**TDD – Test Driven Development**

TDD - way of writing code where you first write tests before you even start writing the actual code.

It helps make sure the code you write actually works.

The main idea is to always test your code before you finish it, so you don’t end up with bugs.

**Working of TDD**

**3 Steps:**

These steps repeat every time you want to add a new feature to your program:

**1.Write a Test (Red):**

First, you write a test for a small piece of functionality you want to add. This test is based on what you want the code to do. Since the code doesn't exist yet, the test will fail.

**Example:** If you’re building a calculator, you might write a test like, "Check if the calculator adds two numbers correctly."

**2.Write Code (Green):**

Next, you write the minimum amount of code required to make the test pass. The goal is to keep it simple and avoid over-complicating things—just enough code to ensure the test works. Once the test passes, you know that the feature is functioning correctly

**Example**: In the calculator, you might write a simple function that adds two numbers. Once that works, the test will pass.

**3.Refactor (Blue):**

After the test passes, you can look at the code you wrote and clean it up. This step is about improving the code, making it more efficient, readable, or organized, but still keeping the test passing. You should never change the functionality of the code during this step — only improve its structure.

**Example**: Maybe you find a way to make your calculator code simpler, or you change the name of a variable to make it easier to understand.

A diagram of a process

AI-generated content may be incorrect.

**Why TDD?**

* Catches bugs early: Tests help you spot problems before they grow.
* Improves code quality: Forces you to write cleaner, more organized code.
* Ensures functionality: Guarantees that your code does exactly what you expect.

**When TDD is Helpful:**

* **In complex projects**: Helps manage large codebases with many moving parts.
* **During maintenance**: Safely add new features or fix bugs without breaking existing code
* **In team development**: Makes sure everyone follows the same rules and expectations.
* **For refactoring**: When you want to improve code without risking old functionality.

TDD is helpful whenever you want to create reliable, maintainable, and bug-free software.

**Distributed Systems**

A system where different parts of a program (or data) are run on multiple machines. These parts work together to complete a task, even though they’re running separately. Think of it like a team of workers — each one does part of the job, and together they get everything done faster.

**Why use distributed systems?**

* **Speed**: You can process much larger amounts of data than you could on just one computer.
* **Reliability**: If one machine fails, the rest can still keep working.
* **Scalability**: As you need to process more data, you can easily add more machines.

**How do distributed systems work?**

1. **Splitting Tasks**: When you have a large task, it’s split into smaller tasks. Each machine handles one of these smaller tasks.
2. **Communicating**: These machines communicate with each other to share information and make sure they’re all working together.
3. **Combining Results**: Once the machines finish their tasks, the results are combined into a final result.

**Example:**

Let’s say you want to process a large dataset. If you did it on a single machine, it would take a long time. But in a distributed system, you can split the data into smaller chunks. Each chunk gets processed by a different machine, and once all the chunks are done, the results are combined.

Some common distributed systems technologies include:

* **Hadoop**: A framework for processing big data across many machines.
* **Spark**: A fast, distributed system for processing large datasets (PySpark is a Python interface for Spark).
* **MapReduce**: A model for processing large data across many machines.

**How PySpark Uses Distributed Systems:**

* PySpark is built on Apache Spark, which is a distributed system.
* Spark allows you to process huge datasets by breaking them up into smaller parts and processing them on multiple machines.
* You can use PySpark to write Python code that runs on a distributed system, which makes it really powerful for big data.

For example, PySpark can read data from a file and split it across many machines to process it faster. It can do operations like filtering, grouping, and joining datasets in parallel, so it’s much faster than traditional methods.

**Pytest**

**What is Pytest?**

Pytest is a Python testing framework used for writing simple to complex test cases efficiently. It is widely used because of its:  
 1.Easy syntax  
 2.Automatic test discovery  
 3.Supports fixtures and parameterized testing

Pytest

A black screen with colorful text

AI-generated content may be incorrect.

**Pytest raises:**

Sometimes, we expect a function to raise an error when given invalid input. Instead of letting the test fail unexpectedly, we can check if the error is raised on purpose using pytest.raises().

This is useful when:  
 We want to test if an exception occurs when the function gets incorrect input.  
 We are handling error cases properly in our code.

A computer screen with text on it

AI-generated content may be incorrect.

**Class based pytest:**

When we write multiple test cases, sometimes we need to group related teststogether. Instead of writing separate functions, we can use a class to organize them.

Using class-based testing in pytest helps in:  
Keeping related test cases together in a structured way.  
Making the test file more readable and well-organized.

A computer screen shot of a code

AI-generated content may be incorrect.

**Fixtures :**

A fixture in pytest is a function that runs before each test function that uses it. Instead of repeating the same setup code in every test, we use fixtures to handle setup tasks like initializing variables, database connections, or any required data.

Fixtures help make tests cleaner, reusable, and easy to manage.

**Why Use Fixtures?**

✔ Avoid Code Duplication – No need to set up the same values in every test.

✔ Better Readability – Tests focus only on logic, not setup.

✔ Reusable – The same fixture can be used across multiple test cases.

**How Fixtures Work**

A fixture function is defined using @pytest.fixture.

The function returns data or objects that will be used in tests.

In the test function, the fixture is passed as an argument.

Pytest automatically injects the returned data from the fixture into the test.

A computer screen shot of a program code

AI-generated content may be incorrect.

**Parameterized testing:**

Instead of writing multiple test cases for different inputs, **parameterized testing** helps us **run the same test function multiple times with different inputs** automatically.

This is useful because:  
It reduces duplicate code (no need to write multiple test functions).  
It makes tests more readable and organized.  
It ensures better coverage with minimal effort.

In pytest, we can use **@pytest.mark.parametrize** to pass multiple sets of values to a test function.

A computer screen with text on it

AI-generated content may be incorrect.

**Xfail and skip:**

Sometimes, we may **have tests that we don’t want to run temporarily** due to various reasons.  
For example:

The test is not relevant for now (e.g., waiting for a bug fix).

A new feature is still in development, but we already wrote a test for it.

Instead of deleting these tests, we can use Xfail or Skip in pytest:

**Xfail (@pytest.mark.xfail)**: The test will run but will be marked as expected to fail. If it fails, pytest ignores it in the final report.  
**Skip (@pytest.mark.skip)**: The test will not run at all, and pytest will skip it in the test report.

**Merge schema and overwrite schema**

**1. Merge Schema**

Merge Schema - used when reading or writing data in PySpark, especially when dealing with Parquet files. It allows you to merge the schema of a new file with the schema of existing files in a directory.

Use Case: This is useful when the schema of the data might change over time (e.g., new columns being added) and you want to make sure the new data matches the existing data schema automatically.

How it works: When you read or write data, PySpark checks the existing schema and merges it with any new schema changes.

Code: df = spark.read.option("mergeSchema", "true").parquet("path\_to\_parquet\_files")

In this example, if there are any new columns in the incoming data that weren't present in the existing data, Merge Schema will add them.

When to use: When the data schema is evolving over time (e.g., adding new columns) and you want to ensure that old and new data remain compatible.

**2. Overwrite Schema**

Overwrite Schema - that if you're writing data to a file, it will replace the existing data and overwrite the schema with the new schema.

Use Case: You might want to overwrite the data when the structure has changed and you no longer care about the old schema or data. It’s like saying, "Forget about the old data and schema, and start fresh."

How it works: When you write data to a file, if you use the overwrite mode, it will remove the existing data and replace it with new data, including a new schema if necessary.

Code: df.write.mode("overwrite").parquet("path\_to\_parquet\_files")

This will overwrite the data and schema in the specified location.

When to use: When you’re okay with deleting the old data and replacing it with fresh data and schema (e.g., when the schema has changed significantly).

**EDD:**

Event-Driven Development (EDD) is a way of building programs where the program reacts to events that happen. An event could be anything — like a user clicking a button, a message coming in, or a timer going off.

In EDD, you set up your program to listen for events and do something when an event happens.

How Does It Work?

Event Happens: Something occurs — for example, a user clicks a button or a file is uploaded.

Listen for the Event: The program is "listening" for the event to happen.

Handle the Event: When the event happens, the program responds by doing something, like showing a message or saving the data.

Example:

Imagine you are playing a game:

Event: You press a "jump" button.

Action: The character in the game jumps.

The game is reacting to the "jump" event!

Why Use EDD?

Real-Time Reactions: It allows systems to react immediately to things that happen, like a user clicking a button or sending a message.

Flexible and Efficient: The program doesn't have to check everything all the time. It only does something when an event happens.

Real-Life Examples:

Websites: When you click a button on a website, the website responds by showing a message or sending information to the server.

Chat Apps: When you receive a new message, the app reacts by showing the message on the screen.

Games: When you click "start", the game reacts by starting the game.

Why It's Cool:

Responsive: The program reacts right away to what's happening, so users don’t have to wait.

Scalable: It’s easier to grow and handle lots of things happening at once, like messages in a chat app or clicks on a website.

Event-Driven Development is all about making programs that respond to events. The program listens for things like clicks, messages, or timers, and then it does something when the event happens. It’s a way of building more interactive and efficient software.

**Pyspark - joins**

In PySpark, a join is a way to combine two datasets (DataFrames) based on a common column or key. It’s like merging two tables that share similar information so you can work with them together.

For example, imagine you have two tables:

One table has Customer Data: Customer ID, Name, etc.

Another table has Order Data: Order ID, Customer ID, and Amount.

By joining them on Customer ID, you can get a new table with both customer and order information combined.

**Types of Joins in PySpark**

**Inner Join:**

Combines rows from both tables where the values in the common column (key) match.

If there's no match, the row won’t be included in the result.

Example:

df1.join(df2, on="customer\_id", how="inner")

**Left Join (Left Outer Join):**

Keeps all rows from the left table (the first one) and the matching rows from the right table (the second one).

If there’s no match in the right table, it fills the missing columns with NULL.

Example:

df1.join(df2, on="customer\_id", how="left")

**Right Join (Right Outer Join):**

Keeps all rows from the right table and the matching rows from the left table.

If there’s no match in the left table, it fills the missing columns with NULL.

Example:

df1.join(df2, on="customer\_id", how="right")

**Full Join (Full Outer Join):**

Keeps all rows from both tables, even if there’s no match.

If there’s no match in one table, it fills the missing values with NULL.

Example:

df1.join(df2, on="customer\_id", how="outer")

**Cross Join:**

Combines every row of the first table with every row of the second table.

This can create a lot of data, so be careful when using it.

Example:

df1.join(df2, on="customer\_id", how="cross")

**When to Use Different Joins?**

Inner Join: When you only want rows that have matching data in both tables.

Left Join: When you want to keep all data from the first table, even if there’s no match in the second table.

Right Join: When you want to keep all data from the second table, even if there’s no match in the first table.

Full Join: When you want to keep all data from both tables, regardless of whether there’s a match or not.

Cross Join: When you need to combine every possible pair of rows from both tables