**Hadoop Developer Training – Hive Lab Book**

**Table of Contents**

[Lab 1: Understanding Hive Tables 3](#_Toc452037494)

[Lab 2: Analyzing Big Data with Hive 10](#_Toc452037495)

[Lab 3: Understanding MapReduce in Hive 17](#_Toc452037496)

[Lab 4: Joining Datasets in Hive 21](#_Toc452037497)

[Lab 5: Computing ngrams of Emails in Avro Format 25](#_Toc452037498)

[Lab 7: Using HCatalog with Pig 32](#_Toc452037499)

[Lab 8 : Advanced Hive Programming 37](#_Toc452037500)

[Lab 9 : Streaming Data with Hive and Python 48](#_Toc452037501)

|  |  |
| --- | --- |
| **Location of Files:** | **/<*userpath*>/labs/** |

**Note: Please use the below paths to avoid error while practicing**

1. **<*userpath*> should be your working directory(user home directory)**
2. **for Cloudera HDFS path is :** user/training/
3. **for Hartonworks HDFS path is :** user/root/

# Lab 1: Understanding Hive Tables

**Perform  the  following**

Step  1: Review  the  Data

**1.1.** Use  the  **ls**  command  to  view  the  contents  of  the  **wh\_visits**  folder.  You  should

see  6  **part-­‐m**  files:

# hadoop fs -ls /apps/hive/warehouse/wh\_visits/

**1.2.** Recall  that  the  Pig  projection  to  create  these  files  had  the  following  schema:

project\_potus = FOREACH potus GENERATE

$0 AS lname:chararray,

$1 AS fname:chararray,

$6 AS time\_of\_arrival:chararray,

$11 AS appt\_scheduled\_time:chararray,

$21 AS location:chararray,

$25 AS comment:chararray ;

In  this  lab,  you  will  define  Hive  table  that  matches  these  records  and  contains

the  exported  data  from  your  Pig  script.

**Step  2**: Define  a  Hive  Script

**2.1.** In  the  **Lab7.1**  folder,  there  is  a  text  file  named  **wh\_visits.hive**.    View  its  contents.  Notice  it  defines  a  Hive  table  named  **wh\_visits**  with  the  following  schema  that  matches  the  data  in  your  **project\_potus**  folder:

# more wh\_visits.hive

create table wh\_visits (

lname string, fname string, time\_of\_arrival string, appt\_scheduled\_time string, meeting\_location string, info\_comment string)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t' ;

create table bd\_sample (

empid int,

fname string,

lname string,

age int)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\t' ;

**NOTE**:  You  cannot  use  **comment**  or  **location**  as  column  names  because

those  are  reserved  Hive  keywords,  so  we  changed  them  slightly.

**2.2.** Run  the  script  with  the  following  command:

# hive -f wh\_visits.hive

**2.3.** If  successful,  you  should  see  “OK”  in  the  output  along  with  the  time  it  took  to

run  the  query.

**Step  3**: Verify  the  Table  Creation

**3.1.** Start  the  Hive  Shell:

# hive hive>

**3.2.** From  the  **hive>**  prompt,  enter  the  “**show  tables**”  command:

hive> show tables;

You  should  see  **wh\_visits**  in  the  list  of  tables.

|  |  |  |
| --- | --- | --- |
| OK |  | |
| lname | string | None |
| fname | string | None |
| time\_of\_arrival | string | None |
| appt\_scheduled\_time | string | None |
| meeting\_location | string | None |
| info\_comment | string | None |

**3.3.** Use  the  **describe**  command  to  view  the  details  of  **wh\_visits**:

hive> describe wh\_visits;

**3.4.** Try  running  a  query  (even  though  the  table  is  empty):

select \* from wh\_visits limit 20;

You  should  see  20  rows  returned.  How  is  this  brand  new  Hive  table  already populated  with  records?

**3.5.** Why  did  the  previous  query  not  require  a  MapReduce  job  to  execute?

**3.6.** Try  the  following  query.  Make  sure  the  output  looks  like  first  names:

hive> select fname from wh\_visits limit 20;

This  time  a  MapReduce  job  executed.  Why?

**Step  4:** Count  the  Number  of  Rows  in  a  Table

**4.1.** Enter  the  following  Hive  query,  which  outputs  the  number  of  rows  in

**wh\_visits**:

hive> select count(\*) from wh\_visits;

**4.2.** How  many  rows  are  currently  in  **wh\_visits**?

**Step  5**: Selecting the  Input  File  Name

**5.1.** Hive  has  two  virtual  columns  that  gets  created  automatically  for  every  table:  **INPUT** **FILE** **NAME**  and  **BLOCK** **OFFSET** **INSIDE** **FILE**.  You  can  use  these  column  names  in  your  queries  just  like  any  other  column  of  the  table. To  demonstrate,  run  the  following  query:

hive> select INPUT\_\_FILE\_\_NAME, lname, fname FROM wh\_visits

WHERE lname LIKE 'Y%';

**5.2.** The  result  of  this  query  is  visitors  to  the  White  House  whose  last  name  starts  with “Y”.  Notice  that  the  output  also  contains  the  particular  file  that  the  record  was  found  in:

hdfs://sandbox.hortonworks.com:8020/apps/hive/warehouse/wh\_ visits/part-m-00001 YOUNG LEDISI hdfs://sandbox.hortonworks.com:8020/apps/hive/warehouse/wh\_ visits/part-m-00002 YARNOLD DAVID

**Step  6:** Drop  a  Table

**6.1.** Let’s  see  what  happens  when  a  managed  table  is  dropped.  Start  by  defining  a

simple  table  called  **names**  using  the  Hive  Shell:

hive> create table names (id int, name string)

> ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

**6.2.** Use  the  Hive  **dfs**  command  to  put  **Lab7.1/names.txt**  into  the  table’s

warehouse  folder:

hive> dfs -put /root/labs/Lab7.1/names.txt

/apps/hive/warehouse/names/;

**6.3.** View  the  contents  of  the  table’s  warehouse  folder:

hive> dfs -ls /apps/hive/warehouse/names;

Found 1 items

root hdfs 78 /apps/hive/warehouse/names/names.txt

**6.4.** From  the  Hive  Shell,  run  the  following  query:

hive> select \* from names

; OK

0 Rich

1 Barry

2 George

3 Ulf

4 Danielle

5 Tom

6 manish

7 Brian

8 Mark

**6.5.** Now  drop  the  **names**  table:

hive> drop table names;

**6.6.** View  the  contents  of  the  table’s  warehouse  folder  again.  Notice  the  **names**

folder  is  gone:

hive> dfs -ls /apps/hive/warehouse/names;

ls: '/apps/hive/warehouse/names': No such file or directory

**IMPORTANT**:  Be  careful  when  you  drop  a  managed  table  in  Hive.  Make  sure  you  either  have  the  data  backed  up  somewhere  else,  or  that  you  no  longer  want  the  data.

**Step  7:** Define an  External  Table

**7.1.** In this  step  you  will  see  how  external  tables  work  in  Hive.  Start  by  putting

**names.txt**  into  HDFS:

hive> dfs -put /root/labs/Lab7.1/names.txt names.txt;

**7.2.** Create  a  folder  in  HDFS  for  the  external  table  to  store  its  data  in:

hive> dfs -mkdir hivedemo;

**7.3.** Define  the  **names**  table  as  external  this  time:

hive> create external table names (id int, name string)

> ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'

> LOCATION '/user/root/hivedemo';

**7.4.** Load  data  into  the  table:

hive> load data inpath '/user/root/names.txt' into table names;

load data inpath '/user/root/jaspreet.txt' into table bd\_sample;

**7.5.** Verify  the  load  worked:

hive> select \* from names;

**7.6.** Notice  the  **names.txt**file  has  been  moved  to  **/user/root/hivedemo**:

hive> dfs -ls hivedemo; Found 1 items

-rw-r--r-- 1 root hdfs 78 hivedemo/names.txt

**7.7.**Similarly,  verify  names.txt  is  no  longer  in  your  /user/root  folder  in  HDFS.  Why  is  it  gone?

**7.8.** Use  the  **ls**  command  to  verify  that  the  **/apps/hive/warehouse**  folder  does  not

contain  a  subfolder  for  the  **names**  table.

**7.9.** Now  drop  the  **names**  table:

hive> drop table names;

**7.10.** View  the  contents  of  **/user/root/hivedemo**.  Notice  that  **names.txt**  is  still   there.

**RESULT**:  As  you  just  verified,  the  data  for  external  tables  is  not  deleted  when  the  corresponding  table  is  dropped.  Aside  from  this  behavior,  managed  tables  and  external  tables  in  Hive  are  essentially  the  same.  You  now  have  a  table  in  Hive  named  **wh\_visits**  that  was  loaded  from  the  result  of  a  Pig  job.  You  also  have  an  external  table  called  **names**  that  stores  its  data  in  **/user/root/hivedemo**.  At  this  point,  you  should  have  a  good  understanding  of  how  Hive  tables  are  created  and  populated.

# Lab 2: Analyzing Big Data with Hive

***Perform  the  following  steps:***

**Step  1:**

**1.1**. Create a  new text file named  whitehouse.hive and save  it  in  your Lab7.2 folder.

**1.2.** In  this  step,  you  will  find  the  first  visitor  to  the  White  House  (based  on  our  dataset).  This  will  involve  some  clever  handling  of  timestamps.  This  will  be  a  long  query,  so  enter  it  on  multiple  lines.  Start  by  selecting  all  columns  where  the  **time\_of\_arrival**  is  not  empty:

select \* from wh\_visits where time\_of\_arrival != ""

**1.3.** To  find  the  first  visit,  we  need  to  sort  the  result.  This  requires  converting  the  **time\_of\_arrival**string  into  a  timestamp.  We  will  use  the  **unix\_timestamp**  function  to  accomplish  this.  Add  the  following  **order  by**  clause:

order by unix\_timestamp(time\_of\_arrival,

'MM/dd/yyyy hh:mm')

**1.4.** Since  we  are  only  looking  for  one  result,  we  certainly  don’t  need  to  return

every  row.  Let’s  limit  the  result  to  10  rows,  so  we  can  view  the  first  10  visitors:

limit 10;

**1.5.** Save  your  changes  to  **whitehouse.hive**.

**1.6.** Execute the  script  **whitehouse.hive**  and  wait  for  the  results  to  be  displayed:

# hive -f whitehouse.hive

**1.7.** The  results  should  be  10  visitors,  and  the  first  visit  should  be  in  2009  since

that  is  when  the  dataset  begins.  The  first  visitor  is  Charles  Kahn  on  3/5/2009.

**Step**2: Find the  Last  Visit

**2.1.** This  one  is  easy  -­‐  just  take  the  previous  query  and  reverse  the  order  by  adding

**desc**  to  the  **order  by**  clause:

order by unix\_timestamp(time\_of\_arrival,

'MM/dd/yyyy hh:mm') **desc**

**2.2.** Run  the  query  again,  and  you  should  see  that  the  most  recent  visit  was  Jackie   Walker  on  3/18/2011.

**Step 3:** Find the Most Common Comment

**3.1.** In this  step,  you  will  explore  the  **info\_comment**  field  and  try  to  determine  the  most  common  comment.  You  will  use  some  of  Hive’s  aggregate  functions  to  accomplish  this.  Start  by  creating  a  new  text  file  named  **comments.hive**.

**3.2.** You  will  now  create  a  query  that  displays  the  10  most  frequently  occurring

comments.  Start  with  the  following  select  clause:

from wh\_visits

select count(\*) as comment\_count, info\_comment

This  runs  the  aggregate  **count**  function  on  each  group  (which  you  will  define  later  in  the  query)  and  names  the  result  **comment\_count**.  For  example,  if  “OPEN  HOUSE”  occurs  5  times,  then  **comment\_count**  will  be  5  for  that  group.

Notice  we  are  also  selecting  the  **info\_comment**  column  so  we  can  see  what  the

comment  is.

**3.3.** Group  the  results  of  the  query  by  the **info\_comment**  column:

group by info\_comment

**3.4.** Order  the  results  by  **comment\_count**,  because  we  are  only  interested  in

comments  that  appear  most  frequently:

order by comment\_count DESC

**3.5.** We  only  want  the  top  results,  so  limit  the  result  set  to  10:

limit 10;

**3.6.** Save  your  changes  to  **comments.hive**  and  execute  the  script.  Wait  for  the   MapReduce  job  to  execute.

**3.7.** The  output  will  be  10  comments  and  should  look  like:

9036

1253 HOLIDAY BALL ATTENDEES/

894 WHO EOP RECEP 2

700 WHO EOP 1 RECEPTION/

601 RESIDENCE STAFF HOLIDAY RECEPTION/

586 PRESS RECEPTION ONE (1)/

580 GENERAL RECEPTION 1

540 HANUKKAH RECEPTION./

540 GEN RECEP 5/

**3.8.** It  appears  that  a  blank  commentis  the  most  frequent  comment,  followed  by  the  HOLIDAY  BALL,  then a variation  of  other  receptions.

**3.9. OPTIONAL**:  Modify  the  query  so  that  it  ignores  empty  comments.  If it  works,

the comment  “GEN  RECEP  6/”  will  show  up  in  your  output.

**Step  4:** Least  Frequent  Comment

**4.1.** Run  the  previous  query  again,  but  this  time  find  the  10  least  occurring

comments.  The  output  should  look  like:

1 merged to u59031

1 WHO EOP/

1 WHO EOP RECLEAR

1 WAITING FOR SUPERMAN VISIT

1 ST. PATRICK'S RECEPTION GUESTS

1 SCIENCE FAIR

1 RES PARTY/

1 PRIVATE MEETING

1 PRIVATE LUNCH

1 POTUS PHOTO W/ US ATTORNEYS/

This  seems  accurate  since  1  is  the  least  number  of  times a  comment  can  appear.  Plus  this  query  reveals  that  Superman  has  visited  the  President at least once!

**Step 5:** Analyze the  Data Inconsistencies

**5.1.** Analyzing the  results  of  the  most  and  least  frequent  comments,  it  appears  that  several  variations  of  GENERAL  RECEPTION  occur.  In  this  step,  you  will  try  to  determine  the  number  of  visits  to  the  POTUS  involving  a  general  reception  by  trying  to  clean  up  some  of  these  inconsistencies  in  the  data.

**NOTE**:  Inconsistencies  like  these  are  very  common  in  big  data,  especially  when  human  input  is  involved.  In  this  dataset,  we  like  have  different  people  entering  similar

comments  but  using  their  own  abbreviations.

**5.2.** Modify  the  query  in  **comments.hive**.  Instead  of  searching  for  empty  comments,   search  for  comments that contain  the  string  “RECEP”.

where info\_comment like "%RECEP%"

**5.3.** Change  the  limit  clause  from  10  to  30:

limit 30;

**5.4.** Run  the  query  again.

**5.5.** Notice  there  are  several  GENERAL  RECEPTION  entries  that  only  differ  by  a

number  at  the  end,  or  use  the  GEN  RECEP  abbreviation:

|  |  |  |
| --- | --- | --- |
| 580 | GENERAL RECEPTION | 1 |
| 540 | GEN RECEP 5/ |  |
| 516 | GENERAL RECEPTION | 3 |
| 498 | GEN RECEP 6/ |  |
| 438 | GEN RECEP 4 |  |
| 31 | GENERAL RECEPTION | 2 |
| 23 | GENERAL RECEPTION | 3 |
| 20 | GENERAL RECEPTION | 6 |
| 20 | GENERAL RECEPTION | 5 |
| 13 | GENERAL RECEPTION | 1 |

**5.6.** Let’s try  one  more  query  to  try  and  narrow  GENERAL  RECEPTION  visist.  Modify  the  WHERE  clause  in  comments.hive  to  include  “%GEN%”:  where info\_comment like "%RECEP%" and info\_comment like "%GEN%"

**5.7.** Leave the  limit at  30,  and  run  the  query  again.

**5.8.** The output this  time  reveals  all  the  variations  of  GEN  and  RECEP.  Let’s  add  up

the total  number  of  them  by  running  the  following  query:

from wh\_visits

select count(\*)

where info\_comment like "%RECEP%" and info\_comment like "%GEN%";

**5.9.** Notice  there  are  2,697  visits  to  the  POTUS  with  GEN  RECEP  in  the  comment

field,  which  is  about  12%  of  the  21,819  total  visits  to  the  POTUS  in  our  dataset.

**NOTE**:  More  importantly,  these  results  show  that  our  first  query  of  1,253  attendees  to  the  HOLIDAY  BALL  does  not  mean  that  the  holiday  ball  is  the  most  likely  reason  to  visit  the  President.  More  than  twice  as  many  visitors  are  there for a  general  reception.  This  type  of  analysis  is  common  in  big  data,  and  it  shows  how  Data  Analysts  need  to  be  creative  when  researching  their  data.

**Step  6:** Verify the  Result

**6.1.** We  have  12%  of  visitors  to  the  POTUS  going  for  a  general  reception,  but  there  were  a  lot  of  statements  in  the  comments  that  contained  WHO  and  EOP.  Modify  the  query  from  the  last  step  and  display  the  top  30  comments  that  contain

“WHO”  and  “EOP”.  The  result  should  look  like:

|  |  |  |  |
| --- | --- | --- | --- |
| 894 | WHO | EOP | RECEP 2 |
| 700 | WHO | EOP | 1 RECEPTION/ | |
| 43 | WHO | EOP | RECEP/ |
| 20 | WHO | EOP | HOLIDAY RECEP/ |
| 13 WHO/EOP #2/ | | | |
| 8 | WHO | EOP RECEPTION | |
| 7 | WHO | EOP RECEP | |
| 1 | WHO | EOP/ | |
| 1 | WHO | EOP RECLEAR | |

**6.2.** Run  a  query  that counts  the  number  of  records  with  WHO  and  EOP  in  the

comments:

from wh\_visits select count(\*) where info\_comment like "%WHO%" and info\_comment like "%EOP%";

You  should  get  1,687  visits,  or  7.7%  of  the  visitors  to  the  POTUS.  So  GENERAL   RECEPTION  still  appears  to  be  the  most  frequent  comment.

**Step  7:** Find  the  Most  Visits

**7.1.** See  if  you  can  write  a  Hive  script  that  finds  the  top  20  individuals  who  visited  the  POTUS  most.  Use  the  Hive  command  from  Step  3  earlier  in  this  lab  as a guide.  **HINT**:  Use  a grouping  by  both  **fname**  and  **lname**.

**7.2.** To  verify  your  script  worked,  here  are  the  top  20  individuals  who  visited  the  POTUS,  along  with  the  number  of  visits:

|  |  |  |  |
| --- | --- | --- | --- |
|  | 16 | ALAN PRATHER |  |
| 15 | CHRISTOPHER | FRANKE |
| 15 | ANNAMARIA MOTTOLA | |
| 14 | ROBERT BOGUSLAW | |
| 14 | CHARLES POWERS | |
| 12 | SARAH HART | |
| 12 | JACKIE WALKER | |
| 12 | JASON FETTIG | |
| 12 | SHENGTSUNG WANG | |
| 12 | FERN SATO | |
| 12 | DIANA FISH | |
| 11 | JANET BAILEY | |
| 11 | PETER WILSON | |
| 11 | GLENN DEWEY | |
| 11 | MARCIO BOTELHO | |
| 11 | DONNA WILLINGHAM | |
| 10 | DAVID AXELROD | |
| 10 | CLAUDIA CHUDACOFF | |
| 10 | VALERIE JARRETT | |
| 10 | MICHAEL COLBURN | |
|  |  |  | |

**RESULT**:  You  have  written  several  Hive  queries  to  analyze  the  White  House  visitor  data.  The  goal  is  for  you  to  become  comfortable  with  working  with  Hive,  so  hopefully you  now  feel  like  you  can  tackle  a  Hive  problem  and  be  able  to  answer  questions  about  your  big  data  stored  in  Hive.

# Lab 3: Understanding MapReduce in Hive

**Step  1:** The  Describe  Command

1.1. Run  the  describe  command  on  the  **wh\_visits**  table:

hive> describe wh\_visits;

OK

lname string None

fname string None

time\_of\_arrival string None

appt\_scheduled\_time string None

meeting\_location string None

info\_comment string None

Time taken: 0.677 seconds

**1.2.** Did  this  query  require  a  MapReduce  job?

**1.3.** What  is  the  name  of  the  Hive  resource  that  was  accessed  to  retrieve  this

schema  information?

**Step  2:** A  Simple  Query

**2.1.** Run  the  following  query:

select \* from wh\_visits where fname = "JOE";

**2.2.** Does  Hive  run  a  MapReduce  job  to  generate  the  result?

**2.3.** Open  your  browser  and  point  it  to  the  JobHistory  UI:

[http://*ipaddress*:19888/](http://ipaddress:19888/)

**2.4.** Notice  the  most  recent  MapReduce  job  executed  is  your  “JOE”  query:

**2.5.** How  many  map  tasks  were  used  to  execute  this  query?

**2.6.** How  many  reduce  tasks  were  used  to  execute  this  query?

**2.7.** How  many  attempts  did  it  take  for  this  task  to  succeed?

**2.8.** How  long  did  it  take  for  this  query  to  execute?

**Step**3: A  Sorted  Query

**3.1.** Run  the  following  query:

hive> select \* from wh\_visits where fname = "JOE" sort by lname;

**3.2.** When  the  MapReduce  job  completes,  find  its  job  details  page  from  the  Job  Browser.

**3.3.** How  many  map  tasks  were  used  to  execute  this  query?

**3.4.** How  many  reduce  tasks  were  used  to  execute  this  query?

**3.5.** The  map  task  outputs  <key,value>  pairs  and  sends  them  to  the  reducer.  What do  you  think  this  MapReduce  job  chose  as  the  key  for  the  mapper’s  output?  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Step 4:** A  Select  Query

**4.1.** Run the  following  query:

hive> select \* from wh\_visits limit 5;

**4.2.** Does  Hive  run  a  MapReduce  job  to  generate  the  results?

**4.3.** What  data  is  read  from  HDFS?

**4.4.** Now  select a  single  column  from  **wh\_visits**:

hive> select fname from wh\_visits limit 5;

**4.5.** Why  did  this require a MapReduce  job  but  “**select  \***”  did  not?

**Step  5:** Using  the  **EXPLAIN**  Command

**5.1.** The  **EXPLAIN**  command  shows  the  execution  plan  of  a  query,  without  actually  executing  the  query.  To  demonstrate,  add  **EXPLAIN**  to  the  beginning  of  the  following  query  that  you  ran  earlier  in  this  lab:

hive> **explain** select \* from wh\_visits where fname = "JOE" sort by lname;

**5.2.** Notice  the  query  is  executed  in  two  stages.  **Stage-­‐0**  performs  the  limit

operator.  This  was  the  first  mapper  that  executed  the  query.

**5.3.** Notice  **Stage-­‐1**  is  a  MapReduce  job  that  has  one  mapper  (look  for  **Map  Operator  Tree**)  and  one  reducer  (under  **Reduce  Operator  Tree**).  As  you  can  see  from  this  execution  plan,  the  mapper  is  doing  most  of  the  work.

**Step  6:** Use EXPLAIN  EXTENDED

**6.1.** Run  the  previous  **EXPLAIN**  again,  except  this  time  add  the  **EXTENDED**  command:

hive> explain **extended** select \* from wh\_visits where fname

= "JOE" sort by lname;

**6.2.** Compare  the  two  outputs.  Notice  the  **EXTENDED**  command  adds a lot of

additional  information  about  the  underlying  execution  plan.

**ANSWERS:**

1.2:  No

1.3:  The  Hive  metastore  contains  the  schema  information  of  all  tables.

2.3:  Yes

2.5:  1  map  task

2.6:  0  reduce  tasks

2.7:  It  probably  succeeded  on  the  first  attempt.  If  not,  you  will  see  multiple  entries  in

the  list  of  attempts.

2.8:  Subtract  the  Execution  Finish  Time  from  the  Execution  Start  Time.  It  should  have

executed  in  about  15-­‐30  seconds.

3.3:  1  map  task

3.4:  1  reduce  task

3.5:  It  makes  sense  for  the  mapper  to  use  **lname**  as  the  key,  which  would  mean  the

visitors  would  already  by  sorted  by  last  name  when  they  got  to  the  reducer.

4.2:  No

4.3:  Hive  simply  reads  the  data  directly  from  the  underlying  file  in  HDFS.

4.5:  Selecting  specific  columns  requires  actual  processing  of  the  contents  of  each  record   to  split  out  the  required  columns.

# Lab 4: Joining Datasets in Hive

|  |  |
| --- | --- |
| **Location  of  Files:** | **/root/labs/Lab7.4** |

**Perform the following steps:**

**Step  1:** Load  the  Data  into  Hive

**1.1.** View  the  contents  of  the  file  **setup.hive**  in  **/root/labs/Lab7.4**:

# more setup.hive

**1.2.** Notice  this  script  creates  three  tables  in  Hive.  The  **nyse\_data**  table  is  filled  with  the  daily  stock  prices  of  stocks  that  start  with  the  letter  “K”,  and  the  **dividends**  table  that  contains  the  quarterly  dividends  of  those  stocks.  The  **stock\_aggregates**  table  is  going  to  be  used  for  a  join  of  these  two  datasets  and  contain  the  stock  price  and  dividend  amount  on  the  date  the  dividend  was  paid.

**1.3.** Run  the  **setup.hive**  script  from  the  **Lab7.4**  folder:

# hive -f setup.hive

**1.4.** To  verify  the  script  worked,  enter  the  following  query  from  the  Hive  Shell:

hive> select \* from nyse\_data limit 20;

hive> select \* from dividends limit 20

You  should  see  daily  stock  prices  and  dividends  from  stocks  that  start  with  the

letter  “K”.

**1.5.** The  **stock\_aggregates**  table  should  be  empty,  but  view  its  schema  to  verify  it  was  created  successfully:

hive> describe stock\_aggregates;

OK

symbol string None

year string None

high float None

low float None

average\_close float None

total\_dividends float None

**Step**2: Join  the  Datasets

**2.1.** The  **join**  statement  is  going  to  be  fairly  long,  so  let’s  create  it  in  a  text  file.  Create  a  new  text  in  the  **Lab7.4**  folder  named  **join.hive**,  and  open  the  file  with  a  text  editor.

**2.2.** We  will  break  the  join  statement  down  into  sections.  First,  the  result  of  the

**join**  is  going  to  put  into  the  **stock\_aggregates**  table,  which  requires  an  **insert**:

insert overwrite table stock\_aggregates

The  **overwrite**  causes  any  existing  data  in  stock\_aggregates  to  be  deleted.

**2.3.** The  data  being  inserted  is  going  to  be  the  result  of  a  select  query  that  contains  various  insightful  indicators  about  each  stock.  The  result  is  going  to  contain  the  stock  symbol,  date  traded,  maximum  high  for  the  stock,  minimum  low,  average  close,  and  sum  of  dividends,  as  shown  here:

select a.symbol, year(a.trade\_date), max(a.high), min(a.low), avg(a.close), sum(b.dividend)

**2.4.** The  from  clause  is  the  nyse\_data  table:

from nyse\_data a

**2.5.** The  join  is  going  to  be  a  left  outer  join  of  the  **dividends**  table:

left outer join dividends b

**2.6.** The  join  is  by  stock  symbol  and  trade  date:

on (a.symbol = b.symbol and a.trade\_date = b.trade\_date)

**2.7.** Let’s  group  the  result  by  symbol  and  trade  date:

group by a.symbol, year(a.trade\_date);

**2.8.** Save  your  changes  to  join.hive.

**Step  3:** Run  the  Query

**3.1.** Run  the  query  and  wait  for  the  MapReduce  jobs  to  execute:

# hive -f join.hive

**3.2.** How  many  MapReduce  jobs  does  it  take  to  perform  this  query?

**Step  4**:Verify  the  Results

**4.1.** Run  a  **select**  query  to  view  the  contents  of  **stock\_aggregates**:

hive> select \* from stock\_aggregates;

The  output  should  look  like:

|  |  |  |
| --- | --- | --- |
| KYO | 2004 | 90.9 66.25 75.79952 0.544 |
| KYO | 2005 | 78.45 62.58 72.042656 0.91999996 |
| KYO | 2006 | 98.01 71.73 85.80327 0.851 |
| KYO | 2007 | 110.01 81.0 93.737686 NULL |
| KYO | 2008 | 100.78 45.41 79.6098 NULL |
| KYO | 2009 | 93.2 52.98 77.04389 NULL |
| KYO | 2010 | 93.83 85.94 90.71 NULL |

stock\_symbol NULL NULL NULL NULL NULL

**4.2.** List  the  contents  of  the  **stock\_aggregates**  directory  in  HDFS.  The   **000000\_0**  file  was  created  as  a  result  of  the  **join**  query:

# hadoop fs -ls -R /apps/hive/warehouse/stock\_aggregates/

41109 /apps/hive/warehouse/stock\_aggregates/000000\_0

**4.3.** View  the  contents  of  the  **stock\_aggregates**  table  using  the  **cat**  command:

# hadoop fs -cat

/apps/hive/warehouse/stock\_aggregates/000000\_0

**RESULT**:  The  **stock\_aggregates**  table  is  a  joining  of  the  daily  stock  prices  and  the  quarterly  dividend  amounts  on  the  date  the  dividend  was  announced,  and  the  data  in  table  is  an  aggregate  of  various  statistics  like  max  high,  min  low,  etc.

# Lab 5: Computing ngrams of Emails in Avro Format

|  |  |
| --- | --- |
| **Location  of  Files:** | /root/labs/Lab7.5 |

**Step**1: View  an  Avro  Schema

**1.1.** Change  directories  to  the  **Lab7.5**  folder.  Notice  this  folder  contains  an  Avro

file  named  **sample.avro**.

**1.2.** Enter  the  following  command  to  view  the  schema  of  the  contents  of  **sample.avro**:

avro cat --print-schema sample.avro

**1.3.** How  many  fields  do  records  in  **sample.avro**  have?

**1.4.** Create  a  schema  file  for  **sample.avro**:

avro cat --print-schema sample.avro > /tmp/sample.avsc

**Step  2:** Create  a  Hive  Table  from  an  Avro  Schema

**2.1.** View  the  contents  of  the  **CREATE  TABLE**  query  defined  in  the

**create\_sample\_table.hive**  file  in  your  **Lab7.5**  folder.

**2.2.** Make  sure  the  **avro.schema.file**  property  points  to  the  schema  file  you  created  in  the  previous  step:

***WITH SERDEPROPERTIES (***

***'avro.schema.url'='file:///tmp/sample.avsc')***

**2.3.** Run  the  CREATE  TABLE  query:

hive -f create\_sample\_table.hive

**Step  3**: Verify  the  Table

**3.1.** Start  the  Hive  shell.

**3.2.** Run  the  **show  tables**  command  and  verify  that  you  have  a  table  named

**sample\_table**.

**3.3.** Run  the  describe  command  on  **sample\_table**.  Notice  the  schema  for

**sample\_table**  matches  the  Avro  schema  from  **sample.avsc**.

**3.4.** Let’s  associate  some  data  with  **sample\_table**.  Copy  **sample.avro**  into  the  Hive

**warehouse**  folder  by  running  the  following  command  (all  on  a  single  line):

hive> dfs –put /root/labs/Lab7.5/sample.avro

/apps/hive/warehouse/sample\_table

**3.5.** View  the  contents  of  **sample\_table**:

hive> select \* from sample\_table;  
 OK

Foo 19 10, Bar Eggs Spam 800

Note  there  is  only  one  record  in  **sample.avro**.

**Step  4:** Create Email  User  Table

**4.1.** There  is  an  Avro  file  in  your  **Lab7.5**  folder  named  **mbox7.avro**,  which  represents  emails  in  an  Avro  format  from  a  Hive  mailing  list  for  the  month  of  July.  Use  the  **-­‐-­‐print-­‐schema**  option  of  **avro**  to  view  the  schema  of  this  file.

**4.2.** How  many  fields  do  records  in  **mbox7.avro**  have?

**4.3.** Run  the  **-­‐-­‐print-­‐schema**  command  again,  but  this  time  output  the  schema  to  a  file  named  **mbox.avsc**:

avro cat --print-schema mbox7.avro > /tmp/mbox.avsc

**4.4.** View  the  contents  of  the  **create\_email\_table.hive**  script  in  your  **Lab7.5**  folder.  Verify  the  **avro.schema.url**  property  is  correct.

**4.5.** Run  the  script  to  create  the  **hive\_user\_email**  table:

hive -f create\_email\_table.hive

**4.6.** Copy  **mbox7.avro**  into  the  warehouse  directory:

hadoop fs -put mbox7.avro

/apps/hive/warehouse/hive\_user\_email/

**4.7.** Verify the  table  has  data  in  it:

select \* from hive\_user\_email limit 20;

**Step  5:** Compute a  Bigram

**5.1.** Start  the  Hive  shell.

**5.2.** Use the  Hive  **ngrams**  function  to  create  a  bigram  of  the  words  in  **mbox7.avro**:

select

ngrams(sentences(content),2 ,10)

from hive\_user\_email;

The output  will  be  kind  of  a  jumbled  mess:

[{"ngram":["2013","at"],"estfrequency":802.0},{"ngram":["of ","the"],"estfrequency":391.0},{"ngram":["I","am"],"estfreq uency":368.0},{"ngram":["I","have"],"estfrequency":340.0},{ "ngram":["J","E9r"],"estfrequency":306.0},{"ngram":["for"," the"],"estfrequency":291.0},{"ngram":["you","are"],"estfreq uency":289.0},{"ngram":["user","hive.apache.org"],"estfrequ ency":289.0},{"ngram":["to","the"],"estfrequency":276.0},{" ngram":["E9r","F4me"],"estfrequency":270.0}]

**5.3.** To  clean  this  up,  use  the  Hive  **explode**  function  to  display  the  output  in  a  more  readable  format:

select

explode(ngrams(sentences(content),2 ,10)) as x from hive\_user\_email;

You  should  see  the a nice, readable list of 10 bigrams:

{"ngram":["2013","at"],"estfrequency":802.0}

{"ngram":["of","the"],"estfrequency":391.0}

{"ngram":["I","am"],"estfrequency":368.0}

{"ngram":["I","have"],"estfrequency":340.0}

{"ngram":["J","E9r"],"estfrequency":306.0}

{"ngram":["for","the"],"estfrequency":291.0}

{"ngram":["you","are"],"estfrequency":289.0}

{"ngram":["user","hive.apache.org"],"estfrequency":289.0}

{"ngram":["to","the"],"estfrequency":276.0}

{"ngram":["E9r","F4me"],"estfrequency":270.0}

**5.4.** Typically  when  working  with  word  comparison  we  ignore  case.  Run  the  query  again,  but  this  time  add  the  Hive  **lower**  function  and  compute  20  bigrams:

select

explode(ngrams(sentences(lower(content)),2 ,20)) as x from hive\_user\_email;

The  output  shoud  look  like  the  following:

{"ngram":["2013","at"],"estfrequency":802.0}

{"ngram":["i","have"],"estfrequency":409.0}

{"ngram":["of","the"],"estfrequency":391.0}

{"ngram":["i","am"],"estfrequency":372.0}

{"ngram":["if","you"],"estfrequency":347.0}

{"ngram":["in","hive"],"estfrequency":337.0}

{"ngram":["for","the"],"estfrequency":309.0}

{"ngram":["j","e9r"],"estfrequency":306.0}

{"ngram":["you","are"],"estfrequency":289.0}

{"ngram":["user","hive.apache.org"],"estfrequency":289.0}

{"ngram":["to","the"],"estfrequency":276.0}

{"ngram":["outer","join"],"estfrequency":271.0}

{"ngram":["2013","06"],"estfrequency":270.0}

{"ngram":["e9r","f4me"],"estfrequency":270.0}

{"ngram":["left","outer"],"estfrequency":270.0}

{"ngram":["in","the"],"estfrequency":252.0}

{"ngram":["gmail.com","wrote"],"estfrequency":248.0}

{"ngram":["17","16"],"estfrequency":248.0}

{"ngram":["06","17"],"estfrequency":246.0}

{"ngram":["wrote","hi"],"estfrequency":234.0}

Step  6: Compute  a  Context  n-­‐gram

**6.1.** From  the  Hive  shell,  run  the  following  query,  which  uses  the  context\_ngrams  function  to  find  the  top  20  terms  that  follow  the  word  “**error**”:

***select explode(context\_ngrams(sentences(lower(content)),***

***array("error", null) ,20)) as x***

***from hive\_user\_email;***

The  output  should  look  like  the  following:

{"ngram":["in"],"estfrequency":102.0}

{"ngram":["return"],"estfrequency":97.0}

{"ngram":["org.apache.hadoop.hive.ql.exec.udfargumenttypeex ception"],"estfrequency":49.0}

{"ngram":["failed"],"estfrequency":49.0}

{"ngram":["is"],"estfrequency":41.0}

{"ngram":["message"],"estfrequency":40.0}

{"ngram":["when"],"estfrequency":39.0}

{"ngram":["please"],"estfrequency":36.0}

{"ngram":["while"],"estfrequency":28.0}

{"ngram":["org.apache.thrift.transport.ttransportexception"

],"estfrequency":28.0}

{"ngram":["datanucleus.plugin"],"estfrequency":26.0}

{"ngram":["during"],"estfrequency":18.0}

{"ngram":["query"],"estfrequency":16.0}

{"ngram":["hive"],"estfrequency":16.0}

{"ngram":["could"],"estfrequency":16.0}

{"ngram":["java.lang.runtimeexception"],"estfrequency":13.0

}

{"ngram":["13"],"estfrequency":12.0}

{"ngram":["error"],"estfrequency":12.0}

{"ngram":["exec.execdriver"],"estfrequency":10.0}

{"ngram":["exec.task"],"estfrequency":10.0}

**6.2.** What  is  the  most  likely  word  to  follow  “**error**”  in  these  emails?

**6.3**. Run  a  Hive  query  that  finds  the  top  20  results  for  words  in  mbox7.avro  that follow  the  phrase  “error  in”.

**RESULT**:  You  have  written  several  Hive  queries  that  computed  bigrams  based  on  the  data  in  the  mbox7.avro  file,  and  also  computed  histograms  using  the  data  in  the  orders  table.  You  should  also  be  familiar  with  working  with  Avro  files,  a  popular  file  format  in  Hadoop.

**SOLUTIONS**:

Step  6.3:

select explode(context\_ngrams(sentences(lower(content)),

array("error", "in", null) ,20)) as x from hive\_user\_email;

# Lab 7: Using HCatalog with Pig

|  |  |
| --- | --- |
| **Location of Files:** | n/a |

**Perform  the  following  steps:**

**Step**1: Start  the  Grunt  Shell

**1.1.** SSH  into  your  HDP  2.0  virtual  machine.

**1.2.** Start  the  Grunt  shell  for  use  with  HCatalog:

# pig -useHCatalog

**Step  2:** Load  a  HCatalog  Table

**2.1.** Define  a  relation  for  the  **wh\_visits**  table  in  Hive  using  the  **HCatLoader()**:

grunt> visits = LOAD 'wh\_visits' USING

org.apache.hcatalog.pig.HCatLoader();

**2.2.** View  the  schema  of  the  **visits**  relation  to  verify  it  matches  the  schema  of  the

**wh\_visits**  table:

visits: {lname: chararray,fname: chararray,time\_of\_arrival: chararray,appt\_scheduled\_time: chararray,meeting\_location: chararray,info\_comment: chararray}

**Step 3:** Run a Pig Query

**3.1.** Let’s execute a  query  to  verify  everything  is  working.  Define  the  following

relation:

grunt> joe = FILTER visits BY (fname == 'JOE');

**3.2.** Dump  the  relation:

grunt> DUMP joe;

The  output  should  be  visitors  from  **wh\_visits**  with  the  firstname  “JOE”.

**Step  4:** Create  an  HCatalog  Schema

**4.1.** Quit  the  Grunt  shell  and  start  the  Hive  shell.

**4.2.** An  HCatalog  schema  is  essentially  just  a  table  in  the  Hive  metastore.  To  define a  schema  for  use  with  HCatalog,  create  a  table  in  Hive:

hive> create table joes (fname string, lname string, comments string);

**4.3.** Verify  the  table  was  created  successfully  using  ***‘show  tables’.***

**4.4.** Use  the  **describe**  command  to  view  the  schema  of  **joes**:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| hive> | describe | joes; |  | |
| OK |  |  |
| fname |  |  | string | None |
| lname |  |  | string | None |

comments string None

**Step 5:** Using  HCatStorer

**5.1.** Exit  the  Hive  shell  and  start  the  Grunt  shell  again.  Be  sure  to  use  the

useHCatalog  option:

# pig -useHCatalog

**5.2.** Define  the  **visits**  and  **joe**  relations  again  (using  the  up  arrow  to  browse

through  the  history  of  Pig  commands).

**5.3.** In  this  step,  you  will  use  **HCatStorer**  in  Pig  to  input  records  into  the  **joes**  table.  To  do  this,  you  need  a  relation  whose  fields  match  the  schema  of  **joes**.  You  can  accomplish  this  using  a  projection.  Define  the  following  relation:

grunt> project\_joe = FOREACH joe GENERATE fname, lname, info\_comment;

**5.4.** Store  the  projection  into  the  HCatalog  table  using  the  **STORE**  command:

grunt> STORE project\_joe INTO 'joes' USING

org.apache.hcatalog.pig.HCatStorer();

This  command  failed.  Why?  \_\_\_\_\_\_\_\_

**5.5.** Notice  the  projection  has  fields  named  **fname**,  **lname**  and  **info\_comment**,  but  the  **joes**  table  in  HCatalog  has  a  schema  with  **fname**,  **lname**  and  **comments**.  The  **fname**  and  **lname**  fields  match,  but  **info\_comment**  needs  to  be  renamed  to  **comments**.  Modify  your  projection  by  using  the  **AS**  keyword:

project\_joe = FOREACH joe GENERATE fname, lname, info\_comment AS comments;

**5.6.** Now  run  the  **STORE**  command  again:

grunt> STORE project\_joe INTO 'joes' USING

org.apache.hcatalog.pig.HCatStorer();

This  time  the  command  should  work  and  a  MapReduce  job  will  execute.

**Step  6:** Verify  the  STORE  Worked

**6.1.** Quit  the  Grunt  shell  and  start  the  Hive  shell  again.

**6.2.** View  the  contents  of  the  **joes**  table:

hive> select \* from joes;

You  should  see  visitors  all  named  “JOE”,  along  with  their  last  name  and  the

comments.

**Step  7:** View  the  Files

**7.1.** You  can  also  check  the  file  system  to  see  if  a  **STORE**  command  worked.  From

the  command  line,  view  the  contents  of  **/apps/hive/warehouse/joes**:

# hadoop fs -ls /apps/hive/warehouse/joes/ Found 1 items

root hdfs 896 /apps/hive/warehouse/joes/part-m-00000

Notice  that  the  file  for  the  **joes**  table  is  named  **part-­‐m-­‐00000**.  Where  did  that

name  come  from?

**7.2.** Use  the  **cat**  command  to  view  the  contents  of  **part-­‐m-­‐00000**:

# hadoop fs -cat /apps/hive/warehouse/joes/part-m-00000

 As  you  can  see,  this  is  the  same  list  of  names  from  the  Hive  select  \*  query,  which should  be  no  surprise  at  this  point  in  the  course!

**RESULT**:  You  have  seen  how  to  run  a  Pig  script  that  uses  HCatalog  to  provide  the  schema

using  HCatLoader  and  HCatStorer.

**ANSWERS**:

5.4:  The  initial  **STORE**  command  failed  because  the  field  names  in  the  relation  you  were  trying to  store  did  not  match  the  column  names  of  the  underlying  table’s  schema.

7.1:  The part-­‐m-­00000  file is a result of the Pig MapReduce  job that executed when you ran the  STORE  command  with  HCatStorer.

# Lab 8 : Advanced Hive Programming

|  |  |
| --- | --- |
| **Location  of  Files:** | **/root/labs/Lab9.1** |

**Step  1:** Create  and  Populate  a  Hive  Table

**1.1.** From  the  command  line,  change  directories  to  the  **Lab9.1**  folder:

# cd ~/labs/Lab9.1

**1.2.** View  the  contents  of  the  **orders.hive**  file  in  the  **Lab9.1**  folder:

# more orders.hive

Notice  it  defines  a  Hive  table  named  orders  that  has  7  columns.  Notice  it  also loads  the  contents  of  **/tmp/shop.tsv**  into  the  orders  table.

**1.3.** Copy  **shop.tsv**  into  the  **/tmp**  folder:

# cp shop.tsv /tmp/

**1.4.** Execute  the  contents  of  **orders.hive**:

# hive -f orders.hive

**1.5.** From  the  Hive  shell,  verify  the  script  worked  by  running  the  following  two

commands:

hive> describe orders;

hive> select count(\*) from orders;

Your  **orders**  table  should  contain  99,999  records.

**Step  2:** Analyze  the  Customer  Data

**2.1.** Let’s  run  a  few  queries  to  see  what  this  data  looks  like.  Start  by  verifying  that

the  **username**  column  actually  looks  like  names:

hive> SELECT username FROM orders LIMIT 10;

You  should  see  ten  first  names.

**2.2.** The  orders  table  contains  orders  placed  by  customers.  Run  the  following

query,  that  shows  the  10  lowest-­‐price  orders:

hive> SELECT username, ordertotal FROM orders ORDER BY

ordertotal LIMIT 10;

The  smallest  orders  are  each  $10,  as  you  can  see  from  the  output:

|  |  |  |
| --- | --- | --- |
| Jeremy 10 | |  |
| Christina | | 10 |
| Jasmine | | 10 |
| Hannah 10 | |  |
| Thomas 10 | |  |
| Michelle | | 10 |
| Brian | 10 | |
| Amber | 10 | |
| Maria | 10 | |

Victoria 10

**2.3.** Run  the  same  query,  but  this  time  use  descending  order:

hive> SELECT username, ordertotal FROM orders ORDER BY

ordertotal DESC LIMIT 10;

The  output  this  time  is  the  10  highest-­‐priced  orders:

Mark 612

Jordan 612

Anthony 612

Brandon 612

Sean 612

Paul 611

Nathan 611

Eric 611

Jonathan 611

Andrew 610

**2.4.** Let’s  find  out  if  men  or  women  spent  more  money:

hive> SELECT sum(ordertotal), gender

FROM orders GROUP BY gender;

Based  on  the  output,  which  gender  has  spent  more  money  on  purchases?

**2.5.** The  **orderdate**  column  is  a  string  with  the  format  **yyyy-­‐mm-­‐dd**.  Use  the  **year**   function  to  extract  the  various  parts  of  the  date.  For  example,  run  the  following  query,   which   computes  the  sum  of  all  orders  for  each  year:

SELECT sum(ordertotal), year(order\_date) FROM orders GROUP BY year(order\_date);

The  output  should  look  like:

|  |  |
| --- | --- |
| 4082780 | 2009 |
| 4404806 | 2010 |
| 4399886 | 2011 |
| 4248950 | 2012 |
| 2570749 | 2013 |

**Step  3**: Multi-­‐File  Insert

**3.1.** In  this  step,  you  will  run  two  completely  different  queries,  but  in  a  single  MapReduce  job.  The  output  of  the  queries  will  be  in  two  separate  directories  in  HDFS.  Start  by  creating  a  new  text  file  in  the  **Lab9.1**  folder  named  **multifile.hive**.

**3.2.** Within  the  text  file,  enter  the  following  query.  Notice  there  is  no  semi-­‐colon

between  the  two  **INSERT**  statements:

FROM ORDERS o

INSERT OVERWRITE DIRECTORY '2010\_orders' SELECT o.\* WHERE year(order\_date) = 2010

INSERT OVERWRITE DIRECTORY 'software'

SELECT o.\* WHERE itemlist LIKE '%Software%';

**3.3.** Save  your  changes  to  **multifile.hive**.

**3.4.** Run  the  query  from  the  command  line:

# hive -f multifile.hive

**3.5.** The above  query  executes  in  a  single  MapReduce  job.  Even  more  interesting, it  only  requires  a  map  phase.  Why  did  this  job  not  require  a  reduce  phase?

**3.6.**   Verify  the  two  queries  executed  successfully  by  viewing  the  folders  in  HDFS:

# hadoop fs -ls

You  should  see  two  new  folders:  **2010\_orders**  and  **software**.

**3.7.** View  the  output  files  in  these  two  folders.  Verify  the  **2010\_orders**  directory  contains  orders  from  only  the  year  2010,  and  verify  the  **software**  directory  contains  only  orders  that  included  **‘Software’**.

**Step  4:** Define  a  View

**4.1.** Define  a  view  named  **2013\_orders**  that  contains  the  **orderid**,  **order\_date**,  **username**,  and  **itemlist**  columns  of  the  **orders**  table  where  the  **order\_date**  was  in  the  year  2013.

**4.2.** Run  the  **show  tables**  command:

hive> show tables;

You  should  see  **2013\_orders**  in  the  list  of  tables.

**4.3.** To  verify  your  view  is  defined  correctly,  run  the  following  query:

hive> SELECT COUNT(\*) FROM 2013\_orders;

The  **2013\_orders**  view  should  contain  13,104  records.

**Step**5: Find  the  Maximum  Order  of  Each  Customer

**5.1.** Suppose  you  want  to  find  the  maximum  order  of  each  customer.  This  can  be

done  easily  enough  with  the  following  Hive  query.  Run  this  query  now:

hive> SELECT max(ordertotal), userid

FROM orders GROUP BY userid;

**5.2.** How  many  different  customers  are  in  the  **orders**  table?

**5.3.** Suppose  you  want  to  add  the  itemlist  column  to  the  previous  query.  Try

adding  it  to  the  **SELECT**  clause:

hive> SELECT max(ordertotal), userid, itemlist

FROM orders GROUP BY userid;

Notice  this  query  is  not  valid  because  **itemlist**  is  not  in  the  **GROUP  BY**  key.

**5.4.** We  can  join  the  result  set  of  the  max-­‐total  query  with  the  **orders**  table  to  add  the  **itemlist**  to  our  result.  Start  by  defining  a  view  named  **max\_ordertotal**  for  the  maximum  order  of  each  customer:

hive> CREATE VIEW max\_ordertotal AS

SELECT max(ordertotal) AS maxtotal, userid

FROM orders GROUP BY userid;

**5.5.** Now  join  the  **orders**  table  with  your  **max\_ordertotal**  view:

hive> SELECT ordertotal, orders.userid, itemlist

FROM orders

JOIN max\_ordertotal ON max\_ordertotal.userid = orders.userid AND

max\_ordertotal.maxtotal = orders.ordertotal;

**5.6.** How  many  MapReduce  jobs  did  this  query  need?

**5.7.** The  end  of  your  output  should  look  like:

600 98

Grill,Freezer,Bedding,Headphones,DVD,Table,Grill,Software,D

ishwasher,DVD,Microwave,Adapter

600 99 Washer,Cookware,Vacuum,Freezer,2-Way

Radio,Bicycle,Washer & Dryer,Coffee

Maker,Refrigerator,DVD,Boots,DVD

600 100 Bicycle,Washer,DVD,Wrench Set,Sweater,2-Way

Radio,Pants,Freezer,Blankets,Grill,Adapter,pillows

**NOTE**:  In  the  next  lab,  you  will  optimize  this  query  using  a  custom  Python script  to  avoid  the  need  for  two  MapReduce  jobs.

**Step  6:** Fixing  the  GROUP  BY  Key  Error

**6.1.** Let’s  compute  the  sum  of  all  of  the  orders  of  all  customers.  Start  by  entering

the  following  query:

SELECT sum(ordertotal), userid FROM orders GROUP BY userid;

Notice  the  output  is  the  sum  of  all  orders,  but  displaying  just  the   userid  is  not  very exciting.

**6.2.** Try  to  add  the  **username**  column  to  the  **SELECT**  clause:

SELECT sum(ordertotal), userid, username

FROM orders

GROUP BY userid;

This  generates  the  infamous  “Expression  not  in  GROUP  BY  key”  error,  because

**username**  column  is  not  being  aggregated  but  the  **ordertotal**  is.

**6.3.** An  easy  fix  is  to  aggregate  the  **username**  values  using  the  **collect\_set**

function,  but  output  only  one  of  them:

SELECT sum(ordertotal), userid, collect\_set(username)[0] FROM orders

GROUP BY userid;

You  should  get  the  same  output  as  before,  but  this  time  the  **username**  is  included.

**Step  7:** Using  the  **OVER**  Clause

**7.1.** Now  let’s  compute  the  sum  of  all  orders  for  each  customer,  but  this  time  use

the  **OVER**  clause  to  not  group  the  output  and  to  also  display  the  **itemlist**  column:

SELECT userid, itemlist, sum(ordertotal) OVER (PARTITION BY userid)

FROM orders;

Notice  the  output  contains  every  order,  along  with  the  items  they  purchased  and

the  sum  of  all  the  orders  ever  placed  from  that  particular  customer.

**Step  8:** Using  the  Window  Functions

**8.1.** It  is  not  difficult  to  compute  the  sum  of  all  orders  for  each  day  using  the

**GROUP  BY**  clause:

select order\_date, sum(ordertotal) FROM orders

GROUP BY order\_date;

Run  the  query  above  and  the  tail  of  the  output  should  look  like:

|  |  |
| --- | --- |
| 2013-07-28 | 18362 |
| 2013-07-29 | 3233 |
| 2013-07-30 | 4468 |
| 2013-07-31 | 4714 |

**8.2.** Suppose  you  want  to  compute  the  sum  for  each  day  that  includes  each  order.   This  can  be  done  using  a  window  that  sums  all  previous  orders  along  with  the   current  row:

SELECT order\_date, sum(ordertotal) OVER

PARTITION BY order\_date ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)

FROM orders;

To  verify  it  worked,  your  tail  of  your  output  should  look  like:

|  |  |
| --- | --- |
| 2013-07-31 | 3163 |
| 2013-07-31 | 3415 |
| 2013-07-31 | 3607 |
| 2013-07-31 | 4146 |
| 2013-07-31 | 4470 |
| 2013-07-31 | 4610 |
| 2013-07-31 | 4714 |

**Step  9:** Using  the  Hive  Analytics  Functions

**9.1.** Run  the  following  query,  which  displays  the  rank  of  the  **ordertotal**  by  day:

SELECT order\_date, ordertotal, rank() OVER

(PARTITION BY order\_date ORDER BY ordertotal) FROM orders;

**9.2.** To  verify  it  worked,  the  output  of  July  31,  2013,  should  look  like:

|  |  |  |
| --- | --- | --- |
| 2013-07-31 | 48 | 1 |
| 2013-07-31 | 104 | 2 |
| 2013-07-31 | 119 | 3 |
| 2013-07-31 | 130 | 4 |
| 2013-07-31 | 133 | 5 |
| 2013-07-31 | 135 | 6 |
| 2013-07-31 | 140 | 7 |
| 2013-07-31 | 147 | 8 |
| 2013-07-31 | 156 | 9 |
| 2013-07-31 | 192 | 10 |
| 2013-07-31 | 192 | 10 |
| 2013-07-31 | 196 | 12 |
| 2013-07-31 | 240 | 13 |
| 2013-07-31 | 252 | 14 |
| 2013-07-31 | 296 | 15 |
| 2013-07-31 | 324 | 16 |
| 2013-07-31 | 343 | 17 |
| 2013-07-31 | 500 | 18 |
| 2013-07-31 | 528 | 19 |
| 2013-07-31 | 539 | 20 |

**9.3.** As  a  challenge,  see  if  you  can  run  a  query  similar  to  the  previous  one  except compute  the  rank  over  months,  instead  of  each  day.

**Step  10:** Histograms

**10.1.** Run  the  following  Hive  query,  which  uses  the  **histogram\_numeric**  function  to  compute  20  (x,y)  pairs  of  the  frequency  distribution  of  the  total  order  amount  from  customers  who  purchased  a  microwave  (using  the  **orders**  table):

select

explode(histogram\_numeric(ordertotal,20)) as x from orders

where itemlist LIKE "%Microwave%";

The  output  should  look  like  the  following:

{"x":14.333333333333332,"y":3.0}

{"x":33.87755102040816,"y":441.0}

{"x":62.52577319587637,"y":679.0}

{"x":89.37823834196874,"y":965.0}

{"x":115.1242236024843,"y":1127.0}

{"x":142.6468885672939,"y":1382.0}

{"x":174.07664233576656,"y":1370.0}

{"x":208.06909090909105,"y":1375.0}

{"x":242.55486381322928,"y":1285.0}

{"x":275.8625954198475,"y":1048.0}

{"x":304.71100917431284,"y":872.0}

{"x":333.1514423076924,"y":832.0}

{"x":363.7630208333335,"y":768.0}

{"x":397.51587301587364,"y":756.0}

{"x":430.9072847682117,"y":604.0}

{"x":461.68715083798895,"y":537.0}

{"x":494.1598360655734,"y":488.0}

{"x":528.5816326530613,"y":294.0}

{"x":555.5166666666672,"y":180.0}

{"x":588.7979797979801,"y":198.0}

**10.2.** Write  a  similar  Hive  query  that  computes  10  frequency  distribution  pairs  for  the  **ordertotal**  from  the  **orders**  table  where  **ordertotal**  is  greater  than  $200.  The  output  should  look  like:

{"x":218.8195174551819,"y":7419.0}

{"x":254.10237580993478,"y":6945.0}

{"x":293.4231618807192,"y":6338.0}

{"x":334.57302573203015,"y":5635.0}

{"x":379.79714934930786,"y":4841.0}

{"x":428.1165628891644,"y":4015.0}

{"x":473.1484734420741,"y":2391.0}

{"x":511.2576946288467,"y":1657.0}

{"x":549.0106899902812,"y":1029.0}

{"x":589.0761194029857,"y":670.0}

**RESULT**:  You  should  now  be  comfortable  running  Hive  queries  and  using  some  of  the more  advanced  features  of  Hive  like  views  and  the  window  functions.

**ANSWERS**:

2.4:  Men  spent  $9,919,847  and  women  spent  $9,787,324.

3.5:  Because  the  query  only  does  a  **SELECT  \***,  no  reduce  phase  was  needed.

5.2:  There  are  100  unique  customers  in  the  **orders**  table.

5.6:  The  query  resulted  in  two  MapReduce  jobs.

**SOLUTIONS**:

Step  4.1:  The  **2013\_orders**  view:

CREATE VIEW 2013\_orders AS

SELECT orderid, order\_date, username, itemlist

FROM orders

WHERE year(order\_date) = '2013';

Step  9.3:  The  rank  query  by  month:

select substr(order\_date,0,7), ordertotal, rank() OVER

(PARTITION BY substr(order\_date,0,7) ORDER BY ordertotal) FROM orders;

Step  10.2:

select

explode(histogram\_numeric(ordertotal,10)) as x from orders

where ordertotal > 200;

# Lab 9 : Streaming Data with Hive and Python

|  |  |
| --- | --- |
| **Location  of Files:** | **/root/labs/Lab9.2** |

**Step  1:** Create  the  max\_ordertotal  View

**1.1.** In  the  previous  lab,  you  defined  a  view  named  **max\_ordertotal**.  Use  the

**describe**  command  to  verify:

hive> describe max\_ordertotal;

OK

maxtotal int None

userid int None

If  you  do  not  have  this  view,  define  it  now  as:

CREATE VIEW max\_ordertotal AS

SELECT max(ordertotal) AS maxtotal, userid

FROM orders GROUP BY userid;

**Step  2:** Think  in  MapReduce

**2.1.** Consider  the  following  join  statement  that  you  executed  in  the  previous  lab:

SELECT ordertotal, orders.userid, itemlist

FROM orders

JOIN max\_ordertotal ON max\_ordertotal.userid = orders.userid AND

max\_ordertotal.maxtotal = orders.ordertotal;

Recall this  join  statement  required  two  MapReduce  jobs  to  execute.

**2.2.** What  if  we  could  send  all  the  orders  by  a  particular  customer  to  the  same reducer?  How  could  we  accomplish  this?

**2.3.** Suppose  we  have  distributed  the  records  so  that  we  know  the  same  reducer  handles  all  orders  from  a  customer.  Then  we  could  sort  the  orders  by  **totalorder**  descending,  and  the  first  order  would  be  their  maximum  order.  Run  the  following  query  to  understand  the  logic  here:

SELECT \* FROM orders DISTRIBUTE BY userid

SORT BY userid, ordertotal DESC;

**2.4.** Look  closely  at  the  output.  Each  customer’s  largest  order  should  appear  first in  his  or  her  respective  list  of  orders.  For  example,  Caitlin  F’s  largest  order  was $600  on  April  25,  2012:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 72094 | 2012-04-25 | 100 | Caitlin | F | 600 | ... |
| 87194 | 2013-01-05 | 100 | Caitlin | F | 588 | ... |
| 53034 | 2011-06-11 | 100 | Caitlin | F | 588 | ... |
| 56003 | 2011-07-30 | 100 | Caitlin | F | 588 | ... |

The  reducer  gets  all  orders  from  a  customer,  and  the  first  order  the  reducer  receives  is  the  largest  one  (which  is  what  we  are  trying  to  find!).  In  the  next  step,  you  will  use  a  custom  reducer  using  Python  that  pulls  off  this  top  value.

**Step  3:** Use  a  Custom  Reducer

**3.1.** Using  a  text  editor,  open  the  file  **max\_order.py**  in  the  **Lab9.2**  folder.

**3.2.** Notice  this  Python  script  prints  the  first  line  that  it  processes.  Then  it  hangs  on to  the  userid  and  skips  all  subsequent  lines  until  the  userid  changes.

**3.3.** Copy  max\_order.py  into  the  /tmp  folder  and  make  it  executable:

# cp max\_order.py /tmp

# chmod +x /tmp/max\_order.py

**3.4.** Start  the  Hive  shell.

**3.5.** Add  **max\_order.py**  as a resource  using  the  **add  file**  command:

hive> add file /tmp/max\_order.py; Added resource: max\_order.py

**NOTE**:  The  **add  file**  command  makes  the  file  available  to  all  mappers  and

reducers  of  this  Hive  query.

**3.6.** Specify  three  reducers  so  we  can  verify  the  logic  of  our  query:

hive> set mapreduce.job.reduces=3;

**3.7.** Now  run  the  following  join  query,  which  uses  the  Python  script  as  its  reducer.  You  may  want  to  type  this  in  a  text  file  so  you  can  rerun  it  easier  if  you  have  a  typo,  and  make  sure  you  use  the  proper  path  to  **max\_order.py**.

from (

select userid,ordertotal,itemlist from orders

distribute by userid

sort by userid,ordertotal DESC)

orders

insert overwrite directory 'maxorders' reduce userid,ordertotal,itemlist

using 'max\_order.py';

The  query  should  execute  a  single  MapReduce  job  this  time,  and  you  should  also  notice  three  reducers.

**Step  4:** View  the  Results

**4.1.** From  the  command  line,  list  the  contents  of  the  **maxorders**  folder  in  HDFS.  You  should  see  three  files,  one  from  each  reducer:

# hadoop fs -ls maxorders

Found 3 items

|  |  |  |  |
| --- | --- | --- | --- |
| root | hdfs | 3611 | maxorders/000000\_0 |
| root | hdfs | 3708 | maxorders/000001\_0 |
| root | hdfs | 3714 | maxorders/000002\_0 |

**4.2.** View  the  contents  of  one  of  the  files:

# hadoop fs -cat maxorders/000000\_0

...

90588 Boots,Grill,Spark Plugs,Vacuum,Coffee Maker,DVD,2-Way

Radio,Dolls,Games,DVD,pillows,Pants

93600 Dishwasher,Table,Grill,DVD,DVD,DVD,Keychain,Dryer, Washer & Dryer,Grill,Coffee Maker,pillows

96600 Table,Jeans,Washer,Wrench Set,Grill,Color Laser

Printer,Dryer,Air Compressor,DVD,Dolls,2-Way Radio,Sweater

99600 Washer,Cookware,Vacuum,Freezer,2-Way Radio,Bicycle,Washer & Dryer,Coffee Maker,Refrigerator, DVD,Boots,DVD

The  output  shows  the  userid,  ordertotal  and  itemlist  of  the  largest  order  placed

by  each  customer.

**RESULT**:  You  used  a  custom  reducer  (a  Python  script)  to  modify  a  Hive  query  that  originally  took  two  MapReduce  jobs  to  execute  so  that  it  can  now  be  executed  in  a  single  MapReduce  job.  You  also  learned  how  to  assign  a  custom  reducer  (or  mapper)  to  a  Hive  query.

**ANSWERS:**

2.2:  Use  the  **DISTRIBUTE  BY**  clause  and  distribute  the  records  by  the  **userid**   column.