマンドの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY TEXT(aTitles;0)
C_TEXT($tQueryTxt)

SQL LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Title, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Sold_Tickets > (SELECT AVG(Sold_Tickets) FROM MOVIES)"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
SQL EXECUTE($tQueryTxt;aTitles;aSoldTickets)
SQL LOAD RECORD(SQL All Records)
SQL LOGOUT
` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aDirectors;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
```

```
SORT ARRAY(aTitles;aDirectors;aMovieYear;aMedias;aSoldTickets;aNrActors;>)
```

- **QUERY BY SQL**コマンドの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY TEXT(aTitles;0)

QUERY BY SQL([MOVIES];"Sold_Tickets > (SELECT AVG(Sold_Tickets) FROM MOVIES)")
ORDER BY([MOVIES];[MOVIES]Title;>)
SELECTION TO ARRAY([MOVIES]Title;aTitles;[MOVIES]Sold_Tickets;aSoldTickets)
` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aDirectors;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
SORT ARRAY(aTitles;aDirectors;aMovieYear;aMedias;aSoldTickets;aNrActors;>)
```

- SQLの*EXECUTE IMMEDIATE*コマンドの場合:

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY TEXT(aTitles;0)
C_TEXT($tQueryTxt)

$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Title, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Sold_Tickets > (SELECT AVG(Sold_Tickets) FROM MOVIES)"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
$tQueryTxt:=$tQueryTxt+" INTO :aTitles, :aSoldTickets"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
` 情報を表示するためにリストボックスの残りの列を初期化
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aDirectors;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
ARRAY LONGINT(aNrActors;Size of array(aTitles))
```

これらをテストするには、**4D SQL Code Samples**データベースを起動してメインウィンドウを表示させます。クエリモードを選択して**Subqueries**ボタンをクリックします。

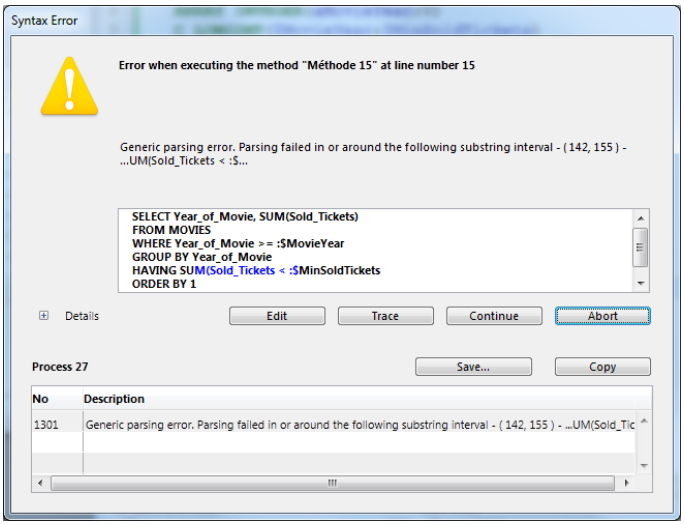
## SQLコードのエラー追跡とデバッグ

---

4Dでは、コードをトレースしたり修正したりする二つの方法があります。**During**を使用してトレースや修正を行うか、**ON ERR CALL**コマンドを呼び出してエラーをキャッチし、適切なアクションを行います。これらの方法を使用して、SQLコード中にある問題を解決することができます。
この例題では、わざと右側の括弧が抜けています。つまり**HAVING SUM(Sold_Tickets <:$MinSoldTickets)**となるべきところが**HAVING SUM(Sold_Tickets <:$MinSoldTickets**となっています。

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
C_LONGINT($MovieYear;$MinSoldTickets)
$MovieYear:=1979
$MinSoldTickets:=10000000

Begin SQL
   SELECT Year_of_Movie, SUM(Sold_Tickets)
   FROM MOVIES
   WHERE Year_of_Movie >= :$MovieYear
   GROUP BY Year_of_Movie
   HAVING SUM(Sold_Tickets < :$MinSoldTickets
   ORDER BY 1
   INTO :aMovieYear,  :aSoldTickets;
End SQL
```

以下のウィンドウのとおり、アプリケーションはエラーを検知し、**Afterシンタックスエラーウィンドウ**を開いて、エラーに関する詳細な情報とエラーの発生場所を提供します。**編集**ボタンをクリックすると、エラーが発生したメソッドが開きます。




エラーがもっと複雑な場合、アプリケーションはスタック情報を含むさらなる情報を提供します。この情報は**詳細**ボタンをクリックすると表示されます。
これをテストするには、**4D SQL Code Samples**データベースのメインウィンドウで、**Debugging SQL code** ボタンをクリックします。

SQLエラーを追跡するもう一つの方法は、**ON ERR CALL**コマンドを使用することです。
この例題ではSQLコードで発生するエラーをキャッチするために、***SQL_Error_Handler***メソッドをエラーハンドラとして使用します。

```
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
C_LONGINT($MovieYear;$MinSoldTickets;MySQL_Error)
$MovieYear:=1979
$MinSoldTickets:=10000000
MySQL_Error:=0

 // エラーをキャッチするためにSQL_Error_Handlerメソッドをインストールする
ON ERR CALL("SQL_Error_Handler")
Begin SQL
   SELECT Year_of_Movie, SUM(Sold_Tickets)
   FROM MOVIES
   WHERE Year_of_Movie >= :$MovieYear
   GROUP BY Year_of_Movie
   HAVING SUM(Sold_Tickets < :$MinSoldTickets
   ORDER BY 1
```

```
   INTO :aMovieYear,  :aSoldTickets;
End SQL
  // SQL_Error_Handlerメソッドを無効にする
ON ERR CALL("")
If(MySQL_Error#0)
   ALERT("SQL Error number: "+String(MySQL_Error))
End if
```

***SQL_Error_Handler***メソッドは以下のようになります:

```
  `(P) SQL_Error_Handler
MySQL_Error:=Error
```

これをテストするには、**4D SQL Code Samples**データベースのメインウィンドウで、**Using ON ERR CALL**ボタンをクリックします。

## データ定義言語

---

SQLのデータ定義言語 (Data Definition Language - DDL) を使用して、データベースストラクチャを定義したり管理したりできます。
DDLコマンドを使用してテーブルやフィールドを作成したり変更したり、さらにデータを追加したり削除したりすることができます。

以下はデーブルを作成、フィールドを追加し、いくつかのデータで埋めるシンプルな例題です。

```
Begin SQL
   DROP TABLE IF EXISTS ACTOR_FANS;

   CREATE TABLE ACTOR_FANS

   (ID INT32,
   Name VARCHAR);

   INSERT INTO ACTOR_FANS
   (ID, Name)
   VALUES(1, 'Francis');

   ALTER TABLE ACTOR_FANS
   ADD Phone_Number VARCHAR;

   INSERT INTO ACTOR_FANS
   (ID, Name, Phone_Number)
   VALUES (2, 'Florence', '01446677888');

End SQL
```

これをテストするには、**4D SQL Code Samples**データベースのメインウィンドウで"**DDL**"ボタンをクリックします。

**Note**:この例題は一度だけ動作します。"DDL"ボタンを2回クリックすると、テーブルが既に存在するというエラーメッセージを受け取ります。

## 外部への接続

---

4Dでは外部データベースを使用することができます。つまりローカルデータベースではないデータベースに対してSQLクエリを実行できます。これを行うために、ODBCを使用して外部データソースに接続するか、直接他の4Dデータベースに接続することができます。
以下は外部データベースとの接続を管理するためのコマンドです:

- **Get current data source** によりアプリケーションが使用するODBCデータソースを知ることができます。
- **GET DATA SOURCE LIST** は、マシンにインストールされたデータソースのリストを取得するために使用します。
- **SQL LOGIN** は、直接あるいはマシンにインストールされたODBCデータソースを経由して外部データベースに接続するために使用します。
- **SQL LOGOUT** は、外部への接続を閉じ、ローカルの4Dデータベースに再接続するために使用します。
- **USE DATABASE** (SQL コマンド) は、4D SQLエンジンを使用して外部4Dデータベースを開くために使用します。

以下の例題は外部データソース (ORACLE) に接続し、ORACLEデータベースからデータを取得、そしてORACLEデータベースとの接続を閉じ、ローカルデータベースに再接続する方法を示してい ます。
システムには"Test_ORACLE_10g"というデータソース名が登録されているものとします。

```
ARRAY TEXT(aDSN;0)
ARRAY TEXT(aDS_Driver;0)
C_TEXT($Crt_DSN;$My_ORACLE_DSN)
ARRAY TEXT(aTitles;0)
ARRAY LONGINT(aNrActors;0)
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)
C_TEXT($tQueryTxt)
REDUCE SELECTION([MOVIES];0)
$MovieYear:=1960
  ` デフォルトでカレントDSNはローカルデータベース (";DB4D_SQL_LOCAL;"、SQL_INTERNAL定数の値)
$Crt_DSN:=Get current data source
ALERT("The current DSN is "+$Crt_DSN)

  ` ローカルデータベースに操作を行う
Begin SQL
   SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets
   FROM MOVIES
   WHERE Year_of_Movie >= :$MovieYear
   INTO :aMovieYear, :aTitles, :aDirectors, :aMedias, :aSoldTickets;
End SQL

  ` ODBCマネージャーに定義されたUserタイプのデータソースを取得
GET DATA SOURCE LIST(User Data Source;aDSN;aDS_Driver)
$My_ORACLE_DSN:="Test_Oracle_10g"
If(Find in array(aDSN;$My_ORACLE_DSN)>0)
  ` 4Dと$My_ORACLE_DSN="Test_Oracle_10g"データソース間の接続を確立

SQL LOGIN($My_ORACLE_DSN;"scott";"tiger";*)

  ` カレントDSNはORACLEデータベース
$Crt_DSN:=Get current data source
ALERT("The current DSN is "+$Crt_DSN)
ARRAY TEXT(aTitles;0)
ARRAY LONGINT(aNrActors;0)
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)

  ` 外部 (ORACLE) データベースに操作を行う
Begin SQL
   SELECT ENAME FROM EMP INTO :aTitles
End SQL

  ` SQL LOGINコマンドで開かれた外部接続を閉じる
SQL LOGOUT
  ` カレントDSNはローカルデータベースになる
$Crt_DSN:=Get current data source
ALERT("The current DSN is "+$Crt_DSN)
```

```
Else
 ALERT("ORACLE DSN not installed")
End if
```

これをテストするには、 "4D SQL Code Samples" データベースのメインウィンドウで**Connect to ORACLE** ボタンをクリックします。

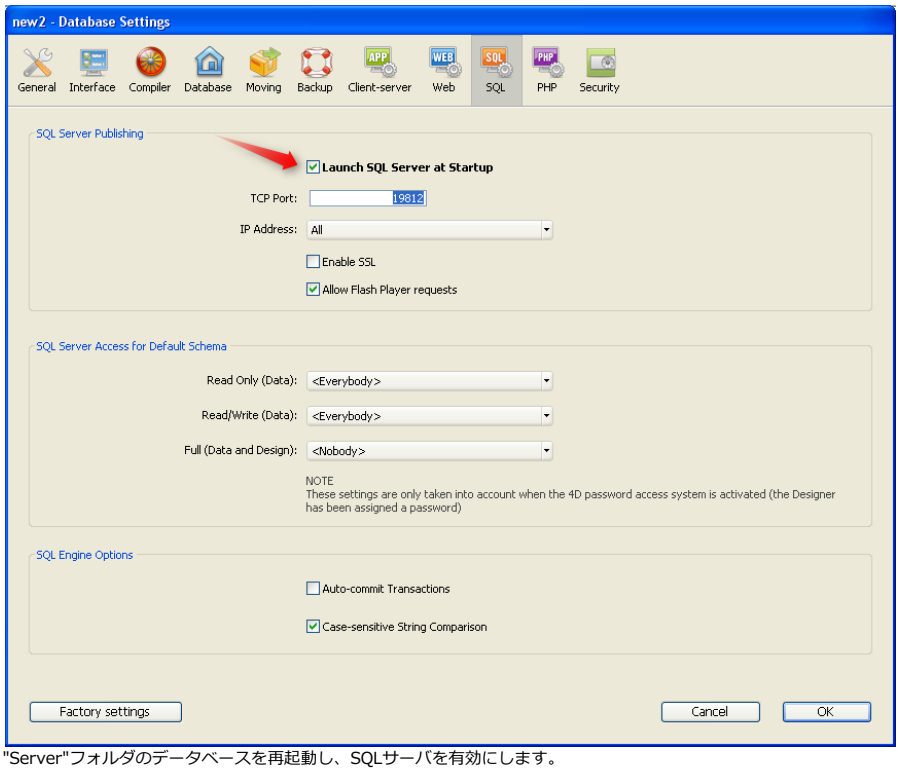
## ODBC Driverを使用した4D SQLエンジンへの接続

---

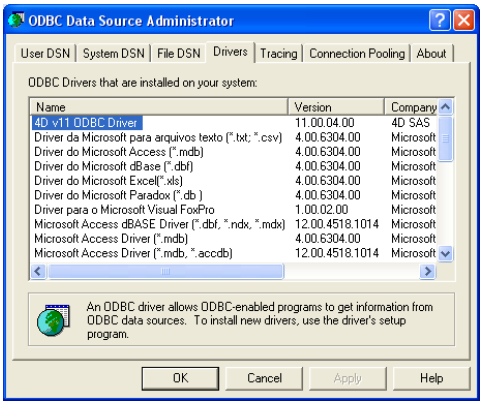
ODBC Driver for 4Dを経由して、外部アプリケーションから4D SQLエンジンに接続できます。

**Note:** この設定は例題として使用されます。4Dアプリケーションに直接SQLで接続することができます。詳細については**SQL LOGIN**コ マンドの説明を参照してください。

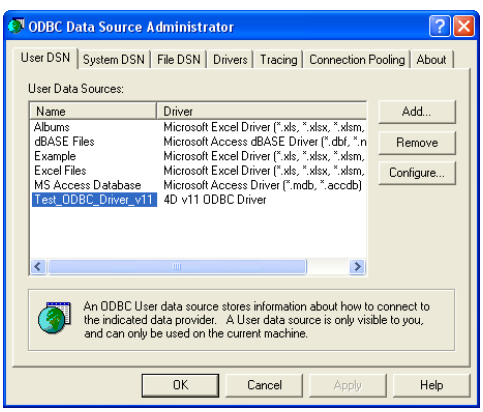
1. チュートリアル例題データベースのコピーを作成します。
2. データベースを含むフォルダの名前を"Client" と"Server"にそれぞれ変更します。
3. "Server"フォルダの例題データベースを起動し、データベース設定のSQL/設定ページで"起 動時にSQLサーバを起動する"チェックボックスにチェックを入れ、アプリケーションの開始時にSQLサーバーが起動されるようにします:



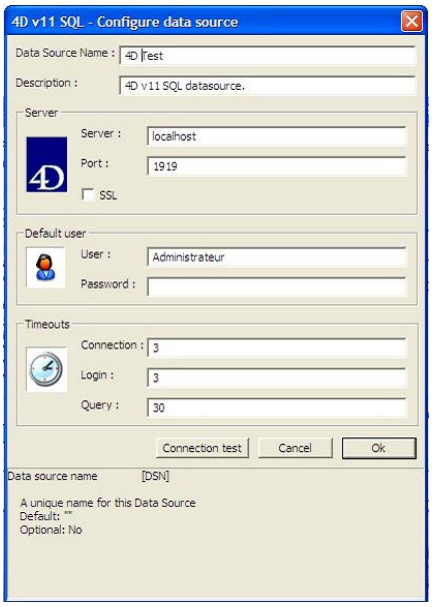

4. "Server"フォルダのデータベースを再起動し、SQLサーバを有効にします。
5. 4D ODBC Driver for 4Dをインストールし、ODBCデータソースアドミニストレータに名前が表示されているか確認します:




6. データソース "Test_ODBC_Driver_v11" を作成します:



そして "**Connection test**" ボタンをクリックして、接続をテストします:




6. "Client"フォルダの例題データベースを起動し、メインウィンドウを開いて、 "Connect to 4D" ボタンをクリックします。このボタンに書かれているコードは以下のとおりです:

```
C_TEXT($Crt_DSN;$My_4D_DSN)
ARRAY TEXT(aDSN;0)
ARRAY TEXT(aDS_Driver;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)

REDUCE SELECTION([MOVIES];0)

  ` デフォルトでカレントDSNはローカルです。
$Crt_DSN:=Get current data source
ALERT("The current DSN is "+$Crt_DSN)

  ` ローカルデータベースに対し操作を行う
Begin SQL
   SELECT Title, Director, Media
   FROM MOVIES
   ORDER BY 1
   INTO :aTitles, :aDirectors, :aMedias;
End SQL

  ` ODBCマネージャーに定義されたUserタイプのデータソースを取得する
GET DATA SOURCE LIST(User Data Source;aDSN;aDS_Driver)
$My_4D_DSN:="Test_ODBC_Driver_v11"
If(Find in array(aDSN;$My_4D_DSN)>0)

  ` ODBC Driver v114Dと他の4D間の接続を確立する
SQL LOGIN($My_4D_DSN;"Administrator";"";*)
If(Ok=1)

  ` カレントDSNは4Dのデータベース
   $Crt_DSN:=Get current data source
   ALERT("The current DSN is "+$Crt_DSN)

   ARRAY TEXT(aTitles;0)
   ARRAY TEXT(aDirectors;0)
   ARRAY TEXT(aMedias;0)

  ` 外部 (4D) データベースに対し操作を行う
   Begin SQL
      SELECT Title, Director, Media
      FROM MOVIES
      ORDER BY 1
      INTO :aTitles, :aDirectors, :aMedias;
   End SQL

  ` USE EXTERNAL DATABASEコマンドで開かれた外部接続を閉じる
   SQL LOGOUT
```

```
    ` カレントDSNはローカルデータベースになる
    $Crt_DSN:=Get current data source
    ALERT("The current DSN is "+$Crt_DSN)

  Else
    ALERT("Unable to connect to the external data source")
  End if
 Else
  ALERT("ODBC Driver data source not found")
 End if
```

ご覧頂いたとおり、メソッドの最初の部分でローカルデー タベースへのクエリを作成しています。次に、ODBC Driverを経由して他の4Dデータベースに接続し、同じクエリを行っています。結果はもちろん同じはずです。

# 4DでSQLを使用する

## 4DでSQLを使用する

---

この節では、4DにおけるSQLの使用に関する一般的な概要を説明します。統合されたSQLエンジンへの接続方法、クエリやデータの取得を行う様々な方法が説明されます。また4D SQLサーバの設定方法や、4DとSQLエンジンの統合に関する原則の概要も説明します。

# 4D SQLエンジンにアクセスする

## 4D SQLエンジンにクエリを送信する

4Dのビルトイン SQLエンジンは3つの方法で呼び出すことができます:

- **QUERY BY SQL**コマンドを使用する。SQLの*SELECT*文の**WHERE**句を*query*引数に渡します。
  例:

```
QUERY BY SQL([OFFICES];"SALES > 100")
```

- 4Dに統合されたSQLコマンドを使用する (“SQL”テーマの**SQL SET PARAMETER**、**SQL EXECUTE**など) 。これらのコマンドはODBCデータソース、またはカレントデータベースの4D SQLエンジンに対して実行されます。
- 4D標準のメソッドエディタを使用する。SQL文を直接4Dのメソッドエディタに記述できます。これを行うには、SQLクエリを**Begin SQL**と**End SQL**タグの間に記述します。4Dのインタプリタは、これらのタグの間に記述されたコードを解析せず、コードはSQLエンジン (または**SQL LOGIN**コマンドで設定されている場合は他のエンジン) によって実行されます。

## 4Dと SQLエンジン間でデータを渡す

### 4D表現式の参照

すべてのタイプの有効な4D表現式 (変数、フィールド、配列、式...) をSQL式の**WHERE**と**INTO**句で参照できます。4D参照であることを示すために、以下のいずれかの記法を使用できます:

- 参照を"<<"と">>"との間に記述する。
- 参照の前にコロン":"を置く。

例:

```
C_TEXT(vName)
vName:=Request("Name:")
SQL EXECUTE("SELECT age FROM PEOPLE WHERE name=<<vName>>")
```

または:

```
C_TEXT(vName)
vName:=Request("Name:")
Begin SQL
   SELECT age FROM PEOPLE WHERE name= :vName
End SQL
```

**注:** インタープロセス変数を使用する場合は、ブラケット [] が必要です (例えば<<[<>myvar]>>または:[<>myvar])。

#### コンパイルモードでローカル変数を使用する

コンパイルモードでは特定の条件下において、SQL文の中でローカル変数参照を使用できます:

- **Begin SQL** / **End SQL**ブロック内でローカル変数を使用できます。ただし**MissingRef**コマンドは除きます;
- 変数を (参照ではなく) 引数内で直接使用する場合、**SQL EXECUTE**コマンドでローカル変数を使用できます。
  例えば以下のコードはコンパイルモードで動作します:

  ```
  SQL EXECUTE("select * from t1 into :$myvar")
  ```

  以下のコードはコンパイルモードでエラーを生成します:

  ```
  C_TEXT(tRequest)
  tRequest:="select * from t1 into :$myvar"
  SQL EXECUTE(tRequest)
  ```

### SQLリクエストから4Dにデータを受け取る

*SELECT*文でのデータ取得は、**Begin SQL**/**End SQL**タグの中の*SELECT*コマンドで**INTO**句を使用する方法と、 "SQL"ランゲージコマンドを使う方法、両方で可能です。

- **Begin SQL**/**End SQL**タグを使用する場合、SQLクエリ内の**INTO**句で、有効な4D式 (フィールド、変数、配列) を参照して値を受け取ります:

```
Begin SQL
   SELECT ename FROM emp INTO <<[Employees]Name>>
End SQL
```

- **SQL EXECUTE**コマンドでは、追加の引数も使用できます:

```
SQL EXECUTE("SELECT ename FROM emp";[Employees]Name)
```

SQLからデータを受け取るこれら (**Begin SQL**/**End SQL**タグとSQLコマンド) 2つの方法の主な違いは、前者は一ステップですべての情報が4Dに返されるのに対し、後者はレコードを明示的に**SQL LOAD RECORD**でロードしなければならないことにあります。

例えばPEOPLEテーブルに100レコードあるとして:

- 4Dの汎用SQLコマンドの場合:

```
ARRAY INTEGER(aBirthYear;0)
C_STRING(40;vName)
vName:="Smith"
$SQLStm:="SELECT Birth_Year FROM PEOPLE WHERE ename= <<vName>>"
SQL EXECUTE($SQLStm;aBirthYear)
While(Not(SQL End selection))
   SQL LOAD RECORD(10)
End while
```

100レコードをすべて取り出すために10回ループします。一回ですべてのレコードをロードするには以下のようにします:

```
SQL LOAD RECORD(SQL All Records)
```

- **Begin SQL**/**End SQL**タグの場合:

```
ARRAY INTEGER(aBirthYear;0)
C_STRING(40;vName)
vName:="Smith"
Begin SQL
   SELECT Birth_Year FROM PEOPLE WHERE ename= <<vName>> INTO <<aBirthYear>>
End SQL
```

この場合、*SELECT*文の実行後、aBirthYear配列のサイズは100となり、各要素は100レコードから取り出された誕生年で埋められます。
配列ではなく (4Dフィールドなどの) カラムにデータを取得すると、4Dは必要なだけ自動でレコードを作成し、データを保存します。先の例題で、 PEOPLEテーブルに100レコードがあると仮定しました:

- 4Dの汎用SQLコマンドの場合:

```
C_STRING(40;vName)
vName:="Smith"
$SQLStm:="SELECT Birth_Year FROM PEOPLE WHERE ename= <<vName>>”
SQL EXECUTE($SQLStm;[MYTABLE]Birth_Year)
While(Not(SQL End selection))
   SQL LOAD RECORD(10)
End while
```

100レコードを取得するために10回のループを行います。ループ毎に[MYTABLE]テーブルに10レコードが作成され、PEOPLEテーブルから取得したBirth_Yearが[MYTABLE]Birth_Yearフィールドに保存されます。

- **Begin SQL**/**End SQL**タグの場合:

```
C_STRING(40;vName)
vName:="Smith"
Begin SQL
   SELECT Birth_Year FROM PEOPLE WHERE ename= <<vName>> INTO <<[MYTABLE]Birth_Year>>
End SQL
```

この場合、*SELECT*文の実行中、[MYTABLE]テーブルに100レコードが作成され、それぞれのBirth_Yearフィールドには PEOPLEテーブルのBirth_Yearカラムの対応するデータが格納されます。

### リストボックスの使用

4Dは、*SELECT*クエリの結果データをリストボックスに配置するために使用できる、特別な自動機能 (**LISTBOX**キーワード) を提供します。詳細はDesign Referenceマニュアルを参照してください。

### クエリの最適化

最適化のために、クエリにはSQL関数よりも4D式を使用することをお勧めします。4D式はクエリ実行前に一度だけ評価されます。SQL関数はそれぞれのレコード毎に評価されます。
例えば以下の文で:

```
SQL EXECUTE("SELECT FullName FROM PEOPLE WHERE FullName=<<vLastName+vFirstName>>")
```

vLastName+vFirstName式はクエリの実行前に一度だけ計算されます。以下の文では:

```
SQL EXECUTE("SELECT FullName FROM PEOPLE WHERE FullName=CONCAT(<<vLastName>>,<<vFirstName>>)")
```

*CONCAT*(<<vLastName>>,<<vFirstName>>) 関数はテーブルのレコード毎に呼び出されます。言い換えればレコードごとに式が評価されます。

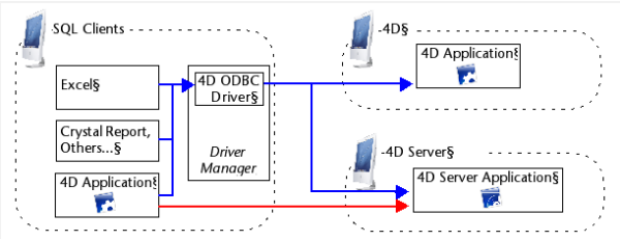
# 4D SQLサーバの設定

4DのSQLサーバは4Dデータベースに格納されたデータへの外部からのアクセスを可能としています。サードパーティのアプリ ケーションや4Dアプリケーションからは、4D ODBC Driverを使用してアクセスします。また、4Dクライアントと4D Serverアプリケーションとの間では、ダイレクトに接続します。そしてすべての接続は TCP/IPプロトコルによって行われます。

4DアプリケーションのSQLサーバはいつでも停止あるいは開始できます。さらにパフォーマンスやセキュリティのためTCP ポートを変更したり、待 ち受けIPアドレスを指定したり、4Dデータベースへのアクセスを制限したりできます。

## SQLサーバへの外部アクセス

4D SQLサーバへの外部アクセスはODBC (すべての設定) 経由、または直接 (4D Serverに接続されている4Dクライアントアプリケーション) 行われます。以下の図はこのことを示しています:




青: ODBC経由の接続
赤: 直接接続

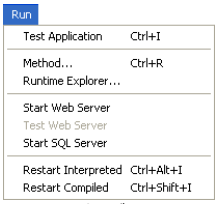
クエリレベルでは、直接外部接続あるいはODBC経由の接続を開くことは、**SQL LOGIN**コ マンドで実行されます。詳細については、このコマンドの説明を参照してください。

- **ODBC経由の接続**: 4Dはあらゆるサードパーティのアプリケーション (Excel® タイプのスプレッドシートや他のDBMS等) 、及び他の4Dアプリケーションが、4DのSQLサーバに接続するためのODBC driverを提供します。4D ODBC driverは、SQLクライアントが動作するマシンにインストールします。4D ODBC driverのインストールと設定は別マニュアルで説明しています。
- **直接接続**: 4D Serverアプリケーションだけが、他の4Dアプリケーションからの直接SQLクエリに応答できます。同様に、 "Professional" 製品ラインの4Dアプリケーションだけが、他の4Dアプリケーションへの直接接続を開くことができます。直接接続が実行されている間のデータ交換は、自動 的にデータ統合と同期性に関連する問題を排除する同期モードで実行されます。1つのプロセスにつき1つの接続だけを許可されます。同時に複数の接続を実行 する場合、必要なだけプロセスを作成しなければなりません。直接接続は、接続先 (4D Server) 側のデータベース設定の"SQL"ページにある**SSL を有効**にするオプションを有効にすることによって、保護できます。直接接続はSQLサーバが開始されている場合のみ、4D Serverにより認可されます。直接接続の主な利点は、データ交換のスピードが速くなることです。

## 4D SQLサーバの開始と停止

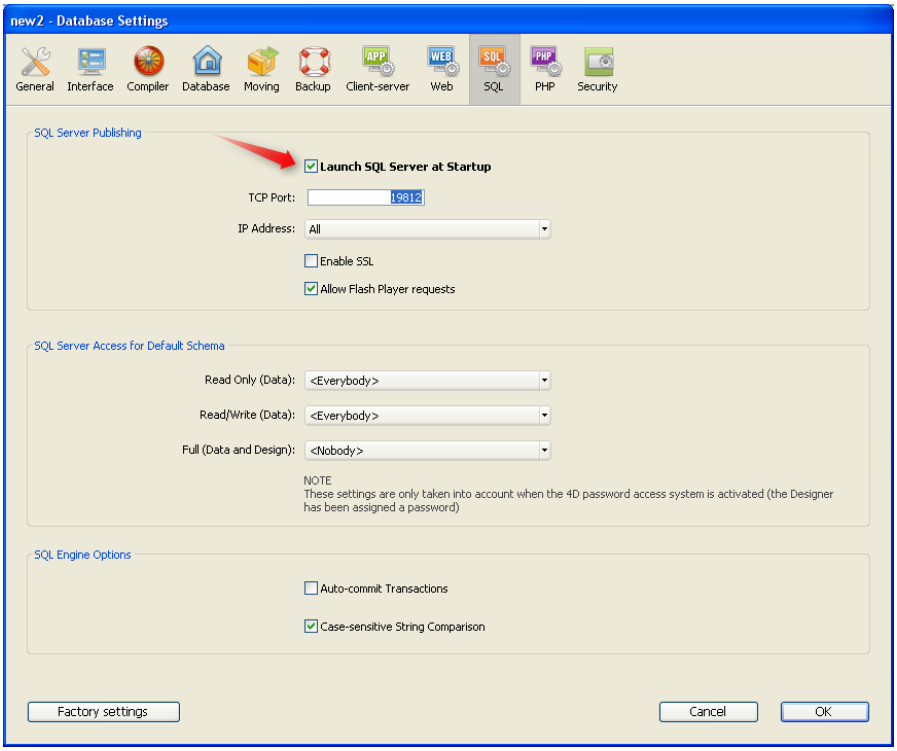
SQLサーバは3つの方法で開始または停止できます:

- 4Dアプリケーションの**実行**メニューから、**SQLサーバ開始/SQLサーバ停止**コマンドを選択する:




**Note:** 4Dサーバでは、このメニューコマンドは、**SQLサーバページ**の ボタンからアクセスできます。
サーバが起動されていると、このメニュー項目は**SQLサーバ停止**に変わります。

- データベース設定でアプリケーションの開始時に自動で起動するよう設定する。これを行うには、**SQL/設定ページ**で**開始時に SQLサーバを起動**するオプションにチェックを入れます。




- (テーマの) **START SQL SERVER**や**STOP SQL SERVER**コマンドを使用してプログラムで制御する。
  SQLサーバが停止している (またはまだ開始されていない場合)、4Dは外部からのSQLクエリに応答しません。
  **Note**: SQLサーバの停止は、4D SQLエンジンの内部動作に影響しません。SQLエンジンは内部クエリに対し常に有効です。

## SQLサーバの公開設定

4Dに統合されたSQLサーバの公開パラメタを変更できます。このパラメタはデータベース設定の**SQL/設定**ページにあります:

- **開始時にSQLサーバを起動**オプションを使用して、アプリケーションの開始時にSQLサーバを起動できます。
- **TCPポート**: デフォルトで4D SQLサーバはTCPポート19812に応答します。このポートが他のサービスで既に使用されていたり、あなたの接続パラメーターが他の設定を必要とする場合 は、4D SQLサーバが使用するポート番号を変更できます。
  **Note**: 0を指定すると、4DはデフォルトのTCPポート19812を使用します。
- **IPアドレス**: SQLサーバがSQLクエリを処理するために使用する、マシンのIPアドレスを限定することができます。デフォルトでは**すべて**が選択されて いて、サーバはすべてのIPアドレスへのクエリに応答します。
  "IPアドレス"ドロップダウンリストには、マシンに設定されているすべてのIPア ドレスがリストされます。特定のアドレスを選択すると、サーバはこのIPアドレスに送信されたクエリのみに応答します。
  これは複数のTCP/IP アドレスを持つマシン上で動作する4Dアプリケーションで有効な設定です。
  **Note**: クライアント側では、アプリケーションが接続するSQLサーバのIPアドレスやポート番号を、ODBCデータソース定義で正しく設定しなければなりませ ん。
- **SSLを有効にする**: このオプションは、SQL接続を処理する際にSQLサーバがSSLプロトコルを使用するかしないかを指定します。このオプションをチェックした時は、**key.pem**と**cert.pem**ファ イルを以下の場所へコピーしてください:
  MyDatabase/Preferences/SQL ("MyDatabase"はデータベースフォルダ/パッケージ)。
  この場合**SQL LOGIN**コマンドで接続を開く際、SQLサーバーのIPアドレスの後ろに":ssl"キーワードを追加しなければならない点に留意してください。
- **Flash Playerリクエストを許可する**: このオプションで、4D SQLサーバによるFlash Playerリクエストをサポートするメカニズムを有効にするかどうかを指定できます。このメカニズムは、データベースの環境設定フォルダ (Preferences/SQL/Flash /) に "socketpolicy.xml" という名前のファイルが存在するかどうかに基づきます。このファイルはFlash Playerがクロスドメインの接続やFlex (Web 2.0) アプリケーションのソケットによる接続を行う場合に必要とされます。
  以前の4Dバージョンでは、このファイルを手作業で追加しなければなりません でした。これからは、**Flash Playerリクエストを許可する**オプションをチェックすることにより有効化されます。このオプションを チェックすると、Flash Playerリクエストが受け入れられ、必要に応じて、データベース用の汎用"socketpolicy.xml" ファイルが作成されます。このオプションの選択を解除した場合、 "socketpolicy.xml" ファイルは無効となります（名前変更）。後にSQLサーバが受け取るすべてのFlash Playerクエリは拒否されます。
  データベースを開い た時、このオプションがチェックされているか否かは、データベースの環境設定フォルダに有効な"socketpolicy.xml" ファイルが存在するかどうかに基づきます。
  **Note**: 4Dの**SQL SET OPTION**コマンドで、外部リクエストを処理する際にSQL サーバが使用するエンコーディングを設定できます。

## デフォルトスキーマに対するSQLアクセスコントロール

セキュリティの理由で、SQLサーバに送信された外部クエリが4Dデータベースで行うことのできるアクションを制限できます。
2つのレベル でコントロールが可能です:

- 許可するアクションのタイプのレベル
- クエリを実行するユーザのレベル

これらはデー タベース設定の**SQL/設定**ページで設定できます:

このダイアログボックスに設定されているパラメタはデフォルトスキーマに適用されます。データベースへの外部接続制御は、SQLスキーマ (**4Dと4D SQLエンジン統合の原則**参照） のコンセプトに基づきます。カスタムスキーマを作成しない場

合、すべてのデータベーステーブルはデフォルトスキーマに含まれます。特定のアクセス権で他の スキーマを作成し、それらをテーブルに割り当てる場合、カスタムスキーマに含まれていないテーブルのみが、デフォルトスキーマに含まれます。

SQLサーバによるデフォルトスキーマに対し、3タイプのアクセス設定ができます:

- "読み込みのみ (データ)": データベーステーブルのすべてのデータに対する無制限の読み込みアクセス。レコードの追加、更新、削除のほか、データベースストラクチャに対する変更はで きません。
- "読み書き (データ)": データベーステーブルのすべてのデータへの読み書き (追加、更新、削除) アクセスが可能です。データベースストラクチャの更新は許可されません。
- "フルアクセス (データとデザイン)": データベーステーブルのすべてのデータへの読み書き (追加、更新、削除) およびデータベースストラクチャの更新 (テーブル、フィールド、リレーションなど) が許可されます。

それぞれのアクセスタイプごとにユーザのセットを割り当てることができます。3つのオプションが選択可能です:

- <**Nobody**>: このオプションを選択すると、指定されたタイプのアクセスは、クエリの発行元にかかわらず拒否されます。このパラメタは4Dパスワードアクセス管理システ ムが有効でない場合でも使用できます。
- <**Everybody**>: このオプションを選択すると、指定されたタイプのアクセスは、制限なしにすべてのクエリに対して許可されます。
- ユーザーグループ: このオプションは、ユーザのグループを作成し、特定のタイプへのアクセスを行うことができるユーザを指定するために使用します。このオプションを使用する ためには4Dパスワードシステムが有効になっていなければなりません。クエリ元のユーザは、SQLサーバへの接続時、ユーザ名とパスワードを提示します。

**警告**: このメカニズムは4Dパスワードシステムに依存します。SQLサーバに対するアクセスコントロールを動作させるためには、(Designerにパスワード を設定して、) 4Dパスワードシステムが有効になっていなければなりません。

**Note**: 4Dプロジェクトメソッドごとに追加のセキュリティオプションを設定できます。詳細は**4Dと4D SQLエンジン統合の原則**の"SQL利用可オプション"を参照してください。

# 4Dと4D SQLエンジン統合の原則

原則として、4D SQLエンジンはSQL-92互換です。つまり使用するコマンドや関数、演算子、シンタックスの詳細な説明が必要な場合、SQL-92リファレンスを参照できます。これらは例えばインターネットで見つけることができます。

しかし、4D SQLエンジンはSQL-92の機能を100%サポートしているわけではありません。また特定の追加の機能をサポートしています。

この節では、4D SQLエンジンの主たる実装と制限について説明しています。

## 一般的な制限

4DのSQLエンジンは4Dデータベースの深い部分に統合されているため、テーブルやカラム (フィールド) 、レコードの最大数や、テーブルやカラムの命名規則に関する制限は、標準の内部的な4Dエンジン (DB4D) と同じです。以下にこれらをリストします。

- テーブルの最大数: 理論的には20億ですが、4Dとの互換性のため32767となっています。
- 1テーブルの カラム (フィールド) の最大数: 理論的には20億ですが、4Dとの互換性のため32767となっています。
- 1テーブルの ロー (レコード) の最大数: 10億
- インデックスキーの最大数: 10億 x 64
- 主 キーにNULL値は許されずまたユニークでなければなりません。主キーカラム (フィールド) にインデックスを付ける必要はありません。
- テー ブルやフィールドの最大文字数: 31文字 (4Dの制限)

異なるユーザにより作成されたとしても、同じテーブル名は許可されません。標準の4Dコントロールが適用されます。

## データタイプ

以下の表では、4D SQLでサポートされるデータタイプと、それに対応する4Dのタイプを説明しています:

| 4D SQL | 説明 | 4D |
|---|---|---|
| Varchar | 文字テキスト | テキストまたは文字 |
| Real | 範囲+/-1.7E308の浮動小数点数 | 実数 |
| Numeric | 範囲+/- 2E64の数値 | 64 bit整数 |
| Float | 浮動小数点数 (事実上無限) | Float |
| Smallint | 範囲-32,768から32,767の数値 | 整数 |
| Int | 範囲-2,147,483,648から2,147,483,647の数値 | 倍長整数 |
| Int64 | 範囲 +/- 2E64の数値 | 64 bit整数 |
| UUID | 16バイトの数値 (128 bit) で32個の16進文字を含む | UUID 文字フォーマット |
| Bit | TRUE/FALSEまたは1/0値のみをとるフィールド | ブール |
| Boolean | TRUE/FALSEまたは1/0値のみをとるフィールド | ブール |
| Blob | 2GBまで: 画像、アプリケーション、ドキュメントなどのバイナリオブジェクト | Blob |
| Bit varying | 2GBまで: 画像、アプリケーション、ドキュメントなどのバイナリオブジェクト | Blob |
| Clob | 2GB文字までのテキスト。この カラム (フィールド) にインデックスは付けられません。これはレコード自身には保存されません。 | テキスト |
| Text | 2GB 文字までのテキスト。このカラム (フィールド) にインデックスは付けられません。これはレコード自身には保存されません。 | テキスト |
| Timestamp | 日付と時間。日付は'YYYY/MM/DD'フォーマットで時間は'HH:MM:SS:ZZ'フォーマット | 個別に処理される日付と時間 (自動変換) |
| Duration | 'HH:MM:SS:ZZ'フォーマットの時間 | 時間 |
| Interval | 'HH:MM:SS:ZZ'フォーマットの時間 | 時間 |
| Picture | 2GBまでのPICTピクチャ | ピク チャ |

数値タイプ間の自動変換が実装されています。
数値を表す文字列は対応する数値に変換されません。特別な*CAST*関数を使用して、タイプ間の変換を行うことができます。
以下のSQLデータタイプは 実装されていません:

- NCHAR
- NCHAR VARYING.

## 4DにおけるNULL値

NULL値は4D SQLランゲージおよび4Dデータベースエンジンに実装されています。しかし4Dランゲージではサポートされていません。にもかかわらず、**Is field value Null**と**SET FIELD VALUE NULL**コマンドを使用して、4DフィールドのNULL値を読み書きできます。

### 処理の互換性とNULL値を空値にマップするオプション

4Dでの互換性のため、4Dデータベーステーブルに格納されたNULL値は、4Dランゲージを使用して操作する際、自動でデフォルト値に変換されます。例えば以下の文において:

```
myAlphavar:=[mytable]MyAlphafield
```

MyAlphafieldフィールドにNULL値が含まれる場合、myAlphavar変数には"" (空の文字列) が代入されます。
デフォルト値はデータ型により異なります:

- 文字およびテキストデータ型: ""
- 実数、整数、および倍長整数型: 0
- 日付データ型: “00/00/00”

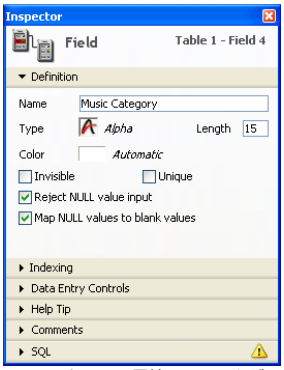
- 時間データ型: “00:00:00”
- ブールデータ型: False
- ピクチャデータ型: 空のピクチャ
- Blobデータ型: 空のBlob

他方、このメカニズムは原則として、クエリのような4Dデータベースエンジンレベルでは適用されません。実際、空の値の検索 (例えばmyvalue=0) はNULL値が格納されたレコードを見つけませんし、逆の場合も同様です。両方のタイプの値 (デフォルト値とNULL) がレコードの同じフィールドに存在する場合、処理が変更されたり、あるいは追加のコードが必要となることがあります。
この不便さを避けるために、4Dランゲージで、すべての処理を標準化するために使用できる**NULL値を空値にマップ**オプションがあります。ストラクチャエディタのフィールドインスペクタにあるこのオプションを使用すると、デフォルト値を使用する原則をすべての処理に拡張できます。NULL値を含むフィールドは、機械的にデフォルト値を含むものとして扱われます。このオプションはデフォルトでチェックされています。

**NULL値を空値にマップ**プロパティはデータベースエンジンの低レベルで考慮されます。特に**Is field value Null**コマンドに影響します。

### NULL値の入力を拒否属性

**NULL値の入力を拒否**フィールドプロパティは、NULL値が格納されることを防ぐ目的で使用されます:

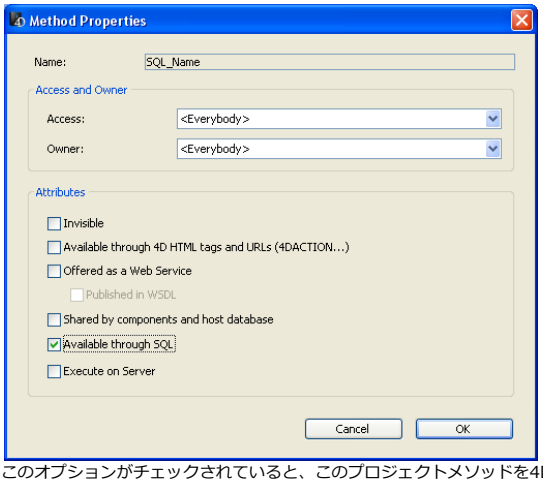



フィールドのこの属性にチェックが入れられていると、そのフィールドにNULL値を格納できなくなります。この低レベルのプロパティはSQLのNOT NULL属性に対応します。
一般的に、4DデータベースでNULL値を使用したい場合、4DのSQLランゲージのみをデータベースで使用することをお勧めします。

**Note**: 4Dでは、フィールドに"必須入力"属性を設定することもできます。2つの設定のコンセプトは似ていますが、スコープが異なります。"必須入力"属性はデータ入力コントロールであり、“NULL値の入力を拒否”属性はデータベースエンジンレベルで動作します。
この設定がされたフィールドがNULL値を受け取ると、エラーが生成されます。

## "SQL利用可"オプション

4Dプロジェクトメソッドにセキュリティ関連のプロパティ、**SQL利用可**が追加されました:




このオプションがチェックされていると、このプロジェクトメソッドを4D SQLエンジンから実行できます。デフォルトではチェックされておらず、4Dプロジェクトメソッドは保護されていて、4D SQLエンジンから呼び出すことはできません。メソッドを4D SQLエンジンから実行可能にするには、このオプションにチェックしてそれを認可しなければなりません。

このプロパティはODBC Driver経由、あるいは**Begin SQL**/**End SQL**タグの間に挿入されたSQLコード、または**QUERY BY SQL**コマンドによる実行など、内部外部問わずすべてのSQLクエリに適用されま す。

**Notes**:

- メソッドに"SQL利用可"属性が設定されていても、実行の際には環境設定レベルおよびメソッドプロパティレベルで設定されたアクセス権 が考慮されます。
- ODBC SQLProcedure関数は、"SQL利用可"属性が設定されたプロジェクトメソッドのみを返します。

## SQLエンジンオプション

- **自動コミットトランザクション**: このオプションを使用してSQLエンジンの自動コミットメカニズムを有効にできます。自動コミットモードの目的は、データの参照整合性を保つことにあります。このオプションがチェックされていると、トランザクションの中で実行されていないすべての**SELECT**、**INSERT**、**UPDATE**、そして**DELETE** (SIUD) クエリは、自動でアドホックなトランザクションに含められます。これによりクエリが完全に実行されるか、エラーが発生した場合にはキャンセルされることが 保障されます。
- すでにトランザクションの中にある (参照整合性がカスタムに管理されている) クエリは、このオプションの影響を受けません。

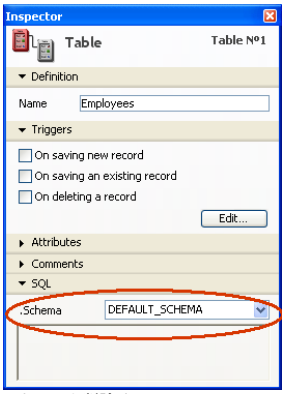
こ のオプションにチェックがされていないと、(**SELECT... FOR UPDATE** クエリを除き) 自動トランザクションは生成されません (**SELECT**コマンド参照)。デフォルトでこ のオプションはチェックされていません。
**SET DATABASE PARAMETER**コ マンドを使用して、プログラムでこのオプションを管理することもできます。
**Note**: 4D SQLエンジンによりクエリされるローカルデータベースのみがこのパラメタの影響を受けます。外部データベースの場合、自動コミットメカニズムはリモート のSQLエンジンが処理します。

- **大文字小文字を区別した文字列比較**: このオプションを使用してSQLクエリ時の文字列比較を変更することができます。このオプションはデフォルトでチェックされていて、SQLエンジンは文字 列比較 (ソートやクエリ) の際に大文字小文字を区別します。 例えば“ABC”=“ABC” ですが “ABC” # “Abc” となります。
SQL エンジンと4Dエンジンの動作を揃えたいなど特定のケースでは、文字列比較で大文字小文字を区別したくない (“ABC”=“Abc”) 場合があります。そのようにするには、このオプションの選択を外します。
**SET DATABASE PARAMETER**コマンドを使用して、プログラムでこのオ プションを管理することもできます。

## スキーマ

4Dはスキーマのコンセプトを実装しています。スキーマはデータベースのテーブルを含む仮想的なオブジェクトです。SQLにおいてスキーマの目的は、異なるデータベースオブジェクトのセットに特定のアクセス権を割り当てることです。スキーマはデータベースを、そのデータベース全体を形成する個々の独立するエンティティに分割します。つまりあるテーブルはいつもただ1つのスキーマに属します。

- スキーマを作成するには*CREATE SCHEMA*コマンドを使用します。そして*GRANT*と*REVOKE*コマンドを使用してスキーマにアクセスタイプを設定できます。
- テーブルをスキーマに割り当てるには、*CREATE TABLE*または*ALTER TABLE*コマンドを呼び出します。また4Dのストラクチャエディタにあるインスペクタの"スキーマ"ポップアップメニューを使用することもできます。このメニューにはデータベースで定義されているすべてのスキーマが表示されます:




- スキーマを削除するには、*DROP SCHEMA*コマンドを使用します。

**Note**: スキーマによるアクセスコントロールは、外部からの接続のみに適用されます。**Begin SQL**/**End SQL**タグ、**SQL EXECUTE**、**QUERY BY SQL**等によって4D内で実行されるSQLコードは、常に完全なアクセスを持ちます。

## システムテーブル

4DのSQLカタログには7個のシステムテーブルが含まれていて、読み込み権限を持つSQLユーザがアクセス可能です。システムテーブルには以下があります: *_USER_TABLES, _USER_COLUMNS*, *_USER_INDEXES*, *_USER_CONSTRAINTS*, *_USER_IND_COLUMNS*, *_USER _CONS_ COLUMNS*, *_USER_SCHEMAS.*

SQLの慣習に従い、システムテーブルはデータベースストラクチャを記述します。以下はシステムテーブルに関する説明とフィールドです:

| ***_USER_TABLES*** | | **データベースのユーザテーブルを記述します** |
|---|---|---|
| TABLE_NAME | VARCHAR | テーブル名 |
| TEMPORARY | BOOLEAN | テーブルが一時的の場合True、そうでなければFalse |
| TABLE_ID | INT64 | テーブル番号 |
| SCHEMA_ID | INT32 | スキーマ番号 |

| ***_USER_COLUMNS*** | | **データベースのユーザテーブルのカラムを記述します** |
|---|---|---|
| TABLE_NAME | VARCHAR | テーブル名 |
| COLUMN_NAME | VARCHAR | カラム名 |
| DATA_TYPE | INT32 | カラムタイプ |
| DATA_LENGTH | INT32 | カラム長 |
| NULLABLE | BOOLEAN | カラムがNULLを受け入れる場合True、そうでなければFalse |
| TABLE_ID | INT64 | テーブル番号 |
| COLUMN_ID | INT64 | カラム番号 |

| ***_USER_INDEXES*** | | **データベースのユーザインデックスを記述します** |
|---|---|---|
| INDEX_ID | VARCHAR | インデックス番号 |
| INDEX_NAME | VARCHAR | インデックス名 |
| INDEX_TYPE | INT32 | インデックスタイプ (1=Bツリー / 複合, 3=クラスタ / キーワード, 7=自動) |
| TABLE_NAME | VARCHAR | インデックスがはられたテーブル名 |
| UNIQUENESS | BOOLEAN | インデックスが重複不可の制約を課す場合True、そうでなければFalse |
| TABLE_ID | INT64 | インデックスがはられたテーブル番号 |

| *_USER_IND_COLUMNS* | | **データベースのユーザインデックスのカラムを記述します** |
|---|---|---|
| INDEX_ID | VARCHAR | インデックス番号 |
| INDEX_NAME | VARCHAR | インデックス名 |
| TABLE_NAME | VARCHAR | インデックスがはられたテーブル名 |
| COLUMN_NAME | VARCHAR | インデックスがはられたフィールド名 |
| COLUMN_POSITION | INT32 | インデックス中のカラムポジション |
| TABLE_ID | INT64 | インデックスがはられたテーブル番号 |
| COLUMN_ID | INT64 | カラム番号 |

| *_USER_CONSTRAINTS* | | **データベースの整合性制約を記述します** |
|---|---|---|
| CONSTRAINT_ID | VARCHAR | 制約番号 |
| CONSTRAINT_NAME | VARCHAR | 制約定義に割り当てられた名前 |
| CONSTRAINT_TYPE | VARCHAR | 制約定義のタイプ (P=主キー、R=参照整合性 - 外部キー、4DR=4Dリレーション) |
| TABLE_NAME | VARCHAR | 制約定義付きのテーブル名 |
| TABLE_ID | INT64 | 制約付きのテーブル番号 |
| DELETE_RULE | VARCHAR | 参照制約の削除ルール – CASCADEまたはRESTRICT |
| RELATED_TABLE_NAME | VARCHAR | リレートしたテーブル |
| RELATED_TABLE_ID | INT64 | リレートしたテーブルの番号 |

| *_USER_CONS_COLUMNS* | | **データベースのユーザ制約のカラムを記述します** |
|---|---|---|
| CONSTRAINT_ID | VARCHAR | 制約番号 |
| CONSTRAINT_NAME | VARCHAR | 制約名 |
| TABLE_NAME | VARCHAR | 制約のあるテーブル名 |
| TABLE_ID | INT64 | 制約のあるテーブル番号 |
| COLUMN_NAME | VARCHAR | 制約のあるカラム名 |
| COLUMN_ID | INT64 | 制約のあるカラム番号 |
| COLUMN_POSITION | INT32 | 制約のあるカラムのポジション |
| RELATED_COLUMN_NAME | VARCHAR | 制約中リレートしたカラムの名前 |
| RELATED_COLUMN_ID | INT32 | 制約中リレートしたカラムの番号 |

| *_USER_SCHEMAS* | | **データベースのスキーマを記述します** |
|---|---|---|
| SCHEMA_ID | INT32 | スキーマ番号 |
| SCHEMA_NAME | VARCHAR | スキーマ名 |
| READ_GROUP_ID | INT32 | 読み込みのみアクセスをもつグループ番号 |
| READ_GROUP_NAME | VARCHAR | 読み込みのみアクセスをもつグループ名 |
| READ_WRITE_GROUP_ID | INT32 | 読み書きアクセスをもつグループ番号 |
| READ_WRITE_GROUP_NAME | VARCHAR | 読み書きアクセスをもつグループ名 |
| ALL_GROUP_ID | INT32 | フルアクセスをもつグループ番号 |
| ALL_GROUP_NAME | VARCHAR | フルアクセスをもつグループ名 |

**Note**: システムテーブルには、**SYSTEM_SCHEMA**という名前の特別なスキーマが割り当てられます。このスキーマを修正・削除することはできません。このスキーマは、テーブルインスペクタパレットに表示されるスキーマリストには含まれません。すべてのユーザは、読み込み専用でこのスキーマにアクセスできます。

## SQLソースへの接続

4D SQLサーバのレベルでは、マルチデータベースのアーキテクチャが実装されています。4Dでは以下のことが可能です:

- **SQL LOGIN**コマンドを使用して、既存のデータベースにログインする。
- あるデータベースから他のデータベースに、**SQL LOGIN**と**SQL LOGOUT**コマンドを使用してスイッチする。
- **USE DATABASE**コマンドを使用して、カレントデータベースに代わり、別の4Dデータベースを開いて使用する。

## 主キー

SQLランゲージでは主キーを使用して、テーブル中のレコード (ロー) を指定するカラム (フィールド) 決定します。主キーの設定は特に4Dテーブルのレコード複製機能において必要となります (**SQLを使用した複製**参照)。

4Dでは2つの方法を使用してテーブルの主キーを管理できます:

- SQLランゲージを使用する
- 4Dのストラクチャエディタを使用する

### SQLランゲージを使用して主キーを設定する

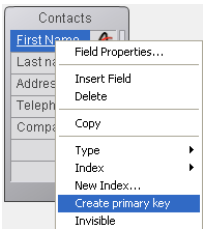
テーブルを作成するとき (*CREATE TABLE*コマンドを使用) またはカラムを追加や変更するとき (*ALTER TABLE*コマンドを使用)、主キーを設定できます。主キーは**PRIMARY KEY**句とそれに続くカラム名またはカラムリストで指定されます。 詳細はを参照してください。

### ストラクチャエディタを使用して主キーを設定する

4Dではストラクチャエディタのコンテキストメニューを使用して直接主キーを作成したり削除したりできます。

主キーを作成するには:

1. テーブルの主キーを構成するフィールドを選択します。
2. 右クリックしてコンテキストメニューから**主キーを作成**コマンドを選択します:



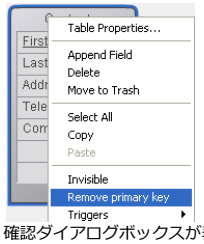

主キーに含められたフィールドにはエディタ中で下線が引かれ、SQL定義にはPRIMARY KEYキーワードが表示されます。

主キーに属するフィールドには重複する値が含まれてはいけません。既にテーブルレコードに重複する値が存在する場合、このことを知らせる警告ダイアログが表示されます。

**Note:** 主キーに属するカラムはNULL値を受け入れません。

テーブルから主キーを取り除くには:

1. 主キーを含むテーブル上で右クリックし、コンテキストメニューから**主キーを取り除く**コマンドを選択します:




確認ダイアログボックスが表示されます。**OK**をクリックすると即座に主キーが取り除かれます。

# SQLを使用した複製

4DはSQLを使用して2つ以上の4Dデータベース間でデータを同期する新しいメカニズムを提供します。このメカニズムを使用して1つ以上のミラーデータベースをセットアップし、データの可用性を保証することができます。

これは以下のように動作します: ローカルデータベースがリモートソースデータベースのデータをローカルに複製します。更新はローカルデータベースが定期的にリモートデータベースからデータを取得することで実行されます。複製はテーブルレベルで行われます。リモートデータベーステーブルのデータをローカルデータベースのテーブルに複製します。
これはスタンプおよびSQLコマンドによって可能となります。

ストラクチャエディタでテーブルプロパティを使用してリモートおよびローカルデータベースで複製メカニズムを有効にできます。ローカル側ではSQLの**REPLICATE**コマンドを使用してリモートデータベースのテーブルからデータを取得し、このデータをローカルデータベースのテーブルに統合します。SQLの**SYNCHRONIZE**コマンドを使用すれば2つのテーブルを同期できます。

## 仮想フィールド

4Dデータベースのそれぞれのテーブルに3つの"仮想"フィールド、__ROW_ID、__ROW_STAMP、そして__ROW_ACTIONが割り当てられることがあります。これらのフィールドは特別なプロパティを持つため、"標準"のフィールドと区別するために"仮想"と呼ばれます。これらのフィールドは自動で値が割り当てられ、ユーザーは値を読むことはできますが書き込むことはできず、そしてデータベースのシステムテーブルには現れません。以下の表はこれらのフィールドと利用モードを説明しています:

| 仮想フィールド | 型 | 内容 | 利用 |
|---|---|---|---|
| __ROW_ID | Int32 | レコードのID | **REPLICATE**や**SYNCHRONIZE**を除くすべてのSQLステートメント |
| __ROW_STAMP | Int64 | レコード複製情報 | すべてのSQLステートメント |
| __ROW_ACTION | Int16 | レコードに対して実行されたアクション: 1 = 追加または更新、2 =削除 | **REPLICATE**または**SYNCHRONIZE**のみ |

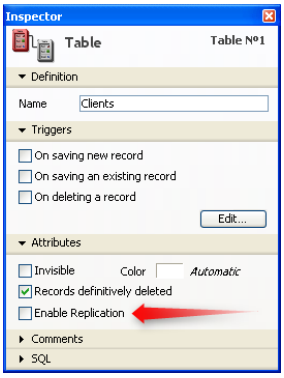
複製メカニズムが有効になると、レコードが作成、更新、削除されるたびに対応する情報が自動でこのレコードの仮想フィールドに保存されます。

## 複製を有効にする

複製を可能にするメカニズムはデフォルトで無効になっています。複製や同期を行うテーブルごとに、リモートおよびローカルのデータベース両方で、明示的にこれを有効にしなければなりません。

メカニズムを有効にしても複製自身が実行されるわけではない点に留意してください。データが実際にローカルまたは同期データベースに複製されるためには、**REPLICATE**や**SYNCHRONIZE**コマンドを使用しなければなりません。

(ローカルおよびリモートデータベース上で) 内部的な複製メカニズムをテーブル毎に有効にするためには、テーブルインスペクタ内の**複製を有効にする**テーブルプロパティを使用します:




**Note:** 複製メカニズムが動作するようにするためには、リモートおよびローカルデータベース側でテーブルの主キーを指定しなければなりません。このキーはストラクチャエディタまたはSQLコマンドで指定できます。主キーが設定されていない場合、このオプションは選択することができません。

このオプションがチェックされると、4Dはテーブルのレコードを複製するために必要な情報 (特にテーブルの主キーに基づき) を生成します。 この情報は仮想 __ROW_STAMP と__ROW_ACTION フィールドに格納されます。

**Note:** SQLの*CREATE TABLE*と*ALTER TABLE*コマンドを使用して複製情報の作成を有効にしたり無効にしたりできます。これを行うためには**ENABLE REPLICATE**と**DISABLE REPLICATE**キーワードを使用します。

**警告**: このオプションをチェックすると、複製メカニズムに必要な情報が公開されるようになります。データベース保護のため、 -- データが公開されている場合にそこへのアクセスを制限するのと同様に -- この情報へのアクセスを制限しなければなりません。結果、このオプションを使用して複製システムを実装する場合、以下の点を確認しなければなりません:

- SQLサーバーが起動されている場合、アクセスは4Dパスワードシステムと、かつ/またはSQLスキーマで制限されなければなりません (**4D SQLサーバの設定**参照)。
- HTTPサーバーが起動されている場合、アクセスは4Dパスワードと、かつ/またはSQLスキーマ (**4D SQLサーバの設定**参照) かつ/または**On Web Authenticationデータベースメソッド**かつ/または**SET TABLE TITLES**と**SET FIELD TITLES**コマンドを使用した仮想ストラクチャーで制限されなければなりません。詳細は**MissingRef**の"4DSYNC/URL"を参照してください。

## ローカルデータベース側の更新

データベースごとのテーブルごとに複製メカニズムが有効になると、ローカルデータベースからSQLの**REPLICATE**コマンドを使用できます。詳細はこのコマンドの説明を参照してください。

# 結合のサポート

4DのSQLエンジンは結合のサポートを拡張します。

結合操作はINNERまたはOUTER、そして明示または暗黙です。暗黙のINNER結合は*SELECT*コマンドでサポートされます。さらに**JOIN**キーワードを使用して生成される明示的なINNER結合とOUTER結合が使用できます。

**Note:** 4D SQLエンジンの結合の現在の実装には以下が含まれていません:

- NATURAL結合
- INNER結合のUSING構成

## 概要

結合操作は2つ以上のテーブル間のレコードの接続を作成し、結果を結合と呼ばれる新しいテーブルに一体化させるために使用します。

結合条件を指定した*SELECT*ステートメントを使用して結合を生成します。明示的な結合では、これらの条件を複合にすることができますが、常に結合に含まれる列間の等価比較に基づくものでなければなりません。例えば >= 演算子を明示的な結合条件に使用することはできません。暗黙的な結合ではすべてのタイプの比較を使用できます。
内部的に、等価比較は4Dエンジンにより直接実行されるので、素早く実行されます。

**Note:** 通常データベースエンジンでは、テーブルの順番は検索の間に指定された順番です。しかし結合を使用する際、テーブルの順番はテーブルリストで決定されます。以下の例題で:

```
SELECT * FROM T1 RIGHT OUTER JOIN T2 ON T2.depID = T1.depID;
```

テーブルの順番は (テーブルリストに現れる通り) T1そしてT2です。 (結合条件に現れる順である) T2そしてT1ではありません。

## 明示的なINNER結合

INNER結合は2つの列間で一致する値を探す比較に基づきます。例えば以下の3つのテーブルがあるとき:

- Employees

| name | depID | cityID |
|---|---|---|
| Alan | 10 | 30 |
| Anne | 11 | 39 |
| Bernard | 10 | 33 |
| Mark | 12 | 35 |
| Martin | 15 | 30 |
| Philip | NULL | 33 |
| Thomas | 10 | NULL |

- Departments

| depID | depName |
|---|---|
| 10 | Program |
| 11 | Engineering |
| NULL | Marketing |
| 12 | Development |
| 13 | Quality |

- Cities

| cityID | cityName |
|---|---|
| 30 | Paris |
| 33 | New York |
| NULL | Berlin |

**Note:** この例のストラクチャはこのセクションを通して使用されます。

暗黙的なINNER結合の例は以下の通りです:

```
SELECT *
        FROM employees, departments
        WHERE employees.DepID = departments.DepID;
```

4Dでは、JOINキーワードを使用して明示的なINNER結合を指定できます:

```
SELECT *
        FROM employees
        INNER JOIN departments
            ON employees.DepID = departments.DepID;
```

このクエリを以下のように4Dコードに挿入できます:

```
ARRAY TEXT(aName;0)
ARRAY TEXT(aDepName;0)
ARRAY INTEGER(aEmpDepID;0
ARRAY INTEGER(aDepID;0)
```

```
Begin SQL
       SELECT *
       FROM employees
               INNER JOIN departments
                   ON employees.depID = departments.depID
               INTO :aName, :aEmpDepID, :aDepID, :aDepName;
End SQL
```

この結合結果は以下のとおりとなります:

| aName | aEmpDepID | aDepID | aDepName |
|---|---|---|---|
| Alan | 10 | 10 | Program |
| Anne | 11 | 11 | Engineering |
| Bernard | 10 | 10 | Program |
| Mark | 12 | 12 | Development |
| Thomas | 10 | 10 | Program |

従業員名がPhilipとMartin、また部署名がMarketingとQualityは結合の結果に現れません。なぜなら:

- Philipに部署は割り当てられていません (NULL値)。
- Martinの部署IDはDepartmentsテーブルに存在しません。
- Quality部署に割り当てられた従業員はいません (ID 13)。
- Marketing部署にIDが割り当てられていません (NULL値)。

### CROSS結合

CROSS結合あるいはデカルト積は**WHERE**も**ON**句も指定されていないINNER結合です。これはあるテーブルのそれぞれの行を他のテーブルのそれぞれの行に関連付けるものです。
CROSS結合の結果はテーブルのデカルト積であり、2つのテーブルの行数の積が総行数として含まれます。この積は両テーブルの行を結合した場合のすべての組み合わせを表します。

以下のシンタックスはすべて同等です:

```
SELECT * FROM T1 INNER JOIN T2
SELECT * FROM T1, T2
SELECT * FROM T1 CROSS JOIN T2;
```

以下はCROSS結合を行う4Dコードの例です:

```
ARRAY TEXT(aName;0)
ARRAY TEXT(aDepName;0)
ARRAY INTEGER(aEmpDepID;0
ARRAY INTEGER(aDepID;0)
Begin SQL
       SELECT *
       FROM employees CROSS JOIN departments
               INTO :aName, :aEmpDepID, :aDepID, :aDepName;
End SQL
```

以下は例題データベースにおけるこの結合の結果です:

| aName | aEmpDepID | aDepID | aDepName |
|---|---|---|---|
| Alan | 10 | 10 | Program |
| Anne | 11 | 10 | Program |
| Bernard | 10 | 10 | Program |
| Mark | 12 | 10 | Program |
| Martin | 15 | 10 | Program |
| Philip | NULL | 10 | Program |
| Thomas | 10 | 10 | Program |
| Alan | 10 | 11 | Engineering |
| Anne | 11 | 11 | Engineering |
| Bernard | 10 | 11 | Engineering |
| Mark | 12 | 11 | Engineering |
| Martin | 15 | 11 | Engineering |
| Philip | NULL | 11 | Engineering |
| Thomas | 10 | 11 | Engineering |
| Alan | 10 | NULL | Marketing |
| Anne | 11 | NULL | Marketing |
| Bernard | 10 | NULL | Marketing |
| Mark | 12 | NULL | Marketing |
| Martin | 15 | NULL | Marketing |
| Philip | NULL | NULL | Marketing |
| Thomas | 10 | NULL | Marketing |
| Alan | 10 | 12 | Development |
| Anne | 11 | 12 | Development |
| Bernard | 10 | 12 | Development |
| Mark | 12 | 12 | Development |
| Martin | 15 | 12 | Development |
| Philippe | NULL | 12 | Development |
| Thomas | 10 | 12 | Development |
| Alain | 10 | 13 | Quality |
| Anne | 11 | 13 | Quality |
| Bernard | 10 | 13 | Quality |
| Mark | 12 | 13 | Quality |
| Martin | 15 | 13 | Quality |
| Philip | NULL | 13 | Quality |
| Thomas | 10 | 13 | Quality |

**Note:** パフォーマンスの理由から、CROSS結合は注意して使用しなければなりません。

## OUTER結合

4DでOUTER結合を生成できるようになりました。OUTER結合では、結合させるテーブルの行間で値が一致する必要はありません。結果のテーブルには、たとえ一致する行がなくても、テーブル (または少なくとも1つの結合されたテーブル)のすべての行が含まれます。 これは、たとえ異なる結合されたテーブル間でぎょうが完全に値を持っていない場合でも、あるテーブルのすべての情報が使用できることを意味します。

OUTER結合には**LEFT**、**RIGHT**、そして**FULL**キーワードで指定される3つのタイプがあります。**LEFT**と**RIGHT**は (**JOIN** キーワードの左あるいは右どちらに位置するかで) すべてのデータが処理されるテーブルを示すために使用されます。**FULL**は両側のOUTER結合を示します。

**Note:** 4Dでは明示的なOUTER結合だけがサポートされます。

### LEFT OUTER結合

LEFT OUTER結合 (またはLEFT結合) の結果には、たとえ結合条件に一致するレコードがキーワードの右側のテーブルにない場合でも、常にキーワードの左にあるテーブルのすべてのレコードが含まれます。これは検索が右側のテーブル内で一致する行を見つけることができない左テーブルの各行についても、結合はそれらを結果に含めることを意味します。しかしこの場合右テーブルの各列にはNULL値が入ります。言い換えればLEFT OUTER結合は左側のテーブルのすべての行と、結合条件に一致する右テーブルの行 (または一致しない場合NULL) を作成します。左テーブルの1つの行に対し、結合述部に一致する右テーブルの行が複数ある場合、左テーブルの値は右テーブルの各行に対し繰り返される点に留意してください。

以下はLEFT OUTER結合を行う4Dコードの例です:

```
ARRAY TEXT(aName;0)
ARRAY TEXT(aDepName;0)
ARRAY INTEGER(aEmpDepID;0
ARRAY INTEGER(aDepID;0)
Begin SQL
       SELECT *
           FROM employees
           LEFT OUTER JOIN departments
               ON employees.DepID = departments.DepID;
               INTO :aName, :aEmpDepID, :aDepID, :aDepName;
End SQL
```

以下は例題データベースでこの結合を実行したときの結果です (赤は追加の行を示します):

| aName | aEmpDepID | aDepID | aDepName |
|---|---|---|---|
| Alan | 10 | 10 | Program |
| Anne | 11 | 11 | Engineering |
| Bernard | 10 | 10 | Program |
| Mark | 12 | 12 | Development |
| Thomas | 10 | 10 | Program |
| Martin | 15 | NULL | NULL |
| Philip | NULL | NULL | NULL |

### RIGHT OUTER結合

RIGHT OUTER結合は完全にLEFT OUTER結合の反対です (前の段落参照)。たとえ結合条件に一致するレコードが左のテーブルに見つからない場合でも、結果には常に**JOIN**キーワードの右側のテーブルのすべてのレコードが含まれます。

以下はRIGHT OUTER結合を行う4Dコードの例です:

```
ARRAY TEXT(aName;0)
ARRAY TEXT(aDepName;0)
ARRAY INTEGER(aEmpDepID;0
ARRAY INTEGER(aDepID;0)
Begin SQL
       SELECT *
           FROM employees
           RIGHT OUTER JOIN departments
               ON employees.DepID = departments.DepID;
               INTO :aName, :aEmpDepID, :aDepID, :aDepName;
End SQL
```

以下は例題データベースでこの結合を実行したときの結果です (赤は追加の行を示します):

| aName | aEmpDepID | aDepID | aDepName |
|---|---|---|---|
| Alan | 10 | 10 | Program |
| Anne | 11 | 11 | Engineering |
| Bernard | 10 | 10 | Program |
| Mark | 12 | 12 | Development |
| Thomas | 10 | 10 | Program |
| NULL | NULL | NULL | Marketing |
| NULL | NULL | 13 | Quality |

### FULL OUTER結合

FULL OUTER結合は単純にLEFT OUTER結合とRIGHT OUTER結合の結果を組み合わせたものです。結果の結合テーブルには左および右のレコードがすべて含まれます。それぞれの側で失われたフィールドはNULL値になります。

以下はFULL OUTER結合を行う4Dコードの例です:

```
ARRAY TEXT(aName;0)
ARRAY TEXT(aDepName;0)
ARRAY INTEGER(aEmpDepID;0
ARRAY INTEGER(aDepID;0)
Begin SQL
       SELECT *
           FROM employees
           FULL OUTER JOIN departments
               ON employees.DepID = departments.DepID;
               INTO :aName, :aEmpDepID, :aDepID, :aDepName;
End SQL
```

以下は例題データベースでこの結合を実行したときの結果です (赤は追加の行を示します):

| aName | aEmpDepID | aDepID | aDepName |
|---|---|---|---|
| Alan | 10 | 10 | Program |
| Anne | 11 | 11 | Engineering |
| Bernard | 10 | 10 | Program |
| Mark | 12 | 12 | Development |
| Thomas | 10 | 10 | Program |
| Martin | 15 | NULL | NULL |
| Philip | NULL | NULL | NULL |
| NULL | NULL | NULL | Marketing |
| NULL | NULL | 13 | Quality |

### 1つのステートメント内で複数の結合

1つの*SELECT*ステートメント内で複数の結合を組み合わせることができます。また暗黙的または明示的なINNER結合と明示的なOUTER結合を組み合わせることもできます。

以下は複数の結合を行う4Dコードの例です:

```
ARRAY TEXT(aName;0)
ARRAY TEXT(aDepName;0)
ARRAY TEXT(aCityName;0)
ARRAY INTEGER(aEmpDepID;0
ARRAY INTEGER(aEmpCityID;0)
ARRAY INTEGER(aDepID;0)
```

```
ARRAY INTEGER(aCityID;0)
Begin SQL
       SELECT *
           FROM (employees RIGHT OUTER JOIN departments
                       ON employees.depID = departments.depID)
                       LEFT OUTER JOIN cities
                       ON employees.cityID = cities.cityID
           INTO :aName, :aEmpDepID, :aEmpCityID, :aDepID, :aDepName, :aCityID, :aCityName;
End SQL
```

以下は例題データベースでこの結合を実行したときの結果です:

| aName | aEmpDepID | aEmpCityID | aDepID | aDepName | aCityID | aCityName |
|---|---|---|---|---|---|---|
| Alan | 10 | 30 | 10 | Program | 30 | Paris |
| Anne | 11 | 39 | 11 | Engineering | 0 | |
| Bernard | 10 | 33 | 10 | Program | 33 | New York |
| Mark | 12 | 35 | 12 | Development | 0 | |
| Thomas | 10 | NULL | 10 | Program | 0 | |

# SQLコマンド

- SQLコマンド
- SELECT
- INSERT
- UPDATE
- DELETE
- CREATE DATABASE
- USE DATABASE
- ALTER DATABASE
- CREATE TABLE
- ALTER TABLE
- DROP TABLE
- CREATE INDEX
- DROP INDEX
- LOCK TABLE
- UNLOCK TABLE
- EXECUTE IMMEDIATE
- CREATE SCHEMA
- ALTER SCHEMA
- DROP SCHEMA
- GRANT
- REVOKE
- REPLICATE
- SYNCHRONIZE

## SQLコマンド

---

SQLコマンド (もしくは文) は一般的に2つのカテゴリに分類されます:

- データ操作コマンド: データベース情報を取得、追加、削除、更新するために使用します。これらは特に*SELECT*、**INSERT**、*UPDATE*、そ して*DELETE*コマンドを指します。
- データ定義コマンド: データベースオブジェクトを作成または削除するために使用します。これらは特に**CREATE DATABASE**、**CREATE TABLE**、*ALTER TABLE*、*DROP INDEX*、*DROP TABLE*そして*CREATE SCHEMA*コ マンドを指します。

シンタックス中で、コマンド名とキーワードは太字で示され、そのまま渡されます。斜体で示される他の要素は、""テーマで別途説明しま す。角括弧[ ]に渡されるキーワードや句は省略可能です。縦線文字は選択可能な候補を表します。要素が波括弧{}の中で縦線文字で区切られている場合には、この中から 一つだけが渡されるべきであることを示しています。

# SELECT

```
SELECT [ALL | DISTINCT]
{* | select_item, ..., select_item}
FROM table_reference, ..., table_reference
[WHERE search_condition]
[ORDER BY sort_list]
[GROUP BY sort_list]
[HAVING search_condition]
[LIMIT {4d_language_reference |int_number | ALL}]
[OFFSET 4d_language_reference |int_number]
[INTO {4d_language_reference, ..., 4d_language_reference}]
[FOR UPDATE]
```

## 説明

*SELECT*コマンドは一つ以上のテーブルからデータを取得するために使用します。
*を渡すと、すべてのカラムを取得します。そうでなければ、取得するカラムを個々に指定するために、1つ以上の*select_item*を (コンマで区切って) 渡すことができます。オプションのキーワード**DISTINCT**を*SELECT*文に渡すと、重複データは返されません。
"*"とフィールドを混在させたクエリを行うことはできません。例えば、以下の文:

```
SELECT *, SALES, TARGET FROM OFFICES
```

は許可されません。対して

```
SELECT * FROM OFFICES
```

が可能です。

**FROM**句は、データを取得するテーブルを*table_reference*で指定するために使用します。標準のSQL名または文字列を渡すことができます。テーブル名の場所にクエリ式を渡すことはできません。オプションのキーワード**AS**を渡して、カラムにエイリアスを割り当てることもできます。このキーワードが渡される際は、エイリアス名を続けなければなりません。SQL名または文字列を使用できます。

オプションの**WHERE**句は、データを選択するための条件を指定します。これは*search_condition*を渡すことで行われ、**FROM**句により取得されるデータに適用されます。*search_condition*は常にブール型の値を返します。

オプションの**ORDER BY**句は、選択されたデータに*sort_list*条件を適用するために使用します。**ASC**や**DESC**キーワードを追加して、並び替えの昇順・降順を指定することもできます。デフォルトで昇順が適用されます。

オプションの**GROUP BY**句は、渡された*sort_list*条件に基づいて、同じデータをグループ化するために使用します。複数のグループカラムを渡すことができます。この句は冗長を避けたり、集約関数 (*SUM*、*COUNT*、*MIN*、*MAX*) を計算するために使用できます。これらはグループに対して適用されます。**ORDER BY**句と同様、**ASC**や**DESC**キーワードを追加できます。

オプションの**HAVING**句は、これらのグループのひとつに*search_condition*を適用するために使用します。**HAVING**句は**GROUP BY**句なしで渡すこともできます。

オプションの**LIMIT**句は、返されるデータ数を*4d_language_reference*変数または*int_number*に制限するために使用します。

オプションの**OFFSET**句は、**LIMIT**句のカウントを開始する前にスキップするデータの数 (*4d_language_reference*変数または*int_number*) を指定するために使用します。

オプションの**INTO**句は、データを格納する変数*4d_language_reference*を指定するために使用します。

**FOR UPDATE**句が指定された*SELECT*コマンドは、選択されたすべてのレコードへの排他的な書き込みロックを試みます。ロックできないレコードがあると、コマンドの実行に失敗し、エラーが生成されます。すべてのレコードがロックされると、カレントのトランザクションがコミットされるかロールバックされるまで、レコードはロックされたままとなります。

## 例題 1

映画のタイトル、公開年、チケット販売数が登録された一つのテーブルからなるデータベースがあります。
1979年以降の映画において、チケット販売総数が年合計1000万未満の年とそのチケット販売総数を取得します。最初の5年をスキップし、その後の10年間のみを表示し、年で並び替えます.

```
C_LONGINT($MovieYear;$MinTicketsSold;$StartYear;$EndYear)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aTicketsSold;0)
$MovieYear:=1979
$MinTicketsSold:=10000000
$StartYear:=5
$EndYear:=10

Begin SQL
   SELECT Year_of_Movie, Sum(Tickets_Sold)
   FROM MOVIES
   WHERE Year_of_Movie >= :$MovieYear
   GROUP BY Year_of_Movie
   HAVING Sum(Tickets_Sold) < :$MinTicketsSold
```

```
   ORDER BY 1
   LIMIT :$EndYear
   OFFSET :$StartYear
   INTO :aMovieYear, :aTicketsSold;
End SQL
```

### 例題 2

これは複合検索条件を使用する例です:

```
SELECT supplier_id
FROM suppliers
WHERE (name = 'CANON')
OR (name = 'Hewlett Packard' AND city = 'New York')
OR (name = 'Firewall' AND status = 'Closed' AND city = 'Chicago');
```

### 例題 3

SALES_PERSONSテーブルから売り上げ情報を取得します。QUOTAは売り上げ目標、SALESは実際の売り上げです:

```
ARRAY REAL(Min_Values;0)
ARRAY REAL(Max_Values;0)
ARRAY REAL(Sum_Values;0)
Begin SQL
   SELECT MIN ( ( SALES * 100 ) / QUOTA ),
   MAX ( ( SALES * 100 ) / QUOTA ),
   SUM ( QUOTA ) - SUM ( SALES )
   FROM SALES_PERSONS
   INTO :Min_Values, :Max_Values, :Sum_Values;
End SQL
```

### 例題 4

これは指定した都市が出身地の俳優を検索する例です:

```
ARRAY TEXT(aActorName;0)
ARRAY TEXT(aCityName;0)
Begin SQL
   SELECT ACTORS.FirstName, CITIES.City_Name
   FROM ACTORS AS 'Act', CITIES AS 'Cit'
   WHERE Act.Birth_City_ID=Cit.City_ID
   ORDER BY 2 ASC
   INTO : aActorName, : aCityName;
End SQL
```

# INSERT

**INSERT INTO** {*sql_name* | *sql_string*}
[(*column_reference*, ..., *column_reference*)]
[**VALUES**(([**INFILE**]*arithmetic_expression* |**NULL**}, ..., {[**INFILE**]*arithmetic_expression* |**NULL**};) |*subquery*]

## 説明

**INSERT**コマンドは既存のテーブルにデータを追加するために使用します。データを追加するテーブルは*sql_name*または*sql_string*を使用して渡されます。オプションの*column_reference*型の引数が渡されると、値が挿入されるカラム名を指定します。*column_reference*が渡されない場合、値はデータベース中の順番通りに格納されます (最初の値は一番目のカラムに、二番目の値は二番目のカラムに)。

**VALUES**キーワードは、指定したカラムに格納する値を渡すために使用します。*arithmetic_expression*または**NULL**を渡すことができます。さらに値として渡す一連のデータを挿入するために、*subquery*を**VALUES**キーワード中に渡すこともできます。
**VALUES**キーワード中に渡す値の数は*column_reference*型の引数で渡した数と一致しなければならず、それぞれの値とカラムのデータ型は一致しているか、少なくとも変換可能でなければなりません。

**INFILE**キーワードは、外部ファイルの内容を使用して新しいレコードの値を指定することを可能にします。このキーワードはVARCHAR型の式とともに使用されなければなりません。**INFILE**キーワードが渡される場合、*arithmetic_expression*値はファイルパス名として評価されます。ファイルが見つかると、ファイルの内容が対応するカラムに挿入されます。テキストまたはBLOB型のフィールドだけが**INFILE**から値を受け取ることができます。ファイルの内容は変換されず、ローデータとして転送されます。
クエリがリモートから行われる場合でも、検索するファイルはSQLエンジンをホストするマシン上に存在しなければなりません。同様に、パス名もSQLエンジンが動作するOSのシンタックスで表現されなければなりません。パスは絶対または相対パスが使用できます。

**INSERT**コマンドはシングルおよびマルチロークエリでサポートされています。しかしマルチローの**INSERT**文はUNIONおよびJOIN操作を許可しません。

4Dエンジンはマルチローの値挿入が可能で、特に大量のデータを挿入するような際にコードを簡略および最適化することができます。マルチローの挿入のシンタックスは以下の通りです:

```
INSERT INTO {sql_name | sql_string}
[(column_ref, ..., column_ref)]
VALUES(arithmetic_expression, ..., arithmetic_expression), ..., (arithmetic_expression, ...,
arithmetic_expression);
```

このシンタックスは例題3と4で説明しています。

## 例題 1

これはtable2のセレクションをtable1に挿入する例題です:

```
INSERT INTO table1 (SELECT * FROM table2)
```

## 例題 2

この例題はテーブルを作成し、値を挿入します:

```
CREATE TABLE ACTOR_FANS
(ID INT32, Name VARCHAR);
INSERT INTO ACTOR_FANS
(ID, Name)
VALUES (1, 'Francis');
```

## 例題 3

複数行のシンタックスにより、うんざりするような行の繰り返しを避けることができます。

```
INSERT INTO MyTable
(Fld1,Fld2,BoolFld,DateFld,TimeFld, InfoFld)
VALUES
(1,1,1,'11/01/01','11:01:01','First row'),
(2,2,0,'12/01/02','12:02:02','Second row'),
(3,3,1,'13/01/03','13:03:03','Third row'),
……
(7,7,1,'17/01/07','17:07:07','Seventh row');
```

## 例題 4

複数行のシンタックスの他、4Dの変数、または配列を使用することもできます。 ただし複数行シンタックスと変数、配列シンタックスを混ぜて使用することはできません。

```
INSERT INTO MyTable
```

```
(Fld1,Fld2,BoolFld,DateFld,TimeFld, InfoFld)
VALUES
(:vArrId, :vArrIdx, :vArrbool, :vArrdate, :vArrL, :vArrText);
```

# UPDATE

```
UPDATE {sql_name | sql_string}
SET sql_name = {arithmetic_expression |NULL}, ..., sql_name = {arithmetic_expression |NULL}
[WHERE search_condition]
```

## 説明

*UPDATE*コマンドは*sql_name*または*sql_string*で指定したテーブルに含まれるデータを更新するために使用します。
**SET**句は新しい値 (*arithmetic_expression*または**NULL**) を*sql_name*型の引数に割り当てるために使用します。
オプションの**WHERE**句は、どのデータ (*search_condition*に該当する) を更新するかを指定します。**WHERE**句が渡されないと、テーブルのすべてのデータに**SET**句で渡された新しい値が割り当てられます。

*UPDATE*コマンドはクエリおよびサブクエリでサポートされています。しかし、位置づけされた*UPDATE*文はサポートされません。
**CASCADE**更新が4Dに実装されています。しかし**SET NULL**と**SET DEFAULT**削除ルールはサポートされていません。

## 例題

この例題では、MOVIESテーブルを更新して、"Air Force One"のチケット販売数を3,500,000にします:

```
UPDATE MOVIES
SET Tickets_Sold = 3500000
WHERE TITLE = 'Air Force One';
```

# DELETE

```
DELETE FROM {sql_name | sql_string}
[WHERE search_condition]
```

## 説明

*DELETE*コマンドは、**FROM**キーワードの後に渡した*sql_name*または*sql_string*で指定されたテーブルから、すべてまたは一部のデータを削除するために使用します。

オプションの**WHERE**句は、(*search_condition*に該当する) データのどの部分を削除するか指定します。これが渡されない場合、テーブルのすべてのデータが削除されます。

位置づけされた*DELETE*文はサポートされていません。**CASCADE**削除は4Dに実装されています。しかし**SET DEFAULT**および**SET NULL**削除ルールはサポートされていません。

## 例題

この例題では、MOVIESテーブルから、2000年以前に公開されたすべての映画を削除します:

```
DELETE FROM MOVIES
WHERE Year_of_Movie <= 2000;
```

# CREATE DATABASE

CREATE DATABASE [IF NOT EXISTS] DATAFILE <*Complete pathname*>

## 説明

**CREATE DATABASE**コマンドを使用して新しいエクスターナルデータベース (.4dbと.4dd ファイル) を指定した場所に作成できます。

**IF NOT EXISTS**制約が渡されていると、指定した場所に同じ名前のデータベースが存在すれば、データベースは作成されずエラーも生成されません。
**IF NOT EXISTS**制約が渡されていない場合、指定した場所に同じ名前のデータベースが既に存在すれば、データベースは作成されず、"データベースは既に存在します。CREATE DATABASEコマンドの実行に失敗しました。"エラーメッセージが表示されます。

**DATAFILE**句に新しいエクスターナルデータベースの完全名 (完全パス名 + 名前) を指定します。ストラクチャファイルの名前を渡さなければなりません。プログラムは (指定されていなければ) 自動で".4db"拡張子をファイルに追加し、データファイルを作成します。パス名はPOSIXシンタックスあるいはシステムシンタックスで指定します。また絶対、あるいはメインの4Dデータベースのストラクチャファイルからの相対で指定できます。

- POSIXシンタックス (URL型): 使用するプラットフォームに関係なく、フォルダ名はスラッシュ ("/") で区切られます。
  例えば: ".../extdatabases/myDB.4db"
  絶対パスの場合、先頭はボリューム名とコロンを渡します。例えば: "C:/test/extdatabases/myDB.4db"
- システムシンタックス: 現在のプラットフォームに基づくシンタックスのパス名。例えば:
  - (Mac OS) HD:Applications:myserv:extdatabases:myDB.4db
  - (Windows) C:¥Applications¥myserv¥extdatabases¥myDB.4db

**CREATE DATABASE**コマンドの実行に成功しても、作成された新しいデータベースは自動ではカレントデータベースにはなりません。これを行うには、新しい**USE DATABASE**コマンドを使用して明示的にカレントデータベースとして宣言しなければなりません。

### エクスターナルデータベースについて

エクスターナルデータベースとはメインの4Dデータベースから独立していますが、メインのデータベースから使用できる4Dデータベースです。エクスターナルデータベースの利用は、このデータベースを一時的にカレントデータベース、言い換えれば4Dにより実行されるSQLクエリのターゲットデータベースとして指定することを意味します。デフォルトでメインデータベースがカレントデータベースです。

エクスターナルデータベースは**CREATE DATABASE**を使用してメインデータベースから直接作成できます。エクスターナルデータベースを作成したら、それを**USE DATABASE**コマンドを使用してカレントデータベースとして指定できます。そして標準のSQLコマンド (*CREATE TABLE*や*ALTER TABLE*等) でストラクチャを変更し、データを格納できます。**DATABASE_PATH**関数を使用していつでもカレントデータベースを知ることができます。
エクスターナルデータベースの主たる関心は、4Dコンポーネントから作成および処理が可能であるということです。この機能により、必要に応じてテーブルやフィールドを作成するコンポーネントが開発できるようになります。

**注:** エクスターナルデータベースは標準の4Dデータベースです。4Dや4D Serverアプリケーションでそれをメインデータベースとして開き、操作することができます。逆に、標準の4Dデータベースをエクスターナルデータベースとして使用することができます。しかしエクスターナルデータベースに (Designerパスワードを設定することで) アクセス管理システムを有効化することはできません。有効にすると、**USE DATABASE**を使用してアクセスすることができなくなってしまいます。

## 例題 1

C:/MyDatabase/にエクスターナルデータベースExternalDB.4DBとExternalDB.4DDを作成する:

```
Begin SQL
   CREATE DATABASE IF NOT EXISTS DATAFILE 'C:/MyDatabase/ExternalDB';
End SQL
```

## 例題 2

メインデータベースのストラクチャファイルと同階層にエクスターナルデータベースTestDB.4DBとTestDB.4DDを作成する:

```
Begin SQL
   CREATE DATABASE IF NOT EXISTS DATAFILE 'TestDB';
End SQL
```

## 例題 3

ユーザが指定した場所にエクスターナルデータベースExternal.4DBとExternal.4DDを作成する:

```
C_TEXT($path)
$path:=Select folder("Destination folder of external database:")
$path:=$path+"External"
Begin SQL
   CREATE DATABASE DATAFILE <<$path>>;
End SQL
```

# USE DATABASE

```
USE [LOCAL | REMOTE] DATABASE
{DATAFILE <Complete pathname> | SQL_INTERNAL | DEFAULT}
[AUTO_CLOSE]
```

## 説明

**USE DATABASE**コマンドを使用してエクスターナルデータベースをカレントデータベースとして指定できます。これによりカレントプロセス内での次のSQLクエリ送信先が変更されます。**Begin SQL**/**End SQL**構造内、**SQL EXECUTE**および**SQL EXECUTE SCRIPT**コマンド等、すべてのタイプのSQLクエリに効果が及びます。

**Note:** エクスターナルデータベースに関する情報は**CREATE DATABASE**コマンドの説明を参照してください。

- シングルユーザ環境で動作している場合、エクスターナルデータベースは4Dと同じマシン上に存在しなければなりません。
- リモートモードで動作している場合、エクスターナルデータベースはローカルマシンまたは4D Serverマシン上に配置できます。

4Dをリモートモードで使用している場合、**REMOTE**キーワードを使用して4D Server上のエクスターナルデータベースを指定できます。
安全のため、このメカニズムはネイティブなリモート接続、つまり4D Serverに接続したリモート4Dデータベースでのみ動作します。ODBCやパススルー経由の接続では使用できません。
キーワードが指定されない場合、デフォルトで**LOCAL**オプションが使用されます。4Dをローカルモードで使用している場合、**REMOTE**および**LOCAL**キーワードは無視されます。接続は常にローカルです。

使用するエクスターネルデータベースを指定するには、その完全パス名 (アクセスパス + 名前) を**DATAFILE**句に渡します。パスはPOSIXシンタックスあるいはシステムシンタックスのいずれかを使用できます。パスは絶対、あるいはメイン4Dデータベースのストラクチャファイルから相対で表します。
リモートモードでは、**REMOTE**キーワードが渡されると、この引数はサーバーマシンからのデータベースパスを指定します。このキーワードが省略されるか**LOCAL**キーワードが渡されると、この引数はローカル4Dマシンのデータベースパスを指定します。

**重要:** アクセスコントロールが有効にされていない (Designerパスワードが指定されていない)、利用可能なエクスターナルデータベースを指定しなければなりません。そうでなければエラーが生成されます。

メインデータベースをカレントデータベースとして再設定するには、**SQL_INTERNAL**または**DEFAULT**キーワードを渡してコマンドを実行します。

使用後、つまりカレントデータベースを変更する際、現在のエクスターナルデータベースを物理的に閉じる場合は**AUTO_CLOSE**キーワードを渡します。エクスターナルデータベースを開く際には時間が消費されるため、最適化の目的で4Dはユーザセッション中に開かれたエクスターナルデータベースの情報をメモリ中に保持します。この情報は4Dアプリケーションが起動されている間保持されます。そのため同じエクスターナルデータベースを再度開く際にはより速く行えます。しかしこの場合、エクスターナルデータベースはそれを最初に開いたアプリケーションにより読み書きモードで開かれたままとなるので、複数の4Dアプリケーション間でエクスターナルデータベースを共有することができなくなります。複数の4Dアプリケーションが同じエクスターナルデータベースを同時に使用できるようにするには、**AUTO_CLOSE**キーワードを渡してエクスターナルデータベースを使用後、物理的に閉じるようにします。

この制限は同じアプリケーション内のプロセスには適用されません。アプリケーション内の異なるプロセスは、閉じる必要なしに、常に読み書きモードで同じエクスターナルデータベースにアクセスできます。
複数のプロセスが同じエクスターナルデータベースを使用する場合、**AUTO_CLOSE**オプションが渡されている場合でも、それを使用する最後のプロセスが閉じられるまで、物理的に解放されない点に留意してください。開発者はエクスターナルデータベースのアプリケーション間での共有や削除に関連する処理を行う際にはこの動作を考慮に入れなければなりません。

## 例題

エクスターナルデータベースを使用後、メインデータベースに戻ります:

```
Begin SQL
   USE DATABASE DATAFILE 'C:/MyDatabase/Names'
   SELECT Name FROM emp INTO :tNames1
   USE DATABASE SQL_INTERNAL
End SQL
```

# ALTER DATABASE

ALTER DATABASE {ENABLE | DISABLE} {INDEXES | CONSTRAINTS}

## 説明

**ALTER DATABASE**コマンドはカレントセッションのカレントデータベースのSQLオプションを有効/無効にします。

このコマンドは、多くのリソースを使用する特定の処理の速度を向上させるために、一時的にSQLのオプションを無効にするために使用されます。例えば大量のデータの読み込みを開始する前にインデックスと制約を無効にすれば、読み込み時間を大きく削減することができます。制約には主キー、外部キー、および重複不可やnull属性が含まれる点に留意してください。

**ALTER DATABASE**はデータベース全体に適用します。つまりあるユーザがオプションを無効にすると、データベースのユーザすべてに対して無効になります。

## 例題

すべてのSQLオプションを無効にして読み込みを行う例:

```
Begin SQL
   ALTER DATABASE DISABLE INDEXES;
   ALTER DATABASE DISABLE CONSTRAINTS;
End SQL
SQL EXECUTE SCRIPT("C:\Exported_data\Export.sql";SQL On error continue)
Begin SQL
   ALTER DATABASE ENABLE INDEXES;
   ALTER DATABASE ENABLE CONSTRAINTS;
End SQL
```

# CREATE TABLE

CREATE TABLE [IF NOT EXISTS] [*sql_name*.]*sql_name*({*column_definition* |*table_constraint*][**PRIMARY KEY**], ... , {*column_definition* |*table_constraint*][**PRIMARY KEY**]) [[**ENABLE** | **DISABLE**] **REPLICATE**]

## 説明

*CREATE TABLE* コマンドは*sql_name*という名称のテーブルを作成するために使用します。フィールドは1つ以上の*column_definition*で指定し、また*table_constraint* 型の引数も指定できます。**IF NOT EXISTS** 制約が指定されると、データベースに同じテーブル名が存在しない場合のみテーブルが作成されます。テーブルがすでに存在する場合、テーブルは作成されず、エラーは生成されません。

1番目の*sql_name* (オプション) 引数は、テーブルを割り当てるスキーマを指定するために使用できます。この引数を渡さないか、指定したスキーマが存在しない場合、テーブルは自動で"DEFAULT_SCHEMA"という名称のデフォルトスキーマに割り当てられます。SQLスキーマに関する詳細は**4Dと4D SQLエンジン統合の原則**を参照してください。

**Note**: 4Dテーブルのインスペクタパレットにある"スキーマ"ポップアップメニューを使用して、テーブルをスキーマに割り当てることもできます。このメニューにはデータベースに定義されたスキーマのリストが含まれています。

*column_definition* はカラムの名称 (*sql_name*) とデータ型 (*sql_data_type_name*) を含み、*table_constraint* はテーブルが格納できる値を制限します。

**PRIMARY KEY**キーワードはテーブルが作成される際に主キーを指定するために使用します。主キーに関する詳細は**4Dと4D SQLエンジン統合の原則**を参照してください。

**ENABLE REPLICATE**と**DISABLE REPLICATE**キーワードはテーブルの複製メカニズムを有効あるいは無効にするために使用します (**SQLを使用した複製**を参照)。

## 例題 1

これは2つのカラムを持つテーブルを作成する単純な例です:

```
CREATE TABLE ACTOR_FANS
(ID INT32, Name VARCHAR);
```

## 例題 2

これは同じ名称のテーブルが存在しない場合のみテーブルを作成します:

```
CREATE TABLE IF NOT EXISTS ACTOR_FANS
(ID INT32, Name VARCHAR);
```

## 例題 3

この例題は"Preferences" テーブルを作成し、"Control" スキーマに割り当てます:

```
CREATE TABLE Control.Preferences
(ID INT32, Value VARCHAR);
```

# ALTER TABLE

```
ALTER TABLE sql_name
{ADD column_definition [PRIMARY KEY]|
DROP sql_name |
ADD primary_key_definition |
DROP PRIMARY KEY |
ADD foreign_key_definition |
DROP CONSTRAINT sql_name |
[{ENABLE | DISABLE} REPLICATE] |
SET SCHEMA sql_name}
```

## 説明

*ALTER TABLE*コマンドは既存のテーブル (*sql_name*) を変更するために使用します。以下のいずれかのアクションを実行できます:
**ADD** *column_definition*を渡すと、テーブルにカラムを追加します。**PRIMARY KEY**はカラムを追加する際に主キーを設定するために使用します。
**DROP** *sql_name*を渡すと、テーブルからカラム*sql_name*を削除します。
**ADD** *primary_key_definition*を渡すと、テーブルに**PRIMARY KEY**を追加します。
**DROP PRIMARY KEY**を渡すと、テーブルの**PRIMARY KEY**を削除します。
**ADD** *foreign_key_definition*を渡すと、テーブルに**FOREIGN KEY**を追加します。
**DROP CONSTRAINT** *sql_name*を渡すと、指定した制約をテーブルから削除します。
**ENABLE REPLICATE**または**DISABLE REPLICATE**はテーブルの複製メカニズムを有効/無効にします (参照)。
**SET SCHEMA** *sql_name*を渡すと、テーブルを*sql_name* スキーマへ転送します。

## 例題

この例題ではテーブルを作成し、データを挿入し、その後Phone_Numberカラムを追加、さらに値を追加して、最後にIDカラムを削除します:

```
CREATE TABLE ACTOR_FANS
(ID INT32, Name VARCHAR);

INSERT INTO ACTOR_FANS
(ID, Name)
VALUES(1, 'Francis');

ALTER TABLE ACTOR_FANS
ADD Phone_Number VARCHAR;

INSERT INTO ACTOR_FANS
(ID, Name, Phone_Number)
VALUES (2, 'Florence', '01446677888');

ALTER TABLE ACTOR_FANS
DROP ID;
```

# DROP TABLE

```
DROP TABLE [IF EXISTS] sql_name
```

## 説明

*DROP TABLE*コマンドはデータベースから*sql_name*という名前のテーブルを削除します。**IF EXISTS**制約が渡されていると、削除するテーブルがデータベースにない場合、コマンドは何も行わず、エラーも生成されません。

このコマンドはテーブルを削除するだけでなく、テーブルに属するデータやインデックス、トリガ、制約なども削除します。**FOREIGN KEY**制約で参照されているテーブルには使用できません。

**Note:** **DROP TABLE**コマンドを実行する際は、書き込みモードでメモリにロードされている*sql_name*テーブルのレコードが無いことを確認しなければなりません。そうでなければエラー1272が生成されます。

## 例題 1

この例題はACTOR_FANSテーブルを削除します:

```
DROP TABLE ACTOR_FANS
```

## 例題 2

この例題は先の例題と同じことを行いますが、ACTOR_FANSテーブルが存在しない場合、エラーは生成されません。:

```
DROP TABLE IF EXISTS ACTOR_FANS
```

# CREATE INDEX

**CREATE** [**UNIQUE**] **INDEX** *sql_name* **ON** *sql_name* (*column_reference*, ... , *column_reference*)

## 説明

*CREATE INDEX*コマンドは既存の (*sql_name*) テーブルの一つ以上の*column_reference*に、名前*sql_name*でインデックスを作成するために使用します。インデックスはユーザに対し透過的で、クエリの速度を上げます。

オプションの**UNIQUE**キーワードを渡して重複値を許可しないインデックスを作成することもできます。

## 例題

この例題はインデックスを作成します:

```
CREATE INDEX ID_INDEX ON ACTOR_FANS (ID)
```

# DROP INDEX

```
DROP INDEX sql_name
```

## 説明

*DROP INDEX*コマンドはデータベースから*sql_name*という名前のインデックスを削除するために使用します。**PRIMARY KEY**や**UNIQUE**制約のために作成されたインデックスには使用できません。

## 例題

この例題はインデックスを削除します:

```
DROP INDEX ID_INDEX
```

# LOCK TABLE

```
LOCK TABLE sql_name IN {EXCLUSIVE | SHARE} MODE
```

## 説明

*LOCK TABLE*コマンドは、*sql_name*テーブルを**EXCLUSIVE**または**SHARE**モードでロックするために使用します。
**EXCLUSIVE**モードでは、テーブルのデータを他のトランザクションで読み込んだり更新したりすることはできません。
**SHARE**モードでは、テーブルのデータを他のトランザクションで読み込むことができます。しかし更新はできません。

## 例題

この例題ではMOVIESテーブルをロックします。他のトランザクションはデータの読み込みはできますが、更新はできません:

```
LOCK TABLE MOVIES IN SHARE MODE
```

# UNLOCK TABLE

```
UNLOCK TABLE sql_name
```

## 説明

*UNLOCK TABLE*コマンドは、*LOCK TABLE*コマンドでロックされたテーブルのロックを解除します。トランザクション中または他のプロセスでロックされたテーブルに対しては動作しません。

## 例題

このコマンドはMOVIESテーブルのロックを解除します:

```
UNLOCK TABLE MOVIES
```

# EXECUTE IMMEDIATE

EXECUTE IMMEDIATE <<*sql_name*>> | <<$*sql_name*>> | :*sql_name* | :$*sql_name*

## 説明

コマンドは動的なSQL文を実行するために使用します。渡す*sql_name*は、実行可能なSQL文を提供するために使用します。

**Notes:**

- このコマンドは**USE EXTERNAL DATABASE** 4Dコマンドで開かれた外部データソースとの接続 (SQLパススルー) には使用できません。
- コンパイル済みモードでは、*EXECUTE IMMEDIATE*コマンドに渡されるクエリ文字列中に、($文字で始まる) 4Dローカル変数は使用できません。

## 例題

この例題では、1960年以降にリリースされた映画の数を取得します:

```
C_LONGINT(NumMovies)
C_TEXT(tQueryTxt)
NumMovies:=0

tQueryTxt:="SELECT COUNT(*) FROM MOVIES WHERE Year_of_Movie >= 1960 INTO :NumMovies;"
Begin SQL
   EXECUTE IMMEDIATE :tQueryTxt;
End SQL

ALERT("The Video Library contains "+String(NumMovies)+" movies more recent or equal to 1960")
```

# CREATE SCHEMA

CREATE SCHEMA *sql_name*

## 説明

*CREATE SCHEMA* コマンドは新しいSQLスキーマを*sql_name*という名称でデータベースに作成するために使用します。

**Note:** スキーマに関する情報は、**SET PROCESS VARIABLE4Dと4D SQLエンジン統合の原則**を参照してください。

新しいスキーマを作成すると、デフォルトで以下のアクセス権が割り当てられます:

- 読み込みのみ (データ): <Everybody>
- 読み書き (データ): <Everybody>
- すべて (データとストラクチャ): <Nobody>

*GRANT* コマンドを使用して、それぞれのスキーマに外部からのアクセス権を与えることができます。

データベースのDesignerとAdministratorのみがスキーマを作成、更新、削除できます。
4Dのアクセス管理システムが有効になっていない場合、(すなわちDesignerにパスワードが設定されていない場合)、制限なしにすべてのユーザがスキーマの作成や更新を行えます。

データベースが4D v11 SQL (release 3 以降) で作成されるか変換されると、デフォルトスキーマが作成されデータベースのすべてのテーブルがこのスキーマに属します。このスキーマは "DEFAULT_SCHEMA" という名称で、削除したり名称を変更したりすることはできません。

## 例題

"Accounting_Rights"というスキーマを作成します:

```
CREATE SCHEMA Accounting_Rights
```

# ALTER SCHEMA

```
ALTER SCHEMA sql_name RENAME TO sql_name
```

## 説明

*ALTER SCHEMA* コマンドは一番目の引数の*sql_name* SQLスキーマを二番目の*sql_name* に名称変更するために使用します。

データベースのDesignerとAdministratorのみがスキーマを変更できます。

## 例題

MyFirstSchemaスキーマをMyLastSchemaに名称変更します:

```
ALTER SCHEMA MyFirstSchema RENAME TO MyLastSchema
```

# DROP SCHEMA

```
DROP SCHEMA sql_name
```

## 説明

*DROP SCHEMA* コマンドは*sql_name*で指定したスキーマを削除するために使用します。

デフォルトスキーマ (DEFAULT_SCHEMA) を除き、すべてのスキーマを削除可能です。スキーマを削除すると、割り当てられていたすべてのテーブルはデフォルトスキーマに移動されます。移動されたテーブルはデフォルトスキーマのアクセス権を継承します。

存在しないか既に削除されたスキーマを削除しようとすると、エラーが生成されます。

データベースのDesignerとAdministratorのみがスキーマを削除できます。

## 例題

(Table1とTable2が属する) MyFirstSchema スキーマを削除します:

```
DROP SCHEMA MyFirstSchema
```

この処理の後、2つのテーブルはデフォルトスキーマに再割り当てされます。

# GRANT

GRANT[READ | READ_WRITE | ALL] ON *sql_name* TO *sql_name*

## 説明

*GRANT* コマンドは*sql_name* スキーマに割り当てるアクセス権を設定するために使用します。これらのアクセス権は、二番目の*sql_name* 引数で指定する4Dユーザグループに割り当てられます。

**READ**、**READ_WRITE**そして**ALL** キーワードを使用して、テーブルに許可されるアクセス権を指定します:

- **READ** は読み込みのみアクセスを設定します (データ)
- **READ_WRITE** は読み書きアクセスを設定します (データ)
- **ALL** はフルアクセスモードを設定します (データとストラクチャ)

アクセスコントロールは外部接続にのみ適用されます。**Begin SQL**/**End SQL**タグや**SQL EXECUTE**などのコマンドで4D内で実行されるSQLコードは依然としてフルアクセスを持っています。

**互換性に関する注意**: 以前のバージョンのデータベースをバージョン11.3以降に変換する際、(アプリケーション環境設定のSQLページで設定された) グローバルアクセス権がデフォルトスキーマに転送されます。

二番目の *sql_name* 引数には、スキーマにアクセス権を割り当てたい4Dユーザグループ名を指定しなければなりません。このグループは4Dデータベースに存在していなければな りません。

**Note:** 4Dはグループ名にスペースやアクセント文字を許可しますが、これはSQL標準で許可されていません。この場合、グループ名を [ と ] の間に記述します。例えば: **GRANT READ ON [my schema] TO [the admins!]**

データベースのDesignerとAdministratorのみがスキーマを変更できます。

## 参照整合性に関する注意

4D はアクセス権とは独立して、参照整合性に関する原則を適用します。例えば2つのテーブルTable1とTable2があり、N対1のリレーション (Table2 -> Table1) で接続されているとします。 Table1はスキーマS1に属し、Table2はスキーマS2に属します。スキーマS1へのアクセス権を持つが、S2へのアクセス権を持たないユーザは、Table1のレコードを削除できます。この場合、参照整合性の原則を適用するため、Table1の削除されたレコードにリレートするTable2のレコードも削除されます。

## 例題

"Power_Users" グループに、MySchema1のデータへの読み書きアクセスを許可します:

```
GRANT READ_WRITE ON MySchema1 TO POWER_USERS
```

# REVOKE

**REVOKE** [**READ** | **READ_WRITE** | **ALL**] **ON** *sql_name*

## 説明

*REVOKE* コマンドは*sql_name* 引数で指定するスキーマから、指定するアクセス権を取り除くために使用します。
実際このコマンドを使用すると、指定したアクセス権に <Nobody> 疑似ユーザグループを割り当てます。

## 例題

MySchema1 スキーマからすべての読み書きアクセス権を削除します:

```
REVOKE READ_WRITE ON MySchema1
```

# REPLICATE

```
REPLICATE replicated_list
FROM table_reference
[WHERE search_condition]
[LIMIT {int_number | 4d_language_reference}]
[OFFSET {int_number | 4d_language_reference}]
FOR REMOTE [STAMP] {int_number | 4d_language_reference}
[, LOCAL [STAMP] {int_number | 4d_language_reference}]
{[REMOTE OVER LOCAL | LOCAL OVER REMOTE]}
{LATEST REMOTE [STAMP] 4d_language_reference
[, LATEST LOCAL [STAMP] 4d_language_reference]}
INTO {target_list | table_reference(sql_name_1,...,sql_name_N)};
```

## 説明

**REPLICATE**コマンドを使用して、データベースAのテーブルのデータをデータベースBのテーブルに複製できます。用語としてコマンドが実行されるデータベースを"ローカルデータベース"、データ複製元のデータベースを"リモートデータベース"呼びます。

このコマンドはデータベースの複製システムのフレームワークでのみ使用することができます。システムが動作するためには、複製がローカルデータベースとリモートデータベースで有効にされ、関連するそれぞれのテーブルが主キーを持たなくてはなりません。このシステムに関する詳細は**SQLを使用した複製** を参照してください。

**Note:** 完全な同期システムを実装したい場合は**SYNCHRONIZE**コマンドを参照してください。

*replicated_list*にはコンマで区切った (仮想あるいは標準の) フィールドリストを渡します。フィールドはリモートデータベースの*table_reference*テーブルに属していなければなりません。
**FROM**句には、*replicated_list*フィールドのデータを複製するリモートデータベースのテーブルを指定する、*table_reference*タイプの引数が続かなくてはなりません。

**Note:**リモートデータベースの仮想フィールドはローカルデータベースの配列にのみ格納できます。

### リモートデータベース側

オプションの**WHERE**句を使用して、リモートデータベースのテーブルのレコードに予備的なフィルタを適用することができます。これにより*search_condition*に合致するレコードのみがこのコマンドの処理対象となります。

そして4Dは**FOR REMOTE STAMP**句で指定されたすべてのレコードの*replicated_list*フィールドの値を取り出します。この句に渡すことにできる値は以下のいずれかです:

- **0より大きい倍長整数値**: この場合、__ROW_STAMP の値がこの値以上のレコードのみが複製されます。
- **0**: この場合、__ROW_STAMPの値が0でないすべてのレコードが複製されます。複製を有効にする前に既に存在したレコードは複製の対象とならない点に留意してください (これらのレコードの__ROW_STAMの値は0のためです)。
- **-1**: この場合、リモートテーブルのすべてのレコード、言い換えると__ROW_STAMPの値が0以上のすべてのレコードが複製されます。 前のケースと異なり、複製が有効になる前に既に存在したレコードも含め、すべてのレコードが対象となります。
- **-2**: この場合、(複製が有効になった後に) リモートテーブルから削除された、言い換えると__ROW_ACTIONの値が2のすべてのレコードが複製されます。

最後にオプションの**OFFSET** かつ/または**LIMIT**句を取得したセレクションに適用できます:

- **OFFSET**句が渡されると、セレクションの最初のXレコードが無視されます (Xは句に渡された値)。
- **LIMIT**句が渡されると、この値は最初のYレコードにセレクションを制限するために使用されます (Yは句に渡された値)。**OFFSET**句も渡されている場合、**LIMIT**句は**OFFSET**実行後に適用されます。

両方の句が適用されると、結果のセレクションがローカルデータベースに送信されます。

### ローカルデータベース側

取り出した値は直接ローカルデータベースの*target_list*または*table_reference*テーブルの*sql_name*で指定した標準フィールドに書き込まれます。*target_list*引数は標準のフィールドまたはリモートフィールドと同じ型の配列リストを含むことができます (両方を組み合わせることはできません)。データの格納先がフィールドリストの場合、仮想__ROW_ACTIONフィールドに格納されたアクションに基づき、ターゲットのレコードが自動で作成、更新、または削除されます。

ターゲットデータベース中に既に存在する複製されたレコードのコンフリクト (同じ主キー値) は優先度を指定する句 (**REMOTE OVER LOCAL** または**LOCAL OVER REMOTE** オプション)を使用して解決します:

- **REMOTE OVER LOCAL** オプションを渡すか優先度を指定する句を省略すると、**FOR REMOTE STAMP** で指定されるすべてのソースレコード (リモートデータベース) がターゲットレコード (ローカルデータベース) を上書きします (既に存在していれば -- いずれかの側で更新されているかどうかに関わらず)。この場合 **LOCAL STAMP** 句を渡すことに意味はなく、無視されます。
- **LOCAL OVER REMOTE** オプションを渡すとコマンドは **LOCAL STAMP** を考慮に入れます。この場合スタンプの値が **LOCAL STAMP** 以下のターゲットレコード (ローカルデータベース) がソースレコード (リモートデータベース) で置き換えられることはありません。例えば **LOCAL STAMP** に100を渡すと、スタンプ値が<=100であるローカルデータベースのすべてのレコードは、リモートデータベースの同等のレコードで置き換えられません。これにより、更新されたデータをローカルに保持し、ローカルテーブル中で置換されるレコードセレクションを減らすことができます。
- **LATEST REMOTE STAMP**かつ/または**LATEST LOCAL STAMP**句を渡すと、4Dはリモートおよびローカルテーブルの最新のスタンプの値を、対応する*4d_language_reference*変数に返します。この情報は同期プロシージャの管理を自動化したい場合に有用です。これらの値は複製処理が完了した直後のスタンプ値に対応します。これらを続く**REPLICATE**または**SYNCHRONIZE**ステートメントで使用するとき、それらは**REPLICATE**コマンドから返される前に自動でインクリメントされるため、開発者がインクリメントする必要はありません。

複製が正しく実行されると*OK*システム変数に1が設定されます。4Dメソッドでこの値をチェックできます。

複製処理中にエラーが発生すると、処理は最初に発生したエラーで中断されます。最新のソース変数は (指定されていれば) エラーが発生したレコードのスタンプに設定されます。OKシステム変数は0に設定されます。生成されたエラーは**ON ERR CALL**コマンドでインストールされるエラー処理メソッドでとらえることができます。

**注: REPLICATE**コマンドで実行される処理はデータ整合性制約を考慮に入れません。これは例えば外部キー、重複不可等を管理するルールが検証されないことを意味します。受信したデータがデータ整合性を壊す可能性がある場合、複製処理終了後

にデータを検証する必要があります。もっとも簡単な方法は4DまたはSQLランゲージを使用して更新されるレコードをロックすることです。

# SYNCHRONIZE

```
SYNCHRONIZE
[LOCAL] TABLE table_reference (column_reference_1,...,column_reference_N)
WITH
[REMOTE] TABLE table_reference (column_reference_1,...,column_reference_N)
FOR REMOTE [STAMP] {int_number | 4d_language_reference},
LOCAL [STAMP] {int_number | 4d_language_reference}
{REMOTE OVER LOCAL | LOCAL OVER REMOTE}
LATEST REMOTE [STAMP] 4d_language_reference,
LATEST LOCAL [STAMP] 4d_language_reference;
```

## 説明

**SYNCHRONIZE**コマンドを使用して異なる2つの4D SQLサーバ上に存在する2つのテーブルを同期できます。いずれかのテーブルに対して行われた変更は他方のテーブルに対しても実行されます。コマンドを実行する4D SQLサーバはローカルサーバと呼ばれ、他方のサーバはリモートサーバと呼ばれます。

**SYNCHRONIZE**コマンドは**REPLICATE**コマンドを内部的に2回呼び出したものです。一回目の呼び出しでリモートサーバからのデータをローカルサーバに複製し、二回目の呼び出しでローカルサーバのデータをリモートサーバに複製します。なので同期されるテーブルは複製用に設定されていなければなりません:

- テーブルは主キーを持っていなければなりません。
- "複製を有効にする"オプションが各テーブルのインスペクタウィンドウでチェックされていなければなりません。

詳細は**REPLICATE**コマンドの説明を参照してください。

**SYNCHRONIZE**コマンドは4つのスタンプ2つの入力スタンプと2つの出力スタンプ (最新の更新) を"引数"として受け入れます。入力スタンプはそれぞれのサーバ上での最新の同期の時期を示すために使用されます。この時期はコード化された情報として、継続時間なしで表現されます。出力スタンプは最新の更新直後のそれぞれのサーバ上での更新スタンプの値を返します。この原則により、**SYNCHRONIZE**コマンドが定期的に呼ばれるとき、次回の入力スタンプとして、最新の同期の出力スタンプを使用できます。

**注:** 入力および出力スタンプは数値として表現され、タイムスタンプとはなりません。これらのスタンプについては**REPLICATE**コマンドの説明を参照してください。

エラーが発生すると、関連するサーバの出力スタンプには、エラーのもととなったレコードのスタンプが戻されます。エラーが同期以外の原因で引き起こされた場合 (例えばネットワークの問題など)、スタンプは0となります。
異なる2つのエラーコードがあります。1つはローカルサイトでの同期エラーを示すもので、もう1つはリモートサイトでの同期エラーを示します。
エラーが発生すると、データの状態はローカルサーバのトランザクションの状態に依存します。リモートサーバ上で、同期は常にトランザクション内で実行されるので、処理によりデータが変更されることはありません。しかしローカルサーバ上では、同期処理は開発者の制御下にあります。**自動コミットトランザクション**環境設定が選択されていない場合、トランザクションの外で処理が実行されます (選択されていればトランザクションコンテキストが自動で作成されます)。開発者はトランザクションを開始するか決定でき、データの同期後にこのトランザクションを有効にするかキャンセルするかも開発者に任されています。

**REMOTE OVER LOCAL** と**LOCAL OVER REMOTE**句を使用して、アプリケーションの特性に応じて、同期の方向を強制的に指定できます。実装メカニズムについては**REPLICATE**コマンドの説明を参照してください。

**注: SYNCHRONIZE**コマンドで実行される処理はデータ整合性制約を考慮に入れません。これは例えば外部キー、重複不可等を管理するルールが検証されないことを意味します。受信したデータがデータ整合性を壊す可能性がある場合、複製処理終了後にデータを検証する必要があります。もっとも簡単な方法は4DまたはSQLランゲージを使用して更新されるレコードをロックすることです。

4Dは **LATEST REMOTE STAMP** と **LATEST LOCAL STAMP** 句の *4d_language_ref* 変数にそれぞれ、リモートおよびローカルの最後のスタンプ値を返します。この情報を使用して同期処理を自動化できます。これらの情報は同期処理終了直後のスタンプ値に対応します。そのあとの **REPLICATE** や **SYNCHRONIZE** 文でこれらを使用する場合、値をインクリメントする必要はありません。**REPLICATE** コマンド実行後、自動でインクリメントされています。

## 例題

同期処理に関連するメカニズムを理解するために、同期された2つのデータベース両側で既存のレコードが変更されたケースを考えます。

同期に使用されるメソッドは以下の形式です:

```
C_LONGINT(vRemoteStamp)
C_LONGINT(vLocalStamp)
C_LONGINT(vLatestRemoteStamp)
C_LONGINT(vLatestLocalStamp)

vRemoteStamp:=X... // 後述の表中の値参照
vLocalStamp:=X... // 後述の表中の値参照
vLatestRemoteStamp:=X... // 前回のLATEST REMOTE STAMPに返された値
vLatestLocalStamp:=X... // 前回のLATEST LOCAL STAMPに返された値

Begin SQL
   SYNCHRONIZE
       LOCAL MYTABLE (MyField)
       WITH
       REMOTE MYTABLE (MyField)
       FOR REMOTE STAMP :vRemoteStamp,
       LOCAL STAMP :vLocalStamp
```

```
        LOCAL OVER REMOTE  // または REMOTE OVER LOCAL, 後述の表参照
        LATEST REMOTE STAMP :vLatestRemoteStamp,
        LATEST LOCAL STAMP :vLatestLocalStamp;
End SQL
```

初期のデータは以下の通り:

- LOCALデータベースのレコードスタンプは値が30で、REMOTEデータベースのレコードスタンプ値は4000。
- MyField フィールドの値は以下の通り:

| **LOCAL** | | **REMOTE** | |
|---|---|---|---|
| 古い値 | 新しい値 | 古い値 | 新しい値 |
| AAA | BBB | AAA | CCC |

- 最後の同期以降に変更された値のみを同期するために、前回の **LATEST LOCAL STAMP** と **LATEST REMOTE STAMP** 句に返された値を使用する。

**LOCAL STAMP** と **REMOTE STAMP** に渡された値や使用される優先度オプションに基づき **SYNCHRONIZE** コマンドにより実行される同期は以下のようになります: ROL = **REMOTE OVER LOCAL**、LOR = **LOCAL OVER REMOTE**:

| **LOCAL STAMP** | **REMOTE STAMP** | **優先度** | **同期後のローカルデータ** | **同期後のリモートデータ** | **ローカル - リモート同期の方向** |
|---|---|---|---|---|---|
| 20 | 3000 | ROL | CCC | CCC | <----> |
| 20 | 3000 | LOR | BBB | BBB | <----> |
| 31 | 3000 | ROL | CCC | CCC | <-- |
| 31 | 3000 | LOR | CCC | CCC | <-- |
| 20 | 4001 | ROL | BBB | BBB | --> |
| 20 | 4001 | LOR | BBB | BBB | --> |
| 31 | 4001 | ROL | BBB | CCC | 同期なし |
| 31 | 4001 | LOR | BBB | CCC | 同期なし |
| 40 | 3000 | ROL | CCC | CCC | <-- |
| 40 | 3000 | LOR | CCC | CCC | <-- |
| 20 | 5000 | ROL | BBB | BBB | --> |
| 20 | 5000 | LOR | BBB | BBB | --> |
| 40 | 5000 | ROL | BBB | CCC | 同期なし |
| 40 | 5000 | LOR | BBB | CCC | 同期なし |

# シンタックスルール

## シンタックスルール

この章では、SQL文で使用されるさまざまな述部要素について説明します。4Dにおける使用方法を一般的に示すため、これらは可能な限り個々の項目に分解され、簡潔に説明されます。太字のキーワードは常にそのまま使用されます。

# 4d_function_call

{**FN** *sql_name* ([*arithmetic_expression*, ..., *arithmetic_expression*]) **AS** *sql_data_type_name*}

## 説明

*4d_function_call*を使用して値を返す4D関数を実行することができます。

関数の*sql_name*は**FN**キーワードの後に書かれ、*arithmetic_expression*型の引数が後に続きます。関数から返される値は渡される*sql_data_type_name*により定義された型です。

## 例題

この例題では、MOVIESテーブルからそれぞれの映画ごとの出演者数を取り出します:

```
C_LONGINT($NrOfActors)
ARRAY TEXT(aMovieTitles;0)
ARRAY LONGINT(aNrActors;0)

$NrOfActors:=7
Begin SQL
   SELECT Movie_Title, {FN Find_Nr_Of_Actors(ID) AS NUMERIC}
   FROM MOVIES
   WHERE {FN Find_Nr_Of_Actors(ID) AS NUMERIC} >= :$NrOfActors
   ORDER BY 1
   INTO :aMovieTitles; :aNrActors
End SQL
```

## 4d_language_reference

*<<sql_name>>* |*<<$sql_name>>* | *<<[sql_name]sql_name>>* | *:sql_name*|*:$sql_name*|*:sql_name.sql_name*

### 説明

*4d_language_reference* 引数はデータが代入される4D変数やフィールド名 (*sql_name*) を指定します。この名前は以下のいずれかの様式で渡すことができます:

*<<sql_name>>*
*<<$sql_name>>* (*)
*<<[sql_name]sql_name>>* (標準の4Dシンタックスに対応: [TableName]FieldName)
*:sql_name*
*:$sql_name* (*)
*:sql_name.sql_name* (標準のSQLシンタックスに対応: TableName.FieldName)

(*) コンパイルモードでは ($記号で始まる) ローカル変数への参照を使用することができません。

# all_or_any_predicate

*arithmetic_expression* {< | <= | = | >= | > | <>} {**ANY** | **ALL** | **SOME**} (*subquery*)

## 説明

*all_or_any_predicate*は、*arithmetic_expression*と*subquery*を比較するために使用します。<、<=、=、>=、>、または<>などの比較演算子や、**ANY**、**ALL**、**SOME**などのキーワードを、*subquery*との比較のために渡すことができます。

## 例題

この例題は最も販売されたソフトウェアを選択するサブクエリを実行します。メインのクエリはSALESとCUSTOMERSテーブルから、Total_valueカラムがサブクエリで選択されたレコードよりも大きいレコードを選択します:

```
SELECT Total_value, CUSTOMERS.Customer
FROM SALES, CUSTOMERS
WHERE SALES.Customer_ID = CUSTOMERS.Customer_ID
AND Total_value > ALL   (SELECT MAX (Total_value)
FROM SALES
WHERE Product_type = 'Software');
```

## arithmetic_expression

*literal* |
*column_reference* |
*function_call* |
*command_parameter* |
*case_expression* |
(*arithmetic_expression*) |
+ *arithmetic_expression* |
− *arithmetic_expression* |
*arithmetic_expression* + *arithmetic_expression* |
*arithmetic_expression* − *arithmetic_expression* |
*arithmetic_expression* * *arithmetic_expression* |
*arithmetic_expression* / *arithmetic_expression* |

### 説明

*arithmetic_expression*には*literal*値、*column_reference*、*function_call*、*command_parameter*、*case_expression*などが含まれます。+、-、*、または/演算子を使用して、*arithmetic_expression*の組み合わせを渡すこともできます。

# between_predicate

*arithmetic_expression* [**NOT**] **BETWEEN** *arithmetic_expression* **AND** *arithmetic_expression*

## 説明

*between_predicate*は、(昇順で渡された) 他の二つの*arithmetic_expression*の間にある値を持つデータを検索するために使用します。オプションの**NOT**キーワードを渡して、この制限の値を取り除くために使用することもできます。

## 例題

この例題では、名前がAからEの間の文字で始まるすべての顧客の名前を返します:

```
SELECT CLIENT_FIRSTNAME, CLIENT_SECONDNAME
FROM T_CLIENT
WHERE CLIENT_FIRSTNAME BETWEEN 'A' AND 'E'
```

# case_expression

*case_expression*

## 説明

*case_expression*は式を選択する際、一つ以上の条件を適用するために使用します。
以下のように使用します:

```
CASE
WHEN search_condition THEN arithmetic_expression
...
WHEN search_condition THEN arithmetic_expression
[ELSE arithmetic_expression]
END
```

または

```
CASE arithmetic_expression
WHEN arithmetic_expression THEN arithmetic_expression
...
WHEN arithmetic_expression THEN arithmetic_expression
[ELSE arithmetic_expression]
END
```

## 例題

この例題はROOM_FLOORカラムの値に基づき、ROOM_NUMBERカラムからレコードを選択します:

```
SELECT ROOM_NUMBER
CASE ROOM_FLOOR
WHEN 'Ground floor' THEN 0
WHEN 'First floor' THEN 1
WHEN 'Second floor' THEN 2
END AS FLOORS, SLEEPING_ROOM
FROM T_ROOMS
ORDER BY FLOORS, SLEEPING_ROOM
```

# column_definition

*sql_name sql_data_type_name [(int_number)*][**NOT NULL** [**UNIQUE**]] [**AUTO_INCREMENT**] [**AUTO_GENERATE**]

## 説明

*column_definition*にはカラムの名前 (*sql_name*) とデータタイプ (*sql_data_type_name*) が含まれます。オプションの*int_number*や**NOT NULL**、**UNIQUE**、**AUTO_INCREMENT**、あるいは**AUTO_GENERATE**キーワードを渡すこともできます。

- **NOT NULL**を*column_definition*に渡すと、カラムはNULL値を受け付けないことを意味します。
- **UNIQUE**を渡すと、同じ値がこのカラムに二回挿入されないことを意味します。**NOT NULL**カラムだけが**UNIQUE**属性を持つことができる点に留意してください。**UNIQUE**キーワードの前には常に**NOT NULL**がなければなりません。
- **AUTO_INCREMENT**を渡すと、そのカラムは新しくローが加えられるたびにユニーク番号を生成します。この属性は数値カラムにのみ使用できます。
- **AUTO_GENERATE**を渡すと、各新規行ごとそのカラムに自動でUUIDが生成されます。この属性はUUIDカラムにのみ使用できます。

それぞれのカラムはデータ型を持っていなければなりません。カラムは"null"または"not null"として定義され、もしこの値が空のままの場合、データベースは"null"をデフォルトとみなします。カラムのデータ型は、カラムにどのようなデータが置かれるかを限定しません。

## 例題

これは二つのカラム (ID と Name) をもつテーブルを作成する例です:

```
CREATE TABLE ACTOR_FANS
(ID INT32, Name VARCHAR NOT NULL UNIQUE);
```

# column_reference

*sql_name* | *sql_name.sql_name* | *sql_string.sql_string*

## 説明

*column_reference*は、以下のいずれかの形式で渡される*sql_name*または*sql_string*から構成されます:

*sql_name*
*sql_name.sql_name*
*sql_string.sql_string*

# command_parameter

? | <<*sql_name*>> | <<$*sql_name*>> | <<[*sql_name*]*sql_name*>> | :*sql_name* | :$*sql_name* | :*sql_name*.*sql_name*

## 説明

*command_parameter*は、疑問符 (?) または以下のいずれかの形式で渡される*sql_name*から構成されます:

*?*
*<<sql_name>>*
*<<$sql_name>>*
*<<[sql_name]sql_name>>*
*:sql_name*
*:$sql_name*
*:sql_name.sql_name*

# comparison_predicate

*arithmetic_expression* {< |<= |= | >= | > | <> } *arithmetic_expression* |
*arithmetic_expression* {< |<= |= | >= | > | <> } (*subquery*) |
(*subquery*) {< |<= |= | >= | > | <> } *arithmetic_expression*

## 説明

*comparison_predicate*はデータに適用する*search_condition*の一部として、<、<=、=、>=、>、または <>のような演算子を使用して、ふたつの*arithmetic_expression*を比較、または*arithmetic_expression*と*subquery*を比較するために使用します。

# exists_predicate

**EXISTS** (*subquery*)

## 説明

*exists_predicate*は*subquery*を示して、返されるものをチェックします。これは**EXISTS**キーワードの後に*subquery*を記述することで行います。

## 例題

この例題は指定した地域に店舗が存在する場合に、売上総額を返します:

```
SELECT SUM (Sales)
FROM Store_Information
WHERE EXISTS
(SELECT * FROM Geography
WHERE region_name = 'West')
```

## foreign_key_definition

```
CONSTRAINT sql_name
FOREIGN KEY (column_reference, ... , column_reference)
REFERENCES sql_name [(column_reference, ... , column_reference)]
[ON DELETE {RESTRICT |CASCADE}]
[ON UPDATE {RESTRICT |CASCADE}]
```

### 説明

*foreign_key_definition*は、データ整合性を確実にする目的で、他のテーブルに設定された主キーフィールド (*column_reference*) に一致させるために使用します。**FOREIGN KEY**制約は、(他のテーブルの主キーに一致する) 外部キーとして定義されるカラム参照 (*column_reference*)を渡すために使用されます。

**CONSTRAINT** *sql_name* 句は、**FOREIGN KEY**制約に名前をつけるために使用します。

続く**REFERENCES**句は、一致する他のテーブル (*sql_name*) の主キーフィールドソースを指定します。**REFERENCES**句で指定されるテーブル (*sql_name*) が、外部キー制約に一致するキーとして使用される主キーを持つ場合、*column_reference*リストを省略できます。

オプションの**ON DELETE CASCADE**句は、(主キーフィールドを含む) 親テーブルからローが削除された時、(外部キーフィールドを含む) 子テーブルの関連するローも削除することを指定します。オプションの**ON DELETE RESTRICT**句を渡すと、他のテーブルが参照するデータが削除されることを防ぎます。

オプションの**ON UPDATE CASCADE**句は、(主キーフィールドを含む) 親テーブルでローが更新されると、(外部キーフィールドを含む) 子テーブルの関連するローも更新されることを指定します。オプションの**ON UPDATE RESTRICT**句を渡すと、他のテーブルが参照するデータが更新されることを防ぎます。

**ON DELETE**と**ON UPDATE**両方の句が渡されるとき、両方は同じタイプでなければなりません (例. **ON DELETE CASCADE**と**ON UPDATE CASCADE**、または**ON DELETE RESTRICT**と**ON UPDATE RESTRICT**)。

**ON DELETE**と**ON UPDATE**句どちらも渡されないとき、**CASCADE**がデフォルトルールとして使用されます。

### 例題

この例題はORDERSテーブルを作成し、Customer_SIDカラムを、CUSTOMERS テーブルのSIDカラムが割り当てられる外部キーとして設定します:

```
CREATE TABLE ORDERS
(Order_ID INT32,
Customer_SID INT32,
Amount NUMERIC,
PRIMARY KEY (Order_ID),
FOREIGN KEY (Customer_SID) REFERENCES CUSTOMER(SID));
```

# function_call

*sql_function_call |*
*4d_function_call*

## 説明

*function_call*は**SQL関数**または4D関数 (*4d_function_call*) からなります。両タイプの関数はデータを操作し、結果を返します。引数を渡すこともできます。

## 例題

この例題はSQLの*COUNT*関数を使用します:

```
C_LONGINT(vPersonNumber)
Begin SQL
   SELECT COUNT (*)
   FROM SALES_PERSONS
   INTO :vPersonNumber;
End SQL
```

# in_predicate

*arithmetic_expression* [**NOT**] **IN** (*subquery*) |

*arithmetic_expression* [**NOT**] **IN** (*arithmetic_expression*, ..., *arithmetic_expression*)

## 説明

*in_predicate*は、*arithmetic_expression*を比較して、値のリストにそれが含まれるか (または**NOT**キーワードが渡されていれば含まれないか) をチェックするために使用します。比較に使用する値のリストは一連の*arithmetic_expression*、または*subquery*の結果です。

## 例題

この例題はorder_idカラムの値が10000, 10001, 10003 または 10005であるORDERSテーブルのレコードを選択します:

```
SELECT *
FROM ORDERS
WHERE order_id IN (10000, 10001, 10003, 10005);
```

# is_null_predicate

*arithmetic_expression* **IS** [**NOT**] **NULL**

## 説明

*is_null_predicate*は、*arithmetic_expression*が**NULL**のものを探すために使用します。**NOT**キーワードを渡して**NULL**でないものを探すためにも使用できます。

## 例題

この例題は重さが１５未満またはColorカラムにNULLが含まれる製品を選択します:

```
SELECT Name, Weight, Color
FROM PRODUCTS
WHERE Weight < 15.00 OR Color IS NULL
```

# like_predicate

*arithmetic_expression* [**NOT**] **LIKE** *arithmetic_expression* [**ESCAPE** *sql_string*]

## 説明

*like_predicate*は、**LIKE**キーワードの後ろに渡した*arithmetic_expression*に一致するデータを取得するために使用します。**NOT**キーワードを渡して、この式と異なるデータを検索することもできます。**ESCAPE**キーワードを使用して、*sql_string*に渡された文字がワイルドカードとして解釈されることを避けることができます。これは通常'%'や'_'文字を検索する場合に使用します。

## 例題 1

この例題は名前に"bob"を含むsuppliersを検索します:

```
SELECT * FROM suppliers
WHERE supplier_name LIKE '%bob%';
```

## 例題 2

suppliersの名前が文字Tで始まらないものを検索する:

```
SELECT * FROM suppliers
WHERE supplier_name NOT LIKE 'T%';
```

## 例題 3

suppliersの名前が"Sm"で始まり"th"で終わるものを探す:

```
SELECT * FROM suppliers
WHERE supplier_name LIKE 'Sm_th'
```

# literal

*int_number* | *fractional_number* | *sql_string* | *hexadecimal_number*

## 説明

*literal*は、*int_number* (数値)、*fractional_number* (分数)、*sql_string*、または*hexadecimal_number*で構成されるデータタイプです。

16進表記 (4D 12.1よりサポート) はバイトとして表現されるどのような型のデータも表現できます。1バイトは常に2つの16進値で指定されます。SQLコマンドにこの表記法を使用していることを示すために、16進を表現するSQLの標準シンタックスを使用します:

X'<16進値>'

例えば10進数の15の場合、**X'0f'** と書きます。空の値 (ゼロバイト) は **X''** と書きます。

注: **SQL EXPORT DATABASE**と**SQL EXPORT SELECTION**コマンドはバイナリーデータがメインの書き出しファイルに含まれる場合、これらのデータを16進フォーマットで書き出します。

# predicate

*predicate*

## 説明

*predicate*は**WHERE**句に続き、データの検索条件として適用するために使用します。以下のいずれかのタイプです:

*comparison_predicate*
*between_predicate*
*like_predicate*
*is_null_predicate*
*in_predicate*
*all_or_any_predicate*
*exists_predicate*

## primary_key_definition

[**CONSTRAINT** *sql_name*] **PRIMARY KEY** (*sql_name*, ... , *sql_name*)

### 説明

*primary_key_definition*は、テーブルの**PRIMARY KEY** (ユニークID)として提供するカラムあるいはカラムの組み合わせの*sql_name*を渡すために使用します。渡されるカラムは重複値や**NULL**値を含んでいてはいけません。

オプションの**CONSTRAINT**を**PRIMARY KEY**の前に置き、カラムに挿入できる値を制限できます。

### 例題

この例題はテーブルを作成し、SIDカラムを主キーに設定します:

```
CREATE TABLE Customer
(SID int32,
Last_Name varchar(30),
First_Name varchar(30),
PRIMARY KEY (SID));
```

# search_condition

*predicate* |
**NOT** *search_condition* |
(*search_condition*) |
*search_condition* **OR** *search_condition* |
*search_condition* **AND** *search_condition* |

## 説明

*search_condition*は、取得するデータに適用する条件を指定します。**AND**や**OR**キーワードを使用した*search_condition*の組み合わせも適用できます。**NOT**キーワードを*search_condition*の前において、指定した条件に一致しないデータを取得することもできます。

*predicate*を*search_condition*として渡すこともできます。

## 例題

これは**WHERE**句に複合検索条件を使用する例です:

```
SELECT supplier_id
FROM suppliers
WHERE (name = 'CANON')
OR (name = 'Hewlett Packard' AND city = 'New York')
OR (name = 'Firewall' AND status = 'Closed' and city = 'Chicago');
```

# select_item

*arithmetic_expression* [[**AS**] [*sql_string*|*sql_name*]]

## 説明

*select_item*は結果に含める一つ以上の項目を指定します。渡された*select_item*ごとにカラムが生成されます。それぞれの*select_item*は*arithmetic_expression*からなります。オプションの**AS**キーワードを渡して、カラムに渡されるオプションの*sql_string*や*sql_name*を指定することもできます。(**AS**キーワードなしでオプションの*sql_string*や*sql_name*を渡しても同じ効果です。)

## 例題

これは、2000年以降に公開された映画を含むMovie_Yearカラムを作成する例です:

```
ARRAY INTEGER(aMovieYear;0)
Begin SQL
   SELECT Year_of_Movie AS Movie_Year
   FROM MOVIES
   WHERE Movie_Year >= 2000
   ORDER BY 1
   INTO :aMovieYear;
End SQL
```

## sort_list

{*column_reference* | *int_number*} [**ASC** | **DESC**], ... , {*column_reference* | *int_number*} [**ASC** |**DESC**]

### 説明

*sort_list*は*column_reference*や*int_number*で構成され、並び替えを適用するカラムを指定します。**ASC**または**DESC**キーワードを渡して並び替えを昇順で行うか降順で行うかを指定できます。デフォルトで昇順の並び替えが行われます。

## sql_data_type_name

```
ALPHA_NUMERIC |VARCHAR | TEXT | TIMESTAMP |INTERVAL |DURATION |BOOLEAN |BIT | BYTE |INT16 |SMALLINT |
INT32 | INT |INT64 |NUMERIC | REAL | FLOAT | DOUBLE PRECISION | BLOB | BIT VARYING | CLOB | PICTURE
```

### 説明

*sql_data_type_name*は、*4d_function_call*中で**AS**キーワードの後に書かれます。以下のいずれかの値です:

**ALPHA_NUMERIC**
**VARCHAR**
**TEXT**
**TIMESTAMP**
**INTERVAL**
**DURATION**
**BOOLEAN**
**BIT**
**BYTE**
**INT16**
**SMALLINT**
**INT32**
**INT**
**INT64**
**NUMERIC**
**REAL**
**FLOAT**
**DOUBLE PRECISION**
**BLOB**
**BIT VARYING**
**CLOB**
**PICTURE**

# sql_name

*sql_name*

## 説明

*sql_name*は、Latinアルファベット文字で始まり、続いてLatin文字、数字、アンダースコアで構成される標準的なSQL名か、角括弧で挟まれた文字列です。右角括弧は二つ重ねることでエスケープされます。

例題:

| 渡す文字列 | sql_name |
| --- | --- |
| MySQLName_2 | MySQLName_2 |
| My non-standard !&^#%!&#% name | [My non-standard !&^#%!&#% name] |
| [already-bracketed name] | [[already-bracketed name]]] |
| name with brackets[] inside | [name with brackets []] inside] |

# sql_string

*sql_string*

## 説明

*sql_string*はシングルクォートで囲まれた文字列です。文字列中のシングルクォートは二つ重ねてエスケープします。
例題:

| 渡す文字列 | sql_string |
| --- | --- |
| my string | 'my string' |
| string with ' inside it | 'string with ' ' inside it' |
| 'string already in quotes' | ' ' 'string already in quotes' ' ' |

# subquery

```
SELECT [ALL | DISTINCT]
{* | select_item, ..., select_item}
FROM table_reference, ..., table_reference
[WHERE search_condition]
[GROUP BY sort_list]
[HAVING search_condition]
[LIMIT {int_number | ALL}]
[OFFSET int_number]
```

## 説明

*subquery*は括弧に挟まれた*SELECT*文のようなもので、他のSQL文 (*SELECT*、**INSERT**、*UPDATE*、*DELETE*) に渡されます。これはクエリ内のクエリとして動作し、しばしば**WHERE**または**HAVING**句の一部として渡されます。

## table_constraint

{*primary_key_definition* | *foreign_key_definition*}

### 説明

*table_constraint*はテーブルに格納することのできる値を制限します。*primary_key_definition*または*foreign_key_definition*を渡すことができます。*primary_key_definition*はテーブルの主キーを設定し、*foreign_key_definition*は (他のテーブルの主キーに一致する) 外部キーを設定するために使用します。

### 例題

この例題はNameを主キーフィールドに設定します:

```
CREATE TABLE ACTOR_FANS
(ID INT32, Name VARCHAR NOT NULL UNIQUE);
```

## table_reference

[*sql_name* | *sql_string*] [[**AS**] [*sql_name*|*sql_string*]]

### 説明

*table_reference*は標準のSQL名または文字列です。オプションの**AS**キーワードを渡してカラムに (*sql_name*または*sql_string*の形式で) エイリアスを割り当てることもできます。(**AS**キーワードなしでオプションの*sql_string*や*sql_name*を渡しても同じです。)

# トランザクション

トランザクション
START
COMMIT
ROLLBACK

## トランザクション

### 説明

トランザクションはグループで実行されるSQL文のセットです。すべてが正しく実行されるか、まったく効果を及ぼさないかです。トランザクションは実行時にデータの整合性を保つためロックを使用します。トランザクションが正しく終了したら、*COMMIT*文を使用して更新を永続化します。そうでなければ、*ROLLBACK*文を使用して更新をキャンセルし、データベースを以前の状態に戻します。

4Dのトランザクションと SQLのトランザクションの間に違いはありません。両タイプのトランザクションは同じデータおよびプロセスを共有します。ローカルデータベースに適用される**Begin SQL**/**End SQL**の間のSQL文、**QUERY BY SQL**、そして統合された汎用SQLコマンドは、常に標準のSQLコマンドと同じコンテキストで実行されます。

**Note:** 4Dは"自動コミット"オプションを提供していて、データ整合性を確保するために、SIUDコマンド (*SELECT*、**INSERT**、*UPDATE*、*DELETE*) 利用時に自動でトランザクションを開始し、受け入れることができます。詳細情報は**SET PROCESS VARIABLE4Dと4D SQLエンジン統合の原則**を参照してください。

以下の例題は、複数の異なるトランザクション使用した場合の動作を説明します:

empテーブルに“John”も“Smith”も追加されません:

```
SQL LOGIN(SQL_INTERNAL;"";"") // 4D SQLエンジンを初期化
START TRANSACTION // カレントプロセスでトランザクションを開始
Begin SQL
   INSERT INTO emp
   (NAME)
   VALUES ('John');
End SQL
SQL EXECUTE("START") // カレントプロセスで他のトランザクションを開始
SQL CANCEL LOAD
SQL EXECUTE("INSERT INTO emp (NAME) VALUES ('Smith')") // この文は同じプロセスで実行される
SQL CANCEL LOAD
SQL EXECUTE("ROLLBACK") // プロセスの内側のトランザクションをキャンセル
CANCEL TRANSACTION // プロセスの外側のトランザクションをキャンセル
SQL LOGOUT
```

empテーブルに“John”のみが追加されます:

```
SQL LOGIN(SQL_INTERNAL;"";"")
START TRANSACTION
Begin SQL
   INSERT INTO emp
   (NAME)
   VALUES ('John');
End SQL
SQL EXECUTE("START")
SQL CANCEL LOAD
SQL EXECUTE("INSERT INTO emp (NAME) VALUES ('Smith')")
SQL CANCEL LOAD
SQL EXECUTE("ROLLBACK") // プロセスの内側のトランザクションをキャンセル
VALIDATE TRANSACTION // プロセスの外側のトランザクションを有効にする
SQL LOGOUT
```

empテーブルに“John”も“Smith”も追加されません。外側のトランザクションが内側のトランザクションもキャンセルします:

```
SQL LOGIN(SQL_INTERNAL;"";"")
START TRANSACTION
Begin SQL
   INSERT INTO emp
   (NAME)
   VALUES ('John');
End SQL
SQL EXECUTE("START")
SQL CANCEL LOAD
SQL EXECUTE("INSERT INTO emp (NAME) VALUES ('Smith')")
SQL CANCEL LOAD
SQL EXECUTE("COMMIT") // プロセスの内側のトランザクションを有効にする
CANCEL TRANSACTION // プロセスの外側のトランザクションをキャンセル
SQL LOGOUT
```

empテーブルに“John”と“Smith”が追加されます:

```
SQL LOGIN(SQL_INTERNAL;"";"")
START TRANSACTION
Begin SQL
   INSERT INTO emp
   (NAME)
   VALUES ('John');
```

```
End SQL
SQL EXECUTE("START")
SQL CANCEL LOAD
SQL EXECUTE("INSERT INTO emp (NAME) VALUES ('Smith')")
SQL CANCEL LOAD
SQL EXECUTE("COMMIT") // プロセスの内側のトランザクションを有効にする
VALIDATE TRANSACTION // プロセスの外側のトランザクションを有効にする
SQL LOGOUT
```

# START

START [TRANSACTION]

## 説明

*START*コマンドは、トランザクションを開始するために使用します。トランザクションがすでに開始されているときにこのコマンドが渡されると、サブトランザクションを開始します。**TRANSACTION**キーワードはオプションです。

## 例題

この例題ではトランザクション内でSELECTを実行します:

```
START TRANSACTION
SELECT * FROM suppliers
WHERE supplier_name like '%bob%';
COMMIT TRANSACTION;
```

# COMMIT

COMMIT [TRANSACTION]

## 説明

*COMMIT*コマンドはトランザクションを有効化して終了します。つまりトランザクション中に行われた更新は永続化され、データベースの一部となります。またトランザクションで使用されていたリソースを解放します。*COMMIT*コマンドの後に*ROLLBACK*文を使用できないことに注意してください。変更はすでに永続化されています。**TRANSACTION**キーワードはオプションです。

## 例題

*START*コマンドの例題を参照してください。

# ROLLBACK

ROLLBACK [TRANSACTION]

## 説明

*ROLLBACK*コマンドは実行中のトランザクションをキャンセルし、トランザクションが開始された時の状態にデータを戻します。またトランザクションで使用されていたリソースを解放します。**TRANSACTION**キーワードはオプションです。

## 例題

この例題例題はrollbackの使用方法を示します:

```
START TRANSACTION
SELECT * FROM suppliers
WHERE supplier_name like '%bob%';
ROLLBACK TRANSACTION;
```

# 関数

SQL関数
ABS
ACOS
ASCII
ASIN
ATAN
ATAN2
AVG
BIT_LENGTH
CAST
CEILING
CHAR
CHAR_LENGTH
COALESCE
CONCAT
CONCATENATE
COS
COT
COUNT
CURDATE
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURTIME
DATABASE_PATH
DATE_TO_CHAR
DAY
DAYNAME
DAYOFMONTH
DAYOFWEEK
DAYOFYEAR
DEGREES
EXP
EXTRACT
FLOOR
HOUR
INSERT
LEFT
LENGTH
LOCATE
LOG
LOG10
LOWER
LTRIM
MAX
MILLISECOND
MIN
MINUTE
MOD
MONTH
MONTHNAME
NULLIF
OCTET_LENGTH
PI
POSITION
POWER
QUARTER
RADIANS
RAND
REPEAT
REPLACE
RIGHT
ROUND
RTRIM
SECOND
SIGN
SIN
SPACE
SQRT
SUBSTRING
SUM
TAN
TRANSLATE
TRIM
TRUNC
TRUNCATE

UPPER
WEEK
YEAR

## SQL関数

---

関数はカラムのデータに対し動作し、特定の結果を返します。このマニュアルで関数名は太字で表示され、そのまま使用されます。必要な引数を含む*arithmetic_expression*が後に続きます。

# ABS

**ABS**（*arithmetic_expression*）

## 説明

関数は*arithmetic_expression*の絶対値を返します。

## 例題

この例題は料金の絶対値に、与えられた数量をかけます:

```
ABS(Price) * quantity
```

# ACOS

ACOS（*arithmetic_expression*）

## 説明

*ACOS*関数は*arithmetic_expression*のアークコサインを返します。これは*COS*関数の逆関数です。*arithmetic_expression*は角度のラジアン表現です。

## 例題

この例題はANGLE_IN_RADIANラジアンのアークコサインを返します:

```
SELECT ACOS(ANGLE_IN_RADIAN)
FROM TABLES_OF_ANGLES;
```

# ASCII

**ASCII** (*arithmetic_expression*)

## 説明

*ASCII*関数は*arithmetic_expression*の一番左の文字を数値として返します。*arithmetic_expression*が空の場合、関数は**NULL**値を返します。

## 例題

この例題はそれぞれの名前の最初の文字を数値として返します:

```
SELECT ASCII(Substring(LastName,1,1))
FROM PEOPLE;
```

# ASIN

**ASIN** (*arithmetic_expression*)

## 説明

*ASIN*は*arithmetic_expression*のアークサインを返します。これは*SIN*関数の逆関数です。*arithmetic_expression*は角度のラジアン表現です。

## 例題

この例題はラジアン (-0.73) で表現された角度のアークサインを返します:

```
SELECT ASIN(-0.73)
FROM TABLES_OF_ANGLES;
```

# ATAN

**ATAN** (*arithmetic_expression*)

## 説明

*ATAN*は*arithmetic_expression*のアークタンジェントを返します。これは*Tan*関数の逆関数です。*arithmetic_expression*は角度のラジアン表現です。

## 例題

この例題はラジアンで表現された角度 (-0.73) のアークタンジェントを返します:

```
SELECT ATAN(-0.73)
FROM TABLES_OF_ANGLES;
```

# ATAN2

**ATAN2** (*arithmetic_expression, arithmetic_expression*)

## 説明

*ATAN2*は"X"と"Y"座標のアークタンジェントを返します。"X"は一番目の*arithmetic_expression*で、"Y"は二番目の*arithmetic_expression*に渡します。

## 例題

この例題は渡されたxとy座標 (それぞれ0.52と0.60) のアークタンジェントを返します:

```
SELECT ATAN2( 0.52, 0.60 );
```

# AVG

AVG ([ALL | DISTINCT] *arithmetic_expression*)

## 説明

*AVG*関数は*arithmetic_expression*の平均値を返します。オプションの**ALL**と**DISTINCT**キーワードは、すべての値を含めるか、重複する値を取り除くか、指定するために使用します。

## 例題

この例題はMOVIESテーブル中、チケット販売数の最小値、最大値、平均値、および総数を返します:

```
SELECT MIN (Tickets_Sold),
MAX (Tickets_Sold),
AVG (Tickets_Sold),
SUM (Tickets_Sold)
FROM MOVIES
```

# BIT_LENGTH

**BIT_LENGTH** (*arithmetic_expression*)

## 説明

*BIT_LENGTH*関数は、ビット単位で*arithmetic_expression*の長さを返します。

## 例題

この例題は8を返します:

```
SELECT BIT_LENGTH( '01101011' );
```

# CAST

**CAST** (*arithmetic_expression* **AS** *sql_data_type_name*)

## 説明

*CAST*関数は*arithmetic_expression*をASキーワードの後に渡された*sql_data_type_name*データ型に変換します。

## 例題

この例題は、映画の公開年を整数に変換しています:

```
SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets
FROM MOVIES
WHERE Year_of_Movie >= CAST('1960' AS INT)
```

# CEILING

**CEILING** (*arithmetic_expression*)

## 説明

*CEILING*関数は、*arithmetic_expression*以上の最小整数値を返します。

## 例題

この例題は、-20.9と同じかそれよりも大きい最小整数値を返します:

```
CEILING (-20.9) `-20を返す
```

# CHAR

**CHAR** (*arithmetic_expression*)

## 説明

*CHAR*関数は、*arithmetic_expression*に渡された型に基づき。、固定長文字列を返します。

## 例題

この例題は、LastNameごとの最初の文字の整数に基づく文字を返します:

```
SELECT
CHAR (ASCII (SUBSTRING (LastName,1,1)))
FROM PEOPLE;
```

# CHAR_LENGTH

**CHAR_LENGTH** (*arithmetic_expression*)

## 説明

*CHAR_LENGTH*は、*arithmetic_expression*の文字数を返します。

## 例題

この例題は、重さが15 lbs未満の製品名の文字数を返します。

```
SELECT CHAR_LENGTH (Name)
FROM PRODUCTS
WHERE Weight < 15.00
```

# COALESCE

COALESCE (*arithmetic_expression, ..., arithmetic_expression*)

## 説明

*COALESCE*は、渡された*arithmetic_expression*のリストから、最初に見つかったNULLでない式を返します。すべての式がNULLの場合、**NULL**が返されます。

## 例題

この例題は、VATが0より大きい、2007年に発行された請求書の請求書番号を返します:

```
SELECT INVOICE_NO
FROM INVOICES
WHERE EXTRACT(YEAR(INVOICE_DATE)) = 2007
HAVING (COALESCE(INVOICE_VAT;0) > 0)
```

# CONCAT

**CONCAT** (*arithmetic_expression, arithmetic_expression*)

## 説明

*CONCAT*関数は、渡された二つの*arithmetic_expression*を結合した一つの文字列にして返します。

## 例題

この例題は、FirstNameとLastNameを一つの文字列にして返します:

```
SELECT CONCAT(CONCAT(PEOPLE.FirstName,' '), PEOPLE.LastName) FROM PERSONS;
```

# CONCATENATE

**CONCATENATE** (*arithmetic_expression, arithmetic_expression*)

## 説明

*CONCATENATE*関数は、渡された二つの*arithmetic_expression*を結合した一つの文字列にして返します。

## 例題

*CONCAT*の例題を参照。

# COS

COS（*arithmetic_expression*）

## 説明

*COS*関数は、*arithmetic_expression*のコサインを返します。*arithmetic_expression*はラジアンで表現された角度です。

## 例題

この例題はラジアンで表現された角度 (degrees * 180 / 3.1416) のコサインを返します:

```
SELECT COS(degrees * 180 / 3,1416)
FROM TABLES_OF_ANGLES;
```

# COT

COT (*arithmetic_expression*)

## 説明

*COT*関数は*arithmetic_expression*のコタンジェントを返します。*arithmetic_expression*はラジアンで表現された角度です。

## 例題

この例題は、ラジアンで表現された角度 (3,1416) のコタンジェントを返します:

```
SELECT COT(3,1416)
FROM TABLES_OF_ANGLES;
```

# COUNT

```
COUNT ([ [ [ALL |DISTINCT] arithmetic_expression] |* ] )
```

## 説明

*COUNT*関数は、*arithmetic_expression*中のNULLでない値の数を返します。オプションの**ALL**と**DISTINCT**キーワードは、すべての値を含めるか、重複する値を取り除くか、指定するために使用します。

代わりに * を渡すと、関数は重複およびNULL値を含む、テーブルの総レコード数を返します。

## 例題

この例題は、MOVIESテーブルに登録された映画の数を返します:

```
SELECT COUNT(*)
FROM MOVIES
```

# CURDATE

CURDATE ( )

## 説明

*CURDATE*関数は現在の日付を返します。

## 例題

この例題はINVOICESテーブルを作成し、現在の日付をINV_DATEカラムに挿入します:

```
ARRAY STRING(30;aDate;0)
Begin SQL
   CREATE TABLE INVOICES
   (INV_DATE VARCHAR(40));

   INSERT INTO INVOICES
   (INV_DATE)
   VALUES (CURDATE());

   SELECT *
   FROM INVOICES
   INTO :aDate;
End SQL
```

`aDate配列にはINSERTコマンドが実行された日付と時間が返される

# CURRENT_DATE

CURRENT_DATE ( )

## 説明

*CURRENT_DATE*は現在の日付をローカル時間で返します。

## 例題

この例題はINVOICESテーブルを作成し、現在の日付をINV_DATEカラムに挿入します:

```
ARRAY STRING(30;aDate;0)
Begin SQL
   CREATE TABLE INVOICES
   (INV_DATE VARCHAR(40));

   INSERT INTO INVOICES
   (INV_DATE)
   VALUES (CURRENT_DATE());

   SELECT *
   FROM INVOICES
   INTO :aDate;
End SQL
```

`aDate配列にはINSERTコマンドが実行された日付と時間が返される

# CURRENT_TIME

CURRENT_TIME ( )

## 説明

*CURRENT_TIME*関数は、現在のローカル時刻を返します。

## 例題

この例題はINVOICESテーブルを作成し、現在の時刻をINV_DATEカラムに挿入します:

```
ARRAY STRING(30;aDate;0)
Begin SQL
   CREATE TABLE INVOICES
   (INV_DATE VARCHAR(40));

   INSERT INTO INVOICES
   (INV_DATE)
   VALUES (CURRENT_TIME());

   SELECT *
   FROM INVOICES
   INTO :aDate;
End SQL
```

`aDate配列にはINSERTコマンドが実行された日付と時間が返される

# CURRENT_TIMESTAMP

```
CURRENT_TIMESTAMP ( )
```

## 説明

*CURRENT_TIMESTAMP*関数は現在の日付とローカル時間を返します。

## 例題

この例題はINVOICESテーブルを作成し、現在の日付と時刻をINV_DATEカラムに挿入します:

```
ARRAY STRING(30;aDate;0)
Begin SQL
   CREATE TABLE INVOICES
   (INV_DATE VARCHAR(40));

   INSERT INTO INVOICES
   (INV_DATE)
   VALUES (CURRENT_TIMESTAMP());

   SELECT *
   FROM INVOICES
   INTO :aDate;
End SQL
```

`aDate配列にはINSERTコマンドが実行された日付と時間が返される

# CURTIME

CURTIME ( )

## 説明

*CURTIME*関数は、現在の時間を秒単位で返します。

## 例題

この例題はINVOICESテーブルを作成し、現在の時刻をINV_DATEカラムに挿入します:

```
ARRAY STRING(30;aDate;0)
Begin SQL
   CREATE TABLE INVOICES
   (INV_DATE VARCHAR(40));

   INSERT INTO INVOICES
   (INV_DATE)
   VALUES (CURTIME());

   SELECT *
   FROM INVOICES
   INTO :aDate;
End SQL
```

`aDate配列にはINSERTコマンドが実行された日付と時間が返される

# DATABASE_PATH

DATABASE_PATH()

## 説明

**DATABASE_PATH**関数はカレントデータベースの完全パス名を返します。カレントデータベースは**USE DATABASE**コマンドを使用して変更できます。デフォルトのカレントデータベースはメインの4Dデータベースです。

返されるパス名はPOSIXフォーマットです。

## 例題

カレントのエクスターナルデータベースがTestBase.4DBという名前で、"C:¥MyDatabases"フォルダにあるとします。以下のコードを実行後:

```
C_TEXT($vCrtDatabasePath)
Begin SQL
   SELECT DATABASE_PATH()
   FROM _USER_SCHEMAS
   LIMIT 1
   INTO :$vCrtDatabasePath;
End SQL
```

*$vCrtDatabasePath*変数には"C:/MyDatabases/TestBase.4DB"が代入されます。

# DATE_TO_CHAR

**DATE_TO_CHAR** (*arithmetic_expression, arithmetic_expression*)

## 説明

関数は、最初の*arithmetic_expression*に渡された日付を、二番目の*arithmetic_expression*で指定されたフォーマットに基づき、文字表現にして返します。一番目の*arithmetic_expression*はTimestampまたはDurationタイプで、二番目はTextタイプです。

使用可能なフラグを以下に示します。一般的に、フォーマットフラグが大文字で始まり、そこに0が来る場合、必要に応じて数字文字列は0から始まります。そうでなければ先頭に0は付加されません。たとえば、ddに7が返される場合、Ddには07が返されます。

日および月の名前のフォーマットで大文字小文字を使い分けると、それは返される値に再現されます。例えば、"day"を渡すと"monday"が、"Day"の場合は"Monday"が、"DAY"のときには"MONDAY"が返されます。

am - 時間の値により、amまたはpm
pm - 時間の値により、amまたはpm
a.m. - 時間の値により、a.m.またはp.m.
p.m. - 時間の値により、a.m.またはp.m.

d - 曜日の数字表現 (1-7)
dd - 日 (1-31)
ddd - 年の日
day - 曜日名
dy - 曜日名の三文字短縮形

hh - 12時間ベースの時数 (0-11)
hh12 - 12時間ベースの時数 (0-11)
hh24 - 24時間ベースの時数 (0-23)

J - ユリウス日

mi - 分 (0-59)
mm - 月数 (0-12)

q - 年の四半期

ss - 秒 (0-59)
sss - ミリ秒 (0-999)

w - 月の週 (1-5)
ww - 年の週 (1-53)

yy - 年
yyyy - 年
[文字列] - ブラケット ([ ]) の間の文字列は解釈されず、そのまま挿入されます。
-.,:; スペース,タブ - は変更されずそのまま残されます。

## 例題

この例題は誕生日の曜日 (1-7) を返します:

```
SELECT DATE_TO_CHAR (Birth_Date,'d')
FROM EMPLOYERS;
```

# DAY

**DAY** (*arithmetic_expression*)

## 説明

*DAY*関数は、*arithmetic_expression*で渡された日付の日を返します。

## 例題

この例題は日付 "05-07-2007"の日を返します:

```
SELECT DAY('05-07-2007'); `returns 7
```

# DAYNAME

**DAYNAME**（*arithmetic_expression*）

## 説明

*DAYNAME*関数は、*arithmetic_expression*で渡された日付の曜日名を返します。

## 例題

この例題は、渡された誕生日の曜日を返します:

```
SELECT DAYNAME(Date_of_birth);
```

# DAYOFMONTH

**DAYOFMONTH** (*arithmetic_expression*)

## 説明

*DAYOFMONTH*関数は、*arithmetic_expression*で渡された日付の日を整数で返します (1から31の範囲)。

## 例題

Date_of_Birthフィールドを持つPEOPLE テーブルがあります。全員の誕生日付の日を取得します:

```
SELECT DAYOFMONTH(Date_of_Birth)
FROM PEOPLE;
```

# DAYOFWEEK

**DAYOFWEEK** (*arithmetic_expression*)

## 説明

*DAYOFWEEK*関数は、*arithmetic_expression*で渡された日付の、曜日の数値 (1から7、1が日曜日で7が土曜日) を返します。

## 例題

Date_of_Birthフィールドを持つPEOPLE テーブルがあります。全員の誕生日付の曜日を取得します:

```
SELECT DAYOFWEEK(Date_of_Birth)
FROM PEOPLE;
```

# DAYOFYEAR

**DAYOFYEAR** (*arithmetic_expression*)

## 説明

*DAYOFYEAR*関数は、*arithmetic_expression*で渡された日付の、年中の日数値 (1から366、1が1月1日) を返します。

## 例題

Date_of_Birthフィールドを持つPEOPLE テーブルがあります。全員の誕生日付の日を取得します:

```
SELECT DAYOFYEAR(Date_of_Birth)
FROM PEOPLE;
```

# DEGREES

**DEGREES**（*arithmetic_expression*）

## 説明

*DEGREES*関数は、*arithmetic_expression*の度数を返します。*arithmetic_expression*はラジアン単位の角度です。

## 例題

この例題はテーブルを作成し、PI値の度数を挿入します:

```
CREATE TABLE Degrees_table (PI_value float);
INSERT INTO Degrees_table VALUES
(DEGREES(PI()));
SELECT * FROM Degrees_table
```

# EXP

EXP（*arithmetic_expression*）

## 説明

*EXP*関数は、*arithmetic_expression*の指数を返します。xが*arithmetic_expression*で渡された値の時、eのx乗。

## 例題

この例題はeの15乗値を返します:

```
SELECT Exp( 15 ); `returns 3269017.3724721107
```

# EXTRACT

**EXTRACT** (**{YEAR** | **MONTH** | **DAY** | **HOUR** | **MINUTE** | **SECOND** | **MILLISECOND}** **FROM** *arithmetic_expression*)

## 説明

*EXTRACT*関数は、*arithmetic_expression*から展開された特定の部分を返します。*arithmetic_expression* はTimestamp型です。

## 例題

この例題は、請求書の発行月が1月の請求書番号を返します:

```
SELECT INVOICE_NO
FROM INVOICES
WHERE EXTRACT(MONTH(INVOICE_DATE)) = 1;
```

# FLOOR

**FLOOR** (*arithmetic_expression*)

## 説明

*FLOOR*関数は、*arithmetic_expression*以下の最大整数値を返します。

## 例題

この例題は、-20.9以下の最大整数値を返します:

```
FLOOR (-20.9); `returns -21
```

# HOUR

HOUR（*arithmetic_expression*）

## 説明

*HOUR*関数は、*arithmetic_expression*で渡された時間の時間部分を返します。返される値は0から23の間です。

## 例題

Delivery_Timeフィールドを持つINVOICESテーブルがあります。配送時刻の時間部分を取得します:

```
SELECT HOUR(Delivery_Time)
FROM INVOICES;
```

# INSERT

**INSERT** (*arithmetic_expression, arithmetic_expression, arithmetic_expression, arithmetic_expression*)

## 説明

**INSERT**関数は、文字列の指定された位置に、他の文字列を挿入します。一番目の*arithmetic_expression*は挿入先文字列です。二番目の*arithmetic_expression*は挿入位置、三番目の*arithmetic_expression*は挿入辞に削除する文字数、そして四番目の*arithmetic_expression*は挿入する文字列です。

## 例題

この例題は"Dear "をPEOPLE テーブルのFirstNameの前に挿入します:

```
SELECT INSERT (PEOPLE.FirstName,0,0,'Dear ') FROM PEOPLE;
```

# LEFT

**LEFT** (*arithmetic_expression, arithmetic_expression*)

## 説明

*LEFT*関数は、渡された*arithmetic_expression*の一番左の部分を返します。二番目の*arithmetic_expression*は取り出す文字数を指定します。

## 例題

この例題は、FirstNameと、LastNameの最初の2文字を返します:

```
SELECT FirstName, LEFT(LastName, 2)
FROM PEOPLE;
```

# LENGTH

**LENGTH** (*arithmetic_expression*)

## 説明

*LENGTH*関数は、*arithmetic_expression*の文字数を返します。

## 例題

この例題は重さが15 lbs未満の製品名の文字数を返します:

```
SELECT LENGTH (Name)
FROM PRODUCTS
WHERE Weight < 15.00
```

# LOCATE

**LOCATE** (*arithmetic_expression, arithmetic_expression, arithmetic_expression*)

**LOCATE** (*arithmetic_expression, arithmetic_expression*)

## 説明

*LOCATE*関数は、一番目の*arithmetic_expression*を二番目の*arithmetic_expression*中で検索し、最初に見つかるオカレンスの位置を返します。三番目の*arithmetic_expression*を渡して検索開始位置を指定することもできます。

## 例題

この例題はLastName中で最初に見つかるXの位置を返します:

```
SELECT FirstName, LOCATE('X',LastName)
FROM PEOPLE;
```

# LOG

LOG（*arithmetic_expression*）

## 説明

*LOG*関数は*arithmetic_expression*の自然対数を返します。

## 例題

この例題は50の自然対数を返します:

```
SELECT LOG ( 50 );
```

# LOG10

**LOG10** (*arithmetic_expression*)

## 説明

*LOG10*関数は10を底とした*arithmetic_expression*の対数を返します。

## 例題

この例題は10を底とした50の対数を返します:

```
SELECT LOG10( 50 );
```

# LOWER

**LOWER** (*arithmetic_expression*)

## 説明

*LOWER*関数は、*arithmetic_expression*で渡された文字列を小文字にして返します。

## 例題

この例題は製品名を小文字で返します:

```
SELECT LOWER (Name)
FROM PRODUCTS;
```

# LTRIM

**LTRIM**（*arithmetic_expression*[, *arithmetic_expression*])

## 説明

*LTRIM*関数は、*arithmetic_expression*の先頭に存在する空のスペースを取り除きます。オプションの二番目の*arithmetic_expression*を使用して、削除する文字を指定することもできます。

## 例題

この例題は製品名の先頭から空のスペースを取り除きます:

```
SELECT LTRIM(Name)
FROM PRODUCTS;
```

# MAX

**MAX**（*arithmetic_expression*）

## 説明

*MAX*関数は*arithmetic_expression*の最大値を返します。

## 例題

*SUM*と*AVG*の例題を参照。

# MILLISECOND

**MILLISECOND** (*arithmetic_expression*)

## 説明

*MILLISECOND*関数は、*arithmetic_expression*で渡された時間のミリ秒部分を返します。

## 例題

Delivery_Timeフィールドを持つINVOICESテーブルがあります。配送時間のミリ秒を表示します:

```
SELECT MILLISECOND(Delivery_Time)
FROM INVOICES;
```

# MIN

**MIN** (*arithmetic_expression*)

## 説明

*MIN*関数は、*arithmetic_expression*の最小値を返します。

## 例題

*SUM*と*AVG*の例題を参照。

# MINUTE

**MINUTE**（*arithmetic_expression*）

## 説明

*MINUTE*関数は、*arithmetic_expression*で渡された時間の分部分を返します。返される値の範囲は0から59です。

## 例題

Delivery_Timeフィールドを持つINVOICESテーブルがあります。配送時間の分を表示します:

```
SELECT MINUTE(Delivery_Time)
FROM INVOICES;
```

# MOD

**MOD**（*arithmetic_expression*, *arithmetic_expression*）

## 説明

*MOD*関数は、一番目の*arithmetic_expression*を二番目の*arithmetic_expression*で割った余りを返します。

## 例題

この例題は10を3で割った余りを返します:

```
Mod(10,3) `returns 1
```

# MONTH

**MONTH** (*arithmetic_expression*)

## 説明

*MONTH*関数は、*arithmetic_expression*で渡された日付の月部を返します (1から12まで)。

## 例題

Date_of_Birthフィールドを持つPEOPLEテーブルがあります。すべてのPEOPLEテーブルの誕生日の月を取得します:

```
SELECT MONTH(Date_of_Birth)
FROM PEOPLE;
```

# MONTHNAME

**MONTHNAME**（*arithmetic_expression*）

## 説明

*MONTHNAME*関数は、*arithmetic_expression*で渡された日付の月名を返します。

## 例題

この例題は渡された誕生日の月名を返します:

```
SELECT MONTHNAME(Date_of_birth);
```

# NULLIF

**NULLIF** (*arithmetic_expression, arithmetic_expression*)

## 説明

*NULLIF*関数は、一番目の*arithmetic_expression*と二番目の*arithmetic_expression*が等しい時**NULL**を返します。そうでなければ一番目の*arithmetic_expression*が返されます。二番目の*arithmetic_expression*は比較可能でなければなりません。

## 例題

この例題は総請求額が0のときNullを返します:

```
NULLIF(INVOICE_TOTAL,0);
```

# OCTET_LENGTH

**OCTET_LENGTH** (*arithmetic_expression*)

## 説明

*OCTET_LENGTH*関数は、続くホワイトスペースを含む、*arithmetic_expression*のオクテット数を返します。

## 例題

この例題はバイナリデータが格納されたカラムのオクテット数を返します:

```
SELECT OCTET_LENGTH (MyBinary_col)
FROM MyTable
WHERE MyBinary_col = '93FB';
` returns 2
```

# PI

**PI** ( )

## 説明

*PI*関数は、Pi定数の値を返します。

## 例題

*DEGREES*の例題を参照。

# POSITION

**POSITION** (*arithmetic_expression* **IN** *arithmetic_expression*)

## 説明

*POSITION*関数は、二番目の*arithmetic_expression*中、一番目の*arithmetic_expression*の開始位置を返します。文字列が見つからない場合、0が返されます。

## 例題

この例題は、LastName中単語"York"の開始位置を返します:

```
SELECT FirstName, POSITION('York' IN LastName)
FROM PEOPLE;
```

# POWER

**POWER** (*arithmetic_expression, arithmetic_expression*)

## 説明

*POWER*関数は、一番目の*arithmetic_expression*の、二番目の*arithmetic_expression*で指定したべき乗を返します。

## 例題

この例題はSourceValuesの3乗を返します:

```
SELECT SourceValues, POWER(SourceValues, 3)
FROM Values
ORDER BY SourceValues
`SourceValuesが2のとき、8が返される
```

# QUARTER

QUARTER (*arithmetic_expression*)

## 説明

*QUARTER*関数は、*arithmetic_expression*で渡した日付の、年の四半期を返します (1から4の範囲)。

## 例題

Date_of_Birthフィールドを持つPEOPLEテーブルがあります。PEOPLE内のレコードの誕生日の四半期を取得します:

```
SELECT QUARTER(Date_of_Birth)
FROM PEOPLE;
```

# RADIANS

**RADIANS** (*arithmetic_expression*)

## 説明

*RADIANS*関数は、*arithmetic_expression*のラジアン値を返します。*arithmetic_expression*は度数で表現された角度です。

## 例題

この例題は30度のラジアン値を返します:

```
RADIANS ( 30 ); ` 0.5236を返す
```

# RAND

**RAND** ([*arithmetic_expression*])

## 説明

*RAND*関数は、0から1の間の乱数小数を返します。オプションの*arithmetic_expression*でシード値を渡すことができます。

## 例題

この例題は、*RAND*関数で生成したIDを挿入します:

```
CREATE TABLE PEOPLE
(ID INT32,
Name VARCHAR);

INSERT INTO PEOPLE
(ID, Name)
VALUES(RAND, 'Francis');
```

# REPEAT

REPEAT（*arithmetic_expression, arithmetic_expression*）

## 説明

*REPEAT*関数は、一番目の*arithmetic_expression*を、二番目の*arithmetic_expression*だけ繰り返して、返します。

## 例題

この例題はこの関数がどのように動作するかを示しています:

```
SELECT REPEAT( 'repeat', 3 ) ` 'repeatrepeatrepeat'が返される
```

# REPLACE

REPLACE（*arithmetic_expression, arithmetic_expression, arithmetic_expression*）

## 説明

*REPLACE*関数は、一番目の*arithmetic_expression*中で二番目の*arithmetic_expression*のすべてのオカレンスを検索し、それぞれを三番目の*arithmetic_expression*で置き換えます。オカレンスが見つからない場合、一番目の*arithmetic_expression*が変更されずに返されます。

## 例題

この例題は"Francs"を"Euro"に置き換えます:

```
SELECT Name, REPLACE(Currency, 'Francs', 'Euro')
FROM PRODUCTS;
```

# RIGHT

**RIGHT**（*arithmetic_expression, arithmetic_expression*）

## 説明

*RIGHT*関数は、渡された*arithmetic_expression*の一番右の部分を返します。二番目の*arithmetic_expression*は取り出す文字数を指定します。

## 例題

この例題は、FirstNameと、LastNameの最後の2文字を返します:

```
SELECT FirstName, RIGHT(LastName, 2)
FROM PEOPLE;
```

# ROUND

**ROUND** (*arithmetic_expression*[, *arithmetic_expression*])

## 説明

*ROUND*関数は、一番目の*arithmetic_expression*を、二番目の*arithmetic_expression*を有効桁数として丸めます。二番目の*arithmetic_expression*が渡されない場合、最も近い整数値に丸められます。

## 例題

この例題は数値を小数点第二位を有効桁数として丸めます:

```
Round (1234.1966, 2) `returns 1234.2000
```

# RTRIM

**RTRIM** (*arithmetic_expression*[, *arithmetic_expression*])

## 説明

*RTRIM*関数は、*arithmetic_expression*の最後尾に存在する空のスペースを取り除きます。オプションの二番目の*arithmetic_expression*を使用して、削除する文字を指定することもできます。

## 例題

この例題は製品名の最後尾から空のスペースを取り除きます:

```
SELECT RTRIM(Name)
FROM PRODUCTS;
```

# SECOND

SECOND (*arithmetic_expression*)

## 説明

*SECOND*関数は、*arithmetic_expression*で渡された時間の秒部分を返します。値の範囲は0から59です。

## 例題

Delivery_Timeフィールドを持つINVOICESテーブルがあります。配送時間の秒を表示します:

```
SELECT SECOND(Delivery_Time)
FROM INVOICES;
```

# SIGN

SIGN (*arithmetic_expression*)

## 説明

*SIGN*関数は、*arithmetic_expression*の符号を返します (正数は+1、負数は-1、または0)。

## 例題

この例題はINVOICESテーブル中、すべての負のAMOUNTを返します:

```
SELECT AMOUNT
FROM INVOICES
WHERE SIGN(AMOUNT) = -1;
```

# SIN

SIN（*arithmetic_expression*）

## 説明

*SIN*関数は、*arithmetic_expression*のサイン値を返します。*arithmetic_expression*はラジアン表現の角度です。

## 例題

この例題はラジアンで表現された角度のサインを返します:

```
SELECT Sin(radians)
FROM TABLES_OF_ANGLES;
```

# SPACE

SPACE（*arithmetic_expression*）

## 説明

*SPACE*関数は、*arithmetic_expression*で指定した数のスペース文字を返します。*arithmetic_expression*が0未満の場合、**NULL**値が返されます。

## 例題

この例題はLastNameの前に、3つのスペースを追加します:

```
SELECT CONCAT(SPACE(3),PERSONS.LastName) FROM PEOPLE;
```

# SQRT

**SQRT** (*arithmetic_expression*)

## 説明

*SQRT*関数は、*arithmetic_expression*の平方根を返します。

## 例題

この例題は運賃の平方根を返します:

```
SELECT Freight, SQRT(Freight) AS "Square root of Freight"
FROM Orders
```

# SUBSTRING

**SUBSTRING** (*arithmetic_expression*, *arithmetic_expression*, [*arithmetic_expression*])

## 説明

*SUBSTRING*関数は、一番目の*arithmetic_expression*の部分文字列を返します。二番目の*arithmetic_expression*は部分文字列の開始位置、オプションの三番目の*arithmetic_expression*は返す文字数を指定します。三番目の*arithmetic_expression*が渡されないと、関数は開始位置から終わりまでの文字を返します。

## 例題

この例題は、Store_nameの2番目から4文字を返します:

```
SELECT SUBSTRING (Store_name,2,4)
FROM Geography
WHERE Store_name = 'Paris';
```

# SUM

**SUM** ([**ALL** |**DISTINCT**] *arithmetic_expression*)

## 説明

*SUM*関数は、*arithmetic_expression*の合計を返します。オプションの**ALL**と**DISTINCT**キーワードは、 すべての値を含めるか、重複する値を取り除くか、指定するために使用します。

## 例題

この例題は、売り上げ予想から実際の売り上げを差し引いた値、および最小売り上げと最大売り上げをそれぞれ売り上げ予想で割って100掛けたものを返します:

```
SELECT MIN ( ( SALES * 100 ) / QUOTA ),
MAX ( ( SALES * 100 ) / QUOTA ),
SUM ( QUOTA ) - SUM ( SALES )
FROM SALES_PERSONS
```

# TAN

**TAN** (*arithmetic_expression*)

## 説明

*TAN*関数は、*arithmetic_expression*のタンジェントを返します。*arithmetic_expression*はラジアン表現の角度です。

## 例題

この例題は、ラジアン表現の角度のタンジェントを返します:

```
SELECT TAN(radians)
FROM TABLES_OF_ANGLES;
```

# TRANSLATE

**TRANSLATE** (*arithmetic_expression, arithmetic_expression, arithmetic_expression*)

## 説明

*TRANSLATE*関数は、一番目の*arithmetic_expression*中、二番目の*arithmetic_expression*文字のそれぞれのオカレンスを、対応する三番目の*arithmetic_expression*文字で置き換えて返します。

この置き換えは一文字ごとに行われます。例えば二番目の*arithmetic_expression*の一文字目が一番目の*arithmetic_expression*中に見つかれば、その文字を三番目の*arithmetic_expression*の一文字目で置き換えます。

三番目の*arithmetic_expression*の文字数が二番目のそれよりも少ない場合、つまり二番目の*arithmetic_expression*に対応する文字が三番目にない場合、その文字は一番目の*arithmetic_expression*から削除されます。例えば二番目の*arithmetic_expression*が5文字あり、三番目の*arithmetic_expression*に4文字しかない場合、5番目の文字が一番目の*arithmetic_expression*で見つかるとその文字は返される値から削除されます。

## 例題

この例題はすべての"a"を"1"に、"b"を"2"に置き換えます:

```
TRANSLATE ('abcd', 'ab', '12') ` returns '12cd'
```

# TRIM

TRIM ([[LEADING |TRAILING |BOTH] [*arithmetic_expression*] FROM] *arithmetic_expression*)

## 説明

*TRIM*関数は、*arithmetic_expression*の先頭および/または最後尾から空のスペース、または (一番目の*arithmetic_expression*が渡された場合) 指定された文字を取り除きます。

**LEADING**を渡して*arithmetic_expression*の始まりのスペースや文字を取り除く、あるいは**TRAILING**を渡して終わりから文字を取り除く、**BOTH**で両方から文字を取り除くなどを指定できます。これらのキーワードが渡されない場合、**BOTH**を渡したのと同じです。つまりスペースまたは文字が*arithmetic_expression*のはじめと終わり両側から取り除かれます。

オプションの一番目の*arithmetic_expression*は、二番目の*arithmetic_expression*から取り除く文字を指定するために使用します。省略されると、空のスペースが取り除かれます。

## 例題

この例題では、製品名から空のスペースを取り除きます:

```
SELECT TRIM(Name)
FROM PRODUCTS;
```

# TRUNC

**TRUNC**（*arithmetic_expression*[, *arithmetic_expression*])

## 説明

*TRUNC*関数は、一番目の*arithmetic_expression*を、小数点の右側の"x"までで切り捨てて返します。"x"はオプションの二番目の*arithmetic_expression*で指定します。二番目の*arithmetic_expression*が渡されない場合、小数部が切り捨てられます。

## 例題

この関数は、渡された数値の小数第二位を切り捨てて返します:

```
Trunc(2.42 , 1) `returns 2.40
```

# TRUNCATE

**TRUNCATE** (*arithmetic_expression*[, *arithmetic_expression*])

## 説明

*TRUNCATE*関数は、一番目の*arithmetic_expression*を、小数点の右側の"x"の位置で切り捨てて返します。"x"はオプションの二番目の*arithmetic_expression*で指定します。二番目の*arithmetic_expression*が渡されない場合、小数部が切り捨てられます。

## 例題

*TRUNC*関数の例題を参照。

# UPPER

**UPPER** (*arithmetic_expression*)

## 説明

*UPPER*関数は、*arithmetic_expression*で渡された文字列を大文字にして返します。

## 例題

この例題は製品名を大文字で返します:

```
SELECT UPPER (Name)
FROM PRODUCTS;
```

# WEEK

**WEEK** (*arithmetic_expression*)

## 説明

*WEEK*関数は、*arithmetic_expression*で渡された日付の、年の週数を返します (1から54の範囲)。週は日曜日から始まり、1月1日が常に最初の週です。

## 例題

この例題は渡された誕生日の週数を返します:

```
SELECT WEEK(Date_of_birth);
```

# YEAR

YEAR (*arithmetic_expression*)

## 説明

*YEAR*関数は、*arithmetic_expression*で渡された日付の年部分を返します。

## 例題

Date_of_Birthフィールドを持つPEOPLEテーブルがあります。誕生日の年を所得します:

```
SELECT YEAR(Date_of_Birth)
FROM PEOPLE;
```

# Appendix

## 4D - SQLリファレンス - コマンドリスト (文字順)