ISIT312 Big Data Management

Session 4, 2021

Exercise 4

Conceptual and Logical Modelling of Data Warehouse and more of Hive

In this exercise, you will learn how to use UMLet to create a conceptual and logical schema of a data warehouse, how to create partitioned tables, how to create bucket tables, how to export and import tables, and how to find a query processing plan in Hive.

Be careful when copying the Linux commands in this document to your working Terminal, because it is error-prone. Maybe you should type those commands by yourself.

Prologue

Login to your system and start VirtualBox.

When ready start a virtual machine `ISIT312-BigDataVM-07-SEP-2020`.

(1) How to start UMLet ?

Find UMLet icon on desktop and use the icon to start UMLet 14.3.

(2) How to create a conceptual schema of a data warehouse ?

(2.1) Read he following specification of a sample data warehouse domain.

A multinational company consists of departments located in different countries. A department is described by a name and mission statement. The employees work on the projects. A project is described by a name and deadline. An employee is described by an employee number and full name. Employees work on projects. When an employee completes a project then he/she is re-employed by a company to work on another project. The company would like to record in a data warehouse information about the total number of employees, length of each employment, total salary paid on each employment per year, per month, per project, per department, per city and per country. For example, it should be possible to find the projects and the total number of employees working on each project, or the total number of employees employed in each in each month of a given year in each department, or average salary in each month of each year and for each project etc.

(2.2) First, we identify a fact entity and the measures.

Consider the following fragment of a specification given above.

The company would like to record in a data warehouse information about the total number of employees, length of each employment, total salary paid on each employment per year, per project, per department, per city and per country.

The fragment contributes to a fact entity EMPLOYMENT described by the measures length and salary.

To create a fact entity EMPLOYMENT drag a graphical component fact entity that looks like a cube (Fact name and Measure1, Measure 2,…) from a panel with the graphical widgets to the drawing area of UMLet. Next, change a name of fact from Fact name to EMPLOYMENT and measures Measure1, Measure 2, … to length and salary (the measures are yellow highlighted in a fragment of the specification above). To do so you have to leftclick at the fact entity to make it blue and then modify its properties in a panel "Properties" in the right lower corner of UMLet window.

(2.3) Next, we add the dimensions and attributes describing the entity types in dimensions.

The following fragment of specification indicates the existence of Time dimension that consists of entity types MONTH and YEAR (see a yellow highlighted text in a fragment below).

The company would like to record in a data warehouse information about the total number of employees, length of each employment, total salary paid on each employment per year, per month, per project, per department, per city and per country.

First, we create Time dimension that consists of the entity types MONTH and YEAR. To do so drag two entity type graphical components (Level name, attribute, …) from a panel with graphical widgets to the drawing area of UMLet. Then, change the names of entity types to MONTH and YEAR and connect the entities with one-to-many relationship graphical component, Finally, use one-to-many relationship graphical component to connect MONTH entity to a fact entity EMPLOYMENT.

Next, add an attribute name to an entity MONTH and attribute number to an entity YEAR. To do so you have to leftclick at an entity to make it blue and then modify its properties in a panel "Properties" in the right lower corner of UMLet window.

In the same way add three more dimensions: PROJECT, EMPLOYEE, and DEPARTMENT and describe each entity type with the attributes listed in the specification (see below).

A project is described by a name and deadline.

An employee is described by an employee number and full name.
A department is described by a name and mission statement.

(2.4) Finally, we create the hierarchies over the dimensions.

In this step we create Location and Time hierarchies over the dimensions DEPARTMENT and TIME(MONTH-YEAR). First add a hierarchy blob Criterion between entity type MONTH and many-to-one relationship leading to an entity YEAR. Replace a text Criterion with a text Time.

Next, create two more entity types CITY and COUNTRY and describe both of them with an attribute name. Add one-to-many relationship between COUNTRY and CITY and one-to-many relationship between CITY and EMPLOYMENT.

Finally, to create Location hierarchy add a blob Criterion between entity DEPARTMENT and many-to-one relationship leading to an entity CITY. Replace a text Criterion with a text Location.

To save your design in a file employment.uxf use File->Save option from the main menu. To create pdf file use File->Export as … option from the main menu. When creating pdf file make sure that none of the graphical component in the main window of UMLet is blue highlighted !

(3) How to create a logical schema of a data warehouse ?

To create a logical schema (a collection of relational schemas) of a data warehouse we start from a diagram of conceptual schema created in the previous step. First, we change in a panel located in the right upper corner of UMLet window a collection of graphical widgets from "Conceptual modeling" to "Logical modeling".

Next, we add to each entity type, except fact entity EMPLOYMENT, a surrogate key. For example, we add a surrogate key year_ID to entity YEAR, month_ID to entity MONTH, project_ID to entity PROJECT, etc. We nominate all _ID attributes to be primary keys in the respective relational tables. The names of relational tables remain the same as the names of the respective entity types.

Next, we migrate the primary keys from one side of one-to-many relationships to the relational tables to many side of the relationships and we nominate the migrated _ID attribute as foreign keys.

Next, we nominate a collection of all foreign keys in a table obtained from fact entity type EMPLOYEMENT as a composite primary key in a relational table EMPLOYMENT.

Finally, we replace one-to-many relationships with arrows directed from the locations of foreign keys to the locations of the respective primary keys. A logical schema can be saved and exported in the same way as a conceptual schema.

(4) How to start Hadoop, Hive Server 2, beeline or SQL Developer ?

Open a new Terminal window and start Hadoop in the following way.

```
$HADOOP_HOME/sbin/.start-all.sh
```

When ready minimize the Terminal window used to start Hadoop. We shall refer to this window as to "Hadoop window" and we shall use it later on.

When ready navigate to a folder where you plan to keep your HQL scripts from this lab (you may have to create such folder now) and start Hive Server 2 in the following way.

To start Hive's metastore service, open ***a Terminal window***, type:

```
$HIVE_HOME/bin/hive --service metastore
```

A message shows that metastore is up and running:

```
SLF4J: Actual binding is of type
[org.apache.logging.slf4j.Log4jLoggerFactory]
```

To start hiveserver2, open ***another Terminal window*** and type:

```
$HIVE_HOME/bin/hiveserver2
```

The same message shows that hiveserver2 is up and running.

[IMPORTANT] Don't use Zeppelin to run the above two commands, because the %sh interpreter has a timeout threshold.

You can use Hive's own interface to interact with Hive. Open another new Terminal window and process the following command.

```
$HIVE_HOME/bin/beeline
```

Process the following cmmand in front of `beeline>` prompt.

```
!connect jdbc:hive2://localhost:10000
```

The statement connects you through JDBC interface to Hive 2 server running on a `localhost` and listening to a port `10000`.

Press Enter when prompted about user name. Press Enter when prompted about password. The system should reply with a prompt

```
0: jdbc:hive2://localhost:10000>
```

To find what databases are available process the following command.

```
show databases;
```

At the moment only `default` database is available. We shall create new databases in the future. To find what tables have been created so far process a statement

```
show tables;
```

If you plan to use SQL Developer then leftclick at SQL Developer Icon at the top of task bar. Then rightclick at `hive` connection icon, pick "Connection" option from a menu, enter any user name and password and leftclick at OK button.

We shall use a `default` database for Hive tables created in this laboratory exercise.

(5) How to create and load data into an internal table ?

Create the following internal table to store information about the items.

```
create table item(
 code  char(7),
 name   varchar(30),
 brand  varchar(30),
 price  decimal(8,2) )
    row format delimited fields terminated by ','
    stored as textfile;
```

Next create a text file `items.txt` with sample data given below and save a file in a folder where you plan to keep HQL scripts from this lab (you already started Hive Server 2 from this folder).

```
B000001,bolt,Golden Bolts,12.34
B000002,bolt,Platinum Parts,20.0
B000003,bolt,Unbreakable Spares,17.25
S000001,screw,Golden Bolts,45.00
S000002,screw,Platinum Parts,12.0
N000001,nut,Platinum Parts,21.99
```

When ready process the following HQL statement to load the contents of a file `items.txt` into `item` table.

```
load data local inpath 'items.txt' into table item;
```

Verify the contents of a table `item` with a simple

```
select * from item;
```

(6) How to create a partitioned internal table ?

We would like to improve performance of processing a table `item` through partitioning We expect that a lot of queries will retrieve only the items that have a given name, like for example a query

```
select min(price) from item where name ='bolt';
```

Therefore, we shall create a new table `pitem` as a partitioned table. Process the following statement to create a partitioned table `pitem`.

```
create table pitem(
 code  char(7),
 brand  varchar(30),
 price  decimal(8,2) )
    partitioned by (name varchar(30))
    row format delimited fields terminated by ','
    stored as textfile;
```

Note, that a column `name` has been removed from a list of columns and added as a partition key.

Next, we shall create the partitions for the values in a column `name` of `item` table. Process the following `alter table` statements to create the partitions for `bolt`, `screw`, and `nut`.

```
alter table pitem add partition (name='bolt');
alter table pitem add partition (name='screw');
alter table pitem add partition (name='nut');
```

Next, process the following statement to check if all partitions have been created.

```
show partitions pitem;
```

Now, we shall copy data from a table `item` into a partitioned table `pitem`. Process the following `INSERT` statements.

```
insert into table pitem partition (name='bolt') select code, brand,
price
from item
```

```
where name='bolt';

insert into table pitem partition (name='screw') select code,
brand, price
from item
where name='screw';

insert into table pitem partition (name='nut') select code, brand,
price
from item
where name='nut';
```

Finally, try

```
select * from pitem;
```

to check if all data has been copied.

(7) How to "explain" a query processing plan ?

Are there any differences in the ways how Hive processes an internal table and an internal partitioned table ?

`explain` statement can be used to find the differences in processing of the following `select` statements.

```
explain extended select * from item where name='bolt';
explain extended select * from pitem where name='bolt';
```

The first statement processed on a table item returns the following processing plan.

```
Explain
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------
STAGE DEPENDENCIES:
  Stage-0 is a root stage

STAGE PLANS:
  Stage: Stage-0
    Fetch Operator
      limit: -1
      Processor Tree:
        TableScan
          alias: item
          Statistics: Num rows: 1 Data size: 203 Basic stats:
COMPLETE Column stats: NONE
          GatherStats: false
          Filter Operator
            isSamplingPred: false
            predicate: (UDFToString(name) = 'bolt') (type: boolean)
            Statistics: Num rows: 1 Data size: 203 Basic stats:
COMPLETE Column stats: NONE
```

```
            Select Operator
              expressions: code (type: char(7)), 'bolt' (type:
varchar(30)), brand (type: varchar(30)), price (type: decimal(8,2))
              outputColumnNames: _col0, _col1, _col2, _col3
              Statistics: Num rows: 1 Data size: 203 Basic stats:
COMPLETE Column stats: NONE
              ListSink


22 rows selected.
```

The second statement processed on a partitioned table `pitem` returns a plan.

```
Explain
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------
STAGE DEPENDENCIES:
  Stage-0 is a root stage

STAGE PLANS:
  Stage: Stage-0
    Fetch Operator
      limit: -1
      Partition Description:
          Partition
            input format: org.apache.hadoop.mapred.TextInputFormat
            output format:
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
            partition values:
              name bolt
            properties:
              COLUMN_STATS_ACCURATE {"BASIC_STATS":"true"}
              bucket_count -1
              columns code,brand,price
              columns.comments
              columns.types char(7):varchar(30):decimal(8,2)
              field.delim ,
              file.inputformat
org.apache.hadoop.mapred.TextInputFormat
              file.outputformat
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
              location
hdfs://localhost:8020/user/hive/warehouse/pitem/name=bolt
              name default.pitem
              numFiles 1
              numRows 3
              partition_columns name
              partition_columns.types varchar(30)
              rawDataSize 86
              serialization.ddl struct pitem { char(7) code,
varchar(30) brand, decimal(8,2) price}
              serialization.format ,
```

```
              serialization.lib
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
              totalSize 89
              transient_lastDdlTime 1534561429
            serde:
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe

              input format:
org.apache.hadoop.mapred.TextInputFormat
              output format:
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
              properties:
                bucket_count -1
                columns code,brand,price
                columns.comments
                columns.types char(7):varchar(30):decimal(8,2)
                field.delim ,
                file.inputformat
org.apache.hadoop.mapred.TextInputFormat
                file.outputformat
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
                location
hdfs://localhost:8020/user/hive/warehouse/pitem
                name default.pitem
                partition_columns name
                partition_columns.types varchar(30)
                serialization.ddl struct pitem { char(7) code,
varchar(30) brand, decimal(8,2) price}
                serialization.format ,
                serialization.lib
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
                transient_lastDdlTime 1534561343
              serde:
org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
              name: default.pitem
            name: default.pitem
      Processor Tree:
        TableScan
          alias: pitem
          Statistics: Num rows: 3 Data size: 86 Basic stats:
COMPLETE Column stats: NONE
          GatherStats: false
          Select Operator
            expressions: code (type: char(7)), brand (type:
varchar(30)), price (type: decimal(8,2)), 'bolt' (type:
varchar(30))
            outputColumnNames: _col0, _col1, _col2, _col3
            Statistics: Num rows: 3 Data size: 86 Basic stats:
COMPLETE Column stats: NONE
            ListSink


68 rows selected.
```

Comparison of `Statistics: Num rows` indicates a smaller amount of data processed in the second case.

```
Statistics: Num rows: 1 Data size: 203 Basic stats: COMPLETE
Column stats: NONE
Statistics: Num rows: 3 Data size: 86 Basic stats: COMPLETE Column
stats: NON
```

(8) How partitions are implemented in HDFS ?

To find how partitions are implemented in HDFS move to "Hadoop window" and process the following command.

```
$HADOOP_HOME/bin/hadoop fs -ls /user/hive/warehouse/pitem
```

The partitions of `pitem` tables are implemented as subfolders in `/user/hive/warehouse/pitem`.

(8) How to create dynamically partitioned tables ?

The partitions of dynamically partitioned table are determined when data is loaded to the table.

Process the following statements to set the parameters that allow for dynamic partitioning.

```
set hive.exec.dynamic.partition.mode=nonstrict;
set hive.exec.max.dynamic.partitions=5;
```

Next, drop a table `pitem` and recreate it in the same way as at the beginning of step 6.

```
create table pitem(
 code  char(7),
 brand  varchar(30),
 price  decimal(8,2) )
    partitioned by (name varchar(30))
    row format delimited fields terminated by ','
    stored as textfile;
```

Next, process the following statement to dynamically the partitions and to copy the rows from a table `item` into a table `pitem`.

```
insert overwrite table pitem partition(name)
select code, brand, price, name
from item;
```

Then process the statements to display the partitions and the contents of a table pitem.

```
show partitions pitem;
```

```
select * from pitem;
```

(9) How to create a bucket table ?

Another method to distribute a table in HDFS is to partition it into *bucket*s. A *bucket* is equivalent to a segment of file in HDFS. A column in a table can be selected to determine a distribution of rows over buckets. The value of this column will be hashed into a number and a row will be stored in a bucket with the same number. The rows with the same value in the selected column will be stored in the same bucket. However, the same bucket may also contain the rows with other values from the selected column. A concept of *bucket* is equivalent to a concept of *clustered hash index* on the traditional relational database systems.

Process the following statement to create a table `bitem` distributed over two buckets.

```
create table bitem(
 code  char(7),
 name   varchar(30),
 brand  varchar(30),
 price  decimal(8,2) )
    clustered by (name) into 2 buckets
    row format delimited fields terminated by ','
    stored as textfile;
```

Process the following statement to copy the contents of a table `item` into a table `bitem`.

```
insert overwrite table bitem
select code, name, brand, price
from item;
```

Display a query processing plan for a bucket table.

```
explain extended select * from bitem where name='bolt';
```

```
Explain
----------------------------------------------------------------------
----------------------------------------------------------------------
----------------------------------------------------------------------
------------------------------------------------------
STAGE DEPENDENCIES:
  Stage-0 is a root stage

STAGE PLANS:
  Stage: Stage-0
    Fetch Operator
      limit: -1
      Processor Tree:
        TableScan
          alias: bitem
          Statistics: Num rows: 6 Data size: 199 Basic stats:
COMPLETE Column stats: NONE
          GatherStats: false
          Filter Operator
            isSamplingPred: false
```

```
              predicate: (UDFToString(name) = 'bolt') (type: boolean)
              Statistics: Num rows: 3 Data size: 99 Basic stats:
  COMPLETE Column stats: NONE
              Select Operator
                expressions: code (type: char(7)), 'bolt' (type:
  varchar(30)), brand (type: varchar(30)), price (type: decimal(8,2))
                outputColumnNames: _col0, _col1, _col2, _col3
                Statistics: Num rows: 3 Data size: 99 Basic stats:
  COMPLETE Column stats: NONE
                ListSink
22 rows selected.
```

(10) How to export a table ?

To transfer Hive tables from one Hadoop installation to another one can use `export` and `import` statements. To export an internal table `item` into a folder `exp-item` in HDFS process the following statement.

```
export table item to '/tmp/exp-item'
```

To verify the storage structures created by `export` statement process the following commands in "Hadoop window".

```
$HADOOP_HOME/bin/hadoop fs -ls /tmp/exp*
$HADOOP_HOME/bin/hadoop fs -ls /tmp/exp-item/data
```

Just for fun you can also list the contents of metadata.

```
$HADOOP_HOME/bin/hadoop fs -cat /tmp/exp-item/_meta.
```

(11) How to import a table ?

Assume that we would like to import the contents of exported table `item` into a new table `imported_item`. Process the following commands.

```
import table imported_item from '/tmp/exp-item';
select * from imported_item;
```

End of Exercise 4