Tips and Tricks: Deriving first Normal Form using pySpark / Databricks

In this article, I present a lesser-known inbuilt Spark function – "stack", which is very useful for data wrangling operations. Code and output in this article were written in pySpark using a Databricks workspace.

When working with data, there often exist attributes that repeat themselves. The old classic – invoice data contains invoice header attributes and the repeating elements that form the invoice lines. Traditionally, these repeating groups are managed through data normalisation; the data design process that involves deriving split data entities thereby restructuring the data to its First Normal Form. Recent moves to cloud technology often led to flattened representation of normalised data and reintroducing repeating attributes. Whilst the best practice of data migration can be argued at length, what should one do when faced with repeating elements.

To give context to the discussion, please refer to Figure 1 which shows an arbitrary set of invoice lines presented as a data matrix - split by both rows and columns. This form of the data can present challenges when applying computation and it would be desirable to either have in a single row with repeating columns or preferably in its First Normal Form.

Figure 1: Sample Invoice Data

	InvoiceDate	InvoiceNumber	InvoiceLineSeq	Line1Amount	Line2Amount	Line3Amount	Line4Amount	Line5Amount
1	2019-05-25	8800	1	9.84	null	null	null	null
2	2019-05-25	8800	2	null	19.24	null	null	null
3	2019-05-25	8800	3	null	null	34.35	null	null
4	2019-05-25	8800	5	null	null	null	null	67.02
5	2019-05-25	8800	6	null	null	null	null	null
6	2019-05-25	8801	1	2.52	0	0	null	0
7	2019-05-25	8801	2	null	11.65	null	null	null

An unpivoting operation is required to derive the desired form. This can be achieve using Spark's inbuilt stack function which can expressed as:

"stack(n, expr1, expr2, ... exprk) as (1, ... m)"

The “stack” function takes a list of “k” expressions and separates them into “n” rows. This also implies the “m” columns are formed, where m = k/n rounded up to nearest whole number.

Examples below demonstrate the relationship between k (expressions), m (columns) and n (rows).

Example: 3 expressions, 2 rows and 2 columns

```
1  display(spark.sql("SELECT stack(2, 1, 3, 5) as (a,b)"))
```

▸ (1) Spark Jobs

	a	b
1	1	3
2	5	null

Showing all 2 rows.

Example: 5 expressions, 2 rows and 3 columns

```
1  display(spark.sql("SELECT stack(2, 1, 3, 5, 7, 9) as (a,b,c)"))
```

▸ (1) Spark Jobs

	a	b	c
1	1	3	5
2	7	9	null

Showing all 2 rows.

Example: 6 expressions, 2 rows and 3 columns

```
1  display(spark.sql("SELECT stack(2, 1, 3, 5, 7, 9, 11) as (a,b,c)"))
```

▸ (1) Spark Jobs

	a	b	c
1	1	3	5
2	7	9	11

From the examples above, it can be inferred that the stack function can be formulated to derive a key, value pair (m = 2) for each column the repeating group. In the normalised form the desired number of “n” rows would be equivalent to the number of columns in the repeating group. The Python list comprehension code below shows dynamic calculation the value of “n”.

Figure 2: List comprehension to calculate n-rows

```
repeatCols = [column for column in invoice_df.columns if column.startswith('Line') and column.endswith('Amount')]
n = len(repeatCols)
print('Columns: ', repeatCols)
print('n: ', n)
```

```
Columns:  ['Line1Amount', 'Line2Amount', 'Line3Amount', 'Line4Amount', 'Line5Amount', 'Line6Amount', 'Line7Amount', 'Line8Amount']
n:  8
```

Next, a list of "k" stack expressions needs to be generated. For the desired key-value pairing, the expressions need to alternate between the column name and column value such that the stack function generate n-rows with two columns.

Code below generates a string array called kvCols. Elements of the array alternate between a "string within a string" and a string. List comprehension for column "Line1Price" returns two strings " 'Line1Amount' " and " Line1Amount ". When these two expressions when passed to the "stack" function, the first string (string within string) will evaluate to the column name. The second string will evaluate the column reference thereby yielding column value in the "stack" function.

The string ".join" operator to concatenates and array of strings int a single string.

Figure 3: List comprehension to build "k" expressions

```
kvCols = [f"'{rcol}', {rcol}" for rcol in repeatCols]
print(kvCols)
stackExpr = ', '.join(kvCols)
print(stackExpr)
```

```
["'Line1Amount', Line1Amount", "'Line2Amount', Line2Amount", "'Line3Amount', Line3Amount", "'Line4Amount', Line4Amount", "'
', Line8Amount"]
'Line1Amount', Line1Amount, 'Line2Amount', Line2Amount, 'Line3Amount', Line3Amount, 'Line4Amount', Line4Amount, 'Line5Amoun
```

The fully encoded "stack" function is show in Figure 4.

Figure 4: Stack function encoding

```
unpivotExpr = f'stack({n},' +  ', '.join([f"'{rcol}', {rcol}" for rcol in repeatCols]) + ') as (Key, Value)'
print(unpivotExpr)
```

```
stack(8,'Line1Amount', Line1Amount, 'Line2Amount', Line2Amount, 'Line3Amount', Line3Amount, 'Line4Amount', Line4Amount, 'Li
mount) as (Key, Value)
```

The encoded "stack" function passed as an expression to a dataframe will return the unpivoted Key, Value columns as shown in Figure 5.

Figure 5: Application of stack to DataFrame

```
1  display(invoice_df.select('*', expr(unpivotExpr)))
```

▸ (4) Spark Jobs

	InvoiceDate	InvoiceNumber	Line1Amount	Line2Amount	Line3Amount	Line4Amount	Line5Amount	Line6Amount
1	2019-05-25	8800	9.84	null	null	null	null	null
2	2019-05-25	8800	9.84	null	null	null	null	null
3	2019-05-25	8800	9.84	null	null	null	null	null
4	2019-05-25	8800	9.84	null	null	null	null	null
5	2019-05-25	8800	9.84	null	null	null	null	null
6	2019-05-25	8800	9.84	null	null	null	null	null
7	2019-05-25	8800	9.84	null	null	null	null	null

The results can be cleaned up by removing redundant rows whose “Value” field are null and additionally removing the original unnormalized columns can be removed.

Figure 6: Cleaned up

```
1  display(
2    invoice_df
3    .select('*', expr(unpivotExpr))
4    .where('Value is not Null')
5    .drop(*repeatCols)
6    .withColumnRenamed('Key', 'InvLineNo')
7    .withColumnRenamed('Value', 'InvLineAmount')
8  )
```

▸ (4) Spark Jobs

	InvoiceDate	InvoiceNumber	InvLineNo	InvLineAmount
1	2019-05-25	8800	Line1Amount	9.84
2	2019-05-25	8800	Line2Amount	19.24
3	2019-05-25	8800	Line3Amount	34.35
4	2019-05-25	8800	Line5Amount	67.02
5	2019-05-25	8800	Line6Amount	84.96
6	2019-05-25	8801	Line1Amount	2.52
7	2019-05-25	8801	Line2Amount	0

The final code put all together and with minor refinements is show Figure 7. Note that the order in which the “select” and “where” clauses are written is highly significant.

Figure 7: Full code block

```
repeatCols = [column for column in invoice_df.columns if column.startswith('Line') and column.endswith('Amount')]
unpivotExpr = f'stack({len(repeatCols)},' +  ', '.join([f"'{rcol.replace('Line','').replace('Amount','')}', {rcol}" for
display(
  invoice_df
  .select('*', expr(unpivotExpr))
  .where('InvLineAmount is not Null')
  .drop(*repeatCols)
)
```

▸ (4) Spark Jobs

	InvoiceDate	InvoiceNumber	InvLineNo	InvLineAmount
1	2019-05-25	8800	1	9.84
2	2019-05-25	8800	2	19.24
3	2019-05-25	8800	3	34.35
4	2019-05-25	8800	5	67.02
5	2019-05-25	8800	6	84.96
6	2019-05-25	8801	1	2.52
7	2019-05-25	8801	2	0

Analysts can now proceed onto computation involving joins and aggregations with other datasets as they would normally do with structured data.

To see how we can help your business unlock the value of your data, please contact a Fujitsu Data & AI specialist now.

Contact

Fujitsu Data & AI

+61 3 9924 3000