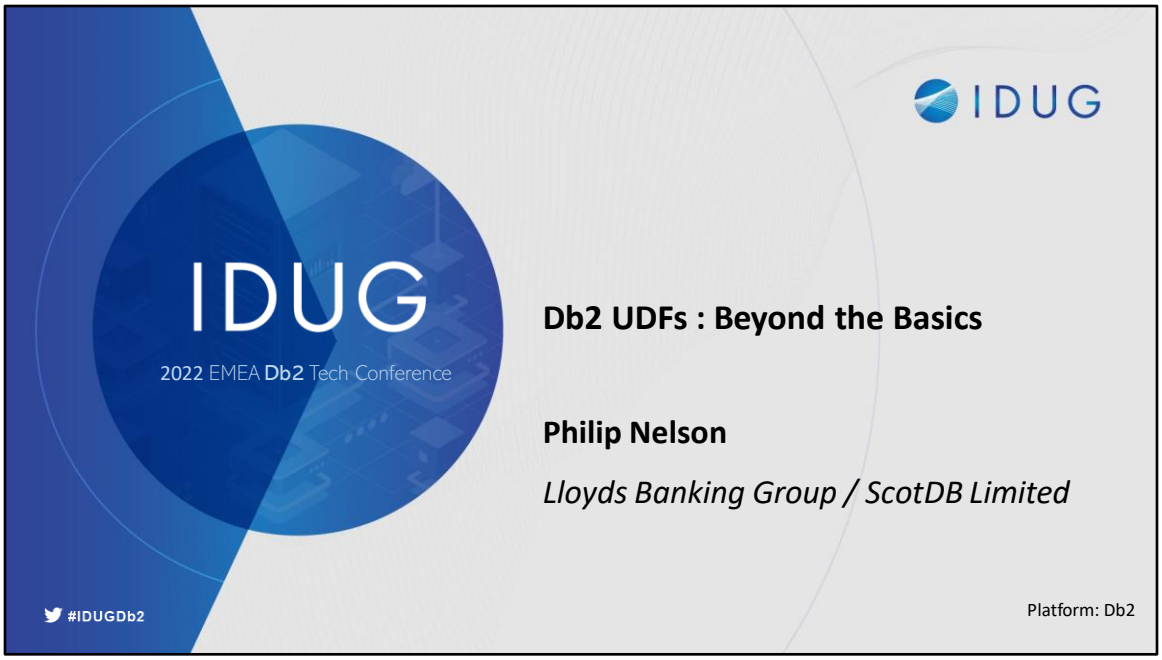
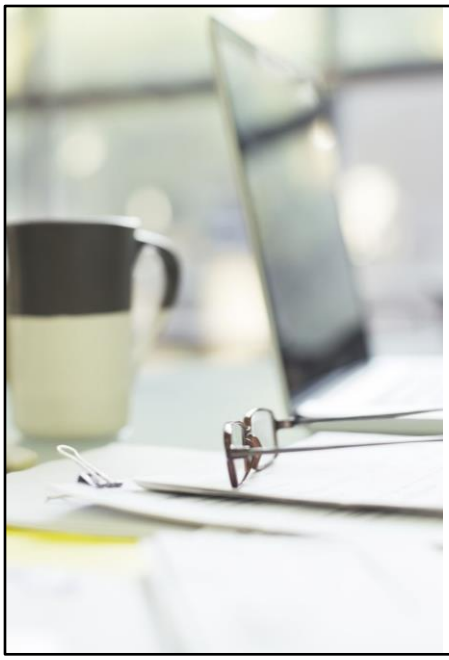


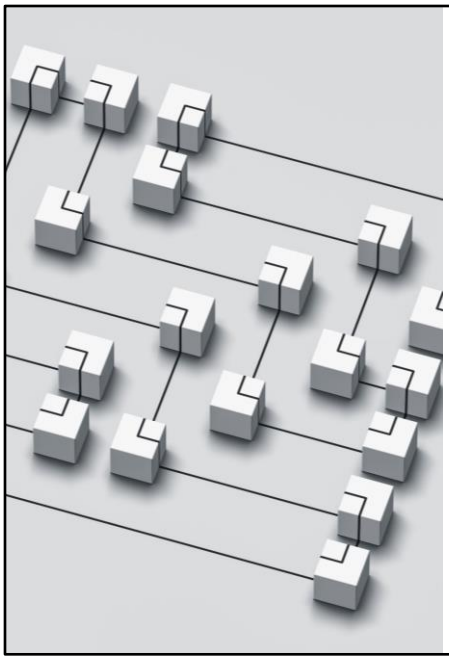
Db2 User Defined Functions (UDFs) are a largely ignored tool in the developer's armoury, providing exciting and in some cases unique capabilities. This presentation will go beyond creating simple UDFs to look at more complex use cases and capabilities. A particular focus will be on table UDFs and why they sometimes are better than stored procedures.

Philip Nelson has used DB2 (for z/OS) since 1989 and DB2 (for LUW) since the beginning. He has been an IDUG volunteer since 1998; and has served as a conference presenter; IDUG Solutions Journal contributor and editor, Content Committee leader and most recently on the EMEA Conference Planning Committee. His interests include database design and performance tuning but especially exploitation of DB2 with new and emerging technologies. He has been an early adopter of pureXML; temporal support; cloud deployments and JSON support.



Agenda

- **Explore types of UDF and UDF syntax**
- Compare table UDFs to alternative solutions such as stored procedures and views
- Discuss some exciting UDF-only features, in particular the use of PIPE within table UDFs



UDF Types

- External : scalar, table and OLE DB table
- Inline UDFs
- SQL : scalar, row and table
- Sourced UDFs, function mappings and function templates
- Aggregate functions

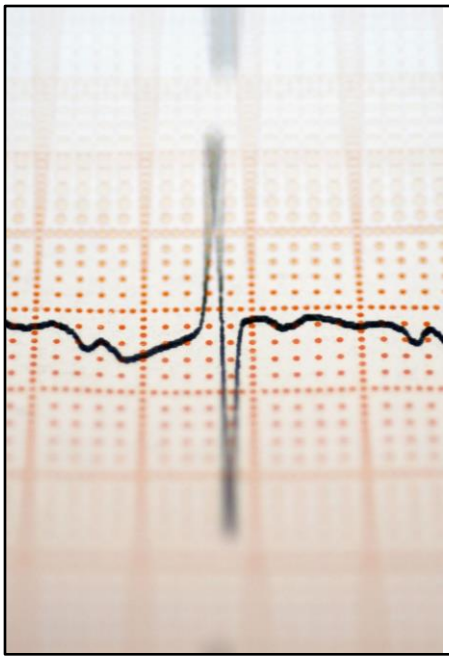
External UDFs

Define a UDF using code written in a third party language

Must specify -

- Input and output parameter details
- EXTERNAL keyword on function definition
- NAME keyword provides details of external code to execute
- LANGUAGE keyword specifies external code language
  - C / JAVA / CLR / OLE / CPP / PYTHON
- PARAMETER STYLE is dependent on source language
  - DB2GENERAL / JAVA : for language JAVA only
  - SQL : for languages C / CLR and OLE
  - NPSGENERIC : for languages CPP and PYTHON

External UDFs are not the topic of this presentation ...



Inline UDFs

- The simplest UDF syntax

```
CREATE OR REPLACE FUNCTION <function-name>
(IN|OUT|INOUT <param-name> <data-type>, …)
RETURNS <data-type>
<option-list>
<SQL-function-body>;
```

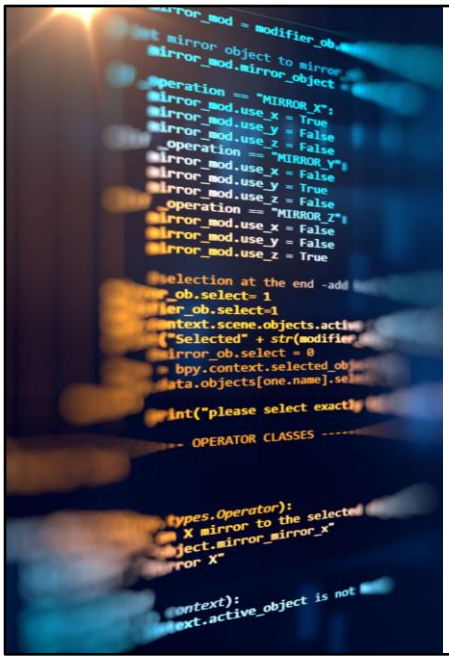
- In the case of inline UDFs the function body is a single SQL expression

Inline UDF : Example

- Simple function to return an input date field as a CHAR(8) expression in YYYYMMDD format

```
CREATE FUNCTION UDF_DATE2CHAR_INLINE
(paramDate DATE)
RETURNS CHAR(8)
LANGUAGE SQL CONTAINS SQL
NO EXTERNAL ACTION DETERMINISTIC
RETURN
CAST(TO_CHAR(paramDate,'YYYYMMDD') as CHAR(8));
```

- Note the single SQL expression
- Parameter type IN is the default and does not need to be specified



SQL UDFs

- Define a UDF using code written in Procedural Language
- Can return a scalar value, a row or a table
- For full SQL/PL support function body inside BEGIN / END clause

Simple Scalar UDF : Example

```
CREATE FUNCTION UDF_DATE2CHAR
(paramDate DATE)
RETURNS CHAR(8)
LANGUAGE SQL
CONTAINS SQL
NO EXTERNAL ACTION
DETERMINISTIC
BEGIN
  RETURN CAST(TO_CHAR(paramDate,'YYYYMMDD') as CHAR(8));
END
```

Comparison : Inline vs Simple Scalar UDF Catalog Details

ROUTINENAME	IMPLEMENTATION	LIB_ID
UDF_DATE2CHAR	db2pvm!pvm_entry	847245216
UDF_DATE2CHAR_INLINE	-	-1

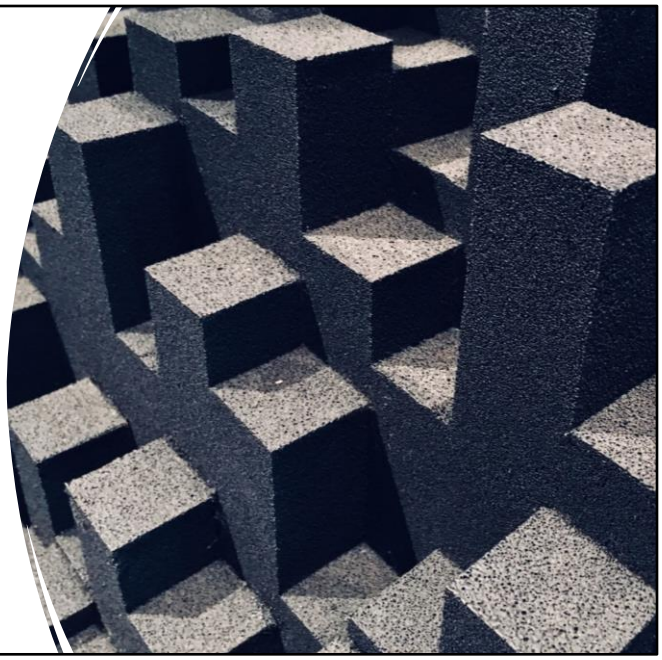
- Selected fields from catalog
  (SYSCAT.ROUTINES)
- INLINE does not have LIB_ID
  - No package exists
  - Function body parsed with calling SQL

Comparison : Inline vs Simple Scalar UDF Performance Considerations

- Tested by Invoking Each UDF approx 200k times
- CPU and elapsed times were similar
  (in Db2 for z/OS showed significant differences)
- Conclusion -
  - Use full scalar UDFs without performance concerns
- Inline UDFs -
  - Limited functionality
  - More difficult to code
  - Cannot have comprehensive error handling

Invoking Scalar UDFs : Example

```
SELECT
UDF_DATE2CHAR(MY_DATE)
FROM MYTABLE
;
```



Row UDFs

- Returns a single row
- Only for use as a transformation on a structured data type

```
CREATE FUNCTION FROM_AIRCRAFT
(A AIRCRAFT)
RETURNS ROW
(AIRCRAFT_TYPE VARCHAR(50), CONTRUCTORS_NUMBER VARCHAR(50))
LANGUAGE SQL CONTAINS SQL NO EXTERNAL ACTION DETERMINISTIC
RETURN VALUES
(A..AIRCRAFT_TYPE, A..CN);
```

Table UDFs

- Return a table of data
- Invoked using -

```
SELECT x.col1, x.col2
FROM TABLE
(my_table_udf (var1)) AS x;
```

- See later for much more on table UDFs

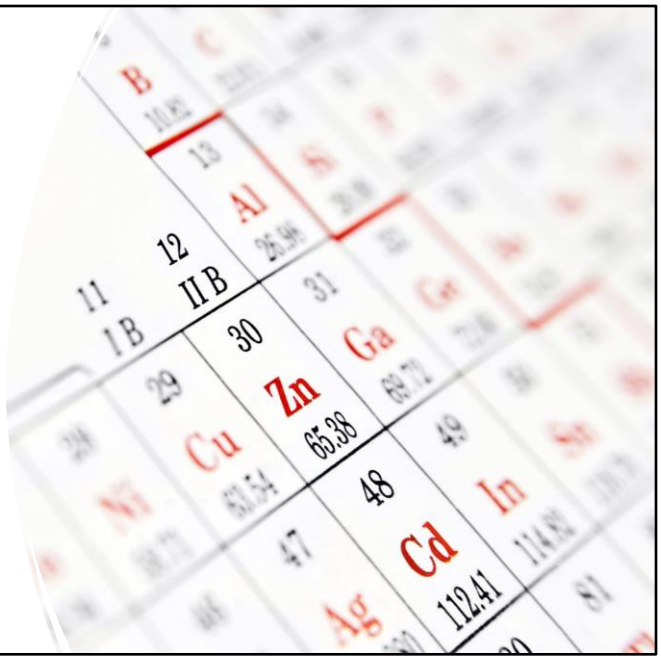


Table UDF Creation : Example

```
CREATE FUNCTION UDF_TABLE_DATA
(paramDate DATE)
RETURNS TABLE
(col1 INTEGER, col2 CHAR(8), col3 TIMESTAMP
LANGUAGE SQL CONTAINS SQL NO EXTERNAL ACTION DETERMINISTIC
BEGIN
  RETURN
  SELECT col1, col2, col3 FROM table
  WHERE coldate = paramDate;
END
```

Sourced UDFs

Define one UDF based on another UDF

- Typically used to provide the same functionality for different input types
- Casting between the input types is done automatically

Typical use case

- Make standard functions available for user defined types

Sourced UDFs : Example

```
CREATE DISTINCT TYPE MONEY AS DECIMAL(9,2) WITH COMPARISONS;

CREATE TABLE ORDER (ORDER_ID BIGINT, SALE_AMOUNT MONEY);

INSERT INTO ORDER VALUES (1,100.25),(2,99.75);

SELECT SUM(SALE_AMOUNT) FROM ORDER;
SQL0440N No authorized routine named "SUM" of type "FUNCTION"
having compatible arguments was found.  SQLSTATE=42884

CREATE FUNCTION SUM (MONEY)
       RETURNS DECFLOAT
       SOURCE  SYSIBM.AVG(DECIMAL(9,2));

SELECT SUM(SALE_AMOUNT) FROM ORDER;  <== now works
```

Function Templates and Mappings

- Provide access to functions on federated systems
- Mapping can be to a single federated system or all federated systems of a particular system type
- Can be used to disable access to native federated functions
  - When local function and federated function has the same name
  - By default optimizer uses the one it deems less expensive
  - Use "DISABLE" option on function mapping to only use local function

Function Template and Mapping Single Federated Server

```
CREATE FUNCTION UDF_CHAR2DATE (CHAR(8))
RETURNS DATE
AS TEMPLATE DETERMINISTIC NO EXTERNAL ACTION;

CREATE FUNCTION MAPPING MY_ZOS_CHAR2DATE
FOR UDF_CHAR2DATE(CHAR(8))
SERVER ZOS1
OPTIONS
(REMOTE_NAME 'CHAR2DATE');
```

Function Template and Mapping
All Federated Servers of a Particular Type

```
CREATE FUNCTION UDF_CHAR2DATE (CHAR(8))
RETURNS DATE
AS TEMPLATE DETERMINISTIC NO EXTERNAL ACTION;

CREATE FUNCTION MAPPING MY_ZOS_CHAR2DATE
FOR UDF_CHAR2DATE(CHAR(8))
SERVER TYPE DB2/ZOS
OPTIONS
(REMOTE_NAME 'CHAR2DATE');
```

Aggregate Functions

- Method of building user-defined aggregation in standard pieces
- Keyword AGGREGATE identifies this as an aggregate function
- State variables used throughout specified in WITH clause
- Three stored procedure components must be provided -
  - INITIALIZE : only OUT parameters (matching state variables)
  - ACCUMULATE : IN parm for input data, INOUT parms matching state variables
  - MERGE : IN and INOUT parms, both matching state variables
- One UDF component must be provided -
  - FINALIZE : parms matching state variables, return is final result

Aggregate Function Example Components : INITIALIZE and ACCUMULATE

```
CREATE OR REPLACE PROCEDURE mymean_initialize
(OUT sum DECFLOAT, OUT count INT)
LANGUAGE SQL CONTAINS SQL
BEGIN
  SET sum = 0;
  SET count = 0;
END #
CREATE OR REPLACE PROCEDURE mymean_accumulate
(IN input DECFLOAT, INOUT sum DECFLOAT, INOUT count INT)
LANGUAGE SQL CONTAINS SQL
BEGIN
  SET sum = sum + input;
  SET count = count + 1;
END #
```

Aggregate Function Example Components : MERGE and FINALIZE

```
CREATE OR REPLACE PROCEDURE mymean_merge
(IN sum DECFLOAT, IN count INT,INOUT mergesum DECFLOAT, INOUT mergecount INT)
LANGUAGE SQL CONTAINS SQL
BEGIN
  SET mergesum = sum + mergesum;
  SET mergecount = count + mergecount;
END #
CREATE OR REPLACE FUNCTION mymean_finalize
(sum DECFLOAT, count INT)
LANGUAGE SQL CONTAINS SQL
RETURNS DECFLOAT(34)
BEGIN
  RETURN (sum /  count);
END #
```

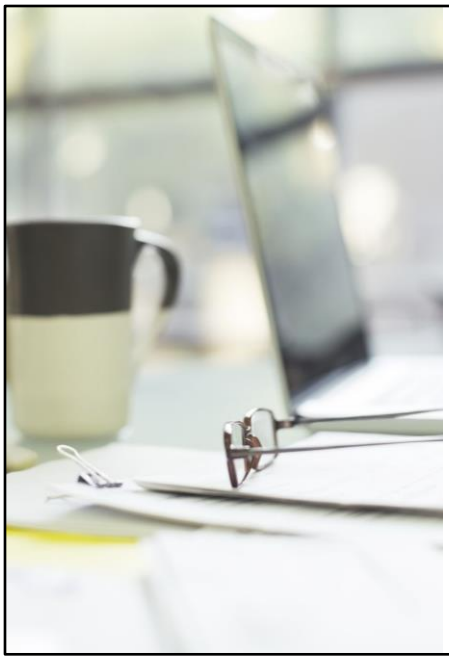
Aggregate Function Example Function Body

```
CREATE OR REPLACE FUNCTION mymean(DECFLOAT)
RETURNS DECFLOAT(34)
AGGREGATE WITH (sum DECFLOAT, count INT)
USING
INITIALIZE PROCEDURE mymean_initialize
ACCUMULATE PROCEDURE mymean_accumulate
MERGE PROCEDURE mymean_merge
FINALIZE FUNCTION mymean_finalize
#
```

Aggregate Function Example: Output

```
db2 "select avg(salary) as SYS_AVG from employee"
SYS_AVG
------------------------------
58155.357142857142857142857142

db2 "select db2inst1.mymean(salary) as MYMEAN from employee"
MYMEAN
-----------------------------------
58155.35714285714285714285714285714
```



Agenda

- Explore types of UDF and UDF syntax
- **Compare table UDFs to alternative solutions such as stored procedures and views**
- Discuss some exciting UDF-only features, in particular the use of PIPE within table UDFs

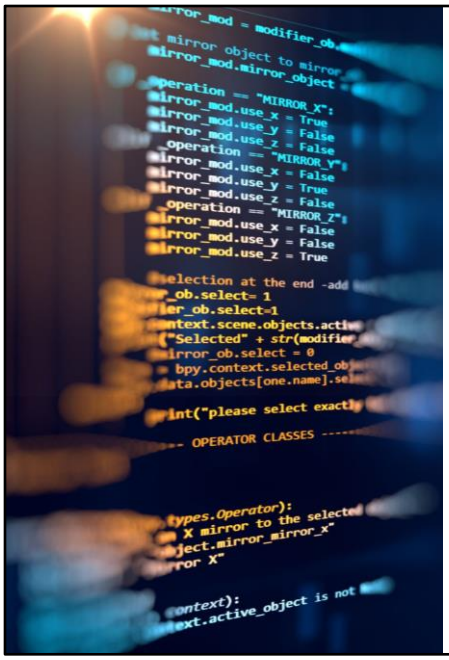


Table UDFs vs Views : Comparison

- Both are accessible from regular SELECT statements
- Views are limited to single SQL expression
  - Complex logic becomes challenging to add
- Views cannot have user-defined error handling
  - Db2 controls what errors (if any) are returned

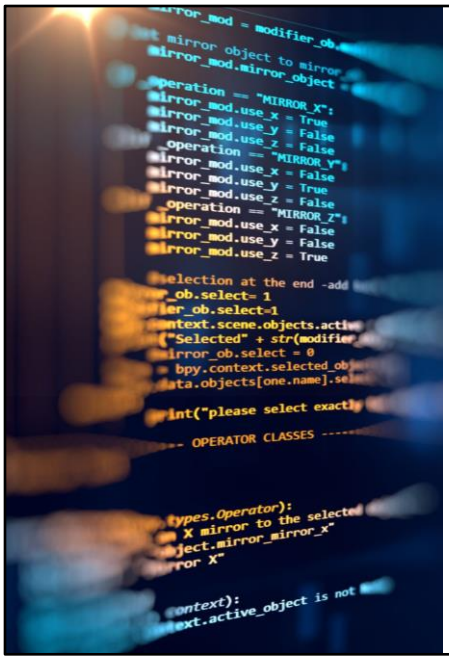


Table UDFs vs Stored Procedures Similarities

- Can return a (single) result set
- Can have multiple SQL statements
  - Complex logic can be implemented
- Can return user defined error codes

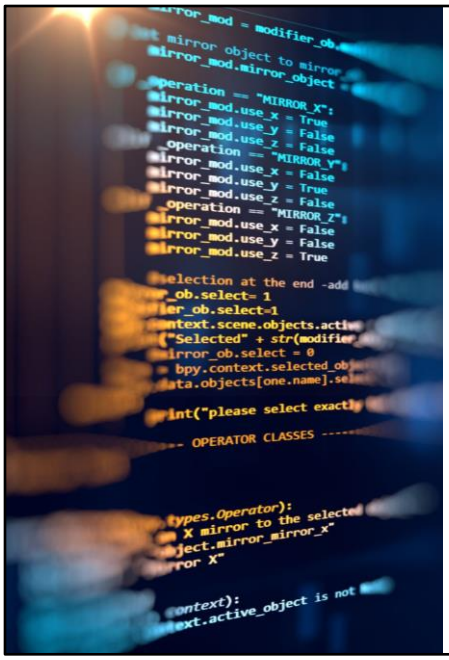
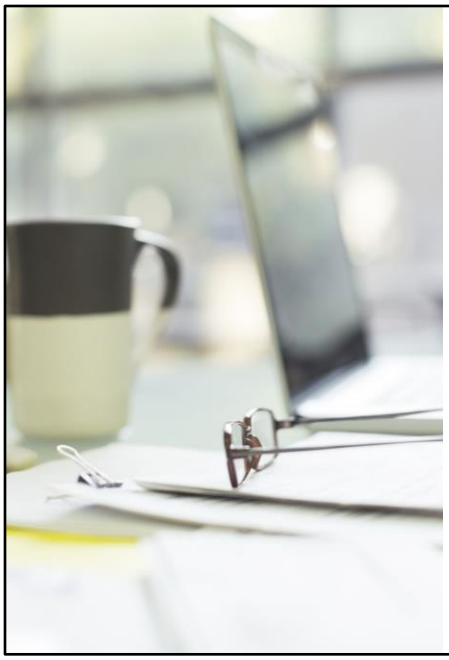


Table UDFs vs Stored Procedures Differences

- Stored procedures can return multiple result sets
  - Rarely used
  - Sometimes not supported by languages
- Stored procedures executed using CALL statement
  - Typically different set of low level APIs used
  - Some frameworks do not support this
- UDFs have some unique functionality
  - Discussion of PIPE follows



Agenda

- Explore types of UDF and UDF syntax
- Compare table UDFs to alternative solutions such as stored procedures and views
- **Discuss some exciting UDF-only features, in particular the use of PIPE within table UDFs**

Returning Results from Complex Logic Traditional Method – SPs and UDFs

- Typically use temporary tables as follows -
  - Define temporary table
  - Define cursor to read temporary table
  - Insert data into temp table whenever required in logic
  - Open cursor
  - Return to application

- Coding is reasonably complicated
- May need additional error handling for table existence
- Particularly in UDFs, getting return to work is challenging

Returning Results from Complex Logic Alternative : UDFs Only

- Simply issue PIPE command for every row to be returned
- Example -

- PIPE (varCol1, varCol2, varCol3, varCol4);

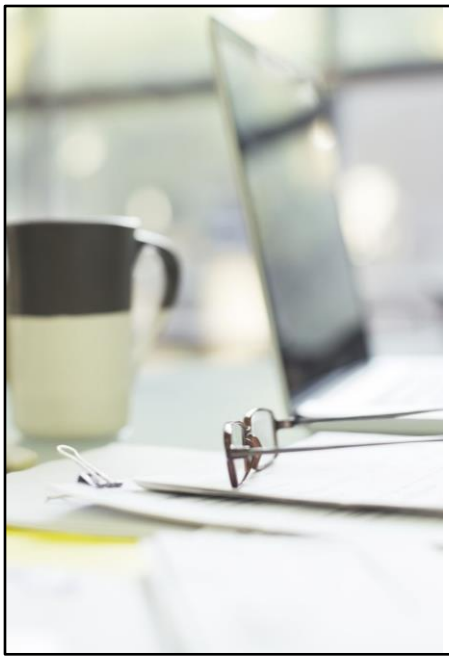
- No temporary tables
- No handling of cursors

Temp Tables vs PIPE Performance Testing

- Test code -
  - Builds and returns a result set
  - Tested with increasing record counts
- Test 1 : Stored procedure using GTT
- Test 2 : Table UDF using GTT
- Test 3 : Table UDF using PIPE

Temp Tables vs PIPE Results

- PIPE syntax is much easier to code and test
- Where results are returned, reasonably close in times
- However …
  - Temp table use soon runs into "log full" issues
  - Tested PIPE returning > 50 million records without issue
- PIPE does not allow final ordering of results
  - Actually very useful for producing reports



Agenda

- Explore types of UDF and UDF syntax
- Compare table UDFs to alternative solutions such as stored procedures and views
- Discuss some exciting UDF-only features, in particular the use of PIPE within table UDFs


IDUG
2022 EMEA Db2 Tech Conference
IDUG
Session C07
Db2 UDFs : Beyond The Basics
Philip Nelson
Please fill out your session evaluation!