4D

SQL Reference
Windows® / Mac OS®

4D SQL Reference
Version 11 for Windows® and Mac OS®

Contents

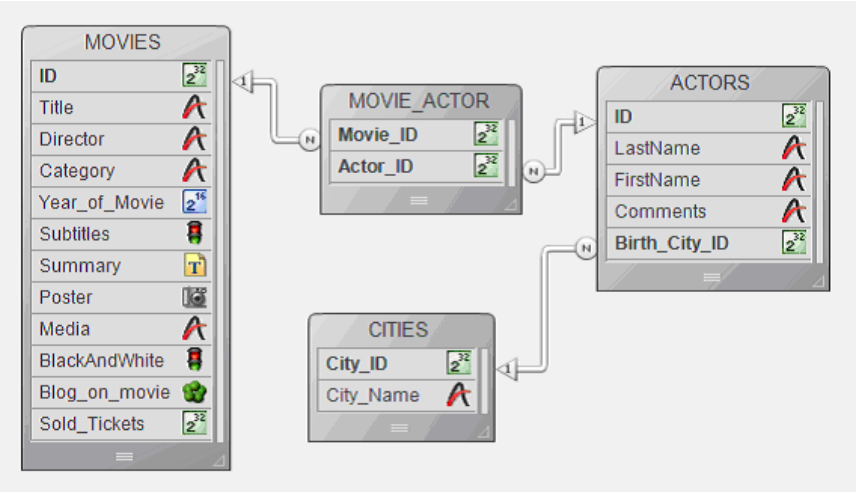
4. Syntax rules....101

1

Tutorial

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.

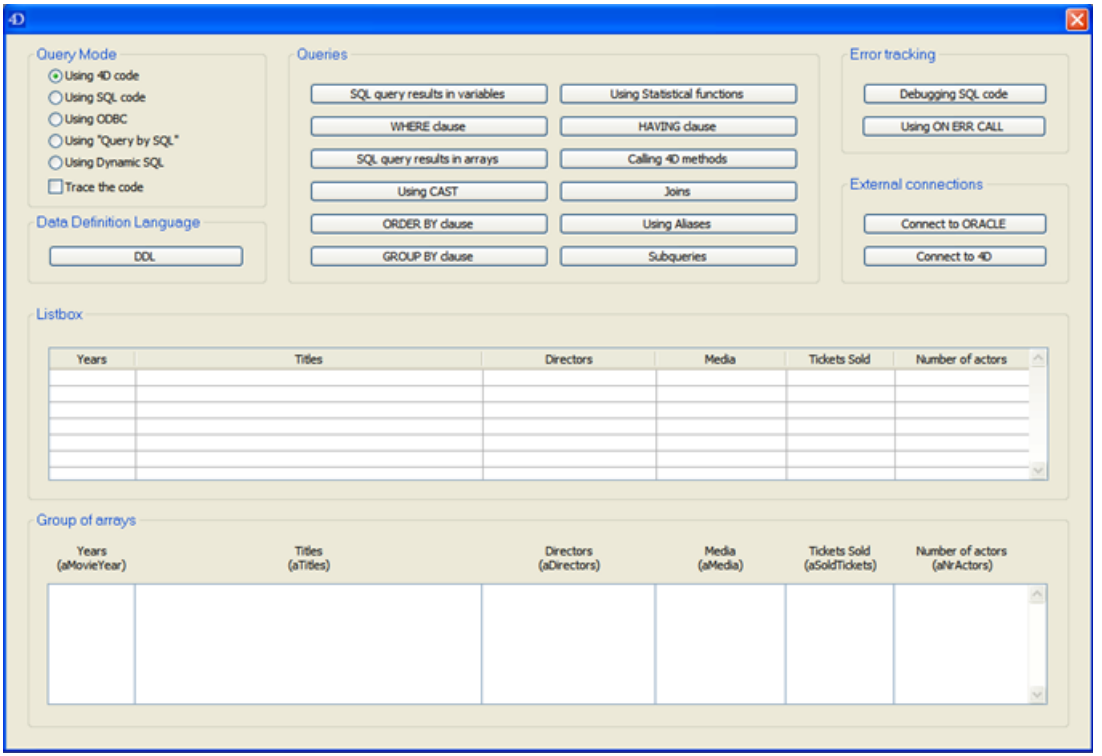
Description of the database that accompanies this tutorial
All the examples that will be detailed in this document were fully tested and verified in one of the example databases named "4D SQL Code Samples".
The structure is as follows:



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:



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:

```
` Using standard 4D Code
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:

```
` Using 4D SQL and the "<<>>" notation for the receiving parameter
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:

```
` Using 4D SQL and the ":" notation for the receiving parameter
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 "]":

```
  ` Using 4D SQL and the "<<>>" notation for  receiving an interprocess parameter
C_LONGINT(<>AllMovies)
<>AllMovies:=0
Begin SQL
    SELECT COUNT(*)
      FROM MOVIES
      INTO <<[<>AllMovies]>>
End SQL
ALERT("The Video Library contains "+String(<>AllMovies)+" movies")
```

The second way to interact with the SQL engine is using ODBC commands. Thus the simple query above becomes:

```
  ` Using ODBC commands
C_LONGINT($AllMovies)
$AllMovies:=0
  ` Initialize a connection with the internal SQL engine
ODBC LOGIN(SQL_INTERNAL;"";"")
  ` Execute the query and return the result in the $AllMovies variable
ODBC EXECUTE("SELECT COUNT(*) FROM MOVIES";$AllMovies)
  ` Retrieve all the records found
ODBC LOAD RECORD(ODBC All Records )
  ` Close the connection
ODBC LOGOUT
ALERT("The Video Library contains "+String($AllMovies)+" movies")
```

For more information concerning ODBC commands, please refer to the *4D Language Reference* manual.

The third way to interact with the new SQL engine is using the 4D QUERY BY SQL command. In this situation, the simple query above becomes:

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

In fact, the QUERY BY SQL command can be used to execute a simple SELECT query that can be written as follows:

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

myTable is the name of the table passed in the first parameter and SQL_Formula is the query string passed as the second parameter:

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

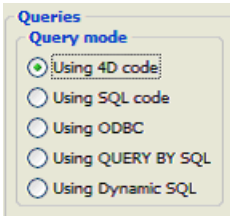
In our case there is no WHERE clause, so we forced one: "ID <> 0". The equivalent in SQL for the whole query would be:

```
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:

```
  ` Using dynamic SQL by EXECUTE IMMEDIATE
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")
```

To test all the above examples, launch the Video Library database and go to the main dialog box. On the left side of the dialog, you can choose the query mode: using standard 4D code, SQL code, ODBC commands, the QUERY BY SQL command or dynamic SQL:



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

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:

```
` Using standard 4D Code
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:

```
` Using 4D SQL
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 ODBC commands, the above query becomes:

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

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

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

```
  ` Using QUERY BY SQL
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:

```
  ` Using dynamic SQL by EXECUTE IMMEDIATE
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 Video Library database and go to the main window. You can then choose the query mode and press the "WHERE clause" button.

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:

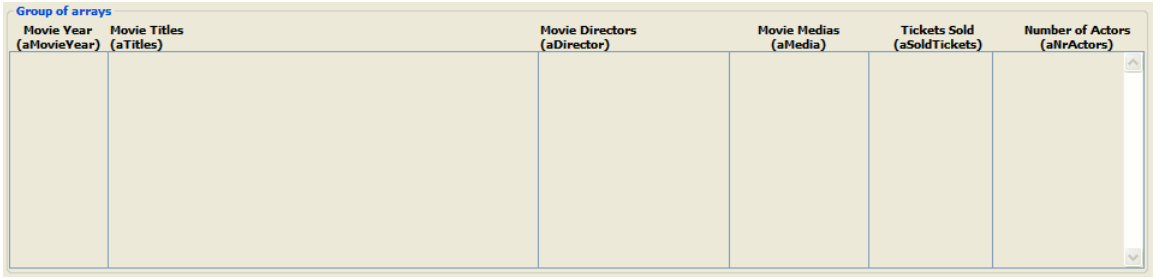
```
` Using standard 4D code
ARRAY LONGINT(aNrActors;0)
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)

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)
```

Using SQL code, the above query becomes:

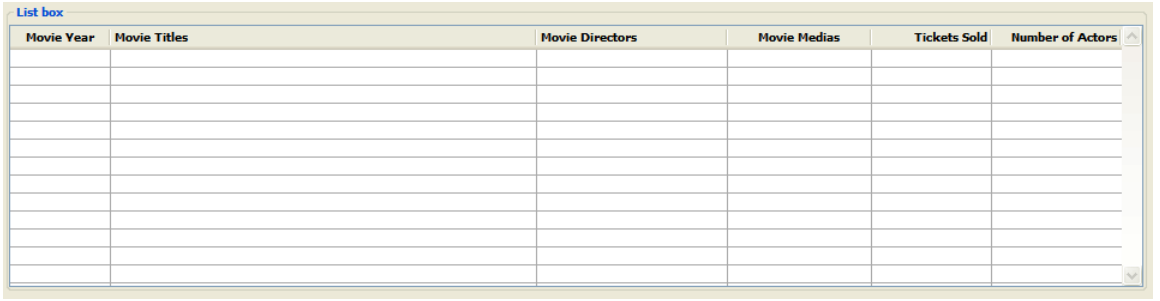
```
` Using 4D SQL
ARRAY LONGINT(aNrActors;0)
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_LONGINT($MovieYear)

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

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 ODBC commands, the above query becomes:

```
` Using ODBC commands
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
ODBC 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 >= :$MovieYear"
ODBC EXECUTE($tQueryTxt;aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets)
ODBC LOAD RECORD(ODBC All Records )
ODBC LOGOUT
```

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

```
` Using QUERY BY SQL
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)
```

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

```
` Using dynamic SQL by EXECUTE IMMEDIATE
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
$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
```

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

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 d not match the types supported by the SQL standard.
You will find below the above query slightly modified in order to use the CAST expression. The initial query in 4D code would be:

```
` Using standard 4D code
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)
` Intialize 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:

```
` Using 4D SQL
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
` Intialize the rest of the list box columns in order to display the information
ARRAY LONGINT(aNrActors;Size of array(aMovieYear))
```

Using ODBC commands, the above query becomes:

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

REDUCE SELECTION([MOVIES];0)
ODBC LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= CAST('1960' AS INT)"
ODBC EXECUTE($tQueryTxt;aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets)
ODBC LOAD RECORD(ODBC All Records)
ODBC LOGOUT
` Intialize 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:

```
` Using QUERY BY SQL
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)

REDUCE SELECTION([MOVIES];0)
QUERY BY SQL([MOVIES];"Year_of_Movie >= CAST('1960' AS INT)")
SELECTION TO ARRAY([MOVIES]Year_of_Movie;aMovieYear;[MOVIES]Title;aTitles;[MOVIES]Director;aDirectors;[MOVIES]Media;aMedias;[MOVIES]Sold_Tickets;aSoldTickets)
` Intialize 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:

```
  ` Using dynamic SQL by EXECUTE IMMEDIATE
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
ARRAY TEXT(aMedias;0)
C_TEXT($tQueryTxt)

REDUCE SELECTION([MOVIES];0)
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, Title, Director, Media, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= CAST('1960' AS INT)"
$tQueryTxt:=$tQueryTxt+" INTO :aMovieYear, :aTitles, :aDirectors, :aMedias, :aSoldTickets;"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
  ` Intialize 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 Video Library database and go to the main window. You can then choose the query mode and press the "Using CAST" button.

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:

```
` Using standard 4D Code
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:

```
` Using 4D SQL
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 ODBC commands, the above query becomes:

```
` Using ODBC commands
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
ODBC 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"
ODBC EXECUTE($tQueryTxt;aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets)
ODBC LOAD RECORD(ODBC All Records)
ODBC LOGOUT
```

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

```
` Using QUERY BY SQL
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:

```
  ` Using dynamic SQL by EXECUTE IMMEDIATE
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 Video Library database and go to the main window. You can then choose the query mode and press the "ORDER BY clause" button.

We would like to get some information about the number of tickets sold each year starting with 1960. 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 1960, 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 ELEMENT(aMovieYear;$vInd;1)
      aMovieYear{$vInd}:=$vCrtMovieYear
      INSERT ELEMENT(aSoldTickets;$vInd;1)
   End if
   aSoldTickets{$vInd}:=aSoldTickets{$vInd}+[MOVIES]Sold_Tickets
   NEXT RECORD([MOVIES])
End for
  ` Intialize the rest of the list box columns in order to visualise 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)
$MovieYear:=1979
Begin SQL
    SELECT Year_of_Movie, SUM(Sold_Tickets)
      FROM MOVIES
      WHERE Year_of_Movie >= :$MovieYear
      GROUP BY Year_of_Movie
      ORDER BY 1
      INTO :aMovieYear,  :aSoldTickets;
End SQL
  ` Intialize the rest of the list box columns in order to visualise 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 ODBC commands, the above query becomes:

```
  ` Using ODBC commands
C_TEXT($tQueryTxt)
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
C_LONGINT($MovieYear)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1979
ODBC LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" GROUP BY Year_of_Movie"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
ODBC EXECUTE($tQueryTxt;aMovieYear;aSoldTickets)
ODBC LOAD RECORD(ODBC All Records)
ODBC LOGOUT
  ` Intialize the rest of the list box columns in order to visualise 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 QUERY BY SQL command, the above query becomes:

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

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1979
QUERY BY SQL([MOVIES];"Year_of_Movie >= :$MovieYear")
ORDER BY([MOVIES];[MOVIES]Year_of_Movie;>)
$vCrtMovieYear:=0
$vInd:=Size of array(aMovieYear)
For ($i;1;Records in selection([MOVIES]))
   If ([MOVIES]Year_of_Movie#$vCrtMovieYear)
      $vCrtMovieYear:=[MOVIES]Year_of_Movie
      $vInd:=$vInd+1
      INSERT ELEMENT(aMovieYear;$vInd;1)
      aMovieYear{$vInd}:=$vCrtMovieYear
      INSERT ELEMENT(aSoldTickets;$vInd;1)
   End if
   aSoldTickets{$vInd}:=aSoldTickets{$vInd}+[MOVIES]Sold_Tickets
   NEXT RECORD([MOVIES])
End for
` Intialize the rest of the list box columns in order to visualise 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
   ` Intialize the rest of the list box columns in order to visualise 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 Video Library database and go to the main window. You can then choose the query mode and press the "GROUP BY clause" button.

Sometimes it can be useful to get statistical information about certain values. SQL includes many aggregate functions like MIN, MAX, AVERAGE, SUM and so on. Using aggregate functions, we would like to get information about the number of tickets sold each year starting with 1960. The result will be sorted by year.
To do this, we must total all the tickets sold for each movie in each year more recent than 1960, and then sort the result by year.
The initial query in 4D code would be:

```
` Using standard 4D code
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($vSum)+Char(13)
ALERT($AlertTxt)
```

Using SQL code, the above query becomes:

```
  ` Using 4D SQL
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 ODBC commands, the above query becomes:

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

$vMin:=0
$vMax:=0
$vAverage:=0
$vSum:=0
ODBC LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT MIN(Sold_Tickets), MAX(Sold_Tickets), AVG(Sold_Tickets), SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
ODBC EXECUTE($tQueryTxt;$vMin;$vMax;$vAverage;$vSum)
ODBC LOAD RECORD(ODBC All Records)
ODBC 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:

```
   ` Using dynamic SQL by EXECUTE IMMEDIATE
  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 Video Library database and go to the main window. You can then choose the query mode and press the "Using Aggregate functions" button.

We would now like to get the total amount of tickets sold per year starting with 1960, 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 1960, 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 ELEMENT(aMovieYear;$vInd;1)
         aMovieYear{$vInd}:=$vCrtMovieYear
         INSERT ELEMENT(aSoldTickets;$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}>=$MinSoldTickets)
   DELETE ELEMENT(aSoldTickets;$vInd;1)
   DELETE ELEMENT(aMovieYear;$vInd;1)
End if
   ` Intialize the rest of the list box columns in order to visualise 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:

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

$MovieYear:=1979
$MinSoldTickets:=10000000
Begin SQL
    SELECT Year_of_Movie, SUM(Sold_Tickets)
      FROM MOVIES
      WHERE Year_of_Movie >= :$MovieYear
      GROUP BY Year_of_Movie
      HAVING SUM(Sold_Tickets) < :$MinSoldTickets
      ORDER BY 1
      INTO :aMovieYear,  :aSoldTickets;
End SQL
  ` Intialize the rest of the list box columns in order to visualise 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 ODBC commands, the above query becomes:

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

$MovieYear:=1979
$MinSoldTickets:=10000000
ODBC LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Year_of_Movie, SUM(Sold_Tickets)"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Year_of_Movie >= :$MovieYear"
$tQueryTxt:=$tQueryTxt+" GROUP BY Year_of_Movie"
$tQueryTxt:=$tQueryTxt+" HAVING SUM(Sold_Tickets) < :$MinSoldTickets"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
ODBC EXECUTE($tQueryTxt;aMovieYear;aSoldTickets)
ODBC LOAD RECORD(ODBC All Records )
ODBC LOGOUT
  ` Intialize the rest of the list box columns in order to visualise 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 QUERY BY SQL command, the above query becomes:

```
` Using QUERY BY SQL
C_TEXT($tQueryTxt)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aSoldTickets;0)
C_LONGINT($MovieYear;$MinSoldTickets;$vCrtMovieYear;$vInd;$i)

REDUCE SELECTION([MOVIES];0)
$MovieYear:=1979
$MinSoldTickets:=10000000
QUERY BY SQL([MOVIES];"Year_of_Movie >= :$MovieYear")
ORDER BY([MOVIES];[MOVIES]Year_of_Movie;>)
$vCrtMovieYear:=0
$vInd:=Size of array(aMovieYear)
For ($i;1;Records in selection([MOVIES]))
   If ([MOVIES]Year_of_Movie#$vCrtMovieYear)
      $vCrtMovieYear:=[MOVIES]Year_of_Movie
      If (aSoldTickets{$vInd}<$MinSoldTickets)
         $vInd:=$vInd+1
         INSERT ELEMENT(aMovieYear;$vInd;1)
         aMovieYear{$vInd}:=$vCrtMovieYear
         INSERT ELEMENT(aSoldTickets;$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}>=$MinSoldTickets)
   DELETE ELEMENT(aSoldTickets;$vInd;1)
   DELETE ELEMENT(aMovieYear;$vInd;1)
End if
` Intialize the rest of the list box columns in order to visualise 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($tQueryTxt)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aSoldTickets;0)
C_LONGINT($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
   ` Intialize the rest of the list box columns in order to visualise 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 Video Library database and go to the main window. You can then choose the query mode and press the "HAVING clause" button.

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 ELEMENT(aMovieYear;$vInd;1)
      aMovieYear{$vInd}:=[MOVIES]Year_of_Movie
      INSERT ELEMENT(aTitles;$vInd;1)
      aTitles{$vInd}:=[MOVIES]Title
      INSERT ELEMENT(aDirectors;$vInd;1)
      aDirectors{$vInd}:=[MOVIES]Director
      INSERT ELEMENT(aMedias;$vInd;1)
      aMedias{$vInd}:=[MOVIES]Media
      INSERT ELEMENT(aSoldTickets;$vInd;1)
      aSoldTickets{$vInd}:=[MOVIES]Sold_Tickets
      INSERT ELEMENT(aNrActors;$vInd;1)
      aNrActors{$vInd}:=$vCrtActors
   End if
   NEXT RECORD([MOVIES])
End for
SORT ARRAY(aMovieYear;aTitles;aDirectors;aMedias;aSoldTickets;aNrActors;>)
```

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

You can see that we are able to call a 4D function inside SQL code using the syntax: {fn <4D function name> AS <4D function result type>}.

Using ODBC commands, the above query becomes:

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

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

```
` Using QUERY BY SQL
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 ELEMENT(aMovieYear;$vInd;1)
   aMovieYear{$vInd}:=[MOVIES]Year_of_Movie
   INSERT ELEMENT(aTitles;$vInd;1)
   aTitles{$vInd}:=[MOVIES]Title
   INSERT ELEMENT(aDirectors;$vInd;1)
   aDirectors{$vInd}:=[MOVIES]Director
   INSERT ELEMENT(aMedias;$vInd;1)
   aMedias{$vInd}:=[MOVIES]Media
   INSERT ELEMENT(aSoldTickets;$vInd;1)
   aSoldTickets{$vInd}:=[MOVIES]Sold_Tickets
   INSERT ELEMENT(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:

```
` Using dynamic SQL by EXECUTE IMMEDIATE
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 Video Library database and go to the main window. You can then choose the query mode and press the "Calling 4D methods" button.

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:

```
   ` Using standard 4D code
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 ELEMENT(aTitles;$vInd;1)
   aTitles{$vInd}:=[ACTORS]FirstName+" "+[ACTORS]LastName
   RELATE ONE([ACTORS]Birth_City_ID)
   INSERT ELEMENT(aDirectors;$vInd;1)
   aDirectors{$vInd}:=[CITIES]City_Name
   NEXT RECORD([ACTORS])
End for
   ` Intialize the rest of the list box columns in order to visualise 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:

```
  ` Using 4D SQL code
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
  ` Intialize the rest of the list box columns in order to visualise 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 ODBC commands, the above query becomes:

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

ODBC LOGIN(SQL_INTERNAL ;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName), CITIES.City_Name"
$tQueryTxt:=$tQueryTxt+" FROM ACTORS, CITIES"
$tQueryTxt:=$tQueryTxt+" WHERE ACTORS.Birth_City_ID=CITIES.City_ID"
$tQueryTxt:=$tQueryTxt+" ORDER BY 2,1"
ODBC EXECUTE($tQueryTxt;aTitles;aDirectors)
ODBC LOAD RECORD(ODBC All Records )
ODBC LOGOUT
  ` Intialize the rest of the list box columns in order to visualise 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 the QUERY BY SQL command, we are unable to carry out the query above because it is not possible to pass more than one table as the first parameter.

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

```
  ` Using dynamic SQL by EXECUTE IMMEDIATE
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
C_TEXT($tQueryTxt)

$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName), CITIES.City_Name"
$tQueryTxt:=$tQueryTxt+" FROM ACTORS, CITIES"
$tQueryTxt:=$tQueryTxt+" WHERE ACTORS.Birth_City_ID=CITIES.City_ID"
$tQueryTxt:=$tQueryTxt+" ORDER BY 2,1"
$tQueryTxt:=$tQueryTxt+" INTO :aTitles, :aDirectors"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
  ` Intialize the rest of the list box columns in order to visualise 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))
```

To test all the above examples, launch the Video Library database and go to the main window. You can then choose the query mode and press the "Joins" button.

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:

```
  ` Using standard 4D code
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 ELEMENT(aTitles;$vInd;1)
   aTitles{$vInd}:=[ACTORS]FirstName+" "+[ACTORS]LastName
   RELATE ONE([ACTORS]Birth_City_ID)
   INSERT ELEMENT(aDirectors;$vInd;1)
   aDirectors{$vInd}:=[CITIES]City_Name
   NEXT RECORD([ACTORS])
End for
  ` Intialize the rest of the list box columns in order to visualise 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:

```
  ` Using 4D SQL code
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
  ` Intialize the rest of the list box columns in order to visualise 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 ODBC commands, the above query becomes:

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

ODBC LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT CONCAT(CONCAT(ACTORS.FirstName,' '),ACTORS.LastName), CITIES.City_Name"
$tQueryTxt:=$tQueryTxt+" FROM ACTORS AS 'Act', CITIES AS 'Cit'"
$tQueryTxt:=$tQueryTxt+" WHERE Act.Birth_City_ID=Cit.City_ID"
$tQueryTxt:=$tQueryTxt+" ORDER BY 2,1"
ODBC EXECUTE($tQueryTxt;aTitles;aDirectors)
ODBC LOAD RECORD(ODBC All Records )
ODBC LOGOUT
  ` Intialize the rest of the list box columns in order to visualise 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 the QUERY BY SQL command, we are unable to carry out the query above because it is not possible to pass more than one table as the first parameter.

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

```
  ` Using dynamic SQL by EXECUTE IMMEDIATE
ARRAY TEXT(aTitles;0)
ARRAY TEXT(aDirectors;0)
C_TEXT($tQueryTxt)

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

To test all the above examples, launch the Video Library database and go to the main window. You can then choose the query mode and press the "Using Aliases" button.

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:

```
  ` Using standard 4D code
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 ELEMENT(aTitles;$vInd;1)
      aTitles{$vInd}:=[MOVIES]Title
      INSERT ELEMENT(aSoldTickets;$vInd;1)
      aSoldTickets{$vInd}:=[MOVIES]Sold_Tickets
   End if
   NEXT RECORD([MOVIES])
End for
  ` Intialize the rest of the list box columns in order to visualise 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:

```
  ` Using 4D SQL code
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
  ` Intialize the rest of the list box columns in order to visualise 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 ODBC commands, the above query becomes:

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

ODBC LOGIN(SQL_INTERNAL;"";"")
$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Title, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Sold_Tickets > (SELECT AVG(Sold_Tickets) FROM MOVIES)"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
ODBC EXECUTE($tQueryTxt;aTitles;aSoldTickets)
ODBC LOAD RECORD(ODBC All Records)
ODBC LOGOUT
  ` Intialize the rest of the list box columns in order to visualise 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 the QUERY BY SQL command, the above query becomes:

```
  ` Using QUERY BY SQL
ARRAY LONGINT(aSoldTickets;0)
ARRAY TEXT(aTitles;0)

QUERY BY SQL([MOVIES];"Sold_Tickets > (SELECT AVG(Sold_Tickets) FROM MOVIES)")
ORDER BY([MOVIES];[MOVIES]Title;>)
SELECTION TO ARRAY([MOVIES]Title;aTitles;[MOVIES]Sold_Tickets;aSoldTickets)
  ` Intialize the rest of the list box columns in order to visualise 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 the SQL EXECUTE IMMEDIATE command, the query above becomes:

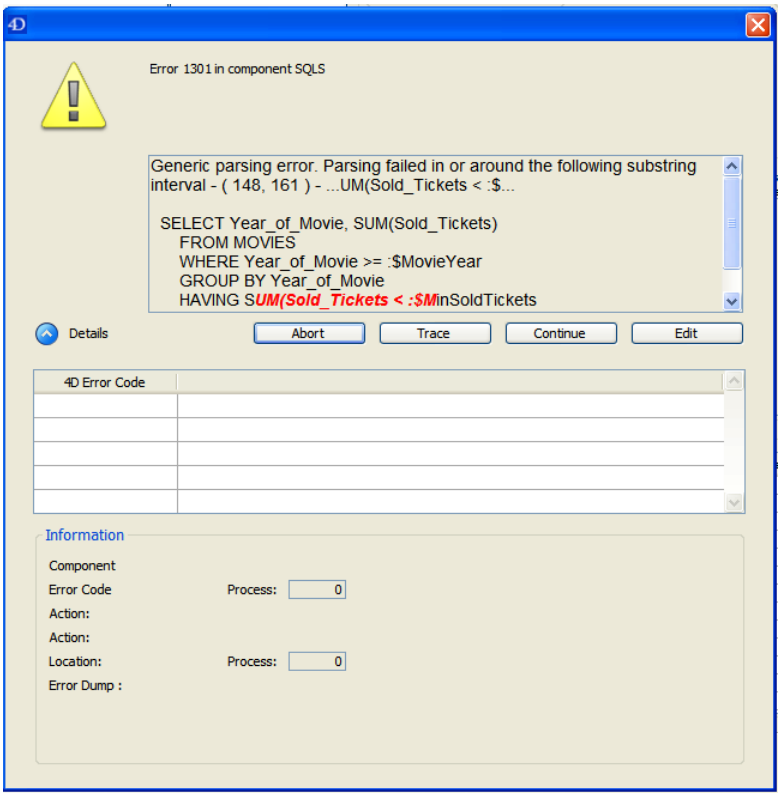
```
  ` Using dynamic SQL by EXECUTE IMMEDIATE
ARRAY LONGINT(aSoldTickets;0)
ARRAY TEXT(aTitles;0)
C_TEXT($tQueryTxt)

$tQueryTxt:=""
$tQueryTxt:=$tQueryTxt+"SELECT Title, Sold_Tickets"
$tQueryTxt:=$tQueryTxt+" FROM MOVIES"
$tQueryTxt:=$tQueryTxt+" WHERE Sold_Tickets > (SELECT AVG(Sold_Tickets) FROM MOVIES)"
$tQueryTxt:=$tQueryTxt+" ORDER BY 1"
$tQueryTxt:=$tQueryTxt+" INTO :aTitles, :aSoldTickets"
Begin SQL
   EXECUTE IMMEDIATE :$tQueryTxt;
End SQL
  ` Intialize the rest of the list box columns in order to visualise 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 Video Library database and go to the main window. You can then choose the query mode and press the "Subqueries" button.

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 SUM(Sold_Tickets), we have SUM(Sold_Tickets.

```
` Debugging SQL code
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, opens the debugger window and gives some more detailed information about the error and the place where it occurred. It is very easy to fix it by simply pressing the Edit button.



If the error is more complex, the application gives much 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 Video Library 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.

```
   ` Using ON ERR CALL command
ARRAY LONGINT(aSoldTickets;0)
ARRAY INTEGER(aMovieYear;0)
C_LONGINT($MovieYear;$MinSoldTickets;SQL_Error)
$MovieYear:=1979
$MinSoldTickets:=10000000
SQL_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 (SQL_Error#0)
   ALERT("SQL Error number: "+String(SQL_Error))
End if
```

The SQL_Error_Handler method is as follows:

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

To test the above example, in the main window of the Video Library database, press the "Using ON ERR CALL" button.

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.

```
` DDL

Begin SQL
   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 Video Library 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.

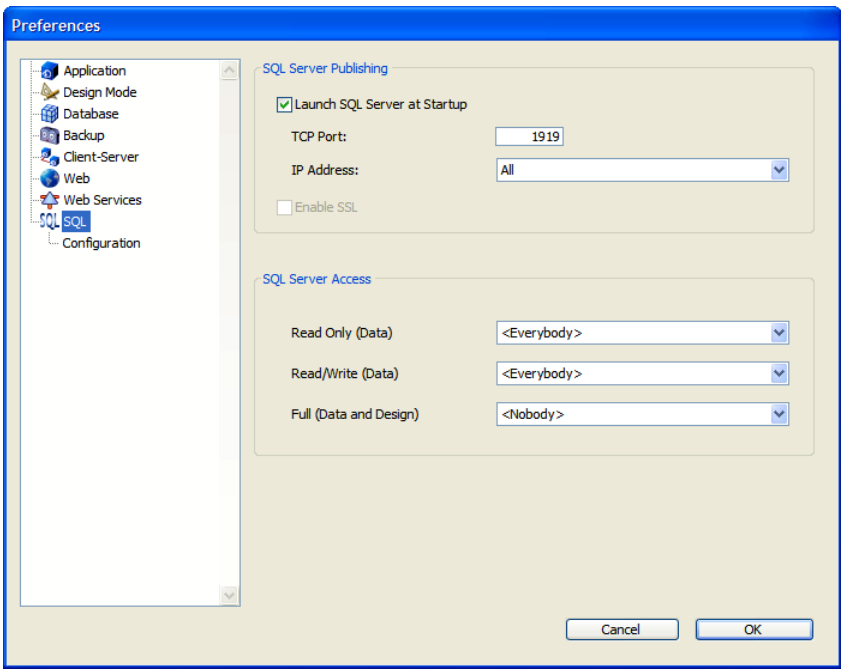
4D v11 allows you to connect to any external ODBC data source directly from the language and execute SQL queries on that external connected database.
Here are the 4D commands that allow you to manage a connection with an external data source:
GET DATA SOURCE LIST can be used to get the list of the data sources installed on the machine.
USE EXTERNAL DATABASE allows you to connect to an external database via a data source installed on the machine.
USE INTERNAL DATABASE can be used to close any external connection and to reconnect to the local 4D database.
Get current data source tells you the current data source of the application.
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.

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

  ` Do someting 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 $My_ORACLE_DSN="Test_Oracle_10g"
    USE EXTERNAL DATABASE($My_ORACLE_DSN;"scott";"tiger")

     ` The current DSN is the ORACLE one
    $Crt_DSN:=Get current data source
    ALERT("The current DSN is "+$Crt_DSN)
    ARRAY TEXT(aTitles;0)
    ARRAY LONGINT(aNrActors;0)
    ARRAY LONGINT(aSoldTickets;0)
    ARRAY INTEGER(aMovieYear;0)
    ARRAY TEXT(aTitles;0)
    ARRAY TEXT(aDirectors;0)
    ARRAY TEXT(aMedias;0)

     ` Do something  on the external (ORACLE) database
  Begin SQL
      SELECT ENAME FROM EMP INTO :aTitles
  End SQL

     ` Close the external connexion opened with the USE EXTERNAL DATABASE command
    USE LOCAL DATABASE
     ` The current DSN becomes the local one
    $Crt_DSN:=Get current data source
    ALERT("The current DSN is "+$Crt_DSN)
Else
    ALERT("ORACLE DSN not installed")
End if
```

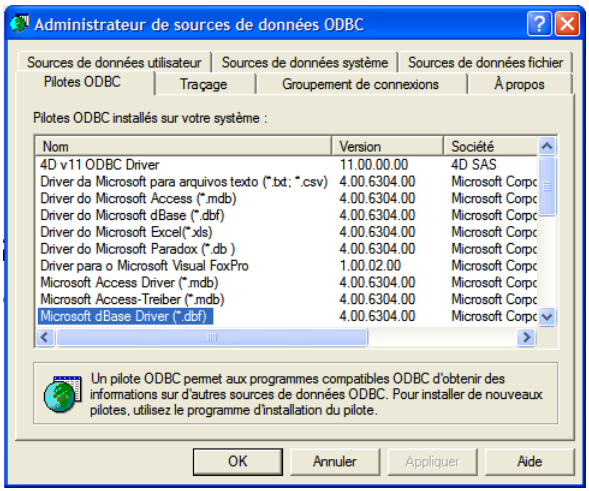
To test the above example, in the main window of the Video Library database, press the "Connect to ORACLE" button.

You can connect to the 4D SQL Engine from any external database via the ODBC Driver for 4D v11.
Here is a small example demonstrating how to connect a 4D database to another 4D database via the ODBC driver:
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 application Preferences, on the SQL/Configuration page:

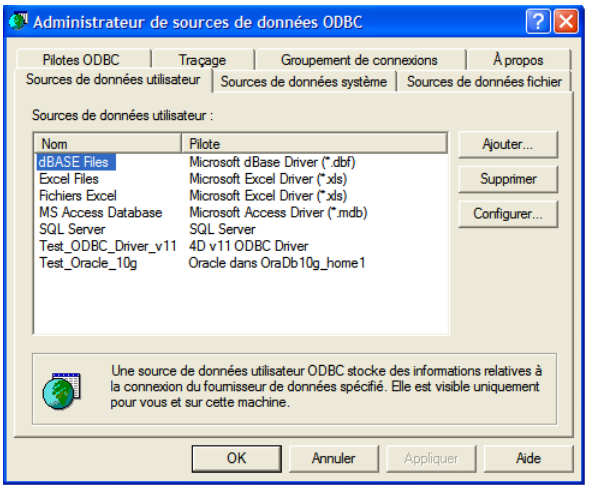


Quit and restart the example database from the Server folder to activate the SQL Server.

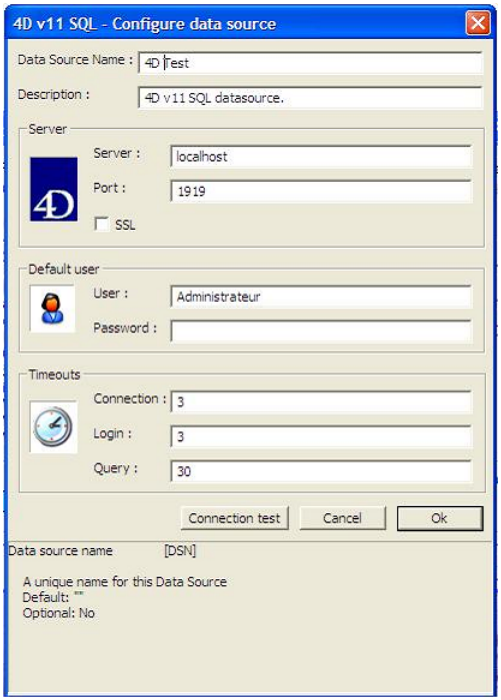
4. Install the 4D ODBC Driver for 4D v11, then check whether it appears in the ODBC Data Source Administrator:



5. Create a new data source named "Test_ODBC_Driver_v11"



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



6. 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:

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

REDUCE SELECTION([MOVIES];0)

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

   ` Do someting 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 of type User 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
   USE EXTERNAL DATABASE($My_4D_DSN;"Administrator";"")
   If (Ok=1)

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

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

      ` 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 connexion opened with the USE EXTERNAL DATABASE command
      USE LOCAL DATABASE
      ` The current DSN becomes the local one
      $Crt_DSN:=Get current data source
      ALERT("The current DSN is "+$Crt_DSN)

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

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.

2

Using SQL in 4D

Using SQL in 4D

version 11

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.

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 ODBC commands of 4D, found in the "External Data Source" theme (ODBC SET PARAMETER, ODBC EXECUTE, etc.). These commands have been modified to work with 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.

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 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_ALPHA(80;vName)
vName:=Request("Name:")
ODBC EXECUTE("SELECT age FROM PEOPLE WHERE name=<<vName>>")
```

or:

```
C_ALPHA(80;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]).

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 External Data Source (ODBC) 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 ODBC EXECUTE command, you can also use the additional parameters:

```
ODBC 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 ODBC 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 ODBC LOAD RECORD.

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

• Using ODBC commands:

```
ARRAY INTEGER(aBirthYear;0)
C_STRING(40;vName)
vName:="Smith"
$SQLStm:="SELECT Birth_Year FROM PEOPLE WHERE ename= <<vName>>"
ODBC EXECUTE($SQLStm;aBirthYear)
While (Not (ODBC End Selection))
    ODBC 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:

```
ODBC LOAD RECORD(ODBC All Records)
```

• Using Begin SQL/End SQL tags:

```
ARRAY INTEGER(aBirthYear;0)
C_STRING(40;vName)
vName:="Smith"
Begin SQL
    SELECT Birth_Year FROM PEOPLE 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 ODBC commands:

```
C_STRING(40;vName)
vName:="Smith"
$SQLStm:="SELECT Birth_Year FROM PEOPLE WHERE ename= <<vName>>"
ODBC EXECUTE($SQLStm;[MYTABLE]Birth_Year)
While (Not (ODBC End Selection))
    ODBC 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 from the PEOPLE table in the [MYTABLE]Birth_Year field.

• Using Begin SQL/End SQL tags:

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

In this situation, 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.

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:

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

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

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

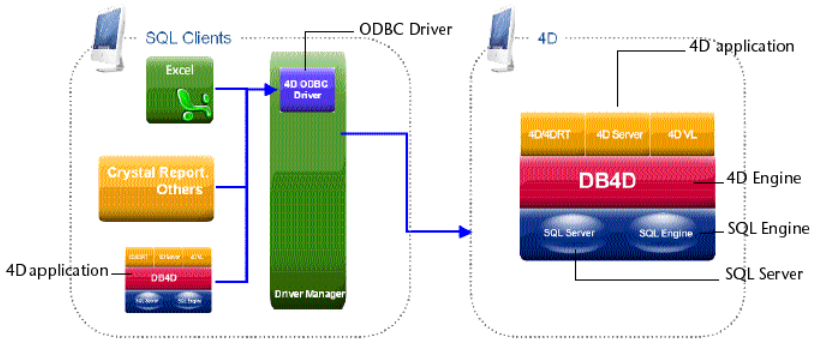
... the CONCAT(<<vLastName>>,<<vFirstName>>) function is called for each record of the table; in other words, the expression is evaluated for each record.

4D v11 includes a powerful SQL server that allows external access to data stored in the 4D database. This access is carried out using a 4D ODBC driver.

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 takes place 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 connect to the SQL server of 4D. This is summarized in the following diagram:

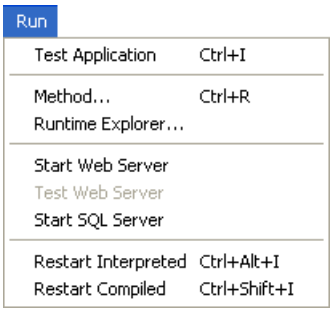


The 4D ODBC driver must be installed on the machine of the SQL Client part. The installation and configuration of the 4D ODBC driver is detailed in a separate manual.

Starting and Stopping the 4D SQL Server

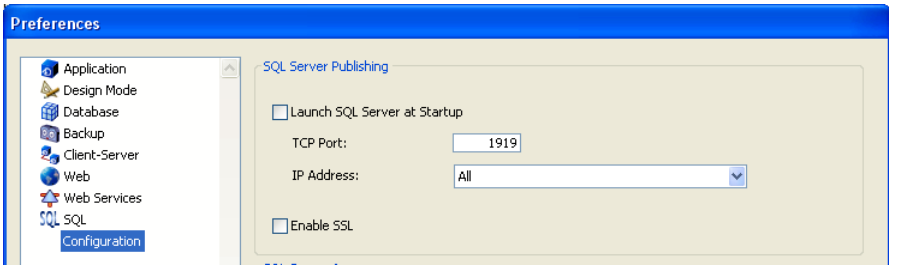
In 4D v11, the SQL server can be started and stopped in three ways:

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



When the server is launched, this menu item changes to **Stop SQL Server**.

• Automatically on startup of the application, via the Preferences. To do this, display the **SQL**/**Configuration** page and check the **Launch SQL Server at Startup** option:

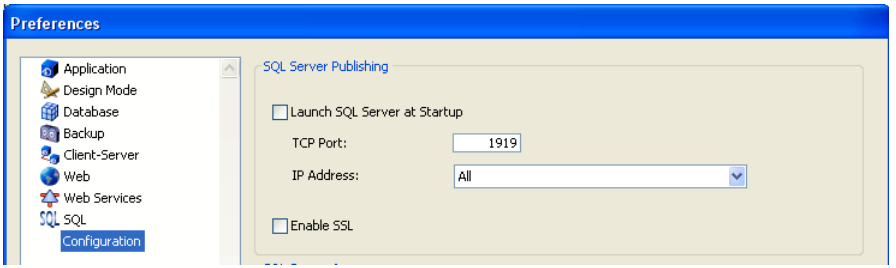


• By programming, using the START SQL SERVER and STOP SQL SERVER commands (“SQL” theme).
When the SQL server is stopped (or when it has not been started), 4D will not respond to any external SQL queries.

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/Configuration** page of the database Preferences:



• 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 1919. 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.

Note: If you pass 0, 4D will use the default TCP port number, i.e. 1919.

• **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.

Note: 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.

• **Enable SSL**: This option indicates whether the SQL server must enable the SSL protocol for processing SQL connections.

4D Database Access Control

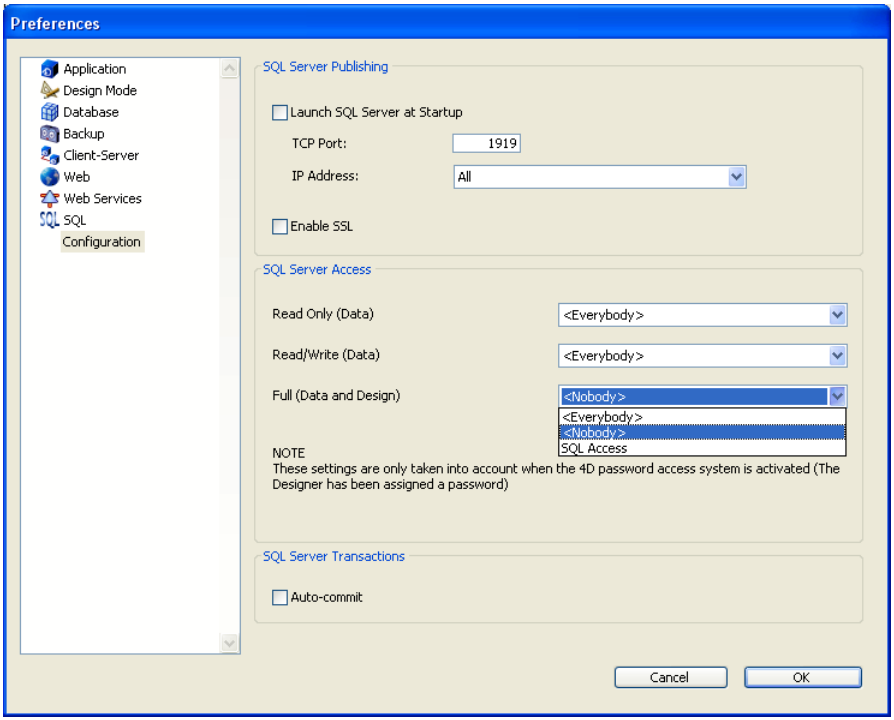
For security reasons, it is possible to limit actions that external queries sent to the SQL server can perform in the 4D database.

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 are made on the **SQL/Configuration** page of the database Preferences:



You can configure three separate types of access to the 4D database 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 via ODBC.

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 "Principles for integrating 4D and the 4D SQL engine" section.

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 v11: 32767.

• Maximum number of columns (fields) in a table: Theoretically two billion columns (fields), but for compatibility reasons with 4D v11: 32767.
In 4D v11 “Standard edition,” the maximum number of columns is limited to 511.

• Maximum number of rows (records) in a table: one billion. For a subrecord field, the limit is one billion subrecords for each record.

• Maximum number of index keys: one billion x 64.

• 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 v11
Varchar	Alphanumeric text	Text
Real	Floating point number in the range of +/-3,4E38	Real
Numeric	Number between +/- 2E64	Integer 64 bits
Float	Floating point number (virtually infinite)	Real
Smallint	Number between -32 768 and 32 767	Integer
Int	Number between -2 147 483 648 and 2 147 483 647	Longint
Bit	A field that can only take the values TRUE or FALSE	Boolean
Boolean	A field that can only take the values TRUE or FALSE	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 in Day Month Year Hours:Minutes:Seconds: Milliseconds format	Date and Time parts handled separately (automatic conversion)
Duration	Time in Day:Hours:Minutes:Seconds:Milliseconds format	Time
Interval	Time in Day:Hours:Minutes:Seconds:Milliseconds 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.

Note: 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 v11, 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

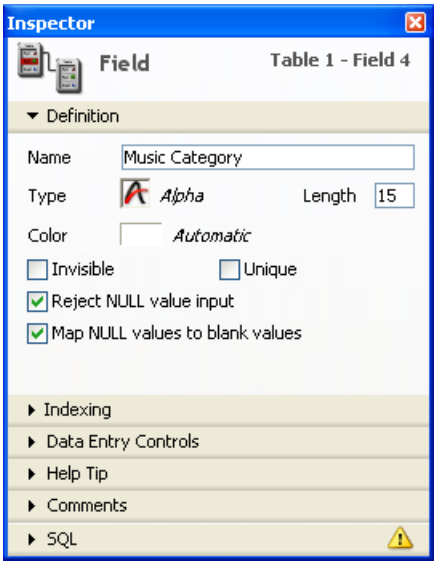
On the other hand, this mechanism in principle does not apply to processing carried out at the level of the 4D database engine, such as queries. In fact, searching for an "blank" value (for example myvalue=0) will not find records storing the NULL value, and vice versa. When both types of values (default 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 v11 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

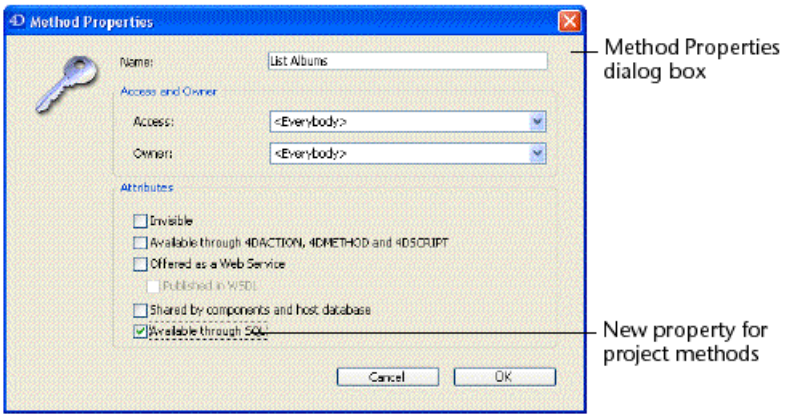
The **Reject NULL value input** field property is used to prevent the storage of NULL values:



When this attribute is checked for a field, it will not be possible to store the 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.

"Available through SQL" Option
A security property has been added for 4D project methods: **Available through SQL**:



When it is 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 Preferences 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.

Auto-commit
An option found on the SQL/Configuration page of the 4D Preferences can be used to activate the auto-commit mechanism in the 4D SQL engine:

The purpose of the auto-commit mode is to preserve the referential integrity of the data. When this option is checked, any SELECT, INSERT, UPDATE and DELETE (SIUD) queries not already carried out within a transaction are automatically included in an ad hoc transaction. This guarantees that the queries will be executed in their entirety or, in the case of an error, completely cancelled.

Queries already included in a transaction (custom management of referential integrity) are not affected by this option.

When this option is not checked, no automatic transaction is generated (except for the SELECT... FOR UPDATE queries, please refer to the SELECT command). By default, this option is not checked.

You can also manage this option by programming using the SET DATABASE PARAMETER command (see the SET DATABASE PARAMETER command in the *4D Language Reference* manual).

Note: Only local databases queried by the 4D SQL engine are affected by this parameter. In the case of external databases, the auto-commit mechanism is handled by the remote SQL engines.

System Tables
The SQL catalogue of 4D includes six 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 and _USER _CONS_ COLUMNS.

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

_USER_TABLES		**Describes the user tables of the database**
TABLE_NAME	VARCHAR	Table name
TEMPORARY	BOOLEAN	True if the table is temporary; otherwise, false
TABLE_ID	INT64	Table number

_USER_COLUMNS		**Describes the columns of the user tables of the database**
TABLE_NAME	VARCHAR	Table name
COLUMN_NAME	VARCHAR	Column name
DATA_TYPE	INT32	Column type
DATA_LENGTH	INT32	Column length
NULLABLE	BOOLEAN	True if column accepts NULL values;otherwise, false
TABLE_ID	INT64	Table number
COLUMN_ID	INT64	Column number

_USER_INDEXES		**Describes the user indexes of the database**
INDEX_ID	VARCHAR	Index number
INDEX_NAME	VARCHAR	Index name
INDEX_TYPE	INT32	Index type
TABLE_NAME	VARCHAR	Name of table with index
UNIQUENESS	BOOLEAN	True if index imposes a uniqueness constraint; otherwise, false
TABLE_ID	INT64	Number of table with index

_USER_IND_COLUMNS		**Describes the columns of user indexes of the database**
INDEX_ID	VARCHAR	Index number
INDEX_NAME	VARCHAR	Index name
TABLE_NAME	VARCHAR	Name of table with index
COLUMN_NAME	VARCHAR	Name of column with index
POSITION	INT32	Position of column in index
TABLE_ID	INT64	Number of table with index
COLUMN_ID	INT64	Column number

_USER_CONSTRAINTS		Describes the user constraints of the database
CONSTRAINT_ID	VARCHAR	Constraint number
CONSTRAINT_NAME	VARCHAR	Constraint name
CONSTRAINT_TYPE	VARCHAR	Constraint type
TABLE_NAME	VARCHAR	Name of table with constraint
TABLE_ID	INT64	Number of table with constraint
DELETE_RULE	VARCHAR	Delete rule – CASCADE or RESTRICT
RELATED_TABLE_NAME	VARCHAR	Name of related table
RELATED_TABLE_ID	INT64	Number of related table

_USER_CONS_COLUMNS		Describes the columns of user constraints of the database
CONSTRAINT_ID	VARCHAR	Constraint number
CONSTRAINT_NAME	VARCHAR	Constraint name
TABLE_NAME	VARCHAR	Name of table with constraint
TABLE_ID	INT64	Number of table withconstraint
COLUMN_NAME	VARCHAR	Name of column with constraint
COLUMN_ID	INT64	Number of column with constraint
POSITION	INT32	Position of column with constraint
RELATED_COLUMN_NAME	VARCHAR	Name of related column in a constraint
RELATED_COLUMN_ID	INT32	Number of related column in a constraint

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 ODBC LOGIN command.
- To switch from one database to another using the USE EXTERNAL DATABASE and USE INTERNAL DATABASE commands.

3

SQL Commands

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. More specifically, this refers to the CREATE TABLE, ALTER TABLE, DROP INDEX and DROP TABLE type 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 "Syntax rules" theme.

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** {*int_number* | **ALL**}]
[**OFFSET** *int_number*]
[**INTO** {*4d_language_reference*, ..., *4d_language_reference* | **LISTBOX** *4d_language_reference*}]
[**FOR UPDATE**]

Description

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 select_item 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 the table_reference(s) 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.

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 an int_number.

The optional OFFSET clause can be used to set a number (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. You may also pass the **LISTBOX** keyword to place the data into a 4d_language_reference listbox.

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.

Examples

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;$Offset;$Limit)
ARRAY INTEGER(aMovieYear;0)
ARRAY LONGINT(aTicketsSold;0)
$MovieYear:=1979
$MinTicketsSold:=10000000
$Offset:=5
$Limit:=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
    OFFSET :$Offset
    LIMIT :$Limit
    ORDER BY 1
    INTO :aMovieYear, :aTicketsSold;
End SQL
```

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');
```

3. Given the table SALES_PERSONS where QUOTA is the amount of sales expected to be earned by a given salesperson and SALES is the actual amount of sales made by a given salesperson.

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

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
```

See Also

4d_language_reference, search_condition, select_item, sort_list, subquery, table_reference.

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

Description

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(s) passed indicate the name(s) of the column(s) where the values are to be inserted. If the column_reference(s) are not 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).

The **VALUES** keyword is used to pass the value(s) to be placed in the column(s) specified. You can either pass arithmetic_expression(s) 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 column_reference(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 INSERT command is supported in both single- and multi-row queries. However, a multi-row INSERT statement does not allow UNION and JOIN operations.

Examples

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

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

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');
```

See Also

arithmetic_expression, column_reference, DELETE, subquery.

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

Description

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(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.

The UPDATE command is supported for both queries and subqueries; however, a positioned UPDATE statement is not supported.
CASCADE update is implemented in 4D, but the SET NULL and SET DEFAULT delete rules are not supported.

Example

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';
```

See Also

arithmetic_expression, DELETE, search_condition.

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

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;
```

See Also

INSERT, search_condition, UPDATE.

CREATE TABLE [**IF NOT EXISTS**] *sql_name*({*column_definition* |*table_constraint*}, ... , {*column_definition* |*table_constraint*})

Description

The CREATE TABLE command is used to create a table named sql_name having the fields specified by passing one or more column_definition(s) and/or table_constraint(s). 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.

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.

Examples

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

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

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);
```

See Also

ALTER TABLE, column_definition, DROP TABLE, table_constraint.

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

Description

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.

Examples

1. Here is a simple example which removes the ACTOR_FANS table:

```
DROP TABLE ACTOR_FANS
```

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
```

See Also

ALTER TABLE, CREATE TABLE.

ALTER TABLE *sql_name*
{**ADD** *column_definition* |
DROP *sql_name* |
ADD *primary_key_definition* |
DROP PRIMARY KEY |
ADD *foreign_key_definition* |
DROP CONSTRAINT *sql_name*}

Description

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.
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.

Example

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;
```

See Also

column_definition, CREATE TABLE, DROP TABLE, foreign_key_definition, primary_key_definition.

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 column_reference(s) of an existing table (sql_name). 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)
```

See Also

column_reference, 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
```

See Also

CREATE INDEX.

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
```

See Also

ALTER TABLE, 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
```

See Also

ALTER TABLE, LOCK TABLE.

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 is used to prepare an executable form which is subsequently destroyed after execution of the command.

Example

This example illustrates how this works:

```
C_LONGINT($NoMovies)
C_TEXT($tQueryTxt)
$NoMovies:=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")
```

4

Syntax rules

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.

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

Description

A 4d_function_call must be entered as follows:

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

The sql_name of the function is preceded by the **FN** keyword and followed by the arithmetic_expression(s) passed. These arithmetic_expression(s) will be returned in the form of the sql_data_type_name passed.

Example

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

```
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
```

See Also

arithmetic_expression, sql_data_type_name, sql_name.

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

Description

A 4d_language_reference specifies the SQL name of the variable(s) to which data will be assigned. This name can be passed in one of the following manners:

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

See Also

SELECT, sql_name.

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.

Example

Here is a simple example:

```
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');
```

See Also

arithmetic_expression, predicate, subquery.

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.

See Also

case_expression, column_reference, command_parameter, function_call, INSERT.

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'
```

See Also

arithmetic_expression, predicate.

case_expression

Description

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

Example

Here is a simple example:

```
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
```

See Also

arithmetic_expression, search_condition.

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

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** and/or **UNIQUE** 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 (except for **NULL** values, which are not considered to be identical).

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);
```

See Also

ALTER TABLE, CREATE TABLE, sql_data_type_name.

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

See Also

CREATE INDEX, INSERT.

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

Description

A command_parameter consists of 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*

See Also

arithmetic_expression.

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

Description

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

See Also

arithmetic_expression, predicate, subquery.

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.

Example

Here is a simple example:

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

See Also

predicate, subquery.

[**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(s)) set in another table in order to ensure data integrity. The **FOREIGN KEY** constraint is used to pass the column_reference(s) to be defined as the foreign keys (which match the primary keys of another table).

An optional **CONSTRAINT** (sql_name) can also precede the **FOREIGN KEY** constraint passed in order to limit the values that can be inserted into the column_reference(s).

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(s) 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**).

Example

Here is a simple example:

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

See Also

ALTER TABLE, column_reference, primary_key_definition.

sql_function_call |
4d_function_call

Description

A function_call can consist of either an sql_function_call or a 4d_function_call. Both types of functions manipulate data and return results and can operate on one or more arguments.

Example

Here is a simple example:

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

See Also

4d_function_call.

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_expression(s) that are passed or the result of a subquery.

Example

Here is a simple example:

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

See Also

arithmetic_expression, predicate.

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

Description

An is_null_predicate is used to find arithmetic_expression(s) with **NULL** values. You can also pass the **NOT** keyword to find those without **NULL** values.

Example

Here is a simple example:

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

See Also

arithmetic_expression, 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.

Example

Here are some simple examples:

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

or

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

or

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

See Also

arithmetic_expression, predicate.

int_number | *fractional_number* | *sql_string*

Description

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

See Also

arithmetic_expression.

predicate

Description

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

See Also

all_or_any_predicate, between_predicate, comparison_predicate, exists_predicate, in_predicate, is_null_predicate, like_predicate, search_condition.

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

Description

A primary_key_definition is used to pass the sql_name(s) of a column or combination of columns 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

Here is a simple example:

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

See Also

ALTER TABLE, foreign_key_definition.

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_condition(s) 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');
```

See Also

DELETE, predicate, SELECT, UPDATE.

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
```

See Also

arithmetic_expression, SELECT, sql_name, sql_string.

{*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.

See Also

column_reference, SELECT.

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

Description

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

See Also

4d_function_call.

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:

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]

See Also

ALTER TABLE, CREATE INDEX, CREATE TABLE, DELETE, DROP INDEX, DROP TABLE, EXECUTE IMMEDIATE, INSERT, LOCK TABLE, UNLOCK TABLE, UPDATE.

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.

For example:

my string	'my string'
string with ' inside it	'string with ' ' inside it'
'string already in quotes'	' ' 'string already in quotes' ' '

See Also

DELETE, INSERT, UPDATE.

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*]

Description

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.

See Also

search_condition, SELECT, select_item, sort_list, table_reference.

{*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).

Example

Here is a simple example:

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

See Also

CREATE TABLE, foreign_key_definition, primary_key_definition.

{*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).

See Also

SELECT, sql_name, sql_string.

5

Transactions

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 ODBC commands applied to the local database are always executed in the same context as standard 4D commands.

The following examples illustrate the different combinations of transactions.

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

```
ODBC 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
ODBC EXECUTE("START")   `Another transaction in the current process
ODBC CANCEL LOAD
    `This statement is executed in the same process
ODBC EXECUTE("INSERT INTO emp (NAME) VALUES ('Smith')")
ODBC CANCEL LOAD
ODBC EXECUTE("ROLLBACK")  `Cancels internal transaction of the process
CANCEL TRANSACTION  `Cancels external transaction of the process
ODBC LOGOUT
```

Only “John” will be added to the emp table:

```
ODBC LOGIN(SQL_INTERNAL ;"";"")
START TRANSACTION
Begin SQL
    INSERT INTO emp
    (NAME)
    VALUES ('John');
End SQL
ODBC EXECUTE("START")
ODBC CANCEL LOAD
ODBC EXECUTE("INSERT INTO emp (NAME) VALUES ('Smith')")
ODBC CANCEL LOAD
ODBC EXECUTE("ROLLBACK")  `Cancels internal transaction of the process
VALIDATE TRANSACTION  `Validates external transaction of the process
ODBC LOGOUT
```

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

```
ODBC LOGIN(SQL_INTERNAL ;"";"")
START TRANSACTION
Begin SQL
    INSERT INTO emp
    (NAME)
    VALUES ('John');
End SQL
ODBC EXECUTE("START")
ODBC CANCEL LOAD
ODBC EXECUTE("INSERT INTO emp (NAME) VALUES ('Smith')")
ODBC CANCEL LOAD
ODBC EXECUTE("COMMIT")  `Validates internal transaction of the process
CANCEL TRANSACTION  `Cancels external transaction of the process
ODBC LOGOUT
```

“John” and “Smith” will be added to the emp table:

```
ODBC LOGIN(SQL_INTERNAL ;"";"")
START TRANSACTION
Begin SQL
    INSERT INTO emp
    (NAME)
    VALUES ('John');
End SQL
ODBC EXECUTE("START")
ODBC CANCEL LOAD
ODBC EXECUTE("INSERT INTO emp (NAME) VALUES ('Smith')")
ODBC CANCEL LOAD
ODBC EXECUTE("COMMIT")  `Validates internal transaction of the process
VALIDATE TRANSACTION   `Validates external transaction of the process
ODBC LOGOUT
```

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 has no effect. The keyword **TRANSACTION** is optional.

Example

Here is a simple example of how to use and validate a transaction:

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

See Also

COMMIT, ROLLBACK.

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 TRANSACTION command.

See Also

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 function:

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

See Also

COMMIT.

6

Functions

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(s) containing the necessary arguments.

ABS (*arithmetic_expression*)

Description

The ABS function returns the absolute value of the arithmetic_expression.

Example

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

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

ACOS (*arithmetic_expression*)

Description

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.

Example

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

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

See Also

COS.

ASCII (*arithmetic_expression*)

Description

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.

Example

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

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

ASIN (*arithmetic_expression*)

Description

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

Example

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

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

See Also

SIN.

ATAN (*arithmetic_expression*)

Description

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

Example

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

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

See Also

TAN.

ATAN2 (*arithmetic_expression*, *arithmetic_expression*)

Description

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.

Example

This example returns the arc tangent of the x and y coordinates passed:

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

See Also

ATAN, TAN.

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.

Example

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
```

See Also

COUNT, SUM.

BIT_LENGTH (*arithmetic_expression*)

Description

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

Example

Here is a simple example:

```
SELECT BIT_LENGTH( '01101011' );
    ` returns 8
```

See Also

OCTET_LENGTH.

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

Description

The CAST function converts the arithmetic_expression to the sql_data_type_name passed.

Example

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 (*arithmetic_expression*)

Description

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

Example

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

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

See Also

FLOOR.

CHAR (*arithmetic_expression*)

Description

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

Example

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 (*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
```

See Also

LENGTH.

COALESCE (*arithmetic_expression_commalist*)

Description

The COALESCE function returns the first non-null expression from the list of arithmetic_expression(s) 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 (*arithmetic_expression*, *arithmetic_expression*)

Description

The CONCAT function returns the two arithmetic_expression(s) 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;
```

See Also

CONCATENATE.

CONCATENATE (*arithmetic_expression, arithmetic_expression*)

Description

The CONCATENATE function returns the two arithmetic_expression(s) passed as a single concatenated string.

Example

See the example for the CONCAT function.

See Also

CONCAT.

COS (*arithmetic_expression*)

Description

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

Example

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;
```

See Also

SIN.

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;
```

See Also

TAN.

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

Description

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 rows in a table, including duplicate and null values.

Example

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

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

See Also

AVG, SUM.

CURDATE ( )

Description

The CURDATE function returns the current date.

Example

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

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

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

    SELECT *
    FROM INVOICES
    INTO :aDate;
End SQL
    `the aDate array will return the INSERT command execution date and time.
```

See Also

CURRENT_DATE, CURTIME.

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

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

    SELECT *
    FROM INVOICES
    INTO :aDate;
End SQL
    `the aDate array will return the INSERT command execution date and time.
```

See Also

CURRENT_TIME, CURRENT_TIMESTAMP.

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

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

   SELECT *
   FROM INVOICES
   INTO :aDate;
End SQL
   `the aDate array will return the INSERT command execution date and time.
```

See Also

CURRENT_DATE, 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 STRING(30;aDate;0)
Begin SQL
   CREATE TABLE INVOICES
   (INV_DATE VARCHAR(40));

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

   SELECT *
   FROM INVOICES
   INTO :aDate;
End SQL
   `the aDate array will return the INSERT command execution date and time.
```

See Also

CURRENT_DATE, CURRENT_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 STRING(30;aDate;0)
Begin SQL
    CREATE TABLE INVOICES
    (INV_DATE VARCHAR(40));

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

    SELECT *
    FROM INVOICES
    INTO :aDate;
End SQL
    `the aDate array will return the INSERT command execution date and time.
```

See Also

CURDATE, CURRENT_TIME.

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 (*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
```

See Also

DAYNAME, DAYOFMONTH, DAYOFWEEK, DAYOFYEAR.

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);
```

See Also

DAY, DAYOFMONTH, DAYOFWEEK, DAYOFYEAR.

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;
```

See Also

DAY, DAYNAME, DAYOFWEEK, DAYOFYEAR.

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;
```

See Also

DAY, DAYNAME, DAYOFMONTH, 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;
```

See Also

DAY, DAYNAME, DAYOFMONTH, DAYOFWEEK.

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
```

See Also

RADIANS.

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
```

See Also

SQRT.

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

Description

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 (*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
```

See Also

CEILING.

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;
```

See Also

MINUTE, SECOND.

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

Description

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 (*arithmetic_expression*, *arithmetic_expression*)

Description

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.

Example

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;
```

See Also

RIGHT.

LENGTH (*arithmetic_expression*)

Description

The LENGTH function returns the number of characters in the arithmetic_expression.

Example

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
```

See Also

CHAR_LENGTH.

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

LOCATE (*arithmetic_expression*, *arithmetic_expression*)

Description

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.

Example

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;
```

See Also

POSITION, SUBSTRING.

LOG (*arithmetic_expression*)

Description

The LOG function returns the natural logarithm of the arithmetic_expression.

Example

This example returns the natural logarithm of 50:

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

See Also

LOG10.

LOG10 (*arithmetic_expression*)

Description

The LOG10 function returns the base 10 logarithm of the arithmetic_expression.

Example

This example returns the logarithm in base 10 of 50:

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

See Also

LOG.

LOWER (*arithmetic_expression*)

Description

The LOWER function returns the arithmetic_expression passed as a string where all the characters are in lowercase.

Example

This example will return the names of products in lowercase:

```
SELECT LOWER (Name)
FROM PRODUCTS;
```

See Also

UPPER.

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 beginningof the first arithmetic_expression.

Example

This example simply removes any empty spaces from the beginning of product names:

```
SELECT LTRIM(Name)
FROM PRODUCTS;
```

See Also

RTRIM, TRIM.

MAX (*arithmetic_expression*)

Description

The MAX function returns the maximum value of the arithmetic_expression.

Example

See the examples from SUM and AVG.

See Also

MIN.

MILLISECOND (*arithmetic_expression*)

Description

The MILLISECOND function returns the millisecond part of the time passed in arithmetic_expression.

Example

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;
```

See Also

MINUTE, SECOND.

MIN (*arithmetic_expression*)

Description

The MIN function returns the minimum value of the arithmetic_expression.

Example

See the examples from SUM and AVG.

See Also

MAX.

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;
```

See Also

HOUR, SECOND.

MOD (*arithmetic_expression*, *arithmetic_expression*)

Description

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 (*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;
```

See Also

DAYOFMONTH, 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);
```

See Also

DAYOFMONTH, MONTH.

NULLIF (*arithmetic_expression*, *arithmetic_expression*)

Description

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 arithmetic_expression(s) must be comparable.

Example

This example returns Null if the total of the invoice is 0:

```
NULLIF(INVOICE_TOTAL,0);
```

OCTET_LENGTH (*arithmetic_expression*)

Description

The OCTET_LENGTH function returns the number of bytes of the arithmetic_expression, including any trailing whitespace.

Example

Here is a simple example:

```
SELECT OCTET_LENGTH (MyBinary_col)
FROM MyTable
WHERE MyBinary_col = '93FB';
    ` returns 2
```

See Also

BIT_LENGTH.

PI ( )

Description

The PI function returns the value of the constant Pi (π).

Example

See example from DEGREES.

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.

Example

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;
```

See Also

LOCATE, SUBSTRING.

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
```

See Also

EXP, SQRT.

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;
```

See Also

MONTH, YEAR.

RADIANS (*arithmetic_expression*)

Description

The RADIANS function returns the number of radians of the arithmetic_expression. The arithmetic_expression represents the angle expressed in degrees.

Example

This example returns the number of radians of a 30 degree angle:

```
RADIANS ( 30 );
    `returns the value 0,5236
```

See Also

DEGREES.

RAND ([*arithmetic_expression*])

Description

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 (*arithmetic_expression*, *arithmetic_expression*)

Description

The REPEAT function returns the first arithmetic_expression repeated the requested number of times (passed in second arithmetic_expression).

Example

Here is a simple example of how it works:

```
SELECT REPEAT( 'repeat', 3 )
    ` returns 'repeatrepeatrepeat'
```

REPLACE (*arithmetic_expression*, *arithmetic_expression*, *arithmetic_expression*)

Description

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.

Example

This example will replace the word "Francs" by "Euro":

```
SELECT Name, REPLACE(Currency, 'Francs', 'Euro')
FROM PRODUCTS;
```

See Also

TRANSLATE.

RIGHT (*arithmetic_expression*, *arithmetic_expression*)

Description

The RIGHT 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;
```

See Also

LEFT.

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
```

See Also

TRUNC, TRUNCATE.

RTRIM (*arithmetic_expression*[, *arithmetic_expression*])

Description

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.

Example

This example removes any empty spaces from the ends of the product names:

```
SELECT RTRIM(Name)
FROM PRODUCTS;
```

See Also

LTRIM, TRIM.

SECOND (*arithmetic_expression*)

Description

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;
```

See Also

HOUR, MINUTE.

SIGN (*arithmetic_expression*)

Description

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 (*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;
```

See Also

COS.

SPACE (*arithmetic_expression*)

Description

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.

Example

This example adds three spaces in front of the last names of the PEOPLE table:

```
SELECT CONCAT(SPACE(3),PERSONS.LastName) FROM PEOPLE;
```

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
```

See Also

EXP.

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.

Example

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';
```

See Also

LOCATE, POSITION.

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.

Example

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
```

See Also

AVG, COUNT.

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 arc cosine of the angle expressed in radians:

```
SELECT TAN(radians)
FROM TABLES_OF_ANGLES;
```

See Also

COT.

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'
```

See Also

REPLACE.

TRIM ({**LEADING** |**TRAILING** |**BOTH**} [*arithmetic_expression*] **FROM** *arithmetic_expression*)

TRIM (*arithmetic_expression*[**FROM** *arithmetic_expression*])

Description

The TRIM function removes empty spaces or specified characters from an arithmetic_expression.

In the first syntax, 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**. The optional first arithmetic_expression passed indicates the specific characters to be removed from the second arithmetic_expression. If it is omitted, only the empty spaces will be removed.

In the second syntax, the first arithmetic_expression indicates the characters to be removed from the second optional arithmetic_expression. If this second optional arithmetic_expression is omitted, only the empty spaces will be removed from the arithmetic_expression passed.

Example

This example removes any empty spaces from the product names:

```
SELECT TRIM(Name)
FROM PRODUCTS;
```

See Also

LTRIM, RTRIM.

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
```

See Also

ROUND, 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.

Example

See the example for the TRUNC function.

See Also

ROUND, TRUNC.

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;
```

See Also

LOWER.

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);
```

See Also

DAYOFWEEK.

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;
```

See Also

DAYOFYEAR.

7

Appendix

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	NOT RUNNING
1004	ACCESS DENIED
1005	FAILED TO LOCK SYNCHRONIZATION PRIMITIVE
1006	FAILED TO UNLOCK SYNCHRONIZATION PRIMITIVE
1007	SQL SERVER IS NOT AVAILABLE

Semantic errors

1101	TABLE DOES NOT EXIST
1102	COLUMN DOES NOT EXIST
1103	TABLE NOT DECLARED IN FROM CLAUSE
1104	AMBIGUOUS COLUMN NAME
1105	TABLE ALIAS SAME AS TABLE NAME
1106	DUPLICATE TABLE ALIAS
1107	DUPLICATE TABLE IN FROM CLAUSE
1108	INCOMPATIBLE TYPES
1109	INVALID ORDER BY INDEX
1110	WRONG AMOUNT OF PARAMETERS
1111	INCOMPATIBLE PARAMETER TYPE
1112	UNKNOWN FUNCTION
1113	DIVISION BY ZERO
1114	ORDER BY INDEX NOT ALLOWED
1115	DISTINCT NOT ALLOWED
1116	NESTED COLUMN FUNCTIONS NOT ALLOWED
1117	COLUMN FUNCTIONS NOT ALLOWED
1118	CAN NOT MIX COLUMN AND SCALAR OPERATIONS
1119	INVALID GROUP BY INDEX
1120	GROUP BY INDEX NOT ALLOWED
1121	GROUP BY NOT ALLOWED WITH SELECT ALL
1122	NOT A COLUMN EXPRESSION

1123	NOT A GROUPING COLUMN IN AGGREGATE ORDER BY
1124	MIXED LITERAL TYPES IN PREDICATE
1125	LIKE ESCAPE IS NOT ONE CHAR
1126	BAD LIKE ESCAPE CHAR
1127	UNKNOWN ESCAPE SEQUENCE IN LIKE
1128	COLUMNS FROM MORE THAN ONE QUERY IN COLUMN FUNCTION
1129	SCALAR EXPRESSION WITH GROUP BY
1130	SUBQUERY HAS MORE THAN ONE COLUMN
1131	SUBQUERY MUST HAVE ONE ROW
1132	INSERT VALUE COUNT DOES NOT MATCH COLUMN COUNT
1133	DUPLICATE COLUMN IN INSERT
1134	COLUMN DOES NOT ALLOW NULLS
1135	DUPLICATE COLUMN IN UPDATE
1136	TABLE ALREADY EXISTS
1137	DUPLICATE COLUMN IN CREATE TABLE
1138	DUPLICATE COLUMN IN COLUMN LIST
1139	MORE THAN ONE PRIMARY KEY NOT ALLOWED
1140	AMBIGUOUS FOREIGN KEY NAME
1141	COLUMN COUNT MISMATCH IN FOREIGN KEY
1142	COLUMN TYPE MISMATCH IN FOREIGN KEY
1143	FAILED TO FIND MATCHING PRIMARY COLUMN
1144	UPDATE AND DELETE CONSTRAINTS MUST BE THE SAME
1145	FOREIGN KEY DOES NOT EXIST
1146	INVALID LIMIT VALUE IN SELECT
1147	INVALID OFFSET VALUE IN SELECT
1148	PRIMARY KEY DOES NOT EXIST
1149	FAILED TO CREATE FOREIGN KEY
1150	FIELD IS NOT IN PRIMARY KEY
1151	FIELD IS NOT UPDATEABLE
1153	BAD DATA TYPE LENGTH
1154	EXPECTED EXECUTE IMMEDIATE COMMAND

Implementation

1203	FUNCTIONALITY IS NOT IMPLEMENTED
1204	FAILED TO CREATE NEW RECORD
1205	FAILED TO UPDATE FIELD
1206	FAILED TO DELETE RECORD
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

Parsing

1301	PARSING FAILED

Runtime language access

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	PARAMETER VALUE NOT SPECIFIED
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 (LANGUAGE RUNTIME)
1430	VARIABLE WAS NOT FOUND (LANGUAGE RUNTIME)

Date parsing

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

Date formatting

1551	FAILED

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
1607	NO VALID TOKENS

4D engine errors

1837	DB4D QUERY FAILED

Cacheable

2000	CACHEABLE NOT INITIALIZED
2001	VALUE ALREADY CACHED
2002	CACHED VALUE NOT FOUND

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

Command Index

4

A

B

C

D

E

F

H

I

L

M

N

O

P

Q

R

S

T

U

W

Y