SECTION 1 UNIX LAB

Structure **Page Nos.**

1.0 INTRODUCTION

You have studied MCS-041, a theoretical course on Operating Systems. In that course we had gone through the features of an OS, different functions of OS and their management. Also, we had touched upon the case studies of two operating systems: namely, WINDOWS 2000 and LINUX. This is the lab component associated with the MCS-041 course. In this section 1 you will be provided the hands on experience on UNIX/LINUX operating system.

UNIX is a computer operating system, a system program that works with users to run programs, manage resources and communicate with other computer systems. UNIX is a multiuser operating system. Its commands are similar to spoken language, commands acting as verbs, command options acting as adjectives and the more complex commands acting akin to sentences. It is a multitasking operating system in which a user can run more than one program or task at a time. UNIX is a layered operating system. The innermost layer is the hardware that provides the services for the OS. The operating system, referred to in UNIX as the **kernel**, interacts directly with the hardware and provides the services to the user programs. User programs interact with the kernel through a set of standard **system calls**. These system calls request services to be provided by the kernel. Such services would include accessing a file: open close, read, write, link, or execute a file; starting or updating accounting records; changing ownership of a file or directory; changing to a new directory; creating, suspending, or killing a process; enabling access to hardware devices; and setting limits on system resources.

By now, you must have obtained the practical skills of LINUX. Refer the course material of MCS-022, Block-2 Linux Operating System for information and help on LINUX. Also refer the lab manual of MCSL-025, Section-2 Operating Systems and Networking Lab. Hope you have done all the exercises given in the practical sessions of this and preserved the lab manual also for further reference.

If you have versions of UNIX at your study centre, you should execute all the sessions given at the end of this Section using UNIX. Or else, you may use the Linux. Operating System for solving them. Try to execute all the example commands/shell scripts/shell programs along with the problems given at the end of this section (session wise). Use the online reference manual provided by the ***man*** command of UNIX to the possible extent to know all the underlying options of the commands.

In order to successfully complete this section, the learner should adhere to the following general guidelines:

- The student should attempt all exercises / problems / assignments given in the list, session wise.
- You may seek assistance in doing the lab exercises from the concerned lab instructor. Since the assignments have credits, the lab instructor is obviously not expected to tell you how to solve these, but you may ask questions concerning the technical problems you are encountering during the sessions.
- For each program you should add comments as far as possible.
- The program should be interactive, general and properly documented with real Input/ Output data.
- You are strongly advised not to copy somebody else's work.
- It is your responsibility to create a separate directory to store all the programs, so that nobody else can read or copy.
- Observation book and Lab record are compulsory.
- The list of the exercises (session-wise) is available to you in this lab manual. For each session, you must come prepare with the necessary commands, algorithms / programs and necessary documentation written in the Observation Book. You should utilise the lab hours for executing the programs, testing for various desired outputs and enhancements of the programs.
- As soon as you have finished a lab exercise, contact one of the lab instructor / Incharge in order to get the exercise successfully completed by you for evaluation and also get the signature from him/her on your Observation book against them.
- Completed lab assignments should be submitted in the form of a Lab Record in which you have to write the commands / algorithm / program code along with comments and output for various inputs given.
- For this UNIX lab, the total no. of lab sessions (3 hours each) are 10 and the list of assignments is provided session-wise. It is important to observe the deadline given for each assignment.

1.1 OBJECTIVES

After going through this section, you should be able to:

- mention the features of UNIX;
- recognize, understand and make use of various UNIX commands;
- gain hands on experience of UNIX commands and shell programs;
- feel more confident about writing the shell scripts and shell programs;
- apply the concepts that have been covered in this manual, and
- know the alternate ways of providing the solutions to the given practical exercises and problems.

1.2 HISTORY OF UNIX

The tabular form given below shows the developments of UNIX year wise.

Year	Developments
1965	Multics project begun as joint venture of AT&T, MIT, and GE to create a new operating system for the GE computer.
1969	AT&T Bell Labs researchers Ken Thompson, Dennis Ritchie, J. F. Ossanna, and R. H. Canaday create a prototype file management system as an alternative to MULTICS. Commercial systems at the time were written entirely in assembly language. One of the goals of UNIX is to have a small kernel written in assembler, and the rest in any high-level language. Unix also has a hierarchical file system and a collection of utility programs.
1970	Brian Kernighan coins the name UNICS (UNiplexed Information and Computing System). Unix development increases with the acquisition of a DEC (Digital Equipment Corporation) PDP-11, a state-of-the-art $65,000 computer with 24 kilobytes of RAM and 512 kilobytes of disk space. Thompson develops B as an alternative to FORTRAN and BCPL.
1971	1st Unix version, V1, used only within Bell Labs. Needing to justify the cost of development, UNIX is used (with the assembly-language-coded *troff*) in the Bell Labs patent department as a one of the first word-processing programs. B is improved upon and its successor is named C.
1972	M. D. McIlroy introduces the novel idea of ‘pipes’. June – Version 2, 10 Unix installations.
1973	Version 3, 16 Unix installations. November – Version 4 is rewritten in C, easing the portability of Unix.
1974	S. R. Bourne develops the Bourne Shell (/bin/sh, indicated with a '$') June – Version 5 Estimated 50 Unix installations.
1975	AT&T leases Version 6 to universities at low cost, making UNIX use widespread. Thompson spends year at UC Berkeley, leads development of a BSD variant of Unix. UC Berkeley graduate student Bill Joy (who later starts Sun Microsystems) develops the C-shell (/bin/csh, indicated with a '%') and the vi text editor. TENEX-style C-shell developed (/bin/tcsh). David Korn from AT&T develops the Korn shell (/bin/ksh).
1976	Emacs originally written by Richard Stallman.
1977	1BSD released. Tom Duff and Byron Rakitzis develop the rc shell.
1978	Students at UC Berkeley, known as "Berkeley Software Distribution", develop their own variant of UNIX, called BSD. 2BSD released, 75 copies distributed. 600 Unix installations worldwide.
1979	Private companies begin porting commercial versions of Unix. BSD releases 3BSD. AT&T releases the 40KB-kernel Version 7.
1980	Microsoft releases Xenix, which is the first attempt to bring Unix to desktop computers. October - BSD releases 4.0 BSD
1982	AT&T releases its first commercial version of Unix, System III. *Ksh* was delivered working in 1982 to AT&T Bell labs.
1983	Computer Research Group (CRG), UNIX System Group (USG), and Programmer's WorkBench (PWB) merge to become UNIX System Development Lab. AT&T releases System V, incorporating Xenix and other variants. BSD releases 4.2BSD which includes complete implementation of TCP/IP networking protocols, including telnet and ftp. SVID, the System 5 Interface Definition, is released in an effort to

	standardize the UNIX flavors as much as possible.
1984	Estimated 100,000 UNIX installations worldwide. U.S. government charges AT&T with monopolistic practices and AT&T is forced to divest its interests. AT&T releases SVR2, incorporating many features from 4.2BSD X/Open consortium of vendors founded, eventually known as The Open Group, gets UNIX trademark. Richard Stallman develops GNU (GNU's Not UNIX) as a free UNIX clone.
1985	February - AT&T releases Version 8 Paul Falstad develops *zsh*.
1986	September - AT&T releases Version 9 DEC, which had been supporting VAX/VMS, is forced to acknowledge and support UNIX as an inexpensive alternative. *Rc* shell upgraded to *es*.
1987	Estimated 100,000 Unix installations worldwide.
1988	Unix International (UI) and Open Software Foundation (OSF) are formed. SVR4 releases as a combo of System V, BSD, and SunOS.
1989	October - AT&T releases V10, the final version. Wanting a free alternative to *ksh*, GNU advocates develop *bash* (Bourne-again shell).
1991	Unix Systems Laboratory (USL) spun off as a separate company, majority-owned by AT&T. OSF releases OSF/1. *Linus Torvalds releases Linux kernel (Linus's Minix, pronounced 'lin-ux')*
1992	July 14 - William and Lynne Jolitz release 386BSD as open source, eventually evolving into NetBSD, FreeBSD, and OpenBSD.
1993	4.4BSD released as final Berkely release. June 16 - Novell buys USL from AT&T.
1994	Torvalds and many others relase version 1.0 of the Linux kernel. Used with Stallman's GNU command-set, users around the world have access to a free UNIX variant known as GNU/Linux, or just Linux.
1995	Santa Cruz Operation (SCO) buys USL from Novell June - 4.4 BSD Lite Release 2 the final distribution.
1996	OSF and X/Open merge to become The Open Group.
1997	The Open Group releases Single UNIX Specification, Version 2.
1998	20 million UNIX installations worldwide.
1999	After years of growing hype about Linux, and Microsoft staunchly refusing to take part in the open source movement, many companies feel compelled to choose between developing software for UNIX or for Windows NT.

Let us study the features of UNIX operating system in the following section.

1.3 FEATURES OF UNIX

The UNIX operating system is a popular Operating system because of its simplicity in its design and functioning. The following are the key features which made the UNIX OS very popular:

- Multiuser system
- Time sharing
- Portability
- Multitasking
- Background processing
- Hierarchical file system
- Security and Protection
- Better communication
- Availability of compilers / tools / utilities

- Shell programming.

1.4 KERNEL AND THE SHELL

The main control program in a UNIX operating system is called the **kernel**. However, the kernel does not allow the user to give its commands directly; instead when the user types commands on the keyboard they are read by another program in the Operating System called a **shell** which parses, checks, translates and then passes them to the kernel for execution.

There are a number of different shells available, with names such as *sh, csh, tcsh, ksh, bash*, each with different rules of syntax; these are partly though not completely responsible for the diversity of UNIX. Later in our discussion we will see what are the criteria to select a shell. Once the command has been interpreted and executed, the kernel sends its reply, which may simply be a **prompt** for the next command to be entered, either directly to the display monitor. This is a program responsible for deciding where and in what form the output will appear on the display monitor. If for any reason the kernel cannot perform the command requested (wrong syntax), for example, the reply will be an error message; the user must then re-enter the corrected command.

1.4.1 Commands and Processes

The kernel and the shell programs running in the CPU are examples of **processes**; these are self-contained programs that may take over complete control of the CPU. Although there can only be one kernel process running in a particular CPU, there may be any number of shell and other processes, subject of course to memory limitations.

Some commands to the shell are **internal** (or **built-in**), that is, they only involve the shell and the kernel. Others are **external** and may be supplied with the OS, or may be user-written. An external command is the name of a file which contains either a single executable program or a **script**. The latter is a text file, the first line of which contains the name of a file containing an executable program, usually but not necessarily a shell, followed by a sequence of commands for that program. A script may also invoke other scripts – including itself. Its purpose is simply to avoid having to re-type all the command it contains.

A command may also be an **alias** for an internal or external command (*e.g.,* the user may not like the UNIX name "***rm***" for the command which deletes files, and may prefer to alias it to "delete").

The external command may optionally cause execution of the shell process to be temporarily suspended, and then run another program, which may then take over input from the keyboard and mouse and send output for display. The shell may or may not wait for the program to finish, before it wakes up again and cause its prompt to be displayed. It is very important that the user be continuously aware of which process is currently reading keyboard input: the shell or another program, because they usually speak completely different languages.

The above is an example of a **parent** process – the shell, and a **child** process – the external program. In fact the child could just as well have been, and often is, another invocation of the same shell, or of a different shell, and the child process can be the parent of other child and so on (almost) *ad infinitum*. Consequently a typical UNIX system has many processes either waiting or running.

1.4.2 UNIX File System

Like other Operating system's, UNIX organises information into **files**, and related files may be conveniently organised in **directories**. Files may contain text, data, executable programs, scripts (which are actually just data for a scripting program such as a shell), and may also be links to other files, or to physical devices or communications channels.

The directory structure is **hierarchical** with the **root directory**, indicated by a forward slash (/), at the base of the tree. This may contain files as well as other directories such as /bin, /etc, /lib, /tmp, /usr, and /d. Actually in this example the root directory will contain directory files called bin, etc, lib, tmp, usr and d, each of which contains a list of files and their locations on the disk for each of the corresponding directories. Each directory may contain further ordinary files and directory files, and so on. The / character is used to delimit the components of the name. For example, the /d directory may contain directories such as /d/user1 and /d/user2 and these may contain the user's **home directories.**

The entire directory hierarchy may reside on a single physical disk, or it may be spread across several disks; the boundaries between physical disks cannot be seen merely by looking at the directory hierarchy. Your system administrator will decide where your home directory is to be both physically and logically located.

Every file in the hierarchy is identified by a **pathname**, this is merely a description of the path you have to traverse to get from the root directory to the file. Strictly, a **filename** is distinct from a pathname; a filename is just one of the components of the pathname delimited by /'s.

Relative pathnames which have a start point anywhere but / in the hierarchy may be used instead, and are often more convenient.

1.4.3 Wild Card Characters

Another shorthand character is the use of "wildcard" character in filenames, technically called filename **globbing**. The ***** character in a filename represents any string of characters, including no characters; the **?** character represents any single character. These wildcard characters may be used more than once, and may appear in combination in a pathname.

Always note that UNIX is always fussy about the case of letters in commands, usernames, passwords and filenames; so Vvs.data is not the same file as vvs.data.

1.4.4 Syntax of UNIX Commands

The general form of a UNIX command is: command [option(s)] [argument(s)] (the [ ] here indicate that the items they contain are optional; they are not part of the syntax).

If a typing error is made the line may be changed using the left/right arrow keys to move the cursor and the *Backspace* key to delete the character to the left of the cursor; new characters are inserted before the cursor. After keying-in the desired command, press the *Enter* key to execute the command. The command may be cancelled before execution by using *Ctrl*-u (hold down the *Ctrl* key and press the u key). If an incorrectly spelled command is entered and spelling correction is enabled in the shell, the shell will attempt to correct the mistake and ask for verification.

If the shell has **filename completion** enabled, use of the *Tab* key after part of a command or filename has been typed will cause the shell to attempt completion of the name, up to the character where the result is unique. Use of the *Ctrl*-d key will cause all names that match what has been typed to be listed.

The options and arguments if present must be separated from the preceding item by at least one space. However, multiple options may or may not need to be separated from each other by at least 1 space; multiple arguments are always separated from each other by at least 1 space. Options usually start with -, but occasionally it is a +, and sometimes the – or + may be omitted.

A line may contain several commands (each possibly followed by options and/or arguments) separated by semicolons (;). The commands are executed in sequence, just as if they had been typed on separate lines. If it is necessary to continue a command onto the next line, end the line with a backslash (\), press *Enter* and continue typing on the next line.

In a script anything after# on a line is treated as a comment, *i.e.,* it is ignored.

To background a command simply add an ampersand (&) on the end.
Commands may be piped using a vertical bar **|** to separate them. As an alternative to piping the output may be **redirected** to write to a file using **>** filename or appended to the file using **>>** filename. This only redirects the **standard output** stream; redirection of error messages (**standard error** stream) requires a different syntax which is shell-dependent. The **standard input** stream may be redirected to read from a file using <filename.

1.4.5 Getting Help

UNIX is an operating system, with hundreds of commands that can be combined to execute thousands of possible actions. UNIX provides complete information about each and every command which is stored in the UNIX *man* pages (*man* stands for manual). We can go through them by giving the *man* command at the prompt as shown below:

$ man cp

$ man ls

$ man grep

1.5 UNIX COMMANDS

With many competing standards (UNIX 98, UNIX95, POSIX.2, SVID3, 4.3BSD, etc.) and most users having to deal with multiple systems, it's crucial to know which commands are important enough to be used on nearly every version of UNIX. The following is a list of **commonly used commands** which are organised under different categories for understanding and ease of use. Keys proceeded by a ^ character are CONTROL key combinations.

Terminal Control Characters

^h	backspace erase previously typed character
^u	erase entire line of input so far typed
^d	end-of-input for programs reading from terminal
^s	stop printing on terminal
^q	continue printing on terminal
^z	currently running job; restart with bg or fg
DEL, ^c	kill currently running program and allow clean-up before exiting
^\	emergency kill of currently running program with no chance of cleanup

Login and Authentication

login	access computer; start interactive session
logout	disconnect terminal session
passwd	change local login password; you MUST set a non-trivial password

Information

date	show date and time
history	list of previously executed commands
pine	send or receive mail messages
msgs	display system messages
man	show on-line documentation by program name

info	on-line documentation for GNU programs
w, who	who is on the system and what are they doing
who am i	who is logged onto this terminal
top	show system status and top CPU-using processes
uptime	show one line summary of system status
finger	find out info about a user@system

File Management

cat	combine files
cp	copy files
ls	list files in a directory and their attributes
mv	change file name or directory location
rm	remove files
ln	create another link (name) to a file
chmod	set file permissions
des	encrypt a data file with a private key
find	find files that match specified criteria

Display Contents of Files

cat	copy file to display device
vi	screen editor for modifying text files
more	show text file on display terminal with paging control
head	show first few lines of a file(s)
tail	show last few lines of a file; or reverse line order
grep	display lines that match a pattern
lpr	send file to line printer
pr	format file with page headers, multiple columns etc.
diff	compare two files and show differences
cmp	compare two binary files and report if different
od	display binary file as equivalent octal/hex codes
file	examine file(s) and tell you whether text, data, etc.
wc	count characters, words, and lines in a file

Directories

cd	change to new directory
mkdir	create new directory
rmdir	remove empty directory (remove files first)
mv	change name of directory
pwd	show current directory

Devices

df	summarize free space on disk device
du	show disk space used by files or directories

Special Character Handling for C-shell

*	match any characters in a file name
~user	shorthand for home directory of "user"
$name	substitute value of variable "name"
\	turn off special meaning of character that follows
‘	In pairs, quote string w/ special chars, except !
“	In pairs, quote string w/ special chars, except !, $
`	In pairs, substitute output from enclosed command

Controlling Program Execution for C-shell

&	run job in background
DEL, ^c	kill job in foreground
^z	suspend job in foreground
fg	restart suspended job in foreground
bg	run suspended job in background
;	delimit commands on same line
( )	group commands on same line

!	re-run earlier command from history list
ps	print process status
kill	kill background job or previous process
nice	run program at lower priority
at	run program at a later time
crontab	run program at specified intervals
limit	see or set resource limits for programs
alias	create alias name for program (in .login)
sh, csh	execute command file

Controlling Program Input/Output for C-shell

|	pipe output to input
>	redirect output to a storage file
<	redirect input from a storage file
>>	append redirected output to storage file
tee	copy input to both file and next program in pipe
script	make file record of all terminal activity

E-mail and communication

pine	process mail with full-screen menu interface or read USENET news groups
msgs	read system bulletin board messages
mail	send e-mail; can be run by other programs to send existing files via e-mail
uuencode	uudecode encode/decode a binary file for transmission via mail
finger	translate real name to account name for e-mail
talk	interactive communication in real-time
rn	read USENET news groups

Editors and Formatting Utilities

sed	stream text editor
vi	screen editor
emacs	GNU emacs editor for character terminals
xemacs	GNU emacs editor for X-Windows terminals
pico	very simple editor, same as used in "pine"
fmt	fill and break lines to make all same length
fold	break long lines to specified length

Printing

lpr	send file to print queue
lpq	examine status of files in print queue
lprm	remove a file from print queue
enscript	convert text files to PostScript format for printing

Interpreted Languages and Data Manipulation Utilities

sed	stream text editor
perl	Practical Extraction and Report Language
awk	pattern scanning and processing language; 1985 vers.
sort	sort or merge lines in a file(s) by specified fields
tr	translate characters
cut	cut out columns from a file
paste	paste columns into a file
dd	copy data between devices; reblock; convert EBCDIC

Networking/Communications

telnet	remote network login to other computer
ftp	network file transfer program
rlogin	remote login to “trusted” computer
rsh	execute single command on remote “trusted” computer
rcp	remote file copy to/from “trusted” computer
host	find IP address for given host name, or vice versa

lynx	Web browser for character based (text-only) terminals
gzip, gunzip	compress/decompress a file
tar	combine multiple files/dirs into single archive
uuencode, uudecode	encode/decode a binary file for transmission via mail

Compilers, Interpreters and Programming Tools

csh	command language interpreter (shell scripts)
f77	DEC Fortran 77 compiler
f2c	convert fortran source code to C source code
cc, c89	DEC ANSI 89 standard C compiler
gcc	GNU C compiler
g++	GNU C++ compiler
pc	DEC Pascal compiler
dbx	symbolic debugger for compiled C or Fortran
make	recompile programs from modified source
gmake	GNU version of make utility
awk	interpreter for awk language
error	analyze and disperse compiler error messages.

1.6 DESCRIPTION OF COMMONLY USED UNIX COMMANDS

The description for the most commonly used UNIX commands is given below in an alphabetic order.

cat

cat allows you to read multiple files and then print them out. You can combine files by using the > operator and append files by using >>.

Syntax: ***cat [argument] [specific file]***

Example:

cat abc.txt

If you want to append three files (abc.txt, def.txt, xyz.txt), give the command as,
cat abc.txt def.txt xyz.txt > all

cd, chdir

cd (or chdir) stands for "change directory". This command is the key command to move around your file structure.

Syntax: ***cd [name of directory you want to move to]***

When changing directories, start with / and then type the complete file path, like
cd /vvs/abc/xyz

chmod

chmod (which stands for "change mode") changes who can access a particular file. A "mode" is created by combining the various options from who, opcode, and permission.

Syntax: ***chmod [option] mode file***

If you look at a list of files using the long list command *ls –l*, you'll see the permissions, owner, file size, modification time, and filename. The first column of the list shows who can read, write, and execute the files or directories, in other words, the permissions. It basically shows who has permission to do what to a given file or directory. ***r*** stands for "read" and means that you're allowed to read the file or directory. ***w*** stands for "write" and gives permission to edit or change the file as well as create, move, rename, or remove a directory. ***x*** stands for "execute" which gives permission to run a file or search a directory. Every file or directory has four sets of ***rwx*** permissions. The first set represents the user (u), the second set represents the group (g), the third set represents other (o), and the fourth set represents all (a). The column will look like this:

rwxrwxrwx

Each set of ***rwx*** represents user, group, and other respectively. Only the owner of a file or a privileged user may change the permissions on a file. There are two ways to change permissions on a file or directory, either numerically or by using lettered commands. Both ways use the command ***chmod***. To add permissions to a file, you use +, to remove permissions you use-.

For example, take a file:

-rw-r--r-- 1 yash mony 476 Apr 14 17:13 vvs.txt

To allow a group (mony, in this case) "write" access, you would type:

chmod g+w vvs.txt

If you wanted to remove "read" ability from "other" you would type:

chmod o-r vvs.txt

It is also possible to specify permissions using a three-digit sequence. This is a more efficient way to change permissions (or at least it requires less typing), so use this method if it doesn't confuse you. Each type of permission is given an octal value. Read is given the value of 4, write is given the value of 2, and execute is given the value of 1. These values are added together for each user category. The permissions are changed by using a three-digit sequence with the first digit representing owner permission, the second digit representing group permission, and the third digit representing other permission. For example, if you wanted to make vvs.txt readable, writable, and executable for the user, readable and writable for the group, and readable for other, you would type:

chmod 764 vvs.txt

The first digit means readable and writable for the user (4+2+1), the second digit means readable and writable for the group (4+2+0), and the third digit means readable for other (4+0+0).

If you want to change the permissions on a directory tree use the -R option. *chmod -R* will recursively change the permissions of directories and their contents.

chown

chown changes who owns a particular file or set of files. New owner files refer to a user ID number or login name that is usually located in the /etc/password directory. The owner of a file or directory can be seen by using the command.

Syntax: ***chown [option] newowner files***

Only the owner of a file or a privileged user can change the permissions on a file or directory. The following example changes the owner of vvs.txt to sridhar

chown sridhar vvs.txt

cp

The ***cp*** command copies files or directories from one place to another. You can copy a set of files to another file, or copy one or more files under the same name in a directory. If the destination of the file you want to copy is an existing file, then the existing file is overwritten. If the destination is an existing directory, then the file is copied into that directory.

Syntax: ***cp [options] file1 file2***

If you want to copy the file ***favourites.html*** into the directory called ***laksh***, you give the command as:

cp favourites.html /vvs/laksh/

A handy option to use with ***cp*** is ***-r***. This recursively copies a particular directory and all of its contents to the specified directory, so you won't have to copy one file at a time.

date

The ***date*** command can be used to display the date or to set a date.

Syntax: ***date [option] [+format]***
date [options] [string]

The first structure shows how date can be used to display the current date. A certain format can be specified in which the date should be displayed. Check the Unix manual for specific formats and options. The second structure allows you to set the date by supplying a numeric string. Only privileged users will be able to use this second command structure.

diff

diff displays the lines that differ between two given files.

Syntax: ***diff [options] [directory options] file1 file2***

diff can be an extremely valuable tool for both checking errors and building new pages. If you run a ***diff*** between two files, you'll be shown what differences the files have line by line. The lines referring to file1 are marked with the < symbol. The lines referring to file2 are marked by the > symbol. If the file is a directory, diff will list the file in the directory that has the same name as file2. If both of the files are directories, diff will list all the lines differing between all files that have the same name.

If you have a file that is not working properly, it can be a great help to check it against a similar file that is working. It will often quickly alert you to a line of code that's missing.

A handy option to use if you want to generally compare two files without noting the complex differences between them is the -h option (h stands for half-hearted). Using -i as an option will ignore differences in uppercase and lowercase characters between files, and -b will ignore repeating blanks and line breaks.

exit

The ***exit*** command allows you to terminate a process that is currently occurring.

For example, if you wanted to leave a remote host that you were logged onto (see rlogin also), you should type exit. This would return you to your home host.

find

find searches through directory trees beginning with each pathname and finds the files that match the specified condition(s). You must specify at least one pathname and one condition.

Syntax: ***find pathname(s) condition(s)***

There are several handy conditions you can use to find exactly what you want. The ***-name*** condition will find files whose names match a specified pattern. The structure for the ***name*** condition is:

find pathname -name pattern

The condition ***-print*** will print the matching files to the pathname specified. ***-print*** can also be used in conjunction with other conditions to print the output.

If you wanted to find all the files named favorites.html in the directory ***Ram***, then you'd do this:

find /Ram -name favorites.html –print

This looks through the directory ***Ram*** and finds all the files in that directory that contain favorites.html, then prints them to the screen. Your output would look like this:

/Ram/sixteen_candles/favorites.html
/Ram/favorites.html
/Ram/breakfast_club/favorites.html

All meta-characters (!, *, ., etc.) used with ***-name*** should be escaped (place a \ before the character) or quoted. Meta-characters come in handy when you are searching for a pattern and only know part of the pattern or need to find several similar patterns. For example, if you are searching for a file that contains the word "favorite", then use the meta-character * to represent matching zero or more of the preceding characters. This will show you all files which contain favorite.

find /Ram -name '*favorite*' -print

This looks through the directory ***Ram*** and finds all the files in that directory that contain the word "favorite". The output would look like this:

/Ram/sixteen_candles/favorites.html
/Ram/favorites.html
/Ram/least_favorites.html
/Ram/breakfast_club/favorites.html
/Ram/favorite_line.html

The ***-user*** condition finds files belonging to a particular user ID or name.

finger

finger displays information about various users as well as information listed in the .plan and .project files in a user's home directory. You can obtain the information on a particular user by using login or last names. If you use the latter, the info on all users with that last name will be printed. Environments that are hooked up to a network recognize arguments (users) in the form of user@host or @ host.

Syntax: ***finger [options] users***

grep

The ***grep*** command searches a file or files for lines that match a provided regular expression ("grep" comes from a command meaning to **g**lobally search for a **r**egular **e**xpression and then **p**rint the found matches).

Syntax: ***grep [options] regular expression [files]***

To exit this command, type 0 if lines have matched, 1 if no lines match, and 2 for errors. This is very useful if you need to match things in several files. If you wanted to find out which files in our ***vvs*** directory contained the word "***mca***" you could use ***grep*** to search the directory and match those files with that word. All that you have to do is give the command as shown:

grep 'mca' /vvs/*

The * used in this example is called a meta-character, and it represents matching zero or more of the preceding characters. In this example, it is used to mean "all files and directories in this directory". So, ***grep*** will search all the files and directories in ***vvs*** and tell you which files contain "***mca"***.

head

head prints the first couple of lines of one or multiple files. ***-n*** is used to display the first ***n*** lines of a file(s). The default number of lines is 10.

Syntax: ***head [-n] [files]***

For example, the following command will display the first 15 lines of favourites.html.
head -15 favourites.html

kill

kill ends the execution of one or more process ID's. In order to do this you must own the process or be designated a privileged user. To find the process ID of a certain job give the command ***ps***.

Syntax: ***kill [options] PIDs***

There are different levels of intensity to the *kill* command, and these can be represented either numerically or symbolically. ***kill -1*** or HUP makes a request to the server to terminate the process, while ***kill -9*** or ***kill KILL*** forces a process to terminate absolutely. Most politely, UNIX users will attempt to kill a process using -1 first before forcing a process to die.

less

less is similar to ***more*** in that it displays the contents of files on your screen. Unlike ***more, less*** allows backward and forward movement within the file. It does not read the whole file before displaying its contents, so with large files less displays faster than more. Press ***h*** for assistance with other commands or ***q*** to quit.

Syntax: ***less [options] [files]***

lprm

lprm removes printer queue requests.

Syntax: ***lprm /usr/ucb/lprm [optons] [job#] [users]***

The ***lprm*** command will remove a job or jobs from a printer's queue. If ***lprm*** is used without any arguments, it will delete the active job if it is owned by the user. If the command is used with **-**, then all the jobs owned by the user will be removed. To remove a specific job, use the job number.

ls

ls will list all the files in the current directory. If one or more files are given, ***ls*** will display the files contained within "name" or list all the files with the same name as "name". The files can be displayed in a variety of formats using various options.

Syntax: ***ls [options] [names]***

ls is a command you'll end up using all the time. It simply stands for list. If you are in a directory and you want to know what files and directories are inside that directory, type ***ls***. Sometimes the list of files is very long and it flies past your screen so quickly you miss the file you want. To overcome this problem give the command as shown below:

ls | more

The character | (called pipe) is typed by using shift and the \ key. | ***more*** will show as many files as will fit on your screen, and then display a highlighted ***"more"*** at the bottom. If you want to see the next screen, hit enter (for moving one line at a time) or the spacebar (to move a screen at a time). **|** ***more*** can be used anytime you wish to view the output of a command in this way.

A useful option to use with ***ls*** command is ***-l***. This will list the files and directories in a long format. This means it will display the permissions (see chmod), owners, group, size, date and time the file was last modified, and the filename.

drwxrwxr-x vvs staff 512 Apr 5 09:34 sridhar.txt
-rwx-rw-r-- vvs staff 4233 Apr 1 10:20 resume.txt
-rwx-r--r-- vvs staff 4122 Apr 1 12:01 favourites.html

There are several other options that can be used to modify the ls command, and many of these options can be combined. ***-a*** will list all files in a directory, including those files normally hidden. ***-F*** will flag filenames by putting / on directories, @ on symbolic links, and * on executable files.

man

The ***man*** command can be used to view information in the online Unix manual.

Syntax: ***man [options] [[section] subjects***]

man searches for information about a file, command, or directory and then displays it on your screen. Each command is a subject in the manual. If no subject is specified, you must give either a keyword or a file. You can also search for commands that serve a similar purpose. For example, if you want more information about the ***chmod*** command, you should type:

man chmod

A screen will then appear with information about ***chmod***. Type ***q*** to quit.

mkdir

mkdir creates a new directory.

Syntax: ***mkdir [options] directory name***

For example, to create a directory called ***parkhyath*** in the present working directory, give the command as,

mkdir prakhyath

more

more displays the contents of files on your screen.

Syntax: ***more [options] [files]***

To have the next line displayed, hit the return key, otherwise press the spacebar to bring up the next screen. Press h for assistance with other commands, n to move to the next file, or q to quit.

mv

mv moves files and directories. It can also be used to ***rename files or directories***.

Syntax: ***mv [options] source target***

If you wanted to rename vvs.txt to vsv.txt, you should give the command as:

mv vvs.txt vsv.txt

After executing this command, vvs.txt would no longer exist, but a file with name vsv.txt would now exist with the same contents.

passwd

The ***passwd*** command creates or changes a user's password. Only the owner of the password or a privileged user can make these changes.

Syntax: ***passwd [options] files***

ps

The ***ps*** command prints information about active processes. This is especially useful if you need to end an active process using the ***kill*** command. Use ***ps*** to find out the process ID number, then use ***kill*** to end the process.

Syntax: ***ps [options]***

pwd

pwd prints the pathname of the current directory. If you wanted to know the path of the current directory you were in you give the command as ***pwd***. You will get the complete path.

rlogin

The ***rlogin*** command, which stands for remote login, lets you connect your local host to a remote host.

Syntax: ***rlogin [options] host***

If you wanted to connect to the remote host ***vsmanyam*** and you were on ***sree***, you would do this:

rlogin vsmanyam password:***

You would then be at ***vsmanyam***

rm

rm removes or deletes files from a directory.

Syntax: ***rm [options] files***

In order to remove a file, you must have write permission to the directory where the file is located. While removing a which does't have write permission on, a prompt will come up asking you whether or not you wish to override the write protection.

The ***-r*** option is very handy and very dangerous. ***-r*** can be used to remove a directory and all its contents. If you use the ***-i*** option, you can possibly catch some disastrous mistakes because it'll ask you to confirm whether you really want to remove a file before going ahead and doing it.

rmdir

rmdir allows you to remove or delete directories but not their contents. A directory must be empty in order to remove it using this command.

Syntax: ***rmdir [options] directories***

If you wish to remove a directory and all its contents, you should use rm -r.

su

su stands for superuser (a privileged user), and can be used to log in as another user. If no user is specified and you know the appropriate password, ***su*** can be used to log in as a superuser.

Syntax: ***su [option] [user] [shell_args]***

tail

The ***tail*** command will print the last ten lines of a file. ***tail*** is often used with the option ***-f***, which tells ***tail*** not to quit at the end of file and instead follow the file as it grows.

Syntax: ***tail [options] [file]***

Use ***ctrl-c*** to exit this command.

telnet

You can communicate with other computers by using the ***telnet*** protocol. The host must be a name or an Internet address. ***telnet*** has two modes: the command mode, which is indicated by the ***telnet > prompt***, and an input mode which is usually a session where you would log on to the host system. The default mode is command mode, so if no host is given it will automatically go into this mode. If you need help while in the command mode, type? or ***help***.

Syntax: ***telnet [host [port]]***

who

The ***who*** command prints out information about the most recent status of the system. If no options are listed, then all of the usernames currently logged onto the system are displayed.

Syntax: ***who [options] [file]***

The option ***am i*** will print the name of the current user. The ***-u*** option will display how long the terminal has been idle.

1.7 INTRODUCTION TO SHELL PROGRAMMING

The UNIX operating system was designed by programmers for programmers. The hardware resources demanded a compact, efficient kernel and flexible file-handling system. UNIX provides tool making tool for programmers with shell. Shells are really interpreted languages, any sequence of commands you wish to run can be placed in a file and run regularly Unix provides, three shells Bourne Shell, C Shell and Korn Shell. User is requested to go through the shell available in your system. This material is assumed to be on Bourne Shell.

1.7.1 History of UNIX Shells

In the beginning there was the Bourne shell /bin/sh (written by S. R. Bourne). It had (and still does) a very strong, powerful syntactical language built into it, with all the features that are commonly considered to produce structured programs; it has particularly strong provisions for controlling input and output and in its expression matching facilities. But no matter how strong its input language is, it had one major drawback; it made nearly no concessions to the interactive user (the only real concession being the use of shell functions and these were only added later) and so there was a gap for something better.

Along came the people from UCB and the C-shell /bin/csh was born. Into this shell they put several concepts which were new, (the majority of these being job control and aliasing) and managed to produce a shell that was much better for interactive use. But as well as improving the shell for interactive use they also threw out the baby with the bath water and went for a different input language.

The theory behind the change was fairly good, the new input language was to resemble C, the language in which UNIX itself was written, but they made a complete mess of implementing it. I/o control problem was solved and bugs appeared. The new shell was simply too buggy to produce robust shell scripts and so everybody stayed with the Bourne shell for that, but it was considerably better for interactive use so changed to the C shell, this resulted in the awkward stupid situation where people use a different shell for interactive work than for non-interactive, a situation which a large number of people still find themselves in today.

Eventually David Korn from AT&T had the bright idea to sort out this mess and the Korn shell /bin/ksh made its appearance. This quite sensibly junked the C shells language and reverted back to the Bourne shell language, but it also added in the many features that made the C shell good for interactive work (you could say it was the best of both worlds), on top of this, it also added some features from other operating systems. The Korn shell became part of System V but had one major problem; unlike the rest of the UNIX shells it wasn't free, you had to pay AT&T for it.

It was at about this time that the first attempts to standardize UNIX started in the form of the POSIX standard. POSIX specified more or less the System V Bourne Shell (by this time the BSD and System V versions had got slightly different). Later the standard is upgraded, and somehow the new standard managed to look very much like ksh.

Also at about this time the GNU project was underway and they decided that they needed a free shell, they also decided that they wanted to make this new shell POSIX compatible, thus bash (the Bourne again shell) was born. Like the Korn shell bash was based upon the Bourne shells language. Bash was quickly adopted for LINUX (where it can be configured to perform just like the Bourne shell), and is the most popular of the free new generation shells.

Meanwhile Tom Duff faced with the problem of porting the Bourne shell to Plan 9, revolts and writes rc instead, he published a paper on it, and Byron Rakitzis re-implemented it under UNIX. With the benefit of a clean start Rc ended up smaller, simpler, more regular and in most peoples opinion a much cleaner shell.

The search for the perfect shell still goes on and the latest entry into this arena is zsh. Zsh was written by Paul Falstad. It is based roughly on the Bourne shell.

1.7.2 Deciding on a Shell

The following are some of the things that are to be considered to decide on which shell to work with.

How much time do I have to learn a new shell?

There is no point in using a shell with a different syntax, or a completely different alias system if you haven't the time to learn it. If you have the time and are presently using *csh* or *tcsh* it is worth considering a switch to a Bourne shell variant.

What do I wish to be able to do with my new shell?

The main reason for switching shells is to gain extra functionality; it's vital you know what you are gaining from the switch.

Do I have to be able to switch back to a different shell?

If you may have to switch back to a standard shell, it is fairly important you don't become too dependent on extra features and so can't use an older shell.

How much extra load can the system cope with?

The more advanced shells tend to take up extra CPU, since they work in *cbreak* mode; if you are on an overloaded machine they should probably be avoided; this can also cause problems with an overloaded network. This only really applies to very old systems nowadays.

What support is given for my new shell?

If your new shell is not supported make sure you have someone you can ask if you encounter problems or that you have the time to sort them out yourself.

Which shell am I using already?

Switching between certain shells of the same syntax is a lot easier than switching between shells of a different syntax. So if you haven't much time a simple upgrade (e.g., csh to tcsh) may be a good idea.

Can I afford any minor bugs?

Like most software all shells have some bugs in them (especially csh), you can afford the problems that may occur because of them.

Do you need to be able to use more than one shell?

If you use more than one system then you may need to know more than one shell at the same time. How different are these two shells and can you manage the differences between them?

1.7.3 Shell Command files

It is possible to repeat a set of commands many times; as if the whole set were just a single command. UNIX allows you to put all your commands in a file in proper order, and then execute them one by one. When you type the file name, shell reads its contents and starts executing the commands in it, one by one. The file is called a shell file and the sequence of commands in it may be called a **shell program** or a **shell procedure**. Let us create a shell file called ***shellp*** (we can give any relevant file name) which contains UNIX commands:

who am i and ***date***

$ cat shellp
who am i
date

To execute this shell program ***shellp,*** first of all we to change the mode of the file using the ***chmod*** command, so that it will be executable.

$ chmod 777 shellp

Then execute the shell program by giving the command as follows:

$ shellp

Let us see how to write the Bourne shell scripts in the following section.

1.8 BOURNE SHELL PROGRAMMING

The focus of this section is to get you to understand and run some Bourne shell scripts. There are example scripts for you to run. Historically, people have been biased towards the Bourne shell over the C shell because in the early days the C shell was buggy. These problems are fixed in many C shell implementations, however many still prefer the Bourne shell. You can write shell programs by creating scripts containing a series of shell commands. The first line of the script should start with **#!** which indicates to the kernel that the script is directly executable. You immediately follow this with the name of the shell, or program (spaces are allowed), to execute, using the full path name. Generally, you can count on having up to 32 characters, possibly more on some systems, and can include one option. So to set up a Bourne shell script the first line would be:

#! /bin/sh
or for the C shell:
#! /bin/csh

You also need to specify that the script is executable by setting the proper bits on the file with **chmod**, e.g.:

% chmod +x shell_script

To introduce ***comments*** within the scripts use **#** (it indicates a comment from that point until the end of the line).

Shell scripting involves chaining several UNIX commands together to accomplish a task. For example, you might run the 'date' command and then use today's date as part of a file name. Let us see how to do this below:

To try the commands below start up a Bourne shell:

Example:

/bin/sh
#A variable stores a string (try running these commands in a Bourne shell)
name= "vvs"
echo $name

The quotes are required in the example above because the string contains a special character (the space)

A variable may store a number as:
num =137

The shell stores this as a string even though it appears to be a number. A few UNIX utilities will convert this string into a number to perform arithmetic:

expr $num + 3

Try defining num as '7m8' and try the *expr* command again
What happens when num is not a valid number?
Now you may exit the Bourne shell with
exit

I/O Redirection

The *wc* command counts the number of lines, words, and characters in a file
wc /etc/passwd
wc -l /etc/passwd

You can save the output of ***wc*** (or any other command) with output redirection
wc /etc/passwd > wc.file

You can specify the input with input redirection
wc < /etc/passwd

Many UNIX commands allow you to specify the input file by name or by input redirection
sort /etc/passwd
sort < /etc/passwd

You can also append lines to the end of an existing file with output redirection
wc -l /etc/passwd >> wc.file

Splitting a file

It may happen that the file you are handling is huge and takes too much time to edit. In such a case you might feel that the file should be split into smaller files. The *split* utility performs this task. Having split a file into smaller pieces, the pieces can be edited singly and then can be concatenated into one whole file again with the ***cat*** command.

To split a file *vvs,* containing 100 lines into 25 lines each, we should give the command as:

$ split - 25 vvs
xaa
xab

xac
xad

Backquotes

The backquote character looks like the single quote or apostrophe, but slants the other way.

It is used to capture the output of a UNIX utility. A command in backquotes is executed and then replaced by the output of the command.

Execute the following commands:
date
save_date= 'date'
echo The date is $save_date

Notice how *echo* prints the output of 'date', and gives the time when you define the *save_date* variable. Store the following in a file named backquotes.sh and execute it (right click and save in a file)

#!/bin/sh
 # Illustrates using backquotes
 # Output of 'date' stored in a variable
Today= 'date'
echo Today is $Today

Execute the script with the following command:

sh backquotes.sh

The above example shows how you can write commands into a file and execute the file with a Bourne shell. Backquotes are very useful, but be aware that they slow down a script if you use them hundreds of times. You can save the output of any command with backquotes, but be aware that the results will be reformatted into one line. Try this:
LS=`ls -l`
echo $LS

Example:

Store the following in a file named *simple.sh* and execute it.
#!/bin/sh
 # Show some useful info at the start of the day
date
echo Good morning $USER
cal
last | head -6

After execution, the output shows current date, calendar, and a six previous logins. Notice that the commands themselves are not displayed, only the results.

To display the commands verbatim as they run, execute with

sh -v simple.sh

Another way to display the commands as they run is with -x

sh -x simple.sh

What is the difference between -v and -x? Notice that with -v you see '$USER' but with -x you see your login name. Run the command 'echo $USER' at your terminal prompt and see that the variable $USER stores your login name. With -v or -x (or both) you can easily relate any error message that may appear to the command that generated it.

When an error occurs in a script, the script continues executing at the next command. Verify this by changing ‘cal’ to ‘caal’ to cause an error, and then run the script again. Run the 'caal' script with 'sh -v simple.sh' and with 'sh -x simple.sh' and verify the error message comes from cal. Other standard variable names include: $HOME, $PATH, $PRINTER. Use echo to examine the values of these variables.

Shell Variables

A variable is a name that stores a string. It’s often convenient to store a filename in a variable. Similar to programming languages, the shell provides the user with the ability to define variables and to assign values to them. A shell variable name begins with a letter (upper or lowercase) or underscore character and optionally is followed by a sequence of letters, underscore characters or numeric characters. The shell gives you the capability to define a named variable and assign a value to it.
The syntax is as follows:

$ variable = value

The value assigned to the variable can then be retrieved by preceding the name of the variable with a dollar sign, that is ***$*** variable. For example,

$ length = 50
$ breadth = 20
$ echo $ length, $ breadth

50 20

The “echo” command produces the output in which the values assigned to the variables are printed.

Let us see another example:
$ message = "please log out within 2 minutes"
$ echo $ message
please logout within 2 minutes

The value assigned to a variable can be defined in terms of another shell variable or even defined in terms of itself.
$ length = 50
$ length = display $ length
$ echo $ length
Display 50

The above can be modified into

$ length = 50
$ length = ${length} value
$ echo $ length
50 value

Store the following in a file named *variables.sh* and execute it.

Example:

```
#!/bin/sh
# An example with variables
filename="/etc/passwd"
echo "Check the permissions on $filename"
ls -l $filename
echo "Find out how many accounts there are on this system"
wc -l $filename
```

Now if we change the value of $filename, the change is automatically propagated throughout the entire script.

Performing Arithmetic

If a shell assigned a numeric value, you can perform some basic arithmetic on the value using the command ***expr***. Let us understand with the help of an example.

$ expr 2 + 3
5

expr also supports subtraction, multiplication and integer division. For example:

$ expr 5 - 6
-1
$ expr 11 '' 4*
44
$ expr 5 / 2
2
$ expr 5%2
1
Another example,
$expr 5 * 7

Backslash required in front of '*' since it is a filename wildcard and would be translated by the shell into a list of file names. You can save arithmetic result in a variable. Store the following in a file named *arith.sh* and execute it

Example:

```
#!/bin/sh
# Perform some arithmetic
x=24
y=4
Result=`expr $x \* $y`
echo "$x times $y is $Result"
```

Comparison Functions

The ***test*** command is used to perform comparisons test will perform comparisons on strings as well as numeric values test will return **1** if the conditions is true and **0** if it is false.

There are three types of operations for which *test* are used. There are numeric comparisons, string comparisons and status test for the file system. To compare two values we use flags, that are placed between the two arguments

Test Operators Flag	Meaning
- eq	True if the numbers are equal
- ne	True if the numbers are not equal
- lt	True if the first number is less than the second number
- le	True if the first numbers is less than or equal to the second number
- gt	True if the first number is greater than the second number
-ge	True if the first number is greater than or equal to the second number.

Let us see some examples:

$ test 5 - eq 4
False
The above example returned false because 5 and 4 are not equal.

$ test abc = Abc
False

For string comparison we use = symbol.

Translating Characters

Prepare a text file namely ***vvs.txt*** in which one of the word's is "fantastic". The utility *tr* translates characters

tr 'a' 'Z' < vvs.txt

This example shows how to translate the contents of a variable and display the result on the screen. Store the following in a file named *tr1.sh* and execute it.

Example:

```
#!/bin/sh

# Translate the contents of a variable

name= "fantastic"
echo $name | tr 'a' 'i'
```

Execute the script and see the output.

This example shows how to change the contents of a variable.

Store the following in a file named *tr2.sh* and execute it.

Example:

```
#!/bin/sh

# Illustrates how to change the contents of a variable with tr

name= "fantastic"
echo "name is $name"
name=`echo $name | tr 'a' 'i'`
echo "name has changed to $name"
```

You can also specify ranges of characters. This example converts upper case to lower case

tr 'A-Z' 'a-z' < file

Now you can change the value of the variable and your script has access to the new value

Looping constructs

Executing a sequence of commands on each of several files with for loops

Store the following in a file named *loop1.sh* and execute it.
Note: You have to have three files namely simple.sh , variables.sh and loop1.sh in the current working directory.

Example;

```
#!/bin/sh
# Execute ls and wc on each of several files
# File names listed explicitly
for filename in simple.sh variables.sh loop1.sh
do
        echo "Variable filename is set to $filename..."
        ls -l $filename
        wc -l $filename
done
```

This executes the three commands: echo, ls and wc for each of the three file names. You should see three lines of output for each file name. Filename is a variable, set by "for" statement and referenced as $filename. Now we know how to execute a series of commands on each of several files.

Using File Name Wildcards in For Loops

Store the following in a file named loop2.sh and execute it.

Example:

```
#!/bin/sh
# Execute ls and wc on each of several files
# File names listed using file name wildcards
for filename in *.sh
do
        echo "Variable filename is set to $filename..."
        ls -l $filename
        wc -l $filename
done
```

You should see three lines of output for each file name ending in *.sh*. The file name wildcard pattern **.sh* gets replaced by the list of filenames that exist in the current directory. For another example with filename wildcards try this command:

*echo *.sh*

Search and Replace in Multiple Files

Sed performs global search and replace on a single file.
Note: To execute this example you need to have required files.

sed -e 's/example/EXAMPLE/g' sdsc.txt > sdsc.txt.new

The original file ***sdsc.txt*** is unchanged. How can we arrange to have the original file over-written by the new version? Store the following in a file named s-and-r.sh and execute it.

Example:

```
#!/bin/sh
# Perform a global search and replace on each of several files
# File names listed explicitly
for text_file in sdsc.txt nlanr.txt
do
      echo "Editing file $text_file"
      sed -e 's/example/EXAMPLE/g' $text_file > temp
      mv -f temp $text_file
done
```

First, ***sed*** saves new version in file *temp.* Then, use mv to overwrite original file with new version.

Command-line Arguments

Command-line arguments follow the name of a command. For example:

ls -l .cshrc /etc

The command above has three command-line arguments as shown below:

-l	(an option that requests long directory listing)
.cshrc	(a file name)
/etc	(a directory name)

An example with file name wildcards:
*wc *.sh*

How many command-line arguments were given to wc? It depends on how many files in the current directory match the pattern ***.sh***. Use 'echo *.sh' to see them. Most

UNIX commands take command-line arguments. Your scripts may also have arguments.

Store the following in a file named *args1.sh*

Example:

```
#!/bin/sh
# Illustrates using command-line arguments
# Execute with
#             sh args1.sh On the Waterfront
echo "First command-line argument is: $1"
echo "Third argument is: $3"
echo "Number of arguments is: $#"
echo "The entire list of arguments is: $*"
```

Execute the script with

sh args1.sh -x On the Waterfront

Words after the script name are command-line arguments. Arguments are usually options like *-l* or *file names.*

Looping Over the Command-line Arguments

Store the following in a file named args2.sh and execute it.

Example:

```
#!/bin/sh
# Loop over the command-line arguments
# Execute with
#           sh args2.sh simple.sh variables.sh
for filename in "$@"
do
            echo "Examining file $filename"
            wc -l $filename
done
```

This script runs properly with any number of arguments, including zero. The shorter form of the *for* statement shown below does exactly the same thing:

```
for filename
do
...
```

But, don't use:

*for filename in $**

It will fail if any arguments include spaces. Also, don't forget the double quotes around $@.

***If* construct and *read* command**

Read one line from ***stdin***, store line in a variable.

read variable_name

Ask the user if he wants to exit the script. Store the following in a file named ***read.sh*** and execute it.

Example:

```
#!/bin/sh
```

```
# Shows how to read a line from stdin
echo "Would you like to exit this script now?"
read answer
if [ "$answer" = y ]
then
	echo "Exiting..."
	exit 0
fi
```

Command *Exit* Status

Every command in UNIX should return an exit status. Status is in range 0-255.Only **0** means success. Other statuses indicate various types of failures. Status does not print on screen, but is available through variable $?.

The following example shows how to examine exit status of a command.

Example:

Store the following in a file named ***exit-status.sh*** and execute it.

```
#!/bin/sh
# Experiment with command exit status
echo "The next command should fail and return a status greater than zero"
ls /nosuchdirectory
echo "Status is $? from command: ls /nosuchdirectory"
echo "The next command should succeed and return a status equal to zero"
ls /tmp
echo "Status is $? from command: ls /tmp"
```

Example given below shows if block using exit status to force exit on failure.

Store the following in a file named ***exit-status-test.sh*** and execute it.

Example:

```
#!/bin/sh
# Use an if block to determine if a command succeeded
echo "This mkdir command fails unless you are root:"
mkdir /no_way
if [ "$?" -ne 0 ]
then
	# Complain and quit
	echo "Could not create directory /no_way...quitting"
	exit 1  # Set script's exit status to 1
fi
echo "Created directory /no_way"
```

Regular Expressions

For searching zero or more characters we use the wild characters.*. Let's see the examples:

*grep 'provided.*access' sdsc.txt*
*sed -e 's/provided.*access/provided access/' sdsc.txt*

To search for text at beginning of line we give the command,
grep '^the' sdsc.txt

To search for text at the end of line we give the command,
grep 'of$' sdsc.txt

Asterisk means zero or more the preceding characters.

a* zero or more a's
aa* one or more a's
aaa* two or more a's

Examples:
Delete all spaces at the ends of lines
sed -e 's/ *$//' sdsc.txt > sdsc.txt.new

Turn each line into a shell comment
sed -e 's/^/# /' sdsc.txt

The *case* statement

The next example shows how to use a *case* statement to handle several contingencies. The user is expected to type one of three words. A different action is taken for each choice.

Store the following in a file named ***case1.sh*** and execute it.

Example:

```
#!/bin/sh
# An example with the case statement
# Reads a command from the user and processes it
echo "Enter your command (who, list, or cal)"
read command
case "$command" in
        who)
                echo "Running who..."
                who
                ;;
        list)
                echo "Running ls..."
                ls
                ;;
        cal)
                echo "Running cal..."
                cal
                ;;
        *)
                echo "Bad command, your choices are: who, list, or cal"
                ;;
esac
exit 0
```

The last case above is the default, which corresponds to an unrecognised entry. The next example uses the first command-line arg instead of asking the user to type a command.

Store the following in a file named ***case2.sh*** and execute it.

Example:

```
#!/bin/sh
# An example with the case statement
# Reads a command from the user and processes it
# Execute with one of
#       sh case2.sh who
#       sh case2.sh ls
#       sh case2.sh cal
echo "Took command from the argument list: '$1'"
case "$1" in
        who)
                echo "Running who..."
```

```
        who
        ;;
    list)
        echo "Running ls..."
        ls
        ;;
    cal)
        echo "Running cal..."
        cal
        ;;
    *)
        echo "Bad command, your choices are: who, list, or cal"
        ;;
esac
```

The patterns in the case statement may use file name wildcards.

The *while* statement

Example given below loops over two statements as long as the variable *i* is less than or equal to ten. Store the following in a file named while1.sh and execute it.

Example:

```
#!/bin/sh
# Illustrates implementing a counter with a while loop
# Notice how we increment the counter with expr in backquotes
i="1"
while [ $i -le 10 ]
do
    echo "i is $i"
    i=`expr $i + 1`
done
```

The example given below uses a *while* loop to read an entire file. The *while* loop exits when the read command returns false exit status (end of file). Store the following in a file named ***while2.sh*** and execute it.

Example:

```
#!/bin/sh
# Illustrates use of a while loop to read a file
cat while2.data |   \
while read line
do
        echo "Found line: $line"
done
```

The entire *while* loop reads its *stdin* from the pipe. Each *read* command reads another line from the file coming from *cat*. The entire *while* loop runs in a subshell because of the pipe. Variable values set inside while loop not available after *while* loop.

1.9 PRACTICAL SESSIONS

Session 1

1) Explore all the UNIX commands given in this manual.
2) Create a directory.
3) Create a subdirectory in the directory created.
4) Change your current directory to the subdirectory.
5) Display the calendar for the current month.
6) Get a directory listing of the parent directory.
7) How many users were logged onto your system?
8) Display your name in the form of a banner.
9) Display the name of device name of your terminal.
10) Move to the root directory.

Session 2

11) Change your directory to the directory ***exercises***. Create a file called ***example1*** using the cat command containing the following text:

    water, water everywhere
    and all the boards did shrink;
    water, water everywhere,
    No drop to drink.

12) Use the man command to obtain further information on the finger command.
13) List all the processes that are presently running.
14) List the text files in your current directory.
15) Make a copy of any text file.
16) Rename one of your text files in the current directory.
17) Delete an unneeded copy of a file.
18) Print out any file on paper.
19) Send a message to another user on your UNIX system, and get them to reply.
20) Create a small text file and send it to another user.

Session 3

21) When you receive a message, save it to a file other than your mailbox.
22) Send a message to a user on a different computer system.
23) Try to move to the home directory of someone else in your group. There are several ways to do this, and you may find that you are not permitted to enter certain directories. See what files they have, and what the file permissions are.
24) Try to copy a file from another user's directory to your own.
25) Set permissions on all of your files and directories to those that you want. You may want to give read permission on some of your files and directories to members of your group.
26) Create a number of hierarchically related directories and navigate through them using a combination of absolute pathnames (starting with "/") and relative pathnames.
27) Try using wildcards ("*" and possibly "?").
28) Put a listing of the files in your directory into a file called *filelist*. (Then delete it!)
29) Create a text file containing a short story, and then use the ***spell*** program to check the spelling of the words in the file.
30) Redirect the output of the ***spell*** program to a file called ***errors***.

Session 4

31) Type the command ***ls -l*** and examine the format of the output. Pipe the output of the command ***ls -l*** to the word count program wc to obtain a count of the number of files in your directory.
32) Use cut to strip away the reference material and leave just the text field.
33) Use tr to strip away any tags that are actually in the text (e.g., attached to the words), so that you are left with just the words.
34) Set a file to be read-only with the **chmod** (from *ch*ange *mod*e) command. Interpret the file permissions displayed by the **ls -l** command.
35) Delete one or more directories with the **rmdir** (from *rem*ove *dir*ectory) command. See what happens if the directory is not empty. Experiment (carefully!) with the **rm -r** command to delete a directory and its content.
36) Experiment with redirecting command output (e.g., **ls -l >file1**). Try ">> " instead of " >" with an existing text file as the output.
37) See whether upper-case versions of any of these commands work as well as the lower-case versions.
38) Use the who command to see users logged into the system.
39) Pipe the output of the ***who*** command to the **sort** command
40) Search for your login name in **whofile** using the **grep** command.

Session 5

41) Compare two text files with the **diff** command.
42) Count lines, words, and characters in a file with the **wc** command.
43) Display your current environment variables with the following command: **set** or **env.**
44) Concatenate all files in a directory redirected to /dev/null and redirecting standard error to "errorFile"?
45) Display information on yourself or another user with the **finger** command.
46) If you wish, experiment with sending and receiving mail using the **pine** email program.
47) Delete all the files in the current directory whose name ends in ".bak".

48) Display lines 10 to 14 of any file which contains 25 lines.

49) Count how many lines contain the word *science* in a word file ***science.txt***.

50) List the statistics of the largest file (and only the largest file) in the current directory.

Session 6

51) Kill any process with the help of the PID and run any process at the background.

52) Select a text file and double space the lines.

53) List all the users from /etc/passwd in the alphabetically sorted order.

54) Create a file with duplicate records and delete duplicate records for that file.

55) Use the ***grep*** command to search the file ***example1*** for occurrences of the string "water".

56) Write grep commands to do the following activities:

- To select the lines from a file that have exactly two characters.
- To select the lines from a file that start with the upper case letter.
- To select the lines from a file that end with a period.
- To select the lines in a file that has one or more blank spaces.
- To select the lines in a file and direct them to another file which has digits as one of the characters in that line.

57) Make a sorted wordlist from the file.

58) Try to execute the example shell scripts given in this manual.

59) Write a shell script that searches for a single word pattern recursively in the current directory and displays the no. of times it occurred.

60) Write a shell script to implement the DISKCOPY command of DOS.

Session 7

61) Write a shell script that accepts a string from the terminal and echo a suitable message if it doesn't have at least 5 characters including the other symbols.

62) Write a shell script to echo the string length of the given string as argument.

63) Write a shell script that accepts two directory names as arguments and deletes those files in the first directory which are similarly named in the second directly. Note: Contents should also match inside the files.

64) Write a shell script to display the processes running on the system for every 30 seconds, but only for 3 times.

65) Write a shell script that displays the last modification time of any file.

66) Write a shell script to check the spellings of any text document given as an argument.

67) Write a shell script to encrypt any text file.

68) Combine the above commands in a shell script so that you have a small program for extracting a wordlist.

69) Write a shell script which reads the contents in a text file and removes all the blank spaces in them and redirects the output to a file.

70) Write a shell script that changes the name of the files passed as arguments to lowercase.

Session 8

71) Write a shell script to translate all the characters to lower case in a given text file.

72) Write a shell script to combine any three text files into a single file (append them in the order as they appear in the arguments) and display the word count.

73) Write a shell script that, given a file name as the argument will write the even numbered line to a file with name ***evenfile*** and odd numbered lines to a file called ***oddfile***.

74) Write a shell script which deletes all the even numbered lines in a text file.

75) Write a script called hello which outputs the following:

- your username
- the time and date
- who is logged on
- also output a line of asterices (*********) after each section.

76) Put the command hello into your .login file so that the script is executed every time that you log on.

77) Write a script that will count the number of files in each of your subdirectories.

78) Write a shell script like a more command. It asks the user name, the name of the file on command prompt and displays only the 15 lines of the file at a time on the screen. Further, next 15 lines will be displayed only when the user presses the enter key / any other key.

79) Write a shell script that counts English language articles (a, an, the) in a given text file.

80) Write the shell script which will replace each occurrence of character ***c*** with the characters ***chr*** in a string ***s***. It should also display the number of replacements.

Session 9

81) Write the shell program ***unique***, which discards all but one of *successive* identical lines from standard input and writes the unique lines to standard output. By default, ***unique*** checks the whole line for uniqueness.

For example, assuming the following input:

List 1

List 2

List 2

List 3

List 4

List 4

List 2

unique should produce the following output as follows:

List 1

List 2

List 3

List 4

List 2

82) Rewrite the ***unique*** program so that it can optionally accept a file name on the command line and redirect the output to that file.

83) Write the shell program which produces a report from the output of **`ls -l`** in the following form:

- Only regular files, directories and symbolic links are printed.
- The file type and permissions are removed.
- A / character is appended to each directory name and the word DIR is printed at the beginning of the line.
- A @ character is appended to each symbolic link name and the word LINK is printed at the beginning of the line.

At the end of the listing, the number of directories, symbolic links, regular files and the total size of regular files should be reported.

84) Write the shell program which removes all the comments from a simple C program stored in your current directory. You can assume that the C source code contains only syntactically correct comments:

- starting with `//`, ending with a newline
- starting with `/*`, ending with `*/` (can be multi-line)
- nesting of comments is not allowed.

Make sure that C source code is not changed.

85) Write a shell program that outputs all integers up to the command line parameter starting from 1 and also should output the same numbers in the reverse order.

Session 10

86) Write a shell program to concatenate to two strings given as input and display the resultant string along with its string length.

87) Write a shell program to find the largest integer among the three integers given as arguments.

88) Write a shell program to sort a given file which consists of a list of numbers, in ascending order.

89) Write a shell program to simulate a simple calculator.

90) Write a shell program to count the following in a text file.

- Number of vowels in a given text file.
- Number of blank spaces.
- Number of characters.
- Number of symbols.
- Number of lines

1.10 SUMMARY

UNIX is a popular operating System, which is mostly using the C language, making it easy to port to different configurations. UNIX programming environment is unusually rich and productive. It provides features that allow complex programs to be built from simpler programs. It uses a hierarchical file system that allows easy maintenance and efficient implementation. It uses a consistent format for files, the byte stream, making application programs easier to write. It is a multi-user, multitasking system. Each user can execute several processes simultaneously. It hides the machine architecture from the user, making it easier to write programs that run on different hardware implementation. It is highly secured system.

1.11 FURTHER READINGS

1) Behrouz A. Forouzan, Richard F. Gilberg, *UNIX and Shell Programming*, Thomson, 2003.

2) Brian W. Kernighan, Rob Pike, *The UNIX Programming Environment*, PHI, 1996

3) K. Srirengan, *Understanding UNIX*, PHI, 2002

4) Sumitabha Das, *Your UNIX- The Ultimate Guide*, TMGH, 2002

5) Sumitabha Das, *UNIX Concepts and Applications, Second Edition*, TMGH, 2002

1.12 WEBSITE REFERENCES

www.unix.org/

www.unixreview.com/

www.ugu.com/sui/ugu/warp.ugu

unixhelp.ed.ac.uk

www.unix.org/resources.html

www.osnews.com/

www.levenez.com/unix/

cnc.k12.mi.us/websites/bsdtree.html/

www.linux.org/

www.kernel.org/

www.linux.com/

SECTION 2 DBMS LAB

Structure **Page Nos.**

2.0 INTRODUCTION

Information Technology (IT) has influenced organisational performance and competitive standing. The increasing processing power and sophistication of analytical tools and techniques require a strong foundation of structured form of data. Thus, the presence of a commercial, secure, reliable database management system is the major requirement of businesses.

This section is an attempt to familiarise you with commercial database management features, so that you are able to apply some of the important features of DBMS in a commercial situation. This would require a sound background of the concepts of DBMS which can be acquired through the courses MCS-023: Introduction to DBMSs and MCS-043: Advanced DBMSs. You would also have to apply the conceptual background to obtain problem solving skills using a commercial DBMS. The topics that have been included in this section are sub-queries, joins, views, indexes, embedded languages including cursors, triggers, and some advanced features like object tables, nested tables, user management etc.

Please note that we have given many examples in this lab section in different sessions, highlighting the basic syntax of various statements; however, they are not exhaustive, as commercial DBMSs implement a large number of statements and functions. Therefore, you must refer to the user reference manuals of the commercial DBMS for more details on these topics. Also, we have given the examples for a typical DBMS; however, you may do practicals on any commercial RDBMS supporting such concepts. However, you should find the changes in the statements used here for the DBMS you may be using. Remember, the objectives of this section consisting of ten sessions will be completed only if you *practice* using any commercial DBMS.

2.1 OBJECTIVES

After going through this section and performing all the practical exercises you should be able to:

- write and run SQL queries using join;
- create and run sub-queries, views, indexes and procedures;
- write simple embedded SQL related programs;
- handle errors and manage transactions;
- write simple triggers;
- create advanced tables and objects, and
- perform simple user management.

2.2 SESSION 1: SUBQUERIES AND JOINS

Objectives

At the end of this session, you should be able to:

- define subqueries and joins
- create different types of subqueries
- perform different types of join queries.

The queries given in this session and most of the later sessions are based on the following relations:

teacher (t_no, f_name, l_name, salary, supervisor, joiningdate, birthdate, title)
class (class_no, t_no, room_no)
payscale (Min_limit, Max_limit, grade)

2.2.1 Subquery

Subquery means a SELECT statement which retrieves the values from the table and passes these values as argument to the main or calling SELECT statement. In simple words, nested queries are subqueries. Subqueries can be of different types:

1) Single-row subquery
2) Multiple-row subquery
3) Multiple-column subquery
4) Correlated query.

Single-Row Subquery

Single-row subquery is the query that returns only one value or row to the calling SELECT statement. For example:

Query 1: Display the name of the teacher who is oldest among all teachers.
Explanation: To know the name of the teacher who is oldest, you need to first find the minimum birth date and then corresponding to that date display the name of the teacher.

```
SELECT f_name, l_name
FROM teacher
WHERE birthdate = (SELECT MIN(birthdate)
                   FROM teacher);
```

Query 2: Display teacher numbers and names of those teachers who are earning less than 'Jatin'.
Explanation: To find the list of teachers earning less than 'Jatin', you need to find first the salary of 'Jatin'.

```
SELECT t_no, f_name, l_name
FROM teacher
WHERE salary < (SELECT salary
                FROM teacher
                WHERE UPPER(f_name) = 'JATIN');
```

Multiple-Row Subquery

Multiple-row subqueries are the queries that return only more than one value or rows to the calling SELECT statement. For example:

Query 3: Display the list of all teachers who are earning equal to any teacher who have joined before '31-dec-94'
Explanation: First you need to know the salaries of all those who have joined before '31-dec-94' and then any teacher whose salary matches any of these returned salaries. IN operator looks for this existence into the set. You can also use Distinct to avoid duplicate salary tuples.

```
SELECT t_no, f_name, l_name
FROM teacher
WHERE salary IN  (SELECT salary
                  FROM teacher
                  WHERE joindate < '31-dec-94');
```

Query 4: Display the list of all those teachers whose salary is greater than any other teacher with job title 'PRT'.
Explanation: First you need to know the salaries of all those who are 'PRT' and then any teacher whose salary is greater than any of these returned salaries. ANY operator looks for inequality.

```
SELECT t_no, f_name, l_name, salary
FROM teacher
WHERE salary >   ANY (SELECT salary
                      FROM teacher
                      WHERE UPPER (title) = 'PRT');
```

Query 5: Display the list of all those teachers whose salary is greater than all the teachers with job title as 'PRT'.
Explanation: First you need to know the salaries of all those who are 'PRT' and then any teacher whose salary is greater than all of these returned salaries. ALL operator looks for inequality.

```
SELECT t_no, f_name, l_name, salary
FROM teacher
WHERE salary > ALL (SELECT salary
```

```
FROM teacher
WHERE UPPER(title) = 'PRT');
```

Single-row Subquery		Multiple-row subquery	
Operator	**Description**	**Operator**	**Description**
=	Equal to	IN	Returns true if any of the values in the list match i.e. equality check
>	Greater than		
<	Less than	ALL	Returns true if all the values returned by the subquery match the condition
>=	Greater than equal to		
<=	Less than or equal to	ANY	Returns true if any of the values returned by subquery match the condition.
<>	Not equal to		

Figure 1: Operators for subqueries

Multiple-Column subquery

Multiple-Column subquery is the subquery that returns more than one column to the calling or main SELECT statement. For example:

Query 6: Display the list of all teachers whose job title and salary is same as that of the employee whose first name is 'Jaideep'.
Explanation: Firstly you need to find the job title and salary of 'Jaideep' and then you need to find all other teachers whose job title and salary exactly matches Jaideep's job title and salary.

```
SELECT t_no, f_name,l_name, title, salary
FROM teacher
WHERE (title, salary) = (SELECT title, salary
                         FROM teacher
                         WHERE LOWER(f_name) = 'jaideep');
```

Correlated Subqueries

This is another type of subquery where the subquery is executed for each and every record retrieved by the calling or main SELECT statement. A correlated subquery returns only a Boolean value (true/false). A *true* is returned if the subquery returns one or more records otherwise a *false* is returned. The operator EXISTS can be used for these types of subqueries. For example:

Query 7: Display the records in the format given below for all class teachers:

Jaideep Kumar is a class teacher

Explanation: The main query uses the EXISTS operator to find whether the teacher is a class teacher or not. If the correlated subquery returns a true value, then the record retrieved by main SELECT statement is accepted otherwise it is rejected.

```
SELECT f_name, l_name, 'is a class teacher'
FROM teacher
WHERE exists (SELECT *
              FROM class
              WHERE class.t_no = teacher.t_no)
```

2.2.2 Joins

Joins means retrieving data from more than one table in a single query. There are several types of joins:

1) **Equijoin** – In this type of join, two or more tables are joined over common columns and common values. For example:

   Query 8: Display names of all the teachers who are class teachers.

```
SELECT f_name, l_name
FROM teacher t, class c
WHERE t.t_no = c.t_no;
```

2) **Non-Equijoin -** In this type of join, two or more tables do not have common columns and common values but they are joined indirectly. For example:

Query 9: Display names, salaries and salary grades of all teachers.

```
SELECT f_name, l_name, salary, grade
FROM teacher, payscale
WHERE salary BETWEEN min_limit AND max_limit;
```

3) **Outer join -** In this type of join, two or more tables are joined over common column but the records may or may not have common values. For example:

Query 10: Display names and class numbers of all the teachers. In addition display the classes of those teachers who are class teachers. Thus, the result should include names of teachers who are not class teachers.

```
SELECT f_name, l_name, class_no
FROM teacher t, class c
WHERE t.t_no = c.t_no (+) ;
```

4) **Self join -** In this type of join, the table is joined to itself. For example:

Query 11: Display teacher number and names of all teachers along with the names of their supervisors and number. Please note that the supervisor of a teacher is also a teacher. Therefore, the query requires a self-join.

```
SELECT t.f_name, t.l_name, t.supervisor, s.f_name, s.l_name
FROM teacher t, teacher s
WHERE t.supervisor = s.t_no;
```

5) **Cartesian join** – In this type of join, each record of one table is joined to each record of the other table i.e., the join condition is not given. For example:

Query 12: Show all possible teacher – class values.

```
SELECT f_name, l_name, class_no
FROM teacher t, class c;
```

2.2.3 Exercise 1

1) Please attempt the following problems for the teacher, class and payscale relations given in this section.

   (a) Display the name of the teacher(s) who is (are) the youngest among all the teachers.

   (b) Display details of all the teachers who have the same job title as that of 'Jaideep'

   (c) Display the list of all the teachers who have joined after '10-Jul-95' and whose salary is equal to that of any of the teachers who joined before '10-Jul-95'.

   (d) Use a correlated query to determine the teachers who are not class teachers.

   (e) Identify all those teachers who are in grade 'B'.

   (f) Display the names and numbers of all teachers who are class teachers and are in grade 'C'.

   (g) Display the names of all teachers who are supervisors.

   (h) Display the teacher id and salaries of all those teachers who are in grade 'A' or 'C' and who have at least two L's in their names.

(i) Display details of all those teachers who are class teachers of classes 1 to 5.

(j) Display the names of all teachers along with their dates of birth whose birthday is in the current month.

2) In an Open University a student's data is to be maintained using relations. The university maintains data of all the students, their batches, Regional Centres and study centre details. Each batch of students has one or more representative students. The batches are taught by same or different faculty.

(a) Design and implement the suitable relations for the University. Make and state suitable assumptions.

(b) Write at least 15 queries (they must include at least one query of each type given in this section 2.2) for the database. Implement these queries using SQL. Also explain the purpose of each query.

3) Design a suitable database system for a bank along with 20 possible queries to the database (the queries should be such that their solution involves subqueries or joins or both). Implement the database and the queries in a commercial DBMS using SQL.

2.3 SESSION 2: CREATING VIEWS, INDEXES AND QUERIES ON THEM

Objectives

At the end of this session, you will be able to :

- define database objects such as views and indexes
- create views and use views
- create and maintain indexes.

The views help you to simplify your complex queries on the tables. Also they increase the secrecy of data. To speed up the performance of queries, you should create indexes on one or more columns. The discussion in this session is based on the following relations:

Teacher(t_no, f_name, l_name, salary, supervisor, joiningdate, birthdate, title)
Class(class_no, t_no, room_no)
Payscale(Min_limit, Max_limit, grade)

2.3.1 Creating a view

Views are actually virtual tables that are derived from base tables at runtime. A view after creation can be used as any other table but a view does not store data and it derives data only when it is called. The CREATE VIEW command is used to create views on the tables.

```
Syntax:  CREATE VIEW [or replace] [force | noforce] AS
         Select statement
         [with check option [constraint]]
         [with read only]
```

or replace: Replaces the existing view with the new one.
force: Allows the view to be created even if the tables don't exist.
No force: Allows the view to be created only when the tables exist.
With check option: Allows the DML operations only on the rows accessible to the view.
With read only: Does not allow any DML operation using view. For example:

Example 1: Create a view that displays teacher number, the names of teachers along with salary, job title, age and grade.

```
CREATE VIEW  teacher_details
        AS  SELECT t_no, f_name||' '||l_name as name, title, salary,
              trunc(months_between(sysdate,birthdate)/12,0) age, grade
    FROM teacher , payscale
    WHERE salary BETWEEN min_limit AND max_limit;
```

Let us make a few queries on this view.

Query (a): Display teacher number, their names, age and grade of all PGT teachers.

```
SELECT t_no, name, age, grade
FROM teacher_details
WHERE  UPPER (title) = 'PGT';
```

Query (b): Display details of all the teachers who are more than 40 years old.

```
SELECT t_no, name, salary, title, age
FROM teacher_details
WHERE age > 40;
```

To remove a view, DROP statement is used. For example:

```
DROP VIEW teacher_details;
```

2.3.2 Indexes

Index is the Oracle object that reduces the record-retrieval time. An index contains the Rowid and the values of the target columns. A rowid is a pointer that determines the location of the record. Indexes can be based on one or more table columns. Two types of indexes can be created:

1. Unique indexes - These indexes are created automatically whenever a PRIMARY KEY or UNIQUE constraint is defined.
2. Non-unique indexes - These indexes are created explicitly by the user on non-unique keys.

Syntax:

```
CREATE INDEX index_name
ON  tablename (column1 , column2 ,………);
```

Example 2: Create an index on the relation teacher on the job title for fast access.

```
CREATE INDEX t_title_index
ON teacher(title);
```

To remove an index, drop statement is used. For example,

```
DROP INDEX t_title_index;
```

2.3.3 Exercise 2

1) Please attempt the following problems for the teacher, class and payscale relations given in this section.

   (a) Create a view named 'supervisor_details' that stores the names and numbers of all the supervisors.

   (b) Create a non-unique index on the foreign key column of the 'class' table.

   (c) Modify the view created in (a) and add details like salary, job title, joining date, birth date etc. of all supervisors.

   (d) Using the view created in (c) display details of all supervisors who have worked for more than 15 years.

   (e) Create a view that stores the details of all the teachers who are TGT and earning more than Rs.12000/-.

   (f) Drop the view created in (e).

(g) Create a non-unique index on the names of teachers in the 'teachers' table.

(h) Drop the index created in (b).

(i) Create a view named 'teacher_info' that only allows users to view the teacher id, name, salary and grade of the teachers. It should not allow any user to change/update any information.

(j) Create a view that displays details of all teachers who are in grade 'B' and are more than 40 years old.

2) Design suitable views for the University database system (Exercise 1, question 2). For example, you can create a view for the faculty giving him/her details of his/her students. Create at least 5 suitable queries on each of the views created by you. Also create indexes for the University database system. For example, you can create an index on student name, thus, allowing faster access to student information in the order of student names. You must explain how the indexes created by you would enhance the performance of the database system. Implement the above in a commercial DBMS using SQL.

3) Design suitable views and indexes for the Bank database system (Exercise 1, question 3). Create at least 5 suitable queries on each of the views. Explain how the indexes created by you would enhance the performance of the database system. Implement the above in a commercial DBMS using SQL.

2.4 SESSION 3: PL/SQL – CONTROL LOOPS AND PROCEDURES

Objectives

At the end of this session, you should be able to:

- use procedural constructs like LOOP
- define and use FOR loop, While loop
- define and use procedures.

PL/SQL extends SQL by adding control loops found in procedural languages, resulting in a structural language that is more powerful than SQL. We will discuss the use of PL/SQL in this session.

2.4.1 Control loops

Loops are the constructs that help to repeat the statement groups on certain conditions. There are three kinds of looping constructs:

- LOOP
- WHILE
- FOR

LOOP Structure

It is the basic loop structure. It is used to repeat statement groups without providing a condition to come out of loop. This structure begins with the keyword LOOP and ends with the keyword END LOOP. To come out of the loop, EXIT statement is used to break the most current loop and the control is passed to the next statement after the END LOOP statement. This basic loop structure allows execution of statement group at least once even if the condition is already true when entering the loop. Let us explain this with the help of examples.

Example 1: A loop that will execute 100 times may be:

```
DECLARE
  i NUMBER := 0;
```

```
BEGIN
  LOOP
     i := i + 1;
     IF i = 100 THEN
        EXIT;
     END IF;
  END LOOP;
END;
```

Example 2: Another way of writing example 1

```
DECLARE
   I   NUMBER := 0;
BEGIN
   LOOP
      I := I + 1;
      EXIT WHEN I = 100;
   END LOOP;
END;
```

Example 3: This example shows insertion of values in 'sample' table.

```
DECLARE
    I NUMBER := 1;
BEGIN
    LOOP
       INSERT INTO sample VALUES(I, I);
       I := I * 5;
       EXIT WHEN I >100;
    END LOOP;
END;
```

FOR Loops

This loop structure is an extension of the basic LOOP structure along with an additional control statement tagged at the front of the LOOP keyword. By default the loop increments the control variable starting from the lower index and going up to the higher index value. The REVERSE clause is used to make the loop decrement the control variable from highest to lowest index value. For example:

Example 4: Using for loop for a counter from 1 to 100.

```
DECLARE
    a NUMBER := 1;
BEGIN
    FOR counter IN 1 .. 100
    LOOP
       a := a + (counter * 5);
       EXIT WHEN a > 10000;
    END LOOP;
END;
```

Example 5: In this example, the 'counter' will start from 100 and reach 1.

```
DECLARE
    a NUMBER := 1;
    lower_index := 1;
    upper_index := 100
BEGIN
     FOR counter IN REVERSE lower_index .. upper_index
     LOOP
```

```
    a := a + (counter * 5);
  END LOOP;
END;
```

WHILE Loops

This loop structure is also an extension of basic LOOP and the statements are iterated based upon a condition test. If condition is *true* then the loop is executed otherwise not.

Example 6: A quantity can be issued if the total amount is below a sanctioned amount.

```
DECLARE
  qty NUMBER := 1;
  sanctioned_amt  NUMBER := 1000;
  unit_price NUMBER := 10;
  tot_amt NUMBER := 0;
BEGIN
    WHILE tot_amt < sanctioned_amt
    LOOP
        tot_amt := unit_price * qty;
        qty := qty + 1;
    END LOOP;
END;
```

2.4.2 Procedures

PL/SQL procedure is a named program block i.e., logically grouped set of SQL and PL/SQL statements that perform a specific task. They can be invoked or called by any PL/SQL block. The syntax is:

```
CREATE OR REPLACE PROCEDURE name
(argument {IN, OUT, INOUT} data type, ……..)
IS
    Variables declarations;
BEGIN
    PL/SQL subprogram body
EXCEPTION
    Exception PL/SQL block
END procedure name;
```

Replace: if procedure is already created then replace it.
IN : Indicates that the parameter will accept a value from the user.
OUT : Indicates that the parameter will return a value to the user.
INOUT : Indicates that the parameter will either accept from or return a value to the user.

Example 7: Following procedure searches the name of a teacher in the relation *teacher*

```
CREATE OR REPLACE PROCEDURE search_teacher
        (o_t_no IN NUMBER,
         o_f_name OUT VARCHAR2,
         o_l_name OUT VARCHAR2)
IS
BEGIN
        SELECT f_name, l_name
        FROM teacher
        WHERE t_no = o_t_no;
END search_teacher;

run;  /* it is used to create the procedure */
```

To call this procedure:

```
DECLARE
o_f_name  teacher.f_name%TYPE;
o_l_name teacher.l_name%TYPE;
BEGIN
    search_teacher( 113, o_f_name, o_l_name);
    DBMS_OUTPUT.PUT_LINE('Employee : 113');
    DBMS_OUTPUT.PUT_LINE('Name :  '|| o_f_name || '  ' || o_l_name);
END;
```

Example 8: To demonstrate use of INOUT parameter

```
CREATE PROCEDURE bonus_calc
        (o_t_no IN INTEGER,
         bonus INOUT INTEGER)
IS
        join_date DATE;
BEGIN
        SELECT salary * 0.20, joiningdate INTO bonus, join_date
         FROM teacher
        WHERE t_no = o_t_no;
        IF MONTHS_BETWEEN(SYSDATE, join_date) > 36 THEN
            bonus := bonus + 1000;
        END IF;
END;
```

Important Notes:

1) Unlike the type specifier in a PL/SQL variable declaration, the type specifier in a parameter declaration must be unconstrained i.e.`CHAR` or `VARCHAR2` should be used instead of `CHAR(5)` or `VARCHAR2` (20).

2) The name of the procedure after the END is optional.

3) A constant or a literal argument should not be passed in for an OUT/INOUT parameter.

2.4.3 Exercise 3

1) Please perform the following using the following relations:

Teacher(t_no, f_name, l_name, salary, supervisor, joiningdate, birthdate, title)
Class(class_no, t_no, room_no)
Payscale(Min_limit, Max_limit, grade)

(a) Calculate the bonus amount to be given to a teacher depending on the following conditions:

I. if salary > 10000 then bonus is 10% of the salary.
II. if salary is between 10000 and 20000 then bonus is 20% of the salary.
III. if salary is between 20000 and 25000 then bonus is 25% of the salary.
IV. if salary exceeds 25000 then bonus is 30% of the salary.

(b) Using a simple LOOP structure, list the first 10 records of the 'teachers' table.

(c) Create a procedure that selects all teachers who get a salary of Rs.20, 000 and if less than 5 teachers are getting Rs.20, 000 then give an increment of 5%.

(d) Create a procedure that finds whether a teacher given by user exists or not and if not then display “teacher id not exists”.

(e) Using FOR loop, display name and id of all those teachers who are more than 58 years old.

(f) Using while loop, display details of all those teachers who are in grade ‘A’.

(g) Create a procedure that displays the names of all those teachers whose supervisor is ‘Suman’.

(h) Calculate the tax to be paid by all teachers depending on following conditions:

I. if annual salary > 1,00,000 then no tax.

II. if annual salary is between 1,00,001 and 1,50,000 then tax is 20% of the annual salary.

III. if annual salary is between 1,50,001 and 2,50,000 then tax is 30% of the annual salary.

IV. if salary exceeds 2,50,000 then tax is 40% of the annual salary.

(i) Create a procedure that finds the names of all teachers with the job title ‘PRT’ and if the number of teachers returned is more than 10 than change the job title to ‘TGT’ for the top 3 ‘PRT’ teachers based on their hiredate.

2) Identify the need of procedures for the University database system; for example, you can create a procedure that awards 2% grace marks for those students who have got 48% marks. Design at least 5 such procedures. Implement these procedures using an embedded SQL.

3) Implement at least five procedures for the Bank Database system using embedded SQL.

2.5 SESSION 4: CURSORS

Objectives

At the end of this session, you should be able to:

- define implicit and explicit cursors.
- perform open, fetch and close operations on the explicit cursors
- create cursor *FOR* loops.

A private work area for an SQL statement is called cursor. The cursors can be implicit or explicit. You can perform several major actions on the explicit cursors. Let us discuss them in more detail.

2.5.1 Cursors

When SQL statement is executed in the PL/SQL block, the Oracle Engine assigns a private work area for that statement. This work area is private to the SQL statement i.e., it stores the statement and the result after execution and is called a *cursor.* Cursors can either be created implicitly or explicitly.

1. Implicit cursors

When an SQL command is executed from within a PL/SQL block, PL/SQL creates an implicit cursor. PL/SQL provides some attributes that allow you to evaluate what happened when the implicit cursor was last used.

2. Explicit cursors

An explicit cursor is defined as a SELECT statement within PL/SQL block. The general syntax is:

```
CURSOR cursorname IS
   SELECT statement;
```

There are a number of actions performed on explicit cursors. They are: -

ACTION	DESCRIPTION	EXAMPLE
DECLARE	It defines the name and structure of the cursor using the SELECT statement.	CURSOR cur_sample IS SELECT * FROM teacher;
OPEN	It executes the query that populates the cursor with rows.	OPEN cur_sample;
FETCH	It loads the row currently referred by the cursor pointer into variables or record and moves the cursor pointer to the next row ready for the next fetch.	FETCH cur_sample INTO teacher_rec **OR** FETCH cur_sample1 INTO cur_t_no, cur_salary;
CLOSE	It releases the data within the cursor and closes it. The cursor can be reopened again.	CLOSE cur_sample;

The SQL cursor attributes that can be used with implicit or explicit cursors are:

ATTRIBUTE	DESCRIPTION
%ROWCOUNT	Returns the number of rows processed by an SQL statement.
%FOUND	Returns TRUE if at least one row was processed.
%NOTFOUND	Returns TRUE if no rows were processed.
%ISOPEN	Return TRUE if cursor is open or FALSE, if cursor has not been opened or has been closed. Only used with explicit cursors.

Example 1: Implicit cursor

```
DECLARE
      total_row_del NUMBER;
BEGIN
      DELETE * FROM teacher WHERE salary < 10000;
      total_row_del := SQL%ROWCOUNT;
END;
```

Example 2: Explicit Cursor

```
DECLARE
     c_t_no     teacher.t_no%TYPE;
     c_f_name  teacher.f_name%TYPE;
     c_l_name  teacher.l_name%TYPE;
     c_salary  teacher.salary%TYPE;
     CURSOR  c1 IS
          SELECT t_no,f_name, l_name, salary
          FROM teacher;
BEGIN
     OPEN c1;
     LOOP
            FETCH c1 INTO c_t_no, c_f_name, c_l_name, c_salary ;
            EXIT WHEN NOT c1%FOUND;
            UPDATE teacher SET salary = salary * 1.10 WHERE salary > 20000;
      END LOOP;
      CLOSE c1;
END;
```

Example 3: Explicit Cursor using record name in FOR loop

```
DECLARE
     CURSOR  c2 IS
          SELECT t_no,f_name, l_name, salary
          FROM teacher ;
```

```
    teacher_rec  c2%ROWTYPE;
BEGIN
    OPEN c2;
    FOR teacher_rec  IN c2
    LOOP
        IF teacher_rec.salary > 20000
            Teacher_rec.title = “SUPERVISOR”;
        ENDIF;
    END LOOP;
    CLOSE c2;
END;
```

2.5.2 Exercise 4

1) Please perform the following using the following relations:

Teacher(t_no, f_name, l_name, salary, supervisor, joiningdate, birthdate, title)
Class(class_no, t_no, room_no)
Payscale(Min_limit, Max_limit, grade)

(a) Create a host language block to declare a cursor for displaying teacher numbers and their names for all teachers having title ‘PGT’.

(b) Create a host language block using a cursor to calculate bonus for teachers as 5% of their salary. Display on screen the teacher details along with the bonus given.

(c) Write a host language block to delete all the rows from the ‘teacher’ table where the salary is less than Rs.5000.

(d) Write a host language code to insert the supervisor information from ‘teacher’ table to another table called ‘supervisor’. The new table should have only those records where the job title is ‘supervisor’.

(e) Write a block in host language that deletes all the rows from ‘teacher’ table if the teacher was hired for more than 10 years.

(f) Write a block in host language using cursor that displays the names of all teachers who will attain the age of 60 years in the current year.

(g) Write a block in host language using cursors that display teacher details along with the tax to be paid by that teacher. The tax is calculated depending on following conditions:

I. if annual salary < 1,00,000 then no tax.

II. if annual salary is between 1,00,001 and 1,50,000 then tax is 20% of the annual salary.

III. if annual salary is between 1,50,001 and 2,50,000 then tax is 30% of the annual salary.

IV. if salary exceeds 2,50,000 then tax is 40% of the annual salary.

(h) Write a block in host language that displays the details of all those teachers who have reached maximum limit of their grade.

2) Write at least four embedded SQL blocks having cursors for the University database system; for example, you can create a cursor to update the examination marks of a student that are given in a list to the student’s database. Implement these procedures using an embedded SQL.

3) Implement at least five embedded SQL blocks having cursors for the Bank Database system.

2.6 SESSION 5: ERROR HANDLING AND TRANSACTION MANAGEMENT

Objectives

At the end of this session you should be able to:

- perform basic error handling
- use commit, rollback and save point specifications.

2.6.1 Error Handling

Exceptions are identifiers that are raised during the execution of a PL/SQL block to terminate its action in case of a problem, error or abnormal condition. A block is always terminated when PL/SQL raises an exception but we can also capture exceptions by defining our own error handlers and perform some final actions before quitting the block.

There are two classes of exceptions. These are:

1) Predefined: It basically refers to Oracle predefined errors which are associated with specific error codes. For example: CURSOR_ALREADY_OPEN, NO_DATA_FOUND.

2) User-defined: These are defined by the user and raised when specifically requested within a block. If an error occurs within a block PL/SQL passes control to the EXCEPTION section of the block. If no EXCEPTION section exists in the block or the EXCEPTION section cannot handle the error then the block is terminated with an unhandled exception. Exceptions execute through nested blocks until an exception handler is found that can handle the error. If no exception handler is found in any block the error is passed out to the host environment. Exceptions occur implicitly when either an Oracle error occurs or you explicitly raise an error using the RAISE statement.

Example 1: Predefined Exception

```
DECLARE
        C_id  teacher.t_no%TYPE;
        C_f_name teacher.f_name%TYPE;
        want_id NUMBER := 110;
BEGIN
         SELECT t_no, f_name INTO c_t_no, c_f_name from teacher
         WHERE t_no = want_id;
           DBMS_OUTPUT.PUTLINE ( “teacher : “|| c_t_no ||’  ‘||c_f_name)
EXCEPTION
           WHEN INVALID_NUMBER THEN
            DBMS_OUTPUT.PUTLINE(want_id || ‘ not a valid teacher id’);
END;
```

Example 2: User Defined Exception

```
DECLARE
    C_title    teacher.title%TYPE;
    CURSOR  c_teacher  IS
       SELECT  t_no,f_name, l_name, title
       FROM teacher;
```

```
    C_teacher_row   c_teacher%ROWTYPE;
BEGIN
    C_title := 'PGT';
    FOR c_teacher_row IN c_teacher
    LOOP
        EXIT WHEN c_teacher%NOTFOUND;
        DECLARE
            emptytitle EXCEPTION;
        BEGIN
            IF c_teacher_row IS NULL THEN
                RAISE emptytitle;
            ELSE
                DBMS_OUTPUT.PUTLINE (c_teacher_row.f_name,
c_teacher_l_name, c_teacher_title);
            ENDIF;
            EXCEPTION
            WHERE emptytitle THEN
                DBMS_OUTPUT.PUTLINE(c_teacher_row.f_name);
        END;
    END LOOP;
    CLOSE c_teacher;
END;
```

2.6.2 Transaction Management

A transaction is a logical unit of work that is composed of one or more Data Manipulation Language (DML) or Data Definition Language (DDL) or Data Control Language (DCL) statements. A transaction starts when an SQL statement is executed and terminates when any of the following occurs:

- Any DDL or DML statement is executed.
- A COMMIT or ROLLBACK statement is given
- User exits SQL.

The transactions can be committed implicitly or explicitly. All DDL or DCL statements are implicitly committed.

There are three explicit transaction specifications:

1) COMMIT: It ends the current transaction and saves permanently all pending changes since last saved or COMMIT or ROLLBACK. The syntax is:

   COMMIT;

2) SAVEPOINT: It marks and saves the current point of the transaction process. It is useful to perform partial rollbacks. The syntax is:

   SAVEPOINT name;

3) ROLLBACK: It ends the transaction but discards all pending changes since last commit or save point. The syntax is:

   ROLLBACK [TO SAVEPOINT name];

Example:

```
SQL >  UPDATE teacher   SET   title = 'PGT' WHERE salary > 15000;
SQL>   COMMIT
                              /* Changes made by Update statement are saved */
SQL> INSERT INTO class(class_no, t_no, room_no) VALUES (10,127,226);
SQL> ROLLBACK;
```

```
                               /* values inserted into 'class' table are discarded */
SQL> UPDATE teacher   SET   title = 'SUPERVISOR' WHERE salary > 20000;
SQL> SAVEPOINT  update_done;
                              /* savepoint created */
SQL> DELETE FROM teacher;
SQL> ROLLBACK TO SAVEPOINT  update_done;
                               /* just delete statement is discarded */
```

2.6.3 Exercise 5

1) Write an embedded SQL block along with exceptions to select the name of the teacher with a given salary. If more than one row is returned than display more than one row retrieved. If no row is returned then display 'no teacher with this salary'.
2) Create a program that updates a record into the 'teacher' table. Trap and handle the exception if the teacher id is not available in the 'teacher' table.
3) Write a program with exceptions that displays the details of all those teachers who have reached maximum limit of their grade. If no row is retrieved then raise the exception "no teacher reached max limit of grade".
4) Insert at least 5 new rows in the 'teacher' table and then try to rollback last 3 rows inserted to the table. (Here, you are required to use save points).
5) Write a program with exceptions that displays the names of all teachers who will attain the age of 60 years in the current year. If no row is retrieved than display suitable exception.
6) Write a PL/SQL block that displays all the rows from 'teacher' table if the teacher was hired for more than 10 years and still a 'PRT'. If no result than display suitable message.
7) In all the embedded SQL program segments that you have created so far for the University and Bank database system, create suitable error handling features.
8) Experiment with DDL and DML commands along with COMMIT and ROLLBACK for the Teacher, University and Banking databases.

2.7 SESSION 6: TRIGGERS AND FUNCTIONS

Objectives

At the end of this section, you should be able to:

- define triggers and functions
- define and create statement and row triggers
- create functions with or without parameters.

For the following discussion, we will consider the following relations:
Teacher(t_no, f_name, l_name, salary, supervisor, joiningdate, birthdate, title)
Class(class_no, t_no, room_no)
Payscale(Min_limit, Max_limit, grade)

2.7.1 Triggers

A trigger is PL/SQL block stored in the database. It is executed by an event that occurs to a database table. Triggers are implicitly invoked by DML commands like INSERT, UPDATE or DELETE. But you cannot use COMMIT, ROLLBACK and SAVEPOINT statements within trigger blocks. Triggers can be called BEFORE or AFTER the events. Triggers can be of two types:

1) STATEMENT type - STATEMENT triggers fire BEFORE or AFTER the execution of the statement that caused the trigger to fire.
2) ROW type - ROW triggers fire BEFORE or AFTER any affected row is processed.

Example 1: A STATEMENT trigger for not allowing work on weekends.

```
CREATE OR REPLACE TRIGGER not_working_hour
   BEFORE DELETE OR INSERT OR UPDATE ON teacher
BEGIN
   IF (TO_CHAR(SYSDATE, 'DY') IN ('SAT', 'SUN'))
   THEN
   DBMS_OUTPUT.PUTLINE ('Sorry! you cannot delete/update/insert on
                                             weekends');
    END IF;
END;
```

Example 2: A ROW trigger for supplying new teacher id and system date as the joining date.

```
CREATE OR REPLACE TRIGGER new_teacher _id
    AFTER INSERT ON  teacher
FOR EACH ROW
DECLARE
    o_t_no  teacher.t_no%TYPE;
   o_joiningdate teacher.joiningdate%TYPE;
BEGIN
   SELECT  t_no_sequence.nextval
      INTO  o_t_no
      FROM  dual;
   :NEW.t_no := o_t_no;
   :NEW.joiningdate := SYSDATE;
END;
```

2.7.2 Functions

Functions are similar to procedures but in function definitions you must explicitly return a value to the calling block via the RETURN statement. A function can have zero, one or more parameters.

Example 3: Find the grade of a teacher (one parameter function).

```
CREATE OR REPLACE FUNCTION get_grade (o_t_no  IN NUMBER)
IS o_grade VARCHAR2(20);
BEGIN
  SELECT grade  INTO o_grade FROM Payscale, teacher
     WHERE t_no = o_t_no  AND salary between min_limit AND max_limit;
  RETURN (o_grade);
END get_grade;
```

This function can be called from PL/SQL block as:

```
o_teacher_grade := get_grade(o_teacher_no);
```

Example 4: Function without parameter

```
CREATE OR REPLACE FUNCTION welcome_note
        RETURN  VARCHAR2
IS
BEGIN
   RETURN  'Good Morning!';
END welcome_note;
```

This function can be called from PL/SQL block as:

```
a : =  welcome_note;
DBMS_OUPUT.PUTLINE (a);
```

To drop a function:

```
DROP FUNCTION <function_name>;
```

2.7.3 Exercise 6

1) Write a trigger that is fired before the DML statement's execution on the TEACHER table. The trigger checks the school timings based on SYSDATE. Beyond the School working hours the trigger raise an exception, which does not allow any work to be happened.
2) Write a trigger that is fired before an UPDATE statement is executed for the teacher table. The trigger should write the name of teacher, user name and system date in an already created table called UPDATE_TABLE.
3) Write a trigger that is fired before any row is inserted in the 'teacher' table.
4) Write a function and pass a job title to it. If the TEACHER table does not contain any row corresponding to that title then return false otherwise true.
5) Write a trigger that verifies the joining date when a new row is inserted in the 'teacher' table. Joining date should be greater or equal to current date.
6) Write a function that gets the teacher id as parameter and returns the class number associated with that teacher. If the teacher is not a class teacher then give suitable message.
7) Write a function and pass a teacher id to it. If the TEACHER table does not contain that id then return false otherwise true.
8) Write a function that takes teacher id as parameter and returns back the name and joining date of the teacher.
9) Write appropriate triggers and functions for the University and Bank database systems.

2.8 SESSION 7: CREATING OBJECT TYPES/TABLES

Objectives

At the end of this session, you should be able to:

- define objects and object type.
- create and use object types.
- create object methods

2.8.1 Objects

An object in ORACLE is a reusable component representing real-world entities. An object can be defined using user-defined data type called an object type that is further used to define object columns in the tables. An object contains a name, attribute and methods. The methods can be used to perform data manipulation on the object tables.

Defining Types

Oracle defines the types similar to the types of SQL.

Syntax:

```
CREATE [ OR REPLACE ] TYPE <name> AS OBJECT
                                          <attribute name> datatype, ..
MEMBER PROCEDURE | FUNCTION <procedure or function spec>,...,
```

```
        [ MAP | ORDER MEMBER FUNCTION <comparison function
        spec>,
            ... ]
        [ PRAGMA RESTRICT_REFERENCES (<what to restrict>,
        restrictions) ]
        );
/
```

Note: The slash at the end processes the type definition in ORACLE.

In the type definition as above:

OR REPLACE – Replace if definition already exists

Name - Oracle identifier
attribute name - A legal PL/SQL identifier for the attribute.

Data type - Any legal Oracle data type *except* LONG, LONG RAW, NCHAR, NCLOB, NVARCHAR2, ROWID, BINARY_INTEGER, BOOLEAN, PLS_INTEGER, RECORD, REF CURSOR, %TYPE, %ROWTYPE, or types that exist only within packages.

Comparison function - Defines a function that allows comparison of object values.

What to restrict - This is either the name of the function or procedure, or the keyword DEFAULT. Using DEFAULT tells Oracle that *all* member functions and procedures in the object type will have the designated restrictions, without having to list each one in its own RESTRICT_REFERENCES pragma.

Restrictions - One or more of the following: RNDS, WNDS, RNPS, and WNPS.

FORCE++ -Tells Oracle that you want to drop a type even if there are other objects with dependencies on it. Even if you use FORCE, you can only drop a type if it has not been implemented in a table; you must first drop the table(s) before dropping the type.

Example 1: Declaring a point type consisting of two numbers:

```
CREATE TYPE pointtype AS OBJECT (
   a NUMBER,
   b NUMBER
);
/
```

This object type can be used like any other type in further declarations of object-types or table-types.

Example 2: Defining a line type by using previously created object type.

```
CREATE TYPE linetype AS OBJECT (
   x  pointtype,
   y  pointtype
);
/
```

These objects can further be used to create a table that is a set of lines with ``line ID's'' a

Example 3: Create object table

```
CREATE TABLE lines (
   line_id  INT,
   line   linetype
);
```

Dropping Types

To remove a type, we use a DROP statement as shown below:

```
DROP TYPE linetype;
```

Note: Before dropping a type, drop all tables and other types that use this type.

Constructing Object Values

Oracle provides built-in constructors for values of a declared type with the name of the type.

Example 4 : Inserting a new row in the Object table 'line' for a line from co-ordinates (0,0) to (4,5).

```
INSERT INTO lines
  VALUES (12, linetype(
                    point type(0.0, 0.0),
                    point type(4.0, 5.0)
                )
            );
```

Declaring and Defining Methods

The object methods can be declared by `MEMBER FUNCTION` or `MEMBER PROCEDURE` in the `CREATE TYPE` statement.

Example 5:

```
CREATE TYPE linetype AS OBJECT (
  x  pointtype,
  y  pointtype,
  MEMBER FUNCTION length(scale IN NUMBER) RETURN NUMBER,
       PRAGMA RESTRICT_REFERENCES (length, WNDS)
);
/
```

Note : The 'pragma' means the `length` function will not modify the database (WNDS means 'write no database state'). It is necessary when you want to use `length` in queries.

Queries on Object tables

The values of the components of an object are accessed with the dot notation. Also, to access fields of an object, you need to give alias of the table name.

Example 6: Find the length of all the lines for scale factor 3.

```
SELECT line_id, line_alias.line.length(3.0)
 FROM lines line_alias;
```

Note: 'line_alias' is table alias.

2.8.2 Exercise 7

1) Assuming that in the teacher relation an attribute of object type called dependent is added. Make dependent a type that may consist of only one dependent. Add few records in the tables and output them using a query on teacher name. Find if you can search on a dependent name.

2) Create at least two object types in both the University and Bank database systems. Use these object types in few relations and enter some data into these relations. Query these databases.

2.9 SESSION 8: NESTED TABLES

Objectives

At the end of this session, you should be able to:

- define and create nested tables
- initialize nested tables.

2.9.1 Nested Table

Nested table is defined as table within another table. Nested tables are single-dimensional arrays of elements that are of same type such as a table of numbers or characters. In nested tables, each row can have multiple rows itself.

Example:

```
CREATE  TYPE new_table  AS   OBJECT
(subject_name  VARCHAR2(25),
credit_hours NUMBER(5));

CREATE TYPE  new_type AS TABLE OF new_table;
```

This will create an object-type table named 'new table'.

Example: To use this nested object table.

```
CREATE TABLE student_credits
(rollno  NUMBER(5),
s_name VARCHAR2(25),
subject_credits  NEW_TYPE)
NESTED TABLE subject_credits STORE AS new_type_table;
```

Example: To insert values in the above created table,

```
INSERT INTO student_credits
VALUES (100, 'suman' , new_table ( new_type ('english' , 30),
                       new_table ( new_type('hindi', 35)));
```

To query from a nested table Oracle provides a new function THE. Function. THE is used to select data from a field in the nested table for which we must have to flatten the table first.

Example:

```
SELECT  s.credit_hours FROM
   THE (SELECT subjects_credit FROM student_credits
         WHERE s_name = 'suman')  s
WHERE s.subject_name = 'english';
```

The nested tables can also be created as

```
CREATE  TYPE new_table  AS   OBJECT
my_table  new_table;
```

This definition creates un-initialised objects. Generally, for these types of un-initialized object type tables, Oracle generates an error that will cause an exception to be raised. These tables can be initialized by either initializing the whole table at declaration time or by initializing partially at declaration time and then later using EXTEND method to define new or extra elements.

Example 1: Initializing at declaration time.

```
 DECLARE
    TYPE new_table IS TABLE OF NUMBER ;
    my_table new_table := new_table (2);
 BEGIN
    my_table (1)  := 13;
    my_table (2)  := 15;
```

```
    DBMS_OUTPUT.PUT_LINE('my_table(1) is '||my_table(1));
    DBMS_OUTPUT.PUT_LINE('my_table(2) is '||my_table(2));
 END;
/
```

The table can also be initialized as:

my_table new_table := new_table (5,9,7,11,4,10);

This will create a new table with 6 elements.

Example 2: Partial initialisation and then using EXTEND

```
 DECLARE
   TYPE new_table IS TABLE OF NUMBER ;
   my_table new_table ;
BEGIN
    my_table := new_table();
   my_table.EXTEND(2);
   my_table(1) := 13;
   DBMS_OUTPUT.PUT_LINE('my_table(1) is '||my_table(1));
   DBMS_OUTPUT.PUT_LINE('my_table(2) is '||my_table(2));
END;
/
```

Note: You do not have a control over where these elements are created because Oracle automatically appends them to the existing table .

Following are the some of the functions that can be used with nested tables.

Functions	**Description**
COUNT	returns the number of elements in the collection
DELETE	deletes one or more elements in the collection - an optional start and end point specify which elements are to be deleted
EXISTS(n)	determines if the specified element has been created and not deleted, returns TRUE if the element exists, FALSE if not
FIRST	returns the subscript of the first element in the nested table
LAST	returns the subscript of the last element in the nested table
PRIOR	returns the subscript of the previous element in the nested table
NEXT	returns the subscript of the next element in the nested table

2.9.2 Exercise 8

1) Add a nested table in the teacher relation. Do some queries using nested tables?

2) Create at least two nested tables for both the University and Bank database systems. Use these tables and enter some data into these relations. Query these databases.

2.10 SESSION 9: STORING BLOBS

Objectives

At the end of this session, you should be able to:

- define LOBs

- types of LOBs – BLOB, CLOB, NCLOB, BFILE
- use blob data types in tables.

2.10.1 BLOBS (Large objects)

LOBs are data types that are capable of storing large volumes of unstructured data like images, sound clips, large texts etc. An LOB data can hold data ranging from 8 terabytes to 128 terabytes depending on how the database is configured. The use of LOBs enables you to access and manipulate data efficiently in your application. LOB data types allow random access to data. The LOB data types now available are BLOB, CLOB, NCLOB, and BFILE.

1) **CLOB(character large object)** stores large blocks of single-byte character data. Generally, used for large strings or documents that use the database character set exclusively.
2) **BLOB(binary large object)** stores large binary objects inside or outside row. Typically used for multimedia data such as images, audio and video.
3) **NCLOB (National Character Set Large Object)** stores string data in National character set format. Used for large strings or documents in the National character set.
4) **BFILE (External Binary File)** A binary file is stored outside the database in the host operating system file system but it can be accessible from database tables. This type of LOB can be accessed from your application on a **read-only** basis. Generally, BFILEs are used to store static data, such as image data, that does not need to be manipulated in applications.

LOB stands for Locator Object, which indicates that it is a pointer to that place where the actual data is stored. This locator will only occupy some bytes in the row.

Example 1:

```
CREATE TABLE message (
   msg_id  NUMBER(8) NOT NULL PRIMARY KEY,
      email_add      VARCHAR(200),
      name           VARCHAR (200),
      message        CLOB,
      posting_time   DATE,
      sort_key       VARCHAR (600));
```

LOBs are accessed via a LOB or BFILE "locator". The techniques you use when accessing a cell in a LOB column differ depending on the state of the given cell. A cell in a LOB Column can be in one of the following states:

- **NULL**: The table cell is created, but the cell holds no locator or value.
- **Empty:** A LOB instance with a locator exists in the cell, but it has no value. The length of the LOB is zero.
- **Populated:** A LOB instance with a locator and a value exists in the cell.

Use built-in functions EMPTY_CLOB() for CLOBs and NCLOBs, EMPTY_BLOB() for BLOBs, and BFILENAME() method to initialize a BFILE column to point to an OS file.

Example 2: Use of BLOB s

```
DECLARE

 Image10        BLOB;
 image_number     INTEGER := 101;
```

```
BEGIN
  SELECT item_blob INTO image10 FROM lob_table10
      WHERE key_value = image_number;
  DBMS_OUTPUT.PUT_LINE('Image size
is:'||DBMS_LOB.GETLENGTH(image10));
    ------
    ------
    ------
    ------
END;
```

Example 3: Inserting lob values in tables.

```
INSERT INTO message1
VALUES (101, 1, EMPTY_BLOB(),'my Oracle', EMPTY_CLOB(),NULL,
    BFILENAME('dir_object', 'my_image'), NULL);
```

Example 4: Selecting blob values from tables.

```
SELECT COUNT (*) FROM message1 WHERE image_graphic IS NOT NULL;
  SELECT COUNT (*) FROM message1 WHERE image_graphic IS NULL;
```

2.10.2 Exercise 9

1) Identify the use of large object types in the teacher's table. Do some queries using these objects.
2) Create at least two large objects for both the University and Bank database systems. Enter some data into these relations. Query these databases.

2.11 SESSION 10: USER MANAGEMENT AND OBJECT PRIVILEGES

Objectives

At the end of this session, you should be able to:

- define and create user accounts
- define the types of privileges
- give and withdraw the privileges.

2.11.1 User Management

Before the database is used by a number of users, they should be given privileges on their user groups. By default, Oracle has two accounts: SYS and SYSTEM. The SYSTEM account is used to give privileges to the users and to create new user accounts.

The user account can be created by the CREATE USER command.

Example 1: Create a user account 'manager' with password 'pass'.

```
CREATE USER manager
    IDENTIFIED BY pass;
```

These user accounts are of use when the privileges are given to them. There are two types of privileges:

1) System privilege: This kind of privilege allows a user to use DDL Commands like CREATE TABLE, DROP INDEX etc.
2) Object privilege: This kind of privilege allows a user to use DML Commands like UPDATE, INSERT etc.

To give and take back the privileges to user accounts, we use GRANT and REVOKE command respectively.

Example 2: Give CREATE and DROP privileges related to table/view to the user account 'manager'.

GRANT CREATE TABLE, DROP TABLE, CREATE VIEW, DROP VIEW TO manager;

Example 3: Withdraw DROP privilege related to table/view given to the user account 'manager'.

REVOKE DROP TABLE, DROP VIEW FROM manager;

2.11.2 Exercise 10

1) Create a user account "class" and give privileges related to table/view creation, deletion, updating and dropping.
2) Create a student account and give permission to this account for only viewing the information on the relation *Class (class_no, t_no, room_no.*
3) Create at least 3 to 4 different types of users for each of the database systems: University and Bank. Design suitable access privileges for the users. Grant these permissions to the users.
4) Consider when you have a large number of students and teachers in the University databases that have different access rights. Is there any mechanism in DBMS that allows defining an account type to some specific role? If yes, then define such types for the University database system.
5) Define different types of users for the Bank database and provide them suitable access rights.

2.12 SUMMARY

This section has tried to broadly cover about ten different aspects of a commercial DBMS. We have covered practical sessions relating to sub-queries and join based queries, creating views, indexes and performing queries on them, embedded SQL with or without the use of cursors, error handling during the embedded SQL operations and using COMMIT and ROLLBACK commands used for transactions. We have also done sessions on triggers and functions, creating types, nested tables, BLOBs and user management.

You should try to solve not only the problems given in these exercises, but also many more problems during your practical sessions. You should consult online help/reference manuals of the commercial DBMS you are using while solving the problems. Practice is the key for achieving the basic objectives of the sessions.

Finally, please keep in mind the fact that the basic objective of this section is NOT to make you an expert DBA, but it is to give you a feel of many different types of things that you may be able to do with a DBMS. You should look forward to applying your theoretical knowledge of MCS-023 and MCS-043 along with the skills obtained during the practical sessions to create solutions with respect to the needs of various business organisations.