Manipulating data with regular expressions (regexp.1.0)

Jean Anderson
Chris Bosch

February 01, 2006
(First published December 21, 2001)

Creates an Informix® DataBlade™ with UDRs that search and modify character and clob data using regular expressions. Includes source code.

© 2001 International Business Machines Corporation. All rights reserved.

Introduction

A regular expression is a sequence of literal characters and of metacharacters that lets you perform complex text search and modification. The UNIX grep, egrep, awk, sed utilities are examples that use regular expressions to manipulate text data.

For a comprehensive description and history of regular expressions, see Mastering Regular Expressions by Jeffrey E. F. Friedl (O'Reilly & Associates, Inc., 1997).

University of Toronto regexp library

Henry Spencer at the University of Toronto re-implemented the regexp(3) library so C programs can use egrep-like regular expressions. The University of Toronto makes that source code available with the following copyright notice:

Copyright (c) 1986 by University of Toronto.
Written by Henry Spencer. Not derived from licensed software.

Permission is granted to anyone to use this software for any purpose on any computer system, and to redistribute it freely, subject to the following restrictions:

1. The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from defects in it.

2. The origin of this software must not be misrepresented, either by explicit claim or by omission.

3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software.

Chris Bosch of Informix Software modified the University of Toronto regexp library to implement the UDRs in this DataBlade project. The source code identifies altered code.
The `regexp` man page is included in the source code distribution for this project.

**Regexp metacharacters**
The `regexp` library implements `egrep`-like capabilities and supports the following metacharacters:

<table>
<thead>
<tr>
<th>Regexp Metacharacters</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Beginning of line</td>
</tr>
<tr>
<td>$</td>
<td>End of line</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[abc]</td>
<td>Match any character enclosed in the brackets.</td>
</tr>
<tr>
<td>[^abc]</td>
<td>Match any character not enclosed in the brackets.</td>
</tr>
<tr>
<td>[a-z]</td>
<td>Match the range of characters specified by the hyphen.</td>
</tr>
<tr>
<td>.</td>
<td>Match any single character.</td>
</tr>
<tr>
<td>( )</td>
<td>Group the regular expression within the parentheses.</td>
</tr>
<tr>
<td>?</td>
<td>Match zero or one of the preceding expression.</td>
</tr>
<tr>
<td>*</td>
<td>Match zero, one, or many of the preceding expression.</td>
</tr>
<tr>
<td>+</td>
<td>Match one or many of the preceding expression.</td>
</tr>
<tr>
<td>\</td>
<td>Escape the metacharacter that follows so that the literal meaning of the character is used.</td>
</tr>
<tr>
<td>&amp;</td>
<td>Reference the entire matched text for string substitution.</td>
</tr>
<tr>
<td>\n</td>
<td>Reference subgroup \n within the matched text (\n can be 1-9).</td>
</tr>
</tbody>
</table>

If you are on a UNIX system, your local `egrep` man page will be helpful.

**Regexp limitations**
The `regexp` library does not support the following features that are found in some versions of `egrep`:

- **Word boundaries**
  The `regexp` library does not support the `\<` begin word and `\>` end word metasequences. Examples in this tech note suggest ways to implement your own word boundary support.

- **Back references**
  The `regexp` library does not support the `\n` notation, which lets you reference a previously matched group.

- **Repetitions**
  The `regexp` library does not support the `{min,max}` metasequence that specifies a minimum and maximum repetition of a regular expression.

**Software Requirements**
`regexp.1.0` comes precompiled for Solaris and Windows 2000. To run it, you need Informix Dynamic Server, release 9.x. If you want to build `regexp.1.0`, you need a C compiler to build the shared object.
regexp.1.0 was tested with the software releases listed below:

SUN:
- Solaris 2.7.1
- IDS 9.21.UC4
- SUNpro C Compiler

Windows 2000:
- Windows 2000, 5.0, service pack 1
- IDS 9.21.TC3
- Microsoft Visual C++ 6.0

**Getting Started**

**Download the Distribution**
Download the regexp.1.0.tar.Z compressed tar archive.

Uncompress it:

```bash
uncompress regexp.1.0.tar
```

Extract the tar archive:

```bash
tar xvf regexp.1.0.tar
```

On Windows 2000 use **WinZip**, MKS Toolkit, or any other utility that can extract compressed UNIX tar files.

The distribution is extracted into a directory hierarchy under regexp.1.0. The directory contents are summarized in the table below.

**Table 2. Directory contents**

<table>
<thead>
<tr>
<th>Directory</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>README</td>
<td>This file.</td>
</tr>
<tr>
<td>demo/</td>
<td>Sample SQL statements.</td>
</tr>
<tr>
<td>doc/</td>
<td>Contains the Toronto regexp man page and copyright.</td>
</tr>
<tr>
<td>functest/</td>
<td>Functional tests for the UDRs.</td>
</tr>
<tr>
<td>scripts/</td>
<td>SQL registration scripts for BladeManager.</td>
</tr>
<tr>
<td>src/</td>
<td>Source code.</td>
</tr>
<tr>
<td>regexp.ibs</td>
<td>BladeSmith project generated with BladeSmith version 4.00.TC2. This project is used for managing SQL registration scripts.</td>
</tr>
</tbody>
</table>

**Build the DataBlade module**

This distribution includes the DataBlade module built for Solaris 2.7.1 and for Windows 2000 on IDS 9.21:
The Solaris shared object is in the src/solaris-sparc subdirectory.

The Windows 2000 dynamic link library (DLL) is in the src/Win2K subdirectory.

If you are on Solaris or Windows 2000 you do not need to build the DataBlade module. You can proceed to the "Install the DataBlade module" step below.

If you are not on Solaris or Windows 2000, or if you modify the source code, then you must compile the DataBlade for your platform. If you do not know how to use a C compiler, then you should probably get a consultant to help you do this. If you are familiar with compiling, follow the instructions below, making any required modifications for your platform.

UNIX

Set the TARGET variable to the platform-specific include file for your platform, which you will find in $INFORMIXDIR/incl/dbdk. The following sample usage is for the C-Shell on Solaris:

```bash
setenv TARGET $INFORMIXDIR/incl/dbdk/makeinc.solaris
```

Next, build the DataBlade module. The regexp project comes with a Solaris SUNWspro makefile called regexp.mak, and a GNU makefile called sol-gcc.mak that has only been verified for Solaris. Both are in the src subdirectory:

- SUNWspro: make -f regexpU.mak
- GNU (gcc): make -f sol-gcc.mak

Disregard warnings that mi_*, ifx_*, and gl_* symbols could not be resolved. The IDS server resolves these symbols at runtime.

Detailed instructions for building a UNIX shared object library are included in the DataBlade Developer's Kit User's Guide.

Windows 2000

src/regexp.mak was generated from a Microsoft DevStudio project, and builds the DataBlade DLL at the DOS prompt. You can build a debug or release version:

- nmake /f regexp.mak CFG="regexp - Win32 Debug"
- nmake /f regexp.mak CFG="regexp - Win32 Release"

The resulting DLL gets output to Debug\regexp.bld (remember this location later when you need to install the DLL).

You can also load regexp.1.0 source code into Microsoft DevStudio by following the instructions in the Building a DataBlade DLL with Microsoft DevStudio tech note. It explains how to create a new project and add the *.c, *.h, and *.def files to that project.

Note: BladeSmith was not used to manage regexp.1.0 source code. So do not try to load the regexp.1.0 source code into DevStudio by clicking on the Bladesmith MSDev button.
Install the DataBlade module

Login as user informix, change to the regexp.1.0 directory into which the distribution was extracted, then follow the steps outlined below for your platform.

SUN Solaris

Note: if you compiled the DataBlade on another UNIX platform, you won't find regexp.bld in a subdirectory called solaris-sparc. Instead, the DataBlade object will be in a subdirectory named for that platform.

```
mkdir $INFORMIXDIR/extend/regexp.1.0
cp ./scripts/* $INFORMIXDIR/extend/regexp.1.0
cp ./src/solaris-sparc/regexp.bld $INFORMIXDIR/extend/regexp.1.0
```

Windows 2000

```
mkdir %INFORMIXDIR%\extend\regexp.1.0
cp .\scripts\* %INFORMIXDIR%\extend\regexp.1.0
cp .\src\Win2K\regexp.bld %INFORMIXDIR%\extend\regexp.1.0
attrib +R %INFORMIXDIR%\extend\regexp.1.0\regexp.bld
```

If you compiled the DLL yourself, remember that the result will be in a subdirectory called Debug or Release, depending on how you built it. The Win2K subdirectory contains the DLL provided with this software distribution.

Create the regexp VPCLASS

The University of Toronto regexp code uses global variables that get updated. To make this code safe for the multi-threaded IDS environment, the Regexp SQL routines run in a user-defined virtual processor class called "regexp". To create this VPCLASS, put the following entry in your ONCONFIG file:

```
VPCLASS regexp,nyield  # Regexp bladelet uses this user-defined vp
```

The VPCLASS must be noyield because global variables in the source code get updated.

You can improve concurrency for multiple users on UNIX machines by creating more than one VPCLASS, like this:

```
VPCLASS regexp,nyield,num=3
```

However, you must not create more than 1 regexp VPCLASS on NT or Windows 2000. Since each virtual processor runs as a thread, multiple virtual processors of the same regexp VPCLASS would share the same global variables.

If the VPCLASS has not been created, the SQL routines will fail and the online.log file will have errors that look like this:

```
13:21:44 (-9799): ERROR: Routine execution VP switch failed -- new VP <regexp>
```
Register the DataBlade module in a database

A database has to already exist before you can register a DataBlade module in it. This example assumes you created a database named `demodb`.

At the UNIX or DOS prompt, use the `blademgr` program to register the DataBlade module:

```
blademgr
shm> register regexp.1.0 demodb
```

You can also run the BladeManager GUI from Windows 2000.

Troubleshooting Problems

SQL routines fail to execute

If the `regexpVPCLASS` has not been created, the SQL routines will fail and the `online.log` file will have errors that look like this:

```
13:21:44 (-9799): ERROR: Routine execution VP switch failed -- new VP <regexp>
```

The section above titled Create the regexp VPCLASS explains how to setup the `regexpVPCLASS`.

DataBlade compile

Windows 2000 requires all symbols to be resolved at compile/build time. So if a symbol is unresolved, it is a fatal error that must be corrected before you attempt to install and register the DataBlade module. DataBlade modules on Windows 2000 link a library called sapi.lib that resolves Informix server symbols.

On most UNIX platforms, symbols resolved by the Informix server are not resolved at compile/build time, so you might see compilation errors or warnings. The UNIX makefile executes a `filtersym.sh` script to try to eliminate warnings for symbols that get resolved at runtime, but this list is incomplete. So, if the compile and `ld` succeeds with warnings, go ahead and install the DataBlade and run the functional tests in the `functest` subdirectory to verify that everything is OK.

The DataBlade developers for this project would like to be able to verify a DataBlade module on each and every platform, but unfortunately does not have resources to do so. If you need to compile this DataBlade for a different platform and have not done so before, consider getting help from a consultant who is experienced with that platform.

DataBlade Registration

If DataBlade registration fails, check the BladeManager log files for details.

The following error means that BladeManager cannot find the software under `$INFORMIXDIR/extend`:

```
blademgr: ERROR: DataBlade module regexp.1.0 not on server.
```
Go back and doublecheck the section on "Install the DataBlade module".

If BladeManager still can’t find the DataBlade distribution, stop and restart the server.

If BladeManager hangs, try registering one of the DataBlade modules that comes with the server -- you will find them in $INFORMIXDIR/extend. If BladeManager still hangs, you need to contact Informix tech support.

Assistance Resolving Problems

Do not call Informix tech support regarding a problem with or question about the regexp DataBlade module. Only call Informix tech support for problems with DataBlade modules that come with the server or are purchased from Informix.

User-Defined Routines (UDRs)

This section describes each SQL routine that the regexp DataBlade module creates.

### Table 3. SQL routines

<table>
<thead>
<tr>
<th>SQL routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>regexp_extract</td>
<td>Return a list of strings that match a regular expression from the source string.</td>
</tr>
<tr>
<td>regexp_match</td>
<td>Return TRUE if a source string matches the regular expression.</td>
</tr>
<tr>
<td>regexp_replace</td>
<td>Match a regular expression in a string and replace it with something else.</td>
</tr>
<tr>
<td>regexp_split</td>
<td>Splits a string into substrings, using the regular expression as the delimiter.</td>
</tr>
<tr>
<td>TraceSet_regexp</td>
<td>Enable tracing for regexp routines.</td>
</tr>
</tbody>
</table>

Sample usage sometimes uses the following test table:

```sql
create table tongue_twisters
(
    id    int,
    twister  lvarchar
);
```

Some examples also assume that the following data from the 1st Intern. Collection of Tongue Twisters - English is loaded into the tongue_twisters table:

```sql
select id, twister
from   tongue_twisters
order by id;
```

<table>
<thead>
<tr>
<th>id</th>
<th>twister</th>
</tr>
</thead>
<tbody>
<tr>
<td>246</td>
<td>Sally sells sea shells by the sea shore. But if Sally sells sea shells by the sea shore then where are the sea shells Sally sells?</td>
</tr>
<tr>
<td>278</td>
<td>Peter Piper picked a peck of pickled peppers. A peck of pickled peppers Peter Piper picked. Peter Piper picked a peck of pickled peppers</td>
</tr>
</tbody>
</table>
peppers, Where's the peck of pickled peppers Peter Piper picked?

id    286
twister  If two witches would watch two watches, which witch would watch which watch?

id    301
twister  Fuzzy Wuzzy was a bear, Fuzzy Wuzzy had no hair, Fuzzy Wuzzy wasn't very fuzzy, was he?

id    306
twister  I slit a sheet, a sheet I slit, and on that slitted sheet I sit.

id    313
twister  Betty bought some bitter butter and it made her batter bitter, so Betty bought some better butter to make her bitter batter better.

id    335
twister  How much wood could a woodchuck chuck if a woodchuck could chuck wood? A woodchuck could chuck as much wood as a woodchuck would chuck if a woodchuck could chuck wood.

id    361
twister  She sells seashells on the seashore. The seashells she sells are seashore seashells.

**regexp_extract**

**Description**

*regexp_extract* steps through a source string and iteratively returns each substring that matches a regular expression. An optional integer argument limits the number of substrings returned.

**Syntax**

regexp_extract(lvarchar, lvarchar) returns lvarchar with (iterator)  
regexp_extract(lvarchar, lvarchar, integer) returns lvarchar with (iterator)

**Table 4. Syntax and arguments**

<table>
<thead>
<tr>
<th>Number</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>lvarchar</td>
<td>The source string to be searched.</td>
</tr>
<tr>
<td>2.</td>
<td>lvarchar</td>
<td>The regular expression to use for the search.</td>
</tr>
<tr>
<td>3.</td>
<td>lvarchar</td>
<td>An optional argument that limits the number of substrings returned. If not specified, all substrings that match the pattern are returned.</td>
</tr>
</tbody>
</table>

**Example**

The goal of this example is to look for each occurrence of "would" or "wood" in the following tongue twister, then output the complete surrounding word (we want the ten words highlighted below):

How much wood could a woodchuck chuck if a woodchuck could chuck wood?  
A woodchuck could chuck as much wood as a woodchuck would chuck if a woodchuck could chuck wood.

This regular expression matches both **wood** and **would**:
wo[ou]l?d

If we use this regular expression, `regexp_extract` shows that the two words occurs ten times in the string, but there isn't any word context that helps spot where it occurs:

```java
execute function
regexp_extract(
    'How much wood could a woodchuck chuck if a woodchuck could chuck wood?
     A woodchuck could chuck as much wood as a woodchuck would chuck
     if a woodchuck could chuck wood.);

woul?d')

10 row(s) retrieved.
```

We can expand the regular expression to include the entire word. The `regexp` library does not include direct word boundary support, so we'll implement our own. First, specify that more lowercase characters may follow the primary subexpression:

```java
wo[ou]l?d[a-z]*
```

Next, specify that the word ends with a space or a punctuation character. For completeness, the example below includes more punctuation characters than actually occur in this tongue twister. Also, note that if a hyphen appears as the first character in a character class it means a literal hyphen, not a range of values.

```java
wo[ou]l?d[a-z]*[- .?!;]
```

Now when we execute `regexp_extract`, we get the actual word in which the pattern occurs:
execute function
regexp_extract('How much wood could a woodchuck chuck if a woodchuck could chuck wood?
A woodchuck could chuck as much wood as a woodchuck would chuck
if a woodchuck could chuck wood.','wo[ou]l?d[a-z]*[- .?!:;]');

(expression) wood
(expression) woodchuck
(expression) woodchuck
(expression) wood
(expression) woodchuck
(expression) wood
(expression) wood
(expression) woodchuck
(expression) wood
(expression) woodchuck

10 row(s) retrieved.

Specifying the optional last argument let us limit results to just the first two:

execute function
regexp_extract('How much wood could a woodchuck chuck if a woodchuck could chuck wood?
A woodchuck could chuck as much wood as a woodchuck would chuck
if a woodchuck could chuck wood.','wo[ou]l?d[a-z]*[- .?!:;]', 2);

(expression) wood
(expression) woodchuck

2 row(s) retrieved.

What if we wanted to include the start of the word in the regular expression? The start could be a space or the beginning of the line, ( |^), followed by upper or lowercase characters:

( |^)[A-Za-z]*

To illustrate with an example, let's look for all occurrences of tter in this tongue twister:

Betty bought some bitter
butter and it made her batter bitter,
so Betty bought some better
butter to make her bitter batter better.

And here is the query:
execute function
regexp_extract(
    'Betty bought some bitter butter and it made her batter bitter,
    so Betty bought some better butter to make her bitter batter better.',
    '( |^)[A-Za-z]*tter'
);

(expression)   bitter
(expression)   butter
(expression)   batter
(expression)   bitter
(expression)   better
(expression)   butter
(expression)   bitter
(expression)   batter
(expression)   better
9 row(s) retrieved.

If the regular expression results in a 0-length match, the query returns "No rows found.". For example, even though `regexp_match` returns `TRUE` for a match on begin-line (`'^'`), `regexp_extract` returns no rows because `'^'` matches something that is 0-length:

> execute function regexp_match('Hello world', '^');

(expression)
   t
1 row(s) retrieved.

> execute function regexp_extract('Hello world', '^');

No rows found.

See Also
See also `regexp_split`.

`regexp_match`  
Description
`regexp_match` returns `TRUE` if the source string matches the regular expression; otherwise, it returns `FALSE`.

Syntax
`regexp_match(lvarchar, lvarchar)` returns boolean
`regexp_match(clob, lvarchar)` returns boolean

Table 5. Syntax and arguments for `regexp_match`

<table>
<thead>
<tr>
<th>Number</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>lvarchar</code> or <code>clob</code></td>
<td>The source string to be searched.</td>
</tr>
<tr>
<td>2.</td>
<td><code>lvarchar</code></td>
<td>Regular expression.</td>
</tr>
</tbody>
</table>
Example

`regexp_match` can be executed directly with the `EXECUTE FUNCTION` statement. For example, this query tests if the word "bladelet", "Bladelet", or "DataBlade" occurs anywhere in a string:

```sql
execute function regexp_match
('Regexp DataBlade module', '[Bb]ladelet|DataBlade');
```

In this next example, the regular expression "wo[ou]l?d" matches both "wood" and "would":

```sql
select id, twister
from   tongue_twisters
where  regexp_match(twister, 'wo[ou]l?d');
```

Example

`regexp_replace` can be used either directly or in an SQL statement. The following examples demonstrate both forms.
Let's say you have a web-based search engine and you want to return search matches with the match in bold, using the `<b>` and `<i>` HTML tags. Furthermore, you want to make the entire word bold in which the match was found. The regular expression in the example below looks for a word in which "she" or "She" occurs, then replaces the matched text with itself (&), enclosed by `<b>` and `<i>` HTML tags:

execute function regexp_replace(
   'She sells seashells on the seashore. The seashells she sells are seashore seashells.',
   '( |^)[A-Za-z]*[Ss]he[a-z]*[ .,$]',
   '&');

(expression) She sells seashells on the seashore. The seashells she sells are seashore sea-shells.

The result displayed on a web page would look like this:

She sells seashells on the seashore. The seashellsshe sells are seashore seashells.

We could restrict the number of matches replaced by using the optional integer argument:

execute function regexp_replace(
   'She sells seashells on the seashore. The seashells she sells are seashore seashells.',
   '( |^)[A-Za-z]*[Ss]he[a-z]*[ .,$]',
   '&',
   3);

The result displayed on a web page would now look like this:

She sells seashells on the seashore. The seashells she sells are seashore seashells.

We can also use `regexp_replace` in a SQL statement:

```
select id,
   regexp_replace(twister, '( |^)[A-Za-z]*[Ss]he[a-z]*[ .,$]', '&')
from   tongue_twisters
where  regexp_match(twister, '[Ss]he');
```

<table>
<thead>
<tr>
<th>id</th>
<th>twister</th>
</tr>
</thead>
<tbody>
<tr>
<td>306</td>
<td>I slit a sheet, a sheet I slit,</td>
</tr>
<tr>
<td></td>
<td>and on that slitted sheet I sit.</td>
</tr>
<tr>
<td>361</td>
<td>She sells seashells on the seashore.</td>
</tr>
<tr>
<td></td>
<td>The seashellsshe sells are seashore seashells.</td>
</tr>
</tbody>
</table>

Web results would look like this:

<table>
<thead>
<tr>
<th>id</th>
<th>twister</th>
</tr>
</thead>
<tbody>
<tr>
<td>306</td>
<td>I slit a sheet, a sheet I slit,</td>
</tr>
<tr>
<td></td>
<td>and on that slitted sheet I sit.</td>
</tr>
<tr>
<td>361</td>
<td>She sells seashells on the seashore.</td>
</tr>
<tr>
<td></td>
<td>The seashellsshe sells are seashore seashells.</td>
</tr>
</tbody>
</table>

 Manipulating data with regular expressions (regexp.1.0)
Finally, we can reference up to nine subgroups within the matched text (\1 through \9), each subgroup bounded by parentheses:

```sql
> execute function regexp_replace ('swap me all around', '(.*)(.*) (.*)(.*)', '\4 \3 \2 \1');
(expression) around all me swap
1 row(s) retrieved.
```

```sql
> execute function regexp_replace ('swap me', '(.*) (.*)', '&: \2 \1');
(expression) swap me: me swap
1 row(s) retrieved.
```

**regexp_split**

**Description**

*regexp_split* splits a string into substrings, using the regular expression as the delimiter.

**Syntax**

```sql
regexp_split(lvarchar, lvarchar) returns lvarchar with (iterator)
regexp_split(lvarchar, lvarchar, integer) returns lvarchar with (iterator)
```

**Arguments:**

<table>
<thead>
<tr>
<th>Number</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>lvarchar</td>
<td>The source string to be searched.</td>
</tr>
<tr>
<td>2.</td>
<td>lvarchar</td>
<td>Regular expression.</td>
</tr>
<tr>
<td>3.</td>
<td>lvarchar</td>
<td>Optional argument that specifies the number of substrings to return. If not specified, all occurrences are returned.</td>
</tr>
</tbody>
</table>

**Example**

*regexp_split* and *regexp_extract* perform opposite actions.

In the next two queries, we're looking for the sequence "ick" and any characters that may precede it in a word:

```sql
( |^)[A-Za-z]*ick
```

*regexp_extract* returns each substring that matches the regular expression:
execute function
   regexp_extract(
     'Jack be nimble, Jack be quick, Jack jump over the candlestick.',
     '( |^)[A-Za-z]*ick'
   );

(expression) quick
(expression) candlestick

2 row(s) retrieved.

**regexp_split** splits a string into a list of substrings, using the regular expression as the delimiter:

execute function
   regexp_split(
     'Jack be nimble, Jack be quick, Jack jump over the candlestick.',
     '( |^)[A-Za-z]*ick')
);

(expression) Jack be nimble, Jack be
(expression) , Jack jump over the
(expression) .

3 row(s) retrieved.

A more likely use might be to split a string up into its separate words, using a space as the delimiter:

execute function
   regexp_split(
     'Jack be nimble, Jack be quick, Jack jump over the candlestick.',
     '
   );

(expression) Jack
(expression) be
(expression) nimble,
(expression) Jack
(expression) be
(expression) quick,
(expression) Jack
(expression) jump
(expression) over
(expression) the
(expression) candlestick.

11 row(s) retrieved.

The optional third argument lets you limit the number of substrings into which the source string is split:
execute function
    regexp_split(
        'Jack be nimble, Jack be quick, Jack jump over the candlestick.',
        ' ',
        5);

(expression) Jack
(expression) be
(expression) nimble,
(expression) Jack
(expression) be quick, Jack jump over the candlestick.

5 row(s) retrieved.

If the delimiter specified in the regular expression matches the entire source string, the query returns "No rows found." because there is nothing left to return:

> execute function regexp_split('Hello world', 'Hello world');
No rows found.

See Also

See also regexp_extract.

TraceSet_regexp

Description

TraceSet_regexp() enables the output of debug messages.

Syntax

TraceSet_regexp(lvarchar, int)

The lvarchar argument is the full pathname of the file to which you want the debug messages output.

The int argument is the level to which you want to set the trace class. Setting it to at least one of the values below results in the described behavior:

- 20: outputs UDR entry and exit messages
- 30: outputs any additional messages

Example

You must set the GLS LOCALE environment variables to en_us.8859-1 for messages to actually get output. The following example is for the UNIX C-Shell:

setenv DB_LOCALE en_us.8859-1
setenv SERVER_LOCALE en_us.8859-1
setenv CLIENT_LOCALE en_us.8859-1
The next command sets the `regexp` trace class level to 20 and specifies the output file for debug messages:

```sql
execute procedure TraceSet_regexp("/tmp/regexp.log", 20);
```

After executing this procedure, you should see output like the following in `/tmp/regexp.log`:

```
Tracing session: 23 on 04/04/2001
14:27:46  TraceSet_regexp: function exit.
```

For more information about this topic

- The man page that comes with the University of Toronto `regexp` distribution, `regexp.3`, is included in the `doc` subdirectory of this DataBlade module distribution.
- `egrep` documentation is also helpful.

Glossary

Terms and acronyms used by this tech note include:

**UDR**

User-defined routine.

**escape character**

An escape character restores the literal meaning of a metacharacter. The backslash (\) is a common escape character. For example, if you wanted to use the literal values of `^`, `$`, `*`, and `.`, you would escape them with the backslash like this: `\^`, `\$`, `\*`, and `\.``

**metacharacter**

A metacharacter is a character that is interpreted in a special way. For example, the `^` symbol means start of string (or line) and `$` means end of string (or line). Two of the most common metacharacters are the `.` (dot) wildcard, which matches any single character, and the `*` (asterisk) wildcard, which matches 0, one, or many characters (more on this one later).

**pattern**

A compiled regular expression.

**regular expression**

A sequence of characters, composed of metacharacters and literals, that is used in performing complex text manipulation.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>File for download</td>
<td>regexp.1.0.tar.Z</td>
<td>12KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2001, 2006

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)

Manipulating data with regular expressions (regexp.1.0)