

---

## **AIMMS Function Reference - Database Functions**

This file contains only one chapter of the book. For a free download of the complete book in pdf format, please visit [www.aimms.com](http://www.aimms.com)

# Database Functions

AIMMS supports the following database related functions:

- CloseDataSource
- CommitTransaction
- DirectSQL
- RollbackTransaction
- StartTransaction
- TestDataSource
- TestDatabaseTable
- TestDatabaseColumn
- GetDataSourceProperty
- SQLCreateConnectionString
- SQLNumberOfColumns
- SQLNumberOfTables
- SQLNumberOfViews
- SQLNumberOfDrivers
- SQLColumnData
- SQLTableName
- SQLViewName
- SQLDriverName

---

## CloseDataSource

With the procedure `CloseDataSource` you can temporarily close the connection to a data source. AIMMS automatically opens the connection to a data source if needed, and closes the connection when the project is exited.

```
CloseDataSource(  
    Datasource      ! (input) a string expression  
)
```

### Arguments:

*Datasource*  
A string containing the name of a data source.

### Remarks:

When `CloseDataSource` is called during a transaction that was explicitly started by calling `StartTransaction` the transaction is rolled back before actually closing the data source. `CurrentErrorMessage` contains a message telling it did so.

---

## CommitTransaction

By default, AIMMS places a transaction around any *single* WRITE statement to a database table. In this way, AIMMS makes sure that the complete WRITE statement can be rolled back in the event of a database error during the execution of that WRITE statement. With the procedure `CommitTransaction` you can commit all the changes to the database (through WRITE statements) made since the last call to `StartTransaction`.

`CommitTransaction`

### Arguments:

*None*

### Return value:

The procedure returns 1 if the transaction was committed successfully, or 0 otherwise.

### See also:

The procedures [StartTransaction](#), [RollbackTransaction](#).

---

## DirectSQL

With the procedure `DirectSQL` you can directly execute SQL statements within a data source.

```
DirectSQL(  
    Datasource,      ! (input) a string expression  
    SQLstatement    ! (input) a string expression  
)
```

### Arguments:

#### *Datasource*

A string containing the name of a data source.

#### *SQLstatement*

A string containing the SQL statement that must be executed within the data source.

### Return value:

The procedure returns 1 if the SQL statement is executed successfully, or 0 if the execution failed. In case of failure, the corresponding error message can be obtained through the predefined string parameter `CurrentErrorMessage`.

### Remarks:

- If the SQL statement also produces a result set, then this set is ignored by AIMMS.
- Note that the SQL dialect used by, for instance, Oracle, SQL Server and Microsoft Access may differ. If a call to `DirectSQL` fails because of such differences, you should inspect `CurrentErrorMessage` for further details.

### See also:

Calling stored procedures and executing SQL queries through AIMMS DATABASE PROCEDURES is discussed in [Section 25.5](#) of the Language Reference.

---

## RollbackTransaction

By default, AIMMS places a transaction around any *single* WRITE statement to a database table. In this way, AIMMS makes sure that the complete WRITE statement can be rolled back in the event of a database error during the execution of that WRITE statement. With the procedure RollbackTransaction you can roll-back (undo) all the changes to the database (through WRITE statements) made since the last call to StartTransaction.

RollbackTransaction

### Arguments:

*None*

### Return value:

The procedure returns 1 if the transaction was rolled back successfully, or 0 otherwise.

### See also:

The procedures [StartTransaction](#), [RollbackTransaction](#).

---

## StartTransaction

By default, AIMMS places a transaction around any *single* WRITE statement to a database table. In this way, AIMMS makes sure that the complete WRITE statement can be rolled back in the event of a database error during the execution of that WRITE statement. With the procedure StartTransaction you can manually initiate a database transaction which can contain multiple READ and WRITE statements.

```
StartTransaction(  
    IsolationLevel      ! (optional) an element expression  
)
```

### Arguments:

#### *IsolationLevel*

Element value into the set [AllIsolationLevels](#), indicating the isolation level at which the transaction has to take place. If omitted, defaults to 'ReadCommitted'.

### Return value:

The procedure returns 1 if the transaction was started successfully, or 0 otherwise.

### Remarks:

You cannot call StartTransaction recursively, i.e. you must call CommitTransaction or RollbackTransaction prior to the next call to StartTransaction.

### See also:

The procedures [CommitTransaction](#), [RollbackTransaction](#). The set [AllIsolationLevels](#).

---

## TestDataSource

With the procedure `TestDataSource` you can test for the presence of a data source on a host computer, before reading or writing to it. If you try to read or write to a non-existing data source, AIMMS will generate error messages which may be confusing for your end users.

```
TestDataSource(  
  Datasource,      ! (input) a string expression  
  interactive,    ! (input/optional) an integer, default 1  
  timeout         ! (input/optional) unit: seconds, default 30  
)
```

### Arguments:

#### *Datasource*

A string containing the name of a data source.

#### *interactive*

When non-zero: if additional (logon) information is required a window is popped up. When zero: if additional (logon) information is required, the procedure will return immediately with the value 0.

#### *timeout*

When the timeout is expired the procedure `TestDataSource` will return with the value 0.

### Return value:

The procedure returns 1 if the data source is present, or 0 otherwise.

### See also:

The procedures [TestDatabaseTable](#) and [TestDatabaseColumn](#).

---

## TestDatabaseTable

With the procedure `TestDatabaseTable` you can check whether a given table name exists in a specific data source.

```
TestDatabaseTable(  
    Datasource,      ! (input) a string expression  
    Tablename       ! (input) a string expression  
)
```

### Arguments:

*Datasource*

A string containing the name of a data source.

*Tablename*

A string containing the name of a table in *Datasource*.

### Return value:

The procedure returns 1 if the database table is present in the given data source, or 0 otherwise.

### Remarks:

The *Tablename* argument of the procedure `TestDatabaseTable` is case sensitive for OLE DB, and for ODBC it depends whether the ODBC driver is case sensitive.

### See also:

The procedures `TestDataSource` and `TestDatabaseColumn`.

---

## TestDatabaseColumn

With the procedure `TestDatabaseColumn` you can check whether a given column is present in a database table on a specific datasource.

```
TestDatabaseColumn(  
    Datasource,      ! (input) a string expression  
    TableName       ! (input) a string expression  
    ColumnName      ! (input) a string expression  
)
```

### Arguments:

*Datasource*

A string containing the name of a data source.

*TableName*

A string containing the name of a table in *Datasource*.

*ColumnName*

A string containing the name of a column in the *TableName*.

### Return value:

The procedure returns 1 if the column name is present in the given database table, or 0 otherwise.

### Remarks:

The *TableName* and *ColumnName* arguments of the procedure `TestDatabaseColumn` are case sensitive for OLE DB, and for ODBC it depends whether the ODBC driver is case sensitive.

### See also:

The procedures `TestDataSource` and `TestDatabaseTable`.

---

## GetDataSourceProperty

With the function `GetDataSourceProperty` you can retrieve some meta-data about a datasource. This is useful, when you don't know beforehand what kind of datasource will be linked with your AIMMS project. It allows you to provide datasource-specific SQL Queries in your project, which you can then call based upon what datasource is actually linked to your project. For example, you can determine with this function that the actual datasource is an Oracle database, and then execute some Oracle-specific SQL Queries.

```
GetDataSourceProperty(  
  Datasource,      ! (input) a string expression  
  Property,        ! (input) an element in the set  
                  AllDataSourceProperties  
)
```

### Arguments:

#### *Datasource*

A string containing the name of a data source.

#### *Property*

An element parameter in the set `AllDataSourceProperties`.

### Return value:

The function returns a string with the requested datasource property in it.

### Remarks:

The actual string which is returned depends on the datasource used and whether this datasource is an ODBC or an OLE DB datasource. For example, OLE DB doesn't support the properties `SQL_KEYWORDS` and `SQL_SERVERNAME`. In the former case, the resulting string contains the text "SQL\_KEYWORDS Property not supported for OLE DB interface.". As an example of the datasource dependency of the function: retrieving the property `SQL_DATA_SOURCE_NAME` may return "null" for a MySQL ODBC datasource, while it returns the actual name of your datasource when you retrieve it for an Oracle database. This means that you should experiment with the return values a bit, to make sure that you understand what values to expect for your specific datasource(s).

---

## SQLNumberOfColumns

With the function `SQLNumberOfColumns` you can determine the number of columns of a database table.

```
SQLNumberOfColumns(  
    Datasource,      ! (input) a string expression  
    TableName,      ! (input) a string expression  
    Owner           ! (input/optional) a string expression  
)
```

### Arguments:

#### *Datasource*

A string containing the name of a data source.

#### *TableName*

A string containing the name of the database table for which the number of columns must be determined.

#### *Owner*

A string containing the owner of the database table for which the number of columns must be determined. If the datasource doesn't support the owner concept, this argument is ignored.

### Return value:

The function returns the number of columns in the specified database table. If the database table doesn't exist, 0 is returned.

### See also:

The functions [SQLNumberOfViews](#), [SQLNumberOfTables](#) and [SQLColumnData](#).

---

## SQLNumberOfDrivers

With the function `SQLNumberOfDrivers` you can determine either the number of installed ODBC drivers, or the number of installed OLE DB providers on your system.

```
SQLNumberOfDrivers(  
    DatabaseInterface, ! (input) an element expression  
)
```

### Arguments:

#### *DatabaseInterface*

Element value into the set `AllDatabaseInterfaces`. Currently, this set contains the values 'ODBC' and 'OLE DB'.

### Return value:

The function returns either the number of installed ODBC drivers on your system (using 'ODBC' as argument), or the number of installed OLE DB providers on your system (using 'OLE DB' as argument). In case none are installed, the value 0 is returned. In case of an error, -1 is returned.

### Remarks:

This function should be used in combination with the function `SQLDriverName`, to determine all ODBC drivers or OLE DB providers installed on your system.

### See also:

The functions `SQLDriverName` and `SQLCreateConnectionString`.

---

## SQLNumberOfTables

With the function `SQLNumberOfTables` you can determine the number of tables in a datasource.

```
SQLNumberOfTables(  
    Datasource,      ! (input) a string expression  
    Owner           ! (input/optional) a string expression  
)
```

### Arguments:

*Datasource*

A string containing the name of a data source.

*owner*

A string containing the owner for which the number of tables must be determined. If the datasource doesn't support the owner concept, this argument is ignored.

### Return value:

The function returns the number of tables in the specified datasource. If there are no tables for the specified datasource and owner, 0 is returned. If an error occurs when determining the number of tables, -1 is returned and an error message is displayed in the error window.

### See also:

The functions [SQLNumberOfViews](#), [SQLNumberOfColumns](#) and [SQLTableName](#).

---

## SQLNumberOfViews

With the function `SQLNumberOfViews` you can determine the number of views in a datasource.

```
SQLNumberOfViews(  
    Datasource,      ! (input) a string expression  
    Owner            ! (input/optional) a string expression  
)
```

### Arguments:

#### *Datasource*

A string containing the name of a data source.

#### *Owner*

A string containing the owner for which the number of views must be determined. If the datasource doesn't support the owner concept, this argument is ignored.

### Return value:

The function returns the number of views in the specified datasource. If there are no views for the specified datasource and owner, 0 is returned. If an error occurs when determining the number of views, -1 is returned and an error message is displayed in the error window.

### See also:

The functions `SQLNumberOfTables`, `SQLNumberOfColumns` and `SQLViewName`.

---

## SQLColumnData

With the function `SQLColumnData` you can determine the characteristics of a certain column of a database table.

```
SQLColumnData(
  Datasource,      ! (input) a string expression
  TableName,       ! (input) a string expression
  ColumnNumber,   ! (input) an integer expression
  Owner,           ! (input/optional) a string expression
  ColumnCharacteristic ! (input/optional) an element in set AllData-
                    ColumnCharacteristics, with default
                    value 'Name'
)
```

### Arguments:

#### *Datasource*

A string containing the name of a data source.

#### *TableName*

A string containing the name of the database table of the column for which to retrieve a characteristic.

#### *ColumnNumber*

An integer containing the number of the column for which to retrieve a characteristic. The maximum value of this argument can be obtained by calling the function `SQLNumberOfColumns` prior to calling this function. The minimum value of this argument is 1.

#### *Owner*

A string containing the owner of the database table. If the datasource doesn't support the owner concept, this argument is ignored.

#### *ColumnCharacteristic*

An element in the set `AllDataColumnCharacteristics`, which contains all possible characteristics to obtain for a column.

### Return value:

The function returns the specified characteristic, as a string value. This means that also the numerical characteristics ('Width', 'NumberOfDecimals' and (possibly) 'DefaultValue') are returned as string values. So, if you want to use these results in their numeric form, please use the function `Val`.

### Remarks:

Typically, this function will be used in a construction like the following, to ensure that the right `ColumnNumber` argument is passed:

```
NumberOfColumns := SQLNumberOfColumns("MyDataSource", "MyTable");
```

```
ColCount := 1;
while ColCount <= NumberOfColumns do
  for IndexDataColumnCharacteristics do
    Characteristic := SQLColumnData(MyDataSource, "MyTable", ColCount, "",
                                     IndexDataColumnCharacteristics);
    ! Do something with the characteristic
  endfor;
  ColCount += 1;
endwhile;
```

**See also:**

The functions [SQLNumberOfColumns](#) and [Val](#).

---

## SQLDriverName

With the function `SQLDriverName` you can determine the name of a certain ODBC driver or OLE DB provider on your system. This function is designed to be used in conjunction with the `SQLNumberOfDrivers` function.

```
SQLDriverName(
  DatabaseInterface,  ! (input) an element expression
  DriverNo,          ! (input) an integer expression
)
```

### Arguments:

#### *DatabaseInterface*

Element value into the set `AllDatabaseInterfaces`. Currently, this set contains the values 'ODBC' and 'OLE DB'.

#### *DriverNo*

An integer containing the number of the ODBC driver or OLE DB provider for which you want to retrieve the name. To determine the maximum value of this argument, please use the function `SQLNumberOfDrivers` prior to calling this function. The minimum value of this argument is 1.

### Return value:

The function returns the name of the ODBC driver or OLE DB provider (specified by the `DatabaseInterface` argument), with the number as specified through the `DriverNo` argument. If you specify a number outside of the correct range, AIMMS will display an error message.

### Remarks:

Typically, this function can best be used in a construction like the following:

```
NumberOfDrivers := SQLNumberOfDrivers('ODBC');

while LoopCount <= NumberOfDrivers do
  DriverName := SQLDriverName('ODBC', LoopCount);
  ! Do something with the retrieved table name here...
endwhile;
```

The retrieved name of an ODBC driver or OLE DB provider, can be used as argument in the function `SQLCreateConnectionString`.

Please note that OLE DB providers have rather cryptical names, for example, 'SQLNCLI10' for the SQL Server Native Client 10.0. In order to make it more transparent for AIMMS users, the `SQLDriverName` function returns the human-readable provider name. You can use this human-readable provider

name as the `DriverName` argument of the `SQLCreateConnectionString` function. This function will automatically translate the human-readable name to the required 'real' provider name, which is required in an OLE DB connection string. Of course, you can also specify this 'real' provider name yourself, if you know it.

**See also:**

The functions `SQLNumberOfDrivers` and `SQLCreateConnectionString`.

---

## SQLTableName

With the function `SQLTableName` you can determine the name of a certain table in a datasource. This function is designed to be used in conjunction with the `SQLNumberOfTables` function.

```
SQLTableName(  
    Datasource,      ! (input) a string expression  
    TableNo,        ! (input) an integer expression  
    Owner           ! (input/optional) a string expression  
)
```

### Arguments:

#### *Datasource*

A string containing the name of a data source.

#### *TableNo*

An integer containing the number of the table for which you want to retrieve the name. To determine the maximum value of this argument, please use the function `SQLNumberOfTables` prior to calling this function. The minimum value of this argument is 1.

#### *Owner*

A string containing the owner of the table for which the name must be determined. If the datasource doesn't support the owner concept, this argument is ignored.

### Return value:

The function returns the name of the table, with the number as specified through the `TableNo` argument.

### Remarks:

Typically, this function can best be used in a construction like the following:

```
NumberOfTables := SQLNumberOfTables("MyDataSource");  
  
while LoopCount <= NumberOfTables do  
    TableName := SQLTableName("MyDataSource", LoopCount);  
    ! Do something with the retrieved table name here...  
endwhile;
```

### See also:

The functions [SQLNumberOfTables](#) and [SQLViewName](#).

---

## SQLViewName

With the function `SQLViewName` you can determine the name of a certain view in a datasource. This function is designed to be used in conjunction with the `SQLNumberOfViews` function.

```
SQLViewName(  
  Datasource,      ! (input) a string expression  
  TableNo,        ! (input) an integer expression  
  Owner           ! (input/optional) a string expression  
)
```

### Arguments:

#### *Datasource*

A string containing the name of a data source.

#### *ViewNo*

An integer containing the number of the view for which you want to retrieve the name. To determine the maximum value of this argument, please use the function `SQLNumberOfViews` prior to calling this function. The minimum value of this argument is 1.

#### *Owner*

A string containing the owner of the view for which the name must be determined. If the datasource doesn't support the owner concept, this argument is ignored.

### Return value:

The function returns the name of the view, with the number as specified through the `ViewNo` argument.

### Remarks:

Typically, this function can best be used in a construction like the following:

```
NumberOfViews := SQLNumberOfViews("MyDataSource");  
  
while LoopCount <= NumberOfViews do  
  ViewName := SQLViewName("MyDataSource", LoopCount);  
  ! Do something with the retrieved view name here...  
endwhile;
```

### See also:

The functions [SQLNumberOfViews](#) and [SQLTableName](#).

---

## SQLCreateConnectionString

The function `SQLCreateConnectionString` assists you in creating a *connection string*, which can be used to specify the Data source attribute of database tables, functions or procedures. Using a connection string to connect to a data source, makes it possible to keep your database passwords hidden.

```
SQLCreateConnectionString(
    DatabaseInterface,          ! (input) an element expression
    DriverName,                ! (input) a string expression
    [ServerName],              ! (optional) a string expression
    [DatabaseName],           ! (optional) a string expression
    [UserId],                  ! (optional) a string expression
    [Password],                ! (optional) a string expression
    [AdditionalConnectionParameters] ! (optional) a string expression
)
```

### Arguments:

#### *DatabaseInterface*

Element value into the set `AllDatabaseInterfaces`. Currently, this set contains the values 'ODBC' and 'OLE DB'.

#### *DriverName*

A string containing the name of the ODBC driver or OLE DB provider to which you want to connect using the resulting connection string. See the functions [SQLNumberOfDrivers](#) and [SQLDriverName](#) on how to obtain the driver/provider name.

#### *ServerName (optional)*

A string containing the name of the server on which the data source to connect to is hosted.

#### *DatabaseName (optional)*

A string containing the name of the database to which you want to connect.

#### *UserId (optional)*

A string containing the user id with which to login on the datasource.

#### *Password*

A string containing the password to use when logging in on the data-source.

#### *AdditionalConnectionParameters (optional)*

A string containing any additional connection parameters to be passed to the data source using the resulting connection string. These additional parameters should be specified in the form `KEYWORD=VALUE`, and these keyword/value pairs must be separated by semi-colons. Different drivers/providers accept different keywords. Please refer to the documentation of your ODBC driver or OLE DB provider for more information.

**Return value:**

The function returns a connection string, which can be used to connect to a data source on your system.

**Remarks:**

The returned connection string can be used as the data source attribute of database related identifiers in AIMMS. Also, it can be used in database related functions (e.g. `SQLDirect`) as the `Datasource` argument. Using connection strings instead of `.dsn` or `.udl` files, offers the advantage that you can keep your passwords hidden. The aforementioned files store the passwords un-encrypted, and are readable with any ASCII editor. The only way to hide passwords from the users was to explicitly not store the passwords in these files and let AIMMS automatically prompt the users upon connecting to the data source. Now, you can provide the necessary passwords in dynamically constructed connection strings, without the need for prompting your users.

When using AIMMS as a component, you didn't really have the possibility to leave out passwords from `.dsn` or `.udl` files (since typically you don't want dialogs to appear in such scenario's). Using connection strings, this is now perfectly possible.

Furthermore, by using connection strings, you are also more flexible in supporting more than one data source. Instead of offering a separate `.dsn` or `.udl` file for each possible data source, you can now dynamically construct connection strings and use these for setting up all your database connections.

Please note that OLE DB providers have rather cryptical names, for example, 'SQLNCLI10' for the SQL Server Native Client 10.0. In order to make it more transparent for AIMMS users, the `SQLDriverName` function returns the human-readable provider name. You can use this human-readable provider name as the `DriverName` argument of the `SQLCreateConnectionString` function. This function will automatically translate the human-readable name to the required 'real' provider name, which is required in an OLE DB connection string. Of course, you can also specify this 'real' provider name yourself, if you know it.

**See also:**

The functions `SQLNumberOfDrivers` and `SQLDriverName`.