IBM® Netezza® Analytics
Release 11.x

User-Defined Analytic Process Developer's Guide

Revised: June 21, 2018

Part Number 00J2009-06 Rev. 9

Note: Before using this information and the product that it supports, read the information in Notices and Trademark, on page 367.

Contents

Running Function Analytic Executables

4 Working with Examples

Scalar Function Examples

Converting to a Simple Table Function Examples

Table Function (Simulated Row Function) Examples

Simple Table Function Examples

Shapers and Sizers Examples

10 Aggregate AE Examples

11 Advanced Developer Principles

12 Environment and Shared Libraries Examples

13 Advanced Analytic Executable Topics

14 Remote Analytic Executables

15 Simple Remote Mode Examples

16 Advanced Remote Mode Examples

17 Debugging Analytic Executables

18 Debugging Examples

19 Integrated Examples

20 Administering AEs and Related Processes

21 Best Practices

APPENDIX A

File Names Used in Examples

APPENDIX B

Adding a Shared Library to the Perl Adapter

APPENDIX C

Extended R Language Functionality

APPENDIX C

Notices and Trademarks

List of Tables

List of Figures

Preface

Audience for This Guide

The guide is intended for developers and partners who are interested in developing analytics on the IBM Netezza appliance. You should be familiar with one or more programming languages, such as C, C++, Java, Fortran, Perl, or Python, as well as the basic operation and concepts of the IBM Netezza system.

Purpose of This Guide

This guide offers introductory and advanced concepts, including examples, to assist in the development of Analytic Executables (AEs) using User-Defined Analytic Processes (UDAPs). For complete details on UDXs, refer to the NPS guide, *IBM Netezza User-Defined Functions Developer's Guide*.

Symbols and Conventions

Note on Terminology: The terms *User-Defined Analytic Process (UDAP)* and *Analytic Executable (AE)* are synonymous.

The following conventions apply:

Italics for emphasis on terms and user-defined values, such as user input.

Upper case for SQL commands, for example, INSERT or DELETE.

Bold for command line input, for example, **`nzsystem stop`**.

Bold to denote parameter names, argument names, or other named references.

Angle brackets (< >) to indicate a placeholder (variable) that should be replaced with actual text, for example, `inza-<release_number>.zip`.

A single backslash ("\") at the end of a line of code to denote a line continuation. Omit the backslash when using the code at the command line, in a SQL command, or in a file.

When referencing a sequence of menu and submenu selections, the ">" character denotes the different menu options, for example, **Menu Name > Submenu Name > Selection**.

If You Need Help

If you are having trouble using the IBM Netezza appliance, IBM Netezza Analytics or any of its components:

Retry the action, carefully following the instructions in the documentation.

Go to the IBM Support Portal at: http://www.ibm.com/support. Log in using your IBM ID and password. You can search the Support Portal for solutions. To submit a support request, click the '**Service Requests & PMRs**' tab.

If you have an active service contract maintenance agreement with IBM, you may contact customer support teams via telephone. For individual countries, please visit the Technical Support section of the IBM Directory of worldwide contacts (http://www14.software.ibm.com/webapp/set2/sas/f/handbook/contacts.html#phone)

Comments on the Documentation

We welcome any questions, comments, or suggestions that you have for the IBM Netezza documentation. Please send us an e-mail message at netezza-doc@wwpdl.vnet.ibm.com and include the following information:

The name and version of the manual that you are using

Any comments that you have about the manual

Your name, address, and phone number

We appreciate your comments.

CHAPTER 1
Introduction

Sample Dataset Configuration

Standard data-mining datasets are used in this guide to provide examples, tutorials, and insight into how the various components of IBM Netezza Analytics might be used in real-world scenarios and how various functions and stored procedures interact in normal operation.

These datasets are not shipped with the IBM Netezza Analytics product. Before these examples can be used, the datasets must be acquired and installed on the NPS by a system administrator. The *IBM Netezza Analytics Administrator's Guide* provides information on how to perform these tasks.

SQL Functions

The SQL language provides a number of built-in functions, such as UCASE() and COUNT(). These functions fall into three categories:

Scalar Functions—Take zero or more data arguments and return a *single* data result per input row.

Table Functions—Take zero or more data arguments and return a *table* result; the returned table may contain a number of rows less than, equal to, or greater than the number of input rows.

Aggregate Functions—Take one or more data arguments and return a single value per group or window; grand aggregates return a single result for all the data

In this guide, these built-in functions are referred to as *SQL functions*.

User Defined Process Overview

The NPS allows you to extend SQL with user-defined functions, referred to as UDXs, as well as User-Defined Analytic Processes, also known as *Analytic Executables* (AEs). The Analytic Executable

concept allows a Netezza user to implement a freestanding, executable data-processing program that runs “out of process,” that is, outside the NPS system, and register it in a database.

UDXs can be thought of as the user-defined counterparts to built-in SQL Functions. They are called from SQL in the same manner and follow the same guidelines for input and output. AEs, while called from SQL like UDXs, are actually applications that run when called.

This section provides an overview of both UDXs and AEs before comparing and contrasting them. While UDXs and AEs are complementary solutions, they do have some significant differences that can help determine which solution is better suited for a given application. Since this guide focuses on AEs, the discussion of UDXs is limited to how they compare with AEs. For complete details on UDXs, refer to the NPS guide, *IBM Netezza User-Defined Functions Developer’s Guide*.

Note: The terms *User-Defined Analytic Process* (*UDAP*) and *Analytic Executable* (*AE*) are synonymous.

UDXs

Netezza's user-defined function[1] capability was implemented to extend SQL and provide advanced, custom functionality. The term *UDX* is used as a generic reference to Netezza's user-defined scalar, table, and aggregate functions. UDXs provide functionality corresponding to the three categories of built-in functions (see SQL Functions, above):

SQL function	UDX Function
Scalar functions	User-defined functions or UDFs
Table functions	User-defined table functions or UDTFs
Aggregate functions	User-defined aggregates or UDAs

UDXs allow you to create these custom functions to perform specific types of analysis for your business reporting and data queries.

User-defined functions and aggregates must be written in C++. While they can use the full standard C library (LIBC), they should avoid interprocess communication (IPC) calls and other low-level operations.

User-Defined Functions

A user-defined function (UDF) is user-supplied code that is executed by the Netezza system in response to SQL invocation syntax. A user-defined function is a scalar function; that is, it returns one value for each input row.

A UDF invocation can appear anywhere inside a SQL statement where a built-in function can appear, including restrictions (WHERE clauses), join conditions, projections (SELECT from lists), and HAVING conditions. A UDF can accept zero or more input values but produces one output value. Input values to a UDF can be literals, column references, or expressions. The data types of inputs and output must

[1] Netezza’s support for user-defined functions generally follows the model used in the 2003 SQL Standard for SQL-Invoked Routines.

be Netezza built-in data types.

User-Defined Table Functions

A user-defined table function (UDTF) is a function that can be invoked in a FROM clause of a SQL statement. A table function can appear in a SQL clause at almost location where a table normally appears. It returns a table shape with columns that have names and types. Unlike the scalar nature of a UDF, a table function can return zero or more rows per input row. A table function can be invoked with arguments, including literals and non-literal expressions containing columns from other tables.

Table functions can be used for tasks such as expanding data from one row into many rows, producing summaries by combining data from many rows, creating custom summaries in table form, or performing “unpivot” operations such as combining the data from several related tables into a new combined table.

Note: A UDTF should be invoked with at least one argument.

User-Defined Aggregates

A user-defined aggregate (UDA) is user-supplied code that implements, on the Netezza system, the various phases of aggregate evaluation, such as initialization, accumulation, and merging.

UDAs provide new types of aggregation functions to expand upon built-in aggregates such as count(), sum(), avg(), max(), or min(). While UDAs can take multiple arguments, in terms of output they are scalar and produce only one output value. UDAs can be used in a SQL statement anywhere a built-in aggregate can appear as either grand, grouped, or windowed aggregates.

Analytic Executables (AEs)

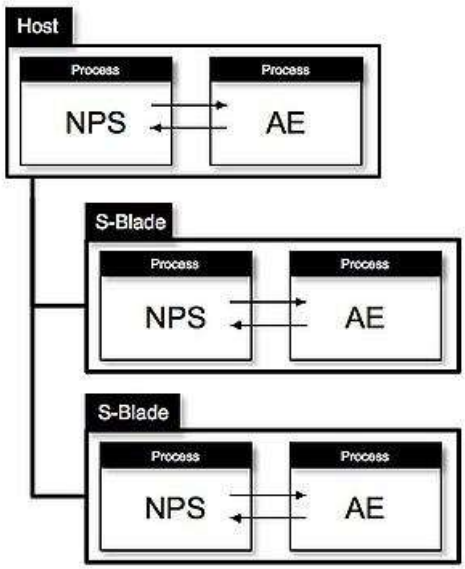
Analytic Executables are extremely powerful tools for analytics development and processing. While Netezza's UDX functionality enables user-supplied C++ code callable from SQL functions, AEs are freestanding executable programs that can be called from SQL functions.

Like SQL functions and UDXs, there are different types of AEs to handle different function types: function and aggregate. That is, scalar and table SQL functions are used to invoke their respective function AEs, while aggregate SQL functions are used to invoke aggregate AEs.

AEs are unique in that they can be written in any language with the ability to call C libraries, including C, C++, Java, Python, Fortran, Perl, and R[2]. Once written, an AE can be published and registered as a function on the nodes of a supporting IBM Netezza appliance. It can then be executed to carry out its intended purpose. An AE can operate either on the host or on the parallel nodes of the appliance. The figure highlights this process.

For information about how to install R, see the IBM developerWorks Netezza Developer Network (NDN) community. You need to register first at developerWorks (https://www.ibm.com/developerWorks). Search for “NDN” to locate the Netezza Developer Network community. Follow the instructions in the overview page to get access to the private part of the community.

Figure 1: AE Operation on Host and S-Blades (SPUs)



AEs provide a number of new capabilities in the development of analytics. Benefits of using AEs include:

- Providing a method for diverting the data stream for a specific operation or purpose.
- Freedom to perform algorithmic operations on the data stream.
- Flexibility to write functions in C, C++, Java, Fortran, Perl, Python, and R.
- Allowing significantly computationally-intensive algorithms where the data resides.
- Use of libraries or languages that are not part of the NPS.

While the Netezza system already supports calls to user C++ functions that are called within an Netezza system process, AEs allow SQL functions to invoke application code in non-Netezza system processes running on the host and SPUs. Therefore, AEs provide more flexibility to build applications inside the Netezza system, including the ability to select programming languages and third-party software libraries that best achieve objectives.

AEs also provide greater control over the application lifespan. The lifespan of a UDX is controlled by the NPS system and is always less than the lifespan of a query. The lifespan on an AE may optionally be controlled by the Netezza system or by the AE itself, and can be indefinite.

Function AEs

A function AE generally uses a file I/O or “standard input/output” data flow paradigm. A SQL scalar function or table function that returns exactly one output row per input row and whose return row contains exactly one column has the following general structure, which varies depending on the programming language:

```
getDataConnectionToNPS();
while (getNextInputRow)
{
     getInputColumns();
     setOutputColumn(); // only one
     outputResultRow();
}
closeDataConnection();
```

The same AE may be invoked from a SQL scalar function and a table function.

Some AE programming language implementations also support a callback paradigm that can be used only when returning exactly one output row per input row.

Often, SQL table functions return more than one output row per input row and define an output row with more than one column. In this case, an AE has the following general structure:

```
getDataConnectionToNPS();
while (getNextInputRow)
{
      getInputColumns();
      limit = { 0 or a positive integer, app specific };
      for (int i = 0; i < limit; i++)
      {
           setOutputColumns();
           outputResultRow();
      }
}
limit = { 0 or a positive integer, app specific };
for (int i = 0; i < limit; i++)
{
     setOutputColumns();
     outputResultRow();
}
closeDataConnection();
```

Notice that there can be more than one column per output row and any number of output rows per input row. After the end of input, the AE can return output rows that are not associated with any input row. This type of AE can only be invoked by a SQL table function.

Aggregate AEs

An aggregate AE uses a different API than a function AE. This difference exists because the structure of an aggregation algorithm is completely different. An aggregate AE has the following general structure, which varies depending on the programming language:

```
getDataConnectionToNPS();
while (getNextMessage())
{
     case INTIALIZE:
           initialize aggregation state
        break;
     case ACCUMULATE:
          optionally set aggregation state based on application algorithm
       break;
     case MERGE:
           merge results from different data
        slices break;
     case FINAL_RESULT:
```

```
            set final result based on application
          algorithm break;
}
closeDataConnection();
```

Languages such as C++ and Java represent this structure as callbacks, for instance, an initialize function is called on initialization, an accumulate function is called on accumulate, and so on.

Programming Model

AEs are invoked from traditional SQL functions that require defined input and output parameters. When invoked from scalar or table functions, AEs deviate from the traditional database-programming model and can use a model of programming that operates in the manner of File I/O to access the NPS system datastream. The datastream consists of multiple rows containing one or more columns. Metaphorically, the rows are like records in a file. In a loop, the AE reads input rows, performs application-specific processing, and writes output rows.

When writing an AE, the structure is similar to file semantics:

```
initializeAE();
while (moreData) {
     getNextRecord();
     doSomething();
     maybeOutputResults();
}
maybeOutputResults();
done();
The approach is to mimic the file i/o pseudo-code:
openFile();
while (notEOF) {
     getNextLine();
     doSomething();
     maybeWriteOutput();
}
maybeWriteOutput();
closeFile();
```

Callback Programming Model

Aggregate AEs always use a callback model. Function AEs that return exactly one output row per input row may optionally use a callback model instead of the File I/O model.

Choosing between AEs and UDXs

While UDX functions are invoked as callbacks where the NPS system calls UDX notification and implementation functions, an AE application runs as a conventional program that calls the NPS system to retrieve and return data. There are three types of user-defined SQL functions, described in the section SQL Functions, above. These functions should not be confused with the generic UDX term for functions that consist of user-defined functions (UDF), user-defined table functions (UDTF), user-defined aggregates (UDA), and user-defined shared libraries. (For example, a UDF is a type of scalar function, but a scalar function does not need to be a UDF.) In other words, the type is referred to as a SQL function, which can be a built-in, UDX, or AE.

When called as scalar functions or table functions, AEs have a different control mechanism than UDFs and UDTFs. The UDX functions are invoked as callbacks such that the NPS system calls UDX notification and implementation functions. In contrast, an AE application runs as a conventional program that calls the NPS system to retrieve and return data. To the UDX it appears that the NPS system calls the UDX, while the AE view is that the AE calls the NPS system. For the user, it means that an AE looks like a typical application, where, conceptually, input rows are similar to standard input and output results similar to standard ouptut.

For aggregate functions, both AEs and UDXs use a notification/callback mechanism. Additionally, some AE Language Adapters, such as those for C++ and Java, provide an additional alternate callback message handler interface for functions.

The existing UDX capability of the NPS system and the new AE capability are complementary solutions. Because UDX operates inside an NPS system process, raw data transfer is faster and the system resource footprint is smaller. Because AEs operate outside the NPS system process, there is much more freedom and control over the application design.

As a general guideline, the three primary responsibilities of AE functionality are:

Process control.

Data marshaling between the NPS system and the AE.

Coordination between the specific SQL function calling mechanism and the AE. (A calling mechanism refers to whether the AE was called as a table function, scalar function, or aggregate function.)

The functionality for these three responsibilities is designed to be language-independent. Although Netezza provides specific AE support for a number of common languages, AEs are designed to support any programming language capable of calling C functions. They support both 32-bit and 64-bit applications.

Since AEs extend the capabilities of the NPS appliance, there are some instances when a UDX may be more suited to the solution. The following table offers some general guidelines when considering between UDXs and AEs.

Table 1: Guidelines for choosing between UDXs and AEs

Application	UDX or AE
C++ with a simple algorithm that processes large amounts of data	While it could be an AE, it typically should be written as a UDX. When algorithms are simple, the majority of application time is used performing data transfer, a task better suited to UDXs.
Java	Since a UDX must be written in C++, any Java applications must be implemented as an AE.
C++ with complex algorithms that makes heavy use of third-party application libraries	Either a UDX or AE works; however the following application requirements suggest or mandate the use of an AE:

Application	UDX or AE
	► The use of a third-party library that is incompatible with the NPS system ► A requirement for life cycle control. UDXs always have a limited life span. AEs may have indefinite life spans and may be run as daemons ► If the AE's conventional program design (as opposed to UDX callbacks) is a better fit for the application.

SQL Invocation of AEs

An AE scalar function can be used wherever normal SQL scalar functions appear, such as the NPS system built-in scalar functions. A scalar function returns one value per input, as shown here:

```
SELECT myScalarFunction('+', 1, 2, 3)
```

A table function can be used almost anywhere SQL allows a table. (Refer to the NPS guide, *IBM Netezza User-Defined Functions Developer's Guide* for more information on table functions.) The table function returns a table that may contain a number of rows less than, equal to, or greater than the number of input rows.

```
SELECT * FROM TABLE WITH FINAL(myTableFunction('+', 1, 2, 3))
```

An aggregate function can be used where SQL allows an aggregate function. An aggregate function returns a single value per group or window. A grand aggregate returns a single result for all the data.

```
SELECT category, myAggregate(f1, f2, f3, f4) FROM mytable GROUP BY category
```

Note that arguments to the SQL function become the input to the AE. The output of the AE, in turn, becomes the results of the SQL function. For a language like C, a data connection is represented as an opaque handle to an API. For a language like C++ or Java, a connection is represented as a class object.

CHAPTER 2
Developer Principles

Understanding the AE Runtime System

The AE runtime system is used to launch local AEs and is usually used to launch remote AEs. The AE runtime system communicates between the Netezza Platform Software (NPS) and AEs.

The AE API can be used in processes that are not started by the AE runtime system. However, it is usually more convenient to allow the AE runtime system to control the lifespan of an AE application, or to use it to launch an AE application that then controls its own lifespan.

AE runtime options can be set and specified using AE and/or environment variables, which are key/value pairs similar to UNIX environment variables. The command line utility to register AEs, **register_ae**, can be used to set these variables implicitly, using command line options. Using the **register_ae --environment** option, AE environment variables can also be set directly, overriding variables set by **register_ae**.

Using the AE runtime system to launch an AE provides considerable control over the process startup state. Items that can be specified include the executable file path, command line arguments to the executable, the value of system environment variables such as PATH and LD_LIBRARY_PATH, and application-specific operating system environment variables. Different environment variable values can be specified on the host instead of the SPU, including the logging level.

The process started by the AE runtime does not have to be the process that uses the AE API to perform data processing. For example, the AE runtime could be used to run a script that then executes a program that uses the AE API.

Marshaling Data

The AE runtime system's primary purpose is to marshal data between the NPS system processes and AE processes. The mechanism to perform the marshaling is hidden from the AE programmer by the API. Data transfer is highly optimized; however, data transfer between processes is not as fast as data transfer within a process.

Accessing the AE Export Directory Tree

The network shared disk space contains the AE export directory tree, which is the primary location for all AE executable files and application data. The host and all SPUs have access to this directory tree. The AE export directory tree also contains system tools such as supported compilers and runtime environments. The AE export directory file share is stored in the Linux NZ_EXPORT_DIR environment variable. In the samples provided in this section, the export directory tree location and $NZ_EXPORT_DIR are assumed to be the default location, /nz/export. For some installations, however, a different path may have been used.

Note: Some features of IBM Netezza Analytics may not work if the default path value is not used.

All AE files should be installed under one of the following:

```
/nz/export/ae/adapters/...
/nz/export/ae/core/...
/nz/export/ae/languages/...
/nz/export/ae/products/...
/nz/export/ae/sysroot/...
/nz/export/ae/utilities/...
/nz/export/ae/workspace/...
```

Utilities Directory

The utilities directory contains the **compile_ae** and **register_ae** scripts. The default location is /nz/export/ae/utilities/bin, which also includes other scripts and binaries, including permission scripts and backup/restore scripts. It may be useful to put this directory on the Linux executable PATH.

```
/nz/export/ae/utilities/bin
```

Other libraries and files needed by the utility binaries are located in the following directory:

```
/nz/export/ae/utilities/lib
```

Applications Directory

The applications directory is the recommended location for runtime files such as binaries, Java class files, and Python script files that are required for a given AE.

The following two locations are recommended:

```
/nz/export/ae/applications/<db>/
/nz/export/ae/applications/<db>/<user>
```

By default, the **compile_ae** and **register_ae** scripts target the /nz/export/ae/applications/<db>/<user> directory. The /nz/export/ae/applications/<db> directory is the only directory tree backed up for a given database by the IBM Netezza Analytics backup utility when using it in single database mode rather than all database mode. For IBM Netezza Analytics backup and restore, the name of the database, <db>, in the path should match the database for which the AE is registered.

Note: This does not apply to the standard Netezza Platform Software (NPS) backup and restore

convention.

Products Directory

The products directory contains individually wrapped AE products. The products subdirectory should be organized as:

```
/nz/export/ae/products/<company>/<product>/<version>
```

Workspace Directory

The workspace directory is used for temporary and semi-temporary data that is created by a specific product. The following path should be used:

```
/nz/export/ae/workspace/<VENDOR_DIR>/<PRODUCT> -> Cross-user data
for a given product
/nz/export/ae/workspace/users/<username>/<VENDOR_DIR>/<PRODUCT> ->
Single-user data for a given product
```

Adapters Directory

The adapters directory, /nz/export/ae/adapters, contains the functionality to adapt a given language to the NPS system so that it can be called as an AE. These adapters define the interface and enable programming of AEs in a particular language.

The only subdirectories that should reside under /nz/export/ae/adapters are those named for the languages supported by the adapters. Under the language, there should be a version that applies to the specific adapter, for example:

```
/nz/export/ae/adapters/<language-name>/<1-digit-version>/
```

Languages Directory

The languages directory contains the language-only installations for the SPU and the host. Subdirectories typically found in this directory might include "java" or "python."

These subdirectories are organized as:

```
/nz/export/ae/languages/<language-name>/<language-version>/spu/
-> The spu version of the language
/nz/export/ae/languages/<language-name>/<language-
version>/host/ -> The host version of the language
```

Core Directory

The core directory, located at /nz/export/ae/core, contains groups of Netezza Analytics functionality provided as part of the core system. These functionality tools may include Intel MKL, Boost, and Open MPI. If there are third-party tools needed by the core functionality, the tools are installed in .../core/tools. If the tool must be used outside the "core" functionality, then it is installed into the .../products directory instead.

The subdirectory should be organized as follows:

```
/nz/export/ae/core/<category>/<2-digit-version>/...
/nz/export/ae/core/thirdParty/<product>/<version>/...
```

```
/nz/export/ae/core/common/<To-be-determined>
```

Sysroot Directory

The /nz/export/ae/utilities/bin/sysroot directory contains the base Linux tool set published by Netezza for AE functionality. This directory provides tools used for compiling binaries and libraries to run on the host and the SPU, including standard system libraries and headers. This directory also provides runtime libraries that can be linked by either the NPS system or AEs. It is important that all of the host applications and all of the SPU applications be consistent in their use of compilers; failure to do so may result in applications being unable to run or encountering bugs. Compiler consistency is particularly important with C++, where different versions of the compiler can result in different versions of the C++ API, which can cause problem ranging from naming problems to exception handling.

The compilers and libraries used for both the host and the SPU are determined by the compilers used by the core NPS product. The compilers in the sysroot are intended to run on the host since SPU utilities are cross compilers; however, these compilers are capable of running on the SPUs as well. Other utilities are for the SPUs only.

This sysroot is necessary as the compilers and libraries that ship with the standard NPS system are stripped down.

32-bit

The 32-bit sysroot containing the host and SPU subdirectories can be found at:

```
$NZ_EXPORT_DIR/sysroot/
```

64-bit

The 64-bit sysroot containing the host64 and spu64 subdirectories can be found at:

```
$NZ_EXPORT_DIR/sysroot/
```

The 64-bit sysroot contains fewer binaries than its 32-bit counterpart.

Compilation and Registration Overview

An AE installation uses a system where code is compiled, deployed, and then registered. Two command line utilities are used to perform these tasks, **compile_ae**, which is used for compilation and deployment, and **register_ae**, which registers the AE and connects it to the NPS.

Both utilities use a series of options, such as **--language**, **--user**, **--db**, and **--template**, which define various configurations for the compilation and registration processes.

User Definition

The same user must be used for both **compile_ae** and **register_ae**. If the **--user** option is not specified, the value set by the NZ_USER environment variable is used by default. If the **--user** switch is supplied in only one command and NZ_USER does not match the value specified by **--user**, **compile_ae** deploys the file to a different location than that used by **register_ae**, causing an error.

Extending compile_ae and register_ae

Both **compile_ae** and **register_ae** are implemented using the *Perl Template Toolkit*. Although not required, extending the **compile_ae** or **register_ae** system may be useful. In this event, it is recommended that you create new template files instead of modifying the Netezza templates.[3]

Assuming the AE export directory is located at /nz/export/ae, the existing templates are located at:

```
/nz/export/ae/adapters/<language>/<version>/templates
```

The **--ldir** option can be used to point to an alternative adapters directory.

The **--define** option can be used to set specific template variables. For example:

```
--define templatevar=value
```

Compiling and Deploying Analytic Executables

An AE installation includes the **compile_ae** command line utility, which is used to compile all languages and deploy the result. Scripting languages such as Python also use **compile_ae** for deployment, as it is aware of the AE backup process used by the NPS. The utility can also be used indirectly as a source of information when creating a custom compile process and subsequent deployment. Using **compile_ae --help** provides help about usage and options.

To see verbose output from the **compile_ae** utility, use the **--echo** option. The output describes what is required to compile AE code and the information can then be used in the custom build process. Depending on which programming language is used, AE compilation may require shared libraries, include files, and JAR files. For languages like C++ and Fortran, which are compiled into native

[3] The following resources may be useful when extending **compile_ae** or **register_ae** options: *Perl Template Toolkit* by Darren Chamberlain, David Cross, and Andy Wardley, O'Reilly, 2003, ISBN: 0-596-00476-1 or the Template Toolkit web site.

machine code, output executable objects are produced for both the host and SPUs.

The basic syntax of **compile_ae** is:

```
compile_ae <options> <source_file_names>
```

This section provides a summary of the most important information about the **compile_ae** utility.

Note: Most of the options begin with two hyphens: "--" which may be difficult to view in various screen-based documentation formats.

User Configuration before Compilation

The **compile_ae** utility can always be run by root or the nz user. However, database users other than root and nz must be added to the Linux group **nz** before they can compile an AE. Users can be added to the group using the following command as user root, where nz specifies the group:

```
gpasswd -a <user> nz
```

File Locations

The location of files produced by **compile_ae** and input files to **register_ae** defaults to:

```
<AE Export Directory>/ae/applications/<ae_db>/<db_username>
```

The location of the AE export directory is usually contained in the NZ_EXPORT_DIR Linux environment variable. The default location of input files can be modified.

Common Options

The following are common options of the **compile_ae** utility.

--language {system | cpp | java | fortran | perl | python64 | python3x |r}
Identifies the programming language used to write the AE. Accepted options are system (for the base C API), cpp (for C++), java, fortran, perl, python64, python3x (for Python version 3.x) and r[4]. For example:

```
--language java
```

--template <registration_type>
Defines the registration type. Look at the language's template directory for a complete list. Common values include:

compile—used to compile non-scripting languages; also deploys.

compile64—used for 64-bit version of compile; note that only C and C++ support 64-bit

For information about how to install R, see the IBM developerWorks Netezza Developer Network (NDN) community. You need to register first at developerWorks (https://www.ibm.com/developerWorks). Search for "NDN" to locate the Netezza Developer Network community. Follow the instructions in the overview page to get access to the private part of the community.

operation.

deploy—used for deploying a script to the export directory; this is the only valid value for Perl and Python AEs.

deploydir—used for deploying the contents of a directory to the export directory.

compilejar—used to produce a Java JAR file.

For example:

```
--template compilejar
```

--version <ae_version>

Sets the AE version in use. This value is tied to the current IBM Netezza analytics version, and is used to track incompatibility. (You can display the contents of /nz/export/ae/adapters/system_ae/ to list versions.) For example:

```
--version 3
```

--db <system_db>

Selects the target Netezza system database and the default location of output files (where the compiled/deployed files are copied). The location must be the same as the location expected by register_ae. Otherwise when the preocess registers the executable, it will refer to a file that does not exist and therefore cannot be run.

By default (if --db is not specified), the target database is the database set in the Linux environment variable NZ_DATABASE. For example:

```
--db dev
```

--user <db_user>

Sets the database user name. This option also helps determine the default location of output files. By default (if --user is not specified) compile_ae uses the Linux environment variable NZ_USER. See User Definition for additional information. For example:

```
--user Joe_User
```

--pw <password>

Sets the database password. By default (if --pw is not specified) compile_ae uses the Linux environment variable NZ_PASSWORD. For example:

```
--pw MyPassword
```

The **compile_ae** utility also supports nzpassword caching; nzpassword is an NPS system command line utility.

--compargs “<args> <args>...<args>”

Specifies additional arguments to pass to the compiler as specified. This option is typically used for compiled languages, but can be used for other purposes by some compile templates. For example:

```
--compargs "-g -Wall"
```

--linkargs “<args> <args>...<args>”

Specifies additional arguments to pass to the linker. This option is typically used for linked languages, but can be used for other purposes by some compile templates. For example:

```
--linkargs "-g"
```

--hostcompargs “<args> <args>...<args>”

Specifies additional arguments relevant to the host that are to be passed to the compiler as specified. This option is typically used for compiled languages, but can be used for other purposes by some compile templates. For example:

```
--hostcompargs "-g -Wall"
```

--hostlinkargs “<args> <args>...<args>”

Specifies additional arguments relevant to the host that are to be passed to the linker as specified. This option is typically used for linked languages, but can be used for other purposes by some compile templates. For example:

```
--hostlinkargs "-g"
```

--spucompargs “<args> <args>...<args>”

Specifies additional arguments relevant to the SPU, which are passed to the compiler as specified. This option is typically used for compiled languages, but can be used for other purposes by some compile templates. For example:

```
--spuhostcompargs "-g -Wall"
```

--spuhostlinkargs “<args> <args>...<args>”

Specifies additional arguments relevant to the SPU, which are passed to the linker as specified. This option is typically used for linked languages, but can be used for other purposes by some compile templates. For example:

```
--spuhostlinkargs "-g"
```

--exe <filename>

Sets the executable name of the executable file or Java JAR file being created. Since Java can use class files directly, the use of **--exe** to specify a JAR file is optional. If **--exe** is not specified, the output file takes the name of the first input file minus the file type suffix. To use the –exe option:

```
-- exe testapply
```

--path <pathname>

Sets the directory path of the location where files are stored. For example:

```
--path /nz/export/ae/applications/special
```

Note: The output location for **compile_ae** files and the input location for **register_ae** files for a given

application must match. The **register_ae** utility searches for files in the location where **compile_ae** deploys files. This can be configured using the default database settings, by specifying the same database options for **compile_ae** and **register_ae**, or by using the **--path** option.

--warn

Provides a warning instead of an error, allowing processing to continue, if the output location does not start with <ae_export_dir>/ae/applications/<ae_db>. Warning applies only if you are using a non-standard output location with the **--path** option. For example:

```
--warn
```

Sample output:

```
WARNING: --path should start with /nz/export/ae/applications/mydb
```

--object <filename>

Provides an optional name of output object file. Use this option to compile one source file into an object file of the specified name. This may be an intermediate file with the final file specified by **--exe**. For example:

```
--object myobj.o
```

Complete List of compile_ae Options

This section provides a complete list of options for **compile_ae**.

The following syntax is supported for using the options:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae [OPTION]... <srcfile> [srcfile]
```

To list all available options, run the following command:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --help
```

The following list shows all supported options:

```
   --echo                 Echo substituted template before running it
as
                          well as shell output
   --file file            Use specified template file instead of
installed
                          template
   --base base            Base directory for NPS instead of env:
NZ_KIT_DIR
                          or /nz/kit. Also available as template
variable
                          ae_kit.
   --ldir base            Base directory for Language configurations
instead
                          of /nz/export/ae/adapters
   --language lang        Specify language configuration to use.
Languages
```

```
                          are cpp, fortran, java, perl, python64,
python3x, system.
                          available as template variable ae_language.
   --version ver          Specify language version to use. Also
available as
                          template variable ae_lang_ver.
   --template name        Specify the template to use. Common ones are
                          compile,deploy
   --user username        NPS username (defaults to env: NZ_USER)
                          Also available as template variable ae_user.
   --pw password          NPS password (defaults to env: NZ_PASSWORD)
                          Also available as template variable
ae_password.
   --db database          NPS database (defaults to env: NZ_DATABASE)
                          Also available as template variable ae_db.

   -h, --help             Show this command usage.
General Template options:
   Predefined template variables include:
       ae_export_dir          Location of shared export directory
       ae_src_file            srcfile argument(s)
       ae_sysroot_spu         Location of system root on spu
       ae_sysroot_host        Location of system root on host
       ae_library_path_spu    LD_LIBRARY_PATH for spu
       ae_library_path_host   LD_LIBRARY_PATH for host
   --path dir             Location where AE files will be
stored.Defaults to
                          <ae_export_dir>/ae/applications/<ae_db>/<ae_u
ser>.
                          Available as template variable
                          ae_application_path.
   --warn                 Warn instead of error if --path does not
start
                          with <ae_export_dir>/ae/applications/<ae_db>.

   --define var=value     Define template variable.
   --idir dir             Install dir for the AE language. Available as
                          template variable ae_install_dir.
   --exe name             Set executable name to name. Available as
template
                          variable ae_exe_name.

   --object name          Set object file name to name. Available as
                          template variable ae_object_name.
   --compargs args        Additional arguments to pass to the compiler
as
                          is. Available as template variable
ae_comp_args.
```

```
   --linkargs args         Additional arguments to pass to the linker as
is.
                           Available as template variable ae_link_args.
   --hostcompargs args     Additional arguments to pass to the compiler
as
                           is. Available as template variable
                           ae_host_comp_args.
   --hostlinkargs args     Additional arguments to pass to the linker as
is.
                           Available as template variable
ae_host_link_args.
   --spucompargs args      Additional arguments to pass to the compiler
as
                           is. Available as template variable
                           ae_spu_comp_args.
   --spulinkargs args      Additional arguments to pass to the linker as
is.
                           Available as template variable
ae_spu_link_args
```

R AE Compilation

When working with R, compilation does not correspond to the typical process of creating machine code from source code, but rather preparing a user-provided source file to work with the R Adapter. A tool that is dedicated to this purpose reads the R source file, checks whether the required objects exist, serializes these objects, and then writes the output to the *Applications Directory*.

An input file is required for **compile_ae.** The R input file must contain a number of objects that are required by the R Adapter. Predefined object names have an **nz.** prefix that is internally removed when the serialization takes place, which is important for the nzLibrary for R client packages but not for processes on the server side. However, all objects present in the file are serialized and then available during the R AE execution.

The following list shows all the standard objects. The purpose for most of these objects is explained in the sections where extended functionality is described.

nz.fun—the main function object; this function is called when its AE is invoked

nz.shaper—the shaper function object

nz.shaper.list—a list defining the output signature; required when **nz.shaper** is not a function, but an **std** character value

nz.mode—a character value that defines the working mode; the default value is **run**

nz.args—a list of optional arguments for **nz.fun**

nz.cols—required in the *tapply* working mode

nz.init, **nz.accum**, **nz.merge**, and **nz.final**—the four functions that are required by user-defined aggregates (UDAs)

Registering Analytic Executables

An AE is connected to the NPS by a SQL function registration. This registration can be a scalar, table, or aggregate function. A single AE can be registered with different signatures (i.e., as more than one function). For example, a given executable could be registered as both a scalar and a table function, with the AE operating properly based on the registration/invocation used.

An AE installation includes a command line utility called **register_ae** , which is used to associate SQL functions with AEs. In addition, **register_ae** sets the following properties of the function.

Type of SQL function, either scalar, table, or aggregate.

The signature of the function using input arguments.

The function return value type or return table definition.

Use the **register_ae** option **--help** to display help about usage and options. The basic syntax of **register_ae** is:

```
register_ae <options>
```

Note: Most of the options begin with two hyphens: “--” which may be difficult to view in various screen-based documentation formats.

Common Options

--sig <function(<arg>)>

Specifies the SQL function name and input argument signature. The signature must be double quoted to prevent the shell from interpreting the parentheses literally. For example:

```
--sig "applyResult(varchar(16), double)"
```

It is possible to define a function that takes a variable number of arguments using the "varargs" keyword. For example:

```
--sig "applyOperation(varargs)"
```

For a complete list of supported data types, see the *Netezza Database User's Guide*. For more information about signatures, see the *Netezza User-Defined Functions Developer's Guide*.

--return <value_type>

Sets the return value or return table of the function. For example, for a scalar function or aggregate:

```
--return double
```

For a table function:

```
--return "table(total double, label varchar(128))"
```

For a table function, the return type must be double quoted to prevent the shell from interpreting the parentheses literally.

It is possible to define a function that returns a table defined at runtime with the shaper API using the "any" keyword. For example:

```
--return "table(any)"
```

--language {system | cpp | java | fortran | perl | python64 | python3x| r}

Sets the programming language used to write the AE. Accepted options are system (for the base C API), cpp (for C++), java, fortran, perl, python64, python3x (for Python version 3.x) and "r"[5]. For example:

```
--language java
```

--template <registration_type>

Defines the registration type. Look at the language's template directory for a complete list. Common option values include:

udtf—Table function

udf—Scalar function

uda—Aggregate function

udtf64—64-bit version of **udtf**

udf64—64-bit version of **udf**

uda64—64-bit version of **uda**

Note: Not all languages support 64-bit operation.

For example:

```
--template udtf
```

--version <version>

Sets the AE version in use. This value is tied to the current IBM Netezza analytics version, and is used to track incompatibility. (You can display the contents of /nz/export/ae/adapters/system_ae/ to list versions.) For example:

```
--version 3
```

--db <system_db>

Selects the target Netezza system database (the database where the registration occurs) and the default location of output files. By default (if –db is not specified), the target database is the database set in the Linux environment variable NZ_DATABASE. For example:

```
--db dev
```

For information about how to install R, see the IBM developerWorks Netezza Developer Network (NDN) community. You need to register first at developerWorks (https://www.ibm.com/developerWorks). Search for "NDN" to locate the Netezza Developer Network community. Follow the instructions in the overview page to get access to the private part of the community.

--user <db_user>

Sets the database user name. This option also helps determine the default location of registration input files. By default (if --user is not specified) register_ae uses the Linux environment variable NZ_USER. See User Definition for additional information. For example:

```
--user Joe_User
```

--pw <password>

Sets the database password. By default (if --pw is not specified) register_ae uses the Linux environment variable NZ_PASSWORD. For example:

```
--pw MyPassword
```

The **register_ae** utility also supports nzpassword caching; nzpassword is a NPS system command line utility.

--level {0 | 1 | 2 | 3}

Controls the log file output. Following are descriptions of the log levels. Each level includes output of the lower level(s). Note that higher numbers provide more information that may be useful for technical support.

Table 2: Log levels

Level	Description
Level 0	No information; disables logging.
Level 1	Basic, user -evel information about the execution run. The amount of data output is fixed.
Level 2	The child (AE) side of the inter-process communication. The amount of data output is variable and can be very large.
Level 3	The parent (NPS) side of the inter-process communication. The amount of data output is variable and can be very large.

For example:

```
--level 1
```

--exe <filename>

Specifies the name of an executable object, a Python script file name, or a Java JAR file. With Java, because exe is only needed when using JAR files, and Java can use class files directly, the use of --exe to specify a JAR file is optional. For example:

```
--exe <executable>
--exe apply.jar
--exe applyop.py
```

--define
An optional switch that can be used to set specific template variables, controlling the output mode. For example:

```
--define templatevar=value
```

--define java_class=<class_name>
Specifies the Java class to execute. The --exe option is used to specify a JAR file. You must specify at least one of thee options, and both can be specified. For example:

```
--define java_class=org.netezza.sample.ApplyDriver
```

Note: This is a specific example of a more general option used within the Template Toolkit used to implement **compile_ae** and **register_ae**. The general form is "--define <template_variable>=<value>".

--path <pathname>
Sets the directory path to the location of the input files. For example:

```
--path /nz/export/ae/applications/special
```

Note: The output location for **compile_ae** files and the input location for **register_ae** files for a given application must match. The **register_ae** utility searches for files in the location where **compile_ae** deploys files. This can be configured using the default database settings, by specifying the same database options for **compile_ae** and **register_ae**, or by using the **--path** option.

--warn
Provides a warning instead of an error, allowing processing to continue, if the output location does not start with <ae_export_dir>/ae/applications/<ae_db>. Warning applies only if you are using a non-standard output location with the **--path** option. For example:

```
--warn
```

Options for Remote AEs

By default, AEs are local where the AE process life cycle is controlled by the NPS system. Remote AEs enable launching and connecting to independently running AE processes. These processes run outside the NPS system process tree "daemon style." Communication with a remote AE is performed through a *connection point*, which is an abstract address consisting of {remote name [,session ID] [,transaction ID] [,dataslice ID]}.

The connection point can be built using a combination of remote name, session ID, transaction ID, and data slice ID.

For example if only remote name is specified there is one AE per SPU/host (each user session shares the same AE instance). If remote name and session ID are specified, there is one AE per user session per SPU/host (a user session has its own private AE instance on each SPU and the host).

Only remote name is required. The other three, described below, are optional.

--remote

Indicates that the AE is a remote AE. If specified, requires the –rname option (below).For example:

```
--remote --rname applyop
```

--rname <ae_name>

Specifies the name of the remote AE. This option is required if the **--remote** option is specified. For example:

```
--remote --rname applyop
```

--rsession

Adds the session ID as an element of the connection point address (optional). For example:

```
--rsession
```

--rtrans

Adds the transaction ID as an element of the connection point address (optional). For example:

```
--rtrans
```

--rdataslice

Adds the dataslice ID as an element of the connection point address (optional). For example

```
--rdataslice
```

--replbysql

Defines that the SQL statement in which the AE is used can be replicated by SQL. To avoid mismatches between the master node and the subordinate node of your replicated system, ensure that the output of your AE is deterministic when you choose this option.

--replbyval

Defines that the SQL statement in which the AE is used should be replicated by value. If not specified otherwise, this is the default setting. Remote AEs are always replicated by value.

Launching a Remote AE

Registration for launching a remote AE is similar to the process for a local AE. For example, while the file to be executed is specified, there is no actual data request involved. A registration for AE launching should always be for a table function with an argument signature of bigint and return TABLE(aeresult varchar(255)).

--launch

Indicates that this is a launcher. For example:

```
--launch
```

AE Environment Variables and register_ae

Based on the arguments specified, **register_ae** sets a number of system AE environment variables. The behavior of the AE process is controlled by these variables, which include executable path, command line arguments, logging options, paths, and so on. To see which variables are set by an invocation of **register_ae**, use the **--echo** option.

--echo

Displays how the compile template is expanded. See the the nzsql **show** output in the **--environment** description for an example of --echo results.

--environment <variable>

Sets an AE system or user-specific environment variable. You can override the environment variables set by **register_ae**, use advanced AE registration variables not supported by **register_ae**, and set user specific environment variables.

For example, an AE system variable:

```
--environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"
```

Note: The starting substring “NZAE” is reserved for AE system environment variables and should not be used for user-specific variables.

User-specific variable example:

```
--environment "'APPLY_INFO'='true'"
```

The full power of AEs are available through the AE system environment variables, which are fully described in the section Introducing AE Environment Variables.

Once a function is registered, you can display the effects of the –environment options (and many other NPS system SQL function settings) using the SQL show command:

Example

A table function that calls a Java AE is registered the following way:

```
register_ae
--language java
--template udtf
--version 3
--sig "applyOperationV2Tf(varargs)"
--return "table(result double)"
--define java_class=org.netezza.education.ApplyDriverV2 --
level 1
--environment "'APPLY_FUNCTION'='OPERATION'"
--environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"
```

Running **show** in nzsql produces the following:

```
SHOW FUNCTION applyOperationV2Tf;

                              RESULT |           FUNCTION | BUILTIN | ARGUMENTS

-------------------------------+--------------------+---------+---
```

```
TABLE(RESULT DOUBLE PRECISION)  | APPLYOPERATIONV2TF |       f | ()
```

Use the verbose option to display additional information. In addition to using the show command, you can run **register_ae** with the **--echo** option to display this output. To fully understand the output, see Introducing AE Environment Variables. Below are the AE environment variables from the applyOperationV2Tf example:

```
SHOW FUNCTION applyoperationV2TF VERBOSE;

APPLY_FUNCTION=OPERATION
NZAE_LOG_DIR=/nz/export/ae/log
NZAE_REGISTER_NZAE_DYNAMIC_ENVIRONMENT=0
NZAE_REGISTER_NZAE_EXECUTABLE_PATH=
/nz/export/ae/languages/java/6.13.0/jdk1.6.0_13/bin/java
NZAE_REGISTER_NZAE_HOST_ONLY_NZAE_DEBUG=1
NZAE_REGISTER_NZAE_HOST_ONLY_NZAE_PREPEND_LD_LIBRARY_PATH=
/nz/export/ae/adapters/java/3/sys/nz/lib/host
NZAE_REGISTER_NZAE_HOST_ONLY_NZAE_REMOTE =0
NZAE_REGISTER_NZAE_HOST_ONLY_NZAE_REMOTE_LAUNCH_VERBOSE=0
NZAE_REGISTER_NZAE_HOST_ONLY_NZAE_REMOTE_NAME=
NZAE_REGISTER_NZAE_HOST_ONLY_NZAE_REMOTE_NAME_DATA_SLICE=0
NZAE_REGISTER_NZAE_HOST_ONLY_NZAE_REMOTE_NAME_SESSION=0
NZAE_REGISTER_NZAE_HOST_ONLY_NZAE_REMOTE_NAME_TRANSACTION=0
NZAE_REGISTER_NZAE_LOG_IDENTIFIER=applyOperationV2Tf(varargs)
NZAE_REGISTER_NZAE_NUMBER_PARAMETERS=1
NZAE_REGISTER_NZAE_PARAMETER1=org.netezza.education.ApplyDriverV2
NZAE_REGISTER_NZAE_PREPEND_CLASSPATH=
/nz/export/ae/applications/dev/admin/java/:
/nz/export/ae/applications/dev/admin/java:
/nz/export/ae/adapters/java/3/sys/nz/java/nzae.jar
NZAE_REGISTER_NZAE_PREPEND_PATH=
/nz/export/ae/languages/java/6.13.0/jdk1.6.0_13/bin
NZAE_REGISTER_NZAE_SPU_ONLY_NZAE_DEBUG=1
NZAE_REGISTER_NZAE_SPU_ONLY_NZAE_PREPEND_LD_LIBRARY_PATH=
/nz/export/ae/adapters/java/3/sys/nz/lib/spu10
NZAE_REGISTER_NZAE_SPU_ONLY_NZAE_REMOTE =0
NZAE_REGISTER_NZAE_SPU_ONLY_NZAE_REMOTE_LAUNCH_VERBOSE=0
NZAE_REGISTER_NZAE_SPU_ONLY_NZAE_REMOTE_NAME=
NZAE_REGISTER_NZAE_SPU_ONLY_NZAE_REMOTE_NAME_DATA_SLICE=0
NZAE_REGISTER_NZAE_SPU_ONLY_NZAE_REMOTE_NAME_SESSION=0

NZAE_REGISTER_NZAE_SPU_ONLY_NZAE_REMOTE_NAME_TRANSACTION=0
```

The prefix NZAE_REGISTER is reserved for internal use by **register_ae**. At runtime this prefix is removed. For example, NZAE_REGISTER_NZAE_NUMBER_PARAMETERS=1 becomes NZAE_NUMBER_PARAMETERS=1.

If in this case NZAE_NUMBER_PARAMETERS was previously defined, the NZAE_REGISTER version is discarded. Thus, an AE environment variable that starts with NZAE_REGISTER can be overridden using the **register_ae --environment** option. As an example, the Java registration above is changed to add a property and a command line argument to the main Java class:

```
register_ae
--language java
--template udtf
--version 3  --db dev
--sig "applyOperationV2Tf(varargs)"
--return "table(result double)"
--define java_class=org.netezza.education.ApplyDriverV2 --
level 1
--environment "'APPLY_FUNCTION'='OPERATION'"
--environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"
--environment "'NZAE_NUMBER_PARAMETERS'='3'"
--environment "'NZAE_PARAMETER1'='-Dmyprop=x'"
```

```
--environment "'NZAE_PARAMETER2'='org.netezza.education.ApplyDriverV2'"
--environment "'NZAE_PARAMETER3'='my parameter'"
```

Note: The **--environment** option overrides are not prefixed with NZAE_REGISTER.

You can use the **register_ae** ability to translate options to system AE environment variables and NPS system function settings when using AE in advanced methods and for diagnosing problems.

Unregistering Environment Variables Set by register_ae

If a registration variable set by **register_ae** must be removed so that it is no longer used by the AE, use the NZAE_UNREGISTER_ prefix For example, to remove NZAE_REGISTER_NZAE_NUMBER_PARAMETERS=0:

```
--environment "'NZAE_UNREGISTER_NZAE_NUMBER_PARAMETERS '='any value'"
```

Note: In the example, the value is irrelevant (and not required), since the variable is being removed.

Scalar Function Options

The following options are available for scalar functions only.

--nondet

Sets that this scalar function is non-deterministic. In other words, it negates the default optimization, where the NPS system assumes that for the same input a scalar function always returns the same output. In this case, the scalar function may only be invoked once for constant input instead of once per input row. For example:

```
--nondet
```

--nullcall

Negates the default NULL call behavior, allowing a scalar function that receives all NULL input to return non-NULL output. By default, the NPS system assumes that for any NULL input a scalar function always returns NULL output and thus does not invoke the scalar function. For example:

```
--nullcall
```

Table Function Options

The following option is available for table functions only.

--noparallel

Runs all query data single-streamed through a single AE, effectively causing the AE to run on the host while processing all data from the dataslices. For example:

```
--noparallel
```

Aggregate Function Options

The following options are available for aggregate functions only.

--state

Specifies the signature of state variables. (For more information on state variables, refer to the NPS guide, *IBM Netezza User-Defined Functions Developer's Guide*.) For example:

```
--state "(double)"
```

--aggtype { ANY | ANALYTIC | GROUPED}

Specifies the aggregate type. (For more information on aggregate types, refer to the NPS guide, *IBM Netezza User-Defined Functions Developer's Guide*.) For example:

```
--aggtype ANY
```

NPS System Shared Library Dependencies

An AE can declare dependencies on one or more of the NPS system shared libraries. (For information on shared libraries, see CREATE [OR REPLACE] LIBRARY in the *Netezza User-Defined Functions Developer's Guide*. Also, see Accessing the Local Drive Space on the Machine.) While shared libraries for AEs may be actual Linux object lib*.so files, they can be any file type.

--deps <lib,lib,...lib>

Declares shared library dependencies. Library names must be comma-separated with no spaces. For example:

```
--deps mylib1,mylib2,mylib3
```

Dynamic AE Environment Variables

This section describes options specific to dynamic AE environment variables.

--dynamic {0 | 2}

Sets the AE dynamic environment level. The default value is 0; a value of 2 activates arguments based on dynamic AE environment variables. (Refer to Setting Dynamic AE Environment Variables for more information.) For example:

```
--dynamic 2
```

Optimization

This section describes options that provide information to NPS that can then be used in planning and scheduling AEs.

--mem <value>[b | k | m | g]

Specifies the amount of memory the AE is expected to use. Memory is typically entered as a number and, optionally, a unit. Valid units are **b** (bytes), **k** (kilobytes), **m** (megabytes), **g** (gigabytes). The default unit is bytes. The NPS appliance uses this value for planning and scheduling queries. Examples:

Example

```
--mem 100
--mem 1m
--mem 100k
```

Complete List of register_ae Options

This section provides a complete list of options for **register_ae**. The syntax for option usage options is:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae [OPTION]...
```

To list the options:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --help

General options:
        --echo                  Echo substituted template before
it
                                running as well as SQL output

        --file file             Use specified template file instead of
                                installed template
        --base base             Base directory for NPS instead of env
                                NZ_KIT_DIR or /nz/kit. Also available
                                as template variable ae_kit.
        --ldir base             Base directory for Language
configurations
                                instead of /nz/export/ae/adapters

        --language lang         Specify language configuration to use.
                                Languages are cpp, fortran, java, perl,
                                python64, python3x, r, system available
                                as template variable
ae_language.

        --version ver           Specify language version to use. Also
                                available as template variable
ae_lang_ver.

        --template name         Specify the template to use. Common
are
                                ones udf,uda, udtf

        --user username         NPS username (defaults to env: NZ_USER)
                                Also available as template variable
ae_user.

        --pw password           NPS password (defaults to env:
NZ_PASSWORD)
                                Also available as template
                                variable ae_password.
        --db database           NPS database (defaults to env:
NZ_DATABASE)
                                Also available as template variable
ae_db.

        -h, --help              Show this command usage.
General Template options:
        Predefined template variables include:
                ae_export_dir           Location of shared export
directory
                ae_sysroot_spu          Location of system root on spu
                ae_sysroot_host         Location of system root on host
                ae_library_path_spu     LD_LIBRARY_PATH for spu
                ae_library_path_host    LD_LIBRARY_PATH for host
        --define var=value      Define template variable

        --path dir              Location where AE files are stored.
                                Defaults to
                            <ae_export_dir>/ae/applications/<ae_db>/<ae_
```

```
user>
                                  Available as template variable
                                  ae_application_path
        --warn                    Warn instead of error if --path does not
                                  start with
                                  <ae_export_dir>/ae/applications/<ae_db>
        --environment val         Environment entry.
                                  ie --environment "'name' = 'value'".
                                  Can be specified multiple times.
                                  Available as template variable
                                  ae_environment_entries (a comma
separated
                                  list)
        --deps libs               Library dependencies. Libs must be
comma
                                  separated. Available as template
variable.
                                  ae_dependencies
        --level level             Debug level for AE (default 0).
Available as
                                  template variable ae_debug_level.
        --idir dir                Install dir for the AE language.
Available as
                                  template variable ae_install_dir.
        --udxtype type            UDX type. Must be one of udf, udtf or
uda.
                                  Available as template variable ae_type.
        --sig signature           Argument signature for function or
aggregate.                        UDXname(arg1, arg2, ...) Will need to be
                                  quoted to stop the shell from
interpreting
                                  the ()'s. Available as template variable
                                  ae_sig.
        --return return           Return type for function or aggregate
                                  For table function, will need to be
quoted
                                  to stop the shell from interpreting the
()'s.
                                  Available as template variable
ae_return.
        --class class             Class name for function or aggregate.
                                  Available as template variable ae_class.
        --mask args               Can be DEBUG or TRACE. Can be
                                  specified multiple times. Available
                                  as template variable ae_mask.
        --mem num                 Memory must be a number and units.
                                  A size can be b (bytes), k (kilobytes),
                                  m (megabytes), g (gigabytes) or can
                                  be empty. ie 100, 1m, 100k
```

```
                                Available as template variable ae_mem.

        --remote                AE remote mode enabled. Available as
                                template variable ae_remote.

        --launch                AE remote launch enabled. Available
                                as template variable ae_launch.

        --dynamic num           Set AE dynamic environment level to
                                Default is 0. Available as template
                                variable ae_dynamic.

        --rtrans                AE remote name transaction enabled.
Available
                                as template variable
ae_remote_transaction.

        --rsession              AE remote name session enabled.
Available
                                as template variable ae_remote_session.

        --rdataslice            AE remote name dataslice enabled.
Available
                                as template variable
ae_remote_dataslice.

        --rname name            Set AE remote name to name. Available
                                as template variable ae_remote_name.
        --exe name              Set AE executable name to name.
Available as
                                template variable ae_exe_name.

Scalar Function Template options:
        --nondet                Non deterministic. Default is
deterministic.
                                Available as template variable
                                ae_deterministic.
        --nullcall              CALLED ON NULL INPUT. Default is
                                RETURNS NULL ON NULL INPUT. Available
                                as template variable ae_nullcall.
Table Function Template options:
        --noparallel            Table function will be created with
                                parallel not allowed. Default is
parallel
                                allowed. Available as template variable
                                ae_parallel.
        --lastcall args         Table function will be created with
                                <args> ALLOWED. Possible values are
'TABLE',                        'TABLE FINAL' or 'TABLE, TABLE FINAL'
                                Default is 'TABLE, TABLE FINAL.'
                                Available as template variable
ae_lastcall.
Aggregate Template options:
        --state state           State signature for aggregate. (state1,
                                state2, ...). Will need to be quoted to
```

```
stop
                                    the shell from interpreting the ()'s.
                                    Available as template variable ae_state.

        --aggtype aggtype           Specify agg type. Can be ANY, ANALYTIC
                                    or GROUPED. Available as template
variable
                                    ae_uda_type.
```

R Language Output Modes

The UDX output signature is the definition of a given function or aggregate result; it can be a scalar (UDF/UDA) or a table (UDTF).

The output signature must be known prior to a UDX invocation, which might differ from what typically takes place in R. A typical R use case is the apply function that accepts a data.frame object, a margin specifying the order of applying, and the function that is to be applied. The output can be a vector, a matrix, or a list, depending on the actual output of the user-provided function.

```
apply(iris, 1, function(x) length(x))
apply(iris[,1:4], 1, function(x) c(length(x),sqrt(as.double(x))))
```

In the R Adapter, the output mode is controlled by the OUTPUT_TYPE environment variable that you should set during registration.

To set this variable, add the following lines to the command line of the *register_ae* script:

For sparse mode:
--define "r_ae_output_type=SPARSE"

For table mode:
--define "r_ae_output_type=TABLE"

Sparse Output Mode

Generally, the output of the user-provided function cannot be restricted to any predefined form. In the sparse output mode, the R AE returns a table of the definition TABLE(columnid INT4, value VARCHAR(16000)), which means that each R AE output column is converted to a character string. If the original value is to be retrieved, you must cast each value to the desired data type manually. You should, however, avoid this practice because it might cause additional rounding errors and affect performance, especially for large data sets.

In this mode, there is no difference between returning output data with a general setOutput function and specific setOutput<*DataType*> functions, where <*DataType*> is a placeholder for a specific data type identifier. All output data is eventually stored as a character string.

Table Output Mode

The alternative table output mode requires one of the following settings for the output signature that is provided during the registration step:

Exact setting
This setting specifies columns with their data types.

Set to TABLE(ANY)
For this setting, you must define a shaper function that specifies the output signature at run time.

It the table output mode, you should avoid the setOutput function. Instead, you should use specific setOutput<**DataType**> functions, where <*DataType*> is a placeholder for a specific data type identifier.

Accessing the Local Drive Space on the Machine

An AE can access the NPS shared library functionality. The NPS system shared libraries, originally designed to provide shared library support (lib*.so files) for UDXs, provide the same support for AE applications. However, the functionality has been extended so that shared libraries can be any type of file, including executable binaries and data files. These files are read and stored in the core database. When a shared library is registered and stored in the database, the user specifies a symbolic name for the library.

As part of registration, AEs can declare dependencies on a shared library, and at query time, NPS ensures that a local copy is available on the host and the SPUs, which means that the library is not located on the shared network drive containing the AE export directory tree. Using this approach, it is possible to design AEs written in languages such as C or C++ so that they are self-contained as UDXs. This self-contained approach is only practical, however, for applications that require a relatively small number of support files.

Finally, AEs have access to a certain amount of temporary local drive space on the host and SPU. This space is also used by the NPS system database and is a limited resource. Since each installation is different, a specific maximum disk space usage recommendation is impossible, but excessive use of local disk space can impact the performance of the NPS system database. Therefore, "very large" temporary files should be placed in the AE export directory tree. Using local disk space such as the temporary space directory, or using networked disk space like the AE export directory tree, has performance implications on the NPS system.

CHAPTER 3

Running Function Analytic Executables

Calling an AE from SQL

This section describes the three types of SQL Functions that can be used to call an Analytic Executable (AE)--scalar, table, and aggregate.

For more about SQL Functions, see:

Netezza Database User's Guide

*Netezza User-Defined Functions Developer's Guid*e

Calling a Scalar Function

A scalar function can be used wherever normal SQL scalar functions appear, such as NPS built-in scalar functions. It returns one value per input row. The following is an example of a scalar function:

```
SELECT applyOperationV1Sf('+', 1, 2, 3);

APPLYOPERATIONV1SF
--------------------
                   6

SELECT f1, f2, f3, f4, applyOperationV1Sf('+', f1, f2, f3, f4)
    FROM edutestdata
    WHERE color = 'red';

   F1 |   F2   |    F3 |  F4     | APPLYOPERATIONV1SF
      +        +       +         +
------1|-------2|------3|--------4.000| --------------------10
  100  |  300   |   0.5 |  0.500  |                    401
 -100  |  300   |  0.5  |  0.500  |                    201
  100  | -300   |  0.5  |  0.500  |                   -199
```

Calling a Table Function

A table function can be used anywhere SQL allows a table. The table function returns a table that may contain a number of rows less than, equal to, or greater than the number of input rows. AEs

require that the **WITH FINAL** clause be used with a table function. Following is the syntax for calling a table function:

```
TABLE WITH FINAL(function_name(arg1, arg2, …))
```

The following are two examples of calling a table function:

```
SELECT * FROM TABLE WITH FINAL(applyOperationV1Tf('+', 1, 2,

3)); RESULT

--------

     6
```

```
SELECT f1, f2, f3, f4, result
  FROM edutestdata,
  TABLE WITH FINAL(applyOperationV1Tf('+', f1, f2, f3, f4))
  WHERE color = 'red';
```

```
RESULT
    F1 |   F2  |  F3|     F4 | RESULT
------+------+-----+-------+--------
     1 |    2  |   3| 4.000 |      10
   100 |  300  | 0.5| 0.500  |    401
  -100 |  300  | 0.5| 0.500  |    201
   100 | -300  | 0.5| 0.500  |   -199
```

Calling an Aggregate Function

An aggregate function can be used almost anywhere SQL allows an aggregate function. An aggregate function returns a single value per group or window. Grand aggregates return a single result for all the data. The following is an example of using an aggregate function:

```
SELECT color, applyAggSum(f1, f2, f3, f4) FROM edutestdata GROUP BY color.

RESULT

  COLOR | APPLYAGGSUM
--------+
   blue |------------413
  green |          413
    red |          413
 yellow |          413
```

Using AEs to Exceed NPS SQL Function Argument Limits

On the NPS, SQL functions are limited to 64 arguments. AEs provide a way to exceed this limit, using an installed scalar function named NZAEMULTIARG. This function takes up to 64 arguments and returns a varchar string containing a compressed serialization of all the arguments. The AE runtime can deserialize these compressed arguments and make them appear to an AE as standard arguments. NZAEMULTIARG calls can be also be nested so that an AE can receive a significantly larger number of arguments. The combined size of all the arguments is limited by the NPS system to 64K bytes. NZAEMULTIARG can also be used with UDXs.

Example

```
SELECT * FROM DATA AS t, TABLE WITH
FINAL(my_table_function_ae(NZAEMULTIARG(t.f1, t.f2, t.f3), 100, 200,
```

```
NZAEMULTIARG('a', 'b', 'c', 'd'), 100.6));
```

This AE receives normal arguments using NZAEMULTIARG:

```
t.f1, f.f2, t.f3, 100, 200, 'a', 'b', 'c', 'd', 100.6
```

In this example, my_table_function_ae is registered with argument signature VARARGS, which is the keyword for any number and type of arguments.

NZAEMULTIARG may not work well with shapers since the output of NZAEMULTIARG is never a literal and shapers only receive literals. However, a combination of NZAEMULTIARG and literal arguments can be used to work around this limitation. (For more information on shapers, see Introduction to Shapers and Sizers.)

Example

```
myfunction(NZAEMULTIARG(f1, 'abc'), 'constant for Shaper');
```

Understanding Locus Control and Analytic Executables

AEs can run on either the SPUs or the host. It is best to run an operation on multiple SPUs versus a single host so that portions of the operation can be run concurrently, leveraging the Netezza appliance's massively parallel architecture. There are situations, however, where running an operation only on the host are desirable. For example:

- When you must see all of the data from all of the dataslices, you can use a table function AE to perform an aggregation-type operation. In a single query, the output of AEs that run on the SPUs becomes the input of another single AE run on the host. The AE on the host performs an aggregation operation and returns a result for the query.
- Debugging on the host is much easier than debugging on the SPUs.

AEs are called from a database system that performs query optimization. Sometimes the database optimizer chooses to run a function in a particular locus (SPUs or host), which means the AE runs in that locus as well. AEs and UDXs are subject to the same locus rules, but for AEs, the outcome can be more problematic, especially for remote AEs. AEs can be used as database extensions, but can also be used as a mechanism for concurrent processing because they run only on the SPUs. If a remote AE is running only on the SPUs and a query or a portion of a query unexpectedly runs on the host, then that query hangs. Therefore, it is useful to understand locus control before you write an AE.

The following sections describe the locus behavior of the scalar, table, and aggregate SQL functions.

Scalar Functions

A scalar function that receives arguments that are all literals in a query with no user tables runs on the host. A scalar function that is deterministic and takes all literals runs on the host. Otherwise, the function may run on the host or SPUs, depending on the decision of the query optimizer. The sizer, however, always runs on the host even if the scalar function logic runs on the SPU. (For more information on sizers, see Introduction to Shapers and Sizers.)

Table Functions

A table function that receives arguments that are all literals runs on the host. A table function registered as **--noparallel** runs on the host. All other table functions typically run on the SPUs, but there are some situations where the NPS Query Optimizer moves processing to the host.

Your choice between host and SPU (whether or not to use --noparallel) for table functions has implications.

a table function AE returning data at the end of the input may return more rows when running on the SPU than on the host, as it executes *n* times for *n* dataslices instead of a single time on the host

if the input is on the host (external table, _v_dual, or a temporary table in the plan), and the optimizer chooses to evaluate it on the SPU, it distributes the data to the SPUs using random distribution.

Shapers always run on the host, even if the table function logic runs on the SPU. (For more information on shapers, see Introduction to Shapers and Sizers.)

Aggregate Functions

An aggregate function that receives arguments that are all literals in a query with no user tables runs on the host. A grand aggregate with normal tables runs on both host and SPU. A grouped aggregate with normal tables runs on both host and SPUs or only on the SPUs. A windowed aggregate with no partition clause runs on the host; in all other cases, it runs on the SPUs.

Introduction to Shapers and Sizers

Often when dealing with functions, you want the shape or size of the output to be dependent on the input. Shapers and sizers enable you to specify what the AE is going to return (that is, the output schema).

Shaping and sizing are both implemented using the same AE API, however the options are much more limited for sizing.

Shapers

Shapers used by AEs are called from table functions to dynamically declare the return table metadata, specifically the number of columns, the column names, types, and precision/scale. Shapers are invoked when a table function is declared as returning TABLE(ANY).

For local AEs, a new instance of the AE process is created on the host to return metadata describing the return table. The new instance is called only to provide this information and is not used for processing data. The AE instance is called before any data is retrieved; it receives metadata about the input arguments and can also retrieve any arguments that are constant. For remote AEs, a notification is sent to an instance of the remote AE running on the host. If no remote AE instance exists on the host with an address that matches the SQL table function, the SQL table function

returns an error.

If an AE is registered as a SQL table function that is declared with return value TABLE(ANY), the AE must be written to perform both "shaping" and data processing. The same process instance never performs both, except in the case of a remote AE. Each AE language has an API function that indicates whether the AE is being used as a shaper or as a normal data processor.

Sizers

Sizers used by AEs are called from scalar functions to dynamically declare the return size of strings and numeric values. For sizing to occur, the SQL scalar function is declared with return value CHAR(ANY), VARCHAR(ANY), NCHAR(ANY), or NVARCHAR(ANY) for strings and NUMERIC(ANY) for numeric values. As with shapers, for local AEs an instance for sizing is created on the host, and for remote AEs, a notification is made on the host. The instance is called only to provide this sizing information and is not used for processing data. The AE instance is called before any data is retrieved; it receives metadata about the input arguments and can also retrieve any arguments that are constant integers of type int4 only (not int1, int2, or int8).

Sizers are more limited than shapers, allowing only specification of the size of the string output or the precision and scale of a numeric output; the type is specified in the registration process. Sizers allow access only to literal int32 fields within the input data.

If an AE is registered as a SQL scalar function, which is declared with string return value CHAR(ANY) or NUMERIC(ANY), the AE must be written to perform both "sizing" and data processing. The same process instance never performs both, except in the case of a remote AE. Each AE language has an API function that indicates whether the AE is being used as a sizer or as a normal data processor.

Introduction to AE Language APIs

Each supported programming language has its own specific API library. Although each API library is different, they follow similar patterns and contain similar API groupings. These groups consist of functions, methods, or classes, depending on the programming language. This section provides a conceptual overview of the programming language APIs and offers a starting point for understanding the examples and using the language-specific API references.

Low-level API

The low-level API consists of a number of groups of functions responsible for:

handling the connection between the NPS and AEs

handling the data flow, including data input and output, type and values conversion, etc.

querying the process environment variables exported by the operating system as well as the NPS

defining the output signature, known as “shapers” and “sizers”

iterating through shared libraries associated with the AE via the NPS shared library mechanism.

general usability functions, including error reporting and gathering Netezza-related runtime

information :

Initialization API

The initializations APIs consist of the functions or classes used to create a data connection. These connections are required before data can move between an AE and the database.

Except for the Language Development Kit (LDK), which is a C-language API, each language supports initialization high-level functionality to create a data connection. They are:

C++: class NzaeApiGenerator

Java: class NzaeApiGenerator

Fortran: subroutine nzaeRun()

Python: derive from class nzae.Ae

Perl: derive from class nzae.Ae

R: function determineAEMode()

The C API contains initialization functions that operate on a lower level. Some languages, such as C++ and Java, also support these lower-level API features. When starting out with Aes (other than with the LDK), use the high-level process to create a connection. The lower-level API is most useful for remote AE processes not launched by NPS.

Each of the language reference guides includes a section or page describing initialization. When using the higher level initialization, most of this documentation is not needed.

The Table Function (Simulated Row Function) Examples and, in Simple Remote Mode Examples, Scalar Function (Remote Mode)" examples for the various supported languages show initialization in full, typically supporting both local and remote AE. In these examples, for the LDK (C API), low level initialization is shown. For the other languages, high-level initialization is shown.

Data Connections

The result of initialization is a data connection corresponding to one of the three AE data connection types: function, aggregate, and shaper/sizer. Each language reference guide contains sections corresponding to these three data connection APIs.

The data connection APIs are used to receive and return data with the database. The AE function API is used with SQL Table and Scalar Functions, the AE Aggregation API is used with SQL Aggregate Functions, and the AE Shaper/Sizer API is used with SQL Table-Function-Shapers and SQL Scalar-Function-Sizers.

The examples in Scalar Function Examples, Simple Table Function Examples, and Table Function (Simulated Row Function) Examples all demonstrate the AE Function API. The Shapers and Sizers Examples demonstrate the AE Shaper/Sizer API. The Aggregate AE Examples examples demonstrate the AE Aggregate API.

Record and Data Type Support

Each AE language API implementation contains common data structures used for all three of the data connection APIs. For example, C++ and Java have an abstraction for input and output rows, a record class containing columns called fields. The class for both is named NzaeRecord. The Python and Perl AE API abstract input columns as a list. The R AE API casts the database data types from the database to be R-specific. Languages such as C and Fortran treat columns individually using index function arguments and do not have an explicit record abstraction. Each language library has an API for the NPS data types it supports.

All of the examples demonstrate data type support.

Advanced NPS Features

The programming language APIs contain functionality for working with advanced NPS features such as AE/UDX Environment Variables and Shared Libraries, as demonstrated in the "Environment & Shared Libraries" examples.

The API Roadmap

An AE first uses the initialization API to get a data connection. It then uses the API that corresponds to the connection type, either Function, Aggregate, or Shaper/Sizer. The AE uses the data type support API to process column and state values.

Study the language examples in this guide to understand the basic structure of an AE. Further information about the functions, methods, and classes found in the examples can be found in the language-specific API references.

CHAPTER 4
Working with Examples

The examples presented in this guide are intended to demonstrate functionality of various types of AEs in the languages supported by the Netezza software.

Assumptions for Working with Examples

Compilation and registration examples that specify the **--db** option assume that there is a valid database called “dev” on the Netezza appliance.

To take advantage of the examples as they are built on through this manual, it is important to use the correct file name. When saving the sample code, use the name supplied at the beginning of the Code section. (This is also the name used in the compile step.) The same file name will be called later in subsequent examples.

Compilation and registration examples that specify the **--user** option assume that there is a valid database user named “nz”. This user should not be confused with the UNIX username.

All examples presented in this guide assume that the reader is working directly on an IBM Netezza host. Invoking scripts such as **register_ae**, editing source files, and other activities must be performed on the Netezza appliance. Some of the presented commands may not exist on the local desktop machine.

Editing Files

When working directly on the Netezza appliance, the nzPlug-in for Eclipse IDE cannot be used for editing files. If files must be edited, use one of the editors available in the standard Linux distribution (e.g., vi) or edit the source files locally and then transfer them to the appliance using a utility such as scp, sftp, or Eclipse.

CHAPTER 5

Scalar Function Examples

The concepts covered in this section include:

- Scalar functions
- Error handling
- Local mode

C Language Scalar Function

The following example shows a simple scalar function that sums a set of numbers. This example starts from the beginning to construct a simple AE. It uses the following file name:

```
func.c
```

Code

Pull in the C interface. This can typically be accomplished using:

```
#include "nzaeapis.h"
```

Write a main. Determine an API to use, in this case a function. Select either a local AE connection or a remote AE connection. This example uses a local AE connection:

```
#include <stdio.h>
#include <stdlib.h>

#include "nzaeapis.h"

int main(int argc, char * argv[])
{
    if (nzaeIsLocal())
    {
        NzaeInitialization arg;
        memset(&arg, 0, sizeof(arg));
        arg.ldkVersion = NZAE_LDK_VERSION;
        if (nzaeInitialize(&arg))
        {
            fprintf(stderr, "initialization failed\n");
            return -1;
```

```
        }
        nzaeClose(arg.handle);
    }
    else
    {
        fprintf(stderr, "Expecting Local AE\n");
        return -1;
    }
    return 0;
}
```

This sets up nzae, which enables the ability to work with AE functions.

Add a stub for calling the function logic:

```
#include <stdio.h>
#include <stdlib.h>

#include "nzaeapis.h"
static int run(NZAE_HANDLE h);
int main(int argc, char * argv[])
{
    if (nzaeIsLocal())
    {
        NzaeInitialization arg;
        memset(&arg, 0, sizeof(arg));
        arg.ldkVersion = NZAE_LDK_VERSION;
        if (nzaeInitialize(&arg))
        {
            fprintf(stderr, "initialization failed\n");
            return -1;
        }
                run(arg.handle);
        nzaeClose(arg.handle);
    }
    else
    {
        fprintf(stderr, "Expecting Local AE\n");
        return -1;
    }
    return 0;
}

static int run(NZAE_HANDLE h)
{

    return 0;
}
```

Implement the full logic:

```
#include <stdio.h>
#include <stdlib.h>

#include "nzaeapis.h"

static int run(NZAE_HANDLE h);
int main(int argc, char * argv[])
{
    if (nzaeIsLocal())
    {
        NzaeInitialization arg;
        memset(&arg, 0, sizeof(arg));
                arg.ldkVersion = NZAE_LDK_VERSION;
        if (nzaeInitialize(&arg))
        {
```

```
            fprintf(stderr, "initialization failed\n");
            return -1;
        }
        run(arg.handle);
        nzaeClose(arg.handle);
    }
    else
    {
        fprintf(stderr, "Expecting Local AE\n");
        return -1;
    }
    return 0;
}

static int run(NZAE_HANDLE h)
{
    NzaeMetadata metadata;
    if (nzaeGetMetadata(h, &metadata))
    {
        fprintf(stderr, "get metadata failed\n");
        return -1;
    }

    #define CHECK(value) \
    { \
        NzaeRcCode rc = value; \
        if (rc) \
        { \
            const char * format = "%s in %s at %d"; \
            fprintf(stderr, format,  \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            nzaeUserError(h, format, \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            exit(-1); \
        } \
    }

    if (metadata.outputColumnCount != 1 ||
        metadata.outputTypes[0] != NZUDSUDX_DOUBLE)
    {
        nzaeUserError(h, "expecting one output column of type
        double"); return -1;
    }

    if (metadata.inputColumnCount < 1)
    {
        nzaeUserError(h, "expecting at least one input column");
        return -1;
    }

    if (metadata.inputTypes[0] != NZUDSUDX_FIXED
        && metadata.inputTypes[0] != NZUDSUDX_VARIABLE)
    {
        nzaeUserError(h, "first input column expected to be a string
        type"); return -1;
    }

    for (;;)
    {
        int i;
        double result = 0;
        NzaeRcCode rc = nzaeGetNext(h);
        if (rc == NZAE_RC_END)
        {
            break;
        }

        NzudsData * input = NULL;
```

```
CHECK(nzaeGetInputColumn(h, 0, &input));
const char * opString;
if (input->isNull)
{
    nzaeUserError(h, "first input column may not be
    null"); return -1;
}
if (input->type == NZUDSUDX_FIXED)
{
    opString = input->data.pFixedString;
} else if (input->type == NZUDSUDX_VARIABLE)
{
    opString = input-
>data.pVariableString; } else
{
    nzaeUserError(h, "first input column expected to be a string type");
    return -1;
}

enum OperatorEnum
{
    OP_ADD = 1,
    OP_MULT = 2
} op;

if (strcmp(opString, "+") == 0)
{
    op = OP_ADD;
} else if (strcmp(opString, "*") == 0)
{
    op = OP_MULT;
    result = 1;
} else
{
    nzaeUserError(h, "unexpected operator %s", opString);
    return -1;
}

for (i = 1; i < metadata.inputColumnCount; i++)
{
    CHECK(nzaeGetInputColumn(h, i, &input));
    if (input->isNull)
    {
        continue;
    }
    switch (op)
    {
    case OP_ADD:
        switch(input->type)
        {
            case NZUDSUDX_INT8:
                result += *input->data.pInt8;
                break;
            case NZUDSUDX_INT16:
                result += *input->data.pInt16;
                break;
            case NZUDSUDX_INT32:
                result += *input->data.pInt32;
                break;
            case NZUDSUDX_INT64:
                result += *input->data.pInt64;
                break;
            case NZUDSUDX_FLOAT:
                result += *input->data.pFloat;
                break;
            case NZUDSUDX_DOUBLE:
                result += *input->data.pDouble;
                break;
```

```
                        case NZUDSUDX_NUMERIC32:
                        case NZUDSUDX_NUMERIC64:
                        case NZUDSUDX_NUMERIC128:
                               unlike Java, C has no native numeric data
                            types break;
                        default:
                               ignore nonnumeric
                            type break;
                    }
                    break;
                case OP_MULT:
                    switch(input->type)
                    {
                        case NZUDSUDX_INT8:
                            result *= *input->data.pInt8;
                            break;
                        case NZUDSUDX_INT16:
                            result *= *input->data.pInt16;
                            break;
                        case NZUDSUDX_INT32:
                            result *= *input->data.pInt32;
                            break;
                        case NZUDSUDX_INT64:
                            result *= *input->data.pInt64;
                            break;
                        case NZUDSUDX_FLOAT:
                            result *= *input->data.pFloat;
                            break;
                        case NZUDSUDX_DOUBLE:
                            result *= *input->data.pDouble;
                            break;

                        case NZUDSUDX_NUMERIC32:
                        case NZUDSUDX_NUMERIC64:
                        case NZUDSUDX_NUMERIC128:
                               unlike Java, C has no native numeric data
                            types break;
                        default:
                               ignore non-numeric type
                            break;
                    }
                    break;
                default:
                    nzaeUserError(h, "internal error, unexpected operator");
                    return -1;
                }
            }

            CHECK(nzaeSetOutputDouble(h, 0, result));
            CHECK(nzaeOutputResult(h));
        }
        nzaeDone(h);
        return 0;
}
```

Once the code is complete, it must be compiled and registered.

Compilation

To compile, use the following:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language system --version 3 \
     --template compile func.c --exe apply
```

The arguments specify the system language—C—using version 3 with the template compile. Additionally, they specify that the process should produce an executable apply.

Registration

Once compiled, register the code as follows:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3 \
     --template udf --exe apply --sig "applyop_c(VARARGS)" --return "double"
```

Running

You can now run the AE in SQL:

```
SELECT applyop_c('+',1,2);
 APPLYOP_C
-----------
         3
(1 row)
```

Notice in the code that you validate the operator and use nzaeUserError if it is incorrect. The following example triggers an error:

```
SELECT applyop_c('-',1,2);
ERROR:  unexpected operator -
```

C++ Language Scalar Function

The following example shows a simple scalar function that sums a set of numbers. This example starts from the beginning to construct a simple AE. It uses the following file name:

```
applyopcpp.cpp
```

Code

Pull in the C++ interface. This can typically be accomplished using:

```
#include <nzaefactory.h>
```

Write a main. Determine an API to use, in this case a function. This can be achieved either by getting a local AE connection or a remote AE connection. This example uses a local AE connection. Two interfaces can be used. One uses NzaeFactory and the other uses NzaeApiGenerator. In this example, use the generator class as it is simpler and can also be used for remote mode.

The program should appear similar to:

```
include <nzaefactory.hpp>
using namespace nz::ae;
int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
    NzaeApi &api = helper.getApi(NzaeApi::FUNCTION);
    return 0;
}
```

This retrieves an NzaeApi reference that contains a usable NzaeFunction object.

Now that the function is available, add a stub for calling the function logic:

```
#include <nzaefactory.hpp>
using namespace nz::ae;

static int run(nz::ae::NzaeFunction *aeFunc);

int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
    NzaeApi &api = helper.getApi(NzaeApi::FUNCTION);
    run(api.aeFunction);
    return 0;
}

static int run(NzaeFunction *aeFunc)
{
    return 0;
}
```

Implement the function.

There are two ways of implementing a function. Either use the NzaeFunction object interface directly or implement an NzaeFunctionMessageHandler-derived class that provides a simpler interface. In this example, the message handler is used. The second method is covered in a separate example. The message handler may only be used for functions that return one row of output per input. Also, the handler covers some error handling details automatically.

```
#include <nzaefactory.hpp>

using namespace nz::ae;

static int run(nz::ae::NzaeFunction *aeFunc);

int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
    NzaeApi &api = helper.getApi(NzaeApi::FUNCTION);
    run(api.aeFunction);
    return 0;
}

class MyHandler : public NzaeFunctionMessageHandler
{
public:

    enum OperatorEnum
    {
        OP_ADD = 1,
        OP_MULT = 2
    } op;


    void evaluate(NzaeFunction& api, NzaeRecord &input, NzaeRecord &result)
        { double res = 0;
        NzaeField &field = input.get(0);
        if (field.isNull())
            throw NzaeException("first input column may not be null");
        NzaeStringField &sf = (NzaeStringField&) input.get(0);
        std::string strop = (std::string)sf;
        if (strop == "+")
            op = OP_ADD;
        else if (strop == "*") {
            op = OP_MULT;
            res = 1;
```

```
            }
            else
                throw NzaeException(NzaeException::format("unexpected
                     \ operator %s", strop.c_str()));
            for (int i=1 ; i < input.numFields(); i++) {
                NzaeField &inf = input.get(i);
                if (inf.isNull())
                    continue;
                double temp = 0;
                bool ignore = false;
                switch (inf.type()) {

                case NzaeDataTypes::NZUDSUDX_INT32:
                {
                   NzaeInt32Field &sf = (NzaeInt32Field&)input.get(i);
                   temp = (int32_t)sf;
                   break;
                }
                default:
                    ignore = true;
                       ignore
                    break;
                }
                if (!ignore) {
                    if (op == OP_ADD)
                        res += temp;
                    else
                        res *= temp;
                }
            }

            NzaeDoubleField &f =
            (NzaeDoubleField&)result.get(0); f = res;
        }
    };

    static int run(NzaeFunction *aeFunc)
    {

        aeFunc->run(new MyHandler());
        return 0;
    }
```

In this example, the run function instantiates the handler object, and then invokes the run method of the function class on the handler. The handler must implement an evaluate method, which takes a reference to the function as well as references to the input record and the output record. For normal functions there is a single column in the output record. The evaluate can then retrieve fields from the input record and use that to set the output field. This code provides a basic, non-error-checking function that should take as the first argument either the string **+** or the string ***** and for subsequent arguments int32s. It then either adds or multiplies the int32s, based on the specified argument value, and returns the result as a double.

The final code preparation step adds error checking and support for additional data types.

```
#include <nzaefactory.hpp>
using namespace nz::ae;

static int run(nz::ae::NzaeFunction *aeFunc);

int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
    NzaeApi &api = helper.getApi(NzaeApi::FUNCTION);
    run(api.aeFunction);
```

```
    return 0;
}

class MyHandler : public NzaeFunctionMessageHandler
{
public:
    MyHandler() {
        m_Once = false;
    }
    enum OperatorEnum
    {
        OP_ADD = 1,
        OP_MULT = 2
    } op;

    void doOnce(NzaeFunction& api) {
        const NzaeMetadata& meta = api.getMetadata();
        if (meta.getOutputColumnCount() != 1 ||
            meta.getOutputType(0) != NzaeDataTypes::NZUDSUDX_DOUBLE) {
            throw NzaeException("expecting one output column of type double");
        }
        if (meta.getInputColumnCount() < 1) {
            throw NzaeException("expecting at least one input column");
        }
        if (meta.getInputType(0) != NzaeDataTypes::NZUDSUDX_FIXED &&
            meta.getInputType(0) != NzaeDataTypes::NZUDSUDX_VARIABLE) { throw
            NzaeException("first input column expected to be a string \
                 type");
        }
        m_Once = true;

    }
    void evaluate(NzaeFunction& api, NzaeRecord &input, NzaeRecord &result) {

        if (!m_Once)
            doOnce(api);
        double res = 0;

        NzaeField &field = input.get(0);
        if (field.isNull())
            throw NzaeException("first input column may not be null");
        NzaeStringField &sf = (NzaeStringField&) input.get(0);
        std::string strop = (std::string)sf;
        if (strop == "+")
            op = OP_ADD;
        else if (strop == "*") {
            op = OP_MULT;
            res = 1;
        }
        else
            throw NzaeException(NzaeException::format("unexpected operator
                 \ %s", strop.c_str()));
        for (int i=1 ; i < input.numFields(); i++) {
            NzaeField &inf = input.get(i);
            if (inf.isNull())
                continue;
            double temp = 0;
            bool ignore = false;
            switch (inf.type()) {
                case NzaeDataTypes::NZUDSUDX_INT8:
                {
                    NzaeInt8Field &sf = (NzaeInt8Field&)input.get(i);
                    temp = (int8_t)sf;
                    break;
                }
                case NzaeDataTypes::NZUDSUDX_INT16:
                {
```

```
                        NzaeInt16Field &sf = (NzaeInt16Field&)input.get(i);
                        temp = (int16_t)sf;
                        break;
                    }
                    case NzaeDataTypes::NZUDSUDX_INT32:
                    {
                        NzaeInt32Field &sf = (NzaeInt32Field&)input.get(i);
                        temp = (int32_t)sf;
                        break;
                    }
                    case NzaeDataTypes::NZUDSUDX_INT64:
                    {
                        NzaeInt64Field &sf = (NzaeInt64Field&)input.get(i);
                        temp = (int64_t)sf;
                        break;
                    }
                    case NzaeDataTypes::NZUDSUDX_FLOAT:
                    {
                        NzaeFloatField &sf = (NzaeFloatField&)input.get(i);
                        temp = (float)sf;
                        break;
                    }
                    case NzaeDataTypes::NZUDSUDX_DOUBLE:
                    {
                        NzaeDoubleField &sf =
                        (NzaeDoubleField&)input.get(i); temp = (double)sf;
                        break;
                    }
                    case NzaeDataTypes::NZUDSUDX_NUMERIC32:
                    case NzaeDataTypes::NZUDSUDX_NUMERIC64:
                    case NzaeDataTypes::NZUDSUDX_NUMERIC128:
                    {
                        NzaeNumericField &sf =
                        (NzaeNumericField&)input.get(i); temp = (double)sf;
                        break;
                    }
                    default:
                        ignore = true;
                           ignore
                        break;
                }
                if (!ignore) {
                    if (op == OP_ADD)
                        res += temp;
                    else
                        res *= temp;
                }
            }

            NzaeDoubleField &f =
            (NzaeDoubleField&)result.get(0); f = res;


        }

        bool m_Once;

    };

    static int run(NzaeFunction *aeFunc)
    {

        aeFunc->run(new MyHandler());
        return 0;
    }
```

This code ensures the correct number and types of input arguments and the correct number and

type of output arguments. It supports all of the integer types—float, double and numeric.

Once the code is complete, it must be compiled and registered.

Compilation

To compile, use the following:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp --template compile
     \ --exe applyopcpp --compargs "-g -Wall" --linkargs "-g" applyopcpp.cpp
     \ --version 3
```

The arguments specify that you are using the cpp language, version 3, with the template compile, and creating the executable applyopcpp. Also, the compiler and linker use the provided additional argument so that the applyopcpp executable has debug symbols.

Registration

Once compiled, register the code:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "applyop_cpp(VARARGS)"
     \ --return "double" --language cpp --template udf --exe applyopcpp \
     --version 3
```

Running

You can now run the AE in SQL:

```
SELECT applyop_cpp('+',1,2);
 APPLYOP_CPP
-------------
           3
(1 row)
```

Notice in the code that you validate the types using doOnce and send an error using NzaeException. The following example triggers an error:

```
SELECT applyop_cpp('-',1,2);
ERROR:  unexpected operator -
```

Java Language Scalar Function

The following example shows a simple scalar function that sums a set of numbers. This example starts from the beginning to construct a simple AE. It uses the following file name:

```
TestJavaInterface.java
```

Code

Pull in the Java interface. This can typically be accomplished using:

```
import org.netezza.ae.*;
```

Write a main. Determine an API to use, in this case a function. Get an AE connection, either local or remote. This example uses a local AE connection. There are two interfaces that can be used to get the connection. One is using NzaeFactory and the other is using NzaeApiGenerator. This example uses the generator class as it is simpler and can be used for remote mode as well. The program should appear similar to:

```
import org.netezza.ae.*;

public class TestJavaInterface {

    public static final void main(String [] args) {
        try {
            mainImpl(args);
        } catch (Throwable t) {
            System.err.println(t.toString());
            NzaeUtil.logException(t, "main");
        }
    }

    public static final void mainImpl(String [] args) {
        NzaeApiGenerator helper = new NzaeApiGenerator();
        final NzaeApi api = helper.getApi(NzaeApi.FUNCTION);
        helper.close();
    }
}
```

This retrieves a NzaeApi reference that contains a usable NzaeFunction object.

Now that the function is available, add a stub for calling the function logic:

```
import org.netezza.ae.*;

public class TestJavaInterface {

    public static final void main(String [] args) {
        try {
            mainImpl(args);
        } catch (Throwable t) {
            System.err.println(t.toString());
            NzaeUtil.logException(t, "main");
        }
    }

    public static final void mainImpl(String [] args) {
        NzaeApiGenerator helper = new NzaeApiGenerator();
        final NzaeApi api = helper.getApi(NzaeApi.FUNCTION);
        run(api.aeFunction);
        helper.close();
    }
    public static int run(Nzae aeFunc)
    {
        return 0;
    }
}
```

There are two ways of implementing a function. One is to use the Nzae object interface directly. The other is to implement an NzaeMessageHandler-derived class that provides a simpler interface. In this example, the message handler is used; the second method is covered in a separate example. The message handler may only be used for functions that return one row of output per input. Also, the handler covers some error handling details automatically.

```
import org.netezza.ae.*;

public class TestJavaInterface {

    public static final void main(String [] args) {
        try {
            mainImpl(args);
        } catch (Throwable t) {
            System.err.println(t.toString());
            NzaeUtil.logException(t, "main");
        }
```

```
    }

    public static final void mainImpl(String [] args) {
        NzaeApiGenerator helper = new NzaeApiGenerator();
        final NzaeApi api = helper.getApi(NzaeApi.FUNCTION);
        run(api.aeFunction);
        helper.close();
    }

    public static class MyHandler implements NzaeMessageHandler
    {
        public void evaluate(Nzae ae, NzaeRecord input, NzaeRecord output) {

            final NzaeMetadata meta = ae.getMetadata();

            int op = 0;
            double result = 0;

            for (int i = 0; i < input.size(); i++) {
                if (input.getField(i) == null) {
                    continue;
                }
                int dataType = meta.getInputNzType(i);
                if (i == 0) {
                    if (!(dataType == NzaeDataTypes.NZUDSUDX_FIXED
                            || dataType == NzaeDataTypes.NZUDSUDX_VARIABLE))

                        { ae.userError("first column must be a string");

                    }
                    String opStr = input.getFieldAsString(0);
                    if (opStr.equals("*")) {
                        result = 1;
                        op = OP_MULT;
                    }
                    else if (opStr.equals("+")) {
                        result = 0;
                        op = OP_ADD;
                    }
                    else {
                        ae.userError("invalid operator = " + opStr);
                    }
                    continue;
                }
                switch (dataType) {
                    case NzaeDataTypes.NZUDSUDX_INT32:
                        switch (op) {
                            case OP_ADD:
                                result +=
input.getFieldAsNumber(i).doubleValue();
                                break;
                            case OP_MULT:
                                result *=
input.getFieldAsNumber(i).doubleValue();
                                break;
                            default:
                                break;
                        }
                        break;

                    default:
                        break;
                }
            } // end of column for loop

            output.setField(0, result);

        }
```

```
        }
        private static final int OP_ADD = 1;
        private static final int OP_MULT = 2;

        public static int run(Nzae ae)
        {
            ae.run(new MyHandler());
            return 0;
        }
    }
```

In this example, the run function instantiates the handler object, and then invokes the run method of the function class on the handler. The handler must implement an evaluate method, which takes a reference to the function as well as references to the input record and the output record. For normal functions, there is only one column in the output record. The evaluate can then retrieve fields from the input record and use that to set the output field. This code provides a basic, non-error-checking function that should take as the first argument either the string **+** or the string ***** and for subsequent arguments INT32 values. It then either adds or multiplies the INT32 values based on the specified argument value and returns the result as a double.

The final code preparation step adds error checking and support for additional data types.

```
import org.netezza.ae.*;

public class TestJavaInterface {

    public static final void main(String [] args) {
        try {
            mainImpl(args);
        } catch (Throwable t) {
            System.err.println(t.toString());
            NzaeUtil.logException(t, "main");
        }
    }

    public static final void mainImpl(String [] args) {
        NzaeApiGenerator helper = new NzaeApiGenerator();
        final NzaeApi api = helper.getApi(NzaeApi.FUNCTION);
        run(api.aeFunction);
        helper.close();
    }

    public static class MyHandler implements NzaeMessageHandler
    {
        public void evaluate(Nzae ae, NzaeRecord input, NzaeRecord output) {

            final NzaeMetadata meta = ae.getMetadata();

            int op = 0;
            double result = 0;

            if (meta.getOutputColumnCount() != 1 || meta.getOutputNzType(0) !=
                NzaeDataTypes.NZUDSUDX_DOUBLE) { throw new
                NzaeException("expecting one output column of type "+
                     "double");
            }
            if (meta.getInputColumnCount() < 1) {
                throw new NzaeException("expecting at least one input column");
            }
            if (meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_FIXED &&
                meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_VARIABLE) {
                throw new NzaeException("first input column expected to be "+
                     "a string type");
            }

            for (int i = 0; i < input.size(); i++) {
                if (input.getField(i) == null) {
                    continue;
                }
                int dataType = meta.getInputNzType(i);
                if (i == 0) {
                    if (!(dataType == NzaeDataTypes.NZUDSUDX_FIXED
                            || dataType == NzaeDataTypes.NZUDSUDX_VARIABLE))

                        { ae.userError("first column must be a string");

                    }
                    String opStr = input.getFieldAsString(0);
                    if (opStr.equals("*")) {
                        result = 1;
                        op = OP_MULT;
                    }
                    else if (opStr.equals("+")) {
                        result = 0;
                        op = OP_ADD;
                    }
                    else {
                        ae.userError("invalid operator = " + opStr);
```

```
                    }
                    continue;
                }
                switch (dataType) {
                    case NzaeDataTypes.NZUDSUDX_INT8:
                    case NzaeDataTypes.NZUDSUDX_INT16:
                    case NzaeDataTypes.NZUDSUDX_INT32:
                    case NzaeDataTypes.NZUDSUDX_INT64:
                    case NzaeDataTypes.NZUDSUDX_FLOAT:
                    case NzaeDataTypes.NZUDSUDX_DOUBLE:
                    case NzaeDataTypes.NZUDSUDX_NUMERIC32:
                    case NzaeDataTypes.NZUDSUDX_NUMERIC64:
                    case NzaeDataTypes.NZUDSUDX_NUMERIC128:
                        switch (op) {
                            case OP_ADD:
                                result +=
input.getFieldAsNumber(i).doubleValue();
                                break;
                            case OP_MULT:
                                result *=
input.getFieldAsNumber(i).doubleValue();
                                break;
                            default:
                                break;
                        }
                        break;

                    default:
                        break;
                }
            } // end of column for loop

            output.setField(0, result);

        }

    }
    private static final int OP_ADD = 1;
    private static final int OP_MULT = 2;

    public static int run(Nzae ae)
    {
        ae.run(new MyHandler());
        return 0;
    }
}
```

This code ensures the correct number and types of input arguments and the correct number and type of output arguments,. It supports all of the integer types, float, double and numeric.

Once the code is complete, it must be compiled and registered.

Compilation

To compile use the following:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java --template
     \ compile TestJavaInterface.java --version 3
```

The arguments specify that you are using the Java language, version 3 with the template compile and creating of the TestJavaInterface.class file from the source file TestJavaInterface.java.

Registration

Once compiled, the code is registered as:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "applyop_java(varargs)" \
     --return "double" --class AeUdf --language java --template udf \
     --define "java_class=TestJavaInterface" --version 3
```

This registers the UDF as applyop_java, taking varying arguments and returning a double. The class is AeUdf and the template is udf, which is true for all AE UDFs. This is registered in database db and the java class to run is TestJavaInterface.

Running

The AE can now be run in SQL:

```
SELECT applyop_java('+',1,2);
 APPLYOP_JAVA
--------------
            3
(1 row)
```

Notice in the code that you validate the types in doOnce and send an error using NzaeException. The following example triggers an error:

```
SELECT applyop_java('-',1,2);
ERROR:  invalid operator = -
```

Fortran Language Scalar Function

The following example shows a simple scalar function that sums a set of numbers. This example starts from the beginning to construct a simple AE. It uses the following file name:

```
applyop.f
```

Code

Create the program and call nzaeRun(); this call sets up the AE and then makes a callback to nzaeHandleRequest(), which is necessary for Fortran implemention. This can typically be accomplished using:

```
program applyopProgram
call nzaeRun()
stop
end

subroutine nzaeHandleRequest(handle)

Our custom Fortran code will go here.

return

end
```

Fill in the custom Fortran code. In this UDF, you pass in a string operator which is either a "+" (add) or a "*" (multiply), as well as two integers. The code returns either the product or the sum of the integers. In SQL, a simple version of this could be called by either:

```
SELECT applyop('+', 3, 5);
```

or

```
SELECT applyop('*', 3, 5);
```

Although the above example only has one input, the NPS system deals with streams of input. Therefore, the NPS system is more likely to use:

```
SELECT applyop(mytable.operator, mytable.value1, mytable.value2) FROM mytable;
```

In the first example, the NPS system takes the constants, "+", 3, and 5 and turns them into a stream of length one containing three elements. Internally, all the above cases are handled in the same manner and the assumption can be made that there are multiple inputs to the applyop AE.

A loop handling input is needed, which in Fortran 77 can be accomplished with an "if" statement and a "goto." For example:

```
program applyopProgram
call nzaeRun()
stop
end

subroutine nzaeHandleRequest(handle)
integer hasNext, leftInput, rightInput, result
character operator

hasNext    = -1

call nzaeGetNext(handle, hasNext)
if (hasNext .eq. 0) then
    goto 20

endif

Our loop work will go here.

goto 10

return

end
```

Note the initialization of the **hasNext** variable. This is necessary for "out" variables that call into the underlying interface. Without this, the hasNext pointer remains unallocated and cannot receive data from the underlying call. The nzaeGetNext() function sets the variable hasNext to 1 (TRUE) if there is another element in the stream, and to 0 (FALSE) if the stream is done.

Add the logic:

```
program applyOpProgram
call nzaeRun()
stop
end

subroutine nzaeHandleRequest(handle)
integer hasNext, leftInput, rightInput, isNull, result

character operator

hasNext    = -1
leftInput  =  0

rightInput =  0
```

```
operator   = 'f'
isNull     = 0

call nzaeGetNext(handle, hasNext)

if (hasNext .eq. 0) then

    goto 20

endif

GET OUR INPUT.
call nzaeGetInputString(handle, 0, operator, isNull)
call nzaeGetInputInt32(handle, 1, leftInput, isNull)
call nzaeGetInputInt32(handle, 2, rightInput, isNull)

OPERATE.
if (operator .eq. '+') then result
    = leftInput + rightInput
else if (operator .eq. '*') then
    result = leftInput * rightInput
else
    call nzaeUserError(handle, 'Unhandled operator.')

endif

OUTPUT.
call nzaeSetOutputInt32(handle, 0, result)

call nzaeOutputResult(handle)

goto 10

return

end
```

Each input in the loop contains the 3 elements that are passed into the function. The function, nzaeGetInputX(), takes the handle that represents one user request, a column index that in this case specifies 0 for operator, 1 for left input and 2 for right input, and the output variable. The function nzaeUserError() reports an error to the SQL user. The function nzaeSetOutputX() outputs the result. Because this is run as a UDF, there can only be one column in the result. A later example demonstrates how to use UDTFs to output multiple columns.

Once the code is complete, it must be compiled and registered.

Compilation

Compile the above code:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language fortran --version 3
     \ --template compile --exe applyopFortran applyop.f
```

The output executables, one for the host and one for the SPU, are placed under the "host" and "spu" directories in /nz/export/ae/applications/$NZ_USER/$NZ_DATABASE.

Registration

Register the executables:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language fortran --version 3
     \ --template udf --exe applyopFortran \
     --sig "applyop_fortran(varchar(1), int4, int4)" --return int4
```

Running

You can now run the AE in SQL:

```
SELECT applyop_fortran('+', 4, 10);
 APPLYOP
---------
      14
(1 row)
```

Notice in the code that when you validate the types in nzaeHandleRequest(), you call nzaeUserError() to send an error. The following example triggers an error:

```
SELECT applyop_fortran('-',1,2);
ERROR:  Unhandled operator.
```

Python Language Scalar Function

The following example shows a simple scalar function that sums a set of numbers. This example starts from the beginning to construct a simple AE. It uses the following file name:

```
applyop.py
```

Code

Derive a class from "nzae.Ae". The base class handles most of the work, so typically you call the class-method "run()". The run() function instantiates the class, sets up error handling, and calls the appropriate derived function. For UDF based AEs such as this one, the _getFunctionResult() function must be overridden, as it gets called once for each row of input. Save the following code in a file called applyop.py:

```
import nzae

class ApplyOpUdfAe(nzae.Ae):
    def _getFunctionResult(self, row):
           OUR CUSTOM CODE WILL GO HERE.


ApplyOpUdfAe.run()
```

Next, fill in the custom Python code. In this UDF, you pass in a string operator which is either a "+" (add) or a "*" (multiply) as well as two integers. The code returns either the product or the sum of the integers. In SQL, a simple version of this could be called by either:

```
SELECT applyop('+', 3, 5);
```

or

```
SELECT applyop('*', 3, 5);
```

Although the above example only has one input, the NPS system deals with streams of input. Therefore, the NPS system is more likely to use:

```
SELECT applyop(mytable.operator, mytable.value1, mytable.value2) FROM mytable;
```

In the first example, the NPS system takes the constants, "+", 3, and 5 and turns them into a stream of length one containing three elements. One row is"seen" by _getFunctionResult()

with the three elements in it. Internally, all the above cases are handled in the same manner and the assumption can be made that there are multiple inputs to the applyop AE.

Because loop handling input is needed to hand off one row to the function at a time, enter custom code to handle one row of input:

```
import nzae

class ApplyOpUdfAe(nzae.Ae):

    def _getFunctionResult(self, row):

           BREAK APART OUR ROW OF INPUT.
        operator, leftInput, rightInput = row
           HANDLE ADDITION.
        if operator == "+":
            return leftInput + rightInput

           HANDLE MULTIPLICATION.
        if operator == "*":
            return leftInput * rightInput

           ERROR ON ALL OTHER.
        self.userError("Unhandled operator to ApplyOp: '" + operator + "'.")

ApplyOpUdfAe.run()
```

The function self.userError() reports an error to the SQL user. Whatever is returned by _getFunctionResult() is the result of the SQL operation. Because this is run as a UDF, there can only be one column in the result. A later example demonstrates how to use UDTFs to output multiple columns.

Once the code is complete, it must be deployed and registered.

Compilation

The Python AEs do not require compilation.

Deployment

Although Python AEs do not require compilation, they do require deployment. The **compile_ae** command is still used with the **--template deploy** option to deploy the script to the default location on the shared export drive.

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language python64
     \ --template deploy ./applyop.py --version 3
```

Registration

Register the Python file:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3 \
     --template udf --exe applyop.py --sig "applyop(varchar(1), int4, int4)"
     \ --return int4
```

Running

The AE can now be run in SQL:

```
SELECT applyop('+', 4, 10);
 APPLYOP
---------
      14
(1 row)
```

Note that to validate types in _getFunctionResult(), self.userError() is called. The following example triggers an error:

```
SELECT applyop('-', 1, 2);
ERROR:  Unhandled operator TO ApplyOp: '-'.
```

Perl Language Scalar Function

The following example shows a simple scalar function that sums a set of numbers. The example uses the following file name:

```
ApplyOp.pm
```

Code

Name the example class using the Perl package command:

```
package ApplyOp;
```

Import the nzae::Ae package and define the example class as a child class of the nzae::Ae class.

```
use nzae::Ae;
our @ISA = qw(nzae::Ae);
```

Define the class. As in the Python examples, the base class handles most of the work. Typically the new() method is called to initialize the class, and the run() method is then called. The run() method calls the appropriate derived function. For UDF based AEs such as this one, the _getFunctionResult() function must be overridden, as it gets called once for each row of input by default:

```
package ApplyOp;
use strict;
use autodie;
use nzae::Ae;

our @ISA = qw(nzae::Ae);
my $ae = ApplyOp->new();
$ae->run();

sub _getFunctionResult(@)
{
#User defined function code goes here
}

1;
```

Add the custom Perl code. In this UDF, a string operator is passed in, which is either a "+" (add) or a "*" (multiply) and two integers. The code returns either the product or the sum of the integers. In SQL, a simple version of this could be called by either:

```
SELECT applyopPl('+', 3, 5);
```

or

```
SELECT applyopPl('*', 3, 5);
```

While the above example only has one input, the NPS system deals with streams of input. Therefore, the NPS system is more likely to use:

```
SELECT applyopPl(mytable.operator, mytable.value1, mytable.value2) FROM  mytable;
```

In the first example, the NPS system takes the constants, "+", 3, and 5 and turns them into a stream of length 1 containing three elements. One row is "seen" by getFunctionResult() with the three elements in it. In addition, as per Perl convention, the calling object is passed in as the first argument to the function. This is picked up by the function using the variable $self. Internally, all the above cases are handled in the same manner and the assumption can be made that there are multiple inputs to the applyop AE.

Because loop handling input is needed to hand off one row to the function at a time, enter custom code to handle one row of input:

```
package ApplyOp;
use nzae::Ae;
use strict;
use autodie;

our @ISA = qw(nzae::Ae);
my $ae = ApplyOp->new();
$ae->run();

sub _getFunctionResult(@)
{
    my $self = shift;
    # BREAK APART OUR ROW OF INPUT.
    my ($operator, $leftInput, $rightInput) = @_;

      HANDLE ADDITION. if
    ($operator eq "+")
    {
        return $leftInput + $rightInput;
    }

      HANDLE MULTIPLICATION.
    if ($operator eq "*")
    {
        return $leftInput * $rightInput;
    }

    # ERROR ON ALL OTHER.
croak(nzae::Exceptions::AeInternalError->new("Unhandled operator to ApplyOp: '" .
$operator . "'."));
}
1;
```

The croak function reports the error to the SQL user and, if logging is enabled, returns a trace to the line of error.

You can use the cluck function in place of croak for writing warnings to the log file. To catch any unhandled exceptions automatically, use autodie. Die and warn can also be used to report a single line error or warning if logging is turned on. Die errors out the execution of the AE. Set log level using the **--level** option during registration. What is returned by _getFunctionResult() is the result of the

SQL operation; because this is run as a UDF, there can only be one column in the result.

In summary, to write a UDF adding only the actual functionality without customizing the methods for running it:

Create a Perl module file (ApplyOp.pm in the example above).

Import and instantiate the nzae::Ae class in the file.

Import autodie to handle unhandled exceptions during execution.

Override_getFunctionResult to implement customized functionality for UDF. Handle errors to be returned to the user using the croak() method.

Execute the run() method of the nzae::Ae object.

Since this is a Perl module, add a "1" at the end of the file ApplyOp.pm.

Once the code is complete, it must be deployed and registered.

Compilation

Perl AEs do not require compilation.

Deployment

Although Perl AEs do not require compilation, they must be deployed. The **compile_ae** command is still used with the **--template deploy** option to deploy the script to the default location on the shared export drive.

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language perl --version 3
\ --template deploy ApplyOp.pm
```

Registration

Register the Perl file:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3 \
     --template udf --exe ApplyOp.pm --sig "applyopPl(varchar(1), int4, int4)"\
     --return int4
```

Running

The AE can now be run in SQL on the system database:

```
SELECT applyopPl('+', 4, 10);
APPLYOPPL
-----------
        14
(1 row)
```

Note that to validate types in _getFunctionResult(), the croak() method is called. The following example triggers an error:

```
SELECT applyopPl('-', 1, 2);
ERROR: Unhandled operator to ApplyOp: '-'. at
/nz/export/ae/applications/system/admin/ApplyOp.pm line 28
```

R Language Scalar Function 1

The following example creates a function returning the total length of all character input columns.

Code

In R AE, the main object that is provided by the user is a function referred to as nz.fun. It encapsulates the main AE loop and uses the R AE API to receive and output data.

The following example shows the *applyop* implementation in R.

Enter the following code in the */tmp/applyop.R* file:

```
nz.fun <- function () {
  while(getNext()) {
    op <- getInputColumn(0)
    X <- c()
    for (i in seq(1, inputColumnCount()-1)) {
      x <- as.numeric(getInputColumn(i))
      if (!is.null(x) && !is.na(x))
        X <- append(X, x)
    }

    if (op != '+' && op != '*')
      stop('incorrect operator: ', op)

    setOutputInt32(0, eval(parse(text=paste(X, collapse=op))))
    outputResult()
  }
}
```

The main loop ends when getNext() returns false, which indicates that there are no more input rows to process. Each input row is accessed by the getInputColumn() that returns the value of the column specified by its index. The index of the first column is zero, and the index of the last column is **inputColumnCount()-1**. The output values are handled by the setOutput* functions, where the asterisk represents a specific data type identifier. To send the output values to the database, the outputResult() function must be called.

If an error occurs, call the standard R stop() function, or use the AE API function named userError(). Both functions interrupt program execution and return the error message to the caller.

Compilation

Compile the code as follows:

```
/nz/export/ae/utilities/bin/compile_ae --language r \
  --version 3 --template compile --user nz --db dev \
  /tmp/applyop.R
```

During the compilation step, the input source file is serialized by using standard R API and prepared for further execution. The output is stored under a predefined path that is accessible from the Host and on the SPUs, depending on the database and user names.

Registration

After the compilation is completed, you must register the code.

For the registration step, the following conditions apply:

As the template name, udf must be specified.

User name, database name, and UDX signature including output type must match the compilation values.

The executable name must match the source file name from the compilation

step. Register the code as follows:

```
/nz/export/ae/utilities/bin/register_ae --language r \
  --version 3 --user nz --db dev --template udf \
  --sig 'applyop(VARARGS)' --return INT4 --exe applyop.R
```

Running

When you run the registered UDAP, you get the following result:

```
SELECT applyop('+', 4, 10);
 APPLYOP
---------
      14
(1 row)

SELECT applyop('*', 3, 5);
 APPLYOP
---------
      15
(1 row)
```

The following example shows error handling:

```
SELECT applyop('-', 3, 5);
ERROR:  Error in function ()  : incorrect operator: -
```

R Language Scalar Function 2

The following function calculates the total length of the character input columns.

Code

Enter the following code in the */tmp/rlength.R* file:

```
nz.fun <- function () {
  while(getNext()) {
    s <- 0
    for (i in seq(inputColumnCount())) {
      x <- getInputColumn(i-1)
      if (is.character(x)) {
        s <- s + nchar(x)
      }
    }
    if (s == 0) {
      stop('no character columns in input')
```

```
    }
    setOutputInt32(0, s)
    outputResult()
  }
}
```

Compilation

Compile the code as follows:

```
/nz/export/ae/utilities/bin/compile_ae --language r \
  --version 3 --template compile --user nz --db dev \
  /tmp/rlength.R
```

To take full advantage of the R code, register the compiled UDAP with the VARARGS input signature. This signature indicates that the given function can handle any number of arguments of arbitrary types:

```
/nz/export/ae/utilities/bin/register_ae --language r \
  --version 3 --user nz --db dev --template udf \
  --sig 'rlength(VARARGS)' --return INT4 --exe rlength.R
```

In these examples the user is specified as nz, which is an example user ID. The value that you use should be a valid database user name.

When you run this function, you get the following result:

```
SELECT rlength('sample text');
 RLENGTH
---------
      11
(1 row)
```

When you run this function for a table **iris**, which is a copy of the standard R *iris* data set , you get the following result:

```
SELECT rlength(SPECIES) FROM iris;
 RLENGTH
---------
       6
       6
       6
(147 more rows)
```

The following example shows the use of the VARARGS input signature:

```
SELECT rlength('sample text',1, 'another text',2,3,4,'and one more text');
 RLENGTH
---------
      40
(1 row)
```

CHAPTER 6

Converting to a Simple Table Function Examples

C Language Conversion

The code from the C Language Scalar Function can be reused as a simple table function by making minor modifications to the registration.

Code

The code is the same as found in the C Language Scalar Function section.

Compilation

Compilation is the same as that performed in local or remote modes.

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language system --version 3 \
     --template compile func.c --exe apply
```

Registration

Register the executables using the modified command, which changes **--template** and **--return** to be appropriate for a table function:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3
     \ --template udtf --exe apply --sig "tapplyop_c(VARARGS)" \
     --return "TABLE (D double)"
```

Running

The AE can now be run in SQL:

```
SELECT * FROM TABLE WITH FINAL(tapplyop_c('+',1,2));
```

```
 D
---
 3
(1 row)
```

C++ Language Conversion

The code from the C++ Language Scalar Function can be reused as a simple table function by making minor modifications to the registration.

Code

The code is the same as found in the C++ Language Scalar Function section.

Compilation

Compilation is the same as that performed in local or remote modes.

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp --template compile
     \ --exe applyopcpp --compargs "-g -Wall" --linkargs "-g" applyopcpp.cpp
     \ --version 3
```

Registration

Register the executables using the modified command, which changes **--template** and **--return** to be appropriate for a table function:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae \
     --sig "tapplyop_cpp(VARARGS)" \
     --return "table(d double)" --language cpp --template udtf
     \ --exe applyopcpp --version 3
```

Running

The AE can now be run in SQL:

```
SELECT * FROM TABLE WITH FINAL(tapplyop_cpp('+',1,2));
 D
---
 3
(1 row)
SELECT * FROM TABLE WITH FINAL(tapplyop_cpp('+',1,2,3,4));
 D
---
 10
(1 row)
```

Java Language Conversion

The code from the Java Language Scalar Function can be reused as a simple table function by making minor modifications to the registration.

Code

The code is the same as found in the Java Language Scalar Function section.

Compilation

Compilation is the same as that performed in local or remote modes.

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java --template
     \ compile TestJavaInterface.java --version 3
```

Registration

Register the executables using the modified command, which changes **--template**, **--return** and **--class** to be appropriate for a table function:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae \
     --sig "tapplyop_java(varargs)" \
     --return "table(d double)" --class AeUdtf --language java \
     --template udtf --define "java_class=TestJavaInterface" --version 3
```

Running

The AE can now be run in SQL:

```
SELECT * FROM TABLE WITH FINAL(tapplyop_java('+', 4,5,1.1,10));
  D
------
 20.1
(1 row)
```

Fortran Language Conversion

The code from the Fortran Language Scalar Function can be reused as a simple table function by making minor modifications to the registration.

Code

The code is the same as found in the Fortran Language Scalar Function section.

Compilation

Compilation is the same as that performed in local mode.

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language fortran --version 3
     \ --template compile --exe applyopFortran applyop.f
```

Registration

Register the executables using the modified command, which changes **--template** and **--return** to be appropriate for a table function:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language fortran --version 3
     \ --template udtf --exe applyopFortran \
     --sig "applyop_table(varchar(1), int4, int4)" \
     --return "table(result int4)"
```

Running

The AE can now be run in SQL:

```
SELECT * FROM TABLE WITH FINAL(applyop_table('+', 4, 5));

 RESULT
--------
 9
(1 row)
```

Python Language Conversion

The code from the Python Language Scalar Function can be reused as a simple table function by making minor modifications to the registration.

Code

The code is the same as found in the Python Language Scalar Function section.

Deployment

Deploy the script using the same steps as found in the Python Language Scalar Function section:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language python64
     \ --template deploy ./applyop.py --version 3
```

Registration

Register the executables using the modified command, which changes **--template**, **--return** and **--sig** to be appropriate for a table function:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3
     \ --template udtf --exe applyop.py \
     --sig "applyop_table(varchar(1), int4, int4)" \
     --return "table(result int4)"
```

Running

The AE can now be run in SQL:

```
SELECT * FROM TABLE WITH FINAL(applyop_table('+', 4, 5));

 RESULT
--------
 9
(1 row)
```

Perl Language Conversion

The code from the Perl Language Scalar Function can be reused as a simple table function by making minor modifications to the registration.

Code

The code is the same as found in the Perl Language Scalar Function section.

Deployment

Deploy the script using the same steps as found in the Perl Language Scalar Function section.

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language perl --version 3
\ --template deploy ApplyOp.pm
```

Registration

Register the executables using the modified command, which changes **--template**, **--return** and **--sig** to be appropriate for a table function:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3
     \ --template udtf --exe ApplyOp.pm \
     --sig "applyopPl_table(varchar(1), int4, int4)" \
     --return "table(result int4)"
```

Running

The AE can now be run in SQL on the system database:

```
SELECT * FROM TABLE WITH FINAL(applyopPl_table('+', 4,
5)); RESULT
--------
9
(1 row)
```

R Language Conversion

The code from the R Language Scalar Function 1 can be reused as a simple table function by making minor modifications to the registration.

Code

The code is the same as in R Language Scalar Function 1.

Compilation

The compilation is the same as the one that is done in local or remote modes.

Registration

Register the executables by using the modified command, changing **--template** and **--return** to be appropriate for a table function:

```
/nz/export/ae/utilities/bin/register_ae --language r --version 3 \
     --template udtf --exe applyop.R --sig "tapplyop_r(VARARGS)" \
     --return "TABLE(RESULT INT4)" --db dev --user nz
```

Running

You can now run the AE in SQL as follows:

```
SELECT * FROM TABLE WITH FINAL(tapplyop_r('+',1,2));
 RESULT
--------
 3
(1 row)
```

CHAPTER 7

Table Function (Simulated Row Function) Examples

This example creates a simulated row function that takes as input a string representation of time (or example, 12:30:55) and outputs a 3-column wide row with the input split into its component parts

C Language Simulated Row Function

This example uses the following file name:

```
split.c
```

Code

```
#include <stdio.h>
#include <stdlib.h>

#include "nzaeapis.h"

static int run(NZAE_HANDLE h);
int main(int argc, char * argv[])
{
    if (nzaeIsLocal())
    {
        NzaeInitialization arg;
        memset(&arg, 0, sizeof(arg));
                arg.ldkVersion = NZAE_LDK_VERSION;
        if (nzaeInitialize(&arg))
        {
            fprintf(stderr, "initialization failed\n");
            return -1;
        }
        run(arg.handle);
        nzaeClose(arg.handle);
    }
    else {
        NZAECONPT_HANDLE hConpt = nzaeconptCreate();
        if (!hConpt)
        {
            fprintf(stderr, "error creating connection point\n");
```

```
            fflush(stderr);
            return -1;
        }
        const char * conPtName = nzaeRemprotGetRemoteName();
        if (!conPtName)
        {
            fprintf(stderr, "error getting connection point name\n");
            fflush(stderr);
            exit(-1);
        }
        if (nzaeconptSetName(hConpt, conPtName))
        {
            fprintf(stderr, "error setting connection point name\n");
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NzaeremprotInitialization args;
        memset(&args, 0, sizeof(args));
        args.ldkVersion = NZAE_LDK_VERSION;
        args.hConpt = hConpt;
        if (nzaeRemprotCreateListener(&args))
        {
            fprintf(stderr, "unable to create listener -
%s\n", args.errorMessage);
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NZAEREMPROT_HANDLE hRemprot = args.handle;
        NzaeApi api;
        int i;
        for (i = 0; i < 5; i++)
        {
            if (nzaeRemprotAcceptApi(hRemprot, &api))
            {
                fprintf(stderr, "unable to accept API - %s\n", \
                     nzaeRemprotGetLastErrorText(hRemprot));
                fflush(stderr);
                nzaeconptClose(hConpt);
                nzaeRemprotClose(hRemprot);
                return -1;
            }
            if (api.apiType != NZAE_API_FUNCTION)
            {
                fprintf(stderr, "unexpected API returned\n");
                fflush(stderr);
                nzaeconptClose(hConpt);
                nzaeRemprotClose(hRemprot);
                return -1;
            }
            printf("testcapi: accepted a remote request\n");
            fflush(stdout);
            run(api.handle.function);
        }
        nzaeRemprotClose(hRemprot);
        nzaeconptClose(hConpt);
    }
    return 0;
}

static int run(NZAE_HANDLE h)
{
    NzaeMetadata metadata;
    if (nzaeGetMetadata(h, &metadata))
    {
        fprintf(stderr, "get metadata failed\n");
        return -1;
```

```
}

#define CHECK(value) \
{ \
    NzaeRcCode rc = value; \
    if (rc) \
    { \
        const char * format = "%s in %s at %d"; \
        fprintf(stderr, format,  \
            nzaeGetLastErrorText(h), __FILE__, __LINE__); \
        nzaeUserError(h, format, \
            nzaeGetLastErrorText(h), __FILE__, __LINE__); \
        exit(-1); \
    } \
}

if (metadata.outputColumnCount != 3 ||
    metadata.outputTypes[0] != NZUDSUDX_INT32 ||
    metadata.outputTypes[1] != NZUDSUDX_INT32 ||
    metadata.outputTypes[2] != NZUDSUDX_INT32)

{
    nzaeUserError(h, "expecting three output column of type
    int32"); nzaeDone(h);
    return -1;
}

if (metadata.inputColumnCount != 1)
{
    nzaeUserError(h, "expecting one input column");
    nzaeDone(h);
    return -1;
}

if (metadata.inputTypes[0] != NZUDSUDX_FIXED
    && metadata.inputTypes[0] != NZUDSUDX_VARIABLE)
{
    nzaeUserError(h, "first input column expected to be a string
    type"); nzaeDone(h);
    return -1;
}

for (;;)
{
    int i;
    double result = 0;
    NzaeRcCode rc = nzaeGetNext(h);
    if (rc == NZAE_RC_END)
    {
        break;
    }

    NzudsData * input = NULL;
    CHECK(nzaeGetInputColumn(h, 0, &input));
    const char * opString;
    if (input->isNull)
    {
        nzaeUserError(h, "first input column may not be
        null"); nzaeDone(h);
        return -1;
    }
    if (input->type == NZUDSUDX_FIXED)
    {
        opString = input->data.pFixedString;
    } else if (input->type == NZUDSUDX_VARIABLE)
    {
        opString = input-
    >data.pVariableString; } else
```

```
        {
            nzaeUserError(h, "first input column expected to be a string type");
            nzaeDone(h);
            return -1;
        }
        int hour, min, sec;
        int nfound = sscanf(opString, "%d:%d:%d", &hour, &min, &sec);
        if (nfound != 3) {
            nzaeUserError(h, "Badly formatted time. Expected h:m:s");
            nzaeDone(h);
            return -1;
        }

        CHECK(nzaeSetOutputInt32(h, 0, hour));
        CHECK(nzaeSetOutputInt32(h, 1, min));
        CHECK(nzaeSetOutputInt32(h, 2, sec));
        CHECK(nzaeOutputResult(h));
    }
    nzaeDone(h);
    return 0;
}
```

Compilation

Compile the code:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language system --version 3 \
     --template compile split.c --exe split
```

Registration

Since there is no built-in concept of a row function in Netezza SQL, this function is registered as a table function, even though it outputs only one row per input row:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3
     \ --template udtf --exe split --sig "split_c(VARCHAR(ANY))" \ --
     return "TABLE (h int4, m int4, s int4)"
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(split_c('12:30:15'));
    |M |S
----+----+----
 12|30|15 (1
row)
```

C++ Language Simulated Row Function

This example uses the following file name:

```
time.cpp
```

Code

```
#include <nzaefactory.hpp>
```

```
using namespace nz::ae;

static int run(nz::ae::NzaeFunction *aeFunc);


int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
       The following line is only needed if a launcher is not
    used helper.setName("testcapi");
    for (int i=0; i < 2; i++) {
        nz::ae::NzaeApi &api =
        helper.getApi(nz::ae::NzaeApi::FUNCTION); run(api.aeFunction);
        if (!helper.isRemote())
            break;
    }

    return 0;
}

class MyHandler : public NzaeFunctionMessageHandler
{
public:
    MyHandler() {
        m_Once = false;
    }

    void doOnce(NzaeFunction& api) {
        const NzaeMetadata& meta = api.getMetadata();
        if (meta.getOutputColumnCount() != 3 ||
            meta.getOutputType(0) != NzaeDataTypes::NZUDSUDX_INT32 ||
            meta.getOutputType(1) != NzaeDataTypes::NZUDSUDX_INT32 ||
            meta.getOutputType(2) != NzaeDataTypes::NZUDSUDX_INT32
            ) {
            throw NzaeException("expecting three output columns of type \
                 int32");
        }
        if (meta.getInputColumnCount() != 1) {
            throw NzaeException("expecting one input column");
        }
        if (meta.getInputType(0) != NzaeDataTypes::NZUDSUDX_FIXED &&
            meta.getInputType(0) != NzaeDataTypes::NZUDSUDX_VARIABLE) { throw
            NzaeException("first input column expected to be a string \
                 type");
        }
        m_Once = true;

    }
    void evaluate(NzaeFunction& api, NzaeRecord &input, NzaeRecord &result) {

        if (!m_Once)
            doOnce(api);
        int32_t hour, min, sec;

        NzaeField &field = input.get(0);
        if (field.isNull())
            throw NzaeException("first input column may not be null");
        NzaeStringField &sf = (NzaeStringField&) input.get(0);
        std::string str = (std::string)sf;
        int nfound = sscanf(str.c_str(), "%d:%d:%d", &hour, &min,
        &sec); if (nfound != 3)
            throw NzaeException(NzaeException::format("Badly formatted time
                 \ %s. Expected h:m:s", str.c_str()));
        NzaeInt32Field &f = (NzaeInt32Field&)result.get(0);
        f = hour;
        NzaeInt32Field &f2 = (NzaeInt32Field&)result.get(1);
```

```
        f2 = min;
        NzaeInt32Field &f3 = (NzaeInt32Field&)result.get(2);
        f3 = sec;
    }

    bool m_Once;

};
static int run(NzaeFunction *aeFunc)
{

    aeFunc->run(new MyHandler());
    return 0;
}
```

Compilation

Compile the code:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp --template compile \
     --exe timeae --compargs "-g -Wall" --linkargs "-g" time.cpp --version 3
```

Registration

Since there is no built-in concept of a row function in Netezza SQL, this function is registered as a table function, even though it outputs only one row per input row:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "timeae(VARCHAR(ANY))" \
     --return "table(hour int4, minute int4, second int4)" --language cpp
     \ --template udtf --exe timeae --version 3
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(timeae('12:30:15'));
 HOUR | MINUTE | SECOND
------+--------+--------
    12|     30 |     15
(1 row)

SELECT * FROM TABLE WITH FINAL(timeae('12:30'));
ERROR:  Badly formatted time 12:30. Expected h:m:s
```

Java Language Simulated Row Function

This example uses the following file name:

```
TestJavaTime.java
```

Code

```
import org.netezza.ae.*;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
```

```
public class TestJavaTime {

    private static final Executor exec =
        Executors.newCachedThreadPool();


    public static final void main(String [] args) {
        try {
            mainImpl(args);
        } catch (Throwable t) {
            System.err.println(t.toString());
            NzaeUtil.logException(t, "main");
        }
    }

    public static final void mainImpl(String [] args) {
        NzaeApiGenerator helper = new NzaeApiGenerator();
        while (true) {
            final NzaeApi api = helper.getApi(NzaeApi.FUNCTION);
            if (api.apiType == NzaeApi.FUNCTION) {
                if (!helper.isRemote()) {
                    run(api.aeFunction);
                    break;
                } else {
                    Runnable task = new Runnable() {
                            public void run() {
                                try {
                                    TestJavaTime.run(api.aeFunction);
                                } finally {
                                    api.aeFunction.close();
                                }
                            }
                        };
                    exec.execute(task);


                }
            }
        }
        helper.close();
    }

    public static class MyHandler implements NzaeMessageHandler
    {
        public void evaluate(Nzae ae, NzaeRecord input, NzaeRecord output) {

            final NzaeMetadata meta = ae.getMetadata();

            int op = 0;
            double result = 0;

            if (meta.getOutputColumnCount() != 3 ||
                meta.getOutputNzType(0) != NzaeDataTypes.NZUDSUDX_INT32 ||
                meta.getOutputNzType(1) != NzaeDataTypes.NZUDSUDX_INT32 ||
                meta.getOutputNzType(2) != NzaeDataTypes.NZUDSUDX_INT32
                ) {
                throw new NzaeException("expecting three output columns of "+
                      "type int32");
            }
            if (meta.getInputColumnCount() != 1) {
                throw new NzaeException("expecting one input column");
            }
            if (meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_FIXED &&
                meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_VARIABLE) {
                throw new NzaeException("first input column expected to be a "+
                     "string type");
```

```
            }

            int hour, min, sec;

            Object field = input.getField(0);
            if (field == null)
                throw new NzaeException("first input column may not be null");

            String sf = (String)field;
            String[] ar = sf.split(":");
            if (ar.length != 3)
                throw new NzaeException("Badly formatted time " + sf + ".
                     "+ "Expected h:m:s");
            hour = Integer.parseInt(ar[0]);
            min = Integer.parseInt(ar[1]);
            sec = Integer.parseInt(ar[2]);

            output.setField(0, hour);
            output.setField(1,min);
            output.setField(2, sec);

        }

    }

    public static int run(Nzae ae)
    {
        ae.run(new MyHandler());
        return 0;
    }

}
```

Compilation

Compile the code:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java
     \ --template compile TestJavaTime.java --version 3
```

Registration

Since there is no built-in concept of a row function in Netezza SQL, this function is registered as a table function, even though it outputs only one row per input row:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "timeae(VARCHAR(ANY))" \
     --return "table(hour int4, minute int4, second int4)" --class AeUdtf \
     --language java --template udtf   --version 3 --define \
     "java_class=TestJavaTime"
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(timeae('12:30:15'));
 HOUR | MINUTE | SECOND
      +        +
------12| --------30| --------15
(1 row)

SELECT * FROM TABLE WITH FINAL(timeae('12:30'));
```

```
ERROR:  Badly formatted time 12:30. Expected h:m:s
```

Fortran Language Simulated Row Function

This example uses the following file name:

```
timeSplit.f
```

Code

```
    program timeSplitProgram
      call nzaeSetConnectionPointName('timeSplit')
      call nzaeRun()
      stop
      end

      subroutine nzaeHandleRequest(handle)
      integer hasNext, isNull, result
      character(10) timeString

      hasNext    = -1
      isNull     = 0
      timeString = '00000000'
      call nzaeGetNext(handle, hasNext)
      if (hasNext .eq. 0) then
          goto 20
      endif
      GET OUR INPUT.
      call nzaeGetInputString(handle, 0, timeString, isNull)

      SET OUR OUTPUT.
      call nzaeSetOutputString(handle, 0, timeString(1:2))
      call nzaeSetOutputString(handle, 1, timeString(4:5))
      call nzaeSetOutputString(handle, 2, timeString(7:8))
      OUTPUT THE ROW.
      call nzaeOutputResult(handle)
      goto 10
      return
      end
```

Compilation

For Fortran compilation, the executable defaults to the stripped Fortran file name (timeSplit in this example):

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language fortran --version 3
     \ --template compile timeSplit.f
```

Registration

Since there is no built-in concept of a row function in Netezza SQL, this function is registered as a table function, even though it outputs only one row per input row:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language fortran --version 3 \
```

```
    --template udtf --exe timeSplit --sig "time_split(varchar(10))" \
    --return "table(hour varchar(2), minute varchar(2), second varchar(2))"
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(time_split('13:22:47'));
 HOUR | MINUTE  | SECOND
      +         +
------13| --------22| --------47
```

Python Language Simulated Row Function

This example uses the following file name:

```
timeSplit.py
```

Code

```
import nzae

class TimeSplitAe(nzae.Ae):

    def _getFunctionResult(self, row):
        return row[0].split(":")

TimeSplitAe.run()
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language python64
     \ --template deploy ./timeSplit.py --version 3
```

Registration

Since there is no built-in concept of a row function in Netezza SQL—although there is one in the Python adapter interface—this function is registered as a table function, even though it outputs only one row per input row:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3
     \ --template udtf --exe timeSplit.py --sig "time_split(varchar(20))" \
     --return "table(hour varchar(2), minute varchar(2), second varchar(2))"
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(time_split('13:22:47'));
 HOUR | MINUTE  | SECOND
      +         +
------13| --------22| --------47
```

Perl Language Simulated Row Function

This example uses the following file name:

```
TimeSplit.pm
```

Code

```
package TimeSplit;

use nzae::Ae;
use strict;
use autodie;

our @ISA = qw(nzae::Ae);

my $ae = TimeSplit->new();
$ae->run();

sub _getFunctionResult(@)
{
    my $self = shift;
    my @currentTime = @_;
    my @time = split(/:/, $currentTime[0]);
    return (@time);
}

1;
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language perl --version 3
     \ --template deploy TimeSplit.pm
```

Registration

Since there is no built-in concept of a row function in Netezza SQL—although there is one in the Perl adapter interface—this function is registered as a table function, even though it outputs only one row per input row:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3 \ -
     -template udtf --exe TimeSplit.pm --sig "time_split(varchar(20))" \ --
     return "table(hour varchar(2), minute varchar(2), second varchar(2))"
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(time_split('13:22:47'));
HOUR | MINUTE | SECOND
                        +---------
13---------- | 22              | 47
```

R Language Simulated Row Function

Code

The following example creates a simulated row function that takes as input a string representation of time, for example, "12:30:55" and outputs a 3-wide row with the input split it into its component parts.

Enter the following code in the */tmp/split.R* file:

```
nz.fun <- function() {
  while(getNext()) {
    chunks <- strsplit(getInputColumn(0), ':')[[1]]
    for (i in seq(chunks))
      setOutputString(i-1, chunks[i])
    outputResult()
  }
}
```

Compilation

Compile the code as follows:

```
/nz/export/ae/utilities/bin/compile_ae --language r --version 3 \
    --template compile --user nz --db dev /tmp/split.R
```

Registration

Because there is no built-in concept of a row function in Netezza SQL, this function is registered as a table function, even though it outputs only one row per input row:

```
/nz/export/ae/utilities/bin/register_ae --language r --version 3
     \ --template udtf --exe split.R --sig "split(VARCHAR(20))" \
     --return "TABLE(hour VARCHAR(2), minute VARCHAR(2), second VARCHAR(2))"
```

Running

When you run this function in nzsql, you get the following result:

```
SELECT * FROM TABLE WITH FINAL(split('13:22:47'));
 HOUR | MINUTE  | SECOND
      +         +
------13| --------22| --------47
```

CHAPTER 8
Simple Table Function Examples

C Language Table Function

This example uses the following file name:

```
sstring.c
```

Code

The code in this example creates a function that splits a string into multiple strings based on the delimiter. This example may return more rows of output than input. Note the additional loop logic in run.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "nzaeapis.h"

static int run(NZAE_HANDLE h);
int main(int argc, char * argv[])
{
    if (nzaeIsLocal())
    {
        NzaeInitialization arg;
        memset(&arg, 0, sizeof(arg));
                arg.ldkVersion = NZAE_LDK_VERSION;
        if (nzaeInitialize(&arg))
        {
            fprintf(stderr, "initialization failed\n");
            return -1;
        }
        run(arg.handle);
        nzaeClose(arg.handle);
    }
    else {
        NZAECONPT_HANDLE hConpt = nzaeconptCreate();
```

```
        if (!hConpt)
        {
            fprintf(stderr, "error creating connection point\n");
            fflush(stderr);
            return -1;
        }
        const char * conPtName = nzaeRemprotGetRemoteName();
        if (!conPtName)
        {
            fprintf(stderr, "error getting connection point name\n");
            fflush(stderr);
            exit(-1);
        }
        if (nzaeconptSetName(hConpt, conPtName))
        {
            fprintf(stderr, "error setting connection point name\n");
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NzaeremprotInitialization args;
        memset(&args, 0, sizeof(args));
        args.ldkVersion = NZAE_LDK_VERSION;
        args.hConpt = hConpt;
        if (nzaeRemprotCreateListener(&args))
        {
            fprintf(stderr, "unable to create listener - %s\n", \
                 args.errorMessage);
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NZAEREMPROT_HANDLE hRemprot = args.handle;
        NzaeApi api;
        int i;
        for (i = 0; i < 5; i++)
        {
            if (nzaeRemprotAcceptApi(hRemprot, &api))
            {
                fprintf(stderr, "unable to accept API - %s\n", \
                     nzaeRemprotGetLastErrorText(hRemprot));
                fflush(stderr);
                nzaeconptClose(hConpt);
                nzaeRemprotClose(hRemprot);
                return -1;
            }
            if (api.apiType != NZAE_API_FUNCTION)
            {
                fprintf(stderr, "unexpected API returned\n");
                fflush(stderr);
                nzaeconptClose(hConpt);
                nzaeRemprotClose(hRemprot);
                return -1;
            }
            printf("testcapi: accepted a remote request\n");
            fflush(stdout);
            run(api.handle.function);
        }
        nzaeRemprotClose(hRemprot);
        nzaeconptClose(hConpt);
    }
    return 0;
}

static int run(NZAE_HANDLE h)
{
    NzaeMetadata metadata;
    if (nzaeGetMetadata(h, &metadata))
```

```
{
    fprintf(stderr, "get metadata failed\n");
    return -1;
}

#define CHECK(value) \
{ \
    NzaeRcCode rc = value; \
    if (rc) \
    { \
        const char * format = "%s in %s at %d"; \
        fprintf(stderr, format,  \
            nzaeGetLastErrorText(h), __FILE__, __LINE__); \
        nzaeUserError(h, format, \
            nzaeGetLastErrorText(h), __FILE__, __LINE__); \
        exit(-1); \
    } \
}

if (metadata.outputColumnCount != 1 ||
    metadata.outputTypes[0] != NZUDSUDX_VARIABLE)

{
    nzaeUserError(h, "expecting one output column of type
    string"); nzaeDone(h);
    return -1;
}

if (metadata.inputColumnCount != 1)
{
    nzaeUserError(h, "expecting one input column");
    nzaeDone(h);
    return -1;
}

if (metadata.inputTypes[0] != NZUDSUDX_FIXED
    && metadata.inputTypes[0] != NZUDSUDX_VARIABLE)
{
    nzaeUserError(h, "first input column expected to be a string
    type"); nzaeDone(h);
    return -1;
}

for (;;)
{
    int i;
    double result = 0;
    NzaeRcCode rc = nzaeGetNext(h);
    if (rc == NZAE_RC_END)
    {
        break;
    }

    NzudsData * input = NULL;
    CHECK(nzaeGetInputColumn(h, 0, &input));
    const char * opString;
    if (input->isNull)
    {
        nzaeUserError(h, "first input column may not be
        null"); nzaeDone(h);
        return -1;
    }
    if (input->type == NZUDSUDX_FIXED)
    {
        opString = input->data.pFixedString;
    } else if (input->type == NZUDSUDX_VARIABLE)
    {
        opString = input->data.pVariableString;
```

```
        } else
        {
            nzaeUserError(h, "first input column expected to be a string \
                 type");
            nzaeDone(h);
            return -1;
        }
        const char *ptr = opString;

        char *token;

        while ((token = strtok(ptr, ",")) != NULL) {
            ptr = NULL;

            CHECK(nzaeSetOutputString(h, 0, token));
            CHECK(nzaeOutputResult(h));
        }

    }
    nzaeDone(h);
    return 0;
}
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language system --version 3 \
     --template compile sstring.c --exe sstring
```

Registration

Register the newly created "sstring" executable:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3
     \ --template udtf --exe sstring --sig "sstring_c(VARCHAR(ANY))" \ --
     return "TABLE (val varchar(2000))"
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(sstring_c('12:30:15'));
   VAL
----------
 12:30:15
(1 row)

SELECT * FROM TABLE WITH FINAL(sstring_c('12:30:15,two,three'));
   VAL
----------
 12:30:15
 two
 three
(3 rows)

SELECT * FROM TABLE WITH FINAL(sstring_c(NULL));
ERROR:  first input COLUMN may NOT be NULL
```

C++ Language Table Function

This example uses the following file name:

```
split.cpp
```

Code

The code in this example creates a function that splits a string into multiple strings based on the delimiter. This example may return more rows of output than input. Because of this, it cannot use the handler model and requires additional error handling logic. Note the try catch block in the main and the loop logic in run. Also note that code must be supplied to retrieve the record objects, output the object, and delete the record objects.

```
#include <nzaefactory.hpp>

using namespace nz::ae;
using namespace std;

static int run(nz::ae::NzaeFunction *aeFunc);


int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
       The following line is only needed if a launcher is not
    used helper.setName("testcapi");
    for (int i=0; i < 2; i++) {
        nz::ae::NzaeApi &api = helper.getApi(nz::ae::NzaeApi::FUNCTION);

        try {
            run(api.aeFunction);
        }
        catch (nz::ae::NzaeException &ex) {
            if (api.aeFunction) {
                api.aeFunction->userError(ex.what());
            }
            break;
        }
        if (!helper.isRemote())
            break;
    }

    return 0;
}

static void split_string(string text,char delim, vector<string>& words)
{
    int i=0;
    char ch;
    string word;

    while((ch=text[i++]))
    {
        if (ch == delim)
        {
            if (!word.empty())
            {
                words.push_back(word);
            }
            word = "";
        }
        else
```

```
        {
            word += ch;
        }
    }
    if (!word.empty())
    {
        words.push_back(word);
    }
}

static int run(NzaeFunction *aeFunc)
{

    const NzaeMetadata& meta = aeFunc->getMetadata();
    if (meta.getOutputColumnCount() != 1 ||
        meta.getOutputType(0) != NzaeDataTypes::NZUDSUDX_VARIABLE) {
        throw NzaeException("expecting one output column of type string");
    }
    if (meta.getInputColumnCount() != 1) {
        throw NzaeException("expecting one input column");
    }
    if (meta.getInputType(0) != NzaeDataTypes::NZUDSUDX_FIXED &&
        meta.getInputType(0) != NzaeDataTypes::NZUDSUDX_VARIABLE) {
        throw NzaeException("first input column expected to be a string type");
    }

    for (;;)
    {
        nz::ae::NzaeRecord *input = aeFunc->next();
        if (!input)
            break;
        nz::ae::NzaeRecord *output = aeFunc->createOutputRecord();

        NzaeField &field = input->get(0);
        if (field.isNull())
            throw NzaeException("first input column may not be null");
        NzaeStringField &sf = (NzaeStringField&) input->get(0);
        std::string str = (std::string)sf;
        vector<string> words;
        split_string(str,',', words);
        for (int i=0; i < (int)words.size(); i++) {
            string tip = words[i];
            NzaeStringField &sf = (NzaeStringField&) output->get(0);
            sf = tip;
            aeFunc->outputResult(*output);
        }


        delete input;
        delete output;

    }
    aeFunc->done();
    return 0;
}
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp --template compile
 \ --exe splitae --compargs "-g -Wall" --linkargs "-g" split.cpp --version 3
```

Registration

Register the newly created "splitae" executable:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "splitae(VARCHAR(ANY))"
     \ --return "table(val varchar(2000))" --language cpp --template udtf
     \ --exe splitae --version 3
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH
FINAL(splitae('12:30:15')); VAL
----------
12:30:15
(1 row)

SELECT * FROM TABLE WITH
FINAL(splitae('12:30:15,two,three')); VAL
----------
12:30:15
two
three
(3 rows)
SELECT * FROM TABLE WITH FINAL(splitae(NULL));
ERROR: first input COLUMN may NOT be NULL
```

Java Language Table Function

This example uses the following file name:

```
TestJavaSplit.java
```

Code

The code in this example creates a function that splits a string into multiple strings based on the delimiter. This example may return more rows of output than input. Because of this, it cannot use the handler model and requires additional error handling logic. Note the try catch block and the loop logic in run. Also note that code must be supplied to retrieve the record objects, output the object, and delete the record objects.

```
import org.netezza.ae.*;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class TestJavaSplit {

    private static final Executor exec =
        Executors.newCachedThreadPool();


    public static final void main(String [] args) {
        try {
            mainImpl(args);
        }     catch     (Throwable     t)     {
            System.err.println(t.toString());
            NzaeUtil.logException(t, "main");
```

```
    }
}

public static final void mainImpl(String [] args) {
    NzaeApiGenerator helper = new NzaeApiGenerator();
    while (true) {
        final NzaeApi api = helper.getApi(NzaeApi.FUNCTION);
        if (api.apiType == NzaeApi.FUNCTION) {
            if (!helper.isRemote()) {
                run(api.aeFunction);
                break;
            } else {
                Runnable task = new Runnable() {
                        public void run() {
                            try {
                                TestJavaSplit.run(api.aeFunction);
                            } finally {
                                api.aeFunction.close();
                            }
                        }
                    };
                exec.execute(task);


            }
        }
    }
    helper.close();
}
public static int run(Nzae ae) {
    try {
        int ret =runReal(ae);
        ae.done();
        return ret;
    }
     catch (NzaeException ex) {
        ae.userError(ex.getMessage());
        ae.done();
        throw ex;
    }
    catch (Throwable ex) {
        ae.userError(ex.getMessage());
        ae.done();
        throw new NzaeException(ex.getMessage());
    }
}
public static int runReal(Nzae ae) {
    final NzaeMetadata meta = ae.getMetadata();


    if (meta.getOutputColumnCount() != 1 ||
        meta.getOutputNzType(0) != NzaeDataTypes.NZUDSUDX_VARIABLE
        ) {
        throw new NzaeException("expecting one output columns of type
            "+ "string");
    }
    if (meta.getInputColumnCount() != 1) {
        throw new NzaeException("expecting one input column");
    }
    if (meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_FIXED &&
        meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_VARIABLE) {
        throw new NzaeException("first input column expected to be a "+
            "string type");
    }

    for (;;) {
        NzaeRecord input = ae.next();
```

```
            if (input == null)
                break;
            NzaeRecord output = ae.createOutputRecord();

            String field = (String) input.getField(0);
            if (field == null)
                throw new NzaeException("first input column may not be null");
            String[] words = field.split(",");
            for (int i=0; i < words.length; i++) {
                output.setField(0, words[i]);
                ae.outputResult(output);
            }

            input = null;
            output = null;
        }
        return 0;
    }

}
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java
     \ --template compile TestJavaSplit.java --version 3
```

Registration

Register the newly created file:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "splitae(VARCHAR(ANY))" \
     --return "table(val varchar(2000))" --class AeUdtf --language java \
     --template udtf --version 3 --define "java_class=TestJavaSplit"
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(splitae('12:30:15'));
   VAL
----------
 12:30:15
(1 row)

SELECT * FROM TABLE WITH FINAL(splitae('12:30:15,two,three'));
   VAL
----------
 12:30:15
 two
 three
(3 rows)

SELECT * FROM TABLE WITH FINAL(splitae(NULL));
ERROR:  first input COLUMN may NOT be NULL
```

Fortran Language Table Function

This example uses the following file name:

```
timeSplitMany.f
```

Code

The code in this example creates a table function that takes a variable number of string representations of time, separated by commas (for example "12:30:55,4:20:18”). The function outputs one row for each time-representation; each row displays the time, split into its component parts.

```
      program timeSplitManyProgram
      call nzaeSetConnectionPointName('timeSplitManyRemote')
      call nzaeRun()
      stop
      end

      subroutine nzaeHandleRequest(handle)
      integer hasNext, isNull, onCharacter, result
      character(1) testChar
      character(10) timeString
      character(1000) timeStrings

      hasNext     = -1
      isNull      = 0
      onCharacter = 0
      timeString  = '0'

      timeStrings = '0'

      EXIT IF THERE ARE NO MORE INPUT ROWS.
10call nzaeGetNext(handle, hasNext) if
      (hasNext .eq. 0) then
          goto 30

      endif

      GET OUR INPUT.
      call nzaeGetInputString(handle, 0, timeStrings, isNull)

      SET THE TIME-STRING.
20    timeString = timeStrings(onCharacter+1:onCharacter+9)

      OUTPUT ONE ROW.
      call nzaeSetOutputString(handle, 0, timeString(1:2))
      call nzaeSetOutputString(handle, 1, timeString(4:5))
      call nzaeSetOutputString(handle, 2, timeString(7:8))
      call nzaeOutputResult(handle)

      LOOP AS APPROPRAITE (IF WE FIND MORE
      COMMAS). onCharacter = onCharacter + 9
      if (timeStrings(onCharacter:onCharacter) .eq. ',')
            then goto 20
      endif

      LOOK FOR MORE INPUT

      ROWS. goto 10

      return

      end
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language fortran --version 3
     \ --template compile timeSplitMany.f
```

Registration

Register the newly created "timeSplitMany" executable:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language fortran --version 3
     \ --template udtf --exe timeSplitMany \
     --sig "time_split_many(varchar(1000))" \
     --return "table(hour varchar(2), minute varchar(2), second varchar(2))"
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL (time_split_many('13:22:47,22:12:45,03:22:09'));
 HOUR | MINUTE  | SECOND
      +--------+--------
------13| 22     | 47
 22    | 12     | 45
 03    | 22     | 09
(3 rows)
```

Python Language Table Function

This example uses the following file name:

```
timeSplitMany.py
```

Code

The code in this example creates a table function that takes a variable number of string representations of time, separated by commas (for example "12:30:55,4:20:18”). The function outputs one row for each time-representation; each row displays the time, split into its component parts. Note that you must take control of the I/O for the AE in the def _runUdtf() function.

```
import nzae

class TimeSplitManyAe(nzae.Ae):

    def _runUdtf(self):
        for row in self:
            for timeString in row[0].split(","):
                self.output(timeString.split(":"))

TimeSplitManyAe.run()
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language python64
     \ --template deploy ./timeSplitMany.py --version 3
```

Registration

Register the table function AE:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3
     \ --template udtf --exe timeSplitMany.py \
     --sig "time_split_many(varchar(1000))" \
     --return "table(hour varchar(2), minute varchar(2), second varchar(2))"
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL (time_split_many('13:22:47,22:12:45,03:22:09'));
 HOUR | MINUTE | SECOND
      +--------+--------
------13| 22     | 47
 22    | 12     | 45
 03    | 22     | 09
(3 rows)
```

Perl Language Table Function

This example uses the following file name:

```
TimeSplitMany.pm
```

Code

The code in this example creates a table function that takes a variable number of string representations of time, separated by commas (for example "12:30:55,4:20:18”). The function outputs one row for each time-representation; each row displays the time, split into its component parts. Note that you must take control of the I/O for the AE in the sub _runUdtf()function.

```
package TimeSplitMany;

use nzae::Ae;
use strict;
use autodie;

our @ISA = qw(nzae::Ae);

my $ae = TimeSplitMany->new();
$ae->run();

sub runUdtf(@)
{
    my $self = shift;
```

```
    while ($self->getNext())
    {
        my @row = $self->getInputRow();
        if ( scalar( @row ) > 0 )
        {
            my @timeArray = split(/,/, $row[0]);

            if ( scalar( @timeArray ) > 0 )
            {
               for (my $i = 0; $i <= $#timeArray; $i++)
              {

                my @time = split(/:/, $timeArray[$i]);
                if ( scalar( @time ) > 0 )
                {
                    $self->output( @time );
                }
                else
                {
                    croak(nzae::Exceptions::AeInternalError->new("Error in
splitting input "));
                }
              }
            }
            else
            {
                 croak(nzae::Exceptions::AeInternalError->new("Error in input
"));
            }
        }
        else
        {
            croak(nzae::Exceptions::AeInternalError->new("Error fetching
row"));
        }
    }
}

1;
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language perl --version 3
     \ --template deploy TimeSplitMany.pm
```

Registration

Register the table function AE:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3
     \ --template udtf --exe TimeSplitMany.pm \
     --sig "time_split_many(varchar(1000))" \
     --return "table(hour varchar(2), minute varchar(2), second varchar(2))"
```

Running

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL (time_split_many('13:22:47,22:12:45,03:22:09'));
```

```
HOUR | MINUTE | SECOND
                        +---------
13---------- | 22             | 47
22           | 12             | 45
03           | 22             | 09
(3 rows)
```

R Language Table Function

Code

This example creates a table function that takes a variable number of string representations of time that is separated by commas, for example "12:30:55,4:20:18". The function outputs one row for each time-representation. Each row displays the time split into its component parts.

Enter the following code in the */tmp/split_many.R* file:

```
nz.fun <- function() {
  while(getNext()) {
    input  <- getInputColumn(0)
    chunks <- strsplit(input, ',')[[1]]
    times  <- strsplit(chunks, ':')
    for (tm in times) {
      for (i in seq_along(tm))the following output
        setOutputString(i-1, tm[1])
      outputResult()
    }
  }
}
```

Compilation

Compile the R source file as follows:

```
/nz/export/ae/utilities/bin/compile_ae --language r --version 3 \
    --template compile --user nz --db dev /tmp/split_many.R
```

Registration

Register the Table Function AE as follows:

```
/nz/export/ae/utilities/bin/register_ae --language r --version 3 \
    --template udtf --exe split_many.R --sig "split_many(VARCHAR(20))" \
    --return "TABLE(hour VARCHAR(2), minute VARCHAR(2), second VARCHAR(2))" \
    --db dev --user nz
```

Running

When you run the registered AE, the following sample output is produced:

```
SELECT * from TABLE WITH FINAL(split_many('13:22:47,22:12:45,03:22:09'));
 HOUR | MINUTE  | SECOND
      +--------+--------
------13| 22     | 47
 22    | 12     | 45
 03    | 12     | 45
(3 rows)
```

Additional R Language Table Functions

To better understand the R AE concepts, consider the following examples.

Table Function Example 1

This example creates a function that returns the total length of character input columns and the total sum of all numeric input columns for each input row.

Enter the following code in the */tmp/sampleudtf.R* file:

```
nz.fun <- function () {
  while(getNext()) {
    tl <- 0
    ts <- 0
    for (i in seq(inputColumnCount())) {
      x <- getInputColumn(i-1)
      if (is.character(x)) {
        tl <- tl + nchar(x)
      }
      if (is.numeric(x)) {
        ts <- ts + as.double(x)
      }
    }
    setOutputInt32(0, tl)
    setOutputDouble(1, ts)
    outputResult()
  }
}
```

Compile the file as follows:

```
/nz/export/ae/utilities/bin/compile_ae --language r --version 3 \
     --template compile --user nz --db dev /tmp/sampleudtf.R
```

Register the AE as follows:

```
/nz/export/ae/utilities/bin/register_ae --language r --version 3 --user nz \
     --db dev --template udtf --sig 'rtotal(VARCHAR(1024),DOUBLE)' \
     --return 'TABLE(length INT4, sum DOUBLE)' --exe sampleudtf.R
```

Run the UDTF as follows:

```
SELECT * FROM TABLE WITH FINAL(rtotal('text', 1));
 LENGTH | SUM
--------+-----
      4 |   1
(1 row)
```

Table Function Example 2

This example shows multiple output rows. Because the total length of character columns is an integer, and the total sum of numeric is of type double, choose the most flexible data type for the second output column based on the data types of the inputs, in this case double.

The R code differs slightly for this example.

Enter the following code in the */tmp/sampleudtf.R* file:

```
nz.fun <- function () {
  while(getNext()) {
    tl <- 0
    ts <- 0
    for (i in seq(inputColumnCount())) {
      x <- getInputColumn(i-1)
      if (is.character(x)) {
        tl <- tl + nchar(x)
      }
      if (is.numeric(x)) {
        ts <- ts + as.double(x)
      }
    }
    setOutputString(0, 'total length of character columns')
    setOutputDouble(1, tl)
    outputResult()
    setOutputString(0, 'total sum of numerics')
    setOutputDouble(1, ts)
    outputResult()
  }
}
```

Compile the file as follows:

```
/nz/export/ae/utilities/bin/compile_ae --language r \
  --version 3 --template compile --user nz --db dev \
  /tmp/sampleudtf.R
```

Register the AE as follows. Due to the difference in output, you must specify a different output signature during the registration.

```
/nz/export/ae/utilities/bin/register_ae --language r \
  --version 3 --user nz --db dev --template udtf \ --
  sig 'rtotal(VARCHAR(1024),DOUBLE)' \
  --return 'TABLE(name VARCHAR(64), value DOUBLE)' \
  --exe sampleudtf.R
```

When you call this function, the following output is produced:

```
SELECT * FROM TABLE WITH FINAL(rtotal('text', 1));
               NAME                 | VALUE
                                    +
 length of character columns        |-------4
 total sum of numerics              |       1
(2 rows)
```

Table Function Example 3

By changing the registration command from the previous example, you can create a function of dynamic input signature by using VARARGS.

The R code and compilation steps are the same as in the previous example.

Register the AE as follows:

```
/nz/export/ae/utilities/bin/register_ae --language r --version 3 \
  --user nz --db dev --template udtf --sig 'rtotal(VARARGS)' \
  --exe sampleudtf.R --return 'TABLE(name VARCHAR(64), value DOUBLE)'
```

You can now pass any number of columns to R:

```
SELECT * FROM TABLE WITH FINAL(rtotal('text', 1, 'second text',
     2, 3, 4, 'the last text'));
               NAME                 | VALUE
------------------------------------+
 total length of character columns | -------28
 total sum of numerics             |     10
(2 rows)
```

CHAPTER 9

Shapers and Sizers Examples

Often when dealing with functions, you want the shape or size of the output to be dependent on the input. Shapers and sizers enable you to specify what the AE is going to return (that is, the output schema). For more information, see Introduction to Shapers and Sizers.

Concepts covered in this section include:

- Scalar functions
- Simple table functions
- Shapers and sizers

C Language Shapers and Sizers

This example uses the following file name:

```
sstring.c
```

Code

The code in this example demonstrates both a shaper and sizer, each of which use the same code and perform essentially the same task. The code returns as output the given input. It does not demonstrate retrieving the literal fields, just the shaper/sizer functionality. This example uses handlers for both the function logic and the shaper logic.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "nzaeapis.h"

static int run(NZAE_HANDLE h);
static int runShaper(NZAESHP_HANDLE h);

int main(int argc, char * argv[])
{

    if (nzaeIsLocal())
```

```
{
    NzaeApi result;
    char errorMessage[1050];
    if (nzaeLocprotGetApi(&result, NZAE_LDK_VERSION,
                          errorMessage, sizeof(errorMessage)))
    {
        fprintf(stderr, "%s\n", errorMessage);
        return -1;
    }
    if (result.apiType == NZAE_API_FUNCTION) {
        run(result.handle.function);
        nzaeClose(result.handle.function);
    }
    else {
        runShaper(result.handle.shaper);
        nzaeShpClose(result.handle.shaper);
    }
}
else {
    NZAECONPT_HANDLE hConpt = nzaeconptCreate();
    if (!hConpt)
    {
        fprintf(stderr, "error creating connection point\n");
        fflush(stderr);
        return -1;
    }
    const char * conPtName = nzaeRemprotGetRemoteName();
    if (!conPtName)
    {
        fprintf(stderr, "error getting connection point name\n");
        fflush(stderr);
        exit(-1);
    }
    if (nzaeconptSetName(hConpt, conPtName))
    {
        fprintf(stderr, "error setting connection point name\n");
        fflush(stderr);
        nzaeconptClose(hConpt);
        return -1;
    }
    NzaeremprotInitialization args;
    memset(&args, 0, sizeof(args));
    args.ldkVersion = NZAE_LDK_VERSION;
    args.hConpt = hConpt;
    if (nzaeRemprotCreateListener(&args))
    {
        fprintf(stderr, "unable to create listener - %s\n", \
             args.errorMessage);
        fflush(stderr);
        nzaeconptClose(hConpt);
        return -1;
    }
    NZAEREMPROT_HANDLE hRemprot = args.handle;
    NzaeApi api;
    int i;
    for (i = 0; i < 5; i++)
    {
        if (nzaeRemprotAcceptApi(hRemprot, &api))
        {
            fprintf(stderr, "unable to accept API - %s\n", \
                 nzaeRemprotGetLastErrorText(hRemprot));
            fflush(stderr);
            nzaeconptClose(hConpt);
            nzaeRemprotClose(hRemprot);
            return -1;
        }
        printf("testcapi: accepted a remote request\n");
        fflush(stdout);
```

```
            if (api.apiType == NZAE_API_FUNCTION) {
                run(api.handle.function);
                nzaeClose(api.handle.function);
            }
            else {
                runShaper(api.handle.shaper);
                nzaeShpClose(api.handle.shaper);
            }
        }
        nzaeRemprotClose(hRemprot);
        nzaeconptClose(hConpt);
    }
    return 0;
}

static int runShaper(NZAESHP_HANDLE h)
{
    char name[2];
    bool upper = true;
    NzaeShpMetadata meta;

    #define CHECK2(value) \
    { \
        NzaeRcCode rc = value; \
        if (rc) \
        { \
            const char * format = "%s in %s at %d"; \
            fprintf(stderr, format,  \
                nzaeShpGetLastErrorText(h), __FILE__, __LINE__); \
            \ nzaeShpUserError(h, format, \
                nzaeShpGetLastErrorText(h), __FILE__, __LINE__); \
            \ exit(-1); \
        } \
    }

    CHECK2(nzaeShpGetMetadata(h, &meta));
    if (!meta.oneOutputRowRestriction)
        CHECK2(nzaeShpSystemCatalogIsUpper(h, &upper));
    if (upper)
        name[0] = 'I';
    else
        name[0] = 'i';
    name[1] = 0;
    if (meta.inputTypes[0] == NZUDSUDX_FIXED || meta.inputTypes[0] ==
        NZUDSUDX_VARIABLE || meta.inputTypes[0] == NZUDSUDX_NATIONAL_FIXED
        || meta.inputTypes[0] == NZUDSUDX_NATIONAL_VARIABLE) {
        CHECK2(nzaeShpAddOutputColumnString(h, meta.inputTypes[0], name, \

             meta.inputSizes[0]));
    }
    else if (meta.inputTypes[0] == NZUDSUDX_NUMERIC128 ||
             meta.inputTypes[0] == NZUDSUDX_NUMERIC64 ||
             meta.inputTypes[0] == NZUDSUDX_NUMERIC32) {
        CHECK2(nzaeShpAddOutputColumnNumeric(h, meta.inputTypes[0], name, \
             meta.inputSizes[0], meta.inputScales[0]));
        }
        else {
            CHECK2(nzaeShpAddOutputColumn(h, meta.inputTypes[0], name));
        }

    CHECK2(nzaeShpUpdate(h));
}

static int run(NZAE_HANDLE h)
{
    NzaeMetadata metadata;
    if (nzaeGetMetadata(h, &metadata))
    {
```

```
        fprintf(stderr, "get metadata failed\n");
        return -1;
    }

    #define CHECK(value) \
    { \
        NzaeRcCode rc = value; \
        if (rc) \
        { \
            const char * format = "%s in %s at %d"; \
            fprintf(stderr, format,  \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            nzaeUserError(h, format, \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            exit(-1); \
        } \
    }

    for (;;)
    {
        int i;
        double result = 0;
        NzaeRcCode rc = nzaeGetNext(h);
        if (rc == NZAE_RC_END)
        {
            break;
        }

        NzudsData * input = NULL;
        CHECK(nzaeGetInputColumn(h, 0, &input));
        const char * opString;
        if (input->isNull)
        {
            nzaeSetOutputNull(h,0);
        }
        else {
            nzaeSetOutputColumn(h,0, input);
        }
        CHECK(nzaeOutputResult(h));
    }
    nzaeDone(h);
    return 0;
}
```

Note in the main that an API object must be retrieved in local mode, which gets both a function API and a shaper API (although not at the same time). Note also that there is a new function to handle the shaper/sizer.

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language system --version 3 \
     --template compile sstring.c --exe shaper
```

Registration

Register the example as a UDF and a UDTF:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3 \
     --template udtf --exe shaper --sig "tshaper_c(VARARGS)" \
     --return "TABLE (ANY)"

$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3 \
```

```
--template udf --exe shaper --sig "sizer_c(VARCHAR(ANY))"
\ --return "VARCHAR(ANY)"
```

Running

Run the query in nzsql. The code returns as output the given input:

```
SELECT sizer_c('test');
 SIZER_C
---------
 test
(1 row)

SELECT * FROM TABLE WITH FINAL(tshaper_c('test'));
 I
---

(1 row)

SELECT * FROM TABLE WITH FINAL(tshaper_c(1.1));
  I
-----
 1.1
(1 row)
```

C++ Language Shapers and Sizers

This example uses the following file name:

```
shaper.cpp
```

Code

The code in this example demonstrates both a shaper and sizer, each of which use the same code and perform essentially the same task. The code returns as output the given input. It does not demonstrate retrieving the literal fields, just the shaper/sizer functionality. This example uses handlers for both the function logic and the shaper logic. Note that you could choose not to use a handler for the shaper, but there is no reason to do so.

```
#include <nzaefactory.hpp>

using namespace nz::ae;

static int run(nz::ae::NzaeFunction *aeFunc);
static int doShaper(nz::ae::NzaeShaper *aeShaper);


int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
       The following line is only needed if a launcher is not
    used helper.setName("testcapi");
    int i = 0;

    while (i < 1) {
        nz::ae::NzaeApi &api = helper.getApi(nz::ae::NzaeApi::ANY); if
        (api.apiType == nz::ae::NzaeApi::FUNCTION) {
            run(api.aeFunction);
            i++;
        }
        else {
```

```
            doShaper(api.aeShaper);
        }
        if (!helper.isRemote())
            break;
    }

    return 0;
}

class MyHandler : public nz::ae::NzaeFunctionMessageHandler
{
public:
    void evaluate(NzaeFunction& api, NzaeRecord &input, NzaeRecord &result)
        { const NzaeMetadata& m = api.getMetadata();
        nz::ae::NzaeField &field = input.get(0);
        if (!field.isNull() ) {
            nz::ae::NzaeField &f = result.get(0);
            if (field.type() == NzaeDataTypes::NZUDSUDX_NUMERIC32 ||
                field.type() == NzaeDataTypes::NZUDSUDX_NUMERIC64 ||
                field.type() == NzaeDataTypes::NZUDSUDX_NUMERIC128)
            {
                NzaeNumericField &nf = (NzaeNumericField&)field;
                if (m.getInputSize(0) != nf.precision() ||
                    m.getInputScale(0) != nf.scale()) {
                    NzaeNumeric128Field* np = nf.toNumeric128(38, 5);
                    f.assign(*np);
                    delete np;
                }
                else
                    f.assign(field);
            }
            else {
                f.assign(field);
            }
        }

    }

};

class MyHandler2 : public nz::ae::NzaeShaperMessageHandler
{
public:
    void shaper(NzaeShaper& api){
        const NzaeMetadata& m =  api.getMetadata();
        char name[2];
        if (api.catalogIsUpper())
            name[0] = 'I';
        else
            name[0] = 'i';
        name[1] = 0;

        if (m.getInputType(0) == NzaeDataTypes::NZUDSUDX_FIXED ||
            m.getInputType(0) == NzaeDataTypes::NZUDSUDX_VARIABLE ||
            m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NATIONAL_FIXED ||
            m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NATIONAL_VARIABLE)
            { api.addOutputColumnString(m.getInputType(0), name,
m.getInputSize(0));
        }
        else if (m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NUMERIC128 ||
                 m.getInputType(0)  == NzaeDataTypes::NZUDSUDX_NUMERIC64  ||
                 m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NUMERIC32) {
            api.addOutputColumnNumeric(m.getInputType(0), name, \
                 m.getInputSize(0), m.getInputScale(0));
        }
        else {
            api.addOutputColumn(m.getInputType(0), name);
        }
```

```
    }
};

static int doShaper(nz::ae::NzaeShaper *aeShaper)
{
    aeShaper->run(new MyHandler2());
    return 0;
}


static int run(nz::ae::NzaeFunction *aeFunc)
{
    aeFunc->run(new MyHandler());
    return 0;
}
```

Note in the main that an API object must be retrieved in local mode, which gets both a function API and a shaper API (although not at the same time). Additionally, the code modifies the loop logic for remote mode so that the shaper is not incremented, allowing the same instance of the program to handle both a shaper and a function call in remote mode before exiting. Finally, it adds the shaper message handler.

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp --template compile
     \ --exe shaperae --compargs "-g -Wall" --linkargs "-g" shaper.cpp \ --
     version 3
```

Registration

Register the example as a UDF and a UDTF:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "shaperae(VARARGS)" \
     --return "table(ANY)"  --language cpp --template udtf --exe shaperae \
     --version 3

$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "sizerae(VARCHAR(ANY))" \
     --return "varchar(ANY)" --language cpp --template udf --exe shaperae \
     --version 3
```

Running

Run the query in nzsql:

```
SELECT sizerae('test');
 SIZERAE
---------
 test
(1 row)

SELECT * FROM TABLE WITH FINAL(shaperae('test'));
  I
------
 test
(1 row)

SELECT * FROM TABLE WITH FINAL(shaperae(1.1));
  I
-----
 1.1
(1 row)
```

Java Language Shapers and Sizers

This example uses the following file name:

```
TestJavaIdentity.java
```

Code

The code in this example demonstrates both a shaper and sizer, each of which use the same code and perform essentially the same task. The code returns as output the given input. It does not demonstrate retrieving the literal fields, just the shaper/sizer functionality. This example uses handlers for both the function logic and the shaper logic.

```
import java.io.*;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import org.netezza.ae.*;

public class TestJavaIdentity {

    private static final Executor exec =
        Executors.newCachedThreadPool();

    public static final void main(String [] args) {
        try {
            mainImpl(args);
        } catch (Throwable t) {
            System.err.println(t.toString());
            NzaeUtil.logException(t, "main");
        }
    }

    public static final void mainImpl(String [] args) {

        NzaeApiGenerator helper = new NzaeApiGenerator();

        while (true) {
            final NzaeApi api = helper.getApi(NzaeApi.ANY);
            if (api.apiType == NzaeApi.FUNCTION) {
                if (!helper.isRemote()) {
```

```
                run(api.aeFunction);
                break;
            } else {
                Runnable task = new Runnable() {
                        public void run() {
                            try {
                                TestJavaIdentity.run(api.aeFunction);
                            } finally {
                                api.aeFunction.close();
                            }
                        }
                    };
                exec.execute(task);
            }
        }
        else {
            if (!helper.isRemote()) {
                doShaper(api.aeShaper);
                break;
            } else {
                Runnable task = new Runnable() {
                        public void run() {
                            try {
                                TestJavaIdentity.doShaper(api.aeShaper);
                            } finally {
                                api.aeShaper.close();
                            }
                        }
                    };
                exec.execute(task);
            }
        }
    }
    helper.close();
}

public static class MyHandler2 implements NzaeShaperMessageHandler
{

   public void shaper(NzaeShaper api){
       NzaeMetadata m =  api.getMetadata();
       String name;
       if (api.catalogIsUpper())
           name = "I";
       else
           name = "i";

       if (m.getInputNzType(0) == NzaeDataTypes.NZUDSUDX_FIXED ||
          m.getInputNzType(0) == NzaeDataTypes.NZUDSUDX_VARIABLE ||
          m.getInputNzType(0) == NzaeDataTypes.NZUDSUDX_NATIONAL_FIXED ||
          m.getInputNzType(0) == NzaeDataTypes.NZUDSUDX_NATIONAL_VARIABLE) {
          api.addOutputColumnString(m.getInputNzType(0), name,
             m.getInputSize(0));
       }
       else if (m.getInputNzType(0) == NzaeDataTypes.NZUDSUDX_NUMERIC128 ||
                m.getInputNzType(0) == NzaeDataTypes.NZUDSUDX_NUMERIC64 ||
                m.getInputNzType(0) == NzaeDataTypes.NZUDSUDX_NUMERIC32) {
            api.addOutputColumnNumeric(m.getInputNzType(0), name,
                m.getInputSize(0), m.getInputScale(0));
       }
       else {
           api.addOutputColumn(m.getInputNzType(0), name);
       }
    }
}

public static class MyHandler implements NzaeMessageHandler
{
```

```
        public void evaluate(Nzae ae, NzaeRecord input, NzaeRecord output)
            { output.setField(0, input.getField(0));
        }
    }

    private static final void run(Nzae ae) {
        ae.run(new MyHandler());
    }

    private static final void doShaper(NzaeShaper aeShaper)
    {
        aeShaper.run(new MyHandler2());
    }
}
```

Note in the main that an API object must be retrieved in local mode, which gets both a function API and a shaper API (although not at the same time). Additionally, the code modifies the loop logic for remote mode so that the shaper is not incremented, allowing the same instance of the program to handle a shaper and a function call in remote mode before exiting. Finally, it adds the shaper message handler.

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java --template compile
     \ TestJavaIdentity.java --version 3
```

Registration

Register the example as a UDF and a UDTF:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "shaperae(varargs)" \
     --return "table(any)" --class AeUdtf --language java --template udtf \
     --version 3 --define "java_class=TestJavaIdentity"

$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "sizerae(varargs)" \
     --return "varchar(any)" --class AeUdf --language java --template udf \
     --version 3 --define "java_class=TestJavaIdentity"
```

Running

Run the query in nzsql. It returns as output the given input:

```
SELECT sizerae('test');
 SIZERAE
---------
 test
(1 row)

SELECT * FROM TABLE WITH FINAL(shaperae('test'));
  I
------
 test
(1 row)

SELECT * FROM TABLE WITH FINAL(shaperae(1.1));
  I
-----
 1.1
(1 row)
```

Fortran Language Shapers and Sizers

This example uses the following file name:

```
shaperSizer.f
```

Code

The code in this example demonstrates both a shaper and sizer, each of which use the same code and perform essentially the same task. The code returns as output the given input. It does not demonstrate retrieving the literal fields, just the shaper/sizer functionality. This example uses handlers for both the function logic and the shaper logic.

```
program shaperSizerProgram
call nzaeRun()
stop
end

subroutine nzaeHandleRequest(handle)
integer isShaper
isShaper = -1

DETERMINE IF WE ARE RUNNING IN SHAPER MODE.
CALL THE APPROPRIATE SUBROUTINE. call
nzaeIsShaper(handle, isShaper) if
(isShaper .eq. 1) then
      call runShaper(handle)
else if (isShaper .eq. 0) then
      call run(handle)
else
      call nzaeUserError(handle, "nzaeIsShaper() failed.")
endif
return
end

subroutine runShaper(handle)
integer isUpper, isUdf, test, inputType, inputSize, inputScale
character(1) name
isUpper           = 1
isUdf             = -1
test              = -1
inputType         = -1
inputSize         = -1
inputScale        = -1

DETERMINE OUTPUT COLUMN NAME.
call nzaeIsShaper(handle, isUdf)
if (isUdf .eq. 1) then
      call nzaeIsSystemCatalogUpperCase(handle, isUpper)
endif
if (isUpper .eq. 1) then
      name = 'I'
else
      name = 'i'
endif

COPY STRING TYPES.
call nzaeGetInputType(handle, 0, inputType)
call nzaeIsAStringType(inputType, test)
if (test .eq. 1) then
      call nzaeGetInputSize(handle, 0, inputSize)
      call nzaeAddOutputColumnString(handle,
+                                    inputType,
+                                    name,
```

```
     +                                        inputSize)
           return
      endif

      COPY NUMERIC TYPES.
      call nzaeIsANumericType(inputType,
      test) if (test .eq. 1) then
            call nzaeGetInputSize(handle, 0, inputSize)
            call nzaeGetInputScale(handle, 0, inputPrecision)
            call nzaeAddOutputColumnNumeric(handle,
      +                                      inputType,
      +                                      name,
      +                                      inputSize
      +                                      inputScale)
            return
      endif

      COPY ALL OTHER TYPES.
      call nzaeAddOutputColumn(handle, inputType, name)
      return
      end

      subroutine run(handle)
      integer hasNext, leftInput, rightInput, result
      character operator
      hasNext    = -1
      leftInput  = 0
      rightInput = 0
      operator   = 'f'

      call nzaeGetNext(handle, hasNext)
      if (hasNext .eq. 0) then
          goto 20

      endif

      COPY OVER THE INPUT COLUMN.
      call nzaeOutputInputColumn(handle, 0,

      0) call nzaeOutputResult(handle)

      LOOP GETTING THE NEXT ROW.

      goto 10

      return

      end
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language fortran --version 3
     \ --template compile shaperSizer.f
```

Registration

Register the example as a UDF and a UDTF:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language fortran --version 3
  \ --template udtf --exe shaperSizer --sig "shaper_sizer_table(VARARGS)" \
  --return "table(ANY)"
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language fortran --version 3
  \ --template udf --exe shaperSizer --sig "shaper_sizer(VARCHAR(ANY))" \ -
  -return "VARCHAR(ANY)"
```

Running

Run the query in nzsql. It returns as output the given input:

```
SELECT shaper_sizer('test');
 SHAPER_SIZER
--------------
 test
(1 row)

SELECT * FROM TABLE WITH FINAL(shaper_sizer_table('test'));
  I
------
 test
(1 row)

SELECT * FROM TABLE WITH FINAL(shaper_sizer_table(1.1));
  I
-----
 1.1
(1 row)
```

Python Language Shapers and Sizers

This example uses the following file name:

```
shaperSizer.py
```

Code

The code in this example demonstrates both a shaper and sizer, each of which use the same code and perform essentially the same task. The code returns as output the given input. It does not demonstrate retrieving the literal fields, just the shaper/sizer functionality.

```
import nzae

class ShaperSizerAe(nzae.Ae):

    def _runSizer(self):
        self._runShaper()

    def _runShaper(self):

           GET THE INPUT TYPE.
        inputType   = self.getInputType(0)
        columnName  = self.getDataTypeName(inputType)

           HANDLE STRINGS.
        if inputType in self.STRING_DATA_TYPES:
            size = self.getInputSize(0)
            self.addOutputColumnString(columnName, inputType, size)

           HANDLE NUMERICS.
        elif inputType in self.NUMERIC_DATA_TYPES:
            precision = self.getInputPrecision(0)
            scale     = self.getInputScale(0)
            self.addOutputColumnNumeric(columnName, inputType, precision, scale)

           HANDLE OTHER.
        else:
            self.addOutputColumn(columnName, inputType)

    def _getFunctionResult(self, row):
```

```
        return row

ShaperSizerAe.run()
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language python64
     \ --template deploy ./shaperSizer.py --version 3
```

Registration

Register the example as a UDF and a UDTF:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3
  \ --template udf --exe shaperSizer.py --sig "sizer(varchar(ANY))" \ --
  return "varchar(ANY)"
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3
  \ --template udtf --exe shaperSizer.py --sig "shaper(VARARGS)" \
  --return "table(ANY)"
```

Running

Run the query in nzsql. It returns as output the given input:

```
SELECT sizer('test');
 SIZER
-------
 test
(1 row)

SELECT * FROM TABLE WITH FINAL(shaper('test'));
 NATIONAL_VARIABLE
-------------------
 test
(1 row)

SELECT * FROM TABLE WITH FINAL(shaper(32.1));
 NUMERIC32
-----------
      32.1
(1 row)

SELECT * FROM TABLE WITH FINAL(shaper(42));
 INT32
-------
    42
(1 row)
```

Perl Language Shapers and Sizers

This example uses the following file name:

```
ShaperSizerAe.pm
```

Code

The code in this example demonstrates both a shaper and sizer, each of which use the same code and perform essentially the same task. The code returns as output the given input. It does not demonstrate retrieving the literal fields, just the shaper/sizer functionality.

```
package ShaperSizerAe;
use nzae::Ae;
use strict 'vars';
use autodie;

our @ISA = qw(nzae::Ae);
my $self = ShaperSizerAe->new();
$self->run();

sub runSizer()
{
    my $self = shift;
    $self->runShaper(@_);
}

sub runShaper()
{
    my $self = shift;
    my $inputType = $self->getInputType(0);
    my $columnName = $self->getDataTypeName($inputType); my
    $stringHash = $self->getDataStringHash();
    my $numericHash = $self->getDataNumericHash();

    if ( exists $stringHash->{$inputType} )
    {
        my $size = $self->getInputSize(0);
        return $self->addOutputColumnString($columnName, $inputType, $size);
    }
    elsif ( exists $numericHash->{$inputType} )
    {
        my $precision = $self->getInputPrecision(0);
        my $scale = $self->getInputScale(0);
        $self->addOutputColumnNumeric($columnName, $inputType, $precision,
$scale);
    }
    else
    {
        $self->addOutputColumn($columnName, $inputType);
    }
}

sub _getFunctionResult()
{
    my $self = shift;
    return @_;
}

1;
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language perl --version 3
     \ --template deploy ShaperSizerAe.pm
```

Registration

Register the example as a UDF and a UDTF

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3 \
     --template udf --exe ShaperSizerAe.pm --sig "sizerPl(varchar(ANY))"
     \ --return "varchar(ANY)"
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3
     \ --template udtf --exe ShaperSizerAe.pm --sig "shaperPl(VARARGS)"
     \ --return "table(ANY)"
```

R Language Shapers and Sizers

Using a Shaper

You can access the shaper functionality in the following ways:

When a new UDTF is registered in one of the following ways:

Setting the nz.shaper variable to a function by using the API described above

Setting the nz.shaper variable to the std character value, and setting the nz.shaper.list variable to a list of which the elements follow the pattern *column-name*=NZ.*data-type*

When the CODE_PLAIN channel is used, by setting the additional SHAPER_LIST Dynamic Environment variable so that it follows the same pattern of *column-name*=NZ.*data-type* Subsequent occurrences of this pattern are separated by commas.

By setting the output.signature parameter in some of the *nzLibrary for R* package functions

Code

This example demonstrates a shaper and a sizer that use the same code and do essentially the same task, returning the given input as output. This example does not demonstrate the retrieval of literal fields.

Put the following code in the */tmp/shapersizer.R* file:

```
nz.shaper <- function() {
     getInputColumnInfo returns a vector (type,isConstant,size,scale)
     the last two elements only if applicable
  inputType <- getInputColumnInfo(0)[1]
  columnName <- names(which(getNpsDataTypes() == inputType))
     HANDLE STRINGS.
  if (inputType %in% c(NZ.FIXED, NZ.VARIABLE, NZ.NATIONAL_FIXED,
        NZ.NATIONAL_VARIABLE)) {
    size <- getInputColumnInfo(0)[3]
    addOutputColumnString(inputType, columnName, size)
  }
     HANDLE NUMERICS.
  else if (inputType %in% c(NZ.NUMERIC32, NZ.NUMERIC64, NZ.NUMERIC128))
    { stop('numeric data types are not supported in R')
    # precision <- getInputColumnInfo(0)[3]
    # scale     <- getInputColumnInfo(0)[4]
    # addOutputColumnNumeric(inputType, columnName, precision, scale)
  }
  #  HANDLE OTHER.
```

```
  else
    addOutputColumn(inputType, columnName)
  updateInfo()
}

nz.fun <- function() {
  while(getNext()) {
    setOutput(0, getInputColumn(0))
    outputResult()
  }
}
```

Compilation

Compile the code as follows:

```
/nz/export/ae/utilities/bin/compile_ae --language r --version 3 \
  --template compile --user nz --db dev /tmp/shapersizer.R
```

Registration

Register the example as a UDF and a UDTF:

```
/nz/export/ae/utilities/bin/register_ae --language r --version 3 \
  --template udf --exe shapersizer.R --sig "sizer(VARCHAR(ANY))" \
  --return "VARCHAR(ANY)" --user nz --db dev

/nz/export/ae/utilities/bin/register_ae --language r --version 3 \
  --template udtf --exe shapersizer.R --sig "shaper(VARARGS)" \
  --return "TABLE(ANY)" --user nz --db dev
```

Running

When you run the AE , you get the following results:

```
SELECT sizer('test');
 SIZER
-------
 test
(1 row)
SELECT * FROM TABLE WITH FINAL(shaper('test'));
 NATIONAL_VARIABLE
-------------------
 test
(1 row)
SELECT * FROM TABLE WITH FINAL(shaper(32.1::double));
 DOUBLE
--------
   32.1
(1 row)
SELECT * FROM TABLE WITH FINAL(shaper(42));
 INT32
-------
    42
(1 row)
```

Another Example

This example must be compiled and registered as outputting TABLE(ANY).

Put the following code in the */tmp/shapersizer2.R* file:

```
nz.fun <- function () {
  while(getNext()) {
    setOutput(0, sqrt(as.double(getInputColumn(0))))
    outputResult()
  }
}
nz.shaper <- function() {
  addOutputColumn(NZ.DOUBLE,'value')
  updateInfo()
}
```

Compilation

Compile the code as follows:

```
/nz/export/ae/utilities/bin/compile_ae --language r --version 3 \
  --template compile /tmp/shapersizer2.R
```

Registration

Register the example as a UDTF:

```
/nz/export/ae/utilities/bin/register_ae  --language  r  --version  3
  \ --template udtf --exe shapersizer2.R --sig "shaper2(VARARGS)"
  \ --return "TABLE(ANY)"
```

Running the following query returns the same results as the previous example:

```
SELECT * FROM nza..iris, TABLE WITH FINAL(nzr..r_udtf_any(sepallength,
  'CODE_PLAIN="while(getNext()){
     setOutput(0, sqrt(as.double(getInputColumn(0)))); outputResult()}",
   SHAPER_LIST="value=NZ.DOUBLE",NZAE_ROW_BUFFER=no'))
```

To keep the connection between the input and output rows, buffering must be turned off. For more information about row buffering, see Row Buffering.

Running

Run the query in nzsql. It returns as output the given input:

```
SELECT sizerPl('test');
SIZER
-------
test
(1 row)

SELECT * FROM TABLE WITH FINAL(shaperPl('test'));
NATIONAL_VARIABLE
-------------------
test
(1 row)

SELECT * FROM TABLE WITH FINAL(shaperPl(32.1));
NUMERIC32
-----------
32.1
(1 row)

SELECT * FROM TABLE WITH FINAL(shaperPl(42));
INT32
-------
42
(1 row)
```

CHAPTER 10
Aggregate AE Examples

The aggregate AE examples show an aggregate function that returns the maximum value from a set of values (except for Fortran, which returns the second highest value). The examples use four methods: **accumulate**, **initializeState**, **merge** and **finalResult**. These methods mirror the standard UDA methods.

In the example, the AE makes a call to determine the next aggregation state from the API. Depending on the next aggregation state, the initialization, accumulation, merge or finalResult functions are called. These functions must be overridden by a user while implementing an aggregate AE. The initialization function sets the state variable as defined in the registration step to a default value. The accumulate step compares the first column of each row with the state variable. If the column value is greater than the state variable, the state variable is set to the column value. The merge step merges the state variable collected from each dataslice and the final result returns the merged state variable.

C Language Aggregates

This example uses the following file name:

```
max.c
```

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "nzaeapis.h"

static int run(NZAEAGG_HANDLE h);

int main(int argc, char * argv[])
{
     if (nzaeIsLocal())
    {
        NzaeApi result;
        char errorMessage[1050];
        if (nzaeLocprotGetApi(&result, NZAE_LDK_VERSION, \
```

```
            errorMessage, sizeof(errorMessage)))
        {
            fprintf(stderr, "%s\n", errorMessage);
            return -1;
        }
        if (result.apiType == NZAE_API_AGGREGATION) {
            run(result.handle.aggregation);
            nzaeAggClose(result.handle.aggregation);
        }
        else {
            fprintf(stderr, "unexpected API returned\n");
            fflush(stderr);
            return -1;
        }
    }
    else {
        NZAECONPT_HANDLE hConpt = nzaeconptCreate();
        if (!hConpt)
        {
            fprintf(stderr, "error creating connection point\n");
            fflush(stderr);
            return -1;
        }
        const char * conPtName = nzaeRemprotGetRemoteName();
        if (!conPtName)
        {
            fprintf(stderr, "error getting connection point name\n");
            fflush(stderr);
            exit(-1);
        }
        if (nzaeconptSetName(hConpt, conPtName))
        {
            fprintf(stderr, "error setting connection point name\n");
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NzaeremprotInitialization args;
        memset(&args, 0, sizeof(args));
        args.ldkVersion = NZAE_LDK_VERSION;
        args.hConpt = hConpt;
        if (nzaeRemprotCreateListener(&args))
        {
            fprintf(stderr, "unable to create listener - %s\n", \
                 args.errorMessage);
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NZAEREMPROT_HANDLE hRemprot = args.handle;
        NzaeApi api;
        int i;
        for (i = 0; i < 5; i++)
        {
            if (nzaeRemprotAcceptApi(hRemprot, &api))
            {
                fprintf(stderr, "unable to accept API - %s\n", \
                     nzaeRemprotGetLastErrorText(hRemprot));
                fflush(stderr);
                nzaeconptClose(hConpt);
                nzaeRemprotClose(hRemprot);
                return -1;
            }
            printf("testcapi: accepted a remote request\n");
            fflush(stdout);
            if (api.apiType == NZAE_API_AGGREGATION) {
                run(api.handle.aggregation);
                nzaeClose(api.handle.aggregation);
```

```
            }
            else {
                fprintf(stderr, "unexpected API returned\n");
                fflush(stderr);
                nzaeconptClose(hConpt);
                nzaeRemprotClose(hRemprot);
                return -1;
            }
        }
        nzaeRemprotClose(hRemprot);
        nzaeconptClose(hConpt);
    }
    return 0;
}

static int run(NZAEAGG_HANDLE h)
{
    NzaeAggMessageType messageType;
    for (;;)
    {
        void * pTemp = nzaeAggNext(h, &messageType);
        if (messageType == NZAEAGG_END)
        {
            break;
        }
        if (messageType == NZAEAGG_ERROR)
        {
            nzaeAggUserError(h, nzaeAggGetLastErrorText(h));
            break;
        }

        switch (messageType)
        {
            case NZAEAGG_INITIALIZE:
            {
                NzaeAggInitializeState * p = (NzaeAggInitializeState *) pTemp;
                p->state.setNull(h, 0);
                break;
            }
            case NZAEAGG_ACCUMULATE:
            {
                NzaeAggAccumulate * p = (NzaeAggAccumulate *) pTemp;
                NzudsData * inputData;
                NzudsData * stateData;
                p->input.getValue(h, 0, &inputData);
                p->state.getValue(h, 0, &stateData);
                if (inputData->isNull) break;

                if (stateData->isNull || *stateData->data.pInt32 < \
                     *inputData->data.pInt32)
                {
                    p->state.setValue(h, 0, inputData);
                }
                break;
            }
            case NZAEAGG_MERGE:
            {
                NzaeAggMerge * p = (NzaeAggMerge *) pTemp;
                NzudsData * inputStateData;
                NzudsData * stateData;
                p->inputState.getValue(h, 0, &inputStateData);
                p->state.getValue(h, 0, &stateData);
                if (inputStateData->isNull)
                {
                    break;  // no data to merge
                }
                if (stateData->isNull || *stateData->data.pInt32 \
                     < *inputStateData->data.pInt32)
```

```
                    {
                        p->state.setValue(h, 0, inputStateData);
                    }
                    break;
                }
                case NZAEAGG_FINAL_RESULT:
                {
                    NzaeAggFinalResult * p = (NzaeAggFinalResult *) pTemp;
                    NzudsData * inputStateData;
                    p->inputState.getValue(h, 0, &inputStateData);
                    if (inputStateData->isNull)
                    {
                        p->result.setNull(h, 0);
                    } else
                    {

                        p->result.setValue(h, 0, inputStateData);

                    }

                    break;
                }
                default:
                    nzaeAggUserError(h, "unexpected message type");
            }
             nzaeAggUpdate(h);
        }

        return 0;
}
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language system --version 3 \
     --template compile max.c --exe max
```

Registration

Register the example as a UDA, using the **state** parameter:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3 \
     --template uda --exe max --sig "max_c(int4)" --return "int4" \
     --state "(int4)"
```

Running

To run, first create a dummy table and then run the aggregate:

```
CREATE TABLE grp_test (grp int4, val int4);
CREATE TABLE
INSERT INTO grp_test VALUES (1, 1);
INSERT 0 1
INSERT INTO grp_test VALUES (1, 2);
INSERT 0 1
INSERT INTO grp_test VALUES (1, 3);
INSERT 0 1
INSERT INTO grp_test VALUES (2, 4);
INSERT 0 1
SELECT max_c(val) FROM grp_test;
 MAX_C
-------
     4
```

```
(1 row)

SELECT grp, max_c(val) FROM grp_test GROUP BY grp;
 GRP | MAX_C
-----+-------
   1 |     3
   2 |     4
(2 rows)

SELECT grp, max_c(val) over (partition BY grp) FROM grp_test;
 GRP  | MAX_C
------+
    1 |-------3
    1 |       3
    1 |       3
    2 |       4
(4 rows)
```

C++ Language Aggregates

This example uses the following file name:

```
max.cpp
```

Code

The code in this example is slightly longer than the C language aggregate example because the code is designed to handle all possible data types.

```
#include <nzaefactory.hpp>
using namespace nz::ae;

static int run(nz::ae::NzaeAggregate *aeFunc);
int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
       The following line is only needed if a launcher is not
    used helper.setName("testcapi");
    for (int i=0; i < 2; i++) {
        nz::ae::NzaeApi &api = helper.getApi(nz::ae::NzaeApi::AGGREGATION);
        run(api.aeAggregate);
        if (!helper.isRemote())
            break;
    }

    return 0;
}

class MyHandler : public nz::ae::NzaeAggregateMessageHandler
{
public:
    void initializeState(nz::ae::NzaeAggregate& api,
                         nz::ae::NzaeRecord &state) {
        nz::ae::NzaeField &field = state.get(0);
        field.setNull(true);


    }
    void accumulate(nz::ae::NzaeAggregate& api,
                    nz::ae::NzaeRecord &input,
                    nz::ae::NzaeRecord &state) {

        nz::ae::NzaeField &field = input.get(0);
        nz::ae::NzaeField &statefield = state.get(0);
```

```
if (field.isNull())
    return;
if (statefield.isNull())
{
    statefield.setNull(false);
    statefield = field;
}
else {
    switch (field.type()) {
    case NzaeDataTypes::NZUDSUDX_VARIABLE:
    case NzaeDataTypes::NZUDSUDX_FIXED:
    case NzaeDataTypes::NZUDSUDX_NATIONAL_VARIABLE:
    case NzaeDataTypes::NZUDSUDX_NATIONAL_FIXED:
    {
        std::string &s = ((nz::ae::NzaeStringField&)statefield);
        std::string &f = ((nz::ae::NzaeStringField&)field);
        if (s < f)
            statefield = field;
        break;
    }
    case NzaeDataTypes::NZUDSUDX_BOOL:
    {
        if ((int32_t)(bool)((nz::ae::NzaeBoolField&)statefield)
            < (int32_t)(bool)((nz::ae::NzaeBoolField&)field))
        {
            statefield = field;
        }
        break;
    }
    case NzaeDataTypes::NZUDSUDX_INT64:
    {
        if ((int64_t)((nz::ae::NzaeInt64Field&)statefield)
            < (int64_t)((nz::ae::NzaeInt64Field&)field))
        {
            statefield = field;
        }
        break;
    }
    case NzaeDataTypes::NZUDSUDX_INT32:
    {
        if ((int32_t)((nz::ae::NzaeInt32Field&)statefield)
            < (int32_t)((nz::ae::NzaeInt32Field&)field))
        {
            statefield = field;
        }
        break;
    }
    case NzaeDataTypes::NZUDSUDX_INT16:
    {
        if ((int16_t)((nz::ae::NzaeInt16Field&)statefield)
            < (int16_t)((nz::ae::NzaeInt16Field&)field))
        {
            statefield = field;
        }
        break;
    }
    case NzaeDataTypes::NZUDSUDX_INT8:
    {
        if ((int8_t)((nz::ae::NzaeInt8Field&)statefield) <
             (int8_t)((nz::ae::NzaeInt8Field&)field))
        {
            statefield = field;
        }
        break;
    }
    case NzaeDataTypes::NZUDSUDX_FLOAT:
    {
        if ((float)((nz::ae::NzaeFloatField&)statefield) <
```

```
        (float)((nz::ae::NzaeFloatField&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_DOUBLE:
{
    if ((double)((nz::ae::NzaeDoubleField&)statefield)
        < (double)((nz::ae::NzaeDoubleField&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_NUMERIC32:
case NzaeDataTypes::NZUDSUDX_NUMERIC64:
case NzaeDataTypes::NZUDSUDX_NUMERIC128:
{
    NzaeNumericField * n = \
         ((nz::ae::NzaeNumericField&)field).toNumeric128(38,5);
    if (((nz::ae::NzaeNumericField&)statefield) <
        *n)
    {
        statefield = *n;
    }
    delete n;
    break;
}
case NzaeDataTypes::NZUDSUDX_DATE:
{
    if ((int32_t)((nz::ae::NzaeDateField&)statefield) <
        (int32_t)((nz::ae::NzaeDateField&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_TIME:
{
    if ((int64_t)((nz::ae::NzaeTimeField&)statefield) <
        (int64_t)((nz::ae::NzaeTimeField&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_TIMETZ:
{
    if (((nz::ae::NzaeTimeTzField&)statefield) <
        ((nz::ae::NzaeTimeTzField&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_INTERVAL:
{
    if (((nz::ae::NzaeIntervalField&)statefield) <
        ((nz::ae::NzaeIntervalField&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_TIMESTAMP:
{
    if ((int64_t)((nz::ae::NzaeTimestampField&)statefield) <
```

```
                    (int64_t)((nz::ae::NzaeTimestampField&)field))
                {
                    statefield = field;
                }
                break;
            }
            default:
                throw nz::ae::NzaeException("Unexpected type");

            }
        }
    }

    void merge(nz::ae::NzaeAggregate& api,
               nz::ae::NzaeRecord &inputState,
               nz::ae::NzaeRecord &state) {

        nz::ae::NzaeField &field = inputState.get(0);
        nz::ae::NzaeField &statefield = state.get(0);
        if (field.isNull())
            return;
        if (statefield.isNull())
        {
            statefield.setNull(false);
            statefield = field;
        }
        else {
            switch (field.type()) {
            case NzaeDataTypes::NZUDSUDX_VARIABLE:
            case NzaeDataTypes::NZUDSUDX_FIXED:
            case NzaeDataTypes::NZUDSUDX_NATIONAL_VARIABLE:
            case NzaeDataTypes::NZUDSUDX_NATIONAL_FIXED:
            {
                std::string &s = ((nz::ae::NzaeStringField&)statefield);
                std::string &f = ((nz::ae::NzaeStringField&)field);
                if (s < f)
                    statefield = field;
                break;
            }
            case NzaeDataTypes::NZUDSUDX_BOOL:
            {
                if ((int32_t)(bool)((nz::ae::NzaeBoolField&)statefield)
                    < (int32_t)(bool)((nz::ae::NzaeBoolField&)field))
                {
                    statefield = field;
                }
                break;
            }
            case NzaeDataTypes::NZUDSUDX_INT64:
            {
                if ((int64_t)((nz::ae::NzaeInt64Field&)statefield)
                    < (int64_t)((nz::ae::NzaeInt64Field&)field))
                {
                    statefield = field;
                }
                break;
            }
            case NzaeDataTypes::NZUDSUDX_INT32:
            {
                if ((int32_t)((nz::ae::NzaeInt32Field&)statefield)
                    < (int32_t)((nz::ae::NzaeInt32Field&)field))
                {
                    statefield = field;
                }
                break;
            }
            case NzaeDataTypes::NZUDSUDX_INT16:
            {
```

```
    if ((int16_t)((nz::ae::NzaeInt16Field&)statefield)
        < (int16_t)((nz::ae::NzaeInt16Field&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_INT8:
{
    if ((int8_t)((nz::ae::NzaeInt8Field&)statefield) <
        (int8_t)((nz::ae::NzaeInt8Field&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_FLOAT:
{
    if ((float)((nz::ae::NzaeFloatField&)statefield) <
        (float)((nz::ae::NzaeFloatField&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_DOUBLE:
{
    if ((double)((nz::ae::NzaeDoubleField&)statefield)
        < (double)((nz::ae::NzaeDoubleField&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_NUMERIC32:
case NzaeDataTypes::NZUDSUDX_NUMERIC64:
case NzaeDataTypes::NZUDSUDX_NUMERIC128:
{

    if (((nz::ae::NzaeNumericField&)statefield) <
        (nz::ae::NzaeNumericField&)field)
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_DATE:
{
    if ((int32_t)((nz::ae::NzaeDateField&)statefield) <
        (int32_t)((nz::ae::NzaeDateField&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_TIME:
{
    if ((int64_t)((nz::ae::NzaeTimeField&)statefield) <
        (int64_t)((nz::ae::NzaeTimeField&)field))
    {
        statefield = field;
    }
    break;
}
case NzaeDataTypes::NZUDSUDX_TIMETZ:
{
    if (((nz::ae::NzaeTimeTzField&)statefield) <
        ((nz::ae::NzaeTimeTzField&)field))
```

```
                    {
                        statefield = field;
                    }
                    break;
                }
                case NzaeDataTypes::NZUDSUDX_INTERVAL:
                {
                    if (((nz::ae::NzaeIntervalField&)statefield) <
                        ((nz::ae::NzaeIntervalField&)field))
                    {
                        statefield = field;
                    }
                    break;
                }
                case NzaeDataTypes::NZUDSUDX_TIMESTAMP:
                {
                    if ((int64_t)((nz::ae::NzaeTimestampField&)statefield)
                        < (int64_t)((nz::ae::NzaeTimestampField&)field))
                    {
                        statefield = field;
                    }
                    break;
                }
                default:
                    throw nz::ae::NzaeException("Unexpected type");

                }
            }
        }

        void finalResult(nz::ae::NzaeAggregate& api,
                         nz::ae::NzaeRecord &inputState,
                         nz::ae::NzaeRecord &result) {

            nz::ae::NzaeField &field = inputState.get(0);
            nz::ae::NzaeField &res = result.get(0);
            if (field.isNull())
                res.setNull(true);
            else
            {
                 res = field;
             }

        }

};

static int run(nz::ae::NzaeAggregate *aeAgg)
{
    aeAgg->runAggregation(new MyHandler());
    return 0;
}
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp \
     --template compile --exe maxae --compargs "-g -Wall" \ -

     -linkargs "-g" max.cpp --version 3
```

Registration

Register the example as a UDA, using the **state** parameter:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "maxae(int)" \
```

```
--return "int4" --state "(int4)" --language cpp --template uda
\ --exe maxae --version 3
```

Running

To run, first create a dummy table and then run the aggregate:

```
CREATE TABLE grp_test (grp int4, val int4);
CREATE TABLE
INSERT INTO grp_test VALUES (1, 1);
INSERT 0 1
INSERT INTO grp_test VALUES (1, 2);
INSERT 0 1
INSERT INTO grp_test VALUES (1, 3);
INSERT 0 1
INSERT INTO grp_test VALUES (2, 4);
INSERT 0 1
SELECT maxae(val) FROM grp_test;
 MAXAE
-------
      4
(1 row)

SELECT grp, maxae(val) FROM grp_test GROUP BY grp;
 GRP |MAXAE
-----+-------
   1 |     3
   2 |     4
(2 rows)

SELECT grp, maxae(val) over (partition BY grp) FROM grp_test;
 GRP |MAXAE
-----+-------
   1 |     3
   1 |     3
   1 |     3
   2 |     4
(4 rows)
```

Java Language Aggregates

This example uses the following file name:

```
TestJavaMax.java
```

Code

The code in this example is slightly longer than the C language aggregate example, since the code is designed to handle all possible data types.

```
import java.io.*;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import org.netezza.ae.*;

public class TestJavaMax {

    private static final Executor exec =
        Executors.newCachedThreadPool();

    public static final void main(String [] args) {
        try {
            mainImpl(args);
```

```
    } catch (Throwable t) {
        System.err.println(t.toString());
        NzaeUtil.logException(t, "main");
    }
}


public static final void mainImpl(String [] args) {
    NzaeApiGenerator helper = new NzaeApiGenerator();
    while (true) {
        final NzaeApi api = helper.getApi(NzaeApi.AGGREGATION); if
        (api.apiType == NzaeApi.AGGREGATION) {
            if (!helper.isRemote()) {
                run(api.aeAggregate);
                break;
            } else {
                Runnable task = new Runnable() {
                        public void run() {
                            try {
                                TestJavaMax.run(api.aeAggregate);
                            } finally {
                                api.aeAggregate.close();
                            }
                        }
                    };
                exec.execute(task);


            }
        }
    }

    helper.close();


}


public static class MyHandler implements
NzaeAggMessageHandler {
    public void initializeState(NzaeAgg api,
                         NzaeRecord state) {
        state.setField(0, null);

    }
    public void accumulate(NzaeAgg api,
                    NzaeRecord input,
                    NzaeRecord state) {
        Object field = input.getField(0);
        if (field == null)
            return;
        Object statefield = state.getField(0);

        if (statefield == null) {
            statefield = field;
        }
        else {
            if (((Comparable)statefield).compareTo(field) < 0)
                statefield = field;
        }
        state.setField(0, statefield);
    }

    public void merge(NzaeAgg api,
                NzaeRecord inputState,
                NzaeRecord state) {
```

```
            Object field = inputState.getField(0);
            if (field == null)
                return;
            Object statefield = state.getField(0);

            if (statefield == null) {
                statefield = field;
            }
            else {
                if (((Comparable)statefield).compareTo(field) < 0)
                    statefield = field;
            }
            state.setField(0, statefield);
        }

        public void finalResult(NzaeAgg api,
                        NzaeRecord inputState,
                        NzaeRecord result) {

            Object field = inputState.getField(0);
            result.setField(0, field);
        }
    }
    private static final void run(NzaeAgg ae) {
        ae.runAggregation(new MyHandler());
    }
}
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java
     \ --template compile TestJavaMax.java --version 3
```

Registration

Register the example as a UDA, using the **state** parameter:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "maxae(int)" \
     --return "int4" --state "(int4)" --class AeUda --language java \
     --template uda --version 3 --define "java_class=TestJavaMax"
```

Running

To run, first create a dummy table and then run the aggregate:

```
CREATE TABLE grp_test (grp int4, val int4);
CREATE TABLE
INSERT INTO grp_test VALUES (1, 1);
INSERT 0 1
INSERT INTO grp_test VALUES (1, 2);
INSERT 0 1
INSERT INTO grp_test VALUES (1, 3);
INSERT 0 1
INSERT INTO grp_test VALUES (2, 4);
INSERT 0 1
SELECT maxae(val) FROM grp_test;
 MAXAE
-------
     4
(1 row)
```

```
SELECT grp, maxae(val) FROM grp_test GROUP BY grp;
 GRP | MAXAE
-----+-------
   1 |     3
   2 |     4
(2 rows)

SELECT grp, maxae(val) over (partition BY grp) FROM grp_test;
 GRP | MAXAE
-----+
   1 |-------3
   1 |       3
   1 |       3
   2 |       4
(4 rows)
```

Fortran Language Aggregates

This example uses the following file name:

```
penultimate.f
```

Code

```
      program penultimateUda
      call nzaeRun()
      stop
      end


      subroutine order(valueOne, valueTwo, valueThree)
      if (valueTwo > valueOne) then
            tempValue = valueOne
            valueOne  = valueTwo
            valueTwo  = tempValue
      endif
      if (valueThree > valueTwo) then
            tempValue  = valueTwo
            valueTwo   = valueThree
            valueThree = tempValue
      endif
      if (valueTwo > valueOne) then
            tempValue = valueOne
            valueOne  = valueTwo
            valueTwo  = tempValue
      endif
      if (valueThree > valueTwo) then
            tempValue  = valueTwo
            valueTwo   = valueThree
            valueThree = tempValue
      endif
      return
      end


      subroutine nzaeHandleRequest(handle)
      integer stateBoolean, value1, value2, value3, value4, tempValue
      stateBoolean      = 0
      value1            = 0
      value2            = 0
      value3            = 0
      value4            = 0
      tempValue         = 0

c     GET THE NEXT AGGREGATION TYPE.
```

```
10    call nzaeGetNextAggregation(handle)


      HANDLE initializeState().
      call nzaeIsAggStateInitializeState(handle, stateBoolean)
      if (stateBoolean .eq. 1) then
            call     nzaeSetAggregateInt32(handle,     0,     -
            2147483647) call nzaeSetAggregateInt32(handle, 1,
            -2147483647) call nzaeSaveAggregateResult(handle)
            goto 10
      endif

      HANDLE accumulate().
      call nzaeIsAggStateAccumulate(handle, stateBoolean)
      if (stateBoolean .eq. 1) then
            call  nzaeGetStateInt32(handle,  0,  value1,  isNull)
            call  nzaeGetStateInt32(handle,  1,  value2,  isNull)
            call nzaeGetInputInt32(handle, 0, value3, isNull)
            if (isNull .eq. 0) then
                  call order(value1, value2, value3)
            endif

            call nzaeSetAggregateInt32(handle, 0, value1)
            call nzaeSetAggregateInt32(handle, 1, value2)
            call nzaeSaveAggregateResult(handle)
            goto 10
      endif

      HANDLE merge().
      The two calls to order() below ensure that value1 and value2
          are the largest two values.
      call nzaeIsAggStateMerge(handle, stateBoolean)
      if (stateBoolean .eq. 1) then

            call nzaeGetStateInt32(handle, 0, value1, isNull)
            call nzaeGetStateInt32(handle, 1, value2, isNull)

            call nzaeGetInputStateInt32(handle, 0, value3, isNull)
            value4 = 22
            call nzaeGetInputStateInt32(handle, 1, value4, isNull)
            if (isNull .eq. 0) then
                  call order(value1, value2, value3)
                  call order(value2, value3, value4)
            endif

            call nzaeSetAggregateInt32(handle, 0, value1)
            call nzaeSetAggregateInt32(handle, 1, value2)
            call nzaeSaveAggregateResult(handle)
            goto 10
      endif

      HANDLE finalResult().
      call nzaeIsAggStateFinalResult(handle, stateBoolean)
      if (stateBoolean .eq. 1) then
            call    nzaeGetStateInt32(handle,    0,    value1,
            isNull)    call    nzaeGetStateInt32(handle,    1,
            value2, isNull) if (value1 < value2) then
                  call nzaeSetAggregateInt32(handle, 0, value1)
            else
                  call nzaeSetAggregateInt32(handle, 0, value2)
            endif
            call nzaeSaveAggregateResult(handle)
            goto 10
      endif

      HANDLE DONE.
      call nzaeIsAggDone(handle, stateBoolean)
```

```
      if (stateBoolean .eq. 1) then
            return
      endif

      HANDLE ERRORS.
      call nzaeIsAggError(handle, stateBoolean)
      if (stateBoolean .eq. 1) then
            call nzaeUserError(handle,
     +                       "Aggregate error state encountered.")
            return
      endif

      HANDLE INVALID STATE.
      call nzaeUserError(handle, "Invalid aggregate state.")
      return
      end
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language fortran --version 3
     \ --template compile penultimate.f
```

Registration

Register the example as a UDA, using a **state** parameter:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language fortran --version 3
     \ --template uda --exe penultimate --sig "penultimate(int4)" \
  --state "(int4, int4)" --return int4
```

Running

To run, first create a dummy table and then run the aggregate:

```
CREATE TABLE grp_test(value int4);
CREATE TABLE
INSERT INTO grp_test VALUES (1);
INSERT 0 1
INSERT INTO grp_test VALUES (2);
INSERT 0 1
INSERT INTO grp_test VALUES (3);
INSERT 0 1
INSERT INTO grp_test VALUES (4);
INSERT 0 1
INSERT INTO grp_test VALUES (5);
INSERT 0 1
SELECT penultimate(value) FROM grp_test;
 PENULTIMATE
-------------
           4
(1 row)
```

Python Language Aggregates

This example uses the following file name:

```
aggregate.py
```

Code

The code in this example is slightly longer than the C language aggregate example, since the code handles all possible data types.

```
import nzae

class AggregateAe(nzae.Ae):
       The _runUda function is the exact function implemented for the AE class.
       Reproducing this function is not needed in user code to run an aggregate
       as it is inherited. The user can override this function to have
       finer control when running the aggregate AE.
    def _runUda(self):

           LOOP, AGGREGATING.

        while True:

               FETCH THE NEXT AGGREGATION. aggregationType

            = self._getNextAggregation()

               INITIALIZE (AS APPROPRIATE).
            if aggregationType == self.AGGREGATION_TYPE__INITIALIZE:
                self._initializeState()
                self._saveAggregateResult()

               ACCUMULATE (AS APPROPRIATE).
            elif aggregationType == self.AGGREGATION_TYPE__ACCUMULATE:
                self._accumulate(self.getState(),  self.getInputRow())
                self._saveAggregateResult()

               MERGE (AS APPROPRIATE).
            elif  aggregationType  ==  self.AGGREGATION_TYPE__MERGE:
                self._merge(self.getState(),
                self.getInputState()) self._saveAggregateResult()

               GET THE FINAL RESULT (AS APPROPRIATE).
            elif aggregationType == self.AGGREGATION_TYPE__FINAL_RESULT:
                result = self._finalResult(self.getState())
                self._setAggregateResult(result, True)
                self._saveAggregateResult()

               END AS APPROPRIATE.
            elif aggregationType == self.AGGREGATION_TYPE__END:
                return

               ERROR AS APPROPRIATE.
            elif aggregationType == self.AGGREGATION_TYPE__ERROR:
                raise Exception("Error calling nzaeAggNext(). Cause unknown.")

            else:
                raise Exception("Received unknown aggregation type.")

      The functions below must be overridden by the user
      while writing an aggregate AE.
    def _initializeState(self):
        self.setState(0, -2147483647)

    def _accumulate(self, instate, row):
        if isinstance(instate, (list, tuple)):
            state = instate[0]
        else:
            state = instate

        if isinstance(row, (list, tuple)):
            for i in row:
                if state is None:
                    state = i
```

```
                    else:
                        if state < i:
                            state =  i

            if state is None:
                state = -2147483647

            self.setState(0, state)

        def _merge(self, instate, inputValues):
            if isinstance(instate, (list, tuple)):
                state = instate[0]
            else:
                state = instate

            if isinstance(inputValues, (list, tuple)):
                for i in inputValues:
                    if state is None:
                        state = i
                    else:
                        if state < i:
                            state = i
            if state is None:
                state = -2147483647

            self.setState(0, state)

        def _finalResult(self, state):
            if isinstance(state, (list, tuple)):
                return state[0]
            else:
                return -2147483647

    AggregateAe.run()
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language python64
     \ --template deploy ./aggregate.py --version 3
```

Registration

Register the example as a UDA, using the **state** parameter:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3
     \ --template uda --exe aggregate.py --sig "maxae(int)" --return int4 \
     --state "(int4)"
```

Running

To run, first create a dummy table and then run the aggregate. Running the aggregate outputs the following results:

```
CREATE TABLE grp_test (grp int4, val int4);
CREATE TABLE
INSERT INTO grp_test VALUES (1, 1);
INSERT 0 1
INSERT INTO grp_test VALUES (1, 2);
INSERT 0 1
INSERT INTO grp_test VALUES (1, 3);
INSERT 0 1
INSERT INTO grp_test VALUES (2, 4);
```

```
INSERT 0 1

SELECT maxae(val) FROM grp_test;
MAXAE
-------
4
(1 row)

SELECT grp, maxae(val) FROM grp_test GROUP BY grp;
GRP | MAXAE
-----+-------
1 | 3
2 | 4
(2 rows)

SELECT grp, maxae(val) over (partition BY grp) FROM
grp_test; GRP | MAXAE
-----+-------
1 | 3
1 | 3
1 | 3
2 | 4
(4 rows)
```

Perl Language Aggregates

This example uses the following file name:

```
Maxae.pm
```

Code

The code in this example is slightly longer than the C language aggregate example, since the code handles all possible data types.

```
package Maxae;

use nzae::Ae;
use strict;
use autodie;

our @ISA = qw(nzae::Ae);

my $ae = Maxae->new();
$ae->run();

   The runUda function here  is the exact function implemented for the Ae class
   Reproducing this function is not needed in user code to run an aggregate
   as it is inherited. The user can however override this function to have
   finer control on the running of the aggregate Ae

sub runUda
{
    my $self = shift;

    while(1)
    {
        my $aggregationType = $self->_getNextAggregation();

        if ( $aggregationType ==  $self->getAggregateTypeInitialize() )
        {
            $self->_initializeState();
            $self->_saveAggregateResult();
        }
```

```
        elsif ( $aggregationType ==  $self->getAggregateTypeAccumulate() )
        {
            $self->_accumulate($self->getState(), $self-
            >getInputRow()); $self->_saveAggregateResult();
        }
        elsif ( $aggregationType ==  $self->getAggregateTypeMerge() )
        {
            $self->_merge($self->getState(), $self->getInputState());
            $self->_saveAggregateResult();
        }
        elsif ( $aggregationType ==  $self->getAggregateTypeFinalResult() )
        {
            my $result = $self->_finalResult($self->getState());
            $self->_setAggregateResult($result, 1); $self-
            >_saveAggregateResult();
        }
        elsif ( $aggregationType ==  $self->getAggregateTypeEnd() )
        {
            return;
        }
        elsif ( $aggregationType ==  $self->getAggregateTypeError() )
        {
            croak(nzae::Exceptions::AeInternalError->new("Error
calling nzaeAggNext(). Cause unknown."));
        }
       else
        {
             croak(nzae::Exceptions::AeInternalError->new("Received
unknown aggregation type."));
        }

    }

}

#the functions below need to be overridden by the
user #while writing an aggregate Ae
sub _initializeState
{
    my $self = shift;
    $self->_setState(0, -2147483647);
}

sub _accumulate
{
    my $self = shift;
    my @instate = shift;
    my @row = shift;
    my $state = pop(@instate);

    if (scalar(@row) > 0)
    {
                if ( defined $row[0] )
                {
                     unless (defined $state)
                     {
                                 if (defined $row[0])
                                 {
                                     $state = $row[0];
                                 }
                     }
                     elsif ( $state < $row[0] )
                     {
                                 if (defined $row[0])
                                 {
                                     $state = $row[0];
                                 }
                     }
```

```
                }
    }

    unless (defined $state)
    {
                $state = -2147483647;
    }

    $self->_setState(0, $state);
}

sub _merge
{
    my $self = shift;
    my @instate = shift;
    my @inputValues = shift;
    my $state = pop(@instate);

    for ( my $i = 0 ; $i < scalar(@inputValues); $i++ )
    {
        unless (defined $state)
        {
            $state = $inputValues[$i];
        }
        elsif ( $state < $inputValues[$i] )
        {
            $state = $inputValues[$i];
        }
    }

    unless (defined $state)
    {
        $state = -2147483647;
    }

    $self->_setState(0, $state);
}

sub _finalResult
{
    my $self = shift;
    my @state = shift;
    my $ret = defined $state[0]? $state[0] : -
    2147483647; return $ret;
}


1;
```

In the example, the runUda function is overridden. As described in the example's comments, this function is already inherited from the nzae::Ae class and does not need to be overridden by the user. However, the function *can* be overridden if finer control over the working of the AE is needed.

In summary, to write a UDA without customization, the steps are:

Create a Perl module file. In the example above this is MaxAe.pm.

Import and instantiate the nzae::Ae class in the file.

Import autodie to handle unhandled exceptions during the course of execution.

Override _initializeState function to initialize the state variable defined in the registration step.

Override the accumulate method to implement aggregation functionality for each dataslice.

Override the merge method to implement aggregation of state variables from each dataslice.

Override the finalResult method to return the merged state variable.

Execute the run() method of the nzae::Ae object.

Since this is a Perl module, the file should contain a “1” at the end of the file

MaxAe.pm. Once the code is complete, it must be deployed and registered.

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language perl --version 3
     \ --template deploy Maxae.pm;
```

Registration

Register the example as a UDA, using the **state** parameter:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3
     \ --template uda --exe Maxae.pm --sig "maxaepl(int)" --return int4
     \ --state "(int4)"
```

Running

To run, first create a dummy table and then run the aggregate. The expected output is the same as the Java Aggregate example:

```
CREATE TABLE grp_test (grp int4, val int4);
CREATE TABLE

INSERT INTO grp_test VALUES (1, 1);
INSERT 0 1

INSERT INTO grp_test VALUES (1, 2);
INSERT 0 1

INSERT INTO grp_test VALUES (1, 3);
INSERT 0 1

INSERT INTO grp_test VALUES (2, 4);
INSERT 0 1

SELECT maxaepl(val) FROM grp_test;
MAXAEPL
---------
        4
(1 row)

SELECT grp, maxaepl(val) FROM grp_test GROUP BY grp;
GRP | MAXAEPL
-----+---------
   2  |        4
   1  |        3
(2 rows)

SELECT grp, maxaepl(val) over (partition BY grp) FROM
grp_test; GRP | MAXAEPL
-----+
    1 |---------3
    1 |         3
    1 |         3
```

```
   2 |       4
(4 rows)
```

R Language Aggregates 1

This example creates a UDA that finds the longest string and the highest numeric value from a set of input values and that returns the sum of these values.

Concepts

The R Language Adapter defines the following functions that must be defined in the input source file.

nz.init

nz.accum

nz.merge

nz.final

These functions are called at run time in subsequent steps. To access the data, the functions should use the same API function as functions and table functions.

The C-Analytic-Executable API defines a special API for processing data in all states. In R, however, the processing has been simplified, and you can use the standard functions to access input data, output data, and state data. Two additional functions that can be called only in aggregation mode are the getState function and the getOutputColumn function.

The input, output, and state have the following meaning in the defined states:

Processing state	Input API	Output API
Initialize	None	State
Accumulate	Input	State
Merge	State	Merge state
Finalize	Merge state	Output

According to the table, the following conditions apply:

In the *initialize* state, there is no input, and the output API is used to set the initial state values

In the *accumulate* state, the input API accesses the input data from the input table, while the output API controls the state variables

In the *merge* state, for each invocation of this function, another set of state variables is accessible on the input coming from the nodes/dataslices performing data processing, while the output API controls the *merge state* variables

In the *finalize* state, the merge state variables are accessible through the input API, and the output API controls the UDA output, which is then reported as the result of the SQL query

Code

This example describes a UDA that finds the maximum value in the specified group.

Enter the following code in the */tmp/maxae.R* file:

```
nz.mode <- 'aggregate'
nz.init <- function(){
  setOutputNull(0)
}
nz.accum <- function(){
  input <- getInputColumn(0)
  if (is.null(input))
    return()
  state <- getOutputColumn(0)
  if (is.null(state) || input > state)
    setOutputInt32(0, input)
}
nz.merge <- function(){
  input <- getInputColumn(0)
  # if NULL is encountered, we can skip this value set
  # since it must to be coming from an empty data slice
  if (is.null(input))
    return()
  state <- getOutputColumn(0)
  if (is.null(state) || input > state)
    setOutputInt32(0, input)
}
nz.final <- function () {
  input <- as.integer(getInputColumn(0))
  if (is.null(input))
    setOutputNull(0)
  else
    setOutputInt32(0, input)
}
```

Note: For the compilation step, you must add another object , the nz.mode variable to which you assign the given value: 'aggregate'. If you do not add this variable, you get an error during the compilation that the nz.fun object is not present. Setting nz.mode tells the compilation script that the AE is an aggregate. The script then looks for the functions that are listed in Concepts.

Compilation

Compile the code as follows:

```
/nz/export/ae/utilities/bin/compile_ae --language r --template compile
     \ --version 3 --db dev --user nz /tmp/maxae.R
```

Registration

In this example, the **-- template** switch takes the new value uda. In addition, the **--state** switch appears to define the aggregate state variables.

```
/nz/export/ae/utilities/bin/register_ae --language r --template uda  \
     --version 3 --db dev --user nz --sig 'maxae(INT4)' --return 'INT4' \
     --state '(INT4)' --exe maxae.R
```

Running

NOTE: When you execute a UDA in the *merge* state, the set of input state variables include state variables from data slices with no data that is relevant to the aggregation query. As a result, there might be NULL input state values that must be handled in the *merge* function.

The following example includes special comments to emphasize this scenario. Before you run the aggregate, you must create a dummy table. After you run the aggregate, you get the following results:

```
CREATE TABLE grp_test (grp int4, val int4);
CREATE TABLE
INSERT INTO grp_test VALUES (1, 1);
INSERT 0 1
INSERT INTO grp_test VALUES (1, 2);
INSERT 0 1
INSERT INTO grp_test VALUES (1, 3);
INSERT 0 1
INSERT INTO grp_test VALUES (2, 4);
INSERT 0 1


SELECT maxae(val) FROM grp_test;
 MAXAE
-------
     4
(1 row)

SELECT grp, maxae(val) FROM grp_test GROUP BY grp;
 GRP | MAXAE
-----+-------
   2 |     4
   1 |     3
(2 rows)

SELECT grp, maxae(val) over (partition BY grp) FROM grp_test;
 GRP | MAXAE
-----+-------
    1 |     3
    1 |     3
    1 |     3
    2 |     4
(4 rows)
```

R Language Aggregates 2

This example finds the longest string and largest number in the given group and returns the sum of the numbers and the length of the string.

All examples are run on a standard R data set with the name **iris**. You can load this data set into the Netezza appliance by using the Netezza R Library client-side package function as.nz.data.frame(). For more details, see the *Netezza Package for R Developer's Guide*.

Code

Enter the following code in the */tmp/sampleuda.R* file:

```
nz.mode <- 'aggregate'
nz.init <- function () {
  setOutputString(0, '')
  setOutputDouble(1, 0)
}
nz.accum <- function () {
  inputstr <- getInputColumn(0)
  statestr <- getOutputColumn(0)
  if (nchar(statestr) < nchar(inputstr))
    setOutputString(0, inputstr)
  inputnum <- getInputColumn(1)
  statenum <- getOutputColumn(1)
  if (inputnum > statenum)
    setOutputDouble(1, inputnum)
}
nz.merge <- function () {
  inputstr <- getInputColumn(0)
  inputnum <- getInputColumn(1)

  # after running nz.accum there is no possibility to
  # get a NULL value in the input state - therefore,
  # if NULL is encountered, we can skip this value set
  # since it must to be coming from an empty data slice
  if (is.null(inputstr) || is.null(inputnum))
    return()

  statestr <- getOutputColumn(0)
  statenum <- getOutputColumn(1)

  if (nchar(statestr) < nchar(inputstr))
    setOutputString(0, inputstr)
  if (inputnum > statenum)
    setOutputDouble(1, inputnum)
}
nz.final <- function () {
  setOutputDouble(0, nchar(getInputColumn(0)) + getInputColumn(1))
}
```

Compilation

Compile and register the UDA as follows:

```
/nz/export/ae/utilities/bin/compile_ae --language r --template compile
     \ --version 3 --db dev --user nz /tmp/sampleuda.R
```

Registration

```
/nz/export/ae/utilities/bin/register_ae --language r -- template uda  \
     --version 3 --db dev  --user nz \
     --sig 'ruda(VARCHAR(255),DOUBLE)' \
     --return 'DOUBLE' -- state '(VARCHAR(255),DOUBLE)' --exe sampleuda.R
```

Running

The following example shows an invocation:

```
SELECT ruda('text',2);
 RUDA
------
    6
(1 row)
```

When the data is distributed among at least two data slices, the result is noteworthy:

```
SELECT DISTINCT(datasliceid) FROM nza..iris;
 DATASLICEID
-------------
           4
           3
           1
           2
(4 rows)
```

This result indicates that the **iris** table is stored on data slices 1-4. The UDA is then invoked as any other SQL aggregate:

```
SELECT ruda(CLASS,SEPALLENGTH) FROM nza..iris;
 RUDA
------
17.9
(1 row)
```

You can validate the result by using the following two queries.

```
SELECT MAX(sepallength) FROM nza..iris;
 MAX
-----
 7.9

SELECT MAX(LENGTH(class)) FROM nza..iris;
 MAX
-----
  10
```

The queries show the same result, a sum of 17.9.

CHAPTER 11
Advanced Developer Principles

Introducing AE Environment Variables

An AE is connected to the NPS by an SQL function registration. This registration may be a scalar, table, or aggregate function.

Note: More than one function may be associated with a given AE.

These function registrations include AE Environment Variables (AE-ENV). An AE-ENV is used for three general purposes:

- To specify options to the AE Runtime System that apply to all application languages
- To specify options that apply to a specific language
- To specify custom options for a specific application

A Netezza appliance installation consists of two machine types: a host and one or more S-Blades (or SPUs). For performance reasons, these two types of machines use slightly different versions of Linux with different compiler libraries. Therefore, when you write an AE, you must compile and link two versions of an executable: one for the host and one for the SPUs. The host and the SPUs share a common network file directory tree--the AE Export Directory Tree. (See Developer Principles for more information on the runtime system and the AE export directory tree.)

The concept of AE-ENVs is similar to that of operating system environment variables. These variables are key/value pairs that hold and transmit information. Some variable key names have a specific meaning to the AE runtime system. Variable names starting with "nz" are reserved by Netezza for this purpose. You can create user-defined AE-ENVs that the system passes on to the executable. The language-specific APIs all have a function to return the values of any AE-ENV.

The AE registration utility **register_ae** automatically sets a number of necessary environment variables. Some AE functionality is available only by directly setting AE-ENV using the **register_ae --environment** option. This option may also be used to override any variables set by **register_ae**. (See AE Environment Variables and register_ae for additional information.)

Defining duplicate environment variable key names at registration time produces an error. However, duplicate variable key names defined dynamically, using SQL or by using AE-ENV include files, can override prior definitions. See Order of Variable Parsing for more information.

Commonly Used AE Environment Variable

The following sections describe commonly used AE environment variables and ENV features.

Adding Application-Specific Environment Variables

Choose a key name that does not begin with NZ and assign it a value.

```
MYKEYNAME=MYVALUE
```

This AE-ENV variable (MYKEYNAME) is available on both the host and the SPU.

In the following example, which uses a location prefix, the variable is available only on the host, under the AE-ENV key name MYHOSTVARIABLE:

```
NZAE_HOST_ONLY_MYHOSTVARIABLE=MYHOSTVALUE
```

Command Line Arguments

This example adds three command line parameters to the AE.

```
NZAE_NUMBER_PARAMETERS=3
NZAE_PARAMETER1=first parameter
NZAE_PARAMETER2=second parameter
NZAE_PARAMETER3=third parameter
```

These parameters are available in main (argc=4) as argv[1], argv[2], arg[3]. There is no AE limit to the number of command line parameters. Per convention, the location prefixes can be used with these variables. The AE can receive a different number of command line parameters on the host and the SPUs.

General Common Variables

NZAE_DEBUG AE

Controls the log file output. This can also be set by the --level option of register_ae.

```
NZAE_DEBUG=<integer>
```

Higher values of NZAE_DEBUG produce greater log output. For example, the default value of NZAE_DEBUG=0 produces no output while NZAE_DEBUG=1 produces minimal log output.

NZAE_ LOG_IDENTIFIER

Assists in identifying log files.

```
NZAE_LOG_IDENTIFIER=<character string>
```

A log file name begins with the character string "nzae-<identifier>-". Setting the NZAE_LOG_IDENTIFIER value, for example: NZAE_LOG_IDENTIFIER=DAVID, creates log file names

that begin with nzae-DAVID-. This is a useful way to more easily identify log files.

NZAE_LOG_DIR

Identifies the shared export drive.

```
NZAE_LOG_DIR=<directory path on the shared export drive>
```

This variable can be used to place all log files in a common directory on the shared export drive.

NZAE_NODATA_NOLAUNCH

Determines whether an AE is launched on a dataslice containing no data. A setting of 1 means the AE is not launched if the dataslice contains no data. The default setting is 0 (the AE is launched).

```
NZAE_NODATA_NOLAUNCH={0 | 1}
```

Compressed Columns

NZAE_COMPRESSED_COLUMNS

Controls the compressed column functionality of the AE system. Compressed columns allow more than one virtual column argument to be contained within a physical column argument to a scalar function or table function[6] AE or UDX. Use this functionality to circumvent the Netezza appliance SQL function physical argument limitation of 64. This functionality is used in conjunction with the nzaemultiarg UDF, which is used to pack multiple SQL function column arguments into a single argument. Compressed columns are covered in greater detail in Using AEs to Exceed NPS SQL Function Argument Limits.

```
NZAE_COMPRESSED_COLUMNS={0 | 1}
```

When set to 1, the AE system looks for compressed columns and automatically unpacks a physical compressed physical column argument into multiple virtual column arguments. This transformation is transparent to an AE. The default setting is 0. When set to 0, the AE system ignores compressed columns and treats them as a single string column.

Row Buffering

NZAE_ROW_BUFFER

Controls the data transmission behavior of input and output rows between the NPS system and an AE. Multiple rows can be packed into a buffer, reducing the number of process context switches between the NPS system and the AE.

```
NZAE_ROW_BUFFER={NO | RESULT | FULL}
```

When set to NO, the default, no buffering takes place. With RESULT, only result rows are buffered. With FULL, input and output rows are buffered. Full buffering of input and output affects SQL behavior; result only buffering does not. Row buffering is covered in greater detail in Understanding Row Buffering.

Debugging

These debugging values are intended only for host processing. They can be used under the test

6 Does not apply to aggregate AEs.

harness or by using the **--noparallel** registration option. This topic is covered in greater detail in Debugging Analytic Executables.

NZAE_TERMINAL

Redirects standard input, output, and errors to a terminal. By default, this variable is not set. To use, set it to the name of a running terminal. Use the Linux **tty** command to get a terminal name, which is in the form /dev/pts/1. Before debugging, run **sleep 1000000** on the terminal to keep the terminal's running shell from interfering with standard IO.

```
NZAE_TERMINAL=<terminal name>
```

NZAE_GDB_PATH

Runs GDB on the AE process. Use this feature in conjunction with NZAE_TERMINAL. The variable should be a full path to GDB or a debugger with GDB command line semantics. By default, this is not set. This feature works well with C, C++, and Fortran.

```
NZAE_GDB_PATH=<full path to GDB>
```

NZAE_SPIN_FILE_NAME

Attaches a debugger to the AE process while the program is "spinning" and then removes or renames the spin file. By default, this feature is not set. If set to an existing file name path, the AE initialization continually loops and sleeps until the named file is deleted or renamed. If it does not exist, the AE continues normal execution. See Local AE and Spin Files for more information.

```
NZAE_SPIN_FILE_NAME=<spin file path>
```

Remote AE

The following section summarizes the remote AE environment variables. See Understanding Remote AEs for greater detail.

NZAE_REMOTE

Indicates whether the AE is a remote function. The value defaults to 0, not remote; a value of 1 sets a function to remote. This value can also be set by the **--remote** option of register_ae.

```
NZAE_REMOTE=[0 | 1]
```

NZAE_REMOTE_NAME

Sets the name of the connection point for a remote AE. You must set this value if NZAE_REMOTE=1. This value can also be set by the **--rname** option of register_ae.

```
NZAE_REMOTE_NAME=<character string>
```

NZAE_REMOTE_NAME_SESSION

Sets the session ID as part of the connection point name. The default value is 0, the ID is not part of the name. When set to 1, session ID becomes part of the connection point name. This value can also be set by the **--rsession** option of register_ae.

```
NZAE_REMOTE_NAME_SESSION=[0 | 1]
```

NZAE_REMOTE_NAME_TRANSACTION

Sets the transaction ID as part of the connection point name. The default value is 0, the ID is not part of the name. When set to 1, transaction ID becomes part of the connection point name. This value

can also be set by the **--rtrans** option of register_ae.

```
NZAE_REMOTE_NAME_TRANSACTION=[0 | 1]
```

NZAE_REMOTE_NAME_DATA_SLICE

Sets the dataslice ID as part of the connection point name. The default value is 0, the ID is not part of the name. When set to 1, dataslice ID becomes part of the connection point name. This value can also be set by the **--rdataslice** option of register_ae.

```
NZAE_REMOTE_NAME_DATA_SLICE=[0 | 1]
```

NZAE_REMOTE_TIMEOUT_SECONDS

Sets the number of seconds the AE runtime system waits for a response before returning a timeout error. The default value is 60 seconds.

```
NZAE_REMOTE_TIMEOUT_SECONDS=<seconds>
```

NZAE_REMOTE_LAUNCH

Sets whether the registration launches a new AE remote process. The default value is 0, a new process is not launched. When set to 1, the registration launches a new remote process.

```
NZAE_REMOTE_LAUNCH=[0 | 1]
```

NZAE_REMOTE_LAUNCH_VERBOSE

Sets whether the registration launches a new AE remote process and displays information describing the new process. The default value is 0, a new process is not launched. When set to 1, the registration launches a new remote process with details. This value can also be set by the **--launch** option of register_ae.

```
NZAE_REMOTE_LAUNCH_VERBOSE=[0 | 1]
```

AE Environment Variable Prefixes

AE environment prefixes are appended to the front of an environment key name, providing instructions to the AE runtime system. The prefixes are processed and removed, leaving only the actual key name. Multiple prefixes may be used, but must appear in a definite order determined by order of precedence.

Precedence Level 0

When present, these prefixes restrict the environment variable to either the host or the S-Blade (SPU).

```
NZAE_HOST_ONLY_
NZAE_SPU_ONLY_
```

Precedence Level 1

When present, this prefix instructs the AE runtime not to perform variable (%{ }) or shared library name substitution (%[ ]).

```
NZAE_NO_SUBSTITUTE_ :
```

Precedence Level 2

These prefixes indicate that the value is to be appended or prepended to an existing environment value. If the key is not defined, then it is created with this value. AE runtime first searches for an AE environment key and then a Linux OS environment key to find an existing value.

```
NZAE_APPEND_
NZAE_PREPEND_
```

The following prefixes are used to include files containing AE environment variables, as described in the section AE Environment Variable Include Files. NZAE_INCLUDE_BEFORE and NZAE_INCLUDE_AFTER_ are allowed in include files but do not affect processing order. The prefixes must appear in precedence order, from low to high. Only one entry from each precedence level can be used with a key. For more information on variable processing, see Order of Variable Parsing.

```
NZAE_INCLUDE_;
NZAE_INCLUDE_BEFORE_;
NZAE_INCLUDE_AFTER_;
```

The following example demonstrates an existing key used only on the S-Blade called MY_RANDOM_ASCII_CHARS. Variable or shared library substitution is not desirable, but could occur since "%[" and "%{" sequences are possible. If an environment value already exists for this key, the value should be appended after it:

```
NZAE_SPU_ONLY_NZAE_NO_SUBSTITUTE_NZAE_APPEND_MY_RANDOM_ASCII_CHARS="a789%[
[[%%% {}}}89 90"
```

This example demonstrates prefix usage in its most general form, although it is likely more complex than most applications require.

Setting Dynamic AE Environment Variables

AE environment variables can be set at query run time, including variables that affect the behavior of the AE runtime system. Application-specific variables can also be set. They are set using an extra string in the last argument, usually a constant, to table or scalar SQL Functions[7]. If the argument comes from a table column, the column must have the same value in every row or the result is undefined. By default, the target AE receives the settings as the AE environment variables and not as an argument.

Example

```
SELECT * FROM demo, TABLE WITH FINAL(aedemo(f1,f2,f3,f4,
'NZAE_EXECUTABLE_PATH=/nz/export/ae/languages/java/6.13.0/jdk1.6.0_13/bin/java,
NZAE_NUMBER_PARAMETERS=1,NZAE_PARAMETER1=DataMatrix,MYAPP1="hello",MYAPP2=="goo
d bye"'));
```

The environment variables are passed within the last string argument. The SQL function must be registered with an extra string argument, such as varchar, or with arguments set to keyword "varargs" (variable number of arguments). The SQL function must also be registered with the option **--dynamic 2**.

7 This feature is not supported for aggregates

More about Dynamic AE Environment Variables

The NZAE_DYNAMIC_ENVIRONMENT variable activates arguments based on dynamic AE environment variables. (It can also be set with the **--dynamic** option of register_ae.) If set to 0, the default, dynamic AE environment variables are not supported. If set to 2, they system uses the last argument of the AE (string) for specifying the values.

```
NZAE_DYNAMIC_ENVIRONMENT={0 | 2} (feature disabled, default)
NZAE_DYNAMIC_ENVIRONMENT=2 (environment variables from last argument)
```

When NZAE_DYNAMIC_ENVIRONMENT is set to 2, the function must be defined to accept a string type argument as an additional, final parameter that can be used to pass one or more additional environment variables. If an AE-ENV key name matches a variable already defined through registration, then the dynamic variable overrides the registered variable.

The environment variables are specified as name1=value1, name2=value2, and so on. Apply the following syntax for environment variables:

Use a comma as variable separator.

Use double quotes as the quoting character.

Values that include white space and/or commas must be enclosed with double quotes; otherwise, white space is ignored.

If a value is quoted, the double quotes are removed after parsing.

If the double quote character is required as part of a value, use the backslash (“\”) character to escape.

Example

```
NZAE_EXECUTBLE_PATH=/bin/ae, data1="1,2,3,4,5,6", data2=wx\"b\"yz,data3="I
am an \"IBMer\""
```

With NZAE_DYNAMIC_ENV enabled, this would be part of the SQL invocation as the last argument, that is:

```
Select func(arg1,'NZAE_EXECUTABLE=/bin/ae...');
```

After parsing, this becomes:

```
NZAE_EXECUTBLE_PATH=/bin/ae
data1=1,2,3,4,5,6
data2=wx"b"yz
data3=I am an "IBMer"
```

After parsing, the dynamic environment variables are treated exactly as registration environment variables. This includes the use of prefixes such a NZAE_APPEND_, NZAE_PREPEND_, NZAE_HOST_ONLY_, and NZAE_SPU_ONLY_ processing.

The following is an example of SQL using dynamic AE environment variables to call a Java AE with main class DataMatrix:

```
SELECT * FROM demo, TABLE WITH FINAL(aedemo_varchar_dyn(f1,f2,f3,f4,
'NZAE_EXECUTABLE_PATH=%
{AEDEV_JAVA_JVM},NZAE_NUMBER_PARAMETERS=1,NZAE_PARAMETER1=DataMatrix'));
```

The string %{AEDEV_JAVA_JVM} from the code snippet above is an AE environment variable substitution, which is covered in greater detail in the section AE Environment Variable Substitution.

Variable substitution can be controlled if the AE sees this last argument as a function single-string argument, as it always sees the result in the AE environment variables (in other words, if the AE sees this function as taking 4 or 5 arguments). The fifth argument, which is the string containing the ENV variables may or may not be passed, depending on the setting of NZAE_DYNAMIC_ENVIRONMENT_PASS_LAST_ARGUMENT.

By default, NZAE_DYNAMIC_ENVIRONMENT_PASS_LAST_ARGUMENT is set to 0, specifying that the last argument should not be passed to the AE, which is the recommended setting. In the example above, when set to 0 the AE sees 4 arguments, when set to 1 it sees 5 arguments.

```
NZAE_DYNAMIC_ENVIRONMENT_PASS_LAST_ARGUMENT={0 | 1}
```

When NZAE_DYNAMIC_ENVIRONMENT is not zero, the AE runtime system expects the last string argument to be specified, and returns an error if one is not found. This includes the case where a dataslice has no data. To suppress an error in the empty dataslice case, specify the following:

```
NZAE_DYNAMIC_ENVIRONMENT_INGORE_LAST_RUN_ERROR
(value 0 or 1, default is 1)
```

If a dataslice produces no data for a query, then it is not possible for the AE runtime to receive the dynamic AE environment variables. By default, AEs are not launched for empty dataslices. This is typically the correct behavior due to the absence of data; however there may be situations where it should be flagged as an error.

The following setting results in an error :

```
NZAE_DYNAMIC_ENVIRONMENT_INGORE_LAST_RUN_ERROR=0
```

Order of Variable Parsing

AE environment variables are processed in a defined order. This can result in variables overriding those of the same name that were processed earlier. The order of processing is:

Table 3: Order of variable parsing

Type	Description	Overrides...
1. Include before	Includes these files before the top-level registration environment variables. See “AE Environment Variable Include Files.”	
2. Registered	Prefaced with NZAE_REGISTER_, which is stripped from the name, See “AE Environment Variables and register_ae ,” the --environment description. These are a special case of top-level registration variables.	Include before
3. Normal	Specified as a variable in a SQL CREATE command (as done by register_ae) . These are also known as top-level registration variables.	Registered or include before

Type	Description	Overrides...
4. Include after	Includes these files after the top-level registration environment variables. See AE Environment Variable Include Files.	Normal, registered, or include before
5. Dynamic	Passed in as an argument to a function. See Setting Dynamic AE Environment Variables.	Include after, normal, registered, or include before

Further detail:

An environment variable can override any variable with the same name that was added in earlier processed section types (see the table above). Additionally, an environment variable in an include file can also override any variables with the same name that have come earlier in the include processing. In all other section types (Registered, Normal, Dynamic) having two variables with the same name in the same section type results in undefined behavior.

Multiple "Include Before" and "Include After" are added in alphabetical order; variables inside the include files are added in definition order.

Top-level environment variables (those specified in a DDL) are processed in alphabetical order. However, the top-level registration environment variable key names must not duplicate other top level variable key names.

Working with AE Shared Libraries

The AE shared library functionality allows shared libraries to be stored in the database so they are available at runtime to UDXs and AEs on the NPS. Originally specific to Linux Shared Libraries (.so files), this functionality has been extended so that any type of file can be stored in the database, including executables, Java class files, and text files. In practice, the shared library file is made available to an AE at runtime through an environment variable or the AE API that provides the location of the file on the file system. The file is read-only and should not be modified or deleted.

Although the AE shared library functionality allows file resources required by an AE to be packaged into the NPS system database, the alternative is to place these file resources in the AE export directory tree on a network shared drive. Selection is based on the characteristics of the application. For example, the AE shared library functionality may provide better organizational efficiency. For some applications, it may be used to avoid network contention on the AE export directory tree. However, the AE shared library functionality might become cumbersome if too many files are involved, making the AE export directory tree a better choice.

As an example, the helloae code shown in the Simple C Language AE section can be modified to use AE shared libraries instead of the AE export directory tree. In the example, the host executable is located at /nz/export/ae/applications/dev/david/host/helloae and the SPU executable is located at /nz/export/ae/applications/dev/david/spu/helloae. First, use an SQL command to store these as

AE shared libraries in the database:

```
CREATE OR REPLACE LIBRARY HELLOAE
MANUAL LOAD
EXTERNAL HOST OBJECT '/nz/export/ae/applications/dev/david/host/helloae'
EXTERNAL SPU OBJECT '/nz/export/ae/applications/dev/david/spu/helloae';
```

Some notes about this SQL:

MANUAL LOAD tells the Netezza system not to automatically load this AE shared library. Only true Linux .so shared library files should be automatically loaded.

Both the host and SPU copies of the helloae executable are stored in the database. The database does not maintain a pointer back to the file versions on disk.

HELLOAE is the external name of this AE shared library.

For advanced NPS system users, an AE shared library that is not a true .so file should not be a dependency for another library.

This AE is registered as follows:

```
--deps inza..LIBNZAEADAPTER,inza..LIBNZAECHILD,HELLOAE

NZAE_EXECUTABLE_PATH=%[HELLOAE]
```

Some notes about this registration:

The **--deps** option tells register_ae or nzudxcompile which shared libraries are required by this AE application. HELLOAE has been added to the default list.

%[HELLOAE] is an example of AE shared library path substitution. At runtime the AE runtime system converts this to a file path pointing to a copy of the shared library. Note the syntax is the percent sign (%) followed by square brackets ([ ]) that enclose the external AE shared library name.

The AE shared library takes care of placing the host and SPU executables on the appropriate machines. Only one NZAE_EXECUTABLE_PATH without a location prefix is required.

This AE application, as shown, does not access the AE export directory tree.

The AE shared library functionality can be used for any file type. AE shared library path substitution can be used in any environment value.

To view only the XML file associated with the helloae application:

```
/home/david/myxml.xml
```

The XML file is not required to be in the export tree since shared libraries do not point back to the source file once created.

Create an AE shared library:

```
CREATE OR REPLACE LIBRARY DAVIDXML
MANUAL LOAD
EXTERNAL HOST OBJECT '/home/david/myxml.xml'
EXTERNAL SPU OBJECT '/home/david/myxml.xml';
```

DAVIDXML is a snapshot of the XML file at the time the SQL command runs. In this case, the same copy is used for the host and SPU, but different files can be used. The XML file can be passed as a command line argument to the **helloae** AE application.

```
NZAE_EXECUTABLE_PATH=%[HELLOAE]
NZAE_NUMBER_PARAMETERS=1
```

```
NZAE_PARAMETER1=%[DAVIDXML]
```

An alternative registration is as follows:

```
NZAE_EXECUTABLE_PATH=%[HELLOAE]
MY_XML=%[DAVIDXML]
```

The following, alternative approach, uses an AE environment API call to retrieve the value of MY_XML, which is a full path to a temporary copy of the file (which is read only).

Understanding Row Buffering

AE row buffering can increase performance. Since AEs run in a separate OS process, a significant part of performance overhead may be the cost of marshaling row data between the NPS process and an AE process. This performance cost is affected by the raw amount of data transferred and by the number of data round trips between the NPS system and an AE. The performance cost occurs because a large number of OS process context switches can be expensive.

One way to lower this cost within an application is to put more information in each row, creating larger row sizes. This approach can be useful for AEs called from any type of SQL function. Keep in mind, however, that the Netezza system row size is limited to 64K bytes.

Another way to improve performance is to use AE row buffering. This functionality can only be used for AEs invoked through SQL table functions. AE row buffering is most useful when inputting or outputting a large number of small rows. When activated, the AE automatically places rows into a buffer and only transmits the buffer between the NPS system and AE when it becomes full or the end of data is reached. AE row buffering can be set for both input and output rows (full buffering) or for output rows only (result only B-buffering).

The size of the buffer is an internal implementation detail of the Netezza system. It is large enough to be useful but not so large that a single function overuses memory resources. The implication is that row buffering increases performance with smaller records (of roughly less than 1000 bytes) and decreases performance with larger records.

Note that when you use full buffering, the SQL output connection between the input and output rows is lost. For example, a table function implemented using an AE that returns exactly one output row per input row is called in this manner:

```
SELECT t.col1, f.result from mytable t, table WITH
final(mytablefunction(t.col1)) f;
```

When not using AE row buffering, the output f.result always corresponds to the input t.col1 that produced it. The use of AE row buffering means that this correspondence is lost in the output. There are many application scenarios where this correspondence is not required., however if it is required, a possible workaround is to have the table function return both the result and the input that produced it so that the values correspond.

AE row buffering is controlled by the AE environment variable NZAE_ROW_BUFFER. See Row Buffering for more information.

CHAPTER 12

Environment and Shared Libraries Examples

The concepts covered in this section include:

AE Environment Variables

Shared Libraries

C Language Example

This example uses the following file name:

```
env.c
```

Code

The environment and shared library classes provide information about the environment and libraries associated with an AE. This example takes one argument, an integer. If the integer equals 0, the AE outputs the environment it sees. If the integer equals 1, the AE looks the environment up as a library. If found, the path to the library is returned, otherwise NULL is returned:

```
#include <stdio.h>
#include <stdlib.h>

#include  "nzaeapis.h"

static int run(NZAE_HANDLE h);
int main(int argc, char * argv[])
{
    if (nzaeIsLocal())
    {
        NzaeInitialization arg;
        memset(&arg, 0, sizeof(arg));
                arg.ldkVersion = NZAE_LDK_VERSION;
        if (nzaeInitialize(&arg))
        {
            fprintf(stderr, "initialization failed\n");
            return -1;
        }
        run(arg.handle);
        nzaeClose(arg.handle);
```

```
    }
    else {
        NZAECONPT_HANDLE hConpt = nzaeconptCreate();
        if (!hConpt)
        {
            fprintf(stderr, "error creating connection point\n");
            fflush(stderr);
            return -1;
        }
        const char * conPtName = nzaeRemprotGetRemoteName();
        if (!conPtName)
        {
            fprintf(stderr, "error getting connection point name\n");
            fflush(stderr);
            exit(-1);
        }
        if (nzaeconptSetName(hConpt, conPtName))
        {
            fprintf(stderr, "error setting connection point name\n");
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NzaeremprotInitialization args;
        memset(&args, 0, sizeof(args));
        args.ldkVersion = NZAE_LDK_VERSION;
        args.hConpt = hConpt;
        if (nzaeRemprotCreateListener(&args))
        {
            fprintf(stderr, "unable to create listener - %s\n", \
                args.errorMessage);
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NZAEREMPROT_HANDLE hRemprot = args.handle;
        NzaeApi api;
        int i;
        for (i = 0; i < 5; i++)
        {
            if (nzaeRemprotAcceptApi(hRemprot, &api))
            {
                fprintf(stderr, "unable to accept API - %s\n", \
                     nzaeRemprotGetLastErrorText(hRemprot));
                fflush(stderr);
                nzaeconptClose(hConpt);
                nzaeRemprotClose(hRemprot);
                return -1;
            }
            if (api.apiType != NZAE_API_FUNCTION)
            {
                fprintf(stderr, "unexpected API returned\n");
                fflush(stderr);
                nzaeconptClose(hConpt);
                nzaeRemprotClose(hRemprot);
                return -1;
            }
            printf("testcapi: accepted a remote request\n");
            fflush(stdout);
            run(api.handle.function);
        }
        nzaeRemprotClose(hRemprot);
        nzaeconptClose(hConpt);
    }
    return 0;
}

static int run(NZAE_HANDLE h)
```

```
{
    NzaeMetadata metadata;
    if (nzaeGetMetadata(h, &metadata))
    {
        fprintf(stderr, "get metadata failed\n");
        return -1;
    }

    #define CHECK(value) \
    { \
        NzaeRcCode rc = value; \
        if (rc) \
        { \
            const char * format = "%s in %s at %d"; \
            fprintf(stderr, format,  \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            nzaeUserError(h, format, \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            exit(-1); \
        } \
    }

    for (;;)
    {
        int64_t ar[6];
        int i;
        double result = 0;
        NzaeRcCode rc = nzaeGetNext(h);
        if (rc == NZAE_RC_END)
        {
            break;
        }

        NzudsData * input = NULL;
        CHECK(nzaeGetInputColumn(h, 1, &input));
        int type = *input->data.pInt32;
        CHECK(nzaeGetInputColumn(h, 0, &input));
        if (type == 0) {
            const char *result;
            nzaeGetEnv(h, input->data.pVariableString, &result);
            if (!result) {
                CHECK(nzaeSetOutputNull(h, 0));
            }
            else
                CHECK(nzaeSetOutputString(h, 0, result));
        }
        else {
            const char *result = nzaeGetLibraryFullPath(h, \
                 input->data.pVariableString, false);
            if (!result) {
                CHECK(nzaeSetOutputNull(h, 0));
            }
            else
                CHECK(nzaeSetOutputString(h, 0, result));
        }
        CHECK(nzaeOutputResult(h));
    }
    nzaeDone(h);
    return 0;
}
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language system --version 3 \
     --template compile env.c --exe env
```

Registration

Register the example:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3 \
     --template udtf --exe env --sig "env_c(varchar(any),int4)" \
     --return "table(output varchar(1000))" --deps inza..LIBNZAEADAPTERS
```

This example assumes you have added the LIBNZAEADAPTERS shared library. For information on how to add shared libraries, see the section Working with AE Shared Libraries.

Running

Note that the output from this example is specific to the environment. Therefore, your actual output will resemble, but not match, the text below.

```
SELECT * FROM TABLE WITH FINAL(env_c('inza..libnzaeadapters', 1));
                OUTPUT
-------------------------------------------------------------------------
/nz/data.1.0/base/1/library/237951/host/libnzaeadapters.so (1
row)
SELECT * FROM TABLE WITH FINAL(env_c('NZAE_DYNAMIC_ENVIRONMENT', 0));
 OUTPUT
--------
 0
(1 row)
SELECT * FROM TABLE WITH FINAL(env_c('NZAE_DYNAMIC_ENVIRONMENT2', 0));
 OUTPUT
--------

(1 row)
```

C++ Language Example

This example uses the following file name:

```
env.cpp
```

Code

The environment and shared library classes provide information about the environment and libraries associated with an AE. This example takes one argument, an integer. If the integer equals 0, the AE outputs the environment it sees. If the integer equals 1, the AE looks the environment up as a library. If found, the path to the library is returned, otherwise NULL is returned:

```
#include <nzaefactory.hpp>

using namespace nz::ae;

static int run(nz::ae::NzaeFunction *aeFunc);


int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
       The following line is only needed if a launcher is not
    used helper.setName("testcapi");
    for (int i=0; i < 2; i++) {
        nz::ae::NzaeApi &api = helper.getApi(nz::ae::NzaeApi::FUNCTION);
```

```
        run(api.aeFunction);
        if (!helper.isRemote())
            break;
    }

    return 0;
}

class MyHandler : public NzaeFunctionMessageHandler
{
public:
    MyHandler() {
        m_Once = false;
    }

    void doOnce(NzaeFunction& api) {
        const NzaeMetadata& meta = api.getMetadata();
        if (meta.getOutputColumnCount() != 1 ||
            meta.getOutputType(0) != NzaeDataTypes::NZUDSUDX_VARIABLE) {
            throw NzaeException("expecting one output column of type string");
        }
        if (meta.getInputColumnCount() != 2) {
            throw NzaeException("expecting at least two input columns");
        }
        if (meta.getInputType(0) != NzaeDataTypes::NZUDSUDX_FIXED &&
            meta.getInputType(0) != NzaeDataTypes::NZUDSUDX_VARIABLE) {
            throw NzaeException("first input column expected to be a string
type");
        }
        if (meta.getInputType(1) != NzaeDataTypes::NZUDSUDX_INT32) {
            throw NzaeException("second input column expected to be int32 type");
        }
        m_Once = true;

    }
    void evaluate(NzaeFunction& api, NzaeRecord &input, NzaeRecord &result) {

        if (!m_Once)
            doOnce(api);


        NzaeField &field = input.get(0);
        if (field.isNull())
            throw NzaeException("first input column may not be null");
        NzaeStringField &sf = (NzaeStringField&) input.get(0);
        std::string strop = (std::string)sf;
        NzaeField &field2 = input.get(1);
        if (field2.isNull())
            throw NzaeException("second input column may not be
        null"); NzaeInt32Field &inf = (NzaeInt32Field&) input.get(1);
        int type = (int32_t)inf;
        NzaeStringField &of= (NzaeStringField&)result.get(0);

        if (type == 0) {
            // env
            const NzaeEnvironment& env = api.getEnvironment();
            const char* val = env.getValue(strop.c_str());
            if (!val)
                of.setNull(true);
            else
                of = std::string(val);

        }
        else {
            const NzaeLibrary& libs = api.getLibrary();
            const NzaeLibrary::NzaeLibraryInfo *info =
        libs.getLibraryInfo(strop.c_str(), false ,
        NzaeLibrary::NzaeLibrarySearchBoth);
```

```
            if (!info)
                of.setNull(true);
            else
                of = info->libraryFullPath;


        }


    }

    bool m_Once;

};
static int run(NzaeFunction *aeFunc)
{

    aeFunc->run(new MyHandler());
    return 0;
}
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp --template compile
     \ --exe envae --compargs "-g -Wall" --linkargs "-g" env.cpp --version 3
```

Registration

Register the example:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "envae(VARCHAR(ANY),int4)" \
     --return "table(val varchar(1000))" --language cpp --template udtf
     \ --exe envae --version 3 --deps inza..LIBNZAEADAPTERS
```

This example assumes you have added the LIBNZAEADAPTERS shared library. For information on how to add shared libraries, see the section Working with AE Shared Libraries.

Running

Note that the output from this example is specific to the environment. Therefore, your actual output will resemble, but not match, the text below.

```
SELECT * FROM TABLE WITH FINAL(envae('inza..libnzaeadapters', 1));
                      VAL
------------------------------------------------------------------------
/nz/data.1.0/base/1/library/237951/host/libnzaeadapters.so (1
row)
SELECT * FROM TABLE WITH FINAL(envae('libnzaeadapters2', 1));
 VAL
-----

(1 row)

SELECT * FROM TABLE WITH FINAL(envae('NZAE_DYNAMIC_ENVIRONMENT', 0));
 VAL
-----
 0
(1 row)

SELECT * FROM TABLE WITH FINAL(envae('NZAE_DYNAMIC_ENVIRONMENT2', 0));
```

```
 VAL
-----

(1 row)
```

Java Language Example

This example uses the following file name:

```
TestJavaEnvironment.java
```

Code

The environment and shared library classes provide information about the environment and libraries associated with an AE. This example takes one argument, an integer. If the integer equals 0, the AE outputs the environment it sees. If the integer equals 1, the AE looks the environment up as a library. If found, the path to the library is returned, otherwise NULL is returned:

```
import org.netezza.ae.*;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class TestJavaEnvironment {

    private static final Executor exec =
        Executors.newCachedThreadPool();


    public static final void main(String [] args) {
        try {
            mainImpl(args);
        } catch (Throwable t) {
            System.err.println(t.toString());
            NzaeUtil.logException(t, "main");
        }
    }

    public static final void mainImpl(String [] args) {
        NzaeApiGenerator helper = new NzaeApiGenerator();
        while (true) {
            final NzaeApi api = helper.getApi(NzaeApi.FUNCTION);
            if (api.apiType == NzaeApi.FUNCTION) {
                if (!helper.isRemote()) {
                    run(api.aeFunction);
                    break;
                } else {
                    Runnable task = new Runnable() {
                            public void run() {
                                try {
                                    TestJavaEnvironment.run(api.aeFunction);
                                } finally {
                                    api.aeFunction.close();
                                }
                            }
                        };
                    exec.execute(task);


                }
            }
        }
        helper.close();
```

```
    }

    public static class MyHandler implements NzaeMessageHandler
    {
        public void evaluate(Nzae ae, NzaeRecord input, NzaeRecord output) {

            final NzaeMetadata meta = ae.getMetadata();

            int op = 0;
            double result = 0;

            if (meta.getOutputColumnCount() != 1 ||
                meta.getOutputNzType(0) != NzaeDataTypes.NZUDSUDX_VARIABLE) {
                throw new NzaeException("expecting one output column of type
string");
            }
            if (meta.getInputColumnCount() != 2) {
                throw new NzaeException("expecting at least two input columns");
            }
            if (meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_FIXED &&
                meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_VARIABLE) {
                throw new NzaeException("first input column expected to be a
string type");
            }
            if (meta.getInputNzType(1) != NzaeDataTypes.NZUDSUDX_INT32) {
                throw new NzaeException("second input column expected to be int32
type");
            }

            String field = (String) input.getField(0);
            if (field == null)
                throw new NzaeException("first input column may not be null");
            Object o = input.getField(1);
            if (o == null)
                throw new NzaeException("second input column may not be null");

            int type = (Integer)o;

            if (type == 0) {
                // env
                NzaeEnvironment env = ae.getEnvironment();
                String val = env.getValue(field);
                if (val == null)
                    output.setField(0,null);
                else
                    output.setField(0,val);
            }
            else {
                NzaeLibrary libs = ae.getLibrary();
                NzaeLibrary.NzaeLibraryInfo info = libs.getLibraryInfo(field,
                     false , NzaeLibrary.SEARCH_BOTH);
                if (info == null)
                    output.setField(0,null);
                else
                    output.setField(0,info.libraryFullPath);
            }
        }
    }

    public static int run(Nzae ae)
    {
        ae.run(new MyHandler());
        return 0;
    }
}
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java
 \ --template compile TestJavaEnvironment.java --version 3
```

Registration

Register the example:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "envae(varchar(any),int)"
     \ --return "table(val varchar(1000))" --class AeUdtf --language java \
     --template udtf --version 3 --define "java_class=TestJavaEnvironment" \
--deps inza..LIBNZAEADAPTERS
```

This example assumes you have added the LIBNZAEADAPTERS shared library. For information on how to add shared libraries, see the section Working with AE Shared Libraries.

Running

Note that the output from this example is specific to the environment. Therefore, your actual output will resemble, but not match, the text below.

```
SELECT * FROM TABLE WITH FINAL(envae('inza..libnzaeadapters', 1));
                            VAL
------------------------------------------------------------------------
/nz/data.1.0/base/1/library/237951/host/libnzaeadapters.so (1
row)
SELECT * FROM TABLE WITH FINAL(envae('libnzaeadapters2', 1));
 VAL
-----

(1 row)

SELECT * FROM TABLE WITH FINAL(envae('NZAE_DYNAMIC_ENVIRONMENT', 0));
 VAL
-----
 0
(1 row)

SELECT * FROM TABLE WITH FINAL(envae('NZAE_DYNAMIC_ENVIRONMENT2', 0));
 VAL
-----

(1 row)
```

Fortran Language Example

This example uses the following file name:

```
libenv.f
```

Code

The environment and shared library classes provide information about the environment and libraries associated with an AE. This example takes one argument, an integer. If the integer equals 0, the AE

outputs the environment it sees. If the integer equals 1, the AE looks the environment up as a library.

```
program logRunUdtf
call nzaeRun()
stop
end

subroutine nzaeHandleRequest(handle)
integer     outputType, hasNext, isNull
hasNext      = -1
outputType  =  -1
isNull      =  -1
GET INPUT.
call nzaeGetNext(handle, hasNext)
if (hasNext .eq. 0) then
      return
endif

OUTPUT EITHER THE ENVIRONMENT OR THE LIBRARIES.
call nzaeGetInputInt32(handle, 0, outputType, isNull)
if (outputType .eq. 0) then
      call outputEnvironment(handle)
else
      call outputLibraries(handle)
endif
return
end


subroutine outputEnvironment(handle)
character(1000)   name, value
integer           hasNext
name              = "init"
value             = "init"
hasNext           = -1

OUTPUT THE FIRST VALUE.
call nzaeGetFirstEnvironmentVariable(handle, name, value)
call nzaeSetOutputString(handle, 0, name)
call nzaeSetOutputString(handle, 1, value)
call nzaeOutputResult(handle)
call nzaeGetNextEnvironmentVariable(handle, name, value,
hasNext) if (hasNext .eq. 0) then
      return
endif
call nzaeSetOutputString(handle, 0, name)
call nzaeSetOutputString(handle, 1, value)
call nzaeOutputResult(handle)
goto 100
end


subroutine outputLibraries(handle)
character(1000)   name, path
integer           number, autoload, onLibrary
name              = "init"
path              = "init"
number            = -1
autoload          = -1
onLibrary         = 0

GET THE NUMBER OF SHARED LIBRARIES.
call nzaeGetNumberOfSharedLibraries(handle, number)

GET SHARED LIBRARY INFO.
200   if (onLibrary .eq. number) then
```

```
      return
endif
call nzaeGetSharedLibraryInfo
           (handle, onLibrary, name, path, autoload)
onLibrary = onLibrary + 1
OUTPUT THE NAME AND PATH.
call nzaeSetOutputString(handle, 0, name)
call nzaeSetOutputString(handle, 1, path)
call nzaeOutputResult(handle)
goto 200
end
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language fortran --version 3
     \ --template compile libenv.f
```

Registration

Register the example:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language fortran --version 3
     \ --template udtf --exe libenv --sig "lib_env(int4)" \
     --return "table(name varchar(1000), value varchar(1000))"
```

Running

Note that the output from this example is specific to the environment. Therefore, your actual output will resemble, but not match, the text below.

```
SELECT * FROM TABLE WITH FINAL(lib_env(1));
      NAME       |                              VALUE
-----------------+
 LIBNZAEADAPTERS |-----------------------------------------------------/nz/DATA/base/1/library/200245/host/libnzaeadapters.so
 LIBNZAECHILD    | /nz/DATA/base/1/library/200247/host/libnzaechild.so
 LIBNZAEPARENT   | /nz/DATA/base/1/library/200244/host/libnzaeparent.so
(3 rows)
SELECT * FROM TABLE WITH FINAL (lib_env(0)) LIMIT 5;
           NAME            |                     VALUE
                          +-------------------------------------------------
----------------------NZAE_NZREP_VERSION    | 6
 NZREP_TEMP_FILE            | /nz/tmp/nzrep_DYxBgF
 NZREP_PARAMETER1           | 12
 NZREP_PARAMETER0           | 11
 NZAE_EXECUTABLE_PATH_RUN |
/nz/export/ae/applications/system/admin/host/libenv (5 rows)
```

Note: The above output will vary somewhat as the name may be preceded by a database name with periods separating it from the library name (e.g., **xxx..LIBNZAECHILD**). Also, the numbers displayed in the value path name will be different and of the form:

```
/nz/DATA/base/<dbOID>/library/<objectOID>/host/<libraryname>.so
```

Python Language Example

This example uses the following file name:

```
libenv.py
```

Code

The environment and shared library classes provide information about the environment and libraries associated with an AE. This example takes one argument, an integer. If the integer equals 0, the AE outputs the environment it sees. If the integer equals 1, the AE looks the environment up as a library.

```
import nzae

class LibEnvAe(nzae.Ae):

    def _run(self):
        for row in self:
            if row[0] == 0:
                for name, value in self.getEnvironment().iteritems():
                    self.output([name, value])
            else:
                for name, value in self.yieldSharedLibraries().iteritems():
                    self.output([name, value])

LibEnvAe.run()
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language python64
     \ --template deploy ./libenv.py --version 3
```

Registration

Register the example:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3
     \ --template udtf --exe libenv.py --sig "lib_env(int4)" \
     --return "table(name varchar(1000), value varchar(1000))"
```

Running

Note that the output from this example is specific to the environment. Therefore, your actual output will resemble, but not match, the text below. The name may be preceded by a database name with periods separating it from the library name (e.g., **xxx..LIBNZAECHILD**). Also, the numbers displayed in the value path name will be different and of the form:

```
/nz/DATA/base/<dbOID>/library/<objectOID>/host/<libraryname>.so
```

```
 SELECT * FROM TABLE WITH FINAL(lib_env(1));
      NAME        |                          VALUE
------------------------------------------------------------------------------
 LIBNZAEADAPTERS | /nz/DATA/base/1/library/200245/host/libnzaeadapters.so
 LIBNZAEPARENT   | /nz/DATA/base/1/library/200244/host/libnzaeparent.so
 LIBNZAECHILD    | /nz/DATA/base/1/library/200247/host/libnzaechild.so
(3 rows)
```

```
SELECT * FROM TABLE WITH FINAL (lib_env(0)) LIMIT 5;
              NAME               |                  VALUE
---------------------------------------------------------------------------
 AE_PYTHON_ROOT                  | /nz/export/ae/languages/python/2.7/host64
 NPS MODEL                       | TT4
 NZAE_SHLIB_AUTO_LIBNZAEADAPTERS | 1
 NZAE_RUNTIME_TRANSACTION_ID     | 8558
 NZAE_OPERATION_TIMEOUT          | 1800
(5 rows)
```

Perl Language Example

This example uses the following file name:

```
Libenv.pm
```

Code

The environment and shared library classes provide information about the environment and libraries associated with an AE. This example takes one argument, an integer. If the integer equals 0, the AE outputs the environment it sees. If the integer equals 1, the AE looks the environment up as a library.

```
package Libenv;

use nzae::Ae;
use strict;
use autodie;

our @ISA = qw(nzae::Ae);

my $ae = Libenv->new();
$ae->run();

sub _run()
{
    my $self = shift;
    $self->iter();
}

sub iter()
{
#use while loop over library function getNext()
    my $self = shift;
    while ($self->getNext())
    {
        my @row = $self->getInputRow();
        my $size = scalar(@row);
        my $env;
        if ($size > 0)
        {
           if ( $row[0] == 0 )
           {
               $env = $self->getEnvironment();
           }
           else
           {
               $env = $self->yieldSharedLibraries();
           }

           unless(defined $env)
           {
                     $self->userError("Error fetching information");
           }
```

```
            my $limit = defined  $row[1]? $row[1] : undef;

    try
    {
      for my $key ( keys %$env )
      {
        $self->output($key, $env->{$key});
        if (   defined $limit && $limit)
        {
          $limit--;
          if ($limit == 0)
          {
                last;
          }
        }
      }
    }
    catch
    {
     croak(nzae::Exceptions::AeInternalError->new(longmess("Error
writing output")));
     };
        }
        else
        {
            $self->userError("Error fetching row");
        }
    }
}

1;
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language perl --version 3
     \ --template deploy Libenv.pm
```

Registration

Register the example:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3
     \ --template udtf --exe Libenv.pm --sig "lib_env(int4, int4)" \ --
     return "table(name varchar(1000), value varchar(1000))"
```

Given a positive integer value, the second argument acts as a limit to the number of results returned. This is used in place of keyword LIMIT to exit the AE cleanly when the number of results returned must be limited.

Running

Note that the output from this example is specific to the environment. Therefore, your actual output will resemble, but not match, the text below. For example, the name may be preceded by a database name with periods separating it from the library name (e.g., **xxx..LIBNZAECHILD**). Also, the numbers displayed in the value path name will be different and of the form:

```
/nz/DATA/base/<dbOID>/library/<objectOID>/host/<libraryname>.so
```

```
SELECT * FROM TABLE WITH FINAL(lib_env(1, 3));
```

```
      NAME        |                                  VALUE
                  +-------------------------------------------------
---------------LIBNZAEPARENT | /nz/data.1.0/base/317943/library/319587/host/libnzaeparent.so
 LIBNZAEADAPTERS |
/nz/data.1.0/base/317943/library/319331/host/libnzaeadapters.so
 LIBNZAECHILD    | /nz/data.1.0/base/317943/library/319332/host/libnzaechild.so

(3 rows)

SELECT * FROM TABLE WITH FINAL (lib_env(0, 5));
            NAME                              VALUE
                                             +---------------
---------------------------------NZAE_RUNTIME_USER_QUERY  | 1
 NPS_PLATFORM                               | xs
 NZAE_OUTPUT_COLUMNS                        | 1,1
 NZAE_DEBUG_LEVEL                           | 0
 NZAE_REMOTE_NAME_DATA_SLICE                | 0

(5 rows)
```

Additional shared libraries can be added during the registration step, by using the **--deps** option along with the library name.

R Language Environment & Shared Libraries

Code

The environment and shared library classes provide information about the environment and libraries that are associated with an AE. This example AE takes one argument, an integer. If the integer is equal to 0, the AE outputs the environment that it sees. If the integer is equal 1, the AE outputs the libraries that it sees.

Enter the following code in the */tmp/libenv.R* file:

```
nz.fun <- function () {
  getNext() # this can be called only once
  input <- getInputColumn(0)
  if (input == 1) {
    apply(getLibraryInfo(), 1, function(ROW) {
      setOutput(0, ROW[1])
      setOutput(1, ROW[2])
      outputResult()
    })
  }
  else if (input == 0) {
    entry <- getFirstEnvironmentEntry()
    while (!is.null(entry)) {
      setOutput(0, entry[1])
      setOutput(1, entry[2])
      outputResult()
      entry <- getNextEnvironmentEntry()
    }
  }
  else
    stop('incorrect input: ', input)
}
```

Compilation

From the directory where the source file is, compile the code by using the udtf template:

```
/nz/export/ae/utilities/bin/compile_ae --language r --template compile
     \ --version 3 --db dev --user nz libenv.R
```

Registration

Register the example as follows:

```
/nz/export/ae/utilities/bin/register_ae --language r --version 3 \
  --template udtf --exe libenv.R --sig "lib_env(INT4)"           \
  --return "TABLE(name VARCHAR(1000), value VARCHAR(1000))"      \
  --db dev --user nz
```

Running

```
SELECT * FROM TABLE WITH FINAL(lib_env(1));
            NAME               |                                VALUE
-------------------------------

 INZA....LIBNZAEADAPTERS |
/nz/data.1.0/base/202701/library/202809/host/libnzaeadapters.so
 INZA....LIBNZAECHILD    |
/nz/data.1.0/base/202701/library/202811/host/libnzaechild.so
 INZA....LIBNZAEPARENT |
/nz/data.1.0/base/202701/library/202808/host/libnzaeparent.so (3
rows)
SELECT * FROM TABLE WITH FINAL(lib_env(0)) limit 5;
       NAME        |                    VALUE
                   +-------------------------------------------
-----------------R_LIBS_SITE |
 R_SESSION_TMPDIR | /tmp/Rtmpp0196y
 R_LIBS_USER      | ~/R/x86_64-unknown-linux-gnu-library/2.11
 TAR              | /bin/gtar
 TR               | /usr/bin/tr
(5 rows)
```

CHAPTER 13
Advanced Analytic Executable Topics

AE Environment Variables: Executables and Shared Libraries

You can use the **register_ae** function to set the executable path and to set the NZAE_EXECUTABLE_PATH environment variable. This section contains examples using AEs with AE-ENVs.

Simple C Language AE

For this example, a simple C-language AE is compiled into an executable called helloae. The AE export directory is located at /nz/export/ae. Compiling produces one version for the host and one for the SPUs:

```
/nz/export/ae/applications/dev/david/host/helloae
/nz/export/ae/applications/dev/david/spu/helloae
```

The AE environment variables point the AE system to the location of the executables:

```
NZAE_HOST_ONLY_NZAE_EXECUTABLE_PATH=/nz/export/ae/applications/dev/david/host/hel
loae
NZAE_SPU_ONLY_NZAE_EXECUTABLE_PATH=/nz/export/ae/applications/dev/david/spu/hello
ae
```

The AE-ENV key names consist of two parts: a root and a location prefix. In the previous example, the root is NZAE_EXECUTABLE_PATH. The prefixes are NZAE_HOST_ONLY and NZAE_SPU_ONLY.

A variable with the NZAE_HOST_ONLY location prefix is only visible on the host. The location prefix is stripped off and the host receives the value:

```
NZAE_EXECUTABLE_PATH=/nz/export/ae/applications/dev/david/host/helloae
```

Similarly, a variable with a NZAE_SPU_ONLY location prefix is visible only to the SPUs. Again, the location prefix is stripped off and the SPUs receive:

```
NZAE_EXECUTABLE_PATH=/nz/export/ae/applications/dev/david/spu/helloae
```

Note: The location prefixes may be used with any AE environment variable.

Simple C Language AE with Shared Libraries

In this example, the AE helloae uses a directory of shared libraries. Set the Linux LD_LIBRARY_PATH for the application. It can be modified because the AE runtime system also uses LD_LIBRARY_PATH for its own operation. First, place the shared libraries in the AE Export Directory Tree.

```
/nz/export/ae/applications/dev/david/libhost/lib*.so
/nz/export/ae/applications/dev/david/libspu/lib*.so
```

This code shows how to get the AE runtime system to append the directory to LD_LIBRARY_PATH:

```
NZAE_HOST_ONLY_NZAE_APPEND_LD_LIBRARY_PATH=/nz/export/ae/applications/dev/david/l
ibhost
NZAE_SPU_ONLY_NZAE_APPEND_LD_LIBRARY_PATH=/nz/export/ae/applications/dev/david/li
bspu
```

These AE-ENV definitions consist of two prefixes and a root. The root is LD_LIBRARY_PATH. The first prefix, which is required to be specified first, is the location prefix discussed above. The second is the append prefix NZAE_APPEND, which tells the AE to append this value to an existing value. If the key has no value then a new value is created. When appending, the AE system first looks for an AE-ENV of the target key name. If not found, the AE system looks in the OS process environment.

The same technique can be used to append to any environment variable, including PATH and CLASSPATH. You can use the prepend prefix, NZAE_PREPEND, to prepend to any AE-ENV or OS environment variable. The location prefix must be specified before an append or prepend prefix. The values appended or prepended are separated by the colon (:) character. The prefixes provide the AE programmer significant control over the environment where the process runs. Variable assignments are performed in a first pass. Append and prepend operations are performed in a second pass after all the assignments have been processed.

AE Environment Variable Include Files

The AE functionality enables inclusion of external files during AE registration. The included files contain environment variable assignments and may include other files as well. The included files can exist on the shared NFS file system or as text-only NPS shared library files. See Working with AE Shared Libraries for more information on NPS system shared library files. Inclusion takes place at runtime, not during registration.

Example

```
NZAE_INCLUDE_BEFORE_1=/nz/export/ae/applications/dev/dave/config/test1.txt
NZAE_INCLUDE_AFTER_1=/nz/export/ae/applications/dev/dave/config/test2.txt
NZAE_INCLUDE_AFTER_2=%[MY_TEXT_FILE]
```

The contents of file test1.txt and test2.txt are application-specific AE environment variable assignments. For example, test1.txt might include:

```
TESTVAR1=VALUE1
TESTVAR2=VALUE2
TESTVAR3=VALUE3
```

The contents of file test2.txt might include:

```
TESTVAR1=VALUE1NEW
TESTVAR4=VALUE2
TESTVAR5=VALUE3
```

In the case of this example, the variable assignment of TESTVAR1 to VALUE1NEW will override the first setting (in test1.txt) of VALUE1. Note that this is only true of include files, not straight registration. If the same variable name is registered twice (*not* using include files) the result is unpredictable. See Order of Variable Parsing for more information.

Top-level registration AE environment variables are those defined directly through **nzudxcompile** or **register_ae** during registration. Entries that start with NZAE_INCLUDE_BEFORE are included before the top-level registration environment variables; entries that start with NZAE_INCLUDE_AFTER are included after the top-level registration environment variables. Multiple "include before" and "include after" statements are added in alphabetical order; variables inside the include files are added in definition order. The top-level environment variables are processed in alphabetical order. This means that under AE, the registration environment variables have a definitive, predictable order. Later defined variables of the same name can replace earlier values. However, the top-level registration environment variable key names must not duplicate other top level variable key names.

While included files are processed in the order that variables are declared, top-level registration variables are processed in alphabetical order based on the value after any location prefixes are processed and removed.

Consider the following declarations:

```
VAR_C=val3
NZAE_HOST_ONLY_VAR_B=val2
VAR_A=val1
```

On the host, in an include file these variables are processed in the order VAR_C, VAR_B, VAR_A, but when set as top-level environment variables, they are processed in order VAR_A, VAR_B, VARC_C.

AE Environment Variable Substitution

The AE functionality supports a general Unix shell-like environment variable substitution of the form **%{key_name}**. Note that the **%** (percent sign) is used instead of the **$** (dollar sign). This distinction exists because the use of the dollar sign conflicts with shell script substitution in Linux scripts written to call **nzudxcompile** or **register_ae**. AE substitution happens at query runtime, not during registration.

An example text file called text1.txt contains the following environment entries.

```
TESTVAR1=VALUE1
TESTVAR2=VALUE2
TESTVAR3=VALUE3
```

Given the following environment variable assignments:

```
NZAE_INCLUDE_BEFORE_1=/nz/export/.../test1.txt
SUBTEST=%{TESTVAR1}
```

The value of SUBTEST at runtime is VALUE1

```
SUBTEST=VALUE1
```

The substitution process looks first for AE environment variables, then at AE system predefined environment variables, and then at Linux OS process variables.

The AE system predefined environment variables consist of the following:

Table 4: Predefined environment variables

Environment Variable	Definition
NZAE_RUNTIME_SESSION_ID	Session ID of the current request
NZAE_RUNTIME_DATA_SLICE_ID	Data slice ID of the current request
NZAE_RUNTIME_TRANSACTION_ID	Transaction ID of the current request
NZ_TMP_DIR	Location of the NPS temporary directory on the current machine

Example

```
MY_SESSIONID=%{NZAE_RUNTIME_SESSION_ID}
```

Assuming the session has an ID of 14776, the result after substitution is MY_SESSIONID=14776. If a variable name cannot be resolved, an error condition occurs. However, you can specify a default value to use if a variable is not found so that no error occurs.

For example:

```
%{VAR_NAME:default value}
```

In the above example, if VAR_NAME is not found, the expression evaluates to the value "default value". The first colon marks the end of the variable name and the beginning of the default value. White space around the colon becomes part of the variable name or default value. No variable substitution is performed on the default value itself.

CHAPTER 14
Remote Analytic Executables

Understanding Remote AEs

Remote AEs demonstrate advanced functionality. This section briefly introduces some of the concepts of remote AEs.

The life cycle of a remote AE can be user-controlled. Conversely, the life cycle of a local AE is controlled directly by the NPS. A remote AE involves the explicit use of concurrency mechanisms such as threading and forking. This section discusses remote AEs and local AEs and describes:

the remote AE communication addressing scheme

how remote AEs can be launched

how a running remote AE can be controlled

how a single AE can be written to be either local or remote, based on usage and purpose.

Comparing Local and Remote AEs

There are two behavioral models of the AE life cycle supported by the NPS, local AE and remote AE.

Local Model

The NPS system controls the complete life cycle of a local AE. At query time, the NPS system automatically launches the local AE in a new process. In a successful run, the AE processes input, produces output, and terminates normally. For unsuccessful runs, the NPS system terminates the AE if the query is complete, or has been terminated, and the AE has not shut down. In this model, the NPS system ensures that for each function call there is one running AE process per dataslice. The NPS system does not allow an *orphan AE*, that is, an AE in which the query has ended, to keep running. If the Linux command **ps -H -e** is executed while an AE is running, the AE is shown as a child process of the NPS system process. Local AEs have a lifespan that is less than a query and technically less than a

subset of a query. This life cycle model is similar to that of UDXs. A local AE always processes the input from exactly one SQL function call.

Connecting to a Local AE

The NPS system automatically connects the SQL function to exactly one AE per dataslice. Because the NPS system starts the AE processes *a priori* for the SQL function, it already has the information on which AE process to use for a given dataslice.

Remote Model

The NPS system does not control the life cycle of the remote AE processes. If the NPS system is used to launch the AE, it is launched daemon-style and disassociated from the NPS system process tree. A remote AE may have a life cycle that ranges from less than a subset of a query up to indefinite, as with a long running daemon. A remote AE processes many SQL function calls simultaneously unless an instance exists per session or dataslice and the AE is invoked only once per SQL statement.

Connecting to a Remote AE

A SQL function must connect to a running instance of a remote AE. It is required that a running instance exists on each SPU, on the host, or on both. There may be more than one instance of the same AE executable running on each machine at the same time.

If more than one instance is running, then a remote AE must have a name; that is, it must be addressable. There must also be a way to tell the NPS system to connect to a specific instance of a running remote AE for a given SQL function. Each running remote AE per operating system instance must have a unique address. In addition the address must be unique per SPU and unique on the host. However, an AE running on different SPUs may have the same address; for the same application (this is typically the case). The remote AE functionality exposes the two kinds of connections between the Netezza system and an AE: data and notification. The data connection is how data flows between an SQL function and an AE. In the case of a local AE, this is the only visible connection. The notification connection is explicitly visible in a remote AE. The primary purpose is to notify a running AE that it has a new data connection to service. It can also be used to ping the remote AE to determine if it is still running, or to request that the remote AE shut down or stop.

Remote Analytic Executable Addressability

A remote AE address is specified as a set of up to four values in the form:

```
Remote Name, [Dataslice ID], [Session ID], [Transaction ID]}
```

Table 5: Remote AE address variables

Value	Description
Remote Name	*Required*. A string name.
Dataslice ID	*Optional*. The ID of a dataslice.

Value	Description
Session ID	*Optional*. The ID of the current Netezza system session that has an nzsql connection.
Transaction ID	*Optional*. A transaction ID.

On the Netezza system side (where the SQL function is called), the AE is registered with settings that specify a target notification connection address to a remote AE process. When the SQL function is called, the Netezza system sends a notification message to the remote AE, which then establishes a data connection.

The following setting indicates that the SQL function connects to a remote AE and does not launch a local AE. (The default is 0, which specifies launch and connect to a local AE.) This value can also be set by the **–remote** option of register_ae.

```
NZAE_REMOTE=1
```

The following setting is the required string name portion of the remote AE address. This can also be set by the **--rname** option of register_ae.

```
NZAE_REMOTE_NAME=MYREMOTEAE
```

The next three settings determine if dataslice ID, session ID, and transaction ID, respectively, are part of the remote AE address. (They can also be set by the **--rdataslice**, **--rsession**, **and rtrans**, respectively, option of register_ae.)The default for these three settings is ", denoting that they are not part of the address. On the remote AE side, the AE API is used to create an AE connection point, which is created using the same address settings that the NPS side is expecting. The remote AE then uses the AE API to "listen" with this connection point.

```
NZAE_REMOTE_NAME_DATA_SLICE= {0 | 1}
NZAE_REMOTE_NAME_SESSION= {0 | 1}
NZAE_REMOTE_NAME_TRANSACTION= {0 | 1}
```

Remote AE Processing

The remote AE listening on an AE connection point receives notifications from the Netezza system that a new instance of an SQL function is being called. The remote AE then creates a new thread, obtains a thread from a pool, or forks a new process to handle the SQL function call. The thread or process handling the SQL function creates a data connection for that function. When the request is completed successfully, it is important that the remote AE calls "done" on the data connection and closes it. If an error occurs, it is important that the AE calls "error" on the data connection and then closes it. The same thread or process can be used to serially process SQL function calls so it is possible to use a thread pool or a process pool.

Some Remote AE Scenarios

Below are four possible scenarios that can be encountered when implementing AEs.

Scenario 1: Language Supports Threads

Using a language that supports threads, possibly Java or C++ with POSIX threads, you run one

instance of the AE daemon-style on each SPU and one on the host. Each SQL function is handled on a thread taken from a pool, so that requests are handled concurrently. In this case, the AE connection point address consists only of the string name and nothing else. The remote AE has one listener on this address.

Scenario 2: Language Does Not Support Threads

Using a language that does *not* support threads, does not support them well, or where threads are undesirable, use the same connection point as Scenario 1. To process SQL functions, you must fork and handle the SQL function in the new process. In this case, the listening process listens for AE environments, which are sets of AE environment variables that describe an AE SQL function request. After forking, the AE environment can be used to create an AE data connection.

Scenario 3: Use of a Single Instance

The solutions for the first two scenarios dictate that when you use the database you use the same single instance of the remote AE per SPU or host. If for security or other reasons you do not want others sharing AEs, you can add session ID to the remote AE address.

To do so, set NZAE_REMOTE_NAME_SESSION=1 (or the register_ae option **--rsession**) when registering the SQL function, and specify the session ID when you create the connection point in the remote AE.

Note that you must still use threads or fork new processes because multiple simultaneous SQL function call notifications are still received. This occurs because a SPU consists of multiple dataslices, and simultaneous function calls come from each dataslice.

You must launch the remote AE after the session starts and stop the remote AE before it ends. (Launching and stopping remote AEs is covered in Launching a Remote Analytic Executable.) Each session uses its own instance of the remote AE process.

Scenario 4: Dataslice in the Remote AE Address

There are two possible approaches that use dataslices in the remote AE address. The first is to launch one AE per dataslice on each SPU. The SQL function is registered with NZAE_REMOTE_NAME_DATA_SLICE=1 using (or the register_ae option **--rdataslice**). Each remote AE can use an API call to return the dataslice for which it was launched. It then listens on a remote address consisting of string name and dataslice ID. If there are four dataslices on a SPU, then there are four remote AE processes. Each remote AE inputs data only for its dataslice.

This approach works well for a programming language like C++, however it may not work as well if each remote AE process has very large resource requirements or is written in a programming language with a resource-intensive runtime environment. In this situation, there is one heavy AE process per dataslice; collectively the multiple processes may be consuming too many system resources, especially memory.

The other approach is a more complex way to use dataslices in the remote AE address. A single remote AE process can use multiple notification connections simultaneously, as long as they have different addresses. In Scenario 1, one notification listener was used for the entire process with a remote AE address string name. For that scenario, consider the case where you run a single query that runs on the SPUs. A remote AE running on a SPU gets multiple SQL function call notifications, one from each dataslice, on the SPU. The notifications are simultaneous but are processed serially by the listener. In Scenario 1, each notification was received and assigned to a thread in the pool to

service the SQL function. The data connections are processed simultaneously but some start before others because the notifications are processed individually.

For efficiency, you want one notification listener per dataslice. If there are four dataslices on a SPU, you create four threads, each listening to an AE connection point specified with an address built from remote string name and dataslice ID.

The SQL function is still registered with NZAE_REMOTE_NAME_DATA_SLICE=1 using the register_ae option **--rdataslice**.

Assume the remote name is "MY_REMOTEAE" and the dataslices on a given SPU are 17, 18, 19, and
In the AE process, the four notification threads are listening on AE connection point addresses:

```
{ name=MY_REMOTEAE, data slice id=17 } {
name=MY_REMOTEAE, data slice id=18 } {
name=MY_REMOTEAE, data slice id=19 } {
name=MY_REMOTEAE, data slice id=20 }
```

When a listening thread receives a notification of a new SQL function call, it retrieves a notification thread from the pool and assigns it to that SQL function request for the assigned dataslice.

Finally, you want a listener to handle AE job control commands. Job control allows **ping** and **stop** commands to be sent to a running remote AE. The connection point is:

```
{ name=MY_REMOTEAE }
```

There is no direct API call to determine how the single AE process running on the SPU determines which dataslices to listen for. There is a database view called _v_dual_dslice that contains one record for each dataslice for the entire database. For example:

```
SELECT * FROM _v_dual_dslice;

DSID
------
1
4
2
3
…
```

To use this view, register a scalar function that connects to the remote AE using the address string name. The scalar takes one integer argument and is called using column DSID in view _v_dual_dslice as an argument. After the remote AE is launched, this scalar function is called. The remote AE on each SPU receives one notification for each dataslice. When the remote AE opens the data connection, it obtains the dataslice for that request and uses that slice to start the listener for address string name, dataslice on a new thread. The scalar function is processed normally to return a value and close the data connection. Since the scalar function is being used to provide the remote AE information, the return value is irrelevant.

To summarize, in Scenario 1, you used one listener. In Scenario 4, approach one, you used one remote AE process per dataslice, each with one listener. For Scenario 4, approach two, you used five different notification listeners in a single remote AE process.

More about Remote AE

The term *flowability* refers to the principle that all dataslices in a query must be actively and

simultaneously processed. Failure to satisfy this requirement risks poor performance at best, deadlock at worst. However, this does not mean that there cannot be a delay, even a long one, between input and output requests. Records should not be processed from one dataslice while ignoring others; the dataslices must be processed in parallel.

The result is that every dataslice of every query requires its own thread of execution, using either a thread or a process fork. Anything less—for example, a task switching algorithm—is not sufficient since one dataslice could be in a blocked state while data is available on another.

For languages that support threads, the use of a thread pool can improve performance.

Launching a Remote Analytic Executable

There are three types of AE registration.

Local AE: For a local AE, the NPS launches the AE process.

Remote AE: For a remote AE, the Netezza system does not launch the AE process; it looks for a previously-running instance, listening on a specific remote AE connection point address. The key **register_ae** setting for a remote AE is **--remote**.

Remote AE Launch: The third type of registration is for launching a remote AE. A launch registration has elements from both local and remote registrations. You must specify the program to execute, as with a local registration. You must specify the connection point settings, as with a remote registration. The key **register_ae** setting for a remote launch registration is **--launch**.

A registration for launching an AE must always be for a table function with an argument signature of bigint and return TABLE(aeresult varchar(255)).

Remote AE Launch Examples

When the launch function is called, it starts the remote AE, similar to a Linux daemon. The remote AE is then ready for requests.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template udtf \
 --version 3
--sig "apply_launch_v1(bigint)"
--return "TABLE(aeresult varchar(255))"
--define java_class=org.netezza.education.ApplyDriverV4 --
launch
--rname apply
--level 1
```

To launch on the host:

```
SELECT aeresult FROM TABLE WITH FINAL(apply_launch_v1(0));
```

To launch on a SPU:

```
SELECT aeresult FROM _v_dual_dslice, TABLE WITH FINAL(apply_launch_v1(dsid));
```

The return column, **aeresult**, contains information about the launched remote AE, such as the remote AE process IDs, that may be useful for troubleshooting. If the launch fails, this function throws an exception.

Example

```
SELECT * FROM TABLE WITH FINAL(apply_launch_v1(0));
AERESULT
--------------------------------------------------------------------------------
------------------
tran: 2704 session: 17646 DATA slc: 0 hardware: 0 machine: hostname process:
24320 thread: 24321
```

For more information about registering a remote AE launch function, see Registering Analytic Executables. For information about controlling a running remote AE, see the next section, Controlling a Running Remote AE.

Controlling a Running Remote AE

The AE capability provides an SQL table function, **nzaejobcontrol**, that controls running remote AEs. The syntax is (with arguments described in the following table):

```
nzaejobcontrol <operation><dataslice for locus><connection point string
name><remote_name>{TRUE | FALSE}{NULL | 0 | <transaction ID>}{NULL | 0 |
<session ID>}
```

Table 6: nzaejobcontrol argument desciptions

Argument	Description
Operation to perform	Operations include pinging a remote AE, stopping a remote AE, returning Linux process information about a remote AE, and sending signals to a remote AE. There are commands for operating on a specific remote AE connection point and on a subset of connection points, including all of them.
Dataslice/locus of the remote AE (SPUs or host)	This argument is specified using literal integer zero (0) for host or a set of non-literal integer dataslice IDs for SPU. The set of dataslice IDs come from system view _v_dual_dslice (or something equivalent).
String name portion of the connection point address	For a single connection point command, this name must be specified. For a multiple connection point command, it may be NULL or '' (empty string), in which case it specifies all names.
Boolean (Dataslice ID)	TRUE means use dataslice ID in connection point address; FALSE means do not use it. For multiple connection point commands, this argument must be FALSE.
Bigint (Transaction ID)	NULL means do not use transaction ID. 0 means use the current transaction ID function that table function nzaejobcontrol was called in; greater than 0 means use the specified transaction ID.

Argument	Description
Bigint (Session ID)	NULL means do not use session ID. 0 means use the current session ID that table function nzaejobcontrol was called in; greater than 0 means use the specified session ID.

Single Connection Point Commands

Single connection point commands operate on exactly one connection point, which might consist of multiple processes on multiple machines. These are the command names that can be used in the operation argument of **nzaejobcontrol**.

ping—Sends a short message to a connection point to determine if it is active and responding. It returns data for all output columns listed below. (See Table Function Output Columns.)

ps—Returns process information about a remote AE listening on a connection point. This command can complete even if the AE is not responding on the connection point. It returns data for all output columns that do not require direct communication with the remote AE.

stop—Stops a remote AE listening on a connection point. If the remote AE process does not respond to the message, the remote AE process is sent a SIGKILL signal.

status—Returns status information from a remote AE listening on a connection point. By creating a remote AE message callback routine, a remote AE application can return its own customized message and return code. Certain Linux signal commands can be sent to remote AEs. The signal sent is indicated in the **operation** field of the table function nzaejobcontrol commands **sighup**, **sigint**, **sigkill**, **sigusr1**, **sigusr2**, **sigterm**, **sigcont**, and **sigstop**.

clean—Deletes temporary synchronization files left on the SPUs. This command is used after the **stop** command has completed. The disk space involved is usually insignificant, except when many different connection point names are being created using session ID and transaction ID.

Multiple Connection Point Commands

The single connection point commands act on exactly one connection point, which may be in use by multiple processes on the SPUs. Except for the **clean** command, all the nzaejobcontrol table function commands have a corresponding multi-connection point version.

For these commands, the connection point arguments to table function nzaejobcontrol act as a selection filter. For example, if remote name is specified as NULL or '' (empty string) the command operates on connection points with all names. It is therefore possible to send a command to all remote AEs or a set of remote AEs, for example, all remote AEs using a connection point created with a particular session ID.

These commands include **pingall**, **stopall**, **psall**, **statusall**, **sighupall**, **sigintall**, **sigkillall**, **sigusr1all**, **sigusr2all**, **sigtermall**, **sigcontall**, **sigstopall**. These commands operate similarly to the single connection point version, but instead for multiple AEs.

Table Function Output Columns

When using the nzaejobcontrol function in the INZA database directly in SQL, the selected column should be **aeresult**. The nzaejobcontrol function can also be used by a GUI program using SQL; therefore, information is also returned in individual columns. Not all commands set all return columns. Table function nzaejobcontrol returns zero to multiple rows.

Table 7: nzaejobcontrol returned columns

Column	Description
aeresult	Status message
aerc	Return code for the command: 0 = success, -1 = error, +1 = stop unsuccessful but subsequent kill signal successful
aetotal	Total rows returned for command, repeated in each row
aename	Connection point name
aedslice	Connection point dataslice ID
aesession	Connection point Session ID
aetrans	Connection point Transaction ID
aehostname	Host name AE process is running on
aepid	Linux process ID of remote AE
aecommand	AE command line
aetid	Linux thread ID that the connection point is being listened on
aebuild	The AE build number of the remote AE
aenzrepver	The nzrep version the remote AE is using
aevermismatch	If TRUE the AE build number or the nzrep version of the remote AE does not match the current the NPS system kit, otherwise FALSE.

Host Examples

The following examples illustrate how single commands work on the host. (They assume that an AE called my_remote_ae exists and can be called.)

Ping a remote AE not using dataslice ID in the connection point:

```
SELECT aeresult FROM TABLE WITH FINAL(inza..nzaejobcontrol('ping', 0,
```

```
     'my_remote_ae', false, NULL, NULL));
```

Ping a remote AE using dataslice ID in the connection point:

```
SELECT aeresult FROM TABLE WITH FINAL(inza..nzaejobcontrol('ping',
     0, 'my_remote_ae', true, NULL, NULL));
```

Ping all remote AEs:

```
SELECT aeresult FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
     NULL, false, NULL, NULL));
```

Ping all remote AEs with a connection point using session ID = 1000:

```
SELECT aeresult FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
     NULL, false, NULL, 1000));
```

SPU Examples

The following examples illustrate how single commands work on a SPU. (They assume that an AE called my_remote_ae exists and can be called.)

Ping a remote AE that has one instance per SPU:

```
SELECT aeresult FROM _v_dual_dslice, TABLE WITH
     final(inza..nzaejobcontrol('ping', dsid, 'my_remote_ae', false, NULL,
NULL));
```

Ping a remote AE that has one instance per dataslice:

```
SELECT aeresult FROM _v_dual_dslice, TABLE WITH
     final(inza..nzaejobcontrol('ping', dsid, 'my_remote_ae', true, NULL, NULL));
```

Ping all remote AEs:

```
SELECT aeresult FROM _v_dual_dslice, TABLE WITH
     final(inza..nzaejobcontrol('pingall', dsid, NULL, false, NULL, NULL));
```

Ping all remote Aes with a connection point using session ID = 1000:

```
SELECT aeresult FROM _v_dual_dslice, TABLE WITH
     final(inza..nzaejobcontrol('pingall', dsid, NULL, false, NULL, 1000));
```

Single Column Examples on the Host with Output

The following shows varied column results of pinging all remote AEs on the host:

```
SELECT aeresult FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
     NULL, false, NULL, NULL));

AERESULT
--------------------------------------------------------------------------------
netezzahost 24872 (datamatrix dataslc:-1 sess:-1 trans:-1) thread: 24873 AE
Build: 10 nzrep version: 9
netezzahost 30264 (testcapi dataslc:-1 sess:-1 trans:-1) thread: 30264 AE Build:
10 nzrep version: 9

SELECT aerc FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0, NULL,
false, NULL, NULL));

AERC
------
0
0

SELECT aetotal FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0, NULL,
```

```
false, NULL, NULL));

AETOTAL
---------
2
2

SELECT aename FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
NULL, false, NULL, NULL));
AENAME
------------
datamatrix
testcapi

SELECT aedslice FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
NULL, false, NULL, NULL));
AEDSLICE
----------
-1
-1

SELECT aesession FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
NULL, false, NULL, NULL));
AESESSION
-----------
-1
-1

SELECT aetrans FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
NULL, false, NULL, NULL));
AETRANS
---------
-1
-1

SELECT aehostname FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
NULL, false, NULL, NULL));
AEHOSTNAME
--------------
netezzahost
netezzahost

SELECT aepid FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
NULL, false, NULL, NULL));
AEPID
-------
24872
30264

SELECT aecommand FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
NULL, false, NULL, NULL));
AECOMMAND
------------------------------------------------------------
/nz/export/ae/languages/java/6.13.0/jdk1.6.0_13/bin/java
/nz/export/ae/aebin/host/testcapi
SELECT aetid FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
NULL, false, NULL, NULL));
AETID
-------
```

```
24873
30264

SELECT aebuild FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
NULL, false, NULL, NULL));
AEBUILD
---------
10
10

SELECT aenzrepver FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall', 0,
NULL, false, NULL, NULL));
AENZREPVER
------------
9
9

SELECT aevermismatch FROM TABLE WITH FINAL(inza..nzaejobcontrol('pingall',
0, NULL, false, NULL, NULL));
AEVERMISMATCH
---------------
f
f
```

Data Connection Management for Remote AEs

The nzaejobcontrol function includes functionality to manage remote AE data connections. This functionality is useful in situations where an issue arises with the AE, for example if a user-created AE fails or a session on the Netezza appliance abruptly terminates.

In either case, one side of an AE data connection could potentially be hung waiting for data. In addition, it is possible that temporary files used for data communication are left behind. Two nzaejobcontrol function commands—**connections** and **repair**—can be used to recognize and repair these problems.

The **connections** command lists all of the remote AE data connections, on either the SPUs or the host. Note that a single query running on the SPUs uses multiple data connections, one per dataslice, while a query running on the host uses only a single data connection.

The following examples are run on a system with one SPU consisting of four dataslices.

Here, the **connections** command is run with SPU locus.

```
SELECT * FROM _v_dual_dslice, TABLE WITH FINAL(inza..nzaejobcontrol('connections',
dsid, null, false, null, null));
 DSID |       AERESULT        | AERC | AETOTAL |   AENAME   | AEDSLICE | AESESSION
| AETRANS | AEHOSTNAME | AEPID |              AECOMMAND              | AETID |
AEBUILD | AENZREPVER | AEVERMISMATCH
                              +------+---------+------------+----------
             +----------+------------+-------+-----------------------------------
+  -------+ +------------          +--------------
    3|OK                     |    0 |       8 | datamatrix |        1 |     17205
|    43310| spu0101    | 25495 | /var/opt/nz/local/tmp/nzrep_D2zMNV |       |
|             |
    3|OK                     |    0 |       8 | datamatrix |        4 |     17205
|    43310| spu0101    | 25495 | /var/opt/nz/local/tmp/nzrep_wIJ5mV |       |
|             |
    3|OK                     |    0 |       8 | datamatrix |        2 |     17205
|    43310| spu0101    | 25495 | /var/opt/nz/local/tmp/nzrep_GWL2dV |       |
```

```
|    3 | OK     |                |    0 |      8 | datamatrix |         3 |    17205
|   43310  | spu0101    | 25495 | /var/opt/nz/local/tmp/nzrep_DpCZeV |       |
|            |
    3 | ERROR: (AE Stopped)   |   -1 |      8 | datamatrix |         3 |    17099
|   43304  | spu0101    | 18120 | /var/opt/nz/local/tmp/nzrep_HAyLm0 |       |
|            |
    3 | ERROR: (AE Stopped)   |   -1 |      8 | datamatrix |         2 |    17099
|   43304  | spu0101    | 18120 | /var/opt/nz/local/tmp/nzrep_uTHvp3 |       |
|            |
    3 | ERROR: (AE Stopped)   |   -1 |      8 | datamatrix |         4 |    17099
|   43304  | spu0101    | 18120 | /var/opt/nz/local/tmp/nzrep_oHGbE0 |       |
|            |
    3 | ERROR: (AE Stopped)   |   -1 |      8 | datamatrix |         1 |    17099
|   43304  | spu0101    | 18120 | /var/opt/nz/local/tmp/nzrep_vROdw0 |       |
|            |
(8 rows)
```

In this example two queries are running, which use a total of eight data connections. The query on session 17205 is running fine, but on session 17099, the remote AE has crashed and the Netezza system session is hung. For local AEs, the Netezza software can detect and resolve this situation; however the Netezza system does not manage remote AE processes and therefore cannot resolve the issue for the remote AE.

The problem can be corrected by running the nzaejobcontrol function's **repair** command. This command aborts any hung connections and cleans up leftover machine resources.

```
SELECT * FROM _v_dual_dslice, TABLE WITH FINAL(inza..nzaejobcontrol('repair', dsid,
null, false, null, null));
 DSID | AERESULT | AERC | AETOTAL |   AENAME   | AEDSLICE | AESESSION | AETRANS |
AEHOSTNAME | AEPID |              AECOMMAND              | AETID | AEBUILD |
AENZREPVER | AEVERMISMATCH---------
-----+----------       + ------+        +-------------+----------+-----------
+------------ +-------+-----------------------------------            +-------   +----------
+------------ +-------------
    1|OK         |     0 |       4 | datamatrix |         3|     17099 |   43304 |
spu0101    | 18120  | /var/opt/nz/local/tmp/nzrep_HAyLm0 |       |          |
|
    1|OK         |     0 |       4 | datamatrix |         2|     17099 |   43304 |
spu0101    | 18120  | /var/opt/nz/local/tmp/nzrep_uTHvp3 |       |          |
|
    1|OK         |     0 |       4 | datamatrix |         4|     17099 |   43304 |
spu0101    | 18120  | /var/opt/nz/local/tmp/nzrep_oHGbE0 |       |          |
|
    1|OK         |     0 |       4 | datamatrix |         1|     17099 |   43304 |
spu0101    | 18120  | /var/opt/nz/local/tmp/nzrep_vROdw0 |       |          |
|
(4 rows)
```

In this case, the hung connections for session 17099 are aborted and the original hung query returns an error message.

If the **connections** command is now run only the working connections remain:

```
SELECT * FROM _v_dual_dslice, TABLE WITH FINAL(inza..nzaejobcontrol('connections',
dsid, null, false, null, null));
 DSID | AERESULT | AERC | AETOTAL |   AENAME   | AEDSLICE | AESESSION | AETRANS |
AEHOSTNAME | AEPID |              AECOMMAND              | AETID | AEBUILD |
AENZREPVER | AEVERMISMATCH---------
-----+----------       + ------+        +-------------+----------+-----------+ ----------
+------------ +-------+-----------------------------------            +-------+----------
+------------ +-------------
    2|OK         |     0 |       4 | datamatrix |         1 |     17205 |   43310 |
spu0101    | 25495 | /var/opt/nz/local/tmp/nzrep_D2zMNV |       |          |

|
```

```
    2 | OK        |     0 |      4 | datamatrix |         4 |    17205  |   43310 |
spu0101     | 25495 | /var/opt/nz/local/tmp/nzrep_wIJ5mV |       |          |
|
    2 | OK        |     0 |      4 | datamatrix |         2 |    17205  |   43310 |
spu0101     | 25495 | /var/opt/nz/local/tmp/nzrep GWL2dV |       |          |
|
    2 | OK        |     0 |      4 | datamatrix |         3 |    17205  |   43310 |
spu0101     | 25495 | /var/opt/nz/local/tmp/nzrep_DpCZeV |       |          |
|
(4 rows)
```

If the **repair** command is run again there is no output since there is nothing to fix.

```
SELECT * FROM _v_dual_dslice, TABLE WITH FINAL(inza..nzaejobcontrol('repair', dsid,
  null, false, null, null));
 DSID | AERESULT | AERC | AETOTAL | AENAME | AEDSLICE | AESESSION | AETRANS |
AEHOSTNAME | AEPID | AECOMMAND | AETID | AEBUILD | AENZREPVER | AEVERMISMATCH
------+----------+------+---------+--------+----------+-----------+---------
+------------+-------+-----------+-------+---------+------------+---------------
(0 rows)
```

These commands can also be run on the host. The host locus versions of the commands are as follows:

```
SELECT * FROM TABLE WITH FINAL(inza..nzaejobcontrol('connections', 0,
null, false, null, null));
SELECT * FROM TABLE WITH FINAL(inza..nzaejobcontrol('repair', 0, null,
false, null, null));
```

Note: These commands apply only to data connections and do not directly stop AE processes. The manner in which the AE is written determines whether the AE stops when a data connection is aborted.

AEs that are both Local and Remote

For a simple local-only AE that does not use shapers, a direct initialize call can be used to get a data handle, such as the C API nzaeInitialize for a function AE and nzaeAggInitialize for an aggregate AE. However, if either shapers or remote AEs are being supported, the AE application must be able to determine how it is invoked.

All-In-One Approach

This example shows the general process for making a data connection. The language adapters have functionality to partially automate this procedure.

```
if (isLocal)
{
     dataConn = getLocalApi
        dataConn is the data connection
     switch (dataConn.apiType)
     {
        case FUNCTION: // Table or Scalar Function
            run(dataConn.function)
            close(dataConn.function)
            break
        case AGGREAGATE: // Aggregate
            Function run(dataConn.aggregate)
            close(dataConn.aggregate)
```

```
            break

         case SHAPER: // Table Shaper or Scalar Sizer
             run(dataConn.shaper)
             close(dataConn.shaper)
             break
      }
}
else
{
      // Remote AE
      connPoint = createConnectionPoint
      listener = createListener(connPoint)
         listener is the notification
      connection for (;;)
      {
         dataConn = acceptRemoteApi(listener)
         switch (dataConn.apiType)
         {
             case FUNCTION: // Table or Scalar
                Function run(dataConn.function)
                close(dataConn.function)
                break

             case AGGREGATE: // Aggregate Function
                run(dataConn.aggregate)
                close(dataConn.aggregate)
                break

             case SHAPER: // Table Shaper or Scalar Sizer
                run(dataConn.shaper)
                close(dataConn.shaper)
                break
         }
      }
         the following code may not be
     reached close(listener)
     close(connPoint)
}
```

Forking Approach

When using a programming language such as Java, which has solid thread support, a threaded approach is recommended. For languages without thread support or with limited thread support, the forking approach works well.

For a programming language that can support both threading and forking approaches, such as C and C++, the requirements of the application or support libraries determine the approach.

When using the forking approach, a data connection can be made under the remote AE. Instead of accepting an API, an AE environment can be accepted. An AE environment is not a data connection but can be used in an initialize call to create a data connection. To fork and use a data connection is not valid; however, to fork and use an AE environment is valid. An AE environment can be first serialized to bytes, then unserialized, and then be used to create a data connection. This example uses an AE environment with fork to run a data request in a new process:

```
for (;;)
{
   aeEnv = acceptRemoteEnvironment(listener)
   fork

   if(parent)
   {
```

```
                    in parent process
                close(aeEnv) continue
                loop
            }

               in child process
            freeResources(listener)
            close(connPoint) switch
            (aeEnv.apiType)
            {
                case FUNCTION: // Table or Scalar Function
                    conn = initializeFunctionAE(aeEnv)
                    run(conn)
                    close(conn)

                    exit

                case AGGREGATE: // Aggregate Function
                    conn = initializeAggregateAE(aeEnv)
                    run(conn)
                    close(conn)
                    close(env)
                    exit

                case SHAPER: // Table Shaper or Scalar Sizer
                    conn = initializeShaperAE(aeEnv)
                    run(conn)
                    close(conn)
                    exit
             }
        }
    }
```

When using Linux fork, avoid creating "zombie" Linux processes by either ignoring signal SIGCHLD or handling the signal and waiting on the child processes. Waiting on the child processes outside the signal handler is not recommended since the parent could become blocked and not handle subsequent AE notifications. When possible, ignoring the signal is the better approach. If the SIGCHLD signal is handled, care must be taken so that no child processes are missed and the parent does not become blocked.

CHAPTER 15

Simple Remote Mode Examples

Remote mode allows a single analytic executable to run as a “server,” serving answers to queries made by the user. These servers are homogenous and are started on the host, and/or on each dataslice or SPU. When a user makes an NPS system query that is registered as being handled by a remote AE, the NPS system then makes a request to the running remote AE "server" for each dataslice that must be handled.

The examples in this section illustrate the following concepts:

Scalar Functions

(Simple) Remote Mode

For R only: Global Assignment Operators

C Language Scalar Function (Remote Mode)

This example uses the following file name:

```
func.c
```

Code

The code in this example shows that, with minor changes, this program works in remote mode:

The change is in the main. The required remote AE connection name, gathered from the launcher, is set and then a loop is added around the accept so it is called five times. This code works like a local AE and handles five iterations as a remote AE.

```
#include <stdio.h>
#include <stdlib.h>

#include "nzaeapis.h"

static int run(NZAE_HANDLE h);
int main(int argc, char * argv[])
{
    if (nzaeIsLocal())
    {
```

```
    NzaeInitialization arg;
    memset(&arg, 0, sizeof(arg));
            arg.ldkVersion = NZAE_LDK_VERSION;
    if (nzaeInitialize(&arg))
    {
        fprintf(stderr, "initialization failed\n");
        return -1;
    }
    run(arg.handle);
    nzaeClose(arg.handle);
}
else {
    NZAECONPT_HANDLE hConpt = nzaeconptCreate();
    if (!hConpt)
    {
        fprintf(stderr, "error creating connection point\n");
        fflush(stderr);
        return -1;
    }
    const char * conPtName = nzaeRemprotGetRemoteName();
    if (!conPtName)
    {
        fprintf(stderr, "error getting connection point name\n");
        fflush(stderr);
        exit(-1);
    }
    if (nzaeconptSetName(hConpt, conPtName))
    {
        fprintf(stderr, "error setting connection point name\n");
        fflush(stderr);
        nzaeconptClose(hConpt);
        return -1;
    }
    NzaeremprotInitialization args;
    memset(&args, 0, sizeof(args));
    args.ldkVersion = NZAE_LDK_VERSION;
    args.hConpt = hConpt;
    if (nzaeRemprotCreateListener(&args))
    {
        fprintf(stderr, "unable to create listener - %s\n", \
             args.errorMessage);
        fflush(stderr);
        nzaeconptClose(hConpt);
        return -1;
    }
    NZAEREMPROT_HANDLE hRemprot = args.handle;
    NzaeApi api;
    int i;
    for (i = 0; i < 5; i++)
    {
        if (nzaeRemprotAcceptApi(hRemprot, &api))
        {
            fprintf(stderr, "unable to accept API - %s\n", \
                 nzaeRemprotGetLastErrorText(hRemprot));
            fflush(stderr);
            nzaeconptClose(hConpt);
            nzaeRemprotClose(hRemprot);
            return -1;
        }
        if (api.apiType != NZAE_API_FUNCTION)
        {
            fprintf(stderr, "unexpected API returned\n");
            fflush(stderr);
            nzaeconptClose(hConpt);
            nzaeRemprotClose(hRemprot);
            return -1;
        }
        printf("testcapi: accepted a remote request\n");
```

```
            fflush(stdout);
            run(api.handle.function);
        }
        nzaeRemprotClose(hRemprot);
        nzaeconptClose(hConpt);
    }
    return 0;
}

static int run(NZAE_HANDLE h)
{
    NzaeMetadata metadata;
    if (nzaeGetMetadata(h, &metadata))
    {
        fprintf(stderr, "get metadata failed\n");
        return -1;
    }

    #define CHECK(value) \
    { \
        NzaeRcCode rc = value; \
        if (rc) \
        { \
            const char * format = "%s in %s at %d"; \
            fprintf(stderr, format,  \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            nzaeUserError(h, format, \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            exit(-1); \
        } \
    }

    if (metadata.outputColumnCount != 1 ||
        metadata.outputTypes[0] != NZUDSUDX_DOUBLE)
    {
        nzaeUserError(h, "expecting one output column of type
        double"); nzaeDone(h);
        return -1;
    }

    if (metadata.inputColumnCount < 1)
    {
        nzaeUserError(h, "expecting at least one input column");
        nzaeDone(h);
        return -1;
    }

    if (metadata.inputTypes[0] != NZUDSUDX_FIXED
        && metadata.inputTypes[0] != NZUDSUDX_VARIABLE)
    {
        nzaeUserError(h, "first input column expected to be a string
        type"); nzaeDone(h);
        return -1;
    }

    for (;;)
    {
        int i;
        double result = 0;
        NzaeRcCode rc = nzaeGetNext(h);
        if (rc == NZAE_RC_END)
        {
            break;
        }

        NzudsData * input = NULL;
        CHECK(nzaeGetInputColumn(h, 0, &input));
        const char * opString;
```

```
if (input->isNull)
{
    nzaeUserError(h, "first input column may not be
    null"); nzaeDone(h);
    return -1;
}
if (input->type == NZUDSUDX_FIXED)
{
    opString = input->data.pFixedString;
} else if (input->type == NZUDSUDX_VARIABLE)
{
    opString = input-
>data.pVariableString; } else
{
    nzaeUserError(h, "first input column expected to be a string type");
    nzaeDone(h);
    return -1;
}

enum OperatorEnum
{
    OP_ADD = 1,
    OP_MULT = 2
} op;

if (strcmp(opString, "+") == 0)
{
    op = OP_ADD;
} else if (strcmp(opString, "*") == 0)
{
    op = OP_MULT;
    result = 1;
} else
{
    nzaeUserError(h, "unexpected operator %s", opString);
    nzaeDone(h);
    return -1;
}

for (i = 1; i < metadata.inputColumnCount; i++)
{
    CHECK(nzaeGetInputColumn(h, i, &input));
    if (input->isNull)
    {
        continue;
    }
    switch (op)
    {
    case OP_ADD:
        switch(input->type)
        {
            case NZUDSUDX_INT8:
                result += *input->data.pInt8;
                break;
            case NZUDSUDX_INT16:
                result += *input->data.pInt16;
                break;
            case NZUDSUDX_INT32:
                result += *input->data.pInt32;
                break;
            case NZUDSUDX_INT64:
                result += *input->data.pInt64;
                break;
            case NZUDSUDX_FLOAT:
                result += *input->data.pFloat;
                break;
            case NZUDSUDX_DOUBLE:
                result += *input->data.pDouble;
```

```
                        break;

                    case NZUDSUDX_NUMERIC32:
                    case NZUDSUDX_NUMERIC64:
                    case NZUDSUDX_NUMERIC128:
                           unlike Java, C has no native numeric data
                        types break;
                    default:
                           ignore nonnumeric
                        type break;
                }
                break;
            case OP_MULT:
                switch(input->type)
                {
                    case NZUDSUDX_INT8:
                        result *= *input->data.pInt8;
                        break;
                    case NZUDSUDX_INT16:
                        result *= *input->data.pInt16;
                        break;
                    case NZUDSUDX_INT32:
                        result *= *input->data.pInt32;
                        break;
                    case NZUDSUDX_INT64:
                        result *= *input->data.pInt64;
                        break;
                    case NZUDSUDX_FLOAT:
                        result *= *input->data.pFloat;
                        break;
                    case NZUDSUDX_DOUBLE:
                        result *= *input->data.pDouble;
                        break;

                    case NZUDSUDX_NUMERIC32:
                    case NZUDSUDX_NUMERIC64:
                    case NZUDSUDX_NUMERIC128:
                           unlike Java, C has no native numeric data
                        types break;
                    default:
                           ignore non-numeric type
                        break;
                }
                break;
            default:
                nzaeUserError(h, "internal error, unexpected operator");
                nzaeDone(h);
                return -1;
            }
        }

        CHECK(nzaeSetOutputDouble(h, 0, result));
        CHECK(nzaeOutputResult(h));
    }
    nzaeDone(h);
    return 0;
}
```

Compilation

Use the standard compile, as in local mode.

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language system --version 3 \
     --template compile func.c --exe apply
```

Registration

Registration is slightly different. First, run this to register:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3 \
     --template udf --exe apply --sig "rem_applyop_c(VARARGS)" \
     --return "double" --remote --rname testcapi
```

Then run the following. It specifies the code be registered as a remote AE with a remote name of testcapi, which matches the code. In addition, it registers a launcher:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3 \
     --template udtf --exe apply --sig "rem_applyop_launch(int8)" \
     --return "TABLE(aeresult varchar(255))" --remote --rname testcapi --launch
```

With the exception of the **--rname**, **--exe** and the name portion of **--sig**, all launchers look like the above. In this case it is a UDTF, since that is the interface for all launchers. The value of **--rname** must match the name in the code.

Running

To run the AE in remote mode, the executable is run as a "server." In this instance it handles only queries run on the host. Usually, AEs are started on the SPUs as well. Start an instance on the host:

```
SELECT * FROM TABLE WITH FINAL(rem_applyop_launch(0));
                                                    AERESULT
-------------------------------------------------------------------------------
--
 tran: 7788 session: 16354 DATA slc: 0 hardware: 0 machine: bdrosendev process:
13117 thread: 13117
(1 row)
```

Notice that a different syntax is used to invoke the table function style launcher. This is the syntax used to call any UDTF-based AE. Now run the AE:

```
SELECT rem_applyop_c('+',1,2);
 REM_APPLYOP_C
---------------
             3
(1 row)
```

Finally, cause an error:

```
SELECT rem_applyop_c(1);
ERROR:  first input COLUMN expected TO be a string type
```

C++ Language Scalar Function (Remote Mode)

This example uses the following file name:

```
applyopcpp.cpp
```

Code

The code in this example shows that, with minor changes, this program works in remote mode:

The change is in the main. The required remote AE connection name, testcapi, is set and then a loop is added around getAPI so it is called two times. This code works like a local AE and handles two iterations as a remote AE.

```
#include <nzaefactory.hpp>

using namespace nz::ae;

static int run(nz::ae::NzaeFunction *aeFunc);
```

```
int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
       The following line is only needed if a launcher is not
    used helper.setName("testcapi");
    for (int i=0; i < 2; i++) {
        nz::ae::NzaeApi &api =
        helper.getApi(nz::ae::NzaeApi::FUNCTION); run(api.aeFunction);
        if (!helper.isRemote())
            break;
    }

    return 0;
}

class MyHandler : public NzaeFunctionMessageHandler
{
public:
    MyHandler() {
        m_Once = false;
    }
    enum OperatorEnum
    {
        OP_ADD = 1,
        OP_MULT = 2
    } op;

    void doOnce(NzaeFunction& api) {
        const NzaeMetadata& meta = api.getMetadata();
        if (meta.getOutputColumnCount() != 1 ||
            meta.getOutputType(0) != NzaeDataTypes::NZUDSUDX_DOUBLE) {
            throw NzaeException("expecting one output column of type double");
        }
        if (meta.getInputColumnCount() < 1) {
            throw NzaeException("expecting at least one input column");
        }
        if (meta.getInputType(0) != NzaeDataTypes::NZUDSUDX_FIXED &&
            meta.getInputType(0) != NzaeDataTypes::NZUDSUDX_VARIABLE) {
            throw NzaeException("first input column expected to be a string
type");
        }
        m_Once = true;
    }

        void evaluate(NzaeFunction& api, NzaeRecord &input, NzaeRecord &result) {

        if (!m_Once)
            doOnce(api);
        double res = 0;

        NzaeField &field = input.get(0);
        if (field.isNull())
            throw NzaeException("first input column may not be null");
        NzaeStringField &sf = (NzaeStringField&) input.get(0);
        std::string strop = (std::string)sf;
        if (strop == "+")
            op = OP_ADD;
        else if (strop == "*") {
            op = OP_MULT;
            res = 1;
        }
        else
          throw NzaeException(NzaeException::format("unexpected operator %s",
                \ strop.c_str()));
        for (int i=1 ; i < input.numFields(); i++) {
```

```
            NzaeField &inf = input.get(i);
            if (inf.isNull())
                continue;
            double temp = 0;
            bool ignore = false;
            switch (inf.type()) {
                case NzaeDataTypes::NZUDSUDX_INT8:
                {
                    NzaeInt8Field &sf = (NzaeInt8Field&)input.get(i);
                    temp = (int8_t)sf;
                    break;
                }
                case NzaeDataTypes::NZUDSUDX_INT16:
                {
                    NzaeInt16Field &sf = (NzaeInt16Field&)input.get(i);
                    temp = (int16_t)sf;
                    break;
                }
                case NzaeDataTypes::NZUDSUDX_INT32:
                {
                    NzaeInt32Field &sf = (NzaeInt32Field&)input.get(i);
                    temp = (int32_t)sf;
                    break;
                }
                case NzaeDataTypes::NZUDSUDX_INT64:
                {
                    NzaeInt64Field &sf = (NzaeInt64Field&)input.get(i);
                    temp = (int64_t)sf;
                    break;
                }
                case NzaeDataTypes::NZUDSUDX_FLOAT:
                {
                    NzaeFloatField &sf = (NzaeFloatField&)input.get(i);
                    temp = (float)sf;
                    break;
                }
                case NzaeDataTypes::NZUDSUDX_DOUBLE:
                                {
                    NzaeDoubleField &sf =
                    (NzaeDoubleField&)input.get(i); temp = (double)sf;
                    break;
                }
                case NzaeDataTypes::NZUDSUDX_NUMERIC32:
                case NzaeDataTypes::NZUDSUDX_NUMERIC64:
                case NzaeDataTypes::NZUDSUDX_NUMERIC128:
                {
                    NzaeNumericField &sf =
                    (NzaeNumericField&)input.get(i); temp = (double)sf;
                    break;
                }
            default:
                ignore = true;
                   ignore
                break;
            }
            if (!ignore) {
                if (op == OP_ADD)
                    res += temp;
                else
                    res *= temp;
            }
        }
        NzaeDoubleField &f =
        (NzaeDoubleField&)result.get(0); f = res;
    }
    bool m_Once;
};
static int run(NzaeFunction *aeFunc)
```

```
{
    aeFunc->run(new MyHandler());
    return 0;
}
```

Compilation

Use the standard compile, as in local mode:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp --template compile
     \ --exe applyopcpp --compargs "-g -Wall" --linkargs "-g" applyopcpp.cpp
     \ --version 3
```

Registration

Registration is slightly different. First run this to register:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "rem_applyop_cpp(VARARGS)" \
     --return "double" --language cpp --template udf --exe applyopcpp \
     --version 3 --remote --rname testcapi
```

Then run the following. It specifies the code be registered as a remote AE with a remote name of testcapi, which matches the code. In addition, it registers a launcher:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "rem_applyop_launch(int8)" \
     --return "TABLE(aeresult varchar(255))" --language cpp --template udtf \
     --exe applyopcpp  --version 3 --remote --launch --rname testcapi
```

With the exception of the **--rname**, **--exe** and the name portion of **--sig**, all launchers look like the above. In this case it is a UDTF since that is the interface for all launchers. The value of **--rname** must match the name in the code.

Running

To run the AE in remote mode, the executable is run as a "server." In this instance, it handles only queries run on the host. Usually, AEs are started on the SPUs as well. Start an instance on the host:

```
SELECT * FROM TABLE WITH
FINAL(rem_applyop_launch(0)); AERESULT
--------------------------------------------------------------------------------
tran: 7192 DATA slc: 0 hardware: 0 machine: spubox1 process: 8306 thread:
8306 (1 row)
```

Notice that a different syntax is used to invoke the table function style launcher. This is the syntax used to call any UDTF-based AE. Now run the AE:

```
SELECT rem_applyop_cpp('+', 4,5,1.1);
REM_APPLYOP_CPP
-----------------
10.1
(1 row)
```

Finally, cause an error:

```
SELECT rem_applyop_cpp(1);
ERROR: first input COLUMN expected TO be a string type
```

Java Language Scalar Function (Remote Mode)

This example uses the following file name:

```
TestJavaInterface.java
```

Code

The code in this example shows that, with minor changes, this program works in remote mode. The change is in the main where a loop is added around getAPI so that it gets called two times. If the connection is remote, it is handed off to be run by a new thread and to wait for a new connection. Finally, if the connection is not remote, break out of the loop after the first iteration. This code works like a local AE and handles two iterations as a remote AE.

```
import org.netezza.ae.*;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class TestJavaInterface {

    private static final Executor exec =
        Executors.newCachedThreadPool();


    public static final void main(String [] args) {
        try {
            mainImpl(args);
        } catch (Throwable t) {
            System.err.println(t.toString());
            NzaeUtil.logException(t, "main");
        }
    }

    public static final void mainImpl(String [] args) {

        NzaeApiGenerator helper = new NzaeApiGenerator();

        while (true) {
            final NzaeApi api = helper.getApi(NzaeApi.FUNCTION);
            if (api.apiType == NzaeApi.FUNCTION) {
                if (!helper.isRemote()) {
                    run(api.aeFunction);
                    break;
                } else {
                    Runnable task = new Runnable() {
                            public void run() {
                                try {
                                    TestJavaInterface.run(api.aeFunction);
                                } finally {
                                    api.aeFunction.close();
                                }
                            }
                        };
                    exec.execute(task);
                }
            }
        }
        helper.close();
    }

    public static class MyHandler implements NzaeMessageHandler
    {
        public void evaluate(Nzae ae, NzaeRecord input, NzaeRecord output) {
```

```
                final NzaeMetadata meta = ae.getMetadata();

                int op = 0;
                double result = 0;

                if (meta.getOutputColumnCount() != 1 || meta.getOutputNzType(0)
                    != NzaeDataTypes.NZUDSUDX_DOUBLE) { throw new
                    NzaeException("expecting one output column of type
double");
                }
                if (meta.getInputColumnCount() < 1) {
                    throw new NzaeException("expecting at least one input column");
                }
                if (meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_FIXED &&
                    meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_VARIABLE) {
                    throw new NzaeException("first input column expected to be a
string type");
                }

                for (int i = 0; i < input.size(); i++) {
                    if (input.getField(i) == null) {
                        continue;
                    }
                    int dataType = meta.getInputNzType(i);
                    if (i == 0) {
                        if (!(dataType == NzaeDataTypes.NZUDSUDX_FIXED
                                || dataType == NzaeDataTypes.NZUDSUDX_VARIABLE))

                            { ae.userError("first column must be a string");

                        }
                        String opStr = input.getFieldAsString(0);
                        if (opStr.equals("*")) {
                            result = 1;
                            op = OP_MULT;
                        }
                        else if (opStr.equals("+")) {
                            result = 0;
                            op = OP_ADD;
                        }
                        else {
                            ae.userError("invalid operator = " + opStr);
                        }
                        continue;
                    }
                    switch (dataType) {
                        case NzaeDataTypes.NZUDSUDX_INT8:
                        case NzaeDataTypes.NZUDSUDX_INT16:
                        case NzaeDataTypes.NZUDSUDX_INT32:
                        case NzaeDataTypes.NZUDSUDX_INT64:
                        case NzaeDataTypes.NZUDSUDX_FLOAT:
                        case NzaeDataTypes.NZUDSUDX_DOUBLE:
                        case NzaeDataTypes.NZUDSUDX_NUMERIC32:
                        case NzaeDataTypes.NZUDSUDX_NUMERIC64:
                        case NzaeDataTypes.NZUDSUDX_NUMERIC128:
                            switch (op) {
                                case OP_ADD:
                                    result +=
input.getFieldAsNumber(i).doubleValue();
                                    break;
                                case OP_MULT:
                                    result *=
input.getFieldAsNumber(i).doubleValue();
                                    break;
                                default:
                                    break;
                            }
                            break;
```

```
                    default:
                        break;
                }
            } // end of column for loop
            output.setField(0, result);
        }

    }
    private static final int OP_ADD = 1;
    private static final int OP_MULT = 2;

    public static int run(Nzae ae)
    {
        ae.run(new MyHandler());
        return 0;
    }
}
```

Compilation

Use the standard compile as in local mode:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java --template
     \ compile TestJavaInterface.java --version 3
```

Registration

Registration is slightly different. First run this to register:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "rem_applyop_java(varargs)" \
     --return "double" --class AeUdf --language java --template udf \
     --define "java_class=TestJavaInterface" --version 3 --remote \
     --rname testjavapi
```

This specifies the code be registered as a remote ae with a remote name of testjavapi, which matches the code. In addition, a launcher is registered:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "rem_applyop_launch(int8)"
     \ --return "TABLE(aeresult varchar(255))" --class AeUdtf --language java
     \ --template udtf --define "java_class=TestJavaInterface" --version 3 \
     --remote --rname testjavapi --launch
```

With the exception of the **--rname**, **--exe** and the name portion of **--sig**, all launchers look like the above. In this case it is a UDTF, since that is the interface for all launchers. The value of **--rname** must match the name in the code.

Running

To run the AE in remote mode, the executable is run as a “server.” In this instance it handles only queries run on the host. Usually, AEs are started on the SPUs as well. Start an instance on the host:

```
SELECT * FROM TABLE WITH FINAL(rem_applyop_launch(0));
                              AERESULT
-----------------------------------------------------------------------
 tran: 14896 session: 18286 DATA slc: 0 hardware: 0 machine: bdrosendev process:
15937 thread: 15938
(1 row)
```

Notice that a different syntax is used to invoke the table function style launcher. This is the syntax used to call any UDTF-based AE. Now run the AE:

```
SELECT rem_applyop_java('+', 4,5,1.1);
 REM_APPLYOP_JAVA
------------------
             10.1
(1 row)
```

Finally, cause an error:

```
SELECT rem_applyop_java(1);
ERROR:  first input column expected to be a string type
```

Fortran Language Scalar Function (Remote Mode)

Code

The code to run in remote mode is the same as the code for local mode. See Fortran Language Scalar Function.

Compilation

Use the standard compile, as in local mode:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language fortran --version 3 \
     --template compile --exe applyopFortran applyop.f
```

Registration

Registration is slightly different:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language fortran --version 3 \
     --template udf --exe applyopFortran \
     --sig "remote_applyop(varchar(1), int4, int4)" --return int4 --remote
     \ --rname applyOpRemote
```

In addition, a launcher is registered:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language fortran --version 3
     \ --exe applyopFortran --sig "launch_remote_applyop(int8)" \
     --return "TABLE(aeresult varchar(255))" --template udtf --remote
     \ --launch --rname applyOpRemote
```

With the exception of the **--rname**, **--exe** and the name portion of **--sig**, all launchers look like the above. In this case it is a UDTF since that is the interface for all launchers. The **--rname** must match the name in the code (if specified).

Running

To run the AE in remote mode, the executable is run as a “server.” In this instance it handles only queries run on the host. Usually, AEs are started on the SPUs as well. Internally, after the server AE gets a request from the NPS system, it "forks" before it calls nzaeHandleRequest(). Start the executable running on the host using the launch command from nzsql:

```
SELECT * FROM TABLE WITH
FINAL(launch_remote_applyop(0)); AERESULT
--------------------------------------------------------------------------------
tran: 7192 DATA slc: 0 hardware: 0 machine: spubox1 process: 8306 thread:
8306 (1 row)
```

Running ps -aef | grep applyOpFortran shows that the executable is running on the host. Now run the AE:

```
SELECT remote_applyop('*', 2, 13);
 REMOTE_APPLYOP
----------------
             26
```

Python Language Scalar Function (Remote Mode)

Code

The code to run in remote mode is the same as the code for local mode. See Python Language Scalar Function.

Deployment

Use the standard deployment, as in local mode:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language python64
     \ --template deploy ./applyop.py --version 3
```

Registration

To register the Python file for remote mode, run:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3
     \ --template udf --exe applyop.py \
     --sig "remote_applyop(varchar(1), int4, int4)" --return int4 --remote
     \ --rname applyOpRemote
```

In addition, a launcher is registered:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3
     \ --template udtf --exe applyop.py --sig "launch_remote_applyop(int8)"
     \ --return "TABLE(aeresult varchar(255))" --remote --launch \
     --rname applyOpRemote
```

With the exception of **--rname**, **--exe** and the name portion of **--sig**, all launchers look like this example. The launcher itself is a UDTF and not a UDF since that is the interface for all launchers. The **--rname** must match up to the name in the code.

Running

To run the AE in remote mode, the executable is run as a “server.” In this instance it handles only queries run on the host. Usually, AEs are started on the SPUs as well. Internally, after the server AE gets a request from the NPS system, it "forks" before it calls nzaeRun(). Start the executable running on the host using the launch command from nzsql:

```
SELECT * FROM TABLE WITH
FINAL(launch_remote_applyop(0)); AERESULT
--------------------------------------------------------------------------
tran: 7192 DATA slc: 0 hardware: 0 machine: spubox1 process: 8306 thread:
8306 (1 row)</pre>
```

Running ps -aef | grep applyop shows that the executable is running on the host. Now run the AE:

```
SELECT remote_applyop('*', 2, 13);
 REMOTE_APPLYOP
----------------
             26
```

Perl Language Scalar Function (Remote Mode)

Code

The code to run in remote mode is the same as the code for local mode. See Perl Language Scalar Function.

Deployment

Use the standard deployment, as in local mode:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language perl --version 3
\ --template deploy ApplyOp.pm
```

Registration

To register the Perl file for remote mode, run:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3
     \ --template udf --exe ApplyOp.pm \
     --sig "remote_applyoppm(varchar(1), int4, int4)" --return int4
     \ --remote --rname applyOpRemotePm
```

In addition, a launcher is registered:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3 \
     --template udtf --exe ApplyOp.pm --sig "launch_remote_applyoppm(int8)" \
     --return "TABLE(aeresult varchar(255))" --remote --launch \
     --rname applyOpRemotePm
```

With the exception of **--rname**, **--exe** and the name portion of **--sig**, all launchers look like this example. The launcher itself is a UDTF and not a UDF since that is the interface for all launchers. The **--rname** must match up to the name in the code.

Running

To run the AE in remote mode, the executable is run as a "server." In this instance it handles only queries run on the host. Usually, AEs are started on the SPUs as well. Internally, after the server AE gets a request from the NPS system, it forks before it calls nzaeRun(). Start the executable running on the host using the launch command from nzsql on the system database:

```
SELECT * FROM TABLE WITH FINAL(launch_remote_applyoppm(0));
```

```
                                AERESULT
-------------------------------------------------------------------------
 tran: 296722 session: 16262 data slc: 0 hardware: 0 machine: vnairsim process:
19842 thread: 19842
(1 row)
```

Running **ps -aef | grep ApplyOp** shows that the executable is running on the host. Now run the AE:

```
 SELECT remote_applyoppm('*', 3, 32);
 REMOTE_APPLYOPPM
------------------
               96
(1 row)
```

The running AE can be stopped as follows:

```
SELECT aeresult FROM TABLE WITH FINAL(inza..nzaejobcontrol('stop',
0, 'applyOpRemotePm',false, NULL, NULL));
                                AERESULT
-------------------------------------------------------------------------
 vnairsim 24631 (applyOpRemotePm dataslc:-1 sess:-1 trans:-1) AE stopped
(1 row)
```

R Language Table Function (Remote Mode) 1

Concepts

In R, the connection point setup is handled automatically for the user through the handleConnection() function, which is a part of the nzrserver package. The user needs to provide only the function itself, optionally with its shaper/sizer. Hence, the *remote* implementation does not differ from its local equivalent.

Code

For the implementation of *applyop* in R, use the same code that is shown in R Language Scalar Function 1.

Compilation

The compilation step is identical to the one in R Language Scalar Function 1.

Registration

Register the launcher and the function as follows:

```
/nz/export/ae/utilities/bin/register_ae --language r --version 3 \
  --template udtf --exe /tmp/applyop.R --sig "launch_remote_applyop(INT8)" \
  --return "TABLE(aeresult varchar(255))" --db dev --user nz \
  --remote --rname applyOpRemote --launch

/nz/export/ae/utilities/bin/register_ae --language r --version 3   \
  --template udf --exe /tmp/applyop.R --return INT4  --db dev --user nz \
  --sig "remote_applyop(VARCHAR(1), INT4, INT4)"                  \
  --remote --rname applyOpRemote
```

Running

To run the remote R AE, first launch the remote process as follows:

```
SELECT * FROM TABLE WITH FINAL(launch_remote_applyop(0));
                                                    AERESULT
------------------------------------------------------------------------------
-----------------
 tran: 21764 session: 16231 data slc: 0 hardware: 0 machine: netezza process:
14611 thread: 14611
(1 row)
```

Then start the computations as follows:

```
SELECT remote_applyop('*', 2, 13);
 REMOTE_APPLYOP
----------------
             26
(1 row)
```

Finally, stop the remote process as follows:

```
SELECT * FROM TABLE WITH FINAL(inza..nzaejobcontrol('stop',0,
     'applyOpRemote',false,NULL, NULL));
                                AERESULT                                |AERC|.
. .

.-------------------------------------------------------------------------. +------+ .
 netezza  14611 (applyOpRemote dataslc:-1 sess:-1 trans:-1) AE stopped |    0 | .
. .
```

R Language Table Function (Remote Mode) 2

This example shows how you can use the remote mode to save the data processing state between subsequent queries.

Code

Define the usual set of R AE objects by saving them in the */tmp/remote.R* file as follows:

```
nz.fun <- function () {
  if (!exists('x', envir=.GlobalEnv))
    x <<- 0
  x <<- x + 1
  getNext()
  setOutput(0, x)
  outputResult()
}
```

Here, the *global assignment* operator <<- ensures that the same counter x is available in all calls.

Compilation

The compilation is identical to the one from the local mode:

```
/nz/export/ae/utilities/bin/compile_ae --language r \
```

```
  --template compile --version 3 --db dev --user nz /tmp/remote.R
```

Registration

Register the launcher as follows:

```
/nz/export/ae/utilities/bin/register_ae --language r \
  --version 3 --template udtf --db dev --user nz \
  --level 4  --mask DEBUG --sig "remote_rae_launch(int8)" \
  --return "TABLE(aeresult varchar(255))" \
  --remote --rname remote_rae --launch
Next, register the actual data-processing interface:
/nz/export/ae/utilities/bin/register_ae --language r \
  --version 3 --template udtf --db dev --user nz --level 4 \
  --mask DEBUG --lastcall 'table final' --exe remote.R \
  --sig "remote_rae(VARARGS)" --return "TABLE(cnt DOUBLE)" \
  --remote --rname remote_rae
```

Running

Running a remote AE requires two steps.

In the first step, launch the remote AE as follows:

```
SELECT * FROM TABLE WITH FINAL(REMOTE_RAE_LAUNCH(0));
```

Sample output might look like the following:

```
 AERESULT
----------------------------------------------------
 tran: 91488 session: 16067 data slc: 0 hardware: 0 \
 machine: netezza process: 13230 thread: 13230
(1 row)
```

Now run the actual data processing query:

```
SELECT * FROM TABLE WITH FINAL(remote_rae(0));
 CNT
-----
   1
(1 row)
```

Subsequent calls should return 2, 3, and so on, as the x global object is incremented each time the remote AE is invoked.

R Language Shapers & Sizers with Remote AEs

If the */tmp/remote.R* file is updated to include the last two lines of the code sample below, the remote R AE can be registered and run with TABLE(ANY) specified as its output.

```
nz.fun <- function () {
  if (!exists('x', envir=.GlobalEnv))
    x <<- 0
  x <<- x + 1
  getNext()
  setOutput(0, x)
  outputResult()
}
nz.shaper <- 'std'
nz.shaper.list <- list(value=NZ.DOUBLE)
```

Note that the new variables, nz.shaper and nz.shaper.list are parsed only if the UDAP is registered with TABLE(ANY). As a result, there is no need to change the compiled file name as the file is overwritten.

Compilation is therefore the same as in the Remote Table Function case. Registration of the launcher UDX is also the same. However, the data-processing interface registration has TABLE(ANY) as its output signature and a different UDX name:

```
/nz/export/ae/utilities/bin/register_ae --language  r       \
  --version 3 -- template udtf -- db dev--user nz -- level 4 \
  --mask DEBUG --  lastcall 'table final' --exe remote.R     \
  --sig "remote rae2(VARARGS)" --  return "TABLE(ANY)"      \
  --remote --rname remote_rae
```

If the previously invoked remote AE is still running, it should be possible to access it through the remote_rae and remote_rae2 UDXs:

```
SELECT * FROM TABLE WITH FINAL(remote_rae(0));
 CNT
-----
   4
(1 row)

SELECT * FROM TABLE WITH FINAL(remote_rae2(0));
 VALUE
-------
     5
(1 row)
```

CHAPTER 16

Advanced Remote Mode Examples

To create a remote mode program that can handle multiple simultaneous connections, each connection must be handled independently to avoid blocking. These examples demonstrate how each language handles this.

Note: R AEs do not support handling multiple simultaneous remote connections.

C Example (Multiple Simultaneous Connections)

There are two ways in C to achieve multiple simultaneous connections. The first is by creating a thread for each one using pthreads. The second is by forking off a new process to handle the connection, which is the method demonstrated in this example. This example is based on the shaper and sizer example, but with changes to the main.

In this example, nzaeRemprotAcceptEnvironment is called and forks twice to disassociate. SIGCHLD is ignored before the fork, and the parent waits for the first child to exit before unblocking, avoiding zombie processes. Only the parent can close the connection point and remote protocol handles. The child processes a single request.

This example uses the following file name:

```
sstring.c
```

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

#include "nzaeapis.h"

static int run(NZAE_HANDLE h);
static int runShaper(NZAESHP_HANDLE h);
```

```
int main(int argc, char * argv[])
{

    if (nzaeIsLocal())
    {
        NzaeApi result;
        char errorMessage[1050];
        if (nzaeLocprotGetApi(&result, NZAE_LDK_VERSION,
                              errorMessage, sizeof(errorMessage)))
        {
            fprintf(stderr, "%s\n", errorMessage);
            return -1;
        }
        if (result.apiType == NZAE_API_FUNCTION) {
            run(result.handle.function);
            nzaeClose(result.handle.function);
        }
        else {
            runShaper(result.handle.shaper);
            nzaeShpClose(result.handle.shaper);

        }
    }
    else {
        NZAECONPT_HANDLE hConpt = nzaeconptCreate();
        if (!hConpt)
        {
            fprintf(stderr, "error creating connection point\n");
            fflush(stderr);
            return -1;
        }
        const char * conPtName = nzaeRemprotGetRemoteName();
        if (!conPtName)
        {
            fprintf(stderr, "error getting connection point name\n");
            fflush(stderr);
            exit(-1);
        }
        if (nzaeconptSetName(hConpt, conPtName))
        {
            fprintf(stderr, "error setting connection point name\n");
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NzaeremprotInitialization args;
        memset(&args, 0, sizeof(args));
        args.ldkVersion = NZAE_LDK_VERSION;
        args.hConpt = hConpt;
        if (nzaeRemprotCreateListener(&args))
        {
            fprintf(stderr, "unable to create listener -
%s\n", args.errorMessage);
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NZAEENV_HANDLE hEnv;
        NZAEREMPROT_HANDLE hRemprot = args.handle;
        NzaeApi api;
        int i;
        for (i = 0; i < 5; i++)
        {
            if (nzaeRemprotAcceptEnvironment(hRemprot, &hEnv))
            {
                fprintf(stderr, "unable to accept API - %s\n", \
                     nzaeRemprotGetLastErrorText(hRemprot));
                fflush(stderr);
```

```
    nzaeconptClose(hConpt);
    nzaeRemprotClose(hRemprot);
    return -1;
}

struct sigaction newaction;
struct sigaction m_save;

memset(&newaction, 0, sizeof(newaction));
newaction.sa_handler = SIG_IGN;

sigaction(SIGCHLD, &newaction, &m_save);

int rcFork = fork();
if (rcFork == -1) {
    fprintf(stderr, "fork failed\n");
    fflush(stderr);
    nzaeconptClose(hConpt);
    nzaeRemprotClose(hRemprot);
}
else if (rcFork > 0) {
    int status;
    int ret = waitpid(rcFork, &status, 0);
    sigaction(SIGCHLD, &m_save, NULL);
    //parent
    nzaeenvClose(hEnv);
    continue;
}
   fork again to disassociate
rcFork = fork();
if (rcFork == -1)
    { exit(1);
}
else if (rcFork > 0) {
    nzaeenvClose(hEnv);
    exit(1);
}
int j;
   fix all signal handlers.
   This will remove any parent signal handlers
for (j=0;j < NSIG; j++) {
    signal(j, SIG_DFL);
}
setpgid(getpid(), 0);
setsid();
sigaction(SIGCHLD, &m_save, NULL);
printf("testcapi: accepted a remote request\n");
fflush(stdout);
NzaeApiTypes apiType = nzaeRemprotGetEnvironmentApiType(hEnv);
if (apiType == NZAE_API_FUNCTION) {
    NzaeInitialization arg;
    memset(&arg, 0, sizeof(arg));
    arg.hEnv = hEnv; arg.ldkVersion =
    NZAE_LDK_VERSION; if
    (nzaeInitialize(&arg))
    {
        fprintf(stderr, "initialization

        failed\n"); return -1;

    }
    run(arg.handle);
    nzaeClose(arg.handle);
}
else {
    NzaeShpInitialization arg;
    memset(&arg, 0, sizeof(arg));
    arg.hEnv = hEnv;
    arg.ldkVersion = NZAE_LDK_VERSION;
```

```
                if (nzaeShpInitialize(&arg))
                {
                    fprintf(stderr, "initialization failed\n");
                    return -1;

                }
                runShaper(arg.handle);
                nzaeShpClose(arg.handle);

            }

               Since you are forked, only run once
            exit(0);
        }
        nzaeRemprotClose(hRemprot);
        nzaeconptClose(hConpt);
    }
    return 0;
}

static int runShaper(NZAESHP_HANDLE h)
{
    char name[2];
    bool upper = true;
    NzaeShpMetadata meta;

    #define CHECK2(value) \
    { \
        NzaeRcCode rc = value; \
        if (rc) \
        { \
            const char * format = "%s in %s at %d"; \
            fprintf(stderr, format,  \
                nzaeShpGetLastErrorText(h), __FILE__, __LINE__); \
            \ nzaeShpUserError(h, format, \
                nzaeShpGetLastErrorText(h), __FILE__, __LINE__); \
            \ exit(-1); \
        } \
    }

    CHECK2(nzaeShpGetMetadata(h, &meta));
    if (!meta.oneOutputRowRestriction)
        CHECK2(nzaeShpSystemCatalogIsUpper(h, &upper));
    if (upper)
        name[0] = 'I';
    else
        name[0] = 'i';
    name[1] = 0;
    if (meta.inputTypes[0] == NZUDSUDX_FIXED || meta.inputTypes[0] ==
        NZUDSUDX_VARIABLE || meta.inputTypes[0] == NZUDSUDX_NATIONAL_FIXED
        || meta.inputTypes[0] == NZUDSUDX_NATIONAL_VARIABLE) {
        CHECK2(nzaeShpAddOutputColumnString(h, meta.inputTypes[0], name, \

             meta.inputSizes[0]));
    }
    else if (meta.inputTypes[0] == NZUDSUDX_NUMERIC128 ||
             meta.inputTypes[0] == NZUDSUDX_NUMERIC64 ||
             meta.inputTypes[0] == NZUDSUDX_NUMERIC32) {
        CHECK2(nzaeShpAddOutputColumnNumeric(h, meta.inputTypes[0], name, \
             meta.inputSizes[0], meta.inputScales[0]));
        }
        else {
            CHECK2(nzaeShpAddOutputColumn(h, meta.inputTypes[0], name));
        }

    CHECK2(nzaeShpUpdate(h));
```

```
}

static int run(NZAE_HANDLE h)
{
    NzaeMetadata metadata;
    if (nzaeGetMetadata(h, &metadata))
    {
        fprintf(stderr, "get metadata failed\n");
        return -1;
    }

    #define CHECK(value) \
    { \
        NzaeRcCode rc = value; \
        if (rc) \
        { \
            const char * format = "%s in %s at %d"; \
            fprintf(stderr, format,  \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            nzaeUserError(h, format, \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            exit(-1); \
        } \
    }

    for (;;)
    {
        int i;
        double result = 0;
        NzaeRcCode rc = nzaeGetNext(h);
        if (rc == NZAE_RC_END)
        {
            break;
        }

        NzudsData * input = NULL;
        CHECK(nzaeGetInputColumn(h, 0, &input));
        const char * opString;
        if (input->isNull)
        {
            nzaeSetOutputNull(h,0);
        }
        else {
            nzaeSetOutputColumn(h,0, input);
        }
        CHECK(nzaeOutputResult(h));
    }

    nzaeDone(h);
    return 0;
}
```

Compilation

Compile as follows:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language system --version 3 \
     --template compile sstring.c --exe sstring
```

Registration

For registration, use a remote variation to demonstrate the fork mode that only works in remote mode:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3 \
```

```
     --template udtf --exe sstring --sig "rem_sstring_c(VARCHAR(ANY))"
     \ --return "TABLE (val varchar(2000))" --remote --rname testcapi
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3 \
     --template udtf --exe sstring --sig "rem_sstring_launch(int8)" \
     --return "TABLE(aeresult varchar(255))"  --remote --rname testcapi \
     --launch
```

Running

To run:

```
SELECT * FROM TABLE WITH FINAL(rem_sstring_launch(0));
                                                  AERESULT
------------------------------------------------------------------------------
-------------------
 tran: 8278 session: 16019 DATA slc: 0 hardware: 0 machine: bdrosendev process:
20470 thread: 20470
(1 row)

SELECT * FROM TABLE WITH FINAL(rem_sstring_c('test'));
 VAL
------
 test
(1 row)
```

C++ Example (Multiple Simultaneous Connections)

In C++, there are two ways to handle multiple simultaneous connections independently. The first is by creating a thread for each one using pthreads. The second is by forking off a new process to handle the connection, which is the method demonstrated in this example. This example is based on the shaper and sizer example, but with changes to the main.

In this example, getApi is called with a TRUE to indicate fork mode, getting a pointer back instead of a reference. In remote mode there are two processes. The parent process gets a NULL back, increments the child counter and does nothing; it does this twice. In the child process after the fork, getApi returns the API with which to handle a shaper or sizer and then exits.

This example uses the following file name:

```
fork.cpp
```

Code

```
#include <nzaefactory.hpp>

using namespace nz::ae;

static int run(nz::ae::NzaeFunction *aeFunc);
static int doShaper(nz::ae::NzaeShaper *aeShaper);


int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
       The following line is only needed if a launcher is not
    used helper.setName("testcapi");
    int i = 0; int
    j=0; while (i
    < 1) {
```

```
        nz::ae::NzaeApi *api = helper.getApi(nz::ae::NzaeApi::ANY,
        true); if (helper.isRemote() && !api) {
                fork mode (parent)
            j++;
            if (j == 2)
                break;
            continue;
        }
        if (api->apiType == nz::ae::NzaeApi::FUNCTION) {
            run(api->aeFunction);
            i++;
        }
        else {
            doShaper(api->aeShaper);
        }
        if (!helper.isRemote())
            break;
        if (api)
            break;
    }

    return 0;
}

class MyHandler : public nz::ae::NzaeFunctionMessageHandler
{
public:
    void evaluate(NzaeFunction& api, NzaeRecord &input, NzaeRecord &result)
        { const NzaeMetadata& m = api.getMetadata();
        nz::ae::NzaeField &field = input.get(0);
        if (!field.isNull() ) {
            nz::ae::NzaeField &f = result.get(0);
            if (field.type() == NzaeDataTypes::NZUDSUDX_NUMERIC32 ||
                field.type() == NzaeDataTypes::NZUDSUDX_NUMERIC64 ||
                field.type() == NzaeDataTypes::NZUDSUDX_NUMERIC128)
            {
                NzaeNumericField &nf = (NzaeNumericField&)field;
                if (m.getInputSize(0) != nf.precision() ||
                    m.getInputScale(0) != nf.scale()) {
                    NzaeNumeric128Field* np = nf.toNumeric128(38, 5);
                    f.assign(*np);
                    delete np;
                }
                else
                    f.assign(field);
            }
            else {
                f.assign(field);
            }
        }

    }

};

class MyHandler2 : public nz::ae::NzaeShaperMessageHandler
{
public:
    void shaper(NzaeShaper& api){
        const NzaeMetadata& m =  api.getMetadata();
        char name[2];
        if (api.catalogIsUpper())
            name[0] = 'I';
        else
            name[0] = 'i';
        name[1] = 0;

        if (m.getInputType(0) == NzaeDataTypes::NZUDSUDX_FIXED ||
```

```
            m.getInputType(0) == NzaeDataTypes::NZUDSUDX_VARIABLE ||
            m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NATIONAL_FIXED ||
            m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NATIONAL_VARIABLE)
            { api.addOutputColumnString(m.getInputType(0), name,
m.getInputSize(0));
        }
        else if (m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NUMERIC128 ||
                 m.getInputType(0)  ==  NzaeDataTypes::NZUDSUDX_NUMERIC64  ||
                 m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NUMERIC32) {
            api.addOutputColumnNumeric(m.getInputType(0),
name, m.getInputSize(0),
                                                m.getInputScale(0));
        }
        else {
            api.addOutputColumn(m.getInputType(0), name);
        }
    }
};

static int doShaper(nz::ae::NzaeShaper *aeShaper)
{
    aeShaper->run(new MyHandler2());
    return 0;
}


static int run(nz::ae::NzaeFunction *aeFunc)
{
    aeFunc->run(new MyHandler());
    return 0;
}
```

Compilation

Compile as follows:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp --template compile \
     --exe forkae --compargs "-g -Wall" --linkargs "-g" fork.cpp --version 3
```

Registration

For registration, use a remote variation to demonstrate the fork mode that only works in remote mode:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "rem_forkae(VARARGS)" \
     --return "table(ANY)"  --language cpp --template udtf --exe forkae \
     --version 3 --remote --rname testcapi

$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "rem_fork_launch(int8)" \
     --return "TABLE(aeresult varchar(255))" --language cpp --template udtf \
     --exe forkae  --version 3 --remote --launch --rname testcapi
```

Running

To run:

```
SELECT * FROM TABLE WITH FINAL(rem_fork_launch(0));
                                AERESULT
------------------------------------------------------------------------
 tran: 7522 DATA slc: 0 hardware: 0 machine: spubox1 process: 3800 thread:
3800 (1 row)
SELECT * FROM TABLE WITH FINAL(rem_fork_launch(0));
                                AERESULT
------------------------------------------------------------------------
```

```
 tran: 7606 DATA slc: 0 hardware: 0 machine: spubox1 process: 6560 thread:
6560 (1 row)
SELECT * FROM TABLE WITH FINAL(rem_forkae('test'));
  I
------
 test
(1 row)
```

Java Language Advanced Remote Mode

To create a remote mode program that can handle multiple simultaneous connections, each connection must be handled independently to avoid blocking. In most of the examples, this is shown by using the runnable class to execute the AE and adding it to the executor pool. See Java Language Scalar Function (Remote Mode) for details.

Since this is already implemented in the remote mode "apply," the compilation, registration, and running are the same.

Fortran (Multiple Simultaneous Connections)

To create a remote mode program that can handle multiple simultaneous connections, each connection must be handled independently to avoid blocking. For other language adapters, this is performed manually either through threading or forking. For the Fortran adapter, the underlying layer automatically forks for new requests and nothing needs to be done by the user.

Python Example (Multiple Simultaneous Connections)

To create a remote mode program that can handle multiple simultaneous connections, each connection must be handled independently to avoid blocking. The Python AE framework performs this task. By default, the Python AE framework uses threads to handle each request. It is possible to override this functionality and have the framework either run single-threaded or use forking to handle multiple simultaneous requests.

When run() is called on the class, the class calls getRequestHandlingStyle(). By default it looks at the class variable, DEFAULT_REQUEST_HANDLING_STYLE. There are two ways to override the way an AE runs: the function getRequestHandlingStyle() can be overloaded when more intricate behavior is desired, or the class variable DEFAULT_REQUEST_HANDLING_STYLE can be overridden. The example shown uses the simple method to force the framework to fork and show that the internal handler is not being run in the same process as the running remote AE by returning the current and the parent process Ids.

This example uses the following file name:

```
fork.py
```

Code

```
import nzae
```

```
class ForkAe(nzae.Ae):

    DEFAULT_REQUEST_HANDLING_STYLE = nzae.Ae.REQUEST_HANDLING_STYLE__FORK

    def _getFunctionResult(self, row):
        return [parentProcessId, os.getpid()]

parentProcessId = os.getpid()

ForkAe.run()
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language python64
     \ --template deploy ./fork.py --version 3
```

Registration

Register both the forking AE and the launcher AE:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3
     \ --template udtf --exe fork.py --sig "remote_fork(int4)" \
     --return "table(parent_id int4, child_id int4)" --remote
     \ --rname forkRemote

$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3
     \ --template udtf --exe fork.py --sig "launch_remote_fork(int4)" \ --
     return "table(aeresult varchar(255))" --remote --launch \
     --rname forkRemote
```

Running

Run the remote AE server:

```
SELECT * FROM TABLE WITH FINAL(launch_remote_fork(0));
                              AERESULT
------------------------------------------------------------------------
 tran: 8592 session: 16283 DATA slc: 0 hardware: 0 machine: TT4-R045 process:
32082 thread: 32082
(1 row)
```

Now run the forking AE:

```
SELECT * FROM TABLE WITH FINAL(remote_fork(0));
 PARENT_ID | CHILD_ID
           +
----------32082 | ----------32154

(1 row)
```

The Python framework is performing the forking. If the DEFAULT_REQUEST_HANDLING_STYLE variable were not overridden, the numbers would match, as no new process was created to handle the request.

Although in these examples the /nz/export path is hard-coded, when creating AEs for release, this path should be retrieved through the command **nzenv** using the variable NZ_EXPORT_DIR.

Perl Example (Multiple Simultaneous Connections)

To create a remote mode program that can handle multiple simultaneous connections, each connection must be handled independently to avoid blocking. The Perl AE framework performs this task. By default, the Perl AE framework runs single-threaded while handling each request. It is possible to override this functionality and have the framework run using forking to handle multiple simultaneous requests.

When run() is called on the class, the class calls getRequestHandlingStyle(). By default it references the class variable, DEFAULT_REQUEST_HANDLING_STYLE. There are two ways to override AE execution. First, you can set the class variable DEFAULT_REQUEST_HANDLING_STYLE using the setDefaultRequestHandlingStyle(), as shown in the example below. Or, the function getRequestHandlingStyle() can be overloaded when more intricate behavior is desired. The example shown uses the simple method to force the framework to fork and show that the internal handler is not being run in the same process as the running remote AE by returning the current and the parent process IDs.

This example uses the following file name:

```
ForkAe.pm
```

Code

```
package ForkAe;
use strict;
use English;
use autodie;

use nzae::Ae;

our @ISA = qw(nzae::Ae);

my $ae = ForkAe->new();
my $parentProcessId = $PID;
$ae->setDefaultRequestHandlingStyle($ae-
>getRequestHandlingStyleFork()); $ae->run();
sub _getFunctionResult($@)
{
    my $self = shift;
    return ($parentProcessId, $PID);
}

1;
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language perl --version 3
    \ --template deploy ForkAe.pm
```

Registration

Register both the forking AE and the launcher AE:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3 \
```

```
    --template udtf --exe ForkAe.pm --sig "remote_fork(int4)"
    \ --return "table(parent_id int4, child_id int4)" --remote
    \ --rname forkRemote
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3
    \ --template udtf --exe ForkAe.pm --sig "launch_remote_fork(int4)"
    \ --return "table(aeresult varchar(255))" --remote --launch \ --
    rname forkRemote
```

Running

Run the remote AE server on the system database:

```
SELECT * FROM TABLE WITH
FINAL(launch_remote_fork(0)); AERESULT
------------------------------------------------------------------------
tran: 8592 session: 16283 DATA slc: 0 hardware: 0 machine: TT4-R045 process:
32082 thread: 32082
(1 row)
```

Now run the forking AE:

```
SELECT * FROM TABLE WITH FINAL(remote_fork(0));
PARENT_ID | CHILD_ID
----------+----------
32082 | 32154
(1 row)
```

The Perl framework is performing the forking. If the DEFAULT_REQUEST_HANDLING_STYLE variable were not overridden, the numbers would match, as no new process is created to handle the request.

The AE can now be terminated using the following SQL command:

```
SELECT aeresult FROM TABLE WITH FINAL(inza..nzaejobcontrol('stop',
0, 'forkRemote',false, NULL, NULL));
                              AERESULT
---------------------------------------------------------------------
 vnairsim 25227 (forkRemote dataslc:-1 sess:-1 trans:-1) AE stopped
(1 row)
```

Although in these examples the /nz/export path is hard-coded, when creating AEs for release, this path should be retrieved through the command **nzenv** using the NZ_EXPORT_DIR variable.

CHAPTER 17
Debugging Analytic Executables

There are a variety of techniques to debug AEs. The first technique described is called printf debugging, after the similarly named C library function. This technique outputs debugging information in string format from the program being debugged to standard output and error. You can collect this output in log files and place the log files from the host and all the SPUs into a single directory in the AE export directory tree. You can also attach debuggers and other diagnostic tools to AE processes running on the host. Another technique is to use NPS logging, which is the same facility used by UDXs.

The following sections examine the use of GDB, the test harness, Java debugging, and sample code for returning runtime system information.

Debugging “printf”-Style

To accomplish printf style debugging, temporarily place all log files from the host and the SPUs in a single shared directory. For local AEs, this can be done on a per function basis using the NZAE_LOG_DIR AE environment variable. For example, if the shared export directory is /nz/export/ae, the **register_ae** option is:

```
--environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"
```

In this example the log directory is not part of the installation. Create a log directory somewhere in the shared export tree. In a production environment, NZAE_LOG_DIR should not be used because of the potential for performance degradation as well as the fact that log file storage space is not automatically released on system restart.

In addition, you must set a logging level greater than zero in the registration. For example:

```
register_ae option --level 1
```

When the SQL function is called invoking the AE, the log files are created in the specified log directory.

Log File Names

Local Log File Names

For local AEs, three log files are created in the log directory per machine (host or SPUs)--a text file, stderr, and stdout.

By default, when using **register_ae** and NZAE_LOG_DIR, the main log file name has the format:

```
NZAE-<function-name>-<machine-name>-<transaction-id>-<undocumented stuff>.txt
```

There is also a standard output and standard error log file with the same name suffixed by _stdout.txt or _stderr.txt.

Examples

```
nzae-applyOperationV2Tf-hostname-2660-24771-10167-17646.txt nzae-
applyOperationV2Tf-hostname-2660-24771-10167-17646.txt_stderr.txt nzae-
applyOperationV2Tf-hostname-2660-24771-10167-17646.txt_stdout.txt
```

When the programming language's specific technique to print to standard output or standard error is used, the output appears in these log files. You should "flush" the stderr and stdout buffers or the output may not appear until the AE completes.

Remote Log File Names

Log file naming differs with remote AEs. When a remote AE is started, it has its own standard output and standard error log files that all function requests share. Therefore, the NZAE_LOG_DIR setting must be used in the launch registration.

Examples

```
nzae-apply_launch_v1-hostname-2662-24771-15233-launch-noexecae-
17646.txt_parent.txt
nzae-apply_launch_v1-hostname-2662-24771-15235-launch-17646.txt
nzae-apply_launch_v1-hostname-2662-24771-15235-launch-17646.txt_stderr.txt
nzae-apply_launch_v1-hostname-2662-24771-15235-launch-17646.txt_stdout.txt
```

Each request has its own main log file but its standard output and error goes to the remote AE log files. If the AE does not reach the printf statements, check the main log file for AE error messages.

Using Debuggers and Other Tools

The following sections describe a general, non-language specific introduction to using a debugger or other diagnostic tool on an AE process.

Running on the Host

As a general rule, you should perform debugging on the host. Therefore, the AEs and requests must be running on the host. There are two techniques for getting requests to run on the host:

For scalar, table, and aggregate functions, the SQL function receives all literal arguments.

```
SELECT applyOperationV1Sf('+', 1, 2, 3);
SELECT * FROM TABLE WITH FINAL(applyOperationV1Tf('+', 1, 2, 3));
```

For a table function only, register the AE with the option **--noparallel**.

This option causes the AE to run on the host so that it can receive data arguments from normal tables as well as literals. The option **--noparallel** cannot be used with scalar functions. However, the table functions and scalar functions use the same AE API, so a scalar function AE can easily be called from an SQL table function for debugging purposes (because a table function has a superset of scalar function capabilities). To do so, register the AE written as a scalar function as a table function. Most table function AEs cannot be registered as scalar functions unless they return exactly one output row for every input row and have exactly one column in a row.

Two Debugger Methods

There are two general approaches when using a diagnostic/debugger tool on an AE process.

Attach to an already running process.

Have the AE runtime run the diagnostic/debugger tool that then loads the AE.

With the attach approach, you must supply the AE process ID, and the process must wait for the tool to be attached.

Local AE and Spin Files

For a local AE, a spin file mechanism is available. When activated, the AE runtime looks for the file specified by the AE ENV variable. If the file exists, the AE runtime sleeps for a while and then checks again for file existence. The AE runtime also prints information about the process to the standard error log file. Looping continues until the file is renamed or deleted.

Example

```
spinning in process id=15796, thread id=15797 on file /tmp/spin_ae.on
```

Once the tool is attached, you delete or rename the file. Then the AE resumes normal execution.

Example

```
mv /tmp/spin_ae.on /tmp/spin_ae.off
```

The spin file mechanism is turned on using an AE environment variable.

```
--environment "'NZAE_HOST_ONLY_NZAE_SPIN_FILE_NAME'='/tmp/spin_ae.on'"
```

Note how the NZAE_HOST_ONLY_ prefix is used to prevent accidental use of the spin file mechanism on the SPUs.

Remote AE

Because a remote AE is usually a long running process, attaching a tool like GDB is usually much easier. Use the nzaejobcontrol function to ping the remote AE for get process ID and other information needed by the debugger.

Using AE to Load a Diagnostic or Debugger Tool

Using the AE runtime to load a diagnostic/debugger tool is more complex than manually attaching via GDB because AEs do not, by default, have a terminal, so you must provide one. To create a terminal:

Open a new Linux terminal on the host.

Run the Linux **tty** command to get its name:

```
  tty
/dev/pts/
4
```

Run a long sleep on the terminal (like sleep 1000000) to keep the shell from interfering with AE and diagnostic tool IO.

Register the AE to use this terminal using the NZAE_TERMINAL AE environment variable:

```
--environment "'NZAE_TERMINAL'='/dev/pts/4'"
```

You must modify the registration to both run the tool and to give the tool the proper parameters to run the AE. Three system AE environment variables are used to accomplish this task:

```
NZAE_HOST_ONLY_NZAE_EXECUTABLE_PATH=<path of tool>
NZAE_NUMBER_PARAMETERS=<integer: number of parameters>
NZAE_PARAMETER1=<tool specific>
NZAE_PARAMETER2=<tool specific>
```

Note that in the above pseudo-code, the number of parameters is tool- and application-specific.

AE functionality has a built-in mechanism to run GDB (see Working with GDB for more information). This example is a model for using GDB and other debuggers and tools. The example uses terminal /dev/pts/7, which is running sleep 1000000.

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language system --version 3
--template compile \
--compargs "-g -Wall" --linkargs "-g" --exe

testcapi /home/nz/src/cpp/ae/testcae/testcapi.c
```

Note in the compilation above that **--compargs** and **--linkargs** are used to set compile and link to produce debugging symbols.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version
3 --template udtf
     --sig "testcapi_debug(int, double, varchar(30))" \
     --return "table(a int, b int, c double, d double, e varchar(255),
     \ f varchar(255))" \
     --exe testcapi --noparallel --level 1 \
     --environment "'NZAE_TERMINAL'='/dev/pts/7'" \
     --environment
"'NZAE_HOST_ONLY_NZAE_EXECUTABLE_PATH'='/nz/kit/sbin/gcc/bin/gdb'"
     \ --environment "'NZAE_NUMBER_PARAMETERS'='2'" \ --environment
     "'NZAE_PARAMETER1'='--args'" \
     --environment \
"'NZAE_PARAMETER2'='/nz/export/ae/applications/mydb/myusr/host/testcapi'"
```

In this example, the executable run by the AE runtime is the debugger itself. The AE application is specified as an argument to the debugger. The **--args** option is a GDB option, not an AE option.

Working with Log Files

As background, the "Debugging printf Style" section discussed the use of AE environment variable NZAE_LOG_DIR. When this variable is not set, the log files are created in the NPS temporary directories on the host and SPUs (which are different than the Linux /tmp directory). Because the SPUs are not directly accessible, utilities are provided to manage log files.

Log files can be used to help troubleshoot AEs. As mentioned in General Common Variables, NZAE_DEBUG (the register_ae **--level** option) and NZAE_LOG_IDENTIFIER can be used to control the creation of log files, which reside in the temporary directory on either the host or the SPU. These log files can be listed, viewed, or deleted using the listlogs, viewlog, and cleanlogs functions provided as part of the Admin Utilities cartridge. When using **register_ae** NZAE_LOG_IDENTIFIER, the default is to the function name. In the descriptions below, the value of NZAE_LOG_IDENTIFIER is shown as IDENTIFIER.

Log Files Types

The following describes each type of log file. In local mode, and some remote mode logs (as specified in the log description), the file name is in the form:

```
nzae-IDENTIFIER-<transaction ID>-<parent process ID>-<child process ID>-
<session ID>.<type>
```

In the remaining remote mode logs, the file name is in the form:

```
nzae-IDENTIFIER-<transaction ID>-<parent process ID>-<child process ID>-
remote-<pointer to the AE>-<session ID>.<type>
```

Local Mode: Child Log

The file name for a child log in local mode is in the form, for example:

```
nzae-IDENTIFIER-0-11510-11512-17070.txt
```

The contents of this file include:

A listing of the environment variable active when the child is created.

Information about the process, including process ID, thread ID, binary, shared libraries and command line. For example:

```
process id=16442, thread id=16442
/u01/home/nz/workspaces/dev-ae/main/src/ae/build/cpp_ae/host/testmax \
     75682 Mon Apr 26 11:19:04 2010
/u01/home/nz/workspaces/usr-bdrosen-
tfunc/main/simdata/base/200103/library/209623/host/libnzaecpp3.so 1619222 \
     Mon Apr 26 11:17:02 2010
/u01/home/nz/workspaces/usr-bdrosen-
tfunc/main/simdata/base/200103/library/200652/host/libnzaechild.so 501191 \
     Mon Apr 26 11:17:02 2010
/lib/libpthread.so.0                          125612 Mon Jan 5 19:53:27 2009
/usr/lib/libstdc++.so.6                       936908 Thu Sep 18 14:06:28 2008
/lib/libm.so.6                                208352 Mon Jan 5 19:53:27 2009
/lib/libgcc_s.so.1                             46476 Thu Sep 18 14:06:28 2008
/lib/libc.so.6                               1606808 Mon Jan 5 19:53:26 2009
/lib/ld-linux.so.2                            125736 Mon Jan 5 19:53:26 2009
    Process Command Line
/u01/home/nz/workspaces/dev-
ae/main/src/ae/languages/cpp/../../build/cpp_ae//host/testmax parm1
```

```
    End of Command Line
```

A log of startup of the AE from the child's point of view. For example:

```
==INFO AE initialization requested
==INFO AE Build = 5
==INFO --- Shared Library Info ---

Number of shared libraries = 4

LIBNZAEADAPTERS  libnzaeadapters.so       3439259 Mon Apr 26 11:17:02 2010
LIBNZAECHILD     libnzaechild.so          501191 Mon Apr  26 11:17:02 2010
LIBNZAECPP2      libnzaecpp3.so           1619222 Mon Apr 26 11:17:02 2010
LIBNZAEPARENT    libnzaeparent.so         920241 Mon Apr  26 11:17:01 2010
==INFO child _nzaerepchild version: 7
==INFO child in nzrep pass thru mode
```

A log of the various AE messages sent between the parent and the child. For example:

```
before nzrepSend #2 NZREP_REQUEST (6) length=0
nzaemessage.c:13 after nzrepSend #2 nzaemessage.c:18
before nzrepReceive #1 nzaemessage.c:20
after nzrepReceive #1 NZREP_DATA (1) length=80 nzaemessage.c:25
```

Local Mode: Parent Log

The file name for a parent log in local mode is in the form, for example:

```
nzae-IDENTIFIER-0-11510-11512-17070.txt_parent.txt
```

This file contains a log of the various AE messages sent between the parent and the child. Output is basically the same as from the child log, but may have slight differences.

Local Mode: Child Stderr

The file name for a child standard error log is in the form, for example:

```
nzae-IDENTIFIER-0-13894-13975-16934.txt_stderr.txt
```

This contains the standard error output from the child.

Local Mode: Child Stdout

The file name for a child standard output log is in the form, for example:

```
nzae-IDENTIFIER-0-13894-13975-16934.txt_stdout.txt
```

This file contains the standard output from the child.

Remote Mode: Child Request Log

The file name for a remote mode child request log is in the form, for example:

```
nzae-IDENTIFIER-0-942-0-remote-179982496-16184.txt
```

Because this is remote AE mode, the child process ID is always 0. Because the child process ID cannot be used to distinguish between multiple AEs in the same parent process, you must use the pointer to make it unique.

The contents of this file are similar to the local mode child log, except it is missing the process information.

Remote Mode: Parent Request Log

The file name is in the form:

```
nzae-IDENTIFIER-0-942-0-remote-179982496-16184.txt_parent.txt
```

Because this is remote AE mode, the child process ID is always 0 and because you can't use the child process id to distinguish between multiple AEs in the same parent process; you must use the pointer to make it unique.

The contents of this file are similar to or the same as the local mode parent log.

Remote Launch Mode: Parent Log

The file name is in the form described in the local mode file format, with the addition of the launch keyword:

```
nzae-IDENTIFIER-6060-943-949-launch-16184.txt_parent.txt
```

The contents of this file contain a few launch AE messages sent between the parent and this child.

Remote Launch Mode: Transient Child Log

The file name is in the form described in the local mode file format, with the addition of the launch keyword:

```
nzae-IDENTIFIER-6060-943-949-launch-noexecae-16184.txt_parent.txt
```

The contents of this file contain a few AE messages sent between the parent and the transient child. The transient child is created for a brief time. It is between the first fork and the second fork when launching a remote AE. Two forks are used to disassociate the remote AE from the parent process that launched it and put it into a different process group.

Remote Launch Mode: Child Log

The file name is in the form described in the local mode file format, with the addition of the launch keyword::

```
nzae-IDENTIFIER-6060-943-952-launch-16184.txt
```

The contents of this file contain the environment and library information.

Remote Launch Mode: Child Stderr

The file name is in the form described in the local mode file format, with the addition of the launch keyword:

```
nzae-IDENTIFIER-6060-943-952-launch-16184.txt_stderr.txt
```

The contents of this file contain the standard error output from the remote AE throughout its lifetime.

Remote Launch Mode: Child Stdout

The file name is in the form described in the local mode file format, with the addition of the launch keyword:

```
nzae-IDENTIFIER-6060-943-952-launch-16184.txt_stdout.txt
```

The contents of this file contain the standard output from the remote AE throughout its lifetime.

Admin Utilities

The following are some examples of admin utilities used for debugging. For all of the utilities, use the dataslice ID parameter to control whether the function is run on the host or the SPU. To run it on the

SPU, run a command like the following:

```
SELECT * FROM _v_dual_dslice, TABLE WITH FINAL(listlogs(-1,dsid));
```

This command correlates the table function with the _v_dual_dslice table that exists on the SPU, causing the table function to run on the SPU and the host (unlike the previous example, which ran only on the host).

Note that the value of NZAE_DEBUG controls what is seen in the log files (0 is nothing, 1 is some information, 2 is child information, 3 is child and parent information).

Listlogs

This function takes two integers--a session ID and a dataslice ID. The session ID can be either a valid session ID or the value "-1" to indicate all sessions. The dataslice ID is ignored by the listlogs function, but is used to control the locus of execution (host or SPU, as well as dataslice). For example:

```
SELECT * FROM TABLE WITH FINAL(listlogs(-1,0));
```

The function returns the dataslice ID (DSID), the type of the log file (TYPE), the modified timestamp (MODIFIED), and the log file name (FILENAME). Table 8 shows sample output, formatted for easier reading.

Table 8: Sample Listlogs Function Output

DSID	TYPE	MODIFIED	FILENAME
0	Remote Mode - Child Request Log	04/26/10 03:23 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-0-remote-171176744-16934.txt
0	Remote Mode - Parent Request Log	04/26/10 03:23 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-0-remote-171176744-16934.txt_parent.txt
0	Remote Mode - Child Request Log	04/26/10 03:23 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-0-remote-171178024-16934.txt
0	Remote Mode - Parent Request Log	04/26/10 03:23 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-0-remote-171178024-16934.txt_parent.txt
0	Remote Mode - Child Request Log	04/26/10 03:23 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-0-remote-171188432-16934.txt
0	Remote Mode - Parent Request Log	04/26/10 03:23 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-0-remote-171188432-16934.txt_parent.txt
0	Remote Mode - Child Request Log	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-0-remote-171209264-16934.txt
0	Remote Mode - Parent Request Log	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-0-remote-171209264-16934.txt_parent.txt
0	Remote Mode - Child Request Log	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-0-remote-171232280-16934.txt

Table 8: Sample Listlogs Function Output

DSID	TYPE	MODIFIED	FILENAME
0	Remote Mode - Parent Request Log	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-0-remote-171232280-16934.txt_parent.txt
0	Local Mode - Child Log	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-13975-16934.txt
0	Local Mode - Parent Log	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-13975-16934.txt_parent.txt
0	Local Mode - Child Stderr	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-13975-16934.txt_stderr.txt
0	Local Mode - Child Stdout	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-13975-16934.txt_stdout.txt
0	Local Mode - Child Log	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14031-16934.txt
0	Local Mode - Parent Log	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14031-16934.txt_parent.txt
0	Local Mode - Child Stderr	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14031-16934.txt_stderr.txt
0	Local Mode - Child Stdout	04/26/10 03:22 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14031-16934.txt_stdout.txt
0	Local Mode - Child Log	04/26/10 03:23 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt
0	Local Mode - Parent Log	04/26/10 03:23 PM	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt
(20 rows)			

Viewlog

This function takes two arguments--the file name and the dataslice ID.

```
SELECT * FROM TABLE WITH FINAL(viewlog('/u01/home/nz/workspaces/usr-bdrosen-
tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt',0));
```

The function returns the dataslice ID (DSID), the log file name (FILENAME), the slice of the file (SLICE), and the line of text (DATA). Table 9 shows the output, formatted for easier reading.

Table 9: Sample Viewlog Function Output

DSID	FILENAME	SLICE	DATA
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	1	before nzrepReceive #1 nzaeaggcontroller.cpp: 247

Table 9: Sample Viewlog Function Output

DSID	FILENAME	SLICE	DATA
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt parent.txt	2	after nzrepReceive #1 NZREP PING (5) length=0 nzaeaggcontroller.cpp: 252
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	3	before nzrepReceive #2 nzaeaggcontroller.cpp: 247
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	4	after nzrepReceive #2 NZREP REQUEST (6) length=0 nzaeaggcontroller.cpp: 252
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt parent.txt	5	before nzrepSend #1 NZREP_METADATA (8) length=4088 nzaeaggcontroller.cpp: 403
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	6	after nzrepSend #1 nzaeaggcontroller.cpp: 408
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	7	before nzrepReceive #3 nzaeaggcontroller.cpp: 247
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	8	after nzrepReceive #3 NZREP_REQUEST (6) length=0 nzaeaggcontroller.cpp: 252
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt parent.txt	9	before nzrepSend #2 NZREP_METADATA (8) length=4088 nzaeaggcontroller.cpp: 403
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	10	after nzrepSend #2 nzaeaggcontroller.cpp: 408
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	11	before nzrepReceive #4 nzaeaggcontroller.cpp: 247
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	12	after nzrepReceive #4 NZREP REQUEST (6) length=0 nzaeaggcontroller.cpp: 252
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt parent.txt	13	before nzrepSend #3 NZREP_DATA (1) length=4088 nzaeaggcontroller.cpp: 91
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	14	after nzrepSend #3 nzaeaggcontroller.cpp: 96

Table 9: Sample Viewlog Function Output

DSID	FILENAME	SLICE	DATA
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	15	before nzrepReceive #5 nzaeaggcontroller.cpp: 308
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	16	after nzrepReceive #5 NZREP_DATA (1) length=4088 nzaeaggcontroller.cpp: 313
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	17	before nzrepReceive #6 nzaeaggcontroller.cpp: 247
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	18	after nzrepReceive #6 NZREP_REQUEST (6) length=0 nzaeaggcontroller.cpp: 252
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt parent.txt	19	before nzrepSend #4 NZREP METADATA (8) length=4088 nzaeaggcontroller.cpp: 403
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	20	after nzrepSend #4 nzaeaggcontroller.cpp: 408
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	21	before nzrepReceive #7 nzaeaggcontroller.cpp: 247
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	22	after nzrepReceive #7 NZREP_REQUEST (6) length=0 nzaeaggcontroller.cpp: 252
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt parent.txt	23	before nzrepSend #5 NZREP DATA (1) length=4088 nzaeaggcontroller.cpp: 213
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	24	after nzrepSend #5 nzaeaggcontroller.cpp: 218
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	25	before nzrepReceive #8 nzaeaggcontroller.cpp: 308
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	26	after nzrepReceive #8 NZREP_DATA (1) length=4088 nzaeaggcontroller.cpp: 313
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	27	before nzrepReceive #9 nzaeaggcontroller.cpp: 247
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt parent.txt	28	after nzrepReceive #9 NZREP_REQUEST (6) length=0 nzaeaggcontroller.cpp:

Table 9: Sample Viewlog Function Output

DSID	FILENAME	SLICE	DATA
			252
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	29	before nzrepSend #6 NZREP_END (3) length=0 nzaeaggcontroller.cpp: 232
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	30	after nzrepSend #6 nzaeaggcontroller.cpp: 237
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	31	before nzrepClose nzaebasecontroller.cpp :90
0	/u01/home/nz/workspaces/usr-bdrosen-tfunc/main/tmp/nzae-IDENTIFIER-0-13894-14302-16934.txt_parent.txt	32	after nzrepClose nzaebasecontroller.cpp :92

(36 rows)

Cleanlogs

This function takes two arguments--the session ID and the dataslice ID. It returns the dataslice ID and text detailing how many files were deleted (with a limit of 20 rows of output).

```
SELECT * FROM TABLE WITH FINAL(cleanlogs(16934,0)) LIMIT 20;

DSID                        TXT
0                           Deleted 229 log files
(1 row)
```

Understanding UDX Logging

Like UDXs, AEs have access to the integrated NPS log facility. More documentation about UDX logging can be found in the *Netezza User-Defined Functions Developer's Guide*.

Enable logging for the entire system using the NPS system nzudxdbg command utility; enable logging for a specific AE through the register_ae –**mask** option. For example

```
--mask DEBUG
--mask TRACE
```

The **--mask** option can be specified twice to include both DEBUG and TRACE output in the log file. The application code can invoke the log function with either DEBUG or TRACE, and if logging is turned on for the system and the matching log facility is enabled during AE registration (DEBUG or TRACE), the log message is delivered.

This is similar to printf style logging. The following is an example using a C++ AE:

Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <nzaefactory.hpp>
using namespace nz::ae;
```

```
static int run(nz::ae::NzaeFunction *aeFunc);
static int doShaper(nz::ae::NzaeShaper *aeShaper);

int main(int argc, char * argv[])
{
	NzaeApiGenerator helper;
	while (true) {
		nz::ae::NzaeApi *api = helper.getApi(nz::ae::NzaeApi::ANY,
			true);
		if (helper.isRemote() && !api) {
			continue;
		}
		if (api->apiType == nz::ae::NzaeApi::FUNCTION) { api-
			>aeFunction->log(nz::ae::NzaeFunction::LOG_TRACE,
				"Got Function");
			run(api->aeFunction);
		}
		else {
			api->aeShaper->log(nz::ae::NzaeShaper::LOG_TRACE,
				"Got Shaper");
			doShaper(api->aeShaper);
		}
		if (!helper.isRemote())
			break;
		if (api)
			break;
	}
	return 0;
}

class MyHandler : public nz::ae::NzaeFunctionMessageHandler
{
public:
	void evaluate(NzaeFunction& api, NzaeRecord &input,
		NzaeRecord &result) {
			const NzaeMetadata& m =  api.getMetadata();
			nz::ae::NzaeField &field = input.get(0);
			if (!field.isNull() ) {
				nz::ae::NzaeField &f = result.get(0);
				f.assign(field);
			}
	}
};

class MyHandler2 : public nz::ae::NzaeShaperMessageHandler
{
public:
	void shaper(NzaeShaper& api){
		const NzaeMetadata& m =  api.getMetadata();
		char name[2];
		if (api.catalogIsUpper())
			name[0] = 'I';
		else
			name[0] = 'i';
		name[1] = 0;
		if (m.getInputType(0) == NzaeDataTypes::NZUDSUDX_FIXED ||
			m.getInputType(0) == NzaeDataTypes::NZUDSUDX_VARIABLE ||
			m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NATIONAL_FIXED
||
			m.getInputType(0) ==
NzaeDataTypes::NZUDSUDX_NATIONAL_VARIABLE) {
				api.addOutputColumnString(m.getInputType(0),
					name, m.getInputSize(0));
		}
		else if (m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NUMERIC128
||
			m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NUMERIC64 ||
			m.getInputType(0) == NzaeDataTypes::NZUDSUDX_NUMERIC32) {
```

```
                                        api.addOutputColumnNumeric(m.getInputType(0),
                                                name, m.getInputSize(0),
                                                m.getInputScale(0));
                        }
                        else {
                                api.addOutputColumn(m.getInputType(0), name);
                        }
                }
};

static int doShaper(nz::ae::NzaeShaper *aeShaper)
{
    aeShaper->run(new MyHandler2());
    return 0;
}
static int run(nz::ae::NzaeFunction *aeFunc)
{
    aeFunc->run(new MyHandler());
    return 0;
}
```

Note the following lines:

```
api->aeFunction->log(nz::ae::NzaeFunction::LOG_TRACE, "Got Function");
api->aeShaper->log(nz::ae::NzaeShaper::LOG_TRACE, "Got Shaper");
```

Compilation

Compile the AE:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp --template compile
     \ --version 3 TestIdentity.cpp --exe testidentity
```

Registration

Register the AE with TRACE enabled:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language cpp --version 3
     \ --template udf --sig "logger(varchar(any))" \
     --return "varchar(any)" --exe testidentity --mask TRACE
```

Enable Debugging

You enable debugging using nzudxdbg

```
$ nzudxdbg --file

Processing 1 spus
.
done
Processing host
done
```

Running

To run the logger, first create a simple table:

```
create table s (s varchar(100));
INZA(ADMIN)=> insert into s values('20101401');
INSERT 0 1
INZA(ADMIN)=> insert into s values('20101303');
INSERT 0 1
```

Run the logger using the following SQL:

```
SELECT logger(s) FROM s;
```

```
LOGGER
----------
20101401
20101303
(2 rows)
```

You then get the following in pg.log:

```
2010-04-27 06:38:44.848757 EDT [27656] DEBUG: Got Shaper
```

And the following in sysmgr.log:

```
2010-04-27 06:38:45.168278 EDT (event1137.1[1]) (dsid=1)[d,udx ] Got Function
```

The above log messages occur because the shaper runs in postgres and the function runs on the SPU.

Working with GDB

C or C++ AEs can be debugged using GDB. As noted above, debugging should be performed on the host and not the SPU. The UDX test harness can be run on the host. For more details, refer to Using a Test Harness for Troubleshooting.

When compiling and linking a program to be used with GDB, debugging symbols (**-g**) must be included. There are three different ways to use GDB--AE enabled, spin, and attach--each of which is described in the following sections.

AE Enabled (Local AE)

You can use GDB when in local AE mode. In this case, add the GDB path to the AE by specifying NZAE_GDB_PATH when registering the AE, as well as by specifying the terminal. For example:

```
--environment "'NZAE_TERMINAL'='/dev/pts/4'" -environment \
      "'NZAE_GDB_PATH'='/nz/kit/sbin/gcc/bin/gdb'"
```

You can set up debugging of the local AE by using the applyopcpp case (see code in the section C++ Language Scalar Function). Note that the **--sig** in register_ae has changed. Therefore, the SQL invocation should use **apply** instead of **applyop_cpp** to mirror the change to **--sig**. In addition, the **--exe** in compile_ae and register_ae have also changed.

1. Add the GDB path to the AE:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp --template compile
\ --exe testapply --compargs "-g -Wall" --linkargs "-g" applyopcpp.cpp \ --
version 3
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language cpp --version 3
     \ --template udf --sig "apply(varargs)" --return "double" \
     --exe testapply --environment "'NZAE_TERMINAL'='/dev/pts/4'"
     \ --environment "'NZAE_GDB_PATH'='/nz/kit/sbin/gcc/bin/gdb'"
```

   Add a stub for calling the function logic by doing the following step:
   In the window that is associated with the terminal, run **sleep 1000000** on the terminal to keep the shell running in that terminal from interfering with standard I/O.

   When the SQL is run, it hangs because the debugger is active and you get the GDB prompt in the terminal window. You can then run normal GDB commands and nzsql waits until you exit

from GDB. After you exit from GDB, kill the sleep to get back to the shell prompt.

```
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
(gdb) bt
No stack.
(gdb) b main

Breakpoint 1 at 0x8049278: file applyopcpp.cpp, line 19.
(gdb) c
The program is not being run.
(gdb) r
Starting program: /nz/export/ae/applications/dev/admin/host/testapply
[Thread debugging using libthread_db enabled]
[New Thread 0xf7f0bad0 (LWP 28566)]
Breakpoint 1, main () at applyopcpp.cpp:19
19 NzaeApiGenerator helper;
(gdb) c
Continuing.
[New Thread 0xf7f0ab90 (LWP 28571)]
in evaluate
[Thread 0xf7f0ab90 (LWP 28571) exited]

Program exited normally.
(gdb) q
```

Spin (Local AE)

Use spin in local AE mode. To do so, register the name of a file with the environment variable NZAE_SPIN_FILE_NAME.

If the file exists, the AE waits until one of the following conditions is met:

You delete the file.

You attach the file to the AE process by using GDB; then you set the stopLoopFlag variable to 1. To register the file name, do the following steps:

Register the file name as follows (using the applyopcpp case):

```
--environment "'NZAE_SPIN_FILE_NAME'='/tmp/spin_ae'"
```

The complete command looks like:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language cpp --version 3
     \ --db dev --template udf --sig "apply(varargs)" --return "double"
     \ --exe testapply --environment "'NZAE_TERMINAL'='/dev/pts/4'" \ -
     -environment "'NZAE_SPIN_FILE_NAME'='/tmp/spin_ae'"
```

Create the spin file:

```
touch /tmp/spin_ae
```

When you run the query, you need to find the process to attach to. Because you specified NZAE_TERMINAL when you registered, you see the following in the terminal window:

```
spinning in process id=28618, thread id=28618 on file /tmp/spin_ae
```

You can then attach to it through GDB:

```
gdb /nz/export/ae/applications/dev/admin/host/testapply 28618
```

```
GNU gdb Fedora (6.8-27.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
Attaching to program:
/nz/export/ae/applications/dev/admin/host/testapply, process 28618
Reading symbols from
/nz/data/base/200103/library/209623/host/libnzaecpp3.so...done.
Loaded symbols for /nz/data/base/200103/library/209623/host/libnzaecpp3.so
Reading symbols from
/nz/data/base/200103/library/200652/host/libnzaechild.so...done.
Loaded symbols for /nz/data/base/200103/library/200652/host/libnzaechild.so
Reading symbols from /lib/libpthread.so.0...done.
[Thread debugging using libthread_db enabled]
[New Thread 0xf7e3bad0 (LWP 28618)]
Loaded symbols for /lib/libpthread.so.0
Reading symbols from /usr/lib/libstdc++.so.6...done.
Loaded symbols for /usr/lib/libstdc++.so.6
Reading symbols from /lib/libm.so.6...done.
Loaded symbols for /lib/libm.so.6
Reading symbols from /lib/libgcc_s.so.1...done.
Loaded symbols for /lib/libgcc_s.so.1
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xffffe410 in __kernel_vsyscall ()
(gdb) bt
#0 0xffffe410 in __kernel_vsyscall ()
#1 0x0088b290 in __nanosleep_nocancel () from /lib/libc.so.6 #2
0x0088b0df in sleep () from /lib/libc.so.6
#3 0xf7e7d1ca in _nzaeCheckForSpinFile (spinFileName=0xffd0a7ff
     \ "/tmp/spin_ae") at nzaediagnostics.c:43
#4 0xf7e75e16 in _nzaeInitializeInternal (arg=0xffd081d0, internalUse=false)
     \ at nzae.c:31
#5 0xf7e75da1 in nzaeInitialize (arg=0xffd081d0) at nzae.c:19
#6 0xf7e80c97 in nzaeLocprotGetApi (result=0xffd08a44, ldkVersion=2,
     \ errorMessage=0xffd08627 "", errorMessageSize=1050)
at nzaeremoteprotocol.c:478
#7 0xf7edd7e1 in nz::ae::NzaeRemoteProtocol::getLocalApi () at
     \ src/nzaeremoteprotocol.cpp:81
#8 0xf7eb4c0f in nz::ae::NzaeFactory::getLocalApi (this=0xf7f09444) at
     \ src/nzaefactory.cpp:67
#9 0x0804a2f1 in nz::ae::NzaeApiGenerator::getApi (this=0xffd08ba0,
     \ type=nz::ae::NzaeApi::FUNCTION, fork=false) \
     at /nz/export/ae/adapters/cpp/3/sys/include/nzaeapigenerator.hpp:76 #10
0x0804ae18 in nz::ae::NzaeApiGenerator::getApi (this=0xffd08ba0, \
     type=nz::ae::NzaeApi::FUNCTION) \
     at /nz/export/ae/adapters/cpp/3/sys/include/nzaeapigenerator.hpp:63 #11
0x080492b2 in main () at applyopcpp.cpp:22
(gdb) n
Single stepping until exit from function
__kernel_vsyscall, which has no line number information.
0x0088b290 in __nanosleep_nocancel () from
/lib/libc.so.6 (gdb)
Single stepping until exit from function
__nanosleep_nocancel, which has no line number information.
0x0088b0df in sleep () from /lib/libc.so.6
(gdb)
Single stepping until exit from function sleep,
which has no line number information.
_nzaeCheckForSpinFile (spinFileName=0xffd0a7ff "/tmp/spin_ae")
at nzaediagnostics.c:30
30 while (!stopLoopFlag)
Current language: auto; currently c
```

```
(gdb) p stopLoopFlag = 1
$1=1
(gdb) b applyopcpp.cpp:24

Breakpoint 1 at 0x80492c3: file applyopcpp.cpp, line 24.
(gdb) c
Continuing.
[New Thread 0xf7e3ab90 (LWP 28647)]
Breakpoint 1, main () at applyopcpp.cpp:24
24 if (!helper.isRemote())
Current language: auto; currently c++
(gdb) c
Continuing.
[Thread 0xf7e3ab90 (LWP 28647) exited]
Program exited normally.
(gdb) q
```

Remember to remove the spin file afterward or the next time it is run it will hang as well.

Attach (Remote AE)

Attaching works for remote AE mode. In this case you register the remote AE and the remote AE launcher:

Register the remote AE:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language cpp --version 3 \
     --db dev --template udf --sig "apply(varargs)" --return "double" \
     --exe testapply --rname testcapi --remote
```

Register the remote AE launcher:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae -- language cpp --version 3 \
     --db dev --template udtf --sig "launch apply(int8)" \
     --return "TABLE(aeresult varchar(255))" --  exe testapply --rname testcapi \
     --remote –launch
```

3. Launch the remote AE:

```
SELECT * FROM TABLE WITH FINAL(launch_apply(0));
                                AERESULT
------------------------------------------------------------------------
tran: 16310 session: 17244 DATA slc: 0 hardware: 0 machine: bdrosendev process:
31041 thread: 31041
(1 row)
```

Then in another window you attach to it:

```
[nz@bdrosendev src]$ gdb
/nz/export/ae/applications/dev/admin/host/testapply 31041
GNU gdb Fedora (6.8-27.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
Attaching to program:
/nz/export/ae/applications/dev/admin/host/testapply, process 31041
Reading symbols from
/nz/data/base/200103/library/209623/host/libnzaecpp3.so...done.
Loaded symbols for /nz/data/base/200103/library/209623/host/libnzaecpp3.so
Reading symbols from
/nz/data/base/200103/library/200652/host/libnzaechild.so...done.
```

```
Loaded symbols for /nz/data/base/200103/library/200652/host/libnzaechild.so
Reading symbols from /lib/libpthread.so.0...done.
[Thread debugging using libthread_db enabled]
[New Thread 0xf7e60ad0 (LWP 31041)]
Loaded symbols for /lib/libpthread.so.0
Reading symbols from /usr/lib/libstdc++.so.6...done.
Loaded symbols for /usr/lib/libstdc++.so.6
Reading symbols from /lib/libm.so.6...done.
Loaded symbols for /lib/libm.so.6
Reading symbols from /lib/libgcc_s.so.1...done.
Loaded symbols for /lib/libgcc_s.so.1
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xffffe410 in __kernel_vsyscall ()
(gdb) bt
#0 0xffffe410 in __kernel_vsyscall ()
#1 0x009756c1 in accept () from /lib/libpthread.so.0
#2 0xf7ea117c in _nzaeCommGetServerAccept (sock=3, sockResult=0xffa2def4)
     \ at ../nzaeparent/nzaecommunication_impl.h:259
#3 0xf7ea534c in acceptEnvironment (handle=0x85bc190, result=0xffa2df68,
     \ timeoutMilliseconds=-1) at nzaeremoteprotocol.c:266
#4 0xf7ea5b1b in nzaeRemprotAcceptEnvironment (handle=0x85bc190,
     \ result=0xffa2df68) at nzaeremoteprotocol.c:438
#5 0xf7ea51c2 in nzaeRemprotAcceptApi (handle=0x85bc190, result=0xffa2dfb4)
     \ at nzaeremoteprotocol.c:216
#6 0xf7f024f8 in nz::ae::NzaeRemoteProtocolImpl::acceptConnection
     \ (this=0x85bc180) at src/nzaeremoteprotocol.cpp:119
#7 0x0804aa7a in nz::ae::NzaeApiGenerator::getApi (this=0xffa2e100,
     \ type=nz::ae::NzaeApi::FUNCTION, fork=false) \
     at /nz/export/ae/adapters/cpp/3/sys/include/nzaeapigenerator.hpp:137 #8
0x0804ae18 in nz::ae::NzaeApiGenerator::getApi (this=0xffa2e100, \
     type=nz::ae::NzaeApi::FUNCTION)
     at /nz/export/ae/adapters/cpp/3/sys/include/nzaeapigenerator.hpp:63 #9
0x080492b2 in main () at applyopcpp.cpp:22
(gdb) b applyopcpp.cpp:23
Breakpoint 1 at 0x80492b5: file applyopcpp.cpp, line 23.
(gdb) c
Continuing.
```

Then you run the remote AE. At this point the GDB hits the set breakpoint:

```
Breakpoint 1, main () at applyopcpp.cpp:23
   run(api.aeFunction);
(gdb) c
Continuing.
```

Because this is a remote AE that has been coded to handle more than one remote connection, the GDB session is still active after the SQL returns. If you are using remote AE and forking a process for each request, the GDB fork mode must be set to debug the parent or child request with GDB command:

`set follow-fork-mode <mode>` (where mode can be parent, child, or ask)

Using a Test Harness for Troubleshooting

Using a UDX test harness may simplify testing and provide improved troubleshooting over the NPS. A detailed description is provided in the *Netezza User-Defined Functions Developer's Guide* and is not duplicated in this guide; however, some examples are provided here.

Use the the applyopcpp code for these examples. Before running this example, be sure to reregister

the code so that it is in local AE mode. All test harness examples are for local AE mode and fail in remote AE mode. See C++ Language Scalar Function for code and registration.

Normal Mode

In this case, run the same test as the applyopcpp case, but with a sample input file of:

```
+,1,2
```

To do so, specify the name of the UDF, the input file, the database, and the argument specification of the input file. You can also choose to print the output.

```
nzudxrunharness --name "apply" --file input.txt --print \
--varargs "varchar(10):int4:int4" --print
(clientmgr) Info: admin: login successful
Selected only choice
1 - APPLY() RETURNS FLOAT8
Executing /nz/kit/bin/adm/udxharness -f APPLY_func.harness -k /nz/kit

starting execution
in evaluate
DOUBLE: 3.000000
Elapsed time: 0m0.006s

External references
vtable for __cxxabiv1::__class_type_info
vtable for __cxxabiv1::__si_class_type_info
operator delete[](void*)
operator delete(void*)
operator new(unsigned int)
__cxa_begin_catch
__cxa_end_catch
__cxa_pure_virtual
__gxx_personality_v0
free
nzaeisoAdapUdfClose
nzaeisoAdapUdfCreate
nzaeisoAdapUdfGetReturn
nzaeisoAdapUdfGetSizerResult
nzaeisoAdapUdfSetController
nzaeisoAdapUdfSwitchToSizerMode
nzaeisoCtrlUdfClose
nzaeisoCtrlUdfCreate
nzaeisoCtrlUdfGetResult
nzaeisoCtrlUdfNewInput
nzaeisoCtrlUdfNoMoreInput
nzaeisoCtrlUdfSetAdapter
sprintf
strcmp
strdup
throwError

Our UDX object used 0 bytes (may be rounded up to nearest page 4096)
Our UDX return value takes up 8 bytes, with 669 bytes for miscellaneous
Our UDX arguments take up 30 bytes, with 34 bytes for miscellaneous
Our UDX state values take up 0 bytes, with 8 bytes for miscellaneous
State information may be doubled, since we need two states for merge
```

GDB

You can debug using the test harness as follows:

```
nzudxrunharness --name "apply" --file input.txt --print \
```

```
--varargs "varchar(10):int4:int4" --print --dbg
(clientmgr) Info: admin: login successful
which: no gdb-6.8 in (/nz/kit/bin:/nz/kit/bin/adm:/nz/kit/sbin:\
 /usr/lib64/qt-3.3/bin:/usr/kerberos/bin:/usr/local/bin:/bin:\
 /usr/bin:/nz/kit/bin:/nz/kit/bin/adm:/nz/kit/sbin:/sbin:\
 /opt/net/tools/bin:/opt/accurev/bin:/home/nz/bin)
Selected only choice
1 - APPLY() RETURNS FLOAT8
Executing /nz/kit/sbin/gcc/bin/gdb

/nz/kit/bin/adm/udxharness --command=APPLY_func.gdb

GNU gdb Fedora (6.8-27.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
Breakpoint 1 at 0x809e84a: file /nz/src/tools/udxharness/udxharnessimpl.cpp,
line 49.
[Thread debugging using libthread_db enabled]
[New Thread 0xf7f13ac0 (LWP 29101)]

Breakpoint 1, startingFunction () at
/nz/src/tools/udxharness/udxharnessimpl.cpp:49
49 printf("starting execution\n");
(gdb) set follow-fork-mode child
(gdb) b main
Breakpoint 2 at 0x809e873: file /nz/src/tools/udxharness/udxharnessimpl.cpp,
line 647.
(gdb) c
Continuing.
starting execution
[Thread debugging using libthread_db enabled]
[New Thread 0xf7f13ac0 (LWP 29107)]
[New process 29107]
Executing new program: /nz/export/ae/applications/dev/admin/host/testapply
Error in re-setting breakpoint 1: Function "startingFunction" not defined.
warning: Cannot initialize thread debugging library: generic
error [Thread debugging using libthread_db enabled]
[Switching to Thread 0xf7f13ac0 (LWP 29107)]

Breakpoint 2, main () at applyopcpp.cpp:19
19 NzaeApiGenerator helper;
(gdb) c
Continuing.
[New Thread 0xf7e95b90 (LWP 29108)]
in evaluate
DOUBLE: 3.000000
Elapsed time: 0m1.874s
External references
vtable for __cxxabiv1::__class_type_info
vtable for __cxxabiv1::__si_class_type_info
operator delete[](void*)
operator delete(void*)
operator new(unsigned int)
__cxa_begin_catch
__cxa_end_catch
__cxa_pure_virtual
__gxx_personality_v0
free
nzaeisoAdapUdfClose
nzaeisoAdapUdfCreate
nzaeisoAdapUdfGetReturn
nzaeisoAdapUdfGetSizerResult
nzaeisoAdapUdfSetController
nzaeisoAdapUdfSwitchToSizerMode
nzaeisoCtrlUdfClose
```

```
nzaeisoCtrlUdfCreate
nzaeisoCtrlUdfGetResult
nzaeisoCtrlUdfNewInput
nzaeisoCtrlUdfNoMoreInput
nzaeisoCtrlUdfSetAdapter
sprintf
strcmp
strdup
throwError

Our UDX object used 0 bytes (may be rounded up to nearest page 4096)
Our UDX return value takes up 8 bytes, with 669 bytes for
miscellaneous Our UDX arguments take up 30 bytes, with 34 bytes for
miscellaneous Our UDX state values take up 0 bytes, with 8 bytes for
miscellaneous State information may be doubled, since we need two
states for merge [Thread 0xf7e95b90 (LWP 29108) exited]

Program exited normally.
(gdb) q
```

Valgrind

The test harness script generates a control file that can be used directly by running this command:

```
Executing /nz/kit/bin/adm/udxharness -f APPLYOP_CPP_func.harness -k /nz/kit
```

The command can be run under valgrind, a tool to test for memory leaks and buffer overruns, as follows:

```
[nz@bdrosendev src]$ /usr/local/bin/valgrind --leak-check=full
     \ --trace-children=yes -v /nz/kit/bin/adm/udxharness \
     -f APPLYOP_CPP_func.harness -k /nz/kit
==31254== Memcheck, a memory error detector.
==31254== Copyright (C) 2002-2008, and GNU GPL'd, by Julian Seward et al.
==31254== Using LibVEX rev 1884, a library for dynamic binary translation.
==31254== Copyright (C) 2004-2008, and GNU GPL'd, by OpenWorks LLP.
==31254== Using valgrind-3.4.1, a dynamic binary instrumentation
framework. ==31254== Copyright (C) 2000-2008, and GNU GPL'd, by Julian
Seward et al. ==31254==
--31254-- Command line
--31254-- /nz/kit/bin/adm/udxharness
--31254-- -f
--31254-- APPLYOP_CPP_func.harness
--31254-- -k
--31254-- /nz/kit
--31254-- Startup, with flags:
--31254-- --leak-check=full
--31254-- --trace-children=yes
--31254-- -v
--31254-- Contents of /proc/version:
--31254--      Linux      version      2.6.18-128.el5      (mockbuild@hs20-bc1-
7.build.redhat.com) (gcc version 4.1.2 20080704 (Red Hat 4.1.2-44)) #1 SMP
Wed Dec 17 11:41:38 EST 2008
--31254-- Arch and hwcaps: X86, x86-sse1-sse2
--31254-- Page sizes: currently 4096, max supported 4096
--31254-- Valgrind library directory: /usr/local/lib/valgrind --
31254-- Reading syms from /lib/ld-2.5.so (0x7dc000)
--31254-- Reading syms from /nz/kit/bin/adm/udxharness (0x8048000)
--31254-- Reading syms from /usr/local/lib/valgrind/x86-
linux/memcheck (0x38000000)
--31254-- object doesn't have a dynamic symbol table
--31254-- Reading suppressions file: /usr/local/lib/valgrind/default.supp
--31254-- REDIR: 0x7f17b0 (index) redirected to
0x38039953 (vgPlain_x86_linux_REDIR_FOR_index)
--31254-- Reading syms from /usr/local/lib/valgrind/x86-

linux/vgpreload_core.so (0x47c0000)
```

```
--31254-- Reading syms from /usr/local/lib/valgrind/x86-
linux/vgpreload_memcheck.so (0x47c6000)
==31254== WARNING: new redirection conflicts with existing -- ignoring it
--31254-- new: 0x007f17b0 (index ) R-> 0x047c97d0 index
--31254-- REDIR: 0x7f1950 (strlen) redirected to 0x47c9990 (strlen)
--31254-- Reading syms from /lib/libdl-2.5.so (0x982000)
--31254-- Reading syms from /lib/libpthread-2.5.so (0x969000)
--31254-- Reading syms from /lib/libm-2.5.so (0x940000)
--31254-- Reading syms from /lib/libc-2.5.so (0x7fa000)
--31254-- REDIR: 0x86aba0 (memset) redirected to 0x47c9cf0 (memset)
--31254-- REDIR: 0x86b090 (memcpy) redirected to 0x47cab90 (memcpy)
--31254-- REDIR: 0x869d00 (rindex) redirected to 0x47c96b0 (rindex)
--31254-- REDIR: 0x869960 (strlen) redirected to 0x47c9970 (strlen)
--31254-- REDIR: 0x865330 (malloc) redirected to 0x47c89f0 (malloc)
--31254-- REDIR: 0x869400 (strcmp) redirected to 0x47c9a40 (strcmp)
--31254-- REDIR: 0x866b30 (free) redirected to 0x47c8590 (free)
--31254-- REDIR: 0x869290 (index) redirected to 0x47c97a0 (index)
starting execution
--31254-- REDIR: 0x86a6a0 (memchr) redirected to 0x47c9b90 (memchr)
--31254-- REDIR: 0x869470 (strcpy) redirected to 0x47cae00 (strcpy)
--31254-- REDIR: 0x869b50 (strncmp) redirected to 0x47c99d0 (strncmp)
--31254-- REDIR: 0x869c50 (strncpy) redirected to 0x47cace0 (strncpy)
--31254-- REDIR: 0x86b940 (rawmemchr) redirected to 0x47c9dc0 (rawmemchr)
--31254-- REDIR: 0x86a840 (bcmp) redirected to 0x47c9c30 (bcmp)
--31254-- REDIR: 0x8690e0 (strcat) redirected to 0x47ca130 (strcat)
--31254-- REDIR: 0x825b00 (setenv) redirected to 0x47c9ea0 (setenv)
--31254-- REDIR: 0x866d20 (realloc) redirected to 0x47c8ab0 (realloc)
--31254-- REDIR: 0x86ac00 (mempcpy) redirected to 0x47ca430 (mempcpy)
--31254-- REDIR: 0x865010 (calloc) redirected to 0x47c7b10 (calloc)
--31254-- Reading syms from
/nz/data/base/200103/library/200649/host/libnzaeparent.so (0x4e52000)
--31254-- Reading syms from /usr/lib/libstdc++.so.6.0.8 (0x9c6000)
--31254-- object doesn't have a symbol table
--31254-- Reading syms from /lib/libgcc_s-4.1.2-20080825.so.1 (0x988000)
--31254-- object doesn't have a symbol table
--31254-- Reading syms from
/nz/data/base/200103/library/200650/host/libnzaeadapters.so (0x7a25000)
--31254-- Reading syms from
/nz/data/base/200103/library/200652/host/libnzaechild.so (0x4eab000)
--31254-- Reading syms from /nz/data/base/200103/udf/217194.oh (0x4c26000)
--31254-- REDIR: 0x86ab30 (memmove) redirected to 0x47c9d40 (memmove)
DOUBLE: 3.000000
Elapsed time: 0m0.903s

External references
vtable for __cxxabiv1::__class_type_info
vtable for __cxxabiv1::__si_class_type_info
operator delete[](void*)
operator delete(void*)
operator new(unsigned int)
__cxa_begin_catch
__cxa_end_catch
__cxa_pure_virtual
__gxx_personality_v0
free
nzaeisoAdapUdfClose
nzaeisoAdapUdfCreate
nzaeisoAdapUdfGetReturn
nzaeisoAdapUdfGetSizerResult
nzaeisoAdapUdfSetController
nzaeisoAdapUdfSwitchToSizerMode
nzaeisoCtrlUdfClose
nzaeisoCtrlUdfCreate
nzaeisoCtrlUdfGetResult
nzaeisoCtrlUdfNewInput
nzaeisoCtrlUdfNoMoreInput
nzaeisoCtrlUdfSetAdapter
sprintf
```

```
strcmp
strdup
throwError

Our UDX object used 0 bytes (may be rounded up to nearest page 4096)
Our UDX return value takes up 8 bytes, with 669 bytes for
miscellaneous Our UDX arguments take up 30 bytes, with 34 bytes for
miscellaneous Our UDX state values take up 0 bytes, with 8 bytes for
miscellaneous State information may be doubled, since we need two
states for merge --31254-- Discarding syms at 0x4eb3fb0-0x4ed1b64 in \
     /nz/data/base/200103/library/200652/host/libnzaechild.so due to munmap()
--31254-- Discarding syms at 0x4c27b90-0x4c28c54 in \
     /nz/data/base/200103/udf/217194.oh due to munmap()
--31254-- Discarding syms at 0x7a41250-0x7a62d44 in \
     /nz/data/base/200103/library/200650/host/libnzaeadapters.so due to
     \ munmap()
--31254-- Discarding syms at 0x4e613f0-0x4e975d4 in
/nz/data/base/200103/library/200649/host/libnzaeparent.so due to munmap()
--31254-- Discarding syms at 0xa05c50-0xa810c4 in /usr/lib/libstdc++.so.6.0.8 \
     due to munmap()
--31254-- Discarding syms at 0x989660-0x990e94 in /lib/libgcc_s-
4.1.2-20080825.so.1 due to munmap()
==31254==
==31254== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 39 from 1)
--31254--
--31254-- supp: 39 dl_relocate_object
==31254== malloc/free: in use at exit: 136 bytes in 4 blocks.
==31254== malloc/free: 464 allocs, 460 frees, 5,522,815 bytes allocated.
==31254==
==31254== searching for pointers to 4 not-freed blocks.
==31254== checked 170,496 bytes.
==31254==
==31254== 12 bytes in 1 blocks are definitely lost in loss record 1 of
4 ==31254== at 0x47C8A58: malloc (vg_replace_malloc.c:207)
==31254== by 0x8153666: operator new(unsigned int) (new_op.cc:57)
==31254== by 0x809D52B: __static_initialization_and_destruction_0(int, int)
     \ (udxharnessimpl.cpp:45)
==31254== by 0x809D583: global constructors keyed to stmt
     \ (udxharnessimpl.cpp:724)
==31254== by 0x8159864: (within /nz/kit/bin/adm/udxharness)
==31254== by 0x809C74C: (within /nz/kit/bin/adm/udxharness)
==31254== by 0x81597D1: __libc_csu_init (in /nz/kit/bin/adm/udxharness)
==31254== by 0x80FE30: (below main) (in /lib/libc-2.5.so)
==31254==
==31254== LEAK SUMMARY:
==31254== definitely lost: 12 bytes in 1 blocks.
==31254== possibly lost: 0 bytes in 0 blocks.
==31254== still reachable: 124 bytes in 3 blocks.
==31254== suppressed: 0 bytes in 0 blocks.
==31254== Reachable blocks (those to which a pointer was found) are not
shown. ==31254== To see them, rerun with: --leak-check=full --show-
reachable=yes --31254-- memcheck: sanity checks: 15 cheap, 2 expensive
--31254-- memcheck: auxmaps: 0 auxmap entries (0k, 0M) in use
--31254-- memcheck: auxmaps_L1: 0 searches, 0 cmps, ratio 0:10
--31254-- memcheck: auxmaps_L2: 0 searches, 0 nodes
--31254-- memcheck: SMs: n_issued = 40 (640k, 0M)
--31254-- memcheck: SMs: n_deissued = 6 (96k, 0M)
--31254-- memcheck: SMs: max_noaccess = 65535 (1048560k, 1023M)
--31254-- memcheck: SMs: max_undefined = 81 (1296k, 1M)
--31254-- memcheck: SMs: max_defined = 66 (1056k, 1M)
--31254-- memcheck: SMs: max_non_DSM = 38 (608k, 0M)
--31254-- memcheck: max sec V bit nodes: 1 (0k, 0M)
--31254-- memcheck: set_sec_vbits8 calls: 1 (new: 1, updates: 0)
--31254-- memcheck: max shadow mem size: 912k, 0M
--31254-- translate: fast SP updates identified: 10,841 ( 88.8%)
--31254-- translate: generic_known SP updates identified: 881 ( 7.2%)
--31254-- translate: generic_unknown SP updates identified: 481 ( 3.9%)
--31254-- tt/tc: 16,569 tt lookups requiring 17,820 probes
```

```
--31254-- tt/tc: 16,569 fast-cache updates, 9 flushes
--31254-- transtab: new 7,737 (178,651 -> 2,537,524; ratio 142:10) [0 scs]
--31254-- transtab: dumped 0 (0 -> ??)
--31254-- transtab: discarded 1,824 (37,849 -> ??)
--31254-- scheduler: 1,505,348 jumps (bb entries).
--31254-- scheduler: 15/10,159 major/minor sched events.
--31254-- sanity: 16 cheap, 2 expensive checks.
--31254-- exectx: 1,543 lists, 1,449 contexts (avg 0 per list)
--31254-- exectx: 2,327 searches, 1,766 full compares (758 per 1000)
--31254-- exectx: 3 cmp2, 178 cmp4, 0 cmpAll
--31254-- errormgr: 17 supplist searches, 515 comparisons during search
--31254-- errormgr: 39 errlist searches, 178 comparisons during search
```

Callgrind

The test harness script generates a control file that can be used directly by running this command:

```
Executing /nz/kit/bin/adm/udxharness -f APPLYOP_CPP_func.harness -k /nz/kit
```

The command can be run under callgrind, a tool to profile code, as follows:

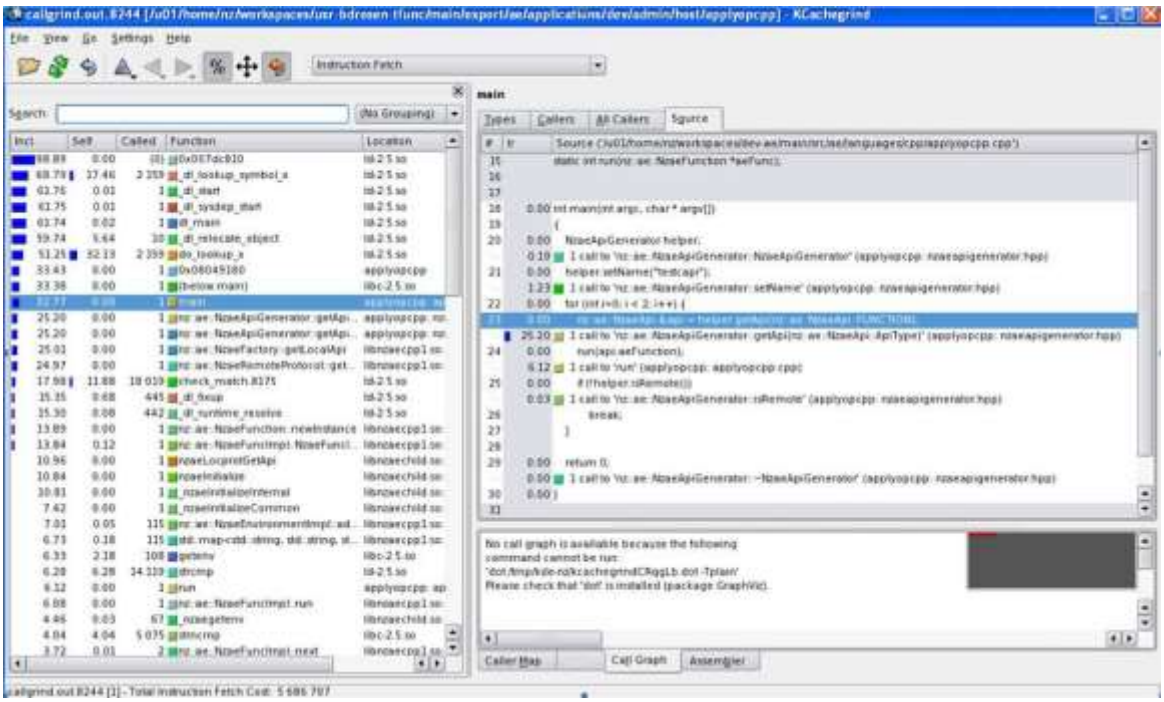
```
[nz@bdrosendev src]$ /usr/local/bin/valgrind --tool=callgrind \
--trace-children=yes -v /nz/kit/bin/adm/udxharness -f APPLYOP_CPP_func.harness -k
/nz/kit
==31290== Callgrind, a call-graph generating cache profiler.
==31290== Copyright (C) 2002-2008, and GNU GPL'd, by Josef Weidendorfer et al.
==31290== Using LibVEX rev 1884, a library for dynamic binary translation.
==31290== Copyright (C) 2004-2008, and GNU GPL'd, by OpenWorks LLP.
==31290== Using valgrind-3.4.1, a dynamic binary instrumentation framework.
==31290== Copyright (C) 2000-2008, and GNU GPL'd, by Julian Seward et al.
==31290==
--31290-- Command line
--31290--    /nz/kit/bin/adm/udxharness
--31290--    -f
--31290--    APPLYOP_CPP_func.harness
--31290--    -k
--31290--    /nz/kit
--31290-- Startup, with flags:
--31290--    --tool=callgrind
--31290--    --trace-children=yes
--31290--    -v
--31290-- Contents of /proc/version:
--31290--   Linux version 2.6.18-128.el5 \
     (mockbuild@hs20-bc1-7.build.redhat.com) (gcc version 4.1.2 20080704 \
     (Red Hat 4.1.2-44)) #1 SMP Wed Dec 17 11:41:38 EST 2008
--31290-- Arch and hwcaps: X86, x86-sse1-sse2
--31290-- Page sizes: currently 4096, max supported 4096
--31290-- Valgrind library directory: /usr/local/lib/valgrind
==31290== For interactive control, run 'callgrind_control -h'.
--31290-- Reading syms from /lib/ld-2.5.so (0x7dc000)
--31290-- Reading syms from /nz/kit/bin/adm/udxharness (0x8048000)
--31290-- Reading syms from /usr/local/lib/valgrind/x86-linux/callgrind \
      (0x38000000)
--31290--    object doesn't have a dynamic symbol table
--31290-- Found runtime_resolve (x86-def): ld-2.5.so +0x12cd0=0x7ef4c0, \
     length 24
--31290-- Reading syms from \
     /usr/local/lib/valgrind/x86-linux/vgpreload_core.so (0x46a0000)
--31290-- Reading syms from /lib/libdl-2.5.so (0x982000)
--31290-- Reading syms from /lib/libpthread-2.5.so (0x969000)
--31290-- Reading syms from /lib/libm-2.5.so (0x940000)
--31290-- Reading syms from /lib/libc-2.5.so (0x7fa000)
--31290-- Symbol match: found runtime_resolve: ld-2.5.so +0x7ef4c0=0x7ef4c0
starting execution
--31290-- Reading syms from
/nz/data/base/200103/library/200649/host/libnzaeparent.so (0x46c7000)
```

```
--31290-- Reading syms from /usr/lib/libstdc++.so.6.0.8 (0x9c6000)
--31290--    object doesn't have a symbol table
--31290-- Reading syms from /lib/libgcc_s-4.1.2-20080825.so.1 (0x988000)
--31290--    object doesn't have a symbol table
--31290-- Reading syms from
/nz/data/base/200103/library/200650/host/libnzaeadapters.so (0x4718000)
--31290-- Reading syms from
/nz/data/base/200103/library/200652/host/libnzaechild.so (0x4763000)
--31290-- Reading syms from /nz/data/base/200103/udf/217194.oh (0x46b3000)
 DOUBLE: 3.000000
Elapsed time: 0m0.610s

External references
vtable for __cxxabiv1::__class_type_info
vtable for __cxxabiv1::__si_class_type_info
operator delete[](void*)
operator delete(void*)
operator new(unsigned int)
__cxa_begin_catch
__cxa_end_catch
__cxa_pure_virtual
__gxx_personality_v0
free
nzaeisoAdapUdfClose
nzaeisoAdapUdfCreate
nzaeisoAdapUdfGetReturn
nzaeisoAdapUdfGetSizerResult
nzaeisoAdapUdfSetController
nzaeisoAdapUdfSwitchToSizerMode
nzaeisoCtrlUdfClose
nzaeisoCtrlUdfCreate
nzaeisoCtrlUdfGetResult
nzaeisoCtrlUdfNewInput
nzaeisoCtrlUdfNoMoreInput
nzaeisoCtrlUdfSetAdapter
sprintf
strcmp
strdup
throwError

Our UDX object used 0 bytes (may be rounded up to nearest page 4096)
Our UDX return value takes up 8 bytes, with 669 bytes for
miscellaneous Our UDX arguments take up 30 bytes, with 34 bytes for
miscellaneous Our UDX state values take up 0 bytes, with 8 bytes for
miscellaneous State information may be doubled, since we need two
states for merge --31290-- Discarding syms at 0x476bfb0-0x4789b64 in \
     /nz/data/base/200103/library/200652/host/libnza¬echild.so due to munmap()
--31290-- Discarding syms at 0x46b4b90-0x46b5c54 in \
     /nz/data/base/200103/udf/217194.oh due to munmap()
--31290-- Discarding syms at 0x4734250-0x4755d44 in \
     /nz/data/base/200103/library/200650/host/libnza¬eadapters.so due to
     \ munmap()
--31290-- Discarding syms at 0x46d63f0-0x470c5d4 in \
      /nz/data/base/200103/library/200649/host/libnza¬eparent.so due to \
      munmap()
--31290-- Discarding syms at 0xa05c50-0xa810c4 in /usr/lib/libstdc++.so.6.0.8
     \ due to munmap()
--31290-- Discarding syms at 0x989660-0x990e94 in \
     /lib/libgcc_s-4.1.2-20080825.so.1 due to munmap()
==31290==
--31290-- Start dumping at BB 1776785 (Prg.Term.)...
--31290-- Dump to /nz/src/callgrind.out.31290
--31290-- Dumping done.
--31290--
--31290-- Distinct objects: 14
--31290-- Distinct files:   147
--31290-- Distinct fns:     1987
--31290-- Distinct contexts:1987
```

```
--31290-- Distinct BBs:      9286
--31290-- Cost entries:      6488 (Chunks 1)
--31290-- Distinct BBCCs:    9697
--31290-- Distinct JCCs:     3140
--31290-- Distinct skips:    1051
--31290-- BB lookups:        9286
--31290-- With full      debug info: 47% (4388)
--31290-- With file/line debug info:  0% (0)
--31290-- With fn name   debug info: 43% (4030)
--31290-- With no        debug info:  9% (868)
--31290-- BBCC Clones:       411
--31290-- BBs Retranslated:  0
--31290-- Distinct instrs:   46884
--31290--
--31290-- LRU Contxt Misses:  1987
--31290-- LRU BBCC Misses:   372
--31290-- LRU JCC Misses:    3141
--31290-- BBs Executed:      1776786
--31290-- Calls:             118594
--31290-- CondJMP followed:  0
--31290-- Boring JMPs:       0
--31290-- Recursive calls:   63
--31290-- Returns:           118594
--31290--
==31290== Events    : Ir
==31290== Collected : 10463471
==31290==
==31290== I    refs:      10,463,471
--31290-- translate:            fast SP updates identified: 0 (    --%)
--31290-- translate:   generic_known SP updates identified: 0 (    --%)
--31290-- translate: generic_unknown SP updates identified: 0 (    --%)
--31290--     tt/tc: 20,206 tt lookups requiring 21,949 probes
--31290--     tt/tc: 20,206 fast-cache updates, 8 flushes
--31290--  transtab: new        9,286 (163,107 -> 1,111,174; ratio 68:10) \
     [0 scs]
--31290--  transtab: dumped     0 (0 -> ??)
--31290--  transtab: discarded  2,324 (34,320 -> ??)
--31290-- scheduler: 1,776,785 jumps (bb entries).
--31290-- scheduler: 17/11,301 major/minor sched events.
--31290--    sanity: 18 cheap, 2 expensive checks.
--31290--    exectx: 769 lists, 0 contexts (avg 0 per list)
--31290--    exectx: 0 searches, 0 full compares (0 per 1000)
--31290--    exectx: 0 cmp2,  0 cmp4, 0 cmpAll
--31290--  errormgr: 0 supplist searches, 0 comparisons during search
--31290--  errormgr: 0 errlist searches, 0 comparisons during search
The above code generated two files that can be loaded in kcachegrind to get
     \ a visual representation.
kcachegrind callgrind.out.8244 &
```

The files get produced by callgrind in the current directory. Use KCachegrind to view them. For example:

Figure 2: Kcachegrind displaying callgrind files



Using the Integrated Java Debugger

Java has an integrated network debugger that can be used for debugging Java AEs. It can be used to run the Java version of the apply test (the Java scalar function), shown here:

```
import org.netezza.ae.*;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class TestJavaInterface {

    private static final Executor exec =
        Executors.newCachedThreadPool();


    public static final void main(String [] args) {
        try {
            mainImpl(args);
        } catch (Throwable t) {
            System.err.println(t.toString());
            NzaeUtil.logException(t, "main");
        }
    }

    public static final void mainImpl(String [] args) {
```

```
        NzaeApiGenerator helper = new NzaeApiGenerator();

        while (true) {
            final NzaeApi api = helper.getApi(NzaeApi.FUNCTION,false);
            if (api.apiType == NzaeApi.FUNCTION) {
                if (!helper.isRemote()) {
                    run(api.aeFunction);
                    break;
                } else {
                    Runnable task = new Runnable() {
                            public void run() {
                                try {
                                    TestJavaInterface.run(api.aeFunction);
                                } finally {
                                    api.aeFunction.close();
                                }
                            }
                        };
                    exec.execute(task);


                }
            }
        }
        helper.close();
    }

    public static class MyHandler implements NzaeMessageHandler
    {
        public void evaluate(Nzae ae, NzaeRecord input, NzaeRecord output) {

            final NzaeMetadata meta = ae.getMetadata();

            int op = 0;
            double result = 0;

            if (meta.getOutputColumnCount() != 1 || meta.getOutputNzType(0)
                != NzaeDataTypes.NZUDSUDX_DOUBLE) { throw new
                NzaeException("expecting one output column of type
double");
            }
            if (meta.getInputColumnCount() < 1) {
                throw new NzaeException("expecting at least one input column");
            }
            if (meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_FIXED &&
                meta.getInputNzType(0) != NzaeDataTypes.NZUDSUDX_VARIABLE) {
                throw new NzaeException("first input column expected to be a
string type");
            }

            for (int i = 0; i < input.size(); i++) {
                if (input.getField(i) == null) {
                    continue;
                }
                int dataType = meta.getInputNzType(i);
                if (i == 0) {
                    if (!(dataType == NzaeDataTypes.NZUDSUDX_FIXED
                              dataType == NzaeDataTypes.NZUDSUDX_VARIABLE))

                        { ae.userError("first column must be a string");

                    }
                    String opStr = input.getFieldAsString(0);
                    if (opStr.equals("*")) {
                        result = 1;
                        op = OP_MULT;
                    }
                    else if (opStr.equals("+")) {
```

```
                            result = 0;
                            op = OP_ADD;
                        }
                        else {
                            ae.userError("invalid operator = " + opStr);
                        }
                        continue;
                    }
                    switch (dataType) {
                        case NzaeDataTypes.NZUDSUDX_INT8:
                        case NzaeDataTypes.NZUDSUDX_INT16:
                        case NzaeDataTypes.NZUDSUDX_INT32:
                        case NzaeDataTypes.NZUDSUDX_INT64:
                        case NzaeDataTypes.NZUDSUDX_FLOAT:
                        case NzaeDataTypes.NZUDSUDX_DOUBLE:
                        case NzaeDataTypes.NZUDSUDX_NUMERIC32:
                        case NzaeDataTypes.NZUDSUDX_NUMERIC64:
                        case NzaeDataTypes.NZUDSUDX_NUMERIC128:
                            switch (op) {
                                case OP_ADD:
                                    result +=
input.getFieldAsNumber(i).doubleValue();
                                    break;
                                case OP_MULT:
                                    result *=
input.getFieldAsNumber(i).doubleValue();
                                    break;
                                default:
                                    break;
                            }
                            break;

                        default:
                            break;
                    }
                } // end of column for loop

                output.setField(0, result);

            }

        }
        private static final int OP_ADD = 1;
        private static final int OP_MULT = 2;

        public static int run(Nzae ae)
        {
            ae.run(new MyHandler());
            return 0;
        }
}
```

Compilation

Compile the code:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java --template compile
     \ --version 3 TestJavaInterface.java
```

Registration

Register the Java AE with the appropriate debugging options. To register a Java AE applyop case to listen on port 8000 for incoming debugging connections, run this command:

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --sig "applyop_java(varargs)" \
    --return "double" \
    --language java --template udf --define "java_class=TestJavaInterface" \
```

```
--version 3 --environment "'NZAE_NUMBER_PARAMETERS'='3'" \
--environment "'NZAE_PARAMETER1'='-Xdebug'" --environment \
"'NZAE_PARAMETER2'='-Xrunjdwp:transport=dt_socket,server=y,address=8000'" \
--environment "'NZAE_PARAMETER3'=' TestJavaInterface'"
```

Running

Then run the AE, which pauses, waiting for the client debugger:

```
SELECT applyop_java('+',1,2);
```

Using the IBM Netezza Eclipse Plug-In

To debug the file using using the IBM Netezza Development Environment Plug-in for Eclipse application:

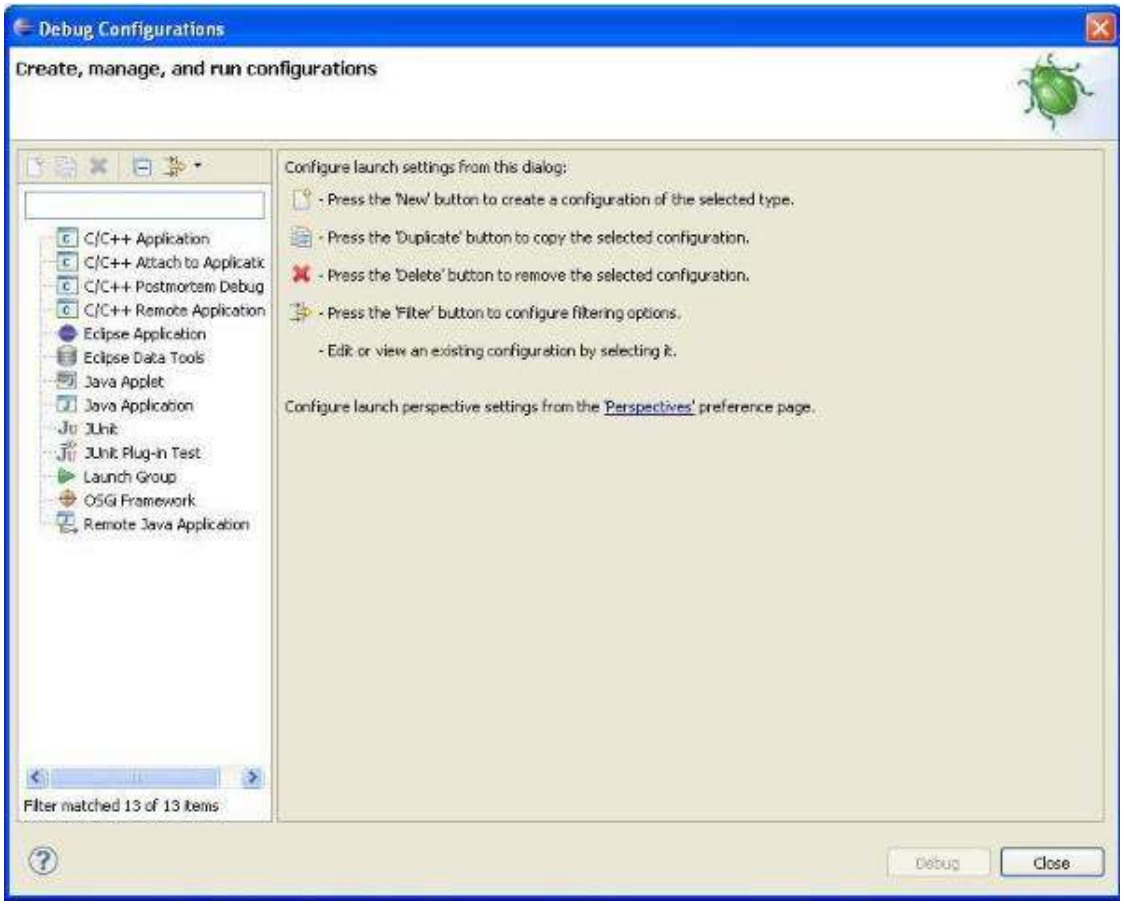
Create a Java project that contains the source code for the AE and has access to the nzae.JAR in the class path.

Set a break point in the main method of the Test Class.

Select **Run > Debug Configurations**.

The Debug Configurations dialog box appears, as shown in Figure 3.

Figure 3: Debug Configurations Dialog Box



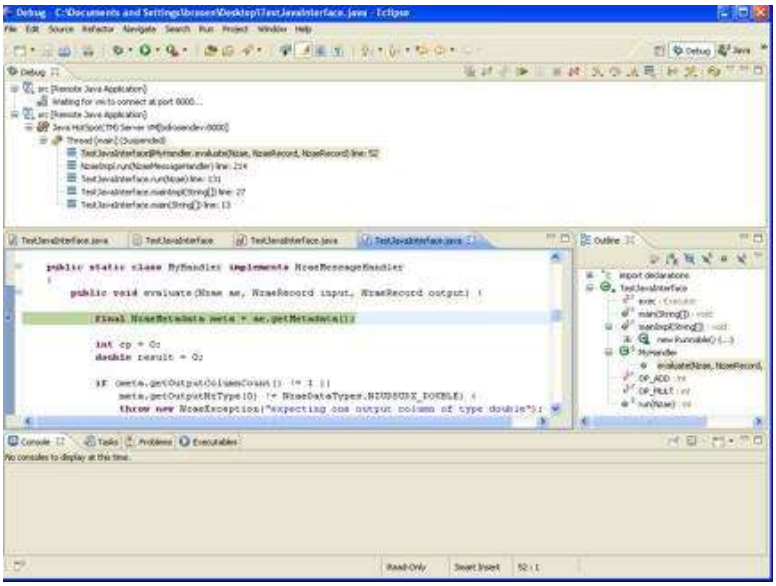
In the dialog box, select **Remote Java Applications** and create a new configuration.

The parameters entered in the Connect tab should reflect Figure 4, with the name of the host being set to the name of the NPS appliance.

Figure 4: Debug Configuration Settings

Click **Debug**. The debugger opens and you can step through the code to debug.

Figure 5: Sample Debugger Screen



When the debugging session is complete, the SQL command finishes.

CHAPTER 18
Debugging Examples

C Language Logging and Runtime Information

This example uses the following file name:

```
log.c
```

Code

The code in this example explores functionality that may not be commonly used. Logging can be used to help trace the execution of an AE. To be of use, the AE must be registered with a log mask and logging must be enabled via **nzudxdbg**. To receive system logging messages in the window where the NPS system was started, you need to start NPS with the '-i' option (for example, 'nzstart -i'). This causes the NPS processes to remain attached to the terminal. Runtime information provides statistics about the NPS system, including the number of dataslices, the number of SPUs, and locus of execution.

```
#include <stdio.h>
#include <stdlib.h>

#include  "nzaeapis.h"

static int run(NZAE_HANDLE h);
int main(int argc, char * argv[])
{
    if (nzaeIsLocal())
    {
        NzaeInitialization arg;
        memset(&arg, 0, sizeof(arg));
                arg.ldkVersion = NZAE_LDK_VERSION;
        if (nzaeInitialize(&arg))
        {
            fprintf(stderr, "initialization failed\n");
            return -1;
        }
        run(arg.handle);
        nzaeClose(arg.handle);
    }
    else {
```

```
        NZAECONPT_HANDLE hConpt = nzaeconptCreate();
        if (!hConpt)
        {
            fprintf(stderr, "error creating connection point\n");
            fflush(stderr);
            return -1;
        }
        const char * conPtName = nzaeRemprotGetRemoteName();
        if (!conPtName)
        {
            fprintf(stderr, "error getting connection point name\n");
            fflush(stderr);
            exit(-1);
        }
        if (nzaeconptSetName(hConpt, conPtName))
        {
            fprintf(stderr, "error setting connection point name\n");
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NzaeremprotInitialization args;
        memset(&args, 0, sizeof(args));
        args.ldkVersion = NZAE_LDK_VERSION;
        args.hConpt = hConpt;
        if (nzaeRemprotCreateListener(&args))
        {
            fprintf(stderr, "unable to create listener - %s\n", \
                 args.errorMessage);
            fflush(stderr);
            nzaeconptClose(hConpt);
            return -1;
        }
        NZAEREMPROT_HANDLE hRemprot = args.handle;
        NzaeApi api;
        int i;
        for (i = 0; i < 5; i++)
        {
            if (nzaeRemprotAcceptApi(hRemprot, &api))
            {
                fprintf(stderr, "unable to accept API - %s\n", \
                     nzaeRemprotGetLastErrorText(hRemprot));
                fflush(stderr);
                nzaeconptClose(hConpt);
                nzaeRemprotClose(hRemprot);
                return -1;
            }
            if (api.apiType != NZAE_API_FUNCTION)
            {
                fprintf(stderr, "unexpected API returned\n");
                fflush(stderr);
                nzaeconptClose(hConpt);
                nzaeRemprotClose(hRemprot);
                return -1;
            }
            printf("testcapi: accepted a remote request\n");
            fflush(stdout);
            run(api.handle.function);
        }
        nzaeRemprotClose(hRemprot);
        nzaeconptClose(hConpt);
    }
    return 0;
}

static void validateRuntime(NZAE_HANDLE handle, NzaeRuntime *aeRun, int64_t
     \ *args, int type)
{
```

```
   argss:
   dataslice, transaction, hardware, numslices, numspus,
locus int64_t dataslice = args[0];
int64_t txid = args[1];
int64_t hwid = args[2];
int64_t nslices = args[3];
int64_t nspus = args[4];
int64_t locus = args[5];
char buf[1024];
if (locus != (int64_t)aeRun->locus)
{
    sprintf(buf, "test fails: 1\n");
    nzaeLog(handle,NZAE_LOG_DEBUG, buf);
}
if (aeRun->suggestedMemoryLimit != 123)
{
    sprintf(buf, "test fails: 7 \n");
    nzaeLog(handle,NZAE_LOG_DEBUG, buf);
}
if (aeRun->userQuery == false)
{
    sprintf(buf, "test fails: 8 \n");
    nzaeLog(handle,NZAE_LOG_DEBUG, buf);
}
if (type != (int64_t)aeRun->adapterType)
{
    sprintf(buf, "test fails: 9 \n");
    nzaeLog(handle,NZAE_LOG_DEBUG, buf);
}
if (locus == (int64_t)NZAE_LOCUS_POSTGRES) {

    return;
}

if (dataslice != (int64_t)aeRun->dataSliceId)
{
    sprintf(buf, "test fails: 2 \n");
    nzaeLog(handle,NZAE_LOG_DEBUG, buf);

}

if (txid != 0 && txid != (int64_t)aeRun->transactionId)
{
    sprintf(buf, "test fails: 3 \n");
    nzaeLog(handle,NZAE_LOG_DEBUG, buf);

}

if (hwid != (int64_t)aeRun->hardwareId)
{
    sprintf(buf, "test fails: 4 \n");
    nzaeLog(handle,NZAE_LOG_DEBUG, buf);

}

if (nslices != (int64_t)aeRun->numberDataSlices)
{
    sprintf(buf, "test fails: 5 \n");
    nzaeLog(handle,NZAE_LOG_DEBUG, buf);

}

if (nspus != (int64_t)aeRun->numberSpus)
{
    sprintf(buf, "test fails: 6 \n");
    nzaeLog(handle,NZAE_LOG_DEBUG, buf);
```

```
    }

}

static int run(NZAE_HANDLE h)
{
    NzaeMetadata metadata;
    if (nzaeGetMetadata(h, &metadata))
    {
        fprintf(stderr, "get metadata failed\n");
        return -1;
    }

    #define CHECK(value) \
    { \
        NzaeRcCode rc = value; \
        if (rc) \
        { \
            const char * format = "%s in %s at %d"; \
            fprintf(stderr, format,  \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            nzaeUserError(h, format, \
                nzaeGetLastErrorText(h), __FILE__, __LINE__); \
            exit(-1); \
        } \
    }

    for (;;)
    {
        int64_t ar[6];
        int i;
        double result = 0;
        NzaeRcCode rc = nzaeGetNext(h);
        if (rc == NZAE_RC_END)
        {
            break;
        }

        NzudsData * input = NULL;
        for (i=0; i < 6; i++) {
            CHECK(nzaeGetInputColumn(h, i, &input));
            if (input->type == NZUDSUDX_INT32)
            {
                ar[i] = *input->data.pInt32;
            }
            else if (input->type == NZUDSUDX_INT64)
            {
                ar[i] = *input->data.pInt64;
            }
        }
        NzaeRuntime aeRun;
        nzaeGetRuntime(h,&aeRun);
        validateRuntime(h, &aeRun, ar, NZAE_ADAPTER_UDTF);
        CHECK(nzaeSetOutputString(h, 0, "done"));
        CHECK(nzaeOutputResult(h));
    }
    nzaeDone(h);
    return 0;
}
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language system --version 3 \
     --template compile log.c --exe log
```

Registration

Register the example using the **--mem** and **--mask** options. The **--mask** option enables logging for DEBUG and the **--mem** option sets the memory runtime information.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language system --version 3
     \ --template udtf --exe log \
     --sig "runtime_c(int8,int8,int8,int8,int8,int8)" \
     --return "table(output varchar(100))" --mask debug --mem 123
```

Running

Before running, enable logging by running **nzudxdbg**:

```
nzudxdbg
Processing 1 spus
.
done
Processing host
done
```

Then run the SQL:

```
SELECT * FROM TABLE WITH FINAL(runtime_c(1,1,1,1,1,1));
 OUTPUT
--------
 done
(1 row)
```

The NPS system logging appears in the window where the NPS system was started:

```
05-03-10 15:03:12 (dbos.18728) [d,udx ] test fails: 2
05-03-10 15:03:12 (dbos.18728) [d,udx ] test fails: 3
05-03-10 15:03:12 (dbos.18728) [d,udx ] test fails: 4
```

This is the expected output since the arguments specified did not match the runtime in those cases.

C++ Language Logging and Runtime Information

This example uses the following file name:

```
runtime.cpp
```

Code

The code in this example explores functionality that may not be commonly used. Logging can be used to help trace the execution of an AE. To be of use, the AE must be registered with a log mask and logging must be enabled via **nzudxdbg**. To receive system logging messages in the window where the NPS system was started, you need to start NPS with the '-i' option (for example, 'nzstart -i'). This causes the NPS processes to remain attached to the terminal. Runtime information provides statistics about the NPS system, including the number of dataslices, the number of SPUs, and locus of execution.

```
#include <nzaefactory.hpp>

using namespace nz::ae;

static int run(nz::ae::NzaeFunction *aeFunc);
```

```
int main(int argc, char * argv[])
{
    NzaeApiGenerator helper;
       The following line is only needed if a launcher is not
    used helper.setName("testcapi");
    for (int i=0; i < 2; i++) {
        nz::ae::NzaeApi &api =
        helper.getApi(nz::ae::NzaeApi::FUNCTION); run(api.aeFunction);
        if (!helper.isRemote())
            break;
    }
    return 0;
}

static void validateRuntime(NzaeFunction *aeFunc, int64_t *args, int type)
{
       argss:
       dataslice, transaction, hardware, numslices, numspus,
    locus int64_t dataslice = args[0];
    int64_t txid = args[1];
    int64_t hwid = args[2];
    int64_t nslices = args[3];
    int64_t nspus = args[4];
    int64_t locus = args[5];
    const NzaeRuntime &aeRun = aeFunc-
    >getRuntime(); char buf[1024];
    if (locus != (int64_t)aeRun.getLocus())
    {
        sprintf(buf, "test fails: 1\n");
        aeFunc->log(NzaeFunction::LOG_DEBUG, buf);
    }
    if (aeRun.getSuggestedMemoryLimit() != 123)
    {
        sprintf(buf, "test fails: 7 \n");
        aeFunc->log(NzaeFunction::LOG_DEBUG, buf);
    }
    if (aeRun.getUserQuery() == false)
    {
        sprintf(buf, "test fails: 8 \n");
        aeFunc->log(NzaeFunction::LOG_DEBUG, buf);
    }
    if (type != (int64_t)aeRun.getAdapterType())
    {
        sprintf(buf, "test fails: 9 \n");
        aeFunc->log(NzaeFunction::LOG_DEBUG, buf);
    }
    if (locus == (int64_t)NzaeRuntime::NZAE_LOCUS_POSTGRES)
        { return;
    }

    if (dataslice != (int64_t)aeRun.getDataSliceId())
    {
        sprintf(buf, "test fails: 2 \n");
        aeFunc->log(NzaeFunction::LOG_DEBUG, buf);
    }

    if (txid != 0 && txid != (int64_t)aeRun.getTransactionId())
    {
        sprintf(buf, "test fails: 3 \n");
        aeFunc->log(NzaeFunction::LOG_DEBUG, buf);

    }

    if (hwid != (int64_t)aeRun.getHardwareId())
    {
```

```
        sprintf(buf, "test fails: 4 \n");
        aeFunc->log(NzaeFunction::LOG_DEBUG, buf);
    }

    if (nslices != (int64_t)aeRun.getNumberDataSlices())
    {
        sprintf(buf, "test fails: 5 \n");
        aeFunc->log(NzaeFunction::LOG_DEBUG, buf);
    }

    if (nspus != (int64_t)aeRun.getNumberSpus())
    {
        sprintf(buf, "test fails: 6 \n");
        aeFunc->log(NzaeFunction::LOG_DEBUG, buf);
    }
}

class MyHandler : public NzaeFunctionMessageHandler
{
public:

    void evaluate(NzaeFunction& api, NzaeRecord &input, NzaeRecord &result) {

        int64_t ar[6];

        for (int i=0; i < 6; i++)
        {
            nz::ae::NzaeField &fi = input.get(i);
            if (fi.type() == NzaeDataTypes::NZUDSUDX_INT64){
                ar[i] = (int64_t)(NzaeInt64Field&)fi;
            }
            else if (fi.type() == NzaeDataTypes::NZUDSUDX_INT32){
                ar[i] = (int32_t)(NzaeInt32Field&)fi;
            }
        }
        validateRuntime(&api, ar, NzaeRuntime::NZAE_ADAPTER_UDTF);

        std::string str = "done";

        NzaeStringField &f =
        (NzaeStringField&)result.get(0); f = str;
    }

};
static int run(NzaeFunction *aeFunc)
{
    aeFunc->run(new MyHandler());
    return 0;
}
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language cpp --template compile
     \ --exe runtimeae --compargs "-g -Wall" --linkargs "-g" runtime.cpp \ --
     version 3
```

Registration

Register the example using the **--mem** and **--mask** options. The **--mask** option enables logging for DEBUG and the **--mem** option sets the memory runtime information.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae \
```

```
--sig "runtimeae(int8, int8, int8,int8,int8,int8)"
\ --return "table(output varchar(100))" \
--language cpp --template udtf --exe runtimeae --version 3
\ --mem 123 --mask debug
```

Running

Before running, enable logging by running **nzudxdbg**:

```
nzudxdbg
Processing 1 spus
.
done
Processing host
done
```

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(runtimeae(1,1,1,1,1,1));
 OUTPUT
--------
 done
(1 row)
```

The NPS system logging appears in the window where the NPS system was started:

```
03-19-10 09:43:59 (dbos.3020) [d,udx ] test fails: 2
03-19-10 09:43:59 (dbos.3020) [d,udx ] test fails: 3
03-19-10 09:43:59 (dbos.3020) [d,udx ] test fails: 4
```

This is the expected output since the arguments specified did not match the runtime in those cases.

Java Language Logging and Runtime Information

This example uses the following file name:

```
TestJavaRuntime.java
```

Code

The code in this example explores functionality that may not be commonly used. Logging can be used to help trace the execution of an AE. To be of use, the AE must be registered with a log mask and logging must be enabled via **nzudxdbg**. To receive system logging messages in the window where the NPS system was started, you need to start NPS with the '-i' option (for example, 'nzstart -i'). This causes the NPS processes to remain attached to the terminal. Runtime information provides statistics about the NPS system, including the number of dataslices, the number of SPUs, and locus of execution.

```
import org.netezza.ae.*;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class TestJavaRuntime {

    private static final Executor exec =
        Executors.newCachedThreadPool();

    public static final void main(String [] args) {
        try {
```

```
        mainImpl(args);
    } catch (Throwable t) {
        System.err.println(t.toString());
        NzaeUtil.logException(t, "main");
    }
}

public static final void mainImpl(String [] args) {

    NzaeApiGenerator helper = new NzaeApiGenerator();

    while (true) {
        final NzaeApi api = helper.getApi(NzaeApi.FUNCTION);
        if (api.apiType == NzaeApi.FUNCTION) {
            if (!helper.isRemote()) {
                run(api.aeFunction);
                break;
            } else {
                Runnable task = new Runnable() {
                        public void run() {
                            try {
                                TestJavaRuntime.run(api.aeFunction);
                            } finally {
                                api.aeFunction.close();
                            }
                        }
                    };
                exec.execute(task);
            }
        }
    }
    helper.close();
}

public static void validateRuntime(Nzae aeFunc, long[] args, int type)
{
       argss:
       dataslice, transaction, hardware, numslices, numspus, locus
    long dataslice = args[0];
    long txid = args[1];
    long hwid = args[2];
    long nslices = args[3];
    long nspus = args[4];
    long locus = args[5];
    NzaeRuntime aeRun =

    aeFunc.getRuntime(); String buf;

    if (locus != aeRun.getLocus())
        {
            buf = "test fails: 1\n";
            aeFunc.log(Nzae.LOG_DEBUG, buf);
        }
    if (aeRun.getSuggestedMemoryLimit() != 123)
        {
            buf = "test fails: 7 \n";
            aeFunc.log(Nzae.LOG_DEBUG, buf);
        }
    if (aeRun.getUserQuery() == false)
        {
            buf ="test fails: 8 \n";
            aeFunc.log(Nzae.LOG_DEBUG, buf);
        }
   if (type != aeRun.getAdapterType())
        {
            buf = "test fails: 9 \n";
            aeFunc.log(Nzae.LOG_DEBUG, buf);
        }
    if (locus == NzaeRuntime.NZAE_LOCUS_POSTGRES) {
        return;
```

```
            }
            if (dataslice != aeRun.getDataSliceId())
                {
                    buf = "test fails: 2 \n";
                    aeFunc.log(Nzae.LOG_DEBUG, buf);
                }
            if ((txid != 0) && (txid != aeRun.getTransactionId()))
                {
                    buf ="test fails: 3 \n";
                    aeFunc.log(Nzae.LOG_DEBUG, buf);

                }
            if (hwid != aeRun.getHardwareId())
                {
                    buf = "test fails: 4 \n";
                    aeFunc.log(Nzae.LOG_DEBUG, buf);
                }
            if (nslices != aeRun.getNumberDataSlices())
                {
                    buf = "test fails: 5 \n";
                    aeFunc.log(Nzae.LOG_DEBUG, buf);
                }
            if (nspus != aeRun.getNumberSpus())
                {
                    buf = "test fails: 6 \n";
                    aeFunc.log(Nzae.LOG_DEBUG, buf);
                }
        }

        public static class MyHandler implements NzaeMessageHandler
        {
            public void evaluate(Nzae ae, NzaeRecord input, NzaeRecord output)
                { final NzaeMetadata meta = ae.getMetadata();
                long ar[] = new long[6];
                for (int i=0; i < 6; i++)
                    {
                        Object o = input.getField(i);
                        NzaeFieldInfo fi = input.getFieldInfo(i);

                        if (fi.getNzType() ==
                            NzaeDataTypes.NZUDSUDX_INT64){ ar[i] = (Long)o;
                        }
                        else if (fi.getNzType() ==
                            NzaeDataTypes.NZUDSUDX_INT32){ ar[i] = (Integer)o;
                        }
                    }
            validateRuntime(ae, ar,
            NzaeRuntime.NZAE_ADAPTER_UDTF); String str = "done";
            output.setField(0,str);
            }
        }

        public static int run(Nzae ae)
        {
            ae.run(new MyHandler());
            return 0;
        }
    }
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java --template compile
     \ TestJavaRuntime.java --version 3
```

Registration

Register the example using the **--mem** and **--mask** options. The **--mask** option enables logging for DEBUG and the **--mem** option sets the memory runtime information.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae \
     --sig "runtimeae(int8,int8,int8,int8,int8,int8)" \
     --return "table(val varchar(1000))" --class AeUdtf --language java
     \ --template udtf --version 3 --define "java_class=TestJavaRuntime"
     \ --mem 123 --mask debug
```

Running

Before running, enable logging by running **nzudxdbg**:

```
nzudxdbg
Processing 1 spus
.
done
Processing host
done
```

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(runtimeae(1,1,1,1,1,1));
 VAL
------
 done
(1 row)
```

The NPS system logging appears in the window where the NPS system was started:

```
04-16-10 12:27:27 (dbos.19600) [d,udx ] test fails: 2
04-16-10 12:27:27 (dbos.19600) [d,udx ] test fails: 3
04-16-10 12:27:27 (dbos.19600) [d,udx ] test fails: 4
```

This is the expected output since the arguments specified did not match the runtime in those cases.

Fortran Language Logging and Runtime Information

This example uses the following file name:

```
logrun.f
```

Code

The code in this example explores functionality that may not be commonly used. Logging can be used to help trace the execution of an AE. To be of use, the AE must be registered with a log mask and logging must be enabled via **nzudxdbg**. To receive system logging messages in the window where the NPS system was started, you need to start NPS with the '-i' option (for example, 'nzstart -i'). This causes the NPS processes to remain attached to the terminal. Runtime information provides statistics about the NPS system, including the number of dataslices, the number of SPUs, and locus of execution.

```
program logRunUdtf
call nzaeRun()
stop
end
```

```
subroutine nzaeHandleRequest(handle)
character(100)    infoString
integer           infoInt, isSpu, isUdf, isUda, isUdtf
double precision  infoDouble
infoString   = "init"
infoDouble   = -1
infoInt      = -1
isSpu        = -1
isUda        = -1
isUdf        = -1
isUdtf       = -1
IGNORE ALL INPUT.
call nzaeGetNext(handle, hasNext)
if (hasNext .eq. 1) then
      goto 10
endif

OUTPUT WHERE WE ARE RUNNING.
call nzaeSetOutputString(handle, 0, "Host or Spu?")
call nzaeIsRunningOnSpu(handle, isSpu)
if (isSpu .eq. 1) then
      call nzaeSetOutputString(handle, 1, "host")
else
      call nzaeSetOutputString(handle, 1, "spu")
endif
call nzaeSetOutputNull(handle, 2)
call nzaeSetOutputNull(handle, 3)
call nzaeOutputResult(handle)
OUTPUT SUGGESTED MEMORY LIMIT.
call nzaeGetSuggestedMemoryLimit(handle, infoDouble)
call nzaeSetOutputString(handle, 0, "Memory Limit")
call nzaeSetOutputNull(handle, 1)
call nzaeSetOutputDouble(handle, 2, infoDouble)
call nzaeSetOutputNull(handle, 3)
call nzaeOutputResult(handle)

OUTPUT IS-USER-QUERY.
call nzaeIsUserQuery(handle, infoInt)
call nzaeSetOutputString(handle, 0, "User
Query?") if (infoInt .eq. 1) then
      call nzaeSetOutputString(handle, 1, "Yes")
else
      call nzaeSetOutputString(handle, 1, "No")
endif
call nzaeSetOutputNull(handle, 2)
call nzaeSetOutputNull(handle, 3)
call nzaeOutputResult(handle)
OUTPUT ADAPTER TYPE.
call nzaeIsUdf(handle, isUdf)
call nzaeIsUda(handle, isUda)
call nzaeIsUdtf(handle, isUdtf)
call nzaeSetOutputString(handle, 0, "Adapter Type")
if (isUdf .eq. 1) then
      call nzaeSetOutputString(handle, 1, "udf")
endif
if (isUda .eq. 1) then
      call nzaeSetOutputString(handle, 1, "uda")
endif
if (isUdtf .eq. 1) then
      call nzaeSetOutputString(handle, 1, "udtf")
endif
call nzaeSetOutputNull(handle, 2)
call nzaeSetOutputNull(handle, 3)
call nzaeOutputResult(handle)
```

```
OUTPUT DATA SLICE ID.
call nzaeGetDataSliceId(handle, infoInt)
call nzaeSetOutputString(handle, 0, "Dataslice ID")
call nzaeSetOutputNull(handle, 1)
call nzaeSetOutputNull(handle, 2)
call nzaeSetOutputInt32(handle, 3, infoInt)

call nzaeOutputResult(handle)

OUTPUT TRANSACTION ID.
call nzaeGetTransactionId(handle, infoString)
call nzaeSetOutputString(handle, 0, "Transaction ID")
call nzaeSetOutputString(handle, 1, infoString) call
nzaeSetOutputNull(handle, 2)
call nzaeSetOutputNull(handle, 3)

OUTPUT HARDWARE ID.
call nzaeGetHardwareId(handle, infoString)
call nzaeSetOutputString(handle, 0, "Hardware
ID") call nzaeSetOutputString(handle, 1,
infoString) call nzaeSetOutputNull(handle, 2)
call nzaeSetOutputNull(handle, 3)

call nzaeOutputResult(handle)

OUTPUT NUMBER OF DATASLICES.
call nzaeGetNumberOfDataslices(handle, infoInt)
call nzaeSetOutputString(handle, 0, "Number of Dataslices")
call nzaeSetOutputNull(handle, 1)
call nzaeSetOutputNull(handle, 2)
call nzaeSetOutputInt32(handle, 3, infoInt)

call nzaeOutputResult(handle)

OUTPUT NUMBER OF SPUS.
call nzaeGetNumberOfSpus(handle, infoInt)
call nzaeSetOutputString(handle, 0, "Number of Spus")
call nzaeSetOutputNull(handle, 1)
call nzaeSetOutputNull(handle, 2)
call nzaeSetOutputInt32(handle, 3, infoInt)

call nzaeOutputResult(handle)

DO SOME LOGGING.
call nzaeLog(handle, "Here is some logging text!")

call nzaeLog(handle, "Here is some more logging

text!") return

end
```

Compilation

Use the standard compile:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language fortran --version 3

     \ --template compile logrun.f
```

Registration

Register the example using the **--mem** and **--mask** options. The **--mask** option enables logging for DEBUG and the **--mem** option sets the memory runtime information.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language fortran --version 3 \
     --template udtf --exe logrun --sig "log_run(int4)" \
     --return "table(name varchar(100), val_str varchar(100), \
     val_double double, val_int int4)" --mem 100k --mask debug
```

Running

Before running, enable logging by running **nzudxdbg**:

```
nzudxdbg
Processing 1 spus
.
done
Processing host
done
```

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(log_run(1));
         NAME          | VAL_STR | VAL_DOUBLE | VAL_INT
                       +---------+------------+---------
--------------------HostORSpu?| spu     |            |
 Memory LIMIT          |         |     102400 |
 User Query?           | Yes     |            |
 Adapter Type          | udtf    |            |
 Dataslice ID          |         |            |         0
 Hardware ID           | 0       |            |
 Number of Dataslices  |         |            |         4
 Number of Spus        |         |            |         1
```

The NPS system logging appears in the window where the NPS system was started:

```
05-05-10 14:33:18 (dbos.18728) [d,udx ] Here is some logging text!
05-05-10 14:33:18 (dbos.18728) [d,udx ] Here is some more logging
text!
```

Python Language Logging and Runtime Information

This example uses the following file name:

```
logrun.py
```

Code

The code in this example explores functionality that may not be commonly used. Logging can be used to help trace the execution of an AE. To be of use, the AE must be registered with a log mask and logging must be enabled via **nzudxdbg**. To receive system logging messages in the window where the NPS system was started, you need to start NPS with the '-i' option (for example, 'nzstart -i'). This causes the NPS processes to remain attached to the terminal. Runtime information provides statistics about the NPS system, including the number of dataslices, the number of SPUs, and locus of execution.

```
import nzae

class LogRunAe(nzae.Ae):

    def _run(self):
        for row in self:
            pass
        self.output("Dataslice ID:           " + repr(self.getDatasliceId()))
        self.output("Hardware ID:            " + repr(self.getHardwareId()))
        self.output("Number of Data Slices:  " +
repr(self.getNumberOfDataSlices()))
        self.output("Number of Spus:         " + repr(self.getNumberOfSpus()))
        self.output("Suggested Memory Limit: " +
```

```
repr(self.getSuggestedMemoryLimit()))
        self.output("Transaction ID:         " + repr(self.getTransactionId()))
        self.output("Is A User Query:        " + repr(self.isAUserQuery()))
        self.output("Is Logging Enabled:     " + repr(self.isLoggingEnabled()))
        self.output("Is Running on a spu:    " + repr(self.isRunningOnSpu()))
        self.output("Is Running in Postgres: " +
repr(self.isRunningInPostgres()))
        self.output("Is Running in DBOS:     " + repr(self.isRunningInDbos()))
        self.output("Is a UDA:               " + repr(self.isUda()))
        self.output("Is a UDF:               " + repr(self.isUdf()))
        self.output("Is a UDTF:              " + repr(self.isUdtf()))
        self.log("Here is some logging text!",  self.LOG_LEVEL__DEBUG)
        self.log("Here is some more logging text!", self.LOG_LEVEL__DEBUG)

LogRunAe.run()
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language python64
     \ --template deploy ./logrun.py --version 3
```

Registration

Register the example using the **--mem** and **--mask** options. The **--mask** option enables logging for DEBUG and the **--mem** option sets the memory runtime information.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language python64 --version 3 \
     --template udtf --exe logrun.py -- sig "log_run(int4)" \
     --return "table(text varchar(1000))" -- mask debug --mem 100k
```

Running

Before running, enable logging by running **nzudxdbg**:

```
nzudxdbg
Processing 1 spus
.
done
Processing host
done
```

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(log_run(1));
             TEXT
------------------------------
 Dataslice ID:           0
 Hardware ID:            0
 Number of DATA Slices:  4
 Number of Spus:         1
 Suggested Memory LIMIT: 102400
 Transaction ID:         8520
 IS A User Query:        True
 IS Logging Enabled:     True
 IS Running ON a spu:    False
 IS Running IN Postgres: False
 IS Running IN DBOS:     True
 IS a UDA:               False
 IS a UDF:               False
 IS a UDTF:              True
(14 rows)
```

The NPS system logging appears in the window where the NPS system was started:

```
05-05-10 14:33:18 (dbos.18728) [d,udx ] Here is some logging text!
05-05-10 14:33:18 (dbos.18728) [d,udx ] Here is some more logging text
```

Perl Language Logging and Runtime Information

This example uses the following file name:

```
LogRunAe.pm
```

Code

The code in this example explores functionality that may not be commonly used. Logging can be used to help trace the execution of an AE. To be of use, the AE must be registered with a log mask and logging must be enabled via **nzudxdbg**. To receive system logging messages in the window where the NPS system was started, you need to start NPS with the '-i' option (for example, 'nzstart -i'). This causes the NPS processes to remain attached to the terminal. Runtime information provides statistics about the NPS system, including the number of dataslices, the number of SPUs, and locus of execution.

```
package LogRunAe;
use strict;
use autodie;
use nzae::Ae;

our @ISA = qw(nzae::Ae);


my $ae = LogRunAe->new();
$ae->run();

sub _run()
{
    my $self = shift;
    while ($self->getNext())
    {
        next;
    }

    $self->output("Dataslice ID: ".$self->getDatasliceId());
    $self->output("Hardware ID: ".$self->getHardwareId());
    $self->output("Number of Data Slices: ".$self->getNumberOfDataSlices());
    $self->output("Number of Spus: ".$self->getNumberOfSpus());
    $self->output("Suggested Memory Limit: ".$self->getSuggestedMemoryLimit());
    $self->output("Transaction ID: ".$self->getTransactionId());

    my $userquery = $self->isAUserQuery()? "false" : "true";
    $self->output("Is A User Query: ".$userquery);

    my $loggingenabled = $self->isLoggingEnabled()? "true":"false";

    $self->output("Is Logging Enabled: ".$loggingenabled);

    my $runningonspu = $self->isRunningOnSpu()? "true" : "false";
    $self->output("Is Running on a spu: ".$runningonspu);

    my $runninginpostgres = $self->isRunningInPostgres()? "true" : "false";
    $self->output("Is Running in Postgres: ".$runninginpostgres);

    my $runningindbos = $self->isRunningInDbos()? "true" : "false";
    $self->output("Is Running in DBOS: ".$runningindbos);
```

```
    my $uda = $self->isUda()? "true" : "false";
    $self->output("Is a UDA: ".$uda);

    my $udf = $self->isUdf()? "true" : "false";
    $self->output("Is a UDF: ".$udf);

    my $udtf = $self->isUdtf()? "true" : "false";
    $self->output("Is a UDTF: ".$udtf);

    $self->log("Here is some logging text!", $self->getLogLevelDebug());
    $self->log("Here is some more logging text!", $self->getLogLevelDebug());
}

1;
```

Deployment

Deploy the script:

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language perl --version 3
     \ --template deploy LogRunAe.pm
```

Registration

Register the example using the **--mem** and **--mask** options. The **--mask** option enables logging for DEBUG and the **--mem** option sets the memory runtime information.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language perl --version 3 \
     --template udtf --exe LogRunAe.pm --sig "log_run(int4)" \
     --return "table(text varchar(1000))" --level 4 --mask debug --mem 100k
```

Running

Before running, enable logging by running **nzudxdbg**:

```
nzudxdbg
Processing 1 spus
.
done
Processing host
done
```

Run the query in nzsql on the system database:

```
SELECT * FROM TABLE WITH FINAL(log_run(1));
             TEXT
--------------------------------
 Dataslice ID: 0
 Hardware ID: 0
 Number of Data Slices: 4
 Number of Spus: 1
 Suggested Memory Limit: 102400
 Transaction ID: 304796
 Is A User Query: false
 Is Logging Enabled: true
 Is Running on a spu: false
 Is Running in Postgres: false
 Is Running in DBOS: true
 Is a UDA: false
 Is a UDF: false
 Is a UDTF: true
(14 rows)
```

The NPS system logging appears in the window where the NPS system was started:

```
05-05-10 14:33:18 (dbos.18728) [d,udx ] Here is some logging text!
05-05-10 14:33:18 (dbos.18728) [d,udx ] Here is some more logging text
```

R Language Logging and Runtime Information

This example explores logging functionality that might not be commonly used.

Code

Logging can help you trace the execution of an AE. The AE must be registered with a log mask, and logging must be enabled thtough nzudxdbg. Runtime information provides statistics about the NPS system including the number of data slices, S-Blades, and locus of execution.

Enter the following code in the */tmp/logrun.R* file:

```
nz.fun <- function(){
  getNext()
  logMessage(NZ.DEBUG, "before getRuntime()")
  r <- getRuntime()
  apply(cbind(names(r),as.character(r)), 1, function(X){
    setOutput(0, X[1])
    setOutput(1, X[2])
    outputResult()
  })
  logMessage(NZ.DEBUG, "after getRuntime()")
}
```

Compilation

Compile the code:

```
/nz/export/ae/utilities/bin/compile_ae --language r --template compile
  \ --version 3 --db dev --user nz /tmp/logrun.R
```

Registration

Register the example using the --**mask** option. The **–mask** option enables logging for DEBUG and the **--mem** option sets the memory runtime information.

```
/nz/export/ae/utilities/bin/register_ae --language r --version 3 \
  --template udtf -- exe logrun.R --sig "logrun(VARARGS)"         \
  --return "TABLE(name VARCHAR(1000), value VARCHAR(1000))"       \
  --db dev --user nz --level 4 -- mask DEBUG
```

Running

Before running, enable logging by running **nzudxdbg**:

```
>nzudxdbg
Processing 1 spus
.
done
Processing host
done
```

Run the query in nzsql:

```
SELECT * FROM TABLE WITH FINAL(logrun());
           NAME           | VALUE
                          +-------
----------------------data.slice.id| 0
 transaction.id           | 21814
 hardware.id              | 0
 number.data.slices       | 4
 number.spus              | 1
 suggested.memory.limit   | 0
 locus                    | 1
 adapter.type             | 1
 user.query               | 1
 session.id               | 16231
(10 rows)
```

The NPS system logging appears in the window where the NPS system was started:

```
04-06-11 09:48:54 (dbos.14577) [d,udx ] before getRuntime()
04-06-11 09:48:54 (dbos.14577) [d,udx ] after getRuntime()
```

CHAPTER 19
Integrated Examples

Java Language Integrated Example

This section contains a series of related examples tying a number of techniques together.

Note: Compilation and registration examples that specify the **--db** option assume that there is a valid database called “dev” on the Netezza appliance.

Concepts

Concepts covered in this section include:

- Combining different source files together to create more complex AE applications and facilitate code reuse.
- Using Java JAR files.
- Calling the same AE from different SQL Functions and choosing different logic paths based on environment variables.
- Execution time streaming of output from one AE into another.
- Table functions that return:
  - exactly one output row after reading all input rows
  - one output row per input row
  - multiple output rows per input row
- Demonstration of the lower level AE connection API compared to using NzaeApiGenerator.

Test Data

This table data is used for all the examples in this section.

```
/* edutestdata.sql */
DROP TABLE edutestdata;
```

```
CREATE TABLE edutestdata (
   distribution_key int4,
    f1 int4,
    f2 int8,
    f3 double,
    f4 numeric(10,3),
   color varchar(32)
) distribute ON (distribution_key);

INSERT INTO edutestdata VALUES(1,     1,    2, 3.0, 4.0, 'red');
INSERT INTO edutestdata VALUES(2,   100,  300, 0.5, 0.5, 'red');
INSERT INTO edutestdata VALUES(3, -100,   300, 0.5, 0.5, 'red');

INSERT INTO edutestdata VALUES(4,   100, -300, 0.5, 0.5, 'red');

INSERT INTO edutestdata VALUES(1,     1,    2, 3.0, 4.0, 'green');
INSERT INTO edutestdata VALUES(2,   100,  300, 0.5, 0.5, 'green');
INSERT INTO edutestdata VALUES(3, -100,   300, 0.5, 0.5, 'green');

INSERT INTO edutestdata VALUES(4,   100, -300, 0.5, 0.5, 'green');

INSERT INTO edutestdata VALUES(1,     1,    2, 3.0, 4.0, 'blue');
INSERT INTO edutestdata VALUES(2,   100,  300, 0.5, 0.5, 'blue');
INSERT INTO edutestdata VALUES(3, -100,   300, 0.5, 0.5, 'blue');

INSERT INTO edutestdata VALUES(4,   100, -300, 0.5, 0.5, 'blue');

INSERT INTO edutestdata VALUES(1,     1,    2, 3.0, 4.0, 'yellow');
INSERT INTO edutestdata VALUES(2,   100,  300, 0.5, 0.5, 'yellow');
INSERT INTO edutestdata VALUES(3, -100,   300, 0.5, 0.5, 'yellow');

INSERT INTO edutestdata VALUES(4,   100, -300, 0.5, 0.5, 'yellow');
```

ApplyOperation

Code in this section is saved in a file called ApplyOperation.java.

Concepts

This is a variation of the “ApplyOp” example, with the following modifications:

The data-handling logic has been separated from the connection creation logic. The standalone class is reused in multiple AE example applications that use differing connection logic.

This AE uses the file-IO paradigm instead of the call-back paradigm to retrieve the input rows. For an example of the call-back paradigm, see MyHandler in Java Language Scalar Function.

The code supports the subtraction operator.

Note that because this function AE writes exactly one output row per input row and the output row contains exactly one column, the AE can be invoked from either a scalar or table Function.

Code

```
  ApplyOperation.java
  Function AE
  Performs arithmetic operations (+, -, or *)
     across columns in a row
  Ignores non-numeric columns
  First argument column must be a string type operation
  Handles a variable number of input columns and data types
  Returns a double result
  Can be called as a table function or a scalar
function package org.netezza.education;

import java.io.*;
import java.math.BigDecimal;
```

```
import java.text.DateFormat;
import java.sql.Date;
import java.sql.Time;
import java.sql.Timestamp;
import org.netezza.ae.*;

public class ApplyOperation {
    private static final int OP_ADD = 1;
    private static final int OP_MULT = 2;
    private static final int OP_SUBTRACT = 3;

    public static void runAe(Nzae ae) {
        final NzaeMetadata meta = ae.getMetadata();

        for (;;) {
            int op = 0;
            double result = 0;
            NzaeRecord input = ae.next();
            if (input == null) {
                break;
            }
            NzaeRecord output = ae.createOutputRecord();

            for (int i = 0; i < input.size(); i++) {
                if (input.getField(i) == null) {
                    continue;
                }

                int dataType = meta.getInputNzType(i);

                if (i == 0) {
                    if (!(dataType == NzaeDataTypes.NZUDSUDX_FIXED
                            || dataType ==
                                NzaeDataTypes.NZUDSUDX_VARIABLE)) {
                        ae.userError(
                            "first column must be a string");
                    }
                    String opStr = input.getFieldAsString(0);
                    if (opStr.equals("*")) {
                        result = 1;
                        op = OP_MULT;
                    } else if (opStr.equals("+"))
                        { result = 0;
                        op = OP_ADD;
                    } else if (opStr.equals("-"))
                        { result = 0;
                        op = OP_SUBTRACT;
                    } else {
                        ae.userError("invalid operator = " + op);
                    }
                    continue;
                }

                switch (dataType) {
                case NzaeDataTypes.NZUDSUDX_INT8:
                case NzaeDataTypes.NZUDSUDX_INT16:
                case NzaeDataTypes.NZUDSUDX_INT32:
                case NzaeDataTypes.NZUDSUDX_INT64:
                case NzaeDataTypes.NZUDSUDX_FLOAT:
                case NzaeDataTypes.NZUDSUDX_DOUBLE:
                case NzaeDataTypes.NZUDSUDX_NUMERIC32:
                case NzaeDataTypes.NZUDSUDX_NUMERIC64:
                case NzaeDataTypes.NZUDSUDX_NUMERIC128:
                    switch (op) {
                    case OP_ADD:
                        result +=
                          input.getFieldAsNumber(i).doubleValue();
                        break;
```

```
                        case OP_SUBTRACT:
                            result -=
                              input.getFieldAsNumber(i).doubleValue();
                            break;
                        case OP_MULT:
                            result *=
                              input.getFieldAsNumber(i).doubleValue();
                            break;
                        default:
                            break;
                        }
                        break;

                    default:
                        break;
                    }
                }   // end of column for loop

                output.setField(0, result);
                ae.outputResult(output);
             }

            ae.done();
        }
    }
```

ApplyDriver Version 1

Code in this section is saved in a file called ApplyDriverV1.java.

Code

Get a Local AE connection handle of type “function.” It can be called from a table or scalar function.

```
// ApplyDriverV1.java
package org.netezza.education;

import org.netezza.ae.*;

public class ApplyDriverV1 {
    public static final void main(String [] args) { NzaeFactory

        factory = NzaeFactory.Source.getFactory();

        if (factory.isLocal()) {
            NzaeInitialization init = new NzaeInitialization();
            Nzae ae = factory.getLocalFunctionApi(init);
            ApplyOperation.runAe(ae);
            ae.close();
        } else {
            throw new RuntimeException("Expecting Local AE only");
        }
    }
}
```

Compilation

Two Java sources are compiled and deployed as class files.

```
# mod1comp1.bsh
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java --template compile
--version 3 \
--db dev "ApplyOperation.java ApplyDriverV1.java"
```

Registration as a Scalar Function

The **--template udf** parameter indicates a scalar function.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udf --version 3 --db dev \
--sig "applyOperationV1Sf(varargs)" --return double \
--define java_class=org.netezza.education.ApplyDriverV1 --level 1
```

Running as a Scalar Function

```
SELECT applyOperationV1Sf('+', 1, 2, 3);
 APPLYOPERATIONV1SF
--------------------
                  6

SELECT f1, f2, f3, f4, '' AS equals, applyOperationV1Sf('+', f1, f2, f3, f4)
     FROM edutestdata WHERE color = 'red';
  F1  |  F2  | F3  |  F4   | EQUALS | APPLYOPERATIONV1SF
------+------+-----+-------+--------+--------------------
    1 |    2 |   3 | 4.000 |        |                  10
 -100 |  300 | 0.5 | 0.500 |        |                 201
  100 | -300 | 0.5 | 0.500 |        |                -199
  100 |  300 | 0.5 | 0.500 |        |                 401
```

Registration as a Table Function

The **--template udtf** parameter indicates a table function.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae -- language java --template udtf
--version 3 --db dev \
     --level 1 --sig "applyOperationV1Tf(varargs)" \
     --return "table(result double)" \
     --define java_class=org.netezza.education.ApplyDriverV1
```

Running as a Table Function

```
SELECT * FROM TABLE WITH FINAL(applyOperationV1Tf('+', 1, 2, 3));
 RESULT
--------
      6

SELECT f1, f2, f3, f4, '' AS equals, result FROM edutestdata,
     TABLE WITH FINAL(applyOperationV1Tf('+', f1, f2, f3, f4))
     WHERE color = 'red';
   F1 |  F2  | F3  |  F4   | EQUALS | RESULT
------+------+-----+-------+--------+--------
  100 | -300 | 0.5 | 0.500 |        |   -199
    1 |    2 |   3 | 4.000 |        |     10
  100 |  300 | 0.5 | 0.500 |        |    401
```

ApplyResult

Code in this section is saved in a file called ApplyResult.java.

Concepts

ApplyResult takes exactly two arguments: a string operator and a double operand. The previous class, ApplyOp, applies an operation to the numeric columns in a row. In contrast, ApplyResult applies an operation to a single column in multiple rows. The two can be used together by streaming the output of ApplyOp into ApplyResult. They could be used together to, for example, add all the numeric columns in a table.

This AE can only be invoked as a table function. After all the input rows are read, a calculation is performed and a single result returned.

Code

```
   ApplyResult.java
   Function AE
   Performs arithmetic operations on an aggregate of rows
      sum, product, max, min, average
   Input is an operation string argument and a double argument
   Returns a double result
   Must be called as a table function
package org.netezza.education;

import org.netezza.ae.*;

public class ApplyResult {
    private static final int RESULT_SUM = 1;
    private static final int RESULT_PRODUCT = 2;
    private static final int RESULT_MAX = 3;
    private static final int RESULT_MIN = 4;
    private static final int RESULT_AVERAGE = 5;

    public static void runAe(Nzae ae) {
        try {
            runAeImpl(ae);
        } catch (Throwable t) {
            t.printStackTrace()
            ;
            try { ae.userError(t.getMessage()); }
            catch (Throwable t2) {}
        }
    }

    private static void runAeImpl(Nzae ae) {
        final NzaeMetadata meta = ae.getMetadata();

        if (meta.getInputColumnCount() != 2) {
            ae.userError("two arguments exepected");
            return;
        }

        int dataType = meta.getInputNzType(0);
        if (!(dataType == NzaeDataTypes.NZUDSUDX_FIXED ||
                dataType ==
                    NzaeDataTypes.NZUDSUDX_VARIABLE ||
                dataType ==
                    NzaeDataTypes.NZUDSUDX_NATIONAL_FIXED ||
                dataType ==
                    NzaeDataTypes.NZUDSUDX_NATIONAL_VARIABLE)) {
            String msg = "first column expected to be a string";
            System.err.println(msg);
            ae.userError(msg);
            return;
        }

        dataType = meta.getInputNzType(1);
```

```
if (dataType != NzaeDataTypes.NZUDSUDX_DOUBLE)
{
    String msg = "second column expected to be a double";
    System.err.println(msg);
    ae.userError(msg);
    return;
}

int op = 0; // operation to perform on input data
double result = 0; // place to store running result
int count = 0; // count of input records

   input row loop
for (;;) {
    NzaeRecord input = ae.next();
    if (input == null) {
        break;
    }
    ++count;
    String opStr = input.getFieldAsString(0);

       retrieve the operation exeactly
    once if (op == 0) {
        if (opStr.equalsIgnoreCase("sum")
                  opStr.equals("+"))
            { op = RESULT_SUM;
        } else if (opStr.equalsIgnoreCase("product")
                  opStr.equals("+")) {
            result = 1;
            op = RESULT_PRODUCT;
        } else if (opStr.equalsIgnoreCase("max"))
            { op = RESULT_MAX;
        } else if (opStr.equalsIgnoreCase("min"))
            { op = RESULT_MIN;
        } else if (opStr.equalsIgnoreCase("average"))
            { op = RESULT_AVERAGE;
        } else {
            String msg = "unexpected operation";
            System.err.println(msg);
            ae.userError(msg);
            return;
        }
    }

    double inputValue =
        input.getFieldAsNumber(1).doubleValue();

    switch (op) {
    case RESULT_SUM:
    case RESULT_AVERAGE:
        result += inputValue;
        break;
    case RESULT_PRODUCT:
        result *= inputValue;
        break;
    case RESULT_MAX:
        if (count == 1 || inputValue > result) {
            result = inputValue;
        }
        break;
    case RESULT_MIN:
        if (count == 1 || inputValue < result) {
            result = inputValue;
        }
        break;
    default:
        String msg =
            "logic error in operation case statement";
```

```
                System.err.println(msg);
                ae.userError(msg);
                return;
            }
        }   // end of for input loop

           output the result
        switch (op) {
        case RESULT_AVERAGE:
            result /= (double)
        count; default:
            NzaeRecord output =
            ae.createOutputRecord(); output.setField(0,
            result); ae.outputResult(output);
        }

           it's critical that done get called
        ae.done();
    }
}
```

ApplyUtil

Code in this section is saved in a file called ApplyUtil.java.

Concepts

This class contains a collection of utility methods. These utilities can provide:

metadata about a particular query

runtime information about the AE process

information about shared libraries

Information about AE environment variables

Code

```
   ApplyUtil.java
   Useful collection of AE Utilty

Functions package org.netezza.education;

import org.netezza.ae.*;

public class ApplyUtil {
    /** print Function AE metadata to standard output */
    public static void printMetadata(Nzae ae) {
        // print input metadata
        final NzaeMetadata meta = ae.getMetadata();
        String indent = " " + " " + " " + " ";

        System.out.println("*** Input (Argument) Metadata ***");
        int numInputColumns = meta.getInputColumnCount();
        System.out.println("Number of input columns: "
            + numInputColumns);
        for (int i = 0; i < numInputColumns; i++) {
            System.out.println(
                "column " + i + ": " +
                    NzaeDataTypes.nzTypeToString(
                        meta.getInputNzType(i)));
            switch (meta.getInputNzType(i)) {
            case NzaeDataTypes.NZUDSUDX_FIXED:
            case NzaeDataTypes.NZUDSUDX_VARIABLE:
            case NzaeDataTypes.NZUDSUDX_NATIONAL_FIXED:
```

```
            case NzaeDataTypes.NZUDSUDX_NATIONAL_VARIABLE:
                System.out.println(indent + "length="
                      meta.getInputSize(i));
                break;
            case NzaeDataTypes.NZUDSUDX_NUMERIC32:
            case NzaeDataTypes.NZUDSUDX_NUMERIC64:
            case NzaeDataTypes.NZUDSUDX_NUMERIC128:
                System.out.println(indent + "size="
                      meta.getInputSize(i)
                      ", scale=" + meta.getInputScale(i));
                break;
            default:
                break;
            }
        }

           print output metadata
        System.out.println();
        System.out.println("*** Output (Result) Metadata ***");

        int numOutputColumns = meta.getOutputColumnCount();
        System.out.println("Number of output columns: "
            + numOutputColumns);
        for (int i = 0; i < numOutputColumns; i++) {
            System.out.println(
                "column " + i + ": " +
                    NzaeDataTypes.nzTypeToString(
                        meta.getOutputNzType(i)));
            switch (meta.getOutputNzType(i)) {
            case NzaeDataTypes.NZUDSUDX_FIXED:
            case NzaeDataTypes.NZUDSUDX_VARIABLE:
            case NzaeDataTypes.NZUDSUDX_NATIONAL_FIXED:
            case NzaeDataTypes.NZUDSUDX_NATIONAL_VARIABLE:
                System.out.println(indent + "length="
                      meta.getOutputSize(i));
                break;
            case NzaeDataTypes.NZUDSUDX_NUMERIC32:
            case NzaeDataTypes.NZUDSUDX_NUMERIC64:
            case NzaeDataTypes.NZUDSUDX_NUMERIC128:
                System.out.println(indent + "size="
                    + meta.getOutputSize(i) +
                    ", scale=" + meta.outputScale(i));
                break;
            default:
                break;
            }
        }
        System.out.println();
    }

    /**  print Shaper AE metadata to standard output */
    public static void printMetadata(NzaeShaper ae) {
        // print input metadata
        final NzaeMetadata meta = ae.getMetadata();
        String indent = " " + " " + " " + " " + " ";

        System.out.println("*** Input (Argument) Metadata ***");
        int numInputColumns = meta.getInputColumnCount();
        System.out.println("Number of input columns: "
            + numInputColumns);
        for (int i = 0; i < numInputColumns; i++) {
            System.out.println(
                "column " + i + ": " +
                    NzaeDataTypes.nzTypeToString(
                        meta.getInputNzType(i)));
            switch (meta.getInputNzType(i)) {
            case NzaeDataTypes.NZUDSUDX_FIXED:
            case NzaeDataTypes.NZUDSUDX_VARIABLE:
```

```
            case NzaeDataTypes.NZUDSUDX_NATIONAL_FIXED:
            case NzaeDataTypes.NZUDSUDX_NATIONAL_VARIABLE:
                System.out.println(indent + "length="
                    + meta.getInputSize(i));
                break;
            case NzaeDataTypes.NZUDSUDX_NUMERIC32:
            case NzaeDataTypes.NZUDSUDX_NUMERIC64:
            case NzaeDataTypes.NZUDSUDX_NUMERIC128:
                System.out.println(indent + "size="
                    + meta.getInputSize(i) +
                    ", scale=" + meta.getInputScale(i));
                break;
            default:
                break;
            }
        }
    }

    /** print runtime information to standard output */
    public static void printRuntime(NzaeBase ae) {
        final NzaeRuntime runtime = ae.getRuntime();
        StringBuilder result = new StringBuilder();
        result.append("*** Runtime Information ***" + sep);
        result.append("sessionId="
              runtime.getSessionId() + sep);
        result.append("dataSliceId="
              runtime.getDataSliceId() +
        sep); result.append("transactionId="
              runtime.getTransactionId() +
        sep); result.append("hardwareId="
              runtime.getHardwareId() + sep);
        result.append("numberDataSlices="
              runtime.getNumberDataSlices() +
        sep); result.append("numberSpus="
              runtime.getNumberSpus() + sep);
        result.append("suggestedMemoryLimit=" +
            runtime.getSuggestedMemoryLimit() +
        sep); result.append("locus=");
        switch (runtime.getLocus()) {
            case NzaeRuntime.NZAE_LOCUS_POSTGRES:
                result.append("NZAE_LOCUS_POSTGRES");
                break;
            case NzaeRuntime.NZAE_LOCUS_DBOS:
                result.append("NZAE_LOCUS_DBOS");
                break;
            case NzaeRuntime.NZAE_LOCUS_SPU:
                result.append("NZAE_LOCUS_SPU");
                break;
            default:
                result.append("?");
        }
        result.append(sep);
        result.append("adapterType="); switch
        (runtime.getAdapterType()) {
            case NzaeRuntime.NZAE_ADAPTER_OTHER:
                result.append("NZAE_ADAPTER_OTHER");
                break;
            case NzaeRuntime.NZAE_ADAPTER_UDTF:
                result.append("NZAE_ADAPTER_UDTF")
                ; break;
            case NzaeRuntime.NZAE_ADAPTER_UDF:
                result.append("NZAE_ADAPTER_UDF");
                break;
            case NzaeRuntime.NZAE_ADAPTER_UDA:
                result.append("NZAE_ADAPTER_UDA");
                break;
            default:
                result.append("?");
```

```
        }
        result.append(sep);
        result.append("userQuery=" +
        runtime.getUserQuery()); System.out.println(result);
    }

    /** print shared library information to standard output */
    public static void printSharedLibraries(NzaeBase ae) {
        NzaeLibrary lib = ae.getLibrary();
        System.out.println("*** Local Libraries ***");
        int count = lib.sizeLocalEntries();
        for (int i = 0; i < count; i++) {
            NzaeLibrary.NzaeLibraryInfo info
                  lib.getLocalLibraryInfo(i);
            System.out.println(info.libraryName + "="
                + info.libraryFullPath
                + " (" + info.autoLoad + ")");
        }
        System.out.println();

        System.out.println("*** Parent Libraries ***");
        count = lib.sizeParentEntries();
        if (count == 0) {
            System.out.println("none");
        }
        for (int i = 0; i < count; i++) {
            NzaeLibrary.NzaeLibraryInfo info =
                lib.getParentLibraryInfo(i);
            System.out.println(info.libraryName + "="
                  info.libraryFullPath
                  " (" + info.autoLoad + ")");
        }
        System.out.println();
    }

    /**  print AE Environment to standard output */
    public static void printEnvironment(NzaeBase ae) {
        System.out.println("*** Environment ***");
        NzaeEnvironment env = ae.getEnvironment();
        String key = env.getFirstKey();
        do {
            System.out.println(key + "=" + env.getValue(key));
            key = env.getNextKey();
        } while (key != null);
    }

    private static final String sep =
        System.getProperty("line.separator");
}
```

ApplyDriver Version 2

Code in this section is saved in a file called ApplyDriverV2.java.

Code

This new version of the application driver uses environment variables to determine which function to call (OPERATION or RESULT). It also uses an environment variable to determine whether to output diagnostic information. This is a local AE only.

```
// ApplyDriverV2.java
package org.netezza.education;

import org.netezza.ae.*;
```

```
public class ApplyDriverV2 {
    public static final void main(String [] args) { NzaeFactory

        factory = NzaeFactory.Source.getFactory();

        if (factory.isLocal()) {
            NzaeInitialization init = new NzaeInitialization();
            Nzae ae = factory.getLocalFunctionApi(init);

            try {
                   determine which function is being invoked
                String whichFunction =
                    ae.getEnvironment().getValue("APPLY_FUNCTION");
                if (whichFunction == null) {
                    ae.userError(
                        "missing APPLY_FUNCTION "
                        + "environment variable");
                    return;
                }
                String printInfo =
                    ae.getEnvironment().getValue("APPLY_INFO");
                if (printInfo != null
                           printInfo.equalsIgnoreCase("true"))
                    { ApplyUtil.printMetadata(ae);
                    ApplyUtil.printRuntime(ae);
                    ApplyUtil.printSharedLibraries(ae);
                    ApplyUtil.printEnvironment(ae);
                }

                if (whichFunction.equals("OPERATION")) {
                    ApplyOperation.runAe(ae);
                } else if (whichFunction.equals("RESULT"))
                    { ApplyResult.runAe(ae);
                } else {
                    ae.userError("unexpected value for " +
                        "environment variable APPLY_FUNCTION");
                }
            } finally {
                ae.close();
            }
        } else {
            throw new RuntimeException("Expecting Local AE only");
        }
    }
}
```

Compilation

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java --template
compile --version 3 --db dev \
"ApplyDriverV2.java ApplyOperation.java ApplyResult.java ApplyUtil.java"
```

Registration as Scalar Function applyOperationV2Sf

This example sets the application-specific APPLY_FUNCTION AE environment variable to the function OPERATION so that the AE applies an operation to columns in a row.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udf --version 3 --db dev \
     --sig "applyOperationV2Sf(varargs)" --return double \
     --define java_class=org.netezza.education.ApplyDriverV2 --level 1
     \ --environment "'APPLY_FUNCTION'='OPERATION'" \
     --environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"
```

Registration as Table Function applyOperationV2Tf

This example also sets the application-specific APPLY_FUNCTION AE environment variable to the

function OPERATION, as above. It uses a table function invocation.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udtf --version 3 --db dev \
     --sig "applyOperationV2Tf(varargs)" --return "table(result double)"
     \ --define java_class=org.netezza.education.ApplyDriverV2 --level 1
     \ --environment "'APPLY_FUNCTION'='OPERATION'" \
     --environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_HOST_ONLY_NZAE_SPIN_FILE_NAME'='/tmp/spin_ae.on'"
     \ --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"
```

Registration as Table Function applyResultV2

This example sets the application-specific APPLY_FUNCTION AE environment variable to the function RESULT so that the AE applies an operation to the second column of each row.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udtf --version 3 --db dev \
     --sig "applyResultV2(varchar(16), double)" \
     --return "table(result double)" \
     --define java_class=org.netezza.education.ApplyDriverV2 --level 1
     \ --noparallel --environment "'APPLY_FUNCTION'='RESULT'" \ --
     environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"
```

Running as Table Function applyResultV2

```
SELECT result FROM edutestdata, TABLE WITH FINAL(applyResultV2('sum', f1))
     WHERE color = 'red';
 RESULT
--------
    101
```

Running as applyOperationV2Tf and applyResultV2

The SQL function applyOperationV2Tf applies the addition operator to the columns in a row and then streams the result to SQL function applyResultV2, which applies the sum operation to the second column in each row.

```
SELECT re.result FROM edutestdata,
     TABLE WITH FINAL(applyOperationV2Tf('+', f1, f2, f3, f4)) op,
     TABLE WITH FINAL(applyResultV2('sum', op.result)) re
     WHERE color = 'red';
 RESULT
--------
    413
```

You can create an alternate version of the ApplyDriverV2 sample, which uses third-party and open source Java code with a Java AE. This alternate version of ApplyDriverV2 uses a JAR file and is deployed to a JAR file. It calls the main method in ApplyDriverV2. Otherwise, the functionality of the AE is the same.

Acquiring the Open Source JAR File

Before you can run this example, you must download and install third-party base64 files. To retrieve and compile the sample open source JAR file, perform the following steps:

Download the Java source zip file from http://iharder.sourceforge.net/current/java/base64/.

Extract the contents into an empty Linux directory.

```
unzip -j Base64-v<version_number>.zip
```

Compile Base64.java:

```
[~/installs/base64]$javac Base64.java
```

To list the contents of the JAR file, enter the following command:

```
[~/installs/base64]$jar tf Base64.jar
META-INF/
META-INF/MANIFEST.MF
Base64$1.class
Base64.class
Base64$InputStream.class
Base64$OutputStream.class
```

Create the JAR file:

```
[~/installs/base64]$jar cf Base64.jar *.class
```

Deploy the newly built JAR file:

```
/nz/export/ae/utilities/bin/compile_ae --language java --version 3 --db dev \
 --template deploy base64.jar
```

The JAR file is now ready to use with a Java AE.

Code for JarApplyDriverV2.java

The following AE uses the newly deployed base64.jar:

```
   JarApplyDriverV2.java
   demonstrates use of 3rd party jar

file package org.netezza.education;

import java.io.IOException;
   from http://iharder.sourceforge.net/current/java/base64/

import sourceforge.iharder.Base64;

public class JarApplyDriverV2 {
    public static final void main(String [] args) {
        try {
            double value = 100;
            String result = Base64.encodeObject(value);
            System.out.println("JarApplyDriverV2: "
                  result);
        } catch (IOException e) {
            System.err.println("JarApplyDriverV2: "
                  e.getMessage());
        }

        ApplyDriverV2.main(args);
    }
}
```

Compile arApplyDriverV2.java

This example shows how to compile using a third-party JAR file with the **--linkargs** parameter and how to compile to a JAR file using the **–exe** parameter. Because the example relies on the third-party JAR **base64.jar**, which was previously deployed to database **dev**, this AE must use the same database. (The path for the base64.jar is defined by the **–linkargs** option path.)

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java --template
compilejar --version 3 --db dev \
```

```
--linkargs /nz/export/ae/applications/dev/admin/java/base64.jar
\ --exe apply.jar \
"JarApplyDriverV2.java ApplyDriverV2.java ApplyOperation.java
\ ApplyResult.java ApplyUtil.java"
```

Registration as Scalar Function applyOperationV2SfJar

This example shows how to register in order to use a class inside a JAR file using AE environment variable NZAE_APPEND_CLASSPATH. Because the example relies on the third-party JAR **base64.jar**, and the AE jar file **apply.jar**, both of which have previously been deployed to database **dev**, this AE must use the same database. The NZAE_APPEND_CLASSPATH environment variable allows the java CLASSPATH environment variable to be set (to locate the previously deployed **base64.jar**). Note the use of **–exe apply.jar**, which ensures that apply.jar is also added to CLASSPATH. (This assumes the compile and register steps are both in the same database, in this case **dev**.).

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udf --version 3 --db dev \
     --sig "applyOperationV2SfJar(varargs)" --return double \
     --define java_class=org.netezza.education.JarApplyDriverV2 \
     --exe apply.jar --level 1 --environment "'APPLY_FUNCTION'='OPERATION'"
     \ --environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'" \ -
     -environment "'NZAE_APPEND_CLASSPATH'=\
     '/nz/export/ae/applications/dev/admin/java/base64.jar'
     "
```

Registration as Table Function applyOperationV2TfJar

This example shows how to register in order to use a class inside a JAR file using AE environment variable NZAE_APPEND_CLASSPATH, as above. It uses a table function invocation.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udtf --version 3 --db dev \
     --sig "applyOperationV2TfJar(varargs)" --return "table(result double)"
     \ --define java_class=org.netezza.education.JarApplyDriverV2 \
     --exe apply.jar --level 1 \
     --environment "'APPLY_FUNCTION'='OPERATION'" \
     --environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'" \
     --environment "'NZAE_APPEND_CLASSPATH'=\
     '/nz/export/ae/applications/dev/admin/java/base64.jar'"
```

Registration as Scalar Function applyResultV2Jar

This example shows how to register in order to use a class inside a JAR file using AE environment variable NZAE_APPEND_CLASSPATH.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udtf --version 3 --db dev \
     --sig "applyResultV2Jar(varchar(16), double)"
     --return "table(result double)" \
     --define java_class=org.netezza.education.JarApplyDriverV2
     \ --exe apply.jar --level 1 --noparallel \
     --environment "'APPLY_FUNCTION'='RESULT'" \
     --environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'" \
     --environment "'NZAE_APPEND_CLASSPATH'=\
     '/nz/export/ae/applications/dev/admin/java/base64.jar'"
```

Running JarApplyDriverV2.java

```
SELECT re.result FROM edutestdata,
     TABLE WITH FINAL(applyOperationV2TfJar('+', f1, f2, f3, f4))

     op, TABLE WITH FINAL(applyResultV2Jar('sum', op.result)) re
```

```
     WHERE color = 'red';
RESULT
-------
   413
```

CloneRows

Code in this section is saved in a file called CloneRows.java.

Concepts

This example shows three major concepts:

A function AE called from a table function that returns more than one output row per input row.

A function AE called from a table function that uses a shaper.

Using dynamic AE environment variables in an SQL function call.

This AE can accept any input signature. Using query metadata it sets the output row data type definition equal to the input row signature. Based on the CLONE_COUNT AE environment variable, it clones the input row the specified number of times. It also uses the CLONE_COLUMN_NAMES environment variable to set the output column names.

Code

Note that this code supports two APIs: the AE shaper API to perform shaping and the AE function API to process the data. This code will be used in the section ApplyDriver Version 3.

```
   CloneRows.java
   Function AE
   Handles a variable number of input columns and data types
   Returns a clone of each input row a multiplier number of times
   Clone multiplier is an integer >= 1
   Uses a shaper
   Receives clone multiplier as an environment variable
   Receives return column names as an environment variable
   Must be called as a table function
package org.netezza.education;

import org.netezza.ae.*;

public class CloneRows {
    public static void runShaper(NzaeShaper ae) {
        try {
            runShaperImpl(ae);
        } catch (Throwable t) {
            t.printStackTrace()
            ;
            try { ae.userError(t.getMessage()); }
            catch (Throwable t2) {}
        }
    }

    private static void runShaperImpl(NzaeShaper aeShp)
        { String strCloneCount =
            aeShp.getEnvironment().getValue("CLONE_COUNT");
        if (strCloneCount == null) {
            String msg =
                "CLONE_COUNT environment variable not set";
            System.err.println(msg);
            aeShp.userError(msg);
            return;
        }
```

```
int cloneCount = 0;
try {
    cloneCount = Integer.parseInt(strCloneCount);
} catch (NumberFormatException e) {}
if (cloneCount < 1) {
    String msg =
        "CLONE_COUNT must be a positive
    integer"; System.err.println(msg);
    aeShp.userError(msg);
    return;
}

String [] columnNames = new String[0];
String strCloneNames =
    aeShp.getEnvironment().getValue(
        "CLONE_COLUMN_NAMES");
if (strCloneNames != null)
{
    columnNames = strCloneNames.split(",");
    for (int i = 0; i < columnNames.length; i++) {
        if (columnNames[i] == null
                    columnNames[i].length() == 0)
            { String msg = "column name "
                + i + " is invalid";
            System.err.println(msg);
            aeShp.userError(msg);
            return;
        }
    }
}

NzaeMetadata meta = aeShp.getMetadata();

if (aeShp.catalogIsUpper()) {
    for (int i = 0; i < columnNames.length; i++) {
        columnNames[i] = columnNames[i].toUpperCase();
    }
}

int numInputColumns = meta.getInputColumnCount();
for (int i = 0; i < numInputColumns; i++) {
    String name;
    if (i < columnNames.length) {
        name = columnNames[i];
    } else {
        name = "F" + (i + 1);
    }

    switch (meta.getInputNzType(i)) {
    case NzaeDataTypes.NZUDSUDX_FIXED:
    case NzaeDataTypes.NZUDSUDX_VARIABLE:
    case NzaeDataTypes.NZUDSUDX_NATIONAL_FIXED:
    case NzaeDataTypes.NZUDSUDX_NATIONAL_VARIABLE:
        aeShp.addOutputColumnString(
            meta.getInputNzType(i),
            name, meta.getInputSize(i));
        break;
    case NzaeDataTypes.NZUDSUDX_NUMERIC32:
    case NzaeDataTypes.NZUDSUDX_NUMERIC64:
    case NzaeDataTypes.NZUDSUDX_NUMERIC128:
        aeShp.addOutputColumnNumeric(
            meta.getInputNzType(i),
            name, meta.getInputSize(i),
            meta.getInputScale(i));
        break;
    default:
        aeShp.addOutputColumn(
```

```
                    meta.getInputNzType(i), name);
                break;
            }
        }
        aeShp.update();
    }

    public static void runAe(Nzae ae) {
        try {
            runAeImpl(ae);
        } catch (Throwable t) {
            t.printStackTrace()
            ;
            try { ae.userError(t.getMessage()); }
            catch (Throwable t2) {}
        }
    }

    private static void runAeImpl(Nzae ae) {
        String strCloneCount =
            ae.getEnvironment().getValue("CLONE_COUNT");
        if (strCloneCount == null) {
            String msg =
                "CLONE_COUNT environment variable not set";
            System.err.println(msg);
            ae.userError(msg);
            return;
        }
        int cloneCount = 0;
        try {
            cloneCount = Integer.parseInt(strCloneCount);
        } catch (NumberFormatException e) {}
        if (cloneCount < 1) {
            String msg =
                "CLONE_COUNT must be a positive
            integer"; System.err.println(msg);
            ae.userError(msg);
            return;
        }
        // input row loop
        for (;;) {
            NzaeRecord input = ae.next();
            if (input == null) break;
            for (int i = 0; i < cloneCount; i++) {
                ae.outputResult(input);
            }
        }

           it's critical that done get called
        ae.done();
    }
}
```

DataConnection

Code in this section is saved in a file called DataConnection.java.

Code

All the connections-to-purposes mappings have been reorganized into the DataConnection class. The AE determines its purpose based on the API type of the data connection and the value of the AE environment variables. This code will be used in the section ApplyDriver Version 3.

```
DataConnection.java
Encapsulates logic used once a data connection
has been obtained
```

```
package org.netezza.education;

import org.netezza.ae.*;

public class DataConnection
{
    static public void useApi(NzaeApi api) {
        Nzae ae;
        switch (api.apiType)
        {
        case NzaeApi.FUNCTION:
            ae = api.aeFunction;
            break;
        case NzaeApi.SHAPER:
            try {
                String whichFunction =
                    api.aeShaper.getEnvironment().getValue(
                        "APPLY_FUNCTION");
                if (whichFunction == null) {
                    String msg =
                        "missing APPLY_FUNCTION " +
                        "environment variable";
                    System.err.println(msg);
                    api.aeShaper.userError(msg);
                } else if (!whichFunction.equals("CLONE"))
                    { String msg = "only clone has a
                    shaper"; System.err.println(msg);
                    api.aeShaper.userError(msg);
                } else {
                    String printInfo =
                        api.aeShaper.getEnvironment().getValue(
                            "APPLY_INFO");
                    if (printInfo != null
                            && printInfo.equalsIgnoreCase(
                                "true")) {
                        ApplyUtil.printMetadata(api.aeShaper);
                        ApplyUtil.printRuntime(api.aeShaper);
                        ApplyUtil.printSharedLibraries(
                            api.aeShaper);
                        ApplyUtil.printEnvironment(
                            api.aeShaper);
                    }
                    CloneRows.runShaper(api.aeShaper);
                    return;
                }
            } finally {
                api.close();
            }
            return;
        default:
        {
            System.err.println("unexpected API");
            api.close();
            return;
        } // end of default }

        // end of case

        try {
               determine which function is being
            invoked String whichFunction =
                ae.getEnvironment().getValue("APPLY_FUNCTION");
            if (whichFunction == null) {
                ae.userError(
                    "missing APPLY_FUNCTION "
                    + "environment variable");
                return;
            }
            String printInfo =
```

```
                ae.getEnvironment().getValue("APPLY_INFO");
            if (printInfo != null
                    printInfo.equalsIgnoreCase("true")) {
                ApplyUtil.printMetadata(ae);
                ApplyUtil.printRuntime(ae);
                ApplyUtil.printSharedLibraries(ae);
                ApplyUtil.printEnvironment(ae);
            }

            if (whichFunction.equals("OPERATION")) {
                ApplyOperation.runAe(ae);
            } else if (whichFunction.equals("RESULT"))
                { ApplyResult.runAe(ae);
            } else if (whichFunction.equals("CLONE")) {
                CloneRows.runAe(ae);
            } else {
                ae.userError("unexpected value for " +
                    "environment variable APPLY_FUNCTION");
            }
        } finally {
            ae.close();
        }
    }
}
```

ApplyDriver Version 3

Code in this section is saved in a file called ApplyDriverV3.java.

Code

The main code for determing the AE logic has been moved from ApplyDriverV2 to DataConnection. Therefore, ApplyDriverV3 is used to retrieve a local AE data connection API and call DataConnection.useApi.

```
   ApplyDriverV3.java
   driver for Module 4 package

org.netezza.education;

import org.netezza.ae.*;

public class ApplyDriverV3 {
    public static final void main(String [] args) { NzaeFactory

        factory = NzaeFactory.Source.getFactory();

        if (factory.isLocal()) {
            DataConnection.useApi(factory.getLocalApi());
        } else {
            throw new RuntimeException(
                "Expecting Local AE only");
        }
    }
}
```

Compilation

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java --template
compile --version 3 --db dev \
     "ApplyDriverV3.java DataConnection.java ApplyOperation.java

     \ ApplyResult.java ApplyUtil.java CloneRows.java"
```

Registration

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udf --version 3 --db dev \
     --sig "applyOperationV3Sf(varargs)" --return double \
     --define java_class=org.netezza.education.ApplyDriverV3 --level 1
     \ --environment "'APPLY_FUNCTION'='OPERATION'" \
     --environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"

$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udtf --version 3 --db dev \
     --sig "applyOperationV3Tf(varargs)" --return "table(result double)"
     \ --define java_class=org.netezza.education.ApplyDriverV3 --level 1
     \ --environment "'APPLY_FUNCTION'='OPERATION'" \
     --environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"

$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udtf --version 3 --db dev \
     --sig "applyResultV3(varchar(16), double)" \
     --return "table(result double)" \
     --define java_class=org.netezza.education.ApplyDriverV3 --level 1
     \ --noparallel --environment "'APPLY_FUNCTION'='RESULT'" \ --
     environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"

$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udtf --version 3 --db dev \
     --sig "cloneRowsV3(varargs)" --return "table(any)" \
     --define java_class=org.netezza.education.ApplyDriverV3 --dynamic 2
     \ --level 1 --environment "'APPLY_FUNCTION'='CLONE'" \ --environment
     "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"
```

Running Table Function cloneRowsV3

This is a direct test of the clone functionality.

```
SELECT * FROM TABLE WITH FINAL(cloneRowsV3(1, 2, 3, 4, 'CLONE_COUNT=3,
     CLONE_COLUMN_NAMES="col1,col2,col3,col4"'));
 COL1 | COL2 | COL3 | COL4
      +      +      +
------1 |------2|------3| ------4
    1 |     2|     3|     4
    1 |     2|     3|     4
(3 rows)
```

Running All Functions

The output of the cloneRowsV3 SQL function is streamed into the applyOperationV3Tf SQL function, which in turn is streamed into the applyResultV3 SQL function.

```
SELECT re.result FROM edutestdata, TABLE WITH FINAL(cloneRowsV3(f1, f2, f3,
     f4, 'CLONE_COUNT=3, CLONE_COLUMN_NAMES="col1,col2,col3,col4"')),
     TABLE WITH FINAL(applyOperationV3Tf(> '+', col1, col2, col3, col4)) op,
     TABLE WITH FINAL(applyResultV3('sum', op.result)) re WHERE color = 'red';
 RESULT
--------
   1239
```

ApplyDriver Version 4

Code in this section is saved in a file called ApplyDriverV4.java.

Concepts

ApplyDriverV4 replaces ApplyDriverV3 and includes logic for obtaining both remote and local data connections. This uses the lower level (less user friendly) methods instead of the easier to use but less functional NzaeApiGenerator class to support both local and remote modes.

This technique is most useful with remote AEs not launched with the AE system. It also illustrates how NzaeApiGenerator works.

When a remote AE is launched by the AE runtime system, the connection point settings typically come from the AE environment indirectly via methods such as getRemoteName. However, the AE connection point settings can be literals or come from any source.

Code

```
   ApplyDriverV4.java
   demonstrates Remote AE

package org.netezza.education;

import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import org.netezza.ae.*;

public class ApplyDriverV4 {
    private static final Executor exec =
        Executors.newCachedThreadPool();

    public static final void main(String [] args) { NzaeFactory

        factory = NzaeFactory.Source.getFactory();

        if (factory.isLocal()) {
            DataConnection.useApi(factory.getLocalApi());
        } else {
            NzaeConnectionPoint conpt =
                factory.newConnectionPoint();
            conpt.setName(conpt.getRemoteName());
            conpt.setSessionId(conpt.getRemoteSessionId());
            conpt.setTransactionId(
                conpt.getRemoteTransactionId());
            conpt.setDataSliceId(conpt.getRemoteDataSliceId());
            System.out.println("listening on: "
                  " name=" + conpt.getName()
                  ", data slice=" + conpt.getDataSliceId()
                  ", transaction=" + conpt.getTransactionId()
```

```
                  ", session=" + conpt.getSessionId());
            NzaeRemoteProtocol remprot =
                factory.createListener(conpt);
            for (;;)
            {
                final NzaeApi api = remprot.acceptConnection();
                Runnable task = new Runnable() {
                    public void run() {
                        DataConnection.useApi(api);
                    }
                };
                exec.execute(task);
            }
        }
    }
}
```

Compilation

```
$NZ_EXPORT_DIR/ae/utilities/bin/compile_ae --language java --template
compile --version 3 --db dev \
     "ApplyDriverV4.java DataConnection.java ApplyOperation.java
     \ ApplyResult.java ApplyUtil.java CloneRows.java"
```

Registration

These registrations are for the same SQL functions used previously, named with different version numbers.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udf --version 3 --db dev \
     --sig "applyOperationRemV4Sf(varargs)" --return double \
     --define java_class="" --remote --rname apply --level 1
     \ --environment "'APPLY_FUNCTION'='OPERATION'" \ --
     environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"

$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udtf --version 3 --db dev \
     --sig "applyOperationRemV4Tf(varargs)" --return "table(result double)"
     \ --define java_class="" --remote --rname apply --level 1 \ --
     environment "'APPLY_FUNCTION'='OPERATION'" \
     --environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"

$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udtf --version 3 --db dev \
     --sig "applyResultRemV4(varchar(16), double)" \
     --return "table(result double)" --define java_class=""
     \ --remote --rname apply --level 1 --noparallel \ --
     environment "'APPLY_FUNCTION'='RESULT'" \ --environment
     "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"

$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template
udtf --version 3 --db dev \
     --sig "cloneRowsRemV4(varargs)" --return "table(any)" \
     --define java_class="" --remote --rname apply --dynamic 2 --level 1
     \ --environment "'APPLY_FUNCTION'='CLONE'" \
     --environment "'APPLY_INFO'='true'" \
     --environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"
```

Registration for Launching

This registration is for an SQL function to launch this remote AE.

```
$NZ_EXPORT_DIR/ae/utilities/bin/register_ae --language java --template udtf
```

```
--version 3 --db dev \
     --sig "apply_launch_v1(bigint)" --return "TABLE(aeresult varchar(255))" \
--define java_class=org.netezza.education.ApplyDriverV4 --launch \
--rname apply --level 1 --environment "'APPLY_FUNCTION'='CLONE'" \
--environment "'APPLY INFO'='true'" \
--environment "'NZAE_LOG_DIR'='/nz/export/ae/log'"
```

Launching on the Host

```
SELECT aeresult FROM TABLE WITH FINAL(apply_launch_v1(0));
    AERESULT
-------------------------------------------------
tran: 17928 session: 16522 DATA slc: 0 hardware: 0
machine: hostlnx1 process: 27184 thread: 27185
```

Pinging on the Host

```
SELECT aeresult FROM TABLE WITH FINAL(inza..nzaejobcontrol('ping', 0,
     'apply', false, NULL, NULL));
                                                AERESULT
-------------------------------------------------------
hostlnx1  27184 (apply dataslc:-1 sess:-1 trans:-1)
thread: 27185 AE Build: 10 nzrep version: 9
```

Launching on the SPUs

```
SELECT aeresult FROM _v_dual_dslice, TABLE WITH FINAL(apply_launch_v1(dsid));
     AERESULT
-------------------------------------------------------
 tran: 17916 session: 16522 DATA slc: 4 hardware: 1002
 machine: spu0101 process: 21751 thread: 21755
```

Pinging on the SPUs

```
SELECT aeresult FROM _v_dual_dslice, TABLE WITH FINAL(inza..nzaejobcontrol(
     'ping', dsid, 'apply', false, NULL, NULL));
     AERESULT
-------------------------------------------------------
 spu0101  21751 (apply dataslc:-1 sess:-1 trans:-1)
 thread: 21755 AE Build: 10 nzrep version: 9
```

Running

```
SELECT re.result FROM edutestdata, TABLE WITH FINAL(cloneRowsRemV4(f1, f2,
     f3, f4, 'CLONE_COUNT=3, CLONE_COLUMN_NAMES="col1,col2,col3,col4"')),
     TABLE WITH FINAL(applyOperationRemV4Tf('+', col1, col2, col3, col4))
     op, TABLE WITH FINAL(applyResultRemV4('sum', op.result)) re
     WHERE color = 'red';
 RESULT
--------
   1239
```

Stopping on the Host

```
SELECT aeresult FROM TABLE WITH FINAL(inza..nzaejobcontrol('stop', 0,
     'apply', false, NULL, NULL));
                             AERESULT
-------------------------------------------------------------
hostlnx1 27184 (apply dataslc:-1 sess:-1 trans:-1) AE stopped

 Stop ApplyDriverV4.java on the S-Blades

SELECT aeresult FROM _v_dual_dslice, TABLE WITH
     FINAL(inza..nzaejobcontrol('stop', dsid, 'apply', false, NULL, NULL));
 AERESULT
-------------------------------------------------------------
 spu0101  21751 (apply dataslc:-1 sess:-1 trans:-1) AE stopped
```

```
SELECT aeresult FROM _v_dual_dslice, TABLE WITH
     final(inza..nzaejobcontrol('cleanup', dsid, 'apply', false, NULL, NULL));
 AERESULT
----------
(0 rows)
```

ApplyDriver Version 5

Code in this section is saved in a file called ApplyDriverV5.java.

Concepts

In contrast to ApplyDriverV4, ApplyDriverV5 supports both local and remote AEs using NzaeApiGenerator. Job control and execution is the same as for the ApplyDriverV4 example.

Code

```
   ApplyDriverV5.java
   demonstrates Local and Remote AE using
NzaeApiGenerator package org.netezza.education;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import org.netezza.ae.*;

public class ApplyDriverV5 {
    private static final Executor exec =
        Executors.newCachedThreadPool();

    public static final void main(String [] args) {
        NzaeApiGenerator helper = new NzaeApiGenerator();
        for (;;) {
            // accept an API type of data connection
            final NzaeApi api = helper.getApi(NzaeApi.ANY);
            if (!helper.isRemote()) {
                DataConnection.useApi(api);
                break;
            }

           Runnable task = new Runnable() {
                public void run() {
                    DataConnection.useApi(api);
                }
            };
            exec.execute(task);
        }
    }
}
```

CHAPTER 20
Administering AEs and Related Processes

Using the Backup and Restore Utilities

You can use the **nzbackup** command to back up a database, including all schema objects and all table data within the database. However, since AEs consist of user-created code that is not stored in the database, the standard NPS backup does not back up AEs. Netezza Analytics provides two additional commands to back up and restore AEs: **inzabackup** and **inzarestore**.

Using inzabackup

The **inzabackup** utility is used to perform a full or partial backup of AEs. The utility backs up AE code, objects, and any files for AEs registered to the appropriate database(s). During backup, the AE files, typically located in /nz/export/ae/applications, are archived in a compressed tar.gz file. You specify the path and name of the tar.gz file.

When performing a backup, a locking mechanism prevents the procedure from running if any other INZA backup, restore, or compile operations are underway, ensuring that the backup is consistent.

Note: The **inzabackup** utility does not back up the database itself, but affects only the files related to AEs. The **nzbackup** command must be used to back up a database.

For more information on using **nzbackup** to back up a database, refer to the *Netezza System Administrator's Guide*.

Usage

```
inzabackup [-d <db>] <path_and_filename>
inzabackup -h
```

Options

The **inzabackup** utility uses the following arguments:

-d <db>—Allows AEs registered to a single database to be backed up. By default, AEs registered to all databases are backed up. Optional.

<path_and_filename>—Specifies the path and name of the archive file to be created. Required.

-h—Prints this help.

Examples

The following command performs a backup of all registered AEs and creates an archive called full.backup.tar.gz located in /tmp.

```
$NZ_EXPORT_DIR/ae/utilities/bin/inzabackup /tmp/full.backup.tar.gz

Starting inza backup.
Archiving ae …
inza backup completed successfully.
```

The following command performs a backup of the AEs registered only to the dev database, and creates an archive called dev.backup.tar.gz located in /tmp.

```
$NZ_EXPORT_DIR/ae/utilities/bin/inzabackup -d  dev /tmp/dev.backup.tar.gz

Starting inza backup.
Archiving ae …
inza backup completed successfully.
```

Using inzarestore

The **inzarestore** utility is used to restore AEs from an archive file. Depending on whether the archive contains a full or partial backup, the restore operation may affect only a specific database, or all databases on the NPS. Once the AEs are restored, you may need to be re-registered to the database before use. This would happen if the state of the database is not consistent with the inzarestore you just applied (for example, if inzarestore restored an older copy that is out of sync with current database).

When performing a restoration, a locking mechanism prevents the procedure from running if any other INZA backup, restore, or compile operations are underway to ensure that the restoration is consistent.

Note: The **inzarestore** utility does not restore a database, but restores only files related to AEs. The **nzrestore** command must be used to restore a database.

For more information on using **nzrestore** to restore a database, refer to the *Netezza System Administrator's Guide*.

Usage

```
inzarestore [-f] <path_and_filename>
inzarestore -h
```

Options

The **inzarestore** utility uses the following arguments:

-f—Forces the command to run without prompting for confirmation.

<path_and_filename>—Specifies the path and name of the archive file to be restored.

-h—Prints this help.

Examples

The examples in this section are based on files resulting from the backup commands found in the Using inzabackup section.

The following command restores AEs from a full backup archive file called full.backup.tar.gz found in /tmp.

```
$NZ_EXPORT_DIR/ae/utilities/bin/inzarestore

/tmp/full.backup.tar.gz Extracting inza data archive …

Restore inza data archived Fri Apr 8 11:09:57 EDT 2011? (y/n) [n] y
Starting inza restorer.
Installing ae contents to '/nz/export/ae/applications/' ...
inza restore completed successfully.
```

The following command restores AEs from a partial backup archive file called dev.backup.tar.gz found in /tmp. This archive was specific to a database named **dev**.

```
$NZ_EXPORT_DIR/ae/utilities/bin/inzarestore

/tmp/dev.backup.tar.gz Extracting inza data archive …

Restore inza data archived Fri Apr 8 11:10:16 EDT 2011? (y/n) [n] y
Starting inza restorer.
Installing ae contents to '/nz/export/ae/applications/dev' ...
inza restore completed successfully.
```

Using the Admin Utilities

Provided as part of the Admin Utilities cartridge, the admin utilities consist of listlogs, viewlog, cleanlogs, and aborthung.

Log Files

Complete documentation concerning log file management can be found in the section Working with Log Files.

Abort Hung

This utility can be used to abort a hung AE. An AE may hang if there are bugs in the AE code.

Abort Hung takes two integers--a session ID and a dataslice ID. The session ID can be either a valid session ID or “-1” for all sessions. The dataslice ID is ignored by the aborthung function, but is used to control the locus of execution (host or SPU, as well as dataslice).

This function returns the dataslice ID and the text indicating what was aborted.

```
SELECT * FROM TABLE(aborthung(-1,0));
```

```
DSID |           TXT
—+———————-
0 | Aborted 1 hung processes
(1 row)
SELECT * FROM TABLE(aborthung(-1,0));
DSID | TXT
—+—
(0 rows)
```

For all of the utilities, the dataslice ID parameter controls whether it is run on the host or the SPU. To run it on the SPU:

```
SELECT * FROM _v_dual_dslice, TABLE WITH FINAL(aborthung(-1,dsid));
```

This command correlates the table function with the _v_dual_dslice table, which exists on the SPU, causing the table function to be run on the SPU.

CHAPTER 21
Best Practices

Considering System Stability and Resource Management

When using the NPS user-defined function for both AEs and UDXs, code is deployed inside the database system extending the NPS appliance's functionality. This capability provides problem-solving power but requires consideration of the impact of the AEs and UDXs on system stability and resource usage.

Working with UDXs and AEs

A classic, unfenced UDX runs inside an NPS system process. Therefore, an unfenced UDX has access to the NPS process memory address space. This implies that bugs that produce "wild pointers" can corrupt the NPS system memory.

If separate process address space is required, use the UDX version called "fenced," which runs the UDX in a separate process. Fenced UDXs use the UDX API, which must be written in C or C++ and have a slower data transfer than unfenced UDXs since they are run in a different process.

AEs run in a separate process address space and do not have this memory corrupting potential. While AEs do not share memory with the NPS system, they do use more system resources than unfenced UDXs. A summary of these resources is shown below:

AEs are processes and not library code, such as an unfenced UDX. Processes inherently use more resources. For instance, a Linux installation encounters performance problems if there are too many simultaneous executing processes. However, remote AEs may use fewer resources if the daemon style is used to service too many requests using threading. This is important for remote AEs where the NPS system does not control their life cycle so that unnecessary remote AEs are not left running in the background.

AEs and UDXs have access to local disk space on each SPU and the host. Using too much temporary disk space, especially on a SPU, can destabilize the NPS system. Also, a high level of AE or UDX local disk I/O can compete with the NPS system database I/O.

AEs and UDXs have access to the AE export directory tree. Because this is located on a network drive, excessive use can impact the network performance of the entire NPS system.

UDXs and AEs compete with the NPS system for CPU, memory, network bandwidth, and disk I/O. The performance of the entire NPS system can be impacted by UDXs or AEs if their combined resource usage becomes too high.

Some programming languages use more resources than others. For example, a script language interpreter, such as the one for Python, typically uses more system resources than the same functionality implemented in C++.

Backup Practices

Because backup and restore works with external contents, you must use a convention on how and where the content is stored on the file system and used with the database. As part of that convention, the expectation is that files required for an AE implementation are stored under:

```
NZ_EXPORT_DIR/ae/applications/<db>
```

Because <db> is replaced by the database name, partial backups and restores of databases can be carried out. Without this convention, there is no way of knowing if performing a partial restore of files for AEs in one database damages AEs in other databases because the files could be co-mingled.

Therefore, deploying the same AE in multiple databases means that there are multiple copies of the files stored under different database directories. Using this convention when using **compile_ae** and **register_ae** is recommended.

APPENDIX A
File Names Used in Examples

Table of File Names

The following table lists the name of the file used for each corresponding example. It also provides a link to the appropriate page.

Table 10: File names used in examples

Example	File Name	Page Number
Scalar Function		
C Language Scalar Function	func.c	65
C++ Language Scalar Function	applyopcpp.cpp	70
Java Language Scalar Function	TestJavaInterface.java	76
Fortran Language Scalar Function	applyop.f	82
Python Language Scalar Function	applyop.py	85
Perl Language Scalar Function	ApplyOp.pm	87
Converting to a Simple Table Function		
C Language Conversion	func.c	93
C++ Language Conversion	applyopcpp.cpp	94
Java Language Conversion	TestJavaInterface.java	94

Example	File Name	Page Number
Python Language Scalar Function (Remote Mode)	applyop.py	239
Perl Language Scalar Function (Remote Mode)	ApplyOp.pm	240
Advanced Remote Mode Multiple Simultaneous Connections		
C Language Example (Multiple Simultaneous Connections)	sstring.c	245
C++ Language Example (Multiple Simultaneous Connections)	fork.cpp	250
Java Language Example (Multiple Simultaneous Connections)	N/A	253
Fortran Language Example (Multiple Simultaneous Connections)	N/A	253
Python Language Example (Multiple Simultaneous Connections)	fork.py	253
Perl Language Example (Multiple Simultaneous Connections)	ForkAe.pm	255
Debugging		
C Language Logging and Runtime Information	log.c	293
C++ Language Logging and Runtime Information	runtime.cpp	297
Java Language Logging and Runtime Information	TestJavaRuntime.java	300
Fortran Language Logging and Runtime Information	logrun.f	303
Python Language Logging and Runtime Information	logrun.py	306

Example	File Name	Page Number
Perl Language Logging and Runtime Information	LogRunAe.pm	308

APPENDIX B
Adding a Shared Library to the Perl Adapter

Netezza analytics includes a framework to support Perl shared library installations, enhancing functionality when using the Perl adapter to write AEs. The framework allows users to add their own Perl libraries, or libraries from CPAN, to the analytics host and the analytics SPU. Those libraries then become available to the Netezza analytics software. The following sections describe using and testing the framework.

Understanding the Perl Environment

The Netezza appliance has multiple Perl installations, supporting various system functionality. One installation is part of the core NPS system. Another is installed on the analytics host, and a third on the analytics SPU. The Perl adapter framework assures that the library modules install into the analytics Perl components, making them available for user-written Perl adapter code. In other words, it installs the modules so that the host and SPU versions of Perl can find and use them. By using the framework, you also produce an installation in which libraries are safe across installs. (As with every AE, you still must re-register modules when you upgrade or downgrade the software.)

Using CPAN (Comprehensive Perl Archive Network)

To add a Perl module from CPAN, you must run the **nzcpan** command and then install your module using the CPAN application commands. Once you have installed the Netezza analytics packages, setup CPAN for hosts and spu and install a module to test:

From a command line, run **nzcpan host**. If you have never run the command before, accept the defaults the command chooses. If the command has run before, you will not be prompted.

```
[nz@host]$ nzcpan host
```

Once you have the cpan prompt, issue the following command:

```
install String::Util
```

When the previous command completes, run **nzcpan spu** from the command line. Again, if the

command has never run before, accept the defaults.

Once you have the cpan prompt, issue the following command:

```
force install String::Util
```

In this case, you must use **force** because the CPAN resources are shared (although though the paths for Perl are different).

When the installations are complete, exit **nzcpan** and create the following AE:

```
#############################################################################
package Unquote;
use nzae::Ae;
use strict;
use String::Util ':all';
use autodie;
our @ISA = qw(nzae::Ae);
my $ae = Unquote->new();
$ae->run();
sub _getFunctionResult(@)
{
   my $self = shift;
     BREAK APART OUR ROW OF INPUT.
   my $str = $_[0];
   return unquote($str);
}
1;
#############################################################################
```

Compile and register the program using the following:

```
commands:/nz/export/ae/utilities/bin/compile_ae --language perl --version
3 --template deploy Unquote.pm
/nz/export/ae/utilities/bin/register_ae --language perl --version 3 --template
udf --exe Unquote.pm --sig "unquote(varchar(10000))" --return "varchar(10000)"
--level 4
```

To test the installation, execute the following:

```
[nz@host]$ nzsql
Welcome to nzsql, the IBM Netezza SQL interactive terminal.
Type: \h for help with SQL commands
? for help on internal slash commands
\g or terminate with semicolon to execute query
\q to quit
SYSTEM(ADMIN)=> select unquote('"hello"');
UNQUOTE
hello

(1 row)
```

Installing a Packaged non-CPAN Perl Module

There are instances when you might want to install a non-CPAN (user-written) packaged library module. For example, you might have written and packaged a library for internal use, or downloaded one from a site other than CPAN. In that case, follow these steps (the package name in this example is Digest::SHA):

Extract the module source code and change to the extracted source directory.

Prepare (clean) the directory for installation, assuming that your make command has a clean

argument available.

```
make clean
```

Execute **nzenv** and add each of the results to your environment. Execute the command as follows to add Netezza-specific environment variables:

```
eval `nzenv`
```

Export the following variables:

```
export EXE_POINTER_SIZE=32
export ABI=32
export CC
export CXX
PATH=$NZ_EXPORT_DIR/ae/sysroot/host/bin:
$NZ_EXPORT_DIR/ae/languages/perl/5.8/host/bin:${PATH} export
LD_LIBRARY_PATH=$NZ_EXPORT_DIR/ae/sysroot/host/lib:
$NZ_EXPORT_DIR/ae/languages/perl/5.8/host/lib
export LD_RUN_PATH=$LD_LIBRARY_PATH
export CFLAGS=-m32
export CPPFLAGS=-m32
export CXXFLAGS=-m32
export LDFLAGS=-m32
```

Using the host Perl module, run the Perl makefile, passing the prefix for the directory:

```
/nz/export/ae/languages/perl/5.8/host/bin/perl Makefile.PL
PREFIX=$ {NZ_EXPORT_DIR}/ae/applications/perl/host
```

Run **make** to build the target.

Run **make install** to perform the install.

Finally, test the installation:

```
/nz/export/ae/languages/perl/5.8/spu/bin/perl -e 'use Digest::SHA;'
```

Installing an Individual non-CPAN Perl Module

There may be instances when you have user-written internal library module that is not packaged (is a single file). In that case, simply copy the file to a directory that can be accessed by the analytics host and SPU. In the example below, the file is called MyPerlMod:

```
cp MyPerlMod.pm $NZ_EXPORT_DIR/ae/applications/perl/host/lib/perl5/ cp
MyPerlMod.pm $NZ_EXPORT_DIR/ae/applications/perl/spu/lib/perl5/
```

APPENDIX C
Extended R Language Functionality

R AE Execution in Details

R AEs are built on top of regular Analytic Executables *(*AEs) to support the R programming language with the R environment on Netezza. The layer handling R on the NPS is called the R Language Adapter, or simply the R Adapter.

The following R code sample, which assumes at least one input column, calculates the square root of the first input column and outputs the result as the only output column.

```
nz.fun  <- function() {
  while(getNext()) {
    x <- getInputColumn(0)
    setOutputDouble(0,sqrt(as.double(x)))
    outputResult()
  }
}
```

Create a */tmp/rae.R* file and enter the code above. After you save the file, you can compile it. The following sample command invokes the compilation tool, assuming that the NZ_EXPORT_DIR environment variable points to /nz/export:

```
/nz/export/ae/utilities/bin/compile_ae --language r \
     --template compile --version 3 --db dev --user nz /tmp/rae.R
```

This command creates the *rae.R* file that is located in /nz/export/ae/applications/dev/nz/. The last two directories match the database name and the user name that is specified in the command, respectively.

The last step to run the R Analytic Executable from SQL is registration. The sample code for registering the compiled file in the *dev* database is shown below:

```
/nz/export/ae/utilities/bin/register_ae --language r \
 --template udf --version 3 --db dev --user nz       \
 --sig 'rae(double)' --return 'double' --exe 'rae.R'
```

The following switches that appear in the above command are:

--sig—defines the UDX name along with the input signature

--return—defines the output signature or result type

--exe—points to the specific file that was compiled
You must save this file under /nz/export/ae/applications/dev/nz/; the full path is determined from the values of the **--db** and **--user** switches

After this command is completed, you can log into the *dev* database and run the new UDX:

```
SELECT rae(column_name) FROM table_name;
```

When you run the UDX, every data partition starts a new R process and invokes the R Adapter that calls the nz.fun function from the file as shown at the beginning of this section.

A sample output might look like following:

```
        RAE
-----------------
 2.5099800796022
 2.4083189157585
(2 rows)
```

Operating Modes and High-level API

The following operating modes are typically used by the client-side code, such as the the *Netezza R Library package* functions:

run—performs the user-provided function without modification

apply—calls the user-provided function for each row of the input table and applies it to the input data stream
The function should accept a parameter x , which is a list that contains all input columns

tapply—applies the user-provided function on groups of rows passed as *data.frames*

install—installs an R extension package provided by the user on Host and SPUs

groupedapply—applies a user-provided function on a number of data subsets that is created based on a user-specified GROUP BY clause

Each of these modes assumes that the user provides a number of input objects, such as a mode identifier, the function itself with any additional arguments in a strictly specified format, and so on.

Note: The meaning of the term mode as it pertains to R depends on the context. Here, it refers to a high-level server-side R function that accepts a number of user-provided parameters, that is, a function with optional arguments. These functions control the part of the flow not directly related to data processing, for example, the loop that builds an input row, then passes it to the user-provided function, and outputs the result of the function. All modes, with the exception of *run*, hide part of the low-level API from the user.

The following example shows the *apply* mode:

```
  we assume that 'fun' is the user-provided
function while (getNext()) {
  row <- list()
  for (i in seq(inputColumnCount()))
    row[[i]] <- getInputColumn(i-1)
  ret <- do.call(fun, c(list(x=row), args))

  if (length(ret)) {
    for (i in 1:length(ret))
      setOutput(i-1, ret[[i]])
    outputResult()
  }
}
```

Passing Code to R AE

Language Adapters for compiled languages like C++ and Fortran are designed as wrappers for the C API and do not allow custom code in the run time. The R language is different because each R snippet is interpreted rather than compiled when it is invoked as an Analytic Executable.

Therefore, it is assumed that when using R, code is passed to the AE through environment variables as serialized and base-64 encoded plain text, or by pointing to a file that is saved in the file system. In some of these modes, the presence of serialized R objects is assumed, while in others the code is passed as is. Common to all modes is that the user input is passed as the last argument of the function or aggregate that is being invoked. This mechanism is called *Dynamic Environment*:

```
SELECT * FROM some_table, TABLE WITH FINAL(r_udtf(some_table.some_column,
'ABSOLUTE_PATH=/path/to/a/file.R'));
```

Available modes are:

ABSOLUTE_PATH—points to any file available in the file system, with the referenced file containing a serialized list with functions and data
If this file is to be accessed from both Host and SPUs, then the best solution is to save it in the NFS share. The NFS share is pointed to by the $NZ_EXPORT_DIR environment variable, namely, $NZ_EXPORT_DIR/ae

WORKSPACE_PATH—points to a file of the same format as above, but located in the workspace directory, which by default is $NZ_EXPORT_DIR/ae/workspace/nz/r_ae.

PLAIN_PATH—points to a file containing R source code, including the same contents as above, however, in this case the referenced file is not serialized.

CODE_SERIALIZED—includes a serialized and base64-encoded list.

CODE_PLAIN—provides code in plain-text; this mode is most suitable for embedding R code in SQL queries and stored procedures, but requires caution to ensure that the code is correctly escaped, as required by SQL and *Dynamic Environment*.

Working Modes

The most straightforward way of constructing an R Analytic Executable requires the definition of the nz.fun object, followed by successful compilation and registration. Keep in mind that the nz.mode object takes the value of **run** by default; however for aggregates, it must be set to **aggregate**.

In general, the nz.mode object controls the working mode.

The following modes are available:

The simplest, the run mode, assumes the user provides all necessary control structures, including the main loop **while(getNext()){...}**

The aggregate mode assumes a somewhat different situation: the UDX is not a UDF or UDTF, but a UDA, which means state-based processing, state variables, and so on

The apply mode removes the requirement for the loop presented in the example for the run mode

The nztapply mode implements a form of *grouped apply* and is designed to be used with the

nzTApply function from the nzLibrary for R client package

The install mode enables CRAN packages installation

The groupedapply mode is an equivalent for the tapply mode that can be executed without the nzLibrary for R client package

run Mode

In this mode, the user has the full control over the data flow and operates on the lowest possible API level.

nz.mode—equals **run**

nz.fun—does not accept any default parameters except for **args**

nz.args—by default an empty list

nz.shaper—required if UDX output defined as TABLE(ANY)

nz.shaper.args—optional

nz.shaper.list—required if **nz.shaper** set to **std**

The following function can be registered as a UDF or a UDTF. It calculates the square root of the first input column and returns the result as the only output column.

```
nz.mode <- 'run'
nz.fun  <- function() {
  while(getNext()) {
    x <- getInputColumn(0)
    setOutput(0, sqrt(as.double(x)))
    outputResult()
  }
}
```

apply Mode

In the apply mode, the loop presented in the example for *run* is no longer necessary. The function should accept at least a parameter **x** containing the whole input row stored in a list. Its value is returned as the output row.

nz.mode—equals **apply**

nz.fun—accepts the parameter **x**, along with additional parameters from **nz.args**

nz.args—contains a list with arguments for **nz.fun** (by default it is an empty list)

nz.shaper—required if UDX output defined as TABLE(ANY)

nz.shaper.args—optional

nz.shaper.list—required if *shaper* set to **std**

An example of the apply setup is presented below:

```
nz.mode <- 'apply'
nz.fun  <- function(x) {
  return(sqrt(as.double(x[[1]])))
}
```

Each input row is passed as the function's sole argument. Since it can contain values of various data types, it is a list. Its elements can be accessed only by positional indices, no elements' names are

provided.

tapply Mode

This mode implements a form of *grouped apply*. It has been designed to be used with the nzTApply function from the nzLibrary for R client package and is not within the scope of this guide.

install Mode

This mode enables CRAN packages installation. Similarly to tapply it should be used indirectly, with the nz.install.package function of the nzLibrary for R client package.

groupedapply Mode

This is equivalent to the tapply mode, but it can be executed without the nzLibrary for R client package. It exhibits similar behavior, but enables you to run an aggregating table function and use an arbitrary SQL query, so long as it conforms with some basic requirements.

The R file must contain:

nz.mode—set to **groupedapply**,

nz.fun—the function run on processed

groups The R file can additionally contain:

nz.shaper—the shaper function; required if output set to TABLE(ANY)

nz.shaper.args—optional

nz.shaper.list—required if *shaper* set to **std**

This R source file may undergo the usual compilation and registration steps, or can be used directly as described In the *Communication Channels* section.

Once a selection is made, move to the next step, which is the SQL query:

```
SELECT ae_output_t.* FROM (SELECT
    ROW_NUMBER() OVER (PARTITION BY pp ORDER BY oo) AS
    rn, COUNT(*) OVER (PARTITION BY pp) AS ct,
    tt.*
  FROM tt) as input_t,
  TABLE WITH FINAL (nzr..r_udtf_any(rn, ct,
    tt.cc,..., '<R code>')) as ae_output_t;
```

The noteworthy elements are:

input table **tt**

control columns **rn** and **ct**, which are processed by Netezza R code invoked when the *groupedapply* mode is chosen

the R Adapter UTDF **r_udtf_any**, which is a standard UDTF internally invoking R and returning TABLE(ANY)

The control columns are used to determine when a new partition starts and ends. The internal R loop looks similar to the one below:

```
while (getNext()) {
  rn <- getInputColumn(0)
  ct <- getInputColumn(1)
  if (rn == 1) {
    # create a new data frame for this partition
  }
    fetch the current row
  if (ct == rn) {
      process the frame calling the user-provided
      function and return the function result
  }
}
```

The R loop assumes that the first two columns are the *row number* and the *row count.* Based on their values, it creates a *data.frame* for each new data partition. When the number of rows reaches the total count, the user function is called and the frame is passed as its argument under the name of **x**.

As an example, consider the contents of grpdapp.R, which should then be saved under /nz/export/ae/workspace:

```
nz.fun <- function(x)return(mean(x))
nz.mode <- 'groupedapply'
nz.shaper <- 'std'
nz.shaper.list <- list(a=NZ.DOUBLE, b=NZ.DOUBLE,
       c=NZ.DOUBLE, d=NZ.DOUBLE)
```

Given there is a table named *iris* (that is a copy of the standard R data set), you can call the following query:

```
SELECT ae_output_t.* FROM (SELECT
  ROW_NUMBER() OVER (PARTITION BY species ORDER BY sepal_length) AS
        rn, COUNT(*) OVER (PARTITION BY species) AS ct,
        iris.*
  FROM iris) AS input_t,
  TABLE WITH FINAL
(nzr..r_udtf_any(rn, ct, sepal_length, sepal_width,
  petal_length, petal_width,
  CAST('PLAIN_PATH=/nz/export/ae/workspace/grpdapp.R' AS
  VARCHAR(64)))) AS ae_output_t;
```

which should produce output similar to:

```
   A    |   B    |   C   |   D
        +-------+-------+-------
      | 2.974 | 5.552 | 2.026
5.006 | 3.428 | 1.462 | 0.246
5.936 | 2.77  | 4.26  | 1.326
(3 rows)
```

The same results can be achieved with the following query:

```
SELECT ae_output_t.* FROM (SELECT
  ROW_NUMBER() OVER (PARTITION BY species ORDER BY sepal_length) AS
        rn, COUNT(*) OVER (PARTITION BY species) AS ct,
        iris.*
  FROM iris) AS input_t,
  TABLE WITH FINAL
(nzr..r_udtf_any(rn, ct, sepal_length, sepal_width, petal_length, petal_width,
  CAST('CODE_PLAIN="return(mean(x))", MODE=groupedapply,
  SHAPER_LIST="a=NZ.DOUBLE,b=NZ.DOUBLE,c=NZ.DOUBLE,
d=NZ.DOUBLE"'
AS VARCHAR(128)))) AS ae_output_t;
```

Communication Channels

The following sections are valid for UDFs and UDTFs. UDA (aggregation) uses a different mechanism for passing user data.

Communication channel is a term describing a method of passing the user-provided code, such as the nz.fun object, to the R Language Adapter. The R Language Adapter design assumes that the code to be run is always passed from outside of the R interpreter. A number of channels have therefore been implemented to allow the code to be passed.

The channels allow code to be

passed by a file located in a NFS directory accessible from all R sessions

passed by a base64-encoded string

Each channel is identified with its predefined name and passed to the R AE through the Dynamic Environment. In this document, it is assumed that Dynamic Environment means passing values in the last parameter of a UDX.

The variety of channels results from different scenarios in which the R Language Adapter might be used on Netezza, such as a pre-registered UDX (*filesystem file*), code sent from R GUI (*workspace file*), or R code embedded in SQL queries (*plain string*).

Standard Objects

When working with R on Netezza, the user must define a number of objects. The list below describes the objects required in the source files when the compilation step takes place. For each channel a specific subset is defined, for example, note that the nz. prefix is omitted.

mode—a character variable that determines the working mode; it can take one of five values: **apply**, **tapply**, **run**, **install**, **groupedapply**

fun—a function object containing the actual user code; in most cases the input data is passed as the first argument under the name of **x**

args—additional arguments for **fun**

shaper—either the shaper function (see “Shapers & Sizers”) or the character value **std**, which means the standard shaper function

shaper.args—additional shaper arguments

shaper.list—table signature for the standard shaper function

cols—the names of the input columns

file—CRAN package name

Workspace File

The Workspace file channel is the default when Netezza is accessed from an R client[8]. The data file is

[8] Note that the R client can be run in many locations, including Netezza itself. The only requirement is ODBC access to the Netezza server. Regardless of where the client is running, user code is transferred via RODBC and put in the workspace

created automatically when R AE is invoked using one of the nzLibrary for R functions. This channel is therefore not designed for manual access.

In this channel the file name is stored in the WORKSPACE_PATH environment variable; the file itself should be located in the workspace folder, which for the R Language Adapter is ${NZ_EXPORT_DIR}/ae/workspace/nz/r_ae. This file must contain a serialized list with the following elements:

required: **mode**, **fun**, **args**

optional: **shaper**, **shaper.args**, **columns**, **shaper.list**, **file**

To create a file, you might invoke the following in R, which assumes that NZ_EXPORT_DIR is equal to /nz/export):

```
x <- list(mode='run', args=list(), fun=function(){
   getNext(); setOutput(0,'output value'); outputResult()})
output <- file('/nz/export/ae/workspace/nz/r_ae/raedata','w')
serialize(x, output, ascii=TRUE)
close(output)
```

To invoke the function, run the following SQL query:

```
SELECT * FROM TABLE WITH FINAL(nzr..r_udtf('WORKSPACE_PATH=raedata'));
```

which results in:

```
 COLUMNID |     VALUE
----------+--------------
         0 | output value

(1 row)
```

Filesystem File

The Filesystem file channel is very similar to the Workspace channel. The difference is that in this case the environment variable is named ABSOLUTE_PATH and contains the full file path instead of just the file name.

```
x <- list(mode='run', args=list(), fun=function(){
  getNext(); setOutput(0,'output value'); outputResult()})
output <- file('/home/nz/raedata','w')
serialize(x, output, ascii=TRUE)
close(output)
```

Invoking this query:

```
SELECT * FROM TABLE WITH FINAL(nzr..r_udtf('ABSOLUTE_PATH=/home/nz/raedata'));
```

results in the same output:

```
 COLUMNID |     VALUE
----------+--------------
         0 | output value

(1 row)
```

directory in the same way.

Plain R File

The Plain R file channel has been designed to work interactively with R and SQL. The user function and any additional data are stored in a plain-text file and directly named by the PLAIN_PATH environment variable. The difference is that the object names must contain the prefix **nz.**, that is, "nz*dot*."

The following objects are supported:

required: **nz.mode** (defaults to *run*), **nz.fun**

optional: **nz.shaper**, **nz.shaper.args**, **nz.cols**, **file**

The following listing is an example of a plain R file:

```
nz.mode <- 'run'
nz.fun  <- function() {
  getNext()
  setOutput(0,'output value')
  outputResult()
}
```

If saved as /home/nz/plain_r this code can be invoked with the following query:

```
SELECT * FROM TABLE WITH FINAL(nzr..r_udtf('PLAIN_PATH=/home/nz/plain_r'));
```

which results in:

```
 COLUMNID |      VALUE
          +--------------
----------0| output value
(1 row)
```

Serialized String

The Serialize string channel is similar to the Workspace file channel. In this case the user code is passed directly as the value of the CODE_SERIALIZED environment variable, without an intermediate file in the workspace directory. The file contains a serialized list and the string variable is also base64-encoded.

This list elements are:

required: **mode**, **fun**, **args**

optional: **shaper**, **shaper.arg***s*, **columns**, **shaper.list**, **file**

The R code creating the contents of the CODE_SERIALIZED variable requires the *caTools CRAN* package:

```
x <- list(mode='run', args=list(), fun=function(){
  getNext(); setOutput(0,'output value'); outputResult()})
base64encode(rawToChar(serialize(x, NULL, ascii=TRUE)))
```

The SQL query is presented below. Note that *backslash* signs ("\") denote a line continuation and should be omitted when pasting to nzsql:

```
SELECT * FROM TABLE WITH
FINAL(nzr..r_udtf('CODE_SERIALIZED=QQoyCjEzMzYzMwoxMzE\
4NDAKNTMxCjMKMTYKMQo5CjMKcnVuCjE5CjAKMTUzOQoxMDI2CjEKO\
Qo2CnNvdXJjZQoxNgoxCjkKNjYKZnVuY3Rpb24oKXtnZXROZXh0KCk\
7XDA0MHNldE91dHB1dCgwLFwnb3V0cHV0XDA0MHZhbHVlXCcpO1wwN\
```

```
DBvdXRwdXRSZXN1bHQoKX0KMjU0CjI1MwoyNTQKNgoxCjkKMQp7CjI\
KNgoxCjkKNwpnZXROZXh0CjI1NAoyCjYKMQo5CjkKc2V0T3V0cHV0C\
jIKMTQKMQowCjIKMTYKMQo5CjEyCm91dHB1dFwwNDB2YWx1ZQoyNTQ\
KMgo2CjEKOQoxMgpvdXRwdXRSZXN1bHQKMjU0CjI1NAoxMDI2CjEKO\
Qo1Cm5hbWVzCjE2CjMKOQo0Cm1vZGUKOQo0CmFyZ3MKOQozCmZ1bgo\
yNTQKAA=='))
```

When passed in a SQL console should once again result in:

```
 COLUMNID |      VALUE
----------+--------------
----------0 | output value
(1 row)
```

Plain String

The Plain string channel is designed for scenarios where R code must be embedded in SQL. The default running mode is **run***;* however, it can be overridden by setting the MODE environment variable. The code is passed as the value of the CODE_PLAIN environment variable. An additional environment variable, SHAPER_LIST, can be used to pass also the output signature.

The SQL query is presented below. Note that *backslash* signs ("\") denote a line continuation and should be omitted when pasting to nzsql:

```
SELECT * FROM TABLE WITH    FINAL(nzr..r_udtf('CODE_PLAIN=\
"getNext();setOutput(0,''output value'');outputResult()"'))
```

The result matches previous examples:

```
 COLUMNID |      VALUE
----------+--------------
----------0 | output value
(1 row)
```

The following example specifies the output signature:

```
SELECT * FROM TABLE WITH FINAL(nzr..r_udtf('CODE_PLAIN=
"getNext();setOutput(0,''output value'');outputResult()",
SHAPER_LIST="value=list(NZ.VARIABLE,32)"'))
```

which results in the same output as previous examples:

```
    value
--------------
 output value
(1 row)
```

The SHAPER_LIST environment variable should contain the comma-separated entries in the form *column name=data type*. In case of character columns (variable and fixed) the *data type* must take the following form: list(NZ.DATATYPE,length); for other data types it is the data type identifier only.

Standard UDXs

The standard R Language Adapter installation includes a number of UDXs that can be used in any user code. The client-side nzLibrary for R package assumes them to be registered in the nzLibrary for R database. These are:

r_udtf—a standard entry point for R AEs; set to *sparse* output type and returning a two-column table: TABLE(columnid INT4, value VARCHAR(16000))

r_udtf_any—a similar UDTF but returning TABLE(ANY), which requires a shaper function for this UDTF to run correctly

r_uda—an aggregate; currently designed to be used from the nzLibrary for R client package

placefile, **createfile**, and **appendfile**—UDXs used by the nzLibrary for R client package to set up data files with user-defined function in the workspace directory

Control Flow

Topics found in this section are described also in Manual Pages of the R-extension package nzrserver, available with IBM Netezza Analytics.

The R Language Adapter consists of the R interpreter, the R-to-Netezza connection layer that includes an R-side package called *nzrserver* and Netezza *Analytic Executables* shared libraries, and a number of R-implemented high-level functions.

Each time a SQL query referencing the R Language Adapter is invoked, a new R session is started, which starts the R interpreter, loads the necessary packages—at a minimum *nzrserver* with its dependencies—reads the user-provided R code and runs the code.

R AE execution may be performed in one of the following ways:

The R AE is executed on Host, as a single process as follows:

```
SELECT * FROM TABLE WITH FINAL(nzr..r_udtf(1, 2.0,
'text', 'WORKSPACE_PATH=rfile'));
```

This processing occurs when

Only literals, that is constant values, are passed to the UDX

One or more table columns are passed to the UDX. but the optimizer gathers the whole table on Host to speed up the execution

A number of R sessions are started on all SPUs (slave nodes) and each R integer instance operates only on this part of the input table which is located on the respective SPU

```
SELECT * FROM input_table,TABLE WITH FINAL(nzr..r_udtf(input_table.col_a,
input_table.col_b, 'WORKSPACE_PATH=rfile'));
```

This processing occurs when an actual table is passed to the UDX and optimizer decides to run the query in parallel.

The process is started once for the whole partition, that is, the part of data located on a *dataslice*), and not separately for each row. This means that sharing and aggregating data across rows is possible.

An Analytic Executable process is expected to acquire the desired number of input rows, from zero to the size of its partition, and output:

exactly one value in UDF mode,

from zero to any number of rows in UDTF mode

The data flow is controlled mainly by the getNext function, which returns **TRUE** when an input row is available for processing.

```
while(getNext()) {  # process input data  # return some output}
```

Error Handling

Error handling is based on the standard R approach, using the stop() function. This function throws an exception to be addressed at the end of the R function stack, where it is passed to NPS via the userError() function, which is part of the AE API. The userError() function can also be called directly, resulting in an immediate R process exit, and the error message being passed to NPS.

APPENDIX C
Notices and Trademarks

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR

IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:
IBM Corporation
26 Forest Street
Marlborough, MA 01752 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only. This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

> (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. Copyright IBM Corp. (enter the year or years). All rights reserved.

Trademarks

IBM, the IBM logo, ibm.com and Netezza are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies:

Adobe is a registered trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

NEC is a registered trademark of NEC Corporation.

UNIX is a registered trademark of The Open Group in the United States and other countries.

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

Red Hat is a trademark or registered trademark of Red Hat, Inc. in the United States and/or other countries.

D-CC, D-C++, Diab+, FastJ, pSOS+, SingleStep, Tornado, VxWorks, Wind River, and the Wind River logo are trademarks, registered trademarks, or service marks of Wind River Systems, Inc. Tornado patent pending.

APC and the APC logo are trademarks or registered trademarks of American Power Conversion

Corporation.

Other company, product or service names may be trademarks or service marks of others.

Regulatory and Compliance

Regulatory Notices

Install the NPS system in a restricted-access location. Ensure that only those trained to operate or service the equipment have physical access to it. Install each AC power outlet near the NPS rack that plugs into it, and keep it freely accessible. Provide approved circuit breakers on all power sources.

Product may be powered by redundant power sources. Disconnect ALL power sources before servicing. High leakage current. Earth connection essential before connecting supply. Courant de fuite élevé. Raccordement à la terre indispensable avant le raccordement au réseau.

Homologation Statement

This product may not be certified in your country for connection by any means whatsoever to interfaces of public telecommunications networks. Further certification may be required by law prior to making any such connection. Contact an IBM representative or reseller for any questions.

FCC - Industry Canada Statement

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to part 15 of the FCC rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio-frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference, in which case users will be required to correct the interference at their own expense.

This Class A digital apparatus meets all requirements of the Canadian Interference-Causing Equipment Regulations.

Cet appareil numérique de la classe A respecte toutes les exigences du Règlement sur le matériel brouilleur du Canada.

CE Statement (Europe)

This product complies with the European Low Voltage Directive 73/23/EEC and EMC Directive 89/336/EEC as amended by European Directive 93/68/EEC.

Warning: This is a class A product. In a domestic environment this product may cause radio interference in which case the user may be required to take adequate measures.

VCCI Statement

この装置は、情報処理装置等電波障害自主規制協議会 (VCCI) の基準に基づくクラスＡ 情報技術装置です。この装置を家庭環境で使用すると電波妨害を引き起越すことがあります。この場合には使用者が適切な対策を講ずるう要求されることがあります。