









4D - SQL Reference

-  Tutorial
-  Using SQL in 4D
-  SQL Commands
-  Syntax rules
-  Transactions
-  Functions
-  Appendix
-  Alphabetische Liste der Befehle

✦ Tutorial

- ✦ Introduction
- ✦ Receiving an SQL query result in a variable
- ✦ Using the WHERE clause
- ✦ Receiving an SQL query result into arrays
- ✦ Using CAST
- ✦ Using the ORDER BY clause
- ✦ Using the GROUP BY clause
- ✦ Using Statistical functions
- ✦ Using the HAVING clause
- ✦ Calling 4D methods inside the SQL code
- ✦ Joins
- ✦ Using Aliases
- ✦ Subqueries
- ✦ SQL code error tracking and debugging
- ✦ Data Definition Language
- ✦ External connections
- ✦ Connection to the 4D SQL engine via the ODBC Driver

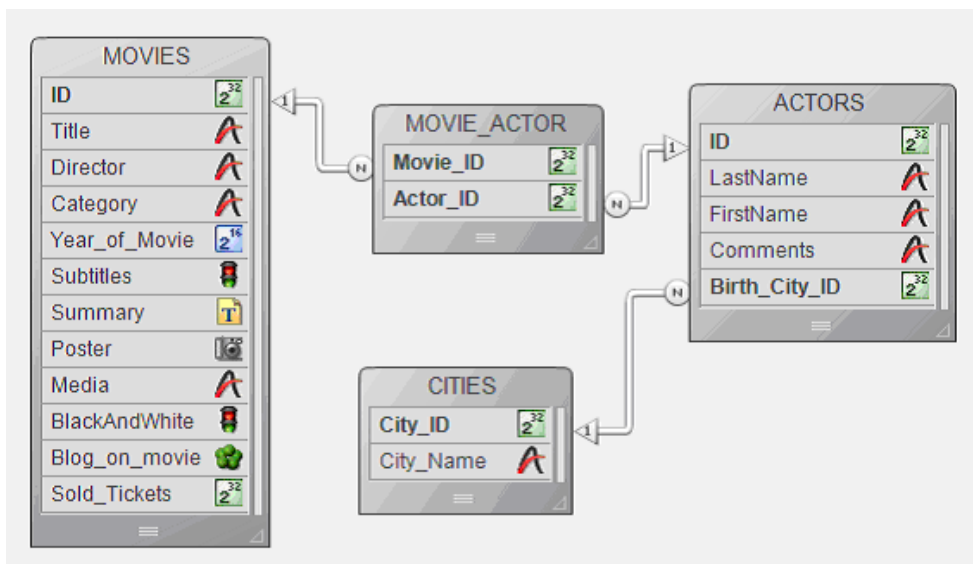
🌱 Introduction

SQL (Structured Query Language) is a tool for creating, organizing, managing and retrieving data stored by a computer database. SQL is not a database management system itself nor a stand-alone product; however, SQL is an integral part of a database management system, both a language and a tool used to communicate with this system.

The goal of this tutorial is not to teach you how to work with SQL (for this you can find documentation and links on the Internet), nor to teach you how to use and/or program in 4D. Instead, its purpose is to show you how to manage SQL inside 4D code, how to retrieve data using SQL commands, how to pass parameters and how to get the results after a SQL query.

Description of the database that accompanies this tutorial

All the examples that will be detailed in this document were fully tested and verified in one of the example databases named "4D SQL Code Samples" [that you can download from our ftp server \(ftp://ftp-public.4d.fr/Documents/Products_Documentation/LastVersions/Line_12/4D_SQL_Code_Samples.zip\)](ftp://ftp-public.4d.fr/Documents/Products_Documentation/LastVersions/Line_12/4D_SQL_Code_Samples.zip). The structure is as follows:



The MOVIES table contains information about 50 movies, such as the title, the director, the category (Action, Animation, Comedy, Crime, Drama, etc.), the year it was released, whether or not it has subtitles, a brief summary, a picture of its poster, the type of media (DVD, VHS, DivX), whether it is in black and white, a blog saved in a BLOB, and the number of tickets sold.

The ACTORS table contains information regarding the actors of the movies such as an ID, their last and first names, any comments and the ID of the city where the actor was born.

The CITIES table contains information regarding the name and ID of the cities where the actors were born.

The MOVIE_ACTOR table is used to simulate a Many-to-Many relation between the MOVIES and ACTORS tables.

All the information you need to launch every example described in the tutorial is situated in the following main window which you can access by selecting the **Demo SQL>Show Samples** menu command:

Ⓜ
✖

Query Mode

Using 4D code

Using SQL code

Using ODBC

Using "Query by SQL"

Using Dynamic SQL

Trace the code

Queries

SQL query results in variables

Using Statistical functions

WHERE clause

HAVING clause

SQL query results in arrays

Calling 4D methods

Using CAST

Joins

ORDER BY clause

Using Aliases

GROUP BY clause

Subqueries

Error tracking

Debugging SQL code

Using ON ERR CALL

Data Definition Language

DDL

Listbox

Years	Titles	Directors	Media	Tickets Sold	Number of actors

Group of arrays

Years (aMovieYear)	Titles (aTitles)	Directors (aDirectors)	Media (aMedia)	Tickets Sold (aSoldTickets)	Number of actors (aNActors)

✚ Receiving an SQL query result in a variable

To start with a very simple query: we would like to know how many movies are in the Video Library. In the 4D language, the code would be:

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

- The first way to interact in a similar manner with the SQL engine is by placing the query between the **Begin SQL** and **End SQL** tags. Thus, the simple query above becomes:

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

- As you can see, you can receive the result of the query in a variable (in our case \$AllMovies) that is enclosed between "<<" and ">>". Another way to reference any type of valid 4D expression (variable, field, array, "expression...") is to place a colon ":" in front of it:

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

Special attention should be paid to inter-process variables, where the notation is a little bit different: you must place an inter-process variable between "[" and "]":

```
C_LONGINT($AllMovies)
<>AllMovies:=0
Begin SQL
    SELECT COUNT(*)
    FROM MOVIES
```



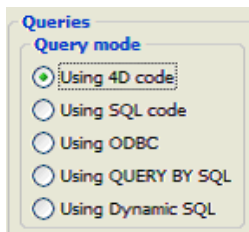
```
SELECT *
FROM MOVIES
WHERE ID <> 0
```

- The fourth way to interact with the new SQL Engine is using the dynamic SQL *EXECUTE IMMEDIATE* command. The query above becomes

```
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")
```

Warning: You can see that in this last example, we use process variables. This is necessary when you want to use the database in compiled mode. In this context, in fact, it is not possible to use local variables with the **EXECUTE IMMEDIATE** command.

To test all the above examples, launch the "4D SQL Code Samples" database and go to the main dialog box. On the left side of the dialog, you can choose the query mode:



Then press the **SQL query results in variables** button.

✚ Using the WHERE clause

If we now want to know how many movies more recent or equal to 1960 are in the Video Library. The code 4D would be:

```
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")
```

- Using SQL code, the above query becomes:

```
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")
```

- Using the generic SQL commands, the above query becomes:

```
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")
```

- Using the **QUERY BY SQL** command, the above query becomes:

```
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")
```

- Using the SQL *EXECUTE IMMEDIATE* command, the query above becomes:

```
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 :$tQueryTxt;
End SQL
ALERT("The Video Library contains "+String($NoMovies)+" movies more recent or equal to 1960")
```

As in the previous section, in order to test all the above examples, simply launch the "4D SQL Code Samples" database and go to the main window. You can then choose the query mode and press the **WHERE clause** button.

✚ Receiving an SQL query result into arrays

Now we want to pass a variable to the SQL query containing the year (and not the year itself, hard-coded) and get all the movies released in 1960 or more recently. In addition, for each movie found, we also want information such as the year, title, director, media used and tickets sold. The solution is to receive this information in arrays or in a list box.

- The initial query in 4D code would be:

```
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)
` Initialize the rest of the list box columns in order to display the information
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- Using SQL code, the above query becomes:

```
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
` Initialize the rest of the list box columns in order to display the information
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

As you can see:

- We can pass a variable (\$MovieYear) to the SQL query using the same notation as for receiving parameters.
- The SQL query result is saved in the aMovieYear, aTitles, aDirectories, aMedias and aSoldTickets arrays. They are displayed in the main window in two ways:
 - Using a group of arrays:
 - Using a list box having columns with the same names:
- Using generic SQL commands, the above query becomes:

```

ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectories;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;aDirectories;aMedias;aSoldTickets)
SQL LOAD RECORD(SQL_all records)
SQL LOGOUT
  ` Initialize the rest of the list box columns in order to display the information
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))

```

- Using the **QUERY BY SQL** command, the above query becomes:

```

ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectories;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;aDirectories;
[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)
  ` Initialize the rest of the list box columns in order to display the information
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))

```

- Using the SQL *EXECUTE IMMEDIATE* command, the query above becomes:

```
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
```

```
` Initialize the rest of the list box columns in order to display the information
```

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

To test all the above examples, launch the "4D SQL Code Samples" database and go to the main window. You can then choose the query mode and press the **SQL query results in arrays** button.

✚ Using CAST

The SQL standard has fairly restrictive rules about combining data of different types in expressions. Usually the DBMS is in charge of automatic conversion. However, the SQL standard requires that the DBMS must generate an error if you try to compare numbers and character data. In this context the *CAST* expression is very important, especially when we use SQL within a programming language whose data types do not match the types supported by the SQL standard. You will find below the query of the **Receiving an SQL query result into arrays** section modified slightly in order to use the *CAST* expression.

- The initial query in 4D code would be:

```
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)
  ` Initialize the rest of the list box columns in order to display the information
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- Using SQL code, the above query becomes:

```
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
  ` Initialize the rest of the list box columns in order to display the information
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

- Using generic SQL commands, the above query becomes:


```
EXECUTE IMMEDIATE :$tQueryTxt;
```

End SQL

` Initialize the rest of the list box columns in order to display the information

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

To test all the above examples, launch the "4D SQL Code Samples" database and go to the main window. You can then choose the query mode and press the **Using CAST** button.

✚ Using the ORDER BY clause

This time we would like to get all the movies that are released in 1960 or more recently, and for each movie we also want additional information such as the year, title, director, media used and tickets sold. The result must be sorted by the year.

- The initial query in 4D code would be:

```
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;>)
```

- Using SQL code, the above query becomes:

```
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
```

- Using generic SQL commands, the above query becomes:


```

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

```

- Using the **QUERY BY SQL** command, the above query becomes:

```

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;>)

```

- Using the SQL *EXECUTE IMMEDIATE* command, the query above becomes:

```

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
```

To test all the above examples, launch the "4D SQL Code Samples" database and go to the main window. You can then choose the query mode and press the **ORDER BY clause** button.

✚ Using the GROUP BY clause

We would like to get some information about the number of tickets sold each year starting with 1979. The result will be sorted by year. To do this, we must total all the tickets sold for every movie in each year more recent than 1979, and then sort the result by year.

- The initial query in 4D code would be:

```
` Using standard 4D code
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
` Initialize the rest of the list box columns in order to display the information
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))
```

- Using the SQL code, the above query becomes:

```
` Using 4D SQL
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)
```



```

If([MOVIES]Year_of_Movie#<math>\$vCrtMovieYear</math>)
  <math>\$vCrtMovieYear</math>:=[MOVIES]Year_of_Movie
  <math>\$vInd</math>:=<math>\$vInd</math>+1
  INSERT IN ARRAY(aMovieYear;<math>\$vInd</math>;1)
  aMovieYear{<math>\$vInd</math>}:=<math>\$vCrtMovieYear</math>
  INSERT IN ARRAY(aSoldTickets;<math>\$vInd</math>;1)
End if
aSoldTickets{<math>\$vInd</math>}:=aSoldTickets{<math>\$vInd</math>}+[MOVIES]Sold_Tickets
NEXT RECORD([MOVIES])
End for
  ` Initialize the rest of the list box columns in order to display the information
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))

```

- Using the SQL *EXECUTE IMMEDIATE* command, the query above becomes:

```

  ` Using dynamic SQL by EXECUTE IMMEDIATE
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
  ` Initialize the rest of the list box columns in order to display the information
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))

```

To test all the above examples, launch the "4D SQL Code Samples" database and go to the main window. You can then choose the query mode and press the **GROUP BY clause** button.

✚ Using Statistical functions

Sometimes it can be useful to get statistical information about certain values. SQL includes many aggregate functions like **MIN**, **MAX**, **AVG**, **SUM** and so on. Using aggregate functions, we would like to get information about the number of tickets sold each year. The result will be sorted by year.

To do this, we must total all the tickets sold for each movie and then sort the result by year.

- The initial query in 4D code would be:

```
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)
$AlertTxt:=""
$AlertTxt:=$AlertTxt+"Minimum tickets sold: "+String($vMin)+Char(13)
$AlertTxt:=$AlertTxt+"Maximum tickets sold: "+String($vMax)+Char(13)
$AlertTxt:=$AlertTxt+"Average tickets sold: "+String($vAverage)+Char(13)
$AlertTxt:=$AlertTxt+"Total tickets sold: "+String($vSum)+Char(13)
```

- Using SQL code, the above query becomes:

```
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
```

```

$AlertTxt:= ""
$AlertTxt:=$AlertTxt+"Minimum tickets sold: "+String($vMin)+Char(13)
$AlertTxt:=$AlertTxt+"Maximum tickets sold: "+String($vMax)+Char(13)
$AlertTxt:=$AlertTxt+"Average tickets sold: "+String($vAverage)+Char(13)
$AlertTxt:=$AlertTxt+"Total tickets sold: "+String($vSum)+Char(13)
ALERT($AlertTxt)

```

- Using generic SQL commands, the above query becomes:

```

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
$AlertTxt:= ""
$AlertTxt:=$AlertTxt+"Minimum tickets sold: "+String($vMin)+Char(13)
$AlertTxt:=$AlertTxt+"Maximum tickets sold: "+String($vMax)+Char(13)
$AlertTxt:=$AlertTxt+"Average tickets sold: "+String($vAverage)+Char(13)
$AlertTxt:=$AlertTxt+"Total tickets sold: "+String($vSum)+Char(13)
ALERT($AlertTxt)

```

- Using the SQL *EXECUTE IMMEDIATE* command, the query above becomes:

```

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

```

```
$AlertTxt:=""  
$AlertTxt:=$AlertTxt+"Minimum tickets sold: "+String($vMin)+Char(13)  
$AlertTxt:=$AlertTxt+"Maximum tickets sold: "+String($vMax)+Char(13)  
$AlertTxt:=$AlertTxt+"Average tickets sold: "+String($vAverage)+Char(13)  
$AlertTxt:=$AlertTxt+"Total tickets sold: "+String($vSum)+Char(13)  
ALERT($AlertTxt)
```

To test all the above examples, launch the "4D SQL Code Samples" database and go to the main window. You can then choose the query mode and press the **Using Aggregate functions** button.

✚ Using the HAVING clause

We would now like to get the total amount of tickets sold per year starting with 1979, but not including those with over 10,000,000 tickets sold. The result will be sorted by year. To do this, we must total all the tickets sold for every movie in each year more recent than 1979, remove those where the total amount of tickets sold is greater than 10,000,000, and then sort the result by year.

- The initial query in 4D code would be:

```
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
  ` Initialize the rest of the list box columns in order to display the information
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))
```

- Using SQL code, the above query becomes:


```

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
  \ Initialize the rest of the list box columns in order to display the information
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))

```

- Using the SQL *EXECUTE IMMEDIATE* command, the query above becomes:

```

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

` Initialize the rest of the list box columns in order to display the information

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))

To test all the above examples, launch the "4D SQL Code Samples" database and go to the main window. You can then choose the query mode and press the **HAVING clause** button.

🔌 Calling 4D methods inside the SQL code

We would now like to know something about the actors for each movie: more specifically, we are interested in finding all the movies with at least 7 actors. The result will be sorted by year. To do this, we will use a 4D function (Find_Nr_Of_Actors) that receives the movie ID as unique parameter and returns the number of actors that played in that movie:

```
`(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])
```

- The initial query in 4D code would be:

```
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;>)
```


- Using the **QUERY BY SQL** command, the above query becomes:

```

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;>)

```

- Using the SQL **EXECUTE IMMEDIATE** command, the query above becomes:

```

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
```

To test all the above examples, launch the "4D SQL Code Samples" database and go to the main window. You can then choose the query mode and press the **Calling 4D methods** button.

🔗 Joins

We would now like to find out the city of birth for each actor. The list of actors is in the ACTORS table and the list of cities is in the CITIES table. To execute this query we need to join the two tables: ACTORS and CITIES.

- The initial query in 4D code would be:

```
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
` Initialize the rest of the list box columns in order to display the information
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)
```

- Using SQL code, the above query becomes:

```
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
` Initialize the rest of the list box columns in order to display the information
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))
```


✚ Using Aliases

If an SQL query is too complex and contains long names that make it difficult to read, it is possible to use aliases in order to improve its readability.
Here is the previous example using two aliases: Act for the ACTORS table and Cit for the CITIES table.

- The initial query in 4D code would be:

```
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
` Initialize the rest of the list box columns in order to display the information
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)
```

- Using SQL code, the above query becomes:

```
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
` Initialize the rest of the list box columns in order to display the information
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
```


📌 Subqueries

We would now like to get some statistical information regarding the tickets sold: what are the movies where the tickets sold are greater than the average tickets sold for all the movies. To execute this query in SQL, we will use a query within a query, in other words, a subquery.

- The initial query in 4D code would be:

```
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
  Initialize the rest of the list box columns in order to display the information
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;>)
```

- Using SQL code, the above query becomes:

```
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
  Initialize the rest of the list box columns in order to display the information
ARRAY INTEGER(aMovieYear;Size of array(aTitles))
ARRAY TEXT(aDirectors;Size of array(aTitles))
ARRAY TEXT(aMedias;Size of array(aTitles))
```



```
$tQueryTxt:=$tQueryTxt+" INTO :aTitles, :aSoldTickets"
```

```
Begin SQL
```

```
EXECUTE IMMEDIATE :$tQueryTxt;
```

```
End SQL
```

```
` Initialize the rest of the list box columns in order to display the information
```

```
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))
```

To test all the above examples, launch the "4D SQL Code Samples" database and go to the main window. You can then choose the query mode and press the **Subqueries** button.

🔌 SQL code error tracking and debugging

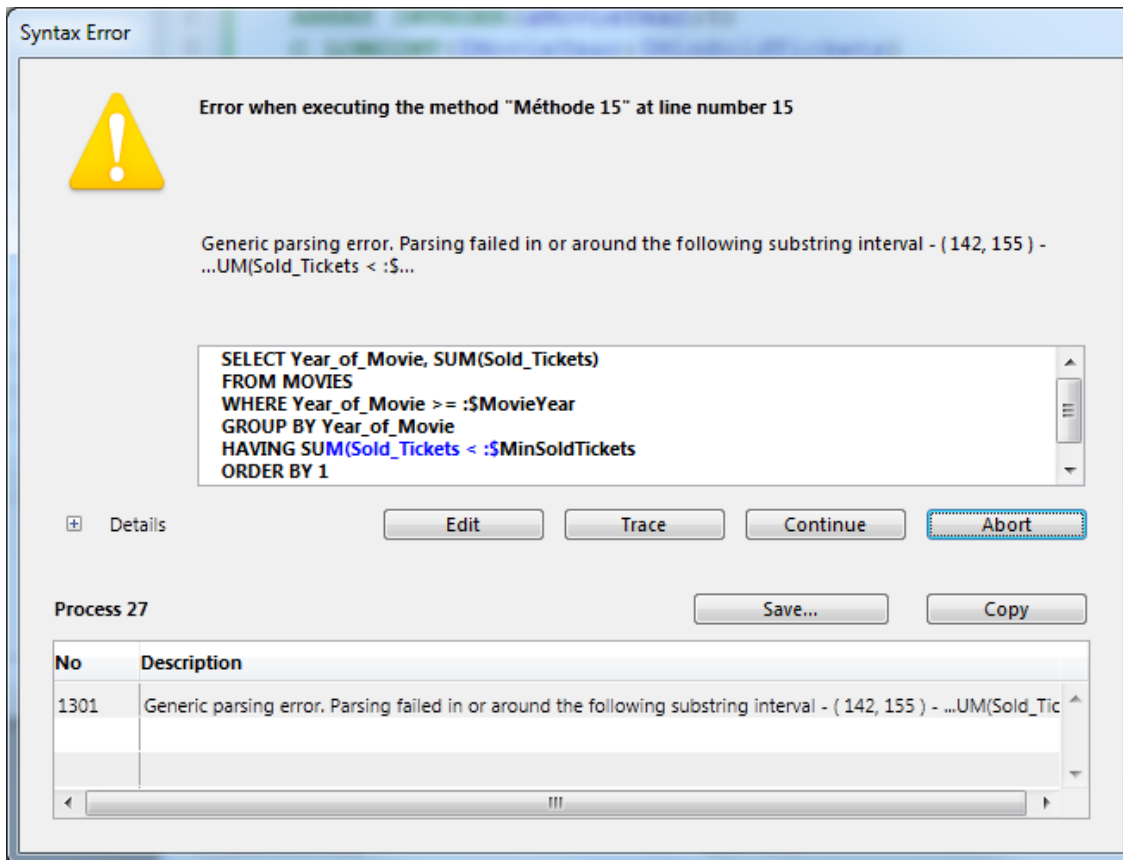
In 4D, there are two main possibilities for tracing and correcting your code: either using the **Debugger** to trace and correct any errors, or calling the **ON ERR CALL** command to catch the error and initiate the appropriate action. We can use both of these techniques to solve problems encountered with the SQL code.

Here is an example where a right parenthesis is missing intentionally: instead of **HAVING SUM(Sold_Tickets <:\$MinSoldTickets)**, we have **HAVING SUM(Sold_Tickets <:\$MinSoldTickets**.

```
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
```

As you can see in the window below, the application detects the error and opens the **Syntax Error Window** which provides more detailed information about the error and the place where it occurred. It is then easy to fix by simply pressing the **Edit** button.



If the error is more complex, the application provides more information including the stack content, which can be displayed by pressing the **Details** button.

To test the above example, in the main window of the "4D SQL Code Samples" database, press the **Debugging SQL code** button.

The second main possibility for tracking SQL errors is using the **ON ERR CALL** command. Here is an example that sets the SQL_Error_Handler method to catch errors encountered in the SQL code.

```

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

```

` Trigger the SQL_Error_Handler method to catch (trap) errors

```

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

```

` Disable the SQL_Error_Handler method

```

ON ERR CALL("")

```

```

If(MySQL_Error#0)

```

```

  ALERT("SQL Error number: "+String(MySQL_Error))

```

```

End if

```

The *SQL_Error_Handler* method is as follows:

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

To test the above example, in the main window of the "4D SQL Code Samples" database, press the **Using ON ERR CALL** button.

Data Definition Language

Using the SQL Data Definition Language (DDL), you can define and manage the database structure.

With DDL commands, you can create or alter tables and fields, as well as add and/or remove data.

Here is a simple example that creates a table, adds a few fields, then fills those fields with some data.

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

To test the above example, in the main window of the "4D SQL Code Samples" database, press the DDL button.

Note: This example will only work once because if you press the "DDL" button a second time, you will get an error message telling you that the table already exists.

External connections

4D allows you to use external databases, in other words to execute SQL queries on databases other than the local one. To do this, you can connect to any external data source via ODBC or directly to other 4D databases.

Here are the commands that allow you to manage connections with external databases:

- **Get current data source** tells you the ODBC data source used by the application.
- **GET DATA SOURCE LIST** can be used to get the list of ODBC data sources installed on the machine.
- **SQL LOGIN** allows you to connect to an external database directly or via an ODBC data source installed on the machine.
- **SQL LOGOUT** can be used to close any external connection and to reconnect to the local 4D database.
- **USE DATABASE** (SQL command) can be used to open an external 4D database using the 4D SQL engine.

The example below shows how to connect to an external data source (ORACLE), how to get data from the ORACLE database, and then how to disconnect from the ORACLE database and return to the local database.

Suppose that there is a valid data source named "Test_ORACLE_10g" installed in the system.

```
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
```

```
`By default the current DSN is the local one, ";DB4D_SQL_LOCAL;", which is the value of the
SQL_INTERNAL constant
```

```
$Crt_DSN:=Get current data source
ALERT("The current DSN is "+$Crt_DSN)
```

```
`Do something on the local database
```

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

```
`Get the data sources of the User type defined in the ODBC manager
```

```
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)
```

```
  `Establish a connection between 4D and the data source $My_ORACLE_DSN="Test_Oracle_10g"
```

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

```
  `The current DSN is the ORACLE one
```

```
$CrLf_DSN:=Get current data source
```

```
ALERT("The current DSN is "+$CrLf_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)
```

```
  `Do something on the external (ORACLE) database
```

```
Begin SQL
```

```
  SELECT ENAME FROM EMP INTO :aTitles
```

```
End SQL
```

```
  `Close the external connection opened with the SQL LOGIN command
```

```
SQL LOGOUT
```

```
  `The current DSN becomes the local one
```

```
$CrLf_DSN:=Get current data source
```

```
ALERT("The current DSN is "+$CrLf_DSN)
```

```
Else
```

```
  ALERT("ORACLE DSN not installed")
```

```
End if
```

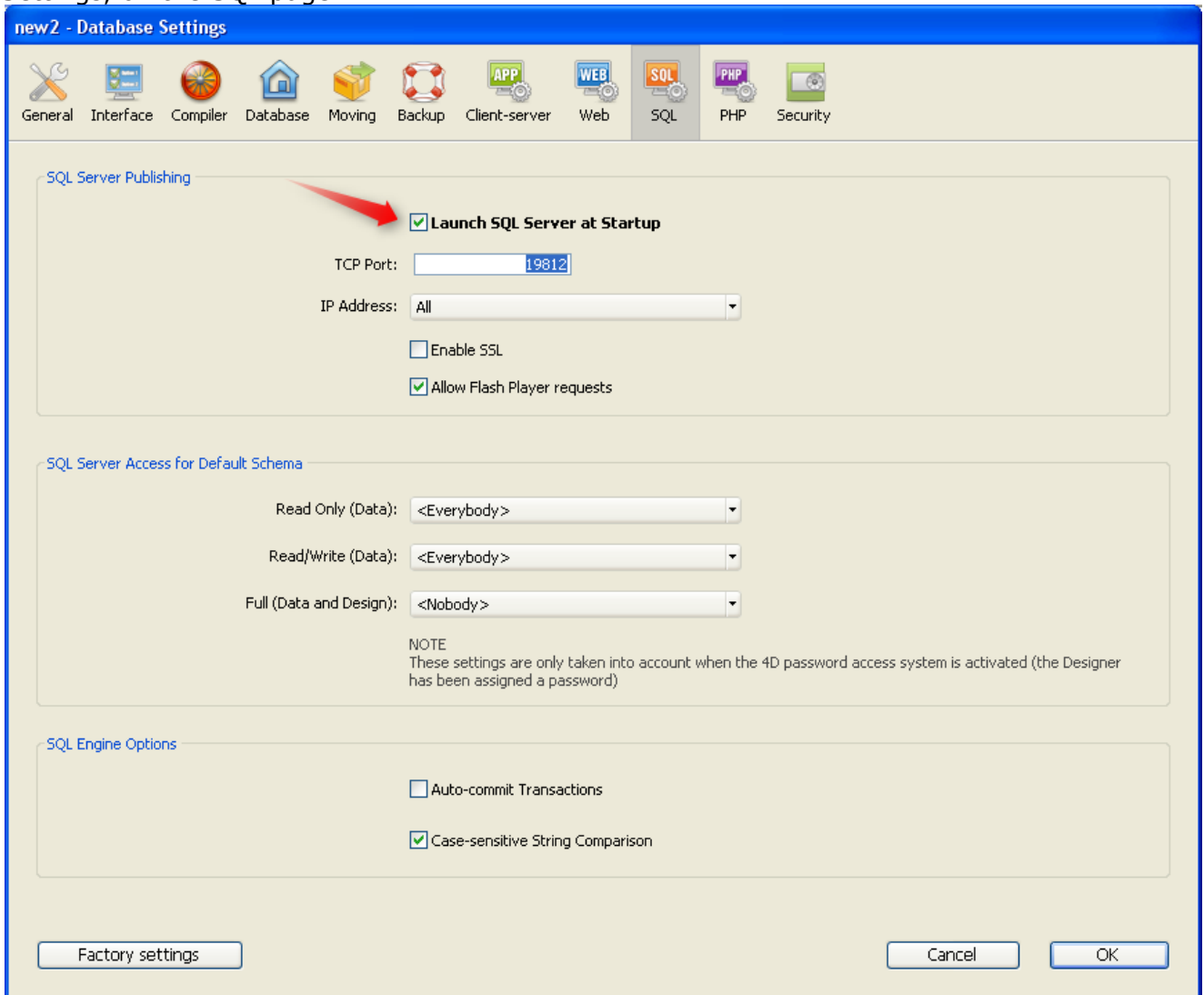
To test the above example, in the main window of the "4D SQL Code Samples" database, press the **Connect to ORACLE** button.

🔧 Connection to the 4D SQL engine via the ODBC Driver

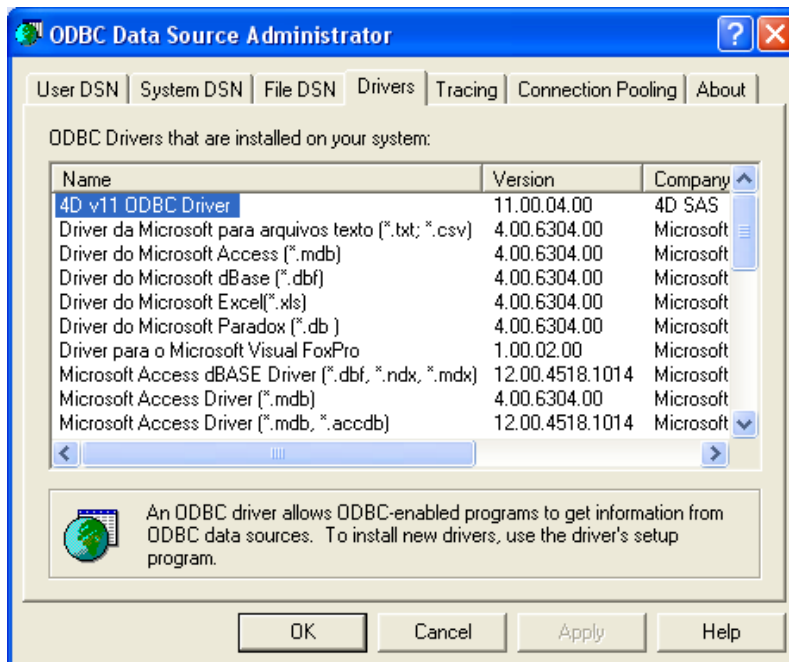
You can connect to the 4D SQL Engine from any external database via the ODBC Driver for 4D.

Note: This configuration is used as an example. It is possible to connect 4D applications directly via SQL. For more information, refer to the description of the **SQL LOGIN** command.

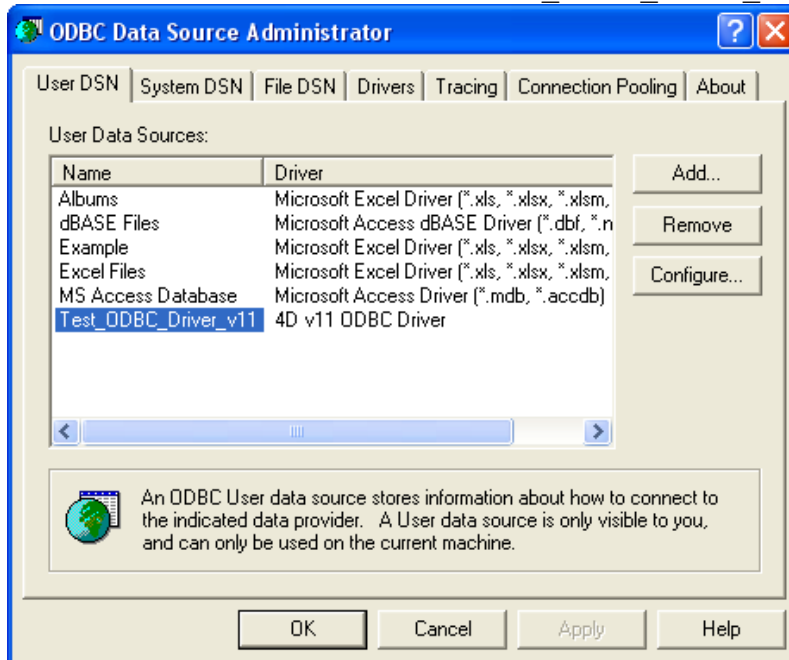
1. Duplicate the example database that comes with this tutorial
2. Rename the two folders containing the databases to "Client" and "Server"
3. Launch the example database inside the Server folder and enable the launching of the SQL Server at startup by checking the "Launch SQL Server at Startup" check-box in the Database Settings, on the SQL page:



4. Quit and restart the example database from the Server folder to activate the SQL Server.
5. Install the 4D ODBC Driver for 4D, then check whether it appears in the ODBC Data Source Administrator:



6. Create a new data source named "Test_ODBC_Driver_v11"



and test it by pressing the **Connection test** button:

4D v11 SQL - Configure data source

Data Source Name : 4D Test

Description : 4D v11 SQL datasource.

Server

Server : localhost

Port : 1919

SSL

Default user

User : Administrateur

Password :

Timeouts

Connection : 3

Login : 3

Query : 30

Connection test Cancel Ok

Data source name [DSN]

A unique name for this Data Source
Default: ""
Optional: No

7. Launch the example database inside the Client folder, go to the main window and press the "Connect to 4D" button. The code behind this button is the following:


```

C_TEXT($CrI_DSN;$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)

  ` By default the current DSN is the local one
  $CrI_DSN:=Get current data source
  ALERT("The current DSN is "+$CrI_DSN)

  ` Do something on the local database
  Begin SQL
    SELECT Title, Director, Media
    FROM MOVIES
    ORDER BY 1
    INTO :aTitles, :aDirectors, :aMedias;
  End SQL

  ` Get the data sources of the User type defined in the ODBC manager
  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)

    ` Establish a connection between 4D and another 4D database via the ODBC Driver v11
    SQL LOGIN($My_4D_DSN,"Administrator",";")
    if (Ok=1)

      ` The current DSN is the 4D one
      $CrI_DSN:=Get current data source
      ALERT("The current DSN is "+$CrI_DSN)

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

      ` Do something on the external (4D) database
      Begin SQL
        SELECT Title, Director, Media
        FROM MOVIES
        ORDER BY 1
        INTO :aTitles, :aDirectors, :aMedias;
      End SQL







      ` Close the external connecton opened with the SQL LOGIN command
      SQL LOGOUT
      ` The current DSN becomes the local one
      $CrI_DSN:=Get current data source
      ALERT("The current DSN is "+$CrI_DSN)
    Else
      ALERT("Unable to connect to the external data source")
    End if
  Else
    ALERT("ODBC Driver data source not found")
  End if

```

As you can see, in the first part of the method we make a query on the local database. Then, in the second part, we connect to the other 4D database via the ODBC driver and make the same query. The result should be the same of course.

Using SQL in 4D

This section provides a general overview of the use of SQL in 4D. It describes how to access the integrated SQL engine, as well as the different ways of sending queries and retrieving data. It also details the configuration of the 4D SQL server and outlines the principles for integrating 4D and its SQL engine.

-  Accessing the 4D SQL Engine
-  Configuration of 4D SQL Server
-  4D SQL engine implementation
-  System Tables
-  Replication via SQL
-  Support of joins

🔌 Accessing the 4D SQL Engine

Sending Queries to the 4D SQL Engine

The 4D built-in SQL engine can be called in three different ways:

- Using the **QUERY BY SQL** command. Simply pass the **WHERE** clause of an SQL *SELECT* statement as a *query* parameter. Example:

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

- Using the integrated SQL commands of 4D, found in the "SQL" theme (**SQL SET PARAMETER**, **SQL EXECUTE**, etc.). These commands can work with an ODBC data source or the 4D SQL engine of the current database.
- Using the standard Method editor of 4D. SQL statements can be written directly in the standard 4D Method editor. You simply need to insert the SQL query between the tags: **Begin SQL** and **End SQL**. The code placed between these tags will not be parsed by the 4D interpreter and will be executed by the SQL engine (or by another engine, if set by the **SQL LOGIN** command).

Passing Data Between 4D and the SQL Engine

Referencing 4D Expressions

It is possible to reference any type of valid 4D expression (variable, field, array, expression...) within **WHERE** and **INTO** clauses of SQL expressions. To indicate a 4D reference, you can use either of the following notations:

- Place the reference between double less-than and greater-than symbols as shown here "<<" and ">>"
- Place a colon ":" in front of the reference.

Examples:

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

or:

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

Note: The use of brackets [] is required when you work with interprocess variables (for example, <<[<>myvar]>> or :[<>myvar]).

Lokale Variablen im kompilierten Modus verwenden

Im kompilierten Modus können Sie unter bestimmten Bedingungen in SQL Statements lokale Variablenreferenzen (mit vorangestelltem Zeichen \$) verwenden:

- Sie können lokale Variablen innerhalb einer Sequenz **Begin SQL / End SQL** einsetzen, außer für den Befehl **EXECUTE IMMEDIATE**;
- Sie können lokale Variablen mit dem Befehl **SQL EXECUTE** einsetzen, wenn diese Variablen direkt im Parameter der SQL Anfrage und nicht über Referenzen verwendet werden. So funktioniert z.B. folgender Code im kompilierten Modus:

```
SQL EXECUTE("select * from t1 into :$myvar") // funktioniert im kompilierten Modus
```

während folgender Code einen Fehler im kompilierten Modus generiert:

```
C_TEXT(tRequest)
tRequest:="select * from t1 into :$myvar"
SQL EXECUTE(tRequest) // Fehler im kompilierten Modus
```

Retrieving Data from SQL Requests into 4D

The data retrieval in a **SELECT** statement will be managed either inside **Begin SQL/End SQL** tags using the **INTO** clause of the **SELECT** command or by the "SQL" language commands.

- In the case of **Begin SQL/End SQL** tags, you can use the **INTO** clause in the SQL query and refer to any valid 4D expression (field, variable, array) to get the value:

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

- With the **SQL EXECUTE** command, you can also use the additional parameters:

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

The main difference between these two ways of getting data from SQL (**Begin SQL/End SQL** tags and SQL commands) is that in the first case all the information is sent back to 4D in one step, while in the second case the records must be loaded explicitly using **SQL LOAD RECORD**.

For example, supposing that in the PEOPLE table there are 100 records:

- Using 4D generic SQL commands:

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

Here we have to loop 10 times to retrieve all 100 records. If we want to load all the records in one step we should use:

```
SQL LOAD RECORD(SQL_all records)
```

- Using **Begin SQL/End SQL** tags:

```
ARRAY INTEGER(aBirthYear;0)
C_TEXT(vName)
```

```
vName:="Smith"
```

```
Begin SQL
```

```
SELECT Birth_Year FROM PERSONS WHERE ename= <<vName>> INTO <<aBirthYear>>
```

```
End SQL
```

In this situation, after the execution of the *SELECT* statement, the aBirthYear array size becomes 100 and its elements are filled with all the birth years from all 100 records.

If, instead of an array, we want to store the retrieved data in a column (i.e., a 4D field), then 4D will automatically create as many records as necessary to save all the data. In our preceding example, supposing that in the PEOPLE table there are 100 records:

- Using 4D generic SQL commands:

```
C_TEXT(vName)
```

```
vName:="Smith"
```

```
$SQLStm:="SELECT Birth_Year FROM PERSONS WHERE ename= <<vName>>"
```

```
SQL EXECUTE($SQLStm;[MYTABLE]Birth_Year)
```

```
While(Not(SQL End selection))
```

```
SQL LOAD RECORD(10)
```

```
End while
```

Here we have to loop 10 times to retrieve all the 100 records. Every step will create 10 records in the [MYTABLE] table and store each retrieved Birth_Year value from the PEOPLE table in the Birth_Year field.

- Using **Begin SQL/End SQL** tags:

```
C_TEXT(vName)
```

```
vName:="Smith"
```

```
Begin SQL
```

```
SELECT Birth_Year FROM PERSONS WHERE ename= <<vName>> INTO
```

```
<<[MYTABLE]Birth_Year>>
```

```
End SQL
```

In this case, during the execution of the *SELECT* statement, there will be 100 records created in the [MYTABLE] table and each Birth_Year field will contain the corresponding data from the PEOPLE table, Birth_Year column.

Using a Listbox

4D includes a specific automatic functioning (**LISTBOX** keyword) that can be used for placing data from *SELECT* queries into a listbox. For more information, please refer to the Design Reference manual.

Optimization of Queries

For optimization purposes, it is preferable to use 4D expressions rather than SQL functions in queries. 4D expressions will be calculated once before the execution of the query whereas SQL functions are evaluated for each record found.

For example, with the following statement:

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

... the vLastName+vFirstName expression is calculated once, before query execution. With the following statement:

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

... the **CONCAT(<<vLastName>>,<<vFirstName>>)** function is called for each record of the table; in other words, the expression is evaluated for each record.

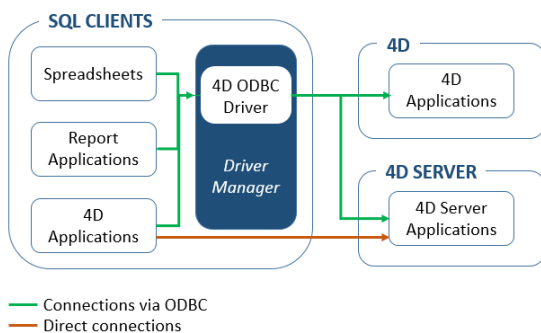
🔧 Configuration of 4D SQL Server

The SQL server of 4D allows external access to data stored in the 4D database. For third-party applications and 4D applications, this access is carried out using a 4D ODBC driver. It is also possible to make direct connections between a 4D client and 4D Server application. All connections are made using the TCP/IP protocol.

The SQL server of a 4D application can be stopped or started at any time. Moreover, for performance and security reasons, you can specify the TCP port as well as the listening IP address, and restrict access possibilities to the 4D database.

External Access to SQL Server

External access to the 4D SQL server can be made either via ODBC (all configurations), or directly (4D client application connected to 4D Server). This is summarized in the following diagram:



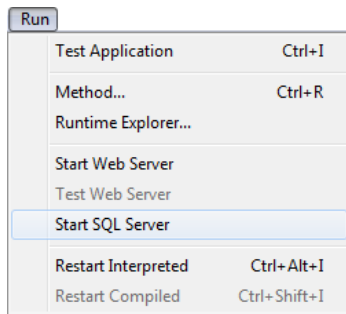
At the query level, opening a direct external connection or a connection via ODBC is carried out using the **SQL LOGIN** command. For more information, please refer to the description of this command.

- **Connections via ODBC:** 4D provides an ODBC driver that allows any third-party application (Excel® type spreadsheet, other DBMS, and so on) or another 4D application to connection to the 4D SQL server. The 4D ODBC driver must be installed on the SQL Client machine. The installation and configuration of the 4D ODBC driver is detailed in a separate manual.
- **Direct connections:** Only a 4D Server application can reply to direct SQL queries coming from other 4D applications. Similarly, only 4D applications of the "Professional" product line can open a direct connection to another 4D application. During a direct connection, data exchange is automatically carried out in synchronous mode, which eliminates questions related to synchronization and data integrity. Only one connection is authorized per process. If you want to establish several simultaneous connections, you must create as many processes as needed. Direct connections can be secured by selecting the **Enable TLS** option on the target side of the connection (4D Server) on the "SQL" tab of the Database Settings. Direct connections are only authorized by 4D Server if the SQL server is started. The main advantage of direct connections is that data exchanges are accelerated.

Starting and Stopping the 4D SQL Server

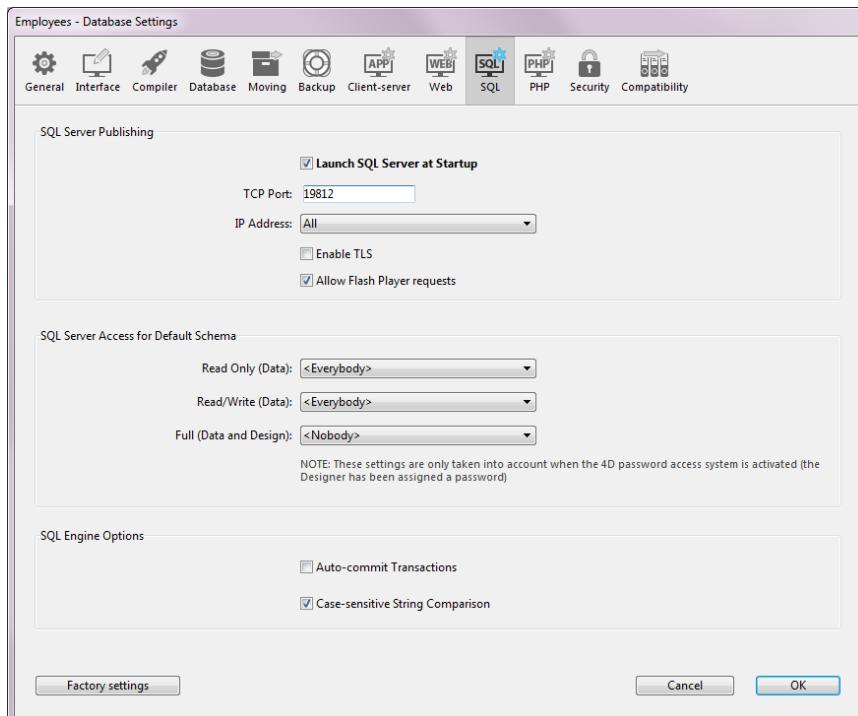
The SQL server can be started and stopped in three ways:

- Manually, using the **Start SQL Server/Stop SQL Server** commands in the **Run** menu of the 4D application:



Note: With 4D Server, this command can be accessed as a button on the **SQL Server Page**. When the server is launched, this menu item changes to **Stop SQL Server**.

- Automatically on startup of the application, via the Database Settings. To do this, display the **SQL** page and check the **Launch SQL Server at Startup** option:



- By programming, using the **START SQL SERVER** and **STOP SQL SERVER** commands ("SQL" theme). When the SQL server is stopped (or when it has not been started), 4D will not respond to any external SQL queries.

Note: Stopping the SQL server does not affect the internal functioning of the 4D SQL engine. The SQL engine is always available for internal queries.

SQL Server Publishing Preferences

It is possible to configure the publishing parameters for the SQL server integrated into 4D. These parameters are found on the **SQL** page of the Database Settings:

- The **Launch SQL Server at Startup** option can be used to start the SQL server on application startup.
- TCP Port:** By default, the 4D SQL server responds on the TCP port 19812. If this port is already being used by another service, or if your connection parameters require another configuration, you can change the TCP port used by the 4D SQL server.

Notes:

- If you pass 0, 4D will use the default TCP port number, i.e. 19812.
- You can programmatically set this value by using the **SQL Server Port ID** selector of the **SET DATABASE PARAMETER** command.

- **IP Address:** You can set the IP address of the machine on which the SQL server must process SQL queries. By default, the server will respond to all the IP addresses (**All** option).

The "IP Address" drop-down list automatically contains all the IP addresses present on the machine. When you select a particular address, the server will only respond to queries sent to this address.

This is intended for 4D applications hosted on machines having several TCP/IP addresses.

Notes:

- On the client side, the IP address and the TCP port of the SQL server to which the application connects must be correctly configured in the ODBC data source definition.
- Starting with 4D v14, the SQL server handles IPv6 address notation. The server accepts either IPv6 or IPv4 connections indiscriminately when the listening "IP address" of the server is set to **All**. For more information, refer to [Unterstützung von IPv6](#).

- **Enable TLS:** This option indicates whether the SQL server must enable the TLS protocol for processing SQL connections. Note that when this protocol is enabled, you must add the ":ssl" keyword to the end of the IP address of the SQL server when you open a connection using the **SQL LOGIN** command.

By default, the SQL server uses internal files for the TLS key and certificate. You can, however, use custom elements: to do this, just copy your own *key.pem* and *cert.pem* files to the following location: MyDatabase/Preferences/SQL (where "MyDatabase" represents the database folder/package).

Notes:

- Beginning with 4D v16 R5, the default protocol is TLS 1.2.
- You can programmatically modify this value by using the [Min TLS version](#) selector with the **SET DATABASE PARAMETER** command.

- **Allow Flash Player requests:** This option can be used to enable the mechanism for supporting Flash Player requests by the 4D SQL server. This mechanism is based on the presence of a file, named "socketpolicy.xml," in the preferences folder of the database (Preferences/SQL/Flash/). This file is required by Flash Player in order to allow cross-domain connections or connections by sockets of Flex (Web 2.0) applications.

In the previous version of 4D, this file had to be added manually. From now on, the activation is carried out using the **Allow Flash Player requests** option: When you check this option, Flash Player requests are accepted and a generic "socketpolicy.xml" file is created for the database if necessary.

When you deselect this option, the "socketpolicy.xml" file is disabled (renamed). Any Flash Player queries received subsequently by the SQL server are then rejected.

On opening of the database, the option is checked or not checked depending on the presence of an active "socketpolicy.xml" file in the preferences folder of the database.

Note: It is possible to set the encoding used by the SQL server for processing external requests using the 4D **SQL SET OPTION** command.

SQL Access Control for the default schema

For security reasons, it is possible to limit actions that external queries sent to the SQL server can perform in the 4D database.

This can be done at two levels:

- At the level of the type of action allowed,
 - At the level of the user carrying out the query.
- These settings can be made on the **SQL** page of the Database Settings.

Note: You can also use the **Datenbankmethode On SQL Authentication** to control in a custom way any external access to the 4D internal SQL engine.

The parameters set in this dialog box are applied to the default schema. The control of external access to the database is based on the concept of SQL schemas (see the section). If you do not create custom schemas, the default schema will include all the tables of the database. If you create other schemas with specific access rights and associate them with tables, the default schema will only include the tables that are not included in custom schemas.

You can configure three separate types of access to the default schema via the SQL server:

- **“Read Only (Data)”**: Unlimited access to read all the data of the database tables but no adding, modifying or removing of records, nor any modification to the structure of the database is allowed.
- **“Read/Write (Data)”**: Read and write (add, modify and delete) access to all the data of the database tables, but no modification of the database structure is allowed.
- **“Full (Data and Design)”**: Read and write (add, modify and delete) access to all the data of the database tables, as well as modification of the database structure (tables, fields, relations, etc.) is allowed.

You can designate a set of users for each type of access. There are three options available for this purpose:

- **<Nobody>**: If you select this option, the type of access concerned will be refused for any queries, regardless of their origin. This parameter can be used even when the 4D password access management system is not activated.
- **<Everybody>**: If you select this option, the type of access concerned will be allowed for all queries (no limit is applied).
- **Group of users**: This option lets you designate a group of users as exclusively authorized to carry out the type of access concerned. This option requires that 4D passwords be activated. The user at the origin of the queries provides their name and password when connecting to the SQL server.

WARNING: Each type of access is set independently from the others. More specifically, if you only assign **Read Only** type access to one group this will not have any effect since this group as well as all the others will continue to benefit from **Read/Write** access (assigned to **<Everybody>** by default). In order to set a **Read Only** type access, you also need to configure the **Read/Write** access.

WARNING: This mechanism is based on 4D passwords. In order for the SQL server access control to come into effect, the 4D password system must be activated (a password must be assigned to the Designer).

Note: An additional security option can be set at the level of each 4D project method. For more information, please refer to the "Available through SQL option" paragraph in the section.

4D SQL engine implementation

Basically, the 4D SQL engine is SQL-92 compliant. This means that for a detailed description of commands, functions, operators or the syntax to be used, you may refer to any SQL-92 reference. These can be found, for instance, on the Internet.

However, the 4D SQL engine does not support 100% of the SQL-92 features and also provides some specific additional features.

This section covers the main implementations and limitations of the 4D SQL engine.

General Limitations

Since the SQL engine of 4D has been integrated into the heart of the 4D database, all the limitations concerning the maximum number of tables, columns (fields) and records per database, as well as the rules for naming tables and columns, are the same as for the standard internal 4D engine (DB4D). They are listed below.

- Maximum number of tables: Theoretically two billion but for compatibility reasons with 4D: 32767.
- Maximum number of columns (fields) in a table: Theoretically two billion columns (fields), but for compatibility reasons with 4D: 32767.
- Maximum number of rows (records) in a table: one billion.
- Maximum number of index keys: 128 billions for alpha, text, and float indexes; 256 billions for other index types (scalar data).
- A primary key cannot be a NULL value and must be unique. It is not necessary to index the primary key columns (fields).
- Maximum number of characters allowed for the table and field names: 31 characters (4D limitation).

Tables with the same name created by different users are not allowed. The standard 4D control mechanism will be applied.

Data Types

The following table indicates the data types supported in 4D SQL and their corresponding type in 4D:

4D SQL	Description	4D
Varchar	Alphanumeric text	Text or Alpha
Real	Floating point number in the range of +/-1.7E308	Real
Numeric	Number between +/- 2E64	Integer 64 bits
Float	Floating point number (virtually infinite)	Float
Smallint	Number between -32 768 and 32 767	Integer
Int	Number between -2 147 483 648 and 2 147 483 647	Longint, Integer
Int64	Number between +/- 2E64	Integer 64 bits
UUID	16-byte number (128 bits) represented by 32 hexadecimal characters	UUID Alpha format
Bit	A field that can only take the values TRUE/FALSE or 1/0	Boolean
Boolean	A field that can only take the values TRUE/FALSE or 1/0	Boolean
Blob	Up to 2 GB; any binary object such as a graphic, another application, or any document	Blob
Bit varying	Up to 2 GB; any binary object such as a graphic, another application, or any document	Blob
Clob	Text up to 2 GB characters. This column (field) cannot be indexed. It is not saved in the record itself.	Text
Text	Text up to 2 GB characters. This column (field) cannot be indexed. It is not saved in the record itself.	Text
Timestamp	Date&Time, Date in 'YYYY/MM/DD' format and Time in 'HH:MM:SS:ZZ' format	Date and Time parts handled separately (automatic conversion)
Duration	Time in 'HH:MM:SS:ZZ' format	Time
Interval	Time in 'HH:MM:SS:ZZ' format	Time
Picture	PICT picture up to 2 GB	Picture

Automatic data type conversion is implemented between numeric types.

A string that represents a number is not converted to a corresponding number. There are special *CAST* functions that will convert values from one type to another.

The following SQL data types are not implemented:

- NCHAR
- NCHAR VARYING.

NULL Values in 4D

The NULL values are implemented in the 4D SQL language as well as in the 4D database engine. However, they are not supported in the 4D language. It is nevertheless possible to read and write NULL values in a 4D field using the **Is field value Null** and **SET FIELD VALUE NULL** commands.

Compatibility of Processing and Map NULL Values to Blank Values Option

For compatibility reasons in 4D, NULL values stored in 4D database tables are automatically converted into default values when being manipulated via the 4D language. For example, in the case of the following statement:

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

... if the MyAlphafield field contains a NULL value, the myAlphavar variable will contain "" (empty string).

The default values depend on the data type:

- For Alpha and Text data types: ""
- For Real, Integer and Long Integer data types: 0

- For the Date data type: "00/00/00"
- For the Time data type: "00:00:00"
- For the Boolean data type: False
- For the Picture data type: Empty picture
- For the Blob data type: Empty blob

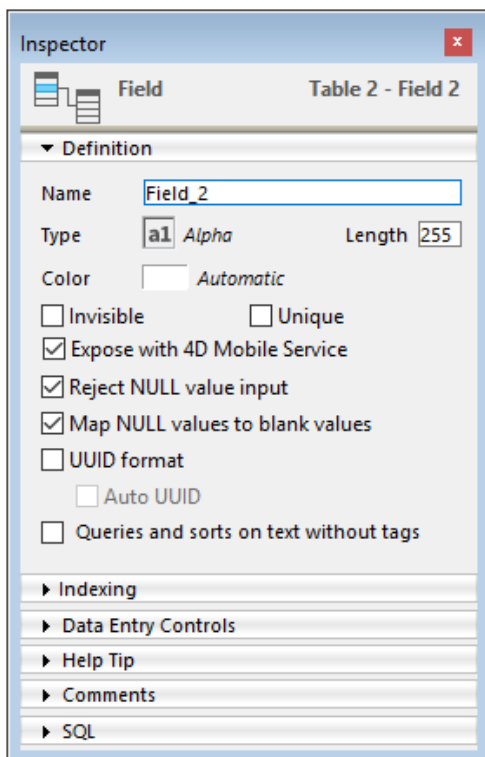
On the other hand, this mechanism in principle does not apply to processing carried out at the level of the 4D database engine, such as queries. In fact, searching for an "blank" value (for example myvalue=0) will not find records storing the NULL value, and vice versa. When both types of values (default values and NULL) are present in the records for the same field, some processing may be altered or require additional code.

To avoid these inconveniences, an option can be used to standardize all the processing in the 4D language: **Map NULL values to blank values**. This option, which is found in the field Inspector window of the Structure editor, is used to extend the principle of using default values to all processing. Fields containing NULL values will be systematically considered as containing default values. This option is checked by default.

The **Map NULL values to blank values** property is taken into account at a very low level of the database engine. It acts more particularly on the **Is field value Null** command.

Reject NULL Value Input Attribute

The **Reject NULL value input** field property is used to prevent the storage of NULL values:



When this attribute is checked for a field, it will not be possible to store a NULL value in this field. This low-level property corresponds exactly to the NOT NULL attribute of SQL.

Generally, if you want to be able to use NULL values in your 4D database, it is recommended to exclusively use the SQL language of 4D.

Note: In 4D, fields can also have the "Mandatory" attribute. The two concepts are similar but their scope is different: the "Mandatory" attribute is a data entry control, whereas the "Reject NULL value input" attribute works at the level of the database engine.

If a field having this attribute receives a NULL value, an error will be generated.

Date and time expressions

Date and time constants

The integrated SQL server of 4D supports date and time constants in accordance with the ODBC API. Here is the syntax for sequences of ODBC date and time constants:

```
{constant_type 'value'}
```

constant_type	value	Description
d	yyyy-mm-dd	Date only
t	hh:mm:ss[.fff]	Time only
ts	yyyy-mm-dd hh:mm:ss[.fff]	Date and time (timestamp)

Note: *fff* indicates milliseconds.

For example, you can use the following constants:

```
{ d '2013-10-02' }  
{ t '13:33:41' }  
{ ts '1998-05-02 01:23:56.123' }
```

Queries on blank dates

The SQL date parser rejects any date expression specifying "0" as the day or month. Expressions such as {d'0000-00-00'} or CAST('0000-00-00' AS TIMESTAMP) generate an error. To perform SQL queries on blank dates (not to be confused with null dates), you must use an intermediate 4D expression. For example:

```
C_LONGINT($count)  
$nullDate:=!00-00-00!  
Begin SQL  
    SELECT COUNT(*) FROM Table_1  
    WHERE myDate = :$nullDate  
    INTO :$count;  
End SQL
```

"Available through SQL" Option

The "Available via SQL" property, available for project methods, allows you to control the execution of 4D project methods via SQL.

Method Properties

Name:

Invisible

Shared by components and host database

Execute on Server

Execution mode: Can be run in preemptive processes
Only used in compiled databases Cannot be run in preemptive processes
 Indifferent

Available through: Web Services
 Published in WSDL
 4D tags and URLs (4DACTION...)
 SQL
 4D Mobile

Table:
Scope:

Access group:

Owner group:

When checked, this option allows the execution of the project method by the 4D SQL engine. It is not selected by default, which means that 4D project methods are protected and cannot be called by the 4D SQL engine unless they have been explicitly authorized by checking this option.

This property applies to all SQL queries, both internal and external — whether executed via the ODBC driver, or via SQL code inserted between the **Begin SQL/End SQL** tags, or via the **QUERY BY SQL** command.

Notes:

- Even when a method is given the "Available through SQL" attribute, the access rights set at the Database Settings level and at the level of the method properties are nevertheless taken into account when it is executed.
- The ODBC SQLProcedure function only returns project methods having the "Available through SQL" attribute.

SQL Engine Optionen

Hier gibt es zwei Optionen, um die Funktionsweise der 4D SQL Engine zu verändern.

SQL Engine Optionen

Auto-commit Transactions

Groß-/Kleinschreibung bei Textvergleich beachten

- **Auto-commit Transactions:** Mit dieser Option können Sie Auto commit in der 4D SQL Engine aktivieren. Dieser Modus sorgt für referentielle Datenintegrität. Ist die Option aktiv, werden alle Suchläufe mit **SELECT, INSERT, UPDATE** und **DELETE** (SIUD), die noch nicht innerhalb einer Transaktion ausgeführt werden, automatisch in eine ad-hoc Transaktion eingesetzt. So ist garantiert, dass die Suchen vollständig ausgeführt oder bei einem Fehler vollständig annulliert werden. Suchen, die bereits in einer Transaktion laufen, werden dabei nicht berücksichtigt (individuelle Verwaltung der Datenintegrität).

Ist diese Option nicht markiert, werden keine automatischen Transaktionen erzeugt. Davon ausgenommen sind Suchläufe mit **SELECT... FOR UPDATE** (siehe SQL Befehl *SELECT*). Diese Option ist standardmäßig nicht markiert.

Über den 4D Befehl **SET DATABASE PARAMETER** können Sie die Option auch per Programmierung verwalten.

Hinweis: Auto commit gilt nur für lokale Datenbanken, die über die 4D SQL Engine durchsucht werden. Bei externen Datenbanken wird **Auto commit** über remote SQL Engines verwaltet

- **Groß-/Kleinschreibung bei Textvergleich beachten:** Mit dieser Option können Sie die Berücksichtigung von Groß-/Kleinschreibung und Akzenten in SQL Anfragen verändern. Diese Option ist standardmäßig aktiviert, d.h. die SQL Engine unterscheidet beim Vergleichen von Strings zwischen Groß- und Kleinschreibung und Akzenten. Das gilt insbesondere für Sortieren und Suchen. Beispiel: "ABC"="ABC" aber "ABC" # "Abc und abc # âbc."

In bestimmten Fällen, z.B. um die Funktionsweise der SQL Engine an die 4D Engine anzupassen, sollen beim Vergleichen von Strings die Groß- und Kleinschreibung und Akzente nicht berücksichtigt werden ("ABC"="Abc"="âbc"). Dazu deaktivieren Sie einfach diese Option in den Datenbank-Eigenschaften.

Über den 4D Befehl **SET DATABASE PARAMETER** können Sie die Option auch per Programmierung verwalten.

Schemas

4D implements the concept of schemas. A schema is a virtual object containing the tables of the database. In SQL, the purpose of schemas is to assign specific access rights to different sets of database objects. Schemas divide the database into independent entities which together make up the entire database. In other words, a table always belongs to one and only one schema.

- To create a schema, you must use the *CREATE SCHEMA* command. You can then use the *GRANT* and *REVOKE* commands to configure the types of access to the schemas.
- To associate a table with a schema, you can call the *CREATE TABLE* or *ALTER TABLE* commands. You can also use the "Schemas" pop-up menu of the **Inspector palette** in the Structure editor of 4D. This menu lists all the schemas defined in the database.
- The *DROP SCHEMA* command can be used to delete a schema.

Note: The control of access via schemas only applies to connections from the outside. The SQL code executed within 4D via **Begin SQL/End SQL** tags, **SQL EXECUTE**, **QUERY BY SQL**, and so on, always has full access.

Connections to SQL sources

Multi-database architecture is implemented at the level of the 4D SQL server. From within 4D it is possible:

- To connect to an existing database using the **SQL LOGIN** command.
- To switch from one database to another using the 4D **SQL LOGIN** and **SQL LOGOUT** commands.
- To open and use another 4D database instead of current database using the **USE DATABASE** command.

Primary key

In the SQL language, a primary key is used to identify the table column(s) (field(s)) responsible for uniquely specifying the table records (rows). In particular, setting a primary key is required for the record replication function in a 4D table (see **Replication via SQL**) and for logging 4D tables as of v14.

4D allows you to manage primary keys in two ways:

- Via the SQL language
- Using the 4D Structure editor.

Note:

- You can also set primary keys using the 4D **Primary key manager** in the Design mode.
- For a description of the use of primary keys, see the **Rules for using primary key fields**.

Setting the primary key via the SQL language

You can set a primary key when a table is created (via the *CREATE TABLE* command) or when adding or modifying a column (via the *ALTER TABLE* command). The primary key is specified using the PRIMARY KEY clause followed by the column name or a list of columns. For more information, refer to the section.

Setting the primary key via the structure editor

4D lets you create and remove primary keys directly via the context menu of the structure editor. For more information about this point, refer to **Primary keys** in the 4D *Design Reference* manual.

SQL views

The integrated SQL engine of 4D supports standard **SQL views**. A view is a virtual table with data that may come from several different database tables. Once a view is defined, you can use it in a **SELECT** statement just like a real table.

Data found in a view are defined using a definition query based on the **SELECT** command. Real tables used in the definition query are called "source tables". An SQL view contains columns and rows just like a standard table, but it does not actually exist; it is only a representation resulting from processing and stored in memory during the session. Only the definition of the view is actually saved temporarily.

Two SQL commands are used to manage views in 4D v14: **CREATE VIEW** and **DROP VIEW**.

System Tables

The SQL catalogue of 4D includes several system tables, which can be accessed by any SQL user having read access rights: `_USER_TABLES`, `_USER_COLUMNS`, `_USER_INDEXES`, `_USER_CONSTRAINTS`, `_USER_IND_COLUMNS`, `_USER_CONS_COLUMNS`, `_USER_SCHEMAS`, `_USER_VIEWS` and `_USER_VIEW_COLUMNS`.

In accordance with the customs of SQL, system tables describe the database structure. Here is a description of these tables and their fields:

<code>_USER_TABLES</code>		Describes the user tables of the database
<code>TABLE_NAME</code>	VARCHAR	Table name
<code>TEMPORARY</code>	BOOLEAN	True if the table is temporary; otherwise, false
<code>TABLE_ID</code>	INT64	Table number
<code>SCHEMA_ID</code>	INT32	Number of schema
<code>REST_AVAILABLE</code>	BOOLEAN	True if column is exposed with REST service; otherwise, False
<code>LOGGED</code>	BOOLEAN	True if table operations are included in log file; otherwise, False

<code>_USER_COLUMNS</code>		Describes the columns of the user tables of the database
<code>TABLE_NAME</code>	VARCHAR	Table name
<code>COLUMN_NAME</code>	VARCHAR	Column name
<code>DATA_TYPE</code>	INT32	Column type
<code>DATA_LENGTH</code>	INT32	Column length
<code>OLD_DATA_TYPE</code>	INT32	4D field type (see the Type command)
<code>NULLABLE</code>	BOOLEAN	True if column accepts NULL values; otherwise, false
<code>TABLE_ID</code>	INT64	Table number
<code>COLUMN_ID</code>	INT64	Column number
<code>UNIQUENESS</code>	BOOLEAN	True if column is declared Unique; otherwise, False
<code>AUTOGENERATE</code>	BOOLEAN	True if column value is generated automatically for each new record; otherwise, False
<code>AUTOINCREMENT</code>	BOOLEAN	True if column value is incremented automatically; otherwise, False
<code>REST_AVAILABLE</code>	BOOLEAN	True if column is exposed with REST service; otherwise, False

<code>_USER_INDEXES</code>		Describes the user indexes of the database
<code>INDEX_ID</code>	VARCHAR	Index number
<code>INDEX_NAME</code>	VARCHAR	Index name
<code>INDEX_TYPE</code>	INT32	Index type (1=BTree / Composite, 3=Cluster / Keyword, 7=Auto, 8=Auto for Object type field)
<code>KEYWORD</code>	BOOLEAN	True if index is a keyword index; otherwise, False
<code>TABLE_NAME</code>	VARCHAR	Name of table with index
<code>UNIQUENESS</code>	BOOLEAN	True if index imposes a uniqueness constraint; otherwise, false
<code>TABLE_ID</code>	INT64	Number of table with index

_USER_VIEW_COLUMNS

Describes the columns of the views of the database users

VIEW_NAME	VARCHAR	Name of view
COLUMN_NAME	VARCHAR	Name of column
DATA_TYPE	INT32	Type of column
DATA_LENGTH	INT32	Size of column
NULLABLE	BOOLEAN	True if column accepts NULL values; otherwise, False

Note: The system tables are assigned to a specific schema named **SYSTEM_SCHEMA**. This schema cannot be modified or deleted. It does not appear in the list of schemas displayed in the table Inspector palette. It can be accessed in read-only by any user.

🌱 Replication via SQL

4D provides a mechanism that allows data to be replicated or synchronized between two or more 4D databases via SQL. This specific functionality can be used to set up one or more mirror databases, guaranteeing permanent availability of data.

The principle is as follows: a target database replicates the data of a remote source database locally. Updates are carried out periodically by the local database which retrieves the data from the remote database. Replication is carried out at the table level: you replicate the data of a remote database table into a table in the local database.

This is made possible by the use of stamps and specific SQL commands.

In the Structure editor, the replication mechanism is enabled via a table property in both the remote and local database. On the local side, the SQL **REPLICATE** command lets you retrieve data from a table in the remote database and then integrate this data into a table of the local database. As for the SQL **SYNCHRONIZE** command, it is used to carry out the synchronization of two tables.

Virtual fields

Each table of the 4D database can be assigned three "virtual" fields: `__ROW_ID`, `__ROW_STAMP` and `__ROW_ACTION`. These fields are called "virtual" to differentiate them from "standard" fields because they have specific properties: they are automatically filled in, can be read but not modified by the users, and do not appear in the system tables of the database. The following table describes these fields as well as their mode of use:

Virtual field	Type	Content	Use
<code>__ROW_ID</code>	Int32	ID of record	In any SQL statement except for REPLICATE or SYNCHRONIZE
<code>__ROW_STAMP</code>	Int64	Record replication information	In any SQL statement
<code>__ROW_ACTION</code>	Int16	Type of action carried out on the record: 1 = Addition or modification, 2 = Deletion	Only with the REPLICATE or SYNCHRONIZE command

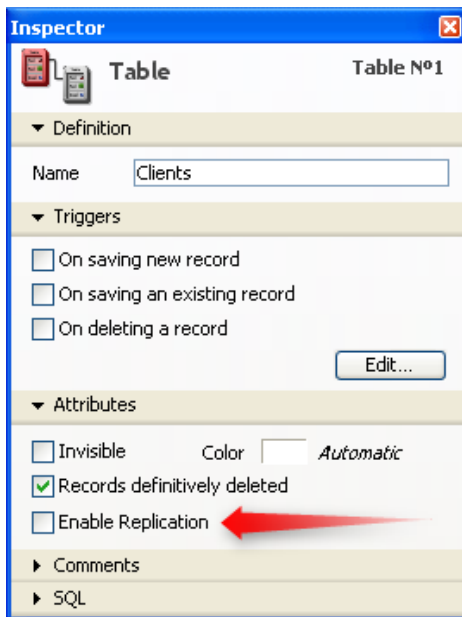
When the replication mechanisms are enabled, as soon as a record is created, modified or deleted, the corresponding information is automatically updated in the virtual fields of this record.

Enabling replication

By default the mechanisms that allow replication are not enabled. You must explicitly enable them in both the remote and local database for each table to be used in the replication or synchronization.

Please note that enabling the mechanism does not trigger the replication itself; in order for the data to actually be replicated in a local or synchronized database, you must use the **REPLICATE** or **SYNCHRONIZE** commands.

To enable the internal replication mechanism, for each table (on the remote and local database) you must use the **Enable Replication** table property that is found in the table Inspector:



Note: In order for the replication mechanism to be able to function, you must specify a primary key for the tables implicated in the remote and local databases. You can create this key via the structure editor or using SQL commands. If no primary key has been specified, the option is grayed out.

When this option is checked, 4D generates the information necessary for replicating the records of the table (based more particularly on the primary key of the table). This information is stored in the virtual `__ROW_STAMP` and `__ROW_ACTION` fields.

Note: It is possible to enable and disable the generation of replication information via the SQL `CREATE TABLE` and `ALTER TABLE` commands, using the `ENABLE REPLICATE` and `DISABLE REPLICATE` keywords. For more information, please refer to the description of these commands.

WARNING: Checking this option causes information needed for replication mechanisms to be published. For security reasons, you must protect access to this information -- just as you protect access to your data when it is published. As a result, when you implement a replication system using this option, you must make sure that:

- if the SQL server is launched, access is protected using 4D passwords and/or SQL schemas (see **Configuration of 4D SQL Server**),
- if the HTTP server is launched, access is protected using 4D passwords and/or SQL schemas (see **Configuration of 4D SQL Server**) and/or the **On Web Authentication Database Method** and/or defining a virtual structure using the **SET TABLE TITLES** and **SET FIELD TITLES** commands. For more information, refer to the "URL 4DSYNC/" paragraph in the **QR Get drop column** section.

Update on local database side

Once the replication mechanism is enabled in the each table of each database, you can use it from the local database via the SQL **REPLICATE** command. For more information, please refer to the description of this command.

🔧 Support of joins

The SQL engine of 4D extends the support of joins.

Join operations may be inner or outer, implicit or explicit. Implicit inner joins are supported via the *SELECT* command. You can also generate explicit inner and outer joins using the SQL **JOIN** keyword.

Note: The current implementation of joins in the 4D SQL engine does not include:

- natural joins.
- the USING construct on inner joins.
- cross inner joins.

Overview

Join operations are used to make connections between the records of two or more tables and combine the result in a new table, called a join.

You generate joins via *SELECT* statements that specify the join conditions.

Starting with 4D v15 R4, outer joins involving two tables and outer joins involving three or more tables are different implementations and do not follow the same rules. Please refer below to the section that correspond to your needs.

Note: Usually, in the database engine, the table order is determined by the order specified during the search. However, when you use joins, the order of the tables is determined by the list of tables. In the following example:

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

... the order of the tables is T1 then T2 (as they appear in the list of tables) and not T2 then T1 (as they appear in the join condition).

Example database

To illustrate how joins work, we are going to use the following database throughout this section:

- Employees

name	depID	cityID
Alan	10	30
Anne	11	39
Bernard	10	33
Fabrice	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

If you want, you can generate this database automatically by executing the following code:

Begin SQL

```

DROP TABLE IF EXISTS Employees;
CREATE TABLE Employees ( depID INT32, name VARCHAR, cityID INT32);
INSERT INTO Employees (name, depID, cityID) VALUES ('Alan', 10, 30);
INSERT INTO Employees (name, depID, cityID) VALUES ('Anne', 11, 39);
INSERT INTO Employees (name, depID, cityID) VALUES ('Bernard', 10, 33);
INSERT INTO Employees (name, depID, cityID) VALUES ('Fabrice', 12, 35);
INSERT INTO Employees (name, depID, cityID) VALUES ('Martin', 15, 30);
INSERT INTO Employees (name, depID, cityID) VALUES ('Philip', NULL, 33);
INSERT INTO Employees (name, depID, cityID) VALUES ('Thomas', 10, NULL);

DROP TABLE IF EXISTS Departments;
CREATE TABLE Departments ( depID INT32, depName VARCHAR );
INSERT INTO Departments (depID, depName) VALUES (10, 'Program');
INSERT INTO Departments (depID, depName) VALUES (11, 'Engineering');
INSERT INTO Departments (depID, depName) VALUES (NULL, 'Marketing');
INSERT INTO Departments (depID, depName) VALUES (12, 'Development');
INSERT INTO Departments (depID, depName) VALUES (13, 'Quality');

DROP TABLE IF EXISTS Cities;
CREATE TABLE Cities ( cityID INT32, cityName VARCHAR );
INSERT INTO Cities (cityID, cityName) VALUES (30, 'Paris');
INSERT INTO Cities (cityID, cityName) VALUES (33, 'New York');
INSERT INTO Cities (cityID, cityName) VALUES (NULL, 'Berlin');

```

End SQL

Explicit inner joins

An inner join is based on a comparison to find matches between two columns.

Here is an example of an implicit inner join:

```

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

```

In 4D, you can also use the JOIN keyword to specify an explicit inner join:

```

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

```

You can insert this query into 4D code as follows:

```

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

```



```
ARRAY INTEGER(aDepID;0)
```

```
Begin SQL
```

```
SELECT Employees.name, Employees.depID, Departments.depID, Departments.depName  
FROM Employees  
INNER JOIN Departments  
ON Employees.depID = Departments.depID  
INTO :aName, :aEmpDepID, :aDepID, :aDepName;
```

```
End SQL
```

Here are the results of this join:

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

Note that neither the employees named Philip or Martin nor the Marketing or Quality departments appear in the resulting join because:

- Philip does not have a department associated with his name (NULL value),
- The department ID associated with Martin's name does not exist in the Departments table,
- There is no employee associated with the Quality department (ID 13),
- The Marketing department does not have an ID associated with it (NULL value).

Outer joins with two tables

You can generate outer joins with 4D. With outer joins, it is not necessary for there to be a match between the rows of joined tables. The resulting table contains all the rows of the tables (or of at least one of the joined tables) even if there are no matching rows. This means that all the information of a table can be used, even if the rows are not completely filled in between the different joined tables.

There are three types of outer joins, specified using the LEFT, RIGHT and FULL keywords. LEFT and RIGHT are used to indicate the table (located to the left or right of the JOIN keyword) where all the data must be processed. FULL indicates a bilateral outer join.

Note: Only explicit outer joins are supported by 4D.

With two-table outer joins, conditions can be complex but they must always be based on an equality comparison between the columns included in the join. For example, it is not possible to use the >= operator in an explicit join condition. Any type of comparison can be used in an implicit join. Internally, equality comparisons are carried out directly by the 4D engine, which ensures rapid execution

Left outer joins

The result of a left outer join (or left join) always contains all the records for the table located to the left of keyword even if the join condition does not find a matching record in the table located to the right. This means that for each row in the left table where the search does not find any matching row in the right table, the join will still contain this row but it will have NULL values in each column of the right table. In other words, a left outer join returns all the rows of the left table plus any of those of the right table that match the join condition (or NULL if none match). Note that if the right table contains more than one row that matches the join predicate for a single row of the left table, the values of the left table will be repeated for each distinct row of the right table.

Here is an example of 4D code with a left outer join:

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

Begin SQL

```
SELECT Employees.name, Employees.depID, Departments.depID, Departments.depName
FROM Employees
LEFT OUTER JOIN Departments
ON Employees.DepID = Departments.DepID
INTO :aName, :aEmpDepID, :aDepID, :aDepName;
```

End SQL

Here is the result of this join with our example database (additional rows shown in red):

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 joins

A right outer join is the exact opposite of a left outer join. Its result always contains all the records of the table located to the right of the JOIN keyword even if the join condition does not find any matching record in the left table.

Here is an example of 4D code with a right outer join:

```
ARRAY TEXT(aName;0)
ARRAY TEXT(aDepName;0)
ARRAY INTEGER(aEmpDepID;0)
ARRAY INTEGER(aDepID;0)
Begin SQL
SELECT Employees.name, Employees.depID, Departments.depID, Departments.depName
FROM Employees
RIGHT OUTER JOIN Departments
ON Employees.DepID = Departments.DepID;
INTO :aName, :aEmpDepID, :aDepID, :aDepName;
End SQL
```

Here is the result of this join with our example database (additional rows shown in red):

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 joins

A full outer join simply combines together the results of a left outer join and a right outer join. The resulting join table contains all the records of the left and right tables and fills in the missing fields on each side with NULL values.

Here is an example of 4D code with a full outer join:

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

```
ARRAY INTEGER(aDepID;0)
```

```
Begin SQL
```

```
SELECT Employees.name, Employees.depID, Departments.depID, Departments.depName  
FROM Employees  
FULL OUTER JOIN Departments  
ON Employees.DepID = Departments.DepID  
INTO :aName, :aEmpDepID, :aDepID, :aDepName;
```

```
End SQL
```

Here is the result of this join with our example database (additional rows shown in red):

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

Outer joins with three or more tables

Starting with 4D v15 R4, the built-in SQL server extends the support of SQL outer joins to queries involving three or more tables. This specific implementation has its own rules and limitations, which are described in this section.

Like two-table outer joins, outer joins with three or more tables can be LEFT, RIGHT, or FULL. For general information on outer joins, please refer to the [Outer joins with two tables](#) paragraph above.

Unlike two-table outer joins, outer joins with three or more tables support several comparison operators, in addition to the equality (=): <, >, >=, or <=. These operators can be mixed within the ON clauses.

Basic rules

- Each explicit outer join ON clause must reference exactly two tables, no more and no less. Each joined table must be referenced at least once in the ON clauses.
- One of the tables must come from the immediate left part of the JOIN clause and the other, from the immediate right.

For example, the following query will be executed with success:

```
SELECT * FROM T1  
LEFT JOIN  
(T2 LEFT JOIN T3 ON T2.ID=T3.ID) -- here T2 is on the left and T3 is on the right  
ON T1.ID=T3.ID -- here T1 is on the left and T3 is on the right
```

With our three tables, this example could be:

```
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 Employees.name, Employees.depID, Employees.cityID, Departments.depID,
Departments.depName, Cities.cityID, Cities.cityName
FROM Departments
LEFT JOIN
(Employees LEFT JOIN Cities ON Employees.cityID=Cities.cityID)
ON Departments.depID=Employees.depID
INTO :aName, :aEmpDepID, :aEmpCityID, :aDepID, :aDepName, :aCityID, :aCityName;
```

End SQL

Here are the results:

aName	aEmpDepID	aEmpCityID	aDepID	aDepName	aCityID	aCityName
Alan	10	30	10	Program	NULL	NULL
Bernard	10	33	10	Program	30	Paris
Anne	11	39	11	Engineering	33	New York
Fabrice	12	35	12	Development	NULL	NULL
Thomas	10	NULL	10	Program	NULL	NULL
NULL	NULL	NULL	NULL	Marketing	NULL	NULL
NULL	NULL	NULL	13	Quality	NULL	NULL

On the other hand, the following three queries will be rejected since they violate certain rules:

```
SELECT * FROM T1
LEFT JOIN
(T2 LEFT JOIN T3 ON T2.ID=T1.ID) -- here T2 is on the left but T1 is not present in the immediate right
ON T1.ID=T3.ID
```

```
SELECT * FROM
(T1 LEFT JOIN T2 ON T1.ID=T2.ID)
LEFT JOIN
(T3 LEFT JOIN T4 ON T3.ID=T4.ID)
ON T3.Name=T4.Name -- here both T3 and T4 come from the right side of the JOIN clause and no tables
at all come from the left side
```

```
SELECT * FROM T1
LEFT JOIN
(T2 LEFT JOIN T3 ON T2.ID=T3.ID)
ON T1.ID=T3.ID AND T1.ID=T2.ID -- here more than two tables are being used in the ON clause: T1, T2,
and T3
```

Support of the ON condition

In general, if tables (Tx1, Tx2..., Txn) on the left of JOIN clause and tables (Ty1, Ty2..., Tym) on the right are being joined, then the ON expression must reference exactly one left table Txa and exactly one right table Tyb.

	Not supported in the ON clause	Supported in the ON clause
Boolean operations	OR	AND and NOT
Predicate and functions	IS NULL, COALESCE	All other predicates and built-in functions (can be used in any combination desired)
4D variable references	-	Supported without restriction
4D method calls	When either left or right side of the current JOIN clause is an explicit outer join	Any other cases (see example below)

The following example with a 4D method call is supported because there are no non-inner sub-joins to join:

```
SELECT * FROM T1
LEFT JOIN T2
ON T1.ID={FN My4DCall (T2.ID) AS INT32}
```

On the other hand, this example of 4D method call is not supported because non-inner sub-joins are being joined:

```
SELECT * FROM
(T1 LEFT JOIN T2 ON T1.ID=T2.ID)
LEFT JOIN -- Both left and right sides of this join clause contain explicit LEFT joins
(T3 LEFT JOIN T4 ON T3.ID=T4.ID)
ON T1.ID={FN My4DCall (T4.ID) AS INT32} -- non-inner sub-joins are being joined
```

























General limitations

- References to **SQL views** are not allowed in the explicit join declaration
- Subqueries that use external joins are not supported. The following will be rejected:

```
SELECT T2.ID FROM T2
WHERE T2.ID=(
SELECT COUNT ( * ) FROM
(T1 LEFT JOIN T3 ON T1.ID=T3.ID)
RIGHT JOIN T4 ON T3.ID=T4.ID)
```

SQL Commands

SQL Commands

-  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
-  CREATE VIEW
-  DROP VIEW
-  GRANT
-  REVOKE
-  REPLICATE
-  SYNCHRONIZE

SQL Commands

SQL commands (or statements) are generally grouped into two categories:

- Data Manipulation Commands, which are used to obtain, add, remove and/or modify database information. More specifically, this refers to the *SELECT*, **INSERT**, *UPDATE* and *DELETE* commands.
- Data Definition Commands, which are used to create or remove database objects or database structure objects. More specifically, this refers to the **CREATE DATABASE**, *CREATE TABLE*, *ALTER TABLE*, *DROP INDEX*, *DROP TABLE* or *CREATE SCHEMA* commands.

In the syntax, command names and keywords appear in bold and are passed "as is." Other elements appear in italics and are detailed separately in the chapter. Keywords and/or clauses that are passed in straight brackets [] are optional. The vertical bar character | separates the various alternatives available. When elements are passed in curly brackets { }, separated by vertical bars, this indicates that only one element of the set should be passed.

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**]

Beschreibung

The *SELECT* command is used to retrieve data from one or more tables.

If you pass ***, all the columns will be retrieved; otherwise you can pass one or more *select_item* type arguments to specify each column to be retrieved individually (separated by commas). If you add the optional keyword **DISTINCT** to the *SELECT* statement, no duplicate data will be returned. Queries with mixed *"**" and explicit fields are not allowed. For example, the following statement:

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

... is not allowed whereas:

```
SELECT * FROM OFFICES
```

...is allowed.

The **FROM** clause is used to specify one or more *table_reference* type arguments for the table(s) from which the data is to be retrieved. You can either pass a standard SQL name or a string. It is not possible to pass a query expression in the place of a table name. You may also pass the optional keyword **AS** to assign an alias to the column. If this keyword is passed, it must be followed by the alias name which can also be either an SQL name or string.

Note: This command does not support 4D fields of the Object type.

The optional **WHERE** clause sets conditions that the data must satisfy in order to be selected. This is done by passing a *search_condition* which is applied to the data retrieved by the **FROM** clause. The *search_condition* always returns a Boolean type value.

The optional **ORDER BY** clause can be used to apply a *sort_list* criteria to the data selected. You can also add the **ASC** or **DESC** keyword to specify whether to sort in ascending or descending order. By default, ascending order is applied.

The optional **GROUP BY** clause can be used to group identical data according to the *sort_list* criteria passed. Multiple group columns may be passed. This clause can be used to avoid

redundancy or to compute an aggregate function (**SUM**, **COUNT**, **MIN** or **MAX**) that will be applied to these groups. You can also add the **ASC** or **DESC** keyword as with the **ORDER BY** clause.

The optional **HAVING** clause can then be used to apply a *search_condition* to one of these groups. The **HAVING** clause may be passed without a **GROUP BY** clause.

The optional **LIMIT** clause can be used to restrict the number of data returned by passing a *4d_language_reference* variable or *int_number*.

The optional **OFFSET** clause can be used to set a number (*4d_language_reference* variable or *int_number*) of data to be skipped before beginning to count for the **LIMIT** clause.

The optional **INTO** clause can be used to indicate *4d_language_reference* variables to which the data will be assigned.

A **SELECT** command that specifies a **FOR UPDATE** clause attempts to obtain exclusive writing locks on all the selected records. If at least one record cannot be locked, then the whole command fails and an error is returned. If, however, all the selected records were locked, then they will remain locked until the current transaction is committed or rolled back.

Beispiel 1

Suppose that you have a movie database with one table containing the movie titles, the year it was released and the tickets sold for that movie.

We would like to get the years starting with 1979 and the amount of tickets sold where the total sold was less than 10 million. We want to skip the first 5 years and to display only 10 years, ordered by the year.

```
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

Beispiel 2

Here is an example where a combination of search conditions are used:

```
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');
```

Beispiel 3

Given a SALESREPS table where QUOTA is the expected sales amount for a sales representative and SALES is the actual amount of sales made.

```
ARRAY REAL(arrMin_Values;0)
ARRAY REAL(arrMax_Values;0)
ARRAY REAL(arrTotal_Values;0)
Begin SQL
  SELECT MIN ( ( SALES * 100 ) / QUOTA ),
  MAX( ( SALES * 100 ) / QUOTA ),
  SUM( QUOTA ) - SUM ( SALES )
  FROM SALESREPS
  INTO :arrMin_Values, :arrMax_Values, :arrTotal_Values;
End SQL
```

Beispiel 4

Here is an example which finds all the actors born in a certain city:

```
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]
```

Beschreibung

The **INSERT** command is used to add data to an existing table. The table where the data is to be added is passed either using an *sql_name* or *sql_string*. The optional *column_reference* type arguments passed indicate the name(s) of the column(s) where the values are to be inserted. If no *column_reference* is passed, the value(s) inserted will be stored in the same order as in the database (1st value passed goes into 1st column, 2nd value into 2nd column, and so on).

Note: This command does not support 4D fields of the Object type.

The **VALUES** keyword is used to pass the value(s) to be placed in the column(s) specified. You can either pass an *arithmetic_expression* or **NULL**. Alternatively, a *subquery* can be passed in the **VALUES** keyword in order to insert a selection of data to be passed as the values.

The number of values passed in the **VALUES** keyword must match the number of columns specified by the *column_reference* type argument(s) passed and each of them must also match the data type of the corresponding column or at least be convertible to that data type.

The **INFILE** keyword lets you use the contents of an external file to specify the values of a new record. This keyword must only be used with VARCHAR type expressions. When the **INFILE** keyword is passed, the *arithmetic_expression* value is evaluated as a file pathname; if the file is found, the contents of the file are inserted into the corresponding column. Only fields of the Text or BLOB type can receive values from an **INFILE**. The contents of the file are transferred as raw data, with no interpretation.

The file searched for must be on the computer hosting the SQL engine, even if the query comes from a remote client. Similarly, the pathname must be expressed respecting the syntax of the operating system of the SQL engine. It can be absolute or relative.

The **INSERT** command is supported in both single- and multi-row queries. However, a multi-row **INSERT** statement does not allow UNION and JOIN operations.

The 4D engine allows the insertion of multi-row values, which can simplify and optimize the code, in particular when inserting large quantities of data. The syntax of multi-row insertions is of the type:

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

This syntax is illustrated in examples 3 and 4.

Beispiel 1

Here is a simple example inserting a selection from table2 into table1:

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

Beispiel 2

This example creates a table and then inserts values into it:

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

Beispiel 3

A multi-row syntax can be used to avoid tedious repetition of rows:

```
INSERT INTO MyTable
(Flid1,Flid2,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');
```

Beispiel 4

You can also use 4D variables or arrays with a multi-row syntax:

```
INSERT INTO MyTable
(Flid1,Flid2,BoolFld,DateFld,TimeFld, InfoFld)
VALUES
(:vArrId, :vArrIdx, :vArrbool, :vArrdate, :vArrL, :vArrText);
```

Note: You cannot combine simple variables and arrays in the same **INSERT** statement.

UPDATE

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

Beschreibung

The *UPDATE* command can be used to modify data contained within a table indicated by passing an *sql_name* or *sql_string*.

The **SET** clause is used to assign new values (either an *arithmetic_expression* or **NULL**) to the *sql_name* type argument(s) passed.

The optional **WHERE** clause is used to specify which data (those meeting the *search_condition*) are to be updated. If it is not passed, all the data of the table will be assigned the new value(s) passed in the **SET** clause.

Note: This command does not support 4D fields of the Object type.

The *UPDATE* command is supported for both queries and subqueries; however, a positioned *UPDATE* statement is not supported.

A **CASCADE** type update is implemented in 4D, but the **SET NULL** and **SET DEFAULT** delete rules are not supported.

Beispiel

Here is an example which updates the MOVIES table so that the tickets sold for the movie "Air Force One" is set to 3,500,000:

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

DELETE

```
DELETE FROM {sql_name | sql_string}
```

```
[WHERE search_condition]
```

Description

The *DELETE* command can be used to remove all or part of the data from a table indicated by passing an *sql_name* or *sql_string* after the **FROM** keyword.

The optional **WHERE** clause is used to indicate which part of the data (those meeting the *search_condition*) are to be deleted. If it is not passed, all the data of the table will be removed.

A positioned *DELETE* statement is not supported. A **CASCADE** type delete is implemented in 4D, but the **SET DEFAULT** and **SET NULL** delete rules are not supported.

Note: This command does not support 4D fields of the Object type.

Example

Here is an example that removes all the movies released in the year 2000 or previously from the MOVIES table:

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

CREATE DATABASE

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

Beschreibung

The **CREATE DATABASE** command lets you create a new external database (.4db and .4dd files) at a specific location.

If the **IF NOT EXISTS** constraint is passed, the database is not created and no error is generated if a database with the same name already exists at the location specified.

If the **IF NOT EXISTS** constraint is not passed, the database is not created and the "Database already exists. Failed to execute CREATE DATABASE command." error message is displayed if a database with the same name already exists at the location specified.

The **DATAFILE** clause lets you specify the complete name (complete pathname + name) of the new external database. You must pass the name of the structure file. The program automatically adds the ".4db" extension to the file if it is not already specified and creates the data file. The pathname can be expressed either in POSIX syntax or in the system syntax. It can be absolute or relative to the structure file of the main 4D database.

- POSIX syntax (URL type): folder names are separated by a slash ("/"), regardless of the platform that you use, for example: ".../extdatabases/myDB.4db"
For an absolute path, pass in first position the volume name and a colon, for example: "C:/test/extdatabases/myDB.4db"
- system syntax: pathname respecting the syntax of the current platform, for example:
 - (Mac OS) Disque:Applications:monserv:extdatabases:mabase.4db
 - (Windows) C:\Applications\myserv\extdatabases\myDB.4db

After successful execution of the **CREATE DATABASE** command, the new database created does not automatically become the current database. To do this, you must explicitly declare it as the current database using the **USE DATABASE** command.

About external databases

An external database is a 4D database that is independent from the main 4D database, but that you can work with from the main 4D database by using the SQL engine of 4D. Using an external database means temporarily designating this database as the current database, in other words, as the target database for the SQL queries executed by 4D. By default, the main database is the current database.

You can create an external database directly from the main database with the **CREATE DATABASE** command. Once created, an external database can be designated as the current database using the **USE DATABASE** command. It can then be modified via standard SQL commands (*CREATE TABLE*, *ALTER TABLE*, etc.) and you can store data in it. The **DATABASE_PATH** function can be used to find out the current database at any time.

The main interest of external databases resides in the fact that they can be created and worked with via 4D components. This allows the development of components that are capable of creating tables and fields according to their needs.

Note: An external database is a standard 4D database. It can be opened and worked with as the main database by a 4D or 4D Server application. Conversely, any standard 4D database can be used as an external database. However, it is imperative that you do not activate the access management system (by assigning a password to the Designer) in an external database, otherwise you will no longer be able to have access to it via the **USE DATABASE** command.

Beispiel 1

Creation of ExternalDB.4DB and ExternalDB.4DD external database files at the location C:/MyDatabase/:

Begin SQL

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

End SQL

Beispiel 2

Creation of TestDB.4DB and TestDB.4DD external database files next to the structure file of the main database:

Begin SQL

```
CREATE DATABASE IF NOT EXISTS DATAFILE 'TestDB';
```

End SQL

Beispiel 3

Creation of External.4DB and External.4DD external database files at the location specified by the user:

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**]

Beschreibung

The **USE DATABASE** command is used to designate an external database as the current database, in other words, the database to which the next SQL queries in the current process will be sent. All types of SQL queries are concerned: queries included in the **Begin SQL/End SQL** structure, **SQL EXECUTE** or **SQL EXECUTE SCRIPT** commands, etc.

Note: For more information about external databases, please refer to the description of the **CREATE DATABASE** command.

- If you are working in a single-user configuration, the external database must be located on the same machine as your 4D.
- If you are working in remote mode, the external database can be located on the local machine or on the 4D Server machine.

If you are using 4D in remote mode, the **REMOTE** keyword can be used to designate an external database located on 4D Server.

For security reasons, this mechanism only works with native remote connections, in other words, in the context of a remote 4D database connected with 4D Server. Connections via ODBC or pass-through connections are not allowed.

If no keyword is specified, the **LOCAL** option is used by default. If you are using 4D in local mode, the **REMOTE** and **LOCAL** keywords are ignored: connections are always local.

To designate an external database to be used, pass its complete pathname (access path + name) in the **DATAFILE** clause. The path can be expressed either in the POSIX syntax, or in the system syntax. It can be absolute or relative to the structure file of the main 4D database.

In remote mode, if the **REMOTE** keyword is passed, this parameter designates the database path from the server machine. If it is omitted or if the **LOCAL** keyword is passed, this parameter designates the database path on the local 4D machine.

Important: You must designate a valid external database, and one where access control has not been activated (by assigning a password to the Designer). Otherwise, an error is generated.

In order to reset the main database as the current database, execute the command while passing the **SQL_INTERNAL** or **DEFAULT** keyword.

Pass **AUTO_CLOSE** if you want to physically close the external database after its use; in other words, when you change the current database. In fact, since opening an external database is an operation that requires some time, for optimization reasons 4D keeps information stored in memory concerning external databases opened during the user session. This information is kept as long as the 4D application remains launched. Subsequent opening of the same external database is therefore faster. However, this prevents the sharing of external databases among several 4D applications because the external database remains open in read/write for the first application that uses it. If several 4D applications must be able to use the same external database simultaneously, pass the **AUTO_CLOSE** keyword in order to physically release the external database after its use.

This restriction does not apply to processes of the same application: different processes of an application can always access the same external database in read/write without it being necessary to force it to close.

Note that when several processes use the same external database, it is physically released only when the last process that uses it is closed, even when the **AUTO_CLOSE** option has been passed.

You should take this functioning into account for operations that involve inter-application sharing or deletion of external databases.

Beispiel

Use of an external database for a request then return to the main database:

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 | TRIGGERS}

Beschreibung

The **ALTER DATABASE** command enables or disables SQL options of the current database for the current session, i.e. for all users and processes until the database is restarted.

This command is intended to allow you to temporarily disable SQL options in order to accelerate certain operations that take up a lot of resources. For example, disabling indexes, constraints and triggers before beginning the import of a large quantity of data can significantly reduce the duration of the import.

Note that constraints include primary keys and foreign keys as well as unique and null attributes. If you want to manage triggers individually for each table, you must use **ALTER TABLE**.

Beispiel

Example of an import with temporary disabling of all SQL options:

Begin SQL

```
ALTER DATABASE DISABLE INDEXES;  
ALTER DATABASE DISABLE CONSTRAINTS;  
ALTER DATABASE DISABLE TRIGGERS;
```

End SQL

```
SQL EXECUTE SCRIPT("C:\\Exported_data\\Export.sql";SQL_On error continue)
```

Begin SQL

```
ALTER DATABASE ENABLE INDEXES;  
ALTER DATABASE ENABLE CONSTRAINTS;  
ALTER DATABASE ENABLE TRIGGERS;
```

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]
```

Beschreibung

The **CREATE TABLE** command is used to create a table named *sql_name* having the fields specified by passing one or more *column_definition* and/or *table_constraint* type arguments. If the **IF NOT EXISTS** constraint is passed, the table is only created when there is no table with the same name already in the database. Otherwise, it is not created and no error is generated.

The first *sql_name* parameter (optional) can be used to designate the SQL schema to which you want to assign the table. If you do not pass this parameter or if you pass the name of a schema that does not exist, the table is automatically assigned to the default schema, named "DEFAULT_SCHEMA." For more information about SQL schemas, please refer to the **4D SQL engine implementation** section.

Note: It is also possible to assign a table to an SQL schema using the "Schemas" pop-up menu found in the 4D table Inspector palette. This menu contains the list of schemas defined in the database.

A *column_definition* contains the name (*sql_name*) and data type (*sql_data_type_name*) of a column and a *table_constraint* restricts the values that a table can store.

Note: This command does not allow a field (column) of the Object type to be added.

The **PRIMARY KEY** keyword is used to specify the primary key when the table is created. For more information about primary keys, please refer to the **4D SQL engine implementation** section.

The **ENABLE REPLICATE** and **DISABLE REPLICATE** keywords are used to enable or disable the mechanism allowing replication of the table (see the **Replication via SQL** section).

Beispiel 1

Here is a simple example for creating a table with two columns:

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

Beispiel 2

This example creates the same table but only if there is no existing table with the same name:

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

Beispiel 3

This example creates a "Preferences" table and assigns it to the "Control" schema:

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

ALTER TABLE

```
ALTER TABLE sql_name
{ADD [TRAILING] 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] |
[ENABLE | DISABLE} LOG] |
[MODIFY sql_name {ENABLE | DISABLE} AUTO_INCREMENT] |
[MODIFY sql_name {ENABLE | DISABLE} AUTO_GENERATE] |
[ENABLE | DISABLE} TRIGGERS] |
SET SCHEMA sql_name}
```

Beschreibung

The `ALTER TABLE` command is used to modify an existing table (*sql_name*). You can carry out one of the following actions:

Passing **ADD** *column_definition* adds a column to the table. The **TRAILING** keyword (which must be placed in front of *column_definition* if it is used) forces the column to be created after the last existing column of the table in the structure file. This option is useful when columns containing data have been deleted from the table (without the data being erased), to prevent existing data from being reassigned to the new column.

Note: This command does not allow a field (column) of the Object type to be added.

The **PRIMARY KEY** keyword is used to set the primary key when a column is added.

Passing **DROP** *sql_name* removes the column named *sql_name* from the table.

Passing **ADD** *primary_key_definition* adds a **PRIMARY KEY** to the table.

Passing **DROP PRIMARY KEY** removes the **PRIMARY KEY** of the table.

Passing **ADD** *foreign_key_definition* adds a **FOREIGN KEY** to the table.

Passing **DROP CONSTRAINT** *sql_name* removes the specified constraint from the table.

Passing **ENABLE REPLICATE** or **DISABLE REPLICATE** enables or disables the mechanism allowing replication of the table (see the [Replication via SQL](#) section).

Passing **ENABLE LOG** or **DISABLE LOG** enables or disables journaling for the table.

Passing **ENABLE AUTO_INCREMENT** or **DISABLE AUTO_INCREMENT** enables or disables the "Autoincrement" option for Longint type fields. Passing **ENABLE AUTO_GENERATE** or **DISABLE AUTO_GENERATE** enables or disables the "Auto UUID" option for Alpha fields of the UUID type. In both these cases, you must first pass the **MODIFY** keyword followed by the *sql_name* of the column to modify.

Passing **ENABLE TRIGGERS** or **DISABLE TRIGGERS** enables or disables triggers for the table. If you want to manage triggers globally at the database level, you need to use **ALTER DATABASE**.

Passing **SET SCHEMA** *sql_name* transfers the table to the *sql_name* schema.

The command returns an error:

- when the optional **ENABLE LOG** parameter is passed and no valid primary key is defined,
- if you attempt to modify or delete the definition of the table's primary key without disabling journaling by means of **DISABLE LOG**.

Beispiel 1

This example creates a table, inserts a set of values into it, then adds a Phone_Number column, adds another set of values and then removes the ID column:

```
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;
```

Beispiel 2

Example for activating the "Autoincrement" option of the Longint type [Table_1]id field:

Begin SQL

```
ALTER TABLE Table_1 MODIFY id ENABLE AUTO_INCREMENT;
```

End SQL

Deactivating the option:

Begin SQL

```
ALTER TABLE Table_1 MODIFY id DISABLE AUTO_INCREMENT;
```

End SQL

Example for activating the "Auto UUID" of the Alpha type [Table_1]uid field:

Begin SQL

```
ALTER TABLE Table_1 MODIFY uid ENABLE AUTO_GENERATE;
```

End SQL

Deactivating the option:

Begin SQL

```
ALTER TABLE Table_1 MODIFY uid DISABLE AUTO_GENERATE;
```

End SQL

DROP TABLE

DROP TABLE [**IF EXISTS**] *sql_name*

Beschreibung

The *DROP TABLE* command is used to remove the table named *sql_name* from a database. When the **IF EXISTS** constraint is passed, if the table to be removed does not exist in the database, the command does nothing and no error is generated.

This command not only removes the table structure, but also its data and any indexes, triggers and constraints that are associated with it. It cannot be used on a table that is referenced by a **FOREIGN KEY** constraint.

Note: You must make sure that when the *DROP TABLE* command is executed, there are not any records of the *sql_name* table that are loaded in memory in write mode. Otherwise, the error 1272 is generated.

Beispiel 1

Here is a simple example which removes the ACTOR_FANS table:

```
DROP TABLE ACTOR_FANS
```

Beispiel 2

This example does the same as the one above except that in this case, if the ACTOR_FANS table does not exist, no error is generated:

```
DROP TABLE IF EXISTS ACTOR_FANS
```


CREATE INDEX

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

Description

The **CREATE INDEX** command is used to create an index (*sql_name*) on one or more columns of an existing table (*sql_name*) designated by one or more *column_reference* type arguments. Indexes are transparent to users and serve to speed up queries.

You can also pass the optional **UNIQUE** keyword to create an index that does not allow duplicate values.

Example

Here is a simple example for creating an index:

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

DROP INDEX

DROP INDEX *sql_name*

Description

The *DROP INDEX* command is used to remove an existing index named *sql_name* from a database. It cannot be used on indexes created for **PRIMARY KEY** or **UNIQUE** constraints.

Example

Here is a simple example for removing an index:

```
DROP INDEX ID_INDEX
```

LOCK TABLE

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

Description

The *LOCK TABLE* command is used to lock the table named *sql_name* in either **EXCLUSIVE** or **SHARE** mode.

In **EXCLUSIVE** mode, the data of the table cannot be read or modified by another transaction.

In **SHARE** mode, the data of the table can be read by concurrent transactions but modifications are still prohibited.

Example

This example locks the MOVIES table so that it can be read but not modified by other transactions:

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

UNLOCK TABLE

UNLOCK TABLE *sql_name*

Description

The *UNLOCK TABLE* command is used to unlock a table that has previously been locked via the *LOCK TABLE* command. It will not work if it is passed within a transaction or if it is used on a table that is locked by another process.

Example

This command removes the lock on the MOVIES table:

```
UNLOCK TABLE MOVIES
```

EXECUTE IMMEDIATE

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

Description

The *EXECUTE IMMEDIATE* command is used to execute a dynamic SQL statement. The *sql_name* passed represents a variable containing a set of SQL statements that will be executed as a whole.

Notes:

- This command cannot be used within a connection to an external data source (SQL pass-through) established using the **USE EXTERNAL DATABASE** 4D command.
- In compiled mode, it is not possible to use local 4D variables (beginning with the \$ character) in the query string passed to the *EXECUTE IMMEDIATE* command.

Example

This example recovers the number of movies in the database that were released in 1960 or more recently:

```
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*

Beschreibung

The CREATE SCHEMA command is used to create a new SQL schema named *sql_name* in the database. You can use any *sql_name* except for "DEFAULT_SCHEMA" and "SYSTEM_SCHEMA".

Note: For more information about schemas, please refer to the **Principles for integrating 4D and the 4D SQL engine** section.

When you create a new schema, by default the following access rights are associated with it:

- Read-only (Data): <Everybody>
- Read/Write (Data): <Everybody>
- Full (Data & Structure): <Nobody>

Each schema can be attributed external access rights using the *GRANT* command.

Only the Designer and Administrator of the database can create, modify or delete schemas.

If the access management system of 4D is not activated (in other words, if no password has been assigned to the Designer), all users can create and modify schemas with no restriction.

When a database is created or converted with 4D v11 SQL (starting with release 3), a default schema is created in order to group together all the tables of the database. This schema is named "DEFAULT_SCHEMA". It cannot be deleted or renamed.

Example

Creation of a schema named "Accounting_Rights":

```
CREATE SCHEMA Accounting_Rights
```

ALTER SCHEMA

ALTER SCHEMA *sql_name* **RENAME TO** *sql_name*

Beschreibung

The ALTER SCHEMA command can be used to rename the *sql_name* (first parameter) SQL schema to *sql_name* (second parameter).

Only the database Designer and Administrator can modify schemas.

Note: You cannot rename the default schema ("DEFAULT_SCHEMA") or the schema containing the system tables ("SYSTEM_SCHEMA") and you cannot use these names in the second *sql_name* parameter.

Example

Renaming of the MyFirstSchema schema to MyLastSchema:

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

DROP SCHEMA

DROP SCHEMA *sql_name*

Beschreibung

The DROP SCHEMA command can be used to delete the schema designated by *sql_name*.

It is possible to delete any schema except for the default schema (DEFAULT_SCHEMA) and the schema containing the system tables ("SYSTEM_SCHEMA"). When you delete a schema, all the tables that were assigned to it are transferred to the default schema. The transferred tables inherit the access rights of the default schema.

If you attempt to remove a schema that does not exist or that has already been deleted, an error is generated.

Only the database Designer and Administrator can delete schemas.

Example

You want to delete the MyFirstSchema schema (to which Table1 and Table2 are assigned):

```
DROP SCHEMA MyFirstSchema
```

After this operation, the two tables, Table1 and Table2, are reassigned to the default schema.

CREATE VIEW

```
CREATE [OR REPLACE] VIEW [schema_name.]view_name [(column_list)] AS select_statement[;]
```

Beschreibung

The **CREATE VIEW** command creates an SQL view named *view_name* (which is a standard *sql_name*) containing the columns defined in the *column_list* parameter. You will need to specify a column name if this column is a function or is derived from an arithmetic operation (scalar). It is also necessary to specify a column name if you want to avoid having different columns with the same name (for example, during a JOIN operation) or when you want to use a different column name than the one from which it is derived.

If the *column_list* parameter is passed, it must contain the same number of columns as there are in the *select_statement* definition query of the view. If *column_list* is omitted, the columns of the view will have the same names as those of the columns in the *select_statement* of the view.

Views and tables must have unique names.

If you pass the **OR REPLACE** option, the view is automatically created again if it already exists. This option can be useful in order to change the definition of an existing view without having to delete/re-create/affect the privileges of objects already defined for the current view.

When the **OR REPLACE** option is not passed and the view already exists, an error is returned.

schema_name is also a standard *sql_name* and you can use it to designate the name of the schema that will contain the view. If you do not pass *schema_name* or if you pass the name of a schema that does not exist, the view is automatically assigned to the default schema, which is entitled "DEFAULT_SCHEMA".

select_statement designates the **SELECT** that is the definition query of the view. The *select_statement* is the same as a standard **SELECT** in 4D, but with the following restrictions:

- You cannot use **INTO**, **LIMIT** or **OFFSET** clauses since the limitation, offset or assignment of variables in 4D will be performed by the **SELECT** that calls the view.
- You cannot use the **GROUP BY** clause.
- Views are in read-only mode and cannot be updated.

View definition is "static" and is not updated when a source table is modified or deleted. More particularly, any columns added to a table do not appear in the view based on this table. Similarly, if you try to access deleted columns by means of a view, this causes an error.

However, a view that refers to a deleted source view will continue to work. In fact, when you create a view, it converts any view reference(s) into references to the source tables.

Views have a global scope. Once a view is created using **CREATE VIEW**, it can be accessed by all parts of the application (4D remote using SQL, external databases created using the **CREATE DATABASE** command, other databases using the **SQL LOGIN** command, etc.) during the session until it is deleted using the **DROP VIEW** command or until the database is closed.

Beispiel

Here are a few examples of view definitions, given a PEOPLE table containing the following columns:

```
ID          INT64
FIRST_NAME  VARCHAR(30)
LAST_NAME   VARCHAR(30)
DEPARTMENT  VARCHAR(30)
SALARY      INT
```

A view with no restrictions:

```
CREATE VIEW FULLVIEW AS
  SELECT * FROM PERSONS;
```

A view with "horizontal" restrictions. For example, you want to only display people in the Marketing department:

```
CREATE VIEW HORIZONTALVIEW (ID, FirstName, LastName, Salary) AS
  SELECT ID, FIRST_NAME, LAST_NAME, SALARY FROM PERSONS
  WHERE DEPARTMENT = 'Marketing';
```

An aggregated view:

```
CREATE VIEW AGGREGATEVIEW (FirstName, LastName AnnualSalary) AS
  SELECT FirstName, LastName, SALARY*12 FROM PERSONS;
```

A view with "vertical" restrictions. For example, you do not want to display the SALARY column:

```
CREATE VIEW VERTICALVIEW (ID, FirstName, LastName, Department) AS
  SELECT ID, FIRST_NAME, LAST_NAME, DEPARTEMENT FROM PERSONS;
```

Once the views are defined, you can use them just like standard tables. For example, if you want to get every person whose salary is greater than 5,000 Euros:

```
SELECT * FROM FULLVIEW
  WHERE SALARY < 5000
  INTO :aID, :aFirstName, :aLastName, :aDepartment, :aSalary;
```

Another example: you want to get every person in the Marketing department whose first name is "Michael":

```
SELECT ID, LastName, Salary FROM HORIZONTALVIEW
  WHERE FirstName='Michael'
  INTO :aID, :aLastName, :aSalary;
```

DROP VIEW

```
DROP VIEW [IF EXISTS] [schema_name.]view_name[:];
```

Beschreibung

The **DROP VIEW** command deletes the view named *view_name* from the database.

When the **IF EXISTS** constraint is passed, the command does nothing and no error is generated if the *view_name* view does not exist in the database.

schema_name is a standard *sql_name* and you can use it to designate the name of the schema that will contain the view. If you do not pass *schema_name* or if you pass the name of a schema that does not exist, the view is automatically considered to belong to the default schema, which is entitled "DEFAULT_SCHEMA".

GRANT

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

Beschreibung

The GRANT command can be used to set the access rights associated with the *sql_name* schema (first parameter). These rights will be assigned to the group of 4D users designated by the second *sql_name* parameter.

The **READ**, **READ_WRITE** and **ALL** keywords can be used to set the access rights allowed for the table:

- **READ** establishes Read-only access (data). By default: <Everybody>
- **READ_WRITE** establishes Read/Write access (data). By default: <Everybody>
- **ALL** establishes full access mode (data and structure). By default: <Nobody>

Note that each type of access is set separately from the others. More specifically, if you assign only the READ access rights to one group, this will not have any effect since the group as well as all the others will continue to benefit from READ_WRITE access (assigned to all groups by default). To set READ access, you must call the **GRANT** command twice (see example 2).

Access control only applies to external connections. The SQL code executed within 4D via the **Begin SQL/End SQL** tags or commands such as **SQL EXECUTE** still have full access.

Compatibility Note: During the conversion of an older database to version 11.3 or higher, the global access rights (as set on the SQL page of the application Preferences) are transferred to the default schema.

The second *sql_name* parameter must contain the name of a group of 4D users to which you want to assign access rights to the schema. This group must exist in the 4D database.

Note: 4D allows group names to include spaces and/or accented characters that are not accepted by standard SQL. In this case, you must put the name between the [and] characters. For example: **GRANT READ ON [my schema] TO [the admins!]**

Only the database Designer and Administrator can modify schemas.

Note regarding referential integrity

4D ensures the principle of referential integrity independently from access rights. For example, suppose that you have two tables, Table1 and Table2, connected by a Many-to-One type relation (Table2 -> Table1). Table1 belongs to schema S1 and Table2 to schema S2. A user that has access rights to schema S1 but not to S2 can delete records in Table1. In this case, in order to respect the principles of referential integrity, all the records of Table2 that are related to records deleted from Table1 will also be deleted.

Beispiel 1

You want to allow read/write access to data of the MySchema1 schema to the "Power_Users" group:

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

Beispiel 2

You want to allow read-only access to the "Readers" group. This case requires assigning at least one group with READ_WRITE access rights (here it is "Admins") so that it is no longer assigned to all groups by default:

```
GRANT READ ON MySchema2 TO Readers /*Assignment of read-only access */  
GRANT READ_WRITE ON MySchema2 TO Admins /*Stop read-write access to all*/
```

REVOKE

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

Description

The *REVOKE* command can be used to remove specific access rights associated with the schema set by the *sql_name* parameter.

In fact, when you execute this command, you assign the <Nobody> pseudo-group of users to the specified access right.

Example

You want to delete all read-write access rights to the MySchema1 schema:

```
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)};
```

Beschreibung

The **REPLICATE** command lets you replicate the data of a table of database A into that of a table of database B. By convention, the database where the command is executed is called the "local database" and the database from which the data are replicated is called the "remote database."

This command can only be used in the framework of a database replication system. In order for the system to work, replication must have been enabled on the local database and the remote database side and each table implicated must have a primary key. For more information about this system, please refer to the [Replication via SQL](#) section.

Note: If you would like to implement a complete synchronization system, please refer to the description of the **SYNCHRONIZE** command.

Pass a list of fields (virtual or standard) separated by commas in *replicated_list*. The fields must belong to the *table_reference* table of the remote database.

The **FROM** clause must be followed by an argument of the *table_reference* type which can be used to designate the table of the remote database from which to replicate the data of the *replicated_list* fields.

Note: The virtual fields of the remote table can only be stored in the arrays of the local database.

Remote database side

The optional **WHERE** clause can be used to apply a preliminary filter to the records of the table in the remote database so that only those records that satisfy the *search_condition* will be taken into account by the command.

4D then recovers the values of the *replicated_list* fields for all the records designated by the **FOR REMOTE STAMP** clause. The value passed in this clause can be either:

- a value of the type **longint > 0**: In this case, records where the value of `__ROW_STAMP` is greater than or equal to this value are recovered.
- **0**: In this case, all the records where the value of `__ROW_STAMP` is different from 0 are recovered. Note that any records that existed before the enabling of replication will therefore not be taken into account (the value of their `__ROW_STAMP` = 0).
- **-1**: In this case, all the records of the remote table are recovered; in other words, all the records where the value of `__ROW_STAMP` >= 0. Unlike the previous case, all the records of the table, including any that existed before replication was enabled, will be taken into account.
- **-2**: In this case, all the records deleted from the remote table (after enabling of replication) are recovered; in other words, all the records where the value of `__ROW_ACTION` = 2.

Finally, you can apply the optional **OFFSET** and/or **LIMIT** clauses to the selection obtained:

- When it is passed, the **OFFSET** clause is used to ignore the first X records of the selection (where X is the value passed to the clause).
- When it is passed, the **LIMIT** clause is used to restrict the result selection to the first Y records (where Y is the value passed to the clause). If the **OFFSET** clause is also passed, the **LIMIT** clause is applied to the selection obtained after the execution of **OFFSET**.

Once both clauses have been applied, the resulting selection is sent to the local database.

Local database side

The values recovered are directly written into the *target_list* of the local database or in the standard fields specified by *sql_name* of the *table_reference* table of the local database. The *target_list* argument can contain either a list of standard fields or a list of arrays of the same type as the remote fields (but not a combination of both). If the destination of the command is a list of fields, the target records will be automatically created, modified or deleted according to the action stored in the virtual `__ROW_ACTION` field.

You resolve conflicts for replicated records that already exist in the target database (identical primary keys) using priority clauses (**REMOTE OVER LOCAL** or **LOCAL OVER REMOTE**):

- If you pass the **REMOTE OVER LOCAL** option or omit the priority clause, all the source records (remote database) designated by the **FOR REMOTE STAMP** clause replace the target records (local database) if they already exist -- regardless of whether they were modified or not, on either side. In this case, it is pointless to pass a **LOCAL STAMP** clause because it is ignored.
- If you pass the **LOCAL OVER REMOTE** option, the command takes the **LOCAL STAMP** into account. In this case, target records (local database) whose stamp values are less than or equal to the one that is passed in **LOCAL STAMP** are not replaced by the source records (remote database). For example, if you pass 100 in **LOCAL STAMP**, all the records of the local database whose stamp is ≤ 100 are not replaced by the equivalent records of the remote database. This lets you keep modified data locally and reduce the selection of records to be replicated in the local table.
- If you pass the **LATEST REMOTE STAMP** and/or **LATEST LOCAL STAMP** clauses, 4D returns the values of the last stamps of the remote and local tables in the corresponding *4d_language_reference* variables. This information can be useful if you want to automate the management of the synchronization procedure. These values correspond to the value of the stamps just after the replication operation was completed: if you use them in a subsequent **REPLICATE** or **SYNCHRONIZE** statement, you do not need to increment them because they were automatically incremented before being returned by the **REPLICATE** command.

If the replication operation is carried out correctly, the OK system variable is set to 1. You can check this value from a 4D method.

If errors occur during the replication operation, the operation is stopped at the first error that occurs. The last source variable (if it has been specified) is valorized with the stamp of the record in which the error occurred. The OK system variable is set to 0. The error generated can be intercepted by an error-handling method installed by the **ON ERR CALL** command.

Note: Operations carried out by the **REPLICATE** command do not take data integrity constraints into account. This means, for instance, that the rules governing foreign keys, uniqueness, and so on, are not checked. If the data received could undermine data integrity, you must check the data after the replication operation is finished. The simplest way is to lock, via the 4D or SQL language, the records that have to be modified.

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;
```

Beschreibung

The **SYNCHRONIZE** command lets you synchronize two tables located on two different 4D SQL servers. Any change made to one of the tables is also carried out in the other. The 4D SQL server that executes the command is called the local server and the other server is called the remote server.

The **SYNCHRONIZE** command is a combination of two internal calls to the **REPLICATE** command. The first call replicates the data from the remote server to the local server and the second carries out the opposite operation: replication of local server data to the remote server. The tables to be synchronized must therefore be configured for replication:

- They must have a primary key,
- The "Enable Replication" option must be checked in the Inspector window of each table.

For more information, please refer to the description of the **REPLICATE** command.

The **SYNCHRONIZE** command accepts four stamps as "parameters": two input stamps and two output stamps (last modification). The input stamps are used to indicate the moment of the last synchronization on each server. The output stamps return the value of the modification stamps on each server right after the last modification. Thanks to this principle, when the **SYNCHRONIZE** command is called regularly, it is possible to use the output stamps of the last synchronization as input stamps for the next one.

Note: Input and output stamps are expressed as number values and not as timestamps. For more information about these stamps, please refer to the description of the **REPLICATE** command.

In the event of an error, the output stamp of the server concerned contains the stamp of the record at the origin of the error. If the error stems from a cause other than the synchronization (network problems for example), the stamp will contain 0.

There are two different error codes, one to indicate a synchronization error on the local site and another for a synchronization error on the remote site.

When an error occurs, the state of the data will depend on that of the transaction on the local server. On the remote server, the synchronization is always carried out within a transaction, so the data cannot be altered by the operation. However, on the local server, the synchronization process is placed under the control of the developer. It will be carried out outside of any transaction if the **Auto-commit Transactions** preference is not selected, (otherwise, a transaction context is automatically created). The developer can decide to start a transaction and it is up to the developer to validate or cancel this transaction after data synchronization.

You can "force" the synchronization direction using the **REMOTE OVER LOCAL** and **LOCAL OVER REMOTE** clauses, depending on the characteristics of your application. For more information about the implementation mechanisms, please refer to the description of the **REPLICATE** command.

Note: Operations carried out by the **SYNCHRONIZE** command do not take data integrity

constraints into account. This means, for instance, that the rules governing foreign keys, uniqueness, and so on, are not checked. If the data received could undermine data integrity, you must check the data after the synchronization operation. The simplest way is to lock, via the 4D or SQL language, the records that must be modified.

In the *4d_language_ref* variables of the **LATEST REMOTE STAMP** and **LATEST LOCAL STAMP** clauses, 4D returns the values of the last stamps of distant and local tables. This information lets you automate the handling of the synchronization procedure. They correspond to the value of the stamps just after the end of the replication operation: if you use them in a subsequent **REPLICATE** or **SYNCHRONIZE** statement, you do not need to increment them; they are incremented automatically before being returned by the **REPLICATE** command.

Beispiel

To understand the mechanisms involved in a synchronization operation, we are going to look at the different possibilities related to updating of an existing record in both of the synchronized databases.

The synchronization method takes the following form:

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

vRemoteStamp:=X... // see values in the array below
vLocalStamp:=X... // see values in the array below
vLatestRemoteStamp:=X... // value returned in a previous LATEST REMOTE STAMP
vLatestLocalStamp:=X... // value returned in a previous LATEST LOCAL STAMP
```

Begin SQL

```
SYNCHRONIZE
  LOCAL MYTABLE (MyField)
  WITH
  REMOTE MYTABLE (MyField)
  FOR REMOTE STAMP :vRemoteStamp,
  LOCAL STAMP :vLocalStamp
  LOCAL OVER REMOTE // or REMOTE OVER LOCAL, see in array below
  LATEST REMOTE STAMP :vLatestRemoteStamp,
  LATEST LOCAL STAMP :vLatestLocalStamp;
```

End SQL

The initial data is:

- The record stamp in the LOCAL database has a value of 30 and the one in the REMOTE database has a value of 4000
- The values of the MyField field are as follows:

LOCAL		REMOTE	
Old value	New value	Old value	New value
AAA	BBB	AAA	CCC





























- We use values returned by previous **LATEST LOCAL STAMP** and **LATEST REMOTE STAMP** clauses in order to synchronize only those values that were modified since the last synchronization.

Here are the synchronizations made by the **SYNCHRONIZE** command according to the values passed in the **LOCAL STAMP** and **REMOTE STAMP** parameters as well as the priority option used: ROL (for **REMOTE OVER LOCAL**) or LOR (for **LOCAL OVER REMOTE**):

LOCAL STAMP	REMOTE STAMP	Priority	LOCAL after sync	REMOTE after sync	LOCAL - REMOTE Synchronization
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	No synchronization
31	4001	LOR	BBB	CCC	No synchronization
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	No synchronization
40	5000	LOR	BBB	CCC	No synchronization

Syntax rules

Syntax rules

-  4d_function_call
-  4d_language_reference
-  all_or_any_predicate
-  arithmetic_expression
-  between_predicate
-  case_expression
-  column_definition
-  column_reference
-  command_parameter
-  comparison_predicate
-  exists_predicate
-  foreign_key_definition
-  function_call
-  in_predicate
-  is_null_predicate
-  like_predicate
-  literal
-  predicate
-  primary_key_definition
-  search_condition
-  select_item
-  sort_list
-  sql_data_type_name
-  sql_name
-  sql_string
-  subquery
-  table_constraint
-  table_reference

Syntax rules

The syntax rules describe the various elements of the predicates used in SQL statements. These have been separated into individual items whenever possible and described as simply as possible to give a general indication of their use within 4D. Keywords (in bold) are always passed "as is" when used.

4d_function_call

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

Description

A *4d_function_call* can be used to execute a 4D function that returns a value.

The *sql_name* of the function is preceded by the **FN** keyword and followed by one or more *arithmetic_expression* type arguments. The value returned by the function will be of the type defined by the *sql_data_type_name* passed.

Example

Here is an example using functions to extract from the MOVIES table the number of actors for each movie having at least 7 actors:

```
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

Description

A *4d_language_reference* argument specifies the 4D variable or field name (*sql_name*) to which data will be assigned. This name can be passed in one of the following manners:

<<sql_name>>

<<\$sql_name>> (*)

<<[sql_name]sql_name>> (corresponds to the standard 4D syntax: [TableName]FieldName)

:sql_name

:\$sql_name (*)

:sql_name.sql_name (corresponds to the standard SQL syntax: TableName.FieldName)

(*) In compiled mode, you cannot use references to local variables (beginning with the \$ symbol).

all_or_any_predicate

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

Description

An *all_or_any_predicate* is used to compare an *arithmetic_expression* with a *subquery*. You can pass comparison operators like <, <=, =, >=, > or <> as well as the **ANY**, **ALL** and **SOME** keywords along with the *subquery* to be used for comparison.

Beispiel

This example carries out a subquery which selects the best software sales. The main query selects records from the SALES and CUSTOMERS tables where the Total_value column is greater than the records selected by the subquery:

```
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 |

Description

An *arithmetic_expression* may contain a *literal* value, a *column_reference*, a *function_call*, a *command_parameter* or a *case_expression*. You can also pass combinations of *arithmetic_expression(s)* using the *+*, *-*, *** or */* operators.

between_predicate

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

Description

A *between_predicate* is used to find data with values that fall within two other *arithmetic_expression* values (passed in ascending order). You can also pass the optional **NOT** keyword to excludes values falling within these limits.

Example

Here is a simple example which returns the names of all the clients whose first name starts with a letter between A and E:

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

case_expression

case_expression

Beschreibung

A *case_expression* is used to apply one or more conditions when selecting an expression. They can be used as follows, for example:

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

Or:

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

Beispiel

This example selects records from the ROOM_NUMBER column according to the value of the ROOM_FLOOR column:

```
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**]

Description

A *column_definition* contains the name (*sql_name*) and data type (*sql_data_type_name*) of a column. You can also pass an optional *int_number* as well as the **NOT NULL**, **UNIQUE**, **AUTO_INCREMENT** and/or **AUTO_GENERATE** keywords.

- Passing **NOT NULL** in the *column_definition* means that the column will not accept null values.
- Passing **UNIQUE** means that the same value may not be inserted into this column twice. Note that only **NOT NULL** columns can have the **UNIQUE** attribute. The **UNIQUE** keyword must always be preceded by **NOT NULL**.
- Passing **AUTO_INCREMENT** means that the column will generate a unique number for each new row. This attribute can only be used with number columns.
- Passing **AUTO_GENERATE** means that a UUID will be generated automatically in the column for each new row. This attribute can only be used with UUID columns.

Each column must have a data type. The column should either be defined as "null" or "not null" and if this value is left blank, the database assumes "null" as the default. The data type for the column does not restrict what data may be put in that column.

Example

Here is a simple example which creates a table with two columns (ID and Name):

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

column_reference

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

Description

A *column_reference* consists of an *sql_name* or *sql_string* passed in one of the following manners:

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

Beschreibung

A *command_parameter* may consist of a question mark (?) or an *sql_name* passed in one of the following forms:

?

<<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*

Description

A *comparison_predicate* uses operators like <, <=, =, >=, > or <> to compare two *arithmetic_expression* type arguments or to compare an *arithmetic_expression* with a *subquery* as part of a *search_condition* applied to the data.

exists_predicate

EXISTS (*subquery*)

Description

An *exists_predicate* is used to indicate a *subquery* and then check whether it returns anything. This is done by passing the **EXISTS** keyword followed by the *subquery*.

Beispiel

This example returns the total sales when there is a store in the region specified:

```
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**}]

Description

A *foreign_key_definition* is used to match the primary key fields (*column_reference*) set in another table in order to ensure data integrity. The **FOREIGN KEY** constraint is used to pass the one or more column references (*column_reference*) to be defined as the foreign keys (which match the primary keys of another table).

The **CONSTRAINT** *sql_name* clause is used to name the **FOREIGN KEY** constraint.

The **REFERENCES** clause that follows is used to specify the matching primary key field sources in another table (*sql_name*). You can omit the list of *column_reference* type arguments if the table (*sql_name*) specified in the **REFERENCES** clause has a primary key that is to be used as the matching key for the foreign key constraint.

The optional **ON DELETE CASCADE** clause specifies that when a row is deleted from the parent table (containing the primary key fields), it is also removed from any rows associated with that row in the child table (containing the foreign key fields). Passing the optional **ON DELETE RESTRICT** clause prevents any data from being deleted from a table if any other tables reference it.

The optional **ON UPDATE CASCADE** clause specifies that whenever a row is updated in the parent table (containing the primary key fields), it is also updated in any rows associated with that row in the child table (containing the foreign key fields). Passing the optional **ON UPDATE RESTRICT** clause prevents any data from being updated in a table if any other tables reference it.

Note that if both the **ON DELETE** and **ON UPDATE** clauses are passed, they must both be of the same type (e.g. **ON DELETE CASCADE** with **ON UPDATE CASCADE**, or **ON DELETE RESTRICT** with **ON UPDATE RESTRICT**).

If neither the **ON DELETE** nor the **ON UPDATE** clause is passed, then **CASCADE** is used as the default rule.

Example

This example creates the ORDERS table then sets the Customer_SID column as the foreign key, associated with the SID column of the CUSTOMERS table:

```
CREATE TABLE ORDERS
(Order_ID INT32,
Customer_SID INT32,
Amount NUMERIC,
PRIMARY KEY (Order_ID),
CONSTRAINT fk_1 FOREIGN KEY (Customer_SID) REFERENCES CUSTOMER(SID));
```

function_call

sql_function_call |
4d_function_call

Beschreibung

A *function_call* can consist of a call to either **SQL Functions** or a 4D function (*4d_function_call*). Both types of functions manipulate data and return results and can operate on one or more arguments.

Example

This example uses the SQL *COUNT* function:

```
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*)

Description

An *in_predicate* is used to compare an *arithmetic_expression* to check whether it is included (or **NOT** included if this keyword is also passed) in a list of values. The list of values used for the comparison can either be a sequence of arithmetic expressions that are passed or the result of a *subquery*.

Beispiel

This example selects the records of the ORDERS table whose order_id column value is equal to 10000, 10001, 10003 or 10005:

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

is_null_predicate

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

Description

An *is_null_predicate* is used to find an *arithmetic_expression* with a **NULL** value. You can also pass the **NOT** keyword to find those without **NULL** values.

Beispiel

This example selects products whose weight is less than 15 or whose Color column contains a NULL value:

```
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*]

Description

A *like_predicate* is used to retrieve data matching the *arithmetic_expression* passed after the **LIKE** keyword. You can also pass the **NOT** keyword to search for data differing from this expression. The **ESCAPE** keyword can be used to prevent the character passed in *sql_string* from being interpreted as a wildcard. It is usually used when you want to search for the '%' or '_' characters.

Beispiel 1

This example selects the suppliers whose name contains "bob":

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

Beispiel 2

Selects suppliers whose name does not begin with the letter T:

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

Beispiel 3

Selects suppliers whose name begins with "Sm" and ends with "th":

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

literal

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

Description

A *literal* is a data type consisting of either an *int_number* (integer), a *fractional_number* (fraction), an *sql_string* or a *hexadecimal_number*.

Hexadecimal notation (introduced in 4D 12.1) can express any type of data represented as bytes. A byte is always defined by two hexadecimal values. To indicate the use of this notation in an SQL command, you must simply use the standard SQL syntax for hexadecimal values:

X'<hexadecimal value>'

For example, for the decimal value 15, you can write **X'0f'**. You can set a blank value (zero byte) by writing **X''**.

Note: The **SQL EXPORT DATABASE** and **SQL EXPORT SELECTION** commands export binary data in hexadecimal format when these data are embedded in the main export file.

predicate

predicate

Beschreibung

A *predicate* follows the **WHERE** clause and is used to apply conditions for searching the data. It can be one of the following types:

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*)

Description

A *primary_key_definition* is used to pass the column or combination of columns (*sql_name*) that will serve as the **PRIMARY KEY** (unique ID) for the table. The column(s) passed must not contain duplicate or **NULL** values.

An optional **CONSTRAINT** can also precede the **PRIMARY KEY** passed in order to limit the values that can be inserted into the column.

Example

This example creates a table and sets the SID column as the primary key:

```
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* |

Description

A *search_condition* specifies a condition to be applied to the data retrieved. A combination of search conditions using **AND** or **OR** keywords can also be applied. You can also precede a *search_condition* with the **NOT** keyword in order to retrieve data that does not meet the specified condition.

It is also possible to pass a *predicate* as a *search_condition*.

Example

Here is an example using a combination of search conditions in the **WHERE** clause:

```
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*}]

Description

A *select_item* specifies one or more items to be included in the results. A column is generated for every *select_item* passed. Each *select_item* consists of an *arithmetic_expression*. You can also pass the optional **AS** keyword to specify the optional *sql_string* or *sql_name* to be given to the column. (Passing the optional *sql_string* or *sql_name* without the **AS** keyword has the same effect).

Example

Here is an example which creates a column named `Movie_Year` containing movies released in the year 2000 or more recently:

```
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]`

Description

A *sort_list* contains either a *column_reference* or an *int_number* indicating the column where the sort will be applied. You can also pass the **ASC** or **DESC** keyword to specify whether the sort will be in ascending or descending order. By default, the sort will be in ascending order.

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**

Beschreibung

An *sql_data_type_name* follows the **AS** keyword in a *4d_function_call* and can have one of the following values:

**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

Description

An *sql_name* is either a standard SQL name starting with a Latin alphabet character and that contains only Latin characters, numbers and/or underscores, or a square-bracketed string. The right square bracket is escaped by doubling.

Examples:

String to pass	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

Description

An *sql_string* contains a single-quoted string. Single quote characters that are located inside a string are doubled and strings that are already single-quoted are double-quoted before being placed within another pair of single quotes.

Examples:

String to pass	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]
```

Beschreibung

A *subquery* is like a separate *SELECT* statement enclosed in parentheses and passed in the predicate of another SQL statement (*SELECT*, **INSERT**, *UPDATE* or *DELETE*). It acts as a query within a query and is often passed as part of a **WHERE** or **HAVING** clause.

table_constraint

{primary_key_definition | foreign_key_definition}

Description

A *table_constraint* restricts the values that a table can store. You can either pass a *primary_key_definition* or a *foreign_key_definition*. The *primary_key_definition* sets the primary key for the table and the *foreign_key_definition* is used to set the foreign key (which matches the primary key of another table).


table_reference

`{sql_name | sql_string} [[AS] {sql_name|sql_string}]`

Description

A *table_reference* can be either a standard SQL name or a string. You may also pass the optional **AS** keyword to assign an alias (in the form of an *sql_name* or *sql_string*) to the column. (Passing the optional *sql_string* or *sql_name* without the **AS** keyword has the same effect).

Transactions

 Transactions

 START

 COMMIT

 ROLLBACK

🔌 Transactions

Description

Transactions are a set of SQL statements that are executed together. Either all of them are successful or they have no effect. Transactions use locks to preserve data integrity during their execution. If the transaction finishes successfully, you can use the *COMMIT* statement to permanently store its modifications. Otherwise, using the *ROLLBACK* statement will cancel any modifications and restore the database to its previous state.

There is no difference between a 4D transaction and an SQL transaction. Both types share the same data and process. SQL statements passed between **Begin SQL/End SQL** tags, the **QUERY BY SQL** and the integrated generic SQL commands applied to the local database are always executed in the same context as standard 4D commands.

Note: 4D provides an "Auto-commit" option which can be used to start and validate transactions automatically when using SIUD commands (*SELECT*, *UPDATE* and *DELETE*) in order to preserve data integrity. For more information, please refer to the **Principles for integrating 4D and the 4D SQL engine** section.

The following examples illustrate the different combinations of transactions.

Neither "John" nor "Smith" will be added to the emp table:

```
SQL LOGIN(SQL_INTERNAL;"";"" ) `Initializes the 4D SQL engine
START TRANSACTION ` Starts a transaction in the current process
Begin SQL
  INSERT INTO emp
  (NAME)
  VALUES ('John');
End SQL
SQL EXECUTE("START") ` Another transaction in the current process
SQL CANCEL LOAD
SQL EXECUTE("INSERT INTO emp (NAME) VALUES ('Smith')") ` This statement is executed in the same
process
SQL CANCEL LOAD
SQL EXECUTE("ROLLBACK")<gen9> ` Cancels internal transaction of the pro-cess
CANCEL TRANSACTION
` Cancels external transaction of the process
SQL LOGOUT</gen9>
```

Only "John" will be added to the emp table:

```
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")<gen9> ` Cancels internal transaction of the pro-cess
```

```
VALIDATE TRANSACTION `Validates external transaction of the process  
SQL LOGOUT</gen9>
```

Neither "John" nor "Smith" will be added to the emp table. The external transaction cancels the internal transaction:

```
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") `Validates internal transaction of the process  
CANCEL TRANSACTION `Cancels external transaction of the process  
SQL LOGOUT
```

"John" and "Smith" will be added to the emp table:

```
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") `Validates internal transaction of the process  
VALIDATE TRANSACTION `Validates external transaction of the process  
SQL LOGOUT
```

START

START [TRANSACTION]

Description

The *START* command is used to set the beginning of a transaction. If this command is passed when a transaction is already underway, it starts a subtransaction. The keyword **TRANSACTION** is optional.

Example

This example carries out a selection within a transaction:

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

COMMIT

COMMIT [TRANSACTION]

Description

The *COMMIT* command sets the end of a successful transaction. It ensures that all the modifications made by the transaction become a permanent part of the database. It also frees any resources used by the transaction. Keep in mind that you cannot use a *ROLLBACK* statement after a *COMMIT* command since the changes have been made permanent. Passing the keyword **TRANSACTION** is optional.

Example

See the example for the *START* command.

ROLLBACK

ROLLBACK [TRANSACTION]

Description





















































The *ROLLBACK* command cancels the transaction underway and restores the data to its previous state at the beginning of the transaction. It also frees up any resources held by the transaction. The **TRANSACTION** keyword is optional.




























Example

This example illustrates the use of the *ROLLBACK* command:

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

Functions

-  SQL Functions
 -  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 Functions

SQL Functions work with column data in order to produce a specific result in 4D. Function names appear in bold and are passed as is, generally followed by one or more *arithmetic_expression* type arguments.

ABS

ABS (*arithmetic_expression*)

Description

The **ABS** function returns the absolute value of the *arithmetic_expression*.

Beispiel

This example returns the absolute value of the prices and multiplies them by a given quantity:

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

ACOS

ACOS (*arithmetic_expression*)

Beschreibung

The **ACOS** function returns the arc cosine of the *arithmetic_expression*. It is the inverse of the **COS** function. The *arithmetic_expression* represents the angle expressed in radians.

Beispiel

This example will return the arc cosine of the angle expressed in radians (-0.73):

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

ASCII

ASCII (*arithmetic_expression*)

Beschreibung

The *ASCII* function returns the leftmost character of the *arithmetic_expression* as an integer. If the *arithmetic_expression* is null, the function will return a **NULL** value.

Beispiel

This example returns the first letter of each last name as an integer:

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

ASIN

ASIN (*arithmetic_expression*)

Beschreibung

The **ASIN** function returns the arc sine of the *arithmetic_expression*. It is the inverse of the sine (**SIN**) function. The *arithmetic_expression* represents the angle expressed in radians.

Beispiel

This example will return the arc sine of the angle expressed in radians (-0.73):

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

ATAN

ATAN (*arithmetic_expression*)

Description

The **ATAN** function returns the arc tangent of the *arithmetic_expression*. It is the inverse of the tangent (**TAN**) function. The *arithmetic_expression* represents the angle expressed in radians.

Beispiel

This example will return the arc tangent of the angle expressed in radians (-0.73):

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

ATAN2

ATAN2 (*arithmetic_expression*, *arithmetic_expression*)

Beschreibung

The *ATAN2* function returns the arc tangent of the "x" and "y" coordinates, where "x" is the first *arithmetic_expression* passed and "y" is the second one.

Beispiel

This example returns the arc tangent of the x and y coordinates passed (0.52 and 0.60 respectively):

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


AVG

AVG ([**ALL** | **DISTINCT**] *arithmetic_expression*)

Description

The *AVG* function returns the average of the *arithmetic_expression*. The optional **ALL** and **DISTINCT** keywords are used to respectively retain or eliminate any duplicate values.

Beispiel

This example returns the minimum value of tickets sold, the maximum value of tickets sold, the average of the tickets sold and the total amount of tickets sold for the MOVIES table:

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

BIT_LENGTH

BIT_LENGTH (*arithmetic_expression*)

Description

The *BIT_LENGTH* function returns the length of the *arithmetic_expression* in bits.

Beispiel

This example returns 8:

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

CAST

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

Beschreibung

The **CAST** function converts the *arithmetic_expression* to the *sql_data_type_name* passed following the **AS** keyword.

Note: The **CAST** function is not compatible with "Integer 64 bits" type fields in compiled mode.

Beispiel

This example converts the year of the movie into an Integer type:

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

CEILING

CEILING (*arithmetic_expression*)

Description

The *CEILING* function returns the smallest integer that is greater than or equal to the *arithmetic_expression*.

Beispiel

This example returns the smallest integer greater than or equal to -20.9:

```
CEILING (-20.9)
`returns -20
```

CHAR

CHAR (*arithmetic_expression*)

Beschreibung

The **CHAR** function returns a fixed-length character string based on the type of the *arithmetic_expression* passed.

Beispiel

This example returns a character string based on the integer of the first letter of each last name:

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

CHAR_LENGTH

CHAR_LENGTH (*arithmetic_expression*)

Description

The *CHAR_LENGTH* function returns the number of characters in the *arithmetic_expression*.

Example

This example returns the number of characters in the name of products where the weight is less than 15 lbs.

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

COALESCE

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

Beschreibung

The *COALESCE* function returns the first non-null expression from the list of *arithmetic_expression* type arguments passed. It will return **NULL** if all the expressions passed are NULL.

Example

This example returns all the invoice numbers from 2007 where the VAT is greater than 0:

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

CONCAT

CONCAT (*arithmetic_expression*, *arithmetic_expression*)

Description

The *CONCAT* function returns the two *arithmetic_expression* type arguments passed as a single concatenated string.

Example

This example will return the first name and last name as a single string:

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


CONCATENATE

CONCATENATE (*arithmetic_expression*, *arithmetic_expression*)

Description

The *CONCATENATE* function returns the two *arithmetic_expression* type arguments passed as a single concatenated string.

Example

See the example for the [CONCAT](#) function.

COS

COS (*arithmetic_expression*)

Description

The **COS** function returns the cosine of the *arithmetic_expression*. The *arithmetic_expression* represents the angle expressed in radians.

Beispiel

This example will return the cosine of the angle expressed in radians (degrees * 180 / 3,1416):

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

COT

COT (*arithmetic_expression*)

Description

The *COT* function returns the cotangent of the *arithmetic_expression*. The *arithmetic_expression* represents the angle expressed in radians.

Example

This example will return the cotangent of the angle expressed in radians (3,1416):

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

COUNT

COUNT ({ [**ALL** | **DISTINCT**] *arithmetic_expression* [*] })

Beschreibung

The *COUNT* function returns the number of non-null values in the *arithmetic_expression*. The optional **ALL** and **DISTINCT** keywords are used to respectively retain or eliminate any duplicate values.

If you pass the * instead, the function returns the total number of records in the *arithmetic_expression*, including duplicate and NULL values.

Example

This example returns the number of movies from the MOVIES table:

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

CURDATE

CURDATE ()

Beschreibung

The *CURDATE* function returns the current date.

Beispiel

This example creates a table of invoices and inserts the current date into the *INV_DATE* column:

```
ARRAY TEXT(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
```

```
` the aDate array will return the INSERT command execution date and time.
```

CURRENT_DATE

CURRENT_DATE ()

Description

The *CURRENT_DATE* function returns the current date in local time.

Example

This example creates a table of invoices and inserts the current date into the *INV_DATE* column:

```
ARRAY TEXT(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
```

```
` the aDate array will return the INSERT command execution date and time.
```

CURRENT_TIME

CURRENT_TIME ()

Description

The *CURRENT_TIME* function returns the current local time.

Example

This example creates a table of invoices and inserts the current time into the INV_DATE column:

```
ARRAY TEXT(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
```

```
` the aDate array will return the INSERT command execution date and time.
```

CURRENT_TIMESTAMP

CURRENT_TIMESTAMP ()

Description

The *CURRENT_TIMESTAMP* function returns the current date and local time.

Example

This example creates a table of invoices and inserts the current date and time into the *INV_DATE* column:

```
ARRAY TEXT(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
```

```
` the aDate array will return the INSERT command execution date and time.
```


CURTIME ()

Description

The **CURTIME** function returns the current time to a precision of one second.

Example

This example creates a table of invoices and inserts the current time into the INV_DATE column:

```
ARRAY TEXT(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
```

```
//the aDate array will return the INSERT command execution date and time.
```

DATABASE_PATH

DATABASE_PATH()

Beschreibung

The **DATABASE_PATH** function returns the complete pathname of the current database. The current database can be modified using the **USE DATABASE** command. By default, the current database is the main 4D database.

The pathname returned is in the POSIX format.

Beispiel

Let us suppose that the current external database is named TestBase.4DB and is located in the "C:\MyDatabases" folder. After execution of the following code:

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

... the *\$vCrtDatabasePath* variable will contain "C:/MyDatabases/TestBase.4DB".

DATE_TO_CHAR

DATE_TO_CHAR (*arithmetic_expression*, *arithmetic_expression*)

Description

The *DATE_TO_CHAR* function returns a text representation of the date passed in the first *arithmetic_expression* according to the format specified in the second *arithmetic_expression*. The first *arithmetic_expression* should be of the Timestamp or Duration type and the second should be of the Text type.

The formatting flags which can be used are given below. In general, if a formatting flag starts with an upper-case character and produces a zero, then the number will start with one or more zeros when appropriate; otherwise, there will be no leading zeros. For example, if *dd* returns 7, then *Dd* will return 07.

The use of upper- and lower-case characters in the formatting flags for day and month names will be reproduced in the results returned. For example, passing "day" will return "monday", passing "Day" will return "Monday" and passing "DAY" will return "MONDAY".

am - am or pm according to the value of the hour

pm - am or pm according to the value of the hour

a.m. - a.m. or p.m. according to the value of the hour

p.m. - a.m. or p.m. according to the value of the hour

d - numeric day of week (1-7)

dd - numeric day of month (1-31)

ddd - numeric day of year

day - name of day of week

dy - abbreviated 3-letter name of day of week

hh - numeric hour, 12-based (0-11)

hh12 - numeric hour, 12-based (0-11)

hh24 - numeric hour, 24-based (0-23)

J - Julian day

mi - minutes (0-59)

mm - numeric month (0-12)

q - year's quarter

ss - seconds (0-59)

sss - milliseconds (0-999)

w - week of month (1-5)

ww - week of year (1-53)

yy - year

yyyy - year

[any text] - text inside brackets ([]) is not interpreted and inserted as is

-.,:; 'space character' 'tab character' - are left as is, without changes.

Example

This example returns the birth date as a numeric day of the week (1-7):

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

DAY

DAY (*arithmetic_expression*)

Description

The *DAY* function returns the day of the month for the date passed in the *arithmetic_expression*.

Example

This example returns the day of the month for the date "05-07-2007":

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

DAYNAME

DAYNAME (*arithmetic_expression*)

Description

The *DAYNAME* function returns the name of the day of the week for the date passed in the *arithmetic_expression*.

Example

This example returns the name of the day of the week for each date of birth passed:

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

DAYOFMONTH

DAYOFMONTH (*arithmetic_expression*)

Description

The *DAYOFMONTH* function returns a number representing the day of the month (ranging from 1 to 31) of the date passed in the *arithmetic_expression*.

Example

Supposing that we have the PEOPLE table with a Date_of_Birth field. To find out the day number of the date of birth for every person in PEOPLE:

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

DAYOFWEEK

DAYOFWEEK (*arithmetic_expression*)

Description

The *DAYOFWEEK* function returns a number representing the day of the week (ranging from 1 to 7, where 1 is Sunday and 7 is Saturday) of the date passed in the *arithmetic_expression*.

Example

Supposing that we have the PEOPLE table with a Date_of_Birth field. To find out the day of the week of the date of birth for every person in PEOPLE :

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

DAYOFYEAR

DAYOFYEAR (*arithmetic_expression*)

Description

The *DAYOFYEAR* function returns a number representing the day of the year (ranging from 1 to 366, where 1 is January 1st) of the date passed in the *arithmetic_expression*.

Example

Supposing that we have the PEOPLE table with a Date_of_Birth field. To find out the day of the year of the date of birth for every person in PEOPLE:

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


DEGREES

DEGREES (*arithmetic_expression*)

Description

The *DEGREES* function returns the number of degrees of the *arithmetic_expression*. The *arithmetic_expression* represents the angle expressed in radians.

Example

This example will create a table and insert values based on the numbers of degrees of the value Pi:

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

EXP

EXP (*arithmetic_expression*)

Description

The **EXP** function returns the exponential value of the *arithmetic_expression*, e.g. e raised to the xth value where "x" is the value passed in the *arithmetic_expression*.

Example

This example returns e raised to the 15th value:

```
SELECT EXP( 15 ); `returns 3269017,3724721107
```

EXTRACT

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

Beschreibung

The *EXTRACT* function returns the specified part which it extracts from the *arithmetic_expression*. The *arithmetic_expression* passed should be of the Timestamp type.

Example

This example returns all the invoice numbers from the month of January:

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

FLOOR

FLOOR (*arithmetic_expression*)

Description

The *FLOOR* function returns the largest integer that is less than or equal to the *arithmetic_expression*.

Example

This example returns the largest integer less than or equal to -20.9:

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

HOUR

HOUR (*arithmetic_expression*)

Description

The *HOUR* function returns the hour part of the time passed in the *arithmetic_expression*. The value returned ranges from 0 to 23.

Example

Supposing that we have the INVOICES table with a Delivery_Time field. To display the hour of the delivery time:

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

INSERT

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

Beschreibung

The **INSERT** function inserts one string into another at a given position. The first *arithmetic_expression* passed is the destination string. The second *arithmetic_expression* is the index where the string passed in the fourth *arithmetic_expression* will be inserted and the third *arithmetic_expression* gives the number of characters to be removed at the given insertion point.

Example

This example will insert "Dear " in front of the first names in the PEOPLE table:

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

LEFT

LEFT (*arithmetic_expression*, *arithmetic_expression*)

Beschreibung

The *LEFT* function returns the leftmost part of the *arithmetic_expression* passed. The second *arithmetic_expression* indicates the number of leftmost characters to return as extracted from the first *arithmetic_expression* indicated.

Beispiel

This example returns the first names and first two letters of the last names from the PEOPLE table:

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

LENGTH

LENGTH (*arithmetic_expression*)

Beschreibung

The **LENGTH** function returns the number of characters in the *arithmetic_expression*.

Beispiel

This example returns the number of characters in the name of products that weigh less than 15 lbs.

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


LOCATE

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

LOCATE (*arithmetic_expression*, *arithmetic_expression*)

Beschreibung

The *LOCATE* function returns the starting position of the 1st occurrence of an *arithmetic_expression* found within a second *arithmetic_expression*. You can also pass a third *arithmetic_expression* to specify the character position where the search must begin.

Beispiel

This example will return the position of the first letter X found in the last names of the PEOPLE table:

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

LOG

LOG (*arithmetic_expression*)

Description

The **LOG** function returns the natural logarithm of the *arithmetic_expression*.

Beispiel

This example returns the natural logarithm of 50:

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

LOG10

LOG10 (*arithmetic_expression*)

Description

The *LOG10* function returns the base 10 logarithm of the *arithmetic_expression*.

Beispiel

This example returns the logarithm in base 10 of 50:

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

LOWER

LOWER (*arithmetic_expression*)

Description

The *LOWER* function returns the *arithmetic_expression* passed as a string where all the characters are in lowercase.

Beispiel

This example will return the names of products in lowercase:

```
SELECT LOWER (Name)
FROM PRODUCTS;
```

LTRIM

LTRIM (*arithmetic_expression* [, *arithmetic_expression*])

Description

The *LTRIM* function removes any empty spaces from the beginning of the *arithmetic_expression*. The optional second *arithmetic_expression* can be used to indicate specific characters to be removed from the beginning of the first *arithmetic_expression*.

Example

This example simply removes any empty spaces from the beginning of product names:

```
SELECT LTRIM(Name)
FROM PRODUCTS;
```

MAX

MAX (*arithmetic_expression*)

Description

The **MAX** function returns the maximum value of the *arithmetic_expression*.

Beispiel

See the examples from **SUM** and *AVG*.

MILLISECOND

MILLISECOND (*arithmetic_expression*)

Beschreibung

The *MILLISECOND* function returns the millisecond part of the time passed in *arithmetic_expression*.

Beispiel

Supposing that we have the INVOICES table with a Delivery_Time field. To display the milliseconds of the delivery time:

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

MIN

MIN (*arithmetic_expression*)

Description

The **MIN** function returns the minimum value of the *arithmetic_expression*.

Example

See the examples from **SUM** and *AVG*.

MINUTE

MINUTE (*arithmetic_expression*)

Description

The *MINUTE* function returns the minute part of the time passed in the *arithmetic_expression*. The value returned ranges from 0 to 59.

Example

Supposing that we have the INVOICES table with a Delivery_Time field. To display the minute of the delivery time:

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

MOD

MOD (*arithmetic_expression*, *arithmetic_expression*)

Beschreibung

The **MOD** function returns the remainder of the first *arithmetic_expression* divided by the second *arithmetic_expression*.

Example

This example returns the remainder of 10 divided by 3:

```
MOD(10,3) `returns 1
```

MONTH

MONTH (*arithmetic_expression*)

Description

The *MONTH* function returns the number of the month (ranging from 1 to 12) of the date passed in the *arithmetic_expression*.

Example

Supposing that we have the PEOPLE table with a Date_of_Birth field. To find out the month of the date of birth for every person in PEOPLE :

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

MONTHNAME

MONTHNAME (*arithmetic_expression*)

Description

The *MONTHNAME* function returns the name of the month for the date passed in the *arithmetic_expression*.

Example

This example returns the name of the month for each date of birth passed:

```
SELECT MONTHNAME(Date_of_birth);
```

NULLIF

NULLIF (*arithmetic_expression*, *arithmetic_expression*)

Beschreibung

The *NULLIF* function returns NULL if the first *arithmetic_expression* is equal to the second *arithmetic_expression*. Otherwise, it will return the value of the first *arithmetic_expression*. The two expressions must be comparable.

Example

This example returns NULL if the total of the invoice is 0:

```
NULLIF(INVOICE_TOTAL,0);
```

OCTET_LENGTH

OCTET_LENGTH (*arithmetic_expression*)

Beschreibung

The *OCTET_LENGTH* function returns the number of bytes of the *arithmetic_expression*, including any trailing whitespace.

Beispiel

This example returns the number of octets for a column comprised of binary data:

```
SELECT OCTET_LENGTH (MyBinary_col)
FROM MyTable
WHERE MyBinary_col = '93FB';
` returns 2
```

PI

PI ()

Beschreibung

The **PI** function returns the value of the constant Pi.

Example

See example from *DEGREES*.

POSITION

POSITION (*arithmetic_expression* **IN** *arithmetic_expression*)

Description

The **POSITION** function returns a value indicating the starting position of the first *arithmetic_expression* within the second *arithmetic_expression*. If the string is not found, the function returns zero.

Beispiel

This example will return the starting position of the word "York" in any last names of the PEOPLE table:

```
SELECT FirstName, POSITION('York' IN LastName)
FROM PEOPLE;
```


POWER

POWER (*arithmetic_expression*, *arithmetic_expression*)

Description

The *POWER* function raises the first *arithmetic_expression* passed to the power of "x", where "x" is the second *arithmetic_expression* passed.

Example

This example raises each value to the power of 3:

```
SELECT SourceValues, POWER(SourceValues, 3)
FROM Values
ORDER BY SourceValues
`returns 8 for SourceValues = 2
```

QUARTER

QUARTER (*arithmetic_expression*)

Description

The *QUARTER* function returns the quarter of the year (ranging from 1 to 4) in which the date passed in the *arithmetic_expression* occurs.

Example

Supposing that we have the PEOPLE table with a Date_of_Birth field. To find out the quarter of the date of birth for every person in PEOPLE:

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

RADIANS

RADIANS (*arithmetic_expression*)

Beschreibung

The *RADIANS* function returns the number of radians of the *arithmetic_expression*. The *arithmetic_expression* represents the angle expressed in degrees.

Beispiel

This example returns the number of radians of a 30 degree angle:

```
RADIANS ( 30 );  
` returns the value 0,5236
```

RAND

RAND (*[arithmetic_expression]*)

Beschreibung

The **RAND** function returns a random Float value between 0 and 1. The optional *arithmetic_expression* can be used to pass a seed value.

Example

This example inserts ID values generated by the **RAND** function:

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

INSERT INTO PEOPLE
(ID, Name)
VALUES(RAND, 'Francis');
```

REPEAT

REPEAT (*arithmetic_expression*, *arithmetic_expression*)

Beschreibung

The *REPEAT* function returns the first *arithmetic_expression* repeated the requested number of times (passed in second *arithmetic_expression*).

Beispiel

This example illustrates how it works:

```
SELECT REPEAT( 'repeat', 3 )
` returns 'repeatrepeatrepeat'
```

REPLACE

REPLACE (*arithmetic_expression*, *arithmetic_expression*, *arithmetic_expression*)

Beschreibung

The *REPLACE* function looks in the first *arithmetic_expression* passed for all the occurrences of the second *arithmetic_expression* passed and replaces each one found with the third *arithmetic_expression* passed. If no such occurrences are found, the first *arithmetic_expression* remains unchanged.

Beispiel

This example will replace the word "Francs" by "Euro":

```
SELECT Name, REPLACE(Currency, 'Francs', 'Euro')
FROM PRODUCTS;
```

RIGHT

RIGHT (*arithmetic_expression*, *arithmetic_expression*)

Description

The *RIGHT* function returns the rightmost part of the *arithmetic_expression* passed. The second *arithmetic_expression* indicates the number of rightmost characters to return as extracted from the first *arithmetic_expression* indicated.

Example

This example returns the first names and the last two letters of the last names from the PEOPLE table:

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

ROUND

ROUND (*arithmetic_expression* [, *arithmetic_expression*])

Description

The **ROUND** function rounds the first *arithmetic_expression* passed to "x" decimal places (where "x" is the second optional *arithmetic_expression* passed). If the second *arithmetic_expression* is not passed, the *arithmetic_expression* is rounded off to the nearest whole number.

Example

This example rounds the given number off to two decimal places:

```
ROUND (1234.1966, 2)
`returns 1234.2000
```


RTRIM

RTRIM (*arithmetic_expression*[, *arithmetic_expression*])

Beschreibung

The *RTRIM* function removes any empty spaces from the end of the *arithmetic_expression*. The optional second *arithmetic_expression* can be used to indicate specific characters to be removed from the end of the first *arithmetic_expression*.

Beispiel

This example removes any empty spaces from the ends of the product names:

```
SELECT RTRIM(Name)
FROM PRODUCTS;
```

SECOND

SECOND (*arithmetic_expression*)

Beschreibung

The *SECOND* function returns the seconds part (ranging from 0 to 59) of the time passed in the *arithmetic_expression*.

Example

Supposing that we have the INVOICES table with a Delivery_Time field. To display the seconds of the delivery time:

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

SIGN

SIGN (*arithmetic_expression*)

Beschreibung

The *SIGN* function returns the sign of the *arithmetic_expression* (e.g., 1 for a positive number, -1 for a negative number or 0).

Example

This example will returns all the negative amounts found in the INVOICES table:

```
SELECT AMOUNT
FROM INVOICES
WHERE SIGN(AMOUNT) = -1;
```

SIN

SIN (*arithmetic_expression*)

Description

The **SIN** function returns the sine of the *arithmetic_expression*. The *arithmetic_expression* represents the angle expressed in radians.

Example

This example will return the sine of the angle expressed in radians:

```
SELECT SIN(radians)
FROM TABLES_OF_ANGLES;
```

SPACE

SPACE (*arithmetic_expression*)

Beschreibung

The *SPACE* function returns a character string made up of the given number of spaces indicated in *arithmetic_expression*. If the value of the *arithmetic_expression* is less than zero, a NULL value will be returned.

Beispiel

This example adds three spaces in front of the last names of the PEOPLE table:

```
SELECT CONCAT(SPACE(3),PERSONS.LastName) FROM PEOPLE;
```

SQRT

SQRT (*arithmetic_expression*)

Description

The **SQRT** function returns the square root of the *arithmetic_expression*.

Example

This example returns the square root of the freight:

```
SELECT Freight, SQRT(Freight) AS "Square root of Freight"  
FROM Orders
```

SUBSTRING

SUBSTRING (*arithmetic_expression*, *arithmetic_expression*, [*arithmetic_expression*])

Description

The **SUBSTRING** function returns a substring of the first *arithmetic_expression* passed. The second *arithmetic_expression* indicates the starting position of the substring and the optional third *arithmetic_expression* indicates the number of characters to return counting from the starting position indicated. If the third *arithmetic_expression* is not passed, the function will return all the characters starting from the position indicated.

Beispiel

This example will return 4 characters of the store name starting with the 2nd character:

```
SELECT SUBSTRING(Store_name,2,4)
FROM Geography
WHERE Store_name = 'Paris';
```

SUM

SUM ([**ALL** | **DISTINCT**] *arithmetic_expression*)

Description

The **SUM** function returns the sum of the *arithmetic_expression*. The optional **ALL** and **DISTINCT** keywords are used to respectively retain or eliminate any duplicate values.

Beispiel

This example returns the sum of the expected sales less the sum of the actual sales, as well as the minimum and maximum value of the actual sales multiplied by 100 and divided by the expected sales for the SALES_PERSONS table:

```
SELECT MIN ( ( SALES * 100 ) / QUOTA ),  
MAX ( ( SALES * 100 ) / QUOTA ),  
SUM ( QUOTA ) - SUM ( SALES )  
FROM SALES_PERSONS
```


TAN

TAN (*arithmetic_expression*)

Description

The **TAN** function returns the tangent of the *arithmetic_expression*. The *arithmetic_expression* represents the angle expressed in radians.

Example

This example will return the tangent of the angle expressed in radians:

```
SELECT TAN(radians)
FROM TABLES_OF_ANGLES;
```

TRANSLATE

TRANSLATE (*arithmetic_expression, arithmetic_expression, arithmetic_expression*)

Description

The *TRANSLATE* function returns the first *arithmetic_expression* with all occurrences of each of the characters passed in the second *arithmetic_expression* replaced by their corresponding characters passed in the third *arithmetic_expression*.

This replacement is carried out on a character by character basis (e.g. 1st character of the second *arithmetic_expression* is replaced each time it occurs in the first *arithmetic_expression* by the 1st character of the third *arithmetic_expression*, and so on).

If there are fewer characters in the third *arithmetic_expression* than in the second one, any occurrences of characters found in the second *arithmetic_expression* that do not have a corresponding character in the third *arithmetic_expression* will be removed from the first *arithmetic_expression* (e.g. if the second *arithmetic_expression* has five characters to be searched for and the third *arithmetic_expression* only contains four replacement characters, each time the fifth character of the second *arithmetic_expression* is found in the first *arithmetic_expression*, it will be removed from the value returned).

Example

This example replaces all occurrences of "a" with "1" and all occurrences of "b" with "2":

```
TRANSLATE ('abcd', 'ab', '12')  
` returns '12cd'
```

TRIM

TRIM ([[**LEADING** |**TRAILING** |**BOTH**] [*arithmetic_expression*] **FROM**] *arithmetic_expression*)

Description

The *TRIM* function removes empty spaces, or specified characters when they are passed in first optional *arithmetic_expression*, from the extremities of the *arithmetic_expression* passed.

You can pass **LEADING** to indicate that the spaces/characters should be removed from the beginning of the *arithmetic_expression*, **TRAILING** to indicate that they should be removed from the end of it, or **BOTH**. If none of these keywords are passed, it is the equivalent of passing **BOTH** (the spaces or characters will be removed from both the beginning and end of the *arithmetic_expression*).

The optional first *arithmetic_expression* passed indicates the specific individual characters to be removed from the second *arithmetic_expression*. If it is omitted, only empty spaces will be removed.

Example

This example removes any empty spaces from the product names:

```
SELECT TRIM(Name)
FROM PRODUCTS;
```

TRUNC

TRUNC (*arithmetic_expression* [, *arithmetic_expression*])

Description

The **TRUNC** function returns the first *arithmetic_expression* truncated to "x" places to the right of the decimal point, where "x" is the second optional *arithmetic_expression*. If this second *arithmetic_expression* is not passed, the *arithmetic_expression* is simply truncated by removing any decimal places.

Example

This function truncates the number passed to 1 place after the decimal point:

```
TRUNC(2.42 , 1)  
`returns 2.40
```

TRUNCATE

TRUNCATE (*arithmetic_expression*[, *arithmetic_expression*])

Description

The *TRUNCATE* function returns the first *arithmetic_expression* truncated to "x" places to the right of the decimal point, where "x" is the second optional *arithmetic_expression*. If this second *arithmetic_expression* is not passed, the *arithmetic_expression* is simply truncated by removing any decimal places.

Beispiel

See the example for the **TRUNC** function.

UPPER

UPPER (*arithmetic_expression*)

Description

The *UPPER* function returns the *arithmetic_expression* passed as a string where all the characters are in uppercase.

Example

This example will return the names of products in uppercase:

```
SELECT UPPER (Name)
FROM PRODUCTS;
```

WEEK

WEEK (*arithmetic_expression*)

Description

The *WEEK* function returns the week of the year (ranging from 1 to 54) of the date passed in the *arithmetic_expression*. The week begins on Sunday and January 1st is always in the first week.

Example

This example returns a number representing the week of the year for each date of birth passed:

```
SELECT WEEK(Date_of_birth);
```

YEAR

YEAR (*arithmetic_expression*)

Description

The *YEAR* function returns the year part of the date passed in the *arithmetic_expression*.

Example

Supposing that we have the PEOPLE table with a Date_of_Birth field. To find out the year of the date of birth for every person in PEOPLE :

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


☰ **Appendix**

➤ Appendix A: SQL Error Codes

Appendix A: SQL Error Codes

The SQL engine returns specific errors which are listed below. These errors can be intercepted using an error-handling method installed by the **ON ERR CALL** command.

Generic errors

- 1001 INVALID ARGUMENT
- 1002 INVALID INTERNAL STATE
- 1003 SQL SERVER IS NOT RUNNING
- 1004 Access denied
- 1005 FAILED TO LOCK SYNCHRONIZATION PRIMITIVE
- 1006 FAILED TO UNLOCK SYNCHRONIZATION PRIMITIVE
- 1007 SQL SERVER IS NOT AVAILABLE
- 1008 COMPONENT BRIDGE IS NOT AVAILABLE
- 1009 REMOTE SQL SERVER IS NOT AVAILABLE
- 1010 EXECUTION INTERRUPTED BY USER

Semantic errors

1101 Table '{key1}' does not exist in the database.
1102 Column '{key1}' does not exist.
1103 Table '{key1}' is not declared in the FROM clause.
1104 Column name reference '{key1}' is ambiguous.
1105 Table alias '{key1}' is the same as table name.
1106 Duplicate table alias - '{key1}'.
1107 Duplicate table in the FROM clause - '{key1}'.
1108 Operation {key1} {key2} {key3} is not type safe.
1109 Invalid ORDER BY index - {key1}.
1110 Function {key1} expects one parameter, not {key2}.
1111 Parameter {key1} of type {key2} in function call {key3} is not implicitly convertible to {key4}.
1112 Unknown function - {key1}.
1113 Division by zero.
1114 Sorting by indexed item in the SELECT list is not allowed - ORDER BY item {key1}.
1115 DISTINCT NOT ALLOWED
1116 Nested aggregate functions are not allowed in the aggregate function {key1}.
1117 Column function is not allowed.
1118 Cannot mix column and scalar operations.
1119 Invalid GROUP BY index - {key1}.
1120 GROUP BY index is not allowed.
1121 GROUP BY is not allowed with 'SELECT * FROM ...'.
1122 HAVING is not an aggregate expression.
1123 Column '{key1}' is not a grouping column and cannot be used in the ORDER BY clause.
1124 Cannot mix {key1} and {key2} types in the IN predicate.
1125 Escape sequence '{key1}' in the LIKE predicate is too long. It must be exactly one character.
1126 Bad escape character - '{key1}'.
1127 Unknown escape sequence - '{key1}'.
1128 Column references from more than one query in aggregate function {key1} are not allowed.
1129 Scalar item in the SELECT list is not allowed when GROUP BY clause is present.
1130 Sub-query produces more than one column.
1131 Subquery must return one row at the most but instead it returns {key1}.
1132 INSERT value count {key1} does not match column count {key2}.
1133 Duplicate column reference in the INSERT list - '{key1}'.
1134 Column '{key1}' does not allow NULL values.
1135 Duplicate column reference in the UPDATE list - '{key1}'.
1136 Table '{key1}' already exists.
1137 Duplicate column '{key1}' in the CREATE TABLE command.
1138 DUPLICATE COLUMN IN COLUMN LIST
1139 More than one primary key is not allowed.
1140 Ambiguous foreign key name - '{key1}'.
1141 Column count {key1} in the child table does not match column count {key2} in the parent table of the foreign key definition.
1142 Column type mismatch in the foreign key definition. Cannot relate {key1} in child table to {key2} in parent table.
1143 Failed to find matching column in parent table for '{key1}' column in child table.
1144 UPDATE and DELETE constraints must be the same.
1145 FOREIGN KEY DOES NOT EXIST
1146 Invalid LIMIT value in SELECT command - {key1}.
1147 Invalid OFFSET value in SELECT command - {key1}.
1148 Primary key does not exist in table '{key1}'.

1149 FAILED TO CREATE FOREIGN KEY
1150 Column '{key1}' is not part of a primary key.
1151 FIELD IS NOT UPDATEABLE
1152 FOUND VIEW COLUMN
1153 Bad data type length '{key1}'.
1154 EXPECTED EXECUTE IMMEDIATE COMMAND
1155 INDEX ALREADY EXISTS
1156 Auto-increment option is not allowed for column '{key1}' of type {key2}.
1157 SCHEMA ALREADY EXISTS
1158 SCHEMA DOES NOT EXIST
1159 Cannot drop system schema
1160 CHARACTER ENCODING NOT ALLOWED

Implementation errors

1203 The functionality is not implemented.
1204 Failed to create record {key1}.
1205 Failed to update field '{key1}'.
1206 Failed to delete record '{key1}'.
1207 NO MORE JOIN SEEDS POSSIBLE
1208 FAILED TO CREATE TABLE
1209 FAILED TO DROP TABLE
1210 CANT BUILD BTREE FOR ZERO RECORDS
1211 COMMAND COUNT GREATER THAN ALLOWED
1212 FAILED TO CREATE DATABASE
1213 FAILED TO DROP COLUMN
1214 VALUE IS OUT OF BOUNDS
1215 FAILED TO STOP SQL_SERVER
1216 FAILED TO LOCALIZE
1217 Failed to lock table for reading.
1218 FAILED TO LOCK TABLE FOR WRITING
1219 TABLE STRUCTURE STAMP CHANGED
1220 FAILED TO LOAD RECORD
1221 FAILED TO LOCK RECORD FOR WRITING
1222 FAILED TO PUT SQL LOCK ON A TABLE
1223 FAILED TO RETAIN COOPERATIVE TASK
1224 FAILED TO LOAD INFILE

Parsing error

1301 PARSING FAILED

Runtime language access errors

1401 COMMAND NOT SPECIFIED
1402 ALREADY LOGGED IN
1403 SESSION DOES NOT EXIST
1404 UNKNOWN BIND ENTITY
1405 INCOMPATIBLE BIND ENTITIES
1406 REQUEST RESULT NOT AVAILABLE
1407 BINDING LOAD FAILED
1408 COULD NOT RECOVER FROM PREVIOUS ERRORS
1409 NO OPEN STATEMENT
1410 RESULT EOF
1411 BOUND VALUE IS NULL
1412 STATEMENT ALREADY OPENED
1413 FAILED TO GET PARAMETER VALUE
1414 INCOMPATIBLE PARAMETER ENTITIES
1415 Query parameter is either not allowed or was not provided.
1416 COLUMN REFERENCE PARAMETERS FROM DIFFERENT TABLES
1417 EMPTY STATEMENT
1418 FAILED TO UPDATE VARIABLE
1419 FAILED TO GET TABLE REFERENCE
1420 FAILED TO GET TABLE CONTEXT
1421 COLUMNS NOT ALLOWED
1422 INVALID COMMAND COUNT
1423 INTO CLAUSE NOT ALLOWED
1424 EXECUTE IMMEDIATE NOT ALLOWED
1425 ARRAY NOT ALLOWED IN EXECUTE IMMEDIATE
1426 COLUMN NOT ALLOWED IN EXECUTE IMMEDIATE
1427 NESTED BEGIN END SQL NOT ALLOWED
1428 RESULT IS NOT A SELECTION
1429 INTO ITEM IS NOT A VARIABLE
1430 VARIABLE WAS NOT FOUND
1431 PTR OF PTR NOT ALLOWED
1432 POINTER OF UNKNOWN TYPE

Date parsing errors

1501 SEPARATOR_EXPECTED
1502 FAILED TO PARSE DAY OF MONTH
1503 FAILED TO PARSE MONTH
1504 FAILED TO PARSE YEAR
1505 FAILED TO PARSE HOUR
1506 FAILED TO PARSE MINUTE
1507 FAILED TO PARSE SECOND
1508 FAILED TO PARSE MILLISECOND
1509 INVALID AM PM USAGE
1510 FAILED TO PARSE TIME ZONE
1511 UNEXPECTED CHARACTER
1512 Failed to parse timestamp.
1513 Failed to parse duration.
1551 FAILED TO PARSE DATE FORMAT

Lexer errors

1601 NULL INPUT STRING
1602 NON TERMINATED STRING
1603 NON TERMINATED COMMENT
1604 INVALID NUMBER
1605 UNKNOWN START OF TOKEN
1606 NON TERMINATED NAME/* closing ']' is missing
1607 NO VALID TOKENS

Validation errors - Status errors following direct errors

1701 Failed to validate table '{key1}'.
1702 Failed to validate FROM clause.
1703 Failed to validate GROUP BY clause.
1704 Failed to validate SELECT list.
1705 Failed to validate WHERE clause.
1706 Failed to validate ORDER BY clause.
1707 Failed to validate HAVING clause.
1708 Failed to validate COMPARISON predicate.
1709 Failed to validate BETWEEN predicate.
1710 Failed to validate IN predicate.
1711 Failed to validate LIKE predicate.
1712 Failed to validate ALL ANY predicate.
1713 Failed to validate EXISTS predicate.
1714 Failed to validate IS NULL predicate.
1715 Failed to validate subquery.
1716 Failed to validate SELECT item {key1}.
1717 Failed to validate column '{key1}'.
1718 Failed to validate function '{key1}'.
1719 Failed to validate CASE expression.
1720 Failed to validate command parameter.
1721 Failed to validate function parameter {key1}.
1722 Failed to validate INSERT item {key1}.
1723 Failed to validate UPDATE item {key1}.
1724 Failed to validate column list.
1725 Failed to validate foreign key.
1726 Failed to validate SELECT command.
1727 Failed to validate INSERT command.
1728 Failed to validate DELETE command.
1729 Failed to validate UPDATE command.
1730 Failed to validate CREATE TABLE command.
1731 Failed to validate DROP TABLE command.
1732 Failed to validate ALTER TABLE command.
1733 Failed to validate CREATE INDEX command.
1734 Failed to validate LOCK TABLE command.
1735 Failed to calculate LIKE predicate pattern.

Execution errors - Status errors following direct errors

1801 Failed to execute SELECT command.
1802 Failed to execute INSERT command.
1803 Failed to execute DELETE command.
1804 Failed to execute UPDATE command.
1805 Failed to execute CREATE TABLE command.
1806 Failed to execute DROP TABLE command.
1807 Failed to execute CREATE DATABASE command.
1808 Failed to execute ALTER TABLE command.
1809 Failed to execute CREATE INDEX command.
1810 Failed to execute DROP INDEX command.
1811 Failed to execute LOCK TABLE command.
1812 Failed to execute TRANSACTION command.
1813 Failed to execute WHERE clause.
1814 Failed to execute GROUP BY clause.
1815 Failed to execute HAVING clause.
1816 Failed to aggregate.
1817 Failed to execute DISTINCT.
1818 Failed to execute ORDER BY clause.
1819 Failed to build DB4D query.
1820 Failed to calculate comparison predicate.
1821 Failed to execute subquery.
1822 Failed to calculate BETWEEN predicate.
1823 Failed to calculate IN predicate.
1824 Failed to calculate ALL/ANY predicate.
1825 Failed to calculate LIKE predicate.
1826 Failed to calculate EXISTS predicate.
1827 Failed to calculate NULL predicate.
1828 Failed to perform arithmetic operation.
1829 Failed to calculate function call '{key1}'.
1830 Failed to calculate case expression.
1831 Failed to calculate function parameter '{key1}'.
1832 Failed to calculate 4D function call.
1833 Failed to sort while executing ORDER BY clause.
1834 Failed to calculate record hash.
1835 Failed to compare records.
1836 Failed to calculate INSERT value {key1}.
1837 DB4D QUERY FAILED
1838 FAILED TO EXECUTE ALTER SCHEMA COMMAND
1839 FAILED TO EXECUTE GRANT COMMAND

Cacheable errors

2000 CACHEABLE NOT INITIALIZED
2001 VALUE ALREADY CACHED
2002 CACHED VALUE NOT FOUND
2003 CACHE IS FULL
2004 CACHING IS NOT POSSIBLE

Protocol errors

3000 HEADER NOT FOUND
3001 UNKNOWN COMMAND
3002 ALREADY LOGGED IN
3003 NOT LOGGED IN
3004 UNKNOWN OUTPUT MODE
3005 INVALID STATEMENT ID
3006 UNKNOWN DATA TYPE
3007 STILL LOGGED IN
3008 SOCKET READ ERROR
3009 SOCKET WRITE ERROR
3010 BASE64 DECODING ERROR
3011 SESSION TIMEOUT
3012 FETCH TIMESTAMP ALREADY EXISTS
3013 BASE64 ENCODING ERROR
3014 INVALID HEADER TERMINATOR
3015 INVALID SESSION TICKET
3016 HEADER TOO LONG
3017 INVALID AGENT SIGNATURE
3018 UNEXPECTED HEADER VALUE