

## Pattern Matching and Searching

By,  
Srikar Reddy Gondesi,  
Junior SQL Server DBA,  
Database Team,  
Miracle Software Systems. Inc,  
Email: [sgondesi@miraclesoft.com](mailto:sgondesi@miraclesoft.com)

# Cut

- Used for text processing
- Used to extract portion of text from a file by selecting columns.

## Examples :

- `cut -c2 --->` Prints 2<sup>nd</sup> column of each line in a file
- `cut -c1-3 --->` Prints 1,2,3 columns of each line in a file
- `cut -c2- --->` Prints 2 character to end of the line

- Where does the name come from?
  - GREP – Global Regular Expression Print
- How it works?
  - grep looks inside file(s) and returns any line that contains the string or expression
  - prints lines matching a pattern to STDOUT

- grep Pattern

- grep Pattern filename

Ex. **grep 'permission' thisFile**

- grep pattern file1 file2

Ex. **grep 'permission' file1 file2**

file1:

file2:

- If grep cannot find a line in any of the specified files that contain the requested pattern, no output is produced.

- Look for patterns one line at a time, like **grep**
- *Change lines of the file*
- Non-interactive text editor
  - Editing commands come in as *script*
  - There is an interactive editor *ed* which accepts the *same commands*
- A Unix filter
  - Superset of previously mentioned tools

# SED Syntax

Syntax: **sed [-n] [-e] ['command'] [file...]**  
**sed [-n] [-f scriptfile] [file...]**

**-n** - only print lines specified with the print command (or the 'p' flag of the substitute ('s') command)

**-f scriptfile** - next argument is a filename containing editing commands

**-e command** - the next argument is an editing command rather than a filename, useful if multiple commands are specified

# SED Examples

- sed '1d' file.txt
- sed '1,6d' file.txt
- sed '2,5!d' file.txt
- sed '2,5p' file.txt
- sed 's/unix/linux/' file.txt
- sed 's/unix/linux/g' file.txt
- sed 's/unix/linux/5' file.txt
- sed 's/unix/linux/w file\_name' file.txt
- sed 's/unix/linux/i' file.txt
- sed '10s/unix/linux/g' file.txt



**Awk**



## Why is it called AWK?



*Aho*



*Weinberger*



*Kernighan*

# AWK Introduction

- awk's purpose: **A general purpose programmable filter that handles text (strings) as easily as numbers**
  - This makes awk one of the most powerful of the Unix utilities
- awk processes fields while sed only processes lines
- nawk (new awk) is the new standard for awk
  - Designed to facilitate large awk programs
  - gawk is a free nawk clone from GNU
- awk gets its input from
  - files
  - redirection and pipes
  - directly from standard input

# Structure of an AWK Program

An awk program consists of:

- An optional BEGIN segment
  - For processing to execute prior to reading input
- pattern - action pairs
  - Processing for input data
  - For each pattern matched, the corresponding action is taken
- An optional END segment
  - Processing after end of input data

```
BEGIN {action}  
pattern {action}  
pattern {action}  
  
.  
.  
.  
  
pattern { action}  
END {action}
```

# An Example

```
ls | awk '  
  BEGIN { print "List of html files:" }  
  /\u005C.\u005Chtml$/ { print }  
  END { print "There you go!" }  
'
```

List of html files:

index.html

as1.html

as2.html

There you go!

# Running an AWK Program

- There are several ways to run an Awk program
  - `awk 'program' input_file(s)`
    - program and input files are provided as command-line arguments
  - `awk 'program'`
    - program is a command-line argument; input is taken from standard input (yes, awk is a filter!)
  - `awk -f program_file input_files`
    - program is read from a file

- Each input line is split into fields.
  - **FS**: field separator: default is whitespace (1 or more spaces or tabs)
  - awk -Fc option sets **FS** to the character *c*
    - Can also be changed in BEGIN
  - **\$0** is the entire line
  - **\$1** is the first field, **\$2** is the second field, ....
- Only fields begin with \$, variables are unadorned

# Computing with AWK

- Counting is easy to do with Awk

```
$3 > 15 { emp = emp + 1}  
END { print emp, "employees worked  
more than 15 hrs"}
```

- Computing Sums and Averages is also simple

```
{ pay = pay + $2 * $3 }  
END { print NR, "employees"  
      print "total pay is", pay  
      print "average pay is", pay/NR  
    }
```

# Selection with AWK

- Awk patterns are good for selecting specific lines from the input for further processing
  - Selection by Comparison
    - `$2 >= 5 { print }`
  - Selection by Computation
    - `$2 * $3 > 50 { printf("%6.2f for %s\n", $2 * $3, $1) }`
  - Selection by Text Content
    - `$1 == "NYU"`
    - `$2 ~ /NYU/`
  - Combinations of Patterns
    - `$2 >= 4 || $3 >= 20`
  - Selection by Line Number
    - `NR >= 10 && NR <= 20`





**THANK YOU**