Uso avanzado de Pig

- Agrupar y agregar datos en Pig
- Coagrupación y unión de datos en Pig
- Funciones definidas por el usuario en Pig

Veremos como procesar múltiples conjuntos de datos utilizando Pig, incluyendo operaciones de unión y coagrupación, así como a procesar datos en estructuras de datos anidadas. También sobre funciones definidas por el usuario y cómo automatizar las rutinas de Pig.

Agrupación de datos en Pig

La agrupación de datos es una técnica común y de uso frecuente en la programación de bases de datos relacionales, normalmente con el propósito de operar sobre datos agregados por una clave, como por ejemplo realizar una operación COUNT, AVG, MIN, MAX, o SUM. Conceptualmente, el enfoque es el mismo en Pig.

Sentencia GROUP

La sentencia GROUP permite agrupar registros por un campo específico. El resultado de una operación GROUP en Pig es una estructura con un registro por cada valor único en el campo por el que se está agrupando. Las tuplas de la relación resultante tienen dos campos:

- Un campo llamado *group* del mismo tipo que el campo por el que se está agrupando de la bolsa de entrada.
- Un campo con el nombre de la relación por la que se está agrupando, que es una bolsa de tuplas de esta relación que contiene el elemento por el que se está agrupando.

Consideremos los conjuntos de datos de ejemplo del Listado 1, que se utilizarán para ilustrar las operaciones de agrupación, coagrupación y unión en Pig.

Listado 1 Conjuntos de datos de muestra

```
--relación salespeople
--schema: salespersonid, name, storeid
1,Henry,100
2,Karen,100
3,Paul,101
4,Jimmy,102
5,Janice,


-- relación stores
--schema: storeid, name
100,Hayward
```

```
101,Baumholder
102,Alexandria
103,Melbourne


-- relación sales
--schema: salespersonid, storeid, salesamt
1,100,38
1,100,84
2,100,26
2,100,75
3,101,55
3,101,46
4,102,12
4,102,67
```

El listado 2 realiza una operación de agrupamiento y muestra la estructura de datos de salida.

Listado 2 Sentencia GROUP

```
grouped = GROUP sales BY salespersonid;
DESCRIBE grouped;
grouped: {group: int,
          sales: {(salespersonid: int,storeid: int,salesamt: int)}}
DUMP grouped;
(1,{(1,100,84),(1,100,38)})
(2,{(2,100,75),(2,100,26)})
(3,{(3,101,46),(3,101,55)})
(4,{(4,102,67),(4,102,12)})
```

Funciones de agregación en Pig

Como ya se ha dicho, normalmente se agrupan los datos como paso previo a la realización de una función de agregación (COUNT, SUM, AVG, etc.). Pig incluye la mayoría de estas funciones como parte de las funciones integradas. El listado 3 muestra una función SUM realizada contra el conjunto de datos agrupados creado en el listado 2.

Listado 3 Función SUM

```
salesbyid = FOREACH grouped GENERATE group AS salespersonid,
            SUM(sales.salesamt) AS total_sales;
```

```
DUMP salesbyid;
(1,122)
(2,101)
(3,101)
(4,79)
```

Para referirse a un campo específico se utiliza la notación *bag.campo* (**sales.salesamt** en el listado anterior)

Para realizar una operación de agregación con todas las tuplas de una bolsa, como contar todas las tuplas de una bolsa o sumar los campos de todas las tuplas de una bolsa, se utiliza la sentencia GROUP ALL. Agrupa todas las tuplas en una estructura que puede contarse mediante el método COUNT o COUNT_STAR (que incluye valores NULL), o sumarse mediante el método SUM. En el Listado 4 se muestra un ejemplo de GROUP ALL con una operación posterior de SUM y COUNT.

Listado 4 Sentencia GROUP ALL

```
allsales = GROUP sales ALL;
DUMP allsales;
(all,{(4,102,67),(4,102,12),(3,101,46),(3,101,55),(2,100,75),...})
--COUNT all tuples in the sales bag
sales_trans = FOREACH allsales GENERATE COUNT(sales);
DUMP sales_trans;
(8)
--SUM all salesamts in the sales bag
sales_total = FOREACH allsales GENERATE SUM(sales.salesamt);
DUMP sales_total;
(403)
```

Sentencias FOREACH anidadas

Además de realizar funciones de agregación contra datos agrupados, se puede operar directamente contra la estructura anidada que se devuelve como resultado de una operación GROUP utilizando una sentencia FOREACH anidada. Hay algunas reglas que se aplican a las operaciones FOREACH anidadas que requieren lo siguiente:

- Sólo se permiten operaciones relacionales dentro de la operación anidada (por ejemplo, LIMIT, FILTER, ORDER).
- `GENERATE` debe ser la última línea del bloque de código anidado.

El listado 5 muestra un ejemplo de una sentencia FOREACH anidada para mostrar las dos principales ventas de cada vendedor utilizando el conjunto de datos agrupados creado en el listado 2.

Listado 5 Sentencia FOREACH anidada

```
top_sales = FOREACH grouped {
            sorted = ORDER sales BY salesamt DESC;
            limited = LIMIT sorted 2;
            GENERATE group, limited.salesamt;
            };
DUMP top_sales;
(1,{(84),(38)})
(2,{(75),(26)})
(3,{(55),(46)})
(4,{(67),(12)})
```

Procesamiento de múltiples conjuntos de datos en Pig

Hasta ahora, todo el procesamiento que hemos realizado ha sido contra un conjunto de datos (una bolsa en una rutina Pig). A menudo, la verdadera magia de la programación de Pig, al igual que con la mayoría de la programación de SQL, implica el análisis de múltiples conjuntos de datos en un solo programa. Veamos algunos de los métodos disponibles para analizar múltiples conjuntos de datos en Pig.

Agrupación de conjuntos de datos

Anteriormente vimos la agrupación de elementos de una sola bolsa, la coagrupación de elementos permite agrupar elementos de múltiples bolsas. El coagrupamiento se realiza utilizando una operación COGROUP que recoge los valores de todas las bolsas en una nueva bolsa. Esta nueva bolsa es bastante similar a la estructura devuelta por la sentencia GROUP; sin embargo, a diferencia de GROUP, que contenía un campo llamado literalmente grupo y una bolsa con el nombre de la relación que se agrupaba, la estructura coagrupada resultante contiene una bolsa por cada bolsa de entrada a la sentencia COGROUP.

Considerando el conjunto de datos que introdujimos en el Listado 1. El listado 6 muestra el uso y la salida de una operación COGROUP (por *storeid*) utilizando este conjunto de datos.

Listado 6 Sentencia COGROUP

```
cogrouped = COGROUP stores BY storeid, salespeople BY storeid;
cogrouped: {group: int,stores: {(storeid: int,name: chararray)},
    salespeople: {(salespersonid: int,name: chararray,storeid: int)}}
DUMP cogrouped;
```

```
(100,{(100,Hayward)},{(2,Karen,100),(1,Henry,100)})
(101,{(101,Baumholder)},{(3,Paul,101)})
(102,{(102,Alexandria)},{(4,Jimmy,102)})
(103,{(103,Melbourne)},{})
(,{},{(5,Janice,)})
```

Unir varios conjuntos de datos

Las operaciones de unión en Pig son análogas a las operaciones de unión que vemos habitualmente en la programación SQL. Las funciones de unión combinan registros de dos conjuntos basados en un campo común.

Tipos de unión

Las uniones operan en dos conjuntos de datos diferentes en los que un campo de cada conjunto de datos se designa como clave (o clave de unión). Los conjuntos de datos se mencionan en el orden en que se especifican. Por ejemplo, el primer conjunto de datos especificado se considera la entidad o conjunto de datos de la izquierda, y el segundo conjunto de datos especificado se considera la entidad o conjunto de datos de la derecha.

Un ***join interno***, a menudo llamado simplemente *join* (donde el "interno" se infiere), devuelve todos los elementos o registros de ambos conjuntos de datos en los que la clave designada está presente en ambos conjuntos de datos.

Una unión externa no requiere que las claves coincidan en ambos conjuntos de datos. Las uniones externas se implementan como una unión externa izquierda, una unión externa derecha o una unión externa completa.

Una **unión externa izquierda** devuelve todos los registros del conjunto de datos izquierdo, o primero, junto con los registros coincidentes sólo (por la clave especificada) del conjunto de datos derecho, o segundo.

Una **unión externa derecha** devuelve todos los registros del conjunto de datos derecho, o segundo, junto con los registros coincidentes sólo (por la clave especificada) del conjunto de datos izquierdo, o primero.

Una **unión externa completa** devuelve todos los registros de ambos conjuntos de datos, tanto si hay una coincidencia de clave como si no.

Uniones en Pig

La bolsa devuelta por una operación JOIN en Pig es una estructura que contiene todos los registros coincidentes de ambas bolsas de entrada.

Consideremos nuestro minorista ficticio del Listado 1, con una lista de tiendas y una lista de vendedores con las tiendas a las que están asignados. La Figura1 es una representación lógica de cómo funciona una operación JOIN en Pig.

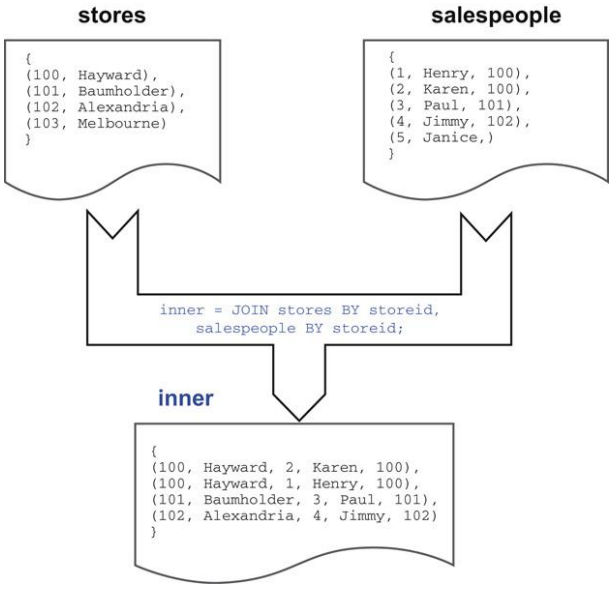

Figura 1 Inner Join en Pig.

Una operación de *join interna* muestra todas las tiendas y vendedores con un *storeid* coincidente. Los vendedores sin tiendas y las tiendas sin vendedores no se incluyen.

Consejo: Optimización de Joins en Pig

A diferencia de la mayoría de las bases de datos, Pig no utiliza índices o estadísticas para optimizar el JOIN, por lo que las optimizaciones que puede proporcionar son esenciales para maximizar el rendimiento. Una de estas optimizaciones es el orden en el que se referencian las bolsas en una sentencia JOIN. Un axioma sencillo de recordar es "unir lo grande con lo pequeño". Esto significa referenciar primero la mayor de las dos bolsas de entrada (la que contiene más tuplas si se conoce), seguida de la menor de las dos bolsas. Esto parecerá extraño para los usuarios que vienen de la programación de bases de datos relacionales, pero a diferencia de los sistemas de bases de datos relacionales, las uniones son relativamente ineficientes en Hadoop.

Las figuras 2, 3 y 4 representan lógicamente un LEFT OUTER JOIN, RIGHT OUTER JOIN y FULL OUTER JOIN en Pig.

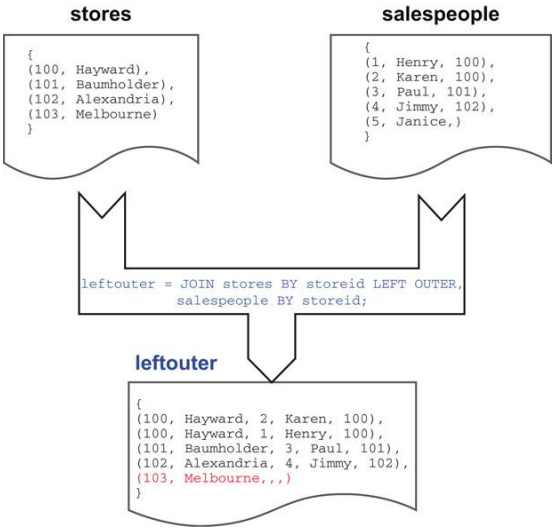

Figura 2 Unión externa izquierda en Pig.

Una operación de *left outer join* muestra todas las tiendas con o sin vendedores.

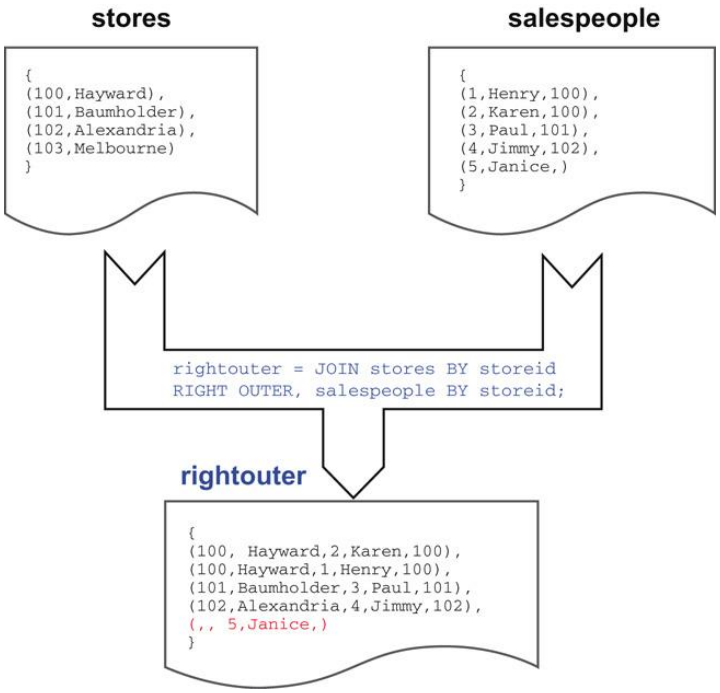

Figura 3 Right Outer Join en Pig.

Una operación *right outer join* muestra todos los vendedores con o sin tiendas.

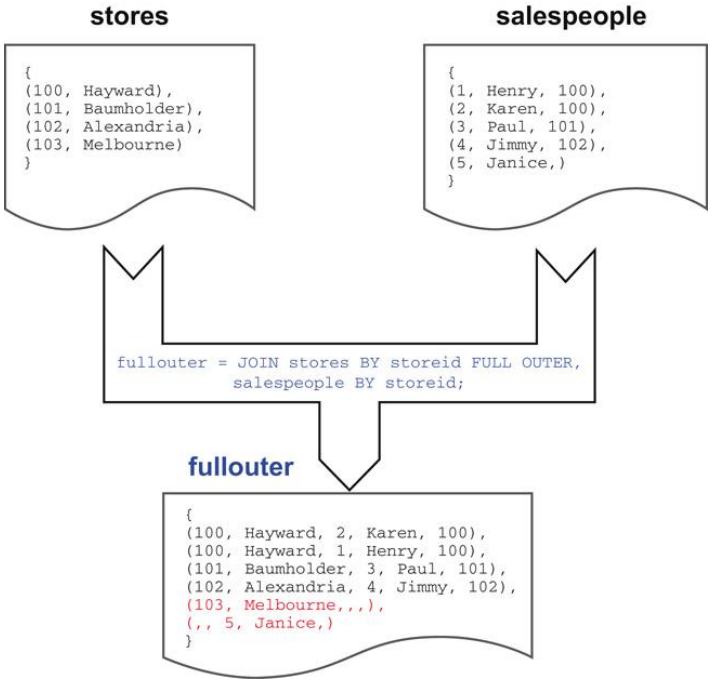

Figura 4 Unión externa completa en Pig.

Una operación full outer join muestra todos los vendedores con todas las tiendas, haya o no asociación.

Declaración JOIN

Una vez comprendidos los tipos de join disponibles y cómo se devuelven los datos de las operaciones join en Pig, veamos algunos ejemplos. El listado 8 muestra ejemplos utilizando

nuestro conjunto de datos, mostrando los datos y el esquema de las bolsas devueltas por una sentencia JOIN. Obsérvese el identificador de objeto distintivo asignado a los campos de la bolsa devuelta. La sintaxis **"::"** se denomina operador de desambiguación, ya que puede haber conflictos de espacio de nombres si los campos tienen el mismo nombre en ambas bolsas. El listado 7 muestra las diferentes operaciones JOIN en Pig.

Listado7 Declaración JOIN

```
--Stores and their salespeople

inner = JOIN stores BY storeid, salespeople BY storeid;

DESCRIBE inner;

inner: {stores::storeid: int,stores::name: chararray,

salespeople::salespersonid: int,salespeople::name: chararray,

salespeople::storeid: int}

DUMP inner;

(100,Hayward,2,Karen,100)

(100,Hayward,1,Henry,100)

(101,Baumholder,3,Paul,101)

(102,Alexandria,4,Jimmy,102)


--All stores with and without salespeople

leftouter = JOIN stores BY storeid LEFT OUTER, salespeople BY storeid;

DUMP leftouter;

(100,Hayward,2,Karen,100)

(100,Hayward,1,Henry,100)

(101,Baumholder,3,Paul,101)

(102,Alexandria,4,Jimmy,102)

(103,Melbourne,,,)


--All salespeople with and without stores

rightouter = JOIN stores BY storeid RIGHT OUTER, salespeople BY storeid;
```

```
DUMP rightouter;

(100,Hayward,2,Karen,100)

(100,Hayward,1,Henry,100)

(101,Baumholder,3,Paul,101)

(102,Alexandria,4,Jimmy,102)

(,,5,Janice,)


--All stores, all salespeople

fullouter = JOIN stores BY storeid FULL OUTER, salespeople BY storeid;

DUMP fullouter;

(100,Hayward,2,Karen,100)

(100,Hayward,1,Henry,100)

(101,Baumholder,3,Paul,101)

(102,Alexandria,4,Jimmy,102)

(103,Melbourne,,,)

(,,5,Janice,)
```

Consejo: Filtrar pronto, filtrar a menudo

Como habrás notado, las bolsas devueltas por las operaciones GROUP, COGROUP y JOIN contendrán campos duplicados; por ejemplo, el campo que se está agrupando, coagrupando o uniendo estará duplicado en las estructuras de datos devueltas. Una convención común es seguir una operación COGROUP con una operación FOREACH para eliminar los campos duplicados. Esto ayudará al optimizador de Pig a hacer más eficiente la ejecución de sus programas.

Operaciones Set en Pig

Pig soporta varias operaciones de conjunto comunes que operan en toda la tupla en lugar de en los campos de una tupla. Veamos ahora algunas de ellas, utilizando los dos conjuntos odds y fibonacci que se muestran en la Figura 5.

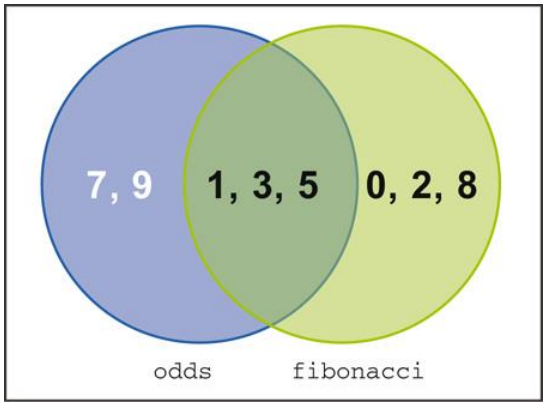

Figura 5 Ejemplos de conjuntos.

Declaración UNION

A veces es necesario concatenar dos conjuntos de datos entre sí, es decir, combinar las tuplas de una bolsa con las de otra. Esto se consigue con el operador UNION. El listado 8 muestra el operador UNION.

Listado8 Declaración UNION

```
unioned = UNION odds, fibonacci;

DUMP unioned;
(0)
(1)
(2)
(3)
(5)
(8)
(1)
(3)
(5)
(7)
(9)
```

Nota: UNION no conserva el orden ni filtra los duplicados

Hay que tener en cuenta que UNION no elimina los duplicados. Para lograrlo, debe seguir una UNION con una operación DISTINCT, de la que hablaremos a continuación. Además, UNION no preserva el orden de ninguna de las bolsas de entrada. Para ello se debe utilizar la sentencia ORDER después de la operación UNION.

Otro ejemplo:

Sentencia DISTINCT

Un requisito común es eliminar registros o tuplas duplicadas de una bolsa en la que cada campo es idéntico a una o más tuplas. El listado 9 muestra el uso de la sentencia DISTINCT.

Listado 9 Sentencia DISTINCT

```
no_duplicates = DISTINCT unioned;
DUMP no_duplicates;
(0)
(1)
(2)
(3)
(5)
(7)
(8)
(9)
```

Sentencia SUBTRACT

El operador SUBTRACT es otra operación de conjunto común que devuelve tuplas de una bolsa que no están presentes en otra bolsa. El listado 10 demuestra el uso del operador SUBSTRACT. Observe que las tuplas que se niegan entre sí, en este caso los números impares, se representan como conjuntos vacíos.

Listado 10 Declaración SUBTRACT

Supongamos que tenemos dos conjuntos de datos:

Archivo `clientes1.txt`:

```
1,Juan

2,Ana

3,Luis

4,Carmen
```

Archivo `clientes2.txt`:

```
2,Ana

4,Carmen
```

Script Pig:

```
-- Cargar los datos

clientes_a = LOAD 'clientes1.txt' USING PigStorage(',') AS (id:int,
nombre:chararray);

clientes_b = LOAD 'clientes2.txt' USING PigStorage(',') AS (id:int,
nombre:chararray);


-- Restar la segunda relación de la primera

solo_a = SUBTRACT clientes_a, clientes_b;


-- Mostrar resultado
```

```
DUMP solo_a;

Salida esperada:

(1,Juan)

(3,Luis)
```

Es decir, se devuelven los registros que están en `clientes_a` pero no en `clientes_b`.

Sentencia CROSS

Pig también incluye un operador CROSS que produce un producto cartesiano de todas las tuplas de una bolsa con todas las tuplas de otra bolsa. Esto puede ser útil para aplicaciones de ciencia de datos en las que se desea probar todas las combinaciones posibles de registros de dos conjuntos de datos dispares.

El listado 11 muestra un ejemplo de una operación CROSS con una salida abreviada.

Listado 11 Declaración CROSS

```
crossed = CROSS odds, fibonacci;

DUMP crossed;
(9,8)
(9,5)
(9,3)
(9,2)
(9,1)
(9,0)
(7,8)
(7,5)
(7,3)
...
```

Precaución: Las uniones cruzadas pueden crear conjuntos de datos masivos

Las uniones cruzadas o productos cartesianos de dos o más conjuntos de datos pueden dar lugar a volúmenes de datos extremadamente grandes y deben utilizarse con moderación. Aunque Hadoop es una plataforma de Big Data por definición, no debe crear inadvertidamente un problema de big data donde no tenía por qué existir.

Dividir conjuntos de datos en Pig

No sólo puede combinar conjuntos de datos utilizando el operador UNION, sino que también puede dividir una bolsa en varias bolsas utilizando el operador SPLIT. SPLIT puede utilizarse para crear conjuntos de datos independientes que no tienen por qué ser mutuamente excluyentes. Estos conjuntos de datos se pueden procesar en paralelo o se pueden utilizar para separar la salida.

El listado 12 proporciona un ejemplo del operador SPLIT.

Listado 12 Declaración SPLIT

Supongamos que tenemos un archivo `alumnos.txt` con los datos:

```
1,Juan,80
2,Ana,60
3,Luis,45
4,Carmen,90
5,Pedro,70
```

Cada registro tiene:
`(id:int, nombre:chararray, nota:int)`

Script Pig:

```
-- Cargar los datos
alumnos = LOAD 'alumnos.txt' USING PigStorage(',') AS (id:int,
nombre:chararray, nota:int);


-- Dividir los alumnos en tres grupos según la nota
SPLIT alumnos INTO
    aprobados IF (nota >= 70),
    regulares IF (nota >= 50 AND nota < 70),
    suspensos IF (nota < 50);


-- Mostrar cada subconjunto
DUMP aprobados;
DUMP regulares;
DUMP suspensos;
```

Salida esperada:

```
aprobados:
(1,Juan,80)
(4,Carmen,90)
(5,Pedro,70)
regulares:
(2,Ana,60)
suspensos:
(3,Luis,45)
```

Funciones definidas por el usuario en Pig

Aunque Pig proporciona algunas funciones incorporadas, de las que hablé en la hora anterior, a menudo hay requisitos para realizar funciones más avanzadas para las que no hay opciones incorporadas. Afortunadamente, Pig soporta y fomenta el uso de funciones incorporadas que pueden ser escritas o obtenidas de la comunidad Pig. Vamos a verlas ahora.

Escribir UDFs para Pig

Como la mayoría de los proyectos del ecosistema Hadoop, Pig es extensible por diseño. El ejemplo más obvio de esto es su soporte para funciones definidas por el usuario (UDFs). Las UDFs en Pig se clasifican generalmente en una de las siguientes categorías:

- Funciones de evaluación
- Funciones de carga/almacenamiento

A diferencia de las extensiones definidas por el usuario para MapReduce, que sólo pueden implementarse en Java, Pig permite escribir UDFs en múltiples lenguajes. Los lenguajes incluyen los siguientes:

- Java
- Python y Jython (código Python compilado para ejecutarse en una JVM)
- JavaScript
- Groovy
- JRuby

Por razones de rendimiento, se recomiendan las plataformas basadas en JVM como Java, Jython, JRuby y Groovy pero también se puede usar Python. Hay mucha más información disponible sobre cómo escribir e implementar UDFs en los distintos lenguajes en https://pig.apache.org.

Pig tiene muchas funciones integradas (COUNT, MAX, LOWER, TOKENIZE, etc.), pero si necesitas algo más específico, puedes definir la tuya propia.

Operador de *Stream* en Pig

Las funciones Map y Reduce pueden ser escritas en otros lenguajes que no sean Java usando la API MapReduce Streaming. Pig expone la misma funcionalidad usando el operador STREAM.

El operador STREAM permite escribir funciones en lenguajes como Perl, BASH y otros lenguajes no soportados para el desarrollo de UDF. Al igual que la API de flujo de MapReduce, los registros se leen como texto delimitado por tabulaciones en STDIN, y los registros se emiten desde su función personalizada como texto delimitado por tabulaciones utilizando STDOUT.

Dada una función en un script Perl llamado `myfunction.pl`, el Listado13 demuestra cómo usar el operador STREAM.

Listado 13 Declaración STREAM

```
DEFINE MYFUNCTION `myfunction.pl` SHIP('myfunction.pl');

output = STREAM recs THROUGH MYFUNCTION AS (col1:chararray, col2:long);

...
```

Ejemplo completo: UDF en Python

Supongamos que tienes un archivo `numeros.txt` con:

```
2
4
5
10
```

Y quieres **calcular el cuadrado** de cada número usando una función en Python.

Archivo Python: `mi_udf.py`

```
#!/usr/bin/env python3
import sys
for line in sys.stdin:
    x = line.strip()
    if not x:
        continue
    print(int(x) * int(x))
```

Guarda este archivo en el mismo directorio y asegúrate de darle permisos de ejecución:

```
chmod +x mi_udf.py
```

Script Pig: `ejemplo_udf.pig`

```
-- Cargar los datos
datos = LOAD 'numeros.txt' USING PigStorage() AS (num:int);
-- Definir la función externa (Python)
DEFINE cuadrado `python mi_udf.py`;
-- Aplicar la función a los datos
resultado = STREAM datos THROUGH cuadrado;
-- Mostrar el resultado
DUMP resultado;
```

Salida esperada:

```
4
16
25
100
```