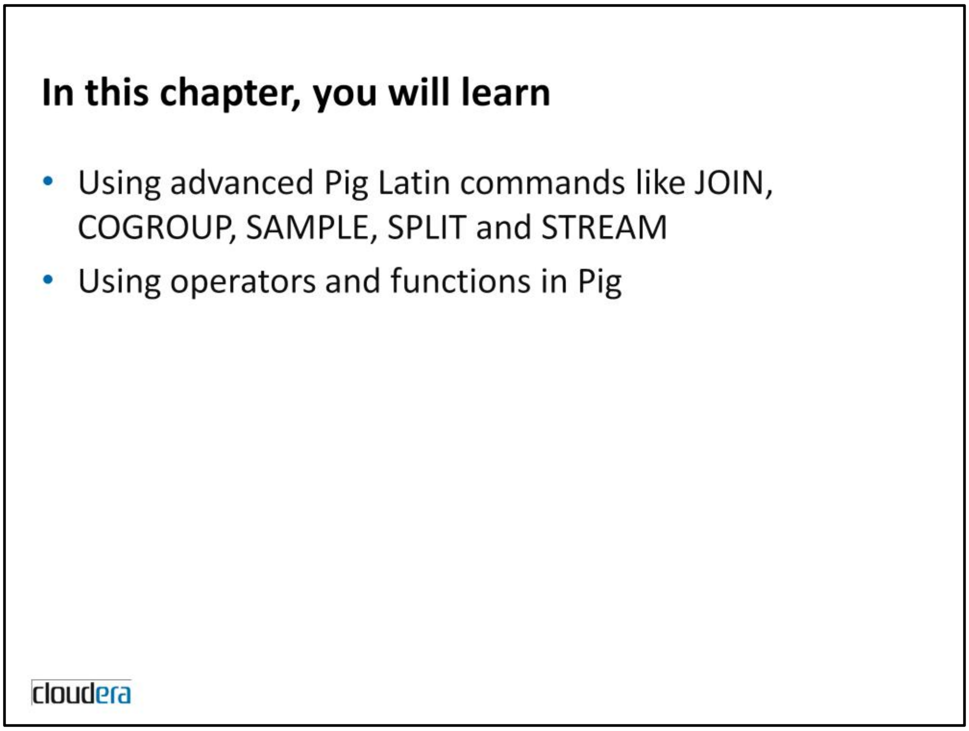
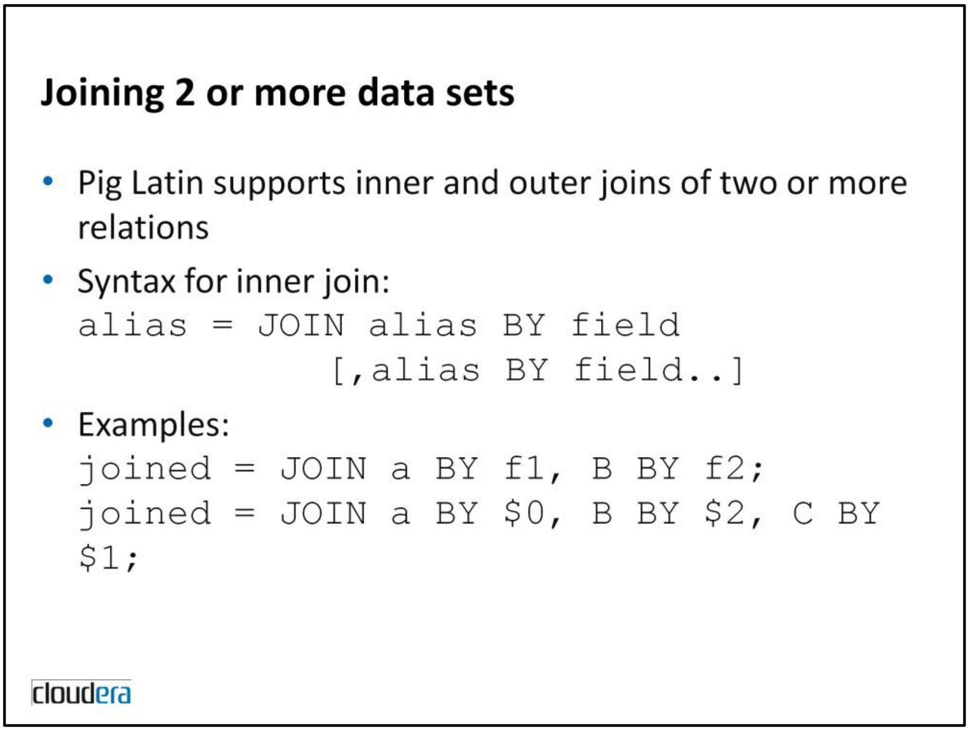
Advanced Pig Latin



In this chapter, you will learn

- Using advanced Pig Latin commands like JOIN, COGROUP, SAMPLE, SPLIT and STREAM
- Using operators and functions in Pig



Joining 2 or more data sets

- Pig Latin supports inner and outer joins of two or more relations
- Syntax for inner join:

```
alias = JOIN alias BY field
              [,alias BY field..]
```

- Examples:

```
joined = JOIN a BY f1, B BY f2;
joined = JOIN a BY $0, B BY $2, C BY
$1;
```

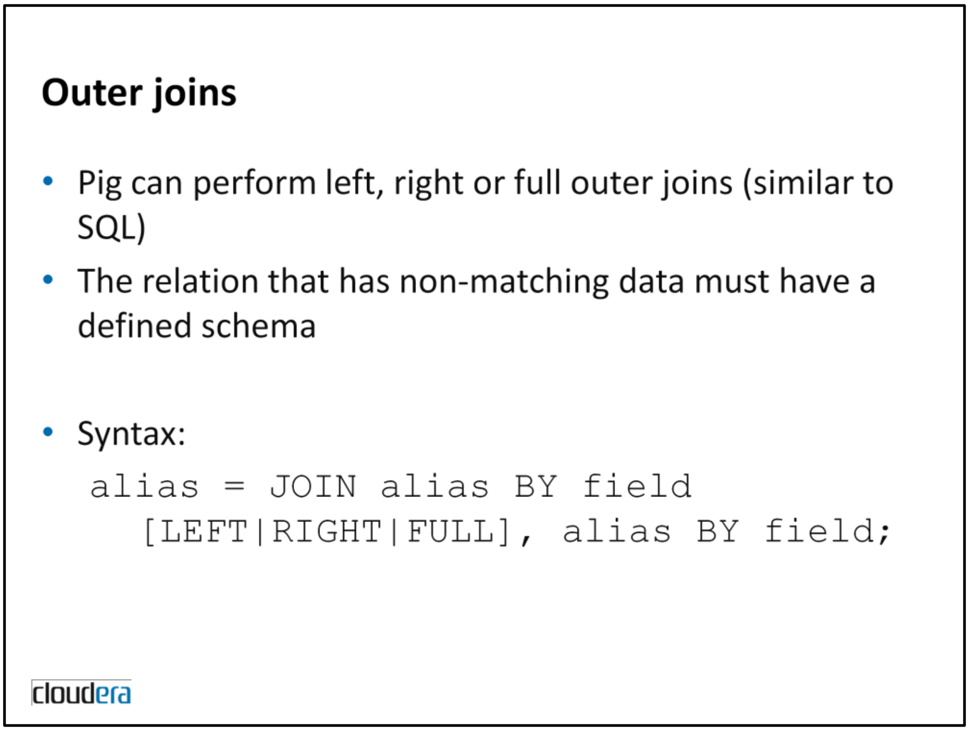
Pig Latin supports joining 2 or more data sets that have a field in common. Both inner and outer joins are supported. To perform an **inner** join, use the JOIN operator:

```
alias = JOIN alias BY expression [,alias BY expression..]
```

Example:

```
grunt> cat pets.txt
Doug       cat
Tom        dog
Mike       cat
Sarah      fish
grunt> cat hobbies.txt
Doug       reading
Tom        swimming
Mike       biking
Sarah      singing
grunt> pets = LOAD 'pets.txt';
grunt> hobbies = LOAD 'hobbies.txt';
grunt> joined = JOIN pets BY $0, hobbies BY $0;
grunt> DUMP joined;
(Tom,dog,Tom,swimming)
(Doug,cat,Doug,reading)
(Mike,cat,Mike,biking)
(Sarah,fish,Sarah,singing)
```

Note: Only equi-joins are supported.



Outer joins

- Pig can perform left, right or full outer joins (similar to SQL)
- The relation that has non-matching data must have a defined schema
- Syntax:

```
alias = JOIN alias BY field
   [LEFT|RIGHT|FULL], alias BY field;
```

In the inner join example, everyone had a hobby and a pet. However, let's say Philip has only a hobby:
Philip reading

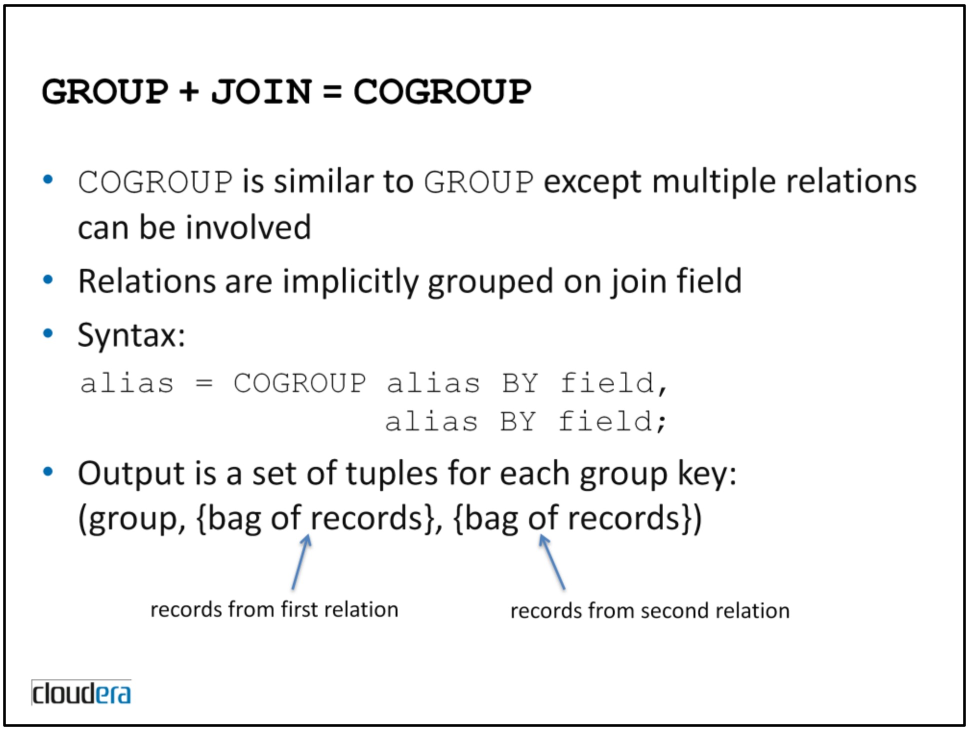
Philip would not be returned in an inner join. However, we can use an outer join to include the non-matching records from one or both relations.

Example:

```
grunt> hobbies = LOAD 'hobbies.txt';
grunt> pets = load 'pets.txt' AS (name:chararray,
hobby:chararray);
grunt> joined = JOIN pets BY name RIGHT, hobbies BY $0;
grunt> DUMP joined;

(Tom,dog,Tom,swimming)
(Doug,cat,Doug,reading)
(Mike,cat,Mike,biking)
(Sarah,fish,Sarah,singing)
(,,Philip,reading)
```

Note: it is necessary to provide a schema for pets because it needs to produce nulls for the non-matching records.



GROUP + JOIN = COGROUP

- COGROUP is similar to GROUP except multiple relations can be involved
- Relations are implicitly grouped on join field
- Syntax:

```
alias = COGROUP alias BY field,
                alias BY field;
```

- Output is a set of tuples for each group key:
  (group, {bag of records}, {bag of records})

records from first relation

records from second relation

COGROUP is a generalization of GROUP that can involve more than 1 relation.

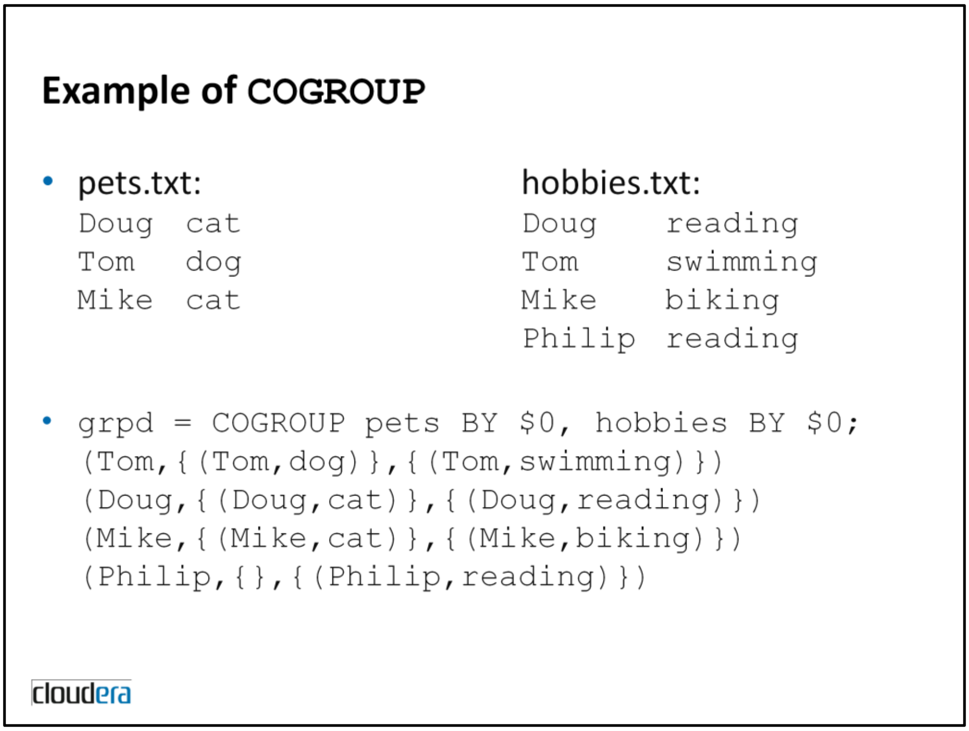
The syntax is:

```
alias3 = COGROUP alias1 BY field [INNER], alias2 BY field
[INNER];
```

The relations (alias1 and alias2) will be joined and grouped on the field they have in common. By default this is a **full outer join**, but the keyword INNER can be included for one or both relations.

The result is a relation where each record is (group key, bag-of-records, bag-of-records). The bags contain records from one of the input relations that match this group.

Although you can use COGROUP in place of a regular GROUP, it is a good idea to use GROUP when only one relation is used and COGROUP for multiple relations.



Example of COGROUP

- pets.txt: hobbies.txt:

```
Doug  cat                       Doug    reading
Tom   dog                       Tom     swimming
Mike  cat                       Mike    biking
                                Philip  reading
```

- `grpd = COGROUP pets BY $0, hobbies BY $0;`

```
(Tom,{(Tom,dog)},{(Tom,swimming)})
(Doug,{(Doug,cat)},{(Doug,reading)})
(Mike,{(Mike,cat)},{(Mike,biking)})
(Philip,{},{(Philip,reading)})
```

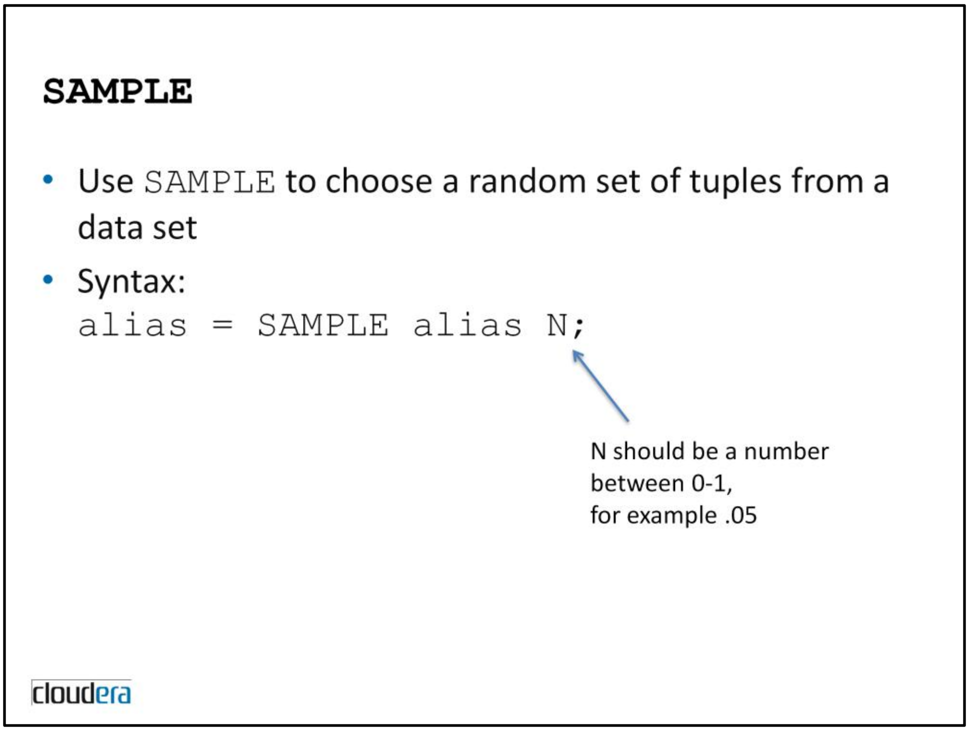
Take these two files as input:

pets.txt   hobbies.txt:

```
Doug      cat                        Doug      reading
Tom       dog                        Tom       swimming
Mike      cat                        Mike      biking
                                     Philip    reading
```

If these files are COGROUPed on the person's name (field $0) the output would be (formatting changed for readability):

```
(Tom,     {(Tom,dog)},         {(Tom,swimming)})
(Doug,    {(Doug,cat)},        {(Doug,reading)})
(Mike,    {(Mike,cat)},        {(Mike,biking)})
(Philip,  {},                  {(Philip,reading)})
```



SAMPLE

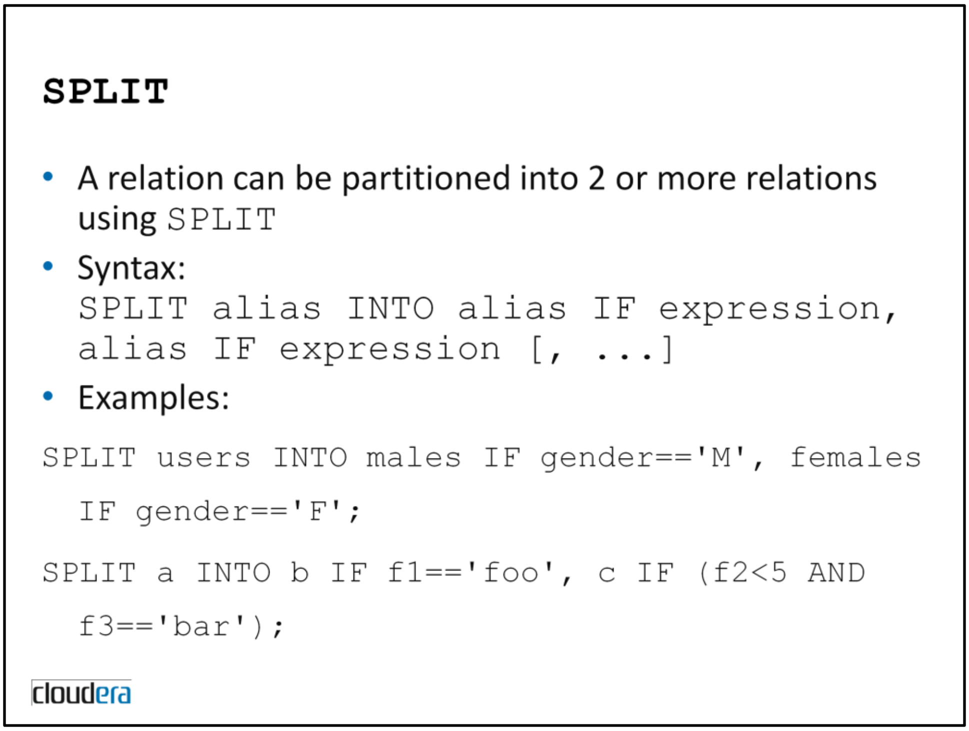
- Use SAMPLE to choose a random set of tuples from a data set
- Syntax:

```
alias = SAMPLE alias N;
```

N should be a number between 0-1, for example .05

Sometimes it is useful to select a small subset of rows from a data set. SAMPLE do just that. When specifying the size of the relation, use a number between zero and one (such as .05 for 5% of the data).

Note that the input relation needs to be fully read in order to create the SAMPLE.



SPLIT

- A relation can be partitioned into 2 or more relations using `SPLIT`
- Syntax:
  `SPLIT alias INTO alias IF expression, alias IF expression [, ...]`
- Examples:

```
SPLIT users INTO males IF gender=='M', females
  IF gender=='F';
SPLIT a INTO b IF f1=='foo', c IF (f2<5 AND
  f3=='bar');
```

cloudera

`SPLIT` is a useful command for dividing a relation into 2 or more relations. The input relation only needs to be scanned once to create the output data sets. The syntax is:

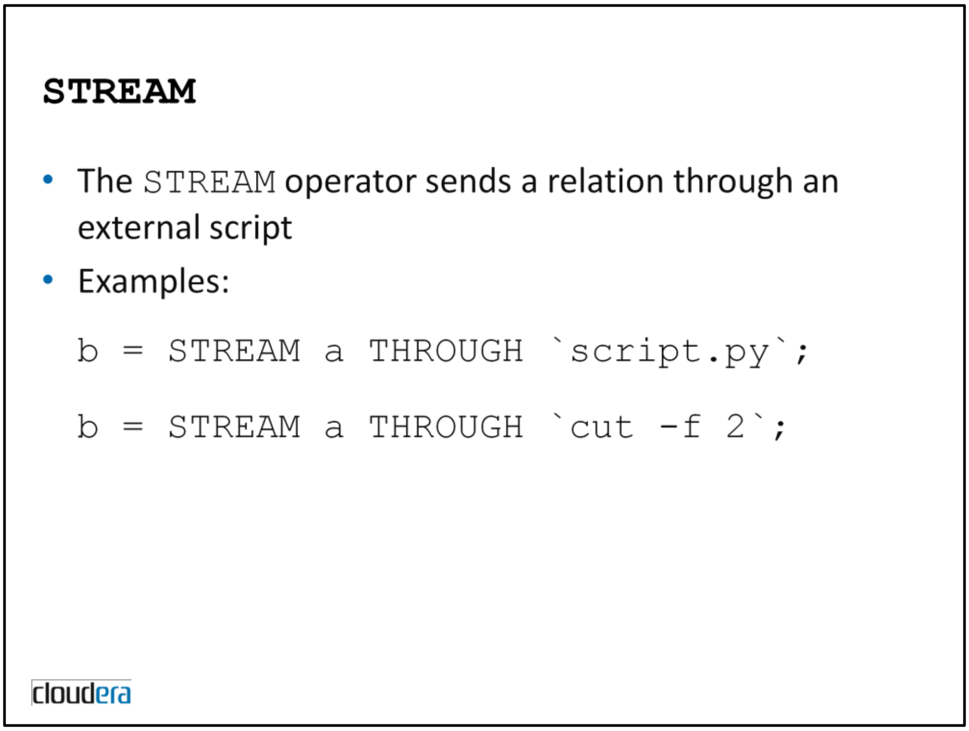
```
SPLIT alias INTO alias IF expression, alias IF expression
[, alias IF expression..];
```

The expression can be simple comparison operator (e.g., `f1=='foo'`) or a compound expression that uses `AND/OR`. For compound expressions, enclose the expression in parenthesis.

The resulting relations can contain the same or different records from the input relation. For example:

```
SPLIT users INTO males IF gender=='M', engineers IF
occupation=='engineer';
```

In the above statement, users becomes two relations: males and engineers. Some males are also engineers and would be outputted to both relations.



STREAM

- The STREAM operator sends a relation through an external script
- Examples:

```
b = STREAM a THROUGH `script.py`;

b = STREAM a THROUGH `cut -f 2`;
```

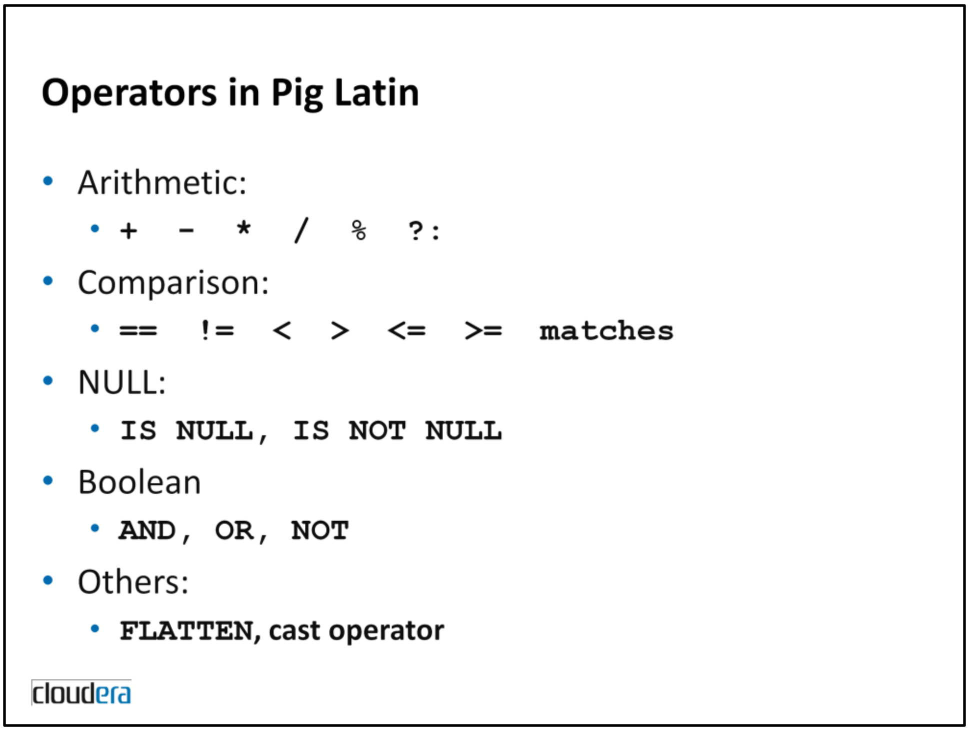
Like Hadoop streaming, Pig can send a data set to a script or program. The script reads the incoming records as tab-delimited lines and should return lines of tab-delimited fields. Common UNIX utilities can also be used, such as `cut`.

This example extracts the second field of the records in `a`:

```
b = STREAM a THROUGH `cut -f 2`;
```

Optionally, you can DEFINE a command for the script (especially useful if the command will be reused):

```
DEFINE mycmd `script.py`;
b = STREAM a THROUGH mycmd;
```



Operators in Pig Latin

- Arithmetic:
  - `+ - * / % ?:`
- Comparison:
  - `== != < > <= >= matches`
- NULL:
  - **`IS NULL, IS NOT NULL`**
- Boolean
  - **`AND, OR, NOT`**
- Others:
  - **`FLATTEN`, cast operator**

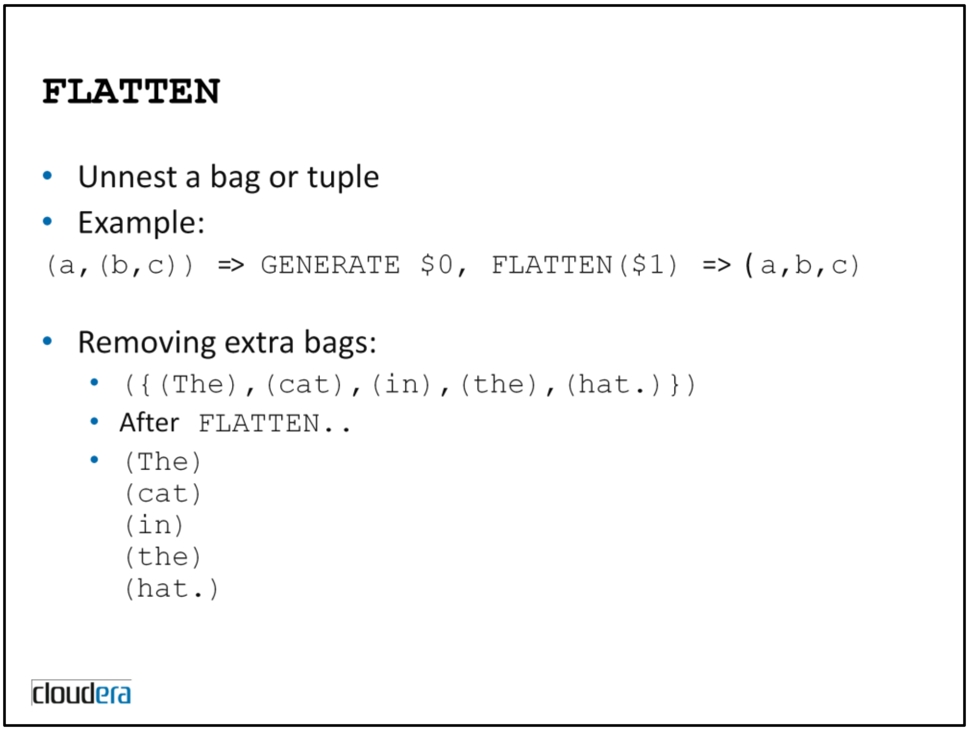
cloudera

Pig Latin supports several operators very similar to most programming languages.

Arithmetic operators:
`+` (addition)
`-` (subtraction)
`*` (multiplication)
`/` (division)
`%` (modulo)
`?` (condition ? if-true : if-false), for example, `(name=='Doug' ? 'Found Doug' : 'Not Doug')`

Comparison operators:
== (equal)
`!=` (not equal)
< (less than), > (greater than)
<= (less than or equal), >= (greater than or equal)
`matches` (regular expression matching, using Java regex format)

Others:
`IS [NOT] NULL` (for NULL comparisons)
`AND, OR, NOT` (compound statements)
`FLATTEN` (see next page)
cast operator (change or identify a data type), for example: (int)$1



FLATTEN

- Unnest a bag or tuple
- Example:

`(a,(b,c)) => GENERATE $0, FLATTEN($1) => (a,b,c)`

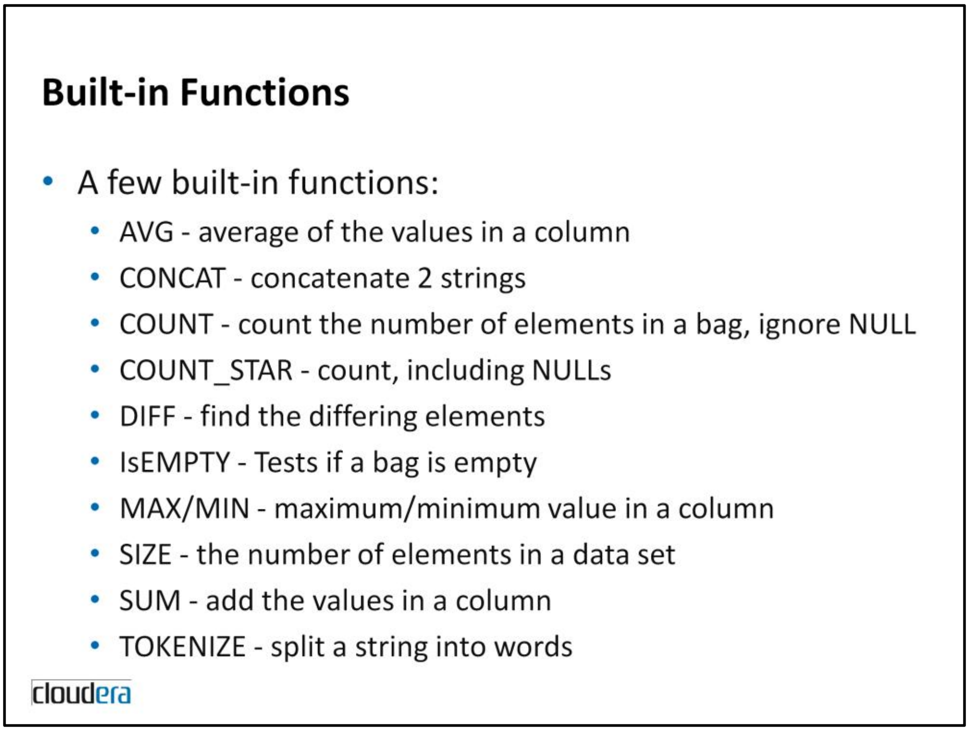
- Removing extra bags:
  - `({(The),(cat),(in),(the),(hat.)})`
  - After `FLATTEN..`

  -
    ```
    (The)
    (cat)
    (in)
    (the)
    (hat.)
    ```

`FLATTEN` is used to remove a level of nesting from a bag or tuples. For example: (a, (b,c)) can be "flattened" into (a,b,c)

FLATTEN is often used to unnest the result of a function. For example:

```
> cat data.txt;
The cat in the hat.
> lines = LOAD 'data.txt' USING TextLoader();
> DUMP lines:
(The cat in the hat.)
> words = FOREACH lines GENERATE TOKENIZE($0);
> DUMP words;
({(The),(cat),(in),(the),(hat.)})
> flat = FOREACH words GENERATE FLATTEN($0);
> DUMP flat;
(The)
(cat)
(in)
(the)
(hat.)
```



Built-in Functions

- A few built-in functions:
  - AVG - average of the values in a column
  - CONCAT - concatenate 2 strings
  - COUNT - count the number of elements in a bag, ignore NULL
  - COUNT_STAR - count, including NULLs
  - DIFF - find the differing elements
  - IsEMPTY - Tests if a bag is empty
  - MAX/MIN - maximum/minimum value in a column
  - SIZE - the number of elements in a data set
  - SUM - add the values in a column
  - TOKENIZE - split a string into words

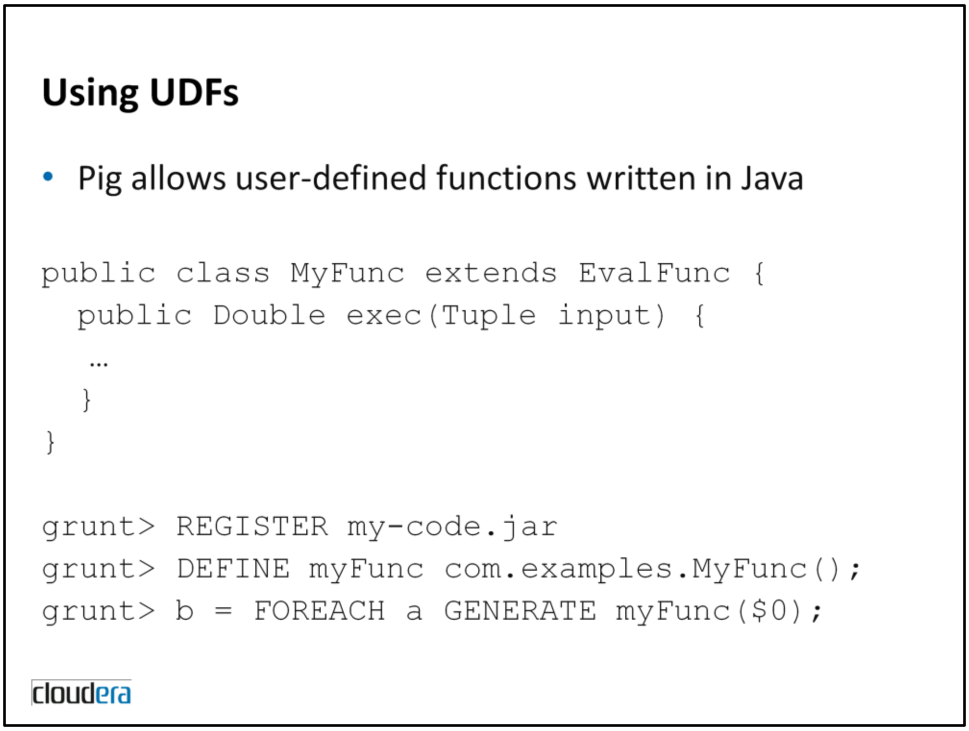
cloudera

Pig comes with a few built-in functions. Many are aggregate functions such as `AVG`, `COUNT`, `MAX`, `MIN` and `SUM`. There are also functions for concatenating strings, find differences between elements, and tokenizing a string.

The aggregate functions require a previous `GROUP` statement. Use `GROUP ALL` for global calculations:

```
users = LOAD 'data';
grpd = GROUP users ALL;
total = FOREACH grpd GENERATE COUNT(users);
```

Remember that function names are case-sensitive.



Using UDFs

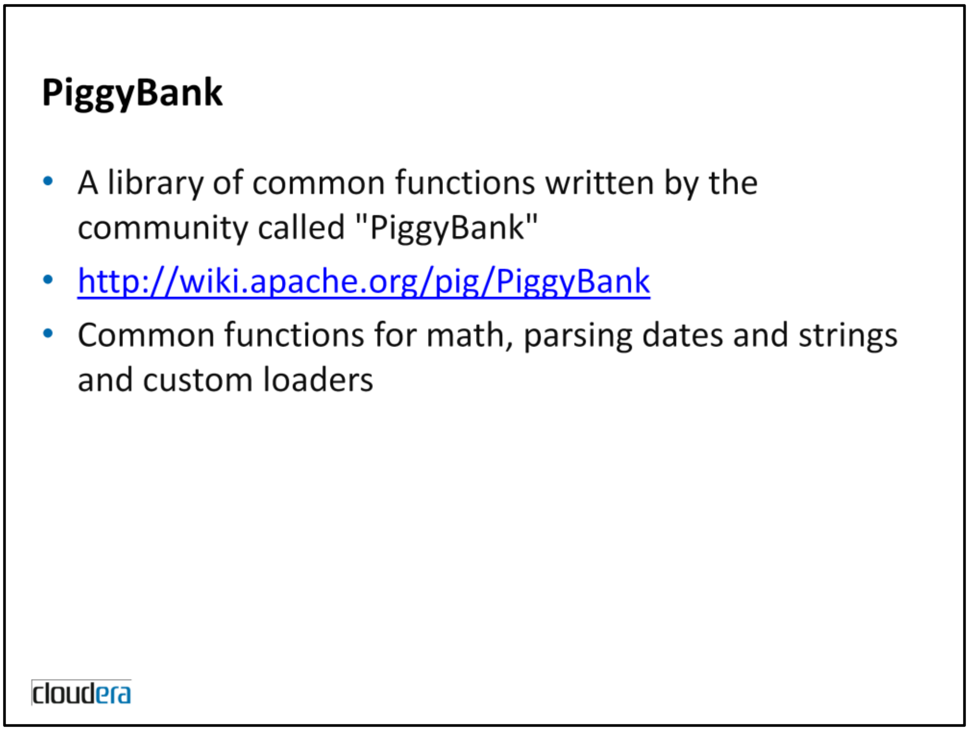
- Pig allows user-defined functions written in Java

```
public class MyFunc extends EvalFunc {
  public Double exec(Tuple input) {
   …
  }
}

grunt> REGISTER my-code.jar
grunt> DEFINE myFunc com.examples.MyFunc();
grunt> b = FOREACH a GENERATE myFunc($0);
```

One way of extending the Pig Latin language is by writing user-defined functions (UDFs). These currently need to be written in Java. The steps required are:

1. Write a class that extends `EvalFunc` and implement the `exec` method.
2. Compile and package into a jar
3. Tell pig about the jar using the `REGISTER` keyword
4. Optionally DEFINE a function name. Without this step, the fully-qualified class name is the function name (e.g., com.examples.MyFunc())
5. Invoke your function in the pig script



PiggyBank

- A library of common functions written by the community called "PiggyBank"
- http://wiki.apache.org/pig/PiggyBank
- Common functions for math, parsing dates and strings and custom loaders

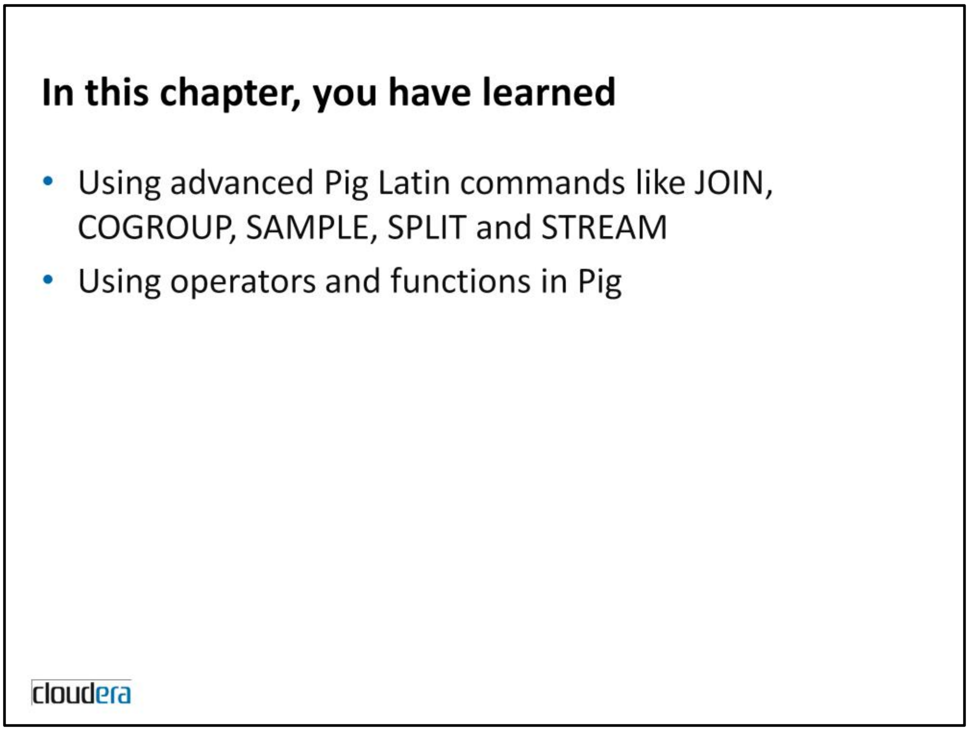
There is also a set of functions written by the community. It's called the PiggyBank.
Details can be found here: http://wiki.apache.org/pig/PiggyBank

Dates:
- CustomFormatToISO - convert arbitrary date format to ISO format
- UnixToISO and ISOToUnix - convert between ISO format and Unix timestamps

Math:
- ABS - absolute value of a number
- LOG - natural log of a number
- POW - a number raised to a power
- RANDOM - return a random number
- ROUND - round numbers to the closest long

Strings:
- INDEXOF - search for a string
- LENGTH - the length of a string
- LOWER - convert to lowercase
- SUBSTRING - extract a portion of a string
- UPPER - convert to uppercase

Storage:
- MyRegExLoader - parse a file given a user-defined regular expression
- SequenceFileLoader - read Hadoop SequenceFile format
- XMLLoader - parses XML files by a user-supplied start/end tag



In this chapter, you have learned

- Using advanced Pig Latin commands like JOIN, COGROUP, SAMPLE, SPLIT and STREAM
- Using operators and functions in Pig