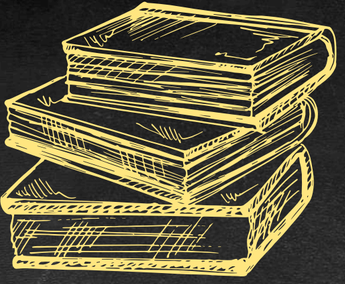
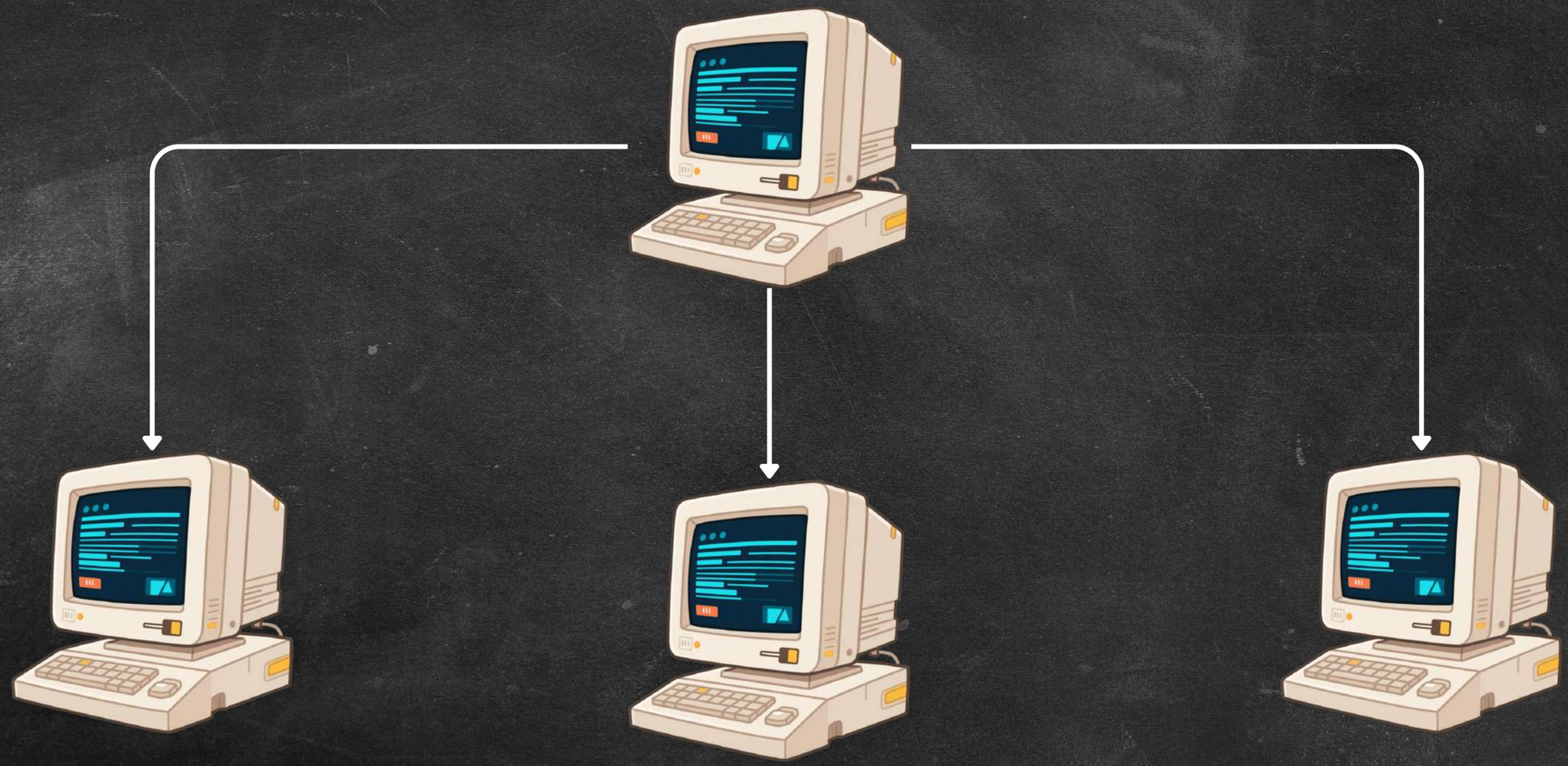
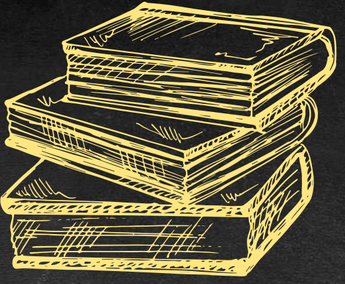

SPARK
DECLARATIVE
PIPELINES

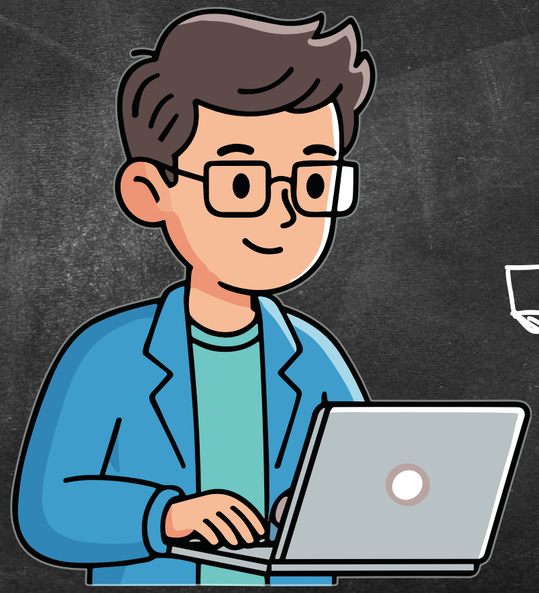


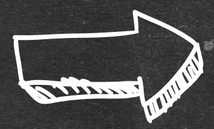
WHAT IS APACHE SPARK?

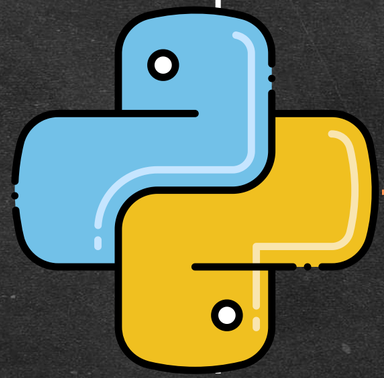


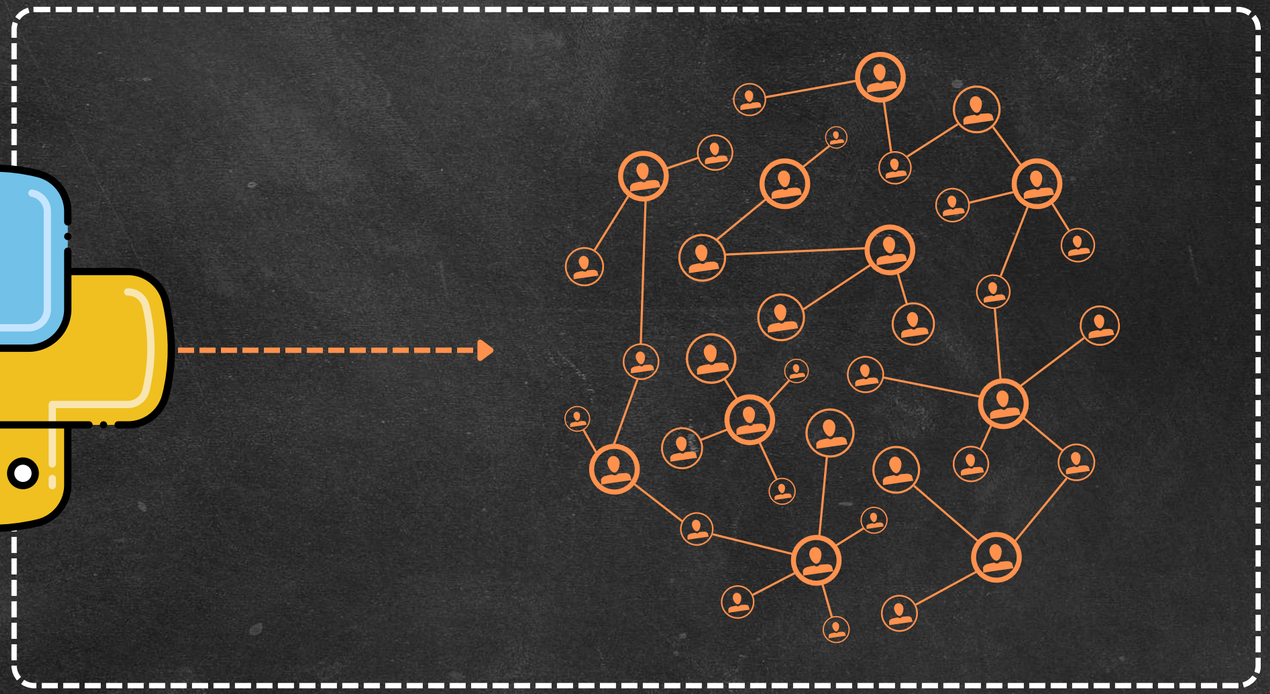


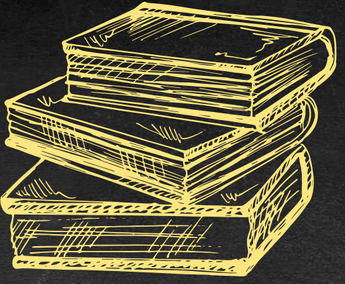
WHAT IS PYSPARK?







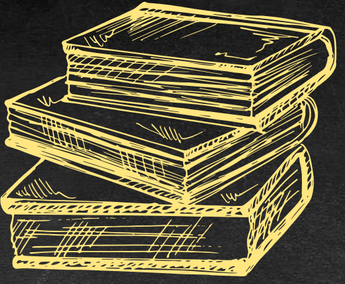




WHAT IS SDP?

When working with data in Spark, we often want to focus on WHAT we want to do with the data, rather than HOW Spark executes it under the hood. This is where declarative pipelines come in.

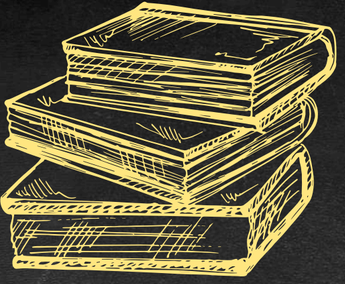
Unlike traditional imperative programming, where you give step-by-step instructions, declarative pipelines let you define a series of transformations and actions on your data in a clean, readable way. Spark then figures out the most efficient execution plan.



BUT, WHY SDP?

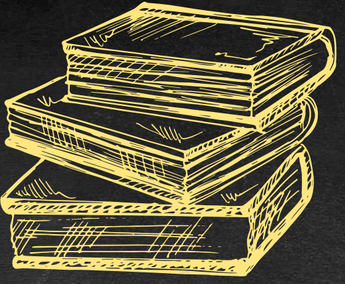
One of the biggest advantages of declarative pipelines is that they remove a lot of the traditional overhead in Spark jobs. Usually, when you write Spark code imperatively, you spend time manually creating partitions, coalescing data, caching tables, and tuning performance.

Declarative pipelines handle these optimizations automatically. You don't need to worry about the low-level mechanics - Spark figures out the most efficient way to process your data.



BUT, WHY SDP?

You don't have to worry about execution details like task scheduling or memory optimization. Instead, you describe a flow of operations - like filtering, joining, or aggregating - and Spark handles the rest. This approach reduces bugs, improves performance, and makes your data pipelines more robust.

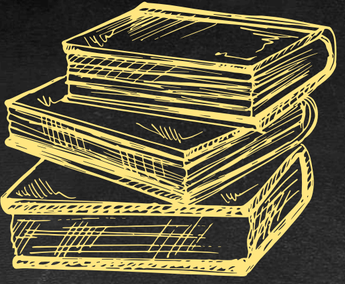


EXAMPLE

Imagine you want to process a sales dataset to find the top products by revenue. In a declarative pipeline, you'd describe your operations like:

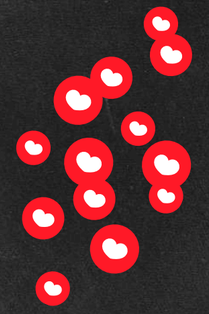
read the dataset → filter for completed orders → calculate revenue → group by product → sort → write results.

You don't tell Spark how to shuffle or partition the data—it decides the best way to execute your plan efficiently.



IT ALSO REDUCES CODE

Declarative pipelines also reduce the long, repetitive code that often comes with data engineering tasks. For example, implementing Slowly Changing Dimensions (SCD) Type 2 manually involves writing lots of steps for detecting changes, updating records, and maintaining history.



With declarative pipelines, you can leverage tools like AutoCDC, which handle these operations automatically. This lets you focus on business logic rather than boilerplate code.

KEY COMPONENTS

FLOWS

A flow is the foundational data processing concept in SDP which supports both streaming and batch semantics. A flow reads data from a source, applies user-defined processing logic, and writes the result into a target dataset.

ref - https://learn.microsoft.com/en-us/azure/databricks/ldp/concepts

KEY COMPONENTS

DATASETS

A dataset is queryable object that's the output of the flows within a pipeline.

- STREAMING TABLE
- MATERIALIZED VIEW
- TEMPORARY VIEW

ref - https://learn.microsoft.com/en-us/azure/databricks/ldp/concepts

DATASETS

STREAM TABLE

A streaming table is a form of Unity Catalog managed table that is also a streaming target for Lakeflow SDP.

MAT VIEW

A materialized view is also a form of Unity Catalog managed table and is a batch target.

TEMP VIEW

A temporary view is a View that holds the query and recomputes it.

ref - https://learn.microsoft.com/en-us/azure/databricks/ldp/concepts

KEY COMPONENTS

PIPELINES

A pipeline can contain one or more flows, streaming tables, and materialized views. It is basically a DAG of all the components.

ref - https://learn.microsoft.com/en-us/azure/databricks/ldp/concepts