# Recap

Time to sum up!

# Summary

- SubDAGs allow you to make your DAG clearer by encapsulating different logic group of tasks together.  A SubDAG is created by using a factory method that returns a DAG Object ( subDAG ) and the SubDagOperator to attach the subDAG to the main DAG.
- SubDags must be scheduled the same as their parent DAGs.
- SubDagOperator uses its own executor which by default is the Sequential Executor. You should stick to it in order to avoid possible bugs and performance degradations.
- SubDags must share the same start_date and schedule_interval than their parent.

# Summary

- A Hook is simply an interface to interact with external systems such as HIVE, PostgreSQL, Spark, SFTP and so on.
- By using a Hook you can act like you are logged into your external system.
- Some operators actually use Hooks internally such as PostgreOperator or MySqlOperator.
- Hooks use connections stored into the metadatabase created from the Connection View.

# Summary

- XCOMs stand for “cross-communication” and allow through messages stored into the metadatabase to share data between multiple tasks.
- Those messages are defined by a key, a value, a timestamp, an execution date, a task id and a dag id.
- Data are pushed by `xcom_push()` and pulled by `xcom_pull()`.
- If a task returns a value ( either from its Operator’s execute() method, or from a PythonOperator’s python_callable function), a XCOM containing that value is automatically pushed.
- By default, xcom_pull() for the keys that are automatically given to XCOMs when they are pushed by being returned from execute functions (as opposed to to XCOMs that are pushed manually).
- XCOMs are suitable for small values to share not for large sets of data

# Summary

- If we have two XCOMs with the same key value, dag id and task id, the XCOM having the most recent `execution_date` will be pulled out by default. If you didn't set an `execution_date`, this date will be equal to the `execution_date` of the DagRun.
- In both operators in which we used the functions xcom_push() and xcom_pull(), the parameter `provide_context` has been set to `True`. When `provide_context` is set to True, Airflow will pass a set of keyword arguments that can be used in your function. Those keyword arguments are passed through `**kwargs` variable. By using the 'ti' key from **kwargs, we get the TaskInstance object representing the task running the python_callable function, needed to pull or push a XCOM.

# Summary

- Branching is the mechanism allowing your DAG to choose between different paths according to the result of a specific task.
- To do this we use the BranchPythonOperator.
- The BranchPythonOperator is like the PythonOperator except that it expects a python_callable that returns a task_id. In other words, the function passed to the parameter python_callable must return the task_id corresponding to the task which will be executed next.
- The task_id returned by the Python function has to be referencing a task directly downstream from the BranchPythonOperator task.
- There is no point to use `depends_on_past=True`on downstream tasks from the BranchPythonOperator as skipped status will invariably lead to block tasks that depend on their past successes.

# What's next?

In this section, we have seen many important concepts allowing you to make your DAG way more dynamic than before. You can build very complex workflows according to different criteria which can change in time.

In the following section, we are going to see how to build plugins to enrich Apache Airflow.

Be ready!