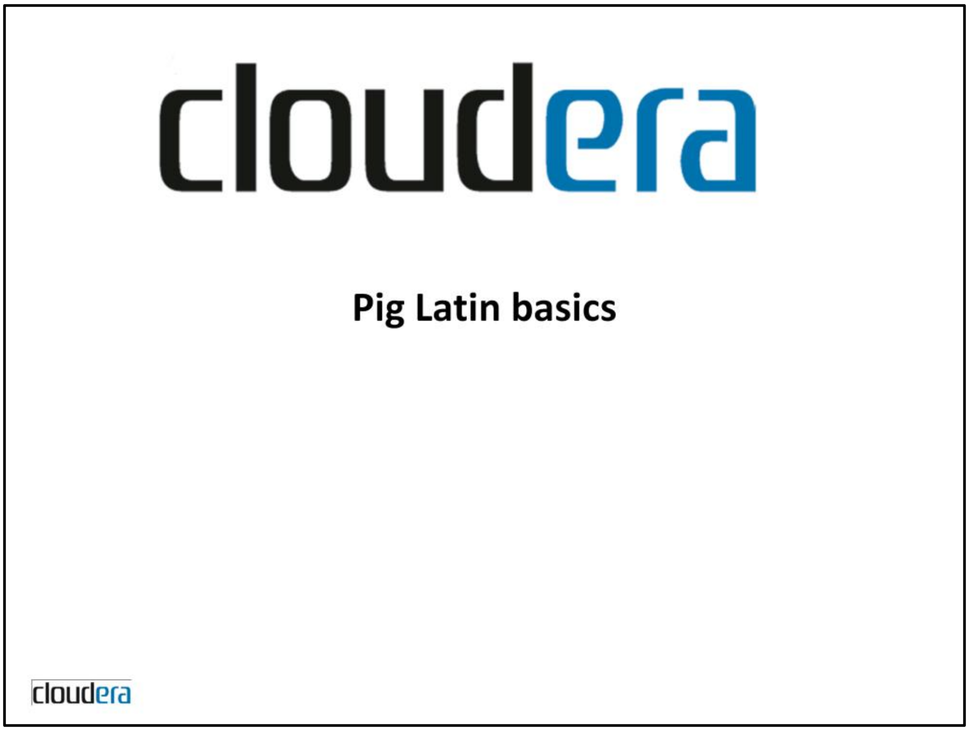
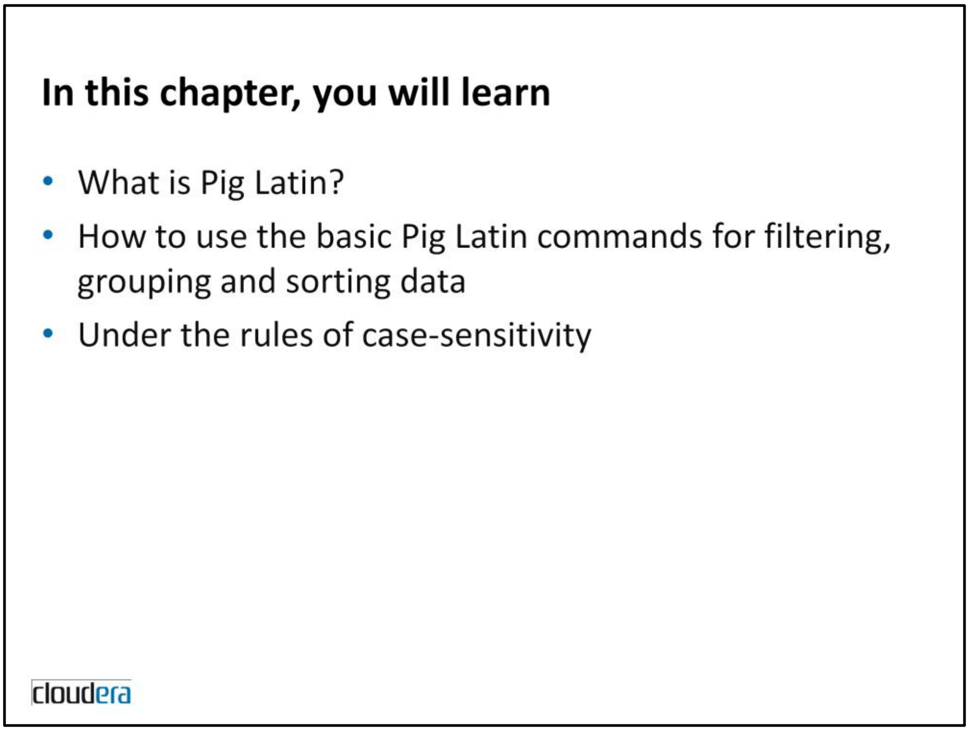
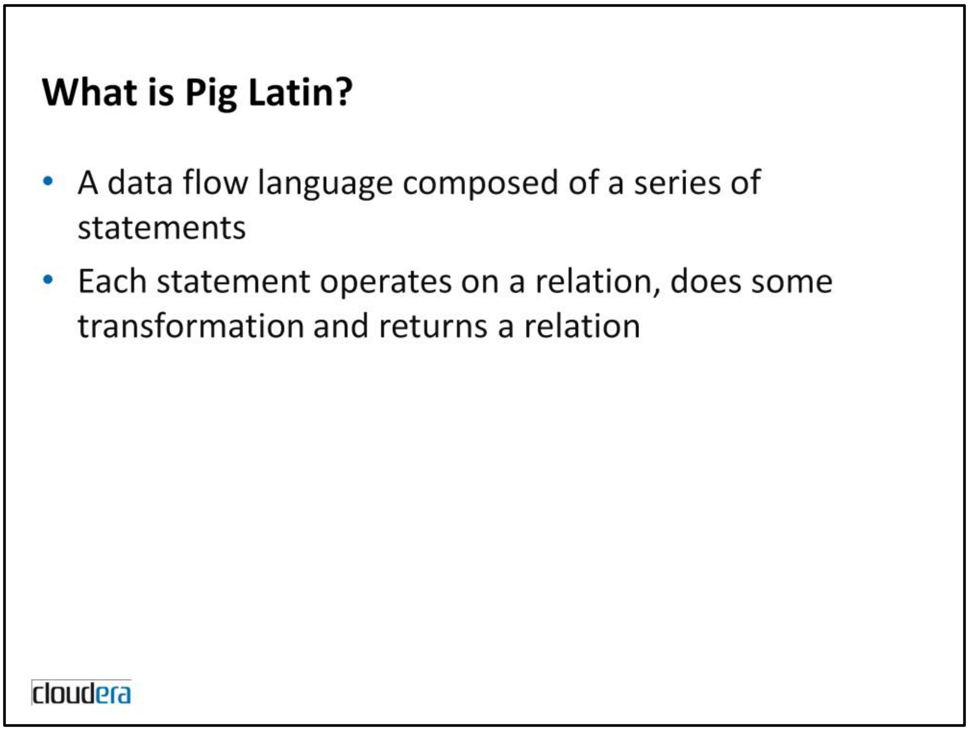


Pig Latin basics



In this chapter, you will learn

- What is Pig Latin?
- How to use the basic Pig Latin commands for filtering, grouping and sorting data
- Under the rules of case-sensitivity



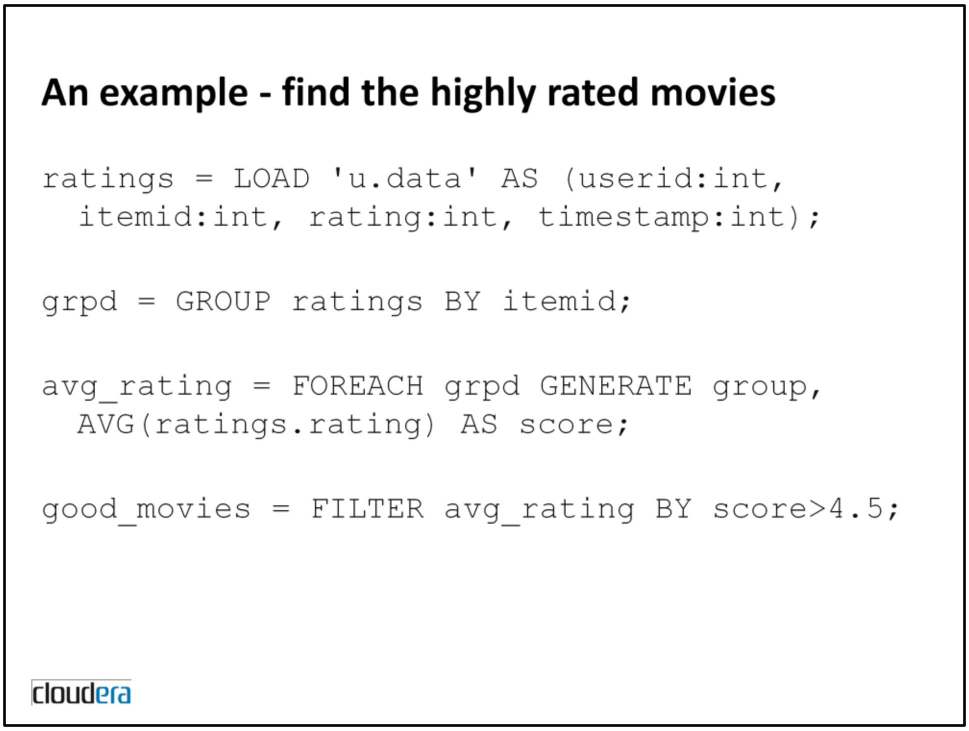
What is Pig Latin?

- A data flow language composed of a series of statements
- Each statement operates on a relation, does some transformation and returns a relation

Pig Latin is a scripting language used to express data flows as a series of statements (also called operations or transformations). Pig will convert the statements into a series of MapReduce jobs. A Pig program can accomplish rather sophisticated data processing with only a few lines of code as opposed to writing 100s of lines of code in Java MapReduce.

Each Pig Latin statement takes a relation as input (except the LOAD statement, which creates the initial relation). The statement does some operation and returns a new relation as output. It is common to give each resulting relation a new alias.

Most statements in Pig Latin end in a semicolon (with the exception of some statements like the Hadoop filesystem commands).



An example - find the highly rated movies

```
ratings = LOAD 'u.data' AS (userid:int,
  itemid:int, rating:int, timestamp:int);

grpd = GROUP ratings BY itemid;

avg_rating = FOREACH grpd GENERATE group,
  AVG(ratings.rating) AS score;

good_movies = FILTER avg_rating BY score>4.5;
```

The u.data file contains 100000 ratings by 943 users on 1682 movies. Each user has rated at least 20 movies. The file contains a tab-separated list of : user id, item id, rating and timestamp.

The following script uses the u.data file to find the movies that received an average rating over 4.5:

-- Load the u.data file which contains 4 integers, tab-separated. Default PigStorage loader is used. "ratings" is the alias, or name, given to the result

```
ratings = LOAD 'u.data' AS (userid:int, itemid:int,
rating:int, timestamp:int);
```

-- Group the movies in the relation "ratings" by their itemid

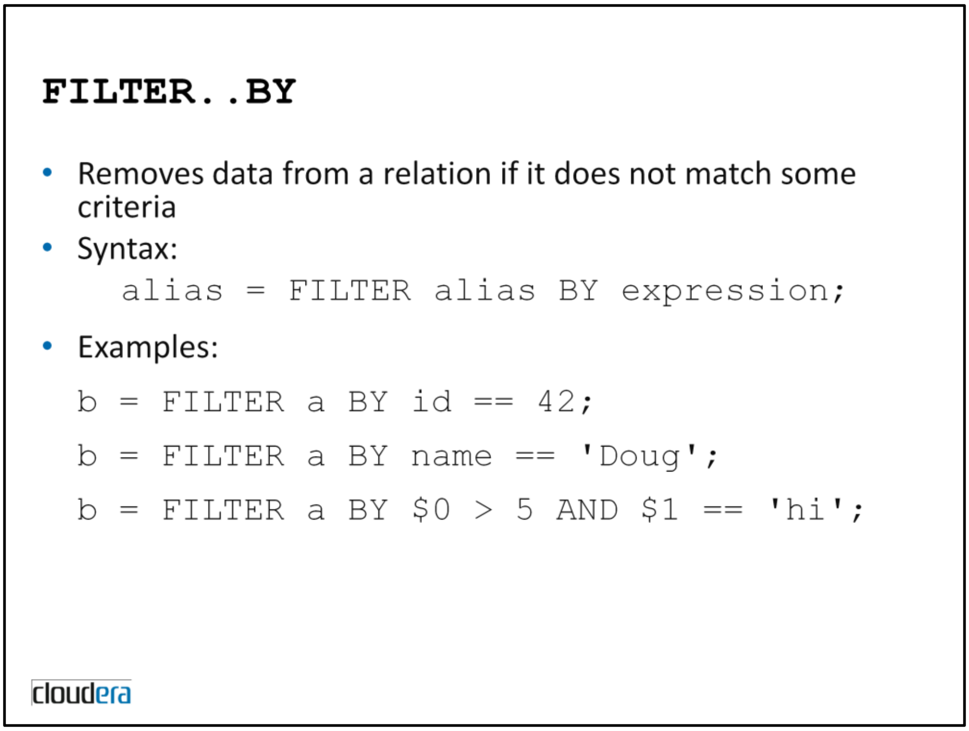
```
grpd = GROUP ratings BY itemid;
```

-- Calculate an average score per group (i.e., per movie). Note: lower case "group" is a keyword that we'll talk about later.

```
avg_rating = FOREACH grpd GENERATE group,
AVG(ratings.rating) AS score;
```

-- Remove all movies that did not score better than 4.5

```
good_movies = FILTER avg_rating BY score > 4.5;
```



FILTER..BY

- Removes data from a relation if it does not match some criteria
- Syntax:

```
alias = FILTER alias BY expression;
```

- Examples:

```
b = FILTER a BY id == 42;
b = FILTER a BY name == 'Doug';
b = FILTER a BY $0 > 5 AND $1 == 'hi';
```

The `FILTER` operator will remove (or filter) records from a relation if they do not match some criteria. This could be a basic expression such as "does the id field equal 42?" or a compound expression that looks at multiple fields. The general syntax is:

```
alias2 = FILTER alias1 BY expression;
```

alias1 is the incoming relation and alias2 is the resulting (probably smaller) relation.

Here are several more examples:
-- Find all records in `a` that have value 42 for their id field:

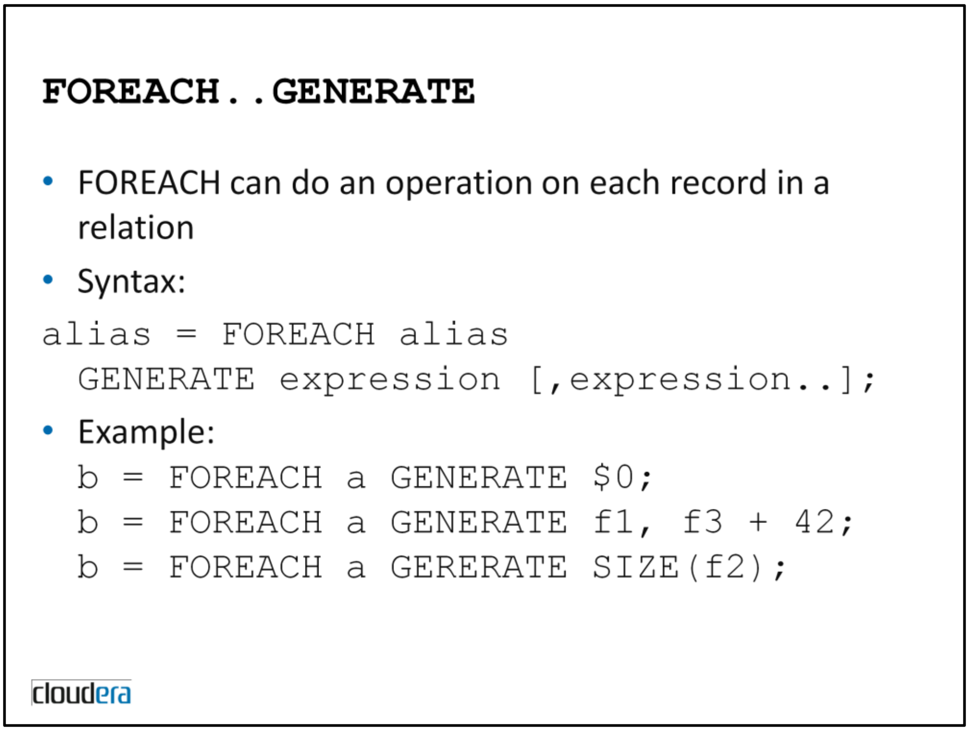
```
b = FILTER a BY id == 42;
```

-- Find all records in `a` that have the exact string 'Doug' in their name:

```
b = FILTER a BY name == 'Doug';
```

-- Find all records in `a` that have the value 5 for their first field and string 'hi' for their second field:

```
b = FILTER a BY $0 > 5 AND $1 == 'hi';
```



FOREACH..GENERATE

- FOREACH can do an operation on each record in a relation
- Syntax:

```
alias = FOREACH alias
  GENERATE expression [,expression..];
```

- Example:

```
  b = FOREACH a GENERATE $0;
  b = FOREACH a GENERATE f1, f3 + 42;
  b = FOREACH a GERERATE SIZE(f2);
```

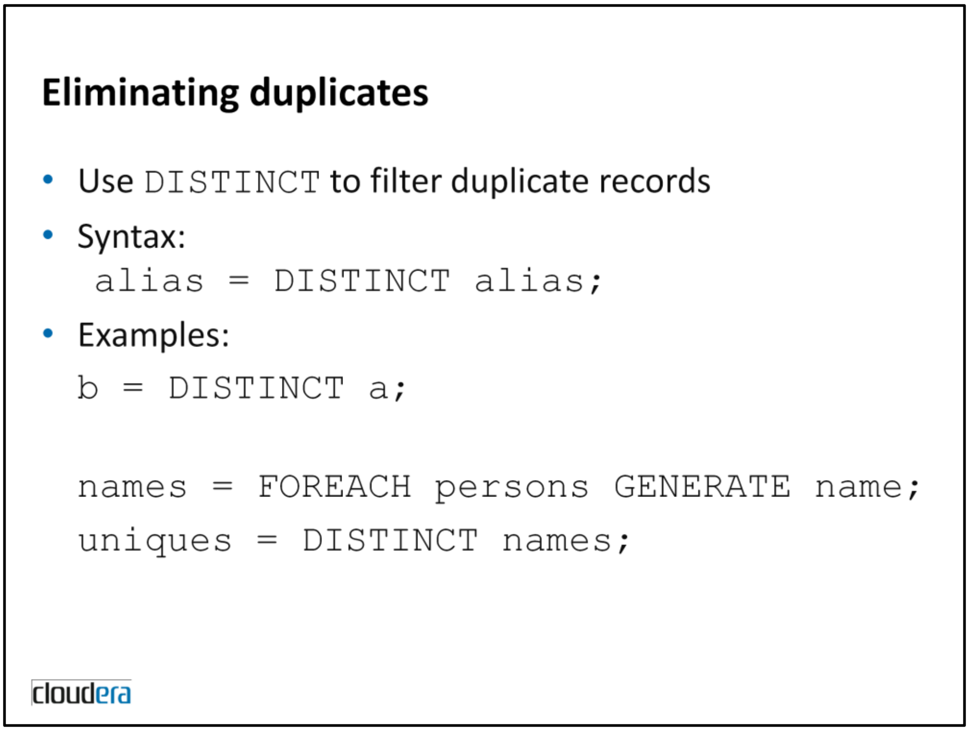
FOREACH can apply an operation on each record of a relation. In other words, it iterates through the records. Some examples:

```
-- For each record in a, output just the first field ($0):
b = FOREACH a GENERATE $0;

-- For each record in a, output the field named f1 and the field f3 plus 42.
b = FOREACH a GENERATE f1, f3 + 42;
```

Some uses of FOREACH:

- Remove or reorder some columns
- Apply operators (+, -, *, /, etc) to the data
- Apply functions to the data (covered later)
- Iterate through a grouped relation (covered later)



Eliminating duplicates

- Use `DISTINCT` to filter duplicate records
- Syntax:

```
 alias = DISTINCT alias;
```

- Examples:

```
b = DISTINCT a;

names = FOREACH persons GENERATE name;
uniques = DISTINCT names;
```

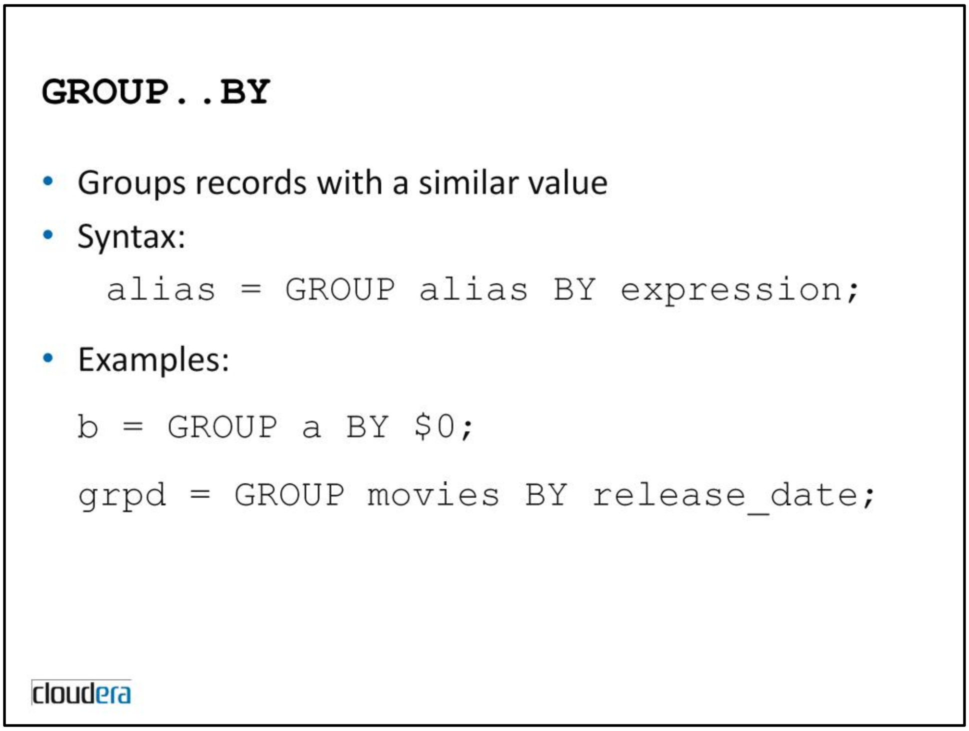
The `DISTINCT` command eliminates duplicate records in a relation. This applies to the whole record - in order to be considered a duplicate, all fields must be equal.

For example, let's take this data:

Doug	Cutting
Doug	Jones

Loading the data and applying `DISTINCT` will not remove any rows because the records have different last names. However, `FOREACH..GENERATE` can be used to create a new relation with only certain fields and then `DISTINCT` can be applied:

```
persons = LOAD 'data' AS (fname:chararray,
lname:chararray);
names = FOREACH persons GENERATE fname;
uniques = DISTINCT fname;
```



GROUP..BY

- Groups records with a similar value
- Syntax:

```
alias = GROUP alias BY expression;
```

- Examples:

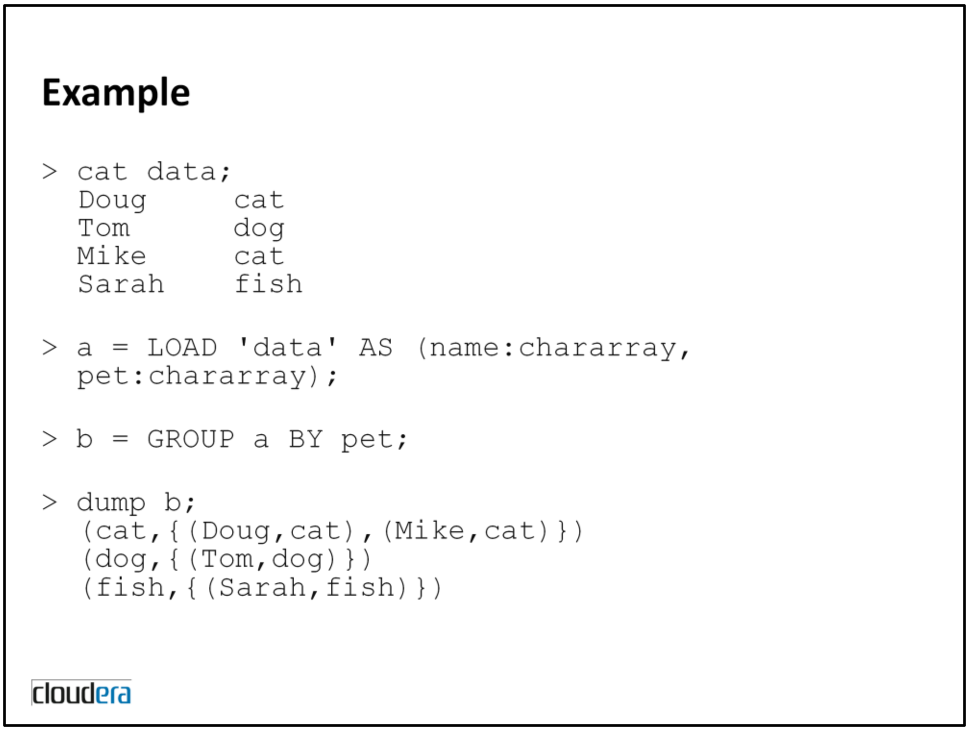
```
b = GROUP a BY $0;

grpd = GROUP movies BY release_date;
```

The GROUP keyword is used to group records in a relation into groups. Items in a group have a value in common (the group key). The basic syntax is:

```
alias2 = GROUP alias1 BY expression;
```

alias1 is the incoming relation. expression is the criterion used to identify the groups. alias2 is the resulting relation which contains one tuple per group. Each tuple looks like:
(group-key, bag-of-tuples-in-group)



Example

```
> cat data;
  Doug     cat
  Tom      dog
  Mike     cat
  Sarah    fish

> a = LOAD 'data' AS (name:chararray,
  pet:chararray);

> b = GROUP a BY pet;

> dump b;
   (cat,{(Doug,cat),(Mike,cat)})
   (dog,{(Tom,dog)})
   (fish,{(Sarah,fish)})
```

Let's say the file had peoples' names and their favorite type of pet:

```
Doug      cat
Tom       dog
Mike      cat
Sarah     fish
```

We could load the data like this:

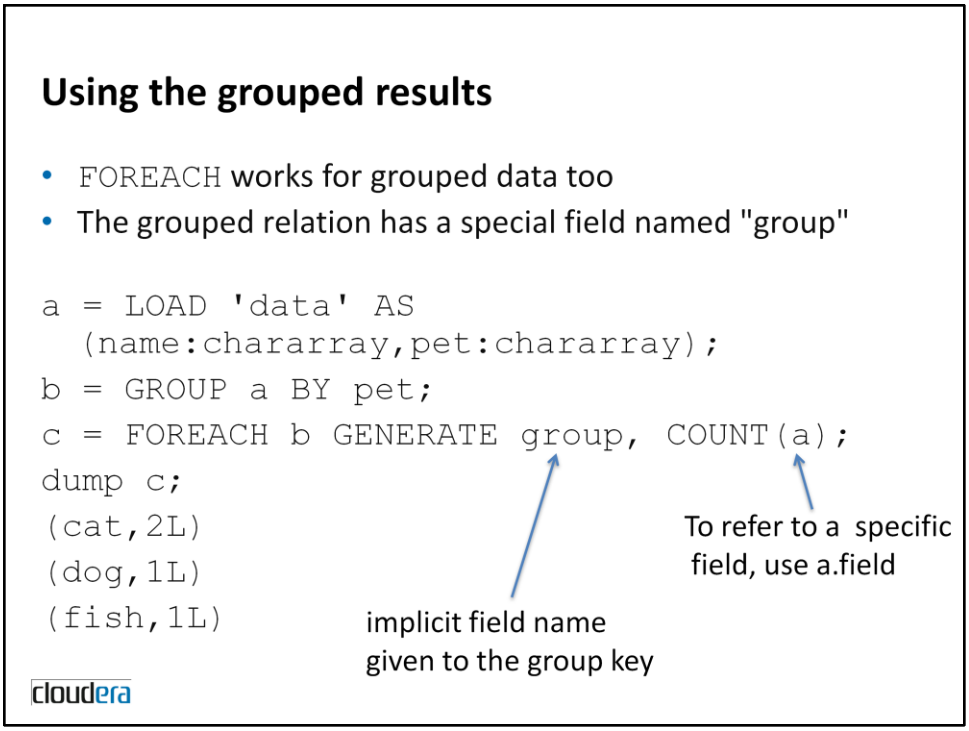
```
a = LOAD 'data' AS (name:chararray, pet:chararray);
```

Then group by the type of pet:

```
b = GROUP a BY pet;
```

The result would be 3 groups (cat, dog and fish):

```
(cat,{(Doug,cat),(Mike,cat)})
(dog,{(Tom,dog)})
(fish,{(Sarah,fish)})
```



Using the grouped results

- FOREACH works for grouped data too
- The grouped relation has a special field named "group"

```
a = LOAD 'data' AS
  (name:chararray,pet:chararray);
b = GROUP a BY pet;
c = FOREACH b GENERATE group, COUNT(a);
dump c;
(cat,2L)
(dog,1L)
(fish,1L)
```

implicit field name given to the group key

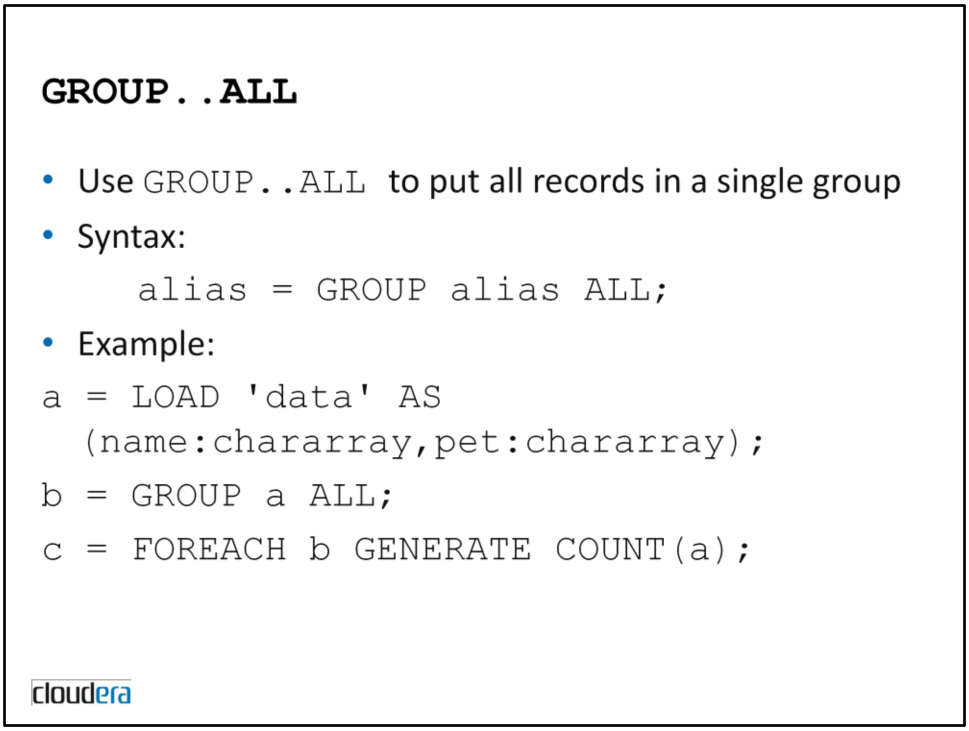
To refer to a specific field, use a.field

Earlier we saw that FOREACH iterates through the records in a relation. This is useful for grouped results. Example:

```
a = LOAD 'data' AS (name:chararray, pet:chararray);
b = GROUP a BY pet;
-- For each record in b, output the group (pet) and the
number of records (a).  An alias can be given to the
result of COUNT using AS
c = FOREACH b GENERATE group, COUNT(a) AS num;
DUMP c;
(cat,2L)
(dog,1L)
(fish,1L)
```

Relation b has a field named "group" which refers to the group key (the pet).

It is common to use aggregate functions, such as COUNT() on groups (covered later).



GROUP..ALL

- Use GROUP..ALL to put all records in a single group
- Syntax:

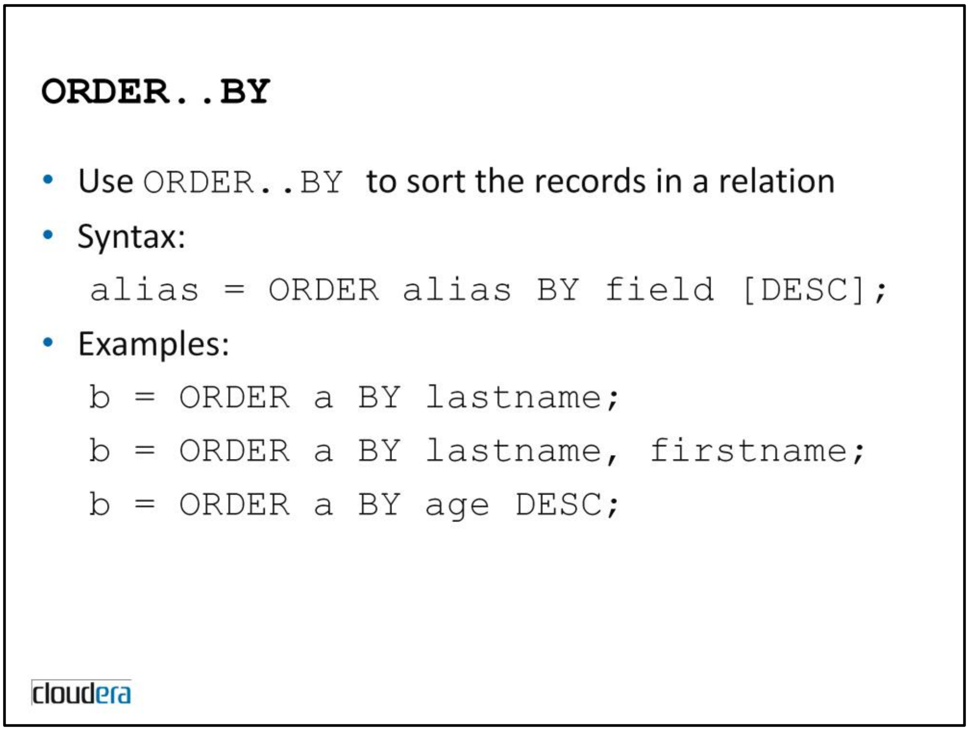
```
alias = GROUP alias ALL;
```

- Example:

```
a = LOAD 'data' AS
  (name:chararray,pet:chararray);
b = GROUP a ALL;
c = FOREACH b GENERATE COUNT(a);
```

Sometimes all the records should be collected into a single group. This is usually done to calculate aggregate functions on ALL records. For example, we can count the number of records in the file using:

```
a = LOAD 'data' AS (name:chararray, pet:chararray);
b = GROUP a ALL;
c = FOREACH b GENERATE COUNT(a);
```



ORDER..BY

- Use `ORDER..BY` to sort the records in a relation
- Syntax:

```
alias = ORDER alias BY field [DESC];
```

- Examples:

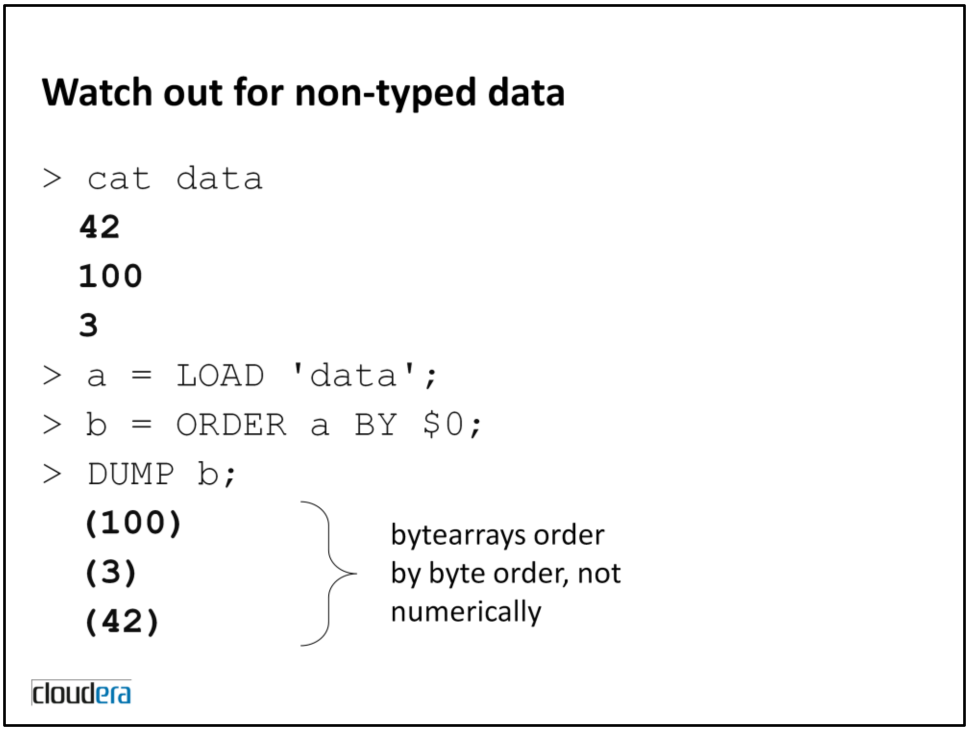
```
b = ORDER a BY lastname;
b = ORDER a BY lastname, firstname;
b = ORDER a BY age DESC;
```

To sort records in a relation by a field or fields, use `ORDER..BY`. By default the sort order is **ascending**. Adding the keyword **DESC** will sort them descending.

Multiple fields can be specified, such as `ORDER..BY lastname, firstname`. This will sort by lastname, with a secondary sort by firstname (like the white pages in a phone book). Each field may choose ascending or descending order.

Pig can take advantage of multiple reducers (see PARALLEL keyword later). A special Partitioner is used to accomplish this.

Note: In the current version of Pig, the order by field must be a column and cannot contain expressions. This will change in a later version (Pig 0.9).



Watch out for non-typed data

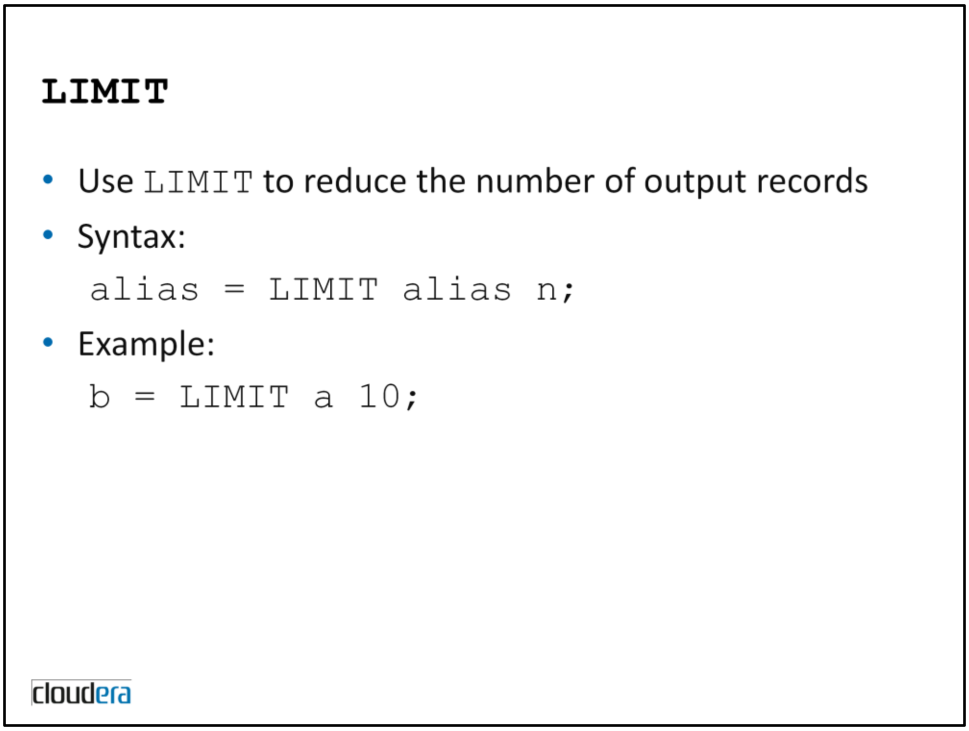
```
> cat data
  42
  100
  3
> a = LOAD 'data';
> b = ORDER a BY $0;
> DUMP b;
  (100)
  (3)
  (42)
```

bytearrays order by byte order, not numerically

Be careful when sorting data with an unspecified schema. If a schema was not given in the `LOAD`, the data is considered type bytearray. Sorting by byte order is not the same as sorting numerically. If you want to sort integers properly, make sure they are typed as integer.

Correct approach:

```
a = LOAD 'data' AS f1:int;
b = ORDER a BY f1;
DUMP b;
 (3)
 (42)
 (100)
```



LIMIT

- Use LIMIT to reduce the number of output records
- Syntax:

```
alias = LIMIT alias n;
```

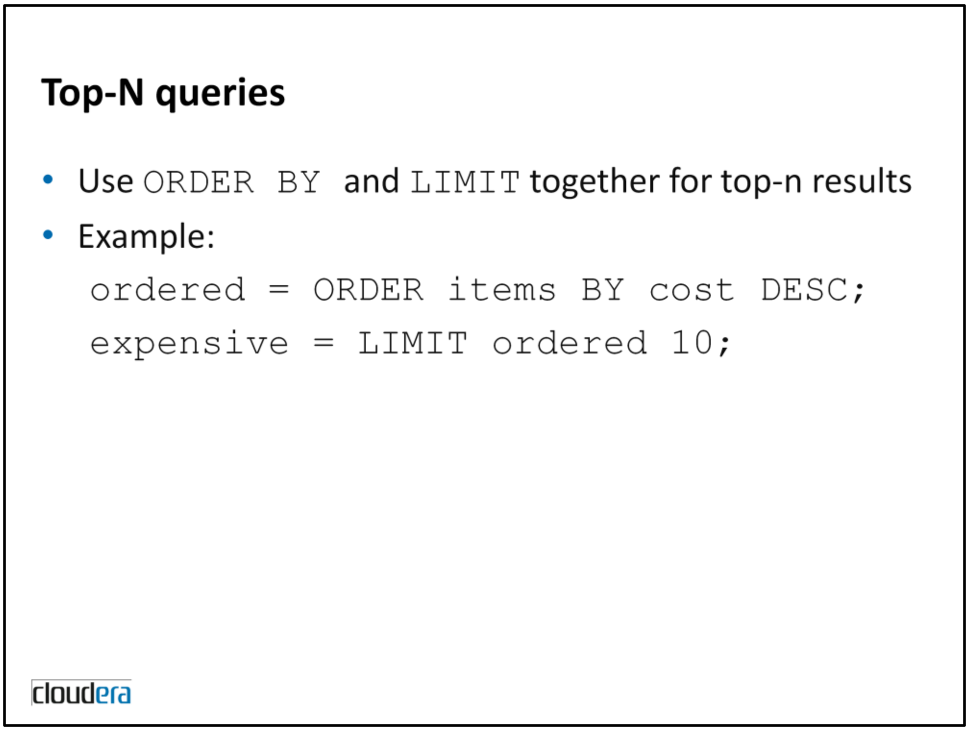
- Example:

```
b = LIMIT a 10;
```

cloudera

The LIMIT keyword reduces the amount of records in a relation. This can be very useful and also makes Pig scripts more efficient by eliminating the amount of data that needs to be processed.

Unless an ORDER BY is also specified, the records returned from a LIMIT are somewhat random and may change from one execution to another.



Top-N queries

- Use ORDER BY and LIMIT together for top-n results
- Example:

```
ordered = ORDER items BY cost DESC;
expensive = LIMIT ordered 10;
```

cloudera

A "Top-N" query is looking for a certain number (N) of the top (i.e., greatest or least) items. For example, the *10 most expensive items* or the *100 worst movies of all time*.

This requires 2 steps:

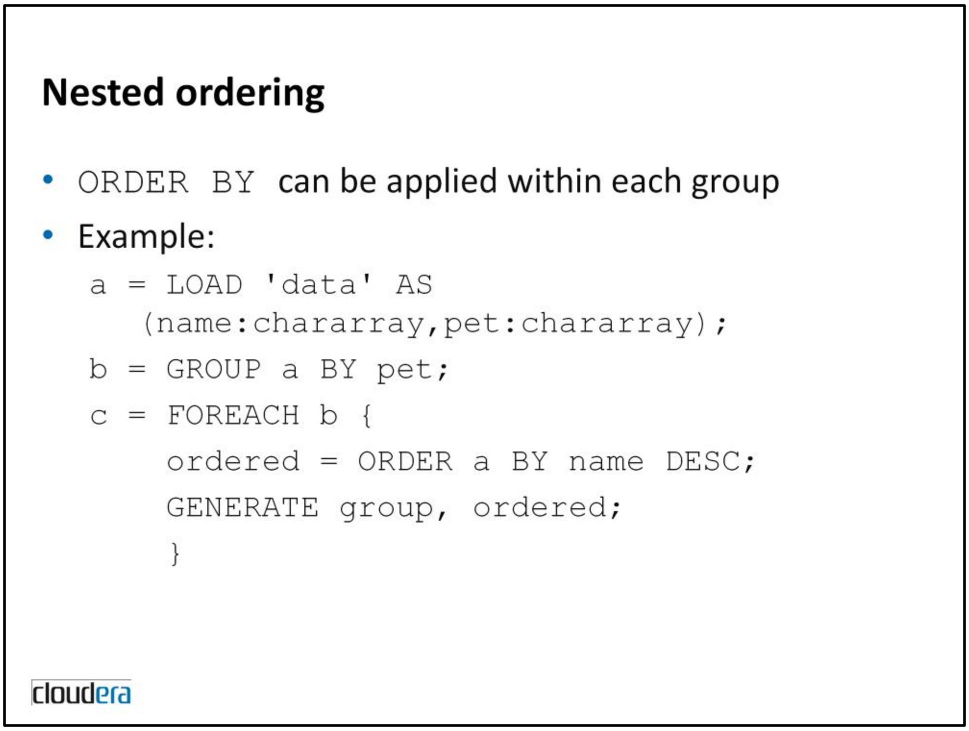
1. Order the results by the criterion that makes sense
2. Use LIMIT to pull off the number of results you want to return

For example, the 10 least expensive items:

```
ordered = ORDER items BY cost;
cheap = LIMIT ordered 10;
```

To ask for the 10 *most* expense, sort descending instead of ascending:

```
ordered = ORDER items BY cost DESC;
expensive = LIMIT ordered 10;
```



Nested ordering

- `ORDER BY` can be applied within each group
- Example:

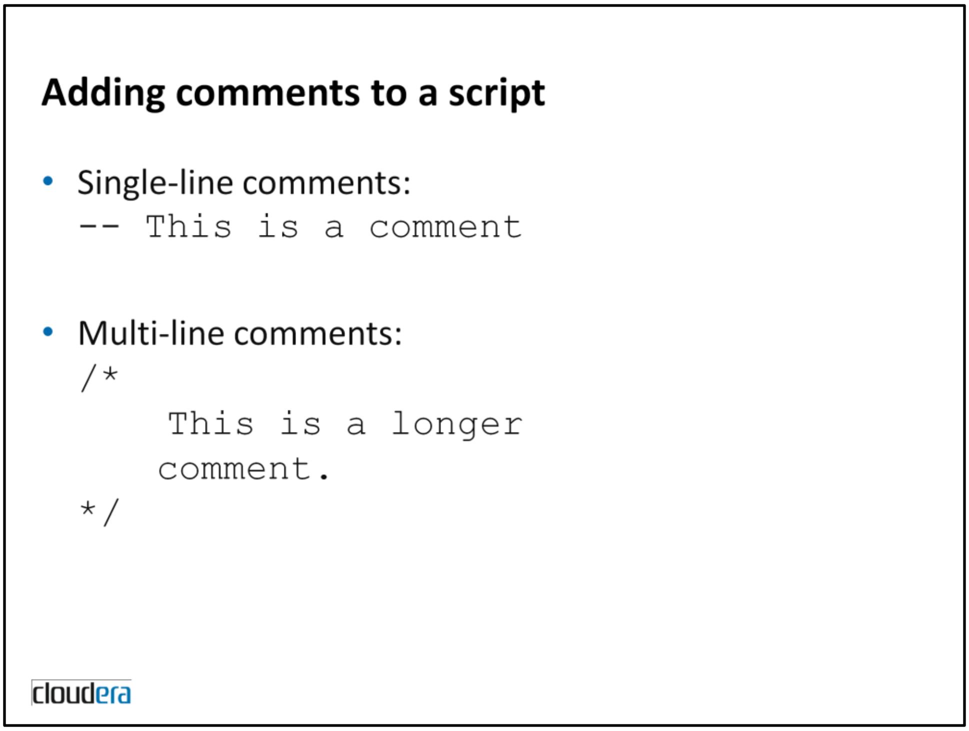
```
a = LOAD 'data' AS
   (name:chararray,pet:chararray);
b = GROUP a BY pet;
c = FOREACH b {
    ordered = ORDER a BY name DESC;
    GENERATE group, ordered;
    }
```

Let's say the file had peoples' names and their favorite type of pet:

```
Doug      cat
Tom       dog
Mike      cat
Sarah     fish
```

```
a = LOAD 'data' AS (name:chararray, pet:chararray);
b = GROUP a BY pet;
-- For each group (cat, dog and fish), order the owners' names alphabetically
c = FOREACH b {
    ordered = ORDER a BY name;
    GENERATE group, ordered;
    }
```

`LIMIT` can also be used in the `FOREACH` block.



Adding comments to a script

- Single-line comments:

```
-- This is a comment
```

- Multi-line comments:

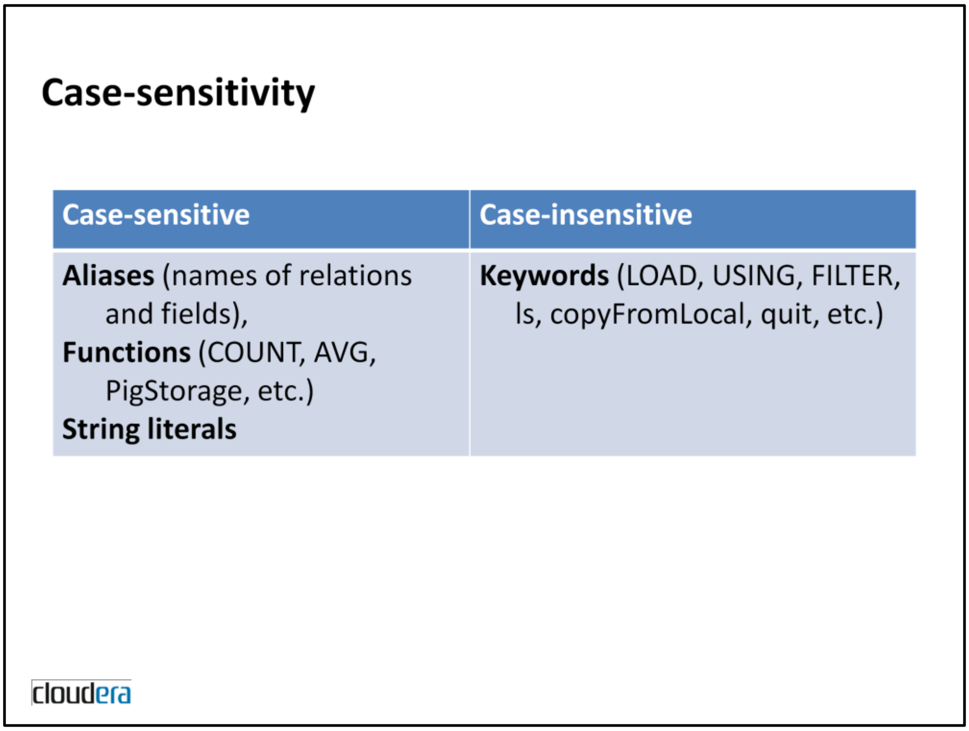
```
/*
    This is a longer
   comment.
*/
```

Commenting code is a good way to add readability to your Pig scripts. There are two types of comments. Double hyphens are a single-line comment: from the hyphens, the rest of the line is ignored. Examples:

```
-- Load the data:
a = LOAD…
-- Find the records that match pattern X:
b = FILTER..
DUMP c; -- Print the results to the screen
```

C-style comments are also supported. They allow multiple lines to be ignored by the Pig parser:

```
/*
    This is a longer
    comment.
*/
```



Case-sensitivity

Case-sensitive	Case-insensitive
Aliases (names of relations and fields), **Functions** (COUNT, AVG, PigStorage, etc.) **String literals**	**Keywords** (LOAD, USING, FILTER, ls, copyFromLocal, quit, etc.)

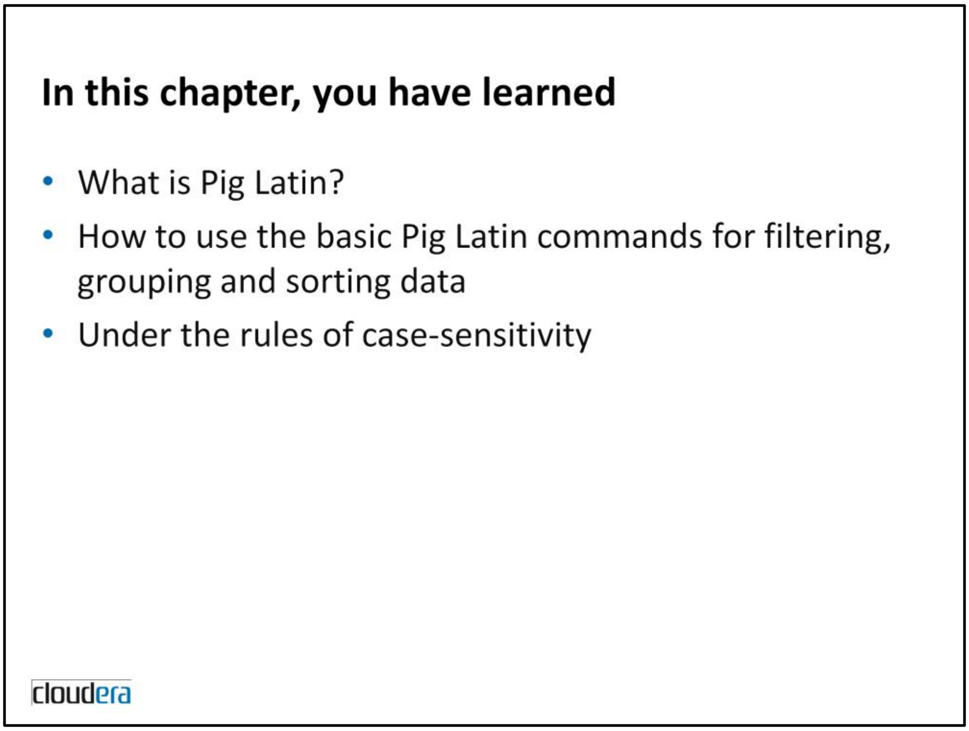
Pig Latin has mixed rules on case-sensitivity. Aliases and functions are case-sensitive, yet keywords and operators are not. Take this example:

```
ratings = LOAD 'u.data' AS (userid:int, itemid:int,
rating:int, timestamp:int);
grpd = GROUP ratings BY itemid;
avg_rating = FOREACH grpd GENERATE group,
AVG(ratings.rating) AS score;
good_movies = FILTER avg_rating BY score > 4.5;
```

In the above script, the aliases and the AVG function **are** case-senstivive. String comparisons such as `name = 'doug'` are also case-sensitive.

However, these are **not** case-sensitive:

```
LOAD, AS, int, GROUP, BY, FOREACH, GENERATE, FILTER
```



In this chapter, you have learned

- What is Pig Latin?
- How to use the basic Pig Latin commands for filtering, grouping and sorting data
- Under the rules of case-sensitivity