GUIDE
SHARE
EUROPE
UK REGION

GSE UK Conference 2019
Dock into the Dark Side

Express yourself!

Marcus Davage

Lead Product Developer, AMI DevOps for Db2
BMC Software

November 2019

Session IO

A regular expression (regex*) is a sequence of characters that define a search pattern. They have been around since the '60s, but only recently have been adopted more ubiquitously across platforms. This session hopes to enlighten the user to the history, use and implementations of regular expressions, and how they can be used in Db2.

*(hard ‘g’, as in ‘get’, not soft, as in ‘gel’)

Agenda

- Introduction to regular expressions
- Terminology and Examples
- ISPF
- DB2
- Rexx
- Additional Information
- Conclusion

Introduction

- Regular expressions (regex) are a way of specifying a string for pattern matching.

Introduction

- Regular expressions (regex) are a way of specifying a string for pattern matching.
- Been around since the ’50s (maths theory), ’60s (Unix), in z/OS since 2.1

Introduction

- Regular expressions (regex) are a way of specifying a string for pattern matching.
- Been around since the ’50s (maths theory), ’60s (Unix), in z/OS since 2.1
- By using meta characters, regular expressions are a flexible and powerful way of specifying patterns.

Introduction

- Regular expressions (regex) are a way of specifying a string for pattern matching.
- Been around since the ’50s (maths theory), ’60s (Unix), in z/OS since 2.1
- By using meta characters, regular expressions are a flexible and powerful way of specifying patterns.
- They can be cryptic, non-intuitive, and time-consuming to create and debug.

Introduction

- Regular expressions (regex) are a way of specifying a string for pattern matching.
- Been around since the ’50s (maths theory), ’60s (Unix), in z/OS since 2.1
- By using meta characters, regular expressions are a flexible and powerful way of specifying patterns.
- They can be cryptic, non-intuitive, and time-consuming to create and debug.
- They can be CPU-intensive to process and, if poorly specified, can affect system performance.

Introduction

- Regular expressions (regex) are a way of specifying a string for pattern matching.
- Been around since the ’50s (maths theory), ’60s (Unix), in z/OS since 2.1
- By using meta characters, regular expressions are a flexible and powerful way of specifying patterns.
- They can be cryptic, non-intuitive, and time-consuming to create and debug.
- They can be CPU-intensive to process and, if poorly specified, can affect system performance.
- Different computing platforms and programming languages may implement slightly different flavours of regex (Python, Perl, Java, C#, ISPF, .NET, PHP, Db2 XML query…)

Usage in Unix

- egrep *'regular expression'* filename

```
mdavage@EM-mdavage-W1:~$ egrep 'Llanfairpwll' words
Llanfairpwllgwyngyll
Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch
```

- echo *'string'* | egrep *'regular expression'*

```
mdavage@EM-mdavage-W1:~$ echo 'tonight tonite toknight' | egrep 'toni(gh)?te?'
tonight tonite toknight
```

- Callable in programs using common libraries
  - C, C++, Java

Terminology (1)

Metacharacter	Description
^	Start of a line
$	End of a line
.	Matches any single character
[…]	A bracket expression – matches any single character within
[^…]	Matches any single character NOT contained within
(…)	A character group, or subexpression
|	“OR” – matches either expression it separates

Examples (1)

- `.at` matches any three-character string ending with “at”, including “hat”, “cat” and “bat”
- `d.g` matches any three-character string starting with “d” and ending with “g”
- `[hc]at` matches “hat” and “cat”
- `(h|c)at` matches “hat” or “cat”
- `[^b]at` matches all strings matched by .at except “bat”
- `[^hc]at` matches all strings matched by .at other than “hat” and “cat”
- `^[hc]at` matches “hat” and “cat”, but only at the beginning of the string or line
- `[hc]at$` matches “hat” and “cat”, but only at the end of the string or line

Terminology (2)

Metacharacter	Description
x?	‘x’ is optional
x+	‘x’ appears one or more times
x*	‘x’ appears zero or more times
x{m}	‘x’ appears ‘m’ times
x{m,n}	‘x’ appears between ‘m’ and ‘n’ times
x{m,}	‘x’ appears at least ‘m’ times

Examples (2)

- `s.*` matches ‘s’ followed by zero or more characters, for example: “s”, “saw”, “seed” and “sphygmomanometer”
- `[Qq]` matches any line with a ‘Q’ or a ‘q’
- `[Qq][^u]` matches any line with a ‘Q’ or a ‘q’ that is not followed by a ‘u’
  - E.g. “Iraqi” but not “Iraq”, as the “q” of “Iraq” is the last letter
- `^m[eaiy]{2,3}n$` matches “main”, “mean”, “mayan” but not “man”, “men”
- `^m[eaiy]{2}n$` matches “main”, “mean” but not “mayan”, “man”, “men”
- `July?` matches “July” and “Jul” (and also “Julienne”)
- `Colou?r` matches “Colour” and “Color” (for our American friends)

Terminology (3)

Metacharacter	Description
(…)	A character group, or subexpression
|	“OR” – matches either expression it separates
\1	Refers to previous 1st matching subexpression
\n	Refers to previous nth matching subexpression
\	Escape character (when you want to search for a metacharacter)

Examples (3)

- `[Qq]([^u]|$)` matches any line with a ‘Q’ or a ‘q’ that is not followed by a ‘u’ or is at the end of a line
  - Now matches “Iraqi” and “Iraq”
- `\[.\]` matches any single character surrounded by “[“ and “]” since the brackets are escaped, for example: “[a]” and “[b]”
- `([a-z])\1` matches any doubled letters
  - Matches “Jenni”, “Jennilee”, “Llangollen”, but not “Aardvark” or “Llanbedr”
- `\<([a-z]+) +\1\>` matches any doubled words
  - Matches “matches any any doubled words”

Terminology (4)

Metacharacter	Description
\d	Any digit
\D	Any non-digit
\w	Any alphanumeric
\W	Any non-alphanumeric
\s	Any whitespace
\S	Any non-whitespace

Metacharacter	Description
\t	A tab character
\n	A newline
\r	A carriage return
\b	A word boundary
\B	Not a word boundary
\<	Beginning of a word
\>	End of a word

Examples (4)

E.g. In the following text,

"Ganymede," he continued, "is the largest moon in the Solar System.“

‘`he`’ would match

*"Ganymede," **he** continued, "is t**he** largest moon in t**he** Solar System.“*

‘`\bhe`’ or ‘`\<he\>`’ would match

*"Ganymede," **he** continued, "is the largest moon in the Solar System."*

Regex in ISPF

and its implementation, idiosyncrasies, usage and examples

ISPF Idiosyncrasies

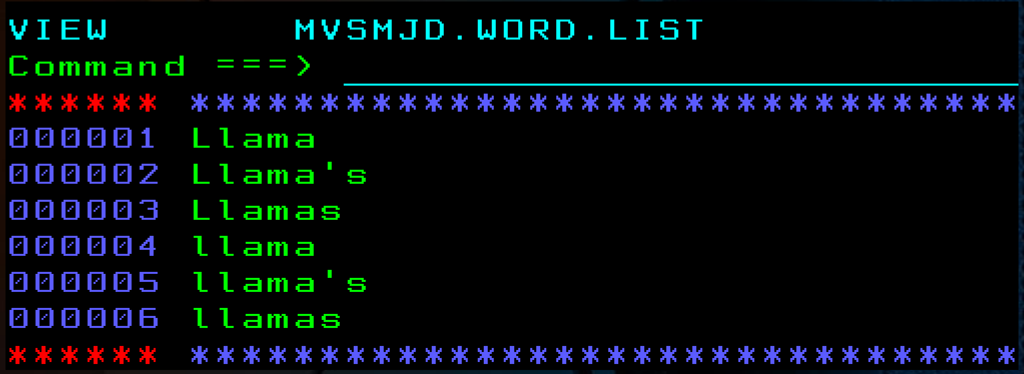
Non-existent Metacharacter	Functional Replacement
\d	[0-9]
\D	[^0-9]
\w	[a-zA-Z]
\W	[^a-zA-Z]
\s	[ ]
\b \< \>	[ ]+x[ ]+ look for ‘x’ between 1 or more spaces
$	$ does exist and work, but use [ ]*$ to trap zero or more trailing spaces

ISPF Usage

In EDIT or VIEW (but not BROWSE)

FIND options

F ‘*string*’	case insensitive
F T‘*text*’	case insensitive
F C‘*characters*’	case sensitive
F R‘*regular expression*’	case insensitive
F RC‘*regular expression*’	case sensitive

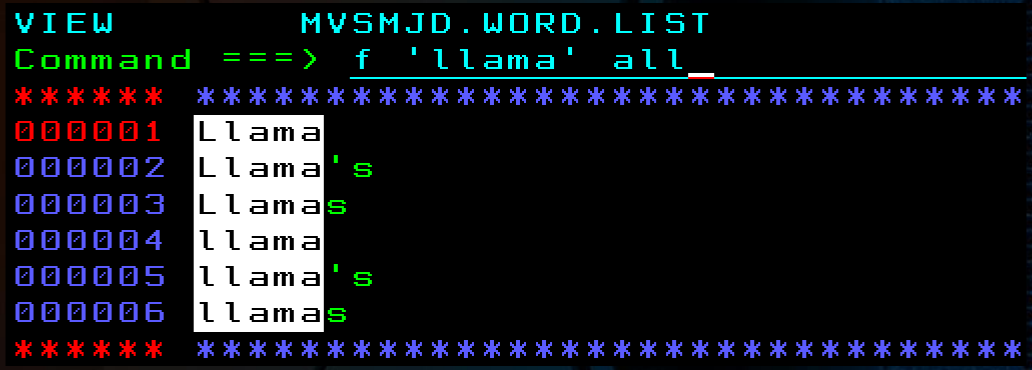


```
VIEW       MVSMJD.WORD.LIST
Command ===>
****** ********************************
000001 Llama
000002 Llama's
000003 Llamas
000004 llama
000005 llama's
000006 llamas
****** ********************************
```

ISPF Examples

F '*llama*'

case insensitive

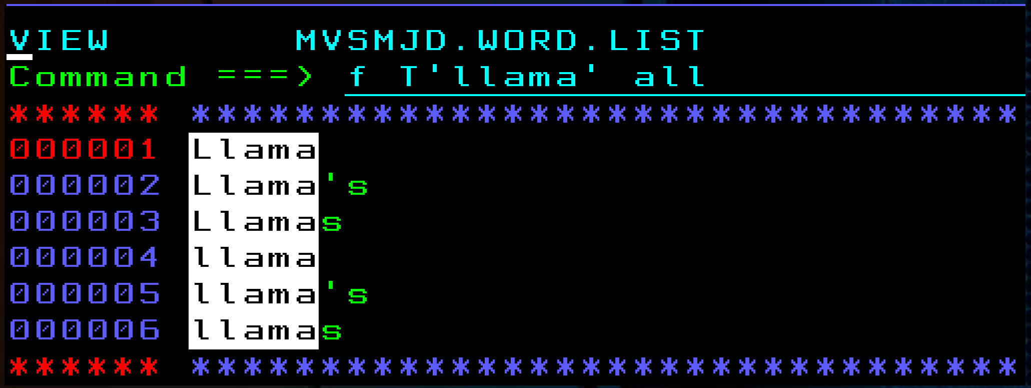


```
VIEW          MVSMJD.WORD.LIST
Command ===> f 'llama' all
****** *************************************
000001 Llama
000002 Llama's
000003 Llamas
000004 llama
000005 llama's
000006 llamas
****** *************************************
```

ISPF Examples

F T‘*llama*’

case insensitive

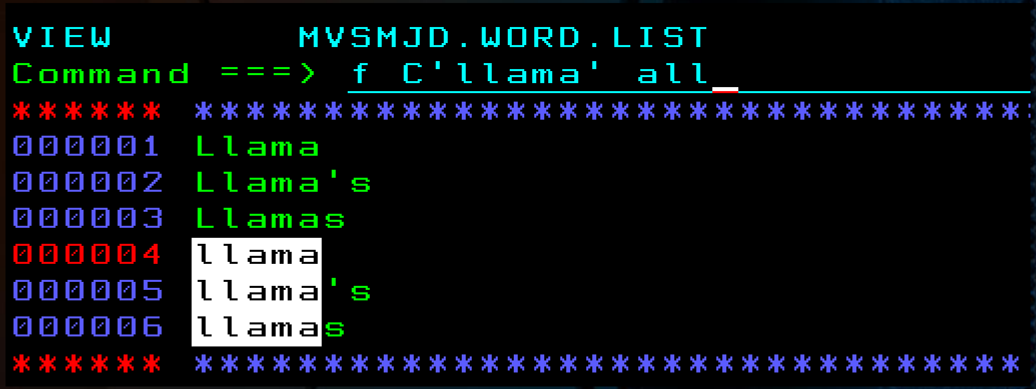


```
VIEW          MVSMJD.WORD.LIST
Command ===> f T'llama' all
****** ********************************
000001 Llama
000002 Llama's
000003 Llamas
000004 llama
000005 llama's
000006 llamas
****** ********************************
```

ISPF Examples

F C‘llama’

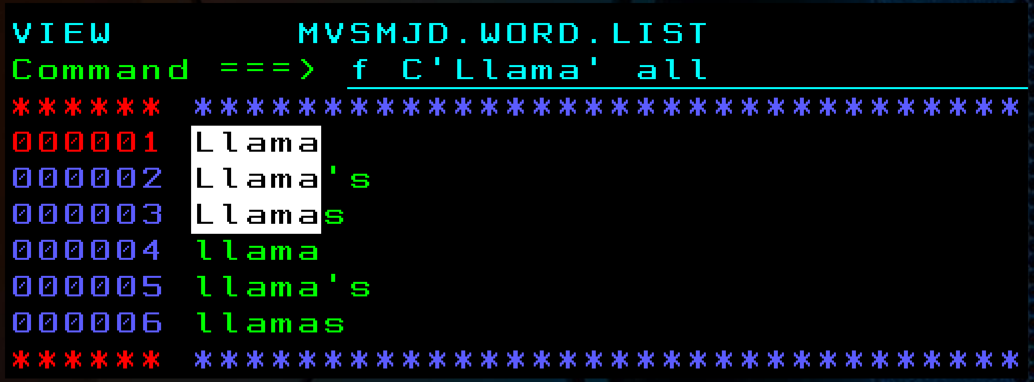
case sensitive



```
VIEW       MVSMJD.WORD.LIST
Command ===> f C'llama' all
****** ****************************************
000001 Llama
000002 Llama's
000003 Llamas
000004 llama
000005 llama's
000006 llamas
****** ****************************************
```

ISPF Examples

F C‘Llama’ case sensitive

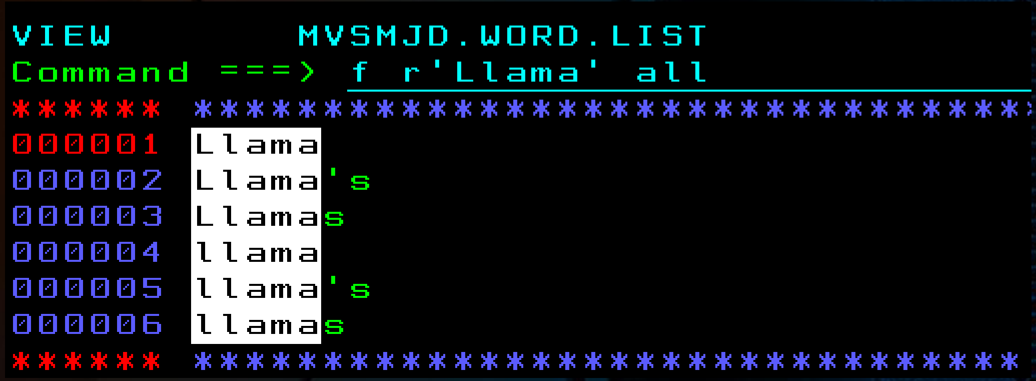


```
VIEW         MVSMJD.WORD.LIST
Command ===> f C'Llama' all
****** *********************************
000001 Llama
000002 Llama's
000003 Llamas
000004 llama
000005 llama's
000006 llamas
****** *********************************
```

ISPF Examples

F R‘Llama’

case insensitive



```
VIEW          MVSMJD.WORD.LIST
Command ===> f r'Llama' all
****** ********************************
000001 Llama
000002 Llama's
000003 Llamas
000004 llama
000005 llama's
000006 llamas
****** ********************************
```

ISPF Examples

```
VIEW       MVSMJD.WORD.LIST                          2 CHARS '\'s'
Command ===>  f r"\'s" all                            Scroll ===> CSR
****** ********************************* Top of Data **********************************
000001 Llama
000002 Llama's
000003 Llamas
000004 llama
000005 llama's
000006 llamas
****** ******************************** Bottom of Data ********************************
```

Find the characters `'s`

ISPF Examples

```
VIEW       MVSMJD.WORD.LIST                          No CHARS '\'s$' found
Command ===>  f r"\’s$" all                             Scroll ===> CSR
****** ********************************* Top of Data **********************************
000001 Llama
000002 Llama's
000003 Llamas
000004 llama
000005 llama's
000006 llamas
****** ******************************** Bottom of Data ********************************
```

Find the characters `'s` at the end of a line
(None found, because of the trailing blanks)

ISPF Examples

```
VIEW       MVSMJD.WORD.LIST                                2 CHARS '\'s[ ]*$’
Command ===>  f r"\’s[ ]*$" all                                  Scroll ===> CSR
****** ********************************* Top of Data **********************************
000001 Llama
000002 Llama's
000003 Llamas
000004 llama
000005 llama's
000006 llamas
****** ******************************** Bottom of Data ********************************
```

Find the characters `'s` at the end of a line with 0 or more trailing spaces

ISPF Examples

```
static int sqlerr (char *title) {
   char sqlemsg  [71];
   char sqlrp     [9];

    printf(" SQL ERROR HAS OCCURED : %s \n",title);
    printf("   SQLCODE  = %d\n",SQLCODE);
    printf("\n");
    strncpy(sqlrp,sqlca.sqlerrp,8);
    sqlrp[8] = '\0';
    strncpy(sqlemsg,sqlca.sqlerrmc,sqlca.sqlerrml);
    sqlemsg[sqlca.sqlerrml] = '\0';
    printf(" ----------------------------------\n");
    printf("SQLCA Code   : %d\n",SQLCODE);
    printf("SQLCA Errmc  : %s\n",sqlemsg);
    printf("SQLCA Errp   : %s\n"   ,sqlrp);
    printf("SQLCA Errd   : %02x %02x %02x %02x %02x %02x \n",
           sqlca.sqlerrd[0],sqlca.sqlerrd[1],sqlca.sqlerrd[2],
           sqlca.sqlerrd[3],sqlca.sqlerrd[4],sqlca.sqlerrd[5]);
    printf("SQLCA Warn   :%11.11s \n",sqlca.sqlwarn);
    printf("SQLCA State  : %5.5s \n" ,sqlca.sqlstate);

    return(SQLCODE);
}
```

Search requirement:

Variable definitions of type ‘char’
In a C program

ISPF Examples

```
static int sqlerr (char *title) {
   char sqlemsg  [71];
   char sqlrp     [9];

    printf(" SQL ERROR HAS OCCURED : %s \n",title);
    printf("   SQLCODE  = %d\n",SQLCODE);
    printf("\n");
    strncpy(sqlrp,sqlca.sqlerrp,8);
    sqlrp[8] = '\0';
    strncpy(sqlemsg,sqlca.sqlerrmc,sqlca.sqlerrml);
    sqlemsg[sqlca.sqlerrml] = '\0';
    printf(" ----------------------------------\n");
    printf("SQLCA Code   : %d\n",SQLCODE);
    printf("SQLCA Errmc  : %s\n",sqlemsg);
    printf("SQLCA Errp   : %s\n"   ,sqlrp);
    printf("SQLCA Errd   : %02x %02x %02x %02x %02x %02x \n",
           sqlca.sqlerrd[0],sqlca.sqlerrd[1],sqlca.sqlerrd[2],
           sqlca.sqlerrd[3],sqlca.sqlerrd[4],sqlca.sqlerrd[5]);
    printf("SQLCA Warn   :%11.11s \n",sqlca.sqlwarn);
    printf("SQLCA State  : %5.5s \n" ,sqlca.sqlstate);

    return(SQLCODE);
}
```

ISPF command

FIND ‘char’ ALL

First occurrence is not a variable definition

ISPF Examples

```
static int sqlerr (char *title) {
   char sqlemsg  [71];
   char sqlrp     [9];

    printf(" SQL ERROR HAS OCCURED : %s \n",title);
    printf("   SQLCODE  = %d\n",SQLCODE);
    printf("\n");
    strncpy(sqlrp,sqlca.sqlerrp,8);
    sqlrp[8] = '\0';
    strncpy(sqlemsg,sqlca.sqlerrmc,sqlca.sqlerrml);
    sqlemsg[sqlca.sqlerrml] = '\0';
    printf(" ----------------------------------\n");
    printf("SQLCA Code   : %d\n",SQLCODE);
    printf("SQLCA Errmc  : %s\n",sqlemsg);
    printf("SQLCA Errp   : %s\n"   ,sqlrp);
    printf("SQLCA Errd   : %02x %02x %02x %02x %02x %02x \n",
           sqlca.sqlerrd[0],sqlca.sqlerrd[1],sqlca.sqlerrd[2],
           sqlca.sqlerrd[3],sqlca.sqlerrd[4],sqlca.sqlerrd[5]);
    printf("SQLCA Warn   :%11.11s \n",sqlca.sqlwarn);
    printf("SQLCA State  : %5.5s \n" ,sqlca.sqlstate);

    return(SQLCODE);
}
```

ISPF command

FIND ‘[’ ALL

Matches all array references

ISPF Examples

```
static int sqlerr (char *title) {
   char sqlemsg  [71];
   char sqlrp     [9];

    printf(" SQL ERROR HAS OCCURED : %s \n",title);
    printf("   SQLCODE  = %d\n",SQLCODE);
    printf("\n");
    strncpy(sqlrp,sqlca.sqlerrp,8);
    sqlrp[8] = '\0';
    strncpy(sqlemsg,sqlca.sqlerrmc,sqlca.sqlerrml);
    sqlemsg[sqlca.sqlerrml] = '\0';
    printf(" ----------------------------------\n");
    printf("SQLCA Code   : %d\n",SQLCODE);
    printf("SQLCA Errmc  : %s\n",sqlemsg);
    printf("SQLCA Errp   : %s\n"   ,sqlrp);
    printf("SQLCA Errd   : %02x %02x %02x %02x %02x %02x \n",
           sqlca.sqlerrd[0],sqlca.sqlerrd[1],sqlca.sqlerrd[2],
           sqlca.sqlerrd[3],sqlca.sqlerrd[4],sqlca.sqlerrd[5]);
    printf("SQLCA Warn   :%11.11s \n",sqlca.sqlwarn);
    printf("SQLCA State  : %5.5s \n" ,sqlca.sqlstate);

    return(SQLCODE);
}
```

ISPF command

FIND R‘\[[0-9]+\]’ ALL

Better – matches array references in regex
(Find me any numbers enclosed in brackets)

ISPF Examples

```
static int sqlerr (char *title) {
   char sqlemsg  [71];
   char sqlrp     [9];

    printf(" SQL ERROR HAS OCCURED : %s \n",title);
    printf("   SQLCODE  = %d\n",SQLCODE);
    printf("\n");
    strncpy(sqlrp,sqlca.sqlerrp,8);
    sqlrp[8] = '\0';
    strncpy(sqlemsg,sqlca.sqlerrmc,sqlca.sqlerrml);
    sqlemsg[sqlca.sqlerrml] = '\0';
    printf(" ----------------------------------\n");
    printf("SQLCA Code   : %d\n",SQLCODE);
    printf("SQLCA Errmc  : %s\n",sqlemsg);
    printf("SQLCA Errp   : %s\n"   ,sqlrp);
    printf("SQLCA Errd   : %02x %02x %02x %02x %02x %02x \n",
           sqlca.sqlerrd[0],sqlca.sqlerrd[1],sqlca.sqlerrd[2],
           sqlca.sqlerrd[3],sqlca.sqlerrd[4],sqlca.sqlerrd[5]);
    printf("SQLCA Warn   :%11.11s \n",sqlca.sqlwarn);
    printf("SQLCA State  : %5.5s \n" ,sqlca.sqlstate);

    return(SQLCODE);
}
```

ISPF command

FIND R‘char[a-zA-Z]+\[[0-9]+\]’ ALL

Find the letters ‘char’

Followed by at least 1 letter or space

Followed by a ‘[‘

Followed by at least 1 number

Followed by a ‘]’

(Doesn’t include numbers or special characters in the variable names, but you get the gist)

Regex in Db2

and its implementation, idiosyncrasies, usage and examples

Db2 LUW

- PureXML added regex support in v9.7 via Xquery with the *matches* function

```
db2 "with val as (
       select t.text
       from texts t
       where xmlcast(
                    xmlquery(
                           `fn:matches(\$TEXT,`'^[A-Za-z 0-9]*$'’)’
                    ) as integer
             ) = 1
       )
       select * from val"
```

Db2 LUW

- Db2 11.1 added built-in regex support with the following functions:
  - Pattern matching (returns a boolean result)
    - REGEXP_LIKE

Db2 LUW

- Db2 11.1 added built-in regex support with the following functions:
  - Pattern matching (returns a boolean result)
    - REGEXP_LIKE
  - Counting/locating a pattern (returns a numeric result)
    - REGEXP_COUNT
    - REGEXP_INSTR
    - REGEXP_MATCH_COUNT (a synonym of REGEXP_COUNT)

Db2 LUW

- Db2 11.1 added built-in regex support with the following functions:
  - Pattern matching (returns a boolean result)
    - REGEXP_LIKE
  - Counting/locating a pattern (returns a numeric result)
    - REGEXP_COUNT
    - REGEXP_INSTR
    - REGEXP_MATCH_COUNT (a synonym of REGEXP_COUNT)
  - Extracting/transforming a pattern (returns a string result)
    - REGEXP_EXTRACT (a synonym of REGEXP_SUBSTR)
    - REGEXP_REPLACE
    - REGEXP_SUBSTR

Db2 LUW

Input:

```
WITH cust(street) AS (
-- Six unwanted PO BOX address rows, each formatted slightly differently
    VALUES ('PO BOX 1'), ('PO  BOX 2'), ('P.O. BOX 3’)
           , ('P O BOX  4'), ('P. O.BOX 5'), ('po box 6’)
-- and one legitimate address row that belongs in the result set
           ,('POBOXTON CT ROAD NO. 3’)
)
SELECT street FROM cust
WHERE NOT REGEXP_LIKE( street, '^\s*P\.?\s*O\.?\s*BOX\b', 'i' );
```

Output:

```
STREET
----------------------
POBOXTON CT ROAD NO. 3

  1 record(s) selected.
```

Db2 z/OS

- XML added regex support in Db2 10 via Xquery with the *matches* function

Db2 z/OS

- XML added regex support in Db2 10 via Xquery with the *matches* function
- Still no built-in support (yet)

Db2 z/OS

- XML added regex support in Db2 10 via Xquery with the *matches* function
- Still no built-in support (yet)
- LUW REGEX functions available in IDAA as a pass thru

Db2 z/OS

- XML added regex support in Db2 10 via Xquery with the *matches* function
- Still no built-in support (yet)
- LUW REGEX functions available in IDAA as a pass thru
- CPU-intensive

Db2 z/OS

- XML added regex support in Db2 10 via Xquery with the *matches* function
- Still no built-in support (yet)
- LUW REGEX functions available in IDAA as a pass thru
- CPU-intensive
- “Stage 3” predicates (non-indexable)
    - Sequential scan
    - Make sure you’ve reduced the result set as much as you can BEFORE invoking regex!

Db2 z/OS

```
SELECT NAME, DBID FROM SYSIBM.SYSDATABASE;
   +--------------------------------------+
   |           NAME           |   DBID    |
   +--------------------------------------+
 1_| DSN00001                 |       266 |
 2_| DSN00002                 |       267 |
 3_| DSNOPTDB                 |       276 |
 4_| DSNATPDB                 |       256 |
 5_| DSN5JSDB                 |       277 |
 6_| DSNADMDB                 |       259 |
 7_| DSNDB01                  |         1 |
 8_| DSNDB04                  |         4 |
 9_| DSNDB06                  |         6 |
10_| DSNMQDB                  |       262 |
(Result truncated)
```

Match databases that begin with DSN, are followed by zero or more characters, but must end with 0-9.

Db2 z/OS

```
SELECT NAME, DBID
FROM SYSIBM.SYSDATABASE
WHERE XMLEXISTS(
             '$newDoc[fn:matches(., "^(DSN).*[0-9]$")]’
             PASSING XMLQUERY(
                    '<doc>{$NameCol}</doc>’
                    PASSING NAME as "NameCol‟
                    )
             as "newDoc‟
       );
```

Match databases that begin with DSN, are followed by zero or more characters, but must end with 0-9.

Db2 z/OS

```
SELECT NAME, DBID
FROM SYSIBM.SYSDATABASE
WHERE XMLEXISTS(
              '$newDoc[fn:matches(., "^(DSN).*[0-9]$")]’
              PASSING XMLQUERY(
                     '<doc>{$NameCol}</doc>’
                     PASSING NAME as "NameCol‟
                     )
             as "newDoc‟
       );
```

The XML function matches only takes XML as input.

Db2 z/OS

```
SELECT NAME, DBID
FROM SYSIBM.SYSDATABASE
WHERE XMLEXISTS(
             '$newDoc[fn:matches(., "^(DSN).*[0-9]$")]’
             PASSING XMLQUERY(
                    '<doc>{$NameCol}</doc>’
                    PASSING NAME as "NameCol“
                    )
            as "newDoc“
       );
```

The XML function matches only takes XML as input. So, we first create an XML document using the NAME column content from SYSIBM.SYSDATABASE using XQuery Constructor.

Db2 z/OS

```
SELECT NAME, DBID
FROM SYSIBM.SYSDATABASE
WHERE XMLEXISTS(
              '$newDoc[fn:matches(., "^(DSN).*[0-9]$")]’
              PASSING XMLQUERY(
                    '<doc>{$NameCol}</doc>’
                    PASSING NAME as "NameCol"
                    )
             as "newDoc"
       );
```

The XML function matches only takes XML as input. So, we first create an XML document using the NAME column content from SYSIBM.SYSDATABASE using XQuery Constructor. Then, we pass the constructed XML (“newDoc”) as input to fn:matches function.

Db2 z/OS

```
SELECT NAME, DBID
FROM SYSIBM.SYSDATABASE
WHERE XMLEXISTS(
             '$newDoc[fn:matches(., "^(DSN).*[0-9]$")]’
             PASSING XMLQUERY(
                    '<doc>{$NameCol}</doc>’
                    PASSING NAME as "NameCol‶
                    )
             as "newDoc‶
       );
```

The XML function matches only takes XML as input. So, we first create an XML document using the NAME column content from SYSIBM.SYSDATABASE using XQuery Constructor. Then, we pass the constructed XML (“newDoc”) as input to fn:matches function. The second parameter of fn:matches is pattern we search for. It supports regular expression.

Db2 z/OS

```
    +-----------------------------------+
    |          NAME          |   DBID   |
    +-----------------------------------+
  1_| DSN00001               |      266 |
  2_| DSN00002               |      267 |
  3_| DSN00005               |      278 |
  4_| DSN00006               |      279 |
  5_| DSN00007               |      280 |
  6_| DSN00008               |      281 |
  7_| DSN00009               |      282 |
  8_| DSNDB01                |        1 |
  9_| DSN00011               |      284 |
 10_| DSN00010               |      283 |
 (Result truncated)
```

Match databases that begin with DSN, are followed by zero or more characters, but must end with 0-9.

Db2 z/OS

- You can also use the same format as the PureXML query from LUW, except you must use PASSING as follows:

```
with val as (
       select t.text
       from texts t
       where xmlcast(
                    xmlquery(`fn:matches($v,`'^[A-Za-z 0-9]*$’’)’
                           PASSING t.text as “v”
                    ) as integer
              ) = 1
       )
select * from val;
```

Regex in Rexx

and its embarrassing lack of existence

IBM-Supplied Regex support in Rexx

IBM-Supplied Regex support in Rexx

- Click to add text

IBM-Supplied Regex support in Rexx

- TSMYOYO

User-supplied Regex support in Rexx

- https://github.com/IBM/zos-tools-and-toys

- Martin Packer (IBM UK) – developerWorks

- BPXWUNIX (USS function in Rexx) – KnowledgeCentre

Additional Information

…for further education…

Performance

- Like SQL, how you code a regex has an impact upon its performance

Performance

- **Like SQL, how you code a regex has an impact upon its performance**

  E.g. to match the words tonight, tonite or toknight, in the phrase "hot tonic tonight", you could specify the search expressions:

Performance

- **Like SQL, how you code a regex has an impact upon its performance**

  E.g. to match the words tonight, tonite or toknight, in the phrase "hot tonic tonight", you could specify the search expressions:

  tonite|tonight|toknight complete words

Performance

- **Like SQL, how you code a regex has an impact upon its performance**

  E.g. to match the words tonight, tonite or toknight, in the phrase “hot tonic tonight”, you could specify the search expressions:

tonite|tonight|toknight	complete words
to(nite|knight|night)	1st removal of commonality (to)

Performance

- **Like SQL, how you code a regex has an impact upon its performance**

E.g. to match the words tonight, tonite or toknight, in the phrase “hot tonic tonight”, you could specify the search expressions:

tonite|tonight|toknight	complete words
to(nite|knight|night)	1st removal of commonality (to)
to(ni(ght|te)|knight)	2nd removal of commonality (ni in night and nite)

Performance

- **Like SQL, how you code a regex has an impact upon its performance**

E.g. to match the words tonight, tonite or toknight, in the phrase “hot tonic tonight”, you could specify the search expressions:

tonite|tonight|toknight	complete words
to(nite|knight|night)	1st removal of commonality (to)
to(ni(ght|te)|knight)	2nd removal of commonality (ni in night and nite)
to(k?night|nite)	optional k

Performance

- **Like SQL, how you code a regex has an impact upon its performance**

E.g. to match the words tonight, tonite or toknight, in the phrase “hot tonic tonight”, you could specify the search expressions:

<table>
<tr><th></th><th></th></tr>
<tr><td>tonite|tonight|toknight</td><td>complete words</td></tr>
<tr><td>to(nite|knight|night)</td><td>1st removal of commonality (to)</td></tr>
<tr><td>to(ni(ght|te)|knight)</td><td>2nd removal of commonality (ni in night and nite)</td></tr>
<tr><td>to(k?night|nite)</td><td>optional k</td></tr>
<tr><td>to(k?ni(ght|te))</td><td>3rd removal of commonality (ni)</td></tr>
</table>

Performance

- **Like SQL, how you code a regex has an impact upon its performance**

E.g. to match the words tonight, tonite or toknight, in the phrase “hot tonic tonight”, you could specify the search expressions:

tonite|tonight|toknight	complete words
to(nite|knight|night)	1st removal of commonality (to)
to(ni(ght|te)|knight)	2nd removal of commonality (ni in night and nite)
to(k?night|nite)	optional k
to(k?ni(ght|te))	3rd removal of commonality (ni)
tok?ni(gh)?te?	further simplification

to required, **k** optional, **ni** required, **gh** optional, **t** required, **e** optional

Performance

- **Like SQL, how you code a regex has an impact upon its performance**

  E.g. to match the words tonight, tonite or toknight, in the phrase “hot tonic tonight”, you could specify the search expressions:

tonite|tonight|toknight	complete words
to(nite|knight|night)	1st removal of commonality (to)
to(ni(ght|te)|knight)	2nd removal of commonality (ni in night and nite)
to(k?night|nite)	optional k
to(k?ni(ght|te))	3rd removal of commonality (ni)
tok?ni(gh)?te?	further simplification

to required, **k** optional, **ni** required, **gh** optional, **t** required, **e** optional

- **Greediness, Laziness and Backtracking**
  - (See References for further info)

Engine Types

- **DFA**
  - Deterministic Finite Automaton
  - Text-Directed
  - Find the longest possible match
  - Very fast
  - Consistent
- **NFA**
  - Nondeterministic Finite Automaton
  - Regex-Directed
  - Greedy

Caveat faber!

(Let the architect beware!)

- **CODE PAGES**
  - Always attune your 3270 emulator to whatever code page TSO is using.
  - Trial and error

Caveat faber!

(Let the architect beware!)

- **CODE PAGES**
  - Always attune your 3270 emulator to whatever code page TSO is using.
  - Trial and error
- **FLAVOURS**
  - Not all programming languages or implementations of regex contain the full set of metacharacters
  - Trial and error

Caveat faber!

(Let the architect beware!)

- **CODE PAGES**
  - Always attune your 3270 emulator to whatever code page TSO is using.
  - Trial and error
- **FLAVOURS**
  - Not all programming languages or implementations of regex contain the full set of metacharacters
  - Trial and error
- **KNOW YOUR DATA**
  - You must know what it is you are looking for before you can find it
  - Trial and error

Caveat faber!

(Let the architect beware!)

- **CPU**
  - No such thing as a free lunch
  - Not CPU-light
  - Db2 access path = sequential scans

Caveat faber!

(Let the architect beware!)

- **CPU**
  - No such thing as a free lunch
  - Not CPU-light
  - Db2 access path = sequential scans
- **Handy for returning breaches of naming standards**

Caveat faber!

(Let the architect beware!)

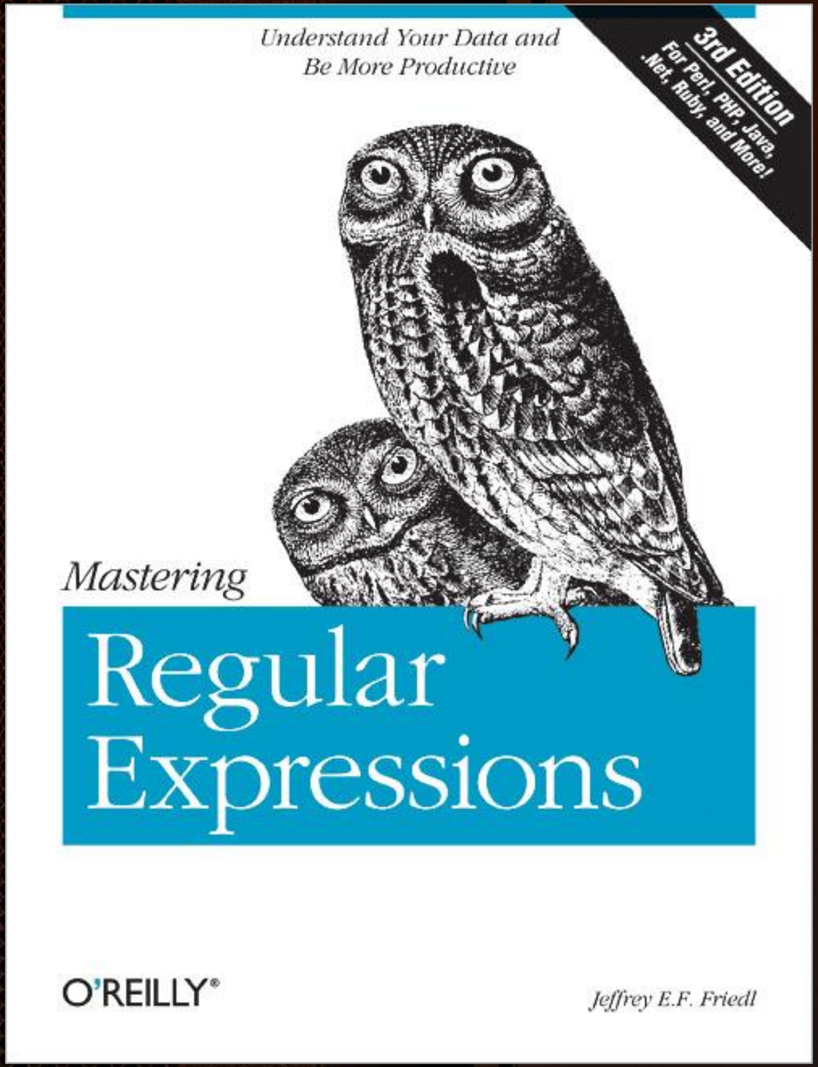
- **CPU**
  - No such thing as a free lunch
  - Not CPU-light
  - Db2 access path = sequential scans
- **Handy for returning breaches of naming standards**
- **Last resort**
  - Try LIKE / REPLACE / TRANSLATE first

Caveat faber!

(Let the architect beware!)

- **CPU**
  - No such thing as a free lunch
  - Not CPU-light
  - Db2 access path = sequential scans
- **Handy for returning breaches of naming standards**
- **Last resort**
  - Try LIKE / REPLACE / TRANSLATE first
- **^(?=(?!(.)1)([^DO:105-93+30])(?-1)(?<!d(?<=(?![5-90-3])d))).[^WHY?]$**
  - Your colleagues will hate you

References



Understand Your Data and Be More Productive

3rd Edition

For Perl, PHP, Java, .Net, Ruby, and More!

Mastering

Regular Expressions

O'REILLY®

Jeffrey E.F. Friedl

References

https://www.idug.org/p/bl/et/blogaid=605

https://www.idug.org/p/bl/et/blogaid=670

https://stackoverflow.com/questions/4763757/regular-expressions-in-db2-sql

https://regexone.com/

https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.f54em00/useofr1.htm#useofr1

https://www.ibm.com/support/knowledgecenter/SSEPEK_12.0.0/xml/src/tpc/db2z_regularexpression.html

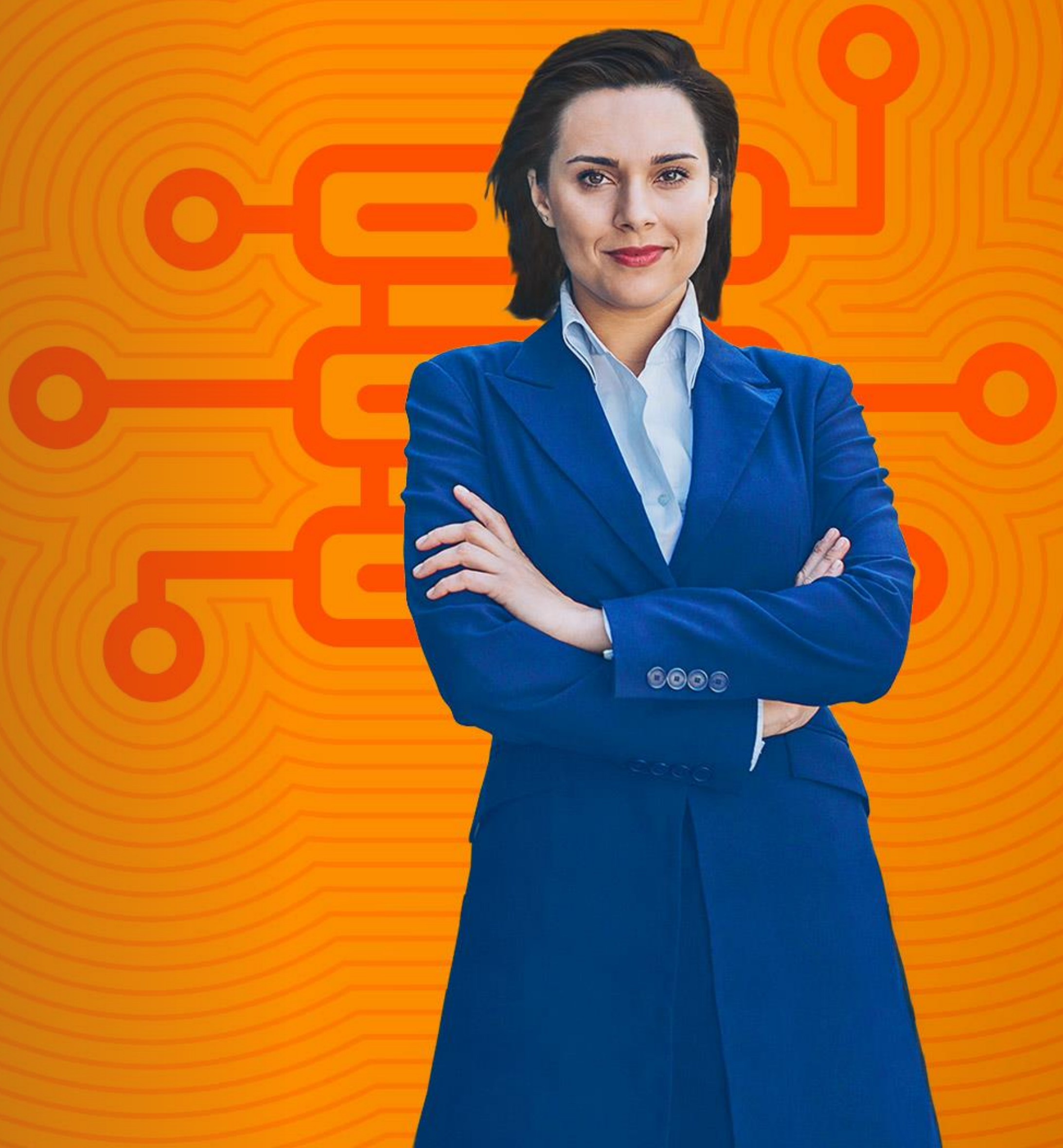
https://www.seg.de/en/2019-01-regular-expressions-in-db2-sql/

https://www.ibm.com/developerworks/community/blogs/MartinPacker/?lang=en

http://www.rexegg.com/

bmc

Thank You



bmc Software - GSE UK Conference 2019

Dock into the Dark Side!

Tuesday 5th November

Start Time	End Time	Stream	Room	Title	Speaker
16:45	17:45	zCMPA	Woodcote	Hiperdispatch – SLA improvements & MSU reductions	Donald Zeunert
16:45	17:45	Db2	Nurburgring	MLC – I'm paying HOW MUCH for Db2?	Phil Grainger

Wednesday 6th November

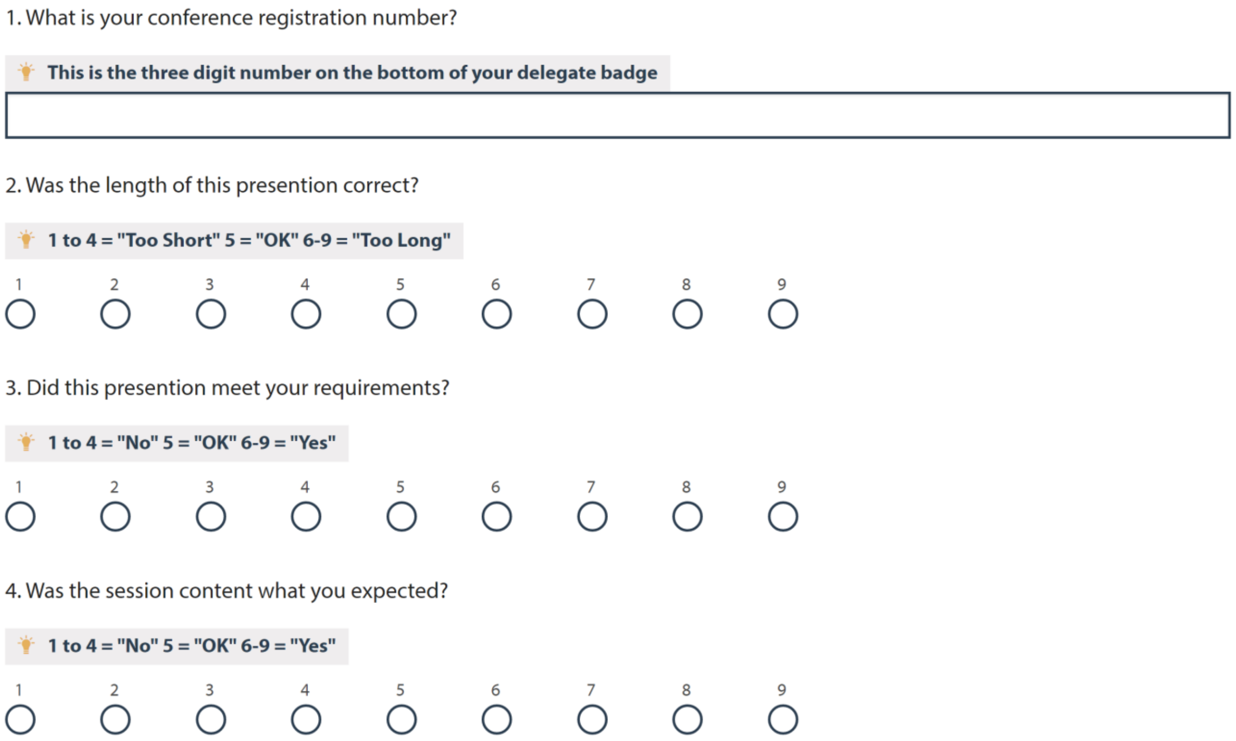
Start Time	End Time	Stream	Room	Title	Speaker
11:45	12:45	IMS	Wellington B	Modernizing IMS Change Management	David Schipper
13:45	14:45	IMS	Wellington B	IMS10: Using Real-Time IMS Data for Security Analysis	Nick Griffin
16:30	19:30	IMS	Wellington B	Innovative Customer Solutions to IMS Challenges	David Schipper

Thursday 7th November

Start Time	End Time	Stream	Room	Title	Speaker
09:00	10:00	Db2	Nurburgring	Putting the capital A in 'Agile on the mainframe'	Tony Poole
11:45	12:45	Db2	Nurburgring	Express Yourself	Marcus Davage

Please submit your session feedback!

- Do it online at http://conferences.gse.org.uk/2019/feedback/IO
- This session is IO



1. What is your conference registration number?

This is the three digit number on the bottom of your delegate badge

2. Was the length of this presention correct?

1 to 4 = "Too Short" 5 = "OK" 6-9 = "Too Long"

1	2	3	4	5	6	7	8	9
○	○	○	○	○	○	○	○	○

3. Did this presention meet your requirements?

1 to 4 = "No" 5 = "OK" 6-9 = "Yes"

1	2	3	4	5	6	7	8	9
○	○	○	○	○	○	○	○	○

4. Was the session content what you expected?

1 to 4 = "No" 5 = "OK" 6-9 = "Yes"

1	2	3	4	5	6	7	8	9
○	○	○	○	○	○	○	○	○