TSQL Commands

docs.intersystems.com/irislatest/csp/docbook/DocBook.UI.Page.cls

This chapter lists the supported TSQL commands for InterSystems IRIS® data platform in the following groups:

- Data Definition Language (DDL) statements:
  CREATE INDEX, DROP INDEX
  CREATE TRIGGER, DROP TRIGGER
  CREATE VIEW, DROP VIEW
  Parsed but ignored: CREATE DATABASE, DROP DATABASE
- Data Management Language (DML) statements:
- Query statements:
- Flow of control statements:
- Assignment statements:
  DECLARE, SET
- Transaction statements:
  Parsed but ignored: SAVE TRANSACTION
- Procedure statements
  CREATE PROCEDURE, DROP PROCEDURE
- Other statements
- InterSystems IRIS extensions
  OBJECTSCRIPT, IMPORTASQUERY

InterSystems IRIS implementation of TSQL accepts, but does not require, a semicolon command terminator. When importing TSQL code to InterSystems SQL, semicolon command terminators are stripped out.

Data Definition Language (DDL) Statements

The following DDL statements are supported.

CREATE TABLE

Defines a table, its fields, and their data types and constraints.

```
CREATE TABLE [schema. | #]tablename (fieldname datatype constraint [,...])
```

CREATE TABLE can specify a table using table (default schema), schema.table, or database..table syntax. See Table References.
A CREATE TABLE can create a temporary table by prefixing a # character to the table name. A temporary table can only be defined from a stored procedure; you cannot define a

temporary table from Dynamic SQL outside of a stored procedure. To create a fully-qualified temporary table name, use quotes around each name element such as the following: "SQLUser"."#mytemp".
A valid table name must begin with a letter, an underscore character (_), or a # character (for a local temporary table). Subsequent characters of a table name may be letters, numbers, or the #, $, or _ characters. Table names are not case-sensitive.

A field name must be a valid TSQL identifier. A field name can be delimited using square brackets. This is especially useful when defining a field that has the same name as a reserved word. The following example defines two fields named Check and Result:

```
CREATE TABLE mytest ([Check] VARCHAR(50),[Result] VARCHAR(5))
```

The optional CONSTRAINT keyword can be used to specify a user-defined constraint name for a field constraint or a table constraint. You can specify multiple CONSTRAINT name type statements for a field.

InterSystems SQL does not retain constraint names. Therefore these names cannot be used by a subsequent ALTER TABLE statement.

The table field constraints DEFAULT, IDENTITY, NULL, NOT NULL, PRIMARY KEY, [FOREIGN KEY] REFERENCES (the keywords FOREIGN KEY are optional), UNIQUE, CLUSTERED, and NONCLUSTERED are supported. The table constraint FOREIGN KEY REFERENCES is supported.

The field definition DEFAULT values can include the following TSQL functions: CURRENT_TIMESTAMP, CURRENT_USER, GETDATE, HOST_NAME, ISNULL, NULLIF, and USER.

The field definition IDENTITY constraint is supported and assigned a system-generated sequential integer. The IDENTITY arguments seed and increment are parsed, but ignored.

The TSQL CREATE TABLE command can create a sharded table. The syntax for the SHARD clause is the same as for the InterSystems SQL CREATE TABLE statement:

```
SHARD [ KEY fieldname { , fieldname2 } ] [ COSHARD [ WITH ] [(] tablename [)] ]
```

The CHECK field constraint is not supported. If a CHECK constraint is encountered while compiling TSQL source InterSystems IRIS generates an error message indicating that CHECK constraints are not supported. This error is logged in the compile log (if active), and the source is placed in the unsupported log (if active).

If the table already exists, an SQLCODE -201 error is issued.

The following Dynamic SQL example creates a temporary table named #mytest with four fields, populates it with data, then displays the results. The LastName field has multiple constraints. The FirstName field takes a default. The DateStamp field takes a system-defined default:

```
SET sql=9
SET sql(1)="CREATE TABLE #mytest (MyId INT PRIMARY KEY,"
SET sql(2)="LastName VARCHAR(20) CONSTRAINT unq_lname UNIQUE "
SET sql(3)="  CONSTRAINT nonull_lname NOT NULL,"
SET sql(4)="FirstName VARCHAR(20) DEFAULT '***TBD***',"
SET sql(5)="DateStamp DATETIME DEFAULT CURRENT_TIMESTAMP)"
SET sql(6)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1224,'Smith','John')"
SET sql(7)="INSERT INTO #mytest(MyId,LastName) VALUES (1225,'Jones')"
SET sql(8)="SELECT MyId,FirstName,LastName,DateStamp FROM #mytest"
SET sql(9)="DROP TABLE #mytest"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
WRITE status,!
SET result=statement.%Execute()
DO result.%Display()
```

Parsed But Ignored

The table constraint clauses WITH, ON, and TEXTIMAGE ON are parsed for compatibility, but are ignored. The index_options clause for the UNIQUE or PRIMARY KEY constraint is parsed for compatibility, but is ignored.

The following SQL Server parenthesized WITH options in a table constraint are parsed but ignored: ALLOW_PAGE_LOCKS, ALLOW_ROW_LOCKS, DATA_COMPRESSION, FILLFACTOR, IGNORE_DUP_KEY, PAD_INDEX, and STATISTICS_NORECOMPUTE.

The field constraints CLUSTERED and NONCLUSTERED are parsed for compatibility, but are ignored.

ALTER TABLE

Modifies the definition of a table, its fields, and their data types and constraints.

The following syntactical forms are supported:

```
ALTER TABLE tablename ADD fieldname datatype [DEFAULT value]
    [{UNIQUE | NOT NULL} | CONSTRAINT constraintname {UNIQUE | NOT NULL} ]
ALTER TABLE tablename ALTER COLUMN fieldname newdatatype
ALTER TABLE tablename DROP COLUMN fieldname [,fieldname2]
ALTER TABLE tablename ADD tableconstraint FOR fieldname
ALTER TABLE tablename DROP tableconstraint
ALTER TABLE tablename DROP FOREIGN KEY role
ALTER TABLE tablename ADD CONSTRAINT constraint DEFAULT defaultvalue FOR fieldname
ALTER TABLE tablename ADD CONSTRAINT constraint FOREIGN KEY
ALTER TABLE tablename DROP CONSTRAINT constraint
```

Specify tablename as described in Table References.

- ALTER TABLE...ADD fieldname can add a field definition or a comma-separated list of field definitions:

    - DEFAULT is supported.

    - NOT NULL is supported if the table contains no data. If the table contains data, you can only specify NOT NULL if the field also specifies a DEFAULT value.

    - UNIQUE is parsed but ignored. To establish a unique constraint use the CREATE INDEX command with the UNIQUE keyword.

  The full supported syntax for ALTER TABLE...ADD fieldname is as follows:

  ```
  ALTER TABLE tablename
     [ WITH CHECK | WITH NOCHECK ]
    ADD fieldname datatype [DEFAULT value]
      [{UNIQUE | NOT NULL} | CONSTRAINT constraintname {UNIQUE | NOT NULL} ]
      [ FOREIGN KEY (field1[,field2[,...]])
       REFERENCES tablename(field1[,field2[,...]]) ]
  ```

  WITH CHECK | WITH NOCHECK is parsed by InterSystems IRIS, but is ignored. In Transact-SQL, WITH CHECK | WITH NOCHECK provides an execution time check of existing data for a new or newly enabled constraint. InterSystems TSQL does not specifically support that, although InterSystems SQL will check existing data against a new constraint.

  The Sybase PARTITION BY clause is not supported.

- ALTER TABLE...ALTER COLUMN fieldname datatype can change the data type of an existing field. This command completes without error when the specified datatype is the same as the field’s existing data type.

- ALTER TABLE...DROP [COLUMN] fieldname can drop a defined field or a comma-separated list of defined fields. The keyword DELETE is a synonym for the keyword DROP.
    - Sybase: the COLUMN keyword is not permitted, the CONSTRAINT keyword is required: ALTER TABLE...DROP fieldname, CONSTRAINT constraint
    - MSSQL: the COLUMN keyword is required, the CONSTRAINT keyword is optional: ALTER TABLE...DROP COLUMN fieldname, constraint
- ALTER TABLE...DROP [CONSTRAINT] constraintname can drop a constraint from a field. The keyword DELETE is a synonym for the keyword DROP.
    - Sybase: the CONSTRAINT keyword is required.
    - MSSQL: the CONSTRAINT keyword is optional.
- ALTER TABLE...ADD CONSTRAINT...DEFAULT syntax does not create a field constraint. Instead, it performs the equivalent of an ALTER TABLE...ALTER COLUMN...DEFAULT statement. This means that InterSystems IRIS establishes the specified field default as the field property's initial expression. Because no field constraint is defined, this "constraint" cannot be subsequently dropped or changed.

  CHECK | NOCHECK CONSTRAINT is not supported by InterSystems IRIS TSQL. Specifying this CHECK or NOCHECK keyword generates an error message.

DROP TABLE

Deletes a table definition.

```
DROP TABLE [IF EXISTS] tablename
```

Deletes a table definition. You can delete both regular tables and temporary tables. (Temporary table names begin with a '#' character.) DROP TABLE ignores a nonexistent temporary table name and completes without error.

Specify tablename as described in Table References.
If tablename has an associated view, you must delete the view before you can delete the table.

The IF EXISTS clause is parsed but ignored.

CREATE INDEX

Creates an index for a specified table or view.

```
CREATE [UNIQUE] INDEX indexname ON tablename (fieldname [,fieldname2])
```

You can create an index on a field or a comma-separated list of fields.

You can create an index on the IDKEY (which is treated as a clustered index), on an IDENTITY field (which create an index on the %%ID field), on the Primary Key, or on other fields.

Specify tablename as described in Table References.
The UNIQUE keyword creates a unique value constraint index for the specified field(s).

The following Transact-SQL features are parsed, but ignored:

- The CLUSTERED/NONCLUSTERED keywords. Other than the IDKEY, which is implicitly treated as a clustered index, InterSystems TSQL does not support clustered indices.
- The ON dbspace clause.
- The ASC/DESC keywords.
- The INCLUDE clause.
- WITH clause options, such as WITH FILLFACTOR=n or WITH DROP_EXISTING=ON. The comma-separated list of WITH clause options can optionally be enclosed in parentheses.
- The ON filegroup or IN dbspace-name clause.

The following Transact-SQL features are not currently supported:

- Sybase index types.
- The IN dbspace clause.
- The NOTIFY integer clause.
- The LIMIT integer clause.
- Using a function name as an alternative to a field name.

The ALTER INDEX statement is not supported.

DROP INDEX

Deletes an index definition. You can delete a single index or a comma-separated list of indices, using either of the following syntax forms:

```
DROP INDEX tablename.indexname [,tablename.indexname]
```

```
DROP INDEX indexname ON tablename [WITH (...)] [,indexname ON tablename [WITH (...)] ]
```

tablename is the name of the table containing the indexed field. Specify tablename as described in Table References.
indexname is the name of the index. It can be a regular identifier or a quoted identifier.
The WITH (...) clause, with any value within the parentheses, is accepted by syntax checking for compatibility, but is not validated and performs no operation.

The IF EXISTS clause is not supported.

CREATE TRIGGER

Creates a statement-level trigger.

```
CREATE TRIGGER triggername ON tablename
[WITH ENCRYPTION]
{FOR | AFTER | INSTEAD OF} {INSERT | DELETE | UPDATE}
[WITH APPEND]
[NOT FOR REPLICATION]
AS tsql_trigger_code
```

You can create a trigger for one event (INSERT), or for a comma-separated list of events (INSERT,UPDATE).

Specify tablename as described in Table References.
The FOR, AFTER, and INSTEAD OF keywords are synonyms. A trigger is always pulled after the event operation is performed.

If there are multiple triggers for the same event or comma-separated list of events they are executed in the order the triggers were created.

The following clauses are parsed but ignored: WITH ENCRYPTION, WITH APPEND, NOT FOR REPLICATION.

InterSystems TSQL does not support row-level triggers.

You cannot include a CREATE TRIGGER statement in CREATE PROCEDURE code.

DROP TRIGGER

Deletes a trigger definition.

```
DROP TRIGGER [owner.]triggername
```

CREATE VIEW

Creates a view definition.

```
CREATE VIEW [owner.]viewname
    [WITH {ENCRYPTION | SCHEMABINDING | VIEW_METADATA}]
    AS select_statement
   [WITH CHECK OPTION]
```

A viewname must be a unique TSQL identifier. Specify viewname as described in Table References. If the view already exists, an SQLCODE -201 error is issued. A viewname can be a delimited identifier. For example, CREATE VIEW Sample.[Name/Age View].
By default, the view fields have the same names as the fields in the SELECT table. To specify different names for the view fields, specify field aliases in the SELECT statement. These aliases are used as the view field names:

```
CREATE VIEW NameAgeV
AS SELECT Name AS FullName,Age AS Years FROM Sample.Person
```

You can specify a WITH clause with a single keyword or a comma-separated list of keywords. For example: WITH SCHEMABINDING, ENCRYPTION, VIEW_METADATA. The ENCRYPTION, SCHEMABINDING, and VIEW_METADATA keywords are parsed but ignored.

The select_statement can only include an ORDER BY clause if this clause is paired with a TOP clause. If you wish to include all of the rows in the view, you can pair an ORDER BY clause with a TOP ALL clause. You can include a TOP clause without an ORDER BY clause. However, if you include an ORDER BY clause without a TOP clause, an SQLCODE -143 error is generated.

The select_statement can contain a UNION or UNION ALL.

The optional WITH CHECK OPTION clause prevents an update through the view that makes the record inaccessible to that view. It does this by checking the WITH clause in the SELECT statement. WITH CHECK OPTION binds to InterSystems SQL using the default of CASCADE.

The ALTER VIEW statement is not supported.

DROP VIEW

Deletes a view definition.

```
DROP VIEW viewname [,viewname2 [,...] ]
```

You can delete a single view, or a comma-separated list of views. Specify viewname as described in Table References.

DROP VIEW is not an all-or-nothing operation. It deletes existing views in the list of views until it encounters a nonexistent view in the list. At that point the delete operation stops with an SQLCODE -30 error.

The IF EXISTS clause is not supported.

CREATE DATABASE

CREATE DATABASE syntax is parsed to provide compatibility. No functionality is provided.

```
CREATE DATABASE dbname
```

Only this basic CREATE DATABASE syntax is parsed.

Sybase additional CREATE DATABASE clauses are not supported.

MSSQL attach a database and create a database snapshot syntax options are not supported.

The ALTER DATABASE statement is not supported.

DROP DATABASE

DROP DATABASE syntax is parsed to provide compatibility. No functionality is provided.

```
DROP DATABASE dbname
```

Data Management Language (DML) Statements

- TSQL can resolve an unqualified table name using a schema search path for a single DML statement in Dynamic SQL.
- TSQL cannot resolve an unqualified table name using a schema search path for multiple DML statements in Dynamic SQL. This includes multiple statements such as an explicit BEGIN TRANSACTION followed by a single DML statement.

DELETE

Deletes rows of data from a table. Both DELETE and DELETE ... FROM are supported:

```
DELETE FROM tablename  WHERE condition
```

```
DELETE FROM tablename FROM matchtablename WHERE tablename.fieldname = matchtablename.fieldname
```

Only very simple theta joins are supported (the FROM table clause is transformed into nested subqueries).

You can specify how DELETE executes by providing one or more execution options as a comma-separated list. You provide these options in a comment with the following specific syntax:

```
/* IRIS_DELETE_HINT: option,option2 */
```

Where option can be the following: %NOCHECK, %NOFPLAN, %NOINDEX, %NOLOCK, %NOTRIGGER, %PROFILE, %PROFILE_ALL. Refer to the InterSystems SQL DELETE command for details.
You can provide optimization hints to the DELETE FROM clause as a comma-separated list. You provide these hints in a comment with the following specific syntax:

```
/* IRIS_DELETEFROM_HINT: hint,hint2 */
```

Where hint can be the following: %ALLINDEX, %FIRSTTABLE tablename, %FULL, %INORDER, %IGNOREINDICES, %NOFLATTEN, %NOMERGE, %NOSVSO, %NOTOPOPT, %NOUNIONOROPT, and %STARTTABLE. Refer to the InterSystems SQL FROM clause for details.
The following table_hints are parsed but ignored: FASTFIRSTROW, HOLDINDEX, INDEX(name), NOLOCK, PAGLOCK, READCOMMITTED, READPAST, READUNCOMMITTED, REPEATABLEREAD, ROWLOCK, SERIALIZABLE, SHARED, TABLOCK, TABLOCKX, UPDLOCK, XLOCK. Table hints can be optionally preceded by the WITH keyword, and, if WITH is specified, optionally enclosed in parentheses. A list of table hints can be separated by either commas or blank spaces.

DELETE sets the @@ROWCOUNT system variable to the number of rows deleted, and the @@IDENTITY system variable to the IDENTITY value of the last row deleted.

The following options are not supported:

- MSSQL rowset functions.
- MSSQL OPTION clause.

INSERT

Inserts rows of data into a table. The following syntactic forms are supported:

```
INSERT [INTO] tablename (fieldname[,fieldname2[,...]]) VALUES (list_of_values)
```

```
INSERT [INTO] tablename (fieldname[,fieldname2[,...]]) SELECT select_list
```

The INTO keyword is optional. Specify tablename as described in Table References.
For the VALUES syntax, the VALUES keyword is mandatory for both MSSQL and Sybase. The (fieldname) list is optional if the list_of_values lists all user-specified fields in the order

defined in the table. If field names are specified, the list_of_values is a comma-separated list of values that matches the list of field names in number and data type.

You can specify how INSERT executes by providing one or more execution options as a comma-separated list. You provide these options in a comment with the following specific syntax:

```
/* IRIS_INSERT_HINT: option,option2 */
```

Where option can be the following: %NOCHECK, %NOFPLAN, %NOINDEX, %NOLOCK, %NOTRIGGER, %PROFILE, %PROFILE_ALL. Refer to the InterSystems SQL INSERT command for details.
The following table_hints are parsed but ignored: FASTFIRSTROW, HOLDINDEX, INDEX(name), NOLOCK, PAGLOCK, READCOMMITTED, READPAST, READUNCOMMITTED, REPEATABLEREAD, ROWLOCK, SERIALIZABLE, SHARED, TABLOCK, TABLOCKX, UPDLOCK, XLOCK. Table hints can be optionally preceded by the WITH keyword, and, if WITH is specified, optionally enclosed in parentheses. A list of table hints can be separated by either commas or blank spaces.

INSERT sets the @@ROWCOUNT system variable to the number of rows inserted, and the @@IDENTITY system variable to the IDENTITY value of the last row inserted.

The following options are not supported:

- (fieldname) DEFAULT VALUES or (fieldname) VALUES (DEFAULT). A field’s default value is used when the field is not specified in the INSERT statement.
- (fieldname) EXECUTE procname.
- Sybase insert load option clauses: LIMIT, NOTIFY, SKIP, or START ROW ID.
- Sybase insert select load option clauses: WORD SKIP, IGNORE CONSTRAINT, MESSAGE LOG, or LOG DELIMITED BY.
- Sybase LOCATION clause.
- MSSQL INSERT TOP clause.
- MSSQL rowset functions.

UPDATE

Updates values of existing rows of data in a table.

```
UPDATE tablename SET fieldname=value [,fieldname2=value2[,...]]
   [FROM tablename [,tablename2]] WHERE fieldname=value

UPDATE tablename SET fieldname=value[,fieldname2=value2[,...]]
   WHERE [tablename.]fieldname=value
```

These syntactic forms are vendor-specific:

- Sybase: the optional FROM keyword syntax is used to specify an optional table (or joined tables) used in a condition. Only very simple theta joins are supported (the FROM table clause is transformed into nested subqueries).
- MSSQL: the tablename.fieldname syntax is used to specify an optional table used in a condition.

The value data type and length must match the fieldname defined data type and length. A value can be a expression that resolves to a literal value or it can be the NULL keyword. It cannot be the DEFAULT keyword.

Specify tablename as described in Table References.
UPDATE supports the use of a local variable on the left-hand-side of a SET clause. This local variable can be either instead of a field name or in addition to a field name. The following example shows a SET to a field name, a SET to a local variable, and a SET to both a field name and a local variable:

```
UPDATE table SET x=3,@v=b,@c=Count=Count+1
```

You can specify how UPDATE executes by providing one or more execution options as a comma-separated list. You provide these options in a comment with the following specific syntax:

```
/* IRIS_UPDATE_HINT: option,option2 */
```

Where option can be the following: %NOCHECK, %NOFPLAN, %NOINDEX, %NOLOCK, %NOTRIGGER, %PROFILE, %PROFILE_ALL. Refer to the InterSystems SQL UPDATE command for details.
You can provide optimization hints to the UPDATE FROM clause as a comma-separated list. You provide these hints in a comment with the following specific syntax:

```
/* IRIS_UPDATEFROM_HINT: hint,hint2 */
```

Where hint can be the following: %ALLINDEX, %FIRSTTABLE tablename, %FULL, %INORDER, %IGNOREINDICES, %NOFLATTEN, %NOMERGE, %NOSVSO, %NOTOPOPT, %NOUNIONOROPT, and %STARTTABLE. Refer to the InterSystems SQL FROM clause for details.
The following table_hints are parsed but ignored: FASTFIRSTROW, HOLDINDEX,

INDEX(name), NOLOCK, PAGLOCK, READCOMMITTED, READPAST, READUNCOMMITTED, REPEATABLEREAD, ROWLOCK, SERIALIZABLE, SHARED, TABLOCK, TABLOCKX, UPDLOCK, XLOCK. Table hints can be optionally preceded by the WITH keyword, and, if WITH is specified, optionally enclosed in parentheses. A list of table hints can be separated by either commas or blank spaces.

UPDATE sets the @@ROWCOUNT system variable to the number of rows updated, and the @@IDENTITY system variable to the IDENTITY value of the last row updated.

The following Dynamic SQL example shows a simple UPDATE operation:

```
SET sql=9
SET sql(1)="CREATE TABLE #mytest (MyId INT PRIMARY KEY,"
SET sql(2)="LastName VARCHAR(20) CONSTRAINT nonull_lname NOT NULL,"
SET sql(3)="FirstName VARCHAR(20) DEFAULT '***TBD***')"
SET sql(4)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1224,'Smith','John')"
SET sql(5)="INSERT INTO #mytest(MyId,LastName) VALUES (1225,'Jones')"
SET sql(6)="INSERT INTO #mytest(MyId,LastName) VALUES (1226,'Brown')"
SET sql(7)="UPDATE #mytest SET FirstName='Fred' WHERE #mytest.LastName='Jones'"
SET sql(8)="SELECT FirstName,LastName FROM #mytest ORDER BY LastName"
SET sql(9)="DROP TABLE #mytest"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
WRITE status,!
SET result=statement.%Execute()
DO result.%Display()
```

The following options are not supported:

- Sybase ORDER BY clause.
- MSSQL OPTION clause.
- MSSQL TOP clause.
- MSSQL rowset functions.

READTEXT

Reads data from a stream field.

```
READTEXT tablename.fieldname textptr offset size
```

The MSSQL READTEXT statement returns stream data from a field of a table. It requires a valid text pointer value, which can be retrieved using the TEXTPTR function, as shown in the following example:

```
DECLARE @ptrval binary(16);
SELECT @ptrval = TEXTPTR(Notes) FROM Sample.Person
READTEXT Sample.Person.Notes @ptrval 0 0
```

The textptr must be declared as binary. A textptr is only defined for a text field that is not null. You can specify an initial non-null value for a text field using the INSERT statement.
The offset can be 0, a positive integer value, or NULL: 0 reads from the beginning of the text. A positive integer reads from the offset position. NULL reads from the end of the text; that is, it completes successfully but returns no value.

The size can be 0 or a positive integer value, or NULL: 0 reads all characters from the offset position to the end of the text. A positive integer reads the size number of characters from the offset position. NULL completes successfully but returns no value.

The MSSQL HOLDLOCK keyword is parsed but ignored.

WRITETEXT

Writes data to a stream field, replacing the existing data value.

```
WRITETEXT tablename.fieldname textptr value
```

The MSSQL WRITETEXT statement writes data to a stream field of a table. It requires a valid text pointer value, which can be retrieved using the TEXTPTR function, as shown in the following example:

```
DECLARE @ptrval binary(16);
SELECT @ptrval = TEXTPTR(Notes) FROM Sample.Person
WRITETEXT Sample.Person.Notes @ptrval 'This is the new text value'
```

The textptr must be declared as binary. A textptr is only defined for a text field that is not null. You can specify an initial non-null value for a text field using the INSERT statement.
The MSSQL BULK keyword is not supported.

The MSSQL WITH LOG keyword phrase is parsed but ignored.

UPDATETEXT

Updates data in a stream field.

```
UPDATETEXT tablename.fieldname textptr offset deletelength value
```

The MSSQL UPDATETEXT statement updates stream data from a field of a table. It requires a valid text pointer value, which can be retrieved using the TEXTPTR function. The following example updates the contents of the Notes stream data field by inserting the word ‘New’ at the beginning of the existing data value:

```
DECLARE @ptrval binary(16);
SELECT @ptrval = TEXTPTR(Notes) FROM Sample.Person
WRITETEXT Sample.Person.Notes @ptrval 0 0 'New'
```

The textptr must be declared as binary. A textptr is only defined for a text field that is not null. You can specify an initial non-null value for a text field using the INSERT statement.
The offset can be an integer value or NULL: 0 inserts the value at the beginning of the existing text. NULL inserts the value at the end of the existing text.

The deletelenth can be an integer value or NULL: 0 or NULL deletes no existing characters from the offset position before inserting the value. A positive integer deletes that number of existing characters from the offset position before inserting the value.

The MSSQL BULK keyword is not supported.

The MSSQL WITH LOG keyword phrase is parsed but ignored.

TRUNCATE TABLE

Deletes all of the data from a table.

```
TRUNCATE TABLE tablename
```

Invokes the InterSystems SQL TRUNCATE TABLE command, which deletes all rows from the specified table and resets the RowId (ID), IDENTITY, and SERIAL (%Counter) row counters and the stream field OID counter values.
You can specify how TRUNCATE TABLE executes by providing one or more execution options as a comma-separated list. You provide these options in a comment with the following specific syntax:

```
/* IRIS_DELETE_HINT: option,option2 */
```

Where option can be the following: %NOCHECK, %NOLOCK. Refer to the InterSystems SQL TRUNCATE TABLE for details.

Query Statements

SELECT

```
SELECT  [DISTINCT | ALL]
  [TOP [(]{ int | @var | ? | ALL}[)]]
  select-item {,select-item}
  [INTO [#]copytable]
  [FROM tablename [[AS] t-alias] [,tablename2 [[AS] t-alias2]] ]
  [[WITH] [(] tablehint=val [,tablehint=val] [)] ]
  [WHERE condition-expression]
  [GROUP BY scalar-expression]
  [HAVING condition-expression]
  [ORDER BY item-order-list [ASC | DESC] ]
```

The above SELECT syntax is supported. The following features are not supported:

- TOP nn PERCENT or TOP WITH TIES
- OPTION
- WITH CUBE
- WITH ROLLUP
- GROUP BY ALL
- GROUP WITH
- COMPUTE clause
- FOR BROWSE

TOP nn specifies the number of rows to retrieve. InterSystems TSQL supports TOP nn with a integer, ?, local variable, or the keyword ALL. The TOP argument can be enclosed in parentheses TOP (nn). These parentheses are retained, preventing preparser substitution. If SET ROWCOUNT specifies fewer rows than TOP nn, the SET ROWCOUNT value is used. The following Dynamic SQL example shows the use of TOP with a local variable:

```
  SET sql=3
  SET sql(1)="DECLARE @var INT"
  SET sql(2)="SET @var=4"
  SET sql(3)="SELECT TOP @var Name,Age FROM Sample.Person"
  SET statement=##class(%SQL.Statement).%New()
  SET statement.%Dialect="MSSQL"
  SET status=statement.%Prepare(.sql)
  SET result=statement.%Execute()
  DO result.%Display()
```

The select-item list can contain the following:

- field names, functions, and expressions

- the $IDENTITY pseudo-field name, which always returns the RowID value, regardless of the field name assigned to the RowID.
- an asterisk: SELECT * is supported. The asterisk means to select all fields in the specified table. You can qualify the asterisk with the table name or table alias: SELECT mytable.*.
- a subquery
- stream fields. A SELECT on a stream field returns the oref (object reference) of the opened stream object.

An INTO clause can be used to copy data from an existing table into a new table. By default, SELECT creates the INTO table with the same field names and data types as the fields selected from the source table. The INTO table cannot already exist. This INTO table can be a permanent table, or a temporary table, as shown in the following examples:

```
SELECT Name INTO Sample.NamesA_G FROM Sample.Person WHERE name LIKE '[A-G]%'
```

```
SELECT Name INTO #MyTemp FROM Sample.Person WHERE name LIKE '[A-G]%'
SELECT * FROM #MyTemp
```

You can specify a different name for an INTO table field by using a field alias, as shown in the following example:

```
SELECT Name AS Surname INTO Sample.NamesA_G FROM Sample.Person WHERE name LIKE '[A-G]%'
```

An INTO clause cannot be used when the SELECT is a subquery or is part of a UNION.

The FROM clause is not required. A SELECT without a FROM clause can be used to assign a value to a local variable, as follows:

```
DECLARE @myvar INT
SELECT @myvar=1234
PRINT @myvar
```

The FROM clause supports table hints with either of the following syntactic forms:

```
FROM tablename (INDEX=indexname)
FROM tablename INDEX (indexname)
```

Table hints can be optionally preceded by the WITH keyword, and optionally enclosed in parentheses. A list of table hints can be separated by either commas or blank spaces. The following table hints are parsed but ignored: FASTFIRSTROW, HOLDINDEX, NOLOCK, PAGLOCK, READCOMMITTED, READPAST, READUNCOMMITTED, REPEATABLEREAD, ROWLOCK, SERIALIZABLE, SHARED, TABLOCK, TABLOCKX, UPDLOCK, XLOCK.

You can provide optimization hints to the SELECT FROM clause as a comma-separated list. You provide these hints in a comment with the following specific syntax:

```
/* IRIS_SELECTFROM_HINT: hint,hint2 */
```

Where hint can be the following: %ALLINDEX, %FIRSTTABLE tablename, %FULL, %INORDER, %IGNOREINDICES, %NOFLATTEN, %NOMERGE, %NOSVSO, %NOTOPOPT, %NOUNIONOROPT, and %STARTTABLE. Refer to the InterSystems SQL FROM clause for details.
A WHERE clause can use AND, OR, and NOT logic keywords. It can group multiple search conditions using parentheses. The WHERE clause supports the following search conditions:

- Equality comparisons: = (equals), <> (not equals), < (less than), > (greater than), <= (less than or equals), >= (greater than or equals).
- IS NULL and IS NOT NULL comparisons.
- BETWEEN comparisons: Age BETWEEN 21 AND 65 (inclusive of 21 and 65); Age NOT BETWEEN 21 AND 65 (exclusive of 21 and 65). BETWEEN is commonly used for a range of numeric values, which collate in numeric order. However, BETWEEN can be used for a collation sequence range of values of any data type. It uses the same collation type as the field it is matching against. By default, string data types collate as not case-sensitive.
- IN comparisons: Home_State IN ('MA','RI','CT').
- LIKE and NOT LIKE comparisons, specified as a quoted string. The comparison string can contain wildcards: _ (any single character); % (any string); [abc] (any value in the set specified as a list of items); [a-c] (any value in the set specified as a range of items). InterSystems TSQL does not support the ^ wildcard. A LIKE comparison can include an ESCAPE clause, such as the following: WHERE CategoryName NOT LIKE 'D_%' ESCAPE '\'.
- EXISTS comparison check: used with a subquery to test whether the subquery evaluates to the empty set. For example SELECT Name FROM Sample.Person WHERE EXISTS (SELECT LastName FROM Sample.Employee WHERE LastName='Smith'). In this example, all Names are returned from Sample.Person if a record with LastName='Smith' exists in Sample.Employee. Otherwise, no records are returned from Sample.Person.
- ANY and ALL comparison check: used with a subquery and an equality comparison operator. The SOME keyword is a synonym for ANY.

WHERE clause and HAVING clause comparisons are not case-sensitive.

A HAVING clause can be specified after a GROUP BY clause. The HAVING clause is like a WHERE clause that can operate on groups, rather than on the full data set. HAVING and WHERE use the same comparisons. This is shown in the following example:

```
SELECT Home_State, MIN(Age) AS Youngest,
  AVG(Age) AS AvgAge, MAX(Age) AS Oldest
 FROM Sample.Person
 GROUP BY Home_State
 HAVING Age < 21
 ORDER BY Youngest
```

The following Dynamic SQL example selects table data into a result set:

```
  SET sql=7
  SET sql(1)="CREATE TABLE #mytest (MyId INT PRIMARY KEY,"
  SET sql(2)="LastName VARCHAR(20),"
  SET sql(3)="FirstName VARCHAR(20))"
  SET sql(4)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1224,'Smith','John')"
  SET sql(5)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1225,'Jones','Wilber')"
  SET sql(6)="SELECT FirstName,LastName FROM #mytest"
  SET sql(7)="DROP TABLE #mytest"
  SET statement=##class(%SQL.Statement).%New()
  SET statement.%Dialect="MSSQL"
  SET status=statement.%Prepare(.sql)
  SET result=statement.%Execute()
  DO result.%Display()
```

The following Dynamic SQL example selects a single field value into a local variable:

```
  SET sql=9
  SET sql(1)="CREATE TABLE #mytest (MyId INT PRIMARY KEY,"
  SET sql(2)="LastName VARCHAR(20),"
  SET sql(3)="FirstName VARCHAR(20))"
  SET sql(4)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1224,'Smith','John')"
  SET sql(5)="INSERT INTO #mytest(MyId,LastName,FirstName) VALUES (1225,'Jones','Wilber')"
  SET sql(6)="DECLARE @nam VARCHAR(20)"
  SET sql(7)="SELECT @nam=LastName FROM #mytest"
  SET sql(8)="PRINT @nam"
  SET sql(9)="DROP TABLE #mytest"
  SET statement=##class(%SQL.Statement).%New()
  SET statement.%Dialect="MSSQL"
  SET status=statement.%Prepare(.sql)
  DO statement.%Execute()
```

An ORDER BY clause can specify ascending (ASC) or descending (DESC) order. The default is ascending. Unlike InterSystems SQL, an ORDER BY may be used in subqueries and in queries that appear in expressions. For example:

```
SET @var = (SELECT TOP 1 name FROM mytable ORDER BY name)
```

JOIN

JOIN (equivalent to INNER JOIN), INNER JOIN, and LEFT JOIN supported. Parentheses can be used to rationalize parsing of multiple joins.

UNION

A union of two (or more) SELECT statements is supported. InterSystems TSQL supports UNION and UNION ALL. If you specify UNION ALL, only the first SELECT can specify an INTO table. This INTO table can be a defined table, or a temporary table generated from the SELECT field list.

FETCH Cursor

The OPEN, FETCH, CLOSE, and DEALLOCATE commands are mainly supported. The following features are not supported:

- OPEN/FETCH/CLOSE @local
- FETCH followed by any qualifier other than NEXT (the qualifier can be omitted).
- Note that DEALLOCATE is supported, but that, by design, it generates no code.

Flow of Control Statements

IF

Executes a block of code if a condition is true.

The IF command is supported with four syntactic forms:

IF...ELSE syntax:

```
IF condition
statement
[ELSE statement]
```

IF...THEN...ELSE single-line syntax:

```
IF condition THEN statement [ELSE statement]
```

ELSEIF...END IF syntax:

```
IF condition THEN
statements
{ELSEIF condition THEN statements}
 [ELSE statements]
END IF
```

ELSE IF (SQL Anywhere) syntax:

```
IF condition THEN statement
{ELSE IF condition THEN statement}
[ELSE statement]
```

The first syntactic form is the TSQL standard format. No THEN keyword is used. You may use white space and line breaks freely. To specify more than one statement in a clause you must use BEGIN and END keywords to demarcate the block of statements. The ELSE clause is optional. This syntax is shown in the following example:

```
  SET sql=4
  SET sql(1)="DECLARE @var INT"
  SET sql(2)="SET @var=RAND()"
  SET sql(3)="IF @var<.5 PRINT 'The Oracle says No'"
  SET sql(4)="ELSE PRINT 'The Oracle says Yes' "
  SET statement=##class(%SQL.Statement).%New()
  SET statement.%Dialect="MSSQL"
  SET status=statement.%Prepare(.sql)
  SET result=statement.%Execute()
  DO result.%Display()
```

The second syntactic form is single-line syntax. The THEN keyword is required. A line break restriction requires that IF condition THEN statement all be on the same line, though only the first keyword of the statement must be on that line. Otherwise, you may use white space and line breaks freely. To specify more than one statement in a clause you must use BEGIN and END keywords to demarcate the block of statements. The ELSE clause is optional. This syntax is shown in the following example:

```
  SET sql=3
  SET sql(1)="DECLARE @var INT "
  SET sql(2)="SET @var=RAND() "
  SET sql(3)="IF @var<.5 THEN PRINT 'No' ELSE PRINT 'Yes' "
  SET statement=##class(%SQL.Statement).%New()
  SET statement.%Dialect="MSSQL"
  SET status=statement.%Prepare(.sql)
  SET result=statement.%Execute()
  DO result.%Display()
```

The third syntactic form provides an ELSEIF clause. You can specify zero, one, or more than one ELSEIF clauses, each with its own condition test. Within an IF, ELSEIF, or ELSE clause you can specify multiple statements. BEGIN and END keywords are permitted but not required.

A line break restriction requires a line break between IF condition THEN and the first statement. Otherwise, you may use white space and line breaks freely. The ELSE clause is optional. The END IF keyword clause is required. This syntax is shown in the following example:

```
SET sql=14
SET sql(1)="DECLARE @var INT "
SET sql(2)="SET @var=RAND() "
SET sql(3)="IF @var<.2 THEN "
SET sql(4)="PRINT 'The Oracle' "
SET sql(5)="PRINT 'says No' "
SET sql(6)="ELSEIF @var<.4 THEN "
SET sql(7)="PRINT 'The Oracle' "
SET sql(8)="PRINT 'says Possibly' "
SET sql(9)="ELSEIF @var<.6 THEN "
SET sql(10)="PRINT 'The Oracle' "
SET sql(11)="PRINT 'says Probably' "
SET sql(12)="ELSE PRINT 'The Oracle' "
SET sql(13)="PRINT 'says Yes' "
SET sql(14)="END IF"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
SET result=statement.%Execute()
DO result.%Display()
```

The fourth syntactic form is compatible with SQL Anywhere. It provides an ELSE IF clause (note space between keywords). You can specify zero, one, or more than one ELSE IF clauses, each with its own condition test. To specify more than one statement in a clause you must use BEGIN and END keywords to demarcate the block of statements. You may use white space and line breaks freely. The ELSE clause is optional. This syntax is shown in the following example:

```
SET sql=6
SET sql(1)="DECLARE @var INT "
SET sql(2)="SET @var=RAND() "
SET sql(3)="IF @var<.2 THEN PRINT 'The Oracle says No'"
SET sql(4)="ELSE IF @var<.4 THEN PRINT 'The Oracle says Possibly'"
SET sql(5)="ELSE IF @var<.6 THEN PRINT 'The Oracle says Probably'"
SET sql(6)="ELSE PRINT 'The Oracle says Yes'"
SET statement=##class(%SQL.Statement).%New()
SET statement.%Dialect="MSSQL"
SET status=statement.%Prepare(.sql)
SET result=statement.%Execute()
DO result.%Display()
```

WHILE

Repeatedly executes a block of code while a condition is true.

```
WHILE condition BEGIN statements END
```

The BREAK keyword exits the WHILE loop.

The CONTINUE keyword immediately returns to the top of the WHILE loop.

The BEGIN and END keywords are required if statements is more than one command.

The following example returns four result sets, each containing a pair of records in ascending ID sequence:

```
DECLARE @n INT;
SET @n=0;
WHILE @n<8 BEGIN
      SELECT TOP 2 ID,Name FROM Sample.Person WHERE ID>@n
      SET @n=@n+2
      END;
```

CASE

Returns a value from the first match of multiple specified values.

```
CASE expression WHEN value THEN rtnval
[WHEN value2 THEN rtnval2] [...]
[ELSE rtndefault]
END
```

The WHEN value must be a simple value. It cannot be a boolean expression.

The ELSE clause is optional. If no WHEN clause is satisfied and the ELSE clause is not provided, the CASE statement returns expression as NULL.

For example:

```
SELECT CASE Name WHEN 'Fred Rogers' THEN 'Mr. Rogers'
                 WHEN 'Fred Astare' THEN 'Ginger Rogers'
                 ELSE 'Somebody Else' END
       FROM Sample.Person
```

The returned value does not have to match the data type of expression.

CASE parses but ignores WHEN NULL THEN rtnval cases.

GOTO and Labels

InterSystems TSQL supports the GOTO command and labels. A label must be a valid TSQL identifier followed by a colon (:). A GOTO reference to a label does not include the colon.

WAITFOR

Used to delay execution until a specific elapse of time or clock time.

```
WAITFOR DELAY timeperiod
WAITFOR TIME clocktime
```

timeperiod is the amount of time to wait before resuming execution, expressed as 'hh:mm[:ss[.fff]] Thus WAITFOR DELAY '00:00:03' provides a time delay of 3 seconds; WAITFOR DELAY '00:03' provides a time delay of 3 minutes; WAITFOR DELAY '00:00:00.9' provides a time delay of nine-tenths of a second. Note that the fractional second divider is a period, not a colon.

clocktime is the time at which to resume execution, expressed as 'hh:mm[:ss[.fff]], using a 24-hour clock. Thus WAITFOR TIME '14:35:00' resumes execution at 2:35pm; WAITFOR TIME '00:00:03' resumes execution at 3 seconds after midnight.

The following options are not supported:

- Sybase CHECK EVERY clause.
- Sybase AFTER MESSAGE BREAK clause.
- MSSQL RECEIVE clause.

Assignment Statements

DECLARE

Declares the data type for a local variable.

```
DECLARE @var [AS] datatype [ = initval]
```

Only the form which declares local variables is supported; cursor variables are not supported. The AS keyword is optional. Unlike InterSystems SQL, you must declare a local variable before you can set it.

@var can be any local variable name. Local variable names are not case-sensitive.

The datatype can be any valid data type, such as CHAR(12) or INT. TEXT, NTEXT, and IMAGE data types are not allowed. For further details on data types, refer to the TSQL Constructs chapter of this document.
The optional initval argument allows you to set the initial value of the local variable. You can

set it to a literal value or to any of the following: NULL, USER, CURRENT DATE (or CURRENT_DATE), CURRENT TIME (or CURRENT_TIME), CURRENT TIMESTAMP (or CURRENT_TIMESTAMP), or CURRENT_USER. The DEFAULT and CURRENT_DATABASE keywords are not supported. Alternatively, you can set the value of a local value using the SET command or the SELECT command. For example:

```
DECLARE @c INT;
SELECT @c=100;
```

You can specify multiple local variable declarations as a comma-separated list. Each declaration must have its own data type and (optionally) its own initial value:

```
DECLARE @a INT=1,@b INT=2,@c INT=3
```

SET

Assigns a value to a local variable or an environment setting.

Used to assign a value to a local variable:

```
DECLARE @var CHAR(20)
SET @var='hello world'
```

Used to set an environment setting:

```
SET option ON
```

These settings have immediate effect at parse time, whether inside a stored procedure or not. The change persists until another SET command alters it – even if the SET is made inside a stored procedure, and accessed outside the SP or in another SP.

The following SET environment settings are supported:

- SET ANSI_NULLS Permitted values are SET ANSI_NULLS ON and SET ANSI_NULLS OFF. If ANSI_NULLS OFF, a=b is true if (a=b OR (a IS NULL) AND (b IS NULL)). See the ANSI_NULLS TSQL system-wide configuration setting.
- SET DATEFIRST integer specifies which day is treated as the first day of the week. Permitted values are 1 through 7, with 1=Monday and 7=Sunday. The default is 7.
- SET IDENTITY_INSERT Permitted values are SET IDENTITY_INSERT ON and SET IDENTITY_INSERT OFF. If ON, an INSERT statement can specify an identity field value. This variable applies exclusively to the current process and cannot be set on linked tables. Therefore, to use this option you should define a procedure in TSQL to perform both the SET IDENTITY_INSERT and the INSERT, then link the procedure and execute the procedure in InterSystems IRIS via the gateway.

- SET NOCOUNT Permitted values are SET NOCOUNT ON and SET NOCOUNT OFF. When set to ON, messages indicating the number of rows affected by a query are suppressed. This can have significant performance benefits.
- SET QUOTED_IDENTIFIER Permitted values are SET QUOTED_IDENTIFIER ON and SET QUOTED_IDENTIFIER OFF. When SET QUOTED_IDENTIFIER is on, double quotes are parsed as delimiting a quoted identifier. When SET QUOTED_IDENTIFIER is off, double quotes are parsed as delimiting a string literal. The preferable delimiters for string literals are single quotes. See the QUOTED_IDENTIFIER TSQL system-wide configuration setting.
- SET ROWCOUNT Set to an integer. Affects subsequent SELECT, INSERT, UPDATE, or DELETE statements to limit the number of rows affected. In a SELECT statement, ROWCOUNT takes precedence over TOP: if ROWCOUNT is less than TOP, the ROWCOUNT number of rows is returned; if TOP is less than ROWCOUNT, the TOP number of rows is returned. ROWCOUNT remains set for the duration of the process or until you revert it to default behavior. To revert to default behavior, SET ROWCOUNT 0. If you specify a fractional value, ROWCOUNT is set to the next larger integer.
- SET TRANSACTION ISOLATION LEVEL See Transaction Statements below.

The following SET environment setting is parsed, but ignored:

    SET TEXTSIZE integer

Transaction Statements

InterSystems TSQL provides support for transactions, including named transaction names. It does not support savepoints. Distributed transactions are not supported.

SET TRANSACTION ISOLATION LEVEL

Supported for the following forms only:

- SET TRANSACTION ISOLATION LEVEL READ COMMITTED
- SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED.

READ VERIFIED and other options are not supported.

Sybase SET TRANSACTION ISOLATION LEVEL n integer option codes (0, 1, 2, 3) are not supported.

BEGIN TRANSACTION

Begins the current transaction.

```
BEGIN TRAN [name]
BEGIN TRANSACTION [name]
```

Initiates a transaction. The optional name argument can be used to specify a named transaction, also known as a savepoint. The name value must be supplied as a literal; it cannot be a variable.

You can issue multiple BEGIN TRANSACTION statements to create multiple nested transactions. You can use the @@trancount special variable to determine the current transaction level. Each transaction level must be resolved by a COMMIT statement or a ROLLBACK statement.

Note:
A Data Management Language (DML) statement that is within an explicit transaction cannot resolve an unqualified table name using a schema search path.

COMMIT TRANSACTION

Commits the current transaction.

```
COMMIT
COMMIT TRAN
COMMIT TRANSACTION
COMMIT WORK
```

These four syntactical forms are functionally identical; the COMMIT keyword, as specified below, refers to any of these syntactical forms. A COMMIT statement commits all work completed during the current transaction, resets the transaction level counter, and releases all locks established. This completes the transaction. Work committed cannot be rolled back.

If multiple BEGIN TRANSACTION statements have created nested transactions, COMMIT completes the current nested transaction. A transaction is defined as the operations since and including the BEGIN TRANSACTION statement. A COMMIT restores the transaction level counter to its state immediately prior to the BEGIN TRANSACTION statement that initialized the transaction. You can use the @@trancount special variable to determine the current transaction level.

A COMMIT cannot specify a named transaction. If you specify a transaction name as part of a COMMIT statement, the presence of this name is parsed without issuing an error, but the transaction name is not validated and it is ignored.

Sybase performs no operation and does not issue an error if a COMMIT is issued when not in a transaction.

ROLLBACK TRANSACTION

Rolls back the specified transaction or all current transactions.

```
ROLLBACK [name]
ROLLBACK TRAN [name]
ROLLBACK TRANSACTION [name]
ROLLBACK WORK [name]
```

These four syntactical forms are functionally identical; the ROLLBACK keyword, as specified below, refers to any of these syntactical forms. The optional name argument specifies a named transaction, as specified by a BEGIN TRANSACTION name statement. The name value must be supplied as a literal; it cannot be a variable.

A ROLLBACK rolls back a transaction, undoing work performed but not committed, decrementing the transaction level counter, and releasing locks. It is used to restore the database to a previous consistent state.

- A ROLLBACK rolls back all work completed during the current transaction (or series of nested transactions), resets the transaction level counter to zero and releases all locks. This restores the database to its state before the beginning of the outermost nested transaction.

- A ROLLBACK name rolls back all work done since the specified named transaction (savepoint) and decrements the transaction level counter by the number of savepoints undone. When all savepoints have been either rolled back or committed and the transaction level counter reset to zero, the transaction is completed. If the named transaction does not exist, or has already been rolled back, ROLLBACK rolls back the entire current transaction.

Sybase performs no operation and does not issue an error if a ROLLBACK is issued when not in a transaction.

SAVE TRANSACTION

The SAVE TRANSACTION [savepoint-name] statement is parsed but ignored in InterSystems TSQL. It performs no operation.

LOCK TABLE

Enables the current user to lock a table.

```
LOCK TABLE tablename IN {SHARE | EXCLUSIVE} MODE [WAIT  numsecs | NOWAIT]
```

The LOCK TABLE statement locks all of the records in the specified table. You can lock a table in SHARE MODE or in EXCLUSIVE MODE. The optional WAIT clause specifies the number of seconds to wait in attempting to acquire the table lock. The LOCK TABLE statement immediately releases any prior lock held by the current user on the specified table.

LOCK TABLE is only meaningful within a transaction. It locks the table for the duration of the current transaction. When not in a transaction, LOCK TABLE performs no operation.

Specify tablename as described in Table References. LOCK TABLE supports locking a single table; it does not support locking multiple tables.
LOCK TABLE supports SHARE and EXCLUSIVE modes; it does not support WRITE mode.

LOCK TABLE does not support the WITH HOLD clause.

WAIT time is specified as an integer number of seconds; LOCK TABLE does not support WAIT time specified as clock time.

Procedure Statements

The following standard Transact-SQL statements are supported.

CREATE PROCEDURE / CREATE FUNCTION

Creates a named executable procedure.

```
CREATE PROCEDURE procname [[@var [AS] datatype [= | DEFAULT value] [,...]] [RETURNS datatype] [AS] code
CREATE PROC procname [[@var [AS] datatype [= | DEFAULT value] [,...]] [RETURNS datatype] [AS] code
CREATE FUNCTION procname [[@var [AS] datatype [= | DEFAULT value] [,...]] [RETURNS datatype] [AS] code
```

You can return a single scalar value result from either a PROCEDURE or a FUNCTION. OUTPUT parameters and default values are also supported. These commands convert the return type from a TSQL type declaration to an InterSystems IRIS type descriptor. Currently, result sets and tables can't be returned.

Supported as either CREATE PROCEDURE or CREATE PROC. CREATE FUNCTION is very similar to CREATE PROCEDURE, but the routine type argument value is "FUNCTION", rather than "PROCEDURE".

- Any statements can be used in a CREATE FUNCTION.
- The RETURN keyword is allowed in a CREATE PROCEDURE. If a procedure completes without invoking a RETURN or RAISERROR statement, it returns an integer value of 0.

- The WITH EXECUTE keyword clause is allowed in a CREATE PROCEDURE and CREATE FUNCTION. It must appear after the RETURN keyword.

A CREATE PROCEDURE can specify a formal parameter list. Formal parameters are specified as a comma-separated list. Enclosing parentheses are optional. The AS keyword between the parameter variable and its data type is optional. Optionally, you can use the DEFAULT keyword or = symbol to assign a default value to a formal parameter; if no actual parameter value is specified, this default value is used. In TSQL an input formal parameter has no keyword indicator; an output formal parameter can be specified by the OUTPUT keyword following the data type. Alternatively, these formal parameters can be prefaced by the optional keywords IN, OUT, or INOUT.

The following example shows the creation of the procedure AvgAge with two formal parameters:

```
CREATE PROCEDURE AvgAge @min INT, @max INT
AS
BEGIN TRY
  SELECT AVG(Age) FROM Sample.Person
  WHERE Age > @min AND Age < @max
END TRY
BEGIN CATCH
  PRINT 'error!'
END CATCH
```

The following statement executes this procedure. In this case, the specified actual parameter values limit the averaging to ages 21 through 65:

```
EXEC AvgAge 20,66
```

The following example creates a procedure that returns the results of a division operation. The RETURNS keyword limits the number of decimal digits in the return value:

```
CREATE PROCEDURE SQLUser.MyDivide @a INTEGER, @b INTEGER, OUT @rtn INTEGER RETURNS
DECIMAL(2,3)
BEGIN
SET @rtn = @a / @b;
RETURN @rtn;
END
```

The following statement executes this procedure:

```
SELECT SQLUser.MyDivide(7,3)
```

The following example shows the creation of procedure OurReply:

```
CREATE PROCEDURE OurReply @var CHAR(16) DEFAULT 'No thanks' AS PRINT @var
```

When executed without a parameter, OurReply prints the default text (“No thanks”); when executed with a parameter OurReply prints the actual parameter value specified in the EXEC statement.

Note that CREATE FUNCTION and CREATE PROCEDURE cannot be issued from a stored procedure.

Importing a CREATE PROCEDURE

If imported TSQL source contains a CREATE PROC statement, then a class method containing the CREATE PROC source will be created. This class method is either placed in an existing class, or in a new class whose name is based on the schema and procedure name.

If the procedure already exists, the existing implementation is replaced. If a class matching the class name generated from the schema and procedure already exists, it is used if it was previously generated by the TSQL utility. If not, then a unique class name is generated, based on the schema and procedure name. The schema defaults to the default schema defined in the system configuration. The resulting class is compiled once the procedure has been successfully created.

If logging is requested, the source statements are logged along with the name of the containing class, class method, and the formal arguments generated. Any errors encountered by the process are also reported in the log. If errors are detected during CREATE PROC processing and a new class was generated, that class is deleted.

ALTER FUNCTION

Supported. The WITH EXECUTE keyword clause is supported.

DROP FUNCTION

Deletes a function or a comma-separated list of functions.

```
DROP FUNCTION funcname [,funcname2 [,...] ]
```

The IF EXISTS clause is not supported.

DROP PROCEDURE

Deletes a procedure or a comma-separated list of procedures.

```
DROP PROCEDURE [IF EXISTS] procname [,procname2 [,...] ]
DROP PROC [IF EXISTS] procname [,procname2 [,...] ]
```

The optional IF EXISTS clause suppresses errors if you specify a non-existent procname. If this clause is not specified, an SQLCODE -362 error is generated if you specify a non-existent procname. DROP PROCEDURE is an atomic operation; either all specified procedures are successfully deleted or none are deleted.

RETURN

Halts execution of a query or procedure. Can be argumentless or with an argument. Argumentless RETURN must be used when exiting a TRY or CATCH block. When returning from a procedure, RETURN can optionally return an integer status code. If you specify no status code, it returns the empty string ("").

EXECUTE

Executes a procedure, or executes a string of TSQL commands.

```
EXECUTE [@rtnval = ] procname [param1 [,param2 [,...] ] ]
```

```
EXECUTE ('TSQL_commands')
```

EXEC is a synonym for EXECUTE.

- EXECUTE procname can be used to execute a stored procedure. Parameters are supplied as a comma-separated list. This parameter list is not enclosed in parentheses. Named parameters are supported.

  EXECUTE procname can optionally receive a RETURN value, using the EXECUTE @rtn=Sample.MyProc param1,param2 syntax.

  EXECUTE procname is similar to the CALL statement, which can also be used to execute a stored procedure. CALL uses an entirely different syntax.

  ```
  CREATE PROCEDURE Sample.AvgAge @min INT, @max INT
    AS
    SELECT Name,Age,AVG(Age) FROM Sample.Person
    WHERE Age > @min AND Age < @max
    RETURN 99

  DECLARE @rtn INT;
  EXECUTE @rtn=Sample.AvgAge 18,65
  SELECT @rtn
  ```

  If the specified procedure does not exist, an SQLCODE -428 error (Stored procedure not found) is issued.

  The WITH RECOMPILE clause is parsed, but ignored.

  The following EXECUTE procname features are not supported: procedure variables, and procedure numbers (i.e. ';n').

- EXECUTE (TSQL commands) can be used to execute dynamic SQL. The TSQL command(s) are enclosed in parentheses. The TSQL commands to be executed are specified as a string enclosed in single quote characters. A TSQL command string can contain line breaks and white space. Dynamic TSQL runs in the current context.

  ```
  EXECUTE('SELECT TOP 4 Name,Age FROM Sample.Person')
  ```

  or

  ```
  DECLARE @DynTopSample VARCHAR(200)
  SET @DynTopSample='SELECT TOP 4 Name,Age FROM Sample.Person'
  EXECUTE (@DynTopSample)
  ```

  The following example shows an EXECUTE that returns multiple result sets:

  ```
  EXECUTE('SELECT TOP 4 Name FROM Sample.Person
          SELECT TOP 6 Age FROM Sample.Person')
  ```

CALL

Executes a procedure.

```
[@var = ] CALL procname ([param1 [,param2 [,...] ] ])
```

The CALL statement is functionally identical to the EXECUTE procname statement. It differs syntactically.

The procedure parameters are optional. The enclosing parentheses are mandatory.

The optional @var variable receives the value returned by the RETURN statement. If execution of the stored procedure does not conclude with a RETURN statement, @var is set to 0.

The following example calls a stored procedure, passing two input parameters. It receives a value from the procedure's RETURN statement:

```
DECLARE @rtn INT
@rtn=CALL Sample.AvgAge(18,34)
SELECT @rtn
```

Other Statements

CREATE USER

CREATE USER creates a new user.

```
CREATE USER username
```

Executing this statement creates an InterSystems IRIS user with its password set to the specified user name. You can then use the Management Portal System Administration interface to change the password. You cannot explicitly set a password using CREATE USER.

User names are not case-sensitive. InterSystems TSQL and InterSystems SQL both use the same set of defined user names. InterSystems IRIS issues an error message if you try to create a user that already exists.

By default, a user has no privileges. Use the GRANT command to give privileges to a user.

GRANT

Grants privileges to a user or list of users.

```
GRANT priveligelist ON tablelist TO granteelist
```

```
GRANT EXECUTE ON proclist TO granteelist
```

- privilegelist: a single privilege or a comma-separated list of privileges. The available privileges are SELECT, INSERT, DELETE, UPDATE, REFERENCES, and ALL PRIVILEGES. ALL is a synonym for ALL PRIVILEGES. The ALTER privilege is not supported directly, but is one of the privileges granted by ALL PRIVILEGES.

- tablelist: a single table name (or view name) or a comma-separated list of table names and view names. Specify a table name as described in Table References.
- proclist: a single SQL procedure or a comma-separated list of SQL procedures. All listed procedures must exist, otherwise an SQLCODE -428 error is returned.

- granteelist: a single grantee (user to be assigned privileges) or a comma-separated list of grantees. A grantee can be a user name, "PUBLIC" or "*". Specifying * grants the specified privileges to all existing users. A user created using CREATE USER initially has no privileges. Specifying a non-existent user in a comma-separated list of grantees has no effect; GRANT ignore that user and grants the specified privileges to the existing users in the list.

Specifying privileges for specified fields is not supported.

The WITH GRANT OPTION clause is parsed but ignored.

Granting a privilege to a user that already has that privilege has no effect and no error is issued.

REVOKE

Revokes granted privileges from a user or list of users.

```
REVOKE privelegelist ON tablelist FROM granteelist CASCADE
```

```
REVOKE EXECUTE ON proclist FROM granteelist
```

Revoking a privilege from a user that does not have that privilege has no effect and no error is issued.

See GRANT for further details.

PRINT

Displays the specified text to the current device.

```
PRINT expression [,expression2 [,...]]
```

An expression can be a literal string enclosed in single quotes, a number, or a variable or expression that resolves to a string or a number. You can specify any number of comma-separated expressions.

PRINT does not support the Sybase arg-list syntax. A placeholder such as %3! in an expression string is not substituted for, but is displayed as a literal.

RAISERROR

```
RAISERROR errnum 'message'
RAISERROR(error,severity,state,arg) WITH LOG
```

Both syntactic forms (with and without parentheses) are supported. Both spellings, RAISERROR and RAISEERROR, are supported and synonymous. RAISERROR sets the value of @@ERROR to the specified error number and error message and invokes the %SYSTEM.Error.FromXSQL() method.
The Sybase-compatible syntax (without parentheses) requires an errnum error number, the other arguments are optional.

```
RAISERROR 123 'this is a big error'
PRINT @@ERROR
```

A RAISERROR command raises an error condition; it is left to the user code to detect this error. However, if RAISERROR appears in the body of a TRY block, it transfers control to the paired CATCH block. If RAISERROR appears in a CATCH block it transfers control either to an outer CATCH block (if it exists) or to the procedure exit. RAISERROR does not trigger an exception outside of the procedure. It is up to the caller to check for the error.

When an AFTER statement level trigger executes a RAISEERROR, the returned %msg value contains the errnum and message values as message string components separated by a comma: %msg="errnum,message".
The Microsoft-compatible syntax (with parentheses) requires an error (either an error number or a quoted error message). If you do not specify an error number, it defaults to 50000. The optional severity and state arguments take integer values.

```
RAISERROR('this is a big error',4,1) WITH LOG
PRINT @@ERROR
```

UPDATE STATISTICS

Optimizes query access for a specified table. The specified table can be a standard table or a # temporary table (see CREATE TABLE for details.) InterSystems IRIS passes the specified table name argument to the $SYSTEM.SQL.TuneTable() method for optimization. UPDATE STATISTICS calls $SYSTEM.SQL.TuneTable() with update=1 and display=0. The returned

%msg is ignored and KeepClassUpToDate defaults to 'false'. All other UPDATE STATISTICS syntax is parsed for compatibility only and ignored. In a batch or stored procedure, only the first UPDATE STATISTICS statement for a given table generates a call to $SYSTEM.SQL.TuneTable(). For further details, see Tune Table in SQL Optimization Guide.
If the TSQL TRACE configuration option is set, the trace log file will contain records of the tables that were tuned.

USE database

Supported, also an extension: USE NONE to select no database. Effective at generation-time, persists as long as the transform object exists (e.g. in the shell or loading a batch).

InterSystems Extensions

TSQL supports a number of InterSystems extensions to Transact-SQL. To allow for the inclusion of these InterSystems-only statements in portable code, InterSystems TSQL also supports a special form of the single-line comment: two hyphens followed by a vertical bar. This operator is parsed as a comment by Transact-SQL implementations, but is parsed as an executable statement in InterSystems TSQL. For further details, refer to the Comments section of the TSQL Constructs chapter of this document.
TSQL includes the following InterSystems extensions:

OBJECTSCRIPT

This extension allows you to include ObjectScript code or InterSystems SQL code in the compiled output. It takes one or more lines of InterSystems code inside curly brackets.

The following Dynamic SQL example uses OBJECTSCRIPT because TSQL does not support the InterSystems SQL %STARTSWITH predicate:

```
SET myquery = "OBJECTSCRIPT {SELECT Name FROM Sample.Person WHERE Name %STARTSWITH 'A'}"
SET tStatement = ##class(%SQL.Statement).%New(,,"Sybase")
WRITE "language mode set to ",tStatement.%Dialect,!
SET qStatus = tStatement.%Prepare(myquery)
 IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
SET rset = tStatement.%Execute()
DO rset.%Display()
WRITE !,"End of data"
```

The following Dynamic SQL example uses OBJECTSCRIPT to include ObjectScript code in a TSQL routine:

```
  SET newtbl=2
  SET newtbl(1)="CREATE TABLE Sample.MyTest(Name VARCHAR(40),Age INTEGER)"
  SET newtbl(2)="OBJECTSCRIPT {DO $SYSTEM.SQL.TuneTable(""Sample.MyTest"") WRITE ""TuneTable
Done"",!}"
  SET tStatement = ##class(%SQL.Statement).%New(,,"Sybase")
  WRITE "language mode set to ",tStatement.%Dialect,!
  SET qStatus = tStatement.%Prepare(.newtbl)
   IF qStatus'=1 {WRITE "%Prepare failed:" DO $System.Status.DisplayError(qStatus) QUIT}
  SET rset = tStatement.%Execute()
  DO rset.%Display()
  WRITE !,"End of data"
```

Note that in the above example the WRITE command specifies a new line (,!); this is necessary because the OBJECTSCRIPT extension does not issue a new line following execution.

IMPORTASQUERY

This extension forces a stored procedure to be imported as a query rather than as a class method. This is useful for stored procedures that contain only an EXEC statement, because InterSystems IRIS cannot otherwise determine at import whether such a stored procedure is a query or not.