

Stored Procedures and User-Defined Functions

Transaction Software GmbH
Willy-Brandt-Allee 2
D-81829 München
Germany
Phone: +49-89-62709-0
Fax: +49-89-62709-11
Email: info@transaction.de
<http://www.transaction.de>

Version 6.8.1.40
November 02, 2010

Contents

1	Introduction	3
1.1	Abbreviations	4
1.2	Copyright Information	4
2	Getting Started	5
2.1	A 'Hello World from Java' Example	5
2.1.1	Write a Stored Procedure	5
2.1.2	Load your code	6
2.1.3	Publishing your Methods in SQL	6
2.1.4	Execute your Methods	6
2.2	A 'Hello World from C' Example	7
2.2.1	Write a User-defined Function	7
2.2.2	Compile your code	7
2.2.3	Load your code	8
2.2.4	Publishing your Methods in SQL	8
2.2.5	Execute your Methods	8
3	SQL Extensions Reference	9
3.1	CREATE EXTERNAL statement	9
3.2	ALTER EXTERNAL statement	10
3.3	ALTER EXTERNAL OPTION statement	11
3.4	DROP EXTERNAL statement	12
3.5	CREATE PROCEDURE/FUNCTION statement	13
3.6	ALTER PROCEDURE/FUNCTION statement	17

3.7	CALL statement	19
3.8	DROP PROCEDURE / FUNCTION statement	19
3.9	GRANT USAGE statement	20
3.10	REVOKE USAGE statement	20
4	Implementation Guide - Java	22
4.1	System Architecture	22
4.2	Transactions	22
4.3	Java Type Mappings	23
4.4	Parameter Modes	24
4.5	User-defined Type Mapping	24
4.6	Method Resolution	25
4.7	Null Values	26
4.8	Build-In Java Functions	26
4.9	Security Manager	28
4.10	Class Loading	28
4.11	Pattern for a Stored Procedure	28
4.12	Parameter Values	29
4.13	Pattern for a User-Defined Value Function	31
4.14	Pattern for a User-Defined Table Function	32
4.15	Pattern for Generic Table Functions	34
4.16	Static Variables	38
4.17	Java Wrappers	38
4.18	Choosing your JVM	39
4.19	Remote Method Debugging	39
4.20	Evaluating Debug Output	40
5	Implementation Guide - Native Libraries	41
5.1	Limitations	41
5.2	Compiling and Linking	41
5.3	Pattern for Generic Table Functions	41

6 Database Client APIs	44
6.1 JDBC	44
6.2 TBX	45
7 Troubleshooting	47
7.1 The retrieval or update of an external resource, method, or option failed, the system catalog might be corrupted (Error 16600)	47
7.2 Internal JDBC Driver not correctly installed (Error 16700)	47
7.3 Cannot create Java VM (Errors 16705 and 16720)	47
7.4 Transbase crashes on first access to a UDF or STP	48
7.5 System hangs or crashes during heavy UDF or STP load on Unix / Unexpected Signal : 11	48
7.6 Exceptions in classes from the Transbase runtime package	48
8 Transbase Data Dictionary Extensions	49
8.1 The sysexternal Table	49
8.2 The sysexternalmethod Table	50
8.3 The sysexternalpriv Table	51

Chapter 1

Introduction

Stored Procedures (STP) and User-Defined Functions (UDF) enable any Transbase user to extend server side functionality by providing user-defined routines explicitly callable via SQL statements or implicitly by event driven database triggers. Additionally a Java enabled Transbase server also allows for implementing, loading, and executing Java code.

The STP extension of Transbase aims to meet the the standard for SQL-invoked routines as defined in American National Standard for Information Technology - Database Language SQL - Part 4: Persistent Stored Modules (SQL/PSM), ANSI/ISO/IEC 9075-4-1999.

Many features of Transbase STP extension for Java are compatible to the SQL standard extension as defined in American National Standard for Information Technology - Database Languages - SQLJ - Part 1: SQLJ Routines using the Java™ Programming Language, ANSI NCITS 331.1-1999.

However, in some cases, it was more suitable to leave the standard aside, in order to full exploit special features available in the Transbase database system. And finally there were cases where the standard was implemented but additional features or variations were introduced to make STPs and UDFs more powerful and easier to use. Deviations from the standard are annotated.

To value the safety of running interpreted code inside a virtual machine over using the performance oriented approach of loading user-defines libraries into the address space of the database kernel, we will first discuss the Java SQL extension. Chapter 2 of this manual is a simple example on how to create and use stored procedures and user-defined functions. The first section addresses code written in Java and the second section attends to code residing in dynamic libraries. Chapter 3 is a reference of the Transbase Extension accompanied by numerous examples to illustrate the SQL extension reference. Chapter 4 covers Java specific SQL extension topics while Chapter 5 addresses the properties of native code in libraries. Chapter 6 illustrates how the STP extension is used via a database

client API. Chapter 7 covers some troubleshooting issues. Finally, chapter 8 is an overview over the Transbase Data Dictionary extensions for the STP extension.

1.1 Abbreviations

In this manual uses a set of common abbreviations. Consult the following table if any abbreviations should be unclear:

JRE:	Java Runtime Environment
JDB:	Java Debugger
JDE:	Java Developer Environment
JNI:	Java Native Interface
JVM:	Java Virtual Machine
PSM:	Persistent Stored Module
SDK:	Standard Developer Kit
STP :	Stored Procedure
TBJRE:	Transbase Java Runtime Environment
UDF:	User-Defined Function

1.2 Copyright Information

Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

The accompanying Java Platform software is copyrighted by Sun Microsystems, Inc. and is used and redistributed by permission, according to the Sun Binary Code License Agreement.

Chapter 2

Getting Started

2.1 A 'Hello World from Java' Example

The following tutorial will guide you through the required steps to write, compile, load, publish, and execute a stored procedure and user-defined function on a Java enabled Transbase Server in two simple 'Hello World' examples.

2.1.1 Write a Stored Procedure

Create a new file, name it ClassHelloWorld.java and paste the following code into it.

Example:

```
import java.sql.*;
public class ClassHelloWorld
{
    // this is a simple stored procedure
    public static void ProcHelloWorld() throws SQLException,
                                           ClassNotFoundException
    {
        Class.forName("transbase.jdbc.Driver");

        Connection conn =
            DriverManager.getConnection("jdbc:default:connection");

        Statement stmt = conn.createStatement();
        stmt.execute("CREATE TABLE HelloWorld (mytext CHAR(*))");
        stmt.execute("INSERT INTO HelloWorld VALUES" +
            "('Hello World from Java!')");
    }
}
```

```
    stmt.close();
    conn.close();
    return;
}

// this is a simple user-defined function
public static String FuncHelloWorld()
{
    return "Hello World from Java!";
}
}
```

2.1.2 Load your code

You can compile and debug your code with your Java Developer Kit or any available Java IDE and load compiled classes into the database. You can also pack your classes into JAR files. There are two ways for adding JARs to the system. JARs are either stored in the system just like classes, refer to 'CreateExternalStatement' for details. Otherwise they are stored on the file system and are referenced via the CLASSPATH option as discussed in the 'AlterExternalOption' section.

Run the Java Compiler javac on your command line:

```
javac ClassHelloWorld.java
```

This generates a new file ClassHelloWorld.class. Have tjbdbc.jar available in you CLASSPATH variable for imports. To get your binary into the database, first connect to it with your tbi and type in the following statement:

```
CREATE EXTERNAL FROM C:\\temp\\ClassHelloWorld.class;
```

2.1.3 Publishing your Methods in SQL

In the next step, you tell Transbase, which of your Methods you want to have available in SQL and how you want to name them. So type the following into tbi:

```
CREATE PROCEDURE CreateHelloWorld() MODIFIES SQL DATA
    EXTERNAL NAME "ClassHelloWorld.ProcHelloWorld";
CREATE FUNCTION HelloWorld() RETURNS CHAR(*) EXTERNAL NAME
    "ClassHelloWorld.FuncHelloWorld";
```

2.1.4 Execute your Methods

To execute a stored procedure use the CALL statement:

```
CALL CreateHelloWorld();
SELECT * FROM HelloWorld;
```

A user-defined function could be used in the select clause:

```
SELECT HelloWorld();
```

To get an overview on already published functions try the following:

```
SELECT methodname FROM sysexternalmethod;
```

For a complete overview of built-in Java functions, please refer to this table.

2.2 A 'Hello World from C' Example

The following tutorial will guide you through the required steps to write, load, publish, and execute a Java stored procedure and user-defined function on a Transbase Server in a simple 'Hello World' example.

2.2.1 Write a User-defined Function

Create a new file, name it HelloWorld.c and paste the following code into it.

Example:

```
/* this is a simple user-defined function */  
char *FuncHelloWorld(void)  
{  
    return "Hello World from C!";  
}
```

2.2.2 Compile your code

Depending on the platform you are using there is a variety on how to compile your code into a dynamic library. Here is a list of important requirements to consider:

- must use `_cdecl` call convention
- make sure your functions are properly exported
- compile as resource library, i.e. without entry point

Here some exemplary command lines

Windows (MS Developer Studio 6.0) :

```
cl /Gd /LD HelloWorld.c /OUT:libHelloWorld.dll
```

Linux (gcc/g++):

```
gcc -shared -fPIC HelloWorld.c -o libHelloWorld.so
```

Solaris (SunOS 5.7):

```
cc -G HelloWorld.c -o libHelloWorld.so
```

AIX (version 3 patch level 4):

```
cc -bM:SRE -bnoentry -bexpall HelloWorld.c -o libHelloWorld.so
```

HP-UX:

```
cc -b HelloWorld.c -o libHelloWorld.so
```

2.2.3 Load your code

To get the code into the database, first connect to it with your tbi and type in the following statement:

```
CREATE EXTERNAL FROM C:\\temp\\libHelloWorld.dll;
```

or

```
CREATE EXTERNAL FROM /tmp/libHelloWorld.so;
```

2.2.4 Publishing your Methods in SQL

In the next step, you tell Transbase, which of your Methods you want to have available in SQL and how you want to name them. So type the following into tbi:

```
CREATE FUNCTION HelloWorld() RETURNS CHAR(*) EXTERNAL NAME  
"libHelloWorld.FuncHelloWorld";
```

2.2.5 Execute your Methods

A user-defined function could be used in the select clause:

```
SELECT HelloWorld();
```

To get an overview on already published functions try the following:

```
SELECT methodname FROM sysexternalmethod;
```

Chapter 3

SQL Extensions Reference

In order create and manipulate external resources, STPs, and UDFs various extensions have been made to TB/SQL. This chapter is the Reference Manual to those extensions.

3.1 CREATE EXTERNAL statement

Function: Loads an external resource and supplies a unique name for it.

Syntax:

```
CreateExternalStatement ::=
    CREATE EXTERNAL
    ResourceSpec
    FROM FileName [LOCAL]
    [[NOT] FOR DEBUG]
    [COMMENT "comment"]
```

```
ResourceSpec ::=
    [CLASS|JAR] ResourceName]
```

Explanation: An external resource of the type specified in the ResourceSpec is loaded to the database and will be available under its unique ResourceName for further use. At this time external resources are always java resources. Thus the available resource types are only java resources, i.e. java class files. Transbase also offers the possibility to load the corresponding source files and you can leave compilation to the Transbase Java Runtime Environment.

For loading an external class without ResourceSpec, the class has to be a top-level class, i.e. the class must not belong to a package. Then FileName is then used to compute the ResourceName of the class. If classes residing in packages are used, then the ResourceName must explicitly state the fully qualifying method name (including package information) in Java notation.

Alternatively packages can be moved to a JAR file. This file can either be loaded into the system via this statement, there it becomes an integral part of the database, or it may be referenced through the CLASSPATH option, as described in 'AlterExternalOption' statement.

For accessing any resources, at least one of their methods has to be published by the corresponding 'CreateProcedureStatement'.

FileName is an identifier for the file. The syntax of FileName is defined via the underlying operating system. Since for Java the filename always specifies the class name and the file extensions specifies the resource type and currently no resources of other programming languages with conflicting resource names can be loaded, one can completely omit ResourceSpec. If both, ResourceSpec and FileName are available, the systems performs cross checks to ensure consistency. Without the LOCAL keyword, the specified file is read by the client application and transferred to the tbkernel process. If the file is accessible by the tbkernel process via FileName, the LOCAL clause can be used to speed up the transfer process: in this case the tbkernel process directly accesses the file under the specified name which must be an absolute path name.

The Debug clause is optional and reserved for future versions, where remote debugging of stored resources on resource basis will be enabled. Remote debugging is currently available through the procedure described in 'RemoteMethodDebugging'.

In the Comment clause you can describe the functionality, version, author, and other additional information on this resource.

Note that the CreateExternalStatement is a custom Transbase SQL extension and is not a standardized part of SQL3 or SQLJ. However, a subset of functionality of this statement offers the same functionality as the SQLJ.INSTALL_JAR built-in procedure introduced by the ANSI standard.

Privileges: The user must have userclass DBA or RESOURCE. For the definition of userclasses see the chapter 'GrantUserclassStatement' in your SQL Reference Manual.

3.2 ALTER EXTERNAL statement

Function: Modify an already loaded external resource.

Syntax:

```
AlterExternalStatement ::=
  ALTER EXTERNAL
  [CLASS {BINARY|SOURCE}]
  [{ENABLE|DISABLE}]
  ResourceName
  [FROM FileName [LOCAL]]
  [[NOT] FOR DEBUG]
  [COMMENT "comment"]
```

Explanation: Enable or Disable resources without changing access rights. You can replace loaded resources with newer versions, change the debug flag or the associated comment. All changes leave the usage privileges of this resource and its routines unchanged.

If you replace a Java class that has already been loaded into your virtual machine, the system will automatically unload this class, and load the new class to the system.

Note that the `AlterExternalStatement` is a custom Transbase SQL extension and is not a standardized part of SQL3 or SQLJ. However, a subset of functionality of this statement offers the same functionality as the `SQLJ.REPLACE_JAR` built-in procedure introduced by the ANSI standard.

Privileges: The user must be owner of the external resource.

3.3 ALTER EXTERNAL OPTION statement

Function: Modifies the options with which your Java Virtual Machine will be started.

Syntax:

```
AlterExternalOption ::=
    ALTER EXTERNAL OPTION
    {JAVA|JDK|CLASSPATH|JAVADEBUG|JAVAPERMISSIONS}
    "optionstring"
```

Explanation: The `JAVA` option is a set of command line options you want to use with your virtual machine. Consult your Java documentation for more information or type `java -h` and `java -X` on the command line to get short information on standard and non-standard Java options, if you have a SDK or JRE installation available on your machine. The default setting of these options for the Transbase Java Runtime Environment is `'-Xms2m -Xmx20m'` in order to get some control over the Java VMs heap allocation by restricting it to a range from two to twenty megabytes.

The option `JDK` points to the base directory of your custom JDK installation. It is strongly recommended to use an SDK (Standard Developer Kit) installation instead of the smaller JRE (Java Runtime Environment) because the class compiler is only available in the SDK. However, if you do not require class compilation, then a simple JRE installation will be sufficient. An empty option string makes the system use the standard JDK shipped with your Transbase installation which is located in the `tbjre` subdirectory. SDKs and JREs prior to version 1.2 will not function with the Transbase system. For reasonable performance it is recommended to use SDKs and JREs featuring the Java HotSpot Server, which is available with release 1.3 and later. MS Windows users have to download and install the HotSpot Server separately for it is not included in the SDK Edition of Java. Note that any Java-enabled Transbase distribution always includes an up-to-date HotSpot Server in its `TBJRE`.

The `CLASSPATH` option holds directories and jar files you want the JVM to check for required classes. If the system fails to find a resource in the set of loaded resources

it consults the CLASSPATH to search the local file system. This is very similar to the CLASSPATH environment variable as known for the JVM. This is a way to make resources available that are not stored in the Transbase system itself, in particular jar files that cannot be loaded to the system with this Transbase version. For information on security issues involved in usage of external resources see 'ClassLoadingAndCompilation'.

The JAVADEBUG option is used to supply the JVM with a set of additional options for starting the JVM in debug mode, leaving out the '-Xrunjdpw:' prefix. Check your java documentation for further information. The default setting of JAVADEBUG in the TBJRE is:

```
transport=dt_socket,address=5000,suspend=y,server=y
```

This defines the transport protocol and the address where the Transbase server listens for a connecting Java Debugger (JDB). A simple sanity check is performed if the JVM is to be started in debug mode, where the existence of the tokens 'transport=', 'address=', 'suspend=y', 'server=y' and the absence of 'Xrunjdpw' is demanded.

The JAVAPERMISSIONS option makes the installed SecurityManager configurable. One can either drop all restrictions by globally permitting any access:

```
ALTER EXTERNAL OPTION JAVAPERMISSIONS "all"
```

or by permitting access on a permission class basis by specifying a comma separated list the full classnames of permissions to be granted:

```
ALTER EXTERNAL OPTION JAVAPERMISSIONS
"java.io.FilePermission, java.net.NetPermission"
```

Note that a change of any option requires a restart of the JVM to take effect.

Unrecognized options will be ignored by the Virtual Machine, in order to guarantee a reliable startup behavior. The actual setting of the JVM properties can be inspected by checking the System.Properties of a Virtual Machine with user-defined functions.

Note that AlterExternalOption is a custom Transbase SQL extension and is not a standardized part of SQL3 or SQLJ. However, a subset of functionality of this statement offers the same functionality as the SQLJ.ALTER_JAVA_PATH built-in procedure introduced by the ANSI standard.

Privileges: The current user must have userclass DBA.

3.4 DROP EXTERNAL statement

Function: Remove an external resource from the system.

Syntax:

```
DropExternalStatement ::=
  DROP EXTERNAL
  [CLASS|JAR]
  ResourceName
```

Explanation: With the DropExternalStatement you may completely remove a resource from your system. All adjoined procedures, functions and privileges connected to resource or routines will also be dropped.

Note that the DropExternalStatement is a custom Transbase SQL extension and is not a standardized part of SQL3 or SQLJ. However, a subset of functionality of this statement offers the same functionality as the SQLJ.REMOVE_JAR built-in procedure introduced by the ANSI standard.

Privileges: The user must be owner of the external resource or of userclass DBA.

3.5 CREATE PROCEDURE/FUNCTION statement

Function: Specify an SQL name for an external function or procedure.

Syntax:

```

CreateProcedureStatement ::=
    CREATE PROCEDURE
    SQLName SQLSignature
    [SQLProperties]
    [DYNAMIC RESULT SETS integer]
    ExternalBlock

CreateFunctionStatement ::=
    CREATE FUNCTION
    SQLName SQLSignature
    RETURNS { SQLDataType | TABLE [OutFieldList] }
    [SQLProperties]
    [CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT]
    ExternalBlock

SQLSignature ::= ( [SQLParamter [, SQLParameter]...] )

SQLParameter ::= [ParameterMode] [SQLIdentifier] SQLDataType

ParameterMode ::= IN|OUT|INOUT|INSTANCE|INSTANCEOUT

OutFieldList ::= ( OutField [, OutField]... )

OutField ::= [OUT] [SQLIdentifier] SQLDataType

SQLProperties ::=
    [[NOT] DETERMINISTIC]
    [DataAccessIndication]
    [WITH[OUT] OWNER RIGHTS]
    [{CHECK|IGNORE} EXCEPTION]

```

```

[COMMENT "comment"]

DataAccessIndication ::=
  {NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA}

ExternalBlock ::= EXTERNAL
  [PARAMETER STYLE {JAVA | C}]
  [LANGUAGE {JAVA | C}]
  NAME "MethodFullName[JavaSignature]"

JavaSignature ::= ( [ JavaParameterList ] ) returns JavaDatatype

JavaParameterList ::= JavaDatatype [, JavaDatatype]...

JavaDatatype ::= -- See below.

```

Definitions:

CreateProcedureStatement and CreateFunctionStatement serve to publish external functions and procedures (i.e. Java methods) to the database system by defining procedure and function name, parameter types and, in case of a function, the result data type of an external function or procedure. Additionally a set of options can be defined for optimization, supervision and documentation of an external resource. Finally a reference to the actual external resource and its defining programming language is delivered.

SQLName This is the name of the newly created procedure or function. With this name you can address a function from SQL statements for execution. However, an external routine is not only identified by its name, but also by its signature and the context it is called from. This has the consequence that a set of routines may have the same name. This concept is well known in object oriented programming languages as name overloading and this behavior was implemented to reproduce this concept in TB/SQL. Finally the context of a function call decides on its result type. In a CALL SQLName-statement a stored procedure returning void will be called while SQLNames used in other TB/SQL statements always refers to a function returning a value of some type.

SQLSignature Specifies the SQL parameter data types for a procedure. A complete list of available SQL data types in Transbase along with their legal mappings to Java data types can be found in the 'JavaTypeMappings' subsection of the chapter Java Specifics in this manual and in the SQL Reference Manual. Length and precision specifications of SQL data types are irrelevant for type mapping and thus may be omitted in the SQLSignature clause. Note that two methods of the same SQLName have to differ in at least one SQLDatatype of their signature, for type mapping has to be well-defined for SQL and Java types. See 'JavaTypeMappings' subsection for more details on name overloading. At most MAXEXTPARAMNO (=64) are allowed.

returns Specifies the result data type of the function. Note that this is an SQL Data type and that for return values always the default type mapping is used. Of course

you may cast results in SQL if another type is required or explicitly map them with the optional `JavaSignature`. Resulttype `TABLE` may only be used with functions. If resulttype is `TABLE` then the function may only have `IN` parameters in its `SQLSignature` and only Parametermode `OUT` is allowed in the `OutFieldList`. If `OutFieldList` is missing, then the function is a Generic Table Function.

ParameterMode Specifies whether the SQL parameter is input only (`IN`), output only (`OUT`) or both input and output (`INOUT`). The `ParameterMode` may only be specified for stored procedures, since user-defined functions only use parameters in `IN` mode. To meet the object paradigm of the Java programming language, two additional modes are introduced: `INSTANCE` and `INSTANCEOUT`. Consult 'ParameterModes' for more detailed information. These two modes are not defined in the ANSI standard. If you do not want to use non-standard `ParameterMode`, you still can write a wrapper class for the exported methods in question, and have this wrapper do the casts and type conversions (see 'JavaWrappers').

deterministic Optimizer hint. The deterministic option tells the optimizer that for a given set of argument values, the procedure or function returns the same values for `OUT` and `INOUT` parameters and function result.

not deterministic Optimizer hint. Specifies that the procedure or function does not have the deterministic property. This is the default, if neither deterministic nor not deterministic is specified.

DataAccessIndication Specifies the SQL facilities that the Java method is allowed to perform. The restrictions apply directly to the specified method itself and to any methods that it invokes, directly or indirectly (e.g. through triggers). However the access indication does not apply to finalizer methods. Finalizer are always executed with a `NO SQL` access indication. If `DataAccessIndication` is not specified, then `contains sql` is the default. Any violation of `DataAccessIndication` will result in an `SQLException` and is a hard error. In that case the current transaction will be rolled back.

no sql The method cannot invoke SQL operations.

contains sql The method can invoke SQL operations, but cannot read or modify SQL data. I.e. the method cannot perform SQL open, close, fetch, select, insert, update, or delete operations. The `contains sql` option is the default `DataAccessIndication`.

reads sql data The method can invoke SQL operations, and can read SQL data, but cannot modify SQL data. I.e. the method cannot perform SQL insert, update, or delete operations.

modifies sql data The method is allowed to invoke SQL operations and to read and modify SQL data.

with[out] owner rights This options allows the programmer to specify, whether any user executing his procedure or function inherits the creator's specific owner rights for the duration of the function call. This allows users to use and manipulate data in a controlled way through 'privileged' stored procedures or user-defined functions. With `owner rights` is the default if none of these options is specified. These options are not contained in the ANSI standard.

check exception Optimizer hint. If check exception is specified, then explicit exception checking is performed after any returning procedure and after any function returning a NULL value, that is in any case when an exception may have occurred. This option is not contained in the ANSI standard.

ignore exception Optimizer hint. The Transbase system will not check for exceptions. The programmer has to take care that his routine does not throw any exceptions and to take appropriate actions to detect and handle exceptions in a reasonable way for his application, e.g. try/catch blocks in Java, indicator variable as INOUT or OUT variable of a stored procedure and null-checking. This option is not contained in the ANSI standard.

comment a user-defined comment on the function. The maximum length of a comment is MAXCOMMENTSIZE (=255).

dynamic result sets Specifies that the Java method may return SQL result sets. dynamic result sets can only be specified in a create procedure statement, not in a create function statement. The specified integer value is the maximum number of result sets that the method will return. The maximum value is 255. The dynamic result sets clause is of no practical use at the moment, since the current JDBC Implementation does not yet support Callable statements. You may use a Table Function returning one table instead.

returns null on null input and called on null input Specifies the action to be taken when an argument of a function call is null. If you specify returns null on null input, then at runtime if the value of any argument is null, then the result of the function is set to null, and the function body is not invoked. If you specify called on null input, then a runtime exception is thrown if an SQL null value is found for an argument whose Java data type is boolean, byte, short, int, long, float, or double. If you do not specify either returns null on null input or called on null input, then called on null input is the default.

external Specifies that the create statement defines an SQL name for a routine written in a programming language other than SQL. Optional since Java is currently the only supported external language.

parameter style java Specifies that the runtime conventions for arguments passed to the external routine are those of the Java programming language. Optional since Java is currently the only supported external language.

Name Specifies the name of a Java method. A reference to the SQL name is effectively a synonym for the specified Java method. The external name is specified in a character string literal representing a fully qualifying method name in Java notation, e.g. "[[jarname:]packagename.]classname.methodname". The surrounding double-quotes are the delimiters of that literal. If a class is to be retrieved from the JDK Archives or via the CLASSPATH option, then the jarname has to be added. Transbase cannot check jarname validity, but will try to retrieve the class via the alternative 'ClassLoader'. Note that classes residing in jars may also belong to packages and the packagename is required for the fully qualifying method name. Note that MethodFullName is case sensitive.

JavaSignature this optional second signature offers the possibility to substitute default type mappings with alternative type mappings as described in table 'JavaTypeMappings'. You can also supplement your custom type mapping, if your custom class

supports this as described in subsection 'User-defined Type Mapping'. Usage of the `JavaSignature` requires a complete signature, i.e. one `JavaDatatype` for every parameter specified in the `SQLParameterList`, but without `ParameterMode` or `SQLIdentifier`. Since `OUT` and `INOUT` parameters are always mapped to arrays of the specified type or class, you may omit brackets '[' in the `JavaSignature` denoting an array here and rely on the system to take care of them for you. However, you have to keep in mind the `ParameterMode` you specified in the in your `SQLSignature` and use arrays for `OUT` and `INOUT` parameters when writing your code in Java. Of course the `JavaSignature` is case sensitive. Table functions use a slightly different syntax:

```
"ClassFullName[JavaInSignature].methodname[JavaOutSignature]"
```

This syntax is based on the related Java construct

```
new ClassFullName(constructorParams).methodname(methodParams)
```

Note that the `SQLSignature` is significant for correct name overloading, not the `JavaSignature`. This means that you cannot create two routines using the same name but varying only in their `JavaSignature`, although this is possible when you code in Java. Here you can simply use another `SQLName` to evade that problem. Consult the 'JavaTypeMappings' subsection for a complete documentation of Java specifics and name overloading.

Privileges: The user must be owner of the external resource or userclass DBA. If `MethodFullName` references a resource that is stored outside the database, e.g. a jar file on the local file system that is referenced through the `CLASSPATH` option, userclass DBA is required.

3.6 ALTER PROCEDURE/FUNCTION statement

Function: Modifies `SQLName`, `SQLSignature`, `ReturnType`, `Options` and external reference of an external routine. All user privileges associated with this routine remain unchanged.

Syntax:

```
AlterProcedureStatement ::=
    ALTER PROCEDURE [{ENABLE|DISABLE}] SQLName SQLSignature
    [NAME NewSQLName NewSQLSignature]
    [SQLProperties]
    [DYNAMIC RESULT SETS integer]
    [ExternalBlock]
```

```
AlterFunctionStatement ::=
    ALTER FUNCTION [{ENABLE|DISABLE}] SQLName SQLSignature
    [NAME NewSQLName [NewSQLSignature]]
    [RETURNS SQLDataType]
```

```

[SQLProperties]
[CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT]
[ExternalBlock]

SQLSignature ::= ( [SQLParamter [, SQLParameter]...] )

SQLParameter ::= [ParameterMode] [SQLIdentifier] SQLDataType

ParameterMode ::= IN|OUT|INOUT|INSTANCE|INSTANCEOUT

SQLProperties ::=
  [[NOT] DETERMINISTIC]
  [{CHECK|IGNORE} EXCEPTION]
  [WITH[OUT] OWNER RIGHTS]
  [{NO SQL | CONTAINS SQL | READS SQL DATA | MODIFIES SQL DATA}]
  [COMMENT "comment"]

ExternalBlock ::= EXTERNAL
  [PARAMETER STYLE JAVA]
  [LANGUAGE JAVA]
  NAME 'MethodFullName[JavaSignature]'

```

Explanation:

Enable/Disable These options offer a quick possibility to globally enable or disable an external routine to all users without having to set and reset any user privileges.

Name The Name clause specifies a new name or signature for the function or procedure.

NewSQLName This will be the name of the routine formerly known as 'SQLName SQLSignature'.

NewSQLSignature optional, if stated this will be used as new SQLSignature for NewSQLName. NewSQLSignature may not be used without NewSQLName.

SQLName, SQLSignature, SQLProperties, and ExternalBlock change assignments of the corresponding routine properties. For details refer to 'CreateProcedureStatement'.

returns Specifies the new result data type of the function. See returns section of CreateFunctionStatement for more details.

Note: AlterProcedureStatement and AlterFunctionStatement are not subject of the ANSI standard.

Privileges: The user must be owner of the external resource or userclass DBA. If MethodFullName references a resource that is stored outside the database, e.g. a jar file on the local file system that is referenced through the CLASSPATH option, userclass DBA is required.

3.7 CALL statement

Function: The CallStatement calls a stored procedure.

Syntax:

```
CallStatement ::=
    CALL SQLName ( [Expression [, Expression]...] )
```

Explanation: The CallStatement executes a stored procedure. By definition a stored procedure has either no result, a tuple consisting of OUT parameters or it returns one or more result sets. In any case an exception is thrown if the execution of the code fails for some reason and a detailed description, including the trace stack of the JVM is printed, but no return code will be available. This might turn out to be a problem, if you call the procedure from an application or from another stored procedure or function. Through careful programming, however you can manage that your stored procedure catches any possible exceptions and passes an error code back by using an additional OUT or INOUT parameter type.

In the current Transbase release stored procedures that return ResultSets are not available. You may use a Table Function returning one Table instead.

Privileges: The current user must have USAGE privilege on the external resource or on the external routine. Note that one routine may call any other routine declared public and available in database or through the CLASSPATH option. USAGE privilege is only checked for the entry-point, whereas SQL privileges such as SELECT, INSERT, UPDATE are checked for every issued SQL statement. SQL privileges will be checked against the resources' owner privileges, if the routine is called WITH OWNER RIGHTS, otherwise the will be checked against the current users' privileges.

3.8 DROP PROCEDURE / FUNCTION statement

Function: Removes the SQLName and USAGE privileges of an external function or procedure from the catalog.

Syntax:

```
DropProcedureStatement ::=
    DROP PROCEDURE SQLName SQLSignature
```

```
DropFunctionStatement ::=
    DROP FUNCTION SQLName SQLSignature
```

Explanation: Drops the reference SQLName SQLSignature to an external routine and all privileges connected to that reference.

Privileges: The user must be owner of the external resource or of userclass DBA.

3.9 GRANT USAGE statement

Function: Extends the SQL grant statement for the USAGE privilege on external resources.

Syntax:

```
GrantUsageStatement ::=
    GRANT USAGE ON Resource
    TO UserList
    [WITH GRANT OPTION]

Resource ::=
    CLASSPATH
  | EXTERNAL ResourceName
  | {PROCEDURE | FUNCTION} SQLName SQLSignature

UserList ::=
    UserID [,UserID] ...

UserID ::=
    PUBLIC | UserName
```

Explanation: Usage privileges for resources loaded with a CREATE EXTERNAL statement can be granted on method or resource basis. Granting an EXTERNAL resource is equivalent to granting all published methods.

Usage privileges on methods from classes accessed via the CLASSPATH, i.e. for built-in methods and methods from user JARs can be globally granted by granting usage on the CLASSPATH resource. Additionally a user of userclass DBA may grant access to single methods of those resources and thereby form a public interface using GRANT USAGE ON PROCEDURE / FUNCTION statements.

Note: Initially a user has not the privilege to call any routine.

Note that the implementation of the GrantUsageStatement is optional in the ANSI NCITS 331.1-1999 standard.

Privileges: The current user must be owner of the external resource, of userclass DBA, or must have all specified privileges with the right to grant them.

3.10 REVOKE USAGE statement

Function: Extends the SQL revoke statement for the USAGE privilege on external resources.

Syntax:

```
RevokeUsageStatement ::=
    REVOKE USAGE ON Resource
    FROM UserList
```

```
Resource ::=
    CLASSPATH
    | EXTERNAL ResourceName
    | {PROCEDURE | FUNCTION} SQLName SQLSignature
```

```
UserList ::=
    UserID [,UserID] ...
```

```
UserID ::=
    PUBLIC | UserName
```

Explanation: If the current user is owner of the resource, 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 resource, 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.

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.

Note: The implementation of the RevokeUsageStatement is optional in the ANSI NCITS 331.1-1999 standard.

Privileges: The current user must be owner of the external resource, have userclass DBA, or must have all specified privileges with the right to grant them.

Chapter 4

Implementation Guide - Java

4.1 System Architecture

The introduction of Java STPs and UDFs into the Transbase system brings a fully enabled Java Virtual Machine into the address space of a Java enabled Transbase kernel. In the context of the Transbase architecture this means that every user that connects to the database gets along with the Transbase kernel a dedicated Virtual Machine assigned to his connection. While the Transbase kernel controls the Virtual Machine, a routine running inside the machine has no control over the kernel process or access to its data. A user has full access to the Virtual Machine's resources (within certain security policies enforced by a SecurityManager and a ClassLoader) and limited access to the database over a JDBC connection. After the user disconnects the Machine remains in that state, aside from some resources that are freed. Potentially another user may connect to this machine later. This approach has the advantage that the JVM does not have to be restarted every time a new user connection is opened. In addition already loaded classes are cached away and the Java HotSpot Technology has a better chance to identify heavily used routines and recompile them to native code for better performance. However, such a long life cycle may also bring drawbacks such as overly inflated virtual machines consuming memory resources and possible side effects from not cleaning up all claimed resources, such as ResultSets or static variables. But most drawbacks can be compensated by following a few programming conventions as described below and the operating system will take care of unused JVM pages by swapping them out of the main memory.

4.2 Transactions

SQL statements issued over the internal JDBC driver are executed within the transaction context from which the STP or UDF was originally called. In particular a STP or UDF may not begin, roll back or commit a transaction. An attempt to do this will always throw an SQLException. This means the transaction context before and after the execution of a STP is exactly the same, if no error occurred. This also means that AutoCommitMode is OFF by default for the internal JDBC Driver, whereas it is ON for any external `transbase.jdbc.Driver` connection. Additionally the consistency level for

TBX Type	SQL Type	Java default type	alternative type
TB_UNDEFTYPE		void	n.a.
TB_TINYINT	TINYINT	byte (non-nullable)	java.lang.Byte
TB_SMALLINT	SMALLINT	short (non-nullable)	java.lang.Short
TB_INTEGER/ TB_IKVALUE	INTEGER	integer (non-nullable)	java.lang.Integer
TB_BIGINT	BIGINT	long (non-nullable)	java.lang.Long
TB_NUMERIC	NUMERIC	java/math/ BigDecimal	n.a.
TB_FLOAT	FLOAT	float (non-nullable)	java.lang.Float
TB_DOUBLE/ TB_REAL	DOUBLE	double (non-nullable)	java.lang.Double
TB_CHAR/ TB_STRING	CHAR(*)	java.lang.String	n.a.
TB_VARCHAR	VARCHAR(*)	java.lang.String	n.a.
TB_DATETIME	DATETIME	java.sql. Timestamp	java.sql.Date/ java.sql.Time
TB_TIMESPAN	TIMESPAN	n.a.	n.a.
TB_BOOL	BOOL	boolean (non-nullable)	java.lang. Boolean
TB_BINCHAR	BINCHAR	byte[]	n.a.
TB_BLOB	BLOB	n.a.	n.a.
TB_BITSS	BITSS	boolean[]	n.a.
TB_BITSS2/ TB_UBVALUE	BITSS2	boolean[]	n.a.
TB_NULLTYP		n.a.	n.a.
TB_BLOBNAME		n.a.	n.a.
TB_FILEREF		n.a.	n.a.

Table 4.1: Java Type Mapping

the transaction may no be changed. Switching AutoCommitMode ON or changing the consistency level for an internal connection will throw an SQLException.

4.3 Java Type Mappings

The most basic topic on implementing Java methods in a database environment is to map SQL data types to Java types and classes. The following table offers an overview of all possible type mapping between SQL types and java types. "Non-nullable" means that Java passes parameters of this type by value, not by reference. Thus there is no representation for an SQL NULL. However you may use the alternative mapping if a representation of SQL NULLs is required in Java. Refer to the 'Null Values'-section in this chapter for more detailed information.

Usually default mapping between SQLTypes and JavaTypes, pretending the existence of a simple one-to-one relation, is performed if your functions do not specify that alternative type mapping is intended. To use the advanced feature of alternative mapping, you have to specify the full corresponding SQLSignature and JavaSignature, refer to 'CreateProcedureStatement' for details. To simplify matters you may also omit the leading 'java.lang.', 'java.math', and 'java.sql' portion of the alternative Java type string. Finally you can use custom type mapping to your own user-defined classes is Java with custom type mapping, as described in subsection 'User-defined Type Mapping'.

Note: For performance critical functionality it is strongly recommended to rely on default type mappings to scalar, non-nullable java types, because for any mapping to a Java Object type at least one instantiation (= one call to the objects constructor method) is necessary for every Object parameter and for every call to this method. Additionally, after a couple of calls to one of those java functions, asynchronous garbage collection start at some time inside the Virtual Machine, consuming even more CPU resources.

4.4 Parameter Modes

Additionally to the types STPs allow to use parameters in different modes, namely in IN, OUT, and INOUT mode. IN tells the database system that this is only an input parameter. Thus the value of this parameter is not relevant if the method returns, whereas an OUT parameter is used as result of a returning method. INOUT, of course, is the combination of these two modes.

In some cases, where you do not have influence on the method signature, e.g. a method from the Java runtime library `rt.jar` or from a jar from a third party vendor, then some additional functionality is required to cover all java specific routine calls. These extensions are Transbase specific and are not defined in the ANSI standard. These exceptions apply only to the first parameter of user-defined functions or stored procedures. For both, functions and procedures, this parameter may use `ParameterMode INSTANCE`, if you want to call a non-static instance method of the Java class mapped to the SQL parameter, and the parameter is in IN mode, e.g.

```
int java.lang.String.length()
```

maps to

```
FUNCTION strlen(INSTANCE CHAR(*)) RETURNS INTEGER.
```

Use `INSTANCEOUT` for procedures if you have an instance method and the first parameter is in INOUT mode.

Note that SQL data types are always mapped to the same java type, regardless of variable length or precision. Thus any length or precision specifications are irrelevant for type mapping. They will be adapted dynamically for Java IN parameters during run-time. However, if you use a user-defined function with OUT or INOUT parameters to insert or update data of a variable length SQL type, the function call should be accompanied by an explicit cast specifying the length of the destination field.

4.5 User-defined Type Mapping

Sometimes you may find it useful to map SQL data types not only to the Java types supplemented by the default or alternative type mapping but to map them directly to your own custom classes. To use this feature, your classes have to bring their own type conversion methods with them, i.e. a constructor method for the source SQL type, respectively its default mapped Java type and a conversion method to map your class back to the original SQL type when your method/function returns. Constructor methods are standard component of classes anyway and converter methods are commonly used throughout Java as means of non-trivial casts.

In order to enable the database system to find these methods, some name conventions are required for converter methods. Conversion to a primitive Java type `defaulttype` is done by a public instance method `defaulttypeValue()` taking no arguments. E.g. an object of class `myclass` converts to and from Java type `int`, which is the default mapping of SQL Integer, has to have a constructor method: `public myclass(int i)` and a converter method: `public int intValue()`.

Conversion to a class or array type `Defaultclass` has to provide a

```
public Defaultclass toDefaultclass()
```

method. E.g. an object of class `myclass` that converts to and from class `String` has to have a constructor method:

```
public myclass(String aString)
```

and a converter method:

```
public String toString().
```

Note: Only default and alternative type mappings are within the scope of the ANSI SQL99 standard.

4.6 Method Resolution

As name overloading is a fundamental concept in java, name overloading has also been made available for Transbase STPs and UDFs. Thus a java method is identified through its name and full signature. However, this concept collides with the concept of value adaptation known in SQL. Transbase tries to bring these two concepts closer together with a small set of rules for function name resolution:

1. The first resolution attempt uses function name, type (function or procedure) and parameter arity only. If resolution succeeds without type information then value adaptation is performed on the parameter values.
2. An explicit CAST on a parameter is never overridden.
3. If the first resolution attempt (rules 1 and 2) was ambiguous, a second attempt using the exact parameter types will commence.
4. If resolution is still ambiguous, then the method is not determinable.

This concept will allow you to write ad-hoc queries without worrying about the actual signature of a function too much, e.g. it is legal to call function `double sqrt(double)` with all sorts of scalar expressions as parameter. On the other hand, an application developer should always keep Rule 4 in mind. If at some point in time a second function `numeric sqrt(numeric)` should be added, e.g. for higher accuracy, then a call `sqrt(integer)` will fail because of Rule 4 and thus an application might stop working as new methods introducing name collisions are added to the database. A cautious application developer will avoid this problem by using Rules 2 and 3, i.e. by explicitly casting any parameter to the correct type or by converting to the exact signature on the client side. This should be no problem, since UDFs and STPs are typically developed especially for one particular schema where parameter types are defined by column types, and type safety is guaranteed.

4.7 Null Values

Usually function parameters in Java, such as objects or arrays, are passed to the function by reference. These types are:

```
java.lang.String, java.math.BigDecimal, byte[], java.sql.Timestamp,  
java.sql.Date, java.sql.Time, java.lang.Boolean, java.lang.Byte,  
java.lang.Short, java.lang.Integer, java.lang.Long, java.lang.Float,  
java.lang.Double
```

So these types may represent SQL NULL values by using a Java null reference. But aside from objects and arrays, Java supports a set of primitive data types, namely

`boolean, byte, short, int, long, float, and double.`

In Java these parameters are always passed by value. Thus, for these types there exists no representation of SQL NULL values. An attempt to call a UDF with a NULL value for such a scalar parameter will raise an error condition. When you code Java methods specifically for use in SQL, you will probably tend to specify Java parameter data types that are the nullable Java data types. You may, however, also want to use Java methods in SQL that were not coded for use in SQL, and that are more likely to specify Java parameter data types that are the scalar (non-nullable) Java data types.

You can call such functions in contexts where null values will occur by invoking them conditionally, e.g. in case expressions, e.g.:

```
SELECT CASE WHEN attr IS NOT NULL  
          THEN myFunc(attr)  
          ELSE NULL  
END FROM table;
```

You can also make such case expressions implicit, by specifying the returns null on null input option in the create function statement, see 'CreateProcedureStatement'. Both approaches are equivalent.

4.8 Build-In Java Functions

The availability of the full Java Runtime Environment, residing in the Java Runtime library `rt.jar` and shipped with each SDK and JRE installation, suggests to make functionality of this standard library available to the Transbase system. In particular the use of the various `java.lang` object instantiations for default and alternative type mapping compel the availability of public class (static) and instance (non-static) methods to the database. The following table is a quick reference for available built-in function.

Since these classes and their methods are subject of changes through the various SDK and JRE versions, the database administrator has full control over these methods, too, which means he may remove and alter existing methods and add new methods derived from classes on the `rt.jar`. See 'CreateProcedureStatement' for further information.

ExternalName	SQLName	Instance	SQLParameter List	SQL Return Type
java.lang.Double.isInfinite	isInfinite	NO	DOUBLE	BOOL
java.lang.Double.isNaN	isNaN	NO	DOUBLE	BOOL
java.lang.Integer.decode	decodeInt	YES	CHAR(*)	INTEGER
java.lang.Integer.parseInt	parseInt	NO	CHAR(*), INTEGER	INTEGER
java.lang.Integer.toBinaryString	toBinaryString	NO	CHAR(*), INTEGER	INTEGER
java.lang.Integer.toHexString	toHexString	NO	CHAR(*), INTEGER	INTEGER
java.lang.Integer.toOctalString	toOctalString	NO	CHAR(*), INTEGER	INTEGER
java.lang.Integer.toString	toString	NO	INTEGER, INTEGER	CHAR(*)
java.math.BigDecimal.abs	abs	YES	NUMERIC	NUMERIC
java.math.BigDecimal.max	getMax	YES	NUMERIC, NUMERIC	NUMERIC
java.math.BigDecimal.min	getMin	YES	NUMERIC, NUMERIC	NUMERIC
java.math.BigDecimal. movePointToLeft	movePointToLeft	YES	NUMERIC, INTEGER	NUMERIC
java.math.BigDecimal. movePointToRight	movePointToRight	YES	NUMERIC, INTEGER	NUMERIC
java.math.BigDecimal.scale	getScale	YES	NUMERIC	INTEGER
java.lang.Math.acos	acos	NO	DOUBLE	DOUBLE
java.lang.Math.asin	asin	NO	DOUBLE	DOUBLE
java.lang.Math.atan	atan	NO	DOUBLE	DOUBLE
java.lang.Math.atan2	atan2	NO	DOUBLE, DOUBLE	DOUBLE
java.lang.Math.ceil	ceil	NO	DOUBLE	DOUBLE
java.lang.Math.cos	cos	NO	DOUBLE	DOUBLE
java.lang.Math.exp	exp	NO	DOUBLE	DOUBLE
java.lang.Math.floor	floor	NO	DOUBLE	DOUBLE
java.lang.Math.log	log	NO	DOUBLE	DOUBLE
java.lang.Math.pow	pow	NO	DOUBLE, DOUBLE	DOUBLE
java.lang.Math.random	random	NO		DOUBLE
java.lang.Math rint	rint	NO	DOUBLE	DOUBLE
java.lang.Math.round	round	NO	FLOAT	INTEGER
java.lang.Math.sin	sin	NO	DOUBLE	DOUBLE
java.lang.Math.sqrt	sqrt	NO	DOUBLE	DOUBLE
java.lang.Math.tan	tan	NO	DOUBLE	DOUBLE
java.lang.Math.toDegrees	toDegrees	NO	DOUBLE	DOUBLE
java.lang.Math.toRadians	toRadians	NO	DOUBLE	DOUBLE
java.lang.String.compareTo	compareTo	YES	CHAR(*), CHAR(*)	INTEGER
java.lang.String.concat	concat	YES	CHAR(*), CHAR(*)	CHAR(*)
java.lang.String.endsWith	endsWith	YES	CHAR(*), CHAR(*)	BOOL
java.lang.String. equalsIgnoreCase	equalsIgnoreCase	YES	CHAR(*), CHAR(*)	BOOL
java.lang.String.indexOf	indexOf	YES	CHAR(*), CHAR(*)	INTEGER
java.lang.String.indexOf	indexOf	YES	CHAR(*), CHAR(*)	INTEGER
			INTEGER	
java.lang.String. lastIndexOf	lastIndexOf	YES	CHAR(*), CHAR(*)	INTEGER
java.lang.String. lastIndexOf	lastIndexOf	YES	CHAR(*), CHAR(*)	INTEGER
			INTEGER	
java.lang.String.length	getLength	YES	CHAR(*)	BOOL
java.lang.String.regionMatches	regionMatches	YES	CHAR(*), INTEGER	BOOL
			CHAR(*), INTEGER	
java.lang.String.regionMatches	regionMatches	YES	CHAR(*), BOOL	BOOL
			INTEGER, CHAR(*)	
			INTEGER, INTEGER	
java.lang.String.startsWith	startsWith	YES	CHAR(*), CHAR(*)	BOOL
java.lang.String.startsWith	startsWith	YES	CHAR(*), CHAR(*)	BOOL
			INTEGER	

Table 4.2: Built-In Java Functions

4.9 Security Manager

A Security Manager is installed by Transbase in the context where user-defined methods are executed. This Security Manager acts very much like Security Managers installed in Web Browsers executing untrusted code in Applets loaded over the network. The Security Manager denies all user-defined classes any access to sockets, threads, file system, system commands, interpreter commands, package access, system properties, networking and graphical output in windows. Additionally the installed ClassLoader denies loading of all java classes providing access to graphical output, sound, etc, i.e. the java.awt classes. This is done to enforce system security and to prevent that system resources are wasted for classes that are apparently of no use in the Transbase server environment. However, it is possible to lower the security policies of the by configuring the installed Security Manager. Refer to the 'AlterExternalOption' section for more details.

4.10 Class Loading

This subsection will give you insight in the mechanism how the JVM finds and retrieves classes. Unlike a normal JVM, where classes are read mostly from the local file system by parsing the CLASSPATH environment variable, the TBJRE has three main sources for its classes, checked consecutively by a simple fall-back mechanism. The first source is the database system itself, where classes can be added with the CreateExternalStatement, and secondly, if not all required classes can be found there, the CLASSPATH option (see 'AlterExternalOption') is consulted. This option is very similar to the CLASSPATH variable mentioned before, but the CLASSPATH variable itself will never be used. This guarantees that only the database administrator can set this option to trusted directories and files. These files can then be protected by carefully setting the privileges on the file system, e.g. to read only. This procedure should prevent replacement of classes used by the database system without knowledge of the database administrator. The third source for still unresolved class names are the jar files in the tbjre subdirectory of your Transbase installation or in any other subdirectory, if the JDK option (see 'AlterExternalOption') is defined. These jars, too, should be subject to a careful setting of file system privileges. If a class cannot be found in any of these locations, then a ClassNotFoundException will be thrown by the ClassLoader. Classname resolution during the compilation process works exactly to same way.

Usually you will not have to worry about this mechanism very much if you do not use duplicate class names. But if you have a class myClass.class available in your database and another myClass.class in a jar file, in your CLASSPATH option, then the class from the jar will never be loaded.

4.11 Pattern for a Stored Procedure

In Transbase, stored procedures usually are implemented according to the a simple pattern. A stored procedure is always a public, static method.

The return type of a stored procedure is always void.

For OUT and INOUT parameters the corresponding parameter has to be of an array type of the mapped java type, e.g. an INOUT parameter of SQL type INTEGER is either an `int[]` using default mapping or `java.lang.Integer[]` using alternative mapping. This array will only contain one element. In case of an INOUT parameter, it holds the input parameter, in case of an out parameter it will be initialized with null. The stored procedure is then responsible to supply a new valid value for output or a null value.

If the stored procedure uses SQL, then a JDBC connection to the current database context can be acquired by opening a JDBC default connection, i.e.

```
Class.forName("transbase.jdbc.Driver");
Connection conn =
    DriverManager.getConnection("jdbc:default:connection");
```

So the body of a stored procedure executing an update query and returning a success or error indicator in a supplied boolean OUT-parameter could look like this

Example:

```
#import java.sql.*

public class myClass1 {

    public static void myProc1(String updatequery, boolean[] success)
        throws SQLException, ClassNotFoundException
    {
        Class.forName("transbase.jdbc.Driver");
        Connection conn =
            DriverManager.getConnection("jdbc:default:connection");

        Statement stmt;
        ResultSet rs;
        try {
            rs=stmt.executeUpdate(updatequery);
        } catch (SQLException e) {success[0]=false; return;}
        success[0]=true;
        return;
    }
}
```

4.12 Parameter Values

OUT parameters are never instantiated before a method is called. Any OUT parameter will be passed as null reference to a stored procedure, thus you always have to create an instance before you assign a value, otherwise your procedure will throw a `NullPointerException`. This is also true for array types such as `byte[]` or `boolean[]` as shown in the following

Example:

```
public static void bitss_procedure(boolean[] [] a)
{
    a[0]=new byte[1]; //instantiation required
    a[0][0]=0xFF;
}
```

whereas

Example:

```
public static void string_procedure(String[] a)
{
    a[0]="Hello World from Java!";
}
```

is legal code, because here an implicit instantiation of `String a[0]` takes place.

INOUT parameters of fixed length, precision, or range, such as CHAR, BINCHAR, BITSS, NUMERIC, DATETIME, and TIMESpan must not return a value of greater length, precision, or range as OUT value, than the corresponding IN value yields. This restriction shall provide for type safety and to ensure that the host variable storing the IN-value is also capable of storing the OUT-value. If you know that there is more space required for the return value, then you can explicitly cast your INOUT parameter to provide a higher or the highest possible space for a return value.

Example:

```
CREATE string_procedure(INOUT aString CHAR(*)) EXTERNAL NAME
    "myClass.stringProc";
```

```
CALL string_procedure('Hello World!');
```

may return any legal CHAR(12) value in aString.

```
CALL string_procedure('Hello World!' CAST CHAR(200));
```

may return any CHAR(200) value.

```
CALL string_procedure('Hello World!' CAST CHAR(*));
```

may return any CHAR(*) value up to the max. possible length (MAXSTRINGSIZE).

OUT parameters always provide enough space for the max. possible OUT-value of that type.

4.13 Pattern for a User-Defined Value Function

Similar to a stored procedure, there is also an implementation pattern for Value UDFs. The return type of an user-defined value function is generally one of the default java types as defined in the default mapping (TypeMapping). Usually a UDF is a java routine declared static. Since UDFs do not allow for INOUT and OUT parameters, all types allowed by Type Mapping may be used, but arrays are not allowed. A default JDBC connection can be acquired by

```
Class.forName("transbase.jdbc.Driver");
Connection conn =
    DriverManager.getConnection("jdbc:default:connection");
```

Example:

```
#import java.sql.*

public class myClass2
{

    public static boolean myProc2(String updatequery)
    {
        try
        {
            Class.forName("transbase.jdbc.Driver");
            Connection conn =
                DriverManager.getConnection("jdbc:default:connection");
        } catch (ClassNotFoundException e) {return false;}

        Statement stmt;
        try
        {
            stmt.executeUpdate(updatequery);
        } catch (SQLException e) {return false;}
        return true;
    }
}
```

Additionally to static methods the Transbase extension for Java offers the possibility to use instance methods. However, instance methods do not perform as well as static methods do, because of the additional overhead for creating and destroying the instance. Thus it is recommended to use instance methods with care and only if it is inevitable, if you do not have influence on the signature, e.g. for methods from `rt.jar` or any other third party jar. With instance methods the first SQL parameter of a value function has to be in parameter mode `INSTANCE` or `INSTANCEOUT`. This parameter will then be used by the Transbase system to construct the java object by calling the parameter classes' standard constructor method. This parameter will then of course not appear in the java methods signature, but it is available through its `this` object self reference, respectively its value

by calling `this.value()`. The return value of the function is set that way too, in case of an `INSTANCEOUT` parameter. An example is provided in 'CreateFunctionStatement'.

Of course arbitrary instance methods can be called from a static context by writing a wrapper method, see 'JavaWrappers'.

4.14 Pattern for a User-Defined Table Function

Transbase offers the possibility to write user-defined functions returning tables. These so-called table functions may be used anywhere where conventional tables may be used, i.e. in the SQL statements `FROM` clause. Table functions use a set of `IN` parameters to construct an object instance that then will return the data rows by successive calls to one of this object's instance methods. To generate this behavior with Java object various conventions are necessary:

The Java class used as table function must have

- one public constructor method taking all `IN` parameters of this method. This method is only invoked once for a pass through a table.
- one public non-static method returning boolean and having the `OUT`-signature defined by the output-table and no `IN`-parameters. The returned value is `true` if the `OUT`-parameters hold valid results or `false` if no more rows are delivered.
- it has be guaranteed that the method returns `false` after a finite number of evaluations.

The programmer may also supply a `public void finalizer()` method for cleaning up before the instance is destroyed. Keep in mind that finalizers always run with `NO SQL` access indication, and therefore no use of `SQL` is allowed here.

Note: If the table function takes part in a nested loop join, then constructor and finalizer will be called several times, once for each loop. Due to the asynchronous nature of Java's garbage collection, that calls the finalizer method immediately before collecting the object, it is recommended to relocate the cleaning up routines from the finalizer to the instance method, just before it returns `false` for the first time. So it is always guaranteed that used `Connections`, `Statements`, `ResultSets`, or other resources are closed `BEFORE` the next loop is invocated. If you join two table function, keep in mind that you have to carefully balance opened resources, since both functions concurrently use limited resources such as cursors for open queries (i.e. `ResultSets`).

Note: Name resolution for table functions uses only the function name and the `IN` parameter list.

The following example illustrates this by implementing a hierarchical depth first search in a table defined by

```
CREATE TABLE hierarchy (id INTEGER NOT NULL, parent INTEGER)
```

and populated by

```
INSERT INTO hierarchy TABLE ((1,NULL),(2,1),(3,2),(4,2),(5,4),
(6,4),(7,1),(8,7),(9,1),(10,9),(11,10),(12,9));
```

and returning records of root and all sons in depth first sort order. The function itself is published to the SQL system by this DDL statement (cp. 'CreateFunctionStatement')

```
CREATE FUNCTION dfs(id INTEGER)
  RETURNS TABLE (id INTEGER, parent INTEGER)
  READS SQL DATA
  EXTERNAL NAME "DFS(Integer).dfs(int [],Integer [])";
```

Example:

```
import java.sql.*;
import transbase.tbx.TBConst;

public class DFS {
  ResultSet[] rs;
  PreparedStatement[] ps;
  Connection con;
  int usedRS;

  public DFS() throws SQLException, ClassNotFoundException {
    rs=new ResultSet[TBConst.MAX_QUERY_CNT];
    ps=new PreparedStatement[TBConst.MAX_QUERY_CNT];
    usedRS=0;
    Class.forName("transbase.jdbc.Driver");
    con = DriverManager.getConnection("jdbc:default:connection");

    ps[usedRS] = con.prepareStatement("SELECT id,parent FROM" +
      " hierarchy WHERE parent IS NULL");
    rs[usedRS]=ps[usedRS].executeQuery();
  }

  public DFS(Integer root) throws SQLException, ClassNotFoundException {
    rs=new ResultSet[TBConst.MAX_QUERY_CNT];
    ps=new PreparedStatement[TBConst.MAX_QUERY_CNT];
    usedRS=0;
    Class.forName("transbase.jdbc.Driver");
    con = DriverManager.getConnection("jdbc:default:connection");

    ps[usedRS] = con.prepareStatement("SELECT id,parent FROM" +
      " hierarchy WHERE id=?");
    ps[usedRS].setInt(1,root.intValue());
    rs[usedRS]=ps[usedRS].executeQuery();
  }

  public boolean dfs(int[] id, Integer[] parent) throws SQLException {
    if(rs[usedRS].next())
```

```

{ id[0]=rs[usedRS].getInt(1);

  if(rs[usedRS].getInt(2)==0 && rs[usedRS].wasNull())
    parent[0]=null;
  else
    parent[0]=new Integer(rs[usedRS].getInt(2));

  ps[++usedRS] = con.prepareStatement("SELECT id,parent FROM" +
    " hierarchy WHERE parent=?");
  ps[usedRS].setInt(1,rs[usedRS-1].getInt(1));
  rs[usedRS]=ps[usedRS].executeQuery();
  return true;
}
else
{ if(usedRS==0)
  { if(!con.isClosed())
    con.close();
    return false;
  }
  else
  { ps[usedRS--].close();
    return dfs(id,parent);
  }
}
}
}
}

```

Finally the function can be called by typing

```
SELECT id, parent FROM FUNCTION dfs(2);
```

4.15 Pattern for Generic Table Functions

In addition to table functions where the structure of a returned table is defined in the CREATE FUNCTION statement, generic table functions can return arbitrary tables. However, this flexibility comes at the expense of some performance.

In Generic Table Functions a Java object instance is created using a set of IN parameters. The structure of the output table is determined dynamically. A Java class used as Generic Table Function must extend the abstract class `transbase.generic.Reader`. This means it must implement the following interface, which are also called in this order:

- First the `public constructor` method is invoked, taking all IN parameters of this method. This method is invoked twice for one pass through a table. The first call comes from the DBMS SQL compiler in order to determine the table structure.
- Then the method `public int getColumnCount()` is called by the DBMS SQL compiler to retrieve the number of columns. Now, for every column the method `public int getColumnType(int column)` is invoked to get column type codes as

defined in `java.sql.Types`. Finally the name of each column is queried by calling `public String getColumnName(int column)`. These three methods are to work exactly as their counterparts defined in the `java.sql.ResultSetMetaData` interface.

- Now `public void close()` is called to conclude the SQL compiler run. Here the class should release all resources, as there is no guarantee that the query is executed later (e.g. in case of an abandoned `PreparedStatement`).
- If the query is executed, the `public constructor` is called a second time with identical input parameters as the first time to create a new instance.
- Then `public boolean next(Object[] [] arr)` is called once for every returned tuple. It's purpose is to fill it's OUT parameter `arr` with the next output tuple. The function returns `true` if the OUT parameters hold valid results or `false` if no more rows are delivered. Note that `arr` is an array with as many entries as appointed by the `getColumnCount` call. Each entry holds another array of one single element. Each of these arrays is of the Java Object type that corresponds best to the type code returned by `getColumnType`, e.g. `Integer[]` for `java.sql.Types.INTEGER`. Also note that scalar Java types are always 'boxed' in their corresponding class objects, i.e. `Integer[]` not `int[]`, so NULL values can be represented.
- A final call to the `public void close()` method ends the flow of control.

The following example illustrates the use of Generic Table Functions on an `ArrayReader` class that returns data from several private arrays. The `ArrayReader` has one boolean IN parameter `showall` which arranges that only the first or both columns are returned.

Example:

```
import java.sql.SQLException;

public class ArrayReader extends transbase.generic.Reader {
    private static String[] text_arr = {"a","b","c"};
    private static int[] numb_arr = {1,2,3};
    private boolean showall;
    private int rowpos=0;

    public ArrayReader(boolean showall) {
        this.showall=showall;
    }

    public int getColumnCount() throws SQLException {
        if(showall)
            return 2;
        else
            return 1;
    }

    public int getColumnType(int col) throws SQLException {
        if(col==1)
```

```

        return java.sql.Types.CHAR;
    else if(showall && col==2)
        return java.sql.Types.INTEGER;
    else throw new SQLException("Column index " + col + " out of bounds");
}

public String getColumnName(int col) throws SQLException {
    if(col==1)
        return "text";
    else if(showall && col==2)
        return "numb";
    else throw new SQLException("Column index " + col + " out of bounds");
}

public void close() throws SQLException { }

public boolean next(Object[][] o) throws SQLException {
    if(rowpos==text_arr.length) return false;
    o[0][0]=text_arr[rowpos];
    if(showall) o[1][0]=new Integer(numb_arr[rowpos]);
    rowpos++;
    return true;
}
}
}

```

The function is published by using the following DDL statement (cp. CreateFunction-Statement)

```

CREATE FUNCTION ReadArray(showall bool)
    RETURNS TABLE
    EXTERNAL NAME "ArrayReader.next"

```

and invoked with

```

SELECT * FROM FUNCTION ReadArray(FALSE)

```

The next example is a SimpleJDBCReader that is capable of querying any JDBC data source using an arbitrary SQL query and makes the data available to the Transbase DBMS. Note that the corresponding JDBC Driver has to be registered before execution. Refer to the `java.sql.DriverManager` documentation for an overview on how to register various drivers. Additionally the used driver has to be available in the system's CLASSPATH option (see 'AlterExternalOption').

Note: An more sophisticated version of this handy Generic Table Function is available as built-in `JDBCReader` in every Transbase database where Generic Table Functions are supported.

Example:

```
import java.sql.*;

public class SimpleJDBCReader extends transbase.generic.Reader {
    static {
        try {
            Class.forName("transbase.jdbc.Driver");
        } catch (Exception e) {}
    }

    private Connection connection;
    private PreparedStatement pstatement;
    private ResultSet result = null;

    public SimpleJDBCReader(String url, String user, String pwd, String sql)
        throws SQLException {
        connection = DriverManager.getConnection(url, user, pwd);
        pstatement = connection.prepareStatement(sql);
    }

    public int getColumnType(int column) throws SQLException {
        return pstatement.getMetaData().getColumnType(column);
    }

    public int getColumnCount() throws SQLException {
        return pstatement.getMetaData().getColumnCount();
    }

    public String getColumnName(int column) throws SQLException {
        return pstatement.getMetaData().getColumnName(column);
    }

    public final void close() throws SQLException {
        connection.close();
    }

    public final boolean next(Object[][] o) throws SQLException {
        if (result == null)
            result = pstatement.executeQuery();

        if (result.next()) {
            for (int i = o.length - 1; i >= 0; i--)
                o[i][0] = result.getObject(i + 1);
            return true;
        }
        else
            return false;
    }
}
```

The built-in JDBCReader is called with:

```
SELECT * FROM FUNCTION JDBCReader(  
    'jdbc:transbase://hostname:2024/dbname',  
    'user','passwd','select * from mytable')
```

The following steps show the necessary configuration for using third-party JDBC drivers to be used by the JDBCReader. 1. Add the third-party driver to the JRE's CLASSPATH. Make sure that the file is accessible for the Transbase service. Note that the CLASSPATH points to the JAR file, not only to the directory:

```
ALTER EXTERNAL OPTION CLASSPATH "/usr/lib/java/acmesql.jar"
```

2. Make sure the driver registers with the system's JDBC driver manager by providing the driver's fully qualified class name

```
ALTER EXTERNAL OPTION JAVA "-Djdbc.drivers=com.acme.jdbc.Driver"
```

3. Allow the driver to access network resources. Possibly other permissions are also required:

```
ALTER EXTERNAL OPTION JAVAPERMISSIONS "java.net.NetPermission"
```

Now the third-party database may be accessed by calling the JDBCReader using the appropriate connection URL.

4.16 Static Variables

Although java static methods can be contained in Java classes that have static variables, and in Java static methods can do both, reference and set static variables, it is strongly recommended not to use static variables at all, since that may lead to unpredictable results in a certain combination of circumstances due to the Transbase system architecture. If you intend that users share global variables concurrently across connections, then you should store this variables in a table rather than declaring them static. This behavior is consistent with the ANSI standard.

4.17 Java Wrappers

There are cases when you will find it impossible to use a certain Java method as a STP or UDF because their signature is not mappable to an SQL type signature. A common example for such methods are `main(String[] args)` methods. In such cases you can still write wrapper methods that have a mappable signature themselves and simple call the method you wanted to call in first place. An example for a wrapper for a main method would be:

Example:

```
public static void callmain(String a, String b, String c) {
    String args[] = {a,b,c};
    return someClass.main(args);
}
```

4.18 Choosing your JVM

The database administrator can strongly influence the characteristics of a Transbase system running a virtual machine by choosing from the available JVM implementations and versions. First of all the administrator can choose from any JVM implementing the Java Native Interface version 1.2 or higher. Thus JVM prior to 1.2.0 will not work with Transbase. As a general rule of thumb, one can say that a higher version of the JVM trades higher memory consumption for faster execution, where higher memory consumption means more allocated disk space (for the JVM and accompanying runtime-libraries, i.e. rt.jar) as well as a bigger memory footprint. In addition one may freely choose between the J2SE (Java 2 Platform Standard Editions) or the J2RE (Java 2 Runtime Environment), where the Runtime-Environment will not allow for any java code compilation, neither in Transbase, nor with the java compiler (javac) on the command line, which may be a desired effect if the database as a whole is to be distributed.

However, it is recommended not to use versions prior to 1.3.0 since this version introduces the Java HotSpot Server that allows not only for running Java Code in interpreted mode but also in the so-called mixed mode. This means that heavily used Java routines will be compiled to platform dependant native code during runtime, and thus significantly speeding up execution.

Although constant efforts are being made to get most out of the newest Java releases, there are also indications that changes in new Java releases may influence reliability and functionality of JVM and Transbase interactions.

4.19 Remote Method Debugging

It is often desirable to test and debug new STPs and UDFs directly in an database environment, namely remote method debugging. In order to have the JVM in Transbase running in debug mode, the JVM has to be started that way. Unfortunately the Java Native Interface used by Transbase to communicate with the JVM is still incomplete, and shutting down and restarting a JVM or starting a second instance of the JVM is not possible even with the newest JDK release. Thus the way to remote method debugging is a little bumpy. You have to tell Transbase that you want to debug methods before the JVM is started. This is done with a special call to a stored procedure:

```
CALL Debugger();
```

This call will always return an error code, either 16718 (JVM_DEBUG_ON) if you successfully switched to debug mode or 16719 (JVM_DEBUG_OFF) if the JVM is already

running and switching to debugging is impossible or if there is something wrong with your debugging configuration (see 'AlterExternalOption'). If you successfully switched to debugging mode, then the server process associated to your connection will go into suspend mode without any further notice, as soon as you call the first Java based external method. Now you can attach to the server process using an implementation of jdb, e.g. any JDE providing remote debugging capabilities. As soon as this debugging connection is complete, the Transbase server will continue executing Java code. Every time it comes across a breakpoint, it will again suspend itself and pass control over to the Java debugger and you can step through your code.

Note: The workaround using 'Call Debugger();' is to be considered temporary and will be removed as soon as a new JVM release provides the required functionality.

4.20 Evaluating Debug Output

Usually, when Transbase runs in server mode, no output is written, i.e. to stdout or stderr. Thus those ports are blocked for a JVM running in the same address space, too. As a consequence calls to `System.out` or `System.err` in Java have no effect, and any other output written by the JVM, e.g. output generated by the '-verbose' or '-Xrunhprof' options of the JVM, is suppressed. However it is often desirable to output some information for debugging. This can be achieved by setting the corresponding database "Redirect JVM output" (-rjo) parameter with the tadmin tool.

If redirection is on, then all output generated by the JVM during one database connection is written to one temporary file in the databases "Scratch" directory. The file names consist of the four characters of the username that generated that output and are consecutively numbered. That files are updated each time the user calls a java method. Note that all output is buffered before it is written into the file, so a complete protocol is usually not available before the method returns. However, the buffer is flushed whenever a New-Line-character ('\n') is found in the stream, so you can force flushing by using `System.out.println()` or `System.err.println()`.

If redirection is off, all output will be discarded.

Chapter 5

Implementation Guide - Native Libraries

5.1 Limitations

In contrast to the framework for STPs from Java, there are a number limitations to the framework for functions from native libraries at the moment.

- SQL queries from the libraries to the local Transbase instance are not supported.
- Stored Procedures are not supported.
- Table Functions are not fully supported. Only generic table functions are available.

5.2 Compiling and Linking

As already stated in the 'Getting Started Section' there are a number of requirements to consider:

- must use `_cdecl` call convention
- make sure your functions are properly exported
- compile as resource library, i.e. without entry point

A list of examples on how to compile and link on various platforms can also be found there.

5.3 Pattern for Generic Table Functions

Just like Generic Table Functions written in Java have to implement a predefined interface, their native library counterparts have to export a related interface. Routines belonging to one set of interface functions all have the same user-defined `prefix`.

```

int <prefix>Open(..., int taid, int cons, void** cctx, void** sctx);
int <prefix>Next(void* sctx, void*** result, int* eod);

int <prefix>GetColumnCount(void* sctx);
char* <prefix>GetColumnName(void* sctx, int column);
int <prefix>GetColumnType(void* sctx, int column);
int <prefix>GetLobPart(void* cctx, Blob *b, Uint4 len, Uint4 offset,
    char* buf, Uint4* read);
void <prefix>Close(void* sctx);
void <prefix>Disconnect(void* cctx);
void <prefix>GetLastError(void* ctx, char* errmsg, size_t size,
    int* errno);

```

The following represents an optional interface extension if additional initialization and/or cleanup routines are required.

```

void TbExtLoad(void);
void TbExtUnload(void);

```

All calls to a table instance implemented as Generic Table Function refer to the same `Context` structs/objects (`sctx` for statement context and `cctx` for connection context). These are allocated once in the `Open()` routine using a freely definable set of input parameters (e.g. connection string, user name, password). Additional parameters for transaction id `taid` and consistency `cons` are compulsory. As results, query and connection contexts are passed to the DBMS which retains them and passes them as first parameter to every subsequent call to routines referring to this table. Finally the statement context is freed in a call to `Close()` and the connection context is closed by calling `Disconnect`. The structure of the output table is determined by the DBMS dynamically via routines interface, which are called in this order:

- First the `Open()` routine is invoked, taking all IN parameters of this method. This method is invoked twice for one pass through a table. The first call comes from the DBMS SQL compiler in order to determine the table structure.
- Then the method `GetColumnCount()` is called by the DBMS SQL compiler to retrieve the number of columns. Now, for every column the method `GetColumnType()` is invoked to get column type codes as defined in `tbx.h`. Finally the name of each column is queried by calling `GetColumnName()`.
- Now `Close()` is called to conclude the SQL compiler run. Here the Table Function should release all resources, including the statement context structure, as there is no guarantee that the query is executed later (e.g. in case of an abandoned stored query).
- If the query is executed, `Open()` is called a second time with identical input parameters as the first time to create a new instance of the statement context. The returned connection context should be identical to the one returned in first call. This resembles connection pooling and guarantees transaction isolation. In particular, the table function should always reuse connections were possible, i.e. when input parameters (including `taid` and `cons`) permit that.

- Then the iterator function `Next()` is called once for every returned tuple. Its purpose is to fill its OUT parameter `result` with the next output tuple. This should then point to an array of `GetColumnCount()` pointers to Transbase data types as defined in `tbx.h`. An SQL NULL is represented by setting this pointer to NULL. If one of the columns returns Blob data, then the corresponding pointer points to a Blob structure (as defined in `tbx.h`). Make sure that the `size` component is set correctly and set the `btype` (blob type) component to a unique non-zero value for this blob type. Do not overwrite the `connid` and `btype` components. The remainder of the structure may be used freely to identify this particular Blob. It is encouraged to store a pointer to whatever structure is required to identify a Blob at `Blob->blobadr`.

`Next()` sets the output parameter `eod` (end-of-data) to `true` if the OUT parameters hold valid results or `false` if no more rows are delivered. It has be guaranteed that the method returns `false` after a finite number of evaluations.

- If one of the columns returned Blob data, any number of `GetLobPart()` calls might follow to retrieve the contents of the Blobs identified via the input parameter `b` of type `Blob*`. This call has to copy `len` bytes of the contents of the Blob beginning at position `offset` to the provided buffer `buf` of `len` bytes. `read` returns the number of bytes transferred.
- The routines `Open()`, `Next()`, and `GetLobPart()` return error codes. All other functions return results. A return code different from zero means an error occurred during the last call. Then the DBMS used `GetLastError()` to retrieve a human readable error description from connection or statement context, dependent upon where the error occurred.
- A call to the `Close()` method ends the flow of control for this statement and releases all resources associated to the statement context and the statement context itself. Note that after this point calls to `GetLobPart()` must still be possible.
- As the local transaction ends a final call to `Disconnect` releases all remaining resources, closes the connection and release the connection context itself. In particular resources connected to Blob data are to be released at this point.

Chapter 6

Database Client APIs

Since user-defined functions are always part of a select, insert, update, or delete statement, there is no separate call interface required for that SQL extension. Stored procedures however represent a new class of SQL statements and extensions to the APIs become necessary.

6.1 JDBC

JDBC provides the `CallableStatement` class for calling stored procedures. For database compatibility call statements may use the JDBC escape syntax. Two different notations are possible:

```
{ call procedure_name[?, ...] }
```

This statement will be passed unmodified to the database, apart from the removal of the curly braces. If the optional parameter list is missing then `()` will be appended.

```
{ ? = call procedure_name[?, ...] }
```

Since Transbase Stored Procedures never return a value, this statement will be modified in such way that the result parameter (ordinal number: 1) will be moved to the parameter list as first parameter. This change will create a statement corresponding to the first syntax while preserving the original parameter numbering.

The following example illustrates how to call a stored procedure from a JDBC client. This procedure accepts three parameters in parameter modes IN, INOUT, OUT. The parameter types are specified by literals. Note that the value of the third (OUT) parameter has no influence on the result, only the type specification will be used for name resolution.

Example:

```
Class.forName("transbase.jdbc.Driver");  
String dburl = "jdbc:transbase://MYHOST:4444/DB";  
String uname = "user";
```

```
String pw = "passwd";

Connection conn = DriverManager.getConnection(url, uname, pw);

// call a Procedure with parameter modes: IN, INOUT, OUT
java.sql.CallableStatement cstmt;
cstmt = conn.prepareCall("{call proc('some string',2,3.3)}");
cstmt.executeQuery();
int i = cstmt.getInt(1);
BigDecimal bd = cstmt.getBigDecimal(2);
```

The same statement using parameters:

Example:

```
// call a Procedure with parameter modes: IN, INOUT, OUT
java.sql.CallableStatement cstmt;
cstmt = conn.prepareCall("{call proc(?,?,?)}");
cstmt.setString(1, "some string");
cstmt.setInt(2, 2);
cstmt.registerOutParameter(3, java.sql.Types.DECIMAL);

cstmt.executeQuery();
int i = stmt.getInt(1);
BigDecimal bd = stmt.getBigDecimal(2);
```

Note: All OUT parameters have to be registered before the query is executed. Otherwise an SQLException will be thrown.

6.2 TBX

On the TBX interface a CALL statement to stored procedures that does not use dynamic parameters, especially no OUT or INOUT parameters, is executed by a RUN statement. The following two examples show query executions where all parameters are in IN mode, i.e. no result is returned, and a query where OUT values are returned as a result.

Example:

```
Result result;
char *statement = "call proc(1,2,3)"; /* all modes are IN */
TbxRun (dbid, taid, statement, & qd, NULL);

char *statement = "call proc(1,2,3)"; /* modes are OUT, INOUT,
or INSTANCEOUT */
TbxRun (dbid, taid, statement, & qd, &result);
```

Otherwise, if parameters are required, stored procedures behave very much like stored queries, where parameter type and value have to be specified for all input parameters, namely IN and INOUT parameters, whereas only type information is required for mere OUT parameters. The following example illustrates this behavior.

Example:

```

Error e;
Integer in[2];
Integer out[2];
Id stmtid, dbid,taid;
Parameters par;
Param param[3];
Query\_descr qd;
Result result;

char *statement = "call proc(?,?,?)"; /* modes are in, inout, out */

/* connect, login and begin transaction omitted */

if(e=TbxStore(dbid,statement,&stmtid,&qd))
goto err0;

par.param\_no = 3;
param[0].type=TB\_INTEGER; /* now supply type information */
param[1].type=TB\_INTEGER; /* for all parameters */
param[2].type=TB\_INTEGER;

in[0]=1;
in[1]=2;
param[0].value=&in[0];
param[1].value=NULL; /* this is an out parameter,
so no input value has to be supplied */
param[2].value=&in[1];
par.param=&param;

if(e=TbxRunStored(dbid,taid,stmtid,&par,&qd,&result))
goto err0;

out[0]=*(Integer*) ptoattornull(result.\_var.tuple,0);
out[1]=*(Integer*) ptoattornull(result.\_var.tuple,1);

```

Note that IN parameters are not included in the result, thus if a procedure call has parameters in IN and OUT, one has to be aware of the fact that the index of a parameter in the result set may differ from its index as a parameter, e.g. the OUT parameter in the upper example has parameter index 1 but result tuple index 0.

Chapter 7

Troubleshooting

7.1 The retrieval or update of an external resource, method, or option failed, the system catalog might be corrupted (Error 16600)

When this error occurs, then the system catalog (sysexternal, sysexternalmethod) does not have the expected format. This error usually occurs if you try to manipulate procedures directly with update or insert statements on the system catalog instead of using the provided DDL. If this error occurs, it is recommended to drop the concerned routine/resource with a DROP statement and then recreate it with the appropriate CREATE statement.

7.2 Internal JDBC Driver not correctly installed (Error 16700)

This Error occurs if Transbase is not able to find or load the internal JDBC driver `tbjdbc.jar`. Usually this driver is located in your `$TRANSBASE/tbjre/lib` directory and will be appended to your `EXTERNAL OPTION CLASSPATH` setting automatically, even if you have installed a custom JRE. Check if the file is available and intact.

7.3 Cannot create Java VM (Errors 16705 and 16720)

Error 16705: Cannot create Java VM. For some reason Transbase was not able to load the Virtual Machine's dynamic libraries. If you have installed a custom JDK, check if you `EXTERNAL OPTION JDK` points to you JDK's base directory. If this does not work, try switching to another version of JDK or back to the JRE shipped with Transbase by typing

```
ALTER EXTERNAL OPTION JDK "";
```

Error 16720: Cannot create Java VM because LD_LIBRARY_PATH does not comprise 'path-to-library' or is not set at all. If you are running Transbase on any other platform than Windows, it is utmost important that LD_LIBRARY_PATH points to the directories where the libraries required to start the JVM are located. By name these are the files libjvm.so and libjava.so and typically they are typically contained in \$JDK/jre/lib/i386 and \$JDK/jre/lib/i386/server for Linux.

This error occurs if LD_LIBRARY_PATH is not set at all or if it does not contain the paths to the libraries libjvm.so and libjava.so in the JDK you chose by setting the EXTERNAL OPTION JDK.

7.4 Transbase crashes on first access to a UDF or STP

As Transbase has to give control to the JVM while starting it up, in this critical phase crashes can occur, if the JVM is not configured properly. Try

```
ALTER EXTERNAL OPTION JAVA "";
```

to start the JVM without any options and check if LD_LIBRARY_PATH (for non-Windows platforms) is set correctly (for more details on LD_LIBRARY_PATH settings, see previous section on Error 16720).

7.5 System hangs or crashes during heavy UDF or STP load on Unix / Unexpected Signal : 11

It was observed that on rare occasions (heavy UDF or STP load in Unix-based environments) the system ceases operations due to an error in the JVM implementation. Usually a diagnostic file named `hs_err_pidXXXX.log` is written into the database home directory reporting an unexpected exception "Unexpected Signal : 11". In these cases it proved useful to set LD_LIBRARY_PATH to \$JDK/jre/lib/i386 and \$JDK/jre/lib/i386/client, i.e. not using the server but the client implementation of the VM or try a newer VM release (see previous section on Error 16720 for details on LD_LIBRARY_PATH settings).

7.6 Exceptions in classes from the Transbase runtime package

If you should experience exceptions in classes of your Transbase runtime environment, e.g. in the `transbase.jdbc` or `transbase.tbx` packages, and you are using or switching to a relatively new JRE or SDK version, then this could be caused by incompatibilities of the two involved runtime packages. Try to back up to an older JRE or SDK or check for a newer `tbjdbc.jar` release.

Chapter 8

Transbase Data Dictionary Extensions

8.1 The sysexternal Table

The Sysexternal table contains a single entry for each external resource stored in the database.

sysexternal	
rkey	INTEGER
rtype	TINYINT
owner	INTEGER
enabled	BOOL
debug	BOOL
lastmodified	DATETIME[YY:SS]
rname	CHAR(*)
rblob	BLOB
sblob	BLOB
comment	CHAR(*)

rkey of sysexternal: Identifies a resource, if rkey>=0. rkey<0 is reserved for storing external options, i.e. options for the virtual machine.

rtype of sysexternal: Denotes whether this resource is e.g. of type java class, java source or java archive (jar)

owner of systable: Denotes the owner of the table or view by the user's userid. To retrieve the owner's username, join the tables sysuser and sysexternal

enabled of sysexternal: Boolean flag, true if this resource is enabled for execution

debug of sysexternal: Boolean flag, true if this resource is to be executed in debug mode

lastmodified of sysexternal: Timestamp of the last modification of this resource, i.e. creation, modification or recompilation.

rname of sysexternal: Unique name identifying this resource, i.e. classname

rblob of sysexternal: The actual external resource in binary format

sblob of sysexternal: The external resource in source code representation. If corresponding rblob is missing, it will be generated by compilation of sblob on execution.

comment of sysexternal: Textual description of this resource's functionality, version, author, and other additional information

8.2 The sysexternalmethod Table

The sysexternalmethod table contains a single entry for each method of each external resource in this database.

sysexternalmethod	
rkey	INTEGER
mkey	INTEGER
enabled	BOOL
callonnull	BOOL
determ	BOOL
checkexcept	BOOL
ownerrights	BOOL
sqlaccess	TINYINT
dynasets	TINYINT
resulttype	CHAR(1)
sqlname	CHAR(*)
sqlsig	CHAR(*)
extsig	CHAR(*)
parammodes	CHAR(*)
paramnames	CHAR(*)
sqltabsig	CHAR(*)
exttabsig	CHAR(*)
methodname	CHAR(*)
comment	CHAR(*)

rkey of sysexternalmethod: Identifies the external resource to which the entry belongs. The name of the resource can be retrieved via a join between sysexternal and sysexternalmethod on the fields rkey.

mkey of sysexternalmethod: Unique numeric key that identifies a method

enabled of sysexternalmethod: Boolean flag, true if this method is enabled for execution

callonnull of sysexternalmethod: Boolean flag, true if this method is to be called if any of the in parameters is NULL.

determ of sysexternalmethod: Boolean flag, true if this method is deterministic, i.e. for a given set of argument values, the procedure or function returns the same values for out and inout parameters and function result.

- checkexcept of sysexternalmethod:** Boolean flag, true if explicit exception checking is to be performed of the method returns a NULL result
- ownerrights of sysexternalmethod:** Boolean flag, true if this method is to be executed with the privileges of it's owner.
- sqlaccess of sysexternalmethod:** Access indication of the routine. Can be NO SQL (=0), CONTAINS SQL (=1), READS SQL DATA (=2), or MODIFIES SQL DATA (=3). Boolean flag, true if this method contains SQL statements. statements.
- dynasets of sysexternalmethod:** Number of result sets this method returns at most (reserved for future versions)
- resulttype of sysexternalmethod:** Character denoting of which SQL type the result of this function is. Stored procedures always return 'V' for void.
- sqlname of sysexternalmethod:** Name by which this method is referenced from SQL statements
- sqlsig of sysexternalmethod:** Single char coded SQL type signature.
- extsig of sysexternalmethod:** JNI style signature string for java methods.
- parammodes of sysexternalmethod:** Defines mode for each parameter, i.e. 'I' for IN, 'O' for OUT, 'B' for both (Inout), 'M' for INSTANCE, 'R' for INSTANCEOUT.
- paramnames of sysexternalmethod:** Comma separated list of names for out parameters
- sqltabsig of sysexternalmethod:** Single char coded SQL type signature for TABLE FUNCTION.
- exttabsig of sysexternalmethod:** JNI style signature string for TABLE FUNCTION.
- methodname of sysexternalmethod:** Name of the actual method in the external resource comment of sysexternalmethod: Textual description of this method's functionality

8.3 The sysexternalpriv Table

Describes the privileges applying to external resources or methods of the database.

sysexternalpriv	
grantee	INTEGER
rkey	INTEGER
mkey	INTEGER
grantor	INTEGER
use_priv	CHAR(1)

- grantee of sysexternalpriv:** Describes whom this privilege is granted
- rkey of sysexternalpriv:** Identifies the external resource that is granted.
- mkey of sysexternalpriv:** Identifies the method of resource rkey to be granted. If mkey is NULL then all methods of resource rkey are granted.

grantor of sysexternalpriv: Id of the user granting the privileges

use_priv os sysexternalpriv: 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. rkey refers to field rkey of table sysexternal and mkey refers to mkey of sysexternalmethod.