Pandas Integration

DocETL provides seamless integration for several operators (map, filter, merge, agg, split, gather, unnest) with pandas through a dataframe accessor. This idea was proposed by LOTUS[1].

Installation

The pandas integration is included in the main DocETL package:

```
pip install docetl
```

Overview

The pandas integration provides a `.semantic` accessor that enables:

- Semantic mapping with LLMs (`df.semantic.map()`)
- Intelligent filtering (`df.semantic.filter()`)
- Fuzzy merging of DataFrames (`df.semantic.merge()`)
- Semantic aggregation (`df.semantic.agg()`)
- Content splitting into chunks (`df.semantic.split()`)
- Contextual information gathering (`df.semantic.gather()`)
- Data structure unnesting (`df.semantic.unnest()`)
- Cost tracking and operation history

Quick Example

```
import pandas as pd
from docetl import SemanticAccessor

# Create a DataFrame
df = pd.DataFrame({
    "text": [
        "Apple released the iPhone 15 with USB-C port",
        "Microsoft's new Surface laptops feature AI capabilities",
        "Google announces Pixel 8 with enhanced camera features"
    ]
})
```

```
# Configure the semantic accessor
df.semantic.set_config(default_model="gpt-4o-mini")

# Extract structured information
result = df.semantic.map(
    prompt="Extract company and product from: {{input.text}}",
    output={
        "schema": {
            "company": "str",
            "product": "str",
            "features": "list[str]"
        }
    }
)

# Track costs
print(f"Operation cost: ${result.semantic.total_cost}")
```

Configuration

Configure the semantic accessor with your preferred settings:

```
df.semantic.set_config(
    default_model="gpt-4o-mini",  # Default LLM to use
    max_threads=64,               # Maximum concurrent threads,
    rate_limits={
        "embedding_call": [
            {"count": 1000, "per": 1, "unit": "second"}
        ],
        "llm_call": [
            {"count": 1, "per": 1, "unit": "second"},
            {"count": 10, "per": 5, "unit": "hour"}
        ]
    }
)
```

Pipeline Optimization

While individual semantic operations are optimized internally, pipelines created through the pandas `.semantic` accessor (sequences of operations like `map` → `filter` → `merge`) cannot be optimized as a whole. For pipeline-level optimizations like operation rewriting and automatic resolve operation insertion, you must use either:

- The YAML configuration interface
- The Python API

For detailed configuration options and best practices, refer to:

- DocETL Best Practices
- Pipeline Configuration
- Output Schemas
- Rate Limiting

Output Modes

DocETL supports two output modes for LLM calls:

Tools Mode (Default)

Uses function calling to ensure structured outputs:

```
result = df.semantic.map(
    prompt="Extract data from: {{input.text}}",
    output={
        "schema": {"name": "str", "age": "int"},
        "mode": "tools"  # Default mode
    }
)
```

Structured Output Mode

Uses native JSON schema validation for supported models (like GPT-4o):

```
result = df.semantic.map(
    prompt="Extract data from: {{input.text}}",
    output={
        "schema": {"name": "str", "age": "int"},
        "mode": "structured_output"  # Better JSON schema support
    }
)
```

When to Use Structured Output Mode

Use `"structured_output"` mode when: - You're using models that support native JSON schema (like GPT-4o) - You need stricter adherence to complex JSON schemas - You want potentially better performance for structured data extraction

The default `"tools"` mode works with all models and is more widely compatible.

Backward Compatibility

The old `output_schema` parameter is still supported for backward compatibility:

```
# This still works (automatically uses tools mode)
result = df.semantic.map(
    prompt="Extract data from: {{input.text}}",
    output_schema={"name": "str", "age": "int"}
)
```

Cost Tracking

All semantic operations track their LLM usage costs:

```
# Get total cost of operations
total_cost = df.semantic.total_cost

# Get operation history
history = df.semantic.history
for op in history:
    print(f"Operation: {op.op_type}")
    print(f"Modified columns: {op.output_columns}")
```

Implementation

This implementation is inspired by LOTUS, a system introduced by Patel et al. [1]. Our implementation has a few differences:

- We use DocETL's query engine to run the LLM operations. This allows us to use retries, validation, well-defined output schemas, and other features described in our documentation.
- Our aggregation operator combines the `resolve` and `reduce` operators, so you can get a fuzzy groupby.
- Our merge operator is based on our equijoin operator implementation, which optimizes LLM call usage by generating blocking rules before running the LLM. See the Equijoin Operator for more details.
- We do not implement LOTUS's `sem_extract`, `sem_topk`, `sem_sim_join`, and `sem_search` operators. However, `sem_extract` can effectively be implemented by running the `map` operator with a prompt that describes the extraction.

---

1. Patel, L., Jha, S., Asawa, P., Pan, M., Guestrin, C., & Zaharia, M. (2024). Semantic Operators: A Declarative Model for Rich, AI-based Analytics Over Text Data. arXiv preprint arXiv:2407.11418. https://arxiv.org/abs/2407.11418 ↩↩