PIG

Partial Schema Specification

₪ Pig Is Omnivorous

- δ It consumes any kind of data
- δ If schema is not known, it will still accept data, guessing along the way.
  - ∞ If an arithmetic operation has been performed on a data type, Pig will assume that the data is Integer or Double.

Pig and Schema Definitions

₪ No schema definition

- δ Pig will assume every field to be of type bytearray

₪ Partial schema definition

- δ Can leave out data types for fields

₪ Complete schema definition

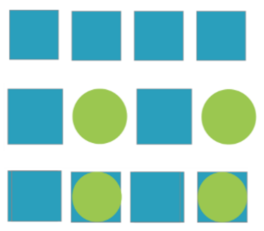
- δ All fields and data types known

Casting and Conversion

₪ Schema defined for the relation

₪ Schema found by the Pig loader

₪ Pig will try and convert the data it found by casting it to the specified schema

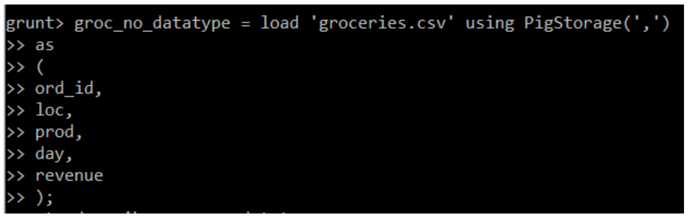


₪ Not all conversions are permitted

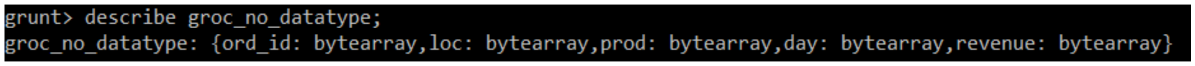
from / to	bag	tuple	map	int	long	float	double	chararray	bytearray	boolean
bag		error	error	error	error	error	error	error	error	error
tuple	error		error	error	error	error	error	error	error	error
map	error	error		error	error	error	error	error	error	error
int	error	error	error		yes	yes	yes	yes	error	error
long	error	error	error	yes		yes	yes	yes	error	error
float	error	error	error	yes	yes		yes	yes	error	error
double	error	error	error	yes	yes	yes		yes	error	error
chararray	error	error	error	yes	yes	yes	yes		error	yes
bytearray	yes	yes	yes	yes	yes	yes	yes	yes		yes
boolean	error	error	error	error	error	error	error	yes	error	

DEMO

- Use groceries.csv
- Specify the names of the fields and not the data type.

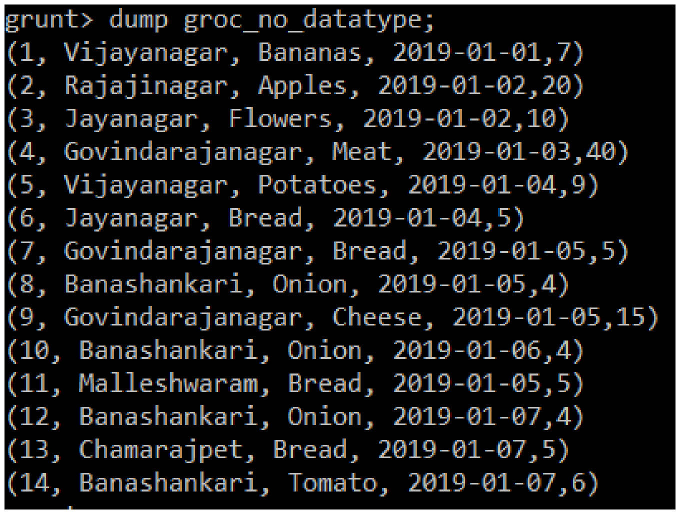


```
grunt> groc_no_datatype = load 'groceries.csv' using PigStorage(',')
>> as
>> (
>> ord_id,
>> loc,
>> prod,
>> day,
>> revenue
>> );
```



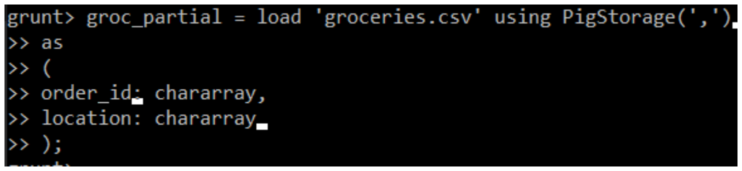
```
grunt> describe groc_no_datatype;
groc_no_datatype: {ord_id: bytearray,loc: bytearray,prod: bytearray,day: bytearray,revenue: bytearray}
```

- Every field is of the type bytearray.

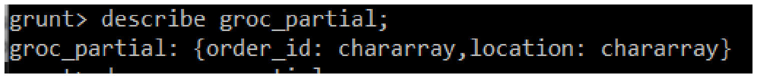


```
grunt> dump groc_no_datatype;
(1, Vijayanagar, Bananas, 2019-01-01,7)
(2, Rajajinagar, Apples, 2019-01-02,20)
(3, Jayanagar, Flowers, 2019-01-02,10)
(4, Govindarajanagar, Meat, 2019-01-03,40)
(5, Vijayanagar, Potatoes, 2019-01-04,9)
(6, Jayanagar, Bread, 2019-01-04,5)
(7, Govindarajanagar, Bread, 2019-01-05,5)
(8, Banashankari, Onion, 2019-01-05,4)
(9, Govindarajanagar, Cheese, 2019-01-05,15)
(10, Banashankari, Onion, 2019-01-06,4)
(11, Malleshwaram, Bread, 2019-01-05,5)
(12, Banashankari, Onion, 2019-01-07,4)
(13, Chamarajpet, Bread, 2019-01-07,5)
(14, Banashankari, Tomato, 2019-01-07,6)
```

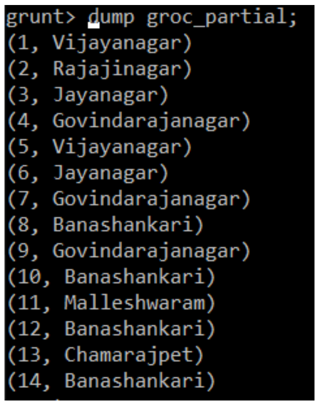
- It can be loaded with partial schema definition too.



```
grunt> groc_partial = load 'groceries.csv' using PigStorage(',')
>> as
>> (
>> order_id: chararray,
>> location: chararray
>> );
```



```
grunt> describe groc_partial;
groc_partial: {order_id: chararray,location: chararray}
```



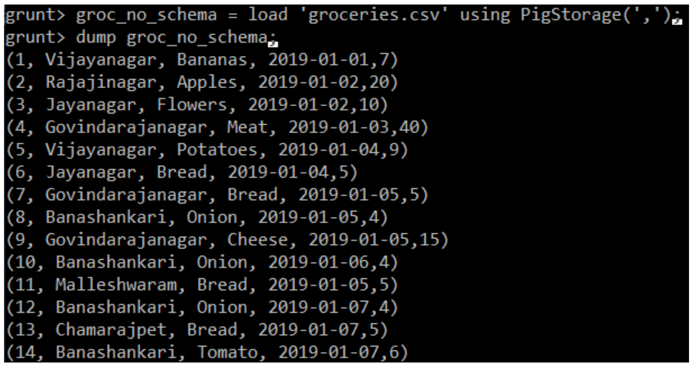
```
grunt> dump groc_partial;
(1, Vijayanagar)
(2, Rajajinagar)
(3, Jayanagar)
(4, Govindarajanagar)
(5, Vijayanagar)
(6, Jayanagar)
(7, Govindarajanagar)
(8, Banashankari)
(9, Govindarajanagar)
(10, Banashankari)
(11, Malleshwaram)
(12, Banashankari)
(13, Chamarajpet)
(14, Banashankari)
```

Foreach and generate

- Foreach iterates through every tuple in a relation (or inner bag) and projects the fields that the user is interested in.
- The projected fields can also be part of expression.
- The result of a foreach generate statements is a relation that is stored in a variable.

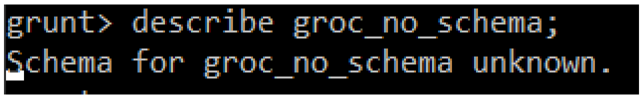
DEMO

- Use groceries.csv and load it without specifying any schema.



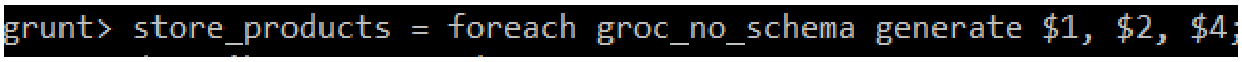
```
grunt> groc_no_schema = load 'groceries.csv' using PigStorage(',');
grunt> dump groc_no_schema;
(1, Vijayanagar, Bananas, 2019-01-01,7)
(2, Rajajinagar, Apples, 2019-01-02,20)
(3, Jayanagar, Flowers, 2019-01-02,10)
(4, Govindarajanagar, Meat, 2019-01-03,40)
(5, Vijayanagar, Potatoes, 2019-01-04,9)
(6, Jayanagar, Bread, 2019-01-04,5)
(7, Govindarajanagar, Bread, 2019-01-05,5)
(8, Banashankari, Onion, 2019-01-05,4)
(9, Govindarajanagar, Cheese, 2019-01-05,15)
(10, Banashankari, Onion, 2019-01-06,4)
(11, Malleshwaram, Bread, 2019-01-05,5)
(12, Banashankari, Onion, 2019-01-07,4)
(13, Chamarajpet, Bread, 2019-01-07,5)
(14, Banashankari, Tomato, 2019-01-07,6)
```

- To access individual fields the indices, have to be accessed.
  - The indices start at 0.



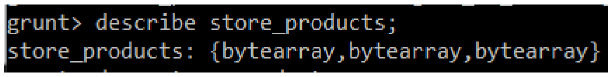
```
grunt> describe groc_no_schema;
Schema for groc_no_schema unknown.
```

  - To access individual fields



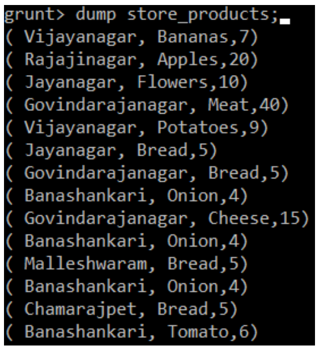
```
grunt> store_products = foreach groc_no_schema generate $1, $2, $4;
```

  - Dump command shows the contents.
  - Describe command will describe the schema



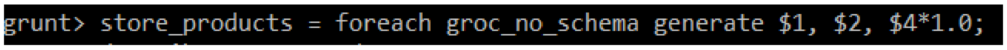
```
grunt> describe store_products;
store_products: {bytearray,bytearray,bytearray}
```

  - Each of these fields are bytearrays but have no name.

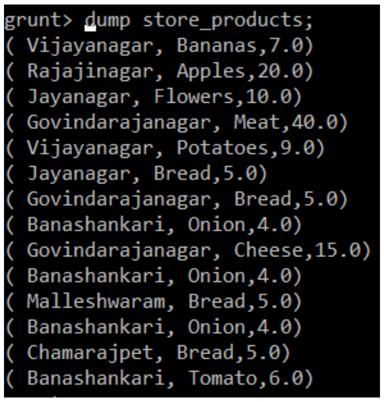


```
grunt> dump store_products;
( Vijayanagar, Bananas,7)
( Rajajinagar, Apples,20)
( Jayanagar, Flowers,10)
( Govindarajanagar, Meat,40)
( Vijayanagar, Potatoes,9)
( Jayanagar, Bread,5)
( Govindarajanagar, Bread,5)
( Banashankari, Onion,4)
( Govindarajanagar, Cheese,15)
( Banashankari, Onion,4)
( Malleshwaram, Bread,5)
( Banashankari, Onion,4)
( Chamarajpet, Bread,5)
( Banashankari, Tomato,6)
```

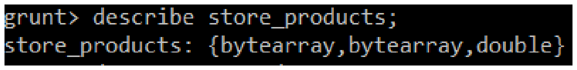
  - Pig trying to understand the data type based on the operation.



```
grunt> store_products = foreach groc_no_schema generate $1, $2, $4*1.0;
```

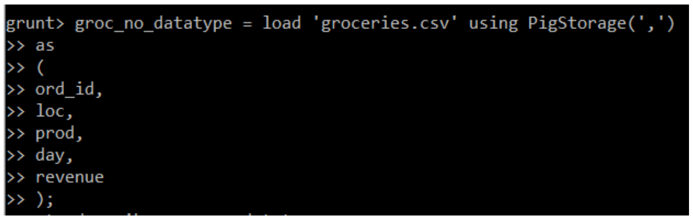


```
grunt> dump store_products;
( Vijayanagar, Bananas,7.0)
( Rajajinagar, Apples,20.0)
( Jayanagar, Flowers,10.0)
( Govindarajanagar, Meat,40.0)
( Vijayanagar, Potatoes,9.0)
( Jayanagar, Bread,5.0)
( Govindarajanagar, Bread,5.0)
( Banashankari, Onion,4.0)
( Govindarajanagar, Cheese,15.0)
( Banashankari, Onion,4.0)
( Malleshwaram, Bread,5.0)
( Banashankari, Onion,4.0)
( Chamarajpet, Bread,5.0)
( Banashankari, Tomato,6.0)
```

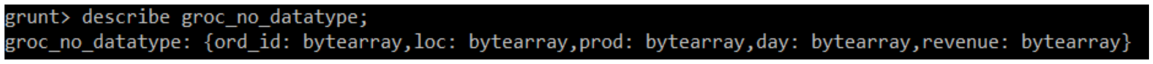


```
grunt> describe store_products;
store_products: {bytearray,bytearray,double}
```

- Use groceries.csv
- Specify the names of the fields and not the data type.

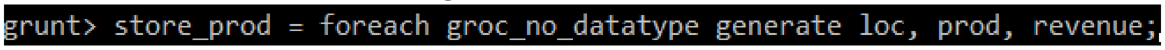


```
grunt> groc_no_datatype = load 'groceries.csv' using PigStorage(',')
>> as
>> (
>> ord_id,
>> loc,
>> prod,
>> day,
>> revenue
>> );
```

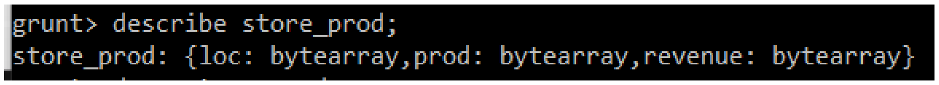


```
grunt> describe groc_no_datatype;
groc_no_datatype: {ord_id: bytearray,loc: bytearray,prod: bytearray,day: bytearray,revenue: bytearray}
```

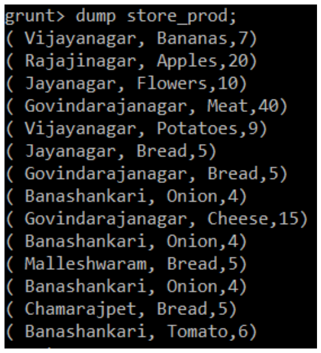
- Use the name of the fields to generate records.



```
grunt> store_prod = foreach groc_no_datatype generate loc, prod, revenue;
```

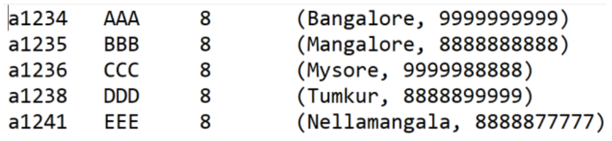


```
grunt> describe store_prod;
store_prod: {loc: bytearray,prod: bytearray,revenue: bytearray}
```

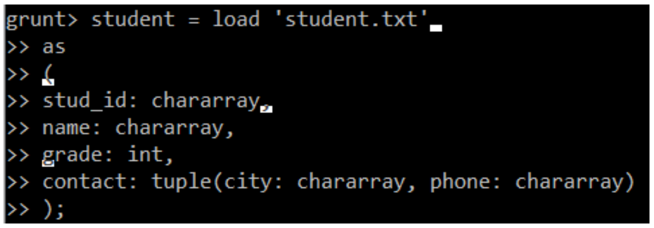


```
grunt> dump store_prod;
( Vijayanagar, Bananas,7)
( Rajajinagar, Apples,20)
( Jayanagar, Flowers,10)
( Govindarajanagar, Meat,40)
( Vijayanagar, Potatoes,9)
( Jayanagar, Bread,5)
( Govindarajanagar, Bread,5)
( Banashankari, Onion,4)
( Govindarajanagar, Cheese,15)
( Banashankari, Onion,4)
( Malleshwaram, Bread,5)
( Banashankari, Onion,4)
( Chamarajpet, Bread,5)
( Banashankari, Tomato,6)
```

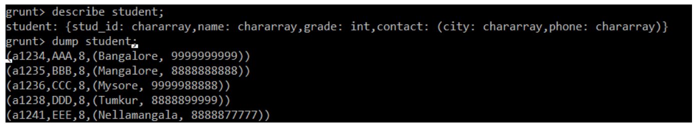
- Use student.txt.



```
a1234   AAA   8     (Bangalore, 9999999999)
a1235   BBB   8     (Mangalore, 8888888888)
a1236   CCC   8     (Mysore, 9999988888)
a1238   DDD   8     (Tumkur, 8888899999)
a1241   EEE   8     (Nellamangala, 8888877777)
```

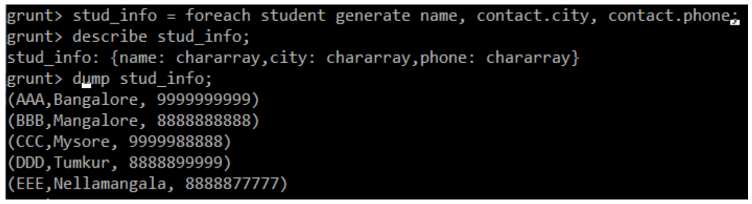


```
grunt> student = load 'student.txt'
>> as
>> (
>> stud_id: chararray,
>> name: chararray,
>> grade: int,
>> contact: tuple(city: chararray, phone: chararray)
>> );
```



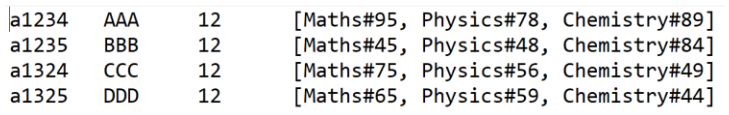
```
grunt> describe student;
student: {stud_id: chararray,name: chararray,grade: int,contact: (city: chararray,phone: chararray)}
grunt> dump student;
(a1234,AAA,8,(Bangalore, 9999999999))
(a1235,BBB,8,(Mangalore, 8888888888))
(a1236,CCC,8,(Mysore, 9999988888))
(a1238,DDD,8,(Tumkur, 8888899999))
(a1241,EEE,8,(Nellamangala, 8888877777))
```

- Use foreach
- The '.' (dot) operator is used to access individual fields in tuples

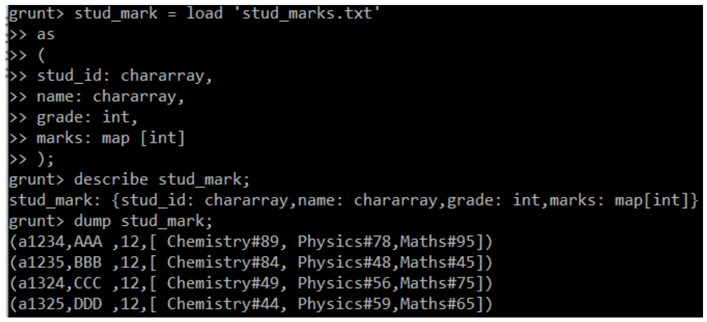


```
grunt> stud_info = foreach student generate name, contact.city, contact.phone;
grunt> describe stud_info;
stud_info: {name: chararray,city: chararray,phone: chararray}
grunt> dump stud_info;
(AAA,Bangalore, 9999999999)
(BBB,Mangalore, 8888888888)
(CCC,Mysore, 9999988888)
(DDD,Tumkur, 8888899999)
(EEE,Nellamangala, 8888877777)
```

- Use stud_marks.txt

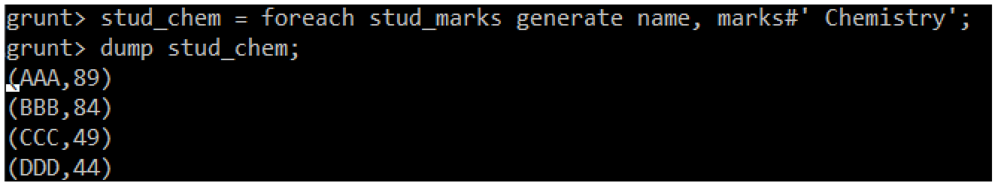


```
a1234    AAA    12    [Maths#95, Physics#78, Chemistry#89]
a1235    BBB    12    [Maths#45, Physics#48, Chemistry#84]
a1324    CCC    12    [Maths#75, Physics#56, Chemistry#49]
a1325    DDD    12    [Maths#65, Physics#59, Chemistry#44]
```



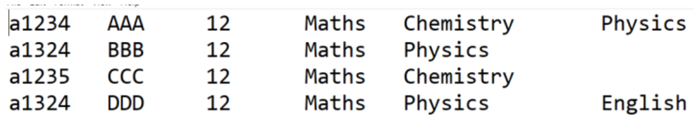
```
grunt> stud_mark = load 'stud_marks.txt'
>> as
>> (
>> stud_id: chararray,
>> name: chararray,
>> grade: int,
>> marks: map [int]
>> );
grunt> describe stud_mark;
stud_mark: {stud_id: chararray,name: chararray,grade: int,marks: map[int]}
grunt> dump stud_mark;
(a1234,AAA ,12,[ Chemistry#89, Physics#78,Maths#95])
(a1235,BBB ,12,[ Chemistry#84, Physics#48,Maths#45])
(a1324,CCC ,12,[ Chemistry#49, Physics#56,Maths#75])
(a1325,DDD ,12,[ Chemistry#44, Physics#59,Maths#65])
```

- To generate the names and marks of students in Chemistry.



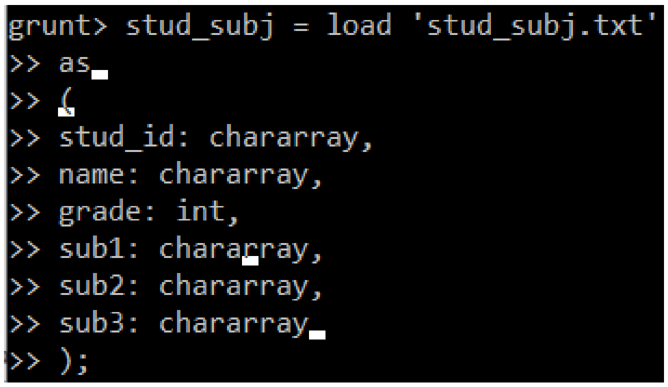
```
grunt> stud_chem = foreach stud_marks generate name, marks#' Chemistry';
grunt> dump stud_chem;
(AAA,89)
(BBB,84)
(CCC,49)
(DDD,44)
```

- Use stud_subj.txt

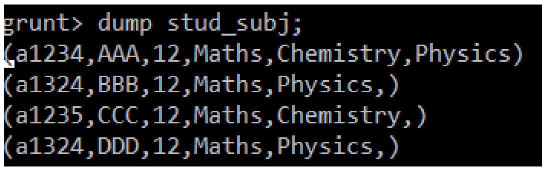


```
a1234    AAA    12    Maths    Chemistry    Physics
a1324    BBB    12    Maths    Physics
a1235    CCC    12    Maths    Chemistry
a1324    DDD    12    Maths    Physics      English
```

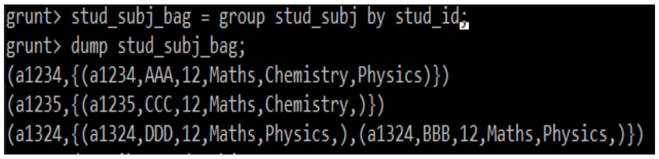
- Load it into a relation stud_subj.



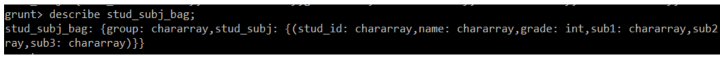
```
grunt> stud_subj = load 'stud_subj.txt'
>> as
>> (
>> stud_id: chararray,
>> name: chararray,
>> grade: int,
>> sub1: chararray,
>> sub2: chararray,
>> sub3: chararray
>> );
```



```
grunt> dump stud_subj;
(a1234,AAA,12,Maths,Chemistry,Physics)
(a1324,BBB,12,Maths,Physics,)
(a1235,CCC,12,Maths,Chemistry,)
(a1324,DDD,12,Maths,Physics,)
```

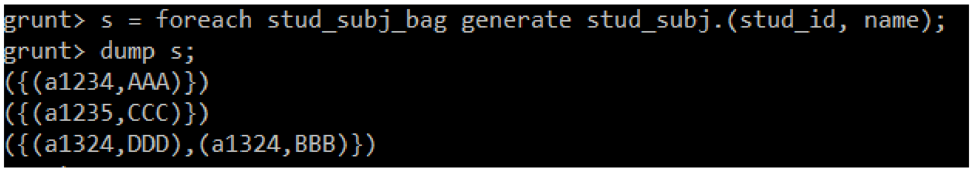


```
grunt> stud_subj_bag = group stud_subj by stud_id;
grunt> dump stud_subj_bag;
(a1234,{(a1234,AAA,12,Maths,Chemistry,Physics)})
(a1235,{(a1235,CCC,12,Maths,Chemistry,)})
(a1324,{(a1324,DDD,12,Maths,Physics,),(a1324,BBB,12,Maths,Physics,)})
```



```
grunt> describe stud_subj_bag;
stud_subj_bag: {group: chararray,stud_subj: {(stud_id: chararray,name: chararray,grade: int,sub1: chararray,sub2
ray,sub3: chararray)}}
```

- A bag is created by using the group by command.
- Every tuple has exactly two fields.
  - The first field will always be the group key, the field by which the original grouping is performed.
  - The second field is always a bag containing the original tuples that are associated with that group key.
- The name of the bag is the name of the original relation on which the group was done.
- It is not possible to access individual records within an inner bag.



```
grunt> s = foreach stud_subj_bag generate stud_subj.(stud_id, name);
grunt> dump s;
({(a1234,AAA)})
({(a1235,CCC)})
({(a1324,DDD),(a1324,BBB)})
```

- In this example access the stud_id and name fields from the inner bag.

Functions in pig

- UDF - User Defined Functions
  - Pig allows developers to write their own custom functions which operates on data
  - Pig supports UDFs in a number of programming languages such as Java, Python, Ruby, JavaScript
  - Pig comes prepackaged with a number of UDFs that can be directly used

Built-in UDF's

- These make Pig very powerful right out of the box.
- Four categories based on the function they perform.
  - Load – loads data into Pig from different kinds of files.
  - Store - serializes data onto HDFS or local file system.
  - Evaluate – runs a whole bunch of basic operations or transformations on fields within individual records.

δ Filter – allows to determine which record should be included in the resultant relation.

Load and Store

₪ PigStorage() – allows serialization and deserialization of files that are tab-delimited. Other delimiters are passed as an argument to PigStorage.

₪ HBaseStorage() – allows to read and write data into the HBase database.

₪ JsonLoader() – used if files are in JSON format.

₪ AvroStorage() – uses AvroStorage function in Pig.

₪ CSVExcelStorage() – used for Excel files in CSV format.

Evaluate Functions

₪ Math: Works with numeric data types

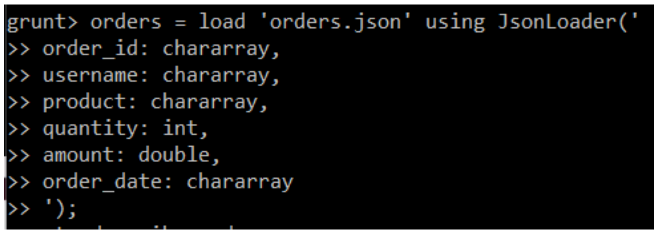
₪ String: Works with the chararray

₪ Date: Works with datetime

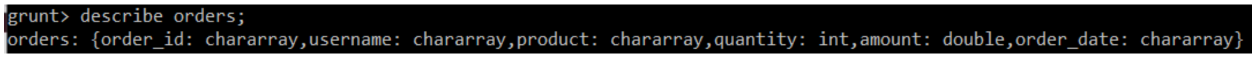
₪ Complex data types: Used with the tuple, bag and map type TOTUPLE(), TOBAG(), TOMAP()

₪ Aggregate: Different functions take different kinds of input SUM(), COUNT()

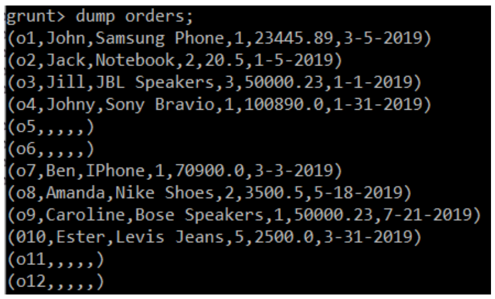
DEMO



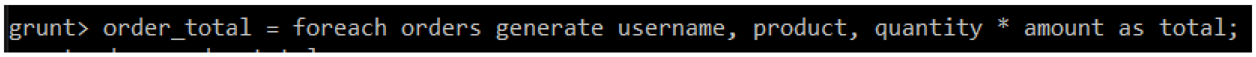
```
grunt> orders = load 'orders.json' using JsonLoader('
>> order_id: chararray,
>> username: chararray,
>> product: chararray,
>> quantity: int,
>> amount: double,
>> order_date: chararray
>> ');
```



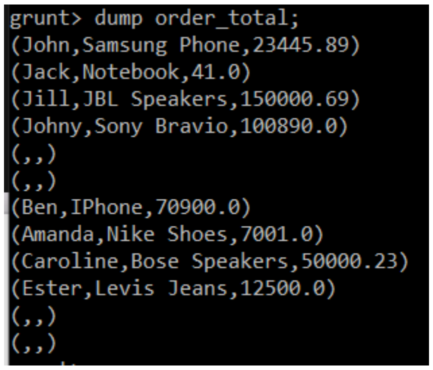
```
grunt> describe orders;
orders: {order_id: chararray,username: chararray,product: chararray,quantity: int,amount: double,order_date: chararray}
```



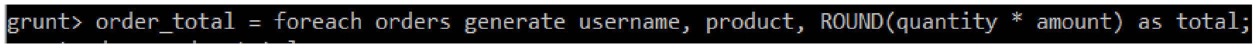
```
grunt> dump orders;
(o1,John,Samsung Phone,1,23445.89,3-5-2019)
(o2,Jack,Notebook,2,20.5,1-5-2019)
(o3,Jill,JBL Speakers,3,50000.23,1-1-2019)
(o4,Johny,Sony Bravio,1,100890.0,1-31-2019)
(o5,,,,,)
(o6,,,,,)
(o7,Ben,IPhone,1,70900.0,3-3-2019)
(o8,Amanda,Nike Shoes,2,3500.5,5-18-2019)
(o9,Caroline,Bose Speakers,1,50000.23,7-21-2019)
(010,Ester,Levis Jeans,5,2500.0,3-31-2019)
(o11,,,,,)
(o12,,,,,)
```



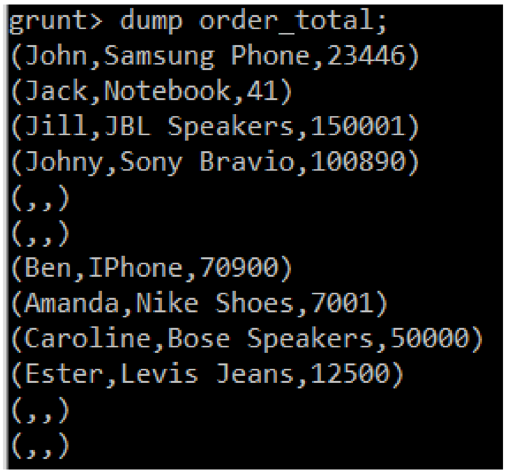
```
grunt> order_total = foreach orders generate username, product, quantity * amount as total;
```



```
grunt> dump order_total;
(John,Samsung Phone,23445.89)
(Jack,Notebook,41.0)
(Jill,JBL Speakers,150000.69)
(Johny,Sony Bravio,100890.0)
(,,)
(,,)
(Ben,IPhone,70900.0)
(Amanda,Nike Shoes,7001.0)
(Caroline,Bose Speakers,50000.23)
(Ester,Levis Jeans,12500.0)
(,,)
(,,)
```

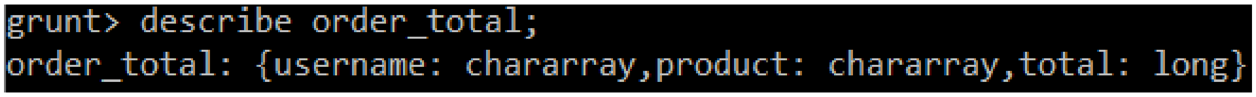


```
grunt> order_total = foreach orders generate username, product, ROUND(quantity * amount) as total;
```



```
grunt> dump order_total;
(John,Samsung Phone,23446)
(Jack,Notebook,41)
(Jill,JBL Speakers,150001)
(Johny,Sony Bravio,100890)
(,,)
(,,)
(Ben,IPhone,70900)
(Amanda,Nike Shoes,7001)
(Caroline,Bose Speakers,50000)
(Ester,Levis Jeans,12500)
(,,)
(,,)
```

₪ Rounds the product to the nearest integer.



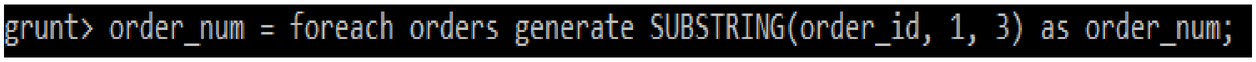
```
grunt> describe order_total;
order_total: {username: chararray,product: chararray,total: long}
```

₪ The total is now a long datatype.

₪ For the different maths functions

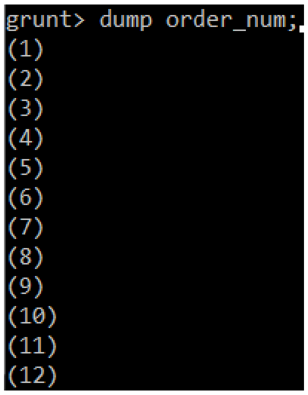
  δ https://pig.apache.org/docs/r0.17.0/func.html#math-functions

SUBSTRING(string, startIndex, stopIndex)



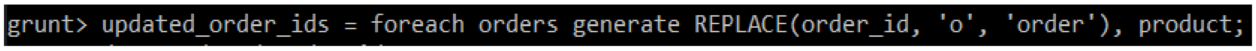
```
grunt> order_num = foreach orders generate SUBSTRING(order_id, 1, 3) as order_num;
```

  δ Uses from startIndex till stopIndex not including stopIndex.

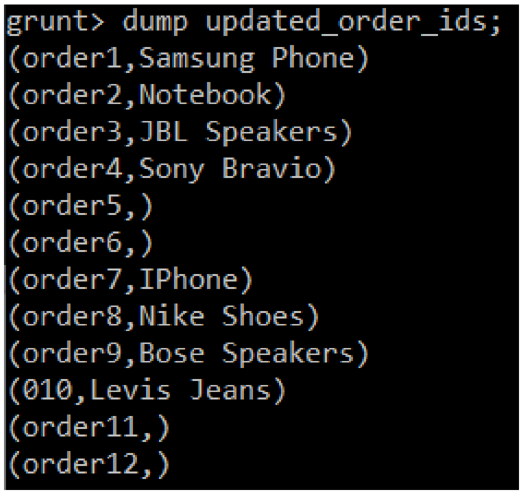


```
grunt> dump order_num;
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)
(9)
(10)
(11)
(12)
```

REPLACE(string, ‘oldCharacters’, ‘newCharacters’)

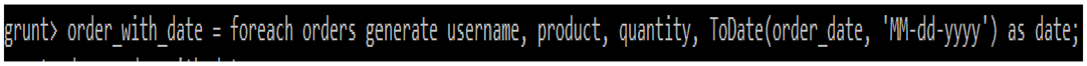


```
grunt> updated_order_ids = foreach orders generate REPLACE(order_id, 'o', 'order'), product;
```

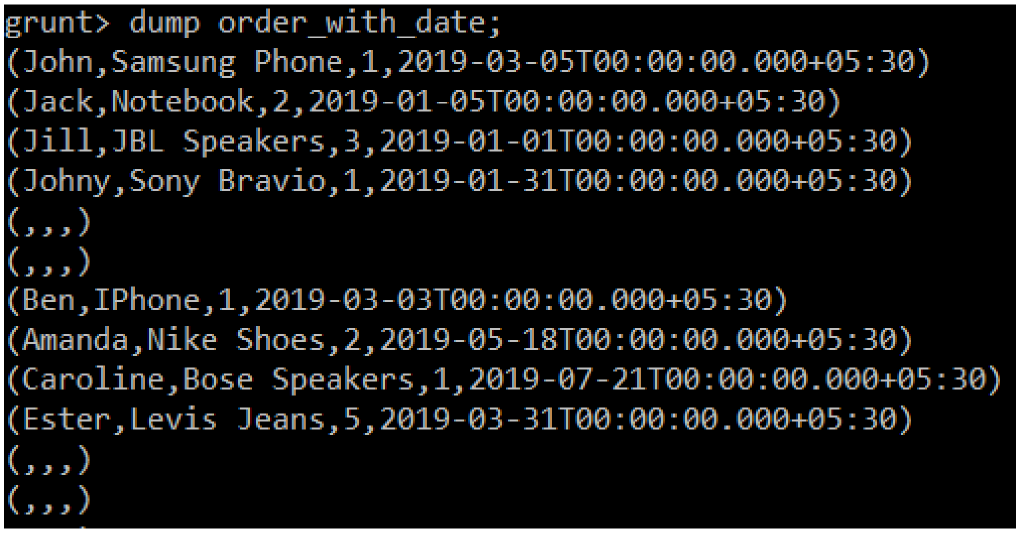


```
grunt> dump updated_order_ids;
(order1,Samsung Phone)
(order2,Notebook)
(order3,JBL Speakers)
(order4,Sony Bravio)
(order5,)
(order6,)
(order7,IPhone)
(order8,Nike Shoes)
(order9,Bose Speakers)
(010,Levis Jeans)
(order11,)
(order12,)
```

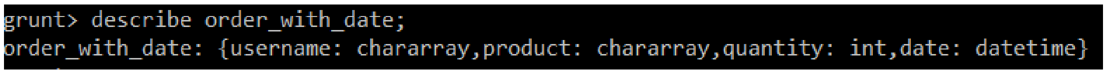
δ https://pig.apache.org/docs/r0.9.1/func.html#string-functions



```
grunt> order_with_date = foreach orders generate username, product, quantity, ToDate(order_date, 'MM-dd-yyyy') as date;
```



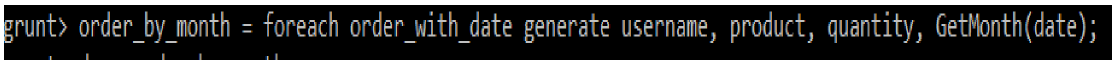
```
grunt> dump order_with_date;
(John,Samsung Phone,1,2019-03-05T00:00:00.000+05:30)
(Jack,Notebook,2,2019-01-05T00:00:00.000+05:30)
(Jill,JBL Speakers,3,2019-01-01T00:00:00.000+05:30)
(Johny,Sony Bravio,1,2019-01-31T00:00:00.000+05:30)
(,,,)
(,,,)
(Ben,IPhone,1,2019-03-03T00:00:00.000+05:30)
(Amanda,Nike Shoes,2,2019-05-18T00:00:00.000+05:30)
(Caroline,Bose Speakers,1,2019-07-21T00:00:00.000+05:30)
(Ester,Levis Jeans,5,2019-03-31T00:00:00.000+05:30)
(,,,)
(,,,)
```



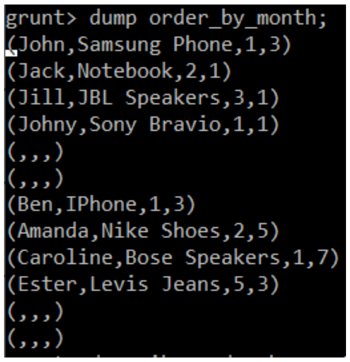
```
grunt> describe order_with_date;
order_with_date: {username: chararray,product: chararray,quantity: int,date: datetime}
```

₪ To convert to a DateTime type using ToDate(string, format)

δ Format is in which the original date is specified.

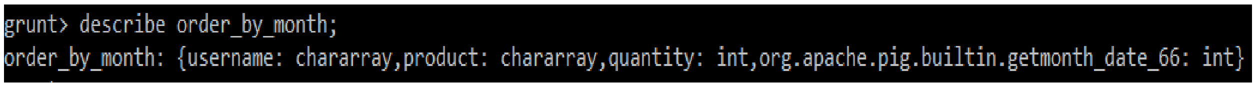


```
grunt> order_by_month = foreach order_with_date generate username, product, quantity, GetMonth(date);
```



```
grunt> dump order_by_month;
(John,Samsung Phone,1,3)
(Jack,Notebook,2,1)
(Jill,JBL Speakers,3,1)
(Johny,Sony Bravio,1,1)
(,,,)
(,,,)
(Ben,IPhone,1,3)
(Amanda,Nike Shoes,2,5)
(Caroline,Bose Speakers,1,7)
(Ester,Levis Jeans,5,3)
(,,,)
(,,,)
```

- GetMonth(field) extracts the Month from the datetime field in the form of an integer.
- https://pig.apache.org/docs/r0.11.1/func.html#datetime-functions



```
grunt> describe order_by_month;
order_by_month: {username: chararray,product: chararray,quantity: int,org.apache.pig.builtin.getmonth_date_66: int}
```

Filter

- A condition determines which record is in the result dataset
  - If Condition = true: Include record
  - If Condition = false: Leave record out
- Equivalent of the where clause in SQL

Conditional Operators for filters

Operators	Work with
==	Scalar, Maps, Tuples
!=	Scalar, Maps, Tuples
>	Scalar
<	Scalar
>=	Scalar
<=	Scalar

Filter functions

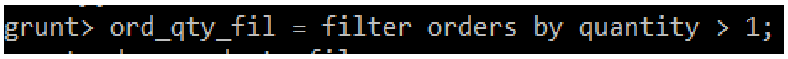
- These allow more powerful and logical ways of filtering the dataset.
- They are built-in UDFs that can be used with filter commands.
- Example

- matches – used with chararray to match a certain pattern and returns true or false
- IsEmpty – used with bag or map to check id there are any entities present in them and returns true or false

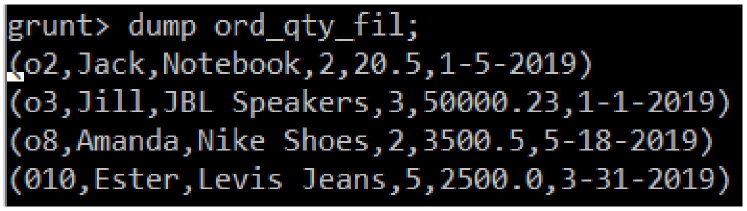
- Example
  - Order_id matches 'OD01'
  - Order_id matches 'OD01.*)
    - Uses regular expression to check orders that have a prefix OD01.

DEMO

- Use the orders relation.

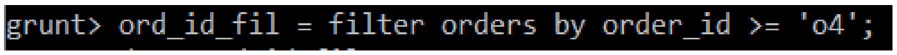


```
grunt> ord_qty_fil = filter orders by quantity > 1;
```

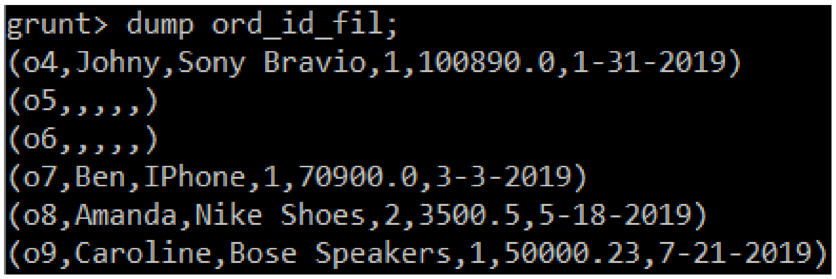


```
grunt> dump ord_qty_fil;
(o2,Jack,Notebook,2,20.5,1-5-2019)
(o3,Jill,JBL Speakers,3,50000.23,1-1-2019)
(o8,Amanda,Nike Shoes,2,3500.5,5-18-2019)
(010,Ester,Levis Jeans,5,2500.0,3-31-2019)
```

- Run a filter command using the quantity field.

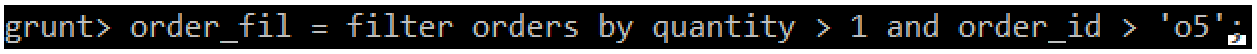


```
grunt> ord_id_fil = filter orders by order_id >= 'o4';
```

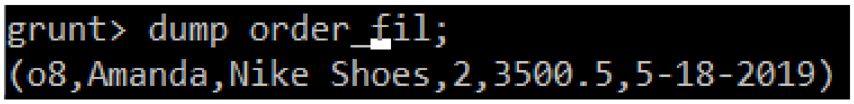


```
grunt> dump ord_id_fil;
(o4,Johny,Sony Bravio,1,100890.0,1-31-2019)
(o5,,,,,)
(o6,,,,,)
(o7,Ben,IPhone,1,70900.0,3-3-2019)
(o8,Amanda,Nike Shoes,2,3500.5,5-18-2019)
(o9,Caroline,Bose Speakers,1,50000.23,7-21-2019)
```

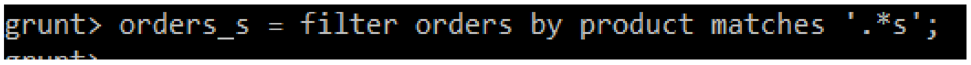
Logical Operators

- Logical Operators of and, or and not can be used with filter functions.
- 

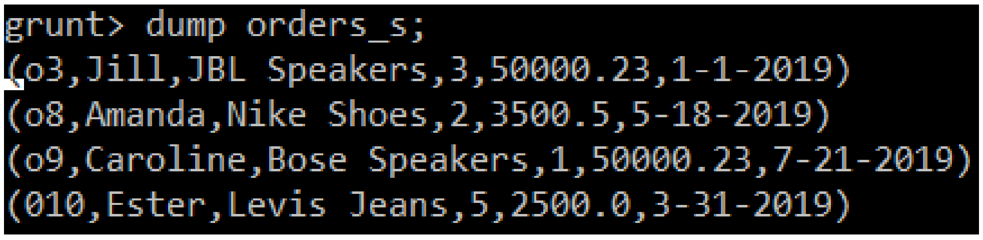
```
grunt> order_fil = filter orders by quantity > 1 and order_id > 'o5';
```

- 

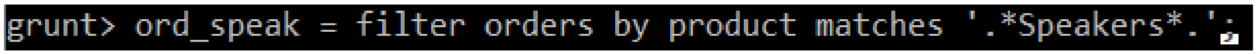
```
grunt> dump order_fil;
(o8,Amanda,Nike Shoes,2,3500.5,5-18-2019)
```

- 

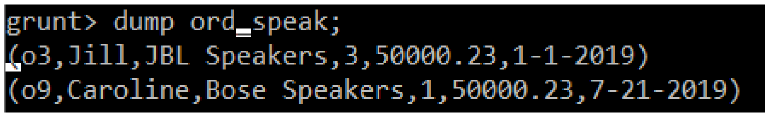
```
grunt> orders_s = filter orders by product matches '.*s';
```

- 

```
grunt> dump orders_s;
(o3,Jill,JBL Speakers,3,50000.23,1-1-2019)
(o8,Amanda,Nike Shoes,2,3500.5,5-18-2019)
(o9,Caroline,Bose Speakers,1,50000.23,7-21-2019)
(010,Ester,Levis Jeans,5,2500.0,3-31-2019)
```

- Match is case sensitive.
- 

```
grunt> ord_speak = filter orders by product matches '.*Speakers*.';
```

- 

```
grunt> dump ord_speak;
(o3,Jill,JBL Speakers,3,50000.23,1-1-2019)
(o9,Caroline,Bose Speakers,1,50000.23,7-21-2019)
```

- Display all records that have 'Speakers' in them.

Distinct

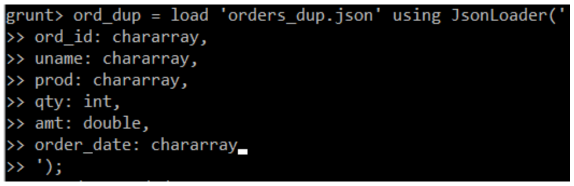
- Remove duplicate tuples from a relation
- Acts on entire records in a relation.
  - Tuples where all fields have the same value are considered duplicates
  - Compares all the fields of a tuple with all fields of the tuple in the same relation and removes duplicate values.
- Does not act on individual fields.

Limit

- Choose a specified number of tuples from a relation.
- It always chooses **the first** specified number of records.
- It does not perform random sampling.

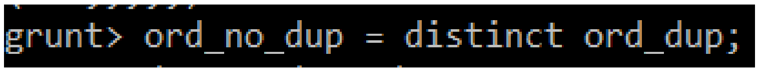
Demo

- Use orders_dup.json.



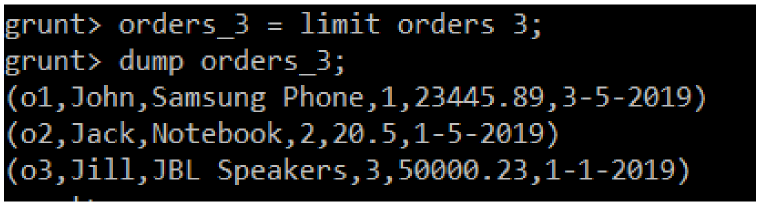
```
grunt> ord_dup = load 'orders_dup.json' using JsonLoader('
>> ord_id: chararray,
>> uname: chararray,
>> prod: chararray,
>> qty: int,
>> amt: double,
>> order_date: chararray
>> ');
```

- To remove duplicates



```
grunt> ord_no_dup = distinct ord_dup;
```

- To limit the number of relations



```
grunt> orders_3 = limit orders 3;
grunt> dump orders_3;
(o1,John,Samsung Phone,1,23445.89,3-5-2019)
(o2,Jack,Notebook,2,20.5,1-5-2019)
(o3,Jill,JBL Speakers,3,50000.23,1-1-2019)
```

Order By

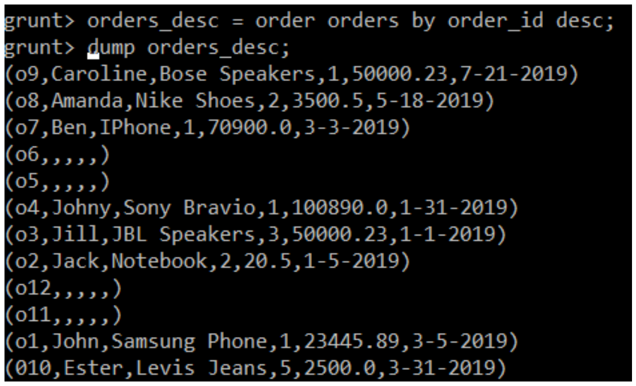
₪ Sort tuples in ascending or descending order based on a column value.

₪ Sorting is done based on a particular column

₪ Default is ascending order

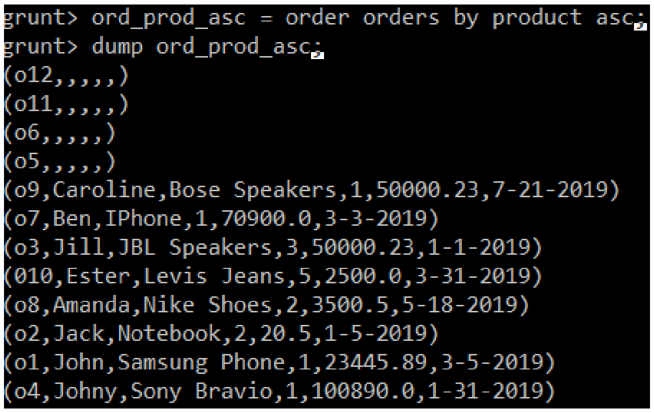
DEMO

₪ To display the relation based on descending sorted order of the order_id.



```
grunt> orders_desc = order orders by order_id desc;
grunt> dump orders_desc;
(o9,Caroline,Bose Speakers,1,50000.23,7-21-2019)
(o8,Amanda,Nike Shoes,2,3500.5,5-18-2019)
(o7,Ben,IPhone,1,70900.0,3-3-2019)
(o6,,,,,)
(o5,,,,,)
(o4,Johny,Sony Bravio,1,100890.0,1-31-2019)
(o3,Jill,JBL Speakers,3,50000.23,1-1-2019)
(o2,Jack,Notebook,2,20.5,1-5-2019)
(o12,,,,,)
(o11,,,,,)
(o1,John,Samsung Phone,1,23445.89,3-5-2019)
(010,Ester,Levis Jeans,5,2500.0,3-31-2019)
```

₪ To display relations based on products in ascending order.



```
grunt> ord_prod_asc = order orders by product asc;
grunt> dump ord_prod_asc;
(o12,,,,,)
(o11,,,,,)
(o6,,,,,)
(o5,,,,,)
(o9,Caroline,Bose Speakers,1,50000.23,7-21-2019)
(o7,Ben,IPhone,1,70900.0,3-3-2019)
(o3,Jill,JBL Speakers,3,50000.23,1-1-2019)
(010,Ester,Levis Jeans,5,2500.0,3-31-2019)
(o8,Amanda,Nike Shoes,2,3500.5,5-18-2019)
(o2,Jack,Notebook,2,20.5,1-5-2019)
(o1,John,Samsung Phone,1,23445.89,3-5-2019)
(o4,Johny,Sony Bravio,1,100890.0,1-31-2019)
```

Split

₪ It acts like a case switch statement.

₪ Multiple conditions are specified and the records that match those conditions are sent to a different relation.

₪ It splits the data flows in the relation based on certain conditions.

₪ Explicitly split data into two or more data flows based on conditions.

₪ The result of a split is two or more relations.

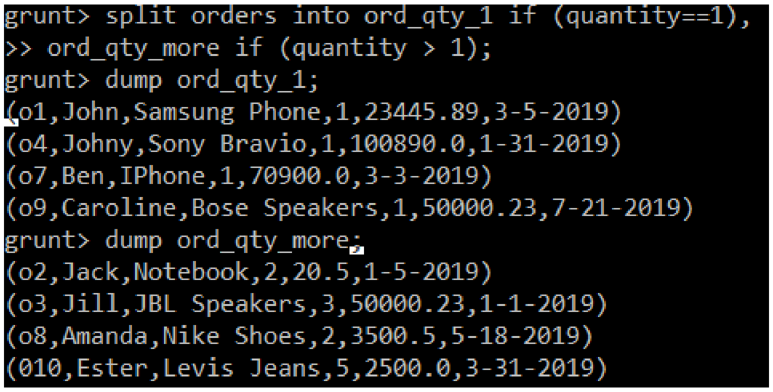
₪ The number of conditions specified determines the number of relations.

₪ The same record can go to multiple legs of the split.

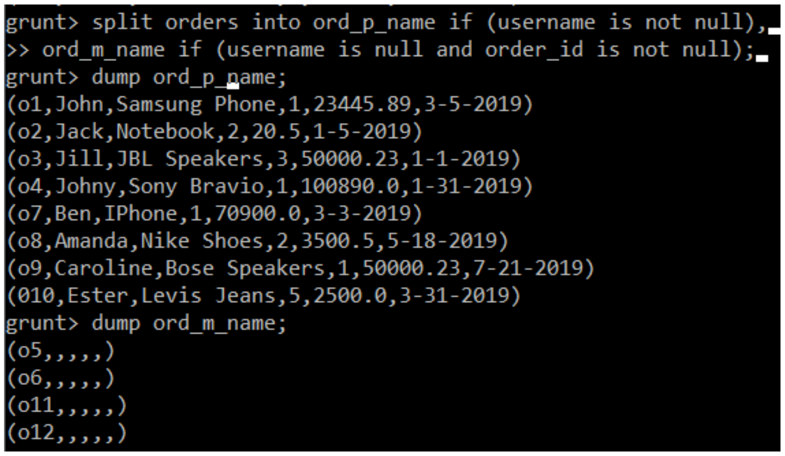
- A single record can match more than one condition and will be duplicated to more than one relation.
- In a split command the conditions need not be mutually exclusive.
- They can overlap.

Demo

- Use orders relation.
- Split the orders based on the quantity.



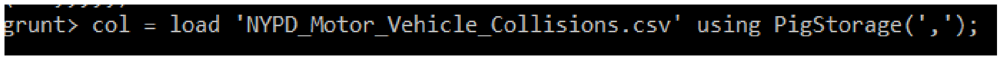
```
grunt> split orders into ord_qty_1 if (quantity==1),
>> ord_qty_more if (quantity > 1);
grunt> dump ord_qty_1;
(o1,John,Samsung Phone,1,23445.89,3-5-2019)
(o4,Johny,Sony Bravio,1,100890.0,1-31-2019)
(o7,Ben,IPhone,1,70900.0,3-3-2019)
(o9,Caroline,Bose Speakers,1,50000.23,7-21-2019)
grunt> dump ord_qty_more;
(o2,Jack,Notebook,2,20.5,1-5-2019)
(o3,Jill,JBL Speakers,3,50000.23,1-1-2019)
(o8,Amanda,Nike Shoes,2,3500.5,5-18-2019)
(010,Ester,Levis Jeans,5,2500.0,3-31-2019)
```



```
grunt> split orders into ord_p_name if (username is not null),
>> ord_m_name if (username is null and order_id is not null);
grunt> dump ord_p_name;
(o1,John,Samsung Phone,1,23445.89,3-5-2019)
(o2,Jack,Notebook,2,20.5,1-5-2019)
(o3,Jill,JBL Speakers,3,50000.23,1-1-2019)
(o4,Johny,Sony Bravio,1,100890.0,1-31-2019)
(o7,Ben,IPhone,1,70900.0,3-3-2019)
(o8,Amanda,Nike Shoes,2,3500.5,5-18-2019)
(o9,Caroline,Bose Speakers,1,50000.23,7-21-2019)
(010,Ester,Levis Jeans,5,2500.0,3-31-2019)
grunt> dump ord_m_name;
(o5,,,,,)
(o6,,,,,)
(o11,,,,,)
(o12,,,,,)
```

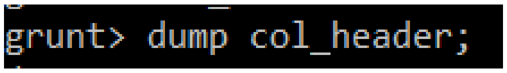
DEMO

- Use the NYPD_Motor_Vehicle_Collisions.csv

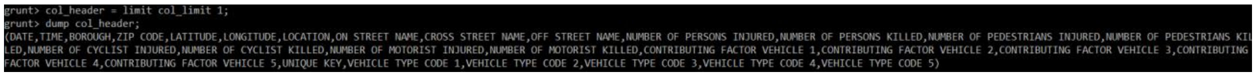


```
grunt> col = load 'NYPD_Motor_Vehicle_Collisions.csv' using PigStorage(',');
```

- Create another relation with only 1000 relations called col_limit.
- To see the headings of the columns

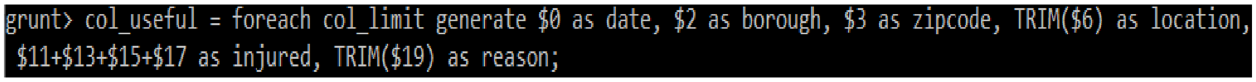


```
grunt> dump col_header;
```



```
grunt> col_header = limit col_limit 1;
grunt> dump col_header;
(DATE,TIME,BOROUGH,ZIP CODE,LATITUDE,LONGITUDE,LOCATION,ON STREET NAME,CROSS STREET NAME,OFF STREET NAME,NUMBER OF PERSONS INJURED,NUMBER OF PERSONS KILLED,NUMBER OF PEDESTRIANS INJURED,NUMBER OF PEDESTRIANS KIL
LED,NUMBER OF CYCLIST INJURED,NUMBER OF CYCLIST KILLED,NUMBER OF MOTORIST INJURED,NUMBER OF MOTORIST KILLED,CONTRIBUTING FACTOR VEHICLE 1,CONTRIBUTING FACTOR VEHICLE 2,CONTRIBUTING FACTOR VEHICLE 3,CONTRIBUTING
FACTOR VEHICLE 4,CONTRIBUTING FACTOR VEHICLE 5,UNIQUE KEY,VEHICLE TYPE CODE 1,VEHICLE TYPE CODE 2,VEHICLE TYPE CODE 3,VEHICLE TYPE CODE 4,VEHICLE TYPE CODE 5)
```

₪ To get useful columns



```
grunt> col_useful = foreach col_limit generate $0 as date, $2 as borough, $3 as zipcode, TRIM($6) as location,
 $11+$13+$15+$17 as injured, TRIM($19) as reason;
```

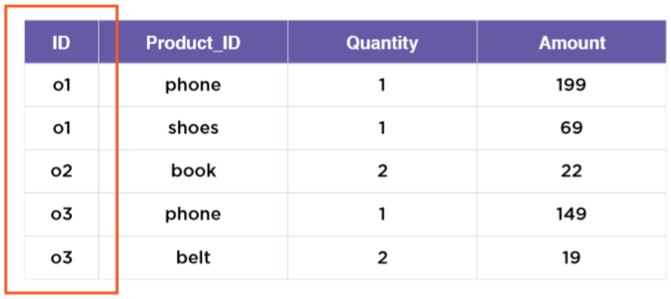
- δ Since number of people injured includes passengers, pedestrians, cyclists, motorists it has to be summed.

Grouping Records on the Same Key

₪ While performing a group by operation on a relation, it is conceptualized as a tuple of fields.

₪ Example

δ To group based on id use the group command



ID	Product_ID	Quantity	Amount
o1	phone	1	199
o1	shoes	1	69
o2	book	2	22
o3	phone	1	149
o3	belt	2	19

GROUP BY SAME KEY

₪ While using the group by command, Pig extracts that one field and finds all tuples that have the same value of that field.

₪ It will create a bag of these tuples and then associates this bag of tuples with the group value.

₪ Pig will

δ Extract every unique value in the group column.

δ Find the tuples which have the same value as the group and put them in a bag.

₪ All records with the same key are grouped into a bag.

₪ The relation got when a group by operation has been performed has only two fields.

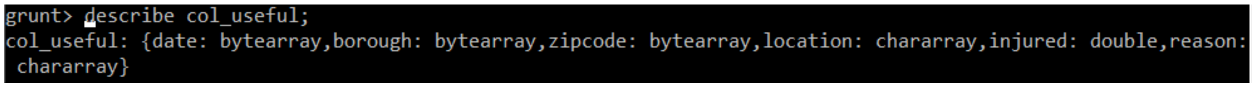
δ The first field is named group and contains all the fields the relation is grouped by.

δ The second field will be a bag with the name based on the name of the relation on which the group by was performed originally.

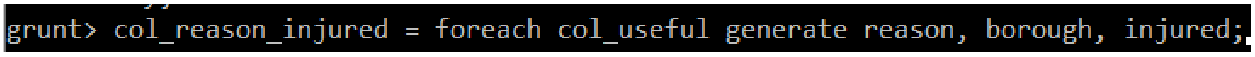
₪ The bag contains all tuples that have the same key.

DEMO

- Use col_useful.

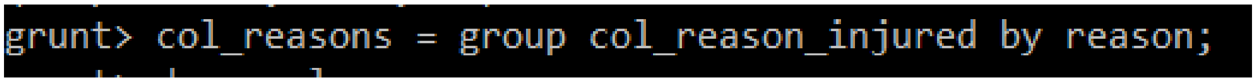


```
grunt> describe col_useful;
col_useful: {date: bytearray,borough: bytearray,zipcode: bytearray,location: chararray,injured: double,reason:
 chararray}
```

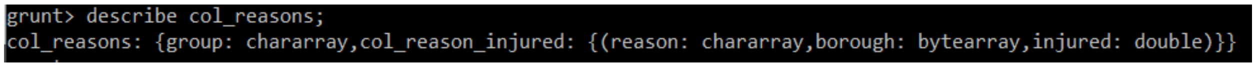


```
grunt> col_reason_injured = foreach col_useful generate reason, borough, injured;
```

- Dump the relation.
- Group the relation based on the reasons.

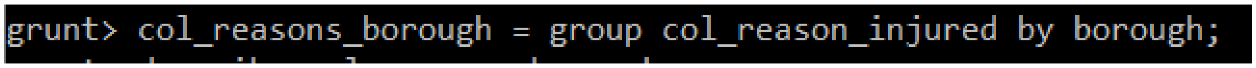


```
grunt> col_reasons = group col_reason_injured by reason;
```

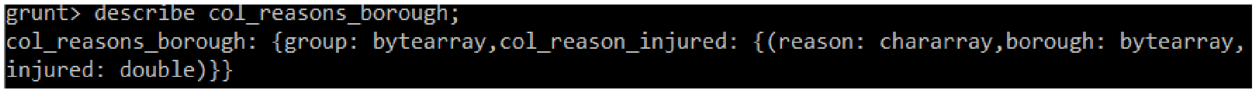


```
grunt> describe col_reasons;
col_reasons: {group: chararray,col_reason_injured: {(reason: chararray,borough: bytearray,injured: double)}}
```

- The relation has two fields.
  - One field is named group that is of type chararray which is the reason for the collision to occur.
  - The next field is a bag that is called col_reason_injured.
    - This field name comes from the original relation to which the grouping operation was applied to get this relation.

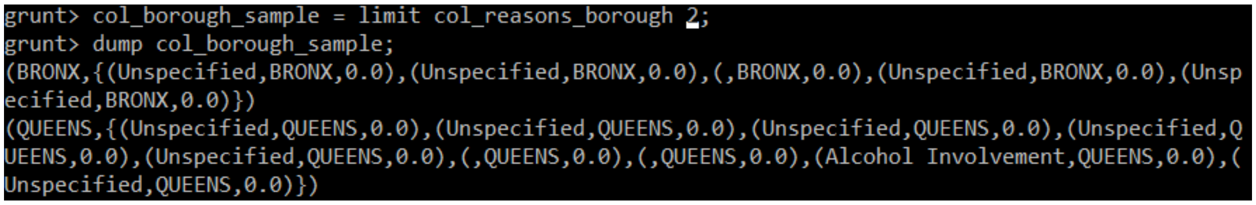


```
grunt> col_reasons_borough = group col_reason_injured by borough;
```



```
grunt> describe col_reasons_borough;
col_reasons_borough: {group: bytearray,col_reason_injured: {(reason: chararray,borough: bytearray,
injured: double)}}
```

- Limit the number if records to 2 and dump the relation.



```
grunt> col_borough_sample = limit col_reasons_borough 2;
grunt> dump col_borough_sample;
(BRONX,{(Unspecified,BRONX,0.0),(Unspecified,BRONX,0.0),(,BRONX,0.0),(Unspecified,BRONX,0.0),(Unsp
ecified,BRONX,0.0)})
(QUEENS,{(Unspecified,QUEENS,0.0),(Unspecified,QUEENS,0.0),(Unspecified,QUEENS,0.0),(Unspecified,Q
UEENS,0.0),(Unspecified,QUEENS,0.0),(,QUEENS,0.0),(,QUEENS,0.0),(Alcohol Involvement,QUEENS,0.0),(
Unspecified,QUEENS,0.0)})
```

Performing aggregations on grouped records

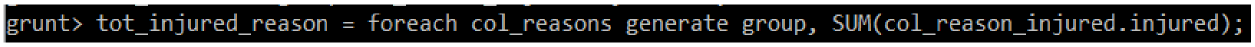
- Aggregations are UDFs which can be applied to field values from multiple records
- Example
  - Count the number of different products in each order – count()
    - Will count the number of different products within each order.
    - It applies within a particular group that is across all tuples in a particular that corresponds to a particular group key.
- Example

δ Find the total amount spent per order.
  - ∞ Group them based on order_id.
  - ∞ Perform the sum operation on the total revenues.
  - ∞ Sum() will sum up the amounts in each bag associated with a particular group key.

DEMO

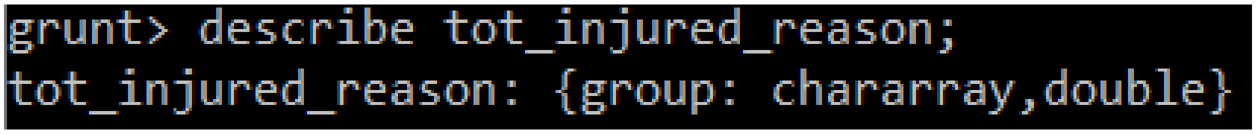
₪ Kind of collisions that caused the most injuries in New York?

₪ Use col_reasons.



```
grunt> tot_injured_reason = foreach col_reasons generate group, SUM(col_reason_injured.injured);
```

₪ Describe the relation

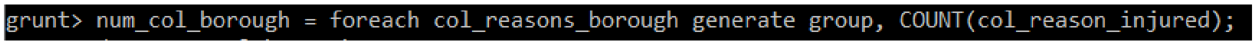


```
grunt> describe tot_injured_reason;
tot_injured_reason: {group: chararray,double}
```

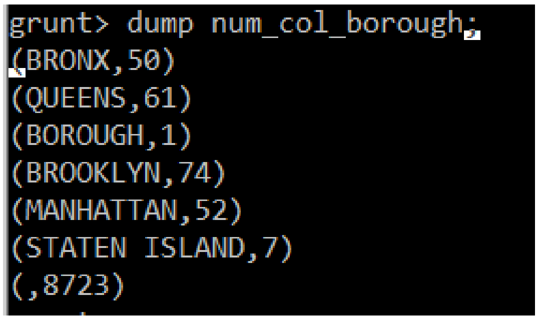
₪ Dump the relation to see the options.

₪ Boroughs that have the most collisions?

₪ Use col_reasons_borough.

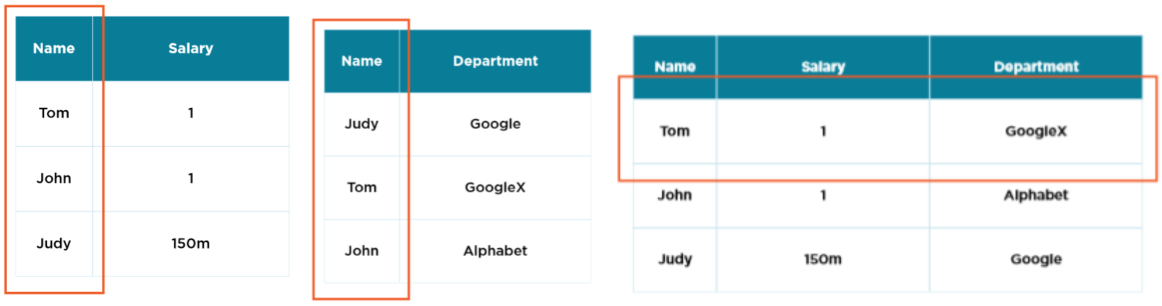


```
grunt> num_col_borough = foreach col_reasons_borough generate group, COUNT(col_reason_injured);
```



```
grunt> dump num_col_borough;
(BRONX,50)
(QUEENS,61)
(BOROUGH,1)
(BROOKLYN,74)
(MANHATTAN,52)
(STATEN ISLAND,7)
(,8723)
```

Join Operations in Pig



Name	Salary
Tom	1
John	1
Judy	150m

Name	Department
Judy	Google
Tom	GoogleX
John	Alphabet

Name	Salary	Department
Tom	1	GoogleX
John	1	Alphabet
Judy	150m	Google

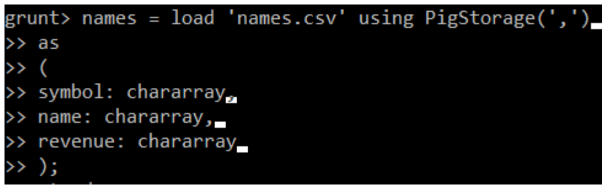
- Join operation requires a joined column or a field, that is the field which is used to perform the match.
- The result of joining two or more relations should give one relation where records from each relation is matched on the join column.
- Pig provides support only for equi-joins.
- The join fields have to be matched exactly.
- Other conditional operators cannot be used on join column.
- The only conditional operator used is the equality.

DEMO

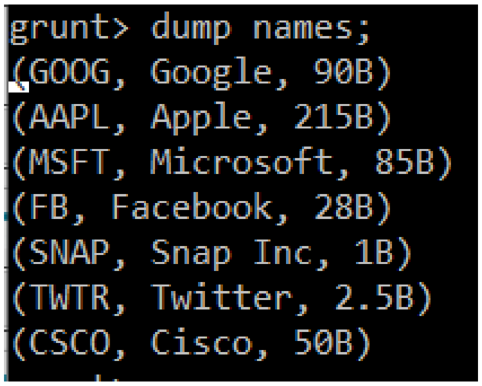
- Use names.csv

```
GOOG, Google, 90B
AAPL, Apple, 215B
MSFT, Microsoft, 85B
FB, Facebook, 28B
SNAP, Snap Inc, 1B
TWTR, Twitter, 2.5B
CSCO, Cisco, 50B
```

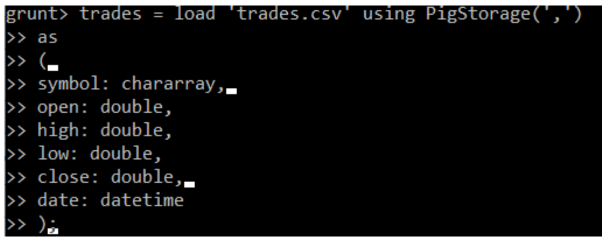
- Use trades.csv
  - The file contains creating information for a number of stocks.
  - Contains the symbol, open, close, high, lows for a particular day and the date on which the stock was created.



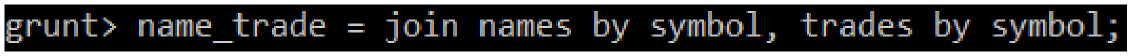
```
grunt> names = load 'names.csv' using PigStorage(',')
>> as
>> (
>> symbol: chararray,
>> name: chararray,
>> revenue: chararray
>> );
```



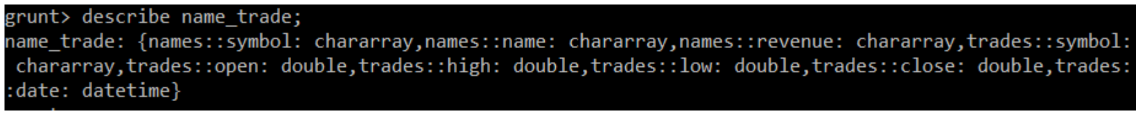
```
grunt> dump names;
(GOOG, Google, 90B)
(AAPL, Apple, 215B)
(MSFT, Microsoft, 85B)
(FB, Facebook, 28B)
(SNAP, Snap Inc, 1B)
(TWTR, Twitter, 2.5B)
(CSCO, Cisco, 50B)
```



```
grunt> trades = load 'trades.csv' using PigStorage(',')
>> as
>> (
>> symbol: chararray,
>> open: double,
>> high: double,
>> low: double,
>> close: double,
>> date: datetime
>> );
```



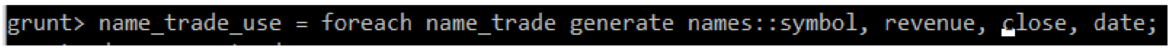
```
grunt> name_trade = join names by symbol, trades by symbol;
```



```
grunt> describe name_trade;
name_trade: {names::symbol: chararray,names::name: chararray,names::revenue: chararray,trades::symbol:
 chararray,trades::open: double,trades::high: double,trades::low: double,trades::close: double,trades:
:date: datetime}
```

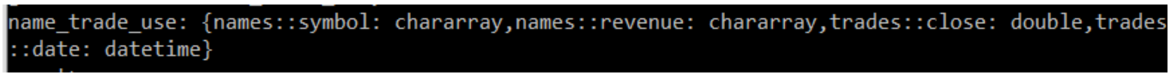
- The symbol field is present in both the relations.

- Every field in the joined relation has a prefix that indicates the original relation that the field came from.
- :: is used as separator between the relation name and the name of the field.



```
grunt> name_trade_use = foreach name_trade generate names::symbol, revenue, close, date;
```

- Fields that are specified in both the relations has to be identified with the name of the relation.



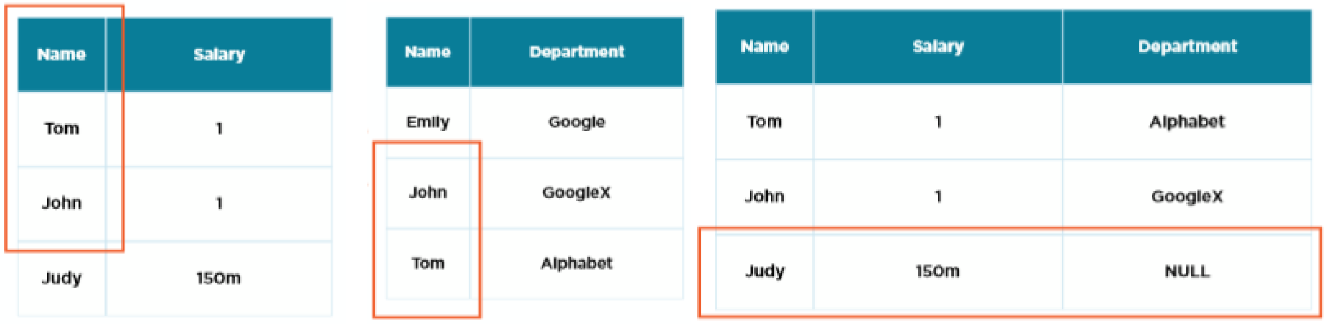
```
name_trade_use: {names::symbol: chararray,names::revenue: chararray,trades::close: double,trades
::date: datetime}
```

Types of joins

- Left Outer Join
- Right Outer Join
- Full outer Join
- Self Join
- Cross Join

Left outer join

- Every record on the left table will be present in the result with a matching record padded with nulls



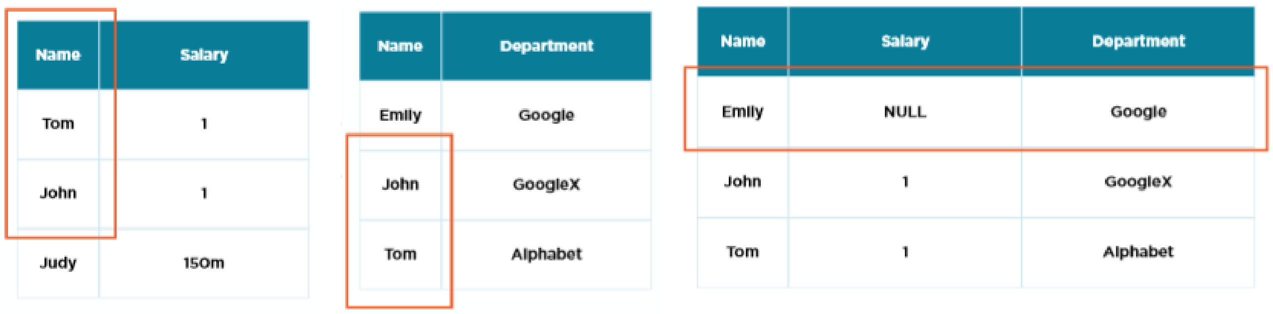
Name	Salary
Tom	1
John	1
Judy	150m

Name	Department
Emily	Google
John	GoogleX
Tom	Alphabet

Name	Salary	Department
Tom	1	Alphabet
John	1	GoogleX
Judy	150m	NULL

Right outer join

- Every record on the right table will be present in the result with a matching record padded with nulls



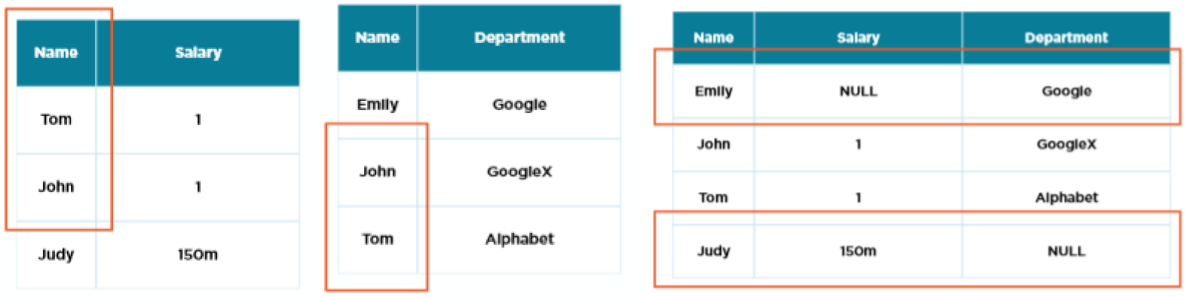
Name	Salary
Tom	1
John	1
Judy	150m

Name	Department
Emily	Google
John	GoogleX
Tom	Alphabet

Name	Salary	Department
Emily	NULL	Google
John	1	GoogleX
Tom	1	Alphabet

Full outer join

- Records from both tables will be present in the result with a matching record padded with nulls



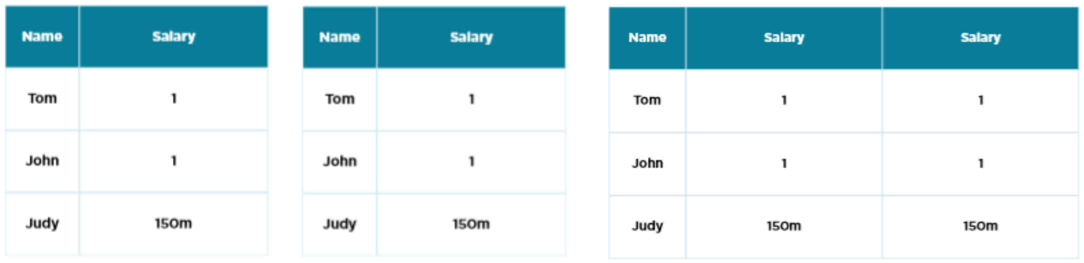
Name	Salary
Tom	1
John	1
Judy	150m

Name	Department
Emily	Google
John	GoogleX
Tom	Alphabet

Name	Salary	Department
Emily	NULL	Google
John	1	GoogleX
Tom	1	Alphabet
Judy	150m	NULL

Self join

- Both tables are exactly the same.



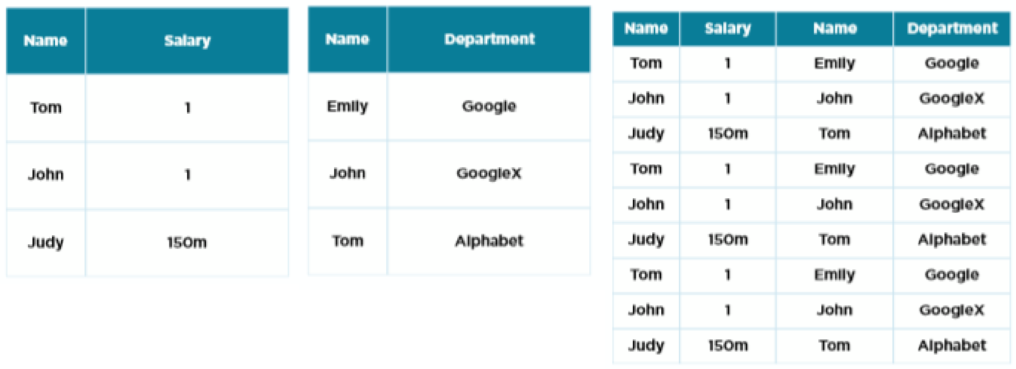
Name	Salary
Tom	1
John	1
Judy	150m

Name	Salary
Tom	1
John	1
Judy	150m

Name	Salary	Salary
Tom	1	1
John	1	1
Judy	150m	150m

Cross join

- Is a cross product where every record in the left relation is matched with every record in the right relation.
- No column values are compared to find matching records.
- The cross join should be very carefully because it is huge resource hog in pig.



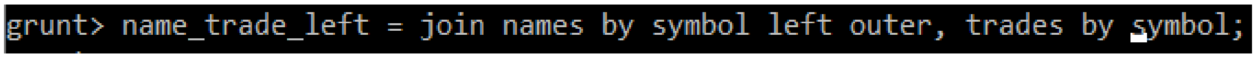
Name	Salary
Tom	1
John	1
Judy	150m

Name	Department
Emily	Google
John	GoogleX
Tom	Alphabet

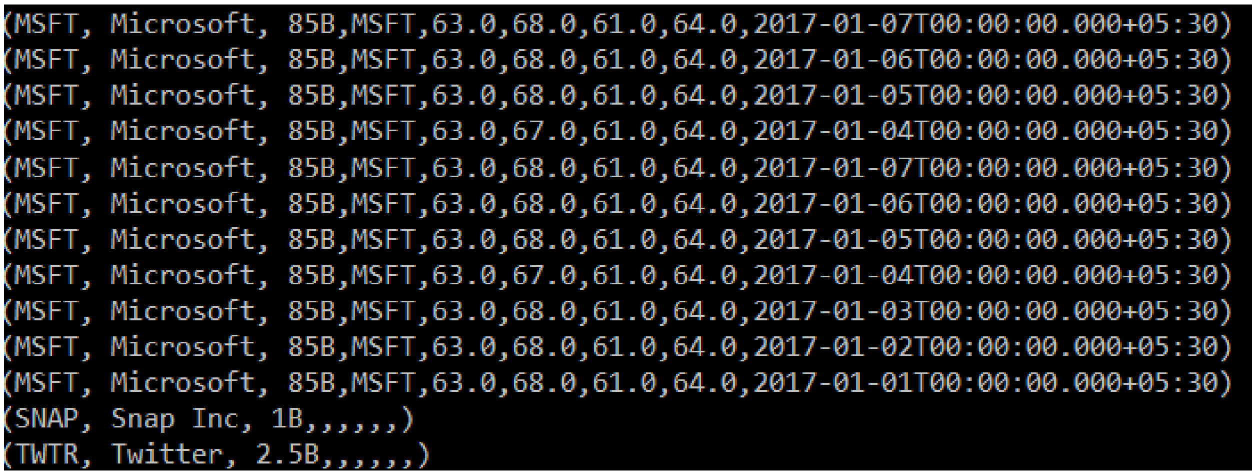
Name	Salary	Name	Department
Tom	1	Emily	Google
John	1	John	GoogleX
Judy	150m	Tom	Alphabet
Tom	1	Emily	Google
John	1	John	GoogleX
Judy	150m	Tom	Alphabet
Tom	1	Emily	Google
John	1	John	GoogleX
Judy	150m	Tom	Alphabet

DEMO

- Use names and trades relation.
- Left outer join

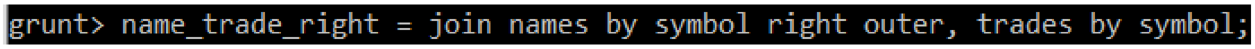


```
grunt> name_trade_left = join names by symbol left outer, trades by symbol;
```



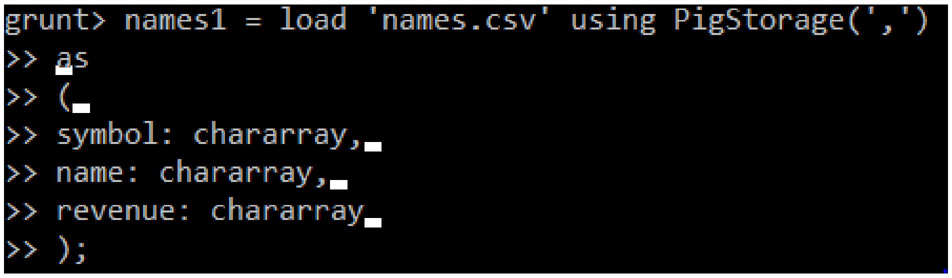
```
(MSFT, Microsoft, 85B,MSFT,63.0,68.0,61.0,64.0,2017-01-07T00:00:00.000+05:30)
(MSFT, Microsoft, 85B,MSFT,63.0,68.0,61.0,64.0,2017-01-06T00:00:00.000+05:30)
(MSFT, Microsoft, 85B,MSFT,63.0,68.0,61.0,64.0,2017-01-05T00:00:00.000+05:30)
(MSFT, Microsoft, 85B,MSFT,63.0,67.0,61.0,64.0,2017-01-04T00:00:00.000+05:30)
(MSFT, Microsoft, 85B,MSFT,63.0,68.0,61.0,64.0,2017-01-07T00:00:00.000+05:30)
(MSFT, Microsoft, 85B,MSFT,63.0,68.0,61.0,64.0,2017-01-06T00:00:00.000+05:30)
(MSFT, Microsoft, 85B,MSFT,63.0,68.0,61.0,64.0,2017-01-05T00:00:00.000+05:30)
(MSFT, Microsoft, 85B,MSFT,63.0,67.0,61.0,64.0,2017-01-04T00:00:00.000+05:30)
(MSFT, Microsoft, 85B,MSFT,63.0,68.0,61.0,64.0,2017-01-03T00:00:00.000+05:30)
(MSFT, Microsoft, 85B,MSFT,63.0,68.0,61.0,64.0,2017-01-02T00:00:00.000+05:30)
(MSFT, Microsoft, 85B,MSFT,63.0,68.0,61.0,64.0,2017-01-01T00:00:00.000+05:30)
(SNAP, Snap Inc, 1B,,,,,,)
(TWTR, Twitter, 2.5B,,,,,,)
```

- Right outer join

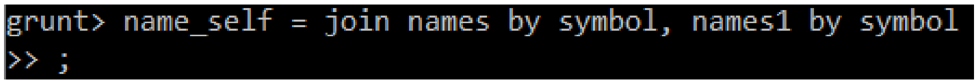


```
grunt> name_trade_right = join names by symbol right outer, trades by symbol;
```

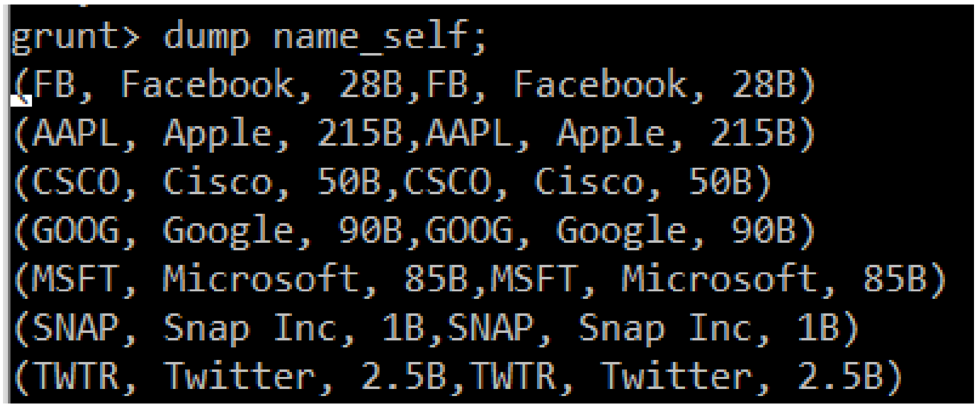
- Self join
  - In pig the same relation cannot be used on either side of the join.
  - Recreate the relation with another name



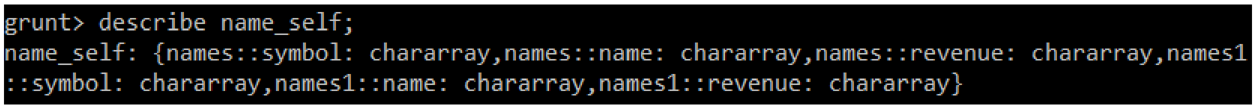
```
grunt> names1 = load 'names.csv' using PigStorage(',')
>> as
>> (
>> symbol: chararray,
>> name: chararray,
>> revenue: chararray
>> );
```



```
grunt> name_self = join names by symbol, names1 by symbol
>> ;
```

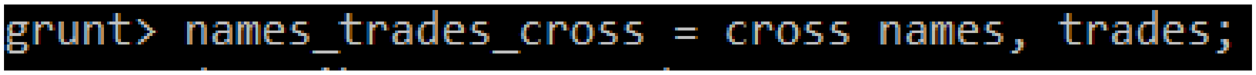


```
grunt> dump name_self;
(FB, Facebook, 28B,FB, Facebook, 28B)
(AAPL, Apple, 215B,AAPL, Apple, 215B)
(CSCO, Cisco, 50B,CSCO, Cisco, 50B)
(GOOG, Google, 90B,GOOG, Google, 90B)
(MSFT, Microsoft, 85B,MSFT, Microsoft, 85B)
(SNAP, Snap Inc, 1B,SNAP, Snap Inc, 1B)
(TWTR, Twitter, 2.5B,TWTR, Twitter, 2.5B)
```

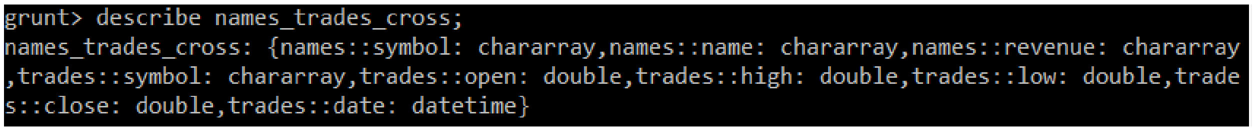


```
grunt> describe name_self;
name_self: {names::symbol: chararray,names::name: chararray,names::revenue: chararray,names1
::symbol: chararray,names1::name: chararray,names1::revenue: chararray}
```

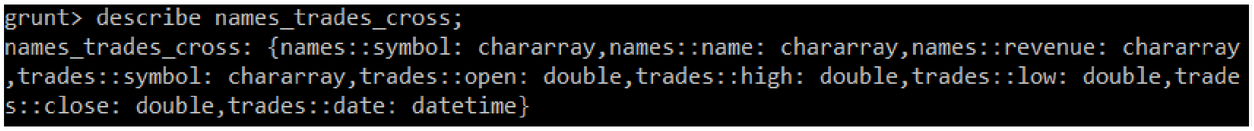
- Cross join



```
grunt> names_trades_cross = cross names, trades;
```



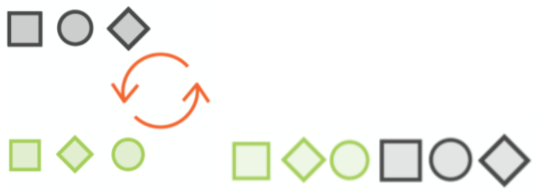
```
grunt> describe names_trades_cross;
names_trades_cross: {names::symbol: chararray,names::name: chararray,names::revenue: chararray
,trades::symbol: chararray,trades::open: double,trades::high: double,trades::low: double,trade
s::close: double,trades::date: datetime}
```



```
grunt> describe names_trades_cross;
names_trades_cross: {names::symbol: chararray,names::name: chararray,names::revenue: chararray
,trades::symbol: chararray,trades::open: double,trades::high: double,trades::low: double,trade
s::close: double,trades::date: datetime}
```

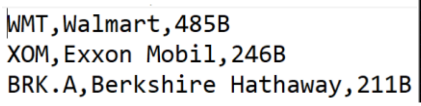
Unions in pig

- The relations involved in a union should have:
  - the same number of fields
  - compatible schema
- The result of a union is all tuples of the individual relations coming together in one relation.
- Does not preserve the order of tuples
- Preserves duplicates
  - If two tuples are identical, union will leave them both in the new relation.

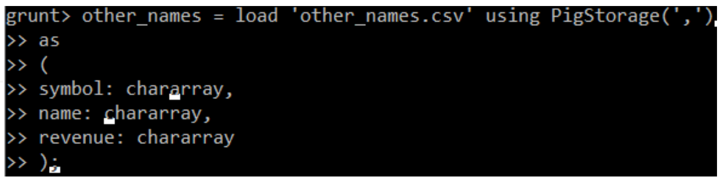


Demo

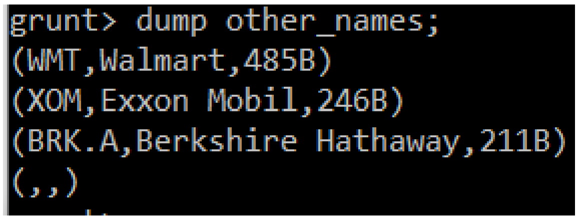
- Use names relation.
- Load another file other_names.csv



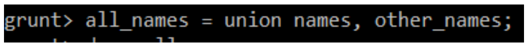
```
WMT,Walmart,485B
XOM,Exxon Mobil,246B
BRK.A,Berkshire Hathaway,211B
```



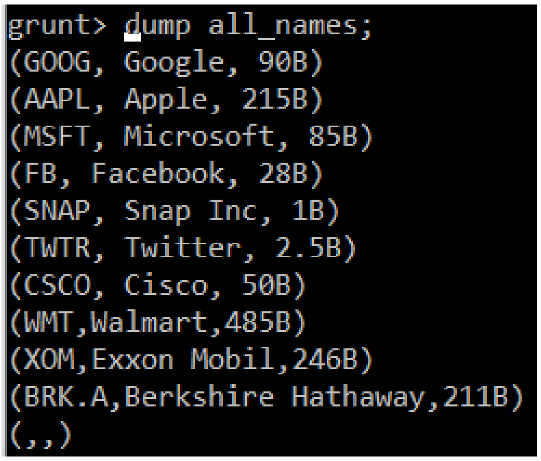
```
grunt> other_names = load 'other_names.csv' using PigStorage(',')
>> as
>> (
>> symbol: chararray,
>> name: chararray,
>> revenue: chararray
>> );
```



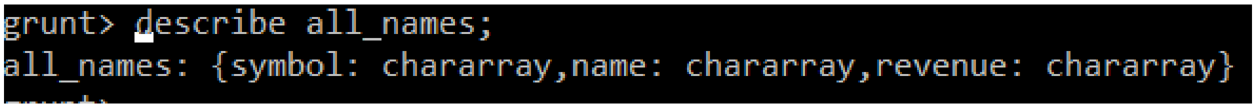
```
grunt> dump other_names;
(WMT,Walmart,485B)
(XOM,Exxon Mobil,246B)
(BRK.A,Berkshire Hathaway,211B)
(,,)
```



```
grunt> all_names = union names, other_names;
```



```
grunt> dump all_names;
(GOOG, Google, 90B)
(AAPL, Apple, 215B)
(MSFT, Microsoft, 85B)
(FB, Facebook, 28B)
(SNAP, Snap Inc, 1B)
(TWTR, Twitter, 2.5B)
(CSCO, Cisco, 50B)
(WMT,Walmart,485B)
(XOM,Exxon Mobil,246B)
(BRK.A,Berkshire Hathaway,211B)
(,,)
```



```
grunt> describe all_names;
all_names: {symbol: chararray,name: chararray,revenue: chararray}
```

Union with different schemas

₪ Union when schema is mismatched

```
R1: (a1: long, a2: long)
R2: (b1: long, b2: long, b3: long)
```

R1 union R2: null

₪ Union when schema types are not the same

```
R1: (a1: long, a2: long)
R2: (b1: (x: int, y: int), b2: long)
```

R1 union R2: (a1: bytearray, a2: long)

```
R1: (a1: long, a2: bytearray, a3: int)
R2: (b1: float, b2: chararray, b3: bytearray)
```

R1 union R2: (a1: float, a2: chararray, a3: int)

₪ Compatible Types Will Be Cast to Higher Type

δ double > float > long > int > bytearray

δ tuple | bag | map | chararray > bytearray

Union with different schemas

₪ Different inner types

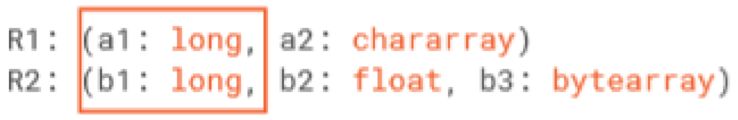
```
R1: (a1:(x:long, y:int), a2:{(n:float, m:chararray)})
R2: (b1:(g:chararray, h:float), b3:{(n:int, m:long)})
```

R1 union R2: (a1: (), a2: {()})

- The union may result in an empty complex type

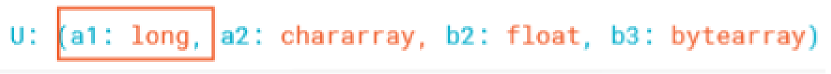
Union onschema for schema mismatches

- Union Onschema Combines Schemas



```
R1: (a1: long, a2: chararray)
R2: (b1: long, b2: float, b3: bytearray)
```

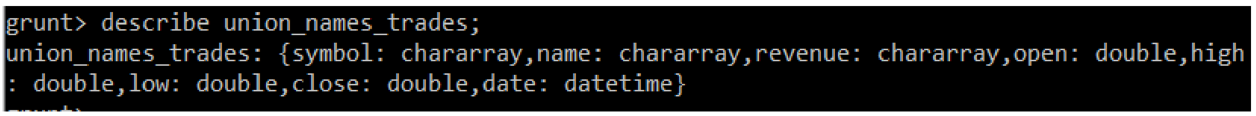
- union onschema R1, R2



```
U: (a1: long, a2: chararray, b2: float, b3: bytearray)
```

DEMO

- Use names and trades.
- They both have only one field in common i.e. symbol.

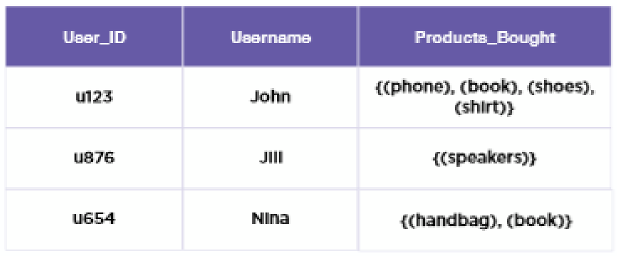


```
grunt> describe union_names_trades;
union_names_trades: {symbol: chararray,name: chararray,revenue: chararray,open: double,high
: double,low: double,close: double,date: datetime}
```

- The resultant relation has just one symbol field.
- It has combined the values that it found in the symbol field of names and of trades and then the remaining fields come from the relations.
- The first records come from the trades relation and the last records come from the names relation.

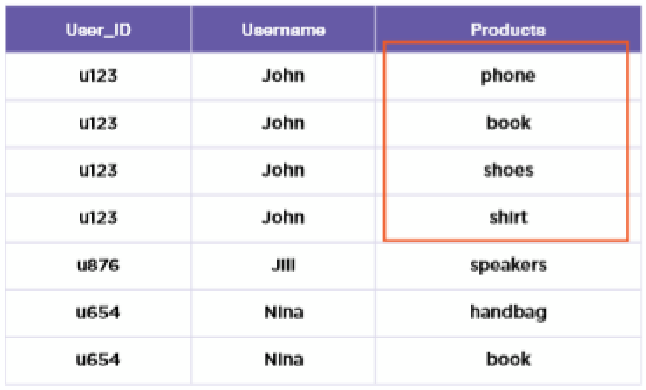
Flatten function

- It is a useful function used to extract entities from a bag as scalar fields.
- The flatten function is applied to a bag of tuples.
- Flattening a bag makes entity in the bag a separate record



User_ID	Username	Products_Bought
u123	John	{(phone), (book), (shoes), (shirt)}
u876	Jill	{(speakers)}
u654	Nina	{(handbag), (book)}

- The products each user has bought is specified as a bag.
- Flattening an empty bag results in null

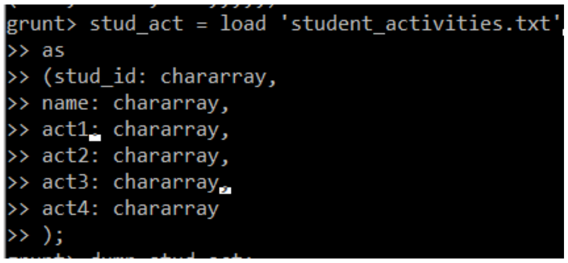


User_ID	Username	Products
u123	John	phone
u123	John	book
u123	John	shoes
u123	John	shirt
u876	Jill	speakers
u654	Nina	handbag
u654	Nina	book

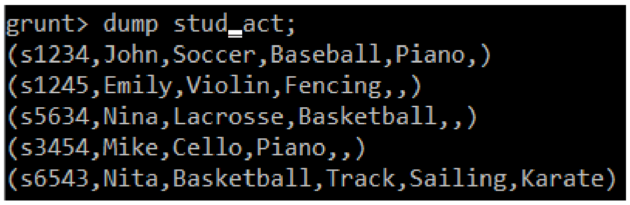
DEMO

₪ Use student_activities.txt

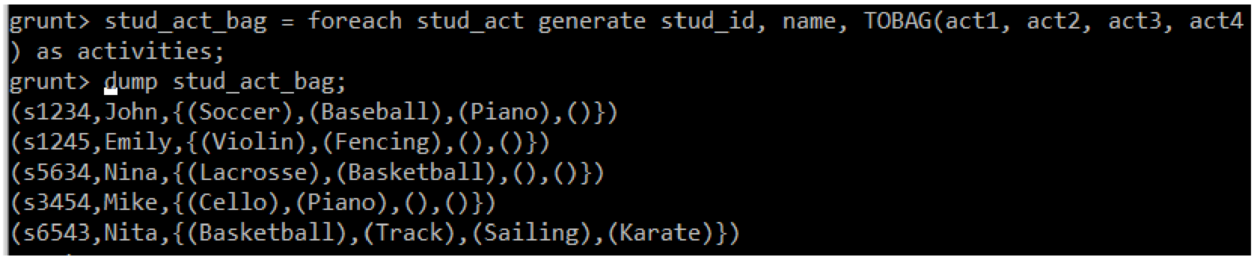
```
s1234   John    Soccer  Baseball        Piano
s1245   Emily   Violin  Fencing
s5634   Nina    Lacrosse        Basketball
s3454   Mike    Cello   Piano
s6543   Nita    Basketball      Track   Sailing Karate
```



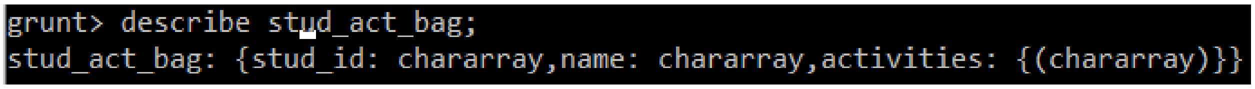
```
grunt> stud_act = load 'student_activities.txt'
>> as
>> (stud_id: chararray,
>> name: chararray,
>> act1: chararray,
>> act2: chararray,
>> act3: chararray,
>> act4: chararray
>> );
```



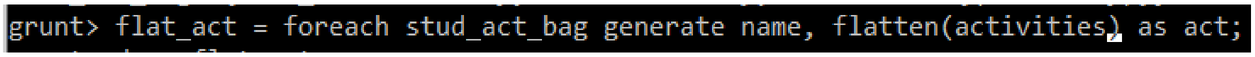
```
grunt> dump stud_act;
(s1234,John,Soccer,Baseball,Piano,)
(s1245,Emily,Violin,Fencing,,)
(s5634,Nina,Lacrosse,Basketball,,)
(s3454,Mike,Cello,Piano,,)
(s6543,Nita,Basketball,Track,Sailing,Karate)
```



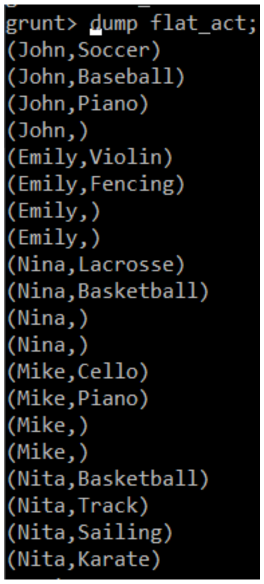
```
grunt> stud_act_bag = foreach stud_act generate stud_id, name, TOBAG(act1, act2, act3, act4
) as activities;
grunt> dump stud_act_bag;
(s1234,John,{(Soccer),(Baseball),(Piano),()})
(s1245,Emily,{(Violin),(Fencing),(),()})
(s5634,Nina,{(Lacrosse),(Basketball),(),()})
(s3454,Mike,{(Cello),(Piano),(),()})
(s6543,Nita,{(Basketball),(Track),(Sailing),(Karate)})
```



```
grunt> describe stud_act_bag;
stud_act_bag: {stud_id: chararray,name: chararray,activities: {(chararray)}}
```



```
grunt> flat_act = foreach stud_act_bag generate name, flatten(activities) as act;
```



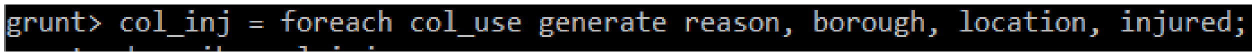
```
grunt> dump flat_act;
(John,Soccer)
(John,Baseball)
(John,Piano)
(John,)
(Emily,Violin)
(Emily,Fencing)
(Emily,)
(Emily,)
(Nina,Lacrosse)
(Nina,Basketball)
(Nina,)
(Nina,)
(Mike,Cello)
(Mike,Piano)
(Mike,)
(Mike,)
(Nita,Basketball)
(Nita,Track)
(Nita,Sailing)
(Nita,Karate)
```

The Nested Foreach Command

- Pig works on huge datasets where relations may hold millions of records
- The foreach keyword iterates through every record
- Multiple iterations will degrade performance
- Nested foreach combines multiple operations over the records of a dataset

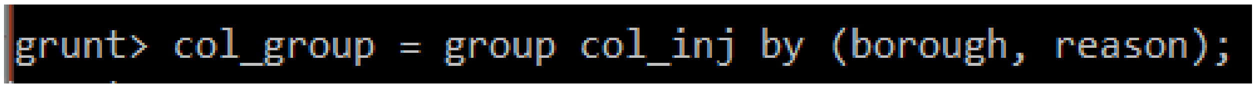
DEMO

- Use col_useful.

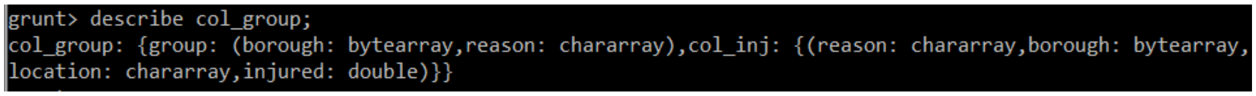


```
grunt> col_inj = foreach col_use generate reason, borough, location, injured;
```

- Calculate the number of collisions on a per-borough, per-reason basis.
    - Group this relation by two fields – borough, reason.

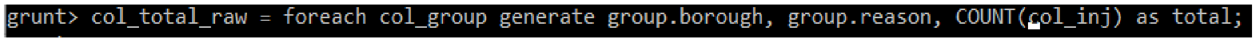


```
grunt> col_group = group col_inj by (borough, reason);
```

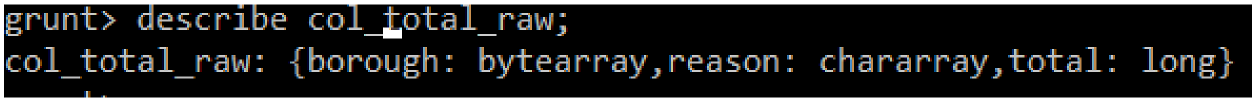


```
grunt> describe col_group;
col_group: {group: (borough: bytearray,reason: chararray),col_inj: {(reason: chararray,borough: bytearray,
location: chararray,injured: double)}}
```

- Count the number of tuples present in the bag.

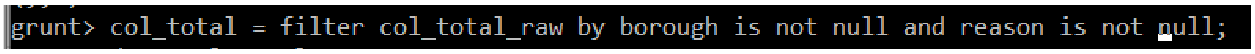


```
grunt> col_total_raw = foreach col_group generate group.borough, group.reason, COUNT(col_inj) as total;
```



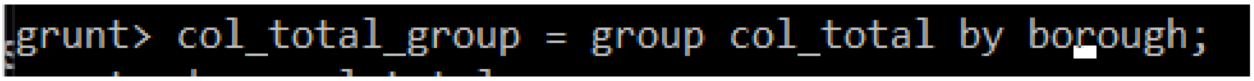
```
grunt> describe col_total_raw;
col_total_raw: {borough: bytearray,reason: chararray,total: long}
```

- When it is dumped, a lot of relations have null values.
- To remove records where either the borough or the reason is null



```
grunt> col_total = filter col_total_raw by borough is not null and reason is not null;
```

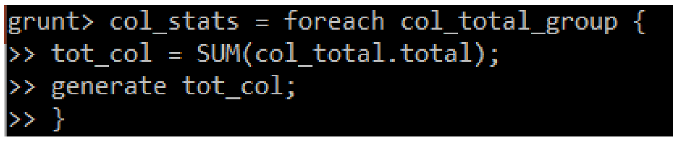
₪ Group the records by group.



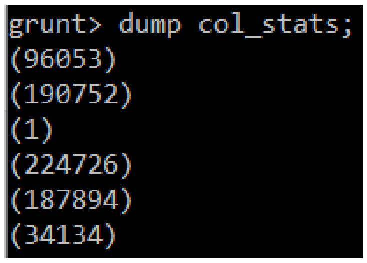
```
grunt> col_total_group = group col_total by borough;
```

₪ Calculate the number of collisions per borough and also the top reason for collisions in each borough.

₪ Use nested foreach.



```
grunt> col_stats = foreach col_total_group {
>> tot_col = SUM(col_total.total);
>> generate tot_col;
>> }
```

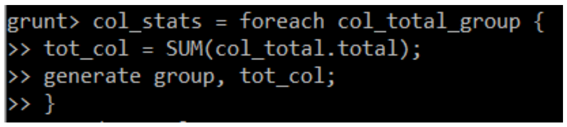


```
grunt> dump col_stats;
(96053)
(190752)
(1)
(224726)
(187894)
(34134)
```

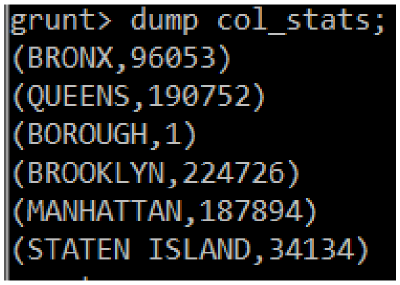
₪ Explanation

- δ Iterating over every record of the col_total_group which is grouped by borough.
- δ Calculate the total number of collisions and assign it to tot_col.
  - ∞ col_total is the name of the bag in the col_total_group
  - ∞ Total is the number of collisions on a per-borough per-reason basis.
- δ The last command should be a generate statement.

₪ To find the boroughs and the number



```
grunt> col_stats = foreach col_total_group {
>> tot_col = SUM(col_total.total);
>> generate group, tot_col;
>> }
```

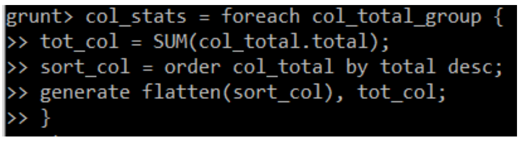


```
grunt> dump col_stats;
(BRONX,96053)
(QUEENS,190752)
(BOROUGH,1)
(BROOKLYN,224726)
(MANHATTAN,187894)
(STATEN ISLAND,34134)
```

- δ The group is the field that refers to the borough.

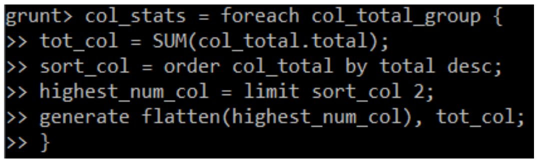
₪ Along with number of collisions, get the sorted order of collisions by reason.

- δ Access the bag that is associated with the borough and sort the tuples within the bag based on the total

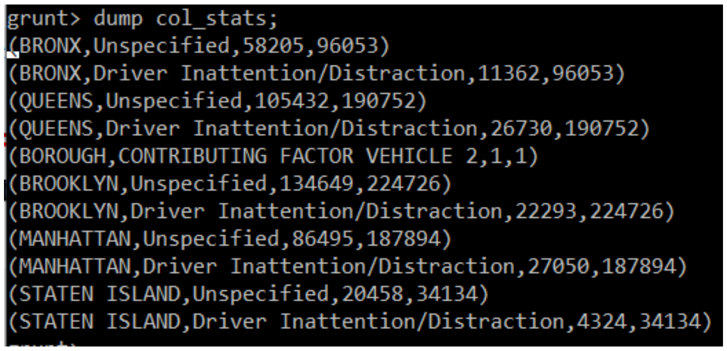


```
grunt> col_stats = foreach col_total_group {
>> tot_col = SUM(col_total.total);
>> sort_col = order col_total by total desc;
>> generate flatten(sort_col), tot_col;
>> }
```

- δ Limit the records per-borough, per-reason to highest 2.



```
grunt> col_stats = foreach col_total_group {
>> tot_col = SUM(col_total.total);
>> sort_col = order col_total by total desc;
>> highest_num_col = limit sort_col 2;
>> generate flatten(highest_num_col), tot_col;
>> }
```



```
grunt> dump col_stats;
(BRONX,Unspecified,58205,96053)
(BRONX,Driver Inattention/Distraction,11362,96053)
(QUEENS,Unspecified,105432,190752)
(QUEENS,Driver Inattention/Distraction,26730,190752)
(BOROUGH,CONTRIBUTING FACTOR VEHICLE 2,1,1)
(BROOKLYN,Unspecified,134649,224726)
(BROOKLYN,Driver Inattention/Distraction,22293,224726)
(MANHATTAN,Unspecified,86495,187894)
(MANHATTAN,Driver Inattention/Distraction,27050,187894)
(STATEN ISLAND,Unspecified,20458,34134)
(STATEN ISLAND,Driver Inattention/Distraction,4324,34134)
```

MapReduce using Pig

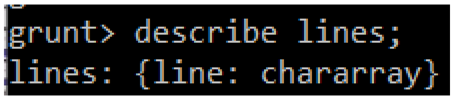
₪ Use words.txt.

₪ Use the following code to represent a mapreduce job.

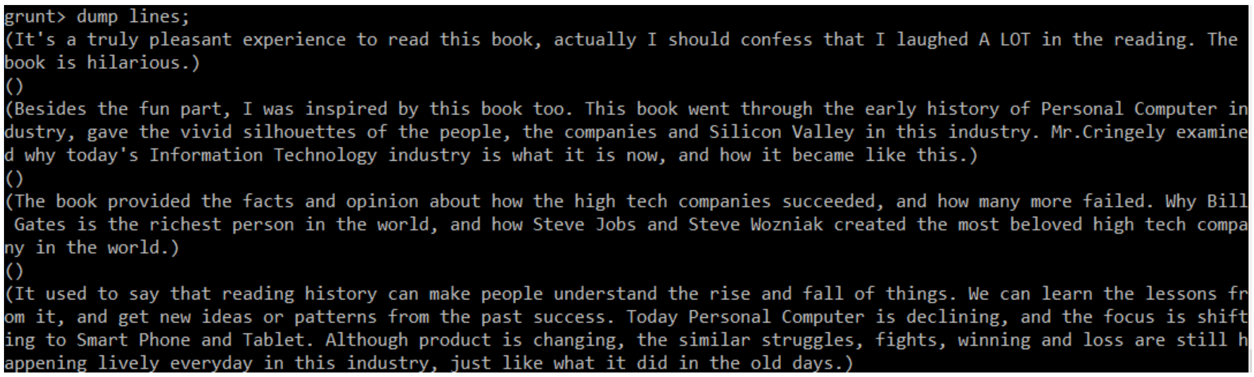
₪ **lines = load 'words.txt' as (line: chararray);**

₪ **describe lines;**

₪ **dump lines;**



```
grunt> describe lines;
lines: {line: chararray}
```



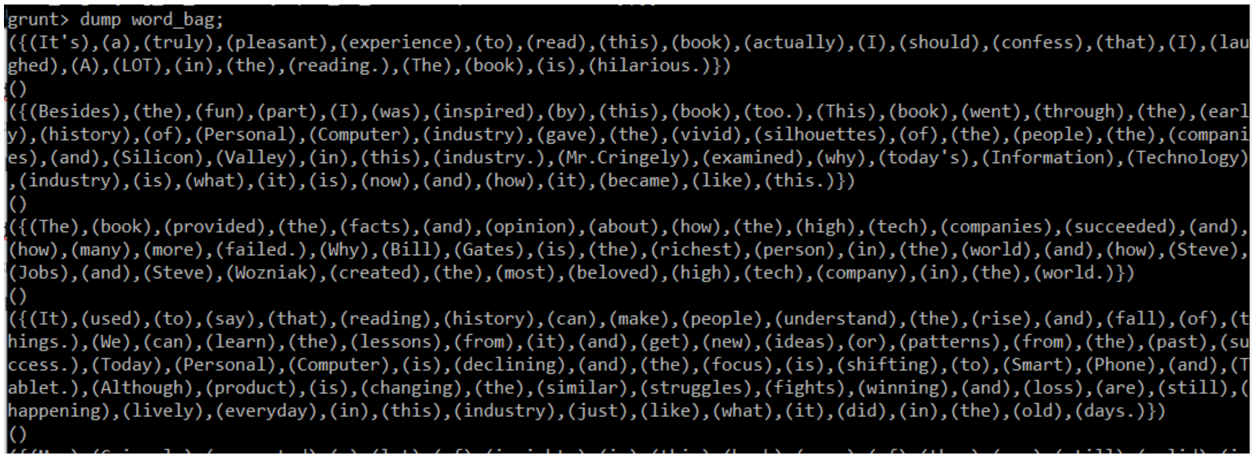
```
grunt> dump lines;
(It's a truly pleasant experience to read this book, actually I should confess that I laughed A LOT in the reading. The
book is hilarious.)
()
(Besides the fun part, I was inspired by this book too. This book went through the early history of Personal Computer in
dustry, gave the vivid silhouettes of the people, the companies and Silicon Valley in this industry. Mr.Cringely examine
d why today's Information Technology industry is what it is now, and how it became like this.)
()
(The book provided the facts and opinion about how the high tech companies succeeded, and how many more failed. Why Bill
 Gates is the richest person in the world, and how Steve Jobs and Steve Wozniak created the most beloved high tech compa
ny in the world.)
()
(It used to say that reading history can make people understand the rise and fall of things. We can learn the lessons fr
om it, and get new ideas or patterns from the past success. Today Personal Computer is declining, and the focus is shift
ing to Smart Phone and Tablet. Although product is changing, the similar struggles, fights, winning and loss are still h
appening lively everyday in this industry, just like what it did in the old days.)
```

- δ Lines contain s all lines of text in the form of a chararray.
- δ Every record is one line from the input file.
- δ Every line is separated from another by a new line.

₪ **word_bag = foreach lines generate TOKENIZE(lines) as bag_of_words;**

₪ **describe word_bag;**

₪ **dump word_bag;**



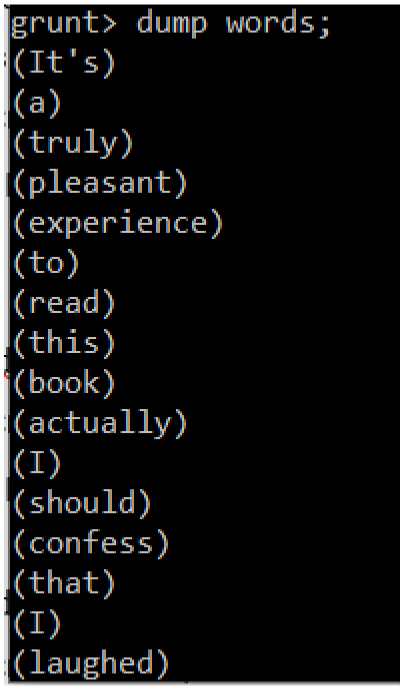
```
grunt> dump word_bag;
({(It's),(a),(truly),(pleasant),(experience),(to),(read),(this),(book),(actually),(I),(should),(confess),(that),(I),(lau
ghed),(A),(LOT),(in),(the),(reading.),(The),(book),(is),(hilarious.)})
()
({(Besides),(the),(fun),(part),(I),(was),(inspired),(by),(this),(book),(too.),(This),(book),(went),(through),(the),(earl
y),(history),(of),(Personal),(Computer),(industry),(gave),(the),(vivid),(silhouettes),(of),(the),(people),(the),(compani
es),(and),(Silicon),(Valley),(in),(this),(industry.),(Mr.Cringely),(examined),(why),(today's),(Information),(Technology)
,(industry),(is),(what),(it),(is),(now),(and),(how),(it),(became),(like),(this.)})
()
({(The),(book),(provided),(the),(facts),(and),(opinion),(about),(how),(the),(high),(tech),(companies),(succeeded),(and),
(how),(many),(more),(failed.),(Why),(Bill),(Gates),(is),(the),(richest),(person),(in),(the),(world),(and),(how),(Steve),
(Jobs),(and),(Steve),(Wozniak),(created),(the),(most),(beloved),(high),(tech),(company),(in),(the),(world.)})
()
({(It),(used),(to),(say),(that),(reading),(history),(can),(make),(people),(understand),(the),(rise),(and),(fall),(of),(t
hings.),(We),(can),(learn),(the),(lessons),(from),(it),(and),(get),(new),(ideas),(or),(patterns),(from),(the),(past),(su
ccess.),(Today),(Personal),(Computer),(is),(declining),(and),(the),(focus),(is),(shifting),(to),(Smart),(Phone),(and),(T
ablet.),(Although),(product),(is),(changing),(the),(similar),(struggles),(fights),(winning),(and),(loss),(are),(still),(
happening),(lively),(everyday),(in),(this),(industry),(just),(like),(what),(it),(did),(in),(the),(old),(days.)})
()
```

- δ Split every line into its individual words.
- δ TOKENIZE function takes the standard word delimiter – space.
  - ∞ Every input record on which the iteration happens using the foreach is split into a bag of words.
  - ∞ The schema will show a bag named bag_of_words within which there is a tuple_of_tokens and each tuple has one field of type chararray whose name is token.
- δ Each line is a bag of words.
- δ There are multiple bags in the relation.

₪ **words = foreach word_bag generate flatten(bag_of_words) as word;**

₪ **describe words;**

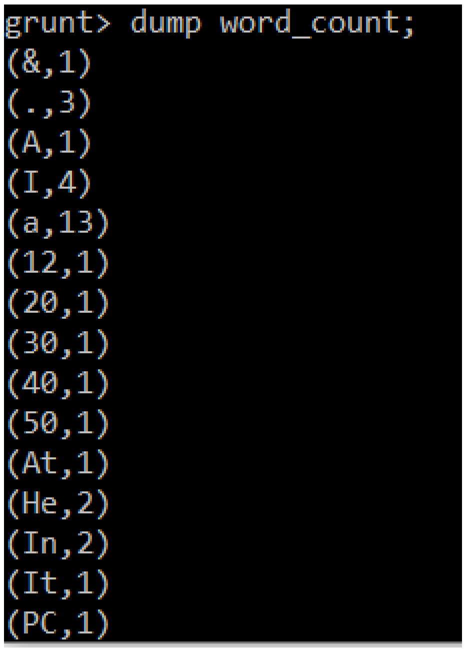
₪ **dump words;**



```
grunt> dump words;
(It's)
(a)
(truly)
(pleasant)
(experience)
(to)
(read)
(this)
(book)
(actually)
(I)
(should)
(confess)
(that)
(I)
(laughed)
```

- Every individual word is a separate record.
- The field name for the word is word.
- The schema of the resultant is one field – word that is of the type chararray.
  - Sometimes the words have punctuations with it.
    - ⇒ This is because TOKENIZE has tokenized on the space character.

- **word_group = group words by word;**
- **describe word_group;**
- **dump word_group;**
  - Group the entire relation by a word to count the number of words.
  - The group key is a word, and the bag that's associated with it contains a bag of that same word.
- **word_count = foreach word_group generate group, COUNT(words);**
- **describe word_count;**
- **dump word_count;**



```
grunt> dump word_count;
(&,1)
(.,3)
(A,1)
(I,4)
(a,13)
(12,1)
(20,1)
(30,1)
(40,1)
(50,1)
(At,1)
(He,2)
(In,2)
(It,1)
(PC,1)
```

- Iterate through every record and count the number of words that are present in the bag.
- Generate the group that is the word itself and an associated count.
- The schema is basically a group of type chararray – word and long – word count.

HIVE

Transactional and Analytical Processing

Transactional Processing	Analytical Processing
Analyzes individual entries	Analyzes large batches of data
Access to recent data, from the last few hours or days	Access to older data going back months, or even years
Updates data	Only reads data
Updates data Fast real-time access	Long running jobs
Usually a single data source	Multiple data sources

Small Data

- Both the objects could be achieved using the same database system.
- Single machine with backup
- Structured, well-defined data
- Can access individual records or the entire dataset
- No replication, updated data available instantaneously
- Different tables store data from different sources

Big Data

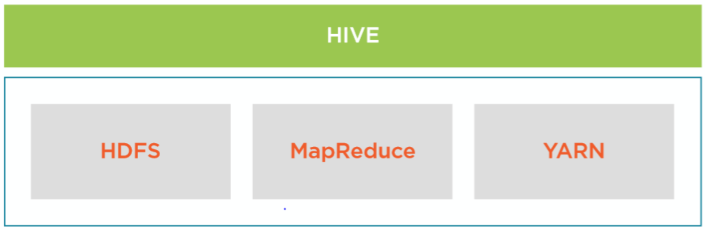
- Very hard to meet all requirements with the same database system
- Data distributed on a cluster with multiple machines
- Semi-structured or unstructured data
- No random access to data
- Data replicated, propagation of updates take time
- Different sources may have different unknown formats

Data Warehouse

₪ A technology that aggregates data from one or more sources so that it can be compared and analyzed for greater business intelligence.
₪ Long running batch jobs
₪ Optimized for read operations
₪ Holds data from multiple sources
₪ Holds data over a long period of time
₪ Data may be lagged, not real-time
₪ Examples of data warehouses – Vertica, Teradata, Oracle, IBM

Apache Hive

₪ It is an open-source data warehouse.
₪ Hive is part of the larger Hadoop ecosystem.
₪ Hive runs on top of the Hadoop distributed computing framework



₪ Hive stores its data in HDFS
₪ Hadoop Distributed File System
  δ Data is stored as files - text files, binary files
  δ Partitioned across machines in the cluster
  δ Replicated for fault tolerance
  δ Processing tasks parallelized across multiple machines
₪ Hive runs all processes in the form of MapReduce jobs under the hood
₪ MapReduce
  δ A parallel programming model
  δ Defines the logic to process data on multiple machines
  δ Batch processing operations on files in HDFS
  δ Usually written in Java using the Hadoop MapReduce library

Hive QL

₪ Hive Query Language

₪ A SQL-like interface to the underlying data.

₪ Modeled on the Structured Query Language (SQL)

₪ Familiar to analysts and engineers

₪ Has simple query constructs

 δ select

 δ group by

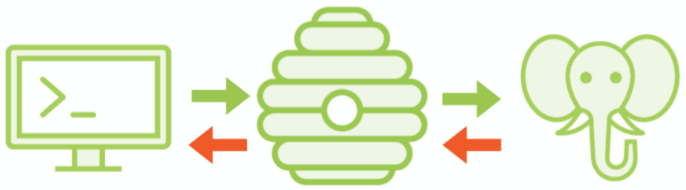
 δ join

₪ Hive exposes files in HDFS in the form of tables to the user

₪ Write SQL-like query in HiveQL and submit it to Hive

₪ Hive will translate the query to MapReduce tasks and run them on Hadoop

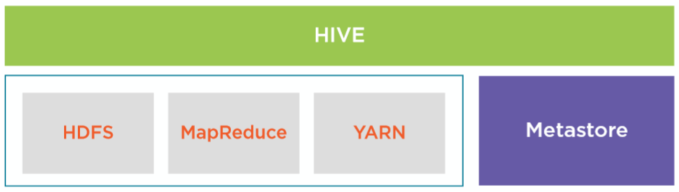
₪ MapReduce will process files on HDFS and return results to Hive



Hive Metastore

₪ A Hive user sees data as if they were stored in tables

₪ Exposes the file-based storage of HDFS in the form of tables



₪ It is the bridge between data stored in files and the tables exposed to users

₪ Stores metadata for all the tables in Hive

₪ Maps the files and directories in Hive to tables

₪ Holds table definitions and the schema for each table

₪ Has information on converting files to table representations

₪ Any database with a JDBC driver can be used as a metastore

₪ Development environments use the built-in Derby database i.e. Embedded metastore

HIVE vs. RDBMS

HIVE	RDBMS
Large datasets	Small datasets
Parallel computations	Serial computations
High latency	Low latency
Read operations	Read/write operations
Not ACID compliant by default	ACID compliant
HiveQL	SQL

- Large datasets
    - Gigabytes or petabytes
    - Calculating Trends
- Small datasets
    - Megabytes or gigabytes
    - Accessing and updating individual records
- Parallel Computations
    - Distributed system with multiple machines
    - Semi-structured data files partitioned across machines
    - Disk space cheap, can add space by adding machines
- Serial Computations
    - Single computer with backup
    - Structured data in tables on one machine
    - Disk space expensive on a single machine
- High Latency
    - Records not indexed, cannot be accessed quickly
    - Fetching a row will run a MapReduce that might take minutes
- Low Latency
    - Records indexed, can be accessed and updated fast
    - Queries can be answered in milliseconds or microseconds
- Read Operations
    - Not the owner of data.
        - Hive stores files in HDFS

- ∞ Hive files can be read and written by many technologies
  - ⇒ Hadoop, Pig, Spark
- ∞ Hive database schema cannot be enforced on these files

- δ Schema-on-read
  - ∞ Number of columns, column types, constraints specified at table creation
  - ∞ Hive tries to impose this schema when data is read
  - ∞ It may not succeed, may pad data with nulls
- δ Read/ Write Operations
  - ∞ Sole gatekeeper for data
  - ∞ Schema on write
- δ Not ACID Compliant
  - ∞ Data can be dumped into Hive tables from any source
- δ ACID Compliant
  - ∞ Only data which satisfies constraints are stored in the database

HIVEQL	SQL
Schema on read, no constraints enforced	Schema on write keys, not null, unique all enforced
Minimal index support	Indexes allowed
Row level updates, deletes as a special case	Row level operations allowed in general
Many more built-in functions	Basic built-in functions
Only equi-joins allowed	No restriction on joins
Restricted subqueries	Whole range of subqueries