AGRESSO SQL Manual

Version 2.6

Compiled from various sources

by Mike Schofield
Technical Consultant

Table of Contents

Introduction

AGRESSO SQL (or ASQL) is a proprietary dialect of Structured Query Language (SQL) that was developed to allow AGRESSO Business World (ABW) server processes to communicate with and manipulate the ABW database using a language that is not specific to any particular database vendor.

ASQL consists of three elements:

1. Commands
2. Functions
3. Macros

ASQL is used in all of the following ways:

- In the asql.exe program, either from the command line, via ASQLGUIUK.exe or via an AG16 query.
- In an AGRESSO Report Writer (ARW) report
- In an AGRESSO Excelerator report that is run on the server and does not use “DBSELECT”
- In programs run as ABW server processes

A Warning to Developers

If you are writing a server process (i.e. a program not an AG16 query) then it is important to know that the ASQL provided by the server API is slightly different to that described in this manual. Notable differences are:

- There is no 'CREATE TABLE AS *select_statement*', you must use a special API method instead
- There is no INTO clause on the SELECT statement
- There is no SEQUENCE statement
- To use the equivalent of a FROM clause on the UPDATE statement you must use a special API method.
- The ‘Comments’, ‘DATABASE keyword’, ‘Database Block’, ‘DEFINE’, ‘Error handling’, ‘IF EXISTS Block’ and ‘Transaction Handling’ ASQL commands are not available

Syntax Conventions Used In This Manual

Words in lower case *italics are*

variable arguments these are generally:

- Column names
- Literals (strings in apostrophes or numbers)
- Formulas (operands may be columns and/or literals)

Words in **bold** (usually also **UPPER CASE**) are

Keywords

[Value1]

Value1 is optional

{Value1|Value2}

Either ‘Value1’ or ‘Value2’ may be used

…

Repeat as required

Commands

ASQL commands fall into the following categories:

- General
- Data Definition Language
- Data Manipulation Language

General

These commands document and control the overall execution of an ASQL script.

- Comments
- DATABASE keyword
- Database Block
- DEFINE (a variable)
- Error Handling
- IF EXISTS Block
- Transaction Handling

Data Definition Language

Data Definition Language (or DDL) statements are used to define and maintain database objects. In ASQL the DDL commands are the CREATE and DROP statements,

DDL statements should **NOT** be used inside transactions. If a specific database server allows this, make sure the transaction is inside a database block

- CREATE TABLE
- CREATE INDEX
- CREATE VIEW
- DROP

Data Manipulation Language

Data Manipulation Language (or DML) statements are used to create, maintain and retrieve data in tables and views. In ASQL the following statements are supported:

- COPY
- DELETE
- INSERT
- SELECT
- SEQUENCE
- UPDATE

Command Syntax

Comments

/*
        comment_text
*/

comment_text	Text that will be ignored by ASQL.

Comments can be added anywhere in an .asq file except inside strings: even inside statements.

For example:
In the following the amount column will not be updated:

```
update mydescription m
    from description d
    set m.description = d.description
/*
        ,m.amount = fmulm(m.amount, abs(d.mult)) * 1.1
*/
    where m.account = d.account
        and m.client = d.client
/
```

COPY Statement

This cannot be called inside a transaction. It is used to transfer data to and from an ASCII file.

The copy statement comes in three different variations for dealing with different types of file.

• Standard copy from file into table	All the records in the file must have the same number of fixed or variable length fields, the data type of each field must be the same on all records and each field must have a value on all records.
• Standard copy from table into file	All the records in the file will have the same number of fixed or variable length fields, the data type of each field will be the same on all records and each field will have a value on all records.
• Special copy from file into table	The records in the file may have different formats but all formats must have a record type in a fixed position. All the records of the same type must have the same number of fixed length fields, the data type of each field must be the same on all records of that type and each field must have a value on all records of that type.

Standard copy from file into table

COPY IN [{**IMPORT**|**EXPORT**}] **FILE** = *file_name*,
    COLSEP = {**F**|**T**|**@**|**£**|**$**|**%**|**&**|**.**|**;**}, **TABLE** = *tab_name*,
    col_name[= [**S**]*n*][, *col_name*[= [**S**]*n*]...]
/

IMPORT	The IMPORT keyword tells the COPY statement to look for the file in the directory pointed to by the AGRESSO_IMPORT environment variable.
EXPORT	The EXPORT keyword tells the COPY statement to look for the file in the directory pointed to by the AGRESSO_EXPORT environment variable.
file_name	The name of the file you want to import. If no path is given it will look either in the current folder, the import folder or the export folder (see the 'IMPORT' and 'EXPORT' parameters). If the file name and/or path contain spaces then the whole parameter must be contained in single quotes.
COLSEP	Setting this parameter to "F" means that the file has fixed width records; the value of each field is in the same position on each record; each column name must be given the width of its corresponding field using the "=*n*" syntax. The other 'COLSEP' values specify that the fields (and therefore the records) are variable in length and separated by a delimiting character. Characters that can be used as delimiters are: T a Tab character (ASCII 9) @ an "at" symbol £ a Sterling sign $ a Dollar sign % a percent sign & an ampersand . a full-stop ; a semi-colon Be aware that strings are not automatically un-quoted (see Quoted Strings).
tab_name	An identifier used to specify the name of the table you want the data to be copied into.
col_name	An identifier used to specify a column name on the receiving table. These must be listed in the order in which they appear on the file.
S	When the 'S' switch is used on a column that column doesn't have a corresponding field on the file, it will be assigned a unique sequence number starting at *n*. The 'S' switch can only be used if the 'COLSEP' value is "F" and then only on one column: which must be defined as an integer.
n	This parameter is used in conjunction with 'COLSEP' value "F". When the file has a fixed format the value for every field takes up the same number of characters on every record. For a normal

field *n* defines the field's length; on the 'S' field it specifies the starting sequence number. Remember that an integer value can have up to 18 digits, and a float or money value up to 25 digits and they are both stored as character strings in the file.

Here is an example:

```
copy in file = :myfile,
    colsep = F,
    table = hlptab,
    account=8,
    amount=25,
    sequence_no=s1
/
```

Standard copy from table into file

COPY TO [{**IMPORT**|**EXPORT**}] **FILE** = *file_name*,
    COLSEP = {**F**|**T**|**@**|**£**|**$**|**%**|**&**|**.**|**;**},
    select_statement
/

IMPORT	The IMPORT keyword tells the copy statement to create the file in the directory pointed to by the AGRESSO_IMPORT environment variable.
EXPORT	The EXPORT keyword tells the copy statement to create the file in the directory pointed to by the AGRESSO_EXPORT environment variable.
file_name	The name of the file you want to create. If no path is given it will create it either in the current folder, the import folder or the export folder (see the 'IMPORT' and 'EXPORT' parameters). If the file name and/or path contain spaces then the whole parameter must be contained in single quotes.
COLSEP	Setting this parameter to "F" means that the file will have fixed width records; the value of each field will be in the same position on each record; each column name in the *select_statement* must be given the width of its corresponding field using the "=*n*" syntax. The other 'COLSEP' values specify that the fields (and therefore the records) will be variable in length and separated by a delimiting character. Characters that can be used as delimiters are:

T	a Tab character (ASCII 9)
@	an "at" symbol
£	a Sterling sign
$	a Dollar sign
%	a percent sign
&	an ampersand
.	a full-stop
;	a semi-colon

Be aware that strings will not be automatically quoted (see Quoted Strings).

select_statement	Is a normal ASQL SELECT statement except that no INTO clause can be specified and with ‘COLSEP’ value “F” each column must be given the length of its corresponding field on the file using the “ =*n*" syntax. Remember that an integer value can have up to 18 digits and a float or money value up to 25 digits. The columns must be listed in the order in which you wish them to appear on the file.

Here is an example:

```
copy to import file = :myfile,
    colsep = F,
    select  apar_id=8, apar_name=50
    from asuheader
    where client = '$client'
        And apar_gr_id = 'MS'
/
```

Special copy from file into table

COPY FROM [**IMPORT**] = ‘*file_name*’,
 TABLE = ‘*tab_name*’,
 KEY = *key_from* - *key_to,*
 ’*key_value*’=*col_name type*(*start-end*)[, *col_name type*(*start-end*)...]
 [’*key*_value’= *col_name type*(*start-end*) [, *col_name type*(*start-end*)...] ...]
/

IMPORT	The IMPORT keyword tells the copy statement to look for the file in the directory pointed to by the AGRESSO_IMPORT environment variable. The ‘EXPORT’ parameter is not supported.
file_name	The name of the file you want to import. If no path is given it will look either in the current folder or the import folder (see the IMPORT parameter), it must be enclosed in single quotes.
tab_name	An identifier used to specify the name of the table you want the data to be copied into, it must be enclosed in single quotes.
KEY	The format of each record is identified by a key. This key is the only field that has to be in the same position throughout the file.
key_from	Start position of the key field. The first character on a record is position 1.
key_to	End position of the key field.
'*key_value*'	The key value that identifies a particular record type (set of columns). The value must be enclosed in single quotes.
col_name	An identifier used to specify the table column in which to store the field.
type	Specifies the type of the field in the record. The *type* determines the conversion of the field (CHAR) to the column's data type. Legal type values are:

Type	Description
c	Character

Type	Description
i	Integer
f*n*	Float with *n* decimals, no decimal separator.
f.*n*	Float with *n* decimals and a decimal separator.
m*n*	Money with *n* decimals, no decimal separator.
m.*n*	Money with *n* decimals and a decimal separator.
d1	Date (mmdd)
d2	Date (yymmdd)
d3	Date (yyyymmdd)
d	Date (yyyymmdd hh:mi:ss)
p1	Period (yypp)
p2	Period (yyyypp)

start-end	The position of the field in the record. The first position on a record is 1. Columns do not have to be listed in the order they appear on the record. **N.B.** "mycol(10-15) means the mycol field is 6 characters long not 5!

Notice that there is no comma between each new key description.

Here is an example:

```
copy from import = 'Tbrk0606.dat',
    table = 'hlptbl',
    key = 1-8,
    '940SWI01' = foreign_acc c(42-76),
        statement_ob c(110-134)
    '940SWI02' = foreign_acc c(42-76),
        trans_date d2(91-96),
        voucher_date d1(97-100),
        dc_cflag c(101-101),
        cur_amount m2(104-118),
        description c(141-156),
        ext_inv_ref c(157-162)
    '940SWI05' = foreign_acc c(42-76),
        statement_cb c(91-115)
/
```

CREATE INDEX statement

CREATE [**PRIMARY**] [**UNIQUE**] **INDEX** *ind_name*
    ON *tab_name* **(***col_name*[, *col_name*...]**)**
    [**WITH STRUCT** = *struct*] [**TABSPACE** = *location*]
/

PRIMARY	This is only used when running against an Ingres database and turns the ’create index’ statement into a ’modify’ statement.
UNIQUE	Adds a unique constraint to the table. Two rows where all the columns listed in the index have the same value are not allowed.
ind_name	An identifier specifying the name of the index.
tab_name	An identifier specifying the name of the table.

col_name	An identifier specifying a column name. You should try and limit the number of columns in the index. AGRESSO does not allow more than 10.
struct	The ‘WITH STRUCT’ clause is only used when running against an Ingres database. It specifies the storage structure of the index.
location	The ‘TABSPACE’ clause is used when you want to put the index in a location (tablespace, segment, dbspace) other than the default location.

CREATE TABLE statement

CREATE TABLE *tab_name*
{**AS** *select_statement* |
(*col_name col_type*[**(***length***)**][, *col_name col_type*[**(***length***)**]...]**)**}
[**TABSPACE** = *location*]
/

tab_name	An identifier specifying the name of the table you are creating. **WARNING** *tab_name* must not be more than 40 characters long otherwise you will not be able to SELECT from the table
select_statement	A normal ASQL SELECT statement except that you cannot use the ‘INTO’ or ‘ORDER BY’ clauses. Each column must have a different name.
col_name	An identifier specifying the name of a column.
col_type[**(***length***)**]	An identifier specifying the type [and length] of the column

Legal values of *col_type*[**(***length***)**] are:

Type	length	Description
char	1 – 255	Fixed length character string. Length normally <= 16.
vchar	1 – 1800	Variable length character string.
bool	N/A	Normally used as a flag. Legal values are 0 or 1
int1	8 bit	Small integers. Legal values from -256 to 255
int2	16 bit	Medium integers. Legal values from –32,768 to 32,767
int	32 bit	Normal integers. Legal values from -2,147,483,648 to 2,147,483,647
int8	64 bit	Big integers. Legal values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 Only available on ABW 5.5 or better
float	N/A	A float value. Legal values depend on the database server.
money	N/A	A money value. Legal values depend on the database server. Don’t expect a precision of more than 16.

Type	length	Description
date	N/A	A date value.
raw	N/A	Used to store binary large objects (BLOBS).

location	The TABSPACE clause is used when you want to put the table in a location (tablespace, segment, dbspace) other than the default location. It is generally best to avoid using this clause if possible

The maximum number of columns allowed is 250, but you should try to limit the number of columns.

Some database servers restrict the total number of bytes for a row to be less than 2KB.

There should never be more than one raw column in a table, or more than one CHAR/VCHAR column longer than 255 bytes.

CREATE VIEW statement

CREATE VIEW *view_name* [(*col_name*[, *col_name*...])]
        AS *select_statement*
/

view_name	An identifier specifying the view name.
col_name	An identifier specifying a column name. The view should not have more than 250 columns.
select_statement	A normal ASQL SELECT statement except that you cannot use the ‘INTO’, ‘ORDER BY’ or ‘UNION’ clauses. Each column must have a different name. If you have specified a *col_name* list then the ‘SELECT’ clause must have the same number of columns in it.

DATABASE Keyword

DATABASE *sql_statement*
/

sql_statement	*Any valid “native” syntax SQL statement*

Sometimes when you are writing an ASQL script you want to make use of a feature of the underlying database platform that is not supported by ASQL. You can access the native SQL syntax of the database by prefixing a native SQL statement with the DATABASE keyword.

You can also use Database Blocks (see below).

Here is an example:
You are running against an Oracle database and you want to use an Oracle SEQUENCE; the following statement won't work because the ASQL CREATE statement doesn't support the SEQUENCE key word:

```
create sequence myseq
/
```

But

```
database create sequence myseq
/
```

will work.

Database Blocks

BEGIN *platform*[, *platform*...]
/
    sql_statement
    /
    …
END
/

platform	The database platform(s) that this block applies to. Legal values for platform are: • anywhere • informix • ingres • oracle • sqlserver[{6|7}] (see below) • sybase. If you use keyword 'sqlserver', this will be used regardless of version. Since system tables changed from version 6.x to 7, you might need to make different scripts for each version: • Keyword 'sqlserver6', will be used if the version on your MS Sql Server is 6.5 or less. • Keyword 'sqlserver7' will be used for version 7 and upward. (MS Sql Server 2000 has version number 8.x.)
sql_statement	An SQL statement in the syntax of the specified platform(s)

While the 'DATABASE' keyword is very useful if you are writing a script that cannot be completely written in ASQL if it is only required to work on a single database platform, it is of no use if this script has to work on multiple database platforms and the syntax of the non-ASQL code is different on these platforms. This is where Database Blocks come to your assistance. The statements inside a Database Block are only executed if the script is run on one of the platforms listed on its BEGIN statement, for example statements in a "BEGIN oracle" block are only executed if the script is run on an Oracle database.

You cannot use the DEFINE statement inside a Database Block but you can use variables defined outside the block.

You cannot nest Database Blocks, not even in IF EXISTS Blocks

Here is an example:
Say you have a script that selects the location of a table given the table name. Each database server has different system tables.

```
define tab_location ident(12)
/
```

```
begin oracle
/
    SELECT tablespace_name
        INTO :tab_location
        FROM all_tables
        WHERE owner = 'agresso'
            AND table_name = 'mytable'
    /
end
/

begin sybase
/
    SELECT s.name
        INTO :tab_location
        FROM sysobjects t, sysindexes i, syssegments s
        WHERE t.type = 'U'
            AND t.id = i.id
            AND i.indid = 0
            AND i.segment = s.segment
            AND t.name = 'mytable'
    /
end
/

begin sqlserver
/
    SELECT s.name
        INTO :tab_location
        FROM sysobjects t, sysindexes i, syssegments s
        WHERE t.type = 'U'
            AND t.id = i.id
            AND i.indid IN (0,1)
            AND i.segment = s.segment
            AND t.name = 'mytable'
    /
end
/

begin ingres
/
    SELECT DISTINCT location_name
        INTO :tab_location
        FROM iitables
        WHERE system_use = 'U'
            AND table_type='T'
            AND table_name = 'mytable'
            AND table_owner='agresso'
    /
end
/
```

DEFINE Statement

DEFINE *name type*[**(***length***)**]

/

name	The name of the variable. This must not be an ASQL keyword. The name must not be quoted.
type[**(***length***)**]	The data type of this variable. Legal values for *type* and *length* are:

Type	Length
CHAR	1 – 255
IDENT	1 – 40
INT	N/A
FLOAT	N/A
DATE	N/A

Only variables of *type* CHAR have their contents enclosed in single quotes automatically when they are used. In particular IDENT variables are strings that will not be quoted and so can be used to hold the names of database objects, e.g. tables, indexes and columns.

Variables of *type* DATE can be used directly as a date in a statement; you do not have to convert them. If you want to use a date variable as a CHAR (e.g. when comparing with a string constant), you have to convert it to CHAR.

In early AGRESSO releases, when a variable was defined, it would stay defined throughout the program. This means that you could not define the same variable in more than one file if those files were going to be run together using the **asql.exe** program's **-h** parameter. This is no longer true.

Example:

```
define description char (60)
/
define mytable ident (16)
/
define last_update date
/
```

DELETE statement

DELETE FROM *tab_name*
    [**WHERE** [**NOT**] *condition* [{**AND** | **OR**} [**NOT**] *condition*…]]
/

tab_name	An identifier specifying the table name.
condition	A comparison between an expression, column or value and another expression, column or value; expressions contain arithmetic and/or function calls. The list as a whole is used to limit the number of rows that are deleted and round brackets (…) can be used to group conditions when a mixture of **AND**s and **OR**s are used.

The ASQL DELETE statement doesn't allow more than one table in its FROM clause. If you want to delete from one table using values from another table then try one of the following techniques:

- Use a sub-query in the WHERE clause.
- Use a preceding UPDATE statement to set one of the columns to some "delete me" value and use this a the condition in the WHERE clause.

DROP statement

DROP {**TABLE** | **INDEX** | **VIEW**} *name*
/

name	An identifier specifying the name of the table, index or view to delete from the database.

Error handling

ON ERROR {**EXIT**|**CONTINUE**|**CONT**}
/

EXIT	When an error occurs, the program will stop processing the current file. If a transaction is active it will be rolled back.
CONTINUE|CONT	This is the default behavior. When an error occurs continue with the next statement.

When a statement doesn’t execute because of an error, the default behavior is to report the error message and then continue with the next statement. This behavior can be modified with the ON ERROR statement. Although ON ERROR is a statement it doesn't execute as it is encountered in the file it just tells ASQL what to do if an error is encountered in a subsequent statement (see also the **asql.exe** program’s **-x** parameter).

IF EXISTS Blocks

IF [**NOT**] **EXISTS** *tab_name*[*col_name*]
/
    Conditional statements
    /
END IF
/

NOT	If this is present then if *tab_name*[.*col_name*] does not exist enter the block otherwise enter the block if *tab_name*[.*col_name*] does exist.
tab_name	The table whose existence is being tested.
col_name	The column on *tab_name* whose existence is being tested, Note that there is no full-stop before *col_name*.

IF EXISTS blocks are typically used to create tables if they don’t exist, drop them if they do, or alter them if a column is missing or defined wrongly.

IF EXISTS blocks cannot be nested, not even in Database Blocks.

Here is an example:

```
if not exists testtab
```

```
/
    create table testtab (col1 int, col2 char(10))
    /
end if
/

if exists testtab col1
/
    database alter table testtab drop col2
    /
end if
/
```

INSERT statement

INSERT INTO *tab_name* [**(***col_name*[, *col_name*...]**)**]
{**VALUES(***value*[, *value*...]**)** | *select_statement*}
/

tab_name	An identifier specifying the table name.
col_name	An identifier specifying the name of a column.
value	A fixed value to insert into a column.
select_statement	A SELECT statement that provides values to insert into the columns.

Although it is allowed, it is not recommended that you use the INSERT statement without listing the columns that you want to insert into. It is also recommended that you list all the columns in the table, supplying default values to all the columns that you don't have explicit values for.

If a *col_name* list was supplied then the *value* list or *select_statement* must provide the same number of values and they must have compatible data types.

If no *col_name* list was supplied then you have to make sure that the *value* list or *select_statement* provides values for all the columns in the table, and that the values are in the correct order (tricky!).

Here is an example:

```
insert into mytable (col1, col2, col3)
    select col1, col2, 0 from myothertable
/
```

SELECT statement

SELECT {*expression*|*col_name*|*value*}[**AS** *col_alias*]
[, {*expression*|*col_name*|*value*}[**AS** *col_alias*]…]
[**INTO** *variable*[, *variable*…]]
[**FROM** *tab_name* [*alias*][, *tab_name* [*alias*]...]]
[**WHERE** [**NOT**] *condition* [{**AND** | **OR**} [**NOT**] *condition*…]]
[**GROUP BY** *col_name*[, *col_name*...]]
[**HAVING** [**NOT**] *condition* [{**AND** | **OR**} [**NOT**] *condition*…]]
[**UNION** *select_statement*[**UNION** *select_statement*…]]
[**ORDER BY** *col_name*[, *col_name*...]]

/

expression	An expression is a simple or complex formula involving functions, operators and column names from several tables
col_name	An identifier used to specify a column name
value	A fixed number or string, enclose strings in single quotes
col_alias	Every column may be identified by an alias which will be the name of the column in the result set
variable	The name of a previously defined variable, including the ":" prefix. There must not be more items in this list than there are in the SELECT clause's *col_name* list, but there can be less. On ABW 5.5, or better, there can only be **one** variable in the list
tab_name	If you need values from other tables, list these tables in the FROM clause, don't forget the aliases. If all the items in the SELECT clause are constants then a FROM clause is not required.
alias	Every table may be identified by an alias. This is usually one letter (but can be more) used to identify which table a column belongs to.
condition	A comparison between an expression, column or value and another expression, column or value. The list as a whole is used to limit the number of rows that are retrieved and round brackets (…) can be used to group conditions when a mixture of **AND**s and **OR**s are used.

The main differences between an ASQL select and a native database one are:

- The list of functions available and their names
- Only inner joins are supported
- Join criteria are in the WHERE clause not the FROM clause
  e.g. "SELECT … FROM acrclient c, agltransact g WHERE a.client = g.client"

SEQUENCE statement

SEQUENCE ON *tab_name*
    (*seq_col*, *seed***)**
    [**GROUP BY** *col_name*[, *col_name*...]]
/

tab_name	An identifier specifying the table name.
seq_col	An identifier specifying the name of the column where you want the sequence number stored.
seed	An integer holding the start number for the sequence.
col_name	An identifier specifying the name of a column.

If there is no GROUP BY clause, the sequence number is increased by one for every row in the table, starting with the *seed* value. If you have columns listed in the GROUP BY clause, the rows are grouped according to the values in the columns. The sequence number is then reset to the *seed* value for every group.

Developers please note that the sequence statement is implemented in asql.exe ASQL but not in the server API.

Here is an example:

```
sequence on mytable (sequence_no, 0)
    group by auto_id, auto_type
/
```

Transaction Handling

BEGIN TRANSACTION
/

Starts a transaction.

ABORT TRANSACTION
/

Rolls back the transaction. You cannot execute **ABORT TRANSACTION** without having executed **BEGIN TRANSACTION**.

END TRANSACTION
/

Commits the transaction. You cannot execute **END TRANSACTION** without having executed **BEGIN TRANSACTION**.

You can group DELETE, INSERT and UPDATE statements into a “transaction” using the BEGIN TRANSACTION and END TRANSACTION statements.

Statements inside a transaction are only committed (written) to the database when the END TRANSACTION statement is executed.

If an error occurs when ON ERROR EXIT is active, or an ABORT TRANSACTION is executed, before the END TRANSACTION is reached then all the statements executed after the BEGIN TRANSACTION are “rolled back” (undone).

Until the transaction is committed other users of the database cannot see the changes you are making.

Make sure you don’t have any Data Definition Language statements (CREATE and DROP) inside a transaction because these force an implicit commit when they are executed.

Try to keep transactions short; while you are inside a transaction the tables that you are manipulating are locked so that other users can’t change them. This can lead to low concurrency and can also lead to deadlocks.

UPDATE statement

UPDATE *table alias*
[**FROM** *tab_name alias*[, (*tab_name alias*...]]
SET *alias.col_name* = *expression*[,*alias.col_name* = *expression*...]
[**WHERE** [**NOT**] *condition* [{**AND** | **OR**} [**NOT**] *condition*…]]

table	An identifier specifying the name of the table that you want to update.
alias	You must identify every table by an alias. This is usually one letter (but can be more) used to identify which table a column belongs to. Every column name referred to in the update statement (both in the WHERE clause and SET clause) has to be prefixed with a table alias.
tab_name	If you need values from other tables, list these tables in the FROM clause, don't forget their aliases. The FROM clause is not required but if present the join criteria must be put in the WHERE clause.
col_name	An identifier used to specify a column name.
expression	An expression can be anything from a constant value to a complex formula involving functions, operators and column names from several tables.
condition	A comparison between an expression, column or value and another expression, column or value; expressions contain arithmetic and/or function calls. The list as a whole is used to limit the number of rows that are updated and round brackets (…) can be used to group conditions when a mixture of **AND**s and **OR**s are used. Remember to identify every column name with its table alias.

When running against Oracle, make sure you join on all key columns; otherwise you might get an error along the lines of 'subquery returns more than one row'.

Similarly the FROM clause can be problematic on Oracle as it will frequently give the same error, if you get the error and you are only updating one column, try using a sub-query in the SET clause instead.

Here is an example:

```
update mydescription m
    from description d
    set m.description = d.description,
        m.amount = fmulm(m.amount,abs(d.mult))*1.1
    where m.account = d.account
        and m.client = d.client
/
```

ASQL Functions

There are two sorts of functions available in ASQL:

- Aggregate functions
- Scalar functions

Aggregate functions

Aggregate functions operate on a collection of values but return a single summarizing value. Aggregate functions can be used wherever an expression is valid in:

- The SELECT clause
- The HAVING clause

The ASQL aggregate functions are:

- AVG
- COUNT
- MAX
- MIN
- SUM

Scalar functions

Scalar functions operate on one or two single values and then return a single value. Scalar functions can be used wherever an expression is valid.
The ASQL scalar functions are:

- Conversion functions
    - CONVERT
    - CTS2DAY
    - DATE2ISO
    - DATE2STR
    - DATETIME2STR
    - INT2STR
    - ISO2DATE
    - STR2DATE
    - TO_CHAR
    - TO_DATE
    - TO_INT
    - TO_FLOAT
    - TO_MONEY
- Date functions
    - DATEDIFF
    - DATEPART
    - DATETRUNC
    - DAYADD
    - GETDATE
    - MONTHADD
    - TS2DAY

- Miscellaneous functions
    - IFNULL
- Number functions
    - ABS
    - FDIV
    - FDIVM
    - FMUL
    - FMULM
    - MOD
    - ROUND
- String functions
    - CONCAT
    - LEFT
    - LENGTH
    - LOWER
    - LPAD
    - LSHIFT
    - RIGHT
    - RPAD
    - RTRIM
    - SPACE
    - SQUEEZE
    - UPPER

Function Syntax

Function	Arguments	Description
ABS	*expr*	Returns the absolute (unsigned) value of numeric expression *expr*
AVG	[{**ALL**|**DISTINCT}**] *expr*	Returns the average of [distinct] values in numeric column *expr*
CONCAT	*expr1*, *expr2*	Returns string *expr1* concatenated with string *expr2* e.g. CONCAT(‘ab’, ‘cde’) returns ‘abcde’
CONVERT	*from*, *to*, *expr*	Converts *expr* from data type *from* to data type *to* Parameters *from* and *to* can each have one of the following values: **CHAR**, **INT**, **FLOAT**, **MONEY** or **DATE**. (In Oracle, **MONEY**, **INT** and **FLOAT** are identical). The following conversions gives unpredictable results: • **DATE ↔ INT** • **DATE ↔ FLOAT** • **DATE ↔ MONEY**

Function	Arguments	Description
COUNT	[{**ALL**|**DISTINCT**}] *expr*	Returns the number of [distinct] non-null values in column *expr*
COUNT	*	Returns the number of selected rows
CTS2DAY	*expr*	Converts string *expr* to a date with no time
DATE2ISO	*expr*	Returns date *expr* converted to a string in the form 'yymmdd'
DATE2STR	*expr*	Returns date *expr* converted to a string in the form 'yyyymmdd'
DATEDIFF	*expr1*, *expr2*	Returns a floating point number (*expr1* – *expr2*). The integer portion is number of days.
DATEPART	*unit*, *expr*	Returns an integer containing the specified part of date *expr*. *unit* can have one of the following values: **DAY**, **HOUR**, **MIN**, **MONTH**, **QUARTER**, **SEC**, **WEEK** or **YEAR**
DATETIME2STR	*expr*	Returns date *expr* converted to a string in the form 'yyyymmdd hh:mn:ss'
DATETRUNC	*unit*, *expr*	Returns a date value that represents date *expr* truncated to the interval expressed in *unit*. *unit* can have one of the following values: **DAY**, **MONTH** or **YEAR**
DAYADD	*n*, *expr*	Adds *n* days to date *expr* and returns the new date. Use a negative number in *n* to subtract
FDIV	*expr1*, *expr2*	Converts *expr1* and *expr2* to FLOAT and returns *expr1* divided by *expr2* as a FLOAT
FDIVM	*expr1*, *expr2*	Same as FDIV except that the type of the returned value is MONEY
FMUL	*expr1*, *expr2*	Converts *expr1* and *expr2* to FLOAT and returns *expr1* multiplied by *expr2* as a FLOAT
FMULM	*expr1*, *expr2*	Same as FMUL except that the type of the return value is MONEY
GETDATE		Returns the current date See also the TODAY macro
IFNULL	*expr1*, *expr2*	If *expr1* is null *expr2* is returned otherwise *expr1*.
INT2STR	*expr*	Converts an integer *expr* to a string
ISO2DATE	*expr*	Converts string *expr* to a date. *expr* must be in the form 'yymmdd' or 'yyyymmdd'
LEFT	*expr*, *n*	Returns the leftmost *n* characters of the string *expr*

Function	Arguments	Description
LENGTH	*expr*	Sybase: Returns the length of *expr* excluding trailing blanks Oracle: Returns the length of *expr* including trailing blanks
LOWER	*expr*	Returns the string *expr* in lowercase
LPAD	*c*, *length*, *expr*	Pads the string *expr* to *length* by prefixing with *c* characters
LSHIFT	*expr, n*	Shifts value of the string *expr n* characters to the left. This function in combination with LEFT or RIGHT is used to implement the missing SUBSTRING function. E.g. LEFT(LSHIFT('ABCD', 2), 1) returns 'C'.
MAX	*expr*	Returns the highest value in column *expr*
MIN	*expr*	Returns the lowest value in column *expr*
MOD	*expr1*, *expr2*	Returns the integer result of *expr1* modulo *expr2*. E.g. MOD(9, 4) returns 1.
MONTHADD	*n, expr*	Adds *n* months to date *expr* and returns the new date. Use a negative number in *n* to subtract
RIGHT	*expr, n*	Returns the rightmost *n* characters of string *expr*
ROUND	*expr,n*	Returns a FLOAT whose value is *expr* rounded to the precision indicated by *n*: • If *n* is positive it indicates the number of significant digits to the right of the decimal point. • If *n* is negative it indicates the number of significant digits to the left of the decimal point.
RPAD	*c*, *length*, *expr*	Pads the string *expr* to *length* by appending *c* characters Warning: "RPAD(' ',*length*,*expr*)" will not work, since all result strings are stripped of trailing blanks
RTRIM	*expr*	Returns the string *expr* without trailing blanks
SPACE	*n*	Returns a string containing *n* spaces (max 1800)
SQRT	*expr*	Returns square root of *expr*
SQUEEZE	*expr*	Returns the string *expr* removing all leading and trailing blanks

Function	Arguments	Description
STR2DATE	*expr*	Converts string *expr* to a date. *expr* must be in the form ‘yyyymmdd’ or ‘yymmdd’
SUM	[{**ALL**|**DISTINCT}**] *expr*	Returns the total of [distinct] values in numeric column *expr*
TO_CHAR	*expr*	Converts *expr* to a string
TO_DATE	*expr*	Converts *expr* to a date. *expr* must be in the form ‘yyyymmdd hh:mn:ss’
TO_INT	*expr*	Converts *expr* to an integer number
TO_FLOAT	*expr*	Converts *expr* to a floating point number
TO_MONEY	*expr*	Converts *expr* to a currency amount
TS2DAY	*expr*	Truncates date *expr* to a date with no time
UPPER	*expr*	Returns the string *expr* in uppercase

ABW Macros

Macros return a single value. Macros can be used wherever an expression is valid.

Macro Syntax

Macro	Description
MAX_DATE	Highest acceptable date (20991231 23:59:59)
MIN_DATE	Lowest acceptable date (19000101 00:00:01)
NO	0 (FALSE)
NOW	Current date and time (returns a DATE not a CHAR)
TODAY	Current date (at 00:00:00) (returns a DATE not a CHAR)
YES	1 (TRUE)

ASQL Program (asql.exe)

This program executes ASQL statements contained either:

- In a file (e.g. installation scripts)
- In an “SQL query”, see below

The SQL in an "SQL Query" is executed in one of two ways:

• AG16	This server process writes the SQL in the query series to a file in the folder pointed to by the AGRESSO_SCRATCH Environment Variable and then runs asql.exe on this file.
• ‘query’ parameter	Server processes that have a ‘query’ parameter, such as GL07, run the query series directly via an API call.

You can use either AGRESSO SQL syntax or native database syntax. If the SQL statements are in a file then its name normally ends with ".asq". This .asq file is referred to by the parameter **-F** at the command line when running the program.

When an .asq file is executed status information is printed to a log file. This file is called asql.log, and is found in the directory pointed to by the AGRESSO_LOG environment variable, if AGRESSO_LOG is not defined, the log file will be created in the current directory.

Parameters

The following is a description of the asql.exe program’s parameters. Some of the parameters differ from platform to platform. We call these the connection parameters. The others are called general parameters and are common to all platforms.

Connection Parameters

Connecting to a database is considerably easier using an AGRESSO Data Source than otherwise

Connecting using an AGRESSO Data Source

- **-D***agresso_data_source*
  This parameter refers to an AGRESSO data source. Before you can run asql.exe in this way you must set up that data source using the AGRESSO Service Manager (see documentation elsewhere).

  NB The **–D** parameter has a different meaning if you connect without using an AGRESSO Data Source.

Connecting without using an AGRESSO Data Source

NB! If you connect to an AGRESSO database without using an AGRESSO Data Source then the Environment variables are not accessible.

In special cases asql.exe is used on a non-AGRESSO database e.g. pre 5.5 AGRESSO Budget (Budwin).

The parameters to connect without using an AGRESSO Data Source are:

- **-U***database_user*
- **-P***user_password*
- **-D***database_name*
  ODBC / MS Sql Server: Only use if you want to override the user's default database.
  Informix: The database name
  Oracle / Sybase: Not used.
- **-S***native_source*

ODBC / MS Sql Server:	ODBC datasource, the database in the connection string is not used, either the user's default or the one specified by the **-D** parameter is used.
Oracle:	Oracle Net Service Name
Sybase:	Sybase Server Name

- **-I***interface*
  Specify RDBMS type:
  - ODBC (the default)
  - ORACLE
  - SYBASE
  - NATIVE
    With NATIVE, you must also use the **-d***db_driver* parameter.
- **-d***db_driver*
  An AGRESSO database driver. On an -INATIVE connection you must specify the driver to use, e.g. agrodbc.dll, agrora.dll, agrsyb.dll, agrinf.dll.

General Parameters

- **-f***path*
  This parameter is not required. If the .asq files you want to use are not in the current directory, you can tell asql.exe which directory they are in using this parameter. It is normally used in conjunction with the **-h** parameter.

  This parameter will override the environment variable AGRESSO_SCRIPT.

  If the **-h** parameter is used, and the list file contains path names, this parameter is ignored.
- **-h***list_file*
  A list file is a file containing the names of one or more .asq files, the default value is "asql.lst". By using the **–h** parameter, asql.exe will run all the .asq files listed in the list file. The files listed in the list file can have no paths or full or relative paths. If a path is given, it will override the **–f** parameter and AGRESSO_SCRIPT environment variable.

  This parameter will override the **–F** parameter.
- **-F***asq_file*
  The name of the file containing the SQL commands to be executed. This file will normally have the ending .asq. You can give a full path name. If no path name is given, asql.exe will first look in the directory pointed to by the environment

variable AGRESSO_SCRIPT. If this environment variable is not defined, it will look in the current directory.

This parameter is overridden by the **–h** parameter.

- **-x**
  This parameter only has meaning when running more than one .asq file (i.e. the **-h** parameter is used). It is used in conjunction with the **ON ERROR EXIT** command. The scope of the **ON ERROR EXIT** command is only the current file. This means that if an error occurs after this command, asql.exe will skip the rest of the current file, but continue with the next one. If you want asql.exe to cancel all files when an error occurs, use the **-x** parameter.
- **-v**[+]
  To increase the "verbosity" of the logged information, in particular to include the text of the SQL commands in the log file use the **-v** parameter.

  Every time you run asql.exe with the **–v** parameter, the current file is replaced with the new one; if you don't want to delete the old file, but append to it, use the **–v+** parameter.

Environment variables

The following environment variables are used by ASQL

- **AGRESSO_EXPORT**
  This environment variable can be used by the COPY statement. It points to the directory where the data file is.
- **AGRESSO_IMPORT**
  This environment variable can be used by the COPY statement. It points to the directory where the data file is.
- **AGRESSO_LOG**
  The directory (with full path) to where the log file is created. If not set, the current directory is used.
- **AGRESSO_SCRATCH**
  The directory (with full path) in which AG16 puts its generated .asq file.
- **AGRESSO_SCRIPT**
  The directory (with full path) in which.asq file(s) can be found. It is not compulsory. If not set, the current directory is used.

Appendices

Using Variables

ASQL allows you to define and use variables. These must be defined outside Database Blocks, but can be used inside them. Variables can be used almost everywhere in a statement.

- You define them using the DEFINE statement.
- You give them values using a SELECT… INTO… statement.
- You access a variable's contents by prefixing its name with a colon (e.g. :myvar).
- You cannot use variables inside strings (e.g. "…SET col_name = 'A:myvar'…" is invalid, whereas "…SET col_name = CONCAT('A', :myvar)…" is valid as long as myvar has been defined as type CHAR)

Example:

```
define description char (60)
/
define mytable ident (16)
/
define last_update date
/
select 'testtab', description, last_update
  into :mytable, :description, :last_update
  from org_table
/
update :mytable
   set description = :description
 where last_update < :last_update
/
```

Quoted Strings

When interpreting a delimited file the problem can arise that a text field contains the character that is being used as the field delimiter, for example in a full-stop delimited file:

```
1.ABC Holdings plc.100000
2.BA Smith & Sons Ltd..5000
```

There are two records with three fields each (apar_id, apar_name, credit_limit) but because the apar_name of the second row contains an extraneous full-stop it will confuse a 'COPY IN' into thinking that this row has four fields:

1.	apar_id	2
2.	apar_name	BA Smith & Sons Ltd
3.	credit_limit	blank
4.	????	5000.

The usual solution to this is to quote the text fields in the file and to ignore delimiter characters inside the quotes, e.g.

```
1.'ABC Holdings plc'.100000
2.'BA Smith & Sons Ltd.'.5000
```

or

```
1."ABC Holdings plc".100000
2."BA Smith & Sons Ltd.".5000
```

If you are unfortunate enough to need a quote inside a quoted string then you “double it”, e.g.:

`Mike's example` or `It's "well wicked"`

become

`'Mike''s example'` or `"It's ""well wicked"""`

This is how a comma separated value (or CSV) file is formatted, the delimiting character being a “,”.

Unfortunately ‘COPY IN’ does not recognize this solution and ‘COPY TO’ doesn’t quote strings automatically.

There isn’t really anything that you can do about ‘COPY IN’, but as long as you don’t have any embedded quotes in your text you can work around the problem in ‘COPY TO’ by using the CONCAT function, e.g.

```
SELECT … CONCAT('''', CONCAT(my_column, '''')) …
```