Optimizing Pipelines with the Python API

You may have your pipelines defined in Python instead of YAML and want to optimize them. Here's an example of how to use the Python API to define, optimize, and run a document processing pipeline similar to the medical transcripts example we saw earlier.

```
from docetl.api import Pipeline, Dataset, MapOp, UnnestOp, ResolveOp,
ReduceOp, PipelineStep, PipelineOutput

# Define datasets
datasets = {
    "transcripts": Dataset(type="file", path="medical_transcripts.json"),
}

# Define operations
operations = [
    MapOp(
        name="extract_medications",
        type="map",
        optimize=True,  # This operation will be optimized
        output={"schema": {"medication": "list[str]"}},
        prompt="Analyze the transcript: {{ input.src }}\nList all medications
mentioned.",
    ),
    UnnestOp(
        name="unnest_medications",
        type="unnest",
        unnest_key="medication"
    ),
    ResolveOp(
        name="resolve_medications",
        type="resolve",
        blocking_keys=["medication"],
        optimize=True,  # This operation will be optimized
        output={"schema": {"medication": "str"}},
        comparison_prompt="Compare medications:\n1: {{ input1.medication
}}\n2: {{ input2.medication }}\nAre these the same or closely related?",
        resolution_prompt="Standardize the name for:\n{% for entry in inputs
%}\n- {{ entry.medication }}\n{% endfor %}"
    ),
    ReduceOp(
        name="summarize_prescriptions",
        type="reduce",
        reduce_key=["medication"],
        output={"schema": {"side_effects": "str", "uses": "str"}},
        prompt="Summarize side effects and uses of {{ reduce_key }} from:\n{%
for value in inputs %}\nTranscript {{ loop.index }}: {{ value.src }}\n{%
endfor %}",
        optimize=True,  # This operation will be optimized
    )
]

# Define pipeline steps
steps = [
    PipelineStep(name="medical_info_extraction", input="transcripts",
operations=["extract_medications", "unnest_medications",
"resolve_medications", "summarize_prescriptions"])
]

# Define pipeline output
output = PipelineOutput(type="file", path="medication_summaries.json")

# Create the pipeline
pipeline = Pipeline(
    name="medical_transcripts_pipeline",
    datasets=datasets,
    operations=operations,
    steps=steps,
    output=output,
    default_model="gpt-4o-mini",
    system_prompt={
        "dataset_description": "a collection of medical conversation
transcripts",
        "persona": "a healthcare analyst extracting and summarizing medication
information",
    }
)

# Optimize the pipeline
optimized_pipeline = pipeline.optimize(model="gpt-4o-mini")

# Run the optimized pipeline
result = optimized_pipeline.run()

print(f"Pipeline execution completed. Total cost: ${result:.2f}")
```

This example demonstrates how to create a pipeline that processes medical transcripts, extracts medication information, resolves similar medications, and summarizes prescription details.

Optimization

Notice that some operations have `optimize=True` set. DocETL will only optimize operations with this flag set to `True`. In this example, the `extract_medications`, `resolve_medications`, and `summarize_prescriptions` operations will be optimized.

Optimization Model

We use `pipeline.optimize(model="gpt-4o-mini")` to optimize the pipeline using the GPT-4o-mini model for the agents. This allows you to specify which model to use for optimization, which can be particularly useful when you want to balance between performance and cost.

The pipeline is optimized before execution to improve performance and accuracy. By setting `optimize=True` for specific operations, you have fine-grained control over which parts of your pipeline undergo optimization.