SMASER

SQL Reference Manual

TRIPsystem
Product Documentation

End User License Agreement

All rights to this software, its documentation and logotypes of the TRIP product family and software (altogether “Software”) supplied by Smaser AG (Smaser) are exclusively owned by Smaser.

The transfer of this Software, solutions or parts thereof requires the prior written agreement of Smaser. Furthermore, the customer has the right to use licensed Software and / or process solutions supplied by Smaser to the extent specified in his contract with Smaser.

The free-to-use non-commercial version doesn’t require a prior written agreement with Smaser but such customers, organizations and/or third parties agree by using the software and / or solution of Smaser to be strongly obliged to keep all rights to this software, documentation and logotypes of the TRIP product family absolutely uninfringed and protected..

Table of Contents

About this Document

This document describes the SQL functionality as it is available in TRIPsystem 8.0, and focuses on how TRIP databases are mapped to the relational model, various limitations and other considerations when using SQL with TRIP.

SQL functionality in TRIP versions older than 8.0 were available in the separately installed TRIPsql product. From TRIP version 8.0 the functionality is integrated into TRIPsystem, although still requiring it to be enabled in the license.

The client drivers via which the SQL functionality is used are not covered by this manual. See the separate installation packages and document sets for them:

- ODBC Driver
- JDBC Driver
- ADO.NET Data Provider

Overview

The purpose of SQL in TRIP is twofold. The first is to give database programmers familiar with SQL and programming frameworks such as ODBC and JDBC a more familiar, albeit somewhat limited, way to access TRIP. The second is to TRIP-enable use of tools such as reporting engines.

The TRIP SQL engine is a server-side implementation and maps SQL statements to a sequence of TRIP commands. It executes in the context of the TRIP kernel and returns result data as XML documents.

Installation and Upgrade

The SQL functionality is part of the TRIPsystem installation, and does not require a separate installation step.

When upgrading from a pre-8.0 version of TRIP where TRIPsql is in use, that version of TRIPsql will no longer be used, even if re-installed. Old TRIPsql installations can be removed after a successful upgrade to TRIP 8.0 or later.

The following table describe the various optional settings for the tdbs.conf non-privileged section that can be used to control the SQL engine:

Symbol in tdbs.conf	Default	Description
TRIPSQL_BASES	(none)	Default table location. This is the name of the directory in which TRIP databases are created by the CREATE TABLE call. **NOTE**: This has been superseded by the TDBS_BASES symbol. The TRIPSQL_BASES symbol is supported for backward compatibility but will only be used for new tables if TDBS_BASES is not defined.
TRIPSQL_LOG	0	SQL engine log level (0=off, 1=fatal errors, 2=errors, 3=warnings, 4=information, 5=debug). Log files are placed in the directory pointed to by TDBS_LOG. Default value is 0 (off).
SQL_BATCHINDEX	0	Controls if TRIP database indexing after modification (INSERT, UPDATE or DELETE) is done in batch (asynchronous) or in a synchronous fashion. Possible values are 0=synchronous, 1=asynchronous.
SQL_LAZYINDEX	1	Controls if TRIP database indexing is done only when required (prior to a SELECT against a database containing unindexed records), or every time there is a modification via UPDATE, INSERT or DELETE. Possible values are 0=every time, 1=only when required.
SQL_SELECTTEXT	0	Determines how fields of type TEXT are returned. Possible values are 0=external, 1=inline. If external retrieval is enabled, TEXT values need to be requested as separate objects (CLOBs). Inline retrieval returns the TEXT values as is, or in base64 if layout is retained.
SQL_SELECTBLOB	0	Determines how fields of type STRING are returned. Possible values are 0=external, 1=inline. If external retrieval is enabled, STRING values need to be requested as separate objects (BLOBs). Inline retrieval returns the STRING values encoded in base64.

Supported SQL Syntax

Concepts

SQL Concept	Description
TABLE	A table in TRIP SQL is a TRIP database.
ROW	What goes into a row depends on the structure of the TRIP database that is used. If a record in a TRIP database lacks part records, the entire record is one row. For records that have part records, each TRIP record will be seen as a collection of rows made out of the product of head and part records. I.e. if a record has 10 part records, the SQL table will show that record as 10 rows, where the data from the head is repeated on each row.
COLUMN	A column in TRIP SQL is a TRIP field. The value of a column is the first subfield for fields of type INTEGER, NUMBER, PHRASE, DATE and TIME. Values from columns of other types (i.e. types that cannot have subfields) are returned whole.
LONG VARCHAR LONG VARBINARY	Fields of type TEXT and STRING, which are mapped to the types LONG VARCHAR and LONG VARBINARY respectively, are treated slightly differently when sent to the client. Since the values in such TRIP fields can be large, they are by default requested separately by the client drivers in order to keep the initial response time as low as possible. There is a session property controls the behavior with regard to LONG VARBINARY/VARCHAR data. See the Special Expressions section for more details.
VALUELIST (subfields)	It is quite common for a field in a TRIP database to have more than one value per record. If an application is using the ODBC driver, a plain SELECT statement will return only the first subfield. In order to get the entire list of subfields, the TRIP SQL extension function VALUELIST can be used to provide a single output with all the subfields in a caller-defined delimited list. The JDBC driver additionally exposes fields with more than one value as an array. The easiest way for a JDBC application to obtain more than one subfield is therefore via the Array interface.
SCHEMA	Although database schemas are not supported by TRIP SQL, the schema qualifier is allowed in table reference clauses. The only time when schema must be used in TRIP SQL is in metadata queries that lists tables and column definitions. See the Special Expressions section for more details.

Assumptions and Dependencies

The TRIP SQL engine supports the minimum SQL grammar, which means that the SQL engine must be able to respond to the following statements:

- Simple SELECT
- INSERT, UPDATE, DELETE
- Simple Expressions
- Data types: CHAR, VARCHAR and LONG VARCHAR
- DDL: Create and drop table

In addition, the following features from core and extended SQL grammar are also built into the SQL engine:

- Aggregate functions: COUNT, SET, MAX, MIN, AVG
- DDL: Grant and revoke, primary keys, referential integrity (REFERENCES)
- DML: Inner and outer joins, DISTINCT, UNION
- Data types: NUMERIC, INTEGER, LONG VARBINARY, DATE, TIME

Data Types

The following data types are supported by the SQL engine. Their corresponding mappings are shown in the table below.

SQL Type	TRIP Field Type
INTEGER	INTEGER
NUMERIC	NUMBER
VARCHAR	PHRASE
DATE	DATE
TIME	TIME
LONG VARCHAR	TEXT
LONG VARBINARY	STRING
SERIAL	INTEGER (record number field)

Querying

This section describes how TRIP queries maps to SQL conditions.

Matching Terms and Phrases

Simple term or phrase matching expressions are pretty straight forward. A CCL expression sequence like

```
BASE ALICE
FIND SPEAKER="White Rabbit"
SHOW F=SPEAKER,PERSON,TXT
```

will look like this in TRIP SQL:

```
SELECT SPEAKER,PERSON,TXT
  FROM ALICE
 WHERE SPEAKER='White Rabbit'
```

Wildcard Matching

TRIP SQL supports '%' for matching zero or more characters, and '_' to match exactly one character. The CCL expression:

```
BASE ALICE
FIND SPEAKER=#OOK
SHOW F=SPEAKER
```

translates to:

```
SELECT SPEAKER
  FROM ALICE
 WHERE SPEAKER LIKE '%OOK'
```

Pattern Matching

There is one kind of pattern matching that always takes place. That is when one is searching for a term in a LONG VARCHAR (TEXT) column like:

```
SELECT TXT
  FROM ALICE
 WHERE TXT='Jabberwocky'
```

The word "Jabberwocky" will match all TXT values that contain the word, regardless of whether the word is the entire value or is one of many in a paragraph or sentence.

The LIKE predicate is also supported to some extent. A typical use in TRIPsql is searching for a truncated term:

```
SELECT *
  FROM ALICE
 WHERE PERSON LIKE 'Jabber%'
```

NULL Values

TRIP SQL supports the IS [not] NULL predicate that can be used to test if a value is (or is not) null. A TRIP field is regarded to be null if the field has no value. Note that a VARCHAR column may have an empty string as its value and still not be equal to null.

Example:

```
SELECT *
  FROM ALICE
 WHERE PERSON IS NULL
```

Closed Range Queries

The BETWEEN predicate is supported by TRIP SQL to facilitate range queries. This CCL expression:

```
BASE CORR
FIND DAY=1984-03-01 TO 1984-03-28
SHOW
```

translates to the following SQL statement:

```
SELECT *
  FROM CORR
 WHERE DAY BETWEEN 1984-01-01 AND 1984-03-28
```

Open-ended Date Range Queries

The BETWEEN predicate cannot be used for open-ended range queries. This CCL expression:

```
BASE CORR
FIND DAY=FROM 1985-01-01
SHOW
```

translates to the following SQL statement (note the required use of the DATE type casting function):

```
SELECT *
  FROM CORR
 WHERE DAY >= DATE('1985-01-01')
```

JOINS

Both inner and outer joins can be used with TRIP SQL to pose a query against more than one table. This is an example of a three-way inner join against a table PERSON with the columns SSN, NAME, MOTHER_SSN and FATHER_SSN.

```
SELECT F.NAME AS 'FATHER', M.NAME AS 'MOTHER', C.NAME AS 'CHILD'
  FROM PERSON F, PERSON M, PERSON C
 WHERE F.SSN = C.FATHER_SSN
   AND M.SSN = C.MOTHER_SSN
```

It is also possible to use the SQL-92 syntax for JOINs. This lists all persons who have known fathers:

```
SELECT F.NAME AS 'FATHER', C.NAME AS 'CHILD'
  FROM PERSON F INNER JOIN PERSON C ON F.SSN=C.FATHER_SSN
```

Subselects

Subselects are supported by TRIP SQL. Below is a typical, albeit somewhat contrived, example of a subselect.

```
SELECT m,name FROM person m
  WHERE m.parent_id = ( SELECT p.id FROM person p
                        WHERE p.name='Donald' )
```

Subselects can also be used with INSERT statements, like so:

```
INSERT INTO donalds (id, name)
       SELECT id, name FROM person WHERE name='Donald'
```

UPDATE statements can use subselects in place of values, if the subselect returns NULL or a single value:

```
UPDATE donalds
   SET name=(SELECT name FROM person WHERE ID=123)
 WHERE id=123
```

Finally, a subselect can be used in the column projection list of a SELECT statement, provided it returns NULL or a single value:

```
SELECT name, (SELECT name FROM countries WHERE id=123)
  FROM person
 WHERE name='Donald'
```

Object Assignment

Assigning large quantities of data to a LONG VARCHAR (TEXT) or LONG VARBINARY (STRING) column is a common task in TRIP applications. For this purpose, TRIP SQL offers a grammar extension called OBJECT. The OBJECT function will usually be used in parameterized statements from one of the client drivers. See the Java JDBC example below.

```
void storeBlob(Connection con, String keyvalue,
               String blobcolumn, byte[] data) {
   try {
      String strStmt = "UPDATE MYDB SET OBJECT(?,?) WHERE MYKEY=?";
      PreparedStatement stmt = con.prepareStatement(strStmt);
      stmt->setString(1,blobcolumn);
      stmt->setBytes(2,data);
      stmt->setString(3,keyvalue);
      stmt->execute();
   }
   catch (SQLException e) {
      e.printStackTrace();
   }
}
```

For further information, refer to the syntax description for the UPDATE command.

Aggregate Functions

Aggregate functions are used to derive a value from data in a column. These are used in the column projection list of SELECT statements, typically qualified by the GROUP BY clause.

Function	Description
AVG	Average value
COUNT	Number of rows
MIN	Smallest value
MAX	Largest value
SUM	Sum of values

Scalar Functions

Scalar functions in TRIP SQL returns a single value, based on an optional input value.

Function	Description
CURRENT_DATE	Returns the current date as a DATE value
CURRENT_TIME	Returns the current time as a TIME value
CURRENT_TIMESTAMP	Returns the current date and time as a VARCHAR-formatted timestamp value.
VALUELIST	Transforms a set of subfield values as a single output value. This function may only be used in the column projection list of a SELECT statement.

DML Statements

SELECT

Syntax

```
select_stmt :== SELECT [DISTINCT] select_list FROM table_list
[ WHERE condition ] [ GROUP BY column(s) ] [ ORDER BY column(s) ] [
UNION select_stmt ]

select_list ::= [select_list,] column_ref | aggregate_ref |
const_ref | select_stmt

column_ref ::= [table.]column [ AS alias ]

aggregate_ref ::= aggregate_expr [ AS alias ]

aggregate_expr ::= COUNT(*|[DISTINCT] column) | SUM(column) |
AVG(column) | MIN(column) | MAX(column) | valuelist_ref

const_ref ::= constant_value AS alias

table_list ::= [table_list,] ( table_ref | qualified_join )

table_ref ::= [schema.]table [alias]
```

```
valuelist_ref ::= VALUELIST ( column_ref, delimiter, vl_range_spec
)

vl_range_spec ::= maxListSize | startNum, endNum

condition ::= expression comparison_operator expression          |
              expression [NOT] BETWEEN expression AND expression |
              expression IS [NOT] NULL       |
              char_expr [NOT] LIKE pattern   |
              subselect_condition            |
              condition boolean_operator condition

comparison_operator ::=  = | != | ~= | > | < | >= | <=

boolean_operator ::= AND | OR

expression ::= [table.]column   |
               text    |
               number

constant_value ::= text|number|DATE('date_expr')|time_expr|NULL

subselect_condition ::= [table.]column comparison_operator (
select_stmt ) | [table.]column [NOT] IN ( select_stmt )

qualified_join ::= table_ref [NATURAL] join_type JOIN table_ref
join_specification

join_type ::= INNER | outer_join_type OUTER

outer_join_type ::= LEFT | RIGHT

join_specification ::= join_condition | named_columns_join

join_condition ::= ON condition

named_column_join ::= USING ( column_name_list )

column_name_list ::= [table.]column [ { , [table.]column } ... ) ]
```

SELECT Examples

This section contains a number of example SELECT statements. These statements can be input via the client drivers.

This one shows a simple select expression that returns the number of rows in the ALICE table.

```
SELECT COUNT(*) As "Count"
  FROM ALICE
```

This example shows a select expression using the VALUELIST function to retrieve a list of TRIP subfields with a custom delimiter.

```
SELECT VALUELIST(PERSON, ", "), SPEAKER
  FROM ALICE
 WHERE TXT="jabberwocky"
```

INSERT

Syntax

```
INSERT [IGNORE] INTO table [ column_list ] value_clause

value_clause ::= VALUES ( value_list )  |  select_stmt
```

INSERT Examples

A simple, typical insert statement:

```
INSERT INTO MYTABLE ( idcol, mycol )
VALUES ("id02","abc123")
```

An insert statement using a select statement to feed data into the insert while ignoring duplicate key errors:

```
INSERT IGNORE INTO orphans ( id, name )
       SELECT id, name FROM person WHERE parent_id IS NULL
```

UPDATE

Syntax

```
UPDATE  table [alias]
    SET  column_assign_list|object_assignment
[ WHERE condition ]

column_assign_list ::= [column_assign_list,] column_assignment
column_assignment ::= [table.]column = text|number|select_stmt
object_assignment ::= OBJECT ( column_ref, object_data )
object_data ::= "FILE=filename"|"DATA=text"|"DATA[BASE64]=binary"
```

UPDATE Examples

This example shows a simple update statement.

```
UPDATE mytable
   SET mycol="newdata"
 WHERE idcol="id01"
```

This example shows an update statement where a binary object is being stored into a TRIP string field. In this case, the TRIP SQL engine is assumed to run on the same machine as the client issuing this statement. If client and server are on different machines, use one of the "DATA" parameters to SET OBJECT.

```
UPDATE mytable
   SET OBJECT(picture,"FILE=c:\pics\me.jpg")
 WHERE idcol="id02"
```

NOTE: When the OBJECT function is used from the JDBC and ODBC drivers, it is important NOT to prepend any of the three prefixes ('FILE=', 'DATA=' or 'DATA[BASE64]=') to the data being uploaded, since it is being taken care of by the drivers.

DELETE

Syntax

```
DELETE [FROM] table [alias] [ WHERE condition ]
```

DELETE Example

This example shows a delete statement.

```
DELETE FROM mytable
 WHERE idcol="id01"
```

DDL Statements

CREATE TABLE

The create table operation actually creates a TRIP database. If a location is not explicitly specified, the TRIP database files for the table are placed in the location specified by the TDBS_BASES setting in tdbs.conf (or TRIPSQL_BASES if that is defined and TDBS_BASES is not).

Syntax

```
CREATE TABLE [ IF NOT EXISTS ] table ( column_spec_list ) [ WITH
table_options ]
column_spec_list ::= [column_spec_list,] column_spec
column_spec ::= column datatype [default_value] [constraint]

constraint ::= notnull | reference | primary_key

default_value ::= DEFAULT defaultvalue

notnull ::= NOT NULL

primary_key ::= PRIMARY KEY

reference ::= REFERENCES table ( column ) [ref_mode_update]
[ref_mode_delete]
ref_mode_update ::= ON UPDATE reference_option
ref_mode_delete ::= ON DELETE reference_option
reference_option ::= RESTRICT | CASCADE | SET NULL | NO ACTION |
SET DEFAULT

table_options ::= [table_options,] table_option
table_option  ::= LOCATION = location | CHARACTER SET = charset
```

Examples

This example shows a basic create table statement.

```
CREATE TABLE mytable (
   idcol VARCHAR PRIMARY KEY,
   mycol VARCHAR )
WITH LOCATION = TDBS_BASES
```

The following example shows create table statements with a default value and a reference dependency.

```
CREATE TABLE master (
   ssn VARCHAR NOT NULL,
   m_name VARCHAR DEFAULT 'John Doe')

CREATE TABLE slave (
   s_name VARCHAR REFERENCES master(m_name)
                  ON DELETE RESTRICT
                  ON UPDATE CASCADE,
   phone VARCHAR)
```

DROP TABLE

Syntax

```
DROP TABLE [IF EXISTS] table
```

Example

This example shows a drop table statement.

```
DROP TABLE mytable
```

GRANT

The grant operation grants table access to a user or group.

Syntax

```
GRANT ALL|READ PRIVILEGES [ ( column_grant_list ) ]
   ON table
   TO user|group

column_grant_list ::= [column_grant_list,] [table.]column
```

Example

This example shows a grant statement.

```
GRANT ALL PRIVILEGES ON mytable TO public
```

REVOKE

The revoke operation revokes table access from a user or group.

Syntax

```
REVOKE ALL|WRITE PRIVILEGES [ ( column_grant_list ) ]
   ON table
   FROM user|group
```

Example

This example shows a revoke statement.

```
REVOKE ALL PRIVILEGES ON mytable FROM public
```

Special Expressions

Metadata Retrieval

TRIP SQL supports retrieval of metadata on tables, columns and foreign key constraints. This is accomplished by prefixing the table names TABLES, COLUMNS and KEYREFS with the special schema name TDBS. Please note the rather rigid syntax for these two special statements. While the SQL parser may accept other combinations, the result can in such cases be unpredictable.

```
SELECT meta_select_list FROM TDBS.TABLES [ WHERE tables_column =
constant_value ]

SELECT meta_select_list FROM TDBS.COLUMNS [ WHERE columns_column =
constant_value ]

SELECT meta_select_list FROM TDBS.KEYREFS [ WHERE keyrefs_column =
constant_value ]

meta_select_list ::= [meta_select_list,] column_ref | const_ref

tables_column ::= NAME | OWNER

columns_column ::= TABLENAME | COLNAME | SQLTYPE | NULLABLE |
ORDINAL | PRIMARYKEY

keyrefs_column ::= FKTABLE_NAME | FKCOLUMN_NAME | PKTABLE_NAME |
                   PKCOLUMN_NAME | DELETE_RULE | UPDATE_RULE
```

The example below shows a statement for requesting column information on a specific table.

```
SELECT COLNAME, SQLTYPE
  FROM TDBS.COLUMNS
 WHERE TABLENAME='ALICE'
```

Array/Subfields

Arrays as column values does not really occur within relational databases, since such constructs violate the first normal form which states that no table must have any repeating groups. However, fields with multiple values (i.e. arrays) are very common within TRIP databases where they are known under the term "subfields". Because of this, TRIP provides an extension to the SQL column reference that enables applications to access multivalued columns using an array index.

The following is an example an UPDATE statement that updates a multivalued column.

```
UPDATE MYTABLE
   SET FLAGCOLOR[1]='BLUE',
       FLAGCOLOR[2]='YELLOW'
 WHERE COUNTRY='SWEDEN'
```

Retrieval of multivalued columns can be done via the JDBC Array interface, or via the VALUELIST function.

The following is an example of a SELECT statement that retrieves the flag colors for the Swedish flag as a comma-separated list.

```
SELECT VALUELIST(FLAGCOLOR,', ') As 'Flag Colors'
  FROM MYTABLE
 WHERE COUNTRY='SWEDEN'
```

If the application is using the JDBC Array interface, the above SELECT statement can be simplified to:

```
SELECT FLAGCOLOR
  FROM MYTABLE
 WHERE COUNTRY='SWEDEN'
```

Properties

A syntax for assigning and getting property values is supported by TRIP. This is a TRIP SQL specific extension to the SQL grammar.

A property is a session-specific value that controls the behavior of TRIPsql in some manner. Each property is also associated with a nonprivileged tdbs.conf symbol that allows for overriding the default property value on a server-wide basis.

Property Name	Value Type	Default	tdbs.conf Symbol	Description
BATCHINDEX	Boolean	FALSE (synchronous)	SQL_BATCHINDEX	Controls if TRIP database indexing after modification is done in batch (asynchronous) or synchronously.
LAZYINDEX	Boolean	TRUE (on)	SQL_LAZYINDEX	Controls if TRIP database indexing is done only when required (true, default), or every time there is a modification via UPDATE, INSERT or DELETE.
LOGGING	Boolean	FALSE (off)	TRIPSQL_LOG	Controls server-side logging. If enabled, log

Property Name	Value Type	Default	tdbs.conf Symbol	Description
				files are created in the directory specified by the TRIPrcs symbol TDBS_LOG .
SELECTTEXT	Boolean	FALSE (separate)	SQL_SELECTTEXT	Controls if TEXT field values are returned inline or have to be fetched separately. This value should be left as FALSE unless all text (LONG VARCHAR) values are small (~1K).
SELECTBLOB	Boolean	FALSE (separate)	SQL_SELECTBLOB	Controls if STRING field values are returned inline or have to be fetched separately. This value should be left as FALSE unless all LONG VARBINARY values always have to be processed or are small (~1K).

Syntax for property retrieval and assignment:

```
GET PROPERTY propertyname

SET PROPERTY propertyname=propertyvalue

propertyvalue::= text | number | boolean
```

Below are examples on how to retrieve and assign property values.

```
GET PROPERTY LOGGING

SET PROPERTY SELECTTEXT=TRUE
```

Partially Supported Expressions

DISTINCT + ORDER BY

The DISTINCT keyword and the ORDER BY clause cannot be used within the same SELECT statement. The following statement is illegal:

```
SELECT DISTINCT NAME, EMAIL FROM CONTACTS ORDER BY EMAIL
```

The following statements are legal:

```
SELECT DISTINCT NAME, EMAIL FROM CONTACTS

SELECT NAME, EMAIL FROM CONTACTS ORDER BY EMAIL
```

Numeric column reference in GROUP BY

The columns listed in a GROUP BY clause must name the columns; column numbers are not legal.

The following statement is illegal:

```
SELECT COUNT(EMAIL), NAME FROM CONTACTS GROUP BY 2
```

The following statement is legal:

```
SELECT COUNT(EMAIL), NAME FROM CONTACTS GROUP BY NAME
```

ORDER BY + GROUP BY

The GROUP BY clause and the ORDER BY clause cannot be used in the same statement.

The following statement is illegal:

```
SELECT COUNT(EMAIL), NAME FROM CONTACTS GROUP BY NAME ORDER BY 1
```

Using DISTINCT with aggregates

A SELECT DISTINCT statement cannot include aggregate functions.

The following statement is illegal:

```
SELECT DISTINCT COUNT(EMAIL), NAME FROM CONTACTS GROUP BY NAME
```

The following statement is legal:

```
SELECT COUNT(DISTINCT EMAIL), NAME FROM CONTACTS GROUP BY NAME
```