Course code: 41S05

UNIT-5

awk- An advanced Filter

awk- An Advanced Filter

- The awk filter combines features of several filters.
- Named after its authors , Aho, Weinberger and Kernighan, **awk**, is the most powerful utility for text manipulation.

Simple awk Filtering:

- the syntax of awk command is :

  awk options 'selection_criteria {action}' file(s)

- The selection_criteria filters input and selects lines for theaction component to act upon.
- the selection_criteria and action is surrounded by a set of single quotes.

- A typically complete **awk** command specifies the selection criteria and action. The following command selects thedirector s from emp.lst:

$ awk '/director/ { print }' emp.lst

9876 | jai sharma |director |production |12/03/50 |7000

2365 |barun gupta |director|personnel |11/05/47 |7800

1006 |chanchal singhavi |director |sales |03/09/38 |6700

- The selection_criteria (/director/) selects lines that are processed in the action section ({ print }).
- If selection_criteria is missing, then action applies to all lines. If action is missing, the entire line is printed. Either of the two (but not both) is optional, but they must be enclosed within a pair of single quotes.
- the **print** statement, when used without any field specifiers, prints the entire line. Also, **print** is the default action of **awk**.
- the following 3 forms are equivalent:

```
awk ' /director/' emp.lst                    //printing is the default action

awk '/director/{ print }' emp.lst            //whitespce is permitted

awk ' /director/ { print $0 }' emp.lst       //$0 is the complete line
```

- For pattern matching, awk uses regular expressions in **sed**-style:

```
$ awk  -F "|"  '/sa[kx]s*ena/'  emp.lst
3212 |shyam   saksena        |d.g.m.        |accounts
|12|12|55 |6000
2345 |j.b.      saxena          |g.m.
|marketing |12|04|67 |8000
```

- The regular expressions used by awk belong to the basic BRE and ERE used by **grep -E** or **egrep**.

Splitting a Line into Fields:

- awk uses special parameter **,$0**, to indicate the entire line. It also identifies fields by $1,$2, $3 .
- awk uses a contiguous sequence of spaces and tabs as a single delimiter.
- To print the name, designation, department and salary of all the sales people:

```
$ awk -F "|" '/sales/ { print $2,$3,$4,$6 }' emp.lst
```

```
a.k shukla g.m. sales 6000
chanchal singhavi diector sales 6700
s.n. dasgupta manager sales 5600
anil aggarwal manager sales 5000
```

- Here, a comma (,) has been used to delimit the filed specifications. This ensures that each filed is separated from the other by a space.

- we can use awk with a line addressing to select lines.

```
$ awk -F "|" 'NR ==3, NR ==6 { print NR, $2,$3,$6 }' empn.lst
```

```
3 n.k. gupta chairman 5400
4 v.k. agarwal g.m. 9000
5 j.b. saxena g.m. 8000
6 sumit chakrobarty d.g.m. 6000
```

printf: Formatting Output

- In the printf format, %os is used for string data and %od for numeric.

```
$ awk  -F"|"  '/[aA]gg?[ar]+wal/  {
>printf "%3d  %-20s %-12s %6d\n", NR,$2,$3,
$6  }' empn.lst
```

```
   4    v.k.              agarwal          g.m.
9000

   9    sumit             Agarwal          executive   7500
  15    anil              aggarwal         manager     5000
```

The name and designation are printed in spaces 20 and 12 characters wide ,respectively. The - symbol **left-justifies** the output.

Redirecting Standard Output:

- every print and printf can be separately redirected with the > and | symbols.

```
printf "%os %o-10s %o-12s %o-8s \n", $1, $3, $4,$6  > "mslist"
```

- The following command **sorts** the output of the **printf** statement:

```
printf "%os %o-10s %o-12s %o-8s \n", $1, $3, $4,$6  |  "sort"
```

- the command or filename that follows the > and | symbols is enclosed within double quotes

Variables and Expressions

- Expressions comprise strings,numbers, variables and entities that are built by combining them with operators eg: (x+5)*12
- awk allows user defined variables but without declaring them;implicitly initialised to zero
- Variables are case sensitive;x is different from X
- Unlike shell variables, awk variables don't use $
- Strings in awk are always double quoted and can contain any character

Variables and Expressions

- awk provides no operator for concatenating strings. Strings are concatenated by simply placing them side by side

```
x="sun";y="com"
print x y   //prints suncom
print x"."y   //prints sun.com
```

awk makes automatic conversions

```
x="5";y=6;z="A"
```

print x y //y converted to string; prints 56
print x+y //x converted to number; prints 11
print y+z //z converted to numeric 0;prints 6

The Comparison Operators:

```
$ awk -F"|" '$3 = = "director" || $3 = ="chairman"
{
> printf "%-20s %-12s %d\n", $2, $3, $6 }' empn.lst
```

```
n.k.        gupta                chairman      5400
lalait      choudhary            director      8200
barun       sengupta             director      7800
jai         sharma               director      7000
chanchal    singahvi             director      6700
```

In the above example, the pattern is matched with a field.

For negating the above condition, we should use the != and && operators:

```
$3!= "director" && $3 ! = "chairman"
```

~ and !~ : The Regular Expression Operators

- The operators ~ and !~ work only with field specifiers ($1, $2 etc.).
- For matching a pattern in a specific field, **awk** offers the ~ and ~! operators to match and negate a match, respectively.

```
$2 ~ /[cC]ho[wu]dh?ury/ || $2 ~ /sa[xk]s?ena/  //matches 2nd field

$2 ~ /[cC]ho[wu]dh?ury|sa[xk]s?ena/      //same as above

$3 !~ /director|chairman/                  //
neither director nor chairman
```

Number Comparison:

- awk can also handle numbers -both integer and floating type-and make relational tests on them.
- The comparison and regular expression matching operators is as shown :

Operator	Significance
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
>=	Greater than or equal to
>	Greater than
~	Matches a regular expression
!~	Doesn't match a regular expression

- For example, to print pay-slips for people whose basic pay exceeds 7500:

```
$ awk -F"|" '$6 > 7500 {
> printf "%-20s %-12s %d \n", $2,$3,$6 }' empn.lst
```

```
v.k.        agarwal      g.m.          9000
j.b.        saxena       g.m.          8000
lalit       choudhary    director      8200
barun       sengupta     director      7800
```

- To match the people born in 1945 or drawing a basic pay greater than 8000:

```
$ awk -F"|" '$6 > 8000 || $5 ~/45$/' empn.lst
```

Here, the context address **/45$/** matches the string 45 at the end ($) of the field.

Number Processing:

- awk can perform computations on numbers using the arithmetic operators **+, -, *, / and %.**
- **awk handles decimal numbers also.**
- to print a rudimentary pay slip for the directors:

```
$ awk -F"|"  '$3 = = "director" {
> printf "%-20s  %-12s %d %d %d\n", $2,$3,$6,
  $6*0.4,$6*0.15 }' empn.lst
```

```
lalait      choudhary       director        8200    3280
1230
barun       sengupta        director        7800    3120
1170
jai         sharma          director        7000
2800        1050
chanchal    singahvi        director        6700    2680
1005
```

Variables:

- awk has certain built-in variables, like NR and $0. It also allows the user to variables of his/her choice.

```
$ awk  -F"|" '$3 = = director" && $6 > 6700 {
>kount = kount + 1
>printf "%3d %-20s %-12s %d\n", kount,$2,$3,
 $6 }' empn.lst
  1    lalit       choudhary       director    8200
  2    barun       sengupta        director    7800
  3    jai         sharma          director    7000
```

- Here, the initial value of kount is zero (0) by default.

The -f option :Storing awk Programs in a File

- the awk programs take the extension **.awk**
- the previous program is stored in the file **empawk.awk**

```
$ cat empawk.awk
    $3 = = "director" && $6 > 6700 {
    printf "%3d -20s %-12s %d\n", ++kount,
    $2,$3,$6 }
```

- now we can use **awk** in the following way to get the same output as before:

```
awk -F"|" -f empawk.awk empn.lst
```

The BEGIN and END sections:

- The BEGIN and END sections are used to do some pre- and post -processing work.
- To print something before processing the first line, for example, a heading, then the BEGIN section can be used.
- The END section is useful in printing some totals after processing is over.
- The BEGIN and END sections are optional and take the form

```
BEGIN   { action }
END     { action }          // both require curly braces
```

- These 2 sections , when present, are delimited by the body of the **awk** program.
- Consider the following awk program , stored in a file **empawk2.awk**. Here, a suitable is heading is printed at the beginning and average salary at the end

```
BEGIN {
    printf  "\t \tEmployee abstract \n \n "
  } $6 > 7500 {        //
      kount++ ; tot+= $6
      printf  ":%3d %-20s %-12s %6d\n", kount, $2,$3,
 $6
  }
 END  {
          printf '\n \t The average basic pay is %6d\n ",
 tot/kount
}
```

To execute this program, use the **-f** option:

```
$ awk  -F "|"  -f empawk2.awk empn.lst
```

Employee abstract

1 v.k. agarwal g.m. 9000

2 j.b. saxena g.m. 8000

3 lalit choudhary director 8200

4 barun sengupta director 7800

The average basic pay is 8250

- We can also perform floating point arithmetic with awk :

```
$ awk 'BEGIN { printf "%f\n", 22/7 }'
3.142857
```

Built-in Variables:

- awk has several built-in variables. They are all assigned automatically

Variable	Function
NR	Cumulative number of lines read
FS	Input filed separator
OFS	Output filed separator
NF	Number of fields in current line
FILENAME	Current input file
ARGC	Number of arguments in command line
ARGV	List of arguments

NR variable: signifies the record number of the current line.

FS variable:

- **awk** uses a contiguous string of spaces as the default field delimiter.
- FS redefines this filed separator as |.
- it must occur in the BEGIN section so that the body of the program knows its value before it starts processing:

```
BEGIN { FS= "|" }
```

- this is an alternative to the -F option which does the same thing.

The OFS variable:

- the awk's default output field separator can be reassigned using the variable OFS in the BEGIN section:

```
BEGIN { OFS="~" }
```

The NF variable:

- by using this variable on a file , say, **empx.lst** , we can locate those lines not having 6 fields as shown :

```
$ awk 'BEGIN { FS = "|" }
> NF ! = 6 {
> print "Record No ", NR, "has" , NF, "FIELDS" }' empx.lst
Record No 6 has 4 fields
Record No 17 has 5 fields
```

The FILENAME variable:

- FILENAME stores the name of the current file being processed.
- awk can also handle multiple filenames the command line.
- By default, awk doesn't print the filename, but we can instruct it to do so:

```
'$6 < 4000 { print FILENAME, $0 }'
```

Arrays:

- An array is a variable that can store a set of values or elements.
- Each value is accessed by a subscript called the index.
- awk arrays are different from the ones used in other programming languages in many ways:
  - They are not formally defined. An array is considered declared the moment it is used.
  - Array elements are initialized to zero or an empty string unless initialized explicitly.
  - Arrays expand automatically.
  - The index can be virtually anything ; it can even be a string.
- Consider the program **empawk3.awk** , which uses arrays to store the totals of the basic pay ,da, hra and gross pay of the sales and marketing people. Assume that the da is 25% , and hra is 50% of basic pay.

```
BEGIN  {
    FS = "|"
    printf   "%46s\n", "Basic        Da      Hra     Gross"
 }   /sales|marketing/   {
   da = 0.25*$6 ;  hra = 0.50*$6  ; gp = $6+hra+da
  tot[1] += $6  ; tot[2] += da  ; tot[3] += hra  ; tot[4] +=
 gp
  kount++
}
END  {
   printf  "\t     Average   %5d  %5d  %5d \n", \
  tot[1]/kount , tot[2]/kount , tot[3]/kount, tot[4]/kount
}
```

Now run this program:

```
$ awk -f empawk3.awk empn.lst
```

```
                        Basic        Da              Hra        Gross
 Average        6812         1703        3406        11921
```

```
$ awk 'BEGIN {
direction["N"]="North";direction["S"]="South";
direction["E"]="East";direction["W"]="West";
printf("N is %os and W is %os\n",direction["N"],direction["W"])
  ;
mon[1]="jan";mon["("]="january";mon["01"]="JAN";
printf("mon[1] is %os\n",mon[1]);
printf("mon[01] is also %os\n",mon[01]);
printf("mon[\"(\"] is also %os\n",mon["("]);
printf("But mon[\"01\"] is %os\n", mon["01"]);
}'
N is North and W is West
Mon[1] is january
Mon[01] is also january
Mon["("] is also january
But mon["01"] is JAN
```

Functions:

- awk has several built-in functions, performing both arithmetic and string operations.
- the arguments are passed to a function , delimited by commas and enclosed by a matched pair of parentheses.
- some functions take a variable number of arguments.
- awk also has some of the common string handling functions .

length:

- determines the length of its argument and if no argument is present, the entire line is assumed to be the argument.
- we can use length (without any argument) to locate lines whose length exceeds 1024 characters:

```
awk -F"|"  'length  > 1024' empn.lst
```

- we can use length with a field as shown:

```
awk -F"|" 'length($2)  <  11' empn.lst
```

index:

- index(s1,s2) determines the position of a string s2 within a larger string s1.

  x = index("abcde", "b")

  returns the value 2.

substr: The **substr (stg,m,n)** function extracts a substring from a string stg. m represents the starting point of extraction and n indicates the number of characters to be extracted.

split : split(stg,arr,ch) breaks up a string **stg** on the delimiter **ch** and stores the fields in an array **arr[ ]**.

```
$ awk -F\| '{ split( $5,ar," /" ) ; print "19" ar[ 3 ] ar[ 2 ] ar[ 1 ] }' empn.lst
19521212
19501203
19431904
"""".
```

system: The **system** function executes a UNIX command within **awk**.

```
BEGIN {
        system( "tput clear" )          //clears the screen
        system( "date" )            }   //executes the UNIX date command
```

```
$ awk -F"|" 'substr ($5,7,2) > 45  &&
  substr($5,7,2)  < 52'  empn.lst

2365 | barun   sengupta |director
  |personnel |11|05|47|7800|2365

3564|sudhirAgarwal|executive|personnel|06|
  07|47|7500|2365

4290|jayantchoudhary|executive|production|0
  7|09|50|6000|9876

9876|jai  sharma |director |production
  |12|03|50 |7000|9876
```

CONTROL FLOW-The if statement:

- The conditional structure (such as **if** statement) and loops (**for** and **while**) execute a body of statements depending on the success or failure of the **control command** .
- The if statement takes the following form:

```
if ( condition is true ) {
        statements
} else {                                    //
else is optional
        statements
}
```

Example:

```
awk -F"|" '{ if ( $6 > 7500 ) printf……
```

- **if statement** can be used with the comparison operators and the special symbols **~** and **!~** to match a regular expression.
- **if statement** can also be used with the logical operators **||** and **&&**.

Examples:

```
if ( NR >=3 && NR <=6 )
        if ( $2 !~ /[aA\gg?[ar]+wal/ )
```

The **if-else** structure:

```
if ( $6 < 6000 )
        da = 0.25*$6
else
        da = 1000
```

The above code can be replaced with a compact conditional structure:

```
$6 < 6000 ? da = 0.25*$6 : da = 1000
```

LOOPING with for:

- The for statement executes the loop body as long as the control command returns a true value . The for has 2 forms. The first form is:

```
for ( K = 1 ;  K<=9  ;  K+=  2 )
```

- This form consists of 3 components: the first component initializes the value of K, the second checks the condition with every iteration , while the third sets the increment used for every iteration.

Using for with an Associative Array:

- The second form of for loop exploits the associative feature of awk's arrays.
- The loop selects each index of an array:

```
for ( k in array )
    commands
```

Here, **k** is the subscript of the array **arr**. The k can also be string and hence we can use this loop to print all environment variables.

```
$ awk 'BEGIN {
> for ( key in ENVIRON )
>         print key "=" ENVIRON [ key ]
> }'
```

```
LOGNAME=sumit
MAIL=/var/mail/sumit
PATH=/usr/bin::/usr/local/bin:/usr/ccs/bin
TERM=xterm
HOME=/home/sumit
SHELL=/bin/bash
.....
```

- We can use any field as the index because the index is actually a string. We can use the string value of $3 as the subscript of the array kount[ ]:

```
$ awk -F"|"  '{  kount[$3]++  }
> END {  for (  desig in kount  )
> print desig,  kount[desig]  }'  empn.lst
```

Output:

```
 g.m.          4
chairman       1
executive      2
director       4
manager        2
 d.g.m         2
```

LOOPING with while:

- The **while** loop repeats a set of instructions as long as its control command returns a true value.
- The previous **for** loop used for centering text can be easily replaced with a **while** loop as shown below:

```
k = 0

  while (k  < (55  - length($0))/2)
 {

        printf "%s", " "

        k++

    }

print $
```

```
$ echo  "
> Income statement\nfor \nthe month of August ,
  2002\nDepartemnt : Sales" |
> awk '{ for (k = 1 ; k < (55 -length($0) )
  /2  ; k++)
> printf  "%s", " "
> printf  $0  }'
```

Output:

Income statement

for

the month of August, 2002

Department : Sales