Relational Databases Review

Terminal, Shell, and Command Line Basics

- **Command line**: A text interface where users type commands.
- **Terminal**: The application that provides access to the command line.
- **Terminal emulator**: Adds extra features to a terminal.
- **Shell**: Interprets the commands entered into the terminal (e.g., Bash).
- **PowerShell / Command Prompt / Microsoft Terminal**: Options for accessing the command line on Windows.
- **Terminal (macOS)**: Built-in option on macOS, with third-party alternatives like iTerm or Ghostty.
- **Terminal (Linux)**: Options vary by distribution, with many third-party emulators like kitty.
- **Terminology**: Though "terminal," "shell," and "command line" are often used interchangeably, they have specific meanings.

Command Line Shortcuts

- **Up/Down arrows**: Cycle through previous/next commands in history.
- **Tab**: Autocomplete commands.
- `Control+L` (Linux/macOS) or typing `cls` (Windows): Clear the terminal screen.
- `Control+C`: Interrupt a running command (also used for copying in PowerShell if text is selected).
- `Control+Z` (Linux/macOS only): Suspend a task to the background; use `fg` to resume it.
- `!!`: Instantly rerun the last executed command.

Bash Basics

- **Bash** (Bourne Again Shell): Widely used Unix-like shell.
  Key commands:

  - `pwd`: Show the current directory.
  - `cd`: Change directories.
    - `..` refers to the parent directory (one level up).
    - `.` refers to the current directory.
  - `ls`: List files and folders.
    - `-a`: Show all files, including hidden files.
    - `-l`: Show detailed information about files.
  - `less`: View file contents one page at a time with navigation options, including scrolling backward and searching.
  - `more`: Display file contents one screen at a time, with limited backward scrolling and basic navigation.
  - `cat`: Show the entire file content at once without scrolling or navigation, useful for smaller files.
  - `mkdir`: Create a new directory.
  - `rmdir`: Remove an empty directory.
  - `touch`: Create a new file.
  - `mv`: Move or rename files.
    - Rename: `mv oldname.txt newname.txt`
    - Move: `mv filename.txt /path/to/target/`

- `cp`: Copy files.
  - `-r`: Recursively copy directories and their contents.
- `rm`: Delete files.
  - `-r`: Recursively delete directories and their contents.
- `echo`: Display a line of text or a variable's value.
  - Use `>` to overwrite the existing content in a file. (e.g., `echo "text" > file.txt`)
  - Use `>>` to append output to a file **without overwriting existing content** (e.g., `echo "text" >> file.txt`).
- `exit`: Exit the terminal session.
- `clear`: Clear the terminal screen.
- `find`: Search for files and directories.
  - `-name`: Search for files by name pattern (e.g., `find . -name "*.txt"`).
- Use `man` followed by a command (e.g., `man ls`) to access detailed manual/help pages.

Command Options and Flags

- **Options** or **flags**: modify a command's behavior and are usually prefixed with hyphens:
  - **Long form (two hyphens)**:
    - Example: `--help`, `--version`
    - Values are attached using an equals sign, e.g., `--width=50`.
  - **Short form (one hyphen)**:
    - Example: `-a`, `-l`
    - Values are passed with a space, e.g., `-w 50`.
    - Multiple short options can be chained together, e.g., `ls -alh`.
- `--help`: You can always use a command with this flag to understand the available options for any command.

Introduction to Relational Databases

- **Relational Databases**: Organize data into related tables made of rows and columns. Each row represents a record, and each column represents an attribute of the data.
- **Advantages of Relational Databases**: Scalable, widely applicable across domains (e.g., healthcare, business, gaming), and structured to maintain reliable data.
- **Common Use Cases**: Web development, inventory systems, e-commerce, healthcare, and enterprise applications.

Key Concepts

- **Schema**: A relational database requires a schema that defines its structure—tables, columns, data types, constraints, and relationships.
- **Primary Keys**: Unique identifiers for each row in a table. They are essential for data integrity and are used to relate records between tables via foreign keys.
- **Foreign Keys**: References to primary keys in another table, used to link related data across tables.
- **Relationships**: By connecting tables through primary and foreign keys, you can structure normalized data and perform meaningful queries.
- **Entity Relationship Diagrams (ERDs)**: Visualize how entities (tables) relate to each other in a database schema.
- **Data Integrity**: Enforced using keys and data types. Ensures consistency and accuracy of stored data.

SQL Basics

- **Queries**: Requests to retrieve specific data from the database.

```
SELECT * FROM dogs WHERE age < 3;
```

- **WHERE clause**: Filter results based on conditions. Use comparison operators like `<`, `=`, `>`, etc.
- **Select with ORDER BY**: Retrieve and sort results based on a column.

```
SELECT columns FROM table_name ORDER BY column_name;
```

Table Operations

- `CREATE TABLE` **Statement**: This statement is used to create a new table in a database.

```
CREATE TABLE first_table();
```

- `ALTER TABLE ADD COLUMN` **Statement**: This statement is used to add a column to an existing table.

```
ALTER TABLE table_name ADD COLUMN column_name DATATYPE;
```

- `ALTER TABLE DROP COLUMN` **Statement**: This statement is used to drop a column from an existing table.

```
ALTER TABLE table_name DROP COLUMN column_name;
```

- `ALTER TABLE RENAME COLUMN` **Statement**: This statement is used to rename a column in a table.

```
ALTER TABLE table_name RENAME COLUMN column_name TO new_name;
```

- `DROP TABLE` **Statement**: This statement is used to drop an entire table from the database.

```
DROP TABLE table_name;
```

- `ALTER DATABASE RENAME` **Statement**: This statement is used to rename a database.

```
ALTER DATABASE database_name RENAME TO new_database_name;
```

- `DROP DATABASE` **Statement**: This statement is used to drop an entire database.

```
DROP DATABASE database_name;
```

Constraints & Data Integrity

- `ALTER TABLE ADD COLUMN` **with Constraint**: This statement is used to add a column with a constraint to an existing table.

```
ALTER TABLE table_name ADD COLUMN column_name DATATYPE CONSTRAINT;
```

- `NOT NULL` **Constraint**: This constraint ensures that a column cannot have NULL values.

```
column_name VARCHAR(50) NOT NULL
```

- `ALTER TABLE ADD PRIMARY KEY` **Statement**: This statement is used to add a primary key constraint to a table.

```
ALTER TABLE table_name ADD PRIMARY KEY(column_name);
```

- `ALTER TABLE DROP CONSTRAINT` **Statement**: This statement is used to drop a constraint from a table.

```
ALTER TABLE table_name DROP CONSTRAINT constraint_name;
```

- `ALTER TABLE ADD COLUMN` **with Foreign Key**: This statement is used to add a foreign key column that references another table.

```
ALTER TABLE table_name ADD COLUMN column_name DATATYPE REFERENCES referenced_table_name(referenced_column_name);
```

- `ALTER TABLE ADD UNIQUE` **Statement**: This statement is used to add a UNIQUE constraint to a column.

```
ALTER TABLE table_name ADD UNIQUE(column_name);
```

- `ALTER TABLE ALTER COLUMN SET NOT NULL` **Statement**: This statement is used to set a NOT NULL constraint on an existing column.

```
ALTER TABLE table_name ALTER COLUMN column_name SET NOT NULL;
```

- `INSERT` **Statement with NULL Values**: This statement demonstrates how to insert NULL values into a table.

```
INSERT INTO table_name(column_a) VALUES(NULL);
-- or
INSERT INTO table_name(column_b) VALUES('value'); -- if column_a allows nulls
```

- **Composite Primary Key**: This constraint defines a primary key that consists of multiple columns.

```
CREATE TABLE course_enrollments (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id)
);
```

Data Manipulation (CRUD)

- `INSERT` **Statement**: This statement is used to insert a single row into a table.

```
INSERT INTO table_name(column_1, column_2) VALUES(value1, value2);
```

- `INSERT` **Statement with Omitted Columns**: This statement shows how to insert values without explicitly listing the column names, relying on the default column order in the table.

```
INSERT INTO dogs VALUES ('Gino', 3);
```

- `INSERT` **Statement with Multiple Rows**: This statement is used to insert multiple rows into a table in a single operation.

```
INSERT INTO dogs (name, age) VALUES
('Gino', 3),
('Nora', 2);
```

- `UPDATE` **Statement**: This statement is used to update existing data in a table based on a condition.

```
UPDATE table_name SET column_name=new_value WHERE condition;
```

- `DELETE` **Statement**: This statement is used to delete rows from a table based on a condition.

```
DELETE FROM table_name WHERE condition;
```

Data Types

- `NUMERIC` **Data Type**: This data type is used to store precise decimal numbers with a specified precision and scale.

```
price NUMERIC(10, 2)
```

- `TEXT` **Data Type**: This data type is used to store variable-length character strings with no specific length limit.

```
column_name TEXT
```

- `INTEGER` **Data Type**: This data type is used to store whole numbers without decimal places.

```
units_sold INTEGER
```

- `SMALLINT` and `BIGINT` **Data Types**: These are variants of INTEGER with smaller and larger ranges respectively.
- `SERIAL` **Data Type**: This data type is used to create auto-incrementing integer columns in PostgreSQL.

```
id SERIAL
```

- `AUTO_INCREMENT` **Attribute**: This attribute is used in MySQL to create auto-incrementing integer columns.

```
id INT AUTO_INCREMENT
```

- `VARCHAR` **Data Type**: This data type is used to store variable-length character strings with a specified maximum length.

```
name VARCHAR(50)
```

- `DATE` **Data Type**: This data type is used to store date values (year, month, day).

```
event_date DATE
```

- `TIME` **Data Type**: This data type is used to store time values (hour, minute, second).

```
start_time TIME
```

- `TIMESTAMP` **Data Type**: This data type is used to store date and time values, optionally with time zone information.

```
event_timestamp TIMESTAMP
event_timestamp TIMESTAMP WITH TIME ZONE
```

- `BOOLEAN` **Data Type**: This data type is used to store true/false values.

```
is_active BOOLEAN
```

Database Relationships

- **Types of Relationships**: These are the different ways tables can be related to each other in a relational database.
    - One-to-one
    - One-to-many
    - Many-to-one
    - Many-to-many
    - Self-referencing (recursive)
- **One-to-One Relationship**: This relationship type means that each record in one table corresponds to exactly one record in another table.

```
One employee is assigned exactly one vehicle.
Tables: employees, vehicles
```

- **One-to-Many Relationship**: This relationship type means that one record in a table can be associated with multiple records in another table.

```
One customer can have many orders.
Tables: customers → orders
```

- **Many-to-Many Relationship via Junction Table**: This relationship type is implemented using a junction table that contains foreign keys from both related tables.

```
CREATE TABLE books_authors (
    author_id INT REFERENCES authors(id),
    book_id INT REFERENCES books(id)
);
```

- **Self-Referencing Relationships**: This relationship type occurs when a table references itself, creating a hierarchical structure.

```
An employee table where each employee may report to another employee.
```

Advanced SQL (Joins)

- `INNER JOIN` **Statement**: This join returns only the rows that have matching values in both tables.

```
SELECT *
FROM products
INNER JOIN sales ON products.product_id = sales.product_id;
```

- `FULL OUTER JOIN` **Statement**: This join returns all rows from both tables, including unmatched rows from either table.

```
SELECT *
FROM products
FULL OUTER JOIN sales ON products.product_id = sales.product_id;
```

- `LEFT OUTER JOIN` **Statement**: This join returns all rows from the left table and matching rows from the right table.

```
SELECT *
FROM products
LEFT JOIN sales ON products.product_id = sales.product_id;
```

- `RIGHT OUTER JOIN` **Statement**: This join returns all rows from the right table and matching rows from the left table.

```
SELECT *
FROM products
RIGHT JOIN sales ON products.product_id = sales.product_id;
```

- `SELF JOIN` **Statement**: This join is used to join a table with itself to compare rows within the same table.

```
SELECT A.column_name, B.column_name
FROM table_name A
JOIN table_name B ON A.related_column = B.related_column;
```

- `CROSS JOIN` **Statement**: This join returns the Cartesian product of two tables, combining every row from the first table with every row from the second table.

```
SELECT *
FROM table1
CROSS JOIN table2;
```

PostgreSQL Specific Commands

- `psql` **Login Command**: This command is used to log in to PostgreSQL with specific username and database.

```
psql --username=freecodecamp --dbname=postgres
```

- `\l` **Command**: This command lists all databases in the PostgreSQL instance.

```
\l
```

- `CREATE DATABASE` **and** `\c` **Commands**: These commands are used to create a new database and connect to it.

```
CREATE DATABASE database_name;
\c database_name
```

- `\d` **Command**: This command lists all tables in the current database.

```
\d
```

- `\d table_name` **Command**: This command displays the schema/structure of a specific table.

```
\d table_name
```

- `\q` **Command**: This command exits the PostgreSQL client.

```
\q
```

Relational vs Non-Relational

- **Non-Relational (NoSQL) Databases**: Store unstructured or semi-structured data. Do not require a rigid schema and are more flexible for evolving data models.
- **Choosing Between Relational and Non-Relational**: Depends on the nature of your data and application requirements.
- **Relational vs Non-Relational**: Choose relational for structured data and consistency; NoSQL for flexibility and fast-changing data.

Popular RDBMS Systems

- **MySQL**: Open-source, reliable, widely used in web development, supported by a large community.
- **PostgreSQL**: Open-source, advanced, extensible. Supports custom data types and server-side programming.
- **SQLite**: Lightweight, file-based, serverless. Ideal for small applications.
- **Microsoft SQL Server**: Proprietary, enterprise-grade database.
- **Oracle Database**: Commercial RDBMS known for large-scale performance and scalability.

Best Practices

- **Naming Convention**: Use `snake_case` (e.g., `delivery_orders`) for table and column names.

Bash Scripting Basics

- **Bash scripting**: Writing a sequence of Bash commands in a file, which you can then execute with Bash to run the contents of the file.
- **Shebang**: The commented line at the beginning of a script (e.g., `#!/bin/bash`) that indicates what interpreter should be used for the script.

```
#!/bin/bash
```

- **Variable assignment**: Instantiate variables using the syntax `variable_name=value`.

```
servers=("prod" "dev")
```

- **Variable creation rules**: Create variables with `VARIABLE_NAME=VALUE` syntax. No spaces are allowed around the equal sign (`=`). Use double quotes if the value contains spaces.

```
NAME=John
MESSAGE="Hello World"
COUNT=5
TEXT="The next number is, "
```

- **Variable usage**: Access variable values by placing `$` in front of the variable name.

```
echo $NAME
echo "The message is: $MESSAGE"
```

- **Variable interpolation**: Use `$variable_name` to access the value of a variable within strings and commands.

```
TEXT="The next number is, "
NUMBER=42
echo $TEXT B:$NUMBER
echo $TEXT I:$NUMBER
```

```
echo "Pulling $server"
rsync --archive --verbose $server:/etc/nginx/conf.d/server.conf configs/$server.conf
```

- **Variable scope**: Shell scripts run from top to bottom, so variables can only be used below where they are created.

```
NAME="Alice"
echo $NAME
```

- **User input**: Use `read` to accept input from users and store it in a variable.

```
read USERNAME
echo "Hello $USERNAME"
```

- **Comments**: Add comments to your scripts using `#` followed by your comment text.
  - Single-line comments start with `#` and continue to the end of the line
  - Comments are ignored by the shell and don't affect script execution

```
# This is a single-line comment
NAME="John"  # Comment at end of line
```

- **Multi-line comments**: Comment out blocks of code using colon and quotes.

```
: '
This is a multi-line comment
Everything between the quotes is ignored
Useful for debugging or documentation
'
```

- **Built-in commands and help**:
  - Use `help` to see a list of built-in bash commands
  - Use `help <command>` to get information about specific built-in commands
  - Some commands (like `if`) are built-ins and don't have man pages
  - Built-in commands are executed directly by the shell rather than as external programs
  - Use `help function` to see information about creating functions

```
help
help if
help function
```

- **Finding command locations**: Use `which` to locate where executables are installed.
  - Shows the full path to executable files
  - Useful for finding interpreter locations (like bash)
  - Helps verify which version of a command will be executed

```
which bash
which python
which ls
```

- **Manual pages**: Use `man` to access detailed documentation for commands.
  - Provides comprehensive information about command usage
  - Shows all available options and examples
  - Use arrow keys to navigate, 'q' to quit
  - Not all commands have manual pages (built-ins use `help` instead)

```
man echo
man ls
man bash
```

- **Help flags**: Many commands support `--help` for quick help information.
  - Alternative to manual pages for quick reference
  - Shows command syntax and common options
  - Not all commands support this flag (some may show error)

```
ls --help
chmod --help
mv --help
```

- **Echo command options**: The `echo` command supports various options:
  - `-e` option enables interpretation of backslash escapes
  - `\n` creates a new line
  - Empty lines are only printed when values are enclosed in quotes
  - Useful for creating formatted output and program titles

```
echo -e "Line 1\nLine 2"
echo ""
echo -e "\n~~ Program Title ~~\n"
echo "Line 1\nLine 2"
```

- **Script arguments**: Programs can accept arguments that are accessible using `$` variables.
  - `$*` prints all arguments passed to the script
  - `$@` prints all arguments passed to the script as separate quoted strings
  - `$<number>` accesses specific arguments by position (e.g., `$1`, `$2`, `$3`)

```
echo $*
echo $@
echo $1
echo $2
```

Double Bracket Expressions `[[ ]]`

- **Double bracket syntax**: Use `[[ ]]` for conditional testing and pattern matching.
  - Must have spaces inside the brackets and around operators
  - Returns exit status 0 (true) or 1 (false) based on the test result

```
[[ $variable == "value" ]]
[[ $number -gt 10 ]]
[[ -f filename.txt ]]
```

- **String comparison operators**: Compare strings using various operators within `[[ ]]`.
  - `==` (equal): Tests if two strings are identical
  - `!=` (not equal): Tests if two strings are different
  - `<` (lexicographically less): String comparison in alphabetical order
  - `>` (lexicographically greater): String comparison in alphabetical order

```
[[ "apple" == "apple" ]]
[[ "apple" != "orange" ]]
[[ "apple" < "banana" ]]
[[ "zebra" > "apple" ]]
```

- **Numeric comparison operators**: Compare numbers using specific numeric operators.
  - `-eq` (equal): Numeric equality comparison
  - `-ne` (not equal): Numeric inequality comparison
  - `-lt` (less than): Numeric less than comparison
  - `-le` (less than or equal): Numeric less than or equal comparison
  - `-gt` (greater than): Numeric greater than comparison
  - `-ge` (greater than or equal): Numeric greater than or equal comparison

```
[[ $number -eq 5 ]]
[[ $count -ne 0 ]]
[[ $age -ge 18 ]]
[[ $score -lt 100 ]]
```

- **Logical operators**: Combine multiple conditions using logical operators.
  - `&&` (and): Both conditions must be true
  - `||` (or): At least one condition must be true
  - `!` (not): Negates the condition (makes true false, false true)

```
[[ $age -ge 18 && $age -le 65 ]]
[[ $name == "John" || $name == "Jane" ]]
[[ ! -f missing_file.txt ]]
```

- **File test operators**: Test file properties and existence.
  - `-e file`: True if file exists
  - `-f file`: True if file exists and is a regular file
  - `-d file`: True if file exists and is a directory
  - `-r file`: True if file exists and is readable
  - `-w file`: True if file exists and is writable
  - `-x file`: True if file exists and is executable
  - `-s file`: True if file exists and has size greater than zero

```
[[ -e /path/to/file ]]
[[ -f script.sh ]]
[[ -d /home/user ]]
[[ -x program ]]
```

- **Pattern matching with `=~`**: Use regular expressions for advanced pattern matching.
  - `=~` operator enables regex pattern matching
  - Pattern should not be quoted when using regex metacharacters
  - Supports full regular expression syntax
  - Case-sensitive by default

```
[[ "hello123" =~ [0-9]+ ]]
[[ "email@domain.com" =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]
[[ "filename.txt" =~ \.txt$ ]]
```

- **Variable existence testing**: Check if variables are set or empty.

- Test if variable is empty: `[[ ! $variable ]]`

```
[[ ! $undefined_var ]]
```

Double Parentheses Expressions `(( ))`

- **Arithmetic evaluation**: Use `(( ))` for mathematical calculations and numeric comparisons.
  - Evaluates arithmetic expressions using C-style syntax
  - Variables don't need `$` prefix inside double parentheses
  - Returns exit status 0 if result is non-zero, 1 if result is zero
  - Supports all standard arithmetic operators

```
(( result = 10 + 5 ))
(( count++ ))
(( total += value ))
```

- **Arithmetic operators**: Mathematical operators available in `(( ))`.
  - `+` (addition): Add two numbers
  - `-` (subtraction): Subtract second number from first
  - `*` (multiplication): Multiply two numbers
  - `/` (division): Divide first number by second (integer division)
  - `%` (modulus): Remainder after division
  - `**` (exponentiation): Raise first number to power of second

```
(( sum = a + b ))
(( diff = x - y ))
(( product = width * height ))
(( remainder = num % 10 ))
(( power = base ** exponent ))
```

- **Assignment operators**: Modify variables using arithmetic assignment operators.
  - `=` (assignment): Assign value to variable
  - `+=` (add and assign): Add value to variable
  - `-=` (subtract and assign): Subtract value from variable
  - `*=` (multiply and assign): Multiply variable by value
  - `/=` (divide and assign): Divide variable by value
  - `%=` (modulus and assign): Set variable to remainder

```
(( counter = 0 ))
(( counter += 5 ))
```

```
(( total -= cost ))
(( area *= 2 ))
(( value /= 3 ))
```

- **Increment and decrement operators**: Modify variables by one.
  - `++variable` (pre-increment): Increment before use
  - `variable++` (post-increment): Increment after use
  - `--variable` (pre-decrement): Decrement before use
  - `variable--` (post-decrement): Decrement after use

```
(( ++counter ))
(( index++ ))
(( --remaining ))
(( attempts-- ))
```

- **Comparison operators**: Compare numbers using arithmetic comparison.
  - `==` (equal): Numbers are equal
  - `!=` (not equal): Numbers are not equal
  - `<` (less than): First number is less than second
  - `<=` (less than or equal): First number is less than or equal to second
  - `>` (greater than): First number is greater than second
  - `>=` (greater than or equal): First number is greater than or equal to second

```
(( age >= 18 ))
(( score < 100 ))
(( count == 0 ))
(( temperature > freezing ))
```

- **Logical operators**: Combine arithmetic conditions.
  - `&&` (and): Both conditions must be true
  - `||` (or): At least one condition must be true
  - `!` (not): Negates the condition

```
(( age >= 18 && age <= 65 ))
(( score >= 90 || extra_credit > 0 ))
(( !(count == 0) ))
```

- **Bitwise operators**: Perform bit-level operations on integers.
  - `&` (bitwise AND): AND operation on each bit

- `|` (bitwise OR): OR operation on each bit
- `^` (bitwise XOR): XOR operation on each bit
- `~` (bitwise NOT): Invert all bits
- `<<` (left shift): Shift bits to the left
- `>>` (right shift): Shift bits to the right

```
(( result = a & b ))
(( flags |= new_flag ))
(( shifted = value << 2 ))
```

- **Conditional (ternary) operator**: Use `condition ? true_value : false_value` syntax.
  - Provides a concise way to assign values based on conditions
  - Similar to the ternary operator in C-style languages
  - Evaluates condition and returns one of two values

```
(( result = (score >= 60) ? 1 : 0 ))
(( max = (a > b) ? a : b ))
(( sign = (num >= 0) ? 1 : -1 ))
```

- **Command substitution with arithmetic**: Use `$(( ))` to capture arithmetic results.
  - Returns the result of the arithmetic expression as a string
  - Can be used in assignments or command arguments
  - Useful for calculations that need to be used elsewhere

```
result=$(( 10 + 5 ))
echo "The answer is $(( a * b ))"
array_index=$(( RANDOM % array_length ))
```

Control Flow and Conditionals

- **Conditional statements:** Use `if` statements to execute code based on conditions.
  - Basic syntax: `if [[ CONDITION ]] then STATEMENTS fi`
  - Full syntax: `if [[ CONDITION ]] then STATEMENTS elif [[ CONDITION ]] then STATEMENTS else STATEMENTS fi`
  - Can use both `[[ ]]` and `(( ))` expressions for different types of conditions
  - **elif (else if)**: Optional, can be repeated multiple times to test additional conditions in sequence
  - **else**: Optional, executes when all previous conditions are false
  - Can mix double parentheses `(( ... ))` and double brackets `[[ ... ]]` in same conditional chain

```
if (( NUMBER <= 15 ))
then
    echo "B:$NUMBER"
```

```
elif [[ $NUMBER -le 30 ]]
then
    echo "I:$NUMBER"
elif (( NUMBER < 46 ))
then
    echo "N:$NUMBER"
elif [[ $NUMBER -lt 61 ]]
then
    echo "G:$NUMBER"
else
    echo "O:$NUMBER"
fi
```

Command Execution and Process Control

- **Command separation**: Use semicolon (`;`) to run multiple commands on a single line.
  - Commands execute sequentially from left to right
  - Each command's exit status can be checked individually

```
[[ 4 -ge 5 ]]; echo $?
ls -l; echo "Done"
```

- **Exit status**: Every command has an exit status that indicates success or failure.
  - Access exit status of the last command with `$?`
  - Exit status `0` means success (true/no errors)
  - Any non-zero exit status means failure (false/errors occurred)
  - Common error codes: `127` (command not found), `1` (general error)

```
echo $?
[[ 4 -le 5 ]]; echo $?
ls; echo $?
bad_command; echo $?
```

- **Subshells and command substitution**: Different uses of parentheses for execution contexts.
  - Single parentheses `( ... )` create a subshell
  - `$( ... )` performs command substitution
  - Subshells run in separate environments and don't affect parent shell variables

```
( cd /tmp; echo "Current dir: $(pwd)" )
current_date=$(date)
```

```
file_count=$(ls | wc -l)
echo "Today is $current_date"
echo "Found $file_count files"
```

- **Sleep command**: Pause script execution for a specified number of seconds.
  - Useful for creating delays in scripts
  - Can be used with decimal values for subsecond delays

```
sleep 3
sleep 0.5
sleep 1
```

Loops

- **While loops**: Execute code repeatedly while a condition is true.
  - Syntax: `while [[ CONDITION ]] do STATEMENTS done`

```
I=5
while [[ $I -ge 0 ]]
do
    echo $I
    (( I-- ))
    sleep 1
done
```

- **Until loops**: Execute code repeatedly until a condition becomes true.
  - Syntax: `until [[ CONDITION ]] do STATEMENTS done`

```
until [[ $QUESTION =~ \?$ ]]
do
    echo "Please enter a question ending with ?"
    read QUESTION
done
until [[ $QUESTION =~ \?$ ]]
do
    GET_FORTUNE again
done
```

- **For loops**: Iterate through arrays or lists using `for` loops with `do` and `done` to define the loop's logical block.

```
for server in "${servers[@]}"
do
    echo "Processing $server"
done
for (( i = 1; i <= 5; i++ ))
do
    echo "Number: $i"
done
for (( i = 5; i >= 1; i-- ))
do
    echo "Countdown: $i"
done
for i in {1..5}
do
    echo "Count: $i"
done
```

Arrays

- **Arrays**: Store multiple values in a single variable.
  - Create arrays with parentheses: `ARRAY=("value1" "value2" "value3")`
  - Access elements by index: `${ARRAY[0]}`, `${ARRAY[1]}`
  - Access all elements: `${ARRAY[@]}` or `${ARRAY[*]}`
  - Array indexing starts at 0

```
RESPONSES=("Yes" "No" "Maybe" "Ask again later")

echo ${RESPONSES[0]}     # Yes
echo ${RESPONSES[1]}     # No
echo ${RESPONSES[5]}     # Index 5 doesn't exist; empty string
echo ${RESPONSES[@]}     # Yes No Maybe Ask again later
echo ${RESPONSES[*]}     # Yes No Maybe Ask again later
```

- **Array inspection with declare**: Use `declare -p` to view array details.
  - Shows the array type with `-a` flag
  - Displays all array elements and their structure

```
ARR=("a" "b" "c")
declare -p ARR # ARR=([0]="a" [1]="b" [2]="c")
```

- **Array expansion**: Use `"${array_name[@]}"` syntax to expand an array into individual elements.

```
for server in "${servers[@]}"
```

Functions

- **Functions**: Create reusable blocks of code.
  - Define with `FUNCTION_NAME() { STATEMENTS }`
  - Call by using the function name
  - Can accept arguments accessible as `$1`, `$2`, etc.

```
GET_FORTUNE() {
    echo "Ask a question:"
    read QUESTION
}
GET_FORTUNE
```

- **Function arguments**: Functions can accept arguments just like scripts.
  - Arguments are passed when calling the function
  - Access arguments inside function using `$1`, `$2`, etc.
  - Use conditional logic to handle different arguments

```
GET_FORTUNE() {
    if [[ ! $1 ]]
    then
        echo "Ask a yes or no question:"
    else
        echo "Try again. Make sure it ends with a question mark:"
    fi
    read QUESTION
}
GET_FORTUNE
GET_FORTUNE again
```

Random Numbers and Mathematical Operations

- **Random numbers**: Generate random values using the `$RANDOM` variable.
  - `$RANDOM` generates numbers between 0 and 32767
  - Use modulus operator to limit range: `$RANDOM % 75`
  - Add 1 to avoid zero: `$(( RANDOM % 75 + 1 ))`

- Must use `$(( ... ))` syntax for calculations with `$RANDOM`

```
NUMBER=$(( RANDOM % 6 ))
DICE=$(( RANDOM % 6 + 1 ))
BINGO=$(( RANDOM % 75 + 1 ))
echo $(( RANDOM % 10 ))
```

- **Random array access**: Use random numbers to access array elements randomly.
  - Generate random index within array bounds
  - Use random index to access array elements
  - Useful for random selections from predefined options

```
RESPONSES=("Yes" "No" "Maybe" "Outlook good" "Don't count on it" "Ask again later")
N=$(( RANDOM % 6 ))
echo ${RESPONSES[$N]}
```

- **Modulus operator**: Use `%` to get the remainder of division operations.
  - Essential for limiting random number ranges
  - Works with `$RANDOM` to create bounded random values
  - `RANDOM % n` gives numbers from 0 to n-1

```
echo $(( 15 % 4 ))
echo $(( RANDOM % 100 ))
echo $(( RANDOM % 10 + 1 ))
```

Environment and System Information

- **Environment variables**: Predefined variables available in the shell environment.
  - `$RANDOM`: Generates random numbers between 0 and 32767
  - `$LANG`: System language setting
  - `$HOME`: User's home directory path
  - `$PATH`: Directories searched for executable commands
  - View all with `printenv` or `declare -p`

```
echo $RANDOM
echo $HOME
echo $LANG
printenv
```

- **Variable inspection**: Use `declare` to view and work with variables.

- `declare -p`: Print all variables and their values
- `declare -p VARIABLE`: Print specific variable details
- Shows variable type (string, array, etc.) and attributes

```
declare -p
declare -p RANDOM
declare -p MY_ARRAY
```

- **Command types**: Different categories of commands available in bash.
  - **Built-in commands**: Executed directly by the shell (e.g., `echo`, `read`, `if`)
  - **External commands**: Binary files in system directories (e.g., `ls`, `sleep`, `bash`)
  - **Shell keywords**: Language constructs (e.g., `then`, `do`, `done`)
  - Use `type <command>` to see what type a command is

```
type echo
type ls
type if
type ./script.sh
```

File Creation and Management

- **File creation**: Use `touch` to create new empty files.
  - Creates a new file if it doesn't exist
  - Updates the timestamp if the file already exists
  - Commonly used to create script files before editing

```
touch script.sh
touch bingo.sh
touch filename.txt
```

Creating and Running Bash Scripts

- **Script execution methods**: Multiple ways to run bash scripts:
  - **`sh scriptname.sh`**: Run with the sh shell interpreter.
  - **`bash scriptname.sh`**: Run with the bash shell interpreter.
  - **`./scriptname.sh`**: Execute directly (requires executable permissions).

```
sh questionnaire.sh
bash questionnaire.sh
./questionnaire.sh
```

File Permissions and Script Execution

- **Permission denied error**: When using `./scriptname.sh`, you may get "permission denied" if the file lacks executable permissions.
- **Checking permissions**: Use `ls -l` to view file permissions.

```
ls -l questionnaire.sh
```

- **Permission format**: The output shows permissions as `-rw-r--r--` where:
  - First character (`-`): File type (- for regular file, d for directory)
  - Next 9 characters: Permissions for owner, group, and others
  - `r` = read, `w` = write, `x` = execute
- **Adding executable permissions**: Use `chmod +x` to give executable permissions to everyone.

```
chmod +x questionnaire.sh
```

- **Script organization**: Best practices for structuring bash scripts.
  - Start with shebang (`#!/bin/bash`)
  - Add descriptive comments about script purpose
  - Define variables at the top
  - Group related functions together
  - Main script logic at the bottom

```
#!/bin/bash
NAME="value"
ARRAY=("item1" "item2")
my_function() {
    echo "Function code here"
}
my_function
echo "Script complete"
```

- **Sequential script execution**: Create master scripts that run multiple programs in sequence.
  - Useful for automating workflows that involve multiple scripts
  - Each script runs to completion before the next one starts
  - Can combine different programs into a single execution flow
  - Arguments can be passed to individual scripts as needed
  - Can include different types of programs (interactive, automated, etc.)

```
#!/bin/bash
./setup.sh
./interactive.sh
./processing.sh
./cleanup.sh
```

Database Normalization

This is the process of organizing a relational database to reduce data redundancy and improve integrity.

Its benefits include:

- Minimizing duplicated data, which saves storage and reduces inconsistencies.
- Enforcing data integrity through the use of primary and foreign keys.
- Making databases easier to maintain and understand.

Normal Forms

- **1NF (First Normal Form)**
  - Each cell contains a single (atomic) value.
  - Each record is unique (enforced by a primary key).
  - Order of rows/columns is irrelevant.
  - Example: Move multiple phone numbers from a `students` table into a separate `student_phones` table.
- **2NF (Second Normal Form)**
  - Meets 1NF requirements.
  - No **partial dependencies**: every non-key attribute must depend on the entire composite primary key.
  - Example: Split `orders` table into `order_header` and `order_items` to avoid attributes depending on only part of the key.
- **3NF (Third Normal Form)**
  - Meets 2NF requirements.
  - No **transitive dependencies**: non-key attributes cannot depend on other non-key attributes.
  - Example: Move `city_postal_code` to a `cities` table instead of storing it with every order.
- **BCNF (Boyce-Codd Normal Form)**
  - Meets 3NF requirements.
  - Every determinant (left-hand side of a functional dependency) must be a superkey.

Tip: Aim for 3NF in most designs for a good balance of integrity and performance.

Key SQL Concepts

- SQL is a Structured Query Language for communicating with relational databases.
- **Basic commands** → `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE`, `ALTER TABLE`, etc.

- `Joins` → Combines data from multiple tables (`INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL JOIN`).

Running SQL Commands in Bash

You can run SQL commands directly from the command line using the `psql` command-line client for PostgreSQL or similar tools for other databases.

For example, to run a SQL file in PostgreSQL:

```
psql -U username -d database_name -c "SELECT * FROM students;"
```

You can also execute MySQL commands directly:

```
mysql -u username -p database_name -e "SELECT * FROM students;"
```

Run SQL from a File

```
# PostgreSQL
psql -U username -d database_name -f script.sql

# MySQL
mysql -u username -p database_name < script.sql
```

Embed SQL in a Bash Script

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"

# Insert student data
psql -U "$DB_USER" -d "$DB_NAME" -c \
"INSERT INTO students (name, age, major) VALUES ('Alice', 20, 'CS');"
```

Use of Variables in SQL

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"
STUDENT_NAME="Bob"
AGE=21
```

```
psql -U "$DB_USER" -d "$DB_NAME" -c \
"INSERT INTO students (name, age) VALUES ('$STUDENT_NAME', $AGE);"
```

Tip: Sanitize variables to avoid SQL injection.

Retrieving and Using SQL Query Results in Bash

When you run SQL queries via `psql`, you can **capture** and **process** the returned values in your Bash scripts.

Capturing a Single Value

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"

# Get total student count
STUDENT_COUNT=$(psql -U "$DB_USER" -d "$DB_NAME" -t -A -c \
"SELECT COUNT(*) FROM students;")

echo "Total students: $STUDENT_COUNT"
```

Output → 42

Retrieving Multiple Columns

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"

# Get top 3 students' names and ages
RESULTS=$(psql -U "$DB_USER" -d "$DB_NAME" -t -A -F"," -c \
"SELECT name, age FROM students LIMIT 3;")

echo "Top 3 students:"
echo "$RESULTS"
```

Output

```
Alice,20
Bob,21
Charlie,22
```

Looping Through Query Results

```
#!/bin/bash
DB_USER="school_admin"
DB_NAME="school"

# Get student names and majors
psql -U "$DB_USER" -d "$DB_NAME" -t -A -F"," -c \
"SELECT name, major FROM students;" | while IFS="," read -r name major
do
    echo "Student: $name | Major: $major"
done
```

Shape of Output

```
Student: Alice | Major: CS
Student: Bob   | Major: Math
Student: Carol | Major: Physics
```

SQL Injection

It is a web security vulnerability where attackers insert malicious SQL code into input fields to manipulate the database.

This can lead to risky actions like:

- Bypassing authentication.
- Stealing sensitive data.
- Modifying or deleting records.

An example of an SQL injection attack:

```
SELECT * FROM users WHERE username = ' " " OR "1"="1" -- ' AND password = 'anything';
```

This query would return all users because the condition `OR "1"="1"` is always true, allowing attackers to bypass login checks.

Preventing SQL Injection

1. **Use Prepared Statements**: These separate SQL code from data, preventing injection. Here's an example (Node.js with pg):

   ```
   client.query('SELECT * FROM users WHERE username = $1 AND password = $2', [username, password]);
   ```

2. **Input Validation**: Sanitize and validate all user inputs to ensure they conform to expected formats.

3. **Least Privilege**: Use database accounts with the minimum permissions necessary for the application.

Note: Never grant admin rights to application accounts.

N+1 Problem

The N+1 problem occurs when an application makes one query to retrieve a list of items (N) and then makes an additional query for each item to retrieve related data, resulting in N+1 queries.

Why It's Bad

- Each query adds network and processing overhead.
- Multiple small queries are slower than one optimized query.

Example of N+1 Pattern

```
-- 1: Get list of orders
SELECT * FROM orders LIMIT 50;

-- N: For each order, get customer
SELECT * FROM customers WHERE customer_id = ...;
```

Solution: Use `JOINs` or other set-based operations.

```
SELECT
  orders.order_id,
  orders.product,
  orders.quantity,
  customers.customer_id,
  customers.name,
  customers.email,
  customers.address
FROM orders
JOIN customers
  ON orders.customer_id = customers.customer_id
WHERE orders.order_id IN (SELECT order_id FROM orders LIMIT 50);
```

Always look for opportunities to combine related data into a single query.

Introduction to Version Control

- **Definition**: A version control system allows you to track and manage changes in your project. Examples of version control systems used in software are Git, SVN, or Mercurial.

Cloud-Based Version Control Providers

- **List of Cloud-Based Version Control Providers**: GitHub and GitLab are popular examples of cloud-based version control providers that allow software teams to collaborate and manage repositories.

Installing and Setting up Git

- **Installing Git**: To check if Git is already installed on your machine you can run the following command in the terminal:

```
git --version
```

If you see a version number, that means Git is installed. If not, then you will need to install it.

For Linux systems, Git often comes preinstalled with most distros. If you do not have Git pre-installed, you should be able to install it with your package manager commands such as `sudo apt-get install git` or `sudo pacman -S git`.

For Mac users, you can install Git via Homebrew with `brew install git`, or you can download the executable installer from Git's website.

For Windows, you can download the executable installer from Git's website. Or, if you have set up Chocolatey, you can run `choco install git.install` in PowerShell. Note that on Windows, you may also want to download Git Bash so you have a Unix-like shell environment available.

To make sure the installation worked, run the `git --version` command again in the terminal.

- **Git Configurations**: `git config` is used to set configuration variables that are responsible for how Git operates on your machine. To view your current setting variables and where they are stored on your system, you can run the following command:

```
git config --list --show-origin
```

Right now you should be seeing only system-level configuration settings if you just installed Git for the first time.

To set your user name, you can run the following command:

```
git config --global user.name "Jane Doe"
```

The `--global` flag is used here to set the user name for all projects on your system that use Git. If you need to override the user name for a particular project, then you can run the command in that particular project directory without the `--global` flag.

To set the user email address, you can run the following command:

```
git config --global user.email janedoe@example.com
```

Another configuration you can set is the preferred editor you want Git to use. Here is an example of how to set your preferred editor to Emacs:

```
git config --global core.editor emacs
```

If you choose not to set a preferred editor, then Git will default to your system's default editor.

Open vs. Closed Source Software

- **Definition**: "Open-source" means people can see the code you publish, propose changes, report issues, and even run a modified version. "Closed-source" means the only people who can see and interact with the project are the people you explicitly authorize.

GitHub

- **Definition**: GitHub is a cloud-based solution that offers storage of version-controlled projects in something called "repositories", and enables collaboration features to use with those projects.
- **GitHub CLI**: This tool is used to do GitHub-specific tasks without leaving the command line. If you do not have it installed, you can get instructions to do so from GitHub's documentation - but you should have it available in your system's package manager.
- **GitHub Pages**: GitHub Pages is an option for deploying static sites, or applications that do not require a back-end server to handle logic. That is, applications that run entirely client-side, or in the user's browser, can be fully deployed on this platform.
- **GitHub Actions**: GitHub Actions is a feature that lets you automate workflows directly in your GitHub repository including building, testing, and deploying your code.

Common Git Commands

- `git init`: This will initialize an empty Git repository so Git can begin tracking changes for this project. When you initialize an empty Git repository to a project, a new `.git` hidden directory will be added. This `.git` directory contains important information for Git to manage your project.
- `git status`: This command is used to show the current state of your working directory - you will be using this command a lot in your workflow.
- `git add`: This command is used to stage your changes. Anything in the staging area will be added for the next commit. If you want to stage all unstaged changes, then you can use `git add .` The period (`.`) is an alias for the current directory you are in.
- `git commit`: This command is used to commit your changes. A commit is a snapshot of your project state at that given time. If you run `git commit`, it will open up your preferred editor you set in the Git configuration. Once the editor is open, you can provide a detailed message of your changes. You can also choose to provide a shorter message by using the `git commit -m` command like this:

```
git commit -m "short message goes here"
```

- `git log`: This will list all prior commits with helpful information like the author, date of commit, commit message and commit hash. The commit hash is a long string which serves as a unique identifier for a commit.
- `git remote add`: This command is used to setup the remote connection to your remote repo.
- `git push`: This command is used to push up your changes to a remote repository.
- `git pull`: This command is used to pull down the latest changes from your remote repository into your local repository.
- `git clone`: This command will clone a repository. This means you will have a copy of the repository. This copy includes the repository history, all files/folders and commits on your local device.
- `git remote -v`: This command will show the list of remote repositories associated with your local Git repository.
- `git branch`: This command will list all of your local branches.
- `git fetch upstream`: This command tells Git to go get the latest changes that are on your upstream remote (which is the original repo).
- `git merge upstream/main`: This command tells Git to merge the latest changes from the `main` branch in the upstream remote into your current branch.
- `git reset`: This command allows you to reset the current state of a branch. Passing the `--hard` flag tells Git to force the local files to match the branch state. This ensures that you have a clean slate to work from.
- `git rebase`: A rebase in Git is a way to move or combine a sequence of commits from one branch onto another.

Working with Branches

- **Definition**: A branch in Git is a separate workspace where you can make changes. The `main` branch will often represent the primary or production branch in a real world application. Developer teams will create multiple branches for new features and bug fixes and then merge those changes back into the `main` branch.
- **Creating a New Branch**: To create a new branch you can run the following command:

```
git branch feature
```

To checkout that branch, you can run the following command:

```
git checkout feature
```

Most developers will use the shorthand command for creating and checking out a branch which is the following:

```
git checkout -b new-branch-name
```

A newer and alternative command would be the `git switch` command. Here is an example for creating and switching to a new branch:

```
git switch -c new-branch-name
```

- **Branching Strategies**: Your `main` branch is your default branch and usually is pretty stable. So it is best to branch off from there to create new branches for items like bug fixes, new features, or other miscellaneous work.
- **Merge Conflicts**: This happens when Git tries to automatically merge changes from different branches but can't decide which changes to keep. This usually happens when there are conflicting changes for the same portion of the file.

Five States for a Git Tracked File

- **"Untracked"**: This means that the file is new to the repository, and Git has not "seen" it before.
- **"Modified"**: This file existed in the previous commit, and has changes that have not been committed.
- **"Ignored"**: You likely won't see ignored files in Git, but your IDE might have an indicator for them. Ignored files are excluded from Git operations, typically because they are included in the `.gitignore` file.
- **"Deleted"**: A deleted file is the opposite of an untracked file - it's a file that previously existed, and has been removed.
- **"Renamed"**: A renamed file is a file where the contents are unchanged, but the name or location of the file was modified. In some cases, a file can be considered renamed even if it has a small amount of changes.

`.gitignore` Files

- **Definition**: The `.gitignore` file is a special type of file related to Git operations. The name suggests that this file is used to tell Git to ignore things, and that's the common use case. But what it actually does is it tells Git to stop tracking a file.

Working with Repositories

- **Definition**: A repository is like a container for a project - if you are working on an app, you would keep the files for that app together in a repository. Repositories can be local on your computer, or remote on a service like GitHub.
- **Public vs. Private Repositories**: A public repository can be viewed and downloaded by anyone. A private repository can only be accessed by you, and anyone you grant explicit access to.

- **Creating Repositories on GitHub**: To create a new repository on GitHub, you can click on the `"New Repository"` button and walk through the GitHub UI of setting up a new repository.
- **Pushing Local Repositories to GitHub**: If you have a local project on your computer, you can push up that repository to GitHub. Here is a step-by-step overview of the process:

1. Initialize an empty git repository in the project directory (`git init`).
2. Make changes to your project.
3. Run the `git status` command to see all changes made that are being tracked by git.
4. Stage your changes (`git add`).
5. Commit your changes (`git commit`).
6. Setup the remote connection (`git remote add`).
7. Push your changes to GitHub (`git push`).

Pull Requests

- **Pull Requests**: A pull request is a request to pull changes in from your branch into the target branch. Pull requests are the flow you use when you want to contribute code changes to a project. This approach allows the maintainers of the project to review your changes. They can leave comments, ask questions, and suggest tweaks. Then once the review process is complete, it can be approved and merged into the main branch.

Contributing to Other Repositories

- **Process**: There are thousands of projects that you can contribute to. Here is the basic process on how to contribute to another repository:

1. Read the contributing documentation
2. Find an available issue to work on
3. Fork the repository
4. Clone your forked copy of the repository
5. Create a new branch
6. Make the changes according to the issue
7. Create a PR (Pull Request)
8. Wait for a review for that PR

Working with SSH and GPG Keys

- **GPG Keys**: GPG, or Gnu Privacy Guard, keys are typically used to sign files or commits. Someone can then use your public GPG key to verify that the file signature is from your key and that the contents of the file have not been modified or tampered with.

To generate a GPG key, you'll need to run:

```
gpg --full-generate-key
```

- **SSH Keys**: SSH, or Secure SHell, keys are typically used to authenticate a remote connection to a server - via the `ssh` utility. You can also use an SSH key to sign commits.

For an SSH key, you'll run:

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

`ed25519` is a modern public-key signature algorithm.

- **Signing Commits with GPG Keys**: In order to sign your commits with your GPG key, you'll need to upload your public key, not the private key, to your GitHub account. To list your public keys, you will need to run the following:

```
gpg --list-secret-keys --keyid-format=long
```

Then, to get the public key, use:

```
gpg --armor --export "<key id>"
```

Then, take the short ID you got from listing the keys and run this command to set it as your git signing key:

```
git config --global user.signingkey <your_gpg_key_id>
```

Then, you can pass the `-S` flag to your `git commit` command to sign a specific commit - you'll need to provide your passphrase. Alternatively, if you want to sign every commit automatically, you can set the autosign config to `true`:

```
git config --global commit.gpgsign true
```

- **Signing Commits with SSH Keys**: To sign with an SSH key, which is a relatively new feature on GitHub, you'll need to start by uploading the key to your GitHub account. Then you'll need to set the signing mode for git to use SSH:

```
git config --global gpg.format ssh
```

Then, to set the signing key, you'll pass the file path instead of an ID:

```
git config --global user.signingkey <path_to_your_ssh_keys>
```