Transbase® SQL Reference Manual

Transbase SQL Reference Manual

Version V8.4

Publication date 2024-07-30
Copyright © 2022 Transaction Software GmbH

ALL RIGHTS RESERVED.

While every precaution has been taken in the preparation of this document, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Table of Contents

Introduction xi
1. General Concepts 1
    1.1. Syntax Notation 1
    1.2. Separators 1
    1.3. Keywords 1
    1.4. Identifiers 1
        1.4.1. User Schemas 2
        1.4.2. Names and Identifiers 3
    1.5. Data Types 4
        1.5.1. Type Compatibility 6
        1.5.2. Type Exceptions and Overflow 7
        1.5.3. CASTing Types from and to CHAR 8
    1.6. Literals 9
        1.6.1. Directory/File Literal 10
        1.6.2. IntegerLiteral 10
        1.6.3. NumericLiteral 11
        1.6.4. RealLiteral 11
        1.6.5. StringLiteral 12
        1.6.6. BinaryLiteral 12
        1.6.7. BitsLiteral 12
        1.6.8. BoolLiteral 13
        1.6.9. DATETIME Literal 13
        1.6.10. TIMESPAN Literal 13
2. Data Definition Language 14
    2.1. Dataspaces 14
        2.1.1. CreateDataspaceStatement 14
        2.1.2. AlterDataspaceStatement 14
    2.2. Users and Schemas 15
        2.2.1. Create User Statement 16
        2.2.2. Drop User Statement 16
        2.2.3. GrantUserclassStatement 17
        2.2.4. RevokeUserclassStatement 17
        2.2.5. AlterPasswordStatement 17
        2.2.6. GrantPrivilegeStatement 18
        2.2.7. RevokePrivilegeStatement 19
    2.3. Domains 20
        2.3.1. CreateDomainStatement 20
        2.3.2. AlterDomainStatement 21
        2.3.3. DropDomainStatement 21
    2.4. Sequences 22
        2.4.1. CreateSequenceStatement 22
        2.4.2. DropSequenceStatement 23
    2.5. CreateTableStatement 23
        2.5.1. Defaults 25
        2.5.2. AUTO_INCREMENT Fields 25
        2.5.3. TableConstraintDefinition FieldConstraintDefinition 26
        2.5.4. PrimaryKey 27
        2.5.5. CheckConstraint 28
        2.5.6. ForeignKey 29
    2.6. AlterTableStatement 31
        2.6.1. AlterTableConstraint 31
        2.6.2. AlterTableChangeField 31
        2.6.3. AlterTableRenameField 32
        2.6.4. AlterTableFields 32
        2.6.5. AlterTableRename 33

2.6.6. AlterTableMove 33
2.6.7. AlterTableInmemory 34
2.7. DropTableStatement 34
2.8. CreateIndexStatement 34
2.8.1. StandardIndexStatement 34
2.8.2. HyperCubeIndexStatement 36
2.8.3. FulltextIndexStatement 37
2.8.4. BitmapIndexStatement 37
2.9. DropIndexStatement 38
2.10. Triggers 38
2.10.1. CreateTriggerStatement 38
2.10.2. DropTriggerStatement 39
2.11. Views 40
2.11.1. CreateViewStatement 40
2.11.2. DropViewStatement 41
3. Data Manipulation Language 42
3.1. FieldReference 42
3.2. User 42
3.3. Expression 43
3.4. Primary, CAST Operator 43
3.5. SimplePrimary 44
3.5.1. MathematicalStdFunction 45
3.5.2. SetFunction 46
3.5.3. WindowFunction 47
3.5.4. StringFunction 49
3.5.5. TocharFunction 53
3.5.6. ResultcountExpression 54
3.5.7. SequenceExpression 54
3.5.8. ConditionalExpression 54
3.5.9. TimeExpression 57
3.5.10. SizeExpression 58
3.5.11. LobExpression 59
3.5.12. ODBC_FunctionCall 60
3.5.13. UserDefinedFunctionCall 60
3.5.14. LastInsertIdFunc 60
3.5.15. LastUpdateFunc 60
3.5.16. CrowdStatusFunc 61
3.5.17. ReplicationStatusFunc 61
3.5.18. SearchCondition 62
3.5.19. HierarchicalCondition 62
3.6. Predicate 64
3.6.1. ComparisonPredicate 64
3.6.2. ValueCompPredicate 64
3.6.3. SetCompPredicate 65
3.6.4. InPredicate 66
3.6.5. BetweenPredicate 67
3.6.6. LikePredicate 67
3.6.7. MatchesPredicate, Regular Pattern Matcher 68
3.6.8. ExistsPredicate 70
3.6.9. QuantifiedPredicate 70
3.6.10. NullPredicate 71
3.6.11. FulltextPredicate 71
3.7. Null Values 72
3.8. SelectExpression (QueryBlock) 73
3.9. TableExpression, SubTableExpression 75
3.10. TableReference, SubTableReference 77
3.11. FlatFileReference - direct processing of text files 79
3.12. CrowdExpression 80

3.12.1. Crowd Queries 80
3.12.2. Crowd Errors 81
3.13. JoinedTable (Survey) 81
3.13.1. INNER JOIN with ON/USING Clause 82
3.13.2. JoinedTable with NATURAL 83
3.13.3. JoinedTable with OUTER JOIN 83
3.14. Scope of TableReferences and CorrelationNames 86
3.15. SelectStatement 87
3.16. WithClause 87
3.17. InsertStatement 88
3.17.1. Insertion with Fieldlist and DEFAULT Values 89
3.17.2. Insertion on AUTO_INCREMENT Fields 89
3.17.3. Insertion on Views 89
3.17.4. Handling of Key Collisions 89
3.17.5. Insertion with ReturningClause 90
3.18. DeleteStatement 91
3.19. UpdateStatement 92
3.20. UpdateFromStatement 93
3.21. MergeStatement 94
3.22. General Rule for Updates 94
3.23. Rules of Resolution 95
3.23.1. Resolution of Fields 95
3.23.2. Resolution of SetFunctions 95
4. Persistent Stored Methods 97
4.1. Functions 97
4.1.1. CREATE FUNCTION Statement 97
4.1.2. DROP FUNCTION Statement 98
4.2. Procedures 98
4.2.1. CREATE PROCEDURE Statement 98
4.2.2. DROP PROCEDURE Statement 98
4.3. Usage Privileges 99
4.3.1. GRANT USAGE Statement 99
4.3.2. REVOKE USAGE Statement 99
4.4. PSM Block Statements 99
4.4.1. PSM blocks and Transactions 100
5. PSM Blocks 101
5.1. PSM Declarations 101
5.1.1. Local Method Declarations 101
5.1.2. PSM Variables 102
5.1.3. PSM cursors 105
5.2. PSM Statements 106
5.2.1. SQL Statements 106
5.2.2. IF Statements 107
5.2.3. CASE Statements 107
5.2.4. LOOP Statements 107
5.2.5. Execute Statements 110
5.3. Exception Clauses 110
5.4. Debugging and Tracing 111
5.4.1. Debug Output 111
5.4.2. Trace Output 111
6. Load and Unload Statements 112
7. Alter Session statements 113
7.1. Sort Buffer Size 113
7.2. Multithreading Mode 113
7.3. Integer Division Mode 114
7.4. Lock Mode 114
7.5. Evaluation Plans 114
7.6. Schema Default 115

7.7. SQL Dialect ... 115
8. Lock Statements ... 117
8.1. LockStatement ... 117
8.2. UnlockStatement ... 117
9. The Data Types Datetime and Timespan ... 118
9.1. Principles of Datetime ... 118
9.1.1. RangeSpec ... 118
9.1.2. SQL Compatible Subtypes ... 118
9.1.3. DatetimeLiteral ... 119
9.1.4. Valid Datetime Values ... 120
9.1.5. Creating a Table with Datetimes ... 120
9.1.6. The CURRENTDATE/SYSDATE Operator ... 121
9.1.7. Casting Datetimes ... 121
9.1.8. TRUNC Function ... 122
9.1.9. Comparison and Ordering of Datetimes ... 122
9.2. Principles of Timespan and Interval ... 123
9.2.1. Transbase Notation for Type TIMESPAN ... 123
9.2.2. INTERVAL Notation for TIMESPAN ... 124
9.2.3. Ranges of TIMESPAN Components ... 124
9.2.4. TimespanLiteral ... 125
9.2.5. Sign of Timespans ... 125
9.2.6. Creating a Table containing Timespans ... 126
9.2.7. Casting Timespans ... 126
9.2.8. Comparison and Ordering of Timespans ... 126
9.2.9. Scalar Operations on Timespan ... 127
9.2.10. Addition and Substraction of Timespans ... 127
9.3. Mixed Operations ... 128
9.3.1. Datetime + Timespan, Datetime - Timespan ... 128
9.3.2. Datetime - Datetime ... 128
9.4. The DAY Operator ... 128
9.5. The WEEKDAY Operator ... 129
9.6. The WEEK Operator ... 129
9.7. The ISOWEEK Operator ... 129
9.8. The QUARTER Operator ... 130
9.9. Selector Operators on Datetimes and Timespans ... 130
9.10. Constructor Operator for Datetimes and Timespans ... 131
10. The Datatypes BITS(p) and BITS(*) ... 133
10.1. Purpose of Bits Vectors ... 133
10.2. Creation of Tables with type BITS ... 133
10.3. Compatibility of BINCHAR and BITS ... 134
10.4. BITS and BINCHAR Literals ... 134
10.5. Spool Format for BINCHAR and BITS ... 135
10.6. Operations for Type BITS ... 135
10.6.1. Bitcomplement Operator BITNOT ... 135
10.6.2. Binary Operators BITAND , BITOR ... 135
10.6.3. Comparison Operators ... 136
10.6.4. Dynamic Construction of BITS with MAKEBIT ... 136
10.6.5. Counting Bits with COUNTBIT ... 136
10.6.6. Searching Bits with FINDBIT ... 137
10.6.7. Subranges and Single Bits with SUBRANGE ... 137
10.7. Transformation between Bits and Integer Sets ... 137
10.7.1. Compression into Bits with the SUM function ... 138
10.7.2. Expanding BITS into Record Sets with UNGROUP ... 138
11. LOB (Large Object) datatypes ... 140
11.1. The Data Type BLOB (Binary Large Object) ... 140
11.1.1. Inherent Properties of BLOBs ... 140
11.1.2. BLOBs and the Data Definition Language ... 140
11.1.3. BLOBs and the Data Manipulation Language ... 141

11.2. The Data Type CLOB (Character Large Object) 142
11.2.1. Inherent Properties of CLOBs 142
11.2.2. CLOBs and the Data Definition Language 142
11.2.3. CLOBs and the Data Manipulation Language 143
12. Fulltext Indexes 145
12.1. FulltextIndexStatement 145
12.1.1. WORDLIST and STOPWORDS 145
12.1.2. CHARMAP 146
12.1.3. DELIMITERS 146
12.1.4. WITH SOUNDEX 147
12.2. Implicit Tables of a Fulltext Index 147
12.3. FulltextPredicate 148
12.4. Examples and Restrictions 150
12.4.1. Examples for Fulltext Predicates 151
12.4.2. Restrictions for Fulltext Predicates 151
12.4.3. Phonetic Search in Fulltext Indexes 152
12.5. Performance Considerations 153
12.5.1. Search Performance 153
12.5.2. Scratch Area for Index Creation 153
12.5.3. Record Deletion 153
13. Data Import/Export 154
13.1. SpoolStatement 154
13.1.1. The DSV Spooler 155
13.1.2. The XML Spooler 160
13.1.3. The JSON Spooler 178
13.2. External data sources 181
13.2.1. Remote Database Access 181
13.2.2. FILE Tables 181
A. The Data Dictionary 183
A.1. The sysdatabase(s) Table 184
A.2. The syssession Table 184
A.3. The sysuser Table 184
A.4. The systable Table 185
A.5. The syscolumn Table 186
A.6. The sysindex Table 188
A.7. The sysview Table 189
A.8. The sysviewdep Table 190
A.9. The sysblob Table 190
A.10. The systablepriv Table 190
A.11. The syscolumnpriv Table 191
A.12. The sysdomain Table 192
A.13. The sysconstraint Table 193
A.14. The sysrefconstraint Table 193
A.15. The sysdataspace Table 194
A.16. The sysdatafile Table 195
A.17. The loadinfo Table 195
A.18. The syskeyword Table 196
A.19. The syspsmproc Table 196
A.20. The syspsmprocparam Table 197
A.21. The syspsmprocpriv Table 197
B. Sample Database 199
C. Precedence of Operators 201

List of Figures

3.1. Hierarchical data and graph 63
13.1. Example of an XML Document and the Document Tree 161
13.2. DSV spool File 162
13.3. XML Spool File 162
13.4. Complex XML spool File 163
13.5. Format Information Header 165
13.6. XML Spool File Containing Blobs 176
13.7. Output DSV Spool File 177
13.8. Output XML Spool File 177

List of Tables

1.1. Transbase Datatypes and Ranges 5
1.2. Arithmetic Data Types 7
1.3. Character Data Types 7
2.1. User Classes 15
3.1. NOT operator 72
3.2. OR operator 72
3.3. AND operator 73
9.1. SQL Types for Datetime 118
9.2. Variants of Timestamp Literals 119
9.3. Variants of Date Literals 119
9.4. Variants of Time Literals 119
9.5. Ranges of Timespan Components 125
9.6. Timespan Literals in Transbase and SQL Notation 125
13.1. Special Characters in Spool Files 158
13.2. Special Characters 161
13.3. Attributes and Their Values 170
B.1. Table SUPPLIERS 199
B.2. Table INVENTORY 199
B.3. Table QUOTATIONS 199

List of Examples

1.1. Valid Identifiers ... 2
1.2. Invalid Identifiers ... 2
2.1. Granting and Revoking Select Privileges ... 19
10.1. Construction of BITS Vectors ... 133
13.1. Spooling a file from a table with LOBs ... 159
13.2. Spooling a table with LOBs from a file ... 159
13.3. JSON spool file: Table SUPPLIERS ... 179
A.1. Table Names and Owners of Non-System Tables ... 186
A.2. Create table statement with resulting entries in syscolumn ... 187

Introduction

TB/SQL is the data retrieval, manipulation and definition language for the relational data base system Transbase. TB/SQL is an SQL implementation compatible to the ISO/DIS standard 9075 with additionally functional extensions which make the SQL language more powerful and easier to use.

This manual is intended for users who already have a basic knowledge of SQL. Heavy use of examples is made in order to clarify syntactic or semantic questions. All examples refer to a *Sample Database* outlined in the appendix of this manual.

1. General Concepts

1.1. Syntax Notation

The vertical line | separates alternatives.

Brackets `[ ]` are delimiters for an optional part.

Braces { } group several items together, e.g. to form complex alternatives. They are functionally equivalent to the standard braces `( )` as used in arithmetic expressions.

An ellipsis `...` indicates that the preceding item may be repeated arbitrarily often.

All non-terminal symbols start with an uppercase letter followed by lowercase letters.

All terminal symbols are represented by themselves without syntactic markups. All terminal symbols consisting of uppercase letters only denote keywords. These are supposed to be case-insensitive and thus be used in SQL in any possible mix of uppercase and lowercase letters.

[1]	[::= [
[2]	] ::=]

1.2. Separators

In a TB/SQL-statement, keywords, identifiers and literals must be separated from each other by at least one separator. As in many programming languages, possible separators in TB/SQL are the space character (blank), the tab character and the newline character.

In all other places separators are permitted but not needed.

1.3. Keywords

TB/SQL keywords are listed in the catalog table *syskeyword* of each database. Names that can be interpreted as keywords need to be denoted as delimited identifiers.

Keywords are case-insensitive.

All keywords shown in the following example are valid.

```
SELECT from wHerE
```

1.4. Identifiers

Syntax:

[3]	Identifier ::=	SimpleIdentifier | DelimitedIdentifier
[4]	SimpleIdentifier ::=	<sequence of ASCII characters [a-z,A-Z,_,0-9] with first character in [a-z,A-Z]>
[5]	DelimitedIdentifier ::=	"Character..." | [Character...]
[6]	Character ::=	<each printable character except double quote>

Explanation:

Identifiers serve to embed the references to database objects (users, tables, views etc.) into SQL statements. The names of these objects may be composed of more than one parts. In this case the identifiers specifying the name parts are combined with some syntactic glue (see e.g. TableIdentifier).

Each identifier unambiguously denotes a certain name, but the mapping from the identifier to the name may depend on the case-insensitive setting of the database.

- A SimpleIdentifier is a sequence of letters (*a-z*, *A-Z*, _) and digits (*0-9*), where the first character is a letter.

  It denotes a name

  - as is, if the database is 'case sensitive'. So the mapping simply is the identity transformation.
  - mapped to uppercase letters if the database is 'case insensitive'.

- A DelimitedIdentifier is any sequence of printable characters with the restriction that the final delimiter character needs to occur in pairs inside the DelimitedIdentifier.

  The transformation to a name simply removes the delimiting characters and reduces the pairwise occurrences of the final delimiter to single occurrences. No case transformation is performed regardless of the database settings.

Identifiers that could be interpreted as keywords need to be denoted as DelimitedIdentifiers.

Example 1.1. Valid Identifiers

```
suppliers
Suppno
xyz_abc
q1
q23p
"5xy"
"select"
```

Example 1.2. Invalid Identifiers

```
5xy
select
SeLecT
x:y
?x
"as"df"
```

1.4.1. User Schemas

A user schema is a container for a set of database objects.

These objects are

- Domains
- Sequences

- Tables
  - Constraints
  - Indices
  - Triggers
- Views
- Functions and Procedures

Only user names are valid schema names. These are stored in table `sysuser` of the data dictionary.

Each database object belongs to exactly one schema and can be identified by the schema name and the objects name. The object's name needs to be unique only within the schema and the object category it belongs to.

Constraints, indices, and triggers belong to the same schema as the table they belong to.

If no schema is given, the default schema of the database is used (either `PUBLIC` (default) or `USER`) with `PUBLIC` configured as fallback. The keyword `USER` represents the name of the current user.

The default schema of the database is set at database creation time and cannot be changed afterwards. By one of the ALTER SESSION statements, the default schema can be changed temporarily for a connection.

Whether a user is allowed to access a specific object only depends on the privileges of the user on the appropriate object.

A table can be moved from one schema to another by the `ALTER TABLE ... RENAME TO ...` statement. All related indices, triggers and constraints are also moved to the new schema.

The data dictionary belongs to the schema `PUBLIC`.

1.4.2. Names and Identifiers

Throughout this manual the following definitions are used:

DataspaceName: refers to the dataspace name denoted by the corresponding DataspaceIdentifier.

[7] DataspaceIdentifier ::= Identifier

UserName: refers to the user name denoted by the corresponding UserIdentifier.

SchemaName: refers to the schema name denoted by the corresponding SchemaIdentifier.

[8] UserIdentifier ::= Identifier | PUBLIC | USER

[9] SchemaIdentifier ::= UserIdentifier

DomainName: refers to the domain name denoted by the corresponding DomainIdentifier.

[10] DomainIdentifier ::= [SchemaIdentifier .] Identifier

SequenceName: refers to the sequence name denoted by the corresponding SequenceIdentifier.

[11] SequenceIdentifier ::= [SchemaIdentifier .] Identifier

TableName: refers to the table name denoted by the corresponding TableIdentifier.

[12] TableIdentifier ::= [SchemaIdentifier .] Identifier

ViewName: refers to the view name denoted by the corresponding ViewIdentifier.
[13] ViewIdentifier : : = [SchemaIdentifier .] Identifier

IndexName: refers to the index name denoted by the corresponding IndexIdentifier.
[14] IndexIdentifier : : = [SchemaIdentifier .] Identifier

TriggerName: refers to the trigger name denoted by the corresponding TriggerIdentifier.
[15] TriggerIdentifier : : = [SchemaIdentifier .] Identifier

FieldName: refers to the field name denoted by the corresponding FieldIdentifier.
[16] FieldIdentifier : : = Identifier

ConstraintName: refers to the constraint name denoted by the corresponding ConstraintIdentifier.
[17] ConstraintIdentifier : : = Identifier

CorrelationName: refers to the correlation name denoted by the corresponding CorrelationIdentifier.
[18] CorrelationIdentifier : : = Identifier

1.5. Data Types

A DataType specifies the type of a field or the target type for a type conversion.

Syntax:

```
[19]  DataType : : = TINYINT | SMALLINT | INTEGER | BIGINT |
                     NUMERIC [(Precision [,Scale])] |
                     DECIMAL [(Precision [,Scale])] |
                     FLOAT | DOUBLE | REAL |
                     VARCHAR [(Precision)] | CHAR [(Precision)] |
                     VARCHAR(*) | CHAR(*) | STRING |
                     BINCHAR [(Precision)] | BINCHAR (*) |
                     BITS (Precision) | BITS (*) |
                     BITS2 (Precision) | BITS2 (*) |
                     BOOL |
                     DATETIME Range | DATE | TIME | TIMESTAMP |
                     TIMESPAN Range |
                     INTERVAL StartIx2 [ TO EndIx2 ] |
                     BLOB | CLOB
[20]  Precision : : = IntegerLiteral
[21]  Scale : : = IntegerLiteral
[22]  Range : : = [ RangeIx1 [ : RangeIx1 ] ]
[1]   [ : : = [
[2]   ] : : = ]
[23]  RangeIx1 : : = YY | MO | DD | HH | MI | SS | MS
[24]  StartIx2 : : = RangeIx2 [ Precision ]
[25]  EndIx2 : : = RangeIx2 [ Precision ]
[26]  RangeIx2 : : = YEAR | MONTH | DAY | HOUR | MINUTE | SECOND
```

Explanation:

Each field of a table has a data type which is defined at creation time of the table. Constant values (Literals) also have a data type which is derived by the syntax and the value of the Literal.

- TINYINT, SMALLINT, INTEGER, BIGINT, NUMERIC, DECIMAL, FLOAT, DOUBLE are called the arithmetic types. DECIMAL is a synonym for NUMERIC.

  Precision in NUMERIC is the maximum total number of digits. Scale is the number of digits behind the decimal point.

  If Scale in NUMERIC is omitted, it is equivalent to 0. If Precision in NUMERIC is omitted it is equivalent to 30.

- CHAR, CHAR(p), CHAR(*), VARCHAR(p) and VARCHAR(*) are called the character types.

  CHAR is equivalent to CHAR(1). Values of type VARCHAR(p) are variable length character sequences of at most p characters. Values of type CHAR(p) are fixed sized character sequences of length p bytes. Values of type [VAR]CHAR(*) are variable length character sequences.

  No character can be the binary 0 character. In their internal representation, VARCHAR and CHAR values are ended with a binary 0 character.

  Please note that the total record size may additionally restrict these values in length when they are to be stored in the database. (see also *Constants and Sizes* [system.xhtml#system_constants_and_limits] in the *Transbase System Guide* [system.xhtml]).

  Please note that in the internal representation ASCII characters take exactly one byte, non-ASCII characters may take up to 6 bytes depending on their internal encoding which is based on the UTF8 format.

- BINCHAR, BINCHAR(p), BINCHAR(*) are called binary types.

  BINCHAR is equivalent to BINCHAR(1). Values of type BINCHAR(p) are fixed sized byte sequences of length p bytes. Values of type BINCHAR(*) are variable length byte sequences. In their internal representation, BINCHAR values have a length field.

- BITS(p), BITS2(p), BITS(*), BITS2(*) are fixed sized or variable sized bit sequences, resp. The maximum value of p (number of bits) is 31968 for both variants.

  Their internal representation resembles BINCHAR whose length is rounded up to the next multiple of 8 bits. BITS2 is more space economic than BITS because it uses a 2-byte length field in contrast to 4-byte in BITS. BITS possibly will allow a higher range in future versions.

  See *The Datatypes BITS(p) and BITS(*)*.

- BOOL is called the logical type. Its values are TRUE and FALSE. The ordering of logical values is: FALSE is less than TRUE.

- DATETIME,DATE,TIME,TIMESTAMP and TIMESPAN,INTERVAL are called the time types. They are used to describe points in time or time distances, resp. Their semantics is described in detail in a separate chapter *The Data Types Datetime and Timespan*.

- BLOB is the type for binary large objects. The type is described in a separate chapter within this manual.

- The CLOB (character large object) datatype is used for character data.

The following table provides a short summary of data types and ranges.

Table 1.1. Transbase Datatypes and Ranges

Datatype	Description	Range
TINYINT	1-byte integer	[-128 : 127]
SMALLINT	2-byte integer	[-32768 : 32767]
INTEGER	4-byte integer	[-2147483648 : 2147483647]
BIGINT	8-byte integer	[-9223372036854775808 : 9223372036854775807]

Datatype	Description	Range
NUMERIC(p,s)	exact numeric with fixed precision and scale	precision: 1 <= p <= 30, scale: -128 <= s < 127
NUMBER, NUMERIC(*,*)	numeric values with varying precision and scale	precision: 1 <= p <= 30, scale: -128 <= s < 127
FLOAT	4-byte floating point	range of IEEE float
DOUBLE	8-byte floating point	range of IEEE double
CHAR(p)	character sequence of length p bytes in internal encoding	1 <= p <= MAXSTRINGSIZE
VARCHAR(p)	character sequence of maximum length p	1 <= p <= MAXSTRINGSIZE
CHAR(*), VARCHAR(*), STRING	variable length character sequence	
BINCHAR(p)	byte sequence of fixed length p	1 <= p <= MAXSTRINGSIZE
BINCHAR(*)	variable length byte sequence	length between 0 and MAXSTRINGSIZE
BITS(p), BITS2(p)	bits sequence of fixed length p	1 <= p <= 32736
BITS(*), BITS2(*)	variable length bits sequence	length between 1 and 32736
BOOL	truth value	TRUE, FALSE
DATETIME[ub:lb]	point in time (see *sql_datetime*)	ub,lb in {YY,MO,DD,HH,MI,SS,MS}
DATE	DATETIME[YY:DD]	
TIME	DATETIME[HH:SS]	
TIMESTAMP	DATETIME[YY:MS]	
TIMESPAN[ub:lb]	time distance (see *sql_timespan*)	ub,lb in {YY,MO} or {DD,HH,MI,SS,MS}
INTERVAL	time distance	see *sql_timespan*
BLOB	binary large object	see *sql_blob*
CLOB	character large object	see *sql_clob*

The maximum length of a character sequence as a field inside a record may be restricted by the necessity that a record must always fit into a storage block. The blocksize is a database specific parameter chosen at creation time (see *Administration Guide* [admin.xhtml]). Whenever character sequences are needed which are longer than these limits permit, the data type CLOB must be used.

1.5.1. Type Compatibility

Whenever values serve as operands for an operation, their types must be compatible. The compatibility rules are as follows:

- All arithmetic types are compatible among each other.
- All binary types are compatible among each other.
- All character types are compatible among each other.
- The logical type is compatible with itself.
- The compatibilities of time types among each other and with other types is described in *The Data Types Datetime and Timespan*.

Arithmetic data types are ordered according to the following type hierarchy:

Table 1.2. Arithmetic Data Types

DOUBLE	Highest Arithmetic Type
FLOAT	
NUMERIC	
BIGINT	
INTEGER	
SMALLINT	
TINYINT	Lowest Arithmetic Type

If values of different arithmetic types are involved in an operation, they are implicitly converted to the highest type among them before the operation is performed. Upward conversion within arithmetic types never causes loss of significant digits, but note that values converted to FLOAT or DOUBLE are not always exactly representable.

Character data types are ordered according to the following type hierarchy:

Table 1.3. Character Data Types

CHAR	Highest Character Type
VARCHAR	Lowest Character Type

If two values of type VARCHAR or CHAR with different length are compared, then the shorter string is padded with the space character `' '` up to the length of the longer string.

If two character types are processed in operations UNION, INTERSECTION and DIFF then the following rules apply for determinig the result type: One participating CHAR(*) yields CHAR(*). 2 input types of CHAR or VARCHAR with precisions p1 and p2 yield output precision p = maximum(p1,p2). If at least one of them is VARCHAR - then VARCHAR(p) is the result type else CHAR(p).

For operations on type BITS see *The Datatypes BITS(p) and BITS(*)*.

1.5.2. Type Exceptions and Overflow

A type exception is the event that a value fails to be in the range of a requested type. The following operations may cause a type exception:

1. Arithmetic computation on values (addition, subtraction etc.)
2. Insertion or Update of records.
3. Explicit casting of a value to a different type (CAST-operator).

In each of these operations, the requested type is defined as follows.

In case (1) - arithmetic computation - the type of the result value is requested to be the same as that of the input operands.

The expression

```
1000000 * 2
```

is legal, because the input type is INTEGER and the result is still in the range of INTEGER.

The expression

```
1000000 * 1000000
```

leads to a type exception because the result is no more in the range of INTEGER.

To avoid this, it would be sufficent to cast one (or both) input operands to a higher ranged type e.g. NUMERIC(30,0) or BIGINT

```
1000000 CAST BIGINT * 1000000
```

or to write one operand as a NUMERIC constant

```
1000000.0 * 1000000
```

In case (2) - Insertion or Update of records - the requested types are those of the corresponding fields of the target table.

With a table T with a field f of type TINYINT, the following statement would cause a type exception:

```
INSERT INTO T (f) VALUES (200)
```

In case (3) - explicit casting - the requested type is the explicitly specified type in the CAST-operator.

The expressions

```
100 CAST SMALLINT          -- legal
'hello' CAST CHAR(10)      -- legal
```

are legal (the latter example pads the string with 5 blanks at the end).

The expressions

```
200 CAST TINYINT           -- illegal
'hello' CAST CHAR(3)       -- illegal
132.64 CAST NUMERIC(4,2)   -- illegal
```

are illegal, since they cannot be converted into the requested types because of overflow.

1.5.3. CASTing Types from and to CHAR

As described in *Type Compatibility*, there are several groups of types, namely arithmetic, character, logical and time types and the BLOB type. All types within one group are compatible among each other.

Additionally, with the exception of type BLOB, there is the possibility to convert values of each type to the type CHAR(*) and (VAR)CHAR(p). This is done by the CAST operator.

1.5.3.1. CASTing to CHAR

The main usage of casting to CHAR is to make string operations (e.g. the LIKE operator or the string concatenation operator '||') available to other types.

For example, assume that field 'birthday' of type DATETIME(YY:DD) is in table person. The following pattern search is possible:

```
SELECT * FROM person
WHERE birthday CAST CHAR(*) LIKE '%19%19%19%'
```

As another example, assume that field 'price' of type NUMERIC(6,2) is in table 'article'. The following query extracts all articles with prices ending in .99:

```
SELECT * FROM article
WHERE price CAST CHAR(*) LIKE '%.99'
```

As a further example, assume that fields 'partno1' and 'partno2' of type INTEGER are in table 'parts'. The following query constructs a composed partno of type CHAR(*) with a '/' in the middle

```
SELECT partno1 CAST CHAR(*) + '/' + partno2 CAST CHAR(*)
FROM parts WHERE ...
```

1.5.3.2. Casting from CHAR

Field values of type CHAR(*) or (VAR)CHAR(p) can be cast to any type (except BLOB) provided that the source value holds a valid literal representation of a value of the target type.

For example, assume that in a table t a field f with type CHAR(*) or VARCHAR(p) contains values of the shape xxxx where xxxx is a 4-digit number. After CASTing to INTEGER one can perform arithmetic calculations with the values.

```
SELECT f CAST INTEGER + 1 , ...
FROM t WHERE ...
```

Note that an error occurs if the source value is not a valid literal of the target type.

1.5.3.3. Implicit CASTing of CHAR to Arithmetic Types

Transbase tries to support implicit type conversions wherever it may make sense and be helpful for existing applications. CHAR literals or CHAR field values combined with arithmetic types trigger an implicit CAST operation to the specific arithmetic type. This operation succeeds if the character value is a legal literal representation of an arithmetic value. The following example combines an arithmetic field with a CHAR value which is implicitly CAST to a number:

```
SELECT '123' + t.ari, ...
FROM t WHERE ...
```

1.6. Literals

Literals are denote constant values of a certain type.

Syntax:

[27] Literal ::= IntegerLiteral | NumericLiteral | RealLiteral |
StringLiteral |
BinaryLiteral | BitsLiteral | BoolLiteral |
DatetimeLiteral | TimespanLiteral |
DirectoryLiteral | FileLiteral

Explanation:

All Literals are defined in the following paragraphs. The data types DATETIME and TIMESPAN are explained in detail in *Datetime and Timespan* .

1.6.1. Directory/File Literal

A FileLiteral denotes a filename and a directory literal denotes a directory name.

It may be a simple name or include a relative or absolute path. The exact syntax of a path may depend on the platform.

Syntax:

[28] FileLiteral : : = StringLiteral |
<any sequence of alphanumeric characters>

[29] DirectoryLiteral : : = FileLiteral

An IntegerLiteral is a non-empty sequence of digits. Note that, by definition, an IntegerLiteral is a positive number without a sign. A negative number is obtained by applying the unary minus operator to an IntegerLiteral (see section on *Expressions* below). Therefore, a separator is permitted between an unary minus and an IntegerLiteral, whereas no separators are permitted within the sequence of digits.

Each IntegerLiteral has a data type which is either INTEGER, BIGINT or NUMERIC with scale 0. The data type is derived by the value of the IntegerLiteral: if the value is inside the range of INTEGER then the type is INTEGER. If the INTEGER range is not sufficient and the value is inside the range of BIGINT then the type is BIGINT else the type is NUMERIC(p,0) where p is the number of digits of the literal.

```
5                        -- INTEGER
33000                    -- INTEGER
-33000                   -- INTEGER
1234567890123            -- BIGINT
12345678901234567890123  -- NUMERIC(23,0)
```

1.6.2. IntegerLiteral

An IntegerLiteral is the representation of a constant number without fractional part.

Syntax:

[30] IntegerLiteral : : = 0 | {1..9}[{0..9}]...

An IntegerLiteral is a non-empty sequence of digits. Note that, by definition, an IntegerLiteral is a positive number without a sign. A negative number is obtained by applying the unary minus operator to an IntegerLiteral (see section on *Expressions* below). Therefore, a separator is permitted between an unary minus and an IntegerLiteral, whereas no separators are permitted within the sequence of digits.

Each IntegerLiteral has a data type which is either INTEGER, BIGINT or NUMERIC with scale 0. The data type is derived by the value of the IntegerLiteral: if the value is inside the range of INTEGER then the type is INTEGER. If the INTEGER range is not sufficient and the value is inside the range of BIGINT then the type is BIGINT else the type is NUMERIC(p,0) where p is the number of digits of the literal.

```
5                        -- INTEGER
33000                    -- INTEGER
-33000                   -- INTEGER
1234567890123            -- BIGINT
12345678901234567890123  -- NUMERIC(23,0)
```

1.6.3. NumericLiteral

A NumericLiteral is the representation of a constant number with fractional part.

Syntax:
[31] NumericLiteral ::= IntegerLiteral . [IntegerLiteral] | . IntegerLiteral

A NumericLiteral either is an IntegerLiteral followed by a decimal point or is an IntegerLiteral followed by a decimal point and another IntegerLiteral or is an IntegerLiteral preceded by a decimal point. NumericLiteral again is a positive number, by definition.

The data type of a NumericLiteral is NUMERIC(p,s) where p is the total number of digits (without leading 0's in the non fractional part) and s is the number of digits behind the decimal point.

```
13.       -- NUMERIC(2,0)
56.013    -- NUMERIC(5,3)
 0.001    -- NUMERIC(3,3)
  .001    -- NUMERIC(3,3)
```

The last two representations are equivalent.

1.6.4. RealLiteral

A RealLiteral is the representation of a constant number with mantissa and exponent.

Syntax:
[32] RealLiteral ::= {IntegerLiteral|NumericLiteral} {e|E}[+|-]IntegerLiteral

A RealLiteral is an IntegerLiteral or a NumericLiteral, followed by 'e' or 'E', followed by an optional minus or plus sign followed by another IntegerLiteral. A RealLiteral again is a positive number, by definition.

Each RealLiteral has a data type which is FLOAT or DOUBLE. The data type is derived by the value of the RealLiteral (see table *datatypes*).

```
5.13e10      -- FLOAT
5.13e+10     -- FLOAT
0.31415e1    -- FLOAT
314.15E-2    -- FLOAT
314e-2       -- FLOAT
-  314e-2    -- FLOAT
1.2e52       -- DOUBLE
```

Note that no separators are allowed within RealLiteral, but are allowed between an eventual unary minus and a RealLiteral. The next example shows incorrect RealLiterals:

```
3.14 e4   -- illegal
3.98E -4  -- illegal
3.98e- 4  -- illegal
```

1.6.5. StringLiteral

A StringLiteral is the representation of a constant string.

Syntax:

[33] StringLiteral ::= CharacterLiteral | UnicodeLiteral | USER
[34] CharacterLiteral ::= <sequence of characters enclosed in single quotes>
[35] UnicodeLiteral ::= <0u followed by sequence of hexadecimal characters>

Explanation:

The data type of a StringLiteral is CHAR(p) where p is the size of the UTF-8 coded equivalent.

A CharacterLiteral is a (possibly empty) sequence of characters in single quotes. If a single quote is needed as character, it must be written twice, as shown in the examples.

A UnicodeLiteral is 0u followed by a number of hexadecimal characters, four per each Unicode character. The keyword `USER` denotes the name of the current user.

```
'xyz'                                     -- CHAR(3)
'string with a single quote '' inside'    -- CHAR(35)
'single quote '''                         -- CHAR(14)
''                                        -- CHAR(0)
0u006D00fc006E                            -- CHAR(4)
0ufeff                                    -- CHAR(3)
0uFC                                      -- illegal
0u00FC                                    -- CHAR(2)
```

1.6.6. BinaryLiteral

[36] BinaryLiteral ::= 0x[{0..9a..f}{0..9a..f}]...

Explanation: A BinaryLiteral is 0x followed by a (possibly empty) even number of 0, 1..9, a..f, A..F. The data type of a BinaryLiteral is BINCHAR(p) where p*2 is the number of hexadecimal characters.

```
0xA0B1C2                                  -- BINCHAR(3)
0xA0B                                     -- illegal
```

1.6.7. BitsLiteral

[37] BitsLiteral ::= 0b[0 | 1]...

see also *The TB/SQL Datatypes BITS(p) and BITS(*)*

1.6.8. BoolLiteral

A BoolLiteral is the representation of a constant boolean value.

Syntax:

[38] BoolLiteral : : = FALSE | TRUE

Boolean values are ordered: FALSE is less than TRUE.

1.6.9. DATETIME Literal

A DatetimeLiteral is the representation of a constant DATETIME value.

Syntax:

[39] DatetimeLiteral : : = DATETIME [RangeSpec] (DatetimeValue) |
DateLiteral | TimeLiteral | TimestampLiteral
[40] RangeSpec : : = [RangeQual [: RangeQual]] |
[41] RangeQual : : = YY | MO | DD | HH | MI | SS | MS
[42] DatetimeValue : : = <a substring of a value in the format yy-mm-dd hh:mi:ss.ms
as appropriate for the specified range>
[43] DateLiteral : : = DATE StringLiteral
[44] TimeLiteral : : = TIME StringLiteral
[45] TimestampLiteral : : = TIMESTAMP StringLiteral

1.6.10. TIMESPAN Literal

A TimespanLiteral is the representation of a constant TIMESPAN value.

Syntax:

[46] TimespanLiteral : : = [-] TIMESPAN [RangeSpec] (DatetimeValue) | IntervalLiteral
[47] IntervalLiteral : : = <a value in the format INTERVAL value range TO range >

2. Data Definition Language

The Data Definition Language (DDL) portion of TB/SQL serves to create, delete or modify database objects as e.g. tables, views and indexes, to grant or revoke user privileges and to install users and passwords. In the following each DDL statements is explained in its own section.

2.1. Dataspaces

2.1.1. CreateDataspaceStatement

Serves to create a user dataspace in the database.

Syntax:

```
[48]    CreateDataspaceStatement ::= CREATE DATASPACE DataspaceIdentifier
                                     [ DATAFILE IN DirectoryLiteral ]
                                     [ SIZE SizeSpec ]
                                     [ AUTOEXTEND SizeSpec ]
                                     [ MAXSIZE SizeSpec ]
[49]                    SizeSpec ::= IntegerLiteral { MB | GB }
```

Explanation: The CreateDataspaceStatement creates a user defined dataspace with the given name.

Initially the dataspace consists of one file with the specified size which is created in the specified directory which must exist. If no directory is specified, the file is created in the "disks" directory inside the database home directory.

The creation of each table optionally may be specified with a user dataspace name. All pages of the table as well as the pages of its BLOB container and of all secondary are allocated in the files belonging to this dataspace.

If a dataspace runs out of space, another file may be added with the ALTER DATASPACE statement. If the AUTOEXTEND option has been specified, a full dataspace is automatically extended by another file with the specified size.

With the MAXSIZE specification the size of the dataspace is limited to the specified size, irrespective whether AUTOEXTEND has been specified or not.

At creation time of a database exactly one dataspace with name `dspace0` exists. All system catalogue tables are stored in dspace0. Furthermore, all tables without explicit DATASPACE clause are also stored there.

Creation of a dataspace is always committed even if the corresponding transaction is aborted.

```
CREATE DATASPACE dspace1 DATAFILE IN /usr/dbusr/dspace1 SIZE 100 MB
CREATE DATASPACE dspace2  SIZE 100MB  AUTOEXTEND 50MB MAXSIZE 800MB

CREATE TABLE T DATASPACE dspace1 (t0 INTEGER, t1 CHAR(*))
```

2.1.2. AlterDataspaceStatement

Serves to alter a dataspace.

Syntax:

[50]	AlterDataspaceStatement ::=	ALTER DATASPACE DataspaceIdentifier { AddFileSpec | StateSpec... }
[51]	StateSpec ::=	OnlineSpec | AutoextendSpec | MaxsizeSpec
[52]	AddFileSpec ::=	ADD DATAFILE [LOBCONTAINER] [IN DirectoryLiteral] SIZE SizeSpec
[53]	OnlineSpec ::=	ONLINE | OFFLINE
[54]	AutoextendSpec ::=	AUTOEXTEND { SizeSpec | OFF }
[55]	MaxsizeSpec ::=	MAXSIZE { SizeSpec | OFF }
[49]	SizeSpec ::=	IntegerLiteral { MB | GB }

Explanation:

The AlterDataspaceStatement is used to alter the properties of a dataspace.

A dataspace may be set offline and reset to online.

A further file may be added to an existing dataspace. If no directory name is specified, the new file is placed into the same directory as specified in the corresponding CREATE DATASPACE statement. If LOBCONTAINER is specified then the file is dedicated for BLOBs and CLOBs, i.e. no other data than BLOBs and CLOBS are ever stored into that datafile.

The AUTOEXTEND property of the dataspace may be included, or changed or dropped.

For a MAXSIZE property of the dataspace, there is the restriction that a new MAXSIZE can only be set if there was already a MAXSIZE property, and the'new value'must not be smaller than the existing. The OFF specification drops a currently existing MAXSIZE limit.

Caution

Alter of a dataspace is always committed even if the corresponding transaction is aborted.

```
ALTER DATASPACE dspace1 ADD DATAFILE SIZE 100MB
ALTER DATASPACE dspace2 ADD DATAFILE IN /dbs/db0/diskfiles SIZE 100MB
ALTER DATASPACE dspace3 OFFLINE
ALTER DATASPACE dspace4 ONLINE AUTOEXTEND 50mb MAXSIZE OFF
```

2.2. Users and Schemas

User accounts serve two purposes in Transbase datatabases.

- Each user accout constitutes a corresponding `schema` that can hold different database objects like e.g. tables, views or sequences. Objects in this schema can be identified by a combination of the schema name and the object's name within this schema. Therefore it makes sense to define user accounts that cannot be used as logins just to define separate name spaces for groups of database objects.
- As logins they provide access to the database. The kind of available access is defined by the *user class* associated with each user. Access to database objects now owned by this user is further limited by the rights granted to this user on particular database objects.

Table 2.1. User Classes

Userclass	Level	Description
DBA	3	All rights. The user can create or remove other user accounts and create database objects and change the definition of database objects regardless of being the owner. The user can

Userclass	Lev-el	Description
		also access any database object and change its contents regardless of the privileges on that object.
RESOURCE	2	The user can create database objects and change the definition and contents of own database objects. The user can also grant and revoke privileges that control access to these objects.
ACCESS	1	The user can access arbitrary database objects according to the privileges granted. The user cannot create or change the definition of own database objects. Such objects may, however, exist as the user may have had a higher userclass before.
NO_ACCESS	0	The user cannot login to the database.

2.2.1. Create User Statement

Serves to install a new user. The user gets user class ACCESS.

Syntax:
[56] CreateUserStatement ::= CREATE USER UserIdentifier

Explanation: If the specified user is not yet installed then the statement installs the user and sets its userclass to ACCESS.

```
CREATE USER jim
```

Installs the user jim with userclass ACCESS. The operation fails if the user jim already exists regardless of his userclass.

Privileges: The current user must have userclass DBA.

2.2.2. Drop User Statement

Serves to remove a user from the list of database users.

Syntax:
[57] DropUserStatement ::= DROP USER UserIdentifier [CASCADE]

Explanation: Removes the specified user from the list of users in this database.

```
DROP USER jim
DROP USER charly CASCADE
```

- If CASCADE is specified, all database objects owned by the user are also deleted. (domain information used by other tables is expanded like in DROP DOMAIN .. CASCADE).

  All objects in the user's schema are deleted, too.

- If CASCADE is not specified, all tables and domains owned by the user remain existent and their ownership is transferred to tbadmin.

  The operation fails if the schema of this user is not empty.

Privileges: The current user must have userclass DBA.

2.2.3. GrantUserclassStatement

Serves to install a new user or to raise the userclass of an installed user.

Syntax:

[58]	GrantUserclassStatement	::=	GRANT Userclass TO UserIdentifier
[59]	Userclass	::=	ACCESS | RESOURCE | DBA

Explanation: If the specified user is not yet installed then the statement installs the user and sets its userclass to the specified one.

If the specified user is already installed then the statement raises its userclass to the specified one. In this case the specified userclass must be of higher level than the user's current userclass.

```
GRANT RESOURCE TO charly
GRANT ACCESS TO jim
```

The userclass of a user defines the permission to login to the database and to create objects (tables, views, indexes). The special userclass DBA serves to grant the privileges of a superuser who has all rights.

Privileges: The current user must have userclass DBA.

2.2.4. RevokeUserclassStatement

Serves to lower the userclass of an installed user or to disable a user from login.

Syntax:

[60]	RevokeUserclassStatement	::=	REVOKE ACCESS FROM UserIdentifier | REVOKE RESOURCE FROM UserIdentifier | REVOKE DBA FROM UserIdentifier

Explanation: The specified user must be an installed user with a userclass which is not lower than the specified one. The statement lowers the userclass of the user to the level immediately below the specified userclass.

```
REVOKE RESOURCE FROM charly  -- now has userclass ACCESS
REVOKE ACCESS FROM jim       -- now has userclass NO_ACCESS
```

In particular, when userclass ACCESS is revoked the user cannot login to the database anymore.

Privileges: The current user must have userclass DBA.

2.2.5. AlterPasswordStatement

Serves to install or change a password.

Syntax:

[61]	AlterPasswordStatement ::=	ALTER PASSWORD FROM Oldpassword TO Newpassword
[62]	Oldpassword ::=	StringLiteral
[63]	Newpassword ::=	StringLiteral

Explanation: Oldpassword must match the password of the current user. The password is changed to Newpassword.

After installing a user the password is initialized to the empty string.

```
ALTER PASSWORD FROM '' TO 'xyz'
ALTER PASSWORD FROM 'xyz' TO 'acb'
```

2.2.6. GrantPrivilegeStatement

Serves to transfer privileges to other users.

Syntax:

[64]	GrantPrivilegeStatement ::=	GRANT Privileges ON TableIdentifier TO UserList [WITH GRANT OPTION]
[65]	Privileges ::=	ALL [PRIVILEGES] | Privilege [, Privilege]
[66]	Privilege ::=	SELECT | INSERT | DELETE | UPDATE [(FieldList)]
[67]	UserList ::=	UserIdentifier [, UserIdentifier]
[8]	UserIdentifier ::=	Identifier | PUBLIC | USER

Explanation: The specified privileges on the table are granted to the specified user.

If the WITH GRANT OPTION is specified, the privileges are grantable, i.e. the users get the right to further grant the privileges, otherwise not.

The table may be a base table or a view.

The variant `ALL` is equivalent to a PrivilegeList where all four Privileges are specified and where all FieldNames of the table are specified in the UPDATE- privilege.

A missing FieldList is equivalent to one with all fields of the table.

If PUBLIC is specified then the privileges are granted to all users (new users also inherit these privileges).

Note

A privilege can be granted both to a specific user and to PUBLIC at the same time. To effectively remove the privilege from the user, both grantings must be revoked.

Tip

UpdatePrivileges can be granted on field level whereas SELECT-privileges cannot. To achieve that effect, however, it is sufficient to create an appropriate view and to grant SELECT-privilege on it.

Privileges: The current user must have userclass DBA or must have all specified privileges with the right to grant them.

```
GRANT SELECT, UPDATE (price, qonorder)
    ON quotations TO jim,john
```

```
GRANT SELECT ON suppliers
    TO mary WITH GRANT OPTION
```

2.2.7. RevokePrivilegeStatement

Serves to revoke privileges which have been granted to other users.

Syntax:

[68]	RevokePrivilegeStatement	::=	REVOKE Privileges ON TableIdentifier FROM UserList
[65]	Privileges	::=	ALL [PRIVILEGES] | Privilege [, Privilege]
[66]	Privilege	::=	SELECT | INSERT | DELETE | UPDATE [(FieldList)]
[67]	UserList	::=	UserIdentifier [, UserIdentifier]
[8]	UserIdentifier	::=	Identifier | PUBLIC | USER

Explanation: If the current user is owner of the table, then the specified privileges are removed from the user such that none of the privileges are left for the user.

If the current user is not owner of the table, then the privilege instances granted by the current user are removed from the specified users. If some identical privileges had been additionally granted by other users, they remain in effect (see Example).

If a SELECT privilege is revoked then all views which depend on the specified table and are not owned by the current user are dropped. If an INSERT or UPDATE or DELETE privilege is revoked then all views which depend on the specified table and are not owned by the current user and are updatable are dropped.

Note

It is not an error to REVOKE privileges from a user which had not been granted to the user. This case is simply treated as an operation with no effect. This enables an error-free REVOKING for the user without keeping track of the granting history.

Example 2.1. Granting and Revoking Select Privileges

User jim:

```
CREATE TABLE jimtable ...
GRANT SELECT ON jimtable TO mary, anne WITH GRANT OPTION
```

User mary:

```
GRANT SELECT ON jimtable TO john
```

User anne:

```
GRANT SELECT ON jimtable TO john
```

If owner jim then says:

```
REVOKE SELECT ON jimtable FROM john
```

then john looses SELECT-privilege on jimtable

If, however, anne or mary (but not both) say:

```
REVOKE SELECT ON jimtable FROM john
```

then john still has SELECT-privilege on jimtable.

2.3. Domains

2.3.1. CreateDomainStatement

Serves to create a domain in the database. A domain is a named type, optionally with default value and integrity constraints.

Domains can be used in CreateTableStatements (for type specifications of fields) and as target type in CAST expressions.

Syntax:

[69]	CreateDomainStatement ::=	CREATE DOMAIN DomainIdentifier [AS] DataType [DEFAULT Expression] [DomainConstraint] ...
[70]	DomainConstraint ::=	[CONSTRAINT ConstraintIdentifier] CHECK (SearchCondition)

Explanation: The CreateDomainStatement creates a domain with the specified *domain name* and with the specified DataType.

If a DEFAULT specification is given then the created domain has the specified value as its default value else the domain has no default value. See the *CreateTableStatement* for the default mechanism of fields.

The default expression must not contain any subqueries or field references.

If DomainConstraints are specified then all values of the specified domains are subject to the specified search conditions, e.g. if a record is inserted or updated in a table and a field of the table is defined on a domain, then the field value is checked against all domain constraints specified on the domain.

The search condition in DomainConstraint must not contain any subqueries or field references. The keyword VALUE is used to describe domain values in the search conditions.

For the check to be performed, the formal variable VALUE in the search condition is consistently replaced by the field value. The integrity condition is violated, if and only if the expression *NOT (SearchCondition)* evaluates to TRUE.

Note

The definition of check constraints is such that NULL values pass the check in most simple cases. For all examples above, a NULL value yields the result "unknown" for the search condition, thus the negation NOT(..) also yields unknown (and thus the constraint is not violated).

To achieve that a NULL value violates an integrity constraint, the constraint must be formulated like

```
CHECK ( VALUE IS NOT NULL AND ... )
```

Whenever a domain constraint is violated, Transbase issues an error message which contains the `ConstraintName`. This may be an internally generated name if no `ConstraintName` was specified for the constraint. It is therefore recommended to specify explicit constraint names.

Example:

```
CREATE DOMAIN percentage AS NUMERIC(5,2)
    CONSTRAINT range100 CHECK (VALUE BETWEEN 0 AND 100)
```

The current user becomes owner of the domain.

Privileges: The user must have userclass DBA or RESOURCE.

Catalog Tables: For each domain, at least one entry into the table *sysdomain* is made. This entry also contains a DomainConstraint if specified. For each further specified DomainConstraint, one additional entry is made.

2.3.2. AlterDomainStatement

Serves to alter a domain in the database, i.e. to set or drop a default, to add or remove Check Constraints.

Syntax:

[71]	AlterDomainStatement	::=	ALTER DOMAIN DomainIdentifier AlterDomainSpec
[72]	AlterDomainSpec	::=	SET DEFAULT Expression | DROP DEFAULT | ADD DomainConstraint | DROP CONSTRAINT ConstraintIdentifier

Explanation: Note that no field values in the database are changed by any of these statements.

- *`SET DEFAULT`* sets the default of the domain to the specified value.
- *`DROP DEFAULT`* drops the default value of the domain.
- *`ADD DomainConstraint`* adds the specified domain constraint to the domain. All table fields based on the domain are checked whether they fulfill the new constraint and the statement is rejected if there are any violations against the new constraint.
- *`DROP CONSTRAINT ConstraintIdentifier`* drops the specified domain constraint from the domain.

Privileges: The user must be owner of the domain.

```
ALTER DOMAIN SupplierID SET DEFAULT -1
ALTER DOMAIN SupplierID DROP DEFAULT
ALTER DOMAIN UnitPrice DROP CONSTRAINT price100
ALTER DOMAIN UnitPrice ADD CONSTRAINT price200
    CHECK (VALUE between 0 AND 200)
```

2.3.3. DropDomainStatement

Serves to remove a domain from the database.

Syntax:

[73]	DropDomainStatement	::=	DROP DOMAIN DomainIdentifier DropBehaviour
[74]	DropBehaviour	::=	RESTRICT | CASCADE

Explanation: The statement removes the specified domain from the database.

If RESTRICT is specified, the statement is rejected if any field of an existing table is based on the domain or if the domain is used in a CAST expression of any view definition.

If CASCADE is specified, the domain is removed also in the cases where the RESTRICT variant would fail. For all table fields based on the domain, the domain constraints (if any) are integrated as table constraints into the table definitions. The domain default (if any) is integrated as field default unless the field has been specified with an explicit DEFAULT value at table definition time.

Note

The semantics of a DROP … CASCADE is such that the integrity constraints defined via the domain (if any) effectively are not lost.

Privileges: The user must be owner of the domain.

```
DROP DOMAIN SupplierID CASCADE
```

2.4. Sequences

2.4.1. CreateSequenceStatement

Creates a sequence.

Syntax:

[75]	CreateSequenceStatement	: : = CREATE SEQUENCE SequenceIdentifier [DataspaceSpec] [StartSpec] [IncrSpec] [MaxSpec] [CYCLE]
[88]	DataspaceSpec	: : = DATASPACE DataspaceIdentifier
[76]	StartSpec	: : = START [WITH] IntegerLiteral]
[77]	IncrSpec	: : = INCREMENT [BY] IntegerLiteral]
[78]	MaxSpec	: : = MAXVALUE IntegerLiteral]

Explanation: Creates a sequence with the specified name. A sequence is an object which can be used to generate increasing numbers of type Bigint. These numbers are unique even if generated by concurrent transactions. Also no lock conflicts arise due to the use of sequences.

For a sequence S, there are 2 operations available namely S.nextval and S.currval.

The first S.nextval operation delivers the value specified in StartSpec or 1 as default. Each S.nextval increases the value of S by the value specified in IncrSpec or 1 as default. If a MaxSpec has been given and the nextval operation would generate a value beyond the maximal value then either an error is generated or (if CYCLE has been specified) the startvalue again is delivered as next value. S.nextval also is permitted as default specification for a field of a table.

The S.currval operation is only allowed if there has been a S.nextval operation within the same transaction and again delivers the last value delivered by S.nextval. S.currval does not increase the current value of S.

Privileges: To create a sequence, the current user must have userclass DBA or RESOURCE.

For nextval the user must have UPDATE privilege on the sequence.

For currval the user must have SELECT privilege on the sequence.

Privileges on sequences are granted and revoked like those on tables.

```
CREATE SEQUENCE S START 1 INCREMENT 2
```

2.4.2. DropSequenceStatement

Drops a sequence.

Syntax:

[79]	DropSequenceStatement	: : = DROP SEQUENCE SequenceIdentifier

Explanation: Drops the sequence with the specified name.

Privileges: The current user must be owner of the sequence.

```
DROP SEQUENCE S
```

2.5. CreateTableStatement

Serves to create a table in the database.

Syntax:

[80]	CreateTableStatement	: : = StdTableStatement | FlatTableStatement | FileTableStatement
[81]	StdTableStatement	: : = CREATE TABLE [IF NOT EXISTS] TableIdentifier [IkSpec] [DataspaceSpec] [InmemorySpec] (TableElem [, TableElem] ...) [KeySpec]
[82]	FlatTableStatement	: : = CREATE FLAT [FtSizeSpec] TABLE [IF NOT EXISTS] TableIdentifier [IkSpec] (TableElem [, TableElem]...)
[83]	FtSizeSpec	: : = (IntegerLiteral [KB | MB])
[84]	FileTableStatement	: : = CREATE FILE (FileLiteral [CodePageSpec] [NullSpec] [DelimSpec]) TABLE [IF NOT EXISTS] TableIdentifier (FieldDefinition [, FieldDefinition]...)
[85]	CodePageSpec	: : = CODEPAGE [IS] CodePage [[WITH | WITHOUT] PROLOGUE]]
[86]	CodePage	: : = UTF8 | UCS | UCS2 | UCS4 | UCS2LE | UCS2BE | UCS4LE | UCS4BE
[87]	IkSpec	: : = { WITH | WITHOUT } IKACCESS
[88]	DataspaceSpec	: : = DATASPACE DataspaceIdentifier
[89]	InmemorySpec	: : = INMEMORY [PRELOAD | NONE]
[90]	TableElem	: : = FieldDefinition | TableConstraintDefinition
[91]	FieldDefinition	: : = FieldIdentifier DataTypeSpec [DefaultClause | AUTO_INCREMENT] [FieldConstraintDefinition]...
[92]	DataTypeSpec	: : = DataType | DomainIdentifier
[93]	DefaultClause	: : = DEFAULT Expression
[94]	KeySpec	: : = StdKeySpec | HCKeySpec
[95]	StdKeySpec	: : = KEY IS FieldList
[96]	HCKeySpec	: : = HCKEY [NOT UNIQUE] IS FieldList
[97]	PrimaryKeySpec	: : = StdPrimaryKeySpec | HCPrimaryKeySpec
[98]	StdPrimaryKeySpec	: : = PRIMARY KEY (FieldList)
[99]	HCPrimaryKeySpec	: : = PRIMARY HCKEY [NOT UNIQUE] (FieldList)

[100] FieldList : : = FieldIdentifier [, FieldIdentifier]...

Explanation: The CreateTableStatement creates a table with the given `TableName`.

The StdTableStatement creates a table as a B-tree. Therefore its data is stored clustered (sorted) along its primary key specification. This allows efficient lookup of data via the primary key. On the other hand, insertions into sorted data are complex and therefore costly.

The FlatTableStatement creates a table without primary key and without clustering. In contrast to standard tables, data is stored in input order. This allow faster data insertion as data is always appended. Via its SizeSpec the table can be restricted to occupy no more than a certain maximum of space. If this maximum is exceeded, the oldest data will automatically be replaced. Thus *Flat Tables* [system.xhtml#FlatTables] are ideally suited for data staging during bulk load processes, as temporary storage and for logging facilities.

The FileTableStatement allows spool files or other compatible file formats to be integrated into the database schema as virtual tables. These *FILE* tables offer read-only access to those files via SQL commands. They can be used throughout SQL *SELECT* statements like any other base table. The table definition supplies a mapping of columns in the external file to column names and Transbase datatypes. No key specifications are allowed on a File table. The creation of secondary indexes is not possible. For details on the optional parameters CodePageSpec, NullSpec and DelimSpec please consult the SpoolTableStatement. FILE tables are primary designed as an advanced instrument for bulk loading data into Transbase and applying arbitrary SQL transformations at the same time.

An error is returned if a table with the same name already exists unless the `IF NOT EXISTS` option is specified. In the latter case no action is performed and no error is returned.

The IkSpec adjusts whether to create a table with or without internal key (IK) access path. IKs are used as row identifier, e.g. for referencing records in the base table after accessing secondary indexes. This IK access path requires additional space of 6 to 8 bytes per record. Alternatively Transbase can use the primary key access path. In this case the base table's primary key is stored in all index records for referencing the base table. Depending on how extensive the primary key is, Transbase will automatically decide at table creation time whether to create a table *WITH* or `WITHOUT IKACCESS`. This guarantees optimal space efficiency. If the primary key occupies no more that the IK, then the table is created *WITHOUT IKACCESS*. Else an IK access path is added by default.

The InmemorySpec allows to specify that a table resides in the main memory cache of the database and will never be swapped out. Specification of NONE is the default (can be used in the AlterTableStatement to reset Inmemory property of the table). If the PRELOAD option is chosen, the table is loaded completely at boottime, else it appears in the cache one by one on a database page basis as it is accessed (the accessed pages then remaining in the cache). The InmemorySpec refers to the table itself and all its secondary indexes, but not on LOB objects possibly belonging to the table. The Inmemory property of a table can be changed with the AlterTableStatement.

Secondary indexes can also be created on *Flat Tables* [system.xhtml#FlatTables]. As these tables do not have a primary key, secondary indexes are only possible on *Flat Tables* [system.xhtml#FlatTables] *WITH IKACCESS*. Typically such secondary indexes are added once the load process is complete, so load performance is not compromised by secondary index maintenance.

It is always possible to override this default mechanism of IkSpec by adding *WITH* or *WITHOUT IKACCESS* to the create table statement.

Each FieldDefinition specifies a field of the table. The ordering of fields is relevant for the *-notation in *SELECT * FROM ...*.

TableConstraintDefinition and FieldConstraintDefinition are explained in the subsequent chapters.

The CreateTableStatement creates a table with the given `TableName`.

If the key specification is omitted, the combination of all fields (except BLOB and CLOB fields)implicitly is the key. No column of type BLOB or CLOB is allowed to be part of an explicitly specified key.

Unless a NOT UNIQUE specification is given, insert and update operations which produce records with the same values on all key fields are rejected.

The "KEY IS .." specification creates a table with a compound B-tree index.

The "HCKEY IS .." specification creates a table with a HyperCube index. Key fields of a HyperCube table are restricted to exact arithmetic types (BIGINT, INTEGER, SMALLINT, TINYINT, NUMERIC). If NOT UNIQUE is specified, then also duplicates on the key combination are allowed. NOT UNIQUE, however, is restricted to HyperCube tables. On each HyperCube key field a NOT NULL constraint and a *CheckConstraint* must exist.

Note

If there exists one field (or one or more field combinations) which is known to be unique in the table, it is strongly recommended to explicitly specify it as key of the table. One advantage is that uniqueness is guaranteed; another advantage is much better performance in update operations (which normally do not change key values).

The current user becomes owner of the table and gets SELECT-privilege, INSERT-privilege, DELETE-privilege on the table and UPDATE-privilege on all fields of the table. All privileges are grantable.

Privileges: The user must have userclass DBA or RESOURCE.

```
CREATE TABLE quotations
(   suppno        INTEGER       DEFAULT -1 NOT NULL,
    partno        INTEGER       DEFAULT -1 NOT NULL,
    price         NUMERIC (6,2) DEFAULT 0  NOT NULL,
    delivery_time INTEGER,
    qonorder      NUMERIC (4)
)
KEY IS suppno, partno

CREATE TABLE geopoints
(   info          INTEGER       NOT NULL,
    longitude     NUMERIC(10,7) NOT NULL
      CHECK(longitude BETWEEN -180 AND 180),
    latitude      NUMERIC(9,7)  NOT NULL
      CHECK(latitude BETWEEN -90 AND 90)
)
HCKEY IS longitude, latitude
```

2.5.1. Defaults

Each field has a (explicitly specified or implicit) default value which is taken as input value if a field value is not explicitly specified in an INSERT statement (or by an INSERT via a view). If a DEFAULT clause is specified with an expression evaluating to d, then d is the (explicitly specified) default value for that field, otherwise, if the field is based on a domain with explicit default value d, then d is the default value, otherwise NULL is the default value.

In the DEFAULT Expression, neither field references nor subqueries are allowed.

2.5.2. AUTO_INCREMENT Fields

An AUTO_INCREMENT field serves to generate unique key values. At most one AUTO_INCREMENT field is allowed in a table. Its data type must be one of TINYINT, SMALLINT, INTEGER, BIGINT.

An AUTO_INCREMENT field always must be specified either as the (single) PRIMARY KEY field or as the lowest weighted component of a compound primary key.

```
CREATE TABLE T
(   Id    BIGINT AUTO_INCREMENT,
    Prop1   VARCHAR(*),
    Prop2   VARCHAR(*),
    PRIMARY KEY (Id)
)
```

In table T, Id is the only key field. For each INSERT statement which does not assign a value for Id but uses a fieldlist (Prop1,Prop2), a unique value for the field Id automatically is generated. Value generation starts with the value 1. See below how the assigned value can be transferred to the application.

An AUTO_INCREMENT field may be explicitly assigned a value, too. For example, if the first INSERT statement for table T is the following:

```
INSERT INTO T VALUES(10, 'Foo1', 'Foo2')
```

then the next generated value would be 11. Automatic value generation always takes the maximum value so far plus one. At definition time of the table,a start value for the automatic numbering may be specified:

```
CREATE TABLE person
(   FirstName     VARCHAR(*)       NOT NULL,
    SecondName    VARCHAR(*)       NOT NULL,
    Id    INTEGER AUTO_INCREMENT=10,
    Birthday   Date,
    ....
    PRIMARY KEY (FirstName, SecondName, Id)
)
```

Here, Id is used to generate unique key combinations among identical pairs of FirstName, SecondName. In contrast to the usage of as SEQUENCE, the AUTO_INCREMENT fields starts numbering each set of identical pairs beginning with the specified number (default is number 1). Whenever a record is inserted with a pair of names which already exists, the numbering is done with 1 + maximum of numbers for that pair.

As in the example above, an explicit numbering again is allowed as long as no key collision is produced.

Note that an AUTO_INCREMENT field may overflow which results in an error. So the data type should be chosen appropriately.

2.5.2.1. Processing implicitly assigned AUTO_INCREMENT values

There are 2 ways for the application to get the implicitly assigned value of an AUTO_INCREMENT field. The RETURNING clause is useful in these cases to see the value which has been assigned.

```
INSERT INTO T (Prop1,Prop2) VALUES( 'Foo1', 'Foo2') RETURNING(Id)
```

An alternative is the function `LAST_INSERT_ID()`. Used as an expression in the SELECT list of a query, it delivers the most recently implicitly assigned value to an AUTO_INCREMENT field.

2.5.3. TableConstraintDefinition FieldConstraintDefinition

Overview syntax for specification of integrity constraints in a CreateTableStatement.

Syntax:
[101]   TableConstraintDefinition ::= [CONSTRAINT ConstraintIdentifier] TableConstraint

[102] FieldConstraintDefinition : : = [CONSTRAINT ConstraintIdentifier] FieldConstraint
[103] TableConstraint : : = PrimaryKeySpec | CheckConstraint | ForeignKey
[104] CheckConstraint : : = CHECK (SearchCondition)
[105] ForeignKey : : = FOREIGN KEY (FieldList) ReferencesDef
[106] ReferencesDef : : = REFERENCES TableIdentifier [(FieldList)] [ON DELETE Action] [ON UPDATE Action]
[107] Action : : = NO ACTION | CASCADE | SET DEFAULT | SET NULL
[108] FieldConstraint : : = NOT NULL | PRIMARY KEY | CheckConstraint | ReferencesDef

Explanation: Explanations are given in the subsequent sections.

The construct FieldConstraint is subsumed by the more general TableConstraint. In certain special cases, the syntactic variant FieldConstraint allows a more compact notation for a TableConstraint. There are no performance differences with the 2 notations.

Note

All constraints are effectively checked after execution of each SQL query.

2.5.4. PrimaryKey

Specify the main key for a table.

Syntax:
[108] FieldConstraint : : = NOT NULL | PRIMARY KEY | CheckConstraint | ReferencesDef
[103] TableConstraint : : = PrimaryKeySpec | CheckConstraint | ForeignKey
[97] PrimaryKeySpec : : = StdPrimaryKeySpec | HCPrimaryKeySpec
[98] StdPrimaryKeySpec : : = PRIMARY KEY (FieldList)
[99] HCPrimaryKeySpec : : = PRIMARY HCKEY [NOT UNIQUE] (FieldList)
[94] KeySpec : : = StdKeySpec | HCKeySpec
[95] StdKeySpec : : = KEY IS FieldList
[96] HCKeySpec : : = HCKEY [NOT UNIQUE] IS FieldList

Explanation: Only one PrimaryKey specification is allowed per table definition.

If no PrimaryKey is specified, all fields except BLOB or CLOB fields form the primary key in the order of specification.

The SQL-2 formulation *PRIMARY KEY (f1, f2, ..,fn)* is equivalent to the alternative (Transbase proprietary) formulation *KEY IS f1,f2,..,fn* (see below for an example). The SQL-2 formulation *PRIMARY HCKEY [NOT UNIQUE] (f1, f2, ..,fn)* is equivalent to the alternative (Transbase proprietary) formulation *HCKEY [NOT UNIQUE] IS f1,f2,..,fn*

For the semantics of the key specification see *CreateTableStatement* . See also the Performance Guide for more details.

The following two examples are equivalent. The first is the official SQL-2 notation supported by Transbase, the second is an alternative notation also supported by Transbase (note that the formulations exclude each other):

```
CREATE  TABLE  quotations
(   suppno        INTEGER,
    partno        INTEGER,
    price         NUMERIC(9,2),
    delivery_time INTEGER,
    PRIMARY KEY   (suppno, partno)
)
```

```
CREATE  TABLE  quotations
(   suppno        INTEGER,
    partno        INTEGER,
    price         NUMERIC(9,2),
    delivery_time INTEGER
 )  KEY  IS suppno, partno
```

The following two examples show alternative formulations of primary key via a TableConstraint and a FieldConstraint - this is possible if and only if one single field constitutes the primary key:

```
CREATE  TABLE  suppliers
(    suppno  INTEGER,
     name    CHAR(*),
     address CHAR(*),
     PRIMARY KEY(suppno)
)
```

```
CREATE  TABLE  suppliers
(    suppno  INTEGER  PRIMARY KEY,
     name    CHAR(*),
     address CHAR(*)
)
```

2.5.5. CheckConstraint

Specify a CheckConstraint for a table.

Syntax:
[104]   CheckConstraint : : = CHECK (SearchCondition)

Explanation: The SearchCondition specifies an integrity condition which must be fulfilled for all records of the table.

In detail, for all records of the table which are inserted or updated an error is reported if the condition *NOT (SearchCondition)* evaluates to TRUE.

If the CheckConstraint is specified with an explicit `ConstraintName`, an integrity violation message concerning this CheckConstraint reports this name, otherwise an implicitly generated name is reported. For the sake of easy error analysis, it is thus recommended to specify explicit and self-explanatory constraint names.

```
CREATE  TABLE  quotations
(    suppno        INTEGER,
     partno        INTEGER,
     price         NUMERIC(9,2),
     delivery_time INTEGER,
     CONSTRAINT    price100 CHECK (price < 100)
)
```

```
CREATE  TABLE  quotations
(    suppno        INTEGER,
     partno        INTEGER,
     price         NUMERIC(9,2),
     delivery_time INTEGER,
     CONSTRAINT    price_deliv
     CHECK (price < 20 OR delivery_time < 3)
)
```

In the first example, only one field is involved. Therefore it can also be formulated using the syntactic variation FieldConstraint:

```
CREATE  TABLE  quotations
(    suppno        INTEGER,
```

```
    partno        INTEGER,
    price         NUMERIC(9,2)
    CONSTRAINT    price100 CHECK (price < 100),
    delivery_time INTEGER
)
```

Note that in a FieldConstraint, there is no comma between the field definition and the constraint definition.

Catalog Tables: One entry into the table `sysconstraint` is made for each check constraint.

Null values: The definition of integrity violation is such that NULL values pass the test in most cases. In the example above, the constraint "price100" is not violated by a record with a NULL value on price, because the SearchCondition evaluates to unknown (thus the negation NOT(..) also evaluates to unknown but not to TRUE).

To make NULL values fail the test, one must explicitly formulate the CheckConstraint like: *CHECK (price IS NOT NULL AND ...)*.

To specify that the value must not be null the shorthand notation NOT NULL can be used in the field definition, but then it is not part of the specified constraint:

```
CREATE  TABLE  quotations
(   suppno        INTEGER,
    partno        INTEGER,
    price         NUMERIC(9,2)  NOT NULL
    CONSTRAINT    price100 CHECK (price < 100),
    delivery_time INTEGER
)
```

2.5.6. ForeignKey

Specify a Referential Constraint between 2 tables.

Syntax:

[105] ForeignKey : : = FOREIGN KEY (FieldList) ReferencesDef
[106] ReferencesDef : : = REFERENCES TableIdentifier [(FieldList)] [ON DELETE Action] [ON UPDATE Action]
[107] Action : : = NO ACTION | CASCADE | SET DEFAULT | SET NULL

Explanation: A referential constraint between 2 tables is specified.

With respect to the constraint, the table containing the fields of the foreign key is called the referencing table, the table which is mentioned after REFERENCES is called the referenced table. Analogously, the fields in the FOREIGN KEY clause and the (explicit or implicit) fields in the REFERENCES clause are called referencing fields and referenced fields, resp. The referencing and referenced fields must have same number and identical types.

If no field name list is specified in the REFERENCES clause, then the primary key combination of the referenced table constitutes the referenced fields. The referenced fields either must constitute the primary key or must have a UNIQUE INDEX.

The referential constraint is as follows:

For each record in the referencing table whose referencing fields do not have any NULL value, there must be one record in the referenced table with identical field values on the corresponding referenced fields.

Let RG and RD the referencing table and the referenced table, resp., i.e. RG references RD.

The following statements potentially violate a referential constraint:

1. INSERT, SPOOL, UPDATE, in RG
2. DELETE, UPDATE in RD.

A referential constraint can be specified to trigger compensating actions.

Specification of NO ACTION effectively is the absence of a triggered action.

If CASCADE is specified:

A deletion of record t in RD triggers the deletion of all matching records in the RG (thus maintaining the referential constraint). An update of a referenced field in RD triggers the corresponding update of all referencing fields in RG to the same value (thus maintaining the referential constraint).

If SET NULL or SET DEFAULT is specified:

A deletion of record t in RD triggers the update of the referencing fields of all matching records in RG to NULL or their DEFAULT value. The first case always maintains the referential constraint, the second case only if there is a matching DEFAULT value record in RD. An update is handled analogously.

```
CREATE  TABLE  quotations
(    suppno         INTEGER,
     partno         INTEGER,
     price          NUMERIC(9,2),
     delivery_time  INTEGER,
     CONSTRAINT     quotrefsupp
         FOREIGN KEY (suppno) REFERENCES suppliers(suppno)
         ON DELETE  SET NULL,
     CONSTRAINT quotrefpart
         FOREIGN KEY (partno) REFERENCES inventory(partno)
         ON DELETE  CASCADE,
)
```

In this (single field reference) example, also the syntactic shorthand variant of FieldConstraint can be used as shown below:

```
CREATE  TABLE  quotations
(    suppno         INTEGER
         CONSTRAINT quotrefsupp
         REFERENCES suppliers(suppno)
         ON DELETE  SET NULL,
     partno         INTEGER
         CONSTRAINT quotrefpart
         REFERENCES inventory(partno)
         ON DELETE  CASCADE,
     price          NUMERIC(9,2),
     delivery_time  INTEGER
)
```

Catalog Tables: For a referential constraint with a n-ary field combination, n records are inserted into *sysrefconstraint*.

Performance: DELETE and UPDATE operations on referenced tables which require the referential check on the referencing table are slow if the referencing table does not have a secondary index (or the primary key) on the referencing fields.

INSERTs and UPDATEs on referencing fields requiring the referential check on the referenced table are fast because by definition there is a index on the referenced fields.

Note

Like all constraints, referential constraints are effectively checked after execution of each SQL query. In general, it is therefore not possible to insert records into tables in arbitrary order if there exists a referential constraint between them.

2.6. AlterTableStatement

Serves to alter fields of a table and to add or remove table constraints.

Syntax:

[109]	AlterTableStatement	::=	AlterTableConstraint | AlterTableChangeField | AlterTableRenameField | AlterTableFields | AlterTableRename | AlterTableMove | AlterTableInmemory

Privileges: The user must be owner of the table.

2.6.1. AlterTableConstraint

Serves to add or remove a table constraint.

Syntax:

[110]	AlterTableConstraint	::=	ALTER TABLE TableIdentifier ConstraintAction
[111]	ConstraintAction	::=	ADD TableConstraintDefinition | DROP CONSTRAINT ConstraintIdentifier

Explanation:

In the TableConstraintDefinition all except the redefinition of PRIMARY KEY is allowed on this position.

The ADDition of a table constraint is rejected if the values in the database do not fulfill the constraint.

```
ALTER  TABLE  quotations DROP CONSTRAINT quotrefpart

ALTER  TABLE  inventory ADD  CONSTRAINT  qonh
    CHECK (quonhand/10*10 = quonhand)

ALTER  TABLE  quotations ADD  CONSTRAINT  quotrefpart
    FOREIGN KEY (partno) REFERENCES  parts2
    ON DELETE CASCADE
```

2.6.2. AlterTableChangeField

Serves to alter a the default specifiation of a field in a table.

Syntax:

[112] AlterTableChangeField : : = ALTER TABLE TableIdentifier ALTER [COLUMN] FieldIdentifier DefaultAction
[113] DefaultAction : : = SET DEFAULT Expression | DROP DEFAULT

Explanation:

`SET DEFAULT` specifies a default value for the field.

`DROP DEFAULT` removes the default value from the field. If the datatype of the field is defined by a domain, the domain's default specification becomes effective. In this case `SET DEFAULT NULL` and `DROP DEFAULT` are not equivalent.

Both statements do not change field values in the database.

```
ALTER  TABLE  quotations ALTER price SET DEFAULT 100.0
ALTER  TABLE  quotations ALTER delivery_time DROP DEFAULT
```

2.6.3. AlterTableRenameField

Serves to rename a field of a table.

Syntax:
[114] AlterTableRenameField : : = ALTER TABLE TableIdentifier
RENAME COLUMN FieldIdentifier TO FieldIdentifier

changes the name of a field of a table. The RENAME operation fails in case of a name conflict.

```
ALTER  TABLE  quotations RENAME COLUMN price TO prize
```

2.6.4. AlterTableFields

Serves to add, modify or drop one or more fields.

Syntax:
[115] AlterTableFields : : = ALTER TABLE TableIdentifier
AlterTableElem [, AlterTableElem] ...
[116] AlterTableElem : : = ADD FieldDefinition [PositionClause] |
MODIFY FieldDefinition [PositionClause] |
DROP FieldIdentifier
[117] PositionClause : : = [FIRST | AFTER FieldIdentifier]
[91] FieldDefinition : : = FieldIdentifier DataTypeSpec
[DefaultClause | AUTO_INCREMENT]
[FieldConstraintDefinition]...

Explanation:

Each AlterTableElem specifies a modification on the given table.

- `ADD FieldDefinition PositionClause` adds a new field and initializes it with the explicitly specified or implicit default value. When no PositionClause is given the field is placed as the last field of the table. Otherwise the field will be inserted at the specified position.

- *`MODIFY FieldDefinition PositionClause`* changes the data type of the field to the specified type. Already existing data will be converted due to this modification. Note that not all data types are compatible among each other. This operation also changes the position of the field within a table.

- *`DROP FieldIdentifier`* deletes a field of the table. A field cannot be dropped, if one of the following holds:

  - The field is part of the primary key.

  - The field is part of a secondary index.

  - The field is used within a constraint.

```
ALTER  TABLE  quotations
    ADD comment CHAR(*) DEFAULT '' FIRST,
    MODIFY delivery_time DOUBLE AFTER comment,
    DROP qonorder
```

2.6.5. AlterTableRename

Serves to rename a table.

Syntax:

[118]	AlterTableRename	: : =	ALTER TABLE TableIdentifier RENAME TO TableIdentifier

changes the name of a table. The RENAME operation fails in case of a name conflict.

```
ALTER  TABLE  geopoints RENAME TO coordinates
```

2.6.6. AlterTableMove

Serves to reorganize a table.

Syntax:

[119]	AlterTableMove	: : =	ALTER TABLE TableIdentifier MOVE [TO] MoveTarget
[120]	MoveTarget	: : =	BLOCK BlockNo | DATASPACE DataspaceIdentifier
[121]	BlockNo	: : =	IntegerLiteral
[7]	DataspaceIdentifier	: : =	Identifier

Explanation: The logical page ordering of segments (tables, indices or lobcontainers) can be reorganized by this statement. The lower bound and upper bound address of the reorganized segment are specified either by BlockNo and the maximal database size or by the size of the target dataspace.

All indices of the table, the lobcontainer (if the table contains one or more lob fields) and the table itself are moved.

This statement is only allowed on Transbase Standard Databases.

```
ALTER TABLE quotations MOVE TO DATASPACE dspace2;
```

```
ALTER TABLE quotations MOVE BLOCK 50000;
```

2.6.7. AlterTableInmemory

Serves to change the inmemory property of a table together with its indexes

Syntax:

[122] AlterTableInmemory ::= ALTER TABLE TableIdentifier InmemorySpec

The same InmemorySpec applies as in the CreateTableStatement

```
ALTER  TABLE T2 INMEMORY PRELOAD
ALTER TABLE T3 INMEMORY NONE --drops any inmemory property
```

2.7. DropTableStatement

Serves to drop a table in the database.

Syntax:

[123] DropTableStatement ::= DROP TABLE [IF EXISTS] TableIdentifier

Explanation: The specified table, all indexes and all triggers defined on that table are dropped. All views which are directly or transitively based on the specified table are also dropped. Error is returned if the table does not exis unless the IF EXISTS option is specified.

Privileges: The current user must have userclass DBA or must be owner of the table.

```
DROP TABLE quotations
```

2.8. CreateIndexStatement

Serves to create an index, fulltext index or a bitmap index on a table.

Syntax:

[124] CreateIndexStatement ::= StandardIndexStatement |
HyperCubeIndexStatement |
FulltextIndexStatement |
BitmapIndexStatement

2.8.1. StandardIndexStatement

Serves to create a standard index on a table.

Syntax:

[125] StandardIndexStatement ::= CREATE [UNIQUE] INDEX IndexIdentifier
ON TableIdentifier (IndexElemList)

		[KEY IS KeyList]
[126]	IndexElemList ::=	Expression [, Expression]...
[127]	KeyList ::=	FieldList

Explanation: An index with the specified name is created on the specified fields or expressions resp. of the specified table.

In most cases, indexes are built on one or several basic fields of a table. This means that the Expressions are simple field names.

It is possible to specify an Expression instead of a field name. All Expressions must be resolvable against the specified base table. Such an index can be efficiently used for query processing if all or a prefix of the Expressions appear in the search condition of a query.

If "CREATE UNIQUE .." is specified then then all records are required to be unique in the index. Insert and update operations which would result in at least two records with the same values on the specified element combination are rejected. See also the special section on *UNIQUE and KEY IS*

Indexes have no effect on query results except for possibly different performance (depending on the query type) and possibly different sort orders of query results (if no ORDER BY-clause is specified in the query).

Indexes on views are not allowed.

BLOB and CLOB fields can only be indexed by *fulltext indexes* .

Note

It is unwise to create a standard B-tree index on the highest weighted key fields because in Transbase an unnamed (multi field) index exists on the key fields anyway.

Privileges: The current user must have userclass DBA or must have userclass RESOURCE and be owner of the table.

```
CREATE INDEX quot_pa_pr
ON quotations (partno,price)
```

Example for efficient usage:

```
SELECT  partno, price from quotations WHERE partno = 221
```

```
CREATE INDEX suppliers_phonetic
ON suppliers (soundex(name), suppno)
```

Example for efficient usage:

```
SELECT  * from suppliers where soundex(name) = soundex('Stewart')
```

2.8.1.1. UNIQUE and KEY IS specification of an Index

Assume a base table T(k,t1,t2,t3) where field k is key field.

Assume an index Tx on T on fields (t1,t2).

Although the (t1,t2) combination is not declared to be key in T, you might want to specify a constraint that the value pairs are unique.

There are 2 different ways to achieve this. One way is to specify the UNIQUE clause for Tx at creation time. The alternative is to use the KEY IS clause.

```
CREATE UNIQUE INDEX Tx on T(t1,t2)
...  or by using the KEY IS clause:
CREATE INDEX Tx on T(t1,t2) KEY IS t1,t2
```

The KEY IS clause is useful if one more field (t3) should be part of the index but the uniqueness property should remain as sharp as before and not be weakened to the triple.

```
CREATE INDEX Tx on T(t1,t2,t3) KEY IS t1,t2
```

This construction is also useful for example for efficiently supporting the following kind of query:

```
SELECT t3 FROM T WHERE t1 = <const> and t2 = <const>
```

The index key on (t1,t2) supports the efficient processing of the search condition, and the integration of field t3 into the index directly supports the delivery of t3 without accessing the base table T.

2.8.2. HyperCubeIndexStatement

Serves to create a Hypercube index on a table.

Syntax:

[128]	HyperCubeIndexStatement : : =	CREATE INDEX IndexIdentifier ON TableIdentifier (FieldList) HCKeySpec
[100]	FieldList : : =	FieldIdentifier [, FieldIdentifier]...
[96]	HCKeySpec : : =	HCKEY [NOT UNIQUE] IS FieldList
[127]	KeyList : : =	FieldList

Explanation: An index with the specified name is created on the specified fields of the specified table. There must not be an index with the same name on any table in the database.

A HyperCube tree is specified instead of a standard compound B-tree. A HyperCube tree should have no fields as part of key which typically are not searched for - therefore fields and keys can be specified separately. The specified keys are UNIQUE by default unless the "NOT UNIQUE" clause is specified. All fields used as HyperCube key fields must be NOT NULL and must have a range check constraint.

As far as the processing of search predicates is concerned, HyperCube index behaves like a HyperCube base table. See the *Performance Guide* [perform.xhtml] for details.

Privileges: The current user must have userclass DBA or must have userclass RESOURCE and be owner of the table.

```
CREATE INDEX quot_pa_pr
ON quotations (partno,price) HCKEY IS partno, price
```

Example for efficient usage:

```
SELECT  partno, price from quotations
WHERE partno between 1000 and 2000 and
      price  between 200 and 300
```

The index supports the evaluation of the interval restriction on both fields which are not efficiently supported by a standard B-Tree index.

2.8.3. FulltextIndexStatement

Serves to create a fulltext index on a VARCHAR, CHAR, BLOB or CLOB field of a table.

Syntax:

[129]	FulltextIndexStatement	::=	CREATE [POSITIONAL] FULLTEXT INDEX IndexIdentifier [FulltextSpec] ON TableIdentifier (FieldIdentifier) [ScratchArea]
[130]	FulltextSpec	::=	[WITH SOUNDEX] [{ Wordlist) | Stopwords }] [Charmap] [Delimiters]
[131]	Wordlist	::=	WORDLIST FROM TableIdentifier
[132]	Stopwords	::=	STOPWORDS FROM TableIdentifier
[133]	Charmap	::=	CHARMAP FROM TableIdentifier
[134]	Delimiters	::=	DELIMITERS FROM TableIdentifier | DELIMITERS NONALPHANUM
[135]	ScratchArea	::=	SCRATCH IntegerLiteral MB

Explanation: All explanations are given in the separate chapter on *Fulltext Indexes* .

2.8.4. BitmapIndexStatement

Serves to create a bitmap index on a *BOOL*, *TINYINT*, *SMALLINT* or *INTEGER* field of a table.

Syntax:

[136]	BitmapIndexStatement	::=	CREATE BITMAP INDEX IndexIdentifier ON TableIdentifier (FieldIdentifier)

Explanation: A bitmap index with the specified name is created on the specified field of the specified table.

Bitmap indexes are preferably used for non-selective columns having few different values (e.g. classifications). Bitmap indexes are innately very space efficient. With their additional compression in average they occupy less than one bit per index row. A bitmap index can be created on any base table (B-Tree or Flat) having a single *INTEGER* field as primary key or an *IKACCESS* path.

Bitmap processing allows inexpensive calculation of logical combinations (*AND*/ *OR*/ *NOT*) of restrictions on multiple non-selective fields using bitmap intersection and unification.

There must not be an index with the same name on any table in the database. There must not be an index on the same field of the table.

2.9. DropIndexStatement

Serves to drop an index.

Syntax:

[137]	DropIndexStatement ::=	DROP INDEX [IF EXISTS] IndexIdentifier

Explanation: The specified index is dropped. Error is returned if the index does not exis unless the IF EXISTS option is specified.

Privileges: The current user must have userclass DBA or must be owner of the table on which the index has been created.

```
DROP INDEX quot_pa_pr
```

2.10. Triggers

2.10.1. CreateTriggerStatement

Serves to create a trigger on a table.

Syntax:

[138]	CreateTriggerStatement ::=	CREATE TRIGGER TriggerIdentifier TriggerActionTime TriggerEvent ON TableIdentifier [REFERENCING OldNewAliasList] TriggeredAction
[139]	TriggerActionTime ::=	BEFORE | AFTER
[140]	TriggerEvent ::=	INSERT | DELETE | UPDATE [OF FieldList]
[100]	FieldList ::=	FieldIdentifier [, FieldIdentifier]...
[141]	OldNewAliasList ::=	OldNewAlias [OldNewAlias]
[142]	OldNewAlias ::=	OLD [ROW] [AS] CorrelationIdentifier | NEW [ROW] [AS] CorrelationIdentifier
[143]	TriggeredAction ::=	[FOR EACH { ROW | STATEMENT }] [WHEN (SearchCondition)] TriggerSQLStatement
[144]	TriggerSQLStatement ::=	DMLorCallStatement | BEGIN ATOMIC DMLorCallStatement [; DMLorCallStatement]... END
[145]	DMLorCallStatement ::=	InsertStatement | UpdateStatement | DeleteStatement | CallStatement

Explanation:

A trigger is a user defined sequence of SQL modification statements or CallStatements which is automatically executed when a INSERT, UPDATE or DELETE statement is executed. The specification of a trigger contains a triggername, a triggerevent on a table (e.g. INSERT ON quotations) which specifies when the trigger is to be fired, a detailed trigger action time (BEFORE or AFTER) which specifies whether the trigger has to fired before

or after the insert action. Furthermore, a trigger has to specified to be either a row trigger (FOR EACH ROW, i.e. to be fired once for each processed record) or a statement trigger (FOR EACH STATEMENT, i.e. to be fired only once for the complete modification statement). The default is FOR EACH STATEMENT.

For a row trigger, the specified actions may refer to the actually processed record values. The syntax NEW.fieldname is the value of fieldname of the inserted record or the (possibly changed) value of fieldname for an record being updated. The syntax OLD.fieldname is the value of fieldname of a deleted record or the original field value for an record being updated.

The firing of a row trigger may be restricted along a condition (SearchCondition) which also may refer to the NEW or OLD field values of the record currently processed.

The keywords NEW and OLD may be overridden by a OldNewAliasList.

When a trigger is fired it runs under the privileges of the creator of the trigger.

If more than one trigger qualifies at the same TriggerActionTime, the order of execution is defined by ascending creation date.

UPDATEs and DELETEs on a table T which has triggers and also is the target of a referential constraint require special consideration. Referential actions (called reference triggers here) may occur if the reference constraint is specified with CASCADE, SET NULL or SET DEFAULT.

Triggers are performed in the following order:

```
  (1) Before-StatementTriggers
  (2) Before-Row Triggers
  (3) Reference Triggers
  (4) After-Row Triggers
  (5) After-Statement Triggers
```

Note that the firing of a trigger may cause the firing of subsequent triggers. It is recommended to use triggers moderately to keep the complexity of nested actions small. In particular, it is strongly discouraged to construct a set of triggers which lead to the effect that a modification of a table T fires a trigger which transitively fires another trigger which also tries to modify table T. This may cause endless loops or nonpredictable effects.

Privileges: The current user must have userclass DBA or RESOURCE and becomes the owner of the trigger. Additionally, the current user must have SELECT privilege on the table on which the trigger is created.

```
CREATE TRIGGER quotations_upd
BEFORE UPDATE OF price ON quotations
FOR EACH ROW
WHEN (NEW.price > OLD.price )
 INSERT INTO logquotationsprice
 VALUES (NEW.suppno,NEW.partno,NEW.price-OLD.price)
```

```
CREATE TRIGGER quotations_ins
BEFORE INSERT ON quotations
FOR EACH ROW
 CALL ProcInsQuot(NEW.suppno,NEW.partno,NEW.price)
```

2.10.2. DropTriggerStatement

Serves to drop a trigger.

Syntax:
[146] DropTriggerStatement : : = DROP TRIGGER TriggerIdentifier

Explanation: The specified trigger is dropped. Note that the trigger also is dropped if the table is dropped on which the trigger is defined.

Privileges: The current user must have userclass DBA or must be owner of the trigger.

```
DROP TRIGGER quotations_upd
```

2.11. Views

2.11.1. CreateViewStatement

Serves to create a view in the database.

Syntax:

```
[147]   CreateViewStatement ::= CREATE [OR REPLACE] VIEW ViewIdentifier
                                [ ( FieldList ) ]
                                AS SelectStatement [ WITH CHECK OPTION ]
[100]             FieldList ::= FieldIdentifier [ , FieldIdentifier ]...
```

Explanation: The CreateViewStatement creates a view with the specified ViewName and FieldName(s). An error is returned if a view with the same name exists unless the REPLACE option is specified. In the latter case the existing view is silently dropped before the new view is created.

An n-ary view must be defined by a SelectStatement which delivers n-ary records.

If no fieldlist is specified then the derived names of the SelectStatment implicitly form the field list. If an element of the SELECT list has no derived name (expression) then an error is returned. Note that by use of the AS clause, each SELECT element can explicitly be given a name.

The rows in a view are not stored in the database, but only the view definition. Queries on views are simply evaluated as if the view definition were incorporated into the query.

The created view is updatable if the SelectStatement is updatable. If the WITH CHECK OPTION is specified, the view must be updatable.

Insert, Update, Delete are only allowed on updatable views.

If the WITH CHECK OPTION is specified for a view v, then Insert and Update operations are rejected whenever any inserted or updated record does not fulfill the SearchCondition of the defining SelectStatement of v or any other view on which v is transitively based.

A view can be used in any SelectStatement like a table. Especially, existing views can be used for the definition of a view.

Indexes on views are not allowed.

The current user becomes owner of the view. If the view is not updatable, the user only gets a non-grantable SELECT-privilege on the view, otherwise the user gets the same view privileges on the view as those on the (one and only) table or view which occurs in the defining SelectStatement.

A view may also contain one ore more *RemoteTableNames*. When evaluating remote views, the privileges of the view owner apply for accessing remote tables. However, the current user must have at least ACCESS privilege for the remote database. If an updatable remote view is is specified as the target of an UPDATE or DELETE operation, all subqueries (if any) must specify tables residing on the same database. However, if the target table is local, any tables (remote or local) may be specified in subqueries.

Privileges: The current user must have userclass DBA or RESOURCE and must have the privileges for the defining *SelectStatement*.

A non-updatable view:

```
CREATE VIEW totalprice (supplier, part, total)
AS
SELECT name, description, price * qonorder
FROM suppliers, quotations, inventory
WHERE suppliers.suppno = quotations.suppno
    AND inventory.partno = quotations.partno
    AND qonorder > 0
```

2.11.2. DropViewStatement

Serves to drop a view in the database.

Syntax:
[148] DropViewStatement : : = DROP VIEW [IF EXISTS] ViewIdentifier

Explanation: The specified view is dropped.

All views which are directly or transitively based on the specified view are also dropped. Error is returned if the view does not exist unless the IF EXISTS option is specified.

Privileges: The current user must have userclass DBA or must be owner of the view.

```
DROP VIEW totalprice
```

3. Data Manipulation Language

The Data Modification Language (DML) portion of TB/SQL serves to extract data records from tables or views (SelectStatement), to delete records (DeleteStatement), to insert records (InsertStatement) and to update records (UpdateStatement). The following paragraphs describe the syntax of the DML botton up, i.e. the language description starts with basic units from which more complex units can be built finally leading to the four kinds of statements mentioned above.

3.1. FieldReference

The construct FieldReference is used to refer to a specific field of a specific table.

Syntax:

[149]	FieldReference : : = [FieldQualifier .] FieldIdentifier
[150]	FieldQualifier : : = TableIdentifier | ViewIdentifier | CorrelationIdentifier
[12]	TableIdentifier : : = [SchemaIdentifier .] Identifier
[13]	ViewIdentifier : : = [SchemaIdentifier .] Identifier
[18]	CorrelationIdentifier : : = Identifier
[16]	FieldIdentifier : : = Identifier

Explanation:

The FieldIdentifier denotes the name of a field of a table. The CorrelationName is a shorthand notation for a table introduced in the FROM-clause of a SelectStatement. See *SelectExpression* and *Rules of Resolution* for more details.

The following examples show the usage of Field in a SelectStatement. The last example explains the use of CorrelationName in QualifiedField.

```
SELECT suppno FROM suppliers
```

```
SELECT suppliers.suppno FROM suppliers
```

```
SELECT s.suppno FROM suppliers s
```

3.2. User

The keyword USER serves to refer to the name of the current user.

It can be used as a SchemaIdentifier and as a StringLiteral. Its value in a statement is the login name of the user who runs the statement. The type of the value is STRING.

```
SELECT suppno FROM suppliers
WHERE name = USER
```

```
SELECT tname FROM systable, sysuser
WHERE schema = userid
AND username = USER
```

3.3. Expression

An Expression is the most general construct to calculate non-boolean values.

Syntax:

[151] Expression ::= [Unary] Primary [Binary [Unary] Primary]...
[152] Unary ::= + | - | BITNOT
[153] Binary ::= + | - | * | / | BITAND | BITOR | StrConcat
[154] StrConcat ::= ||

Explanation: For Primaries of arithmetic types all operators are legal and have the usual arithmetic meaning. Additionally, the binary '+' is also defined for character types and then has the meaning of text concatenation.

Some of these operators are also defined for *The Data Types Datetime and Timespan*.

The operator precedences for arithmetic types are as usual: Unary operators bind strongest. BITAND / BITOR bind stronger than '*' and '/' which in turn bind stronger than binary '+' and '-'.

The operator // denotes concatenation of string values and is an alternative for + for strings, see example below.

Associativity is from left to right, as usual. See *Precedence of Operators* for a complete list of precedences.

Note

Computation of an Expression may lead to a type exception if the result value exceeds the range of the corresponding type. See *Type Exceptions and Overflow* [sql_type_exceptions]. See also *Null Values* .

```
- 5.0
-5.0
 price * -1.02
'TB/SQL' + ' ' + 'Language'
'Dear' + title + name
'Dear'  ||  title || name
+  1.1 * (5 + 6 * 7)
```

In all but the last example, the constituting Primaries are Fields or Literals. In the last example, the second *Primary* is itself an *Expression* in parentheses.

For the operands BITOR and BITAND see *The TB/SQL Datatypes BITS(p) and BITS(*)*.

3.4. Primary, CAST Operator

A Primary is the building unit for Expressions.

Syntax:

[155] Primary ::= SimplePrimary | CastPrimary
[156] CastPrimary ::= SimplePrimary CAST DataTypeSpec |
CAST (SimplePrimary AS DataTypeSpec)

Explanation: The functional notation CAST(…) is the official SQL standard syntax, the postfix notation is the traditional Transbase syntax.

A CAST operator serves to adapt the result of a SimplePrimary to a desired data type. The specified data type must be compatible with the result type of the SimplePrimary (but see also *CASTING to/from CHAR*).

If the CAST operator is used on NUMERIC, FLOAT or DOUBLE values to map them into BIGINT, INTEGER, SMALLINT or TINYINT values, truncation occurs. See the example below how to round values instead.

The function `TO_CHAR(<expr>)` is equivalent to `CAST(<expr> as CHAR(*))`.

Caution

CASTing produces a *type exception* when the value exceeds the range of the *target type*.

```
name CAST CHAR(30)
price CAST INTEGER          -- truncation
(price + 0.5) CAST INTEGER  -- rounding
```

3.5. SimplePrimary

A SimplePrimary is the building unit for CastPrimaries or Expressions.

Syntax:

```
[157]   SimplePrimary ::= Literal | User |
                          FieldReference |
                          Parameter |
                          PSM_Primary |
                          ( Expression ) |
                          ( SubTableExpression ) |
                          MathematicalStdFunction |
                          SetFunction |
                          WindowFunction |
                          ConditionalExpression |
                          TimeExpression |
                          SizeExpression |
                          LobExpression |
                          StringFunction |
                          SignFunction |
                          ResultcountExpression |
                          SequenceExpression |
                          ODBC_FunctionCall |
                          FunctionCall |
                          LastInsertIdFunc |
                          LastUpdateFunc |
                          CrowdStatusFunc |
                          ReplicationStatusFunc
[158]       Parameter ::= # IntegerLiteral ( DataType ) |
                          Colon SimpleIdentifier |
                          Questionmark
[159]           Colon ::= :
[160]    Questionmark ::= ?
```

Explanation:

A PSM_Primary is allowed only inside a PSM_Block. Please refer to the section on *PSM Blocks* for further explanations.

Parameter is the means to specify a parameter for a stored query in an application program. The notations without data type specification can be used wherever the type of the parameter can be deduced from its context (e.g. *Field = ?*). This can be enforced by a CAST operation on the parameter.

A SimplePrimary can be a parenthesized Expression which simply models that an Expression in parentheses is evaluated first.

If a *SubTableExpression* is used as a SimplePrimary, its result must not exceed one value (i.e. a single record with a single field), otherwise an error occurs at runtime. If its result is empty, it is treated as a *null value*.

```
price
0.5
(price + 0.5)
(SELECT suppno FROM suppliers WHERE name = 'TAS')
```

3.5.1. MathematicalStdFunction

A set of predefined mathematical functions

Syntax:

[161] MathematicalStdFunction : : = TrigonometricFunction | PiFunction | ExpLogFunction | PowLogFunction | SqrtFunction | FloorFunction | CeilFunction | SignFunction | WidthBucketFunction

[162] TrigonometricFunction : : = { SIN | COS | TAN | SINH | COSH | TANH | ASIN | ACOS | ATAN } (Expression)

[163] PiFunction : : = PI()

[164] ExpLogFunction : : = { EXP | LOG10 | LN } (Expression)

[165] PowLogFunction : : = { POWER | LOG } (Expression , Expression)

[166] SqrtFunction : : = SQRT (Expression)

[167] FloorFunction : : = FLOOR (Expression)

[168] CeilFunction : : = CEIL (Expression)

[169] SignFunction : : = SIGN (Expression)

[170] WidthBucketFunction : : = WIDTH_BUCKET (Expression , Expression , Expression , Expression)

Explanation:

Any NULL valued argument produces a NULL result.

The arguments of all functions must be of numerical type.

SIN(x), COS(x), TAN(x) compute the sine, cosine and tangent of x, resp.

SINH(x), COSH(x), TANH(x) compute the hyberbolic sine, hyberbolic cosine and hyberbolic tangent of x, resp.

ASIN(x), ACOS(x), ATAN(x) compute the inverse of sine, cosine and tangent of x, resp.

For ASIN, ACOS, an argument outside the range of -1 to +1 leads to error.

PI() delivers an approximation of the number PI.

EXP(x) computes the value of the exponential function of x.

LN(x), LOG10(x) compute the value of the natural logarithm and the base 10 logaritm of x, resp.

POWER(x,y) delivers the value of x raised to the power of y.

LOG(x,y) delivers the logarithm of y to the base of x.

SQRT(x) delivers the sqare root of of x.

FLOOR(x) computes the largest integer value less or equal to x. CEIL(x) computes the smallest integer value greater or equal x. The result type corresponds to the argument type, in case of input type NUMERIC(p,s) the result type has scale 0.

Function SIGN computes the sign of a numerical expression. It returns -1 for negaive values, +1 for positive values, else 0.

WIDTH_BUCKET(x, lwb, upb, count) is useful for calculating histograms. "count" divides the range between "lwb" anf "upb" in "count" parts with identical size ("buckets"). Buckets are numbered from 1 to count. The function delivers 0 if x is less than lwb, (count+1) if x is larger than upb, else the number of the bucket where x lies in (left bucket border included, right bucket border excluded). If "lwb" is greater than "upb", then the range from upd to lwb is the base for bucket definition, but rhe bucket numbering increases from right to left (also the inclusion/exclusion of bucket endpoints is swapped).

```
WIDTH_BUCKET(-1,  0,  10, 5)    delivers 0
WIDTH_BUCKET(2.1, 0,  10, 5)    delivers 2
WIDTH_BUCKET(8,   0,  10, 5)    delivers 5
WIDTH_BUCKET(11,  0,  10, 5)    delivers 6
WIDTH_BUCKET(10,  0,  10, 5)    delivers 6  (right endpoint of bucket 5 not included)
WIDTH_BUCKET(10,  10,  0, 5)    delivers 1  (inverse, right end point now included)
WIDTH_BUCKET(2.1, 10,  0, 5)    delivers 4  (inverse)
```

3.5.2. SetFunction

A SetFunction computes one value from a set of input values or input records.

Syntax:

[171]	SetFunction	: : = COUNT (*) | DistinctFunction | AllFunction
[172]	DistinctFunction	: : = { COUNT | SUM | AVG | MAX | MIN } (DISTINCT Expression)
[173]	AllFunction	: : = { SUM | AVG | MAX | MIN | VAR_POP | VAR_SAMP | VAR | STDDEV_POP | STDDEV_SAMP | STDDEV } ([ALL] Expression)

Explanation:

SetFunctions are typically used in the SELECT-clause or HAVING-clause of a *SelectExpression*.

COUNT (*) delivers the cardinality of the set of input records.

For all other forms of a SetFunction, the input is the set of values which results from application of the Expression to the input records. If a DistinctFunction is specified, all duplicate values are removed before the SetFunction is applied. Functions COUNT, SUM, AVG, MAX, MIN compute the cardinality, the sum, the average, the maximum and the minimum value of the input value set, resp.

Functions COUNT, MIN and MAX are applicable to all data types.

Functions AVG and SUM are applicable to arithmetic types and to TIMESPAN.

The result type of COUNT is BIGINT. The result type of AVG on arithmetic types is DOUBLE. For all other cases the result type is the same as the type of the input values. Of course, the CAST operator can be used to force explicit type adaptation.

SetFunctions except COUNT ignore null values in their input. If the input set is empty, COUNT delivers 0, all others deliver the *null value*.

Note

Function SUM and AVG may lead to a type exception if the sum of the input values exceeds the range of the result type. See *Type Exceptions and Overflow* [sql_type_exceptions].

The functions with suffix _POP and _SAMP calculate variance and standard deviation of an input set X1, .. ,XN. They convert arithmetic input to DOUBLE and deliver DOUBLE.

Let XM be the mean value of Xi.

Let XDS be SUM(Xi - XM) POW 2 , i.e. the sum of squared distances between Xi and XM (over all Xi).

Function VAR_POP delivers XDS / N .

Functions VAR and VAR_SAMP deliver XDS / (N-1) . If N is less than 2, the function returns NULL.

Function STDDEV_POP delivers sqrt(VAR_POP).

Functions STDDEV and STDDEV_SAMP deliver sqrt(VAR_SAMP).

Examples:

- How many distinct parts are ordered?

```
SELECT COUNT (DISTINCT partno)
FROM quotations WHERE qonorder > 0
```

- How many parts are delivered by those suppliers who deliver more than 2 parts?

```
SELECT suppno, COUNT (*) FROM quotations
  GROUP BY suppno HAVING COUNT (*) > 2
```

- What is the average order for each part?

```
SELECT partno, AVG(qonorder)
  FROM quotations GROUP BY partno
```

3.5.3. WindowFunction

While SetFunctions aggregate a set of input rows into one result row, a WindowFunction calculates one result row for every input row. Here the aggregates are calculated over a set of rows in the vicinity of the current input row.

Syntax:

[174]	WindowFunction : : =	WindowAggregate (ExpressionList) OVER ([PartitionClause] [OrderByClause] [WindowClause])
[175]	WindowAggregate : : =	{ SUM | AVG | COUNT | MIN | MAX | RANK | DENSE_RANK | VAR_POP | VAR_SAMP | VAR | STDDEV_POP | STDDEV_SAMP | STDDEV }
[176]	PartitionClause : : =	PARTITION BY { (ExpressionList) | ExpressionList }
[177]	OrderByClause : : =	ORDER BY { (ExpressionList) | ExpressionList }
[178]	WindowClause : : =	{ ROWS | RANGE } { PrecedingClause | BETWEEN LowerboundClause AND UpperboundClause }
[179]	PrecedingClause : : =	UNBOUNDED PRECEDING | ValueExpression PRECEDING |

		CURRENT ROW
[180]	LowerboundClause : : =	UNBOUNDED PRECEDING | ValueExpression { PRECEDING | FOLLOWING } | CURRENT ROW
[181]	UpperboundClause : : =	UNBOUNDED FOLLOWING | ValueExpression { PRECEDING | FOLLOWING } | CURRENT ROW
[182]	ValueExpression : : =	<a logical or physical offset>

Explanation: WindowFunction are useful for calculating rankings and running totals or moving averages. They are typically used in the SELECT clause of a *SelectExpression*. They operate on a query result set, i.e. after FROM, WHERE, GROUP BY and HAVING clauses are evaluated. First this result is partitioned according to the PartitionClause. Then each partition is processed row by row, so every row will become the current row once. The aggregate for the current row is calculated OVER a set of rows (window) in this partition, as defined by the WindowClause.

OVER() distinguishes a WindowFunction from a SetFunction.

ROWS specifies that the windows is defined between absolute boundary offsets. If ROWS is specified, there are no restrictions to the following OrderByClause and it is completely optional. Windows boundaries refer to row positions relative to the current row.

If the limits of a ROWS window are BETWEEN CURRENT ROW AND 5 FOLLOWING, then the current row and the five following rows are within the window. Therefore this ROWS window contains at most 6 rows.

RANGE specifies that the window is defined between relative boundary offsets. If RANGE is specified with a ValueExpression boundary, the OrderByClause is mandatory and must contain exactly one expression. These ValueExpression windows boundaries refer to the one field used in the OrderByClause.

If the limits of a RANGE window are BETWEEN CURRENT ROW AND 5 FOLLOWING, then the window contains all rows whose sort field is

(1) equal or larger than the sort expression of the current row and

(2) equal or smaller than the sort expression of the current row + 5.

Therefore this RANGE window can contain any number of rows.

ValueExpression is a logical or physical offset. For a ROWS window it must be a positive INTEGER constant or an expression that evaluates to a positive INTEGER value. For a RANGE window it must be a positive constant or expression of arithmetic type or of type TIMESPAN/INTERVAL. If ValueExpression FOLLOWING is the start point, then the end point must be ValueExpression FOLLOWING. If ValueExpression PRECEDING is the end point, then the start point must be ValueExpression PRECEDING.

Defaults:

If the PartitionClause is missing, the defaults is PARTITION BY NULL, i.e. no partitioning is applied.

If the OrderByClause is missing the WindowClause defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

If the OrderByClause is present the WindowClause defaults to RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. For RANK and DENSE_RANK the OrderByClause is mandatory.

OVER() is equivalent to OVER(PARTITION BY NULL RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING). This is also equivalent to a standard SetFunction without GROUP BY.

3.5.4. StringFunction

StringFunctions accept character expressions (strings) as input and compute integers or strings. NULL is returned when one of the operands is NULL.

Syntax:

```
[183]    StringFunction ::= PositionFunction |
                            InstrFunction |
                            LengthFunction |
                            UpperFunction |
                            LowerFunction |
                            TrimFunction |
                            SubstringFunction |
                            ReplaceFunction |
                            ReplicateFunction |
                            TocharFunction
```

The concatenation of strings is denoted using the infix operators '+' or '//' (see *Expression*).

3.5.4.1. PositionFunction

The POSITION function searches a string inside another string and computes the position of its first occurrence if any.

Syntax:

```
[184]    PositionFunction ::= POSITION ( Search IN SourceExpr )
[185]              Search ::= Expression
[186]          SourceExpr ::= Expression
```

Explanation: Source may be of type CLOB, VARCHAR, CHAR. Search must be CHAR or VARCHAR. Result-type is INTEGER.

If Search is the empty string, the function returns 1.

In general, the function checks whether Search occurs as substring in Source: if not it returns 0 else the position of the first occurrence (positions start with 1).

The search is made case sensitive. No wildcard mechanism is supported.

```
POSITION ( 'ssi' IN 'Mississippi' )     -->  3
POSITION ( 'mis' IN 'Mississippi' )     -->  0
```

3.5.4.2. InstrFunction

The INSTR function searches a string inside another string. It provides a superset of the functionality of POSITION.

Syntax:

[187] InstrFunction : : = INSTR (Source, Search [, Startpos, Occurrence])
[185] Search : : = Expression
[340] Source : : = VALUES (ValueList) |
TABLE ((ValueList) [, (ValueList)]...) |
TableExpression |
DEFAULT VALUES
[239] Startpos : : = Expression
[188] Occurrence : : = Expression

Explanation: Source may be of type CLOB, VARCHAR, CHAR. Search must be CHAR, VARCHAR or BINCHAR. Startpos and Occur must be arithmetic expressions of type INTEGER. Resulttype is INTEGER.

Default values for Startpos and Occurrence are 1.

Let s be the value of Startpos and o be the value of Occurrence.

In general, the function searches the string "Search" in "Source" starting the search on the s-th character of "Source" (for s >= 1). If o > 1 then the o-th occurrence of "Search" is searched.

If s <= -1 then the search is made backwards starting with the */s/-th* character counted relative to the end of "Source".

The search is made case sensitive. No wildcard mechanism is supported. The function returns 0 if the search is unsuccessful else the position of the detected substring.

```
INSTR ('Mississippi','ssi )         -->  3
INSTR ('Mississippi','ssi, 4 )      -->  6
INSTR ('Mississippi','ssi, -1 )     -->  6
INSTR ('Mississippi','ssi, -1, 2 )  -->  3
```

3.5.4.3. LengthFunction

The CHARACTER_LENGTH function computes the length of a CHAR, VARCHAR or CLOB value in characters.

Syntax:
[189] LengthFunction : : = CHARACTER_LENGTH (Source)
[340] Source : : = VALUES (ValueList) |
TABLE ((ValueList) [, (ValueList)]...) |
TableExpression |
DEFAULT VALUES

Explanation: Note that the function delivers number of characters, not number of bytes. For counting bytes use the operator SIZE.

3.5.4.4. UpperFunction, LowerFunction

The UPPER and LOWER function maps uppercase letters to lowercase letters and vice versa. *Syntax:*

Syntax:
[190] LowerFunction : : = LOWER (Source)
[191] UpperFunction : : = UPPER (Source)

[340] Source ::= VALUES (ValueList) |
TABLE ((ValueList) [, (ValueList)]...) |
TableExpression |
DEFAULT VALUES

Explanation: Source may be of type CLOB, VARCHAR, CHAR. Resulttype is same as Sourcetype.

The function UPPER (LOWER) replaces all lowercase (uppercase) letters by corresponding uppercase (lowercase) letters and leaves all other characters unchanged.

Which characters are mapped to their lowercase (uppercase) equivalent, is determined by the Locale setting of the database. All ASCII characters (a..z and A..Z) are always mapped. When e.g. the Locale setting of the database is a German one, German Umlaut characters are mapped.

```
UPPER ( 'Line:24' )        -->  'LINE:24'
```

3.5.4.5. TrimFunction

The TRIM function eliminates in a string leading and/or trailing characters belonging to a specifiable character set.

Syntax:

[192] TrimFunction ::= TRIM ([[Trimspec] [Trimset] FROM] Source) |
LTRIM (Source [, Trimset]) |
RTRIM (Source [, Trimset])

[193] Trimspec ::= LEADING | TRAILING | BOTH

[194] Trimset ::= Expression

[340] Source ::= VALUES (ValueList) |
TABLE ((ValueList) [, (ValueList)]...) |
TableExpression |
DEFAULT VALUES

Explanation: Source must be CLOB, CHAR or VARCHAR. Trimset must be CHAR or BINCHAR. Resulttype is same as Sourcetype.

FROM must be specified if and only if at least one of Trimset or Trimspec is specified.

If Trimspec is not specified, BOTH is implicit.

If Trimset is not specified, a string consisting of one ' ' (blank) is implicit.

Depending on whether LEADING, TRAILING, BOTH is specified, the TRIM function delivers a string which is made from Source by eliminating all leading characters (trailing characters, leading and trailing characters, resp.) which are in Trimspec.

Error occurs if Trimset is the empty string.

LTRIM(Source,Trimset) equals TRIM(LEADING Trimset FROM Source).

RTRIM(Source,Trimset) equals TRIM(TRAILING Trimset FROM Source).

```
TRIM ( '   Smith   ' )                  -->  'Smith'
TRIM ( ' ' FROM '   Smith   ' )         -->  'Smith'
TRIM ( BOTH ' ' FROM '   Smith   ' )    -->  'Smith'
TRIM ( LEADING ' ' FROM '   Smith   ' ) -->  'Smith   '
TRIM ( 'ijon' FROM 'joinexpression' )   -->  'express'
```

3.5.4.6. SubstringFunction

The SUBSTRING function extracts a substring from a string value.

Syntax:

```
[195]   SubstringFunction ::= SUBSTRING ( Source FROM Startpos [ FOR Length ] ) |
                              SUBSTR ( Source, Startpos [, Length ] )
[340]              Source ::= VALUES ( ValueList ) |
                              TABLE ( ( ValueList ) [, (ValueList) ]... ) |
                              TableExpression |
                              DEFAULT VALUES
[239]            Startpos ::= Expression
[240]              Length ::= Expression
```

Explanation: Source must be CHAR, VARCHAR or CLOB. Startpos and Length must be arithmetic. Resulttype is same as Sourcetype.

The function constructs a string which results from Source by extracting Length letters beginning with the one on position Startpos. If Length is not specified or is larger than the length of the substring starting at Startpos, the complete substring starting at Startpos constitutes the result.

If Startpos is less equal zero then Length (if specified) is set to Length + Startpos and Startpos is set to 1 .

Error occurs if Length is specified and less than zero.

If Startpos is larger than the length of Source, the result is the empty string.

`SUBSTR(Source,Startpos,Length)` is equivalent to `SUBSTRING(Source FROM Startpos FOR Length)`

```
SUBSTRING ( 'joinexpression' FROM 5 )        --> 'expression'
SUBSTRING ( 'joinexpression' FROM 5 FOR 7)   --> 'express'
SUBSTRING ( 'joinexpression' FROM 5 FOR 50)  --> 'expression'
SUBSTRING ( 'joinexpression' FROM -2 FOR 6)  --> 'join'
```

3.5.4.7. ReplaceFunction

The REPLACE function replaces substrings or characters in a string.

Syntax:

```
[196]     ReplaceFunction ::= REPLACE ( Subs1 BY Subs2 IN Source [ , Subsspec ] )
[197]            Subsspec ::= WORDS | CHARS
[198]               Subs1 ::= Expression
[199]               Subs2 ::= Expression
[340]              Source ::= VALUES ( ValueList ) |
                              TABLE ( ( ValueList ) [, (ValueList) ]... ) |
                              TableExpression |
                              DEFAULT VALUES
```

Explanation: Source must be CLOB, CHAR or VARCHAR. Subs1, Subs2 must be CHAR or BINCHAR. Resulttype is same as Sourcetype.

The function constructs from Source a result string by substituting certain substrings in Source.

If Subsspec is not defined or defined as WORDS, then all occurrences of Subs1 are replaced by Subs2 (after substitution, the inserted string Subs2 is not further checked for substitution).

If Subsspec is defined as CHARS, then Subs1 and Subs2 must have same length and each character in Source which is equal to the i-th character in Subs1 for some i is replaced by the i-th character of Subs2.

Subs1 must have length greater equal to 1.

```
REPLACE ( 'iss' BY '' IN 'Mississippi' )          --> 'Mippi'
REPLACE ( 'act' BY 'it' IN 'transaction' )        --> 'transition'
REPLACE ( 'TA' BY 'ta' IN 'TransAction' , CHARS ) --> 'transaction'
```

3.5.4.8. ReplicateFunction

The REPLICATE function replicates a string a specified number of times.

Syntax:

[200]	ReplicateFunction : : = REPLICATE (Source , Times)
[340]	Source : : = VALUES (ValueList) |
	TABLE ((ValueList) [, (ValueList)]...) |
	TableExpression |
	DEFAULT VALUES
[201]	Times : : = Expression

Explanation: Source must be CLOB,CHAR or VARCHAR. Times must be arithmetic. Resulttype is same as Sourcetype.

The function constructs a result string by concatenating Source t times where t is the value of Times. Error occurs if t is less than zero.

```
REPLICATE ( 'a' , 3 )    -->  'aaa'
```

3.5.4.9. SoundexFunction

The SoundexFunction generates a phonetic representation of its input string. 2 strings that sound similar should be mapped to the same representation by this function.

The output type of Soundex is not VARCHAR(4) as in several implementations but grows with the length of the input, thus is VARCHAR(*).

Syntax:

[202]	SoundexFunction : : = SOUNDEX (Expression)

Note that it is possible to build a phonetic secondary index onto a field of a table (see *CREATE INDEX statement*).

`SOUNDEX('Stewart') = SOUNDEX('Stuart')` evaluates to TRUE

3.5.5. TocharFunction

The Tocharfunction is a shorthand notation for a CAST operator to STRING.

Syntax:

[203] TocharFunction : : = TO_CHAR (Expression)

Explanation: `TO_CHAR(expr)` is equivalent to `CAST ( expr AS STRING )`.

3.5.6. ResultcountExpression

Numbers result records of a select query.

Syntax:
[204] ResultcountExpression : : = RESULTCOUNT

The keyword RESULTCOUNT is an expression which successively evaluates to 1, 2, 3 and so on. If it is used in the SELECT list of the outermost SELECT block, it will perform a numbering of the result records of the query.

The usage of RESULTCOUNT is also allowed in more than one block (for example in a UNION of blocks) and also in nested blocks, but the effect then is not deterministic.

If a query contains several parallel blocks combined with UNION or similar set operators, then a numbering of the result records is achieved by surrounding it with: `SELECT RESULTCOUNT, * FROM ( <original query> )`.

```
SELECT RESULTCOUNT, suppno, partno FROM quotations
```

3.5.7. SequenceExpression

Performs a nextval or currval operation on a sequence.

Syntax:
[205] SequenceExpression : : = SequenceIdentifier . NEXTVAL |
SequenceIdentifier . CURRVAL

Explanation: See *CreateSequenceStatement*.

```
INSERT INTO T VALUES(S.NEXTVAL,13,100);
```

3.5.8. ConditionalExpression

ConditionalExpressions compute one of several values depending on one or several conditions. They are introduced by keywords IF, CASE, DECODE, COALESCE, NVL, NULLIF.

Syntax:
[206] ConditionalExpression : : = IfExpression |
CaseExpression |
DecodeExpression |
CoalesceExpression |
NvlExpression |
NullifExpression

3.5.8.1. IfExpression

The IfExpression is the simplest ConditionalExpression. It computes one of 2 values depending on one condition.

Syntax:

[207] IfExpression : : = IF SearchCondition THEN Expression ELSE Expression FI

Explanation: The result value of the IfExpression is determined by the SearchCondition: if the SearchCondition evaluates to TRUE then the value of the Expression in the THEN-part is delivered else the value of the Expression in the ELSE-part.

The data types of the two Expressions must be compatible. If the types differ then the result is adapted to the higher level type.

```
SELECT suppno, partno, price *
IF suppno = 54 THEN 1.1 ELSE 1 FI
FROM quotations

SELECT suppno, partno,
price * IF suppno = 54
        THEN 1.1
        ELSE
        IF suppno = 57 THEN 1.2 ELSE 1 FI
    FI
FROM quotations
```

3.5.8.2. CaseExpression

The CaseExpression is the most general ConditionalExpression. It comes in the variants simple CASE and searched CASE.

Syntax:

[208] CaseExpression : : = SearchedCaseExpression |
SimpleCaseExpression
[209] SearchedCaseExpression : : = CASE
SearchedWhenClause [SearchedWhenClause]...
[ELSE Expression]
END
[210] SearchedWhenClause : : = WHEN SearchCondition THEN Expression
[211] SimpleCaseExpression : : = CASE CaseOperand
SimpleWhenClause [SimpleWhenClause]...
[ELSE Expression]
END
[212] SimpleWhenClause : : = WHEN WhenOperand THEN Result
[213] CaseOperand : : = Result
[214] Result : : = Expression
[215] WhenOperand : : = Expression

Explanation:

The SearchedCaseExpression successively evaluates the SearchConditions of its SearchedWhenClauses and delivers the value of the Expression in the THEN clause of the first SearchedWhenClause whose condition evaluates

to TRUE. If no condition evaluates to TRUE then the value of the Expression in the ELSE clause is delivered if it exists else NULL.

The SimpleCaseExpression successively compares the CaseOperand to the WhenOperand of its SimpleWhenClauses and delivers the value of the Expression in the THEN clause of the first matching SimpleWhenClause.

It is equivalent to a SearchedCaseExpression with multiple SearchConditions of the form `<CaseOperand> = <WhenOperand>` where the WhenOperand of the i-th SearchCondition is taken from the i-th SimpleWhenClause and the THEN clauses and ELSE clause (if existent) are identical.

For both variants, all Result expressions in the THEN clauses as well as the Result of the ELSE clause (if existent) must be type compatible. The result type of the CaseExpression is the highest level type of all participating result expressions.

For the SimpleCaseExpression, the types of the CaseOperand and all WhenOperands must be type compatible.

```
UPDATE quotations
SET price = price * CASE
    WHEN price > 25      THEN  1.5
    WHEN price > 19.5    THEN  1.4
    WHEN price > 5       THEN  1.3
    WHEN price > 1       THEN  1.2
    ELSE   1.1
    END

SELECT suppno, partno, price * CASE
    WHEN suppno = 54 THEN 1.1
    WHEN suppno = 57 THEN 1.2
    ELSE    1
    END
FROM quotations

SELECT suppno, partno, price * CASE  suppno
    WHEN 54 THEN 1.1
    WHEN 57 THEN 1.2
    ELSE    1
    END
FROM quotations
```

3.5.8.3. DecodeExpression

The DecodeExpression is an alternative way to denote a CaseExpression of variant SimpleCaseExpression.

Syntax:

[216]	DecodeExpression	: : = DECODE (CompareExpr , MapTerm [, MapTerm]... [, DefaultExpr])
[217]	MapTerm	: : = WhenExpr , ThenExpr
[218]	CompareExpr	: : = Expression
[219]	WhenExpr	: : = Expression
[220]	ThenExpr	: : = Expression
[221]	DefaultExpr	: : = Expression

Explanation: The CompareExpr is successively compared with the WhenExprs of the MapTerms. If the comparison matches then the corresponding ThenExpr is delivered as result. If none of the comparisons matches then DefaultExpr is delivered as result if specified otherwise the Null value. All expressions must be type compatible. The result type is the highest level type of all participating expressions.

```
SELECT suppno, partno, price *
        DECODE (suppno, 54, 1.1, 57, 1.2, 1)
```

```
FROM quotations
```

3.5.8.4. CoalesceExpression, NvlExpression, NullifExpression

COALESCE and NVL are shorthand notations for a CASE or IF which maps an Expression from the NULL value to a defined value. The NULLIF is a shorthand notation for a CASE or IF which maps an expression from a defined value to the NULL value.

Syntax:

[222]	CoalesceExpression	: : = COALESCE (ExpressionList)
[223]	NvlExpression	: : = NVL (Expression , Expression)
[224]	NullifExpression	: : = NULLIF (Expression , Expression)

Explanation:

All involved expressions must be of compatible types. The result type is the highest level type of the result expressions.

COALESCE delivers the first expression which does not evaluate to NULL if there exists such an expression otherwise NULL. Thus it is equivalent to an expression of the form:

```
CASE
    WHEN x1  IS NOT NULL  THEN  x1
    WHEN x2  IS NOT NULL  THEN  x2
    ...
ELSE NULL
END
```

Note that with COALESCE, each involved expression is denoted only once in contrast to an equivalent CASE or IF construction. Therefore, in general, the COALESCE runs faster.

NVL is equivalent to COALESCE but restricted to 2 arguments.

NULLIF delivers NULL if the comparisons of both Expressions evaluates to TRUE else it delivers the value of the first Expression. Thus it is equivalent to an expression of the form:

```
IF x1 = x2 THEN NULL ELSE x1 FI
```

NULLIF in general runs faster than an equivalent CASE or IF construction because the first expression is evaluated only once. It is most often used to map an explicitly maintained non-NULL default value of a field back to its NULL semantics when used for computation.

3.5.9. TimeExpression

A TimeExpression is an expression which is based on value of type DATETIME or TIMESPAN

Syntax:

[225]	TimeExpression	: : = [Selector OF] { Constructor | TimePrimary }
[226]	Selector	: : = DAY | WEEKDAY | WEEK | ISOWEEK | QUARTER | YY | MO | DD | HH | MI | SS | MS
[227]	Constructor	: : = CONSTRUCT Timetype (ConstituentList)
[228]	Timetype	: : = { DATETIME | TIMESPAN } [RangeSpec]

[229] ConstituentList : : = Constituent [, Constituent]...
[230] Constituent : : = Expression | SubTableExpression
[231] TimePrimary : : = DatetimeLiteral |
TimespanLiteral |
FieldReference |
Parameter |
CURRENTDATE | SYSDATE |
(Expression) |
(SubTableExpression) |
SetFunction |
TruncFunction |
ConditionalExpression
[232] TruncFunction : : = TRUNC (Expression)

Explanation: For all semantics see *The Data Types Datetime and Timespan*.

Note that a selector as well as a constructor binds more strongly than a CAST operator (see also *Precedence of Operators*).

```
DATETIME[YY:MS](1989-6-8 12:30:21.032)
CURRENTDATE
HH OF CURRENTDATE
WEEKDAY OF CURRENTDATE
CONSTRUCT TIMESPAN(:year,:month,:day)
```

3.5.10. SizeExpression

The SIZE [OF] Operator computes the size (length) of a CHAR, CLOB, BINCHAR or BLOB Expression in bytes.

For the number of characters use the CHARACTER_LENGTH function.

Syntax:
[233] SizeExpression : : = SIZE [OF] Literal |
SIZE [OF] FieldReference |
SIZE [OF] Parameter |
SIZE [OF] (Expression) |
SIZE [OF] (SubTableExpression) |
SIZE [OF] SetFunction |
SIZE [OF] ConditionalExpression |
SIZE [OF] LobExpression

Explanation: The resulting type of the argument of the SIZE operator must be CHAR(*), (VAR)CHAR(p), BINCHAR(*), BINCHAR(p), BITS(*), BITS(p), BLOB or CLOB. The resulting type of the operator is INTEGER.

If the argument of the operator is the NULL value, then the operator delivers NULL. Otherwise the operator delivers a value that denotes the size (in bytes, for BITS in bits) of its argument. If the argument is CHAR(*) or (VAR)CHAR(p) then the trailing *\0* is not counted. If the argument is BINCHAR(*) or BINCHAR(p) then the length field is not counted. If the argument is BLOB then the number of bytes that the BLOB object occupies is delivered. If the argument is CLOB then the number of bytes (not characters) that the CLOB object occupies is delivered.

Note also the strong binding of the SIZE operator (see *Precedence of Operators*).

```
SIZE OF 'abc'             --> 3
SIZE OF 0x0a0b0c          --> 3
SIZE OF bl                --> length of the BLOB column bl
```

```
SIZE OF bl SUBRANGE (1,10)   --> 10 if bl is at least 10 long.
```

3.5.11. LobExpression

A LobExpression delivers a LOB value or a part of a LOB value.

Syntax:

```
[234]   LobExpression ::= BlobExpression | ClobExpression
[235]  BlobExpression ::= FieldReference [ SUBRANGE ( Lwb , Upb ) ]
[236]  ClobExpression ::= FieldReference [ SUBRANGE ( Lwb , Upb ) ] |
                          SUBSTRING ( FieldReference FROM Startpos [ FOR Length ] ) |
                          SUBSTR ( FieldReference , Startpos [ , Length ] ) |
[237]             Lwb ::= Expression
[238]             Upb ::= Expression
[239]        Startpos ::= Expression
[240]          Length ::= Expression
```

3.5.11.1. BlobExpression

A BlobExpression delivers a BLOB value or a subrange of a BLOB value.

The field must be of type BLOB, Lwb and Upb must be of type TINYINT, SMALLINT, INTEGER or BIGINT. The resulting value of Lwb and Upb must not be less or equal 0.

If SUBRANGE is not specified, then the resulting value is the BLOB object of the denoted FieldReference. If one of FieldReference, Lwb or Upb is the NULL value then the resulting value is also the NULL value. Otherwise the BLOB object restricted to the indicated range is delivered.

The smallest valid Lwb is 1. If Upb is greater than (SIZE OF Field) then it is equivalent to (SIZE OF Field).

If the value of Upb is less than the value of Lwb then a BLOB object of length 0 is delivered.

Let bl a BLOB object of length 100:

```
bl SUBRANGE (1,1)       --> first byte of bl as BLOB
bl SUBRANGE (1,SIZE bl) --> bl as it is
bl SUBRANGE (50,40)     --> empty BLOB object
```

3.5.11.2. ClobExpression

The field must be of type CLOB, Lwb, Upb,Startpos and Length must be of type TINYINT, SMALLINT, INTEGER or BIGINT.

If SUBRANGE, SUBSTRING or SUBSTR is not specified, then the resulting value is the CLOB object of the denoted FieldReference.

The SUBSTRING or SUBSTR function extracts Length characters beginning with the one on position Startpos and the SUBRANGE function extracts (Upb-Lwb) bytes beginning at position Lwb. The smallest valid Lwb and Startpos is 1. If the value of Upb is less than the value of Lwb then a CLOB object of length 0 is delivered.

3.5.12. ODBC_FunctionCall

Syntax:
[241] ODBC_FunctionCall ::= LCBr FN FunctionIdentifier (Expression) RCBr
[242] FunctionIdentifier ::= Identifier
[243] LCBr ::= {
[244] RCBr ::= }

Explanation: By the ODBC function call syntax, an embedding of the ODBC functions is provided to the Transbase SQL syntax.

3.5.13. UserDefinedFunctionCall

Syntax:
[245] FunctionCall ::= FunctionIdentifier (ExpressionList)
[242] FunctionIdentifier ::= Identifier
[264] ExpressionList ::= Expression [, Expression)]...

Explanation: A UserDefinedFunction (written as function returning one value) can be called at any place in the SQL statement where one value is accepted as result. Parameters of the function may be Expressions delivering one value (including dynamic parameters '?' supplied by the application at runtime).

```
SELECT sqrt(field)
FROM T
WHERE field > 0
```

3.5.14. LastInsertIdFunc

Syntax:
[246] LastInsertIdFunc ::= LAST_INSERT_ID()

Explanation: LAST_INSERT_ID() delivers the value of a table's *AUTO_INCREMENT* field which has been most recently assigned a value via an INSERT statement which did not explicitly assign a value to it.

```
SELECT LAST_INSERT_ID() , ...
FROM T
WHERE ...
```

Refer to the section *AUTO_INCREMENT Fields* [AUTO_INCREMENT_Fields] for a more elaborated example.

3.5.15. LastUpdateFunc

The LastUpdFunc delivers date and time of the most recently committed update operation performed on the current database. This point in time may be delivered in local time (default) or in UTC.

Syntax:

[247] LastUpdateFunc : : = LAST_UPDATE([LastUpdOptions [SECOND]])
[@ ConnectionIdentifier]
[248] LastUpdOptions : : = UTC | LOCALTIME
[317] ConnectionIdentifier : : = Identifier < containing a ConnectionString >
[320] ConnectionString : : = [ssl:]//[Host]/<Dbname>[?OptionList] |
file://DirectoryLiteral[?OptionList] |
<Dbname>[@Host]

Explanation:

Without any parameter or with the option LOCALTIME, the time of the last update is delivered as a value of type DATETIME[YY:MS] inside the local timezone.

The option UTC transforms the result to a UTC time.

For each of the 2 variants, the optional parameter SECOND delivers the time value as Epoch value in type Bigint (seconds elapsed since (1970-1-1 00:00:00) UTC). This format discards the millisecond value.

With the optional ConnectionIdentifier the function addresses a remote database.

```
LAST_UPDATE()           --> DATETIME[YY:MS]: most recent update in local timezone
LAST_UPDATE(UTC)        --> DATETIME[YY:MS]: most recent update as UTC time
LAST_UPDATE(UTC SECOND) --> Bigint: most recent update in era
```

3.5.16. CrowdStatusFunc

The CrowdStatusFunc delivers the current status of the crowd service.

Syntax:
[249] CrowdStatusFunc : : = CROWD_STATUS ( )

Explanation:

The function returns either ACTIVE or INACTIVE.

ACTIVE means that the database is currently connected to the crowd.

INACTIVE means that there is no active connection between the database and the crowd.

```
SELECT CROWD_STATUS()
```

3.5.17. ReplicationStatusFunc

The ReplicationStatusFunc delivers the current status of a replication update process.

Syntax:
[250] ReplicationStatusFunc : : = REPLICATION_STATUS ( )

Explanation:

The function returns either ACTIVE or INACTIVE.

ACTIVE means that the replica is currently connected to the original database.

INACTIVE means that there is no active connection between the replica and the original database.

```
SELECT REPLICATION_STATUS()
```

3.5.18. SearchCondition

SearchConditions form the WHERE-clause and the HAVING-clause of a SelectExpression and return a boolean value for each record and group, resp. They also appear in ConditionalExpressions to choose one of two Expressions.

Syntax:

[251] SearchCondition : : = [NOT] Predicate [Boolop [NOT] Predicate]...]
[252] Boolop : : = AND | OR

Explanation: The precedence of operators is: 'NOT' before 'AND' before 'OR' (see *Precedence of Operators*). Additional parentheses may be used as usually to override precedence rules (see *Predicate*).

```
(suppno = 54 OR suppno = 57) AND qonorder > 0
```

```
NOT ((suppno <> 54 AND suppno <> 57) OR qonorder <= 0)
```

If no null values are involved, these two SearchConditions are equivalent.

3.5.19. HierarchicalCondition

In addition to the SearchCondition in the WHERE clause hierarchical data can be queried using HierarchicalConditions.

Syntax:

[253] HierarchicalCondition : : = [START WITH SearchCondition]
CONNECT BY [NOCYCLE] ConnectByCondition
[254] ConnectByCondition : : = <Predicate using HierarchicalExpression>
[255] HierarchicalExpression : : = LEVEL |
CONNECT_BY_ISCYCLE |
CONNECT_BY_ISLEAF |
PRIOR Expression |
CONNECT_BY_ROOT Expression |
SYS_CONNECT_BY_PATH (Expression, StringLiteral)

Explanation:

START WITH defines the set of root rows for a hierarchical query. It is formulated as an SearchCondition without HierarchicalExpressions. If this optional clause is omitted every row the set defined by the FROM clause is considered as root row.

CONNECT BY defines the relationship between the rows of a hierarchy. References to a prior or root rows or other hierarchical relations can be phrased using HierarchicalExpressions. NOCYCLE controls the behaviour if a cycle in the hierarchical data is encountered. Usually, if a cycle in hierarchical data is found, then this will result in an error, since otherwise the query would produce an infinite loop of records. If NOCYCLE is specified cycles are ignored, i.e. the algorithm will not follow a path that leads to a record that has already been printed.

HierarchicalExpressions can be used like FieldReferences throughout the current and outer query blocks, except in the START WITH clause.

LEVEL stands for the hierarchy level of the current row, i.e. a root node is on LEVEL 1, its successors are on LEVEL 2 and so on.

CONNECT_BY_ISCYCLE and CONNECT_BY_ISLEAF return integer values. Here a value of 1 indicates that the current row is the beginning of a cycle in a hierarchy or a leaf in the hierarchy, resp.

PRIOR and CONNECT_BY_ROOT are unary operators that indicate that Expression refers to FieldReferences from the prior or root row.

SYS_CONNECT_BY_PATH is a built-in function that calculates the path from the root row to the current row by concatenating the results of Expression for every visited predecessor, separating each with StringLiteral.

Please consider the following hierarchical data sample and queries.

Figure 3.1. Hierarchical data and graph

id	parent_id
1	2
2	1
3	2
4	2
5	6
6	5
7	7

```
SELECT id, parent_id, LEVEL "level",
CONNECT_BY_ISLEAF isleaf, CONNECT_BY_ISCYCLE iscyle,
SYS_CONNECT_BY_PATH(id,'/') path
FROM hierarchy
WHERE level < 4
START WITH id = 1
CONNECT BY NOCYCLE parent_id = PRIOR id
```

id	parent_id	level	isleaf	iscylce	path
1	2	1	0	0	/1
2	1	2	0	1	/1/2
3	2	3	1	0	/1/2/3
4	2	3	1	0	/1/2/4

At each point in time of the depth-first search, there exists a "current row". At the beginning, one of the root rows satisfying the START WITH condition is chosen as "current row". In this example it is the row with id 1. To find the next row, the ConnectByCondition is evaluated whereby the PRIOR expressions are those which refer to the current row (thus representing defined and constant value for the actual search) and the remaining expressions are treated like in a standard search. In the example, given the row id 1 as the current row, the search condition "parent_id = PRIOR id" effectively is "parent_id = 1" as PRIOR id evaluates to 1 in the current row. The first result record of this search then becomes the new "current row" and the algorithm proceeds depth-first down the hierarchy until no more successors are found. If a search on one level delivers more than one result record, the remaining records successively become the "current row" after the recursive searches have finished.

The result indicates that a cycle begins in row 2 since it leads back to the root row. This is also why the NOCYCLE option is required. Rows 3 and 4 are leaf rows.

```
SELECT id, parent_id, level "level",
SYS_CONNECT_BY_PATH(id,'/') path
FROM hierarchy
START WITH id > 4
CONNECT BY NOCYCLE parent_id = PRIOR id
```

id	parent_id	level	path
5	6	1	/5
6	5	2	/5/6
6	5	1	/6
5	6	2	/6/5
7	7	1	/7

Here query rows 5, 6, and 7 satisfy the START WITH condition. The query result is equivalent to a union of three queries with each of these rows successively acting as root rows.

3.6. Predicate

Predicates are the building units for SearchConditions.

Syntax:

[256] Predicate ::= (SearchCondition) | ComparisonPredicate | BetweenPredicate | LikePredicate | MatchesPredicate | ExistsPredicate | QuantifiedPredicate | NullPredicate | FulltextPredicate

3.6.1. ComparisonPredicate

A ComparisonPredicate compares two values or two sets of records or checks one value/record to be in a set of values/records.

Syntax:

[257] ComparisonPredicate ::= ValueCompPredicate | SetCompPredicate | InPredicate

3.6.2. ValueCompPredicate

A ValueCompPredicate compares two values.

Syntax:

[258] ValueCompPredicate ::= Expression ValCompOp Expression

[259] ValCompOp : : = < | <= | = | <> | > | >=

Explanation: The meaning of the operators are:

<	less than
<=	less than or equal to
=	equal
<>	not equal
>	greater than
>=	greater than or equal to

The data types of the Expressions must be compatible. If TableExpressions are used, they must deliver a single value.

The comparison operators are defined for all data types.

If two character sequences (strings) with different length are compared, then the shorter string is padded with the space character ' ' up to the length of the longer string.

For the following examples of correct ValueCompPredicates, assume that q is a correlation name for the table quotations.

```
suppno < 54
```

```
price * qonorder < 100.50
```

```
q.price >
(SELECT AVG (price)
FROM quotations
WHERE partno = q.partno)
```

```
(SELECT MAX(price) - MIN(price)
FROM quotations
WHERE partno = q.partno)
>
(SELECT AVG (price)
FROM quotations
WHERE partno = q.partno) * 0.5
```

The last example would be a suitable SearchCondition to find out partnos from records q in quotation with a big variance in price offerings.

3.6.3. SetCompPredicate

A SetCompPredicate compares two sets of records.

Syntax:
[260] SetCompPredicate : : = (SubTableExpression) SetCompOp (SubTableExpression)

[261] SetCompOp : : = [NOT] SUBSET [OF] | = | <>

Explanation: Let q1 and q2 be the two SubTableExpressions and s1 and s2 their result sets of records, resp. q1 and q2 must be compatible, i.e. their result sets must have the same arity n (n > 0) and each pair of types of the corresponding fields must be type compatible (see *Data Types*). Two n-ary records t1 and t2 *match* if they have the same values on corresponding fields. *q1 SUBSET q2* yields TRUE if for each record t1 from Result of q1 there is a matching record t2 in Result set of q2.

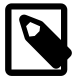
The following notations are equivalent:

```
q1 NOT SUBSET q2
NOT (q1 SUBSET q2)
```

Parentheses can be omitted, see *Precedence of Operators* *q1 = q2* yields TRUE if s1 and s2 are identical, i.e. for each record t1 in s1 there is a matching record t2 in s2 and vice versa.

The following notations are equivalent:

```
q1 <> q2
NOT q1=q2
```



Note

Duplicate records in any of s1 or s2 do not contribute to the result, i.e. sets are treated in the mathematical sense in all set comparison operators.

List all supplier numbers who deliver (at least) the same parts and price offerings as supplier 54

```
SELECT DISTINCT suppno
FROM quotations q
WHERE
    (SELECT partno, price
    FROM quotations
    WHERE suppno = 54)
        SUBSET
    (SELECT partno, price
    FROM quotations
    WHERE suppno = q.suppno)
```

3.6.4. InPredicate

The InPredicate checks if an explicitly specified record is in a set of records or checks if a value is in a set of values.

Syntax:

[262] InPredicate : : = ValueInPredicate | RecordInPredicate

[263] ValueInPredicate : : = Expression [NOT] IN { (ExpressionList) | (SubTableExpression) }

[264] ExpressionList : : = Expression [, Expression)]...

[265] RecordInPredicate : : = Record [NOT] IN (SubTableExpression)

[266] Record : : = (ExpressionList)

Explanation: A ValueInPredicate checks a value against a set of values. If an ExpressionList is specified, the predicate yields TRUE if the value of the left hand Expression is equal to one of the values of the ExpressionList. If a SubTableExpression is specified it must deliver unary records which then are interpreted as a set of values like above.

A RecordInPredicate checks a record against a set of records. It yields TRUE if the left hand record matches one of the result records of the SubTableExpression. Compatibility rules for Record and TableExpression are analogous to those of SetCompPredicate.

Note

The notation `x NOT IN y` is equivalent to `NOT x IN y`

```
SELECT *
FROM suppliers
WHERE suppno IN (54,61,64)
```

```
SELECT * FROM suppliers
WHERE suppno IN
    (SELECT suppno
    FROM quotations)
```

List suppliers who deliver at least one part for the same price as supplier 57

```
SELECT DISTINCT suppno FROM quotations
WHERE  (partno, price) IN
    (SELECT partno, price
    FROM quotations
    WHERE suppno = 57)
```

3.6.5. BetweenPredicate

The BetweenPredicate tests a value against an interval of two values. Each of the two interval boundaries can be specified as inclusive or exclusive.

Syntax:

[267]	BetweenPredicate ::=	Expression [NOT] BETWEEN Expression [BetweenQualifier] AND Expression [BetweenQualifier]
[268]	BetweenQualifier ::=	INCLUSIVE | EXCLUSIVE

Explanation: If a BetweenQualifier is omitted it is equivalent to INCLUSIVE. The notation *e1 BETWEEN e2 AND e3* therefore is equivalent to *e1>=e2 AND e1 <=e3*. The notation *e1 BETWEEN e2 EXCLUSIVE AND e3* is equivalent to *e1>e2 AND e1<=e3*. The notation *e1 NOT BETWEEN e2 AND e3* is equivalent to *NOT (e1 BETWEEN e2 AND e3)*

```
price BETWEEN 0.10 AND 0.30
```

```
q.price NOT BETWEEN
    (SELECT MAX (price) FROM quotations
    WHERE partno = q.partno) * 0.8  EXCLUSIVE
AND
    (SELECT MIN (price) FROM quotations
    WHERE partno = q.partno) * 1.2  EXCLUSIVE
```

3.6.6. LikePredicate

The LikePredicate tests a string value against a pattern.

Syntax:

[269]	LikePredicate ::=	Expression [NOT] LIKE Sensspec

		Pattern [ESCAPE EscapeChar]
[270]	Sensspec	: : = SENSITIVE | INSENSITIVE
[271]	Pattern	: : = Expression
[272]	EscapeChar	: : = Expression

Explanation: All specified Expressions must be of character type. The type of EscapeChar must be CHAR (1), i.e. a character string of byte length 1. EscapeChar is restricted to be a single character with Unicode value less than 128 (whose byte length is 1).

Note that all Expressions including the Pattern need not be constants but may also be calculated at runtime.

The result of Pattern is interpreted as a search pattern for strings where two special characters have the meaning of wild cards:

- The percent sign % matches any string of zero or more characters
- The underscore sign _ matches any single character

If EscapeChar is specified (let its value be c) then all occurrences of wild card characters in Pattern which are preceeded by a c are not interpreted as wild card characters but as characters in their original meaning and the EscapeChar c is not interpreted as part of the pattern in these occurences.

If Sensspec is not specified or is specified with SENSITIVE then the search pattern is interpreted case sensitive, otherwise the search is performed case insensitive.

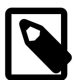
The insensitive character comparison depends on the Locale setting of the database.

The notations are equivalent:

```
s NOT LIKE p ESCAPE c
NOT (s LIKE p ESCAPE c)
```

```
description LIKE 'B%'
description LIKE INSENSITIVE '%_r'
description LIKE '%#%' ESCAPE '#'
```

The first example yields TRUE for values in description which begin with 'B', the second analogously for all values which end with 'r' or 'R' and have at least 2 characters. The third example yields TRUE for all values which end with the percent sign.



Note

If no wildcard is used in the pattern, e.g. *description LIKE 'xyz'* then this expression is *not* equivalent to *description = 'xyz'* because the string comparison ignores trailing blanks whereas the LIKE operator is sensitive with respect to trailing blanks.

3.6.7. MatchesPredicate, Regular Pattern Matcher

The MatchesPredicate tests a string value against a pattern denoted as a regular expression.

Syntax:

[273]	MatchesPredicate	: : = Expression [NOT] MATCHES Sensspec RegPattern [ESCAPE EscapeChar]
[270]	Sensspec	: : = SENSITIVE | INSENSITIVE
[274]	RegPattern	: : = Expression
[272]	EscapeChar	: : = Expression

Explanation: All specified Expressions must be of character type. The type of EscapeChar must be CHAR (1), i.e. a string of byte length 1.

EscapeChar is restricted to be a single character with Unicode value less than 128 (whose byte length is 1).

The result of RegPattern is interpreted as a regular expression. Regular expressions are composed of characters and metacharacters. Metacharacters serve as operands for constructing regular expressions. The following characters are metacharacters: *( ) { } [ ] * . , ? + - |*

In all following examples, the patterns and values are written in StringLiteral notation (i.e. with surrounding single quotes).

Characters and Character Classes:

Patterns may be composed of characters and character classes. A character in a pattern matches itself. For example, the pattern 'xyz' is matched by the value 'xyz' and nothing else (in case sensitive mode). A character class is either a dot sign (*'.'*) or a construct in square brackets *[]*. The dot sign is matched by any character. For example, the pattern 'x.z' is matched by values 'xaz', 'xbz', etc. A character class in *[]* is matched by any character listed in *[]*. The list is either a sequence of single characters like in *[agx]*, or it is a character range like *[a-z]* as a shorthand notation for all characters between a and z (in machine code), or it is a combination of both. For example, *[ad-gmn]* is matched by any of the characters a, d, e, f, g, m, n. Note that blanks would be interpreted as matchable characters, so don't write *[a b]* or *[ab]* if you mean *[ab]*. It is an error to specify character ranges like *[c-a]* where the machine code of the upperbound character is less than that of the first character.

Alternatives:

The | sign separates alternatives in a pattern. For example, the pattern *abc|yz* is matched by *abc* as well as by *yz*. The implicit character concatenation binds stronger than the alternative sign, so to match either *abcz* or *abyz* one has to specify the pattern *ab(c|y)z* (of course also *abcz|abyz* would work). Note also that character classes are nothing else but a shorthand notation for otherwise possibly lengthy alternatives, so *ab[cy]z* is equivalent to *ab(c|y)z*, too.

Repetition factors:

When an asterisk * occurs in a pattern, then zero or arbitray many occurrences of the preceding pattern element must occur in the value to match. For example, the pattern *abc** is matched by *ab*, *abc*, *abcc* etc. All repetition factor operands bind most strongly, so the pattern *(abc)** must be specified to match *abc*, *abcabc*, etc. The '+' sign means one or more occurrences of the preceding pattern element, so *x+* is identical to *xx**. The '?' sign means zero or one occurrences. At least n but maximal m occurrences of a pattern element x can be specified by the notation *x{n,m}* where n and m must be integer constants. For example *ag{1,3}z* is matched by *agz*, *aggz*, *agggz*.

Precedence of operands:

Three levels of precedence are given, namely the repetition factors which bind stronger than concatenation which binds stronger than the alternative. To overrule the precedence of operators, round precedence brackets can be used as shown in the above examples.

Escaping the metacharacters:

Whenever a metacharacter is to stand for itself (i.e. is not wanted in its meta meaning) it must be escaped. If EscapeChar is specified (let its value be c) then all occurrences of metacharacters in the pattern which are preceeded by the specified character are not interpreted as metacharacters but as characters in their original meaning and the escape character is not interpreted as part of the pattern in these occurences. For example, in the expression *value MATCHES '\/\?' ESCAPE '\'* the value */?* matches and any other value does not.

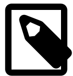
If the escape character is needed as normal character, it must be written twice (normally one can avoid this situation by choosing another escape character).

If Sensspec is not specified or is specified with SENSITIVE then the search pattern is interpreted case sensitive, otherwise the search is performed case insensitive. For example, the expression *s MATCHES INSENSITIVE 'ab.*z'* is equivalent to *s MATCHES SENSITIVE '(a|A)(b|B).*(z|Z)'*

Note that in case of INSENSITIVE, the ranges in character classes are somewhat restricted, i.e. if one of the characters is a lowercase (uppercase) character then the other must also be a lowercase (uppercase) character. For example, the ranges *[b-G]* or *[B-g]* are erroneous.

The notations are equivalent:

```
s NOT MATCHES p ESCAPE c
NOT (s MATCHES p ESCAPE c)
```



Note

The MatchesPredicate is more powerful than the LikePredicate which, however, is supported for compatibility. A pattern in a LikePredicate can be transformed to a regular patterns by substituting each non-escaped `%` by `.*` and each non-escaped _ by `.`.

3.6.8. ExistsPredicate

The ExistsPredicate tests the result of a SubTableExpression on emptyness.

Syntax:

[275] ExistsPredicate ::= EXISTS (SubTableExpression)

Explanation: The predicate evaluates to TRUE if the result of the SubTableExpression is not empty.

Which suppliers supply at least 1 part:

```
SELECT suppno, name FROM suppliers s
WHERE EXISTS
    (SELECT *
    FROM quotations
    WHERE suppno = s.suppno)
```

3.6.9. QuantifiedPredicate

A QuantifiedPredicate compares one value against a set of values.

Syntax:

[276] QuantifiedPredicate : : = Expression ValCompOp Quantifier (SubTableExpression)
[259] ValCompOp : : = < | <= | = | <> | > | >=
[277] Quantifier : : = ALL | ANY | SOME

Explanation: The SubTableExpression must deliver unary records (i.e. a set of values) whose type is compatible with that of Expression.

If ALL is specified, the predicate is TRUE if the specified comparison is true for all values delivered by the SubTableExpression or if the SubTableExpression delivers no value.

If ANY or SOME is specified, the predicate is TRUE if the TableExpression delivers at least one value for which the specified comparison is TRUE. Note that ANY and SOME have precisely the same meaning.

List suppliers and parts for which there is no cheaper offering

```
SELECT suppno, partno FROM quotations q
WHERE price <= ALL
    (SELECT price
    FROM quotations
    WHERE partno = q.partno)
```

List all other suppliers

```
SELECT suppno, partno FROM quotations q
WHERE price > ANY
    (SELECT price
    FROM quotations
    WHERE partno = q.partno)
```

3.6.10. NullPredicate

A Null-Predicate checks the result of an Expression against the null value.

Syntax:

[278] NullPredicate : : = Expression IS [NOT] NULL |
Expression = NULL |
Expression <> NULL

The following notations are equivalent:

```
Expression IS NULL
Expression = NULL
```

The following notations are equivalent, too:

```
Expression IS NOT NULL
NOT (Expression IS NULL)
Expression <> NULL
```

For the semantics of the NullPredicate see *Null Values*.

3.6.11. FulltextPredicate

On fulltext-indexed fields search expressions of type FulltextPredicate can be issued.

Syntax:

[279] FulltextPredicate ::= FieldIdentifier CONTAINS [SOUNDEX] (FulltextTerm)
[280] FulltextTerm ::= FulltextFactor [OR FulltextFactor]
[281] FulltextFactor ::= FulltextPhrase [Andnot FulltextPhrase]
[282] Andnot ::= AND | NOT
[283] FulltextPhrase ::= (FulltextTerm) |
Atom [[DistSpec] Atom]...
[284] Atom ::= SingleValueAtom | MultiValueAtom
[285] SingleValueAtom ::= StringLiteral | Parameter | FtExpression
[286] MultiValueAtom ::= ANY (TableExpression)
[287] DistSpec ::= [[MinBetween ,] MaxBetween]
[288] MinBetween ::= <Expression of type Integer>
[289] MaxBetween ::= <Expression of type Integer>
[290] FtExpression ::= <Expression without FieldReference to same block>
[1] [::= [
[2]] ::=]
[158] Parameter ::= # IntegerLiteral (DataType) |
Colon SimpleIdentifier |
Questionmark
[33] StringLiteral ::= CharacterLiteral | UnicodeLiteral | USER

Explanation: All explanations are given in *FulltextIndexes* .

3.7. Null Values

A record may have undefined values (null values) as field-values (if the corresponding CreateTableStatement of the table allows it).

A special constant NULL is provided to test a result of Expressions against the null value inside a SearchCondition.

```
price = NULL
price <> NULL
```

The first expression delivers TRUE if the field price is null-valued, it delivers FALSE if the field price has a known value.

If a null-valued field participates in an arithmetic operation (+, -, *, /), the result is again null-valued.

If a null-valued Expression participates in a ValueCompPredicate, the result of the ValueCompPredicate is UNKNOWN.

The evaluation rules for boolean operators are given in tables *not* , *or* , *and* .

Table 3.1. NOT operator

NOT	TRUE	FALSE	UNKNOWN
	FALSE	TRUE	UNKNOWN

Table 3.2. OR operator

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

Table 3.3. AND operator

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

If the result of a SearchCondition is UNKNOWN, it is equivalent to FALSE.

Assume that a field named price is null-valued for a specific record, the following SearchConditions effectively evaluate as follows:

```
price < 10.0   -- FALSE
price >= 10.0  -- FALSE
price = null   -- TRUE
```

Further rules for Null-values:

- SetFunctions ignore null-values in their input.
- As far as grouping and duplicate elimination (DISTINCT) is concerned all null-valued fields form one group and are considered equal, resp.
- In the ORDER BY clause, all NULL values are sorted before any other value.

3.8. SelectExpression (QueryBlock)

A SelectExpression derives records from tables in the database. QueryBlock is a synonym for SelectExpression.

Syntax:

[291]	SelectExpression	: : =	SelectClause	/
			[FromClause	* 6 *//
			[WhereClause]	* 1 *//
			[UngroupClause]	* 2 *//
			[GroupClause]	* 3 *//
			[HavingClause]	* 4 *//
			[FirstClause]]	* 5 */
[292]	SelectClause	: : =	SELECT [ALL | DISTINCT] SelectList	
[293]	FromClause	: : =	FROM TableReference [, TableReference]...	
[294]	WhereClause	: : =	[WHERE SearchCondition]	
			[HierarchicalCondition]	
[295]	UngroupClause	: : =	UNGROUP BY FieldReference	
[296]	GroupClause	: : =	[GROUP BY Expression [, Expression]...]	
[297]	HavingClause	: : =	[HAVING SearchCondition]	
[298]	SelectList	: : =	SelectElem [, SelectElem]...	
[299]	SelectElem	: : =	Expression [AS FieldIdentifier] |	
			[CorrelationIdentifier .] *	
[300]	FirstClause	: : =	FIRST (CountSpec [SortSpec])	
[301]	CountSpec	: : =	[StartNum TO] EndNum	
[302]	StartNum	: : =	IntegerLiteral	

[303] EndNum : : = IntegerLiteral
[334] SortSpec : : = ORDER BY ALL |
ORDER BY SortElem [, SortElem]...

Explanation: Each TableReference must identify an existing table or view in the database.

Each specified `CorrelationName` must be unique within all `TableName`s and `CorrelationName`s of that FROM clause.

All specified Expressions in the GROUP-BY-clause must have a resolution in the given QueryBlock (see *Rules of Resolution*).

Each TableReference exports a set of field names (see TableReference). These names can be used in the defining QueryBlock to reference fields of the result records of the FROM clause.

The QueryBlock also exports a set of field names: the i-th field of the block exports name "f" if it is a field reference with fieldname "f" or "q.f" or if it is specified with an "AS f" clause, otherwise the i-th field is "unnnamed".

The result of a QueryBlock is defined by conceptually performing the following steps:

1. The Cartesian product of the results from the TableReferences in the FROM-clause is constructed.

2. a. All joins defined by the SearchCondition of the WHERE-clause are performed.

   b. The HierarchicalCondition is processed. A depth-first search is carried out starting with one of the root rows that satisfies the START WITH predicate. For this root row the first child rows satisfying the CONNECT BY condition is selected. Then the hierarchical search will proceed down through the generations of child rows until no more matching rows are found.

   c. The SearchCondition of the WHERE-clause is applied to all records resulting from the previous steps. The result of (2) is the set of records which satisfy the SearchCondition and HierarchicalCondition. In addition then the result is sorted in depth-first order with respect to the HierarchicalCondition, if any.

3. The GROUP-BY-clause partitions the result of (2) into groups of records. All records with the same values on the specified Expressions form one group. Thus, the number of groups is equal to the number of different value combinations on the specified Fields. If a GROUP-BY-clause is specified the following conditions must hold: Asterisk (*) is not permitted in the SELECT-clause. Each Expression in the SELECT-clause and in the HAVING-clause which refers to the given QueryBlock (i.e. whose resolution block is the given QueryBlock, see *Rules of Resolution*) either must be an identical grouping Expression or must be inside a SetFunction whose resolution block is the given QueryBlock.

4. The SearchCondition of the HAVING-clause is applied to each group. The result of (4) is the set of groups which satisfy the SearchCondition. If no GROUP-BY-clause is specified, the whole set of records from the previously executed step forms one group.

5. Result records according to the SELECT-clause are constructed. If neither GROUP-by nor HAVING is specified, each record from the previously executed step contributes to one result record and each SetFunction which has a local resolution refers to all input records of (5). If GROUP-BY and/or HAVING is specified, each group contributes to one result record and each local SetFunction is computed separately for each group.

6. If DISTINCT is specified, duplicate records are removed from the result of (5). By default or if ALL is specified, duplicate records are not removed.

7. If a FirstClause is specified, only a subset of the result records is delivered. These are the records with ordinal numbers from "StartNum" to "EndNum". If StartNum is not specified it defaults to 1. A SortSpec is mandatory if the FirstClause is not in the outermost query block.

The asterisk notations in the SelectList are shorthand notations for a list of field names. The pure * stands for a list of all field names exported by all TableReferences of the FROM-clause in their original order. The notation *Identifier.** stands for a list of all field names of the TableReference with `CorrelationName` or `TableName` Identifier. The asterisk notations can be freely mixed among each other and other Expressions.

Updatability:

A SelectExpression is updatable if all following conditions hold:

- No GROUP BY-clause and no HAVING-clause is specified.
- No DISTINCT is specified in the SELECT-clause.
- The SELECT-clause consists of * or each Expression in the ExpressionList only consists of a FieldReference (i.e. no operators, no SetFunction) and each *FieldName* only occurs once.
- The WHERE-clause does not contain a subquery.
- The FROM-clause only contains one single TableReference and this TableReference is updatable.

Note

The UngroupClause is a very special operator only used in conjunction with the type BITS(*). It is explained in *The Datatypes BITS(p) and BITS(*)*.

Which parts are delivered by more than 1 supplier?

```
SELECT partno, COUNT (*)
FROM quotations
GROUP BY partno
HAVING COUNT (*) > 1
```

What is the average number of suppliers for a part?

```
SELECT AVG(cnt) FROM
   (SELECT COUNT (*) AS cnt
   FROM quotations
   GROUP BY partno)
```

To introduce a field name `cnt` for the unnamed field `COUNT(*)` this is specified as a FieldIdentifier.

Which suppliers deliver part 221 (all suppliers and part information delivered):

```
SELECT * FROM quotations q, suppliers s
WHERE q.suppno = s.suppno AND q.partno = 221
```

Which suppliers deliver part 221 (Only suppliers information delivered)

```
SELECT DISTINCT s.* FROM quotations q, suppliers s
WHERE q.suppno = s.suppno AND q.partno = 221
```

3.9. TableExpression, SubTableExpression

A TableExpression and SubTableExpression construct UNIONs, INTERSECTions and set DIFFerences from the result sets of TableReferences.

SubTableExpression is a slightly restricted form of TableExpression (is made up of SubTableReferences instead of TableReferences).

Syntax:

[304]	TableExpression ::= TableTerm [UnidiffSpec TableTerm]
[305]	SubTableExpression ::= SubTableTerm [UnidiffSpec SubTableTerm]

[306] TableTerm : : = TableReference [IntersectSpec TableReference]
[307] SubTableTerm : : = SubTableReference [IntersectSpec SubTableReference]
[308] UnidiffSpec : : = UnidiffOp [CorrespondingSpec]
[309] UnidiffOp : : = UNION [ALL] | DIFF | EXCEPT
[310] IntersectSpec : : = INTERSECT [CorrespondingSpec]
[311] CorrespondingSpec : : = CORRESPONDING [BY (FieldList)]

Explanation: UNION computes the set theoretical union of both input sets. If ALL is specified then duplicate records are retained otherwise they are removed. DIFF and INTERSECT compute the set theoretical difference and intersection, resp. Duplicate records are removed.

EXCEPT is a synonym for DIFF.

Note that according to the grammar rules, INTERSECT binds stronger than UNION and DIFF. Associativity is from left to right (see also *Precedence of Operators*).

- The expression

  ```
  A setop CORRESPONDING BY (C1, …, Cn) B
  ```

  where setop is one of the set operators is equivalent to

  ```
  (SELECT C1, …, Cn FROM A) setop (SELECT C1, …, Cn FROM B)
  ```

- The expression

  ```
  A setop CORRESPONDING B
  ```

  is equivalent to

  ```
  A setop CORRESPONDING BY (C1,..,Cn) B
  ```

  where C1, …, Cn are the fields with common names in A and B in the order of A. If A and B have no fields in common, an error is returned.

The result types of DIFF are those of the left operand. With UNION and INTERSECT, the type adaption rules (see *Data Types* and *Type Compatibility*) are applied to determine the result types.

DIFF preserves the naming of its left operand, i.e. if the i-th field of the left operand of DIFF is named 'xyz' (or is unnamed), then the i-th field of the result of DIFF is also named 'xyz' (is unnamed, resp.).

The i-th result field of a UNION or INTERSECT with operands A and B is unnamed if the i-th field of either A or B is unnamed or if their names differ, otherwise it is named and has the common input name as name.

Updatability:

A TableExpression is updatable if no UNION, INTERSECT, DIFF is specified and if the underlying TableReference is updatable.

CorrelationNames:

A TableExpression exports no *`CorrelationName`* if one of the set operators UNION, INTERSECT, DIFF is specified, otherwise it exports the *`CorrelationName`* of the constituting TableReference.

```
SELECT * FROM quotations
UNION CORRESPONDING BY (suppno)
SELECT * FROM suppliers
```

3.10. TableReference, SubTableReference

A TableReference or SubTableReference is the constituent of the FROM clause and of UNION / INTERSECTION / DIFF expressions (TableExpressions).

Syntax:

[312]	TableReference ::=	[TABLE] TableSpec [[AS] Alias] | FUNCTION TableFunction | SelectExpression | (TableExpression) [[AS] Alias] | JoinedTable | FULLTEXT <SpecialFulltextTable> | FlatFileReference | CrowdExpression
[313]	SubTableReference ::=	SelectExpression | (SubTableExpression)
[314]	TableSpec ::=	LocalTableSpec | RemoteTableSpec
[315]	LocalTableSpec ::=	TableIdentifier
[316]	RemoteTableSpec ::=	TableIdentifier @ ConnectionIdentifier]
[317]	ConnectionIdentifier ::=	Identifier < containing a ConnectionString >
[318]	Alias ::=	[CorrelationIdentifier] [(FieldList)]
[100]	FieldList ::=	FieldIdentifier [, FieldIdentifier]...
[319]	TableFunction ::=	FunctionIdentifier (ExpressionList)
[264]	ExpressionList ::=	Expression [, Expression)]...
[320]	ConnectionString ::=	[ssl:]//[Host]/<Dbname>[?OptionList] | file://DirectoryLiteral[?OptionList] | <Dbname>[@Host]
[321]	Host ::=	<Hostname>[:<Portnumber>]
[322]	OptionList ::=	Option[&Option]...
[323]	Option ::=	<Key>=<Value>

Explanation:

TableReference and SubTableReference are the building blocks for TableExpression and SubTableExpression.

Note

TableExpression constitutes the top query block and subqueries in FROM clause, SubTableExpression constitutes subqueries in SELECT, WHERE, HAVING clause.

Tip

If a TableExpression TE is needed on a place where only a SubTableExpression is allowed, then the equivalent expression *(SELECT * FROM (TE))* can be used. As can be seen from the grammar, this is a SubTableReference and thus also a SubTableExpression.

If Alias is specified with a FieldList, the exported field names are defined by the specified FieldIdentifiers. The number of FieldNames specified must be equal to the arity of `TableName` or TableExpression, resp.

If no FieldList is specified, the exported field names are derived from the underlying syntactic construct (e.g. TableSpec exports the field names of the fields of the specified table).

Updatability:

A TableReference is updatable if one of the following holds:

- A TableSpec T is specified and T is a basetable or an updatable view.
- An updatable SelectExpression is specified.
- An updatable (TableExpression) is specified.

CorrelationNames:

see *TableReferences, CorrelationNames and Scopes* .

RemoteTableName:

Host names are always treated case insensitive.

Database names are always treated case sensitive.

Important

The identifiers in the remote TableIdentifiers, ViewIdentifiers or Identifiers of other user-defined database objects are mapped to names according to the local case-sensitivity settings.

When a field `SomeThing` in a table `Table` is referenced from a case insensitive database, then the field and table must be identified in the SQL statement like this:

```
SELECT "SomeThing" FROM "Table"@db@host
```

When a field `SOMETHING` in a table `TABLE` is referenced from a case sensitive database, then the field and table must be identified in the SQL statement like this:

```
SELECT SOMETHING FROM "TABLE"@db@host
```

Important

The current user requires the necessary access privileges on the remote database using the same credentials on the remote database as for the local database.

Examples:

In the following example, an alias is used for each of the TableReferences, the first consisting of a *`CorrelationName`* only, the second with a field list.

```
SELECT  q.partno, supp.sno, supp.addr, supp.name
```

```
FROM quotations q, suppliers supp (sno, name,addr)
WHERE q.suppno = supp.sno
```

The following example needs an Alias for its subquery to reference its result field - it works with or without a CorrelationName. Both solutions are shown:

```
SELECT AVG (q.cnt) FROM
   (SELECT COUNT (*)
   FROM quotations
   GROUP BY partno) q (cnt)
```

```
SELECT AVG(cnt) FROM
    (SELECT COUNT (*)
    FROM quotations
    GROUP BY partno)  (cnt)
```

The following example is a distributed join using a remote database otherdb@server5.

```
SELECT  q.partno, supp.sno
FROM quotations q, suppliers@otherdb@server5 supp
WHERE q.suppno = supp.sno
```

3.11. FlatFileReference - direct processing of text files

A FlatFileReference enables to use a flat text file as a base table in a SELECT statement. The text file must be in standard SPOOL format, i.e. the field values must be separated by tabs.

Syntax:

[324] FlatFileReference ::= (FileLiteral) [CorrelationIdentifier] (Fieldspec [, Fieldspec]...)

[325] Fieldspec ::= FieldIdentifier DataType

Explanation:

Note that the name of the file holding the data must be enclosed in single quotes.

Each data field in the file must be specified by an arbitrary name and by its datatype. The syntax of the data values in the file must correspond to that of the SPOOL statement. The tabulator character is used to separate data values.

Example:

Given a file `Persons` in directory `/usr/x/data` which stores some lines each consisting of two character strings and a number, the file may be processed like this:

```
SELECT * FROM
  ( '/usr/x/data/Persons') (firstname CHAR(*), secondname CHAR(*), info NUMBER)
  WHERE info > 100
  ORDER BY secondname
```

3.12. CrowdExpression

A CrowdExpression is used to distribute a subquery to a set of connected databases. The fields of the retrieved result records may be treated like any other basetable fields.

Transbase Crowd Queries [crowd.xhtml#Introduction] describes the concept and common use cases of crowd queries.

Syntax:

[326]	CrowdExpression	::=	CROWD (CrowdIdentifier [, CrowdIdentifier]...) [RETURN AFTER] IntegerLiteral SECONDS [OR IntegerLiteral MEMBERS] (CrowdSubquery) CorrelationIdentifier (CrowdFieldspecList) | LastCrowdErrors
[327]	CrowdIdentifier	::=	Identifier
[328]	CrowdSubquery	::=	SelectExpression
[329]	CrowdFieldspecList	::=	Fieldspec [, Fieldspec]...
[325]	Fieldspec	::=	FieldIdentifier DataType

3.12.1. Crowd Queries

Example:

The actual date and time of all databases, which are connected to crowd "my_crowd", is collected by the following query:

```
SELECT h.c, h.crowd_members
FROM CROWD (my_crowd) RETURN AFTER 3 SECONDS
(SELECT currentdate) h (c datetime[yy:ss])
```

Explanation:

A *CrowdSubquery* is distributed to all databases whose crowd name matches the *CrowdIdentifier*. The query is sent to all members who are already connected and to all those who still go online during the given time window. Databases specify the crowd master, to which they connect to, through the database property *crowd_master* in the catalog table *sysdatabase*. The crowd identifier is set by the connection string property *crowd* in the former database property. Note that this crowd identifier can be named arbitrarily.

```
ALTER DATABASE SET CROWD_MASTER='//localhost:2024/crowd_master?crowd=my_crowd';
ALTER DATABASE SET CROWD_CONNECT=true;
```

In the example above a database connects to crowd master "crowd_master" on the localhost with port 2024 and the crowd identifier is "my_crowd".

The first result record will be delivered after the time window has expired or, if specified, after the desired number of crowd members have returned a complete result. If the time window expires without a member delivering a complete result, an empty result set is delivered as the overall result.

Note that the subquery *CrowdSubquery* is not beeing compiled on this level. So it can be considered as a black box. But the compiler needs to know, how the result records look like. Therefore the user has to specify each data field by an arbitrary name and by its datatype (*CrowdFieldspecList*).

Each CrowdExpression has an additional (hidden) field *crowd_members* in its result records, which delivers the total amount of connected databases, which returned a complete result.

3.12.2. Crowd Errors

Example:

The following query delivers all errors of the previously executed crowd query:

```
SELECT crowd_member, crowd_level, crowd_errorcode, crowd_errortext FROM LastCrowdErrors;
```

Explanation:

Records of the expression *LastCrowdErrors* contain four fields: *crowd_member*, *crowd_level*, *crowd_errorcode*, *crowd_errortext*.

crowd_member is the UUID of database, which caused the error.

crowd_level is the nested level where the error occurred. Level zero corresponds to the database, where the crowd query was executed, level one corrresponds to its members and so on.

crowd_errorcode contains the Transbase error code and *crowd_errortext* is the corresponding error message.

3.13. JoinedTable (Survey)

A JoinedTable combines tables with an explicit join operator.

Syntax:

```
[330]  JoinedTable ::= TableReference CROSS JOIN TableReference |
                       TableReference UNION JOIN TableReference |
                       TableReference NATURAL [ Jointype ] JOIN TableReference |
                       TableReference [ Jointype ] JOIN TableReference [ Joinpred ] |
                       ( JoinedTable )
[331]  Jointype ::=    INNER |
                       { LEFT | RIGHT | FULL } [ OUTER ]
                       /* default join type */
                       /* OUTER has no effect */
[332]  Joinpred ::=    ON SearchCondition |
                       USING ( FieldList )
```

Explanation:

CROSS JOIN is a syntactic variant for Cartesian product, i.e. the following expressions are semantically equivalent:

```
A CROSS JOIN B
SELECT * FROM A,B
```

The expression *A UNION JOIN B* (where A has a fields and B has b fields) is semantically equivalent to :

```
SELECT  A.*, NULL, NULL, ...   -- b NULLs
  FROM A
UNION
SELECT  NULL, NULL, ..., B.*   -- a NULLs
  FROM B
```

The result table has a+b fields and each record either has the first a or the last b fields all NULL.

The other join variants are described in the following chapters.

CorrelationNames:

A JoinedTable exports the `CorrelationName`s of both participating TableReferences - i.e. none, one or both CorrelationNames.

```
SELECT A.*, B.*
FROM  A UNION JOIN B.
```

3.13.1. INNER JOIN with ON/USING Clause

Explanation:

- *Case (1), ON Clause specified:* Let searchcond be the search condition. The expression

  ```
  A [INNER] JOIN B ON searchcond
  ```

  semantically is equivalent to

  ```
  SELECT * FROM A, B WHERE searchcond
  ```

- *Case (2), USING Clause specified:* If the USING Clause is specified then let C1,~…,~Cn denote the FieldList. All Ci's must be fields of both A and B. The expression

  ```
  A [INNER] JOIN B USING (C1, ... ,Cn)
  ```

  semantically is equivalent to

  ```
  SELECT A.C1, A.C2, ... , A.Cn,
         <other A fields>, <other B fields>
  FROM A, B
  WHERE  A.C1=B.C1 AND ... AND A.Cn=B.Cn
  ```

  Each of the result fields Ci appears only once in the result table (i.e. the number of result fields isa+b-n). The result fields C1, ..., Cn have no CorrelationNames (even if the constituting TableReferences A and B export CorrelationNames, say "a" and "b"). Thus, in the surrounding SelectExpression, C1, …, Cn can only be referenced by their unqualified name. The remaining fields of A and B have CorrelationNames "a" and "b" of A and B (if they exist). Note that also *a.** and *b.** refer to the remaining fields in their original order without any Ci.

3.13.2. JoinedTable with NATURAL

Explanation: The expression

```
A NATURAL [INNER] JOIN B
```

is equivalent to

```
A [INNER] JOIN B USING (C1,..Cn)
```

where C1, …, Cn are the fields (in the order of A) which are common to A and B.

If no fields are common, the expression degenerates to *A UNION JOIN B*.

The following statements all deliver the same result:

1.
```
SELECT  q.partno, q.suppno, q.price, q.delivery_time,
    q.qonorder, i.description, i.qonhand
FROM quotations q, inventory i
WHERE q.partno= i.partno
```

2.
```
SELECT  q.partno, q.suppno, q.price, q.delivery_time,
        q.qonorder, i.description, i.qonhand
FROM quotations q  JOIN  inventory i
ON q.partno= i.partno
```

3.
```
SELECT  partno, q.*, i.*
FROM quotations q  JOIN  inventory i USING (partno)
```

4.
```
SELECT  partno, q.*, i.*
FROM quotations q  NATURAL JOIN  inventory i
```

Note the meaning of q.* and i.* in the context of USING and NATURAL. Note also that suppno and partno are opposite to their original order in quotations.

3.13.3. JoinedTable with OUTER JOIN

Explanation:

We discuss the join variants

- *Case (1), ON Clause specified:* Assume the expressions LJ, RJ, FJ, IJ as:

LJ	A LEFT JOIN B ON searchcond
RJ	A RIGHT JOIN B ON searchcond
FJ	A FULL JOIN B ON searchcond
IJ	A INNER JOIN B ON searchcond

Let *innerjoin* denote the result of IJ. Then the result sets of LJ, RJ, FJ are defined as:

Result LJ	innerjoin UNION ALL leftouter
Result RJ	innerjoin UNION ALL rightouter
Result FJ	innerjoin UNION ALL fullouter

where *leftouter* , *rightouter*, *fullouter* are defined as follows:

- leftouter the set of all records a from A which do not participate in innerjoin, extended to the right with NULL values up to the arity of innerjoin.
- rightouter set of all records b from B which do not participate in the set innerjoin, extended to the left with NULL values up to the arity of innerjoin.
- fullouter leftouter UNION ALL rightouter.

- *Case (2), USING Clause specified:*

  Let JU denote the join expression

  ```
  A lrf JOIN B USING (C1, ... ,Cn)
  ```

  where lrf is one of LEFT, RIGHT, FULL.

  Let searchcond be the following search condition:

  ```
  A.C1=B.C1 AND ... AND A.Cn=B.Cn
  ```

  Then the result of JU is defined to be equivalent to:

  ```
  SELECT  COALESCE(A.C1,B.C1), ..., COALESCE(A.Cn,B.Cn),
      <other fields of A> , <other fields of B>
  FROM A lrf JOIN B ON searchcond
  ```

  The USING variant works like the ON variant except that the specified common fields appear only once in the result (always the not-NULL part of the field if any appears). Note that the COALESCEd fields do not have a *CorrelationName* and that the CorrelationNames exported by A and B do not include the COALESCEd fields.

- *Case (3), NATURAL specified:* Let NJ denote the expression:

  ```
  A NATURAL lrf JOIN B
  ```

  where lrf is one of LEFT, RIGHT, FULL.

  NJ is equivalent to

  ```
  A lrf JOIN B ON (C1, ..., Cn)
  ```

  where C1, …, Cn are all fields with *identical* names in A and B (in the order as they appear in A).

For all following examples, we assume excerpts S and Q from suppliers and quotations as shown in tables *S* and *Q* below.

Example Data:

- Excerpt from Table S

suppno	name
51	DEFECTO PARTS
52	VESUVIUS,INC

suppno	name
53	ATLANTIS CO.

- Excerpt from Table Q

suppno	partno
50	221
51	221
53	222
53	232

Examples:

- `SELECT * FROM S LEFT JOIN Q ON S.suppno = Q.suppno`

S.suppno	S.name	Q.suppno	Q.partno
51	DEFECTO PARTS	51	221
52	VESUVIUS INC	NULL	NULL
53	ATLANTIS CO	53	222
53	ATLANTIS CO	53	232

- `SELECT * FROM S RIGHT  JOIN Q ON S.suppno = Q.suppno`

S.suppno	S.name	Q.suppno	Q.partno
NULL	NULL	50	221
51	DEFECTO PARTS	51	221
53	ATLANTIS CO	53	222
53	ATLANTIS CO	53	232

- `SELECT * FROM S FULL  JOIN Q ON S.suppno = Q.suppno`

S.suppno	S.name	Q.suppno	Q.partno
NULL	NULL	50	221
51	DEFECTO PARTS	51	221
52	VESUVIUS INC	NULL	NULL
53	ATLANTIS CO	53	222
53	ATLANTIS CO	53	232

- `SELECT * FROM S LEFT  JOIN Q USING (suppno)`

suppno	S.name	Q.partno
51	DEFECTO PARTS	221
52	VESUVIUS INC	NULL
53	ATLANTIS CO	222
53	ATLANTIS CO	232

Note that the first result field can only be referenced by the unqualified name *suppno*.

- `SELECT * FROM S NATURAL FULL JOIN Q`

suppno	S.name	Q.partno
50	NULL	221
51	DEFECTO PARTS	221
52	VESUVIUS INC	NULL
53	ATLANTIS CO	222
53	ATLANTIS CO	232

Note that the first result field can only be referenced by the unqualified name *suppno*.

3.14. Scope of TableReferences and CorrelationNames

CorrelationNames may be used in qualified field references - they are of the form *q.field* where *q* is a CorrelationName.

A TableReference which constitutes a FROM clause operand exports a CorrelationName in the following cases:

- If the TableReference specifies an Alias, then the *`CorrelationName`* specified in the Alias is exported.
- If the TableReference is a *`TableName`* without a specified Alias, then a *`CorrelationName`* which is identical to the *`TableName`* implicitly is exported.
- If the TableReference is a JoinedTable then the CorrelationName(s) exported by JoinedTable are exported (see JoinedTable).

The Scope of the *`CorrelationName`* is the SelectExpression that immediately contains the TableReference. However, excluded are SelectExpressions which are nested in the containing one and define a *`CorrelationName`* with the same name.

In the following example, TableReferences "quotations" and "suppliers s" export CorrelationNames "quotations" and "s", resp.:

```
SELECT  quotations.*, s.name
FROM quotations, suppliers s
WHERE quotations.suppno = s.suppno
```

In the following example, the JoinedTable exports CorrelationNames "q" and "s":

```
SELECT s.name, q.price
FROM quotations q JOIN suppliers s ON q.suppno = s.suppno
```

In the following example, the JoinedTable also exports CorrelationNames "q" and "s", but the common result field "suppno" has no CorrelationName and q.* and s.* do not include the field "suppno":

```
SELECT suppno, s.*, q.*
FROM  quotations q NATURAL JOIN suppliers s
```

3.15. SelectStatement

The SelectStatement is the top level construct of TB/SQL to retrieve records.

Syntax:

```
[333]    SelectStatement ::= [ WithClause ] TableExpression
                             [ { SortSpec | FOR UPDATE } ]
[334]    SortSpec ::= ORDER BY ALL |
                      ORDER BY SortElem [, SortElem ]...
[335]    SortElem ::= { FieldIdentifier | IntegerLiteral } [ ASC | DESC ]
```

Explanation: The WithClause is explained in the next section.

The ORDER-BY-clause sorts the result records of the TableExpression. If more than one SortElem is specified, a multi-field sort is performed with the first SortElem being the highest sort weight etc.

If a SortElem is given via an IntegerLiteral i, it refers to the i-th result field of the TableExpression. Field numbering starts at 1. Otherwise there must be an identically named result field of the TableExpression and the SortElem refers to that field. The *next example below* shows two equivalent sort specifications.

By default the sort order is ascending unless explicitly descending order is required ('DESC').

Note

A SelectStatement is updatable if no ORDER BY-clause is specified and if the TableExpression itself is updatable (see *TableExpression*).

Caution

There is no predictable ordering of result records, if no order-by-clause is specified.

A SelectStatement is called a `SELECT FOR UPDATE` query if the `FOR UPDATE` is specified. It is necessary if and only if a subsequent UPDPOS or DELPOS statement (Update- or Delete-Positioned) is intended against the query. In case of a `FOR UPDATE`, the SelectStatement must be updatable.

Privileges: The current user must have SELECT privilege on each table (base table or view) specified in any FROM-clause of any QueryBlock which occurs in the SelectStatement.

Locks: All tables and views referenced in the SelectStatement are automatically read locked.

If the `FOR UPDATE` is given, the (single) table in the outermost QueryBlock is update locked.

```
SELECT partno, price FROM quotations ORDER BY partno

SELECT partno, price FROM quotations ORDER BY 1

SELECT * FROM quotations FOR UPDATE
```

3.16. WithClause

The WithClause defines temporary tables being used by the subsequent SelectExpression.

Syntax:

[333] SelectStatement : : = [WithClause] TableExpression
[{ SortSpec | FOR UPDATE }]
[336] WithClause : : = WITH WithElement [, WithElement]...
[337] WithElement : : = TableIdentifier [(FieldList)] AS (TableExpression)
[100] FieldList : : = FieldIdentifier [, FieldIdentifier]...

Explanation:

Each WithElement creates a (temporary) table with the specified table name. The main TableExpression in the SelectStatment can refer to all tables defined by the WithElements, and each WithElement may refer inside the defining TableExpressions to the preceding WithElements. The field names of the temporary tables are exported either by an explicit FieldList or by the field names of the SELECT lists or (in case of expressions) via Aliases introduced via the keyword AS.

WithElements are processed and stored in the order of writing. After the evaluation of the main TableExpression of the SelectStatement the temporary tables are deleted.

The WithClause is useful if a subexpression is used more than once in a complex query.

```
WITH
WE1 as (SELECT r1+r2 AS we1_1, r0 FROM R WHERE r0 > 0),
WE2 as (SELECT s1+s2 AS we2_1, s0 FROM S WHERE s0 > 0)
SELECT MAX(we1_1) FROM WE1
UNION ALL
SELECT we2_1 FROM WE1 ,WE2 WHERE r0 = s0

In this example, the definition of WE1 is useful, but the
definition of WE2 is not.

WITH
WE1 (w1,w2)  as (SELECT r1+r2 , r0 FROM R WHERE r0 > 0)
SELECT MAX(w1) FROM WE1
UNION ALL
SELECT w2 FROM WE1

In this example, the field names of WE1 are exported via an explicit
FieldList in WE1.
```

3.17. InsertStatement

The InsertStatement inserts one or several constant records or a computed set of records into a table or updatable view. Optionally, for each inserted record a result record is returned.

Syntax:

[338] InsertStatement : : = INSERT [OrClause] INTO TableSpec [(FieldList)] Source [ReturningClause]
[339] OrClause : : = OR { IGNORE | REPLACE | UPDATE }
[314] TableSpec : : = LocalTableSpec | RemoteTableSpec
[100] FieldList : : = FieldIdentifier [, FieldIdentifier]...
[340] Source : : = VALUES (ValueList) |
TABLE ((ValueList) [, (ValueList)]...) |
TableExpression |
DEFAULT VALUES
[341] ValueList : : = Expression [, Expression]...)
[342] ReturningClause : : = RETURNING (ExpressionList)

Explanation: The table specified by TableSpec must be updatable (i.e. a base table or an updatable view.

All fields in the specified FieldList must be unique and must be fields of the specified table.

If no FieldList is specified then there is an implicitly specified FieldList with all fields of the specified table in the order of the corresponding CreateTableStatement or CreateViewStatement, resp.

The number of Expressions in a ValueList or the number of fields of the result of the TableExpression must match the number of fields in the FieldList and the corresponding types must be compatible.

Each ValueList or each result record of the TableExpression, resp., represents a record which is inserted into the table.

If DEFAULT VALUES is specified, then a record consisting of the default value of each field is inserted.

3.17.1. Insertion with Fieldlist and DEFAULT Values

For each record t to be inserted, the i-th *`FieldName`* in the FieldList specifies to which table field the i-th field of t is assigned. For all fields of the table which are not specified in the FieldList and are not specified with AUTO_INCREMENT, the DEFAULT value of the field is inserted. If the field has no specified DEFAULT value but is defined on a domain with a DEFAULT value, then that DEFAULT value is used, otherwise the null value is inserted as a general fallback.

3.17.2. Insertion on AUTO_INCREMENT Fields

AUTO_INCREMENT fields can be considered as fields with a special DEFAULT mechanism. Unless they are explicitly specified, an automatic value generation is done such that the resulting key is unique. Details are described in section *AUTO_INCREMENT_Fields* .

3.17.3. Insertion on Views

If the specified table is a view, then insertion is effectively made into the underlying base table and all fields of the base table which are not in the view are filled up with their DEFAULT values.

For a view v where the WITH CHECK OPTION is specified the insertion of a record t fails if t does not fulfill the SearchCondition of the view definition of v or any other view on which v is based.

The InsertStatement returns an error if a type exception occurs (see *Type Exceptions and Overflow* [sql_type_exceptions]) or if a NULL constraint is violated. The InsertStatement fails if a key constraint or a UNIQUE constraint (defined by a `CREATE UNIQUE INDEX ...`) would be violated.

3.17.4. Handling of Key Collisions

A key collision is the event that one or several records of the Source have key values that already are present in the target table.

The 4 variants of the InsertStatement behave differently w.r.t. key collisions: The InsertIntoStatement returns an error on a key collision and no record is inserted.

The other 3 variants never return error. All records which do not conflict on the key are inserted and conflicting records are handled as follows:

The InsertIgnoreStatement ignores conflicting source records.

The InsertReplaceStatement replaces the record in the target by the conflicting record of the Source (one might think that the record in the target is deleted before the source record is inserted).

The InsertUpdateStatement replaces those fields of the target record which are specified in the FieldList with the corresponding values of the source records. Other field values of the target record remain unchanged.

Note that InsertReplaceStatement and InsertUpdateStatement work identically if no FieldList is specified.

In case of a FieldList, InsertReplaceStatement fills non specified fields with their default value, whereas InsertUpdateStatement leaves non specified fields unchanged.

3.17.5. Insertion with ReturningClause

If a ReturningClause is specified then the InsertStatement behaves like a scroll cursor on a SELECT statement. For each inserted record, a result record is delivered which is constructed according to the expressions in the clause. The expressions may refer to the field names of the target table, even if some of them do not appear in the FieldList of the INSERT statement. The arity of the ReturningClause is independent from the arity of the target table. Typically, the ReturningClause serves to catch field values of those fields whose values are assigned automatically via a default mechanism (sequence or auto_increment specification).

The RETURNING clause might also contain subqueries, but in case of a distributed INSERT, the table references of the RETURNING clause must be against the same database as the target table reference.

The current user must have INSERT privilege on the specified table.

If a TableExpression is specified, the user must additionally have the corresponding SELECT-privileges as if the TableExpression were run as a SelectStatement (see *SelectStatement*).

Locks: The table referenced by TableSpec is update locked automatically.

```
INSERT INTO suppliers VALUES (80, 'TAS', 'Munich')

INSERT INTO suppliers (name,suppno) VALUES
 ('Smith & Co', (SELECT MAX (suppno)+1 FROM suppliers) )

INSERT INTO suppliers TABLE
( (81,'xy','ab'),
  (82,'yz','bc')
)

INSERT INTO suppliers@otherdb@server5 (name,suppno)
VALUES ('Smith & Co',
       (SELECT MAX (suppno)+1 FROM
               suppliers@otherdb@server5) )

INSERT INTO suppliers SELECT * FROM suppliers2
```

Assume a table T with an INTEGER field "key" which should get its values from a sequence S (i.e. "key" has been specified with a DEFAULT S.nextval):

```
INSERT INTO T(t0,t1) VALUES (123 ,'asd') RETURNING(key)
```

The RETURNING clause makes the value assigned to the key field available for further processing.

Assume that table suppliers contains the record (80, 'TAS', 'Munich'):

```
INSERT INTO suppliers VALUES (80, 'TAS2', 'Berlin');
 -- returns error;

INSERT OR IGNORE INTO suppliers VALUES (80, 'TAS2', 'Berlin');
 -- returns no error but has no effect;

INSERT OR REPLACE INTO suppliers (suppno,address) VALUES (80,'Berlin');
 -- replaces record (80,'TAS','Munich')  by (80,NULL,'Berlin');

INSERT OR UPDATE INTO suppliers (suppno,address) VALUES (80, 'Berlin');
 -- replaces record (80,'TAS','Munich')  by (80, 'TAS', 'Berlin');
```

3.18. DeleteStatement

The DeleteStatement deletes records from a table or an updatable view.

Syntax:

[343] DeleteStatement ::= DELETE FROM TableSpec [CorrelationIdentifier] [WHERE SearchCondition]

[314] TableSpec ::= LocalTableSpec | RemoteTableSpec

Explanation: The table specified by `TableName` must be updatable (i.e. a base table or an updatable view).

All records from the specified table which satisfy the SearchCondition are deleted.

If the SearchCondition is omitted, all records from the specified table are deleted.

If records are deleted from an updatable view, the deletion is made on the underlying base table.

Note

It is allowed to refer to the table to be modified in a subquery of the DeleteStatement (in the Search-Condition). See the examples below and also *General Rule for Update* .

Deletion of many records may be slow if secondary indexes exist. However, deletion of all records in a table (WHERE clause is omitted) is very fast.

Privileges: The current user must have DELETE-privilege on the specified table.

If TableExpressions occur in the SearchCondition, the user must additionally have the corresponding SELECT-privileges as if the TableExpressions were run as SelectStatements (see *SelectStatement*).

Locks: The table referenced by `TableName` is update locked automatically.

If a remote table is specified as the target of the DELETE operation, all subqueries (if any) must specify tables residing on the same database. However, if the target table is local, any tables (remote or local) may be specified in subqueries.

```
DELETE FROM quotations

DELETE FROM suppliers
WHERE suppno = 70

DELETE FROM suppliers
WHERE suppno =
    (SELECT MAX (suppno)
```

```
    FROM suppliers@otherdb@server5)

DELETE FROM suppliers@otherdb@server5 s
WHERE NOT EXISTS
    (SELECT *
    FROM quotations@otherdb@server5
    WHERE suppno = s.suppno)
```

See also *General Rule for Updates* .

3.19. UpdateStatement

The UpdateStatement updates a set of records in a table or an updatable view.

Syntax:

[344]	UpdateStatement : : =	UPDATE TableSpec [CorrelationIdentifier] SET AssignList [WHERE { SearchCondition | CURRENT }]
[314]	TableSpec : : =	LocalTableSpec | RemoteTableSpec
[345]	AssignList : : =	Assignment [, Assignment]...
[346]	Assignment : : =	FieldIdentifier = Expression

Explanation:

The effect of the UpdateStatement is that all records of the specified table which fulfill the SearchCondition are updated. If no SearchCondition is specified then *all* records of the table are updated.

For each record to be updated the fields on the left hand sides of the Assignments are updated to the value of the corresponding Expression on the right hand side. Unspecified fields remain unchanged.

If the specified table is a view then the update is effectively made on the underlying base table and all fields of the base table which are not in the view remain unchanged.

For a view v where the WITH CHECK OPTION is specified the update of a record t fails if the updated record would not fulfill the SearchCondition of the view definition of v or any other view on which v is based.

The UpdateStatement fails if a NULL-Constraint or a key constraint or a UNIQUE constraint (defined by a CREATE UNIQUE INDEX ...) would be violated or if a type exception occurs (see *Type Exceptions and Overflow* [sql_type_exceptions]).

The specification of *CURRENT* is only allowed at the programming interface level for an *UPDPOS* call.

Note

It is allowed to update primary key fields, but this runs considerably slower than update of non-key fields only.

Note

It is allowed to refer to the table to be updated in a subquery of the UpdateStatement (in the AssignList or SearchCondition). See the example and also *General Rule for Update* .

Privileges:

The current user must have UPDATE-privilege on all fields specified on the left hand sides of the Assignments.

If there are TableExpressions on the right hand side of the Assignments or in the SearchCondition, the user additionally must have corresponding SELECT-privileges as if the TableExpressions were run as SelectStatements (see *SelectStatement*).

Locks:

The table referenced by `TableName` is update locked automatically.

If a remote table is specified as the target of the UPDATE operation, all subqueries (if any) must specify tables residing on the same database. However, if the target table is local, any tables (remote or local) may be specified in subqueries.

```
UPDATE quotations
SET price = price * 1.1, delivery_time = 10
WHERE suppno = 53 AND partno = 222

UPDATE quotations@otherdb@server5 q
SET price = price * 1.1
WHERE price =
    (SELECT MIN (price)
    FROM quotations@otherdb@server5
    WHERE suppno = q.suppno)
```

3.20. UpdateFromStatement

The UpdateFromStatement updates a target table from an existing or computed source table.

Syntax:

[347]	UpdateFromStatement ::=	UPDATE TargetTable [CorrelationIdentifier] [WITHOUT DELETE] FROM SourceExpression
[348]	TargetTable ::=	TableIdentifier
[349]	SourceExpression ::=	TableExpression

Explanation:

The target table must not be a view. Field number and field types of SourceExpression and TargetTable must be compatible.

If the WITHOUT DELETE option is not specified then the UpdateFromStatement updates the target table such that the records are identical to those of the source.

If the WITHOUT DELETE option is specified, the UpdateFromStatement merges target and source: each record from source whose field values on target's key positions do not exist in target is inserted. Each record from source whose field values on target's key position match a record in target replaces this matching record.

With the WITHOUT DELETE option, the UpdateFromStatement thus is a special case of the MergeStatement.

Caution

All triggers and referential constraints on the target table are ignored for this statement. Secondary indexes are maintained.

Privileges: The current user must have INSERT, UPDATE and DELETE privileges on the target table.

Locks: The table referenced by `TableName` is update locked automatically.

```
UPDATE quotations FROM quotations_new
UPDATE quotations WITHOUT DELETE FROM addquot@remotedb@remotehost
```

3.21. MergeStatement

The MergeStatement serves as a combination of the InsertStatement with the UpdateStatement. It combines the effects of both these statements within a single one.

Syntax:

[350]	MergeStatement	: : = MERGE INTO TargetTable USING SourceExpression ON (JoinPredicate) MatchClause NonMatchClause
[348]	TargetTable	: : = TableIdentifier
[349]	SourceExpression	: : = TableExpression
[351]	JoinPredicate	: : = SearchCondition
[352]	MatchClause	: : = WHEN MATCHED THEN UPDATE SET AssignList
[353]	NonMatchClause	: : = WHEN NOT MATCHED THEN INSERT [(FieldList)] VALUES ValueList
[341]	ValueList	: : = Expression [, Expression]...)

Explanation:

The table specified by TargetTable must be updatable, SourceExpression must be an expression delivering a set of records. The JoinPredicate refers to fields of TargetTable and SourceExpression.

The merge process can be seen as a loop executed on the SourceExression S. For each record of S there must be either no record or exactly one record in TargetTable which matches. In the first case, the NonMatchClause is executed which inserts fields of the current record of S into the TargetTable. In the second case, the MatchClause is executed to update fields of the matching record in TargetTable as specified in the AssignList.

Privileges: The current user must have INSERT and UPDATE privilege on TargetTable.

Locks: The TargetTable is update locked automatically.

```
MERGE INTO suppliers tar
     USING (SELECT * FROM newsuppliers) src
        ON (tar.suppno = src.suppno)
 WHEN MATCHED THEN UPDATE SET tar.address = src.address
 WHEN NOT MATCHED THEN INSERT VALUES (suppno, name, address)
```

3.22. General Rule for Updates

The semantics of all modification operations (INSERT, UPDATE, DELETE) is that of a *deferred update*, i.e. conceptually the modification is performed in two phases:

1. Compute the whole modification information (records to be inserted, records to be changed and their values to replace existing records, records to be deleted, resp.). In this phase the table to be modified remains unchanged.

2. Execute the modification.

This principle allows to specify the modification referring to the old state of the table and defines the modification as an atomic step.

3.23. Rules of Resolution

For the semantics of nested QueryBlocks it is necessary that any FieldReference and any SetFunction can be resolved against exactly one QueryBlock. This block is then called the resolution block of the Field or the SetFunction, resp.

3.23.1. Resolution of Fields

An unqualified Field *fld* (see *FieldReference*) has as its resolution block the innermost surrounding QueryBlock q whose FROM-clause contains a TableReference which exports a field named *fld*. If there is more than one such TableReference in q, the resolution fails and an error is returned.

A qualified Field *r.fld* has as its resolution block the innermost surrounding QueryBlock q whose FROM-clause contains a `TableName` or `CorrelationName` *r*. If the corresponding TableReference does not export a field named *fld*, the resolution fails and an error is returned.

3.23.2. Resolution of SetFunctions

In most cases, the resolution block of a SetFunction is the innermost surrounding QueryBlock.

```
SELECT partno
FROM quotations
GROUP BY partno
HAVING
    MAX(price) - MIN(price)
    >
    AVG(price) * 0.5
```

The resolution of all three SetFunctions is the only (and innermost) QueryBlock. Thus, they are computed for each group of parts.

In general, the resolution of *count(*)* is always the innermost surrounding QueryBlock; for all other forms of SetFunctions, the resolution block is the innermost resolving QueryBlock over all fields inside the SetFunction.

```
SELECT partno
FROM quotations q1
GROUP BY partno
HAVING
    (SELECT COUNT(*) FROM quotations q2
    WHERE q2.partno = q1.partno AND
    q2.price = MIN (q1.price))
    > 1
```

Here, *COUNT(*)* refers to the inner QueryBlock whereas *MIN(q1.price)* refers to the outer QueryBlock and thus computes as the minimum price over the current group of parts.

Arbitrary (single-valued) Expressions are allowed as arguments of SetFunctions. It is even allowed to nest SetFunctions as long as the resolution block of the inner SetFunction surrounds the resolution block of the outer SetFunction.

For each SetFunction s with resolution block q, s must not appear in the WHERE-clause of q.

4. Persistent Stored Methods

The PSM extensions enhance Transbase SQL with procedural language elements as described in the SQL/PSM Standard.

As we only allow Functions and Procedures to be stored persistently in the database we call these extensions 'Persistent Stored Methods' as opposed to 'Persistent Stored Modules' used in the SQL Standard.

These extensions can be used inside PSM blocks and thus in persistent procedures and functions defined using these PSM blocks.

PSM Blocks, however, can not only appear as defining bodies of functions and procedures but can be supplied as single SQL statements. In this case the entire PSM block is processed as a single SQL statement.

[354] PSM_MethodDeclaration ::= PSM_CreateFunction | PSM_CreateProcedure

4.1. Functions

4.1.1. CREATE FUNCTION Statement

The CREATE FUNCTION Statement is used to define functions that are stored in the database and can be incorporated in arbitrary SQL statements where an expression of the specified function return type is permitted.

Functions are defined by their identifiers only. Polymophic function definitions are not supported in this version.

The creator of the function becomes its owner regardless of the schema it belongs to. Only the owner has the right to manipulate or drop the function.

[355] PSM_CreateFunction ::= CREATE FUNCTION FuncIdentifier FuncDeclaration
[356] FuncIdentifier ::= Identifier
[357] FuncDeclaration ::= ([FuncParameterList]) RETURNS DataTypeSpec AS PSM_Block
[358] FuncParameterList ::= FuncParameter [, FuncParameterList]
[359] FuncParameter ::= SimpleIdentifier DataTypeSpec

The actual function definition consists of a function signature and a defining body that is essentially a PSM block. The signature is defined by a - possibly empty - list of parameters of arbitrary TB/SQL types and a return type. All parameters are input parameters.

The following example defines a function that computes the `n`th Fibonacci number in the schema `math` which needs to be available.

```
create function math.fib(n integer) returns bigint as
  DEclare
    sum  bigint := 1;
    sum0 bigint := 0;
    sum1 bigint := 1;
    cnt integer := 1;
  begin
    if n = 0 then return 0;
    elsif n = 1 then return 1;
    end if;
```

```
    while cnt < n
      loop
        sum := sum0 + sum1;
        sum0 := sum1;
        sum1 := sum;
        cnt := cnt + 1;
    end loop;
    return sum;
  end;
```

4.1.2. DROP FUNCTION Statement

This statement drops a persistent PSM function.

The user must be the owner of the function or have USERCLASS DBA.

[360] PSM_DropFunctionStatement : : = DROP FUNCTION Identifier

```
drop function fib;
```

4.2. Procedures

4.2.1. CREATE PROCEDURE Statement

[361]	PSM_CreateProcedure	: : = CREATE PROCEDURE Identifier ProcDeclaration
[362]	ProcDeclaration	: : = ([ProcParameterList]) AS PSM_Block
[363]	ProcParameterList	: : = ProcParameter [, ProcParameterList]
[364]	ProcParameter	: : = [INOUT_Spec] SimpleIdentifier DataTypeSpec]
[365]	INOUT_Spec	: : = IN | OUT | INOUT

The following procedure records the timestamp of the last update for each user. It could be used in triggers for some tables.

```
create procedure tbadmin.lastupdate(out last timestamp) as
declare
 ts datetime := currentdate;
begin
  update tbadmin.lastupdate set latest = ts, count = count+1 where login = user;
  update tbadmin.lastupdate set latest = ts, count = count+1 where login = 'all';
end;
```

4.2.2. DROP PROCEDURE Statement

This statement drops a persistent PSM procedure.

[366] PSM_DropProcedureStatement : : = DROP PROCEDURE Identifier

The following example drops the procedure `lastUpdates`.

```
drop procedure lastUpdate;
```

4.3. Usage Privileges

Persistent stored methods belong to the creator who has the exclusive right to manipulate and use these objects. The right to use these methods can be granted to other users or to PUBLIC, i.e. to any user present or created later.

4.3.1. GRANT USAGE Statement

```
grant usage on procedure lastUpdate to public;
```

[367] GrantUsageStatement : : = GRANT USAGE ON
{ FUNCTION | PROCEDURE } Identifier
TO UserList [WITH GRANT OPTION]

4.3.2. REVOKE USAGE Statement

```
revoke usage on procedure lastUpdate from public;
```

[368] RevokeUsageStatement : : = REVOKE USAGE
ON { FUNCTION | PROCEDURE } Identifier
FROM UserList

4.4. PSM Block Statements

```
## outputx into "psmoutput.txt"
## trace into "pmstrace.txt"
<< testblock >>
declare
  a string := "Hello world!";
begin
  outputx( a );
end
```

[369] PSM_Block_Statement : : = [PSM_Directives]
PSM_Block

4.4.1. PSM blocks and Transactions

In the context of transaction handling PSM blocks are considered to be single SQL statements. Therefore transaction control is only possible outside PSM blocks and PSM methods.

In autocommit transactions SQL statements executed in PSM blocks are not committed separately - as one might expect. The entire PSM block is treated as a single SQL statement that is committed or rolled back in total. So this is also the case for nested PSM blocks.

5. PSM Blocks

PSM blocks are not only the main parts of persistent stored methods. PSM block statements can also be submitted as SQL statements in any Transbase frontend or application program.

[370] PSM_Block : : = [PSM_Label]
[DECLARE PSM Declaration...]
BEGIN
PSM_Statement...
PSM_ExceptionClause...
END ;

[371] PSM_Label : : = << PSM_Identifier >>

PSM_Directives are covered in the section on *Debugging and Tracing*.

Thus SQL statements with procedural elements as e.g. conditional statements or loops can also be submitted on the fly or even be constructed dynamically.

```
BEGIN
  IF EXISTS ( SELECT ...)
  THEN
    UPDATE ...
  ELSE
    INSERT ...
  END IF;
END;
```

5.1. PSM Declarations

[372] PSM Declaration : : = PSM_LocalMethodDeclaration |
PSM_TypeDeclaration |
PSM_VariableDeclaration |
PSM_CursorDeclaration

[377] PSM_VariableDeclaration : : = PSM_Variable PSM_DataTypeSpec
[:= Expression] ;

[373] PSM_CursorDeclaration : : = PSM_Identifier CURSOR FOR SelectStatement ;

```
DECLARE
  x0 INTEGER := 20+3;
  y0 DATE := currentdate;
  z0 VARCHAR(*) := 'hu';
  C1 CURSOR FOR SELECT tname, colno FROM systable WHERE colno = x0;
BEGIN
   ...
END
```

5.1.1. Local Method Declarations

Local method declarations are valid only inside the PSM block they are declared in. They are identified by a simple name as e.g. variables. Apart from that there is no difference between persistent and local methods.

[374] PSM_LocalMethodDeclaration ::= PSM_LocalFunctionDeclaration |
PSM_LocalProcedureDeclaration

5.1.1.1. Local Function Declarations

[375] PSM_LocalFunctionDeclaration ::= FUNCTION PSM_Identifier FuncDeclaration
[357] FuncDeclaration ::= ([FuncParameterList]) RETURNS DataTypeSpec
AS PSM_Block
[358] FuncParameterList ::= FuncParameter [, FuncParameterList]
[359] FuncParameter ::= SimpleIdentifier DataTypeSpec

```
declare
  nums integer[]

  function sumarray(arr integer[]) returns bigint
  declare
    ssum bigint := 0;
  begin
    FOR ii IN 1 .. UPB(arr)
    loop
      if null<> arr[ii] then
        ssum := ssum + arr[ii];
      end if;
    end loop;
    return ssum;
  end;
begin
end
```

5.1.1.2. Local Procedure Declarations

[376] PSM_LocalProcedureDeclaration ::= PROCEDURE PSM_Identifier ProcDeclaration
[362] ProcDeclaration ::= ([ProcParameterList])
AS PSM_Block
[363] ProcParameterList ::= ProcParameter [, ProcParameterList]
[364] ProcParameter ::= [INOUT_Spec] SimpleIdentifier DataTypeSpec]
[365] INOUT_Spec ::= IN | OUT | INOUT

5.1.2. PSM Variables

PSM data types are the types that parameter and variables inside PSM blocks can assume. In addition to the standard TB/SQL types these can also be arrays of a single type or records with mixed type components.

Variables are introduced in the DECLARE section of a PSM_Block. Transbase supports different kinds of variables:

1. *Simple Variables*: They can assume any TB/SQL type.

2. *Arrays*: They represent arrays of arbitrary TB/SQL types. Their length can be fixed or dynamic.

3. *Records*: They represent records of mixed TB/SQL types. Record variables refer to a preceding record type definition defining the number, names and types of their components.

4. *Cursor Variables*: They represent the results of select queries that can be used to process these results on a record by record base.

The initial value of simple variables or the components of arrays or records is never undefined. If they are not initialised explicity the initial value is always the SQL `NULL` value.

5.1.2.1. Simple Variables

```
DECLARE
  s string;
  d date := currentdate;
  n integer := 17;

BEGIN
  -- variable assignments:
  n := 1;
  s := 'Hello!';

  n := sqr(17);
  n := n*n;
  s := 'Now is ' || currentdate cast string;

  s := select min(tname) from systable;
END
```

The right side of a variable assignment can be any expression that yields a value that this variable can hold. The result of the expression need not match the type of the variable exactly. It is only necessary that the result can be converted implicitly into the variable's type.

[377] PSM_VariableDeclaration ::= PSM_Variable PSM_DataTypeSpec
[:= Expression] ;
[378] PSM_Variable ::= PSM_Identifier
[398] PSM_Identifier ::= SimpleIdentifier

[379] PSM_DataTypeSpec ::= DataTypeSpec |
PSM_ArrayTypeSpec |
PSM_RecordTypeSpec |

[380] PSM_Assignment ::= PSM_Identifier := Expression ;

Variables are simply referenced by their identifiers.

5.1.2.1.1. Predefined Variables

Name	Type	Description
FOUND	bool	`true` if a record is available after the last FETCH statement. `false` when there are no more records.
ROWCOUNT	bigint	number of records processed after INSERT or UPDATE or DELETE statement

Name	Type	Description
SQLSTATE	string	The SQLSTATE as defined in the SQL standard after the last statement
PROCESSING_LINE	integer	last row of source code executed

5.1.2.2. PSM Arrays

PSM arrays can either be defined with a fixed length specified in the type declaration or as dynamic arrays without fixed size.

```
DECLARE

  -- definition of array variables:
  as1 string[7];  -- fixed length
  as2 string[];   -- dynamic length

  -- definition of record type variables:
  TYPE RecPV is RECORD ( pos integer; val bigint := 17; );
  pi0 p := ( 0, 0 );
  pi1 p := ( 1, 1 );
  pi2 p;

  -- definition of cursor variables:
  c1 CURSOR FOR SELECT username as "schema", tname as "table" FROM systable, sysuser WHERE "schema" = userid;

BEGIN
  ...
END
```

In both cases the function UPB(<PSM array>) yields the upper bound of the supplied array.

Array indexes start with 1 and end with length.

Dynamic arrays are extended implicitly when a component beyond the upper bound is written.

Read access to components beyond the upper bound always yields NULL as a result. The upper bound is not increased.

Read access to components not initialized also yields NULL.

[381] PSM_ArrayTypeSpec : : = DataTypeSpec [[Expression]]
[1] [: : = [
[2]] : : =]

5.1.2.3. PSM Records

```
DECLARE

  -- definition of record type variables:
  TYPE RecPV is RECORD ( pos integer, val bigint );
  pi0 p := ( 0, 0 );
  pi1 p := ( 1, 1 );
  pi2 p;

BEGIN
  -- ...
END
```

[382] PSM_TypeDeclaration : : = TYPE PSM_Identifier [IS RECORD] (PSM_RecComponent [, PSM_RecComponent]...)
[383] PSM_RecComponent : : = PSM_Identifier { DataTypeSpec | PSM_RecordTypeSpec }
[384] PSM_RecordTypeSpec : : = PSM_Identifier

5.1.3. PSM cursors

5.1.3.1. Declarations

```
DECLARE

  -- definition of cursor variables:
  c1 CURSOR FOR SELECT username as "schema", tname as "table" FROM systable, sysuser WHERE "schema" = userid;

BEGIN
  ...
END
```

[373] PSM_CursorDeclaration : : = PSM_Identifier CURSOR FOR SelectStatement ;

5.1.3.2. Cursor Statements

[385] PSM_Cursor_Statement : : = PSM_OPEN_Statement | PSM_FETCH_Statement | PSM_CLOSE_Statement
[386] PSM_OPEN_Statement : : = OPEN [CURSOR] PSM_Identifier ;
[387] PSM_FETCH_Statement : : = FETCH [CURSOR] PSM_Identifier INTO PSM_Identifier [, PSM_Identifier]... ;
[388] PSM_CLOSE_Statement : : = CLOSE [CURSOR] PSM_Identifier ;

The PSM_OPEN_Statement executes the SelectStatement specified in the declaration of the cursor.

With subsequent PSM_FETCH_Statements the rows in the cursor are processed. Each call of a PSM_FETCH_Statement advances the cursor to the next row. When no more rows are available the prededfined variable FOUND becomes false. This variable should always be checked before the result variables used in the PSM_FETCH_Statement are accessed.

The PSM_CLOSE_Statement closes the cursor and frees all resources bound by it. PSM blocks should be structured in such a way that cursors are always closed.

```
DECLARE
  s0 STRING := '';
  i0 INTEGER := 0;
  C1 CURSOR FOR SELECT tname, colno FROM systable order by colno desc;
BEGIN
  OPEN CURSOR C1;
  FETCH C1 INTO s0, i0;    -- Automatic Type Adaptions
  IF FOUND THEN           -- Predefined variable becomes FALSE when no more rows available
    processRecord(z0,y0);
  ELSE
```

```
    CLOSE C1;
  END IF;
END
```

Alternative using *PSM Records*:

```
DECLARE
  TYPE recSI IS RECORD ( s0 STRING, i0 INTEGER );
  CurRec recSI;
  C1 CURSOR FOR SELECT tname, colno FROM systable order by colno desc;
BEGIN
  OPEN CURSOR C1;
  FETCH C1 INTO CurRec;    -- Automatic Type Adaptions
  IF FOUND THEN           -- Predefined variable becomes FALSE when no more rows available
    processRecord(CurRec.s0, CurRec.i0);
  ELSE
    CLOSE C1;
  END IF;
END
```

5.2. PSM Statements

This sections describes all the statements that can appear in a PSM block

[389] PSM_Statement : : = PSM_Block |
SQLStatement |
PSM_Assignment |
PSM_IF_Statement | PSM_CASE_Statement |
PSM_LOOP_Statement |
PSM_CONTINUE_Statement | PSM_EXIT_Statement |
PSM_FuncRETURN_Statement | PSM_ProcRETURN_Statement
|
PSM_Cursor_Statement |
PSM_EXECUTE_Statement

[390] PSM_FuncRETURN_Statement : : = RETURN Expression ;

[391] PSM_ProcRETURN_Statement : : = RETURN ;

PSM_FuncRETURN_Statements can only appear inside function declarations.

PSM_ProcRETURN_Statements can only appear inside procedure declarations.

PSM_CONTINUE_Statements and PSM_EXIT_Statements can only appear inside PSM_LOOP_Statements.

PSM_Blocks can be nested.

A SQLStatement is an arbitrary SQL statement from the DDL or DML section.

5.2.1. SQL Statements

PSM_Blocks contain arbitrary SQL statements as described in the sections on DDL or DML statements. The only significant difference is that Expressions in these statements can contain a number of additional SimplePrimary elements that we compiled into PSM_Primary category for clarity.

Additionally some special statements may appear that are allowed inside PSM_Blocks only. So SQLStatement

[392] SQLStatement : : = <any SQL statement from DDL or DML Section>
[393] PSM_Primary : : = PSM_Parameter |
PSM_Variable |
PSM_CursorField |
PSM_ArrayField |
PSM_RecordField |
[394] PSM_Parameter : : = PSM_Identifier
[395] PSM_CursorField : : = PSM_Identifier . Identifier
[396] PSM_ArrayField : : = PSM_Identifier [Expression]
[397] PSM_RecordField : : = PSM_Identifier . PSM_Identifier
[398] PSM_Identifier : : = SimpleIdentifier

5.2.2. IF Statements

[399] PSM_IF_Statement : : = IF Condition THEN PSM_Statement...
[ELSIF Condition THEN PSM_Statement...]...
[ELSE PSM_Statement...]
END IF ;
[400] Condition : : = SearchCondition

```
IF x0 < 4 THEN
  x0 := x0 + 2;
ELSIF x0 < 10 THEN
  x0 := x0 + 1;
ELSE
  x0 := x0 / 2;
END IF;
```

5.2.3. CASE Statements

[401] PSM_CASE_Statement : : = CASE [WHEN Condition THEN PSM_Statement...]... ELSE PSM_Statement...
END CASE ;

```
CASE
WHEN x0 < 4 THEN
  x0 := x0 + 2;
WHEN x0 < 10 THEN
  x0 := x0 + 1;
ELSE
  x0 := x0 / 2;
END CASE;
```

5.2.4. LOOP Statements

The general LOOP statement provides the most flexible iteration of the statements provided in the LOOP body.
[402] PSM_LOOP_Statement : : = [PSM_Label]

[PSM_WHILE_Clause | PSM_FOR_Clause | PSM_Cursor_Clause]
PSM_BasicLoop

[403] PSM_BasicLoop : : = LOOP
PSM_Statement...
END LOOP ;

[404] PSM_CONTINUE_Statement : : = CONTINUE [WHEN Condition]

[405] PSM_EXIT_Statement : : = EXIT [PSM_Label_Reference][WHEN Condition]

[406] PSM_Label_Reference : : = PSM_Identifier

A PSM_CONTINUE_Statement causes the subsequent statements in the loop body to be skipped when the specified expression in the WHEN clause evaluates to `true` or no such clause is present. Processing continues with the next iteration of the statements in the loop body unless a preceding WHILE of FOR clause terminates the loop.

An PSM_EXIT_Statement causes the loop to be abandoned immediately when the specified expression in the WHEN clause evaluates to `true` or no such clause is present. Processing continues with the next PSM_Statement after the loop if present.

CONTINUE and EXIT statements may appear inside loop statements only.

5.2.4.1. Basic LOOP Statement

The most simple LOOP statement consists of the basic loop only without preceding clauses controlling the repetitions in the loop body.

Termination of the loop is achieved with PSM_EXIT_Statements only. The user needs to take care that the loop is terminated in any case.

[403] PSM_BasicLoop : : = LOOP
PSM_Statement...
END LOOP ;

```
declare
  x0 integer := 0;
  N  integer := 1;
begin
  loop
    exit when N > 10
    x0 := x0 + N;
    N  := N + 2
  end loop;          --> x0 = 25
end;
```

5.2.4.2. Conditional LOOP Statement

When a WHILE clause is present then these statements are executed as long as the specified condition evaluates to TRUE at the beginning of each iteration or an PSM_EXIT_Statement terminates the loop.

[407] PSM_WHILE_Clause : : = [WHILE Condition

[403] PSM_BasicLoop : : = LOOP
PSM_Statement...

END LOOP ;

```
declare
  x0 integer := 1;
begin
  while N <= 10      -- variable N = 1,3,5,7,9
  loop
    x0 := x0 + N;
    N  := N + 2
  end loop;          --> x0 = 25
end;
```

This example is completely equivalent to the preceding example with the PSM_IF_Statement

5.2.4.3. Counted LOOP Statements

The FOR statement is loop statement with a PSM_Identifier. The statements in the loop body are executed with different values of the implicitly defined loop variable.

[408]	PSM_FOR_Clause ::=	FOR PSM_Identifier IN Start .. End [BY Step]
[403]	PSM_BasicLoop ::=	LOOP PSM_Statement... END LOOP ;
[409]	Start ::=	Expression
[410]	End ::=	Expression
[411]	Step ::=	Expression

The loop variable is defined implicitly and may not be declared beforehand. It assumes the values specified by the Start, End and the optional Step expression that defaults to 1 if not present.

The statements in the loop body are executed once for each instance of the loop variable. In the first iteration the loop variable assumes the Start value. Before each following operation it is incremented by Step. The loop is executed until the loop variable has passed the End value.

```
declare
  x0 integer := 0;
begin
  for N in 1 .. 10 by 2      -- variable N = 1,3,5,7,9
  loop
    x0 := x0 + N;
  end loop;
  ...                        --> x0 = 25
end;
```

In the above example the loop variable `N` does not exactly reach the end value `10` but the loop is terminated when it becomes bigger than the end value.

5.2.4.4. Cursor LOOP Statements

Cursor loop statements provide a convenient and simple way to process the results of select statements in a row by row fashion.

[412]	PSM_Cursor_Clause ::=	FOR PSM_Identifier IN SelectStatement

The cursor variable is declared implicitly in this statement and need not be declared in the declare section of the surrounding PSM block. The fields in the result record being processed are referenced simply by qualifying the field name with the cursor name.

The PSM statements are executed for each record in turn. No explicit cursor commands are allowed or necessary.

```
for C2 in (SELECT * from "systable" where colno > z0)
  loop
    c2colno := C2.colno;      --  <cursor>.<column>
    c2tname := C2.tname;
    ...
  end loop;
```

5.2.5. Execute Statements

Execute Statements provide the ability to execute dynamically created SQL statements. So this is the most flexible way to create and execute statements that completely depend on the data gathered at the point of execution.

[413] PSM_EXECUTE_Statement ::= EXECUTE Expression ;

The specified Expression must evaluate to a correct SQL statement in the runtime context of the PSM_EXECUTE_Statement.

PSM blocks can also contain DDL statements. The following sequence of statements, however, does not work because the table T is not defined when the PSM block is compiled.

```
create table T (...);
insert into T (...) values (...);
```

In order to avoid this problem the insert statement is evaluated in a completely dynamic way.

```
create table T (...);
EXECUTE 'insert into T (...) values (...)';
```

By putting the table creation in an EXECUTE statement tables can be created in a completely data driven manner.

```
DECLARE
  tname string;
BEGIN
  tname := ...
  EXECUTE 'create table ' || tname || ...);
  EXECUTE 'insert into ' || tname || (...) values (...)';
END;
```

5.3. Exception Clauses

Exception clauses enable the PSM program to handle certain runtime exceptions or all such exceptions that may occur during the evaluation of a PSM block.

[414] PSM_ExceptionClause ::= EXCEPTION
[WHEN Condition THEN PSM_Statement...]...

```
BEGIN
insert into xy values (....)
EXCEPTION
  WHEN SQLSTATE = 23505
    THEN insert into xy_rejected values( currentdate, ... )
END;
```

5.4. Debugging and Tracing

In order to provide some debugging capabilities to the application developer PSM blocks can be equipped with some directives to write debug and trace output to specified files.

[415] PSM_Directives : : = [PSM_Directive]...
[416] PSM_Directive : : = { PSM_Output_Directive | PSM_Trace_Directive }...

The output directives can not be altered dynamically inside PSM blocks. Therefore they are intended for development purposes only and should not appear in production code.

5.4.1. Debug Output

In order to debug PSM methods and PSM blocks the values of variables can be written to a PSM output file. The actual values of the variables specified in the output statments are written to the specified file.

[417] PSM_Output_Directive : : = ## OUTPUTX INTO FileLiteral
[418] PSM_OutputStatement : : = outputx(PSM_Identifier [, PSM_Identifier]...)

The PSM_Output_Directive specifies the file the output is written to.

Currently only variables can be written to the PSM output file.

5.4.2. Trace Output

Trace ouput protocols the statements executed inside a PSM block in the specified file.

[419] PSM_Trace_Directive : : = ##TRACE INTO FileLiteral

The PSM_Trace_Directive specifies the file the PSM trace output is written to.

6. Load and Unload Statements

In addition to the TB/SQL language, the Transbase Publishing System (formerly called Transbase CD) offers language constructs to control the data transfer from Retrieval Databases to the disk cache and to unload the disk cache.

These constructs are called *Load Statements* [tbcd.xhtml#ch_Switch_Load_Statements] and *Unload Statements* [tbcd.xhtml#ch_Unload_Statement].

They are only relevant and valid in Retrieval Databases and are explained in the *Transbase Publishing Guide* [tbcd.xhtml].

7. Alter Session statements

ALTER SESSION Statements serve to configure runtime parameters of a Transbase session. They are not subject to the transaction concept; i.e. they can be issued inside or outside a transaction.

They are entered in interactive applications or used in application programs like any other SQL statement.

The effect of a ALTER SESSION statement is restricted to the session, i.e. the connection which issues it.

7.1. Sort Buffer Size

This statement serves to configure the size of the main memory buffer for sorting operations (relevant for ORDER BY, GROUP BY, sort merge joins etc.). The configuration comes into effect at the start of the next transaction.

Syntax:

```
[420]   AlterSessionSortbufferStmt ::= ALTER SESSION SET SORT_BUFFER_KB = Size
[421]                         Size ::= IntegerLiteral
```

Explanation: Size is the desired size of the sortercache in KB. Big sizes favour the computing time for sorter operations but need more resources.

The default size of the buffer is 2048 KB.

Note that the feasibility of sorter operations does not depend on the size of the buffer.

Note that the reconfiguration of the sortercache is deferred until the start of the next transaction. For the sake of clarity, it is thus recommended to issue that statement outside of a transaction.

7.2. Multithreading Mode

This statement sets the Transbase query optimizer to one of several possible levels of parallel query execution. This setting is valid throughout the current session.

Syntax:

```
[422]   AlterSessionMultithreadStmt ::= ALTER SESSION SET QUERY_THREADING = Querythread-
                                        Config
[423]             QuerythreadConfig ::= OFF | MEDIUM | HIGH
```

Explanation:

In mode MEDIUM or HIGH, each suitable query is processed by several threads which run in parallel. The execution time may decrease considerably. This might be at the expense of other processes running on the same machine.

Without any AlterSessionMultithreadStmt or with mode OFF, queries are processed with the corresponding threading mode defined by the creation of the database.

Without multithreading Transbase guarantees that data is always processed and returned in the same deterministic sort order, even if no *ORDER BY* is specified. The SQL specification does not demand any reproducible sort order if no *ORDER BY* is used. With querythreading switched to *MEDIUM* or *HIGH* it is likely that data is returned

in different orders. Thus a query will return the same result but possibly in different order if no *ORDER BY* is specified. Only the specification of an *ORDER BY* guarantees a deterministic result sort order.

Setting Transbase for maximum parallalism:

```
ALTER SESSION SET QUERY_THREADING = HIGH
```

7.3. Integer Division Mode

This statement specifies the processing of an arithmetic division of two INTEGER values.

According to the SQL standard, an INTEGER division delivers INTEGER again, i.e. the fractional part is omitted.

With the following statement, the result of integer division can be defined to be a NUMERIC which also contains the fractional part. With its counterpart, one can return to the standard behaviour.

Syntax:

[424] AlterSessionIntdivStmt ::= ALTER SESSION SET INTEGER_DIVISION =
{ NUMERIC | STANDARD }

7.4. Lock Mode

This statement specifies a fixed lock granularity or the default locking strategy of Transbase. These statements *do not* lock objects but influence the lock granularity of the automatic locking of Transbase.

Syntax:

[425] AlterSessionLockmodeStmt ::= ALTER SESSION SET LOCK_MODE = Lockgran
[426] Lockgran ::= PAGES | TABLES | MIXED

Explanation: If PAGES is specified, all subsequent locks are on page basis.

If TABLES is specified, all subsequent locks are on table basis. In this mode, at most one lock is set for a table including all its secondary indexes and its LOBs.

If MIXED is specified, the locking strategy is chosen implicitly by the system which also is the default.

Note

These statements do not lock any objects. Carefully distinguish the SET LOCKMODE statements from the LOCK and UNLOCK statements which set TABLE locks on random tables.

7.5. Evaluation Plans

This statement enables and disables the generation of evaluation plans.

Syntax:

[427] AlterSessionPlansStmt ::= ALTER SESSION SET EVALUATION_PLANS =
{ COUNTERS | TIMES | OFF }

Explanation: If *PLANS* are enabled a textual query execution plan (QEP) is generated when a query is run. The QEP is stored in the scratch directory of the database but also can be retrieved into the application via the appropriate API call.

If mode *COUNTERS* is enabled, each operator of the QEP contains the number of records processed by that part.

If mode *TIMES* is enabled, each operator of the QEP additionally contains the elapsed time of processing the operator.

Switching to the mode *OFF* ends the QEP generation.

Note

Tracing the execution times via mode *TIMES* may cause a significant overhead in the query's total elapsed time. The times denoted in the produced QEP are adjusted to compensate this additional overhead.

7.6. Schema Default

This statement serves to change the default schema of the database for the current connection.

Syntax:

[428] AlterSessionSchemaStmt ::= ALTER SESSION SET SCHEMA_DEFAULT = SchemaIdentifier
[9] SchemaIdentifier ::= UserIdentifier
[8] UserIdentifier ::= Identifier | PUBLIC | USER

Explanation: If an object is specified without schema information, the default schema of the database is used. This ALTER SESSION statement can change the default setting temporarily for a connection.

7.7. SQL Dialect

This statement serves to adapt the SQL to the dialect of ODBC or MySQL.

The only difference that we are currently aware of is that single quotes in string literals are escaped using a backslash character in MySQL dialect.

In the Transbase dialect they are escaped by doubling them as specified in the SQL standard.

```
[Transbase]
'It''s cold outside!'
[MySQL]
'It\'s cold outside!'
```

Syntax:

[429] AlterSessionSetDialectStmt ::= ALTER SESSION SET DIALECT = SqlDialect
[430] SqlDialect ::= MYSQL | TRANSBASE

Explanation: The default dialect of a session is TRANSBASE

8. Lock Statements

Transbase locks database objects (i.e. pages or tables or views) automatically. If, however, explicit control of locking is needed, Transbase allows to lock and unlock objects explicitly with table locks.

Two statements, namely a LockStatement and an UnlockStatement, are provided for that purpose.

Carefully distinguish the LOCK and UNLOCK statements which set TABLE locks on random tables from the *ALTER SESSION SET LOCKMODE statement* which influences the lock granularity for automatic locking by Transbase.

8.1. LockStatement

Serves to explicitly lock tables and views.

Syntax:

[431]	LockStatement : : = LOCK LockSpec [, LockSpec]...
[432]	LockSpec : : = LockObject LockMode
[433]	LockObject : : = TableIdentifier | ViewIdentifier
[434]	LockMode : : = READ | UPDATE | EXCLUSIVE

Explanation: For each LockSpec, the specified lock is set on the specified object. If a view is specified, the lock is effectively set on the underlying base table(s).

For the semantics of locks see *Transbase System Guide* [system.xhtml].

Privileges: The current user needs SELECT-privilege on the specified objects. System tables (the data dictionary) cannot be locked explicitly.

```
LOCK suppliers READ, quotations UPDATE
```

8.2. UnlockStatement

Serves to remove a READ lock.

Syntax:

[435]	UnlockStatement : : = UNLOCK LockObject
[433]	LockObject : : = TableIdentifier | ViewIdentifier

Explanation: The specified object is unlocked, i.e. implicitly set or explicitly requested locks are removed.

Error occurs if the object is not locked or if the object is update locked, i.e. an *InsertStatement*, *UpdateStatement*, *DeleteStatement* or an explicit *LockStatement* with UPDATE or EXCLUSIVE mode has been issued within in the transaction.

9. The Data Types Datetime and Timespan

9.1. Principles of Datetime

The data type DATETIME is used to describe absolute or periodic points in time with a certain precision. A datetime value is composed of one or more components. For example, the birthday of a person consists of a year, a month and a day and is an absolute point in time with a certain precision. If the hour and minute of the birth is added, then the absolute point in time is described with a higher precision. Examples for periodic points in time are the 24-th of December, the birthday of a person without the year indication, or 12:00:00 (twelve o'clock).

9.1.1. RangeSpec

The occupied components of a datetime value constitute its range. The components have symbolic names. All names occur in 2 equivalent variants which come from the original Transbase notation and from the evolving SQL standard. An overview is given in the following table.

Notation	Meaning	Allowed Values
YY	year	1 - 32767
MO	month	1 - 12
DD	day	1 - 31
HH	hour	0 - 23
MI	minute	0 - 59
SS	second	0 - 59
MS	millisecond	0 - 999

The range indices MS, …, YY are ordered. MS is the smallest, YY is the highest range index. An explicit range spec is written as *[ub:lb]* where ub is the upperbound range and lb is the lowerbound range. For example *[YY:DD]* is a valid range spec. *[DD:YY]* is an invalid range spec.

Upperbound and lowerbound range may be identical, i.e. of the form *[ab:ab]*. A datetime value with such a range has a single component only. The corresponding range spec can be abbreviated to the form *[ab]*.

9.1.2. SQL Compatible Subtypes

The SQL standard defines the following subtypes which also are supported by Transbase:

Table 9.1. SQL Types for Datetime

SQL Type	Transbase Equivalent
DATE	DATETIME[YY:DD]

SQL Type	Transbase Equivalent
TIME	DATETIME[HH:SS]
TIMESTAMP	DATETIME[YY:MS]

For each case, the notations are equivalent.

Note that there are types that can be defined in Transbase but have no equivalent SQL standard formulation. DATETIME[MO:DD] describes yearly periodic datetimes based on day granularity, e.g. birthdays. DATETIME[HH:MI] describes daily periodic datetimes based on minute granularity, e.g. time tables for public traffic.

9.1.3. DatetimeLiteral

DatetimeLiteral defines the syntax for a constant value inside the SQL query text or inside a host variable of type char[]. Transbase supports a variety of literal notations, namely a native Datetime literal representation with maximal flexibility, the SQL standard representation and a notation compatible with the ODBC standard. All 3 variants are sufficiently explained by the following examples. A fourth variant is supported in a Transbase spoolfile.

Table 9.2. Variants of Timestamp Literals

Variant	Notation
Native TB	DATETIME[YY:MS](2002-12-24 17:35:10.025) DATETIME(2002-12-24 17:35:10.025)
SQL	TIMESTAMP '2002-12-24 17:35:10.025'
ODBC	{ ts '2002-12-24 17:35:10.025' }
Spoolfile only	'2002-12-24 17:35:10.025'

- In contrast to the colon separators in the time part the dot separator between SS and MS has the meaning of a fractional point.
- All variants except ODBC are also supported in a spoolfile if enclosed in single quotes. Thereby the quotes in the SQL variant has to be escaped by a backslash.
- As a general rule, in the native Transbase notation, the range specification [upb:lwb] can be omitted if upb is YY.

Table 9.3. Variants of Date Literals

Variant	Notation
Native TB	DATETIME[YY:DD](2002-12-24)
Native TB	DATETIME(2002-12-24)
SQL	DATE '2002-12-24'
ODBC	{ d '2002-12-24' }
Spoolfile only	'2002-12-24'

Table 9.4. Variants of Time Literals

Variant	Notation
Native TB	DATETIME[HH:SS](17:35:10)
SQL	TIME '17:35:10'

Variant	Notation
ODBC	{ t '17:35:10' }
Spoolfile only	'17:35:10'

There are literals which are only representable in native Transbase notation, because SQL (as well as ODBC) only supports subtypes of the most flexible DATETIME type. See the following examples:

The 24-th of December:

```
DATETIME[MO:DD](12-24)
```

Twelve o'clock:

```
DATETIME[HH](12)
```

Note the following rules illustrated by the examples above:

- The range spec can be omitted if and only if the upperbound range is YY.
- If upperbound and lowerbound range are identical, the range spec can be abbreviated to the form [YX].

9.1.4. Valid Datetime Values

Independent of the notation syntax, some datetime values are not accepted as legal values:

Transbase assumes that the Julian calendar is to be applied from Saturday `datetime(1-1-1)` to Thursday `datetime(1582-10-04)`. The Gregorian calendar is applied from the following day Friday `datetime(1582-10-15)` until Sunday `datetime(32767-12-31)`. Each year starts at `datetime[MO:DD](1-1)` and ends at `datetime[MO:DD](12-31)`.

Only datetimes are accepted that might be legal values according to this premise.

- If MO and DD are inside the range, then the DD value must not exceed the highest existing day of the MO value.
- If YY and DD are inside the range then leap year rules of the Julian/Gregorian time periods specified above apply.

```
DATETIME[MO:DD](4-31)     -- Illegal: no date with such components exists
DATETIME[MO:DD](2-29)     -- Legal:   there are dates with such components
DATETIME(1988-2-29)       -- Legal:   leap year
DATETIME(1900-2-29)       -- Illegal: no leap year
DATETIME(1582-10-14)      -- Illegal: 1582 was a rather short year anyway
```

9.1.5. Creating a Table with Datetimes

The type specifier for a datetime field of a table in a CreateTableStatement consists of the keyword DATETIME followed by a RangeSpec.

```
CREATE TABLE myfriends
(   name        CHAR(*),
    birthday    DATETIME [YY:DD],  -- alternative syntax: DATE
    firstmet    DATETIME [YY:HH]   -- no alternative
```

)

This table would be appropriate to describe persons with their name and birthday and the time when met first or talked to first.

Note that although all datetime values in the table are exactly of the specified format, it is possible to insert records with datetime fields of a different precision. Implicit conversions (CAST operators) then apply as described in the following chapters.

9.1.6. The CURRENTDATE/SYSDATE Operator

An operator CURRENTDATE is provided which delivers the actual date and time. SYSDATE is a synonym for CURRENTDATE. The result type is DATETIME[YY:MS] or TIMESTAMP but note that the effective precision may be less (e.g. *[YY:SS]*) depending on the system clock of the underlying machine. In the latter case, it is tried, however, to set the MS field in such a way that even several successive calls of CURRENTDATE *do not* deliver the same date (see also below).

The operator CURRENTDATE may be used wherever a datetime literal can be used.

CURRENTDATE may also appear as value in a spool file.

When used in a statement the CURRENTDATE operator is evaluated only once and its resulting value remains the same during the whole execution of the statement.

As long as a cursor is open, any interleaving cursor sees the same CURRENTDATE result as the already open cursor. In other words, a consistent view of the CURRENTDATE is provided for interleaving queries inside an application. In all other situations (non interleaving queries), it is tried to evaluate successive calls of CURRENT-DATE such that different results are delivered (see above).

9.1.7. Casting Datetimes

A datetime value can be cast to a different range. A cast operation can be performed explicitly by the CAST operator or implicitly occurs before an operation with the datetime value is performed (see *Type Exceptions and Overflow*).

In an explicit CAST operation, the syntax of the type specifier is the same as the one used in a *CreateTableStatement* .

```
CURRENTDATE CAST DATETIME[YY:DD] -- current year/month/day
CURRENTDATE CAST DATE            -- equivalent to above
CURRENTDATE CAST DATETIME[YY:MO] -- current year and month
CURRENTDATE CAST DATETIME[HH:SS] -- current hour/minute/second
CURRENTDATE CAST TIME            -- equivalent to above
```

In the sequel, the range boundaries of the value to be cast are called the upperbound source range and the lowerbound source range. The range boundaries of the target range are called the upperbound target range and the lowerbound target range.

The rules to construct the result datetime value from the given one are as follows:

- *DTC1:* All components having a range index smaller than the source lowerbound range are set to the smallest possible value (0 for MS, SS, MI, HH and 1 for DD, MO, YY).
- *DTC2:* All components having a range index higher than the source upperbound range are set to the corresponding components of CURRENTDATE.

- *DTC3:* The other components (i.e. having range index between source lowerbound index and source upperbound index) are set as specified by the source datetime fields.

Examples:

```
-- Assuming that the CURRENTDATE is                    DATETIME [YY:MS] (1989-6-8...):
DATETIME [HH] (12) CAST DATETIME [YY:MS]          -->  DATETIME [YY:MS] (1989-6-8 12:0:0.0)
DATETIME [YY:MO] (1989-6) CAST DATETIME [MO:DD]   -->  DATETIME [MO:DD] (6-1)
```

DD is set to the smallest possible value, namely 1.

9.1.8. TRUNC Function

The TRUNC function is a shortcut to cast a datetime value to the type DATE, i.e. DATETIME[YY:DD]

```
TRUNC ( DATETIME [YY:MS] (1989-6-8  12:0:0.0) )
```

yields

```
DATETIME [YY:DD] (1989-6-8)
```

9.1.9. Comparison and Ordering of Datetimes

Datetime values are totally ordered. A datetime d1 is greater than d2 if d1 is later in time than d2.

The SQL operators <=, <, = etc. as used for arithmetic types are also used for comparison of datetimes.

If datetimes values with different ranges are compared, they are implicitly cast to their common range before the comparison is performed. The common range is defined by the maximum of both upperbound and the minimum of both lowerbound ranges. Note, however, special rules for CURRENTDATE as described below!

Thus, it is possible that one or both datetime values are implicitly cast according to the casting rules described in the preceding chapter.

The DATETIME comparison

```
DATETIME[YY:MI](1989-6-8 12:30] = DATETIME[YY:DD](1989-6-8)
```

yields FALSE, because the second operand is implicitly cast to the value *DATETIME[YY:MI] (1989-6-8 00:00)*

The comparison

```
DATETIME[MO:DD](6-8) = DATETIME[YY:DD](2000-6-8)
```

will yield TRUE in the year 2000 and FALSE in other years.

To retrieve all persons who have been met since February this year:

```
SELECT * FROM Persons
WHERE talked >= DATETIME [MO](2)
```

To retrieve all persons whose sign of the zodiac is Gemini:

```
SELECT * FROM Persons
WHERE birthday CAST DATETIME [MO:DD]
BETWEEN DATETIME [MO:DD] (5-21)
AND DATETIME [MO:DD] (6-20)
```

Note that the CAST operator applied to birthday is necessary to restrict the common range to [MO:DD]. If the explicit CAST were omitted, the common range would be [YY:DD] and the constant comparison operators would be extended by the current year so that the query would not hit any person.

To retrieve all persons ordered by their age (i.e. ordered by their birthday descendingly):

```
SELECT * FROM Persons
ORDER BY birthday DESC
```

An exception of the type adaption rule is made for the CURRENTDATE operator. In comparisons (=, <>, <, <=, >, >=) the CURRENTDATE value is automatically adapted to the range of the comparison operand. In most situations this is useful and avoids explicit CAST operations.

```
SELECT * FROM Persons
WHERE talked CAST DATETIME[YY:DD] = CURRENTDATE
```

This example retrieves all records whose field value "talked" matches the current day (year, month, day). Without the type adaption rule for CURRENTDATE, one would also have to cast CURRENTDATE on the range [YY:DD].

9.2. Principles of Timespan and Interval

The data type TIMESPAN (and INTERVAL, as a SQL conformant variant) is used to describe distances between absolute or periodic points in time with a certain precision. Examples for TIMESPAN values are the result times of a sports event (measured in hour, minutes, seconds and/or milliseconds), the average life time of a material or product or the age of a person.

9.2.1. Transbase Notation for Type TIMESPAN

The concepts of components, range and range indices are similar to the type DATETIME. The following example shows the strong syntactic analogies between DATETIME and TIMESPAN. However, the semantics are clearly different:

- DATETIME[HH:MI] : suitable for points in time which are periodic on a daily basis (for example taking off time for a flight).
- TIMESPAN[HH:MI] : suitable for describing time intervals on a minute precision for example duration of a flight.

The following important rule applies:

- The range of a TIMESPAN must not span the MO-DD border.

This means that the ranges of all TIMESPAN types must either be inside [YY:MO] or inside [DD:MS]. For example, the following type definitions are illegal:

- TIMESPAN[YY:DD] -- illegal
- TIMESPAN[MO:HH] -- illegal

The reason is that the number of days is not the same for all months. So the arithmetic rules for timespan calculations would be compromised.

The set of allowed values on the components are also different from DATETIME. Obviously, the day component of a TIMESPAN value may have the value 0 whereas a DATETIME value containing a day component shows a value >= 1. The legal values in a TIMESPAN are shown in table *timespan* .

9.2.2. INTERVAL Notation for TIMESPAN

The SQL standard notation uses keyword INTERVAL (opposed to TIMESPAN) and different range identifiers with a different syntactic encapsulation. The following examples show the notation in contrast to the TIMESPAN notation.

SQL standard notation	Transbase notation
INTERVAL YEAR	TIMESPAN[YY]
INTERVAL YEAR(4) TO MONTH	TIMESPAN[YY:MO]
INTERVAL DAY	TIMESPAN[DD]
INTERVAL DAY(5)	TIMESPAN[DD]
INTERVAL HOUR TO SECOND	TIMESPAN[HH:SS]
INTERVAL HOUR TO SECOND(3)	TIMESPAN[HH:MS]
INTERVAL SECOND(3)	TIMESPAN[SS]
INTERVAL SECOND(5,3)	TIMESPAN[SS:MS]

If SQL notation is used, then the type is internally mapped to the corresponding TIMESPAN type. Thereby the optional precision on the start range is ignored.

If the end range is SECOND, then a precision indicates a fractional part so the end range effectiveley becomes milliseconds (MS).

If SECOND is start range (thereby automatically also end range) then a simple precision like (3) is ignored like in all start ranges - especially this precision does not specifiy a fractional part so the mapping is to SS.

If SECOND is start range (thereby automatically also end range) then a specification of a fractional part must be given as (m,n) as it is done in the last example.

9.2.3. Ranges of TIMESPAN Components

On the upperbound range of a TIMESPAN value, always values from 0 through MAXLONG are allowed.

On all components different from the upperbound components only those values are allowed which are below a unity of the next higher component. The allowed values are shown in the table.

By these rules, it is possible for example to express a time distance of 3 days, 1 hour and 5 minutes as 73 hours, 5 minutes i.e. as a TIMESPAN[HH:MI]. However, it is illegal to express it as 72 hours and 65 minutes.

Table 9.5. Ranges of Timespan Components

Transbase Notation	SQL Standard Notation	Allowed values if not upperbound range
YY	YEAR	0 - MAXLONG
MO	MONTH	0 - 11
DD	DAY	0 - MAXLONG
HH	HOUR	0 - 23
MI	MINUTE	0 - 59
SS	SECOND	0 - 59
MS	--	0 - 999

9.2.4. TimespanLiteral

There are 2 variants for TIMESPAN literals which correspond to the 2 variants of TIMESPAN type definition (TIMESPAN and INTERVAL). The following table shows examples in both notations.

Table 9.6. Timespan Literals in Transbase and SQL Notation

SQL standard notation	Transbase notation
INTERVAL '2-6' YEAR TO MONTH	TIMESPAN[YY:MO](2-6)
INTERVAL '2:12:35' HOUR TO SECOND	TIMESPAN[HH:SS](2:12:35)
INTERVAL '2 12' DAY TO HOUR	TIMESPAN[DD:HH](2 12)
INTERVAL -'1' YEAR	- TIMESPAN[YY](1)
INTERVAL -'4.5' SECOND	- TIMESPAN[SS:MS](4.500)

Note that negative TIMESPANs are reasonable (e.g. as the result of a subtraction of a DATETIME value from a smaller DATETIME value). In SQL syntax, literals with a negative value incorporate the '-' sign within the literal syntax whereas in Transbase native notation the '-' sign is written (as a separate token) in front of the TIMESPAN token. See also chapter *Sign_of_timespans* Sign of Timespans.

9.2.5. Sign of Timespans

A timespan value is positive or zero or negative. It is zero if all components of its range are zero. A negative timespan may result from a computation (see the following chapters) or can also be explicitly represented as a timespan literal prefixed with an unary '-' (in terms of the TB/SQL grammar this is an Expression).

The following literal denotes a negative timespan of 3 hours and 29 minutes.

```
- TIMESPAN [HH:MI] (3:29)       -- Transbase notation
  INTERVAL -'3:29' HOUR TO MINUTE   -- SQL standard syntax
```

Note

It is illegal to attach a '-' sign to any component of a timespan literal.

9.2.6. Creating a Table containing Timespans

The type specifier for a timespan field of a table in a *CreateTableStatement* consists of a TIMESPAN type specifier either in Transbase native syntax or in SQL standard syntax.

```
CREATE TABLE Marathon
(     name    CHAR(*)
      time    TIMESPAN [HH:MS] -- or INTERVAL HOUR TO SECOND(3)
)
```

Note that although all timespan values in the table are exactly of the specified format, it is possible to insert records with timespan fields of a different precision. Implicit conversions (CAST operators) then apply as described in the following chapters.

9.2.7. Casting Timespans

Similarily to datetimes, a timespan value can be explicitly or implicitly cast to a target range. Timespan casting, however, has a complete different semantics than datetime casting (recall Chapter Datetime Casting). A timespan cast transfers a value into another unit by keeping the order of magnitude of its value unchanged - however a loss of precision or overflow may occur.

The following rules and restrictions apply:

- *TSC1:* The target range must be valid, i.e. it must not span the MO-DD border.
- *TSC2:* The target range must be compatible with the source range, i.e. both ranges must be on the same side of the MO-DD border.
- *TSC3:* If the lowerbound target range is greater than the lowerbound source range then a loss of precision occurs.
- *TSC4:* If the upperbound target range is smaller than the upperbound source range then the component on the upperbound target range is computed as the accumulation of all higher ranged components. This may lead to overflow.

Examples:

```
TIMESPAN[DD](90)         CAST TIMESPAN[MO:DD] --x violates TSC1
TIMESPAN[DD](90)         CAST TIMRSPAN[MO]    --x violates TSC2
TIMESPAN[MO](63)         CAST TIMESPAN[YY:MO] --> TIMESPAN[YY:MO](5-3) -- exact conversion
TIMESPAN[YY:MO](5-3)     CAST TIMESPAN[MO]    --> TIMESPAN[MO](63)     -- exact conversion
TIMESPAN[SS](3666)       CAST TIMESPAN[HH:MI] --> TIMESPAN[HH:MI](1:1) -- loss of precision
TIMESPAN[DD:MI](3 12:59) CAST TIMESPAN[HH]    --> TIMESPAN[HH](84)     -- loss of precision
TIMESPAN[DD](365)        CAST TIMESPAN[MS]    --x overflow on MS component
```

9.2.8. Comparison and Ordering of Timespans

TIMESPAN values are totally ordered and can be compared, sorted etc. like *DATETIME* values. If their ranges differ, they are cast implicitly to an appropriate range.

```
TIMESPAN [MI](69) = TIMESPAN [HH](1)        -- false
```

```
TIMESPAN [MI] (69) = TIMESPAN [HH:MI] (1:9) -- true
```

9.2.9. Scalar Operations on Timespan

A timespan value can be multiplied by a scalar and divided by a scalar. The result is again a timespan value with the same range as the input timespan value. The scalar can be any arithmetic value but it is cast to type INTEGER before the operation is performed.

Multiplication:

The semantics of multiplication is that all components of the timespan are multiplied and the resulting value is normed according to the rules of valid timespans. Overflow occurs if the upperbound range value exceeds MAXLONG.

```
TIMESPAN [MI:SS] (30:10) * 10         -->  TIMESPAN [MI:SS] (301:40)
TIMESPAN [DD:MS] (1 6:50:07.643) * 4  -->  TIMESPAN [DD:MS] (5 3:20:30.572)
TIMESPAN [MI:SS] (214748365:10) * 10  -->  Overflow!
TIMESPAN [MI:SS] (214748365:10)
          CAST TIMESPAN [HH:SS] * 10  -->  TIMESPAN [HH:SS] (35791394:11:40)
TIMESPAN [MI:SS] (214748365:10)
          CAST TIMESPAN [DD:SS] * 10  -->  TIMESPAN [HH:SS] (1491308 02:11:40)
```

Division:

The semantics of division is as follows: first the timespan value is cast to its lowerbound range (a virtual cast which never yields overflow!), then the division is performed as an INTEGER division and then the result is cast back to its original range.

```
TIMESPAN [YY] (1) / 2                 -->  TIMESPAN [YY] (0)
TIMESPAN [YY:MO] (1-5) / 2            -->  TIMESPAN [YY:MO] (0-8)
TIMESPAN [DD:MS] (5 3:20:30.572) / 4  -->  TIMESPAN [DD:MS] (1 6:50:7.643)
```

9.2.10. Addition and Substraction of Timespans

Two timespans with compatible ranges (see Rule TSC2 in *Casting Timespans*) can be added or substracted.

The result is a timespan value whose range is the common range of the input values. The common range is again defined by the maximum of both upperbounds and the minimum of both lowerbounds. The input values are cast to their common range before the operation is performed.

```
TIMESPAN [DD] (1000) + TIMESPAN [DD] (2000) -->  TIMESPAN [DD] (3000)
TIMESPAN [YY] (1)    + TIMESPAN [MO] (25)   -->  TIMESPAN [YY:MO] (3-1)
TIMESPAN [YY] (1)    - TIMESPAN [MO] (27)   --> -TIMESPAN [YY:MO] (1-3)
```

To retrieve the time difference between the winner and the looser of the Marathon as well as the average time:

```
SELECT MAX(time) - MIN(time), AVG(time)
FROM Marathon
```

9.3. Mixed Operations

9.3.1. Datetime + Timespan, Datetime - Timespan

If a timespan is added to or substracted from a datetime, the result is again a datetime. The range of the result is the common range of the two input operands as defined above.

```
DATETIME [YY:DD] (1989-6-26) + TIMESPAN [DD] (30)         --> DATETIME [YY:DD] (1989-7-26)
DATETIME [HH:MI] (12:28)     - TIMESPAN [SS] (600)        --> DATETIME [HH:SS] (12:18:00)
DATETIME [YY:MO] (1989-2)    + TIMESPAN [DD:MI] (3 20:10) --> DATETIME [YY:MI] (1989-2-4 20:10)
```

If the upperbound range of the input datetime value is less than YY, then the datetime is always cast to [YY:lb] before the operation is performed (lb is the lowerbound range of the datetime).

```
DATETIME [MO:DD] (2-28) + TIMESPAN [HH] (24)  --> DATETIME [MO:HH] (2-29 0)
```

if run in a leap year; in other years it yields

```
DATETIME [MO:DD] (2-28) + TIMESPAN [HH] (24)  --> DATETIME [MO:DD] (3-1 0)
```

When the range of the input timespan is on the left side of the MO-DD border Transbase requires the range of the input datetime to be completely on the left side of the MO-DD border, too. The reason is to avoid the semantic ambiguities that migth arise in some cases.

```
DATETIME [YY:DD] (1992-2-29) + TIMESPAN [YY] (1)  --> Invalid
DATETIME [YY:MO] (1992-2) + TIMESPAN [YY] (1)     -->  DATETIME [YY:MO] (1993-2)
```

9.3.2. Datetime - Datetime

When two datetimes are subtracted from each other the result is a timespan.

```
DATETIME [YY:MO] (1989-3) - DATETIME [YY:MO] (1980-4)  -->  TIMESPAN [YY:MO] (8-11)
```

The result of a datetime subtraction may, of course, also be negative.

Except to one case (see below) the range of the result is again the common range of the two input values. If the input ranges are different the two input values are cast to their common range before the operation is performed.

```
DATETIME [HH:MI] (12:35) - DATETIME [HH] (14)  -->  TIMESPAN [HH:MI] (1:25)
```

One slight complication arises when the range of the resulting timespan would span the MO-DD border and thus would be invalid. In this case, the upperbound of the result range is always DD.

```
DATETIME [YY:DD] (1989-6-26) - DATETIME [YY:DD] (1953-6-8)  -->  TIMESPAN [DD] (13167)
```

9.4. The DAY Operator

The DAY operator is applicable to Datetime and delivers the corresponding day of the year as an INTEGER value between 1 and 366.

```
DAY OF DATETIME (2000-12-31)
```

yields 366.

9.5. The WEEKDAY Operator

The WEEKDAY operator is applicable to Datetime and delivers the corresponding weekday as an INTEGER value between 0 and 6, where 0 means Sunday, 1 means Monday etc.

```
WEEKDAY OF DATETIME (2000-1-1)
```

yields 6 (meaning Saturday).

To retrieve all persons who have been met on a Sunday

```
SELECT * FROM Persons
WHERE WEEKDAY OF talked = 0
```

9.6. The WEEK Operator

The WEEK operator is applicable to Datetime and delivers the corresponding week as an INTEGER value between 1 to 54, according to the following rules:

- On every Sunday begins a new week.
- The first week of a year begins on January 1st - regardless of its weekday.

Consequently, the first and last week of a year may contain less than 7 days. A leap year ending on a Sunday has 54 weeks.

```
WEEK OF DATETIME (2000-12-31)
```

yields 54.

9.7. The ISOWEEK Operator

The ISOWEEK operator is applicable to Datetime and delivers the corresponding isoweek as an INTEGER value between 1 to 53, according to the following rules of ISO 8601 / DIN 1355:

- Each week begins on a Monday and ends on a Sunday.
- The first week of a year contains at least four days of the same year.

Consequently, there are no fragmented weeks, i.e. each week contains exactly 7 days. The 29th, 30th, and 31st of December may belong to the first week of the following year. The 1st, 2nd, and 3rd of January may belong to the last week for the previous year.

```
ISOWEEK OF DATETIME (2000-1-1)
```

yields 52, as that Saturday belongs to the 52th week of 1999.

9.8. The QUARTER Operator

The QUARTER operator is applicable to Datetime and delivers the corresponding quarter of the year as an INTEGER value between 1 and 4.

```
QUARTER OF DATETIME (2000-12-31)
```

yields 4.

9.9. Selector Operators on Datetimes and Timespans

A selector operator extracts a single component from a datetime or timespan value converted into the type INTEGER. A selector operation consists of one of the keywords YY, MO, DD, HH, MI, SS, MS, followed by an expression of type DATETIME or TIMESPAN.

Error occurs if the selector is not inside the range of the value.

Note that selecting a component semantically is not the same as casting to the range of the selector as shown by the examples:

```
MS OF TIMESPAN [SS:MS] (2.032)
```

yields 32.

```
TIMESPAN [SS:MS] (2.032) CAST TIMESPAN [MS]
```

yields

```
TIMESPAN [MS] (2032)
```

Note that the selector operator simply extracts one component without regarding the semantics of DATETIME or TIMESPAN. However, the selector operators (as well as the CONSTRUCT operator, see below) are useful because they provide for a bridge between DATETIME/TIMESPAN and the basic arithmetic types of SQL. For

example an application program can retrieve the components of datetime or timespan values into integer program variables for further arithmetic processing.

9.10. Constructor Operator for Datetimes and Timespans

The constructor operator (CONSTRUCT) is inverse to the selector operators. It constructs a datetime or timespan value from a list of arithmetic expressions. Syntactically it consists of the keyword CONSTRUCT followed by a syntax which is similar to that of a datetime or timespan literal. However, the components can be arithmetic expressions and are separated by commata. The arithmetic expressions are automatically cast to INTEGER before the CONSTRUCT operator is performed. Let e1, e2, e3 be arithmetic expressions in the following examples.

```
CONSTRUCT DATETIME [YY:DD] (1986,10,6)
```

is equivalent to

```
DATETIME [YY:DD] (1986-10-6)
```

```
CONSTRUCT DATETIME [MO:HH] (e1,e2,e3)
```

If omitted the range is assumed to start with YY. The following literal therefore denotes a range of [YY:MO].

```
CONSTRUCT TIMESPAN (e1,e2)
```

Note that if all values are constants, the CONSTRUCT operator is in no way superior to an equivalent datetime or literal representation which is also better readable.

CONSTRUCT is appropriate to build datetime and timespan values from components which are evaluated at runtime. For example, it is very useful for application programs which insert records with datetime or timespan values built up at runtime.

In an application program, the following SQL command would be suitable to insert a record into the Persons table:

```
INSERT INTO Persons(name,birthday)
       VALUES (:name, CONSTRUCT DATETIME(:year, :month, :day) );
```

The constructor and selector operators together allow to perform every manipulation on datetime and timespan values and also to override the built-in semantics. This may be necessary only occasionally as shown below.

Assume that in the table Persons several values for birthdays have been inserted (falsely) without the century of the year (e.g. 53-6-8 instead of 1953-6-8). The following statement would correct all such entries:

```
UPDATE Persons
SET birthday = CONSTRUCT DATETIME
    (YY OF birthday + 1900,
     MO OF birthday,
     DD OF birthday)
WHERE YY OF birthday < 100
```

In effect, the above statement does not express a semantically reasonable operation on datetimes but a correction of wrong datetime values. Note that this correction cannot be performed by adding a timespan value TIMESPAN [YY] (1900) because of the subtle semantics of the addition of timespans to datetimes.

10. The Datatypes BITS(p) and BITS(*)

The TB SQL datatype BITS(p) and BITS(*) represent bits vectors with fixed or variable size.

10.1. Purpose of Bits Vectors

Bits vectors are suited to represent certain 1-to-many relationships in a very compact manner.

Example 10.1. Construction of BITS Vectors

Assume a table TI with a field FK of arbitrary type and a key field FI of type INTEGER or SMALLINT or TINYINT.

FK	FI(INTEGER)
a	1
a	4
a	7
a	8
b	3
b	10
b	11

A representation using bitvectors yields the following table TB with fields FK and FB where FB is of type BITS(*):

FK	FB (BITS(*))
a	0b10010011
b	0b00100000011

The used notation here is that of bits literals (0-1 sequence starting with 0b).

10.2. Creation of Tables with type BITS

The notation in a DDL Statement is analogous to that of CHAR.

Creation of a table TI with a variable sized BITS field:

```
CREATE TABLE TI
(
    FK INTEGER,
    FB BITS(*),
    ...
)
```

Creation of a table with a fixed sized BITS field:

```
CREATE TABLE relb
```

```
(
    k INTEGER,
    b BITS(512),
    ...
)
```

The number p in BITS(p) is the number of bits that a value or a field of that type may hold. The maximum number of p is MAXSTRINGSIZE*8-4 , where MAXSTRINGSIZE depends on the pagesize. A value of type BITS(p) or BITS(*) semantically is a series of 0 or 1-bits. The bit positions are numbered and the leftmost position has the number 1.

10.3. Compatibility of BINCHAR and BITS

The types BINCHAR and BITS are compatible among each other. They form a hierarchy BINCHAR, BITS in increasing order (i.e. BITS is the higher of the 2 types).

Analogously to the arithmetic types, the value of the lower level type is automatically cast to the higher level type when an operation requires a higher level type input or when two values of different types are compared or combined.

10.4. BITS and BINCHAR Literals

A BITS literal is a sequence of the digits 0 and 1 prefixed by 0b.

```
0b0101       -- Type is BITS(4)
0b111100001  -- Type is BITS(9)
```

Inside the 0-1-sequence a positive repetition factor can be used as a shorthand notation for a series of equal bits:

```
0b0(4)1(5)0
```

is a shorthand notation for

```
0b0000111110
```

A repetition factor is a IntegerLiteral in round brackets.

No computed expression is allowed here. With a little bit care, also BINCHAR literals can be used for constants of type BITS, because BINCHAR is implicitly cast to BITS where needed. Note however that the values are not identical, e.g. the SIZE operator delivers different results.

```
0xaf08             -- is a BINCHAR literal
0b1010111100001000 -- is a BITS literal
```

They are not identical because

```
SIZE OF 0xaf08             -- delivers 2
SIZE OF 0b1010111100001000 -- delivers 16
```

The following expression, however, is identical to the above BITS literal.

```
0xaf08 CAST BITS(*)
```

A further shorthand notation is given by a dynamic bits constructor MAKEBIT (see below).

Note

When a BINCHAR value (e.g. a Literal) of type BINCHAR(p) is used as input for an operation which requires the type BITS, it is automatically cast to the type BITS(p*8).

10.5. Spool Format for BINCHAR and BITS

The spool format as produced by Transbase is the BinaryLiteral representation. The accepted format for spooling from file to tables is BITS Literal as well as BINCHAR Literal.

10.6. Operations for Type BITS

In the following paragraphs, the notations bexpr and iexpr are used. bexpr denotes a value of type BITS(p) or BITS(*). iexpr denotes a value of type TINYINT/SMALLINT/INTEGER. Both notations stand for constants as well as for computed expressions, e.g. subqueries.

10.6.1. Bitcomplement Operator BITNOT

Syntax:

```
BITNOT bexpr
```

Explanation: Computes the bitwise complement of its operand. The result type is the same as the input type.

```
BITNOT 0b001101           --> 0b110010
```

10.6.2. Binary Operators BITAND , BITOR

Syntax:

```
bexpr1 BITAND bexpr2
bexpr1 BITOR  bexpr2
```

Explanation: BITAND computes the bitwise AND, BITOR the bitwise OR of its operands. The shorter of the two input operands is implicitly filled with 0-bits up to the length of the longer input operands. If one of bexpri is type BITS(*) then the result type is also BITS(*) else the result type is BITS(p) where p is the maximum of the input type lengths.

```
0b1100 BITAND 0b0101       --> 0b0100
0b1100 BITOR  0b0101       --> 0b1101
```

10.6.3. Comparison Operators

All comparison operators (< , <= , = , <> , > , >=) as known for the other Transbase types are also defined for BITS. Length adaption is done as for BITAND and BITOR. A BITS value b1 is greater than a BITS value b2 if the first differring bit is 1 in b1 and 0 in b0.

10.6.4. Dynamic Construction of BITS with MAKEBIT

Syntax:

```
MAKEBIT ( iexpr1, [ , iexpr2 ] )
```

Explanation: If both iexpr1 and iexpr2 are specified: iexpr1 and iexpr2 describe a range of bit positions. Both expressions must deliver exactly one value which is a valid bit position (>= 1). MAKEBIT constructs a bits value which has 0-bits from position 1 to iexpr1-1 and has 1-bits from position iexpr1 to iexpr2.

If only iexpr1 is specified: iexpr1 describes one bit position or (in case of a subquery) a set of bit positions. MAKEBIT constructs a bits value which has 1-bits exactly on those positions described by iexpr1.

The result type is BITS(*).

```
MAKEBIT ( 3 , 7 )          --> 0b0011111
MAKEBIT ( SELECT ... )     --> 0b00101001
                           -- assuming the subquery delivers
                           -- values 3, 5, 8
```

10.6.5. Counting Bits with COUNTBIT

Syntax:

```
COUNTBIT ( bexpr )
```

Explanation: Returns number of 1-bits in bexpr, i.e. a non-negative INTEGER.

```
COUNTBIT ( 0b01011 )          --> 3
```

10.6.6. Searching Bits with FINDBIT

Syntax:

```
FINDBIT ( bexpr [ , iexpr ] )
```

Explanation: If iexpr is not specified it is equivalent to 1. If iexpr is greater or equal to 1, FINDBIT returns the position of the iexpr-th 1-bit in bexpr if it exists else 0. If iexpr is 0, FINDBIT returns the position of the last 1-bit in bexpr if there exists one else 0.

The result type is INTEGER.

```
FINDBIT ( 0b001011 , 1 )    --> 3
FINDBIT ( 0b001011 , 2 )    --> 5
FINDBIT ( 0b001011 , 4 )    --> 0
FINDBIT ( 0b001011 , 0 )    --> 6
```

10.6.7. Subranges and Single Bits with SUBRANGE

Syntax:

```
bexpr SUBRANGE ( iexpr1 [ , iexpr2 ] )
```

Explanation: If iexpr2 is specified then SUBRANGE constructs from bexpr a bits value which consists of the bits from position iexpr1 until position iexpr2 (inclusive). If iexpr2 exceeds the highest bit position of bexpr then 0-bits are implicitly taken.

If iexpr2 is not specified then SUBRANGE returns the iexpr1-th bit from bexpr as a INTEGER value (0 or 1).

In all cases iexpri must be valid bit positions > 0).

The result type is BITS(*) if iexpr2 is specified else INTEGER.

```
0b00111011 SUBRANGE (4, 6)    --> 0b110     (BITS(*))
0b00111011 SUBRANGE (6, 10)   --> 0b01100   (BITS(*))
0b00111011 SUBRANGE (2)       --> 0         (INTEGER)
0b00111011 SUBRANGE (3)       --> 1         (INTEGER)
```

10.7. Transformation between Bits and Integer Sets

Two operations are defined which serve to transform 1-n relationships into a compact bits representation and vice versa. Assume again the sample tables TI and TB given in *Purpose of Bits Vectors*. The following picture illustrates how the tables can be transformed into each other by an extension of the GROUP BY operator and a complementary UNGROUP BY operator. The operators are explained in detail in the following sections.

FK	FI(INTEGER)
a	1
a	4
a	7
a	8
b	3
b	10
b	11

FK	FB (BITS(*))
a	0b10010011
b	0b00100000011

10.7.1. Compression into Bits with the SUM function

The set function SUM, originally defined for arithmetic values, is extended for the type BITS(p) and BITS(*). For arithmetic values, SUM calculates the arithmetic sum over all input values. Applied to BITS values, SUM yields the BITOR value over all input values where a start value of 0b0 is assumed.

In combination with a GROUP BY operator and MAKEBIT operator, the table TI can be transformed to the table TB (see *Purpose of Bits Vectors*):

```
SELECT FK , SUM ( MAKEBIT ( FI ) )
FROM RI
GROUP BY FK
```

Also the notation OR instead of SUM is legal here.

10.7.2. Expanding BITS into Record Sets with UNGROUP

Given a table of the shape of TB (i.e. with at least one field of type BITS(p) or BITS(*), one can expand each record into a set of records where the BITS field is replaced by an INTEGER field.

An UNGROUP operator is defined which can be applied to a field of type BITS(p) or BITS(*).

The following statement constructs table TI from table TB (see *Purpose of Bits Vectors*):

```
SELECT * FROM RB
       UNGROUP BY FB
```

The UNGROUP BY operator can be applied to exactly one field and this field must be of type BITS.

For completeness, the full syntax of a SelectExpression (QueryBlock) is:

[291] SelectExpression ::= SelectClause /
[FromClause * 6 *//

```
                                   [ WhereClause ]                        * 1 *//
                                   [ UngroupClause ]                      * 2 *//
                                   [ GroupClause ]                        * 3 *//
                                   [ HavingClause ]                       * 4 *//
                                   [ FirstClause ] ]                      * 5 */
[295]          UngroupClause ::= UNGROUP BY FieldReference
```

The numbers at the left margin show the order in which the clauses are applied. It shows that the UngroupClause takes the result of the WhereClause as input: it constructs from each input record t a set of records where the BITS value of t at position *FieldName* is replaced by INTEGER values representing those bit positions of t which are set to 1.

11. LOB (Large Object) datatypes

This chapter gives an overview of LOBs (BLOBs and CLOBs) in Transbase. Described are the DDL statements and SQL language extensions.

11.1. The Data Type BLOB (Binary Large Object)

11.1.1. Inherent Properties of BLOBs

BLOB is a data type for fields of tables. Arbitrary many fields of a table can be declared with type BLOB. BLOBs are variable sized.

11.1.1.1. Overview of operations

Transbase does not interpret the contents of a BLOB. Each field of type BLOB either contains the NULL value or a BLOB object. The only operations on BLOBs are creation, insertion, update of a BLOB, testing a BLOB on being the NULL value, extracting a BLOB via the field name in the SELECT clause, extracting a subrange of a BLOB (i.e. an adjacent byte range of a BLOB), and extracting the size of a BLOB.

11.1.1.2. Size of BLOBs

BLOB fields are variable sized. The size of a BLOB object is restricted to the positive byte range of a 4-byte integer (2^{31} Bytes) minus some per-page-overhead of about 1%. The sum of sizes of all BLOBs of one table is restricted to 2^{42} Bytes (about 4 Terabytes) minus some overhead of about 1.5 %.

11.1.2. BLOBs and the Data Definition Language

The keyword BLOB describes the data type of a BLOB field in the CreateTableStatement.

```
CREATE TABLE GRAPHIK
(
    GRAPHIK_NAME  CHAR(20),
    GRAPHIK_TYP   INTEGER,
    IMAGE         BLOB
)
KEY IS GRAPHIK_NAME
```

A BLOB field can be declared NOT NULL. No secondary index can be built on a BLOB field.

11.1.3. BLOBs and the Data Manipulation Language

11.1.3.1. BLOBs in SELECT Queries

A SELECT Query that contains result fields of type BLOB prepares the database server to deliver the BLOB objects, however, it requires an extra call to fetch the BLOBs contents.

BLOB fields can appear in the ExprList of the SELECT clause of a QueryBlock, either explicitly or via the '*' notation.

No operators (except the subrange operator and the SIZE OF operator, see below) are allowed on BLOB fields.

```
SELECT GRAPHIK_NAME, IMAGE
FROM GRAPHIK
```

With the SUBRANGE operator (n,m) where n and m are positive integers, a part of a BLOB can be retrieved. The following example retrieves the first 100 bytes of all image fields:

```
SELECT GRAPHIK_NAME, IMAGE SUBRANGE (1,100)
FROM GRAPHIK
```

With the SIZE OF operator, one can retrieve the size in bytes of a BLOB field. SIZE OF delivers NULL if the field is NULL. The following example retrieves the sizes of all BLOB objects in the sample table.

```
SELECT GRAPHIK_NAME, SIZE OF IMAGE
FROM GRAPHIK
WHERE IMAGE IS NOT NULL
```

A BLOB field can appear in the SearchCondition of the WHERE clause only inside a NullPredicate. It is important to note that the DISTINCT clause in the ExprList of a SELECT clause does not eliminate 'identical' BLOB objects. This means that any two BLOB objects are considered different in the database even if their contents actually are identical. Analogously, the GROUP BY operator if applied BLOB objects forms one GROUP for every BLOB object.

BLOB objects have no meaningful order for the user. It is not an error to apply the ORDER BY clause to BLOB fields but the ordering refers to internal BLOB addresses and thus the result is of no use in the user's view.

11.1.3.2. BLOBs in INSERT Queries

BLOB values can be specified as binary literals in insert queries, but usually they are specified as parameters of prepared insert queries.

11.1.3.3. Spooling BLOBs

The SPOOLing of tables with BLOB objects is described in Chapter 'Spooling Lob Objects' within the Section 'The Data Spooler'.

11.2. The Data Type CLOB (Character Large Object)

CLOBs are used for storing large character data.

11.2.1. Inherent Properties of CLOBs

CLOB is like BLOB a variable sized data type for fields of tables. Arbitrary many fields of a table can be declared with type CLOB.

11.2.1.1. Overview of operations

Each field of type CLOB either contains the NULL value or a CLOB object. The only operations on CLOBs are creation, insertion, update of a CLOB, testing a CLOB on being the NULL value, extracting a CLOB via the field name in the SELECT clause, extracting a subrange of a CLOB (i.e. an adjacent byte range of a CLOB), extracting a substring of a CLOB and extracting the size (number of characters) of a CLOB.

11.2.1.2. Size of CLOBs

CLOB fields are variable sized. The size (in bytes) of a CLOB object is restricted to the positive byte range of a 4-byte integer (2^{31} Bytes) minus some per-page-overhead of about 1%. Thus the maximum number of characters of a CLOB dependes on the size of the UTF8-encoded text. The sum of sizes of all LOBs of one table is restricted to 2^{42} Bytes (about 4 Terabytes) minus some overhead of about 1.5 %.

11.2.2. CLOBs and the Data Definition Language

The keyword CLOB describes the data type of a CLOB field in the CreateTableStatement.

```
CREATE TABLE BOOK
(
    BOOK_TITLE  CHAR(100),
    BOOK_CONTENT  CLOB
)
```

```
KEY IS BOOK_TITLE
```

A CLOB field can be declared NOT NULL. No secondary indexes except a *fulltext index* can be built on a CLOB field.

11.2.3. CLOBs and the Data Manipulation Language

11.2.3.1. CLOBs in SELECT Queries

A SELECT Query that contains result fields of type CLOB causes the database server to deliver the CLOB objects, but an extra call per CLOB is needed to fetch the contents.

CLOB fields can appear in the ExprList of the SELECT clause of a QueryBlock, either explicitly or via the '*' notation.

The following operators are allowed on CLOBS: SUBRANGE, SUBSTRING, SIZE OF, CHARACTER_LENGTH and TO_CHAR (which is equivalent to a CAST VARCHAR(*)).

```
SELECT BOOK_TITLE, BOOK_CONTENT CAST VARCHAR(*)
FROM BOOK
```

Note that without the CAST VARCHAR(n) operation (or the TO_CHAR function) the result type would still be CLOB and not a representation as a printable string. This is necessary to enable the insertion of a (possibly modified) CLOB object into another CLOB field of a table.

With the SUBRANGE operator (n,m) where n and m are positive integers, a part of a CLOB can be retrieved. The SUBSTRING function is equivalent. The following examples retrieve the first 100 bytes of all book_content fields:

```
SELECT BOOK_TITLE, BOOK_CONTENT SUBRANGE (1,100) CAST VARCHAR(*)
FROM BOOK
```

```
SELECT BOOK_TITLE, TO_CHAR ( SUBSTRING(BOOK_CONTENT FROM 1 FOR 100) )
FROM BOOK
```

With the CHARACTER_LENGTH operator, one can retrieve the size in characters of a CLOB field. The two operators deliver NULL if the field is NULL.

11.2.3.2. CLOBs in INSERT Queries

CLOB values can be specified as string literals in insert queries, but usually they are specified as parameters of prepared insert queries.

```
INSERT INTO BOOK VALUES ('title of book','content of book')
```

11.2.3.3. Spooling CLOBs

The SPOOLing of tables with CLOB objects is described in Chapter 'Spooling Lob Objects' within the Section 'The Data Spooler'.

12. Fulltext Indexes

Transbase fulltext search is supported on fields of type CLOB, CHAR(p), CHAR(*), VARCHAR(p) and VARCHAR(*).

12.1. FulltextIndexStatement

A FulltextIndexStatement is provided which creates a fulltext index on one field.

Syntax:

[129]	FulltextIndexStatement ::=	CREATE [POSITIONAL] FULLTEXT INDEX IndexIdentifier [FulltextSpec] ON TableIdentifier (FieldIdentifier) [ScratchArea]
[130]	FulltextSpec ::=	[WITH SOUNDEX] [{ Wordlist) | Stopwords }] [Charmap] [Delimiters]
[131]	Wordlist ::=	WORDLIST FROM TableIdentifier
[132]	Stopwords ::=	STOPWORDS FROM TableIdentifier
[133]	Charmap ::=	CHARMAP FROM TableIdentifier
[134]	Delimiters ::=	DELIMITERS FROM TableIdentifier | DELIMITERS NONALPHANUM
[135]	ScratchArea ::=	SCRATCH IntegerLiteral MB

Explanation: A fulltext index is the prerequisite for fulltext search on the specified field (Fulltext-Predicate). Depending on whether POSITIONAL is specified or not, the fulltext index is called positional index or word index.

A word index allows so called word search whereas a positional index additionally offers so called phrase search. Word search and phrase search are explained below.

Beside the two variants called word index and positional index, fulltext indexes come in three further independent variants. The specifications WORDLIST, STOPWORDS, CHARMAP and DELIMITERS influence the contents of the fulltext index. They are explained below. All four specifications include a `TableName` which is a user supplied table. The contents of the table(s) supply information to the FulltextIndexStatement at the time it is performed.

After the statement's execution, the contents of the tables are integrated into the index and the tables themselves do not further influence the behaviour of the created index. They can be dropped by the user if they are not needed any more for other purposes.

The SCRATCH Clause is explained in Chapter 'Scratch Area for Index Creation'.

12.1.1. WORDLIST and STOPWORDS

By default, if neither WORDLIST nor STOPWORDS is specified, *all* words from the indexed field are indexed.

By WORDLIST, a positive list of words can be specified, i.e. specified words are indexed *only*.

By STOPWORDS, a negative list of words is specified, i.e. all words *except* those in the stopword list are indexed.

The tables supplied as Wordlist or Stopwords must have a single field of type `STRING` or any other string type.

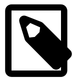
The WORDLIST and STOPWORDS variant mutually exclude each other.

If WORDLIST or STOPWORDS are specified, the fulltext index typically becomes much smaller because less words are indexed. On the other hand, if the fulltext predicate contains words which are not indexed, records which contain not-indexed words do not appear in the result set.

12.1.2. CHARMAP

By specifying CHARMAP, a character mapping algorithm can be supplied. It is specified by first inserting binary records into a binary table (let's say CTable) with fields VARCHAR(1) and VARCHAR(*) and by specifying CTable in the CHARMAP clause. For example, the table could contain a mapping from the German 'umlauts' ä, into ae, ö into oe, etc. such that the search need not rely on German keyboards.

'ä'	'ae'
'ö'	'oe'
'ü'	'ue'



Note

The character mapping is applied to the indexed words as well as to all search arguments in the FulltextPredicate. In the example above, the word 'lösen' would be stored as 'loesen' and a search pattern 'lö%' in a query would be transformed to 'loe%'.

It is also possible to specify the empty string as the target string for a certain character. Consequently, this causes all occurrences of that character to be ignored. For example, a record in CTable of the form

'.'	''

causes all occurrences of dot to be ignored. Thus, the word 'A.B.C.D' would be stored as 'ABCD' (and search for 'A.B.C.D' would hit as well as a search for 'ABCD'). Note, however, that in this example, a missing blank (delimiter, to be exact) after the concluding dot of a sentence would have the undesired effect to combine 2 words into one.

By default, a fulltext index works in case sensitive mode. Case insensitive search can be achieved by supplying a character mapping table which maps each upper case letter to its corresponding lower case letter.

12.1.3. DELIMITERS

The DELIMITERS clause specifies the word processing in the indexing process. If no DELIMITERS clause is specified, the indexing procedure handles each longest sequence of non-white-space characters as one word, i.e. by default, words are separated by white-space characters (blank, tabulator and newline). Also non-printable characters are treated as delimiters.

For example, the preceeding sentence would produce, among others, the words 'specified,' and 'non-white-space'. It is often convenient to supply additional word delimiters like '(', '.' or '-'.

Different delimiters can be specified by the DELIMITERS clause. If a Delimiters Table is specified, it must have 1 field of type VARCHAR(1) or VARCHAR(*) and must contain characters (strings of length 1). However, *non-printable* character are always treated as delimiters.

The NONALPHANUM specification provides a shorthand notation for the convenient case that all characters which are not alphanumeric are to be treated as delimiters.

Note that search patterns in Fulltext Predicates are not transformed with respect to delimiters (in contrast to CHARMAP!).

For example, if default delimiters have been used (white space) and a fulltext predicate contains a search component with a blank (e.g 'database systems'), then no record fulfills the predicate. In this case, one would have to formulate a fulltext phrase with two successive words ==- this is described later.

In all following examples for CreateIndexStatements, let f be a table which contains a CLOB field fb, and wl, sw, del be unary tables containing a wordlist, a stopword list, a delimiter character list, resp. Let cm be a binary table containing a character mapping.

```
CREATE FULLTEXT INDEX fbx
ON f(fb)

CREATE POSITIONAL FULLTEXT INDEX fbx
ON f(fb)

CREATE FULLTEXT INDEX fbx
WORDLIST FROM wl
ON f(fb)

CREATE FULLTEXT INDEX fbx
STOPWORDS FROM sw
CHARMAP FROM cm
DELIMITERS FROM del
ON f(fb)

CREATE FULLTEXT INDEX fbx
DELIMITERS NONALPHANUM
ON f(fb)
```

12.1.4. WITH SOUNDEX

The WITH SOUNDEX option adds the facility to search the fulltext index phonetically. This can be specified in the fulltext search predicate "CONTAINS" by also adding a SOUNDEX specification there: "CONTAINS SOUNDEX". See the example below and the section *FulltextPredicate* .

The WITH SOUNDEX option triggers a moderate space overhead which tends to become a negligible fraction as the base table grows.

```
... creating a fulltext index with phonetic search capability
CREATE FULLTEXT INDEX fbx WITH SOUNDEX ON f(fb)
... standard sarch is unaffected: searching for the name "Stuart"  ...
SELECT * FROM f WHERE fb CONTAINS ( 'Stuart' )
... searching for names which sound like "Stuart" (also hits for "STEWART" e.g.)  ...
SELECT * FROM f WHERE fb CONTAINS SOUNDEX ( 'Stuart' )
```

12.2. Implicit Tables of a Fulltext Index

Each fulltext index has a wordlist which contains the words that have been indexed so far (or, in the case of a Wordlist clause have been defined as the positive wordlist). The wordlist can be accessed by SQL statements as a pseudo table via a pseudo name described below.

For each of the STOPWORDS, CHARMAP and DELIMITERS clause, another pseudo table is created and is accessible like a normal table via a pseudo name. These tables should not be confused with the tables specified in the Stopwords, Charmap and Delimiters clause of a CreateIndexStatement. The latter are user defined tables used to define the contents of the pseudo tables at statement execution time. Any successive update to these user tables does not have any influence to the index and its pseudo tables.

If the WITH SOUNDEX option has been specified, an additional pseudo table SOUNDEX exists.

The names of the pseudo tables are derived from the name of the fulltext index. The table and field names as well as their types are given as follows (assume that the fulltext index has the name fbx):

```
FULLTEXT WORDLIST OF fbx ( word VARCHAR(*), wno INTEGER )

FULLTEXT STOPWORDS OF fbx ( word VARCHAR(*) )

FULLTEXT CHARMAP OF fbx ( source VARCHAR(1), target VARCHAR(*) )

FULLTEXT DELIMITERS OF fbx ( delimword VARCHAR(1) )

FULLTEXT SOUNDEX OF fbx (sound VARCHAR(*), word VARCHAR(*) )
```

For example, to see the words indexed up so far or to see the valid delimiters (if a DELIMITERS clause had been specified) one could say:

```
SELECT word FROM FULLTEXT WORDLIST OF fbx

SELECT * FROM FULLTEXT DELIMITERS OF fbx
```

The pseudo tables are not recorded in the catalog table *systable*.

It is also possible to update the internal WORDLIST OF table or STOPWORDS OF table in a restricted manner:

- The allowed update operating on a WORDLIST OF table is DELETE.
- The allowed update operating on a STOPWORDS OF table is INSERT.

By modifications of these internal tables one can influence the indexing behaviour of the fulltext index for future INSERTs into the base table. The current contents of the fulltext index are not changed.

12.3. FulltextPredicate

Search expressions on fulltext-indexed fields are expressed with a FulltextPredicate.

Syntax:

[279] FulltextPredicate : : = FieldIdentifier CONTAINS [SOUNDEX] (FulltextTerm)
[280] FulltextTerm : : = FulltextFactor [OR FulltextFactor]
[281] FulltextFactor : : = FulltextPhrase [Andnot FulltextPhrase]
[283] FulltextPhrase : : = (FulltextTerm) |
Atom [[DistSpec] Atom]...
[282] Andnot : : = AND | NOT
[284] Atom : : = SingleValueAtom | MultiValueAtom
[285] SingleValueAtom : : = StringLiteral | Parameter | FtExpression
[286] MultiValueAtom : : = ANY (TableExpression)
[287] DistSpec : : = [[MinBetween ,] MaxBetween]
[288] MinBetween : : = <Expression of type Integer>
[289] MaxBetween : : = <Expression of type Integer>
[158] Parameter : : = # IntegerLiteral (DataType) |

		Colon SimpleIdentifier | Questionmark
[290]	FtExpression : : =	<Expression without FieldReference to same block>

Explanation: The `FieldName` of a FulltextPredicate must refer to a field which has a fulltext index. The result type of SingleValueAtom must be VARCHAR(n) or VARCHAR(*).

A FulltextPredicate consists of a `FieldName`, the operator CONTAINS and a FulltextTerm in parentheses. The FulltextTerm is an expression consisting of FulltextPhrases and the operators AND, OR, NOT. The precedence is NOT before AND before OR. Parentheses may be used.

FulltextPhrases are of different complexities. The simplest form is a single Atom (e.g. a CHAR literal like 'database' or an application host variable). More complex forms have sequences of Atoms separated by DistSpecs.

A FulltextPredicate whose FulltextPhrases all consists of single Atoms only, is called a 'word search'.

A FulltextPredicate which contains a FulltextPhrase which is not a single Atom (i.e. contains at least 1 DistSpec) is called a 'phrase search'.

Note

If the FulltextPredicate is a phrase search then the fulltext index must be a POSITIONAL fulltext index.

A POSITIONAL fulltext index uses about three times the space of a non-positional fulltext index.

The following statements show word searches:

```
SELECT * FROM f WHERE fb CONTAINS ( 'database' )

SELECT * FROM f WHERE fb CONTAINS ( :hvar )   -- with named parameter

SELECT * FROM f WHERE fb CONTAINS ( 'data%' AND 'systems' )

SELECT * FROM f WHERE
fb CONTAINS ( 'database' NOT 'object' OR 'SQL' NOT '4GL')
```

The following statements show phrase searches:

```
SELECT * FROM f WHERE
fb CONTAINS ( 'database'   'systems' )

SELECT * FROM f WHERE
fb CONTAINS ( 'object%' [0,1] 'database' 'systems'
OR 'distributed' [1] 'systems' )
```

Wildcards:

Wildcard characters *'%'* and *'_'* have the same semantics as in the second operand of the LIKE predicate.

Escaping Wildcards:

The character '\' is reserved to escape a wildcard. If '\' is needed as a character of a word it must also be escaped. These rules are the same as in the LIKE predicate with a specified ESCAPE '\'.

Word Set of an Atom:

An Atom A in a FulltextPredicate specifies a word set WS(A) defined as follows.

- *If Atom A is a SingleValueAtom with result value SV:* If the result value of SV does not contain a wildcard then WS(A) consists of SV only, otherwise if SV contains wildcard(s), WS(A) consists of all words matching the pattern SV where matching is defined like in the explanation of the SQL LIKE predicate (with the '\' character as ESCAPE character).

- *If Atom A is a MultiValueAtom with result set MV:* WS(A) is the union of all WS(A') where A' are Atoms for the single elements of MV.

Semantics of Fulltext Predicates:

- *fb CONTAINS (Atom)* yields TRUE if and only if the field fb contains one of the words of WS(Atom), the word set specified by Atom.

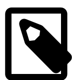
- *fb CONTAINS (Atom1 [n,m] Atom2)* where n and m are integer values, then the predicate yields TRUE if and only if the field fb contains a word w1 of the WS(Atom1) and a word w2 of WS(Atom2) and the number of words between w1 and w2 is at least n and at most m.

- *Atom [m] Atom* is equivalent to: Atom [0,m] Atom A missing distance specification is equivalent to [0]. Especially, a phrase for a series of adjacent words can be simply expressed as Atom Atom Atom ….

- *FulltextPhrase1 NOT FulltextPhrase2* delivers TRUE if and only if fb CONTAINS(FulltextPhrase1) delivers TRUE and fb CONTAINS(FulltextPhrase2) does not deliver TRUE.

- *FulltextPhrase1 AND FulltextPhrase2* is equivalent to: fb CONTAINS(FulltextPhrase1) AND fb CONTAINS(FulltextPhrase2)

- *FulltextFactor1 OR FulltextFactor2* is equivalent to: fb CONTAINS(FulltextFactor1) OR fb CONTAINS(FulltextFactor2)



Note

Do not omit the separating blank characters in the series of words of a phrase search! For example, consider the following specification:

```
fb CONTAINS( 'object''database''systems' )
```

effectively searches for a single word consisting of 23 characters including two single apostrophes. Note that the rules for SQL string literals apply.

12.4. Examples and Restrictions

In the following examples let F be a table with field fb of type CLOB where a fulltext index on fb has been created. Let WT be a table with a field word of type CHAR(*).

12.4.1. Examples for Fulltext Predicates

1.
```
SELECT * FROM F
WHERE fb CONTAINS ( 'database'  [ 0,1 ]  'systems' )
== delivers records where fb contains
== the series of the two specified words
== with at most one word in between.
```

2.
```
SELECT * FROM F
WHERE fb CONTAINS ( 'object'   'database'  'systems' )
== yields TRUE for records where "fb"
== contains the series of the three specified words.
```

3.
```
SELECT word FROM FULLTEXT WORDLIST WT
WHERE EXISTS
(SELECT * FROM F WHERE fb CONTAINS ( WT.word ) )
== delivers the values of "word"
== which occur as words in the field "fb" of any record of F.
```

4.
```
SELECT * FROM F
WHERE fb contains ( ANY ( SELECT  LOWER(word)  FROM WT) )
== delivers the records of "F"
== whose "fb" value contains at least one lowercase word
== of the word set described by field "word" of table "WT".
```

(*enum:3*) shows an application of a SingleValueAtom where the Atom is not a simple Literal or Primary.

(*enum:4*) shows an application of a MultiValueAtom.

12.4.2. Restrictions for Fulltext Predicates

Although the fulltext facility of Transbase is of considerable power, it also exhibits some syntactic restrictions which, however, can be circumvented.

Restriction 1:

A SingleValueAtom must not start with a '('.

For example, a SingleValueAtom of the form

```
( 'a' ||  :hvar) CAST CHAR(30)
```

is illegal because the '(' syntactically introduces a FulltextTerm of FulltextPhrase.

In these very rare cases replace the SingleValueAtom SVA by

```
'' || (SVA)
```

which is a string concatenation with the empty string.

Restriction 2:

An Atom must not contain a field reference to the same block where the fulltext table occurs.

Assume a table FT with fulltext field fb and fields fk and fc, where fc is of type CHAR(*) and fk is the key.

The following is illegal:

```
SELECT * FROM FT
WHERE fb CONTAINS (fc) -- ILLEGAL
```

This query must be formulated by using a subquery which combines FT with FT via the key fk:

```
SELECT * FROM FT ftout
WHERE EXISTS (
   SELECT  *
   FROM FT ftin
   WHERE ftout.fk = ftin.fk
   AND ftin.fb CONTAINS (ftout.fc)
   )
```

This query is legal because an outer reference in a fulltext atom is legal.

12.4.3. Phonetic Search in Fulltext Indexes

If a fulltext index has been specified with the `WITH SOUNDEX` option at creation time, the search via the CONTAINS operator may contain the phonetic option `SOUNDEX`.

Syntax:
[279] FulltextPredicate ::= FieldIdentifier CONTAINS [SOUNDEX] (FulltextTerm)

search a phonetic fulltext index for names sounding like "Stuart":

```
SELECT * FROM FT WHERE fb CONTAINS SOUNDEX ( 'Stuart' )
```

Note that the arguments of the "CONTAINS SOUNDEX" operator are automatically mapped onto their phonetic representation. It is thus unwise to apply the SOUNDEX operator onto the argument itself.

erroneous application of the "CONTAINS SOUNDEX" operator:

```
++++  SELECT * FROM FT WHERE fb CONTAINS SOUNDEX ( SOUNDEX('Stuart') )
++++  erroneous semantics
```

It is also unwise in most cases to apply the standard "CONTAINS" operator with an argument mapped to phonetic representation:

further erroneous try to do phonetic search:

```
++++  SELECT * FROM FT WHERE fb CONTAINS ( SOUNDEX('Stuart') )
++++  ...  not a phonetic search

...  correct phonetic search:
```

```
SELECT * FROM FT WHERE fb CONTAINS SOUNDEX ( 'Stuart' )
```

This concept stems from the fact that the arguments of "CONTAIN" might be delivered also by a subquery processed at runtime.

12.5. Performance Considerations

12.5.1. Search Performance

The fulltext index enables very good search times for fulltext searches. It, however, also causes some Performance limitations in database processing. This is described in the following chapters.

12.5.2. Scratch Area for Index Creation

Creation of a Fulltext Index is a time-consuming task if the base table and/or the field values (BLOBs) are large.

The processing time considerably depends on the amount of available temporary disk space. Transbase breaks all info to be fulltext-indexed into single portions to be processed at at time. The performance increases with the size of the portions.

It is therefore recommended to specify in a CREATE FULLTEXT INDEX statement the capacity of the available disk space in the scratch directory. For example if it is assured that 60 MB will be available, then the statement might look like:

```
CREATE FULLTEXT INDEX x ON f(fb) SCRATCH 60 MB
```

Note, however, that the scratch area is shared among all applications on the database.

12.5.3. Record Deletion

Deletion of records from a table is slow if the table has at least one fulltext index. The deletion takes time which is proportional (linear) to the size of the fulltext index.

Note additionally, that it is much faster to delete several records in one single DELETE statement rather than to delete the records one at a time with several DELETE statements.

13. Data Import/Export

Mass data import from various data sources is supported in Transbase®. Data can origin from external data sources such as

- *databases*,
- from *flat files storing delimiter separated values*
- and from *XML files*
- or *JSON files*.

There are two spool statements with the following functions:

- transfer of external data from a file into the database (SpoolTableStatement)
- transfer of a query results into a text file (SpoolFileStatement).

The first command is useful for building up a database from external data (residing on textfiles in a standard format, see below). The latter command is for extracting data from the database into textfiles. Also some Transbase® tools like tbarc (the Transbase® Archiver) use the facilities of the spooler.

Also LOBs (Large OBjects) can be handled by the spooler, although - of course - the corresponding files then do not contain text in general.

There is the possibility to choose between the DSV, the XML and the JSON mode of the data spooler. Both modes are explained next.

13.1. SpoolStatement

Syntax:

```
[436]        SpoolStatement ::= SpoolTableStatement | SpoolFileStatement
[437]   SpoolTableStatement ::= SPOOL LocalTableSpec
                                FROM [SORTED] FileLiteral [LOCAL]
                                { DSVSpec | XMLSpec | JSONSpec }
                                [ClobInlineSpec] [BlobInlineSpec]
[438]    SpoolFileStatement ::= SPOOL
                                INTO FileLiteral [LOCAL]
                                { DSVSpec | XMLSpec | JSONSpec }
                                [ClobInlineSpec] [BlobInlineSpec]
                                [LobFileSpec]
                                SelectStatement
[439]               DSVSpec ::= [ FORMAT DSV ]
                                [CodePageSpec]
                                [NullSpec]
                                [DelimSpec]
                                [QuoteSpec]
[440]               XMLSpec ::= FORMAT XML [NullSpec]
[441]              JSONSpec ::= FORMAT JSON
[442]              NullSpec ::= NULL [ IS ] StringLiteral
[443]             DelimSpec ::= DELIM [ IS ] { TAB | StringLiteral }
[444]             QuoteSpec ::= QUOTE [ IS ] StringLiteral
[445]        ClobInlineSpec ::= CLOBSINLINE
```

[446] BlobInlineSpec : : = BLOBSINLINE [HEX | BASE64]
[447] LobFileSpec : : = LOBFILESIZE = IntegerLiteral [KB|MB]
[85] CodePageSpec : : = CODEPAGE [IS] CodePage
[[WITH | WITHOUT] PROLOGUE]]
[86] CodePage : : = UTF8 | UCS | UCS2 | UCS4 |
UCS2LE | UCS2BE | UCS4LE | UCS4BE

13.1.1. The DSV Spooler

The DSV Spooler (Delimiter Separated Values) works with quite simple text documents as spool files. This means, that each record needs to have a value for each column of the destination table. Furthermore, the ordering of the record fields and the table columns have to be same. More details about the DSV spool files can be found in the section *External File Format*.

Syntax:
[437] SpoolTableStatement : : = SPOOL LocalTableSpec
FROM [SORTED] FileLiteral [LOCAL]
{ DSVSpec | XMLSpec | JSONSpec }
[ClobInlineSpec] [BlobInlineSpec]
[438] SpoolFileStatement : : = SPOOL
INTO FileLiteral [LOCAL]
{ DSVSpec | XMLSpec | JSONSpec }
[ClobInlineSpec] [BlobInlineSpec]
[LobFileSpec]
SelectStatement
[439] DSVSpec : : = [FORMAT DSV]
[CodePageSpec]
[NullSpec]
[DelimSpec]
[QuoteSpec]
[442] NullSpec : : = NULL [IS] StringLiteral
[443] DelimSpec : : = DELIM [IS] { TAB | StringLiteral }
[444] QuoteSpec : : = QUOTE [IS] StringLiteral
[85] CodePageSpec : : = CODEPAGE [IS] CodePage
[[WITH | WITHOUT] PROLOGUE]]
[86] CodePage : : = UTF8 | UCS | UCS2 | UCS4 |
UCS2LE | UCS2BE | UCS4LE | UCS4BE

Explanation:

The SpoolTableStatement inserts records from the specified file into the specified table (base table or view). The specified table must exist (but need not necessarily be empty). Thus, the SpoolTableStatement can be used as a fast means for insertion of bulk data.

The SpoolTableStatement has very good performance if the records in the table are ascendingly sorted by the table key(s) (best performance is achieved if the table additionally is empty). If the records are not sorted then Transbase inserts on-the-fly those records which fulfill the sortorder, the others are collected, then sorted, then inserted. For very large unsorted spoolfiles, it can be advantageous to split and spool them into pieces depending on the available disk space which is additionally needed temporarily for sorting.

Usage of the keyword SORTED allows to test if the input is actually sorted (if specified, an error is reported and the TA is aborted when a violation of the sort order is detected). This feature does not influence the spool algorithm but only checks if the input was suited to be spooled with maximal performance.

Without the LOCAL keyword, the specified file is read by the client application and transferred to the Transbase service process. If the file is accessible by the service process, the LOCAL clause can be used to speed up the spool process: in this case the service process directly accesses the file under the specified name which must be a complete path name.

For the checking of integrity constraints (keys, null values) the same rules as in *InsertStatement* apply.

The SpoolFileStatement stores the result records of the specified SelectStatement into the specified file (which is created if it does not yet exist, overwritten otherwise).

The spool files are searched or created in the current directory by default if they are not absolute pathnames.

For C programmers at the TCI interface a special property TCI_ATTR_DATA_DIRECTORY is available to change the default.

The StringLiteral in a DelimSpec must be of type CHAR(1) or BINCHAR(1), i.e. of byte length 1.

The StringLiteral in a NullSpec must be of type VARCHAR(1).

The StringLiteral in a QuoteSpec must be of type VARCHAR(1). Allowed values are '"', '"' or ''. The latter case implies *"null is ''"* and deactivates any quoting.

The optional ClobInlineSpec and BlobInlineSpec influence the representation of CLOB and BLOB fields.

See *External File Format* for the meaning of these specifications.

The codepage specification UTF8 means that the external file is UTF8-coded.

The codepage specification UCS2LE means that the external file is UCS2 (2 byte fixed length, little-endian). The codepage specification UCS2BE means that the external file is UCS2 (2 byte fixed length, big-endian). The codepage specification UCS2 means that the external file is UCS2 (2 byte fixed length, default format).

The codepage specification UCS4LE means that the external file is UCS4 (4 byte fixed length, little-endian). The codepage specification UCS4BE means that the external file is UCS4 (4 byte fixed length, big-endian). The codepage specification UCS4 means that the external file is UCS4 (4 byte fixed length, default format).

The codepage specification UCS means that the external file is the default UCS in default format, which is e.g. UCS2 on Windows platforms and UCS4 on UNIX platforms.

The optional PROLOGUE clause can be applied if the external file is prologued with the Unicode character 0uFEFF. If no PROLOGUE clause is given on input and no byte-order is specified, the byte order is determined automatically. If a byte-order is specified, and a differing PROLOGUE character is found, a runtime error is reported.

If no codepage is specified, the external file is assumed to be coded in UTF8.

```
SPOOL suppliers FROM suppliersfile

SPOOL suppliers FROM /usr/transb/data/suppliersfile LOCAL

SPOOL INTO suppliers_bak SELECT * FROM suppliers
```

13.1.1.1. FILE Tables

Syntax:

[84] FileTableStatement ::= CREATE
FILE (FileLiteral [CodePageSpec] [NullSpec] [DelimSpec])

TABLE [IF NOT EXISTS] TableIdentifier
(FieldDefinition [, FieldDefinition]...)

[91] FieldDefinition ::= FieldIdentifier DataTypeSpec
[DefaultClause | AUTO_INCREMENT]
[FieldConstraintDefinition]...

[92] DataTypeSpec ::= DataType | DomainIdentifier

Data stored in spool files or other compatible file formats may be integrated into the database schema as virtual tables. These *FILE* tables offer read-only access to those files via SQL commands. They can be used throughout SQL *SELECT* statements like any other base table.

The table definition supplies a mapping of columns in the external file to column names and Transbase datatypes.

Currently a File table can only be created *WITHOUT IKACCESS* and no key specifications are allowed. Therefore the creation of secondary indexes is currently not possible. These restrictions might be dropped in future Transbase versions.

For details on the optional parameters CodePageSpec, NullSpec and DelimSpec please consult the SpoolTableStatement.

FILE tables are primary designed as an advanced instrument for bulk loading data into Transbase and applying arbitrary SQL transformations at the same time.

```
CREATE FILE (/usr/temp/data_file)
  TABLE file_table WITHOUT IKACCESS
  (a INTEGER, b CHAR(*))

SELECT a+10, upper(b) from file_table
SELECT b FROM file_table, regular_table
  WHERE file_table.a=regular_table.a
```

13.1.1.2. External File Format

For building up a Transbase database from given text files, the DelimSpec and the NullSpec are of importance for scanning the text files. With the DelimSpec the separator between 2 field values in the text file can be specified (the default value is the tabulator). With the NullSpec the textual encoding of a SQL NULL Value is specified (the default value is a question mark `?`).

If not explicitly stated differently, the following description of records in text files both applies to the format generated by the spooler and to the format accepted by the spooler:

- Each line of text corresponds to one record.
- By default, fields are separated by one or more tabulators (TAB) unless differently specified by the DelimSpec. The DelimSpec always is exactly one character.
- By default, the character *?* represents a null value of any type unless differently specified by the NullSpec. The NullSpec always is exactly one character.
- The representation of INTEGER, REAL, NUMERIC, BOOL, DATETIME and TIMESPAN values corresponds to those of IntegerLiteral, RealLiteral, NumericLiteral, BoolLiteral, DatetimeLiteral and TimespanLiteral in the TB/SQL syntax. Integers, reals, numerics and timespans can be preceded by an - sign.

For text strings, the following rules apply:

- The empty string is represented as a sequence of two single quotes.
- A non-empty string x_1 ... x_n is spooled out with single quotes and as sequence of transformed characters 'T(x_1) ... T(x_n)'. In most cases T(x_i) = x_i holds. However, characters which have a special meaning must be escaped. Thus, for some characters x, T(x) is a two-character-sequence of a backslash ('\') and the character x. Special characters and their representation are shown in the table below.

As input for the spooler, the string can be represented as x_1 … x_n as well as 'x_1 … x_n' (i.e. the spooler eliminates surrounding quotes).

Special characters and their representation inside strings are shown in the following table.

Table 13.1. Special Characters in Spool Files

Special Character	Representation
'	\'
<tab>	\t
<newline>	\n
\	\\

Special Rule for Type `Binchar`

As stated above, when spooling tables from external files, the spooler accepts strings in the form xyz as well as 'xyz', although the form xyz is not a valid SQL literal for the type (VAR)CHAR(p) or CHAR(*). This is comfortable for data migration into Transbase but has the consequence that table spooling compromises *type compatibility* in the case of CHAR and BINCHAR. Inside a spool file, values for a BINCHAR field must be written as *BinaryLiteral*s, e.g. in the form `0xa0b1c2`. Whereas a value in the form xyz is accepted for a CHAR field, the same value is not accepted for a BINCHAR field because special values in that form would not be parsable in a unique manner, e.g. 0xa0b1c2 could be interpreted as a 8 byte CHAR value or a 3 byte BINCHAR value.

13.1.1.3. Key Collisions

When a table is spooled then Transbase rejects the data if there are 2 different records with the same key. In this situation the data to be spooled is inconsistent with the table creation specification. It may be advantageous to use Transbase to find out all records which produce a key collision. For this, recreate the table with the desired key but extended to all fields. For example, if the table T has the key on k1 and other fields f2,f3,f4, then create a table TFK with the clause: KEY IS k1,f2,f3,f4.

Then spool the table which in any case now works (except there are syntactical errors in the spoolfile). To find out all records with the same key, issue the query:

```
SELECT *
FROM TFK
WHERE k1 IN
(   SELECT k1
    FROM TFK
    GROUP BY k1
    HAVING COUNT(*) > 1
)
ORDER BY k1
```

13.1.1.4. Spooling LOB objects

For a table which has one or several LOB fields, the corresponding data in a spool file does not contain the LOB objects themselves but contains file names instead. Each LOB object is represented by a file name and the LOB object itself is stored in that file.

This is the default behaviour. BLOBs and CLOBs may also be stored inline if BLOBSINLINE or CLOBSINLINE, resp., is specified in the SPOOL statement.

Example 13.1. Spooling a file from a table with LOBs

Assume a 4-ary table "graphik" with field types CHAR(20), INTEGER, BLOB, CLOB.

- `SPOOL INTO spfile SELECT * FROM graphik`

  This SPOOL command creates a file `spfile` and a subdirectory `b_splfile`. The BLOB and CLOB objects are stored in files (one file per object) in the subdirectory. The names of the files are `B0000001` etc. with increasing numbers.

  The created file spfile would look like:

  ```
  'image31' 123    b_spfile/B0000001.003 b_spfile/B0000001.004
  'image32' 321    b_spfile/B0000002.003 b_spfile/B0000002.004
  'image33' 987    b_spfile/B0000003.003 b_spfile/B0000003.004
  ```

- SPOOL Option LOBFILESIZE:

  To reduce the number of files in the subdirectory, the LOBFILESIZE specification can be used, for example: LOBFILESIZE=10 MB

  ```
  SPOOL INTO spfile LOBFILESIZE=10 MB SELECT * FROM graphik
  ```

  Then a set of BLOBs whose size is not bigger than 10 MB is stored in a single file instead of n different files. Without a KB or MB qualification, the specified number is interpreted as MB.

- BLOBs and CLOBs as Inline Values:

  BLOBs and CLOBs may also be spooled as inline values inside the main target spool file.

  The option CLOBSINLINE outputs all CLOB fields as inline values similar to CHAR and VARCHAR fields.

  The option BLOBSINLINE BASE64 writes BLOB fields in a BASE64 representation.

  The option BLOBSINLINE HEX writes BLOB fields in hexadecimal form. This representaion is more space consuming than BASE64.

  ```
  SPOOL INTO spfile CLOBSINLINE BLOBSINLINE BASE64 SELECT * FROM graphik
  ```

  Note that for spooling a table from such a file also the corresponding INLINE specifications are mandatory in the SPOOL command.

Example 13.2. Spooling a table with LOBs from a file

Assume again the table graphik described above.

- spool data from a file `spoolfile` into the table graphik:

  ```
  SPOOL graphik FROM spoolfile
  ```

  The file spoolfile may look like:

  ```
  'image31' 123    b_spfile/B0000001.003 b_spfile/B0000001.004
  'image32' 321    b_spfile/B0000002.003 b_spfile/B0000002.004
  'image33' 987    b_spfile/B0000003.003 b_spfile/B0000003.004
  ```

- BLOBs and CLOBs as Inline Values:

  If BLOBs and/or CLOBs are given as INLINE values in the source spool file, then the SPOOL command must look like:

  ```
  SPOOL graphik FROM spoolfile CLOBSINLINE
  ```

  or (for BLOBs in BASE64)

  ```
  SPOOL graphik FROM spoolfile BLOBSINLINE BASE64
  ```

  or (if both are INLINE)

  ```
  SPOOL graphik FROM spoolfile CLOBSINLINE BLOBSINLINE BASE64
  ```

  The file could also contain absolute path names.

13.1.1.5. Filename Adaption on non-UNIX Operating Systems

If the application and/or the server is running on a non-UNIX operating system, the filename syntax requires some consideration. In the following, the filename translation mechanisms that Transbase uses are described.

In Transbase SQL, filenames occur in 3 different places: in the SpoolTableStatement as specification of the source file, in the SpoolFileStatement as specification of the target file and inside spoolfiles as LOB placeholders.

On all three places mentioned above, Transbase SQL allows filenames in UNIX syntax as described in the preceeding chapters. This means that all examples about data spooling and LOB filenames in spoolfiles also would be legal when the application and/or the database server run on MS WINDOWS.

When communicationg with the operating system, Transbase translates the filenames into valid system syntax. The '/' character is thereby interpreted as the delimiter between different directory levels.

For example, on a Windows machine the UNIX filename */usr/tmp/BLOBFILE003* would be mapped onto *\usr\tmp\BLOBFILE003*.

It is also legal to use WINDOWS filename syntax if the application is running on WINDOWS. For example, the statement

```
SPOOL graphik FROM \usr\tmp\graphikspf
```

would be legal on a Windows client.

Also note that Transbase maps UNIX-like filenames to WINDOWS-like style but not vice versa. If portability is required for applications and/or spoolfiles with LOBs, filenames should be written in UNIX syntax.

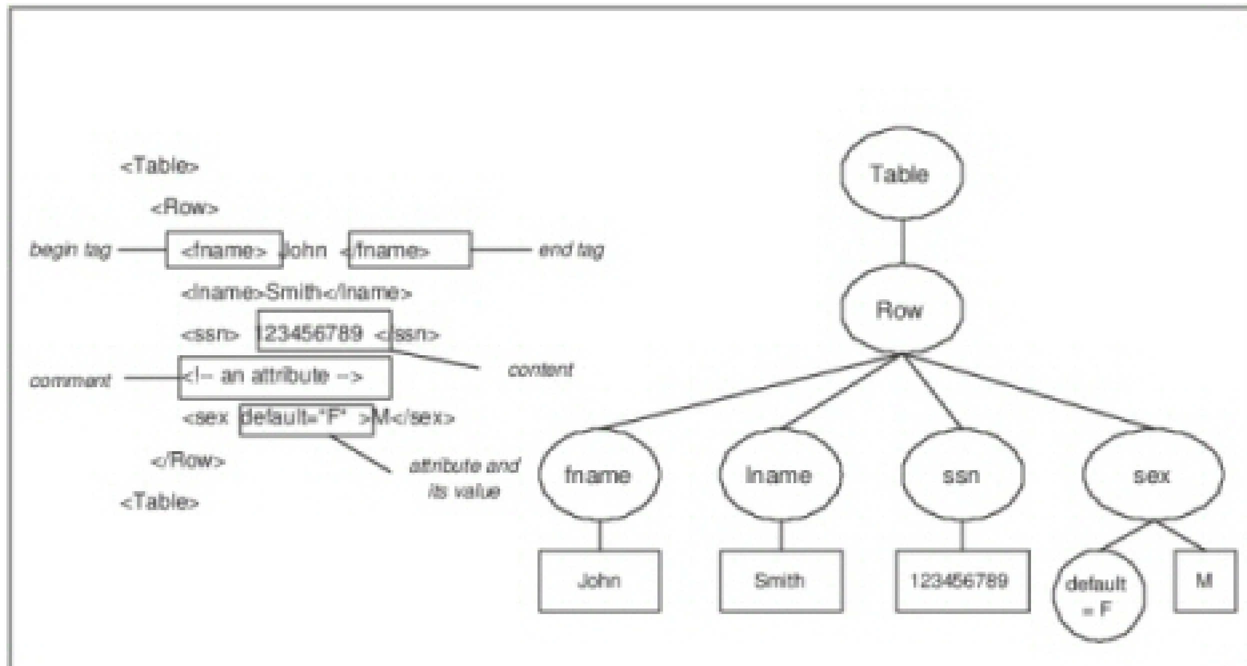
13.1.2. The XML Spooler

13.1.2.1. General Characteristics of XML

The eXtensible Markup Language (XML) is used to represent structural data within text documents. An XML document consists of nested elements describing the document structure. Nesting means, that each element may

have one or more child elements, each containing further elements. The application data is stored within fields or the content of elements. The usage of tags containing the element names makes XML data self-describing. Since an XML document may have only a single root element, the hierarchical structure of an XML document can be modeled as a tree, also known as document tree. In the following figure a small XML document and its document tree is shown.

Figure 13.1. Example of an XML Document and the Document Tree



The context of XML elements must not contain the signs >, <, &, ", and '. They have to be replaced with escape sequences. In the following table the special characters and their escape sequences are shown.

Table 13.2. Special Characters

Character	XML Representation
<	<
>	>
&	&
"	"
'	'

13.1.2.1.1. The Syntax of the XML Spool File

The XML spooler works only with XML documents that have a restricted structure. An XML spool file may only have four levels: the root level, whose tag is labeled with *Table*. The next level serves as delimiter between records and its tags are named *Row*. The third level consists of XML elements displaying the fields of the record. There are two possibilities to present elements of the third level:

1. The names of the elements have to be identical to the column labels of the destination table.

2. The elements are labeled with *Field* and have to carry an field *name* whose value displays the name of the table column.

```
<lname>Smith</lname>
<Field name="lname">Smith</Field>
```

These two line both have the same meaning.

Finally, values are presented as content of XML elements at the fourth level. The XML elements also may carry attributes. At the first level, the attributes *name* and *nullrep* are defined. The first defines the name of the destination table. The later one is used for the definition of the null representation. Its meaning is explained in section: *The Representation of Null Values* . For the second level (i.e. Row), only the attribute *name* is known by the spooler. The attributes defined for the third level and their meanings are declared in section: *XML Attributes Known by the XML Spooler* .

An example document with the document tree containing the four levels is shown in Figure: *Example of an XML Document and the Document Tree* .

According to these syntax rules, there are tag names with special meaning in the spool file called delimiter tags: *Table*, *Row*, *Field* and *Column* (see section: *The Usage of Format Information*). The XML spooler is case insensitive concerning these labels, i.e. one can write ROW or row and so on.

In contrast to this, in the DSV data spooler (Delimiter-Separated Values mode) the values of each record are presented in one line in the spool file. Usually, the values are separated by a tabulator ('*\t*') sign. Each record must have the same number of elements as there are columns in the destination table. Furthermore, these elements need to have the same ordering as the table columns. NULL values are usually presented as '?' per default.

In Figure: *DSV spool File* , a spool file suited for the spooler in the DSV mode is shown. It is used to transfer data into the table *SUPPLIERS*. The CREATE statement of that table is defined as follows:

```
CREATE TABLE supplier(supno INTEGER NOT NULL,
                      name VARCHAR(*),
                      address VARCHAR(*),
                      PRIMARY KEY(supno))
```

Figure 13.2. DSV spool File

```
5   DEFECTO PARTS       16 BUM ST., BROKEN HAND WY
52  VESUVIUS, INC.      512 ANCIENT BLVD., POMPEII NY
53  ATLANTIS CO.        8 OCEAN AVE., WASHINGTON DC
```

The spool file shown in Figure: *DSV spool File* has three records. In the XML spool file, additionally the structural information, as described above, is required.

Figure 13.3. XML Spool File

```
<Table>
   <Row>
      <Field name="supno">5</Field>
      <Field name="name">DEFECTO PARTS</Field>
      <Field name="address">16 BUM ST., BROKEN HAND WY</Field>
   </Row>
   <Row>
      <Field name="supno">52</Field>
      <Field name="name">VESUVIUS, INC.</Field>
      <Field name="address">512 ANCIENT BLVD., POMPEII NY</Field>
   </Row>
   <Row>
      <Field name="supno">53</Field>
      <Field name="name">ATLANTIS CO.</Field>
      <Field name="address">8 OCEAN AVE., WASHINGTON DC</Field>
   </Row>
</Table>
```

Figure: *XML Spool File* shows an XML spool file containing the same data as shown in the DSV spool file from Fig. *dsv_spf* .

In contrast to the DSV spool file, within an XML spool file the order of the record fields does not matter. Furthermore, additional elements may be present or elements can be missing (see also section: *Transfering XML Data*

Into the Database). This provides more flexibility in order to transfer query results into a database table whose scheme does not exactly match the output of the query.

Note

The Transbase XML spooler is not able to read XML documents containing a Document Type Description nor is it able to manage documents with namespace declarations.

13.1.2.2. Principal Functionality of the XML Spooler

13.1.2.2.1. Transfering XML Data Into the Database

Syntax:

[437]	SpoolTableStatement ::=	SPOOL LocalTableSpec FROM [SORTED] FileLiteral [LOCAL] { DSVSpec | XMLSpec | JSONSpec } [ClobInlineSpec] [BlobInlineSpec]
[440]	XMLSpec ::=	FORMAT XML [NullSpec]
[442]	NullSpec ::=	NULL [IS] StringLiteral

Explanation:

<tablename> is the name of the destination table where the records of the spool file are inserted.

<file> presents the file name of the spool file.

DSV stands for delimiter separated values. If the statement contains no format option the DSV mode is used per default. XML signals the XML spooling mode.

With the 'NULL IS' option, the null representation can be defined (see also section: *The Representation of Null Values*).

In case of the XML mode, the spooler scans the provided spool file and reads until the end of a record is reached (signaled by the end tag</Row>). If a field is missing, the default value is inserted in the database. If no default value is available, the NULL value is used for that field. If there are additional fields for which no column in the destination table can be found, these fields are ignored.

Figure 13.4. Complex XML spool File

```
<Table>
   <Row>
      <address>64 TRANQUILITY PLACE, APOLLO MN</address>
      <anything>?</anything>
      <supno>57</supno>
   </Row>
</Table>
```

So for example, the spool file of Figure: *Complex XML spool File* contains one record for the table *SUPPLIERS* (see section: *The Syntax of the XML Spool File*). The ordering of the fields does not match with the ordering of the table columns. The field 'name' is missing and since no default value is present, this field gets the value NULL. Furthermore, the record of the spool file contains a field labeled 'anything' which is ignored because the table *SUPPLIERS* does not have any column of that name.

13.1.2.2.2. Extracting Query Results Into an XML Document

Syntax:

[438] SpoolFileStatement : : = SPOOL
INTO FileLiteral [LOCAL]
{ DSVSpec | XMLSpec | JSONSpec }
[ClobInlineSpec] [BlobInlineSpec]
[LobFileSpec]
SelectStatement
[440] XMLSpec : : = FORMAT XML [NullSpec]
[442] NullSpec : : = NULL [IS] StringLiteral

Explanation:

<file> presents the name of the output spool file.

If the format option XML is used the result of the entered statement is formatted to XML. The SelectStatement can be any valid select statement.

As explained in section: *The Syntax of the XML Spool File*, the root of an XML spool file is labeled *Table*. The information of each record is presented within the begining and ending *Row* tag. For each record field, the name of the associated column is presented as attribute of the beginning *Field* tag. Between the beginning and the ending *Field* tags, the query result for this field is printed (see Figure *XML Spool File*).

13.1.2.3. Extended Functionality of the XML Spooler

13.1.2.3.1. Reading the XML Declaration

XML documents optionally may have an XML declaration which always is located at the beginning of the document. Among the version number, this declaration may include information about the encoding. The latter one may be of interest for the XML spooler.

```
<?xml version="1.0" encoding="UTF-8" ?>
```

The XML Spooler notices only the value of the encoding attribute within the declaration. All other information is ignored. However, at the moment, the XML spooler supports only UTF-8 encoded XML documents.

13.1.2.3.2. The Usage of Format Information

The XML spooler provides the opportunity to add format information as a header in front of the records. Such information are declared for the datetime and timespan types, so far. They are used to specify the range of these types for the complete spool file. Nevertheless, another format description may be entered as attributes within a record field. Furthermore, within the format information, the null representation and the default value of a table column may be defined. The format information has to be placed after the root tag (i.e. before the first record). For each column of the destination table, which requires additional information, an XML element named *Column* carrying several attributes is defined. This kind of information is called format information header here after wards.

Figure 13.5. Format Information Header

```
<Table>
   <column name="bdate" type="datetime[yy:dd]" nullrep="x"/>
   <column name="age" type="timespan[dd:ss]" default="22592 12:2:4"/>
   <Row>
        ...
   </Row>
</Table>
```

Figure *Format Information Header* shows an example of the format information header at the beginning of the spool file. For the *Column* element the attributes *name*, *type*, *nullrep*, and *default* are defined. With the value of the *name* attribute, the column of the destination table is identified. Accordingly, the *type* attribute carries the type definition and the range specification of this column as value. If the *nullrep* and/or the *default* attributes are present they define the representation of the null and/or the default value for the according table column. Because of the format information shown in Figure: *Format Information Header*, the XML Spooler supposes that values of the column *bdate* for example are of type datetime and are formatted beginning with the year and ending with days. Accordingly, values of the column *age* are of type timespan, beginning with days and ending with seconds. If this field is missing in one of the record, the default value '22592 12:2:4' is inserted.

The meaning of the *nullrep* attribute is explained in section: *The Representation of Null Values* . Morover, the usage of the *default* attribute is explained in section: *The Default Values*.

The Usage of Format Information when Transferring Data Into the Database

Together with the option explained above, there are four possibilities how the XML spooler determines which format should be used for a record field:

1. Datatime or timespan values can be represented in the TB/SQL syntax, i.e. the format information is written in front of the data (see section *External File Format*).

   ```
   <Table>
      <Row>
         ...
         <age>timespan[dd:ms](19273 12:2:4:1)</age>
      </Row>
   </Table>
   ```

2. The type and range specification is declared by an XML attribute. According to this, the XML parser listens for the attribute named *type* within the beginning tag of a record field.

   ```
   <Table>
      <Row>
         ...
         <bdate type="datetime[yy:dd]">1945-12-8</bdate>
      </Row>
   </Table>
   ```

   If the parser determines this attribute, it remembers its value until it can be used for type checking before inserting the record in the database.

Note

There is also the possibility to enter format information as TB/SQL syntax and additionally provide the concerning XML attributes. In this case, the attributes are ignored.

```
<today type="datetime[yy:dd]">
   datetime[yy:hh](2007-12-11 15)
</today>
```

In this case, the spooler assumes that the range specification of [yy:hh] is correct.

3. A header containing the format information as described above may be present. This information is only used, if the parser did not find such information within the field declaration (either as XML attributes or as TB/SQL representation).

4. If there is neither any format information within the field declaration nor any format information header present, the XML spooler assumes that the appropriate value has the format as defined in the column of the destination table.

Note

If the format of the value does not match the format to be used according to the added format information or the database scheme, an error handling is started (see section: *Error Reports*).

Writing the Format Information for Query Results

If a query result contains fields of type timespan or datetime, a header containing the format information as described above is generated and written into the output spool file.

```
<Table>
   <column name="bdate" type="datetime[yy:dd]"/>
   <column name="age" type="timespan[dd:ss]"/>
   <Row>
      ...
  </Row>
</Table>
```

13.1.2.3.3. The Representation of Null Values

With the XML spooler, there are several opportunities, to declare the representation of the null value: the definition of a single character within the spool statement, to add a null representation string as attribute of the *Table* element, or to use the *nullrep* attribute within the *Column* element. If none of these three possibilities is used, the default ('?') is used for the representation of the null value.

Table Spool Statement with a Null Representation

The *spool table statement* provides the option to enter a single character representing the Null value.

```
spool employee from test.xml format xml null is 'x'
```

If an 'x' is scanned for a field value, the spooler generates a NULL to be inserted in the database. If the value 'x' should be inserted instead of NULL, in the spool file the attribute *null* has to be set to false.

```
<Table>
   <Row>
      <Field name="lname" null="false">x</Field>
```

```
      ...
   </Row>
   ...
</Table>
```

Note

The XML spooler also supports the representation of null values by setting the *null* attribute of the *Field* element to true. Hence, the following two lines have the same meaning, if the null representation is set to 'x':

```
<Field name="lname">x</Field>
<Field name="lname" null="true"/>
```

The Null Representation for the Complete Document

As mentioned in section: *The Syntax of the XML Spool File* , the *Table* element may have an attribute named *nullrep*. Its value displays the representation of the null value for the remaining document. In contrast to the representation of the table spool statement, this value may be a string, not only a single character. If the *nullrep* attribute is present within the *Table* tag, the null representation of the spool statement - if any - is ignored. Again, if for a record field the same value as for the null representation should be inserted in the database, the *null* attribute has to be set to false.

```
<Table nullrep="xyz">
   <Row>
      <Field name="lname">x</Field>
      <Field name="rname" null="false>xyz</Field>
      <Field name="address">xyz</Field>
      ...
   </Row>
</Table>
```

Since the *nullrep* attribute is present, the value 'x' for the field *lname* is not interpreted as null although it was defined as null representation by the spool table statement. Thus, the following record values are inserted in the database: x, xyz, NULL, ...

The Null Representation for a Single Column

Within the format information header described in section: *The Usage of Format Information* , it is possible, to declare a value for the null representation. This is done with the *nullrep* attribute within the *column* element. As for the null representation of the *Table* element, the value may be a string. If this attribute is entered there, the value defines the null representation only for the column of the specified name. Other null representations (that from the *Table* element or that of the spool statement) then are not applied to the specified column. Again, if a value to be inserted in the database is the same as the null representation value, the attribute *null* has to be set to false.

```
<Table nullrep="xyz">
   <Column name="lname" nullrep="abc"/>
   <Row>
      <Field name="lname">abc</Field>
      <Field name="rname">xyz</Field>
      ...
   </Row>
   <Row>
      <Field name="lname" null="false">abc</Field>
```

```
      <Field name="rname" null="false">xyz</Field>
      ...
   </Row>
    <Row>
      <Field name="lname">xyz</Field>
      <Field name="rname">x</Field>
      ...
   </Row>
 </Table>
```

Although, if in the spool statement the NULL IS 'x' option was used, the following record values are generated and inserted in the database:

```
NULL, NULL, ...
abc,  xyz,  ...
xyz,  x,    ...
```

The Default Value for the Null Representation

If no null representation is present (neither in the spool statement nor nor in the spool file), the default null symbol ('?') is used. This is also true for the DSV Spooler. Also in this case, it is necessary, to set the field attribute *null* to false if the value ? has to be inserted in the database.

Writing the Null Representation

When writing the query result in an XML document, the *Table* element gets the attribute *nullrep* in any case. At the moment, the value of this attribute can be only a single character. The value is either the default null symbol ('?') or was entered with the file spool statement. Furthermore, it is not possible to define a null representation for a single column.

```
spool into test.xml format xml null is 'x' select * from employee
```

In this case, the output document looks as follows:

```
<Table nullrep="x">
   <Row>
      <Field ....
   </Row>
   ...
</Table>
```

13.1.2.3.4. The Default Values

After a complete record was scanned by the XML spooler, for fields that are not present the default value if any available is inserted in the data base. Otherwise, for these fields a NULL is inserted. There are two possibilities to define the default values: first, default values can come from the table description. Second, within the format information header, an attribute declaration can be used to define the default value. These possibilities are explained next.

Default Values from the Table Description

Default values that come from the table description are declared within the CREATE TABLE statement or with an ALTER TABLE statement.

In the following example, a spool file is used to transfer data in a table called *employee*. The CREATE TABLE statement of this destination table looks as follows:

```
CREATE TABLE employee (..., fname VARCHAR(*) DEFAULT 'MARY', ...)
```

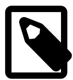
For each record, where the field *fname* is not present, the value "Mary" is inserted.

If the default value of the table description represents a sequence, this sequence has to be updated each time the default value is used.

In the following, parts of a spool file are shown that should be transfered into a data base table:

```
<Table>
  <Row>
     <Field name="ssn">20</Field>
     ...
  </Row>
  ...
</Table>
```

The field 'ssn' is of type integer and has as default a sequence. In the first record, this field is present. In all other records not shown, the field is missing and hence, the sequence is increased each time the default value is inserted.



Note

If there are more than one sequences per table, all sequences are increased at the same time. Hence, more sequences may result in confusing values.

Default Values within the Format Information Header

In order to declare a default value within the format information header, the attribute named *default* is used.

```
<column name="fname" default="Max"/>
```

In this case, for missing fields with the name *fname*, the default value "Max" is inserted.

If the attribute *default* is present within the format information header, the XML spooler checks if its value is valid according to the type declaration of the table description. If an uncorrect value was entered the definition of the default value is ignored.

Note

The definition of the default value within the format information header has a higher priority than that of the table description. I.e. if both, the table description and the format information header contain a default value for the same field, the default value of the table description is ignored.

13.1.2.3.5. XML Attributes Known by the XML Spooler

The following table shows the attributes known by the XML spooler and their possible values.

Table 13.3. Attributes and Their Values

Attribute Name	Possible Values
name	any string
nullrep	any string
type	datetime[cc:cc] | timespan[cc:cc]
null	true | false
default	any string
blobfile	any string
clobfile	any string
offset	any number
blobsize	any number
clobsize	any number
encoding	any string

If the spool file contains other attributes as declared within the above table, these attributes are ignored by the spooler. Similarly, if the parser encounters a not expected attribute value, depending on the location, an error is generated as explained in chapter: *Error Reports*.

Attributes Describing Format Information

As described in section; *The Usage of Format Information*, the parser of the XML spooler has to know the following attributes within a *Column* element: *name*, *type*, and *default*. The attributes *name* and *type* are also known within the beginning tag of a record field.

The Attributes for Null Values

As explained in section: *The Representation of Null Values* , for both - the DSV and the XML spooler, the default null symbol is presented by the '?' sign. Furthermore, a single character for the null representation may be entered with the spool statement. Within an XML spool file, the attribute labeled with *nullrep* may be used to overwrite this null representation for the complete document or only for a single column (see section: *The Representation of Null Values*). Additionally, the attribute 'null' can be used to signal the usage of a null value for a particular field. If this attribute carries the value 'true', a NULL is inserted for the appropriate record field. There are three possibilities, to declare a NULL field with this attribute:

1. The null attribute is set to true and no value is entered. In this case, usually no closing tag is required because the opening tag is closed after the attribute declaration.

   ```
   <today null="true"/>
   ```

2. Although no value is entered, it is valid to use the opening and closing tag within the XML document.

   ```
   <today null="true"></today>
   ```

3. The null field may carry a value.

```
<today null="true">2007-12-07</today>
```

Since the attribute value null is set to true, the data entered between opening and closing tag is ignored and a NULL is inserted for this record field.

Attributes Defining Lob Characteristics

The attributes *blobfile*, *clobfile*, *blobsize*, *clobsize* and *offset* are used when spooling lob. More details for these attributes can be found in chapter: *Spooling of Lobs with the XML Spooler* .

Attributes of the XML Declaration

As already explained in section: *Reading the XML Declaration*, an XML document optionally may have an XML declaration including attributes. The parser only remembers the value of the 'encoding' attribute, all other attributes within this declaration are ignored.

13.1.2.4. Error Reports

The transbase error handling differs between hard and soft errors. If a hard error occurs, the insertion process is stopped immediately and a roll back is performed. Hence, in case of an hard error, no record from the spool file is inserted. If a soft error is encountered, the appropriate record is ignored and skipped and all correct records are inserted.

13.1.2.4.1. Hard Errors

Concerning the XML spool mode, hard errors occur in connection with lobs or if an unexpected tag is encountered. If at least one column of the destination table is of type blob, each encountered error is handled as an hard error. The error of an unexpected tag occurs especially in connection with the delimiter tags defined in section: *The Syntax of the XML Spool File*. So for example, an XML spool file my begin only with an XML declaration or with the *Table* tag. After the beginning *Table* tag, the XML spooler accepts only a *Column* or a *Row* Tag. At the end of a *Column* tag, a further *Column* tag or a beginning *Row* tag is required. Finally, after a closing *Row* tag, only a beginning *Row* or an ending *Table* tag is allowed. If the spooler encounters another tag as expected, the spool process is aborted since no realistic recovery point is available.

13.1.2.4.2. Soft Errors

XML Syntax Errors

According to the error treating, there are three classes of XML syntax errors: hard errors as unexpected tags, syntax errors forcing the skipping of a record, and syntax errors leading in a scanning to the next recovery point. The first error class is already explained in section: *Hard Errors*, the other two classes will be explained next.

1. XML Syntax Errors Leading in a Skip Operation: If an XML syntax error occurs that still allows the correct interpretation of the following XML segments (i.e. tag, attribute, value, ...), the rest of the record is skipped. This means, the record is not inserted into the database but is written in the error report with an appropriate error message as XML comment. The end tag differs from the beginning tag as shown next.

   ```
   <Row>
      <fname>John</fname>
      <lname>Smith</lname>
      <ssn>123456789</ssn>
      <address>731 Fondren, Houston, TX</add>
      <sex>M</sex>
   </Row>
   ```

   In the example, the fourth record field starts with an *address* tag and ends with an *add* tag. In this case, the complete record is written in the error file that contains all incorrect records along with the proper error message. The spooling then starts at the beginning of the next record.

   Error Message:

   ```
   <Row>
      <fname>John</fname>
      <lname>Smith</lname>
      <ssn>123456789</ssn>
      <!== missmatch between open and closing tag ==>
      <address>731 Fondren, Houston, TX</add>
      <sex>M</sex>
   </Row>
   ```

2. XML Syntax Errors Leading in an Ignore Operation: If in the XML spool file an unexpected sign occurs, the spooler is not able to interpret the meaning of the following segments. Hence, nothing of the record is inserted in the database. The spooler ignores everything until to the next recovery point. A recovery point can be found at the beginning of each record, i.e. if a beginning *Row* tag is encountered. Such errors are for example missing angles, forgotten inverted commas, and so forth. Due to the restricted syntax of XML spool files, the transbase spooler also interprets mixed content as syntax error.

   ```
   <Row name="row1">
      <field name="fname">?</field>
      this text is mixed content
      <lname>?</lname>
      ...
   </Row>
   <Row name="row2"'>
      ...
   </Row>
   ```

   After a closing *Field* tag, only an opening *Field*, a closing *Row* tag, or a XML comment is expected. When scanning the text after the first record field, the spooler ignores the rest of the record and starts the spooling process at the begin of the next record <Row name="row2>). In the error report, the following error message is written.

   ```
   <Row name="row1">
      <field name="fname">?</field>
      <!== XML Syntax error found, scanning to begin of next record  ==>
   </Row>
   ```

Errors Occurring During the Spooling of a record

There are several errors that may occur during the parsing process of the record. If such an error is determined, the rest of the record is skipped. The incorrect record is written in an error report file where an error message is inserted before the faulty record field. Especially wrong types, invalid null definitions, or invalid field values may occur during the record spooling. These errors are explained next.

1. Invalid Null Definition: If in the table description a field is declared to be not null and in the spool file for this field the null attribute is set to true or the value for the null representation was entered, this record is skipped. In the following example, the field address must not be null according to the table description.

   ```
   <Row>
      <fname>Franklin</fname>
      <lname>Wong</lname>
      <ssn>333445555</ssn>
      <address null="true"/>
      <sex>M</sex>
   </Row>
   ```

   In the error file the following error message is entered:

   Error Message:

   ```
   <Row>
      <fname>Franklin</fname>
      <lname>Wong</lname>
      <ssn>333445555</ssn>
      <!== field must not be null ==>
      <address null="true"/>
      <sex>M</sex>
   </Row>
   ```

2. Wrong Types: Such errors occur for example, if a string is added where a numeric value is supposed.

   ```
   <Row>
      <fname>Joyce</fname>
      <lname>English</lname>
      <ssn>453453453</ssn>
      <address>563 Rice, Houston, TX</address>
      <sex>M</sex>
      <salary>error</salary>
   </Row>
   ```

   In the example above, the field salary is of type numeric. During the record spool, a string value is scanned and hence the error handling is started.

   Error Message:

   ```
   <Row>
      <fname>Joyce</fname>
      <lname>English</lname>
      <ssn>453453453</ssn>
      <address>563 Rice, Houston, TX</address>
      <sex>M</sex>
      <!== numeric error ==>
      <salary>error</salary>
   </Row>
   ```

   Errors Concerning XML Attributes: In table: *Attributes and Their Values*, the attributes and their possible values are listed. Errors concerning XML attributes are encountered if a not expected value was entered. The attributes can be classified in three categories: attributes describing format information, attributes describing characteristics of lobs and attributes belonging to the *Field* element. The error handling for attributes depends on this classification.

3. a. Errors Within Field Attributes: Usually, errors are encountered during the spooling stage which causes the skipping of the record and the generation of an appropriate error message. An example of an incorrect attribute value and the according error message is shown below.

```
<Row>
   <fname>James</fname>
   <lname>Borg</lname>
   <ssn>888665555</ssn>
   <bdate null="nonsense">1975-11-30<bdate>
   <sex>M</sex>
</Row>
```

In this case, the attribute 'null' of the XML element 'bdate' has the value 'nonsense'. Since for this attribute only the values 'true' or 'false' are defined, the following error message is generated.

```
 <Row>
   <fname>James</fname>
   <lname>Borg</lname>
   <ssn>888665555</ssn>
   <!== not expected attribute value (nonsense)  ==>
   <bdate null="nonsense">1975-11-30<bdate>
</Row>
```

b. Errors at Attributes Describing Lob Functionality: As mentioned above, if the destination table of the spool statement contains at least one lob column, each error is classified to be an hard error. Due to this, wrong values for the attributes *offset*, and *blobsize* result in an hard error. Hence, in such a case, no record of the spool file is inserted in the database.

c. Errors at Attributes Describing Format Information: As described in section: *The Usage of Format Information*, attributes that describe format information may occur in the format information header or within the begin tag of the record field. The correctness of the type and range specifications is verified at the insertion stage when the complete value is available. If there is an error encountered, the record is skipped and an error message is generated. By the reason of this, if there was entered an incorrect type (especially in the range part) within the format information header this results in the skipping of all records that use this information in the remaining spool file.

Errors Occurring at the Insertion Stage

After a record is scanned completely, problems may occur before or at the insertion step. So for example, which record fields are missing can be determined only after the complete record was parsed. Furthermore, integrity violations and key collisions can be recognized only when trying to insert the record. If such an error occurs, the record is not inserted in the database. It is written in the error file together with a proper error message. Since the error concerns the complete record and not only a single record field, the error message is placed in front of the record declaration. The spooling process goes ahead with spooling of the next record. An example of such an error is explained below.

As explained in section: *The Syntax of the XML Spool File*, it is not necessary to declare all record fields within an XML spool file. For a missing field the default value is inserted. If no default value is declared the NULL value is used. However, if such a field may not be null according to the table description the record must not be inserted in the database and hence, the error handling is started.

```
<Row>
   <fname>James</fname>
   <lname>Borg</lname>
   <ssn>888665555</ssn>
   <sex>M</sex>
</Row>
```

In this example, the field address which was declared to be not null is not present within the record. Hence, if no default value is available, the following error message is generated.

Error Message

```
<!== The field address must be declared - not null ==>
<Row>
```

```
    <fname>James</fname>
    <lname>Borg</lname>
    <ssn>888665555</ssn>
    <sex>M</sex>
</Row>
```

13.1.2.4.3. Attempt to Use an XML Document in the DSV Spooling Mode

The spool statement allows optional to choose the spooling mode (DSV or XML). If the format part is not entered, the DSV spooler is used per default. It may happen, that the format specification was forgotten and the user attempts to spool an XML document in the DSV spooling mode. In such a case, at the end of the spooling process, a proper error message is generated (error in DSV spooler - possibly forgot to enter 'format xml' option in statement).

For this error message, two conditions have to be fulfilled:

1. The first scanned symbol may be the start of an XML document.

2. There is at least one error for each line: If an XML document is used with the DSV spooling mode usually no correct record is encountered in the spool file, i.e. there are as many errors as spooled lines.

13.1.2.5. Spooling of Lobs with the XML Spooler

13.1.2.5.1. Transferring Lobs Into the Database

In the mode of delimiter separated values, the spool file usually contains file names for each lob field. The lob data is stored in the associated files (see section: *Spooling LOB objects*). The spooling process is performed by two runs: in the first scan, the file names are collected and requested from the client. The client than sends these files to the server. In the second scan, the remaining values of the spool file are read and the complete records are inserted in the data base. There is also the option to spool several lobs stored in one file by the usage of offset and size values.

If the XML mode is used, the file names of the blobs are entered as values of the attribute *blobfile* and the file names of the clobs are entered as values of the attribute *clobfile* at the according record fields. The spool statement is the same as explained in section: *Transfering XML Data Into the Database*. Figure *XML Spool File Containing Blobs* shows an example of an XML spool file containing blob file names. It is used to spool records in the table `blobex` which was created with the following statement:

```
CREATE TABLE blobex (nr INTEGER NOT NULL,
                     picture BLOB,
                     PRIMARY KEY (nr))
```

Figure 13.6. XML Spool File Containing Blobs

```
<Table>
   <Row>
      <nr>1</nr>
      <picture blobfile="B0000001.001"/>
   </Row>
   <Row>
      <nr>4</nr>
      <picture blobfile="maske.jpg"/>
   </Row>
   <Row>
      <nr>6</nr>
      <picture blobfile="photo.jpg"/>
   </Row>
</Table>
```

13.1.2.5.2. Writing Lobs from a Query Result

If a query result contains lob columns, the lobs usually are written in a separate file. The output spool file then contains the name of these files. In order to this, the spool file may look like that shown in Figure: *XML Spool File Containing Blobs* .

13.1.2.5.3. Inline Lobs

As in the DSV spooling mode, the lob data may also be entered as inline information. In an XML spool file, the inline lob data is presented as value between the proper opening and closing field tags. For inline blobs, the attributes *blobfile*, *blobsize* and *offset* are not present. Hence, if none of those attributes was entered, the spooler assumes that the value between open and closing tag belongs to an inline blob. Inline lobs are only useful if the lob is not too large. Inline lobs have to be encoded with hex representation or with the base64 (for pictures).

```
<picture>/9j/4AAQSkZ    ...            </picture>
```

In this example, the value of the blob represents parts of the code of a jpg-encoded picture.

While in the DSV spooling mode, mixing of inline lobs and lobs data located in a file is not possible, this mechanism is allowed in the XML spooling mode. The spooler decides because of the attributes that are available or not if the lob data is located in the spool file or if it has to be loaded from a file.

13.1.2.5.4. Storing Several Lobs in One File

Spooling Several Lobs into One File

As in the delimiter separated value mode, also in the XML mode it is possible to spool several lobs into one file.

In the following, an example statement is presented. It allows the spooling of the content from the table `blobex` containing one blob column in the file `blobexample`:

```
SPOOL INTO blobexample LOBFILESIZE=100 mb SELECT * FROM blobex
```

Figure: *Output DSV Spool File* shows the output document that is generated for the statement above when using the DSV mode. For each lob optionally the file name and a the byte offset is printed. The size of the lob always has to be present (see Figure: *Output DSV Spool File*).

Figure 13.7. Output DSV Spool File

```
1   'B0000001.001<0:11505>'    'M'
4   '<11505>'                  'M'
6   '<11505>'                  'M'
7   '<11505>'                  'M'
```

This output specifies that the first lob can be found in the file `B0000001.001` at byte offset 0 and has a size of 11,505 bytes. Since for the second and all further lobs no file name is add the same file is used. For those lobs only the size is specified. This means, the lob starts with the end of the lob before.

In the XML mode, the size and offset values are written as XML attributes (see Figure: *Output XML Spool File*).

Figure 13.8. Output XML Spool File

```
<Table>
   <Row>
      <nr>1</nr>
      <picture offset="0" blobsize="11505" blobfile="B0000001.001"/>
      <sex>M</sex>
   </Row>
   <Row>
      <nr>4</nr>
      <picture blobsize="11505"/>
      <sex>M</sex>
   </Row>
   <Row>
      <nr>6</nr>
      <picture blobsize="11505"/>
      <sex>M</sex>
   </Row>
   <Row>
      <nr>7</nr>
      <picture blobsize="11505"/>
      <sex>M</sex>
   </Row>
   <Row name="nr4">
      <nr>8</nr>
      <picture blobfile="photo.jpg"/>
   </Row>
</Table>
```

Spooling Several Lobs from One File

In the spool file, for each lob, optionally the filename and the byte offset has to be entered. The lob size is always required. In the XML mode, this information has to be presented as shown in the output document from figure: *Output XML Spool File*. Since for the second and all further records no file name is present in the spool file, the spooler uses the file (`B0000001.001`) from the record before. Furthermore, no byte offsets are available for these records. Hence, the spooling of the second lob starts and the end of the first lob and so on. If there is a blob field in a record where only the attribute *blobfile* is present but no size and no offset, then the spooler supposes that the complete file data belongs to one blob. So for example, for the last record of figure: *Output XML Spool File*, the complete content of the file 'photo.jpg' is loaded in the concerning lob container.

13.1.3. The JSON Spooler

In addition to formats DSV and XML Transbase® supports data import and export from/to files formatted as JSON.

13.1.3.1. Characteristics of JSON

JSON (JavaScript Object Notation) is an open-standard human-readable file format commonly used e.g. for data exchange and consists auf attribute:value pairs within object and array data types.

JSON syntax (general grammar)

[448] json : : = element
[449] value : : = object | array | string | number | boolean | nullvalue
[450] object : : = object_begin [whitespaces members] object_end
[451] members : : = member [, members]
[452] member : : = whitespaces string whitespaces : element
[453] array : : = array_begin [whitespaces elements] array_end
[454] elements : : = element [, elements]
[455] element : : = whitespaces value whitespaces
[456] string : : = " characters "
[457] characters : : = | character characters
[458] character : : = { 0u0020..0u10ffff ^ " ^ \ } | \ escape
[459] escape : : = " | \ | / | b | n | r | t | u hex hex hex hex
[460] hex : : = digit | { A..F | a..f }
[461] number : : = int frac exp
[462] int : : = digit | onenine digits | - digit | - onenine digits
[463] digits : : = digit [digits]
[464] digit : : = 0 | onenine
[465] onenine : : = { 1..9 }
[466] frac : : = | . digits
[467] exp : : = | { E | e } sign digits
[468] sign : : = | + | -
[469] boolean : : = true | false
[470] nullvalue : : = null
[471] object_begin : : = {
[472] object_end : : = }
[473] array_begin : : = [
[474] array_end : : =]
[475] whitespaces : : = | { 0009 | 000a | 000d | 0020 } whitespaces

13.1.3.2. Transbase® JSON spool format

The Transbase® JSON spooler only uses a flat JSON structure, i.e.

- a <table> is a JSON array: [<table>] containing zero or more records separated by comma:
- a <record> is a JSON object: { <record> } containing zero or more attribute-value pairs seperated by comma;

- a <field> is a JSON attribute-value pair: "<fieldname>": <fieldvalue> separated by colon.

Therefore the overall structure of a Transbase® spool file looks like (assuming the table has m records with n fields):

```
[
 {
  "<field_1>": <value_1-1>,
  "<field_2>": <value_1-2>,
  ...
  "<field_n>": <value_1-n>
 },
 {
  "<field_1>": <value_2-1>,
  "<field_2>": <value_2-2>,
  ...
  "<field_n>": <value_2-n>
 },
 ...
 {
  "<field_1>": <value_m-1>,
  "<field_2>": <value_m-2>,
  ...
  "<field_n>": <value_m-n>
 }
]
```

Example 13.3. JSON spool file: *Table SUPPLIERS*

```
[
  {
    "supno": 51,
    "name": "DEFECTO PARTS",
    "address": "16 BUM ST., BROKEN HAND WY"
  },
  {
    "supno": 52,
    "name": "VESUVIUS, INC.",
    "address": "512 ANCIENT BLVD., POMPEII NY"
  },
...
  {
    "supno": 64,
    "name": "KNIGHT LTD.",
    "address": "256 ARTHUR COURT, CAMELOT"
  }
]
```

13.1.3.3. Transbase / JSON data types

Transbase data types are mapped onto *JSON value types* as follows:

Transbase	JSON	example
null	*null*	null
bool	*bool*	true | false
tinyint	*number*	127
smallint	*number*	32767
integer	*number*	2.147.483.647
bigint	*number*	9.223.372.036.854.775.807
numeric[(p[,s])]	*number*	3.14159265

Transbase	JSON	example
float	*number*	2.71828182
double	*number*	2.99792458e8
char[({p|})]*	*string*	"x" | "Test"
varchar[({p|})]*	*string*	"?" | "Hello"
string	*string*	"Transbase"
binchar[({p|})]*	*string*	"0x48656c6c6f205472616e7362617365" (binchar literal)
bits[({p|})]*	*string*	"0b1010" (bits literal)
bits2[({p|})]*	*string*	"0b0101" (bits2 literal)
date	*string*	"date '2019-06-26'" (date literal)
time	*string*	"time '11:55:00'" (time literal)
timestamp	*string*	"timestamp '2019-06-26 11:55:00'" (timestamp literal)
datetime	*string*	"datetime[yy:ss](2019-06-26 11:55:00)" (datetime literal)
timespan	*string*	"timespan[hh:ss](11:55:00)" (timespan literal)
interval	*string*	"interval '2:12:35' hour to second" (interval literal)
clob	*string*	"Hello Transbase" (clob literal)
blob	*string*	"48656c6c6f205472616e7362617365" (blob literal)

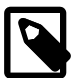
13.1.3.4. Fields - Order / Unkown / Missing

While all fields must be specified in their defined order in the DSV spooler, there is more freedom in the JSON spooler:

- fields can be specified in any order;
- unknown fields - i.e. fields not defined in the table - will be ignored simply;
- missing fields - i.e. fields defined in the table but not specified in the JSON spool file - are replaced by a NULL value or the default value if any has been defined for this field.

13.1.3.5. NULL values - Explicite / Implicite

Missing fields with no defined default value are implictly replaced by a NULL value. Of course the spool file can explicitly specify a NULL value for a field, too. In both cases, if the field does not accept NULL values (NOT NULL) an error is returned.



Note

AUTO_INCREMENT fields are defined automatically as NOT NULL and with default value AUTO_INCREMENT. If explicitly a NULL value is specified in the JSON spool file an error is returned. If the field is not specified in the JSON spool file the default value is taken.

13.1.3.6. Default values

As mentioned above for all fields defined in the table without a specified value in the JSON spool file the *default value* is taken. If no default value is defined for the field the NULL value is taken.

13.1.3.7. LOB objects

Spooling large objects (LOBs) - as separate files, possibly summarized, or inline - all criteria are valid as noted by the *DSV spooler: Spooling LOB objects*.

13.2. External data sources

13.2.1. Remote Database Access

Transbase offers direct and transparent read/write access to remote Transbase databases for distributed queries and data import. Please consult *TableReference* for details on how to connect to a remote Transbase site in an SQL statement.

The following example is a distributed join using two remote databases.

```
INSERT INTO T
SELECT  q.partno, supp.sno
FROM quotations@//server3/db1 q, suppliers@//server5/db2  supp
WHERE q.suppno = supp.sno
```

13.2.2. FILE Tables

Data stored in files may be integrated into the database schema as virtual tables. These FILE tables offer read-only access to those files via SQL commands. They can be used throughout SQL SELECT statements like any other base table.

```
CREATE FILE ('/usr/temp/data.csv')
  TABLE file_table WITHOUT IKACCESS
  (a INTEGER, b CHAR(*))

SELECT a+10, upper(b) from file_table
SELECT b FROM file_table, regular_table
  WHERE file_table.a=regular_table.a
```

If the FILE table is not needed persistently in the schema it can also be used inline within the SELECT statement:

```
SELECT a+10, upper(b)
```

```
FROM ('/usr/temp/data.csv') (a INTEGER, b CHAR(*))
WHERE a > 100
```

FILE tables are primarily designed as an advanced instrument for bulk loading data into Transbase and applying arbitrary SQL transformations at the same time.

Appendix A. The Data Dictionary

The Transbase data dictionary is a set of system tables which define the database structure.

- *Permissions on SystemTables:* The data dictionary is owned by `tbadmin` - usually a database administrator (`dba`), but it belongs to the schema `public`.

  All users are allowed to read the data dictionary, i.e. to retrieve information about the database structure. Reading the data dictionary is in no way different from reading user tables.

- *Locks on SystemTables:* For read access of system tables, a read lock is set as for user tables. However, to avoid bottlenecks on the system tables the read locks are released immediately after the evaluation of the corresponding query.

  Repeated read accesses to the system tables might produce different results, if - in the meantime - a DDL transaction has been committed.

- *Summary of SystemTables:*

  The data dictionary consists of the following tables:

  - *sysdatabase* contains information about database parameters.
  - *syssession* contains information about session parameters.
  - *sysuser* contains users/schemas.
  - *systable* contains tables, views and sequences.
  - *syscolumn* contains colum definitions of tables and views.
  - *sysindex* contains indexes of all tables.
  - *sysview* contains view definitions of all views.
  - *sysviewdep* contains dependencies of views on tables or other views.
  - *systablepriv* contains INSERT, UPDATE and SELECT privileges on tables or views.
  - *syscolumnpriv* contains UPDATE privileges on distinguished fields of tables or views.
  - *sysblob* contains imformation about LOB containers used for BLOBs and CLOBs.
  - *sysconstraint* contains information about CHECK constraints on tables.
  - *sysrefconstraint* contains information about reference constraints on tables. Reference constraints are often referred to as foreign key constraints.
  - *sysdomain* contains the definitions of domains.
  - *sysdataspace* contains information about data spaces.
  - *sysdatafile* contains information about the data files the data spaces are stored in.
  - *loadinfo* contains information about blocks from ROM-files cached on the file system for faster access or modification. It is present only in retrieval databases.
  - *syspsmproc* contains information about the PSM methods defined.
  - *syspsmprocparam* contains information about parameters of the PSM methods defined.

- *syspsmprocpriv* contains information about the privileges granted for the PSM methods defined.

A.1. The sysdatabase(s) Table

The sysdatabase table contains entries for all configuration parameters of a database.

The sysdatabases table in the admin database contains entries for all configuration parameters of all databases managed by the corresponding Transbase service.

Field	Explanation
database_name	The name of the database concerned. [sysdatabases only]
property	The database property or database configuration parameter.
value	The current value of this property for the specified database.
unit	The unit in which the current value is specified where this applies.
datatype	The data type of the property.
comment	Gives useful information on this property. Mostly the collection of possible values and the default value.

A.2. The syssession Table

The syssession table contains entries for all configuration parameters of a session.

Field	Explanation
property	The session property or session configuration parameter.
value	The current value of this property for the current session.
unit	The unit in which the current value is specified where this applies.
datatype	The data type of the property.
comment	Gives useful information on this property. Mostly the collection of possible values and the default value.

A.3. The sysuser Table

The sysuser table contains an entry for each registered database user.

Column	Type
username	CHAR(*)
userclass	CHAR(*)
passwd	CHAR(*)

Column	Type
userid	INTEGER

Primary key: (username)

- *username:* the name of the user/schema.
- *userclass:* has one of the following values:
  - `no access` cannot login on the database.
  - `access` is allowed to access database tables according to the privileges granted, but is not allowed to create, alter or drop database objects like tables, views or indexes.
  - `resource` has privileges as in class `access` plus the right to create, alter or drop database objects, e.g. tables, views and indexes.
  - `dba` has all access rights including the right to add or drop users and to alter or drop objects without being the owner.

    `dba` implicitly has all privileges on any database object including the right to revoke privileges granted by others. Strictly speaking `dba` bypasses all privilege checks.
- *passwd:* Contains the encrypted password of the user. Note that the user's password is not stored anywhere in unencrypted form.
- *userid:* Gives a unique identification for the user. This userid is used in other system tables to refer to users, e.g. in the systable table to denote the owner and the schema of a table.

Explanation:

Upon creation of a database two users are already predefined and cannot be dropped:

- `public` with userclass `no access` and userid `0`
- `tbadmin` with userclass `dba` and userid `1`

These users cannot be dropped but their userclass can be changed. The keywords `public` and `tbadmin` are provided to refer to these users in case-insensitve form. The user `public` is of particular importance in the *GrantPrivilegeStatement*.

Security Alert

The password for `tbadmin` should be changed asap. for security reasons.

A.4. The systable Table

The systable table contains a single entry for each table, view or sequence defined in the database.

Column	Type
tname	CHAR(*)
owner	INTEGER
ttype	CHAR(*)
segno	INTEGER

Column	Type
colno	INTEGER
date	DATETIME[YY:MS]
cluster	SMALLINT
version	INTEGER
indextype	CHAR(*)
schema	INTEGER

Primary key: (schema, tname)

- *tname:* The name of the table, view or sequence.
- *owner:* Denotes the owner of the table or view by the user's userid. To retrieve the owner 's username, join the tables sysuser and systable (see example below).
- *ttype:* The entry has one of the values "R","r","V","v","S".

  A user table has the value "R" or "r", where "R" is used for a table created WITH IKACCESS (this is the default) and "r" for a table created WITHOUT IKACCESS.

  User views are denoted with entry "V". System tables which are visible to the user have entry "v". In effect, all system tables described in this manual are views.

  Sequences are denoted with entry "S".
- *segno:* A positive integer value which denotes the physical segment number of the table or is a negative integer value if the entry is a view. segno is used in other system tables to identify a table or view uniquely.
- *colno:* Contains the number of the columns (fields) of the given table or view.
- *date:* Contains the creation time of the table or view (type {tt DATETIME[YY:MS]}).
- *cluster:* Contains the dataspace number of the table. Tables created without DATASPACE option contain the number 0 here (system data space). Tables created with in a user defined DATASPACE carry a number > 0 which is an internal identifier for the user created DATASPACE. The mapping between dataspace names and numbers is visible in the (virtual) system table "sysdataspace".
- *version:* Contains the version number of the table.
- *indextype:* Contains the index type of the table.
- *schema:* Contains the schema id of the table.

Example A.1. Table Names and Owners of Non-System Tables

```
SELECT s.username as "schema", tname as "name", o.username as "owner"
  FROM systable t, sysuser s, sysuser o
 WHERE schema = s.userid and owner = o.userid  AND ttype in ('R', 'r')
```

A.5. The syscolumn Table

The syscolumn table contains a single entry for each field of each table or view defined in the database.

Column:	**Type:**
tsegno	INTEGER

Column:	Type:
cname	CHAR(*)
ctype	CHAR(*)
domainname	CHAR(*)
defaultvalue	CHAR(*)
suppressed	INTEGER
cpos	INTEGER
ckey	INTEGER
notnull	CHAR(*)
surrid	INTEGER
surref	INTEGER
domainschema	INTEGER

Primary key: (tsegno, cpos)

- *tsegno:* Identifies the table or view to which the entry belongs. The name of the table can be retrieved via a join between systable and syscolumn on the fields tsegno and segno of syscolumn and systable , resp.
- *cname:* Contains name of the column of the table or view.
- *ctype:* contains the base data type of the column. The data type is given as specified in the corresponding `CREATE TABLE` statement. Although data type specifications in TB/SQL are case-insensitive, strings stored in field ctype are all lower-case.
- *domainname:* Contains the domainname if a domain has been used for field definition else NULL. Note that even if a domain has been used, its base type is recorded in field ctype.
- *defaultvalue:* Contains the literal representation of the default value of the field. If no default value has been specified, the value NULL (again explicitly represented as literal) is stored.
- *cpos:* Gives the position of the field in the table (first position is 1).
- *ckey:* Contains the position of this column in the primary key of a table starting with 1 or 0 if the column is not part of the primary key.
- *notnull:* If the `CREATE TABLE` statement has specified a `NOT NULL` clause, this field has the value "Y", otherwise "N".
- *surrid:* If the column contains a surrogate this is a key to the *syssurrogate* table.
- *surref:* If the column contains a surrogate this is a key to the *syssurrogate* table.
- *domainschema:* Contains the schema id of the domain when a domain was specified in the type definition of this column.

Example A.2. Create table statement with resulting entries in syscolumn

```
CREATE TABLE quotations ( Partno    INTEGER,
                          suppno    INTEGER,
                          price    NUMERIC(8,2) NOT NULL,
                          PRIMARY KEY (suppno, partno)
                        )
```

produces the following three entries in syscolumn (not all fields are shown!):

cname	ctype	cpos	ckey	notnull
partno	integer	1	2	N

cname	ctype	cpos	ckey	notnull
suppno	integer	2	1	N
price	numeric(8,2)	3	0	Y

A.6. The sysindex Table

For each index defined for the database, entries in the sysindex table are made. If an n-field index is defined, n entries in sysindex are used to describe the index structure.

Column	Type
tsegno	INTEGER
iname	CHAR(*)
weight	INTEGER
cpos	INTEGER
isuniq	CHAR(*)
isegno	INTEGER
wlistsegno	INTEGER
stowosegno	INTEGER
charmapsegno	INTEGER
delimsegno	INTEGER
ftxttype	CHAR(*)
wlisttype	CHAR(*)
schema	INTEGER
func	CHAR(*)
soundexsegno	INTEGER

Primary key: (tsegno, iname, weight)

- *tsegno:* Identifies the base table which the index refers to. To retrieve the name of the table, perform a join between systable and sysindex on fields tsegno,segno.
- *iname:* Stores the name of the index. Index names are unique with respect to the database.
- *weight:* Serial number for the fields of one index, starting at 1 with the first field specified in the CreateIndexStatement.
- *cpos:* Identifies a field of the base table which the index refers to. To retrieve the name of the field, perform a join between syscolumn and sysindex on the fields (tsegno, cpos) in each table (see the example below).
- *isuniq:* Contains string "Y" if the index is specified as "UNIQUE" , "N" otherwise.
- *isegno:* The field isegno contains the physical segment number of the index. Note that indexes are stored as B*-trees in physical segments.
- *wlistsegno:* Contains for a fulltext index the segment number of the wordlist table, else NULL.
- *stowosegno, charmapsegno, delimsegno:* Contain NULL value for a non-fulltext index. Contain segment number for the stopword table, charmap table, delimiters table, resp., if they have been defined else NULL.

- *ftxttype:* Contains NULL value for a non-fulltext index else one of values "positional" or "word". "positional" is for a POSITIONAL fulltext index, "word" is the default.

- *wlisttype:* Contains NULL value for a non-fulltext index. Contains value "fix" for a specified wordlist, "var" is the default.

The DDL statement

```
CREATE INDEX partprice ON quotations (partno, price)
```

would produce the following two entries in sysindex (only some fields of sysindex are shown):

iname	**…**	**weight**	**cpos**	**isuniq**
partprice	…	1	1	N
partprice	…	2	3	N

To retrieve the names of the index fields in proper order, the following statement is used:

```
SELECT t.tname, c.cname, i.iname, i.weight
FROM systable t, syscolumn  c,
sysindex  i
WHERE t.segno=c.tsegno AND c.tsegno=i.tsegno AND c.cpos=i.cpos
  AND i.iname='partprice' and i.schema = 0 -- == schema public
ORDER BY i.weight
```

A.7. The sysview Table

Contains one record for each view defined in the database.

Column	**Type**
vsegno	INTEGER
viewtext	CHAR(*)
checkopt	CHAR(*)
updatable	CHAR(*)
viewtree	CHAR(*)

Primary key: (vsegno)

- *vsegno:* Contains (negative) integer value which uniquely identifies the view. The same value is used e.g. in systable and syscolumn to refer to the view.

- *viewtext:* Contains SQL SELECT statement which defines the view. This is is used by Transbase whenever the view is used in a SQL statement.

- *checkopt:* Contains "Y" if the view has been defined WITH CHECK OPTION else "N".

- *updatable :* Contains "Y" if the defined view is updatable else "N".

  If a view is not updatable, only SELECT statements may be applied to the view. If a view is updatable, UPDATE, INSERT, and DELETE statements may be applied, too.

  For the definition of updatability and for the semantics of updates on views see the TB/SQL Reference Manual.

- *viewtree:* Reserved for internal use of Transbase.

Note

Additional information for a view is contained in *systable* and *syscolumn* like for base tables (join systable, syscolumn, sysview).

A.8. The sysviewdep Table

Contains dependency information of views and base tables.

A view may be defined on base tables as well as on other previously defined views. If a base table or view is dropped, all views depending on this base table or view are dropped automatically. For this reason the dependency information is stored in sysviewdep.

sysviewdep	
basesegno	INTEGER
vsegno	INTEGER

Primary key: (basesegno, vsegno)

- *basesegno:* Contains the segment number of the base table or view on which the view identified by vsegno depends. basesegno is positive or negative depending on being a base table or view.
- *vsegno:* Identifies the view which depends on the base table or view identified by basesegno. vsegno always is negative.

A.9. The sysblob Table

Contains the information about all BLOB container segments. For each user base table which has at least one column of type BLOB there is one record in the sysblob table. All BLOB objects of the table are stored in the denoted BLOB container.

Column	**Type**
rsno	INTEGER
bcont	INTEGER

Primary key: (rsno)

- *rsno:* Contains segment number of the base table containing BLOB field(s).
- *bcont:* Contains segment number of the BLOB container of the base table.

A.10. The systablepriv Table

Describes the privileges applying to tables of the database. Note that UPDATE privileges can be specified columnwise; those privileges are defined in table *syscolumnpriv*.

systablepriv contains for each table all users who have privileges on this table. Furthermore it contains who granted the privilege and what kind of privilege it is.

systablepriv	
grantee	INTEGER
tsegno	INTEGER
grantor	INTEGER
sel_priv	CHAR(*)
del_priv	CHAR(*)
ins_priv	CHAR(*)
upd_priv	CHAR(*)

Primary key: (grantee, tsegno, grantor)

- *grantee,tsegno,grantor:* These three fields describe who (i.e. the grantor) has granted a privilege to whom (i.e. the grantee) on which base table or view (tsegno). The kind of privilege(s) is described in the other fields. grantor and grantee refer to field userid of table sysuser. tsegno refers to field segno of table systable.
- *sel_priv, del_priv, ins_priv, upd_priv:* These fields describe privileges for SELECT, DELETE, INSERT, UDATE. Each contains one of the values "N", "Y", "G", or in case of updpriv also "S". "Y" means that grantee has received from grantor the corresponding privilege for the table or view identified by tsegno. "G" includes "Y" and additionally the right to grant the privilege to other users, too. "S" (only in updpriv) stands for "selective UPDATE privilege" ; it indicates that there exist entries in table syscolumnpriv which describe column-specific UPDATE privileges granted from grantor to grantee on table tsegno. "N" is the absence of any of the above described privileges. For each record, at least one of the fields selpriv, delpriv, inspriv, updpriv is different from "N".
- *Default:* The owner of a table always has all privileges with GRANT OPTION on the table, by default. Those privileges are recorded in systablepriv, too. Namely, if user 34 creates a table identified by 73, the following entry in systablepriv is made:

  (34, 73, 34, 'G', 'G', 'G', 'G')

A.11. The syscolumnpriv Table

The table syscolumnpriv describes the UPDATE privileges on columns of the database.

Column	**Type**
grantee	INTEGER
tsegno	INTEGER
grantor	INTEGER
cpos	INTEGER
upd_priv	CHAR(*)

Primary key: (grantee, tsegno, grantor, cpos)

- *grantee, tsegno, grantor, cpos:* These four fields describe who (*grantor*) has granted a privilege to whom (*grantee*) on which field (*cpos*) on which base table or view (*tsegno*). The kind of privilege(s) is described in

the other fields. *grantor* and *grantee* refer to field *userid* of table sysuser. *tsegno* refers to field *segno* of table systable.

- *upd_priv:* Contains one of the strings "Y" or "G". "Y" means that *grantee* has received from *grantor* the UPDATE privilege for the specified field on the specified table or view. "G" includes "Y" and additionally the right to grant the privilege to other users, too.

Note

If a user grantee has been granted the same update privilege ("Y" or "G") on all fields on tsegno from the same grantor, then these privileges are recorded via a corresponding single record in table systablepriv.

A.12. The sysdomain Table

Contains at least one record for each created DOMAIN of the database. Several records for one domain are present if more than one check constraints are defined on the domain.

Column	**Type**
name	CHAR(*)
owner	INTEGER
basetype	CHAR(*)
defaultvalue	CHAR(*)
constraintname	CHAR(*)
attributes	CHAR(*)
constrainttext	CHAR(*)
schema	INTEGER

Primary key: (name,constraintname)

- *name:* Contains the name of the domain.
- *owner:* Contains the userid of the creator of the domain.
- *basetype:* Contains the basetype of the domain.
- *defaultvalue:* Contains the defaultvalue (in literal representation) of the domain if there has been declared one otherwise the literal NULL.
- *constraintname, attributes, constrainttext:* These fields describes a domain constraint if there has been defined one else they are all NULL. Field attributes contains the value `IMMEDIATE` (for use in later versions), constraintname stores the user defined constraintname if any else NULL, constrainttext stores the search condition of the constraint.
- *schema:* Contains the schema id of the table.

Note

If $n > 1$ constraints are defined, all n records redundantly have the same owner, basetype, defaultvalue.

A.13. The sysconstraint Table

Contains one record for each table constraint. Constraint types are PRIMARY KEY or CHECK(…) constraints.

Column	Type
segno	INTEGER
constraintname	CHAR(*)
attributes	CHAR(*)
constrainttext	CHAR(*)
cpos	INTEGER

Primary key: (segno,constraintname)

- *segno:* Contains the segment of the table the constraint refers to.
- *constraintname:* stores the user defined constraintname if any or a unique system defined constraintname.
- *attributes:* Field attributes has the constant value `IMMEDIATE` (for more general use in later versions).
- *constrainttext:* stores the constraint text which is either of the form PRIMARY KEY (...) or CHECK(...).
- *cpos:* has value `-1` except in the case that the constraint was initially defined in a domain that was used for the type of a field in the table the constraint refers to. In this particular case the constrainttext contains the keyword VALUE instead of an explicit field reference and field cpos contains the position of the field.

Note

Reference constraints (FOREIGN KEY ...) are stored in table sysrefconstraint.

A.14. The sysrefconstraint Table

Contains records to describe reference constraints (FOREIGN KEY ...).

Field	Type
constraintname	CHAR(*)
attributes	CHAR(*)
srcsegno	INTEGER
srccpos	INTEGER
tarsegno	INTEGER
tarcpos	INTEGER
delact	CHAR(*)
updact	CHAR(*)

Primary key: (srcsegno,constraintname)

- *constraintname:* Contains the name of the constraint (explicitly by user or system defined).

- *attributes:* Contains value `IMMEDIATE` (for more general use in later versions).
- *srcsegno, tarsegno:* Contain the segment number of the referencing table and the referenced table, resp.
- *delact:* Contains the action to be performed with a referencing record rf when rf looses its referenced record rd due to a deletion of rd. The field has one of the string values "NO ACTION" , "CASCADE" , "SET NULL" , "SET DEFAULT". For the semanticsof these actions, (see TB/SQL Reference Manual, CreateTableStatement, ForeignKey).
- *updact:* Contains the action to be performed on a referencing records rf when rf looses its referenced record rd due to a update of rd. The field has the constant string values "NO ACTION" and is for more general use in later versions. For the semantics of this action, (see TB/SQL Reference Manual, CreateTableStatement, ForeignKey).
- *srccpos, tarcpos:* Contains a pair of field positions (>=1) which correspond to the referencing and referenced field of the reference constraint. If the reference constraint is defined over a n-ary field combination, then n records are in sysrefconstraint where all values except srcpos and tarcpos are identical and srcpos and tarcpos have the values of the corresponding field positions.

```
CREATE TABLE sampletable
(   keysample INTEGER,
    f1        INTEGER,
    f2        INTEGER,
    CONSTRAINT ctref
        FOREIGN KEY (f1,f2)
        REFERENCES reftable(key1,key2)
        ON DELETE CASCADE
)
```

Assume that sampletable and reftable have numbers 13 and 19, resp., and that reftable has fields key1 and key2 on positions 7 and 8.

Then the following two records are inserted in sysrefconstraint:

constraint-name	**attributes**	**srcsegno**	**srccpos**	**tarsegno**	**tarcpos**	**delact**	**updact**
ctref	IMMEDIATE	13	2	19	7	CASCADE	NO ACTION
ctref	IMMEDIATE	13	3	19	8	CASCADE	NO ACTION

A.15. The sysdataspace Table

Contains records to describe the dataspaces.

Field	**Type**
name	VARCHAR(*)
autoextend	INTEGER
dspaceno	INTEGER
maxsize	INTEGER
online	BOOL
defaultpath	VARCHAR(*)
available_pg	INTEGER
size_pg	INTEGER

Primary key: (name)

- *name:* User defined name for the dataspace.
- *autoextend:* Dataspace is automatically extended by a new datafile having the specified size. For disabling set *autoextend* to *OFF*
- *dspaceno:* Corresponding number for the dataspace.
- *maxsize:* Maximum size for the dataspace.
- *online:* Contains `false` if the dataspace has been set offline else `true`.
- *defaultpath:* The directory where the datafiles of this dataspace are stored by default.
- *available_pg:* Number of actually free pages for the dataspace.
- *size_pg:* Number of actually allocated pages for the dataspace.

A.16. The sysdatafile Table

Contains records to describe the files attached to dataspaces.

Field	**Type**
name	VARCHAR(*)
dspaceno	INTEGER
datafile	VARCHAR(*)
size	INTEGER
lobcontainer	BOOL
available	INTEGER

Primary key: (name, datafile)

- *name:* User defined name for the dataspace.
- *dspaceno:* Corresponding number for the dataspace
- *datafile:* Pathname of file attached to this dataspace. The filenames for user defined dataspaces are system generated. They contain 2 numbers where the first number is the dspaceno of the dataspace where the file belongs to, and the second number is a running number over all existing files in the dataspace.
- *size:* Maximum size of the datafile.
- *lobcontainer:* Datafile is dedicated to LOB's Contains `true` if the datafile is dedicated to LOB's else `false`.
- *available:* Number of available pages.

A.17. The loadinfo Table

Describes the status of the diskcache in Retrieval Databases.

For each page which is in the diskcache there is one record in loadinfo.

Field	**Type**
segment	INTEGER
rompage	INTEGER
diskpage	INTEGER
flag	INTEGER

Primary key: (segment, rompage)

- *segment:* Contains the segment number of the page.
- *rompage:* Contains the page number of the page in the ROM address space.
- *diskpage:* Contains the page number of the page in the diskfile address space.
- *flag:* Contains the value 1 if the page has just been cached via a LOAD STATEMENT and the value 2 if the page has been changed with some INSERT or UPDATE or DELETE operation.

A.18. The syskeyword Table

List all SQL keywords recognized by the SQL compiler of the connected database..

Field	**Type**
keyword	STRING

Primary key: (keyword)

A.19. The syspsmproc Table

The syspsmproc table contains a single entry for each PSM method in this database.

Column	**Type**
schema	INTEGER
procname	CHAR(*)
procid	INTEGER
owner	INTEGER
params	INTEGER
procbody	CLOB
returntype	CHAR(*)

- *schema:* Identifies the schema the method belongs to. The name of the resource can be retrieved via a join between sysexternal and sysexternalmethod on the fields rkey.
- *procname:* Unique name that identifies the method in the schema it belongs to.

- *procid:* Internal ID of the method
- *owner:* Identifies the owner (creator) of the method.
- *params:* The number of parameters of this method
- *procbody:* The defining body of this method.
- *returntype:* The result type of the method if it is a function. For procedures it is NULL.

A.20. The syspsmprocparam Table

The syspsmprocparam table contains a single entry for each parameter of each method of each PSM method in this database.

Column	**Type**
procid	INTEGER
parampos	INTEGER
paramtype	CHAR(*)
inoutmode	CHAR(*)

- *procid:* Unique numeric key that identifies a method.
- *parampos:* The parameter position starting with 1.
- *paramtype:* The SQL data type of this parameter.
- *inoutmode:* The parameter mode, i.e. IN, OUT or INOUT.

A.21. The syspsmprocpriv Table

Describes the privileges needed to use a PSM method.

Column	**Type**
grantee	INTEGER
procid	INTEGER
grantor	INTEGER
use_priv	CHAR(1)

- *grantee:* Describes whom this privilege is granted.
- *procid:* Identifies the method to be granted.
- *grantor:* Id of the user granting the privileges
- *use_priv:* Describes the granted privileged. 'Y' means that grantee has USAGE privilege and 'G' means that grantee may also grant USAGE to other users.

Grantor and grantee refer to field userid of table sysuser.

Appendix B. Sample Database

The database `SAMPLE` used in the exercises throughout this manual consists of three tables *SUPPLIERS*, *QUOTATIONS* and *INVENTORY* that are described below.

Table B.1. Table SUPPLIERS

suppno	name	address
51	DEFECTO PARTS	16 BUM ST., BROKEN HAND WY
52	VESUVIUS, INC.	512 ANCIENT BLVD., POMPEII NY
53	ATLANTIS CO.	8 OCEAN AVE., WASHINGTON DC
54	TITANIC PARTS	32 LARGE ST., BIG TOWN TX
57	EAGLE HARDWARE	64 TRANQUILITY PLACE, APOLLO MN
61	SKY PARTS	128 ORBIT BLVD., SIDNEY
64	KNIGHT LTD.	256 ARTHUR COURT, CAMELOT

Table B.2. Table INVENTORY

partno	description	qonhand
207	GEAR	75
209	CAM	50
221	BOLT	650
222	BOLT	1250
231	NUT	700
232	NUT	1100
241	WASHER	6000
285	WHEEL	350
295	BELT	85

Table B.3. Table QUOTATIONS

suppno	partno	price	delivery_time	qonorder
51	221	.30	10	50
51	231	0.10	10	0
53	222	0.25	15	0
53	232	0.10	15	200
53	241	0.08	15	0
54	209	18.00	21	0
54	221	0.10	30	150
54	231	0.04	30	200
54	241	0.02	30	200
57	285	21.00	4	0
57	295	8.50	21	24
61	221	0.20	21	0
61	222	0.20	21	200

suppno	partno	price	delivery_time	qonorder
61	241	0.05	21	0
64	207	29.00	14	20
64	209	19.50	7	7

Appendix C. Precedence of Operators

The Table below defines the precedence of operators. A precedence of 1 means highest precedence.

Associativity within a precedence level is from left to right.

Precedence	Operators
1	SUBRANGE, SIZE OF
2	<timeselector> OF, CONSTRUCT
3	CAST
4	PRIOR, +, – (unary) , BITNOT
5	BITAND, BITOR
6	*, /
7	+, – (binary), ||, < <= = <> >= >
8	LIKE, MATCHES, IN, BETWEEN, SUBSET, CONTAINS
9	NOT
10	AND
11	OR
12	SELECT
13	INTERSECT
14	UNION, DIFF